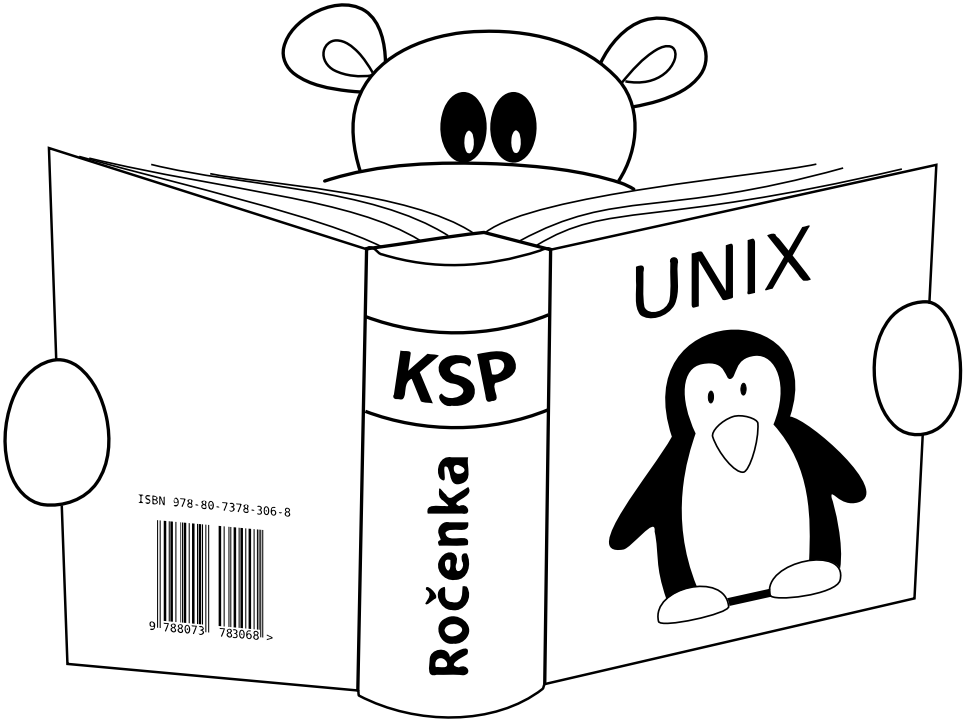
JIŘÍ SETNIČKA A KOLEKTIV

Korespondenční seminář z programování

XXVII. ročník – 2014/2015



ISBN 978-80-7378-306-8

matfyzpress

VYDAVATELSTVÍ
MATEMATICKO-FYZIKÁLNÍ FAKULTY
UNIVERZITY KARLOVY V PRAZE

JIŘÍ SETNIČKA A KOLEKTIV

Korespondenční seminář z programování

XXVII. ročník – 2014/2015

matfyzpress

Praha 2015

Vydáno pro vnitřní potřebu fakulty.
Publikace není určena k prodeji.

ISBN 978-80-7378-306-8

Úvod

Korespondenční seminář z programování (dále jen *KSP*), jehož dvacátý sedmý ročník se vám dostává do rukou, patří k nejznámějším aktivitám pro zájemce o informatiku a programování z řad studentů (nejen) středních škol. Aktivním zapojením do semináře, zejména řešením úloh, získávají studenti praxi ve zdolávání nejrůznějších algoritmických problémů, jakož i hlubší náhled na mnohé disciplíny informatiky.

Od 26. ročníku je KSP rozděleno do dvou kategorií, hlavní a začátečnické. Obě kategorie jsou rozděleny do několika *sérií*, hlavní do pěti, začátečnická do čtyř. Na začátku série pošleme řešitelům zadání sady úloh. Ty jsou různého typu, některé teoretické (úkolem je vymyslet a popsat efektivní algoritmus), některé praktické (úkolem je algoritmus nejen vymyslet, ale také naprogramovat a odladit). V hlavní kategorii bývá navíc zařazen *seriál*, který je kromě soutěžních úložek tvořen zejména povídáním o nějakém zajímavém informatickém tématu; seriál je rozložený do celého ročníku a jeho díly v jednotlivých sériích na sebe navzájem navazují.

Řešitelé pak mají několik týdnů na to, aby si úlohy rozmysleli a dali dokupy jejich řešení, které nám odevzdají. Výsledky praktických úloh mají řešitelé k dispozici hned po odevzdání, řešení teoretických úloh po termínu série opravíme, okomentujeme a pošleme zpět.

Většina věcí dnes probíhá paralelně s využitím klasické pošty a prostřednictvím Internetu. Na svých stránkách vystavujeme zadání a vzorová řešení, umožňujeme stažení opravených řešení, odpovídáme na dotazy k zadání, ...

Velkou událostí jsou dvě týdenní *soustředění*. Jarní je určené hlavně řešitelům začátečnické kategorie, ale přihlášky otvíráme i pro ty, kteří s programováním zatím nemají žádné zkušenosti a chtěli by se ho naučit. Podzimní soustředění probíhá na začátku následujícího ročníku a zveme na něj primárně nejlepší řešitele hlavní kategorie. V obou případech je pro účastníky soustředění připravený bohatý program od odborných přednášek na informatická témata až po zcela neodborné hraní a dovádění v přírodě. Navíc mají účastníci možnost potkat další lidi s podobnými zájmy.

KSP se ale v průběhu let rozrostlo, a přestože korespondenční řešení úloh a příprava soustředění zůstává jádrem naší činnosti, stíháme toho během ročníku mnohem víc.

Letos jsme přidali několik článků do naší Encyklopedie.[1] Na podzim jsme již potřetí vyrazili přednášet na střední školy v rámci Putovních přednášek.[2] V březnu jsme pak uspořádali čtvrtý ročník on-line soutěže Kasiopea.[3]

[1] `http://ksp.mff.cuni.cz/encyklopedie/`

[2] `http://ksp.mff.cuni.cz/akce/putovni-prednasky/2014/`

[3] `http://ksp.mff.cuni.cz/akce/kasiopea/2015/`

(Nejen) u úloh v této knize lze zahlédnout několik značek pro urychlení orientace:

Některé značky používáme primárně k označení typu úlohy:


V začátečnické kategorii tímto symbolem označujeme teoretické úlohy, tedy ty „klasické“. Úkolem řešitelů je vymyslet efektivní algoritmus, slovně ho popsat a tento popis odevzdat. (V hlavní kategorii jsou teoretické všechny úlohy, které nejsou přímo označené jako praktické.)


Tento symbol označuje praktickou úlohu, konkrétně takovou, které říkáme *open-data*. Úkolem řešitelů je nejen vymyslet algoritmus, ale také ho zapsat jako program a tento program odladit. Odevzdání probíhá tak, že si řešitel stáhne vstupní data a odevzdá příslušný výstup, přičemž počet pokusů není omezen.


I tato úloha je praktická, obvykle pro ni ale používáme termín *codexovka*, podle systému, do kterého se odevzdává. Řešitel napíše a odladí program, který pak nahraje do systému, kde je puštěn na neveřejných datech.


V každém ročníku KSP rozebíráme na pokračování nějaké zajímavé informatické téma do hloubky. Poslední úloha série je pokračováním takového *seriálu* – obsahuje kromě samotného zadání ještě text, ve kterém se můžete dozvědět o tématu něco nového. Jelikož díly seriálu na sebe navazují, vyplatí se mít nastudované i předchozí série.

Jiné značky slouží k označení obtížnosti a doporučených zdrojů inspirace:


Takto označenou úlohu považujeme za řešitelnou i pro začátečníky, zkušení řešitelé ji jistě zvládnou levou zadní. Pro její vyřešení by neměly být potřeba žádné speciální znalosti.


Aby si i pokročilí přišli na své, zařazujeme někdy do zadání těžkou úlohu, která se může stát leckomu noční můrou. Na její pokoření jsou často potřeba hlubší znalosti algoritmů a datových struktur, odměnou je však vyšší bodový zisk.


Protože chápeme, že k „uvaření“ řešení jsou často potřeba znalosti základních algoritmů a datových struktur, obvykle též přikládáme do každé série tzv. *kuchařku,* ze které se můžete takové věci naučit. Často je také v zadání úloha, již lze řešit algoritmem z kuchařky. A pozor – další kuchařky najdete na našich webových stránkách.


Dále tímto symbolem označujeme místa, jejichž pochopení může vyžadovat větší zamyšlení, případně nějaké předchozí znalosti.

Chcete-li se na cokoliv zeptat, ať už ohledně semináře, studia na naší fakultě nebo nějakého informatického či programátorského problému, neváhejte a napište nám na diskusní fórum na stránce *http://ksp.mff.cuni.cz/forum/* nebo na naši poštovní adresu:

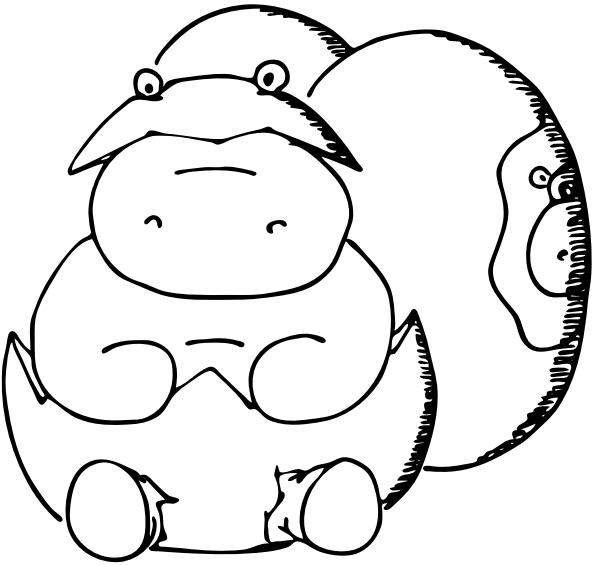
Korespondenční seminář z programování
KAM MFF UK
Malostranské náměstí 25
118 00 Praha 1

e-mail: *ksp@mff.cuni.cz*
www: *http://ksp.mff.cuni.cz/*

Obsah

KSP-Z

Začátečnická kategorie KSP



Zadání úloh KSP-Z

První série

27-Z1-1 Na zastávce — 8 bodů

Kevin jel se třídou na výlet do zoo, ale nejeli tam sami. Na autobusové zastávce stálo plno skupinek výletníků, turistů, rodin s kočárky, školních tříd, studentů a dalších cestujících, kteří tím směrem chtěli jet taky. Byla to pěkná tlačenice. Žádná skupinka se nechtěla rozdělit, aby autobusem odjela jenom její část, ani nechtěla pustit žádnou jinou skupinku před sebe. Všechny skupinky stály ve frontě v pořadí, v jakém na zastávku přišly.

Naštěstí každých deset minut přijížděl prázdný autobus. Vešlo se do něj M lidí. Když se některá skupinka už nevešla celá, nechala autobus odjet ne zcela zaplněný a čekala na další. A čekala a čekala.

Takové čekání je samozřejmě strašná nuda. Kevina by tedy zajímalo, jak dlouho ještě bude čekat. Povíte mu to?

Tvar vstupu: Na prvním řádku budou dvě čísla M a N oddělená mezerou. M udává, kolik lidí se vejde do jednoho autobusu, bude to mezi 20 a 250. N je počet skupinek, které na zastávce čekají, těch bude mezi 1 a 10^5. Následovat bude N čísel udávajících velikosti skupinek, každé na samostatném řádku. Velikosti budou v rozmezí 1 až M včetně. Kevin je v poslední skupince.

Tvar výstupu: Na výstup vypište jedno celé číslo udávající, kolikátým autobusem odjede poslední skupinka ze zastávky.

Ukázkový vstup:

```
80 5
2
40
40
2
42
```

Ukázkový výstup:

```
3
```

V každém ze tří autobusů pojede 42 lidí.

27-Z1-2 Kalkulačka — 10 bodů

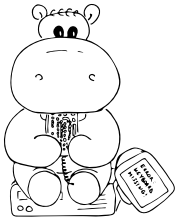
Ajéje! Stala se strašná věc. Kevin si jednou před písemkou půjčil od kamaráda Petra kalkulačku. Petr ji chce vrátit, ale Kevin ji nemůže najít. Buďto ji má někde doma, nebo ji ztratil, nebo mu ji snědl hroch, nebo...

Každopádně Petr ji musí dostat zpět. A Kevin ji nemá. Ta kalkulačka ale byla dost jednoduchá. Uměla počítat jen s celými čísly – sčítat je (`+`), odčítat (`-`), násobit (`*`) a celočíselně dělit (`/`). Neovládala závorky ani přednost násobení před sčítáním, všechny operace vyhodnocovala v pořadí, v jakém byly zadány.

Navíc pokaždé, když jste zmáčkli tlačítko s operátorem, vypsala vám dosavadní mezivýsledek.

Když se někdo pokoušel dělit nulou, nezmizela ani neshořela, jen nenápadně místo toho provedla operaci *přičtení* nuly, doufajíc, že si toho uživatel nevšimne.

Kevin se tedy rozhodl, že Petrovi kalkulačku vyrobí, aby mu ji mohl vrátit. Pomůžete mu s tím?



Tvar vstupu: Na jednom jediném řádku vstupu budete mít korektní výraz k vyhodnocení. Ten je tvořen nezápornými celými čísly oddělenými operátory `+`, `-`, `*`, `/`. Mezi každou dvojicí sousedních čísel je právě jeden operátor. Navíc jsou čísla oddělena od operátorů mezerami.

Vstupní čísla budou z intervalu $\langle 0, 10^9 \rangle$, všechny mezivýsledky se vejdou do $\langle -10^9, 10^9 \rangle$. Vstup má délku nejvýše 500 znaků, je ukončen `=`.

Tvar výstupu: Na i-tý řádek výstupu vypište to, co bude na displeji kalkulačky po zmáčnutí i-tého operátoru (jinými slovy hodnotu části výrazu od začátku po i-tý operátor). Na posledním řádku výstupu nechť je finální výsledek, který se objeví po stisku `=` (hodnota celého výrazu).

Všechny operace ve výrazu vyhodnocujte zleva doprava bez ohledu na priority operátorů. Při dělení zaokrouhlujte záporná čísla směrem k nule (tedy $-5/3 = -1$). Kdykoli se ve výrazu objeví sekvence „`/ 0`“, interpretujte ji jako „`+ 0`“.

Ukázkový vstup:

```
2 * 0 + 333 / 50 / 0 - 5 =
```

Ukázkový výstup:

```
2
0
333
6
6
1
```

27-Z1-3 Slovník T9 — 10 bodů

„Píp píp!“ Ne, nejsme v zoo v pavilonu ptáků, jsme u hrochů. Kevinovi právě přišla esemeska od spolužačky Sáry. Někde se v tom stádu ztratila.

„`Leti led si? Para`“. Co to je? Takové zmatené zprávy občas posílá Kevinova babička. A Sára.

Sára totiž používá slovník T9. Když píše esemesky, nemusí jednu klávesu zmáčknout třeba čtyřikrát, když chce napsat S. Stačí za každé písmeno stisknout příslušnou klávesu jen jednou, slovník z kombinace stisknutých kláves pozná, co chtěla napsat, a to slovo napíše.

Pro připomenutí, klávesnice většiny starších mobilních telefonů vypadala nějak takto:

1	**2** ABC	**3** DEF
4 GHI	**5** JKL	**6** MNO
7 PQRS	**8** TUV	**9** WXYZ

Ne vždy ale T9 pozná přesně to, co Sára myslela. Více slov totiž může mít stejnou kombinaci kláves. Občas se stane, že slovník napoví špatně. Třeba když namačkáte tlačítka pro „`Kevi kde si? Sara`“, může se objevit i to, co teď Sára poslala Kevinovi.

Kevina by zajímalo, která slova jsou v tomto ohledu nejnebezpečnější, tedy je může T9 zaměnit s co nejvíce jinými slovy. A to už je úkol pro vás.

Tvar vstupu: Na prvním řádku dostanete číslo $1 \leq N \leq 10^5$ udávající velikost slovníku. Na dalších N řádcích bude vždy jedno slovo složené z písmen anglické abecedy, každé slovo bude dlouhé 1 až 15 znaků.

Tvar výstupu: Vypište největší skupinu slov, která mají všechna v T9 stejný zápis. Každé ať je na samostatném řádku. Pokud takových skupin existuje více, můžete si vybrat libovolnou.

Ukázkový vstup:

```
6
punc
jana
lama
runa
suma
puma
```

Ukázkový výstup:

```
puma
runa
suma
punc
```

27-Z1-4 Lyžař — 12 bodů

Je léto, sluníčko svítí, ptáčci zpívají, po sněhu ani památky... Tak proč si nezalyžovat? Máme přece travní lyže!

Kevin je právě na horách a lyžuje. Stojí na vrcholu sjezdovky před svou poslední jízdou, za chvíli půjde na chatu opékat párky. Sjezdovku už dobře zná, každé místo projel snad stokrát a pamatuje si, jak se mu které líbí. Všechna jsou podle toho ohodnocená celými čísly.

```
   1
  2 3
 4 1 1
2 5 0 2
```

Kevin by tedy chtěl, aby součet ohodnocení míst, přes která při poslední jízdě projede, byl co největší.

Na travních lyžích se dá jezdit z kopce a přitom zatáčet doleva nebo doprava. Z každé pozice na svahu se tak dá dostat na nejvýše dvě další pozice ležící pod ní.

Tvar vstupu: Pro jednoduchost bude kopec zadán po řádcích. Na prvním řádku bude číslo N udávající výšku kopce. Na dalších N řádcích bude vždy na i-tém řádku i celých čísel udávajících ohodnocení míst na kopci na i-té hladině od vrcholu, čísla budou oddělena mezerami.

Jízda doleva je v našem zápisu svahu ekvivalentní sestupu na další řádek a jízda doprava sestupu na další řádek a posunutí o pozici doprava.

Tvar výstupu: Na výstup vypište jedno celé číslo udávající maximální součet ohodnocení poslední Kevinovy jízdy z prvního řádku na libovolné místo na spodním řádku.

Ukázkový vstup:

```
4
1
2 3
4 1 1
2 5 0 2
```

Ukázkový výstup:

```
12
```

Optimální cesta je pořád doleva (v našem zápisu rovně dolů), jen mezi předposledním a posledním řádkem zatočíme doprava. Její ohodnocení je skutečně $1 + 2 + 4 + 5 = 12$.

27-Z1-5 Céděčko z koncertu — 12 bodů

Kevin hraje na bicí ve studentské rockové kapele *Velká tlama*. V létě měli velký koncert, který nahrávali. Teď by chtěli začít fanouškům prodávat céděčko se záznamem tohoto koncertu. Bohužel, vydat můžou jenom jedno a celý koncert se na něj nevejde, takže tam dají jenom jeden souvislý úsek. (Vystříhat jen něco nemohou, to by nedávalo posluchačům smysl.)

Koncert je pro naše účely posloupnost písniček, každá je určená svojí délkou. Jinak jsou nerozlišitelné.

Kapela by chtěla vybrat jednu souvislou část koncertu, a to tak, aby součet délek písniček přesně naplnil kapacitu CD. Fanoušci jsou totiž nároční a jedině takto budou spokojeni.

Které písničky má kapela vybrat?

Jinými slovy, na vstupu dostanete posloupnost celých čísel a číslo K. Navrhněte co nejefektivnější algoritmus, který nalezne v posloupnosti souvislý úsek se součtem právě K, nebo ohlásí, že v ní žádný takový není. Snažte se co nejlépe zdůvodnit, proč je váš algoritmus správný a efektivní.

KSP-Z

zadání

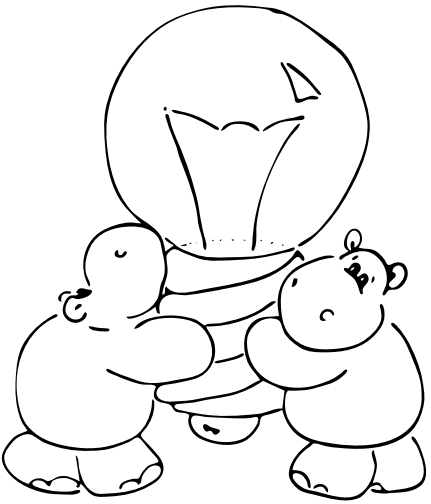
27-Z1-6 Žárovky — 14 bodů

Na chatě Kevin s tatínkem spravuje elektriku, tedy vlastně ji spíš dělají celou znova. Teď zrovna světla v kuchyni.

V kuchyni mají N žárovek. Chtěli by k nim připojit vypínače tak, aby jejich nastavením mohli rozsvítit libovolný počet žárovek z rozsahu 0 až N. Jeden vypínač může rozsvěcet více žárovek.

Navrhněte pro ně řešení, které bude potřebovat co nejméně vypínačů pro zadané N. Pro plný počet bodů také dokažte, že menší počet jim stačit nebude.

Například pro $N = 5$ potřebujeme tři vypínače: první rozsvěcí žárovku 1, druhý žárovky 2 a 3 a třetí ovládá 4 a 5. Snadno si rozmyslíte, že dva nestačí.



Druhá série

27-Z2-1 Závorky z cereálií — 8 bodů

„Vyřešte hlavolam a pojeďte na exkurzi k německé dálniční policii!“ hlásal slogan na obalu cereálií, které si Kevin koupil k snídani. Po zatáhnutí za papírek na něj z krabice vypadla dlouhá rulička papíru potištěná spoustou různých závorek.

Úkolem bylo odhadnout, kolik nejméně závorek je potřeba přidat, aby výraz byl správně uzávorkovaný (aby se závorky nekřížily a všechny byly správně spárované). Například výraz `()()` správně uzávorkovaný je, ale `)(` ani `))((` nejsou. „Jaké odhadování?“ řekl si Kevin, „já to spočítám přesně!“

Tvar vstupu: Na vstupu dostanete na jediném řádku posloupnost otevíracích `(` a zavíracích `)` závorek dlouhou maximálně 100 000 znaků.

Tvar výstupu: Na výstup vypište jedno celé číslo udávající, kolik nejméně závorek je potřeba přidat, aby byl výraz správně uzávorkovaný.

Ukázkový vstup:	*Ukázkový výstup:*
`)())`	2
`(())()`	0

27-Z2-2 Hrnce od Horsta — 10 bodů

Kevin se Sárou jedou do Německa! Sice si Kevin výhru představoval jinak než jako cestu autobusem plným důchodců, ale aspoň něco. Jejich autobus se teď dokonce zastavil v nějakém nákupním centru a dav se vyhrnul ven s voláním: „Poběžte, uvidíme živého Horsta, Horsta Fuchse!“

Kevin se Sárou se vydali za davem, snadno předešli běžící důchodce a stanuli před pultem plným hrnců s podivným chlapíkem za ním. „Nakupte sadu našich Ultra hrnců! Když si ji koupíte do tří minut, nedostanete jednu sadu navíc, ani dvě sady navíc. Ne, dostanete tři sady navíc a k tomu magnetku na lednici!“

Pamatuje na to, že slíbil dovézt z Německa babičce nějaký dárek, Kevin jednu sadu koupil, a byl obtěžkán několika dalšími. Teď řeší problém, jak poskládat hrnce do sebe, aby v autobuse zabraly co nejméně místa.

Každý hrnec má svůj průměr v centimetrech a dá se do něj vložit jakýkoliv hrnec s menším průměrem, do něhož se opět dá vložit další ještě menší hrnec a tak dál. Kevina by zajímalo, do kolika nejméně „komínků“ může hrnce seskládat.

Tvar vstupu: Na prvním řádku vstupu dostanete číslo N udávající počet hrnců. Na druhém řádku poté bude N přirozených čísel udávajících průměry hrnců. Bude platit, že $N \leq 50\,000$ a průměry hrnců budou mezi 1 a 700 000.

Tvar výstupu: Na první řádek výstupu vypište číslo K udávající minimální počet komínků, do kterých se dají hrnce seskládat. Na dalších K řádcích pak vypište hrnce v jednotlivých komínících (v pořadí od největšího hrnce). Pokud existuje více možností poskládání, vypište libovolné z nich.

Ukázkový vstup:

```
6
1 6 7 1 6 3
```

Ukázkový výstup:

```
2
7 6 3 1
6 1
```

27-Z2-3 Nápis na tričku — 10 bodů

Kevinova exkurze se konečně dostala do místa, kam se těšil nejvíce – na stanici německé dálniční policie. Nebyl by to ale Kevin, aby se nezatoulal, kam neměl. Ani nevěděl jak, ale ocitl se na průhledné straně poloprůhledného zrcadla a stal se svědkem výslechu.

Policista se zrovna vyptával na nápis na firemním tričku podezřelého. Ale jediné, co dokázal ze svědkyně dostat, bylo: „No víte… já si ten nápis nepamatuji, ale kdyby se k němu přidalo pár písmenek a tahle se přeházela, dalo by to slovo pampeliška, pane policisto. Přísahám vám, já luštím křížovky!“

Policista bezradně rozhodil rukama, vyšel z místnosti na chodbu… a tam potkal Kevina. Kevin tu rozhodně neměl co dělat. Než se na něj ale stihl policista obořit, rozhodl se Kevin rychle improvizovat. Nenapadlo ho však říci nic lepšího než: „Můžu vám pomoci najít ten název firmy.“

Tím si to ale zavařil, protože byl posazen k počítači s databází všech německých firem a teď potřebuje rychle spočítat, kolik různých názvů firem se dá poskládat z vybraných písmen zadaného slova.

Tvar vstupu: Na prvním řádku vstupu dostanete jediné slovo S. Na druhém řádku bude číslo N udávající počet slov ve slovníku a na dalších N řádcích naleznete slova slovníku (na každém řádku jedno). Slov ve slovníku bude maximálně 30 000, všechna slova budou dlouhá nanejvýš 100 znaků a všechna budou tvořena z malých písmen anglické abecedy („ch“ chápeme jako dvě písmena).

Tvar výstupu: Na výstup vypište ta slova ze slovníku, která se dají poskládat z vybraných písmen slova S, na každý řádek právě jedno. Slova vypisujte v pořadí, v jakém se objevila ve vstupním souboru.

Ukázkový vstup:

```
pampeliska
4
liska
mapa
zeli
kapka
```

Ukázkový výstup:

```
liska
mapa
```

27-Z2-4 Hořící auto — 12 bodů

„Proč já?“ problesklo Kevinovi hlavou. Jako poděkování za pomoc vzal Semir jeho a Sáru na projížďku po německé dálnici. Ale kdo měl vědět, že po nich bude někdo střílet!

Teď mají poškozené řízení, díru v nádrži, vytéká jim benzín a z vysílačky se ozývá jen: „Kobro 11, jestli zničíte ještě jedno auto, dostanete příště šlapací tříkolku!“ Aby toho nebylo málo, ta vytékající palivová čára za nimi právě vzplanula.

Auto je poškozené tak, že může zatáčet jen doprava, a to vždy jen o pravý úhel. Navíc těsně za ním hoří čára vytékajícího benzínu, a pokud auto znovu překříží hořící čáru, vybuchne. Autonavigace Semirovi vnucuje nějaký plán cesty a Kevin by chtěl vědět, jak moc je bezpečná.

Tvar vstupu: Na prvním řádku vstupu bude číslo N a na druhém pak řada N celých kladných čísel. Znamenají, že auto ujede a_1 metrů rovně, zabočí doprava, ujede a_2 metrů rovně, zabočí doprava, ujede a_3 metrů rovně, zabočí doprava a tak dále. Bude platit, že $N \leq 2\,000\,000$, a délky jednotlivých úseků budou nanejvýš 10^9 metrů.

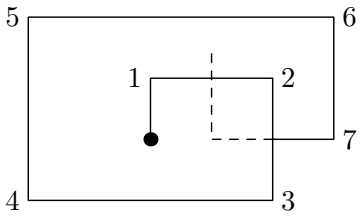
Tvar výstupu: Na jediný řádek výstupu uveďte, po kolika zatáčkách auto překříží hořící dráhu (za překřížení počítáme i dotyk), zatáčky počítejte od jedničky. Pokud k překřížení nedojde, vypište na výstup nulu.

Ukázkový vstup:

```
9
1 2 2 4 3 5 2 2 2
```

Ukázkový výstup:

```
7
```



27-Z2-5 Hledání stromů — 12 bodů

Kevinovi se z drsné jízdy autem z minulé úlohy udělalo trochu špatně, a tak se po návratu na policejní stanici posadil do nejbližšího křesla a pro uklidnění si začal na papír kreslit nějaké body a čáry mezi nimi. Dalo by se říci, že si kreslil na papír *grafy*.

Sára se mezitím někam zatoulala, a tak jí Kevin jeden obzvláště pěkný graf poslal SMSkou (třeba jako seznam sousedů, tedy pro každý bod seznam,

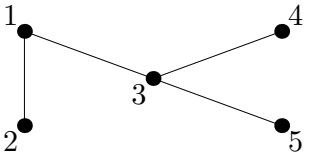
se kterými jinými body je tento bod spojený čárou). Sáru by teď zajímalo, jestli je graf, který dostala, *stromem* – tak se říká grafům, které jsou souvislé (z každého vrcholu se dá dostat do každého jiného) a neobsahují žádný cyklus.

Zkuste pečlivě popsat postup, jak to algoritmicky zjistit. Graf si můžete představit zadaný *seznamem sousedů* jako na ukázce níže. Pokud budete chtít, můžete si i rozmyslet, jestli se postup nějak změní, pokud graf dostaneme jako *matici sousednosti* (ale na plný počet bodů stačí zamyslet se jen nad verzí se seznamem sousedů).

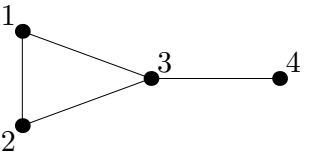
Pokud tápete v pojmech grafů, stromů a jejich reprezentace, podívejte se do naší kuchařky o základních algoritmech.[4]

Příklad: Pro grafy níže: levý graf je stromem, pravý ale není, jelikož obsahuje cyklus $1, 2, 3$.

```
1: 2,3
2: 1
3: 1,4,5
4: 3
5: 3
```




```
1: 2,3
2: 1,3
3: 1,2,4
4: 3
```



27-Z2-6 Povrch dálnice — 14 bodů

Nadešel čas, kdy se měli Kevin se Sárou zase vrátit z Německa domů. Semir věnoval Kevinovi na památku zachráněný volant z jeho služebního auta a ještě naposled je vzal na procházku na místo, kde se opravovala dálnice po jejich zběsilé jízdě z předchozích úloh.

Povrch byl pokrytý obroušenými zbytky gumy, olejem a spálenou klikatící se čárou. Jeden z policistů obcházel místo s kolečkem na měření vzdáleností a počítal, kolik metrů čtverečních dálnice bude potřeba vyčistit a opravit.

Policista chodí pravoúhle vždy o celé metry a jeho kroky by se daly popsat třeba pomocí světových stran jako posloupnost 5×jih, 2×západ, 3×sever, atd. Navíc nikdy nepřekříží svoji dřívější cestu a nakonec se vrátí zpátky do místa, odkud vyšel. Obejde tedy obvod nějakého uzavřeného obrazce.

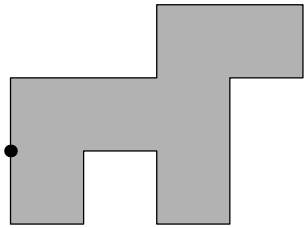
Kevina by teď zajímalo, jak velký kus dálnice to vlastně je, tedy jaký má obsah.

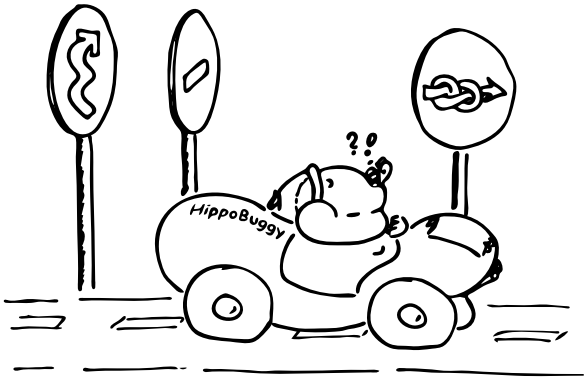
[4] `http://ksp.mff.cuni.cz/viz/kucharky/zakladni-algoritmy`

Zadání úloh KSP-Z – 2. série

Příklad: Následující obrazec mohl vzniknout tím, že policista vyrazil po níže uvedené trase. Obsah obrazce je $7\,\mathrm{m}^2$.

```
1S,2V,1S,2V,1J,1Z,2J,1Z,1S,1Z,1J,1Z,1S
```





Třetí série

27-Z3-1 Kevin nabíječ, s.r.o. — 8 bodů

Většina Kevinových spolužáků velice aktivně používá smartphone. Možná ale až moc, protože se jim pořád vybíjí. Kevin je na rozdíl od nich podnikavý typ, a tak dostal nápad. Začne podnikat s jejich nabíjením. Ve třídě však mají jen jednu zásuvku, tak Kevin skočil do elektra a koupil prodlužovačky s různými počty výstupů. Nyní by jej zajímalo, kolik nejvíce smartphonů zároveň může nabíjet, pokud prodlužovačky zapojí optimálně.

Tvar vstupu: Na prvním řádku vstupu dostanete číslo N ($1 \leq N \leq 1\,000\,000$) udávající počet Kevinových prodlužovaček. Na druhém řádku bude N čísel oddělených mezerou udávající počty zdířek jednotlivých prodlužovaček. Všechny prodlužovačky budou mít nejméně nula a nejvíce sto zdířek.

Tvar výstupu: Na výstup vypište jediné číslo udávající nejvyšší možný počet volných zdířek, kterých Kevin může optimálním zapojením prodlužovaček dosáhnout.

Ukázkový vstup:

```
2
3 5
```

Ukázkový výstup:

```
7
```

27-Z3-2 Nedej vitagen — 10 bodů

Kevinova sestra Zuzka dostala od Ježíška písmenkovou skládačku, pomocí které se učí číst slova popředu… a taky pozpátku. Z písmenek poskládala několik slov a všimla si, že některá se čtou popředu stejně jako jiná pozpátku. Hned se běžela pochlubit Kevinovi a zeptala se jej, které nejdelší slovo vypadá stejně jako jiné pozpátku.

Tvar vstupu: Na prvním řádku vstupu dostanete číslo N, počet slov, které Zuzka poskládala. Na dalších N řádcích bude jedno slovo složené z malých písmen anglické abecedy o maximálně 100 znacích. Slov bude maximálně 100 000.

Tvar výstupu: Na výstup vypište nejdelší slovo, které má mezi slovy na vstupu i svou verzi napsanou pozpátku. Pokud takových slov existuje více, vypište to, které je lexikograficky nejmenší (tj. ve slovníku by bylo napsané jako první).

Ukázkový vstup:

```
5
kecup
ves
vrabec
pucek
sev
```

Ukázkový výstup:

```
kecup
```

27-Z3-3 Superstromy — 10 bodů

Kevinův kamarád Petr bude pořádat párty ve své nové zahradě. Ještě předtím by v ní ale chtěl vysadit N rychlorostoucích superstromů, přičemž o každém přesně ví, kolik dní po zasazení doroste. Se sázením je docela práce, a tak každý den zvládne zasadit právě jeden superstrom. Párty bude uspořádána právě den poté, co doroste poslední ze superstromů. Petr se teď potřebuje s Kevinem poradit a zjistit, kdy nejdříve párty může uspořádat.

Tvar vstupu: Na prvním řádku vstupu bude číslo N udávající celkový počet superstromů ($1 \leq N \leq 1\,000\,000$). Na druhém řádku bude N čísel oddělených právě jednou mezerou udávající doby růstu jednotlivých superstromů ve dnech. Tyto hodnoty budou v rozsahu 1 až 1 000.

Tvar výstupu: Na výstup vypište číslo dne, na kdy Petr může naplánovat párty, pokud superstromy bude sázet v optimálním pořadí. První den, kdy Petr sází, má číslo 1.

Ukázkový vstup:	*Ukázkový výstup:*
`4` `3 6 7 2`	`9`

27-Z3-4 Robo Rally — 12 bodů

Kevin dostal k Vánocům deskovou hru Robo Rally. Zjednodušeně to je hra, kde se na herní ploše o rozměrech $W \times H$ pohybuje N robotů. Na začátku každý robot začíná na své pozici a je natočený jedním ze čtyř směrů. Pak roboti střídavě plní příkazy, v jeden moment plní příkaz vždy pouze jeden robot. Příkazy mohou být tří typů: „otoč se doleva“, „otoč se doprava“ a „popojdi rovně o jedno políčko“.

Příkazy jsou předem určené. Každý příkaz v sobě obsahuje číslo robota, pro kterého je určen, akci, kterou má robot vykonat, a počet opakování pro danou akci. Například zápis `3 K 5` znamená, že robot číslo 3 udělá 5 kroků vpřed, `2 L 1`, že robot 2 se jednou otočí o 90 ° doleva a `4 P 3`, že robot 4 se třikrát otočí o 90 ° doprava.

To ale není všechno. Roboti občas do sebe můžou narazit nebo dojet na kraj herní plochy. Pokud robot narazí na kraj, zůstane na stejném políčku. Pokud narazí do jiného robota, tak jej posune o jedno políčko směrem, kterým jede. Stejně tak pokud narazí do řady robotů, tak daným směrem posune celou řadu. Pokud je ale na konci řady okraj desky, celá řada se zastaví.

Kevin poskládal na desku roboty do počáteční pozice a vytáhl pro ně M kartiček s příkazy. Zajímalo by ho, jak pozice na desce bude vypadat po vyplnění všech M příkazů v pořadí, v jakém je vytáhl. Pomůžete mu to zjistit?

Tvar vstupu: Na prvním řádku vstupu budou čtyři čísla: W, H, N, M udávající postupně šířku a výšku desky, počet umístěných robotů a počet příkazů. ($1 \leq W, H \leq 100$, $1 \leq N, M \leq 2500$ a $N \leq W \cdot H$)

Dalších N řádků udává pozice jednotlivých robotů, každý je ve tvaru $X\ Y\ S$, kde $0 \leq X \leq W-1$, $0 \leq Y \leq H-1$ a S je jeden ze znaků `S, V, J, Z`, tj. světová strana, na kterou je robot otočený. Souřadnici [0,0] považujeme za severozápadní roh hrací desky. Pozice robotů jsou navzájem různé.

Na dalších M řádcích jsou definovány příkazy. Příkaz je tvaru $R\ P\ I$, kde R představuje číslo robota ($0 \leq R \leq N-1$), P znak příkazu (`L` pro rotaci doleva, `P` pro rotaci doprava a `K` pro pohyb dopředu) a I počet opakování ($1 \leq I \leq 1\,000\,000\,000$).

Tvar výstupu: Na výstup vypište N řádek udávající koncové pozice robotů. Na i-tém z nich vypište x-ovou a y-ovou souřadnici a natočení robota číslo i a to ve stejném formátu jako na vstupu.

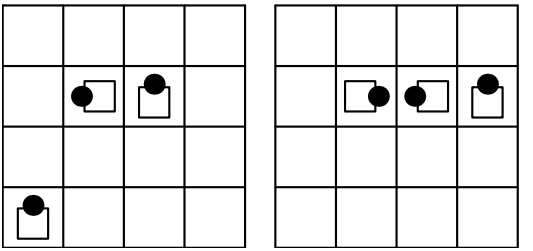
Ukázkový vstup:

```
4 4 3 3
0 3 S
1 1 Z
2 1 S
0 K 2
0 P 1
0 K 10
```

Ukázkový výstup:

```
1 1 V
2 1 Z
3 1 S
```

Hrací deska na začátku a na konci hry vypadá následovně:



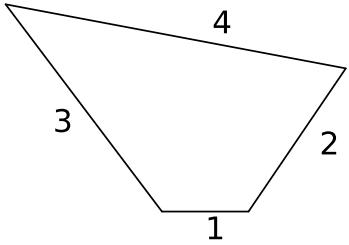
27-Z3-5 Dřevěná slacklajna — 12 bodů

Kevin a Sára rádi chodí po slacklajně, jen jim to zatím moc nejde. A tak se rozhodli, že si postaví vlastní, jednodušší, ze dřeva. K dispozici mají n prken o délkách $d_1, \ldots, d_n$. Dvě prkna k sobě můžou přidělat (spojit za konce) pod libovolným úhlem. Z prken by si chtěli postavit co nejdelší okruh (myšleno mnohoúhelník), po kterém by mohli chodit kolem dokola.

Navrhněte pro ně co nejefektivnější algoritmus, který pro zadané délky prken spočítá největší obvod okruhu, který je možné z prken postavit. Okruh může být jak konvexní, tak nekonvexní, ale nesmí sám sebe protínat. Není nutné použít všechna prkna. Nezapomeňte pořádně zdůvodnit, proč algoritmus funguje a proč počítá tak, jak počítá.

Všechna prkna mají stejnou, zanedbatelnou šířku, tedy s nimi můžete pracovat jako s úsečkami.

Například pro prkna délek $15, 1, 2, 3, 4$ je maximální obvod $1+2+3+4 = 10$. Zbylé prkno použít nelze. Výsledný okruh může vypadat například takto:



27-Z3-6 Red Bull dává křídla — 14 bodů

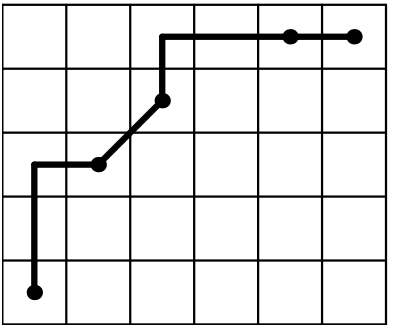
„Už nikdy nebudu pít!“ pronesl Kevin po tom, co se na Nový rok probudil u sebe v posteli. „Aspoň ne Red Bull! … Bože, to byla zas cesta domů!“

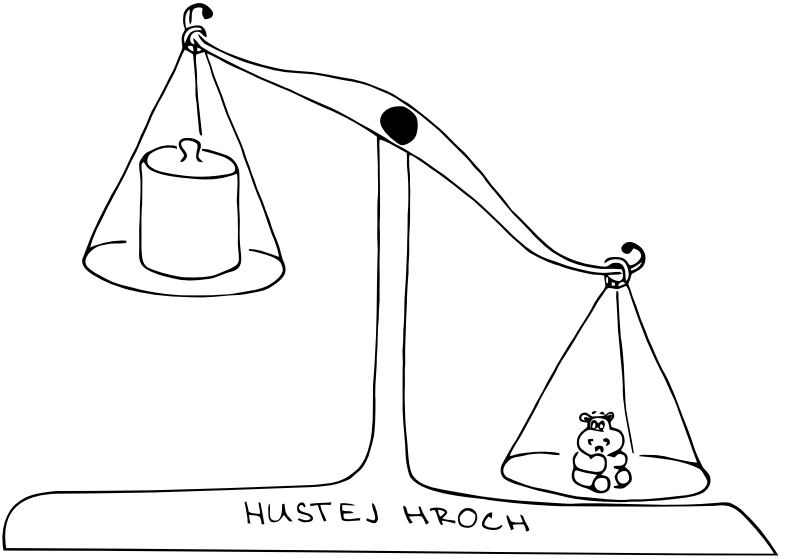
To byl takhle Kevin na silvestrovské párty u Petra a na závěr, aby měl dost síly dojít domů, si dal čtyři Red Bully. A šlo se. A taky skákalo. Vlastně každý druhý krok byl skok.

Přesněji šel Kevin po takové čtvercové síti velké $W \times H$ ze startu na políčku $[x_s, y_s]$ do cíle na políčku $[x_c, y_c]$. Každý svůj lichý krok skákal, tj. hýbal se jako šachový kůň a každý svůj sudý krok normálně šel, tj. hýbal se jako šachový král. Navrhněte algoritmus, který Kevinovi najde nejkratší cestu ze startu do cíle, pokud cestuje za těchto podmínek.

Jen připomeneme, že skok šachového koně je vždy posunem o dvě políčka v jedné souřadnici a o jedno políčko v souřadnici druhé. Šachový král se může pohnout na jedno z osmi sousedních políček.

Například optimální cesta mezi protějšími rohy šachovnice 6×5 má 4 kroky a vypadá takto:





Čtvrtá série

27-Z4-1 Záhada Pražského orloje — 8 bodů

Sára, Kevinova dobrá kamarádka, má velký zájem v kultuře, zvlášť pak v architektuře. Kevin se jí rozhodl udělat radost a domluvil se svým známým prohlídku Pražského orloje zevnitř. Sára z prohlídky byla celá nadšená a do detailů se na všechno vyptávala, zatímco Kevina tyto řeči moc nebraly a více jej zaujala ozubená kolečka, která byla všude kolem. Zvlášť dvě do sebe zasazená, která se pravidelně každou vteřinu pootočila. Jedno mělo dohromady A zubů a druhé B zubů.

Jeden zub na prvním kolečku byl označen křížkem a stejně tak jedna zdířka druhého kolečka. Kevin si všiml, že tyto křížky se pravidelně setkávají. Zajímalo by ho, za jak dlouho se to stane.

Tvar vstupu: Na prvním řádku vstupu dostanete číslo N udávající počet dotazů. Na každém z dalších N řádků se pak nacházejí čísla A a B udávající velikosti koleček. $1 \leq A, B \leq 1\,000\,000$ a $N \leq 5\,000$.

Tvar výstupu: Na výstup vypište pro každý dotaz jedno číslo udávající dobu, po které se obě značky na kolečkách setkávají.

Ukázkový vstup:

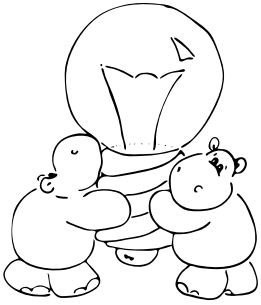
```
3
3 6
12 27
4 17
```

Ukázkový výstup:

```
6
108
68
```

27-Z4-2 Unavení u oken — 10 bodů

Když se Kevin jednoho dne vracel z florbalového tréninku, byl hodně, ale opravdu hodně zničený. Tentokrát už nevydržel jít dál a musel si na sídlišti sednout na lavičku. Jak tam tak unavený seděl, zahleděl se do svítících oken paneláku naproti. Ten byl H pater vysoký a v každém měl W oken.



Seděl, civěl a skoro usínal, když v tom se najednou z vedlejší lavičky ozval Petrův hlas: „Taky počítáš svítící okna? Já se tím po tréninku vždycky nejvíc odreaguju!“ „Jo?“ odpovídá vylekaně Kevin. „Kolik je jich asi nejvíce spojených dohromady?“ pokračuje zaujatě Petr.

Tvar vstupu: Na vstupu na prvním řádku dostanete dvě čísla oddělená mezerou $1 \leq W, H \leq 1\,000$. Na dalších H řádcích pak bude vždy W znaků '`.`' nebo '`#`', kde '`.`' značí zhasnuté okno a '`#`' značí rozsvícené okno.

Tvar výstupu: Na výstup vypište jedno číslo udávající velikost největší souvislé svítící plochy (v počtu svítících oken). Dvě svítící okna považujeme za sousední, pokud jsou buďto přímo nad sebou, nebo přímo vedle sebe. Za spojená nepovažujeme okna sousedící pouze přes roh.

Ukázkový vstup:

```
6 3
#.#.##
##.#.#
##.###
```

Ukázkový výstup:

```
7
```

27-Z4-3 Běžkaři v Praze! — 10 bodů

A je to tady! Do Prahy dorazilo 20 kamiónů se sněhem a můžou se tam uspořádat běžkařské závody. Sněhu ale nebylo dost, a tak se organizátorům povedlo vyrobit jen jeden okruh dlouhý S kilometrů s jednou dráhou. No jo, jak ale teď změřit čas všem K závodníkům? Když jeden druhého dožene, tak už jej pak nemá jak předjet, a tedy jeho naměřený čas bude špatný.

Organizátoři vymysleli následující systém. Fixně určí pořadí závodníků a budou je pravidelně po 1 minutě pouštět na trať a v cíli měřit čas. Pokud závodník dojede v těsném závěsu za jiným, tak se jeho čas počítat nebude a pojede pak znova. Jinými slovy vyškrtnou ze startovního listiny všechny závodníky, u kterých již naměřili správný čas a zbytek si závod zopakuje.

Kevin zná všechny závodníky a pro každého závodníka i ví, jakou stálou rychlostí v_i jezdí. Při znalosti těchto hodnot je možné spočítat, kolik závodů dohromady bude muset proběhnout. Kolik?

Tvar vstupu: Na vstupu na prvním řádku dostanete počet závodníků K a délku tratě S v kilometrech. Na druhém řádku budou čísla $v_1, \ldots, v_K$ udávající rychlosti závodníků v kilometrech za hodinu v pořadí, v jakém za sebou startují.

$1 \leq K \leq 10\,000$, $1 \leq S \leq 1\,000$, $1 \leq v_i \leq 100$.

Tvar výstupu: Na výstup vypište jediné číslo udávající, kolikrát musí závod proběhnout, aby byl všem závodníkům čas změřen správně.

Ukázkový vstup:

```
4 10
2 3 5 4
```

Ukázkový výstup:

```
3
```

Poznámka: Na většinu vstupů by měla stačit přesnost na sekundy (tedy když závodníci doběhnou v jiné sekundě, nemusí se druhý posílat na trať znovu, jinak ano). Naše referenční řešení však počítá výsledky úplně přesně (dokonce bez nutnosti práce s desetinnými čísly – ano, jde to) a je možné, že pro některé vstupy budete tuto přesnost potřebovat také. Zkuste to :-)

27-Z4-4 Koňské skoky — 12 bodů

Malé Zuzce, Kevinově sestřičce, už je 5 let a Kevin ji začal učit hrát šachy. Zuzka všechny figurky docela chápe, ale dělá jí problémy správně skákat koněm. Na to jí Kevin vymyslel následující cvičení. Postavil jí na šachovnici 5 stejných černých koní a označil jí 5 cílových pozic, do kterých s nimi má doskákat. Jejím úkolem je přesunout koně do cílové pozice na co nejméně skoků. Aby pro ni úkol nebyl příliš těžký, tak během toho může postavit více koní na stejné políčko.

Tvar vstupu: Na vstupu na prvním řádku bude číslo N udávající rozměr šachovnice. $5 \leq N \leq 5\,000$. Na dalších pěti řádcích jsou vždy dvě čísla určující x-ové a y-nové souřadnice koní (indexujeme od nuly, tedy pozice $0 \ldots N-1$). Na posledních pěti řádcích jsou opět dvě čísla určující x-ové a y-nové souřadnice cílových pozic. Jednotlivé pozice na vstupu se mohou libovolně opakovat.

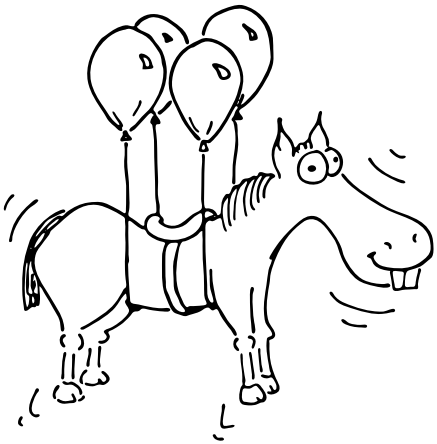
Tvar výstupu: Na výstup vypište jedno celé číslo udávající nejmenší počet skoků, kterými je možné se dostat ze startovní do cílové pozice. Jakýkoliv kůň může doskákat na jakoukoliv cílovou pozici, na pořadí nezáleží.

Ukázkový vstup:

```
První:     Druhý:
9          14
6 1        7 7
2 2        6 4
6 5        5 7
2 6        0 1
3 7        5 3
8 2        10 3
5 3        12 5
3 3        7 11
7 8        9 5
1 8        0 6
```

Ukázkový výstup:

```
První:     Druhý:
8          13
```



27-Z4-5 Poškolní trest — 12 bodů

Kevin dostal čtyřku z písemky z dějepisu. Už zase. To ho pěkně naštvalo a celou písemku roztrhal na malinké kousíčky. Dosáhl tak uspokojivého pocitu zadostiučinění. Alespoň do chvíle, kdy paní učitelka chtěla písemky vybrat zpátky. Kevin je poslušný hoch, tak písemku bez odmlouvání paní učitelce vrátil a vysloužil si tím dvě hodiny po škole.

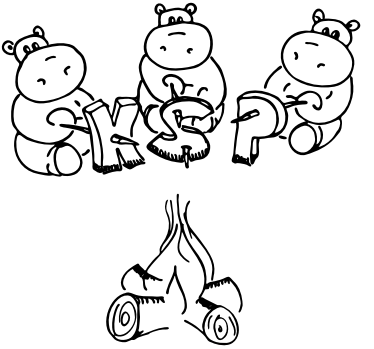
Tam paní učitelce musel pomáhat se tříděním papírů. Dostal jednu hromadu papírů očíslovaných na přeskáčku čísly 1 až N. Jelikož je líný, chtěl by to udělat následujícím způsobem: V každém kroku buď vzít vrchní, nebo spodní papír původní hromádky a položit jej na vršek, nebo spodek výstupní hromádky (která je na začátku prázdná). Po vás by chtěl navrhnout algoritmus, který pro zadanou hromadu papírů rozhodne, zda je, či není možné ji takovým způsobem setřídit.

Příklad: Hromadu papírů $2, 3, 4, 1$ setřídit lze, zatímco hromadu $6, 4, 5, 2, 1, 3$ setřídit nelze.

27-Z4-6 Příprava grilovačky — 14 bodů

Kevin, Sára a Petr plánují uspořádat monstrózní grilovačku, na kterou pozvou všechny své spolužáky. To bude akce! Ale taky spousta práce. Musí připravit pozvánky, upéct maso, ale ještě předtím jej koupit a rozdělat oheň. Než rozdělají oheň, tak zas musí nasekat dříví a tak dále. Všechny povinnosti si sepsali a vyšel jim seznam s dohromady N činnostmi. Mezi ně si pak nakreslili šipky podle toho, jak na sobě závisí. Například z pečení masa vede šipka na koupení masa a na rozdělání ohně a z rozdělání ohně vede šipka na nasekání dřeva.

Nyní se dohadují, v jakém pořadí jednotlivé činnosti nejlépe plnit. Jinými slovy chtějí najít takové pořadí vykonávání činností, aby před plněním konkrétní činnosti již určitě byly splněné všechny činnosti, na kterých daná činnost závisí. Navrhněte pro ně algoritmus, který takové pořadí najde, a nebo rozhodne, že žádné takové neexistuje.



Vzorová řešení KSP-Z

27-Z1-1 Na zastávce

Autobusovou úlohu vyřešíme tím nejjednodušším postupem, jaký vůbec může být: přímou simulací. To znamená, že v programu budeme provádět operace odpovídající tomu, co by se dělo v reálném světě (nástup skupinky do autobusu, odjezd autobusu), jednu po druhé ve stejném pořadí, v jakém by se odehrály doopravdy.

Samozřejmě většina programovacích jazyků neumí pracovat ani s lidmi, ani s autobusy, tak si místo toho pořídíme několik číselných proměnných, které budou náš svět popisovat:

- počet lidí v autobusu stojícím na zastávce
- počet autobusů, které již odjely
- pořadové číslo skupinky, která je aktuálně na začátku fronty

Nyní stačí postupně projít všechny skupinky a pro každou z nich upravit tyto proměnné tak, aby popisovaly situaci po odbavení dané skupinky dle pravidel v zadání. To je poměrně přímočaré a podrobněji si to můžete prohlédnout v ukázkovém programu.

Pozor je třeba dát si zejména na „plus/minus jedničkové“ chyby: tedy například nezapomenout započítat i poslední nezaplněný autobus, nebo naopak pokud se poslední autobus zcela zaplní, nezapočítat navíc ještě jeden prázdný.

Přímá simulace obvykle nepatří mezi nejefektivnější řešení, ale v případě naší úlohy jím je. Časová složitost řešení je lineární v počtu čekajících skupinek, a lépe to určitě nejde, neb v kratším čase by program ani nestihl přečíst svůj vstup.

Program (C): `http://ksp.mff.cuni.cz/viz/27-Z1-1.c`

Filip Štědronský

27-Z1-2 Kalkulačka

Myšlenkový postup řešení úlohy s kalkulačkou byl přímočarý, stačilo jen postupně načítat operátory a čísla a provádět s nimi zadané operace. Důležité ale bylo rozmyslet implementační detaily.

Načítat čísla a operátory ze vstupu je nejlepší dělat v cyklu – a to buď v lichých krocích čísla a v sudých operátory, nebo rovnou v každém kroku celou dvojici čísla a operátoru.

Ať to budeme provádět jakkoliv, jeden krok výpočtu vždy provedeme ve chvíli načtení dalšího čísla. V nějaké proměnné si budeme držet dosavadní výsledek, pak se podíváme na operátor (abychom nemuseli psát série if-podmínek, nabízejí některé jazyky zkratku konstrukcí switch) a provedeme to, co je po nás

KSP-Z

řešení

požadováno. Zde je také správné místo k ošetření dělení nulou, při dělení nulou neprovedeme nic.

Zbývají dvě otázky. První z nich je, kdy provádět vypisování mezivýsledků. Zadání úlohy vyžadovalo, abychom mezivýsledek vypisovali při každém načtení operátoru. Stačí si uvědomit, že to je to samé, jako když mezivýsledek vypíšeme ve chvíli jeho spočítání (protože další na vstupu přijde vždy operátor), a tak to také uděláme.

Poslední věcí je, jak výpočet odstartovat a jak ukončit. Ukončení je jednoduché, budeme i '=' brát jako operátor, jen se speciálním významem ukončení programu (všimněte si, že výsledek k tomuto operátoru už vypsala poslední operace).

Začátek výpočtu je složitější, protože vždy po načtení nového čísla chceme provést výpočet. Jaký ale provést pro první číslo? Abychom to nemuseli řešit speciální podmínkou, inicializujeme na začátku proměnnou s výsledkem na nulu a operátor na plus. A to je celé, na implementaci se podívejte v programech níže.

Program (C): `http://ksp.mff.cuni.cz/viz/27-Z1-2.c`

Program (Python 3): `http://ksp.mff.cuni.cz/viz/27-Z1-2.py`

Jirka Setnička

27-Z1-3 Slovník T9

Začneme tím, že každé slovo přeložíme na jeho zápis v T9. Poté čísla rozdělíme do skupin tak, aby v jedné skupině skončila slova, která se v T9 píší stejně. A nakonec najdeme největší skupinu.

Jak to udělat konkrétně? V Pythonu si můžeme pořídit slovník, jehož klíče budou jednotlivé zápisy v T9. Ke každému klíči přiřadíme pole, kam budeme ukládat všechna slova s tímto zápisem. Pak už jenom projdeme všechny klíče slovníku a najdeme ten, jehož pole je největší.

Rozmysleme si, jak rychlé naše řešení bude. Každá operace se slovníkem zabírá v průměru lineární čas s délkou klíče, nezávisle na tom, jak je slovník velký. (Slovník uvnitř funguje jako hešovací tabulka. Pokud vás zajímají detaily, nakoukněte do kuchařky o hešování.[5] Zde stačí vědět, že hešovací tabulky dovedou být velice rychlé, ale jenom v průměru; nejhorší případ může být až lineární s velikostí slovníku.)

Celý program proto poběží v průměrně lineárním čase se součtem délek všech slov, tedy s velikostí vstupu.

Program (Python 3): `http://ksp.mff.cuni.cz/viz/27-Z1-3.py`

Na Céčkovém řešení si předvedeme jiný přístup: nejprve vytvoříme dvojice (*původní slovo*, *převedené slovo*). Pak tyto dvojice setřídíme podle převedených

[5] `http://ksp.mff.cuni.cz/viz/kucharky/hesovani`

slov – můžeme se inspirovat kuchařkou o třídění,[6] případně použít knihovní funkci `qsort`.

Setříděním se dostanou k sobě slova se stejným zápisem, takže je snadno poznáme a najdeme největší takovou skupinu.

Opět si rozmysleme časovou složitost. Setřídění n hodnot kvalitním třídicím algoritmem (třeba MergeSortem) trvá $\mathcal{O}(n \log n)$ porovnání, projití setříděného slovníku spotřebuje dalších $\mathcal{O}(n)$ porovnání. A jelikož zadání omezuje slova na 15 písmen, můžeme předpokládat, že slova umíme porovnávat v konstantním čase. Časová složitost proto činí $\mathcal{O}(n \log n)$.

Program (C): `http://ksp.mff.cuni.cz/viz/27-Z1-3.c`

Martin „Medvěd“ Mareš

27-Z1-4 Lyžař

Nejprve si pojďme rozmyslet, jak bychom řešili situaci pro kopec výšky dva, který by vypadal třeba takto:

```
  1
2 3
```

U takového kopce se stačí podívat doleva a doprava, a kde je vyšší číslo, tam pojedeme. Teď si pojďme ukázat, že i z velkého kopce se dá postupně vyrobit kopec výšky dva. Nejprve si to ukážeme na kopci výšky tři:

```
  1
2 3
3 2 1
```

Pro snazší orientaci budeme jednotlivá místa na kopci označovat podle toho, v kolikáté jsou řadě seshora, a v kolikáté jsou řadě zleva, zapisovat to budeme jako $[3, 1]$ (což v tomto případě značí trojku ve třetím řádku vlevo).

Nejprve se podíváme na místo $[2, 1]$: Kdyby kopec začínal zde, stačí se podívat jen doleva a doprava dolů a víme, kam se máme z tohoto místa vydat. Tedy k místu $[2, 1]$ přičteme hodnotu v místě $[3, 1]$, protože je větší než v místě $[3, 2]$.

Teď si představíme, že kopec začíná v místě $[2, 2]$, a provedeme pro něj stejný postup jako pro místo $[2, 1]$ (jen k němu přičteme hodnotu z $[3, 2]$, protože je větší než v $[3, 3]$). Nyní můžeme zapomenout na třetí řádek, protože už víme, do kterých míst se nám vyplatí jet z druhého řádku.

Po zapomenutí třetího řádku máme opět kopec výšky dva a pro něj už umíme zjistit řešení jednoduše. Tento postup ale nezáležel na tom, že je kopec vysoký zrovna tři řady. Když tento postup zobecníme, budeme umět vyřešit i kopec libovolné výšky.

[6] `http://ksp.mff.cuni.cz/viz/kucharky/trideni`

Konstrukce zespodu

Zkusíme tedy zobecnit postup pro kopec výšky tři na kopec výšky N. Budeme ho postupně zespodu snižovat:

1. Pro i od 1 do $N-1$:
2. $\quad max \leftarrow$ maximum z $[N,i]$ a $[N,i+1]$
3. $\quad [N-1,i] \leftarrow [N-1,i] + max$

Na poslední řádek teď můžeme zapomenout. Tímto z kopce výšky N vyrobíme kopec výšky $N-1$ a opakováním postupu se dostaneme až na kopec výšky jedna, který už je sám o sobě řešením.

Tento postup bude pro kopec výšky N trvat $\mathcal{O}(N^2)$, protože na každé místo se podíváme maximálně třikrát a všech míst je dohromady $\mathcal{O}(N^2)$.

Řešení, které jsme si právě předvedli, se dá považovat za jednu z technik *dynamického programování*, neboli skládání řešení velkých problémů z řešení malých.

Program (C): `http://ksp.mff.cuni.cz/viz/27-Z1-4-dynamika.c`

Rekurzivní náhled

Druhý náhled na úlohu může být shora dolů. Kdybychom znali maximální součet na celé cestě začínající pod námi vlevo nebo vpravo, tak bychom si z nich už snadno vybrali, kam se máme vydat.

Toho můžeme využít pro řešení pomocí rekurze. Pokaždé se našeho programu zeptáme, jaký je součet vlevo a vpravo, porovnáme je, a pak jako výsledek vrátíme součet toho většího z nich a hodnoty v aktuálním místě.

Jen to nelze naprogramovat takto přímo. Kdybychom totiž pokaždé počítali výsledek pro všechna nižší místa, výpočet by se spouštěl pro každé místo mnohokrát. Kolikrát přesně může nastínit schéma níže. Znalejší z vás si možná všimli, že je to vlastně *Pascalův trojúhelník*:

```
      1
     1 1
    1 2 1
   1 3 3 1
  1 4 6 4 1
. . . . . .
```

Vidíte, že počet rekurzivních volání roste velmi rychle. Na prvním řádku se ptáme na dvě hodnoty pod námi, na druhém se ptáme na tři hodnoty pod námi, z toho ale na jednu dvakrát, na třetím řádku se ptáme již čtyřikrát, na čtvrtém osmkrát a tak dále.

Dalo by se napsat, že se na i-tém řádku ptáme řádově na 2^i hodnot pod námi, což nám dává celkovou časovou složitost $\mathcal{O}(2^N)$. Druhý možný náhled dávající stejnou složitost vypadá tak, že se podíváme na možné cesty, kudy se můžeme

vydat. Při cestě dolů z kopce se na každém místě rozhodujeme mezi dvěma směry a toto rozhodnutí děláme N-krát, což nám opět dává $\mathcal{O}(2^N)$ možností.

Abychom tak zbytečně mnohokrát nepočítali něco, co už víme, vždy si vypočtenou hodnotu uložíme do pomocného pole stejné velikosti, jako má sjezdovka, a zapamatujeme si, že pro tuto cestu již výsledek známe.

Když se pak v programu budeme ptát na hodnotu cesty, nejprve zjistíme, jestli už ji máme spočítanou, a pokud ano, jen vrátíme výsledek. Jinak spočítáme cestu, uložíme výsledek opět do pole a zapamatujeme si, že už pro toto místo cestu spočítanou máme.

Na každé místo se tak budeme ptát maximálně dvakrát, což bude trvat $\mathcal{O}(N^2)$, neboli lineárně s velikostí vstupu. Lépe to jistě nepůjde, neboť vstup musíme určitě přečíst celý. Kdybychom ho celý nečetli, můžeme do některého nepřečteného místa dosadit dostatečně velkou hodnotu, aby změnila optimální cestu, ale náš program by takové změněné řešení neměl jak poznat.

Ukázali jsme tedy dvě různá řešení, která ve výsledku vedou k něčemu velmi podobnému. První implementaci jsme ukázali již výše, na druhou se můžete podívat zde:

Program (C): `http://ksp.mff.cuni.cz/viz/27-Z1-4-rekurze.c`

Vojta Sejkora

27-Z1-5 Cédéčko z koncertu

Pomalé řešení

Ze zadání víme, že máme najít souvislý úsek písniček takový, že součet jejich délek je přesně K. Písniček je N, a tak si můžeme zvolit až N různých písniček, kterými by CDčko mohlo začínat. Ke každému možnému začátku existuje až N možných posledních písniček, což nám dává řádově N^2 možností (dvojic začátků a konců). Tak si je zkusme všechny projít a pro každou spočítat délku písniček mezi nimi (včetně jich samotných).

Jak to provedeme? Pojedeme v nějakém cyklu začátkem přes celou posloupnost písniček, v něm dalším cyklem přes možné konce a v každém takovém úseku ještě třetím cyklem spočítáme součet délek všech v něm obsažených písniček. Spočtení každého úseku bude trvat čas $\mathcal{O}(N)$, a pro N^2 úseků tedy celkově $\mathcal{O}(N^3)$.

To se nám ale moc nelíbí, tak to zkusíme zlepšit použitím prefixových součtů z naší základní kuchařky.[7] Čas potřebný na spočítání úseku se tím sníží na $\mathcal{O}(1)$ a celkově tedy na $\mathcal{O}(N^2)$, ale stále zbytečně počítáme pořád dokola skoro ty stejné věci. Ovšem my máme rádi rychlé algoritmy, nedá se to tedy zlepšit?

[7] `http://ksp.mff.cuni.cz/viz/kucharky/zakladni-algoritmy`

KSP-Z

řešení

Zrychlujeme

Algoritmus bude pracovat následovně. Budeme mít tři proměnné: začátek a a konec b, které budou ukazovat na první, respektive poslední písničku aktuálního úseku, a součet S délek písniček aktuálního úseku (na začátku bude velký jako délka první písničky). V každém kroku výpočtu porovnáme S s číslem K, mohou nastat tři možnosti:

- $S = K$: V tomto případě jsme vyhráli a nalezli jsme úsek se součtem K začínající písničkou a a končící b.
- $S < K$: Zde vidíme, že se nám na CD ještě něco vejde, tak zkusíme přidat další písničku. To znamená, že konec úseku posuneme o jednu písničku dál (b zvýšíme o jedna) a délku b-té písničky přičteme k S. Opakujeme porovnání.
- $S > K$: Tady už přidání další písničky nemůže pomoci (rozdíl by se pouze zvětšoval), tedy víme, že a-tou písničkou nemůže hledaný úsek začínat. Ovšem další písničkou ano, proto od S odečteme délku a-té písničky a a posuneme o jednu pozici dál na následující písničku. Opakujeme porovnání.

Pokud začátek nebo konec chceme posunout, ale už není na jakou písničku, tak ohlásíme, že neexistuje úsek písniček, který by bylo možné na CD zapsat (ve skutečnosti stačí kontrolovat pouze konec, neboť pokud bychom a zvýšili tak, že by „ukazovalo“ na neexistující písničku, tak by byl součet nulový, a tedy bychom posouvali b).

Důkaz správnosti

Algoritmus úspěšně seběhl, tak si ověřme, že bude fungovat vždy. Už víme, že možných úseků je řádově N^2, musíme je ale procházet všechny? Pokud je součet našeho úseku od a do b příliš velký, tak odebráním první písničky současně vyřadíme všechny další nezkontrolované úseky začínající touto písničkou (zkontrolovali jsme jenom úseky končící maximálně v b).

Ovšem každý takový úsek by měl součet určitě delší než úsek (a, b), který už sám byl příliš dlouhý. Vyloučili jsme tedy jen úseky, které by nás stejně nezajímaly.

Obdobným argumentem se můžeme podívat na posouvání b o jedna dál (všechny vyloučené ještě nezpracované úseky by byly příliš malé). Náš algoritmus tak vylučuje pouze ty úseky, u nichž už víme, že by nevyhovovaly. Všechny ostatní zkontrolujeme, tudíž pokud řešení existuje, najdeme ho.

Už víme, že algoritmus funguje, tak se pojďme podívat, jak dlouho mu to trvá. Při hledání úseku v každém kroku posuneme a nebo b o jedna dál, písniček je N, každá může být nejvýše jednou označena jako a a nejvýše jednou jako b. Tedy nejpozději po $2N$ krocích dojdeme na konec a ukončíme prohledávání. Celý algoritmus má tedy časovou složitost $\mathcal{O}(N)$.

Program (Python 3): `http://ksp.mff.cuni.cz/viz/27-Z1-5.py`

Katka Zákravská & Jirka Setnička

27-Z1-6 Žárovky

Pre lepšie pochopenie riešenia si trošku upravíme zápisy. Ku žiarovkám pridáme ešte jednu, ktorá bude stále svietiť, pomenujme ju žiarovka *Nádej*.

Pôvodný počet žiaroviek si označíme ako $n-1$ a pridaním Nádeje budeme mať n žiaroviek. Nádej vždy svieti, a teda nám problém úlohy s $n-1$ žiarovkami upravuje na problém s n žiarovkami. Prečo tomu tak je? V pôvodnom prípade rozsvecujeme žiarovky v počtoch $0, 1, 2, \ldots, n-1$. V druhom prípade to upravíme na $1, 2, \ldots, n$, čo je vlastne Nádej $+ (0, 1, 2, \ldots, n-1)$.

Zjednodušená úloha

Úlohu si na začiatok ešte trošku zjednodušíme. Obmedzíme počet žiaroviek len na mocniny dvojky $(1, 2, 4, 8, \ldots)$. Na túto zjednodušenú úlohu použijeme nasledovný algoritmus:

1. Žiarovky si rozdelíme na dve časti (polovice).
2. K časti, ktorá neobsahuje Nádej, pridelíme jeden spoločný vypínač.
3. Ak časť obsahuje len Nádej, tak program ukončíme (táto žiarovka už nepotrebuje vypínač, lebo sa nedá vypnúť). Inak opakujeme algoritmus od kroku jedna zavolaním sa na časť obsahujúcu Nádej.

Naším prvým krokom bude ukázať, že takéto rozdelenie vypínačov je správne – teda, že rozsvieti ľubovoľný počet žiaroviek z intervalu 1 až n. Využijeme k tomu indukciu. Mať rozsvietenú práve jednu žiarovku vieme pomocou žiarovky Nádej, pričom všetky ostatné vypínače sú vypnuté.

Implikáciu ukážeme takto: ak vieme postupne rozsvietiť $k/2$ žiaroviek (do stavu 1 až $k/2$ rozsvietených) a zároveň máme vypínač, ktorý rozsvieti druhú polovicu (teda presne $k/2$ žiaroviek), tak vieme rozsvietiť aj ľubovoľný počet žiaroviek od 1 do k. Stačí nám na to rozsvietiť druhú polovicu jedným vypínačom. A potom k takto rozsvietenej polovici vieme pridať 1 až $k/2$ rozsvietených žiaroviek.

Ďalej si ukážeme, že naše riešenie je optimálne. Náš algoritmus potrebuje i vypínačov pre 2^i žiaroviek. Každý vypínač sa môže nachádzať v jednej z dvoch polôh, a to buď zapnutý, alebo vypnutý. S i vypínačmi môžeme popísať práve 2^i rôznych stavov. Z rôzne „postláčaných“ vypínačov chceme získať rôzny počet rozsvietených žiaroviek. Ak by sme vypínačov mali len $i-1$, tak s nimi vieme dosiahnuť maximálne 2^{i-1} stavov. My ale potrebujeme 2^i rôznych zasvietení (1 až 2^i).

Pôvodná úloha

Na záver sa vrátime k pôvodnému zadaniu úlohy. Teda hľadáme riešenie pre ľubovoľný počet žiaroviek, nie len pre mocninou dvojky. No nezúfajme, naše predošlé riešenie sme nerobili zbytočne. Skupinu n žiaroviek si rozdelíme na dve časti. Prvá časť bude obsahovať našu Nádej a počet žiaroviek v tejto časti bude

rovný najväčšej mocnine dvojky, ktorá je menšia alebo rovná n. Exponent tejto mocniny si označíme ako i.

Druhá časť bude obsahovať všetky zvyšné žiarovky a bude určite menšia ako tá prvá časť. Je to preto, lebo ak by bola druhá časť väčšia alebo rovná ako tá prvá, tak by sme potom mohli vziať väčšiu mocninu dvojky v prvej časti, než sme vzali. Prvú časť, ako sme dokázali v zjednodušenej úlohe, vieme najlepšie vyriešiť s i vypínačmi. Celú druhú časť pripojíme na jeden spoločný vypínač. Prvými i vypínačmi vieme rozsvietiť 1 až 2^i žiaroviek a posledným vypínačom zvyšok, teda $n - 2^i$ žiaroviek. Ak chceme rozsvietiť viac ako 2^i žiaroviek, stačí nám rozsvietiť druhú časť a k tomu doplňujúci počet žiaroviek z prvej časti.

Ešte potrebujeme ukázať, že potrebný počet vypínačov je najmenší možný. Pre n žiaroviek si nájdeme najbližšiu menšiu mocninu dvojky než n. Túto mocninu si označíme ako 2^i. Následne využijeme podobnú úvahu, akú sme použili pri 2^i žiarovkách. Ak by nám stačilo i vypínačov, tak vieme popísať 2^i stavov. Pritom vieme, že $2^i < n$, teda potrebujeme najmenej $i + 1$ vypínačov.

Janka Bátoryová & Karolína „Karryanna“ Burešová

27-Z2-1 Závorky z cereálií

Dostali jsme několik závorek. A naším úkolem je zjistit minimální počet závorek, které musíme doplnit, aby výsledná posloupnost byla správně uzávorkovaná. Tento počet je roven počtu nespárovaných závorek uvnitř posloupnosti.

Nespárované závorky budeme hledat tak, že projdeme pole a cestou si budeme pamatovat počet zatím nespárovaných závorek. Řekneme si, že za každou otevírací závorku '(', na kterou narazíme, zvýšíme počet zavíracích závorek ')' potřebných k doplnění. Tento počet nazveme P. Analogicky nám závorky) budou P snižovat. A po průchodu máme v P uložený počet závorek), které musíme doplnit, abychom měli všechny (v pořádku.

Ještě nám zbývá dořešit počet otevíracích závorek k doplnění. Ty budeme počítat v proměnné L. Proměnnou L zvýšíme vždy, když při průchodu najdeme zavírací závorku, ale nemáme žádnou otevírací, se kterou bychom ji spárovali.

Na konci průchodu máme v P uložen počet pravých a v L počet levých závorek, které je potřeba doplnit do posloupnosti, aby byla správně uzávorkovaná. Celkový počet pak získáme součtem $P + L$.

Časová složitost tohoto algoritmu je $\mathcal{O}(n)$, kde n je velikost vstupu. Tedy lineární, protože nám na zjištění výsledku stačí jenom jeden průchod zadaného vstupu. Paměťová složitost záleží na načítání vstupu. Pokud bychom načítali právě jeden znak, tak by paměťová složitost byla konstantní. Ale pro naše vzorové řešení je paměťová složitost lineární, protože si celý vstup pamatujeme najednou.

Program (Python 3): `http://ksp.mff.cuni.cz/viz/27-Z2-1.py`

Martin Šerý

27-Z2-2 Hrnce od Horsta

Abychom pomohli hrnce poskládat, budeme muset vyřešit několik podproblémů a začneme tím, že si posloupnost načteme do paměti. Po načtení do paměti posloupnost setřídíme (ve vzorovém řešení je setříděna vzestupně). Nyní musíme udělat ještě dvě věci. Spočítat, kolik hromádek hrnců budeme mít, a poté nějaké vytvořit. To uděláme pěkně postupně.

Spočítání počtu hromádek hrnců není nijak těžké a zvládneme ho za jeden průchod. Budeme si pamatovat, kolik hromádek právě teď máme (na začátku máme právě jednu o právě jednom hrnci – tom prvním), a počet hromádek zvětšíme pokaždé, když v posloupnosti objevíme více hrnců stejného průměru za sebou, než je počet hromádek. Pokud tedy máme zatím tři hromádky o maximálním průměru deset a narazíme na sedm hrnců s průměrem jedenáct, budeme potřebovat hromádek sedm, do tří hrnců dáme ty tři menší hromádky a čtyři nové musíme založit. Toto zvládneme v $\mathcal{O}(N)$.

Nyní potřebujeme nějaké hromádky vytvořit. To můžeme udělat například takto – víme, kolik hromádek budeme potřebovat, takže si je napřed založíme (ve formě spojových seznamů či dostatečně velkých polí). Pokaždé načteme všechny hrnce stejné velikosti a přidáme je po jednom do hromádek. Pokud byl hrnec dané velikosti jen jeden, přidáme ho jen do první, pokud byly dva, přidáme je do první a druhé, atd. Na konci jen hromádky vypíšeme.

V našem příkladu máme tedy maximálně sedm hrnců o stejném průměru. Založíme si tedy sedm hromádek a všechny unikátní hrnce dáme do první, hrnce s průměrem deset byly tři, ty tedy rozházíme po jednom do prvních třech hromádek a poté již sedm hrnců s průměrem jedenáct dáme do všech sedmi.

Vypsání opět zvládneme v lineárním čase, a tudíž nejnáročnější částí našeho programu je třídění, které seběhne v čase $\mathcal{O}(N \log N)$. To je tedy časová složitost vzorového řešení, nicméně ani pomalejším řešením se složitostí $\mathcal{O}(N^2)$ by vstupy neměly trvat o mnoho déle. Paměti spotřebujeme lineárně, pouze načteme vstup a poté vytvoříme hromádky. Paměťová složitost je tedy $\mathcal{O}(N)$.

Program (C++): `http://ksp.mff.cuni.cz/viz/27-Z2-2.cpp`

Program (Python 3): `http://ksp.mff.cuni.cz/viz/27-Z2-2.py`

Štěpán Hojdar

27-Z2-3 Nápis na tričku

Tato úloha mohla na první pohled vypadat složitě, ve skutečnosti je přímočará. Stačilo si spočítat četnost jednotlivých písmen. Jak na to?

Pořídíme si pole 26 čísel – tolik je písmen anglické abecedy, kterou používáme. Potom stačí přečtené slovo vzít znak po znaku, a patřičný chlívek pole zvětšit o jedna. V drtivé většině programovacích jazyků se dá k řetězci přistupovat jako k poli znaků, tj. dívat se na jeho jednotlivé znaky. A většina jazyků má též funkce pro převod znaku na číslo, třeba podle ASCII tabulky. Například v jazyce `C` je znak a číslo to samé, rozlišíte je až podle použití. Proto stačí od daného znaku odečíst `'a'` a máme index do pole četností. Podobný trik bude fungovat skoro všude.

Když už si umíme spočítat četnosti znaků, spočítáme si je nejprve pro slovo S ze zadání. To je jediné, co si potřebujeme pamatovat. Pak už stačí pro každé slovo spočítat četnosti, jedním cyklem ověřit, jestli se každý znak vyskytuje nejvýše tolikrát, kolikrát se vyskytuje v S. Pokud ano, vypíšeme ho, jinak pokračujeme.

Na závěr jen dodáme, že tabulce četnosti se obvykle říká *histogram*. Také doufáme, že jste si všimli opravdového seznamu německých firem v jednom ze vstupů.

Program (C++): `http://ksp.mff.cuni.cz/viz/27-Z2-3.cpp`

Ondra Hlavatý

27-Z2-4 Hořící auto

Úloha se na první pohled může jevit jako problém hledání průsečíků přímek, což je docela obtížný problém. Zkusme si to ale nejdřív nakreslit. Těch čar může být hodně, kreslit to rukou by trvalo dlouho. Nedal by se k tomu využít počítač?

Dal. V některých programovacích jazycích existují knihovny na takzvanou *želví grafiku*. Želva tam dělá to samé, co auto. Chodí rovně, zatáčí a přitom za sebou kreslí čáru. Dají se tím kreslit všelijaké pěkné obrázky. Tato úloha je opendatová, řešení můžeme vytvořit, jak chceme. Necháme tedy želvu, aby nám nakreslila obrázek, potom se na něj podíváme a zjistíme, po které zatáčce auto vybuchlo.

Program – želví grafika: `http://ksp.mff.cuni.cz/viz/27-Z2-4-zelva.py`

Pár bodů se tímto řešením dalo získat, v některých vstupech ale bylo čar hodně, takže to nešlo tak lehce. Vykreslováním ovšem zjistíme, že díky pravoúhlému zatáčení se tento problém dá řešit mnohem jednodušeji.

Pomalé, ale přímočaré řešení

Jedno z nejpřímočařejších řešení je pamatovat si všechny čáry a počítat, jestli se nová neprotne s nějakou předchozí.

Díky tomu, že čáry jsou pravoúhlé, tak to, zda se dvě přímky protínají, se dá určit velice snadno. Čáry budeme mít reprezentované jako dvě dvojice bodů, tedy $[x_z, y_z], [x_k, y_k]$ (index z je začátek, k konec, a navíc zajistíme, aby platilo, že $x_z \leq x_k$ a to stejné pro y).

Pak pro každou vodorovnou přímku (to je ta, pro kterou je $y_z = y_k$) zjistíme, jestli $x_z \leq x_{z_i} \leq x_k$ a $y_{z_i} \leq y_z \leq y_{k_i}$, kde i ukazuje na i-tou čáru. Obdobně to provedeme pro všechny svislé přímky, jen ve formuli prohodíme x a y.

To je ale pomalé, protože pro každou čáru musíme otestovat i všechny předchozí, z toho plyne časová složitost $\mathcal{O}(N^2)$.

Řešení přes velikost volného prostoru

Na první pohled vidíme, že křivka vzniklá jízdou auta může být spirála ze středu ven (té budeme říkat zvětšující se), nebo spirála z vnějšku do středu (té budeme říkat zmenšující se). Také se může stát, že chvíli křivka bude zvětšující se spirála, a pak přejde do zmenšující se spirály. Důležité je, že jakmile se auto dostane do zmenšující se spirály, pak už se z ní nedostane, protože by nutně překřížilo křivku, po které již jelo, a tedy by vybuchlo.

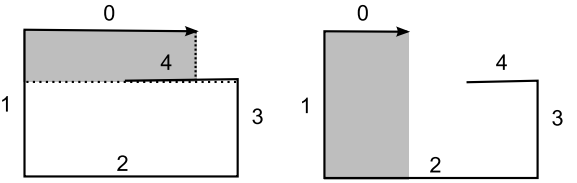
Nejprve si pojďme rozebrat ty dva jednodušší případy, tedy pouze zvětšující se spirálu a pouze zmenšující se spirálu. Pro zvětšující se spirálu platí, že dokud auto v daném směru (myšleno vodorovně, nebo svisle) ujede více než minule (tedy pokud délka aktuální čáry je delší než délka čáry o dva kroky dříve), pak nemůže protnout křivku. Pro zmenšující se spirálu obdobně platí, že dokud v daném směru ujede méně než minule, pak nemůže protnout křivku. Problém nastává,

KSP-Z

když přecházíme ze zvětšující se spirály do zmenšující se. Tehdy si potřebujeme spočítat, kolik místa máme v kolmém směru na směr, kterým jsme zrovna jeli (tedy jak dlouhá je čára o krok dříve). V aktuálním směru je právě tolik místa, kolik jsme ujeli.

řešení

Tehdy mohou nastat dvě možnosti:



Na obrázcích je šedé místo prostor, ve kterém se může nacházet zmenšující se spirála. Tento prostor si uložíme jako minulý v kolmém směru (tedy jako délku čáry o krok dříve), tedy jakoby zmenšíme vzdálenost, kterou jsme minule urazili v kolmém směru (tento krok nám mírně zjednodušší implementaci).

A to je nejsložitější část. Z obrázků a po rozmyšlení implementace je vidět, že nám stačí si pamatovat posledních pět čar, což můžeme snadno udělat pomocí pole délky 5 a modulení. Pozor na to, že v některých programovacích jazycích záporné číslo modulo kladné je záporné číslo, což my nechceme. Jednoduše přičteme 5, protože tím výsledek neovlivníme (ve smyslu, že $5 \bmod 5 = 0$). Přitom zamezíme tomu, abychom dostali záporná čísla. Druhá možnost je ukládat si čáry zprava doleva, tedy že délka minulé čáry je na pozici $(aktualni + 1) \bmod 5$, předminulé na $(aktualni + 2) \bmod 5$, ...

Nyní ještě zbývá vymyslet, v kterém „módu“ začínat. Jelikož zatím nemáme žádné čáry, tak jistě první i druhá bude vést na zvětšující se spirálu, tedy můžeme začít v módu zvětšující se spirály.

Ještě se zamyslíme nad časovou složitostí. Pokud jsme v módu zvětšující se spirály, tak provedeme jen jedno porovnání, to stejné pro zmenšující se spirálu. Přechod ze zvětšující se do zmenšující se spirály nastane maximálně jednou a práce tam provedeme jen konstantně (podíváme se maximálně na pět posledních a z nich spočítáme volné místo). Tedy každou čáru umíme vyhodnotit v konstantním čase ($\mathcal{O}(1)$), čar je N, a proto časová složitost je $\mathcal{O}(N)$.

Program – zmenšování volného prostoru (C):
`http://ksp.mff.cuni.cz/viz/27-Z2-4-misto.c`

Ukážeme vám i další naprogramované řešení. Základní idea je podobná, avšak v tomto řešení neuvažujeme o dvou různých spirálách, ale jen se díváme na pět posledních čar. Více detailů naleznete v komentářích zdrojového kódu.

Program – pět posledních čar (C):
`http://ksp.mff.cuni.cz/viz/27-Z2-4-prusecik.c`

Vojta Sejkora

27-Z2-5 Hledání stromů

Našou úlohou je zistiť, či je zadaný graf stromom. K tomu potrebujeme overiť či graf neobsahuje žiadne cykly a či je súvislý. Využijeme pritom prehľadávanie do hĺbky (DFS), o ktorom sa dočítate aj v kuchárke.[8]

Pozrime sa na jednoduché prirovnanie. Graf si predstavíme ako mesto. Medzi každými dvoma križovatkami (vrcholmi) existuje len jedna cesta – to znamená, že tam neexistujú cykly, inak by bolo viacero spôsobov ako sa z jednej križovatky dostať do druhej. Ako keď sa v kruhu môžeme vybrať dvoma smermi. Mesto začneme prehľadávať pred domom, v ktorom bývame. Ak z neho vedie viac ciest, tak si vyberieme ľubovoľnú. Ak len jedna, tak sa vyberieme ňou. Vždy, keď prídeme na križovatku, vyberieme si cestu, ktorou sme ešte nešli. Ak dôjdeme na križovatku, z ktorej vedie jediná cesta a to je tá, ktorou sme prišli, tak sa ňou vrátime o križovatku späť. Ak na križovatke nájdeme cestu, ktorou sme ešte nešli, tak sa ňou vydáme. Ak nenájdeme, tak sa vrátime po ceste, ktorou sme sa prvýkrát dostali k aktuálnej križovatke. Bude to cesta k predchádzajúcej križovatke smerom k domovu.

Náš postup zhrnieme do dvoch krokov:

1. ak existuje z križovatky cesta, ktorou sme ešte nešli, tak sa ňou vydáme a dôjdeme na ďalšiu križovatku
2. ak taká cesta neexistuje, tak sa vrátime o križovatku „vyššie“

K tomu nám poslúži rekurzia. Každú časť mesta prehľadávame rovnakým spôsobom. Vždy keď prídeme na križovatku, tak vykonáme uvedené kroky. Skončíme vtedy, ak už nemáme kam ísť.

Takéto mesto s križovatkami a cestami predstavuje graf. Ak naše mesto neobsahuje cykly a je súvislé, potom musí byť stromom. Ak sa v grafe vyskytuje cyklus, tak sa na niektorú križovatku vrátime opäť a to po inej ceste, ako sme z tej križovatky odišli. Teda v grafe vedie hrana do vrcholu, ktorý sme už raz navštívili. Taktiež je potrebné v každom vrchole vedieť, z ktorého vrcholu sme doň prišli. Do predchádzajúceho už navštíveného vrcholu vedie hrana, no hrana naspäť netvorí cyklus. Na koniec overíme, či je graf súvislý. Spýtame sa o každej križovatke, či sme na nej boli. S týmito splnenými podmienkami je graf určite stromom.

Na koniec si rozmyslíme časovú a pamäťovú zložitosť riešenia. Počet križovatiek si označíme ako N a počet hrán ako M. Načítanie vstupu nám zaberie čas $\mathcal{O}(N+M)$. Prehľadanie celého grafu do hĺbky nám zaberie taktiež čas $\mathcal{O}(N+M)$. Je to preto, že v rekurzii každý vrchol navštívime maximálne raz a každou hranou prejdeme maximálne dva krát (tam a späť). Keďže si pamätáme celý graf a počas prehľadávania si naviac ešte pamätáme, ktoré vrcholy sme navštívili, pamäťová zložitosť bude taktiež $\mathcal{O}(N + M)$.

[8] `http://ksp.mff.cuni.cz/viz/kucharky/grafy`

Môžme ešte uvažovať prípad, že vstupný graf už máme uložený v pamäti a tá sa do výslednej pamäťovej zložitosti nezapočítava. Keďže strom má vždy $N-1$ hrán, stačí nám na začiatku overiť, či táto podmienka platí (a následne graf prehľadať). Prehľadávanie grafu o $N-1$ hranách a N vrcholoch nám zaberie čas $\mathcal{O}(N)$. Ďalej si počas prehľadávania musíme pamätať, ktoré vrcholy sme navštívili. Preto pamäťová zložitosť v tomto prípade bude $\mathcal{O}(N)$.

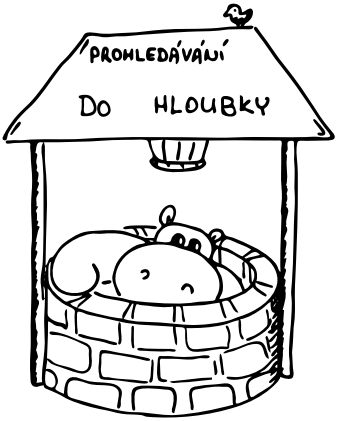
Drobná poznámka: Ak načítavame vstup a vieme dopredu počet vrcholov (N), potom nám stačí prečítať iba prvých $N-1$ hrán. Ak je hrán na vstupe menej, graf bude nesúvislý. Ak vstup obsahuje viac hrán, potom graf bude obsahovať cyklus. Ak nemusíme prečítať pri riešení úlohy celý vstup, môžme ho po $N-1$ hrán zarezať a tým docieliť časovú zložitosť $\mathcal{O}(N)$ aj v takomto prípade.

Pár slov k bonusu – graf zadaný maticou susednosti: Susedia vrcholu X v takejto tabuľke sú tí, ktorí na pozícii $[X, i]$ pre i od 1 do N majú hodnotu „1“. Teda susedov nemáme v zozname, ale zapísaných v matici. Graf budeme prehľadávať úplne rovnako. Matica má veľkosť NxN, teda načítavanie vstupu ako aj pamäťová zložitosť bude $\mathcal{O}(N^2)$. Zistenie všetkých susedov jedného vrcholu znamená prejsť jeden riadok tabuľky, čo je N položiek. Pri prehľadávaní do hĺbky potrebujeme zistiť všetkých susedov pre každý vrchol. Časová zložitosť prehľadávania bude teda $\mathcal{O}(N^2)$.

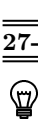
Ak budeme uvažovať prípad, že maticu už máme načítanú v pamäti, potom sa nám pamäťová zložitosť zlepší na $\mathcal{O}(N)$ (nerátame do toho pamäťovú zložitosť vstupu). Je to preto, že si stačí pamätať, ktoré vrcholy sme už navštívili a ktoré ešte len navštívime. Časová zložitosť sa ale kvôli zisťovaniu susedov nezmení a bude aj v tomto prípade $\mathcal{O}(N^2)$.

Program (Python 3): `http://ksp.mff.cuni.cz/viz/27-Z2-5.py`

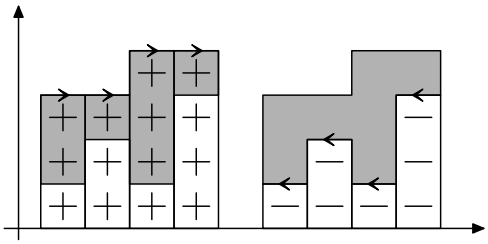
Janka Bátoryová & Pali Rohár



27-Z2-6 Povrch dálnice



Pro začátek si představme, že obrazec je nakreslený v souřadnicovém systému nad osou x. Hlavní myšlenkou je, že budeme přičítat a následně odečítat obsahy obdélníčků. Lépe je to vidět na následujících obrázcích. Obdélníčky vlevo přičítáme a vpravo odečítáme.



Představu už máme, tak si to pojďme lépe popsat. Budou nám stačit dvě proměnné y a S. Proměnná y, jak již prozrazuje její název, odpovídá naší pozici na ose y. V proměnné S si budeme postupně počítat obsah obrazce. Při pohybu na sever k y jednoduše přičteme ušlou vzdálenost a na jih zase odečteme. Pohyb na východ o vzdálenost l vymezí obdélníček o rozměrech $l \times y$, jehož obsah $l \cdot y$ přičteme do S. Při cestě na západ naopak obdélníček odečteme.

Po zpracování celého vstupu máme obsah obrazce v proměnné S. Ještě se nám může stát, že plocha vyjde záporně. To nastane v případě, že jsme obrazec obcházeli proti směru hodinových ručiček. Směr nás ale nezajímá, takže výsledkem je absolutní hodnota S.

Možná se ptáte, jakou nastavit hodnotu y na začátku, když neznáme souřadnice startu. V podstatě je to jedno, neboť to pouze znamená posun osy x, a tedy zvětšení všech obdélníčků o konstantu. Máme však zaručeno, že končíme na startu, takže ke každému přičtenému obdélníčku máme jeden odečtený. Tudíž jsme konstantu vždy jednou přičetli a jednou odečetli, což výsledek nikterak neovlivní.

Dokonce ani nepotřebujeme, aby bylo y po celou dobu kladné. U obdélníčků pod osou x nám stačí prohodit, kdy se obsah přičítá a odečítá. Protože y je záporné, vyjde i hodnota $l \cdot y$ záporně. Původně při pohybu na západ obsah odečítáme, to tedy znamená, že odečteme zápornou hodnotu. Tím jsme ovšem dostali přičítání. Stejně získáme odečítání obsahu při pohybu na východ (přičítáme zápornou hodnotu).

Paměťová složitost je konstantní, protože vstup nikam neukládáme, ale rovnou jej zpracováváme během čtení. Časová složitost je lineární, jelikož se nikde nezdržujeme a pro každý pohyb provedeme pouze jednoduchou operaci.

Program (Python 3): `http://ksp.mff.cuni.cz/viz/27-Z2-6.py`

Katka Zákravská & Jenda Hadrava

KSP-Z

řešení

27-Z3-1 Kevin nabíječ, s.r.o.

Když jste nad úlohou chvíli přemýšleli, případně si zkoušeli různá zapojení, asi jste přišli na to, že na způsobu zapojení vůbec nezáleží, dokud budou všechny prodlužovačky připojeny k té jedné ve zdi. Každá prodlužovačka jednu zdířku použije pro své napájení a k jich poskytne pro libovolné další použití. Pokud použijeme všechny dostupné prodlužovačky, můžeme prostě spočítat součet všech $k-1$. To uděláme nejsnáze tak, že sečteme všechna k a na konci odečteme N (a přidáme jedničku za zásuvku ve zdi).

V zadání byl ale malý háček, který jsme se snažili naznačit obrázkem. Kevin si nakoupil i prodlužovačky, které měly nula zdířek, a takové zapojovat nechceme. Takže je stačí prostě z řešení vynechat. Tento chyták byl jen v posledním testu, takže i pokud jste si ho nevšimli, mohli jste dostat většinu bodů. Na takové chytáčky si ale dávejte pozor.

Program (Python 3): `http://ksp.mff.cuni.cz/viz/27-Z3-1.py`

Ondra Hlavatý

27-Z3-2 Nedej vitagen

Maximální délka slova na vstupu byla v zadání 100. To je dostatečně malá konstanta na to, abychom se jí nemuseli zabývat. Soustředíme se hlavně na to, aby náš program byl co nejrychlejší vzhledem k počtu slov na vstupu.

Připravíme si spojový seznam všech zadaných slov a druhý spojový seznam, ve kterém budou také všechna slova, akorát napsaná pozpátku. Teď stačí jen najít slova, která jsou v obou seznamech.

To se dá dělat více způsoby. Nejsnazší je prostě každé slovo z prvního seznamu porovnat se všemi ze druhého a zapsat si ho stranou, pokud se s nějakým shoduje. To by mělo časovou složitost $\mathcal{O}(N^2)$, kde N je počet slov na vstupu.

O něco mazanější a výrazně rychlejší způsob je oba seznamy setřídit abecedně a potom je šikovně zkoumat najednou. Na třídění posloupnosti je spousta algoritmů, které skončí v čase $\mathcal{O}(N \log N)$.[9]

Jak přesně tedy budeme seznamy prohlížet? Budeme postupně procházet oba seznamy najednou a v každém kroku se podíváme na první slova v obou seznamech. Pokud jsou stejná, nalezli jsme požadovanou shodu. Jinak z nich vybereme to lexikograficky menší (neboli to, které by se ve slovníku objevilo dřív) a to smažeme. Tím se v tomto seznamu posuneme o jedno dál.

Pokaždé tak pracujeme pouze se začátkem nějakého spojového seznamu, což je velmi rychlé, protože nahlédnutí na první prvek i jeho smazání ve spojovém seznamu trvá konstantní čas (nezávisí na velikosti seznamu).

[9] `http://ksp.mff.cuni.cz/viz/kucharky/trideni`

Pokud je nějaké slovo v obou seznamech, někdy během mazání se stane, že oba dva seznamy budou mít toto slovo na začátku. Proto stačí před každým mazáním zjistit, jestli se náhodou slova na začátcích seznamů neshodují. Mezi dvěma mazáními slov provedeme konstantní počet operací a mazání je $2N$, všechna dohromady budou tedy trvat $\mathcal{O}(N)$. Z toho plyne, že časová složitost celého algoritmu je $\mathcal{O}(N \log N + N) = \mathcal{O}(N \log N)$.

Najít mezi nimi to nejdelší už je triviální, můžeme to dělat třeba tak, že si v průběhu pamatujeme jenom dočasného adepta na vítěze a přepíšeme ho leda delším slovem.

Program (Python 3):
`http://ksp.mff.cuni.cz/viz/27-Z3-2.py`

Martin Španěl

27-Z3-3 Superstromy

Podle počtu správných řešení se zdá, že jste s úlohou neměli příliš problémů. Ono také nebylo potřeba vymýšlet nic světoborného. Stačilo si jen uvědomit, že si nikdy neuškodíme tím, že stromy rostoucí pomalu zasadíme jako první.

Proč to platí? Vždy alespoň jeden strom doroste jako poslední. Jak bychom mohli naše řešení zlepšit? Jedině tím, že tento poslední strom zasadíme v nějaký dřívější den. Pokud jsme však v dřívějších dnech sázeli pouze pomalejší stromy, prohozením si párty akorát odložíme na později.

Optimální pořadí stromů tedy dostaneme tak, že si je seřadíme sestupně dle počtu dní, které rostou. Následně jen stačí najít maximum ze součtu doby růstu a čísla dne, ve který daný strom zasadíme.

Dostali jsme tak řešení s časovou složitostí $\mathcal{O}(N \log N)$ a paměťovou $\mathcal{O}(N)$. To bohatě stačilo na plný počet bodů. Kdo si však všiml nízkého limitu na dobu růstu, mohl řešení o něco vylepšit.

Program (Python 3):
`http://ksp.mff.cuni.cz/viz/27-Z3-3.py`

Protože existuje pouze $T = 1\,000$ různých typů stromů, můžeme si v poli velikosti T uložit, kolik kterých z nich máme. Díky tomu zvládneme čísla třídit v čase $\mathcal{O}(N+T)$. Dokonce si i vystačíme jen s $\mathcal{O}(T)$ pamětí, protože si nemusíme pamatovat celý vstup.

Ačkoli popsaná myšlenka není vůbec složitá, vysloužila si svůj vlastní název. Takovému třídění čísel se říká *counting sort*. Vyplatí se jej použít v případech, kdy chceme seřadit velké množství stejných hodnot.

Program – counting sort (Python 3):
`http://ksp.mff.cuni.cz/viz/27-Z3-3-counting.py`

Jenda Hadrava

KSP-Z

27-Z3-4 Robo Rally

V této úloze stačí vymyslet, jak v programu reprezentovat herní plochu a roboty a krok po kroku odsimulovat, co se děje.

řešení

Je mnoho možností, jak k problému přistoupit. Hlavní je nějakým způsobem reprezentovat robota, třeba jako objekt nebo klidně jako seznam čísel udávajících jeho pozici a orientaci. Například seznam $[7, 1, 2, \texttt{D}]$ by mohl reprezentovat robota číslo 7, který je v prvním řádku, druhém sloupci a dívá se směrem dolů.

Potom stačí vymyslet, kam roboty dát. Je rozumné mít je v poli indexovaném podle jejich identifikátorů, abychom se snadno dostali k robotovi, kterého zrovna potřebujeme.

Často navíc budeme zjišťovat, co je na nějakém políčku. Nejjednodušší způsob je prozkoumat souřadnice všech robotů a zjistit, jestli se nějaká neshoduje se zkoumaným políčkem. To by ale mohlo trvat dlouho a děláme to často. Pokud nám záleží na rychlosti, můžeme si udělat pomocné dvourozměrné pole, kde na každém políčku bude číslo robota, který na něm je (nebo rovnou odkaz na něj), nebo -1 (resp. nulový odkaz), pokud tam žádný není.

Teď už nebude problém načíst pozice a orientace robotů a umístit je do naší datové struktury. Otáčení robotů je taky triviální. Pokud směry reprezentujeme písmeny, stačí k tomu několik podmínek. Pokud reprezentujeme směry čísly $0, 1, 2, 3$, můžeme si pomoci operací modulo, která nám umožní zjistit zbytek po dělení čtyřmi. Tedy když se chceme otočit o tři doprava, tak k současnému směru přičteme trojku a výsledek vymodulíme čtyřmi.

Zajímavější je pohyb. Musíme si dát pozor, že je potřeba simulovat krok po kroku. Tedy posun o pět políček budeme muset rozložit na pět kroků a každý vyhodnotit zvlášť. Pokud nám v nějakém kroku nestojí nic v cestě, tak je to snadné. Stačí přepsat posunutému robotovi souřadnice a případně upravit pomocnou tabulku.

Pokud nám stojí něco v cestě, řekneme tomu, ať udělá krok stejným směrem a podle stejných pravidel jako my. Tedy pokud mu bude stát něco v cestě, tak tomu taky řekne, ať se to posune. Takto se o tom dozví celá řada. Je důležité ošetřit případ, kdy se řada už nemá kam posunout (je před ní zeď). Kvůli tomu je potřeba ještě před samotným posunem nejdříve ověřit, zda se vůbec můžeme pohybovat, a teprve potom posun provést.

Po provedení všech kroků stačí vypsat postupně souřadnice a orientace robotů.

Program (Python 3):
`http://ksp.mff.cuni.cz/viz/27-Z3-4.py`

Martin Španěl

27-Z3-5 Dřevěná slacklajna

Jak už všichni víme z hodin matematiky, trojúhelník, jehož nějaká strana je delší než součet všech ostatních, nejde nakreslit. A stejně tak to platí i pro mnohoúhelník. Kdyby Kevin se Sárou položili všechna prkna kromě nejdelšího za sebe do jedné linie a tato linie by byla kratší než ono nejdelší prkno, nemohou si okruh vůbec vyrobit.

Základem řešení bude tedy tato myšlenka. Délky prken si nejprve setřídíme, ideálně od největšího po nejmenší. Součet všech délek označíme S, délku nejdelšího prkna D.

Pokud je:

- $S - D > D$, okruh lze postavit ze všech prken, jejichž délka je D nebo menší.
- $S - D \leq D$, tak nejdelší prkno nelze použít. Od S odečteme D a celé porovnání provedeme znova, akorát D již bude délka dalšího nejdelšího prkna v pořadí.

Na konci stačí zkontrolovat, že nám zbyla alespoň tři prkna, ze kterých může Kevin se Sárou okruh postavit.

Časová složitost bude $\mathcal{O}(N \log N)$, kde N je počet prken. Projití setříženého seznamu délek prken bude lineární, jelikož pro každé prkno provedeme pouze konstantně mnoho operací. Ovšem setřízení prken bude trvat $\mathcal{O}(N \log N)$.

Program (Python 3): `http://ksp.mff.cuni.cz/viz/27-Z3-5.py`

Katka Zákravská

27-Z3-6 Red Bull dává křídla

Nedostali jsme příliš mnoho řešení, což nás mrzí. Pojďme si ukázat, že tato úloha nebyla tak těžká, jak se vám možná zdála.

Jelikož procházíme stavovým prostorem (což neznamená nic jiného, než že máme hrací plochu a na každém políčku můžeme být buď jako král, nebo jako kůň), tak si musíme rozmyslet, jak jej budeme reprezentovat.

Pro tuto konkrétní úlohu se nám zdá vhodné mít dvě dvourozměrná pole, která budou odpovídat velikosti hrací plochy, tedy velikosti $M \times N$.

Jedno pole bude pro místa, kam jsme došli jako král, a druhé bude pro místa, kam jsme došli jako kůň.

Vždy z každého dosaženého políčka zkusíme jít do všech políček ve druhém poli, na která se daná figurka může dostat a ještě jsme tam nebyli. (Ve druhém poli, protože se tahy koněm a králem střídají.)

To, jestli jsme v některém stavu už byli, si budeme značit číslem udávajícím, v kolikátém kroku jsme tam došli. Což nám pomůže při debugování, neboť víme, kam nám to kdy skočilo. Ale taky podle toho dokážeme zrekonstruovat cestu.

Pro nalezení cesty pak stačí jít z cíle do startu po stavech, která mají právě o jedna menší číslo, než ve kterém jsme. Před výpisem tuto posloupnost musíme ještě otočit. Rozmyslete si, že v tomto případě nelze udělat trik, že bychom hledali cestu z cíle do startu, a tím se vyhnuli otáčení posloupnosti.

Posledním krokem je si rozmyslet, jak rychle zjistit, kam jsme již skočili. Nejlepší bude si to pamatovat v nějaké struktuře, do které budeme umět v konstantním čase vkládat prvky na konec a vyjmout první prvek. Takové struktuře se říká fronta.

Nyní již jen zbývá zanalyzovat, jak dobrý algoritmus jsme vymysleli. Projití všech stavů nám bude trvat $\mathcal{O}(N \cdot M)$, protože právě tolik je stavů a v každém stavu děláme konstantní množství operací, a stejně tolik paměti budeme potřebovat.

Závěrem ještě dodáme, že použít jen jedno společné pole pro kroky koněm i králem nestačí, protože na tom, jakou figurkou jsme se do které pozice dostali, závisí i následné možné pohyby z tohoto políčka dál. A pokud políčka zvládneme dosáhnout ve dvou různých tazích koněm i králem, tak je nutné zkoumat obě varianty. Zkuste si to třeba na příkladu Kevina na pozici 5×5 a cíle na pozici 2×2. Na takovéto políčko se dá dostat za 3 tahy, ale pokud bychom neuvažovali obě možnosti (pokud bychom použili jen jedno společné pole dosažitelnosti), tak nám algoritmus bude říkat, že se tam dostaneme za 4 tahy.

Program (Python 3): `http://ksp.mff.cuni.cz/viz/27-Z3-6.py`

Vojta Sejkora

27-Z4-1 Záhada Pražského orloje

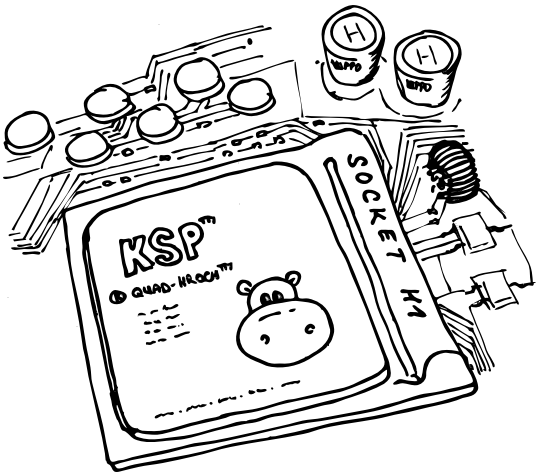
Napsání programu pro tuto úlohu nebylo vůbec těžké, jak se můžete přesvědčit ve vzorovém řešení, nicméně bylo nutné uvědomit si jednu základní myšlenku.

Problém si můžeme představit tak, že naše dvě kolečka namočíme do barvy a uděláme s nimi stopy na papíře. Takto nám vzniknou dvě úsečky dlouhé jako obvody jednotlivých ozubených kol (délka obvodu, protože jsme s každým kolečkem otočili dokola). V této představě by náš problém byl jako vyskládání několika úseček délky obvodu prvního kolečka do jedné čáry a úseček délky obvodu druhého kolečka do druhé tak, aby obě čáry měly stejnou délku. Po chvíli bádání můžeme nahlédnout, že tento problém vyřeší nejmenší společný násobek daných dvou délek.

I řešení našeho původního problému je nejmenší společný násobek počtu zubů našich koleček. To proto, že se obě kolečka otočí o stejný počet zubů za jednotku času a pokaždé, kdy se kolečka potkají, se každé z nich otočí o celý počet otáček. Tedy když se potkají, tak první kolečko udělalo k svých otoček, druhé pak ℓ svých otoček. Pro předsavu třeba předpokládejme otočení o jeden zub za jednu sekundu. Každý čas setkání bude násobek počtu otočení a počtu zubů (tj. perioda) příslušného kolečka a toto pro obě kolečka bude stejné, tedy:

$$k \cdot \text{počet zubů prvního} = \ell \cdot \text{počet zubů druhého.}$$

Čas jejich prvního setkání nastane pro nejmenší možná ℓ a k a bude to nejmenší společný násobek obou period.



KSP-Z

řešení

Jak spočteme nejmenší společný násobek? Pravděpodobně všichni známe rozklad na součin prvočísel, což ale v počítači není tak jednoduché a existuje mnohem rychlejší cesta. Ta vede přes Euklidův algoritmus zjištění největšího společného dělitele a vztah největšího společného dělitele (nsd) a nejmenšího společného násobku (nsn):

$$x \cdot y = \mathrm{nsn}(x, y) \cdot \mathrm{nsd}(x, y).$$

Tento vztah si můžete rozmyslet například právě díky zmíněnému prvočíselnému rozkladu.

Euklidův algoritmus funguje tak, že opakovaně odečítá od většího z čísel to menší, než se obě vyrovnají. Proč přesně funguje a jak ho zrychlit, se dozvíte v naší kuchařce o teorii čísel.[10] Zde prozradíme pouze to, že jeho časová složitost je $\mathcal{O}(x + y)$ a u zrychlené verze z kuchařky dokonce $\mathcal{O}(\log \min(x, y))$.

Celé řešení tedy načte N dvojic, pro každou dvojici spočítá nejmenší společný násobek a vypíše ho na výstup. Euklidův algoritmus tedy spouštíme N-krát, pro každou dvojici jednou. Paměti zabereme pouze konstantně, protože můžeme zpracovávat dotazy postupně, aniž bychom si je nejprve všechny načetli.

Program (Python 3): `http://ksp.mff.cuni.cz/viz/27-Z4-1.py`

Martin Šerý & Štěpán Hojdar

27-Z4-2 Unavení u oken

Vida, rozsvícené okno. Jak zjistíme, do jaké souvislé svítící oblasti patří? Začneme rozsvíceným oknem a prozkoumáme jeho sousedy. Pokud jsou také rozsvícení, započítáme je do oblasti a prozkoumáme i jejich sousedy. Jsou-li rovněž rozsvícení, zase je přidáme do oblasti a tak dále. Nechceme ovšem jedno okno započítat vícekrát, takže ledva nějaké započítáme, hned ho zhasneme.

Jak ale zařídit, abychom se v sousedech sousedů sousedů (atd.) neztratili? Pořídíme si *frontu,* v níž budeme skladovat všechna políčka, která jsme objevili, ale dosud jsme nezpracovali jejich sousedy. (Pokud se s frontou ještě neznáte, můžete si představit, že je to nějaké pole, ve kterém se nové prvky přidávají na konec a staré se odebírají ze začátku.)

Na počátku bude ve frontě jen to první rozsvícené okno. V každém kroku pak odebereme jedno okno z fronty a podíváme se na jeho sousedy. Je-li kterýkoliv z nich rozsvícený, zhasneme ho a přidáme do fronty. To opakujeme, dokud se fronta nevyprázdní.

Snadno si všimneme, že takto projdeme celou svítící oblast a bude nám to trvat řádově tolik času, kolik oken v oblasti leží. (Mimochodem, tomuto postupu se říká *prohledávání do šířky* a hodí se na ledacos dalšího.)

[10] `http://ksp.mff.cuni.cz/viz/kucharky/teorie-cisel`

Jednu oblast tedy najít umíme. Zbývá domyslet, jak najít všechny. Budeme postupně procházet všechna okna, a kdykoliv najdeme nějaké rozsvícené, prohledáme a zhasneme celou jeho oblast. Pak pokračujeme v hledání dalšího rozsvíceného okna atd.

Kolik času nám to celkem zabere? Hledání rozsvícených oken samo o sobě sáhne na každé okno právě jednou. Prohledávání všech oblastí dohromady sáhnou na každé okno nejvýše jednou (jedno okno nemůže ležet ve více oblastech současně). Náš program tedy má lineární časovou složitost s počtem všech oken.

Program (Python 3): `http://ksp.mff.cuni.cz/viz/27-Z4-2.py`

Program (C): `http://ksp.mff.cuni.cz/viz/27-Z4-2.c`

Martin Mareš & Jakub Maroušek

27-Z4-3 Běžkaři v Praze!

Na tuto úlohu asi není žádný chytřejší postup, stačí si celou situaci odsimulovat. Nebudeme si ale posouvat figurkami závodníků po virtuálním okruhu, půjdeme na to chytřeji.

Mějme dva závodníky, kteří běží za sebou, a jejich rychlosti v_1 a v_2. Jediný spolehlivý způsob, jak poznat, jestli se na trati potkají, je spočítat časy, kdy by doběhli nezávisle (prozatím jako desetinná čísla v hodinách), a ty porovnat. Rovnice časů vypadají takto:

$$t_1 = \frac{S}{v_1} \qquad t_2 = \frac{1}{60} + \frac{S}{v_2}.$$

Budeme si udržovat seznam závodníků, u kterých neznáme čas. Jak toto udělat pohodlně, najdete ve zdrojáku v Pythonu, jak to udělat rychle ve zdrojáku v C++. Vždy vezmeme prvního, který určitě závod dokončí, a vyhodíme jej ze seznamu. Následně přeskočíme všechny, co prvního doběhnou ($t_1 \geq t_2$, nezapomeňte ale na různou velikost zpoždění na startu). Podobně projdeme celý seznam, a celou proceduru opakujeme dokud nezměříme všechny závodníky.

KSP-Z

řešení

V poznámce jsme psali, že se dá vyhnout počítání s desetinnými čísly – přesnost počítání s nimi není nekonečná, a pokud by se dva závodníci teoreticky potkali až těsně před cílovou páskou, mohlo by na přesnosti záležet. Také s nimi počítače počítají zpravidla o něco pomaleji. Pokud to lze, je dobré se jim vyhnout. Pojďme se podívat, jak na to.

Napišme si nerovnici pro dva závodníky, druhého zpožděného o m minut:

$$t_1 = \frac{S}{v_1} \;\geq\; \frac{m}{60} + \frac{S}{v_2} = t_2.$$

A trochu si ji upravme vynásobením $60 \cdot v_1 v_2$ (kladné číslo):

$$60 \cdot S\, v_2 \geq m\, v_1 v_2 + 60 \cdot S\, v_1.$$

A protože pouze násobíme a všechna čísla jsou celá, stačí nám už počítat s celými čísly. Pokud je tato nerovnice splněna, druhý závodník prvního doběhl.

Program (Python 3): `http://ksp.mff.cuni.cz/viz/27-Z4-3.py`

Program (C++11): `http://ksp.mff.cuni.cz/viz/27-Z4-3.cpp`

Ondra Hlavatý

27-Z4-4 Koňské skoky

Pro začátek zkusme vyřešit stejnou úlohu, akorát jenom s jedním koněm. Tedy dostaneme políčko, kde kůň začíná, a políčko, kam má doskákat. Naším úkolem je potom zjistit, na kolik skoků se tam dokáže dostat.

To není nic těžkého – použijeme jednoduchý algoritmus prohledávání do šířky, který je podrobně vysvětlen v naší grafové kuchařce.[11]

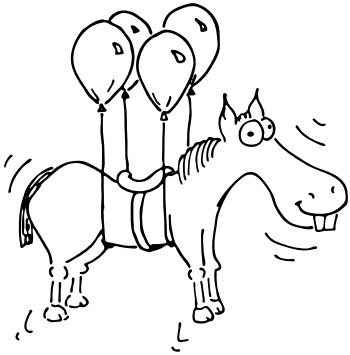
Jenže jak ho použít, když algoritmus povídá o grafech, ale my tu máme šachovnici? Prostě si podle šachovnice graf vytvoříme. Vrcholy grafu budou políčka šachovnice a hrana povede mezi dvěma políčky právě tehdy, když z jednoho na druhé může skočit kůň.

Prohledávání do šířky v našem případě dělá to, že nejprve zapíše jedničku do všech políček, kam se kůň může dostat jedním skokem ze začátečního políčka, potom dvojku na nenavštívená políčka, kam se může dostat z políčka s jedničkou, a tak dále, dokud nemáme na každém políčku napsáno, na kolik skoků se tam dokážeme dostat. Na konci si už jenom přečteme, v kolikátém kroku jsme byli v cíli.

Prohledávání do šířky skončí v čase $\mathcal{O}(N^2)$, protože šachovnice má N^2 políček a přibližně $4N^2$ hran mezi nimi.

[11] `http://ksp.mff.cuni.cz/viz/kucharky/grafy`

Tak to by bylo. V zadané úloze je ale problém výrazně složitější v tom, že není jasné, který kůň má jít do kterého cíle. S tím se vypořádáme tak, že prostě vyzkoušíme všechny možnosti.



Zjistíme pro každou dvojici startovního a cílového políčka, jak dlouhá je mezi nimi cesta. Užitečné je, že prohledávání do šířky nám pro nějaké startovní políčko řekne vzdálenosti do všech ostatních políček, takže nám bude stačit pustit ho pětkrát (pro každé startovní políčko) a pokaždé si poznamenat vzdálenosti do všech pěti možných cílů.

Máme tedy tabulku 5×5 se vzdálenostmi mezi starty a cíli. Teď už stačí jenom vyzkoušet všechny možnosti, jak je spárovat. Možností je $5 \cdot 4 \cdot 3 \cdot 2 \cdot 1 = 120$. Tu nejlepší by šlo najít i s tužkou a papírem. Jako správní programátoři jsme ale líní, a tak to naprogramujeme.

Můžeme si třeba vygenerovat seznam všech permutací čísel od 1 do 5, neboli seznam způsobů, jak tato čísla seřadit. Každá permutace potom bude popisovat, jak přiřazovat starty k cílům. Pak už stačí jenom pro každou permutaci sečíst příslušné hodnoty z tabulky a najít nejnižší součet.

Ukážeme rekurzivní algoritmus, jak všechny permutace čísel od 1 do 5 vygenerovat: Vytvoříme funkci p, která dostane prvních několik pozic permutace pevně zadaných a vypíše všechny možnosti, jak pokračovat. Pokud nám zbyla jediná pozice, kam něco dát, a tedy i jediná hodnota, kterou tam dát, tak tuto jedinou permutaci zaznamenáme. Jinak vyzkoušíme všechny možnosti, jak pokračovat na další pozici, a zbytek si vždy objednáme pomocí naší funkce p.

Stačí zavolat funkci p a nedat jí žádné omezení na to, co má být na začátku, a dostaneme všechny permutace.

Dodejme ještě, že se můžeme obejít bez generování permutací tak, že podobným rekurzivním způsobem budeme hledat rovnou minimální součet délek cest.

Program (C): `http://ksp.mff.cuni.cz/viz/27-Z4-4.c`

Martin Španěl

KSP-Z

27-Z4-5 Poškolní trest

řešení

Nejprve se zamyslíme, jak bychom postupovali se skutečnou hromádkou papírů, a teprve poté toto řešení převedeme do řeči počítačů.

Všimněme si, že vytvářená hromádka papírů musí po celou dobu práce být správně uspořádaná. Tím myslíme to, že každý papír v ní má číslo o jedna větší než předchozí. Pokud by to neplatilo, pak v hromádce jsou dva papíry v sestupném pořadí (větší číslo před menším) nebo „díra“ (např. sousedí dvojice 1, 5). Ani jednu z těchto chyb už přidáváním dalších papírů nemůžeme odstranit, tedy ani na konci taková hromádka nebude správně uspořádaná.

Z toho už je vidět, jak musíme postupovat při třídění. Papír můžeme přidat na konec výstupní hromádky pouze tehdy, když má číslo o jedna větší než dosavadní koncový papír. Podobně pro přidávání na začátek, kde naopak musí být přidávané číslo o jedna menší. Zároveň si snadno rozmyslíte, že takovýmto přidáním nemůžeme nic pokazit. Z toho už je vidět výsledný postup: v každém kroku se podíváme na oba krajní papíry vstupní hromádky. Pokud některý z nich lze přidat na výstupní hromádku v souladu s popsanými pravidly, učiníme tak. Pokud je možné přidat oba, vybereme si libovolný (ten druhý můžeme přidat v příštím kroku).

Pokud se někdy dostaneme do situace, kdy ani jeden ze dvou krajních papírů nejde přidat na výstupní hromádku, pak setřídění není možné.

To vše má ovšem jeden háček. Popsali jsme si, jak postupovat, když už na výstupní hromádce nějaké papíry jsou. Ale jak začít? Například posloupnost papírů $3, 1, 2, 4, 5$ lze setřídit, pokud jako první na výstupní hromádku položíme číslo 5, ale nejde, pokud začneme trojkou (v takovém případě se zastavíme hned ve druhém kroku). Nejjednodušším řešením je prostě vyzkoušet obě možnosti. Nejprve začít prvním papírem, a pokud se to nepovede, znovu si vzít původní posloupnost a zkusit to z opačného konce. Tím se celý postup zpomalí jen dvakrát, což je ve světě algoritmů obvykle zanedbatelné.

Nyní se zamysleme, jak z toho všeho vyrobit program. Začněme tím, jak reprezentovat vstupní a výstupní hromádku. Vstupní si určitě musíme na začátku celou načíst (např. do pole), abychom se mohli dívat na oba konce. Poté z ní chceme odebírat papíry. To bychom mohli dělat tak, že prostě budeme z pole mazat prvky (např. v Pythonu příkazem `del pole[0]`), ale to je pomalé. Jediný způsob, jak smazat prvek ze začátku pole, je posunout všechny ostatní o jednu pozici níž. K tomu je potřeba řádově tolik operací, jako délka pole, tedy $\mathcal{O}(n)$, což je zbytečně moc.

Snadno jde toho samého dosáhnout v konstantním čase, a to hned dvěma způsoby. Prvním je použít místo pole strukturu zvanou spojový seznam, o které si můžete přečíst v naší základní kuchařce.[12] Z (obousměrného) spojového sezna-

[12] `http://ksp.mff.cuni.cz/viz/kucharky/uvodni`

mu pak dokážeme odstraňovat prvky z libovolného konce v konstantním čase. Například v Pythonu můžete použít implementaci spojových seznamů ve třídě collections.deque.[13]

Pokud byste si je ale museli implementovat sami (např. v C), je to spousta práce, a v takovém případě je jednodušší druhé řešení: vůbec z pole nic nemazat. Namísto toho si budeme jen pamatovat, kterou pozici v poli aktuálně považujeme za první a poslední, a vždy pracovat pouze s touto částí pole. Ostatní prvky tam pořád budou, ale program je bude ignorovat. To je trik, který se při programování často hodí: někdy není třeba doopravdy měnit nějaká data, stačí změnit způsob, jakým se na ně díváme. Tuto variantu najdete i ve vzorovém programu.

Reprezentace výstupní hromádky je ještě jednodušší. Všimněme si, že nikdy nepracujeme s žádným jiným než prvním či posledním jejím prvkem. Stačí nám proto pamatovat si místo celé hromádky dvojici čísel: aktuální první a poslední prvek. Pokud chceme přidat papír na některý konec, prostě příslušné krajní číslo přepíšeme. To původní už stejně nikdy nebudeme potřebovat.

Každý krok třídění zvládneme v konstantním čase, tedy celý algoritmus bude mít lineární časovou složitost.

Program (C): `http://ksp.mff.cuni.cz/viz/27-Z4-5.c`

Filip Štědronský

27-Z4-6 Příprava grilovačky

Cieľom tejto úlohy bolo usporiadať zadané činnosti podľa stanoveného pravidla. Každá činnosť, ktorá závisí od inej, sa musí vykonať až potom, čo sa vykonajú činnosti, na ktorých závisí. Pre jednoduchosť si jednotlivé činnosti a závislosti zakreslíme do grafu. Vrcholy grafu budú tvoriť činnosti a hrany budú šípky určujúce závislosť. Keďže šípky, ktoré vychádzajú od činností majú definovaný smer (podľa závislosti medzi činnosťami), tak aj hrany v grafe budú orientované v rovnakom smere. Teda, ak činnosť A musí byť vykonaná pred činnosťou B, potom v grafe bude existovať orientovaná hrana z B do A.

Na začiatok jednoduché pozorovania. Graf, ktorý tvorí závislosť činností, nemusí byť súvislý. To nastane vtedy, ak budeme mať nejakú množinu činností, ktorá nezávisí na ostatných a ani žiadna iná činnosť nezávisí na nej. Ak v grafe bude existovať orientovaný cyklus, znamená to, že činnosti obsahujú kruhovú závislosť (napr. A závisí na B, B na C a C na A). A takúto kruhovú závislosť nie je možné žiadnym spôsobom usporiadať.

Na riešenie tejto úlohy použijeme jednoduchý algoritmus na prehľadávanie grafu do hĺbky. Počas jeho behu budeme postupne tvoriť výsledné poradie činností. A to tak, že do čiastočne zostaveného zoznamu budeme činnosti pridávať

[13] `http://docs.python.org/3/library/collections.html#collections.deque`

nakoniec. Začneme s prázdnym zoznamom činností. Vrchol, ktorý sa už bude nachádzať vo výslednom poradí, si označíme ako „spracovaný“. Ďalej si pre každý vrchol budeme počas behu algoritmu pamätať, či sme ho už niekedy navštívili. Naviac si k nemu poznamenáme, z ktorého vrcholu sme sa doňho prvýkrát dostali.

Graf začneme prehľadávať z ľubovoľného vrcholu. Pozrieme sa na orientované hrany, ktoré z neho vedú k ďalším vrcholom. Vyberieme sa po ľubovoľnej hrane, ktorá nevedie do už spracovaného vrcholu.

Ak sme sa ocitli vo vrchole, v ktorom sme už raz boli, znamená to, že sme prešli po orientovaných hranách, ktoré tvoria kružnicu z nespracovaných vrcholov. Teda takýto zoznam činností tvorí kruhovú závislosť a tú usporiadať nejde. Prehľadávanie v tomto prípade ukončíme a oznámime neexistenciu riešenia.

Ak sme sa dostali do vrcholu, v ktorom sme ešte neboli, pozrieme sa opäť na susedov, do ktorých sa vieme dostať. Opäť si niektorého vyberieme a takto pokračujeme, až kým sa nedostaneme do vrcholu, z ktorého sa nedá pokračovať ďalej (a to už preto, že z neho nevedie žiadna hrana alebo preto, že všetci susedia sú už označení ako spracovaní). V takomto prípade sme sa dostali do vrcholu reprezentujúci činnosť, ktorá nemá žiadne nespracované závislosti. Keby mal nejaké nespracované závislosti, tak by musela existovať hrana z aktuálneho vrcholu niekam. Avšak neexistuje, a teda sme našli činnosť, ktorá bude mať určite splnené všetky závislosti (ak nejaké má) v už zostavenom výstupnom poradí činností.

Túto činnosť pridáme do aktuálne zostavujúceho sa poradia nakoniec za všetky už spracované činnosti. Vrchol, ktorým je činnosť reprezentovaná, sa v tomto momente stane spracovaným. Potom sa vrátime späť do vrcholu, z ktorého sme sa prvýkrát dostali do aktuálneho (ideme proti smeru orientácie hrany), a pokračujeme ďalej v prehľadávaní z toho vrcholu.

Ak skončíme prehľadávanie, tak graf buď bude celý prejdený, alebo v ňom nájdeme cyklus (a usporiadanie neexistuje), alebo ostanú v grafe ešte neprehľadané vrcholy. To, že v grafe ostanú neprehľadané vrcholy, môže nastať aj keď je graf súvislý. Napr. vtedy, ak si zvolíme začiatočný vrchol, z ktorého nevedú žiadne orientované hrany.

V takomto prípade, keď sme ešte nenavštívili všetky vrcholy (a teda výstupné poradie ešte nie je úplné), musíme spustiť prehľadávanie opäť z ďalšieho ešte nenavštíveného vrcholu. Máme stále zaručené, že činnosti, ktoré sú už vo výslednom poradí, nebudú závislé na nespracovaných činnostiach (nenavštívených vrcholoch). Ináč by viedla z nich hrana a pri prehľadávaní by sme ju spracovali.

Prehľadávanie budeme spúšťať vždy z niektorého nenavštíveného vrcholu, až kým nebudú všetky vrcholy navštívené a spracované. Na konci, keď už bude každý vrchol označený ako navštívený, bude zároveň každá činnosť vo výstupnom poradí, a teda už budeme mať hotové výsledné poradie.

Keďže vrcholov je konečný počet a každý vrchol navštívime maximálne toľkokrát, koľko má susedov (z každého suseda sa doňho späť vrátime), a naviac každou hranou prejdeme maximálne dvakrát (raz v smere orientácie a raz proti smeru, keď sa budeme vracať), tak sa tento algoritmus určite raz zastaví. Z toho aj rovno vypozorujeme, že časová zložitosť algoritmu bude lineárna od počtu hrán a keďže navštívime každý vrchol, tak aj lineárna od počtu vrcholov.

Pre jednoduchosť si zadaný graf činností a ich závislostí budeme reprezentovať nezápornými celými číslami. V pamäti budeme mať pre každý vrchol uložený zoznam jeho susedov. Okrem toho si počas behu algoritmu budeme musieť pamätať, či sme daný vrchol už navštívili (a odkiaľ prvýkrát) a či už je vo výslednom poradí spracovaný. Čiže pamäťová zložitosť bude lineárna od počtu vrcholov a hrán. Výstup algoritmu bude tvoriť usporiadanie jednotlivých vrcholov, a teda nám pamäťovú zložitosť nezmení.

Ná záver malá poznámka. Problém popísaný v tejto úlohe sa označuje aj ako topologické usporiadanie orientovaného grafu a môžete sa o ňom dočítať aj v našej grafovej kuchárke.[14] V nej nájdete aj ďalší alternatívny algoritmus, ktorý rieši tento problém taktiež v lineárnom čase od veľkosti zadaného grafu.

Program (Python 3): `http://ksp.mff.cuni.cz/viz/27-Z4-6.py`

Pali Rohár

[14] `http://ksp.mff.cuni.cz/viz/kucharky/grafy`

Pořadí řešitelů KSP-Z

Pořadí	*Jméno*	*Škola*	*Ročník*	*Úloh*	*Bodů*
0.				24	264.0
1.	Jakub Pelc	G UherBrod	1	24	250.8
2.	Jan Kaifer	GČesBrod	−1	24	233.0
3.	Jakub Matěna	GČeskoliPH	3	24	229.0
4.	Lukáš Červený	G Trutnov	1	24	215.5
5.	Jiří Štěpanovský	G MNám Třb	3	21	206.0
6.	Martin Scheubrein	G MNám Třb	3	21	203.0
7.	Vojtěch Lukeš	GPikaPL	3	19	191.0
8.	Miroslav Hrabal	GTomkovaOL	1	19	177.0
9.	Lukáš Mičan	GČeskáČB	1	19	168.3
10.	Jiří Moravčík	GUHradiště	1	16	160.0
11.	Jakub Jirkal	GJungmanLT	0	19	158.5
12.	Lukáš Vlček	GMikulášPL	1	16	145.0
13.	Michal Töpfer	G DrJPekMB	2	15	137.0
14.	Roman Beňo	GJHroncaBA	2	14	131.0
15.	Tereza Kotěšovcová	GKlatovy	4	16	128.8
16.	Karolína Kuchyňová	GMLerchaBO	4	13	128.0
17.	David Ucháč	eduSOŠ PA	2	14	124.5
18.	Tomáš Troján	G Cheb	−1	14	117.0
19.	Jakub Rozlivek	GPikaPL	3	13	112.0
20.	Matěj Fencl	GOA Chodov	1	14	110.5
21.	Tomáš Terem	GTajBanBys	3	12	107.0
22.	Zdeněk Pavlátka	GMikulášPL	3	11	106.0
23.	Lukáš Fruněk	GLesníZlín	2	11	100.0
24.	Pavel Souček	G Nymburk	3	10	98.0
25.	Daniel Nigrin	GÚstavníPH	2	11	92.0
26.	Jan Mráz	G Holice	1	12	84.0
27.	Matúš Maďar	GHorMichal	3	10	83.8
28.	David Žáček	GZborovPH	2	12	82.0
29.	Michal Převrátil	GKlatovy	2	8	80.0
30.	Tomáš Chvosta	GPří	4	9	76.3
31.	Nhat Minh Dinh Huy	GKadaň	2	8	72.0
32.	Jan Burda	G Holice	0	9	69.5
33.	Jakub Neruda	GTNovákBO	4	7	69.0
34.	Pavel Turinský	G Brandýs	2	7	63.8
35.	Zuzana Šimečková	GČeskáČB	4	11	58.0
36.–37.	David Bělíček	GSOŠ Podb	3	6	57.0
	Michaela Štolová	G Sokolov	3	9	57.0
38.	Daniel Pluskal	G BO-Řeč	1	7	56.0

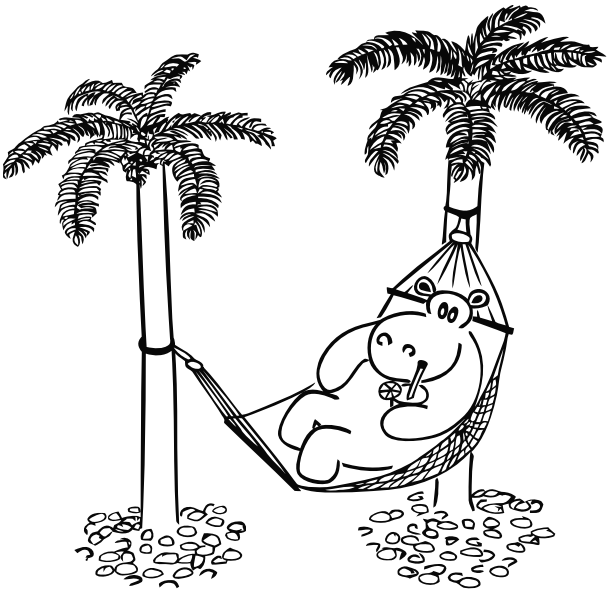
Pořadí řešitelů KSP-Z

39.	Zuzana Drázdová	GČeskáČB	4	8	52.0
40.–41.	David Nápravník	GLitoměřPH	2	6	50.0
	Ondřej Švanda	G BO-Řeč	4	7	50.0
42.	Václav Fabík	ZŠKřídloBO	0	5	48.0
43.	Josef Vávra	SJec	4	6	47.4
44.–45.	Petr Klanica	GJarošeBO	2	5	45.0
	Zoltán Onódy	SPŠE NZám	4	5	45.0
46.	Jonáš Přibyl	ZŠ ČBrod	0	5	44.0
47.	Michaela Svatošová	GKepleraPH	1	6	43.0
48.	Ondřej Měkota	SPŠMasarLI	3	6	42.8
49.–54.	Patrik Bak	G Sobrance	4	4	40.0
	Tat Dat Duong	G Wicht	2	4	40.0
	Jakub Lukeš	GNAlejíPH	2	4	40.0
	Jiří Sejkora	GVoděraPH	3	4	40.0
	Jan Soukup	GKlatovy	4	4	40.0
	Jan Václavek	GUnOrl	3	4	40.0
55.	Janek Hlavatý	ZŠ DukelČB	−4	5	39.0
56.	David Tvrdý	GHeyrovPH	2	4	38.0
57.	Jan Vozár	G UherBrod	1	6	37.5
58.	Roman Ondráček	GBoskovice	1	13	37.0
59.	Benedikt Žour	G UherBrod	0	6	36.0
60.–61.	Milan Kubala	GTajBanBys	3	4	32.0
	Andrej Čermák	G JF Šaľa	1	9	32.0
62.–64.	Lukáš Holeczy	GTep	3	4	28.0
	Ivana Krumlová	GJarošeBO	2	3	28.0
	Samuel Schneider	GTajBanBys	3	3	28.0
65.	Jan Neumann	GNAlejíPH	1	4	27.0
66.	Michael Kozel	GZborovPH	1	5	25.0
67.	Dominika Tanglová	G Nymburk	2	3	22.0
68.	Jan Sliacky	G Benesov	4	3	21.0
69.–70.	Ľuboš Kolumber	SpojŠ Popr	3	8	20.0
	Victoria María Nájares Romero	GZborovPH	1	3	20.0
71.	Vít Gadurek	Neuvedená	0	5	19.0
72.–76.	Mirolsav Březík	GLesníZlín	0	2	18.0
	Michal Nekvinda	BiGyBBHK	4	2	18.0
	Alexej Popovič	SlovanGOL	3	2	18.0
	Vojta Staněk	PORGPha	1	2	18.0
	Petr Šíma	GKlatovy	1	2	18.0
77.	Martin Zima	G Holice	1	3	17.0
78.	Vojtěch Pejša	G Kolín	4	2	15.0
79.	Filip Priečinský	SG Žilina	2	9	13.5
80.	Markéta Machalová	G Wicht	2	9	12.0

81.	Matěj Hudec	CírkG Plzeň	3	3	10.4
82.–87.	Michaela Bačová	G UherBrod	4	1	10.0
	Antonín Bruštík	G UherBrod	4	1	10.0
	David Dvořáček	G UherBrod	4	1	10.0
	Viktor Kovařík	G UherBrod	4	1	10.0
	Martin Picek	GJirsíkaČB	0	2	10.0
	Jakub Šmahovský	G Pezinok	3	2	10.0
88.	Eva Matoušková	G Sokolov	4	2	9.0
89.–92.	Matej Hockicko	TAPoprad	1	3	8.0
	Ján Pavlus	GTNovákBO	2	1	8.0
	Martin Sklenár	GTajBanBys	3	1	8.0
	Petr Zelina	GJarošeBO	2	1	8.0
93.	Roman Solař	GJarošeBO	3	2	5.0
94.	Lukáš Kostka	GaHŠ Pha	0	1	2.0

KSP

Hlavní kategorie KSP



Zadání úloh KSP

První série

Tento rok jsme se rozhodli vám v každé sérii přinášet nějaký zajímavý příběh točící se okolo určité zajímavé programátorské chyby. Takové chyby dělá občas každý z nás, leč přesto někdy přerostou v příběh hodný zapamatování. V jednotlivých dílech příběhu se budeme pokoušet držet skutečných událostí, ale dovolíme si i jistou uměleckou licenci.

V prvním dílu se můžete začíst do příběhu, u kterého se nám nepodařilo zjistit, kde se přesně odehrál. Některé zdroje vsazují události tohoto příběhu do vojenského cvičení u Mrtvého moře a některé do Války v Zálivu (konflikt mezi Irákem a koalicí západních zemí v čele s USA v roce 1991), my jsme se rozhodli držet se verze z Perského zálivu.

Náš příběh začíná na palubě letadlové lodě USS Dwight D. Eisenhower *přezdívané posádkou familiárně* Ike*...*

⋆⋆⋆

Poručík Stromboli rázoval chodbou k briefingové místnosti perutě. Ike *byla dlouhá loď, na délku přes 300 metrů, a on dostal kajutu zrovna na opačném konci. Vyhnul se probíhající četě námořních pěšáků a během hlášení palubního rozhlasu o příletu zásobovacího letu se protáhl průchodem do briefingové místnosti.*

Už od rána byla loď napjatá tím, kdy se konečně zapojí do probíhajících operací. Včera v noci vedly některé lodě operačního svazu masivní raketový útok na irácká radarová postavení a proslýchalo se, že dnes v noci se dostane řada i na ně.

A piloti už se nemohli dočkat, aspoň tak působil šum v místnosti, když Stromboli vešel a pokusil se sednout si na nějaké volné místo. Každý chtěl vidět co nejlépe na hlavní bojový plán, ale zároveň mezi piloty existoval určitý kodex, kterého se drželi i při uvolňování místa k sezení.

27-1-1 Zasedací pořádek — 8 bodů

Piloti v briefingové místnosti si postupně sedají na volné židle. Zasedací pořádek si můžeme představit tak, že máme nekonečně dlouhou řadu židlí (židlí je mnohem více, než dorazí pilotů) a každý nový pilot se nejdříve pokusí posadit na prostřední židli.

Pokud je židle, na kterou si chce pilot sednout, volná, je vše v pořádku, pilot se posadí a čeká na briefing. Pokud je však židle již obsazená jiným pilotem, tak, než aby se hádali, sedne si jeden z pilotů na židli o jedno místo vlevo a druhý na židli o jedno místo vpravo (původní židli tak nechají volnou).

Pokud by se při tomto rozsazování náhodou situace opakovala (pilot by se opět chtěl posadit na židli, kde už někdo sedí), bude se postup opakovat tak dlouho, dokud na každé židli nebude sedět maximálně jeden pilot.

Chceme po vás dvě věci:

a) Jak bude řada židlí vypadat po příchodu N pilotů?

b) Dostanete zapsaný nějaký zasedací pořádek pilotů (obsazené a neobsazené židle). Rozhodněte, zdali mohl vzniknout tímto postupem.

Konečně se všichni posadili, do místnosti vešel velitel a začal briefing. Vypadá to dobře, dneska se konečně odlepí od letové paluby, zasnil se Stromboli, a tak si skoro nevšiml na něj mířené otázky.

„Tak Stromboli, přestaňte lelkovat a poslouchejte!“ napomenul ho velitel, „Říkal jsem, že dneska odpoledne provedete s Thompsonem průzkumný let v nízké výšce. Přesné pokyny a letový plán obdržíte během několika hodin, zatím se připravte. Technici vám zrovna na vašeho ptáčka montují průzkumnou výbavu.“

„Ano pane!“ odpověděl spěšně Stromboli a s úsměvem mrknul na svého navigátora Thompsona.

Schůze ještě chvíli pokračovala, než se rozdaly úkoly pro všechny piloty, a pak Stromboli v závěsu se svým navigátorem vyrazil směrem k hangárové palubě. Chtěl si ještě před akcí promluvit s vrchním zbrojmistrem a vybrat si vybavení.

Ve chvíli, kdy dorazil do zbrojnice, trochu se zděsil. Z vedlejšího zbrojního skladu se totiž ozýval děsivý lomoz, a tak tam opatrně nakoukl – vrchní zbrojmistr stál uprostřed místnosti a dirigoval sundavání palet se zbraněmi z vysokých polic.

27-1-2 Zbrojní sklad — 9 bodů

Vrchní zbrojmistr na letadlové lodi potřebuje vyndat ze skladu několik palet s výzbrojí. Sklad je ale zaskládaný do veliké výšky a manévrování s neohrabanými vysokozdvižnými vozíky je v něm celkem nebezpečné, aspoň do doby, než se část věcí vyndá.

Ve skladu operují dva různé vysokozdvižné vozíky (určené pro palety dvou velikostí) a bezpečností předpisy dovolují na začátku sundavat pouze palety uložené na policích v maximální výšce h_0. Na vstupu dostanete popis všech N palet ve skladu, paleta i je velikosti v_i (velká nebo malá), je uložena ve výšce h_i a má nebezpečnost x_i.

Vozíky se musí v sundavání palet střídat (velká paleta, malá paleta, velká paleta, ...) a po vyndání palety s nebezpečností x_i mohou oba vozíky začít sundavat palety z výšky o x_i větší než dosud.

Hlavního zbrojmistra by zajímalo, kolik palet může ze skladu vyvézt ven, aniž by porušil bezpečností předpisy.

Konečně byla Stromboliho stíhačka připravená a vyzbrojená k průzkumné misi. Zašel si tedy na velmi pozdní oběd a pak se opět vydal do hangáru ke svému stroji. Usedl do kokpitu a pustil se do předletové přípravy. Po jejím dokončení pak ukázal palubnímu mechanikovi zdvižený palec a nechal se vyvézt výtahem na letovou palubu, kde počkal, než na něj dojde řada se startem.

Konečně, rameno katapultu se zakleslo za přední podvozkovou nohu „ef čtrnáctky“, Stromboli ukázal technikovi zdvižený palec, přidal tah motorů a pak už se jejich F14 Tomcat vyřítil po krátké vzletové dráze vstříc slunci.

Rychle vystoupali do výšky několika kilometrů a tam začali kroužit. Museli počkat, než dostanou od velitelství povolení k provedení akce. Thompson na zadním sedadle mezitím zapnul nový navigační systém, počkal, než se přijímač GPS ustálí, a začal prověřovat jeho funkčnost a propojení s průzkumným kontejnerem s kamerami, který měli zavěšený pod pravým křídlem.

Stromboli nechal Thompsona hrát si, navedl letadlo na kruhovou vyčkávací dráhu a čekal na finální pokyn k zahájení akce. Vzdušný prostor aktuálně brázdilo mnoho spojeneckých letounů – hlídky, zásobovací stroje i další průzkumné mise – a tak bylo potřeba udržovat přesně vymezené letecké koridory, aby se nikdo s nikým nesrazil. Naštěstí měla jejich mise nejvyšší prioritu.

27-1-3 Letecké koridory — 10 bodů

Vzdušný prostor nad spojeneckým námořním svazem není vůbec prázdný, a tak všechny letouny, které se v něm pohybují, musí dodržovat předepsané letové trasy neboli koridory.

Koridory mají předepsaný směr, kterým se jimi dá proletět, a vedou mezi určenými místy vzdušného prostoru (koridory a místa tak tvoří hrany a vrcholy orientovaného grafu).

Letoun se potřebuje dostat od letadlové lodi k místu plnění své mise, tedy je potřeba nalézt orientovanou cestu mezi dvěma zadanými místy.

Pilot letounu chce letět nejkratší trasou a zajímá ho, kolik má možností volby, tedy kolik různých nejkratších orientovaných cest vedoucích mezi těmito dvěma místy existuje (různé cesty jsou takové, které se liší alespoň v jedné hraně).

Už začal přicházet soumrak, když konečně dostali očekávané rozkazy. Stromboli vysílačkou potvrdil příjem, uchopil knipl a začal s Tomcatem klesat. Když se přiblížili k pobřeží a sestoupili do několika desítek metrů nad vodu, snížil rychlost a změnil šípovitost křídel na pomalý let. Nastavitelná křídla, to byl důvod, proč tyhle starší stroje pořád miloval.

Letoun sestoupil ještě o kus níž, už letěli jen pár metrů nad vlnami. Pod nimi se mihla pláž, Stromboli navedl Tomcat do jedné prolákliny a vtom se to stalo!

Najednou za táhlého pískání zablikaly a zhasly všechny displeje v kokpitu a stroj se hrozivě otřásl, jak se začal naklánět na bok. Stromboli hned popadl knipl a přitáhl ho, šlo to mnohem hůř než obvykle. „Co se stalo, zasáhlo nás něco?“ křikl dozadu na Thompsona. „Nevím. Já... najednou všechno zhaslo, asi porucha.“

Stromboli zaklel, zatracená elektronika, pomyslel si. Jeho pohled zabloudil k panelu vysílačky u levého kolena, ta ještě jako jedna z mála svítila, nebyla připojená k modernizovanému palubnímu počítači. „Mayday, mayday. Tady Krysa jedna, volám základnu. Mayday, mayday. Těžká porucha palubní elektroniky, stroj stěží ovladatelný, vracíme se na základnu.“

Ještě že v zapadajícím slunci byla jasně vidět černá tečka letadlové lodi. Stromboli k ní zamířil a doufal, že to stroj zpět na loď zvládne, bez asistence palubní elektroniky totiž vůbec nevěděl, jaký je jeho stav. Thompson se mezitím vzadu pokoušel zprovoznit alespoň průhledový Head-up display, aby Stromboli při přiblížení viděl před sebou jejich rychlost.

27-1-4 Head-up display — 10 bodů

Head-up display (HUD) zobrazuje informace v zorném poli pilota, a ten tak nemusí sklánět oči dolů a pak se jimi zase vracet. Bohužel je však docela citlivý na vyladění barev a kontrastu, a pokud se nastaví nesprávně, spíš pilota ruší.

HUD má N různých prvků. Každému z nich můžeme nastavit kontrast na nějakou hodnotu mezi 0 a K včetně. Prvky jsou na HUDu uspořádány vedle sebe, takže si je můžeme představit jako řadu N čísel.

Když poprvé spustíme HUD, dostaneme nějak nastavený kontrast. Chceme přenastavit kontrast všech prvků tak, aby se žádné dva prvky vedle sebe nelišily o více než D jednotek, a zároveň chceme provést co nejmenší celkovou změnu kontrastu (součet změn bude nejmenší možný).

Formát vstupu: Na prvním řádku budou čísla N, K a D, na druhém řádku pak N čísel udávajících výchozí kontrast všech prvků. Čísla jsou na řádku oddělena mezerou.

Formát výstupu: Na první řádek vypište součet provedených změn, na druhý pak uveďte nové hodnoty kontrastu pro všechny prvky (tedy N čísel oddělených mezerou). Pokud existuje více optimálních řešení, vyberte si libovolné z nich.

Ukázkový vstup:

```
6 30 3
2 7 9 13 16 14
```

Ukázkový výstup:

```
3
4 7 10 13 16 14
```

Toto je praktická open-data úloha. V odevzdávacím systému si necháte vygenerovat vstupy a odevzdáte příslušné výstupy. Záleží jen na vás, jak výstupy vyrobíte.

Stromboli navedl Tomcat na přistání a vysunul brzdící hák. Tohle přiblížení bez přístrojů nechtěl opakovat a byl rád, že slunce ještě nestihlo zapadnout. Už mu však moc nescházelo, a tak Stromboli s pozorností vybičovanou na maximum zahájil závěrečnou fázi přistávacího manévru.

Zadní kola dosedla na přistávací dráhu. Hák sice minul první brzdící lano, ale o druhé se již pevně zasekl a Tomcat, brzděný pružným lanem, zpomalil na několika metrech dráhy na nulu. Stromboli si vydechl, vypnul oba motory, sundal si helmu a prohrábl si zpocené vlasy. Tohle zvládli, teď bylo potřeba přijít na to, co se stalo.

Ještě ten večer si technici vzali jejich Tomcat do parády. Začali k němu připojovat všemožné diagnostické přístroje a zkoumali jejich údaje. Připojení diagnostických přístrojů k nefunkční elektronice ale není tak jednoduché, každý přístroj má totiž mírně odlišné požadavky na napájení.

27-1-5 Napájení přístrojů — 10 bodů

Technici připojují diagnostické přístroje k rozbité elektronice. Každý kus elektroniky má nějaký svůj povolený rozsah napájení (minimální a maximální hodnotu napětí, při které bezpečně funguje).

Technici mají k dispozici laboratorní zdroje, které je možné nastavit na přesné napětí. Každý laboratorní zdroj může napájet neomezeně mnoho kusů elektroniky.

Protože jsou ale laboratorní zdroje hodně používaná věc a na technické palubě letadlové lodě je o ně velký zájem, chtějí jich technici použít co možná nejméně (aby jich co nejvíce zbylo na ostatní práce). Pomozte jim zjistit nejmenší možný počet zdrojů, se kterými ještě dokážou uspokojit požadavky napájení všech kusů elektroniky dohromady.

Technici pracovali celou noc, ale na žádnou závadu na hardware nepřišli. Pro jistotu vyměnili jednotku palubního počítače a navigační systém. Další den večer se průzkumný let měl opakovat a rozkaz dostali opět Stromboli s Thompsonem.

Tentokrát se vznesli už za tmy a rovnou zamířili k oblasti, kterou měli v nízké výšce prolétnout. Stromboli opět klesl s letounem při nízké rychlosti do kaňonu a začali snímkovat kaňonem vedoucí silnici.

Pak Stromboli přitáhl Tomcat do těsného stoupání na konci kaňonu. Teď půjde do tuhého, pomyslel si, a navedl letoun nad hlavní cíl jejich průzkumu, těsný průlet nad irácký letištěm. Díky letu v nízké výšce o nich do poslední chvíle nevěděli, a tak se protiletecká palba začala objevovat až s dlouhým zpožděním.

První zareagovala nějaká hlídka. Spustila palbu z ručních zbraní, se kterými ale neměli skoro žádnou šanci Tomcat zasáhnout. První protiletadlový kanón vystřelil až ve chvíli, kdy dokončili oblet letiště a začali se stáčet směrem zpět k základně.

Stromboli spustil přídavné spalování a za bojového pokřikování navedl letoun do táhlé zatáčky, která je dostala mimo dosah protiletecké palby. Rád by udržoval přídavné spalování déle a vychutnával si ten příval adrenalinu, když ho silné přetížení tlačilo do sedačky, ale pohled na ukazatel paliva mu to rozmluvil.

Stáhl výkon zpět do normálních hodnot a začal svým letem kopírovat zemi, aby se držel mimo dosah posledních fungujících iráckých radarů. Když v tom se to stalo znovu...

Tomcat právě klesal do nějaké prohlubně, když zhasly všechny displeje v kabině a přestal fungovat radar, kterým se Stromboli řídil při nízkém letu nad zemí. Instinktivně přitáhl letoun o kus výš, aby se vyhnul překážkám, které teď neviděl, a opět zahlásil do vysílačky celkové selhání palubní elektroniky.

Teď byla ale situace vážnější, byla noc a návrat na základnu byl o to těžší. Z letecké patroly v blízkosti byla odvelena jedna F15, která je rychle dohnala a s rozsvícenými pozičními světly se usadila půl kilometru před nimi. Takhle je jako pasáček ztracenou ovci dovedla až nazpět k Ike, kde mezitím palubní personál řešil další problém s navigačními světly.

27-1-6 Přistávací světla — 12 bodů

Navigační světla na palubě letadlové lodě jsou tvořena třemi barvami: červenou, zelenou a modrou. Jsou rozmístěna podél dráhy v řadě N světel.

Palubní důstojník chce rozsvítit nějaký úsek světel tak, aby při pohledu skrz všechna svítící světla působil bíle. A zároveň chce, aby světlo bylo co nejsilnější, tedy aby svítilo co možná nejvíce světel.

Potřebuje tedy najít nejdelší souvislý úsek, ve kterém jsou všechny tři barvy zastoupeny ve stejném počtu.

⇧ **Lehčí varianta (za 8 bodů):** Vyřešte stejnou úlohu, ale jen pro dvě různé barvy.

Díky dobře osvětlené přistávací dráze a skvělému pilotnímu umu se Strombolimu povedlo i podruhé usadit letoun do brzdících lan bez jakékoliv navigační pomoci. Teď už byl však rozhněvaný, dvakrát stejná závada se mu vůbec nelíbila. Během převozu letadla výtahem na hangárovou palubu tedy prohodil několik nevybíravých vět s vrchním mechanikem a po ohlášení u velitele a krátkém hlášení padl vyčerpaný do postele. Alespoň že snímky tentokrát dovezli, a mise tak byla hotová.

* * *

Technici mezitím znovu prolezli celou F14 a hledali závadu v hardware. Nikde však žádnou nenašli, a tak obrátili svůj pohled k softwaru. A zde je čekalo velké překvapení, jedno drobné přehlédnutí, které způsobilo pád celého palubního počítače.

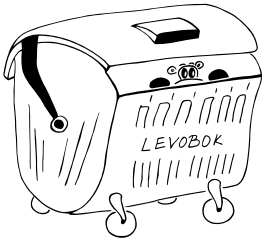
Letoun byl totiž vybaven novou verzí systému GPS, která mimo jiné počítá podle signálu z družic i nadmořskou výšku. Ukázalo se však, že se v jednom místě nadmořskou výškou dělí, aniž by byla zkontrolována nenulovost této hodnoty. A jelikož poručík Stromboli navedl letoun při obou letech do nízkého průletu kaňonem, jehož nejnižší bod se nacházel pod úrovní referenční mořské hladiny používané v GPS, došlo v obou případech k dělení nulou.

To pak vlivem propojení přístrojů v F14 zapříčinilo pád zbytku elektroniky (řízení ale ovlivněno nebylo, to je v F14 přenášeno ještě hydraulicky a mechanicky). Technici se z tohoto problému snad poučili a aktualizovali software zbytku amerických letadel – alespoň od té doby žádné podobné příběhy nejsou. Nebo vlastně... ale o tom zase třeba příště.

Jirka Setnička

Druhá série

Stejně jako v první sérii, i teď zavítáme k nějaké zajímavé programátorské chybě. V minulém díle jsme se potkali s chybou v GPS způsobenou dělením nulou, typické to softwarové přehlédnutí. Dnešní chyba však bude ukryta ještě hlouběji, je to na první pohled těžko tušitelné selhání v návrhu celého systému.



Podobně jako minulý příběh nás i dnešní zavede do války v Perském zálivu, tentokrát ale do města Dhahran *na západním břehu Perského zálivu v Saudské Arábii. Bohužel však tato chyba bude mít mnohem temnější následky...*

⋆⋆⋆

Bylo 25. února ráno a spojenecká základna v Dhahranu *se probouzela do dalšího dne, hlídky přebíraly nové skupiny vojáků a z kantýny se začala linout vůně připálených vajíček. Svobodník George Matthews do sebe rychle naházel snídani, vajíčkům se raději vyhnul, a vydal se na hlídkovou věž vystřídat jiného strážného.*

Cestou ještě podrbal svého psa, kterého měl jako pyrotechnik už mnoho let přiděleného. Dopey byl už za svých třináct let zvyklý na vojenský život na základnách, a tak jen šťastně zavrtěl ocasem, na svém řetězu doběhl k jedné z podpůrných noh blízkého přívěsu a označkoval ji.

„Systém za desítky milionů a ty si ho tady budeš značkovat?“ zasmál se George a vydal se okolo přívěsu s radarem protiraketového systému Patriot *dál. Systém to byl rozhodně impozantní a i díky němu si připadal v bezpečí. V bateriích v blízkosti radaru se nacházelo skoro třicet kusů protistřel, kterými systém sestřeloval blížící se irácké rakety* Scud. *Vždy si pro sestřel vybral tu nejvhodnější protistřelu a tu odpálil.*

27-2-1 Systém Patriot — 9 bodů

Protiraketový systém *Patriot* má k dispozici mnoho protistřel, které může proti blížící se hrozbě odpálit. Pro jednoduchost můžeme každou protistřelu charakterizovat pomocí jejího dostřelu (kladné reálné číslo). Za normální situace (pokud neurčí lidský operátor jinak) vyšle systém protistřelu, jejíž dostřel je mediánem mezi aktuálně dostupnými protistřelami.

Medián je prvek, který by se v setříděné posloupnosti nacházel přesně uprostřed. Pokud má posloupnost sudý počet prvků (tedy uprostřed leží dva prvky), budeme v tomto případě brát ten větší z nich (protistřelu s vyšším doletem).

Protistřely se do systému i doplňují (tak, jak jsou dopravovány na základnu), a proto by od vás spojenecká armáda potřebovala vybudovat rychlou datovou strukturu podporující dva typy operací:

1. Přidej protistřelu s doletem d_i do systému.
2. Odpal protistřelu s doletem, který je mediánem mezi aktuálními dolety (výsledkem by mělo být odebrání protistřely a vrácení jejího doletu).

⇧ **Lehčí varianta (za 3 body):** Vyřešte úlohu pro případ, kdy systém vysílá střelu s nejvyšším dostřelem.

Už se blížil čas oběda, když se základnou rozezněly poplašné sirény. George se rychle otočil na systém Patriot. Viděl, jak se jedna z baterií protistřel natočila směrem na severovýchod, a očekával odpal. Ale vteřiny ubíhaly a nic se nedělo. Po deseti vteřinách čekání skoro v ten samý moment většině přítomných vojáků došlo, že protistřela už nevyletí, že se něco porouchalo.

George se nedíval na ostatní, ale rychle sjel po žebříku z věže a sprintem se vrhl do blízkého úkrytu. Pak dopadl irácký Scud a obloha potemněla. Byl to den, kdy opěvovaný systém Patriot selhal, a nikdo zatím nevěděl proč.

* * *

Hned, jak lehce opadl mrak sutin a prachu, začali se z různých úkrytů vynořovat více či méně zranění vojáci. Ti zázrakem nezranění a ti s lehkými zraněními se hned vrhli do odhrabávání trosek zřícených budov.

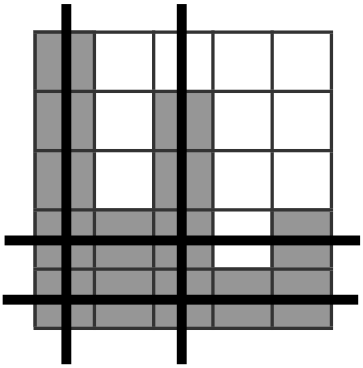
27-2-2 Prohledávání budov — 10 bodů

Po raketovém útoku stojí na základně několik poničených budov. Je nutné je všechny projít, odklidit trosky a hledat přeživší. Již se organizuje několik skupin záchranářů, ale je potřeba rozmyslet plán prohledávání.

Jedna skupina záchranářů může projít buď jednu budovu od přízemí do nejvyššího patra, nebo může naopak projít i-té patro v každé budově. Každé patro v každé budově je nutné prohledat alespoň jednou, vícenásobné prohledání nám nevadí.

Dostanete seznam budov a jejich počty pater. Rozmyslete, jak je všechny prohledat s co nejmenším počtem záchranných týmů.

Například pro výšky $(5, 2, 4, 1, 2)$ stačí čtyři záchranné týmy, jak je vidět na následujícím obrázku:



Už několik hodin pomáhal George vytahovat z trosek oběti. Většina z nich to přežije, ale už objevili i pár takových, kteří tolik štěstí neměli. Právě rozebírali pozůstatky po kantýně v centru základny, když pod troskami spatřil známou věc – lesklý obojek.

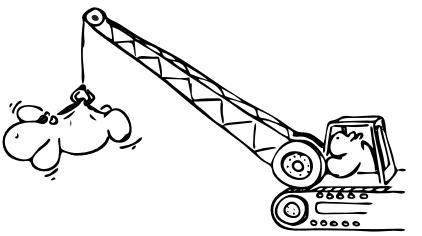
Rychle odházel stranou několik kovových nosníků a sevřel v náručí psí tělo. Dopey vypadal, že jen spí, ale nebylo tomu tak. Na George náhle dolehly události několika posledních hodin plnou silou, a tak se jen posadil na trosky a několik minut jenom hleděl do dáli.

„Třináct let a čtyřicet dva dní, pane," řekl důstojníkovi, který se u něj objevil, a pak ještě dodal: „Tolik mu bylo." „To je mi líto Matthewsi, ale teď sem potřebujeme rychle dostat nějaké jeřáby, aby nám pomohly s odklízením trosek." George odložil tělo Dopeyho opatrně ke stěně, osušil slzy a vrhl se na hordu map na plánovacím stole.

27-2-3 Průjezd jeřábu 10 bodů

Potřebujeme dostat autojeřáb z jednoho konce města *Dhahran* na druhý, aby pomohl s odklízením trosek na vojenské základně. Stavební firma sídlící na druhém okraji města je ochotná půjčit jakýkoliv ze svých jeřábů. My bychom chtěli k troskám dostat co největší jeřáb, ale čím větší, tím také vyšší – a ulice *Dhahranu* jsou prošpikované nízko zavěšenými elektrickými dráty.

Na vstupu máme mapu města zadanou jako síť ulic a křižovatek. Ulice jsou obousměrné, ale pro každou ulici máme údaj, jaké nejvyšší vozidlo jí ještě může bezpečně projet. Mapa města tedy představuje ohodnocený graf.



Navíc dostaneme ještě dvě označené křižovatky, sídlo firmy a vojenskou základnu. Najděte cestu mezi těmito dvěma místy, kterou může projet co nejvyšší jeřáb.

Konečně se jim povedlo všechny vyprostit, a také znovu zabezpečit základnu. Konečný účet byl 28 mrtvých spojeneckých vojáků a jeden pes k tomu. Zraněných bylo několik desítek. Všem také vrtalo hlavou, proč systém Patriot ani nevystřelil. Na to přijel hledat odpověď i vyšetřovací tým, který dorazil ještě toho dne večer.

Přesuneme se teď v příběhu od svobodníka Matthewse, který sehrál důležitou roli při záchranných operacích, k poručíkovi Blairovi, vedoucímu vyšetřovacího týmu.

Poručík Blair začal ihned shánět všechny informace o selhání. Protiraketový systém fungoval tak, že radar kontinuálně snímal celou oblast. Ve chvíli, kdy zachytil blížící se raketu Scud, přepnul se do přesnějšího módu, omezil snímání jen na oblast, ve které se raketa nacházela, a tím zpřesnil zaměření před odpálením protistřely.

Podle očitých svědků radar něco zaregistroval a připravil jednu z baterií protistřel na odpal. K samotnému odpalu ale již nedošlo. První věcí, kterou vyšetřovací tým potřeboval, bylo stáhnout družicové snímky oblasti z okamžiku vypálení Scudu. Zhruba každých deset sekund snímkovala americká družice oblast Iráku a čas startu rakety a její trajektorie by mohly pomoci s objasněním, co se vlastně stalo.

Problémem, se kterým se ale vyšetřovací tým musel nějak poprat, bylo to, že rychlost místního připojení k družicové informační síti byla příliš pomalá – dostačovalo k předávání běžných zpráv, ale na stáhnutí mnoha kompletních snímků z družic již ne. Naštěstí si Američané již před časem pro podobné věci vypracovali postup.

27-2-4 Stahování map — 12 bodů

Přes pomalé připojení potřebujeme přenést několik snímků o rozměrech $R \times S$ políček. Každý snímek můžeme přenést buď samostatně nezávisle na ostatních, pak nás jeho přenesení stojí $R \cdot S$ času, nebo jako diferenci od jiného, již přeneseného snímku. V takovém případě se přenášejí jen rozdílná políčka, ale je nutné počítat s režií přenosu W navíc (například proto, že nestačí jen přenést hodnotu na políčku, ale musíme ještě udat jeho souřadnice, tedy posíláme tři čísla namísto jednoho).

Konkrétně, pokud bychom chtěli snímek A_i přenést jako diferenci oproti již přenesenému snímku A_j a D_{ij} by nám vyjadřovalo počet rozdílných políček obou snímků, pak by náklady na přenos A_i byly $D_{ij} \cdot W$.

Na vstupu dostanete počet snímků N, jejich rozměry R a S, režii přenosu W a pak všech N snímků. Vaším cílem bude uspořádat snímky v nějakém pořadí a u každého zvolit, jestli se má přenášet celý, či jako diference od nějakého zvoleného snímku, aby celkové náklady na přenos byly nejmenší možné.

Při řešení úlohy předpokládejte $N \leq 500$, $R, S \leq 20$.

Lehčí varianta (za 4 body):
Řešte stejnou úlohu, ovšem s omezením $N = 3$.

„Tohle je všechno správně...“ vzdychl jeden z techniků po prohlédnutí stažených družicových snímků. „Scud přilétl skoro přímo doprostřed zorného pole Patriotu, takže ho musel detekovat. Baterie protistřel se také aktivovala, ale pak už od řídícího systému nedostala finální zaměření a pokyn k odpalu.“

„Dobře. Zkuste se podívat na jakákoliv hlášení související s chybami a údržbou systému Patriot za poslední dva měsíce,“ rozkázal poručík jednomu z desátníků. „My zatím zkusíme prozkoumat instrukční sadu, jestli není chyba v ní,“ dodal ke zbytku týmu.

27-2-5 Nejdelší příkaz — 12 bodů

Nervovým centrem každého systému *Patriot* je řídící počítač s několikanásobnou zálohou. Při zkoumání důvodů problému již technici vyloučili fyzické selhání počítačů, a tak padl jejich zrak na software.

Při programování se používá specifický programovací jazyk, ve kterém se jednotlivé příkazy skládají z posloupnosti klíčových slov. Každý příkaz může začít libovolným klíčovým slovem, ale každé navazující klíčové slovo může vzniknout jen vložením jednoho písmene do předchozího (posloupnost `UA,DUA,DUHA,DUCHA` je korektní, ale posloupnosti `DUA,DUCHA` ani `DUA,DUHA,DUHY` již ne).

Techniky by zajímalo, jaký nejdelší příkaz (co do počtu klíčových slov) lze v jazyce sestavit a jestli náhodou touto délkou nepřekročí délku vestavěného příkazového zásobníku, a nemůže tak způsobit systémové selhání.

Toto je praktická open-data úloha. V odevzdávacím systému si necháte vygenerovat vstupy a odevzdáte příslušné výstupy. Záleží jen na vás, jak výstupy vyrobíte.

Formát vstupu: První řádek obsahuje počet klíčových slov, na dalších řádcích jsou uvedena jednotlivá klíčová slova. Slova se mohou opakovat.

Formát výstupu: První řádek obsahuje délku nejdelšího příkazu, na dalších řádcích jsou vyjmenována klíčová slova tohoto příkazu. Pokud je nejdelších příkazů více, vypište libovolný z nich.

Ukázkový vstup:

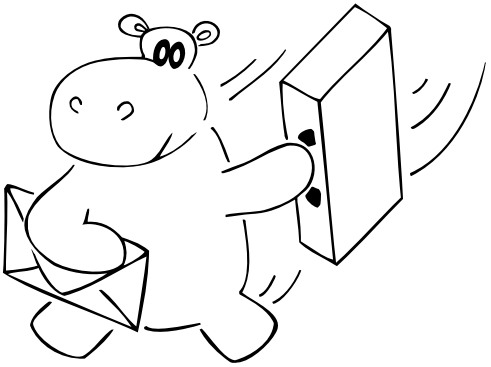
```
6
abc
aba
aa
b
bc
dacb
```

Ukázkový výstup:

```
3
b
bc
abc
```

„Program se zdá v pořádku, budeme si tedy muset ušpinit ruce," řekl znaveně poručík a odvedl svůj tým techniků ven k lehce poškozenému radaru. „Zkusíme zjistit, jestli jsou všechny části systému na svých místech a komunikují spolu."

„Ale pane, to bude hrozně moc práce, rozebrat to a prověřit každý kabel nám bude trvat dny!" „Máte snad lepší nápad?" zeptal se Blair. Mladý desátník se chvíli zamyslel, pak odběhl dovnitř, vytáhl ven jeden z počítačů a přinesl kupu propojovacích kabelů.



„Můžeme zkusit do jednotlivých uzlů systému vysílat signály a sledovat, za jak dlouho se dostanou do jiných. To by nám mělo dát odpověď na to, jestli jsou spoje v pořádku."

27-2-6 Testování odezvy — 14 bodů

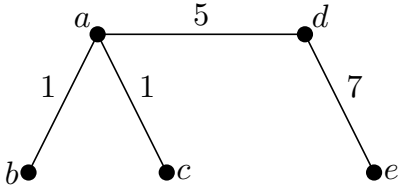
Technici naměřili na propojeném počítačovém systému, sestávajícím z N uzlů, jak dlouho trvá signálu dostat se z každého uzlu do každého jiného. Systém je tvořený propojenými dráty, a vyslané signály tak mohou volně procházet skrz celou síť, jen jim překonání každého drátu trvá určitý čas.

Z provedeného měření jsme dostali čtvercovou matici A o rozměrech $N \times N$. Vaším úkolem je sestrojit ohodnocený strom o N vrcholech, v němž vzdálenost mezi vrcholem i a vrcholem j odpovídá hodnotě A_{ij}, nebo říct, že žádný takový strom neexistuje.

Například pro matici

	a	b	c	d	e
a	0	1	1	5	12
b	1	0	2	6	13
c	1	2	0	6	13
d	5	6	6	0	7
e	12	13	13	7	0

je výsledkem následující strom:



Naopak si snadno rozmyslíte, že pro matici

	a	b	c
a	0	1	2
b	1	0	10
c	2	10	0

řešení neexistuje.

Lehčí varianta (za 6 bodů): Řešte úlohu s předpokladem, že strom, kterému matice odpovídá, je neohodnocený (tj. každá jeho hrana má jednotkovou délku).

Když vyšetřovací tým vyloučil softwarové selhání i nefunkčnost radaru, začali být skutečně bezradní. Vtom ale do místnosti vešel desátník, kterého Blair poslal zkoumat stará hlášení, a nesl v ruce desky s jednou zprávou. Bez vysvětlení je podal poručíkovi a Blairovy oči se rozšířily, když mu došlo, co právě objevili.

Byla to zpráva z jedné izraelské základny stará asi dva týdny. Naměřili tam, že po osmi hodinách provozu se střed zaměřovací oblasti Patriotu při přesnějším módu odchýlil o zhruba dvacet procent od místa, kde se reálně nacházela sledovaná střela. Taková odchylka ještě systému nevadila, ale zpráva uváděla, že se tím začal zabývat výrobce protiraketového systému.

Rychlé prolistování skrz hlášení z místní základny odhalilo to, že v tomto případě byl systém Patriot v nepřetržitém provozu skoro 100 hodin – a to už vadilo.

Interně si totiž Patriot počítal čas od svého spuštění jako celé číslo, ale při výpočtu dráhy střely a odhadu místa, kam by měl zaměřit přesnější režim sledování, si rychlost cíle a čas přepočítával do 24-bitového čísla s plovoucí desetinnou čárkou. A jelikož bylo po 100 hodinách provozu číslo udávající čas již příliš veliké, převod do floatu a následný výpočet vyústil v odchylku zaměření skoro 600 metrů.

Systém Patriot tak správně zaregistroval blížící se Scud, připravil protistřelu k odpalu a přepnul se do přesnějšího módu. Kvůli špatnému výpočtu však útočící střelu v přesnějším módu již neviděl, zkrátka proto, že se díval na špatné místo. Co systém nevidí, to nemůže sestřelit, a tak ani nebyla odpálena žádná protistřela.

Nejvíce ironické na celém incidentu je to, že softwarová oprava Patriotu, na které firma pracovala již od izraelského hlášení, dorazila do Dhahranu o den později. Dorazit o den dříve, tato katastrofa by se vůbec stát nemusela...

Příběh pro vás převyprávěl

Jirka Setnička

Třetí série

Letos se v jednotlivých sériích ohlížíme za zajímavými programátorskými chybami, a nejinak tomu bude i dnes. V předchozích sériích jsme viděli dělení nulou, ale také zrádnou chybu vzniklou převodem mezi celým číslem a floatem. Dnes nás oproti tomu čeká chyba, která vznikla zejména lidským přehlédnutím a strojovou kontrolou by byla těžko odhalitelná.

Také již opustíme válku v Perském zálivu a přesuneme se o několik let v čase, do doby, kdy většina z vás už byla na světě. Dnešní chyba ani nebude mít tak tragické následky, za oběť jí padlo „pouze“ několik set milionů dolarů. Teď se ale pojďme podívat do září 1999 na Patrickovu leteckou základnu.

* * *

James zaujatě pozoroval jednu z fotografií na zdi, zatímco hučení za ním sílilo. Když se ozvalo charakteristické cvaknutí oznamující, že voda je uvařená, vzal rychlovarku a zalil si kávu. Kuchyňkou se rozlila typická vůně.

Vyzbrojený milovaným nápojem se James vrátil do řídicí místnosti, kde se přidal ke svým kolegům navigátorům. Teď neměli mnoho práce, ale už za pár dní budou jejich znalosti velmi potřeba. Blížil se totiž čas, kdy Mars Climate Orbiter *vstoupí na oběžnou dráhu Marsu.*

Malé pozdvižení se ovšem dostavilo mnohem dříve. Na Zemi dorazila první fotografie Marsu. Pravda, obraz byl zkomprimovaný a možná patřičně nepřesný, ale navigátoři hned začali zkoumat, jestli na něm neobjeví vhodné místo k přistání. Po Mars Climate Orbiter, který má zkoumat atmosféru Marsu z jeho oběžné dráhy, totiž přijdou další sondy, a ty již budou na Rudé planetě přistávat.

27-3-1 Plocha k přistání — 14 bodů

Na Zemi dorazila fotografie zkomprimovaná do *kvadrantového kódu*. Nás zajímá, jaké místo na ní by bylo nejvhodnější k přistání, to znamená, kde je největší souvislá plocha.

Kvadrantový kód se používá pro dvoubarevné obrázky. Funguje tak, že se obraz nejprve rozdělí na čtvrtiny, které se postupně zakódují (pořadí kódování čtvrtin je „po řádcích“). Má-li celá plocha stejnou barvu (či je již tvořená jen jediným pixelem), zakóduje se jako jedno číslo (1 pro černou nebo 0 pro bílou barvu), v opačném případě se zpracovává rekurzivně.

Příklad takového kvadrantového kódu, který vznikl zakódováním z dvoubarevného obrázku, připojujeme níže. Tento zápis kvadrantového kódu je konzistentní s pátou úlohou, která ho také využívá.

```
1 1 0 0
1 1 0 0   ==>  (10(1100)(1010))
1 1 1 0
0 0 1 0
```

Vaším úkolem je v kvadrantovém kódu najít největší souvislou bílou oblast. Za sousední pixely považujeme jen ty, které spolu sousedí hranou (roh nestačí). Počítejte s tím, že se rozkódovaný obraz nevejde do paměti (tedy převést kvadrantový kód na obrázek a hledat oblast až v něm správné řešení není).

Poznámka: Kvadrantový kód funguje pěkně pro čtvercové obrázky o hraně délky nějaké mocniny dvou, ale dá se obdobně definovat i třeba pro obdélníkové obrázky. Protože to ale nepřináší nic nového, omezíme se v řešení úlohy jen na čtvercové obrázky o hraně délky mocniny dvou.

James po chvíli nechal své kolegy dál zkoumat a sám se ponořil do vzpomínek . . .

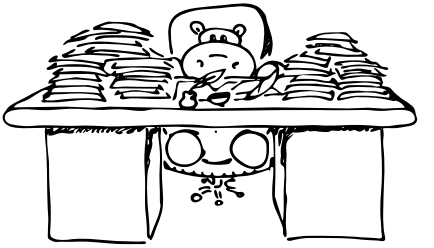
⋆ ⋆ ⋆

Když bylo v srpnu 1993 jen těsně před vstupem na oběžnou dráhu ztraceno spojení se sondou Mars Observer*, a tím podstatně oddáleny šance na bližší poznání Rudé planety, byl to šok, zvlášť pro Jamese a jeho tým.*

Netrvalo ale dlouho a začaly se připravovat nové mise. Problém vesmírných misí ovšem je, že stojí spoustu peněz, které na ně musí někdo přidělit. Za Jamesem brzy přišel šéf, že bude potřeba napsat žádost o grant. Naštěstí tehdy dobře věděli, na co jednotliví členové komise, která bude o schválení rozhodovat, slyší; mohli jim tedy napsat návrh na míru. Zajímalo je ale, jakou mají vlastně konkurenci.

27-3-2 Návrhy pro komisi — 12 bodů

Je potřeba podat návrh komisi a nás by zajímalo, kolik různých návrhů komise schválí. Komise má C členů, kteří všichni sami za sebe rozhodují o schválení návrhu. Jako celek pak komise návrh schválí, pokud ho schválí alespoň K jejích členů.



Návrhy jsou ovšem dlouhé a členům se nechce číst je celé. Každý člen má proto nějaký seznam slov, která se mu líbí, a schvaluje právě ty návrhy, které začínají některým z jeho oblíbených slov. Návrhy i oblíbená slova jsou řetězce složené z malých písmen anglické abecedy a navíc panuje dohoda, že každý správný návrh má délku právě D písmen.

Na vstupu tedy dostanete počet členů komise a pro každého z nich jeho oblíbená slova. Dále dostanete počet členů nutných ke schválení a přijatelnou délku návrhů D. Vaším úkolem je zjistit, kolik různých návrhů (tvořených jen z malých písmen anglické abecedy) může komisí projít jako schválené.

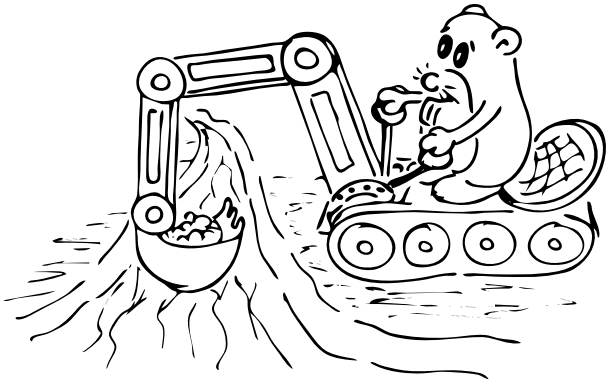
Zajímá nás jen počet těchto návrhů, nemusíte je generovat. Navíc se nemusíte zabývat tím, že se vám toto číslo nevejde do běžné číselné proměnné (toto není úloha na velká čísla).

Příklad: Uvažme trojčlennou komisi, ve které jsou potřeba alespoň dva její členové ke schválení návrhu, a návrhy délky čtyř písmen. Oblíbená slova jednotlivých členů vyjadřuje tabulka níže.

```
1. člen: pes psa
2. člen: psal kun
3. člen: pest ps
```


Je jasné, že návrh musí začínat na `p`, jinak by ho neschválili alespoň dva členové (na slovo `kun` tedy můžeme zapomenout). Možnosti, které nám zbývají, jsou tedy buď `pest` nebo `psaX`, kde `X` může být libovolné písmeno (všimněte si, že třeba `psbX` už je přijímané jen jedním členem komise, `psal` všemi a `psat` alespoň dvěma).

Možností je tedy dohromady $26 + 1 = 27$.



Snad právě proto, že znali preference jednotlivých členů komise, nebylo pro Jamesův tým těžké peníze získat. Po vyřešení finanční otázky ovšem přišly na řadu otázky další, techničtější a v mnohém složitější.

Většinu konstrukčních záležitostí řešila společnost Lockheed Martin*, se kterou NASA uzavřela smlouvu na výrobu sondy, přesto občas některé řešené problémy probublaly i k Jamesovi. K těm zajímavějším patřila konstrukce antény.*

Jednou z klíčových vlastností každé vesmírné sondy je totiž schopnost komunikovat s lidmi na Zemi. Od začátku bylo jasné, že na straně Země se k tomuto účelu využije síť Deep Space Network*, která byla na Zemi vybudovaná již koncem šedesátých let a využívá se pro komunikaci s jinými sondami.*

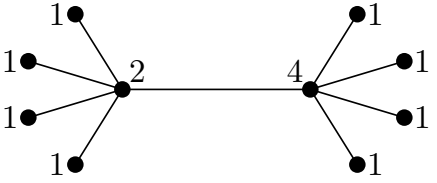
Aby mohla sonda do této sítě posílat informace, musí být ovšem vybavená dostatečně silnou anténou. Taková anténa se skládá z mnoha vysílačů, jejichž volba byla trochu oříšek. Tím spíš, že ač peníze byly, plýtvat se jimi nemohlo.

27-3-3 Výběr vysílačů 13 bodů

Anténa vesmírné sondy má stromovou strukturu, přičemž v každém uzlu se nachází nějaký vysílač. Kvůli rušení ale v žádných dvou sousedních uzlech nesmí být vysílače stejných typů.

Různé vysílače mají různou cenu, i-tý typ vysílače stojí 2^i dolarů (číslujeme od 0). Na vstupu dostanete popis antény, tedy který uzel sousedí s kterým. Určete, kolik nejméně dolarů bude stát umístění vysílačů do všech anténních uzlů.

Příklad: Na anténě níže vidíte, že v tomto případě je nejvýhodnější použít tři typy vysílačů (s cenami $2^0, 2^1, 2^2$ neboli $1, 2, 4$). Použít jen dva typy vysílačů by v tomto případě vyšlo dráž.



Lehčí varianta (za 3 body): Jako součást řešení vymyslete nějaký rozumně malý příklad antény, na které je potřeba použít čtyři různé druhy vysílačů, aby výsledná cena byla co nejmenší.

Rozumně malým příkladem nemyslíme nutně, aby měl co nejméně vrcholů to jde, ale spíše aby byl rozumně jednoduše zkonstruovatelný (jednoduchý popis konstrukce je lepší než obrovský obrázek o tisíci vrcholech).

⋆ ⋆ ⋆

Uběhlo několik dní od chvíle, kdy Jamese hlas jednoho z jeho kolegů vytrhl ze vzpomínání a vrátil do reality. To navigátoři museli vyměnit hledání souvislé oblasti na fotografii za počítání, kontrolování, konzultování, nové počítání a tak stále dokola. Sonda se totiž rychle blížila k Marsu a bylo třeba navést ji na takovou dráhu, z které se dostane do správné výšky nad povrchem planety.

Ještě ten den spočítali vše potřebné. O týden později, ve středu 15. září 1999, byl provedený čtvrtý manévr upravující trasu letu. Očekávalo se, že až se sonda 23. září dostane do blízkosti Marsu, bude se nad jeho povrchem nacházet ve výšce 226 kilometrů. Teď, tři dny před očekávaným vstupem na oběžnou dráhu, ovšem navigátorům vycházelo, že při zachování trajektorie bude výška mnohem menší.

James se zamračil na obrazovku počítače. Pak rychle něco naťukal do kalkulačky, kterou měl položenou před sebou, ale stále mu vycházelo málo. 158. 158 kilometrů nad povrchem Marsu místo očekávaných 226. To bylo o dobrou

třetinu méně. Zatím to nebylo kritické, Mars Climate Orbitter by měl s patřičnou úpravou oběžné rychlosti přežít ještě ve výšce 80 kilometrů, ale komu by se líbilo, když se realita takovým způsobem liší od očekávání? James se navíc děsil, že další den vyjde ještě méně.

Přitom od počátku mise probíhala dobře ...

* * *

Psal se 11. prosinec 1998 a spousta lidí v čele s konstruktéry a navigátory sledovala start nosné rakety Delta II*, která měla Mars Climate Orbiter dopravit na Hohmannovu elipsu.*

Mezi sledujícími James pochopitelně nemohl chybět, ačkoliv on kromě rakety důsledně sledoval i lecjaké naměřené údaje. Po odpočtu, během kterého ještě víc vystoupalo očekávání všech zapojených, byly zažehnuty motory. Objevil se jasný záblesk, který přešel v ohnivou čáru, a Delta II vystřelila vstříc modrému nebi, a ještě dál.

Tak začala 669 milionů kilometrů dlouhá cesta sondy, která měla odpovědět na mnoho otázek pozemšťanů.

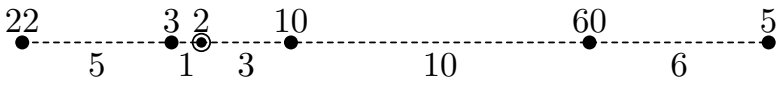
Let probíhal dobře, jen v jedné chvíli navigátoři zvažovali, zda by se nevyplatilo nechat sondu chvíli poletovat tam a zpět, aby její solární panely nasbíraly co nejvíc energie.

27-3-4 Doplňování energie — 12 bodů

Sonda prolétá vesmírem, kde některými místy prochází výjimečně silné sluneční paprsky. Solární panely dokáží z těchto paprsků získat energii, ovšem na přelet mezi místy vzdálenými i spotřebuje sonda i jednotek energie. Navíc odpadní látky zastíní paprsek, takže z jednoho místa lze energii čerpat pouze jednou.

Na vstupu dostanete popsáno, jaké množství energie se nachází v jednotlivých místech, a vzdálenosti mezi těmito místy. Dále dostanete určený výchozí bod, na kterém se sonda nachází. Určete, s jakou největší energií může sonda skončit.

Například pro situaci níže (horní čísla představují množství energie, dolní vzdálenosti mezi místy) se začátkem ve třetím bodě může sonda skončit maximálně se 64 jednotkami energie. Nejlepší řešení se z výchozího místa vydá těmito přelety: LRRLLLRRRR (L – doleva, R – doprava).



Jamesovy obavy nebyly plané, během dalších dvou dní klesla očekávaná výška, v které by sonda měla k planetě přiletět, o dalších 50 kilometrů. To by ale ještě stále mělo stačit. A dál už očekávaná výška klesat nemohla, chvíle, kdy Mars Climate Orbiter vstoupí na oběžnou dráhu Marsu, již byla na dosah.

Právě proto bylo v řídicí místnosti rušno jako málokdy, přestože ještě nebyly ani čtyři hodiny ráno. Blížil se jeden z převratných okamžiků kosmonautiky.

Stále nebylo jasné, proč se očekávání a realita tak rozchází. Výška, ač aktuálně odhadovaná na málo přes 100 kilometrů nad povrchem Marsu, ovšem dostačovala a vstup na oběžnou dráhu byl zahájen. Sonda složila své solární panely, vhodně se vůči planetě natočila a zažehla hlavní motor.

Ve čtyři hodiny a čtyři minuty bylo spojení se sondou zničehonic přerušeno. Navigátoři si vyměnili několik vyděšených pohledů. Snažili se obnovit kontakt, ale nedařilo se.

Ani ne o dvě minuty později měla sonda navíc vstoupit do zákrytu Marsu, kdy by tak jako tak nebylo možné s ní komunikovat. Nedaří se navázat spojení, jen protože je sonda v zákrytu, nebo protože se stalo něco mnohem ošklivějšího?

Jamesovi padl pohled na fotografii, která před dvěma týdny ze sondy dorazila. Tehdy to ještě šlo všechno skvěle!

* * *

Kdyby měla sonda lidské pocity, asi by se na své cestě dost nudila. Po zajímavém startu a troše poletování tam a zpět za světelnými paprsky již nic zajímavého nepřišlo. Zůstal jen dlouhý let černou tmou zpestřený pouze světly hvězd.

Po dlouhých devíti měsících sonda konečně doletěla na dohled Marsu. Ještě z velké dálky pořídila jeho fotografii, a protože na fotografii planety z vesmíru je mnoho tmavého místa, stejně jako mnoho světlého místa, rozhodl se počítač odeslat ji na zemi kvadrantisticky zkomprimovanou.

27-3-5 Komprese obrazu — 10 bodů

Sonda posílá snímek Marsu. Nejprve ho ovšem za pomoci ztrátové kvadrantistické komprese převede do kvadrantového kódu (popsaného v první úloze).

Při *kvadrantistické kompresi* se jedna čtvrtina obrazu prohlásí za celočernou, jedna za celobílou a zbylé dvě se zpracují rekurzivně. Pokud se rekurze dostane až na úroveň jednotlivých pixelů, může být už barva rekurzivních částí jakákoliv. Pro čtverec 2×2 ale ještě platí, že jedna jeho čtvrtina musí být celočerná, jedna celobílá a zbylé dvě libovolné. Pořadí kvadrantů je „po řádcích“.

Na vstupu dostanete původní obraz. Vaším úkolem je vypsat kvadrantový kód takové jeho kvandrantistické komprese, která se od původního obrazu liší v co nejméně pixelech. Konkrétněji bude mít vstup podobu popisu obrázku ve formátu PBM.[15] To je jednoduchý formát na ukládání černobílých obrázků.

[15] `http://en.wikipedia.org/wiki/Netpbm_format`

Obrázek je v něm kódovaný po řádcích, vždy jedno číslo (1 nebo 0) na jeden pixel. Na řádku jsou mezi jednotlivými čísly mezery a na konci každého řádku se nachází znak nového řádku, nic jiného se zde nevyskytuje. Platný PBM soubor je také uvozen na prvním řádku znaky `P1` a na druhém řádku mezerou oddělenými čísly udávajícími jeho šířku a výšku (v tomto pořadí).

Obrázky v této úloze budou pro jednoduchost vždy čtvercové o hraně 2^K pixelů a jejich velikost nepřesáhne 1024×1024 pixelů.

Na výstup vypište nejprve na první řádek počet změněných pixelů a následně na druhý řádek kvadrantový kód kvadrantistické komprese.

Ukázkový vstup:

```
P1
8 8
1 1 1 1 1 1 1 0
1 1 1 0 1 1 1 0
1 0 0 0 1 1 0 0
0 0 0 0 1 0 0 0
0 0 0 0 0 0 0 0
1 1 1 1 0 0 0 0
1 1 1 1 0 0 0 0
0 0 0 0 0 0 0 0
```

Ukázkový výstup:

```
8
((1(1110)(1000)0)
 (1(1010)(1110)0)
 1
 0
)
```

Poznámka k příkladu: Pro přehlednost příkladu jsme druhý řádek výstupu rozlomili po jednotlivých kvadrantech, v reálném výstupu by vše od první do poslední závorky bylo na jediném řádku.

Ve webovém zadání naleznete ještě jeden další ukázkový vstup.

Toto je praktická open-data úloha. V odevzdávacím systému si necháte vygenerovat vstupy a odevzdáte příslušné výstupy. Záleží jen na vás, jak výstupy vyrobíte.

K prohlédnutí obrázku ve formátu PBM můžete využít na Linuxu např. program Eye of Gnome (eog), na Windowsech programy Irfan View nebo XnView. Na obou systémech si s PBM poradí i Gimp či OpenOffice Draw.

⋆⋆⋆

Jamesova myšlenka, že tehdy *to ještě šlo skvěle, se časem ukázala jako nepříjemně přesná. Mars Climate Orbitter se totiž navigátorům neozval nejen po dvaceti minutách, kdy se měl opět dostat mimo zákryt Marsu, ale ani po hodině, ani po dvou dnech.*

Po těchto dvou dnech byla sonda oficiálně prohlášena za ztracenou a mise za neúspěšnou. Navigátoři zpětně spočítali, že sonda se ve skutečnosti dostala do výšky pouhých 57 kilometrů nad povrchem Marsu, kde ji zřejmě spálila atmosféra.

Ještě před oficiálním ukončením mise bylo zahájeno vyšetřování s cílem zjistit, co se vlastně stalo a proč se sonda pohybovala mnohem níž, než všichni oče-

kávali. James si rychle zvykl, že se teď kolem něj pohybuje mnohem víc lidí, že se zkoumá hned tu, hned ono. I jeho samotného zajímala příčina tohoto selhání a snažil se přijít věci na kloub.

Do místnosti právě vešel i jeden z techniků. „Hej, lidi, pomůžete mi někdo uklidit přepravky ve skladu?“ ptal se hned místo pozdravu. James usoudil, že trocha fyzické aktivity mu jen prospěje a přidal se k několika ochotným pomocníkům.

KSP

zadání

27-3-6 Ukládání přepravek — 9 bodů

Ve skladu je třeba uspořádat přepravky, a to tak, aby zabíraly co nejméně místa. Přepravky jsou kulaté, každá má svůj vnější a vnitřní průměr. Pokud je vnější průměr jedné přepravky menší než vnitřní průměr druhé přepravky, dají se vložit do sebe (a do nich případně ještě menší přepravka, vznikají tak jakési „komínky“).

Na vstupu dostanete vnitřní a vnější průměry všech N přepravek. Vaším úkolem je zjistit, do kolika nejméně komínků se dají uspořádat.

„Tedy, tohle bude mít pěkných pár liber,“ prohlásil jeden z pomocníků zvedaje pořádný komínek mnoha přepravek.

„Cos to řekl?“ vytřeštil oči Jamesův kolega Thomas.

„Jen že je to těžké. . . “ bránil se pomocník.

Ostatní, včetně Jamese, Thomase jen nechápavě pozorovali.

„O to nejde. Jde o ty libry! A o to, že libry nejsou kilogramy,“ pokračoval vzdor nechápavým pohledům Thomas. „A taky o to, že kilogramy jsou to, co ta sonda očekávala.“

Ozvala se hlasitá rána. To Jamesovi z rukou vypadlo několik přepravek. A podle výrazů ostatních byla spíš náhoda, že se to samé nestalo více lidem.

⋆⋆⋆

Vyšetřování potvrdilo, že příčinou selhání byla neshoda v používaných jednotkách. Řídicí středisko ze Země odesílalo instrukce s imperiálními mírami, kdy sílu udávalo v silových librách. Sonda je ovšem očekávala v metrické podobě, tedy sílu čekala v Newtonech.

Jelikož silová libra je více než čtyřnásobek Newtonu, došlo při výpočtech k chybám, které byly pro úspěšnost vstupu na oběžnou dráhu fatální. Rozkol mezi očekávanou a naměřenou pozicí byl zaznamenaný a v týmu zodpovídajícím za let družice se uvažovalo o provedení ještě dalšího, pátého, manévru korigujícího dráhu, ten ale nebyl nikdy provedený.

Neúspěšnou misi s vámi sledovala

Karolína „Karryanna“ Burešová

Čtvrtá série

Stejně jako v předchozích sériích, i v této budeme věnovat pozornost programátorské chybě, která měla, i přes svoji zdánlivou nevinnost, nedozírné důsledky. Situaci, o níž bude dnes řeč, nahrála i lidská nedbalost a kvůli tomu se mohla projevit jedna z nejzákeřnějších softwarových chyb, jež je oříškem i pro ostřílené programátory.

Celý příběh se odehrál na severovýchodě USA, v jednom horkém srpnovém dni roku 2003. Přesuňme se teď do dispečinku firmy FirstEnergy, amerického dodavatele elektřiny, kdesi v severním Ohiu...

⋆⋆⋆

Frank, jeden z operátorů na směně, odběhl z řídicí místnosti do kuchyňky a vzal lahev s vodou nejen pro sebe, ale také pro svého kolegu. „Díky moc," vydechl lehce obtloustlý Denis a otřel si z čela pot. Třicetistupňové teploty nebyly nic pro něj a celou směnu se snažil nastavit ventilátor tak, aby vanul přímo do jeho tváře.

Snad každá kancelář na severovýchodě Států teď měla zapnutou klimatizaci, čemuž odpovídala zvýšená spotřeba energie. Bylo potřeba zajistit její stabilní přísun. Frank už od rána mnohokrát upravoval parametry rozvodné sítě a několikrát žádal jižněji položené elektrárny o jejich nevyužitý výkon. Výstražný systém ohlašující každý problém se teď naštěstí na chvíli odmlčel a Frank měl čas se podívat na úkoly od svého nadřízeného. Díky novému systému, který ve firmě zavedli, jich naštěstí nebylo tolik.

27-4-1 Zadávání úkolů — 10 bodů

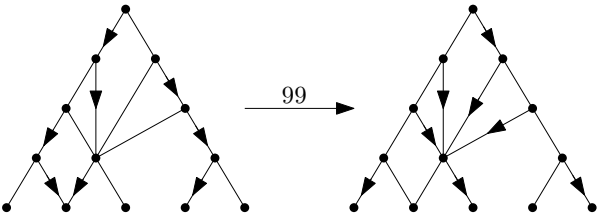
Ve FirstEnergy je přesně určena organizační struktura. Každý zaměstnanec v pozici vedoucího má právě dva podřízené (kteří mohou, ale nemusí být vedoucími), ostatní zaměstnanci na nikoho nedohlížejí. Jeden zaměstnanec může mít více nadřízených, v celém schématu však existuje právě jeden ředitel. Ředitel je vedoucí, který nemá žádné nadřízené a všichni ostatní zaměstnanci mu jsou (alespoň nepřímo) podřízeni. Hierarchii bychom tedy mohli označit jako souvislý acyklický hranově orientovaný graf (DAG).

Pouze ředitel může zadávat úkoly, každý úkol předá jednomu ze svých podřízených. Ten, pokud není vedoucím, musí úkol provést, jinak jej opět předá jednomu ze svých podřízených, a tak dále. Aby byly úkoly rozdělovány rovnoměrně, každý vedoucí (ředitele nevyjímaje) je musí předávat střídavě jednomu a pak druhému podřízenému.

O každém vedoucím víte, kterému ze svých dvou podřízených předá první úkol, který k němu dorazí. Ředitel zadá celkový počet N úkolů, které jsou postupně předávány celou hierarchií. Váš úkol se týká momentu, kdy jsou všechny

tyto úkoly vykonány. Určete pro každého vedoucího, kterému ze svých podřízených by předal další úkol, který by k němu dorazil. Pokuste se, aby vaše řešení bylo efektivní i pro velké hodnoty N (velkou hodnotou myslíme například bilión).

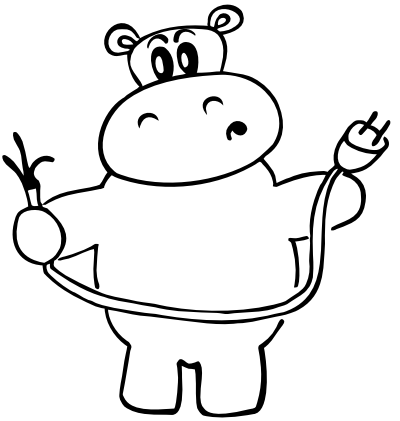
Na obrázcích vidíte příklad takové hierarchie. Šipky vyznačují, komu bude nový úkol předán. Na prvním obrázku je znázorněna výchozí situace, na tom druhém stav po 99 zadaných úkolech:



Lehčí varianta (za 5 bodů): Navrhněte řešení pro případ, kdy má každý zaměstnanec jen jednoho nadřízeného (grafem hierarchie je tedy strom).

Místností se rozezněl zvuk telefonu. Frank se natáhl a zvedl sluchátko. „Nazdar člověče, tady je MISO," ozvalo se. „Teď jsme zaznamenali výpadek vedení Star-South Canton. Jenom na malou chvíli, už zase běží. Všimli jste si toho?"

Organizace MISO koordinovala tok elektřiny mezi sítěmi jednotlivých společností. Frank sjel pohledem na obrazovku počítače a potřásl hlavou. „Tady nic nevidíme, alespoň chvíli tu máme klid. Není někde u vás chyba?" zvědavě se zeptal. „Hm... Máme tu nový software. Ještě se na to podívám," řekl operátor nejistě a zavěsil.*



* Midcontinent Independent System Operator

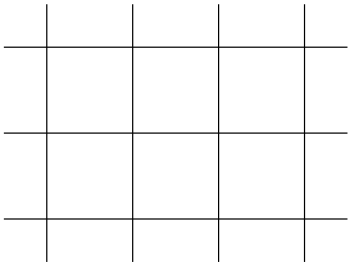
Zadání úloh KSP – 4. série

Není čeho se bát, pomyslel si Frank. Před dvěma hodinami přestalo fungovat 345kV vedení Stuart-Atlanta, směřující na jih – kvůli velkému průtoku proudu se dráty mezi sloupy začaly prověšovat a protože společnost nechala pod vedením přerůst stromy, vodiče se dotkly jejich špiček a došlo ke zkratu. Zátěž se však rozložila na jiná vedení a vše stále fungovalo stabilně. Rozvodná síť v USA byla od počátku navržená tak, aby ji takový výpadek nemohl rozhodit.

27-4-2 Čtverce v síti — 11 bodů

Projektanti rozvodné soustavy se rozhodují mezi několika návrhy rozmístění vedení. Aby určili míru spolehlivosti návrhu, potřebují zjistit, kolik se v něm dá najít různých čtverců, které jsou složené z vodičů.

Navrhněte algoritmus, jenž na vstupu obdrží přímky představující vedení, zadané v některém z běžných tvarů (např. obecnou rovnicí, případně pomocí dvou bodů), a určí, kolik navzájem různých čtverců tyto přímky tvoří.



Počítáme i vzájemně se překrývající čtverce, tudíž přímky na obrázku tvoří celkem osm čtverců.

V dozorně organizace MISO se Leonard, vedoucí směny, nedůvěřivě podíval na elektronickou mapu. Před chvílí došlo k neplánovanému odstavení jednoho z bloků jaderné elektrárny a mnoho linek se začalo barvit ze zelené do jemně žluté barvy. Jeden z pracovníků se po telefonu bavil s jiným dispečinkem o nejlepším řešení situace.

Mapa v reálném čase vykreslovala data vypočtená stavovým estimátorem. Program zachycoval údaje ze senzorů na vedení a počítal zatížení linek. Šlo o velice užitečný systém, ušetřující pracovníky od spousty výpočtů, ovšem za pouhé dva týdny, po které byl nainstalovaný, neměl vychytané všechny mouchy. Některé ze senzorů na něj ještě nebyly přímo napojené, a jejich stav bylo třeba aktualizovat ručně. Je to stejně daleko jednodušší než před dvaceti lety, utěšoval se Leonard. Vzpomněl si na jednoho ze starších techniků, s nímž se před týdnem bavil o martýriu při zjišťování napětí v síti.

27-4-3 Vysoké napětí — 11 bodů

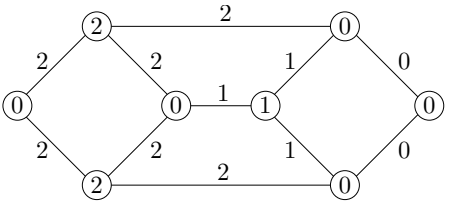
Pracovníci dispečinku potřebují zjistit, jaké napětí se vyskytuje na různých bodech sítě. Sice ví, že v uzlových bodech soustavy se vyskytují jen tři různé hladiny napětí – 0, 100 a 200 kilovoltů, ale informace mají pouze ze senzorů na vodičích, které tyto body spojují a které měří rozdíl napětí mezi koncovými body (nevíme však, kde je napětí větší, jinými slovy, ze senzorů vyčteme pouze absolutní hodnotu rozdílu).

KSP

zadání

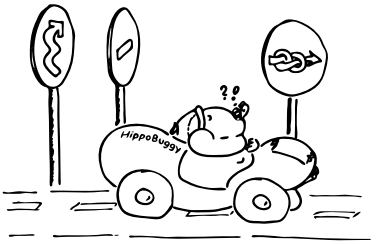
K dispozici jsme dostali mapu sítě (tvořenou uzlovými body a vodiči – jinde než v bodech se vodiče nekříží) a pro každý vodič hodnotu rozdílu napětí mezi uzly, které spojuje (buď 0, 100, nebo 200 kV). Máme za úkol určit hodnoty napětí na koncových bodech, aby rozdíly na vodičích odpovídaly (pokud je více možných řešení, stačí nalézt jedno z nich), nebo zjistit, že došlo k chybě a žádné takové řešení neexistuje.

Příklad: Vrcholům grafu na obrázku (hodnoty jsou ve stovkách kilovoltů) napětí přiřadit lze. Jedno možné přiřazení napětí vrcholům je zakreslené přímo do obrázku.



Lehčí varianta (za 7 bodů): Vyřešte úlohu pro dvě napěťové hladiny v bodech, 0 a 100 kV.

Nepříjemně hlasitý zvuk alarmu přerušil tok Leonardových vzpomínek. Okamžitě přiběhl k nejbližšímu ovládacímu pultu. „Vypadlo nám jedno z hraničních vedení, 345 kilovoltů," ukázal mu jeden z dispečerů na obrazovce místo poruchy. „Jak to?" Leonard překvapením pozvedl obočí. „Běželo na osmdesáti procentech zátěže. . . " rychle uvažoval, jakým přičiněním k tomu mohlo dojít.



A náhle mu na mysl přišla jedna událost z dopoledne. Stuart-Atlanta, nyní vypojené vedení a jedno z páteřních v celé oblasti, patřilo k těm spojením, se kterými estimátor počítal nepřímo, jen přes manuální zadávání stavů! „To snad ne," zamumlal a rychle se přepnul na obrazovku systému. A navzdory sytě zelené barvě, kterou bylo Stuart-Atlanta značené, ho polil mráz.

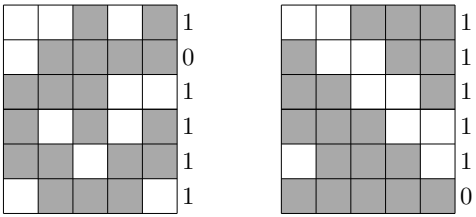
„Máme chybu v systému," zvolal vyděšeně. „Stavový estimátor nám počítá nesmysly!" Dispečerům okamžitě došlo, jakou chybu udělali, jeden z nich usedl k počítači a zadal správné hodnoty. Během asi minuty, kdy probíhaly nové výpočty, přemýšlel Leonard, jakým způsobem by šlo síť, teď už notně oslabenou, odlehčit. Tak jako je tomu při řešení hlavolamů, byla v tuto chvíli klíčová schopnost vhodně kombinovat...

27-4-4 NP-úplný hlavolam — 11 bodů

Hlavolam, se kterým si budeme hrát v této úloze, sestává z $R \times S$ políček uspořádaných do mřížky s R řádky a S sloupci. Některá z políček jsou průhledná, ostatní jsou neprůhledná.

Pod sloupce mřížky můžeme zasouvat barevné proužky. Zasuneme-li proužek pod zvolený sloupec, pak všechna průhledná políčka tohoto sloupce uvidíme nyní jako barevná. U každého řádku je číslo – nula nebo jednička, které udává, kolik barevných políček má být v tomto řádku vidět.

Na následujícím obrázku vidíte dva hlavolamy. Průhledná políčka jsou znázorněna bíle, neprůhledná políčka šedivě. Řešením prvního hlavolamu je zasunutí proužků do druhého, třetího a pátého sloupce. Druhý hlavolam řešení nemá.



Po vás však nechceme návod na řešení hlavolamu. Raději dokažte, že úloha rozhodování toho, zda má zadaný hlavolam řešení, je NP-úplná. Pokud se vám to nebude dařit, dokažte alespoň NP-úplnost této úlohy pro obecnější hlavolamy, kdy u každého řádku může být požadován libovolný počet barevných políček, nikoliv pouze nula nebo jedna.

Na dispečink FirstEnergy mezitím dorazili technici. Jejich provinilý výraz ve tváři bylo snad to jediné, co Frankovi zabránilo, aby je popadl za límec košile a začal s nimi třást. Po téměř hodině od poslední zprávy výstražného systému a

mnoha telefonátech od sousedících společností, které je upozorňovaly na blížící se nebezpečí kolapsu, byla zřejmá závažnost celé situace.

„Spadl nám server, asi před půlhodinou," přiznal jeden z techniků. „Moc jsme tomu nevěnovali pozornost, jenom jsme zkontrolovali, že běží záložní počítač. Ale... Ten před deseti minutami... spadl taky." „Takže proto nám všechny programy tady běží hlemýždí rychlostí?" zeptal se Frank. Jindy celkem svižný systém najednou potřeboval na získání nových informací téměř minutu. „Jistě," přikývl technik, „ale je tu ještě jedna věc. Až po pádu toho druhého serveru jsme zjistili, že... totiž... nefunguje výstražný systém. Snad víc než hodinu," vyklopil ze sebe.

Denis mezitím skončil telefonický rozhovor. „Zase MISO," oznámil. „Nezbývá nám prý nic jiného, než snížit napětí u odběratelů. Nějak se to začíná hroutit!" Frank přikývl, šlo o nouzové, ale stále poměrně rozumné řešení. Jenže – „jak se to dělá?" zeptal se Denise. „Naposledy jsme to dělali – kdy vůbec?" odpověděl Denis a poděšeně se podíval na techniky, kteří jen rezignovaně pokrčili rameny.

⋆⋆⋆

V MISu už bylo jasné, že se situace stává kritickou. Estimátor konečně spočítal skutečný stav sítě a ten byl daleko méně optimistický než předtím.

V jedné chvíli se těsně za sebou vypojilo pět vedení, jednoduše proto, že procházející proud byl až příliš velký. Další elektrárny hlásily odstavení a začala se objevovat první místa kompletně odpojená od elektřiny. Neustále se ozývalo zvonění telefonů, to jak se operátoři snažili odpojit nestabilní soustavu od ostatních částí.

A pak, několik minut po čtvrté hodině, se vše začalo hroutit jako dominové kostky. Během několika vteřin vypadly všechny zbývající linky dopravující elektřinu do Ohia – bylo jich tolik, že Leonard ani nestačil číst jejich označení na displeji – elektrický proud si našel cestu přes Pensylvánii a New York a během mžiku odstavil většinu tamější sítě, spolu s elektrárnami. Padesát milionů lidí se ocitlo bez elektřiny. Provoz vlaků a hromadné dopravy ve městech byl přerušen.

⋆⋆⋆

Přestože na některých místech ke zprovoznění stačilo několik hodin, celá rozvodná soustava fungovala až po několika dnech. Město New York, které na elektřinu čekalo do tří hodin ráno, se stalo symbolem celého výpadku. Fotografie davů lidí jdoucích pěšky po Brooklynském mostě, dopravních kolapsů nebo potemnělého Manhattanu, ozařovaného pouze svitem zapadajícího slunce, obletěly celý svět. Naštěstí se nevyskytly případy rabování, došlo ale k několika požárům vzniklých od zapálených svíček. Protože byl horký letní den, většina restaurací začala nabízet své jídlo komukoliv, kdo o ně požádal, protože by se bez chlazení stejně zkazilo. Když Leonard později v televizi sledoval reportáž se záběry na techniky opravující elektrickou síť, kteří přišli v montérkách do luxusně vypadající restaurace a konzumovali očividně drahé pokrmy, nemohl se tomu nezasmát.

27-4-5 Večeře pro opraváře — 12 bodů

Elektrotechnik Larry strávil celé odpoledne zjišťováním poruch vedení ve městě, a jelikož mu je jasné, že se práce dnes protáhnou do pozdních nočních hodin, chce se na takový úkol pořádně navečeřet. Do zjednodušené mapy Manhattanu (čtvercová síť s vyznačenými budovami, policisty na křižovatkách a ostatními věcmi, které nelze procházet; všechna ostatní pole ano) si proto zakreslil hospody, restaurace, ba i zmrzlinářství, kde se chce najíst. Je mu ale jasné, že času není nazbyt a že zásoby jídla nejsou bezedné.

Proto by potřeboval najít nejkratší trasu chůze po Manhattanu, během níž všechny zakreslené podniky navštíví. Můžete předpokládat, že podniků nebude více než dvacet. A pospěšte si, zmrzlina už teče!

Toto je praktická open-data úloha. V odevzdávacím systému si necháte vygenerovat vstupy a odevzdáte příslušné výstupy. Záleží jen na vás, jak výstupy vyrobíte. Přesný popis formátu společně s ukázkovými vstupy nalezenete ve webové verzi zadání.

Okamžitě po opětovném spuštění soustavy se rozběhlo vyšetřování, které odhalilo množství lidských i technických chyb. Dispečeři MISa byli kritizováni za spoléhání se na systém, jehož části byly teprve ve vývoji. FirstEnergy nedostatečně proklesťovala stromy pod vedením: to kvůli zkratům vypadávalo i při mírně nadprůměrném zatížení. Navíc nedostatečně školila obslužný personál: jinak by nedošlo k tomu, že by dispečeři nevěděli, jak ručně snížit napětí u odběratelů.

Ale co zhroucení serverů a výstražného systému? Aby vyšetřovatelé zjistili, v jakém stavu se programy nacházely před tím, než přestaly fungovat, začali procházet jejich záznamy. Ty se ve společnosti stále ještě ukládaly na magnetické pásky.

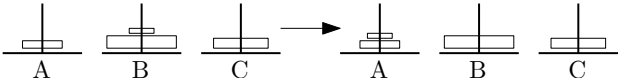
27-4-6 Stěhování pásek — 11 bodů

Pásky s důležitými daty jsou namotány na ploché kotouče, které však mají různý průměr. Ve FirstEnergy se k ukládání záznamů používá poměrně malá místnost, kde jsou kotouče položeny na sebe a seřazeny takovým způsobem, že ten největší je vespod a ten nejmenší navrchu.

Problém takovéhoto věžovitého uspořádání spočívá v obtížném stěhování. Před několika týdny museli pracovníci všechny kotouče přesunout, aby se do skladu vešla i část firemního archivu. Aby se pásky nepoškodily, museli postupovat podle striktních pravidel: v jedné chvíli mohli přesouvat pouze jeden kotouč (pokud je víc kotoučů na sobě, mohou přesunout pouze ten, který je navrchu), a aby se věž složená z kotoučů nezhroutila, nesměli položit kotouč s větším průměrem na kotouč s průměrem menším.

Protože je sklad skutečně stísněný, kromě místa označeného `A`, kde se kotouče vyskytovaly nejdříve, a místa `B`, kam byly přesunuty, mohli pracovníci kotouče odkládat pouze na jedno další místo `C` – všude je však museli skládat na sebe za dodržení uvedených pravidel.

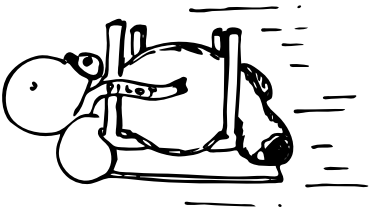
Vyšetřovatel získal fotografie vzniklé při tomto stěhování a chce se ujistit, že během něj nedošlo k manipulaci s daty. Na vstupu dostanete každou fotografii popsanou jako řetězec znaků `A`, `B` a `C`, kde k-tý znak určuje polohu k-tého nejmenšího disku na fotce (protože jsou na každém místě disky seřazeny od největšího po nejmenší, je tím jejich poloha určena jednoznačně). Na obrázku vidíte rozmístění kotoučů odpovídající řetězci `BACB` a také rozmístění odpovídající řetězci `AACB`, které vzniklo z prvního rozmístění přesunutím nejmenšího kotouče na místo `A`.



Víme, že pracovníci přemístili všechny kotouče z místa `A` na místo `B` způsobem, při kterém byl počet přesunů kotoučů mezi místy nejmenší možný. Seřaďte zadané fotografie podle času jejich pořízení.

Ukázkový vstup:	*Ukázkový výstup:*
`CCA`	`AAA`
`CCB`	`BCA`
`BCA`	`CCA`
`AAA`	`CCB`

Podezření, že počítač byl napaden virem, se ze záznamů nepotvrdilo. I hardwarové selhání bylo téměř vyloučeno, a proto nezbylo nic jiného, než přikročit ke zkoumání zdrojového kódu. Tým programátorů se několik týdnů pokoušel simulovat různé situace a testoval program na nejrůznějších vstupech. A nakonec ji nalezli – téměř neviditelnou chybu projevující se jen při kombinaci různých podmínek, které se právě toho nešťastného odpoledne splnily.



Výstražný systém pro řazení událostí používal datovou strukturu, do níž mohlo simultánně zapisovat více vláken, tedy procesů, které spolu sdílejí data. V takovém případě se používají různé mechanismy zajišťující, aby se taková struktura

nerozbila. Přestože systém toto obvykle zvládal dobře, při splnění podmínek – jednou z nich bylo i velké množství přicházejících výstrah – mohla vlákna zapisovat do stejného místa ve struktuře současně. Takový chybový stav se obecně nazývá race condition, a aby ve vícevláknové aplikaci nemohl nastat, musí být kódu věnována velká pozornost.

Poškození dat vedlo k chybám při přidávání nových zpráv, procesy se chovaly nekorektně, zacyklily se a přehlcený server následně spadl. Záložnímu serveru však nezměněná data předal, k chybovým situacím docházelo nadále a i ten nakonec přestal fungovat.

Ve FirstEnergy bohužel nefungovalo nic, co by dispečery na vypnutá varování včas upozornilo. Kvůli tomu je ani nenapadlo sledovat síť pomocí jiných prostředků a na to, co způsobili, začali přicházet až po samotném kolapsu.

Celou událost převyprávěl

Jakub Maroušek

Pátá série

O čem bude příběh této série? Opět o nějaké velké programátorské chybě, která stála desítky miliard dolarů, nebo alespoň desítky lidských životů? Ne, tentokrát ušetříme. Dočtete se kromě jiného o chybě nechybě, problému neproblému, jaký byl problém s rokem 2000 neboli Y2K.

Přesuňme se nyní do Prahy krátce před začátek druhého milénia.

⋆⋆⋆

KSP

zadání

Vážený pane,

dovolujeme si Vás znovu upozornit, že dlužíte za pojištění 0.00 Kč. Tuto částku jste měl uhradit již před třemi měsíci a neučinil jste tak ani po třetí upomínce.

Žádáme Vás, abyste dlužnou částku zaplatil do 7 dnů. Pokud tak neučiníte, bude na Vás vymáhána soudní cestou.

Tento dopis dostal Karel od pojišťovny, když se jednoho dne vrátil z práce.

Co s ním měl dělat? Zahodit, jako ty tři upomínky předtím? V té pojišťovně asi nevědí, že zaplatit nula korun vyjde nastejno jako nic nezaplatit, což udělal. Výhružka soudem přece není jen tak. Mohl by tam zajít osobně, ale je to docela daleko. Nebo to zkusit zaplatit...

A tak se stalo. Karel se rozhodl, že půjde na poštu a zaplatí to. Ještě si ale prohlédne ostatní dopisy. Hurá, má si vyzvednout balík od maminky z Anglie!

27-5-1 Šíření poplašné zprávy — 10 bodů

Firma, ve které pracuje Karlova matka v Anglii, má spoustu kanceláří a místností v jedné velké budově.

Jednou za čas tam testují poplašný systém. Z centrály mohou zavolat telefonem naráz do dvou kanceláří poplašnou zprávu „Hoří, všichni rychle opusťte budovu!“. Nato z daných kanceláří vyběhnou zaměstnanci do všech sousedních kanceláří a tam jim předají tuto zprávu. Všichni doběhnou ve stejném okamžiku, tomu celému můžeme říkat třeba jeden takt. V dalším taktu i z těchto kanceláří vyběhnou zaměstnanci do všech sousedních kanceláří, ve kterých se ještě o zprávě nedozvěděli, a tak dál.

Při poplachu se ale smějí používat pouze chodby a schodiště označené symbolem zeleného utíkajícího panáčka, to jsou ty, které vedou k únikovému východu. Tyto chodby tvoří neorientovaný strom.

Nás by zajímalo, do kterých dvou kanceláří máme zavolat, aby se zpráva rozšířila co nejrychleji po celé budově.

V Anglii je mnoho velkých firem, které používají svůj starý informační systém z 50. let napsaný v COBOLu. V té době byla paměť počítačů drahá, a tak se

jí šetřilo, co to šlo. Proto například místo letopočtu ukládali pouze jeho poslední dvě číslice, protože ta 19 na začátku je přece všude stejná, to si každý domyslí.

To ale nepočítali s tím, že ty programy budou chtít používat i po roce 2000, a teď se bojí, že jim to přestane fungovat. Ale místo toho, aby si nechali naprogramovat svůj software znovu úplně od začátku, což stojí spoustu peněz a času, si radši najmou programátora, který umí COBOL, a ty části, kde se používá datum, jim opraví. Za to mu dají spoustu peněz, hlavně aby měli jistotu, že to bude fungovat i dál.

Poptávka po takovýchto lidech je ale najednou větší než nabídka. A tak jednou jeden pán z Anglie zavolal i Karlově mamince. Ta umí programovat v COBOLu, protože byla sekretářka a za jejího mládí se COBOL i v Československu používal pro hromadné zpracování dat, tedy téměř pro stejné věci, jako dnes Excel. Zeptal se jí, jestli by pro ně pár měsíců nechtěla pracovat a přitom si pěkně vydělat. A tak je teď v Anglii.

Cestou na poštu šel Karel kolem autobusové zastávky u jednoho velkého supermarketu. Byla tam spousta lidí, kteří se připravovali na to, že 1. ledna 2000 přestanou fungovat všechny počítače na světě a zavládne naprostý chaos, nebudou létat letadla, obchody budou zavřené nebo vyrabované, nastane třetí světová válka kvůli (původně) omylem odpáleným raketám a tak dále. Říká se, že to všechno nastane kvůli tomu, že všechny počítačové programy na celém světě přestanou fungovat, protože nebudou umět zacházet s letopočtem 2000.

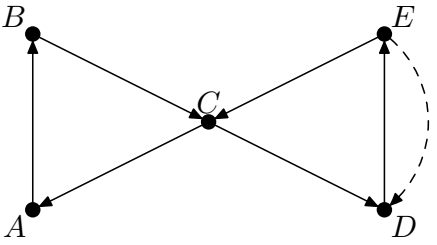
27-5-2 Survivalisté — 12 bodů

Tito lidé (říká se jim survivalisté) si v supermarketu nakoupili velké zásoby trvanlivých potravin, aby v následující krizi přežili co nejdéle. Teď si krátí čas při čekání na autobus tím, že si navzájem ukazují, co si kdo koupil. Postávají v hloučcích, podávají si konzervy, balíčky nebo lahve, a ochutnávají.

Žádný z nich ale nechce podat svou věc úplně každému. Například jsou mezi nimi lidé, kteří se neznají, kteří odmítli ochutnat nabízenou věc nebo patří k jiné skupině, než k té, se kterou budou trávit konec světa v bunkru a následně obnovovat lidskou populaci. Takoví lidé se spolu nebaví a věci si nepůjčují. Survivalisty tedy můžeme popsat orientovaným grafem, z každého survivalisty vede hrana do těch, kterým je ochoten věc podat.

Teď každý člověk vytáhne z tašky jednu věc a předá ji někomu jinému, aby ji ochutnal. Zároveň ale chce, aby někdo jiný podal nějakou věc jemu. Na vás je, abyste rozhodli, zda je toto možné.

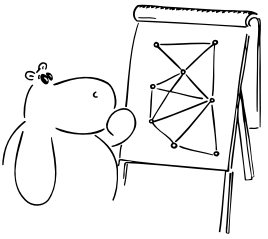
Například survivalisté na následujícím obrázku si takto věci předávat nemohou. Pokud bychom však přidali hranu $E \to D$, už by předávání mohlo proběhnout.



Na poště byla hodně dlouhá fronta, Karel vyplnil složenku na 0 korun a zařadil se.

27-5-3 Čekání na poště — 9 bodů

Když je na poště tolik lidí, že není snadné poznat, kde fronta začíná a kde končí nebo kdo je za kým na řadě, rozmístí na podlahu sloupky a natáhnou mezi nimi pásku. Na začátku a na konci fronty je samozřejmě mezera, kterou se prochází, ale pro jednoduchost si představme, že i tam je páska, i když pomyslná. Sloupky a páska tak tvoří mnohoúhelník, který neprotíná sám sebe. Lidé čekají uvnitř tohoto mnohoúhelníka.



Váš program dostane na vstupu rozmístění sloupků, tedy N bodů v rovině (nebudou tvořit přímku). Na vás je, abyste z nich vytvořili mnohoúhelník, který neprotíná sám sebe.

Formát vstupu: Na prvním řádku dostanete číslo N, tedy počet bodů. Poté následuje N řádků, kde na každém bude mezerou oddělená x-ová a y-ová souřadnice jednoho bodu. Pro všechny vstupy platí $3 \leq N \leq 250\,000$, $0 \leq x, y \leq 1\,000\,000$.

Formát výstupu: Výstupem programu bude jediný řádek obsahující čísla bodů oddělená mezerami seřazená podle výskytu na obvodu mnohoúhelníka. Body jsou číslované od nuly podle pořadí na vstupu.

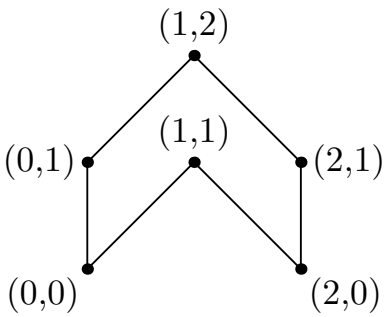
Pozor, výstup nemusí být jednoznačně určený (vypište libovolný korektní).

Ukázkový vstup:

```
6
0 0
2 0
1 2
1 1
0 1
2 1
```

Ukázkový výstup:

```
0 4 2 5 1 3
```



Toto je praktická open-data úloha. V odevzdávacím systému si necháte vygenerovat vstupy a odevzdáte příslušné výstupy. Záleží jen na vás, jak výstupy vyrobíte.

Konečně se Karel dostal na řadu.

„Dobrý den, já bych chtěl poslat tuto složenku.“ Podal ji pošťačce, ta se na ni podívala, vytáhla ze stojánku prázdnou a podala mu ji se slovy: „Tady máte novou složenku, vyplňte to prosím znova, napsal jste tam nula korun.“

„Paní, to není vyplněno špatně, já opravdu chci poslat nula korun.“

„Ale to nejde. Nula korun se nedá poslat. Nejméně, co můžete poslat složenkou, je 10 haléřů. Ale k tomu zaplatíte ještě 10 korun poštovné.“

„Ale to musí jít, vždyť mně přišla upomínka na nula korun od pojišťovny, podívejte se. Když ty peníze nepošlu, dají mě k soudu.“

„Bohužel, nejde to. Chcete poslat 10 haléřů? Nebo nic?“

„Vy mi tu složenku nepošlete? Tak mi prosím zavolejte ředitele této pobočky.“

A tak dále. Karel se rozhodně nenechal odbýt. Ředitele sice nezavolali, protože již nebyl v práci, místo toho ale sháněli nějakého zaměstnance, který prošel speciálním školením o poštovním řádu.

27-5-4 Školení zaměstnanců — 10 bodů

Pošta, jako každá firma, má hierarchickou strukturu zaměstnanců, a tato struktura tvoří strom.

Manažeři ze školícího oddělení čas od času vysílají některé zaměstnance na speciální školení, kde se naučí, jak se vypořádat s určitými situacemi, které někdy mohou nastat, ale nejsou tak obvyklé, aby školení využil každý zaměstnanec. Navíc zjistili, že pro zaměstnance je jednodušší se nové věci dozvídat od svých

kolegů než se ptát úplně cizího odborníka. A počítají i s tím, že vyškolení se budou chlubit novými znalostmi kolegům, a tak se informace snadno rozšíří po celé firmě.

Konkrétně si řekli, že by bylo dobré, aby se každý zaměstnanec mohl obrátit k vyškolenému zaměstnanci, který prošel například týdenním kurzem razítkování dopisů, přes méně než K spolupracovníků. To znamená, že každý vrchol stromu musí ležet do vzdálenosti K od nějakého zaměstnance, který prošel školením.

Na vás je, abyste pro zadaný strom a číslo K vybrali co nejméně vrcholů tak, aby každý vrchol stromu ležel do vzdálenosti K od jednoho z vybraných vrcholů.

Po důkladném prostudování poštovních předpisů a po čtyřiceti minutách přesvědčování, mezitím, co se lidé z fronty vytratili nechtějíce už čekat, pošťačky konečně zjistily, že vlastně žádný předpis, který by zakazoval poslat nula korun, neexistuje, a tak Karel rád zaplatil 10 korun poštovného a složenku poslal. Tím měl odbyto. Možná se problém převedl někam jinam, ale to ho netrápilo.

Ještě by si ale mohl postěžovat do knihy přání a stížností. Přece jenom by si přál, aby 0 korun pošťačku nezaskočilo a netrvalo to tak dlouho. Co to v té knize je? Taková sprostá slova, to si tedy za rámeček nedají.

27-5-5 Kniha přání a stížností — 12 bodů

Jistý pan Václav šel na poštu pro důchod. Byl velice rozladěn, že pošťačkám tak dlouho trvá poslat nějakou složenku a mezitím neobsluhují ostatní zákazníky, že si vyžádal knihu přání a stížností a napsal tam hodně dlouhý text o tom, co si o nich myslí a kam s těmi službami mají zalézt. A nepoužíval přitom nijak slušná slova.

Na vás je, abyste jeho text zkultivovali. Jinými slovy, váš program dostane seznam zakázaných slov, která by se v této situaci neměla používat (slovo je posloupnost symbolů z nějaké abecedy), a vstupní text (to je vlastně také slovo, jen obvykle trochu delší). Máte ze vstupního textu vymazat všechna zakázaná slova (to znamená podslova ze vstupního textu, která jsou zakázanými slovy), ale ani písmenko navíc!

A pozor, tím, že vymažete zakázané slovo, může vzniknout jiné zakázané slovo. To musíte vymazat také. Máte ale zaručeno, že žádné zakázané slovo není prefixem ani suffixem jiného zakázaného slova. Pokud nastane více možností mazání, vymažte to slovo, které končí dříve.

Ukázkový vstup:	*Ukázkový výstup:*
`Vstupní text: aaaaaabbbbbbc` `Zakázaná slova: aaaaaaa, ab`	`c`

Zadání úloh KSP – 5. série

Na ulici před poštou Karel potkal kamaráda Petra.

„Ahoj Karle, to nebudeš věřit, co se mi stalo. Měl jsem u operátora dluh 25 korun. Nechal jsem jim na pobočce stovku, ať si zbytek nechají, a teď mi pořád dokola posílají upomínku na dluh −75 korun. Snad stokrát jsem jim tam volal, ale buďto to nechápou, nebo to neberou. Co myslíš, dá se tady na poště poslat složenka na −75 korun?“

* * *

Tady oba přátele opustíme, jenom bychom chtěli říct, jak k těm chybám nejspíše došlo. Upomínka na nula korun vznikla velmi pravděpodobně zaokrouhlovací chybou.

V pojišťovně měli program, který na peněžní částku používal reálnou proměnnou. Tím sice mohou snadno uložit částky v rozsahu od haléřů po miliardy, problém ale je se zaokrouhlováním. Pojišťovna zřejmě přičítala úroky, což jsou zlomky z částky. Zákazník tak může mít na účtu částku 0.001 koruny i méně, splácet ale může jen jednotky haléřů, takže se nikdy nemůže dostat na nulu. Při tisku upomínek se částka zaokrouhlí na dvě desetinná místa, a tím vznikla upomínka na nula korun.

Další zrádností reálné proměnné s plovoucí desetinnou čárkou je to, že například 1/10 ve dvojkové soustavě nemá konečný zápis, proto se ukládá zaokrouhleně. Když tedy do floatové nuly přičítáte 0.1, dokud se nerovná 1.0, vyrobíte nekonečný cyklus a zrádný program, který nedělá to, co by na první pohled dělat měl. Nebo také `0.1!=1-0.9`*. Při porovnávání reálných čísel na rovnost byste proto měli být velmi obezřetní, nebo se mu raději vyhnout.*

A co dluh −75 korun? Ten vznikl tak, že při upomínání dlužníků v nějaké databázi se částka porovnávala na rovnost s nulou, a těm, co měli nenulový zůstatek, byla automaticky poslána upomínka. To byla chyba v návrhu programu, která se dlouho neprojevila, zákazník totiž jen zřídka platí víc, než musí.

Přesuňme se nyní k roku 2000. Po celém světě se média předháněla ve vymýšlení katastrofických scénářů o tom, co všechno přestane fungovat. Některé země vydaly mnoho peněz na prevenci, tedy na to, aby odborníci ještě jednou prošli zdrojové kódy programů a vyzkoušeli, zda jsou na nové milénium připraveny. Dvě nuly by totiž mohly způsobit problémy s porovnáváním letopočtů.

Přesně 1. ledna 2000 se však nic hrozného nestalo, nanejvýš na některých webových stránkách se objevilo 19100 místo 2000. Média a politici začali spekulovat o tom, zda nebyly vydané prostředky na řešení problému přemrštěné nebo zda problém nebyl smyšlený a úmyslně přehnaný.

⋆ ⋆ ⋆

První týden v lednu Karla probudil telefon.

„Karle, ještě pořád chceš koupit nové auto? Jeď do toho autobazaru v Libni. Teď jsem jel kolem, mají tam za plotem skoro novou pěknou felicii za 150 korun! To neuvěříš!“

Tak se Karel rychle zvedl, vzal peněženku a jel tam.

27-5-6 Autobazar — 10 bodů

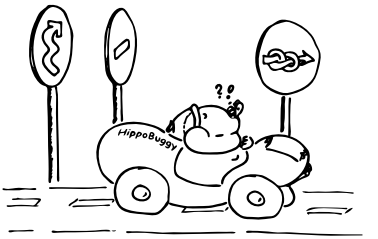
Autobazar, to je jedna dlouhá, dlouhá řada aut. Modré, červené, červené, černé, zelené, fialové, červené... Karel si vzpomněl na hru, kterou někdy hrají děti v autě. Každý počítá jednu svou barvu aut, která potkávají, a kdo jich má nejvíc, vyhraje.

Karla by zajímalo, zda je v autobazaru nějaká barva v nadpoloviční většině, aby s ní tu hru vyhrál proti komukoliv.

V tomhle autobazaru je ale těch barev hodně, taková červená s oranžovým víkem kufru je něco jiného než červená s rezatým nárazníkem. Barvami byste nemohli indexovat paměť počítače.

A těch aut je ještě víc. Takové dlouhé číslo, které udává počet aut, by se vám také nikam nevešlo. Jinými slovy máte k dispozici jenom konstantně mnoho paměti. Můžete ale (i víckrát) procházet řadu aut a dívat se, jakou mají barvu.

Popište postup, jak zjistit, zda je v posloupnosti barev délky N alespoň $N/2$ výskytů nějaké barvy. Použít smíte ale jen konstantní paměť a lineární čas.



Zadání úloh KSP – 5. série

Bohužel, měli zavřeno. Cedulky s cenami dával na auta člověk a ten tu chybu odhalil dřív, než nějaké auto prodali. Jejich program totiž měl nějaká kritéria, jak určovat ceny aut. A tím prvním byla doba od poslední technické kontroly. On věděl, že je rok 2000, ale myslel si, že technická prošla krátce po roce 1900, a tak ta stará auta doporučoval prodat za pár korun.

Poté ještě Karlově stoleté prababičce přišla pozvánka na kojeneckou prohlídku, tím byl konec s problémem roku 2000. S programátorskými chybami ale bohužel konec jenom tak nebude. Chyby dělají lidé, ne zlomyslné počítače, a lidskou chybu nemůžete úplně vyloučit, dokud z programování nevyloučíte lidský faktor.

Důležité je se z chyb poučit. K podobnému problému by mohlo v budoucnu dojít ještě jednou. Mnohé programy používají znaménkový 32-bitový typ, který reprezentuje čas jako počet vteřin od 1. 1. 1970, a ten přeteče 19. ledna 2038. Ale pokud do té doby začnou používat 64-bitový typ, přesouvá se problém do roku 292 277 026 596.

To ale neznamená, že nemáte řešit KSP. S chutí do toho!

Příběh vyprávěl

Dominik Macháček

Seriál – UNIX

27-1-7 Učíme se s UNIXem — 14 bodů

V letošním seriálu jsme se rozhodli trochu vás seznámit s UNIXovými systémy, přesněji hlavně s tím, jak efektivně (a efektně) používat jejich příkazovou řádku. Naučíme vás, že UNIXová příkazová řádka je kamarád, kterého se nemusíte bát – neoplývá sice (většinou) klikacím barevným rozhraním, kterým vás provede virtuální pomocník v podobě *pana Sponky*,[16] ale o to je mocnější.

KSP

Seriál bude směřován hlavně praktickým způsobem a jednotlivé úkoly by vás měly naučit UNIX skutečně používat. Nejprve se ale nevyhneme malému historickému úvodu o vývoji UNIXu a různých shellů, hlavně bashe.

seriál

UNIX, POSIX, shell, bash, ... Co to všechno je?

Historie UNIXu se začala psát v roce 1969, kdy v Bellových laboratořích vznikl tak trochu potají nový operační systém (vedení firmy v nově vyvíjeném operačním systému tehdy nevidělo velkou budoucnost, a tak ho jeho vývojáři před vedením vydávali za textový editor, aby na vývoj získali čas a zdroje). Teprve začátkem 70. let dostal systém oficiální podporu a začal se překotně vyvíjet.

UNIX je ale v současnosti registrovaná ochranná známka a mohou ji využívat pouze systémy, které splňují určené podmínky a k tomu platí licenční poplatky (navíc licenci získává vždy jen určitá verze systému, a licencování je tak pro rychle se vyvíjející systémy finančně i časově neúnosné).

Z tohoto důvodu vzniklo několik systémů, které jsou od UNIXu pouze odvozené. Všechny ale splňují společný standard zvaný POSIX,[17] který zaručuje vzájemnou kompatibilitu, a obecně se o nich mluví jako o UNIXových systémech. Nejrozšířenějším z nich je Linux, který sám existuje v záplavě různých variant (tzv. *distribucí*). Dalším známým systémem je například BSD vyvíjený na Kalifornské univerzitě v Berkeley.

Všechny tyto systémy ale spojuje příkazová řádka, ve světě UNIXu se jí říká *shell* a běží v *terminálu*. Terminál je přímo věc, která se stará o čtení vstupu z klávesnice a zobrazení výstupu na monitor, ale nemá žádnou vnitřní logiku, o tu se stará shell (který se zase nemusí zajímat o klávesnici a monitor, ale má už jen svůj *standardní vstup a výstup*).

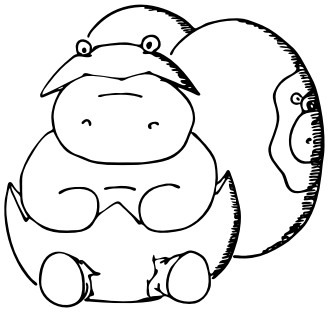
Shell je jednoduše řečeno textové rozhraní, které umožňuje spouštět příkazy a pomocí nich ovládat celý systém. Je to takový předchůdce grafických rozhraní a existují stroje (například některé servery), kde grafické rozhraní vůbec není nainstalováno a celé ovládání se děje právě jen přes shell.

[16] personifikovaná nápověda v jednom nejmenovaném kancelářském balíku

[17] `http://cs.wikipedia.org/wiki/POSIX`

Ale i shell sám existuje v několika různých variantách, i on se postupně vyvíjel a byl obohacován o nové vlastnosti a příkazy. Shell, se kterým se dneska setkáme skoro na všech linuxových strojích, se nazývá *bash* (zkratka za *Bourne again shell*, což je odkaz na starší *Bourne shell*). V něm se budeme pohybovat většinu času, ale pokusíme se zdůrazňovat, které příkazy jsou univerzální a budou fungovat ve všech POSIXových shellech, a které jsou jen specialitou bashe.

Jak si bash pořídit?



Pokud máte Linux, skoro určitě máte bash nainstalovaný. Stačí ve vašem systému pouze spustit *Terminál* (či nějak podobně nazvanou aplikaci) a objeví se vám (většinou černé) okno, kam je možné psát příkazy a prohlížet si jejich výsledky. Je ale možné, že na svém stroji nebudete mít nastavený bash jako výchozí a spustí se vám nějaký jiný, jednodušší, shell. V takovém případě v něm jen spusťte příkaz `exec bash` a jste v bashi.

Ve Windows je situace o trochu složitější, ale i zde si můžete UNIXový bash pořídit (organizátoři KSP ho při práci ve Windows doporučují jako věc, která vám usnadní život). Nejlepším řešením bude instalace programu *Cygwin*, který vám nainstaluje bash a spoustu šikovných utilit. Jeho instalaci máme popsanou v naší Encyklopedii.[18]

První kroky po systému

Než vůbec uděláme první krok, měli bychom si v rychlosti představit UNIXový souborový systém. Jeho hlavní myšlenkou je, že vše je uspořádáno ve stromu, do kterého se na různá místa zapojují správné věci, třeba různé disky. *Kořen* souborového systému se označuje jako adresář `/` (lomítko), domovským adresářem uživatele `hroch` pak adresář `/home/hroch`. Je klidně možné, že celý adresář `/home` sídlí fyzicky na jiném disku (třeba i síťovém), který se do stromu souborů připojil na správné místo.

Názvy souborů i adresářů mohou tvořit libovolné znaky (mezery, písmena s diakritikou, ...) s jedinou výjimkou, a tou je lomítko, to se totiž používá pro oddělování názvů adresářů v cestě. Nedoporučujeme ale vytváření názvů obsahujících různé speciální znaky jako `[]"?*` a podobně (může se dokonce stát, že některé programy a systémy budou – pro vaši ochranu – vytváření takových souborů blokovat).

V názvech také záleží (na rozdíl od operačních systémů Windows) na velikosti písmen, `fotka.jpg` a `fotka.JPG` jsou rozdílné soubory.

[18] `http://ksp.mff.cuni.cz/encyklopedie/cygwin.html`

Když poprvé spustíme bash, uvidíme podobný řádek:

```
hroch@ksp:~$
```

Ten nám říká náš login (jméno, pod kterým jsme se přihlásili), stroj, na kterém náš bash běží, a také nám prozrazuje jméno adresáře, ve kterém se aktuálně nacházíme. Za znakem `$` pak můžeme psát příkazy. Ale počkat, co je vlnka za adresář a kde je připojený ve stromu souborů? Není to nic tajuplného, je to jen zkratka za náš domovský adresář, tedy za `/home/hroch`, aby ve výpisu nezabíral tolik místa.

Začneme se shellovým Hello World: Napište příkaz `echo Hello World` a spusťte ho klávesou Enter. Příkaz `echo` (anglicky „ozvěna“) udělá to, že opíše všechny své parametry na svůj výstup.

Většina příkazů v shellu totiž může být ovlivněna zadanými parametry, některé příkazy bez zadaných parametrů dokonce ani nefungují. Parametry jsou od sebe a od příkazu v shellu odděleny mezerami a rozlišujeme dva základní typy parametrů – přepínače a poziční argumenty. K jejich pořádnému vysvětlení se dostaneme za chvíli, zatím si pojďme ještě chvíli hrát.

Chcete nějaký zajímavější příkaz než `echo`? Zkuste si spustit příkaz `pwd`. Tento příkaz vypíše cestu do aktuálního adresáře (je to zkratka za „print working directory“). Jak se ale přepnout do nějakého jiného? K tomu slouží příkaz na změnu adresáře `cd` („change directory“), tomu do parametru můžeme napsat, kam nás má přepnout (bez parametru nás přepne do našeho domovského adresáře).

Zkuste si třeba `cd /` nebo `cd /home`. Cestám zapsaným s lomítkem na začátku říkáme *absolutní* a udávají přesné místo v adresářovém stromě, kam se přesunout. Druhou možností je použít cestu *relativní* vzhledem k pracovnímu adresáři, ta se píše bez úvodního lomítka. Například spuštění `cd hroch` v adresáři `/home` nás přesune do `/home/hroch`. Můžeme si to představit tak, že před takovouto cestu se automaticky připojí výstup příkazu `pwd` a lomítko.

Speciálním „podadresářem“, který se vyskytuje všude, je adresář `..` (dvě tečky). To je odkaz ukazující o úroveň výš, když tedy ve svém domovském adresáři spustíme `cd ..`, přepne nás to do adresáře `/home`. Dalším speciálním adresářem je adresář `.` (tečka), která odkazuje na aktuální adresář – to vám teď může připadat matoucí a nadbytečné, ale v dalších dílech ukážeme, na co se tento odkaz používá.

Poslední, co k pohybu po systému potřebujeme, je příkaz, který by nám vypsal obsah aktuálního adresáře. To je příkaz `ls` (od anglického slovesa „list“). V základní verzi nám vypíše všechny adresáře a soubory v aktuálním adresáři kromě skrytých (začínajících tečkou), později se s ním naučíme některé další šikovné věci.

Pokud si budete chtít nějaký ze zmíněných příkazů více prostudovat, můžete použít **manuálové stránky**. Pro zobrazení manuálových stránek k příkazu `abc`

zadejte v bashi příkaz `man abc`, a pokud k příkazu `abc` tato stránka existuje, zobrazí se vám (k příkazům `cd` a `pwd` ale v některých systémech manuálové stránky neexistují, jelikož se jedná o interní příkazy shellu a ne o samostatné programy).

V manuálové stránce je většinou uvedený základní popis příkazu, možné parametry a občas i ukázkové použití. Pro ukončení prohlížení a návrat do shellu stiskněte `q`. Doporučujeme si zběžně pročíst manuálové stránky dále zmiňovaných příkazů, mohou se vám hodit.

Vytváření a mazání adresářů a souborů

Již umíme procházet po adresářích, pojďme si také nějaké vytvořit a smazat:

- Vytvoření prázdného adresáře provedeme zavoláním příkazu `mkdir` *název adresáře*.
- Smazání adresáře (musí být prázdný) uděláme pomocí příkazu `rmdir` *název adresáře*, nesmíme se v tu chvíli ale nacházet uvnitř tohoto adresáře.
- Vytvoření prázdného souboru můžeme provést pomocí příkazu `touch` *název souboru*. Pokud takový soubor neexistuje, příkaz ho vytvoří; pokud existuje, nastaví mu datum modifikace na aktuální okamžik.
- Smazání souboru zařídíme zavoláním `rm` *název souboru*.

Všechny názvy můžeme uvádět i jako cesty, zavolání `touch adresar/soubor` vytvoří `soubor` v `adresar`, pokud `adresar` již existuje (jinak skončí zavolání chybou). Pokud chceme najednou spustit více příkazů, dá se to provést jejich zápisem na jednu řádku a oddělením pomocí středníku.

Úkol 1 [2b]: Vytvořte prázdný soubor `test` umístěný v nově vytvořeném podadresáři `~/a/b/c/d` (vlnka tu, jak je zvykem, zastupuje váš domovský adresář). Pak zase adresáře i soubor vymažte. Zkuste použít co možná nejméně příkazů.

U všech úloh odevzdávejte posloupnost příkazů vedoucích ke splnění dané úlohy (pokud nebude uvedeno jinak).

Ještě doplníme další tři příkazy, které souvisejí s prací se soubory:

- Příkaz `cp` (zkratka za *copy*) slouží ke zkopírování (klidně více) souborů do zadaného umístění. Původní soubory zůstanou na místě a v cílovém umístění se vytvoří jejich kopie.
- Příkaz `mv` (zkratka za *move*) dělá podobnou věc jako `cp`, jen soubory nekopíruje, ale přesouvá (a umí přesouvat i adresáře). Stojí za poznámku, že pokud je původní i cílové umístění na stejném fyzickém disku, je `mv` řádově rychlejší, než kombinace `cp` a `rm` – jen se totiž upraví záznam v tabulce souborů a data se fyzicky nemusejí nikam přesouvat (dalo by se říci, že vlastně dojde jen k přejmenování a přepsání adresy).
- Příkaz `cat` (zkratka *concatenate*) vypisuje obsah zadaných souborů na terminál. Hodí se jednak pro prohlížení obsahu krátkých souborů, jednak pro

svůj původní účel (konkatenaci – zřetězení). Když zadáme více názvů souborů, `cat` je všechny spojí a vypíše na terminál (později se dozvíme, jak výstup na terminál přesměrovat někam, kde se nám hodí víc).

Příkazy `cp` a `mv` přebírají libovolně mnoho parametrů: vezmou všechny parametry až na poslední jako zdrojové soubory a zkopírují/přesunou je do místa, kam odkazuje poslední parametr. Teď by stálo za to pořádně si rozebrat, co všechno může být obsaženo v parametrech a jaké triky s nimi umíme.

Parametry: Přepínače a poziční argumenty

Přepínače jsou, jak již název napovídá, parametry, které upravují nějakým způsobem běh příkazu. Jsou uvozeny jednou nebo dvěma pomlčkami (je zvykem, že jednou pomlčkou jsou uvozeny jednopísmenné a dvěma pomlčkami vícepísmenné, ale neplatí to vždy).

Například nám již známý příkaz `ls` má přepínač `-l` zapínající dlouhý výstup. Když si tedy spustíme příkaz `ls -l`, vypadne na vás pravděpodobně výpis podobný tomuto:

```
drwxr-xr-x  hroch ksp 4096 čen 16 12:00 adresar
-rw-r--r--  hroch ksp    0 čen 16 12:00 soubor
```

Zde se dozvíme (popořadě) přístupová práva k souboru, jeho vlastníka a skupinu (těmito věcmi se budeme zabývat v některém z příštích dílů), velikost, datum poslední změny a na úplném konci nalezneme název.

Při zápisu více přepínačů je můžeme psát buď všechny samostatně (`prikaz -a -b -c`), nebo můžeme jednopísmenné i sdružit dohromady (`prikaz -abc`), fungovat budou stejně. Pokud nějaký přepínač bude přijímat doprovodný parametr (většinou to je číslo), může zápis vypadat třeba takto: `prikaz -ab 3 -c` (tady přepínač `b` přijal parametr 3), nebo dokonce `prikaz -acb3`.

Pořadí přepínačů je u většiny základních příkazů libovolné (když nebude, upozorníme vás), ale u některých programů, které nejsou napsané tak pečlivě jako základní shellové příkazy, na jejich pořadí záležet už může. V takovém případě doporučujeme pročíst manuál od daného programu.

Druhý typ parametrů nazýváme **poziční argumenty**. Ty se zadávají bez nějakých uvozujících pomlček a tradičně až za všemi přepínači. Často to jsou například názvy souborů a adresářů (viz příkazy `cp` a `mv`).

Zvídavější z vás možná ve spojitosti s výše jmenovanými příkazy napadla jedna otázka: „Jak zkopírovat/smazat soubor s mezerou v názvu?“ Na to se dá jít dvěma způsoby:

- Speciální znaky (mezi něž patří i mezera) můžeme *escapovat*, tedy zbavit je jejich speciálního účinku, předřazením zpětného lomítka. Na příklad napsáním `rm deravy\ nazev` tedy předáme příkazu `rm` jediný parametr – název souboru obsahujícího mezeru.

- Druhou možností je použití uvozovek, příkaz výše bychom mohli přepsat na `rm "deravy nazev"` se stejným účinkem. Použít se dá i zápis s jednoduchými uvozovkami (apostrofy) a v tomto případě by byly ekvivalentní, rozdíl je však v tom, že ve dvojitých se expandují proměnné, kdežto v jednoduchých ne (více o proměnných v příštích dílech).

Poslední důležitou věcí ohledně parametrů je, jak oddělit přepínače od pozičních argumentů. Představme si například, že bychom chtěli smazat soubor, který by se jmenoval „`-f`“. Nemůžeme napsat jenom `rm -f`, protože to se vyhodnotí jako přepínač, a ani `rm "-f"` nám nepomůže, protože bash stejně předá příkazu `rm` jenom parametr `-f`.

Řešením je oznámit příkazu místo, kde končí přepínače. To uděláme pomocí osamocené dvojice pomlček. Za tímto místem se již nemohou nacházet žádné přepínače a příkaz všechno zbylé vyhodnotí jako poziční argumenty. Řešení tedy vypadá takto: `rm -- -f`.

Úkol 2 [2b]: Prostudujte si manuálové stránky příkazů `head` a `tail`, hlavně jejich parametry, a zjistěte, jak vypsat prvních a posledních patnáct řádků souboru a jak vypsat všechno až na prvních patnáct řádek. Vyzkoušejte si to třeba na souboru `/etc/passwd`.

Doplňování a wildcardy

Představme si, že se chceme přepnout do adresáře, jenž má hrozně dlouhý název. Bash nám to dokáže usnadnit: Pokud totiž při psaní názvu zmáčkneme tabulátor, pokusí se doplnit (podle již napsaného začátku) zbytek názvu souboru nebo adresáře.

Pokud existuje několik souborů nebo adresářů, které mají stejný prefix jména, doplní bash po stisku tabulátoru nejdelší společnou část a po dalším stisku zobrazí jména, kterými se dá pokračovat. Pak stačí jen napsat další část názvu, opět stisknout tabulátor a nechat si doplnit zbytek. Věřte, že to řádově urychlí pohyb po adresářovém stromě a je to jedna z nejpoužívanějších kláves. :-)

Dalším dobrým trikem jsou šipky nahoru a dolů, které nám dovolí listovat v historii příkazů a znovu je spouštět nebo upravovat.

Dobře, teď již umíme s bashem pracovat efektivněji, ale co když budeme chtít zkopírovat stovky souborů (třeba fotek z výletu), to je musíme vážně všechny vypisovat? Bash nám pomůže i v tomto případě, užitím zástupných značek neboli *wildcardů*.

Wildcardy fungují jako žolíky, umožní nám nahradit část názvu souboru zástupným znakem. Ve skutečnosti se stane to, že bash najde všechny soubory, které odpovídají použitým zástupným znakům, a nahradí jimi výraz s wildcardy v příkazu (říkáme, že se *expanduje* na tyto soubory). Samotný příkaz tedy nevidí wildcardy, ale dostane od bashe rovnou seznam odpovídajících souborů. Jedinou výjimkou je, když žádné takové soubory neexistují, v takovém případě nechá bash výraz s wildcardy beze změny.

Mezi wildcardy patří:

- Otazník `?` zastupuje libovolný znak: `mal?.txt` tedy odpovídají například soubory `mala.txt`, `maly.txt` a `male.txt` (naopak `mal.txt` neodpovídá).
- Hvězdička `*` zastupuje libovolný (i nulový) počet nějakých znaků: `fot*.jpg` odpovídají `foto001.jpg` i `fot.jpg`. Speciální výjimkou jsou skryté soubory, na ty se wildcardy neexpandují (pokud explicitně nenapíšeme tečku na začátku názvu).
- Hranaté závorky `[]` se používají pro výčet nebo rozsah: výrazu `[13579]` bude odpovídat libovolná lichá číslice, výrazu `[0-9]` libovolná číslice z rozsahu 0 až 9 a výrazu `[0-9A-F]` zase libovolná šestnáctková číslice. Pokud jako první znak v závorce uvedeme stříšku, funguje celá závorka jako negace (výrazu `[^0-5]` odpovídá všechno až na číslice 0 až 5). Stejného efektu docílíme ve většině moderních shellů také vykřičníkem.

Další speciální konstrukcí shellu, která se často kombinuje s wildcardy, jsou složené závorky `{}`. Nejsou to wildcardy, takže se vůbec nedívají na to, jestli nějaké jimi popisované soubory existují nebo ne, ale daly by se přirovnat spíše k syntaktické zkratce.

Jejich zápis je tvořen několika výrazy oddělenými čárkami (ve spojení s wildcardy například `*.{jpg,mp[34]}`) a bash udělá to, že ještě před zpracováním klasických wildcardů rozepíše výrazy obsahující složené závorky na všechny jejich možné varianty – vytvoří samostatný výraz pro každou z variant uvedených ve složené závorce (z příkladu výše tak vznikne dvojice `*.jpg *.mp[34]`, která se teprve zpracovává dál).

Pokud budeme pracovat v bashi (jiné shelly podobnou funkci obecně mít nemusí, i když zase mohou obsahovat jiná vylepšení), můžeme ve složených závorkách použít i rozsah. Zápis `{1..20}` je ekvivalentní s vypsáním dvaceti čísel oddělených čárkou ve složených závorkách.

Rozdíl oproti wildcardům se dá pozorovat třeba mezi chováním příkazů `mkdir adresar{5,6,7}` a `mkdir adresar[567]`. První provede, co bychom od něj očekávali (vzniknou tři nové adresáře), ale druhý pro neexistenci daných adresářů vytvoří adresář s hranatými závorkami v názvu (wildcardy se neexpandují).

Pro zkoušení wildcardů v nějakém adresáři můžete použít příkaz `echo`, který vám vypíše všechno, co dostane jako parametry – tedy všechny expandované názvy souborů v aktuálním adresáři, nebo původní wildcard, pokud se expanze nepovedla.

Úkol 3 [1b]: Jak byste smazali soubor, který obsahuje v názvu nějaký wildcard? Například jak byste smazali soubor `a?c` a přitom nesmazali `abc`, nebo smazali `adresar[567]`, ale již ne existující `adresar5`?

Úkol 4 [3b]: Vymyslete co nejkratší zápis pomocí wildcardů, kterému budou odpovídat právě všechny soubory s příponami `jpg`, `jpeg` nebo `gif` v podadresářích aktuálního adresáře; názvy podadresářů musí obsahovat buď alespoň dvě číslice 0 až 9 nebo alespoň dvě písmena anglické abecedy (pozor na velikost písmen). Důkladně popište, co která část výrazu dělá. Můžete předpokládat, že od každého typu bude alespoň jeden soubor a adresář existovat.

Roury a přesměrování vstupu a výstupu

Kromě parametrů pracují ještě shellové příkazy se vstupem a výstupem. Parametry většinou slouží k nastavení chování příkazu, kdežto data, která chceme příkazem zpracovat, patří na jeho vstup.

První metodou zadávání vstupu je přímo jeho psaní na terminál. Zkuste si spustit například příkaz `wc` (zkratka za *word count*). Ten slouží k počítání slov, písmen a řádek souborů, které dostane jako parametry, což ale teď nebudeme používat – bez pozičních argumentů totiž `wc` provádí to samé se svým standardním vstupem a na svůj standardní výstup vypisuje výsledek. Ve výchozím nastavení směřují standardní vstup i výstup na terminál.

Když `wc` spustíme, můžeme psát libovolný text, Zadávání ukončíme speciálním znakem `EOF` (*End-of-file*), který napíšeme stiskem Ctrl+D na prázdném řádku. Tím ukončíme vstup a `wc` provede svoji práci – vypíše dané počty. Pokud bychom chtěli příkaz ukončit bez toho, aby vypsal výsledek, můžeme ho násilně zastavit pomocí Ctrl+C.

Toto ale není příliš praktické použití, mnohem lepší by bylo přesměrovat na vstup nějaký soubor. To uděláme pomocí operátoru šipky: Když třeba zapíšeme `wc < soubor.txt`, tak přesměrujeme obsah zadaného souboru na standardní vstup příkazu `wc`. Zkuste si to. Stejně tak můžeme zápis převrátit a jako první napsat přesměrování. Dokonce nemusíme ani okolo operátoru přesměrování psát žádné mezery (lze tedy psát `<soubor.txt wc`). Syntaxe je dost volná, stačí si zvolit styl, který se vám bude nejvíce líbit.

Stejně jako vstup můžeme přesměrovat i výstup, to uděláme šipkou ukazující na druhou stranu, směrem k souboru. Příkaz `ls > seznam.txt` přepíše soubor `seznam.txt` a uloží do něj výstup z příkazu `ls`, na terminálu se v takovém případě žádný výstup neobjeví. Kdybychom namísto přepsání souboru chtěli jenom připojit nové řádky na jeho konec, můžeme použít dvojitou šipku, takto: `ls >> seznam.txt`.

Co když budeme chtít použít výstup jednoho příkazu jako vstup pro druhý? Určitě bychom mohli použít pomocný soubor, tedy bychom mohli psát na příklad `a > tempfile ; b < tempfile`, ale shell nám nabízí mnohem elegantnější věc, a tou je takzvaná **roura.**

Roura se zapisuje jako svislá čára (na anglické klávesnici ji najdeme nad Enterem) a funguje tak, že vezme standardní výstup příkazu nalevo od sebe a použije ho jako vstup pro příkaz napravo. Zápis `a < soubor | b | c` (nebo ekvivalentní `<soubor a|b|c`) znamená to, že spustíme příkaz `a`, kterému předáme na vstupu soubor a jeho výstup použijeme jako vstup pro příkaz `b`. Výstup příkazu `b` pak použijeme jako vstup pro `c` a výstup `c` zobrazíme na terminálu. Jednotlivé programy přitom běží současně a mezivýsledky si rovnou předávají, ty tedy nezabírají žádné místo na disku.

Pokud budete používat přesměrování do souboru, dejte si ale pozor na jednu věc. Nelze najednou načítat a zapisovat stejný soubor, shell totiž jako první smaže původní soubor (pokud používáme přesměrování jednou šipkou), a pak teprve by se ho pokoušel načíst. Pozor na to hlavně při úpravách již hotového textu, dá se takto nenávratně smazat několikahodinová práce.

Úkol 5 [1b]: Do souboru `datum` vypište na první řádek „Dnes je:“ a na druhý aktuální datum a čas v jakémkoliv formátu. Asi vám k tomu pomůže příkaz `date` a pro zkontrolování obsahu souboru můžete použít buď již zmíněný `cat`, nebo prohlížeč souborů `less` – ten se ukončuje stiskem `q`.

Úkol 6 [1b]: Spočtěte nějakým příkazem počet adresářů a souborů ve svém domovském adresáři (kromě skrytých).

Úkol 7 [2b]: Napište příkaz využívající roury, který ze souboru `soubor.txt` vezme jedenáctou až třicátou řádku včetně a spočítá počet slov na nich. Můžete předpokládat, že `soubor.txt` je dostatečně dlouhý.

Úkol 8 [2b]: Vypište velikost v bajtech všech souborů v aktuálním adresáři, které obsahují v názvu alespoň jednu číslici (nezapomeňte na skryté soubory).

Závěr

Dnes jsme si prošli základní příkazy a principy použitelné v bashi. Zopakujme si všechny příkazy, které jsme se naučili:

- Navigace: `pwd`, `cd`, `ls`
- Manipulace se soubory: `touch`, `cp`, `mv`, `rm`, `mkdir`, `rmdir`
- Obsah souborů: `cat`, `head`, `tail`, `wc`
- Další: `echo`, `less`, `date`
- Manuál: `man`
- Wildcardy, roury a přesměrování vstupu a výstupu.

Příští díl se již můžete těšit na některé pokročilejší UNIXové techniky, my se budeme těšit na vaši účast. :-)

Jirka Setnička

27-2-7 Shellové skripty — 12 bodů

V minulém dílu jste se naučili pár základních příkazů shellu. Víte, jak vypadá adresářová struktura systému, a umíte si trochu hrát se soubory. Po přečtení tohoto dílu seriálu byste měli být schopni použít shell jako plnohodnotný programovací jazyk. Naučíte se vytvořit *skript,* používat proměnné a podobné věci.

Skripty

Možná víte, že programovací jazyky se dají rozdělit na kompilované a interpretované. Programem v interpretovaném jazyce je samotný zdrojový kód, který si interpret přečte a provede. A protože je shell mocný nástroj, umí sloužit jako interpretovaný jazyk.

Skript je tedy obyčejný textový soubor, který obsahuje příkazy pro shell. Otevřete si svůj oblíbený editor a v něm třeba soubor `skript.sh`. Koncovka souboru není rozhodně nutná, název souboru je informací jen pro uživatele. Do něj můžete napsat třeba toto:

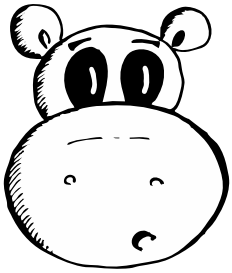
```
echo "Jsi v adresáři:"
pwd
```

Věřím, že poznáte, co skript dělá. Pokud ne, můžete ho zkusit spustit:

```
hroch@ksp:~$ bash skript.sh
Jsi v adresáři:
/home/hroch
```

Takto spustíme nový shell. Pokud dáme shellu jako první poziční argument existující soubor, spustí jej. To je pěkné, ale není to „program“. Aby šlo skript spouštět samostatně, musíte ještě pár věcí zařídit.

Minule jsme si říkali, že soubory mají nějaká práva. Podrobnější rozepsání očekávejte v příštích dílech, dnes jen krátce. Každý soubor má různá práva pro svého vlastníka, skupinu a ostatní. Tři základní práva jsou Read, Write a eXecute, neboli čtení, zápis a spuštění. Aby byl shell ochoten soubor spustit, musíte mít právo spuštění a samozřejmě čtení.



Na změnu práv souboru použijte příkaz `chmod`. V prvním argumentu lze říct o jaká práva jde, dalšími jsou pak změněné soubory. Následující příklad učiní náš skript spustitelným:

```
hroch@ksp:~$ ls -l skript.sh
-rw-r--r-- hroch ksp 29 1. říj 12:00 skript.sh
hroch@ksp:~$ chmod +x skript.sh
hroch@ksp:~$ ls -l skript.sh
-rwxr-xr-x hroch ksp 29 1. říj 12:00 skript.sh
```


Ještě musíme říct systému, čím že to má skript spustit. Doplňte tuto konstrukci na první řádek souboru:

```
#!/bin/bash
```


Za normálních okolností značí mříž (`#`) na začátku řádku komentář, spolu s vykřičníkem na prvním řádku ale říká, jaký program se má spustit k interpretaci daného souboru. Takto už systém ví, že má použít program `/bin/bash`, tedy bash.

Při psaní příkazu v shellu jsme zatím první slovo na řádku označovali jako *příkaz.* Není to ale úplně pravda – prvním slovem je název souboru nebo interní příkaz shellu. Shell má několik adresářů, ve kterých programy hledá, pokud nejsou zadány s cestou. Jedním z nich je určitě `/bin`, takže když napíšete jen „bash“, shell si domyslí `/bin/` a spustí `/bin/bash`.

Pokud ale dáte na začátek řádky soubor i s adresou, shell si nic domýšlet nebude a zadaný soubor prostě spustí (pokud je spustitelný).

```
hroch@ksp:~$ /home/hroch/skript.sh
```

Možná si vzpomenete na všudypřítomný adresář `.` (tečka). Teď si ukážeme, k čemu se dá využít. Pokud byste se pokusili spustit jen příkaz

```
hroch@ksp:~$ skript.sh
```

tak se asi nic nestane. Většina shellů z bezpečnostních důvodů nehledá spustitelné soubory v aktuálním adresáři. Je tedy nutné shellu zadat plnou cestu k souboru – no a abychom nemuseli psát celou cestu, použijeme adresář tečka. Ten totiž odkazuje na adresář, ve kterém je umístěn. Můžeme toho využít i zde:

```
hroch@ksp:~$ ./skript.sh
```

Procesy

Na chvilku odbočíme od skriptování k samotnému UNIXu. Jistě jste si všimli, že v systému může běžet více programů současně. Dokonce můžete spustit jeden program dvakrát, třeba s jinými parametry. Jak tohle systém dělá?

Každý běžící program je zabalen do struktury, které říkáme *proces.* V ní najdeme například PID (Process ID, identifikátor procesu). To je nějaké číslo, unikátní pro každý běžící proces. Jakmile dojdou volná PID, systém vám již nedovolí další proces spustit.

Dále si proces pamatuje uživatele, pod kterým běží, stav, ve kterém se nachází, PID svého otce nebo třeba terminál, ke kterému je připojen.

Jak se ale takový proces vytváří? Předně, vyrobit „čistý“ proces není možné. To udělá při startu systému jádro, které spustí program `init`. Nadále se procesy můžou jen kopírovat.

Takové operaci se říká *fork* a můžete si ji představit jako rozdvojení. Před zavoláním systémového volání *fork* jste měli jeden proces, po něm máte dva. Úplně stejné, až na PID. Jednomu zůstane a stává se rodičovským procesem, ten druhý má PID nové a stává se synem.

Pro vypsání procesů můžete použít příkaz `ps`. Jeho parametry nejsou dané normou, ale něco jako `ps ax` by mělo vypsat všechny procesy běžící v systému.

Dalším systémovým voláním je *exec*, kterým se ukončí váš program a nahradí se jiným. Například, pokud v terminálu spustíte program, váš shell provede *fork*. Tím vznikne kopie shellu, která vzápětí zavolá *exec* na váš program.

Každý uživatelský proces má tedy svého otce. Může od něj dostat nějaké informace, třeba v paměti před *fork*em. Zpět je to ale složitější. Dokud oba běží, existují prostředky meziprocesové komunikace. Jak ale oznámit otci „Je mi líto, chyba, končím?“

Když proces ukončí svůj běh, není okamžitě vymazán. Místo toho mu jádro vystaví úmrtní list a čeká, než si ho jeho otec vyzvedne. Pokud mezitím skončil také otec, vyzvedne si ho `init`. Pokud otec běží, ale na svého syna zapomněl, bude syn čekat v paměti navěky. Operaci vyzvednutí můžete najít jako systémové volání *wait*.

Součástí ukončeného procesu je návratová hodnota. To je jednobajtové číslo (0–255) a jeho interpretace je čistě na otci. Můžete si předávat klidně výsledek nějaké operace, ale k tomu je jeden bajt docela málo. Místo toho se návratová hodnota používá pro signalizaci úspěchu nebo chyby. V UNIXovém prostředí je zvykem, že nula značí úspěch, nenula nějakou chybu.

Proměnné

Když už máme za sebou spuštění prvního skriptu, můžeme se podívat na jeho obsah. Na každém řádku leží příkaz se svými přepínači a argumenty. Jednotlivé příkazy může oddělovat také středník. Jako příkaz může sloužit i shellový skript.

Opravdová legrace začne až s proměnnými. Shell má proměnné z různých důvodů pouze typu řetězec. Nelze s nimi provádět nic zázračného, lze do nich jen zapsat hodnotu nebo naopak proměnnou *expandovat* (nahradit, použít, ...). Tyto dvě operace ale stačí ke všem myslitelným potřebám.

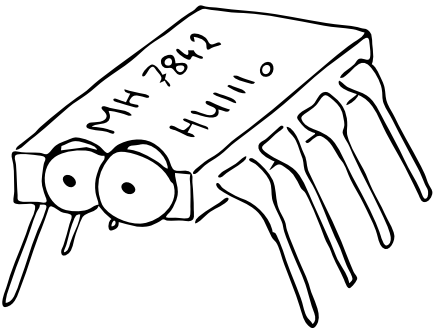
```
prom=ksp
echo $prom
```

Na prvním řádku jsme do proměnné `prom` přiřadili „ksp“. Důležité je, že před rovnítkem není mezera – podle toho shell pozná, že chceme přiřadit do

proměnné a ne spustit příkaz `prom`. Všimněte si také, že součástí názvu proměnné není znak `$`.

Na druhém potom spouštíme příkaz `echo`. Znak `$` říká „vlož sem proměnnou“. Shell nejprve vyhledá odpovídající proměnnou a vloží ji na dané místo ještě před spuštěním příkazu.

Tomuto procesu se říká expanze a popíšeme si jej přesněji. Na začátku má shell řádku, která mu byla zadána. Tu rozdělí poprvé na *slova,* posloupnosti nebílých znaků, případně části ohraničené uvozovkami. Poté provede nahrazení složených závorek `{}` a tildy `~` (viz minulý díl). Pak přijdou na řadu proměnné. Bash nabízí spoustu rozšíření expanze proměnných, odvážným doporučuji podívat se do manuálu. Neexistující proměnné jsou prázdné řetězce.



Další krok expanze si popíšeme podrobněji. Často se hodí si někam schovat výstup nějakého příkazu. Už to umíte se soubory. Bez souborů to lze s použitím jedné z těchto dvou konstrukcí pro substituci výstupu:

```
cas=‘date‘
cas=$(date)
```

Zpětný apostrof (backtick) je kratší a používanější zápis, má ale několik nevýhod. Ty se projeví zejména pokud plánujete volání zanořit, jako v následujícím, trochu umělém, příkladě:

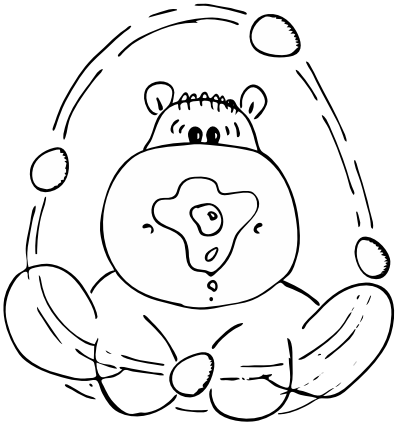
```
cas=‘date +\‘cat format-\\\‘date +%Y\\\‘\‘‘
cas=$(date +$(cat format-$(date +%Y)))
```

Ten nejprve zjistí, jaký je rok, poté přečte soubor, např. `format-2014`, a jeho obsah použije jako formátovací řetězec pro příkaz date. Zpětné apostrofy se musí escapovat, a escapovací lomítka na druhé úrovni také. Kolik zpětných lomítek musíte použít v třetí úrovni?

V dalším postupu expanze se jednotlivé shelly liší. Společné je jen další rozdělení výsledků předchozích expanzí na slova. Na to pozor, kdekoli použijete

proměnnou, její obsah se znovu rozdělí. Pokud chcete použít obsah proměnné jako jeden argument, je potřeba zapsat ji jako `"$prom"`, nestačí jen `$prom`.

Následně proběhne vyhodnocení wildcardů a odstranění escapovacích zpětných lomítek. Až na konci tohoto procesu shell vezme první slovo, spustí jej a předá zbylá slova jako argumenty.



Teď už si můžeme říct pořádně, jaký je rozdíl mezi použitím uvozovek ", apostrofů ’ a ničeho. Pokud napíšete text jen tak, použijí se všechny expanze. V uvozovkách se ztrácí význam všech speciálních znaků vyjma `$`, ‘ a \, takže se použijí jen expanze proměnných a substituce výstupu. V apostrofech nemá speciální význam vůbec nic, dokonce ani zpětné lomítko. Napsat apostrof do literálu v apostrofech tedy nelze.

```
echo ’Napsat apostrof (’\’’) není snadné.’
```

Důležitý je rozsah platnosti proměnných. Každá proměnná, kterou ve skriptu máte, je viditelná jen a pouze pro váš skript. Nepřežije konec skriptu a nepředá se do vnořených spuštěných příkazů. To druhé lze změnit tzv. *exportem* proměnné.

Pomocí interního příkazu `export prom` zařídíte, že obsah proměnné `prom` bude dostupný i v programech, které spustíte zevnitř vašeho skriptu.

Naopak, proměnné z „vnějšku“ budou dostupné jako tzv. proměnné prostředí. V shellových skriptech se jejich obsah dostane do vnitřních proměnných, např. v C se k nim dá přistoupit pomocí funkce `getenv`.

V souvislosti s rozsahem platosti proměnných se hodí příkaz `source`, který provede vložení. Pokud nějaký skript zavoláte, dostanete z něj pouze výstup a návratovou hodnotu – není způsob, jak by mohl předat obsah proměnných. Pokud jej ale vložíte, spustí se jeho obsah ve stejném procesu. Má tedy přístup ke všem proměnným vašeho programu a obráceně.

K nahlédnutí do proměnných prostředí se dá použít příkaz `env`. Pokud chcete spatřit i neexportované proměnné, pomůže interní příkaz `set`.

Dále má shell spoustu magických proměnných:

- `$0` obsahuje název skriptu tak, jak byl volán
- `$1`...`$9` poziční argumenty
- `$#` počet argumentů
- `$*` a `$@` všechny argumenty (rozdíl níže)
- `$$` identifikátor procesu shellu
- `$IFS` znaky používané pro oddělení slov v konečné fázi expanze

`$*` a `$@` jsou obě zkratkou za vypsání všech pozičních argumentů, jen se každá chová jinak podle uvozovek:

`$*`, `$@` → `$1 $2 $3`
`"$@"` → `"$1" "$2" "$3"`
`"$*"` → `"$1c$2c$3"`, kde `c` je první znak `IFS`

Výchozí hodnota proměnné `IFS` (z angl. Internal Field Separator) je mezera, tabulátor a nový řádek. Díky tomu se chová dělení na slova před expanzí stejně jako po ní. `IFS` ale můžete změnit a tím si upravit chování některých příkazů k obrazu svému – nebo taky způsobit divné chování shellu.

Řídící struktury

Shell samozřejmě poskytuje běžné řídící struktury jako podmínky a cykly. Způsob, jakým to dělá, je ale poněkud... nezvyklý.

Začněme podmínkou. Její syntax je:

```
if cmd; then ...; else ...; fi
```

Část else je nepovinná. Důležitý poznatek – shell nemá nic jako číselnou, natožpak logickou proměnnou. Nemá tedy ani nic jako výraz v podmínce. Místo toho se větev `then` provede tehdy, když byl příkaz `cmd` úspěšný, neboli vrátil nulu. Ještě jednou pro jistotu – za `if` se píše příkaz.

Velmi důležitý je příkaz `test`, který umožňuje porovnávat svoje parametry, a to dokonce i v číselném kontextu. Užitečný je jeho manuál, určitě se do něj podívejte (`man test`). Pár používaných testů:

neprázdnost:	`test -n "$prom"`
řetězce:	`test "$USER" = "hroch"`
čísla $a = b$:	`test "$$" -eq 1`
$a \geq b$:	`test "$a" -ge "$b"`
existence souboru:	`test -f "$file"`
adresáře:	`test -d "$file"`

Příkaz `test` ale v mnoha zdrojových kódech nenajdete. Existuje na něj totiž alias, který vypadá přirozeněji – příkaz `[`. Pokud ale `test` voláte tímto jménem, musí být jeho posledním argumentem `]`.

```
if [ "$i" -ge "$j" ]
```

Z principu fungování podmínky a příkazu `test` je nutné všechny části oddělit mezerou. Nezapomeňte na uvozovky okolo proměnných – bez nich to sice zpravidla bude fungovat, ale jinak, než byste chtěli. Příkaz `test` je většinou jak samostatný program, tak interní příkaz shellu, to aby se kvůli každé podmínce nemusel spouštět další proces.

Syntaxe cyklů je v jistém smyslu podobná.

```
while cmd; do ...; done
```


Opět, cyklus se opakuje, dokud příkaz `cmd` vrací nulovou návratovou hodnotu. Místo negace existuje cyklus `until`. Pokud byste chtěli něco jako `do-while`, ten se v shellu dělá těžko.

Zajímavější je cyklus for. Neprve ukázka:

```
for i in 1 2 3 4 5; do ...; done
```

Cyklus iteruje přes seznam argumentů, které do proměnné postupně dosazuje. Může se to zdát nepraktické, ale vzpomeňte, co všechno umí shell udělat při expanzi.

Pokud vynecháte `in` a seznam slov za ním, bude for iterovat přes poziční argumenty.

Úkol 1 [2b]: Vypište názvy těch souborů v pracovním adresáři, které jsou prázdné.

Další užitečnou konstrukcí je něco, co připomíná `switch` z klasických programovacích jazyků. Protože ale shell pracuje jen s texty, umožňuje vybírat mezi variantami podle tvaru řetězce:

```
case "vstup" in
        vzor) ... ;;
        vzor1|vzor2) ... ;;
        [a-z]) ... ;;
        *) ... ;;
esac
```

Shell postupně zkouší, jestli vstup neodpovídá některému ze vzorů, a vybere první splňující větev. Vzory jsou klasické shellové wildcardy. Je možno jich uvést více, jako na druhém řádku, a vzájemně je oddělit svislou čarou.

Ve vzorech můžete použít většinu expanzí (proměnné, ‘‘, wildcardy, ...). Každou větev musíte ukončit jedním z operátorů `;;` a `;&`. První jmenovaný ukončí zpracovávání `case`, tedy již nespustí žádnou větev, kdežto po druhém bude shell pokračovat v porovnávání. Expanze vzorů probíhá až těsně před porovnáním.

Protože pravá zavírací závorka rozbíjí jednoduché zvýrazňování syntaxe, je možno v dnešních shellech před vzor napsat nepovinnou závorku do páru.

Na co nesmíme zapomenout, jsou operátory `&&` a `||`, můžeme číst jako *and* a *or*. Používají se místo oddělovačů příkazů, tedy tam, kde byste použili rouru nebo středník.

Podstatné je, že se vyhodnocují zkráceně. To znamená, máme-li seznam příkazů spojených `&&`, spouštějí se po sobě a první, který vrátí nenulu, sekvenci ukončí. Naopak, posloupnost příkazů spojených `||` ukončí první úspěšný příkaz.

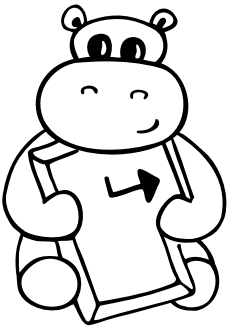
Dají se využít k příjemnému podmínění vykonání příkazů:

```
[ -f soubor ] && ...
mkdir .lock || exit
```


Zejména druhý příklad si zapamatujte, `mkdir` vrátí nulu tehdy a jen tehdy, pokud adresář neexistoval a podařilo se ho vytvořit (znovu připomínáme, že ve světě UNIXu značí nulová návratová hodnota úspěch). Dá se tedy dobře použít jako virtuální zámek.

Pokud byste potřebovali návratovou hodnotu příkazu negovat, můžete tak učinit připsáním `!` před příkaz. Vykřičník a příkaz ale musí být dvě oddělená slova, musí mezi nimi být mezera.

Úkol 2 [3b]: Napište skript, který vypíše všechny své argumenty v opačném pořadí, než v jakém je dostal.



Složený příkaz

Kdekoli, kde je možno spustit jeden příkaz, je možno místo toho použít příkaz složený. Příklad použití:

```
{
        echo "Soubor 1:"
        cat s1
} >s2
```

Zde se výstup celého složeného příkazu zapíše do souboru `s2`. Možná jste viděli i použití s kulatými závorkami místo složených. S těmi se vnitřní příkazy spustí v tzv. *subshellu.* V subshellu se spustí příkaz i pokud do něj nebo z něj vede roura, pokud je spuštěn ve zpětných apostrofech nebo pokud jej spouštíte na pozadí.

To znamená, že shell provede *fork*, a obsah závorek provede v synovském procesu. Přesměrování vstupů a výstupů tedy provádíte na nové instanci shellu. V subshellu nelze nijak ovlivnit prostředí shellu otcovského, všechny proměnné jsou lokální pro subshell. V kontextu subshellu obsahuje proměnná `$$` id procesu otcovského shellu, přestože jde o jiný proces.

Vstup

Do proměnné lze snadno vložit obsah souboru, ale jak do ní přečíst vstup? Na to je k dispozici příkaz `read`.

```
echo prvni druha | {
        read prom1 prom2
        echo "prom1: $prom1"
        echo "prom2: $prom2"
}
```

`read` přečte řádek ze standardního vstupu, rozdělí jej na slova podle `IFS`, první slovo vloží do první proměnné, druhé do druhé a tak dále.

Pokud je méně slov na vstupu než proměnných, zbylé proměnné se vyprázdní. Pokud je tomu naopak, do poslední se uloží celý zbytek řádku bez oddělovačů na konci.

Úkol 3 [1b]: Poslední příklad by bez složených závorek jen tak nefungoval. Proč je musíme použít?

Úkol 4 [3b]: Pro každého uživatele z `/etc/passwd` vyrobte v aktuálním adresáři soubor, jehož název bude odpovídat uživatelskému jménu daného uživatele. Obsahem tohoto souboru budiž uživatelův výchozí shell. O struktuře `/etc/passwd` se dočtete v manuálu, `man 5 passwd`.

Úkol 5 [3b]: Napište skript, který bude zpracovávat textový log příchodů na velké setkání KSP. Řádky obsahují akce (`prichod`/`odchod`), pohlaví (`M`/`F`) a pak jméno (jednoslovné nebo i víceslovné), oddělené mezerou. Úkolem skriptu bude načíst tento soubor a každý řádek vypsat hezky ve tvaru „Prisla Jana Novakova“, „Odesel Petr“ a podobně. Dávejte pozor na správný tvar prvního slova podle pohlaví.

Závěr

Abychom si připomněli a ujasnili, jaké věci jsme se dnes naučili a jaké můžete použít v řešení jednotlivých úkolů, zde je jejich přehled:

- Psaní a spouštění skriptů (adresář tečka)
- Vytváření a život procesů v UNIXu (*fork*, *exec*, PID)

- Proměnné (proměnné pro argumenty, `$IFS`)
- Řídící struktury (`if`, `else`, `while`, `for`, `case`)
- Příkaz `test` a `&&` s `||`
- Složené příkazy a `read`.

V dalším pokračování se můžete těšit na některé pokročilejší UNIXové příkazy, jež vám umožní třeba třídit a filtrovat velká data jen pár stisky kláves. Doufáme, že nám zachováte přízeň i nadále.

Ondra Hlavatý

KSP

seriál

27-3-7 UNIXové déjà vu — 15 bodů

Dnešní díl seriálu ve vás možná vyvolá pocit, že jste ho už někdy četli, ale ne tak docela. Vrátíme se totiž k mnohému z toho, co už umíte, a pronikneme ještě o kousek hlouběji. Připravte se na vydatnou porci povídání o nápovědě, o souborovém systému, uživatelích a právech, o řídicích strukturách a funkcích v shellu a o formátovaném výstupu.

Z předchozích dílů seriálu máte k dispozici shell, nejspíš Bash, umíte se v něm pohybovat po souborovém systému a zvládáte psát jednoduché skripty pro manipulaci s obsahem textových souborů. Také když zapomenete přepínače konkrétního příkazu, umíte si je v manuálových stránkách najít.

Umíte si ale pomoct, když zapomenete, jak se nějaký příkaz jmenuje?

Nápověda

Je nemožné si pamatovat všechny vlastnosti každého nainstalovaného programu, natož stíhat sledovat změny. Nápověda, manuál nebo dokumentace jsou základními prameny informací pro uživatele libovolného software a UNIX v tomto ohledu není výjimkou. Sebelepší informace je ale k ničemu, když ji neumíte najít.

Příkaz `man` už znáte. Napadlo vás podívat se na `man man`?

Zjistíte tam, že k hledání řetězců v popisech příkazů slouží přepínač `-k`. Pokročilejší možnosti nabízí utilita `apropos`.

Také narazíte na přepínač `-s`, jehož hodnotou je sekce manuálu a někdy jde dokonce přepínač vynechat a psát jen sekci (`man 1 man`). Počkat, co jsou sekce? Jsou očíslované, každá stránka je v nějaké zařazena a obvykle ji má uvedenou v závorce za svým jménem (např. `cat(1)`, `shells(5)` nebo `standards(7)`). Konkrétní význam a číslování se liší, POSIXový standard (který si zmíníme dále) o nich nemluví vůbec. Pro představu se podívejme na Debian Linux:

1 Spustitelné programy nebo příkazy shellu
2 Systémová volání (funkce poskytované jádrem)
3 Knihovní volání (funkce v programových knihovnách)
4 Speciální soubory (obvykle nalézané v `/dev`)

5 Souborové formáty a konvence (např. `/etc/passwd`)
6 Hry
7 Směs (včetně balíků maker a konvencí)
8 Příkazy administrace systému (obvykle jen pro roota)
9 Funkce jádra

V různých sekcích můžete najít stejnojmenné stránky. Například `passwd(1)` je utilita pro změnu hesla a `passwd(5)`[19] dokumentuje databázi uživatelů, tedy soubor `/etc/passwd`. Musíte buď sekci znát, nebo použít přepínač `-a` a postupně si prohlédnout všechny.

Když už konkrétní stránku máte, pořád není vyhráno. Může být dlouhá a nepřehledná. Pak vám pomůže váš stránkovač. Příkaz `man` by mohl vysypat horu textu přímo do terminálu se škodolibým úsměvem a slovy „poraďte si“, jako příjemnější se ale ukázalo, když pustí `less` nebo starší a standardní `more` a manuál vám ukáže v něm. V prvním dílu jsme vám prozradili, že se oba zavírají klávesou `q` (quit), přidáváme `h` (help) pro nápovědu, `/` (lomítko) pro vyhledávání (potvrdíte enterem) a `n` (next) pro vyhledání dalšího výskytu.

Aby to nebylo příliš jednoduché, k manuálu existuje alternativa: infostránky. Některé jsou mnohem obsáhlejší než odpovídající manuálová stránka a jsou členěné, nevypadají jako jeden dlouhý dokument. Jejich prohlížeč se jmenuje `info`, nápovědu v něm získáte napsáním otazníku, zbytek už zjistíte sami.

Bash k nápovědě přistupuje po svém. Na `man bash` najdete i popis jeho vestavěných příkazů, jako je `cd` nebo `pwd`, kdo by ovšem chtěl hledat jehlu v kupce sena? Vysvobodí vás jeho příkaz `help`. Mrkněte na `help help`, je vcelku intuitivní.

Pokročilejší z vás by mohlo zajímat, které utility a jejich přepínače mají být dostupné na všech UNIXech, ať už je to Gnu/Linux, Solaris, OS X nebo nějaká odnož BSD. Taková znalost slouží k psaní přenositelných skriptů, tedy skriptů, které budou fungovat i na jiném systému, než na kterém jste je napsali. Vaši zvědavost ukojí norma POSIX, které se certifikované UNIXy držet musejí a ty ostatní aspoň plus minus chtějí. Kdykoliv se budeme odvolávat na normu nebo POSIX, myslíme POSIX 2013.[20] Na jeho stránce je vpravo dole odkaz ke stažení té kupky HTML stránek v jednom archivu, z neoficiálních zdrojů je možné sehnat POSIX i v podobě manuálových stránek. V Debianu je takovým zdrojem balík `manpages-posix` v repozitáři non-free.

[19] sekce 5 na Linuxu, jinde nejspíš jiná

[20] `http://pubs.opengroup.org/onlinepubs/9699919799/`

Souborový systém

První díl seriálu se vás snažil nezahltit a nerozptylovat, o souborovém systému řekl jen to nejnutnější, minule jste nakoukli do práv souborů, když jste vytvářeli spustitelný skript. Je načase povědět o souborovém systému víc.

Logicky je souborový systém jediný, s kořenem `/` („root“), fyzicky jich ale bývá víc, z nichž některé mohou sídlit třeba jen v operační paměti nebo dokonce na úplně jiném stroji. Všechny dostupné na aktuálním stroji si můžete prohlédnout příkazem `df`. Vypíše pro každý souborový systém do tabulky název, velikost, využití a kam v logickém souborovém systému je připojený. Často je k dispozici přepínač `-T`, se kterým `df` ukáže i typ souborového systému, a přepínač `-h`, se kterým vypíše obsazené a volné místo v lidsky čitelných jednotkách.

Prostor zabraný konkrétním souborem umí spočítat `du`. Pokud dostane adresář, rozpitvá statistiku na jeho položky, podobně jako je tomu u `ls`. Příkazu `ls` to můžete zakázat přepínačem `-d`, obdobně u `du -s`.

Nenechte se zmást tím, že `df` i `du` přemýšlejí v blocích. Je to dáno běžnou strukturou disků, soubor zabírající blok jen z části nemůže jeho zbytek přenechat jinému souboru, přebytečné místo zůstane nevyužité. Velikost bloku se obvykle liší mezi normou, utilitami a diskem, buďte tedy obezřetní a uvědomujte si, jaká velikost se u vás kde používá.

Příkaz `ls` s přepínačem `-l` zobrazuje velikost souboru v bajtech a na zabrané bloky se nijak neohlíží. Počet zabraných bloků nechá zobrazit přepínač `-s`. Celkovou velikost zabraných bloků v lidsky čitelných jednotkách ukazuje `du -h`.

Úkol 1 [1b]: Zjistěte, jak velké bloky používá váš disk a vaše utilita `du`, která by podle normy měla používat 512B bloky. Svá zjištění doložte použitými příkazy, jejich výstupy a popisem své úvahy.

Konkrétní použitý souborový systém s sebou nese svá omezení. Na Windows se kdysi používal formát FAT, později NTFS, v Linuxu jsou doma ext2 až ext4, v BSD a Solarisu ufs. Lišit se mohou maximální délkou jména souboru, maximální velikostí souboru, maximální využitelnou velikostí disku, povolenými znaky v názvech souborů, (ne)podporou ukládání různých metadat, ...

Norma vyžaduje, aby souborový systém rozlišoval malá a velká písmena a názvy souborů neobsahovaly lomítko (oddělovač komponent cesty) a NUL (`\0` v jazyce C, bajt s hodnotou 0). Jazyk C vznikl pod UNIXem a UNIX do něj byl po čase přepsán, jsou spolu dodnes hodně prolnuté. Když zakážeme znak NUL, máme zaručeno, že je možné název souboru považovat za řetězec jazyka C a používat na něm řetězcové funkce, např. `strlen()`.

Výše uvedená omezení jsou dnes běžně opravdu jediná vynucená, všechno ostatní funguje.[21] Čímž neříkáme, že celý zbytek Unicode v názvech souborů najdete, nebo dokonce že můžete obskurními znaky soubory beztrestně pojmenovávat.

Díky absenci NUL v názvu souboru máme jisté, že když za sebe naskládáme jména souborů oddělená znakem NUL, budeme je umět opět jednoznačně rozdělit. Toho využívají některé běžné utility, bohužel pomocí nestandardních přepínačů. Tuto jistotu nám norma nedává, pokud použijeme jako obvykle znak LF (`\n` v C, „konec řádku“).

Proto se z bílých znaků používá jen mezera, LF v názvu naštěstí není běžné, tedy můžeme být v klidu. Kdo takovou zvyklost poruší, následky nechť si nese sám. Zkuste si nějaký soubor s LF v názvu vyrobit a pohrát si s ním!

Soubory s diakritikou, interpunkcí a mezerami v názvech se opravdu dají potkat, takže by s nimi vaše skripty měly umět pracovat. Pomohou vám k tomu znalosti z prvního dílu: escapování, uvozovkování a ve vzácném případě minusu na začátku názvu souboru parametr `--`.

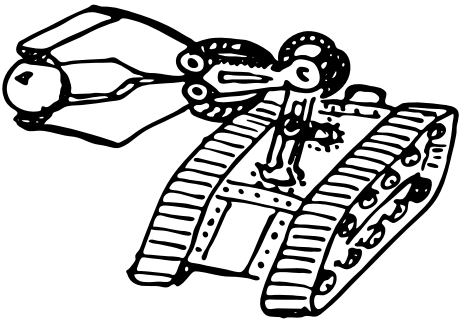
Soubory systému a programů obvykle dodržují ještě mnohem striktnější omezení, než je to popsané výš. Volí jména souborů, která

1. obsahují jen písmena velké a malé anglické abecedy, číslice, podtržítko, tečku a minus (`[A-Za-z_.-]`), a navíc
2. nezačínají minusem (aby se nepletla s přepínači).

[21] Nepočítáme-li obskurní starožitný FAT, přežívající na některých flashdiscích.

Soubory s tečkou na začátku jsou skryté před wildcardy a `ls`. Druhé běžné použití tečky je oddělení přípony od zbytku jména souboru, jinak se tímto znakem šetří.

Shell se hodí na jednoduché skripty spořící čas, ne na psaní neprůstřelných programů (to v něm ani dobře nejde). Pokud v něm budete pracovat víc, dojdete ke kompromisu mezi omezováním se a nutností uvozovkovat při běžné práci moc často. Vynecháte nejspíš běžné oddělovače (LF, mezeru, tabulátor, dvojtečku, středník a čárku), speciální znaky shellu (`$‘"’#!?*[]{}();|\<>&`), a dáte si pozor na minus na začátku názvu. Možná si navíc budete šetřit čas při psaní vynecháním diakritiky a velkých písmen, ačkoliv to zas tolik nepomůže. Důležité je hlavně trefit začátek slova a zbytek už zařídí doplňování tabulátorem.



Části cesty, přípony

Přípony. Ve Windows se podle nich točí svět, binárku bez přípony `.exe` spustíte těžko. UNIX se s nimi vypořádal jinak – přípony považuje za informaci pro uživatele, sám se řídí prvními několika bajty souboru, kde obvykle je „magic number“. Podle něj umí formát určit třeba i utilita `file`:

```
hroch@ksp:~$ file /etc/passwd
/etc/passwd: UTF-8 Unicode text
hroch@ksp:~$ file /usr/bin/vimtutor
/usr/bin/vimtutor: POSIX shell script text exe-
cutable
```

U binárních programů toho `file` umí zjistit hodně. Kdybychom ho neměli, museli bychom binárku prohlížet nějak ručně a třeba si všimnout toho, že na začátku jsou znaky DEL, E, L a F, přičemž ELF je jméno formátu spustitelných souborů pro UNIX.

```
hroch@ksp:~$ od -c -Ax -tx1 -N10 /bin/sh
000000 177   E   L   F 002 001 001  \0  \0  \0
        7f  45  4c  46  02  01  01  00  00  00
00000a
```

Zkuste si sami pomocí `od` nebo rozšířeného, ale nestandardního `hd` prohlédnout nějaký obrázek PNG, dokument PDF, ... Pokud chcete být drsní, vynechte u `od` přepínač `-c` a ve vedlejším terminálu si otevřete `man ascii`. ;-)

Příponu tedy běžně není potřeba od zbytku jména souboru oddělovat, obzvlášť u textových a spustitelných souborů často ani žádná přípona použita není. Zato bychom někde ve skriptu mohli chtít získat zvlášť jméno souboru a zbytek cesty. Poslouží nám příkazy `basename` a `dirname`:

```
hroch@ksp:~$ dirname /usr/bin/less
/usr/bin
hroch@ksp:~$ basename /usr/bin/less
less
```

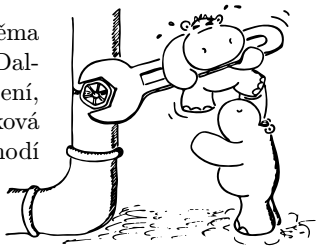

Jejich opakovaným použitím můžete rozebrat cestu na jednotlivé komponenty.

Typy souborů

Když už jsme nakousli soubory v UNIXu, podívejme se na ně blíž. UNIXová filosofie se totiž drží zásady, že skoro všechno je soubor. Běžné textové soubory nebo soubory s binárními daty (fotky, videa, ...) nás asi nepřekvapí. Ale UNIX jako soubory reprezentuje i takové věci jako vstup z klávesnice (systém odtud čte po znacích) nebo výstup do zvukové karty. Podstatné je, jak se který soubor chová při používání.

Soubor je na disku typicky reprezentován jedním *inodem*, každý z nich má v rámci souborového systému svoje unikátní číslo. Uvnitř mezi dalšími metadaty systém ukládá informace o právech a vlastnících souboru, jeho typ a velikost, počítadlo odkazů (viz dále) a hlavně odkazy na jednotlivé datové bloky se samotným obsahem.

Je důležité, že jméno souboru si nepamatuje sám soubor, ale pamatuje si ho nadřazený adresář (což je jen speciální typ souboru). Inode reprezentující adresář obsahuje ve své datové části jména a příslušná čísla inodů pro všechny v něm obsažené soubory.



V předchozích dílech jsme se věnovali jen dvěma typům souborů: běžným souborům a adresářům. Dalšími jsou již zmiňovaná vstupní a výstupní zařízení, která sídlí hlavně v adresáři `/dev` a dělí se na bloková (disk) a znaková (terminál). V neposlední řadě se hodí vědět o rourách.

Zatím jsme potkali jen roury anonymní, které shell natahuje mezi dvěma příbuznými (společně spouštěnými) procesy: `ls -l /bin | head`. Mezi nepříbuznými procesy (spouštěnými třeba i dvěma různými uživateli) anonymní rouru natáhnout nejde, ale oba mohou znát cestu k pojmenované rouře. Jeden z ní čte, druhý do ní zapisu-

je a jméno potřebují jenom k tomu, aby ji mohli otevřít. A kde pojmenovanou rouru sebereme? Vytvoří ji příkaz `mkfifo`.

V shellu je bezesporu nejpoužívanějším speciálním souborem `/dev/null` neboli „černá díra“. Je to ideální místo, kam zahazovat věci, které na nic nepotřebujeme. Můžeme ho využít, pokud nás nezajímá standardní výstup příkazu, a jde nám jenom o jím vyrobený soubor nebo jeho návratovou hodnotu. Pokud bude příkaz chtít vstup a my mu budeme chtít dát prázdný soubor, `/dev/null` je také vhodné použít.


```
echo Windows > /dev/null
cat /dev/null
```


Podobně se chová `/dev/zero`, až na to, že při čtení dodává nekonečně dlouhou posloupnost znaků NUL. Soubor délky 1024 bajtů lze vytvořit příkazem `head -c 1024 /dev/zero > soubor`. Pěkné hraní je i se soubory `/dev/random` a `/dev/urandom`.[22]

Někdy přesměrujeme našemu skriptu výstup do souboru, a přesto bychom chtěli vybrané informace o jeho provádění vidět na terminálu. Může nám je ukazovat tak, že je bude zapisovat do `/dev/tty`, který reprezentuje aktuální terminál. Podobně když máme přesměrovaný vstup a chceme z terminálu (z klávesnice) číst.

```
(
        echo 'Potvrdte "ano":' > /dev/tty
        read odpoved < /dev/tty
        [ "$odpoved" = ano ] || exit 1
        cat
) < soubor1 > soubor2
```

Málem bychom zapomněli... Ve výstupu `ls -l` poznáte jednotlivé typy souborů podle prvního znaku na řádku, ještě před právy. Kdyby nějaké písmeno nebylo jasné, v manuálu `ls` jsou zkratky vysvětleny. Minus je běžný soubor.

Odkazy v souborovém systému

Občas se nám hodí pořídit si zkratku, rychlý odkaz na nějaký soubor či adresář. Na takové věci se v UNIXu využívají *hardlinky* (linky) a *symlinky*.

Hardlink je odkaz, který „natvrdo“ ukazuje na stejný inode jako odkazovaný soubor. V adresáři je pod jménem linku uloženo přímo číslo inode, na který odkazuje. Při vytvoření linku se v daném inode zvedne počítadlo odkazů, při jeho smazání se zase sníží, a když dojde na nulu, odstraní se i samotná data. Výjimkou jsou otevřené soubory – dokud nějaký proces má soubor otevřený, souborový systém přepsání jeho dat neumožní. Tak může mít proces bezpečně otevřený i soubor, který už nemá žádné jméno.

[22] `http://cs.wikipedia.org/wiki//dev/random`

Po vytvoření hardlinku jsou původní a nový soubor od sebe nerozeznatelné, ale nese to s sebou několik omezení. Předně musí všechny linky sídlit na stejném diskovém oddílu, kde jsou uložena samotná data, a také nejde vytvořit hardlink na adresář, jen na soubor. (Kdo by se vyznal v souborovém systému, kde může být adresář umístěný sám v sobě?) Při přesunutí (přejmenování) nebo vymazání původního souboru bude hardlink ukazovat stále na původní verzi. Protože mohou být hardlinky zrádné, doporučujeme pro běžnou práci používat spíše symlinky.

Symlink, neboli *symbolic link*, je jen jednoduchý zástupce, který ukazuje na nějakou cestu v souborovém systému (na libovolném připojeném oddílu a na libovolný soubor či adresář). Ve skutečnosti vypadá skoro jako malý textový soubor, který má v sobě zapsanou absolutní (začínající lomítkem) či relativní cestu ke svému cíli.

Při vymazání nebo přesunutí původního souboru přestane symlink fungovat, pokud neobsahuje relativní cestu a není přesunut spolu s cílem. Ukazuje totiž na cestu, ne na konkrétní soubor. Pokud cílový soubor smažeme a nahradíme novým, bude symlink ukazovat na tento nový soubor.

K symlinkům je třeba dodat dvě varování, abyste se nedivili a nedomýšleli si něco, co není pravda.

1. Předně, symlink a zástupce z Windows se chovají zásadně jinak. Zástupci jsou běžné soubory, podobné `.desktop` souborům na Linuxu, jen jsou binární.
2. Druhé varování se týká výstupu `ls -s`. Na mnoha systémech bude u symlinků ukazovat nulový počet zabraných bloků. Pokud je totiž symlink dostatečně malý, není problém ho celý uložit přímo uvnitř jeho inode. Jak úsporné! :-) Říká se tomu inlining a nové souborové systémy (třeba ext4) umí toto chování zapnout i u jiných typů souborů.

Hardlinky a symlinky se vytvářejí příkazem `ln`. Bez přepínače vyrobí hardlink, s přepínačem `-s` symlink:

```
ln cesta/k/souboru novy_hardlink
ln -s cesta/k/souboru relativni_symlink
ln -s /cesta/k/souboru absolutni_symlink
```

Cíl symlinku běžně vyčtete z `ls -l`:

```
lrwxrwxrwx 1 hroch users 3 8. pro 08:00 zdroj -> cil
```

Ve skriptech se nám bude hodit spíš příkaz `readlink`, který vypíše jenom cíl odkazu. Také má užitečný přepínač `-f`, se kterým vypisuje absolutní cestu získanou z argumentu nahrazováním všech symlinků na cestě skutečnými cestami, kam ukazují. Příkazu `readlink -f` tedy má smysl dávat nejenom symlinky, ale i běžné soubory.

Pokud budete uvnitř adresáře, do kterého jste se dostali přes symlink, můžete si plnou cestu nechat vypsat pomocí `pwd -P`, přepínač `-P` zajistí rozepsání všech symlinků v cestě. Chová se stejně jako `readlink -f .` (všimněte si tečky).

Úkol 2 [2b]: Pusťte si `ls -ld` na nějaký adresář a všimněte si, kolik má hardlinků. Odkud vedou?

Úkol 3 [2b]: Vysvětlete, jak se v souvislosti s hardlinky liší

```
cat </dev/null >soubor
```

a

```
rm soubor
cat </dev/null >soubor
```

KSP

seriál

Uživatelé, skupiny, vlastníci a práva

Teď už máme představu o tom, co všechno v souborovém systému můžeme najít. Zatím jsme v něm ale beznadějně sami. Sotva přijdou další uživatelé, potřebujeme před nimi občas něco schovat, nenechat je měnit naše soubory, ... Tedy budeme potřebovat systém oprávnění, který jsme v minulém dílu načali právem ke spuštění. Vůbec jsme ale nemluvili o tom, komu `chmod +x` právo ke spuštění přidělí.

UNIX je odpradávna víceuživatelský systém. Oddělené účty uživatelů a dodržování principu minimálních oprávnění mu zajišťují slušnou míru bezpečnosti. Soubor `/etc/passwd` obsahující databázi uživatelů už jste potkali, jeho dokumentaci najdete na Linuxu v `man 5 passwd`, jinde musíte hledat v jiné sekci. Možná vás zajímalo, kde se ukládají hesla – pak vězte, že v dřevních dobách to bylo opravdu tam, ale moderní UNIXy mají na citlivé údaje oddělenou databázi jinde v adresáři `/etc`. Hesla se navíc ukládají zakódovaná. Na Linuxu vám víc prozradí `man shadow`.

Vžijte se na chvíli do role administrátora školního UNIXového serveru. Máte na něm účty desítek, možná i stovek uživatelů a chcete jim přidělit nějaká oprávnění (jaká, k tomu se dostaneme později). Chtělo by se vám u každého zvlášť přemýšlet, co smí a nesmí? Asi byste si všimli, že studentské účty mají mít obecně jiná oprávnění než účty učitelů, že dokumenty k maturitnímu plesu jedné třídy nemá co upravovat nikdo z jiných tříd, tedy pokud nemají dvě třídy ples společný, atd. Přáli byste si, abyste mohli systému o třídách a učitelích říct a on vám dovolil oprávnění přidělovat hromadně.

Přání se vám splnilo, řešení má jméno *skupiny*. Jsou to pojmenované množiny uživatelů, je jim možné nastavovat společná práva, jejich databáze sídlí v `/etc/group` a více si o ní můžete přečíst v `man group`. V `/etc/passwd` má každý uživatel navíc skupinu, pod kterou se přihlašuje, neboli login group. Do ní automaticky patří.

Co znamená, že se uživatel pod nějakou skupinou přihlašuje? Inu, tato skupina je skupinovým vlastníkem jím vytvářených souborů. Co to pro ně znamená?

Když se podíváte na výstup `ls -l`, uvidíte řádky jako:

```
-rwxr-x--x 1 hroch users 42 8. pro 08:00 soubor
```


Uživatelským vlastníkem souboru `soubor` je `hroch`, skupinovým je `users`. Pokud se řekne jenom vlastník, myslí se uživatelský vlastník. Nenechte se zmást, až někde uvidíte `hroch` v obou sloupcích – skupiny a uživatelé se mohou jmenovat stejně a na některých systémech se takto pro každého uživatele zakládá jeho vlastní login group.

Na uživatelského vlastníka (user, owner) se vztahuje jiné nastavení oprávnění než na toho skupinového (group, group owner), a na toho zase úplně jiné než na všechny zbývající uživatele (others, world) kromě uživatele `root`.

Root, nebo také superuživatel, je takovým místním bohem. Smí všechno. Jeho oprávnění je potřeba např. k zakládání nových účtů nebo ke změně vlastníka souboru. Změnit skupinu souboru na některou ze skupin, ve které je členem, může ale i vlastník souboru.

Tolik pravomocí s sebou nese i hodně zodpovědnosti a moc velký průšvih, když se k účtu roota dostane někdo nepovolaný. Přestože se hledají alternativní cesty, jak si poradit bez superuživatele, root s námi ještě nějaký čas bude.

Pomocí příkazu `su` můžeme pouštět příkazy rootovým jménem, nebo i jménem jiného uživatele, pokud známe jeho heslo. Pokud si pustíme rootovský shell, bude mít v promptu místo dolaru mřížku (`#`). Alternativou `su` je příkaz `sudo`, který nám dovoluje pouštět příkazy pod jiným vlastníkem po zadání k tomuto účelu speciálně nastaveného hesla.

Pokud root zrovna nejste a snažíte se přistupovat k souboru, jaká pro vás platí oprávnění? Ve výše uvedeném příkladu

- `rwx` pokud jste `hroch`, jinak
- `r-x` pokud jste ve skupině `users`,
- `--x` ve všech ostatních případech.

Tento *symbolický zápis práv* není až tak spletitý, jak na první pohled vypadá. První pozice v každé trojici je Read (čtení), druhá Write (zápis), třetí eXecute (spuštění). Minus znamená, že právo není přiděleno. Při čtení práv ve výstupu `ls -l` nezapomeňte, že těsně před nimi je typ souboru. Připomínáme, že minus znamená běžný soubor.

Často se používá ještě druhý způsob zápisu práv, *numerický*, nebo také *oktalový*, tedy pomocí osmičkové číselné soustavy. Práva vlastníka, skupinového vlastníka a ostatních v něm jsou reprezentovaná třemi osmičkovými ciframi. Jedna osmičková cifra má tři bity. 4 = `r`, 2 = `w`, 1 = `x`. Skládání se dělá bitovým součtem (or). 000 jsou tedy doslova nulová práva, 777 všechno všem, 700 všechno vlastníkovi, 755 všechno vlastníkovi a ostatním jen čtení a spouštění, 640 čtení a psaní vlastníkovi, čtení skupině a nic ostatním.

U běžných souborů jsou práva téměř intuitivní. Drobné zádrhele mohou být, že interpretované programy potřebují kromě hashbangu (např. `#!/bin/bash`, viz te předchozí díl seriálu) a práva ke spuštění i právo ke čtení. Interpret se k textu programu holt nějak musí dostat. U binárek problém není. Naopak binárky nespustíte bez práva ke spuštění, kdežto u skriptu v Bashi si snadno poradíte zavoláním `bash skript.sh`.

Větší potíže bývají s významem práva k zápisu. My jsme si ale už vysvětlili, jak fungují linky na soubor a že data souborů a záznamy v adresářích jsou na sobě do značné míry nezávislé, takže to máme snazší. Právo k zápisu mluví u souboru o zápisu do jeho dat. K přejmenování nebo smazání se vztahuje právo zápisu do adresáře. Přesto utilita `rm` váhá, když má mazat soubor, který nemá právo k zápisu. Je potřeba smazání ručně potvrdit nebo předem přidat přepínač `-f` (force – „Na nic se neptej a maž!“).

Práva souboru se ukládají s jeho inode – pokud je změníme přes jeden link, projeví se změna i ve všech ostatních. Zde se hodí třetí varování k symlinkům: práva u nich nemají smysl. Symlink se často chová transparentně a rozhodující jsou práva cílového souboru. Nenechte se zmást, `ls` mu přisuzuje práva 777.

U adresářů jsme už právo k zápisu vyřídili o dva odstavce výš. Právo ke čtení adresáře potřebují ke své správné funkci wildcardy, `ls`, doplňování tabulátorem a vůbec cokoliv, co potřebuje vidět obsah adresáře. Právu `x` se u adresářů říká search (prohledávání). Dovoluje nám se znalostí jména souboru přistoupit k jeho obsahu, tedy při hodně nízkoúrovňovém pohledu vlastně přečíst z adresáře číslo inode souboru, pokud dodáme jeho jméno.

Bez práva ke čtení, ale s právem k prohledávání můžeme jaksi „poslepu“ pracovat s obsaženými soubory známých jmen, a v závislosti na právu k zápisu je můžeme i vytvářet a mazat. S právem ke čtení, ale bez práva k prohledávání můžeme obsah adresáře jenom vypsat, a to ještě mizerně. U každé položky uvidíme v `ls -l` jen jméno (a na některých souborových systémech i typ), ostatní metadata jsou uvnitř inode a místo nich se zobrazí jen otazníky. Ani nám pak nebude vadit, že bez práva k prohledávání nemůžeme nastavit adresář jako svůj pracovní (`cd adresar`).

Když už víme, k čemu práva jsou, jak je změnit? Příkaz `chmod` už jsme párkrát zmínili. Teď už navíc budete rozumět jeho manuálu, kde se můžete dočíst pikantní podrobnosti o právech včetně těch, které se sem nevešly.

Příkaz `chmod` může pracovat s právy zadanými oktalově i symbolicky. Se symbolickými právy umí nastavovat i jednotlivá práva při zachování ostatních. Kombinací je docela hodně, uveďme tedy jen příklad: `chmod u+x,go-rw soubor` přidá uživateli právo ke spuštění a skupině i ostatním sebere práva ke čtení a k zápisu, ostatní práva nechá netknutá.

Vlastníka umí měnit příkaz `chown`, skupinového `chgrp`. Příkaz `chown` navíc umí měnit oba najednou, stačí je zadat oddělené dvojtečkou:

```
ksp:~# ls -l s; chown palec:ksp s; ls -l s
-rw-r--r-- 1 hroch users 1 8. pro 08:00 s
-rw-r--r-- 1 palec ksp   1 8. pro 08:00 s
```

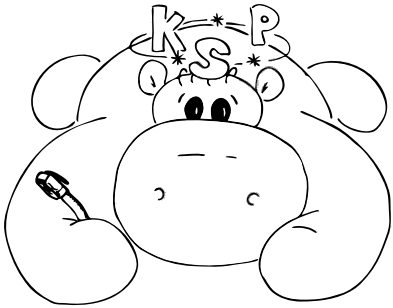

Pokud u `chown` vynecháme uživatele před dvojtečkou, změní jen skupinového vlastníka. Příkazy `chmod`, `chown` i `chgrp` mají přepínač `-R`, který je nechá změnu aplikovat na daný adresář a rekurzivně na všechen jeho obsah.

Když už soubory existují, poradit si s nimi umíme. Teď si povíme, s jakými právy a vlastníky se zakládají.

Stejně jako soubor, i každý proces má vlastníka. Vlastník procesu se použije jako vlastník souborů tímto procesem vytvářených. Se skupinovým vlastníkem je to složitější, ten se občas může dědit od adresáře. Přesný algoritmus výběru závisí na systému a jeho nastavení. Náš shell má jako vlastníka nás a jako skupinového vlastníka naši login group. Procesy, které spouští, tyto vlastníky zdědí. Všechny „běží pod námi“.

Výše zmíněné příkazy `su` a `sudo` umožňují ovlivňovat vlastníka, o skupinového vlastníka se postará `newgrp`. Informace o aktuálním uživateli, skupině a ostatních skupinách, ve kterých uživatel je, poskytuje příkaz `id`.

Výchozí práva běžných souborů jsou 666, výchozí práva adresářů 777. Při vytváření se ale aplikuje maska, která zruší v přidělovaných právech ty bity, které jsou v ní nastavené na jedničku. Maska je stejně jako vlastník a skupina vlastností procesu a stejně jako oni se dědí. U shellu ji můžeme zjistit příkazem `umask` bez parametrů a nastavit stejným příkazem, kterému předáme novou masku jako argument. Typické nastavení je `umask 002` („nedávej `w` celému světu“) nebo `umask 022` („`w` nech jen vlastníkovi“).



Úkol 4 [3b]: Jak root zařídí, aby domácí adresář uživatele hroch nebyl přístupný ostatním a aby na tom hroch nemohl nic změnit? Dodejme, že je dobrým zvykem, aby domácí adresář patřil příslušnému uživateli.

Řídicí struktury a proměnné – podruhé na návštěvě

Ve druhém dílu seriálu jsme se potkali se základními řídicími strukturami. Pokud si nepamatujete použití podmínky `if` nebo dvou základních cyklů níže, připomeňte si je.

```
if cmd; then ...; else ...; fi
while cmd; do ...; done
for i in 1 2 3 4 5; do ...; done
```


Dnes k těmto základům přidáme mocnější zbraně. Nejdříve si pořídíme na hraní nějký nekonečný `while` cyklus, tedy cyklus, jehož podmínka bude vždy splněná. Abychom nemuseli psát podmínku stylu „nula je menší než jedna“, nabízí nám shell příkazy `true` a `false`.

Ano, nepřepsali jsme se, nejsou to konstanty nabývající hodnoty pravda a nepravda jako ve většině programovacích jazyků. V shellu se jedná o samostatné příkazy, které doslova dělají nic, a to buď úspěšně, nebo neúspěšně. Podívejte se na jejich manuálové stránky.

Náš nekonečný cyklus, ve kterém si třeba budeme k proměnné `X` na konec připisovat `D` a to celé vypisovat, může vypadat takto:

```
X=
while true; do
        X=${X}D
        echo $X
done
```

Všimněte si složených závorek. Dovolíme si zde malé odbočení k proměnným. Konstrukce `${X}` funguje stejně jako `$X`, jen ji shell i v tomto případě správně pozná. Jméno proměnné smí obsahovat písmena anglické abecedy, číslice (kromě prvního znaku) a podtržítko. Shell čte tyto znaky za dolarem, dokud se nezasekne o něco jiného, a teprve pak hledá, jestli proměnnou zná. Kdybychom jako přiřazení použili `X=$XD`, shell by neúspěšně hledal proměnnou `XD`, tedy by se do `X` přiřazoval prázdný řetězec.

Mohli byste namítat, že stejně dobře a elegantněji bychom mohli přidávat místo na konec na začátek pomocí `X=D$X` a složeným závorkám se vyhnout. Máte pravdu, tady to jde. Jenže ne vždy je možné se z průšvihu vylhat. Ještě můžeme psát `X=$X""D`, což je opravdu divné, a kdybychom už uvozovky používali, musíme víc přemýšlet. Za chvíli bychom ještě byli rádi za složené závorky.

Cyklus, který jsme vyrobili, je nám zatím docela nanic, protože běží donekonečna. Hodilo by se nám umět ho ve vhodnou chvíli zastavit, k tomu slouží příkaz `break`. Chová se podobně jako v jiných programovacích jazycích, tedy ukončí provádění celého cyklu. Pokud ho spojíme se šikovnou podmínkou, můžeme cyklus zastavit ve chvíli, kdy délka generovaného řetězce písmen `D` překročí deset.

```
X=
while true; do
        X=D$X
        delka=‘echo -n $X | wc -c‘
        if [ $delka -gt 10 ]; then break; fi
        echo $X
done
```

Všimli jste si, jak je zjišťování délky proměnné neohrabané? Ještě si musí člověk dávat pozor, jestli náhodou nepočítá i znak konce řádku... Taková běžná operace, to přeci musí jít lépe! A taky že jo: podobně jako je `${X}` expandováno na obsah proměnné `X`, tak je `${#X}` expandováno na její délku.

Společně s příkazem `break` jde ruku v ruce příkaz `continue`, který přeskočí všechny za ním následující příkazy, až před test podmínky další iterace cyklu. Vynechání řetězců kratších pěti znaků se sice dá udělat i lépe, přesto použijme příkaz `continue`:

```
X=
while true; do
        X=D$X
        delka=${#X}
        if [ $delka -lt 5 ]; then continue; fi
        if [ $delka -gt 10 ]; then break; fi
        echo $X
done
```

Až budete pracovat se zanořenými cykly a budete chtít příkazem `break` nebo `continue` ovlivnit jiný než nejvnitřnější z nich, vzpomeňte si, že oba příkazy mají nepovinný parametr. Zbytek už najdete. Pokud jste zvyklí na C, příkaz goto byste hledali marně.

Jako třetí mezi podobnými přichází na řadu příkaz `exit`, který najde použití hlavně ve skriptech. Jeho zavolání ukončí shell (je tedy silnější než `break`), a když se mu předá číslo (třeba `exit 42`), nastaví ho jako *návratovou hodnotu*. S ní jsme se potkali ve druhém dílu. Ve zkratce nula znamená úspěšné ukončení, cokoliv nenulového neúspěšné.

Návratová hodnota je na Linuxu 8-bitová (tedy může nabývat hodnoty 0–255), na většině jiných UNIXových systémů je však jen 7-bitová (0–127). Bash sám o sobě využívá hodnoty 126 a 127 pro své vlastní potřeby, tedy pokud chcete psát univerzální programy, omezte se jen na hodnoty 0–125.

Pokud chcete zjistit, s jakou návratovou hodnotou skončil poslední provedený příkaz, můžete k tomu využít speciální proměnnou `$?`. Zkuste si třeba zavolat příkazy `true`, `false` nebo `(exit 42)` a hned po nich `echo $?`. Pokud `exit` žádnou návratovou hodnotu nedostane jako argument, použije právě hodnotu této proměnné.

Nacyklili jsme se dost, podívejme se ještě na složitější podmínky. Poprvé potkáváme `elif`. Funguje analogicky ke stejnojmenné konstrukci v Pythonu, `elsif` v Perlu, `elseif` v PHP a prosté kombinaci `else if` v C, Javě a Pascalu. Pokud na ni při vyhodnocování dojde řada, spustí podmínku[23] a kontroluje, jestli uspěla. Pokud ano, nechá spustit kód za svým `then`, jinak pokračuje další větví podmínky (další `elif` a jako poslední `else`).

V košatějších podmínkách se může hodit nějakou větev mít, ale nic v ní nedělat. Když hned za `then` napíšete středník, shell si postěžuje na syntaktickou chybu. Mohli byste použít příkaz, který nic nedělá, třeba `true`, ale víc se hodí funkčně shodná dvojtečka (`:`). Její primární účel je naznačení, že „tady nic není“, pro svou krátkost se ovšem zneužívá i na místech, kam by se víc hodilo `true`.


```
X=$(($RANDOM % 100))
if [ "$X" -lt 10 ]; then
        echo malé
elif [ "$X" -le 42 ]; then
        : nevím, něco mezi, mlčím
else
        echo velké
fi
```

Zde je `$RANDOM` speciální proměnná Bashe, která obsahuje při každé expanzi nové náhodné číslo mezi 0 a 32767. Tento úryvek kódu tedy vygeneruje náhodné číslo mezi 0 a 99 a rozhodne, jestli je malé, nebo velké. Pokud je vygenerované číslo mezi 10 a 42, neumí se rozhodnout a mlčí.

Konstrukce `$((...))` je *aritmetická expanze*. Shell výraz uvnitř vyhodnotí a nahradí ji jeho hodnotou, přitom se k výrazu chová, jako by byl ve dvojitých uvozovkách (expanduje proměnné, vyhodnotí zpětné apostrofy a výrazy `$(...)`). Pokud najde jméno proměnné (bez dolaru), přečte si její hodnotu. POSIX vyžaduje, aby se v proměnných hledaly aspoň konstanty s volitelným znaménkem, Bash jde ještě dál. Pokud obsah proměnné je možné interpretovat jako výraz (třeba i jen jako jméno jiné proměnné), zkusí ho vyhodnotit. Teprve když se mu to nepovede, vyhlásí chybu.

```
X=21+21
Y=X
Z=Y
echo $((Z))
echo $(($Z))  # funguje, jen zbytečně složitěji
```

Výrazy jsou jinak přejaté z jazyka C. Podporovány jsou desítkové, oktalové a hexadecimální konstanty, většina aritmetických, bitových a logických operátorů, závorky, podmínkový operátor (dvojice `?` a `:`) a operátory přiřazení. Bash

[23] Nezapomínejte, že podmínka je příkaz!

podporuje i operátory `++` a `--`. Vyhodnocuje se ve znaménkovém celočíselném typu – norma vyžaduje `signed long` nebo větší. Proběhlá přiřazení do proměnných shell uvidí i po dokončení expanze.

Alternativou k shellové aritmetice jsou příkaz `expr`, který podle nás nemá žádné výhody, a příkaz `bc`, který má vlastní jazyk a neomezenou přesnost.

Funkce

V shellu už umíme kdeco z toho, co umí běžné programovací jazyky. Proměnné, příkazy, řídicí konstrukce, aritmetiku, vstup a výstup, ... O jednom důležitém konceptu z programovacích jazyků ale dosud nepadlo slovo. O funkcích.

V běžných jazycích jsou funkce posloupností příkazů, která má nějaké (formální) parametry. Při volání se funkci předají argumenty (skutečné parametry), které se dosadí do formálních parametrů, a příkazy se spustí. V shellu je situace úplně stejná.

Mnoho ze znalostí o skriptech, které jste si přinesli z předchozího dílu, platí i pro funkce. Funkce také dostává poziční parametry, také má proměnné `$#`, `$@`, `$1` (až `$9`), má návratovou hodnotu a volání funkce i skriptu vypadají jako každý jiný příkaz (žádné závorky kolem argumentů!).

Na rozdíl od skriptu se pro funkci nespouští zvláštní shell, její příkazy běží ve stejném procesu, který ji volá. Má tedy stejný obsah proměnné `$0`, může využívat (a trvale měnit!) i neexportované proměnné a při zavolání `exit` neskončí jen ona sama, ale i celý shell. Pro nastavení návratové hodnoty používá příkaz `return`.

Funkce se tedy chová stejně jako skript vložený příkazem `.` (bashovsky také `source`), jen se jinak definuje a nemusí být ve vlastním souboru.

Ukažme si pro ilustraci definici funkce a její volání.

```
# definice
echo_t() {
        test -t 1 || echo "$@" > /dev/tty
        echo "$@"
}
# volání
echo_t hroch > soubor
```

Tato funkce se jmenuje `echo_t`, všechny své argumenty vytiskne na svůj standardní výstup, a pokud standardní výstup nejde na terminál (je přesměrovaný a ne zrovna do terminálu), argumenty vytiskne i přímo na terminál.

Definice funkce začíná jejím jménem, následovaným kulatými závorkami. Pro jméno funkce platí stejná pravidla jako pro jméno proměnné. Jmenné prostory funkcí a proměnných jsou oddělené – můžeme mít stejně pojmenovanou funkci i proměnnou, obě budou fungovat správně. Před jménem funkce Bash dovoluje ještě (zbytečné) klíčové slovo `function`.

Za jménem a kulatými závorkami následuje téměř obyčejný složený příkaz, jak ho znáte z minula. Zvláštní je v tom, že místo aby se v něm hned expandovaly proměnné a vůbec dělo všechno, co shell se svým vstupem provádí, shell si ho zapamatuje beze změny a expanze provádí až při volání.

Za zmínku stojí, že za složeným příkazem mohou být přesměrování vstupů a výstupů, která se stanou součástí definice funkce; snadno tedy můžete napsat třeba hloupou funkci pro záznam zpráv opatřených časem pořízení záznamu:


```
log() {
        date
        echo "$@"
        echo
} >> zaznamy_chyb
```

Co kdybychom ale chtěli každý argument zalogovat jako samostatný záznam a na každý záznam mít jeden řádek? Mohli bychom použít cyklus, který projde jednotlivé argumenty. Proč bychom se ale nudili už dobře známým, když si můžeme ukázat pěknou novou utilitu?

Úkol 5 [5b]: Napište funkci, která dostane jako svůj parametr jméno souboru se zdrojovým kódem, načte a předzpracuje jej pro kompilátor a výsledek vypíše na standardní výstup.

Zdrojový kód obsahuje jednotlivé textové tokeny oddělené právě jednou mezerou. Komentáře jsou uvozené tokenem `COMMENT` a platí do konce řádku. Program je ukončený buď koncem souboru, nebo tokenem `BYE`.

Úkolem funkce je opsat zdrojový kód až do konce programu s vynecháním komentářů. Na výstupu by se neměly objevit ani žádné prázdné řádky.

Poznámka: Na tuto úlohu nepoužívejte příkazy `grep`, `sed` ani žádný jiný jazyk (AWK, Perl, ...).

Formátovaný výstup

Už jsme vám až příliš dlouho zamlčovali `printf`. Mnoho jazyků má funkci toho jména a ve všech má velmi podobný význam parametrů a stejný účel – pěkné a snadné formátování výstupu. Na rozdíl od `echo` implicitně neodřádkovává, takže často budete na konec jejího prvního argumentu muset psát `\n`. To je další rozdíl proti běžnému `echo`, umí céčkové escape-sekvence jako `\n` pro konec řádku nebo `\t` pro tabulátor (`echo` s přepínačem `-e` nebo POSIXové `echo` escape-sekvence také vyhodnocují).

První argument `printf` je šablona, do které dosazuje zbylé argumenty. Pokud je argumentů málo, domyslí si pár prázdných navíc. Pokud jich je moc, zopakuje šablonu. Toho využívá i výše slíbená vylepšená funkce `log`:

```
log() {
        d="$(date)"   # znak % neobsahuje
        printf "[$d] %s\n" "$@"
} >> zaznamy_chyb
```

V šabloně jsou formátovací direktivy, které poznáte podle znaku `%` na začátku. (Pokud procento potřebujete vypsat doslova, pište `%%`.) Za každou direktivu dosadí `printf` pozicí odpovídající argument. Běžně budete potřebovat `%s` pro string a `%d` pro desítkový zápis čísla. Mimo jiné `printf` umí i převádět čísla do šestnáctkové (`%x`) a osmičkové soustavy (`%o`). Direktivy mají kromě povinného typu (`s` pro string, ...) ještě dost nepovinných částí. Na jejich samostudium se dlouhé zimní večery náramně hodí. ;-)

Úkol 6 [2b]: Napište skript, který formátuje `/etc/passwd` do podoby tabulky.

Závěr

Tento díl byl delší než oba předchozí dohromady. Dotáhli jsme v něm ale do konce spoustu rozdělaných věcí a poskytli vám tak mnohem ucelenější pohled na UNIXový svět. Pokud bychom měli zrekapitulovat, co si z tohoto dílu máte odnést, mohl by takový seznam vypadat následovně:

- Nápověda: vyhledávání pomocí `apropos`, sekce nápovědy, `info`, norma POSIX
- Souborové systémy: `df [-Th]`, `du -h`, konvence na názvy souborů, (ne)potřebnost přípon, `file`
- Typy souborů: inodes; běžné soubory, adresáře a zařízení reprezentovaná soubory (`/dev/null`, `/dev/tty`, ...)
- Symlinky a hardlinky: `ln -s` a `ln`, `readlink`
- Uživatelé a práva: význam skupin a práv pro soubory a adresáře, `su` a `sudo`
- Řídící struktury: `true`, `false` a dvojtečka, `continue`, `break` a `exit`, vnořené cykly, expanze délky proměnné (`${#promenna}`), `$RANDOM` a aritmetická expanze
- Funkce v shellu: definice, volání, `return`, souvislost se skripty
- Formátovaný výstup: `printf`

V příštím díle se už konečně podíváme na slíbené utility pro práci s textem. Přesným obsahem se nechte příjemně překvapit. :-)

Tomáš „Palec“ Maleček

27-4-7 Nástroj pro zpracování textu 14 bodů

Již v prvním dílu jsme si řekli, že Unix byl zpočátku z politických důvodů oficiálně vyvíjen jako „nástroj pro zpracování textu“. Je pomalu načase představit si jeho schopnosti v této oblasti. Ale nejprve motivační příklad.

Příklad: Počasí

Svého času bývalo populární nechat si zobrazovat na ploše různé aktuální informace: například o počasí. Samozřejmě existovala spousta programů (zvláště na Windows), která dělala právě tohle: zobrazila na ploše informace o počasí. To je sice užitečné, ale jen omezeně. Co když si vedle toho chcete zobrazit aktuální zprávy, čas do odjezdu nejbližšího autobusu, aktuálně přehrávanou písničku, čas východu a západu slunce, kurz dánské koruny nebo cokoli jiného? A co když tyhle všechny věci chcete naopak zobrazit někde jinde, řekněme třeba na nástěnných hodinách?

Určitě chápete, že na každou z takovýchto věcí vám samostatný program nikdo nenapíše. Unixový svět se k tomu postavil trochu jinak. V něm najdeme programy, které umí zobrazit na ploše *cokoliv*,[24] a jak asi tušíte, tím *cokoliv* je v tomto případě výstup nějakého jiného programu či skriptu. To nám nabízí poměrně bohaté možnosti, neb na rozdíl od programu kreslícího na plochu, je snadné vytvořit shellový skript, který něco vypíše.

Zkusme si právě to: napsat skript, který na svůj výstup vypíše aktuální teplotu v nějakém městě. Pokud bydlíte v Praze, dobrým zdrojem informací o počasí je meteostanice Planetária ve Stromovce.[25] Relevantní kousek HTML kódu této stránky vypadá následovně:

```
<TD VALIGN="top" WIDTH="200">
<FONT [...]>aktuální teplota vzduchu</FONT>
</TD>
<TD VALIGN="top" WIDTH="150">
<FONT [...]>-2.3°C</FONT>
</TD>
```

Pokud jste nikdy o HTML neslyšeli, zkonzultujte např. Wikipedii. Pro účely seriálu bude stačit vědět, že každá webová stránka je ve skutečnosti textový soubor, který popisuje, co se má uživateli zobrazit, a právě tento textový soubor vám `curl` zmíněný níže vypíše.

Podtržený je údaj o teplotě, který bychom rádi získali, `[...]` jsou vynechané nezajímavé části. Jen vás upozorníme na obtíž se zpracováním speciálních znaků (závislých například na kódování, jako jsou diakritická písmenka, nebo třeba znak

[24] Třeba programek jménem `conky`. Ten sice počasí už umí zobrazovat sám, ale i tak je zajímavé cvičení ho to naučit po svém.

[25] `http://www.planetarium.cz/meteo/PL_meteo.htm`

stupňů). Až s nimi budeme pracovat dále v různých příkazech, budou typicky nahrazeny otazníky.

Potřebovali bychom stránku stáhnout, vybrat z ní správný řádek a z něj oddělit číselnou hodnotu teploty. A nepříliš překvapivě na každou z činností poslouží jiný nástroj.

Všechny ty internety...

Již známe spoustu zajímavých unixových nástrojů pro zpracování informací (a na konci tohoto dílu budeme znát ještě víc), ale zatím jediné, s čím umí pracovat, jsou soubory na našem disku. Kdybychom jim tak dokázali „podstrčit" data získaná z Internetu, otevře se nám nepřeberné množství nových možností, stále s těmi stejnými nástroji.

To můžeme zařídit prográmkem `curl`, který na spoustě unixových systémů najdeme ve výchozí instalaci, případně si jej lze snadno doinstalovat (i v Cygwinu). Jeho použití je přímočaré: jako parametr dostane URL webové stránky (či stáhnutelného souboru) a na standardní výstup vypíše její obsah (HTML kód). Odtud jej můžeme přesměrovat do souboru či poslat kamkoli rourou, jak jsme zvyklí.

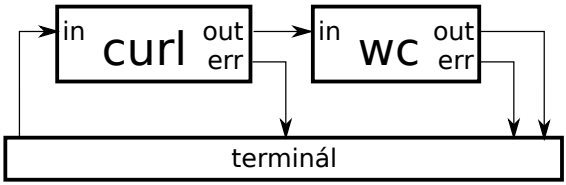
Zkuste si spustit například

```
curl http://ksp.mff.cuni.cz/ | wc -c
```

Asi vás překvapí, že kromě očekávaného výstupu se objeví několik podivných řádek, které vypadají jako informace o průběhu stahování. Jak je to možné, když je výstup příkazu `curl` přesměrován?

Ve skutečnosti má každý proces kromě svého standardního vstupu (též *stdin*) a výstupu (*stdout*) ještě třetí „datový kanál", takzvaný *standardní chybový výstup* (*stderr*). Ten směřuje na terminál, i když je výstup příkazu přesměrován do souboru či roury. Jak již název napovídá, slouží k tomu, že když za běhu příkazu nastane chyba, uvidí ji uživatel namísto toho, aby se zapsala doprostřed výstupního souboru. Ale nepoužívá se jen pro chyby a varování, nýbrž i pro informace o stavu a průběhu programu. A právě program `curl` na něj vypisuje informace o průběhu stahování, díky čemuž když stahujete velký soubor příkazem `curl http://adresa >soubor`, vidíte, kolik již je staženo a kolik času ještě zbývá.

Naše *pipeline* (tímto pojmem se označuje řada příkazů pospojovaných rourami) ve skutečnosti z pohledu operačního systému vypadá takto:



Teď už je jasné, kudy se ona hlášení na terminál dostanou. I stderr se dá přesměrovat, a to operátorem `2>`. Nejčastěji se to používá k umlčení všech hlášení, například takto: `curl http://... 2>/dev/null | wc -c`. O speciálním souboru `/dev/null` byla řeč v minulém díle. Je třeba dávat pozor, ke kterému příkazu přesměrování patří. Pokud byste ho napsali na konec, kýženého efektu nedosáhnete, neb přesměrujete stderr procesu `wc`, nikoli `curl`. V případě `curl`-u ale přesměrování používat nemusíte, neboť nabízí přepínač `-s`, který stavové zprávy utiší.

Na závěr dodáme, že ke stejnému účelu jako `curl` lze použít i příkaz `wget`, který ovšem ve výchozím nastavení ukládá staženou stránku do souboru. K vypisování na stdout ho přimějete parametrem `-O -`, hlášení o průběhu umlčíte `-q`. Více v manuálové stránce.

Filtrování řádek: grep

Občas by se nám hodilo z textového souboru vybrat řádky splňující nějaké kritérium. Nejjednodušším nástrojem, který takovou věc dělá, je `grep`. Ten jako parametr přijímá slovo (kus textu), čte postupně řádky ze svého vstupu a na výstup vypisuje jen ty, které zadané slovo obsahují (myšleno obsahují jako podřetězec, nemusí být oddělené mezerami). S parametrem `-v` (inVert) naopak vypisuje jen řádky **ne**obsahující dané slovo. Pokud hledané slovo obsahuje nějaké „divné“ znaky, je třeba `grep`-u dát ještě parametr `-F`. Které přesně znaky jsou divné a proč, si vysvětlíme později, prozatím můžete za bezpečná považovat písmena a číslice. S parametrem `-i` ignoruje `grep` při hledání slova velikost písmen (v závislosti na nastavení systému nemusí fungovat pro české znaky).

Pokud dáte `grep`-u víc parametrů, další jsou brány jako názvy souborů, ve kterých se má hledat. Tedy `grep slovo soubor1 soubor2` se chová podobně jako `cat soubor1 soubor2 | grep slovo`, s tím rozdílem, že pokud je soubor víc než jeden, `grep` před každý vyhovující řádek napíše název souboru, ze kterého pochází (to se dá vypnout přepínačem `-h`). Například kdybych chtěl zjistit, zda jsme v některém díle seriálu zmiňovali proměnnou `$RANDOM`, použiji:

```
$ grep -F '$RANDOM' serial*
serial3.tex:X=$(($RANDOM % 100))
[...]
```

a vidím, že to bylo ve třetí sérii. Pokud nás zajímají jen názvy souborů, ve kterých se slovo vyskytuje, můžeme použít přepínač `-l` (dobře se pamatuje podle `ls`, které taky vypisuje názvy souborů), případně `-L` pro opačnou operaci (výpis souborů neobsahujících ani jednou dané slovo). S přepínačem `-r` můžeme `grep`-u předávat za parametry i názvy adresářů; v těch pak prohledá všechny soubory rekurzivně. `grep -lr bagr ~` najde v domovském adresáři všechny soubory obsahující slovo `bagr`. Takovéto hledání může chvíli trvat.

Přepínačem `-C` K (Context) řeknete `grep`-u, že má kromě vyhovujících řádek vypsat i K řádek okolo každé z nich. Pokud místo `-C` použijete `-B` (Before)

či `-A` (After), bude vypsán jen kontext jednostranný (K řádek před, resp. za každým vyhovujícím). Pokud by se kontexty překrývaly, jsou slity do jednoho bloku, žádný vstupní řádek se na výstupu neobjeví dvakrát. Tedy např. příkaz `grep -C 999 :` na `/etc/passwd` vypíše to samé, co `cat`. Pokud na sebe sousední kontexty nenavazují, jsou odděleny řádkem s dvěma pomlčkami pro snazší vizuální orientaci.

Pokud používáte `grep` ručně, mohl by vás zajímat přepínač `--color`, který výskyty hledaného slova barevně zvýrazní. Ve skriptech by se vám naopak mohl hodit přepínač `-q`, který způsobí, že se na výstup nevypíše nic (jako `>/dev/null`). K čemu je taková věc dobrá? Zatím jsme vám zatajili, že `grep` vrací nulovou návratovou hodnotu, pokud našel alespoň jeden vyhovující řádek, jinak nenulovou. Takže můžete použít `if grep -q slovo soubor` pro test, zda je v souboru obsaženo slovo.

Teď už máme asi vše potřebné k výběru správného řádku:

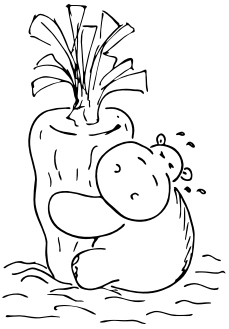
```
$ curl -s ... | grep -a -A 3 "teplota vzduchu" \
              | head -n 4 | tail -n 1
<FONT COLOR="#FFFF00" FACE="Arial">0.4?C</FONT>
```

Od začátku psaní se trochu oteplilo. ;-)

První `head` je potřeba, protože se text na stránce vyskytuje vícekrát (a kvůli podivně kódovaným českým znakům nemůžeme hledat slovo „aktuální“). Přepínač `-a` slouží k tomu, aby `grep` vypsal obvyklý výstup, i pokud považuje soubor za binární (obsahuje nějaké neobvyklé znaky).

Ještě snad dodejme, že zpětné lomítko na konci prvního řádku znamená, že příkaz pokračuje na řádku následujícím. Do skriptu to můžete napsat přesně takto, jen je třeba dát si pozor, aby za lomítkem nebyla žádná mezera. Pokud byste chtěli příkaz spustit v terminálu, je lepší napsat vše na jeden řádek (a lomítko vynechat).

Sekání a slepování řádek: cut, paste a spol.



Už umíme vybírat ze souboru celé řádky. Občas by se nám mohlo hodit získat i jejich části. K tomu nám poslouží příkaz `cut`. Ten se nejčastěji používá se soubory „tabulkového“ charakteru (např. `/etc/passwd`), kde každý řádek je rozdělen na několik „sloupečků“ nějakým oddělovačem (v Unixu typicky dvojtečka či libovolná posloupnost bílých znaků – s tou si ale `cut` neporadí). Důležité přepínače jsou `-d`, který nastavuje oddělovač (musí být jednoznakový) a `-f`, jenž říká, které sloupečky chceme na výstupu. Jeho parametrem může být jedno číslo, rozsah čísel (`3-5`), případně seznam čísel a rozsahů oddělených čárkami (`1,3,5-10`). Sloupečky jsou číslovány od jedničky.

Kombinací příkazů `cut` a `grep` můžeme nyní třeba zjistit ID uživatele `hroch`:

```
$ grep hroch /etc/passwd | cut -d: -f3
4242
```

To ale není úplně spolehlivé, například se to rozbije, pokud se někdo bude jmenovat `druhyhroch`; později si ukážeme lepší způsob. Stejně jako spousta jiných příkazů, pokud `cut` dostane jako parametr jeden nebo více názvů souborů, čte z nich, jinak čte ze standardního vstupu. Zapisuje vždy na standardní výstup. Sloupečky nejde prohazovat: `-f 3,1` je to samé, jako `1,3`.

`cut` má ještě alternativní režim, kdy místo sloupečků vysekává z řádků jednotlivé znaky, např. `cut -c 1-3` vybere z každého řádku první tři znaky.

Inverzní operací ke `cut` je `paste`, který dostane za parametry několik souborů, které považuje za jednotlivé sloupečky. Pak vezme první řádek z každého souboru a všechny spojí zadaným oddělovačem (`-d`), čímž vznikne první řádek výstupu. A tak dále pro další řádky. S ním bychom mohli, byť trochu neobratně, zařídit například zmiňované prohození sloupečků:

```
$ cut -d: -f1 /etc/passwd >jmena
$ cut -d: -f3 /etc/passwd >uid
$ paste -d: uid jmena
0:root
4242:hroch
[...]
```

Formát „sloupečků“ oddělených dvojtečkami je sice příjemný pro strojové zpracování, ale člověku se čte o dost hůře než opravdové sloupečky, kde jsou odpovídající si hodnoty zarovnané pod sebou. Takovýto oku přívětivý formát lze vyrobit programem `column`.

Ten pracuje ve dvou režimech. Prvním z nich je tabulkový (`column -t -s` *oddělovač*), ten načte ze vstupu soubor v „oddělovačovém“ formátu a na výstup ho vypíše jako hezkou tabulku:

```
cut -d: -f 1-4 /etc/passwd | column -s: -t
root   x  0     0
hroch  x  4242  4242
[...]
```

Sloupcový režim očekává na vstupu jednoduchý seznam položek, které vypíše na výstup ve vícesloupcové sazbě, podobně, jako to dělá `ls`. Tím se dá šetřit místo na obrazovce, pokud jednotlivé vstupní řádky jsou krátké. Například kdybychom chtěli vypsat seznam všech uživatelských jmen v systému:

```
$ cut -d: -f1 /etc/passwd | column
root     hrosik   hacker
hroch    guest    nobody
```

Počet sloupců je zvolen automaticky, dá se ovlivnit přepínači, stejně jako se dá zařídit, aby se hodnoty vyplňovaly po řádkách namísto po sloupcích (používejte jen pokud víte, co děláte, čte se to hrozně).

Nahrazování znaků: tr

V nejjednodušším použití `tr` nahradí všechny výskyty jednoho znaku (určeného prvním parametrem) na svém vstupu za jiný (druhý parametr) a výsledek vypíše na výstup. Každý z parametrů může být i celým seznamem znaků, zadaným buď vyjmenováním těsně za sebou, rozsahem (např. `a-z`) nebo kombinací obojího. Pak platí, že každý výskyt nějakého znaku z prvního seznamu se nahradí za odpovídající znak z druhého seznamu.

Například `tr a-z A-Z` převede svůj vstup na velká písmena (funguje jen pro znaky anglické abecedy, ostatní nechá beze změny), `tr a-zA-Z A-Za-z` prohodí velikost písmen (z velkých udělá malá a z malých velká). Pokud některý ze seznamů je kratší, je doplněn zopakováním posledního znaku. Např. `tr aeiouy e` nahradí všechny (malé) samohlásky v textu za e.

S přepínačem `-d` očekává `tr` jen jeden seznam znaků a všechny znaky na tomto seznamu ze vstupu smaže. Přepínač `-c` vezme místo prvního seznamu znaků jeho doplněk. Nejčastěji se používá spolu s `-d` pro smazání všech znaků kromě zvolených nebo s jednoznakovým pravým seznamem. `tr -c a-zA-Z _` nahradí všechny znaky kromě písmen za podtržítka.

Řešení: Počasí

Nyní už máme všechny střípky k vyřešení našeho úvodního příkladu:

```
$ curl -s \
  http://www.planetarium.cz/meteo/PL_meteo.htm \
        | grep -a -A 3 "teplota vzduchu" \
        | head -n 4 | tail -n 1 \
        | cut -d'>' -f2 |cut -d'<' -f1 \
        | tr -cd '0-9.-'
-0.7
```

Takovéto číslo si pak můžeme nejen nechat někde zobrazit, ale také s ním libovolně dál pracovat. Například by nebylo těžké napsat skript, který se před vypnutím počítače podívá na aktuální teplotu, a pokud je méně než 15, zobrazí upozornění „Vezmi si bundu!“

Rest: seq

Při povídání o for-cyklech jsme vám zatajili velmi užitečný příkaz `seq`. `seq` A B na svůj výstup vypíše všechna čísla od A po B (obojí včetně, A lze vynechat, pak se vypisuje od jedničky), každé na samostatný řádek. Spolu s `for`-em se dá použít pro zopakování nějakých příkazů n-krát.

Přeuspořádání řádků: sort, shuf, tac

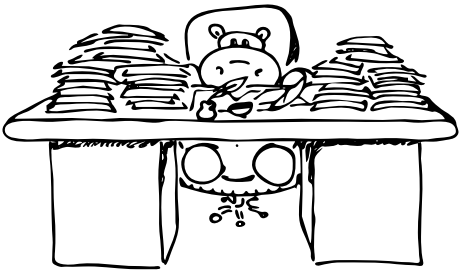
Představíme si skupinu programů, která nějakým způsobem mění pořadí řádků na svém vstupu. Nejdůležitějším z nich je `sort`, který setřídí vstup či soubor dle zadaných kritérií. Ve výchozím nastavení třídí řádky abecedně vzestupně. Můžeme použít přepínače `-r` pro sestupné třídění, `-f` pro ignorování velikosti písmen, `-n` pro číselné porovnávání (abecedně by se zatřídila „100“ před „11“) a `-k` pro třídění podle některého sloupečku (ve stejném významu jako u `cut`-u, oddělovač se nastavuje `-t`, výchozím oddělovačem je whitespace, tedy libovolná posloupnost bílých znaků). Například `sort -t: -k3 -n /etc/passwd` setřídí záznamy v `/etc/passwd` podle ID uživatele.

S parametrem `-R` `sort` místo třídění vstupní řádky náhodně zamíchá. To samé dělá `shuf`, jen nabízí nějaké parametry navíc, například vybrat ze vstupu náhodně jen K řádek (`-n` K) nebo povolit vybrat jeden řádek vícekrát (`-r`).

Například pokud byste byli učitel a měli v nějakém souboru uloženou hromadu otázek, ze kterých chcete vygenerovat 30 náhodných písemek po 10 otázkách, můžete napsat:

```
for i in ‘seq 30‘; do
    shuf -n 10 otazky.txt >pisemka-$i.txt
done
```

`tac` vypíše řádky vstupního souboru v opačném pořadí (od posledního po první) a uvádíme jej zde hlavně, abyste se taky mohli těšit z kreativity, kterou do názvů příkazů raní unixoví programátoři vložili ;-) Tak trochu duálním příkazem k `tac` je `rev`, který pořadí řádků nemění, ale zato každý z nich napíše pozpátku. Takže pokud byste chtěli obrátit celý soubor (od posledního znaku po první), `tac | rev` zařídí přesně to.



Třídit soubory nemusíme jen z vlastního zájmu, ale také proto, že některé příkazy svůj vstup setříděný vyžadují. Jedním z nich je `uniq`, který ze vstupu vyhodí duplicitní řádky, ale pouze, pokud leží všechny duplikáty vedle sebe (což zařídíte právě setříděním). `uniq` má spoustu zajímavých přepínačů, jako např. `-c`, který před každý výstupní řádek napíše, kolikrát se vyskytl na vstupu. Třeba pokud máme dlouhý seznam jmen uživatelů a chtěli bychom vědět, která křestní jména se u nich vyskytují nejčastěji, můžeme použít:

```
$ sort jmena.txt | cut -d' ' -f 1 \
  | uniq -c | sort -nr | head -n 3
     64 Jan
     51 Martin
     42 Kateřina
```


S parametrem `-d` se naopak vypisují pouze duplicitní řádky (každý jen jednou), `-i` při porovnávání ignoruje velikost písmen a mnoho dalších zajímavých parametrů najdete v manuálové stránce. Protože `sort | uniq` je tak častá kombinace, existuje za ni zkratka `sort -u`.

Porovnávání souborů: comm, cmp, diff

Dalším příkazem, kterému se hodí setříděný vstup, je `comm`. Slouží pro porovnání dvou množin reprezentovaných řádky setříděných souborů. Ve výchozím nastavení vypíše výstup do tří sloupečků: v prvním jsou řádky obsažené pouze v prvním souboru, v druhém řádky unikátní pro druhý soubor a ve třetím řádky oběma souborům společné. To je hezké pro vizuální porovnání, ale ve skriptech víceméně nepoužitelné. Vhodným nastavením parametrů můžeme zajistit vypsání jen jednoho z těchto sloupečků, ale syntaxe není příliš intuitivní: parametrem `-`n říkáme, že n-tý sloupeček **ne**chceme zobrazit. Takže typické použití je např. `comm -12 soubor1 soubor2` pro zobrazení řádků vyskytujících se v obou souborech.

Pokud chceme dva soubory jen porovnat na shodnost (pro to už pochopitelně nemusí být setříděné), můžeme použít příkaz `cmp`. Ten skončí s nulovou návratovou hodnotou, pokud je obsah souborů shodný, jinak s nenulovou (dá se tedy použít v rámci příkazu `if`). Při použití ve skriptech doporučujeme přidat parametr `-s`, aby se při neshodě nevypisovalo informativní hlášení.

Dalším nástrojem pro porovnávání souborů je `diff`, který umí zobrazit „v čem“ se dva soubory liší. Nejčastěji se používá pro porovnání dvou verzí téhož souboru, například když přemýšlíme, proč stará verze našeho programu fungovala a nová už ne. Doporučujeme používat přepínač `-u` (zapne trochu smysluplnější výstupní formát). S `-N -r` lze porovnávat rekurzivně celé adresáře. Zkuste si s ním pohrát.

Výstupu programu diff se obvykle říká buď „diff“ (mezi nějakou verzí a nějakou jinou), nebo „patch“. Druhé označení je odvozeno od příkazu `patch`, který s `diff`-em úzce souvisí. Vstupem příkazu `patch` je stará verze souboru

a diff mezi starou a novou verzí, výstupem pak zrekonstruovaná nová verze. Například po spuštění

```
diff -u A B >AB.diff
patch -o C A AB.diff
```

bude mít soubor `C` stejný obsah jako `B`. To se dá používat k snadnému šíření úprav: pokud třeba uděláte malou změnu ve velkém programu a chcete se o ni s někým podělit, nemusíte mu posílat celý kód, stačí jen patch.

Ještě větší výhodou je to, že patch lze obvykle aplikovat, i když se mezitím původní soubor (`A` v našem příkladu) změnil. Tohle umožňuje několika lidem nezávisle na sobě změnit nějaký soubor a poté všechny změny automaticky sloučit. Stačí, když každý vyrobí patch mezi společnou původní verzí a svou upravenou verzí a nakonec se všechny tyto patche na soubor postupně aplikují. Problém nastane pouze v případě, že se dva lidé pokusí změnit stejnou část souboru; v takovém případě dojde k takzvanému *konfliktu*, který je třeba vyřešit ručně. Ale to se stává překvapivě málo často.

Na tomto principu je založen vývoj mnoha open source projektů. Přispěvatelé si lokálně program mění a testují, a hotové změny posílají autorům ve formě patchů. Ale i vytváření a aplikování patchů a udržování historie vývoje je spousta ruční práce, pročež se tyto procesy snaží automatizovat takzvané verzovací systémy, jako např. git.[26]

Příklad: Spam

Někteří jste si již možná všimli, že v KSPčku většinu hromadných mailů posíláme každému s vlastním oslovením, včetně správně vyskloňovaných tvarů slov dle pohlaví adresáta. Jak to děláme? Obvykle k mailu vytvoříme šablonu, prostý textový soubor, který může vypadat třeba takto:

```
Mil[ý|á] $osloveni,

byl[a] jsi vybrán[a] jako $co na soustředění...
```

a pak seznam lidí, kterým se má poslat:

```
kc@example.net:Květoslave Čeňku:M:náhradník
po@example.net:Pokusná Osobo:F:účastník
hroch@example.net:Hrochu:M:maskot
```

A zbytek zařídí jednoduchý shellový skript. Pojďme si ho zkusit napsat. Stačilo by nám vymyslet, jak vyrobit z šablony mail pro jednoho konkrétního adresáta, hromadné zpracování už pak snadno zajistíme nějakým `while read`.

Potřebovali bychom umět v textu nahradit všechny řetězce v nějakém tvaru (např. `[`*`něco`*`|`*`něco`*`]`) za jiné řetězce, a ještě k tomu v náhradě použít nějaké části řetězce původního.

[26] `http://git-scm.org/`

Regexy

Regex alias regulární výraz je řada písmenek a speciálních znaků, která právě dokáže popsat „řetězce nějakého tvaru“, jako v příkladu výše. O libovolném řetězci lze rozhodnout, jestli danému výrazu *vyhovuje* (má správný tvar), nebo nikoli. Regex tak vlastně popisuje nějakou (potenciálně nekonečnou) množinu řetězců. S něčím podobným jsme se již setkali: byly to wildcardy. Například nahrazovaný řetězec z příkladu výše bychom se mohli pokusit popsat wildcardem `\[*\|*\]`. Ale možnosti wildcardů jsou poměrně omezené; s regexy se dají dělat mnohem zajímavější věci.

Na straně 147 najdete seznam konstrukcí použitelných v regexech. Literál je libovolná posloupnost znaků, které nemají speciální význam (nejsou použity v prvním sloupečku tabulky).

Kulaté závorky lze vynechat tam, kde by se v nich nacházel jediný znak nebo jiný nedělitelný element (např. množina), v případě alternativ, když by měly být kolem celého regexu.

Pár příkladů:

- `[a-zA-Z_][a-zA-Z0-9_]*` vyhovuje platný identifikátor, jak je definován většinou programovacích jazyků – tedy neprázdná posloupnost písmen, číslic a podtržítek nezačínající číslicí.
- `([01]?[0-9]|2[0-3]):[0-5][0-9]` vyhovuje čas zapsaný v 24-hodinovém formátu (např. `0:42`).

Podrobnější úvod do regexů najdete v seriálu 23. ročníku[27] či v desítkách internetových tutoriálů.

Nepleťte si regexy s wildcardy! Sice řeší podobný problém (popis množiny řetězců), ale jinak spolu nemají nic společného. Například pozor na to, že `*` v regexech je kvantifikátor, který značí opakování toho, co stojí před ní, samostatně stojící `*` nemá smysl. Obdobou wildcardové hvězdičky je regex `.*`. Liší se také použitím: wildcardy interpretuje shell a dají se použít pouze pro hledání názvů souborů, regexy se obvykle předávají nějakým pomocným utilitám a používají se pro hledání kusů textu.

Použití regexů: hledání a nahrazování

Kde lze regexy v Unixu použít? Například v nám dobře známém příkazu `grep`. Pokud mu místo přepínače `-F` (Fixed, hledej pevný řetězec) dáme `-E`, hledá řetězec vyhovující nějakému regexu. E znamená Extended a zapíná takzvanou rozšířenou syntaxi regexů (tu jsme si vysvětlili v předchozí kapitole). Existuje ještě „základní“ syntaxe (starší), která se používá, pokud `grep`-u nedáte žádný přepínač. Ta je ale matoucí a nekonzistentní, tak ji raději nepoužívejte. Za `grep -E` existuje zkratka `egrep`.

[27] `http://ksp.mff.cuni.cz/viz/23-1-7`

Nezapomeňte, že `grep` hledá řádky *obsahující* řetězec vyhovující regexu, ne řádky celé *vyhovující* regexu. Například řádek `abcd` nevyhovuje regexu `[a-z]`, ale obsahuje `a`, které mu vyhovuje, a tedy `echo abcd | grep -E '[a-z]'` jej vypíše. Pokud bychom chtěli hledat řádky celé vyhovující regexu, stačí použít `^`*regex*`$`.

Slíbili jsme spolehlivější zjištění ID uživatele `hroch`, zde je:

```
grep -E '^hroch:' /etc/passwd | cut -d: -f3
```


Tady hledáme slovo `hroch` pouze na začátku řádku a následované dvojtečkou, nezmate nás tedy uživatel `druhyhroch` či `hrochodlak`, ani někdo, kdo si nastaví jako shell `/bin/hrochsh`.

`grep` má ještě jeden přepínač, který je užitečný až s regexy: `-o`. Ten zajišťuje, že se nevypisují celé vyhovující řádky, nýbrž jen nalezené výskyty regexu. Pokud je na jednom řádku více výskytů, každý se vypíše na samostatný výstupní řádek. Například takto bychom mohli najít seznam všech identifikátorů (názvů funkcí, proměnných apod.) použitých v nějakém programu:

```
grep -Eio '[a-z_][a-z0-9_]*' program.c | sort -u
```

Pro plnou funkčnost bychom ještě museli odfiltrovat komentáře, stringové literály a klíčová slova. K tomu by nám mohl pomoci nástroj, který si představíme za chvíli.

Zamysleme se ještě nad jednou věcí. Pokud je v programu například identifikátor `bagr`, nachází se uvnitř něj i další platné identifikátory, jako např. `ag`. `grep -o` v takovém případě udělá to, co téměř vždy chceme: z každé množiny překrývajících se výskytů vypíše pouze ten nejlevější a nejdelší (což bude přesně odpovídat identifikátorům v programu doopravdy použitým).

Při předávání regexů jako parametrů příkazům je třeba dát si **velký pozor** na to, že mnoho regexových speciálních znaků má zvláštní význam i pro shell, a tedy pokud chceme, aby se např. ke `grep`-u dostaly nezměněné, musíme je

Syntaxe	Význam	Příklad	Vyhovují	Nevyhovují
literál	Řetězec shodný s literálem.	`bagr`	`'bagr'`	`'bagrovat'`, `'lopata'`
speciální-znak	Escapuje speciální znak (udělá z něj literál).	`\*`	`'*'`	`'\*'`, `'\'`
`.`	Libovolný znak.	`.`	`'a'`, `'%'`	`'abc'`, `''`
`[`*množina*`]`	Libovolný znak patřící do množiny (jako v shellových wildcardech)	`[a-z_.]`	`'q'`, `'.'`	`'A'`, `'-'`, `'aa'`
(*regex1*`|`*regex2*`|`…)	Řetězec vyhovující alespoň jedné z možností.	`hro(ch|sik)`	`'hroch'`, `'hrosik'`	`'hrosi'`
(*regex*)`*`	Nula nebo více opakování.	`.*`	`''`, `'abc 123'`	
(*regex*)`+`	Jedno nebo více opakování.	`(ab)+`	`'ab'`, `'abab'`	`''`, `'baba'`
(*regex*)`?`	Nepovinný prvek (0-1 opakování).	`[1-9]?[0-9]`	`'5'`, `'42'`	`'01'`, `'333'`
(*regex*)`{`*m*`}`	*m* opakování.	`[A-Z]{2}`	`'AA'`, `'ZQ'`	`'X'`, `'ERF'`
(*regex*)`{`*m*`,`*n*`}`	*m* až *n* opakování (obojí včetně).	`[0-9]{1,3}`	`'4'`, `'007'`	`'1337'`, `''`, `'xyz'`
`^`	Začátek řádku.			
`$`	Konec řádku.			

před shellem zaescapovat. Nejjednodušším řešením je psát všechny regexy do apostrofů, kde se nic escapovat nemusí.

Pokud přece jen z nějakého důvodu escapovat budeme, je třeba si uvědomit, že máme co do činění se dvěma úrovněmi escapování. Například pokud napíšeme `grep ^\\\* soubor`, nejprve dostane řetězec do rukou shell, který ví, že `\\` ve skutečnosti znamená `\` atp., všechny escapy odstraní a `grep`-u předá jako první parametr řetězec `^\*`. `grep` zase ví, že `\*` znamená literál „`*`“, tedy tento příkaz tedy vybere ze souboru řádky začínající hvězdičkou.

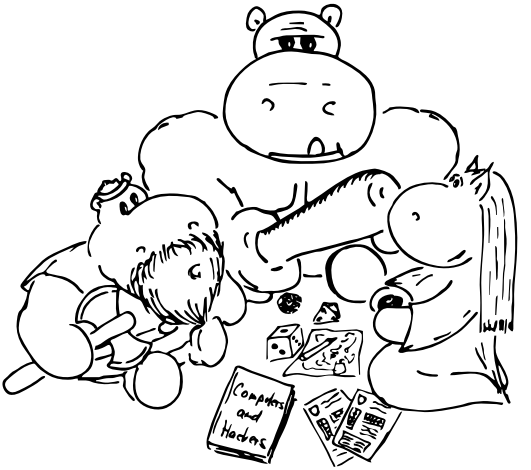
Když už umíme výskyty regexů v textu hledat, hodilo by se také umět je nahrazovat něčím jiným. Řešení si představíme zatím jen jako zaklínadlo `sed -re 's/`*regex*`/`*náhrada*`/g'`. Nebojte, za chvíli si ho vysvětlíme.

Jak jsme slibovali na začátku, v textu náhrady se lze odkazovat na části původního textu: přesněji na obsah libovolné kulaté závorky. Obalit závorkou můžeme cokoliv, aniž by se tím změnil význam regexu. Počítají se všechny závorky, včetně těch vynucených např. kvůli ohraničení skupiny alternativ. Obsah závorky do náhrady vložíme speciální sekvencí `\`*číslo*, kde číslo je pořadové číslo závorky (přesněji řečeno, páry závorek se číslují od jedničky v pořadí jejich otevíracích závorek). To mimo jiné znamená, že i v náhradě musíme escapovat zpětná lomítka, pokud je tam chceme vložit doslovně.

Například pokud chceme v textu nahradit slovo `bagr` ve všech tvarech za slovo `kombajn`, můžeme použít příkaz

```
sed -re 's/bagr(|u|em|y|ů|ům|ech)/kombajn\1/g'
```

Tenhle trik samozřejmě funguje pouze, pokud mají slova stejný (pod)vzor a žádné nepravidelnosti.



Řešení: Spam

Nyní už máme téměř vše, co potřebujeme k řešení spamovacího příkladu. Můžeme si jej zkusit načrtnout. Předpokládejme, že v shellových proměnných `$osloveni`, `$pohlavi` a `$co` máme příslušné údaje k vyplnění.

```
tvar=$(echo $pohlavi | tr MF 23)
<sablona sed -re "s/\\\$osloveni/$osloveni/g" \
   | sed -re "s/\\\$co/$co/g" \
   | sed -re 's/\[((.*)\|)?(.*)\]/\'$tvar/g
```


Jak to funguje? První dva `sed`-y jsou přímočaré nahrazení jednoho řetězce za jiný, jen pozor na escapování dolarů (od shellu i regexu). Poslední regex hledá řetězce tvaru `[`*`něco`*`|`*`něco`*`]`, kde každé „něco“ načte do jedné závorky. V proměnné `$tvar` pak máme číslo závorky, kterou chceme vybrat. Parametr `sed`-u bude po expandování proměnných vypadat např. jako:

```
s/\[((.*)\|)?(.*)\]/\3/g
```

První část („*něco*`|`“) je nepovinná, pokud mužský tvar není uveden, předpokládá se prázdný.

Toto řešení skoro funguje, ale ještě ne úplně. Zapomněli jsme totiž upozornit na jednu věc: regexy jsou *žravé*. To znamená, že při hledání vždy vyberou nejdelší kus textu, který jim vyhovuje. Tedy například pro text `Mil[ý|á] účatn[íku|ice]` nebudou nalezeny očekávané dva výskyty, nýbrž jeden velký, kde v první závorce skončí „`ý|á] účatn[íku`“ a v druhé „`ice`“. Snadno si rozmyslíte, že to vyhovuje našemu regexu. Nejjednodušeji to spravíme tak, že uvnitř jednotlivých tvarů zakážeme používat `|` a `]`. Tedy místo `.*` musíme psát `[^]|]*` (stříška na začátku množiny znamená doplněk a pokud napíšeme `]` jako první, neukončí se tím množina, nýbrž do ní vložíme znak `]`).

Nyní už to opravdu dělá, co má. Celý rozesílací skript by mohl vypadat takto:

```
cat adresati \
| while IFS=: read mail osloveni pohlavi co; do
t=$(echo $pohlavi |tr MF 23)
<sablona sed -re "s/\\\$osloveni/$osloveni/g" \
 | sed -re "s/\\\$co/$co/g" \
 | sed -re 's/\[(([^]|]*)\|)?([^]|]*)\]/\'$t/g
 | mail -s "Pozvánka na soustředění" "$mail"
done
```

Kde `mail` je jedním z mnoha příkazů, které umožňují odeslat e-mail. K tomu samozřejmě potřebuje správné nastavení, které je nad rámec tohoto seriálu. Parametrem `-s` určujeme předmět.

sed pro pokročilé

Je na čase naše sedové zaklínadlo rozklíčovat. `sed` ve skutečnosti dostane text a aplikuje na něj posloupnost příkazů. Tu mu můžeme předat buď jako parametr s použitím přepínače `-e` *příkaz*, nebo načíst ze souboru parametrem `-f` *soubor* (hodí se pro delší a složitější příkazy; v souboru nemusíme řešit shellové escapování). Jednotlivé příkazy se oddělují středníkem nebo koncem řádku. Též můžeme více příkazů zadat několikanásobným použitím příkazu `-e`.

Přepínač `-r` zapíná rozšířenou syntaxi regexů, podobně jako u `grep`-u `-E`. Doporučujeme používat v podstatě vždy.

Všechny příkazy mají jednoznakový název. My jsme dosud používali příkaz `s` (substitute) sloužící k nahrazování. Ten má tvar `s/`*regex*`/`*náhrada*`/`*modifikátory*. Místo lomítek můžeme použít jakýkoli jiný znak (kromě písmen). Můžeme se tak vyhnout problémům s regexy obsahujícími lomítka. Oblíbenými volbami jsou např. `|` či `#` pro svou vizuální výraznost. Modifikátor `g` (global) znamená nahrazení všech výskytů na řádku (jinak by se na každém řádku nahradil jen první). Dalším zajímavým modifikátorem `i` (ignoruj velikost písmen).

`sed` zpracovává vstup po řádcích a na každém z nich zvlášť provede všechny příkazy v pořadí, ve kterém jsou zapsány. Ve skutečnosti se každý řádek načte do řetězcové proměnné, které se říká *pattern space*, na ní se provádějí jednotlivé příkazy a po jejich skončení je výsledná hodnota pattern space vypsána na výstup, není-li `sed` spuštěn s parametrem `-n` („nevypisovat“).

Před většinu příkazů lze napsat takzvanou *adresu* a tím určit, že se budou provádět jen na některém řádku či řádcích. Adresou může být například:

- *číslo* – Této adrese vyhovuje právě tolikátý řádek vstupu, počítáno od jedničky.
- `$` – Poslední řádek.
- `/`*regex*`/` – Řádek obsahující výskyt regexu. Chceme-li použít místo lomítka jiný oddělovač, musíme na začátek napsat backslash, např. `\#/dev/null#` adresuje řádky obsahující řetězec „`/dev/null`“.
- *adresa*`!` – Negace adresy, vyhovují řádky nevyhovující původní adrese.

Nyní už má smysl vysvětlit si některé další příkazy, které bez adresování nejsou příliš užitečné:

- `p` – vypíše aktuální obsah pattern space na výstup. Nejčastěji se používá spolu s parametrem `-n` pro selektivní vypisování řádků. Například použít příkaz `sed -nre '/`*regex*`/ p'` je to samé, jako `grep -E '`*regex*`'`. Kombinací `s/^.*$/`*text*`/` a `p` můžeme na výstup vypsat libovolný text.
- `d` – „smaže“ řádek (zabrání jeho vypsání a skočí na další). To znamená, že `sed -re '/`*regex*`/ d'` funguje jako `grep -vE`.
- `y/`*znaky*`/`*znaky*`/` – nahradí znaky z jednoho seznamu za odpovídající znaky z druhého, stejně jako `tr`, včetně stejného zápisu výčtů a rozsahů.

Příkazu můžeme dát místo jedné adresy také dvojici adres oddělených čárkami. Tím zadáváme rozsah – příkaz se provádí na všech řádcích od první adresy po druhou. Přesněji řečeno kdykoli se narazí na řádek vyhovující první adrese, příkaz se začne provádět a když se narazí na řádek vyhovující druhé adrese, zase se provádět přestane. Rozsah vždy zahrnuje oba krajní řádky (vyhovující příslušným adresám). Takto se může příkaz provést i pro několik bloků řádek, pokud každý z nich začíná řádkem vyhovujícím první adrese a končí řádkem vyhovujícím druhé.

Některé formáty (například e-mailové zprávy) mají takovou strukturu, že obsahují nejprve hlavičky (s informacemi jako odesilatel, předmět atp.), potom prázdný řádek, a teprve za ním tělo (text zprávy). Pokud bychom chtěli blok hlaviček odstranit, můžeme použít příkaz:

```
sed -re '1,/^$/ d'
```

Ukázky, jak pomocí `sed`-u nahradit `grep`, nebyly samoúčelné. `sed` totiž oproti `grep`-u má jednu velkou výhodu, totiž přepínač `-i` (inplace). Již v prvním díle jsme se bavili o tom, že nemůžeme z jednoho souboru zároveň načítat a zároveň do něj zapisovat (`grep slovo <soubor >soubor` neudělá to, co byste chtěli). `sed -i` tohle umí zařídit. Jen mu dáte jako parametr (tedy nikoli shellové přesměrování) název souboru, a on z něj načte vstup a do toho samého souboru uloží svůj výstup. Interně to dělá tak, že vytvoří dočasný soubor, do kterého výstup zapisuje, a po svém skončení s ním nahradí (pomocí ekvivalentu příkazu `mv`) atomicky původní soubor. Tedy `sed -i -re ... soubor` je ekvivalentní posloupnosti příkazů:

```
sed -re ... <soubor >soubor.tmp
mv soubor.tmp soubor
```

Toto je velmi častý unixový idiom, který je dobré si zapamatovat. Hodí se nejen když chceme zapisovat do stejného souboru, ze kterého čteme, ale víceméně kdykoli nahrazujeme či vytváříme nějaký soubor. Tím, že data zapisujeme do nějakého dočasného souboru, který kromě nás nikdo jiný nepoužívá, a teprve když je „hotový“, jej přesuneme na cílové místo, se nemůže stát, že se nějaká jiná aplikace pokusí číst soubor v průběhu vytváření, kdy ještě neobsahuje smysluplná data. Také pokud můžeme snadno zajistit (bashovým operátorem `&&`), že pokud náš příkaz modifikující nějaký soubor selže, `mv` se neprovede a zůstane zachovaná původní verze.

sed pro šílence

Se `sed`-em se dají dělat i větší šílenosti. Jeho příkazy tvoří vlastně jednoduchý programovací jazyk.[28] Kromě pattern space máte k dispozici ještě druhou(!) stringovou proměnnou, které se říká *hold space*. Pomocí příkazů `h` a `H` můžete

[28] Je turingovsky úplný, ale asi v podobném smyslu, jako Brainfuck. Dá se najít implementace Turingova stroje v `sed`-u. Nebo Sokoban. Zagooglete si.

aktuální obsah pattern space zapsat do hold space, resp. připojit na jeho konec. `g` a `G` dělají to samé opačným směrem. Při připojování je nový obsah od původního oddělen znakem konce řádku (`\n`). Příkazem `x` lze prohodit obsah pattern a hold space.

Pokud bychom si chtěli řídit načítání řádek nějak přesněji, než že pro každý řádek je náš program spuštěn znovu od začátku, můžeme. K tomu slouží příkazy `n`, který do pattern space další řádek (původní zahodí) a `N`, který připojí další řádek na konec pattern space (oddělený `\n`, podobně jako `G`). To vše se děje stále v rámci jedné iterace našeho skriptu. Pokud doběhne na konec, `sed` automaticky načte první ještě nenačtený řádek do pattern space a spustí náš program znovu od začátku. Tyto příkazy nám umožňují nezpracovávat jednotlivé řádky jen nezávisle, ale dělat i nějaké složitější úpravy napříč řádky.

Proměnné už máme, ale správný programovací jazyk ještě potřebuje nějaké řídící konstrukce. `sed` nabízí návěští (`:` *název*), skoky (`b` *název*) a podmíněné skoky (`t` *název*). Podmíněný skok se provede, pokud od posledního podmíněného na tomto řádku vstupu došlo k alespoň jednomu úspěšnému nahrazení příkazem `s` (existuje i verze `T`, která má podmínku invertovanou).

Pokud chceme testovat, zda uspěl jeden konkrétní nahrazovací příkaz, musíme si nejdřív případné dřívější úspěchy na daném vstupním řádku „vyresetovat“ prostřednictvím příkazu `t`:

`t reset; : reset; s/`*regex*`/`*náhrada*`/; t cil;`

Nechceme-li nic nahrazovat a rádi bychom jen skákali podle toho, zda pattern space vyhovuje nějakému regexu, můžeme použít příkaz `b` podmíněný adresou: `/`*regex*`/ b`.

Všechny verze skoku při vynechání argumentu skáčou za poslední příkaz. `q` a `Q` ukončí celý `sed` s vypsáním aktuálního pattern space, resp. bez něj. Umí nastavit návratovou hodnotu. Dále existují příkazy `r` a `w` pro čtení/zápis z/do pomocných souborů. Příkazy `[qQT]` jsou rozšířením GNU `sed`-u (nejběžněji se vyskytující verze) a nemusí být dostupné v jiných verzích.

Zkusme si například napsat skript, který spojuje příkazy rozdělené na několik řádek pomocí zpětných lomítek do jednoho řádku:

```
sed -re ': loop; s/\\$//; T; N; s/\n//; t loop'
```

Ukázkový vstup:

```
prvni
dr\
uh\
y
```

Ukázkový výstup:

```
prvni
druhy
```

Už byste měli zvládnout si rozmyslet, jak funguje.

Další nástroje

Posledním významným nástrojem, který jsme nezmínili, je `awk`. To by nejspíš vydalo na samostatný díl. Slouží primárně k složitější práci s tabulkovými soubory. Na rozdíl od `cut`-u umí používat složitější než jednoznakové oddělovače, např. libovolnou posloupnost bílých znaků (ta je u `awk` dokonce výchozím oddělovačem). To se hodí při práci se soubory, jako je `/etc/fstab`. Program v `awk` je podobně jako v `sed`-u posloupnost příkazů, která se spouští pro každý řádek a příkazům lze předřadit podmínku omezující, na kterých řádcích běží. Jazyk `awk` má daleko blíže k plnohodnotnému programovacímu jazyku než sed: má pojmenované proměnné, asociativní pole a další vychytávky.

Uvedeme jen několik málo příkladů použití, měly by být zčásti samovysvětlující, zčásti interpretovatelné s pomocí manuálové stránky:

- Vyseknutí sloupečku odděleného obecnou posloupností bílých znaků z tabulky:

```
awk '{print $2}' /etc/fstab
```

- Nalezení uživatele s daným ID:

```
awk -F: '($3 == 0) { print $1 }' /etc/passwd
```

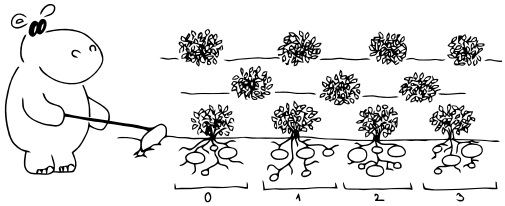
- Sečtení všech (číselných) řádků v souboru:

```
awk '{ sum += $1 } END { print sum }'
```

Postavení `awk` na půli cesty mezi jednoduchou utilitkou a plnohodnotným programovacím jazykem z něj činí trochu zvláštní nástroj. Někdy je lepší použít místo něj `cut` či `sed`, jindy naopak opravdový programovací jazyk. Na zpracování textu nejlépe poslouží Perl, který má bohatou podporu pro regexy a spoustu syntaktických zkratek, jež v něm umožňují většinu jednoduchých věcí napsat podobně krátce jako v jednoúčelových nástrojích.

Jakou malou ochutnávku Perlu vám ukážeme skript, který z HTML dokumentu vypíše titulky všech hypertextových odkazů:

```
perl -0777 -ne 'for (m{<a.*?>.*?</a>}gcs) {
    s/<.*?>//g; s/(^\s+|\s+$)//g;
    print "$_\n" if $_;
}'
```



Úkoly

V řešení se nebojte používat pomocné soubory, kde je to na místě, klidně pro jednoduchost s pevnými jmény. Můžete používat vše, co jsme se naučili, a další utilitky podobného ražení, můžete používat `awk`. Perl ani jiné „velké“ programovací jazyky nepoužívejte.

Úkol 1 [2b]: Ve vstupním souboru máte seznam jmen (sudý počet, jedno na řádek). Napište skript, který z nich vytvoří náhodné dvojice a vypíše je na výstup ve formátu *první osoba*`:`*druhá osoba*.

Ukázkový vstup:

```
A
B
C
D
```

Ukázkový výstup:

```
C:A
B:D
```

Úkol 2 [4b]: Napište skript řešící úlohu 27-Z3-2.[29] Ve vstupním souboru dostanete slovník (jedno slovo na řádku), na výstup vypište nejdelší slovo, které má ve slovníku i svou verzi napsanou pozpátku.

Ukázkový vstup:

```
kecup
ves
vrabec
pucek
sev
```

Ukázkový výstup:

```
kecup
```

Řešení s pomocí bashových cyklů je nudné, pro plný počet bodů to zkuste bez nich. Mohlo by se vám hodit `awk` a jeho funkce `length`.

Úkol 3 [4b]: V nějakém adresáři máte staženou spoustu dílů svého oblíbeného seriálu (z legálních zdrojů, pochopitelně). A jak už to tak u legálních zdrojů chodí, soubory jsou pojmenované naprosto neuspořádaně. Jediné, čím si můžete být jisti, je, že název obsahuje číslo série a číslo epizody v tomto pořadí, mezi nimi je alespoň jeden nečíselný znak a pro jednoduchost název žádné jiné číslice neobsahuje.

Napište skript, který stáhne z Wikipedie (či jiného příhodného zdroje) názvy dílů a všechny soubory přejmenuje tak, aby v jednotném formátu obsahovaly číslo série a číslo a název epizody. Chcete-li, můžete předpokládat, že všechny soubory jsou ze stejné série.

Možná se vám snáz než HTML bude parsovat zdrojový wikitext, který získáte připojením `?action=raw` na konec URL článku.[30]

[29] `http://ksp.mff.cuni.cz/viz/27-Z3-2`

[30] `http://en.wikipedia.org/wiki/The_Big_Bang_Theory_(season_4)?action=raw`

Úkol 4 [4b]: Vylepšete příklad vypisující všechny identifikátory v Céčkovém programu tak, aby ignoroval obsah řetězců a komentářů, případně základní klíčová slova. Pro plný počet bodů by měl zvládnout jednořádkové (`// ...`) i víceřádkové (`/* ... */`) komentáře a neměl by se nechat zmást escapovanými uvozovkami a zpětnými lomítky v řetězcích. Ale určitě má smysl poslat i jednoduché či částečné řešení. Obskurnosti jako komentář uvnitř řetězce (nebo naopak) ošetřovat nemusíte, pokud vyloženě nechcete. Předpokládejte samozřejmě, že program je syntakticky správný.

Ukázkový vstup:

```
/* print a greeting
   with quoted name */
printf("Hello, \"%s\"",
       name);
```

Ukázkový výstup:

```
name
printf
```


Pokud si s Céčkem nerozumíte, můžete si vybrat nějaký jiný srovnatelně složitý (tedy třeba by měl ideálně mít víceřádkové komentáře či řetězce) programovací jazyk – třeba Python nebo Pascal.

Z cvičných důvodů zkuste nenačítat celý vstupní soubor do paměti najednou. Ano, bylo by to jednodušší a pro praktické účely možná nejlepší řešení, ale tolik se toho na něm nenaučíte.

Filip Štědronský

27-5-7 Shellová automatizace — 15 bodů

Poslední díl letošního seriálu o UNIXu a jeho příkazovém řádku se ponese v duchu automatizace úkonů a lepšího provázání našich skriptů se systémem. Ukážeme si třeba, jak lze spustit stejný příkaz na všech souborech nějakého typu, jak v nějakém složitějším procesu zpracování dat (nebo třeba kompilace programů) zpracovávat jen to, co ovlivní změněné soubory, a také jak například zajistit, aby déle běžící skript po sobě uklidil, pokud se ho rozhodneme ukončit v průběhu práce.

Všechno to jsou drobné pomůcky, které nám krásně zapadnou do všeho ostatního, co jsme se přes rok naučili, a pomohou vám ještě lépe využívat sílu shellu. Pokud se tedy nebojíte, račte vstoupit.

Hledání souborů

V minulých dílech jsme vám ukázali, jak třeba pomocí wildcardů vybrat všechny soubory v aktuální složce s příponou `.pdf`. To pomocí nich umíme jednoduše, horší to ale začíná být ve chvíli, kdy chceme prohledat rekurzivně třeba i všechny podsložky včetně jejich podsložek a tak dále.

Asi by se dal napsat nějaký skript, který by si nechal vypsat příkazem `ls` všechny složky a na nich by se zavolal znova, ale existuje mnohem snadnější řešení. Zkuste si třeba ve svém domovském adresáři spustit příkaz `find`.

```
.
./poznamky.txt
./Obrazky
./Obrazky/Kevin.jpg
./Obrazky/Sara.jpg
./Obrazky/Petr.jpg
./Reseni
./Reseni/KSP
./Reseni/KSP/27-5-7.pdf
...
```

Jak vidíte, `find` vám vypsal úplně všechny složky a soubory ležící ve stromě souborů pod aktuálním umístěním. Pokud mu totiž nezadáme, jakou složku má prohledávat, tak použije aktuální adresář `.` (a také ho vypíše jako první prohledaný a připojí ve výpisu před všechny nalezené složky a soubory). Kdybychom ve stejném umístění spustili třeba příkaz `find Reseni`, výpis by pak vypadal takto:

```
Reseni
Reseni/KSP
Reseni/KSP/27-5-7.pdf
```

To je pěkné, na takovýto výstup bychom už mohli použít třeba příkaz `grep` a vyfiltrovat z něj s trochou práce třeba jen PDF soubory. Ale `find` to umí sám a ještě spoustu věcí navíc.[31]

Než se pustíme do dalšího experimentování, tak se na příkaz `find` podíváme i se všemi jeho skupinami parametrů:

```
find <kde hledat> <kritéria> <prováděný příkaz>
```

- První skupina je asi jasná, určuje místo, kde má `find` začít své prohledávání. Je možné předat i více umístění, `find` je prohledá všechna. Pokud není žádné umístění zadané, tak se použije aktuální adresář `.`, jak jsme ukázali výše.

[31] Dokonce může být vnější filtrování pomocí `grep`u výrazně pomalejší, protože se mezi těmito dvěma příkazy musí přenést skrz *rouru* velké množství dat.

- Druhá skupina parametrů nastavuje různá kritéria omezující výběr souborů a složek. Pokud není nic nastaveno, nefiltruje se nic a vypisují se všechny nalezené složky a soubory. Tím se budeme zabývat vzápětí.
- Třetí skupina parametrů určuje, co se má pak s nalezenými názvy souborů a složek dít. Pokud nezvolíme žádnou akci sami, tak `find` použije akci `-print` a vše jen vypíše na výstup. Můžete si zkusit ho s touto akcí na konci jeho seznamu parametrů spustit.

Mezi další jeho akce patří pak třeba formátovaný výpis nebo spuštění nějakého příkazu. Tomu se budeme věnovat po kritériích výběru.

Mocnější find

Co kdyby nás zajímaly všechny `README` soubory třeba ve složce `/etc`? V tu chvíli nám stačí použít kritérium `-name` a spustit příkaz:

```
find /etc -name README 2>/dev/null
```

Protože složka `/etc` obsahuje pravděpodobně několik podsložek, na jejichž čtení nebudeme mít práva, může nám `find` vynadat několika chybovými hláškami (jednou za každou nepřístupnou složku). Aby se nám výstup mezi těmito hláškami neztratil, je dobré vzpomenout si na minulé díly seriálu a chybový výstup „odfiltrovat“ jeho přesměrováním do `/dev/null`, jak jsme v příkladu výše rovnou udělali.

Můžete dokonce použít i shellové wildcardy ke specifikování názvu. Jen pozor na to, že wildcardy se musí dostat až k `find`u, nesmí je tedy zpracovat už samotný shell a tedy je nutné je buď escapovat, nebo zabalit do uvozovek:

```
find -name "*.pdf"
```

Pokud by vám nestačily shellové wildcardy, je možné podobným způsobem použít i regulární výrazy, ale už s jiným přepínačem. Následující příkaz najde všechny PDF soubory začínající od písmene `a`:

```
find -regex ".*/a[^/]*\.pdf"
```

Mezi další zajímavá kritéria patří například specifikace typu souboru pomocí přepínače `-type` (`-type d` je složka, `-type f` běžný soubor, více v manuálové stránce). Velmi pěkné je také filtrování podle toho, kdy byl soubor naposledy modifikovaný. Viz následující příklady:

```
find -mtime 7    # Modifikace v posledním týdnu
find -mmin 10    # Modifikace v posledních 10min
```

Další možné filtrování je například podle vlastníka nebo podle přístupových práv. Dokonce umí hledat i podle čísla inode, a tedy lze použít k nalezení všech hardlinků na konkrétní soubor (hardlinky jsme zmiňovali ve třetím díle seriálu). Další užitečný přepínač, který sice není standardizovaný, ale Linuxová verze ho podporuje, je `-maxdepth 2` omezující hloubku prohledávání.

Závěrem povídání o `findu` se zmíníme o dalších možných akcích. S defaultní akcí `-print` jsme se už potkali, ta vytiskne nalezené soubory oddělené znakem nového řádku. Kdybychom očekávali, že se nám může ve filesystému objevit soubor se znakem nového řádku v názvu, mohla by se nám hodit akce `-print0`, která jednotlivé soubory na výstupu odděluje nulovým bajtem.

Další akce je třeba `-delete`, které nalezené soubory smaže, `-printf`, které zvládá tisknout formátovaný výstup, nebo `-exec`, které spustí daný příkaz pro všechny nalezené soubory. Na jejich použití a na více vyhledávacích kritérií se podívejte do manuálové stránky, zde ukážeme jen jednoduché příklady:


```
# Otevření všech HTML stránek ve Firefoxu:
find -name "*.html" -exec firefox '{}' \;
# Přidání přípony .txt všem souborům:
find -exec mv {} {}.txt \;
```

Konstrukce příkazu pomocí xargs

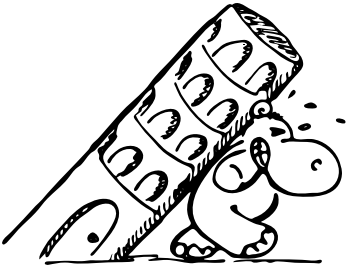
Jak padlo výše, tak `find` umí spouštět pro každý nalezený soubor nějaký příkaz, ale dělá to bohužel pro každý soubor samostatně. Pokud bychom například chtěli všechny takto nalezené soubory smazat, někam zkopírovat, nebo přidat do společného archivu, bude to zbytečně pomalé nebo komplikované.

Tento problém ale řeší příkaz `xargs`. Ten v podstatě dělá to, že vezme svůj standardní vstup (který může přijít třeba od příkazu `find` skrz rouru) a použije ho jako argumenty pro zadaný příkaz (tento příkaz samozřejmě musí podporovat proměnlivý počet argumentů).

Smazání všech PDF souborů třeba můžeme v kombinaci s příkazem `find` udělat takto:

```
find -name "*.pdf" | xargs rm
find -name "*.pdf" -print0 | xargs -0 rm
```

Argumenty jsou připojené na konec zadaného příkazu a ten je vykonán. Druhý řádek je bezpečnější, pokud by se nám v názvech souborů vyskytovaly mezery nebo znaky nového řádku, ale jinak dělá úplně to samé. Prostě jen `find` i `xargs` přepneme do módu oddělování null-bytem, o kterém z minulých dílů víme, že se v názvu souboru vyskytnout nemůže.



Možná vás ale napadla otázka: Co když nechceme argumenty připojit na konec konstruovaného příkazu? Co když, třeba jako u příkazu `cp` chceme jako poslední argument mít název složky, do které chceme zkopírovat nalezené soubory?

V takovou chvíli využijeme přepínače `-I`, kterým příkazu `xargs` nastavíme výraz, jež pak bude v konstruovaném příkazu nahrazen argumenty. Tradičně se používá výraz `{}`, ale není problém použít cokoliv jiného. Dva příkazy níže jsou tedy ekvivalentní:

```
find -name "*.pdf" | xargs -I {} cp {} ~/backup/
find -name "*.pdf" | xargs -I F cp F ~/backup/
```

Závěrem povíme, že `xargs` předává argumenty příkazu najednou, pokud jich není moc. Samotný systém má totiž jistá omezení a třeba spuštění příkazu `rm *` ve složce obsahující příliš mnoho (třeba milióny) souborů už vám neprojde. Příkaz `xargs` ale tato omezení zná a rozsekává příkazy po správně velkých blocích. Pokud tedy neprojde příkaz výše, stačí spustit `find | xargs rm`, `xargs` sám spustí několik příkazů `rm` a každému předá jen zvládnutelný počet argumentů.

Pokud si nejsme jisti počtem argumentů, které dostaneme na vstupu, a nechceme příkaz spouštět pro nulový počet argumentů, můžeme příkazu `xargs` přidat parametr `-r` říkající „naprázdno nedělej nic“.

Úkol 1 [1b]: Spočtěte počet řádek ve všech souborech s příponou `.txt` ležících přímo v aktuálním adresáři nebo v jeho podadresářích (do libovolné hloubky). Výstupem by mělo být jediné číslo.

Úkol 2 [2b]: Najděte všechny prázdné podadresáře aktuálního adresáře (do libovolné hloubky).

Úkol 3 [2b]: Změňte všem souborům v aktuálním adresáři (nebo jeho podadresářích do libovolné hloubky) s příponou `.tvuj` příponu na `.muj`. Myslete i na to, že se v názvech mohou objevit podivné znaky.

Procesy a paralelizace

Pokud jste si na práci v terminálu trochu zvykli a spouštěli jste ve větším množství i nějaké déle běžící programy, možná vás napadlo, že by bylo dobré pouštět je paralelně – nemuseli byste tak čekat, až předchozí skončí. Operační systém samozřejmě umí pouštět víc příkazů, zatím jsme si ale pořádně neukázali, jak na to v shellu.

Dosud jsme se setkali s rourou. Pokud příkazy oddělíme `|`, shell je spustí současně a správně na sebe napojí jejich vstupy a výstupy. (Detaily si můžete přečíst ve čtvrtém dílu seriálu.) Další příkaz z našeho skriptu ovšem shell vykoná, až když všechny propojené rourou skončí.

Procesy v popředí i na pozadí

Nechceme-li čekat, stačí za příkaz napsat ampersand `&`. Pomocí něj spustíme danou úlohu *na pozadí*. V zásadě `&` můžeme oddělovat příkazy podobně jako pomocí středníku nebo konce řádku. Rozdíl je v tom, že u ampersandu nebude shell čekat, až se daná úloha dokončí. Standardní a chybový výstup však stále povedou na terminál.

Pokud by se však úloha běžící na pozadí rozhodla číst ze vstupu, má problém. Na terminál je už připojený standardní vstup shellu, případně vstup nějaké jiné úlohy běžící *v popředí*. Úloha na pozadí proto bude zastavena, dokud ji znovu nepustíme.

Pozastavenou úlohu můžeme spustit na popředí pomocí vestavěného příkazu shellu `fg` (foreground), či na pozadí `bg` (background). Pokud byla úloha pozastavena, protože chce číst ze vstupu, jejím spuštěním na pozadí ji okamžitě pozastavíme znovu. Příkaz `bg` ale má stále své využití. I úlohy běžící v popředí totiž můžeme z terminálu snadno pozastavit. Většinou stačí zmáčknout `Ctrl+Z`. Hned si vysvětlíme, jakým způsobem toto pozastavování funguje.

* * *

Připomeňme si ještě, co dalšího o UNIXových procesech víme. Letmo jsme se s nimi seznámili ve druhém díle. Pověděli jsme si, že systém si pro každý proces pamatuje jeho identifikační číslo PID, stav, seznam otevřených souborů, uživatele, s jehož právy běží, a mnoho dalších informací. Seznam všech běžících procesů dostaneme příkazem `ps ax`. Tyto znalosti využijeme v další části.

Pokud bychom chtěli počkat na dokončení některého procesu běžícího na pozadí, poslouží nám k tomu interní příkaz `wait PID`. Pokud žádný parametr nedostane, počká jednoduše na všechny podprocesy.

Signály a meziprocesová komunikace

Možná se ptáte, jakými prostředky spolu vůbec mohou procesy komunikovat. Používali jsme již rouru (a to jak nepojmenovanou, tak pojmenovanou vzniklou příkazem `mkfifo`.) Z shellu si ale můžeme snadno vyzkoušet ještě jeden komunikační kanál. Jsou jím signály.

Signál si můžeme představit jako takové malé šťouchnutí. Procesy si je mohou mezi sebou navzájem posílat a tím si vlastně předávat informace. Rozhodně to není způsob, kterým byste chtěli přenášet kilobyty dat (natož více). Signály slouží spíš k upozorňování na asynchronní události. Na běžném dnešním Linuxu jich najdeme 64, na OpenBSD jen 32.

Každý signál má své jméno a číslo. Pozor na to, že různé operační systémy mohou mít přiřazení čísel signálů různé.

A k čemu se signály hodí? Například při vypínání počítače by bylo dobré dát všem programům vědět, že se mají vypnout a případně uložit rozdělanou práci. K tomu se používá signál SIGTERM. Pokud by na to program nereagoval a nechtěl se vypnout, můžeme jeho činnost ukončit natvrdo signálem SIGKILL.

Podobně existují také signály SIGTSTP a SIGSTOP, které způsobí zastavení běžícího procesu a naopak SIGCONT, pro opětovné spuštění. SIGTSTP pošleme procesu právě pomocí stisku `Ctrl+Z`. Naopak `Ctrl+C` posílá SIGINT.

Za zmínku stojí ještě SIGHUP a SIGCHLD. Kdykoli nějaký proces skončí, je na to upozorněn jeho rodič signálem SIGCHLD. Stejné upozornění přijde i v případě, že je synovský proces pozastaven, případně znovu spuštěn. SIGHUP má hned několik významů. Tento signál obdrží programy, pokud jim zavřeme terminál, ve kterém běží. Většina aplikací se proto ukončí. U *daemonů*, programů určených k tomu, aby běžely na pozadí a s člověkem nekomunikovaly pomocí terminálu, SIGHUP obyčejně způsobí znovunačtení konfigurace z konfiguračních souborů.

Chceme-li procesu poslat signál, stačí z shellu zavolat `kill [-signál] PID`. Pro označení signálu můžeme použít jak číslo, tak název. Pokud signál nespecifikujeme, posílá se SIGTERM. Jako identifikátor procesu můžeme zvolit i `-1`. Potom je daný signál poslaný úplně všem procesům, kterým můžeme nějaký signál poslat. Signály totiž můžeme posílat pouze svým vlastním procesům. (Jenom root má výjimku a umí signalizovat všem.)

Signály mají za sebou bouřlivou historii a jejich význam se průběžně trochu měnil. Například někdy narazíme na to, že signály nezačínají na SIG. Máme pak INT, TERM, TSTP,... Pokud by vás zajímaly detaily, podívejte se do manuálových stránek `man 7 signal`. V zásadě můžeme signály dělit podle toho, jestli proces ukončí, ukončí a vytvoří obraz jeho paměti (tzv. core dump), pozastaví, znovu spustí, případně jestli jsou ignorovány a nedělají nic.

U většiny signálů si proces může výchozí chování přenastavit. Jedinou výjimkou jsou SIGKILL a SIGSTOP – ty vždy znamenají to samé. Díky tomu jde každý proces ukončit, případně zastavit. Typicky chceme některé signály ignorovat, nebo odchytit vlastní funkcí. Pokud pak náš proces obdrží signál, operační systém přeruší aktuálně vykonávanou práci a spustí naši funkci. Signály můžeme odchytávat i v shellu.

Číháme na signál

Abychom mohli signál zachytit, potřebujeme na něj nejprve nalíčit past. Jednoduše pomocí `trap` řekneme, co se má provést v případě, že daný signál přijde. Například po zavolání `trap "echo baf" SIGUSR1 SIGINT` shell vypíše na svůj výstup nápis „baf“ vždy, když obdrží signál SIGUSR1 nebo SIGINT. Zavolání `trap` bez parametrů vypíše, jaké příkazy bude shell při kterém signálu provádět.

Úkol 4 [3b]: Napište skript, který vám pomůže se smysluplným využitím dnešních vícejádrových počítačů naplno. Váš skript dostane jediný parametr N. Na standardním vstupu pak bude číst podobně jako shell příkazy a bude je spouštět. Řádky by však měl provádět paralelně. Vždy se smí provádět nejvýše N řádek současně.

Proč /proc?

Aby nebylo nutné vytvářet pro zjištění všech možných informací specializovaná systémová volání, vznikl v Linuxu virtuální filesystém `/proc`. V proc najdeme pro každý spuštěný proces adresář s celou řadou zajímavých souborů. Například v `/proc/PID/fd` najdeme jako symlinky seznam všech souborů otevřených daným programem. Podrobnosti najdete v manuálu `man 5 proc`.

KSP

seriál

Úkol 5 [2b]: Napište malou náhradu programu `ps ax`. Na výstupu vašeho skriptu by se měl objevit o každém procesu jeho PID, nějaký identifikátor uživatele, příkaz i se všemi jeho parametry a alespoň jeden další údaj podle vašeho výběru. Můžete si vybrat cokoli, co se vám bude zdát aspoň trochu zajímavé či užitečné. Všechna potřebná data čtěte přímo z `/proc`.

Make – základy

Poslední velký pomocník, kterého si letos ukážeme, je příkaz `make`. Ten se stará o automatizaci procesu nějaké kompilace, překladu nebo třeba jen zpracování dat. Většinou je řízen souborem s názvem `Makefile` (všimněte si velkého prvního písmena) v adresáři, kde `make` zavoláme.

A co že přesně umí? Základem celého procesu je, že se `make` pokouší splnit nějaké v `Makefile` definované cíle, což většinou znamená vyrobit nějaké soubory. Pokud Makefile zjistí, že požadovaný cíl ještě neexistuje, zkusí ho podle pravidel uvedených v `Makefile` vyrobit. Pravidla pro výrobu cíle se v něm zapisují odsazená tabulátorem pod názvem cíle. Takový jednoduchý `Makefile` tedy může vypadat takto:

```
datum.txt:
	echo Datum: > datum.txt
	date >> datum.txt
```

Pokud tedy ve složce s tímto `Makefile` zavoláme příkaz `make datum.txt`, tak se provedou příkazy definované výše a vznikne nám tento soubor. Pokud ale zkusíme teď stejný příkaz zavolat znova, tak se už nic neprovede – `make` už totiž vidí, že je tento cíl splněn (soubor existuje) a že tedy není potřeba nic vyrábět.

Make a závislosti a virtuální cíle

Nejsilnější zbraní, kterou ale `make` disponuje, jsou závislosti. Dá se prohlásit, že cíl závisí na určitých souborech, například že vyráběný soubor se statistikou závisí na zdrojových datech měření. Když je pak `make` požádán o splnění nějakého cíle, tak se nejdříve pokusí splnit všechny závislosti.

Pro každou závislost se podívá, jestli neexistuje stejně pojmenovaný cíl, a zkusí ho také splnit. Takto může rekurzivně postupovat prakticky do neomezené hloubky.

Poté, co má nějaký cíl splněné všechny své závislosti, tak se `make` ještě rozhodne, jestli je nutné provádět i tělo tohoto cíle. Pokud soubor stejného jména, jako je jméno cíle, ještě neexistuje, není proč váhat a tělo se provede. Pokud ale takový soubor už existuje, je to zajímavější. Pak se `make` podívá na časy modifikace všech souborů, na kterých aktuální cíl závisí, a porovná je s časem modifikace již existujícího souboru.

Když zjistí, že již existující soubor je novější než všechny jeho závislosti, tak není potřeba dělat nic. Pokud se ale nějaká závislost změnila od doby jeho vytvoření (tedy je alespoň jedna závislost s novějším časem modifikace, než má již existující soubor), tak se tělo cíle provede.

Abychom si to ukázali na příkladě, tak uvažujme následující `Makefile` používaný ke generování seznamu kapitol a nějakých statistik ze sepisované knížky:

```
all: kapitoly.txt statistika.txt

kapitoly: knizka.txt
        grep "^Kapitola:" <knizka.txt >kapitoly

statistika.txt: knizka.txt kapitoly
        echo Počet řádků > statistika.txt
        wc -l knizka.txt >> statistika.txt
        echo Počet kapitol >> statistika.txt
        wc -l kapitoly >> statistika.txt

clean:
        rm -f statistika.txt
        rm -f kapitoly

.PHONY: all clean
```

Jako první jste si asi všimli podivných cílů `all` a `clean`. Oba jsou to takzvané virtuální cíle, tedy jim neodpovídají žádné skutečně vytvářené soubory, ale slouží pro speciální účely. Aby `make` nebyl zmaten, kdyby se přece jen objevily soubory tohoto jména, tak mu to raději sdělíme mírně magickou formulí `.PHONY:` na konci ukázky – ta zajistí to, že `make` bude stejně se jmenující soubory ignorovat a vyjmenované cíle brát vždy jako virtuální a vždy se provedou jejich těla, jako by soubory neexistovaly.

Nyní můžeme spustit příkazy `make all` pro výrobu všeho, na čem cíl `all` závisí, nebo `make clean` pro odstranění pracovních souborů. Cíl `all` je navíc uveden jako první proto, že když zavoláme jen `make` bez cíle, provede se první cíl.

Když jsme teď ukázali pár nových triků, pojďme se vrátit k závislostem. Řekněme, že jsme už provedli `make all` a máme tedy v adresáři všechny tři soubory. Když teď zavoláme `make kapitoly` nebo `make statistika.txt`, tak se nic nestane. Při volání `make statistika.txt` se sice `make` rekurzivně zavolá na splnění cíle `kapitoly`, ale protože ten nevygeneruje žádný novější soubor, tak se neprovede ani tělo cíle `statistika.txt`.

Pokud ale nejdříve změníme obsah souboru `knizka.txt`, začne to být mnohem zajímavější. Po opětovném zavolání `make statistika.txt` se `make` znovu pokusí obstarat všechny jeho závislosti. Pro soubor `knizka.txt` žádný cíl nemá a tedy ho bere tak, jak je, ale pro cíl `kapitoly` cíl existuje a tak se ho pokusí rekurzivně splnit.

Při plnění cíle `kapitoly` zjistí, že jsou závislosti novější, než je vygenerovaný soubor, a tedy spustí tělo příkazu `kapitoly` a vygeneruje tak nový soubor. Cíl `statistika.txt` pak zjistí, že dokonce obě jeho závislosti (soubory `knizka.txt` i `kapitoly`) jsou novější, než vygenerovaný soubor, a proto taky spustí své tělo a vygeneruje nový obsah souboru `statistika.txt`.

Zde je vidět, že `make` vždy dělá jen tu nejmenší nutnou práci, spouští jen těla těch cílů, jejichž zdroje, na kterých závisí, se změnily. U malého projektu je nám to asi jedno, ale kdybychom třeba `make` používali k překladu Linuxového jádra a provedli jsme změnu jen v jednom zdrojovém souboru, budeme rádi, že `make` během několika málo sekund přegeneruje jen to, co potřebuje, a nemusíme tak čekat dlouhé minuty, než by se provedlo přegenerování úplně všeho, i když to nebylo potřeba.

Make a speciální proměnné

Možná vám přijde, že třeba přejmenovat soubor se statistikami v příkladu výše by bylo docela pracné a máte pravdu. Znamenalo by to na spoustě míst přepsat jeho název na něco jiného, hlavně v těle cíle, jež ho vyrábí. Na to má ale `make` docela pěknou léčbu v podobě speciálních proměnných.

V tělech cílů se dají používat třeba tyto tři základní:

- `$@` zastupuje název cíle (jméno vytvářeného souboru)
- `$<` zastupuje název první závislosti
- `$^` zastupuje názvy všech závislostí

Klíčové cíle z dřívější ukázky by se tak daly přepsat třeba takto:

```
kapitoly: knizka.txt
        grep "^Kapitola:" < $< > $@

statistika.txt: knizka.txt kapitoly
        echo Počet řádků > $@
        wc -l knizka.txt >> $@
        echo Počet kapitol >> $@
        wc -l kapitoly >> $@
```

Velmi mocným nástrojem je také konstruování obecných cílů. Pokud bychom třeba chtěli mít obecná pravidla, která nám pro každý textový soubor (třeba pro každou knížku, kterou jako úspěšní spisovatelé sepisujeme) vygeneruje statistiku jako výše, můžeme k tomu právě tyto obecné cíle využít.

Obecný cíl vytvoříme tak, že v jeho názvu použijeme zástupnou část reprezentovanou znakem `%`. Za tu pak `make` při hledání cílů může doplnit cokoliv chce, ale zástupný znak můžeme použít jen na jednom místě v názvu cíle (nelze například vytvořit cíl `KSP-%-5-%.pdf`). Můžeme jej pak ale libovolně používat v závislostech (lze tak třeba vyjádřit, že libovolný přeložený program závisí na svém zdrojáku v jazyce C a podobně).

Když zástupný znak použijeme, tak se nám pak pro použití v těle cíle přidává ještě jedna speciální proměnná:

- `$*` obsahuje to, co se dosadilo za zástupný znak `%`

Hlavní část tohoto obecného `Makefile` by tedy mohla vypadat třeba takto:

```
%-kapitoly: %.txt
	grep "^Kapitola:" < $< > $@

%-stat.txt: %.txt %-kapitoly
	echo Počet řádků > $@
	wc -l $*.txt >> $@
	echo Počet kapitol >> $@
	wc -l $*-kapitoly >> $@
```

Teď můžeme volat třeba `make detektivka-stat.txt`, ale také například `make roman-stat.txt` a stačí nám jen, aby `detektivka.txt` a `roman.txt` existovaly.

Někdy se nám dokonce může hodit mít nějaké obecné pravidlo a pak pro jednotlivé soubory přidávat závislosti navíc. Pokud uvedeme jen hlavičku cíle bez těla obsahujícího příkazy, tak se pro tento konkrétní cíl jen přidají závislosti. Třeba příklad níže má pro soubory `prvni.txt` i `druhy.txt` definovaný stejný příkaz, ale díky speciálním proměnným se pro každý zpracuje jiný zdrojový soubor:

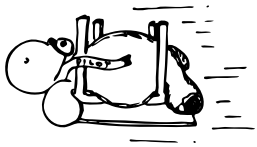
```
%.txt:
	grep "body" <$< >$@

prvni.txt: KSP_serie1.txt
druhy.txt: KSP_serie2.txt
```

Poslední věcí, kterou ve spojení s příkazem `make` zmíníme, je definování a použití vlastních konstant. V podstatě to jsou proměnné, ale doporučujeme vám používat je skutečně jako konstanty (tedy do každé přiřadit pouze jednou), jinak se to celé zamotává. Typické použití je třeba v `Makefile` pro překlad zdrojáků jazyka C, kde se jednou globálně nastaví všechny přepínače kompilátoru, a pak se používají. Hodnotu v proměnné použijete zapsáním `${nazev_promenne}`.

Ukázkový `Makefile` využívající definované proměnné, speciální proměnné i obecné cíle vidíte níže. Používá se jak verze `${promenna}`, tak i verze `$(promenna)`.

```
PROG=mujprogram
OBJS=mujprogram.o knihovna.o

CFLAGS=-Wall -c
LDFLAGS=-lm -lpthread
CC=gcc

%.o: %.c
	${CC} ${CFLAGS} -o $@ $<

${PROG}: ${OBJS}
	${CC} ${LDFLAGS} -o $@ $^
```


K tomu ještě dodejme to, že `make` zpřístupňuje i proměnné prostředí (tedy proměnné ze shellu). Tak si třeba v shellu můžeme nastavit proměnnou, která nám ovlivní prováděné příkazy. Když `make` žádnou definovanou proměnnou daného jména nenajde (ani interní, ani v prostředí), dosadí místo ní prázdný řetězec.



Úkol 6 [2b]: Představte si, že máte tři zdrojové soubory dat: `A.data`, `B.data` a `C.data`. Dále máte program `generuj <vstupy>`, kterému jako parametry můžete předhodit libovolný počet vstupních souborů a on na svůj standardní výstup vypíše nějaký vygenerovaný obsah. Ten by se měl ukládat do souborů, které budeme označovat jako `A`, `AB` a podobně (podle jejich vstupů).

Pak máte tři finální soubory, které se vytvářejí obdobným příkazem `finalizuj <vstupy>` (opět vezme libovolně mnoho vstupních souborů a vygenerovaný obsah vypíše na svůj standardní výstup). Soubor `FIN1` se vytváří ze souborů `A`, `AB` a `B`, soubor `FIN2` ze souborů `BC` a `C` a soubor `FINAL` ze souborů `A`, `AB`, `B`, `BC` a `C`.

Sepište `Makefile`, který bude odpovídat těmto závislostem (zkuste co nejvíce pravidel nějak šikovně seskupit), a pak se zamyslete (a odpovězte), co vše se přegeneruje, když postupně provedeme následující příkazy:

```
make FINAL
make FIN1
touch A.data  # změníme soubor A.data
make FIN2
touch C.data  # změníme soubor C.data
make FIN1
```

Úkol 7 [3b]: V KSPčku používáme `make` i na překlad zdrojáků v TEXu. Ale když chceme někde použít automaticky generované obsahy, nastává problém – TEX totiž generuje soubor s obsahem během generování svého výstupu, a je tak potřeba často spustit první překlad TEXu „naprázdno“ a teprve při druhém překladu použít již vygenerovaný soubor s obsahem, který se vloží na správné místo zdrojáku.

Protože se soubor s obsahem (označme ho `toc`) vkládá do zdrojáku (`tex`) a z něj se pak generuje `pdf` soubor, měly by závislosti být `toc` ← `tex` ← `pdf`. Jenže čas změny `toc` souboru se změní vždy s překladem nového `pdf` a tedy vzniká vlastně cyklus.

Zkuste vymyslet řešení a sepsat základní `Makefile`, který bude toto řešení ilustrovat. Může se vám hodit třeba příkaz `diff`.

Závěr

Dnes jste se dozvěděli další kousky velké skládačky o UNIXu a jeho příkazovém řádku. Pověděli jsme si o příkazech `find` a `xargs`, podívali se na zpracování signálů a obsah `/proc` a nakonec jsme si ukázali mocného pomocníka v podobě příkazu `make`.

Existuje ještě spousta kousků této skládačky, o kterých jsme neměli čas se zmínit, ale doufáme, že jsme vám poskytli aspoň to, co jsme považovali za základ, a pomohli vám třeba k tomu, abyste se o UNIX začali zabývat sami. Děkujeme, že jste s námi a s naším seriálem přes celý rok vydrželi. :-)

Poslední díl seriálu pro vás připravili

Jirka Setnička & Jenda Hadrava

Recepty z programátorské kuchařky

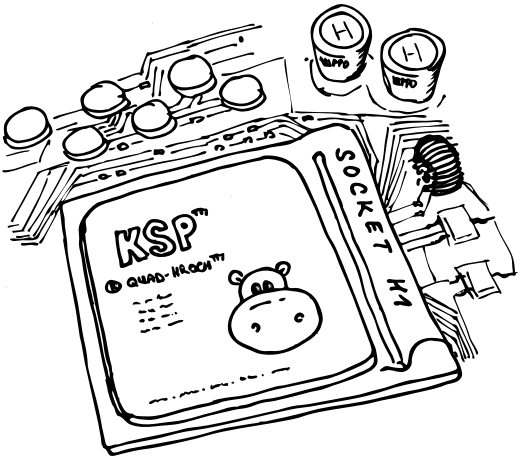
Kuchařka první série – základní algoritmy

Tato naše kuchařka je nejzákladnější ze základních a je určena hlavně pro začínající řešitele. To však neznamená, že zkušenější řešitelé do ní nahlédnout nemůžou – třeba na nějakou konkrétní programátorskou techniku, kterou by si potřebovali osvěžit.

V první části kuchařky se seznámíme hlavně se základními principy programování, uchovávání dat v počítači a základy rychlé manipulace s nimi. Po přečtení této části bychom měli být schopní převést své myšlenky z hlavy na papír či do počítače a měli bychom vědět, proč je námi zvolený postup rozumný.

Druhá část nás poté seznámí se základními postupy, jak řešit určité konkrétní problémy. Naučíme se například, jak rychle vyhledávat v uspořádané posloupnosti hodnot, nebo jak si pomocí předpočítání usnadnit řešení těžké úlohy.

Většinu klíčových částí se pokusíme též ukazovat v podobě zdrojového kódu ve dvou různých jazycích (v nízkoúrovňovém C, kde je zápis blízký tomu, jak počítač doopravdy pracuje, a v Pythonu, ve kterém se píše o něco příjemněji). Nebudeme ale probírat základy syntaxe těchto jazyků, ty si případně můžete nastudovat jinde.[32] Pokud žádný z těchto jazyků neumíte, nezoufejte. KSP můžete řešit i bez toho, stačí když svá řešení důkladně slovně popíšete (konkrétní jazyk se pak můžete naučit až během dalších sérií).



[32] `http://ksp.mff.cuni.cz/study/odkazy.html`

Část první: Základní pojmy

Algoritmus a program

Pod tajemným slovem *algoritmus* se skrývá jen jiný výraz pro postup. Můžete si to představit jako příkaz od maminky „Běž do krámu, kup chleba, a když budou mít měkké rohlíky, tak jich vem tucet“.[33]

Takovýto příkaz klidně můžeme nazvat algoritmem, ačkoliv to bude asi znít nezvykle – pojem algoritmus se totiž používá hlavně ve světě počítačů. Je to tedy nějaká posloupnost základních příkazů, která řeší nějaký problém. Výběr konkrétního programovacího jazyka rozhoduje o tom, jaké základní příkazy budeme mít k dispozici. Většinou jsou ale skoro stejné.

Mezi základní příkazy patří:

- Manipulace s daty v paměti (uložení či načtení hodnoty, detailněji v další kapitole).
- Provedení nějakého numerického výpočtu $(+, -, *, /)$.
- Vyhodnocení nějaké podmínky a odpovídající větvení programu: *Pokud platí A, tak proveď B, jinak proveď C.* Přitom B i C mohou být klidně celé *bloky kódu*, tedy libovolně mnoho dalších základních příkazů.
- Opakování nějakého příkazu: *Dokud platí A, dělej B.* Takové konstrukci říkáme *cyklus* a podobně jako u podmínky může být B blok kódu, který se celý opakuje.
- Vstup a výstup programu (typicky vstup od uživatele z klávesnice či načtení vstupu ze souboru; výstup pak může znamenat vypsání výsledku na obrazovku nebo třeba zapsání dat do souboru).

Z těchto základních stavebních kamenů se skládá každý algoritmus. Programem potom rozumíme realizaci algoritmu v nějakém konkrétním programovacím jazyce.

U složitějších programů se pak často setkáte s problémem, že budete mít nějakou posloupnost příkazů, která se bude na spoustě míst programu opakovat, což zbytečně prodlužuje a znepřehledňuje kód.

Řešením tohoto problému je použití *funkcí*. Funkci si můžeme představit jako nějakou pojmenovanou část programu (s vlastní pamětí), kterou můžeme opakovaně použít tím, že ji v různých částech programu *zavoláme*. Funkci při zavolání předáme parametry (například seznam čísel), které se dostanou do její vnitřní paměti.

Funkce pak na základě obdržených parametrů může provádět nějaké operace, při kterých pracuje se svojí vnitřní pamětí (mluvíme o *lokální* paměti, změny

[33] A jako slušně vychovaní se tedy vydáte do krámu a koupíte tucet chlebů, protože měli měkké rohlíky :-)

v ní se neprojeví nikde mimo funkci). Na konci nám funkce může vrátit nějaký výsledek. Pokud funkce během svého běhu změní i nějaká data v *globální* paměti, či provede nějakou globální operaci (například výpis textu na monitor), mluvíme pak o funkci s *vedlejšími efekty* (neboli side-efekty).

Konkrétním příkladem může být funkce, která nám spočítá odmocninu ze zadaného čísla. Ta dostane jako svůj parametr číslo, uvnitř si provede nějaký výpočet, o který se jako uživatel funkce nemusíme starat, a jako výstup nám vrátí spočtenou odmocninu.

KSP

Reprezentace dat v počítači

Celkem často si v průběhu výpočtu našeho algoritmu potřebujeme pamatovat nějaké hodnoty. K tomu nám programovací jazyky dávají nástroj s názvem *proměnná*. Ta představuje jakési pojmenované místo v paměti (přihrádku), do kterého si můžeme data ukládat a pak je odtud zase načítat.

recepty

Typickým příkladem může být počítání součtu čísel, která nám uživatel zadá na vstupu. Na začátku nejdříve do nějakého místa v paměti uložíme hodnotu 0. Poté postupně, jak nám uživatel zadává čísla, tuto proměnnou pokaždé přečteme, k její hodnotě přičteme nově zadané číslo a výsledek opět uložíme na stejné místo.

Takovéto použití jedné proměnné je velmi jednoduché (tak jednoduché, že ho takto podrobně do řešení KSPčka nepište, není to potřeba), ale také celkem omezené. Co kdybychom si chtěli pamatovat třeba celou zadanou posloupnost čísel? Mohlo by nám stačit vyrobit si spoustu různě pojmenovaných proměnných, ale nejde to lépe? Jde.

Jednotlivé proměnné se mohou kombinovat do složitějších konstrukcí, které obecně nazýváme *datovými strukturami*. Zkusíme si ty nejzákladnější představit.

Pole

První datovou strukturou, kterou si představíme a která se na výše nastíněnou situaci náramně hodí, je *pole*. To představuje spoustu přihrádek (proměnných) naskládaných v paměti za sebou, ke kterým typicky přistupujeme přes jeden společný název pole a jejich pořadové číslo neboli index (jako `NazevPole[0], NazevPole[1], ...`).[34]

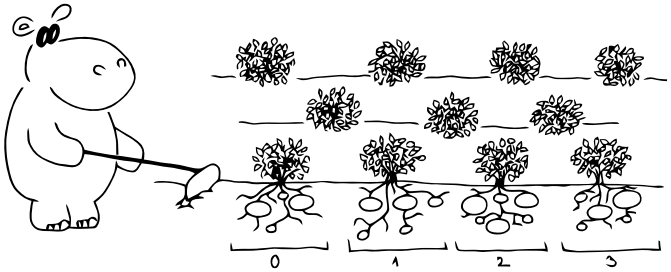
Ve většině základních jazyků je pole jen *statické*, tedy v okamžiku jeho vytváření musíme počítači říct, jak ho chceme velké. Některé vyšší jazyky ale nabízejí i pole, které se dynamicky zvětšuje, takovou konstrukci si ukážeme ve druhé části kuchařky.

Abychom nebyli omezeni jen jedním rozměrem, můžeme si klidně vyrobit pole dvourozměrné (případně obecně n-rozměrné). Dvourozměrné pole je vlastně tabulka hodnot, nazýváme ji také někdy *matice*, a může se nám hodit napří-

[34] Pozor, ve světě počítačů se velmi často indexuje od nuly, tedy první prvek má v tomto případě index 0.

klad při reprezentaci různých map (plán bludiště) nebo, jak uvidíme níže, pro reprezentaci dalších datových struktur.

U pole již má smysl přemýšlet, jak dlouho bude která operace trvat. Díky tomu, že jsou jednotlivé prvky v poli naskládané pevně za sebou, když se počítače zeptáme na obsah přihrádky `pole[42]`, přesně ví, na kterém místě v paměti se její obsah nachází, a proto nám hodnotu vrátí ihned.



Tomu budeme říkat *operace v konstantním čase* a budeme značit, že trvá čas $\mathcal{O}(1)$. Efektivitu programu totiž nepočítáme v sekundách (protože každý z nás má asi jinak rychlý počítač), ale v počtu základních operací, které musí program řádově vykonat. Více o časové složitosti si můžete přečíst v kuchařce o složitosti,[35] nejdříve však doporučujeme dočíst tuto kuchařku.

Přidání nového prvku na konec pole také zvládneme v konstantním čase. Problém je přidání nového prvku někam doprostřed (což se nám typicky stane, pokud budeme chtít udržovat hodnoty v poli seřazené a zároveň do něj vkládat nové). V takovém případě se totiž všechny prvky za vkládaným musí posunout o jednu pozici dál, aby se vkládaný prvek vešel na své místo. Taková operace tedy může pro pole délky N prvků trvat řádově až N kroků, což zapisujeme jako $\mathcal{O}(N)$ a říkáme, že je to vzhledem k N *lineární časová složitost*.

To je docela značná nevýhoda oproti struktuře, kterou si ukážeme za chvíli. Určitě ale pole nezavrhujme. Je to základní datová struktura, která nalezne použití ve spoustě programů, a jak si ve druhé části kuchařky ukážeme, můžeme ho použít třeba k rychlému hledání hodnoty metodou *binárního vyhledávání*. Nyní ale již slibovaná další datová struktura.

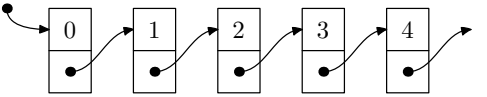
Spojový seznam a ukazatele

Pole jsme měli v paměti určené jenom tím, že počítač věděl, kde je jeho začátek a kolik místa v paměti zabírají jeho prvky. Při dotazování na konkrétní index pak podle indexu a podle velikosti prvků počítač přesně věděl, kam do paměti se má podívat, aby našel námi požadovaný prvek (to vše zvládl v konstantním

[35] `http://ksp.mff.cuni.cz/viz/kucharky/slozitost`

čase). Jednotlivé prvky si tedy vůbec nemusely pamatovat, kde se nachází jejich sousedi, protože všechny prvky seděly v paměti za sebou.

Představme si ale teď situaci, kdy by si každý prvek ještě pamatoval pozice sousedů. Pak bychom mohli mít prvky libovolně rozházené v paměti a jen by se na sebe vzájemně odkazovaly (první prvek by tvrdil, že druhý je na pozici X, druhý by tvrdil, že třetí je na pozici Y, a tak dále).

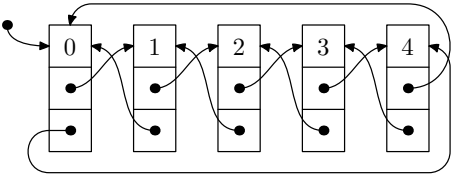


K lepšímu pochopení tohoto principu je důležité si vysvětlit, co to je *ukazatel* (nebo také *odkaz* či anglicky *pointer*). Každé místo v paměti počítače má své číselné označení, kterému říkáme *adresa*. Když si vytváříme nějakou pojmenovanou proměnnou, ta se vlastně odkazuje na určité místo v paměti a na tomto místě v paměti je její hodnota.

Co kdyby ale hodnota proměnné byla adresa nějakého jiného místa v paměti? Pak takové proměnné říkáme *pointer* a umožňuje nám vytvářet výše popsanou strukturu rozházených prvků v paměti.

Spojový seznam je tedy určený svým prvním prvkem (máme v jedné proměnné pointer na tento prvek, který se často nazývá *kořen*, protože z něj „vyrůstá“ zbytek struktury) a poté u každého dalšího prvku máme za sebou uloženou hodnotu tohoto prvku a odkaz (pointer) na další prvek. Odkazy mezi prvky mohou být i obousměrné, mohou vést dokola (poslední ukazuje na první) či mohou dokonce tvořit nějakou složitější strukturu (pak to ale již nebude čistý spojový seznam).

Pokud pointer nemá nikam ukazovat, realizuje se to odkázáním tohoto pointeru na adresu `NULL`. To skoro doslovně říká „Neukazuji nikam“.



Co nám takto vystavěná struktura umožňuje v porovnání s polem? Přístup na konkrétní prvek v ní stojí lineárně času, protože ho musíme „odkrokovat“ od prvního prvku (na který máme pointer), tedy musíme udělat až $\mathcal{O}(N)$ kroků. Pokud bychom však pointer na daný prvek už nějak měli, samozřejmě na něj můžeme přistoupit v konstantním čase.

Naopak přidávání prvků na konkrétní místo (i jejich odebírání) máme v podstatě zadarmo a spojový seznam můžeme rozšiřovat, dokud na něj máme v počítači paměť. Ve chvíli, kdy chceme přidat nový prvek za prvek, na který máme

pointer, jen šikovně přepojíme ukazatele. Pokud předtím ukazatele vedly $A \to B$, teď povedou $A \to C \to B$ (a při odebírání naopak).

Zde můžete vidět ukázku pointerů a spojových seznamů v jazyce C, kde jsou tyto věci mnohem více nízkoúrovňové (ale zato rychlejší):

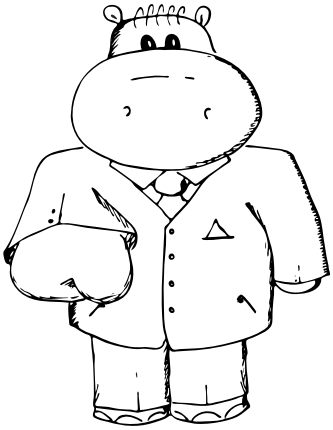
```
#include <stdio.h>
#include <stdlib.h>
// Příkazy výše načetly do programu
// standardní knihovny a funkce z˜nich.

// Struktura pro prvek obsahující dopředné
// i zpětné odkazy. Zkráceně tomuto typu
// budeme říkat "tprvek".
typedef struct prvek tprvek;
struct prvek {
    int hodnota;
    tprvek *dalsi;
    tprvek *predchozi;
};

// Vytvoří nový prvek:
tprvek *novy(int i) {
    tprvek *aktualni =
        malloc(sizeof(tprvek));
    aktualni->dalsi = NULL;
    aktualni->predchozi = NULL;
    aktualni->hodnota = i;
    return aktualni;
}

// Odstraní prvek a vrátí pointer na další
// prvek (vrácení pointeru se hodí při
// odstraňování kořene):
tprvek *odstran(tprvek *aktualni) {
    if (aktualni->predchozi != NULL)
        aktualni->predchozi->dalsi =
            aktualni->dalsi;
    if (aktualni->dalsi != NULL)
        aktualni->dalsi->predchozi =
            aktualni->predchozi;

    tprvek *pomocna = aktualni->dalsi;
    free(aktualni);
    return pomocna;
}
```

```
// Vloží a vrátí pointer na nový prvek:
tprvek *vloz_za(tprvek *aktualni, int i) {
    tprvek *pomocna = aktualni->dalsi;
    aktualni->dalsi = novy(i);

    if (pomocna != NULL)
       pomocna->predchozi = aktualni->dalsi;

    aktualni->dalsi->dalsi = pomocna;
    return aktualni->dalsi;
}

// Použití:
int main(void) {
    tprvek *koren = novy(1);
    tprvek *aktualni = vloz_za(koren, 2);

    aktualni = koren;
    while (aktualni != NULL) {
        printf("%d\n",aktualni->hodnota);
        aktualni = aktualni->dalsi;
    }

    return 0;
}
```


Zde je ukázka spojových seznamů v Pythonu, kdybychom si je podobně jako v C chtěli naprogramovat sami (Python totiž obsahuje spoustu základních struktur již hotových, podívejte se na modul jménem `collections`):

```
class Prvek:
    def __init__(self, hodnota):
        self.hodnota = hodnota
        self.dalsi = None
        self.predchozi = None

class Spojak:
    def __init__(self):
        self.koren = None

    def Vypis(self, aktualni):
        if aktualni is not None:
            print aktualni.hodnota
            self.Vypis(aktualni.dalsi)

    def VlozPo(self, prvek, zaPrvek = None):
        if zaPrvek is not None:
            prvek.dalsi = zaPrvek.dalsi
```

```
                prvek.predchozi = zaPrvek
                zaPrvek.dalsi = prvek
            if prvek.dalsi is not None:
                prvek.dalsi.predchozi = prvek
            if self.koren is None:
                self.koren = prvek

    def Odstran(self, prvek):
        if prvek.predchozi is not None:
            prvek.predchozi.dalsi = \
                prvek.dalsi
        if prvek.dalsi is not None:
            prvek.dalsi.predchozi = \
                prvek.predchozi

# Použití:
prvekA = Prvek("A")
prvekB = Prvek("B")
prvekC = Prvek("C")
prvekD = Prvek("D")

seznam = Spojak()

seznam.VlozPo(prvekB)
seznam.VlozPo(prvekD, prvekB)
seznam.VlozPo(prvekC, prvekD)
seznam.VlozPo(prvekA, prvekC)
seznam.Odstran(prvekC)

seznam.Vypis(seznam.koren)
```



Fronta a zásobník

S použitím spojových seznamů (nebo v jednodušším případě dokonce i polí) můžeme zkonstruovat dvě velmi užitečné datové struktury, frontu a zásobník.

Fronta funguje tak, jak si ji asi každý z nás představuje: ten, kdo se do fronty postaví první, ten také první přijde na řadu. Můžeme si ji také představit jako trubku, do které na jedné straně sypeme nějaké věci a na druhé je odebíráme. Anglicky je též nazývaná *FIFO* (*„First In, First Out“*).

Praktickou realizaci uděláme jednoduše spojovým seznamem. Budeme si držet dva ukazatele, jeden na začátek seznamu, druhý na konec. Když se objeví nový prvek, který do fronty budeme chtít vložit, přidáme ho na konec, zatímco při odebírání z fronty využijeme druhého ukazatele a vezmeme prvek ze začátku.

Druhou velmi podobnou datovou strukturou je *zásobník*. Jak už ale plyne z anglického názvu *LIFO* (*„Last In, First Out“*), funguje spíše jako plný šuplík: Nahoru na něj přidáváme nové prvky, a když chceme nějaký odebrat, vezmeme také zvrchu. To znamená, že první se na řadu dostane naposledy vložený prvek.

Implementace je velmi obdobná jako u fronty, jen bude ukazatel pouze jeden a bude ukazovat jenom na jeden konec spojového seznamu.

Knihovny

Tyto základní struktury už jsou často předpřipravené jako součást nějakých *knihoven* v daném jazyce. Knihovna je většinou sbírka nějakých navzájem souvisejících funkcí, které již někdo sepsal a které si můžeme do našeho programu načíst a používat. Ukázku načtení knihoven můžete vidět například ve výše zmíněném kódu v jazyce C.

Je ale velmi důležité rozumět tomu, jak knihovní funkce vnitřně fungují. Protože jedině když budeme vědět, co je jak rychlé a efektivní, budeme schopni psát rychlé programy.

Teď již víme, jak reprezentovat nejzákladnější datové struktury v počítači, ale mohlo by se nám hodit zastavit se ještě chvíli u dalších struktur. Tentokrát je už budeme studovat trochu teoretičtěji.

Stromy a grafy v informatice

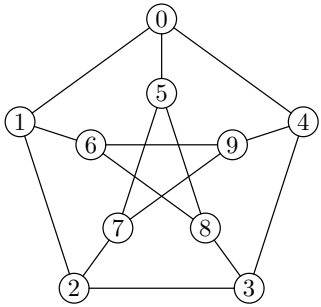
Grafy

S nějakými grafy jste se již možná potkali, ale tento pojem je bohužel docela přetěžovaný. Jedním jeho významem jsou „koláčové grafy“ a jiné další diagramy znázorňující nějaký poměr (ať už to jsou výsledky voleb, nebo poměr lidí, kteří sledovali v televizi Večerníček).

Další význam můžeme nalézt v analytické matematice, kde se potkáme s grafy průběhu nějakých funkcí. My však nemáme na mysli ani jedno z výše zmíněných, teď se budeme bavit o *kombinatorických grafech*.

Grafem tedy máme na mysli nějakou množinu objektů, říkejme jim *vrcholy*, a nějaké vztahy mezi nimi. Tyto vztahy nazýváme *hranami* a jsou vyjádřené

dvojicemi vrcholů, mezi kterými vedou. Ukázku takového grafu vidíme třeba na následujícím obrázku.



Jako praktickou ukázku grafu si můžeme například představit silniční síť nějakého státu: vrcholy budou města a hrany budou silnice, které mezi nimi vedou.

Občas se můžete setkat s pojmem *souvislý* graf. Ten znamená jen to, že mezi každými dvěma vrcholy existuje nějaká cesta. Pokud tomu tak není, je graf *nesouvislý* a dá se rozložit na několik menších grafů, které již souvislé jsou a říká se jim *komponenty souvislosti*.

Samotný graf poté můžeme doplnit tím, že si v každém vrcholu nebo na každé hraně budeme pamatovat nějakou hodnotu (například cenu nejlevnějšího benzínu ve městech a délku v kilometrech na silnicích). Pamatování si hodnot ve vrcholech je docela obvyklá technika a nemá speciální název, ale pokud budeme mít graf, který si pamatuje hodnoty na hranách, budeme o něm mluvit jako o *ohodnoceném grafu*.

Další možnou úpravou je, že každá hrana povede jen jedním směrem (jednosměrné silnice), takovým grafům říkáme *orientované* (pokud pak v orientovaném grafu chceme silnici oběma směry, prostě do něj přidáme dvě hrany, jednu v každém směru).

Poslední, co nám schází k praktickému použití grafů, je naučit se, jak je reprezentovat v počítači. Existuje několik možností (n bude značit počet vrcholů, m počet hran):

- **Seznam sousedů** – vrcholy grafu budeme mít uložené v poli a u každého vrcholu budeme mít (spojový) seznam čísel dalších vrcholů, do kterých z aktuálního vrcholu vede hrana. Zabírá místo $\mathcal{O}(n + m)$ a hodí se pro řídké grafy (tedy grafy, kde je m řádově stejné jako n).
- **Matice sousednosti** – tabulka $n \times n$, kde na souřadnicích $[i, j]$ je jednička (případně jiná hodnota, v případě ohodnoceného grafu), pokud z i do j vede hrana, a nula, pokud tam hrana není (u neorientovaných grafů je navíc matice syme-

trická – je jedno, jestli vezmeme $[i, j]$ nebo $[j, i]$). Hodí se pro husté grafy, kde $m \sim n^2$.

- **Matice incidence** – řádky reprezentují vrcholy, sloupce hrany. V každém sloupci jsou právě dvě jedničky – indexy vrcholů, mezi kterými hrana vede. Zabírá však $\mathcal{O}(mn)$ a její použití bývá dost neohrabané, takže je většinou lepší dát přednost jiné reprezentaci grafu. Je však dobré o ní vědět.

KSP

Grafy jsou velmi široké téma. Můžeme hledat jejich minimální kostry, můžeme v nich hledat nejkratší cesty či skrze ně pouštět pod tlakem vodu. Více o nich si tedy můžete přečíst v některé z našich specializovaných grafových kuchařek, které odkazujeme z našeho kuchařkového rozcestníku.[36]

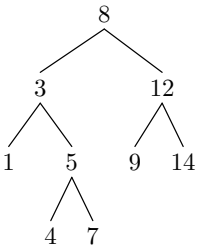
recepty

Stromy

Možná si říkáte, co má informatika u všech elektronů společného s lesnictvím? Kupodivu celkem mnoho a bez stromů bychom se v leckterém případě jen těžko obešli. Informatické stromy sice nejsou většinou tak zelené, mají ale, na rozdíl od svých dřevnatých sourozenců, mnoho jiných pěkných vlastností.

Strom je vlastně speciálním případem souvislého grafu, který neobsahuje žádnou kružnici (cyklus). To znamená, že mezi každými dvěma vrcholy stromu existuje právě jedna cesta.

Díky této vlastnosti můžeme nějaký zvolený vrchol prohlásit za *kořen* a náš strom za něj pomyslně zavěsit (tak, že strom roste od kořene směrem dolů), této operaci se říká *zakořenění*. Pak můžeme mluvit o tom, že z kořene směrem dolů (informatické stromy mají tradičně kořen nahoře) vyrůstají nějaké *podstromy*.



Pokud je strom zakořeněný, můžeme v něm mluvit o *hloubce* každého vrcholu, neboli o jeho vzdálenosti od kořene. Hloubka celého stromu je pak nejdelší ze vzdáleností od kořene k nějakému z *listů* (tak říkáme vrcholům, které již nemají žádné *syny*, tedy vrcholy, které by z nich vyrůstaly). Podle hloubky poté můžeme vrcholy stromu uspořádat do jednotlivých *hladin*.

Velmi často používáme stromy, které jsou nějak pravidelné. Příkladem jsou třeba *binární stromy*, které mají v každém vrcholu maximálně dva syny (říkáme

[36] `http://ksp.mff.cuni.cz/study/cooks/`

jim *levý a pravý podstrom*). Reprezentovat se dají buď obecně jako každý jiný strom (v každém vrcholu spojový seznam podstromů), nebo velmi pěkně i v poli.

Stačí si pomyslně doplnit binární strom na *úplný* (to je takový, který má všechny své hladiny plné) a pak ho od kořene směrem dolů po hladinách očíslovat (kořen dostane číslo nula, jeho synové čísla jedna a dva, další hladina čísla tři až šest, atd.).

Můžeme si všimnout, že pokud si v takovém očíslování vezmeme jakýkoliv vrchol s číslem (indexem) i, tak jeho synové jsou právě vrcholy s indexy $2i + 1$ a $2i + 2$. Do pole níže je zapsaný binární strom z obrázku výše.

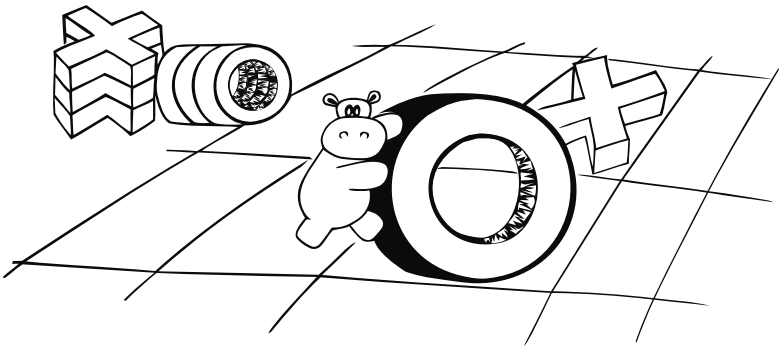
index	0	1	2	3	4	5	6	7	8	9	10
hodnota	8	3	12	1	5	9	14	–	–	4	7

Jak plyne z očíslování, pro úplný binární strom je uložení v poli efektivní a neplýtváme místem. Pokud ale strom úplný nebude, zůstanou nám v poli volná místa. Uložení v poli se tedy vyplatí jen pro stromy, které se od úplných příliš neliší.

Speciálním případem binárních stromů jsou pak ještě *binární vyhledávací stromy*. Jsou to normální binární stromy, pro něž navíc platí, že ať si vezmeme libovolný vrchol, budou hodnoty vrcholů v jeho levém podstromě menší než hodnota tohoto vrcholu, a hodnoty v jeho pravém podstromě naopak větší.

V takovém stromě pak zvládneme snadno vyhledávat. Budeme ho postupně procházet od kořene a v jednotlivých vrcholech budeme porovnávat hledanou a aktuální hodnotu a podle toho sestupovat do správného podstromu. Podobná technika je detailněji popsaná ve druhé části kuchařky, v kapitolce *Rozděl a panuj*.

Na složitější datové struktury stavějící na těchto základních (haldy, intervalové stromy, ...) se můžete podívat do některé z našich dalších kuchařek, na jejichž přehled jsme vás už odkázali o kapitolu výše.



Část druhá: Programátorské techniky

Tato část by měla sloužit jako rychlý přehled a ukázka různých technik, které se dají použít při řešení úloh z KSPčka, nebo při programování obecně.

Rekurze

Rekurze je velmi důležitá programátorská technika. V podstatě znamená definování nějaké věci (ať už je to nějaký objekt či postup výpočtu) pomocí sebe sama.

Rekurzivně může být například zadána nějaká datová struktura. Například stromy jsou pěkným příkladem rekurzivně definované datové struktury – každý vrchol stromu může mít syny, a každý z těchto synů je sám o sobě strom (tedy i osamocený vrchol bez synů je stromem).

Prakticky je to realizováno tak, že každý vrchol má svou hodnotu a pak ještě seznam ukazatelů vedoucích na další případné podstromy. S ukazateli jsme se již potkali a s jejich pomocí jsme si postavili spojový seznam. A přesně tak, spojový seznam je také ve své podstatě rekurzivní datová struktura.

Mimo rekurzivních datových struktur se ale často potkáváme i s rekurzivním postupem výpočtu nějakého programu, nejčastěji realizovaným ve formě funkce, která volá sama sebe (většinou s jinými parametry, jinak by to asi nemělo smysl), takové funkci se říká *rekurzivní funkce*.

U rekurzivních funkcí je nejdůležitější věc definovat nějakou *koncovou podmínku*, tedy podmínku, při níž už se rekurze zastaví. Jinak by se totiž mohlo stát, že by rekurze běžela donekonečna.

Přesněji rekurze by se i tak v nějakou chvíli zastavila, ale skončila by chybou, protože by jí došla paměť – každá volání funkce si totiž ukousne kus paměti (musí si pamatovat, kam se po skončení má vrátit) a pokud má rekurzivní funkce ještě nějaké lokální proměnné, musíme si někde uložit všechny lokální proměnné funkcí, z kterých jsme se doposud nevrátili.

Rekurzivní funkce a převod na nerekurzivní cyklus

Typickým příkladem rekurzivní funkce je výpočet Fibonacciho čísel. Ta jsou definována tak, že $f_0 = 1$, $f_1 = 1$ a n-té Fibonacciho číslo je součtem dvou předchozích ($f_n = f_{n-1} + f_{n-2}$). To nám dává posloupnost čísel 0, 1, 1, 2, 3, 5, 8, 13, ... pokračující donekonečna. Pokud toto přepíšeme do programového kódu, tak dostáváme následující zápisy:

V jazyce C:

```
int fib(int n) {
        if (n==0) return 0;
        else if (n==1) return 1;
        else return fib(n-1) + fib(n-2);
}
```

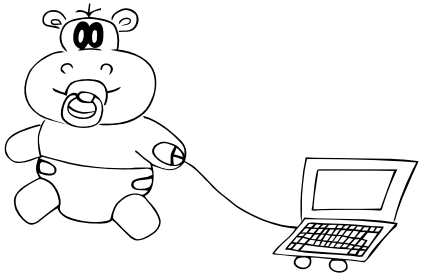
V Pythonu:

```
def fib(n):
        if n==0:
                return 0
        elif n==1:
                return 1
        else:
                return fib(n-1) + fib(n-2)
```

Jak vidíme, je přepis celkem přímočarý. Pokud by se nám však rekurze v nějakém případě nelíbila, můžeme se každé rekurze zbavit. Rekurzivní volání totiž můžeme šikovně přepsat na nějaký cyklus se *zásobníkem*.

Pak jen v cyklu odebíráme prvky ze zásobníku, dokud není prázdný, a za každé rekurzivní zavolání do zásobníku přidáme parametry, se kterými bychom naši funkci volali. Tímto postupem převedeme každou rekurzivní funkci na nerekurzivní.

Ještě doplníme poznámku, že ve většině programovacích jazyků každé volání funkce stojí nějaký čas, sice malý, ale když se volání provádí opakovaně, tak se to už nasčítá. Pro reálnou implementaci je tedy nejlepší pokusit se rekurzi převést na nerekurzivní volání, pokud to nějak rozumně jde.



Občas to jde dokonce i jednodušeji a bez zásobníku. Podívejte se na alternativní variantu výpočtu Fibonacciho čísel níže a rozmyslete si, co dělá.

V jazyce C:

```
int fib2(int n) {
        if (n==0) return 0;
        else if (n==1) return 1;

        int a = 0; int b = 1;
        for(int i = 2; i<=n; i++) {
                int c = a + b;
                a = b;
                b = c;
        }
        return b;
}
```

V Pythonu:

```
def fib2(n):
        if n==0:
                return 0
        elif n==1:
                return 1

        a = 0; b = 1
        while n>1:
                (a, b) = (b, a+b)
                n-=1
        return b
```

Jak vidíte, je i tato funkce elegantní a navíc běhá mnohem rychleji, než její rekurzivní varianta. Tato funkce běhá v $\mathcal{O}(n)$, kdežto rekurzivní varianta počítala stejné věci mnohokrát dokola (zkuste si nakreslit nějaký strom volání předchozí funkce, případně se podívat dopředu do kapitoly Předpočítané mezivýsledky).

Rekurzivní varianta tedy běžela až v čase $\mathcal{O}(2^n)$, což je pro velká n mnohem pomaleji než $\mathcal{O}(n)$ (avšak šla by celkem snadno zachránit, aby běžela také v $\mathcal{O}(n)$, zkuste si rozmyslet jak).

Backtracking

S rekurzí silně souvisí i pojem *backtracking*, česky by se snad dalo říci „metoda pokusu a omylu“. Tímto pojmem označujeme proces, kdy postupně zkoušíme všechny možnosti, jak vyřešit nějaký problém.

Metoda pokusu a omylu se tento proces nazývá proto, že pokud již nemůžeme pokračovat dál (třeba v případě, že v bludišti dojdeme do slepé uličky), vrátíme se kus zpět a zkusíme jinou (zatím nevyzkoušenou) možnost. Takto po-

stupně zkusíme každou možnost, a buď nalezneme námi hledané řešení, nebo se vrátíme až na výchozí pozici a zjistíme, že řešení neexistuje.

Backtracking bývá často realizován pomocí rekurze, ukážeme si to na příkladu hledání rozkladu zadané částky na mince o hodnotách 5 Kč a 3 Kč (všimněte si, že v takto omezeném peněžním systému nejde složit třeba částka 7 Kč). Naše funkce dostane jako parametr zbývající částku a zkusí rekurzivně provést rozklad na jednotlivé mince:

V jazyce C:

```
bool rozloz(int castka) {
        // Koncová podmínka rekurze
        if (castka == 0) return true;
        else if (castka < 0) return false;
        else if (rozloz(castka-5)) {
                printf(" 5 Kc");
                return true;
        } else if (rozloz(castka-3)) {
                printf(" 3 Kc");
                return true;
        } else return false;
}
```

V Pythonu:

```
def rozloz(castka):
        if castka == 0:
                return True
        elif castka < 0:
                return False
        elif rozloz(castka-5):
                print " 5 Kc"
                return True
        elif rozloz(castka-3):
                print " 3 Kc"
                return True
        else:
                return False
```

V každém kroku zkusíme nejdříve použít pětikorunovou minci a zavoláme se na zbylou částku, a když náš rozklad nevyjde, zkusíme v tomto kroku použít ještě tříkorunu. Takto se rozhodujeme v každém kroku rekurze a případně se vracíme z neúspěšných větví výpočtu a zkoušíme další možnosti.

Takovým postupem ale vyzkoušíme až exponenciálně mnoho možností (tedy $\mathcal{O}(2^n)$), což není moc rychlé. Proto je doporučováno se backtrackování raději vyhnout, nebo ho nějak chytře vylepšit. Je však dobré o backtrackování vědět, protože existují problémy, které efektivněji řešit neumíme.

Rozděl a panuj

Jednou ze základních technik je rozdělení složitějšího problému na menší části, které opět můžeme rozdělit na menší a tak dále, dokud se nedostaneme k problémům tak malým, že je už umíme triviálně vyřešit.

Binární vyhledávání v poli

Představte si, že máme seřazené pole n prvků a chceme zjistit, jestli se v něm nachází prvek s hodnotou k. Určitě můžeme projít celé pole v lineárním čase (tím, že budeme brát jeden prvek za druhým a kontrolovat, zda je roven hodnotě k), ale to je zbytečně pomalé a nevyužívá toho, že máme pole seřazené.

Můžeme totiž začít s velkým problémem a ten postupně zmenšovat na stále menší a menší. Nejdříve hledáme k v celém poli. Podíváme se na jeho prostřední prvek:

- Pokud je roven k, jsme hotovi.
- Je-li větší než k, víme, že se k musí nacházet nalevo od něj. Můžeme tedy hledat znovu, ale tentokrát se omezit jen na levou polovinu pole.
- Analogicky, je-li menší než k, můžeme hledat jen v pravé polovině.

Když tímto postupným dělením problémů na menší dojdeme až k poli o velikosti jednoho prvku, stačí tento prvek jenom porovnat, dál už se pole nepokoušíme rozdělovat.

Jelikož se nám každým krokem problém zmenší na polovinu, tak se maximálně po $\log n$ krocích dostaneme na pole velikosti jedna. Říkáme, že algoritmus má *logaritmickou časovou složitost*, píšeme $\mathcal{O}(\log n)$.[37]

Prakticky postup provádíme tak, že si udržujeme levý a pravý okraj aktuálně zpracovávaného úseku a postupně je k sobě přibližujeme.

Ukázka hlavní smyčky v C:

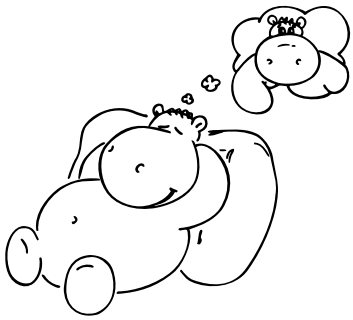
```
int pole[] = {1,2,5,7,12,16,42};
int hledane = 8;
int L = 0, R = 6;
int x;
```

[37] Pokud není řečeno jinak, znamená pro nás v informatice značka log *dvojkový logaritmus*, což je funkce opačná k funkci 2^n a roste o hodně pomaleji než funkce lineární. Pro velká n platí: $1 < \log n < n$ a například $\log 2 = 1, \log 8 = 3, \log 1024 = 10$.

```
do {
    int prostredni = (L+R)/2;
    x = pole[prostredni];
    if (x == hledane)
        printf("Pole obsahuje hledane\n");
    else if (x < hledane)
        L = prostredni + 1;
    else
        R = prostredni;
} while (L < R && x != hledane);

if (x != hledane)
    printf("Hledane neni v poli\n");
```


Ukázka v Pythonu jako funkce vracející index prvku nebo -1, pokud hledané číslo nenalezne:


```
def bin_vyhled(pole, hledane, L=0, R=None):
    if R is None:
        R = len(pole)
    while L < R:
        prostredni = (L+R)//2
        x = pole[prostredni]
        if x < hledane:
            L = prostredni + 1
        elif x > hledane:
            R = prostredni
        else:
            return prostredni
    return -1

# Zavolání:
print bin_vyhled([1,2,5,7,12,16,42], 8)
```



Další aplikace

Další typickou aplikací postupu rozděl a panuj je například třídění posloupnosti pomocí *Mergesortu*. Ten v základu funguje tak, že každou posloupnost, kterou dostane k setřídění, rozdělí na poloviny a každou z nich setřídí rekurzivním zavoláním sebe sama. Zanořování se zastaví ve chvíli, kdy třídíme posloupnost délky jedna (ta už je z podstaty setříděná). Pak jen v každém kroku ze dvou setříděných menších posloupností vyrobí jejich sléváním setříděnou posloupnost dvojnásobné délky.

KSP

Více se o metodě Rozděl a panuj můžete dozvědět ve stejnojmenné kuchařce.[38]

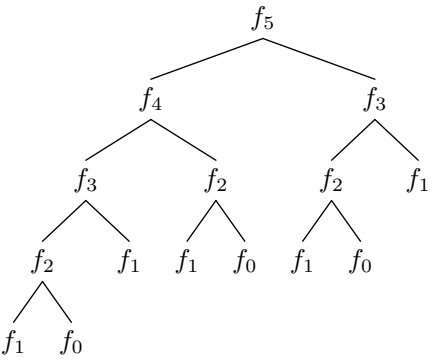
Předpočítané mezivýsledky

Motivací k této kapitole je následující motto: „Proč počítat něco vícekrát, když nám to stačí spočítat jednou a zapamatovat si to?“.

recepty

Velmi často se totiž setkáváme s tím, že něco počítáme stále dokola. Jako příklad si můžeme vzít naši rekurzivní implementaci počítání Fibonacciho čísel z kapitolky Rekurze.

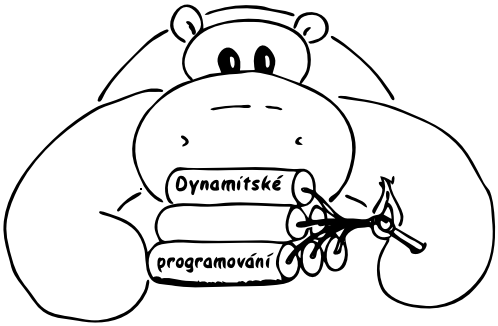
Když se podíváme na výpočet čísla `fib(5)`, vidíme, že pro něj voláme `fib(4)` a `fib(3)`, `fib(4)` volá `fib(3)` a `fib(2)`, `fib(3)` volá `fib(2)` a `fib(1)` a tak dále. Všimli jste si, kolikrát se nám tyhle výpočty opakují? Některá Fibonacciho čísla spočteme totiž zbytečně mnohokrát.



Kdybychom si je namísto opakovaného počítání někde pamatovali, mohli bychom pak odpověď na dotaz na již vypočtené číslo vytáhnout jako králíka z klobouku v konstantním čase. Zavedením jednoho globálního pole, ve kterém si tyto hodnoty pro jednotlivá n budeme pamatovat, nám sníží časovou složitost z $\mathcal{O}(2^n)$ na pěkných $\mathcal{O}(n)$. Takovému postupu se obecně říká *dynamické programování*.

[38] `http://ksp.mff.cuni.cz/viz/kucharky/rozdel-a-panuj`

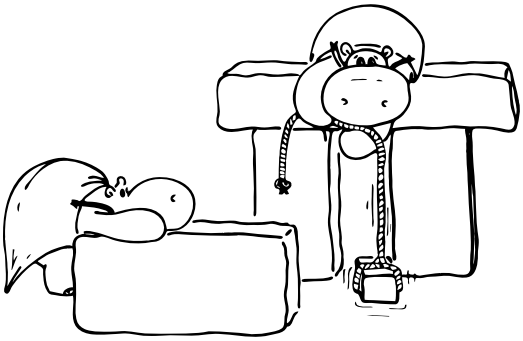
Dynamické programování



Nejprve uveďme na pravou váhu výraz „dynamické“ v názvu. Nevystihuje tak úplně podstatu této techniky a jeho historické pozadí je celkem složité, avšak dnes je tento název již tak zažitý, že se už pravděpodobně nezmění.

Slovo „dynamické“ částečně odkazuje na to, že se dynamicky (za běhu programu) postupně staví řešení jednodušších problémů, která jsou následně použita pro řešení složitějších. Jeho hlavní podstatou je tedy ukládání a opětovné použití již jednou vypočtených údajů.

Hodí se na úlohy, které se dají dělit na podúlohy, které jsou si podobné a mohou se opakovat. Výsledky takovýchto podúloh si poté ukládáme a při dotazu na stejnou podúlohu vrátíme jen uložený výsledek a výpočet již neprovádíme.



Pro další prohloubení znalostí můžete na našem webu nahlédnout do další kuchařky, tentokrát nesoucí (překvapivě) název Dynamické programování.[39]

[39] `http://ksp.mff.cuni.cz/viz/kucharky/dynamicke-programovani`

Prefixové součty

Velmi často se nám hodí si ještě před samotným výpočtem předpočítat a uložit nějaké hodnoty, které poté použijeme.

Představme si například problém nalezení souvislého úseku s největším součtem v nějaké posloupnosti kladných i záporných čísel. Že to není úplně jednoduchý příklad, si ukažme na následující posloupnosti:

$$1, -2, 4, 5, -1, -5, 2, 7$$

KSP

Máme zde dvě ryze kladné souvislé posloupnosti, každou se součtem 9 ($4, 5$ a $2, 7$). Ale přesto je výhodnější vzít i nějaké záporné hodnoty a vytvořit tak souvislou posloupnost o součtu 12 (zkuste ji nalézt).

Mohlo by nás napadnout, že prostě zkusíme vzít všechny možné začátky a všechny možné konce. To nám dává $\mathcal{O}(n^2)$ možných posloupností (máme n možných začátků a ke každému z nich řádově n možných konců), pro každou posloupnost si spočteme součet (to zvládneme v $\mathcal{O}(n)$) a budeme si pamatovat ten největší nalezený. Celý náš postup tak trvá $\mathcal{O}(n^3)$.

recepty

To není pro takhle jednoduchou úlohu zrovna ten nejpěknější čas, zkusme ho zlepšit. Ukážeme si, jak vypočítat součet libovolné posloupnosti v konstantním čase. Celý princip je vlastně až kouzelně jednoduchý, ale zároveň velmi mocný. Na začátku výpočtu si do pomocného pole P stejné délky jako posloupnost na vstupu (té říkejme S) uložíme takzvané *prefixové součty:* i-tý prefixový součet je součet prvních $i + 1$ prvků S, neboli $P[i] = S[0] + S[1] + \ldots + S[i]$.

Pro náš ukázkový případ a pro vstupní pole označené S by to dopadlo takto:

i	-1	0	1	2	3	4	5	6	7
$S[i]$		1	-2	4	5	-1	-5	2	7
$P[i]$	0	1	-1	3	8	7	2	4	11

Pole prefixových součtů umíme získat v lineárním čase – prostě jen od začátku procházíme vstupní pole, počítáme si průběžný součet a ten zapisujeme.

Součet libovolného úseku $a \ldots b$ pak získáme v konstantním čase jako prefixový součet od začátku do indexu b minus prefixový součet od začátku do indexu a. Zapsáno programově to pak je:

```
soucet = P[b] - P[a-1];
```

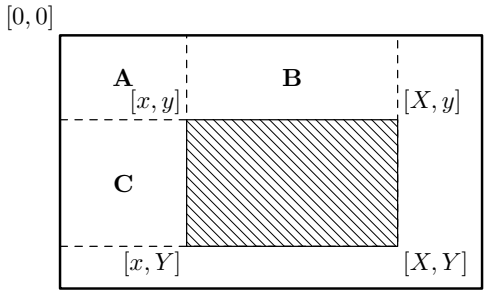
To nám umožňuje snížit čas potřebný na řešení této úlohy na $\mathcal{O}(n^2)$. To je už lepší čas, prozradíme však, že tuto úlohu lze řešit dokonce v lineárním čase, ale to je již nad rámec této kuchařky.

Dvourozměrné prefixové součty

Prefixové součty se dají zobecnit i do více rozměrů, ale princip je vždy stejný. Například dvourozměrné prefixové součty u matice fungují tak, že si předpočítáme součty podmatic začínajících levým vrchním políčkem a končící na indexu $[x, y]$.

Z toho je vidět, že prefixový součet zpravidla obsadí stejně velký prostor jako původní data, v tomto případě tedy budeme mít matici hodnot prefixových součtů končících na zadaných souřadnicích. Jak ale získat součet nějaké podmatice, která se nachází někde „uprostřed“ naší matice?

Použijeme stejný princip jako u jednorozměrného případu: Přičteme větší část, kterou chceme započítat, a odečteme od ní části, které započítat nechceme. Pro případ podmatice začínající vlevo nahoře na pozici $[x, y]$ a končící napravo dole na $[X, Y]$ to ilustruje následující obrázek:



Nejdříve přičteme celý prefixový součet končící na pozici $[X, Y]$. Tím jsme ale započítali i části A, B a C z obrázku, které započítat nechceme. Tak odečteme prefixové součty končící na indexech $[X, y]$ a $[x, Y]$. Ale pozor, teď jsme odečetli jednou $A + B$ a jednou $A + C$, tedy část A (prefixový součet končící na pozici $[x, y]$) jsme odečetli dvakrát, musíme ji proto ještě jednou přičíst.

Celý vzorec tedy vypadá takto:

```
soucet = P[X,Y] - P[X,y] - P[x,Y] + P[x,y];
```

Tento princip přičítání a odečítání se dá zobecnit i pro libovolné vyšší rozměry, ale chce to již trošku představivosti, co se má přičíst a kolikrát. Říká se tomu také *princip inkluze a exkluze* a najde použití nejen u vícerozměrných prefixových součtů.

Vyvážení délky předvýpočtu a hlavního výpočtu

Správně vyvážit, kolik času můžeme věnovat na předvýpočet a kolik času na hlavní výpočet, je velice důležitá věc a spousta i zkušenějších řešitelů v tom občas chybuje. Přitom to při troše počítání není vůbec nic složitého.

Jako první je potřeba vědět, kolikrát nám předvýpočet během běhu programu pomůže. Předvýpočtem si totiž vybudujeme za nějaký čas určitou datovou strukturu, pomocí které pak dokážeme rychle odpovídat na zadané dotazy.

Označme si počet takovýchto dotazů, které program za běhu dostane, jako Q. Buď to může být hodnota přímo ze zadání typu „Zkonstruujte datovou strukturu pro n hodnot, která zvládne rychle odpovídat na dotazy daného typu, a očekávejte řádově m dotazů“, nebo se může jednat o nějaký interní dotaz

v rámci běhu programu (příklad interního dotazu je třeba výše uvedený algoritmus na hledání souvislé podposloupnosti s co největším součtem, který se za běhu ptal na součty nějakých úseků).

Dále si označme jako $\mathcal{O}_p$ čas, který nám zabere předvýpočet a jako $\mathcal{O}_q$ čas, který nám ušetří každý předvypočítaný dotaz. Celkový čas, který ušetříme, je pak vlastně $Q \cdot \mathcal{O}_q$. Pokud je tento čas řádově větší než $\mathcal{O}_p$, pak má předvýpočet smysl.

Naopak nemá smysl trávit předvýpočtem řádově více času, než by trval samotný výpočet bez použití předpočítaných hodnot.

Jako příklad uvažujme problém o velikosti n, u kterého máme tři možnosti, které můžeme zvolit. Můžeme buď předvýpočet úplně vynechat a na každý dotaz odpovídat v čase $\mathcal{O}(n)$, nebo provést předvýpočet v čase $\mathcal{O}(n \log n)$ a poté odpovídat na každý dotaz v čase $\mathcal{O}(\log n)$, nebo provést předvýpočet v čase $\mathcal{O}(n^2)$ a pak odpovídat v čase $\mathcal{O}(1)$ na dotaz.

Kdy se nám co hodí?

- Pokud bychom dostali jen jeden dotaz, nemá smysl si cokoliv předpočítávat a odpovíme jednou v čase $\mathcal{O}(n)$.
- Pokud bude dotazů řádově n, má smysl použít první předvýpočet. Pak budeme mít čas na předvýpočet i na samotný výpočet $\mathcal{O}(n \log n)$, což je optimum.
- Naopak pokud by dotazů bylo řádově n^2 nebo víc, tak se nám již první předvýpočet nevyplatí, dostali bychom se totiž na čas $\mathcal{O}(n^2 \log n)$. Zde se hodí použít druhý, delší předvýpočet a pak se dostat na celkovou časovou složitost $\mathcal{O}(n^2 + n^2 \cdot 1) = \mathcal{O}(n^2)$.

Hladové algoritmy

Věřte nebo ne, ale i počítač se někdy cítí hladový. Po namáhavé práci mu můžeme dopřát to potěšení, aby si ukousl co největší kus dat. A ukážeme, že někdy je to i ku prospěchu. Řeč bude o *hladových algoritmech*.

Takovými algoritmy rozumíme ty, které hledají řešení celé úlohy po jednotlivých krocích a splňují následující dvě podmínky:

- V každém kroku zvolí lokálně nejlepší řešení.
- Provedené rozhodnutí již nikdy neodvolává (tedy nebacktrackuje).

Lokálně nejlepší řešení je takové, které v aktuálním kroku vybere tu možnost, která nám na tomto místě nejvíce pomůže (bez jakéhokoliv ohledu na globální stav). Může to být třeba nejvyšší hodnota, nebo nejkratší cesta v grafu.

Pokud ale od hladového algoritmu chceme, aby nám našel globálně nejlepší řešení, musí naše úloha splnit předpoklad, že si výběrem lokálně nejlepšího řešení nezhoršíme to globální. Tento předpoklad se nedá formulovat obecně a je nutné se nad ním zamyslet zvlášť u každé úlohy.

Příklady hladových algoritmů

První hladovou úlohou bude (jak jinak) automat na jídlo vracející mince. Automat by měl vracet peníze nazpět tak, aby vrátil daný obnos v co možná nejmenším počtu mincí. Pro náš měnový systém (máme mince hodnot 1, 2, 5, 10, 20 a 50 Kč) lze tuto úlohu řešit hladovým algoritmem – v každém kroku algoritmu vrátíme tu největší minci, kterou můžeme (tedy pro vrácení 42 Kč to bude $42 = 20 + 20 + 2$ Kč).

Měnové systémy většiny států jsou postavené tak, aby fungovaly takto pěkně, neplatí to ale obecně. Zkusme si vrátit 42 Kč, pokud bychom měli jen mince hodnoty 20, 10 a 4 Kč. Správným řešením je $42 = 20 + 10 + 4 + 4 + 4$ Kč, hladový algoritmus by ale zkusil vrátit $42 = 20 + 20 + \ldots$ a tady by selhal.

Dále se velmi často dají hladovým algoritmem řešit nějaké úlohy přidávání nebo odebírání skupin prvků. Typickým příkladem je třeba rozvržení naplánovaných přednášek do učeben. Seřadíme si začátky přednášek podle času a postupně bereme jednu za druhou a umísťujeme je do volných učeben s nejnižším číslem.

Tím jsme si určitě nic nerozbili, protože v nějaké učebně přednáška být musí. Určitě budeme potřebovat tolik učeben, kolik je maximálně přednášek v jeden čas, a díky tomu si umístěním přednášky do nějaké učebny nezablokujeme místo pro jinou přednášku, jelikož nám vždy zbude dostatek volných učeben.

Kdybychom ale naopak měli pevně zadaný počet učeben a chtěli jsme do nich umístit co možná nejvíce přednášek, tak se již nejedná o úlohu řešitelnou hladovým algoritmem, v takovém případě je potřeba zvolit nějaký chytřejší postup.

Závěr

Doufám, že jste si z tohoto rozsáhlého textu odnesli nějaké nové znalosti a poznatky, které vám pomohou nejen v řešení KSP.

Pokud jste začínajícími řešiteli, zkuste s pomocí kuchařky vyřešit několik lehčích úloh a jejich řešení poslat – nově nabyté znalosti je totiž nejlepší co nejdříve protrénovat. Nic si nedělejte z toho, pokud napoprvé nevyřešíte všechno, s postupným zkoušením se budou vaše znalosti jen zlepšovat. Zkušenější řešitelé možná v kuchařce nalezli nějaké ujasnění pojmů, či si některé techniky osvěžili.

A pokud tento text považujete za dobrý, budeme jen rádi, pokud ho doporučíte svým kamarádům a spolužákům, kteří chtějí s programováním začít.

Úvodním kurzem vaření podle kuchařky vás provedl

Jirka Setnička

Kuchařka druhé série – minimální kostra

Představme si následující problém: chceme určit silnice, které se budou v zimě udržovat sjízdné, a to tak, abychom celkově udržovali co nejméně kilometrů silnic, a přesto žádné město od ostatních neodřízli.

Města a silnice si můžeme představit jako graf, o kterém nyní budeme předpokládat, že je souvislý. Kdyby nebyl, náš problém nijak vyřešit nelze. Výsledný podgraf/seznam silnic, který řeší náš problém se sněhem, nazývají matematici *minimální kostra grafu*.

Pokud vůbec netušíte, co je to graf, přečtěte si úvodní grafovou kuchařku na našem webu.[40]

Co se v souvislém grafu přesně myslí pod pojmem *kostra*? Nazveme jí libovolný podgraf, který obsahuje všechny vrcholy a zároveň je stromem. Definici stromu a jejich vlastnostem se blíže věnuje dříve zmíněná grafová kuchařka; zde řekněme, že jsou to přesně ty grafy, které jsou souvislé (z každého vrcholu „dojedeme“ do každého jiného) a bez kružnice (takže nemáme v silniční síti žádné přebytečné cesty).

Pokud každou hranu grafu ohodnotíme nějakou *vahou,* což v našem případě bude vždy kladné číslo, dostaneme *ohodnocený graf.* V takových grafech pak obvykle hledáme mezi všemi kostrami *kostru minimální,* což je taková, pro kterou je součet vah jejích hran nejmenší možný.

Graf může mít více minimálních koster – například jestliže jsou všechny váhy hran jedničky, všechny kostry mají stejnou váhu $n - 1$ (kde n je počet vrcholů grafu), a tedy jsou všechny minimální.

Pro vyřešení problému hledání minimální kostry se nám bude hodit datová struktura *Disjoint-Find-Union* (DFU). Ta umí pro dané disjunktní množiny (disjunktní znamená, že každé 2 množiny mají prázdný průnik neboli žádné společné prvky) rychle rozhodnout, jestli dva prvky patří do stejné množiny, a provádět operaci sjednocení dvou množin.

Algoritmus

Algoritmus na hledání minimální kostry, který si předvedeme, je typickou ukázkou tzv. hladového algoritmu. Nejprve setřídíme hrany vzestupně podle jejich váhy. Kostru budeme postupně vytvářet přidáváním hran od té s nejmenší vahou tak, že hranu do kostry přidáme právě tehdy, pokud spojuje dvě (prozatím) různé komponenty souvislosti vytvořeného podgrafu. Jinak řečeno, hranu do vytvářené kostry přidáme, pokud v ní zatím neexistuje cesta mezi vrcholy, které zkoumaná hrana spojuje.

Je zřejmé, že tímto postupem získáme kostru, tj. acyklický souvislý podgraf grafu (pokud vstupní graf je souvislý, což mlčky předpokládáme). Než si ukážeme, že nalezená kostra je opravdu minimální, podívejme se na časovou složitost

[40] `http://ksp.mff.cuni.cz/viz/kucharky/grafy`

našeho algoritmu: pokud vstupní graf má N vrcholů a M hran, tak úvodní setřídění hran vyžaduje čas $\mathcal{O}(M \log M)$ (použijeme některý z rychlých třídicích algoritmů popsaných v kuchařce o třídění) a poté se pokusíme přidat každou z M hran.

V druhé části kuchařky si ukážeme datovou strukturu, s jejíž pomocí bude M testů toho, zda mezi dvěma vrcholy vede hrana, trvat nejvýše $\mathcal{O}(M \log N)$. Celková časová složitost našeho algoritmu je tedy $\mathcal{O}(M \log N)$ (všimněte si, že $\log M \leq \log N^2 = 2 \log N$). Paměťová složitost je lineární vzhledem k počtu hran, tj. $\mathcal{O}(M)$.

Důkaz správnosti

Zbývá dokázat, že nalezená kostra vstupního grafu je minimální. Bez újmy na obecnosti můžeme předpokládat, že váhy všech hran grafu jsou navzájem různé: pokud tomu tak není již na začátku, přičteme k některým z hran, jejichž váhy jsou duplicitní, velmi malá kladná celá čísla tak, aby pořadí hran nalezené naším třídicím algoritmem zůstalo zachováno. Tím se kostra nalezená hladovým algoritmem nezmění, a pokud bude tato kostra minimální s modifikovanými váhami, bude minimální i pro původní zadání.

Označme si nyní T_{alg} kostru nalezenou hladovým algoritmem a T_{min} nějakou minimální kostru. Co by se stalo, kdyby byly různé? Víme, že všechny kostry mají stejný počet hran, takže musí existovat alespoň jedna hrana e, která je v T_{alg}, ale není v T_{min}. Ze všech takových hran si vyberme tu, která má nejmenší váhu, tedy kterou algoritmus přidal jako první. Když se podíváme na stav algoritmu těsně před přidáním e, vidíme, že sestrojil nějakou částečnou kostru F, která je ještě součástí jak T_{min}, tak T_{alg}.

Přidejme nyní hranu e ke kostře T_{min}. Tím vznikl podgraf vstupního grafu, který zjevně obsahuje nějakou kružnici C – už před přidáním hrany e totiž T_{min} byla souvislá. Protože kostra T_{alg} neobsahuje žádnou kružnici, na kružnici C musí být alespoň jedna hrana e', která není v T_{alg}.

Všimněme si, že hranu e' nemohl algoritmus zpracovat před hranou e: hrana e' neleží v T_{min} na žádném cyklu, takže tím spíš netvoří cyklus v F, a kdyby ji algoritmus zpracoval, musel by ji přidat do F, což, jak víme, neučinil. Z toho plyne, že váha hrany e' je větší než váha hrany e. Když nyní z kostry T_{min} odebereme hranu e' a přidáme místo ní hranu e, musíme opět dostat souvislý podgraf (e a e' přeci ležely na společné kružnici), tudíž kostru vstupního grafu. Jenže tato kostra má celkově menší váhu než minimální kostra T_{min}, což není možné. Tím jsme došli ke sporu, a proto T_{min} a T_{alg} nemohou být různé.

Cvičení

- V důkazu jsme předpokládali, že váhy hran jsou různé.Není potřeba i v samotném algoritmu přičítat velmi malá čísla k hranám se stejnou vahou?
- Dokažte, že pokud jsou váhy hran různé, minimální kostra je určena jednoznačně.

Další algoritmy

Kromě tohoto algoritmu (říká se mu Kruskalův) můžeme minimální kostru hledat i jinými způsoby. Jejich bližší popis naleznete ve skriptíčkách z algoritmů a datových struktur,[41] my zde jen nastíníme myšlenku:

- *Jarníkův algoritmus* nechává strom rozrůstat se z jednoho vrcholu. Počáteční vrchol zvolí libovolně a pak vždy vybírá nejlehčí hranu vedoucí z už sestrojené části kostry do zbytku grafu.
- *Borůvkův algoritmus* postupně spojuje stromy. Začne triviálními jednovrcholovými stromy bez hran. Pak si každý strom vybere nejlehčí hranu, která z něj vede ven, a všechny tyto hrany přidá do kostry. Tím se stromy propojí do větších stromů a to se opakuje, dokud nezbude jediný strom.

KSP

Disjoint-Find-Union

recepty

Datová struktura *DFU* slouží k udržování rozkladu množiny na několik disjunktních podmnožin (čili takových, že žádné dvě nemají společný prvek). To znamená, že pomocí této struktury můžeme pro každé dva z uložených prvků říci, zda patří či nepatří do stejné podmnožiny rozkladu.

V algoritmu hledání minimální kostry budou prvky v DFU vrcholy zadaného grafu a budou náležet do stejné podmnožiny rozkladu, pokud mezi nimi v již vytvořené části kostry existuje cesta. Jinými slovy podmnožiny v DFU budou odpovídat komponentám souvislosti vytvářené kostry.

S reprezentovaným rozkladem umožňuje datová struktura DFU provádět následující dvě operace:

- **find:** Test, zda dva prvky leží ve stejné podmnožině rozkladu. Tato operace bude v případě našeho algoritmu odpovídat testu, zda dva vrcholy leží ve stejné komponentě souvislosti.
- **union:** Sloučení dvou podmnožin do jedné. Tuto operaci v našem algoritmu na hledání kostry provedeme vždy, když do vytvářené kostry přidáme hranu (tehdy spojíme dvě různé komponenty souvislosti dohromady).

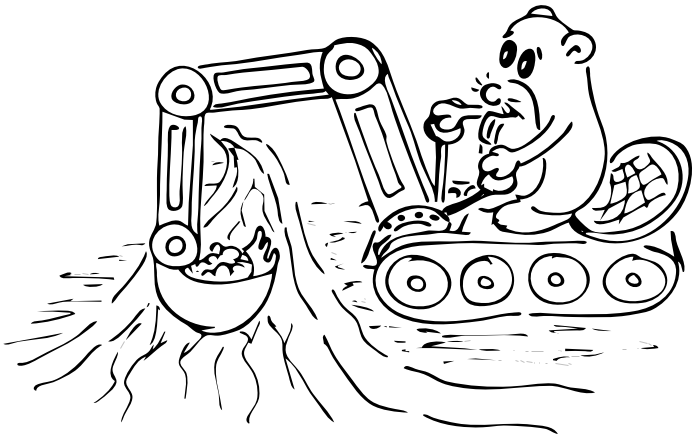
Povězme si nejprve, jak budeme jednotlivé podmnožiny reprezentovat. Prvky obsažené v jedné podmnožině budou tvořit zakořeněný strom. V tomto stromě však povedou ukazatele (trochu nezvykle) od listů ke kořeni. Operaci *find* lze pak jednoduše implementovat tak, že pro oba zadané prvky nejprve nalezneme kořeny jejich stromů. Jsou-li tyto kořeny stejné, jsou prvky ve stejném stromě, a tedy i ve stejné podmnožině rozkladu. Naopak, jsou-li různé, jsou zadané prvky v různých stromech, a tedy jsou i v různých podmnožinách reprezentovaného rozkladu. Operaci *union* provedeme tak, že mezi kořeny stromů reprezentujících slučované podmnožiny přidáme ukazatel a tím tyto dva stromy spojíme dohromady.

[41] `http://mj.ucw.cz/vyuka/ads/`

Implementace dvou výše popsaných operací, jak jsme ji právě popsali, následuje. Pro jednoduchost množina, jejíž rozklad reprezentujeme, bude množina čísel od 1 do N. Rodiče jednotlivých vrcholů stromu si pak pamatujeme v poli *parent*, kde 0 znamená, že prvek rodiče nemá, tj. že je kořenem svého stromu. Funkce *root*(v) vrátí kořen stromu, který obsahuje prvek v.

```
class DFU:
    def __init__(self, n):
        self.parent = [0] * n

    def root(self, v):
        if self.parent[v] == 0:
            return v
        else:
            return self.root(self.parent[v])

    def find(self, v, w):
        return (self.root(v) == self.root(w))

    def union(self, v, w):
        v = self.root(v)
        w = self.root(w)
        if v != w:
            self.parent[v] = w
```


S právě předvedenou implementací operací *find* a *union* by se ale mohlo stát, že stromy odpovídající podmnožinám budou vypadat jako „hadi“, a pokud budou obsahovat N prvků, na nalezení kořene bude potřeba čas $\mathcal{O}(N)$.



Ke zrychlení práce DFU se používají dvě jednoduchá vylepšení:

- **union by rank:** Každý prvek má přiřazen *rank*. Na začátku jsou ranky všech prvků rovny nule. Při provádění operace *union* připojíme strom s kořenem menšího ranku ke kořeni stromu s větším rankem. Ranky kořenů stromů se v tomto případě nemění. Pokud kořeny obou stromů mají stejný rank, připojíme je libovolně, ale rank kořenu výsledného stromu zvětšíme o jedna.
- **path compression:** Ve funkci *root*(v) přepojíme všechny prvky na cestě od prvku v ke kořeni rovnou na kořen, tj. změníme jejich rodiče na kořen daného stromu.

Než si obě metody blíže rozebereme, podívejme se, jak se změní implementace funkcí *root* a *union*:


```
class DFU:
    def __init__(self, n):
        self.parent = [0] * n
        self.rank = [0] * n

    # Změna: path compression
    def root(self, v):
        if self.parent[v] == 0:
            return v
        else:
            self.parent[v] = \
              self.root(self.parent[v])
            return self.parent[v]

    # Stejná implementace
    def find(self, v, w):
        return (self.root(v) == self.root(w))

    # Změna: union by rank
    def union(self, v, w):
        v = self.root(v)
        w = self.root(w)
        if v == w:
            return

        if self.rank[v] == self.rank[w]:
            self.parent[v] = w
            self.rank[w] = self.rank[w]+1

        elif self.rank[v] < self.rank[w]:
            self.parent[v] = w

        else:
            self.parent[w] = v
```

Zaměřme se nyní blíže na metodu *union by rank.* Nejprve učiníme následující pozorování: pokud je prvek v s rankem r kořenem stromu v datové struktuře DFU, pak tento strom obsahuje alespoň 2^r prvků.

Naše pozorování dokážeme indukcí podle r. Pro $r = 0$ tvrzení zřejmě platí. Nechť tedy $r > 0$. V okamžiku, kdy se rank prvku v mění z $r-1$ na r, slučujeme dva stromy, jejichž kořeny mají rank $r-1$. Každý z těchto dvou stromů má dle indukčního předpokladu alespoň 2^{r-1} prvků, a tedy výsledný strom má alespoň 2^r prvků, jak jsme požadovali.

Z našeho pozorování ihned plyne, že rank každého prvku je nejvýše $\log_2 N$ a prvků s rankem r je nejvýše $N/2^r$ (všimněme si, že rank prvku v DFU se nemění po okamžiku, kdy daný prvek přestane být kořen nějakého stromu).

Když tedy provádíme jen *union by rank,* je hloubka každého stromu v DFU rovna ranku jeho kořene, protože rank kořene se mění právě tehdy, když zvětšujeme hloubku stromu o jedna. A protože rank každého prvku je nanejvýš $\log_2 N$, hloubka každého stromu v DFU je také nanejvýš $\log_2 N$. Potom ale procedura *root* spotřebuje čas nejvýše $\mathcal{O}(\log N)$, a tedy operace *find* a *union* stihneme v čase $\mathcal{O}(\log N)$.

Amortizovaná časová složitost

Abychom mohli pokračovat dále, musíme si vysvětlit, co je *amortizovaná* časová složitost. Řekneme, že nějaká operace pracuje v amortizovaném čase $\mathcal{O}(t)$, pakliže provedení libovolných k takových operací trvá nejvýše $\mathcal{O}(kt)$. Přitom provedení kterékoliv konkrétní z nich může vyžadovat čas větší. Tento větší čas je pak v součtu kompenzován kratším časem, který spotřebovaly některé předchozí operace.

Nejdříve si předveďme tento pojem na jednoduchém příkladě. Řekněme, že máme číslo zapsané ve dvojkové soustavě. Přičíst k tomuto číslu jedničku jistě netrvá konstantní čas, neboť záleží na tom, kolik jedniček se vyskytuje na konci zadaného čísla.

Pokud se nám ale povede ukázat, že N přičtení jedničky k číslu, které je na počátku nula, zabere čas $\mathcal{O}(N)$, pak můžeme říci, že každé takové přičtení trvalo amortizovaně $\mathcal{O}(1)$.

Jak tedy ukážeme, že N přičtení jedničky k číslu zabere čas $\mathcal{O}(N)$? Použijeme k tomu „penízkovou metodu“. Každá operace nás bude stát jeden penízek, a pokud jich na N operací použijeme jen $\mathcal{O}(N)$, bude tvrzení dokázáno.

Každé jedničce, kterou chceme přičíst, dáme dva penízky. V průběhu celého přičítání bude platit, že každá jednička ve dvojkovém zápisu čísla má jeden penízek (když začneme jedničky přičítat k nule, tuto podmínku splníme).

Přičítání bude probíhat tak, že přičítaná jednička se „podívá“ na nejnižší bit (tj. ve dvojkovém zápise na poslední cifru) zadaného čísla (to ji stojí jeden penízek). Pokud je to nula, změní ji na jedničku a dá jí svůj zbylý penízek. Pokud to je jednička, vezme si přičítaná jednička její penízek (čili už má zase dva), změní zkoumanou jedničku na nulu a pokračuje u dalšího bitu, atd.

Takto splníme podmínku, že každá jednička v dvojkovém zápisu čísla má jeden penízek. Tedy N přičítání nás stojí $2N$ penízků. Protože počet penízků utracených během jedné operace je úměrný spotřebovanému času, vidíme, že všech N přičtení proběhne v čase $\mathcal{O}(N)$. Není těžké si uvědomit, že přičtení některých jedniček může trvat až $\mathcal{O}(\log N)$, ale amortizovaná časová složitost přičtení jedné jedničky je konstantní.

Dokončení analýzy DFU

Pokud bychom prováděli pouze *path compression* a nikoliv *union by rank*, dalo by se dokázat, že každá z operací *find* a *union* vyžaduje amortizovaně čas $\mathcal{O}(\log N)$, kde N je počet prvků. Toto tvrzení nebudeme dokazovat, protože tím bychom si nijak oproti samotnému *union by rank* nepomohli. Proč tedy vlastně hovoříme o obou vylepšeních? Inu proto, že při použití obou metod současně dosáhneme mnohem lepšího amortizovaného času $\mathcal{O}(\alpha(N))$ na jednu operaci *find* nebo *union*, kde $\alpha(N)$ je inverzní Ackermannova funkce. Její definici můžete nalézt na konci kuchařky, zde jen poznamenejme, že hodnota inverzní Ackermannovy funkce $\alpha(N)$ je pro všechny praktické hodnoty N nejvýše čtyři. Čili dosáhneme v podstatě amortizovaně konstantní časovou složitost na jednu (libovolnou) operaci DFU.


Dokázat výše zmíněný odhad časové složitosti funkcí $\alpha(N)$ je docela těžké, my si zde předvedeme poněkud horší, ale technicky výrazně jednodušší časový odhad $\mathcal{O}((N+L)\log^* N)$, kde L je počet provedených operací *find* nebo *union* a $\log^* N$ je tzv. iterovaný logaritmus, jehož definice následuje. Nejprve si definujeme funkci $2 \uparrow k$ rekurzivním předpisem:

$$2 \uparrow 0 = 1, \quad 2 \uparrow k = 2^{2 \uparrow (k-1)}.$$

Máme tedy $2 \uparrow 1 = 2$, $2 \uparrow 2 = 2^2 = 4$, $2 \uparrow 3 = 2^4 = 16$, $2 \uparrow 4 = 2^{16} = 65536$, $2 \uparrow 5 = 2^{65536}$, atd. A konečně, iterovaný logaritmus $\log^* N$ čísla N je nejmenší přirozené číslo k takové, že $N \leq 2 \uparrow k$. Jiná (ale ekvivalentní) definice iterovaného logaritmu je ta, že $\log^* N$ je nejmenší počet, kolikrát musíme číslo N opakovaně zlogaritmovat, než dostaneme hodnotu menší nebo rovnu jedné.

Zbývá provést slíbenou analýzu struktury DFU při současném použití obou metod *union by rank* a *path compression*. Prvky si rozdělíme do skupin podle jejich ranku: k-tá skupina prvků bude tvořena těmi prvky, jejichž rank je mezi

$(2 \uparrow (k-1)) + 1$ a $2 \uparrow k$. Např. třetí skupina obsahuje ty prvky, jejichž rank je mezi 5 a 16. Prvky jsou tedy rozděleny do $1 + \log^* \log N = \mathcal{O}(\log^* N)$ skupin. Odhadněme shora počet prvků v k-té skupině:

$$\frac{N}{2^{(2\uparrow(k-1))+1}} + \cdots + \frac{N}{2^{2\uparrow k}} = \frac{N}{2^{2\uparrow(k-1)}} \cdot \left(\sum_{i=1}^{2\uparrow k - 2\uparrow(k-1)} \frac{1}{2^i} \right) \leq$$

$$\leq \frac{N}{2^{2\uparrow(k-1)}} \cdot 1 = \frac{N}{2 \uparrow k}.$$

Teď můžeme provést časovou analýzu funkce *root*(v). Čas, který spotřebuje funkce *root*(v), je přímo úměrný délce cesty od prvku v ke kořeni stromu. Tato cesta je pak následně rozpojena a všechny prvky na ní jsou přepojeny přímo na kořen stromu. Rozdělíme rozpojené hrany této cesty na ty, které „naúčtujeme“ tomuto volání funkce *root*(v), a ty, které zahrneme do faktoru $\mathcal{O}(N \log^* N)$ v dokazovaném časovém odhadu. Do volání funkce *root*(v) započítáme ty hrany cesty, které spojují dva prvky, které jsou v různých skupinách. Takových hran je zřejmě nejvýše $\mathcal{O}(\log^* N)$ (všimněte si, že ranky prvků na cestě z listu do kořene tvoří rostoucí posloupnost).

Uvažme prvek v v k-té skupině, který již není kořenem stromu. Při každém přepojení rank rodiče prvku v vzroste. Tedy po $2 \uparrow k$ přepojeních je rodič prvku v v $(k+1)$-ní nebo vyšší skupině. Pokud v je prvek v k-té skupině, pak hrana z něj na cestě do kořene nebude účtována volání funkce *root*(v) nejvýše $(2 \uparrow k)$-krát. Protože k-tá skupina obsahuje nejvýše $N/(2 \uparrow k)$ prvků, je počet takových hran pro všechny prvky této skupiny nejvýše N. A protože počet skupin je nejvýše $\mathcal{O}(\log^* N)$, je celkový počet hran, které nejsou započítány voláním funkce *root*(v), nejvýše $\mathcal{O}(N \log^* N)$. Protože funkce *root*(v) je volána $2L$-krát, plyne časový odhad $\mathcal{O}((N+L) \log^* N)$ z právě dokázaných tvrzení.

Inverzní Ackermannova funkce $\alpha(N)$

Ackermannovu funkci lze definovat následující konstrukcí:

$$A_0(i) = i + 1, \quad A_{k+1}(i) = A_k^i(i) \text{ pro } k \geq 0,$$

kde výraz A_k^i zastupuje složení i funkcí A_k, např. $A_1(3) = A_0(A_0(A_0(3)))$. Platí tedy následující rovnosti:

$$A_0(i) = i + 1, \quad A_1(i) = 2i, \quad A_2(i) = 2^i \cdot i.$$

Ackermannova funkce s jedním parametrem $A(k)$ je pak rovna hodnotě $A_k(2)$, takže $A(2) = A_2(2) = 8$, $A(3) = A_3(2) = 2^{11}$, $A(4) = A_4(2) \approx 2 \uparrow 2048$ atd... Hodnota inverzní Ackermannovy funkce $\alpha(N)$ je tedy nejmenší přirozené číslo k takové, že $N \leq A(k) = A_k(2)$. Jak je vidět, ve všech reálných aplikacích platí, že $\alpha(N) \leq 4$.

Dan Kráľ, Martin Mareš a Milan Straka

Kuchařka třetí série – rozděl a panuj

Dnešní díl programátorské kuchařky se bude zabývat algoritmy založenými na metodě *Rozděl a panuj*. Slušelo by se začít tím, jaká je myšlenka této metody: Často se setkáme s úlohami, které lze snadno rozdělit na nějaké menší úlohy a z jejich výsledků zase snadno složit výsledek původní velké úlohy. Přitom menší úlohy můžeme řešit opět týmž algoritmem (zavoláme si ho rekurzivně), leda by již byly tak maličké, že dokážeme odpovědět triviálně bez jakéhokoliv počítání. Zkrátka jak říkali staří římští císařové: Divide et impera. Uveďme si pro začátek jeden staronový příklad:

Quicksort

QuickSort (alias QS) je algoritmus pro třídění posloupnosti prvků. Už jste si o něm mohli přečíst v kuchařce o třídění. Tentokrát se na něj podíváme trochu podrobněji a navíc nám poslouží jako ingredience pro další algoritmy.

QS v každém svém kroku zvolí nějaký prvek (budeme mu říkat *pivot*) a přerovná prvky v posloupnosti tak, aby napravo od pivota byly pouze prvky větší než pivot a nalevo pouze menší. Pokud se vyskytnou prvky rovné pivotu, můžeme si dle libosti vybrat jak levou, tak pravou stranu posloupnosti, funkčnost algoritmu to nijak neovlivní. Tento postup pak rekurzivně zopakujeme zvlášť pro prvky nalevo a zvlášť pro prvky napravo od pivota, a tak získáme setříděnou posloupnost.

Implementací QS je mnoho a mimo jiné se liší způsobem volby pivota. My si předvedeme jinou, než jsme ukazovali v třídící kuchařce (hlavně proto, že se nám z ní pak budou snadno odvozovat další algoritmy) a pro jednoduchost budeme jako pivota volit poslední prvek zkoumaného úseku:

```
pole = [1,2,8,42, 9, 17, -5, 20, 2]

# Přerovnej část pole od L do P - 1
def prerovnej(pole, L, P):
  pivot = pole[P - 1]

  # i je nejlevější nepřerovnaný prvek
  i = L
  # j je aktuální probíraný prvek
  for j in range(L, P - 1):
    if (pole[j] <= pivot):
      # Prohodíme tyto dva prvky
      (pole[i], pole[j]) = (pole[j], pole[i])
      i += 1

  # Dáme pivota na správné místo
  (pole[P-1], pole[i]) = (pole[i], pole[P-1])
  return i
```

```
def quicksort(pole, L, P):
  if (L >= P):
    return

  # Přerovnáme úsek...
  pivot = prerovnej(pole, L, P)
  # ... a zavoláme se rekurzivně na oba podúseky
  quicksort(pole, L, pivot)
  quicksort(pole, pivot + 1, P)
```


Bohužel volit pivota právě takto je docela nešikovné, protože se nám snadno může stát, že si vybereme nejmenší nebo největší prvek v úseku (rozmyslete si, jak by vypadala posloupnost, ve které to nastane pokaždé), takže dostaneme posloupnost délky N, rozdělíme ji na úseky délek $N-1$ a 1, načež pokračujeme s úsekem délky $N-1$, ten rozdělíme na $N-2$ a 1, atd. Přitom pokaždé na přerovnání spotřebujeme čas lineární s velikostí úseku, celkem tedy $\mathcal{O}(N + (N-1) + (N-2) + \ldots + 1) = \mathcal{O}(N^2)$.

Na druhou stranu pokud bychom si za pivota vybrali vždy *medián* z právě probíraných prvků (tj. prvek, který by se v setříděné posloupnosti nacházel uprostřed; pro sudý počet prvků zvolíme libovolný z obou prostředních prvků), dosáhneme daleko lepší složitosti $\mathcal{O}(N \log N)$. To dokážeme snadno:

Přerovnávací část algoritmu běží v čase lineárním vůči počtu prvků, které máme přerovnat. V prvním kroku QS pracujeme s celou posloupností, čili přerovnáme celkem N prvků. Následuje rekurzivní volání pro levou a pravou část posloupnosti (obě dlouhé $(N-1)/2 \pm 1$); přerovnávání v obou částech dohromady trvá opět $\mathcal{O}(N)$ a vzniknou tím části dlouhé nejvýše $N/4$. Zanoříme-li se v rekurzi do hloubky k, pracujeme s částmi dlouhými nejvýše $N/2^k$, které dohromady dají nejvýše N (všechny části dohromady dají prvky vstupní posloupnosti bez těch, které jsme si už zvolili jako pivoty). V hloubce $\lceil \log_2 N \rceil$ už jsou všechny části nejvýše jednoprvkové, takže se rekurze zastaví. Celkem tedy máme $\lceil \log_2 N \rceil$ hladin (hloubek) a na každé z nich trávíme lineární čas, dohromady $\mathcal{O}(N \log N)$.

V tomto důkazu jsme se ale dopustili jednoho podvodu: Zapomněli jsme na to, že také musíme medián umět najít. Jak z této nepříjemné situace ven?

- *Naučit se počítat medián.* Ale jak?
- *Spokojit se se „lžimediánem“:* Kdybychom si místo mediánu vybrali libovolný prvek, který bude v setříděné posloupnosti „v prostřední polovině“ (čili alespoň čtvrtina prvků bude větší a alespoň čtvrtina menší než on), získáme také složitost $\mathcal{O}(N \log N)$, neboť úsek délky N rozložíme na úseky, které budou mít délky nejvýše $(1 - 1/4) \cdot N$, takže na k-té hladině budou úseky délek nejvýše $(1 - 1/4)^k \cdot N$, čili hladin bude maximálně $\log_{1-1/4} N = \mathcal{O}(\log N)$. Místo $1/4$ by fungovala i libovolná jiná konstanta mezi nulou a jedničkou, ale ani to nám nepomůže k tomu, abychom uměli lžimedián najít.

- *Recyklovat pravidlo* typu „vezmi poslední prvek“ a jen ho trochu vylepšit. To bohužel nebude fungovat, protože pokud budeme při výběru pivota hledět jenom na konstantní počet prvků, bude poměrně snadné přijít na vstup, pro který toto pravidlo bude dávat kvadratickou složitost, i když obvykle půjde dokázat, že takových vstupů je „málo“. (Také se tak často QS implementuje.)
- *Volit pivota náhodně* ze všech prvků zkoumaného úseku. K náhodné volbě samozřejmě potřebujeme náhodný generátor a s těmi je to svízelné, ale zkusme na chvíli věřit, že jeden takový máme nebo alespoň že máme něco s podobnými vlastnostmi. Jak nám to pomůže? Náhodně zvolený pivot nebude sice přesně uprostřed, ale s pravděpodobností 1/2 to bude lžimedián, takže po průměrně dvou hladinách se ke lžimediánu dopracujeme. Proto časová složitost takovéhoto randomizovaného QS bude v průměru 2-krát větší, než lžimediánového QS, čili v průměru také $\mathcal{O}(N \log N)$. Jednoduše řečeno, zatímco fixní pravidlo nám dalo dobrý čas pro průměrný vstup, ale existovaly vstupy, na kterých bylo pomalé, randomizování nám dává dobrý průměrný čas pro všechny možné vstupy.

Hledání k-tého nejmenšího prvku

Nad QuickSortem jsme zvítězili, ale současně jsme při tom zjistili, že neumíme rychle najít medián. To tak nemůžeme nechat, a proto rovnou zkusíme vyřešit obecnější problém: najít k-tý nejmenší prvek (medián dostáváme pro $k = \lfloor N/2 \rfloor$).

První řešení této úlohy se nabízí samo. Načteme posloupnost do pole, prvky pole setřídíme nějakým rychlým algoritmem a kýžený k-tý nejmenší prvek nalezneme na k-té pozici v nyní již setříděném poli. Má to však jeden háček. Pokud prvky, které máme na vstupu, umíme pouze porovnat, pak nedosáhneme lepší časové složitosti (a to ani v průměrném případě) než $\mathcal{O}(N \log N)$ – rychleji prostě třídit nelze, důkaz můžete najít například v třídící kuchařce.

O něco rychlejší řešení je založeno na výše zmíněném algoritmu QuickSort (často se mu proto říká QuickSelect). Opět si vybereme pivota a posloupnost rozdělíme na prvky menší než pivot, pivota a prvky větší než pivot (pro jednoduchost budeme předpokládat, že žádné dva prvky posloupnosti nejsou stejné).

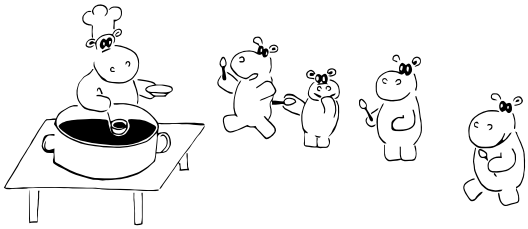
Pokud se pivot nalézá na k-té pozici, je to hledaný k-tý nejmenší prvek posloupnosti, protože právě $k-1$ prvků je menších. Zbývají dva případy, kdy tomu tak není. Pakliže je pozice pivota v posloupnosti větší než k, pak se hledaný prvek nalézá nalevo od pivota a postačí rekurzivně najít k-tý nejmenší prvek mezi prvky nalevo. V opačném případě, kdy je pozice pivota menší než k, je hledaný prvek v posloupnosti napravo od pivota. Mezi těmito prvky však nebudeme hledat k-tý nejmenší prvek, ale $(k-p)$-tý nejmenší prvek, kde p je pozice pivota v posloupnosti.

Časovou složitost rozebereme podobně jako u QuickSortu. Nešikovná volba pivota dává opět v nejhorším případě kvadratickou složitost. Pokud bychom naopak volili za pivota medián, budeme nejprve přerovnávat N prvků, pak

jich zbude nejvýše $N/2$, pak nejvýše $N/4$ atd., což dohromady dává složitost $\mathcal{O}(N + N/2 + N/4 + \ldots + 1) = \mathcal{O}(N)$. Pro lžimedián dostaneme rovněž lineární složitost a opět stejně jako u QS můžeme nahlédnout, že náhodnou volbou pivota dostaneme v průměru stejný čas jako se lžimediánem.

Program bude velmi jednoduchý, využijeme-li přerovnávací proceduru od QuickSortu:

```
def kty(pole, k, L, P):
  pivot = prerovnej(pole, L, P)
  if (k == pivot):
    return pole[pivot]
  if (k < pivot):
    return kty(pole, k, L, pivot)
  else:
    return kty(pole, k, pivot + 1, P)
```

k-tý nejmenší podruhé, tentokrát lineárně a bez náhody

Existuje však algoritmus, který řeší naši úlohu lineárně, a to i v nejhorším případě. Je založený na ďábelském triku: zvolit vhodného pivota (jak ukážeme, bude to jeden ze lžimediánů) rekurzivním voláním téhož algoritmu. Zařídíme to takto:

1. Pokud jsme dostali méně než 6 prvků, použijeme nějaký triviální algoritmus, například si posloupnost setřídíme a vrátíme k-tý prvek setříděné posloupnosti.
2. Rozdělíme prvky posloupnosti na pětice; pokud není počet prvků dělitelný pěti, poslední pětici necháme nekompletní.
3. Spočítáme medián každé pětice. To můžeme provést například rekurzivním zavoláním celého našeho algoritmu, čili v důsledku tříděním. (Také bychom si mohli pro 5 prvků zkonstruovat rozhodovací strom s nejmenším možným počtem porovnání, což je rychlejší, ale jednak pouze konstanta-krát, jednak je to daleko pracnější.)
4. Máme tedy $N/5$ mediánů. V nich rekurzivně najdeme medián m (označíme mediány pětic za novou posloupnost a na ní začneme opět od prvního bodu).

5. Přerovnáme vstupní posloupnost po quicksortovsku a jako pivota použijeme prvek m. Po přerovnání je pivot, podobně jako v předchozím algoritmu, na $(z+1)$-ní pozici v posloupnosti, kde z je počet prvků s menší hodnotou, než má pivot.
6. Opět, podobně jako u předchozího algoritmu, pokud je $k = z + 1$, pak je právě pivot m k-tým nejmenším prvkem posloupnosti. V případě, že tomu tak není a $k < z + 1$, budeme hledat k-tý nejmenší prvek mezi prvními z členy posloupnosti, v opačném případě, kdy $k > z + 1$, budeme hledat $(k - z + 1)$-ní nejmenší prvek mezi posledními $n - z - 1$ prvky.


```
def prerovnej_podle(pole, L, P, podle):
  q = L
  while (pole[q] != podle):
    q += 1
  pole[q], pole[P - 1] = pole[P - 1], pole[q]
  return prerovnej(pole, L, P)

def kty(pole, k, L, P):
  pocet = P - L
  # Jednoduché případy
  if (pocet <= 1):
    return pole[L]
  if (pocet <= 5):
    quicksort(pole, L, P)
    return pole[k]

  # Rozdělení na pětice
  petic = (pocet + 4) // 5;
  medians = [0] * petic
  for i in range(0, pocet, 5):
    if (i + 5 > pocet):
      break # Ignorujeme neúplnou pětici
    quicksort(pole, i, i + 5)
    medians[i // 5] = pole[i + 2]

  # Nalezneme medián medián pětic
  median = kty(medians, petic // 2, 0, petic)
  pivot = prerovnej_podle(pole, L, P, median)

  if (pivot == k):
    return median
  if (pivot < k):
    return kty(pole, k, L, pivot)
  else:
    return kty(pole, k, pivot + 1, P)
```

Zbývá dokázat, že tato dvojitá rekurze má slíbenou lineární složitost. Zkusme se proto podívat, kolik prvků posloupnosti po přerovnání je větších než prvek m. Všech pětic je $N/5$ a alespoň polovina z nich (tedy $N/10$) má medián menší než m. V každé takové pětici pak navíc najdeme dva prvky menší než medián pětice, takže celkem existuje alespoň $3/10 \cdot N$ prvků menších než m. Větších tedy může být maximálně $7/10 \cdot N$. Symetricky ukážeme, že i menších prvků může být nejvýše $7/10 \cdot N$.

Rozdělení na pětice, hledání mediánů pětic a přerovnávání trvá lineárně, tedy nejvýše cN kroků pro nějakou konstantu $c > 0$. Pak už algoritmus pouze dvakrát rekurzivně volá sám sebe: nejprve pro $N/5$ mediánů pětic, pak pro $\leq 7/10 \cdot N$ prvků před/za pivotem. Pro celkovou časovou složitost $t(N)$ našeho algoritmu tedy platí:

$$t(N) \leq cN + t(N/5) + t(7/10 \cdot N).$$

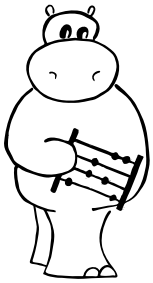
Nyní zbývá tuto rekurzivní nerovnici vyřešit, což provedeme drobným úskokem: uhodneme, že výsledkem bude lineární funkce, tedy že $t(N) = dN$ pro nějaké $d > 0$. Dostaneme:

$$dN \leq (c + 1/5 \cdot d + 7/10 \cdot d) \cdot N.$$

To platí např. pro $d = 10c$, takže opravdu $t(N) = \mathcal{O}(N)$.

Násobení dlouhých čísel

Dalším pěkným příkladem na rozdělování a panování je násobení dlouhých čísel – tak dlouhých, že se už nevejdou do integeru, takže s nimi musíme počítat po číslicích (ať už v jakékoliv soustavě – teď zvolíme desítkovou, často se hodí třeba 256-ková). Klasickým „školním“ algoritmem pro násobení na papíře to zvládneme na kvadratický počet operací, zde si předvedeme efektivnější způsob.



Libovolné $2N$-ciferné číslo můžeme zapsat jako $10^N A + B$, kde A a B jsou N-ciferná. Součin dvou takových čísel pak bude $(10^N A + B) \cdot (10^N C + D) = (10^{2N} AC + 10^N (AD + BC) + BD)$. Sčítat dokážeme v lineárním čase, násobit mocninou deseti také (dopíšeme příslušný počet nul na konec čísla), N-ciferná čísla budeme násobit rekurzivním zavoláním téhož algoritmu. Pro časovou složitost tedy bude platit $t(N) = cN + 4t(N/2)$. Nyní tuto rovnici můžeme snadno vyřešit, ale ani to dělat nebudeme, neboť nám vyjde, že $t(N) \approx N^2$, čili jsme si oproti původnímu algoritmu vůbec nepomohli.

Přijde trik. Místo čtyř násobení čísel poloviční délky nám budou stačit jen tři: spočteme AC, BD a $(A+B)\cdot(C+D) = AC+AD+BC+BD$, přičemž pokud od posledního součinu odečteme AC a BD, dostaneme přesně $AD + BC$, které jsme předtím počítali dvěma násobeními. Časová složitost nyní bude $t(N) = c'N + 3t(N/2)$. (Konstanta c' je o něco větší než c, protože přibylo sčítání a

odčítání, ale stále je to konstanta. My si ovšem zvolíme jednotku času tak, aby bylo $c' = 1$, a ušetříme si tak spoustu psaní.)

Jak naši rovnici vyřešíme? Zkusíme ji dosadit do sebe samé a pozorovat, co se bude dít:

$$\begin{aligned} t(N) &= N + 3(N/2 + 3t(N/4)) = \\ &= N + 3/2 \cdot N + 9t(N/4) = \\ &= N + 3/2 \cdot N + 9/4 \cdot N + 27t(N/8) = \ldots = \\ &= N + 3/2 \cdot N + \ldots + 3^{k-1}/2^{k-1} \cdot N + 3^k t(N/2^k). \end{aligned}$$

Pokud zvolíme $k = \log_2 N$, vyjde $N/2^k = 1$, čili $t(N/2^k) = t(1) = d$, kde d je nějaká konstanta. To znamená, že:

$$t(N) = N \cdot (1 + 3/2 + 9/4 + \ldots + (3/2)^{k-1}) + 3^k d.$$

Výraz v závorce je součet prvních k členů geometrické řady s kvocientem $3/2$, čili $((3/2)^k - 1)/(3/2 - 1) = \mathcal{O}((3/2)^k)$. Tato funkce však roste pomaleji než zbylý člen $3^k d$, takže ji klidně můžeme zanedbat a zabývat se pouze oním posledním členem:

$$3^k = 2^{k \log_2 3} = 2^{\log_2 n \cdot \log_2 3} = (2^{\log_2 n})^{\log_2 3} = n^{\log_2 3} \approx n^{1.58}$$

Konstanta d se nám „schová do $\mathcal{O}$-čka“, takže algoritmus má časovou složitost přibližně $\mathcal{O}(n^{1.58})$. Existují i rychlejší algoritmy se složitostí až $\mathcal{O}(n \log n)$, ale ty jsou mnohem ďábelštější a pro malá n se to sotva vyplatí.

Program si pro dnešek odpustíme, šetřímeť naše lesy.

Poznámky na ubrousku aneb Rozmyslete si

- Při hledání k-tého nejmenšího prvku jsme předpokládali, že všechny prvky jsou různé. Prohlédněte si algoritmy pozorně a rozmyslete si, že budou fungovat i bez toho. Opravdu?
- Proč jsme zvolili zrovna pětice? Jak by to dopadlo pro trojice? A jak pro sedmice? Fungoval by takový algoritmus? Byl by také lineární?
- Ve výpočtu $t(N)$ jsme si nedali pozor na neúplné pětice a také jsme předpokládali, že pětic je sudý počet. Ono se totiž nic zlého nemůže stát. Jak se to snadno nahlédne? Proč nestačí na začátku doplnit vstup „nekonečny“ na délku, která je mocninou deseti?
- Kdybychom neuhodli, že $t(N)$ je lineární, jak by se na to dalo přijít?
- Ještě jednou QS: Představte si, že budujete binární vyhledávací strom vkládáním prvků v náhodném pořadí. Obecně nemusí být vyvážený, ale v průměru v něm půjde vyhledávat v čase $\mathcal{O}(\log N)$. Žádný div: Stromy, které nám vzniknou, odpovídají přesně možným průběhům QuickSortu.

David Matoušek & Martin Mareš

Kuchařka čtvrté série – těžké problémy

Občas se v informatice potkáme s problémem, který nám připadá skutečně těžký, s problémem, na který zatím nikdo nezná efektivní algoritmus. V tomto textu se pokusíme si lépe vysvětlit, co vlastně pro informatika znamená sousloví *těžký problém*.

Úvod a třída problémů $\mathcal{P}$

Když mluvíme o efektivních algoritmech řešících nějaký problém, tak většinou máme na mysli algoritmus běžící v nějakém polynomiálním čase ve vztahu k velikosti vstupu. Například pro problém se vstupem velikosti N to jsou algoritmy, jejichž časovou složitost v nejhorším případě lze omezit shora nějakým polynomem závisejícím na N (sem spadají časové složitosti jako třeba $\mathcal{O}(N)$, $\mathcal{O}(N \log N)$ nebo i $\mathcal{O}(N^5)$). Jestli v základech časové složitosti tápete, nahlédněte do naší kuchařky o složitosti.[42]

Pokud na problém existuje alespoň jedno známé polynomiální řešení, tak o problému můžeme prohlásit, že leží ve třídě $\mathcal{P}$, neboli skupině úloh řešitelných v polynomiálním čase (třída tu je jen pomocné označení pro nekonečnou množinu). Stále můžeme problém zkoumat a nacházet rychlejší polynomiální řešení, ale pro teorii složitosti stačí, že máme alespoň nějaké.

Jak je to ale s úlohami, u kterých žádný polynomiální algoritmus neznáme? To mohou být třeba problémy, kde nejlepší známé řešení vede přes vyzkoušení všech možností, a jejichž časová složitost je tak třeba $\mathcal{O}(2^N)$, neboli exponenciální. Je důležité si uvědomit, že funkce jako 2^N rostou mnohem rychleji, než jakékoliv polynomiální funkce (na názornou tabulku se můžete podívat do již zmíněné kuchařky o složitosti).

I takové problémy chceme nějakým způsobem zařadit do hierarchie složitostních tříd. Než tak ale učiníme, uděláme si malou odbočku – co kdyby nám někdo k problému poskytl i nápovědu, nějaký tahák?

S mapou v bludišti

Představme si, že jsme v bludišti a hledáme nejkratší cestu ven. Můžeme určitě použít prohledávání do šířky[43] a cestu najít v čase lineárním k velikosti bludiště. To je asymptoticky nejlepší možné řešení, v nejhorším případě bude totiž bludiště jedna dlouhá nudle a i nejkratší cesta bude dlouhá lineárně vůči velikosti bludiště.

Jak by se změnila naše situace, kdybychom si ale od kamaráda půjčili tahák – mapu bludiště s vyznačenou nejkratší cestou? Pak by stačilo držet se této cesty a vyběhli bychom nejkratší cestou ven, aniž bychom kdekoliv ztráceli čas.

V nudlovém bludišti (nejkratší cesta má zhruba stejně vrcholů jako celý graf) jsme si vůbec nepomohli (takže je řešení asymptoticky stejně dobré). V alespoň

[42] `http://ksp.mff.cuni.cz/viz/kucharky/slozitost`

[43] `http://ksp.mff.cuni.cz/viz/kucharky/grafy`

trochu spletitém bludišti už budeme v cíli dříve než náš kamarád, který bloudí (prohledává) do šířky.

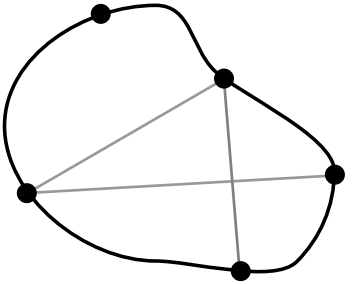
V tomto případě nám nápověda tedy zase tolik nepomohla, nalezení cesty z bludiště ven je totiž úloha, kterou umíme vyřešit v polynomiálním čase (patří do třídy $\mathcal{P}$). Pojďme si ale úlohu trochu zkomplikovat a podívejme se, jestli s nápovědou tentokrát umíme dosáhnout lepšího výsledku než bez ní.

Opět jsme v bludišti, ale tentokrát jsou na všech stanovištích umístěny koláčky. Labyrint je to zvláštní, cesty se v něm nekříží, ale je tam plno nadchodů a podchodů (lze tedy říci, že je to obecný nerovinný graf).

Naším cílem je najít okružní cestu ze startovního místa zpátky na start, abychom každé stanoviště s koláčkem prošli právě jednou (protože víc než jeden koláček nám na žádném stejně nedají).

Kdybychom tady chtěli použít procházení do šířky, bylo by to opět možné – ale tentokrát bychom se museli mnohokrát vracet, protože posloupnost stanovišť (začátek, první, druhé) může být špatná, neboť nám může zablokovat další cestu. Zároveň ale posloupnost (začátek, druhé, první) už může být dobrá.

Nebude tedy už platit, že každý vrchol při prohledávání navštívíme maximálně jednou, ale každou *posloupnost* stanovišť navštívíme maximálně jednou. Takových posloupností je ale exponenciálně mnoho vzhledem k velikosti bludiště.



Pokud si pořídíme nový tahák, na kterém bude vyznačena optimální cesta přes všechna stanoviště, tak jsme na tom ale stále dobře. Tahák bude mít lineární velikost vzhledem k počtu stanovišť (cestou proběhneme každé z nich právě jednou) a umožní nám tak problém vyřešit v lineárním čase prostým následováním vyznačené cesty.

Našli jsme tedy problém, který nevíme jak vyřešit bez nápovědy v polynomiálním čase (a tedy ho nemůžeme s klidným svědomím zařadit do třídy $\mathcal{P}$), ale s pomocí nápovědy už to umíme. Nedá se takovým způsobem definovat také nějaká třída složitosti? Dá! A to dokonce velmi důležitá.

Certifikáty a nedeterminismus

Vraťme se znovu k naší úloze s koláčky v bludišti. Zde celý problém tkví v tom, že se v některých chvílích prohledávání musíme rozhodnout, jakou z mnoha možností zkusíme nejdříve. Kdybychom pokaždé zvolili správně, tak zvládneme bludiště projít v lineárním čase.

Typický algoritmus, které napíšeme, většinou v případě více možností pokračování zvolí tu první (jeho volba je pevně určená, říkáme jí *deterministická*). Také ale můžeme přemýšlet o algoritmu, který si na každém takovém místě hodí kostkou a podle toho se rozhodne. Takový algoritmus nám na stejném vstupu

může dát při různých spuštěních různé výsledky – jeho výpočet není „předurčen“ a proto mu říkáme *nedeterministický*.

Přidejme ale k nedeterministickému algoritmu naši nápovědu neboli *certifikát*. Je to nějaká (vzhledem k velikosti vstupu) polynomiálně velká informace. Můžeme si jej představit jako data, která náš program nalezne v pomocném vstupním souboru, ke kterému program z třídy $\mathcal{P}$ nemá přístup.

Certifikát nám pomůže v každém místě, kde nevíme kudy dál, zvolit tu správnou cestu. Bez něj bychom se museli zkusit vydat každou z nabízených možností, abychom objevili tu správnou, ale s jeho pomocí se vždy vydáme správně a existenci takové cesty ověříme rychle.

Pokud náš algoritmus s použitím takového věšteckého orákula (či křišťálové koule, chcete ji), jakým je certifikát, zvládne ověřit řešení problému v polynomiálním čase, říkáme o problému, že je *nedeterministicky polynomiální*, neboli že náleží do třídy $\mathcal{NP}$.

Rozhodovací problémy a třída $\mathcal{NP}$

Aby se nám problémy lépe formálně popisovaly a zařazovaly do tříd, omezíme se v dalším textu jen na *rozhodovací problémy*. To jsou vlastně otázky, na které existují jen dvě možné odpovědi: ANO, nebo NE. Například:

- Existuje cesta z bludiště délky k?
- Je součet čísel $8 + 3$ roven 5?

Jestli se obáváte, že to výrazně sníží množství problémů, které umíme řešit, tak se nemáte proč obávat – skoro vždy se rychlé řešení rozhodovacího problému dá převést na rychlé řešení příslušného vyhledávacího problému jen s nějakým malým zpomalením. Třeba nalezení délky nejkratší cesty z bludiště můžeme udělat pomocí binárního vyhledávání a opakovaného dotazu na existenci cesty délky k (detaily si jako cvičení domyslete).

V úvodu jsme si už řekli, že třída $\mathcal{P}$ představuje problémy řešitelné v polynomiálním čase (u rozhodovacího problému to bude znamenat, že existuje polynomiální algoritmus odpovídající na zadaný vstup korektně ANO, nebo NE). U třídy $\mathcal{NP}$ si ale už musíme dát trošku pozor.

Třída $\mathcal{NP}$ je také třídou problémů. Problém do ní náleží ve chvíli, kdy existuje algoritmus a ke každému zadání, na nějž má být odpověď ANO, navíc i certifikát, pomocí kterého zvládne algoritmus existenci řešení ověřit v polynomiálním čase. Ověřením se myslí to, že odpoví ANO tehdy a jen tehdy, když řešení skutečně existuje.

Zde si dejme pozor na to, že definice nedovoluje „podvádět na druhou“, nemůžeme si do pomocného souboru prostě uložit ANO a pak jej vypsat. Tak by se pak dal řešit libovolně složitý problém, i problémy mimo třídu $\mathcal{NP}$!

Když si certifikát představíme jako ono orákulum, které nám vždy napoví správnou cestu, může být algoritmus nějaké nedeterministické řešení daného

problému. Orákulum, ať bude napovídat jakkoliv špatně, ho nikdy nemůže přesvědčit o existenci nějakého řešení, pokud takové neexistuje.

V reálné situaci (při dokazování, viz níže) si pak často za orákulum (za certifikát) zvolíme optimální řešení úlohy, kterého se stačí držet a najdeme hledanou odpověď (třeba dokážeme, že existuje cesta kratší než k).

Příslušnost do třídy $\mathcal{NP}$ tedy znamená schopnost s pomocí certifikátu dokázat existenci kladného řešení. (To vůbec nemusí znamenat, že dovedeme dokázat jeho *ne*existenci – to by byla zase jiná třída, které se říká *co*$\mathcal{NP}$.)

Asi je vám jasné, že celá třída $\mathcal{P}$ (všechny problémy z ní) jsou součástí i třídy $\mathcal{NP}$ (stačí si za certifikát zvolit třeba prázdný soubor a problém vyřešit normálním polynomiálním algoritmem). A jak už jsme naznačili výše, existují i problémy, jež leží „ještě za třídou $\mathcal{NP}$“, tedy takové, které neumíme vyřešit v polynomiálním čase ani s pomocí certifikátu. Ale dokazování toho, že takové problémy existují, už je nad rámec této kuchařky.

Je $\mathcal{P}$ rovno $\mathcal{NP}$?

Ukázali jsme, že celé $\mathcal{P}$ leží uvnitř $\mathcal{NP}$. Existuje však vůbec nějaký problém, který by byl v $\mathcal{NP}$, ale nebyl by v $\mathcal{P}$? To je otázka, jež trápí informatiky už mnoho let, jeden z nejslavnějších otevřených problémů informatiky.

Vezměme si za příklad problém z povídání o bludišti. Říká se mu *Hamiltonovská kružnice*.

Název problému: Hamiltonovská kružnice

Vstup: Neorientovaný graf.

Problém: Existuje v zadaném grafu kružnice procházející všemi vrcholy právě jednou?

Certifikát: Posloupnost vrcholů hamiltonovské kružnice.

Ověření v polynomiálním čase s certifikátem: Projdeme postupně vrcholy a ověříme, že jsou opravdu zapojeny do kružnice a kružnice je správné délky. Vrátíme NE, pokud tomu tak není.

Zatím nikdo nepřišel s řešením, které by nepoužívalo vůbec žádný certifikát. Dokonce zatím nikdo nenalezl problém, který by byl v $\mathcal{NP}$, ale bez certifikátu už jej nelze řešit v polynomiálním čase. Kdyby takový neexistoval, třídy $\mathcal{P}$ a $\mathcal{NP}$ by se rovnaly. Díky převoditelnosti problémů v $\mathcal{NP}$, které si nyní ukážeme, by dokonce stačilo najít polynomiální řešení bez certifikátu na jediný $\mathcal{NP}$-úplný problém.

Převoditelnost a $\mathcal{NP}$-úplnost

Když řešíme nějakou algoritmickou úlohu, obvykle přijdeme na nějaké řešení využívající základních technik (prohledávání do šířky, dynamické programování, zametací přímka). Vzácně se může i stát, že v problému rozpoznáme problém jiný – občas lze geometrický problém převést na třídění čísel nebo umíme popsat situaci vhodným grafem.

Ukazuje se, že se ve třídě $\mathcal{NP}$ často vyplatí problémy převádět, neboť přímá řešení jsou zde vzácná. Dokonce tak můžeme i zjistit, do které z probíraných tříd problém patří.

Převodem budeme rozumět polynomiální algoritmus, který upraví vstup jednoho problému na vstup jiného problému. Musí navíc problémy převést tak, aby správná odpověď (ANO nebo NE) na vstup prvního problému byla tatáž, jako správná odpověď na vstup druhého problému.

Jednoduchým převodem je úprava problému *Existuje cesta z bludiště ze zadaného políčka délky d?* na *Existuje cesta v grafu délky c začínající v zadaném vrcholu?*.

Do výstupního grafu za každou křižovatku dáme vrchol, za každou cestu mezi křižovatkami hranu a ke hraně si poznamenáme, jak dlouhá byla. Hodnotu c pak můžeme nechat stejně velkou jako d.

Pokud najdu správnou cestu v tomto grafu, pak nutně podobná cesta je i v bludišti, a pokud cesta v grafu není, pak není ani v bludišti. Převod je tedy korektní.

Zadefinujme si nyní pojem, který nám bude sloužit jako zkratka za to, že problém je ve třídě $\mathcal{NP}$, ale není zároveň lehký (v $\mathcal{P}$). Nemůžeme jen tak ledabyle říci „je v $\mathcal{NP}$ a není v $\mathcal{P}$“, protože to nevíme. To je právě ta slavná otázka.

Uděláme tedy krok stranou – budeme říkat, že problém je $\mathcal{NP}$*-úplný*, pokud onen problém je v $\mathcal{NP}$ a zároveň jdou všechny ostatní problémy v $\mathcal{NP}$ převést na tento problém.

Všechny problémy v $\mathcal{NP}$ na něj jdou převést? Pokud tuto definici vidíte poprvé, asi to působí dost zvláštně – je těžké si představit, že všechny grafové, geometrické, počítací problémy, o kterých víte, že jsou v $\mathcal{P}$ (a tedy i v $\mathcal{NP}$) jdou převést na nějaký $\mathcal{NP}$-úplný superproblém.

Ale je to správně, ba co víc, Cookova věta[44] říká, že existuje alespoň jeden takový problém. (Samotná definice $\mathcal{NP}$-úplného problému nezaručuje, že takový problém vůbec existuje.)

Ukazuje se však, že není sám, jsou jich stovky. Dokazovat existenci dalších $\mathcal{NP}$-úplných problémů je však o dost lehčí než dokázat Cookovu větu! Stačí totiž jen najít následující dva kroky:

- Dokázat, že problém je v $\mathcal{NP}$ – najít certifikát a polynomiální algoritmus, co jej využívá.
- Převést zadání libovolného $\mathcal{NP}$-úplného problému na zadání našeho problému tak, že náš algoritmus vlastně vyřeší onen $\mathcal{NP}$-úplný problém.

To postačí, protože pak libovolný jiný problém v $\mathcal{NP}$ nejprve převedeme na zvolený $\mathcal{NP}$-úplný problém a pak pustíme námi vymyšlený převod. Zřetězení

[44] `http://en.wikipedia.org/wiki/Cook%E2%80%93Levin_theorem`

dvou polynomiálních algoritmů (převodů) je opět polynomiální algoritmus, takže podmínka převoditelnosti je splněna.

Ukážeme si důkaz $\mathcal{NP}$-úplnosti jednoho problému na příkladu, pokud nám uvěříte, že již probíraný problém *Hamiltonovská kružnice* je $\mathcal{NP}$-úplný. Nejprve zadefinujme jiný problém:

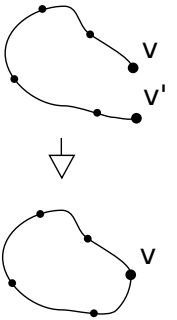
Název problému: Hamiltonovská cesta.

Vstup: Neorientovaný graf, dva speciální vrcholy x a y.

Problém: Existuje cesta z x do y (posloupnost vrcholů, ve které se žádné dva neopakují), která prochází každým vrcholem právě jednou?

Certifikát: Posloupnost vrcholů tvořící správnou cestu.

Řešení v $\mathcal{NP}$: Projděme cestu z certifikátu a ověřme, že vrcholy jdou za sebou, je jich správný počet a žádný jsme nevynechali.



Důkaz $\mathcal{NP}$-úplnosti: Převedeme předchozí problém (hamiltonovskou kružnici) na hledání hamiltonovské cesty. Uvažme graf G, ve kterém chceme najít hamiltonovskou kružnici.

Vyberme si libovolný vrchol v a vytvořme vrchol v', který bude kopií vrcholu v – do grafu přidáme hranu mezi u a v', pokud už v něm je hrana mezi u a v.

Na upravený graf zavoláme řešení problému *Hamiltonovská cesta* mezi vrcholy v a v'. Pokud taková cesta existuje, tak nutně v původním grafu G existuje hamiltonovská kružnice. Cesta z vrcholu v' přesně odpovídá pokračováni kružnice poté, co přijde do vrcholu v.

Pseudopolynomiální algoritmy

Znáte problém batohu? Jeho varianty jsou oblíbené na programovacích soutěžích. Zadat se může třeba takto: mějme na vstupu seznam N dvojic kladných přirozených čísel, kde každá dvojice označuje váhu a cenu nějakého předmětu. Nakonec dostaneme na vstupu ještě číslo B, které udává nosnost našeho batohu.

Otázka zní: Jaký je nejcennější možný náklad, který přesto nepřesahuje váhový limit batohu?

Možná víte, že úloha jde řešit dynamickým programováním – vytvořím si pole `podbatoh[]` od 1 do B, kde `podbatoh[`i`]` je maximální hodnota, kterou bych si odnesl v batohu o nosnosti i. Postupně od první věci do poslední pak projdu celé pole `podbatoh[]` „zprava doleva“ od B do 1 a zkusím, jestli je výhodnější do batohu vložit novou věc a volné místo doplnit starými (optimální volné místo pro předchozí věci máme napočítané), nebo si nechat jen ty staré. Tuto hodnotu pak zapíšeme jako aktuální pro váhu i na místo `podbatoh[`i`]`.

Po N průchodech tohoto pole dostaneme řešení pro všechny věci dohromady na políčku `podbatoh[`B`]`. Celková složitost je $\mathcal{O}(NB)$, to je polynom, algoritmus je tedy polynomiální.

Světe div se, toto řešení je ve skutečnosti exponenciální. Kde jsme v řešení udělali chybu? Nikde – naše složitost závisela na B, ovšem když se podíváme do vstupních dat, tak pokud jsou zapsána v binárním (nebo ternárním a vyšším) tvaru, zápis čísla B byl veliký $\mathcal{O}(\log_2 B)$, ale naše složitost závisela na $B = 2^{\log_2 B}$, tedy exponenciálně vzhledem k velikosti vstupu.

Problém batohu, respektive jeho rozhodovací verze, je dokonce $\mathcal{NP}$-úplný problém.

Algoritmům, které řeší nějakou úlohu a jsou polynomiální vůči *hodnotě* čísel na vstupu, ale exponenciální ve *velikosti zápisu* těchto čísel, říkáme *pseudopolynomiální algoritmy*. Některé další $\mathcal{NP}$-úplné problémy mají pseudopolynomiální řešení (jako například *Dva loupežníci* níže), ale dá se dokázat, že na jiné problémy pseudopolynomiální algoritmus neexistuje (pokud $\mathcal{P} \neq \mathcal{NP}$).

Mimochodem: pokud bychom na vstupu zapisovali čísla v unárním zápisu (tedy místo každého čísla by bylo třeba tolik hvězdiček, jakou hodnotu představuje), každý pseudopolynomiální problém by ležel v $\mathcal{P}$.

Poznámky na závěr

Otázku „Je třída $\mathcal{P}$ rovna $\mathcal{NP}$?“ se již snažilo rozlousknout mnoho matematiků a informatiků. Tato teorie přinesla spoustu zajímavých výsledků, například už se podařilo dokázat, že některými technikami tuto domněnku nelze nikdy dokázat, ani vyvrátit.

Kdyby platilo $\mathcal{P} = \mathcal{NP}$, pak by mnoho lidi zajásalo – mnoho přirozených problémů, které nastávají i v reálném životě, by najednou bylo řešitelných rychle. Navíc by krachlo dosavadní šifrování a bylo by možné najít rychle důkaz ke každému pravdivému tvrzení výrokové logiky.

Tato rovnost by se dala hypoteticky ukázat velice snadno – stačilo by najít jeden polynomiální algoritmus pro libovolný $\mathcal{NP}$-úplný problém! Většina informatiků studujících složitost se však domnívá, že se třídy nerovnají.

To ale neznamená, že si to nemáte zkusit dokázat! Naopak, bojovat s $\mathcal{NP}$-úplnými problémy je užitečné i v reálném světě – mnohdy jde vymyslet třeba dobrá aproximace řešení.

Například nenajdeme hamiltonovskou kružnici v polynomiálním čase, ale nalezneme nějakou relativně dlouhou kružnici, která nám v praxi může stačit, pokud podle ní třeba chceme vést náročný cyklistický závod.

O aproximacích je toho v literatuře napsáno mnoho zajímavého, pokud byste si o nich chtěli přečíst více v češtině, zkuste třeba kapitolu připravovaných skript z předmětu ADS na Matfyzu.[45]

[45] `http://mj.ucw.cz/vyuka/ads/49-prevody.pdf`

Více o třídě $\mathcal{NP}$ i o dalších aspektech složitosti můžete najít na stejné adrese, nebo zkuste vynikající anglicky psanou knížku *Algorithms* od profesorů exotických jmen Dasgupta, Papadimitriou a Vazirani.

Jak už jsme zmínili, existují i problémy, které jsou mimo $\mathcal{P}$ i $\mathcal{NP}$, a dokonce existuje spousta různých dalších tříd problémů. Je jich celá zoologická zahrada – pěkný souhrn můžete najít na stránkách Univerzity ve Waterloo.[46]

Seznam $\mathcal{NP}$-úplných problémů

Sedíte-li nad zatím nevyřešenou úlohou, kterou stále nemůžete rozlousknout, je možné, že bude $\mathcal{NP}$-úplná. Abyste mohli mezi $\mathcal{NP}$-úplnými úlohami převádět, tak je dobré znát jich aspoň hrstku, podle toho, je-li problém grafový, rovnicový, a tak dále.

V následujícím seznamu najdete několik úloh, které jsou zaručeně $\mathcal{NP}$-úplné. Převody se nám sem už sice nevešly, ale většinu z nich (ne-li všechny) zvládnete vymyslet sami – zkuste to!

Název problému: Hamiltonovská kružnice

Vstup: Neorientovaný graf.

Problém: Existuje v zadaném grafu kružnice procházející všemi vrcholy právě jednou?

Název problému: Hamiltonovská cesta.

Vstup: Neorientovaný graf, dva speciální vrcholy x a y.

Problém: Existuje cesta z x do y (posloupnost vrcholů, ve které se žádné dva neopakují), která prochází každým vrcholem právě jednou?

Název problému: Splnitelnost

Vstup: Logická formule. Tu tvoří proměnné a logické spojky negace $\neg$, konjunkce $\wedge$ a disjunkce $\vee$. Například

$$(x \wedge (\neg y)) \vee z.$$

Problém: Můžeme proměnným přiřadit hodnoty 0 nebo 1 tak, že výsledná vyhodnocená formule má hodnotu 1?

Název problému: Součet podmnožiny

Vstup: Seznam nezáporných celých čísel, speciální číslo k.

Problém: Existuje podmnožina čísel, jejíž součet je přesně k?

---

[46] `https://complexityzoo.uwaterloo.ca/`

Název problému: Batoh
Vstup: Seznam dvojic nezáporných čísel, kde dvojice označuje hodnotu a váhu předmětu. Přirozené číslo b – nosnost batohu, přirozené číslo k.
Problém: Umíme vložit do batohu předměty o hodnotě alespoň k, aniž bychom přešli přes limit váhy b?

Název problému: Dva loupežníci
Vstup: Seznam nezáporných celých čísel.
Problém: Existuje rozdělení seznamu na dvě hromádky tak, že každé číslo bude v právě jedné hromádce a v každé hromádce bude stejný součet čísel?

Název problému: Klika
Vstup: Neorientovaný graf, číslo k.
Problém: Existuje v grafu úplný podgraf o velikosti k, tedy k vrcholů takových, že mezi každými dvěma z nich vede hrana?

Název problému: Nezávislá množina
Vstup: Neorientovaný graf, číslo k.
Problém: Existuje v grafu prázdný podgraf o velikosti k, tedy k vrcholů, že žádné dva z nich nejsou spojeny hranou?

Název problému: Trojbarevnost grafu
Vstup: Neorientovaný graf.
Problém: Lze vrcholy tohoto grafu obarvit třemi barvami tak, že každá hrana sousedí s vrcholy dvou různých barev?

Název problému: Rozparcelování roviny
Vstup: Seznam bodů v rovině, kde každý má přiřazenu jednu z b barev, číslo k.
Problém: Umíme rozdělit rovinu pomocí k přímek tak, že v každé oblasti jsou jen body té samé barvy?

Název problému: 3D párování
Vstup: Seznam mužů, žen a zvířátek, následovaný seznamem kompatibilních trojic tvaru $\{\text{muž}, \text{žena}, \text{zvířátko}\}$. Tyto trojice říkají, která trojice muž, žena a zvířátko by se dohromady snesla.
Problém: Můžeme všechny muže, ženy a zvířátka z prvního seznamu rozdělit do trojic tak, že každá trojice je kompatibilní a každá bytost je právě v jedné trojici?

Kuchařku sepsali

Martin Böhm & Jirka Setnička

Kuchařka páté série – hledání v textu

Řetězec je v podstatě jakákoli posloupnost symbolů zapsaná za sebou a s nimi budeme v této kapitole pracovat. Každého napadne „vyhledávání v textu“ nebo „hledání jmen v telefonním seznamu“, ale řetězce najdeme i na nižších úrovních informatiky. Například celé číslo zakódované v binární soustavě, které dostaneme na vstupu programu, je také jen řetězec nul a jedniček.

Jiný příklad použití řetězců (a jejich algoritmů) najdeme v biologii. DNA není o mnoho více, než chytré uložení posloupnosti čtyř znaků/nukleových bazí – a chceme-li hledat vzory anebo konkrétní podposloupnosti, bude se nám hodit znalost základních algoritmů pro práci s řetězci.

Nemáme bohužel šanci vysvětlit *všechny* algoritmy s řetězci, protože je příliš mnoho možných věcí, co s řetězci dělat. Převáděním řetězců na čísla (hešováním) jsme se věnovali v jiné kuchařce, v této se budeme soustředit na algoritmy, které se objevují spíše v práci s textem.

Kromě úvodu popíšeme dva *stavební kameny* textových algoritmů, což bude jedna datová struktura pro slovníky (trie) a jedno vyhledání v textu s předzpracováním hledaného slova (a jeho rozšíření pro více slov). S jejich znalostí se pak mnohem snáze vymýšlí řešení složitějších, reálnějších problémů.

Jak řetězce chápat

Když programátor dělá první krůčky, často moc netuší, co s těmi řetězci vlastně může a nesmí dělat. V programovacím jazyce to je jasné – něco mu jazyk dovolí a na něco nejsou prostředky. Ale jak to je na úrovni ryze teoretické?

Jak jsme si řekli na začátku, řetězec bude posloupnost nějakých symbolů, kterým říkáme *znaky*. Tyto znaky jsou z nějaké množiny, které říkáme *abeceda*. Abeceda může být jen {`0`, `1`} pro čísla v binárním zápisu, klasické {`A-Z`, `a-z`} pro anglickou abecedu anebo plný rozsah univerzální znakové sady Unicode, která má až 2^{31} znaků. Nezapomínejme, že nejenom písmena a číslice, ale i mezery a interpunkce jsou znaky!

Vidíme, že zanedbat velikost abecedy při odhadu složitosti by bylo příliš troufalé, a tak budeme velikost abecedy označovat $|\Sigma|$. Abeceda sama se v textech o řetězcích často značí řeckým Σ.

O znacích samotných předpokládáme, že jsou dostatečně malé, abychom s nimi mohli pracovat v konstantním čase, podobně jako s celými čísly v ostatních kapitolách.

Nyní hlavní otázka – máme chápat řetězec jako pole znaků, nebo jako spojový seznam? Šalamounská odpověď: můžeme s ním pracovat tak i tak. Když budeme potřebovat převést řetězec na spojový seznam (protože se nám hodí rychlé přepojování řetězců), tak si jej převedeme. Tento převod nás samozřejmě bude stát čas lineárně závislý na *délce* řetězce. Budeme ji dále značit L; časová složitost převodu bude $\mathcal{O}(L)$.

Standardně se ale počítá s tím, že řetězec je uložen v poli někde v paměti (již od začátku algoritmu), takže ke každému znaku můžeme přistupovat v konstantním čase.

Jelikož jsme řetězce definovali jako posloupnosti, nesmíme zapomínat ani na *prázdný řetězec* ε. A když už máme řetězec, určitě máme i *podřetězec* – souvislou podposloupnost znaků jiného řetězce. Například `BAR`, `RET`, ε i `KABARET` jsou podřetězce slova (řetězce) `KABARET`; `KAT` však podřetězcem není.

Často nás budou zajímat dva zvláštní druhy podřetězců. Pokud ze slova odstraníme nějaký souvislý úsek na konci, vznikne podřetězec, kterému říkáme *prefix* (česky předpona), a pokud odstraníme nějaký souvislý úsek ze začátku, dostaneme *suffix* neboli příponu. `RET` je suffix slova `KABARET`, `KABA` je zase jeho prefixem.

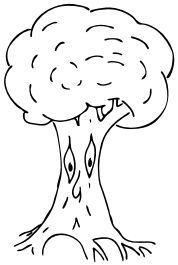
Terminologie dovoluje zepředu i zezadu odstranit prázdný řetězec – to znamená, že slovo je samo sobě prefixem i suffixem. Pokud chceme mluvit o prefixech, suffixech nebo obecně podřetězcích, kde jsme museli alespoň jeden znak odtrhnout, označíme takové podřetězce jako *vlastní*.

Pro některá použití řetězců je důležité, abychom je mohli porovnávat – když máme řetězce R a S, chceme rozhodnout, který je menší, a který je větší. Jaké přesně toto uspořádání bude, závisí na naší aplikaci, ale mnohdy se používá tzv. *lexikografické uspořádání*.

Pro lexikografické uspořádání potřebujeme nejprve zadané lineární uspořádání na znacích. Tím se myslí takové, kde jsou všechny prvky navzájem porovnatelné a v podstatě to znamená, že jsou uspořádány v jedné řadě za sebou (kromě binárního $\texttt{0} < \texttt{1}$ se často používá „telefonní“ $\texttt{A} = \texttt{a} < \texttt{B} = \texttt{b} < \ldots < \texttt{Z} = \texttt{z}$, které je ovšem lineární až na velikost znaků).

Když máme zadané uspořádání na znacích, na všechny řetězce je rozšíříme následovně: Nejkratší je prázdný řetězec. Ostatní řetězce třídíme podle jejich prvního znaku. Jestliže se první znak shoduje, tak podle druhého znaku, atd. Pokud přitom znaky jednoho z řetězců dojdou dřív, prohlásíme tento řetězec za menší.

Platí tedy třeba $\varepsilon < \texttt{A} < \texttt{AUTO} < \texttt{AUTOBUS} < \texttt{AUTOGRAM} < \texttt{AUTOR} < \texttt{BAMBITKA} < \texttt{BARNABAS} < \texttt{Z}$.



Adresář pomocí trie

Typický „textový“ problém je udržování množiny řetězců – můžete si představit třeba slovník. Slova ve slovníku si chceme šikovně předzpracovat, abychom pak mohli efektivně odpovídat na otázky typu: „Je slovo S obsaženo ve slovníku?“ Můžeme také po předzpracování chtít přidávat nové položky, nebo i odebírat staré.

KSP

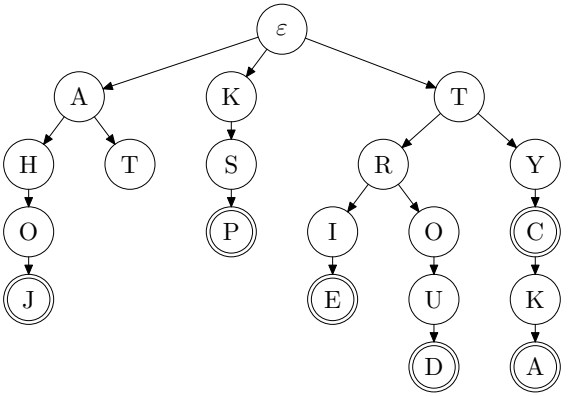
Pokud bychom nemuseli odebírat slova, můžeme použít hešování, které je rychlé a účinné. Více o něm najdete v hešovací kuchařce.[47] Má však tu nevýhodu, že při velkém zaplnění se začne chovat pomaleji a mírně nepředvídatelně.

Ukážeme si jiné řešení, které je také asymptoticky rychlé a není ani příliš náročné na paměť. Využívá stromové struktury a říká se mu *trie* (vyslovujeme česky „tryje“ a anglicky jako část slova „retrieval“; z něhož slovo trie vzniklo). V češtině se občas používá také označení „písmenkový strom“.

recepty

Trie bude zakořeněný strom. V prvním patře se bude větvit podle prvního písmene slova, ve druhém podle dalšího, a tak dále.

Obrázek vydá za tisíc definic, pojďme se podívat, jak vypadá trie pro slova `AHOJ`, `AT`, `KSP`, `TRIE`, `TROUD`, `TYC`, `TYCKA`. Pro přehlednost písmena místo na hrany kreslíme do následujících vrcholů:



Všimněte si, že vrcholy v hloubce h (tedy v h-tém patře trie) odpovídají prefixům délky h zadaných slov. Například prefixy délky 2 jsou `AH`, `AT`, `KS`, `TR` a `TY`. Hrana mezi prefixy vede právě tehdy, lze-li jeden z druhého získat připsáním písmene na konec.

Jak bychom takovou trii postavili algoritmem? Přesně, jak jsme ji definovali: každé slovo ze slovníku budeme procházet znak po znaku a bude-li nějaká hrana chybět, tak ji vytvoříme a pokračujeme dále podle slova.

[47] `http://ksp.mff.cuni.cz/viz/kucharky/hesovani`

Z takto popsané trie bohužel nepoznáme, kde končí slovo ze slovníku a kde končí jen jeho prefix. Standardní způsoby, jak to vyřešit, jsou dva: buď si do každého vrcholu přidáme informaci o tom, je-li koncem celého slova nebo ne (jak je to naznačeno dvojitými kroužky v obrázku), anebo si rozšíříme abecedu o speciální znak, který se v ní předtím nevyskytoval – třeba `$` – a pak všem slovům přilepíme tento `$` na konec.

Budeme-li se později ptát, bylo-li slovo ve slovníku, po průchodu trií zkontrolujeme ještě, jestli z konečného vrcholu vede hrana odpovídající znaku `$`.

Ještě jsme si nerozmysleli, jak budeme v jednotlivých vrcholech trie reprezentovat hrany do delších prefixů. Abychom mohli vyhledávat skutečně lineárně, potřebovali bychom umět v konstantním čase odpovědět na otázku „má vrchol P potomka přes hranu se znakem c?“.

Abychom zajistili konstantní čas odpovědi, museli bychom mít v každém vrcholu pole indexované znaky abecedy. To ovšem znamená, že takové pole budeme muset vytvořit, a tedy alokovat $|\Sigma|$ políček v každém znaku.

To zvýší paměťovou náročnost trie (a časovou náročnost její stavby) na $\mathcal{O}(D \cdot |\Sigma|)$, kde D značí velikost vstupu, čili součet délek všech slov ve slovníku. To je naprosto přijatelné pro malé abecedy, ale už pro {`A-Z`, `a-z`} je tento faktor roven 52 a pro Unicode je taková alokace nemyslitelná.

Jak z toho ven? Můžeme oželet konstantní rychlost dotazu a použít namísto pole třeba binární vyhledávací strom nebo hešovací tabulku všech znaků, kterými aktuální prefix může pokračovat. Nebo můžeme každý znak velké abecedy zapsat pomocí několika znaků menší abecedy. Tou menší abecedou může být třeba {`0`, `1`}. Tehdy nahradíme každý znak původní abecedy $\lceil \log_2 |\Sigma| \rceil$ novými (jeho zápisem ve dvojkové soustavě). Tím se časová slozitost konstrukce zlepší na $\mathcal{O}(D \cdot \log |\Sigma|)$ a časová složitost dotazu na slovo délky L zhorší na $\mathcal{O}(L \cdot \log |\Sigma|)$.

A jsme hotovi! S trií můžeme v lineárním čase odpovídat na dotazy „Vyskytuje se dané slovo ve slovníku?“, přidávat a odebírat další položky za běhu a nejen to – víc o tom ve cvičeních.

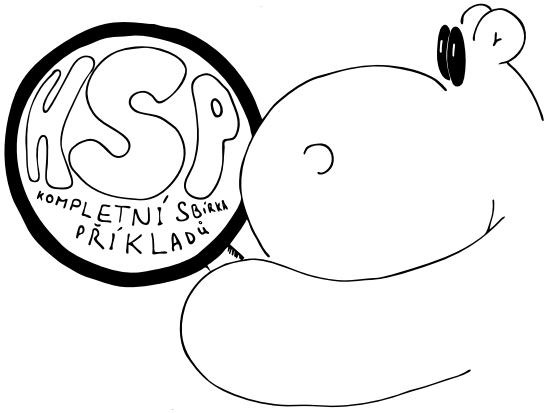
Poznámky

- Chcete-li algoritmus konstrukce trie vidět napsaný v Pascalu, podívejte se do knihy *Algoritmy a programovací techniky*.
- Triím se také říká *prefixové stromy*, což popisuje, že každý vrchol odpovídá prefixu některého slova ve slovníku.
- Kdybychom chtěli, mohli bychom pomocí trie vyhledávat v textu v lineárním čase. Můžeme přeci postavit slovník ze všech slov v daném textu, a pak procházet příslušnou trii. Má to ale pár háčků: jednak je často hledaný řetězec krátký, ale text se nevejde do paměti. Druhak, pokud bychom použili jako oddělovač mezery, mohli bychom hledat jen jednotlivá slova, a nikoli jejich konce nebo delší kusy věty.

- Asi se po poslední poznámce ptáte – existuje nějaká modifikace trie, která umí hledat libovolnou část textu? Ano, jmenuje se *suffixový strom* a jdou s ní dělat spousty krásných kousků. Říká se, že každou řetězcovou úlohu lze řešit v lineárním čase pomocí suffixových stromů. Víc se o nich dočtete třeba v knížce *Krajinou grafových algoritmů.*[48]

Cvičení

- Řekněme, že chceme slovník na vstupu setřídit v lexikografickém pořadí (definovaném v sekci „Jak řetězce chápat“). Problémem pro klasické třídící algoritmy je to, že porovnání dvou řetězců není bohužel konstantně rychlé. Vymyslete způsob, jak setřídit takový slovník rychle pomocí trie.
- *Komprese trie.* Co kdybychom chtěli odstranit přebytečné vrcholy trie, tedy ty, v nichž se slova nevětví? Rozmyslete si, jestli by něčemu vadilo místo takovýchto cest mít jen jednotlivé hrany. Zesložití se konstrukce nebo vyhledávání? Mimochodem, je celkem jasné, že takováto *komprimovaná trie* přinese jen konstantní zrychlení dotazů i prostoru, a tak na soutěžích apod. stačí použít základní variantu.

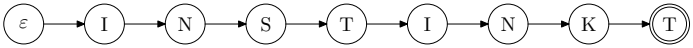


Vyhledávání v textu

Začátek situace je asi zřejmý – máme na vstupu zadán dlouhý text a krátké slovo. Slovo si můžeme nějak předzpracovat, načež projdeme co nejrychleji text a nahlásíme jeden nebo všechny výskyty slova. Zajímají nás při tom i výskyty, které se navzájem překrývají: v textu `NANANA` se slovo `NANA` vyskytuje dvakrát. Často se hovoří o „hledání jehly v kupce sena“, pročež se textu přezdívá *seno* a hledanému slovu *jehla*. Délku jehly označíme J a délku textu S.

[48] `http://mj.ucw.cz/vyuka/ga/`

Představme si nejdříve hledané slovo jako spojový seznam, třeba slovo `INSTINKT`:



Mohli bychom text začít procházet znak po znaku a kontrolovat, zda se text shoduje s naším slovem/spojovým seznamem. Pokud by si znaky odpovídaly, skočíme na další znak z textu a i na další znak v seznamu. Co když se ale neshodují? Pak nemůžeme jen skočit na další znak textu – co kdybychom v textu narazili na slovo `INSTINSTINKT`?

Musíme se tedy vrátit nejen na začátek spojového seznamu, ale i zpátky v textu na druhý znak, který jsme označili jako odpovídající, a zkoušet porovnávat s jehlou znovu od začátku. To už naznačuje, že takto získaný algoritmus nebude lineární, protože se musí vracet zpět v textu o délku jehly.

Sice je předchozí popis skutečně v nejhorším případě složitý $\mathcal{O}(S \cdot J)$, avšak stačí malá úprava a složitost přejde na lineární $\mathcal{O}(S + J)$. Ve skutečnosti algoritmus nezpomalovalo vracení se – za špatnou složitost mohl fakt, že jsme se vraceli *příliš zpátky*.

Třeba v našem příkladu s textem `INSTINSTINKT` se nemusíme vracet ve spojovém seznamu na začátek, jakmile načteme `INSTINS`. Mohli jsme se vrátit jen na druhý znak, tedy do prvního `N`, a pak kontrolovat, jaký znak pokračuje dál. Když následuje `S` jako v našem případě, můžeme pokračovat dále v čtení a nevracíme se v textu. Kdyby text byl jiný, třeba `INSTINB`, vrátili bychom se po načtení `B` na začátek spojového seznamu a v textu bychom pokračovali dále bez zastavení.

Pro každý znak ve spojovém seznamu si tedy určíme políčko spojového seznamu, na které skočíme, pokud se následující znak v textu liší od toho očekávaného. Pořadové číslo tohoto políčka nám poradí tzv. *zpětná funkce* F, což bude funkce definovaná pomocí pole, kde $F[i]$ bude pořadové číslo políčka, na které se má skočit z políčka číslo i. Porovnávat pak budeme s následujícím znakem. Pokud $F[i] = 0$, znamená to, že máme začít porovnávat úplně od prvního znaku jehly.

Pokud máte rádi grafovou terminologii, můžete se na náš spojový seznam dívat jako na graf a hovořit o *zpětných hranách*.

Zatím jsme ale přesně nepopsali, na které políčko přesně bude zpětná funkce ukazovat. Nechť chceme určit zpětné políčko pro druhé `N` ve slově `INSTINKT`. Pracujeme teď s prefixem `INSTIN`. Selsky řečeno, chceme najít „konec slova `INSTIN` takový, že je stejný, jako začátek slova `INSTIN`“.

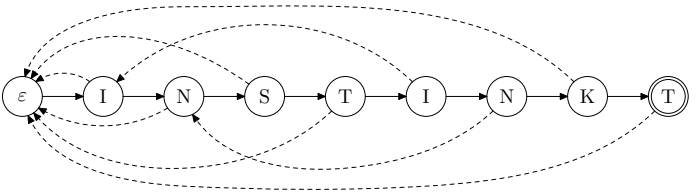
Abychom náš požadavek upřesnili, zamyslíme se nad zpětným políčkem pro jiné slovo. Co kdyby jehlou bylo slovo `ABABABC` a my určovali zpětné políčko pro `ABABAB`? Kdybychom ukázali na první písmenko `B`, nebylo by to správně, protože

pak bychom pro text `ABABABABC` nezahlásili výskyt jehly, což je jasná chyba. Musíme se vrátit už na `ABAB`!

Zajímá nás tedy ne libovolný suffix, který je stejný jako začátek, ale nejdelší takový konec/suffix. A ještě navíc ne jen ten nejdelší, ale nejdelší „netriviální" – slovo `INSTIN` je samo sobě prefixem a suffixem, ale zpětná funkce pro `N` by se neměla cyklit, měla by vést zpátky.

Řekněme to tedy znova, zcela formálně: pokud bychom právě určovali hodnotu zpětné funkce pro znak číslo i, kterému odpovídá prefix P, pak její hodnota bude *délka nejdelšího vlastního suffixu* slova P, pro který ještě platí, že *je zároveň prefixem* P.

Pro slovo `INSTINKT` vypadá spojový seznam se zpětnou funkcí (zakreslenou pomocí ukazatelů) takto:



Nyní vyvstávají dvě otázky: Jakou to má celé časovou složitost? A jak spočítat zpětnou funkci?

Poperme se nejdříve s tou první. Pro každý znak vstupního textu mohou nastat dva případy: Buď znak rozšiřuje aktuální prefix, nebo musíme použít zpětnou funkci. První případ má jasně konstantní složitost, druhý je horší, neboť zpětná funkce může být pro jeden znak volána až J-krát.

Při každém volání však klesne pořadové číslo aktuálního stavu (políčka) alespoň o jedna, zatímco kdykoliv stav prodlužujeme, roste jen o jeden znak. Proto všech zkrácení dohromady může být nejvýše tolik, kolik bylo všech prodloužení, čili kolik jsme přečetli znaků textu. Celkem je tedy počet kroků automatu lineární v délce textu, $\mathcal{O}(S)$.

Konstrukci zpětné funkce provedeme malým trikem. Všimněme si, že $F[i]$ je přesně číslo stavu, do nějž se dostaneme při spuštění našeho vyhledávacího algoritmu na řetězec, který tvoří prefix délky i z jehly bez prvního znaku.

Proč to tak je? Zpětná funkce říká, jaký je nejdelší vlastní suffix daného stavu, který je také stavem, zatímco políčko, ve kterém po i krocích skončíme, označuje nejdelší suffix textu, který je stavem. Tyto dvě věci se přeci liší jen v tom, že ta druhá připouští i nevlastní suffixy, a právě tomu zabráníme odstraněním prvního znaku.

Takže F získáme tak, že spustíme vyhledávání na část samotné jehly. Jenže k vyhledávání zase potřebujeme funkci F. Jak z toho ven? Budeme zpětnou funkci vytvářet postupně od nejkratších prefixů. Zřejmě $F[1] = 0$. Pokud již

máme $F[i]$, pak výpočet $F[i+1]$ odpovídá spuštění automatu na slovo délky i a při tom budeme zpětnou funkci potřebovat jen pro stavy délky i nebo menší, pro které ji již máme hotovou.

Navíc nemusíme pro jednotlivé prefixy spouštět výpočet vždy znovu od začátku – $(i+1)$-ní prefix je přeci prodloužením i-tého prefixu o jeden znak. Stačí tedy spustit algoritmus na jehlu bez prvního znaku a sledovat, jakými stavy bude procházet – to budou přesně hodnoty zpětné funkce.

Vytvoření zpětné funkce se nám tak nakonec zredukovalo na jediné vyhledávání v textu o délce $J-1$, a proto poběží v čase $\mathcal{O}(J)$. Časová složitost celého algoritmu tedy bude $\mathcal{O}(S+J)$. Dodáme už jen, že tento algoritmus poprvé popsali pánové Knuth, Morris a Pratt a na jejich počest se mu říká KMP. Naprogramovaný bude vypadat následovně (čtení vstupu jsme si odpustili):


```
jehla = "INSTINKT"
seno = "INSTINSTINKTINSTINKT"
J = len(jehla)
S = len(seno)
F = [None] * J  # Zpětná funkce

def krok(i, znak):
    if i < J and jehla[i] == znak:
        return(i + 1)
    elif i > 0:
        return krok(F[i - 1], znak)
    else:
        return 0

# Konstrukce zpětné funkce
F[0] = 0
for i in range(1, J):
    F[i] = krok(F[i - 1], jehla[i])

# Procházení textu
stav = 0
for i in range(S):
    stav = krok(stav, seno[i])
    if stav == J:
        print(i - J + 1, "až", i)
```

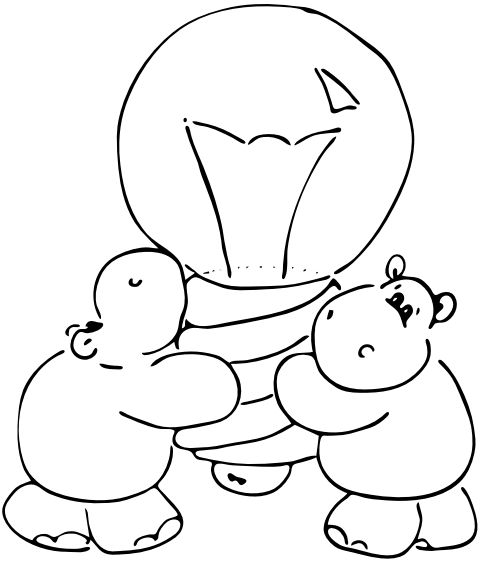
Poznámky

- Pro anglický nebo český text je použití takto sofistikovaného algoritmu skoro škoda, protože v obou jazycích se stává jen málokdy, že bychom měli několik slov spojených dohromady. Prakticky bude stačit i na začátku zmíněný naivní algoritmus. Na soutěžích a olympiádách ale pište raději algoritmus KMP.

- Hešování lze použít i na vyhledávání řetězce v textu. Obzvláště vhodné jsou na to *rolling hash functions* (neboli „okénkové hešovací funkce“), které umí v konstantním čase přepočítat heš, ubereme-li nějaký znak na začátku a přidáme-li jiný na konci – jako kdybychom se dívali na text skrz posouvající se okénko.

Cvičení

- Rozmyslete si, že když vyhledáváme více slov, ne jen jedno, a algoritmus musí vypsat všechny výskyty na výstup, můžeme se dobrat vyšší než lineární složitosti v závislosti na vstupu. Na čem potom taková časová složitost také záleží?
- Vymyslete nějakou vhodnou okénkovou hešovací funkci pro vyhledávání jedné jehly.



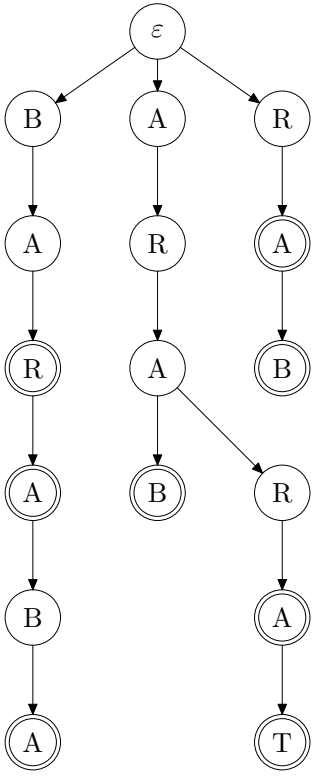
Vyhledávání jehelníčku

Co kdybychom neměli jen jednu jehlu/hledané slovo, ale celý jehelníček, čili seznam hledaných slov? I to lze řešit podobnou metodou, jakou jsme hledali jedno slovo. Tento algoritmus se nazývá po tvůrcích *algoritmus Aho-Corasicková* a spočívá v tom, že jednoduchý spojový seznam nahradíme trií a do trie opět přidáme zpětné hrany.

Budeme postupovat podobně jako u KMP. Nejprve naskládáme jehelníček do trie. Pro příklady v této kuchařce použijeme jehelníček `ARAB`, `ARARA`, `ARARAT`, `BAR`, `BARA`, `BARABA`, `RA` a `RAB`.

Dalším krokem v KMP bylo sestrojení zpětných hran. Nejprve jsme sestrojili zpětnou hranu pro první znak slova, pak pro druhý atd. Ve trii to bude o něco složitější.

Na první pohled se může zdát, že bychom mohli automat sestrojit tak, že bychom vyrobili KMP pro první slovo, pak KMP pro druhé slovo s využitím struktury prvního atd., ale to má háček.



Zpětné hrany totiž nemusí vést do předka. Například pro slovo `BARAB` povede zpětná hrana do slova `ARAB`, z toho do slova `RAB` a z toho do `B`. Kdybychom ale zkonstruovali automat výše popsaným způsobem (a začali slovem `BARAB`), nebude existovat v trii ani `ARAB`, ani `RAB`, takže bychom vedli zpětnou hranu chybně do `B`.

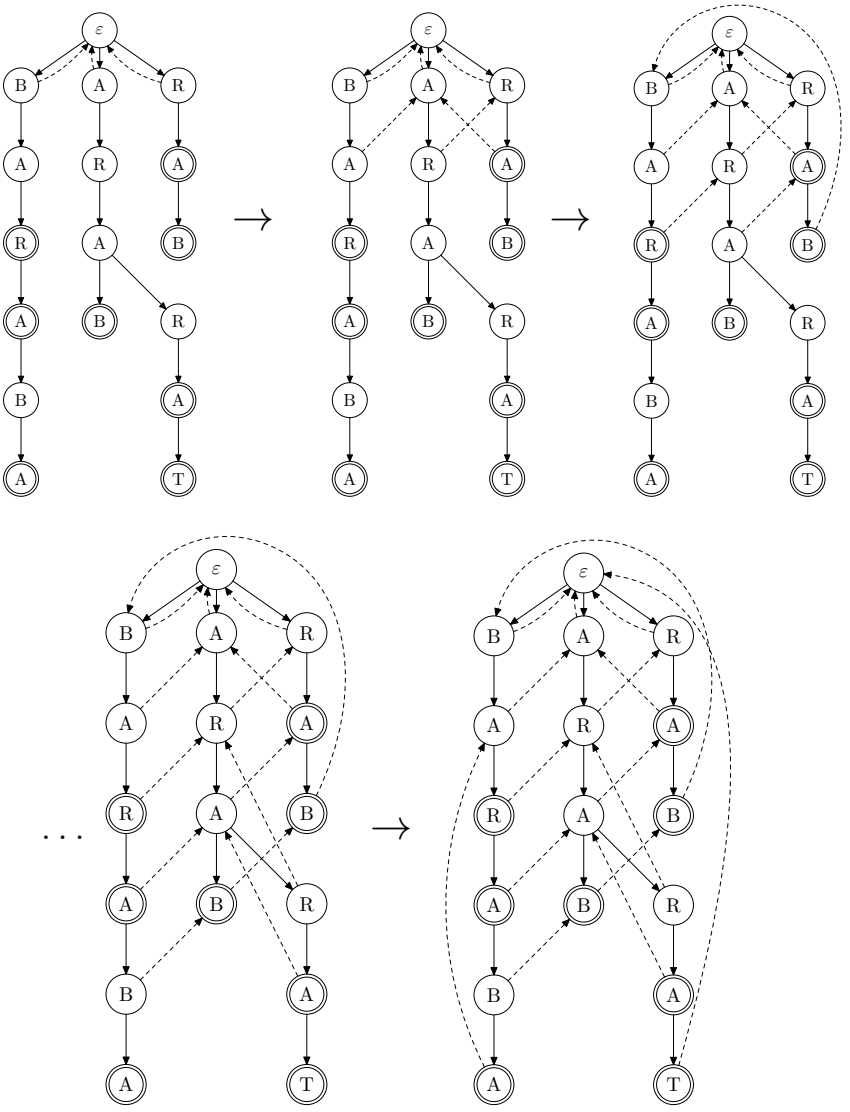
Můžeme se ale opřít o stejný trik, jako při konstrukci KMP. Budeme opět vyhledávat nejdelší vlastní suffix. Kam dojde výpočet po jeho vyhledání, tam povede zpětná hrana.

Zkusíme tedy nejprve sestrojit celou trii a pak postupně vyhledat nejdelší vlastní suffix pro každé ze slov. Ouha, to ale také nefunguje. Když začneme slovem `BARABA` a budeme tedy vyhledávat `ARABA`, nalezneme v trii úspěšně prefix `ARAB`, ale `ARABA` již v trii není. Měli bychom přejít ze slova `ARAB` po zpětné hraně, ale tu ještě nemáme zkonstruovanou.

Rozdělíme si trii na *vrstvy* – první znaky slov budou první vrstva, druhé znaky budou tvořit druhou vrstvu atd., až i-té znaky slov budou tvořit i-tou vrstvu.

Zpětná hrana jistě povede do kratšího slova. Z i-té vrstvy tedy povede do vrstvy s nižším pořadovým číslem. Pokud tedy budeme zpětné hrany konstruovat po vrstvách, dojdeme kýženého výsledku.

Ještě zbývá otázka, jak konstruovat zpětné hrany efektivně, když je musíme vyrábět po vrstvách. Mohli bychom prostě vzít slovo, pro které hledáme zpětnou hranu, utrhnout mu první znak a vyhledat. Jenže to budeme dělat spoustu práce zbytečně.

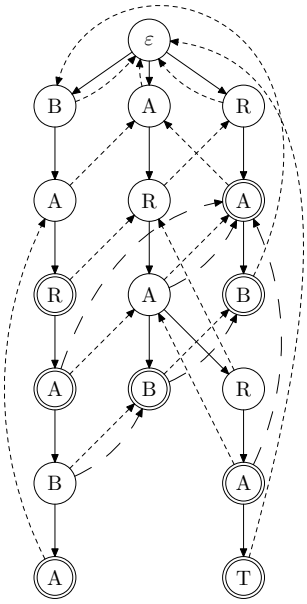


Například pro slovo `BARABA` bychom mohli vyhledávat `ARABA` v již zkonstruované části automatu, ale proč to dělat celé, když jsme při konstrukci předchozí vrstvy vyhledávali `ARAB` při konstrukci zpětné hrany pro `BARAB`?

Při konstrukci další zpětné hrany tedy najdeme akorát, kde jsme minule skončili, a odtamtud pokračujeme dál. Jak to najdeme? Z otce našeho vrcholu tam přece vede zpětná hrana. Takže můžeme postup shrnout do bodů:

1. $c =$ poslední znak slova (znak stavu P, pro který hledáme zpětnou hranu);
2. přesuneme se do otce;
3. přesuneme se po zpětné hraně;
4. dokud neexistuje syn se znakem c nebo nejsme v kořeni, přesouváme se po zpětných hranách;
5. pokud existuje syn se znakem c, natáhneme do něj zpětnou hranu z P, jinak ji natáhneme do kořene.

Automat je zkonstruován. Časová složitost konstrukce sestává z konstrukce trie v $\mathcal{O}(J \cdot |\Sigma|)$, resp. $\mathcal{O}(J \cdot \log |\Sigma|)$ (pokud použijeme binární strom ve vrcholech) a z předpočítání zpětných hran. Při předpočítávání uděláme nějaký konstantní počet operací pro každý vrchol (celkem tedy $\mathcal{O}(J)$) a také paralelně vyhledáváme všechny jehly z jehelníčku, jejichž vyhledání nás stojí $\mathcal{O}(J)$, resp. $\mathcal{O}(J \cdot \log |\Sigma|)$.



Tedy konstrukce trvá celkem $\mathcal{O}(J \cdot |\Sigma|)$, resp. $\mathcal{O}(J \cdot \log |\Sigma|)$, paměťová náročnost je stejná jako u trie – $\mathcal{O}(J \cdot |\Sigma|)$, resp. $\mathcal{O}(J)$, přidali jsme jen $\mathcal{O}(J)$ zpětných hran.

Zkusme tedy automatem projít text `BARABARARAT`. Ohlásí postupně nález slov `BAR`, `BARA`, `BARABA`, `BAR`, `BARA`, `ARARA` a `ARARAT`.

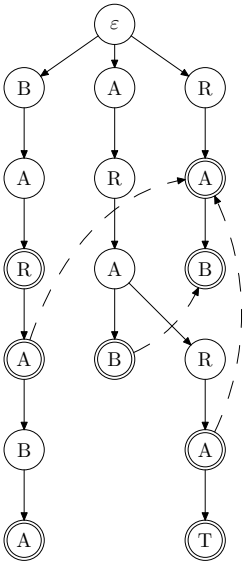
Nenalezl však všechno. Chybí mu např. `ARAB`, který začíná druhým znakem a končí pátým. Dále chybí několik výskytů `RA` a jeden `RAB`.

Když byl na pátém znaku, byl ve stavu `BARAB`, jehož suffixem je `ARAB`. Obecně na suffixy zapomínáme. Narozdíl od KMP, kde suffix aktuálního stavu nikdy nebyl jehla, tady jehlou být může.

V každém stavu bychom tedy měli projít veškeré suffixy a zkontrolovat je, jestli náhodou nejsou jehlami. Jak najdeme všechny suffixy? Projdeme postupně po zpětných hranách až do kořene. Má to jen jeden problém – je to pomalé.

Představme si například slovník obsahující `A` a `AAAA...A` (délky $J - 1$). Budeme-li jím vyhledávat v textu `AAAA...A` délky $S > J$, projdeme prakticky pro každý znak až $J - 1$ zpětných hran, čímž složitost naroste až na nepoužitelných $\mathcal{O}(S \cdot J)$.

Všimněme si však, že většinou zpětných hran jsme prošli úplně zbytečně. Předpočítáme si tedy *zkratky* – z vrcholu vede zkratka do nejdelšího jeho suffixu, který je jehlou. Na obrázku jsou vyznačeny dlouze čárkovanými šipkami.



Předpočítání zpětných hran časovou složitost konstrukce automatu jistě nezhorší, neboť vyžaduje v nejhorším případě projít všechny zpětné hrany ještě jednou.

Potřebujeme-li ohlásit všechny výskyty slov včetně pozice, kde se nacházejí, jsme hotovi. Výsledná časová složitost prohledávání v takovém případě bude $\mathcal{O}(S+O)$, resp. $\mathcal{O}(S \cdot \log|\Sigma|+O)$, kde O je velikost výstupu – počet výskytů všech slov.

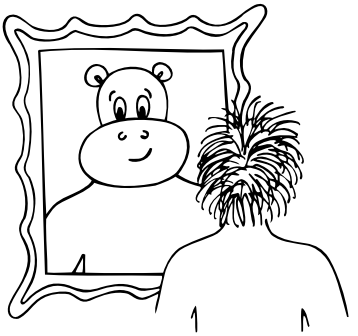
Celková časová složitost prohledávání včetně stavby automatu tedy bude $\mathcal{O}(O+S+J \cdot |\Sigma|)$, resp. $\mathcal{O}(O+(S+J) \cdot \log|\Sigma|)$.

Jak velký může být výstup? Obecně až S^2. Extrémně velký výstup je možné vygenerovat třeba slovníkem obsahujícím všechny prefixy slova `AAAA...A` délky S a senem taktéž `AAAA...A` délky S. Automat pak hlásí výskyt pro každé podslovo, kterých je řádově S^2.

Pokud nám stačí u každého slova jen počet výskytů, nemusíme zoufat – závislost na počtu výskytů umíme odstranit.

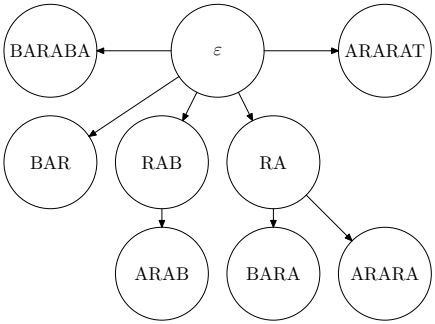
Použijeme trik – na každé pozici započítáme pouze nejdelší jehlu, která tam končí (u každé jehly si budeme udržovat čítač). Nebudeme tedy v každém kroku poskakovat po zkratkách až do aleluja, ale maximálně jednou. Díky tomu nám z časové složitosti zmizí velikost výstupu.

V našem příkladu se senem `BARABARARAT` tedy na konci budeme mít uloženo, že `ARAB` se vyskytnul 1×, `ARARA` 1×, `ARARAT` 1×, `BAR` 2×, `BARA` 2× a `BARABA` 1×. `RA` a `RAB` nemají hlášený žádný výskyt.



Nyní si zkonstruujeme strom jenom ze zkratek a pro každý vrchol spočítáme součet celého jeho podstromu.

Tedy po přepočtu bude mít `RA` tři výskyty a `RAB` jeden výskyt; celkový počet výskytů pak bude 12.



Poznámky

- Dalším krokem po KMP a Aho-Corasickové jsou konečné automaty a regulární výrazy, o kterých jsme měli seriál ve 23. ročníku.
- Není moc rozumné snažit se implementovat Aho-Corasickovou v rozumné době například při soutěži, pokud tento algoritmus nemáte opravdu pod kůží. Radši zkuste použít hešování, pokud budete něco takového potřebovat.

Cvičení

- Redukci o velikost výstupu můžeme provést i pro případ, kdy výstup nebudeme vypisovat, ale stačí nám mít jej uložený v paměti. Vymyslete vhodnou úpravu triku s čítačem.
- Zkuste si naimplementovat Aho-Corasickovou vlastnoručně ve svém oblíbeném jazyce, abyste si byli jisti, že doopravdy chápete všechny záludnosti tohoto algoritmu.

Martin Böhm, Jan Matějka, Martin Mareš a Petr Škoda

Vzorová řešení KSP

27-1-1 Zasedací pořádek

Piloti na letadlové lodi s vámi mohou být spokojeni – jak pokročilí řešitelé, tak i začátečníci přišli na správné rozsazení osob u stolu. Body jsme však museli strhávat za neúplné (nebo úplně chybějící) popisy druhé podúlohy, případně za drobnosti, jako například chybějící složitost.

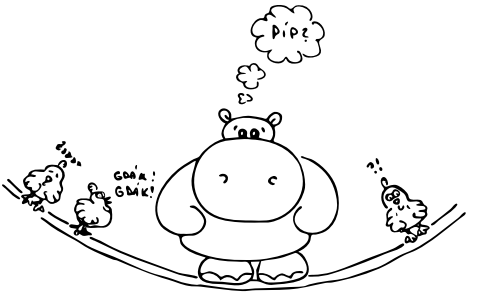
KSP

Jak budou piloti u stolu rozsazeni, záleží na tom, zda je jejich počet sudý, nebo lichý. Při lichém počtu budou všichni piloti sedět vedle sebe bez mezer, přičemž na centrální židli bude sedět prostřední z nich; jinak bude centrální židle prázdná a piloti budou sedět na obě strany od ní, tvoří tedy dvě souvislé řady délky $N/2$.

Na řešení nebylo těžké přijít simulací rozsazovacího popisu na papíře pro malá N, jen někteří z vás se ale pokusili dokázat, že výše zmíněná pravidla platí pro jakýkoliv počet pilotů. U takto jednoduché úložky jsme za to body nestrhávali, ale pamatujte si, že komplikovanější problémy důkaz vyžadují – všechno „pouhým okem“ neuvidíte!

řešení

Správnost zde dokážeme matematickou indukcí. Pro malá N experimentálně ověříme, že náš popis funguje. Teď musíme ukázat, že se „sudá“ řada při příchodu pilota změní na „lichou“ řadu a naopak: první případ je jednoduchý, příchozí usedne na centrální židli.



Komplikovanější je přechod z liché řady na sudou: v momentě, kdy přichází nový pilot, je centrální židle obsazená – oba se přesunou na židle vedle té centrální, a pokud jsou také obsazené, proces se nutně opakuje, dokud nedojdeme k volným židlím na kraji řady, kam si piloti sednou, a řadu tak rozšíří. Aniž bychom zatím přemýšleli nad piloty, kteří se musí přesunout směrem do středu (představme si, že zatím stojí), vypadá řada takto: `1000...0001` (`1` značí obsazenou, `0` volnou židli).

S piloty, kteří se musí ze svých původních míst posunout doprostřed, se řada změní na `1011...2...1101`. Ouha, vše je v pořádku, kromě prostřední

židle, o kterou mají zájem dva piloti – opět tedy musí nastat přesouvání, které ale dopadne podobně, jako to předchozí (jen se díky mezerám na koncích zkrátí délka řady, na které dochází k přesunům, o dva piloty, kteří sedí nyní na konci a jsou odděleni od ostatních prázdnou židlí).

Každou takovou iterací se krajní prázdná místa posunou směrem doprostřed. Přesouvání bude trvat tak dlouho, dokud se prázdná místa nesetkají na centrální židli, a řada tím pádem bude vypadat tak, jak jsme popisovali.

Nyní k řešení prvního úkolu: Kdybychom přesouvání jen simulovali podle zadaných pravidel, trvalo by nám to dlouho. Ale to není potřeba, díky výše dokázanému pozorování zvládneme vygenerovat řadu pilotů v lineárním čase a konstantní paměti, stačí sestavit jedničky a nuly podle pravidel.

Program, který ověří, zda je zadaný zasedací pořádek správně, si postupně načítá informace o obsazenosti židlí a kontroluje, zda se řada skládá buď z jedné liché řady jedniček (obsazených míst), nebo dvou stejně dlouhých a sudých řad jedniček, oddělených právě jednou nulou. To zvládneme v čase $\mathcal{O}(N)$, kde N je délka vstupu. Paměťová složitost je konstantní, protože nás nic nenutí řadu do paměti načítat.

Program (C): `http://ksp.mff.cuni.cz/viz/27-1-1.c`

Kuba Maroušek & Jenda Hadrava

27-1-2 Zbrojní sklad

Většina z řešení, která nám přišla, využívala principu hladového algoritmu a vždy vybírala palety s co největší nebezpečností, což byl správný postup. Takový výběr palety nám totiž umožní v příštím kroku dosáhnout nejvýš. Ukážeme si tedy, jak se to dá udělat efektivně.

Abychom mohli rychle hledat palety, na které dosáhneme, rozdělíme si je nejprve na malé a velké, a potom je vzestupně setřídíme podle výšky do dvou polí.

Poté si založíme dvě maximové haldy, odděleně pro malé a velké palety. V nich si budeme udržovat, jaké palety můžeme v každém kroku odebrat a která z nich má tu největší nebezpečnost. V každém kroku tedy vybereme příslušnou haldu pro velké nebo malé palety a odstraníme z ní paletu s nejvyšší nebezpečností. Nyní musíme do obou hald přidat všechny prvky, na které díky odebrání dosáhneme teď.

K tomu využijeme setřízená pole pro malé a velké palety. U každého si budeme pamatovat index prvku, který jsme přidali naposledy. Pokaždé, když budeme chtít přidat nově dosažitelné palety, jen zkontrolujeme všechny palety od tohoto indexu dále, dokud nenarazíme na paletu, která vyžaduje větší výšku, než je aktuálně dovolená. Všechny nalezené palety s výškou menší nebo rovnou dovolené budeme průběžně přidávat do příslušných hald.

Toto budeme opakovat, dokud jedna z hald nebude prázdná (tedy jsme právě narazili na omezení bezpečnostních předpisů), nebo nám v našich polích nedojdou palety (pak jsme všechny přidali do haldy, tedy na všechny dosáhneme, neboli můžeme všechny palety vyndat bez porušení předpisů).

Velká většina vás narazila na problém volby počáteční palety a mnoho z vás si závorku v zadání (velká, malá, velká) vyložilo tak, že budeme začínat velkou paletou. Po dlouhé diskusi jsme za toto nestrhávali body, ačkoli původně byla úloha myšlená tak, že nebude blíže určeno, kterou paletou začínáme. Řešením, u kterých nebylo očividné, zda si to autoři rozmysleli nebo ne, jsme to připsali do poznámky, která je s tímto vysvětlením snad již trochu jasnější.

KSP

Časová složitost algoritmu vychází jednak ze třídění, které nezvládneme obecně rychleji než v čase $\mathcal{O}(N \log N)$, a také z hald (musíme v nich probublat až N prvků, každý v čase $\mathcal{O}(\log N)$. Celková časová složitost je tedy $\mathcal{O}(N \log N)$. Paměti zabereme lineárně s velikostí vstupu (každou paletu uložíme do haldy a pole jednou).

Program (C++): `http://ksp.mff.cuni.cz/viz/27-1-2.cpp`

řešení

Štěpán Hojdar & Jan „Oggy“ Škoda

27-1-3 Letecké koridory

Nejprve bychom se chtěli omluvit za určité nepřesnosti v zadání samotné úlohy. Konkrétně nebylo jasné, zda se úloha má řešit pro ohodnocený či neohodnocený graf. Plný počet bodů jsme proto udělovali za optimální řešení libovolné z těchto variant.

Stejně dlouhé koridory

Většina z vás, kteří jste brali v úvahu neohodnocený graf, přišla na to, že prohledávání do šířky[49] je správný směr k optimálnímu řešení úlohy. Ve vašich řešeních se pak vyskytovaly nanejvýš drobné implementační chyby.

Graf tedy budeme procházet po hladinách od počátečního vrcholu. Budeme si udržovat frontu vrcholů, jež máme postupně zpracovat, do které na začátku umístíme pouze počáteční vrchol. Pro každý vrchol si navíc zapamatujeme délku nejkratší cesty do tohoto vrcholu (pokud jsme nějakou našli) a počet doposud nalezených nejkratších cest. Na začátku víme, že do počátečního vrcholu vede pouze jedna nejkratší cesta délky nula. O ostatních vrcholech nevíme nic.

Při zpracovávání každého vrcholu u vyjmutého z fronty postupně projdeme všechny jeho hrany. Cesta, která vede do u a poté po zkoumané hraně do vrcholu v, má délku rovnou délce nejkratší cesty do u zvětšené o jedničku. Tuto délku potenciálně nejkratší cesty porovnáme s délkou známé nejkratší cesty do v. Mohou nám nastat tři situace. Buď cestu do vrcholu v neznáme – v tom případě

[49] `http://ksp.mff.cuni.cz/viz/kucharky/grafy`

je nejkratší cesta do v právě ta námi uvažovaná (pokud by existovala kratší, přišli bychom na ni při zpracovávání předchůdce koncového vrcholu, který se na této nejkratší cestě nachází). Počet takovýchto cest je roven počtu nejkratších cest do u. Protože jsme tento vrchol ještě neviděli, přidáme jej do fronty vrcholů ke zpracování.

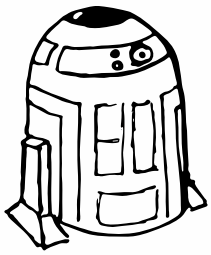
Pokud je délka naší navržené cesty stejně dlouhá jako známá délka nejkratší cesty do v, přišli jsme na další nejkratší cesty, které končí v tomto vrcholu. Zvětšíme tedy počet možných nejkratších cest, jež do vrcholu v vedou, o počet nejkratších cest vedoucích do vrcholu u. Do fronty vrchol v přidávat nebudeme, neboť jsme jej tam přidali, když jsme poprvé určili délku nejkratší cesty. A pokud je délka navržené cesty delší než známá délka nejkratší cesty do koncového vrcholu, nebudeme dělat nic, protože námi navržená cesta určitě není nejkratší.

V okamžiku, kdy frontu vyprázdníme, vypíšeme na výstup počet nejkratších cest vedoucích do cílového vrcholu a algoritmus ukončíme. Můžeme jej ukončit i v případě, když zpracovávané cesty jsou delší než nejkratší cesta do cílového vrcholu, v takovém případě totiž už nenalezneme žádnou další nejkratší cestu, nicméně toto nijak nezlepší asymptotickou časovou složitost. Výsledek je pak počet nalezených nejkratších cest do cílového vrcholu.

Správnost algoritmu můžeme ukázat indukcí. Na začátku vede pouze jedna cesta do počátečního vrcholu. V okamžiku, kdy vyjmeme nějaký vrchol z fronty, zpracovali jsme již všechny jeho předchůdce, po kterých vede nejkratší cesta (všechny vrcholy, které jsme dosud nezpracovali, leží na stejné hladině jako zkoumaný vrchol, nebo na hladině ještě vyšší, proto přes ně nemůže vést nejkratší cesta do zkoumaného vrcholu). Z čehož vyplývá, že pro každý vrchol vyjmutý z fronty nalezneme všechny nejkratší cesty. Z popisu algoritmu přímo plyne, že jsme žádnou cestu nezapočítali dvakrát, čímž je dokončen důkaz správnosti.

Každý vrchol přidáváme do fronty nanejvýš jednou a stejně tak každou hranu zkoumáme maximálně jednou. Časová složitost je tedy $\mathcal{O}(N+M)$, kde N je počet vrcholů a M je počet hran. U každého vrcholu si musíme pamatovat délku a počet nejkratších cest. Dále si musíme pamatovat všechny hrany. Dostáváme tedy i paměťovou složitost $\mathcal{O}(N+M)$.

Program (C++): `http://ksp.mff.cuni.cz/viz/27-1-3a.cpp`



Ohodnocené hrany

V případě, že pracujeme s ohodnocenými grafy, řešení bude podobné tomu předchozímu. S obyčejným průchodem do šířky si však nevystačíme. Místo něj budeme vycházet z Dijkstrova algoritmu,[50] který se prohledávání do šířky podobá, jen místo toho, aby bral nejbližší vrcholy podle počtu hran, je bere podle vzdálenosti. Hrany tedy nemůžeme uchovávat v obyčejné frontě, ale musíme je mít ve frontě prioritní, tedy haldě. V tomto řešení budeme pro jednoduchost uvažovat haldu binární. Další rozdíl mezi řešením pro neohodnocené grafy a tímto bude, že při přidání hrany do cesty nezvětšíme délku cesty o jedna, ale o celou délku dané hrany.

Při zpracování vrcholu u, který označíme jako *definitivní*, budeme v daných případech postupovat stejně jako při zpracovávání vrcholu v předchozím algoritmu. Tedy když neznáme cestu do sousedního vrcholu v, prohlásíme nově nalezenou cestu za nejkratší. Počet nejkratších cest vedoucích do vrcholu v tak bude stejný jako počet nejkratších cest do vrcholu u, navíc vrchol v přidáme do haldy. Když nalezneme stejně dlouhou cestu, upravíme počet nejkratších cest, a když nalezneme delší cestu, nic neděláme.

Nyní však už nemáme garantováno, že když jsme vrchol navštívili, nenalezneme ještě lepší cestu než tu, kterou jsme nalezli poprvé. Když takováto situace nastane, postupujeme stejně jako v případě, kdy bychom žádnou cestu do v neznali – zahodíme informace o počtu a délce nejkratších cest do vrcholu v, jako délku nejkratší cesty určíme nejkratší cestu do u zvětšenou o délku hrany a počet nejkratších cest do v bude stejný jako počet nejkratších cest do u. Vrchol do haldy ale přidávat nebudeme, neboť se v ní již nachází.

Nesmíme ale zapomenout změnu nejkratší cesty do haldy k vrcholu zaznamenat. Abychom daný vrchol v haldě rychle našli, budeme si navíc ještě bokem uchovávat pole, které nám říká, kde v haldě se daný vrchol nachází. Toto pole budeme aktualizovat při každé změně pozice nějakého vrcholu v haldě.

Změnou délky nejkratší cesty se nám však mění i pozice, kde by se vrchol v haldě měl nacházet. V případě, kdy tuto délku měníme, necháme vrchol probublat v haldě výš (nikdy nepotřebujeme posunovat vrchol níže, protože když měníme délku nejkratší cesty, vždy ji nahrazujeme cestou ještě kratší).

V haldě si v každém okamžiku uchováváme nanejvýš všechny vrcholy. Hloubka haldy je tedy maximálně $\log N$. Jedno vyjmutí vrcholu z haldy nám zabere s následným přerovnáním $\mathcal{O}(\log N)$ času. Každý vrchol odebereme nanejvýš jednou, celkově tedy $\mathcal{O}(N \log N)$. Každý vrchol do haldy nanejvýš jednou přidáme, jedno přidání zabere logaritmický čas, dohromady tedy opět $\mathcal{O}(N \log N)$. A konečně při změně nejkratší cesty do nějakého vrcholu musíme upravit pozici daného vrcholu v haldě. Pro každý vrchol to zabere $\mathcal{O}(\log N)$ času a těchto operací provádíme nanejvýš M. Tyto operace tedy zaberou $\mathcal{O}(M \log N)$ ča-

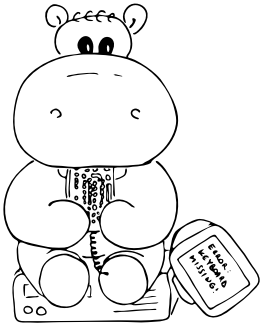
[50] `http://ksp.mff.cuni.cz/viz/kucharky/haldy-a-cesty`

su. Celý algoritmus bude mít časovou složitost $\mathcal{O}(M \log N + N \log N)$ neboli $\mathcal{O}((M+N) \log N)$, což je stejná složitost, jakou má běžný Dijkstrův algoritmus.

Pamatovat si potřebujeme u každého vrcholu délku nejkratší dosud nalezené cesty, počet takových cest a umístění v haldě. Dále si potřebujeme pamatovat všechny hrany. Paměťová složitost bude tedy $\mathcal{O}(N+M)$.

Program (C++): `http://ksp.mff.cuni.cz/viz/27-1-3b.cpp`

Lukáš Folwarczný & Dominik „Drecker“ Smrž



27-1-4 Head-up display

Displej má N pixelů, každé jeho nastavení má hodnotu, do které přispívá každý pixel rozdílem kontrastu oproti původnímu nastavení. Ze všech přípustných variant změny kontrastu vybíráme tu nejmenší. Na to se můžeme dívat jako na nejkratší cestu v grafu.

Jak takový graf vypadá? Má $N \cdot K$ vrcholů, pro každý pixel a každou možnou hodnotu kontrastu jeden. (Ono vlastně bude stačit N-krát nejvyšší hodnota kontrastu na vstupu, je snadno vidět, že v nejkratší cestě nikdy výš nepůjdeme, ale v nejhorším případě jich stejně bude $N \cdot K$.) Můžeme si představit, že jsou uspořádány v tabulce, vodorovně jich je N, svisle K.

Potřebujeme zajistit, aby se hodnoty kontrastu v sousedních sloupcích (neboli sousedních pixelech displeje) nelišily o více než o D. Přidáme proto do grafu hrany. Budou orientované, povedou zleva doprava vždy mezi vrcholy v sousedních sloupcích, a to nanejvýš o D řádků výš nebo níž. Tím pádem bude každá cesta z prvního sloupce do posledního korektní nastavení displeje. To nastavení přečteme snadno, napíšeme si za sebe navštívené řádky a ty přesně odpovídají hodnotám kontrastu jednotlivých pixelů.

Jaká bude hodnota cesty? Součet změn pixelů. Hranám dáme ohodnocení. Je-li pixel nastaven na hodnotu 10, pak všechny hrany vedoucí do 10. řádku tohoto sloupce budou mít hodnotu 0. Všechny hrany vedoucí do 11. a 9. řádku

budou mít hodnotu 1, protože hodnotu pixelu měníme o jedničku. Hrany do 12. a 8. řádku budou mít dvojku a tak dále. Hodnota cesty je samozřejmě součet vah hran.

Teď už jen najít nejkratší cestu z prvního sloupce do posledního. Pokud znáte pouze algoritmus, který hledá nejkratší cesty z jednoho vrcholu do dalšího, a ne z K vrcholů do některého z jiných K vrcholů, nezoufejte, upravíme si graf, abychom ho mohli použít. Přidáme si do grafu dva speciální vrcholy. Jeden bude startovní, z něho budou vycházet hrany do všech řádků prvního sloupce, ohodnoceny budou změnou kontrastu prvního pixelu oproti jeho původní hodnotě. Druhý speciální vrchol bude ten cílový, ze všech vrcholů posledního sloupce povedou hrany do něj. Ohodnoceny budou nulou.

Náš graf je orientovaný acyklický neboli po anglicku zkráceně DAG. Pro něj existuje rychlejší algoritmus na nejkratší cesty, než je Dijkstrův. Pro každý vrchol si budeme pamatovat délku zatím nejkratší nalezené cesty (na začátku bude na startu nula, ve zbytku grafu nekonečno) a vrchol, ze kterého jsme do aktuálního vrcholu přišli (podle toho pak zrekontruujeme nejkratší cestu). Značit je budeme jako $delka(u)$ a $odkud(u)$.

Budeme odebírat vrcholy v topologickém pořadí, (tak se to dělá u obecných DAGů), pro nás to znamená po sloupcích zleva doprava. Vezmeme vrchol a *zrelaxujeme* všechny jeho hrany. Tak se říká postupu, kdy zkusíme najít kratší cesty procházející danou hranou. Pokud si budeme váhu hrany značit jako $w(u, v)$, znamená to, že pro hranu (u, v) uděláme toto:

1. Pokud $cesta(u) + w(u, v) < cesta(v)$:
2. $\quad cesta(v) \leftarrow cesta(u) + w(u, v)$
3. $\quad odkud(v) \leftarrow u$

Řečeno slovy, známe již nejkratší cestu do u, a vede-li hrana z u do v, pak se do v mohu dostat za cenu rovnou délce cesty do u a váze hrany (u, v). Pokud je to lepší než dosud nejkratší nalezená cesta, tak si novou délku uložím a zapamatuji si, odkud jsem přišel.

A to je všechno. Už jenom vypíšeme délku nejkratší cesty a díky zpětným odkazům zrekonstruujeme průběh cesty (jen ho ještě musíme obrátit, dostaneme ho totiž pozpátku).

Určíme časovou složitost. Bereme do ruky každý vrchol, těch je $N \cdot K$, a každou hranu. Z každého vrcholu vede nanejvýš $2D + 1$ hran, takže jich je celkem $\mathcal{O}(NKD)$, a to je i celková časová složitost. Paměti nám ale stačí jen $\mathcal{O}(NK)$. Hrany totiž nemusíme mít nikde uloženy, snadno si spočítáme, která hrana existuje a která ne.

Program (C): `http://ksp.mff.cuni.cz/viz/27-1-4.c`

Program (Python): `http://ksp.mff.cuni.cz/viz/27-1-4.py`

Dominik Macháček

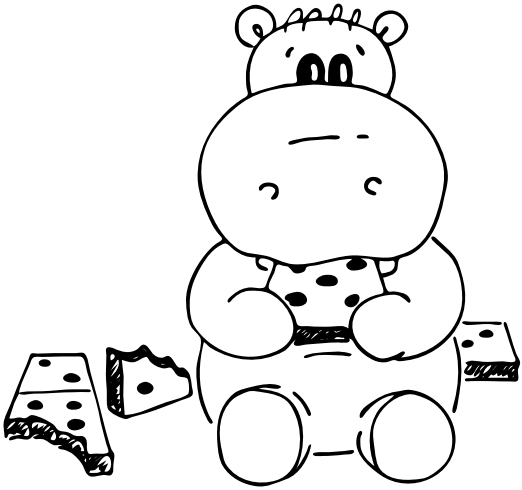
27-1-5 Napájení přístrojů

Tolik přístrojů a tolik různých požadavků na napětí… no naříkat na standardizaci můžeme jindy, teď pojďme vyřešit úlohu.

Vezměme si na chvíli požadavky na napájení jednotlivých přístrojů jako intervaly. Je jasné, že přístroje, jejichž intervaly se nepřekrývají, nemohou sdílet stejný zdroj napájení. A pokud budeme mít trojici intervalů, kde první se překrývá s druhým a druhý s třetím (ale první a třetí průnik nemají), můžeme sice vždy alespoň dva přístroje napájet z jednoho zdroje, ale třetí do toho nezařadíme. To je sice pěkné, ale co z toho?

Každý přístroj musíme z nějakého zdroje napájet. Když si vezmeme naše intervaly a podíváme se na interval (a, b), který končí nejdříve (jehož konec má nejnižší hodnotu), tak určitě musíme nějaký napájecí zdroj umístit mezi a a b, jinak bychom tento přístroj nepokryli. No ale má smysl nastavovat první napájení na menší hodnotu než b?

Protože všechny další konce intervalů jsou až za b, nemá. Nastavíme tedy první zdroj na b, připojíme na něj všechny přístroje, kterým vyhovuje, a odebereme jejich intervaly. Pak se podíváme na zbylé intervaly a budeme tento *hladový* postup opakovat, dokud nezapojíme všechny přístroje.



Teď je potřeba si rozmyslet dvě věci. První z nich je, jak rychle algoritmus běhá. Pokud už dostaneme intervaly setříděné podle konců, stačí nám je v čase $\mathcal{O}(N)$ od nejmenšího projít a postupně je označovat za zapojené, pokud je setříděné mít nebudeme, zvedne se nám čas tříděním na $\mathcal{O}(N \log N)$.

Druhou a zajímavější otázkou je, zdali to skutečně funguje. To by chtělo dokázat. Postupným zapojováním nějakých přístrojů na vyhovující zdroje vytvoříme nějakých K množin intervalů (tvořených z původních povolených intervalů napájení přístrojů obsažených v každé množině). Každou množinku si můžeme sjednocením převést na jeden nový interval, čímž nám vznikne K různých intervalů.

No ale protože do množinky pokaždé zapojíme všechny vyhovující přístroje, jsou jednotlivé vzniklé intervaly navzájem disjunktní. A pokrýt K disjunktních intervalů určitě nejde pomocí méně než K zdrojů, tím jsme dokázali optimálnost našeho řešení.

Program (C): `http://ksp.mff.cuni.cz/viz/27-1-5.c`

Program (Python): `http://ksp.mff.cuni.cz/viz/27-1-5.py`

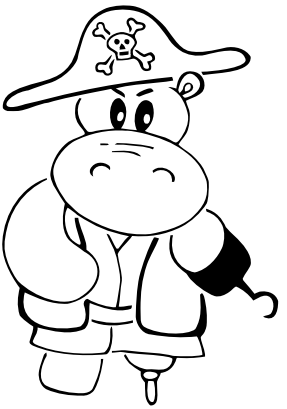
Jirka Setnička & Dominik Macháček

27-1-6 Přistávací světla

Dvojbarevná verze

Začneme jednodušší verzí úlohy, která používá dvě barvy (říkejme jim třeba **X** a **Y**) a chce po nás najít nejdelší „bílý“ úsek, tedy takový, v němž je obou barev stejně.

Úlohu převedeme na podobný problém s čísly: barvu **X** přepíšeme na $+1$ a **Y** na -1. Bílé úseky jsou nyní přesně ty se součtem 0. Poznáme je pomocí prefixových součtů z kuchařky: označíme p_i součet čísel na prvních i pozicích ($p_0 = 0$) a kdykoliv $p_i = p_j$, znamená to, že na pozicích $i + 1$ až j leží bílý úsek.



Chceme tedy v posloupnosti prefixových součtů najít dvě stejné hodnoty, které od sebe leží co nejdále. To provedeme snadno: budeme procházet vstup zleva doprava a průběžně si počítat prefixové součty. Pro každou možnou hodnotu prefixového součtu ($-n$ až n) si budeme v nějakém poli pamatovat, jestli jsme ji už viděli, a pokud ano, tak kde poprvé. Kdykoliv pak narazíme na hodnotu prefixového součtu, kterou jsme už viděli, spočítáme vzdálenost od prvního výskytu – to je délka nejdelšího bílého úseku končícího aktuálním prvkem – a započítáme ji do průběžného maxima.

Časová složitost bude lineární: $\mathcal{O}(n)$ na inicializaci pomocného pole a pak $\mathcal{O}(1)$ na zpracování každého prvku. Paměti nám také stačí $\mathcal{O}(n)$.

Program (C): `http://ksp.mff.cuni.cz/viz/27-1-6a.c`

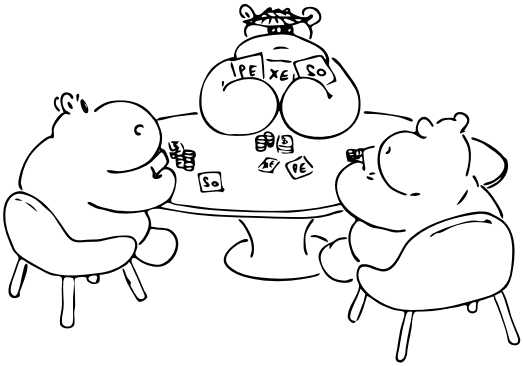
Od dvou barev ke třem

„Plnotučná“ verze úlohy pracuje se třemi barvami (třeba **R**, **G** a **B**) a chce po nás úseky, kde je všech třech stejně.

Pomůžeme si drobným úskokem: bílý úsek je ten, ve kterém je stejně **R** jako **G** a současně **G** jako **B**. To nám dává dvě instance dvojbarevné verze, které chceme vyřešit současně.

Vstup proto budeme překládat na posloupnost dvojic čísel (dvojsložkových vektorů): **R** bude $(1, 0)$, **G** bude $(-1, 1)$ a konečně **B** přeložíme na $(0, -1)$. Bílé úseky teď budou ty, které mají součet $(0, 0)$, přičemž sčítáme zvlášť první složku všech vektorů a zvlášť druhou.

Opět do práce zapřáhneme prefixové součty – co na tom, že teď to nebudou čísla, ale dvojice čísel, fungovat budou stejně.



Jen už nemůžeme hodnotami prefixových součtů indexovat pole: muselo by být dvojrozměrné, takže by jeho inicializace trvala $\mathcal{O}(n^2)$ a zkazila nám jinak lineární složitost.

Místo pole si proto pořídíme chytřejší datovou strukturu, třeba vyvážený vyhledávací strom (dvojice porovnáváme lexikograficky). Dvojic je ve stromu nejvýše n, takže každá operace se stromem trvá $\mathcal{O}(\log n)$, a celý algoritmus tudíž potřebuje $\mathcal{O}(n \log n)$ času a $\mathcal{O}(n)$ prostoru.

Kdybychom si místo pole pořídili hešovací tabulku, bude jedna operace s dvojicemi trvat průměrně $\mathcal{O}(1)$, takže celý algoritmus poběží v čase průměrně $\mathcal{O}(n)$ a prostoru $\mathcal{O}(n)$.

Skutečně lineární řešení

Existuje ovšem řešení, které má lineární složitost i v nejhorším případě. Inspirujeme se přechozí úvahou o prefixových součtech, ale stejné dvojice nebudeme hledat on-line pomocí datové struktury, nýbrž dávkově tříděním.

Vytvoříme posloupnosti trojic: kdykoliv má i-tý prefixový součet hodnotu (a_i, b_i), přidáme trojici (a_i, b_i, i). Tyto trojice pak setřídíme lexikograficky, čímž

se nám dostanou k sobě všechny výskyty stejných prefixových součtů. Snadno pak mezi nimi najdeme ty nejvzdálenější.

No dobrá, třídění trvá $\mathcal{O}(n \log n)$ a takhle rychlý algoritmus jsme už měli. Nenechte se mýlit, třídit jde i rychleji. V našem případě jsou složky trojic malá celá čísla, takže zafunguje přihrádkové třídění, které to zvládne v lineárním čase. Pokud ho ještě neznáte, je nejvyšší čas podívat se do kuchařky o třídění.[51]

Zde ho popíšeme aspoň stručně: pořídíme si přihrádky indexované od $-n$ do n (rozsah možných hodnot složek). Nejprve trojice rozházíme do přihrádek podle poslední složky a zase je vysbíráme (v pořadí od nejnižší přihrádky k nejvyšší). Pak provedeme totéž podle druhé složky a nakonec ještě jednou totéž podle první. Přitom u trojic, které padnou do téže přihrádky, stále zachováváme pořadí z předchozích průchodů. Proto nám nakonec vyjde přesně lexikografické pořadí.

Pokaždé strávíme čas $\mathcal{O}(n)$ rozhazováním trojic do přihrádek a $\mathcal{O}(n)$ jejich vysbíráváním. To provedeme celkem třikrát a pak ještě sekvenčně projdeme setříděné trojice, což nám všechno zabere $\mathcal{O}(n)$ času. Paměti nám stále postačí $\mathcal{O}(n)$ buněk.

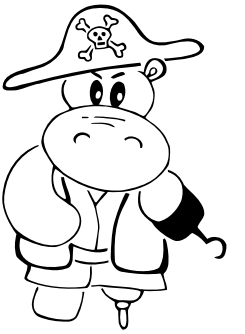
Dodejme ještě, že kdybychom měli b barev namísto tří, použijeme b-tice a vše nám bude trvat $\mathcal{O}(bn)$ s pamětí $\mathcal{O}(bn)$. Též bychom k barvám mohli přidat i „anti-barvy“, které se chovají opačně – tím se bychom se přiblížili k prapůvodní fyzikální inspiraci úlohy teorií kvarků.

Program (C): `http://ksp.mff.cuni.cz/viz/27-1-6b.c`

Poznámka o rekurzi

Existuje i jiné lineární řešení, také založené na dvojím použití dvojbarevné verze úlohy.

Nejprve řešíme úlohu pro **R** a **G**. Tím nám vzniknou nějaké prefixové součty a pro každé dvě pozice i, j se stejným prefixovým součtem chceme vyřešit úlohu pro **G** a **B** mezi těmito pozicemi.



Rozházíme si tedy pozice ve vstupu do přihrádek podle hodnoty prefixového součtu pro **R** a **G**. Pro každou přihrádku pak spustíme novou instanci úlohy pro **G** a **B**; nebudeme-li pořád dokola inicializovat datové struktury a počítat prefixové součty, zvládneme to lineárně s množstvím pozic v této přihrádce. V součtu přes všechny přihrádky tedy $\mathcal{O}(n)$.

Implementační detaily nejsou triviální, ale pokud se jimi prokoušeme, uvědomíme si, že jsme právě popsali přihrádkové třídění „naruby“, tedy od nejvyššího řádu k nejnižšímu. Nic nového pod sluncem.

[51] `http://ksp.mff.cuni.cz/viz/kucharky/trideni`

Poznámka o komplexních číslech

Ještě zmíníme jeden mírně bláznivý způsob, jak řešit tříbarevnou úlohu. Je zajímavý tím, že pro 4 barvy by už nefungoval a že v něm hrají hlavní roli komplexní čísla.

Pořídíme si komplexní třetí odmocniny z jedničky, totiž čísla $c_1 = 1$, $c_2 = -1/2 + i \cdot \sqrt{3}/2$ a $c_3 = -1/2 - i \cdot \sqrt{3}/2$. Tato tři čísla jsou rovnoměrně rozmístěna po jednotkové kružnici po násobcích 120°.

Nyní přeložíme **R**, **G** a **B** na čísla c_1, c_2 a c_3 a nahlédneme, že bílé úseky jsou přesně ty se součtem 0. (Rovnost imaginárních částí vynucuje, že všech **G** je stejně jako **B**. Z rovnosti reálných částí pak dostaneme, že **R** je stejně jako **G** i **B**.)

Nabízí se opět použít prefixové součty. Jenže radost nám poněkud kazí, že musíme počítat s divokými iracionálními čísly. Proto ještě změníme měřítko reálné osy vynásobením 2 a měřítko imaginární osy vydělením $\sqrt{3}/2$. Tím jsme nezměnili, které úseky mají nulový součet (reálná a imaginární složka se při sčítání neovlivňují), a dostali jsme samá celá čísla: $c'_1 = 2$, $c'_2 = -1+i$, $c'_3 = -1-i$.

Tato úvaha dále vede na podobná lineární řešení.

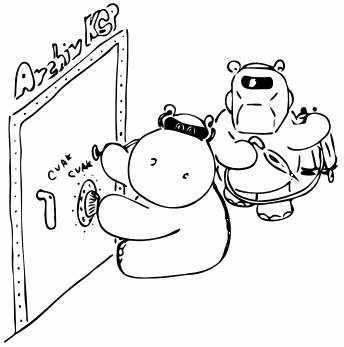
Martin „Medvěd“ Mareš

27-1-7 Učíme se s UNIXem

Jsme velmi rádi, že jste se UNIXu nezalekli a přišlo nám tolik vašich řešení. Většina úkolů byla jednoduchá, ale sem tam se objevily nějaké zrádnosti nebo zbytečně obtížné konstrukce.

Často bylo bodování obtížné, ale pokoušeli jsme se hodnotit praktičnost a jednoduchost vašich řešení a podle nich přidělovat nějaké zlomky bodů.

Pokusíme se tedy v autorském řešení ukázat, jaký je podle nás nejlepší přístup ke všem úkolům.



Úkol 1 – složky

Zde bylo správné podívat se do manuálových stránek příkazů `mkdir` a `rm`. Když se pokusíte pomocí prvního zmíněného vytvořit složku ve složce, která ještě neexistuje, vynadá vám. Pokud ale zavoláte `mkdir -p`, přepínač `-p` způsobí vytvoření všech složek po cestě.

Při mazání je možné mazat také pomocí `rmdir -p`, ale předtím je nutné vymazat vytvořený soubor. Pokud ale použijeme `rm -r` (`-r` jako zkratka za *rekurzivní*), můžeme rovnou vymazat všechno.

```
mkdir -p ~/a/b/c/d
touch ~/a/b/c/d/test
rm -r ~/a
```

Úkol 2 – head a tail

Zde bylo řešení velmi jednoduché. Rychlým pohledem do manuálových stránek jde zjistit, že příkazy `head` i `tail` mají oba přepínač `-n K`, který vypíše prvních, respektive posledních K řádků ze souboru. Jak ale vypsat všechno až na prvních K řádků?

Pečlivějším pročtením manuálových stránek se dá přijít na to, že zavoláním `tail -n+K` (všimněte si, že ani není potřeba mezera mezi přepínačem a jeho hodnotou) vypíše celý soubor začínající od K-tého řádku. My ale chceme K řádků vynechat, tedy chceme začít až od $(K+1)$-ního:

```
head -n15
tail -n15
tail -n+16
```

Úkol 3 – escapování

Problém zapisování podivných znaků se dá vyřešit dvěma způsoby: *escapováním* speciálních znaků nebo uzavřením do uvozovek. Při escapování musíme zrušit význam znaků jako závorka, mezera nebo otazník, při zavírání do uvozovek nám stačí dávat si pozor na jiné uvozovky. Správným řešením tedy mohlo být:

```
rm a\?c
rmdir "adresar[567]"
```

Úkol 4 – wildcardy

Toto byl první trochu více tvůrčí úkol a jsme rádi, že nám na něj přišla spousta různých elegantních řešení. Nakonec jsme se rozhodli hodnotit vaše wildcardové výrazy podle toho, jak efektivně jste použili dostupné prostředky.

Asi nejkratší rozumný výraz, který odpovídá zadaným pravidlům je tento:

```
*{[0-9]*[0-9],[a-zA-Z]*[a-zA-Z]}*/*.{jp{e,}g,gif}
```

Část `{[0-9]*[0-9],[a-zA-Z]*[a-zA-Z]}` říká to, že se někde v názvu musí nacházet buď dvě čísla, nebo dvě písmena oddělená libovolnými jinými znaky

mezi sebou. K nim přidáme libovolné množství znaků před a za (obalení hvězdičkami) a tím máme část složky hotovou.

Název souboru může být libovolný a pak následuje přípona `{jp{e,}g,gif}`. Všimněte si hlavně elegantně použité vnořené složené závorky: ta nám říká, že se zde může nacházet buď `e`, nebo nic.

Přišlo nám i několik málo ještě šílenějších konstrukcí, ale výraz uvedený výše považujeme asi za optimální vzhledem k jeho složitosti a plně nám dostačoval.

Úkol 5 – datum v souboru

Hlavní věcí, na které byl tento úkol postaven, bylo přesměrování výstupu. Použití jedné šipky vždy soubor přepisuje celý, ale použití dvou šipek pouze přidává na konec. Správným řešením tedy bylo:

```
echo "Dnes je:" > datum
date >> datum
```

Poznámka: U příkazu `echo` by uvozovky ani nebyly potřeba, `echo` opíše na výstup všechny své parametry. Ale přijde nám, že s použitím uvozovek je příkaz přehlednější.

Úkol 6 – počítání složek a adresářů

V tomto případě stačilo přesměrovat výstup z příkazu `ls` do příkazu `wc` a tím spočítat řádky:

```
ls | wc -l
```

Část z vás možná zmátlo to, že `ls` vypisuje všechny soubory na jeden řádek a používali jste s ním přepínač `-1`. To ale není potřeba, jelikož `ls` je trochu speciální příkaz. Při spuštění si totiž „osahá“ svoje okolí a zjistí, jestli je spuštěný v rouře, nebo zobrazuje výstup uživateli. V tom druhém případě se pokusí svůj výstup zkrátit a vypisuje soubory na jednu řádku, v tom prvním je ale vypíše všechny normálně pod sebou.

Častou chybou v tomto úkolu bylo použití přepínače `-w` u `wc` namísto přepínače `-l`. Na první pohled se může zdát, že oba vracejí správný výsledek, ale to jen do chvíle, kdy se objeví soubor obsahující v názvu mezeru. V tom případě `-w` počítající slova již dá špatný výsledek.

Úkol 7 – kombinace head a tail

Hlavní kouzlo spočívalo v tom správně si spočítat řádky a pak nakombinovat příkaz. Jedenáctou až třicátou řádku získáme pomocí některého z následujících příkazů:

```
head -n30 < soubor.txt | tail -n+11 | wc -w
tail -n+11 < soubor.txt | head -n20 | wc -w
```

Zde se ale také objevila jedna z nejpodivnějších věcí na celém řešení, zhruba dvě pětiny z vás totiž místo třicáté řádky přečetli v zadání řádku třináctou a podle toho jste pak psali příkaz. Přivedlo nás to k zajímavému lingvistickému rozhovoru o čtení velmi podobných slov. :-)

Úkol 8 – velikosti souborů

V posledním úkolu jsme jasně nespecifikovali, že chceme jen soubory v aktuálním adresáři a nemusíte vypisovat rekurzivně všechny soubory v podsložkách (na což stejně v tomto dílu seriálu ještě ani nebyly ukázány vhodné příkazy a konstrukce). Omlouváme se za to.

Těm, kteří ale pro řešení použili nějaké složitější ještě nevysvětlené příkazy a povedlo se jim správně vyřešit obtížnější verzi, jsme také dávali plný počet bodů. Pokud jste ale zbytečně komplikované příkazy použili na řešení jen lehčí verze, o nějakou tu desetinku bodu jste přišli.

Nejtěžší na celém úkolu bylo zkonstruovat ten správný wildcard, kterému by vyhovovaly i skryté soubory (wildcardová hvězdička se totiž normálně na skryté soubory uvedené tečkou *nerozexpanduje*). Šlo to udělat například jako `{.,}*[0-9]*` (tuto konstrukci se složenou závorkou jsme mohli potkat už ve čtvrtém úkolu).

Na spočítání velikosti souborů pak šel využít příkaz `wc -c` a pokud jsme chtěli jen celkový součet (který `wc -c` vypisuje na poslední řádek), mohla celá konstrukce vypadat třeba takto:

```
wc -c {.,}*[0-9]* | tail -n1
```

Doufáme, že UNIXu zůstanete věrní i v další sérii.

Jirka Setnička & Marek Dobranský

27-2-1 Systém Patriot

Toto je úloha, která má více než jedno správné řešení. Předvedeme si to, které vymyslela většina řešitelů. Jiná řešení však nemusí být špatně.

Střely si nastrkáme do dvou hald, A a B. Halda A bude maximová (na vrcholu sedí maximum), B minimová. Dále, všechny prvky v A budou nejvýše tak velké, jako libovolný prvek B (tedy formálněji, $\forall a \in A, b \in B; a \leq b$). A počty prvků v haldách se budou lišit maximálně o 1.

K čemu nám takové komplikované podmínky budou? Celkem jednoduše nahlédneme, že medián se nachází na vrcholu jedné z hald a z velikostí hald lze vždy určit, na vrcholu které.

Nyní si rozmyslíme, jak s touto datovou strukturou pracovat. Když chceme střelu přidat, porovnáme ji s aktuálním mediánem a určíme, do které haldy ji přidáme (pokud je větší než medián, jde do B, pokud menší, tak do A, u stejných prvků je to jedno). Poté zjistíme, jestli nám tato halda příliš nevyrostla. Pokud by byla o 2 prvky větší, tak její vrchol přestěhujeme do druhé haldy, aby se to vyrovnalo.

Jak vystřelit střelu? Medián najít umíme, takže jej stačí odebrat a poté opět zkontrolovat, že haldy mají dostatečně podobnou velikost.

V každé operaci provádíme konstantně mnoho operací vložení a odebrání z hald, tedy jedna operace nám zabere $\mathcal{O}(\log n)$ – takovou složitost má operace s haldou. Paměť nám bude stačit lineární, opět proto, že halda potřebuje lineární množství paměti s množstvím prvků v ní uložených.

Chceme ještě nahlédnout, že to opravdu funguje. Protože nalevo i napravo od hranice mezi haldami je stejné množství prvků a na vrcholu je ten prvek „k hranici“, opravdu je medián jedním z vrcholů. Nakonec si všimneme, že pravidla z druhého odstavce neporušíme ani jednou operací, a tedy medián neztratíme.

Program (C): `http://ksp.mff.cuni.cz/viz/27-2-1.c`

Michal „vorner“ Vaner

27-2-2 Prohledávání budov

Než pošleme záchranné týmy prodírat se hromadami trosek v budovách, nejprve učiníme několik pozorování.

Tím prvním je, že nám nezáleží na pořadí budov. Můžeme si je přeuspořádat, jak chceme, a stejně to na počtu nutných týmů nic nezmění. Pro jednoduchost si tedy budovy seřadíme podle velikosti od nejmenší po největší.

Druhým důležitým pozorováním je to, že budovy začínají všechny u země. Na tom není asi nic objevného, ale díky tomu platí, že u země jsou patra budov zastoupena nejhustěji a směrem nahoru jich jen ubývá (jak jednotlivé budovy postupně končí).

Pokud tedy chceme nějaká patra prohledat horizontálně, tak to určitě bude nějaký úsek pater začínajících u země a končící v nějakém patře K. Budovy přesahující nad toto patro pak prohledáme vertikálně. Kdyby v nějakém optimálním řešení netvořila horizontálně prohledávaná patra souvislý úsek od země, tak takové řešení můžeme bez jakéhokoliv zhoršení převést na stav se souvislým úsekem.

Našemu řešení tedy stačí jen najít toto číslo K. Všech N budov určitě umíme prohledat pomocí N týmů (do každé budovy jeden vertikálně), takže začneme s $K = 0$ a budeme ho zkoušet zvedat. Jako V si označíme počet vertikálně prohledávaných budov (na počátku tedy $V = N$).

KSP

Budeme naší hranicí postupně skákat budovu po budově nahoru a budeme si pamatovat, kdy byl počet týmů $(V + K)$ nejmenší. Vždy, když odebereme budovu, snížíme V o jedna a zvedneme K na výšku této budovy.

Hlavní část řešení zvládneme v lineárním čase vůči velikosti vstupu, ale kvůli úvodnímu třídění máme časovou složitost bohužel $\mathcal{O}(N \log N)$. Neumíme to ještě zrychlit?

řešení

Stačí si přidat další pozorování: Budovy vyšší než N určitě prohledáme vertikálně, protože K nikdy nemá smysl zvedat nad N. Můžeme si tedy tyto budovy na začátku v lineárním čase vyřadit a uvažovat jen budovy nižší než N. A ty již umíme setřídit přihrádkovým tříděním v lineárním čase. Dostali jsme se tedy k paměťové i časové složitosti $\mathcal{O}(N)$.

Program (Python 3): `http://ksp.mff.cuni.cz/viz/27-2-2.py`

Jirka Setnička

27-2-3 Průjezd jeřábu

Tato CodExovka se ukázala býti těžší než předchozí (soudě dle vámi získaných bodů). Nicméně podobně jako předchozí CodExovka, tak i tato má celkem jednoduché řešení. Ukážeme si dva možné přístupy.

Spojování komponent

Budeme si sestavovat mapu oblastí, mezi kterými křižovatkami se můžeme s jak vysokým jeřábem pohybovat. Přidání do mapy ještě nepoužité silnice, po které se dá jet s nejvyšším jeřábem, nemůže nic pokazit. Principem prvního řešení je toto pravidlo opakovaně použít na zbylé silnice, dokud nespojíme základnu se skladištěm.

Na začátku představuje každá křižovatka osamocenou komponentu, žádné silnice jsme zatím ještě nepoužili. Všechny silnice si na začátku setřídíme a poté je po jedné odebíráme a propojujeme dvě komponenty, které spojuje tato silnice. Pokračujeme, dokud nezískáme komponentu, ve které se nachází jak základna, tak sklad. Pak vypíšeme, že nejvyšší jeřáb může mít výšku odpovídající hodnotě

na poslední přidávané silnici. Pak pro nalezení cesty stačí projít ze základny tu jednu komponentu do hloubky, dokud nenarazíme na sklad.

Řešení má tedy časovou složitost $\mathcal{O}(M \log M)$, protože tak dlouho nám trvá setřídit všechny silnice, kterých je M. A pokud použijeme chytře implementovanou datovou strukturu, která umí rychle najít, zda dva vrcholy jsou v jedné komponentě, tak si tím časovou složitost nezhoršíme, protože celkem všechny testy a spojení zaberou $\mathcal{O}(M \log N)$.

Pro podrobnosti se podívejte na datovou strukturu Disjoint-Find-Union (DFU),[52] popsanou v kuchařce druhé série. Projití do hloubky nám zabere času nejvýše $\mathcal{O}(N + M)$, tedy opravdu celková časová složitost je $\mathcal{O}(M \log M)$. Paměťová složitost je $\mathcal{O}(N + M)$, protože máme uložený celý vstup a DFU přidá ke každé silnici jen konstantně mnoho informací.

Program (C++) – Disjoint-Find-Union:
`http://ksp.mff.cuni.cz/viz/27-2-3-dfu.cpp`

Inspirace Dijkstrovým algoritmem

Jistě si říkáte, že předchozí řešení asi nebude nejlepší, protože se nám může mnohokrát stát, že spojujeme křižovatky, které nakonec stejně nebudou v té komponentě, kde je základna a sklad. A máte pravdu, lze udělat o trošku lepší řešení.

Začneme silnice procházet do šířky ze základny. Přitom budeme rozšiřovat oblast, do které se umíme dostat. Vždy použijeme silnici vedoucí z prozkoumané oblasti někam dál, po které může jet nejvyšší jeřáb, což je v podstatě princip Dijkstrova algoritmu.[53] Předchozí krok opakujeme až do chvíle, kdy se dostaneme do skladu, nebo nám dojdou silnice. Pak stačí zrekonstruovat cestu, po kterých silnicích jsme šli, protože víme, že lepší cesta nemůže existovat. Důkaz platnosti tohoto pozorování necháváme na rozmyšlení si čtenářům.

Pro každou křižovatku si tak budeme pamatovat hodnotu, kterou si označíme `value`. Bude představovat to, jakým nejvyšším jeřábem jsme schopní dojet na tuto křižovatku. U každé silnice víme, jaký nejvyšší jeřáb jí může projet. Tuto hodnotu si označíme jako `h`. A nakonec v `lastValue` bude `value` křižovatky, ze které jsme přijeli.

Pak při každém použití některé silnice nastavíme na křižovatku, do které dojedeme, hodnoty takto:

```
value = max(value, min(h, lastValue))
```

Abychom zbytečně nemuseli zjišťovat, jestli jsme někdy na dané křižovatce již byli, tak na začátku všechny křižovatky mají `value = 0`, jen křižovatka základny (start) má nastaveno `value =` ∞ (v programu nějaké dostatečně vysoké číslo).

[52] `http://ksp.mff.cuni.cz/viz/kucharky/minimalni-kostra`
[53] `http://ksp.mff.cuni.cz/viz/kucharky/halda-a-cesty`

Toto řešení má časovou složitost $\mathcal{O}(M \log N)$, pokud aktualizujeme hodnoty v haldě. Pokud pouze vkládáme, tak je časová složitost $\mathcal{O}(M \log M)$, což je až $\mathcal{O}(M \log N^2) = \mathcal{O}(2 \cdot M \log N)$. Tedy na soutěžích je asi rozumné nezdržovat se programováním aktualizování hodnot v haldě, protože je to asi nejsložitější operace, navíc musíte vědět, kde se který vrchol v haldě nachází. V praxi je dobré aktualizování implementovat, protože budeme mít zhruba dvakrát rychlejší algoritmus a navíc zabereme méně paměti. Asymptoticky si ale nepomůžeme, protože musíme mít uložený celý vstup. To vede na paměťovou složitost $\mathcal{O}(N + M)$.

KSP

Program (C) – Dijkstra s aktualizovanou haldou:
`http://ksp.mff.cuni.cz/viz/27-2-3-dijkstra.c`

Vojta Sejkora

27-2-4 Stahování map

řešení

Lehčí varianta

Nejprve se zastavme u lehčí varianty úlohy, kde máme jenom $N = 3$ mapy o rozměrech $R \times S$, kde $R, S \leq 20$, a číslo W, které udává režii přenosu mapy diferencí.

Pojmenujme si mapy A, B a C. Vytvoříme si z nich vrcholy grafu. Natáhneme mezi nimi hrany, pokud $D_{AB}W < RS$, kde D_{AB} je počet rozdílných políček mezi mapami A a B. Všimněme si, že $D_{AB} = D_{BA}$, hrany jsou tedy neorientované.

Hrana mezi A a B znamená, že přenést mapu B diferencí od A se vyplatí, protože to bude stát méně, než kdybychom mapu B přenášeli samostatně. Hranám přidáme ohodnocení, to bude právě $D_{AB}W$.

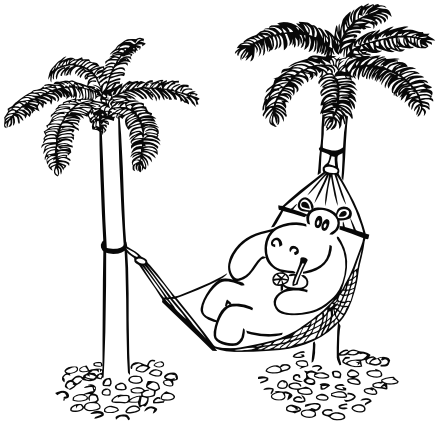
Kolik hran v grafu teď můžeme mít?

- 0 – v tom případě všechny mapy stáhneme přímo ze serveru.
- 1 – jednu mapu, ze které vede hrana, stáhneme přímo, druhou diferencí od ní. Nezáleží na tom, která bude která, obojí nás bude stát stejně. Třetí mapu nám nezbývá než přenést přímo.
- 2 – graf má tedy tvar cesty na třech vrcholech. Vybereme jednu mapu, například tu prostřední, tu stáhneme přímo, zbylé diferencí od ní. Všimněme si, že kdybychom krajní mapu stáhli přímo, od ní diferencí prostřední, a od ní diferencí tu poslední, také nás to bude stát stejně.
- 3 – Nejradši bychom všechny mapy přenášeli diferencí, tak to ale nejde. Nejméně jednu mapu musíme přenést ze serveru, a zbylé dvě od ní. Aby nás přenášení stálo co nejméně, použijeme dvě nejlehčí hrany. Tu nejtěžší z grafu vyjmeme. Tímto jsme situaci převedli na známý případ se dvěma hranami.

Toto by k vyřešení lehčí varianty úplně stačilo. Podívejme se ale na graf ještě trochu jinak. Představme si, že máme čtyři vrcholy, mapy A, B, C a server. Ze serveru vede do každé mapy hrana o váze RS, všechny ostatní hrany mají váhu danou počtem rozdílných políček krát W.

Z tohoto grafu potřebujeme vybrat tři hrany, protože celkem budeme provádět tři přenosy. Vybírat ale nesmíme úplně libovolně. Každý vrchol musí být připojen nějakou hranou k ostatním, protože kdyby nebyl, zůstane nějaká mapa nestažená nebo stáhneme všechny diferencí, ale nic ze serveru jako první.

Vrcholy jsou čtyři, hrany jsou tři, výsledný podgraf musí být souvislý a neobsahovat cyklus, to znamená, že to bude strom. Navíc hledáme nejlehčí strom, takový, který má součet hran nejmenší možný. Takovýto strom se nazývá minimální kostra.



Generické řešení

A je to. Máme řešení i pro plnou variantu úlohy, kde je map více než tři, ale ne víc než 500. Vytvoříme si z map a serveru graf, ohodnotíme hrany diferencí map krát W. (Tu spočítáme snadno, projdeme mapy políčko po políčku a spočítáme si počet těch rozdílných.) V tomto grafu nalezneme minimální kostru algoritmem z kuchařky. Ne nadarmo v ní byly zmíněny tři. Vyberme si třeba *Kruskalův algoritmus*.

(Mimochodem, když neřekneme přesně, který algoritmus na minimální kostru se má použít, bude naše řešení generické. Ten, kdo ho bude programovat, si musí sám nějaký vybrat a dosadit.)

Pořadí stahování map pak snadno vypíšeme procházením stromu od serveru do šířky nebo do hloubky v čase $\mathcal{O}(N)$.

Z toho vyplývá časová složitost, ta je $\mathcal{O}(N^2(RS+\log N))$, protože $\mathcal{O}(N^2RS)$ trvá postavení grafu a $\mathcal{O}(M \log N)$ zabere Kruskalův algoritmus (M je počet hran, v našem případě N^2).

Paměťová složitost je $\mathcal{O}(N^2 + NRS)$, potřebujeme si pamatovat celý vstup a všechny hodnoty diferencí.

Jelikož jste mohli předpokládat, že $N \leq 500$ a $R, S \leq 20$, je toto řešení dostatečně rychlé na to, aby mohlo být v praxi použitelné. Většina z vás se u něčeho takového tedy zastavila a neměli jste motivaci přemýšlet nad ještě rychlejším řešením. Proto jste i za toto mohli získat plný počet bodů.

Někteří z vás si všimli, že existuje vlastně jen RS možných ohodnocení hran, počet rozdílných políček ve dvou mapách může být jedině mezi 0 a RS. To znamená, že je můžeme setřídit přihrádkově, v čase lineárním s jejich počtem, tedy v $\mathcal{O}(N^2)$.

Kruskalův algoritmus s využitím DFU (s *union by rank* a s *path compression*) pak vytvoří minimální kostru v čase $\mathcal{O}(M\alpha(N))$, což je v našem případě $\mathcal{O}(N^2\alpha(N))$, kde α je inverzní *Ackermannova funkce* (její definici najdete v kuchařce, zde stačí, že roste extrémně pomalu).

Když sečteme vytváření grafu, třídění a vytváření minimální kostry, vyjde nám časová složitost $\mathcal{O}(N^2(RS + \alpha(N))$.

S naším omezením je $RS \leq 400$ a $\alpha(N) \leq \alpha(500) \leq 4$. Můžeme tedy v klidu říct, že celková složitost je zhruba $\mathcal{O}(N^2RS)$. Paměťová zůstává.

Program – Kruskalův algoritmus (Python 3):
`http://ksp.mff.cuni.cz/viz/27-2-4-kruskal.py`

Zlepšení o slepičí krok

Tím se ale nespokojíme, existuje pěkné řešení, které je asi tak o slepičí krok lepší, není „téměř“ lineární, ale „úplně“ lineární s velikostí grafu. Použijeme upravený Jarníkův algoritmus.

Ten staví kostru postupně od jednoho vrcholu a vždy k ní přidává tu nejlehčí hranu, která vede z kostry do zbytku grafu. K tomu používá nějakou datovou strukturu Q, která je vlastně množinou vrcholů schopnou vydat minimum. Ještě si připravíme dvě pole t a d indexované vrcholy; d udává nejkratší hranu z kostry do vrcholu, v t bude na konci algoritmu minimální kostra zadaná jako postup přilepování hran stromu od kořene.

Na začátku si do Q vložíme všechny vrcholy, d bude pro každý vrchol kromě startovního nekonečno, pro startovní vrchol 0, t nechť je pro každý vrchol NULL, neexistující vrchol. Tato hodnota nakonec zbude jen u startovního vrcholu.

Potom se provedou následující kroky:

1. dokud $Q \neq \emptyset$:
2. $\quad u \leftarrow$ vyjmi minimum z Q podle d
3. $\quad$ pro každého souseda v vrcholu u dělej:
4. $\quad\quad$ pokud $v \in Q$ a $d(v) > w(u, v)$:
5. $\quad\quad\quad d(v) \leftarrow w(u, v); t(v) \leftarrow u$

Výstup: t

Co bude datová struktura Q? Podobnou jistě znáte z Dijkstrova algoritmu, tam se používá halda. My ale využijeme toho, že hrany mají jen RS různých hodnot, a tak použijeme pole. Indexovat ho budeme od 0 do RS. Výběr minima bude trvat $\mathcal{O}(RS)$, pole budeme procházet od začátku do konce a v nejhorším případě najdeme první vrchol až na konci. Každý vrchol má $\mathcal{O}(N)$ sousedů, pro každého se bude provádět změna hodnoty, ta trvá pro každého ale jen $\mathcal{O}(1)$, vrchol se pouze přemístí na jiný index.

Časová složitost tedy bude $\mathcal{O}(N^2RS)$. Kolikrát budeme potřebovat znát váhu hrany? Jen jednou. Nemusíme si ji tedy nikam ukládat, ale spočítáme si ji ve chvíli, kdy ji poprvé využijeme. Díky tomu nám stačí pouze lineární paměť k velikosti vstupu, $\mathcal{O}(N \cdot RS)$.

Zbývá si už jen domyslet pár implementačních detailů. K tomu vám může pomoci zdroják:

Program – Jarníkův algoritmus (Python 3):
`http://ksp.mff.cuni.cz/viz/27-2-4-jarnik.py`

Dominik Macháček

27-2-5 Nejdelší příkaz

Úlohu si nejdříve zanalyzujeme. Potřebujeme v zadaných slovech najít co nejdelší žebřík, tj. posloupnost slov, kde každé další vznikne vložením právě jednoho písmene do slova předešlého. Pokud se na žebřík podíváme z druhé strany, hledáme posloupnost slov, kde každé další vznikne vyškrtnutím právě jednoho písmene.

Pro jedno konkrétní slovo můžeme zkusit všechny možnosti vyškrtnutí a kontrolovat, zda nově vzniklé slovo máme na seznamu. Pokud ano, vyzkoušíme pro něj to samé rekurzivně. Výše popsaný proces zkusíme začít v každém slově a tam, kde se dostaneme v rekurzi nejhlouběji, je naše řešení.

Pro dokončení popisu řešení ještě potřebujeme najít vhodnou datovou strukturu pro uchovávání slov a jejich hledání. Takovou je třeba trie. Pokud jste o ní ještě neslyšeli, můžete se o ní dočíst v naší kuchařce o hledání v textu.[54]

Tím dostáváme funkční řešení. Kvůli rekurzi ale bohužel ne dost dobré. Můžeme si všimnout, že při výpočtu můžeme zbytečně spouštět výpočet vícekrát pro stejné slovo. Tomu se budeme snažit vyvarovat a výpočet pro jedno slovo vždy spouštět pouze jednou (použijeme myšlenku dynamického programování).[55]

Trii budeme stavět od nejkratších slov po nejdelší a v koncovém vrcholu slova si budeme pamatovat délku nejdelšího žebříku pro dané slovo nebo nulu, pokud ve vrcholu trie žádné slovo nekončí.

[54] `http://ksp.mff.cuni.cz/viz/kucharky/hledani-v-textu`
[55] `http://ksp.mff.cuni.cz/viz/kucharky/dynamicke-programovani`

Při přidávání slova do trie tedy nejdřív zkusíme najít všechny možné předchůdce v dosavadní trii, z nich vybereme toho s nejvyšší hodnotou žebříku a aktuální slovo přidáme s hodnotou o jedna větší. Na konci pak jen najdeme největší hodnotu v trii a je to! Z toho pak jednoduše získáme výsledný žebřík.

Časová složitost je $\mathcal{O}(NL \cdot$(složitost trie)), což je pro konstantní abecedu $\mathcal{O}(NL^2)$, kde N je počet slov na vstupu a L je délka nejdelšího z nich. Pro každé slovo provádíme $\mathcal{O}(L)$ dotazů na trii, kde každý má složitost $\mathcal{O}(L)$.

Pro více detailů můžete nahlédnout do vzorové implementace v jazyce C++.

KSP

Program (C++): `http://ksp.mff.cuni.cz/viz/27-2-5.cpp`

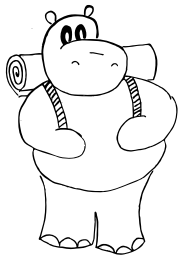
Karel Tesař

27-2-6 Testování odezvy

Na vstupu máme matici A o rozměrech $N \times N$ a chceme určit, jestli to může být matice vzdáleností v ohodnoceném stromě, a najít takový strom T. Z příkladu je jasné, že nám jde o neorientovaný graf (matice je symetrická, tedy $A_{ij} = A_{ji}$) bez smyček (na diagonále jsou nuly: $A_{ii} = 0$).

řešení

Pokud strom existuje, ve vstupní matici je spousta hodnot, které odpovídají cestám s více hranami. My ale potřebujeme vědět, které hodnoty odpovídají jednotlivým hranám. Kdybychom to věděli, strom už si ze seznamu hran snadno převedeme do příjemnější reprezentace a stejně snadno ověříme, že A je jeho matice vzdáleností.



Při ověřování můžeme využít toho, že cesta mezi každými dvěma vrcholy je ve stromu určena jednoznačně, takže vzdálenost všech vrcholů od jednoho pevně zvoleného jde snadno spočítat během průchodu (třeba DFS). Postupně tedy spustíme průchod z každého vrcholu stromu T a ověříme, že ve vstupní matici A jsou správné hodnoty. Strom by měl mít N vrcholů a $N-1$ hran (udělejte si cvičení z grafové kuchařky[56] a dokažte to!), takže všech N průchodů by dohromady trvalo $\mathcal{O}(N^2)$. To je lineární s velikostí kontrolované matice, kterou je potřeba projít celou, takže jsme na optimu.

Dobře, dobře, kontrolu matice vzdáleností pro daný strom jsme vymysleli, kde ale ten strom sebrat?

V lehčí variantě úlohy měl být strom T neohodnocený, lépe řečeno ohodnocený jedničkami. Stačilo tedy vzít všechny jedničkové hrany a ověřit, že tvoří strom. Žádná jedničková hrana nemůže odpovídat cestě s více než jednou hranou, protože by pak musela mít ohodnocení alespoň 2, takže jsme do T nepřidali žádnou hranu, která tam být neměla. Ze zadání a výběru hran plyne, že tam ani

[56] `http://ksp.mff.cuni.cz/viz/kucharky/grafy`

žádná nechybí. Ještě by se nám ale mohlo stát, že jsme dostali něco, co vůbec není strom. To můžeme zkontrolovat více způsoby, podle použité charakterizace stromů.

Strom je souvislý graf bez kružnic. Pokud prohledáním grafu (třeba DFS) najdeme všechny vrcholy, je souvislý. Kružnici detekujeme, pokud najdeme zpětnou hranu ve stromě průchodu do hloubky. (Pokud nevíte, o čem je řeč, osvěžte si paměť pohledem do již zmiňované grafové kuchařky.) Alternativně můžeme testovat, že má strom T správný počet vrcholů i hran. Oba tyto algoritmy běží v čase i prostoru $\mathcal{O}(N)$. Ostatní charakterizace stromů nejsou pro testování vhodné. Mimochodem, tahle část našeho algoritmu je shodná s úlohou 27-Z2-5.[57]

V lehčí variantě tedy máme snadný tip na strom T a ověříme, že to opravdu strom je a že vzdálenosti v něm odpovídají vstupní matici A. Načíst vstup zvládneme v čase i prostoru $\mathcal{O}(N^2)$, najít všech $N-1$ jedničkových hran a postavit z nich rozumně reprezentovaný strom taktéž, průchod stromem pro ověření souvislosti zabere dokonce jen čas $\mathcal{O}(N)$ a konečnou kontrolu matice vzdáleností jsme už spočítali na $\mathcal{O}(N^2)$. Celkově se tedy vejdeme do času $\mathcal{O}(N^2)$, což je lineární s velikostí vstupu, tedy zaručeně optimální. V paměti bude nejvíc místa zabírat matice A, zbytek se v tom ztratí, takže i paměťové nároky budeme mít lineární.

V těžší variantě se musíme zamyslet o trochu víc. Možná nás napadne se na matici A koukat jako na matici sousednosti úplného grafu K, kde strom T hledáme jako podgraf. Hm, a T musí mít stejný počet vrcholů jako $K\ldots$ takže je jeho kostrou! Nebude to minimální kostra, když už je popsaná v kuchařce k této sérii?

Úplný graf je souvislý, takže vždycky nějakou kostru má, speciálně tedy i minimální kostru T. Dokažme, že pokud má úloha řešení, je jím minimální kostra. Pro spor předpokládejme, že existuje řešení S a není to minimální kostra T. Vezmeme nejlehčí hranu uv z T, která není v S, a podíváme se na běh Kruskalova algoritmu, který v každém kroku spojí dvě komponenty souvislosti nejlehčí hranou mezi nimi.

V řešení S jsou vrcholy u a v spojené cestou $S[u,v]$.

- Pokud je $S[u,v]$ tvořena pouze hranou uv, dostáváme spor s volbou uv, protože jsme chtěli, aby nebyla v S.
- Jinak je $S[u,v]$ tvořena aspoň dvěma hranami. Jelikož při přidávání hrany uv do T musely být vrcholy u a v v různých komponentách a Kruskalův algoritmus zpracovává hrany v pořadí rostoucí váhy, musí v $S[u,v]$ být aspoň jedna hrana aspoň stejně těžká jako uv. To je spor s tím, že S je řešení, jelikož by uv nutně byla lehčí než $S[u,v]$, ale měla by být stejně těžká.

[57] `http://ksp.mff.cuni.cz/viz/27-Z2-5`

A máme dokázáno. Nebo ne? Celou dobu uvažujeme jen kladné váhy hran. Jestli jste předpokládali na vstupu matici sousednosti kladně ohodnoceného úplného grafu automaticky, nic se neděje, za to jsme body nestrhávali. Úloha byla zamýšlená s kladným ohodnocením a příklad tomu nasvědčoval. Přesto bychom rádi v rychlosti zmínili také rozšířenou variantu i se zápornými nebo nulovými vahami, která některé z vás napadla.

KSP

Minimální kostru pro **libovolné celočíselné váhy hran** zvládnou všechny standardní algoritmy, dokonce bez modifikace. Přinejhorším bychom mohli nezáporné váhy získat odečtením váhy nejzápornější hrany w^- od váhy každé hrany. Tím bychom každou kostru ztížili o $(N-1)\cdot w^-$, takže bychom uspořádání koster podle váhy nezměnili.

Potíž je v tom, že se zápornými vahami minimální kostra neřeší naši úlohu. Představte si třeba trojúhelník s hranami $1, 2, -1$, obsahující cestu $2, -1$. Pro úlohu se zápornými hranami neznáme efektivní řešení. Ale ani nemáme důkaz její NP-těžkosti, takže kdybyste na jedno nebo druhé přišli, dejte nám vědět na fóru. :-)

řešení

Nulové hrany umíme bez újmy na obecnosti spravit kontrakcí na začátku a dekontrakcí na konci algoritmu. V podstatě tak uvedeme v realitu, co nulová hrana neformálně říká: že její krajní vrcholy jsou vlastně vrchol jeden. Kontrakce hrany uv se projeví vyhozením řádku v a sloupce v z matice A. Přitom kontrolujeme, že se shodují s řádkem u a sloupcem u, jinak strom neexistuje. Tato úprava se hodí jen pro účely analýzy, algoritmus poběží správně i bez ní (zdůvodnění si rozmyslete).

Celý algoritmus v pseudokódu:

1. Načti matici A a zkontroluj, že je symetrická, má nuly na diagonále a jinak obsahuje jen kladné hodnoty. Pokud kontrola neuspěje, vrať „strom neexistuje“.
2. Najdi minimální kostru T úplného grafu s maticí sousednosti A.
3. Ověř, že získaná kostra T má matici vzdáleností rovnou matici A. Pokud se matice liší, vrať „strom neexistuje“.
4. Vrať strom T.

Zbývá vybrat algoritmus pro **nalezení minimální kostry**. Vymýšlet vlastní nemá smysl, jen bychom znovu vynalézali kolo a mořili se s důkazem korektnosti a složitosti.

Kruskalův algoritmus máme rozebraný v kuchařce, nejdéle na něm trvá třídění hran v čase $\mathcal{O}(M \log M) = \mathcal{O}(M \log N)$, což by pro nás bylo $\mathcal{O}(N^2 \log N)$. Hodí se pro řídké grafy a kvůli snadné implementaci, když zrovna nemáte po ruce haldu.

Pro husté grafy, kterým úplný graf bezesporu je, ale je vhodnější Jarníkův algoritmus. S Fibonacciho haldou běží v čase $\mathcal{O}(M + N \log N)$, což by pro náš případ bylo $\mathcal{O}(N^2 + N \log N) = \mathcal{O}(N^2)$.

Kniha The Algorithm Design Manual od Stevena S. Skieny tvrdí, že Jarník s párovacími haldami je nejrychlejším praktickým řešením pro řídké i husté grafy, se stejnou asymptotickou složitostí jako s Fibonacciho haldou. Nás z knihy bude zajímat implementace Jarníka, která běží vždycky v $\mathcal{O}(N^2)$ a ani nepotřebuje žádnou složitou datovou strukturu. V češtině ji najdete popsanou v Medvědových poznámkách z ADS.[58]

Tato varianta Jarníka je k vidění ve vzorovém řešení. Nalezení kostry je nejtěžší částí algoritmu, tedy i těžší varianta úlohy má řešení v lineárním čase i prostoru.

Program (C): `http://ksp.mff.cuni.cz/viz/27-2-6.c`

Tomáš „Palec“ Maleček

27-2-7 Shellové skripty

Snad každý, kdo se do úloh pustil, vyřešil všechny – to nás těší. Řešení obsahovala většinou jen drobné nedostatky, hlavně u vypisování proměnných. Když vypisujete obsah proměnné, který může obsahovat mezery, je potřeba ji vypsat v uvozovkách, jinak se mezery ztratí během rozdělování na slova.


```
echo "$prom"
```

Teď už k řešení samotných úloh. První byla velmi jednoduchá, stačilo se podívat do manuálu k příkazu `test`. Častou chybou nicméně bylo, že jste se snažili testovat na neprázdnost i adresáře. Taky se často v řešení vyskyla konstrukce:

```
for f in $(ls)
```

To není špatně (ani jsme za to nestrhávali body), jen je to zbytečně pomalé. Stejnou službu umí vykonat shell sám expanzí:

```
for f in *
```

Občas se taky v řešení objevilo testování neprázdnosti pomocí `wc -c`. Tuto metodu jsme použili v řešení minulé série, protože jsme ještě neznali lepší prostředky. Musí vám být ale jasné, že takový test bude ukrutně pomalý.

Jak tedy mohlo vypadat ideální řešení?

```
for f in *; do
        [ -f "$f" -a ! -s "$f" ] && echo "$f"
done
```

Druhá úloha byla trochu triková. Někteří se k tomu snažili použít pole, je to ale zbytečně silný a nepřenositelný kanón. Dnes už je situace s podporou polí v různých shellech lepší než před lety, stále se ale budeme snažit používat jednodušší prostředky.

[58] `http://mj.ucw.cz/vyuka/1011/ads1/7-kostry.pdf`

Naproti tomu, využít k řešení malou utilitku `tac`, která vypíše řádky vstupu v opačném pořadí, je moc pěkný nápad:

```
IFS='
'
echo "$*" | tac
```

Slepit malé utilitky, které za nás udělají špinavou práci, je přesně filozofie programování v shellu. Jak to udělat bez ní, je zamýšlené autorské řešení:

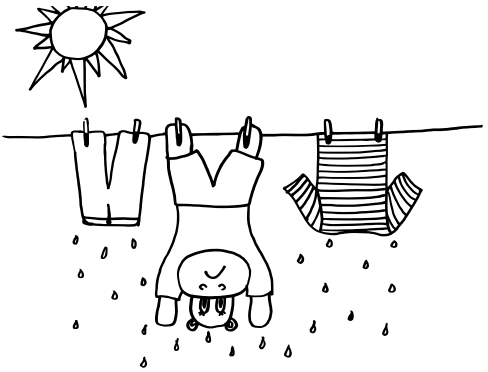
KSP

```
for arg; do
        p="$arg $p"
done
echo "$p"
```

Jestli budou argumenty na oddělených řádcích, nebo na jedné, nebylo důležité, stejně tak jestli se přidají apostrofy, uvozovky atp. okolo. Důležité ale bylo, aby se neztrácely mezery a argumenty tvořené pouze bílými znaky.

řešení

Třetí úloha byla jen krátký test pozornosti. Pokud použijeme rouru, odsuzujeme příkazy k tomu, aby se spustily v subshellu. Pokud v subshellu změníme nějaké proměnné, změní se jen pro subshell – a zaniknou spolu s ním. Proto musíme `read` a `echo` provést v jednom složeném příkazu.



Ve čtvrté úloze jste si měli vyzkoušet sílu proměnné `IFS` ve spolupráci s příkazem `read`. Někteří se snažili řádky zpracovávat příkazem `cut`, nebo ještě kostrbatěji pomocí `sed`u. To bohužel většinou vedlo k tomu, že jste soubor četli pro každý řádek celý znovu, a to opravdu není správný přístup.

```
while IFS=: read user x x x x x shell; do
        [ -n "$user" ] && echo "$shell" >"$user"
done </etc/passwd
```

Všimněte si pěkné zkratky na posledním řádku.

A konečně poslední pátá úloha byla na procvičení možností `case`. Jen málokdo to ale pochopil, proto jste do řešení často psali šílené podmínkové konstrukce. Také jste často zapomínali uvozovkovat alespoň jméno. Jinak ale nebyl na úloze žádný chyták, proto pojďme rovnou k řešení:

```
while read akce pohl jmeno; do
        case "$akce $pohl" in
                "prichod M")
                        echo "Prisel $jmeno" ;;
                "odchod M")
                        echo "Odesel $jmeno" ;;
                "prichod F")
                        echo "Prisla $jmeno" ;;
                "odchod F")
                        echo "Odesla $jmeno" ;;
        esac
done
```

Díky za pěkná řešení, těšíme se na další!

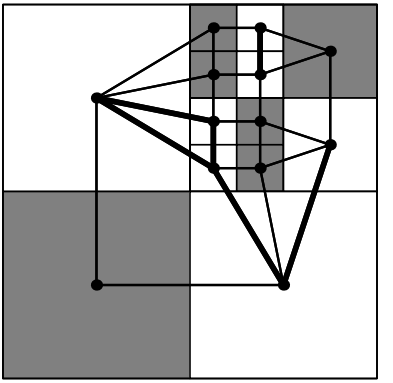
Ondra Hlavatý

27-3-1 Plocha k přistání

Většina z vás asi ví, jak se největší bílá oblast hledá, pokud jde o obyčejný bitmapový obrázek. Na obrázek se lze dívat jako na graf, kde pixely představují vrcholy a každá dvojice sousedních bodů je spojena hranou (tedy graf má tvar mřížky). V takovémto grafu bychom chtěli najít „komponenty bílé souvislosti“ – tedy maximální souvislé části grafu, které jsou celé bílé. K tomu můžeme použít klasický algoritmus na hledání komponent souvislosti prohledáváním do šířky či hloubky (pokud jej neznáte, nahlédněte do naší grafové kuchařky),[59] jen s tím rozdílem, že při prohledávání zcela ignorujeme černé vrcholy – vůbec je nenavštěvujeme. Snadno si rozmyslíte, že takto získáme očekávaný výsledek. Průběžně počítáme velikost nalezených komponent a nakonec jen vybereme maximum. Na tento postup jde pohlížet také jako na opakované použití klasického algoritmu *flood fill*.

Zkusme se tím inspirovat. I kvadrantový obrázek má něco jako „pixely“, tedy elementární jednobarevné plochy – jsou to všechny nepodrozdělené čtverce. Jen je každý jinak velký a může mít víc než čtyři sousedy. Navíc není vůbec jasné, jak tyto sousedy najít. Ale pokud by nám někdo dal graf, jehož vrcholy jsou elementární čtverce ohodnocené svou plochou a hrany představují jejich sousednosti, dokázali bychom už jednoduše úlohu vyřešit.

Graf sousednosti může vypadat například takto (tučně jsou vyznačeny bílé komponenty):

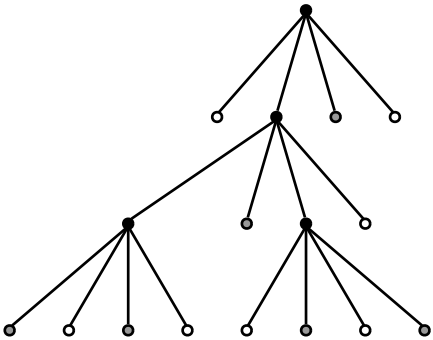


Zbytek řešení bude pojednávat o tom, jak si takový graf pořídit, a to hned dvěma různými způsoby. Na úvod ovšem povězme pár věcí oběma řešením společných. Řešení víceméně každé úlohy nad kvadrantovým kódem začíná tím, že z kódu vytvoříme takzvaný *kvadrantový strom*. Kořen tohoto stromu reprezentuje

[59] `http://ksp.mff.cuni.cz/viz/kucharky/grafy`

celý obrázek. Pokud je jednobarevný, pamatuje si svou barvu a nemá žádné potomky; v opačném případě má právě čtyři syny představující kvadrantové stromy pro jednotlivé čtvrtiny.

Strom pro kvadrantový kód `(0((1010)1(0101)0)10)` (odpovídá obrázku výše) vypadá takto:



Každý vrchol kvadrantového stromu představuje čtverec $2^r \times 2^r$, kde r je takzvaný *řád* vrcholu (čtverce). Kořen má řád h (výška stromu), jeho synové $h-1$, atd.

Strom vytvoříme přímočarou rekurzivní funkcí. Pokud se vyhneme zbytečnému kopírování řetězců (například si kód uložíme v globální proměnné a rekurzivním voláním budeme předávat jen index znaku, od kterého začít), zvládneme to v lineárním čase. Podrobněji ve zdrojáku.

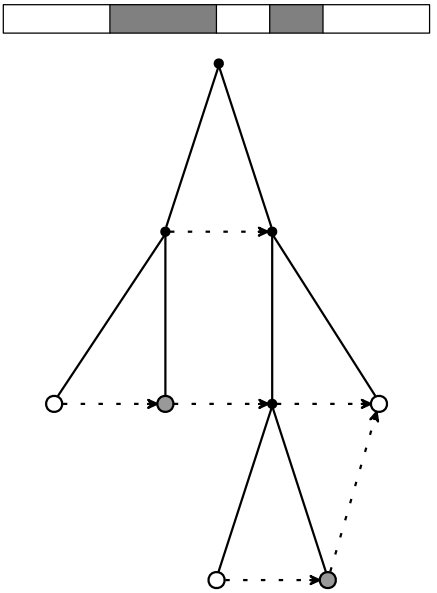
Řešení průchodem do šířky

Graf sousednosti vytvoříme tak, že pro každý jednobarevný čtverec (tomu odpovídá list kvadrantového stromu) najdeme seznam jednobarevných čtverců s ním v obrázku sousedících. To je ale těžké, neb jeden čtverec může mít sousedů hodně, a to i v docela vzdálených částech stromu. Nám ovšem stačí, když každou sousednost „objeví“ jen jeden ze zúčastněných čtverců, např. ten menší. Pak nám stačí pro každý čtverec hledat jen sousedy stejného nebo vyššího řádu (příslušné listy jsou ve stromu stejně vysoko nebo výš). A takový je v každém směru nejvýše jeden. Tím získáme všechny hrany grafu sousednosti orientované směrem od menšího čtverce k většímu, opačný směr doplníme jejich otočením, což můžeme dělat i průběžně. Z toho je také hned vidět, že hran je lineárně mnoho.

Samotné hledání sousedů provedeme průchodem stromu po patrech (neboli do šířky od kořene). Když navštívíme nějaký vrchol v, určíme jeho nejhlubšího (obrázkového) souseda na každé straně, který ale není hlouběji než on sám. Toho si označíme $S_s(v)$ (kde $s \in \{L, P, H, D\}$ je příslušná strana). Všimněme si, že pokud je v tím hlubším z nějaké dvojice sousedních listů, je $S_s(v)$ právě druhým vrcholem této dvojice, a tedy jsme objevili jednu hranu grafu sousednosti (a po

průchodu všech vrcholů objevíme všechny). V opačném případě je $S_s(v)$ nějaký vnitřní vrchol stromu, který se ale bude dále při hledání hodit.

To by si určitě zasloužilo obrázek. Abyste se neztratili v obrovském množství čar, ukážeme si to na příkladu jednorozměrného „kvadrantového“ obrázku. To je dlouhá „nudle“ rozměrů $2^k \times 1$, která je buď celá bílá, celá černá, nebo rozdělená na dvě poloviny, rekurzivně splňující stejnou definici. Kvadrantový strom takovéhoto obrázku je binární a má tu výhodu, že směr vlevo a vpravo ve stromu odpovídá stejnému směru v obrázku.



Tečkované šipky značí ukazatele S_P (vedou od v k $S_P(v)$).

Zbývá si rozmyslet, jak $S_s(v)$ určit. Pro jednoduchost uvažujme $s = P$, ostatní směry vyřešíme obdobně. Označme si p rodiče vrcholu v. Pokud čtverec odpovídající v leží v levé polovině čtverce p, stačí vzít správného sourozence v. Např. je-li v levým dolním synem p, $S_P(v)$ je pravý dolní syn p. V opačném případě se podíváme na vrchol $u := S_P(p)$ (p je ve vyšším patře, tedy jeho S už máme spočítané). Pokud u je list, pak $S_P(v) = u$. Pokud není list, víme, že leží na stejné úrovni jako p (kdyby ležel výš, nebyl by *nejhlubším* sousedem p, neb některý z jeho synů by také sousedil s p a byl hlouběji). V takovém případě $S_P(v)$ musí být jeden ze synů u, ten, který správně přiléhá k v. Například je-li v pravým horním synem p, pak $S_P(v)$ je levý horní syn u. Zvlášť musíme ošetřit případ, kdy v je v daném směru na kraji obrázku.

Určení $S_P(v)$ nás stojí konstantní čas, tedy celý průchod stromu zvládneme v lineárním čase a v jeho průběhu sestrojíme graf sousednosti jednobarevných čtverců. A už víme, že sestrojení kvadrantového stromu, rozklad grafu sousednosti na komponenty a výběr největší zvládneme rovněž v lineárním čase. Tedy i celý algoritmus si vystačí s $\mathcal{O}(n)$ času i paměti, kde n je délka vstupního kódu.

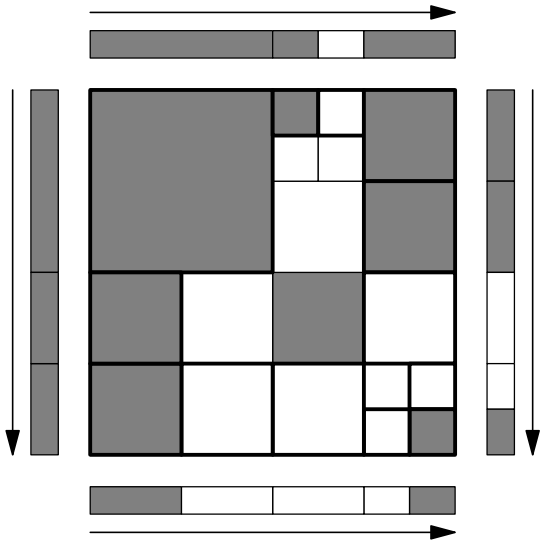
Program (Python): `http://ksp.mff.cuni.cz/viz/27-3-1-bfs.py`

Řešení průchodem do hloubky

Předchozí řešení vytvářelo sousednosti po hladinách shora dolů. Ukážeme jiný způsob, který postupuje naopak zdola nahoru a dosahuje stejné časové složitosti.

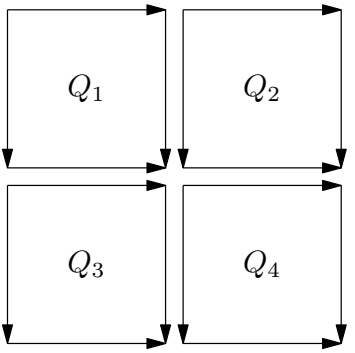
Strom budeme procházet do hloubky a průběžně budovat graf sousednosti. Přesněji řečeno, kdykoliv se při procházení stromu budeme vracet z nějakého čtverce, budou už zaznamenány všechny sousednosti mezi jeho potomky. Sousednosti vedoucí přes hranici aktuálního čtverce doplníme později.

Proto se nám bude hodit předávat do vyšších pater stromu informace o tom, jaké černé a bílé podčtverce leží na hranici aktuálního čtverce. To uložíme do čtyř seznamů. *Levý* seznam bude popisovat podčtverce přiléhající k levé hraně aktuálního čtverce, uspořádané shora dolů. Z každého podčtverce nás zajímá jen to, jak se dotýká hranice, což je nějaký *interval* y-ových souřadnic. Za každý interval přidáme do seznamu dvojici (v, r), kde v je příslušný vrchol grafu sousednosti a r řád podčtverce. Podobně vytvoříme *pravý* seznam (též uspořádaný shora dolů), *horní* a *dolní* (oba zleva doprava).



Vracíme-li se z jednobarevného čtverce (listu stromu), nemusíme vytvářet žádné sousednosti. Všechny čtyři seznamy budou obsahovat odkazy na tento čtverec.

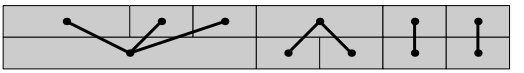
Zajímavější věci se dějí, pokud se vracíme z vícebarevného čtverce (vnitřního vrcholu stromu). Nechť Q_1 až Q_4 jsou kvadranty aktuálního čtverce. Z rekurze už známe sousednosti uvnitř kvadrantů a také podčtverce na hranicích kvadrantů. Nyní potřebujeme rozpoznat sousednosti vedoucí přes hranice kvadrantů a sestrojit hranici celého čtverce.



Nejprve vyřešíme hranici: levý seznam celého čtverce získáme spojením levého seznamu kvadrantu Q_1 s levým seznamem kvadrantu Q_3. Podobně získáme ostatní seznamy slepením seznamů z vnějších hranic kvadrantů.

Nyní sousednosti: uvažujme čtverce sousedící přes společnou hranici kvadrantů Q_1 a Q_2 (ostatní hranice zpracujeme obdobně), tedy intervaly z pravého seznamu Q_1 a levého seznamu Q_2. Kdykoliv se interval z Q_1 překrývá s intervalem z Q_2, chceme propojit hranou příslušné vrcholy v grafu sousednosti.

Propojování probíhá takto: podíváme se na počáteční interval každého seznamu – těmto intervalům říkejme třeba x a y. Pokud jsou stejného řádu, vytvoříme hranu. Pokud se řády liší (bez újmy na obecnosti x má vyšší řád), vezmeme x a z protilehlého seznamu budeme odebírat intervaly, dokud se jejich délky nenasčítají na délku x. (Všimněte si, že k tomu musí dojít přesně na hranici intervalu, neboť všechny intervaly vznikly postupným půlením délky hrany celého obrázku.) Kdykoliv odebíráme nějaký interval, vytvoříme hranu. Poté pokračujeme zbytkem obou seznamů, než se oba vyprázdní. Dopadne to například takto:



(Pokud vám na obrázku chybí hrana např. mezi třetím a čtvrtým intervalem v prvním řádku, uvědomte si, že odpovídá sousednosti v kolmém směru, takže vznikne při propojování jiných seznamů než těchto dvou.)

Umíme tedy zpracovat jak listy stromu, tak jejich vnitřní vrcholy. Nakonec se prohledávání vrátí z kořene stromu a graf sousednosti je hotov.

Zbývá určit časovou složitost. V každém vrcholu stromu strávíme konstantní čas spojováním seznamů – to je celkem $\mathcal{O}(n)$ za celý strom – a nějaký další čas zpracováním vnitřních hranic. Vnitřní hranice přitom může být poměrně komplikovaná, ale stačí si uvědomit, že kdykoliv sáhneme na nějaký interval, vypadne tento interval ze svého seznamu a už se nikdy do žádného seznamu nevrátí.

Čas strávený zpracováním všech hranic dohromady je tedy shora omezen počtem všech intervalů, které vložíme do seznamů. Vkládáme ale pouze v listech stromu: 4 intervaly za každý list. Tak vznikne celkem $\mathcal{O}(n)$ intervalů, takže jejich odebíráním strávíme čas $\mathcal{O}(n)$.

Program (C): `http://ksp.mff.cuni.cz/viz/27-3-1-dfs.c`

Trampoty s velkými čísly

Naše algoritmy počítají s délkami intervalů a plochami čtverců jako s normálními čísly. Tato čísla ale rostou exponenciálně s hloubkou stromu, takže by se mohlo stát, že se nevejdou do běžné celočíselné proměnné. Zadání o této možnosti nevinně mlčelo, ale zkusme ji aspoň na chvilku připustit. Držte si klobouky...

První z algoritmů na konstrukci grafu s délkami ani plochami nepočítá, takže ho měnit nemusíme.

Druhý algoritmus používá délky při propojování seznamů intervalů. Můžeme si pomoci takto: podíváme se na intervaly x a y na začátku seznamů. Pokud jsou stejného řádu, nic se nemění. Pokud je (BÚNO) x vyššího řádu, rozdělíme ho na dva intervaly o 1 nižšího řádu a postup opakujeme. Části vzniklé rozdělením zdědí informace o vrcholu grafu a o barvě z původního intervalu. Jako cvičení ponecháváme dokázat, že vytvoříme nejvýše lineárně mnoho nových intervalů (nápověda: představte si strom popisující postupné dělení intervalů).

Hledání komponent souvislosti zůstane beze změny, ale pak potřebujeme spočítat plochy komponent. Nejprve každému čtverci přiřadíme číslo komponenty. Poté projdeme celý strom po hladinách shora dolů, a kdykoli narazíme na jednobarevný čtverec, přidáme ho k příslušné komponentě. Pro každou komponentu tudíž vznikne seznam jejích čtverců uspořádaný sestupně podle řádu.

Nyní v tomto pořadí budeme sčítat plochy čtverců. To jsou potenciálně obrovská čísla. Budeme je zapisovat ve dvojkové soustavě a do paměti ukládat jako seznam pozic jedniček ve dvojkovém zápisu, uložený od nejvyššího bitu k nejnižšímu. Každý další čtverec přitom přispěje plochou, která je menší nebo rovna zatím nasčítané ploše. Můžeme tedy připsat jedničkový bit na konec čísla, jen se nám mohlo stát, že už tam jeden bit tohoto řádu mohl být, takže dojde k přenosu. Vyřizováním všech přenosů ovšem strávíme lineární čas, protože s každým

přenosem klesne celkový počet jedničkových bitů o 1. (To je podobná úvaha jako onehdy s intervaly.)

Plochy máme spočítány, zbývá z nich najít maximum. Kdykoliv při tom porovnáváme dvě čísla, procházíme je od nejvyššího řádu a jakmile se přestanou shodovat, porovnávání ukončíme. Čas strávený porovnáváním přitom účtujeme jedničkám v menším z obou čísel, které se už žádného dalšího porovnávání nezúčastní. Takto celkem naúčtujeme každé jedničce čas $\mathcal{O}(1)$ a všech jedniček za celou dobu běhu algoritmu vznikne $\mathcal{O}(n)$.

KSP

I s obrovskými čísly jsme tedy dokázali udržet lineární složitost algoritmu. Kouzlo se podařilo ;-)

Martin „Medvěd“ Mareš & Filip Štědronský

27-3-2 Návrhy pro komisi

řešení

V úloze se pracuje s řetězci, zkušený řešitel si tedy přečte zadání, zamyslí se a řekne si „Ha, trie!“. To je speciální strom, ve kterém se hranám přiřazují písmenka a cesta od kořene nějak odpovídá vstupním řetězcům; více o ní píšeme v kuchařce o hledání v textu.[60]

Jenže jak přesně trii na naši úlohu použít? Můžeme jednoduše počítat, kolik členů komise schvaluje určité slovo – stačí si do vrcholů přidat počítadlo. Když se pak vydáme od kořene dolů a budeme tyto členy postupně sčítat, zajímá nás, jestli v daném vrcholu součet dosáhne alespoň hodnoty K. Pokud ano, jakýkoliv návrh začínající slovem odpovídajícím aktuálnímu vrcholu bude schválen.

Kolik takových návrhů existuje? Když aktuální hloubku označíme jako d, dá se jejich počet vyčíslit jako 26^{D-d}. Za každý ze znaků, který chybí do přijatelné délky návrhu D, totiž smíme zvolit libovolný znak anglické abecedy.

Také z uzlu, v kterém se nasbíralo alespoň K hlasů, nechceme postupovat níž – všechna možná pokračování budou začínat schvalovaným řetězcem, takže jsme je už zahrnuli. Při dosažení hodnoty K se tedy zastavíme, resp. vrátíme o úroveň výš a necháme procházení postupovat dál.

To zní dobře. Ale bude to opravdu fungovat? Co kdyby nějaký člen schvaloval jak slovo „psa“, tak slovo „psal“? Ouha! To bychom ho započítali vícekrát, což nechceme – a zadání nám rozhodně neslibuje, že taková situace nenastane.

Připomeňme si naznačenou myšlenku: pokud člen (či komise) schvaluje nějaké slovo, schválí i jakékoliv další, které tímto slovem začíná. Kdybychom tedy věděli, že člen už schvaluje nějaký prefix aktuálního slova, můžeme právě zpracovávané slovo s klidem zahodit.

Tady se nabízejí dva přístupy. Můžeme slova každého člena lexikograficky seřadit a přidávat je do trie už seřazená. Jen si musíme rozmyslet, jak v trii komise

[60] `http://ksp.mff.cuni.cz/viz/kucharky/hledani-v-textu`

poznat, zda jsme do aktuálního vrcholu už přičetli hlas právě zpracovávaného člena.

Alternativu, se kterou pracuje i vzorový program, představuje vybudování druhé trie, tentokrát specifické pro člena. Do ní jednoduše naházíme všechna jeho oblíbená slova (značíme si ve vrcholech, jestli tam končí slovo). Následně ji projdeme a vždy, když narazíme na konec slova, odpovídající slovo přidáme do trie pro komisi a hned se vrátíme.

Po zpracování slov všech členů trii projdeme tak, jak jsme popsali na začátku. Teď už hlas každého člena započítáme pro libovolný řetězec maximálně jednou, takže program vydá správný výsledek. Zbývá se zamyslet nad složitostí.

Co je vlastně vstup? Kromě parametrů C, K a D to jsou zejména všechna oblíbená slova. Označme si počet všech znaků v nich jako ℓ.

Operace s trií, když máme konstantně velkou abecedu, můžeme bez obav prohlásit za konstantní. Do trie pro komisi přidáme maximálně ℓ znaků, čímž vytvoříme maximálně ℓ vrcholů. Při závěrečném průchodu navštívíme každý vrchol nanejvýš jednou, zpracování trie pro celou komisi tedy trvá $\mathcal{O}(\ell)$ času a zabírá $\mathcal{O}(\ell)$ paměti.

To samé platí pro trie jednotlivých členů. Zdánlivě tedy máme celkovou složitost $\mathcal{O}(C \cdot \ell)$. Jenže trie všech členů nemůžou mít dohromady víc než ℓ znaků, takže zpracování všech členských trií dohromady nemůže zabrat víc než $\mathcal{O}(\ell)$.

Započítali jsme všechny operace? Skoro. Neměli bychom zapomenout na mocnění, ač ve vašich řešeních jsem jeho zohlednění nevyžadovala. Zatímco násobení za konstantní prohlásit můžeme, u mocnění by to bylo poněkud odvážné, zvlášť když by teoreticky mohlo nastat $D \gg \ell$. Trochu si ovšem život usnadníme a jen prohlásíme, že mocnění trvá $\mathcal{O}(m)$. Pak má celý náš program časovou složitost $\mathcal{O}(\ell \cdot m)$, paměťovou $\mathcal{O}(\ell)$.

(Kdybychom se rozhodli slova nevkládat do trií, ale řadit, zvládneme to také lineárně – díky pevné velikosti abecedy to jde pomocí RadixSortu.)

Za zmínku možná stojí, že v téhle úloze občas realita spráská asymptotiku a pošle ji stydět se do kouta. I nepříliš optimalizovaná řešení se složitostí $\mathcal{O}(\ell \log \ell)$ můžou pro rozumně velké vstupy doběhnout rychleji než řešení lineární. Souvisí to mimo jiné s tím, že trie je budovaná pomocí ukazatelů, a s efekty keší… ale to už by byl úplně jiný příběh.

Program (C): `http://ksp.mff.cuni.cz/viz/27-3-2.c`

Karolína „Karryanna“ Burešová

27-3-3 Výběr vysílačů

Úlohou je vlastně obarvit strom barvami 1, 2, 4, 8, ... tak, aby sousední vrcholy měly různé barvy a součet hodnot barev přes všechny vrcholy byl co nejmenší. Připomeneme, že bez požadavku na minimální součet lze každý strom korektně obarvit dvěma barvami. To můžeme udělat například průchodem stromu do hloubky. Ten může vypadat například takto:

```
1. obarvi(v, b):
2.     barva[v] = b
3.     for (u in soused(v)):
4.         if (barva[u]==0): obarvi(u, 3 - b)
```

Nejdříve obarvíme libovolný vrchol barvou 1, pak víme, že jeho sousedi musí mít barvu 2, jejich sousedi zase barvu 1, a tak dále. Tento jednoduchý algoritmus má časovou složitost $\mathcal{O}(N)$ a budeme na něm dále stavět.

Tím, že strom umíme obarvit barvami 1 a 2, dostáváme obarvení se součtem maximálně $2N$. Tedy hledané obarvení s minimálním součtem určitě nepoužije barvy větší než $2N$. To zároveň znamená, že barev použijeme maximálně $\log N + 1$.

Nyní pomalu přejdeme k popisu algoritmu. Strom si zakořeníme v libovolném vrcholu a z něj pustíme prohledávání do hloubky. To vždy nejdříve spočítá řešení pro podstromy tvořené syny vrcholu a z těchto řešení složí řešení pro daný vrchol. Tento postup je takovým standardním dynamickým programováním na stromě. A co přesně v podstromech budeme počítat?

Pro každou barvu $c = 2^i$ pro $i = 1, \ldots, \log N + 1$ spočítáme nejlepší možné obarvení podstromu, ve kterém je kořen obarven barvou c. Abychom zjistili nejlepší obarvení pro barvu c, tak stačí pro každého syna vybrat nejlepší barvu jinou než c. To pro konkrétního syna vždy bude buď jeho nejlepší barva, anebo jeho druhá nejlepší barva. Tudíž nám stačí si pro každý podstrom pamatovat pouze dvě nejlepší možnosti.

Po zavolání výpočtu v kořeni stromu dostaneme výsledek. Během výpočtu pro každý vrchol zkoušíme $\log N + 1$ barev plus $(\log N + 1)$-krát u něj vybíráme jednu ze dvou barev pro otce. Tedy časová složitost algoritmu je $\mathcal{O}(N \log N)$. My ale algoritmus ještě vylepšíme.

Všimneme si, že pokud se má vyplatit vrcholu přiřadit barvu 2^i, tak všechny barvy z $2^0, \ldots, 2^{i-1}$ musí být zastoupeny v jeho synech. Tedy u vrcholu s k syny nám pro získání dvou nejlepších možností stačí vyzkoušet $k + 2$ barev (druhá nejlepší možnost pořád může mít barvu $k + 2$).

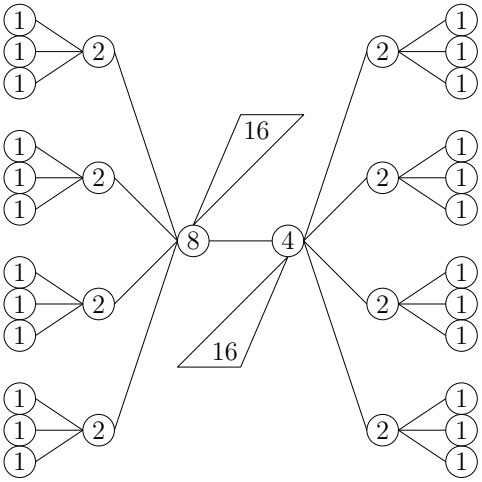
Další věc, kterou na předchozím algoritmu můžeme vylepšit, je opětovné vybírání z nejlepší a druhé nejlepší barvy synů. Víme, že vždy vybereme tu nejlepší kromě případu, kdy pro kořen zvolíme stejnou barvu, pak vybereme druhou nejlepší. Stačí nám spočítat součet nejlepších možností synů a pro každou

barvu $c = 2^i$ pro $i = 1, \ldots, k+2$ si předpočítat, o kolik se dohromady liší nejlepší a druhé nejlepší možnosti synů, kteří jako svou nejlepší barvu mají c. To pro vrchol s k syny můžeme jednoduše udělat v čase $\mathcal{O}(k)$.

Jelikož každý vrchol je synem právě (maximálně) jednoho jiného vrcholu, dostáváme se na časovou složitost $\mathcal{O}(N)$.

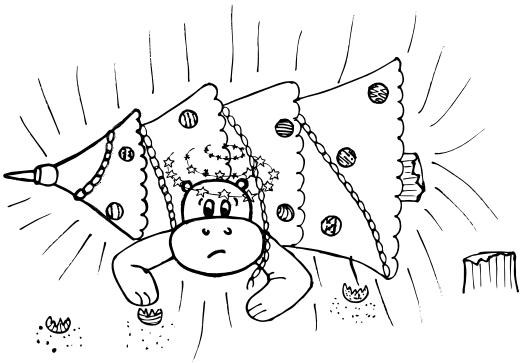
Pro zajímavost ještě řekneme, že stejný algoritmus funguje i pro verzi úlohy, kde strom barvíme barvami $1, 2, 3, \ldots, N$. Sami si můžete rozmyslet proč.

A kdo je zvědavý, tak strom, pro který jsou potřeba alespoň čtyři barvy, může vypadat například takto:



„Šestnáctkové“ trojúhelníky zastupují šestnáct přímých potomků uzlu, kde je každý obarvený jedničkou.

Karel Tesař



27-3-4 Doplňování energie

První, co nás může napadnout, je vyzkoušet všechny možnosti průchodů. Těch je však velmi mnoho, protože nám nic nebrání chodit přes některá místa vícekrát.

Ujasněme si nejprve, jak bude vypadat přelet, který může získat maximální množství energie. Protože nemáme žádné omezení na uchovávanou energii, vyplatí se nám vždy při prvním navštívení nějakého místa z něj veškerou energii vyčerpat. Nemá tedy smysl si ji zde šetřit na později. Stejně tak je zbytečné nějaké místo při přeletu vynechat a nezastavovat se na něm.

KSP

Díky tomu můžeme předpokládat, že při každém čerpání energie máme projitou souvislou oblast, a navíc se nacházíme na jejím kraji. Tím jsme výrazně snížili počet možností, které chceme vyzkoušet. Nyní máme pouze N levých a N pravých konců, celkem N^2 možných koncových stavů. Stav je tedy dvojice čísel (a, k), kde a je aktuální pozice a k druhý konec prošlé oblasti.

řešení

Jak ve stavech spočítat optimální množství energie? Pro stav s oběma konci intervalu na startovní pozici M energii určíme snadno. Bude rovna právě energii, kterou z této pozice můžeme čerpat.

Do ostatních stavů se můžeme dostat nejvýše ze dvou předchozích. Konkrétně pro $a > k$ se do stavu (a, k) dostaneme ze stavu $(a-1, k)$ nebo $(k, a-1)$. A pro $a < k$ to jsou stavy $(a+1, k)$ a $(k, a+1)$. Do nich se umíme dostat zase ze dvou předchozích, a tak dál. V každém případě můžeme začínat pouze ve stavu (M, M).

Stačí tedy z počátečního stavu procházet do šířky. Tím máme zaručeno, že projdeme všechny dosažitelné stavy a navíc je projdeme v tom pořadí, v jakém následují při průletu sondy. Můžeme tedy průběžně ve všech stavech počítat největší získatelné množství energie.

Při přechodu z (a, k) do $(a+1, k)$ odebereme energii rovnou vzdálenosti mezi místy a a $a+1$. Pokud však přecházíme přes prošlou oblast, tedy například z (a, k) do $(k-1, a)$, musíme odečíst i součet vzdáleností mezi všemi již prošlými místy. Nakonec nesmíme zapomenout přidat energii získanou na nové pozici.

Celý průchod je nakonec velmi jednoduchý, jenom se nesmíme ztratit v indexování a přičítání ± 1. Na detaily se podívejte do zdrojového kódu. Časová složitost celého algoritmu je $\mathcal{O}(N^2)$, protože každý z kvadraticky mnoha stavů zkoušíme nejvýše jednou. Díky tomu, že nepotřebujeme optimální cestu sondy zrekonstruovat, vystačíme si s lineární pamětí.

Program (C):
`http://ksp.mff.cuni.cz/viz/27-3-4.c`

Jenda Hadrava

27-3-5 Komprese obrazu

Protože na vstupu dostaneme vždy obrázek o rozměrech $2^K \times 2^K$, v každé úrovni rekurzivního komprimování pracujeme vždy se čtvercem. Pokud má na aktuální úrovni čtverec rozměry 1×1, nic již nekomprimujeme a jen vrátíme jeho barvu.

Co přesně máme v každém kroku kvadrantistické komprese dělat? Potřebujeme vybrat dvě čtvrtiny uvažovaného čtverce a každou „odbýt“ jednou barvou, respektive jednu prohlásit za celobílou a druhou za celočernou. Řekněme, že cenou výsledné komprese je počet pixelů, které musely být přebarveny.

To nás může svádět k řešení, ve kterém pro každou čtvrtinu spočteme počet černých a bílých pixelů. Následně „nejčernější“ z nich obarvíme černě, nejbělejší bíle a zbylé čtvrtiny pak zpracujeme rekurzivně, přičemž rekurzivní funkce bude vracet cenu zakomprimování dané části. Toto řešení má dva problémy – jednak není úplně jasné, jak vybírat nejbělejší a nejčernější čtvrtiny (může jich být více se stejným počtem bílých/černých pixelů), a navíc nemusí vést k optimálnímu řešení.

Mohlo by se totiž stát, že sice v jedné čtvrtině je hodně černých pixelů, více než v ostatních, ale díky rozložení černých pixelů se dobře kvadrantisticky komprimuje. Pak by jí tento algoritmus obarvil celočerně a namísto toho se kvadrantisticky snažil komprimovat čtvrtinu, která má pixely rozložené dost nepravidelně.

Zkusme to trochu jinak – na každé úrovni se pro každou čtvrtinu zeptáme naší rekurzivní funkce, kolik by stálo její zkomprimování. Budeme si pamatovat i počet černých a bílých pixelů v každé čtvrtině. Pokud známe tyto informace, můžeme prostě vyzkoušet všech 12 možností, jak vybrat celočernou a celobílou čtvrtinu, a seskládat pro každou z možností její cenu. Ze všech dvanácti cen si pak vybereme tu nejmenší.

Kolik na takový výpočet potřebujeme času? Povšimněme si, že rekurzivní funkce se nikdy nevolá znovu pro stejný čtverec, protože uvažované čtverce stejné velikosti se nikde nepřekrývají. Víme, že na každém čtverci velikosti $2^m \times 2^m$ strávíme $\mathcal{O}(4^m)$ času, a takových čtverců uvažujeme na každé úrovni rekurze $\frac{4^K}{4^m}$. Celkem tedy potřebujeme $\mathcal{O}(4^K \cdot K)$ času i paměti.

Toto řešení by šlo ještě zrychlit: určit počet bílých pixelů pro každý čtverec lze v čase $\mathcal{O}(1)$, protože stačí sečíst výsledky ze čtyř menších čtvrtin. A pokud si vybrané „odbyté“ čtvrtiny budeme uchovávat trochu lépe, dostaneme řešení, které si vystačí s lineárním časem i pamětí v počtu pixelů.

Program (C++):
`http://ksp.mff.cuni.cz/viz/27-3-5.cpp`

Ondřej Hübsch

27-3-6 Ukládání přepravek

Někteří z vás poznali, že jde jen o jiné zadání daleko známější úlohy, totiž barvení intervalových grafů. Tato úloha se dá ve zkratce zadat jako hledání nejmenšího počtu poslucháren, které potřebujeme pro přednášky zadané svými časy začátků a konců.

Tato úloha se dá řešit hladově, a to ve velkém množství variant, proto byla většina řešení funkční. Hlavní část řešení je ovšem obhájit, že hladově řešit opravdu jde.

KSP

Začneme tím, že si přepravky seřadíme podle vnějšího průměru sestupně. Na vnitřním průměru tady nezáleží, jak později uvidíme. Budeme si udržovat seznam potřebných komínků, jakési průběžné řešení. Jediná hodnota, kterou z každého komínku potřebujeme, je vnitřní průměr nejmenší přepravky.

řešení

Přepravky budeme postupně probírat a snažit se je vložit do nějakého komínku. Protože víme, že žádná větší přepravka už nepřijde, stačí ze seznamu vybrat maximum. Bude se nám tedy hodit reprezentovat komínky maximovou haldou.

Pokud je maximum menší než vnější průměr aktuální přepravky, pak žádný vhodný komínek neexistuje a proto přidáme nový. Na konci běhu algoritmu prostě jen spočítáme počet komínků ve stromu.

Konečnost algoritmu je zřejmá, podívejme se na správnost. Dokážeme ji indukcí podle velikosti průběžného řešení. Dokud máme jen jeden komínek (případně žádný), nemůže existovat lepší řešení.

Mějme tedy k komínků. Jediné, co může k zvýšit, je pak situace, kdy neexistuje pro nějakou přepravku vhodný komínek. V tu chvíli ale existuje alespoň k přepravek, co mají vnější průměr alespoň takový jako aktuální přepravka (díky setřízení), a vnitřní průměr ostře menší (neexistuje vhodný komínek). S aktuální přepravkou je to tedy $k + 1$ přepravek, ze kterých žádné dvě nejdou vložit do sebe. Optimální řešení tedy nemůže používat méně než $k + 1$ komínků.

Jaké vlastnosti algoritmus má? Už kvůli prvotnímu setřízení vidíme, že nepoběží rychleji než v $\mathcal{O}(N \log N)$. Další kroky algoritmu závisí na počtu komínků, tedy na řešení. Ten nemůžeme nijak rozumně odhadnout – počet komínků může být až N. Každý krok tedy zabere až $\mathcal{O}(\log N)$ času. Tedy nakonec se opravdu do $\mathcal{O}(N \log N)$ vejdeme. Paměti spotřebujeme $\mathcal{O}(N)$, více nepotřebujeme.

Vzorový kód je napsaný v C++ a využívá standardní implementaci haldy.

Program (C++):
`http://ksp.mff.cuni.cz/viz/27-3-6.cpp`

Ondra Hlavatý

27-3-7 UNIXové déjà vu

S třetím dílem seriálu o UNIXu jste se poprali statečně. Nejvíc zádrhelů bylo paradoxně v nejméně hodnoceném prvním úkolu, jehož vzorové řešení si necháme až na konec, abyste se nelekli a neutekli hned na začátku. Trochu jste také zápasili se skriptovacími úkoly, kde se projevily jednak nedostatek zkušeností s efektivním kombinováním shellových utilit, jednak znalosti získané jinde. Ty některým z vás dovolily napsat neportabilní kód, který by v jiném shellu než Bashi neběžel. Používejte shell častěji, zkuste si s jeho pomocí šetřit práci, hrajte si s ním. Stručné, elegantní a efektivní vyjadřování vám v něm pak půjde snáz a nebude vás to tolik svádět k používání céčkových konstrukcí v shellu.

Úkol 2 – Hardlinky na adresáře

Pohledem do `man ls` nebo do předchozího dílu seriálu můžeme zjistit, že `ls -ld adresar` vypíše podrobnosti o daném adresáři, nikoliv o souborech v něm umístěných, jako by to udělal bez přepínače `-d`. Počet linků je první číslo vpravo od práv. Experimentální pozorování na všech slušných systémech prozradí, že toto číslo je vždycky aspoň 2. Porovnáváním adresářů s nízkým a vysokým počtem linků přijdeme na to, že se o jedničku zvětší za každý podadresář. Po troše přemýšlení a přečtení výkladu v seriálu by mělo být jasné, že jeden link vede z nadřazeného adresáře, druhý z adresáře samotného (jmenuje se `.` – tečka) a zbytek jsou dvě tečky (`..`) z podadresářů.

Mezi slušné systémy tu nemůžeme počítat Cygwin, který o každém adresáři tvrdí, že má jediný link. Tečku a dvě tečky implementuje úplně speciálně, a to hlavně kvůli podpoře windowsových souborových systémů. Při hlubším pohledu do POSIXu jsem s podivem zjistil, že norma takové chování dovoluje – konkrétně v definici prázdného adresáře.

Tenhle úkol se ukázal být celkem jednoduchým na slušných i neslušných systémech.

Úkol 3 – Mazání obsahu a linku

Příkaz `cat </dev/null >soubor` zkrátí délku souboru na nulu. Pokud před jeho zavoláním soubor smažeme příkazem `rm` (té operaci se říká také unlink – maže se jenom jeden link), původní data se nemění a vytvoří se nový prázdný soubor. Důsledkem je, že původní data jsou pořád přístupná přes zbývající linky, pokud nějaké existují. Pokud se soubor předem neunlinkuje, přepíše se původní obsah a změna se projeví při přístupu přes libovolný link.

Na tomto místě bychom rádi připomněli, že inode je datová struktura, která si pamatuje metadata souboru a odkazy na jeho datové bloky. Link ukazuje na inode. Je to záznam v adresáři, který má jméno (jméno souboru) a číslo inode souboru. Data adresáře se vlastně skládají jen z takových záznamů. V těchto pojmech jste měli v řešení dost zmatku.

Úkol 4 – Nuceně privátní home

Při klasickém umístění domovských adresářů, tedy v `/home/<uživatel>`, je úloha představenými prostředky neřešitelná. Vlastník souboru totiž vždycky může měnit jeho práva, takže bychom museli buď měnit práva adresáři `home` a odstřihnout tak přístup do vlastních domovských adresářů ostatním uživatelům, nebo používat nějaký speciální mechanismus. Norma POSIX ale žádný neposkytuje, jen na souborovém systému ext2 (a vyšších) jste objevili příkaz `chattr`, který si dovolím ignorovat jako těžce neportabilní.

Nemusíme se ale tak moc omezovat, v `/etc/passwd` je možné cestu k domovskému adresáři nastavit. To je dobré vědět, abyste se nikdy na klasické umístění domovského adresáře nespoléhali a vždycky ve skriptech používali expanzi vlnky (`~`, `~hroch`, ...) nebo obsah proměnné prostředí `HOME` (`$HOME` expanduje na cestu k domovskému adresáři aktuálního uživatele).

Potřebný nápad je použití nadřazeného adresáře pro kontrolu přístupu. Root si může přivlastnit adresář nadřazený domovskému adresáři hrocha, sebrat práva světu (others), jako skupinového vlastníka nastavit skupinu obsahující jen hrocha a této skupině dát právo k prohledávání (x). Nikdo jiný než root a hroch tak nemá možnost se k hrochovu domovskému adresáři dostat, ať se snaží sebevíc.

Budeme tedy potřebovat skupinu, ve které je hroch sám. Často taková skupina již existuje a jmenuje se `hroch`. Pokud neexistuje, snadno ji vytvoříme:

```
groupadd hroch
usermod -aG hroch hroch
```

Teď už jen vytvořit „neprůhledný“ adresář a přesunout do něj hrochův stávající domovský adresář:

```
usermod -md /home/hroch_priv/hroch hroch
chown root:hroch /home/hroch_priv
chmod 750 /home/hroch_priv
```

Úkol 5 – Preprocessing

Od pohledu chceme použít jeden cyklus přes řádky a jeden cyklus přes tokeny na řádku. Když na řádku najdeme token `COMMENT`, zahodíme všechno až do konce řádku – tomu odpovídá příkaz `break` ve vnitřním cyklu. Token `BYE` odpovídá volání `return` – funkce v ten okamžik má skončit a víc ze vstupu nečíst.

Budeme číst zdrojáky, takže se nám bude hodit přepínač `-r` příkazu `read`. Jinak by zpětná lomítka dělala neplechu.

Při volání funkce se hodí možnost mít lokální proměnné. Třeba když měníme proměnnou `IFS`, není pěkné ji nechat změněnou po návratu z funkce, jiné kusy kódu se pak mohou chovat neočekávaně. POSIX v tomto ohledu nabízí dvě řešení, obě dost špatná.

- Můžeme uložit starou hodnotu `IFS` do jiné proměnné, změnit `IFS` a nakonec vrátit `IFS` původní hodnotu. Na to se ale snadno zapomíná, funkci bývá mož-

né opustit více způsoby a po návratu z funkce zůstane nastavená zbytečná proměnná.

- Také můžeme celé volání funkce provádět v subshellu – stačí tělo funkce obalit do kulatých závorek a místo `return` volat rovnou `exit`. Nevýhody jsou nasnadě: pouští se další proces, nejde snadno nastavovat nelokální proměnné nebo ukončit shell.

Bash nabízí třetí řešení, které jiné shelly nemají. Tím je vestavěný příkaz `local`, o kterém se více dozvíte v `help local`. Používá se podobně jako `export` a příklad použití najdete ve vzorovém řešení.

Už ve druhém dílu seriálu jste potkali konstrukce `for`, `case` a podmínky. Podmínky jste se učili i pomocí `||`. Přidejme k tomu obsah třetího dílu a řešení je na světě.

Program (shell): `http://ksp.mff.cuni.cz/viz/27-3-7-preprocess.sh`

Zvolil jsem nejjednodušší realizaci smyčky přes tokeny. Nechal jsem shell pomocí `IFS` rozdělit řádku vstupu podle mezer a přes výslednou množinu slov jsem pustil for-cyklus. Ve složitějším programu by bylo nepříjemné, že se ve smyčce nejde rozumně podívat na tokeny před a za aktuálně zpracovávaným. To by se dalo napravit použitím jiné smyčky. Příkaz `read` může opakovaně odkousávat první token; smyčka se zastaví, když zbytek řádku ke zpracování už je prázdný.


```
while [ -n "$line" ]; do
        IFS=' ' read -r token line <<EOF
$line
EOF
        # tělo cyklu
done
```

Tuto syntaktickou vlastnost shellu možná vidíte poprvé. Slovo `<<EOF` je přesměrování vstupu. Vstup se bere z následujících řádek, až do slova `EOF`, které je úplně samo na řádku (nesmí předcházet ani bílé znaky!). Říká se tomu *here document*. Místo `EOF` je možné použít i jiný řetězec. Na začátku (za `<<`) může být v uvozovkách, v tom případě může obsahovat i mezery, na konci je ale vždycky bez uvozovek. Při použití jednoduchých uvozovek (`<<'EOF'`) se celý here document chová jako řetězec v jednoduchých uvozovkách, tedy se v něm neexpandují proměnné ani se neprovádí substituce příkazů.

Kdybychom se chtěli použití here document vyhnout, jde to, ale není to pěkné:

```
r() { IFS=' ' read -r token line; }
token="$(echo "$line" | { r; echo "$token"; })"
line="$(echo "$line" | { r; echo "$line"; })"
```

Opět si uvědomte, co se stane při vynechání složených závorek, co je za zádrhel s readem v pipeline, jak se v pipeline chová `exit`.

Trochu potíží nadělá ještě vynechávání prázdných řádků. Projděte si vzorový kód a rozmyslete si, že a jak funguje. Všimněte si použití podmínek a příkazů `true` a `false`. Podotýkám, že pro vypsání stringu bez konce řádku na `echo -n` všude spoléhat nejde. Příkaz `printf` je portabilní, chová se předvídatelně a POSIX ho doporučuje používat místo `echo`.

Úkol 6 – Ztabulkování /etc/passwd

KSP

O formátu `/etc/passwd` jste si měli přečíst v `man 5 passwd` nebo analogickém zdroji. Pomocí standardních prostředků si s úlohou můžete snadno poradit za použití `while`, `read`, `IFS` a `printf`.

Použití tabulátorů jako oddělovačů sloupců nestačí, protože pak se tabulka rozpadne, když se liší délky položek jednoho sloupce po vydělení osmi (rozestup tabulačních zarážek). Předpokládal jsem, že si víc přečtete o `printf` a jeho formátovacích direktivách, jak napovídalo i umístění úlohy. Najít jste potřebovali nastavení šířky, na jakou se string má doplnit mezerami. Na plný počet bodů stačilo maximální délku položky v každém sloupci odhadnout konstantou, obludně dlouhý login klidně smí tabulku rozbít. Šlo by šířku sloupců i počítat, ale to za druhý průchod souborem a dost kódu navíc nestojí.

řešení

```
tplt='%-13s\t%-1s\t%5s\t%5s\t%-35s\t%-25s\t%s\n'
while IFS=: read login passwd uid \
        gid name home shell; do
        printf "$tplt" \
                "$login" \
                "$passwd" \
                "$uid" \
                "$gid" \
                "$name" \
                "$home" \
                "$shell"
done < /etc/passwd
```

Backslash na konci řádku dovolí v příkazu pokračovat na dalším řádku a logický řádek tak fyzicky rozdělit. Nesmí za ním být do konce řádku už nic jiného, ani bílé znaky.

Vzorové řešení už znáte, ale zmíním ještě řešení, kterým jste mě překvapili. Není portabilní, zato na úlohu padne jako ulité a za celkem rozšířené se rozhodně dá považovat. Je jím příkaz `column`, který slouží k výpisu dat do tabulky. Bez parametrů považuje vstup za seznam jednotlivých položek a řádek může ukončit mezi libovolnými dvěma z nich, s přepínačem `-t` respektuje řádkové zlomy ve vstupu a může nechávat v tabulce na koncích řádků prázdné buňky. Přepínač `-s` nastavuje oddělovač sloupců na vstupu a `-n` zakazuje ignorování prázdných buněk.

```
column -t -s: -n /etc/passwd
```

Vzorová řešení KSP – 3. série

Úkol 1 – Velikosti bloků

Jako poslední nám zbývá první úkol. Zadání se nepodařilo zformulovat tak, abyste se ho nesnažili řešit cestami, které vás protáhnou detaily současných disků. Seriál tím směrem nemířil, ale cestu do pekel si mnozí z vás našli sami.

Jak jsem doufal, že k řešení přistoupíte? Já bych si zakládal soubory různých velikostí a zkoumal na nich výstup příkazu `du` s přepínačem `-h` a bez něj. Vyrobit takový soubor velikosti N bytů můžeme třeba s pomocí přesměrování příkazem `head -c N /dev/zero > soubor`, jak bylo ukázáno v textu seriálu.

Pokud tedy nemáme zapnutý inlining, o kterém jsem také psal, vlastně stačí soubor velikosti 1: `echo > soubor`. Ten určitě bude zabírat nejmenší možné nenulové množství paměti, tj. jeden diskový blok. (Prázdný soubor by často zabral jen inode, který se do počtu zabraných bloků nepočítá.) S inliningem by bylo potřeba soubor postupně zvětšovat a někde ve větších velikostech hledat velikost, při které přidání jediného bytu zvětší počet bloků zabraných souborem. Pak by stačilo odečíst velikost před zvětšením.

Příkaz `du -h soubor` nebo lépe `du -B 1 soubor` nám ukáže, jak velký takový blok je. Když použijeme `du soubor` bez přepínačů, dozvíme se, kolik bloků soubor podle představ utility `du` zabírá, z čehož už snadno dopočítáme, jak velký blok utilita používá.

Na Linuxu to bývá 1024 B. Ovšem když nastavíme proměnnou prostředí `POSIXLY_CORRECT` (na libovolnou hodnotu), `du` se přizpůsobí požadavkům POSIXu a začne používat bloky o velikosti 512 B. Není bez zajímavosti, že tato proměnná mívala alias `POSIX_ME_HARDER`, což ilustruje názor autorů na některé části normy. Zajímavé čtení o této proměnné[61] najdete na stránce Waikatské skupiny uživatelů Linuxu.

Utilita `ls` tu vůbec nebyla potřeba. Spousta z vás se snažila využít její přepínač `-s` a zjišťovala tak velikost bloku používaného utilitou `ls` namísto toho používaného utilitou `du`.

Kdybychom chtěli vyloučit i možnost obskurní velikosti bloku, která není mocninou dvojky, můžeme pozorovat stejným způsobem velký soubor známé velikosti. Ve velkém počtu bloků by se i jedničkový rozdíl nasčítal, takže by počet bloků neseděl přesně.

Doplníme, že jeden diskový blok často mívá velikost 4 kB. Poučení z úlohy mělo být, že reálná velikost diskového bloku a velikost bloku používaná utilitami spolu nemají nic společného. Není blok jako blok.

Jenže se to zkomplikovalo, když jste začali hledat a používat utility, které nepracují s diskem přes souborový systém, ale koukají na něj přímo. A tak se přestaňme schovávat za představu ideálního světa, kde existuje něco jako diskový blok, a povězme si, jak se to s bloky na disku má doopravdy.

[61] `http://wiki.wlug.org.nz/POSIXLY_CORRECT`

Bloků existuje víc různých typů, konkrétně jste narazili na tři. Fyzický sektor, logický sektor a alokační jednotka (neboli cluster). Nakonec jako bonus přidám ještě IO blok.

Začněme od nejnižších vrstev abstrakce, u fyzického sektoru. To je blok, který interně používá pevný disk a jeho firmware. Vůbec by nás jako uživatele nemusel zajímat, kdyby jeho velikost nesouvisela s výkonem programů, které s diskem intenzivněji pracují. U starších disků má velikost 512 B, u novějších (od roku 2011) 4096 B. Formátu novějších sektorů se říká Advanced Format (AF) – pod tímto termínem vygooglíte víc.

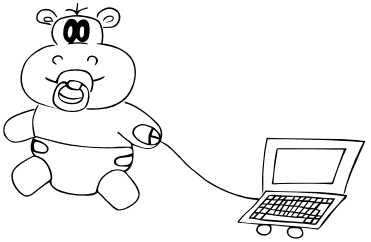
KSP

O vrstvu abstrakce výš leží logický sektor. V nich disk přemýšlí, když komunikuje se zbytkem počítače, typicky přes svůj ovladač v operačním systému. U starších disků se nelišil od fyzického, AF podle něj definuje dvě kategorie zařízení. První je 512e (e jako emulace) s 512B logickým sektorem, druhá 4Kn (n jako nativní) s 4096B logickým sektorem. Kategorie 512e se snaží být aspoň navenek zpětně kompatibilní, uvnitř už ale vyzvedává a ukládá celé 4kB fyzické sektory.

řešení

Teprve nad ovladačem disku sedí souborový systém a diskový prostor dělí na alokační jednotky, čili clustery. Po takových blocích přiděluje paměť jednotlivým souborům, a právě na ně se snažil první úkol seriálu mířit. Velikost alokační jednotky už není zadrátovaná v hardware, bývá možné ji nastavit při vytváření souborového systému, ale zejména kvůli efektivitě se obvykle volí jako celočíselný násobek velikosti sektoru.

Pro úplnost uvedeme ještě čtvrtý typ bloku, na který jste už nenarazili. Operační systém má nějaké povědomí o tom, jak by se na disk mělo přistupovat, s jak velkými bloky by se ideálně mělo pracovat najednou. Říká se jim IO bloky (I je input, O output). Dosud jsme psali především o klasických pevných discích. Pro ně mívají stejnou velikost jako alokační jednotka souborového systému, vnitřní struktura SSD disků ale vyžaduje IO bloky zpravidla mnohem větší, aby se zbytečně nepřepisovaly velké kusy paměti a nezkracovala se tak životnost zařízení. Program intenzivněji pracující s diskem si pak může přečíst, po jakých blocích je dobré číst, nebo si to může nechat nadiktovat od uživatele, jako to dělá třeba utilita `dd` pro čtení a zápis po blocích. Je prastará a na nestandardní syntaxi jejích parametrů je to vidět, nenechte se vyděsit. :-)



Když se začnete vrtat ve specifikách jednotlivých souborových systémů, najdete džungli specialit, na které běžně ve svém počítači nenarazíte. Více detailů si v zájmu zachování vašeho duševního zdraví dovolíme nerozebírat. Pokud si ho neváţíte, koukněte na pojmy block suballocation, sparse file, inlining nebo na souborový systém jménem ZFS.

Další debatu s vámi rád povedu na fóru, ať už se bude týkat vašich řešení, tohoto textu, UNIXu nebo disků a souborových systémů.

Samé příjemné zážitky s UNIXem přeje

Tomáš „Palec“ Maleček

27-4-1 Zadávání úkolů

Napřed uděláme jedno jednoduché pozorování. Kdybychom věděli, kolik úkolů přes konkrétního zaměstnance projde, je snadné určit, komu bude posílat další. Pokud prošel sudý počet úkolů, bude to stejný podřízený jako na začátku, v lichém případě to bude ten druhý.

Stejně tak lze snadno určit, kolik úkolů takový vedoucí předá kterému podřízenému. Pokud je počet sudý, oba podřízení dostanou polovinu. V lichém případě dostane ten začínající o jeden úkol více.

KSP

Jak tedy určit, kolik úkolů skrz kterého vedoucího projde? Začneme u ředitele. Ten svůj počet úkolů ví, máme ho tedy vyřešeného. Jeho přímým podřízeným přidáme úkoly. Poté si vybereme některého zaměstnance, který už všechny úkoly od svých nadřízených dostal; už tedy také ví, co ho čeká a nemine. Opět rozdělíme jeho úkoly a opět si vybereme někoho, kdo má všechny své nadřízené vyřešené. To děláme tak dlouho, dokud ještě někdo zbývá. Zjednodušeně, vedoucí začne rozdělovat, až když dostane všechny své úkoly.

řešení

A proč se nám nemůže stát, že sice máme ještě nějaké nevyřešené zaměstnance, ale všichni mají nějakého svého nadřízeného ještě nevyřešeného? Pro spor si představme, že se nám přesně taková nepříjemná věc stala. Vezměme tedy libovolného nevyřešeného zaměstnance. A z něj postupme do některého jeho nevyřešeného nadřízeného – takový musí z definice této nepříjemné situace existovat. A u toho nadřízeného si zase vybereme některého jeho nevyřešeného nadřízeného. Takto budeme pokračovat „nahoru“ v hierarchii, ale nikdy neskončíme. Zaměstnanců však musí být konečný počet (zřejmě v dané společnosti mohou pracovat maximálně všichni lidé na Zemi), musíme tedy jednou navštívit některého vícekrát. To ale znamená, že je v grafu cyklus, a ten máme zadáním zakázaný.

Nyní nám tedy zbývá rozmyslet, jak to napsat s co nejlepšími složitostmi. Napřed si spočítáme, kolik má který zaměstnanec nevyřešených nadřízených. Všichni začnou na nule, projdeme všechny vedoucí a každému podřízenému vždy přičteme jedničku. To zvládneme v konstantním čase na každého vedoucího. Založíme si skladiště na zaměstnance bez nevyřešeného nadřízeného (třeba zásobník, ten je příjemně jednoduchý) a při dalším průchodu přes zaměstnance do něj nastrkáme všechny, kteří mají nulu (v dobře fungující společnosti by to měl být jen ředitel). To opět zvládneme v celkově lineárním čase.

Nyní opakovaně vyndáme zaměstnance ze skladiště, vyřešíme ho a oběma jeho podřízeným odečteme jedničku. Pokud číslo u některého z nich (nebo obou) klesne na nulu, přidáme ho do skladiště také. Vyřešení jednoho nám opět bude trvat konstantní čas.

Celkově tedy zvládneme celý výpočet v lineárním čase. Lépe to nepůjde, protože jen nastavení výsledku každého zaměstnance bude trvat takovou dobu.

Co se týče paměťové složitosti, potřebujeme si ke každému zaměstnanci zapamatovat konstantně mnoho informací a potřebujeme skladiště, do kterého uložíme každého zaměstnance maximálně jednou. Tedy si vystačíme s lineární paměťovou složitostí.

A pro znalé: ano, je to topologické třídění.

Program (C): `http://ksp.mff.cuni.cz/viz/27-4-1.c`

Michal „vorner“ Vaner

27-4-2 Čtverce v síti

Nejdříve se zamysleme nad triviálním řešením. Můžeme zkusit vzít každou čtveřici přímek a podívat se, jestli náhodou netvoří čtverec. To pro každou čtveřici zkontrolujeme snadno – ověříme, že dvě a dvě z nich jsou rovnoběžné, tyto dvojice rovnoběžek jsou na sebe kolmé, a navíc jsou od sebe stejně daleko. Pokud si potřebujete trochu připomenout základy analytické geometrie, nahlédněte do naší kuchařky o geometrii.[62]

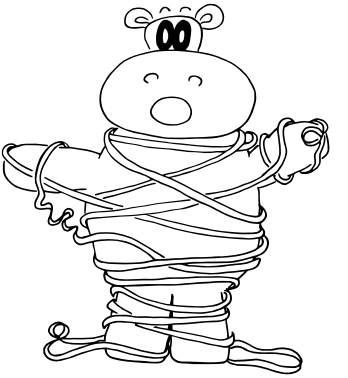
Takový přístup nám zabere celkově čas $\mathcal{O}(N^4)$. Neumíme to ale lépe? Již jsme si všimli toho, že čtverce tvoří vždy dvě dvojice rovnoběžek, toho jistě můžeme nějak využít. Tím nám ale vzniká nová otázka, a to jak rychle najít rovnoběžky mezi N přímkami v rovině.

Pokud bychom místo přímek měli třeba reálná čísla, stačilo by nám je v čase $\mathcal{O}(N \log N)$ setřídit, čímž by se stejné hodnoty dostaly k sobě, a pak bychom jedním lineárním průchodem našli všechny duplicity. Přímky sice nejsou reálná čísla, ale můžeme si je jimi popsat.

[62] `http://ksp.mff.cuni.cz/viz/kucharky/geometrie`

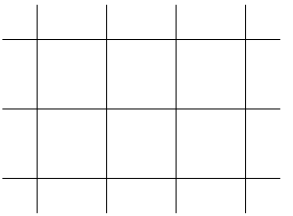
V této fázi nás zajímá jen směr přímek, ne jejich poloha, takže nám stačí namísto popisu celé přímky vzít její *směrový vektor* (tedy dvojici čísel (x, y) vyjadřujících směr přímky vůči souřadným osám). Směrové vektory bychom museli ještě „znormalizovat“, tedy upravit je všechny tak, aby třeba $x = 1$, čímž z nich vlastně získáme jedno reálné číslo, a to jejich *směrnici*.

Nyní si tedy můžeme všechny přímky v čase $\mathcal{O}(N \log N)$ setřídit, dostat tak rovnoběžky k sobě a pak takto setříděné přímky lineárně projít. Pro každou nalezenou skupinu rovnoběžek budeme chtít nalézt rovnoběžky na ně kolmé. To můžeme pokaždé dělat binárním vyhledáváním v čase $\mathcal{O}(\log N)$ na dotaz, nebo na to můžeme jít chytřeji.



Stačí nám držet si v setříděném seznamu přímek dva ukazatele posunuté od sebe o 90° a posouvat je oba najednou. Pokud nám druhý ukazatel skočí až na skupinu rovnoběžek, které ve směru otáčení svírají s rovnoběžkami na první pozici úhel větší než 90°, posuneme zase první ukazatel a tak stále dokola. Takto projdeme všechny skupiny na sebe kolmých rovnoběžek, a to v lineárním čase. Celkově nám to zabere čas $\mathcal{O}(N \log N)$ (kvůli třídění).

Pokud by vždy byly na sebe kolmé jen dvě a dvě rovnoběžky, bylo by ověření, jestli tvoří čtverec, triviální (jen bychom zkontrolovali jejich vzdálenosti, jestli jsou stejné). Za takové řešení jste mohli získat většinu bodů, ale ne všechny. Pro plný počet bodů bylo potřeba zamyslet se i nad situací, kdy se vyskytne velké množství na sebe kolmých rovnoběžek – třeba v případě přímek uspořádaných v pravoúhlé síti (jako na obrázku ze zadání níže).



Co dělat v takové chvíli? Bylo by možné vyzkoušet každou dvojici rovnoběžek z první skupiny s každou dvojicí rovnoběžek z kolmé skupiny. Ale to by nám vlastně zdegenerovalo až na naše původní triviální řešení v čase $\mathcal{O}(N^4)$. Nás však nezajímají jednotlivé přímky, ale jen vzdálenosti mezi nimi. V každé skupině si tedy vezmeme všechny možné vzdálenosti mezi přímkami (těch je K^2 pro K přímek, tedy $\mathcal{O}(N^2)$ pro nejhorší případ), ty si v čase $\mathcal{O}(N^2 \log N^2) = \mathcal{O}(N^2 \log N)$ utřiďme a pak tyto dvě setříděné posloupnosti porovnejme.

Stačí nám pro každou vzdálenost, která se vyskytne v jedné z posloupností, spočítat součin počtu výskytů této vzdálenosti v první posloupnosti a počtu výskytů této vzdálenosti ve druhé posloupnosti (součin proto, že čtverec tvoří každé dvě dvojice). Všechny tyto součiny sečteme a dostaneme tak počet vytvořených čtverců.

Pro případy, kdy se na vstupu vyskytne mnoho rovnoběžných přímek, umíme dosáhnout času $\mathcal{O}(N^2 \log N)$ s paměťovou složitostí $\mathcal{O}(N)$. Pro případy, kdy bude rovnoběžných přímek málo, se bude čas běhu našeho postupu blížit spíše $\mathcal{O}(N \log N)$.

Jirka Setnička

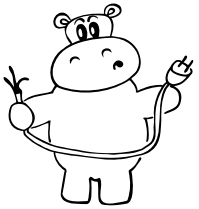
27-4-3 Vysoké napětí

Lehčí varianta

Většina z vás si všimla, že máme-li nějaké korektní řešení, můžeme vyrobit další správné řešení tím, že prohodíme všechna 100kV napětí v uzlech za 0kV a 100kV za 0kV. Z toho speciálně dostáváme, že si můžeme vybrat libovolný uzel a přiřknout mu hodnotu 0 kV. Existuje-li totiž nějaké korektní řešení, kde tento uzel má napěťovou hladinu 100 kV, existuje korektní řešení, kde má hladinu 0 kV.

Pak si stačí všimnout, že napěťová hladina v prvním uzlu jednoznačně určuje napěťové hladiny v sousedních uzlech, ty zase ve svých sousedech a tak dále, až se jednoznačně určí celý graf. Toto řešení můžeme tedy nalézt jednoduchým průchodem například do hloubky.

Vždy když zkoumáme nějaký uzel, který ještě nemá určenou napěťovou hladinu, určíme ji podle napěťové hladiny předchozího uzlu a rozdílu napětí na vodiči, jež tyto uzly spojuje – v případě, že rozdíl napětí byl 0 kV, budou napětí v uzlech stejná, pokud byl rozdíl 100 kV, budou napětí opačná. Pak začneme prohledávat všechny jeho sousedy.



Pokud zkoumaný uzel již má určenou napěťovou hladinu, jen zkontrolujeme, jestli tato hladina souhlasí s hladinou, kterou bychom jí jinak přiřkli. Jestliže nesouhlasí, dostáváme spor a graf nelze ohodnotit. Pokud souhlasí, tak je vše v pořádku (jeho sousedy již prohledávat nemusíme, neboť jsme je prohledali při první návštěvě tohoto uzlu).

Musíme si pamatovat celý graf a u každého uzlu a hrany konstantní množství informací, paměťová složitost bude tedy $\mathcal{O}(n + m)$, kde n je počet uzlů a m počet hran. Celý graf musíme načíst a pak pro každou hranu provedeme konstantní množství kroků (zpracování jednoho uzlu), dostáváme tedy opět $\mathcal{O}(n + m)$.

Program (C++): `http://ksp.mff.cuni.cz/viz/27-4-3a.cpp`

Tři napěťové hladiny

Dobrý nápad je využít řešení jednodušší varianty. Vezměme náš graf a odstraňme z něj všechny vodiče s rozdílem napětí 100 kV. Tímto se nám graf rozpadne na několik komponent. Smažme ty, které neobsahují žádný vodič s rozdílem 200 kV. Zůstavší komponenty vyřešíme podobně jako jednodušší variantu. Jen musíme místo 100 kV pracovat s 200 kV a také nám tu nastává ten problém, že máme více komponent. Musíme tedy prohledávání z jednodušší varianty postupně spouštět ze všech uzlů. Rozmyslete si, že to nám nijak nezhorší asymptotickou časovou složitost (většinou se spustíme na uzel s již určenou hladinou, a tedy hned skončíme).

Máme tedy určené napěťové hladiny všech uzlů sousedících s vodičem s rozdílem 200 kV a uzlů, které jsou z těchto uzlů jednoznačně určené. Zbývá tedy určit napěťové hladiny uzlů, které s žádným 200kV vodičem nesousedí, a zkontrolovat, jestli hladiny, které jsme určili, souhlasí s uzly k nim připojenými 100kV vodičem (0kV vodiče řešit nemusíme, neboť ty jsme řešili již v prvním kroku).

Rozmysleme si, že ohodnocené komponenty jsou navzájem spojeny cestičkami z vodičů s rozdílem 100 nebo 0 kV, kde navíc první a poslední vodič v každé cestičce je 100kV. Uzel na druhém konci tohoto vodiče musí mít hladinu 100 kV, protože výchozí uzel má hladinu buď 200 kV, nebo 0 kV. Kromě toho musíme ještě zkontrolovat, že toto přiřazení nenarušilo dosavadní přiřazení hladin (to by nastalo v případě, že by byly dvě komponenty spojeny právě jedním vodičem), pak by řešení neexistovalo.

Nyní si stačí dočasně odmyslet již ohodnocené komponenty (až na ty poslední uzly s hladinou 100 kV). Zbudou nám tedy pouze uzly s hladinou 100 kV a vodiče s rozdíly napětí 0 kV a 100 kV. Nabízí se tedy opět využít řešení jednodušší varianty. Spor s dosud ohodnocenými komponentami nám určitě nenastane, neboť s nimi jsme spojeni pouze přes původní 100kV uzly. Takto tedy dojdeme ke sporu a zjistíme, že řešení neexistuje, nebo nalezneme korektní řešení.

Opět si stačí pamatovat graf, takže paměťová složitost bude $\mathcal{O}(n+m)$. Co se týká časové, tak nejprve provedeme jednou jednodušší variantu na ohodnocení části grafu, poté v lineárním čase označíme nějaké 100kV uzly a poté znovu provedeme variantu jednoduššího algoritmu. Opět tedy dostaneme časovou složitost $\mathcal{O}(n+m)$.

Program (C++): `http://ksp.mff.cuni.cz/viz/27-4-3b.cpp`

Dominik Smrž

27-4-4 NP-úplný hlavolam

Jak dobře víte z kuchařky, důkaz $\mathcal{NP}$-úplnosti obvykle sestává ze dvou kroků – důkazu toho, že problém leží ve třídě $\mathcal{NP}$, a převodu některého problému, o kterém již víme, že je $\mathcal{NP}$-úplný, na tento náš problém.

První krok je v našem případě velice snadný. Jako certifikát použijeme seznam sloupců, pod které jsou zasunuty barevné proužky. Ověření certifikátu určitě v polynomiálním čase zvládneme, stačí totiž pro každý řádek přímočaře spočítat, kolik barevných polí je vidět.

Druhý krok lze provést více způsoby. Pokud jste pečlivě prohledávali informatickou literaturu (či Wikipedii), mohli jste zjistit, že náš problém (přirozeně trochu jinak formulovaný) lze najít už ve slavné knize *Computers and Intractability: A Guide to the Theory of NP-Completeness* pánů Gareyho a Johnsona z roku 1979 pod kódem LO4.

Zde si předvedeme převod z problému Trojbarevnosti grafu. Jeho znění pro jistotu připomínáme.

Název problému: Trojbarevnost grafu

Vstup: Neorientovaný graf.

Problém: Lze vrcholy tohoto grafu obarvit třemi barvami tak, že každá hrana sousedí s vrcholy dvou různých barev? (V obarvení musí mít každé dva sousední vrcholy různou barvu.)

Nechť $G = (V, E)$ je neorientovaný graf, jehož trojbarevnost máme rozhodnout. Značíme $n = |V|$ a $m = |E|$. Naším cílem je konstrukce hlavolamu, který má řešení právě tehdy, když je graf G trojbarevný. Tento hlavolam bude mít celkem $3n + 3m$ sloupců. Pro každý vrchol $v \in V$ mějme tři sloupce $č_v$, m_v a z_v. Pro každou hranu $e \in E$ a každou barvu $b \in \{č, m, z\}$ sloupec $s_{e,b}$.

Nejprve zajistěme, že každé řešení hlavolamu vůbec reprezentuje nějaké obarvení grafu. Pro každý řádek přidáme řádek, který vynucuje, že právě jeden ze sloupců $č_v$, m_v a z_v je barevný. Jak poznáme, jakou barvu má vrchol v? Podíváme se, který ze sloupců $č_v$, m_v a z_v je barevný.

Zbývá zajistit, aby každé řešení hlavolamu reprezentovalo obarvení, ve kterém mají sousední vrcholy různé barvy. Pro každou barvu $b \in \{č, m, z\}$ a každou hranu $e = \{u, v\}$ přidáme řádek, který říká, že právě jeden ze sloupců b_u, b_v a $s_{b,e}$ je barevný. Podotkněme, že sloupec $s_{b,e}$ má úlohu ryze pomocnou – umožňuje nám říci „nejvýše jeden ze sloupců b_u a b_v je barevný“.

Každé řešení hlavolamu tudíž odpovídá správnému trojobarvení. Naopak i pro každé správné trojobarvení, jak si snadno rozmyslíte, lze nalézt odpovídající řešení zadaného hlavolamu.

Náš převod je tedy korektní a zřejmě je možné jej realizovat v polynomiálním čase. Důkaz je hotov.

Lukáš Folwarczný

27-4-5 Večeře pro opraváře

V úloze procházíme čtvercovou síť a cílem je najít nejkratší cestu, která vede přes všechny vyznačené hospody a restaurace, těch je maximálně $n \leq 20$. Na zadání se podíváme jako na úplný ohodnocený graf s n vrcholy reprezentujícími hospody. Ohodnocení získáme jako vzdálenosti mezi hospodami ve čtvercové síti, které můžeme získat spuštěním průchodu do šířky z každé hospody zvlášť. Určitě se nám nevyplatí používat jiné než tyto nejkratší cesty, protože mezi dvěma hospodami se vždy vyplatí jít přímo bez jakéhokoliv odbočování. Toto převedení zvládneme v čase $\mathcal{O}(n \cdot WH)$, kde W a H značí šířku a výšku mapy.

KSP

Teď stačí najít nejkratší cestu v úplném ohodnoceném grafu, která každý vrchol navštíví právě jednou. Tento problém je známý pod jménem „Problém obchodního cestujícího“ (TSP) a je $\mathcal{NP}$-úplný. To znamená, že není známý žádný polynomiální algoritmus, který by jej řešil. My si ukážeme algoritmus, který jej řeší v čase $\mathcal{O}(n^2 2^n)$ a v prostoru $\mathcal{O}(n2^n)$, což je pro $n \leq 20$ dostačující.

řešení

Úloha by se dala řešit obyčejným backtrackem. Víme, v jakém vrcholu aktuálně stojíme (pro počáteční vrchol zkusíme všechny možnosti) a které vrcholy jsme již navštívili. Pro další v pořadí postupně zkusíme všechny možnosti nenavštívených vrcholů, přesuneme se tam a pokračujeme v backtracku z nich. Takové řešení by ale mělo časovou složitost až $\mathcal{O}(nn!)$, protože vlastně postupně zkoušíme všechny permutace vrcholů.

My ale toto řešení dokážeme vylepšit. Stačí nám do backtracku přidat ještě myšlenku dynamického programování: Při každém volání backtrackové funkce se stejnými parametry (aktuální vrchol, množina navštívených vrcholů), musíme nutně dostat i stejný výsledek. Tedy jakmile jednou výsledek pro konkrétní parametry spočítáme, tak si jej uložíme do paměti a při dalším volání jej v konstantním čase vrátíme. Počet možností pro aktuální vrchol je n a počet možností pro množinu navštívených vrcholů je 2^n (každý vrchol v ní buď je, anebo není). Celkově tedy backtracková funkce proběhne maximálně $\mathcal{O}(n2^n)$-krát.

Zbývá určit čas, který nám zabere jedno volání backtrackové funkce. Ta jen pro každý z n vrcholů zkontroluje, zda je v množině již navštívených vrcholů a pokud ne, tak v něm rekurzivně zavolá backtrack. Pokud testování, zda vrchol je součástí množiny, zvládneme v konstantním čase, dostáváme se na časovou složitost $\mathcal{O}(n^2 2^n)$ a paměťovou $\mathcal{O}(n2^n)$.

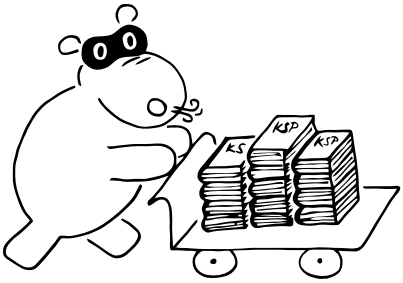
Jelikož vrcholů je maximálně 20, můžeme si jakoukoliv podmnožinu vrcholů reprezentovat jako n-bitové číslo. Pak, pokud i-tý bit má hodnotu 1, vrchol je v podmnožině a pokud 0, vrchol v ní není. Přidávání do množiny a zjišťování, jestli v ní vrchol je, tedy zvládneme v konstantním čase pomocí bitových operací, vizte vzorový kód.

Program (C++): `http://ksp.mff.cuni.cz/viz/27-4-5.cpp`

Karel Tesař

27-4-6 Stěhování pásek

Jak jste možná poznali, přesouvání kotoučů ze zadání odpovídá slavné úloze o Hanojských věžích, kterou poprvé popsal francouzský matematik Édouard Lucas.[63] Pro úplnost dodejme, že v původní podobě úlohy se místa `A`, `B` a `C` nazývají tyčemi, kotoučů je celkem čtyřiašedesát, všechny kotouče jsou z ryzího zlata a na začátku jsou umístěny na jedné tyči. Mnichové každý den jeden kotouč přesunou z tyče na tyč (stále platí, že nesmí být větší kotouč položen na menší). V okamžiku, kdy se jim podaří přemístit všechny kotouče na třetí tyč, nastane podle legendy konec světa.



Než se pustíme do samotného uspořádávání fotografií pořízených při stěhování dat, potřebujeme porozumět tomu, jak se kotouče správně přesouvají. Konkrétně si ukážeme, jak lze přemístit všechny kotouče na co nejmenší počet tahů, a společně s tím dokážeme, že se jedná o jediný možný postup s nejmenším počtem tahů. To nám dá jistotu, že tentýž postup použili i pracovníci vystupující v zadání úlohy. Celý postup popíšeme rekurzivně (pokud si s rekurzí příliš nerozumíte, můžete v případě nesnází nahlédnout do naší kuchařky o základních algoritmech).[64] Celkově máme N kotoučů, očíslujeme si je od nejmenšího po největší čísly 1 až N.

Funkce *přesun(k, výchozí, cílová, pomocná)*:

1. Pokud $k = 0 \rightarrow$ skonči
2. Zavolej *přesun*($k-1$, *výchozí, pomocná, cílová)*
3. Přesuň kotouč k z tyče *výchozí* na tyč *cílová*
4. Zavolej *přesun*($k-1$, *pomocná, cílová, výchozí)*

[63] Zadání úlohy z pera autora si můžete přečíst i nyní. Je sepsáno ve třetím svazku jeho díla *Récréations mathématiques* na straně 55, dostupno on-line: `https://archive.org/details/recretionmatedou03lucarich`

[64] `http://ksp.mff.cuni.cz/viz/kucharky/zakladni-algoritmy`

Funkce *přesun(k, výchozí, cílová, pomocná)* předpokládá, že na tyči *výchozí* se nachází kotouče $1, 2, \ldots, k$, a zařídí přesun všech těchto kotoučů na tyč *cílová*. Nyní pomocí matematické indukce dokážeme, že nejmenší možný počet tahů pro přesunutí všech N kotoučů je $2^N - 1$ a lze jej dosáhnout pouze pomocí našeho postupu.

Když nemáme žádné kotouče, přesuneme je pomocí nula tahů a je to jediný možný způsob přesunu (nebo spíš nepřesunu). Platí rovnost $2^0 - 1 = 0$ a základ indukce je ověřen.

Předpokládejme nyní, že jsme již toto tvrzení dokázali pro $N = k - 1$. Dokažme jej pro $N = k$. Klíčové je následující pozorování: v okamžiku, kdy je přesouván největší kotouč mezi dvěma tyčemi, musí být všech $k - 1$ ostatních kotoučů umístěno na třetí tyči. Z předpokladu víme, že jediný způsob, jak přesunout $k - 1$ kotoučů z výchozí tyče na pomocnou tyč na co nejmenší počet tahů, je použít náš postup, který potřebuje $2^{k-1} - 1$ tahů. V dalším kroku je přesunut největší kotouč na cílovou tyč a zbývá na tuto tyč přesunout zbylých $k - 1$ kotoučů. Opět díky předpokladu víme, že na nejmenší počet tahů toho docílíme jedině s použitím našeho postupu. Ukázali jsme tak, že v případě $N = k$ náš postup potřebuje $2(2^{k-1} - 1) + 1 = 2^k - 1$ tahů a každý jiný postup by tahů potřeboval víc.

Když už rozumíme tomu, jak kotouče přesunovat, můžeme se pustit do samotného řazení fotografií. Postupně budeme rekonstruovat pro zadané fotografie, v jaké fázi přesunu se nacházíme. Na začátku si fotografie můžeme rozdělit do dvou přihrádek podle toho, kde se nachází největší kotouč. Pokud se nachází na výchozí tyči, znamená to, že teprve přesouváme menší kotouče z výchozí na pomocnou tyč; takové fotografie určitě předchází všechny fotografie, u kterých se největší kotouč nachází na tyči cílové, tam už jsme ve fázi přesunu menších kotoučů z pomocné na cílovou tyč. Podobný postup můžeme zopakovat pro uspořádání každé z hromádek, když se podíváme na polohu druhého největšího kotouče. Tento postup lze opakovat, dokud nesetřídíme všechny fotografie. Celou proceduru shrnuje pseudokód.

Funkce *uspořádej(k, fotografie, v, c, p)*:

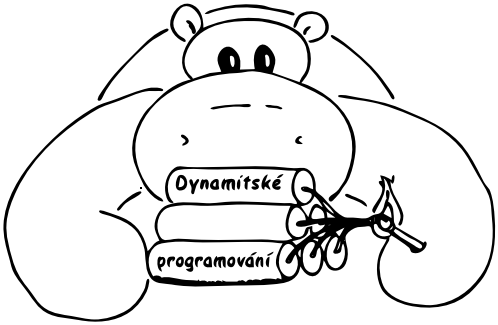
1. Pokud $k = 0 \rightarrow$ skonči
2. Pro každou fotografii z pole *fotografie*:
3. Pokud se k-tý kotouč nachází na tyči v, umísti fotografii do pole h_1
4. Jinak umísti fotografii do pole h_2
5. *seřazení1* = *uspořádej*($k - 1$, h_1, *v, p, c*)
6. *seřazení2* = *uspořádej*($k - 1$, h_2, *p, c, v*)
7. *Výstup:* Zřetězení *seřazení1* a *seřazení2*

Uspořádání všech fotografií docílíme následujícím zavoláním: *uspořádej(N, fotografie, A, B, C)*, kde *fotografie* označuje pole všech fotografií.

Celkový počet fotografií označme jako F. Každá fotografie je v průběhu řazení zpracována celkem N-krát. Pokud bychom proceduru implementovali doslova podle pseudokódu, dosáhli bychom složitosti $\mathcal{O}(FN^2)$, protože bychom pokaždé potřebovali čas $\mathcal{O}(N)$ na zpracování jedné fotografie, například při umisťování do přihrádek. Nám ovšem stačí pracovat pouze s odkazy na fotografie, což nám dá výslednou časovou složitost $\mathcal{O}(FN)$. Paměťová složitost je přirozeně taktéž $\mathcal{O}(FN)$.

Program (C): `http://ksp.mff.cuni.cz/viz/27-4-6.c`

Lukáš Folwarczný



27-4-7 Nástroj pro zpracování textu

Seriál opět patřil mezi vaše oblíbené úlohy, patřil mezi tři úlohy s nejvíce odevzdáními. Proto doufáme, že se ze řešení naučíte třeba nové triky a že s námi zůstanete i v poslední sérii.

Úkol 1 – Náhodné dvojice

Většina řešitelů se shodla na základním postupu: vstupní soubor zamícháme (pomocí `shuf`) a poté spárujeme sousední dvojice řádek. Ukážeme si hned několik způsobů, které se mezi řešeními objevily.

Asi nejjednodušší na vymyšlení je použití bashového cyklu a `read`ů:

```
shuf | while read a; do
           read b
           echo "$a:$b"
       done
```

O něco elegantněji a stále jednoduše se dá využít `sed`u:

```
shuf | sed -re 'N; s/\n/:/;'
```

Jak již víme, příkaz `N` načte další řádek a přilepí ho do pattern space oddělený znakem `\n`. Ten stačí nahradit dvojtečkou a jsme hotovi. Při příští iteraci se pokračuje nejbližším ještě nezpracovaným řádkem, tedy třetím, pak pátým, atd.

Velmi podobná věc se dá udělat i pomocí `awk` (inspirováno řešením Štěpána Hudečka):

```
awk '{ printf "%s:", $0; getline; print; }'
```

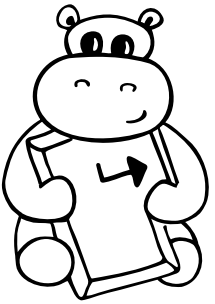
A na závěr jedno mile bláznivé řešení podle Jakuba Tětka:

```
shuf | awk '{ ORS = NR%2 ? ":" : "\n" } 1'
```


Zkuste schválně vymyslet, co dělá ta jednička na konci ;-).

Tato úloha byla původně zamýšlena jako cvičení na `paste`, leč ukázalo se, že naše řešení je výrazně složitější než ta ukázaná výše. Základní myšlenka je zamíchat vstup, rozdělit do dvou souborů jeho první a druhou polovinu, a ty poté spojit do dvojic právě pomocí `paste`:


```
shuf >jmena.rand
half=$(( $(wc -l jmena.rand) / 2 ))
head -n $half >prvni.tmp
tail -n $half >druzi.tmp
paste -d: prvni.tmp druzi.tmp
```



Úkol 2 – Převrácená slova

Nejprve vyřešíme nalezení vyhovujících slov, výběr nejdelšího doplníme na konci. Dobrým začátkem určitě bude vytvořit si soubor s převrácenými verzemi všech slov ze slovníku, což nám zařídí onen podivný příkaz `rev`. Snadno si rozmyslíte, že hledaným řešením jsou právě slova, která se vyskytují jak v původním slovníku, tak v tomto pomocném souboru. K hledání průniku dvou souborů přímočaře poslouží příkaz `comm` – jen si nejdřív oba musíme setřídit. Řešení by mohlo vypadat takto:

```
sort -u slovnik >slovnik.sort
rev slovnik | sort -u >slovnik.rev
comm -12 slovnik.sort slovnik.rev
```

Někteří řešitelé přišli s alternativním postupem, který namísto `comm` používá příkaz `uniq`. Hledání průniku můžeme provést taky tak, že oba soubory (původní slovník a jeho reverzi) slepíme do jednoho a z něj vybereme všechny duplicitní řádky. K tomu lze (po setřídění) použít příkaz `uniq -d`, který přikazuje vypisovat pouze řádky s více než jedním opakováním.

Alternativně lze použít `uniq -c` a z výsledku odgrepovat všechny řádky s jedničkou na místě počtu opakování. Za předpokladu, že původní slovník neobsahoval duplicity, jeden z výskytů duplicitního řádku musel pocházet z jednoho souboru a druhý z druhého, tedy patří do průniku. A případných duplicit se snadno na začátku zbavíme příkazem `sort -u`.


```
{ sort -u slovnik; rev slovnik | sort -u; } \
        | sort | uniq -d
```

Nyní k výběru nejdelšího. To snadno vyřešíme jednoduchým skriptem pro `awk`, který bude v nějakých proměnných průběžně udržovat dosud nejdelší slovo a jeho délku a na konci ho jen vypíše:


```
awk '{
         if (length > maxlen) {
             maxlen = length;
             word = $0;
         }
     }
     END { print word; }'
```

Pokud vám přijde toto řešení příliš „céčkové“, dá se postupovat i jinak. Necháme `awk` jen připsat ke každému řádku jeho délku a pak využijeme příkazu `sort`. Takové řešení má sice složitost $\mathcal{O}(N \log N)$ namísto lineární, ale to nás v praxi příliš netrápí.

```
awk '{print length,$0}' | sort -nr | head -n 1 \
        | cut -d' ' -f2-
```

Úkol 3 – Seriály

Tato úloha byla spíš technickým cvičením a asi se nedá vymyslet úplně hezké a elegantní řešení. K úloze lze přistoupit ze dvou stran:

a) Projít soubory v cílovém adresáři, z každého z nich zkusit vytáhnout správná čísla, najít informace v seznamu epizod a přejmenovat ho.

b) Projít stažený seznam epizod, pro každou se pokusit najít odpovídající soubor a přejmenovat ho.

Většina řešitelů se přiklonila k první variantě. Ta se ale ukázala poměrně nebezpečnou, neb téměř nikdo neřešil, co se stane se soubory, jejichž název není ve správném formátu, případně pokud příslušná epizoda není nalezena v seznamu. Ve většině případů to skončilo tím, že se čísla série a epizody naparsovaly

jako prázdné řetězce, název se v seznamu nenašel a z toho vznikl název typu `S00E00_.avi`. Pokud by bylo v adresáři nerozpoznaných souborů víc, všechny byly přejmenovány na tento stejný název, čímž se navzájem přepsaly. Například většina skriptů byla ochotna takto přejmenovat i sebe sama, pokud si je člověk uloží do stejného adresáře, případně i své pomocné soubory.

Tento úkol mě přesvědčil, že opravování KSP je riziková činnost. Jeden z došlých skriptů totiž začal takovýmto způsobem přepisovat soubory v mém domovském adresáři. Za to mohl (společně s problémem popsaným výše) nevinně vypadající příkaz na začátku skriptu:

KSP

```
cd $1
```

Pokud takovýto skript omylem spustíte bez parametrů, dle pravidel bashové expanze se `$1` zcela zahodí a spustí se příkaz `cd` bez parametrů, který skočí do domovského adresáře uživatele.

Samozřejmě všechny tyto problémy se dají ošetřit (testováním, zda soubory mají správný formát názvu, přidáním uvozovek na vhodná místa, použitím `mv -i`, atd.), ale je to docela otrava a není jednoduché na nic nezapomenout. Proto si raději ukážeme druhý způsob, který těmito neduhy netrpí.

řešení

Celé řešení může vypadat třeba takto:

```
w='http://en.wikipedia.org/wiki/'
for S in $(seq 1 8); do
  art="The_Big_Bang_Theory_(season_$S)"
  curl -s "$w/$art?action=raw"            \
    | grep -E '\|(EpisodeNumber2|Title)'  \
    | cut -d= -f 2                        \
    | sed -re 's/^ +//'                   \
          -e 's/^\[\[.*\|(.*)\]\]$/\1/;' \
    | tr -d '[]' | tr ' ' .               \
    | while read E; do
       read title
       newfn="$(printf 'S%02dE%02d.%s.avi' \
                   "$S" "$E" "$title")"
       re="[^0-9]0*$S[^0-9]+0*$E[^0-9].*\.avi"
       oldfn="$(ls |grep -E "$re" |head -n1)"
       if [ -n "$oldfn" ]; then
         mv -vi "$oldfn" "$newfn"
       fi
     done
done
```

První část (před `while`) stáhne seznam epizod a převede jej do zpracovatelného formátu. Po `grep`u vypadá seznam takto:

```
|EpisodeNumber2  = 1
|Title           = [[Pilot (TBBT)|Pilot]]
|EpisodeNumber2  = 2
|Title           = The Big Bran Hypothesis
...
```

Dvojité hranaté závorky značí odkaz na jiný článek, kterého se musíme zbavit. Má tvar `[[`*název článku*`]]` nebo `[[`*název článku*`|`*text odkazu*`]]`. Seznam sloužící jako vstup pro while cyklus vypadá po zpracování takto:


```
1
Pilot
2
The.Big.Bran.Hypothesis
...
```

Poté načítáme dvojice řádků stejným trikem jako v prvním úkolu. Pro každou epizodu pak sestavíme nový název a regex, kterému musí vyhovovat původní název (bohužel se nedá jednoduše matchovat pomocí wildcardů). U obojího si musíme dát pozor na úvodní nuly u čísla série a epizody. Původní názvy je mít mohou a nemusí, nové názvy by měly, kvůli správnému řazení. Poté se stačí jen podívat, jestli existuje soubor vyhovující danému regexu, a pokud ano, přejmenovat jej. Parametr `-v` informuje uživatele o tom, jaká přejmenování byla provedena, a `-i` předchází nechtěnému přepisování souborů.

Úkol 4 – Identifikátory

Tento úkol měl dvě netriviální části: odstranění řetězců a víceřádkových komentářů. Začněme řetězci. Na céčkový řetězec se dá dívat jako na posloupnost „elementů“, kde každý z nich je buď obyčejný znak, nebo escape sekvence. Escape sekvence jsou obvykle ve tvaru `\`*znak*, existují i složitější, ale ty si dovolíme ignorovat, princip je podobný. Lze tedy sestavit jednoduchý regex, kterému vyhovuje právě jeden céčkový řetězec, i když obsahuje escapované znaky, včetně uvozovek:

```
"([^\\"]|\\.)*"
```

Tento regex vlastně docela blízce simuluje to, jak skutečný céčkový překladač parsuje zdrojový kód. Rozmyslete si, že opravdu namatchuje, co má, bez ohledu na počet escapovaných uvozovek a zpětných lomítek, včetně případů jako `"\\\"\\"`. Někteří řešitelé přišli s o trochu méně elegantním, ale správným a myšlenkově jednodušším postupem: nejdřív odstraníme ze zdrojáku všechny escape sekvence (nemusíme kontrolovat, jestli jsou uvnitř řetězce, jinde se legálně vyskytnout nemohou) a poté už řetězce najdeme jednoduše:

```
sed -re 's/\\.//g; s/"[^"]*"//g;'
```

Nyní zbývá odstranění víceřádkových komentářů. To jde jednoduše vyřešit tak, že z celého zdrojáku uděláme jeden řádek a chováme se k nim jako k jedno-

řádkovým. Leč z cvičných důvodů jsme chtěli, abyste to vyřešili jinak, a na to se velmi dobře hodí pokročilý `sed`.

Ale ještě než se začneme starat o víceřádkovost, musíme vyřešit opačný problém: více komentářů na jednom řádku, např. takto:

```
int h /*pocet hrochu*/, b /*pocet bagru*/;
```

Ne, že bychom tento styl kódu doporučovali, ale náš skript by se jím neměl nechat zaskočit. Díky žravosti regexů nemůžeme napsat prostě `/\*.*\*/`, neb tomuto výrazu by vyhovovalo vše od prvního `/*` k poslednímu `*/`, tedy i celý úsek mezi komentáři obsahující platný identifikátor `b`. To můžeme vyřešit podobně jako u řetězců: zakázat výskyt ukončovacího oddělovače uvnitř komentáře. Tady je to jen trochu těžší v tom, že je dvouznakový. Dalo by se to udělat třeba takto:

KSP

```
sed -re 's#/\*([^*]|\*[^/])*\*+/# #g'
```

Nyní se zamysleme, co s víceřádkovými komentáři. Postup bude jednoduchý. Pro každý řádek opakujeme následující kroky, dokud se něco děje:

1. Odstraň všechny ukončené komentáře.
2. Dokud existuje neukončený komentář, smaž jeho obsah až do konce řádku a načti další řádek.

řešení

Do `sed`u to přeložíme takto:

```
sed -re '
: loop;
s#/\*([^*]|\*[^/])*\*+/# #g;
t reset
: reset
s#/\*.*$#/*#;
T;
N;
b loop;
'
```

Pokud neuspěje druhý příkaz `s`, znamená to, že na řádku není neukončený komentář. Všechny ukončené komentáře byly již odstraněny, takže s aktuálním řádkem jsme hotovi. Příkaz `T` v takovém případě skočí na konec skriptu, což způsobí vypsání aktuálního obsahu pattern space a přechod na další řádek. V opačném případě je příkazem `N` přilepen následující řádek na konec aktuálního a celý proces se opakuje. Pokud už jsme našli konec komentáře, odstraní se celý, jinak načítáme další řádky.

Dát to vše dohromady a přidat odstranění jednořádkových komentářů a klíčových slov by již mělo být jednoduché cvičení.

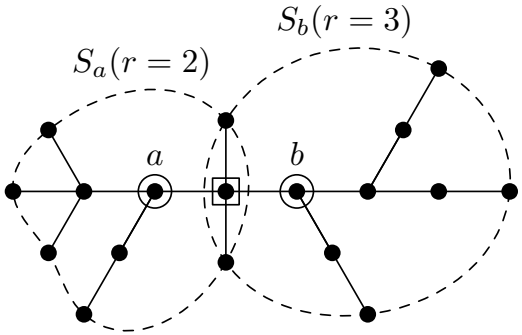
Filip Štědronský

27-5-1 Šíření poplašné zprávy

K postavení rychlého řešení si připomeneme něco o stromech. Kdybychom volali jenom do jedné kanceláře, hledali bychom *střed* stromu: vrchol, který má maximální vzdálenost do jiných vrcholů nejmenší. *Poloměr* stromu je „poloměr kružnice opsané“: největší vzdálenost mezi středem stromu a jinými vrcholy. Strom obsahuje jednu nebo více *nejdelších cest*. Nejdelší cesta jde najít v lineárním čase například některým z postupů v řešení úlohy 12-1-2.[65] Prostřední vrchol nejdelší cesty je střed stromu a poloměr stromu je polovina délky nejdelší cesty. (Pokud má nejdelší cesta liše hran, je poloměr ta větší polovina a střed není jednoznačný.)

Představme si teď, že ty dva vrcholy, do kterých je nejlepší zatelefonovat, už známe, a označme je jako a a b. Rozdělíme si vrcholy stromu na dva *sektory* S_a a S_b podle toho, ze kterého z vrcholů a, b dorazí signál dříve. Pokud někam dorazí z obou směrů zároveň, patří daný vrchol do obou sektorů. Místo, kde se sektory poprvé potkají, je buď společný vrchol, nebo jedna hrana.

Na následujícím obrázku je příklad rozdělení stromu na sektory s poloměry 2 a 3. Při takovémto rozdělení se poplašná zpráva rozšíří za 3 jednotky času. Snadno si rozmyslíte, že lépe to nejde. Přestože sektory mají složitější průnik, na nejdelší cestě (vodorovné) z něj najdete jen jediný bod (označen čtverečkem). Právě tomu budeme říkat hraniční bod. Kdybychom místo vrcholu b vyslali signál z jeho pravého souseda, dostaneme stejně dobré řešení, ale hranici mezi sektory bude tvořit hrana nejdelší cesty.



[65] http://ksp.mff.cuni.cz/viz/12-1-2/reseni

Ukážeme si nejdřív pomocné tvrzení: když si vybereme nějakou nejdelší cestu, pak v některém optimálním řešení na této nejdelší cestě leží hranice sektorů.

Poloměr S_a i S_b musí být nejvýše stejný jako poloměr celého stromu. V případě, že některý ze sektorů má stejný poloměr jako celý strom, tak víme, že optimální řešení by kromě zvoleného $\{a, b\}$ bylo například zavolat do středu celého stromu a jednoho z konců nejdelší cesty. Když si zvolíme tenhle pár kanceláří, dostaneme optimální řešení, ve kterém na zvolené nejdelší cestě leží hranice sektorů.

Když jsou poloměry S_a i S_b ostře menší než poloměr celého stromu, pak nemůže celá naše nejdelší cesta ležet uvnitř jednoho sektoru: nevešla by se tam, protože sektory mají menší poloměr než celý strom. Nejdelší cesta tedy leží v obou sektorech, a proto obsahuje jejich hranici.

Když teď víme, že na libovolné nejdelší cestě leží hranice optimálních sektorů, nějakou nejdelší cestu si vybereme a hranici tam zkusíme najít. Když jako hranici zkoušíme hranu, bude *levý* a *pravý* sektor tvořený půlkami stromu, které vzniknou po odebrání hrany. Když jako hranici zkoušíme vrchol, musíme se ještě rozhodnout, co uděláme s případnými hranami do vrcholu, které nevedou po nejdelší cestě. Podstromy, do kterých tyto hrany vedou, připojíme do obou sektorů: jestli je opravdu optimální rozdělit sektory tímto vrcholem, tak signál do vrcholu dorazí z obou stran ve stejnou chvíli, a proto ve stejnou chvíli dorazí i do ostatních hran pověšených na tento vrchol.

Každý levý sektor se tedy skládá z nějakého začátku nejdelší cesty a ze všech podstromů, které na této části nejdelší cesty „visí“. Délku začátku nejdelší cesty si označme i. Pro každou hodnotu i si spočítáme poloměr příslušného levého sektoru a označíme ho jako $R_1[i]$. Podobně pro všechny délky j pravého konce spočítáme poloměry pravých sektorů $R_2[j]$.

Když má nejdelší cesta ℓ vrcholů, projdeme všech $O(\ell)$ možných rozdělení na sektory a vybereme to, ve kterém bude obvolání celého stromu trvat co nejkratší čas. Dělit sektory můžeme buď v hraně, nebo ve vrcholu. Když dělíme v i-té hraně, bude nám obvolání celého stromu trvat čas $\max\{R_1[i], R_2[\ell - i + 1]\}$, a když v i-tém vrcholu, bude to trvat čas $\max\{R_1[i], R_2[\ell - i + 2]\}$.

Zbývá vyřešit, jak budeme počítat poloměry sektorů. Kdybychom se spokojili se složitostí $\mathcal{O}(N^2)$, stačilo by každý poloměr spočítat v lineárním čase. My však použijeme dynamické programování a dostaneme optimální čas $\mathcal{O}(N)$. Postup si ukážeme na levém sektoru. Vrcholy nejdelší cesty si očíslujeme zleva doprava.

Všimneme si, že nejmenší možný sektor je list, který pod sebou nemá žádné hrany, protože bychom jinak mohli o tuto hranu protáhnout nejdelší cestu. Obecněji: když si odmyslíme hrany v nejdelší cestě, tak v podstromu pod i-tým vrcholem sektoru nesmí být větev hlubší než i. Dále si všimneme, že nejdelší cesta uvnitř sektoru vede vždycky z nejhlubší větve pod jedním vrcholem nejdelší cesty do nejhlubší větve pod jiným vrcholem nejdelší cesty.

Místo poloměru sektoru budeme udržovat délku jeho nejdelší cesty, ze které jde poloměr spočítat podělením dvěma. Dynamické programování bude postupně k sektoru přidávat *segmenty*, které se skládají z přidaného vrcholu nejdelší cesty a nového podstromu pod ním. Poslednímu vrcholu nejdelší cesty, který jsme do sektoru přidali, říkejme *konec sektoru*. Nejdříve si pro každý vrchol na nejdelší cestě předpočítáme maximální hloubku jeho podstromů (když ignorujeme hrany nejdelší cesty) a uložíme je do pomocného pole A.

Dynamické programování bude udržovat:

- M: délku nejdelší cesty v zatím prošlém sektoru.
- D: délku nejdelší cesty z konce zatím prošlého sektoru.

Po přidání nového segmentu číslo i může M buď zůstat stejné, nebo můžeme zjistit, že nejdelší cesta do nového konce sektoru zleva plus nejhlubší větev pod novým koncem sektoru tvoří delší cestu délky $D + A[i] + 1$. Nová hodnota M tedy bude $\max\{M, D + A[i] + 1\}$. Nejdelší cesta z konce sektoru se buď prodlouží o jeden vrchol, nebo změní na nejhlubší cestu do stromu pověšeného pod novým vrcholem, proto D upravíme na $\max\{D + 1, A[i]\}$.

Náš algoritmus tedy najde nejdelší cestu, nad kterou dynamickým programováním spočítá poloměry levých a pravých sektorů, a nakonec najde nejlepší místo k rozdělení. Kanceláře, do kterých chceme poslat signál, pak můžeme dopočítat jako středy sektorů, na které strom rozdělíme. Stačí nám jen $\mathcal{O}(N)$ času i paměti.

Program (Python): `http://ksp.mff.cuni.cz/viz/27-5-1.py`

Michal „Prvák“ Pokorný

27-5-2 Survivalisté

KSP

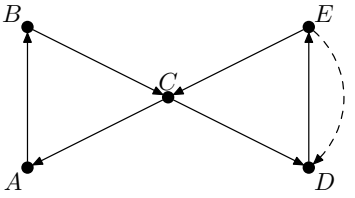
řešení

Zadání požaduje, aby každý člověk *právě* jednu věc dal jinému a *alespoň* jednu dostal. Ale snadno nahlédneme, že pokud jsou tyto podmínky splněny, musí každý i dostat právě jednu věc. V oběhu je N věcí (kde N je počet survivalistů), a pokud by někdo dostal dvě z nich, na jiného žádná nezbude.

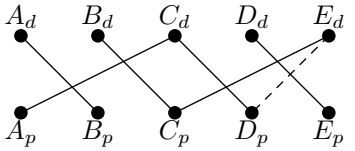
Chceme tedy vybrat nějaké dvojice (dárce, příjemce) takové, že každý je právě v jedné dvojici jako dárce a právě v jedné jako příjemce. To velice připomíná problém maximálního párování v bipartitním grafu. Bez znalosti tohoto pojmu úloha příliš řešit nešla, takže pokud jej potkáváte poprvé, nahlédněte do naší Encyklopedie.[66]

Náš graf sice není bipartitní, ale snadno z něj bipartitní vyrobíme. Od každého vrcholu (u) vyrobíme dvě kopie: jedna bude reprezentovat daného survivalistu v roli dárce (u_d), druhá jako příjemce (u_p). Každou hranu z původního grafu povedeme z odpovídajícího dárcovského do odpovídajícího přijímajícího vrcholu – tedy z původní hrany uv vytvoříme v novém grafu hranu $u_d v_p$. Tím nám přirozeně vznikne bipartitní graf s partitami dárců a příjemců.

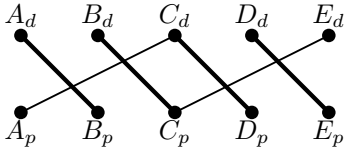
Například z grafu v zadání:



vznikne následující:



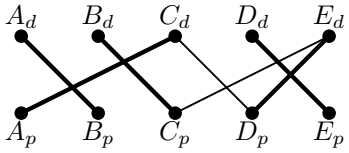
Maximální párování v tomto grafu (bez čárkované hrany) má velikost 4:



[66] `http://ksp.mff.cuni.cz/encyklopedie/parovani.html`

Každá hrana tohoto párování popisuje jednu předanou věc: například A předá něco B. V případě tohoto grafu požadavek ze zadání splnit nelze – E nic nedostane. Snadno si rozmyslíte, že zadání splníme právě tehdy, když jsou spárovány všechny vrcholy (takovému párování říkáme *perfektní*). Pokud perfektní párování existuje, určitě je maximální. Tedy není-li nalezené maximální párování perfektní, graf zadání nesplňuje.

Pokud zahrneme do vstupního grafu čárkovanou hranu, perfektní párování již existuje:



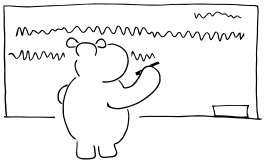
Graf s čárkovanou hranou tedy, jak už koneckonců víte ze zadání, požadavky splňuje.

Algoritmus bude vypadat tak, že v lineárním čase vytvoříme ze vstupu odpovídající bipartitní graf a spustíme na něj nějaký párovací algoritmus. Pokud je velikost nalezeného maximálního párování rovna počtu survivalistů, odpovíme „ano“, jinak odpovíme „ne“. Například při použití Hopcroftova-Karpova algoritmu[67] dosáhneme časové složitosti $\mathcal{O}(M\sqrt{N})$, kde M je počet hran a N počet vrcholů. Vystačíme si s lineární pamětí ($\mathcal{O}(N+M)$).

Program (Python): `http://ksp.mff.cuni.cz/viz/27-5-2.py`

Jako třešničku na dortu pro zkušenější řešitele ukážeme, že řešení pomocí párování je optimální. Použijeme k tomu stejný trik, jaký se používá při dokazování NP-úplnosti:[68] ukážeme, že lze problém perfektního párování v bipartitním grafu převést na řešení naší úlohy.

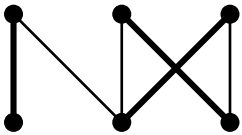
Předpokládejme, že máme zadaný nějaký bipartitní graf, ve kterém chceme najít perfektní párování (resp. ověřit jeho existenci). Aby to mělo smysl, musí být obě partity stejně velké. Naším úkolem je sestrojit z něj takový vstup pro Survivalisty, který bude korektní právě tehdy, když původní graf má perfektní párování.



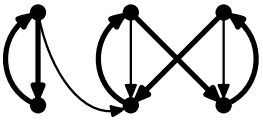
[67] `http://ksp.mff.cuni.cz/encyklopedie/hopcroft-karp.html`

[68] `http://ksp.mff.cuni.cz/viz/kucharky/tezke-problemy`

To je ale jednoduché: každou hranu zorientujeme z horní partity do dolní, a navíc přidáme „zpětné hrany“, které povedou vždy z i-tého vrcholu dolní partity do i-tého vrcholu horní. Například z grafu (zvýrazněno perfektní párování)



vznikne vstup (zvýrazněna korektní množina předání)



Nyní si snadno rozmyslíte obě implikace. Pokud existuje perfektní párování, snadno z něj utvoříme řešení Survivalistů: použijeme párovací hrany a všechny zpětné. Naopak každé korektní řešení Survivalistů musí nutně použít všechny zpětné hrany (z libovolného vrcholu dolní partity vede jen jedna hrana – zpětná – takže musí být použita), jejich odebráním dostaneme perfektní párování.

Tím jsme ukázali, že *libovolný* algoritmus řešící naši úlohu můžeme použít jako trochu zvláštní párovací algoritmus: připravíme mu vstup se zpětnými hranami (to zvládneme v lineárním čase, který můžeme zanedbat, neboť lineární čas potřebujeme i na pouhé načtení vstupu), zeptáme se na řešení a víme, zda původní graf obsahoval perfektní párování. Tedy žádné řešení Survivalistů nemůže být rychlejší než nejrychlejší algoritmus, který umí rozhodnout o existenci perfektního párování v bipartitním grafu, protože bychom pomocí něj uměli vytvořit rychlejší párovací algoritmus, což je ve sporu s tím, že ten původní byl nejrychlejší.

Filip Štědronský

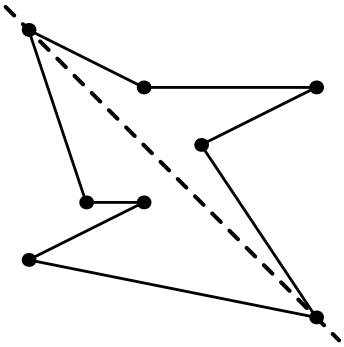
27-5-3 Čekání na poště

Nejprve si úlohu trochu zjednodušíme. Budeme uvažovat frontu lidí skutečně jako frontu a ne jako uzavřený okruh. Naším úkolem tedy bude pospojovat body v rovině lomenou čarou tak, aby se nikde neprotínala. Toho můžeme docílit jednoduše tím způsobem, že si body seřadíme podle y-ové souřadnice (v případě rovnosti pak podle x-ové).

Tento setříděný seznam bude přesně popisovat pořadí bodů, ve kterém je bude lomená čára procházet. Nikde se neprotne, neboť každá úsečka je ve všech bodech níže než ta předchozí (v případě dvou bodů se stejnou y-ovou souřadnicí je celá napravo).

Jak nám toto pozorování pomůže k vyřešení původní úlohy? Určitě nebude stačit body vypsat v setříděném pořadí, protože pak by nám mohla poslední úsečka spojující první a poslední bod protínat nějaké předchozí. Dobrým trikem ale je rozdělit si body na dvě části a tyto dvě části vyřešit předcházejícím algoritmem. V jedné části začneme od nejvýše položeného bodu a postupně budeme klesat až do nejnižšího bodu (tato část bude tvořit jakousi „levou polovinu“ mnohoúhelníka). Ve druhé části naopak začneme od nejníže položeného bodu a postupně budeme po zbylých bodech stoupat, až se opět dostaneme k výchozímu, nejvýše položenému bodu (tato lomená čára bude tvořit „pravou polovinu“ mnohoúhelníka). Budeme tím pádem chtít, aby každý bod byl v právě jedné části, s výjimkou nejvýše a nejníže položeného bodu, které můžeme pomyslně zařadit do obou částí.

Musíme ještě body do těchto dvou částí rozdělit. Potřebujeme, aby se úsečky z jedné části nekřížily s těmi z části druhé. Jedno řešení je přijít s nějakou přímkou a body nalevo od této přímky přiřadit do první části a body napravo do druhé. Toto nám zaručí, že žádná část lomené čáry nepřekročí tuto rozdělující přímku, a tím spíš se nebude křížit s žádnou částí druhé lomené čáry. Jelikož chceme nejvyšší a nejnižší bod v obou těchto částech, nabízí se vzít právě přímku určenou těmito dvěma body.



Pro určení, na které straně přímky bod leží, stačí vzít determinant matice, jejíž první řádek je směrový vektor naší rozdělující přímky a druhý řádek je vektor určený nejnižším bodem a zkoumaným bodem. Podle znaménka tohoto determinantu pak můžeme určit, na které straně se zkoumaný bod nachází. Pokud nám to nevěříte, tak nahlédněte do naší geometrické kuchařky,[69] kde naleznete podrobnější popis.

Celkovým výstupem algoritmu bude seznam bodů v setříděném pořadí nejprve z jedné části a pak z druhé. Nesmíme ale zapomenout na to, že náš původní algoritmus obě části seřadil odshora dolů, chceme tedy jednu z částí vypisovat v opačném pořadí.

KSP

Jak je to s časovou a paměťovou složitostí? Setřídění prvků zvládneme v čase $\mathcal{O}(N \log N)$. Roztříděním do dvou částí strávíme na každém bodě konstantní čas, tedy dohromady $\mathcal{O}(N)$, a samotné vypsání pak stihneme také v lineárním čase. Takže celková časová složitost je $\mathcal{O}(N \log N)$. Pamatovat si musíme pouze body na vstupu, takže si vystačíme s lineární pamětí.

Program (C): `http://ksp.mff.cuni.cz/viz/27-5-3.c`

řešení

Dominik Smrž

27-5-4 Školení zaměstnanců

V zadání jste dostali pěkně zakořeněný strom, to přímo vybízí k tomu ho nějak prohledat. Ukážeme si řešení využívající prohledání do hloubky.

Máme-li podstrom hloubky ostře menší než K, žádný vyškolený zaměstnanec v něm zatím být nemusí. Jakmile ale dostaneme podstrom s hloubkou právě K, už v něm nějakého zaměstnance vyškolit musíme – z vyšších pater stromu už bychom nedosáhli do listů tohoto podstromu. Vhodným kandidátem je kořen právě prozkoumávaného podstromu, žádný jiný vrchol nemusí dosáhnout do všech větví.

Kdybychom pouze takto odřezávali podstromy, nemusí nám vyjít správné řešení, protože ignorujeme dosah zaměstnance nahoru po stromě. Myšlenku si tedy zobecníme a zavedeme si u každého vrcholu *vyškolenost*.

Vyškolenost zaměstnance, kterého na školení pošleme, bude K. Směrem od něj se bude vyškolenost snižovat. Všimněte si nyní, že řešení splňující podmínky musí mít na konci v každém vrcholu vyškolenost alespoň nula.

Nastavíme vyškolenost listů na nulu a budeme konstruovat řešení rekurzivně pro vnitřní vrcholy. Na chvíli si dovolíme, aby vyškolenost klesla u některých vrcholů do záporných čísel, a teprve až bude příliš nízká, tak ji spravíme vyškolením zaměstnance v kořeni.

[69] `http://ksp.mff.cuni.cz/viz/kucharky/geometrie`

Jak tedy spočítáme vyškolenost vnitřního vrcholu? Podíváme se na minimum a maximum vyškolenostní synů. Pokud má některý ze synů vyškolenost tak vysokou, že pokryje nedostatky ostatních synů ($max + min > 0$), můžeme vyškolenost aktuálního vrcholu nastavit na $max - 1$ a tím je celý podstrom vyřešen.

Jinak musíme respektovat nejméně vyškoleného syna. Pokud je $min = -K$, pak nezbývá než poslat na školení aktuální vrchol, a tím všechny syny spravit (vyškolenost bude K). Jinak vrátíme $min - 1$ a odložíme vyřešení na později.

Z této úvahy se vymyká již jen kořen celého stromu, kde nelze řešení odkládat. Proto jej na konci přidáme, pokud musíme. Minimalita nalezeného řešení vychází z úvahy ve druhém odstavci.

Algoritmus poběží v lineárním čase k počtu vrcholů, stejně tolik spotřebuje paměti. Jen pozor, Python při přímočaré implementaci rekurzí příliš plýtvá místem na zásobníku pro volání funkce, proto v něm úloha byla řešitelná, pouze pokud jste použili explicitní zásobník jen na vrcholy a rekurzi jste se vyhnuli.

Program (C++): `http://ksp.mff.cuni.cz/viz/27-5-4.cpp`

Ondra Hlavatý

27-5-5 Kniha přání a stížností

V úloze je nutné ve vstupním textu hledat výskyty různých slov, navíc byla úloha v letáku označena jako kuchařková. Jak jste někteří sami zformulovali, to přímo vybízí k použití Aho-Corasickové.

Ale teď jak ji použít. Předně, můžeme si ji trochu zjednodušit: jelikož slova nejsou svými sufixy, nemusíme vůbec řešit zkratky.

Naivní řešení může pomocí Aho-Corasickové hledat výskyt libovolné jehly (tedy libovolného zakázaného slova) v celém vstupu. Kdykoliv nějakou najde, smaže ji a hledání se opět spustí od začátku. To je zaručeně správný postup, ovšem běží v $\mathcal{O}(S^2)$, kde S značí délku vstupu. Přitom řetězcové úlohy se, zejména v soutěžích, obvykle dají řešit lineárně.

Můžeme si rozmyslet, že stačí vracet se ve vstupním řetězci ne na začátek, ale pouze o délku nejdelší jehly. Delší jehlu jsme vytvořit nemohli, a kdyby se nějaká v řetězci už vyskytovala, našli bychom ji dřív, než bychom došli k aktuálnímu znaku. Tím jsme se sice z $\mathcal{O}(S^2)$ dostali na $\mathcal{O}(S \cdot j_{\max})$, nicméně to stále není lineární.

Hlavní problém naivního řešení je, že spoustu věcí zbytečně počítá opakovaně. Náš průchod automatem (resp. trií) bude v té části řetězce, která se nezměnila, stále stejný. Pokud jsme při zpracování řetězce došli na pozici i, a tím se dostali do vrcholu v, i když smažeme nějaké znaky $i+1, \ldots, i+j$ a vrátíme se v řetězci na začátek, stejně na pozici i zase skončíme ve vrcholu v.

Kdybychom tedy věděli, v jakém vrcholu jsme na pozici i byli, můžeme se po smazání jehly začínající na pozici $i+1$ prostě „přepnout“ do daného vrcholu

a nerušeně pokračovat ve zpracovávání řetězce. To zvládneme snadno, stačí nám pořídit si pole, do kterého si pro každou pozici uložíme odpovídající vrchol trie.

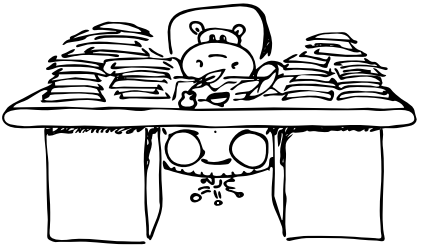
Jeden problém jsme tím ovšem vyrobili. Přesněji řečeno jsme si rozbili časovou složitost. Původní argument pracuje s tím, že při načtení znaku se sice můžeme vracet o mnoho vrstev trie nahoru, ale nemůžeme se celkem vracet vícekrát, než kolikrát jsme sestoupili níž. A protože při přečtení znaku sestoupíme maximálně o jednu vrstvu, bude i návratů nejvýš lineárně.

Jenže přepínáním stavů se za každý načtený znak můžeme přesunout o mnohem víc vrstev dolů, a tedy i těch návratů může být mnohem víc. Hodilo by se nám proto přímo vědět, ve kterém stavu se vracet přestaneme a budeme moci zase přejít o úroveň níž.

To si (alespoň pro rozumně velkou abecedu) můžeme předpočítat. Musíme to ale udělat šikovně. Půjdeme na to opět po vrstvách. Začneme s kořenem, pro ten je to jednoduché – pro každý znak abecedy buď vede hrana někam dolů, nebo zůstáváme v kořeni.

Pro každý další vrchol v a každý další znak abecedy z pak bude platit, že buď z v existuje hrana dolů označená z, nebo se vracíme tam, kam bychom při čtení z došli z toho vrcholu u, do kterého vede zpětná hrana z v. Protože postupujeme po vrstvách, to, kam bychom při čtení z došli z u, už určitě víme. Zpracování každé dvojice vrchol a znak tak zabere jen konstantní množství času, dohromady tedy $\mathcal{O}(J \cdot |\Sigma|)$. Časovou složitost konstrukce trie jsme si tedy nezhoršili.

Návraty v trii nahoru jsme vyměnili za konstantní přepnutí stavu. Sestupů dolů bude maximálně lineárně, protože při přečtení znaku se stále posuneme maximálně o jednu úroveň níž. Celková velikost výstupu (toho, co nám bude Aho-Corasicková postupně vracet) bude díky mazání již nalezených jehel maximálně S. Hledání samo o sobě tedy zabere $\mathcal{O}(S)$.



Zbývá rozmyslet si, kolik času nám zabere samotné smazání jehly ze vstupu. Tady totiž záleží, jak se rozhodneme se vstupem pracovat. Pokud si ho uložíme do pole, narazíme na to, že mazání v poli je obecně drahé – přinejhorším lineární. Asi nejpříjemnější řešení je pořídit si ještě druhé pole, v kterém budeme postupně vytvářet výstup. Při zpracování znaku ho zkopírujeme do výstupního pole, při mazání jehly ji mažeme právě z výstupního pole.

Tím vždy mažeme z konce, tedy smazání znaku je konstantní, smazání jehly lineární v její délce. A protože nemůžeme smazat víc znaků, než jsme jich na vstupu dostali, zaberou všechna mazání dohromady maximálně $\mathcal{O}(S)$. Celý algoritmus tak poběží v čase $\mathcal{O}(S + J \cdot |\Sigma|)$ a spotřebuje stejné množství paměti.

Pro úplnost dodejme, že kdyby byla abeceda příliš velká na předpočítání, můžeme návraty počítat „za běhu“. Stavy, do kterých se přepínat, si můžeme ukládat do binárního stromu stejně jako stavy, do kterých vedou běžné hrany dolů. Při načtení znaku se podíváme, zda už máme stav spočítaný, a pokud ne, spočítáme ho (a při návratu z výpočtu uložíme přepínaný stav i všem vrcholům, přes které jsme prošli). Přepnutí stavu bych pak bylo $\mathcal{O}(\log |\Sigma|)$, celková složitost $\mathcal{O}((S + J) \log |\Sigma|)$.

Program (C): `http://ksp.mff.cuni.cz/viz/27-5-5.c`

Karolína „Karryanna“ Burešová

27-5-6 Autobazar

Nejprve uveďme na pravou míru pár nešťastných formulací ze zadání, které tiskařský šotek propašoval několika koly korektur a které se naštěstí ujasnily v diskusi ve fóru. Číslo vyjadřující počet aut se pochopitelně do paměti vejde (jinak by úloha vůbec nebyla řešitelná). To, co se nevejde, je libovolná datová struktura obsahující všechna auta nebo všechny jejich barvy. Celkově smíme používat jen konstantně velkou paměť, ovšem neměříme ji v bitech. Jako jednotku prostorové složitosti používáme zde, jakož i jinde v KSPčku, čísla velká srovnatelně s těmi ze vstupu (případně polynomiálně větší). Za zmatky se každopádně omlouváme.

Binární vyhledávání

Úkolem je najít v posloupnosti n čísel takové, které se vyskytuje více než $(n/2)$-krát. Takovému číslu budeme říkat *vítěz*.

Náš první algoritmus na nalezení vítěze bude založený na binárním vyhledávání. Začneme tím, že spočítáme minimum a maximum ze zadaných čísel, označíme si je třeba m a M. Pak interval mezi nimi rozdělíme na poloviny a pro každou z polovin spočítáme, kolik čísel se v ní vyskytuje. Pokud existuje vítěz, pak ta z polovin, v níž leží, musí obsahovat aspoň $n/2$ čísel. Tuto polovinu opět rozdělíme na poloviny a tak dále, až interval omezíme na jedinou hodnotu.

Celkem provedeme $\mathcal{O}(\log(M - m))$ kroků, každý z nich jednou přečte celý vstup. Celý algoritmus tedy běží v čase $\mathcal{O}(n \cdot \log(M - m))$.

Hlasování o číslicích

Jiný způsob s podobnou časovou složitostí je založený na následující úvaze: Kdyby byla všechna čísla řekněme dvojciferná a vítězem bylo číslo 42, pak nadpoloviční většina čísel musí začínat čtyřkou (tou začínají všechny výskyty ví-

téze a možná ještě nějaká další čísla). Podobně musí nadpoloviční většina končit dvojkou.

Můžeme si tedy všechna čísla rozložit na číslice v desítkovém zápisu a uspořádat hlasování o nejpopulárnější číslici. To pro každou pozici trvá $\mathcal{O}(n)$ a pozic je celkem $\mathcal{O}(\log M)$.

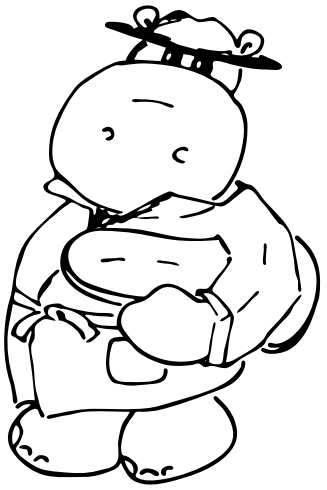
Pakliže vítěz existuje, musí být tvořen odhlasovanými číslicemi. Pozor ale na to, že i ve vstupu bez vítěze může na každé pozici mít nějaká číslice nadpoloviční většinu – triviální příklad je třeba vstup 12, 13, 23. Odhlasované číslo je tedy potřeba dodatečně ověřit.

Tento algoritmus má složitost $\mathcal{O}(n \cdot \log M)$. Dodejme ještě, že implementaci by zjednodušilo, kdybychom použili místo desítkové soustavy dvojkovou.

Optimální řešení

Všechny tyto úvahy o číslech nás ale od optimálního řešení spíš odvádějí. Zapomeňme na to, že barvy aut mají nějakou strukturu, a považujme je za něco, co lze jenom porovnávat na rovnost. Tím jsme algoritmu dovolili v zásadě jen pamatovat si nějakých konstantně mnoho barev (víc se nám do paměti nevejde) a počítat, kolikrát se vyskytly. To nás dovede k překvapivě jednoduchému řešení.

V každém okamžiku si budeme pamatovat jednu barvu, té budeme říkat *kandidát,* a udržovat si počítadlo výskytů této barvy.



Na počátku výpočtu se kandidátem stane první prvek vstupu a počítadlo nastavíme na jedničku. Kdykoliv pak narazíme na další výskyt téže hodnoty, počítadlo o jedna zvýšíme. Pokud na výskyt jakékoliv jiné barvy, počítadlo o jedničku snížíme. A pokud počítadlo klesne na nulu, zapomeneme na všechno, co jsme viděli, a prohlásíme za kandidáta bezprostředně následující prvek.

Tvrdíme, že existuje-li vítěz, je roven tomu kandidátovi, který nám zbyl na konci výpočtu.

Jakmile dokážeme, že je to pravda, bude algoritmus hotový: prvním průchodem budeme počítat kandidáty, druhým průchodem ověříme, že finální kandidát je skutečně vítězem. To zabere čas $\mathcal{O}(n)$ a konstantní prostor (stačí nám čtyři proměnné: aktuální prvek, kandidát, počítadlo a celkový počet prvků).

Pro potřeby důkazu rozdělíme vstup na *epochy.* Epocha skončí buďto vynulováním počítadla, nebo tím, že dojde vstup. Například takto:

3 3 1 3 2 4 | 4 3 | 3 3 1 |

První prvek epochy se stane kandidátem a zůstává jím až do konce epochy. Pro každou epochu kromě poslední platí, že počet zvýšení počítadla se musel rovnat počtu snížení, takže kandidát je roven právě polovině prvků v epoše. Jen poslední epocha může končit kladnou hodnotou počítadla, takže kandidát se v ní může vyskytovat vícekrát než ostatní prvky.

Teď už si stačí všimnout, že pokud je nějaký prvek vítězem, musí se vyskytovat v nadpolovičním počtu případů v alespoň jedné epoše. Už ale víme, že prvek s touto vlastností může ležet pouze v poslední epoše a musí to být její kandidát. Hotovo.

Martin „Medvěd“ Mareš

27-5-7 Shellová automatizace

Podúloh v tomto díle seriálu bylo mnoho, pojďme se do nich pustit a vyřešit je pěkně jednu po druhé.

Úkol 1 – Počítání řádek v souborech

Tento úkol měl vlastně dva jednoduché kroky: prvním z nich bylo získat všechny soubory s příponou `.txt` a pak je vhodným způsobem poslat do příkazu `wc` a nechat spočítat řádky v nich.

V podstatě tedy šlo jen o zavolání příkazů `find` a přes `xargs` příkazu `wc`. Nakonec se ještě pomocí `tail` a `awk` dalo z výstupu `wc` vyseknout jen celkový součet na posledním řádku:

```
find . -name "*.txt" -print0 | xargs -0 wc -l
  | tail -n1 | awk '{print $1}'
```

Úkol 2 – Hledání prázdných podadresářů

Při zadávání tohoto úkolu jsme si neuvědomili, že samotný `find` má přepínač `-empty` a stačilo tak pouze zavolat následující příkaz (`-mindepth` je zde z důvodu, aby nebyl vypsán i aktuální adresář, kdyby byl prázdný):

```
find . -mindepth 1 -type d -empty
```

Naše původní (a výrazně pomalejší) řešení spočívalo v tom, že si necháme vypsat příkazem `find` všechny složky a jednu po druhé budeme testovat jejich prázdnost (třeba podle toho, jestli `ls -A` něco vypíše):

```
find -mindepth 1 -type d | while read -r dir; do
        [ -z "$(ls -A "$dir")" ] && echo "$dir";
done
```

Úkol 3 – Změna přípony

Úkolem bylo změnit všechny přípony `.tvuj` na `.muj`, na první pohled jednoduchá práce. Nalezení všech souborů, jichž se to týká, je už jen jednoduché použití známého příkazu `find`, změna přípony je ale záludnější.

Kdyby šlo pouze o to příponu přidat, bylo by to jednoduché použití `mv` ve stylu `mv "$0" "$0.muj"`. Ale takto je to o trochu složitější.

Zde se hodí zmínit *expanzi a substituci v proměnných*, o které v seriálu zmínka nepadla. Pokud napíšeme `${promenna%.txt}`, tak dostane obsah proměnné oseknutý o koncové `.txt`. Bez tohoto by nový název souboru šel zkonstruovat třeba voláním subshellu (pomocí `‘...‘` nebo `$(...)`) a v něm příkazu `sed`. Ve vzorovém řešení níže ale použijeme kratší zápis.

KSP

Pokud bychom dostali název souboru v proměnné `$0`, vypadal by pak příkaz jako níže (uvozovky jsou tam třeba kvůli mezerám v názvech souborů):

```
mv "$0" "${0%.tvuj}.muj"
```

Když budeme příkaz dávat do `-exec` části příkazu `find`, musíme ještě navíc udělat jeden trik. Samotný `find` nám žádnou proměnnou, na které by se dala provádět tato expanze, neposkytne, ale můžeme si zavolat shell, kterému hodnotu od `find`u předáme jako první parametr a pak ji budeme mít uvnitř dostupnou právě jako proměnnou `$0`:

řešení

```
sh -c 'mv "$0" "${0%.tvuj}.muj"' "{}"
```

Poslední záludnou věcí, na kterou se hodí pamatovat, je to, že na `.tvuj` může končit i jméno adresáře a ten bychom měli také přejmenovat. Ale když to uděláme dříve, než přejmenujeme soubory v tomto adresáři, máme problém – k těmto souborům se už s původní cestou nedostaneme a museli bychom složitě modifikovat to, co nám vrátil `find`. Co ale kdybychom nejdříve zpracovali celý obsah adresáře a samotný adresář přejmenovali až na konci? A přesně k tomu slouží přepínač `-depth`.

Celý zkonstruovaný příkaz pak vypadá takto:

```
find . -depth -name "*.tvuj" -exec \
  sh -c 'mv "$0" "${0%.tvuj}.muj"' "{}" \;
```

Jiná verze používající nezmíněný, ale šikovný příkaz `rename`:

```
find . -depth -name "*.tvuj" -exec \
  rename "s/tvuj$/muj/" "{}" \;
```

Úkol 4 – Paralelizace

Při vymýšlení řešení jste mohli narazit na několik záludností. Aby bylo možné v bashovském skriptu odchytávat signál SIGCHLD, je potřeba zapnout job control pomocí `set -m`. Stále však můžeme narazit na to, že se stejné signály neřadí do fronty. Pokud skončí dva paralelní úkoly současně, může se stát, že zaznamenáme jen jeden signál.

Mohli bychom to obejít tím, že každý z řádků obalíme naší vlastní funkcí. Ta nás bude o dokončení informovat nějakým jiným způsobem – například zápisem řádku do jednoho společného souboru.

Také si musíme dát pozor na další zákeřnost. Není úplně dobré číst jeden vstup ve více paralelně běžících vláknech. Nikdo nám totiž nezaručí, že se data rozdělí přesně po celých řádcích.

Zkusíme se tedy těmto úskalím vyhnout:

```
max="$1"
while read cmd; do
  cnt="`jobs | wc -l`"
  if [ "$cnt" -ge "$max" ]; then
    wait -n
  fi
  eval "$cmd" &
done
wait
```

Vždy ve smyčce překontrolujeme počet běžících jobů – řádků vstupu. Pokud jich je méně než maximum, spustíme další. Jinak pomocí `wait -n` počkáme na konec libovolného z nich. V některých shellech tento parametr chybí. Pro ně můžeme kontrolu nahradit aktivní smyčkou.

Pozor na to, že `` `jobs | wc -l` `` může běžet v subshellu, ze kterého již nebudou vidět naše spouštěné příkazy. V některých shellech je potřeba místo tohoto řádku používat přesměrování do souboru. Výsledek by pak vypadal následovně:

```
max="$1"
tmp="`mktemp`"
trap "rm -f "$tmp";echo;exit 0" INT QUIT
while read cmd; do
  jobs > "$tmp"
  cnt="`wc -l < "$tmp"`"
  while [ "$cnt" -ge "$max" ]; do
    sleep 1
    jobs > "$tmp"
    cnt="`wc -l < "$tmp"`"
  done
  eval "$cmd" &
done
wait
rm -f "$tmp"
```

Vaší pozornosti doporučujeme ještě použitý příkaz `mktemp` pro vytvoření dočasných souborů. Ten vytvoří soubor (případně adresář `-d`) s unikátním názvem. Už nikdy si tak pomocnými soubory nepřepíšete důležitá data, případně nezanesete svůj pracovní adresář či home.

Pro zajímavost si ještě ukážeme, že paralelizace můžeme dosáhnout i využitím `make`. Zavoláme-li jej s parametrem `-j [N]`, bude se provádět vždy nejvýše

N cílů současně. Make přitom dodrží všechny závislosti. Čtvrtý úkol tedy šlo vyřešit i následovně:

```
mf=‘mktemp‘
cat -n | sed -r > "$mf" \
  ’s/[[:space:]]*([0-9]+)\t(.*)/pr\1:\n\t\2\n/’
make -sBf "$mf" -j $1 ‘grep ^pr "$mf" | tr -d :‘
rm -f "$mf"
```

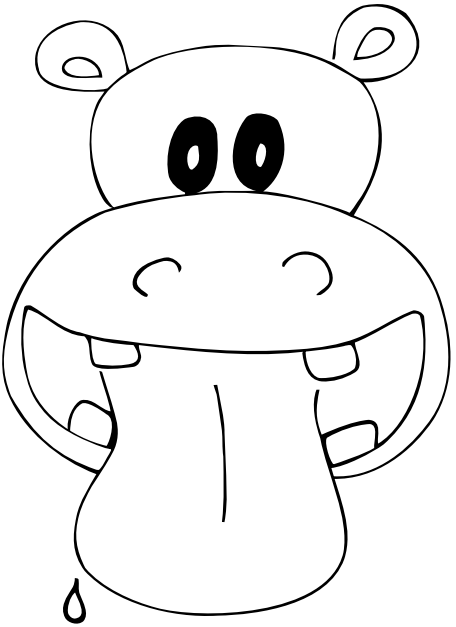
KSP

Vstup převedeme na Makefile, kde každému řádku odpovídá jedno pravidlo, a následně necháme make paralelně provést všechna pravidla. Důležitý je parametr `-B`, díky kterému se znovu provede vše nezávisle na existenci souborů se stejným názvem jako pravidlo. Prakticky tím děláme ze všech cílů .PHONY. Parametr `-s` zařídí, aby make nevypisoval právě prováděný příkaz.

Trik s make je pro obecné skripty trochu nepraktický, ale pokud chcete například hromadně vytvářet náhledy fotek, vyjde výroba Makefile a shellového skriptu přibližně nastejno.

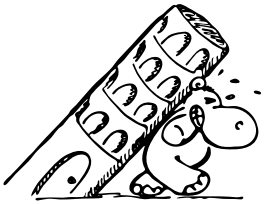
řešení

Pro úplnost dodejme, že v GNU rozšíření `xargs` existuje parametr `-P`, kterým můžeme paralelizaci snadno získat. Řešení pak zkrátíme na volání příkazu `xargs -P $1 -n 1 -d "\n" bash -c`. Jenom jsme museli omezit počet parametrů pro jedno spuštění příkazu pomocí `-n 1` a vybrat nový řádek jako jediný oddělovač.



Úkol 5 – Výpis procesů

Zde nebylo skoro co řešit. Jednoduše stačilo v každém adresáři složeném pouze z čísel (resp. začínajícím na číslo) přečíst pár souborů a hezky je vypsat – s tím nám pomůže starý známý `column` ze čtvrtého dílu seriálu.



Seznam všech argumentů dostaneme z /proc/PID/cmdline. Jenom jsou oddělené pomocí nulového bytu, který v terminálu není vidět. Můžeme ho snadno zobrazit pomocí `tr "\0" " "`, nebo `xargs -0 echo`. Protože `echo` je defaultní příkaz pro `xargs`, nemusíme jej ani psát.


```
{ echo "PID#User#     RSS#CWD#Command"
for i in /proc/[0-9]*; do
  cd "$i" || continue
  pid="${i#/proc/}"
  uid="`grep ^Uid: status | cut -f 3`"
  unm="`getent passwd "$uid" | cut -f1 -d:`"
  rss="`grep ^VmRSS: status | cut -f 2`"
  cwd="`readlink cwd`"
  cmd="`xargs -0 < cmdline`"
  [ -z "$unm" ] && unm="$uid"
  [ -z "$rss" ] && rss="       ?"
  [ -z "$cwd" ] && cwd="?"
  [ -z "$cmd" ] && cmd="?"
  [ -e ./ ] || continue
  echo "$pid#$unm#$rss#$cwd#$cmd"
done } 2>/dev/null | sort -n | column -s "#" -t
```

Protože náš skript chvíli poběží, mohou mezitím některé procesy skončit. Kdybychom měli opravdu velkou smůlu, vznikne v průběhu jiný proces se stejným PID. Pak by se mohlo stát, že na jednom řádku budeme mít kombinaci informací o dvou procesech.

Popsanému problému jsme se však vyhli tím, že měníme náš pracovní adresář. Pokud vznikne nový proces se stejným PID, vznikne také nový adresář se stejným jménem. Ten starý, ve kterém jsme, již existovat nebude. Pokud tedy proces skončí dřív, než o něm zjistíme veškeré informace, raději je nevypíšeme vůbec. O to se postará `[ -e ./ ] || continue`.

Úkol 6 – Jednoduchý Makefile

Toto bylo v podstatě cvičení na to, jestli jste pochopili smysl Makefilů. Pro většinu z vás to nebylo nic těžkého, Makefile odpovídající zadání by mohl vypadat třeba takto:

```
A: A.data
B: B.data
C: C.data
AB: A.data B.data
BC: B.data C.data

%:
	generuj $^ >$@

FIN1: A AB B
FIN2: BC C
FINAL: A AB B BC C

FIN%:
	finalizuj $^ >$@
```

Abychom nemuseli příkaz psát ke každému cíli, tak jsme pro každý cíl specifikovali jen jeho závislosti a příkazy jsme napsali vždy pro celou skupinu cílů najednou. Jak jste si mohli ozkoušet, `make` volí vždy ten nejvíce specifický cíl, takže bylo možné příkaz `generuj` umístit do obecného cíle `%` a příkaz `finalizuj` do (o trochu méně) obecného cíle `FIN%`.

Závislosti u finálních souborů byly natolik specifické, že je bylo nutné vypsat ručně, ale nešla by nějak zautomatizovat tvorba základních souborů? Šla a Richard Hladík přišel s velmi elegantním postupem. Dá se využít prostá shellová expanze wildcardů, kdy se `[AB].data` expanduje na `A.data B.data` (pokud tyto existují, což jsme ale měli slíbeno).

Tímto velmi elegantním způsobem šlo místo pěti pravidel pro výrobu „písmenkových“ souborů a jednoho společného pravidla použít jen upravené společné pravidlo a přídavné pravidlo, že `%.data` na ničem nezávisí. Zkrácená verze tedy vypadá takto:

```
%.data:
FIN1: A AB B
FIN2: BC C
FINAL: A AB B BC C

%: [%].data
	generuj $^ >$@

FIN%:
	finalizuj $^ >$@
```

Teď ke generování:

- `make FINAL`: Dojde k vyrobení všech „písmenkových“ souborů a souboru `FINAL`.
- `make FIN1`: Protože už jsou soubory `A`, `AB` i `B` vygenerované, vyrobí se jen soubor `FIN1`.
- `touch A.data`
- `make FIN2`: Na souborech, na kterých závisí tento cíl (`BC` a `C` a tranzitivně `B.data` a `C.data`) se nic nezměnilo, a tak se vyrobí jen soubor `FIN2`.
- `touch C.data`
- `make FIN1`: Protože se od doby vygenerování `FIN1` změnil soubor `A.data`, musí se znovu vygenerovat soubory `A`, `AB` a teprve po nich `FIN1`.

Úkol 7 – Cyklický Makefile

Poslední úkol se možná ukázal trošku složitějším na správné pochopení zadání, ale když se na něj člověk chvíli díval (a třeba si závislosti nakreslil na papír), tak byl řešitelný celkem jednoduše.

Pokud si napíšeme Makefile jako tento níže (a když víme, že při zavolání příkazu `pdftex` nám vznikne i nový `obsah.toc`), tak už je problém vidět. Každé zavolání `make kniha.pdf` nám způsobí přegenerování všeho, i když se zdroják vůbec neměnil.

```
kniha.tex: zdrojak.tex obsah.toc
        cat $^ >$@

kniha.pdf: kniha.tex
        pdftex $<
```

Řešením je používat pomocný soubor s obsahem, a jen pokud se ten změní, přenést změny i do hlavního souboru s obsahem (a tím změnit jeho čas modifikace). Třeba takto:

```
obsah2.toc: obsah.toc
        diff $< $@ || cp $< $@

kniha.tex: zdrojak.tex obsah2.toc
        cat $^ >$@

kniha.pdf: kniha.tex
        pdftex $<
```

Pravidlo `obsah2.toc` přepíše tento soubor, pouze pokud se liší od souboru `obsah.toc`. Při opakovaném spuštění `make kniha.pdf` se tedy jen porovnají změny (pokud nejsou, opakovaný překlad TEXu se neprovede).

Jirka Setnička & Jenda Hadrava

Pořadí řešitelů KSP

Pořadí	*Jméno*	*Škola*	*Ročník*	*Úloh*	*Bodů*
0.				35	300.0
1.	Jan Špaček	G Wicht	4	32	284.4
2.	Richard Hladík	GOAMarLaz	2	32	254.0
3.	Stanislav Lukeš	GPísnickáPH	2	28	240.5
4.	Václav Volhejn	GKepleraPH	2	25	207.5
5.	Štěpán Hudeček	G Litovel	3	23	207.1
6.	Martin Scheubrein	G MNám Třb	3	24	206.0
7.	Marek Černý	G Chrudim	4	23	197.1
8.	Michal Převrátil	GKlatovy	2	19	162.4
9.	Michal Töpfer	G DrJPekMB	2	27	161.7
10.	Jakub Tětek	CírkG Plzeň	1	21	157.9
11.	Václav Šraier	GČeskoliPH	2	19	151.6
12.	Přemysl Šťastný	GŽamberk	2	21	145.2
13.	Jan Tománek	GPelhřimov	4	20	135.9
14.	Lukáš Ulrich	SSŠVTPraha	4	18	114.5
15.	Jan Kočur	G Wicht	4	16	113.4
16.	Adrián Goga	SPŠNitra	4	15	110.8
17.	Pavel Turinský	G Brandýs	2	11	110.2
18.	Jan Knížek	G Strakon	4	17	107.7
19.	Jakub Zárybnický	GTomkovaOL	4	13	102.7
20.	Anna Gajdová	GFPValMez	4	13	97.3
21.	Róbert Selvek	G KošiceS	3	13	96.4
22.	Václav Rozhoň	GJirsíkaČB	4	10	87.6
23.	Jiří Vozár	G UherBrod	3	9	75.9
24.	Jan Gocník	GJŠkodyPŘ	3	9	74.1
25.	Jiří Sejkora	GVoděraPH	3	8	68.7
26.	Matěj Konečný	GJírovcČB	4	6	60.1
27.	Jan Bouček	GKepleraPH	2	6	54.2
28.	Jan Pokorný	G Bučovice	3	7	50.0
29.	David Cholewa	GMatOS	4	8	46.2
30.	Václav Končický	GSOŠ FrMís	4	6	43.1
31.	Martin Zoula	GNadKavaPH	3	7	38.6
32.	Barbora Sedláková	GKonštanPV	4	8	35.9
33.	Jakub Matěna	GČeskoliPH	3	5	34.4
34.	Jan Soukup	GKlatovy	4	4	33.0
35.	Eva Matoušková	G Sokolov	4	4	25.8
36.	Filip Bialas	GOpatovPHA	2	2	20.0
37.	Vít Macura	GOAMarLaz	2	3	18.9
38.	Jakub Lukeš	GNAlejíPH	2	1	14.0

Pořadí řešitelů KSP

39.	Dalimil Hájek	GKepleraPH	4	1	13.4
40.	Martin Kubeša	GJŠkodyPŘ	3	1	12.8
41.	David Juřica	GNadŠtolPH	2	2	10.9
42.	Jan Kaifer	GČesBrod	−1	2	10.6
43.	Zuzana Svobodová	G FrýdlNOs	3	1	10.0
44.	Jan Burda	G Holice	1	1	9.0
45.	Václav Steinhauser	GDačice	1	1	7.9
46.	Roman Ondráček	GBoskovice	1	2	6.6
47.	Josef Vávra	SJec	4	2	5.7
48.	Jan Mráz	G Holice	1	5	4.4
49.	František Dostál	VSPŠEOc	4	1	4.0
50.	Roman Solař	GJarošeBO	3	1	2.4
51.	Michael Novák	SSŠVTPraha	4	1	2.0

Jiří Setnička a kolektiv

Korespondenční seminář z programování XXVII. ročník

Autoři a opravující úloh:
Jana Bátoryová, Karolína Burešová, Marek Dobranský, Lukáš Folwarczný
Jan Hadrava, Ondřej Hlavatý, Štěpán Hojdar, Ondřej Hübsch
Dominik Macháček, Tomáš Maleček, Martin Mareš, Jakub Maroušek
Michal Pokorný, Pavol Rohár, Vojtěch Sejkora, Jiří Setnička
Dominik Smrž, Martin Šerý, Jan Škoda, Martin Španěl,
Filip Štědronský, Karel Tesař, Michal Vaner, Kateřina Zákravská

Autoři příběhů v zadání:
Karolína Burešová, Dominik Macháček, Jakub Maroušek, Jiří Setnička

Autoři seriálu:
Jan Hadrava, Ondřej Hlavatý, Tomáš Maleček, Jiří Setnička, Filip Štědronský

Vydal MATFYZPRESS
vydavatelství Matematicko-fyzikální fakulty Univerzity Karlovy v Praze
Sokolovská 83, 186 75 Praha 8
jako svou 500. publikaci.

TEX-ová makra pro sazbu ročenky vytvořili Martin Mareš, Jan Matějka, Radim Cajzl a Jiří Setnička.

S jejich pomocí ročenku vysázeli Karolína Burešová a Pavol Rohár.

Obrázek na obálce nakreslila Petra Pelikánová.

Sazba byla provedena písmem Computer Modern v programu TEX.

Vytisklo Reprostředisko UK MFF.

Vydání první, 314 stran
Náklad 200 výtisků
Praha 2015

Vydáno pro vnitřní potřebu fakulty.
Publikace není určena k prodeji.

ISBN 978-80-7378-306-8

ISBN 978-80-7378-306-8