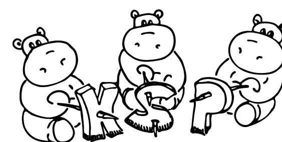
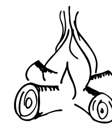
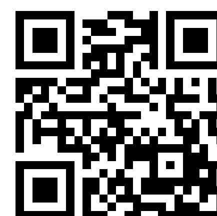
Korespondenční Seminář z Programování

27. ročník **KSP** **Květen 2015**

Milí řešitelé, řešitelky a řešitelčata!







Je to tady! Poslední série hlavní kategorie KSP byla doručena expresní hroší poštou až k vám. Pojďte zapřemýšlet třeba nad tím, jak efektivně šířit poplašné zprávy nebo jak se vysílají kouřové signály na UNIXu. A abyste se v textu neztratili, máme pro vás kuchařku o vyhledávání v textu. To celé doprovázené příhodami například o roku 2000 či o záporném dluhu.

Připomínáme, že každému řešiteli, který v tomto ročníku z každé série dostane alespoň 5 bodů, darujeme propisku, blok a tužku. To vše s logem KSP.

Za úspěšné řešení KSP je možno být přijat na MFF UK bez přijímacích zkoušek. Úspěšným řešitelem se stává ten, kdo získá za celý ročník alespoň 50 % bodů (tedy alespoň 150 z maxima 300 bodů).

Termín série: **Pondělí 1. června 2015 v 8:00 SELČ**
Odevzdávání: Přes web na adrese *https://ksp.mff.cuni.cz/submit/*
Informace: Další podrobnosti o fungování KSP naleznete na *http://ksp.mff.cuni.cz/*. Pokud budete mít jakoukoliv otázku, neváhejte se nás zeptat na *ksp@mff.cuni.cz* nebo na našem fóru. Přejeme hodně štěstí! ;-)
Odměna série: **Čokoládu** pošleme každému, kdo **z úloh této série získá alespoň 42 bodů**.

Pátá série dvacátého sedmého ročníku KSP

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.

* * *

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.

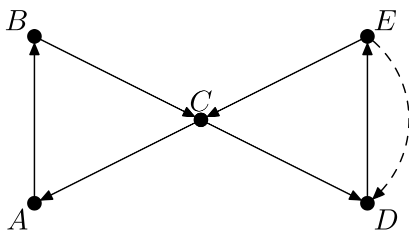
27-5-2 Survivalisté — 12 bodů

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.

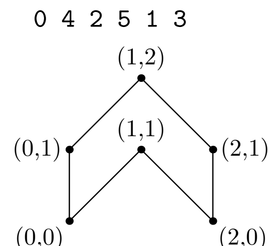
Pozor, výstup nemusí být jednoznačně určený (vypište libovolný korektní).

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.

Tato úloha je praktická a řeší se ve vyhodnocovacím systému CodEx.[1] Přesný formát vstupu a výstupu, povolené jazyky a další technické informace jsou uvedeny v CodExu přímo u úlohy.

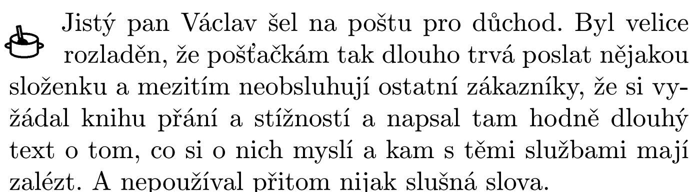
Po důkladném prostudování poštovních předpisů a po čtyřiceti minutách přesvědčování, mezitím, co se lidé z fronty

[1] http://ksp.mff.cuni.cz/viz/codex

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`	`c`
`Zakázaná slova: aaaaaaa, ab`	

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

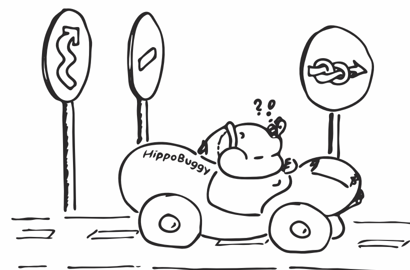
Autobazar, to je jedna dlouhá, dlouhá řada aut. Modré, červené, červené, černé, zelené, fialové, červené... Karel si vzpomněl na hru, kterou někdy hrají děti v autě. Každý počítá jednu svou barvu aut, která potkávají, a kdo jich má nejvíc, vyhraje.

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.



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

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.[2]

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.
- 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

[2] Dokonce může být vnější filtrování pomocí `grep`u výrazně pomalejší, protože se mezi těmito dvěma příkazy musí přenést skrz *rouru* velké množství dat.

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 `find`u 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

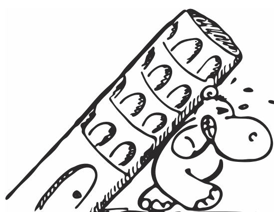
Jak padlo výše, tak `find` umí spouštět pro každý nalezený soubor nějaký příkaz, ale dělá to bohužel pro každý soubor samostatně. Pokud bychom například chtěli všechny takto nalezené soubory smazat, někam zkopírovat, nebo přidat do společného archivu, bude to zbytečně pomalé nebo komplikované.

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.)

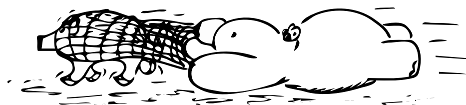
Signály mají za sebou bouřlivou historii a jejich význam se průběžně trochu měnil. Například někdy narazíme na to, že signály nezačínají na SIG. Máme pak INT, TERM, TSTP,... Pokud by vás zajímaly detaily, podívejte se do manuálových stránek `man 7 signal`. V zásadě můžeme signály dělit podle toho, jestli 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`.

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` nebo třeba `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: 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 $\{\texttt{0}, \texttt{1}\}$ pro čísla v binárním zápisu, klasické $\{\texttt{A-Z}, \texttt{a-z}\}$ 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.

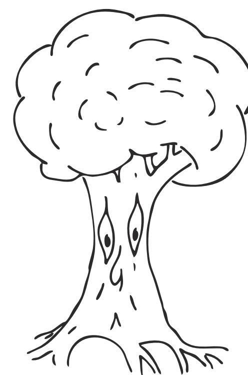
Terminologie dovoluje zepředu i zezadu odstranit prázdný řetězec – to znamená, že slovo je samo sobě prefixem i suffixem. Pokud chceme mluvit o prefixech, suffixech nebo obecně podřetězcích, kde jsme museli alespoň jeden znak odtrhnout, označíme takové podřetězce jako *vlastní*.

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.

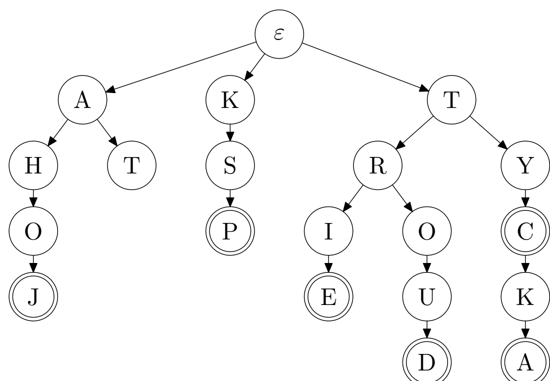
Pokud bychom nemuseli odebírat slova, můžeme použít hešování, které je rychlé a účinné. Více o něm najdete v hešovací kuchařce.[3] Má však tu nevýhodu, že při velkém zaplnění se začne chovat pomaleji a mírně nepředvídatelně.

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

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.“

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.

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 $\{\mathtt{0}, \mathtt{1}\}$. 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 $\mathcal{O}(L \cdot \log |\Sigma|)$.

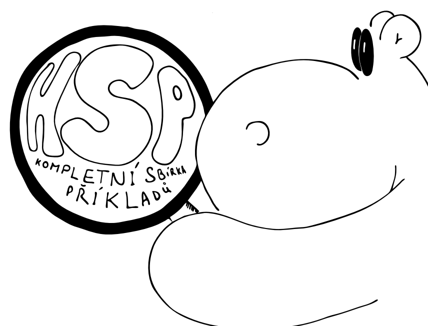
A jsme hotovi! S trií můžeme v lineárním čase odpovídat na dotazy „Vyskytuje se dané slovo ve slovníku?“, přidávat a odebírat další položky za běhu a nejen to – víc o tom ve cvičeních.

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.*[4]

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.

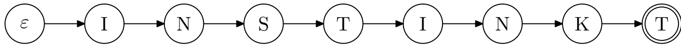


[4] `http://mj.ucw.cz/vyuka/ga/`

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.

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*.

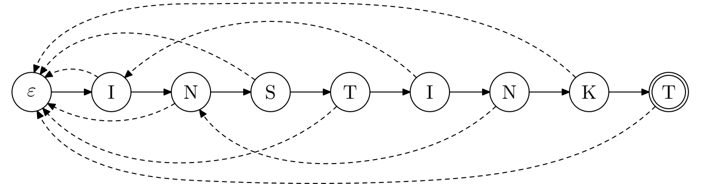
Zatím jsme ale přesně nepopsali, na které políčko přesně bude zpětná funkce ukazovat. Nechť chceme určit zpětné políčko pro druhé `N` ve slově `INSTINKT`. Pracujeme teď s prefixem `INSTIN`. Selsky řečeno, chceme najít „konec slova `INSTIN` takový, že je stejný, jako začátek slova `INSTIN`“.

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:

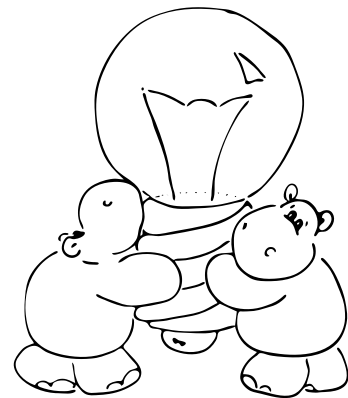


Nyní vyvstávají dvě otázky: Jakou to má celé časovou složitost? A jak spočítat zpětnou funkci?

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.

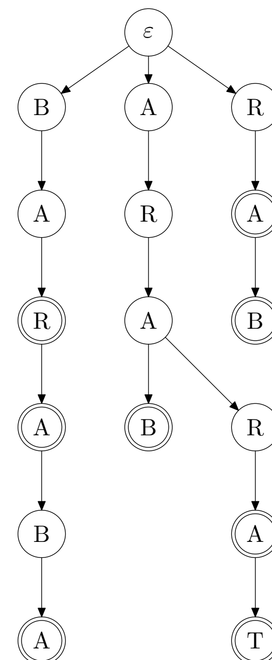
Vyhledávání jehelníčku

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.

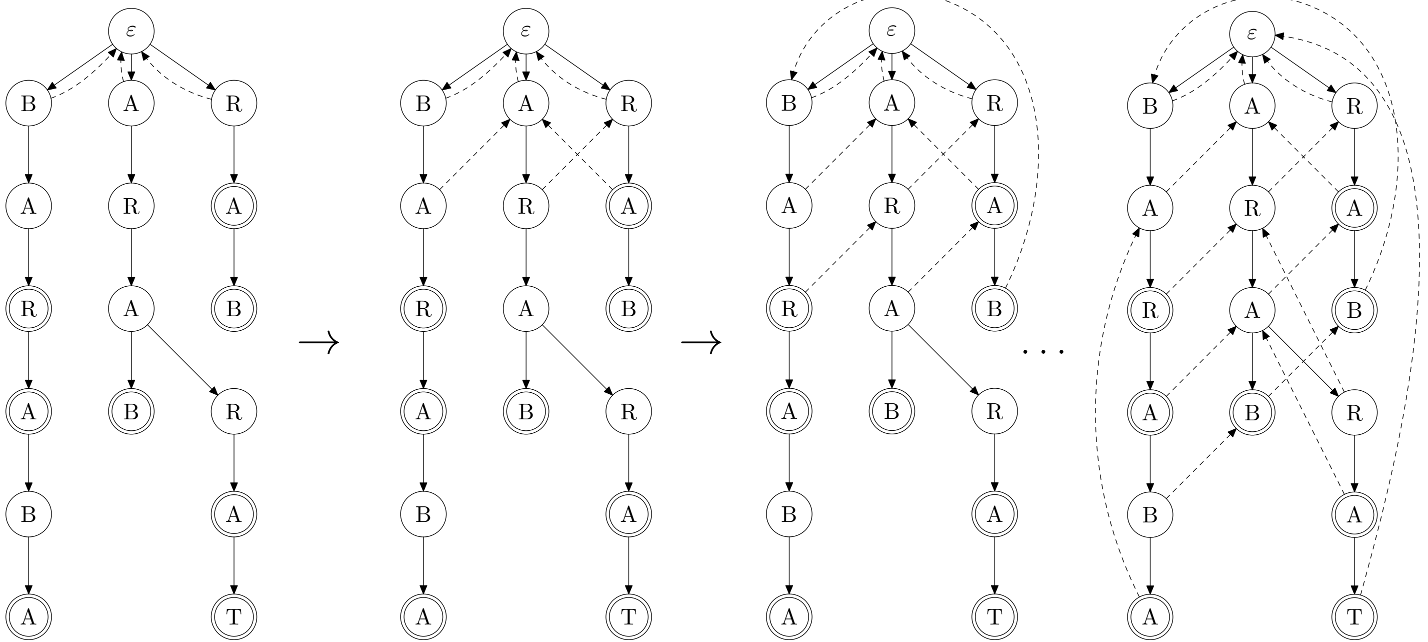
Zkusíme tedy nejprve sestrojit celou trii a pak postupně vyhledat nejdelší vlastní suffix pro každé ze slov. Ouha, to ale také nefunguje. Když začneme slovem `BARABA` a budeme tedy vyhledávat `ARABA`, nalezneme v trii úspěšně prefix `ARAB`, ale `ARABA` již v trii není. Měli bychom přejít ze slova `ARAB` 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`?

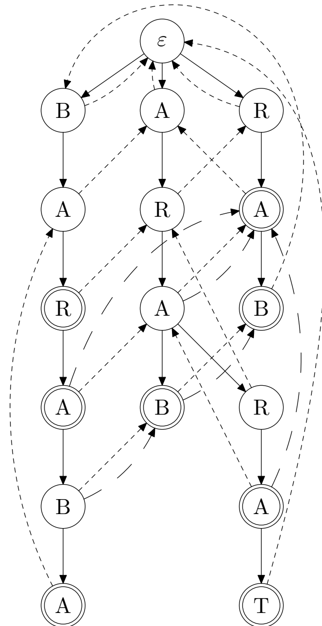


Při konstrukci další zpětné hrany tedy najdeme akorát, kde jsme minule skončili, a odtamtud pokračujeme dál. Jak to najdeme? Z otce našeho vrcholu tam přece vede zpětná hrana. Takže můžeme postup shrnout do bodů:

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 všechny 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.

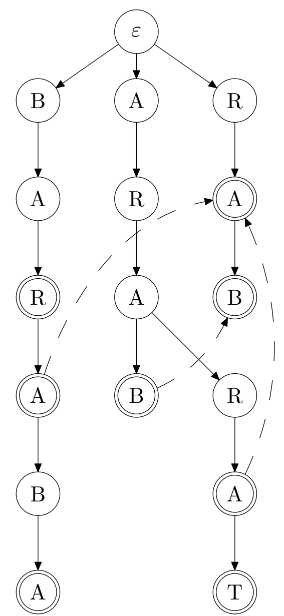
Představme si například slovník obsahující `A` a `AAAA...A` (délky $J - 1$). Budeme-li jím vyhledávat v textu `AAAA...A` délky $S > J$, projdeme prakticky pro každý znak až $J - 1$ zpětných hran, čímž složitost naroste až na nepoužitelných $\mathcal{O}(S \cdot J)$.

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 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.

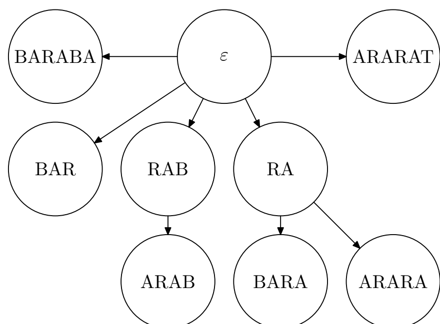
Použijeme trik – na každé pozici započítáme pouze nejdelší jehlu, která tam končí (u každé jehly si budeme udržovat čítač). Nebudeme tedy v každém kroku poskakovat po zkratkách až do aleluja, ale

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.



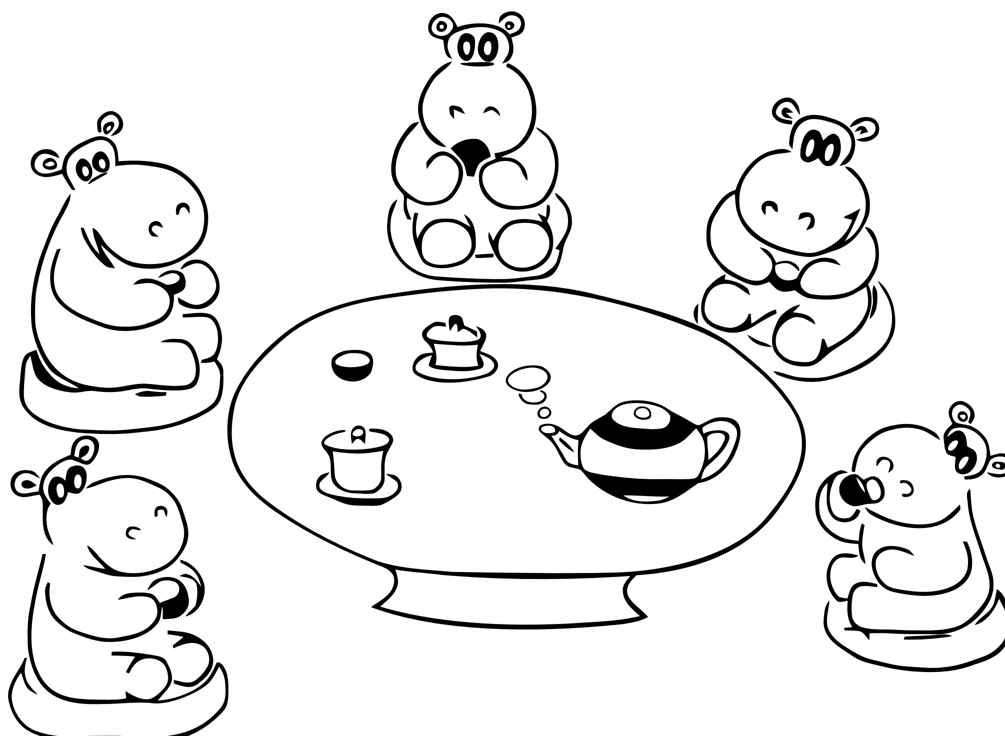
Poznámky

- 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 implementovat 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 čtvrté série dvacátého sedmého ročníku KSP

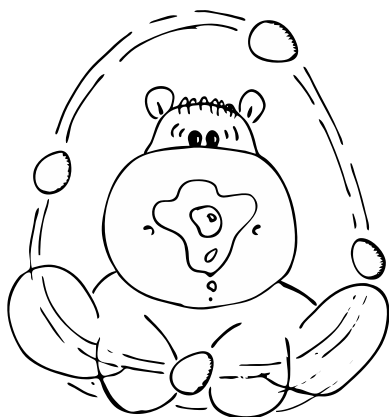
27-4-1 Zadávání úkolů

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.

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.

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.[5]

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.

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*.

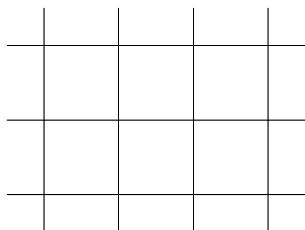
Nyní si tedy můžeme všechny přímky v čase $\mathcal{O}(N \log N)$ setřídit, dostat tak rovnoběžky k sobě a pak takto setříděné přímky lineárně projít. Pro každou nalezenou skupinu rovnoběžek budeme chtít nalézt rovnoběžky na ně kolmé. To můžeme pokaždé dělat binárním vyhledáváním v čase $\mathcal{O}(\log N)$ 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

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

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.

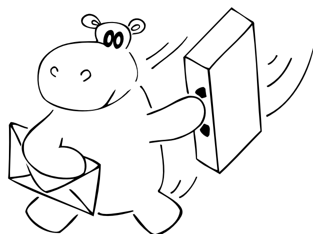
Opět si stačí pamatovat graf, takže paměťová složitost bude $\mathcal{O}(n + m)$. Co se týká časové, tak nejprve provedeme jednou jednodušší variantu na ohodnocení části grafu, poté v lineárním čase označíme nějaké 100kV uzly a poté znovu provedeme variantu jednoduššího algoritmu. Opět tedy dostaneme 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.

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.

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.

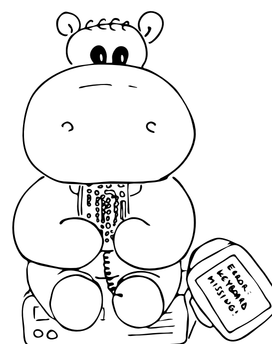
My ale toto řešení dokážeme vylepšit. Stačí nám do backtracku přidat ještě myšlenku dynamického programování: Při každém volání backtrackové funkce se stejnými parametry (aktuální vrchol, množina navštívených vrcholů), musíme nutně dostat i stejný výsledek. Tedy jakmile jednou výsledek pro konkrétní 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.

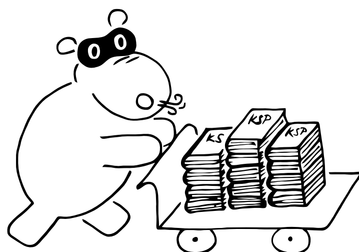
Program (C++):
`http://ksp.mff.cuni.cz/viz/27-4-5.cpp`

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.[6] 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).[7] Celkově máme N kotoučů, očíslujeme si je od nejmenšího po největší čísly 1 až N.

Funkce *přesun(k, výchozí, cílová, pomocná)*:

1. Pokud $k = 0 \to$ 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)*

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)*:

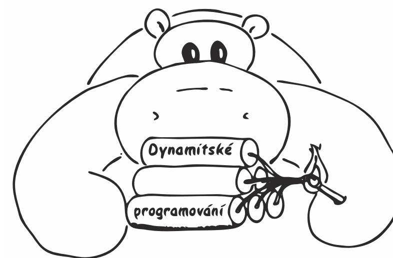
1. Pokud $k = 0 \to$ skonči
2. Pro každou fotografii z pole *fotografie*:
3. Pokud se k-tý kotouč nachází na tyči v, umísti fotografii do pole h_1
4. Jinak umísti fotografii do pole h_2
5. *seřazení1 = uspořádej(*$k-1$*,* h_1*, v, p, c)*
6. *seřazení2 = uspořádej(*$k-1$*,* h_2*, p, c, v)*
7. *Výstup:* 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.

[6] 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`

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

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:

```
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.

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 jednořádkovým. 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:

```
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.

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

Výsledková listina čtvrté série dvacátého sedmého ročníku KSP

	řešitel	*škola*	*ročník*	*sérií*	*4-1*	*4-2*	*4-3*	*4-4*	*4-5*	*4-6*	*4-7*	*série*	*celkem*
0.					10	11	11	11	12	11	14	59,0	241,0
1.	Jan Špaček	G Wicht	4	9	10	11	11	11	12	11	14	59,0	232,9
2.	Richard Hladík	GOAMarLaz	2	14	10		9	11	12	11	9	52,3	212,6
3.	Stanislav Lukeš	GPísnickáPH	2	5	10	7	11		12		2	44,9	206,6
4.	Marek Černý	G Chrudim	4	9					12	11		23,0	183,7
5.	Martin Scheubrein	G MNám Třb	3	4			7		5	11	8,5	36,4	177,0
6.	Štěpán Hudeček	G Litovel	3	4	4	10	11			11	9,5	50,8	176,2
7.	Václav Volhejn	GKepleraPH	2	14	10	11	10	1	12	11		53,6	162,8
8.	Jakub Tětek	CírkG Plzeň	1	4	2		4	11			2	23,3	156,9
9.	Michal Převrátil	GKlatovy	2	4	8	10	7		12	11		51,0	153,4
10.	Přemysl Šťastný	GŽamberk	2	7	2	2	4		12			21,7	143,0
11.	Václav Šraier	GČeskoliPH	2	4	9				6			15,6	142,6
12.	Jan Tománek	GPelhřimov	4	4								0,0	135,9
13.	Michal Töpfer	G DrJPekMB	2	4	6	7	8		0		8	34,4	130,7
14.	Lukáš Ulrich	SSŠVTPraha	4	3								0,0	114,5
15.	Jan Kočur	G Wicht	4	3								0,0	113,4
16.	Adrián Goga	SPŠNitra	4	3								0,0	110,8
17.	Jan Knížek	G Strakon	4	16								0,0	107,7
18.	Jakub Zárybnický	GTomkovaOL	4	8								0,0	102,7
19.	Anna Gajdová	GFPValMez	4	4								0,0	97,3
20.	Róbert Selvek	G KošiceS	3	3								0,0	96,4
21.	Pavel Turinský	G Brandýs	2	4			11				11,5	24,0	89,4
22.	Jan Gocník	GJŠkodyPŘ	3	3								0,0	74,1
23.	Jiří Sejkora	GVoděraPH	3	3								0,0	68,7
24.	Václav Rozhoň	GJirsíkaČB	4	9	10		11	11	12	11		55,0	68,6
25.	Jiří Vozár	G UherBrod	3	4							8	10,6	68,1
26.	David Cholewa	GMatOS	4	2								0,0	46,2
27.	Jan Bouček	GKepleraPH	2	3					12			12,0	45,2
28.	Václav Končický	GSOŠ FrMís	4	3					4		4,5	10,2	43,1
29.	Jan Pokorný	G_Bučovice	3	6	9							9,5	41,0
30.	Martin Zoula	GNadKavaPH	3	3								0,0	38,6
31.	Barbora Sedláková	GKonštanPV	4	3					0		5	8,5	35,9
32.	Jan Soukup	GKlatovy	4	3						11		11,0	33,0
33.	Eva Matoušková	G_Sokolov	4	1	8	8					1,5	23,4	23,4
34.	Filip Bialas	GOpatovPHA	2	5								0,0	20,0
35.	Vít Macura	GOAMarLaz	2	2								0,0	18,9
36.	Jakub Lukeš	GNAlejíPH	2	1								0,0	14,0
37.	Dalimil Hájek	GKepleraPH	4	15								0,0	13,4
38.	Martin Kubeša	GJŠkodyPŘ	3	1								0,0	12,8
39.	Jakub Matěna	GČeskoliPH	3	2					1			1,0	11,0
40.	David Juřica	GNadŠtolPH	2	2								0,0	10,9
41.	Jan Kaifer	GČesBrod	−1	1								0,0	10,6
42.	Matěj Konečný	GJírovcČB	4	2								0,0	10,0
43.	Jan Burda	G_Holice	1	1								0,0	9,0
44.	Václav Steinhauser	GDačice	1	1								0,0	7,9
45.	Josef Vávra	SJec	4	1								0,0	5,7
46.	Jan Mráz	G_Holice	1	1	0	0,5	1	0	1			4,4	4,4
47.	František Dostál	VSPŠEOc	4	1								0,0	4,0
48.	Michael Novák	SSŠVTPraha	4	1								0,0	2,0

Chcete-li s námi komunikovat šifrovaně a bezpečně, můžete si ověřit náš HTTPS certifikát – jeho SHA1 fingerprint je: 0E:D9:B6:E5:6F:B0:51:D9:66:EB:E9:29:E4:58:AB:5F:99:D6:FD:A3.