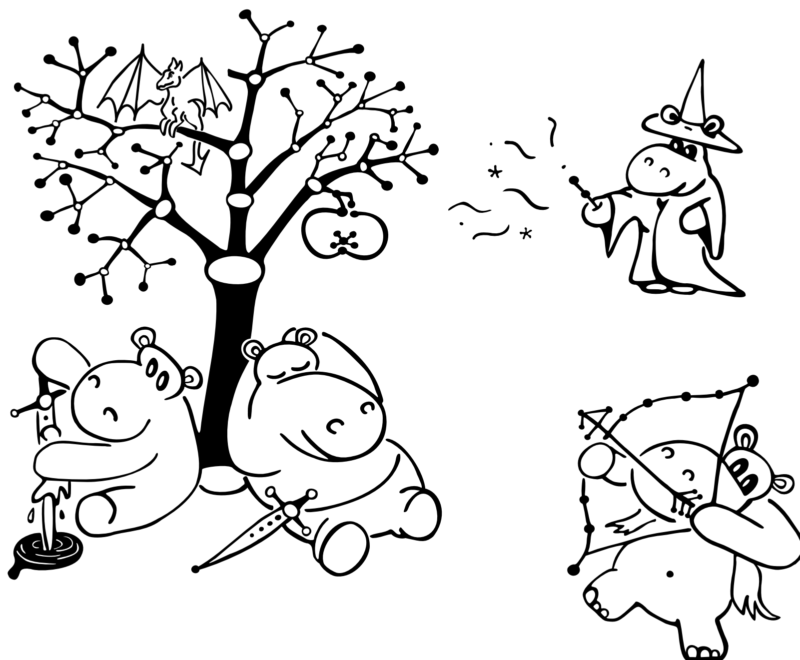
JIŘÍ SETNIČKA A KOLEKTIV

Korespondenční seminář z programování

XXIX. ročník – 2016/2017



matfyzpress

NAKLADATELSTVÍ MATEMATICKO-FYZIKÁLNÍ FAKULTY
UNIVERZITY KARLOVY

MATEMATICKO-FYZIKÁLNÍ
FAKULTA
Univerzita Karlova

JIŘÍ SETNIČKA A KOLEKTIV

Korespondenční seminář z programování

XXIX. ročník – 2016/2017

matfyzpress

Praha 2017

© Jiří Setnička a kolektiv, 2017
© MatfyzPress, nakladatelství Matematicko-fyzikální fakulty
Univerzity Karlovy, 2017

ISBN 978-80-7378-355-6

Úvod

Právě máte před sebou ročenku 29. ročníku Korespondenčního semináře z programování (dále jen *KSP*), který i tento rok pokračoval ve své činnosti. Po dobu své historie patří k nejznámějším aktivitám pro zájemce o informatiku a programování z řad studentů (nejen) středních škol. Díky aktivnímu zapojování se do řešení úloh získalo mnoho řešitelů praxi ve zdolávání nejrůznějších algoritmických problémů, jakož i hlubší náhled do tajů informatiky.

Zároveň to letos byly již čtyři roky od založení začátečnické kategorie zvané *KSP-Z*, které se daří a stále přitahuje nové a nové řešitele. Vedle ní samozřejmě dále pokračovala i hlavní kategorie, která pokračuje nepřetržitě od založení KSP.

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

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

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

Stejně jako v loňském úvodu ročenky zopakujeme i pokračování dalších úspěšných aktivit, které původně vzešly z KSP. Podobně jako loni pokračovaly svým již pátým rokem Putovní přednášky[1] na středních školách po celé republice. Naše spřátelená, ale nyní již samostatná soutěž Kasiopea[2] uspořádala již svůj druhý ročník ve velkém formátu včetně velkého finále pro dvacet nejlepších soutěžících.

[1] `http://ksp.mff.cuni.cz/akce/putovni-prednasky/2016/`

[2] `http://kasiopea.matfyz.cz`

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

Některé značky používáme primárně k označení typu úlohy:

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

Tento symbol označuje praktickou úlohu. V KSP se můžete potkat s takzvanými *open-data* úlohami. Úkolem řešitelů je nejen vymyslet algoritmus, ale také ho zapsat jako program a tento program odladit. Odevzdání probíhá tak, že si řešitel stáhne vstupní data a odevzdá příslušný výstup, přičemž počet pokusů není omezen.

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

Jiné značky slouží k označení obtížnosti a doporučených zdrojů inspirace:

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

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

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

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

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

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

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

Obsah

Úvod ... 3
Obsah ... 6
KSP-Z ... 7
Zadání úloh KSP-Z ... 8
- První série ... 8
- Druhá série ... 12
- Třetí série ... 16
- Čtvrtá série ... 21

Vzorová řešení KSP-Z ... 25
- První série ... 25
- Druhá série ... 33
- Třetí série ... 42
- Čtvrtá série ... 51

Pořadí řešitelů KSP-Z ... 58
KSP ... 63
Zadání úloh KSP ... 64
- První série ... 64
- Druhá série ... 73
- Třetí série ... 80
- Čtvrtá série ... 88
- Pátá série ... 98

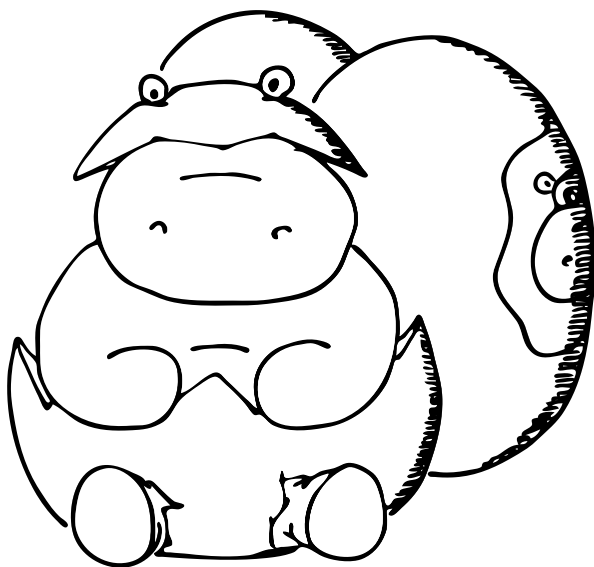
Seriál – Stromy ... 107
Recepty z programátorské kuchařky ... 137
- Kuchařka první série – základní algoritmy ... 137
- Kuchařka druhé série – hledání v textu ... 161
- Kuchařka třetí série – rozděl a panuj ... 175
- Kuchařka čtvrté série – geometrie ... 183
- Kuchařka páté série – dynamické programování ... 193

Vzorová řešení KSP ... 203
- První série ... 203
- Druhá série ... 219
- Třetí série ... 231
- Čtvrtá série ... 247
- Pátá série ... 262

Pořadí řešitelů KSP ... 275

KSP-Z

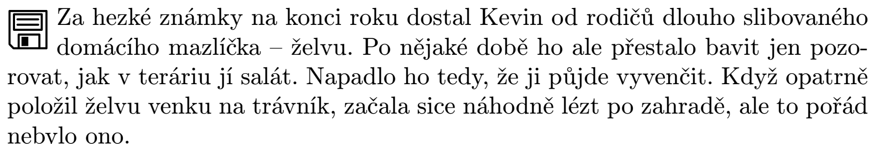
Začátečnická kategorie KSP



Zadání úloh KSP-Z

První série

29-Z1-1 Kevinova želva — 8 bodů



Za hezké známky na konci roku dostal Kevin od rodičů dlouho slibovaného domácího mazlíčka – želvu. Po nějaké době ho ale přestalo bavit jen pozorovat, jak v teráriu jí salát. Napadlo ho tedy, že ji půjde vyvenčit. Když opatrně položil želvu venku na trávník, začala sice náhodně lézt po zahradě, ale to pořád nebylo ono.

Pak to Kevina napadlo: želvu si vycvičí! Naučí ji lézt jen v kolmých směrech. A nejen to, želva bude lézt právě do směrů světových stran, tj. na sever, na jih, na západ a na východ. Kevin vždy dá želvě pokyn a ona popoleze o jednu želví délku v tom směru.

Takové cvičení funguje velmi dobře. Možná až moc. Tomu ještě nedávno línému spotřebiči salátu teď začíná být zahrada malá. Kevin ji vždy položí uprostřed zahrady, a než ji pustí, potřeboval by zjistit, kde podle jeho plánu příkazů skončí. Protože má ale plné ruce želvy, nedokáže to. Pomůžete mu?

Tvar vstupu a výstupu: Na vstupu dostanete na prvním řádku číslo, které udává počet příkazů, a na druhém tyto příkazy vypsané. Vaším úkolem je vypsat souřadnice, kde želva skončí. Začíná v bodě `0 0`, kde první číslo udává vzdálenost od počátku ve směru východ-západ (východ je směr kladný) a druhé číslo je vzdálenost ve směru sever-jih (sever je směr kladný). Jedna želví délka je dlouhá 1.

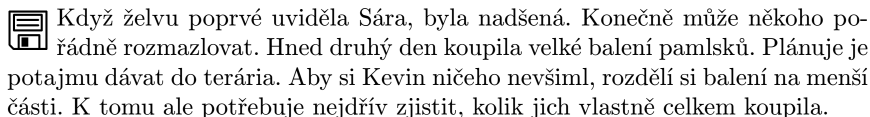
Ukázkový vstup:

```
11
SVJJVZSZVSS
```

Ukázkový výstup:

```
1 2
```

29-Z1-2 Sářiny pamlsky — 10 bodů



Když želvu poprvé uviděla Sára, byla nadšená. Konečně může někoho pořádně rozmazlovat. Hned druhý den koupila velké balení pamlsků. Plánuje je potajmu dávat do terária. Aby si Kevin ničeho nevšiml, rozdělí si balení na menší části. K tomu ale potřebuje nejdřív zjistit, kolik jich vlastně celkem koupila.

Aby se při počítání nespletla, bude si vždy značit každé tři a každých pět započtených pamlsků. Násobky tří si označí křížkem a násobky pěti si označí kolečkem. V případě, že dojde k číslu, které je násobkem trojky i pětky, pamlsek raději sní sama.

Po chvíli je Sára z té jednotvárné práce už unavená, pomůžete jí chvilku počítat?

Tvar vstupu: Dostanete jeden řádek se dvěma čísly – číslo pamlsku, u kterého je Sára teď, a druhé číslo, ke kterému se máte dopočítat, než to Sáře zase předáte. Čísla jsou oddělená mezerou.

Tvar výstupu: Vaším úkolem je vždy, když narazíte na násobek tří, vypsat křížek (`#`), u násobků pěti kolečko (velké písmeno `O`). Pokud jde o násobek obou, tak číslo nepište vůbec. Všechna ostatní čísla vypište tak, jak jsou.

Ukázkový vstup:

```
8 18
```

Ukázkový výstup:

```
8 # O 11 # 13 14 16 17
```

29-Z1-3 Petrova statistika — 10 bodů

Když Sára s vaší pomocí vše spočítala a pečlivě rozdělila pamlsky, začala jimi želvu krmit. Nikdo ale neodolá smutnému želvímu pohledu dlouho, a tak jí dávala pamlsků někdy více. Někdy se zase snažila, aby želva jedla zdravě, a tak pamlsky omezila. Nakonec jí dávala různé dny různé množství pamlsků.

Když to uviděl kamarád Petr, byl z jejího počínání celý zmatený a rozhodl se v tom udělat nějaký pořádek. Chtěl by znázornit, kolikrát dala Sára želvě jaký počet pamlsků, ale neví, jak na to. Zkusíte to vy?

Tvar vstupu: Na prvním řádku dostanete číslo udávající počet dnů. Na dalším řádku je vypsané, kolik pamlsků želva za den dostala.

Tvar výstupu: Vaším úkolem je znázornit pomocí hvězdiček, kolikrát dala Sára želvě jaký počet pamlsků. Začněte nejmenším číslem na vstupu a u každého čísla (i toho, které se na vstupu nevyskytuje vůbec) vypište hvězdičku za každý pamlsek. Skončete s největším číslem, které se na vstupu ještě objeví.

Ukázkový vstup:

```
6
2 4 2 6 3 2
```

Ukázkový výstup:

```
2:***
3:*
4:*
5:
6:*
```

29-Z1-4 Zuzčin výlet — 12 bodů

Když po pár dnech Kevinova mladší sestra Zuzka viděla, že ti tři dělají z želvy vším tím venčením, cvičením a pamlsky psa, rozhodla se je na chvíli vytáhnout na výlet. Vymyslela, že všichni půjdou na koupaliště, a želva si tak bude moci na chvíli odpočinout.

Na koupališti je spousta skluzavek a tobogánů. Aby si nějaké mohli zkusit, musí vylézt jedno velmi vysoké schodiště, nebo zaplatit nehoráznou částku za výtah.

Na nejvyšším místě začíná několik skluzavek. Ty ale nemusí vést až dolů na zem, některé končí v přestupních bazéncích. Ke každému bazénku vede jedna skluzavka shora a několik dalších vede směrem dolů. Zuzka s sebou má i mapu všech těchto skluzavek a tobogánů. Dokážete vymyslet, jakou skluzavku si mají kdy vybrat, aby zkusili co nejvíce různých skluzavek?

Bazénky jsou očíslované čísly 1 až N. Nejvyšší bazének hned u výstupu z výtahu má číslo 1 a bazénky, ze kterých už nevede žádná skluzavka dolů, jsou na úrovni země.

Tvar vstupu: Dostanete nejprve číslo N. Následuje $N-1$ řádků dvojic i j, které říkají, že z bazénku i vede skluzavka dolů do bazénku j.

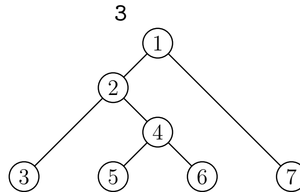
Tvar výstupu: Vypište jediné číslo – nejvyšší počet skluzavek, který může Zuzka vyzkoušet za jednu cestu dolů.

Ukázkový vstup:

```
7
1 2
2 3
2 4
4 5
4 6
1 7
```

Ukázkový výstup:

```
3
```



29-Z1-5 Dva seznamy — 12 bodů

Když Kevin a Sára vyzkoušeli skluzavky, napadlo je, že by si mohli zahrát nějakou hru. Všimli si totiž, že na koupališti je spousta jejich spolužáků, se kterými by se dalo něco podniknout.

Zatímco Kevin plánoval hrát s ostatními u bazénu volejbal, Sára by chtěla uspořádat velkou bitvu s balónky naplněnými vodou. Rozhodli se tedy, že zjistí, kdo z jejich kamarádů by chtěl hrát co.

Kevin obcházel všechny a ptal se, kdo chce hrát volejbal. Sára mezitím začala zjišťovat, kdo by chtěl vodní bitku, a obcházela spolužáky také, akorát v jiném pořadí.

Když se opět sešli, zjistili, že mají dva seznamy, na kterých je hodně jmen, a tak je napadlo, že si zahrají obojí. Chtěli by tedy teď zjistit, kdo všechno by se chtěl účastnit obou her zároveň.

Pomozte Kevinovi se Sárou a vymyslete, jak byste situaci vyřešili programem. Máte k dispozici dvě pole řetězců, která obsahují A a B jmen. Vymyslete co nejrychlejší algoritmus vzhledem k A a B, který vytvoří třetí pole, kde budou jen ta jména vyskytující se v obou vstupních polích.

Zajímá nás časová a paměťová složitost vašeho algoritmu v nejhorším případě. Pokud vůbec nevíte, o čem mluvíme, určitě se podívejte do naší programátorské kuchařky o složitosti.[3]

Při počítání složitosti uvažujte, že jména mají maximálně 42 znaků, tedy na délce jednotlivých jmen příliš nezáleží.

Ukázkový vstup:

```
Kevin Zuzka Petr
Petr Zuzka Sára
```

Ukázkový výstup:

```
Petr Zuzka
```

29-Z1-6 Devadesát devět pater — 14 bodů

Než Kevin a Sára řeknou ostatním, co se bude hrát, Petr a Zuzka už začínají připravovat balónky. Práce je to docela nudná, a tak u ní vymýšlejí pořádnou strategii na hru.

Všimli si, že balónky jsou docela odolné, takže když se hodí moc zblízka, neprasknou. Kdyby ale balónky házeli příliš daleko, je malá šance, že protivníka trefí. Jaká je tedy ta nejlepší vzdálenost, ze které balónek házet?

Petr a Zuzka to chtějí zjistit tak, že budou házet naplněné balónky z jednotlivých pater schodiště ke skluzavkám. Budou zkoumat, ze kterého nejnižšího patra balónek po pádu praskne.

Bohužel pro ně, pokaždé, když balónek hodí, hrozí, že je při tom chytí plavčík. Házet balónky z výšky do prostoru, kde se pohybují lidé, je totiž nebezpečné a určitě by je vykázal ven z koupaliště. Chtějí tedy najít způsob, jak zjistit tu správnou výšku na co nejméně hození balónku.

Předpokládejte, že Zuzka má dostatečnou zásobu balónků a že schodiště má celkem 99 pater. Poraďte Petrovi algoritmus, jakým má patra zkoušet, aby našel tu správnou výšku. Počítejte s tím, že může mít velkou smůlu, a uveďte u algoritmu počet pokusů v tom nejhorším možném případě.

Pokud si s devadesáti devíti patry nevíte rady, můžete získat část bodů, pokud vyřešíte úlohu pouze pro devět pater. Naopak, pokud si troufnete, zkuste napsat řešení, které bude nejlepší možné vzhledem k neznámému libovolnému počtu pater P.

Příklad: Petr může zkoušet patra postupně odspodu. Pokud balónek nikdy nepraskne, vyzkouší všech 99 pater. Tato strategie je tedy ze všech správných nejhorší možná :)

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

Druhá série

29-Z2-1 Krocení zlé želvy 8 bodů

Kevin a jeho kamarádi mají za sebou první nudné týdny školy a už zase vymýšlejí, co zajímavého by mohli podniknout. Před prázdninami dostal Kevin od rodičů želvu jménem Želva a tenkrát se ji rozhodl vycvičit. Tak proč dnes při dlouhém podzimním večeru v krocení nepokračovat?

Spolu se Sárou dávají Želvě pokyny „popojdi dopředu“, „otoč se na místě doprava“ a „otoč se na místě doleva“. Protože není Kevinův pokojíček moc velký, potřebují si radši předem rozmyslet, kde Želva po sérii pokynů vlastně skončí.

Želvičku na začátku položí do středu pokojíku na pomyslné souřadnice $[0, 0]$ natočenou severním směrem a zajímalo by je, kde skončí. Když vám Kevin a Sára postupně poví plánované příkazy, dokážete říct, kam se Želva dostane?

Tvar vstupu: Na prvním řádku od nich dostanete celkový počet pokynů, tedy jedno číslo. Na druhém řádku bude odpovídající počet znaků, kde `>` značí „otoč se doprava“, `<` znamená „otoč se doleva“ a `A` je „popojdi dopředu“.

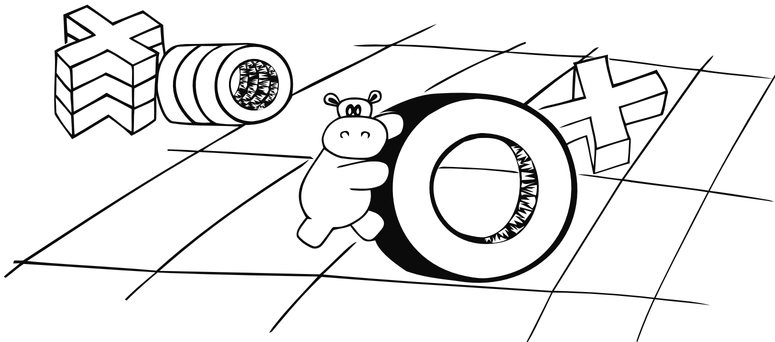
Tvar výstupu: Od vás by potřebovali dostat dvě čísla (souřadnice) na jednom řádku oddělená mezerou. První říká, jak daleko je želva v pomyslném západně-východním směru, a to druhé odpovídá severo-jižnímu směru.

Ukázkový vstup:

```
7
>A><AA>
```

Ukázkový výstup:

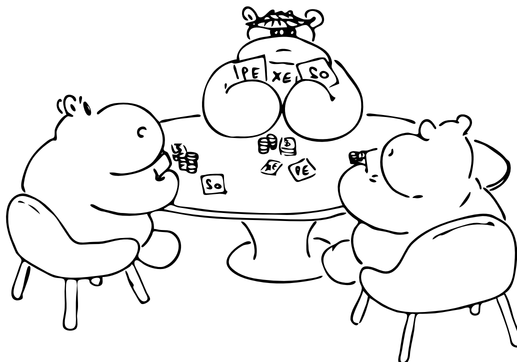
```
3 0
```



29-Z2-2 Sářina volba 10 bodů

Když už Sáru přestalo krocení želvy Želvy bavit, začala přemýšlet nad plány na víkend. Uvažovala, že vyrazí s Petrem za dobrodružstvím. To se Kevinovi moc nezdálo a říkal, že by raději šel na výlet. Jak teď rozhodnout?

No jak jinak než si dát kámen-nůžky-papír. Oba se shodli, že rozhodnutí bude spravedlivější, když kámen-nůžky-papír zopakují několikrát za sebou. Plány na víkend rozhodne ten, který vyhraje vícekrát. Zjistíte, kdo to bude, jestliže víte, kolikrát celkem hráli a jak táhli?



Tvar vstupu: Na prvním řádku dostanete počet her, na druhém uvidíte, jaké tahy zvolila Sára, a na třetím jak volil Kevin (písmena K, N a P).

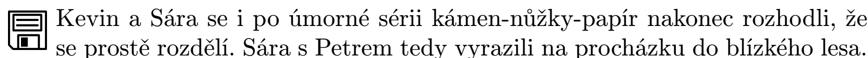
Tvar výstupu: Na jeden řádek vypište dvě čísla oddělená mezerou. První bude počet výher Sáry a to druhé udá, kolikrát zvítězil Kevin. V případě, že oba táhnou stejně, výhra se nepočítá ani jednomu.

Ukázkový vstup:

```
4
KNPK
NPPP
```

Ukázkový výstup:

```
2 1
```



29-Z2-3 Petr v říši divů 10 bodů

Kevin a Sára se i po úmorné sérii kámen-nůžky-papír nakonec rozhodli, že se prostě rozdělí. Sára s Petrem tedy vyrazili na procházku do blízkého lesa.

Ale co vypadalo jako obyčejný smrkový háj, se velmi rychle stalo nepronik-nutelným bludištěm. Jako by se najednou ocitli v nějaké pohádce. Petr a Sára z něj nemohou najít cestu ven. Petr začíná panikařit a tam Sára rychle přichází s plánem, jak zmapovat situaci.

Není náhoda, že lesy v pohádkách připomínají pravidelnou mřížku. Rozhodne se tedy, že zkusí zjistit počet pro ně dostupných políček. Dostupná políčka jsou taková, na která lze dojít z počátečního, pokud smíme dělat kroky jen vodorovně a svisle, ne šikmo. Když vám ukážeme mapu lesa, dokážete Sáře poradit?

Tvar vstupu: Na prvním řádku najdete šířku a výšku mapy, v tomto pořadí. Místo, kde s Petrem stojí, je označené `P`, volná políčka jsou `.` a neprostupná značíme `#`.

Ukázkový vstup:

```
8 4
.###..##
..#.P..#
##...###
####....
```

Ukázkový výstup:

```
0.9
13
```

29-Z2-4 Zuzka: Cesta tam a zase zpátky — 12 bodů

Zatímco Petr a Sára bloudili v lese, Kevin vyrazil se Zuzkou na túru do hor. Rozhodli se, že budou velmi pečlivě zaznamenávat údaje o své cestě, protože nikdy nevíte, na co se to může hodit. Mají GPS, která každou sekundu zaznamenává jejich nadmořskou výšku.

Po pár kilometrech se už Zuzka cítí dost zničená a připadá si, jako by chodili pořád jen do kopce. Tvrdí, že už nevydrží jít do kopce ani minutu. Kevin jí to chce vymluvit, ale neobejde se bez pádných argumentů.

Chtěl by v záznamech najít co nejdelší úsek cesty, ve kterém šli maximálně minutu do kopce. Protože si ale Zuzka může vymyslet jakýkoliv jiný časový údaj než minutu, rád by to uměl spočítat pro zadané K. Pomůžete mu?

Dostanete čísla N a K, následované N záznamy nadmořské výšky. Najděte nejdelší úsek, ve kterém šli Kevin se Zuzkou nejvýše K sekund do kopce a vypište, kolik sekund tento úsek trval.

Ukázkový vstup:

```
9 2
1 2 4 3 1 -1 0 1 3
```

Ukázkový výstup:

```
5
```

Hledaný nejdelší úsek začíná dvojkou a končí nulou. Úsek obsahuje šest záznamů, ale mezi těmito záznamy uplynulo pět sekund.

29-Z2-5 Dva roky bez prázdnin — 12 bodů

Sára a Petr prohledávali les a najednou se před nimi zničeho nic objevil kocour Šklíba. Prý jim ukáže cestu, když mu pomohou vyřešit jeden problém.

Šklíba má starosti jako každý smrtelník, a tak ho mimo jiné trápí nekonečné množství spamů a řetězových e-mailů. Určitě to také znáte, jeden hrozí za nepřeposlání vynálezem zkázy, jiný slibuje každému, kdo jej nepřepošle alespoň dvaceti pohádkovým bytostem, strávit zbytek života dvacet tisíc mil pod mořem.

Poslední dobou se v říši divů rozmohl obzvláště otravný e-mail hrozící, že každý, kdo ho nepřepošle, bude dva roky bez prázdnin. Tohle bylo už i na Šklíbu moc, protože se rozšířil ještě více než ty předchozí. Po Petrovi a Sáře chce vypátrat, kdo je autorem tohoto spamu.

Původce e-mail rozeslal K bytostem. Každá z nich jej buď smazala, nebo přeposlala zase K lidem. Dobrou zprávou je, že nikdo nedostal e-mail dvakrát. Šklíba má k dispozici informace, mezi kterými dvojicemi putoval, ale nedokázal už zjistit, kterým směrem (tzn. kdo jej poslal komu). Dokážete vymyslet způsob jak odhalit, kdo začal tento řetězový e-mail a pomoci tak Sáře s Petrem?

29-Z2-6 Devět trpaslíků — 14 bodů

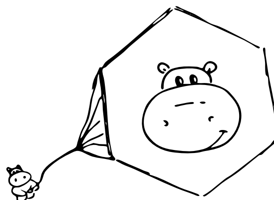
Zuzka s Kevinem pokračovali v cestě, když se najednou začalo až překvapivě rychle stmívat. Rozhodli se nic neriskovat a vrátit se zpátky, ale kudy? Tam, kde se v mapce nachází turistická stezka, je ve skutečnosti hustý neproniknutelný les.

Chvíli bloudili sem a tam, tam a sem, a když už jim začínalo být úzko, potkali Sáru s Petrem. Ty poslal Šklíba hledat pomoc k devíti trpaslíkům. Kevin se Zuzkou se k nim přidali a po chvíli Sněhurku i se všemi devíti trpaslíky našli.

Ti už pro naše čtyři kamarády měli připravený další úkol. Dostanou řadu N čísel v jistém pořadí a čas na přípravu. Pak jim budou trpaslíci klást spoustu dotazů typu: „Jaký je součet čísel mezi a-tým a b-tým číslem?“ Poradíte jim, jak se co nejlépe připravit, aby takový dotaz mohli zodpovědět co nejrychleji?

Sněhurka chce navíc vědět, kolik času je na přípravu potřeba, co si budou muset pamatovat navíc a samozřejmě i čas potřebný na zodpovězení jednoho dotazu.

Pokud to potřebujete, předpokládejte, že dotazů bude řádově N.



Třetí série

29-Z3-1 Želva na dvorku — 8 bodů

„Stůj! Vždyť nám zbouráš sněhuláka!“ křičí Kevin. Výcvik Želvy šel totiž dobře, skutečně se naučila některé základní povely. Proto zkusili vzít Želvu ven, aby viděli, jak se popasuje s terénem. Jenže se Sárou se občas na povelech neshodnou a Želva slepě naráží do překážek.

Na čtverečkovaném zasněženém trávníku leží P překážek, u každé znáte její souřadnice a víte, že zabírá právě jedno políčko. Tam kde je překážka, nemůže Želva vkročit. Dokážete z povelů určit, kde Želva skončí?

Tvar vstupu: Na prvním řádku dostanete čísla P a N, kde P je počet překážek a N je počet povelů. Následuje P řádků se souřadnicemi překážek, pokaždé nejprve vzdálenost v západně-východním směru, potom v severo-jižním.

Vstup končí řádkem s N znaky povelů. Želva rozumí třem povelům: `<` znamená otoč se proti směru hodinových ručiček, `>` otoč se po směru hodinových ručiček a `A` popojdi o jedno políčko dopředu.

Želva začíná jako obvykle na políčku $[0, 0]$, otočená na sever. Pokud povel nelze provést, Želva jej přeskočí. Překážka se nikdy nebude nacházet na počátečních souřadnicích Želvy.

Tvar výstupu: Vypište dva řádky. Na jednom budou souřadnice Želvy po tom, co vykoná nebo přeskočí všechny povely. Na druhém číslo, kolikrát želva v průběhu narazila do překážky.

Ukázkový vstup:

```
1 5
0 1
AAA>A
```

Ukázkový výstup:

```
1 0
3
```

Želva se v příkladu třikrát pokusí narazit do překážky, teprve pak se otočí a popojde.

29-Z3-2 Písemka z angličtiny — 10 bodů

Želva ovšem není zvíře zvyklé na zimu, proto museli rychle domů. Také Kevin se už musí učit, hned po prázdninách píšou písemku z angličtiny. Aby prospěl, musí se naučit velkou spoustu slovíček.

Ze zoufalství Kevina napadlo, že si vyrobí tahák. Ne že by ho chtěl při písemce použít, to se přeci nesmí, ale výrobou taháku se člověk nejvíce naučí.

Slovíčka, která se Kevin učí, mají zajímavou vlastnost: velmi často se stává, že přilepením přípony za jedno slovíčko vznikne druhé. Dalo by se také říci, že první slovíčko je *prefixem* druhého.

Například slovo „hra“, je prefixem slova „hračka“. Nebo třeba slova „a“, „advent“ a „adventure“ jsou prefixy slova „adventurer“.

Aby Kevin uspořil co nejvíce místa, napadlo ho, že napíše na tahák takové slovíčko, které má v sobě nejvíce jiných slovíček jako prefix. Jednotlivé prefixy pak označí a přeloží zvlášť, ale nejprve by potřeboval takové slovíčko najít.

Tvar vstupu: Vstupem je slovník. Na prvním řádku je číslo N, označující počet slov ve slovníku. Následuje N řádků se slovy složenými z malých písmen anglické abecedy.

Tvar výstupu: Na výstup vypište jediné slovo, které má nejvíce jiných slov ze slovníku jako prefix.

29-Z3-3 Šestková čísla — 10 bodů

Z pilného učení vytrhl Kevina až Petr, který přiběhl s revoluční myšlenkou. Vymyslel totiž nový způsob kódování čísel – šestková čísla.

Každému, kdo zná římská čísla, bude jejich popis povědomý. Šestková čísla používají místo arabských číslic písmena latinky. Každé písmeno má svou hodnotu podle tabulky.

$$J = 1 \qquad T = 36 \qquad W = 1296 \qquad Y = 6^6$$
$$S = 6 \qquad D = 216 \qquad X = 6^5 \qquad Z = 6^7$$

Písmena, tedy vlastně šestkové číslice, se píší od největší k nejmenší. Potom je hodnota šestkového čísla rovna součtu hodnot jednotlivých číslic. Příklady:

$$SJ = 7, \quad JJJJ = 4$$

Pokud jsou číslice uvedené v jiném pořadí, znamená to, že se hodnoty odečítají. Je však možné odečítat pouze číslici o jedna menšího řádu, a to nejvýše dvakrát. Odčítané číslice navíc musí být zapsané právě před poslední číslicí většího řádu. Například:

$$JS = 6 - 1 = 5$$
$$SJJS = 6 + (6 - 1 - 1) = 10$$
$$DST = 216 + (36 - 6) = 246$$
$$DSSTJ = 216 + (36 - 6 - 6) + 1 = 241$$
$$DSTJS = 216 + (36 - 6) + (6 - 1) = 251$$

Netrvalo dlouho, než si naši kamarádi uvědomili, že tento zápis není jednoznačný. Například číslo čtyři se dá zapsat buď s odčítáním jako JJS, nebo bez něj jako $JJJJ$. Rozhodli se, že jako hlavní označí zápis, který nemá nikdy čtyři stejné číslice za sebou. Pouze číslice Z smí být zapsána, kolikrát je potřeba.

Navíc, číslo v hlavním zápisu nesmí mít nikdy zbytečné číslice navíc. V čísle JSJ se jednička odečítá, aby se vzápětí zase přičetla. Hlavní zápis čísla šest je samozřejmě S.

Aby se jim s čísly lépe pracovalo, rádi by si pořídili takový program, který dokáže načíst číslo v libovolném platném zápisu a převést ho do zápisu hlavního. A protože to chtějí rozjet ve velkém, potřebují, aby to program dokázal s více čísly v jednom souboru.

Tvar vstupu: Na prvním řádku je číslo T (v obyčejném desítkovém zápisu), které označuje, kolik vstupů se v souboru nachází. Následuje T řádků, na každém je jedno číslo v šestkovém zápisu. Každé číslo je platné podle výše uvedených pravidel, ale nemusí být v hlavním zápisu.

Tvar výstupu: Vypište čísla ze vstupu v hlavním šestkovém zápisu, každé na zvláštní řádek.

Ukázkový vstup:

```
3
JJJJ
SSSSSS
JJSJJ
```

Ukázkový výstup:

```
JJS
T
S
```

29-Z3-4 Zdobení stromečku 12 bodů

Zatímco si kluci hrají s čísly, Sára a Zuzka vzpomínají, jak zdobily vánoční stromeček. To byla krása, takových ozdob! Už dlouho se jim to takhle nepovedlo.

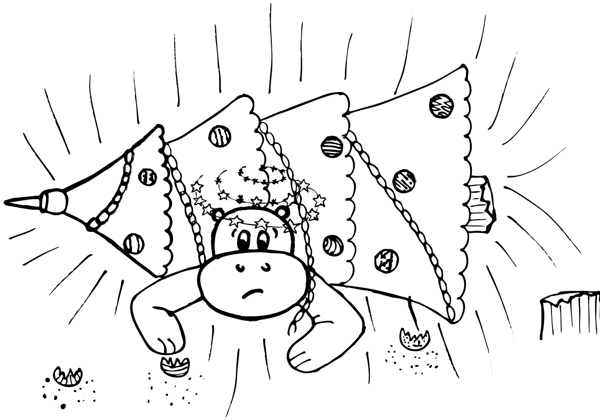
Každá ozdoba má tvar barevného blýskavého řetězu, který se zavěsí mezi dvě větve. Ozdobné řetězy se můžou libovolně křížit, z jedné větve jich může vést libovolně mnoho. Řetězy se ovšem nedají věšet jinam než mezi dvě špičky větví. Navíc, řetěz se nesmí pověsit mezi dvě větve, mezi kterými už řetěz visí.

Na původně holý stromeček holky postupně zavěšovaly řetězy, dokud si toho Sára nevšimla. „Tohle přeci není strom, vždyť obsahuje kružnice!“ A měla pravdu. V terminologii grafů bychom mohli říct, že větve jsou vrcholy a řetězy mezi nimi tvoří hrany. Po chvíli zdobení se na našem stromečku objevil cyklus.

Bez použití grafových pojmů to znamená, že začneme-li na větvi S, můžeme po řetězích přeskákat několik jiných větví a skončit opět na větvi S, aniž bychom použili nějaký řetěz dvakrát.

Teď se ale kamarádky nemůžou dohodnout, která z nich vlastně přidala onen řetěz, který vytvořil první kružnici. Pomůžete jim?

Tvar vstupu: Na prvním řádku jsou čísla N a M. N je počet větví, které jsou očíslovány $1 \ldots N$. M je počet řetězů. Na dalších M řádcích jsou vždy dvě čísla větví, mezi kterými je pověšen řetěz. První řetěz má číslo 1. Řetězy jsou v souboru v takovém pořadí, v jakém byly věšeny na stromeček.



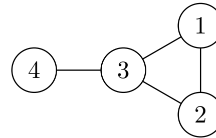
Tvar výstupu: Na výstup vypište číslo řetězu, který vytvořil první kružnici, tedy udělal z lesa obecný graf.

Ukázkový vstup:

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

Ukázkový výstup:

```
4
```



29-Z3-5 Prvočíselné rozklady — 12 bodů

Vrátíme se k číslům. Určitě už jste slyšeli, co to je prvočíslo. Pro jistotu: prvočíslo je takové číslo, které je beze zbytku dělitelné pouze jedničkou a sebou samým.

Každé číslo je možné zapsat jako součin jiných prvočísel, potom mluvíme o prvočíselném rozkladu. Ten je až na zbytečné jedničky a pořadí činitelů jednoznačný.

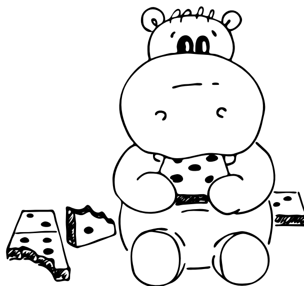
Jak rychle dokážete spočítat prvočíselný rozklad? A jak rychle to dokážete, pokud víte, že dotazů bude více?

Podobně jako v minulé sérii, dostanete číslo M a nějaký čas na přípravu. Potom bude přicházet N dotazů na rozklad čísla $1 < x \leq M$, které byste měli zodpovědět co nejrychleji.

Nezapomeňte ve svém řešení vyjádřit, kolik času a paměti potřebujete na předvýpočet vzhledem k M, a kolik času na jeden dotaz.

29-Z3-6 Trable s dominem 14 bodů

Nevíme jak vy, ale naši seriáloví přátelé o Vánocích také pekli cukroví. Na ty loňské si pořídili speciální formičky, které mají tvar dominové kostky. Každý výsledný kousek cukroví má dvě poloviny a každá z nich má na sobě nějaký vánoční symbol.



Formičky se špatně myjí, proto se kamarádi rozhodli, že použijí jen některé, zato bude od každého zašpiněného druhu dostatečný počet kousků cukroví. Máme tedy jen některé kombinace symbolů, od každé kombinace ale neomezené množství dominových kostiček.

Když dopekli, lákalo je si s cukrovím taky trochu pohrát a postavit dlouhého hada. Rozdělili se na dvě skupiny a každá začala stavět hada z jedné strany. Samozřejmě s tím, že dvě kostičky za sebou musí navazovat společným symbolem.

Když ale dostavěli až k sobě tak zjistili, že nemají způsob, jak dvě části hada spojit. Ať hledali jak hledali, potřebné kostičky jim chyběly. Dokonce jim ani nepomohlo, když části hada posouvali a tím měnili počet kostiček, které se mezi ně vejdou.

Příště už by se tomu již chtěli vyvarovat. Dokážete vymyslet algoritmus, který pro zadaný seznam formiček rozhodne, jestli jde každé dva hady spojit?

Čtvrtá série

29-Z4-1 Šíření viru — 8 bodů

Kevin dostal od svého učitele informatiky domácí úkol, ve kterém si měl vybrat nějaký počítačový virus a popsat jeho chování. Při vybírání Kevina zaujal jeden zvlášť speciální kousek.

Ten se umí šířit počítačovou sítí tak, že každou minutu se nakazí každý počítač, který má alespoň polovinu svých sousedních počítačů nakaženou. Kevina napadlo, že by v rámci domácího úkolu prozkoumal, jak rychle se umí tento virus šířit v určitých počítačových sítích.

Pomozte Kevinovi zjistit, jak dlouho potrvá nakažení celé počítačové sítě, když vybere, které počítače budou nakažené zpočátku.

Tvar vstupu: Na prvním řádku jsou čísla N a K, kde N značí počet počítačů číslovaných $0 \ldots N-1$, z toho K z nich je nakažených v první minutě. Následuje řádek s K čísly nakažených počítačů. Poté následuje N řádků, každý ve tvaru $p, d, s_1, s_2, \ldots, s_d$; kde p je index počítače, d je počet jeho sousedů a s_i jsou jeho sousedé. Vše je odděleno mezerami.

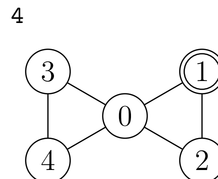
Tvar výstupu: Na výstup vypište minutu, ve které se nakazí poslední počítač. Máte jistotu, že nikdy nenastane situace, že zůstanou nenakažené počítače.

Ukázkový vstup:

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

Ukázkový výstup:

```
4
```



V první minutě je nakažený pouze počítač 1, ve druhé se nakazí 2, ve třetí 0 a ve čtvrté 3 a 4.

29-Z4-2 Vybírání atrakcí — 10 bodů

Skončila škola a Kevin se velmi radoval, že ze svého informatického domácího úkolu dostal nejlepší známku. Proto pozval své kamarády, že se spolu zajdou pobavit do zábavního parku.

Nezbývalo však mnoho času a navíc zábavní park byl ten den velmi vytížen. Na každé atrakci mohl tudíž ten den být jen jeden člověk. Naštěstí atrakcí bylo v tomto parku K-krát více, než byla velikost Kevinovy skupinky.

Kevin ví, že každý ze skupinky snese jinak rychlé atrakce. Někteří mají rádi rychlé a šílené jízdy, někdo jiný zase zvládne nejvýše pomaloučké houpání.

Pomozte Kevinovi přidělit každému kamarádovi jeho K atrakcí tak, aby každá atrakce pro něj byla zvládnutelná a každá atrakce byla celkově použita právě jednou.

Tvar vstupu: Na prvním řádku jsou čísla N a K, kde N je počet lidí ve skupince a $K \cdot N$ je počet atrakcí. Na dalších N řádcích následuje dvojice čísel: nejmenší a největší rychlost atrakce, kterou daný člověk snese. Následuje řádek se seznamem rychlostí jednotlivých atrakcí.

Tvar výstupu: Na jednom řádku bude pro každou atrakci index člověka (indexujeme od nuly), kterému byla daná atrakce přidělená. Atrakce jsou v původním pořadí, jako na vstupu. Máte zaručeno, že řešení vždy existuje, nemusí však být jednoznačné.

Ukázkový vstup:

```
3 3
1 9
4 6
8 12
7 5 10 3 9 4 11 6 5
```

Ukázkový výstup:

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

29-Z4-3 Želva v akváriu — 10 bodů

Další den měl Kevin ve škole biologii, kde zrovna paní učitelka pustila pořad o vodních želvách. Kevin si okamžitě vzpomněl na svou vycvičenou Želvu a věděl, jaký další výcvik ji bude následovat.

Hned po škole se domluvil se Sárou, že si od Petra půjčí akvárium. Po jeho přinesení a naplnění vodou si rozmysleli, že v takovém akváriu plném vody může Želva plavat i nahoru nebo dolů. Tím ale informace, kterým směrem se Želva dívá, přestává být dostatečná na přesný popis polohy Želvy.

Když se Želva dívá na sever a otočí se nahoru o pravý úhel, už se nedívá na žádnou světovou stranu. Stejně tak, pokud se Želva dívá na sever a nakloní se, sice se stále dívá na sever, ale směr „vpravo“ již není na východ. Sára tudíž vymyslela Želvě další čtyři pokyny k otáčení: „Otoč se nahoru“, „Otoč se dolů“, „Nakloň se doleva“ a „Nakloň se doprava“.

Pro akvárium dále Kevin zavedl pomyslné souřadnice, kde $[1, 0, 0]$ míří na sever, $[0, 0, 1]$ směřuje na východ a $[0, 1, 0]$ ukazuje nahoru. Na začátku vždy položí želvu do vody na souřadnice $[0, 0, 0]$ tak, aby se dívala směrem na sever a její pravý bok mířil na východ. Poté je zajímá, na jakých souřadnicích Želva skončí. Předpokládejte, že akvárium je nekonečné do všech šesti směrů.

Když vám Kevin se Sárou postupně poví jednotlivé příkazy, dokážete říct, na jaké souřadnice Želva doplave?

Tvar vstupu: Na prvním řádku je jedno číslo, a to celkový počet příkazů. Na dalším řádku budou samotné příkazy, kde `D` značí „Posuň se dopředu“ a příkazy

`<`, `>`, `^`, `v`, `\` a `/` značí otočení vlevo, vpravo, nahoru, dolů a naklonění vlevo a vpravo.

Tvar výstupu: Od vás budeme potřebovat na jednom řádku tři čísla oddělená mezerou, jenž značí souřadnice, kde želva skončí.

Ukázkový vstup:

```
11
DD>D^D\DDvD
```

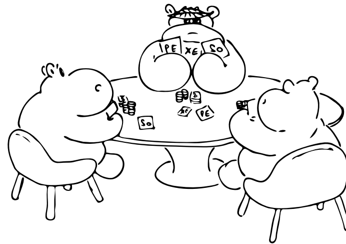
Ukázkový výstup:

```
1 3 1
```

29-Z4-4 Hledání součtu — 12 bodů

Kevin a Sára si bohužel nerozmysleli, že i vycvičená suchozemská želva odmítne plavat pod vodou. Petrovi tudíž následující den po škole akvárium vrátili.

Po navracení akvária nalezli doma u Petra jednu starou deskovou hru, o které dlouho nikdo nevěděl. Jelikož měli domácí úkoly hotové, Kevin, Sára i Petr se rozhodli si hru zahrát.



Hra se skládá z kartiček s kladnými čísly. Na začátku kola se vedle sebe vyloží kartičky s čísly. Dále si každý z hráčů vylosuje své číslo.

Každý z hráčů dále hledá souvislý úsek kartiček tak, aby součet čísel na nich napsaných co nejvíce blížil jejích vylosovanému číslu. Pak vyhrává ten, kdo se svému vylosovanému číslu ve vybraném úseku přiblížil nejlépe. Je jedno, zda je tento součet menší nebo větší, než vylosované číslo. Pomozte Kevinovi tento součet najít.

Na vstupu dostanete na prvním řádku počet kartiček a vylosované číslo. Na druhém řádku následují mezerou oddělená čísla z kartiček, jak jdou vedle sebe.

Na výstup vypište tři čísla oddělená mezerou – index první a poslední karty, které jsou součástí vybraného úseku, a jejich součet. Jako obvykle, indexujeme od nuly. Pokud takových úseků existuje více, vyberte libovolný z nich.

Ukázkový vstup:

```
5 30
28 44 2 21 8
```

Ukázkový výstup:

```
2 4 31
```

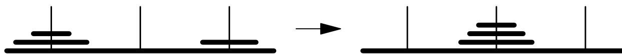
29-Z4-5 Hanojské věže jinak 12 bodů

Kevin pověděl své sestře Zuzce svůj zážitek se starou deskovou hrou. Zuzce se příběh líbil, hned sama začala prohledávat staré krabice s hračkami, jestli nenajde podobný poklad.

Po půl hodince našla starý dřevěný hlavolam podobný Hanojským věžím. Jde o tři tyče, přičemž na tyčích jsou rozmístěny tři disky, každý jiné velikosti. Hlavolam má základní pravidlo, a to že na disku mohou být položeny pouze menší disky. Od skutečných Hanojských věží se liší tím, že počáteční poloha není sestavená věž.

Cílem v této verzi hlavolamu je sestavit věž z disků na libovolnou tyč jejich přesouváním. Jedno přesunutí probíhá tak, že se vždy vezme jeden disk z vrchu jedné tyče a ten se přesune na vršek jiné. Při přesunu se samozřejmě nesmí porušit základní pravidlo.

Na obrázku si můžete prohlédnout možný počáteční a konečný stav hlavolamu.



Po několika málo pokusech se Zuzce podařilo hlavolam vyřešit. Přišlo jí, že to není velmi těžký hlavolam. Rozhodla se jej proto rozšířit tím, že přidala další disky. I po tomto rozšíření Zuzku pokaždé napadlo řešení. Nevěděla však, zda je nejrychlejší.

Vaším úkolem je najít způsob, jak pro dané počáteční rozmístění J disků, splňující základní pravidlo, sestavit na libovolnou tyč celou věž a přitom použít co nejméně přesunů. Součástí úkolu je i ukázat, že je váš způsob ten nejlepší.

29-Z4-6 Tajná síť taháků 14 bodů

Nastal čas pololetních písemek a v Kevinově třídě začala vznikat tajná síť taháků. Podvádění se Kevinovi ale nelíbilo. Už jen z toho důvodu, že kdyby byl kdokoliv chycen, odskáče si to celá třída.

Kevin zjistil, že pokud kdokoliv odešle tahák jednomu kamarádovi, nemůže mu tahák přijít zpět od jiného kamaráda, tedy síť neobsahuje cykly. To znamená, že pokud by někdo ze sítě odešel, síť by se rozpadla na několik menších. A co víc, když nějaká část sítě bude moct posílat taháky mezi méně než polovinu všech původních účastníků sítě, tato část se stane příliš riskantní vzhledem k zisku a všichni účastníci skončí.

Přesvědčit spolužáka, aby ze sítě odešel, je však velmi pracné a zdlouhavé. Navíc je co se učit na písemky. Kevin tudíž ví, že má čas přesvědčit nejvýše jednoho ze svých spolužáků.

Kevin sežene popis sítě. Vaším úkolem bude vybrat spolužáka tak, aby se po jeho přesvědčení síť rozpadla na tak malé části, že celá síť skončí.

Vzorová řešení KSP-Z

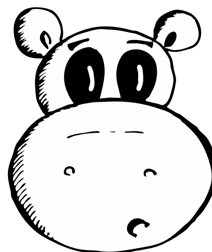
První série

29-Z1-1 Kevinova želva

Pozice želvy je dána dvěma souřadnicemi, proto nám stačí dvě proměnné, do kterých si budeme průběžně ukládat polohu želvy.

Ze vstupu budeme číst písmenka směrů a podle nich vždy upravíme souřadnice, které tento pohyb změnil. Mohou tedy nastat čtyři případy:

Pokud jde želva na sever (`S`), tak se pohybuje kladným směrem a její souřadnice `y` se tedy zvětší o 1, zatímco `x` se nezmění.



Naopak pokud půjde na jih (`J`), jde opačným směrem a `y` se zmenší o 1.

Ve směru na východ (`V`) se pro změnu zvětší pouze souřadnice `x` o 1.

A pokud želva míří na západ (`Z`), její souřadnice `x` se zmenší o 1.

Program (Python 3):

`http://ksp.mff.cuni.cz/viz/29-Z1-1.py`

Pro tyto účely má velká část programovacích jazyků konstrukci, která se jmenuje `switch` a umožňuje snazší větvení programu podle proměnné, která může nabývat několika různých hodnot.

Program (C):

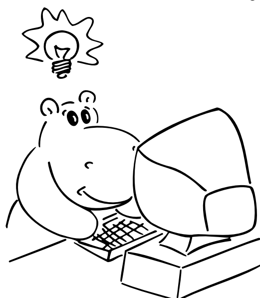
`http://ksp.mff.cuni.cz/viz/29-Z1-1.c`

Také můžeme využít slovníku, který uchovává změnu souřadnic pro každý možný směr. To je dobrý přístup, zvláště kdybychom měli ještě více směrů, kterými se želva může vydat.

Program (Python 3):

`http://ksp.mff.cuni.cz/viz/29-Z1-1-dict.py`

Jirka Sejkora & Míša Štolová



29-Z1-2 Sářiny pamlsky

Naším úkolem bylo vypsat čísla z intervalu, přičemž násobky tří jsme měli nahradit křížkem, násobky pěti kolečkem a násobky obou vynechat.

Základní myšlenka je jednoduchá: Projdeme v cyklu všechna čísla z intervalu a u každého zkontrolujeme, jestli je násobkem 3 nebo 5, a podle toho s ním naložíme. Víc nebylo třeba vymýšlet, zato bylo potřeba tuhle myšlenku naprogramovat. Co nás tedy může zaskočit?

Hranice intervalu – první číslo se bere včetně, druhé bez. To ovšem můžeme jednoduše vykoukat ze zadání. Takové pojetí intervalů je navíc v programování obvyklé, mimo jiné proto, že se pak průchod přímočaře zapisuje (např. v Pythonu) jako `for i in range(od, do)`.

Testování dělitelnosti. Zde se určitě hodí znát operátor modulo, který je ve většině programovacích jazyků a obvykle se zapisuje procentem, `%`. Modulo nám vrací zbytek po celočíselném dělení. Jestli je číslo `x` dělitelné třeba 3, tak jednoduše otestujeme jako `if x % 3 == 0`.

Dělitelnost obojím. S čísly, která jsou násobky jak 3, tak 5, musíme být opatrní. Asi nejjednodušší přístup je na začátku zkontrolovat, jestli je číslo dělitelné obojím, a pokud ano, přeskočit ho (v Pythonu buď použitím `continue`, nebo umístěním zbytku těla cyklu do `else` větve).

Pokud předpokládáme, že modulo funguje v konstatním čase, pro každé číslo z intervalu provedeme maximálně konstatní počet kontrol a maximálně jedno vypsání, tedy časová složitost je $\mathcal{O}(N)$, kde N je velikost intervalu.

Čísla, resp. nahrazující znaky, můžeme rovnou vypisovat, paměťová složitost je tak konstantní, $\mathcal{O}(1)$.

Pro zajímavost ještě dodejme, že naše pohádka skrývá jinak velmi známou úlohu, obvykle označovanou jako Fizz Buzz. Občas ji někdo používá při vstupním pohovoru na programovací profese.

Program (C):

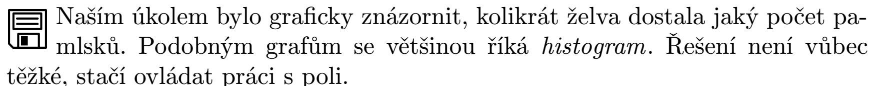
`http://ksp.mff.cuni.cz/viz/29-Z1-2.c`

Program (Python):

`http://ksp.mff.cuni.cz/viz/29-Z1-2.py`

Karry Burešová

29-Z1-3 Petrova statistika



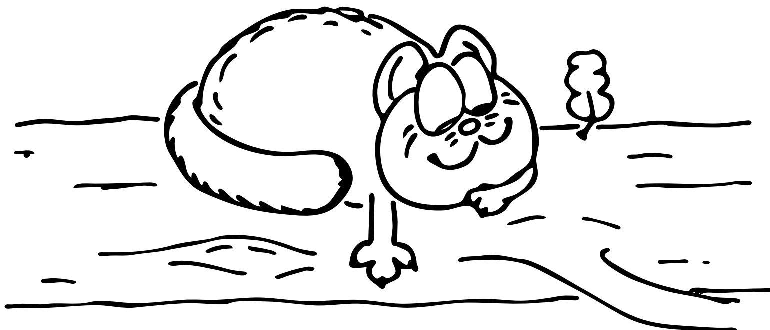
Naším úkolem bylo graficky znázornit, kolikrát želva dostala jaký počet pamlsků. Podobným grafům se většinou říká *histogram*. Řešení není vůbec těžké, stačí ovládat práci s poli.

Pořídíme si jedno velké pole, říkejme mu třeba `četnost`. V poli na indexu i budeme uchovávat počet dní, ve kterých dostala želva i pamlsků. Pokud to váš jazyk neudělá sám, je potřeba ho na začátku naplnit nulami.

Jak bude pole velké? Ze zadání víme, že počet pamlsků nepřesáhne 1 000 000. Jeden milion čísel se nám do paměti vejde i pokud zkoušíme řešit KSP na telefonu.

Pak už jen projdeme postupně počty pamlsků a za každé i zvýšíme hodnotu `četnost[`i`]` o jedničku. Tím máme připravena data, která chceme vypsat.

Zadání po nás ovšem chtělo, abychom ukázali jen relevantní část histogramu. Proto si ještě spočítáme minimum a maximum ze všech i, která jsme potkali. To můžeme dělat průběžně, nebo si čísla načíst všechna a projít je ještě dvakrát.



Pokud by se nám nelíbilo, že si vytváříme zbytečně veliké pole, můžeme počítání minima a maxima provést ještě před počítáním četností. Potom už nám nic nebrání pořídit si pole jen pro hodnoty mezi extrémy.

Lepším řešením je však použít slovník. Co to je, se můžete dočíst v našich kuchařkách, ale například v Pythonu jej použijete velice snadno. S trochou odlehčení stačí nahradit hranaté závorky složenými, případně použít kolekci `defaultdict`, jako to dělá ukázkový zdrojový kód.

Program (Python 3):

`http://ksp.mff.cuni.cz/viz/29-Z1-3.py`

Program (C):

`http://ksp.mff.cuni.cz/viz/29-Z1-3.c`

Ondra Hlavatý

29-Z1-4 Zuzčin výlet

Zuzka chce vymyslet, jaká posloupnost kterých skluzavek je třeba k tomu, aby jich zkusili za jednu cestu dolů co nejvíce. K takovému zkoumání je dobré použít nějaký systém, aby na žádnou možnost nezapomněla, ale také aby jí takové hledání netrvalo zbytečně dlouho.

Zuzku napadlo, že bude postupovat zprava doleva – to znamená, že se podívá v mapce na skluzavku, která vede z horní plošiny co nejvíce vpravo. Poté vybere další, pro kterou platí dvě podmínky: aby na tu předchozí navazovala a aby vedla zase co nejvíce vpravo. Takhle postupuje dál a dál, dokud nebude žádná další skluzavka nebo tobogán, který by navazoval.

Teď si Zuzka poznamená, kolik různých skluzavek by cestou dolů vyzkoušela a podívá se po jiné variantě.

Pro zachování systému se vrátí o jednu skluzavku nahoru, odečte jedna od poznamenaného čísla a bude pokračovat zase dolů – ale jinudy. Nyní zvolí možnost, která nebude ta nejpravější, ale o jednu více vlevo, a opět postupuje směrem dolů a vpravo. Když už nebude moci nikam dál, zase si poznamená, kolik celkem skluzavek by vyzkoušela v této variantě od horní plošiny až sem dolů.

Občas se ale bude muset vrátit dokonce více než o jednu skluzavku, aby skutečně vyzkoušela všechny možnosti odshora dolů.

Takhle Zuzka postupuje, až dokud neprojde variantu, která je úplně na opačné straně – co nejvíce vlevo. Teď má poznamenáno několik údajů, které říkají, kolik nejvíce skluzavek a togobánů lze vyzkoušet v různých variantách. Nakonec z nich stačí vybrat největší číslo. Takovému postupu procházení skluzavek se říká procházení grafu do hloubky a více si o něm můžete přečíst v našich programátorských kuchařkách.[4]

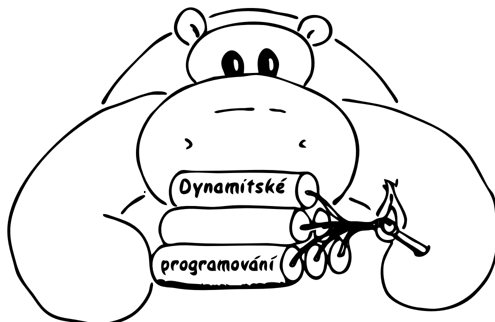
Na závěr ještě jedno vylepšení předchozího programu. Aby si Zuzka nemusela poznamenávat zbytečně mnoho čísel, stačí, když si vždy bude pamatovat jen dosavadní maximum a když najde cestu s více skluzavkami, zapamatuje si to.

Program (Python 3):

`http://ksp.mff.cuni.cz/viz/29-Z1-4.py`

Zuzka Drázdová

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



29-Z1-5 Dva seznamy

Kevin a Sára si po chvilce rozmýšlení všimli, že vždy když je zajímá, jestli je daný kamarád na některém seznamu, musí ho celý přečíst. Sáru proto napadlo, že by bylo nejlepší si seznamy seřadit.

Pro jednoduchost si můžeme představit, že kamarádi nemají jména ale čísla a budeme třídit ta. Pokud bychom pracovali se jmény, musíme si třídící algoritmus upravit pro práci s řetězci. Jeho složitost by pak byla přenásobená délkou nejdelšího řetězce, což jsme ovšem v zadání omezili na konstantu 42, takže nás nezajímá.

Třídit je možné kupou různých algoritmů, o některých pojednává naše kuchařka o třídění.[5] O té slyšel už i Kevin a po prostudování textu se rozhodl pro MergeSort.

Když už máme seznamy setříděné, bude nejlepší postupovat tak, jak radí intuice: Kevin i Sára si budou prstem ukazovat do svého seznamu, aby věděli, kde jsou (trochu podobně jako při slévání v MergeSortu, který už si nacvičili), a porovnají, zda oba ukazují na jméno stejného kamaráda. Pokud ano, jméno si poznamenají. Pokud ne, tak ten, u kterého je jméno pod prstem v abecedě dřív, posune prst dál.

Teď je vhodný okamžik zhodnotit, zda vše funguje tak, jak si naše dvojka přeje. Nejprve trochu formalizujme úlohu. Na vstupu dostáváme dvě množiny a vrátit máme jejich průnik, tedy právě ty prvky, které se vyskytují v obou množinách.

Začněme tím, že nahlédneme, že jsme žádný takový prvek nevynechali: postup s ukazováním prsty způsobuje (stejně jako v MergeSortu), že procházíme prvky obou množin dohromady v setříděném pořadí. Tedy pokud se prvek vyskytoval v obou množinách, neodejdeme z něj v jednom seznamu dokud na něj

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

nenarazíme ve druhém. Posuneme se až tehdy, když je ten druhý prvek v abecedě za námi. Takže každý prvek patřící do průniku najdeme.

Také se nám nemůže stát, že jsme do průniku zahrnuli prvek, který tam nepatří – přidáme ho jedině tehdy, když byl v obou seznamech, a tedy tam patří.

Ještě nám zbývá zjistit, jak je jejich řešení časově a paměťově náročné, označme si jako N počet dětí v delším seznamu. Na setřídění seznamů bylo potřeba $\mathcal{O}(N \cdot \log N)$ času a $\mathcal{O}(N)$ paměti. V druhé fázi už Kevin se Sárou seznamy jen prošli v čase $\mathcal{O}(N)$, a potřebují místo, kam si poznamenat nadšené kamarády – $\mathcal{O}(N)$.

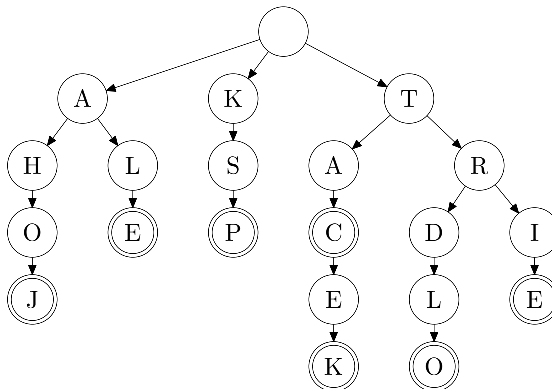
Možná by vás mohlo zarazit, že tu nemluvíme o složitostech týkajících se kratšího seznamu. Jelikož $\mathcal{O}$-čková notace vyjadřuje složitost v nejhorším případě a se seznamy provádíme stejné operace, stačí nám uvažovat složitost právě pouze pro ten delší.

Za takovéto řešení získáte plný počet bodů.

Program (Python 3):

`http://ksp.mff.cuni.cz/viz/29-Z1-5.py`

Nicméně jde to i rychleji, např. když použijeme datovou strukturu zvanou *trie*.[6] Trie je strom, v jehož každém vrcholu je jeden znak. Slova jsou reprezentována znaky na cestě z kořene. Trii pro slova `AHOJ`, `ALE`, `KSP`, `TRIE`, `TRDLO`, `TAC` a `TACEK` si můžete prohlédnout na obrázku. Dvojitým kolečkem značíme, že daným znakem končí slovo.



Jak to tedy celé uděláme? Nejprve si postavíme trii z jednoho ze seznamů, nezáleží ze kterého. Začínáme s prázdným stromem a postupně do něj vkládáme jednotlivá jména.

---

[6] `http://ksp.mff.cuni.cz/viz/kucharky/hledani-v-textu`

Teď chceme zjistit, která jména z druhého seznamu byla i v tom prvním. Jednoduše postupně zkusíme každé vyhledat v trii a pokud kamaráda najdeme, chce hrát obě hry a poznamenáme si jej.

Nezapomeňte kontrolovat, že jsme opravdu skončili v označeném vrcholu, nechceme mít v seznamu jméno Petr, když mezi kamarády máme pouze Petru. Pokud jméno v trii nenalezneme, kamaráda si nepoznamenáme.

Kolik nám to zabralo času? Označme si A délku seznamu, ze kterého stavíme trii, B délku druhého seznamu a jako d délku (libovolného) jména.

Vytvoření trie trvalo nejdéle $\mathcal{O}(A \cdot d)$, protože jsme pro každé jméno a každé jeho písmenko museli přidat/projít jeden vrchol. Vyhledání jednoho jména trvá nejvýše $\mathcal{O}(d)$, postupně porovnáváme každý znak, a vyhledáváme B jmen, dohromady tedy $\mathcal{O}(B \cdot d)$.

Ovšem my ze zadání víme, že d je nejvýše 42, je to tedy (pro tuto úlohu malá) konstanta a můžeme ji zanedbat. Výsledná časová složitost tak je $\mathcal{O}(N)$, kde N je maximum z A a B.

Ve skutečnosi je časová složitost lineární s celkovou velikostí vstupu (součet písmen ve slovech) i bez omezení na délku jmen. To se projeví např. kdybychom měli spoustu krátkých slov a jedno hodně dlouhé, kdy bude přibližně rovna součtu délky seznamu a délky dlouhého slova.

Paměťová složitost je $\mathcal{O}(A \cdot d)$, druhý seznam dokonce ani nepotřebujeme načítat do paměti celý.

Program (Python 3):

`http://ksp.mff.cuni.cz/viz/29-Z1-5-trie.py`

Zuzka Šimečková & Katka Zákravská

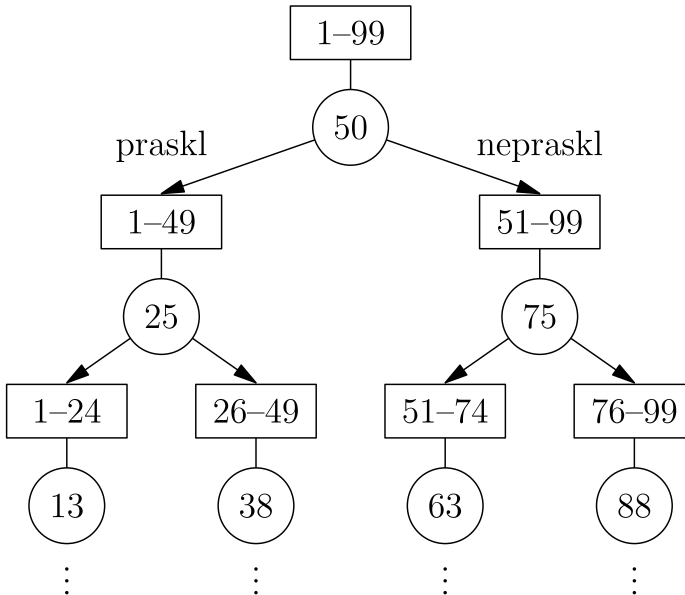
29-Z1-6 Devadesát devět pater

Pokaždé, když Petr shodí balónek z n-tého patra, mohou nastat dvě situace: Pokud se balónek nerozbije, je jasné, že musí házet z větší výšky, a tedy už stačí prozkoušet patra $n+1$, $n+2$, …, 99. Pokud se ovšem balónek rozbije, přicházejí v úvahu již pouze patra 1, 2, …, n. Při hodu z větší vzdálenosti se balónek určitě taky rozbije, ale z větší vzdálenosti by Petr měl zbytečně menší pravděpodobnost, že protivníka trefí.

Protože se snažíme dosáhnout co nejmenšího počtu pokusů i v nejhorším případě, chceme si být jistí, že se každým pokusem zbavíme co největšího počtu pater. Máme tedy úsek pater, které ještě přichází v úvahu, a shodíme balonek z prostředního patra tohoto úseku. Přesněji řečeno, shodíme balonek z patra $n/2$, pro liché n zaokrouhlíme (můžete si ověřit, že je jedno na kterou stranu).

Všimneme si, že tím se každým pokusem poloviny pater zbavíme, až nám nakonec zbyde pouze jedno jediné. Pro našich 99 pater tedy nejdřív shodíme

z 50. patra. Pokud praskne, shodíme z 25., pokud nepraskne, ze 75. patra. V obou případech nám po těchto dvou pokusech zbývá prověřit už jen 25 pater.



Počet pater, které přichází v úvahu, se bude postupně zmenšovat z 99 na 50, 25, 13, 7, 4, 2 a 1 patro. Po nejvýše sedmi pokusech tedy vždycky zjistíme ideální vzdálenost.

Lehčí varianta pro 9 pater funguje identicky, akorát nám na zjištění stačí čtyři pokusy (5, 3, 2, 1).

Pro obecné P určitě umíme postupovat stejně, jak spočítáme celkový počet kroků? Víme, že počet možných pater se bude postupně zmenšovat z P na $\lceil \frac{P}{2} \rceil$, $\lceil \frac{P}{4} \rceil$, $\lceil \frac{P}{8} \rceil, \ldots, 1$. Potřebujeme tedy odpovědět na otázku, kolikrát můžeme P vydělit dvěma, abychom došli až k jedničce. Přesně na tuto otázku odpovídá funkce logaritmus, a řekneme, že celkem provedeme $\lceil \log_2 P \rceil$ pokusů.

Tento algoritmus se nazývá binární vyhledávání nebo také metoda půlení intervalů. Více se o něm můžete dočíst v naší kuchařce.[7]

Jan Gocník & Dominik Smrž

[7] `http://ksp.mff.cuni.cz/viz/kucharky/vyhledavaci-stromy`

Druhá série

29-Z2-1 Krocení zlé želvy

Naším úkolem bylo ze zadané posloupnosti příkazů zjistit, kde skončí Kevinova želva, až všechny tyto příkazy vykoná. Jako první je dobré si uvědomit, že úlohu můžeme řešit bez toho, abychom si vytvářeli velké pole a pohyb želvy po něm simulovali.

Tedy, abychom použili analogovou analogii, není potřeba vzít do ruky (velký) čtverečkovaný papír a postupně si kreslit kterými políčky želva projde. Namísto toho si stačí pamatovat vždy jen aktuální pozici a směr želvy.

Jakmile totiž zjistíme, jak ji posouvat, můžeme namísto simulace na velkém poli či po čtverečkovém papíře, prostě jen aplikovat změnu na zapamatovanou pozici či směr. V případě pokynu A konkrétně změnit pozici a při pokynech `<` a `>` otočení.

Pozice se skládá ze dvou souřadnic. Můžeme si představit třeba čtverečkovaný papír se čtverečkem $(0, 0)$ někde uprostřed.

Směr je ještě jednodušší, stačí si pamatovat, zda jsme otočení na sever, západ, jih nebo východ. Prakticky tedy postačí libovolná číselná proměnná. Na začátku je želva otočená nahoru.

Pro pořádky indexace si směry otočení seřadíme jak jsou napsané výše. Otočení s indexem 0 (přeci jen většina programovacích jazyků indexuje od 0, tak buďme konzistentní) bude na sever, s indexem 1 na západ a dále logicky proti směru hodinových ručiček.

Jakmile si vytvoříme takovouto reprezentaci, můžeme si všimnout, že otočení doleva je vždy zvýšení čísla otočení a doprava snížení. Problém jsou jen krajní hodnoty – když máme směr s indexem 3 (tj. na východ) a otočíme se doleva, dostaneme se na otočení na sever, tedy index 0.

Naštěstí ale nemusíme tyto krajní případy řešit separátně. Stačí místo upraveného indexu otočení vzít jeho zbytek po dělení čtyřmi, tj. např. $((i+1) \bmod 4)$, a dostaneme přesně to, co chceme.

Otáčení máme vyřešeno. Stačí už jen vymyslet, jak aktuální směr aplikovat v případě příkazu `A`, tedy posunu želvy dopředu. Nejjednodušší řešení je připravit si dvě pole o čtyřech prvcích. Jedno pro změnu souřadnice v x-ové a druhé v y-ové ose pro každý možný směr otočení.

Pro osu x může dané pole vypadat třeba takto: $[0, -1, 0, 1]$. Při otočení nahoru se při pohybu vpřed naše x-ová souřadnice nezmění. Při otočení doleva se zmenší o jedna a obdobně dál.

Při provádění příkazu `A` se pak jen podíváme do obou polí na hodnotu indexovanou otočením a danou hodnotu přičteme k odpovídající souřadnici aktuální pozice.

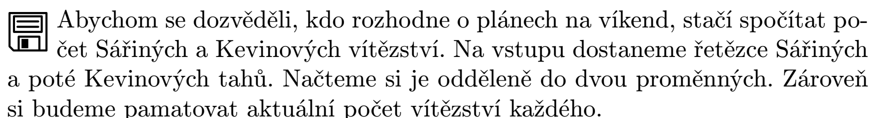
Časová složitost algoritmu je lineární s počtem příkazů, na každém totiž strávíme konstantně času. Paměťová je konstantní, pokud nepočítáme, že si je třeba pamatovat vstup. V průběhu algoritmu si totiž pamatujeme nezávisle na velikosti vstupu jen dvě pomocné proměnné.

Program (Python 3):

`http://ksp.mff.cuni.cz/viz/29-Z2-1.py`

Petr Houška

29-Z2-2 Sářina volba



Abychom se dozvěděli, kdo rozhodne o plánech na víkend, stačí spočítat počet Sářiných a Kevinových vítězství. Na vstupu dostaneme řetězce Sářiných a poté Kevinových tahů. Načteme si je odděleně do dvou proměnných. Zároveň si budeme pamatovat aktuální počet vítězství každého.

Pak již jen stačí procházet jednotlivé hry. Nejdříve vyhodnotíme první hru (tj. na indexu 0 v obou řetězcích), pak hru na indexu 1 až k poslední $n - 1$. K tomu nám poslouží například cyklus `for`, v jehož každém průchodu vítězi přičteme jedno vítězství.

Při procházení potřebujeme rozhodnout, kdo danou hru vyhrál. Uvědomíme si, že hra může skončit pouze devíti stavy (každý má tři možnosti jak zahrát), které můžeme snadno otestovat podmínkami.

Pokud oba zahráli stejně, nevyhrál nikdo a přeskočíme rovnou na další hru.

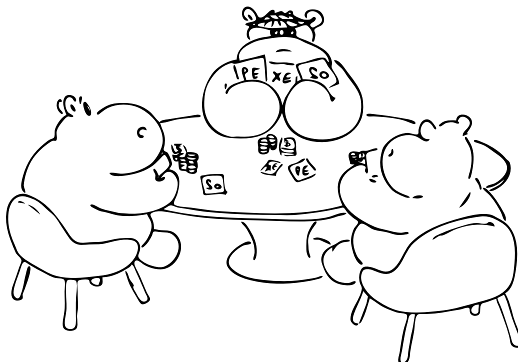
Pokud Sára hrála kámen a Kevin nůžky, pak vyhrála Sára a připočteme ji jedno vítězství. Stejně tak v případě Sára nůžky + Kevin papír a Sára papír + Kevin kámen.

Pokud nenastala ani jedna z předchozích kombinací, pak nutně vyhrál Kevin, a proto mu připočteme jedno vítězství.

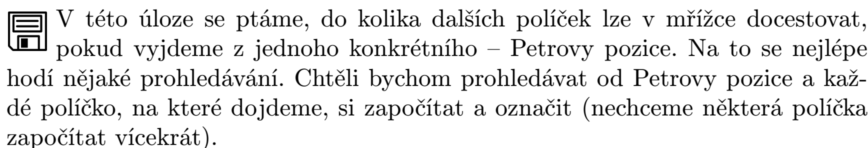
Program (Python 3):

`http://ksp.mff.cuni.cz/viz/29-Z2-2.py`

Jan Knížek



29-Z2-3 Petr v říši divů



V této úloze se ptáme, do kolika dalších políček lze v mřížce docestovat, pokud vyjdeme z jednoho konkrétního – Petrovy pozice. Na to se nejlépe hodí nějaké prohledávání. Chtěli bychom prohledávat od Petrovy pozice a každé políčko, na které dojdeme, si započítat a označit (nechceme některá políčka započítat vícekrát).

Prohledávat budeme následovně: vytvoříme si frontu, ve které budeme skladovat všechna políčka, která ještě chceme projít. Na začátku do ní přidáme jen políčko, na kterém stojí Petr. Pokaždé z fronty vytáhneme prvního kandidáta a podíváme se na jeho sousední políčka. Do fronty potom přidáme všechna políčka, která jsme ještě nenavštívili a nejsou to zdi.

Sousední políčka poznáme tak, že se jejich souřadnice liší o ± 1 od aktuálního políčka (například políčko napravo má souřadnici x větší o 1, a y stejnou). Každé políčko, které jsme do fronty přidali, započteme, označíme a potom vytáhneme další.

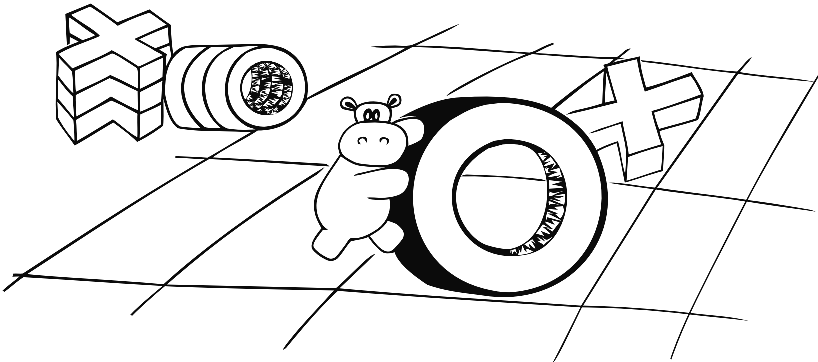
Jakmile bude fronta prázdná, tak jsme dokončili procházení z Petrovy pozice a nikam jinam se už nedostaneme. Zbývá tedy vypsat číslo, které jsme napočítali a skončit.

Poslední malý problém k vyřešení je, jak najít pozici, na které se nachází Petr a Sára. Protože mohou být kdekoliv v mapě, nepomůže nám žádný chytrý algoritmus a musíme zkontrolovat všechna políčka. Budeme procházet od začátku jedno po druhém, dokud nenarazíme na znak `P` a jeho pozici si zapamatujeme.

Zbývá si rozmyslet, zda se nemůže stát, že v lese existuje políčko, na které by se Petr a Sára mohli dostat, ale my jsme ho nezapočítali. To se nestalo, protože do fronty se během prohledávání dostalo každé dostupné políčko. Naopak, nezapočetli jsme náhodou nějaká navíc? Protože jsme přičítali pouze volná políčka, na která jsme se mohli dostat, nezapočetli jsme nic navíc.

Jak je to celé rychlé a kolik to žere paměti v počítači? Nalezení Petra se Sárou trvá $\mathcal{O}(R \cdot S)$, prohledávání trvalo také $\mathcal{O}(R \cdot S)$, protože každé políčko se do fronty dostalo maximálně jedenkrát. Celý program je tedy lineární a dokonce trvá pouze $\mathcal{O}(R \cdot S + R \cdot S) = \mathcal{O}(R \cdot S)$

Paměťová složitost je také lineární $\mathcal{O}(R{\cdot}S)$, protože jediné, co si potřebujeme pamatovat je mapa, která má $R{\cdot}S$ políček, a fronta, ve které se nikdy více políček než jich je v mapě neobjeví. Pokud si nejsi jistý v tom, co v tomto odstavci řešíme, můžeš si přečíst kuchařku o složitosti.[8]



Na závěr podotkněme, že program se určitě zastaví, protože hledání Petra skončí po maximálně N krocích a naše prohledávání se zastaví, neboť každé políčko dáme do fronty maximálně jednou.

Jako třešničku na dortu přidáme, že algoritmus, který jsme použili při řešení je ve světě informatiky tak rozšířený, že má své vlastní jméno. Jmenuje se BFS – prohledávání do šířky a dá se použít na různé grafové problémy. Pokud nevíš, co graf je, mohla by tě zajímat naše kuchařka o grafech.[9] Pokud už grafy znáš, tak si můžeš rozmyslet, že čtverečková síť, jako naše mapa ze zadání, je vlastně graf. Stačí každé políčko prohlásit za vrchol a spojit ho čtyřmi hranami s jeho sousedy.

Program (Python 3):

`http://ksp.mff.cuni.cz/viz/29-Z2-3.py`

Štěpán Hojdar

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

29-Z2-4 Zuzka: Cesta tam a zase zpátky

Zadání po nás chtělo, abychom Zuzce nalezli nejdelší úsek, ve kterém půjdeme nejvýše K sekund do kopce. Nejprve vyzkoušíme jednoduché řešení.

Pro každé místo v posloupnosti vyzkoušíme, jestli náhodou ta nejdelší podposloupnost nezačíná zrovna tam. Tedy si naprogramujeme funkci, které řekneme začátek, a ona od něj najde nejdelší možný úsek, ve kterém půjde maximálně K sekund do kopce.

To je přeci jednoduché – pořídíme si energetickou kasičku, do které vložíme K penízků, a postavíme Kevina na začátek. Poté necháme Kevina jít co nejdále to půjde. Z kopce a po rovině půjde zadarmo, ale do kopce bude chtít penízek z kasičky. Jakmile bude chtít Kevin penízek, ale kasička bude prázdná, skončíme. Stejně tak pokud dojde Kevin na úplný konec trasy.

Z takto nalezených posloupností už jen snadno vybereme tu nejdelší. Jak to celé bude rychlé? Představte si dlouhou cestu, která bude ovšem celá z kopce. Naše funkce tedy pokaždé dojde až na konec cesty a celkem vykoná $N + (N - 1) + \ldots + 1$ kroků, což je ovšem součet aritmetické posloupnosti, který vyjde v $\mathcal{O}(N^2)$.

Chudák Kevin musí totiž pořád chodit tu samou trasu znovu a znovu. Proveďme jednoduché pozorování: pokud posuneme začátek doprava, Kevin může dojít jen dál – konec se nemůže posunout vlevo. Kevin se tedy nemusí vůbec vracet!

Na začátku algoritmu postavíme jak Zuzku tak Kevina na začátek trasy. Kevina pošleme kupředu, aby došel co nejdále, ale maximálně K sekund šel do kopce. Od této chvíle bude platit následující *invariant* (tvrzení, jehož platnost se v průběhu algoritmu nemění): mezi Kevinem a Zuzkou bude právě K sekund chůze do kopce.

Teď přijde řada na Zuzka. Ta půjde dopředu, ale pouze z kopce či po rovině. Kdykoli by měla jít do kopce, zavolá Kevinovi, a sekundu do kopce půjdou

společně. Poté si Zuzka může odpočinout a Kevin utéct Zuzce co nejdále z kopce.

Zajímá nás chvíle, kdy budou Kevin se Zuzkou nejdále od sebe. To je totiž díky invariantu chvíle, kdy Kevin se Zuzkou vymezují nejdelší podposloupnost podle zadání.

Jakmile Kevin dojde na konec trasy, můžeme rovnou skončit, protože Zuzka už se bude jen přibližovat.

Teď už víme, že algoritmem vydané řešení bude vždy správné, ale ještě nám zbývá ukázat, že pokud řešení existuje, algoritmus ho najde. To je naštěstí jednoduché, protože každé řešení musí někde začínat a přes to místo musí Zuzka někdy přejít.

Algoritmicky samozřejmě nebudeme posílat Zuzce a Kevinovi itinerář cesty, ale budeme si posouvat dva ukazatele nad polem. Když si uvědomíte, že oba ukazatele posouváte pouze vpravo, vyjde z toho optimální časová složitost $\mathcal{O}(N)$.

Paměťovou složitost máme ovšem zatím také lineární. Pokud bychom chtěli ušetřit, musíme se vydat ještě o krůček dál. Než si ale přečtete další odstavce, načíst celý soubor se vstupem do paměti k řešení většinou bohatě stačí. Následující informace tedy berte jako teoretický bonus.

Všimněte si, že v celém algoritmu nás vlastně vůbec nezajímaly konkrétní hodnoty nadmořských výšek. Místo nich nám stačí uvažovat, kdy cesta vedla do kopce a kdy z kopce. Dokonce ani nepotřebujeme mít jednotlivé sekundy v paměti rozdělené – stačí nám koukat na to, jak dlouhé jsou úseky z kopce mezi jednotlivými sekundami do kopce.

Mohli bychom tedy algoritmus pozměnit tak, že by načítal jednotlivá čísla s tím, jak přesouvá Kevina, a pro Zuzku už by si pamatoval jen délku dalších K úseků z kopce. Tím bychom srazili paměťovou složitost na $\mathcal{O}(K)$.

Program (Python 3):

`http://ksp.mff.cuni.cz/viz/29-Z2-4.py`

Ondra Hlavatý

29-Z2-5 Dva roky bez prázdnin

Od Šklíby jsme dostali úkol najít původce řetězu putování spamu mezi bytostmi. Nejprve si můžeme rozmyslet, že řetěz rozesílání e-mailu nám tvoří nějaký orientovaný graf. Seznam dvojic, jenž Šklíba získal, není nic jiného, než seznam hran, avšak neorientovaných, v tomto grafu. Vrcholy potom reprezentují jednotlivé bytosti, které se dostaly do styku se spamem.

Dále víme, že každé bytosti spam přišel nejvýše jednou a možná jej odeslala dalším K bytostem. V řeči grafů to znamená, že nejvýše jedna hrana do vrcholu (bytost) vstupuje a vystupuje z něj buď K hran, nebo žádná.

Vzorová řešení KSP-Z – 2. série

Stupněm vrcholu budeme značit počet hran, které vedou do nebo z daného vrcholu. Podívejme se tedy na možnosti, jaký může být stupeň vrcholu bytosti:

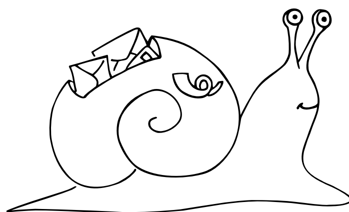
- Pokud bytosti spam přišel, do jejího vrcholu vede právě jedna hrana, stupeň se tedy zvýší o 1.
- Pokud bytost spam odeslala dalším K bytostem, stupeň jejího vrcholu se zvýší o K.

Pojďme z toho prozkoumat, jakého tvaru tento graf nabývá. Již víme, že každý vrchol má nejvýše jednoho předchůdce. Také má každý vrchol právě 0 nebo K potomků. Takový graf nápadně připomíná strom.

Kdybychom se nezabývali orientací hran, opravdu o strom jde. Kořenem je v tomto případě původce spamu a listy jsou doručitelé, kteří dále spam nerozeslali. Dále si můžeme všimnout, že je tento strom k-ární, jelikož každý vrchol, jenž není list, má právě k potomků.

Stupně vrcholů v našem grafu tedy můžou být 1 pro „listy“, K pro „kořen“, nebo $K+1$ pro ostatní. Z toho můžeme usoudit, že chceme-li najít původce spamu, stačí nám najít kořen v pomyslném stromě, který má stupeň právě K. Žádný jiný vrchol stejný stupeň už mít nebude, původce je jen jeden.

Ten můžeme najít takto: Postupně projdeme jednotlivé dvojice a budeme si poznamenávat pro každou bytost, kolikrát se vyskytla v nějaké z dvojic. Všimněme si, že tento počet odpovídá právě stupni jejího vrcholu. Potom stačí vyhlásit bytost s právě K výskyty jako původce spamu.



Jak dlouho nám hledání původce potrvá a kolik při tom spotřebujeme paměti? Každou z M dvojic musíme projít právě jednou. Celkově nad tímto procházením strávíme $\mathcal{O}(M)$ času. Potom projdeme každou z N bytostí a zkoumáme počet výskytů. Časová složitost je tedy $\mathcal{O}(N+M)$. Dále si musíme pro každou bytost něco málo pamatovat, paměti tak spotřebujeme $\mathcal{O}(N)$.

Kolik dvojic ale může být celkem? Jelikož každému, až na jediného původce, přišel spam právě jednou, bude těchto dvojic právě $N-1$. Časová složitost nám tedy ve skutečnosti sejde na $\mathcal{O}(N)$.

Václav Končický

29-Z2-6 Devět trpaslíků

Kevin a Zuzka dostanou řadu N čísel v určitém pořadí a u ní mají za úkol rychle odpovídat na určité dotazy. Ty se týkají součtů čísel mezi a-tým a b-tým číslem řady.

Zjevné řešení je pro každý dotaz posloupnost znovu projít a od a-tého do b-tého indexu čísla sčítat. Pokud by se trpaslíci ptali pořád na součet celé řady, procházeli bychom vždy všech N čísel znovu. Kvůli tomu je časová složitost na jeden dotaz $\mathcal{O}(N)$.

Zkusíme zvolit úplně jiný postup. Budeme na něj sice potřebovat více času na přípravu, ale budeme doufat, že se nám to vyplatí. Naším cílem je odpovídat co nejrychleji. Využijeme toho, že sčítání má dobré vlastnosti (například pro násobení by naše řešení nefungovalo), a předpočítáme si čísla, ze kterých budeme schopni rychle vykoukat řešení.

V dalších odstavcích budeme součet čísel mezi a-tým a b-tým indexem značit $s(a, b+1)$. Následujeme tedy běžné programátorské pojetí intervalů, kdy $s(a, b)$ značí součet takového úseku, do kterého a patří ale b už ne.

Představme si, že známe součty na dvou úsecích, které mají společný začátek: $s(a, b)$ a $s(a, c)$. Z těchto čísel jsme schopni odvodit $s(b, c)$ – ten je roven $s(a, c) - s(a, b)$. Rozepište si to. Uvidíte, že se čísla na začátku odečtou a zůstanou jen ta správná.

Tohoto faktu využijeme a spočítáme si součet $s(0, x)$ pro každé x. Úsek začínající na začátku posloupnosti se nazývá prefix posloupnosti, a proto se posloupnosti jejich součtů obvykle říká prefixové součty.

Uvedeme příklad. Máme řadu čísel $1, 5, 4, 3, 6$ a chceme k ní znát všechny prefixové součty. Ty budou vypadat následovně: $0, 1, 6, 10, 13, 19$. Spočítat všechny prefixové součty můžeme průběžně při načítání – každé číslo stačí přičíst k předchozímu, už sečtenému, prefixu, a máme součet o jedna delšího prefixu. Díky tomu nám počítání zabere pouze $\mathcal{O}(N)$.

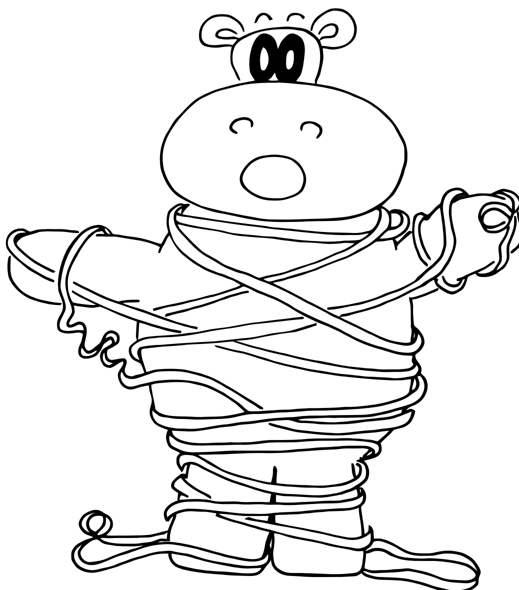
```
součty[0] = 0
for i in range(N+1):
    součty[i] = součty[i - 1] + čísla[i - 1]
```

Abychom mohli počítat s celou řadou na vstupu, přidáme si na konec součtů ještě jeden prvek navíc, součet celé řady (proto N+1). Všimněte si ale, že čas na přípravu nás vlastně moc nezdržuje – pokud nechceme čísla číst rovnou ze souboru, nevyhneme se načtení do paměti. To už ale trvá stejně dlouho, jako počítání prefixových součtů.

A když už máme pole prefixových součtů připravené, dotazy se zodpovídají velmi snadno. Hodnota součtu čísel mezi a-tým a b-tým je rovna `součty[b + 1] - součty[a]`. Na dotazy tedy umíme odpovídat v konstantním čase, ale potřebujeme $\mathcal{O}(N)$ času na přípravu. Paměťová složitost je $\mathcal{O}(N)$. Pokud se chcete

o prefixových součtech a jejich využití dozvědět více, podívejte se na kapitolu kuchařky o intervalových stromech.[10]

Ondra Hlavatý



[10] `http://ksp.mff.cuni.cz/viz/kucharky/intervalove-stromy`

Třetí série

29-Z3-1 Želva na dvorku

řešení

Úloha byla opět jen drobným rozšířením Krocení zlé želvy z druhé série. Želva se chovala úplně stejně, přibyly pouze překážky. Přesto ale řešení neodbudeme a ukážeme si ještě jeden, trochu jiný, způsob simulace pohybu želvy.

V minulém řešení jsme si pamatovali aktuální souřadnice želvy jako dvě čísla a směr želvy jako číslo třetí. Pak jsme měli připravená pole, která říkala, v jakém směru máme o kolik změnit jednu ze souřadnic.

Tento přístup se hodí, pokud například chceme procházet všech 8 políček, na které můžeme skočit šachovým koněm z aktuálního políčka. Součástí programu bude statické pole obsahující políčka, na které může kůň skočit ze souřadnic `[0, 0]`. Pokud zrovna nestojí na tomto políčku, stačí souřadnice patřičně posunout.

Nám ale stačí pamatovat si aktuální směr želvy, a proto můžeme program na pohled trochu zjednodušit. Nebudeme si směr pamatovat jako nějaké magické číslo, ale jako směrový vektor. Slovní spojení je to děsivé, ale vysvětlení je prosté. Směrový vektor je rozdíl příštího a aktuálního políčka – pokud tedy želva udělá krok. Želva je na začátku otočená na sever, tomu odpovídá vektor $(0, 1)$.

Na povel *krok* zareagujeme snadno, prostě směrový vektor přičteme k aktuální souřadnici. Otáčení je složitější na pochopení, ale o to snazší na napsání.

Pokud je směrový vektor (dx, dy) a želva se otáčí za levou rukou, vezmeme jako nový směrový vektor $(-dy, dx)$. Vyzkoušejte, funguje to. Můžete si zkusit napsat směrové vektory pro všechny čtyři směry a uvidíte, že vždy obsahují jednu nulu a jednu (mínus) jedničku. Prohlédněte si, jak otáčením jednička „cestuje".

Teď ale to hlavní: překážky. Potřebujeme zařídit, aby želva nechodila skrz ně. Představme si na chvíli, že umíme snadno poznat, jestli na daném políčku je překážka. Potom si můžeme před každým krokem spočítat pozici cílového políčka a podívat se, jestli je obsazené překážkou. Pokud ano, započítáme naražení do překážky a želva se nepohne. Pokud ne, změníme souřadnice želvy na ty vypočtené.

Jak ale poznat, jestli je na zvoleném políčku překážka? Pokud si všechny překážky vložíme za sebe jen tak do pole, budeme jej muset před každým krokem celé projít. To by mělo časovou složitost lineární s počtem překážek, $\mathcal{O}(P)$ pro každý krok želvy. Tedy až $\mathcal{O}(P \cdot N)$ pro celý běh programu.

Pokud si ale souřadnice překážek po načtení lexikograficky seřadíme, můžeme v nich hledat binárním vyhledáváním.[11] Tím se poměrně snadno dostaneme

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

na časovou složitost $\mathcal{O}(N \log P + P)$, což je rozhodně o něco lepší. Pokud přemýšlíte proč $+P$, pak protože musíme překážky také načíst, což trvá $\mathcal{O}(P)$ (a P může být větší než $N \log P$, takže se to do prvního členu neschová).

Naprogramovat či popsat binární vyhledávání je dobré cvičení, které bychom po vás chtěli v teoretické domácí úloze. Pokud ale píšete program, můžete využít knihoven vašeho programovacího jazyka. Existuje totiž často používaná datová struktura, která umí prvky rychle přidávat a testovat jejich existenci. Běžně se jí říká množina, *set*.

Zejména v Pythonu je její použití extrémně jednoduché. Opravdu stačí místo hranatých závorek (případně funkce `list`) použít funkci `set`, která vrátí množinu prvků z parametru. Testování na existenci funguje úplně stejně jako s polem pomocí operátoru `in`.

Program (Python 3):

`http://ksp.mff.cuni.cz/viz/29-Z3-1.py`

Ondra Hlavatý

29-Z3-2 Písemka z angličtiny

Uvědomit si přímočaré řešení není vůbec těžké. Stačí si uložit všechna slova do pole, pro každé jednotlivé slovo pole projít a počítat si jeho prefixy. Pak vypíšeme slovo s nejvyšším počtem nalezených prefixů. Celý výpočet proběhne v $\mathcal{O}(N^2)$, kde N je součet délek všech slov.

Zkusíme najít něco lepšího. Pokud jste si vzpomněli na řešení úlohy 29-Z1-5,[12] správně vás napadlo použít *trii*. Pro připomenutí: trie je zakořeněný strom, kde vrcholy reprezentují písmena a každé slovo ve slovníku představuje cestu z kořene do listu. Každý vrchol tedy můžeme chápat jako prefix: musíme si ale označit vrcholy, kde nějaké slovo končí.

Vyrobíme si trii ze všech slov na vstupu. Ke každému vrcholu chceme určit S, počet slov ze slovníku, která jsou jeho prefixem. Proto projdeme trii do hloubky: do zásobníku si kromě vrcholu uložíme i jeho S. Na začátku bude zásobník obsahovat kořen trie a $S = 0$. Cílem je najít slovo s nejvyšším S.

Jeden krok průchodu se bude skládat z vyjmutí vrcholu a jeho S ze zásobníku. Pokud ve vrcholu končí slovo, zvýšíme S o 1. Pak do zásobníku vložíme všechny potomky vrcholu s vypočteným S.

Po celou dobu si ukládáme vrchol s nejvyšším S a jakmile skončíme průchod, vypíšeme slovo, které vrcholu odpovídá. Abychom to mohli udělat, musíme se z vrcholu vrátit až do kořene a po cestě si ukládat písmena.

[12] `http://ksp.mff.cuni.cz/viz/29-Z1-5/reseni`

To, že ve vrcholu s nejvyšším S končí slovo ze slovníku, nemusíme vůbec kontrolovat. Stačí si uvědomit, že to vždy bude list stromu, tedy vrchol bez potomků (a v něm vždy nějaké slovo končí).

Paměťová složitost je jistě $\mathcal{O}(N)$, vrcholů nebude více, než je součet délek slov. Časová složitost však závisí na tom, jakým způsobem budeme ve vrcholech ukládat odkazy na potomky.

Používáme anglickou abecedu s malými znaky, proto je potomků nejvýše 26. Můžeme ve vrcholech vytvořit pole odkazů s touto délkou: k vrcholu sice dokážeme přistoupit v konstantním čase, ale spotřeba paměti nehezky naroste (pro slovník „normálních“ slov bude obvykle obsazena jen malá část pole).

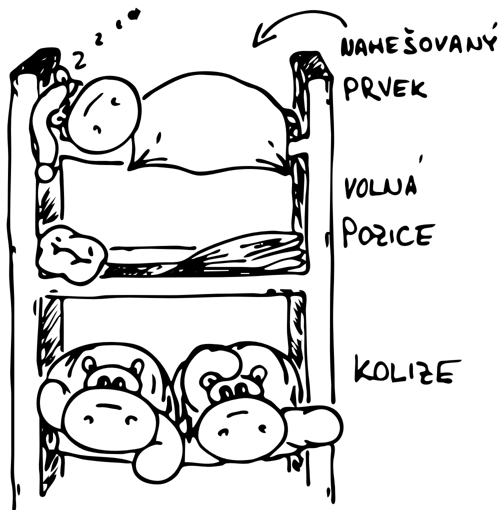
Lepší možností je si ukládat ukazatele do vyváženého binárního stromu, případně využít hešování. U první možnosti bude vytvoření trie i průchod v $\mathcal{O}(N \cdot \log 26) = \mathcal{O}(N)$, u druhé také v $\mathcal{O}(N)$.

Autorský zdrojový kód využívá hešování. V Pythonu se heš tvoří přímočaře, obvykle se ale této struktuře říká *slovník (dictionary)*. Hodí se však vědět, jak funguje uvnitř, což popisujeme v naší hešové kuchařce.[13]

Program (Python 3):

`http://ksp.mff.cuni.cz/viz/29-Z3-2.py`

Kuba Maroušek



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

29-Z3-3 Šestková čísla

Řešení úlohy si rozdělíme na dvě části. Nejdříve šestkový zápis převedeme na číselnou hodnotu a poté pro výsledné číslo vypíšeme jeho hlavní zápis.

Pokud by šestkové číslice byly pouze v sestupném pořadí, stačilo by projít vstup a sečíst jejich hodnoty. Může se ale stát, že skupinu stejných šestkových číslic následuje číslice vyšší hodnoty a my musíme předchozí odečíst.

Budeme si proto držet dvě pomocné proměnné h a p. První h bude označovat hodnotu poslední číslice, kterou jsme viděli, druhá p pak kolik těchto číslic jsme viděli nepřerušovaně za sebou.

Vstup projdeme číslici po číslici a budeme průběžně aktualizovat pomocné proměnné i tu, která drží celkový výsledek.

Pokud je aktuálně zpracovávaná číslice hodnoty h, stačí p zvýšit o jedna.

Pokud je nižší hodnoty, k celkovému výsledku přičteme hodnoty číslic reprezentovaných v pomocných proměnných. Poté nastavíme p na jedničku a h na hodnotu aktuální.

Pokud je hodnota aktuální číslice vyšší, přičteme k výsledku hodnotu aktuální číslice a odečteme hodnotu těch, které jsou reprezentovány pomocnými proměnnými. Víme, že v tomto případě následující číslice určitě bude menší než aktuální, proto stačí nastavit pomocné proměnné p na nula a h na hodnotu aktuální číslice. Díky tomu se při příštím průchodu pomocné proměnné nastaví na nové hodnoty.

V druhé části máme číslo (pojmenujme jej c), které chceme vypsat v hlavním šestkovém zápisu. Představme si, že nemůžeme odečítat. Pak stačí procházet číslice od největší po nejmenší. V každém průchodu pak, dokud je c větší než aktuální číslice, odčítáme od c tuto číslici a zároveň ji vypisujeme na výstup.

My ale můžeme i odečítat, což potřebujeme, abychom se vyhnuli zakázaným čtyřem stejným číslicím za sebou. Označme si hodnotu aktuální číslice jako a. Pokud je $c \geq a$ zachováme se stejně jako v předchozím odstavci.

Dále si všimněme, že nejnižší hodnotou, kterou můžeme reprezentovat aktuální číslicí spolu s odčítáním, je $\frac{4}{6}a$, protože můžeme odečíst maximálně dvakrát, tj. $\frac{2}{6}a$. Pokud je tedy $c < \frac{4}{6}a$, nic neděláme a jdeme zpracovávat další číslici v pořadí.

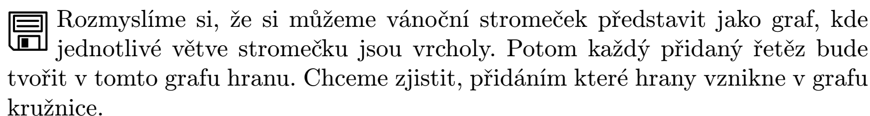
Pokud je $\frac{4}{6}a \leq c < a$ vypíšeme aktuální číslici a před ní jednu ($c < \frac{5}{6}a$) či dvě ($c \geq \frac{5}{6}a$) číslice o jedna menšího řádu. Zároveň od c odečteme a a přičteme vypsané menší číslice. Všimněme si, že potom nutně $c < \frac{1}{6}a$, a proto se nám nestane, že bychom nějakou číslici odečetli a následně opět přičetli.

Program (Python 3):

`http://ksp.mff.cuni.cz/viz/29-Z3-3.py`

Honza Knížek

29-Z3-4 Zdobení stromečku



Rozmyslíme si, že si můžeme vánoční stromeček představit jako graf, kde jednotlivé větve stromečku jsou vrcholy. Potom každý přidaný řetěz bude tvořit v tomto grafu hranu. Chceme zjistit, přidáním které hrany vznikne v grafu kružnice.

Pojďme si nejprve vyřešit jednodušší úlohu: dostali jsme nějaký graf a chceme zjistit, jestli v něm kružnice je. Na to nám stačí použít prohledávání do šířky či hloubky.[14] Ke každému navštívenému vrcholu si uložíme informaci, odkud jsme přišli. Pak se podíváme na všechny jeho sousedy. Pokud některý z nich už byl navštívený a není to ten, ze kterého jsme právě přišli, znamená to, že jsme objevili cyklus. Tohle ověření zvládneme v lineárním čase, $\mathcal{O}(N+M)$, kde N je počet vrcholů a M je počet hran.

Nyní se nabízí jednoduché řešení celé úlohy: postupně přidáváme do grafu hrany a po každém přidání ověříme, jestli už v grafu není cyklus. Ale pak ověření provádíme celkem M-krát, časová složitost je tedy $\mathcal{O}(M \cdot (N+M))$. To je celkem pomalé.

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

Zkusme to lépe. Všimneme si jedné věci: v průběhu přidávání hran nějakou dobu graf pořád neobsahuje cyklus, pak se tam objeví a od té doby už graf stále obsahuje cyklus.

My hledáme předěl mezi těmito dvěma stavy. K tomu můžeme použít binární vyhledávání. Zkusíme do grafu přidat prvních $M/2$ hran a podíváme se, jestli obsahuje cyklus. Pokud ano, musel vniknout přidáním některé z prvních $M/2$ hran, jinak musí vzniknout přidáním některé z druhých $M/2$.

Předpokládejme, že nastal první případ. Rozpůlíme tedy interval od 1 do $M/2$ – vytvoříme nový graf, do kterého přidáme jen prvních $M/4$ hran. Opět ověříme, jestli obsahuje kružnici, a podle toho víme, že provinilá hrana leží buď v intervalu od 1 do $M/4$, nebo od $M/4$ do $M/2$. Takhle pokračujeme v půlení, dokud nám nezbude jediná hrana – ta hledaná.

Abychom nemuseli graf stavět pořád znovu a znovu, stačí si u každé hrany pamatovat „čas“, ve kterém vznikla. Binární vyhledávání pak odpovídá půlení časových intervalů. Samotné prohledávání grafu na existenci cyklu upravíme tak, aby ignorovalo hrany, které teprve vzniknou.

Jeden pokus nám trvá lineární čas a binární vyhledávání provede $\mathcal{O}(\log M)$ kroků, dohromady $\mathcal{O}((N + M) \log M)$.

Ukážeme si ještě jedno, avšak pokročilejší, řešení. K tomu se nám bude hodit pojem *komponenta souvislosti* (viz opět grafovou kuchařku).

Na začátku jsou všechny vrcholy izolované a každý z vrcholů tvoří svou vlastní komponentu souvislosti. Pojďme se podívat, co přidávání hran do grafu s těmito komponentami provede.

První možnost je, že jsme přidali hranu mezi dvěma vrcholy, které patří do různých komponent. Potom jsme mezi nimi vytvořili novou cestu, a tedy jsme tyto dvě komponenty spojili v jednu.

Jestliže však byly oba vrcholy součástí stejné komponenty, existovala již dříve mezi nimi nějaká cesta. Potom přidáním této hrany vznikne mezi oběma vrcholy druhá cesta, která spolu s první utvoří cyklus.

Díky tomuto pozorování můžeme místo hledání cyklů v grafu kontrolovat, zda před přidáním každé hrany byly oba vrcholy ve stejné komponentě souvislosti. Jak takový stav ale zjistíme?

Mohli bychom jednoduše kontrolovat, zda jsou oba vrcholy součástí stejné komponenty pomocí prohledání do hloubky nebo do šířky. Avšak takové prohledání může potenciálně projít celý graf. Časová složitost takové kontroly je tedy $\mathcal{O}(N + M)$, která se pro každou hranu sečte na časovou složitost algoritmu $\mathcal{O}(M(N + M))$. To jsme si moc nepomohli.

Pojďme to zkusit lépe použitím něčeho, což už vymysleli informatici před námi. V algoritmu používáme dvě základní operace, a to přidání hrany a zjišťování komponenty souvislosti dvou vrcholů. Těmto operacím se v informatické hantýrce říká *union* a *find*. Existuje datová struktura zvaná *DFU*, která je umí

provádět velmi efektivně. Je popsaná v naší kuchařce o minimálních kostrách.[15] Využitím této struktury se poté úloha změní na postupné volání *find* a *union* pro každou hranu. Časová složitost takového algoritmu poté bude $\mathcal{O}(M \log N)$.

Program (Python 3):
`http://ksp.mff.cuni.cz/viz/29-Z3-4.py`

Václav Končický

29-Z3-5 Prvočíselné rozklady

V úloze jsme po vás chtěli najít postup, kterým rozložíte přirozené číslo na součin prvočísel. Odborněji řečeno se takovému procesu říká *faktorizace* a patří k velmi složitým problémům matematiky. Dodnes nikdo nezná algoritmus, který by dokázal faktorizovat velká (řádově tisíciciferná) čísla dost rychle na to, abychom se dočkali výsledku. Této vlastnosti se využívá v mnoha šifrovacích algoritmech, mimo jiné i ve známém RSA.

Popíšeme si autorské řešení. Na začátku běhu programu dostaneme číslo M a máme slíbeno, že jakékoliv číslo x, které dostaneme k faktorizaci, bude menší. Jeho rozklad se tudíž bude skládat z prvočísel menších než M. Pokud bychom je všechna dokázali najít, můžeme rozklad snadno určit: postupně projdeme prvočísla a každým z nich dělíme x tak dlouho, dokud je prvočíslem dělitelné.

Také si uvědomíme, že nám ve skutečnosti stačí prvočísla menší nebo rovna $\sqrt{M}$. Větší prvočíslo se v rozkladu nachází nejvýše jednou a získáme ho jako to, co nám zbude z x, když už nejde dělit žádným z prvočísel menších než $\sqrt{M}$ (rozmyslete si).

Jeden rozklad bude trvat čas $\mathcal{O}(k + \ell)$, kde k je počet prvočísel menších než $\sqrt{M}$ a ℓ je délka výsledného rozkladu. Rozklad nebude obsahovat více než $\mathcal{O}(\log M)$ prvočísel – každý činitel je alespoň dvojka. Odhadnout k shora pomocí M je těžké, spokojíme se tedy s $\mathcal{O}(\sqrt{M})$.

Ale jak prvočísla najít? Použijeme Eratosthenovo síto, které možná znáte z hodin matematiky. S tužkou a papírem funguje následovně: nejdřív si vypíšeme

[15] `http://ksp.mff.cuni.cz/viz/kucharky/kostry`

do řady všechna čísla od 2 do $\sqrt{M}$. Postupně budeme chtít zakroužkovat všechna prvočísla a škrtnout všechna složená.

Jeden krok algoritmu vypadá takto: vezmeme první neoznačené číslo (ani zakroužkované ani škrtnuté) a zakroužkujeme ho. Potom škrtneme všechny jeho násobky (pokud jsme zakroužkovali k, škrtneme $2k$, $3k$, …).

Takže v prvním kroku zakroužkujeme dvojku a škrtneme všechna sudá čísla. V druhém zakroužkujeme trojku a škrtneme všechny násobky tří. A tak dále. Vyzkoušejte a rozmyslete si, že to funguje.

Pokud ho chceme naprogramovat, místo papíru použijeme pole *sito* velikosti $\sqrt{M}$. V něm na i-té pozici bude jednička, pokud číslo i je škrtnuté, jinak nula. Navíc si budeme pamatovat poslední nalezené prvočíslo p. Pak je postup jednoduchý:

1. Vyplníme pole *sito* samými nulami (na začátku není nic škrtnuté).
2. $p \leftarrow 2$
3. Opakujeme dokud $p \leq \sqrt{M}$:
4. Přidáme p do seznamu prvočísel.
5. Škrtneme všechny násobky p: nastavíme $sito[2p]$, $sito[3p]$, $sito[4p]$, … na 1.
6. Posuneme se na nejbližší neškrtnuté číslo napravo: zvyšujeme p, dokud nebude $sito[p] = 0$.

Časová složitost síta velikosti $\sqrt{M}$ je $\mathcal{O}(\sqrt{M} \cdot \log\log \sqrt{M})$: důkaz není úplně jednoduchý, takže ho zde nebudeme uvádět. Můžete si ho přečíst v řešení úlohy 24-3-5.[16]

Tímto způsobem vyřešíme úlohu s $\mathcal{O}(\sqrt{M})$ pomocné paměti. Na předvýpočet budeme potřebovat $\mathcal{O}(\sqrt{M} \cdot \log\log \sqrt{M})$ času a na jeden dotat $\mathcal{O}(\sqrt{M})$. Pokud si ale dovolíme použít více paměti a času na předvýpočet, dostaneme další velmi dobrý algoritmus.

Když už složené číslo x škrtáme, známe prvočíslo p, které ho dělí. Pokud si ho zapamatujeme jako p_x, bude spočítání rozkladu triviálně jednoduché: Dostaneme-li x, do rozkladu přidáme p_x. Opakujeme s x/p_x, dokud nám nezbude prvočíslo. Takto rozklad spočítáme v $\mathcal{O}(\log x)$, tedy $\mathcal{O}(\log M)$.

Bohužel tím musíme rozšířit síto na všech M čísel. To nás bude stát $\mathcal{O}(M)$ pomocné paměti a výpočtem síta strávíme $\mathcal{O}(M \cdot \log\log M)$ času. I tak se to ale vyplatí, pokud bude počet dotazů N dostatečně velký.

Program (Python 3):

`http://ksp.mff.cuni.cz/viz/29-Z3-5.py`

Kuba Maroušek

[16] `http://ksp.mff.cuni.cz/viz/24-3-5/reseni`

29-Z3-6 Trable s dominem

Ačkoli to tak na první pohled nevypadá, tato úloha je grafová. Pokud toho o grafech ještě nevíte dost, určitě si přečtěte naši kuchařku o grafech.[17]

Jaký graf si tedy pod zadáním představíme? Obyčejný neorientovaný. Vrcholy nám budou tvořit symboly, které se vyskytují na dominu. Každá kostička obsahuje právě dva symboly, tedy nám v grafu budou tvořit hrany. Nebo lépe, hrany budou druhy kostiček, které máme k dispozici.

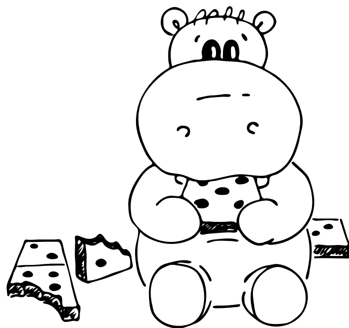
Nyní si vyberte vrchol v grafu, a zkuste přes hrany přecházet do jiných vrcholů. Taková posloupnost vrcholů a hran tvoří *sled* v grafu. Čemu to odpovídá v dominové analogii? Možná jste uhodli že postaveným hadům z cukroví.

Každý had vypadá jako sled v našem grafu a naopak, každý sled jde postavit jako hada z domina. Naši kamarádi se v příběhu dostali do situace, ve které nemohli dva hady spojit žádným „mezihadem". To znamená, že mezi koncovými symboly neexistoval v našem grafu žádný sled.

To se v grafu může stát pouze tehdy, pokud je graf *nesouvislý*. To znamená, že symboly jde roztřídit na (alespoň) dvě hromádky tak, aby každá kostička používala symboly pouze z jedné a žádná kostička dvě hromádky nepropojuje.

Mimo jiné to znamená, že dva hadi z příběhu neměli žádný symbol společný – jinak by šlo hady propojit přes něj.

Řešením úlohy je tedy vytvořit graf a ověřit, zdali je souvislý. To jde provést velice snadno prohledáním, například do hloubky. Během prohledávání budeme vrcholy obarvovat, a pokud nám na konci zbyl nějaký neobarvený, graf je nutně nesouvislý, a naši kamarádi si musí dát příště větší pozor.



Postavení grafu a jeho prohledání stihneme v $\mathcal{O}(N + M)$, kde N je počet vrcholů a M počet hran. Protože ale graf nemá izolované vrcholy (každý symbol se musel vyskytnout na nějaké kostičce), musí být $M \geq N$ a tedy je časová složitost pouze $O(M)$. Při rozumné reprezentaci v paměti nám bude stačit i $\mathcal{O}(M)$ paměti.

Program (Python 3):

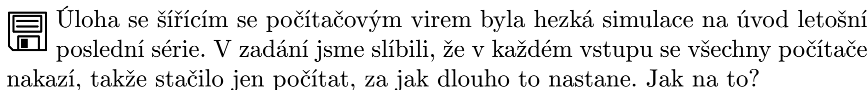
`http://ksp.mff.cuni.cz/viz/29-Z3-6.py`

Ondra Hlavatý

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

Čtvrtá série

29-Z4-1 Šíření viru



Úloha se šířícím se počítačovým virem byla hezká simulace na úvod letošní poslední série. V zadání jsme slíbili, že v každém vstupu se všechny počítače nakazí, takže stačilo jen počítat, za jak dlouho to nastane. Jak na to?

U každého počítače si budeme pamatovat, v jakém kroku byl nakažen (maximum z těchto čísel pak bude finální odpovědí) a kolik má nakažených sousedů (na počátku u všech počítačů nula). Zatím nezpracované nakažené počítače si budeme držet ve frontě, do které na počátku vložíme nakažené počítače ze vstupu (a nastavíme jim krok, ve kterém byly nakaženy, na nulu).

Pak nám stačí postupně vybírat počítače z fronty a zpracovávat je. Každému zatím nenakaženému sousedovi zpracovávaného počítače zvedneme počítadlo nakažených sousedů o jedna a pokud tím soused přesáhne kritickou mez (jeho počítadlo nakažených sousedů dosáhne alespoň poloviny počtu jeho sousedů), nakazíme ho taky – vložíme ho na konec fronty nakažených počítačů a nastavíme mu krok nakažení o jedna vyšší, než má aktuálně zpracovávaný počítač.

Tímto postupem postupně nakazíme všechny počítače (zastavíme se ve chvíli, kdy už ve frontě nebude žádný nezpracovaný nakažený počítač) a díky tomu, že na ukládání nezpracovaných nakažených počítačů používáme frontu, spočítáme správně i krok nakažení – počítače zpracováváme jakoby „ve vlnách“, nejprve zpracujeme všechny počítače, které se nakazí v kroku k (což může vést k nakažení nějakých počítačů v kroku $k+1$), než postoupíme dál. Pro nalezení odpovědi si tak stačí průběžně pamatovat maximální krok, ve kterém nakazíme nějaký z počítačů.

Přidáme ještě drobný implementační tip: Lepší, než porovnávat, že $A \geq B/2$ je porovnávat $2A \geq B$, vyhnete se tak hloupým zaokrouhlovacím chybám.

Program (Python 3):

`http://ksp.mff.cuni.cz/viz/29-Z4-1.py`

Jirka Setnička

29-Z4-2 Vybírání atrakcí

Tato úloha šla efektivně řešit *hladovým algoritmem*. Hlavní trik spočívá v tom, že před samotným přiřazováním atrakcí si vše setřídíme. Atrakce setřídíme jednoduše podle jejich rychlosti. Osoby setřídíme vzestupně podle nejvyšší snesitelné rychlosti a v případě rovnosti je dále setřídíme podle snesitelné rychlosti nejnižší.

Setřízené atrakce nyní přesuneme do seznamu nepoužitých atrakcí. Dále pro každou osobu nalezneme pozici nejlehčí nepoužité atrakce, kterou již zvládne. Od této pozice vybereme K za sebou jdoucích atrakcí a ty osobě přidělíme. Přidělené atrakce poté odstraníme ze seznamu nepoužitých atrakcí. Takto postupujeme, dokud nevyužijeme všechny atrakce.

Proč tohle může fungovat? Vezměme prvního člověka v popsaném pořadí, pojmenujme ho A. Nemáme příliš volnosti, nějakých K atrakcí mu přidělit prostě musíme. Každopádně má smysl uvažovat pouze ty, které snese. Mezi nimi náš algoritmus zvolí ty, jejichž rychlost je nejmenší. Klíčové v tuto chvíli je, že horní mez osob už pouze poroste – mezi lidmi, pro které tyto atrakci nejsou příliš pomalé, má osoba A nejnižší horní mez. Pokud bychom tyto atrakce přidělili komukoli jinému, mohlo by se stát, že všechny další atrakce budou na A příliš těžké. Naopak nikoli, tedy můžeme atrakce přidělit právě osobě A.

V algoritmu bychom dvakrát tvrdě narazili, pokud bychom neměli zaručeno, že řešení existuje. Poprvé tehdy, kdy přidělujeme K po sobě jdoucích atrakcí osobě, o které víme, že zvládne tu nejpomalejší z nich. Díky způsobu přidělování atrakcí ale víme, že pokud by osoba na nějakou atrakci nestačila, znamená to, že v seznamu nepoužitých už není více atrakcí, na které by osoba stačila – řešení nemůže existovat. Podruhé pak když předpokládáme, že s dokončením seznamu atrakcí jsme přiřadili K atrakcí každému člověku.

Nyní se zaměřme na časovou složitost. Počet atrakcí si označíme $M = NK$. Jelikož třídíme osoby i atrakce, nedostaneme se na složitost lepší, než $\mathcal{O}(M \log M)$.

Jak to je ale s vybíráním atrakcí? Většina operací závisí na správném výběru datové struktury pro ukládání zatím nepoužitých atrakcí.

Mohli bychom použít pole a v něm vždy vyhledat první použitelnou atrakci binárně. Bohužel, jedno odebrání (i více najednou) prvků v poli trvá $\mathcal{O}(M)$ – po odebrání musíme zacpat vzniklou díru přesunutím všech prvků napravo. Složitost bychom si tím pohoršili na $\mathcal{O}(NM + M \log M)$.

Každopádně, na plný počet bodů tento postup už stačil. Komu to ale nestačí, a věří že to musí jít lépe, nechť čte dál.

Na reprezentaci seznamu místo pole použijeme vyhledávací strom.[18] Po setřídění atrakcí z nich sestavíme vyvážený vyhledávací strom, to nám stačí i

[18] http://ksp.mff.cuni.cz/viz/kucharky/stromy

v $\mathcal{O}(M \log M)$. Vyhledání prvku i jeho odstranění je rovněž rychlé, obě tyto operace vyhledávací strom zvládne v $\mathcal{O}(\log M)$. Prvních K prvků od daného místa vybereme jednoduše pomocí hledání následníka, které trvá rovněž $\mathcal{O}(\log M)$.

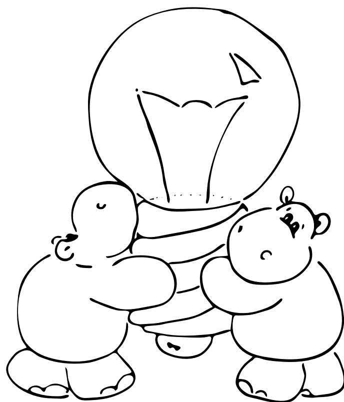
Podotkněme, že není potřeba žádný speciální samovyvažovací strom. Stačí strom na začátku programu postavit rozumně vyvážený a později z něj jen odebírat. Tím se strom sice znevyváží, pořád ale bude mít hloubku nejvýše $\mathcal{O}(\log M)$.

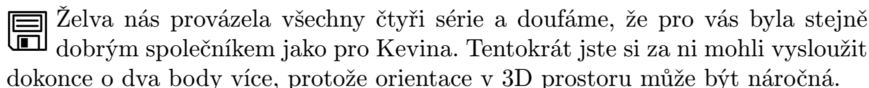
Celkem v algoritmu provedeme M vložení a odstranění, to nám celkově zabere $\mathcal{O}(M \log M)$ času. N atrakcí jsme vyhledali, zbylých $M - N$ jsme našli jako následníky. Po sečtení všech operací nám tudíž vyjde, že časovou složitost udržíme na $\mathcal{O}(M \log M)$.

Program (Python 3):

`http://ksp.mff.cuni.cz/viz/29-Z4-2.py`

Václav Končický





29-Z4-3 Želva v akváriu

Želva nás provázela všechny čtyři série a doufáme, že pro vás byla stejně dobrým společníkem jako pro Kevina. Tentokrát jste si za ni mohli vysloužit dokonce o dva body více, protože orientace v 3D prostoru může být náročná.

Jak jsme psali v zadání, tentokrát nestačí pamatovat si jen směr, kterým se želva dívá. Je potřeba si pamatovat ještě alespoň jeden. Například, kterým směrem má natočený krunýř, nebo kam míří její pravý bok.

Z těchto tří směrů si stačí pamatovat jen libovolné dva – třetí se dá vždy dopočítat. Úplně nejsnáze si ale řešení úlohy pořídíte, pokud si budete pamatovat všechny tři směry. Přepočítávat totiž stačí vždy pouze dva – směr v ose, okolo

které se želva otáčí, se nikdy nemění. Jinými slovy, při každém otočení se změní pouze dva směry.

Všechny směry si budeme pamatovat jako *vektory* jednotkové délky. Takový vektor má tři složky, každou v jedné ze tří souřadných os. Každá složka obsahuje buď 0, 1 nebo −1, navíc je právě jedna složka nenulová. Například vektor ukazující na sever (směr pohledu želvy na začátku programu) si zapamatujeme jako $[1, 0, 0]$ – stejně jako v zadání.

Představme si nejprve jednoduché otočení želvy doprava. Směr nahoru, tedy kam míří krunýř, se touto rotací nezmění. Směr, kam se želva dívá, stačí nahradit za ten, kam byl předtím natočen pravý bok. Zbývá už jen spočítat nový směr pravého boku. Ten je rovnoběžný s tím, kam se předtím želva dívala – jen je opačným směrem.

Pro celou operaci otočení doprava tedy stačí prohodit směr pohledu a pravého boku, a následně pravý bok obrátit – každou složku vektoru vynásobit −1. Ostatní rotace jsou už jen variací na stejné téma. Vždy stačí jen některé dva směry prohodit a jeden z nich obrátit.

Program (Python 3):

`http://ksp.mff.cuni.cz/viz/29-Z4-3.py`

Ondra Hlavatý

29-Z4-4 Hledání součtu

Zadání úlohy po nás chtělo najít v posloupnosti N čísel takovou její souvislou podposloupnost, že součet prvků této podposloupnosti bude co nejblíže zadanému číslu C. Pro vyřešení několika prvních vstupů postačil jednoduchý přístup k řešení – stačilo vyzkoušet všechny možné podposloupnosti a z nich vybrat tu s nejmenším rozdílem jejího součtu a čísla C. Úloha se ale dá snadno vyřešit v lineárním čase, jak si za chvíli ukážeme.

Zkusme si nejprve úlohu trochu zjednodušit. Nebudeme hledat podposloupnost s nejbližším součtem k číslu C, ale podposloupnost, jejíž součet je roven právě C. Každá souvislá podposloupnost začíná a končí na nějakých indexech původní posloupnosti. Označme si tato místa jako *zac* a *kon*, přičemž vždy platí $zac \leq kon$. Na začátku položíme $zac = kon = 0$. Tyto indexy budeme při běhu programu postupně zvyšovat. Zavedeme si dále proměnnou S, ve které budeme mít uložen součet prvků zadané posloupnosti mezi místy *zac* a *kon*.

Provedeme nyní několik pozorování. Pokud je $S > C$, nemá cenu zvyšovat *kon* a raději zvýšíme *zac*. Zadání předpokládá pouze kladná čísla, pokud bychom *kon* o 1 zvýšili, zvýšilo by se nám i S o nově přidaný prvek. My ale chceme, aby $S = C$. Pokud naopak zvýšíme o 1 index *zac*, S se nám sníží, což je v této situaci žádoucí. Obdobně, pokud je $S < C$, nemá cenu zvyšovat *zac*. Raději zvýšíme *kon* o 1. A nakonec, pokud je $S = C$, máme nalezeno řešení a můžeme skončit.

Tato pozorování nám dávají přímý návod na sestavení algoritmu pro zjednodušenou verzi úlohy. Jak toto řešení rozšířit tak, aby zvládlo i úlohu ze zadání? Velmi jednoduše! Stačí si při běhu ukládat dosud nejlepší nalezené řešení a v každé iteraci pak kontrolovat, jestli aktuální řešení není náhodou lepší, než to doposud nejlepší nalezené. O tom, které řešení je lepší, lze rozhodnout velmi jednoduše – z absolutní hodnoty rozdílu S a C. Máme-li dvě řešení A, B a k nim příslušné S_A respektive S_B, porovnáme $|S_A - C|$ s $|S_B - C|$ a vybereme menší z nich. Výpočet můžeme ukončit ve chvíli, kdy máme $S = C$ nebo kdybychom *zac* zvýšili na N. (pozn. indexujeme od 0, poslední prvek posloupnosti má index $(N - 1)$.)

Povšimněme si, že způsob manipulace s indexy v podstatě odpovídá přidávání prvků zadané posloupnosti do fronty a jejich odebírání z ní. Každý prvek se přitom může do fronty dostat maximálně jednou. Odtud nám plyne i časová složitost řešení $\mathcal{O}(N)$.

Program (C++):

`http://ksp.mff.cuni.cz/viz/29-Z4-4.cpp`

Jan „Toman“ Tománek

29-Z4-5 Hanojské věže jinak

Dostali jsme nějaké rozestavení, jak máme vlastně začít? Dobrým prvním krokem by bylo rozmyslet si na jaké tyči chceme věž postavit. Všimněte si, že je vždy nejlepší věž postavit na tyči, kde už je největší disk. Proč tomu tak je? Kdybychom chtěli věž postavit jinde, musíme uvolnit největší disk (nesmí na něm být žádné další disky) a celou tyč, kde chceme věž postavit, na poslední tyč tedy musíme vyskládat všechny zbylé disky. Pak přesuneme největší disk a pokračujeme ve stavbě. Nicméně krok přesunu největšího disku si můžeme odpustit a a rovnou stavět na největším disku, nijak si nepohoršíme.

Nyní jsme si vybrali tyč, kde budeme stavět věž a navíc tam máme už první, největší disk. Dřív než než na něj začneme skládat menší disky musíme na něj nějak dostat druhý největší. Představme si, že největší disk neexistuje. Snažíme se tedy vystavět věž s výškou o jeden disk menší než původně, tentokrát ale na konkrétní tyč – na tu, kde se nachází disk největší. Poznamenejme, že tím, že je největší disk největší, se nám nic nezmění tím, že ho budeme ignorovat – můžeme na něj dát libovolný disk, stejně jako na prázdnou tyč.

Pomohli jsme si tedy vůbec? Stále máme za úkol postavit věž, byť o kousek menší, navíc na konkrétní tyč. Ukážeme si, že ano. Označme si druhý největší disk (popř. největší, který neignorujeme) jako D. Jako první musíme dostat D na cílovou tyč. Můžeme mít štěstí a D tam už bude, pak můžeme tento krok přeskočit, pokud máme ale smůlu, chce to trochu úsilí, pojďme se na to podívat.

Abychom mohli přesunout D na cílovou tyč, musí být cílová tyč prázdná (až na ignorovaný disk) a D volná (nesmí na něm být žádný další disk). Všechny ostatní disky tedy musí být vyskládané do věže na poslední tyči. Poté můžeme přesunout disk D a poté celou vyskládanou věž na cílovou tyč.

To vypadá, jako bychom se zase nikam nedostali. Abychom vyřešili problém přeskládání věže, musíme vyřešit problém přeskládání dvou věží (nejprve na uvolnění cílové tyče a D, poté na přeskládání zbylých disků na cílovou tyč). Nicméně klíčové je, že tyto věže jsou opět o jeden disk nižší. Můžeme je vyřešit právě tím samým algoritmem! Abychom uvolnili cílovou věž a D, můžeme na chvíli ignorovat D a jako D dočasně označit další největší disk. Pak necháme proběhnout celý algoritmus, poté přesuneme naše původní D a pak znova necháme náš algoritmus pracovat.

Jenže skončí toto někdy? Vždyť v každém průběhu algoritmu vytváříme dva další. Musíme si uvědomit, že až budeme pracovat s jedním diskem, nepotřebujeme vůbec spouštět nějaký složitý algoritmus, stačí přesunout disk na správné místo, pokud tam ještě není. Ti zkušenější z vás v tomto postupu zcela správně vidí analogii s tzv. *rekurzí*.

Nakonec ještě připomeneme, proč toto vyskládání je nejrychlejší možné. Na začátku jsme opatrně vybrali tu nejlepší tyč, na které věž stavět. Poté už všechny kroky byly nutné (uvolnit disk a cílovou tyč, přeskládat zbylé disky), takže jsme tento postup nemohli vůbec urychlit.

Dominik Smrž

29-Z4-6 Tajná síť taháků

Nejprve si zkusíme problém trochu formalizovat. Místo tajné sítě taháků si můžeme rovnou představit graf. Jeho vrcholy budou Kevinovi spolužáci, hrany povedou mezi těmi spolužáky, kteří si mezi sebou předávají taháky napřímo.

Nebude to ale jen tak ledajaký graf – v zadání jsme vám slíbili, že nebude obsahovat cykly. Navíc, protože jde o jednu síť, můžeme předpokládat, že jde o graf souvislý. Graf, který je souvislý a neobsahuje cykly, je strom.

Bez újmy na obecnosti si strom libovolně zakořeníme. Nyní se podívejme, co se stane, pokud ze stromu odtrhneme libovolný vrchol v. Strom se rozpadne na několik podstromů: jeden za každého syna v, a jeden navíc za otce v. Tyto části mají dohromady $|V| - 1$ vrcholů. Jak také jinak, že :)

Na pomoc si přizveme obyčejné prohledávání do hloubky. Budeme postupovat stejně, jako bychom chtěli spočítat počet vrcholů ve stromě: z rekurze do listů vrátíme 1, jinak součet hodnot vrácených ze synů zvýšený o jedna.

Tento algoritmus jednoduše modifikujeme. Během návratu z rekurze víme, jaká je velikost všech podstromů. Pokud odečteme jejich součet od celkového počtu vrcholů (a odečteme ještě jedničku navíc), získáme velikost zbylé části

grafu. Není-li žádná z nich větší než polovina, vyhráli jsme, hledaným vrcholem je ten aktuální.

Takto najdeme toho správného spolužáka v lineárním čase. Najdeme ho ale vždy?

Tak, jak byla úloha zadána, se může stát, že žádný takový neexistuje. Vezměte si třeba strom tvořený dvěma vrcholy spojenými hranou. Ať odebereme libovolný z nich, zbylý podstrom bude mít vždy alespoň polovinu původního počtu vrcholů.

Protože ale náš algoritmus projde nakonec všechny vrcholy, tak si můžeme být jistí, že pokud ten správný vrchol existuje, tak jej najdeme.

Kdybychom ale požadovali ostře více než polovinu původního počtu vrcholů, aby síť zůstala v provozu, situace by byla zajímavější. O tom zas ale někdy příště...

Ondra Hlavatý

Pořadí řešitelů KSP-Z

Pořadí	*Jméno*	*Škola*	*Ročník*	*Úloh*	*Bodů*
0.				24	264.0
1.	Jakub Šťastný	G BO-Řeč	2	24	258.0
2.	Lucia Krajčoviechová	GJHroncaBA	1	22	238.0
3.	Karel Balej	G_Rokycany	2	24	236.0
4.	Daniel Skýpala	GTomkovaOL	−1	24	222.7
5.	Petr Borňás	G_Roudnice	1	24	219.3
6.	Petr Aubrecht	GHeyrovPH	2	21	218.0
7.	Petr Šimůnek	G_Hořice	4	21	205.0
8.	Michaela Bobeničová	GPošKošice	2	19	189.0
9.	Vladimír Chudý	ZŠRonov	0	20	180.0
10.	Ondřej Jamelský	G Cheb	−1	21	179.4
11.	Terézia Strišovská	GJHroncaBA	1	18	179.3
12.	Ondřej Gonzor	G Brandýs	0	19	165.0
13.	Petr Budai	G JGJ PH	0	17	162.0
14.	Jiří Löffelmann	GLitoměřPH	3	16	160.0
15.	Michal Kodad	SPŠ_Smíchov	1	16	149.0
16.	Jan Kotovský	GPísnickáPH	−2	19	145.0
17.	Andrej Pajtaš	GLitoměřPH	1	14	140.5
18.	Jakub Šuráň	GStrážnice	2	15	140.0
19.	Erik Kučák	GHorMichal	4	13	139.0
20.	Vojtěch Březina	GCoubTábor	0	16	134.0
21.	Dávid Šutor	GTerVans	2	12	127.5
22.	Erik Berta	GAlejKošice	2	18	123.7
23.	Robert Jaworski	GÚstavníPH	−1	15	123.0
24.–25.	Jan Kaifer	GČesBrod	1	14	120.0
	Jan Vodstrčil	G VMýto	0	14	120.0
26.–27.	Martin Bencko	GOhradníPH	0	20	119.0
	Kateřina Čížková	G_Rokycany	3	13	119.0
28.	Anna Hollmannová	GSRandyJN	0	13	116.0
29.	Tomáš Domes	MendelG_OP	4	11	108.0
30.	Jakub Jirkal	GJungmanLT	2	12	107.0
31.	Jakub Ucháč	ŠMaVVzt	1	12	103.0
32.	Vincent Orlovský	GTerVans	2	11	99.0
33.	Dalibor Kramář	G BO-Řeč	2	11	98.0
34.	Zuzana Urbanová	GFXŠaldyLI	3	13	97.0
35.	Jozef Mikuláš	CZŠJBosca	0	10	95.0
36.	Vojtěch Hudec	G_ČTřebová	3	9	94.0
37.	Jan Kučera	GFKřižíka	0	12	93.0
38.	Jaroslav Knápek	GLesníZlín	0	12	92.3

Pořadí řešitelů KSP-Z

Pořadí	*Jméno*	*Škola*	*Ročník*	*Úloh*	*Bodů*
39.	Ondřej Wrzecionko	GTěš	2	13	92.0
40.	Lucie Vomelová	GŠpitálsPH	1	18	88.0
41.	Radoslav Hašek	G_Čáslav	3	11	84.0
42.	Jakub Brož	PČGKarVary	3	10	83.0
43.	Andrej Tomči	GHorMichal	4	8	82.0
44.	David Nápravník	GLitoměřPH	4	10	81.7
45.	Lukáš Caha	GZborovPH	3	8	81.0
46.–47.	Vojtěch Brož	GBudějovPH	2	9	78.0
	Gabriela Pachlová	G_ČTřebová	2	9	78.0
48.	Petr Macháček	G_TýnNVlt	1	8	77.0
49.	Vojtěch Káně	G Brandýs	1	11	76.0
50.	Kateřina Nová	G_Vimperk	4	11	73.7
51.	Václav Kelímek	SPŠBruntál	4	9	72.0
52.	Dominik Pilný	G_Ostrov	2	9	69.0
53.	Martin Sobotka	GLitoměřPH	1	7	68.0
54.	Dávid Oravec	G DubNVáh	2	7	66.0
55.–56.	Jaroslav Paidar	SPŠMasarLI	3	10	65.0
	Rajmund Hruška	GPošKošice	4	6	65.0
57.	Martin Zmitko	G FrýdlNOs	1	9	64.0
58.	Josef Polášek	GKepleraPH	1	8	63.7
59.–60.	František Kmječ	G Brandýs	1	6	62.0
	Barbora Plačková	GHlu	−1	7	62.0
61.	Ondřej Krsička	GJarošeBO	1	6	61.0
62.	Michal Mlčoch	G UherBrod	2	9	60.0
63.–65.	Daniela Hrbáčová	G Wicht	3	12	58.0
	Pavel Martinec	GLesníZlín	3	6	58.0
	Filip Masár	PiarGNitra	3	6	58.0
66.	Ondřej Cach	SPSE_Pard	1	6	54.0
67.–68.	Tomáš Sládek	GJHroncaBA	2	8	49.0
	Ladislav Töpfer	G DrJPekMB	2	5	49.0
69.	Janek Hlavatý	GJirsíkaČB	−2	6	48.0
70.–71.	Jan Chybík	SPŠMasarLI	2	6	46.0
	Vojtěch Kuchař	ZŠ Sobotka	0	6	46.0
72.–73.	Martin Hofbauer	G BO-Řeč	2	5	44.0
	Tomáš Vítek	G_Břeclav	0	5	44.0
74.	Jan Koška	GJírovcČB	−3	5	43.0
75.–77.	Timea Szöllősová	G_Gröss_BA	1	5	41.0
	Matouš Vondrášek	GJírovcČB	1	5	41.0
	Jan Štěch	GJirsíkaČB	0	8	41.0

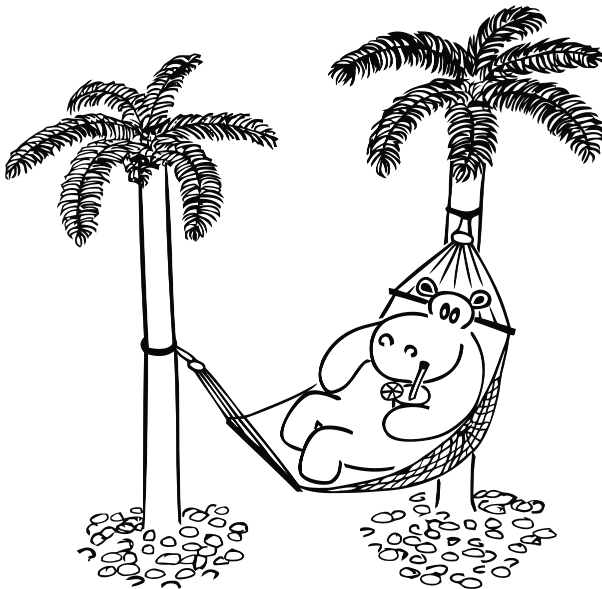
Pořadí	*Jméno*	*Škola*	*Ročník*	*Úloh*	*Bodů*
78.–82.	Martin Horáček	GŠumperk	4	4	40.0
	Martin Melicher	GPošKošice	2	4	40.0
	Adam Perinay	GJHroncaBA	3	4	40.0
	Dennis Pražák	GJirsíkaČB	2	4	40.0
	Jakub Szymsza	CmGy PV	1	4	40.0
83.	Vít Beran	MasG_Plzeň	3	5	38.0
84.	Radim Buráň	G UherBrod	2	6	37.0
85.–87.	Jan Juračka	GBystnPern	2	4	36.0
	Michael Kozel	GZborovPH	3	4	36.0
	Magdaléna Rýdlová	GLesníZlín	3	4	36.0
88.	Václav Čermák	GKlatovy	4	4	34.0
89.	Tomáš Chabada	SPŠMasarLI	1	4	32.0
90.–95.	Jan Bíl	GDašickáPA	4	4	30.0
	Dominik Dinh	GNVPláníPH	2	5	30.0
	Tomáš Dulava	GMatOS	3	3	30.0
	Vít Gaďurek	Neuvedená	2	4	30.0
	Vojtěch Krupka	GJungmanLT	3	4	30.0
	Erik Řehulka	ŠPMNDaGB	1	4	30.0
96.–98.	Tuan Anh Hoang	GZborovPH	3	5	29.0
	Vladimír Holý	CírkG Plzeň	2	6	29.0
	Jiří Kvapil	GTomkovaOL	−1	7	29.0
99.–108.	Michal Grňo	BiGyBBHK	4	3	28.0
	Jaroslav Horáček	GymVyš	3	3	28.0
	Lucie Kubíčková	GFXŠaldyLI	3	3	28.0
	Tomáš Nguyen	SPŠÚžlabPH	2	4	28.0
	Adéla Návratová	ZŠ MTyrš	−1	3	28.0
	Ondrej Potúček	G_GolNitra	3	3	28.0
	Vojtěch Poupa	CírkG Plzeň	−1	3	28.0
	Adrián Rošinec	GHorMichal	4	3	28.0
	Eliška Vlčinská	GHladnov	2	3	28.0
	Benedikt Žour	G UherBrod	2	4	28.0
109.	Jiří Janoušek	GBudějovPH	1	5	27.0
110.	Jan Martínek	GTomkovaOL	2	3	26.0
111.	Evgeniya Knyazeva	GNVPláníPH	3	4	25.0
112.–113.	Roman Ondráček	GBoskovice	3	3	24.0
	Lukáš Vavřík	GNeumannŽR	3	4	24.0
114.	Tomáš Dostál	MendelG_OP	2	4	23.0
115.–116.	Alexandra Géciová	GJHroncaBA	1	8	22.0
	Marek Zelený	GVoděraPH	3	3	22.0

Pořadí řešitelů KSP-Z

Pořadí	*Jméno*	*Škola*	*Ročník*	*Úloh*	*Bodů*
117.–119.	Ondřej Buček	GJarošeBO	3	3	20.0
	Matouš Bílek	GJŠkodyPŘ	2	3	20.0
	Ondřej Gajda	GTěš	0	3	20.0
120.–127.	Dávid Daubner	GVaršŽilina	2	3	18.0
	Tereza Hladíková	JazG HK	3	2	18.0
	Štěpán Košan	GKlatovy	4	2	18.0
	Václav Luňák	GDašickáPA	4	2	18.0
	Ondřej Mašek	GEBenešeKL	3	2	18.0
	Tomáš Novotný	G BO-Řeč	3	2	18.0
	Pavel Turinský	G Brandýs	4	2	18.0
	Matěj Šmíd	SPŠÚžlabPH	3	2	18.0
128.–130.	Martin Hubata	GMikulášPL	1	5	16.0
	Arian Adam Ott	GSOŠRok	0	2	16.0
	David Rajchman	MasG_Plzeň	0	2	16.0
131.	Petr Doubravský	AkademG_PH	1	3	15.0
132.	Dominik Tulak	SPŠMasarLI	1	3	14.0
133.–134.	Ondřej Hráček	GOlgHavl	0	2	12.0
	Vojtěch Lengál	GZborovPH	3	1	12.0
135.	Jindřich Dítě	VOSPŠŽďár	1	1	11.0
136.–137.	Mária Ďuračková	GJHroncaBA	2	1	10.0
	Václav Šraier	GČeskoliPH	4	2	10.0
138.–144.	Matúš Ferech	GJHroncaBA	4	1	8.0
	Frantisek Hanzlik	ZŠ Elem	−1	1	8.0
	Jan Hřebenář	20. ZŠ	0	1	8.0
	Filip Kastl	GKepleraPH	1	1	8.0
	Jan Kutálek	GSOŠNovJicin	3	3	8.0
	Šimon Prokop	21. ZŠ	0	2	8.0
	Natálie Volfová	GJírovcČB	1	1	8.0
145.–148.	Petr Chotěborský	GSla	0	1	6.0
	Jakub Hroník	GJiříPoděb	3	1	6.0
	Daniel Rozehnal	GJWolkraPV	1	1	6.0
	Jakub Švojgr	GČeskáČB	−2	1	6.0
149.	Petr Kabourek	G BO-Řeč	1	2	4.0
150.	Tomáš Husák	GLitoměřPH	3	3	3.0
151.–152.	Filip Bouda	SŠStrŠvejc	1	1	2.0
	Matej Nižník	GPošKošice	2	1	2.0
153.	Štěpán Henrych	GŽat	1	1	0.3

KSP

Hlavní kategorie KSP



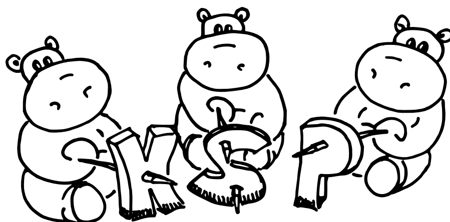
Zadání úloh KSP

První série

„Nepovídej mi, že máme zase místo placek chlebové uhlí!" poznamenal naoko výhružně Warin a poklepal si na meč zavěšený za pasem. Samozřejmě to nemyslel vážně, ale s Rheou se rádi navzájem škádlili. Znal se s ní déle než s kýmkoliv jiným z jejich malé družinky a prožili toho spolu již hodně – on jako rytíř řádu, ona jako nadaná kouzelnice.

Zbytek jejich družiny nacházející se v severské pustině tvořili ještě nadaný zloděj a lukostřelec Gorf a druhý rytíř, mladý Lian. Do této prapodivné skupinky je svedl důležitý úkol a už více než týden putovali daleko za známými a bezpečnými cestami království.

Teď ale byl jejich hlavní starostí ukrutný hlad. Připálené placky sice nejsou zrovna lahůdka, ale když se z nepřipálené strany namažou máslem, tak se jíst dají. Jenom při rychlém sundavání z ohně je Rhea poskládala náhodně na jednu hromádku.

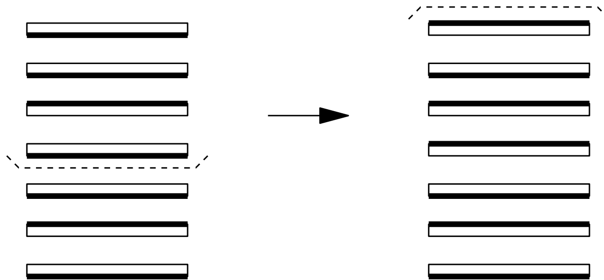


29-1-1 Připálené placky — 8 bodů

Máme na sobě položenou spoustu placek, každá z nich je z jedné strany připálená a z druhé strany krásně dozlatova. Rádi bychom všechny placky otočili nepřipálenou stranou nahoru, abychom je pak mohli všechny rychle namazat máslem.

Bohužel placky jsou ještě příliš horké na to, abychom je brali do rukou. Jediné, co můžeme dělat, je podebrat si několik vrchních placek pánvičkou a celou tuto část otočit. Jak to vypadá, si můžete prohlédnout na následujícím obrázku.

Dostanete zadáno, jak vypadá hromada placek (posloupnost říkající, které z placek leží nahoru připálenou stranou a které nepřipálenou). Vaším úkolem je najít co nejkratší posloupnost podebrání a otočení takovou, aby se po jejím provedení všechny placky nacházely nepřipálenou stranou nahoru.



Další den ráno uklidila družina chvatně své ležení a vydala se dál k úbočí kopce, který se před nimi rýsoval. Nacházelo se zde jedno z těch samostatných měst vzdorujících místní divočině a občasným nájezdům goblinů. Tohle vypadalo, že i docela prosperuje.

Protože jim docházely zásoby a navíc potřebovali zjistit nějaké informace, vydal se Warin s ostatními k městské bráně. Warin s Lianem schovali svá brnění do nenápadných ranců na nákladním mezku a štíty se symboly řádu zakryli plátnem. Bezpečnější bylo tvářit se jako nějací náhodní dobrodruzi.

Když u městské brány uplatili strážného dvěma zlaťáky vtisknutými do dlaně, nemuseli ani odpovídat na žádné otázky a byli vpuštěni dovnitř. Po přiblížení se k tržišti se ale družinka stala svědky nějaké hádky. Skupina místních kupců se dohadovala, kdo má komu co zaplatit. Rhea se rozhodla, že se mezi ně vetře, zkusí jim poradit a přitom získat nějaké informace.

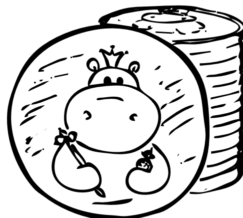
29-1-2 Kupecké počty — 11 bodů

Skupina kupců se společně podílela na jedné velké investici. Každý platil nějakou část, některé z nich to stálo více a některé naopak méně. Nyní by si chtěli všechny náklady rovnoměrně rozdělit tak, aby ve výsledku investovali všichni stejně.

Kupci se mohou vyrovnat tím, že ti, kteří platili málo, dají nějaký obnos těm, kteří platili hodně. O každém předání peněz se ovšem musejí dělat záznamy v účetních knihách, a tak by kupci chtěli provést těchto převodů peněz co možná nejméně.

Navíc se ale žádný z kupců nechce zadlužit více, než už je, nebo se naopak nechat přeplatit víc, než už přeplacený je. Není tak třeba možné kupci, který má dostat 100 zlatých, dát 120 zlatých, protože by se tím stal o 20 zlatých přeplacený. Stejně tak není možné poslat jakékoliv peníze kupci, který ostatním dluží (zadlužil by se ještě víc).

Vymyslete postup, který pro zadané částky zaplacené jednotlivými kupci spočítá, kdo komu má kolik dát tak, aby byla respektována uvedená pravidla a převodů bylo málo.



Spočítat řešení s úplně nejmenších počtem převodů je těžký problém a tak to po vás ani nechceme (jako dobrovolné cvičení si však můžete zkusit rozmyslet, proč je to těžký problém). K vyřešení úlohy stačí, když vymyslíte postup vedoucí k maximálně dvojnásobnému počtu převodů peněz, než je optimum. Důležitou částí je i důkaz, že uděláte nejvýše dvojnásobek převodů, než je nezbytně nutné udělat.

Rhea se rychle prodrala davem a využila svého přirozeného půvabu k tomu, aby si od kupců získala pozornost. Tu prohodila nějaké slovo, jinde poradila a kupci se rychle dostali k vzájemné dohodě.

Za necelých dvacet minut se Rhea opět vynořila u zbytku skupinky a vítězoslavně zahlásila: „To byla hračka, to mě bavilo. Od tamtoho obchodníka s kožešinami jsem se dozvěděla, kdo by mohl vědět víc o té potvoře. Je to stopař, kterého najdeme prý v kasárnách městské gardy.“

Vydali se tedy do kasáren stojících u východní městské brány. Jak už to tak v těchto malých městech bývá, výzbroj místních gardistů byla dost různorodá – od různě dlouhých mečů přes všelijaká kopí až k náhodně vypadající sbírce halaparten. Co ale oba rytíře příjemně překvapilo, byla důslednost, se kterou se místní velitelé věnovali výcviku. Teď zrovna cvičili gardisté s kopími.

29-1-3 Střídání zbraní — 12 bodů

Gardisté městské gardy mají k dispozici přesně tolik kopí, kolik jich v gardě slouží. Každé kopí je ale jinak dlouhé a jednotliví gardisté jsou také různě vysocí.

Když si gardisté vybírají, se kterým kopím půjdou bojovat, jsou ochotni si vzít jenom kopí nanejvýš tak dlouhé, jak jsou oni vysocí (neboli gardista nemůže mít delší kopí, než je jeho výška).

Při výcviku dbají velitelé na to, aby si co nejvíce gardistů zkusilo bojovat s co nejvíce různými kopími. Zajímalo by je tedy, kolik existuje různých možností, jak si gardisté mohou rozdělit kopí tak, aby každý dostal právě jedno a to nebylo vyšší, než je on sám. Vaším úkolem to je pro zadané délky kopí a výšky gardistů spočítat.

Příklad: Pro kopí o délkách $7, 3, 6, 1$ a pro gardisty vysoké $3, 7, 4, 8$ existují celkem 4 možnosti, jak si kopí mohou rozdělit. Gardisté ve výše uvedeném pořadí dostanou kopí délek $(1, 6, 3, 7)$, $(3, 6, 1, 7)$, $(1, 7, 3, 6)$ nebo $(3, 7, 1, 6)$.

Zadání úloh KSP – 1. série

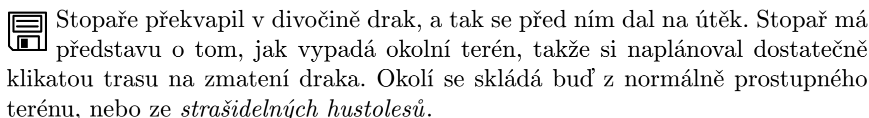
Na dvoře kasáren se rozdělili a začali se co nejnenápadněji vyptávat osamocených gardistů. Nechtěli moc rozhlašovat svoji příslušnost ke královskému rytířskému řádu, to by tady daleko v severních krajích mohlo způsobovat problémy. Lepší bylo zůstat neznámými pocestnými.

Gorfovo písknutí je po chvíli všechny svolalo dohromady. Na jedné straně nádvoří Gorf objevil malého chlapíka, který odpovídal popisu od kupce.

„Prý jsi zahlédl nějakou velkou potvoru,“ spustil Warin a vtiskl muži do ruky pár zlaťáků. Muž si Warina nervózně prohlédl a pak je všechny posunkem vyzval, aby ho následovali za roh. Tam jim začal líčit své setkání s drakem.

Popsal jim, že procházel místem, kudy už předtím šel mnohokrát, když tu se přes skalní převis nad ním přehnala obrovská potvora – drak. Chvíli se před ním schovával a pozoroval ho ukrytý ve křoví, čekající na příležitost. A když se naskytla, tak vyběhl, jak nejrychleji dovedl.

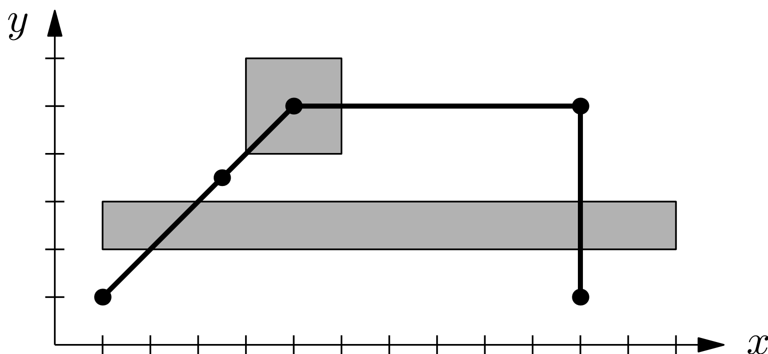
29-1-4 Zběsilý útěk — 10 bodů



Stopaře překvapil v divočině drak, a tak se před ním dal na útěk. Stopař má představu o tom, jak vypadá okolní terén, takže si naplánoval dostatečně klikatou trasu na zmatení draka. Okolí se skládá buď z normálně prostupného terénu, nebo ze *strašidelných hustolesů*.

Stopařova trasa je tvořena lomenou čárou (neboli na sebe navazujícími úsečkami), která může procházet i skrz hustolesy, ale v takovém případě v nich stopař zvládne běžet jen poloviční rychlostí. Hustolesy jsou obdélníkové oblasti, které budou, stejně jako stopařova trasa, zadány na vstupu.

Navíc máme slíbeno, že každou úsečku na trase protíná nejvýše jeden hustoles a že hustolesů je zhruba stejně mnoho, jako úseček trasy. Představu můžete získat třeba z obrázku níže. Vaším úkolem bude spočítat, jak dlouho bude stopařovi proběhnutí celé trasy trvat.



Při řešení by se vám mohlo hodit nahlédnout do geometrické kuchařky.[19]

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

Formát vstupu: Na prvním řádku vstupu dostanete tři celá čísla oddělená mezerou: Nejprve rychlost stopaře v normálním terénu V vyjádřenou v metrech za sekundu. Dále pak číslo N udávající počet bodů na stopařově trase (mezi těmito body se stopař pohybuje po úsečce) a nakonec číslo H udávající počet hustolesů (pozor, jeden hustoles může zasahovat i do více úseček).

Na dalších N řádcích naleznete vždy dvě čísla, každá dvojice udává souřadnice jednoho z bodů na stopařově trase. A nakonec na dalších H řádcích najdete vždy 4 čísla a_x, a_y, b_x, b_y udávající souřadnice levého dolního a pravého horního rohu daného hustolesa. Všechny souřadnice jsou zadány v metrech.

Formát výstupu: Na výstup vypište počet sekund, za které stopař překoná celou trasu, zaokrouhlený na celé sekundy.

Ukázkový vstup:

```
10 5 2
100 100
350 350
500 500
1100 500
1100 100
100 200 1300 300
400 400 600 600
```

Ukázkový výstup:

```
205
```

Stopař celkově uběhne cca 1566 m, z toho cca 483 m hustolesem.

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

Po setkání se stopařem se Gorf vydal zařizovat něco do města, možná se zkontaktovat s místním cechem zlodějů, což byl také velmi dobrý zdroj informací. Zbytek družiny se usadil v krčmě a nad džbánkem probíral, co dál.

Byli sem vysláni, aby zjistili více o záhadném nebezpečí ze severu. V království už nějakou dobu kolovaly historky, že na severu se sbíhají nějaké temné síly, ale zatím ani král, ani řád neměli žádné důkazy. Jen spoustu mlhavých příběhů od obchodníků.

Živého draka nikdo neviděl po několik set let. Většinu z nich pobili za dávných dob drakobijci, a pokud nějací draci přežili, tak se myslelo, že spí nekonečným spánkem ve svých hlubokých slujích. Tenhle jeden ale rozhodně nevypadal na to, že by byl vyhuben, ani na to, že by spal nekonečným spánkem. Musela ho probudit nějaká veliká temná síla.

Warin se zamyslel nad jejich nynější situací a přitom se přes svůj džbánek zahleděl na druhou stranu stolu. S úsměvem si všiml, jak Lian pokukuje po Rhee. Rheu znal už spoustu let a vždycky ji vnímal jen jako dobrou přítelkyni. Ale musel uznat, že je to okouzlující kouzelnice a vůbec se nedivil, že mladý rytíř z ní může mít hlavu v oblacích. Rhea samotná si toho pravděpodobně byla vědoma, ale vypadalo to, že jí to vůbec nevadí. Jen kdyby ji Lian pořád neoslovoval „Mylady“ ...

Tok Warinových myšlenek přerušil Gorf, který rozrazil dveře do krčmy, doběhl k jejich stolu a polohlasem pronesl „Warine, Rheo, Liane, na severní hradby útočí drak!“

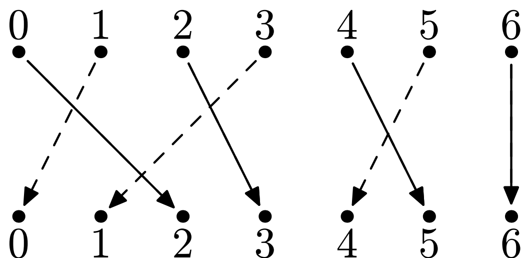
Warin se nerozmýšlel moc dlouho a poslal Gorfa s Rheou přímo na hradby, aby pomohli obráncům. On společně s Lianem se zatím rozeběhli ke stájím, kde měli svého nákladního mezka, aby se převlékli do brnění. Byli sice jen čtyři, ale všichni byli zatraceně dobří bojovníci.

Gorf s Rheou doběhli na hradby právě ve chvíli, kdy se nad nimi přehnal drak, spálil jeden dům těsně za hradbami a začal se otáčet k dalšímu kolečku. Proti němu vylétlo sporadicky několik šípů, ale bylo jich málo a navíc se nezkušení lukostřelci ohrožovali spíše navzájem, než aby trefovali draka. Gorf stáhl ze zad svůj luk a začal je organizovat.

29-1-5 Lučištníci — 10 bodů

Lučištníci stojící v řadě na hradbě mají vyhlédnuté své cíle. Každý z nich míří lukem na nějaké místo, ale je neúčinné a nebezpečné, aby všichni stříleli, jak se jim zachce. Rádi by svou palbu koncentrovali a navíc by měli střílet jen ti, jejichž dráhy palby se nekříží.

Lučištníky můžeme popsat pomocí bodů, na které míří. Například na obrázku níže nultý lučištník míří na bod 2, první na bod 0, druhý na bod 3, třetí na bod 1, čtvrtý na bod 5, pátý na bod 4 a šestý na bod 6.



Ze všech lučištníků chceme vybrat co největší skupinu, jejíž dráhy střelby se nekříží. V uvedeném příkladě jsou to třeba lučištníci 0, 2, 4 a 6 (zvýraznění plnými čarami). Stejně tak by fungovali třeba lučištníci 1, 2, 5 a 6.

Navrhněte algoritmus, který takovou skupinu najde. Pokud existuje více řešení, stačí oznámit libovolné jedno z nich.

Ve chvíli, kdy na hradby doběhli Warin s Lianem v těžké kroužkové zbroji s bílomodrými štíty, už létaly šípy v mohutných salvách. Drak se raději držel dál, ale z lesa naproti městské bráně začali vybíhat goblini.

„Kdo k sakru jste?“ vykřikl překvapeně kapitán městské stráže, když se k němu Warin a Lian rozeběhli. „Rytíři Alvarezova řádu, pokud si chcete zachránit město, poslouchejte nás!“

Protože oba ozbrojenci vypadali, že jsou mnohem bojeschopnější, než kdokoliv z jeho mužů, nemluvě o těžké výstroji, kterou nesli, tak kapitán bez větších námitek souhlasil. Goblini začali dorážet na hradby pomocí žebříků a část z nich se pokoušela i o proražení brány.

Lian si vzal na starost obranu vršku hradeb a Warin začal šikovat gardisty dole pod hradbami, aby se připravili na prolomení brány.

Drak se ale nevzdával. Jakoby ho modrobílé postavičky obou rytířů vyburcovaly k ještě větší bojechtivosti, začal se střemhlavě vrhat na hradby a svým mocným dechem je začal spalovat tak, až pukal kámen.

Rhea se ale také činila. Připravovala si štítové kouzlo a nyní ho začala rozprostírat.

29-1-6 Štítové kouzlo 10 bodů

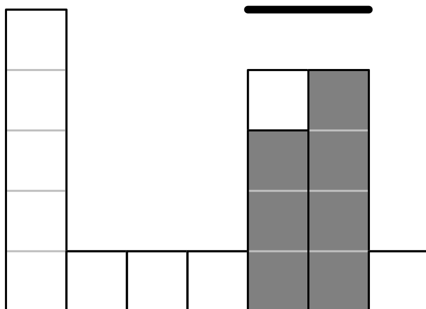
Na městské hradby útočí drak a postupně je ničí. Městské hradby si můžeme představit jako řadu různě vysokých kusů. Drak při každém svém náletu ubere všem nechráněným kusům hradeb jednu jednotku výšky. Níže, než úplně do základů, je ale spálit nemůže (výška hradeb nemůže jít do záporných čísel).

Hradby můžeme chránit pomocí štítového kouzla, ale kouzelnice Rhea umí roztahovat štít vždy jen o jeden kus hradeb mezi každými dvěma drakovy útoky. Navíc chráněná oblast hradeb musí být souvislá.

Začneme tedy tvořit štít nad libovolným kusem hradeb, který je od této chvíle chráněn a již se neničí. V každém dalším kroku ho můžeme rozšířit o jedna doleva, nebo doprava.

Vaším cílem je pro zadané výšky hradeb poradit kouzelnici Rhee, kde má začít a jak má štít postupně rozšiřovat, aby se v součtu uchránilo co nejvíce hradeb (tedy aby součet zbylých výšek hradeb byl co největší).

Příklad: Pro hradby vysoké $5, 1, 1, 1, 4, 4, 1$ je nejlepší začít se štítem nad předposledním úsekem hradeb a pak ho rozšiřovat směrem doleva. Tím se nám povede uchránit celkově 7 dílů hradeb. Kdybychom začali na páté pozici a rozšiřovali doprava, uchráníme stejně dílů hradeb, kdežto kdybychom začali se štítem nad nejvyšší hradbou, tak se nám povede uchránit nejvýše 5 dílů hradeb – vysoké hradby na opačné straně padnou dříve, než nad ně stihneme rozšířit štít.



Na obrázku jsou znázorněny výšky hradeb a části, které z nich zůstanou, pokud začneme stavět štít nad předposlední hradbou a roztáhneme ho doleva (další stavění štítu pak už nic jiného nezachrání).

Jeden z posledních drakových útoků vedl na část hradeb, kde Lian odrážel útočící gobliny. Těsně předtím, než se i zde zhmotnil štít, pronikl drakův útok skrz a smetl všechny obránce společně s hromadou sutin dolů. Rhea se vyděšeně ohlédla, Liana však nikde neviděla. Nesměla ale polevit v posilování štítu, na zachraňování bude čas později!

Městskou bránu mezitím prorazili goblini, ale ani ve snu nebyli připraveni na modrobílý uragán, který se mezi ně vrhl. Warin rozdával rány na všechny strany, městští gardisté se sice také snažili, ale úrovně cvičeného alvarezského rytíře dosáhnout nemohli.

Drak už mezitím utrpěl příliš mnoho zranění od šípů a viděl, že jeho útoky jsou odráženy štítem, a tak s mocným zařváním svůj pokus vypálit město vzdal a dal se na ústup. Ve spojení s krvavou lázní u městské brány to na gobliny bylo asi moc. Většina z nich zpanikařila a začala utíkat nazpět do lesa. Po necelých deseti minutách už na dohled od brány nezůstal jediný živý goblin.

Když se Warin vrátil s gardisty zpět za městskou bránu, uviděl, jak Rhea a Gorf usilovně odhazují trámy a kusy zdiva u jedné zřícené části hradeb, Rhea měla slzy v očích. Hned mu bylo jasné, co se stalo. Odložil zbraně a dal se také do odhrabování. Vyprostili několik živých a bohužel i několik mrtvých vojáků, když tu pod sutinami zahlédli modrou a bílou. Po odhození pár trámů Liana konečně mohli vytáhnout. Nehýbal se.

Pak se náhle ostře nadechl, otevřel oči a rozkašlal. Rhea ho beze slova sevřela do náruče tak silně, až mu málem vymáčkla dech. Zase byli všichni čtyři pohromadě a živí a čekala je tady na severu určitě ještě spousta dobrodružství. Třeba o nich ještě uslyšíme . . .

Příběh pro vás vyprávěl

Jirka Setnička

Druhá série

„To ten den skvěle začíná," prolétlo Erice hlavou, když chvatně přesouvala svůj notebook z kolejního stolu do batohu. Hodiny na zdi navzdory výhružným pohledům ukazovaly patnáct minut do začátku výuky, která ovšem probíhala přes půl hodiny odtud.

Budík ji měl vzbudit už o půl osmé, ale když pak na chvilku zavřela oči, musela usnout, protože najednou bylo skoro devět. Konečně měla vše potřebné a mohla vyběhnout. Napadlo ji, kolik vlastně existuje stejně rychlých cest do školy, ale raději si zakázala experimentovat.

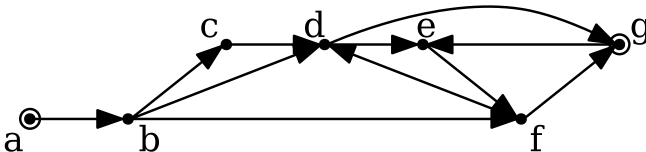
29-2-1 Cesty do školy 10 bodů

Studentka se chce dostat do školy za právě K minut – ne více, ale ani ne méně (to by musela zbytečně čekat na chodbě). Zajímalo by ji, kolika různými způsoby to jde udělat. Protože venku už začíná být docela zima, nechce se ani po cestě nikde zastavovat. Raději celých K minut stráví chůzí, i kdyby to znamenalo trochu si zajít, nebo třeba jít kus tam a zpátky.

K dispozici má mapu, ve které je zakresleno N význačných míst – kromě startovního a cílového bodu (koleje a školy) také různé křižovatky a další místa, přes která je možné procházet. Dále mapa obsahuje seznam povolených přesunů: každá jeho položka říká, že z místa i jde dojít na jiné místo j, a to za přesně jednu minutu. Jinudy než podle povolených přesunů se pohybovat nelze. Pozor, u přesunů záleží na směru. Může se stát, že je povoleno jít z i do j, ale ne z j do i (třeba protože v opačném směru je to moc do kopce).

Formálněji: je dán orientovaný graf. Jeho vrcholy odpovídají místům a hrany povoleným přesunům. Zajímá nás, kolik v něm existuje různých sledů mezi danými dvěma vrcholy s a t dlouhých přesně K hran. Sled je něco jako cesta, jen se na něm mohou opakovat vrcholy a hrany.

Uvažujme například následující mapu a $K = 6$, $s = a$, $t = g$:



V ní existuje právě pět sledů délky 6 vedoucích z a do g. Jsou to *abcdefg*, *abfgefg*, *abdgefg*, *abdefdg*, *abfdefg*.

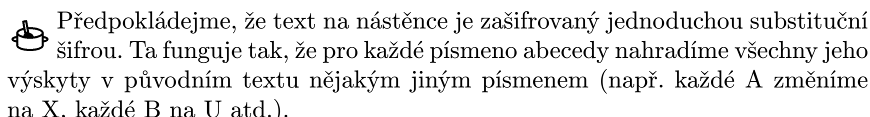
Obvyklý způsob dopravy dnes zafungoval. Erice se povedlo chytit autobus, tramvaj (přes hlavy lidí viděla přiskakující minuty) i další tramvaj a krátce před půl už přebíhala Malostranské náměstí, až skoro smetla nějakou podobně starou

dívku. Schody vzala po více najednou, přeběhla celou chodbu a zkusila se dobýt do učebny — která se ovšem tvářila tiše, a hlavně zamčeně.

Erika hned vytáhla mobil a na Hangoutu napsala svému spolužákovi: `Ahoj, prosím Tě, analýza se někam přesunula?` *Vzápětí zůstala nevěřícně koukat na čas 8:28 vedle své zprávy. Odpověď přišla záhy.* `Ne ale zacina prece v 9.` *A pak přišla další zpráva:* `Vis ze je zimni cas?`

Jen malým zázrakem nedošlo k násilí na nevinném telefonu. Místo toho se Erika vydala zkoumat nástěnky na chodbách. Na jednu někdo vylepil papír se zašifrovaným textem a výzvou k rozluštění. Spíš než opravdové řešení teď ale Erika toužila najít něco jako „pomsta vynálezci letního času“.

29-2-2 Hledání pomsty 13 bodů



Předpokládejme, že text na nástěnce je zašifrovaný jednoduchou substituční šifrou. Ta funguje tak, že pro každé písmeno abecedy nahradíme všechny jeho výskyty v původním textu nějakým jiným písmenem (např. každé A změníme na X, každé B na U atd.).

Navíc platí, že dvě různá písmena nikdy nenahradíme stejným, protože pak by se text nedal jednoznačně dešifrovat. Předpisu, které písmenko nahrazujeme kterým, se říká *klíč*.

Korektními způsoby, jak zašifrovat slovo `PAPIR`, jsou např. `UWUXI` nebo `PEPZN`, ale nikoli `SDFGH` (každé P jsme nahradili za něco jiného) nebo naopak `CLCKC` (P i R jsme nahradili stejným písmenem).

Dostanete zašifrovaný text (neznámým klíčem) a hledaný řetězec (nezašifrovaný). Vaším úkolem je najít všechny pozice, na kterých se v původním textu mohl zadaný řetězec vyskytovat. Různé výskyty mohou předpokládat různé klíče.

Například slovo `POTOPA` v textu `ZAGHAHGZGHLQWUW` můžeme najít na dvou místech:

```
  POTOPA
ZAGHAHGZGHLQWUW
      POTOPA
```

V prvním případě klíč překládá P→G, O→H, T→A, A→Z. Ve druhém je správné přiřazení P→H, O→G, T→Z, A→L.

Snadno si rozmyslíte, že jinde už se toto slovo vyskytovat nemůže.

Po přednášce, která proběhla překvapivě v klidu (jen jeden nejasný důkaz a jen dvě rýpnutí od spolužáků, kteří změnu času zaregistrovali), čekalo Eriku ještě programovací cvičení. Obvykle ho měla ráda, ale dnes se jí povedlo jedním středníkem navíc vyrobit nekonečný cyklus. Přitom na původ chyby přišla až po hodině, takže se dnes ze školy vyloženě těšila.

Zvlášť když si na dnešek naplánovaly sraz s kamarádkou ze střední! Potkat se na zastávce Karlovo náměstí znělo jako skvělý nápad, dokud Erika nezjistila,

že těch zastávek je více kus od sebe. Zavolat kamarádce znělo jako skvělý nápad, dokud telefon suše neoznámil, že „volaný účastník není dostupný“.

A tak Erika několikrát přebíhala mezi jednotlivými zastávkami. Přitom si všimla, že tu stále visí nejrůznější volební reklamy.

29-2-3 Billboardová většina — 13 bodů

Erika probíhá ulicemi, kde visí spousta volební reklamy: billboardy, plakáty, … Víme, v jakém pořadí okolo reklamních materiálů proběhla a které strany propagovaly.

Rádi bychom uměli pro libovolnou část její cesty zjistit, jestli v tomto úseku měla nějaká strana nadpoloviční většinu reklamních materiálů (a tedy by přesvědčila lidi, kteří projdou jen tuto část cesty).

Na vstupu dostanete nejdřív posloupnost N přirozených čísel $(a_0, \ldots, a_{N-1})$. Ta udává, které strany propagují jednotlivé plakáty, v pořadí, v jakém je Erika míjela (tedy a_i je číslo strany propagované i-tou reklamou). Čísla stran můžou být libovolně velká.

Poté bude následovat Q dotazů. Každý dotaz je tvořen dvojicí čísel (k, ℓ). Úkolem vašeho algoritmu je pro každý dotaz určit, zda v úseku $a_k, a_{k+1}, \ldots, a_{\ell-1}$ posloupnosti má nějaké číslo strany nadpoloviční zastoupení.

Například pro posloupnost

a_0	a_1	a_2	a_3	a_4	a_5	a_6	a_7	a_8	a_9
1	4	4	7	4	1	4	1	1	7

a dotazy $(1,6), (3,6), (2,5), (5,10), (0,10)$ jsou správnými odpověďmi $4, -, 4, 1, -$(kde $-$ značí, že v daném úseku nemá většinu nikdo). První úsek $(1,6)$ je výše znázorněn podtržením.

Vyhodnocovat každý dotaz zvlášť by bylo pomalé. Zkuste si na začátku pro posloupnost něco předpočítat, abyste pak zvládli dotazy vyřizovat rychleji. Předpokládejte, že počet dotazů bude řádově srovnatelný s N.

Při třetím návratu na původní zastávku se ovšem zadařilo a obě dívky se konečně potkaly. V blízké kavárně pak nadšeně propovídaly dvě hodiny jako nic. Při loučení se si slíbily, že se zase brzy uvidí, a pak už každá vyrazila za svým dalším programem.

V Eričině případě to znamenalo vrátit se na kolej a sbalit si vše, co by mohla potřebovat na hodině powerjógy. K jejímu provozování se nechala přesvědčit už na začátku semestru Hankou z koleje, která nechtěla chodit sama. Když Erika dorazila na sraz s Hankou, bylo už dost pozdě. Přesto se Hanka nejvíc ze všeho tvářila zmateně.

„Máš určitě všechno?“ zeptala se nejistě. „No jasně,“ mávla Erika rukou… ve které, jak si právě uvědomila, neměla sportovní tašku. „Eh, ne, počkej, hned jsem zpátky!“

Hanka měla z Eriky náramnou legraci a dobírala si ji i po lekci, kdy se Erika chystala vydat za dalšími kamarády. „Nemám Tě raději doprovodit na místo?“ ptala se. „Huš,“ zkontrolovala Erika na mobilu jízdní řády, „za chvilku mi jede autobus a přestoupit na metro zvládnu.“

Dívky se rozloučily a Erika za chvíli skutečně nastoupila do autobusu. Když ale ani po pěti zastávkách nebyla v dočasném cíli, znejistěla a na další zastávce raději vystoupila. Zaujalo ji náměstíčko protkané chodníčky. Mezi nimi se nacházely květinové záhony podivuhodně nepravidelných tvarů. Sedmiúhelníkový záhon, kdo to kdy viděl!

29-2-4 Nejsložitější záhon — 9 bodů

Na náměstíčku tvaru N-úhelníku jsou různé chodníčky, které jsou ale vedené tak, že se navzájem nekříží a vycházejí jen z pomyslných rohů, nikoliv ze samotných stran.

Oblasti mezi chodníčky tvoří květinové záhony. Nás by zajímalo, který záhon má nejsložitější tvar, tedy je tvořen mnohoúhelníkem s nejvíce stranami.

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

Formát vstupu: Na prvním řádku vstupu dostanete čísla N a K udávající počet vrcholů N-úhelníka a počet chodníčků. Na dalších K řádcích následuje popis chodníčků – každý chodníček je určen dvěma čísly udávajícími, mezi kterými dvěma vrcholy náměstíčka vede. Vrcholy číslujeme od 0 do $N-1$ v pořadí na obvodu.

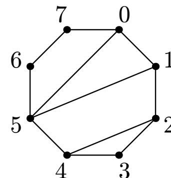
Formát výstupu: Na výstup na prvním řádku počet vrcholů na obvodu největší souvislé oblasti (záhonu) a na druhém řádku mezerou oddělená čísla *zajímavých* vrcholů ohraničujících tuto oblast – to jsou takové vrcholy, kde přecházíme z jednoho chodníčku na druhý. Čísla vypište v rostoucím pořadí, v jakém se vyskytují na obvodu této oblasti. Pokud existuje více takových oblastí, vyberte libovolnou.

Ukázkový vstup:

```
8 3
5 1
0 5
2 4
```

Ukázkový výstup:

```
4
0 5
```



Nejsložitější záhon má tvar čtyřúhelníku – můžeme si vybrat $(1, 2, 4, 5)$ nebo $(0, 5, 6, 7)$. Na prvním jsou významné všechny body, na druhém jen body 0 a 5.

Erika si ovšem záhy vzpomněla, že její hlavní starostí je něco jiného. Pohled na ceduli u zastávky jí prozradil, že zřejmě vystoupila nikoliv z autobusu 136, nýbrž z autobusu 135. Potlačila zanadávání na plus minus jedničky, našla si nový spoj a po další tři čtvrtě hodině úspěšně dorazila do čajovny určení.

„Jéje, Erika nám to určitě pokazí,“ uvítali ji se smíchem. „Co, co?“ „Petr bude příští semestr ve Švédsku, tak už si plánujeme, kdy se kdo z nás pozve na návštěvu,“ vysvětlili jí kamarádi vesele. A všichni se znovu sklonili nad poznámkami.

29-2-5 Plánování návštěv — 10 bodů

Petr stráví příštích N týdnů ve Švédsku. Každý z jeho K kamarádů by rád přijel na návštěvu. Každý víkend může Petr ubytovat právě jednoho svého kamaráda. Zároveň se kvůli různým již naplánovaným akcím každému z kamarádů hodí právě 2 víkendy.

O každém z kamarádů se dozvíte, které 2 víkendy by se mu pro návštěvu hodily. Rozhodněte, zda se mohou u Petra vystřídat všichni, nebo zda bude muset Petr některé odmítnout.

Můžete předpokládat, že $K \leq N$.

Od vzrušeného dohadování se nad diáři se celá společnost postupně přesunula k mnoha dalším tématům. Ovšem čas nechtěl brát ohled na jejich veselí, ani se nenechal ukolébat klidem zbytku čajovny, poskakoval a postupně přinesl únavu.

Zrovna když část lidí řešila, jak si pomocí obyčejné mince vybrat ze tří čajů, zvedla se Erika k odchodu. Navzdory historkám celého dne vyrazila sama a bezpečně došla zpět na metro. Za jedinou komplikaci by mohla považovat zhasínající lampu, ale už tu párkrát šla a zrovna tahle lampa zhasínala pokaždé, když procházela okolo.

Na Muzeu přestupovala v zamyšlení, proplétala se mezi pár cestujícími. Najednou k ní dolehlo pobavené zavolání: „Tak Markovci udrželi Knot zas jenom dva dny!“

Erika se překvapeně rozhlédla. Z ostatních cestujících tu mezitím zůstal jen nějaký muž a dívka tak v jejím věku, která teď obarvila dvě políčka v tabulce na zdi. Čmárání na zeď v metru? Erika se zarazila. A lehce sebou trhla, když si nad tabulkou všimla nápisu „Linka α“.

„Ale koukám, že se jim i tak velmi daří,“ prohlásil muž.

Dívka zavrtěla hlavou. „Tak to působí kvůli tomu, jak je to nakreslené.“

29-2-6 Souvislá plocha — 11 bodů

Několik skupin lidí se přetahuje o určitý předmět. Do tabulky o R řádcích a S sloupcích barvami vyznačujeme, kdo ho vlastnil který den. Zakreslujeme po řádcích, když dojdeme na konec řádku, pokračujeme na dalším.

Informace k nám ale chodí, jen když se majitel změní. Dostaneme tak třeba informaci, že první skupina měla předmět 5 dní, druhá skupina 2 dny, první skupina 3 dny, ...

Nejúspěšněji působí skupina, jejíž barva je nejvíc vidět, a nejvíc vidět je souvislá oblast. Oblast je souvislá, když jsou v ní všechna políčka sousední, tj. sdílí hranu (nestačí sousedit rohem). Nás zajímá zejména to, která skupina má největší souvislou oblast.

Předpokládejte, že skupin je málo, maximálně 200. Naopak tabulka může být obrovská, počítejte s tím, že se vám nemusí vejít do paměti. Ale jeden řádek se do paměti určitě vejde.

Na vstupu dostanete čísla R, S, K, Z, popisující rozměry tabulky, počet změn skupin a počet změn vlastnictví (včetně prvotního získání). Na dalších Z řádcích jsou popsané jednotlivé změny vlastnictví — vždy která skupina předmět získala a na jak dlouho. Na výstup vypište číslo skupiny, které patří největší souvislá oblast.

Ukázkový vstup:

```
3 10 2
0 2
1 2
0 2
1 2
0 4
1 6
0 12
```

Ukázkový výstup:

```
0
```

Tabulka z příkladu vypadá následovně:

■	■	□	□	■	■	□	□	■	■
■	■	□	□	□	□	□	□	■	■
■	■	■	■	■	■	■	■	■	■

Největší souvislou oblast má skupina 0 (na obrázku šedá).

„Ale není to jediné, co působí jinak, než to opravdu je,“ pokračovala dívka. „Věříš tomu, že ta holka má zvláštní moc.“

„Jo! Do háje, vždyť má svou lampu na Starým městě,“ vyhrkla dívka rozčileně. Vzápětí se uklidnila a pokračovala: „Vážně, dneska jsem se s ní velmi zblízka potkala na Malostranském náměstí a je to z ní cítit.

Vlastně jsem se chtěla porozhlédnout i na jejím pokoji na koleji, ale přestože jsem se po budově potloukala, dokud neměla být pryč, jakmile jsem zamířila k jejímu pokoji, proběhla okolo mě, přímo k tomu pokoji. Takže jsem to radši vzdala. Ale když jsem si pak úplně jinde znova procházela, co o ní víme, najednou stála přede mnou. Koukla po náměstí, hodila po mně výhružný pohled, pak ještě koukla na zastávku, něco si poznamela a zmizela. Nepřišlo by ti to zvláštní?

A vůbec,“ podívala se dívka přímo na Eriku a její tón se změnil na pobavený, „kdyby neměla naši moc, jak by se sem jen tak dostala?“

Erika lehce lapla po dechu. Muž chvíli nechápavě stál, pak se najednou otočil na Eriku. Několikrát přejel pohledem mezi oběma dívkami. Mírně se usmál. Nakonec prohlásil: „To zní ovšem jako úplně jiný příběh…“

… který už s vámi nemohla sledovat

Karry Burešová

Třetí série

Naše hrdiny jsme v první sérii opustili potom, co pomohli zachránit jedno severské město před útokem draka a hordy goblinů. Tajemná síla ze severu se však nenechala odradit, a tak se za nimi vrátíme k branám města Leyfast a budeme jejich osudy sledovat dál.

„Sire Warine!“ přivítal celou skupinu starosta Leyfastu. „Děkuji za přivítání, dovolte mi představit zbytek mé skupinky – člena Alvarezova řádu rytíře Liana, mocnou kouzelnici Rheu a jednoho z nejschopnějších lučištníků, co znám, Gorfa.“

„Město vám všem děkuje za vaše služby... ale povězte, co máme dělat teď?“

Následující půlhodinu Warin se starostou probírali, jak by mohli posílit vojenskou posádku Leyfastu a současně tady na severu zřídit alespoň nějakou bojovou sílu. Silných mužů bylo v okolních usedlostech hodně, ale naverbovat je všechny nemohli, nebylo by pak lovců, kovářů a jiných nezbytných profesí. Ještě, že v Leyfastu měli vedené velmi přesné záznamy o okolních obyvatelích.

29-3-1 Verbování — 8 bodů

Město Leyfast potřebuje co nejvíce posílit svoji armádu, ale současně nemůže sebrat každého bojeschopného muže z okolí. Starosta města vyslal skupinu verbířů, která má za úkol obejít okolní usedlosti a vrátit se s co nejvíce bojeschopnou skupinou mužů.

Verbíři budou procházet domy v předem daném pořadí a díky pečlivým záznamům vědí, kdo v jakém domě bydlí. Dokonce pro každý dům vědí, jak silný muž v něm bydlí a jaké mají doma zbraně.

Pro i-tý dům se mohou verbíři rozhodnout, jestli jeho obyvatele nechají být (což bojeschopnost armády nijak neovlivní), jestli z něj naverbují nového brance (což přispěje bojeschopnosti armády číslem V_i), nebo jestli si pro vyzbrojení nějakého brance vezmou od obyvatel zbroj a zbraně (což přispěje bojeschopnosti armády číslem Z_i).

Zbroj a zbraně si verbíři můžou vzít pouze tehdy, pokud z minulého domu naverbovali nějakého brance (obyvatelé jsou ochotni dát své věci jen nejbližším sousedům). Verbíři také nikdy neudělají to, že by z jednoho domu současně odvedli brance a odnesli zbraně. Kromě toho také verbíři nikdy neodvedou brance z dvou domů těsně po sobě.

Pokud si tedy budeme v posloupnosti značit jako `V` verbování, jako `Z` sebrání zbraní a jako `-` žádnou akci, tak:

- `-V-VV-` je špatně: obsahuje dvě verbování po sobě.
- `-V-Z-` je také špatně: obsahuje braní zbraní, kterému těsně nepředcházelo verbování.
- `-VZVZV-V-` je správně.

Verbíři by při dodržení pravidel uvedených výše chtěli zvýšit bojeschopnost armády, co nejvíce to půjde.

Formát vstupu: Na prvním řádku vstupu dostanete číslo N udávající počet domů, které plánují verbíři obejít. Na druhém řádku se bude nacházet N čísel V_1 až V_N oddělených mezerou udávajících „zisky bojeschopnosti“ při provedení verbování v jednotlivých domech, na třetím řádku pak obdobně naleznete čísla Z_1 až Z_N udávajících to samé, ale pro braní zbraní z jednotlivých domů. Všechna V_i i Z_i budou nezáporná celá čísla.

Formát výstupu: Na první řádek výstupu vypište maximální zisk bojeschopnosti, který je možný dosáhnout, a na druhý řádek pak vypište N znaků `V`, `Z` nebo `-` (neoddělujte je mezerami) udávajících plán verbování pro jednotlivé domy. Pokud je více možností, jak dosáhnout stejného zisku, můžete si vybrat libovolnou z nich.

Ukázkový vstup:

```
10
5 2 1 3 4 6 3 1 6 7
2 1 3 2 3 5 1 1 2 1
```

Ukázkový výstup:

```
29
VZ-VZVZVZV
```

Dalším způsobem, jak dosáhnout stejného zisku bojeschopnosti, pak jsou `VZVZVZVZ-V` a `VZVZVZVZVZ`, jiné způsoby nejsou.

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

Když se postarali o to, že v okolí Leyfastu vznikne účinná bojová síla, nabrali dobrodruzi zásoby jídla a podle rad místních stopařů vyrazili směrem do horského průsmyku. Jejich cílem bylo najít draka a dozvědět se co možná nejvíc o tajemné síle, která za tím vším stojí.

V horském průsmyku sice sídlila horda goblinů a podle všeho se tu objevovali i trollové, ale stopaři jim prozradili, že průsmyk podchází starý trpasličí důl. Existovala sice i bezpečnější cesta okolo hor na druhou stranu, kde by gobliny asi nepotkali, ale ta by jim zabrala přes dva týdny. Rozhodli se tedy vydat se do trpasličího dolu.

Přesně podle rad stopařů nalezli zpola zasypaný vchod a vnikli dovnitř. Ušli ve světle mihotavé hvězdy vznášející se Rhee nad rukou pořádný kus cesty, až dospěli k důlnímu výtahu. Důl byl opuštěný sotva padesát let, což je pro trpasličí techniku krátký čas. Výtah skoro fungoval, jen ho bylo potřeba vyvážit.

29-3-2 Trpasličí závaží — 10 bodů

Dobrodruzi stojící před starým trpasličím důlním výtahem by potřebovali tento výtah vyvážit. K tomu by potřebovali umět rychle porovnávat váhu zátěže.

Závaží, kterými se vyvažuje trpasličí důlní výtah, mají váhy $1, 2, 4, 8, \ldots, 2^N$ a každé z nich jde umístit jako zátěž nebo jako protizátěž. Pokud si umístění závaží budeme značit 1 pro zátěž a -1 pro protizátěž a budeme zapisovat všechna závaží od největšího až k závaží o váze 1, bude zápis $-1, 0, 0, 1, -1, 0$ znamenat celkovou zátěž $(-1) \cdot 32 + 0 \cdot 16 + 0 \cdot 8 + 1 \cdot 4 + (-1) \cdot 2 + 0 \cdot 1 = -30$.

Můžeme si všimnout, že stejné zátěže lze dosáhnout i jiným poskládáním závaží, například $-1, 0, 0, 0, 1, 0$. Porovnávání zátěží proto asi nebude úplně jednoduchý úkol. Vymyslete postup, jak v co nejkratším čase pro dva předpisy umístění závaží (zadané na vstupu jako takováto čísla v podivné dvojkové soustavě) určit, který z nich značí větší zátěž.

Předpokládejte, že celková zátěž bude tak velká, že se nevejde do běžné celočíselné proměnné a není tak možné oba předpisy převést a pak porovnat – je potřeba je porovnávat bez převodu (ale upravovat si zápis z -1, 0 a 1 lze).

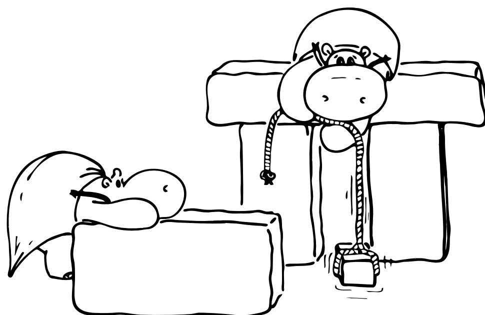
Zkusili několik sad závaží a po vyvážení se výtah konečně rozjel. Trpasličí ozubená kola se sice párkrát zadrhla, ale poté, co se z nich obrousila rez, už je výtah lehce dovezl několik set metrů do hloubky.

Cesta skrz zbytek dolu byla dlouhá a museli se párkrát vracet, ale nakonec, potom co cestou i přespali, zahlédli na konci jedné úzké chodbičky denní světlo. Dostali se na malou římsu, kde úzký vchod do štoly zakrýval okolní porost. Pod nimi se jim naskytl pohled na velké, narychlo zbudované ležení skřetí tlupy.

Nebylo to příliš mnoho skřetů, ale zároveň jich ani nebylo málo. A vypadalo to, že je v jejich táboře docela ruch.

„Poznáš, jaký je to klan?“ zeptala se Rhea Warina. Ten se dlouze zadíval a pak odpověděl: „Těžko říct, budou někde zdaleka a na tuhle dálku nevidím pořádně jejich klanové barvy. Spíše se dá soudit podle toho, jak vypadá jejich tábor. Liane. . . “, zavolal si pak mladého rytíře.

„Vidíš ty jejich věže? Pokud jsou to skřeti z Kolibu, tak budou pravidelné, ti jsou prý posedlí symetrií.“

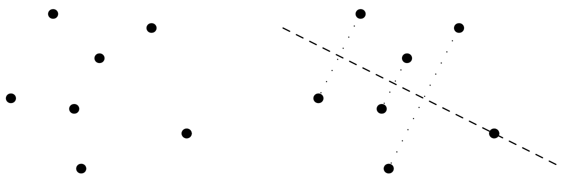


29-3-3 Skřetí věže 11 bodů

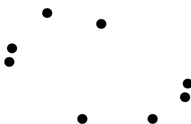
Průzkumníci se dostali nad skřetí ležení a zvláště je zaujaly jejich hlídkové věže. Vypadaly nezvykle symetricky, ale chtěli by ověřit, že jsou skutečně symetrické.

Věže by měly být symetrické podle nějaké osy a průzkumníci by tuto osu chtěli nalézt. Pro zadané souřadnice věží nalezněte osu, podle které jsou věže symetrické (případně rozhodněte, že žádná osa symetrie neexistuje).

Pozor na to, že bod může být symetrický i sám k sobě, pokud bude ležet přímo na ose symetrie. Například pro první příklad symetrii nalezneme, i když má lichý počet bodů:



Pro druhý příklad už ale symetrie neexistuje:



„Tak jsou to skřeti z Kolibu, zajímavé…“ zamyslel se Warin. Co tady jen dělají, pomyslel si. Od Kolibu to byla cesta na mnoho týdnů a někdo nebo něco je sem muselo povolat. Otázkou je, kdo nebo co to bylo.

„Dračí sluj!“ hlesl náhle Gorf polohlasem, když svým bystrým zrakem zahlédl na samé hranici dohledu, daleko za skřetím ležením, velkou opálenou díru do skály.

Teď už bylo jasné, kam se vydají dál. Pokud mají přijít na to, co se zde děje, je dračí sluj rozhodně zajímavým místem, kde zahájit průzkum. Pokud nějaká síla zvládla povolat sem na sever skřety a probudit i draka, tak tam po ní snad naleznou nějaké stopy.

Problém byl, že mezi nimi a skalní slují se nacházelo jednak skřetí ležení a za ním pak ještě bažina. Skrz bažinu sice vedly nějaké světlé proužky, asi cesty vyskládané z dřevěných hatí, ale na dálku to šlo jen těžko poznat. Každopádně skřeti byli první překážkou.

Přes ty skřety se ve dne dostat nezvládnou, tak se utábořili a připravili se na to, že v noci zkusí proklouznout. Blýskavá brnění zakryla černá látka a ujistili se také, že mají všechnu výzbroj pořádně připevněnou a že jim nebude nic cinkat.

Pak vyrazili s cílem proklouznout okolo skřetích hlídek posazených u strážních ohňů.

29-3-4 Mezi hlídkami 11 bodů

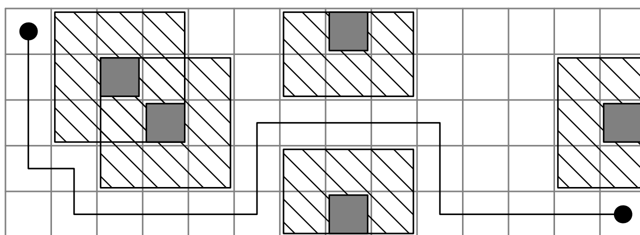
Skupina bojovníků potřebuje v noci proklouznout okolo skřetích hlídek. Hlídky jsou nehybné, sedí okolo strážních ohňů a nevšimnou si osamoceného bojovníka, pokud neprojde přímo okolo nich. Skupina se tedy chce rozdělit a každý z nich se pokusí projít osamoceně, aby byl tišší.

Pláň, na které jsou posazeny hlídky, si můžeme představit jako velikou čtvercovou síť (o velikosti $N \times M$) a hlídky jsou posazené na některých políčcích. Hlídek je řádově méně, než je počet políček pláně, a jejich pozice se mezi průchody jednotlivých bojovníků nemění.

Vymyslete datovou strukturu, kterou si v nějakém rozumném čase předpočítáte a pak pomocí ní zvládnete rychle plánovat nejkratší bezpečné cesty (vzdálené alespoň jedno políčko od jakékoliv hlídky) pro jednotlivé bojovníky.

Každý bojovník se bude chtít dostat mezi nějakou zadanou dvojicí bodů a pro plánování je potřeba umět v čase $\mathcal{O}(1)$ zjistit, jaká je nejkratší vzdálenost mezi touto dvojicí bodů (ale již není potřeba vypsat trasu této cesty, jde jen o délku). Vymyslete datovou strukturu, která toto umí zajistit a která zároveň předvýpočtem stráví co nejméně času. Všechny časové složitosti vyjadřujte nejen vzhledem k velikosti pláně, ale i k počtu hlídek K. A pamatujte, že hlídek je výrazně méně, než je velikost pláně.

Na obrázku můžete vidět ukázku nejkratší cesty mezi dvěma vyznačenými body, správná odpověď by tak v tomto případě byla 21:



Na opačné straně skřetího ležení se opět všichni čtyři shromáždili a vyrazili dál. Překonání bažiny po hatích už bylo celkem snadné, i když na jednom místě narazili na podivný obrazec, kde byly položené dlouhé dřevěné tyče pomalované podivnou světélkující barvou a seskládané do jakéhosi obrazce. Nevěnovali jim ale příliš pozornosti a pokračovali k dračí sluji.

Po pár minutách k ní dorazili. Všude byl klid a tak vešli opatrně dovnitř.

Zadání úloh KSP – 3. série

Vevnitř to páchlo spáleninou a ještě něčím těžko popsatelným. Ušli jen pár metrů a už zaslechli jakési přehrabování. Rytíři vytáhli své meče, Gorf natáhl luk a pomalu pokračovali.

Došli až na okraj veliké členité jeskyně. Hned na straně byl výklenek, ve kterém byly naskládané nějaké věci. Vypadaly oproti zbytku jeskyně podivně srovnaně a jako by je používal nějaký člověk. Všemu kralovala vykládaná mithrilová truhlice s podivným zámkem.

V tu chvíli si jich všiml drak. Mocně zařval a vyrazil k nim. Když už jsou tady, musí získat to, co je v té truhlici! „Braňte se! Gorfe, odemkni to!" vykřikl Warin a kryjící se za štítem se vrhl drakovi vstříc.

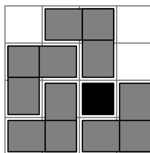
Gorf doběhl k truhlici a již si chystal paklíče, když tu mu došlo, že tady jeho paklíče budou k ničemu...

29-3-5 Dračí zámek 9 bodů

Gorf potřebuje otevřít truhlu zamknutou podivným zámkem. Na truhle je čtvercová mřížka veliká $N \times N$ políček pro $N = 2^k$, ve které je právě jedno políčko plné. Vedle truhly se pak válí mnoho dílků ve tvaru L, které přesně pasují do mřížky na truhle.

Je potřeba dílky poskládat do mřížky na truhle tak, aby byla všechna políčka vyplněná, ale současně, aby se žádné dva dílky nepřekrývaly. Vymyslete postup, který toho (v nějakém rozumném čase) dosáhne.

Ukázku jednoho poskládání, které nezačalo správně, můžete vidět níže (černě je vyznačeno zaplněné políčko):



„Mám to!" zakřičel vítězoslavně Gorf, otevřel truhlu a popadl z ní kupu nějakých podivných svitků a něco jako deník. Nacpal to vše do pytle a ohlédl se na ostatní.

Oba rytíři i kouzelnice si hráli s drakem na kočku a tři myši – velký drak měl v jeskyni problém s otáčením a odvážné trojici se vždy povedlo na poslední chvíli uskočit, než na místo, kde před chvílí stáli, dopadl těžký dračí ocas. Drakovi ale docházela trpělivost a začínal plivat krátké záblesky ohně, jeden se právě Lianovi rozprskl o štít a na chvíli ho celého zalil do ohnivé koule. Byl nejvyšší čas zmizet.

Gorf střelil přesně mířeným šípem drakovi do oka a tím získal ostatním čas. Vyběhli nazpět do chodby, dostali se z jeskyně ven a skrčili se kousek od vchodu za velkým kamenem. Drak je zatím nepronásledoval a tak si všichni oddechli. Lian

ze sebe setřepal spálené zbytky svého pláště. Ještě, že jejich brnění bylo částečně protkáno i mithrilem a zásah od draka neprošel skrz.

Rhea mezitím studovala ukradené zápisky. „Tak takhle mu tedy přikazují, pomocí těch hatí v bažině. Proto tam byly ty světélkující klacky! Drak se na ně dívá ze vzduchu a vidí v nich obrazce!“

Z nitra jeskyně se ozvala zařvání, rychle jim docházel čas. „A dokážeš mu říci, aby odletěl pryč?“ zeptal se Gorf.

„Snad... tady! Tady je náčrt něčeho říkajícího mu, aby usnul. Běžte s Lianem do bažiny, já s Gorfem vylezeme na nějaké vyvýšené místo, ať to pořádně vidíme.“

Jak řekla, tak se také stalo. Dva silní rytíři odklusali po hatích směrem ke světélkujícím tyčím, Rhea s Gorfem si našli místo, odkud na obrazec viděli, a začali je navigovat.

29-3-6 Obrazec pro draka — 13 bodů

Drak je ovládán s pomocí obrazce na zemi vyskládaného v konvexním mnohoúhelníku. Mezi vrcholy tohoto mnohoúhelníku jsou položené dlouhé světélkující tyče a to tak, že se nekříží a celý obsah mnohoúhelníku je jimi rozdělen na trojúhelníky (informatici by řekli, že je *triangulován*).

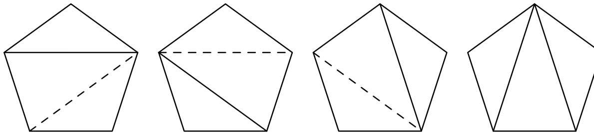
Nyní je v obrazci vyskládán jeden příkaz pro draka, ale my bychom ho chtěli změnit na jiný. V jednu chvíli můžeme pohybovat pouze s jednou tyčí a navíc ji můžeme přemístit zase jen tak, aby se s žádnou jinou tyčí nekřížila. Což znamená, že můžeme jen zvednout tyč, čímž nějaké dva trojúhelníky spojíme v jeden čtyřúhelník, a vzniklý čtyřúhelník můžeme zvednutou tyčí zase rozdělit na dva jiné trojúhelníky – budeme této operaci říkat *překlopení*.

Ze zadaného výchozího stavu chceme nějakou posloupností těchto překlopení změnit obrazec na jiný. Vstupem tedy bude dvojice triangulací konvexního N-úhelníku a vaším cílem je najít nějakou posloupnost překlopení převádějící jednu triangulaci na druhou. Pro obě triangulace je jasně dáno, který vrchol se má převést na který.

Nalezená posloupnost překlopení nemusí být nejkratší možná, stačí nalézt jakoukoliv fungující. Odhadněte ještě, kolik jich při vašem postupu maximálně může být.

Posloupnost několika překlopení může vypadat třeba jako níže (čárkovaně je vždy vyznačena tyč, se kterou chceme v dalším kroku pohybovat). Bystrý čtenář si jistě všimne, že toho samého lze dosáhnout o jeden krok kratším postupem, ale nám stačí jakákoliv posloupnost překlopení.

Zadání úloh KSP – 3. série



Když táhli poslední ze světélkujících tyčí, tak drak vylétl z hory ven. Na poslední chvíli, oddechli si oba rytíři. Drak naštvaně kroužil okolo hory a plival oheň na všechny strany. V měsíčním světle bylo vidět zbytky zapíchaných šípů v křídlech, které si odnesl od obránců Leyfastu, ale jeho letu to asi nijak nebránilo.

Pak si drak všiml obrazce na zemi a rázem jako by ho ovládlo něco jiného. V tu chvíli přestal plivat plameny, ještě párkrát oblétl horu a pak opatrně přistál před svou slují a pomalu vkráčel dovnitř.

„To je neuvěřitelné, jak někdo může takhle kontrolovat draka, to jsem ještě neviděla.“ pronesla Rhea, když se zase sešli. Měla v ruce zbytek poznámek, které sebrala ve sluji, poznámek, které by je mohly nakonec dovést až ke strůjci tohoto všeho. Ještě je asi všechny čeká dlouhá cesta... ale o tom zase někdy jindy.

Další příběh ze severu vyprávěl

Jirka Setnička

Čtvrtá série

Chodbou se rozléhal hovor, smích i šustění papírů, jak se někteří studenti ještě snažili honem rychle něco doučit. Vilém pobaveně sledoval svého kamaráda, který rozebíral, jaké všechny otázky by v testu nechtěl potkat.

„A hlavně doufám, že se nebudou ptát na Novákovy spisky o štěstí a náhodě,“ máchl Bernard rukama.

„Tak u nich snad stačí vědět, že je postupně někdo krade z Archivu, ne?“

„Další!“ přerušil kamarády mocný hlas. Oba chlapci se usmáli a Vilém se vydal na klasickou kontrolu před testem.

Kontrolující muž mu položil několik obvyklých otázek a začal kontrolovat Vilémovo oblečení. „Kapesníčky?“ ujišťoval se, když pohmatem našel předmět v kapse kalhot. Vilémovi se rozbušilo srdce, zatajil dech a jen němě přikývl. Snad proto ho muž chvíli nejistě pozoroval, pak rázně sáhl do kapsy… a vytáhl malou měkkou podkovu. Chvíli se tvářil zmateně, pak mu zazářily oči, když si všiml několika čtyřlístků umně maskovaných ve vzoru Vilémovy košile.

„Čtyřlístky na zkoušku a podkova na kontrolu? Dobrej nápad, chlape,“ poplácal Viléma po zádech a čtyřlístky jeden po druhém zabavil. „Máte štěstí, že studujete zrovna na Institutu náhodných a nápadně nenáhodných jevů,“ dodal ještě a s úsměvem ho poslal do zkouškové místnosti.

Vilém rozhodně na čtyřlístky nespoléhal, a tak navzdory jejich zabavení za hodinu a půl odevzdával hustě popsanou písemku. Dva zkoušející na sebe mrkli. „Oni mají body ze semestru, že? Můžeme při odevzdání přijímat jen písemky od lidí, kteří mají víc bodů než jejich předchůdce?“

29-4-1 Odevzdávání písemek — 10 bodů

Studenti utvořili frontu na odevzdání písemek, každý ji odevzdá jednomu ze dvou zkoušejících. Každý student má také nějaké body ze semestru. Aby ale mohl student písemku odevzdat, musí mít více bodů než poslední student, který odevzdal písemku stejnému opravujícímu; předbíhat se nesmí. Komu má kdo odevzdávat, aby mohli odevzdat všichni?

Formálněji: je dána posloupnost celých čísel. Vaším úkolem je rozdělit ji na dvě rostoucí podposloupnosti (případně zjistit, že to nejde).

Můžeme si představit, že každý prvek posloupnosti obarvíme jednou ze dvou barev (třeba modrou nebo červenou). A chceme to udělat tak, aby modré prvky tvořily rostoucí podposloupnost – tedy pokud budeme číst zleva doprava jen modré prvky (červené přeskakujeme), budou přečtená čísla v rostoucím pořadí.

To samé musí platit pro červené prvky. Žádný prvek nemůže být obarven dvěma barvami ani zůstat neobarvený.

Formát vstupu: Posloupnost celých čísel.

Formát výstupu: Na prvním řádku modrá podposloupnost (všechny modré prvky čtené zleva doprava v původním pořadí), na druhém červená.

Ukázkový vstup:	*Ukázkový výstup:*
`9 4 14 5 17 8 26`	`9 14 17 26` `4 5 8`

Ukázkový vstup:	*Ukázkový výstup:*
`8 1 11 12 0 6`	`NELZE`

Nezapomeňte důkladně zdůvodnit, proč vaše řešení funguje.

Kdo ví, nakolik zkoušející jen provokovali, rozhodně se všem studentům povedlo písemku odevzdat. Vilém si ještě zašel s Bernardem na chvíli sednout ven a pak už hurá domů.

Následující ráno ho při pohledu do zrcadla přivítala černá plocha. „Zase? Tyhle krámy nic nevydrží. . . “ povzdechl si a vyměnil baterky. Ale pořád je to asi lepší než mít doma skleněné zrcadlo. Příběhů o strašidelných koncích už slyšel víc než dost.

Tenhle trend začal před pár lety, když se do aut místo zpětných zrcátek začaly instalovat zadní kamery. Tím sice neubylo dopravních nehod, ale výrazně se zmírnily jejich následky.

Smartphony po svém příchodu zase rychle vytlačily malá osobní zrcátka, se kterými vás dnes nepustí ani do letadla.

Jakmile dostatečně klesly ceny a spotřeba velkých LCD panelů, začala být i velká domácí zrcadla nahrazována elektronickými. Zpočátku to byly jednoduchoučké přístroje tvořené jen kamerou a displejem. Ale ve světě, kde i rychlovarné konvice mají Wi-Fi, nemohla zrcadla zůstat dlouho pozadu.

Dnes jsou z nich velmi chytrá (někteří stále trvají na tom, že příliš *chytrá) a flexibilní zařízení. Nejenže si na nich ráno přečtete noviny či prohlédnete kalendář, ale například mezi náctiletými děvčaty je velmi oblíbená aplikace* Zrcadlo, zrcadlo*. . .*

Jako každému složitějšímu zařízení, ani zrcadlům se nevyhnuly počítačové viry a útoky. Nejčastěji v podobě šmírování kamerou, ale rozmohly se i různé vtípky od nevinných (obraz vzhůru nohama) po celkem necitlivé (vidíte za zády strašidelný stín či svou podobiznu digitálně zestárlou o deset let). Ale to je malá daň za bezpečnost. V posledních měsících se dokonce mluví o plošném zákazu skleněných zrcadel v EU. . .

Dost filozofování o zrcadlech, za chvíli začíná ústní zkouška! Vilém se rychle sbalil a vyrazil.

* * *

Čekání na zkoušku si krátil prohlížením různých hracích automatů, které byly rozmístěny po chodbě. Žádné výhry nevydávaly, ale studenti si na nich mohli vyzkoušet své štěstí a různé způsoby, jak ho ovlivnit.

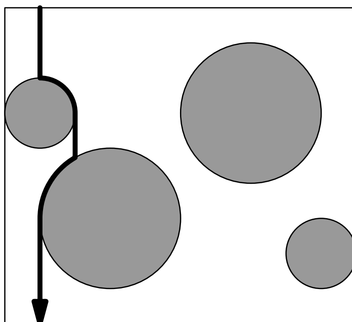
Jeden z automatů ho zvláště zaujal. Vypadal pro účely výzkumu štěstí až podezřele deterministicky. . .

29-4-2 Hrací automat 12 bodů

Na zdi visí hrací automat. Je tvořen úzkým prostorem mezi dvěma skly, do kterého je možné z libovolného místa podél horní hrany vhodit kuličku. Ta pak padá dolů a cestou naráží na kruhové překážky, po kterých se vždy skutálí a dál padá opět kolmo k zemi. Nakonec dopadne na dolní hranu automatu, podél které je stupnice udávající skóre.

Celou situaci si můžeme představit dvojrozměrně. Plocha automatu je obdélník, po kterém jsou nepravidelně rozmístěné překážky ve tvaru kruhu. Překážky se nikdy nepřekrývají. Vhozená kulička je ve srovnání s těmito kruhy velmi malá, takže si ji můžeme představit jako bod. Volným prostorem padá vždy kolmo dolů. Pokud dopadne na kruhovou překážku, skutálí se po jejím horním okraji tak, jak byste čekali – doleva, pokud dopadla nalevo od středu překážky, doprava, pokud napravo.

Pro jednoduchost předpokládejte, že když kulička dopadne přesně na střed překážky, padá vždy vpravo.



Na vstupu dostanete nejdřív popis automatu: výšku H, šířku W a počet překážek N. Následují pro každou překážku tři čísla udávající střed a poloměr. Poté následuje Q dotazů. Každý dotaz je popsán jedním reálným číslem udávájícím x-ovou souřadnici místa, kde na horním okjraji automatu vhodíme kuličku. Pro každý dotaz chceme určit x-ovou souřadnici místa na dolním okraji automatu, kam kulička dopadne. Předpokládejte, že dotazů bude řádově alespoň tolik jako překážek.

Střih. Mezitím na nedaleké policejní stanici. . .

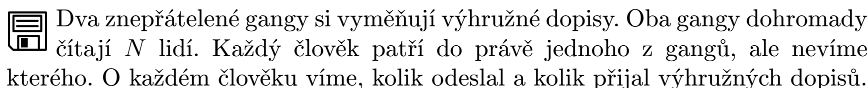
„Hlášení přibývá. Podivné nehody, zmizení. . . Včera jednoho člověka dvakrát zasáhl blesk! Oba zásahy přežil, ale bylo to na železničním přejezdu, a než se stačil

probrat, přejel ho vlak. Kolemjdocí prý tou dobou v okolí zahlédli černou kočku. Ale na takovouhle smůlu černá kočka nestačí…“

„Může to samozřejmě být nějaký přirozený zdroj smůly, ale kolegové z kriminálky se domnívají, že za tímhle případem, a spoustou podobných, stojí nějaká organizovaná zločinecká banda.“

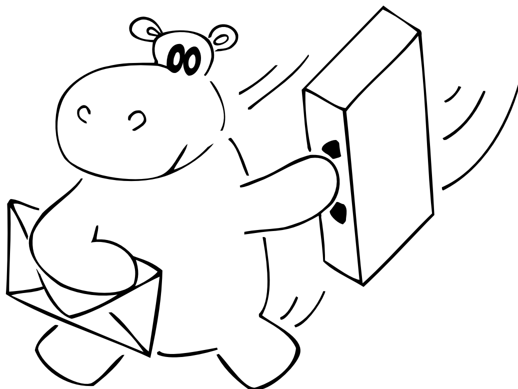
„Dokonce dvě,“ přihodil nově příchozí policista. „Zachytili nějaké výhružné dopisy, které si posílají mezi sebou. Máme spoustu podezřelých, ale zatím netušíme, kdo kam patří…“

29-4-3 Výhružné dopisy — 11 bodů



Dva znepřátelené gangy si vyměňují výhružné dopisy. Oba gangy dohromady čítají N lidí. Každý člověk patří do právě jednoho z gangů, ale nevíme kterého. O každém člověku víme, kolik odeslal a kolik přijal výhružných dopisů.

Rádi bychom zjistili, kdo komu posílal dopisy. To nejspíš nepůjde jednoznačně, ale stačí nám najít jedno libovolné řešení. Jedna dvojice odesílatel-adresát si mohla poslat více dopisů, v takovém případě nás zajímá kolik. Výhružné dopisy se posílají pouze mezi gangy, nikdy uvnitř gangu.



Jinými slovy, hledáme orientovaný bipartitní multigraf (tedy graf, kde mezi jednou dvojicí vrcholů může vést více hran) – vrcholy představují lidi, partity gangy a každá hrana jeden poslaný dopis (orientovaná od odesílatele k příjemci). Pro každý vrchol máme předepsán jeho vstupní a výstupní stupeň (kolik hran v něm má začínat a kolik končit). Rozdělení vrcholů na partity není určeno, to je třeba najít. Parity mohou být různě velké.

Formát vstupu: Na prvním řádku bude přirozené číslo N udávající počet lidí (vrcholů). Dále pro každého člověka řádek se dvěma nezápornými celými čísly udávajícími, kolik dopisů daný člověk přijal a kolik odeslal.

Formát výstupu: Pro každou dvojici odesílatel-adresát, která si poslala alespoň jeden dopis, vypište trojici čísel: pořadová čísla odesílatele a adresáta (0 až $N-1$) a počet dopisů, které odesílatel poslal adresátovi. Pokud existuje více řešení, vypište libovolné. Vstupy budou zadány tak, aby vždy alespoň jedno řešení existovalo.

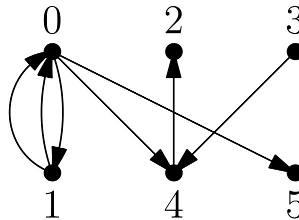
Ukázkový vstup:

```
6
2 3
1 2
1 0
0 1
2 1
1 0
```

Ukázkový výstup:

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

Odpovídající bipartitní multigraf vypadá následovně:



I v tomto případě jde jen o jedno z mnoha možných řešení.

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

Do místnosti vešel starší strážník, patrně náčelník zdejší stanice. „Tak," odmlčel se před nepříjemnou otázkou, „kdo tady bude v pátek?"

„Já ne!" ozval se jeden z policistů. „Viděli jste můj horoskop?"

Ostatní si vytáhli každý po jedné sirce.

Náčelník přešel do vedlejší místnosti, kde mezitím pokračovala přestavba místní počítačové sítě. Byla tam spousta počítačů, switchů, routerů a mezi nimi čím dál více kabelů. „Pozor kabel!" zavolal jeden z instalujících.

„Na co tu máme víc počítačů než celý útvar informační bezpečnosti? O ty kabely se brzo někdo přerazí. . . "

„Evropská unie," odvětil montér, nepřestávaje tahat kabely. „Pokud neutratíme všechno, musíme dotaci vrátit. Ale oficiálně – jako zálohu pro případ výpadku."

29-4-4 Policejní síť — 8 bodů

Policejní síť tvoří N počítačů, každý může být propojen s několika dalšími. Síť tvoří strom.

Navíc máme určeno K důležitých počítačů (vrcholů). Ty chceme spárovat do dvojic, které se budou navzájem zálohovat: pokud jeden počítač z dané dvojice přestane fungovat, zastoupí jeho činnost ten druhý.

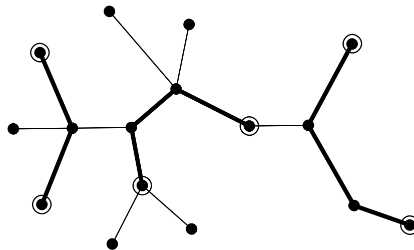
Aby to bylo možné, musí si počítače ve dvojici průběžně navzájem předávat všechna svá data – ta tečou po cestě, která dané dva vrcholy ve stromu spojuje (říkejme jí *spojení*).

Není určeno, který počítač máme spárovat s kterým, můžeme si vybrat libovolně. Ale protože nakoupené počítače přes svou cenu nejsou příliš výkonné, nechceme, aby jakýmkoli počítačem procházelo víc než jedno spojení.

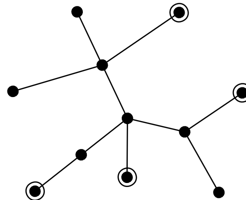
Tedy hledáme takové rozdělení důležitých vrcholů do dvojic, aby cesty spojující vrcholy v každé dvojici byly navzájem disjunktní – neměly žádné společné vrcholy ani hrany.

Pokud existuje více možných řešení, vypište všechna.

Na následujícím obrázku je příklad takového stromu. Důležité vrcholy (dostanete na vstupu) jsou označeny kroužkem. Nalezená spojení (výstup algoritmu) jsou zvýrazněná tučně. V tomto případě existuje jediné řešení.



Ale pro následující zadání žádné řešení neexistuje:



Ráno se Vilémovi povedlo při vypínání budíku spadnout z postele a v koupelně si málem zlomil nohu. Když mu jeho chytré zrcadlo s kalendářem prozradilo, že je čtvrtek 12., raději nepřemýšlel o tom, jak zlý bude příští den, a rozhodl se doplnit zásoby.

Jelikož nemohl najít oblíbenou tašku, vzal si prostě svůj běžný batoh a přihodil raději pár talismanů. Jim navzdory ho málem praštily vchodové dveře. Ještě že na ulici nebylo moc lidí, takže si netrhl takovou ostudu.

Neprošel ani celou ulicí, když se mu rozvázala tkanička, on o ni zakopl a při pádu hodil klíče do kanálu. Tehdy ho dostihla děsivá myšlenka. Jeho středeční *zkouška byla předevčírem, přesto jeho zrcadlo včera tvrdilo, že je středa, a to znamená… že je pátek třináctého!*

Na chvíli ho popadl vztek. Dávno bylo schváleno, že ten den budou vždy od časného rána do večera znít sirény, a zatím ticho. Holt byly sirény asi rozbité, jako obvykle. Vztek ale vystřídala zvědavost. Kdo je tedy těch pár lidí, kteří si troufli vyjít ven?

V té chvíli se ovšem jeden z těch lidí vydal právě k Vilémovi. „Co tu sakra vyvádíš? Vy nováčci 'ste pořád větší amatéři!“ nadával, popadl Viléma a vlekl ho ulicí. Vilémovi se hlavou honily myšlenky a moc nestíhal vnímat cestu. Najednou byli před jakousi budovou, najednou se v ní proplétali různými neznačenými schodišti a východy… a najednou byli na místě. Mohli být vážně ve třináctém patře?

Vilém se opatrně rozhlédl kolem sebe. Jeho pozornost upoutala zejména knihovna u zdi. Byly v ní vyskládané různé knihy a sešítky a… to snad nebylo možné! Novákovy spisky o štěstí a náhodě. Mohly všechny ukradené kopie z Archivu dokumentů o mezipřirozenosti zmizet sem?

29-4-5 Chybějící spisek — 12 bodů

Z Archivu dokumentů o mezipřirozenosti bylo postupně ukradeno mnoho očíslovaných spisků. Vilém nyní M těchto spisků vidí v knihovně v tajné skrýši a zajímalo by ho, jaké nejmenší číslo spisku tu chybí.

Protože ale v místnosti není sám, nemůže spisky jen tak přerovnávat, stejně tak si nemůže prostě vytáhnout papír a tužku a psát si poznámky. Musí si vystačit s omezeným množstvím paměti poskytnutým vlastní hlavou…

Formálněji: v paměti máte k dispozici nesetříděnou posloupnost N čísel. Tato část paměti je jen pro čtení, to zejména znamená, že si posloupnost nemůžete setřídit. Dále máte k dispozici konstantní množství pomocné paměti. Určete nejmenší přirozené číslo, které se v posloupnosti nevyskytuje.

Ukázkový vstup:	*Ukázkový výstup:*
`3 0 9 8 6 1`	`2`

Něco Viléma vrátilo do přítomnosti, upoutalo jeho pozornost zpátky do místnosti. Mezitím se tu nahromadilo dost lidí, a ti všichni teď umlkli a pozorovali přicházejícího muže.

Vilém si nebyl jistý, jestli ho víc znepokojuje černá kočka v mužově náručí, nebo fakt, že nikoho jiného tato maličkost zjevně netrápí.

„Jsem rád, že vás tu všechny zase vidím," ujal se muž slova. „Jak jistě víte, černá kočka je ideálním prostředkem k neutralizaci nežádoucích osob. Výsledek nejenže vypadá jako nehoda, ona to doopravdy je nehoda.

Musíte správně odhadnout počet. Jedna obvykle nestačí, ale pokud jich použijete příliš," pohlédl přísně na jednoho z přítomných, „jsou druhý den noviny plné zpráv o zásazích blesku na železničním přejezdu. Jenže i když se trefíte, občas někdo kočku zahlédne…

Proto jsem vážně rád, že se nám tehdy povedlo zfalšovat utracení téhle krásky," zdvihl kočku spočívající mu v náručí. „Určitě si vzpomínáte, jak byla vláda nesvá z černé kočky, která nenosí smůlu, ale nejspíš ani oni nevěděli, jak silnou mají v ruce zbraň."

„Naši výzkumníci se ovšem pustili do práce a zjistili…," muž se zamyšleně zamračil, pak mávl rukou, „nějakou genovou odlišnost. A protože tu od nás mají skvělé vybavení a jsou šikovní, dovolte mi představit vám," dramaticky se otočil a ukázal k přicházející hnědé kočce.

Obezřetně k ní došel, opatrně ji vzal do náruče a pak si z přihlížejících vyvolal jednoho neochotného dobrovolníka. O pár spadlých předmětů, vylitých skleniček, zakopnutí a dalších pozoruhodných náhod později začínalo být jasné, jak mocnou zbraň má skupina k dispozici.

„Hele, a proč vlastně nenabarvíme nějakou černou kočku?" zeptal se tiše někdo poblíž Viléma. „Nepamatuješ si snad, co se stalo Dlouhánovi, když se o to pokusil?" odpověděl hned jiný a pak byl klid, dokud se slova opět neujal šéf.

„Věřím, že s naším novým miláčkem se nám bude dobře dařit a pěkně se rozrosteme. Proto si taky už brzy pořídíme nové sídlo, hned jak vymyslíme, jak velký pozemek vlastně chceme koupit."

29-4-6 Nové sídlo — 11 bodů

Zločinecká organizace si chce postavit nové sídlo. To má půdorys ve tvaru konvexního mnohoúhelníku. Ráda by koupila nejmenší možný obdélníkový pozemek, na který se budova vejde. Ve městě jsou pozemky drahé a jiný než obdélníkový vám neprodají. Zároveň by ale chtěla, aby budova alespoň jednou stranou přiléhala k ulici (tedy k okraji pozemku).

Formálně: je dán konvexní mnohoúhelník. Najděte (obsahem) nejmenší obdélník, do kterého se mnohoúhelník celý vejde. Obdélník může být v obecné poloze (libovolně natočený), ale chceme, aby alespoň jedna strana mnohoúhelníku přiléhala ke straně opsaného obdélníku.

Formát vstupu: Počet vrcholů mnohoúhelníka N a souřadnice jeho vrcholů v pořadí na obvodu. Můžete předpokládat, že souřadnice jsou celočíselné.

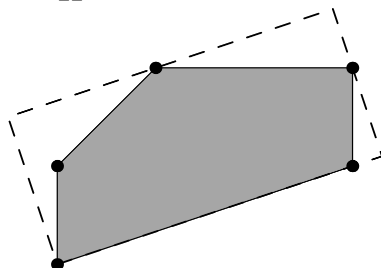
Formát výstupu: Jedno reálné číslo udávající obsah nejmenšího opsaného obdélníku splňujícího popsaná kritéria.

Ukázkový vstup:

```
5
0 0
6 2
6 4
2 4
0 2
```

Ukázkový výstup:

```
22
```



Tohle je zlé...

Vilém tušil, že by měl něco udělat, ale neměl ponětí co. Zavolat policii zřejmě nepomůže. Dnes si nikdo zasáhnout netroufne. I kdyby troufl, nejspíš by to skončilo fiaskem.

V zoufalství prohrábl kapsy. Zalaminovaný čtyřlístek měl jeden list ulomený. Pochopitelně.

Krom jiného vytáhl i malé klubíčko vlněné příze, kterým občas bavíval své domácí kočky. Teprve při pohledu na něj si pořádně uvědomil, že nebezpečná kočka nosící smůlu je pořád kočka. A kočky si rády hrají.

Pár metrů odmotal, zbytek zajistil uzlem a hodil směrem k hnědé kočce. Plán byl jednoduchý: upoutat její pozornost, přitáhnout klubíčko zpět a s trochou štěstí (ehm) ji tím přilákat.

Ale přece byste nečekali, že v pátek třináctého nějaký plán *vyjde. Zhruba ve stejnou chvíli se staly dvě věci: šéf se ohlédl Vilémovým směrem a provázek se přetrhl. Kočka viděla klubíčko dopadnout asi metr před sebe. Najednou pocítila zvláštní nutkání líně odhlédnout a odejít, které přicházelo zdánlivě odnikud.*

Ale sama měla jiný názor. Tady se objevil zajímavý předmět, který chce prozkoumat, a pokud si vesmír myslí, že by měla znuděně odejít, tak si vesmír může s prominutím trhnout.

Neslyšně se připlížila ke klubíčku a párkrát do něj šťouchla packou. Vždy se o kus pohnulo a skutálelo zpátky.

„Co tam blbneš? Nehraj si a dávej pozor!“ adresoval nevrle šéf Vilémovi.

Ono se to nehýbe! Asi spí... Kočka se napřáhla a prudce vymrštila směrem ke klubíčku, odrážejíc ho o několik metrů kupředu. Wihíí!

Šéf, nezpozorovav toto dění, se pomalu rozešel směrem k Vilémovi právě ve chvíli, kdy kočka vyběhla za klubíčkem — a zkřížila tím šéfovi cestu.

* * *

Na policejní stanici byl toho dne klid. Seděl tu jediný strážník, který si vytáhl kratší sirku, a pozorně si prohlížel protější zeď. Raději si netroufl ani číst noviny.

Zadání úloh KSP – 4. série

Když tu z ničeho nic do služebny zmateně vběhl vyděšený muž. Vypadal, jako by spatřil nějaký přízrak, a pravděpodobně netušil, kde se nachází. Proběhl kanceláří bez ohlédnutí otevřenými dveřmi do sousední místnosti… kde zakopl o jeden z nově natažených síťových kabelů.

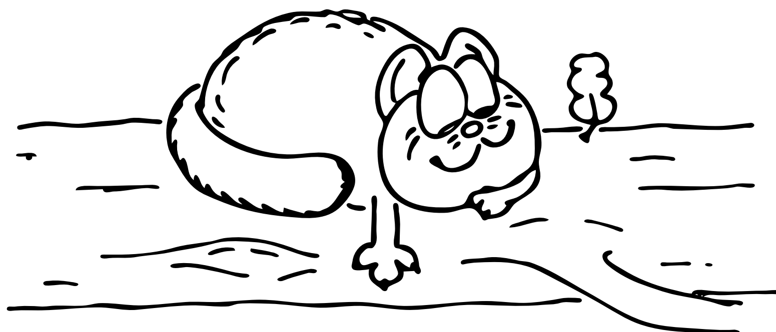
Při pádu mu vypadla spousta schémat a náčrtků popisujících plány jeho zločinecké organizace, poznámky o genetice černých koček, seznamy likvidovaných osob. Strážník k němu opatrně přišel a papíry prolistoval.

„Kdopak se to chytil – do naší sítě?“ nemohl si odpustit úšklebnou poznámku. „Z toho si nic nedělejte, tohle se tu stává pravidelně. V pátek třináctého zločince nechytáme. Nejenže to není bezpečné, ale hlavně to není potřeba. Přichází sami…“

Nepodceňujte kočky.

Příběh pro vás přichystali

Karry Burešová & Filip Štědronský



Pátá série

Naposledy se vrátíme k našim udatným hrdinům, se kterými jsme se potkali už v první a třetí sérii. Opustili jsme je potom, co zažehnali problém s drakem, a chystali se vydat dále na sever, aby zjistili, kdo za tím vším stojí.

⋆⋆⋆

Sirovi Warinovi, členovi Alvarezova řádu, právě končila hlídka. Už dva dny pozorovali hrad, kam je dovedly zápisky v deníku z jeskyně s drakem, a došli k tomu, že tu musí být nějaký velmi silný temný mág.

„Vydáme se dovnitř," řekl Warin ostatním, „ale nejdříve dáme vědět králi. Gorfe, můžeš připravit poštovního holuba?" obrátil se s dotazem k jejich lučištníkovi a nadanému zloději. „A my ostatní," podíval se na kouzelnici Rheu a na mladého rytíře Liana, „se zatím připravíme na proniknutí těmi tajnými chodbami."

Gorf přikývl a šel chystat holuba. Poslat zprávu holubí poštou ale nebylo jen tak, bylo potřeba naplánovat, kudy má putovat.

29-5-1 Holubí pošta — 10 bodů

Skupina hrdinů potřebuje co nejrychleji poslat zprávu holubí poštou do hlavního královského města. Je to ale velká vzdálenost, takže je potřeba zprávu poslat přes několik mezilehlých měst, kde se vždy vystřídají holubi.

Hrdinové mají mapu království jako graf, ve kterém vrcholy jsou města a hrany značí, mezi jakými městy je možné zprávu poslat (hrany jsou orientované). Každá hrana je ohodnocená časem, jak dlouho trvá holubovi přelet.

Ale aby to nebylo tak jednoduché, tak z každého města neodesílají zprávy nonstop, ale odesílají je jen v pracovní dobu této poštovní stanice. Pokud přiletí holub v průběhu pracovní doby, je zpráva hned předána dál, pokud ale přiletí mimo pracovní dobu, je zpráva odeslána dál až s opětovným zahájením pracovní doby.

Pro každé město budete mít zadané pravidelné intervaly pracovní doby a vaším úkolem je v této síti holubí pošty naplánovat časově nejkratší cestu mezi zadaným startem a královským hlavním městem.

Formát vstupu: Na prvním řádku dostanete počet měst N, počet hran M, index počátečního města S (indexujeme od nuly) a index královského města K. Poté bude na dalších N řádcích následovat popis měst a jejich pracovní doby a na dalších M řádcích pak popis hran. Všechny časové údaje jsou v celých hodinách a pokud holub přiletí na konci pracovní doby, tak je také zpráva odeslána ještě hned (tedy pracovní dobu bereme včetně koncových hodin). První holub vylétá vždy v čase 0.

Na i-tém řádku popisujícím města je zadaný popis města i jako trojice čísel $interval_i$, $delka_i$, $offset_i$ udávající interval, po jakém se opakující pracovní doby,

délku pracovní doby a offset, s jakým je začátek první pracovní doby posunutý. Tedy například popis 5 2 1 znamená, že pracovní doba je mezi hodinami 1 až 3 (offset 1 a délka 2) a opakuje se po 5 hodinách (tedy další je mezi hodinami 6 až 8, další pak 11 až 13 atd.). Vždy bude platit, že délka i offset budou maximálně tak velké, jako interval.

Popis hran je jednoduchý, na j-tém řádku popisů hran je trojice čísel a, b, h udávající, že j-tý holub letí z města a do města b a trvá mu to h hodin.

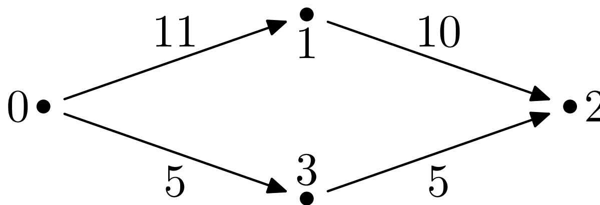
Formát výstupu: Na první řádek výstupu vypište délku cesty v hodinách, na druhém řádku pak vypište mezerami oddělenou posloupnost měst (včetně prvního a posledního), kterými tato nejkratší cesta vede.

Ukázkový vstup:

```
4 4 0 2
2 2 0
4 2 0
8 8 8
20 3 1
0 1 11
1 2 10
0 3 5
3 2 5
```

Ukázkový výstup:

```
22
0 1 2
```



I když doby čistého letu přes vrchol 3 jsou kratší, tak zde hůře vychází pracovní doba v mezilehlém městě (odeslání z města 3 by proběhlo až v čase 21, kdežto z města 1 odlétne holub již v čase 12).

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

Holub se zprávou na noze odlétl do podmračeného nebe a čtyři dobrodruzi se vydali lesem ke vstupu do jeskyní pod hradem. Všimli si, že skrz hradní bránu sice chodí mnoho skřetů, ale jeskynní vstup používají i různé tajemné postavy v pláštích.

Jeskyně byla osvětlená pochodněmi a nikdo ji nehlídal, ale kousek za vstupem byla velká brána. A ten, kdo ji zkonstruoval, se pravděpodobně vyžíval v roztodivných zámcích, podobně jako u truhlice v dračí jeskyni. Tady to vypadalo trochu,

jako kdyby zámek zkonstruovali trpaslíci – byly to různě natažené dráty, po kterých přeskakovaly modré jiskry magie.

Rhea vytáhla ukořistěný deník a začetla se do něj. Minuty ubíhaly a oba rytíři pozorně sledovali okolí, jestli se odněkud nevynoří nějaký skřet. I Gorf začínal být nejistý a žmoulal v ruce připravený šíp. Náhle Rhea nalezla správnou pasáž a odcitovala „Nejtenčí na každém kruhu, ten pozbyt má být svých druhů.“

Když se pozorně podívali na dveře, skutečně spatřili, že každý drát je jinak tlustý a každý se dá odpojit.

29-5-2 Odcyklení zámku — 11 bodů

Hrdinové se opět dostali k podivnému zámku. Tento zámek vypadá tak, že se skládá z mnoha vrcholů propojených dráty, po kterých přeskakují magické výboje. Každý drát má nějakou svoji tloušťku.

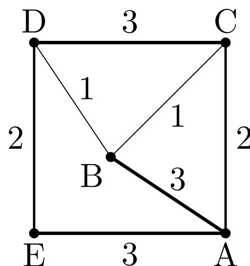
Podle pokynů z deníku je potřeba odpojit nějaké dráty tak, aby zanikly všechny cykly.

Není to ale tak jednoduché, v každém cyklu lze vždy odpojit jen ten nejtenčí drát (jinak by se obíhající mana vyzkratovala a to nikdo nechce).

Najděte tedy nějakou posloupnost drátů, které budete odpojovat a které ve chvíli odpojení musí být tím nejtenčím drátem (nebo jedním z více nejtenčích drátů) na alespoň jednom cyklu.

Jak už bylo řečeno výše, může existovat více drátů se stejnou tloušťkou a řešení tedy nemusí být jednoznačné. Nemusí být dokonce jednoznačné ani v počtu drátů, které je potřeba odpojit. Stačí najít jedno libovolné řešení.

Například pro následující graf:



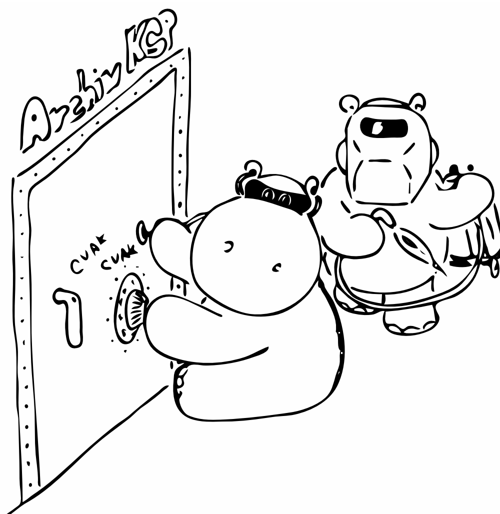
je správným řešením odebrat hrany např. v pořadí AC (nejkratší na cyklu $ACDEA$), BC, BD. Jiným správným pořadím může být BC, ED, BD.

Rhea jednou omylem sáhla na jiný než nejtenčí drát a dostala ránu, po které ji na pár minut ochrnula pravá ruka. Pak si ale již dávala pozor a brána se před nimi po odpojení posledního z drátů pomalu otevřela. Opatrně prošli dovnitř a dostali se po pár metrech na rozcestí chodeb. Brána se za nimi zase neslyšně uzavřela.

Zadání úloh KSP – 5. série

Z jedné strany se ozval hluk, a tak se hrdinové ukryli na chvíli do malé jeskyně na straně. Za pár sekund okolo překlusala malá skupinka skřetů, takže se hrdinové mohli vydat dál.

Pod hradem bylo celé bludiště chodeb. Kryti maskovacím kouzlem vyslechli rozhovor skřetů, kteří se bavili o nejlepší technice sekání hlav, později zase schováni za hromadou sudů sledovali skupinku mágů oděných v černém, jak provádějí nějaký okultní rituál. Bylo tu mnoho skřetů, o něco menší počet goblinů, zahlédli i skupinu trollů a sem tam se mihlo pár temných mágů. Některé části jeskyní kypěly životem, takže se k nim radši moc nepřibližovali, v jiných částech se zase táhly dlouhé temné chodby bez života. V jedné takové se po pár hodinách posadili nad rychlým jídlem.



„Nějaké nápady, kde by mohl být jejich vůdce?“ řekl potichu Warin, když ukusoval chleba. „Ve skřetí části ne, té se ti temní mágové vyhýbají,“ přemýšlel Lian, „ale jinak nemám ani potuchy, je to tu moc rozlehlé. A navíc jsme jen čtyři, těžko se budeme někam probíjet!“

„Hmm… a co někoho unést? Promluví, já už ho k tomu nějak přinutím!“ potěžkal Gorf svůj lovecký nůž. „Zadrž, mám lepší řešení,“ zastavila ho Rhea, „umím namíchat sérum pravdy.“

Rychle dojedli a vydali se zpátky k obydleným částem jeskyní. Vyhlédli si jednoho osamoceného mága a opatrně ho sledovali. Když zašel do boční chodby, byl jejich. Gorf k němu rychle přiskočil a přitiskl mu nůž pod krk: „Pohni prstem a je po tobě!“ sykl. Dotáhli ho dál od obydlených částí a Rhea začala míchat své lahvičky.

29-5-3 Sérum pravdy — 8 bodů

Kouzelnice Rhea by potřebovala namíchat sérum pravdy, aby od zajatého temného mága zjistila důležité informace.

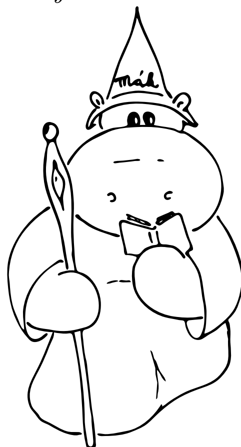
Jednou z ingrediencí je i rosa sbíraná o půlnoci. Ale musí jí být správné množství a co je ještě důležitější, tak se nedá bezmyšlenkovitě míchat rosa z náhodných nocí. Pokud už se nějaká musí míchat, tak jedině z několika po sobě navazujících nocí.

Rhea má teď před sebou postavenou řadu lahviček s kapkami rosy nasbíranými každou noc. Vzhledem k váze zajatého mága ví, že bude potřebovat množství co nejbližší K kapkám rosy.

Pomozte ji vybrat úsek lahviček, které dají dohromady součet kapek nejvíce se blížící zadanému K (obsah každé lahvičky je potřeba použít celý). Z některých nocí také může být v lahvičce jen rosná mlha (lahvičky s obsahem 0 kapek), ale ty je potřeba uvážit taky.

Příklad: Například pro $K = 12$ a lahvičky s počty kapek $15, 3, 6, 0, 4, 0, 0, 7, 8$ existuje více optimálních řešení. Jedno z nich je vzít čtyři lahvičky $3, 6, 0, 4$ se součtem 13, jiným řešením je třeba vzít lahvičky $4, 0, 0, 7$ se součtem 11.

Po podání séra pravdy temný mág promluvil a pověděl jim o málo používané cestě k jejich vůdci. Sice je na této cestě čekají asi nějaké pasti, ale aspoň se nebudou muset probíjet skrz hordy skřetů.



Uspaného temného mága nechali v temném koutě a vydali se cestou podle jeho rad. Opatrně překročili několik natažených lan spouštějících samostříly ve stěnách a postupovali dál. Po chvíli se zepředu začaly ozývat divné klapavé zvuky. Přitisknuti ke stěnám se plížili opatrně dál, než se dostali na okraj větší jeskyně.

Před nimi se jim naskytla podívaná na spoustu ozubených kol a rotujících kotoučů s ostrými čepelemi, které jim zahrazovaly cestu na druhou stranu. Také tu stál starý důlní vozík, který se dal roztlačit po kolejích skrz rotující čepele.

29-5-4 Rotující čepele — 11 bodů

Dobrodruzi se potřebují dostat na druhou stranu místnosti plné rotujících čepelí. Čepele rotují příliš rychle na to, aby mezi nimi šlo proběhnout, ale po podlaze místnosti vedou koleje a mohou se pokusit projet skrz důlním vozíkem.

Důlní vozík lze rozjet nějakou rychlostí (z rozsahu minimální a maximální rychlosti vozíku) a touto rychlostí pak projede celou místností (nemůže už brzdit ani zrychlovat).

Každá z rotujících čepelí je postavená kolmo na směr jízdy, má nějakou vzdálenost od začátku jeskyně a víme pro ni délky intervalů, kdy je jí bezpečné projet a kdy ne (ty se periodicky opakují, protože čepel rotuje stále stejnou rychlostí). V čase 0 jsou všechny čepele na začátku svého bezpečného intervalu.

Vymyslete postup, který pro zadané čepele a zadanou minimální a maximální rychlost vozíku najde jednu rychlost, kterou vozík zvládne projet skrz všechny čepele až na druhou stranu jeskyně (vozík vždy vyrazí z bodu 0 v čase 0), nebo rozhodněte, že takovou rychlost nalézt nejde.

Například mějme vozík s rozsahem rychlostí 1,5 až 4 m/s a dvě čepele. První má bezpečný i nebezpečný interval dlouhý 2 s a je vzdálena 12 m. Druhá má bezpečné okno 5 s, nebezpečné 3 s a je vzdálena 36 m.

V tomto případě je správným řešením např. rychlost 3 m/s. Při ní projedeme první čepelí v čase 4 s (těsně na začátku druhého bezpečného okna) a druhou v čase 12 s (uvnitř bezpečného intervalu od 8 s do 13 s).

Jiným správným řešením je rychlost 2 m/s.

Na druhé straně si všichni oddychli, trefit správnou chvíli na průjezd skrz čepele nebylo snadné.

Z konce jeskyně stoupaly nahoru dlouhé točité schody. Vydali se po nich opatrně nahoru. Po nějaké chvíli se stěny jeskyně změnily v stěny z kamenných kvádrů, to když vystoupali až do samotného hradu. Po chvíli je jejich kroky zavedly do malé místnosti, na jejímž opačném konci byly kamenné dveře a vedle nich socha znázorňující sfingu.

„Vítej u dveří… Beliarových… kolik… podob mám?“ přečetl Lian otázku napsanou starými runami nad hlavou sfingy. „Co to znamená?“

„V deníku o tom nic není, ale je tady jedna dlouhá zašifrovaná pasáž,“ odpověděla Rhea.

„A nepomůže nám vědět, že je v ní zapsaná tahle otázka?“ napadlo Gorfa. Rhea se zamyslela a zadívala se na zašifrovaný text…

29-5-5 Zašifrovaný text — 12 bodů

V deníku se nachází dlouhá pasáž zašifrovaná pomocí Vigenerovy šifry. Ta funguje tak, že vezme dlouhou zprávu a nějaký (typicky kratší) klíč a šifruje jednotlivé znaky zprávy do posunutých abeced. Posun abecedy pro konkrétní písmeno vždy určí odpovídající znak klíče – pokud šifrujeme k-té písmeno zprávy, šifrujeme do abecedy posunuté tak, aby začínala na k-tý znak klíče (tedy pokud je k-tý znak zprávy `a`, je posunutá abeceda stejná jako normální). Pokud je zpráva delší než klíč, tak klíč opakujeme dokola.

Ukázka zprávy zašifrované pomocí klíče `beliar`:

```
  vitejudveribeliarovych
+ beliarbeliarbeliarbeli
= wmemjlezpzisfptirfwcnp
```

Prolomit Vigenerovu šifru bez nějaké další informace je docela těžké. Naštěstí naši hrdinové znají větu, která by se ve zprávě měla objevit, a navíc ví, že tato věta je řádově delší než klíč.

Vymyslete algoritmus, který pro danou zašifrovanou zprávu a pro větu, která se v původním textu vyskytuje, nalezne klíč, kterým lze zprávu dešifrovat. Tím myslíme najít nějaký klíč, jehož odečtením od zašifrované zprávy dostaneme zprávu, ve které se někde vyskytuje zadaná věta.

Pokud je délka klíče K, máte slíbeno, že délka známé věty bude alespoň K^2. Pro některé texty a věty může existovat více řešení, můžete vybrat libovolné z nich.

Skutečně, po chvíli snažení se jim díky odhadnuté větě povedlo pasáž dešifrovat a nalézt v ní odpověď. Po jejím vyřčení se dveře se skřípotem otevřely. Skrz dveře se dostali do velké místnosti obehnané gobelíny.

Než se však stihli rozkoukat, řítila se k nim vysoká postava v černém plášti, okolo které vyzařovala rudá aura. „Co tu děláte!?!“ zahřímal hlas, který vůbec nezněl jako z tohoto světa. To musel být vůdce skřetích hord, temný mág Beliar!

Než stihli zareagovat, vrhl k nim Beliar blesk. Gorf na poslední chvíli uskočil a blesk rozštípl kamennou zeď za jeho zády. Odletující úlomky kamene na chvíli

vyvedly z rovnováhy i samotného Beliara a daly tak naší skupince čas se rozptýlit po místnosti.

Gorf vyslal několik šípů, ale ty se odrazily o silový štít, který Beliar okolo sebe vytvořil. Z boku místnosti se začali hrnout skřeti, a tak Warin s Lianem vyběhli tím směrem mávaje meči a působíce zmatek a zděšení.

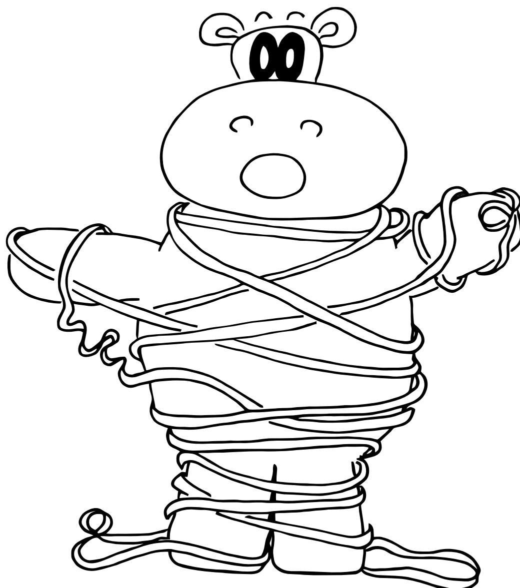
Kouzla létala vzduchem. Ani Rhea, ani Beliar neměli nad tím druhým navrch, ale Rhee rychle docházely síly. Držela svůj ochranný štít a pokoušela se sestavit z dostupných sil co nejsilnější kouzlo.

29-5-6 Nejsilnější kouzlo 10 bodů

Kouzelnice Rhea bojuje s mágem Beliarem a potřebovala by seslat zvlášť mocné kouzlo. Mocná kouzla se čtou ze svitků a v tomto typu kouzlení většinou platí, že čím je kouzlo delší, tím je také silnější. A ta nejsilnější kouzla mají často i nějaké speciální vlastnosti, třeba že jsou palindromy (tedy že se stejně čtou ze začátku i z konce).

Vymyslete algoritmus, který v zadaném textu (posloupnosti písmen) najde nejdelší palindrom (v případě více nejdelších palindromů libovolný z nich).

Příklad: V textu `aasqaanaaqqaaert` je nejdelším palindromem `qaanaaq` o délce 7 znaků.



Ani Beliar ale nezahálel. Těsně před tím, než svoje kouzlo měla připravené Rhea, vyslal svoje kouzlo k ní. Warin vše sledoval jako ve zpomaleném filmu. Jak Beliar zvedá ruku s holí. Jak se on sám odráží z paty a mečem rozpolcuje jednoho skřeta vedví. Jak se na Beliarových prstech tvoří koule magie. Cítil vlastní štít a to, jak běží a skáče. A pak si magická koule našla jeho štít a rozprskla se o něj. . .

⋆⋆⋆

Probudil se a okolo bylo nesnesitelné světlo. Zamrkal. Pak ještě jednou a okolo se začaly vynořovat obrysy místnosti. Místnosti, kde sváděli boj. Ale teď tu bylo ticho. Tedy skoro.

„No tys nám dal. Myslím, že budeš potřebovat nový štít," smála se radostí Rhea, nad kterou se skláněl ještě Gorf. Warin se podíval dolů na své spálené brnění a na zbytky pokrouceného kovu, které bývaly štítem z mithrilu. Ruka ho pekelně bolela, ale hýbat s ní mohl. „Co se. . . ?"

„Vyhráli jsme. To, jak jsi vlétl do toho kouzla, bylo hrozně hrdinské, ale hrozně nezodpovědné," pokárala ho Rhea, „ale dal jsi mi čas dokončit kouzlo a tím porazit Beliara. Po zhroucení jeho sil se rozpadl na prach a všechny skřety jako by popadl amok. Začali se zabíjet navzájem. Lian ještě čistí tohle patro, ale myslím si, že jsme vyhráli."

S Gorfem pomohli Warinovi na nohy a vydali se k balkonu. Lian se k nim vzápětí připojil a společně vyšli ven. Dole se od hradních bran ještě vzdalovaly malé skupinky skřetů. Potom, co zmizela vůle ovládající je, utíkali zpátky domů. Severní země byly (aspoň nyní) zachráněny, a to díky této udatné skupince hrdinů.

Jejich cestu s vámi sledoval

Jirka Setnička

Seriál – Stromy

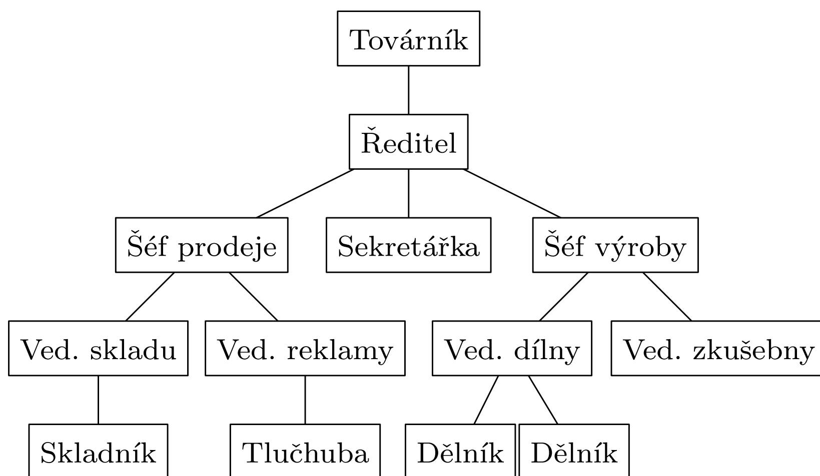
29-1-7 Stromy kolem nás 15 bodů

Pojďte, v letošním seriálu spolu budeme zkoumat stromy. Ne snad že bychom vás chtěli přeučit na botaniky – půjde nám o stromy ryze abstraktní, matematické. A ještě více o algoritmy pro práci s nimi.

Klíč k určování stromů

Definici stromu najdete v kuchařce této série: je to souvislý graf bez kružnic. To zní učeně, ale kupodivu je to i velmi praktické: příklady takových stromů potkáváme všude kolem sebe.

Hierarchie – kupříkladu továrna má továrníka, pak ředitele, ten své náměstky, jejich podřízenými jsou šéfové oddělení, jim podléhají mistři v dílnách, a tak dále až k řadovým dělníkům. Příslušný strom sestává z vrcholů (což jsou zaměstnanci) a hran (vztahy „být přímým podřízeným"). [20]



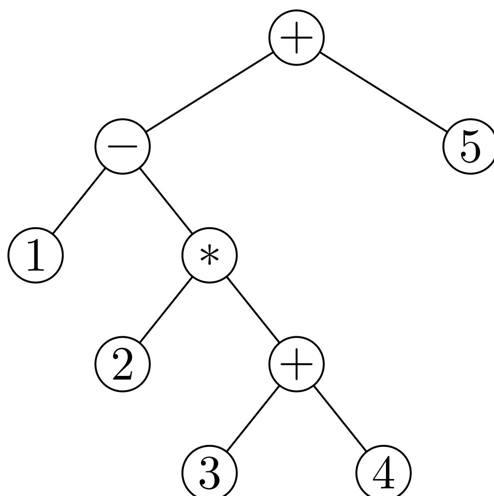
Stromy popisující nějakou hierarchii mají většinou jeden význačný vrchol – v našem příkladě je to pan továrník, obecně se mu říká *kořen* stromu. Jakmile nějaký kořen zvolíme, hned je jasné, kterým směrem je to ve stromu *nahoru* (ke kořeni) a kterým *dolů*. Nenechte se prosím zmást tím, že na rozdíl od biologů matematici kreslívají stromy kořenem nahoru. Podle směru také můžeme sousedy

[20] Sám název *hierarchie* pochází z byzantské řečtiny: *hieros* znamená svatý, z toho *hiereus* je kněz; *arché* značí moc, vládu. Jak to souvisí: Původně se hierarchií nazývaly složité vztahy podřízenosti mezi církevními hodnostáři.

každého vrcholu rozdělit na *otce* (směrem ke kořeni, v našem příkladě nadřízený) a *syny* (směrem od kořene, tedy podřízení). Kořen nemá otce, ostatní vrcholy mají právě jednoho otce. *Listy* jsou ty vrcholy, jež nemají syny.

Libovolný vrchol v spolu se všemi svými potomky (to jsou jeho synové, pak synové synů, atd.) tvoří *podstrom,* jehož kořenem je v.

Aritmetický výraz – mějme nějaký výraz s čísly a aritmetickými operacemi, třeba `1-2*(3+4)+5`. Pořadí, v jakém se jednotlivé operace vyhodnocují, můžeme popsat stromem:



Listy odpovídají zadaným číslům, ostatní (vnitřní) vrcholy jednotlivým podvýrazům. V každém vnitřním vrcholu bydlí jedna operace, která dostane mezivýsledky ze svých synů a pošle svůj výsledek otci. V kořeni pak získáme výsledek celého výrazu. Jelikož běžné operace pracují s právě dvěma čísly, půjde o strom *binární* – tedy každý vrchol kromě listů bude mít právě dva syny.

Městečko lakomců – mapu nějakého městečka si můžeme představit jako obecný graf: hrany odpovídají ulicím, vrcholy křižovatkám; slepé konce budeme považovat za speciální případ křižovatek. Pokud ale je pan starosta držgrešle a chce udržovat co nejméně silnic, brzy z městečka zbude jen *kostra,* tedy strom propojující všechny křižovatky. To je příklad stromu, který nemá žádný přirozený kořen. Můžeme ho tedy zakořenit libovolně, třeba na náměstí s radnicí.

Úkol 1 [1b]: Najděte nějaký další pěkný příklad stromu. Čím méně bude podobný těm našim, tím lépe.

Reprezentace stromu

Jak stromy reprezentovat v paměti počítače? Zajisté můžeme použít totéž, co u obecných grafů: vrcholy očíslujeme a pro každý z nich si zapamatujeme seznam čísel sousedů. Jinými slovy, každý vrchol si pamatuje seznam všech hran, které z něj vedou.

Zakořeněné stromy obvykle prohledáváme od kořene dolů. Tehdy si stačí v každém vrcholu pamatovat seznam jeho synů. Někdy se hodí zapamatovat si navíc i otce vrcholu (případně informaci, že jde o kořen).

Snadno odvodíme, že strom o n vrcholech má přesně $n-1$ hran: Zakořeníme ho v libovolném vrcholu a všimneme si, že z každého vrcholu kromě kořene vede právě jedna hrana do otce. Tak jsme každou hranu započítali právě jednou.

Budeme se proto snažit, aby všechny algoritmy pro práci se stromy běžely v čase $\mathcal{O}(n)$, kde n je počet vrcholů. To postačí na projití všech vrcholů a hran, ale už ne na cokoliv náročnějšího.

Prohledávání do hloubky

Chceme-li navštívit všechny vrcholy stromu, můžeme to udělat třeba *prohledáváním do hloubky.* Začneme v kořeni, a kdykoliv přijdeme do nějakého vrcholu, rekurzivně se zavoláme na všechny jeho syny. Přesněji řečeno, nejprve se zavoláme na prvního syna, až se z rekurze vrátíme (prohledali jsme podstrom pod tímto synem), zavoláme se na druhého syna, a tak dále.

Abychom lépe viděli, jak prohledávání do hloubky probíhá, doplníme do něj výpis levé závorky při vstupu do vrcholu a pravé při jeho opuštění.

DFS(v):

1. Vypíšeme `(`.
2. Pro všechny syny s vrcholu v:
3. Zavoláme DFS(s)
4. Vypíšeme `)`.

Spustíme-li algoritmus DFS (tak se mu říká podle anglického názvu *depth-first search*) na náš příklad stromu výrazu, vypíše `((()(()(()())))())`. Všimněte si, že jsme strom jakoby „obešli po obvodu“ – kdykoliv jdeme po hraně dolů, píšeme levou závorku, kdykoliv nahoru, pravou. Navíc přidáme jeden úplně vnější pár závorek.

Celý průchod stromem trvá $\mathcal{O}(n)$: v každém vrcholu strávíme čas $\mathcal{O}(1 +$ počet synů$)$. Pokud to posčítáme přes všechny vrcholy, jedničky se sečtou na n a počty synů na $n-1$.

Během procházení stromu můžeme také zpracovávat hodnoty uložené ve vrcholech, třeba je zase vypisovat. Můžeme je vypsat hned při navštívení vrcholu (tomu se říká *preorder* neboli *prefixový* výpis), nebo až při opuštění (*postorder*, *postfixový* výpis), případně mezi návštěvami synů (*inorder*, *infixový*). Porovnejme, jak to dopadne:

prefixový	`(+(-(1)(*(2)(+(3)(4))))(5))`
postfixový	`(((1)((2)((3)(4)+)*)-)(5)+)`
infixový	`(((1)-((2)*((3)+(4))))+(5))`

Infixový zápis tedy (až na přebytečné závorky) odpovídá tomu, jak obvykle výrazy zapisujeme. Aby se hodnoty v listech neztratily, samozřejmě je vypisujeme přímo, i když mezi žádnými syny neleží.

Postfixový zápis má zase tu hezkou vlastnost, že je jednoznačný i bez závorek. I z očesaného `1 2 3 4 + * - 5 +` můžeme rekonstruovat celý strom. Podobně pro prefixový zápis.

Úkol 2 [2b]: Vymyslete algoritmus, který vytvoří strom z jeho postfixového výpisu bez závorek. Vyzkoušejte si to třeba na výrazech.

Úkol 3 [1b]: Ukažte dva různé binární stromy se stejným infixovým zápisem bez závorek. Jak vypadá jejich prefixový a postfixový zápis?

Výpočty pomocí DFS

Spoustu vlastností stromů můžeme počítat rekurzivně: stačí je umět spočítat pro listy a pak říci, jak z výsledků pro podstromy stanovit výsledek pro celý strom. Takový výpočet jde jednoduše zabudovat do DFS. Začněme triviálním příkladem.

Počet listů – stačí vracet z listů jedničku a ve vnitřních vrcholech hodnoty sčítat:

PočetListů(v):

1. Je-li v list, vrátíme 1.
2. $\ell \leftarrow 0$
3. Pro všechny syny s vrcholu v:
4. $\quad \ell \leftarrow \ell + \text{PočetListů}(s)$
5. Vrátíme ℓ.

Jelikož jsme každý krok prohledávání jenom konstanta-krát zpomalili, algoritmus má opět lineární složitost.

Hloubka stromu – tak se říká délce nejdelší cesty z kořene do listu (délka cesty se měří v hranách). Opět ji spočítáme úpravou DFS: nahlédneme, že hloubka stromu je o jedna více než maximum z hloubek podstromů (nejdelší cesta z kořene do listu musí vést z kořene do některého podstromu a pak pokračovat nejdelší cestou uvnitř podstromu). Hloubka listu je 0. Lineární algoritmus následuje:

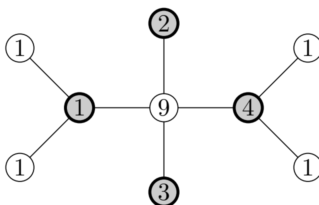
Hloubka(v):
1. Je-li v list, vrátíme 0.
2. $h \leftarrow 0$
3. Pro všechny syny s vrcholu v:
4. $h \leftarrow \max(h, \text{Hloubka}(s))$
5. Vrátíme $h + 1$.

Úkol 4 [4b]: Vymyslete, jak spočítat *průměr* stromu. Tak se říká délce nejdelší cesty. Pozor, není to totéž jako hloubka stromu, protože nejdelší cesta nemusí vést „shora dolů“ (v příkladu s továrnou jedna taková vede mezi skladníkem a některým z dělníků).

Vyhodnocení výrazu – výraz reprezentovaný stromem můžeme snadno vyhodnotit: kdykoliv se vracíme z vrcholu, spočítáme výsledek příslušného podvýrazu. Pro list vrátíme číslo v něm uložené, ve vnitřních vrcholech vezmeme hodnoty ze synů a aplikujeme na ně operaci uloženou ve vrcholu.

Strážníci – ve „stromovém“ městečku bují zločin (lidé kreslí křídou na chodníky karikatury radních!). Starosta chce na vybrané křižovatky rozestavět strážníky tak, aby každá ulice byla z alespoň jedné strany hlídaná. Jak už ale víme, je to skrblík. Pro každou křižovatku proto zjistil, kolik musí zaplatit strážníkovi, aby tam byl ochoten stát, a hledá celkově nejlevnější rozestavění strážníků.

Formálně řečeno, hledáme podmnožinu vrcholů stromu, která z každé hrany obsahuje alespoň jeden vrchol a navíc je součet *cen* vrcholů v množině nejmenší možný.



Nabízí se opět zapřáhnout DFS a vracet z podstromů, jak nejlaciněji je jde ohlídat. Jenže pak nedovedeme z výsledků pro podstromy zkombinovat výsledek pro celý strom. Pomůže, když si pro každý podstrom místo jednoho čísla spočítáme rovnou dvě. Budeme jim říkat h a o. To první („hlídaná cena“) bude udávat, kolik nejméně stačí zaplatit na ohlídání hran podstromu za podmínky, že v kořeni podstromu bude stát strážník. Naopak o („opuštěná cena“) hlásá minimální cenu za podmínky, že kořen je opuštěný.

V listu je triviálně h rovno ceně listu a $o = 0$ (podstrom nemá žádné hrany, takže není potřeba nic hlídat).

Nyní uvažujme obecný podstrom s kořenem v ceny c_v se syny $s_1, \ldots, s_k$, pro které už známe $h_1, \ldots, h_k$ a $o_1, \ldots, o_k$. Počítejme h: pokud je kořen hlídaný,

jeho strážník ohlídá všechny hrany vedoucí do synů. V synech si proto mužeme vybrat, zda tam umístíme strážníka či nikoliv, takže pro i-tý podstrom postačí $\min(h_i, o_i)$ peněz. Celkově tedy $h = c_v + \sum_{i=1}^{k} \min(h_i, o_i)$.

A nyní o: Jestliže do kořene strážníka nedáme, musí být strážníci ve všech synech (jinak by některá z hran do synů nebyla hlídaná). Proto $o = \sum_{i=1}^{k} h_i$.

Všechna h a o tedy hravě spočítáme pomocí DFS v čase $\mathcal{O}(n)$. Odpověď na starostovu otázku pak získáme jako minimum z h a o kořene.

Strážníci(v):

1. $c_v \leftarrow$ cena vrcholu v
2. Je-li v list, vrátíme $(c_v, 0)$.
3. $h \leftarrow c_v$, $o \leftarrow 0$
4. Pro všechny syny s vrcholu v:
5. $(h_s, o_s) \leftarrow$ Strážníci(s)
6. $h \leftarrow h + \min(h_s, o_s)$
7. $o \leftarrow o + h_s$
8. Vrátíme (h, o).

Úkol 5 [4b]: Strážníci si pořídili drony a dokáží hlídat i „za jeden roh“. Přesněji řečeno, ulice je hlídaná, pokud stojí strážník na alespoň jedné z krajních křižovatek, nebo na křižovatce, která s krajní sousedí. Pomozte najít nejlevnější rozestavění strážníků pro ohlídání všech ulic.

Vandalská indukce a centrum stromu

Ukážeme si ještě jeden způsob, jak zkoumat stromy. Tentokrát je budeme procházet od listů směrem „dovnitř“. Začneme dvěma jednoduchými pozorováními, přičemž *netriviální* budeme říkat stromům s alespoň dvěma vrcholy.

Pozorování 1: Každý netriviální strom má nějaký list.

Důkaz: Strom si zakořeníme v libovolném vrcholu a podíváme se na nejhlubší vrchol (nejvzdálenější od kořene). Ten nemá žádné syny a vede z něj jediná hrana do otce. Proto je to list.

Pozorování 2: Odstraníme-li z netriviálního stromu list, vznikne opět strom.

Důkaz: Je třeba ověřit, že graf zůstal souvislý a bez kružnic. Odstraněním vrcholu jsme sotva mohli vytvořit novou kružnici. Pokud byly nějaké dva neodebrané vrcholy spojené cestou, budou spojené i nadále, protože odebraný list nemůže ležet uvnitř cesty.

Z toho plyne následující, poněkud vandalská, technika indukce: ve stromu najdeme list, odtrhneme ho, čímž získáme další strom. Postup opakujeme, dokud nám nezůstane triviální strom. Jestliže zaznamenanou posloupnost odebírání listů obrátíme, dostali jsme postup, jak z jednoho vrcholu postupným přilepováním listů vytvořit původní strom.

Pokud má ovšem celý algoritmus seběhnout v lineárním čase, nemůžeme listy hledat pokaždé znovu. Budeme si udržovat frontu objevených, ale dosud

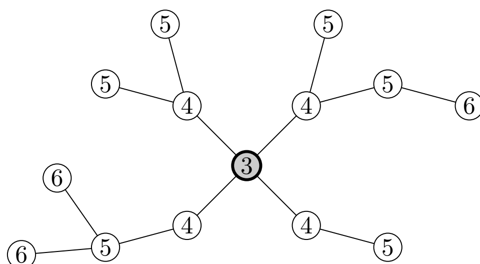
neutržených listů. Na začátku spočítáme stupně všech vrcholů a listy přidáme do fronty. V každém dalším kroku odebereme jeden list z fronty a odstraníme ho ze stromu.

Jeho sousedovi snížíme stupeň o jedničku, a pokud se tento soused stal listem, přidáme ho do fronty. Každý vrchol se dostane do fronty právě jednou a jeho obsluhou strávíme konstantní čas. Celé otrhávání tedy beží v čase $\mathcal{O}(n)$, jak jsme chtěli.

Můžete si zkusit vymyslet, jak tuto „vandalskou indukci“ použít na libovolný z příkladů, které jsme řešili pomocí DFS. Není divu – když se DFS vrací z rekurze, také postupuje od listů ke kořeni. My zde ukážeme, jak najít „střed“ stromu, což se přímo pomocí DFS dělá obtížně.

Definice: Pro libovolný vrchol v zavedeme jeho *excentricitu* (výstřednost) jako maximum ze vzdáleností z v do ostatních vrcholů. (Pokud bychom tedy strom ve v zakořenili, bude nám to říkat, jak hluboko je nejhlubší list.) *Centrum* stromu říkáme množině všech vrcholů s nejmenší možnou excentricitou.

Příklad vidíte na následujícím obrázku. Čísla udávají excentricity, zvýrazněný vrchol je centrem stromu.



Dokážeme, že centrum každého stromu je tvořeno buďto jedním vrcholem, anebo dvěma vrcholy spojenými hranou. Navíc ho lze najít v lineárním čase.

Nejprve si všimneme, že centrum stromu neobsahuje žádný list, neboť soused listu má o jedničku nižší excentricitu. Aby tato úvaha fungovala, nesmí být soused listu také list, takže strom musí mít aspoň 3 vrcholy.

Nyní se nám bude hodit, že odstraněním všech listů se sníží excentricity všech ostatních vrcholů právě o 1. Každá nejdelší cesta z vnitřního vrcholu totiž končila v nějakém listu, tím pádem jsme ji zkrátili o jednu hranu.

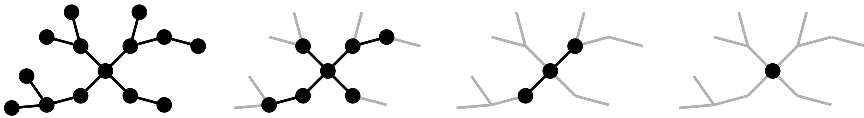
Zkombinujeme-li tyto dvě vlastnosti, zjistíme, že „očesáním“ všech listů dostaneme menší strom, který má stejné centrum. Opakováním tohoto postupu graf „oloupeme“ až na jedno- nebo dvouvrcholový strom. U těch snadno nahlédneme, že jejich centrum obsahuje všechny vrcholy.

K nalézání listů nám opět poslouží fronta. Jen musíme umět rozlišit, kde končí listy z jedné „slupky“ a začínají ty z další. K tomu stačí do fronty přidá-

vat zarážku za konec slupky, případně střídat dvě fronty. V obou případech to stihneme v lineárním čase. Může to vypadat třeba následovně:

1. Založíme dvě prázdné fronty F a G.
2. Všem vrcholům spočítáme stupně.
3. Listy přidáme do F.
4. Dokud strom obsahuje alespoň 3 vrcholy:
5. Dokud je F neprázdná:
6. $v \leftarrow$ další vrchol z F
7. Odebereme v.
8. Pro všechny sousedy s vrcholu v:
9. Snížíme stupeň s o 1.
10. Pokud stupeň klesl na 1, přidáme s do G.
11. Prohodíme F a G.
12. Zbývající vrcholy prohlásíme za centrum.

Pro strom z předchozího obrázku proběhne výpočet takto:



Úkol 6 [3b]: Navrhněte a naprogramujte algoritmus, který v daném stromu spočítá excentricity všech vrcholů.

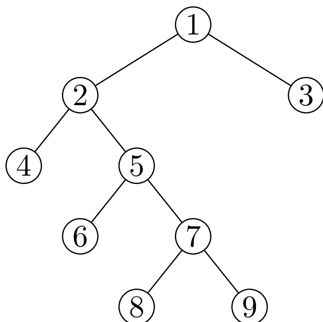
Martin „Medvěd“ Mareš

29-2-7 Strom ve stromu 15 bodů

V druhém dílu našeho stromového seriálu ukážeme, jak popisovat podstromy pomocí DFS očíslování. Hlavně si ale předvedeme, jak vytvářet datové struktury pro stromy tím, že daný strom uložíme do úplně jiného stromu, totiž intervalového.

DFS očíslování

Začněme opakováním z minula. Spustíme-li na zadaném stromu prohledávání do hloubky (DFS), můžeme jeho průběh popsat posloupností levých a pravých závorek: levou zapíšeme, kdykoliv shora vstoupíme do vrcholu, pravou, jakmile se chystáme vystoupit nahoru.



Pro strom na obrázku vyjde posloupnost závorek `((()(()(()())))())`. Každému vrcholu jsme takto přidělili jeden pár závorek a tyto páry se navzájem nekříží (říkáme, že tvoří dobré uzávorkování).

Navíc si ke každému vrcholu v zapamatujeme čísla $in(v)$ a $out(v)$. Ta budou říkat, na kolikáté pozici v řetězci závorek se nachází levá, resp. pravá závorka odpovídající vrcholu v. Pro náš ukázkový strom to vyjde takto:

v	1	2	3	4	5	6	7	8	9
$in(v)$	1	2	16	3	5	6	8	9	11
$out(v)$	18	15	17	4	14	7	13	10	12

Můžeme si to představit také tak, že máme nějaké hodiny (počítadlo), které odtikávají jednotlivé kroky DFS, a hodnoty *in* a *out* pro daný vrchol udávají čas prvního vstupu a posledního výstupu.

Všechny *in*y a *out*y dokážeme spočítat v lineárním čase. Potom nám prozradí ledacos zajímavého o tvaru stromu. Například pomocí nich můžeme poznat vzájemný vztah dvou vrcholů. Představme si nějaké dva vrcholy u a v:

- Pokud u je potomkem v (ať už přímým či nepřímým), musí být pár závorek patřících k u uvnitř toho od v, takže musí platit

$$in(v) < in(u) < out(u) < out(v).$$

- Pokud je naopak v potomkem u, dostáváme

$$in(u) < in(v) < out(v) < out(u).$$

- V ostatních případech musí jeden pár závorek skončit, než druhý začne (páry se přeci nekříží), takže nastane jedna z těchto možností:

$$in(u) < out(u) < in(v) < out(v),$$
$$in(v) < out(v) < in(u) < out(u).$$

Dovedeme tedy v konstantním čase zjistit, jaký je „příbuzenský vztah“ u a v.

Nestromové hrany

Občas se potkáme s případy, kdy mezi vrcholy stromu vedou ještě nějaké další hrany, které nejsou přímo součástí stromu (jako když vánoční stromeček ozdobíme řetězy). Těmto hranám říkáme *nestromové* a často nás zajímá, mezi jakými částmi stromu vedou.

Úkol 1 [2b]: Je dán strom na n vrcholech a m nestromových hran. Předpočítejte v čase $\mathcal{O}(n+m)$ tabulku a pak pomocí ní v konstantním čase odpovídejte na dotazy typu „vede nějaká nestromová hrana ven z podstromu s kořenem v?“.

Intervalové stromy

Brzy se nám bude hodit uložit všechno, co víme o daném stromu, do šikovné datové struktury – překvapivě opět stromové (její struktura má ale s podobou původního stromu pramálo společného).

Intervalový strom je datová struktura, která si pamatuje nějakou posloupnost $x_1, \ldots, x_m$ a umí s ní provádět následující operace:

- *Init*$(x_1, \ldots, x_n)$ – *inicializace* – $\mathcal{O}(n)$
  vytvoří nový strom se zadanými hodnotami
- *Get*(i) – *bodový dotaz* – $\mathcal{O}(\log n)$
  vrátí aktuální hodnotu x_i
- *Set*(i, t) – *bodový update* – $\mathcal{O}(\log n)$
  nastaví x_i na hodnotu t
- *RangeMin*(i, j) – *intervalový dotaz* – $\mathcal{O}(\log n)$
  spočítá minimum z $x_i, x_{i+1}, \ldots, x_j$
- *RangeAdd*(i, j, δ) – *intervalový update* – $\mathcal{O}(\log n)$
  přičte ke všem prvkům $x_i, \ldots, x_j$ hodnotu δ

Existuje mnoho verzí intervalových stromů. Ty nejjednodušší popsané v naší kuchařce[21] neumí intervalový update, ale zato dokáží provádět bodové dotazy v konstantním čase. Nám se bude více hodit pokročilejší podoba s líným vyhodnocováním změn. Ta už zvládne všechny operace. Detaily si prosím přečtěte v Medvědově knížce,[22] v kapitole o datových strukturách.

Operace intervalového stromu lze navíc snadno ohýbat, aby počítaly něco trochu jiného. Intervalové dotazy mohou kromě maxima počítat třeba minimum nebo součet; intervalový update může například najednou nastavit všechny prvky v intervalu na novou hodnotu. Princip zůstává stejný. (Kdykoliv ale při řešení seriálu budete potřebovat nějakou atypickou operaci, nestačí si ji vymyslet – je nezbytné popsat, jak přesně ty standardní upravíte.)

---

[21] `http://ksp.mff.cuni.cz/viz/kucharky/intervalove-stromy`

[22] `http://mj.ucw.cz/vyuka/ads/`

Strom ve stromu

Pojďme si hrát. Dostaneme nějaký strom T, v jehož každém vrcholu je uloženo jedno číslo, na počátku nulové. Chceme umět provádět dvě operace: změnit číslo uložené ve vrcholu a zjistit minimum ze všech čísel uložených v zadaném podstromu.

To je něco podobného, jako umí intervalové stromy, ovšem s podstromy namísto intervalů. Pojďme jim tedy vstup trochu „předžvýkat". Strom „rozvineme" do posloupnosti podle toho, jak jím prochází DFS. Pro každý vrchol v definujeme $x_{in(v)}$ jako číslo uložené ve v a $x_{out(v)} = +\infty$.

Tím vznikla posloupnost délky $2n$. V ní podstrom ležící pod vrcholem v odpovídá intervalu $x_{in(v)}, \ldots, x_{out(v)}$. Pro výpočet minima podstromu tedy stačí položit intervalový dotaz (nekonečna v *out*-ech výsledek nijak neovlivní). Změna hodnoty vrcholu v je triviální: požádáme intervalový strom o bodový update prvku $x_{in(v)}$, na $x_{out(v)}$ není třeba sahat.

Obě operace tedy pracují v čase $\mathcal{O}(\log n)$, jen nesmíme zapomenout, že jsme také spotřebovali čas $\mathcal{O}(n)$ na vytvoření intervalového stromu.

Ukázali jsme tedy, jak pomocí DFS očíslování překládat strom na posloupnost, přičemž podstromy se přeloží na intervaly v posloupnosti. S těmi se pak často hodí zacházet pomocí nějakého druhu intervalového stromu. Pojďme si to vyzkoušet na dalších příkladech…

Úkol 2 [4b]: Vytvořte datovou strukturu pro strom s obarvenými vrcholy. Na počátku jsou všechny vrcholy zelené. Chceme umět provádět následující operace: *SetColor*(v, c) – nastaví barvu vrcholu v na červenou, zelenou nebo modrou; *CountColor*(v, c) – zjistí, jaká barva je v podstromu pod vrcholem v nejčastější (je-li to nerozhodně, odpoví, že nerozhodně).

Úkol 3 [6b]: Ještě jedna datová struktura. Na začátku dostaneme strom s vrcholy ohodnocenými přirozenými čísly, Chceme umět operaci *Touch*(v), která v podstromu pod v provede následující: nalezne minimum z nenulových čísel ve vrcholech, toto minimum od všech nenulových čísel odečte a nakonec ohlásí, jestli už se povedlo všechna čísla v podstromu vynulovat.

Dvojrozměrné intervalové stromy

Další zajímavé triky můžeme provádět s dvojrozměrnými intervalovými stromy. Do těch ukládáme dvojice čísel, které si můžeme představovat jako body v rovině. Roli intervalů pak hrají libovolné obdélníky.

Pro naše účely postačí velmi jednoduchá podoba 2D intervalových stromů, která umí takovéto operace:

- *Init*$((x_1, y_1), \ldots, (x_n, y_n))$ – *inicializace* – $\mathcal{O}(n \log n)$
  vytvoří nový strom s body o zadaných souřadnicích

- *RangeCount*$(x_{min}, x_{max}, y_{min}, y_{max})$ – *obdélníkový počítací dotaz* – $\mathcal{O}(\log^2 n)$ spočítá, kolik ze zadaných bodů leží v daném obdélníku, tedy pro kolik různých i je $x_{min} \le x_i \le x_{max}$ a současně $y_{min} \le y_i \le y_{max}$.

Jak takový 2D strom sestrojit, najdete například ve vzorovém řešení úlohy 24-4-7,[23] dokonce včetně mazaného triku, který zrychlí intervalový dotaz na $\mathcal{O}(\log n)$.

V následujícím úkolu nás ale konkrétní implementace nemusí zajímat, stačí použít 2D strom jako černou skříňku.

Úkol 4 [3b]: Dostaneme strom s nestromovými hranami. Chceme si něco předpočítat tak, abychom uměli rychle odpovídat na dotazy typu „existuje nestromová hrana mezi podstromem pod u a podstromem pod v?“.

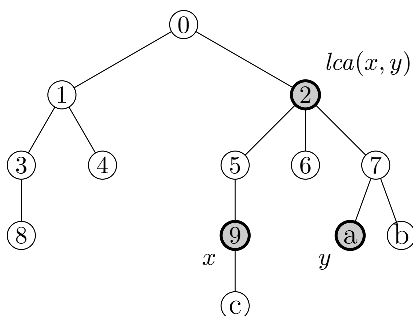
Martin „Medvěd“ Mareš

29-3-7 Stromoví předci 15 bodů

Seriál o stromech pokračuje dalším dílem, tentokrát o hledání (pra)předků vrcholů a užitečné technice zdvojování. Z předchozích dílů se nám bude hodit prohledávání do hloubky s DFS očíslováním a také intervalové stromy. Pokud si už nepamatujete, jak fungují, zalistujte minulými sériemi.

Společní předci (LCA)

Jako červená nit se naším dnešním vyprávěním povine následující problém: Dostaneme nějaký zakořeněný strom a dva jeho vrcholy x a y. Chceme najít jejich *nejbližšího společného předka,* tedy nejhlubší vrchol, který je jak předkem x, tak předkem y. Značit ho budeme $lca(x, y)$ podle anglického *lowest common ancestor.*



Elementární řešení by mohlo vypadat třeba tak, že se nejprve vydáme z x do kořene a označíme všechny vrcholy, přes které jsme prošli. Pak se do kořene

[23] http://ksp.mff.cuni.cz/viz/24-4-7/reseni

vydáme pro změnu z y a první označený vrchol, na nějž narazíme, prohlásíme za společného předka.

To je snadný algoritmus, ale v nejhorším případě spotřebuje $\mathcal{O}(n)$ času, kde n jako obvykle značí počet vrcholů stromu. Je to hodně, nebo málo? Pokud by nám stačilo najít společného předka pro jednu dvojici vrcholů, je to málo. Často ale potřebujeme hledat společné předky pro více dvojic a tam už by nás algoritmus byl příliš pomalý. Za chvíli se to naučíme dělat efektivněji. Ovšem teď je čas na první úkol.

Úkol 1 [1b]: Upravte algoritmus pro hledání společného předka značkováním tak, aby doběhl v čase $\mathcal{O}(d_x + d_y)$, kde d_x je počet hran mezi vrcholem x a společným předkem a podobně d_y. Můžete předpokládat, že strom už máte načtený v paměti.

(Pra)$^{\mathbf{k}}$předci a skočky

Na chvilku odbočme k jinému, příbuznému problému. Máme zakořeněný strom, jehož každý vrchol v si pamatuje svého otce $P(v)$. Pokud je v kořen, položíme $P(v) = \emptyset$. Dědeček vrcholu v je pak přirozeně $P(P(v))$, pradědeček $P(P(P(v)))$ atd. Obecně můžeme definovat k-tého předka $\mathit{Pra}(v,k)$ jako k-tý vrchol na cestě od v do kořene. Tedy $\mathit{Pra}(v,0)$ je v sám, $\mathit{Pra}(v,1)$ jeho otec, $\mathit{Pra}(v,2)$ dědeček a obecně $\mathit{Pra}(v,k+1) = P(\mathit{Pra}(v,k))$. Bude se nám též hodit, že platí $\mathit{Pra}(v,i+j) = \mathit{Pra}(\mathit{Pra}(v,i),j)$.

Počítat k-tého prapředka podle definice trvá $\mathcal{O}(k)$. Ukážeme zajímavý předvýpočet, s nímž to půjde rychleji.

Jistě si můžeme předpočítat všechna $\mathit{Pra}(v,k)$, ale uznejte, že by to trvalo neúnosně dlouho, totiž $\mathcal{O}(n^2)$. Raději si výsledky zapamatujeme jen pro ta k, která jsou mocninami dvojky. Pak vymyslíme, jak z nich dopočítat všechno ostatní.

Pořídíme si tabulku S definovanou předpisem $S(v,i) = \mathit{Pra}(v,2^i)$ pro $i = 0,\ldots,\lfloor \log_2 n \rfloor$. Jistě pro ni platí

$$\begin{aligned} S(v,0) &= \mathit{Pra}(v,1) = P(v), \\ S(v,i+1) &= \mathit{Pra}(v,2^{i+1}) = \mathit{Pra}(v,2^i+2^i) = \\ &= \mathit{Pra}(\mathit{Pra}(v,2^i),2^i) = S(S(v,i),i). \end{aligned}$$

Celou tabulku tedy můžeme snadno spočítat při průchodu stromem do hloubky nebo do šířky: kdykoliv vstoupíme do nějakého vrcholu v, spočítáme $S(v,i)$ pro všechna i. Využijeme k tomu hodnoty S v předcích vrcholu v, které už jsou tou dobou spočítané. V každém vrcholu strávíme čas $\mathcal{O}(\log n)$, celkem tedy $\mathcal{O}(n \log n)$.

Hodnotám v tabulce se říká *jump pointery,* protože nám umožňují přeskočit přes více předků najednou. Česky bychom takové zpětné hraně skákající přes několik pater mohli říkat třeba *skočka.*

Pro strom z předchozího obrázku by skočky vypadaly následovně:

v	0	1	2	3	4	5	6	7	8	9	a	b	c
$S(v,0)$	∅	0	0	1	1	2	2	2	3	5	7	7	9
$S(v,1)$	∅	∅	∅	0	0	0	0	0	1	2	2	2	5
$S(v,2)$	∅	∅	∅	∅	∅	∅	∅	∅	∅	∅	∅	∅	0

Pojďme si teď rozmyslet, jak pomocí skoček skákat do libovolné výšky. Chceme-li zjistit $Pra(v,k)$, zapíšeme k ve dvojkové soustavě jako $2^{i_1}+2^{i_2}+\ldots+2^{i_t}$ a pak vyhodnotíme $S(\ldots S(S(v,i_1),i_2),\ldots),i_t)$. Jelikož dvojkový zápis má nejvýše $\log_2 n + 1$ bitů, zvládneme celý výpočet v čase $\mathcal{O}(\log n)$.

Naprogramovat bychom to mohli například takto:

$Pra(v,k)$:
1. Pro $i = \lfloor \log_2 n \rfloor, \ldots, 1, 0$:
2. Je-li $k \geq 2^i$:
3. $k \leftarrow k - 2^i$
4. $v \leftarrow S(v,i)$
5. Vrátíme výsledek v.

Každý průchod cyklem opravdu zvládneme v čase $\mathcal{O}(1)$: dvojkový logaritmus si můžeme uložit při budování S, mocniny 2^i snadno získáme bitovými posuny (v Céčku `1 << i`).

Umíme si tedy v čase $\mathcal{O}(n \log n)$ pořídit datovou strukturu, která dokáže na dotazy odpovídat v čase $\mathcal{O}(1)$. Krátce budeme říkat, že je to struktura se složitostí $\mathcal{O}(n \log n)/\mathcal{O}(1)$.

Úkol 2 [3b]: Mějme strom s hranami ohodnocenými celými čísly. Předpočítejte něco podobného skočkám, co bude umožňovat vypočítat minimum z ohodnocení hran na libovolné „svislé" cestě, tedy cestě mezi určeným vrcholem a jeho zadaným (pra)předkem. Dosáhněte složitosti $\mathcal{O}(n \log n)/\mathcal{O}(\log n)$.

Úkol 3 [2b]: Ukažte, že budeme-li se ptát na součet místo minima, lze předchozí úkol zrychlit na $\mathcal{O}(n)/\mathcal{O}(1)$.

LCA skočkami

Pojďme se vrátit k hledání společných předků. Předpokládejme, že jsme si pro zadaný strom předpočítali hloubky vrcholů $d(v)$ a všechny skočky.

Nejprve ukážeme, že stačí umět spočítat $lca(x,y)$ v případech, kdy x a y jsou stejně hluboko. Kdyby totiž byl (řekněme) vrchol x hlouběji než y, stačí x nahradit jeho předkem v hloubce $d(y)$ a výsledek se nezmění. Jinými slovy pokud $d(x) > d(y)$, pak

$$lca(x,y) = lca(Pra(x, d(y) - d(x)), y).$$

Stačí se tedy zabývat případy, kdy $d(x) = d(y)$. Pojďme najít, o kolik hladin výše leží nejhlubší společný předek. Hledáme tedy nejmenší takové k, pro

které je $Pra(x,k) = Pra(y,k)$. To můžeme najít následující modifikací binárního vyhledávání.

Předpokládajme, že vzdálenost ke společnému předkovi leží v intervalu $\langle 0, h\rangle$ (na počátku volíme třeba $h = n$). Zkusíme se podívat do vzdálenosti $\ell = h/2$. Spočítáme $x' = Pra(x,\ell)$ a $y' = Pra(y,\ell)$. Je-li $x' = y'$, pak víme, že nejhlubší společný předek leží ve vzdálenosti nejvýše ℓ. Jsou-li naopak x' a y' různé, víme, že $lca(x',y')$ je totéž jako $lca(x,y)$. Proto můžeme x a y nahradit dvojicí x' a y', čímž jsme se ke společnému předkovi přiblížili na vzdálenost nejvýše $h-\ell \leq h/2$.

V obou případech jsme tedy interval zmenšili dvakrát, takže po $\mathcal{O}(\log n)$ krocích už bude nejbližší společný předek přímo otcem x i y. Každý krok přitom zahrnuje dva výpočty funkce *Pra*, což obecně trvá logaritmicky dlouho. Celý výpočet proto potrvá $\mathcal{O}(\log^2 n)$. Pokud ovšem budeme h volit jako mocninu dvojky, všechna ℓ během výpočtu budou také mocniny dvojky, takže všechna potřebná *Pra* budou přímo skočky. Tím jsme časovou složitost snížili na $\mathcal{O}(\log n)$.

V pseudokódu to vyjde velmi jednoduše:

$lca(x,y)$:
1. Pokud $x = y$, vrátíme výsledek x.
2. Pokud $d(x) < d(y)$, prohodíme x a y.
3. Pokud $d(x) > d(y)$, položíme $x \leftarrow Pra(x, d(x) - d(y))$.
4. Pro $i = \lfloor \log_2 n \rfloor, \ldots, 0$:
5. $\quad x' \leftarrow S(x,i)$, $y' \leftarrow S(y,i)$
6. $\quad$ Pokud $x' \neq y'$: $x \leftarrow x'$, $y \leftarrow y'$.
7. Vrátíme výsledek $S(x,0)$.

Zkusme si to na stromu z úvodního obrázku. Kdybychom hledali $lca(\mathrm{c},\mathrm{a})$, po kroku 3 by bylo $x = 9$ a $y = \mathrm{a}$. Následně by pro všechna $i > 0$ vyšlo $S(x,i) = S(y,i)$, takže by se x ani y dlouho neměnily. Až v posledním průchodu s $i = 0$ bychom přešli do $x = 5$, $y = 7$. Nakonec bychom provedli poslední krůček do společného předka 2.

Úkol 4 [1b]: Opět mějme strom s celočíselně ohodnocenými hranami. Chceme umět spočítat minimum či součet na cestě mezi libovolnými dvěma zadanými vrcholy.

ET-posloupnosti

Společní předci se dají počítat i jinak. Strom projdeme do hloubky a kdykoliv projdeme vrcholem, zaznamenáme si tento vrchol a jeho hloubku. Tím vznikne takzvaná ET-posloupnost vrcholů (kdepak mimozemšťané, je to podle anglického *Euler Tour sequence,* neb poslupnost souvisí i s eulerovskými tahy). Pro strom z obrázku by vypadala takto:

vrchol	**0**	**1**	**3**	**8**	3	1	**4**	1	0	**2**	**5**	**9**	**c**	9	5	2	**6**	2	**7**	**a**	7	**b**	7	2	0
hloubka	0	1	2	3	2	1	2	1	0	1	2	3	4	3	2	1	2	1	2	3	2	3	2	1	0

Chvíli meditujme nad vlastnostmi ET-poslupnosti. Vrchol o s synech se v ní bude nacházet právě $(s+1)$-krát: jednou do něj vstoupíme shora a pak znovu po návratu z každého ze synů. Libovolný jeden z těchto výskytů prohlásíme za *hlavní výskyt.* V příkladu jsme za hlavní volili nejlevější výskyty a vyznačili jsme je tučně.

Jak dlouhá je celá posloupnost, spočítáme také snadno: DFS projde každou hranou dvakrát a pokaždé do posloupnosti zapíše jeden vrchol. Jelikož hran je $n-1$, zapíše takto $2n-2$ vrcholů. Nesmíme ale zapomenout na kořen stromu, do nějž jsme poprvé nepřišli po hraně, takže délku posloupnosti opravíme na $2n-1$.

ET-posloupnost tedy vytvoříme v čase $\mathcal{O}(n)$. Samotný výpočet $lca(x,y)$ pak bude přímočarý: najdeme hlavní výskyty vrcholů x a y v ET-poslupnosti a ze všech vrcholů ležících mezi nimi vybereme ten, jehož hloubka je nejmenší. V našem příkladu tedy hledáme minimum v podtrženém úseku posloupnosti.

Proč to funguje? DFS navštíví nejdříve společného předka (říkejme mu p), pak se vydá k jednomu ze zadaných vrcholů (řekněme k x), pak se vrátí do p, projde případné další potomky p, načež sestoupí do y, aby se z něj časem vrátil opět do p. Mezi návštěvami x a y je tedy aspoň jedna navštěva p a nemohli jsme navštívit žádný vrchol vyšší než p, neboť k nim se dostaneme až po definitivním opuštění p. Navíc nezáleží na tom, které výskyty jsme si zvolili jako hlavní, protože mezi každými dvěma výskyty téhož vrcholu projde DFS pouze nějaké jeho potomky.

Úkol 5 [4b]: Uvažme strom s ohodnocenými hranami a jeho ET-posloupnost, do které tentokrát zapisujeme hrany. Při průchodu hranou směrem dolů píšeme ohodnocení této hrany, při průchodu nahoru totéž s opačným znaménkem. Ukažte, jak pomocí této posloupnosti spočítat součet ohodnocení hran na cestě mezi vrcholem a jeho potomkem.

LCA a RMQ

Převedli jsme tedy problém LCA na hledání nejmenšího prvku v zadaném úseku posloupnosti. Obecněji řečeno: Známe nějakou posloupnost čísel $x_1, \ldots, x_n$, chceme pro ni něco předpočítat a pak rychle odpovídat na dotazy typu „které x_i je nejmenší v úseku $x_i, x_{i+1}, \ldots, x_j$". Tato úloha je známá pod názvem RMQ (*Range Minimum Query*) a existuje na ni přehršel různých algoritmů. Aby se nám o ni lépe vyprávělo, budeme mluvit o hledání minima, i když ve skutečnosti budeme hledat *polohu* minima, nejen jeho hodnotu.

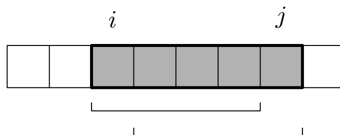
Jak na RMQ? Můžeme například použít intervalové stromy z minulého dílu. V čase $\mathcal{O}(n)$ vytvoříme pro naši poslupnost intervalový strom, jehož vnitřní vrcholy si budou pamatovat, kde v příslušném podstromu leží minimum. Minimum z obecného úseku pak vyhodnotíme v čase $\mathcal{O}(\log n)$. Tak získáme datovou strukturu pro LCA pracující v čase $\mathcal{O}(n)/\mathcal{O}(\log n)$.

Čas na dotaz můžeme ještě snížit za cenu zpomalení předvýpočtu. Předvýpočet odpovědí pro všechny možné dotazy rovnou zavrhneme, trval by $\mathcal{O}(n^2)$.

Ale nabízí se provést podobný trik jako u skoček: předpočítat minima všech úseků délky 2^k, ať už začínají kdekoliv. Budeme počítat tabulku M velikosti $n \times \log n$, kde $M(i,k)$ je minimum úseku $x_i, x_{i+1}, \ldots, x_{i+2^k-1}$. Tuto tabulku můžeme vyplnit v čase $\mathcal{O}(n \log n)$ tak, že minimum každého úseku spočítáme z minim jeho polovin:

1. Pro $i = 1, \ldots, n$:
2. $\quad M(i,0) = x_i$
3. Pro $k = 1, \ldots, \lfloor \log_2 n \rfloor$:
4. $\quad$ Pro $i = 1, \ldots, n - 2^k + 1$:
5. $\qquad M(i,k) = \min(M(i,k-1), M(i+2^{k-1}, k-1))$

Dobrá, máme tabulku. Nyní přijde dotaz na nějaký úsek $x_i, \ldots, x_j$. Zaokrouhlíme délku tohoto úseku dolů na nejbližší mocninu dvojky (tedy najdeme největší k takové, že $2^k < j - i + 1$). Uvážíme dva úseky velikosti 2^k: jeden bude přiražený k začátku našeho dotazu, druhý ke konci. Všimněte si, že pro tyto úseky už minima známe a navíc oba úseky společně pokrývají celý dotaz, byť některé prvky dvakrát. To ovšem nevadí, protože do mininima můžeme prvek započítat, kolikrát chceme.



Stačí tedy spočítat minimum z $M(i,k)$ a $M(j-2^k+1, k)$. To jistě zvládneme v konstantním čase, jen musíme dořešit, kde rychle seženeme největší 2^k menší než délka úseku. To je v podstatě celočíselný dvojkový logaritmus. Váš procesor na něj nejspíš má instrukci, ale i kdyby ji neměl, pomoc je snadná: máme dost času na to, abychom si předpočítali tabulku logaritmů čísel 1 až n.

Tak získáme datovou strukturu pro RMQ, a tedy i LCA, pracující v čase $\mathcal{O}(n \log n)/\mathcal{O}(1)$.

(Krátké zamyšlení: jak se tato technika liší od intervalových stromů? Ty si také pamatují minima všech intervalů délky mocniny dvojky, ovšem jenom těch „správně zarovnaných“, tedy začínajících na násobku své délky. My si pamatujeme i ty nezarovnané, takže umíme obecný úsek pokrývat dvěma intervaly namísto logaritmického počtu.)

Dodejme, že existuje ještě rychlejší struktura. Funguje v čase $\mathcal{O}(n)/\mathcal{O}(1)$ a je mnohem magičtější. Kdybyste se chtěli příslušné kouzlo naučit, najdete ho v knížce Krajinou grafových algoritmů,[24] v kapitole o dekompozici stromů.

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

Úkol 6 [4b]: Navrhněte datovou strukturu pro následující problém: máme vrchol x a nějakého jeho předka p. Chceme zjistit, který ze synů vrcholu p leží „směrem k x", tedy na cestě z p do x. Můžete předpokládat, že máte k dispozici strukturu pro RMQ se složitostí $\mathcal{O}(n \log n)/\mathcal{O}(1)$.

Martin „Medvěd" Mareš

KSP

29-4-7 Rozebíráme stromy 15 bodů

seriál

Vítejte u dalšího dílu stromového seriálu. Tentokrát se zaměříme na *cestové operace.* Tím myslíme takové, které dostanou dva vrcholy stromu a mají něco provést s cestou mezi nimi. Třeba zjistit, jak je tato cesta dlouhá, nebo najít na ní hranu s největším ohodnocením.

Pro mělké stromy je to snadné: hledáme-li cestu mezi vrcholy x a y, stačí z obou vrcholů stoupat směrem ke kořeni, až se obě cesty poprvé protnou v nejbližším společném předchůdci p (to jsme prozkoumali v minulém dílu). Hledanou cestu pak můžeme poskládat ze dvou částí: cesty mezi x a p a cesty mezi y a p. Vše zvládneme v čase lineárním s hloubkou stromu.

Snadné to je i pro stromy, které se vůbec nevětví, tedy samy mají tvar cesty. V druhém dílu jsme se naučili přimět intervalové stromy, aby v logaritmickém čase vyhodnocovaly dotazy na libovolné intervaly posloupnosti. Můžeme si tedy pořídit intervalový strom pro posloupnost hran na cestě a intervaly pak budou odpovídat jejím podcestám. Dokonce pomocí líného vyhodnocování zvládneme rychle měnit ohodnocení hran na podcestě.

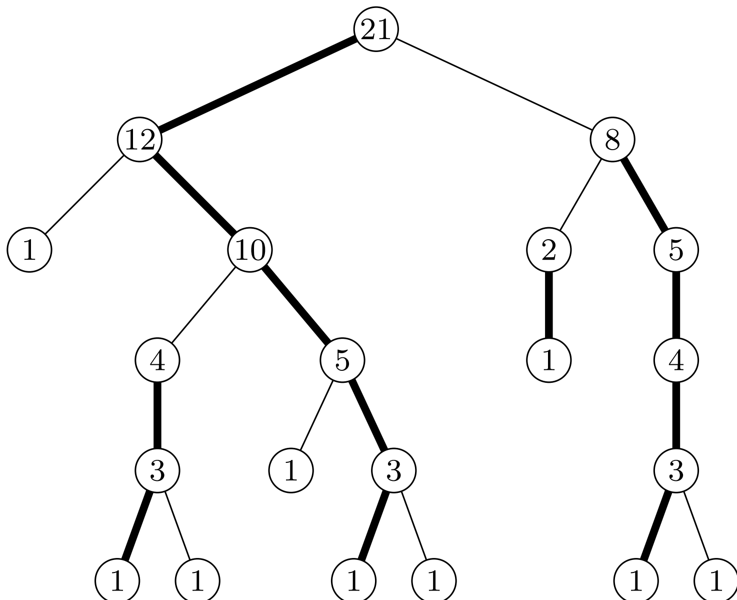
V tomto dílu si ukážeme, jak tyto dvě techniky spojit. Zavedeme takzvanou *dekompozici stromu na lehké a těžké hrany* neboli *heavy-light dekompozici*. Ta nám pomůže k rychlému vyhodnocování cestových dotazů v libovolně hlubokém a libovolně košatém stromu. Jen se nám strom nebude smět pod rukama měnit, tedy až na ohodnocení vrcholů a hran.

Heavy-light dekompozice (HLD)

Nejprve pár definic. Mějme nějaký zakořeněný strom. Označíme $T(v)$ *podstrom* složený z vrcholu v a všech jeho (i nepřímých) potomků. *Velikostí podstromu* míníme počet jeho vrcholů a budeme ji značit $size(v)$.

Hrany z každého vrcholu do jeho synů rozdělíme na *lehké* a *těžké* následovně: hranu do největšího podstromu prohlásíme za těžkou, všechny ostatní za lehké. Existuje-li více největších podstromů, vybereme si libovolný jeden z nich.

Jak to dopadne pro jeden konkrétní strom, vidíme na následujícím obrázku. Čísla ve vrcholech jsou jejich *size.*



Těžké hrany jsou na obrázku vyznačeny tučně a zjevně tvoří cesty. To není náhoda, platí to v každém stromu. Stačí si uvědomit, že z každého vrcholu může vést dolů nejvýše jedna těžká hrana. (Dokonce víme, že není-li vrchol list, vede z něj právě jedna taková.)

Navíc platí, že lehkých hran není nikdy mnoho za sebou. Přesněji řečeno, na cestě z kořene do libovolného vrcholu v leží nejvýše $\log_2 n$ lehkých hran (n jako obvykle značí velikost celého stromu).

Pojďme to dokázat. Vydejme se z kořene do v a sledujme, jak se mění *size* aktuálního vrcholu. Za chvíli uvidíme, že kdykoliv projdeme po lehké hraně, klesne *size* aspoň dvakrát. Po k lehkých hranách tedy klesne aspoň 2^k-krát, takže kdyby bylo $k > \log_2 n$, nezbyly by v podstromu žádné vrcholy.

Dobrá, proč tedy na lehké hraně *size* tolik klesá? Uvažme lehkou hranu z nějakého vrcholu u do jeho syna ℓ. Z definice musí existovat i těžká hrana do jiného syna t a platí $\mathit{size}(t) \geq \mathit{size}(\ell)$. Jenže podstromy $T(\ell)$ a $T(t)$ jsou součástí $T(u)$, takže $\mathit{size}(u) > \mathit{size}(t) + \mathit{size}(\ell)$ a z toho ihned $\mathit{size}(u) > 2\,\mathit{size}(\ell)$.

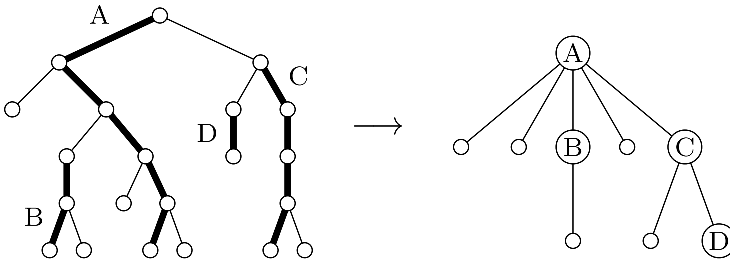
Pojďme zopakovat, co jsme zjistili:

- Heavy-light dekompozice rozkládá strom na *těžké cesty*, které jsou propojené *lehkými hranami.*
- Každý vrchol leží na právě jedné těžké cestě. (Tedy připustíme-li i cesty složené z jediného vrcholu.)

- Každá lehká hrana vede z vrcholu nějaké těžké cesty do nejvyššího vrcholu jiné těžké cesty.
- „Lehká hloubka“ je logaritmická. Přesněji řečeno, mezi každými dvěma těžkými cestami leží $\mathcal{O}(\log n)$ lehkých hran.

Úkol 1 [2b]: Někdy se používá jiná definice těžkých hran: hrana z vrcholu v do syna s je těžká, pokud $size(s) > size(v)/2$. Rozmyslete, jak se takto definovaná dekompozice bude lišit od té naší. Překreslete podle toho předchozí obrázek.

Dekompozici si můžeme představit i tak, že každou těžkou cestu zkontrahujeme do jediného vrcholu. Tím dostaneme jiný strom, v němž zbudou jen původní lehké hrany a bude logaritmicky hluboký. Jak vypadá, je vidět na následujícím obrázku. Vrcholy bez písmenek odpovídají triviálním (jednovrcholovým) těžkým cestám.



Výpočet dekompozice

Nyní se podíváme, jak HLD reprezentovat v paměti a zejména jak ji rychle sestrojit.

V každém vrcholu v našeho stromu si budeme pamatovat:

- $size(v)$ – velikost podstromu pod v
- $hson(v)$ – do kterého syna vede těžká hrana (nebo $\emptyset$, pokud žádný není, což je možné jen tehdy, je-li v list)
- $path(v)$ – odkaz na těžkou cestu, na níž vrchol leží
- $index(v)$ – kolikátý v pořadí na těžké cestě je (číslujeme od nuly od spodního vrcholu cesty)

Těžké cesty si pamatujeme bokem. Pro cestu p uložíme:

- $lparent(p)$ – otec *kořene cesty* (tak budeme říkat jejímu nejvyššímu vrcholu, tedy vrcholu s nejvyšším indexem); tedy vrchol, z nějž vede do kořene lehká hrana. Může být $\emptyset$, pokud je kořen cesty také kořenem celého stromu.
- $plen(p)$ – délka cesty (počet hran na ní)
- $pvertex(p)$ – pole vrcholů cesty v pořadí jejich indexů

Dekompozici můžeme snadno spočítat dvojím prohledáním do hloubky. Při tom prvním stanovíme velikosti podstromů, rozhodneme, které hrany jsou lehké a těžké, a vypočteme indexy. Druhé sestrojí popisy jednotlivých těžkých cest.

První prohledání vypadá následovně. Spouštíme ho v kořeni stromu a při návratu z rekurze počítá jednotlivé vlastnosti vrcholů podle definice.

$HLD_1(v)$:

1. $size(v) \leftarrow 1$
2. $h \leftarrow \emptyset$ ◁ *kam povede těžká hrana?*
3. Pro všechny syny s vrcholu v:
4. $\quad HLD_1(s)$
5. $\quad size(v) \leftarrow size(v) + size(s)$
6. $\quad$ Pokud $h = \emptyset$ nebo $size(s) > size(h)$:
7. $\quad\quad h \leftarrow s$
8. $hson(v) \leftarrow h$
9. Pokud $h = \emptyset$: ◁ *jsme v listu*
10. $\quad path(v) \leftarrow$ nová těžká cesta
11. $\quad index(v) \leftarrow 0$
12. Jinak: ◁ *pokračování těžké cesty*
13. $\quad path(v) \leftarrow path(h)$
14. $\quad index(v) \leftarrow index(h) + 1$

Druhé prohledávání spouštíme opět z kořene. Jako druhý parametr předáváme otce aktuálního vrcholu, pro kořen je to $\emptyset$. Vždy posbíráme vrcholy na jedné těžké cestě a pak se zavoláme na všechny jejich syny.

$HLD_2(v, o)$:

1. $p \leftarrow path(v)$ ◁ *vrchol v je kořenem těžké cesty p*
2. $lparent(p) \leftarrow o$
3. $plen(p) \leftarrow index(v)$
4. $pvertex(p) \leftarrow$ nové pole délky $plen(p) + 1$
5. Dokud $v \neq \emptyset$: ◁ *procházíme těžkou cestu*
6. $\quad pvertex(p)[index(v)] \leftarrow v$
7. $\quad$ Pro všechny syny s vrcholu v:
8. $\quad\quad HLD_2(s, v)$ ◁ *při tom rekurze na syny*
9. $\quad v \leftarrow hson(v)$ ◁ *pokračujeme po těžké cestě*

Obě prohledání provedou konstantní množství práce pro každý vrchol a hranu stromu, celkem tedy $\mathcal{O}(n)$. Dodejme, že by se všechno dalo zvládnout během jediného DFS, ale ve dvou nám to přijde přehlednější.

Stromoví předchůdci

Abychom si osahali, jak se s HLD zachází, zkusíme ji použít na úlohu hledání nejbližšího společného předchůdce z minulého dílu.

Vzpomeňme si na primitivní algoritmus, který z každého ze zadaných vrcholů prošel po cestě do kořene, značkoval vrcholy a číhal, kde se obě cesty poprvé potkají. Teď na to půjdeme podobně, ale místo jednotlivých vrcholů budeme značkovat najednou celé těžké cesty.

Pojmenujme zadané vrcholy u a v. Nejprve budeme stoupat z u ke kořeni. Kdykoliv jsme v nějakém vrcholu x, z $path(x)$ se dozvíme, na které těžké cestě p se nacházíme. Pak hned vyskočíme z kořene těžké cesty po lehké hraně do vrcholu $lparent(p)$. Ještě si k těžké cestě poznačíme novou položku $enter(p)$ říkající, kudy jsme na cestu vstoupili. Nenavštívené cesty budou mít $enter(p) = \emptyset$.

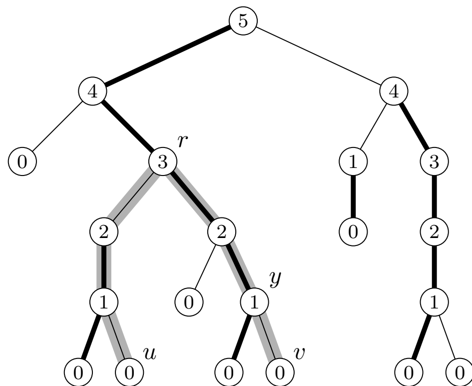
Poté budeme stoupat z v. Stejným způsobem, ale místo značení těžkých cest budeme naopak testovat, zda už nejsou označené. Dříve či později narazíme na těžkou cestu p, na níž jsme už byli při stoupání z u. Přitom víme, kudy jsme se na tuto cestu v obou případech napojili – v jednom místě napojení teď stojíme, druhé máme uložené v $enter(p)$. Hledaným společným předchůdcem je vyšší z těchto dvou míst, což poznáme podle indexů vrcholů.

V pseudokódu to vypadá takto:

$lca(u, v)$:

1. $x \leftarrow u$ ◁ *z u do kořene*
2. Dokud $x \neq \emptyset$, opakujeme:
3. $\quad p \leftarrow path(x)$
4. $\quad enter(p) \leftarrow x$
5. $\quad x \leftarrow lparent(p)$
6. $y \leftarrow v$ ◁ *z v do kořene*
7. Dokud $enter(path(y)) = \emptyset$, opakujeme:
8. $\quad y \leftarrow lparent(path(y))$
9. $r \leftarrow enter(path(y))$ ◁ *místa napojení: r a y*
10. Dokud $u \neq \emptyset$, opakujeme: ◁ *smažeme značky*
11. $\quad enter(path(u)) \leftarrow \emptyset$
12. $\quad u \leftarrow lparent(path(u))$
13. Pokud $index(y) > index(r)$: ◁ *vrátíme vyšší z r a y*
14. $\quad$ Vrátíme y.
15. Jinak:
16. $\quad$ Vrátíme r.

Průběh algoritmu můžeme sledovat na následujícím obrázku. Čísla ve vrcholech jsou jejich indexy, tučně jsou zvýrazněny těžké cesty, šedivě je podbarvena cesta mezi u a v.



Časová složitost funkce *lca* je $\mathcal{O}(\log n)$, neboť při každém stoupání navštíví nejvýše logaritmicky těžkých cest a každou z nich zpracuje v konstantním čase. Získali jsme tedy datovou strukturu pro společné předchůdce, které stačí předvýpočet v čase $\mathcal{O}(n)$ a odpovídá na dotazy v $\mathcal{O}(\log n)$.

Úkol 2 [3b]: Navrhněte algoritmus, který bude pomocí HLD odpovídat na dotazy na *vzdálenost* zadaných dvou vrcholů (tedy počet hran na cestě mezi nimi).

Kudy, kudy cestička?

Nyní se podíváme na to, jak pomocí HLD odpovídat na cestové dotazy. Předvedeme si to na příkladu cestových minim. Dostaneme zadaný strom, jehož hrany budou ohodnoceny celočíselnými *cenami*. Pak nám budou přicházet dotazy na nejlevnější hranu na cestě mezi zadanými vrcholy.

Pro strom si opět spočítáme HLD a každou těžkou cestu navíc opatříme intervalovým stromem, který bude umět odpovídat na intervalová minima cen na této cestě. Ceny těžkých hran si tedy budeme pamatovat v intervalových stromech, zatímco ceny lehkých hran uložíme samostatně. Jeden intervalový strom vybudujeme v čase lineárním v délce jeho těžké cesty a jelikož každý vrchol leží na právě jedné těžké cestě, sestrojíme celou datovou strukturu v čase $\mathcal{O}(n)$.

Nyní přijde dotaz na cestu mezi nějakými vrcholy u a v. Spustíme algoritmus pro $lca(u, v)$ a během výpočtu si zapamatujeme, kterými lehkými hranami a kterými částmi těžkých cest jsme prošli. Pak stačí najít nejlevnější z těchto lehkých hran a minim částí těžkých cest. Lehkých hran je $\mathcal{O}(\log n)$ a každou zpracujeme v konstantním čase, těžkých cest je také $\mathcal{O}(\log n)$ a na každé provádíme intervalový dotaz v čase $\mathcal{O}(\log n)$. Dohromady tedy strávíme čas $\mathcal{O}(\log^2 n)$.

Podobně můžeme provádět cestový update. Pořídíme si intervalové stromy s líným vyhodnocováním, které dovedou intervalový update v logaritmickém čase. Kdykoliv třeba budeme chtít zdražit všechny hrany na nějaké cestě o δ, rozložíme update na zdražení $\mathcal{O}(\log n)$ lehkých hran a $\mathcal{O}(\log n)$ intervalových updatů částí těžkých cest v jejich intervalových stromech. Vše opět stihneme v čase $\mathcal{O}(\log^2 n)$.

Úkol 3 [5b]: Zrychlete cestový dotaz na $\mathcal{O}(\log n)$ za předpokladu, že ceny hran se od inicializace struktury již nezmění. Inicializace by měla stále pracovat v lineárním čase.

Úkol 4 [5b]: Mějme graf s ohodnocenými hranami a jeho minimální kostru.[25] Pro každou hranu, která neleží v kostře, spočítejte, o kolik nejvýše můžeme její ohodnocení snížit, aby kostra stále zůstala minimální.

Martin „Medvěd" Mareš

KSP

29-5-7 Stromy v pohybu 15 bodů

seriál

Vítejte u posledního dílu stromového seriálu. Propukne v něm malá revoluce: chystáme se porušit předpoklad ze všech předchozích dílů, že celý strom známe na začátku výpočtu a pak už jeho tvar zůstává navěky stejný. Postupně vybudujeme datovou strukturu, která bude umět udržovat obecný les a stromy libovolně spojovat a rozdělovat. Bude inspirována Link-Cut Trees od pánů Sleatora a Tarjana.

Opakování vyhledávacích stromů

Podobně jako jsme dříve reprezentovali cesty pomocí intervalových stromů, teď využijeme *binární vyhledávací stromy (BVS).* Pokud jste se s nimi ještě nesetkali, nahlédněte do kuchařky o vyhledávacích stromech.[26]

Aby bylo jasné, kdy mluvíme o původních stromech a kdy o BVS, pomocí nichž původní stromy reprezentujeme, budeme vrcholům BVS říkat *uzly.*

Představme si binární vyhledávací strom, v němž je uložena jistá množina čísel $x_1 < \ldots < x_n$. Pokud tato čísla chceme vypsat od nejmenšího do největšího, můžeme BVS projít rekurzivně v takzvaném in-orderu: kdykoliv vstoupíme do nějakého uzlu u, nejprve rekurzivně projdeme levý podstrom, pak vypíšeme číslo v uzlu u, a nakonec rekurzivně projdeme pravý podstrom.

Nyní k BVS přidáme takzvané *externí uzly:* kdykoliv nějakému uzlu stromu chybí syn, připojíme na místo tohoto syna nový uzel. Strom jsme tedy opatřili ještě jednou vrstvou listů (když si představíte reprezentaci stromu v programu, externí uzly budou na místech původních `NULL` pointerů).

Zajímavou vlastností externích uzlů je, že odpovídají intervalům mezi čísly v *interních* (původních) uzlech. Skutečně: při hledání libovolného čísla z intervalu (x_i, x_{i+1}) dopadnou všechna porovnání v interních uzlech stejně, takže skončíme v tomtéž externím uzlu.

Při in-orderovém průchodu stromem tedy začneme v nejlevějším externím uzlu (ten odpovídá intervalu $(-\infty, x_1)$) a pak se pravidelně střídají interní a externí uzly, až skončíme v nejpravějším externím uzlu, tedy intervalu $(x_n, +\infty)$.

[25] `http://ksp.mff.cuni.cz/viz/kucharky/minimalni-kostry`

[26] `http://ksp.mff.cuni.cz/viz/kucharky/vyhledavaci-stromy`

Často se nám bude hodit najít nejmenší číslo uložené ve stromu. K němu dojdeme tak, že se z kořene vydáme stále doleva. Když už to nejde dál, nacházíme se v minimu: všechny prvky, přes něž jsme prošli, jsou větší, a stejně tak vše, co leží od nich doprava. Časová složitost této operace je zjevně lineární v hloubce stromu.

Úkol 1 [2b]: Vymyslete, jak v BVS nalézt *následníka* zadaného uzlu. Tím myslíme uzel s nejmenším číslem, které je větší než to zadané. Dosáhněte složitosti lineární s hloubkou stromu. Můžete předpokládat, že každý uzel si kromě ukazatelů na své syny pamatuje i ukazatel na otce.

Splay stromy

Jelikož operace s BVS trvají lineárně s hloubkou stromu, musíme stromy udržovat vyvážené – tehdy mají příjemnou logaritmickou hloubku. Existuje mnoho způsobů vyvažování (třeba AVL stromy nebo červeno-černé stromy), my tentokrát zvolíme jeden poněkud netradiční: takzvané splay stromy.

Jejich úplný popis naleznete v Medvědově knížce[27] v kapitole o amortizaci. Zde si vystačíme se základními principy.

Kdykoliv budeme pracovat s nějakým uzlem u, „vyrotujeme" ho do kořene stromu. Této operaci se říká *splayování* uzlu u, a když se udělá správně (viz knížka), zabraňuje degeneraci stromu. Občas se sice může stát, že nějaký uzel bude hodně hluboko, takže přístup k němu bude pomalý. Ale až na něj sáhneme, splayování dlouhou cestu rozkošatí a další operace budou zase rychlé.

Obecně platí, že jedno splayování může trvat až $\mathcal{O}(n)$, ale posloupnost jakýchkoliv k po sobě jdoucích splayování trvá $\mathcal{O}(n \log n + k \log n)$. Dlouhodobě se tedy splayování chová, jako by mělo logaritmickou složitost (říkáme, že je *amortizovaně logaritmické*).

Nyní si rozmyslíme, jak se ve splay stromu hledá minimum. Půjdeme stále doleva dolů, jako v obecném BVS, a až dorazíme do minima, vysplayujeme ho do kořene. Hledání minima trvalo lineárně s hloubkou minima a stejně tak splayování. Ovšem splayování je amortizovaně logaritmické, takže pro hledání minima to musí platit také. (Můžeme si také představit, že jsme průchod jednotlivými hranami shora dolů naúčtovali jejich průchodu zdola nahoru během splayování, čímž jsme splayování zpomalili konstanta-krát.)

Úkol 2 [1b]: Zkombinujte hledání následníka z prvního úkolu se splayováním tak, aby mělo amortizovaně logaritmickou složitost.

Teď zkusíme splay stromy rozdělovat a spojovat. Mějme nějaký uzel u a chceme strom rozdělit na dva stromy: v jednom budou hodnoty menší než v uzlu u, v druhém ty větší. Samotný uzel u zmizí. Zatímco v AVL stromech by to byla docela obtížné, ve splay stromu je to triviální: vysplayujeme u do kořene

[27] `http://pruvodce.ucw.cz/`

a všimneme si, že všechny menší hodnoty jsou momentálně v levém podstromu pod u a všechny větší v tom pravém. Stačí tedy u smazat.

Spojování je ještě jednodušší: dostaneme nějakou hodnotu x a dva stromy – v prvním budou všechny hodnoty menší než x, v druhém větší. Vytvoříme nový uzel s hodnotou x, který se stane kořenem nového stromu. Jako levého syna mu připojíme kořen prvního stromu, jako pravého syna kořen druhého.

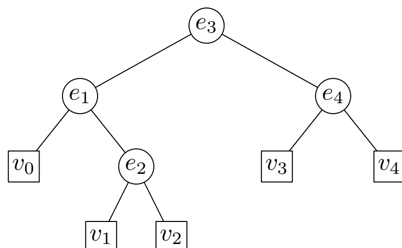
Rozdělování a spojování můžeme například použít ke vkládání a mazání hodnot. Tyto operace ale překvapivě nebudeme potřebovat.

Reprezentace cest

Splay stromy (SS) nyní využijeme k reprezentaci cest. Uvažujme nějakou orientovanou cestu s vrcholy $v_0, \ldots, v_k$, mezi nimiž vedou hrany $e_1, \ldots, e_k$. Vytvoříme BVS s k interními uzly, ve kterých sice nebudou uložena žádná čísla (uvidíme, že to vůbec nevadí), ale jejich in-orderové pořadí bude odpovídat hranám cesty. Pak přidáme externí uzly, které budou odpovídat $k + 1$ vrcholům cesty. Při in-orderovém průchodu tedy budeme navštěvovat postupně

$$v_0, e_1, v_1, e_2, v_2, \ldots, v_{k-1}, e_k, v_k.$$

Cestu se čtyřmi hranami můžeme popsat třeba takto:



Postupně ukážeme, jak v této reprezentaci provádět některé základní operace se souborem cest. Pro každou cestu si pořídíme jeden SS a zapamatujeme si, kterému vrcholu a hraně cesty odpovídá který uzel SS.

Ještě si rozmyslíme okrajové případy: cesta o jedné hraně je reprezentovaná SS s kořenem (to je ta hrana), pod nímž visí dva externí uzly (krajní vrcholy hrany). Jednovrcholová cesta bez hran odpovídá degenerovanému SS, jenž nemá interní uzly a samotný kořen je externí.

Nyní operace:

- *Path*(v) – zjištění, do které cesty patří vrchol v: najdeme odpovídající externí uzel SS a vystoupáme z něj až do kořene SS.
- *First*(p) – nalezení prvního vrcholu dané cesty: stačí najít minimum příslušného SS, tedy jít pořád doleva. Symetricky *Last*(p) pro poslední vrchol.
- *Next*(v) – nalezení následníka vrcholu (to je hrana, která vede z v dále po cestě). Najdeme příslušný externí uzel x ve SS a půjdeme do jeho následníka

v in-orderu. Podobně $Next(e)$ pro následníka hrany, což je vrchol, a $Prev(x)$ pro předchůdce vrcholu či hrany.

- $Split(e)$ – rozdělení cesty na dvě odebráním hrany e. K tomu použijeme už popsané rozdělení SS na menší a větší prvky.
- $Split(v)$ – rozdělení cesty odebráním vrcholu v a hran, které se ho dotýkají. Samotné v odpovídá externímu uzlu SS, takže ho nelze jen tak smazat. Ale můžeme najít $Prev(v)$ a $Next(v)$, což jsou hrany před a za v, a tyto hrany smazat. Tím se cesta rozpadne na tři části: vše před v, vše za v a samotné v. Stačí tedy smazat třetí část (ta má pouze externí kořen).
- $Join(p_1, p_2)$ – spojení dvou cest hranou (za konec cesty p_1 přidáme novou hranu a na ní napojíme začátek cesty p_2). K tomu stačí založit nový uzel SS odpovídající nové hraně cesty a jako syny tohoto uzlu připojit kořeny obou SS.
- *Reverse* – otočení orientace cesty (poslední vrchol se stane prvním a naopak). Do každého uzlu SS uložíme značku, zda je v celém podstromu pod tímto uzlem prohozená levá a pravá strana. Kdekoliv v podstromu může být samozřejmě další značka, která strany opět prohodí. Značky budeme vyhodnocovat líně: kdykoliv při operacích se SS dojdeme do vrcholu se značkou, prohodíme v něm ukazatele na syny a znegujeme značky v synech. Na samotnou operaci *Reverse* pak stačí znegovat značku v kořeni.

KSP

seriál

Úkol 3 [2b]: Navrhněte operaci pro spojení dvou cest za krajní vrcholy. Poslední vrchol první cesty tedy splyne s prvním vrcholem druhé cesty.

Úkol 4 [3b]: Navrhněte, jak reprezentaci cest upravit, aby si u hran pamatovala i celočíselné ohodnocení. Chceme umět operace „nastav hraně e ohodnocení x“ a „zjisti minimum z ohodnocení hran mezi vrcholy u a v“.

Dynamická dekompozice na cesty

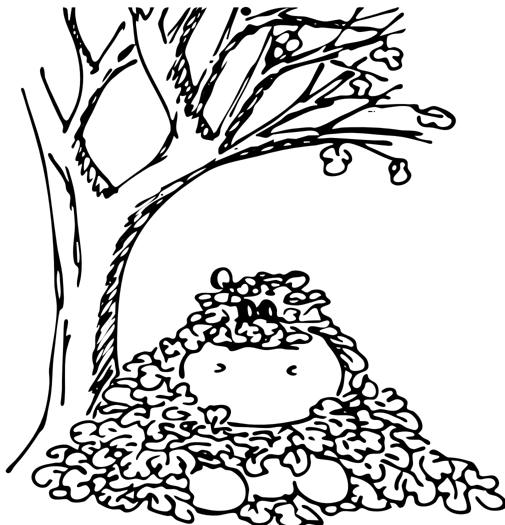
Nyní vymyslíme, jak z cest skládat obecné stromy. Inspirujeme se dekompozicí na lehké a těžké hrany z minulého dílu, ale tentokrát nebudou druhy hran určeny velikostmi podstromů, nýbrž historií struktury, tedy posloupností operací, které jsme zatím provedli.

Strom zakořeníme a všechny hrany zorientujeme směrem do kořene. Hrany rozdělíme na *tlusté* a *tenké*. Která hrana bude tlustá, si můžeme vybrat libovolně, ale musíme dodržet, že do každého vrcholu vede nejvýše jedna tlustá hrana. Tlusté hrany tedy hrají podobnou roli jako těžké hrany v HLD, tenké jako lehké.

Tlusté hrany proto tvoří cesty (orientované směrem ke kořeni) a každý vrchol leží na právě jedné tlusté cestě (možná na triviální jednovrcholové). Z horního vrcholu tlusté cesty může vést tenká hrana, kterou je cesta napojena k nadřazené tlusté cestě.

Každou tlustou cestu budeme reprezentovat již popsaným způsobem pomocí splay stromu. Poslední vrchol cesty w (v původním stromu leží nejvýše, ve splay stromu je to nejpravější externí uzel) si bude pamatovat informace o tenké hraně

do nadřazené cesty: vrchol $tparent(w)$, do nějž tenká hrana vede. Vede-li tlustá cesta až do kořene, položíme $tparent(w) = \emptyset$.

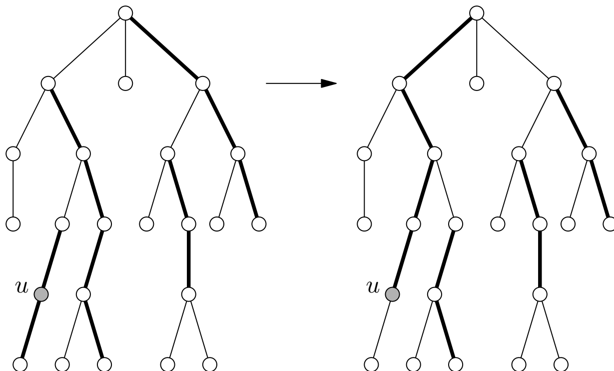


Nyní definujeme operaci $Expose(v)$. Jejím úkolem je přestavět reprezentaci stromu tak, aby z v do kořene vedla tlustá cesta, a navíc byl vrchol v jejím začátkem. Budeme postupovat takto:

1. $p \leftarrow Path(v)$
2. Pokud $First(p) \neq v$:
3. $\quad e \leftarrow Prev(v)$
4. $\quad$ Rozdělíme p operací $Split(e)$ na cesty p_1 (dolní) a p_2 (horní).
5. $\quad tparent(Last(p_1)) \leftarrow v$
6. $\quad p \leftarrow p_2$
7. Dokud $tparent(w) \neq \emptyset$, kde $w = Last(p)$:
8. $\quad x \leftarrow tparent(w)$
9. $\quad q \leftarrow Path(x)$
10. $\quad$ Pokud $First(q) \neq x$:
11. $\quad\quad f \leftarrow Prev(x)$
12. $\quad\quad$ Rozdělíme q operací $Split(f)$ na cesty q_1 (dolní) a q_2 (horní).
13. $\quad\quad tparent(Last(q_1)) \leftarrow x$
14. $\quad\quad q \leftarrow q_2$
15. $\quad p \leftarrow Join(p, q)$

Kroky 2 až 6 ošetřují případ, kdy v není nejnižším na své tlusté cestě p. Tehdy tuto cestu rozdělíme na dvě, které propojíme tenkou hranou. V krocích 7 až 15 cestu p postupně rozšiřujeme až do kořene: dokud ještě nevede do kořene, je připojena tenkou hranou pod nějaký vrchol x ležící na jiné tlusté cestě q. V krocích 10 až 14 zařizujeme, aby x byl nejnižším na q (jinak cestu q rozdělíme). Jakmile x je nejnižší, můžeme cesty p a q propojit do jediné tlusté cesty a pokračovat výš.

Na následujícím obrázku vidíme výsledek *Expose*(u):



Také se nám bude hodit operace *Evert*(v), která strom překoření do vrcholu v. To se provede snadno: nejprve zavoláme *Expose*(v), čímž zařídíme, aby mezi v a starým kořenem vedla jedna tlustá cesta, a tu pak operací *Reverse* obrátíme.

Nyní ukážeme, jak udržovat les zakořeněných stromů a provádět operace s jejich strukturou. Každý strom bude reprezentovaný výše uvedeným způsobem pomocí tlustých cest spojených tenkými hranami.

- *Root*(v) – vrátí kořen stromu, ve kterém se nachází vrchol v. Jednoduše provede *Expose*(v) a pak se pomocí *Last* zeptá na poslední vrchol vzniklé tlusté cesty.
- *Parent*(v) – vrátí otce vrcholu v (nebo $\emptyset$, pokud v je kořen). Pokud následník *Next*(v) na příslušné tlusté cestě není $\emptyset$, vrátíme tohoto následníka. Jinak z v vede tenká hrana, takže vrátíme *tparent*(v).
- *Cut*(v) – není-li v kořen, přeruší hranu mezi v a jeho otcem, čímž strom rozdělí na dva. Může například provést *Expose*(v) a pak *Split* vzniklé tlusté cesty ve vrcholu v.
- *Link*(u, v) – je-li u kořen jednoho stromu a v libovolný vrchol jiného stromu, spojí oba stromy přidáním hrany z u do v. Na to stačí nastavit *tparent*$(u) \leftarrow v$. Pokud chceme přidat hranu mezi dvěma vrcholy, které nejsou kořeny, stačí použít *Evert* a jeden ze stromů překořenit.

Sleator s Tarjanem dokázali, že operace *Expose* má amortizovanou složitost $\mathcal{O}(\log n)$. Důkaz tohoto tvrzení je bohužel mimo možnosti našeho úvodního textu. Je ale jasné, že z toho plyne, že i ostatní operace s dynamickými stromy mají amortizovaně logaritmickou časovou složitost.

Úkol 5 [4b]: Upravte dynamickou dekompozici, aby si u každé hrany pamatovala i její celočíselné ohodnocení. Chceme umět operace „nastav hraně e ohodnocení x“ a „zjisti minimum z ohodnocení hran na cestě mezi vrcholy u a v“.

Úkol 6 [2b]: Navrhněte datovou strukturu pro inkrementální udržování minimální kostry. Na počátku máme graf bez hran a postupně přidáváme ohodnocené hrany. Po každém přidání chceme zjistit, jak se změnila minimální kostra.

Martin „Medvěd“ Mareš

Recepty z programátorské kuchařky

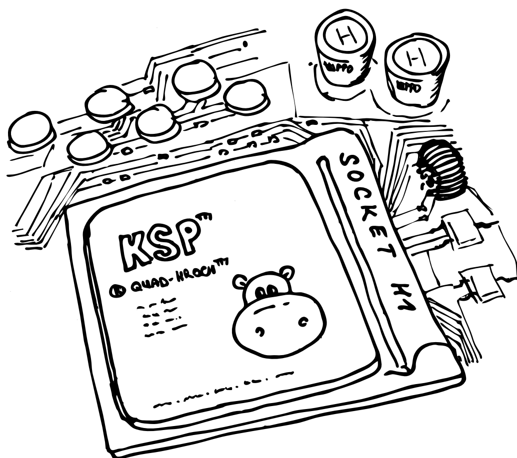
Kuchařka první série – základní algoritmy

Tato naše kuchařka je nejzákladnější ze základních a je určena hlavně pro začínající řešitele. To však neznamená, že zkušenější řešitelé do ní nahlédnout nemůžou – třeba na nějakou konkrétní programátorskou techniku, kterou by si potřebovali osvěžit.

V první části kuchařky se seznámíme hlavně se základními principy programování, uchovávání dat v počítači a základy rychlé manipulace s nimi. Po přečtení této části bychom měli být schopní převést své myšlenky z hlavy na papír či do počítače a měli bychom vědět, proč je námi zvolený postup rozumný.

Druhá část nás poté seznámí se základními postupy, jak řešit určité konkrétní problémy. Naučíme se například, jak rychle vyhledávat v uspořádané posloupnosti hodnot, nebo jak si pomocí předpočítání usnadnit řešení těžké úlohy.

Většinu klíčových částí se pokusíme též ukazovat v podobě zdrojového kódu ve dvou různých jazycích (v nízkoúrovňovém C, kde je zápis blízký tomu, jak počítač doopravdy pracuje, a v Pythonu, ve kterém se píše o něco příjemněji). Nebudeme ale probírat základy syntaxe těchto jazyků, ty si případně můžete nastudovat jinde.[28] Pokud žádný z těchto jazyků neumíte, nezoufejte. KSP můžete řešit i bez toho, stačí když svá řešení důkladně slovně popíšete (konkrétní jazyk se pak můžete naučit až během dalších sérií).



[28] http://ksp.mff.cuni.cz/study/odkazy.html

Část první: Základní pojmy

Algoritmus a program

Pod tajemným slovem *algoritmus* se skrývá jen jiný výraz pro postup. Můžete si to představit jako příkaz od maminky „Běž do krámu, kup chleba, a když budou mít měkké rohlíky, tak jich vem tucet“.[29]

Takovýto příkaz klidně můžeme nazvat algoritmem, ačkoliv to bude asi znít nezvykle – pojem algoritmus se totiž používá hlavně ve světě počítačů. Je to tedy nějaká posloupnost základních příkazů, která řeší nějaký problém. Výběr konkrétního programovacího jazyka rozhoduje o tom, jaké základní příkazy budeme mít k dispozici. Většinou jsou ale skoro stejné.

Mezi základní příkazy patří:

- Manipulace s daty v paměti (uložení či načtení hodnoty, detailněji v další kapitole).
- Provedení nějakého numerického výpočtu $(+, -, *, /)$.
- Vyhodnocení určité podmínky a odpovídající větvení programu: *Pokud platí A, tak proveď B, jinak proveď C.* Přitom B i C mohou být klidně celé *bloky kódu*, tedy libovolně mnoho dalších základních příkazů.
- Opakování nějakého příkazu: *Dokud platí A, dělej B.* Takové konstrukci říkáme *cyklus* a podobně jako u podmínky může být B blok kódu, který se celý opakuje.
- Vstup a výstup programu (typicky vstup od uživatele z klávesnice či načtení vstupu ze souboru; výstup pak může znamenat vypsání výsledku na obrazovku nebo třeba zapsání dat do souboru).

Z těchto základních stavebních kamenů se skládá každý algoritmus. Programem potom rozumíme realizaci algoritmu v nějakém konkrétním programovacím jazyce.

U složitějších programů se pak často setkáte s problémem, že budete mít nějakou posloupnost příkazů, která se bude na spoustě míst programu opakovat, což zbytečně prodlužuje a znepřehledňuje kód.

Řešením tohoto problému je použití *funkcí*. Funkci si můžeme představit jako nějakou pojmenovanou část programu (s vlastní pamětí), kterou můžeme opakovaně použít tím, že ji v různých částech programu *zavoláme*. Funkci při zavolání předáme parametry (například seznam čísel), které se dostanou do její vnitřní paměti.

Funkce pak na základě obdržených parametrů může provádět nějaké operace, při kterých pracuje se svojí vnitřní pamětí (mluvíme o *lokální* paměti, změny

[29] A jako slušně vychovaní se tedy vydáte do krámu a koupíte tucet chlebů, protože měli měkké rohlíky :-)

v ní se neprojeví nikde mimo funkci). Na konci nám funkce může vrátit nějaký výsledek. Pokud funkce během svého běhu změní i nějaká data v *globální* paměti, či provede nějakou globální operaci (například výpis textu na monitor), mluvíme pak o funkci s *vedlejšími efekty* (neboli side-efekty).

Konkrétním příkladem může být funkce, která nám spočítá odmocninu ze zadaného čísla. Ta dostane jako svůj parametr číslo, uvnitř si provede nějaký výpočet, o který se jako uživatel funkce nemusíme starat, a jako výstup nám vrátí spočtenou odmocninu.

Reprezentace dat v počítači

Celkem často si v průběhu výpočtu našeho algoritmu potřebujeme pamatovat nějaké hodnoty. K tomu nám programovací jazyky dávají nástroj s názvem *proměnná*. Ta představuje jakési pojmenované místo v paměti (přihrádku), do kterého si můžeme data ukládat a pak je odtud zase načítat.

Typickým příkladem může být počítání součtu čísel, která nám uživatel zadá na vstupu. Na začátku nejdříve do nějakého místa v paměti uložíme hodnotu 0. Poté postupně, jak nám uživatel zadává čísla, tuto proměnnou pokaždé přečteme, k její hodnotě přičteme nově zadané číslo a výsledek opět uložíme na stejné místo.

Takovéto použití jedné proměnné je velmi jednoduché (tak jednoduché, že ho takto podrobně do řešení KSPčka nepište, není to potřeba), ale také celkem omezené. Co kdybychom si chtěli pamatovat třeba celou zadanou posloupnost čísel? Mohlo by nám stačit vyrobit si spoustu různě pojmenovaných proměnných, ale nejde to lépe? Jde.

Jednotlivé proměnné se mohou kombinovat do složitějších konstrukcí, které obecně nazýváme *datovými strukturami*. Zkusíme si ty nejzákladnější představit.

Pole

První datovou strukturou, kterou si představíme a která se na výše nastíněnou situaci náramně hodí, je *pole*. To představuje spoustu přihrádek (proměnných) naskládaných v paměti za sebou, ke kterým typicky přistupujeme přes jeden společný název pole a jejich pořadové číslo neboli index (jako `NazevPole[0], NazevPole[1], ...`).[30]

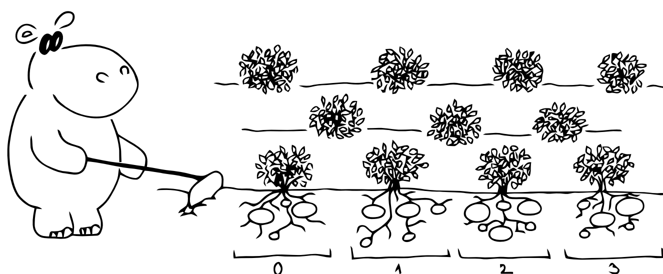
Ve většině základních jazyků je pole jen *statické*, tedy v okamžiku jeho vytváření musíme počítači říct, jak ho chceme velké. Některé vyšší jazyky ale nabízejí i pole, které se dynamicky zvětšuje, takovou konstrukci si ukážeme ve druhé části kuchařky.

Abychom nebyli omezeni jen jedním rozměrem, můžeme si klidně vyrobit pole dvourozměrné (případně obecně n-rozměrné). Dvourozměrné pole je vlastně tabulka hodnot, nazýváme ji také někdy *matice*, a může se nám hodit napří-

[30] Pozor, ve světě počítačů se velmi často indexuje od nuly, tedy první prvek má v tomto případě index 0.

klad při reprezentaci různých map (plán bludiště) nebo, jak uvidíme níže, pro reprezentaci dalších datových struktur.

U pole již má smysl přemýšlet, jak dlouho bude která operace trvat. Díky tomu, že jsou jednotlivé prvky v poli naskládané pevně za sebou, když se počítače zeptáme na obsah přihrádky `pole[42]`, přesně ví, na kterém místě v paměti se její obsah nachází, a proto nám hodnotu vrátí ihned.



Tomu budeme říkat *operace v konstantním čase* a budeme značit, že trvá čas $\mathcal{O}(1)$. Efektivitu programu totiž nepočítáme v sekundách (protože každý z nás má asi jinak rychlý počítač), ale v počtu základních operací, které musí program řádově vykonat. Více o časové složitosti si můžete přečíst v kuchařce o složitosti,[31] nejdříve však doporučujeme dočíst tuto kuchařku.

Přidání nového prvku na konec pole také zvládneme v konstantním čase. Problém je přidání nového prvku někam doprostřed (což se nám typicky stane, pokud budeme chtít udržovat hodnoty v poli seřazené a zároveň do něj vkládat nové). V takovém případě se totiž všechny prvky za vkládaným musí posunout o jednu pozici dál, aby se vkládaný prvek vešel na své místo. Taková operace tedy může pro pole délky N prvků trvat řádově až N kroků, což zapisujeme jako $\mathcal{O}(N)$ a říkáme, že je to vzhledem k N *lineární časová složitost*.

To je docela značná nevýhoda oproti struktuře, kterou si ukážeme za chvíli. Určitě ale pole nezavrhujme. Je to základní datová struktura, která nalezne použití ve spoustě programů, a jak si ve druhé části kuchařky ukážeme, můžeme ho použít třeba k rychlému hledání hodnoty metodou *binárního vyhledávání*. Nyní ale již slibovaná další datová struktura.

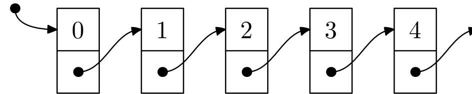
Spojový seznam a ukazatele

Pole jsme měli v paměti určené jenom tím, že počítač věděl, kde je jeho začátek a kolik místa v paměti zabírají jeho prvky. Při dotazování na konkrétní index pak podle indexu a podle velikosti prvků počítač přesně věděl, kam do paměti se má podívat, aby našel námi požadovaný prvek (to vše zvládl v konstantním

[31] `http://ksp.mff.cuni.cz/viz/kucharky/slozitost`

čase). Jednotlivé prvky si tedy vůbec nemusely pamatovat, kde se nachází jejich sousedi, protože všechny prvky seděly v paměti za sebou.

Představme si ale teď situaci, kdy by si každý prvek ještě pamatoval pozice sousedů. Pak bychom mohli mít prvky libovolně rozházené v paměti a jen by se na sebe vzájemně odkazovaly (první prvek by tvrdil, že druhý je na pozici X, druhý by tvrdil, že třetí je na pozici Y, a tak dále).

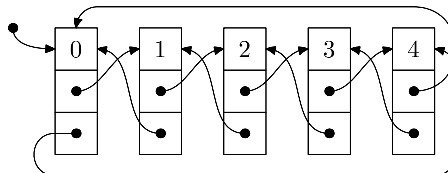


K lepšímu pochopení tohoto principu je důležité si vysvětlit, co to je *ukazatel* (nebo také *odkaz* či anglicky *pointer*). Každé místo v paměti počítače má své číselné označení, kterému říkáme *adresa*. Když si vytváříme nějakou pojmenovanou proměnnou, ta se vlastně odkazuje na určité místo v paměti a na tomto místě v paměti je její hodnota.

Co kdyby ale hodnota proměnné byla adresa nějakého jiného místa v paměti? Pak takové proměnné říkáme *pointer* a umožňuje nám vytvářet výše popsanou strukturu rozházených prvků v paměti.

Spojový seznam je tedy určený svým prvním prvkem (máme v jedné proměnné pointer na tento prvek, který se často nazývá *kořen*, protože z něj „vyrůstá“ zbytek struktury) a poté u každého dalšího prvku máme za sebou uloženou hodnotu tohoto prvku a odkaz (pointer) na další prvek. Odkazy mezi prvky mohou být i obousměrné, mohou vést dokola (poslední ukazuje na první) či mohou dokonce tvořit nějakou složitější strukturu (pak to ale již nebude čistý spojový seznam).

Pokud pointer nemá nikam ukazovat, realizuje se to odkázáním tohoto pointeru na adresu `NULL`. To skoro doslovně říká „Neukazuji nikam“.



Co nám takto vystavěná struktura umožňuje v porovnání s polem? Přístup na konkrétní prvek v ní stojí lineárně času, protože ho musíme „odkrokovat“ od prvního prvku (na který máme pointer), tedy musíme udělat až $\mathcal{O}(N)$ kroků. Pokud bychom však pointer na daný prvek už nějak měli, samozřejmě na něj můžeme přistoupit v konstantním čase.

Naopak přidávání prvků na konkrétní místo (i jejich odebírání) máme v podstatě zadarmo a spojový seznam můžeme rozšiřovat, dokud na něj máme v počítači paměť. Ve chvíli, kdy chceme přidat nový prvek za prvek, na který máme

pointer, jen šikovně přepojíme ukazatele. Pokud předtím ukazatele vedly $A \to B$, teď povedou $A \to C \to B$ (a při odebírání naopak).

Zde můžete vidět ukázku pointerů a spojových seznamů v jazyce C, kde jsou tyto věci mnohem více nízkoúrovňové (ale zato rychlejší):

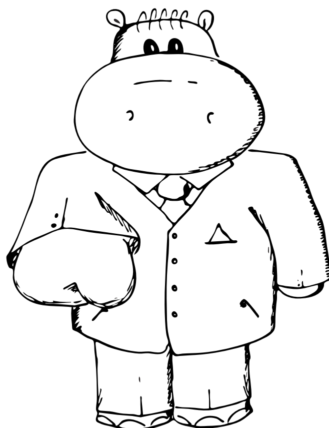

```
#include <stdio.h>
#include <stdlib.h>
// Příkazy výše načetly do programu
// standardní knihovny a funkce z˜nich.

// Struktura pro prvek obsahující dopředné
// i zpětné odkazy. Zkráceně tomuto typu
// budeme říkat "tprvek".
typedef struct prvek tprvek;
struct prvek {
    int hodnota;
    tprvek *dalsi;
    tprvek *predchozi;
};

// Vytvoří nový prvek:
tprvek *novy(int i) {
    tprvek *aktualni =
        malloc(sizeof(tprvek));
    aktualni->dalsi = NULL;
    aktualni->predchozi = NULL;
    aktualni->hodnota = i;
    return aktualni;
}

// Odstraní prvek a vrátí pointer na další
// prvek (vrácení pointeru se hodí při
// odstraňování kořene):
tprvek *odstran(tprvek *aktualni) {
    if (aktualni->predchozi != NULL)
        aktualni->predchozi->dalsi =
            aktualni->dalsi;
    if (aktualni->dalsi != NULL)
        aktualni->dalsi->predchozi =
            aktualni->predchozi;

    tprvek *pomocna = aktualni->dalsi;
    free(aktualni);
    return pomocna;
}
```

```
// Vloží a vrátí pointer na nový prvek:
tprvek *vloz_za(tprvek *aktualni, int i) {
    tprvek *pomocna = aktualni->dalsi;
    aktualni->dalsi = novy(i);

    if (pomocna != NULL)
       pomocna->predchozi = aktualni->dalsi;

    aktualni->dalsi->dalsi = pomocna;
    return aktualni->dalsi;
}

// Použití:
int main(void) {
    tprvek *koren = novy(1);
    tprvek *aktualni = vloz_za(koren, 2);

    aktualni = koren;
    while (aktualni != NULL) {
        printf("%d\n",aktualni->hodnota);
        aktualni = aktualni->dalsi;
    }

    return 0;
}
```

Zde je ukázka spojových seznamů v Pythonu, kdybychom si je podobně jako v C chtěli naprogramovat sami (Python totiž obsahuje spoustu základních struktur již hotových, podívejte se na modul jménem `collections`):

```
class Prvek:
    def __init__(self, hodnota):
        self.hodnota = hodnota
        self.dalsi = None
        self.predchozi = None

class Spojak:
    def __init__(self):
        self.koren = None

    def Vypis(self, aktualni):
        if aktualni is not None:
            print aktualni.hodnota
            self.Vypis(aktualni.dalsi)

    def VlozPo(self, prvek, zaPrvek = None):
        if zaPrvek is not None:
            prvek.dalsi = zaPrvek.dalsi
```

```
                prvek.predchozi = zaPrvek
                zaPrvek.dalsi = prvek
            if prvek.dalsi is not None:
                prvek.dalsi.predchozi = prvek
            if self.koren is None:
                self.koren = prvek

        def Odstran(self, prvek):
            if prvek.predchozi is not None:
                prvek.predchozi.dalsi = \
                    prvek.dalsi
            if prvek.dalsi is not None:
                prvek.dalsi.predchozi = \
                    prvek.predchozi

# Použití:
prvekA = Prvek("A")
prvekB = Prvek("B")
prvekC = Prvek("C")
prvekD = Prvek("D")

seznam = Spojak()

seznam.VlozPo(prvekB)
seznam.VlozPo(prvekD, prvekB)
seznam.VlozPo(prvekC, prvekD)
seznam.VlozPo(prvekA, prvekC)
seznam.Odstran(prvekC)

seznam.Vypis(seznam.koren)
```



Fronta a zásobník

S použitím spojových seznamů (nebo v jednodušším případě dokonce i polí) můžeme zkonstruovat dvě velmi užitečné datové struktury, frontu a zásobník.

Fronta funguje tak, jak si ji asi každý z nás představuje: ten, kdo se do fronty postaví první, ten také první přijde na řadu. Můžeme si ji také představit jako trubku, do které na jedné straně sypeme nějaké věci a na druhé je odebíráme. Anglicky je též nazývaná *FIFO* („*First In, First Out*“).

Praktickou realizaci uděláme jednoduše spojovým seznamem. Budeme si držet dva ukazatele, jeden na začátek seznamu, druhý na konec. Když se objeví nový prvek, který do fronty budeme chtít vložit, přidáme ho na konec, zatímco při odebírání z fronty využijeme druhého ukazatele a vezmeme prvek ze začátku.

Druhou velmi podobnou datovou strukturou je *zásobník*. Jak už ale plyne z anglického názvu *LIFO* („*Last In, First Out*“), funguje spíše jako plný šuplík: Nahoru na něj přidáváme nové prvky, a když chceme nějaký odebrat, vezmeme také zvrchu. To znamená, že první se na řadu dostane naposledy vložený prvek.

Implementace je velmi obdobná jako u fronty, jen bude ukazatel pouze jeden a bude ukazovat jenom na jeden konec spojového seznamu.

Knihovny

Tyto základní struktury už jsou často předpřipravené jako součást nějakých *knihoven* v daném jazyce. Knihovna je většinou sbírka nějakých navzájem souvisejících funkcí, které již někdo sepsal a které si můžeme do našeho programu načíst a používat. Ukázku načtení knihoven můžete vidět například ve výše zmíněném kódu v jazyce C.

Je ale velmi důležité rozumět tomu, jak knihovní funkce vnitřně fungují. Protože jedině když budeme vědět, co je jak rychlé a efektivní, budeme schopni psát rychlé programy.

Teď již víme, jak reprezentovat nejzákladnější datové struktury v počítači, ale mohlo by se nám hodit zastavit se ještě chvíli u dalších struktur. Tentokrát je už budeme studovat trochu teoretičtěji.

Stromy a grafy v informatice

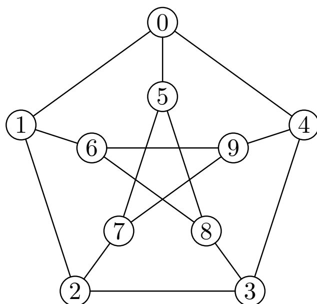
Grafy

S nějakými grafy jste se již možná potkali, ale tento pojem je bohužel docela přetěžovaný. Jedním jeho významem jsou „koláčové grafy“ a jiné další diagramy znázorňující nějaký poměr (ať už to jsou výsledky voleb, nebo poměr lidí, kteří sledovali v televizi Večerníček).

Další význam můžeme nalézt v analytické matematice, kde se potkáme s grafy průběhu nějakých funkcí. My však nemáme na mysli ani jedno z výše zmíněných, teď se budeme bavit o *kombinatorických grafech*.

Grafem tedy máme na mysli nějakou množinu objektů, říkejme jim *vrcholy*, a nějaké vztahy mezi nimi. Tyto vztahy nazýváme *hranami* a jsou vyjádřené

dvojicemi vrcholů, mezi kterými vedou. Ukázku takového grafu vidíme třeba na následujícím obrázku.



Jako praktickou ukázku grafu si můžeme například představit silniční síť nějakého státu: vrcholy budou města a hrany budou silnice, které mezi nimi vedou.

Občas se můžete setkat s pojmem *souvislý* graf. Ten znamená jen to, že mezi každými dvěma vrcholy existuje nějaká cesta. Pokud tomu tak není, je graf *nesouvislý* a dá se rozložit na několik menších grafů, které již souvislé jsou a říká se jim *komponenty souvislosti*.

Samotný graf poté můžeme doplnit tím, že si v každém vrcholu nebo na každé hraně budeme pamatovat nějakou hodnotu (například cenu nejlevnějšího benzínu ve městech a délku v kilometrech na silnicích). Pamatování si hodnot ve vrcholech je docela obvyklá technika a nemá speciální název, ale pokud budeme mít graf, který si pamatuje hodnoty na hranách, budeme o něm mluvit jako o *ohodnoceném grafu*.

Další možnou úpravou je, že každá hrana povede jen jedním směrem (jednosměrné silnice), takovým grafům říkáme *orientované* (pokud pak v orientovaném grafu chceme silnici oběma směry, prostě do něj přidáme dvě hrany, jednu v každém směru).

Poslední, co nám schází k praktickému použití grafů, je naučit se, jak je reprezentovat v počítači. Existuje několik možností (n bude značit počet vrcholů, m počet hran):

- **Seznam sousedů** – vrcholy grafu budeme mít uložené v poli a u každého vrcholu budeme mít (spojový) seznam čísel dalších vrcholů, do kterých z aktuálního vrcholu vede hrana. Zabírá místo $\mathcal{O}(n + m)$ a hodí se pro řídké grafy (tedy grafy, kde je m řádově stejné jako n).
- **Matice sousednosti** – tabulka $n \times n$, kde na souřadnicích $[i, j]$ je jednička (případně jiná hodnota, v případě ohodnoceného grafu), pokud z i do j vede hrana, a nula, pokud tam hrana není (u neorientovaných grafů je navíc matice syme-

trická – je jedno, jestli vezmeme $[i, j]$ nebo $[j, i]$). Hodí se pro husté grafy, kde $m \sim n^2$.

- **Matice incidence** – řádky reprezentují vrcholy, sloupce hrany. V každém sloupci jsou právě dvě jedničky – indexy vrcholů, mezi kterými hrana vede. Zabírá však $\mathcal{O}(mn)$ a její použití bývá dost neohrabané, takže je většinou lepší dát přednost jiné reprezentaci grafu. Je však dobré o ní vědět.

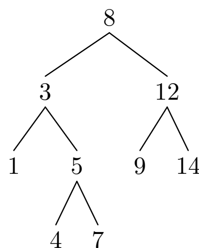
Grafy jsou velmi široké téma. Můžeme hledat jejich minimální kostry, můžeme v nich hledat nejkratší cesty či skrze ně pouštět pod tlakem vodu. Více o nich si tedy můžete přečíst v některé z našich specializovaných grafových kuchařek, které odkazujeme z našeho kuchařkového rozcestníku.[32]

Stromy

Možná si říkáte, co má informatika u všech elektronů společného s lesnictvím? Kupodivu celkem mnoho a bez stromů bychom se v leckterém případě jen těžko obešli. Informatické stromy sice nejsou většinou tak zelené, mají ale, na rozdíl od svých dřevnatých sourozenců, mnoho jiných pěkných vlastností.

Strom je vlastně speciálním případem souvislého grafu, který neobsahuje žádnou kružnici (cyklus). To znamená, že mezi každými dvěma vrcholy stromu existuje právě jedna cesta.

Díky této vlastnosti můžeme nějaký zvolený vrchol prohlásit za *kořen* a náš strom za něj pomyslně zavěsit (tak, že strom roste od kořene směrem dolů), této operaci se říká *zakořenění*. Pak můžeme mluvit o tom, že z kořene směrem dolů (informatické stromy mají tradičně kořen nahoře) vyrůstají nějaké *podstromy*.



Pokud je strom zakořeněný, můžeme v něm mluvit o *hloubce* každého vrcholu, neboli o jeho vzdálenosti od kořene. Hloubka celého stromu je pak nejdelší ze vzdáleností od kořene k nějakému z *listů* (tak říkáme vrcholům, které již nemají žádné *syny*, tedy vrcholy, které by z nich vyrůstaly). Podle hloubky poté můžeme vrcholy stromu uspořádat do jednotlivých *hladin*.

Velmi často používáme stromy, které jsou nějak pravidelné. Příkladem jsou třeba *binární stromy*, které mají v každém vrcholu maximálně dva syny (říkáme

[32] `http://ksp.mff.cuni.cz/study/cooks/`

jim *levý a pravý podstrom*). Reprezentovat se dají buď obecně jako každý jiný strom (v každém vrcholu spojový seznam podstromů), nebo velmi pěkně i v poli.

Stačí si pomyslně doplnit binární strom na *úplný* (to je takový, který má všechny své hladiny plné) a pak ho od kořene směrem dolů po hladinách očíslovat (kořen dostane číslo nula, jeho synové čísla jedna a dva, další hladina čísla tři až šest, atd.).

Můžeme si všimnout, že pokud si v takovém očíslování vezmeme jakýkoliv vrchol s číslem (indexem) i, tak jeho synové jsou právě vrcholy s indexy $2i + 1$ a $2i + 2$. Do pole níže je zapsaný binární strom z obrázku výše.

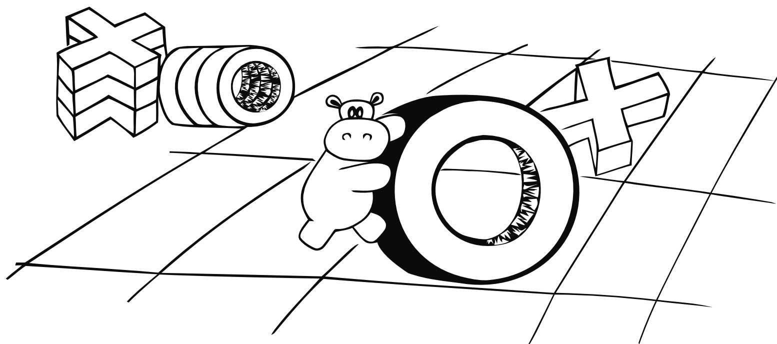
index	0	1	2	3	4	5	6	7	8	9	10
hodnota	8	3	12	1	5	9	14	–	–	4	7

Jak plyne z očíslování, pro úplný binární strom je uložení v poli efektivní a neplýtváme místem. Pokud ale strom úplný nebude, zůstanou nám v poli volná místa. Uložení v poli se tedy vyplatí jen pro stromy, které se od úplných příliš neliší.

Speciálním případem binárních stromů jsou pak ještě *binární vyhledávací stromy*. Jsou to normální binární stromy, pro něž navíc platí, že ať si vezmeme libovolný vrchol, budou hodnoty vrcholů v jeho levém podstromě menší než hodnota tohoto vrcholu, a hodnoty v jeho pravém podstromě naopak větší.

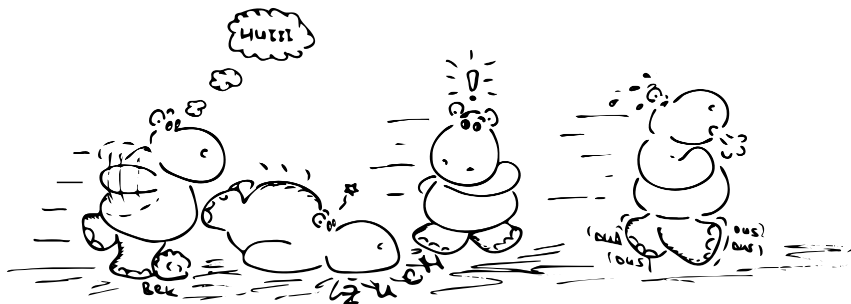
V takovém stromě pak zvládneme snadno vyhledávat. Budeme ho postupně procházet od kořene a v jednotlivých vrcholech budeme porovnávat hledanou a aktuální hodnotu a podle toho sestupovat do správného podstromu. Podobná technika je detailněji popsaná ve druhé části kuchařky, v kapitolce *Rozděl a panuj*.

Na složitější datové struktury stavějící na těchto základních (haldy, intervalové stromy, . . .) se můžete podívat do některé z našich dalších kuchařek, na jejichž přehled jsme vás už odkázali o kapitolu výše.



Část druhá: Programátorské techniky

Tato část by měla sloužit jako rychlý přehled a ukázka různých technik, které se dají použít při řešení úloh z KSPčka, nebo při programování obecně.



Rekurze

Rekurze je velmi důležitá programátorská technika. V podstatě znamená definování nějaké věci (ať už je to nějaký objekt či postup výpočtu) pomocí sebe sama.

Rekurzivně může být například zadána nějaká datová struktura. Například stromy jsou pěkným příkladem rekurzivně definované datové struktury – každý vrchol stromu může mít syny, a každý z těchto synů je sám o sobě strom (tedy i osamocený vrchol bez synů je stromem).

Prakticky je to realizováno tak, že každý vrchol má svou hodnotu a pak ještě seznam ukazatelů vedoucích na další případné podstromy. S ukazateli jsme se již potkali a s jejich pomocí jsme si postavili spojový seznam. A přesně tak, spojový seznam je také ve své podstatě rekurzivní datová struktura.

Mimo rekurzivních datových struktur se ale často potkáváme i s rekurzivním postupem výpočtu nějakého programu, nejčastěji realizovaným ve formě funkce, která volá sama sebe (většinou s jinými parametry, jinak by to asi nemělo smysl), takové funkci se říká *rekurzivní funkce*.

U rekurzivních funkcí je nejdůležitější věc definovat nějakou *koncovou podmínku*, tedy podmínku, při níž už se rekurze zastaví. Jinak by se totiž mohlo stát, že by rekurze běžela donekonečna.

Přesněji rekurze by se i tak v nějakou chvíli zastavila, ale skončila by chybou, protože by jí došla paměť – každá volání funkce si totiž ukousne kus paměti (musí si pamatovat, kam se po skončení má vrátit) a pokud má rekurzivní funkce ještě nějaké lokální proměnné, musíme si někde uložit všechny lokální proměnné funkcí, z kterých jsme se doposud nevrátili.

Rekurzivní funkce a převod na nerekurzivní cyklus

Typickým příkladem rekurzivní funkce je výpočet Fibonacciho čísel. Ta jsou definována tak, že $f_0 = 1$, $f_1 = 1$ a n-té Fibonacciho číslo je součtem dvou předchozích ($f_n = f_{n-1} + f_{n-2}$). To nám dává posloupnost čísel 0, 1, 1, 2, 3, 5, 8, 13, … pokračující donekonečna. Pokud toto přepíšeme do programového kódu, tak dostáváme následující zápisy:

V jazyce C:

```
int fib(int n) {
        if (n==0) return 0;
        else if (n==1) return 1;
        else return fib(n-1) + fib(n-2);
}
```

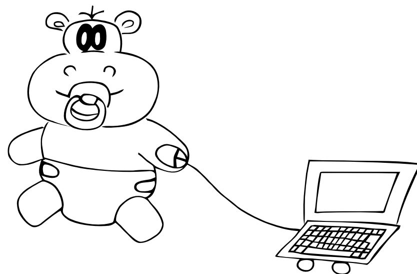
V Pythonu:

```
def fib(n):
        if n==0:
                return 0
        elif n==1:
                return 1
        else:
                return fib(n-1) + fib(n-2)
```

Jak vidíme, je přepis celkem přímočarý. Pokud by se nám však rekurze v nějakém případě nelíbila, můžeme se každé rekurze zbavit. Rekurzivní volání totiž můžeme šikovně přepsat na nějaký cyklus se *zásobníkem*.

Pak jen v cyklu odebíráme prvky ze zásobníku, dokud není prázdný, a za každé rekurzivní zavolání do zásobníku přidáme parametry, se kterými bychom naši funkci volali. Tímto postupem převedeme každou rekurzivní funkci na nerekurzivní.

Ještě doplníme poznámku, že ve většině programovacích jazyků každé volání funkce stojí nějaký čas, sice malý, ale když se volání provádí opakovaně, tak se to už nasčítá. Pro reálnou implementaci je tedy nejlepší pokusit se rekurzi převést na nerekurzivní volání, pokud to nějak rozumně jde.



Občas to jde dokonce i jednodušeji a bez zásobníku. Podívejte se na alternativní variantu výpočtu Fibonacciho čísel níže a rozmyslete si, co dělá.

V jazyce C:

```
int fib2(int n) {
        if (n==0) return 0;
        else if (n==1) return 1;

        int a = 0; int b = 1;
        for(int i = 2; i<=n; i++) {
                int c = a + b;
                a = b;
                b = c;
        }
        return b;
}
```

V Pythonu:

```
def fib2(n):
        if n==0:
                return 0
        elif n==1:
                return 1

        a = 0; b = 1
        while n>1:
                (a, b) = (b, a+b)
                n-=1
        return b
```

Jak vidíte, je i tato funkce elegantní a navíc běhá mnohem rychleji, než její rekurzivní varianta. Tato funkce běhá v $\mathcal{O}(n)$, kdežto rekurzivní varianta počítala stejné věci mnohokrát dokola (zkuste si nakreslit nějaký strom volání předchozí funkce, případně se podívat dopředu do kapitoly Předpočítané mezivýsledky).

Rekurzivní varianta tedy běžela až v čase $\mathcal{O}(2^n)$, což je pro velká n mnohem pomaleji než $\mathcal{O}(n)$ (avšak šla by celkem snadno zachránit, aby běžela také v $\mathcal{O}(n)$, zkuste si rozmyslet jak).

Backtracking

S rekurzí silně souvisí i pojem *backtracking*, česky by se snad dalo říci „metoda pokusu a omylu". Tímto pojmem označujeme proces, kdy postupně zkoušíme všechny možnosti, jak vyřešit nějaký problém.

Metoda pokusu a omylu se tento proces nazývá proto, že pokud již nemůžeme pokračovat dál (třeba v případě, že v bludišti dojdeme do slepé uličky), vrátíme se kus zpět a zkusíme jinou (zatím nevyzkoušenou) možnost. Takto po-

stupně zkusíme každou možnost, a buď nalezneme námi hledané řešení, nebo se vrátíme až na výchozí pozici a zjistíme, že řešení neexistuje.

Backtracking bývá často realizován pomocí rekurze, ukážeme si to na příkladu hledání rozkladu zadané částky na mince o hodnotách 5 Kč a 3 Kč (všimněte si, že v takto omezeném peněžním systému nejde složit třeba částka 7 Kč). Naše funkce dostane jako parametr zbývající částku a zkusí rekurzivně provést rozklad na jednotlivé mince:

V jazyce C:

```
bool rozloz(int castka) {
        // Koncová podmínka rekurze
        if (castka == 0) return true;
        else if (castka < 0) return false;
        else if (rozloz(castka-5)) {
                printf(" 5 Kc");
                return true;
        } else if (rozloz(castka-3)) {
                printf(" 3 Kc");
                return true;
        } else return false;
}
```

V Pythonu:

```
def rozloz(castka):
        if castka == 0:
                return True
        elif castka < 0:
                return False
        elif rozloz(castka-5):
                print " 5 Kc"
                return True
        elif rozloz(castka-3):
                print " 3 Kc"
                return True
        else:
                return False
```

V každém kroku zkusíme nejdříve použít pětikorunovou minci a zavoláme se na zbylou částku, a když náš rozklad nevyjde, zkusíme v tomto kroku použít ještě třikorunu. Takto se rozhodujeme v každém kroku rekurze a případně se vracíme z neúspěšných větví výpočtu a zkoušíme další možnosti.

Takovým postupem ale vyzkoušíme až exponenciálně mnoho možností (tedy $\mathcal{O}(2^n)$), což není moc rychlé. Proto je doporučováno se backtrackování raději vyhnout, nebo ho nějak chytře vylepšit. Je však dobré o backtrackování vědět, protože existují problémy, které efektivněji řešit neumíme.

Rozděl a panuj

Jednou ze základních technik je rozdělení složitějšího problému na menší části, které opět můžeme rozdělit na menší a tak dále, dokud se nedostaneme k problémům tak malým, že je už umíme triviálně vyřešit.

Binární vyhledávání v poli

Představte si, že máme seřazené pole n prvků a chceme zjistit, jestli se v něm nachází prvek s hodnotou k. Určitě můžeme projít celé pole v lineárním čase (tím, že budeme brát jeden prvek za druhým a kontrolovat, zda je roven hodnotě k), ale to je zbytečně pomalé a nevyužívá toho, že máme pole seřazené.

Můžeme totiž začít s velkým problémem a ten postupně zmenšovat na stále menší a menší. Nejdříve hledáme k v celém poli. Podíváme se na jeho prostřední prvek:

- Pokud je roven k, jsme hotovi.
- Je-li větší než k, víme, že se k musí nacházet nalevo od něj. Můžeme tedy hledat znovu, ale tentokrát se omezit jen na levou polovinu pole.
- Analogicky, je-li menší než k, můžeme hledat jen v pravé polovině.

Když tímto postupným dělením problémů na menší dojdeme až k poli o velikosti jednoho prvku, stačí tento prvek jenom porovnat, dál už se pole nepokoušíme rozdělovat.

Jelikož se nám každým krokem problém zmenší na polovinu, tak se maximálně po $\log n$ krocích dostaneme na pole velikosti jedna. Říkáme, že algoritmus má *logaritmickou časovou složitost*, píšeme $\mathcal{O}(\log n)$.[33]

Prakticky postup provádíme tak, že si udržujeme levý a pravý okraj aktuálně zpracovávaného úseku a postupně je k sobě přibližujeme.

Ukázka hlavní smyčky v C:

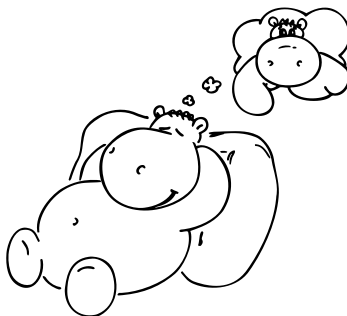
```
int pole[] = {1,2,5,7,12,16,42};
int hledane = 8;
int L = 0, R = 6;
int x;
```

[33] Pokud není řečeno jinak, znamená pro nás v informatice značka log *dvojkový logaritmus*, což je funkce opačná k funkci 2^n a roste o hodně pomaleji než funkce lineární. Pro velká n platí: $1 < \log n < n$ a například $\log 2 = 1, \log 8 = 3, \log 1024 = 10$.

```
do {
    int prostredni = (L+R)/2;
    x = pole[prostredni];
    if (x == hledane)
        printf("Pole obsahuje hledane\n");
    else if (x < hledane)
        L = prostredni + 1;
    else
        R = prostredni;
} while (L < R && x != hledane);

if (x != hledane)
    printf("Hledane neni v poli\n");
```


Ukázka v Pythonu jako funkce vracející index prvku nebo −1, pokud hledané číslo nenalezne:


```
def bin_vyhled(pole, hledane, L=0, R=None):
    if R is None:
        R = len(pole)
    while L < R:
        prostredni = (L+R)//2
        x = pole[prostredni]
        if x < hledane:
            L = prostredni + 1
        elif x > hledane:
            R = prostredni
        else:
            return prostredni
    return -1

# Zavolání:
print bin_vyhled([1,2,5,7,12,16,42], 8)
```



Další aplikace

Další typickou aplikací postupu rozděl a panuj je například třídění posloupnosti pomocí *Mergesortu*. Ten v základu funguje tak, že každou posloupnost, kterou dostane k setřídění, rozdělí na poloviny a každou z nich setřídí rekurzivním zavoláním sebe sama. Zanořování se zastaví ve chvíli, kdy třídíme posloupnost délky jedna (ta už je z podstaty setříděná). Pak jen v každém kroku ze dvou setříděných menších posloupností vyrobí jejich sléváním setříděnou posloupnost dvojnásobné délky.

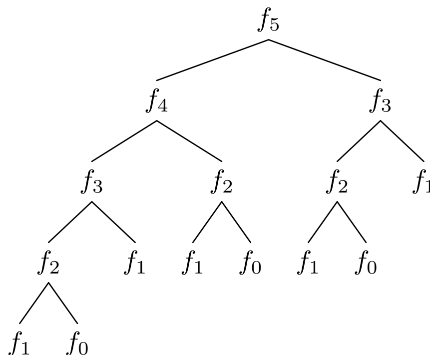
Více se o metodě Rozděl a panuj můžete dozvědět ve stejnojmenné kuchařce.[34]

Předpočítané mezivýsledky

Motivací k této kapitole je následující motto: „Proč počítat něco vícekrát, když nám to stačí spočítat jednou a zapamatovat si to?“.

Velmi často se totiž setkáváme s tím, že něco počítáme stále dokola. Jako příklad si můžeme vzít naši rekurzivní implementaci počítání Fibonacciho čísel z kapitolky Rekurze.

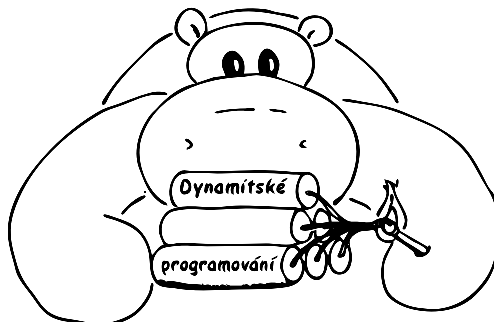
Když se podíváme na výpočet čísla `fib(5)`, vidíme, že pro něj voláme `fib(4)` a `fib(3)`, `fib(4)` volá `fib(3)` a `fib(2)`, `fib(3)` volá `fib(2)` a `fib(1)` a tak dále. Všimli jste si, kolikrát se nám tyhle výpočty opakují? Některá Fibonacciho čísla spočteme totiž zbytečně mnohokrát.



Kdybychom si je namísto opakovaného počítání někde pamatovali, mohli bychom pak odpověď na dotaz na již vypočtené číslo vytáhnout jako králíka z klobouku v konstantním čase. Zavedením jednoho globálního pole, ve kterém si tyto hodnoty pro jednotlivá n budeme pamatovat, nám sníží časovou složitost z $\mathcal{O}(2^n)$ na pěkných $\mathcal{O}(n)$. Takovému postupu se obecně říká *dynamické programování*.

[34] `http://ksp.mff.cuni.cz/viz/kucharky/rozdel-a-panuj`

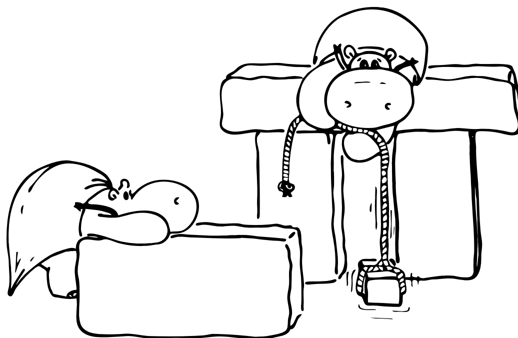
Dynamické programování



Nejprve uveďme na pravou váhu výraz „dynamické" v názvu. Nevystihuje tak úplně podstatu této techniky a jeho historické pozadí je celkem složité, avšak dnes je tento název již tak zažitý, že se už pravděpodobně nezmění.

Slovo „dynamické" částečně odkazuje na to, že se dynamicky (za běhu programu) postupně staví řešení jednodušších problémů, která jsou následně použita pro řešení složitějších. Jeho hlavní podstatou je tedy ukládání a opětovné použití již jednou vypočtených údajů.

Hodí se na úlohy, které se dají dělit na podúlohy, které jsou si podobné a mohou se opakovat. Výsledky takovýchto podúloh si poté ukládáme a při dotazu na stejnou podúlohu vrátíme jen uložený výsledek a výpočet již neprovádíme.



Pro další prohloubení znalostí můžete na našem webu nahlédnout do další kuchařky, tentokrát nesoucí (překvapivě) název Dynamické programování.[35]

[35] http://ksp.mff.cuni.cz/viz/kucharky/dynamicke-programovani

Prefixové součty

Velmi často se nám hodí si ještě před samotným výpočtem předpočítat a uložit nějaké hodnoty, které poté použijeme.

Představme si například problém nalezení souvislého úseku s největším součtem v nějaké posloupnosti kladných i záporných čísel. Že to není úplně jednoduchý příklad, si ukažme na následující posloupnosti:

$$1, -2, 4, 5, -1, -5, 2, 7$$

Máme zde dvě ryze kladné souvislé posloupnosti, každou se součtem 9 ($4, 5$ a $2, 7$). Ale přesto je výhodnější vzít i nějaké záporné hodnoty a vytvořit tak souvislou posloupnost o součtu 12 (zkuste ji nalézt).

Mohlo by nás napadnout, že prostě zkusíme vzít všechny možné začátky a všechny možné konce. To nám dává $\mathcal{O}(n^2)$ možných posloupností (máme n možných začátků a ke každému z nich řádově n možných konců), pro každou posloupnost si spočteme součet (to zvládneme v $\mathcal{O}(n)$) a budeme si pamatovat ten největší nalezený. Celý náš postup tak trvá $\mathcal{O}(n^3)$.

To není pro takhle jednoduchou úlohu zrovna ten nejpěknější čas, zkusme ho zlepšit. Ukážeme si, jak vypočítat součet libovolné posloupnosti v konstantním čase. Celý princip je vlastně až kouzelně jednoduchý, ale zároveň velmi mocný. Na začátku výpočtu si do pomocného pole P stejné délky jako posloupnost na vstupu (té říkejme S) uložíme takzvané *prefixové součty:* i-tý prefixový součet je součet prvních $i+1$ prvků S, neboli $P[i] = S[0] + S[1] + \ldots + S[i]$.

Pro náš ukázkový případ a pro vstupní pole označené S by to dopadlo takto:

i	-1	0	1	2	3	4	5	6	7
$S[i]$		1	-2	4	5	-1	-5	2	7
$P[i]$	0	1	-1	3	8	7	2	4	11

Pole prefixových součtů umíme získat v linerárním čase – prostě jen od začátku procházíme vstupní pole, počítáme si průběžný součet a ten zapisujeme.

Součet libovolného úseku $a \ldots b$ pak získáme v konstantním čase jako prefixový součet od začátku do indexu b minus prefixový součet od začátku do indexu a. Zapsáno programově to pak je:

```
soucet = P[b] - P[a-1];
```

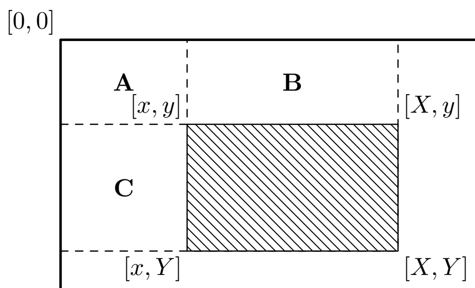
To nám umožňuje snížit čas potřebný na řešení této úlohy na $\mathcal{O}(n^2)$. To je už lepší čas, prozradíme však, že tuto úlohu lze řešit dokonce v lineárním čase, ale to je již nad rámec této kuchařky.

Dvourozměrné prefixové součty

Prefixové součty se dají zobecnit i do více rozměrů, ale princip je vždy stejný. Například dvourozměrné prefixové součty u matice fungují tak, že si předpočítáme součty podmatic začínajících levým vrchním políčkem a končící na indexu $[x, y]$.

Z toho je vidět, že prefixový součet zpravidla obsadí stejně velký prostor jako původní data, v tomto případě tedy budeme mít matici hodnot prefixových součtů končících na zadaných souřadnicích. Jak ale získat součet nějaké podmatice, která se nachází někde „uprostřed" naší matice?

Použijeme stejný princip jako u jednorozměrného případu: Přičteme větší část, kterou chceme započítat, a odečteme od ní části, které započítat nechceme. Pro případ podmatice začínající vlevo nahoře na pozici $[x, y]$ a končící napravo dole na $[X, Y]$ to ilustruje následující obrázek:



Nejdříve přičteme celý prefixový součet končící na pozici $[X, Y]$. Tím jsme ale započítali i části A, B a C z obrázku, které započítat nechceme. Tak odečteme prefixové součty končící na indexech $[X, y]$ a $[x, Y]$. Ale pozor, teď jsme odečetli jednou $A + B$ a jednou $A + C$, tedy část A (prefixový součet končící na pozici $[x, y]$) jsme odečetli dvakrát, musíme ji proto ještě jednou přičíst.

Celý vzorec tedy vypadá takto:

```
soucet = P[X,Y] - P[X,y] - P[x,Y] + P[x,y];
```

Tento princip přičítání a odečítání se dá zobecnit i pro libovolné vyšší rozměry, ale chce to již trošku představivosti, co se má přičíst a kolikrát. Říká se tomu také *princip inkluze a exkluze* a najde použití nejen u vícerozměrných prefixových součtů.

Vyvážení délky předvýpočtu a hlavního výpočtu

Správně vyvážit, kolik času můžeme věnovat na předvýpočet a kolik času na hlavní výpočet, je velice důležitá věc a spousta i zkušenějších řešitelů v tom občas chybuje. Přitom to při troše počítání není vůbec nic složitého.

Jako první je potřeba vědět, kolikrát nám předvýpočet během běhu programu pomůže. Předvýpočtem si totiž vybudujeme za nějaký čas určitou datovou strukturu, pomocí které pak dokážeme rychle odpovídat na zadané dotazy.

Označme si počet takovýchto dotazů, které program za běhu dostane, jako Q. Buď to může být hodnota přímo ze zadání typu „Zkonstruujte datovou strukturu pro n hodnot, která zvládne rychle odpovídat na dotazy daného typu, a očekávejte řádově m dotazů", nebo se může jednat o nějaký interní dotaz

v rámci běhu programu (příklad interního dotazu je třeba výše uvedený algoritmus na hledání souvislé podposloupnosti s co největším součtem, který se za běhu ptal na součty nějakých úseků).

Dále si označme jako $\mathcal{O}_p$ čas, který nám zabere předvýpočet a jako $\mathcal{O}_q$ čas, který nám ušetří každý předvypočítaný dotaz. Celkový čas, který ušetříme, je pak vlastně $Q \cdot \mathcal{O}_q$. Pokud je tento čas řádově větší než $\mathcal{O}_p$, pak má předvýpočet smysl.

Naopak nemá smysl trávit předvýpočtem řádově více času, než by trval samotný výpočet bez použití předpočítaných hodnot.

Jako příklad uvažujme problém o velikosti n, u kterého máme tři možnosti, které můžeme zvolit. Můžeme buď předvýpočet úplně vynechat a na každý dotaz odpovídat v čase $\mathcal{O}(n)$, nebo provést předvýpočet v čase $\mathcal{O}(n \log n)$ a poté odpovídat na každý dotaz v čase $\mathcal{O}(\log n)$, nebo provést předvýpočet v čase $\mathcal{O}(n^2)$ a pak odpovídat v čase $\mathcal{O}(1)$ na dotaz.

Kdy se nám co hodí?

- Pokud bychom dostali jen jeden dotaz, nemá smysl si cokoliv předpočítávat a odpovíme jednou v čase $\mathcal{O}(n)$.
- Pokud bude dotazů řádově n, má smysl použít první předvýpočet. Pak budeme mít čas na předvýpočet i na samotný výpočet $\mathcal{O}(n \log n)$, což je optimum.
- Naopak pokud by dotazů bylo řádově n^2 nebo víc, tak se nám již první předvýpočet nevyplatí, dostali bychom se totiž na čas $\mathcal{O}(n^2 \log n)$. Zde se hodí použít druhý, delší předvýpočet a pak se dostat na celkovou časovou složitost $\mathcal{O}(n^2 + n^2 \cdot 1) = \mathcal{O}(n^2)$.

Hladové algoritmy

Věřte nebo ne, ale i počítač se někdy cítí hladový. Po namáhavé práci mu můžeme dopřát to potěšení, aby si ukousl co největší kus dat. A ukážeme, že někdy je to i ku prospěchu. Řeč bude o *hladových algoritmech*.

Takovými algoritmy rozumíme ty, které hledají řešení celé úlohy po jednotlivých krocích a splňují následující dvě podmínky:

- V každém kroku zvolí lokálně nejlepší řešení.
- Provedené rozhodnutí již nikdy neodvolává (tedy nebacktrackuje).

Lokálně nejlepší řešení je takové, které v aktuálním kroku vybere tu možnost, která nám na tomto místě nejvíce pomůže (bez jakéhokoliv ohledu na globální stav). Může to být třeba nejvyšší hodnota, nebo nejkratší cesta v grafu.

Pokud ale od hladového algoritmu chceme, aby nám našel globálně nejlepší řešení, musí naše úloha splnit předpoklad, že si výběrem lokálně nejlepšího řešení nezhoršíme to globální. Tento předpoklad se nedá formulovat obecně a je nutné se nad ním zamyslet zvlášť u každé úlohy.

Příklady hladových algoritmů

První hladovou úlohou bude (jak jinak) automat na jídlo vracející mince. Automat by měl vracet peníze nazpět tak, aby vrátil daný obnos v co možná nejmenším počtu mincí. Pro náš měnový systém (máme mince hodnot 1, 2, 5, 10, 20 a 50 Kč) lze tuto úlohu řešit hladovým algoritmem – v každém kroku algoritmu vrátíme tu největší minci, kterou můžeme (tedy pro vrácení 42 Kč to bude $42 = 20 + 20 + 2$ Kč).

Měnové systémy většiny států jsou postavené tak, aby fungovaly takto pěkně, neplatí to ale obecně. Zkusme si vrátit 42 Kč, pokud bychom měli jen mince hodnoty 20, 10 a 4 Kč. Správným řešením je $42 = 20 + 10 + 4 + 4 + 4$ Kč, hladový algoritmus by ale zkusil vrátit $42 = 20 + 20 + \ldots$ a tady by selhal.

Dále se velmi často dají hladovým algoritmem řešit nějaké úlohy přidávání nebo odebírání skupin prvků. Typickým příkladem je třeba rozvržení naplánovaných přednášek do učeben. Seřadíme si začátky přednášek podle času a postupně bereme jednu za druhou a umísťujeme je do volných učeben s nejnižším číslem.

Tím jsme si určitě nic nerozbili, protože v nějaké učebně přednáška být musí. Určitě budeme potřebovat tolik učeben, kolik je maximálně přednášek v jeden čas, a díky tomu si umístěním přednášky do nějaké učebny nezablokujeme místo pro jinou přednášku, jelikož nám vždy zbude dostatek volných učeben.

Kdybychom ale naopak měli pevně zadaný počet učeben a chtěli jsme do nich umístit co možná nejvíce přednášek, tak se již nejedná o úlohu řešitelnou hladovým algoritmem, v takovém případě je potřeba zvolit nějaký chytřejší postup.

Závěr

Doufám, že jste si z tohoto rozsáhlého textu odnesli nějaké nové znalosti a poznatky, které vám pomohou nejen v řešení KSP.

Pokud jste začínajícími řešiteli, zkuste s pomocí kuchařky vyřešit několik lehčích úloh a jejich řešení poslat – nově nabyté znalosti je totiž nejlepší co nejdříve protrénovat. Nic si nedělejte z toho, pokud napoprvé nevyřešíte všechno, s postupným zkoušením se budou vaše znalosti jen zlepšovat. Zkušenější řešitelé možná v kuchařce nalezli nějaké ujasnění pojmů, či si některé techniky osvěžili.

A pokud tento text považujete za dobrý, budeme jen rádi, pokud ho doporučíte svým kamarádům a spolužákům, kteří chtějí s programováním začít.

Úvodním kurzem vaření podle kuchařky vás provedl

Jirka Setnička

Kuchařka druhé série – hledání v textu

Řetězec je v podstatě jakákoliv 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 algoritmů s nimi pracujících) najdeme v biologii. Například DNA není o mnoho více, než chytré uložení posloupnosti čtyř znaků/nukleových bazí – 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 do práce s řetězci popíšeme dva *stavební kameny* textových algoritmů, což bude datová struktura pro slovníky – *trie* a vyhledání v textu s předzpracováním hledaného slova a jeho rozšíření pro více slov. S jejich znalostí pak bude mnohem snazší vymýšlet 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.

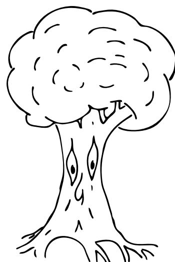
Terminologie dovoluje zepředu i zezadu odstranit prázdný řetězec – to znamená, že slovo je samo sobě prefixem i suffixem. Pokud chceme mluvit o prefixech, suffixech nebo obecně podřetězcích, kde jsme museli alespoň jeden znak odtrhnout, označíme takové podřetězce jako *vlastní*.

Pro některá použití řetězců je důležité, abychom je mohli porovnávat – když máme řetězce R a S, chceme umět 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 $\mathtt{0} < \mathtt{1}$ se často používá „telefonní“ $\mathtt{A} = \mathtt{a} < \mathtt{B} = \mathtt{b} < \ldots < \mathtt{Z} = \mathtt{z}$, 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 $\varepsilon < \mathtt{A} < \mathtt{AUTO} < \mathtt{AUTOBUS} < \mathtt{AUTOGRAM} < \mathtt{AUTOR} < \mathtt{BAMBITKA} < \mathtt{BARNABAS} < \mathtt{Z}$.



Adresář pomocí trie

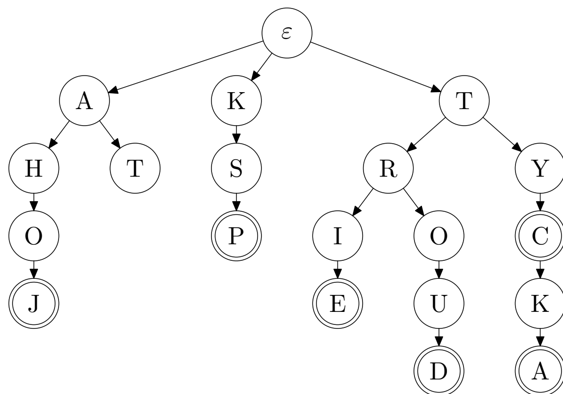
Typický „textový“ problém je udržování množiny řetězců – můžete si představit třeba slovník. Slova ve slovníku si chceme šikovně předzpracovat, abychom pak mohli efektivně odpovídat na otázky typu: „Je slovo S obsaženo ve slovníku?“ Můžeme také po předzpracování chtít přidávat nové položky, nebo i odebírat staré.

Pokud bychom nemuseli odebírat slova, můžeme použít hešování, které je rychlé a účinné. Více o něm najdete v hešovací kuchařce.[36] Má však tu nevýhodu, že při velkém zaplnění se začne chovat pomaleji a mírně nepředvídatelně.

Ukážeme si jiné řešení, které je také asymptoticky rychlé a není ani příliš náročné na paměť. Využívá stromové struktury a říká se mu *trie* (vyslovujeme česky „tryje“ a anglicky jako část slova „retrieval“, z něhož slovo trie vzniklo). V češtině se občas používá také označení „písmenkový strom.“

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.

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

Z takto popsané trie bohužel nepoznáme, kde končí slovo ze slovníku a kde končí jen jeho prefix. Standardní způsoby, jak to vyřešit, jsou dva: buď si do každého vrcholu přidáme informaci o tom, je-li koncem celého slova, nebo ne (jak je to naznačeno dvojitými kroužky v obrázku), anebo si rozšíříme abecedu o speciální znak, který se v ní předtím nevyskytoval – třeba `$` – a pak všem slovům přilepíme tento `$` na konec.

Budeme-li se později ptát, bylo-li slovo ve slovníku, po průchodu trií zkontrolujeme ještě, jestli z konečného vrcholu vede hrana odpovídající znaku `$`.

Ještě jsme si nerozmysleli, jak budeme v jednotlivých vrcholech trie reprezentovat hrany do delších prefixů. Abychom mohli vyhledávat skutečně lineárně, potřebovali bychom umět v konstantním čase odpovědět na otázku „má vrchol P potomka přes hranu se znakem c?“.

Abychom zajistili konstantní čas odpovědi, museli bychom mít v každém vrcholu pole indexované znaky abecedy. To ovšem znamená, že takové pole budeme muset vytvořit, a tedy alokovat $|\Sigma|$ políček v každém znaku.

To zvýší paměťovou náročnost trie (a časovou náročnost její stavby) na $\mathcal{O}(D \cdot |\Sigma|)$, kde D značí velikost vstupu, čili součet délek všech slov ve slovníku. To je naprosto přijatelné pro malé abecedy, ale už pro $\{\texttt{A-Z}, \texttt{a-z}\}$ 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 $\{\texttt{0}, \texttt{1}\}$. 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.

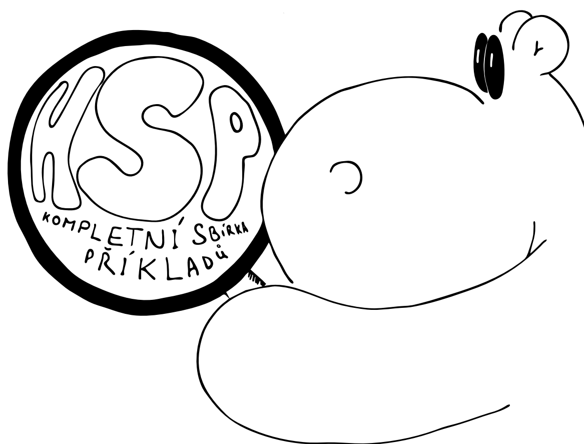
Poznámky

- 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; jednak 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*.[37]

Cvičení

- Řekněme, že chceme slovník na vstupu setřídit v lexikografickém pořadí (definovaném v sekci „Jak řetězce chápat“). Problémem pro klasické třídící algoritmy je to, že porovnání dvou řetězců není bohužel konstantně rychlé. Vymyslete způsob, jak setřídit takový slovník rychle pomocí trie.
- *Komprese trie.* Co kdybychom chtěli odstranit přebytečné vrcholy trie, tedy ty, v nichž se slova nevětví? Rozmyslete si, jestli by něčemu vadilo místo takovýchto cest mít jen jednotlivé hrany. Zesložití se konstrukce nebo vyhledávání? Mimochodem, je celkem jasné, že takováto *komprimovaná trie* přinese jen konstantní zrychlení dotazů i prostoru, a tak na soutěžích apod. stačí použít základní variantu.

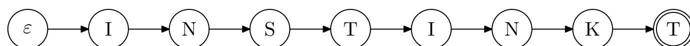


Vyhledávání v textu

Začátek situace je asi zřejmý – máme na vstupu zadán dlouhý text a krátké slovo. Slovo si můžeme nějak předzpracovat, načež projdeme co nejrychleji text a nahlásíme jeden nebo všechny výskyty slova. Zajímají nás při tom i výskyty, které se navzájem překrývají: v textu `NANANA` se slovo `NANA` vyskytuje dvakrát. Často se hovoří o „hledání jehly v kupce sena“, pročež se textu přezdívá *seno* a hledanému slovu *jehla*. Délku jehly označíme J a délku textu S.

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

Představme si nejdříve hledané slovo jako spojový seznam, třeba slovo INSTINKT:



Mohli bychom text začít procházet znak po znaku a kontrolovat, zda se text shoduje s naším slovem/spojovým seznamem. Pokud by si znaky odpovídaly, skočíme na další znak z textu a i na další znak v seznamu. Co když se ale neshodují? Pak nemůžeme jen skočit na další znak textu – co kdybychom v textu narazili na slovo INSTINSTINKT?

Musíme se tedy vrátit nejen na začátek spojového seznamu, ale i zpátky v textu na druhý znak, který jsme označili jako odpovídající, a zkoušet porovnávat s jehlou znovu od začátku. To už naznačuje, že takto získaný algoritmus nebude lineární, protože se musí vracet zpět v textu o délku jehly.

Sice je předchozí popis skutečně v nejhorším případě složitý $\mathcal{O}(S \cdot J)$, avšak stačí malá úprava a složitost přejde na lineární $\mathcal{O}(S + J)$. Ve skutečnosti algoritmus nezpomalovalo vracení se – za špatnou složitost mohl fakt, že jsme se vraceli *příliš zpátky*.

Třeba v našem příkladu s textem INSTINSTINKT se nemusíme vracet ve spojovém seznamu na začátek, jakmile načteme INSTINS. Mohli jsme se vrátit jen na druhý znak, tedy do prvního N, a pak kontrolovat, jaký znak pokračuje dál. Když následuje S jako v našem případě, můžeme pokračovat dále v čtení a nevracíme se v textu. Kdyby text byl jiný, třeba INSTINB, vrátili bychom se po načtení B na začátek spojového seznamu a v textu bychom pokračovali dále bez zastavení.

Pro každý znak ve spojovém seznamu si tedy určíme políčko spojového seznamu, na které skočíme, pokud se následující znak v textu liší od toho očekávaného. Pořadové číslo tohoto políčka nám poradí tzv. *zpětná funkce* F, což bude funkce definovaná pomocí pole, kde $F[i]$ bude pořadové číslo políčka, na které se má skočit z políčka číslo i. Porovnávat pak budeme s následujícím znakem. Pokud $F[i] = 0$, znamená to, že máme začít porovnávat úplně od prvního znaku jehly.

Pokud máte rádi grafovou terminologii, můžete se na náš spojový seznam dívat jako na graf a hovořit o *zpětných hranách*.

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

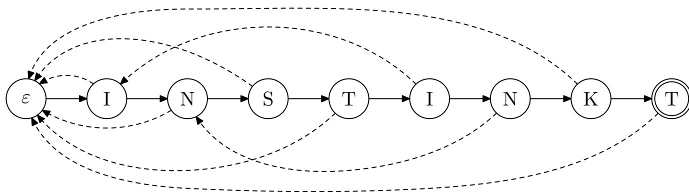
Abychom náš požadavek upřesnili, zamyslíme se nad zpětným políčkem pro jiné slovo. Co kdyby jehlou bylo slovo ABABABC a my určovali zpětné políčko pro ABABAB? Kdybychom ukázali na první písmenko B, nebylo by to správně, protože

pak bychom pro text `ABABABABC` nezahlásili výskyt jehly, což je jasná chyba. Musíme se vrátit už na `ABAB`!

Zajímá nás tedy ne libovolný suffix, který je stejný jako začátek, ale nejdelší takový konec/suffix. A ještě navíc ne jen ten nejdelší, ale nejdelší „netriviální“ – slovo `INSTIN` je samo sobě prefixem a suffixem, ale zpětná funkce pro `N` by se neměla cyklit, měla by vést zpátky.

Řekněme to tedy znova, zcela formálně: pokud bychom právě určovali hodnotu zpětné funkce pro znak číslo i, kterému odpovídá prefix P, pak její hodnota bude *délka nejdelšího vlastního suffixu* slova P, pro který ještě platí, že *je zároveň prefixem* P.

Pro slovo `INSTINKT` vypadá spojový seznam se zpětnou funkcí (zakreslenou pomocí ukazatelů) takto:



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

Poperme se nejdříve s tou první. Pro každý znak vstupního textu mohou nastat dva případy: Buď znak rozšiřuje aktuální prefix, nebo musíme použít zpětnou funkci. První případ má jasně konstantní složitost, druhý je horší, neboť zpětná funkce může být pro jeden znak volána až J-krát.

Při každém volání však klesne pořadové číslo aktuálního stavu (políčka) alespoň o jedna, zatímco kdykoliv stav prodlužujeme, roste jen o jeden znak. Proto všech zkrácení dohromady může být nejvýše tolik, kolik bylo všech prodloužení, čili kolik jsme přečetli znaků textu. Celkem je tedy počet kroků automatu lineární v délce textu, tj. $\mathcal{O}(S)$.

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)
```

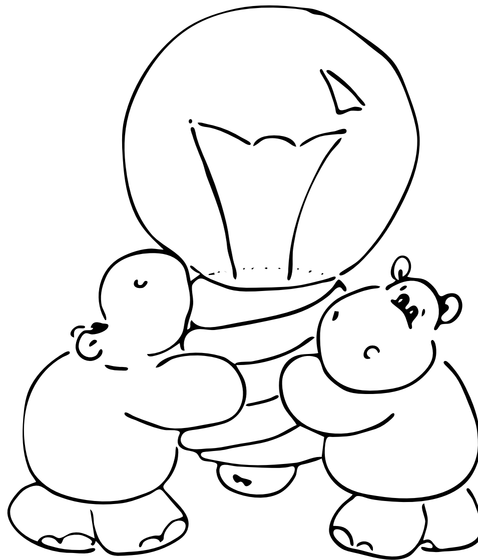
Poznámky

- 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.



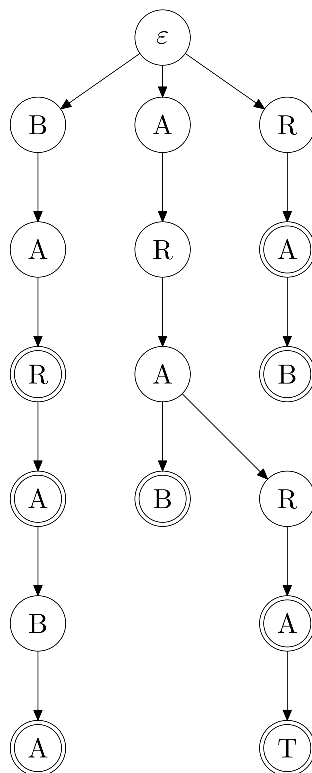
Vyhledávání jehelníčku

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. V 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`.

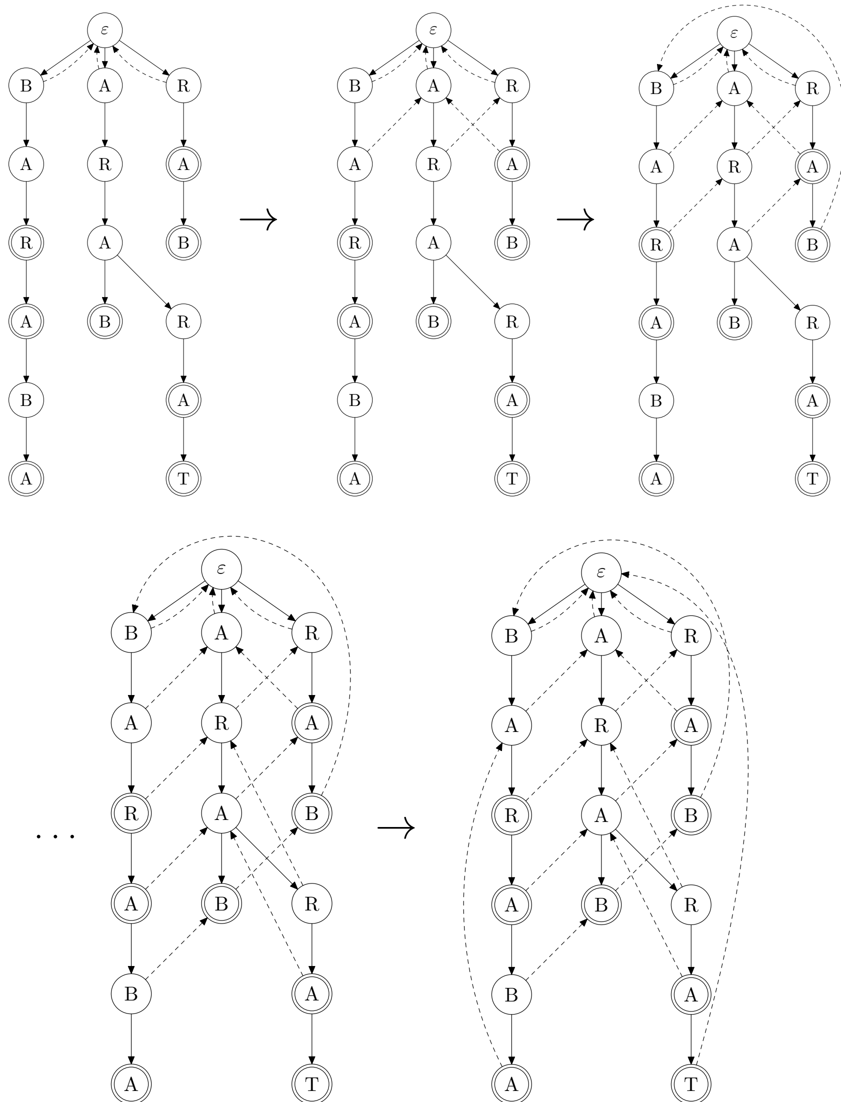
Můžeme se ale opřít o stejný trik, jako při konstrukci KMP. Budeme opět vyhledávat nejdelší vlastní suffix. Kam dojde výpočet po jeho vyhledání, tam povede zpětná hrana.

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 tak 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.

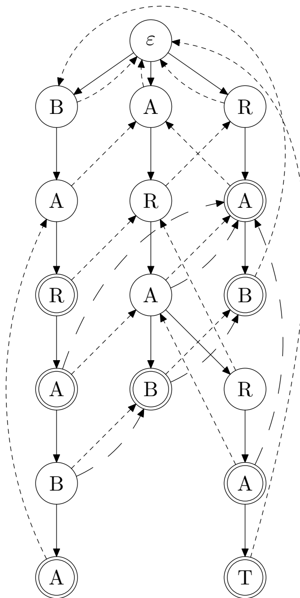


Například pro slovo `BARABA` bychom mohli vyhledávat `ARABA` v již zkonstruované části automatu, ale proč to dělat celé, když jsme při konstrukci předchozí vrstvy vyhledávali `ARAB` při konstrukci zpětné hrany pro `BARAB`?

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`.

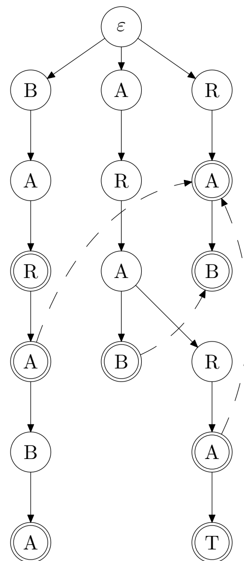
Nenalezl však všechno. Chybí mu např. `ARAB`, který začíná druhým znakem a končí pátým. Dále chybí několik výskytů `RA` a jeden `RAB`.

Když byl algoritmus na pátém znaku, byl ve stavu `BARAB`, jehož suffixem je `ARAB`. Obecně na suffixy zapomínáme. Na rozdíl od KMP, kde suffix aktuálního stavu nikdy nebyl jehla, tady jehlou být může.

V každém stavu bychom tedy měli projít veškeré suffixy a zkontrolovat je, jestli náhodou nejsou jehlami. Jak najdeme všechny suffixy? Projdeme postupně po zpětných hranách až do kořene. Má to jen jeden problém – je to pomalé.

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

Všimněme si však, že většinou zpětných hran jsme prošli úplně zbytečně. Předpočítáme si tedy *zkratky* – z vrcholu vede zkratka do nejdelšího jeho suffixu, který je jehlou. Na obrázku jsou vyznačeny dlouze čárkovanými šipkami.



Předpočítání zpětných hran časovou složitost konstrukce automatu jistě nezhorší, neboť vyžaduje v nejhorším případě projít všechny zpětné hrany ještě jednou.

Potřebujeme-li ohlásit všechny výskyty slov včetně pozice, kde se nacházejí, jsme hotovi. Výsledná časová složitost prohledávání v takovém případě bude $\mathcal{O}(S+O)$, resp. $\mathcal{O}(S \cdot \log|\Sigma| + O)$, kde O je velikost výstupu – počet výskytů všech slov.

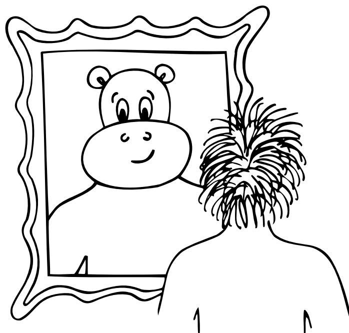
Celková časová složitost prohledávání včetně stavby automatu tedy bude $\mathcal{O}(O+S+J \cdot |\Sigma|)$, resp. $\mathcal{O}(O+(S+J) \cdot \log|\Sigma|)$.

Jak velký může být výstup? Obecně až S^2. Extrémně velký výstup je možné vygenerovat třeba slovníkem obsahujícím všechny prefixy slova `AAAA...A` délky S a senem taktéž `AAAA...A` délky S. Automat pak hlásí výskyt pro každé podslovo, kterých je řádově S^2.

Pokud nám stačí u každého slova jen počet výskytů, nemusíme zoufat – závislost na počtu výskytů umíme odstranit.

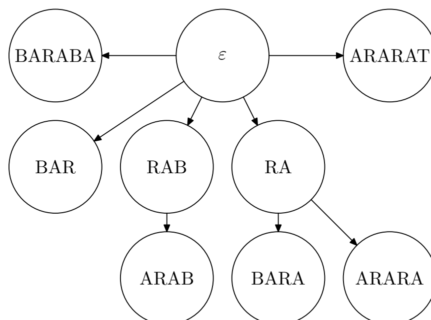
Použijeme trik – na každé pozici započítáme pouze nejdelší jehlu, která tam končí (u každé jehly si budeme udržovat čítač). Nebudeme tedy v každém kroku poskakovat po zkratkách až do aleluja, ale maximálně jednou. Díky tomu nám z časové složitosti zmizí velikost výstupu.

V našem příkladu se senem `BARABARARAT` tedy na konci budeme mít uloženo, že `ARAB` se vyskytnul 1×, `ARARA` 1×, `ARARAT` 1×, `BAR` 2×, `BARA` 2× a `BARABA` 1×. `RA` a `RAB` nemají hlášený žádný výskyt.



Nyní si zkonstruujeme strom jenom ze zkratek a pro každý vrchol spočítáme součet celého jeho podstromu.

Tedy po přepočtu bude mít `RA` tři výskyty a `RAB` jeden výskyt; celkový počet výskytů pak bude 12.



Poznámky

- Dalším krokem po KMP a Aho-Corasickové jsou konečné automaty a regulární výrazy, o kterých jsme měli seriál ve 23. ročníku.
- Není moc rozumné snažit se implementovat Aho-Corasickovou v rozumné době například při soutěži, pokud tento algoritmus nemáte opravdu pod kůží. Radši zkuste použít hešování, pokud budete něco takového potřebovat.

Cvičení

- Redukci o velikost výstupu můžeme provést i pro případ, kdy výstup nebudeme vypisovat, ale stačí nám mít jej uložený v paměti. Vymyslete vhodnou úpravu triku s čítačem.
- Zkuste si naimplementovat Aho-Corasickovou vlastnoručně ve svém oblíbeném jazyce, abyste si byli jisti, že doopravdy chápete všechny záludnosti tohoto algoritmu.

Martin Böhm, Jan Matějka, Martin Mareš a Petr Škoda

Kuchařka třetí série – rozděl a panuj

Dnešní díl programátorské kuchařky se bude zabývat algoritmy založenými na metodě *Rozděl a panuj.* Slušelo by se začít tím, jaká je myšlenka této metody: Často se setkáme s úlohami, které lze snadno rozdělit na nějaké menší úlohy a z jejich výsledků zase snadno složit výsledek původní velké úlohy. Přitom menší úlohy můžeme řešit opět týmž algoritmem (zavoláme si ho rekurzivně), leda by již byly tak maličké, že dokážeme odpovědět triviálně bez jakéhokoliv počítání. Zkrátka jak říkali staří římští císařové: Divide et impera. Uveďme si pro začátek jeden ilustrační příklad:

Quicksort

QuickSort (alias QS) je algoritmus pro třídění posloupnosti prvků. Už jste si o něm mohli přečíst v kuchařce o třídění. Tentokrát se na něj podíváme trochu podrobněji a navíc nám poslouží jako ingredience pro další algoritmy.

QS v každém svém kroku zvolí nějaký prvek (budeme mu říkat *pivot*) a přerovná prvky v posloupnosti tak, aby napravo od pivota byly pouze prvky větší než pivot a nalevo pouze menší. Pokud se vyskytnou prvky rovné pivotu, můžeme si dle libosti vybrat jak levou, tak pravou stranu posloupnosti, funkčnost algoritmu to nijak neovlivní. Tento postup pak rekurzivně zopakujeme zvlášť pro prvky nalevo a zvlášť pro prvky napravo od pivota, a tak získáme setříděnou posloupnost.

Implementací QS je mnoho a mimo jiné se liší způsobem volby pivota. My si předvedeme jinou, než jsme ukazovali v třídící kuchařce (hlavně proto, že se nám z ní pak budou snadno odvozovat další algoritmy), a pro jednoduchost budeme jako pivota volit poslední prvek zkoumaného úseku:

```
pole = [1, 2, 8, 42, 9, 17, -5, 20, 2]

# Přerovnej pole od levého do pravého indexu
def prerovnej(pole, L, P):
    pivot = pole[P - 1]
    # i je nejlevější nepřerovnaný prvek
    i = L
    # j je aktuální probíraný prvek
    for j in range(L, P - 1):
        if (pole[j] <= pivot):
            # Prohodíme prvek s nejlevějším
            pole[i],pole[j] = pole[j],pole[i]
            i += 1
    # Dáme pivota na správné místo
    pole[P - 1],pole[i] = pole[i],pole[P - 1]
    return i

def quicksort(pole, levy_index, pravy_index):
```

```
        if (levy_index >= pravy_index):
            return
        # Přerovnáme úsek a najdeme pivota...
        p = prerovnej(pole, levy_index, pravy_index)
        # ... a zavoláme se rekurzivně na podúseky
        quicksort(pole, levy_index, p)
        quicksort(pole, p + 1, pravy_index)
```


Bohužel volit pivota právě takto je docela nešikovné, protože se nám snadno může stát, že si vybereme nejmenší nebo největší prvek v úseku (rozmyslete si, jak by vypadala posloupnost, ve které to nastane pokaždé), takže dostaneme posloupnost délky N, rozdělíme ji na úseky délek $N-1$ a 1, načež pokračujeme s úsekem délky $N-1$, ten rozdělíme na $N-2$ a 1, atd. Přitom pokaždé na přerovnání spotřebujeme čas lineární s velikostí úseku, celkem tedy $\mathcal{O}(N+(N-1)+(N-2)+\ldots+1)=\mathcal{O}(N^2)$.

Na druhou stranu pokud bychom si za pivota vybrali vždy *medián* z právě probíraných prvků (tj. prvek, který by se v setříděné posloupnosti nacházel uprostřed; pro sudý počet prvků zvolíme libovolný z obou prostředních prvků), dosáhneme daleko lepší složitosti $\mathcal{O}(N\log N)$. Ale raději si to dokažme pořádně:

Přerovnávací část algoritmu běží v čase lineárním vůči počtu prvků, které máme přerovnat. V prvním kroku QS pracujeme s celou posloupností, čili přerovnáme celkem N prvků. Následuje rekurzivní volání pro levou a pravou část posloupnosti (obě dlouhé $(N-1)/2\pm1$); přerovnávání v obou částech dohromady trvá opět $\mathcal{O}(N)$ a vzniknou tím části dlouhé nejvýše $N/4$. Zanoříme-li se v rekurzi do hloubky k, pracujeme s částmi dlouhými nejvýše $N/2^k$, které dohromady dají nejvýše N (všechny části dohromady dají prvky vstupní posloupnosti bez těch, které jsme si už zvolili jako pivoty). V hloubce $\lceil\log_2 N\rceil$ už jsou všechny části nejvýše jednoprvkové, takže se rekurze zastaví. Celkem tedy máme $\lceil\log_2 N\rceil$ hladin (hloubek) a na každé z nich trávíme lineární čas, dohromady $\mathcal{O}(N\log N)$.

V tomto důkazu jsme se ale dopustili jednoho podvodu: Zapomněli jsme na to, že také musíme medián umět najít. Jak z této nepříjemné situace ven?

- *Naučit se počítat medián.* Ale jak?
- *Spokojit se se „lžimediánem“:* Kdybychom si místo mediánu vybrali libovolný prvek, který bude v setříděné posloupnosti „v prostřední polovině“ (čili alespoň čtvrtina prvků bude větší a alespoň čtvrtina menší než on), získáme také složitost $\mathcal{O}(N\log N)$, neboť úsek délky N rozložíme na úseky, které budou mít délky nejvýše $(1-1/4)\cdot N$, takže na k-té hladině budou úseky délek nejvýše $(1-1/4)^k\cdot N$, čili hladin bude maximálně $\log_{1-1/4} N=\mathcal{O}(\log N)$. Místo 1/4 by fungovala i libovolná jiná konstanta mezi nulou a jedničkou, ale ani to nám nepomůže k tomu, abychom uměli lžimedián najít.
- *Recyklovat pravidlo* typu „vezmi poslední prvek“ a jen ho trochu vylepšit. To bohužel nebude fungovat, protože pokud budeme při výběru pivota hle-

dět jenom na konstantní počet prvků, bude poměrně snadné přijít na vstup, pro který toto pravidlo bude dávat kvadratickou složitost, i když obvykle půjde dokázat, že takových vstupů je „málo“. (Proto se také QS často implementuje právě s náhodnou volbou pivota.)

- *Volit pivota náhodně* ze všech prvků zkoumaného úseku. K náhodné volbě samozřejmě potřebujeme náhodný generátor a s těmi je to svízelné, ale zkusme na chvíli věřit, že jeden takový máme nebo alespoň že máme něco s podobnými vlastnostmi. Jak nám to pomůže? Náhodně zvolený pivot nebude sice přesně uprostřed, ale s pravděpodobností 1/2 to bude lžimedián, takže po průměrně dvou hladinách se ke lžimediánu dopracujeme. Proto časová složitost takového randomizovaného QS bude v průměru 2-krát větší, než lžimediánového QS, čili v průměru také $\mathcal{O}(N \log N)$. Jednoduše řečeno, zatímco fixní pravidlo nám dalo dobrý čas pro průměrný vstup, ale existovaly vstupy, na kterých bylo pomalé, randomizování nám dává dobrý průměrný čas pro všechny možné vstupy.

Hledání k-tého nejmenšího prvku

Nad QuickSortem jsme zvítězili, ale současně jsme při tom zjistili, že neumíme rychle najít medián. To tak nemůžeme nechat, a proto rovnou zkusíme vyřešit obecnější problém: najít k-tý nejmenší prvek (medián dostáváme pro $k = \lfloor N/2 \rfloor$).

První řešení této úlohy se nabízí samo. Načteme posloupnost do pole, prvky pole setřídíme nějakým rychlým algoritmem a kýžený k-tý nejmenší prvek nalezneme na k-té pozici v nyní již setříděném poli. Má to však jeden háček. Pokud prvky, které máme na vstupu, umíme pouze porovnat, pak nedosáhneme lepší časové složitosti (a to ani v průměrném případě) než $\mathcal{O}(N \log N)$ – rychleji prostě třídit nelze, důkaz můžete najít například v třídící kuchařce.

O něco rychlejší řešení je založeno na výše zmíněném algoritmu QuickSort (často se mu proto říká QuickSelect). Opět si vybereme pivota a posloupnost rozdělíme na prvky menší než pivot, pivota a prvky větší než pivot (pro jednoduchost budeme předpokládat, že žádné dva prvky posloupnosti nejsou stejné).

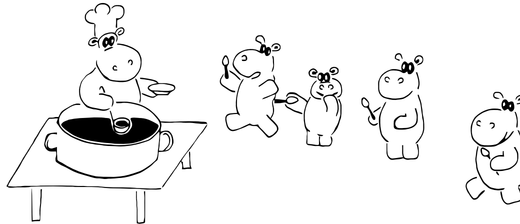
Pokud se pivot nalézá na k-té pozici, je to hledaný k-tý nejmenší prvek posloupnosti, protože právě $k-1$ prvků je menších. Zbývají dva případy, kdy tomu tak není. Pakliže je pozice pivota v posloupnosti větší než k, pak se hledaný prvek nalézá nalevo od pivota a postačí rekurzivně najít k-tý nejmenší prvek mezi prvky nalevo. V opačném případě, kdy je pozice pivota menší než k, je hledaný prvek v posloupnosti napravo od pivota. Mezi těmito prvky však nebudeme hledat k-tý nejmenší prvek, ale $(k-p)$-tý nejmenší prvek, kde p je pozice pivota v posloupnosti.

Časovou složitost rozebereme podobně jako u QuickSortu. Nešikovná volba pivota dává opět v nejhorším případě kvadratickou složitost. Pokud bychom naopak volili za pivota medián, budeme nejprve přerovnávat N prvků, pak jich zbude nejvýše $N/2$, pak nejvýše $N/4$ atd., což dohromady dává složitost

$\mathcal{O}(N + N/2 + N/4 + \ldots + 1) = \mathcal{O}(N)$. Pro lžimedián dostaneme rovněž lineární složitost a opět stejně jako u QS můžeme nahlédnout, že náhodnou volbou pivota dostaneme v průměru stejný čas jako se lžimediánem.

Program bude velmi jednoduchý, využijeme-li přerovnávací proceduru od QuickSortu:

```
def kty(pole, k, L, P):
    pivot = prerovnej(pole, L, P)

    if (k == pivot):
        return pole[pivot]
    if (k < pivot):
        return kty(pole, k, L, pivot)
    else:
        return kty(pole, k, pivot + 1, P)
```



k-tý nejmenší podruhé, tentokrát lineárně a bez náhody

Existuje však algoritmus, který řeší naši úlohu lineárně, a to i v nejhorším případě. Je založený na triku: zvolit vhodného pivota (jak ukážeme, bude to jeden ze lžimediánů) rekurzivním voláním téhož algoritmu. Zařídíme to takto:

1. Pokud jsme dostali méně než 6 prvků, použijeme nějaký triviální algoritmus, například si posloupnost setřídíme a vrátíme k-tý prvek setříděné posloupnosti.
2. Rozdělíme prvky posloupnosti na pětice; pokud není počet prvků dělitelný pěti, poslední pětici necháme nekompletní.
3. Spočítáme medián každé pětice. To můžeme provést například rekurzivním zavoláním celého našeho algoritmu, čili v důsledku tříděním. (Také bychom si mohli pro 5 prvků zkonstruovat rozhodovací strom s nejmenším možným počtem porovnání, což je rychlejší, ale jednak pouze konstanta-krát, jednak je to daleko pracnější.)
4. Máme tedy $N/5$ mediánů. V nich rekurzivně najdeme medián m (označíme mediány pětic za novou posloupnost a na ní začneme opět od prvního bodu).
5. Přerovnáme vstupní posloupnost po quicksortovsku a jako pivota použijeme prvek m. Po přerovnání je pivot, podobně jako v předchozím algoritmu, na $(z+$

1)-ní pozici v posloupnosti, kde z je počet prvků s menší hodnotou, než má pivot.

6. Opět, podobně jako u předchozího algoritmu, pokud je $k = z + 1$, pak je právě pivot m k-tým nejmenším prvkem posloupnosti. V případě, že tomu tak není a $k < z + 1$, budeme hledat k-tý nejmenší prvek mezi prvními z členy posloupnosti, v opačném případě, kdy $k > z + 1$, budeme hledat $(k - z + 1)$-ní nejmenší prvek mezi posledními $n - z - 1$ prvky.

```
# Příprava pro přerovnávání z QuickSortu
# -> chceme pivota jako poslední prvek
def prerovnej_podle(pole, L, P, podle):
    q = L
    while (pole[q] != podle):
        q += 1
    pole[q], pole[P-1] = pole[P-1], pole[q]
    return prerovnej(pole, L, P)

def kty(pole, k, L, P):
    pocet = P - L
    # Jednoduché případy
    if (pocet <= 1):
        return pole[L]
    if (pocet <= 5):
        quicksort(pole, L, P)
        return pole[k]

    # Rozdělení na pětice
    petic = (pocet + 4) // 5
    mediany = [0] * petic
    for i in range(0, pocet, 5):
        if (i + 5 > pocet):
            # Ignorujeme neúplnou pětici
            break
        quicksort(pole, L + i, L + i + 5)
        mediany[i // 5] = pole[i + 2]

    # Nalezneme medián mediánů pětic
    median = kty(mediany, petic//2, 0, petic)
    pivot = prerovnej_podle(pole, L, P, median)

    if (pivot == k):
        return median
    if (k < pivot):
        return kty(pole, k, L, pivot)
    else:
```

```
return kty(pole, k, pivot + 1, P)
```

Zbývá dokázat, že tato dvojitá rekurze má slíbenou lineární složitost. Zkusme se proto podívat, kolik prvků posloupnosti po přerovnání je větších než prvek m. Všech pětic je $N/5$ a alespoň polovina z nich (tedy $N/10$) má medián menší než m. V každé takové pětici pak navíc najdeme dva prvky menší než medián pětice (takže jsou zde tři prvky menší, než m). Celkem tak existuje alespoň $3/10 \cdot N$ prvků menších než m. Větších tedy může být maximálně $7/10 \cdot N$. Symetricky ukážeme, že i menších prvků může být nejvýše $7/10 \cdot N$.

Rozdělení na pětice, hledání mediánů pětic a přerovnávání trvá lineárně, tedy nejvýše cN kroků pro nějakou konstantu $c > 0$. Pak už algoritmus pouze dvakrát rekurzivně volá sám sebe: nejprve pro $N/5$ mediánů pětic, pak pro $\leq 7/10 \cdot N$ prvků před/za pivotem. Pro celkovou časovou složitost $t(N)$ našeho algoritmu tedy platí:

$$t(N) \leq cN + t(N/5) + t(7/10 \cdot N).$$

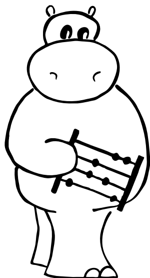
Nyní zbývá tuto rekurzivní nerovnici vyřešit, což provedeme drobným úskokem: uhodneme, že výsledkem bude lineární funkce, tedy že $t(N) = dN$ pro nějaké $d > 0$. Dostaneme:

$$dN \leq (c + 1/5 \cdot d + 7/10 \cdot d) \cdot N.$$

To platí např. pro $d = 10c$, takže opravdu $t(N) = \mathcal{O}(N)$.

Násobení dlouhých čísel

Dalším pěkným příkladem na rozdělování a panování je násobení dlouhých čísel – tak dlouhých, že se už nevejdou do integeru, takže s nimi musíme počítat po číslicích (ať už v jakékoliv soustavě – teď zvolíme desítkovou, často se hodí třeba 256-ková). Klasickým „školním" algoritmem pro násobení na papíře to zvládneme na kvadratický počet operací, zde si předvedeme efektivnější způsob.



Libovolné $2N$-ciferné číslo můžeme zapsat jako $10^N A + B$, kde A a B jsou N-ciferná. Součin dvou takových čísel pak bude $(10^N A + B) \cdot (10^N C + D) = (10^{2N} AC + 10^N (AD + BC) + BD)$. Sčítat dokážeme v lineárním čase, násobit mocninou deseti také (dopíšeme příslušný počet nul na konec čísla), N-ciferná čísla budeme násobit rekurzivním zavoláním téhož algoritmu. Pro časovou složitost tedy bude platit $t(N) = cN + 4t(N/2)$. Nyní tuto rovnici můžeme snadno vyřešit, ale ani to dělat nebudeme, neboť nám vyjde, že $t(N) \approx N^2$, čili jsme si oproti původnímu algoritmu vůbec nepomohli.

Přijde trik. Místo čtyř násobení čísel poloviční délky nám budou stačit jen tři: spočteme AC, BD a $(A+B) \cdot (C+D) = AC + AD + BC + BD$, přičemž pokud od posledního součinu odečteme AC a BD, dostaneme přesně $AD + BC$, které jsme předtím počítali dvěma násobeními. Časová složitost nyní bude $t(N) =$

$c'N + 3t(N/2)$. (Konstanta c' je o něco větší než c, protože přibylo sčítání a odčítání, ale stále je to konstanta. My si ovšem zvolíme jednotku času tak, aby bylo $c' = 1$, a ušetříme si tak spoustu psaní.)

Jak naši rovnici vyřešíme? Zkusíme ji dosadit do sebe samé a pozorovat, co se bude dít:

$$\begin{aligned} t(N) &= N + 3(N/2 + 3t(N/4)) = \\ &= N + 3/2 \cdot N + 9t(N/4) = \\ &= N + 3/2 \cdot N + 9/4 \cdot N + 27t(N/8) = \ldots = \\ &= N + 3/2 \cdot N + \ldots + 3^{k-1}/2^{k-1} \cdot N + 3^k t(N/2^k). \end{aligned}$$

Pokud zvolíme $k = \log_2 N$, vyjde $N/2^k = 1$, čili $t(N/2^k) = t(1) = d$, kde d je nějaká konstanta. To znamená, že:

$$t(N) = N \cdot \left[1 + 3/2 + 9/4 + \ldots + (3/2)^{k-1}\right] + 3^k d.$$

Výraz v hranaté závorce je součet prvních k členů *geometrické řady* s *kvocientem* (neboli podílem dvou po sobě jdoucích prvků) 3/2. Tuto geometrickou řadu si můžeme sečíst jako:

$$\frac{\left(\frac{3}{2}\right)^k - 1}{\frac{3}{2} - 1} = \mathcal{O}\left(\left(\frac{3}{2}\right)^k\right)$$

Tato funkce však roste pomaleji než zbylý člen $3^k d$, takže ji klidně můžeme zanedbat a zabývat se pouze oním posledním členem:

$$3^k = 2^{k \log_2 3} = 2^{\log_2 n \cdot \log_2 3} = (2^{\log_2 n})^{\log_2 3} = n^{\log_2 3} \approx n^{1.58}$$

Konstanta d se nám „schová do $\mathcal{O}$-čka“, takže algoritmus má časovou složitost přibližně $\mathcal{O}(n^{1.58})$. Existují i rychlejší algoritmy se složitostí až $\mathcal{O}(n \log n)$, ale ty jsou mnohem ďábelštější a pro malá n se to sotva vyplatí.

Program si pro dnešek odpustíme, šetřímeť naše lesy.

K zamyšlení

- Při hledání k-tého nejmenšího prvku jsme předpokládali, že všechny prvky jsou různé. Prohlédněte si algoritmy pozorně a rozmyslete si, že budou fungovat i bez toho. Opravdu?
- Proč jsme zvolili zrovna pětice? Jak by to dopadlo pro trojice? A jak pro sedmice? Fungoval by takový algoritmus? Byl by také lineární?
- Ve výpočtu $t(N)$ jsme si nedali pozor na neúplné pětice a také jsme předpokládali, že pětic je sudý počet. Ono se totiž nic zlého nemůže stát. Jak se to snadno nahlédne? Proč nestačí na začátku doplnit vstup „nekonečny“ na délku, která je mocninou deseti?

- Kdybychom neuhodli, že $t(N)$ je lineární, jak by se na to dalo přijít?
- Ještě jednou QS: Představte si, že budujete binární vyhledávací strom vkládáním prvků v náhodném pořadí. Obecně nemusí být vyvážený, ale v průměru v něm půjde vyhledávat v čase $\mathcal{O}(\log N)$. Žádný div: Stromy, které nám vzniknou, odpovídají přesně možným průběhům QuickSortu.

David Matoušek & Martin Mareš

Kuchařka čtvrté série – geometrie

Geometrické algoritmy

V dnešním díle našeho kuchařkového speciálu se budeme učit vařit geometrické problémy. A co že si představujeme pod pojmem geometrický problém? Trochu analytické geometrie, například zjištění, na které straně orientované přímky bod leží, trocha plotů, neboli konvexních obalů, a obecně mnoho zametání.

V celé kuchařce se omezíme pouze na dvourozměrné problémy, tedy na algoritmy v rovině. Některé postupy se dají zobecnit pro trojrozměrné, a většinou i pro n-rozměrné problémy, ale to je již nad rámec této kuchařky.

Geometrické základy

Nejdříve trocha středoškolské analytické geometrie pro ty, kdo ji ještě neměli. Ostatní mohou tuto sekci přeskočit.

Každý bod v rovině můžeme určit jeho souřadnicemi vůči osám. Nejběžněji se používá takzvaný *kartézský souřadný systém,* tedy dvě na sebe kolmé osy označované jako x-ová osa (vodorovná) a y-ová osa (svislá). Obvykle se uvažuje, že hodnoty na osách rostou směrem doprava (osa x) a směrem nahoru (osa y), my se toho budeme v naší kuchařce držet.

Místo, kde se obě osy protínají, se označuje jako *počátek* soustavy souřadnic. Samotné *souřadnice* bodu zapisujeme jako dvojici čísel, která udávají, o kolik jednotek se musíme posunout ve směru které z os, abychom z počátku dorazili do bodu, kterému souřadnice patří. Počátek má souřadnice $[0,0]$. Bod se souřadnicemi $[a,b]$ leží na pozici, kterou získáme tak, že se od počátku posuneme o a jednotek ve směru první osy (x-ové) a o b jednotek ve směru druhé osy (y-ové).

Vše ostatní funguje tak, jak jsme se učili při geometrii na základní škole, tedy úsečka je určena dvěma krajními body, obdélník čtyřmi a podobně. Ještě si ale řekneme, co je to vektor, a zavedeme některé další pojmy.

Často potřebujeme popsat vzájemnou polohu dvou bodů. Můžeme například udat jejich vzdálenost a směr (třeba jako úhel vzhledem k ose x). Praktičtější ale bývá říci, o kolik se liší jejich x-ové a y-ové souřadnice. To nám dá dvojici čísel, které říkáme *vektor.*

Pokud například k bodu $[1,1]$ přičteme vektor $a = (2,-1)$, dostaneme se do bodu $[3,0]$. Stejně tak, pokud odečteme například bod $[4,2]$ od bodu $[1,3]$, tak dostaneme vektor $b = (-3,1)$ udávající jejich vzájemnou polohu.

Pomocí vektoru a bodu tedy lze určit přímku. Bod nám určí, kam umístit vektor, a vektor nám určí směr přímky z daného bodu. Tomuto vektoru se říká *směrový vektor,* nebo také někdy *směrnice,* dané přímky nebo úsečky.

Samotné vyjádření přímky nebo úsečky poté může být ve dvou tvarech. Prvním z nich je *parametrický tvar.* Základem je nějaký bod $A = [a_x, a_y]$. Od toho se ve směru směrového vektoru $u = (u_x, u_y)$ můžeme pohybovat libovolně

a stále budeme na přímce. To nám vede na následující tvar, kde t je libovolný reálný parametr, neboli proměnná, za kterou si můžeme dosadit jakékoliv reálné číslo a vždy nám vyjde bod na přímce. Parametrický tvar vypadá takto:

$$x = a_x + tu_x$$

$$y = a_y + tu_y$$

To samé můžeme vyjádřit i vektorově, tedy $X = A + tu$.

Pro ilustrování funkce parametru, když bude $t = 0$, tak dostaneme výchozí bod přímky. Pokud poté budeme s parametrem hýbat od $-\infty$ do $+\infty$, dostaneme postupně všechny body na přímce.

Druhým způsobem zápisu je *obecný tvar přímky.* K jeho vyjádření budeme potřebovat kolmý vektor ke směrovému vektoru, tomu se také říká *normálový vektor.* V rovině ho získáme jednoduše. Pokud je $v = (v_x, v_y)$ směrnice přímky, tak vektor na něj kolmý má tvar $n = (v_y, -v_x)$. Jako poznámku pro zvídavé můžeme uvést, že *skalární součin* těchto vektorů, tedy součin po složkách ($v \cdot n = ab + b(-a)$), je roven 0, což je také jedna z definic kolmosti.

A jak tedy vypadá slibovaný obecný tvar přímky? Pokud je $n = (a, b)$ normálový vektor přímky, tak obecný tvar přímky je rovnice $ax + by + c = 0$. Dobře, a a b máme, jak ale zjistit c? Normálový vektor určuje směr, kterým přímka povede, ale stále ji můžeme libovolně posouvat. Potřebujeme ještě znát jeden bod, který na naší přímce leží, aby byla určená jednoznačně.

Když dosadíme souřadnice takového bodu do rovnice přímky s neznámou c, získáme tak rovnici pro c, kterou vyřešíme. A máme hotovo, známe hodnoty všech koeficientů v rovnici. Ještě si můžeme všimnout, že pro $c = 0$ prochází přímka počátkem.

Takovéto tvary se hodí nejen pro nějaké zapsání přímek, ale také pro *zjištění jejich průsečíku.* Když hledáme průsečík, hledáme vlastně místo, kde mají obě přímky navzájem stejné x-ové a y-ové souřadnice. A to vede na jednoduché soustavy lineárních rovnic, které jistě již vyřešit umíte.

Ještě si ale zdůrazněme rozdíl úseček oproti přímkám. V případě parametrického tvaru omezujeme velikost parametru t (například $t \in \langle 0, 1\rangle$) a v případě obecného tvaru omezujeme rozsah jedné ze souřadnic (například $x \in \langle -2, 2\rangle$). V případě, že bychom chtěli vyjádřit polopřímku, si parametr nebo souřadnici omezíme pouze z jedné strany.

Nakonec si ukážeme jednu základní aplikaci parametru a parametrického vyjádření úsečky. Jak snadno spočítat střed nějaké úsečky AB? V takovém případě není nic jednoduššího, než si vzít vektor $B - A$, přenásobit ho parametrem $1/2$ (střed úsečky je v polovině její délky) a přičíst k bodu A. Triviální úpravou pak zjistíme, že střed úsečky můžeme spočítat jako aritmetický průměr jejích krajních bodů:

$$A + \frac{1}{2} \cdot (B - A) = \frac{A + B}{2}$$

Jako příklad na rozkoukání si ukážeme, jak zjistit, na které straně přímky leží bod.

Zjištění polohy bodu vůči přímce

Nejdříve si zavedeme pojem orientovaná přímka. Když budeme mít přímku určenou dvojicí bodů A a B, budeme se na ni dívat, jako kdybychom stáli v prvním bodě (bod A) a dívali se směrem ke druhému (bod B). Pak již máme jasně definovanou pravou a levou stranu a můžeme říci, kde vůči přímce bod leží.

Vezměme si tedy přímku určenou body A a B a bod X. Určíme si vektory $u = X - A$ a $v = B - A$ (s prvky u_x, u_y, respektive v_x, v_y) a porovnáme úhel mezi nimi.

Pokud jste už měli analytickou geometrii, určitě znáte vzoreček na výpočet úhlu mezi dvěma vektory. Vzoreček má tvar:

$$\cos\alpha = \frac{u \cdot v}{|u||v|}$$

Jeho nevýhodou je, že výpočet inverzní funkce $\cos^{-1}$ trvá dlouho. Je proto lepší použít jiný způsob výpočtu, kde si vystačíme pouze s násobením.

Tím jiným způsobem je výpočet determinantu matice určené těmito vektory. *Matice* je pouze tabulka, kde jsou vektory poskládány pod sebe (ta naše tedy bude velká 2 na 2 políčka).

Determinant matice této velikosti nám udává obsah rovnoběžníku určeného zadanými vektory. A navíc znaménko determinantu nám říká, jestli je úhel mezi vektory (měřený v kladném směru, tedy proti směru hodinových ručiček) menší než π, nebo větší než π.

Kdo se ještě s determinanty nesetkal, může brát následující vzorec pro výpočet determinantu matice dva krát dva jako kouzelnou formuli. Kdo přesto chce zdůvodnění, může si zkusit udělat rozbor všech vzájemných poloh dvou přímek (a jejich směrových vektorů), které mohou nastat. Po chvíli dojdete ke vztahu přesně odpovídajícímu následujícímu vzorečku:

$$d = u_x v_y - u_y v_x.$$

Pokud vyjde d kladné, je bod napravo od přímky, pokud vyjde d záporné, je bod nalevo od přímky, a konečně, pokud vyjde $d = 0$, tak bod leží na přímce.

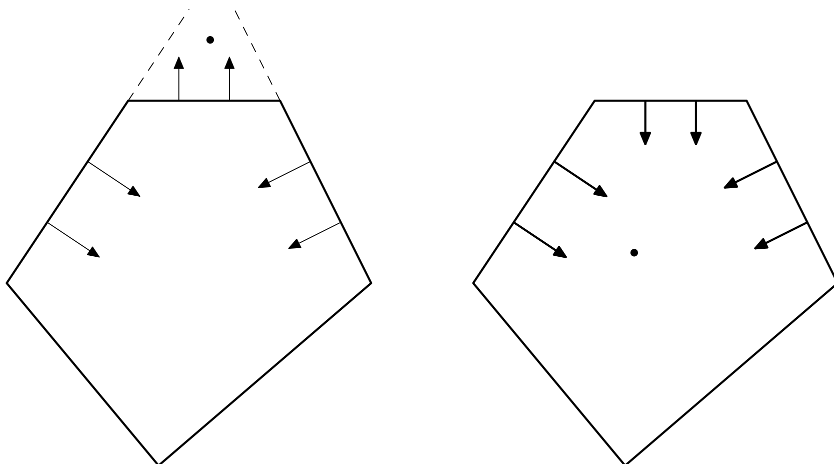
Bod a konvexní mnohoúhelník

Konvexní mnohoúhelník je takový, který nemá žádný vnitřní úhel větší než 180°. Jinou definicí je, že pokud si zvolíme libovolné dva body v mnohoúhelníku a natáhneme úsečku mezi nimi, nikdy nám nevyleze z mnohoúhelníku ven.

Když už víme, co konvexní mnohoúhelník je, jak zjistíme, jestli nějaký bod leží v něm, nebo ne? Využijeme vlastnosti konvexnosti. Stačí nám jít po hranách

na obvodu a zjišťovat, jestli hledaný bod leží na stejné straně všech hran (tedy přímek určených koncovými body hran), nebo neleží.

Pokud bod leží na stejné straně všech hran, nachází se uvnitř mnohoúhelníku. Pokud se ale vůči jen jediné hraně nachází na jiné straně než vůči ostatním, leží bod vně mnohoúhelníku. Nejlépe to vysvětlí obrázek:



Tomuto postupu se také někdy říká test polorovinami. Každá kontrola nám zabere konstantně mnoho času. Časová složitost tohoto postupu je tedy lineární vzhledem k počtu hran, neboli $\mathcal{O}(N)$.

Bod a nekonvexní mnohoúhelník

Pro nekonvexní útvary je již postup o něco těžší, protože jak si můžeme všimnout, postup s kontrolováním polohy bodu vůči všem hranám fungovat nebude.

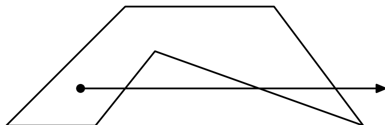
Můžeme si ale na chvíli zahrát na Robina Hooda a ze zkoumaného bodu vystřelit šíp, respektive vést polopřímku. Pěkně se nám bude počítat, pokud polopřímku povedeme rovnoběžně s nějakou z os (třeba ve směru $(1,0)$). Celé řešení pak spočívá v počítání, kolikrát polopřímka protne hranici mnohoúhelníku.

Můžeme si totiž všimnout, že finálně polopřímka skončí venku a nikdy více již do mnohoúhelníku nevstoupí. A pokaždé, když do mnohoúhelníku vstoupí, musí z něj zase někdy vystoupit. Pokud tedy bod leží venku, začali jsme polopřímku vést zvenku, a tedy bude počet protnutí sudý, pokud bod leží uvnitř, tak bude počet protnutí hranice lichý.

Jediné, na co je potřeba dát pozor, je situace, kdy polopřímka povede přesně skrz nějaký vrchol. V takovém případě se musíme podívat na opačné krajní body hran, které se v tomto vrcholu stýkají. Pokud se obě nachází ve stejné polorovině

určené polopřímkou, jen jsme se vrcholu dotkli, ale neprošli jsme skrz (a tedy nepočítáme žádný průsečík). Pokud se ale krajní body hran nachází v opačných polorovinách, znamená to, že jsme ve vrcholu hranici proťali a musíme započítat jeden průsečík.

Jako cvičení na rozmyšlenou necháme situaci, kdy se druhý krajní bod jedné z hran nachází na polopřímce.

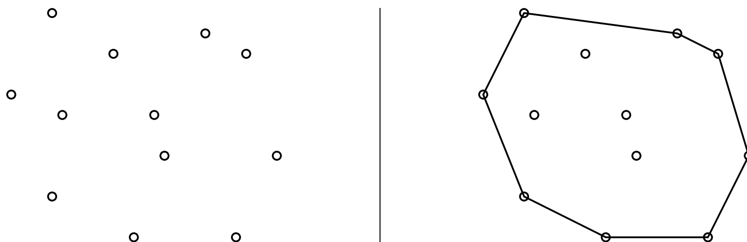


Opět musíme zkontrolovat polopřímku vůči všem hranám, takže časová složitost je znovu $\mathcal{O}(N)$ (i když s o něco vyšší konstantou, protože spočítání průsečíku je více početních operací než jeden test polorovinou).

Konvexní obal a zametání roviny

Podíváme se na jeden z nejznámějších geometrických problémů, totiž hledání konvexního obalu množiny bodů v rovině. *Konvexní obal* je nejmenší konvexní mnohoúhelník, který obsahuje všechny zadané body. Můžeme si všimnout, že všechny vrcholy výsledného mnohoúhelníka musí být nějaké body ze zadané množiny, jinak bychom mohli mnohoúhelník ještě zmenšit (a nebyl by to konvexní obal).

Jako motivaci si představte třeba situaci, že máte sad ovocných stromů a chcete je oplotit co nejkratším plotem. Jak takový plot, nebo obecně obal, nalézt?



Vlevo neobalené body, vpravo obalené.

Ukážeme si postup, kterému se říká *zametání roviny.* Je to trik, který najde uplatnění u mnoha různých geometrických problémů a vyplatí se ho umět.

Základní myšlenka spočívá v tom, že nějakou přímkou, říkejme jí *zametací přímka,* přejedeme přes celou rovinu (od minus nekonečna do plus nekonečna, zleva doprava nebo shora dolů) a vždy, když zametací přímka protne nějaký pro nás zajímavý bod, zpracujeme příslušnou událost. *Událost* je něco významného, co souvisí s příslušným bodem (průsečík přímek, vrchol mnohoúhelníka apod.)

Ale jak jet přímkou postupně od minus nekonečna do plus nekonečna? To není vůbec nutné. Pohyb přímky můžeme začít v nějakém startovním bodě (většinou první událost v setříděné posloupnosti událostí) a ukončit ho po zpracování všech událostí. Navíc nebudeme přímkou pohybovat plynule, ale budeme jí vždy skákat z události na událost (protože mezi událostmi se nic zajímavého neděje).

Vraťme se k našemu problému s konvexním obalem. Jako události budeme brát všechny body, které dostaneme na vstupu. V tomto případě nám žádné nové události v průběhu výpočtu vznikat nebudou, takže frontu událostí můžeme implementovat jako lineární spojový seznam.

Na začátku si body setřídíme podle jejich x-ové souřadnice (zatím budeme pro jednoduchost předpokládat, že žádné dva body nemají stejnou x-ovou souřadnici), začneme je zametací přímkou postupně procházet zleva doprava a budeme si udržovat konvexní obal bodů, které jsme už zpracovali.

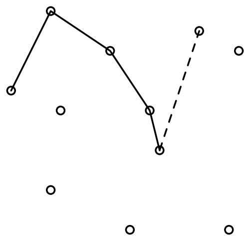
V průběhu výpočtu si budeme konstruovat horní a dolní *obálku*. Obě obálky budou určitě začínat v nejlevějším a končit v nejpravějším bodě (jednoduchým pozorováním lze nahlédnout, že tyto body do obalu určitě patří). A jak už název napovídá, horní obálka půjde vrchem a bude se zatáčet stále doprava, a dolní obálka naopak půjde spodem a bude se stále zatáčet doleva.

Můžeme se pro zjednodušení dohodnout, že nejlevější i nejpravější bod patří do obou obálek. Když pak horní a dolní obálku spojíme, dostaneme konvexní obal.

Horní (respektive dolní) obálku si budeme udržovat jako lineární seznam vrcholů.

Teď si ukážeme, jak bude probíhat jeden krok zpracování. Výpočet se bude provádět samostatně pro horní a dolní obálku, my si ho ukážeme jen pro horní (pro dolní je až na zrcadlení stejný).

Uvažujme, že už máme nějakou část horní obálky, skočili jsme zametací přímkou na další bod a ten teď chceme přidat. Podíváme se na poslední bod v horní obálce a zkontrolujeme úhel poslední hrany v obálce a úsečky mezi posledním bodem obalu a novým bodem.



K tomu můžeme využít například test polorovinami z úvodu kuchařky (pokud nový bod leží vůči poslední hraně horní obálky napravo, je vnitřní úhel konvexní, pokud nalevo, je úhel konkávní). Jestliže se horní obálka zatáčí doprava, máme vyhráno, přidáme nový bod do obálky a můžeme se posunout na další bod. Zajímavější je ale situace, kdy se nám obálka stočí doleva a vznikne konkávní úhel.

Pokud se podíváme na obrázek výše, jasně vidíme, že je potřeba dosavadní poslední bod obálky odebrat a zkusit spojit nově přidávaný bod s předposledním. Odstraníme tedy poslední bod obálky a budeme test opakovat s předposledním bodem.

Takto budeme pokračovat (a případně vyhazovat další body), buď než bude úhel hran konvexní, nebo dokud nám v obálce nezůstane pouze jeden bod (počáteční). Pak nový bod přidáme do obálky a pokračujeme s dalším.

Výše popsaný postup je nejvýhodnější provádět najednou pro obě dvě obálky. Tedy každý bod se pokusíme připojit k horní i dolní obálce a podle toho obě obálky příslušně upravíme.

Proč tento postup funguje? Postupně projdeme všechny body a každý z nich se alespoň na chvíli stane posledním bodem obálky. Při změně obálky se obsažená plocha v konvexním obalu vždy pouze zvětší a žádný bod nám tedy nemůže zůstat mimo konvexní obal.

Ještě jsme zapomněli na případ, kdy úhel není ani konvexní, ani konkávní. V takovém případě se rozhodneme, jestli budeme vrchol tohoto úhlu započítávat mezi vrcholy konvexního obalu. Obvykle se takový vrchol z konvexního obalu vyhazuje, ale nakonec vždycky záleží, k čemu ten konvexní obal vlastně potřebujeme.

Skončíme, až zametací přímkou skočíme na poslední bod a zpracujeme ho. V tomto bodě se nám obálky spojí a dostaneme celý konvexní obal. Teď ale přichází otázka, kolik času nám tento postup zabere?

Může se zdát, že hodně, protože při vyhazování bodů z obálky můžeme postupně vyhodit skoro všechny body. Označme si velikost zadané množiny (počet bodů na vstupu programu) N. Musíme si uvědomit, že každý bod do obálky přidáme pouze jednou a vyhodíme ho také maximálně jednou, tedy časová složitost je lineární k velikosti množiny, čili $\mathcal{O}(N)$, v případě, že již máme setříděný vstup. Pokud ne, musíme ještě přičíst čas potřebný k setřídění bodů, tedy $\mathcal{O}(N \log N)$ při použití nějakého rychlého třídicího algoritmu.[38]

Nakonec ještě zbývá dořešit více bodů se stejnou x-ovou souřadnicí. Pokud to nejsou krajní body, tak nám to v postupu nevadí. Menším problémem je, když to jsou počáteční, nebo koncové body. Problém ale snadno vyřešíme tím, když body seřadíme lexikograficky, tedy nejdříve podle x, a pokud je stejné, pak podle y. To nám jednoznačně určí pořadí bodů a počáteční i koncový bod.

Také si to můžeme představit tak, že rovinu nepatrně natočíme. Tím se určitě konvexní obal (až na natočení) nezmění, nikde nebudou dva body nad sebou a z pohledu algoritmu je to vlastně totéž, jako bychom prošli body v lexikografickém pořadí.

Hledání průsečíků úseček

Nakonec si ukážeme ještě jeden typický zametací problém, který principu zametání využívá o trochu více než konvexní obal. Představte si, že máte v rovině N úseček a chcete najít všechny jejich průsečíky.

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

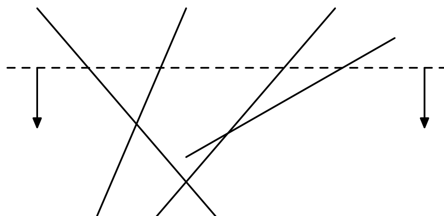
Hledáme samozřejmě co nejrychlejší algoritmus vzhledem k N a počtu průsečíků P.

Bystří si již jistě spočítali, že průsečíků může být v extrémním případě až N^2, a tedy nic rychlejšího než zkontrolovat každou úsečku se všemi dalšími v tomto případě není.

Ale takové případy se moc často nestávají, spíše naopak. Uvažujme tedy, že průsečíků je řádově tolik, kolik je úseček a v tom případě je výše popsaný algoritmus již pomalý.

Předpokládejme pro zjednodušení, že v žádném bodě se neprotínají tři a více úseček, žádné dvě úsečky nemají více než jeden společný bod (neleží přes sebe) a žádná úsečka není ani přesně svislá, ani přesně vodorovná. Vyřešení takovýchto případů spočívá v snadných úpravách uvedeného řešení.

Použijeme opět zametací přímku (pro lepší představu teď jdoucí shora dolů, obecně ale nemá směr zametání význam), kterou budeme skákat přes události, a na ní si budeme udržovat aktuální stav. Nazvěme ji třeba *průřezem.* Jak už název napovídá, bude udržovat pořadí úseček, které aktuálně protínají zametací přímku. Jelikož se průřez bude po každé události měnit, budeme pro něj potřebovat šikovnou datovou strukturu. Ale na to se podíváme až potom, co si rozebereme události, ať víme, co od průřezu budeme chtít.



Stejně jako v minulém případě budou mezi událostmi všechny body na vstupu (tedy počáteční i koncové body úseček), vyskytnou se tam ale i další. Pojďme si tedy trochu lépe rozebrat události a akce, které se při nich mají stát:

- *Začátek úsečky:* Přidáme úsečku na správné místo do průřezu, spočítáme případné průsečíky s okolními úsečkami a přidáme je do seznamu událostí.
- *Konec úsečky:* Smažeme úsečku z průřezu, a jelikož se nám dvě okolní úsečky dostanou smazáním této k sobě, musíme ještě spočítat jejich případný průsečík a přidat ho do seznamu událostí.
- *Průsečík:* Započítáme a zapíšeme si průsečík úseček, prohodíme pořadí těchto dvou úseček na průřezu, a jelikož se nám k sobě na průřezu dostaly nové úsečky, musíme spočítat, jestli se někde protínají, a případně průsečíky přidat do seznamu událostí.

Spočítání průsečíků úseček je jednoduchá analytická geometrie. Nejdříve porovnáme jejich směrnice. Pokud jdou od sebe, nemusíme se o nic starat, pokud

jdou k sobě, spočítáme, ve kterém bodě se protnou. A když máme tento bod, jenom ověříme, jestli leží na obou úsečkách (neboli že úsečky nekončí ještě před spočítaným průsečíkem).

Když se podíváme na požadavky, hodilo by se nám umět v průřezu rychle vyhledávat, přidávat a mazat, k čemuž nám nejlépe poslouží vyhledávací strom. Ale co za informace si budeme o úsečkách ve vrcholech stromu pamatovat? Jejich aktuální x-ovou pozici (tedy přesněji x-ovou souřadnici bodu této úsečky na úrovni zametací přímky)? Tu bychom museli po každé události u všech úseček přepočítat, budeme na to tedy muset jít chytřeji.

Ve vrcholech stromu si budeme ukládat pouze nějaký rovnicový tvar úsečky (například její obecnou rovnici, nebo směrnici a bod) a vždy, když budeme vyhledávat ve stromu, tak si na základě aktuální y-ové pozice zametací přímky spočítáme v konstantním čase aktuální x-ovou pozici úsečky (jednoduchým doplněním do obecné rovnice) a podle toho se budeme ve vyhledávacím stromu pohybovat.

Máme tedy datovou strukturu pro průřez, ale jak dlouho budou trvat operace s ní? Jelikož v každou chvíli bude ve vyhledávacím stromu maximálně N vrcholů (tedy maximálně tolik, kolik je úseček), budou všechny operace se stromem trvat $\mathcal{O}(\log N)$.

Do seznamu událostí budeme potřebovat také přidávat prvky, takže tentokrát se nám mnohem více hodí použití nějaké haldy. Opět si můžeme uvědomit, že v haldě bude najednou pouze $\mathcal{O}(N)$ prvků (za každou úsečku její začátek a konec a průsečíky úseček vedle sebe na průřezu, tedy maximálně $N-1$ průsečíků), takže operace v ní bude trvat $\mathcal{O}(\log N)$.

Když už máme vybudované datové struktury, podívejme se na to, jak algoritmus poběží. Na začátku přidáme do průřezu první úsečku a do seznamu událostí všechny začátky i konce úseček. Pak již jen postupujeme po událostech, každou událost zpracujeme podle postupu výše a skončíme ve chvíli, kdy nám dojdou všechny události.

Algoritmus funguje správně, jelikož postupně projde přes všechny průsečíky (když jedna úsečka protíná více dalších, tak postupným prohazováním v průřezu se dostanou všechny tyto dvojice vedle sebe a všechny průsečíky přidáme do událostí) a žádný průsečík neprojdeme vícekrát.

Zpracování jakékoliv události nás stojí konstantní množství operací s datovými strukturami, a protože každá z těchto operací stojí maximálně $\mathcal{O}(\log N)$, tak nás zpracování jedné události stojí $\mathcal{O}(\log N)$. Počet událostí je $2N+P$ kde N je počet úseček a P počet průsečíků na výstupu, tedy celková časová složitost je $\mathcal{O}((N+P)\log N)$. Pro pořádek ještě uveďme paměťovou složitost, které je díky použitým datovým strukturám $\mathcal{O}(N)$.

Můžeme si všimnout, že pokud by průsečíků bylo řádově N^2, tak jsme si vlastně pohoršili. Předpokládali jsme ale situaci, kdy je průsečíků řádově stejně

jako úseček. V tomto případě je náš algoritmus výrazně rychlejší.

Závěr

Prošli jsme si základní geometrické algoritmy pro rovinnné problémy a ukázali jejich základní myšlenky. Různou aplikací a kombinací těchto postupů můžeme řešit většinu lehčích geometrických problémů v rovině, které potkáme.

Jen jako ochutnávku si ještě uvedeme například *Voroného diagramy,* což je rozklad roviny na oblasti, které jsou vždy nejblíž danému bodu (motivací může být například přiřazení obcí na mapě k nejbližšímu krajskému městu). Při jejich konstrukci se také uplatní zametání roviny, ovšem tentokrát již ne přímkou, ale pomocí zametacích parabol.

A jak jsme si uvedli na začátku, mnohé z uvedených postupů lze zobecnit z roviny i do prostoru, ale o tom někdy jindy. Pokud máte zájem o další informace o geometrických algoritmech, tak vás můžeme odkázat na studijní text k přednášce ADS[39] na stránkách Martina Mareše.

Pokud stále nemáte geometrie dost, můžete si ještě zkusit vyhledat pojmy *kombinatorická a výpočetní geometrie.* Dostanete se tak ke spoustě dalších zajímavých materiálů.

Jirka Setnička

[39] `http://mj.ucw.cz/vyuka/ads/43-geom.pdf`

Kuchařka páté série – dynamické programování

Rekurzivní funkce je taková funkce, která při svém běhu volá sama sebe, často i více než jednou. To typicky vede na exponenciální časovou složitost algoritmu.

Dynamické programování je technika, kterou lze z pomalého rekurzivního algoritmu vyrobit pěkný polynomiální, tedy až na výjimečné případy. A jako ukázku si představíme algoritmus, který nalezne délky nejkratších cest mezi každými dvěma městy na mapě.

Ale nepředbíhejme, nejdříve se podíváme na jednoduchý příklad rekurze.

Fibonacciho čísla

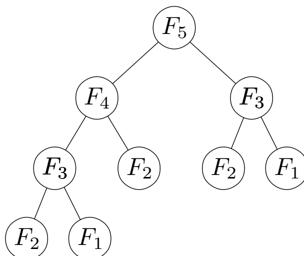
Budeme počítat n-té číslo Fibonacciho posloupnosti. To je posloupnost, jejíž nultý člen je nula, první je jednička ($F_0 = 0$, $F_1 = 1$) a každý další člen je součtem dvou předchozích ($F_n = F_{n-1} + F_{n-2}$ pro $n > 1$). Začíná takto:

$$0 \quad 1 \quad 1 \quad 2 \quad 3 \quad 5 \quad 8 \quad 13 \quad 21 \quad 34 \quad 55 \quad 89 \quad \ldots$$

Pro nalezení n-tého členu (v našem značení F_n) si napíšeme rekurzivní funkci `fib(n)`, která bude postupovat přesně podle definice – zeptá se sama sebe rekurzivně, jaká jsou dvě předchozí čísla, a pak je sečte. Možná více řekne program:

```
def fib(n):
    if n == 0:
        return 0
    elif n == 1:
        return 1
    else:
        return fib(n-1) + fib(n-2)
```

To, jak funkce volá sama sebe, si můžeme snadno nakreslit třeba pro výpočet čísla F_5:



Vidíme, že se program rozvětvuje, což tvoří strom volání. V každém vrcholu tohoto stromu trávíme konstantní čas, takže časová složitost celého algoritmu je až na konstantu rovna počtu vrcholů tohoto stromu. Kolik to je, spočítáme jednoduchou úvahou.

Každý vrchol stromu vrací hodnotu, která je součtem hodnot v jeho synech. Proto je hodnota v kořeni rovna součtu hodnot v listech. V listech jsou ovšem jedničky (F_1 a F_2), takže listů musí být právě F_n a všech vrcholů dohromady aspoň F_n.

Proto na spočítání n-tého Fibonacciho čísla spotřebujeme čas alespoň takový, kolik je ono číslo samo. Ale jak velké takové F_n vlastně je? Můžeme třeba využít toho, že

$$F_n = F_{n-1} + F_{n-2} \geq 2 \cdot F_{n-2},$$

z čehož indukcí dokážeme

$$F_n \geq 2^{n/2} \quad \text{pro } n \geq 6.$$

Funkce `Fibonacci` má tedy alespoň exponenciální časovou složitost, což není nic vítaného.

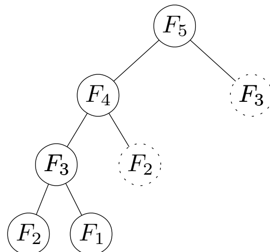
Jak najít efektivnější algoritmus? Všimneme si, že některé podstromy jsou shodné. Zřejmě to budou ty části, které reprezentují výpočet stejného Fibonacciho čísla – v našem příkladě třeba třetího. Tyto výpočty opakujeme stále dokola.

Nenabízí se proto nic snazšího, než si jejich výsledky uložit a pak je kdykoliv vytáhnout jako pověstného králíka z klobouku[40] s minimem námahy.

Bude nám k tomu stačit jednoduché pole P o n prvcích, které na počátku inicializujeme hodnotami značícími nespočítané hodnoty. Kdykoliv budeme chtít spočítat některý člen, nejdříve se podíváme do pole, zda jsme ho již jednou nespočetli. A naopak jakmile hodnotu spočítáme, hned si ji do pole poznamenáme:

```
P = [None] * (MaxN + 1)
P[0] = 0; P[1] = 1
def fibonacci(n):
    if P[n] is None:
        P[n] = fibonacci(n-1) + fibonacci(n-2)
    return P[n]
```

Podívejme se, jak vypadá strom volání nyní:



---

[40] Právě zde je zmínka o králících příhodná. Legenda o Fibonacciho číslech vypráví, že k jejich objevu došlo při výzkumu rozmnožování králíků, kdy první dva měsíce měl 1 pár, další měsíc měl 2 páry, pak 3, pak 5, …

Na každý člen posloupnosti se tentokrát ptáme maximálně dvakrát – k výpočtu ho potřebují dva následující členy. To ale znamená, že funkci `Fibonacci` zavoláme maximálně $2n$-krát, čili jsme touto jednoduchou úpravou zlepšili exponenciální složitost na lineární.

Zdálo by se, že abychom získali čas, museli jsme obětovat paměť, ale to není tak úplně pravda. V prvním příkladu sice nepoužíváme žádné pole, ale při volání funkce si musíme zapamatovat některé údaje, jako je třeba návratová adresa, parametry funkce a její lokální proměnné, a na to samotné potřebujeme určitě paměť lineární s hloubkou vnoření, v našem případě tedy lineární s n.

Určitě vás už také napadlo, že n-té Fibonacciho číslo se dá snadno spočítat i bez rekurze. Stačí si prvky posloupnosti počítat postupně od začátku – kdykoliv známe $F_{1...k}$ (všechny prvky posloupnosti až do indexu k), dokážeme snadno spočítat i F_{k+1}:

```
def fibonacci2(n):
    if n == 0:
        return 0
    a = 0; b = 1
    while n > 1:
        (a, b) = (b, a+b)
        n -= 1
    return b
```


Zopakujme si, co jsme postupně udělali – nejprve jsme vymysleli pomalou rekurzivní funkci, kterou jsme zrychlili zapamatováváním si mezivýsledků. Nakonec jsme ale celou rekurzi „obrátili naruby“ a mezivýsledky počítali od nejmenšího k největšímu, aniž bychom se starali o to, jak se na ně původní rekurze ptala.

V případě Fibonacciho čísel je samozřejmě snadné přijít rovnou na nerekurzivní řešení, a díky pamatování si jen posledních dvou hodnot snížit i paměťovou složitost na konstantní.

Zmíněný obecný postup zrychlování rekurze nebo rovnou řešení úlohy od nejmenších podproblémů k těm největším funguje i pro řadu složitějších úloh a říkáme mu technika *dynamického programování*. Ukážeme si další problém řešitelný touto technikou.

Problém batohu

Je dáno N předmětů o hmotnostech $m_1, \ldots, m_N$ (celočíselných) a také číslo M (nosnost batohu). Úkolem je vybrat některé z předmětů tak, aby součet jejich hmotností byl co největší, ale přitom nepřekročil M. Předvedeme si algoritmus, který tento problém řeší v čase $\mathcal{O}(MN)$.

Náš algoritmus bude používat pomocné pole $A[0 \ldots M]$ a jeho činnost bude rozdělena do N kroků. Na konci k-tého kroku bude prvek $A[i]$ nenulový právě tehdy, jestliže z prvních k předmětů lze vybrat předměty, jejichž součet hmotností je přesně i.

Před prvním krokem (po nultém kroku) jsou všechny hodnoty $A[i]$ pro $i > 0$ nulové a $A[0]$ má nějakou nenulovou hodnotu, řekněme -1.

Všimněme si, jak kroky algoritmu odpovídají podúlohám, které řešíme – v prvním kroku vyřešíme podúlohu tvořenou jen prvním předmětem, ve druhém kroku prvními dvěma předměty, pak prvními třemi předměty atd.

Popišme si nyní k-tý krok algoritmu. Pole A budeme procházet od konce, tj. od $i = M$. Pokud je hodnota $A[i]$ stále nulová, ale hodnota $A[i-m_k]$ je nenulová, změníme hodnotu uloženou v $A[i]$ na k (později si vysvětlíme, proč zrovna na k).

Nyní si rozmyslíme, že po provedení k-tého kroku odpovídají nenulové hodnoty v poli A hmotnostem podmnožin z prvních k předmětů (*podmnožina* je v podstatě jen výběr nějaké části předmětů).

Pokud je hodnota $A[i]$ nenulová, pak buď byla nenulová před k-tým krokem (a v tom případě odpovídá hmotnosti nějaké podmnožiny prvních $k-1$ předmětů), anebo se stala nenulovou v k-tém kroku.

Potom ale hodnota $A[i-m_k]$ byla před k-tým krokem nenulová, a tedy existuje podmnožina prvních $k-1$ předmětů, jejíž hmotnost je $i-m_k$. Přidáním k-tého předmětu k této podmnožině vytvoříme podmnožinu předmětů hmotnosti přesně i.

Naopak, pokud lze vytvořit podmnožinu X hmotnosti i z prvních k předmětů, pak takovou podmnožinu X lze buď vytvořit jen z prvních $k-1$ předmětů, a tedy hodnota $A[i]$ je nenulová již před k-tým krokem, anebo k-tý předmět je obsažen v takové množině X.

Potom ale hodnota $A[i-m_k]$ je nenulová před k-tým krokem (hmotnost podmnožiny X bez k-tého prvku je $i-m_k$) a hodnota $A[i]$ se stane nenulovou v k-tém kroku.

Po provedení všech N kroků odpovídají nenulové hodnoty $A[i]$ přesně hmotnostem podmnožin ze všech předmětů, které máme k dispozici. Speciálně největší index i_0 takový, že hodnota $A[i_0]$ je nenulová, odpovídá hmotnosti nejtěžší podmnožiny předmětů, která nepřekročí hmotnost M.

Nalézt jednu množinu této hmotnosti také není obtížné: V k-tém kroku jsme měnili nulové hodnoty v poli A na hodnotu k, takže v $A[i_0]$ je uloženo číslo jednoho z předmětů nějaké takové množiny, v $A[i_0 - m_{A[i_0]}]$ číslo dalšího předmětu atd. Zdrojový kód tohoto algoritmu lze nalézt na další straně.

Časová složitost algoritmu je $\mathcal{O}(NM)$, neboť se skládá z N kroků, z nichž každý vyžaduje čas $\mathcal{O}(M)$. Paměťová složitost činí $\mathcal{O}(N+M)$, což představuje paměť potřebnou pro uložení pomocného pole A a hmotností daných předmětů.

Cvičení a poznámky

- Proč pole A procházíme pozadu a ne popředu?
- Složitost algoritmu vypadá jako polynomiální, ale to je trochu podvod. Závisí totiž na hodnotě M. Pokud tuto hodnotu na vstupu zapíšeme obvyklým

způsobem, tedy v desítkové nebo dvojkové soustavě, použijeme řádově $\log M$ cifer.

Naše M proto bude vzhledem k délce vstupu až exponenciálně velké. To je typický příklad takzvaného *pseudopolynomiálního* algoritmu – tedy takového, jenž je vzhledem k hodnotám na vstupu polynomiální, ale k délce vstupu exponenciální. Podrobnosti si můžete přečíst v kuchařce o těžkých úlohách.[41]

```
# Již existující proměnné:
# N - počet předmětů
# M - hmotnostní omezení
# hmotnosti - pole hmotností dílčích předmětů

A = [0] * M

A[0] = 1
for k in range(N):
  for i in range(M, hmotnost[k]-1, -1):
    if (A[i-hmotnost[k]] != 0) and (A[i] == 0):
      A[i] = k
  i = M
  while A[i] == 0:
    i -= 1
  print("Maximální hmotnost: {}" .format(i))
  print("Předměty v množině:", end="")
  while A[i] != -1:
    print(" {}" .format(A[i]), end="")
    i = i - hmotnost[A[i]]
```


Nejkratší cesty a Floydův-Warshallův algoritmus

Náš další příklad bude z oblasti grafových algoritmů, ale zkusíme si jej nejdříve říci bez grafů:

Bylo-nebylo-je N měst. Mezi některými dvojicemi měst vedou (obousměrné) *silnice,* jejichž délky jsou dány na vstupu. Předpokládáme, že silnice se jinde než ve městech nepotkávají (pokud se kříží, tak mimoúrovňově).

Úkolem je spočítat nejkratší vzdálenosti mezi všemi dvojicemi měst, tj. délky nejkratších cest mezi všemi dvojicemi měst. *Cestou* rozumíme posloupnost měst takovou, že každá dvě po sobě následující města jsou spojené silnicí, a délka cesty je součet délek silnic, které tato města spojují.

V grafové terminologii tedy máme daný ohodnocený neorientovaný graf a chceme zjistit délky nejkratších cest mezi všemi dvojicemi jeho vrcholů.

Půjdeme na to následovně – vzdálenosti mezi městy jsou na začátku algoritmu uloženy ve dvourozměrném poli D, tj. $D[i][j]$ je vzdálenost z města i

[41] http://ksp.mff.cuni.cz/viz/kucharky/tezke-problemy

do města j. Pokud mezi městy i a j nevede žádná silnice, bude $D[i][j] = \infty$ (v programu bude tato hodnota rovna nějakému dostatečně velkému číslu).

V průběhu výpočtu si budeme na pozici $D[i][j]$ udržovat délku nejkratší dosud nalezené cesty mezi městy i a j.

Algoritmus se skládá z N fází. Na konci k-té fáze bude v $D[i][j]$ uložena délka nejkratší cesty mezi městy i a j, která může procházet skrz libovolná z měst $1, \ldots, k$.

V průběhu k-té fáze tedy stačí vyzkoušet, zda je mezi městy i a j kratší stávající cesta přes města $1, \ldots, k-1$, jejíž délka je uložena v $D[i][j]$, nebo nová cesta přes město k.

Pokud nejkratší cesta prochází přes město k, můžeme si ji rozdělit na nejkratší cestu z i do k a nejkratší cestu z k do j. Délka takové cesty je tedy rovna $D[i][k] + D[k][j]$.

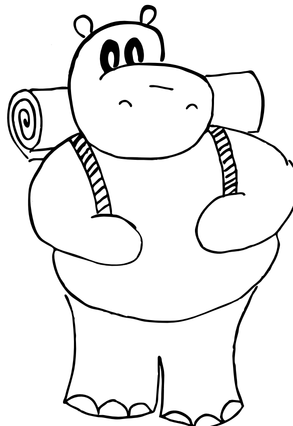
Pokud je součet $D[i][k] + D[k][j]$ menší než stávající hodnota $D[i][j]$, nahradíme hodnotu na pozici $D[i][j]$ tímto součtem, jinak ji ponecháme.

Z popisu algoritmu přímo plyne, že po N-té fázi je na pozici $D[i][j]$ uložena délka nejkratší cesty z města i do města j.

Protože v každé z N fází algoritmu musíme vyzkoušet všechny dvojice i a j, vyžaduje každá fáze čas $\mathcal{O}(N^2)$. Celková časová složitost našeho algoritmu tedy je $\mathcal{O}(N^3)$. Co se paměti týče, vystačíme si s polem D a to má velikost $\mathcal{O}(N^2)$.

Program bude vypadat následovně:

```
for k in range(N):
  for i in range(N):
    for j in range(N):
      if d[i][k] + d[k][j] < d[i][j]:
        d[i][j] = d[i][k] + d[k][j]
```



Popišme si ještě, jak bychom postupovali, kdybychom kromě vzdáleností mezi městy chtěli nalézt i nejkratší cesty mezi nimi.

To lze jednoduše vyřešit například tak, že si navíc budeme udržovat pomocné pole $E[i][j]$ a do něj při změně hodnoty $D[i][j]$ uložíme nejvyšší číslo města na cestě z i do j délky $D[i][j]$ (při změně v k-té fázi je to číslo k).

Máme-li pak vypsat nejkratší cestu z i do j, vypíšeme nejprve cestu z i do $E[i][j]$ a pak cestu z $E[i][j]$ do j. Tyto cesty nalezneme stejným (rekurzivním) postupem.

Poznámky

- Popis algoritmu vysloveně svádí k záludné otázce: Jak víme, že spojením dvou cest, které provádíme, vznikne zase cesta (tj. že se na ní nemohou nějaké vrcholy opakovat)?

  To samozřejmě nevíme, ale všimněme si, že kdykoliv by to cesta nebyla, tak si ji nevybereme, protože původní cesta bez vrcholu k bude vždy kratší nebo alespoň stejně dlouhá…

  Tedy dokud se v naší zemi nevyskytuje cyklus záporné délky. To bychom měli přidat do předpokladů našeho algoritmu, kdybychom byli pedanti (ale hrany záporné délky stále dovolujeme, jen nesmí utvořit cyklus se záporným součtem).
- Pozor na pořadí cyklů – program vysloveně svádí k tomu, abychom psali cyklus pro `k` jako vnitřní… jenže pak samozřejmě nebude fungovat.

Cvičení

- Jak by algoritmus fungoval, kdyby silnice byly jednosměrné?
- Hodnoty v poli si přepisujeme pod rukama, takže by se nám mohly poplést hodnoty z předchozí fáze s těmi z fáze současné. Ale zachrání nás to, že čísla, o která jde, vyjdou v obou fázích stejně. Proč?

Nejdelší společná podposloupnost

Poslední příklad dynamického programování, který si předvedeme, se bude týkat posloupností. Mějme dvě posloupnosti čísel A a B. Chceme najít jejich *nejdelší společnou podposloupnost* (NSP), tedy takovou posloupnost, kterou můžeme získat z A i B odstraněním některých prvků. Například pro posloupnosti

$$A = 2\ 3\ 3\ 1\ 2\ 3\ 2\ 2\ 3\ 1\ 1\ 2$$
$$B = 3\ 2\ 2\ 1\ 3\ 1\ 2\ 2\ 3\ 3\ 1\ 2\ 2\ 3$$

je jednou z nejdelších společných podposloupností tato posloupnost:

$$C = 2\ 3\ 1\ 2\ 2\ 3\ 1\ 2.$$

Jakým způsobem můžeme takovou podposloupnost najít? Nejdříve nás asi napadne vygenerovat všechny podposloupnosti a ty pak porovnat.

Jakmile si ale spočítáme, že všech podposloupností posloupnosti o délce n je 2^n (každý prvek nezávisle na ostatních buď použijeme, nebo ne), najdeme raději nějaké rychlejší řešení.

Zkusme využít následující myšlenku: vyřešíme tento problém pouze pro první prvek posloupnosti A. Pak najdeme řešení pro první dva prvky A, přičemž využijeme předchozích výsledků. Takto pokračujeme pro první tři, čtyři, … až n prvků.

Nejprve si rozmyslíme, co všechno si musíme v každém kroku pamatovat, abychom z toho dokázali spočíst krok následující. Určitě nám nebude stačit pamatovat si pouze nejdelší podposloupnost, jenže množina všech společných podposloupností je už zase moc velká.

Podívejme se tedy detailněji, jak se změní tato množina při přidání dalšího prvku k A: Všechny podposloupnosti, které v množině byly, tam zůstanou a navíc přibyde několik nových, končících právě přidaným prvkem.

Ovšem my si podposloupnosti pamatujeme proto, abychom je časem rozšířili na nejdelší společnou podposloupnost.

Takže pokud známe nějaké dvě stejně dlouhé podposloupnosti P a Q končící nově přidaným prvkem v A a víme, že P končí v B dříve než Q, stačí si z nich pamatovat pouze P.

V libovolném rozšíření Q-čka totiž můžeme Q vyměnit za P a získat tím stejně dlouhou společnou podposloupnost.

Proto si stačí pro již zpracovaných a prvků posloupnosti A pamatovat pro každou délku l tu ze společných podposloupností $A[1 \ldots a]$ a B délky l, která v B končí na nejlevějším možném místě. Dokonce nám bude stačit si místo celé podposloupnosti uložit jen pozici jejího konce v B. K tomu použijeme dvojrozměrné pole $D[a, l]$.

Ještě si dovolíme jedno malé pozorování: Koncové pozice uložené v poli D se zvětšují s rostoucí délkou podposloupnosti, čili $D[a, l] < D[a, l + 1]$, protože posloupnosti délky $l + 1$ nejsou ničím jiným než rozšířeními posloupností délky l o 1 prvek.

Teď již výpočet samotný. Pokud už známe celý a-tý řádek pole D, můžeme z něj získat $(a + 1)$-ní řádek. Projdeme postupně posloupnost B. Když najdeme v B prvek $A[a + 1]$ (ten právě přidávaný do A), můžeme rozšířit všechny podposloupnosti končící před aktuální pozicí v B.

Nás bude zajímat pouze ta nejdelší z nich, protože rozšířením všech kratších získáme posloupnost, jejíž koncová pozice je větší než koncová pozice některé posloupnosti, kterou již známe. Rozšíříme tedy tu nejdelší podposloupnost a uložíme ji místo původní podposloupnosti.

Toto provedeme pro každý výskyt nového prvku v posloupnosti B. Všimněme si, že nemusíme procházet pole s podposloupnostmi stále od začátku, ale můžeme se v něm posouvat od nejmenší délky k největší.

Ukážeme si, jak vypadá zaplněné pole hodnotami při řešení problému z našeho příkladu. Abychom nemuseli listovat, tak si zde zadání příkladu uvedeme ještě jednou – hledáme NSP těchto dvou posloupností:

$$A = 2\ 3\ 3\ 1\ 2\ 3\ 2\ 2\ 3\ 1\ 1\ 2$$
$$B = 3\ 2\ 2\ 1\ 3\ 1\ 2\ 2\ 3\ 3\ 1\ 2\ 2\ 3$$

Nyní už slíbená tabulka znázorňující výpočet. Řádky jsou pozice v A, sloupce délky podposloupností.

D	1	2	3	4	5	6	7	8	9	10	11	12
1	2	–	–	–	–	–	–	–	–	–	–	–
2	1	5	–	–	–	–	–	–	–	–	–	–
3	1	5	9	–	–	–	–	–	–	–	–	–
4	1	4	6	11	–	–	–	–	–	–	–	–
5	1	2	5	7	12	–	–	–	–	–	–	–
6	1	2	3	7	9	14	–	–	–	–	–	–
7	1	2	3	7	8	12	–	–	–	–	–	–
8	1	2	3	7	8	12	13	–	–	–	–	–
9	1	2	3	5	8	9	13	14	–	–	–	–
10	1	2	3	4	6	9	11	14	–	–	–	–
11	1	2	3	4	6	9	11	14	–	–	–	–
12	1	2	3	4	6	7	11	12	–	–	–	–

Zbývá popsat, jak z těchto dat zvládneme rekonstruovat hledanou nejdelší společnou podposloupnost (NSP).

Ukážeme si to na našem příkladu – jelikož poslední nenulové číslo na posledním řádku je v 8. sloupci, má hledaná NSP délku 8.

$D[12, 8] = 12$ říká, že poslední písmeno NSP je na pozici 12 v posloupnosti B. Jeho pozici v posloupnosti A určuje nejvyšší řádek, ve kterém se tato hodnota také vyskytuje, v našem případě je to řádek 12. Druhé písmeno tedy budeme určovat z $D[10, 7]$, třetí z $D[9, 6]$, atd.

Jednou z hledaných podposloupností je tedy:

poslupnost:	2	3	1	2	2	3	1	2
indexy v A:	1	2	4	5	7	9	10	12
indexy v B:	2	5	6	7	8	9	11	12

Již zbývá jen odhadnout složitost algoritmu. Časově nejnáročnější byl vlastní výpočet hodnot v poli, který se skládá ze dvou hlavních cyklů o délce $|A|$ a $|B|$, což jsou délky posloupností A a B.

Vnořený cyklus while proběhne celkem maximálně $|A|$-krát a časovou složitost nám nezhorší. Můžeme tedy říct, že časová složitost je $\mathcal{O}(|A| \cdot |B|)$.

Posloupnosti jsme si prohodili tak, aby první byla ta kratší, protože pak jsou maximální délka společné podposloupnosti i počet kroků algoritmu rovny délce kratší posloupnosti, a tedy i velikost pole s daty je kvadrát této délky.

Paměťovou složitost odhadneme $\mathcal{O}(N^2 + M)$, kde N je délka kratší posloupnosti a M té delší.

```
# Načtení posloupností s dovolením vynecháme
if lenA > lenB:  # v A bude kratší
    (A, B) = (B, A)
    (lenA, lenB) = (lenB, lenA)
d = [[lenB] * lenA for i in range(lenA)]

maxLen = 0
for i in range(lenA):
    l = 0
    # Máme minimálně to samé, co minule
    if i > 0:
        for j in range(lenA):
            d[i][j] = d[i - 1][j]

    for j in range(lenB):
        if B[j] == A[i]:
            while i > 0 and d[i - 1][l] < j:
                l += 1
            if d[i][l] >= j:
                d[i][l] = j
    maxLen = max(l + 1, maxLen)

j = lenA - 1
C = [None] * maxLen
for i in range(maxLen):
    ii = maxLen - i - 1
    while j > 0 and d[j][ii] == d[j - 1][ii]:
        j -= 1
    C[ii] = A[j]
    j -= 1
# Nyní je v C spočtená NSP posloupností A a B
```

Dnešní menu servírovali
Martin Mareš a Petr Škoda

Vzorová řešení KSP

První série

29-1-1 Připálené placky

Než se pustíme do samotného řešení, tak si úlohu trochu upravíme. Namísto hromady placiček otočených tím či oním způsobem budeme otáčet pole jedniček a nul.

Jednička bude representovat placičku otočenou správně (tedy spálením dolů), nula otočenou špatně. Jedinou dovolenou operací je překlopení (jedničky na nulu a nuly na jedničku) všech hodnot od první až do nějaké i-té (včetně). Tedy vlastně negace prefixu (souvislé podčásti začínající na levém okraji) pole.

Hned na začátku si je třeba uvědomit jednu velmi důležitou věc ze zadání. Nechceme otáčení hodnot, tedy překlápění placiček, opravdu provést. Stačí nám říct, kde všude by to bylo třeba.

Toto pozorování způsobí, že naše řešení bude nakonec lineární a ne kvadratické, jak by se na první pohled mohlo zdát.

Pro začátek se pokusme na pole hodnot podívat jako na soustavu stejnorodých úseků jedniček a nul. Snadno si za chvilku dokážeme, že překlápět hodnotu má smysl jen u prefixů, které končí právě v přechodech mezi souvislými úseky jedniček a nul (či obráceně).

Určitě platí, že v konečném stavu, tedy když jsou všechny hodnoty jedničky, nejsou v poli žádné přechody. Celá věž je totiž jeden stejnorodý úsek.

Otočením hodnot úseku končícím na rozhraní určitě zrušíme jeden přechod. Změníme totiž hodnotu těsně před rozhraním a beze změny ponecháme tu těsně za ním. Což, pokud máme jen dvě možné hodnoty, které se před tím lišily (a vytvářely tak přechod), znamená, že nyní musí být stejné.

Operací také nikterak neovlivníme jiné úseky před nebo za původním rozhraním. Těch za rozhraním se totiž ani nedotkneme a těm před pouze otočíme hodnoty.

Pokud by úsek, v našem případě prefix, který operací změníme, končil mimo přechod, tak nejenže se žádného rozhraní nezbavíme, ale dokonce vytvoříme nové. Znegujeme totiž hodnoty první poloviny nějakého předtím stejnorodého úseku, čímž ho nutně rozdělíme na dvě nové části s různými hodnotami.

Všem ostatním úsekům před koncem prefixu pak pouze přehodíme hodnoty, tedy žádný neodstraníme, a na ty dále za nikterak nezměníme. Skončíme tedy jen s přidáním jednoho přechodu. Je vidět, že má smysl uvažovat jen o úsecích končících v místech již existujících přechodů.

Algoritmus je již nyní snadný. Vytvoříme si pomocnou proměnnou, která si bude pamatovat typ posledního úseku. Na začátku ji nastavíme na hodnotu prvního prvku v poli. Následně pro každý další prvek pole uděláme jednu ze dvou věcí.

Pokud je jeho hodnota stejná jako v pomocné proměnné, tak pokračujeme normálně dál. Pokud je jiná, tj. nastal zlom, tak zahlásíme otočení až do posledního indexu (včetně) a pomocnou proměnnou aktualizujeme hodnotou prvku, který právě zpracováváme; tedy hodnotou která je těsně za zlomem.

Takhle dojdeme až na konec, kde ještě zkontrolujeme, zda je poslední hodnota 1. Když není, tak zahlásíme operaci nad celým polem, tj. otočení všech placek.

Algoritmus zjevně potřebuje pouze jeden lineární průchod plackami. Časová složitost je tedy $\mathcal{O}(N)$, kde N je počet placek.

U paměťové složitosti záleží na tom, jak si ji zavedeme. Pokud si povolíme číst vstup postupně a stejně tak i vypisovat výstup, tak nám stačí konstantní množství paměti. V průběhu si totiž stačí pamatovat onu pomocnou proměnnou. Pokud však použijeme staromdní definici, tak si musíme pamatovat celý vstup najednou, což automaticky znamená i prostorovou složitost $\mathcal{O}(N)$.

Petr Houška

29-1-2 Kupecké počty

Každý kupec ví, kolik zaplatil, tedy se domluví, napíšou všechny částky na papír a spočítají z nich průměr. To je částka, kterou každý z nich chce nakonec zaplatit. Někteří zaplatili víc, jiní méně. V seznamu kupců tedy od výše každé investice odečteme průměr.

Tím dostaneme pro každého kupce buďto kladné číslo – o tolik zaplatil do projektu víc, a tedy tolik má od ostatních dohromady dostat – nebo záporné číslo – o tolik zaplatil méně, a tedy tyto peníze nějak rozdělí svým kolegům.

Můžou nám vyjít i nuly. Nulový kupec zaplatil právě tolik, kolik zaplatit měl, a nemusí se tedy s nikým vyrovnávat. Vyřadíme jej ze seznamu, neboť takových kupců by klidně mohlo být docela dost a zbytečně by kazili časovou i paměťovou složitost zbytku algoritmu.

Kupci se mezi sebou budou vyrovnávat tak, že se vždy potká nějaký dlužník (to je ten, co má platit) s nějakým věřitelem (to je ten, co má peníze dostat) a zaplatí si nějakou částku.

Kolik si můžou zaplatit? Označíme-li celkovou výši dluhu jako D a celkovou výši pohledávky jako P, můžou se vyrovnat nejvýše o $\min(D, P)$. Kdyby si chtěli snad předat víc peněz, buď dojde k předlužení, nebo k přeplacení, což zadání zakazuje.

Jaké je nejmenší množství transakcí, které by teoreticky mohlo proběhnout? Jedna transakce má vždy dvě strany, ovlivní tedy dva kupce. Jednou transakcí tedy dojde k vynulování nejvýše dvou kupců. Je-li tedy kupců celkem N, je nejmenší množství transakcí k vyrovnání $\lceil \frac{N}{2} \rceil$.

Zadání nám tedy dovoluje provést N transakcí. To je ale velmi milé, protože každou transakcí dokážeme alespoň jednoho kupce vynulovat. Stačí když převedeme nejvyšší možnou částku: pokud $\min(D, P) = D$, právě jsme vynulovali dlužníka; pokud $\min(D, P) = P$, vynulovali jsme věřitele.

Rýsuje se nám tedy jednoduchý algoritmus:

Nejprve si spočítáme, kolik měl každý zaplatit, a rozdělíme vstup na dluhy a pohledávky, to vše v čase ($\mathcal{O}(N)$). Poté, dokud budou nějaké pohledávky a dluhy zbývat, vybereme libovolný dluh a libovolnou pohledávku (třeba první ze seznamu) a převedeme maximální možnou částku. Když už žádné pohledávky a dluhy nejsou, máme hotovo a můžeme se jít klouzat.

Proč to funguje? Každým převodem vynulujeme alespoň jednoho kupce, tedy s konečným množstvím převodů budou vynulováni všichni. Též z toho plyne, že převodů provedeme nejvýše $N - 1$, takže jsme splnili zadání.

Jak je to rychlé? Předzpracování trvá $\mathcal{O}(N)$. Každý krok pak trvá konstatní čas: výběr dluhu, výběr pohledávky i samotné vyrovnání.

Paměťová složitost zahrnuje uložení několika proměnných, vstupu velikosti N a dvou seznamů o celkové délce taktéž N, celkem tedy $\mathcal{O}(N)$.

Někteří řešitelé tvrdili, že je potřeba vstup setřídit, nedokázali ale přijít s žádným rozumným argumentem, proč by to mělo být potřeba. Je to lákavé, ale můžeme netřídit a díky tomu si nezavléct do složitosti logaritmus ($\mathcal{O}(N \log N)$ místo $\mathcal{O}(N)$, a to za to stojí, ne?

Jiní se mě zase snažili přesvědčit, že po každé transakci musíme proběhnout celé pole a vyházet z něj ty, kdo už platili. To však není vůbec potřeba. Vždyť jediné dva, kterých se to týká, jsme právě měli v ruce. Takovým postupem se dalo dosáhnout až (pro tuto úlohu jistě impozantní) složitosti $O(N^2)$.

Též se objevilo několik řešení, kde řešitelé porušovali podmínky dané v zadání, například vesele přepláceli, případně si dokonce pořídili banku, která všechno zprostředkovala. Někteří se tohoto prohřešku dopustili skrytě, jiní to dokonce drze deklarovali. Pro odvahu postavit se organizátorům a hrdě řešit jinou úlohu jsem však neměl valného pochopení a uděloval pouze body útěchy.

Úlohu jsme zadali úmyslně s požadavkem na maximálně dvojnásobný počet převodů. Důvod je prostý. Optimální řešení totiž získáme tak, že bychom našli rozdělení množiny kupců do co nejvíce podmnožin tak, aby součet v každé z podmnožin byl nulový. Nicméně už jenom otázka, jestli vůbec existuje podmnožina, jejíž součet je nulový, je NP-úplný problém (anglicky Subset Sum Problem).[42] Jistě pochopíte, že po vás nemůžeme chtít vymyslet řešení takové úlohy v polynomiálním čase.

Jan „Moskyto“ Matějka

[42] `https://en.wikipedia.org/wiki/Subset_sum_problem`

29-1-3 Střídání zbraní

Na začátku si seřadíme gardisty podle výšky od nejnižšího po nejvyššího. Představme si, že si v tomto pořadí chodí vybrat kopí. Označme si m_i počet kopí kratších než i-tý gardista (počítáme od nuly). První gardista si může vybrat ze všech kopí, které je schopen používat, má tedy m_0 možností.

Druhý gardista je schopen používat m_1 různých kopí. Ale vybrat si může pouze z $m_1 - 1$, protože jedno z nich už si vzal první gardista (první gardista je menší než druhý, tedy kopí, které si vzal, by mohl používat i druhý, ten tak o jednu možnost přišel).

Analogicky třetí gardista dokáže používat m_2 kopí, ale dvě z nich už si zabrali první dva gardisté, má tedy na výběr z $m_2 - 2$ možností. Obecně i-tý gardista si může vybrat z $m_i - i$ kopí.

Počet možností, které má i-tý gradista, nezávisí na tom, jaká kopí si vybrali předchozí. Tady je vidět, proč je důležité gardisty brát v pořadí od nejmenšího. Kdyby byl první gardista vyšší než druhý, počet kopí, které zbudou na druhého gardistu, by závisel na tom, jak vysoké kopí si vzal první.

Takhle víme, že pro každou z m_0 možností výběru prvního gardisty existuje právě $m_1 - 1$ možností volby druhého. Celkem tedy je $m_0 \cdot (m_1 - 1)$ možností, jak si může vybrat první dvojice.

Obecně když provádíme několik výběrů po sobě a počet možností, ze kterých v každém kroku vybíráme, nezávisí na volbách v předchozích krocích, je celkový počet možností prostě součinem počtů možností v jednotlivých krocích. Tomuto principu se říká *pravidlo kombinatorického součinu*.

Dostáme tedy celkem

$$m_0 \cdot (m_1 - 1) \cdot (m_2 - 2) \cdot \ldots \cdot (m_N - N)$$

(kde N je počet gardistů) možností, jak si gardisté mohou rozdělit kopí.

Dlužno podotknout, že celkový počet možností samozřejmě nezávisí na tom, v jakém pořadí si gardisté vybírají kopí. Jen když si představujeme, že si vybírají v pořadí od nejmenšího, je o dost snažší možnosti spočítat.

Teď v podstatě stačí dosadit do vzorečku. K tomu bychom ale nejdřív potřebovali spočítat jednotlivá m_i. To je celkem jednoduché. Na začátku si kromě gardistů i kopí seřadíme vzestupně podle délky. Teď budeme postupně procházet jednotlivé gardisty a průběžně si udržovat index největšího kopí menšího než aktuální gardista (tento index odpovídá právě m_i).

Když přejdeme na následujícího gradistu, tento index zvyšujeme, dokud nenarazíme na kopí vyšší než on. Takto spočítáme všechna m_i jedním průchodem přes obě pole v lineárním čase. Podrobněji ve vzorovém programu. Jde o podobný princip jako při slévání setříděných posloupností.

Celková časová složitost je $\mathcal{O}(N \log(N))$, protože tolik času strávíme tříděním a zbytek stihneme v lineárním čase. Paměťová složitost bude lineární.

Program (Python 3):

`http://ksp.mff.cuni.cz/viz/29-1-3.py`

Většina z vás na základní myšlenku přisla. Co jste často opomíjeli, je zdůvodnit, proč řešení funguje. Z výše uvedeného vzorečku není bez komentáře vůbec vidět, že by měl platit. Po přečtení spousty řešení nebylo ani jasné, proč vlastně vstup třídí, když se poté nikdy nezmínily, k čemu to využijí, či dokonce že bez třídění by vzoreček neplatil. Zdůvodnění jsme tentokrát považovali za poměrně důležitou součást úlohy (už proto, že algoritmus samotný je poměrně jednoduchý) a patřičně strhávali body.

Také jste se málokdy zamysleli, co se stane, když neexistuje žádná možnost, jak gardistům kopí rozdat (například některá kopí jsou vyšší než všichni gardisté). V takovém případě se může stát, že některé ze závorek $m_i - i$ vyjdou nulové či dokonce záporné.

V tomhle případě platí, že pokud některá z nich vyjde záporná, bude mezi nimi i nějaká nulová (rozmyslete si), tedy dosazením do vzorečku dostanete správný výsledek: nulu. Ale není to vůbec zřejmé a slušelo by se nad tím v řešení zamyslet.

Ještě se zmíníme o jednom detailu, který je asi přijatelné na úrovni KSP příliš neřešit: výsledný počet možností může být velmi velké číslo. Počítáme součin N čísel velkých až N (v nejhorším případě jsou všechna kopí kratší než všichni gardisté), tedy výsledek může být velký řádově až N^N, což se kromě nejmenších vstupů do běžných celočíselných typů nevejde.

S tím se musíme nějak vypořádat v závislosti na tom, čeho přesně chceme dosáhnout. Pokud by nám výsledek například stačil přibližně či řádově, nejjednodušším řešením je použít nějaký typ s plovoucí čárkou (float, double), který umí uchovávat velmi velká čísla, byť s omezenou přesností, a zacházet s nimi v konstantním čase a prostoru.

Druhou možností je počítat s velkými čísly poctivě přesně. Ale to už obecně nezvládneme v konstantním čase: číslo velké řádově K zabere v paměti místo $L = \mathcal{O}(\log K)$ a vynásobení dvou takovýchto čísel zvládneme poměrně jednoduchým algoritmem v čase zhruba $\mathcal{O}(L^{1.5})$ (jde to i rychleji).

V našem případě $L = \mathcal{O}(\log N^N) = \mathcal{O}(N \log N)$, tedy jedno násobení stihneme v čase $\mathcal{O}(N^{1.5} \log^{1.5} N)$, počítáme N součinů, takže celková časová složitost vzroste na $\mathcal{O}(N^{2.5} \log^{1.5} N)$, paměťová na $\mathcal{O}(N \log N)$.

Filip Štědronský

29-1-4 Zběsilý útěk

Úloha byla velmi jednoduchá, až triviální. Prostě pro každou úsečku projdu všechny hustolesy, zjistím, se kterým se protíná, spočítám dráhu, kterou

urazím hustolesem a kterou řídkolesem, obojí vydělím příslušnou rychlostí a výsledné časy sečtu.

Takové řešení má časovou složitost $\mathcal{O}(U \cdot H)$, kde U je počet úseček a H je počet hustolesů. Tyto hodnoty mají být řádově stejné, aneb $U = \mathcal{O}(N)$, $H = \mathcal{O}(N)$, celková složitost je tedy $\mathcal{O}(N^2)$. Není to moc? Je to moc. Áha. Úloha nebyla tak jednoduchá a triviální, jak by se na první pohled mohla zdát. Přesto i na triviálním řešení šlo nasbírat nemálo bodů.

Připomeňme si techniku zametání roviny přímkou.[43] Projdeme celou oblast například zdola nahoru a poznamenáme si, kde začíná a kde končí který hustoles. Na to si musíme seznam hustolesů setřídit, což zabere $O(H \log H)$. Seznam začátků a konců hustolesů si označíme jako SZKH.

Tahle metoda se nám hodí při procházení lesem. Na začátku stojím v bodě y_0, spočítám si, které hustolesy procházejí přímkou o souřadnici $y = y_0$, a budu tedy znát jejich seznam. Bude se hodit, aby byl setříděný, zabere nám to jen $O(H \log H)$.

Pak posunu přímku do y_1. Abych ji nemusel počítat celou odznova, podívám se do SZKH a zpracuju všechny události, které jsou mezi y_0 a y_1. A celou dobu se přitom dívám, jestli se na přímce náhodou neobjevil nějaký hustoles se souřadnicí mezi x_0 a x_1, a pokud ano, tak jej hned podrobím zkoumání, jestli náhodou nebyl proťat právě zpracovávanou úsečkou.

Pak posunu stejným způsobem přímku do y_2 (a zpracovávám druhou úsečku), pak do y_3 … Postupně tak zpracuju všechny úsečky na vstupu.

Implementačně se pro účely ukládání stavu na přímce bude patrně hodit použít nějaký vyvažovaný vyhledávací strom, třeba červenočerný. Složitost zpracování každé úsečky pak bude $\mathcal{O}(K \log H)$, kde K je počet kroků, které je potřeba udělat.

Tím jsme si ale vůbec nepomohli, protože úsečka zrovna mohla vést z levého okraje území na pravý, takže stejně musíme vyzkoušet všechny hustolesy, i když víme, že protnout má nejvýše jeden.

Neuvědomili jsme si ale, že vůbec nemusíme zkoušet všechny hustolesy mezi x_0 a x_1, ale často jen malý úsek. Nechť aktuální vodorovná přímka má souřadnici y_α a nejbližší následující má souřadnici y_β. Zkoumaná úsečka mezi body (x_0, y_0) a (x_1, y_1) protne tyto dvě přímky na souřadnicích $x_\alpha = x_0 + \frac{(x_1 - x_0)(y_\alpha - y_0)}{y_1 - y_0}$ a $x_\beta = x_0 + \frac{(x_1 - x_0)(y_\beta - y_0)}{y_1 - y_0}$.

Co se stane v obdélníku, který jsme si vytyčili souřadnicemi $(x_\alpha, y_\alpha, x_\beta, y_\beta)$? Především uvnitř něj nemůže ležet žádná vodorovná hrana žádného hustolesa. Kdyby v našem obdélníku snad chtěla ležet horní nebo dolní hrana nějakého hustolesa, musela by být vložena též v SZKH, ale pak bychom se naším obdélníkem vůbec nemohli zabývat, protože mezi y_α a y_β by ještě existovalo nějaké y_υ, takže

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

by y_α a y_β nebyly hned po sobě jdoucí polohy zametací přímky, tedy máme spor, kterým jsme dokázali, že tento obdélník je horních a dolních hran hustolesů zcela prost.

Pokud tedy nějaký hustoles v obdélníku leží, vždy obsáhne celou jeho výšku, a tedy se vždy musí protnout s úsečkou jdoucí po jeho úhlopříčce. Stejně tak obráceně, pokud nějaký hustoles v obdélníku neleží, tak se rozhodně mezi souřadnicemi y_α a y_β nemůže protnout se zkoumanou úsečkou.

Kromě hustolesů tedy budeme mít na naší zametací přímce ještě bod, který bude označovat průsečík zkoumané úsečky s aktuální přímkou, a při každém posunutí přímky s ním pohneme na správné místo a zkontrolujeme, jestli jsme tím náhodou netrefili hustoles.

Co když ale úsečka vede shora dolů, takže musíme projít všechny polohy zametací přímky? Jednou by to nevadilo, ale mohly by takto vést třeba všechny úsečky a pak bychom si se složitostí vůbec nepomohli. Stále totiž každý posun zametací přímky zabere $\mathcal{O}(\log H)$ času, takže bychom si zhoršili složitost na $\mathcal{O}(U \cdot H \log H)$. To nám je platné jak mrtvému zimník.

Všimneme si tedy, že při posouvání zametací přímky sem-tam počítáme pořád dokola totéž – její stav.

Pořídíme si tedy datovou strukturu, která dokáže nějak rozumně uložit všechny možné polohy zametací přímky, a zároveň nebude přehnaně velká. Pokud jsme ochotni obětovat až $\mathcal{O}(H^2)$ paměti, stačí si postupně uložit všechny stavy zametací přímky. Jenomže na vygenerování $\mathcal{O}(H^2)$ dat potřebujeme také $\mathcal{O}(H^2)$ času, takže jsme zase nahraní.

Ne tak docela. Ukládáním všech stavů zametací přímky bychom zase ledacos ukládali redundantně, takže si pořídíme nějaký vhodný druh komprese. Každý hustoles budeme reprezentovat dvěma seznamy ukazatelů: seznamem levých sousedů a pravých sousedů. Seznam levých sousedů říká pro každý interval souřadnice y nejbližší sousední hustoles směrem vlevo. Analogicky vypadá seznam pravých sousedů.

Seznamy postavíme tak, že projdeme zametací přímkou přes všechny události v SZKH. Na začátku není v oblasti žádný hustoles, ale vyrobíme si dva virtuální se souřadnicemi $x_\lambda = -\infty$, $x_\pi = \infty$.

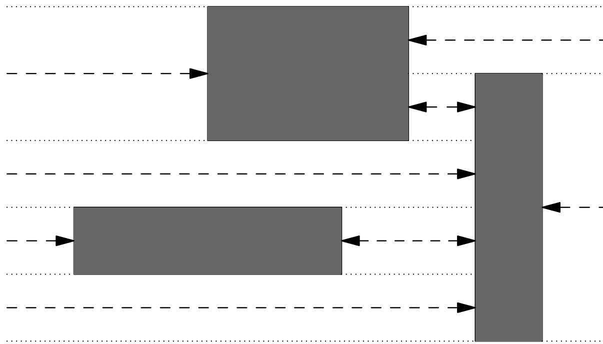
Když potkáme událost „začátek hustolesa“, podíváme se, *mezi které dva* jiné hustolesy jej máme přidat. (Vždycky tam budou alespoň ty dva virtuální.) Do seznamu levých sousedů pravého hustolesa a do seznamu pravých sousedů levého hustolesa tedy přidáme odkaz na právě přidávaný hustoles, do seznamu levých sousedů právě přidávaného hustolesa vložíme odkaz na levý hustoles a do seznamu pravých sousedů právě přidávaného hustolesa vložíme odkaz na pravý hustoles.

Když potkáme událost „konec hustolesa“, podíváme se taktéž, mezi kterými dvěma jinými hustolesy byl. Seznamy levých a pravých sousedů odebíraného

hustolesa uzavřeme, do seznamu levých sousedů pravého hustolesa přidáme levý hustoles a do seznamu pravých sousedů levého hustolesa přidáme pravý hustoles.

Jak dlouho to trvá? Každý posun zametací přímkou potřebuje $\mathcal{O}(\log H)$ času kvůli aktualizaci jejího stavu. K tomu vyrobíme 4 nebo 2 nové odkazy, což je konstanta, která logaritmem nehne. Celkem nás bude vytvoření datové struktury stát $\mathcal{O}(H \log H)$ času, což je stále stejně jako počáteční potřeba setřídění.

Jak velká bude zkonstruovaná datová struktura? To se dá spočítat z algoritmu, kterým ji vyrábíme. Každý hustoles vygeneruje v SZKH jednu událost začátku a jednu událost konce hustolesa. Zpracováním těchto dvou událostí vznikne v datové struktuře právě 6 nových odkazů. Datová struktura tedy spotřebuje celkem $\mathcal{O}(H)$ paměti. To vypadá nadějně.



Na vstupech, které generovalo naše odevzdávátko, už tato úprava běžela dostatečně rychle na získání plného počtu bodů, protože drtivá většina zadaných úseček byla v porovnání s velikostí oblasti velmi krátká. Při zpracování každé úsečky totiž stačilo projít několik málo odkazů – prázdných oblastí, přes které zrovna procházela, i když byla třeba poměrně dlouhá.

Stále se nám sice může stát, že budeme při zpracování každé úsečky projít přes $\mathcal{O}(H)$ odkazů, takže jsme si pomohli zpátky na $\mathcal{O}(U \cdot H)$ v nejhorším případě.

Bylo by fér spočítat, že v průměrném případě na tom budeme líp, nicméně počítat statistiku nad úsečkami a obdélníky v rovině je poněkud ošemetné, jak naznačuje kupř. Bertrandův paradox, tak si to raději odpustíme.

Program s řešením v čase $\mathcal{O}(N^2)$ (C):

`http://ksp.mff.cuni.cz/viz/29-1-4-slow.c`

Program s rychlejším řešením v čase až $\mathcal{O}(N^2 \log N)$ (C):

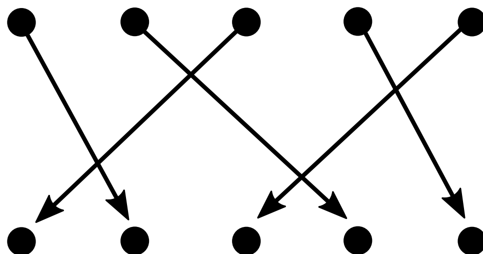
`http://ksp.mff.cuni.cz/viz/29-1-4-fast.c`

Jan „Moskyto“ Matějka

29-1-5 Lučištníci

Na vstupu mějme N lučištníků. Zleva doprava si je očíslujme od 1 do N, stejně jako cíle, na které se míří. Platí, že cíl s číslem k se nachází naproti střelci k. Jako c_k si označme číslo cíle lučištníka k.

Mnoho z vás se snažilo najít nejlepší řešení tímto způsobem: pro každého lučištníka našlo počet drah, s nimiž se ta jeho kříží, a následně postupně odstraňovalo lučištníky s největším možným počtem křížení. Takový postup ale obecně nefungoval. Například v zadání na obrázku byste mohli odstranit i druhého lučištníka, který do správného řešení očividně patří, ačkoliv se jeho dráha kříží se dvěma dalšími drahami.



Uvedeme si řešení běžící v čase $\mathcal{O}(N^2)$ a to následně zlepšíme. Zaveďme pojem *nekřížící se skupina* pro takovou podmnožinu střelců, že jejich dráhy se navzájem nekříží.

Pro každého lučištníka k teď chceme najít co největší nekřížící se skupinu A_k, pro níž platí, že k do ní náleží (tj. bude střílet) a jinak se skládá jen ze střelců nalevo od k. Všimněte si, že v této skupině míří na nejpravější cíl právě lučištník k.

Projdeme teď lučištníky zleva a pro každého budeme chtít spočítat číslo d_k, což je velikost skupiny A_k.

Jistě $A_1 = \{1\}$ a tedy $d_1 = 1$. Pro $k > 1$ vytvoříme skupinu A_k tak, že prodloužíme nějakou předešlou skupinu (každá větší skupina je nutně rozšířením nějaké předešlé skupiny: stačí si uvědomit, že odebráním střelce nejvíce napravo dostaneme kratší skupinu). Jistě chceme vzít takovou s co největším počtem střelců; stále ale musí platit, že se dráhy nekříží. Pokud dáme dohromady všechna pravidla, vyjde nám, že $d_k = \max d_i + 1$, kde $i < c$ a $c_i < c_k$. Právě druhé pravidlo zajistí, že dráha k-tého střelce není v konfliktu s ostatními drahami.

Po doběhnutí cyklu pak nejvyšší d_k označuje maximální počet lučištníků, kteří mohou střílet najednou. Pokud chceme znát konkrétní čísla střelců, stačí algoritmus jednoduše rozšířit: vždy po vypočtení nového d_k si zapamatujeme číslo střelce, rozšířením jehož skupiny vznikla A_k. Na konci cyklu pak najdeme nejvyšší d_k a od něj tyto zpětné odkazy projdeme a vypíšeme.

Zbývá určit složitost. U k-tého lučištníka musíme projít všech $k-1$ předchůdců, dohromady nám to tedy zabere $\mathcal{O}(1+2+\ldots+(N-1)) = \mathcal{O}(\frac{N(N+1)}{2}) = \mathcal{O}(N^2)$ času. Paměťová složitost je $\mathcal{O}(N)$, u každého střelce si ukládáme pouze konstantní počet hodnot.

Existuje však lepší řešení. Nejprve si pro jakoukoliv nekřížící se skupinu lučištníků definujme *pravý okraj*, což je číslo cíle, kam míří lučištník nejvíce napravo. To je zároveň nejvyšší c_i skupiny.

Může se stát, že máme více stejně velkých skupin, a chceme je rozšířit o lučištníka, který stojí napravo od ostatních střelců ve skupinách. Pak upřednostňujeme skupinu s nejnižším pravým okrajem (c_k nového lučištníka musí být větší, než c_k ostatních).

Zaveďme posloupnost $m_i, i \in \{0, \ldots, N\}$. Pokud si vezmeme všechny nekřížící se skupiny velikosti i, m_i bude nejmenší z jejich pravý okrajů. V případě, že skupina velikosti i neexistuje, pak $m_i = \infty$. Platí, že $m_{i-1} \leq m_i$: rozmyslete si, že ze skupiny s i střelci lze vytvořit skupinu s $i-1$ střelci takovou, že pravý okraj zůstane stejný. Pokud pro zadané lučištníky a jejich terče dokážeme vypočítat posloupnost m_i, je maximálním počtem lučištníků nejvyšší takové i, že $m_i < \infty$.

A jak tedy m_i získat? Opět projdeme střelce zleva doprava; na začátku zvolíme $m_0 = -1$, pro $k > 0$ bude $m_k = \infty$. Pro k-tého střelce potom vylepšíme posloupnost takto: protože posloupnost m_i je celou dobu neklesající, dokážeme najít takové l, že $m_l < c_k \leq m_{l+1}$.

Všimněte si, že přidáním k-tého lučištníka nevylepšíme m_i, kde $i \leq l$: pro skupiny velikosti i jsme už nalezli menší pravé okraje. Můžeme však zlepšit m_{l+1}: představte si, že ze skupiny o $l+1$ lučištnících s pravým okrajem m_{l+1} odstraníme střelce nejvíce napravo a přidáme k-tého lučištníka – tím m_{l+1} snížíme, nebo necháme nezměněné. Pro $i > l+1$ toto udělat nemůžeme, museli bychom odstranit dva a více lučištníků a velikost skupiny by se změnila.

Ačkoliv i zde uděláme N kroků, najít správné l lze pomocí binárního vyhledávání, následně už jen upravíme m_{l+1}. Celkový čas je tedy $\mathcal{O}(N \log N)$.

Na závěr důležité pozorování: pokud $c_0, c_1, \ldots, c_N$ zapíšeme jako posloupnost, odpovídá nalezené řešení nejdelší rostoucí podposloupnosti v c_k. Není těžké převést tyto úlohy mezi sebou: ostatně většina řešitelů s plným počtem bodů si této spojitosti všimla.

Kuba Maroušek

Program s řešením v $\mathcal{O}(N^2)$ (Python 3):

`http://ksp.mff.cuni.cz/viz/29-1-5-pomale.py`

Program s řešením v $\mathcal{O}(N \log N)$ (Python 3):

`http://ksp.mff.cuni.cz/viz/29-1-5-rychle.py`

29-1-6 Štítové kouzlo

Nabízí se triviální algoritmus, který vyzkouší všechny možnosti. Jenomže toto nebude fungovat vůbec rychle. Pojďme se podívat proč.

Jednotlivé kroky rozšiřování štítu lze zapsat do posloupnosti $\{L, P\}^{n-1}$, kde n je počet všech hradeb. Nechť pak v této posloupnosti L reprezentuje rozšíření štítu vlevo a P rozšíření štítu vpravo. V této definici můžeme jednoduše získat počáteční úsek hradby spočítáním všech L (musíme začít dostatečně vpravo, aby rozšiřovaní vlevo mělo smysl), takto každá posloupnost jednoznačně určuje postup rozšiřování štítu.

Počet všech takových posloupností činí 2^{n-1}. Vyzkoušení všech možností by nám tedy trvalo vždy $\Omega(2^n)$ času. Jak se z toho vybrodit?

Hladové řešení nefunguje. Vybírat vždy nejvyšší úsek hradby je již vyvráceno příkladem v zadání. Stejně tak nás zradí rozšiřování směrem, kde je větší součet stojících hradeb. Ukažme si to na jednoduchém příkladu. Mějme následující hradby:

$$10\ 10\ \overline{10}\ 0\ 0\ 0\ 0\ 0\ 100$$

Zde se hladový algoritmus rozhodne rozšiřovat štít vpravo. Jenže než se dostane k nejpravějšímu úseku, oba úseky nalevo ztratí příliš mnoho hodnoty. Nakonec zachrání jen $(100 - 5) + (10 - 6) + (10 - 7) = 102$ hradeb. Kdybychom nejprve ochránili část hradeb vlevo a až potom se vydali vpravo, zachráníme jich $10 + 9 + (100 - 7) = 112$, tedy mnohem více.

Hlavní potíž při zkoumání všech možností spočívá ve skutečnosti, že takto spoustu společných postupů mnohokrát zbytečně znovu počítáme. Třeba tyto dvě varianty aktuálních štítů $\overline{A\ B\ C\ D}\ E$ a $A\ \overline{B\ C\ D\ E}$ vychází ze štítu $A\ \overline{B\ C\ D}\ E$, který při zkoumání všech možností celý přepočítáme dvakrát. Takový problém řeší princip *dynamického programování.*[44]

Označme $\check{S}(a, b)$ nejlepší možný součet výšek hradeb v úseku od a do b, který chrání náš štít v kroce $k = b - a$. Dále si označme $V(k, i)$ výšku i-tého nechráněného úseku hradby v k-tém kroce, jejíž hodnota se bude rovnat $\max(0, h[i] - k)$.

Všimněme si, že $\check{S}(a, b)$ se rovná buďto $\check{S}(a, b - 1) + V(k, b)$, anebo $\check{S}(a + 1, b) + V(k, a)$, podle té z možností poskytující vyšší součet chráněných hradeb. Tedy vzorec pro $\check{S}(a, b)$ je $\max\left\{\check{S}(a, b - 1) + V(k, b), \check{S}(a + 1, b) + V(k, a)\right\}$.

Nyní můžeme spočítat $\check{S}(0, n-1)$, jenž odpovídá optimálnímu rozšíření štítu nad celou hradbou. Můžeme postupovat podle definice rekurzí, kde si již spočítaná $\check{S}(a, b)$ zapamatujeme do tabulky, abychom je nepočítali znova.

Můžeme postupovat i jiným způsobem, a to spočítat všechna $\check{S}(a, a + k)$ iterativně po „vrstvách“, tedy nejprve všechna $\check{S}(a, a)$, potom všechna $\check{S}(a, a+1)$,

[44] `http://ksp.mff.cuni.cz/viz/kucharky/dynamika`

…, až nakonec získáme hledané $\check{S}(0, n-1)$. Při takovém způsobu počítání nám stačí si pamatovat předchozí vrstvu, tabulka tedy není nutná.

Jestliže bychom navíc chtěli znát postup, jak jsme štít rozšiřovali, budeme si kromě $\check{S}(a, b)$ pamatovat i $D(a, b)$ říkající, přidáním kterého prvku (nebo ze kterého směru) byl nejlepší štít mezi a a b rozšířen. Projitím $D(0, n-1)$ potom získáme postup rozšiřování štítu v opačném pořadí.

Nyní stačí spočítat časovou a paměťovou složitost tohoto algoritmu. Všech dvojic a, b je $\mathcal{O}(n^2)$, každou spočítáme nejvýše jednou. Na vypočítání každého $\check{S}(a, b)$ spotřebujeme konstantní čas. Celková časová složitost je tedy $\mathcal{O}(n^2)$. Jestliže nám stačí znát pouze nejlepší součet zachráněných hradeb, vystačíme si s $\mathcal{O}(n)$ paměti při použití iterativní metody. Jinak je paměťová složitost shodná s časovou složitostí $\mathcal{O}(n^2)$.

Program (C):

`http://ksp.mff.cuni.cz/viz/29-1-6.c`

Václav Končický

29-1-7 Stromy kolem nás

Úkol 1: Příklady stromů

Děkujeme za pěkné příklady stromů: rodokmen (já a všichni mí biologičtí předkové), řeka se svými přítoky, adresářová struktura na disku, nebo třeba všechny možnosti, jak se může vyvíjet partie deskové hry. Přidáme k nim strukturu webových stránek (do sebe zanořené elementy HTML), programů v programovacích jazycích a nádavkem ještě vztahy mezi větnými členy či větami v souvětí. Stačí? :)

Úkol 2: Strom z postfixového výpisu

Úkol byl formulován trochu nepořádně: pokud o vrcholech stromu vůbec nic nevíme, tvar stromu se z jeho postfixového výpisu určit nedá. Pokud nám ale někdo řekne, kolik má mít který vrchol synů, už to možné je. U stromů výrazů je tato podmínka triviálně splněna: čísla jsou listy, operace mají právě dva syny. Pro jednoduchost budeme nadále předpokládat, že pracujeme se stromem výrazu.

Postfixový zápis budeme procházet zleva doprava a postupně ho překládat na stromy: pokud jsme z `1234++*` přečetli `1234+`, máme zatím hotové jednovrcholové stromy `1` a `2` a třívrcholový strom s kořenem `+`, pod nímž jsou listy `3` a `4`. Tyto stromy postupně slepíme do jednoho velkého, ale z toho, co jsme zatím přečetli, dosud není jasné jak.

Budeme si tedy pamatovat nějakou posloupnost rozpracovaných stromů, říkejme jí *alej*. Kdykoliv ze vstupu přečteme číslo, vytvoříme jednovrcholový strom s tímto číslem a přidáme ho na konec aleje. Přečteme-li naopak nějakou operaci, odpojíme z konce aleje dva stromy, spojíme je pod nově založený kořen

s danou operací a výsledek opět zapíšeme na konec aleje. Obecně objeví-li se na vstupu zápis vrcholu stupně s, spojujeme posledních s stromů z aleje.

Časová složitost tohoto algoritmu je zřejmě lineární. Kdybychom chtěli poctivě dokázat korektnost, pak třeba takto: Uvažme průběh prohledávání stromu do hloubky. Právě stojíme v nějakém vrcholu v a chystáme se ho opustit, tedy do postfixového zápisu toto v vypsat. Vrcholy na cestě z kořene do v jsme už objevili, ale na vypsání teprve čekají. Vše vlevo od této cesty jsme už vypsali, vše napravo naopak ani neobjevili.

O dekódovacím algoritmu pak platí následující invariant: při zpracování v se v aleji nachází posloupnost podstromů visících vlevo od cesty do v (v pořadí shora dolů) následovaných podstromy ležícími pod v (zleva doprava). Jakmile algoritmus uvidí v, podstromy správně poslepuje a invariant bude nadále platit.

Za odměnu vám ukážeme ještě jedno, mírně magické řešení: Postfixový zápis otočíme, čímž se z něj stane prefixový (až na opačné pořadí synů). Ten můžeme poskládat do stromu jednoduchým rekurzivním algoritmem: pokud narazíme na číslo, vytvoříme z něj jednovrcholový strom a skončíme. Pokud na operaci, dvakrát se rekurzivně zavoláme a získané dva stromy spojíme pod společný kořen. Hotovo, $\mathcal{O}(n)$. Formální důkaz složitosti by se vedl stejně jako u minulého algoritmu.

Úkol 3: Nejednoznačný infixový zápis

Prosté, milý Watsone: stačí uvážit zápis `1+2+3`. Ten můžeme uzávorkovat dvěma způsoby: `(1+2)+3` a `1+(2+3)`. Pokaždé vyjde jiný strom. Prefixově tyto stromy zapíšeme `++123` a `+1+23`, postfixově `12+3+` a `123++` – vše bez závorek, z předchozího úkolu už přeci víme, že nejsou třeba.

Úkol 4: Průměr stromu

Jak zadání naznačuje, k měření délky nejdelší cesty se skutečně bude hodit něco počítat během DFS. Kdykoliv se budeme vracet z podstromu s kořenem v, spočítáme dvě čísla: $h(v)$ udávající hloubku podstromu a $\ell(v)$ – maximální délku cesty v podstromu.

Výpočet $h(v)$ už známe ze zadání: v listu platí $h(v) = 0$, ve vnitřním vrcholu se syny $s_1, \ldots, s_k$ musí být $h(v) = \max(h(s_1), \ldots, h(s_k)) + 1$.

Jak tedy s $\ell(v)$? Rozmysleme si možné polohy vrcholu v vzhledem k této cestě:

1. v je list – tehdy je celý podstrom jednovrcholový, pročež $\ell(v) = 0$
2. v vůbec na cestě neleží – tehdy jsme cestu již započítali do některé z délek $\ell(s_1)$ až $\ell(s_k)$.
3. v je koncovým vrcholem cesty – cesta vede z v dolů do nejvzdálenějšího listu, takže měří $h(v)$.
4. v je „zlomem“ cesty – cesta přichází zespoda z některého s_i a zase odchází dolů do nějakého jiného s_j. První část určitě vede z listu, takže měří $h(s_i)$, pak

následuje hrana $s_i v$, hrana $v s_j$ a závěrečná část do listu, délky $h(s_j)$. Vrcholy s_i a s_j samozřejmě volíme tak, abychom použili největší a druhé největší $h(s_i)$.

Pro vrcholy s jedním synem může nastat druhá a třetí možnost, tedy $\ell(v) = \max(\ell(s_1), h(v)+1)$. Je-li synů více, třetí možnost je vždy horší než čtvrtá, takže spočítáme $\ell(v) = \max(\ell(s_1), \ldots, \ell(s_k), m_1 + m_2 + 2)$, kde m_1 a m_2 je první a druhé největší $h(s_i)$.

Nejdelší cestu v celém stromu pak najdeme v $\ell(v)$ kořene.

Našimi výpočty strávíme v každém vrcholu lineární čas s počtem synů, což odpovídá složitosti samotného DFS. Celý algoritmus tedy poběží v lineárním čase.

Průměr stromu podruhé

Předveďme ještě jeden způsob, jak změřit nejdelší cestu. Strom zakořeníme v libovolném vrcholu a. Pak najdeme nejhlubší vrchol v (to už umíme jedním DFS). Tento vrchol prohlásíme za nový kořen a opět najdeme nejhlubší vrchol w. Tvrdíme, že cesta vw je jedna z nejdelších cest.

Tento algoritmus je evidentně lineární, ale není zřejmé, jestli funguje. Pojďme to dokázat sporem: předpokládejme, že existují nějaké stromy, na nichž algoritmus nefunguje. Vyberme z nich nějaký strom T s nejmenším počtem vrcholů (tomu se obvykle říká *minimální protipříklad*). T má alespoň 3 vrcholy – menší stromy algoritmus jistě zvládne.

Bez újmy na obecnosti můžeme předpokládat, že vrchol a není list – v opačném případě místo a vybereme jeho souseda, čímž jsme chování algoritmu nezměnili.

Nyní si všimneme, že v i w jsou listy (jinak by šlo cestu do nich ještě prodloužit). Sousedé těchto vrcholů, označme si je v' a w', listy určitě nejsou.

Proto si ze stromu T sestrojíme strom T' otrháním všech listů. V T' určitě leží vrcholy a, v' a w'. A jelikož přes listy žádné cesty nevedou, náš algoritmus spuštěný v T' z vrcholu a dojde nejprve do v' a pak do w'. Jelikož T byl minimální protipříklad, v T' musí algoritmus vydat správný výsledek, takže cesta $v'w'$ je nejdelší.

Vezměme nyní nějakou nejdelší cestu P ve stromu T. Ta určitě vede z listu do listu, takže odtržením listů získáme nějakou cestu v T', jež jistě nemůže být delší než nejdelší cesta $v'w'$. Proto pro délky cest platí $|P| \leq 2 + |v'w'|$. Pravá strana nerovnosti je ovšem rovna délce cesty vw, takže vw je také nejdelší. Hotovo.

Úkol 5: Strážníci s drony

Mějme nějaký podstrom s kořenem v a přemýšlejme, jak může být hlídaný. Buďto k ohlídání všech jeho hran postačí strážníci, které jsme rozmístili uvnitř podstromu – tehdy mu budeme říkat *samostatný*. Nebo potřebujeme, aby dovnitř dronem doletěl nějaký strážník zvenku (což nutně znamená, že stojí o jednu hladinu nad v; z vyšších pozic by možná doletěl do v, ale už ne dovnitř podstromu) – tehdy hovoříme o hlídání *s výpomocí*.

Navíc u samostatných podstromů potřebujeme rozlišovat, jak vysoko z nich dron vyletí. Tomuto číslu budeme říkat *síla hlídání* a všimneme si, že se pohybuje od 0 do 2. Sílu poznáme podle toho, na jaké nejvyšší hladině se nachází strážník: strážník v kořeni (na 0. hladině) dává sílu 2, strážník na 1. hladině sílu 1, na 2. hladině sílu 0; od 3. hladiny dál nelze hlídat bez výpomoci. Navíc dodefinujeme sílu -1 pro hlídáni s výpomocí.

Nyní budeme pro každé v počítat čísla $h_i(v)$, kde $i \in \{-1, 0, 1, 2\}$. Budou vyjadřovat minimální cenu za ohlídání síly i. Podobně jako u hlídání bez dronů můžeme tato čísla spočítat při návratech z vrcholů, jen rozbor případů bude trochu chlupatější.

Nejprve uvažujme ohlídání síly 2. Tehdy v kořeni leží strážník (snad bychom měli říci, že stojí, ale dron se dá jistě ovládat i vleže), takže musíme započítat cenu $c(v)$ za umístění strážníka. Podstromy ležící pod jednotlivými syny kořene pak mohou být ohlídané libovolně silně včetně síly -1. Platí tedy:

$$h_2(v) = c(v) + \sum_s \min_{i \in \{-1,0,1,2\}} h_i(s),$$

přičemž suma sčítá přes všechny syny vrcholu v.

Dobrá, snížíme sílu na 1. V kořeni jistě strážník není, ale musí být v alespoň jednom vrcholu na 1. hladině. Podstrom pod tímto vrcholem má tedy sílu 2, díky čemuž jsou ohlídány i všechny hrany mezi 0. a 1. hladinou (dron do nich doletí). Ve zbývajících podstromech zakořeněných na 1. hladině si tudíž můžeme vybrat libovolnou sílu od 0 do 2 (-1 nelze). Proto:

$$h_1(v) = \min_s \left(h_2(s) + \sum_{s' \neq s} \min_{i \in \{0,1,2\}} h_i(s') \right).$$

Počítat přímo podle tohoto vztahu by ale bylo moc pomalé: zkoušíme všechny dvojice synů a těch může být až kvadraticky mnoho. Proto si spočítáme sumu

$$\sum_s \min_i h_i(s)$$

přes všechny syny s a pak postupně zkusíme každý člen $\min_i h_i(s)$ odečíst a přičíst místo něj $h_2(s)$. To už jde lineárně s počtem synů.

Snižujeme dále: síla 0. Na 0. a 1. hladině nejsou strážníci, takže každá hrana mezi 0. a 1. hladinou musí být hlídaná zespoda (víme, že nevyužíváme výpomoc shora). Proto podstrom pod každým synem s musí být hlídaný silou alespoň 1. Ale nemůže to být ani víc než 1, protože pak by celková síla vyšla 1. Z toho plyne:

$$h_0(v) = \sum_s h_1(s).$$

Konečně zbývá případ se silou -1, tedy s výpomocí shora. Všechny hrany mezi 0. a 1. hladinou jsou hlídané shora a na 0. ani 1. hladině nestojí strážníci (jinak bychom výpomoc nepotřebovali). Vrcholy na 1. hladině tedy mohou být hlídané silou 0 nebo 1. Proto:

$$h_{-1}(v) = \sum_s \min(h_0(s), h_1(s)).$$

To nám dává kompletní recept na výpočet všech $h_i(v)$ během prohledávání do hloubky. V každém vrcholu tím trávíme čas lineární s počtem synů, takže jsme DFS nezpomalili.

Teď už se stačí jen podívat na hodnoty h_0, h_1 a h_2 v kořeni a vzít z nich minimum (h_{-1} neuvažujeme: už nezbývá, kdo by nám vypomohl). Uff.

Úkol 6: Excentricity

Uvažujme nějaký list ℓ a označme jeho souseda s. Nejdelší cesta z ℓ určitě vede přes s, takže excentricita ℓ je o 1 větší než excentricita s. Stačí tedy odstranit všechny listy, stanovit excentricity zbývajících vrcholů a pak dopočítat excentricity listů.

Můžeme snadno upravit algoritmus ze zadání na hledání centra stromu. Při otrhávání listů si zapisujeme, v jakém pořadí jsme je odebírali. Až nám zbude centrum, spočítáme jeho excentricitu (třeba algoritmem na výpočet hloubky stromu spuštěným v původním stromu z centra). Pak listy v opačném pořadí připojujeme zpět a dopočítáváme jim excentricity.

Všechny tři části algoritmu jistě běží v lineárním čase.

Program (Python):

`http://ksp.mff.cuni.cz/viz/29-1-7-ukol6.py`

Martin „Medvěd" Mareš

Druhá série

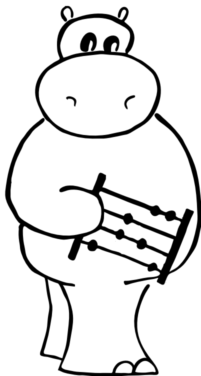
29-2-1 Cesty do školy

Úlohu cesty do školy bylo nejvíce radno řešiti programováním dynamickým. Dostali jsme od vás několik různých způsobů, jak úlohu pomocí dynamického programování řešit, a některé si ukážeme.

Jak ale přijít na to, že se úlohu mám snažit řešit dynamickým programováním? U této úlohy máme dvě vodítka. Prvním je, jak nám skoro všichni z vás napsali, že pokud bychom se problém snažili vyřešit hloupě, dostaneme exponenciální složitost (sled má délku K, tedy díky kombinatorice víme, že existuje N^K možností, jak může sled vypadat).

Druhá nápověda je v tom, že úloha se chová „iterativně“: pokud víme, kolik sledů délky i existuje z vrcholu v do všech vrcholů grafu, víme zároveň, kolik sledů délky $i+1$ existuje z v do libovolného vrcholu x: je to prostý součet počtu sledů délky i pro všechny u, z kterých vede hrana do x.

Na tomto principu bude založeno naše řešení. Formálněji: budeme počítat $D(v,i)$ – tedy počet sledů délky i ze startu do vrcholu v, indukcí podle i. $D(v,i)$ spočítáme jako součet $D(u,i-1)$ přes všechny u, ze kterých vede hrana do v.



Jak takový algoritmus reálně napsat? Například si můžeme u každého vrcholu pamatovat dvě čísla – $D(v,i)$ a $D(v,i+1)$ – a počítat $D(v,i+1)$ ve vlnách. Procházejme všechny vrcholy a u všech, které mají $D(v,i)$ nenulové, budeme propagovat jejich $D(v,i)$ po hranách $v \to u$, které z nich vedou, a přičítat je k $D(u,i+1)$. Až obsloužíme všechny vrcholy, které mají $D(v,i)$ nenulové, stačí nám $D(v,i)$ zapomenout a pokračovat dále tak, že z $D(v,i+1)$ počítáme $D(v,i+2)$.

Jak je vidět, v každém kroku probereme všechny vrcholy, to je N operací, a maximálně jednou propagujeme po každé hraně, což je M operací. Počet kroků je K, tedy dohromady máme složitost $\mathcal{O}(K \cdot (M+N))$. Protože si u každého vrcholu pamatujeme jen dvě čísla, máme paměťovou složitost $\mathcal{O}(N)$ (pokud bychom si pamatovali všechna předchozí $D(v,i)$, hrozilo by, že pro velká K vytečeme z paměti, protože si celkem budeme pamatovat $N \cdot K$ čísel).

Další možností, jak dosáhnout stejné časové složitosti s krapet jinou implementací, je místo probírání všech $D(v,i)$, abychom spočetli $(i+1)$-ní vlnu, probírat všechny hrany vedoucí do v a po nich „tahat“ čísla do v. Opět se dostáváme k tomu, že probereme všechny vrcholy a skrze každou hranu „táhneme“ číslo jen jednou, tedy opět $\mathcal{O}(K \cdot (M+N))$.

Objevila se i řešení využívající prohledávání do hloubky (DFS), která fungují na principu kešování (někdy se lze potkat i s výrazem „memoizace“) – zapama-

tování si něčeho, co jsme již spočítali, pro použití později. Tady konkrétně si v nějakém poli pro každý vrchol u a číslo i pamatujeme počet sledů délky i z u do cíle.

Pokud v DFS voláme rekurzivně další DFS, abychom zjistili $D(v \to s, i)$, tak si po skončení fujnkce můžeme výsledek uložit. Pokud ho budeme někdy potřebovat, místo nového DFS ho jen vytáhneme z paměti. Opět nahlédneme, že časová složitost je $\mathcal{O}(K\cdot(M+N))$, neboť každou hranou projdeme prohledáváním do hloubky K-krát.

Řešení z úplně jiného soudku, které stojí za zmínku, je trochu čarovný trik založený na umocňování matic – pokud ještě nevíte, jak se to dělá, začtěte se do řešení úlohy 28-Z1-6.[45]

Vezměme matici sousednosti grafu. To je tabulka $N \times N$, naplněná 0 a 1. Do i-tého řádku a j-tého sloupce dáme 1 právě tehdy, když v grafu existuje hrana $i \to j$. Pro neorientované grafy tak matice bude symetrická.

Tato matice má magickou vlastnost: podíváme-li se na její K-tou mocninu, číslo v i-tém řádku a j-tém sloupci udává počet orientovaných sledů z i do j. Naše úloha tedy šla vyřešit pomocí tohoto triku jednoduše tak, že graf si reprezentujeme pomocí matice sousednosti, tu umocníme na K-tou a potom odevzdáme políčko (start, cíl). Jak je to rychlé? Násobení matic trvá $O(N^3)$ (ten „obyčejný způsob“, existuje rychlejší algoritmus) a potřebujeme ji vynásobit K-krát. Tedy $O(K \cdot N^3)$.

Ale to není vše. Mocnit můžeme i rychleji: $X \cdot X = X^2$, $X^2 \cdot X^2 = X^4$, $\ldots$, čímž můžeme spočítat K-tou mocninu pomocí $\mathcal{O}(\log K)$ maticových násobení – detaily opět viz 28-Z1-6. Výsledná časová složitost tedy bude $\mathcal{O}(\log K \cdot N^3)$, což už je čas, který pro malé a husté grafy a velká K konkuruje našemu dynamickému programování. Jako cvičení doporučujeme promyslet si, proč mocnění matice sousednosti má tuto vlastnost, neboť je to jen další příklad dynamického programování ;-)

Štěpán Hojdar

29-2-2 Hledání pomsty

Úloha byla označená jako kuchařková, takže bude dobré začít tam. V kuchařce jste se mohli dozvědět o algoritmu KMP, který hledá slovo v nějakém textu. Naše situace je rozdílná pouze tím, že text je zašifrovaný. Přímočaré použití zjevně stačit nebude, bude si tedy potřeba nějak pomoct.

Dále ukážeme, že je jedno jestli máme hledat slovo `POTOPA` nebo `ABCBAD`. Představme si, že jsme totiž našli nějaké místo, kde mohla být `POTOPA`. Pak `P` z potopy odpovídá nějakému konkrétnímu písmenku z textu, řekněme třeba `X`.

[45] `http://ksp.mff.cuni.cz/viz/28-Z1-6/reseni`

Právě `X` bude odpovídat i `A` z `ABCBAD`. Analogicky to bude platit i pro zbylá písmenka.

Takže nás nezajímá, jaká konkrétní písmena jsou kde použita, ale pouze to, kde jsou písmena stejná. Lze tedy jednotlivá stejná písmena nějak „seskupit“. Můžeme tedy například nechat písmena, aby ukazovala na jiné své výskyty.

Nahraďme tedy jednotlivá písmena v hledaném slově číslem, které nám bude říkat, o kolik míst zpět se nacházelo stejné písmeno (a nula v případě, že se jedná o první výskyt daného písmene). Pro slovo `POTOPA` dostaneme `000240`.

Tuto úpravu můžeme stihnout v lineárním čase. Stačí si udržovat pole, kde si budeme pro každé písmeno pamatovat jeho poslední výskyt. Toto pole na počátku inicializujeme například -1. Potom postupně projdeme hledané slovo. Pro každé písmeno se podíváme do pole. Pokud jsme jej nikdy neviděli, zaznamenáme do výsledku `0`. Pokud jsme jej již viděli, tak do výsledku zaznamenáme rozdíl aktuálního indexu a posledního výskytu. V obou případech příslušně upravíme pole posledních výskytů.

Všimněte si, že když stejně upravíme i text, tak nám už obyčejné KMP někdy najde správné řešení. Představme si, že hledáme slovo `POTOPA` v `GHAHGZGHL`. Hledaná `POTOPA` se změní na `000240` a text na `000240240`. První výskyt daného slova nyní přesně odpovídá prvním šesti znakům v textu, našlo by ho tedy i obyčejné KMP. Druhý výskyt však odpovídá v textu `240240`. Všimněte si, že neodpovídají první dvě čísla. Tato čísla znamenají, že poslední výskyt příslušného písmene byl již před „oknem“ `240240`, což sedí s tím, že v jehle máme na příslušných místech nuly. Zbylá čísla již odpovídají přesně.

Stačí tedy upravit KMP tak aby nula v jehle odpovídala kromě nuly v textu i jakémukoliv dostatečně vysokému číslu. To znamená číslu, které je větší než je index dané nuly v jehle. Všimněte si, že tato úprava časovou složitost KMP nijak nezhorší.

Úvodní úpravu na čísla zvládneme v lineárním čase. Stejně tak i stavbu a použití KMP. Celý algoritmus tedy běží v lineárním čase.

Program (Python 3):
`http://ksp.mff.cuni.cz/viz/29-2-2.py`

Janka Bátoryová & Dominik Smrž

29-2-3 Billboardová většina

Ze všeho nejdříve vyřešíme to, že čísla stran mohou být velká. Přečíslujeme si tedy strany celými čísly $0, \ldots, n-1$. To uděláme tak, že si nejdříve čísla všech stran setřídíme, zbavíme se duplicit (třeba tím, že si do nového pole přidáme každé číslo, když ho při procházení setřízeného seznamu čísel uvidíme poprvé). V poli bez duplicit pak budeme mít na indexu odpovídajícímu novému číslu strany její původní číslo.

Kdykoliv pak potřebujeme přeložit číslo strany na číslo, které jsme mu nově přiřadili, tak ho můžeme najít pomocí binárního vyhledávání v poli s již odstraněnými duplikáty. To nám celé bude trvat $\mathcal{O}(n \log n)$ a každý překlad čísla strany bude trvat $\mathcal{O}(\log n)$.

Nyní už jen potřebujeme problém vyřešit pro strany očíslované celými čísly $0, \ldots, n-1$. Postupovat budeme tak, že si pro každý dotaz nejprve najdeme jednoho kandidáta, tedy stranu, která by v tom intervalu mohla mít většinu a o které víme, že pokud většinu nemá, tak ji nemá ani žádná jiná strana. Poté ověříme, zda kandidát opravdu většinu má.

Jak ale kandidáta získáme? Ukážeme si velmi elegantní řešení, se kterým přišel Ríša Hladík. Nejdříve si předpočítáme $\lceil \log n \rceil$ tabulek prefixových součtů pro přeloženou posloupnost stran. Přitom k-té prefixové součty budou udávat, kolik přeložených čísel stran mělo na k-té pozici v binárním zápisu jedničku. Rozmysleme si, jak vypadá k-tý bit přeloženého čísla strany, která má v daném intervalu nadpoloviční většinu (za předpokladu, že taková strana existuje).

Předpokládejme na chvíli, že v daném intervalu existuje jedna strana s nadpoloviční většinou. Označme si b_k hodnotu bitu, kterou má tato strana na k-té pozici. Můžeme nahlédnout, že hodnota b_k je stejná, jako hodnota, kterou má v daném intervalu většina přeložených čísel stran – více než polovina všech čísel v tom intervalu je totiž přeložené číslo strany s většinou.

Většinovou hodnotu k-tého bitu čísel v daném intervalu můžeme spočítat v konstantním čase pomocí prefixových součtů pro k-té pozice. Když takto určíme všechny bity čísla, dostaneme jediného možného kandidáta. Pokud tedy v daném intervalu má nějaká strana většinu, tak ji umíme najít v čase $\mathcal{O}(\log n)$ a předvýpočty jsme strávili $\mathcal{O}(n \log n)$.

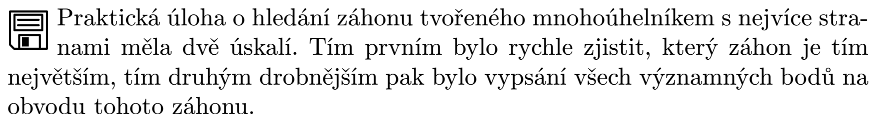
Nyní už jen potřebujeme umět rychle ověřit, zda má v daném intervalu opravdu nalezený kandidát nadpoloviční většinu. Pro každou stranu si uděláme jednu přihrádku a do každé uložíme pole obsahující pozice výskytů dané strany v posloupnosti (to můžeme vytvářet už během přečíslovávání stran).

Kandidáta můžeme ověřit tak, že si pomocí binárního vyhledávání v příslušné přihrádce najdeme index prvního a posledního výskytu kandidáta v intervalu. Rozdíl těchto indexů plus 1 je počet výskytů kandidáta v daném intervalu. No a tak si můžeme jednoduše ověřit, zda má opravdu v intervalu kandidát nadpoloviční většinu. Předvýpočty v této fázi algoritmu zaberou $\mathcal{O}(n)$ času a prostoru a ověřit kandidáta nám bude trvat $\mathcal{O}(\log n)$.

Celková časová složitost je tedy $\mathcal{O}(n \log n)$ na předvýpočet a $\mathcal{O}(\log n)$ na dotaz. Paměťová složitost je $\mathcal{O}(n \log n)$, protože tolik jsme potřebovali na uložení prefixových součtů jednotlivých pozic v druhé fázi algoritmu a více paměti jsme nikde nepoužili.

Kuba Tětek & Jenda Hadrava

29-2-4 Nejsložitější záhon

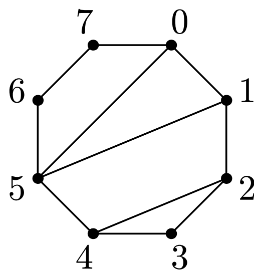


Praktická úloha o hledání záhonu tvořeného mnohoúhelníkem s nejvíce stranami měla dvě úskalí. Tím prvním bylo rychle zjistit, který záhon je tím největším, tím druhým drobnějším pak bylo vypsání všech významných bodů na obvodu tohoto záhonu.

Pojďme se nejdříve zamyslet nad tím, jak může nějaký záhon vypadat. Chodníky na náměstí nám vedou pouze mezi body na jeho obvodu (nikdy se žádné dva chodníky nestýkají někde „uvnitř“ náměstí) a nepřidávají nám žádné nové vrcholy, všechny vrcholy (rohy) záhonů tedy budou tvořeny z původních vrcholů na obvodu náměstí.

Jeden záhon tak bude tvořen vždycky nějakou posloupností vrcholů na obvodu náměstí, pak skokem po chodníku na jiný vrchol na obvodu náměstí a navazující posloupností vrcholů, potom dalším chodníkem a tak dále, dokud se nevrátíme zpět do výchozího vrcholu.

Když si jako příklad vezmeme chodníčky na náměstí ze zadání (pro připomenutí na následujícím obrázku) a budeme ho obcházet po směru hodinových ručiček od vrcholu s číslem nula, potkáme postupně několik záhonů. Na začátku začneme v záhonu 0567, ale hned v nultém vrcholu vstoupíme do záhonu 015, pak ve vrcholu číslo jedna do záhonu 1245 a pak ve vrcholu číslo dva do záhonu 234.



Žádný z těchto záhonů jsme ještě neobešli celý a tak je budeme všechny považovat za *aktivní*. Když se nyní vydáme po obvodu náměstí dál, budeme potkávat zbylé vrcholy těchto aktivních záhonů. Nejdříve ve vrcholu číslo čtyři potkáme poslední vrchol záhonu 234 a tím ho ukončíme. A dál budeme podobným způsobem ukončovat i ostatní aktivní záhony a to v opačném pořadí, než v jakém nám vznikaly.

Toto pozorování nám dává návod k implementaci – aktivní záhony si budeme ukládat do zásobníku. Vždy, když nám vznikne nový záhon, tak tento skončí dříve, než záhon, který byl aktivní před ním (a je tak v zásobníku níže), jinak by se nám někde chodníčky křížily.

Každý chodníček nám bude na zásobníku záhonů zakládat nový záhon. Pokud ze stejného vrcholu vychází více chodníčků, tak chceme nejdříve založit ty záhony, které skončí později. Jinak řečeno na vršku zásobníku chceme ten ze záhonů, který skončí nejdříve – k tomuto záhonu budeme navíc ukládat vrcholy, kterými cestou po obvodu náměstí projdeme.

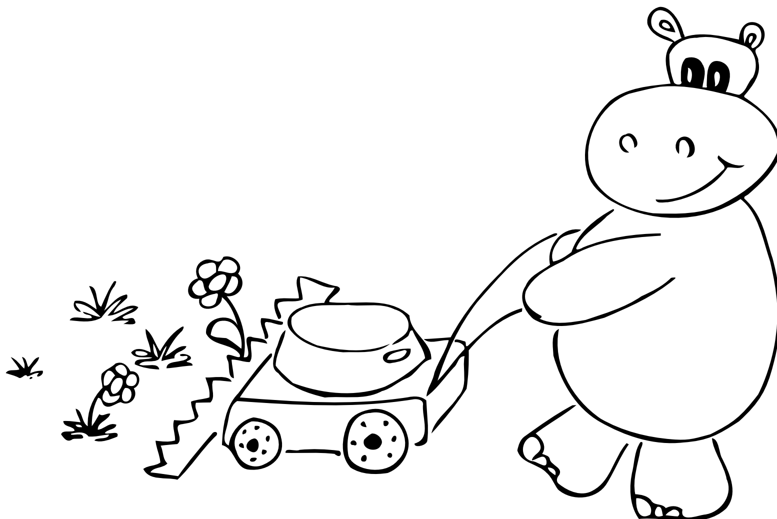
Potřebujeme si tedy seřadit předpisy chodníčků tak, jak je budeme zpracovávat. Předpis chodníčku si vždy upravíme, aby nám chodníček vedl z vrcholu s menším číslem do vrcholu s větším číslem. Pak si je seřadíme od nejmenšího

výchozího vrcholu a chodníčky se stejným výchozím vrcholem pak naopak od největšího cílového vrcholu.

Nyní již máme všechny potřebné stavební kameny, tak si pojďme algoritmus rozmyslet jako celek. Na začátku si utřídíme předpisy chodníčků, přidáme na zásobník aktivní první záhon a pak postupně obcházíme vrcholy na obvodu náměstí. Pro každý vrchol uděláme:

1. Přidáme k aktivnímu záhonu tento vrchol.
2. Pokud tu končí aktivní záhon, tak ho vyjmeme ze zásobníku (a zkontrolujeme znovu – může jich tu končit více).
3. Pro všechny chodníčky začínající v tomto vrcholu přidáme nový záhon na vršek zásobníku.
4. Přesuneme se na další vrchol na obvodu.

Přitom můžeme lehce zjistit, který ze záhonů je největší, a nakonec nám stačí vypsat jeho zajímavé vrcholy (tedy vrcholy, mezi kterými skáčeme po chodníčcích). Jediný problém tohoto řešení je, že je závislé na velikosti náměstí, které může být obrovské (třeba náměstí s velikostí milion se třemi chodníčky). Pojďme to opravit.



Nejvíce se zdržujeme s tím, že skáčeme po jednotlivých vrcholech na obvodu. Namísto toho bod 4 algoritmu změníme tak, aby skočil až na nejbližší zajímavý vrchol. To je buď další začátek nového chodníčku (který získáme jednoduše, protože chodníčky procházíme v utříděném pořadí), nebo na nejbližší konec oblasti (což je konec aktivní oblasti na vršku zásobníku).

Takto se vyhneme zdlouhavému obcházení celého náměstí a skáčeme jenom po zajímavých vrcholech, kterých pro K chodníčků bude $2K$. A namísto toho, abychom k jednotlivým záhonům ukládali všechny vrcholy, tak k nim rovnou uložíme jen tyto zajímavé.

Času nám to teď zabere $\mathcal{O}(K \log K)$ na utřídění chodníčků a $\mathcal{O}(K)$ na jejich obejití, celkově tak $\mathcal{O}(K \log K)$. Paměti spotřebujeme jenom lineárně k počtu chodníčků, neboli $\mathcal{O}(K)$, a potenciálně obrovskému N se tak ve složitosti úplně vyhneme.

Program (C):
`http://ksp.mff.cuni.cz/viz/29-2-4.c`

Jirka Setnička

29-2-5 Plánování návštěv

Nejprve zkusíme úlohu přeformulovat tak, aby se nám s ní lépe pracovalo. V původním znění jsme se ptali, zda se u Petra v N víkendech najde místo pro K kamarádů.

Jedna z možných (a v řešení častých) interpretací úlohy je hledání maximálního párování v bipartitním grafu, o kterém pojednává naše kuchařka o tocích v sítích.[46] Jednu partitu představují kamarádi, druhou víkendy, hrany značí volný čas. Spuštěním algoritmu na hledání párování úlohu vyřešíme velice snadno. Zato ovšem nijak nevyužíváme faktu, že každý kamarád má čas právě ve dvou víkendech, a úlohu jsme vyřešili převedením na složitější problém.

Zkusme si proto zadání představit jako jiný grafový problém. Vrcholy tentokrát budou pouze víkendy, neorientované hrany mezi nimi budou kamarádi – každý spojuje právě dva víkendy. Takto sice vznikne multigraf, to nám ale v úvahách nebude překážet.

Nyní se ptáme, zdali můžeme zorientovat hrany tak, aby vstupní stupeň každého vrcholu byl nejvýše jedna. Hezky se to představuje na papíře – z každé hrany děláme šipku, která ukazuje na víkend, ve kterém přijede daný kamarád.

Sluší se připomenout, že úloha po nás nechtěla najít řešení, nýbrž pouze rozhodnout, zda řešení existuje. Algoritmus se tím zjednoduší, místo ukládání orientace bude vyřešené hrany mazat.

Začneme tím, že už během načítání vstupu budeme počítat stupně vrcholů. S každou smazanou hranou aktualizujeme napočítané stupně.

Jak vypadá načtený multigraf? Nemusí být souvislý, to nám ale nevadí, každou komponentu vyřešíme zvlášť. Může obsahovat izolované vrcholy – to jsou víkendy, ve kterých nemá čas žádný kamarád. Ty můžeme rovnou smazat, do řešení nijak nezasahují.

[46] `http://ksp.mff.cuni.cz/viz/kucharky/toky`

Pokud se někde vyskytuje list (vrchol stupně 1), můžeme jeho hranu BÚNO zorientovat směrem k listu, tím určitě nic nezkazíme. Jak už jsme řekli, vyřešené hrany budeme z grafu mazat. Smazáním hrany vedoucí do listu ovšem může vzniknout další list, proto tento postup opakujeme.

Nyní máme graf, jehož každá komponenta obsahuje vrcholy stupně alespoň dva. Protože každá hrana spojuje dva vrcholy, tak pokud je v komponentě stejně hran jako vrcholů, pak musí být všechny stupně právě dva. Taková komponenta je ale obyčejný cyklus! Ten můžeme zorientovat libovolným směrem a prohlásit jej za vyřešený.

Pokud je ovšem v nějaké komponentě více hran než vrcholů, pak máme více kamarádů než volných víkendů, a řešení nutně nemůže existovat. Tento případ poznáme snadno – existuje vrchol, který má stupeň ostře větší než dva.

Na první pohled složitý algoritmus je tedy vlastně hrozně jednoduchý. Rekurzivně odstraníme všechny hrany do listů a podíváme se, jestli zbylé stupně jsou nula nebo dva. Pokud ano, řešení by šlo vytvořit, v opačném případě máme v ruce důkaz, že nejde.

Algoritmus má paměťovou i časovou složitost $\mathcal{O}(N)$. To je ale příliš! Představte si vstup, který má obrovské N, ale pouze málo hran. Při vhodném formátu máme tedy maličký vstupní soubor obsahující $\mathcal{O}(K+1)$ čísel a algoritmus, který běží v čase lineárním s jejich velikostí. Jinak zformulováno, algoritmus spotřebuje čas exponenciální s počtem čísel na vstupu.

V ostatních úlohách to většinou nevadí, my ovšem dokážeme jednoduchým trikem srazit obě složitosti v průměru na $\mathcal{O}(K)$, což je výrazně lepší. Každá hrana totiž musí být na vstupu popsána a algoritmus poběží v čase lineárním k počtu bitů na vstupu. Stačí místo polí velikosti N používat hešovací tabulky, ve kterých vrcholy vytvoříme, až když budou někde potřeba. Tím všechny iterace přes vrcholy poběží v čase $\mathcal{O}(K)$, a hešovací tabulku můžeme zkonstruovat tak, aby se vešla do $\mathcal{O}(K)$ paměti.

Program (Python 3):

`http://ksp.mff.cuni.cz/viz/29-2-5.py`

Ondra Hlavatý

29-2-6 Souvislá plocha

Problém nalezení největší souvislé plochy se mnoho z vás pokoušelo řešit procházením obrázku po řádcích a spojováním sousedících oblastí. Pojďme si nejdříve nastínit tento způsob a pak ho dotáhneme ještě o kousek dál s pomocí grafů.

Základem všech řešení je procházet oblasti postupně řádek po řádku a nějakým způsobem spojovat stejné oblasti na navazujících řádcích (oblasti, které se táhnou přes více řádků, můžeme rozsekat na koncích řádků).

Úplně triviálním řešením by bylo držet si rozkomprimované vždy dva řádky nad sebou (ty se do paměti podle zadání vejdou), ale průchod přes ně může trvat neúměrně mnoho času vzhledem k velikosti vstupu (představte si třeba vstup obsahující dva jednobarevné dlouhé řádky).

Lepší bude tedy procházet dva na sebe navazující řádky nerozkomprimované. To lehce zařídíme pomocí dvou pointerů, kterými budeme po definicích řádků pohybovat. Na každém řádku si budeme pointerem ukazovat na aktivní oblast a při postupu dál vždy pohneme pointerem, jehož oblast končí dříve (případně oběma, pokud končí na stejné pozici).

Při tomto postupu projdeme dva nad sebou ležící řádky a přitom můžeme nějakým způsobem zpracovat navazující oblasti náležející stejné skupině. Tady se dvě zmíněná řešení rozdělují, ukážeme si obě.

Řešení spojováním

Na každém řádku si budeme chtít držet seznam oblastí a k jaké *nadoblasti* náležejí. Když nám vznikne nová oblast, která nenavazuje na žádnou oblast na předchozím řádku, pořídíme si pro ní i novou nadoblast (reprezentovanou třeba pořadovým číslem).

Podobně jednoduché to bude, i když nová oblast naváže na jednu oblast z předchozího řádku (nebo i více oblastí, ale všechny náležející té stejné nadoblasti), pak jen nové oblasti nastavíme stejné číslo nadoblasti a k nadoblasti připočteme velikost přidávané oblasti.

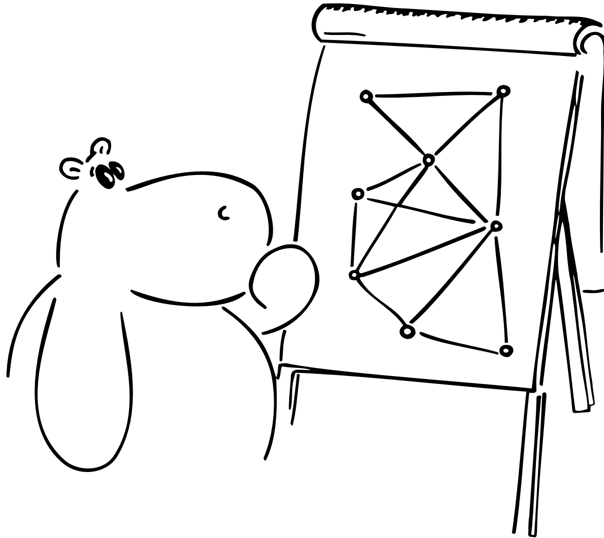
Problematickým ale bude, když nějaká oblast spojí dvě (nebo více) různých nadoblastí z předchozího řádku. V takovém případě musíme tyto nadoblasti spojit, což znamená sjednotit ve všech oblastech těchto nadoblastí číslo nadoblasti na to samé. Tohle potřebujeme, protože tím můžeme ovlivnit i zbytek předchozího řádku (představte si třeba dvoje „hrábě“, jejichž krajní zuby spojí nová oblast).

Tomuto problému se většinou říká *Union-Find* a pokud ho budeme implementovat tak, že přečíslujeme vždy menší nadoblast na větší, tak skončíme s časem $\mathcal{O}(Z \log Z)$ (kde Z představuje počet změn, neboli počet různých oblastí, které potkáme). Dá se dosáhnout i lepšího času (až $\mathcal{O}(Z \log^* Z)$), ale na to vás již odkážeme do kuchařky o minimálních kostrách,[47] kde se tomuto problému věnujeme víc do hloubky.

Grafové řešení

Elegantnější řešení se zakládá na tom postavit si vhodný graf. Každá oblast nám bude představovat jeden vrchol ohodnocený velikostí této oblasti a pokaždé, když se na navazujících řádcích potkají dvě oblasti patřící stejné skupině, tak mezi nimi natáhneme hranu.

[47] `http://ksp.mff.cuni.cz/viz/kucharky/minimalni-kostry`



Na takto vzniklém grafu pak budeme pomocí prohledávání do šířky nebo hloubky hledat souvislé nadoblasti – pro každou oblast si budeme držet, jestli už patří do nějaké nadoblasti, a pokud ne, tak z ní spustíme prohledávání a přitom počítáme velikost.

Jak dlouho nám to bude trvat, závisí na velikosti grafu. Vzniklý graf bude mít vzhledem k Z lineárně vrcholů i hran (hrany plynou z toho, že je to rovinný graf, což lehce nahlédneme). Takže výsledná časová složitost bude jen $\mathcal{O}(Z)$ a ještě si nemusíme komplikovat řešení implementací Union-Find, což je jistě lepší.

Program (Python 3):

`http://ksp.mff.cuni.cz/viz/29-2-6.py`

Jirka Sejkora & Jirka Setnička

Medvědí poznámka: Pokud bychom chtěli šetřit pamětí, můžeme zkombinovat obě řešení do jednoho: procházet obrázek po řádcích a k přečíslovávání nadoblastí použít hledání komponent souvislosti grafu překryvů oblastí sestrojeného vždy znovu pro aktuální dvojici řádků. Tím zachováme lineární časovou složitost a v paměti nám bude stačit udržovat pouze dvojici řádků.

29-2-7 Strom ve stromu

Ve všech úkolech druhého dílu seriálu se nám bude hodit DFS očíslování (čas prvního vstupu $in(v)$ a posledního výstupu $out(v)$ při prohledávání do hloubky) a intervalové stromy.

Úkol 1: Nestromové hrany

Máme pro každý vrchol v zjistit, zda z podstromu pod v vede nějaká nestromová hrana ven. Využijeme toho, ze potomci v jsou právě ty vrcholy, jejichž *in* leží mezi $in(v)$ a $out(v)$. Proto se stačí podívat na minimum a maximum *in*ů přes všechny vrcholy, do kterých vede nějaká nestromová hrana z potomků v.

Pokud je interval od minima do maxima obsažený v intervalu $\langle in(v), out(v)\rangle$, všechny nestromové hrany vedou uvnitř podstromu; jinak aspoň jedna vede ven. (Abychom si nemuseli dělat starosti s tím, jak je minimum a maximum definované, když z podstromu nevedou žádné nestromové hrany, domyslíme si v každém vrcholu smyčku.)

Minima a maxima si můžeme předpočítat během DFS. Vracíme-li se z vrcholu v, do jeho minima započítáme minima všech synů a *in*y vrcholů, do nichž přímo z v vede nestromová hrana. Na každou stromovou i nestromovou hranu se přitom podíváme konstanta-krát, takže celý předvýpočet seběhne v čase $\mathcal{O}(n + m)$. Na každý dotaz pak umíme odpovědět v konstantním čase.

Úkol 2: Nejčastější barva

Vrcholy uspořádáme podle *in*ů a pořídíme si intervalový strom, který nám bude udržovat zvlášť počty červených, zelených a modrých vrcholů. Aktualizace stromu při změně barvy nás stojí logaritmický čas. Chceme-li zjistit majoritní barvu v podstromu, zeptáme se intervalového stromu na zastoupení jednotlivých barev v intervalu mezi *in*em a *out*em kořene podstromu. To stihneme také v čase $\mathcal{O}(\log n)$.

Úkol 3: Odčítání minima

Dostaneme strom s ohodnocenými vrcholy. Pak nám někdo ukazuje na podstromy a my v nich máme od všech hodnot vrcholů odečíst jejich minimum. Aspoň jednomu vrcholu tím hodnotu vynulujeme. Takové vrcholy nadále ignorujeme: ani je nezapočítáváme do minima, ani od nich neodčítáme.

Na chvíli zapomeneme na ignorování nulových vrcholů. Pak je úloha snadno řešitelná pomocí líných intervalových stromů (viz kapitola medvědí knížky odkazovaná ze zadání). Vrcholy uspořádáme podle jejich *in*ů a nad touto posloupností postavíme intervalový strom, jehož vrcholy (odpovídající kanonickým intervalům) si budou pamatovat dva údaje: minimum hodnot v intervalu a instrukci, o kolik se mají snížit všechny hodnoty v intervalu.

Chceme-li spočítat minimum přes podstrom s kořenem v, zeptáme se intervalového stromu na minimum přes interval $\langle in(v), out(v)\rangle$. K tomu stačí zkombinovat spočítaná minima v $\mathcal{O}(\log n)$ kanonických intervalech.

Chceme-li pak od všech vrcholů odečíst nějaké číslo δ, vezmeme stejný interval jako při hledání minima, rozložíme ho na $\mathcal{O}(\log n)$ kanonických intervalů a do každého umístíme instrukci „až tudy půjdeš příště, všechny hodnoty sniž o δ“. Instrukce pak budeme vyhodnocovat líně: intervalovým stromem budeme během všech operací procházet od kořene dolů a pokaždé, když narazíme na nějakou in-

strukci, přesuneme ji do obou synů (případně zkombinujeme s instrukcí, která se už v synech nacházela) a příslušně upravíme jejich minima. Díky tomu platí, že v části stromu, kterou projdeme, už žádné instrukce nezbývají, a operace mohou probíhat, jako by instrukci nebylo.

Líné vyhodnocování přitom vše zpomalí konstanta-krát, tudíž jak hledání minima, tak odečítání trvají $\mathcal{O}(\log n)$.

Pojďme nyní vrátit do hry ignorování nulových vrcholů. To zařídíme tak, že kdykoliv nějaký vrchol vynulujeme, změníme jeho hodnotu na $+\infty$, takže už nadále nebude minimum ovlivňovat. Při odečítání budeme ctít, že $+\infty - cokoliv = +\infty$, takže nekonečna zůstanou nekonečny.

Po každé operaci odečtení tedy potřebujeme najít všechny vzniklé nuly. To zařídíme následujícím průchodem intervalovým stromem do hloubky: Podíváme se na kořen intervalového stromu (ten pokrývá celou posloupnost). Jestliže jeho minimum není 0, nikde v celém stromu neleží žádná 0, takže skončíme. Pokud minimum je 0, rekurzivně se zavoláme na oba syny.

Takto postupně objevíme všechny nuly v listech intervalového stromu. Prošli jsme při tom vrcholy, které leží na cestách z kořenového intervalu do jednotlivých nul, a případně ještě jejich syny (těch je ale nejvýš dvakrát tolik). Stačí tedy každé nule naúčtovat čas $\mathcal{O}(\log n)$ na projití cesty z kořene do této nuly. Stejný čas pak budeme potřebovat na přenastavení nuly na $+\infty$ a přepočítání minim na cestě mezi nulou a kořenem (to můžeme dělat rovnou při návratu z rekurze).

Nalezení a smazání jedné nuly nás tedy stojí čas $\mathcal{O}(\log n)$. Jednou smazaná nula zmizí navždy, pročež vechna mazání nul za celou dobu života datové struktury trvají $\mathcal{O}(n \log n)$.

Co tedy víme o časové složitosti struktury? Inicializace obnáší jen vytvoření intervalového stromu, takže ji stihneme v $\mathcal{O}(n)$. Každá operace stojí $\mathcal{O}(\log n)$, ale ještě musíme za všechny operace dohromady zaplatit $\mathcal{O}(n \log n)$ za mazání nul. Provedení k operací tedy stojí celkem $\mathcal{O}((n + k) \log n)$.

Úkol 4: Hrana mezi podstromy

Nejprve si rozmyslíme, jak bychom nestromové hrany evidovali, kdyby byly orientované. Hranu z vrcholu x do y budeme reprezentovat bodem o souřadnicích $(in(x), in(y))$. Budeme-li chtít zjistit, zda z podstromu pod u vede nějaká nestromová hrana do podstromu pod v, stačí se zeptat na existenci bodu v obdélníku určeném vrcholy $(in(u), in(v))$ a $(out(u), out(v))$.

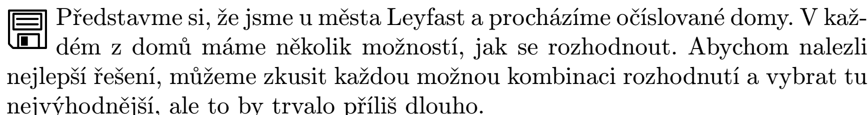
Použijeme-li na obdélníkové dotazy 2D intervalový strom, budeme je vyřizovat v čase $\mathcal{O}(\log n)$. Vypěstování stromu nás bude pro m nestromových hran stát $\mathcal{O}(m \log n)$.

Neorientované nestromové hrany pak můžeme buďto ukládat v obou orientacích, nebo se naopak ptát jak na hrany z u-čkového podstromu do v-čkového, tak opačně. Obojí obnáší jen konstantní zpomalení.

Martin „Medvěd“ Mareš

Třetí série

29-3-1 Verbování



Představme si, že jsme u města Leyfast a procházíme očíslované domy. V každém z domů máme několik možností, jak se rozhodnout. Abychom nalezli nejlepší řešení, můžeme zkusit každou možnou kombinaci rozhodnutí a vybrat tu nejvýhodnější, ale to by trvalo příliš dlouho.

Můžeme také zkusit vybírat další krok *hladově*, neboli vybírat možnost neporušující pravidla, která nám lokálně (pro tento dům) dá nejlepší výsledek. To bude sice rychlé, ale nedostaneme takhle správnou odpověď. Zkuste si to na nějakých vstupech, k rozbití tohoto postupu stačí již ukázkový vstup ze zadání.

Problémem hladového řešení je, že nijak nerespektuje to, že volba v i-tém domě ovlivňuje možné volby v okolních domech. Připomeňme si, co můžeme v domě udělat. Pokud skrz domy půjdeme odzadu, tak můžeme:

- Naverbovat vojáka, pokud jsme tak neučinili v předchozím a neučiníme-li tak v ani následujícím domě.
- Vzít zbraně, pak musíme nutně naverbovat vojáka v následujícím domě.
- Nevzít zbraně ani nenaverbovat vojáka.

Volba, která ovlivňuje výběr v okolních domech, je verbování. Pokud se zkusíme podívat na problém omezený jen na prvních i domů, tak by nás pro další rozhodování mohlo zajímat, jaké nejvyšší bojeschopnosti umíme dosáhnout, pokud si v i-tém domě dovolíme naverbovat vojáka a pokud si zde vojáka nepovolíme naverbovat.

Postavíme si pro to rekurzivní funkci $B(i, (true|false))$, která bude počítat přesně toto. Pokud se nám povede ji spočítat, tak celkovou maximální bojeschopnost získáme zavoláním $B(N, true)$.

Teď si budeme muset sestavit funkci (pro připomenutí, v Z_i je zisk bojeschopnosti při braní zbraní z i-tého domu, v V_i to samé, ale pro verbování z i-tého domu).

- Pro $i \leq 0$ bude mít funkce vždy hodnotu 0.
- $B(i, false)$ bude maximum z:
  - $Z_i + V_{i-1} + B(i-2, false)$ (budeme brát zbraně, což vynutí verbování v $i-1$; lze použít jen pro $i > 1$)
  - $B(i-1, true)$ (nebudeme dělat nic)
- $B(i, true)$ bude maximum z $B(i, false)$ a navíc:
  - $V_i + B(i-1, false)$ (budeme verbovat)

Takovouto funkci lze jednoduše naprogramovat. Horší je exponenciální časová složitost způsobená větvením výpočtu v každém domě. Naštěstí funkce, kterou jsme právě definovali, závisí pouze na dvou parametrech: i a *verbovat*.

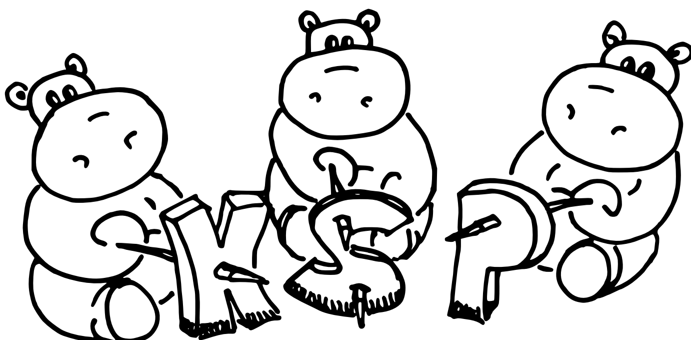
Výsledky volání si tak můžeme ukládat do tabulky velikosti $2N$. Při opakovaném zavolání pak stačí vrátit už dříve spočítaný výsledek. Spočítáme tedy nejvýše $2N$ hodnot funkce B, proto dostáváme lineární složitost vzhledem k počtu hodnot na vstupu.

Zbývá domyslet, jak navíc zjistit jeden z plánů verbování nabývající hodnoty $B(N, true)$. Například můžeme spustit znovu trochu modifikovanou funkci počítající B, která bude nyní vracet odkaz na i-tý prvek spojového seznamu, který reprezentuje jeden ze znaků (z,v,-). Všimněme si, že takto použijeme jen N položek seznamu, protože už máme spočítány hodnoty B a v každém kroku tak voláme pouze jednou naši modifikovanou funkci.

Na trochu (ale jen konstantně) elegantnější řešení s nahrazením spojového seznamu řetězcem se můžete podívat do vzorového programu v C++.

Program (C++):
`http://ksp.mff.cuni.cz/viz/29-3-1.cpp`

Marek Černý



29-3-2 Trpasličí závaží

K porovnávání váhy sad závaží a protizávaží se dalo použít několik různých postupů. Jedním z nich bylo převést zápis na takový, který bude obsahovat jen jedničky a nuly, a tento pak porovnat jako se porovnávají binární čísla. Druhou možností je porovnat zápisy v této „rozšířené dvojkové soustavě“ přímo.

Za chvíli si ukážeme obě možnosti, nejdříve ale proveďme pozorování. Podobně jako v klasické dvojkové soustavě má každá pozice dvojnásobnou hodnotu než pozice předchozí. Když si budeme postupně sčítat hodnoty na dalších pozicích (za nějakou pozicí s hodnotou x), tak nejdříve dostaneme polovinu x, z další pozice čtvrtinu (neboli polovinu té zbývající poloviny do x) a tak dále. Každá další pozice nám součet více přiblíží k hodnotě x, ale nikdy ho nepřesáhne. Protože pozic v binárním čísle je konečně mnoho, přiblíží se na jedničku a víc už ne

– matematici by řekli:

$$\sum_{i=0}^{n} 2^i = 2^{n+1} - 1$$

Takže pokud bude nejlevější nenulová pozice čísla zapsaného v této soustavě záporná, tak i kdyby všechny ostatní pozici již byly kladné, dostaneme nejvýše -1 (a tedy celé číslo bude záporné). A naopak, pokud bude kladná, tak číslo bude kladné.

Podle tohoto můžeme udělat první porovnání a pokud mají nejvyšší pozice obou porovnávaných čísel rozdílná znaménka, můžeme rovnou oznámit výsledek a končíme. Dál tedy budeme zabývat jen případy, kdy jsou znaménka na nejvyšších pozicích stejná.

Další triviální pozorování je, že překlopením všech znamének na opačná vlastně jen změníme znaménko celého čísla.

Převod do dvojkové soustavy

Pro jednoduchost budeme popisovat převod pro čísla, jejichž nejvyšší nenulová pozice je kladná (kdyžtak si je podle předchozího pozorování překlopíme a zapamatujeme si, že je číslo vlastně záporné).

Budeme se chtít zbavit všech výskytů -1 v zápisu čísla. Všimněme si, že následující zápisy můžeme bez změny hodnoty převádět:

- $(1, -1) \rightarrow (0, 1)$
- $(0, -1) \rightarrow (-1, 1)$

V obou situacích děláme to, že přičteme dvojici $(-1, +2)$, což je v součtu nula, a tím vlastně posíláme mínus jedničku dál doleva.

Můžeme tedy zahájit převod od nejmenšího řádu zprava a takto si mínus jedničky průběžně eliminovat, nebo si je posílat dál doleva. Máme ale jednu situaci, kterou jsme si nepopsali – co když se nám vedle sebe objeví dvě mínus jedničky?

Na chvíli si povolíme použít i hodnotu -2 a podívejme se, co se nám při přičtení dvojice $(-1, +2)$ může stát:

- $(-1, -1) \rightarrow (-2, 1)$
- $(-1, -2) \rightarrow (-2, 0)$
- $(0, -2) \rightarrow (-1, 0)$
- $(1, -2) \rightarrow (0, 0)$

Zkusme si to na čísle 5 zapsaném jako $1, 0, -1, -1$. Při převodu zprava dostaneme postupně $1, 0, -2, 1$, pak $1, -1, 0, 1$ a nakonec $0, 1, 0, 1$, což odpovídá číslu 5.

Převod tedy umíme udělat lineárním průchodem číslem od nejmenšího řádu k největšímu a eliminováním mínus jedniček pomocí přičítání vzoru $(-1, +2)$.

Pokud takto převedeme obě čísla, už je snadno porovnáme binárně (průchodem od největšího řádu a hledáním první pozice, kde se liší).

Porovnání odečtením

Pokud vám převod do normální dvojkové soustavy přijde jako nehezký trik, dá se porovnání udělat i odečtením jednoho čísla od druhého. Podle toho, jestli nám výsledek vyjde kladný, nebo záporný (což poznáme podle znaménka nejvyšší jedničky), určíme snadno, které číslo je větší.

Odečítání můžeme dělat klasickým školním postupem od nejmenšího řádu. Pokud nám výsledek odečtení vyjde 1, 0 nebo -1, je vše v pořádku. Pokud nám vyjde menší, než -1, tak musíme udělat převod -1 do vyššího řádu (a k tomu současnému přičteme $+2$, vlastně opět aplikujeme vzor $(-1,+2)$). Pokud nám vyjde naopak větší než 1, přičteme -2 a pošleme převod 1 (tedy použijeme vzor $(+1,-2)$).

Pojďme se podívat na průběh výpočtu třeba čísla $1,-1,-1$, od kterého odečteme číslo $1,1$ (neboli $1-3=-2$). V prvním kroku nám na poslední pozici výsledku vznikne -2, což převedeme na 0 a do vyššího řádu pošleme -1. Na druhé pozici dostaneme -3 (i s převodem), což převedeme na -1 a do vyššího řádu pošleme -1. A nakonec na nejvyšší pozici dostaneme $1-1=0$, čímž získáme správný výsledek $0,-1,0$.

Tento postup zabere také lineární čas vzhledem k velikosti vstupních čísel. Na oba postupy se můžete podívat v přiloženém programu.

Program (Python 3):

`http://ksp.mff.cuni.cz/viz/29-3-2.py`

Jirka Setnička

29-3-3 Skřetí věže

Mnoho z vás přišlo s nápadem zkoušet různé přímky a ověřit, jestli se náhodou nejedná o hledanou osu. Všechny možné přímky však určitě vyzkoušet nemůžeme, těch je nekonečně mnoho. Které přímky tedy připadají v úvahu?

Využijeme toho, že každý bod musí mít při osové souměrnosti svůj obraz. Očíslujeme si tedy body postupně P_0, P_1, …, P_{N-1}. Budeme nejprve předpokládat, že bod P_0 se zobrazí na nějaký jiný bod a ne sám na sebe.

Všimněte si, že pokud bychom věděli, na který bod se P_0 zobrazí, je osa souměrnosti jednoznačně určená: musí to být osa úsečky spojující P_0 s jeho obrazem. My samozřejmě nevíme, na který bod se P_0 zobrazí, ale můžeme vyzkoušet všechny možnosti. Tím získáme $N-1$ přímek, mezi nimiž se určitě hledaná osa nachází (za předpokladu, že nějaká osa existuje a bod P_0 není obrazem sebe sama).

Rozmyslíme si ještě případ, kdy P_0 je sám sobě obrazem a nachází se tedy přímo na ose. Nejjednodušší je vzít místo P_0 bod P_1 a k němu stejným postupem

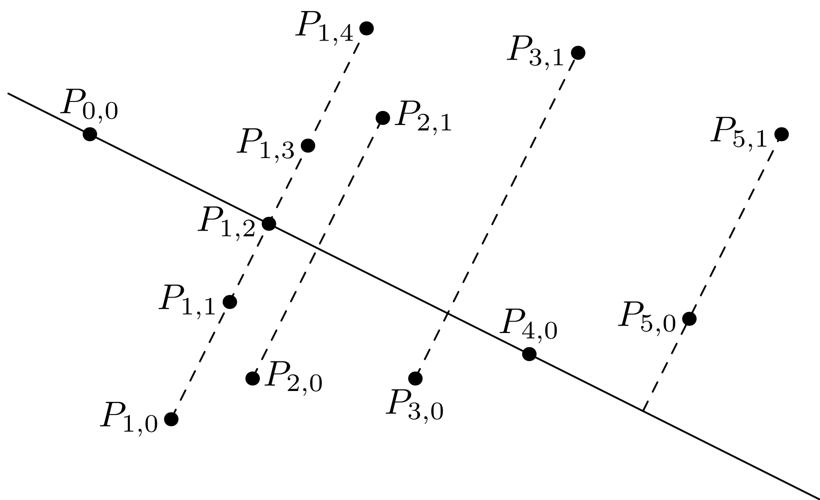
zkoušet body P_2 až P_{N-1}. Takto vyřešíme případy, kdy alespoň jeden z bodů P_0 a P_1 neleží na ose. Pokud by oba ležely na ose, je osou přímka P_0P_1 – tu také přidáme do seznamu kandidátů.

Tímto postupem jsme tedy získali $2N - 2$ přímek, mezi nimiž se určitě osa nachází (existuje-li). Stačí pro každou z přímek ověřit, jestli osou skutečně je, tj. jestli má každý bod svůj obraz.

Nejprve si pro každý bod spočítáme, kam by se při dané ose zobrazil. Pokud bod leží přímo na ose, je sám sobě obrazem. Pokud na ose není, chce to trochu počítání, ale nic náročného. Vezmeme přímku procházející daným bodem, která je kolmá na osu, a spočítáme její průsečík s osou. Tento průsečík se musí nacházet ve středu úsečky spojující daný bod a obraz, takže souřadnice obrazu se už jednoduše dopočítají.

Pro každý bod tedy víme, kam se zobrazí. Teď už stačí zkontrolovat, že v místě obrazu leží nějaký jiný bod – v opačném případě zkoumaná přímka osou není. Pokud bychom při kontrolování obrazu pokaždé procházeli všechny body, bude nám kontrola jednoho obrazu trvat $\mathcal{O}(N)$, kontrola všech obrazů $\mathcal{O}(N^2)$ a prozkoumání všech $2N - 2$ přímek $\mathcal{O}(N^3)$.

Tento postup lze zrychlit vhodným setříděním bodů. Pro každou potenciální osu body setřídíme podle polohy jejich průmětu na osu (to je pata kolmice k ose, na níž leží daný bod). Všimněte si, že tento průmět jsme už spočítali při hledání souřadnic obrazu. Můžeme využít toho, že pokud má být bod P_k obrazem P_ℓ, budou souřadnice jejich průmětů na osu stejné.



Pokud máme tedy takto setříděné body, můžeme je brát postupně. Vždy vezmeme všechny body se stejnými souřadnicemi průmětu a uložíme je do dalšího

pole. Toto další pole opět setřídíme, tentokrát podle pozice na přímce, na které se všechny nacházejí (to je nějaká přímka kolmá k ose). Je zřejmé, že první bod v tomto menším seznamu musí být obrazem posledního, druhý předposledního atd. Pokud si nějaká tato dvojice není navzájem obrazem, některý bod z dvojice nemá obraz a zkoumaná přímka není osou.

Původní setřídění bodů zvládneme v čase $\mathcal{O}(N \log N)$, třídění menších seznamů stihneme ještě rychleji. Kontrolu po setřídění stihneme v lineárním čase. Jelikož zkoumaných přímek je stále lineárně, činí celková složitost $\mathcal{O}(N^2 \log N)$.

Celý tento algoritmus můžeme ještě zrychlit použitím hešovací tabulky.[48] Jednoduše si souřadnice všech bodů do jedné takové tabulky uložíme. Pak pro každý bod spočítáme souřadnice jeho obrazu podle dané osy a podíváme se do tabulky, jestli se tam bod s takovými souřadnicemi nachází. Jelikož zjištění existence v hešovací tabulce proběhne v průměrně konstantním čase, získáme ověření osy v čase průměrně lineárním. Celkově tedy v průměrném čase $\mathcal{O}(N^2)$. Toto řešení už stačilo na získání plného počtu bodů.

Těžiště na pomoc

Pojďme se ale ještě podívat na řešení jiného typu, třeba povede k ještě lepším výsledkům. Někteří z vás chytře využili toho, že těžiště bodů se jistě nachází na ose souměrnosti. Proč tomu tak vlastně je? Těžiště můžeme počítat postupně, tedy tak, že si body rozdělíme do skupinek, spočítáme těžiště skupinek a pak spočítáme těžiště těchto těžišť (každé z těchto těžišť ještě vážíme počtem bodů z příslušné skupinky).

Můžeme tedy vzít body po dvojicích – vždy si vezmeme bod a jeho obraz (body, co leží přímo na ose, necháme samostatně). Těžiště každé této dvojice (tj. střed příslušné úsečky) se nachází přesně na ose. Těžiště samostatného bodu je přímo tento bod. Každé takto spočítané těžiště se nachází na ose, tedy i těžiště těchto těžišť – jakkoli vážené – se bude opět nacházet na ose. Tím pádem tam bude ležet i těžiště původních bodů.

Poznamenáme ještě, že těžiště dokážeme spočítat v lineárním čase. Stačí spočítat průměr x-ových a průměr y-ových souřadnic jednotlivých bodů. Výsledkem jsou souřadnice těžiště.

Takto získáme jeden bod osy, musíme ještě přijít na druhý. Jeden způsob je opět zkoušet středy úseček P_0P_k pro ostatní k. Tento způsob je zdánlivě lepší oproti předchozímu v tom, že můžeme poměrně rychle odmítat přímky, které nejsou osami. Protože aby se P_0 zobrazilo na P_k, musí být přímka P_0P_k se středem S kolmá na osu TS (T je těžiště) a navíc musí TS protínat P_0P_k ve středu úsečky P_0P_k. Všechno toto dokážeme zkontrolovat v konstantním čase a rychle tak odmítnout spoustu potenciálních os.

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

Když však všechny tyto rychlé kontroly uspějí, musíme opět ověřit, jestli je daná přímka skutečně osou. To můžeme provést jedním ze způsobů popsaným v předchozí části.

Nicméně v obecném případě jsme si moc nepomohli. Jako účinný protipříklad se ukáže množina vrcholů pravidelného n-úhelníku sjednocená s množinou bodů rovnostranného trojúhelníku se stejným těžištěm, která způsobí, že celá množina osově symetrická není.

Sami si můžete vyzkoušet, že pokud budou vrcholy z trojúhelníku brány až jako poslední v pořadí, skutečně jsme si, co se rychlosti týká, oproti předchozímu postupu vůbec nepomohli – stále budeme muset zkoušet $\mathcal{O}(N)$ os a žádnou se nám nepodaří odmítnout rychlým způsobem. Každou osu budeme muset ověřit pomalým způsobem, celkově jsme tedy stále na složitosti $\mathcal{O}(N^2)$. Pro jiné přístupy je ještě horší případ, kdy všechny body z trojúhelníku i z n-úhelníku mají od těžiště stejnou vzdálenost.

Zdá se, že najít těžiště nám vlastně vůbec nepomohlo. Kdepak, opak je pravdou. Na následujících řádcích si ukážeme ještě rychlejší řešení, které se právě o těžiště opírá.

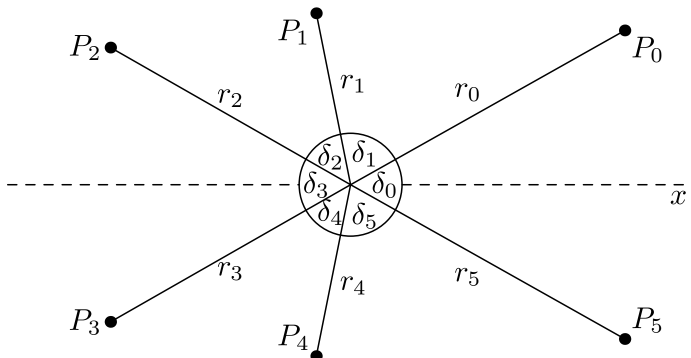
Jde to ještě rychleji

Odrazíme se od souřadnic těžiště. Všechny body posuneme tak, aby těžiště bylo na počátku souřadnicového systému a nadále budeme počítat s těmito upravenými souřadnicemi. Pak víme, že osa souměrnosti, pokud existuje, bude procházet počátkem (tj. těžištěm).

Dále budeme pracovat s takzvanými polárními souřadnicemi, tj. místo x-ové a y-ové souřadnice budeme mít u každého bodu úhel, který svírá x-ová osa s přímkou spojující daný bod s počátkem (těžištěm), a vzdálenost od počátku.

Potom si seřadíme body podle úhlu (prozatím budeme předpokládat, že žádné dva body nemají stejný úhel). Máme tedy u každého bodu P_i úhel φ_i. V tomto setříděném pořadí budeme nadále vrcholy zpracovávat, ale ukáže se, že důležité pro nás bude pamatovat si rozdíl úhlu oproti předchozímu vrcholu. Tedy pro každý vrchol spočítáme $\delta_i = \varphi_i - \varphi_{i-1}$ a pro nultý vrchol $\delta_0 = \varphi_0 + 360° - \varphi_{N-1}$. Všimněte si, že součet přes všechna δ_i nám dá 360°.

Pokud vzdálenost jednotlivých bodů budeme značit pomocí r_i, můžeme teď úhly a vzdálenosti zapsat do řetězce $s = \delta_0 r_0 \delta_1 r_1 \ldots \delta_{N-1} r_{N-1}$. Tedy jednotlivé r_i a δ_i budeme chápat jako jednotlivé znaky řetězce. Všimněte si, že z tohoto řetězce lze zpětně zrekonstruovat původní rozložení bodů (až na rotaci okolo středu, která neovlivňuje osovou souměrnost). Prostě se vždy otočíme o daný úhel δ_i a nakreslíme bod ve vzdálenosti r_i od počátku.



Představme si na chvíli, že osa x je hledanou osou souměrnosti. Uvažujme případ, kdy žádný bod neleží na pravé straně této osy (tedy neexistuje bod s $\varphi_i = 0$). Pak není těžké si představit, že některé úhly a vzdálenosti si musí odpovídat. Konkrétně $r_0 = r_{N-1}$, $\delta_1 = \delta_{N-1}$, $r_1 = r_{N-2}$, $\delta_2 = \delta_{N-2}$ atd. Pro případ, kdy bod leží na pravé straně osy x dostaneme podobné rovnosti, jen trochu posunuté, $\delta_0 = \delta_{N-1}$, $r_1 = r_{N-1}$ atd.

Každopádně si všimněte, že oba případy znamenají, že řetězec s je *skoropalindrom* (palindrom je řetězec, který se stejně čte zepředu i zezadu, tedy že první znak je stejný jako poslední, druhý je stejný jako předposlední atd.). Přesněji řečeno v prvním případě dostaneme palindrom, když za s připíšeme první znak s (tj. δ_0), ve druhém, když před s připíšeme jeho poslední znak (tedy r_{N-1}).

Bohužel nemáme zaručeno, že osou souměrnosti bude osa x. Takže musíme řetězec s vhodně *zrotovat*, tj. opakovaně brát poslední znak řetězce a dát jej na první místo. Všimněte si, že pokud zrotujeme do nějakého stavu

$$r_k\delta_{k+1}\ldots\delta_{N-1}r_{N-1}\delta_0 r_0\ldots\delta_{k-1}$$

a na konec zkopírujeme první znak (r_k), výsledek je palindromem právě tehdy, když osa prochází bodem r_k. Analogicky pokud rotací dostaneme řetězec

$$\delta_k r_k\ldots\delta_{N-1}r_{N-1}\delta_0 r_0\ldots\delta_{k-1}r_{k-1}$$

a opět zkopírujeme δ_k, výsledek je palindromem právě tehdy, když osa prochází středem úsečky mezi body P_{k-1} a P_k.

Uvědomme si, že to jsou jediné možnosti, kde osa souměrnosti může ležet. Máme-li totiž těžiště T (nebo jiný libovolný bod na ose souměrnosti), bod A a jeho obraz A', tak úhel mezi přímkou TA a osou souměrnosti musí být stejný jako mezi osou a přímkou TA'. Představme si, že by tedy osa dělila nějaký úhel δ_k na úhly δ'_k a δ''_k. Nechť je δ'_k ten menší z nich a bod při tomto úhlu (P_k nebo P_{k-1}) označíme K. Bod K se musí zobrazit na jiný bod, který dává s osou úhel δ'_k

(resp. $360° - \delta'_k$, podle toho, jak se na to díváte), ale všechny ostatní body dávají úhel větší (resp. menší), K tedy nemá obraz a zkoumaná přímka není osa.

Stačí vyzkoušet všechny tyto rotace a podívat se zda nejsou *skoropalindromem*. Pokud si budeme uchovávat řetězec ve spojovém seznamu s ukazatelem na začátek i konec, další rotaci vytvoříme v konstantním čase, stačí odebrat prvek z konce seznamu a dát jej na začátek. Samotná kontrola, jestli je řetězec *skoropalindromem*, bude v lineárním čase. Stačí zkontrolovat jestli je první prvek shodný s předposledním, druhý s předpředposledním atd. To zvládneme pomocí dvou ukazatelů, které vždy posuneme o jednu pozici.

Nicméně to opět vypadá, že jsme si vůbec nepomohli. Jednu rotaci zkontrolujeme v čase $\mathcal{O}(N)$, ale rotací je také $\mathcal{O}(N)$, dohromady dostaneme opět $\mathcal{O}(N^2)$. Nicméně častým trikem, když hledáme vhodnou rotaci, je negenerovat nové a nové rotace, nýbrž řetězec zkopírovat dvakrát za sebe. Zkusme to také.

Původně jsme hledali rotaci s palindromem délky $2N - 1$ (nezapomínejme, že N je počet bodů, délka s je tedy $2N$), stejně tak můžeme hledat palindrom délky $2N - 1$ ve zdvojeném řetězci. To můžeme udělat tak, že najdeme nejdelší palindrom liché délky, pokud je delší než $2N - 1$, můžeme jej jednoduše zkrátit (odebíráním vždy dvojicí znaků z kraje) na tuto délku. A jak najít nejdelší palindrom? Na to se podíváme spolu v páté sérii. Zatím jen prozradíme, že to zvládneme v lineárním čase.

Už jsme téměř na konci, ale nesmíme zapomenout ještě na jednu věc. Na začátku tohoto řešení jsme předpokládali, že žádné dva body nebudou mít přiřazený stejný úhel φ_i.

S tím se už dá celkem snadno vypořádat. Trochu upravíme konstrukci řetězce s. Když budeme mít několik bodů stejný úhel, napíšeme jejich vzdálenosti do tohoto řetězce hned za sebou v setříděném pořadí. Protože ale potřebujeme, aby se sekvence bodů se stejným úhlem četla stejně popředu i pozpátku (abychom mohli problém převést na hledání palindromu), tak ji hned za ni zapíšeme znova, v opačném pořadí. Náš řetězec může vypadat například takto: $\delta_0 r_0 r_1 r_2 r_2 r_1 r_0 \delta_3 r_3 r_3 \delta_4 \ldots$

Odpovídajícím způsobem upravíme i délku hledaného palindromu. Ta je $2N + A$, kde N je počet bodů a A je počet nenulových δ_i.

Pojďme si to shrnout a podívat se na výslednou složitost. Posunout body podle těžiště, stejně tak spočítat polární souřadnice zvládneme v lineárním čase. Setříděním bodů podle úhlů strávíme $\mathcal{O}(N \log N)$. Zkonstruovat a zdvojit řetězec opět zvládneme lineárně a konečně jsme slíbili, že nalezení samotného palindromu jde také rychle. Celkově jsme se tedy konečně dostali na časovou složitost $\mathcal{O}(N \log N)$.

Program (Python 3):
`http://ksp.mff.cuni.cz/viz/29-3-3.py`

Dominik Smrž & Martin „Medvěd“ Mareš

29-3-4 Mezi hlídkami

Ze všeho nejdříve úlohu převedeme na variantu, kde je zakázáno vstupovat pouze na políčka s hlídkou, ale na sousední šlápnout můžete. Uděláme to tak, že přidáme „virtuální“ hlídku na všechna pole sousedící s hlídkami ze zadání. Odpověď pro takto upravenou úlohu a vstup bude stejná jako pro původní zadání.

Jednou z možností, jak se úloha dala řešit, bylo si na začátku najít pomocí prohledávání do šířky nejkratší cesty mezi všemi dvojicemi políček. Při odpovídání na dotaz už budeme mít délku nejkratší cesty předpočítanou a můžeme ji jen vrátit.

Protože počítáme cesty na poli velikosti $N \times M$ políček, zabere nám vyhledání nejkratších cest z jednoho políčka do všech ostatních čas $\mathcal{O}(NM)$ (jedno prohledávání do šířky). Jelikož potřebujeme počítat vzdálenost mezi všemi dvojicemi políček, musíme prohledávání spustit pro každé políčko samostatně, což zabere $\mathcal{O}((NM)^2)$ času a $\mathcal{O}((NM)^2)$ paměti. To není příliš rychlé, zkusíme to vylepšit.

Rychlejší postup

Můžeme si všimnout, že vzhledem k malému počtu hlídek nejkratší cesta půjde většinu času po části pláně, kde žádné hlídky nejsou. Toho jde využít a dosáhnout tak lepší časové i paměťové složitosti. Dál v řešení budeme pracovat se souřadnicemi startu x_s, y_s a souřadnicemi cíle x_c, y_c.

Řešení si rozdělíme na dva případy – buď neexistuje hlídka, která je v obdélníku definovaném startem a cílem, a délka nejkratší cesty je tedy $|x_s - x_c| + |y_s - y_c|$, nebo nám po cestě nějaká hlídka bude překážet a budeme to muset vyřešit. Tyto dva případy také musíme být schopni odlišit, což vyřešíme v poslední části řešení.

Chtěli bychom si předpočítat nejkratší cestu mezi každými dvěma vrcholy, jenže to by trvalo příliš dlouho. Všimneme si ale, že ve sloupcích a řádcích, kde není hlídka, se „nic neděje“. Pokud je takových řádků nebo sloupců více vedle sebe, tak vždy všechny sloučíme (zkontrahujeme) do jednoho a poznamenáme si ke každému políčku, kolika políčkům odpovídá ve vertikálním a horizontálním směru. Jednotlivá zkontrahovaná políčka tak mohou odpovídat i docela velkých obdélníkům v původní pláni. Nejkratší cesty si pak předpočítáme až na této upravené pláni.

Při slučování si u každého políčka z původní pláně navíc zaznamenáme, kde je jeho „sloučená verze“ v kontrahované pláni. To se nám bude později hodit a takto se k této informaci dostaneme v konstantním čase.

Na této kontrahované pláni budeme chtít hledat nejkratší cesty. Může se zdát, že po úpravě, kdy některá políčka reprezentují větší vzdálenosti, nebude běžné prohledávání do šířky stačit, ale můžeme si rozmyslet, že díky kontrahování vždy celých řádků a sloupců bude i obyčejné prohledávání do šířky stále

dostávat políčka ve správném pořadí – takové prohledávání do šířky totiž přiřadí políčkům stejná ohodnocení, jako kdybychom ho spustili na původní pláni. Nad tímto prohledáváním do šířky také můžeme přemýšlet jako nad Dijkstrovým algoritmem, který namísto haldy používá frontu.

Protože hlídek bylo k, tak takto upravená pláň má rozměry $O(k^2)$ (mezi sousedními hlídkami je maximálně jeden zkontrahovaný sloupec nebo řádek) a předpočítat si všechny nejkratší cesty tedy bude trvat $O(k^4)$.

Potřebujeme ještě umět zjistit, zda je v obdélníku definovaném startem a cílem hlídka. To můžeme udělat jednoduše pomocí dvoudimenzionálních prefixových součtů (o nich si můžete přečíst třeba v naší kuchařce základních algoritmů).[49] Hlídky budeme považovat za jedničky a prázdná políčka za nuly. V daném obdélníku pak bude hlídka právě tehdy, když je v něm nenulový součet.

Dotaz pak bude vypadat následovně: Pokud mezi políčky není žádná hlídka, délka nejkratší cesty je $|x_s - x_c| + |y_s - y_c|$. Pokud mezi nimi hlídka je, využijeme naší kontrahované plánĕ, kde máme pro každou dvojici políček předpočítanou nejkratší cestu

Drobnou nesnází je, že start (nebo cíl) mohou ležet uvnitř nějakého kontrahovaného obdélníku. Můžeme si rozmyslet, že část cesty, která je v tomto obdélníku, může jít libovolnou nejkratší cestou do rohu nejbližšího k cíli (respektive startu), tuto část cesty spočítáme jako v případě výše.

A dál už pak vyhledáváme jen ve zkontrahované pláni, respektive v předpočítané datové struktuře délek cest mezi dvojicí zkontrahovaných políček.

Předpočítání kontrahované pláně bude trvat $O(MN + k^4)$ a stejně prostoru bude zabírat výsledná datová struktura. Na dotazy pak budeme schopni odpovídat v konstantním čase.

Kuba Tětek

29-3-5 Dračí zámek

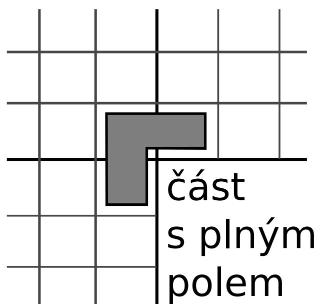
K zadání této úlohy jste všichni dostali nápovědu, totiž kuchařku o metodě Rozděl a panuj. Toho jste všichni správně využili, a třebaže došlá řešení využívala různé přístupy, vždy byly založeny na této metodě.

Takže jak se k úloze správně postavit? Připomeňme, že čtvercová mřížka má rozměry $N \times N$, kde N je nějaká mocnina dvojky. Je snadné si všimnout, že pro $N = 1$ má úloha triviální řešení: máme jediné plné pole.

Pro větší N chceme celé zadání rozdělit na menší úlohy. Mřížku uprostřed rozsekneme na čtyři podmřížky, každou s rozměry $\left(\frac{N}{2}\right) \times \left(\frac{N}{2}\right)$. Na podmřížku, kde se vyskytuje plné pole, můžeme rekurzivně zavolat stejný algoritmus. Pro

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

ostatní podmřížky si pomůžeme tak, že do rohu, který vzájemně tvoří, vložíme navíc dílek:



V každé z nich se teď nachází plné pole, tudíž i na ně se můžeme zavolat rekurzivně.

Celý algoritmus slouží zároveň i jako důkaz, že kteroukoliv mřížku velikosti $N = 2^k$ lze pokrýt dílky. Počáteční pozorování pro $k = 0$ je indukčním předpokladem, rozdělení na podmřížky indukčním krokem.

Pokud bychom chtěli algoritmus implementovat (což jsme od vás nepožadovali), je zajímavé se podívat na časovou složitost. Budeme následovat výše uvedený postup a vytvoříme funkci, která vždy položí nový dílek a navíc pro $N > 1$ zavolá rekurzivně čtyřikrát sama sebe. Samotný průběh funkce má konstantní časovou složitost (pouze vypočítáme polohu plného pole), takže zbývá vyřešit, kolikrát se zavolá.

Zkusíme pro změnu přemýšlet odspodu: voláme funkci na každou podmřížku velikosti 1×1, a těch se v mřížce nachází N^2; dále na každou podmřížku velikosti 2×2, těch je celkem $\frac{N^2}{4}$. Dostáváme se tak k součtu řady: $N^2 + \frac{N^2}{2} + \frac{N^2}{4} + \ldots + 1$. K jejímu řešení můžeme využít například kuchařkovou větu (Master Theorem), která nám odpoví, že celková časová složitost je $O(N^2)$.

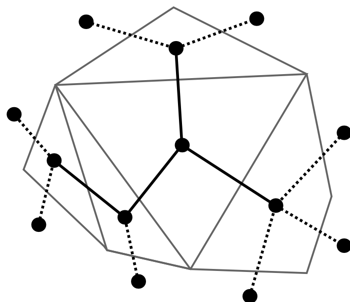
Program (Python):

`http://ksp.mff.cuni.cz/viz/29-3-5.py`

Kuba Maroušek

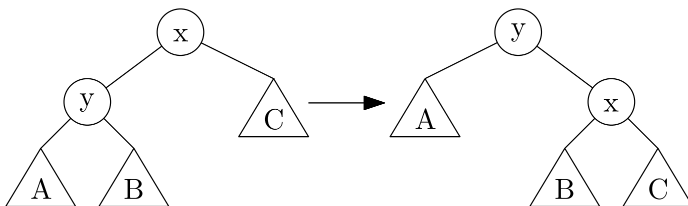
29-3-6 Obrazec pro draka

Pro jednodušší řešení úlohy je dobré převést si mnohoúhelník na něco, s čím se pracuje lépe. Vytvoříme si graf, jehož vrcholy představují trojúhelníky a hrany reprezentují tyče. Vrcholy sousedních trojúhelníků jsou tedy spojené hranou. Ještě se nám bude hodit, když i strany mnohoúhelníku budou hrany a za každou stranou budeme mít také vrchol. Můžeme si všimnout, že tento graf je stromem.

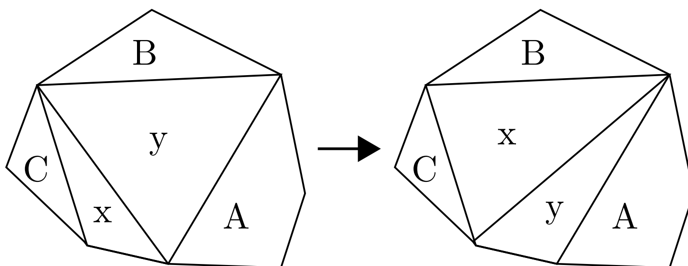


Nyní si můžeme vybrat jeden z vrcholů mimo mnohoúhelník a prohlásit ho za kořen našeho, nyní binárního, stromu. Teď by nás zajímalo, co udělá s naším stromem jedno překlopení tyče. Ukážeme si, že odpovídá operaci stromové rotace.

Rotace je „otočení“ hrany mezi dvěma vrcholy, kde zachováme pořadí vrcholů a podstromy převěsíme, viz obrázek.



Přesně tohle udělá zvednutí tyče a její umístění napříč:



Počáteční i cílový obrazec převedeme na binární strom, kde za kořen zvolíme ten stejný vrchol (neboli vrchol za stejnou stranou mnohoúhelníku). Teď hledáme, jak převést pomocí rotací jeden na druhý. Ještě je dobré si očíslovat listy – tedy vrcholy za hranami mnohoúhelníku. Aby dva stromy reprezentovaly stejnou triangulaci, tak musí sedět i očíslování listů.

Stromové rotace mají jednu důležitou vlastnost, totiž zachovávají pořadí listů. Nestane se nám tak, že by se pořadí listů nějakým způsobem pomíchalo (což by znamenalo, že by se nám i mnohoúhelník musel nějak překlápět). Zachování

této vlastnosti je důležité, protože bychom jinak mohli sice vymyslet způsob, jak přejít od jednoho stromu k druhému, ale neseděly by nám listy, tedy by vlastně vůbec nemuselo jít o tu samou triangulaci.

Nyní už jsme velmi blízko cíle. Připomínáme, že hledáme libovolnou posloupnost rotací, nemusí být nutně nejkratší. Pokud budeme opakovat dostatečně dlouho rotace do jednoho směru, třeba doleva, získáme lineární strom.

Stačí začít u kořene a opakovat rotace, dokud není vpravo jenom list. Poté půjdeme k jeho pravému synovi a budeme to opakovat. V každé rotaci jeden vrchol zařadíme do lineárního stromu a tedy nám to bude trvat $\mathcal{O}(n)$ kroků.

To samé můžeme provést s cílovým stromem. Využijeme toho, že k rotaci vpravo je rotace vlevo inverzní operací. Abychom získali hledanou posloupnost překlopení, můžeme provést rotace stromu počátečního obrazce na lineární strom a pak pozpátku ty, co jsme provedli s cílovým stromem.

Strom sestrojíme v lineárním čase, obě převedení na lineární strom zvládneme $\mathcal{O}(n)$ rotacemi, a protože každá trvá jen konstantní čas, tak celkový čas bude $\mathcal{O}(n)$. Počet rotací bude také $\mathcal{O}(n)$.

Najít nejkratší posloupnost rotací, která převádí jeden binární strom na druhý, v polynomiálním čase zatím bohužel neumíme. Jestli to vůbec jde, je stále otevřený problém. Umožnilo by nám to efektivně spočítat rotační vzdálenost dvou stromů, totiž kolik nejméně rotací je potřeba pro převedení jednoho stromu na jiný. To je hezká metrika – způsob, jak měřit „vzdálenost“ (rozdílnost) dvou stromů.

Jirka Sejkora

29-3-7 Stromoví předci

Úkol 1: Chytřejší značkování

Ke kořeni chceme stoupat z obou vrcholů současně. Jelikož nám asi nedali paralelní počítač, budeme to co nejvěrněji simulovat. Vždycky jeden krok na cestě z prvního vrcholu, pak jeden z druhého, a tak dále. Vrcholy na obou cestách značkujeme a jakmile první vrchol dostane obě značky, je to hledaný společný předchůdce (LCA).

Jak dlouho to trvá? Označme d_1 a d_2 vzdálenosti k LCA. Tento LCA dostane první značku po d_1 krocích, druhou po d_2. Algoritmus se tedy zastaví po $\mathcal{O}(\max(d_1, d_2))$ krocích, což je totéž jako požadovaných $\mathcal{O}(d_1 + d_2)$.

Úkol 2: Minimum svislé cesty

Chceme počítat minimum ohodnocení hran na „svislé“ cestě mezi vrcholem x a jeho předkem p. Předpočítáme si hloubky vrcholů $d(v)$, takže dotaz umíme přeložit na minimum na cestě mezi x a $Pra(x, d(x) - d(p))$.

V zadání jsme ukázali, jak si předpočítat skočky, tedy hrany z v do $Pra(v, 2^i)$, a pak počítat $Pra(w, k)$ složením $\mathcal{O}(\log n)$ skoček. Nyní si pro každou skočku

předpočítáme ještě minimum z ohodnocení přeskakovaných hran. To pro každou zvládneme v konstantním čase složením dvou už spočítaných minim. A při skládání $Pra(v, 2^k)$ ze skoček rovnou složíme i příslušná minima.

Předvýpočet trvá $\mathcal{O}(n \log n)$, pak odpovídáme v $\mathcal{O}(\log n)$.

Úkol 3: Součet svislé cesty

Součet je mnohem jednodušší. Spočítáme si analogii prefixových součtů, tedy součty $S(v)$ všech hran z kořene do v. Součet na cestě mezi x a jeho předkem p pak je prostě $S(x)-S(p)$. Předvýpočet trvá $\mathcal{O}(n)$, na dotazy odpovídáme v $\mathcal{O}(1)$.

Úkol 4: Minimum obecné cesty

Minimum nebo součet na obecné cestě mezi x a y spočteme tak, že nejdříve nalezneme $\ell = lca(x, y)$. Pokud je $x = \ell$ nebo $y = \ell$, cesta je svislá a jsme hotovi. V opačném případě cestu rozložíme na dvě svislé cesty: z x do ℓ a z ℓ do y, pro které už umíme odpovědět.

Úkol 5: Součet pomocí ET-posloupnosti

Dostaneme ET-posloupnost, do níž jsme při průchodu hranou směrem dolů napsali její ohodnocení, a při návratu nahoru minus ohodnocení. Chceme počítat součty na svislých cestách, opět označíme x nižší vrchol cesty a p ten vyšší.

Nejprve dokážeme, že součet všech čísel mezi dvěma výskyty téhož vrcholu v v ET-posloupnosti je nulový. Určitě to stačí dokázat pro „sousední" výskyty, tedy takové, mezi nimiž jsme v nenavštívili. Nechme DFS běžet tak dlouho, než dospěje do prvního z našich dvou výskytů vrcholu v. Pak bude pokračovat do některého ze synů vrcholu v, načež proleze celý podstrom pod tímto synem, a nakonec se vrátí zpět do v, což bude druhý z výskytů. Kdykoliv při tomto průchodu prošel po nějaké hraně dolů, vrátil se po ní pak nahoru, takže k celkovému součtu tato hrana přispěje nulou.

Nyní dokážeme, že součet všech čísel mezi jakýmkoli výskytem vrcholu p a jakýmkoli výskytem vrcholu x je roven součtu cesty mezi x a p. Vzhledem k předchozímu odstavci si můžeme vybrat konkrétní výskyty: pro p si vybereme ten, z nějž odejdeme hranou vedoucí směrem k x; pro x zvolíme první výskyt.

Uvažujme, co DFS provede mezi těmito dvěma výskyty. Určitě prošlo po cestě z p do x. Všechny podstromy odpojující se od této cesty doleva, kompletně prošlo, takže celkem přispěly nulou. Podstromy odpojující se vpravo vůbec nenavštívilo. Nenulou tedy přispěly pouze hrany na cestě.

K odpovídání na daný typ dotazů tedy stačí předpočítat prefixové součty pro ohodnocenou ET-posloupnost, a pamatovat si pro každý vrchol jeho libovolný výskyt. To zvládneme v lineárním čase, na dotazy pak odpovídáme odečtením dvou prefixových součtů, tedy v konstantním čase.

Úkol 6: Syn v zadaném směru

Dostaneme vrchol x a jeho předchůdce p. Chceme najít toho ze synů vrcholu p, který leží na cestě z p do x. Použijeme podobný trik jako pro výpočet

LCA. ET-posloupnost ohodnotíme hloubkami a budeme hledat vrchol s minimální hloubkou ležící mezi libovolným výskytem vrcholu x a posledním výskytem vrcholu p.

Z toho přímo nic nezjistíme: minimum se evidentně nabývá pro vrchol p. Ale pokud v zadaném intervalu nalezneme *nejlevější* minimum, je to ten z výskytů vrcholu p, do nějž jsme se z x vrátili. Těsně před ním v posloupnosti leží hledaný syn.

Stačí nám tedy vylepšit strukturu pro intervalová minima, aby vždy našla nejlevější výskyt minima v intervalu. To se dá zařídit třeba tak, že do ET-posloupnosti pro i-tý výskyt vrcholu v místo hloubky $d(v)$ zapíšeme uspořádanou dvojici $(d(v), i)$ a dvojice budeme porovnávat lexikograficky. Nebo můžeme dvojici zakódovat do přirozeného čísla $d(v) \cdot n + i$.

Takto upravená struktura bude stejně rychlá jako ta původní, takže s předvýpočtem v $\mathcal{O}(n \log n)$ dokážeme odpovídat v konstantním čase.

Martin „Medvěd“ Mareš

Čtvrtá série

29-4-1 Odevzdávání písemek

O třech slibných algoritmech

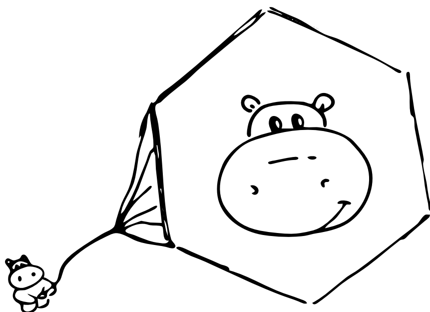
Za úkol máme rozdělit zadanou posloupnost na dvě rostoucí podposloupnosti: červenou a modrou. Nabízí se následné víceméně evidentní algoritmy. Všechny začnou se dvěma prázdnými posloupnostmi a postupně do nich budou přidávat jednotlivé prvky vstupu.

1. Na počátku prohlásíme červenou posloupnost za aktivní. Každý prvek vstupu se nejprve pokusíme přidat na konec aktivní posloupnosti, a když to nejde (už by nerostla), prohlásíme za aktivní opačnou posloupnost a přidáváme nadále tam. Pokud prvek nepůjde přidat ani do jedné posloupnosti, ohlásíme neúspěch.
2. Každý prvek se nejprve pokusíme přidat na konec červené, a nejde-li to, zkusíme to ještě na konec modré.
3. Pokud můžeme prvek přidat jen do jedné posloupnosti, uděláme to. Pokud do obou, vybereme si tu, která končí větším prvkem (prázdná posloupnost končí prvkem $-\infty$). Končí-li obě stejně, vybereme si červenou.

Všechny tři algoritmy běží v lineárním čase a mají za sebou nějakou slibnou myšlenku. Ale prozradíme vám, že právě jeden z nich nefunguje (pro některé vstupy selže), zatímco zbylé dva jsou správně. Na chvíli se zastavte a zkuste přijít na to, který je ten špatný.

Chvíle napětí… nefunkční je algoritmus 1. Doběhneme ho třeba na vstupu 1, 10, 2, 11, 9. Nejprve dá 1 a 10 do červené posloupnosti, pak 2, 11 do modré a nakonec bezradně drží v ruce 9, která se nehodí ani do jedné. Korektní rozdělení přitom existuje: 1, 2, 9 a 10, 11.

Spoléhat se na intuici se nám tedy vymstilo. Správnost zbylých dvou algoritmů radši poctivě dokážeme.



Preference první posloupnosti

Snazší to bude s algoritmem 2. Pokud uspěl, vydal určitě korektní výstup: obě posloupnosti jsou po celou dobu výpočtu rostoucí a každý prvek jsme do některé z nich umístili. Nyní ukážeme, že v případech, kdy algoritmus selže, žádné korektní rozdělení neexistuje.

Zastavme algoritmus v tom okamžiku, kdy se právě chystá oznámit neúspěch. Drží v ruce nějaký prvek x, který je menší nebo roven koncům obou posloupností: červenému konci c a modrému konci m. Pak se podívejme do minulosti na okamžik, kdy jsme přidávali prvek m. Tehdy jsme ho nedali do červené posloupnosti, což znamená, že červená končila nějakým prvkem $c' \geq m$.

Na vstupu se tudíž vyskytly (v tomto pořadí) nějaké tři prvky $c' \geq m \geq x$. Jenže ať už obarvíme vstup dvěma barvami jakkoliv, dostanou dva z těchto tří prvků stejnou barvu. To znamená, že posloupnost této barvy není rostoucí. Hotovo.

Větší bere

Konečně se podíváme na zoubek algoritmu 3. Dokážeme o něm, že vydá stejný výsledek jako algoritmus 2, jehož správnost jsme už ověřili.

V druhém algoritmu totiž platí, že červený konec je stále větší nebo roven modrému konci. Vskutku: buď nový prvek přidáváme na konec červené posloupnosti (takže červený konec ještě zvětšíme), nebo to nejde, protože nový prvek je větší nebo roven červenému konci, takže ho učiníme modrým koncem a nerovnost stále platí.

Třetí algoritmus tedy pokaždé učiní stejné rozhodnutí jako druhý.

Martin „Medvěd“ Mareš

29-4-2 Hrací automat

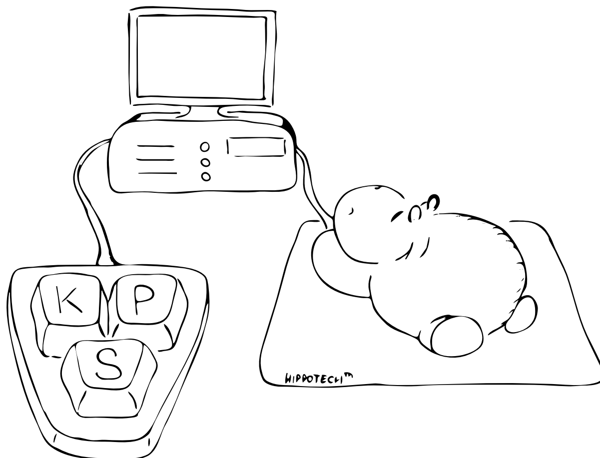
Ze všeho nejdříve si všimněme, že nezáleží na tom, že se jedná o kruhy. Celou situaci si také můžeme představit tak, že máme nějaké intervaly, přičemž u každého intervalu máme danou x-ovou souřadnici, na které míček bude pokračovat v pádu, pokud spadne na tento interval. Za každý kruh pak přidáme dva takové intervaly – jeden odpovídající levé polovině kruhu a jeden odpovídající pravé polovině kruhu.

Nyní bychom chtěli postavit datovou strukturu, která bude umět odpovídat na naše dotazy. Budeme ji stavět odspoda nahoru. Setřídíme si všechny kruhy podle y-ové souřadnice jejich středu a nyní je budeme chtít přidávat do naší datové struktury.

Abychom byli schopni rychle hledat, do kterého již vytvořeného intervalu spadá daná x-ová souřadnice, a abychom mohli intervaly průběžně měnit, bude-

me si vše ukládat do vyváženého vyhledávacího stromu.[50]

Můžeme si všimnout, že intervaly přidané do stromu budou disjunktní, takže je vždy jasné, který ze dvou intervalů je více vlevo, a můžeme je tedy jednoduše porovnávat. K intervalům si budeme také připisovat, na jaké x-ové pozici kulička vypadne, pokud na daný interval spadne.



Budeme procházet kruhy podle y-ové souřadnice od nejmenší. Nejdříve z vyhledávacího stromu odstraníme intervaly, které se celé nacházejí přímo pod aktuálně zpracovávaným kruhem. Ty, které pod něj sahají jen částečně, upravíme tak, že je zkrátíme, aby pod něj už nesahaly.

Pro každý kruh přidáme dva intervaly – jeden od jeho středu do jeho levého konce a jeden od středu do jeho pravého konce. Místa, na kterých kulička při spadnutí na tyto dva intervaly vypadne, zjistíme tak, že se rozestavěné datové strukturu zeptáme, kde míček vypadne, pokud ho vhodíme na levém, resp. pravém konci intervalu.

Dotaz nyní vypadá tak, že pomocí vyhledávacího stromu zjistíme, do kterého intervalu daný bod spadá, a vypíšeme x-ovou souřadnici, na které kulička vypadne. Tu máme předpočítanou, takže nám to bude trvat jen $\mathcal{O}(\log N)$ na práci s vyhledávacím stromem, N značí počet kruhů.

Pokud se nám stane, že zadaná x-ová souřadnice nespadá do žádného intervalu, kulička na žádný kruh nespadla a vypadne na stejném místě, na kterém jsme ji vhodili.

Při stavbě datové struktury budeme jednou třídit a uděláme $\mathcal{O}(N)$ operací s vyhledávacím stromem – každý interval totiž nejvýše jednou přidáme a nejvýše

[50] `http://ksp.mff.cuni.cz/viz/kucharky/stromy`

jednou smažeme, což obojí trvá $\mathcal{O}(\log N)$. Celkově nám tedy předzpracování bude trvat $\mathcal{O}(N \log N)$. Předpočítaná datová struktura zabere $\mathcal{O}(N)$ prostoru.

Kuba Tětek

29-4-3 Výhružné dopisy

Nejprve si uvědomíme, že v této úloze ve skutečnosti šlo jen o to, rozdělit správně zločince do dvou gangů.

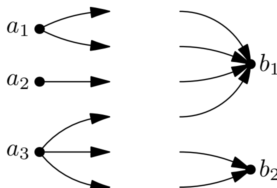
Co od takového rozdělení požadujeme? Protože každý dopis odeslaný někým z gangu A byl přijat někým z gangu B, musel gang B celkem přijmout přesně tolik dopisů, kolik jich gang A celkem odeslal. To samé musí platit v opačném směru. Takovému rozdělení budeme říkat *vyvážené*.

Zatím odložme, jak vyvážené rozdělení najít. Ale pokud bychom nějaké dostali, je už snadné vyřešit zbytek úlohy. Dopisy budeme zpracovávat pro každý směr zvlášť, nejdříve třeba od A pro B.

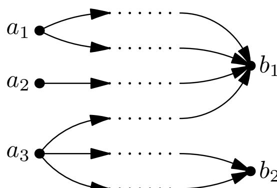
Představme si třeba následující situaci: gang A má tři členy, kteří poslali po řadě 2, 1 a 3 dopisy. Gang B má dva členy, kteří přijali 4 a 2 dopisy. Rozdělení je ve směru A→B vyvážené: tímto směrem bylo odesláno i přijato 6 dopisů.

Při sestavování výsledného multigrafu se nám bude hodit přemýšlet o *půlhranách*. Ty si lze představit tak, že jsme vzali nějakou orientovanou hranu a uprostřed ji přestřihli. Zbude výstupní půlhrana, která má počáteční vrchol, ale ne koncový, a vstupní půlhrana, jež má naopak jen koncový.

V našem případě mají vrcholy gangu A jednu výstupní půlhranu za každý odeslaný dopis, v gangu B jednu vstupní za každý přijatý:



Teď stačí utvořit celé hrany tak, že každou výstupní půlhranu spárujeme s jednou vstupní. Snadno si rozmyslíte, že to můžeme udělat naprosto libovolně a vždy získáme korektní řešení. Například hladově při průchodu vrcholy obou gangů v nějakém pořadí (zde shora dolů):



Tím dostáváme jedno možné řešení: a_1 odeslal dva dopisy b_1, a_2 jeden dopis b_1 a a_3 poslal jeden dopis b_1 a tři b_2.

Obecný algoritmus by mohl vypadat třeba takto:

1. Vložíme všechny vrcholy gangu A do fronty F_A v libovolném pořadí, analogicky pro F_B.
2. U každého vrcholu $u \in F_A$ si pamatujeme číslo $z(u)$: kolik dopisů ještě zbývá danému člověku odeslat (resp. přijmout pro $u \in F_B$). Na začátku jsou to čísla ze zadání.
3. Dokud nejsou obě fronty prázdné:
4. $a \leftarrow$ první prvek F_A, $b \leftarrow$ první prvek F_B
5. $m \leftarrow \max(z(a), z(b))$ (maximální počet dopisů, které a ještě může poslat b)
6. Vypíšeme „$a\ b\ m$“ (a poslal m dopisů b).
7. Snížíme $z(a)$ i $z(b)$ o m.
8. Pokud některá z těchto hodnot klesla na nulu (člověk už poslal všechny dopisy, které měl), vyřadíme odpovídající vrchol z fronty.

Na konci musí být všechna z nulová a každý odeslal/přijal tolik dopisů, kolik měl.

Po každém kroku odstraníme alespoň jeden vrchol z fronty, takže vše stihneme v čase $\mathcal{O}(N)$ (kde N je počet lidí). Celý postup zopakujeme pro opačný směr (dopisy od B pro A).

Hledání vyváženého rozdělení

Označme si o_i a p_i počet dopisů odeslaných, resp. přijatých i-tým člověkem. Dále si pro nějakou množinu lidí X označme $o(X) := \sum_{i \in X} o_i$ celkový počet dopisů odeslaných členy této skupiny, analogicky $p(X)$. Dále si označme V množinu úplně všech lidí ze vstupu. Aby vůbec mohlo existovat řešení, musí platit $o(V) = p(V)$, tedy celkem bylo přijato stejně dopisů jako odesláno. Tento celkový počet dopisů si označíme M.

Hledáme rozdělení lidí na dvě množiny A a B takové, že $o(A) = p(B)$ a $p(A) = o(B)$. Vzhledem k tomu, že platí $o(A) + o(B) = M$ a $p(A) + p(B) = M$, můžeme podmínku vyváženosti upravit na: $o(A) = M - p(A)$, $p(A) = M - o(A)$. Stačí najít podmnožinu A splňující tuto vlastnost. Obě tyto podmínky jsou ve skutečnosti jedna a ta samá:

$$o(A) + p(A) = M.$$

Jinými slovy, rozdělení je vyvážené právě tehdy, když celkové množství dopisů v obou směrech je pro oba gangy stejné.

Označme si proto ještě $w_i := o_i + p_i$ celkové množství dopisů odeslaných a přijatých daným člověkem a $w(X)$ součet w_i pro všechny lidi v množině X.

Hledáme takovou množinu X, pro kterou platí $w(X) = M$. To není nic jiného než dobře známý problém batohu (resp. dvou loupežníků).[51]

K řešení použijeme obvyklý algoritmus pro batoh, který je popsán v naší kuchařce o dynamickém programování.[52]

Pokud dokážeme naplnit batoh předměty o celkové váze přesně M, jim odpovídající lidé tvoří např. gang B, zbytek gang A. Pokud batoh přesně naplnit nelze, vyvážené rozdělení neexistuje.

Jaká je časová složitost? Algoritmus pro batoh potřebuje čas $\mathcal{O}$(počet předmětů · nosnost batohu), v našem případě $\mathcal{O}(N \cdot M)$. Rekonstrukce hran trvá v každém směru $\mathcal{O}(N)$. Dohromady si tedy vystačíme s $\mathcal{O}(N \cdot M)$ času a $\mathcal{O}(N + M)$ paměti.

Program (C):
`http://ksp.mff.cuni.cz/viz/29-4-3.c`

Filip Štědronský

29-4-4 Policejní síť

Tentokrát jste nám poslali mnoho zcela odlišných a povětšinou zcela správných řešení. My se tu spolu nyní na pár přístupů podíváme.

Nejprve si strom zakořeníme. Tedy vyberme si libovolný vrchol a o něm prohlásíme, že je to kořen. Poté můžeme rozdělit sousedy každého vrcholu na otce (ten soused blíž kořeni) a syny (ostatní sousedé). Všechny vrcholy až na kořen budou mít tedy jednoho otce. Konečně, jako podstrom vrcholu v budeme chápat část stromu, kde je v, jeho synové, synové jejich synů atd.

Podívejme se na nějaký důležitý vrchol. Pokud v jeho podstromu není žádný jiný důležitý vrchol, je zřejmé, že jeho spojení musí směřovat přes otce. Toto poměrně jednoduché pozorování je klíčové pro jeden přístup k řešení.

Na začátku totiž můžeme strom zbavit zbytečných větví (tj. takových podstromů, které neobsahují žádný důležitý vrchol – takovými podstromy ani nemůže vést žádné důležité spojení), takže listy (vrcholy bez synů) budou vždy důležité vrcholy. Víme, že od každého listu nyní musí vést důležité spojení přes jeho

[51] `http://ksp.mff.cuni.cz/viz/kucharky/tezke-problemy`
[52] `http://ksp.mff.cuni.cz/viz/kucharky/dynamicke-programovani`

otce, otce jeho otce atd., dokud nenarazíme na rozcestí. Takto ke každému listu nakreslíme část spojení.

Jelikož každá větev (tedy cesta k listu) je zakončena důležitým vrcholem, jistě se nám v nějakém vrcholu potkají části několika spojení. Pokud budou alespoň tři, víme, že přes toto rozcestí musí vést spojení všech těchto vrcholů. Protože ale můžeme spojit jen dva, spojení třetího by muselo procházet spojením zbylých dvou, což máme ale zakázáno. V takovémto případě tedy řešení neexistuje.

Pokud se setkají spojení dvou vrcholů, jednoduše těmito větvemi spojíme zmíněné dva vrcholy a tyto větve odstraníme. Stejně tak odstraníme i nově vzniklou větev bez důležitých vrcholů. Poté celý postup opakujeme.

Když takto postupně odstraníme všechny vrcholy, znamená to, že jsme našli spárování pro všechny důležité počítače. Všimněte si, že vždy jsme vyznačovali pouze tu část spojení, o které jsme věděli, že danými hranami vést musí. Pokud tedy řešení existuje, je jen jedno a nemá smysl hledat další.

Už toto je správný algoritmus. Jeho poměrně přímočará implementace má časovou složitost $\mathcal{O}(N^2)$. Pokud jej napíšeme šikovně, můžeme vytvořit i optimální řešení, které pracuje v čase $\mathcal{O}(N)$. My se ale společně podíváme na další řešení, které je také optimální, ale navíc se i dobře implementuje.

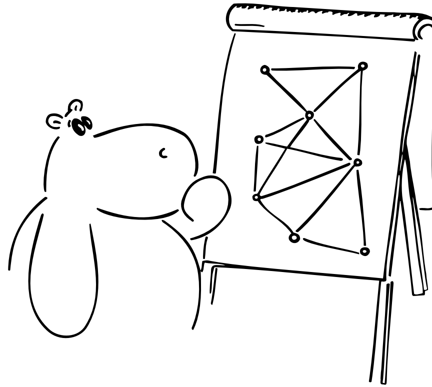
Od kořene k synům

Na problém se podíváme teď trošku opačně, místo toho abychom řešení postupně budovali, tak se posadíme na kořen a představíme si, že řešení už skoro máme. Konkrétně budeme předstírat, že známe všechna spojení, která nevedou kořenem a navíc, že víme kterými hranami sousedícími s kořenem musí vést spojení (dle pravidel z předchozího odstavce).

Dokončit toto řešení už je hračka. Pokud kořen není důležitý, stačí postupovat podle pravidel, která už známe. Pokud jsou částečná spojení dvě, spojíme je, pokud žádné, tak už jsme vlastně skončili s existujícím řešením a jinak řešení neexistuje. Jestli kořen důležitý je, tak je naopak jediný vyhovující případ, když máme pouze jedno další částečné spojení, které se spojí s kořenem. Jakýkoliv jiný případ znamená, že řešení neexistuje.

Jenže jak si zařídit abychom toto všechno věděli? Jednoduše se podíváme na všechny jeho syny a použijeme úplně stejný algoritmus – s jednou malou změnou. Pokud ze synů nějakého syna dostaneme jedno spojení, nemusíme ještě házet flintu do žita, ale můžeme doufat, že toto neúplné spojení ještě spojíme přes kořen, oznámíme tedy kořenu, že z tohoto podstromu musí vést jedno spojení.

Stejně tak v případě, že tento syn nedostal ze svých synů žádné spojení a sám je důležitým vrcholem. Všechny další případy buď znamenají, že řešení neexistuje nebo existuje a ke kořenu nevede žádné spojení.



Abychom ale algoritmus byli schopní spustit na nějakém synovi, budeme muset stejný algoritmus použít pro jeho syny, ty budou potřebovat použít algoritmus pro své syny a tak do nekonečna… nebo alespoň do té doby než dojdeme k listům.

U listů už nepotřebujeme nic vypočítávat pro jejich syny (ani žádné nemají), ale požadovaná odpověď je snadná. V samotném listu nic nespojíme, takže nás zajímá pouze to, zda vede od tohoto listu výš nějaké spojení. A to přímo odpovídá tomu, jestli je list důležitým vrcholem, či nikoliv.

Dostali jsme tedy pěkné rekurzivní řešení. Jelikož řešení na každém vrcholu stráví konstantně mnoho času (práci počítání u vrcholu připočítáme jeho synům), tak nám vychází celková složitost lineární.

Program (Python 3):
`http://ksp.mff.cuni.cz/viz/29-4-4.py`

Janka Bátoryová & Dominik Smrž

29-4-5 Chybějící spisek

Rozmysleme si, že úloha vyhledat první chybějící číslo je ekvivalentní s problémem, kde chceme najít v posloupnosti největší interval čísel $[0, k-1]$ takový, že žádné číslo v tomto intervalu nechybí. První chybějící číslo poté bude k.

Dále si všimněme, že umíme zjistit použitím pouze konstantního množství paměti, zda se v seznamu vyskytuje každé číslo z intervalu $[a, b]$. Stačí lineárně projít seznam, za každé relevantní nalezené číslo přičíst výskyt a nakonec porovnat výsledek s $b - a + 1$. Nám bude stačit $a = 0$.

Nechť $f(l)$ odpovídá na otázku, zda seznam obsahuje všechna čísla v intervalu $[0, l]$. Potom tato funkce vypadá tak, že $f(l) = 1$ pro $l = 0, \ldots, k-1$ a pro $l = k, \ldots, n$ je již $f(l) = 0$. Díky této pěkné vlastnosti můžeme k binárně vyhledat.

Jestliže víme, že k se nachází v intervalu $[a, b]$, umíme tento interval upřesnit. Nechť $c = \frac{a+b}{2}$, pak se rozhodneme podle $f(c)$:

- $f(c) = 1$, tedy v intervalu $[0, c]$ jsou všechna čísla. Potom k musí být v intervalu $[c + 1, b]$.
- $f(c) = 0$, v intervalu $[0, c]$ něco chybí. Tedy k najdeme v intervalu $[a, c]$.

Takto redukujeme interval $[a, b]$ až dokud nedojdeme k rovnosti a, b. V takovém případě již s jistotou víme, že k je přesně a (nebo b).

Dále si rozmysleme, jaké nejvyšší číslo může chybět, pokud máme n-prvkový seznam. V případě, že žádné číslo v seznamu nechybí, obsahuje každé „hlavní“ číslo z $[0, n-1]$. Pokud v této posloupnosti najdeme čísla jiná, potom určitě nějaké „hlavní“ číslo chybí. Toto nám dává horní odhad na hodnotu chybějícího čísla.

Samotný algoritmus tedy na začátek projde seznam a spočítá si počet prvků $n+1$. Nejprve zkontroluje, zda vůbec nějaké číslo chybí tím, že spočítá $f(n)$. Poté použitím iterace binárně vyhledá k počínaje intervalem $[0, n]$.

Časovou složitost není těžké spočítat. Každý výpočet $f(l)$ trvá $\mathcal{O}(n)$ času. Každý krok binárního vyhledávání zmenší možný interval o polovinu, nejvýše tedy provede $\mathcal{O}(\log n)$ kroků. Celková časová složitost algoritmu činí $\mathcal{O}(n \log n)$.

Nyní už jen ukážeme, že paměťová složitost je konstantní. Již víme, že $f(l)$ na odpověď postačí konstantní paměť. U binárního vyhledávání si stačí pamatovat interval $[a, b]$ a výsledek $f(c)$. Jen je třeba si dát pozor, že použití rekurze na půlení intervalu by spotřebovalo část zásobníku pro každé zavolání funkce a složitost by vzrostla na $\mathcal{O}(\log n)$. My jsme však použili iteraci, kde tento problém není, a tedy paměťová složitost je skutečně $\mathcal{O}(1)$.

Program (Python 3):

`http://ksp.mff.cuni.cz/viz/29-4-5.py`

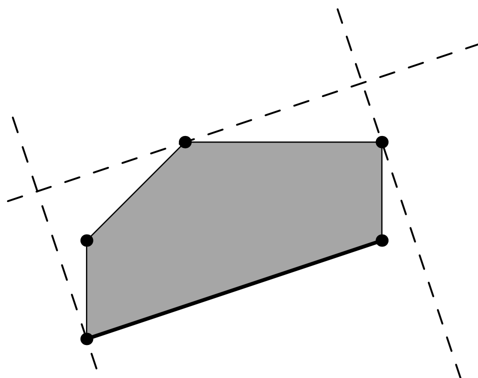
Václav Končický

29-4-6 Nové sídlo

Když řešíme úlohu, u které pořádně nevíme, jak na to půjdeme, osvědčilo se již mnohokrát rozmyslet si nejprve to úplně nejpomalejší přímočaré řešení, které nás napadne.

Zadání nám dává jeden záchytný bod – aspoň jedna hrana mnohoúhelníkového sídla musí ležet přímo na hranici pozemku. Zkusíme tedy postupně všechny hrany, pro každou z nich si na chvíli představíme, že právě ona je tou hraniční, a najdeme příslušný mnohoúhelníku opsaný obdélník.

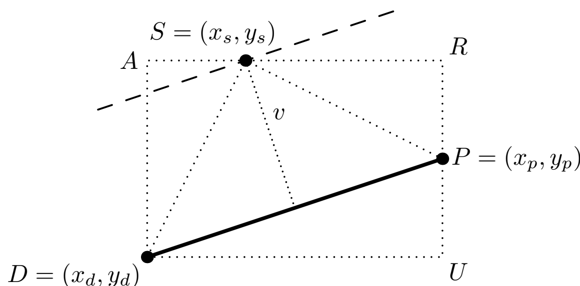
Ze všech takto nalezených opsaných obdélníků pak vybereme ten nejmenší. U mnohoúhelníka majícího $\mathcal{O}(M)$ hran bude celý algoritmus trvat $\mathcal{O}(M \cdot \phi(M))$, kde $\phi(M)$ je složitost vyhledání opsaného obdélníka.



Odbočíme tedy a vymyslíme, jak najít mnohoúhelníku opsaný obdélník, víme-li, která jedna jeho hrana je na hranici pozemku. Konkrétně hledáme tři přímky, které se mnohoúhelníku dotýkají, přičemž jedna z nich je rovnoběžná se zadanou hranou a dvě další jsou kolmé.

Nechť zadaná hrana vede z vrcholu D ležícího na souřadnicích (x_d, y_d) do vrcholu $P = (x_p, y_p)$.

Nejprve najdeme rovnoběžku, resp. stačí nám vzdálenost té rovnoběžky od zadané hrany (hledáme maximum). Na papíře se rovnoběžka vede snadno, pokud vám zrovna neujede ruka, ale jak na to v počítači?



Vzdálenost bodu S na souřadnicích (x_s, y_s) od přímky se měří na kolmici (tečkovaně), což je zároveň výška v trojúhelníku DPS. Pro tu známe například vztah pro obsah trojúhelníka $S(DPS) = \frac{v \cdot |DP|}{2}$, přičemž dokážeme jednoduše spočítat obsahy okolních trojúhelníků DAS, SRP a PUD, stejně jako obdélníka $RUDA$.

Zjevně platí, že

$$S(DPS) + S(DAS) + S(SRP) + S(PUD) = S(RUDA)$$

Obsahy vyjádříme pomocí souřadnic bodů D, P a S:

$$\frac{v \cdot |DP|}{2} + \frac{(x_s - x_d)(y_s - y_d)}{2} + \frac{(x_p - x_s)(y_s - y_p)}{2} + \frac{(x_p - x_d)(y_p - y_d)}{2} = (x_p - x_d)(y_s - y_d)$$

$$v \cdot |DP| = x_d(y_p - y_s) + x_p(y_s - y_d) + x_s(y_d - y_p)$$

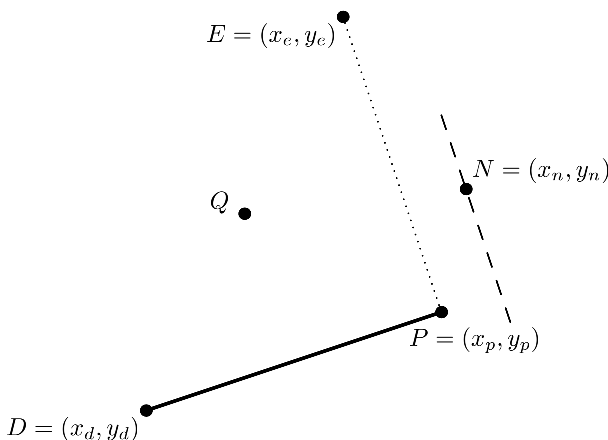
Vzdálenost v může vyjít kladná nebo záporná; vyjadřuje, jestli je bod S vlevo nebo vpravo od přímky DP. Na laskavém čtenáři ponecháváme, aby se předvědčil, že tentýž vzorec je možné aplikovat i pro jiné vzájemné polohy bodů D, U, P, R, S a A.

Konstantní vzdálenost $|DP|$, kterou umíme spočítat Pythagorovou větou, zatím ponecháme nevyjádřenou, neboť může vyjít iracionální (narozdíl od čitatele, jehož hodnota je celočíselná, neboť všechny souřadnice na vstupu jsou taktéž celá čísla). Kvůli přesnosti je výhodné počítat co nejdéle s celými čísly.

Projdeme tedy všechny vrcholy mnohoúhelníka a pro každý z nich si poznamenáme, jak je daleko od přímky DP. Tím nejvzdálenějším vede rovnoběžná hrana opsaného obdélníka, který hledáme; označíme si jej Q.

Nyní hledáme další dva vrcholy mnohoúhelníka, kterými budou procházet kolmé hrany opsaného obdélníka.

Za tímto účelem si pořídíme bod E jako otočení bodu D kolem bodu P o 90° a budeme počítat vzdálenosti všech vrcholů mnohoúhelníka od přímky PE. Bod, jehož vzdálenost od přímky PE právě počítáme, si označíme N.



$$x_e = x_p - (y_p - y_d); \quad y_e = y_p + (x_p - x_d)$$

KSP

řešení

Ve výše uvedeném vztahu pro $v \cdot |DP|$ nahradíme body D a S za body E a N a dostaneme:[53]

$$v' \cdot |DP| = x_e(y_p - y_n) + x_p(y_n - y_e) + x_n(y_e - y_p)$$

Po dosazení a algebraických úpravách dostaneme jednoduchý vztah pro (orientovanou) vzdálenost bodu N od přímky EP:

$$v' \cdot |DP| = (x_d - x_p)(x_p - x_n) + (y_d - y_p)(y_p - y_n)$$

Najdeme-li tedy minimum a maximum této hodnoty, dostaneme vrcholy, kterými prochází dvě kolmé hrany opsaného obdélníka.

Zbývá spočítat obsah tohoto obdélníka:

$$S = v(v'_{\max} - v'_{\min})$$

My sice nemáme uložené v a v', ale jen jejich součiny s $|DP|$, ale to nevadí; když použijeme tyto součiny místo v a v', dostaneme $S \cdot |DP|^2$, což je celé číslo, stejně jako $|DP|^2$. Víme tedy, že S je racionální číslo (podíl dvou celých čísel).

Pokud chceme počítat ultra přesně, uložíme si obě čísla zvlášť, tedy místo S si uložíme dvojici $(S \cdot |DP|^2, |DP|^2)$, a při hledání nejmenšího opsaného obdélníka pak můžeme porovnávat zlomky $S = \frac{S \cdot |DP|^2}{|DP|^2}$ algoritmem pro přesné porovnávání racionálních čísel.[54]

Jak dlouho trvá najít takový obdélník? Spočítat vzdálenost zabere konstantní čas, to budeme činit dvakrát pro každý bod (jednou hledáme rovnoběžku, podruhé kolmici) a ze vzdáleností budeme vybírat maxima a minima, celkem tedy $\phi(M) = \mathcal{O}(M)$.

Tento přímočarý algoritmus nám tedy zabere pro celý mnohoúhelník $\mathcal{O}(M^2)$ času. Při rozumné implementaci hledání maxim a minim nám postačí konstantní množství paměti navíc, tedy $\mathcal{O}(M)$ včetně uložení vstupu.

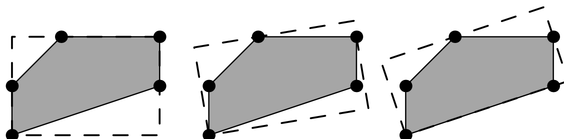
Takový algoritmus avšak není nejrychlejší. Především si můžeme všimnout, že při hledání rovnoběžky vzdálenost vrcholu nejprve roste a pak klesá; při hledání kolmic nejprve roste, pak klesá a pak zase roste. Drobnou úpravou binárního vyhledávání dokážeme zrychlit vyhledání kolmic a rovnoběžky na $\mathcal{O}(\log M)$; celý algoritmus tedy stihneme v čase $\mathcal{O}(M \log M)$.

Jde to však ještě rychleji; použijeme metodu známou v angličtině jako Rotating Calipers, česky se to nedá smysluplně přeložit, možná jako „otáčení svěrákem“.

---

[53] Tiše též využíváme skutečnosti, že $|DP| = |EP|$.

[54] Platí, že $\frac{p}{q} > \frac{r}{s} \Leftrightarrow ps > rq$, pokud $q > 0$, $s > 0$.



Nejprve si tedy zvolíme jednu hranu mnohoúhelníka a najdeme pro ni příslušný opsaný obdélník.

Pak si pro každý ze čtyř bodů/hran dotyku najdeme následující hranu, což jsou právě ty hrany, ke kterým se přitiskne čelist našeho obdélníkového svěráku při otáčení. Vybereme si tu nejbližší, což určíme podle úhlu, který svírá s blízkou čelistí.

Ta hrana, která má nejmenší úhel, se totiž bude dotýkat čelisti svěráku v následujícím kroku. Ostatní body dotyku budou stále body dotyku; tam, kde se dotýkala čelist hrany, bude bodem dotyku „ten druhý vrchol", čili ten pozdější v seznamu vrcholů na vstupu.

Aby se totiž mohl svěrák přesunout z jednoho vrcholu na další bod dotyku, musí se nejdříve dotknout hrany mezi těmito dvěma vrcholy. Tímto postupem tedy zajistíme, že svěrák postupně projde všechny hrany, a to sice ne v pořadí, ve kterém jsou zadané na vstupu, ale v pořadí podle jejich směru (sklonu).

Jakmile se dostaneme s čelistmi svěráku zase k první hraně, jsme nutně hotovi, můžeme vybrat minimum a ohlásit výsledek.

Všímavý řešitel si všimne, že takto se celý pomyslný svěrák otočí za celou dobu jenom o čtvrtkruh, neboť procházíme obvod mnohoúhelníka zároveň na čtyřech místech.

Toto řešení má časovou složitost $\mathcal{O}(M)$; musíme na začátku v $\mathcal{O}(M)$ najít první opsaný obdélník a pak nám na každý krok svěráku stačí konstantní množství času; kroků je také $\mathcal{O}(M)$, neboť na každou hranu sáhneme právě jednou.

Do paměti si neukládáme téměř nic, stačí pár pomocných proměnných. K tomu musíme započítat velikost vstupu, neboť jej neumíme zpracovat proudově, tedy $\mathcal{O}(M)$.

Jan „Moskyto" Matějka

29-4-7 Rozebíráme stromy

Úkol 1: Odlišná definice

Má-li těžká hrana vést do nadpolovičně velikého podstromu namísto největšího podstromu, nic podstatného se nezmění.

Nově se sice může stát, že všechny hrany vedoucí z vrcholu dolů jsou lehké (to se stane třeba v úplném binárním stromu). Platí ovšem stále, že dolů vede nejvýše jedna těžká hrana, takže těžké hrany tvoří vrcholově disjunktní cesty.

A velikosti podstromů směrem dolů exponenciálně klesají, takže lehká hloubka opět vyjde logaritmická.

Alternativní definice je tedy stejně dobrá, jako ta původní.

Úkol 2: Vzdálenost vrcholů

Nejprve nalezneme nejbližšího společného předka ℓ zadaných vrcholů x a y. Pak pro vzdálenosti vrcholů platí

$$d(x, y) = d(x, \ell) + d(\ell, y).$$

Stačí tedy umět počítat vzdálenosti na „svislých“ cestách.

V dekompozici stromu se cesta z (řekněme) x do ℓ skládá z maximálně logaritmicky mnoha lehkých hran a částí těžkých cest. Lehké hrany ošetříme … inu, lehce: přispívají ke vzdálenosti jedničkou. Těžké cesty nejsou o moc pracnější: pokud jsme na nějakou vstoupili ve vrcholu a a vystoupili v b, prošli jsme přesně $index(b) - index(a)$ hran.

Postačí tedy projít dekompozicí zdola nahoru a posčítat $\mathcal{O}(\log n)$ hodnot. Ani nepotřebujeme předpočítávat nic dalšího.

Úkol 3: Rychlejší cestová minima

Výsledek dotazu skládáme z lehkých hran (těch je logaritmicky a každou z nich zpracujeme v konstantním čase) a částí těžkých cest (těch je také logaritmicky mnoho, ale pro každou z nich jsme se potřebovali zeptat intervalového stromu, což trvalo rovněž logaritmicky).

Stačí si ale uvědomit, že cestujeme-li stromem zdola nahoru, neptáme se na obecné části cest, ale na *suffixy:* části do vstupního vrcholu až na konec cesty (čili do jejího nejvyššího bodu). Jedinou výjimkou je poslední navštívená cesta, tedy ta, na níž leží LCA – tam už je to opravdu obecný interval.

Kromě intervalových stromů si předpočítáme ještě suffixové součty. Pak každý suffix cesty vyhodnotíme v konstantním čase a ten jediný obecný interval v $\mathcal{O}(\log n)$. Celkem tím strávíme čas $\mathcal{O}(\log n \cdot 1 + 1 \cdot \log n) = \mathcal{O}(\log n)$. Předvýpočet jsme asymptoticky nezpomalili – prefixové součty si hravě pořídíme v čase $\mathcal{O}(n)$.

Úkol 4: Jak se změní kostra?

Ukážeme, že zadaný úkol lze přímočaře převést na hledání cestových maxim, které zvládneme v čase $\mathcal{O}(n)$ na předvýpočet a $\mathcal{O}(\log n)$ na dotaz použitím HLD s optimalizací podle předchozího úkolu.

Mějme neorientovaný graf se zadanými *vahami* hran a nějakou jeho minimální kostru M. Uvažme nějakou hranu xy váhy $w(xy)$, která neleží v minimální kostře. Vrcholy x a y jsou spojené nějakou cestou P v kostře, označme pq nejtěžší hranu této cesty a $w(pq)$ její váhu.

Nejprve nahlédneme, že $w(pq) \leq w(xy)$. Pokud by totiž hrana xy byla lehčí než pq, můžeme do kostry přidat xy a smazat pq. Tím nejprve vznikne z cesty P

kružnice a pak se z ní opět stane cesta. Dostaneme tedy nějakou jinou kostru. Ta je ovšem lehčí než původní minimální kostra, což je spor.

Takže pokud $w(xy)$ snížíme pod $w(pq)$, minimální kostra se určitě změní. Zbývá nahlédnout, že pokud ji snížíme na cokoliv mezi $w(pq)$ a $w(xy)$, bude zadaná kostra M stále minimální.

Spustíme Kruskalův algoritmus (viz kuchařka o minimálních kostrách)[55] s původními vahami. Až bude třídit hrany podle vah, srovnáme hrany stejné váhy tak, aby nejprve šly ty, které leží v M, a po nich všechny ostatní. Nahlédneme, že najde právě naši kostru M.

Nyní snížíme váhu hrany xy a spustíme algoritmus znovu. V okamžiku, kdy se dostane k hraně xy, bude už celá cesta P součástí kostry (všechny její hrany jsou v uvažovaném pořadí hran před xy). Hranu xy tedy nepřidáme, protože by vytvořila kružnici. Vyjde tedy stejná minimální kostra jako předtím.

Martin „Medvěd" Mareš

[55] `http://ksp.mff.cuni.cz/viz/kucharky/minimalni-kostry`

Pátá série

29-5-1 Holubí pošta

Cílem této úlohy bylo nalézt nejkratší cestu pro zaslání zprávy holubí poštou. Ale aby nebylo hledání nejkratší cesty tak jednoduché, mohlo se stát, že v některých vrcholech budeme muset chvíli počkat, než otevřou poštovní stanici a pošlou holuba dál.

Pokud by úloha neobsahovala čekání ve vrcholech, stačil by na vyřešení úplně klasický Dijkstrův algoritmus, o kterém máme dokonce i kuchařku.[56] Čekání nám však úlohu zkomplikuje jen drobně.

Dijkstrův algoritmus je postavený na myšlence, že si držíme seznam otevřených vrcholů a u každého otevřeného vrcholu máme poznamenanou délku nejkratší cesty, kterou se do něj umíme dostat. Na počátku obsahuje tento seznam pouze startovní vrchol (se vzdáleností nula) a všechny ostatní vrcholy mají vzdálenost nastavenou na nekonečno.

V každém kroku ze seznamu otevřených vrcholů vezmeme takový, do kterého se umíme dostat nejkratší cestou. Pokud jsou všechny hrany v grafu nezáporné, tak víme, že do tohoto vrcholu se už kratší cestou ze žádného jiného otevřeného vrcholu nedostaneme, a můžeme ho tedy prohlásit za finální a *uzavřít*.

Při uzavírání vrcholu se podíváme na všechny jeho sousedy a pokud se do některého z nich umíme dostat kratší cestou (tedy délky cesty do uzavíraného vrcholu plus délka hrany bude menší, než vzdálenost poznamenaná v sousedovi), tak vzdálenost v sousedovi aktualizujeme.

Správnost tohoto postup je dána tím, že do uzavíraného vrcholu se už nemůžeme dostat jinou kratší cestou, což už jsme si ukázali výše. Nyní pojďme Dijkstrův algoritmus lehce modifikovat pro náš případ a pak si opět dokážeme správnost takto upraveného algoritmu.

Budeme potřebovat umět zjistit, jestli jsme do města přiletěli v otevírací dobu pošty a pokud ne, tak zjistit, kolik hodin zbývá do jejího nejbližšího otevření. Jelikož jsou otevírací doby pravidelné, tak nám stačí jenom odečíst offset, spočítat zbytek po dělení periodou a vyjde nám, v jakém čase periody jsme dorazili. Z toho už lehce vyvodíme, jestli je otevřeno, nebo musíme počkat.

Pak budeme postupovat jako v Dijkstrově algoritmu – pořídíme si minimovou haldu, do které na začátku vložíme start s časem odletu nula. V každém kroku pak vezmeme nejmenší vrchol z haldy a pro každého souseda spočítáme čas, ve kterém do souseda přiletíme, a čas, ve kterém budeme moci ze souseda odletět dál (pokud přiletíme v otevírací době, tak budou časy stejné, jinak bude čas odletu v okamžiku nejbližšího dalšího otevření pošty).

[56] `http://ksp.mff.cuni.cz/viz/kucharky/halda-a-cesty`

Takto upravený Dijkstrův algoritmus bude stále fungovat – pokud vezmeme otevřený vrchol s nejmenším časem odletu, tak se do něj už ze žádného jiného otevřeného vrcholu nedostaneme s menším časem.

Ještě nám zbývají dvě poslední drobnosti: u vrcholů si musíme pamatovat i čas příletu (to je důležité u cílového vrcholu, abychom pak správně nalezli nejrychlejší cestu) a také to, odkud jsme do každého vrcholu přiletěli s nejmenším časem pro rekonstrukci nejkratší cesty. Na detaily implementace se můžete podívat do našeho vzorového řešení.

Program (C):
`http://ksp.mff.cuni.cz/viz/29-5-1.c`

Jirka Setnička

29-5-2 Odcyklení zámku

Pomalé řešení

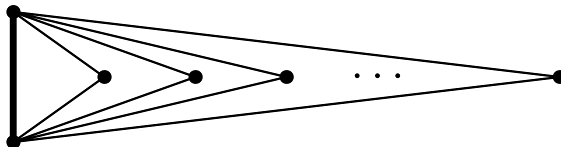
Nabízí se řešit úlohu přímočaře: najdeme nějaký cyklus, vybereme z něj nejtenčí hranu, tu odstraníme. To celé opakujeme, dokud v grafu nějaké cykly jsou.

Pro hledání cyklu se nám bude hodit prohledávání do hloubky. Platí, že v neorientovaném grafu existuje cyklus právě, když DFS najde zpětnou hranu – tedy hranu vedoucí do již dříve navštíveného vrcholu (ale ne toho, ze kterého jsme právě přišli). Podrobnosti o klasifikaci hran pomocí DFS na stromové a zpětné najdete v naší grafové kuchařce.[57]

Pokud si u každého vrcholu pamatujeme, odkud jsme do něj přišli (tedy rodiče v DFS stromě), snadno celý cyklus obejdeme, najdeme nejtenčí hranu a odstraníme ji. Odstraněním hrany (pokud nebyla zpětná) ale porušíme strukturu DFS stromu, takže když chceme hledat další cyklus, musíme provést DFS znovu od začátku.

Jedno DFS trvá čas $\mathcal{O}(N + M)$. Kolikrát ho budeme provádět? Určitě maximálně M-krát, protože v každém kroku odstraníme jednu hranu. Ale může to být opravdu tolik?

Ano, uvažte například následující graf (hrana vlevo má tloušťku 3, ostatní 1):



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

V každém kroku odebereme jednu hranu tloušťky 1 a zbavíme se tak jednoho trojúhelníku. Celkem uděláme $(M-1)/2 = \Theta(M)$ kroků. Celková časová složitost je tedy opravdu $\Theta(M(M+N)) = \Theta(M^2 + MN)$.

Rychlejší řešení

Předchozí řešení se asi nedá nijak snadno přímo zrychlit. Pokud chceme dosáhnout lepší složitosti, musíme opustit odstraňování cyklů po jednom a podívat se na problém celistvěji.

Předpokládejme pro jednoduchost, že graf je souvislý. Chceme z něj postupně odstranit všechny cykly, to znamená, že na konci nám zbude strom. Potenciálně by to mohl být i les, ale to se nestane, protože odstraněním hrany ležící na cyklu nelze porušit souvislost grafu (rozmyslete si).

Dostáváme tedy strom propojující všechny vrcholy původního grafu, neboli jeho kostru.[58]

A jelikož se celou dobu snažíme odstraňovat nejtenčí možné hrany, na kostru zbudou ty tlustší. V tuto chvíli si můžeme odvážně tipnout, že výsledná kostra bude maximální: tedy s největším možným součtem tlouštěk hran (v kostrové terminologii jim obvykle spíš říkáme *váhy*) mezi všemi kostrami.

Odtud se rýsuje algoritmus: nejprve najdeme maximální kostru. Většina známých algoritmů hledá kostru minimální, ale jednoduše je k tomuto účelu upravíme. Například u Kruskalova algoritmu popsaného v kuchařce stačí na začátku setřídit váhy sestupně místo vzestupně. Jiné algoritmy lze též snadno upravit, například tak, že všechny váhy vynásobíme -1, nebo prostě v algoritmu obrátíme všechna porovnání vah.

Potom dráty k odpojení jsou právě ty hrany, které nepatří do nalezené maximální kostry. Můžeme je dokonce odpojit v libovolném pořadí.

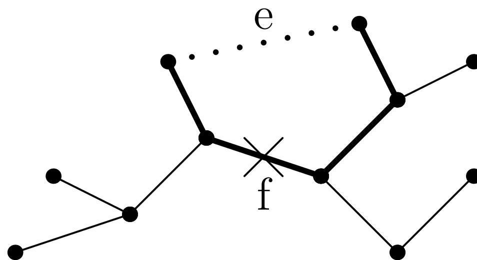
Proč to funguje?

Uvažujme libovolnou hranu e nepatřící do maximální kostry. Ta spolu s příslušnou částí kostry uzavírá cyklus (může ležet i na dalších cyklech, ale ty nás nezajímají). Ukážeme, že má na tomto cyklu minimální váhu.

Kdyby na témže cyklu existovala hrana f s menší vahou, můžeme z kostry odstranit f a přidat místo ní e:

---

[58] `http://ksp.mff.cuni.cz/viz/kucharky/kostry`



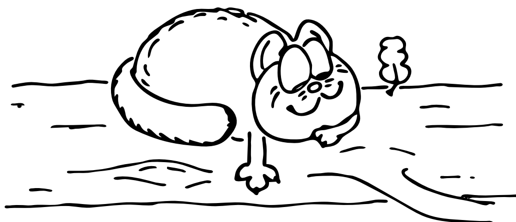
Tím bychom dostali novou kostru s větší celkovou vahou, což nemůžeme, protože původní kostra byla maximální. Tedy odpojovaná hrana e musí být nejlehčí na tučně vyznačeném cyklu.

A protože celý zbytek cyklu patří do kostry, a tedy zůstane v grafu až do konce, bude v době odpojování tento cyklus určitě ještě existovat. Tedy skutečně odpojujeme nejtenčí hranu na nějakém cyklu a každé takovéto odpojení je korektní.

Tím máme hotovo. Kostru nalezneme v čase $\mathcal{O}(M \log N)$, zbytek algoritmu zvládneme v lineárním čase.

Program (Python 3):
`http://ksp.mff.cuni.cz/viz/29-5-2.py`

Filip Štědronský



29-5-3 Sérum pravdy

Nejprve si můžeme uvědomit, že vzhledem k tomu, že všechna množství kapek jsou v lahvičkách nezáporná, při zvětšení počtu použitých lahviček nikdy neklesne součet všech kapek.

Jelikož ze všech lahviček vybíráme souvislý úsek, můžeme si jej pamatovat pomocí dvou indexů a, b tak, že lahvičky vybrané mají index i, kde $a \leq i \leq b$. Jestliže indexy se navzájem rovnají, máme prázdný úsek.

Postupujme ve hledání nejbližšího součtu následovně:

Na počátku nechť $a = b = 0$. Dále mějme v každém kroku vybraný úsek a k němu indexy a, b a součet kapek S. V případě, že rozdíl $|S - K|$ je zatím

nejmenší, co jsme potkali, zapamatujeme si jej včetně indexů a, b. Poté se podíváme na vztah S a K. Mohou nastat tři možnosti:

- $S = K$. Potom jsme našli optimální řešení a můžeme skončit.
- $S < K$. Potom je zbytečné se snažit součet zmenšit, přidáme tedy první lahvičku za úsekem do něj, tudíž b se zvýší o 1.

  V případě, že před zvýšením byl b roven počtu prvků, nemáme se kam posunout dále, a proto skončíme.
- $S > K$. Analogicky, v tomto případě chceme součet kapek zmenšit, proto první prvek z úseku odstraníme. Tudíž a se zvýší o 1.

  Rozmysleme si, že a nemůže nikdy předčit b. Jakmile $a = b$, je $S = 0$, a proto nemůže pro nezáporné K tato situace nastat.

V případě, že jsme skončili, vypíšeme nejlepší možné nalezené řešení. Všimněme si, že algoritmus funguje správně – prozkoumali jsme všechny možnosti a, b, které měly součet S dostatečně blízko K.

Jak si efektivně pamatovat aktuální součet S? Jedna možnost je použít prefixové součty. Poté umíme odpovědět konstantním časem na součet úseku.

Existuje však způsob, jenž nepotřebuje lineární množství paměti, ale stále umí součet udržovat v konstantním čase. Na začátku je určitě $S = 0$. Navíc při jednom kroku buď právě jednu lahvičku přidáme nebo odebereme. Tudíž, jestliže lahvičku přidáváme, její počet kapek přičteme k S. Podobně v případě odebírání zase její počet kapek od S odečteme.

Jakou má tento algoritmus časovou složitost? Už jsme nahlédli, že součet daného úseku umíme počítat v konstantním čase. Dále každou lahvičku navštívíme nejvýše dvakrát – jednou, když ji přidáváme do úseku, a podruhé, když ji z úseku odebíráme. Celková složitost je tedy $\mathcal{O}(N)$ vzhledem k počtu lahviček.

Program (Python 3):

`http://ksp.mff.cuni.cz/viz/29-5-3.py`

Vašek Končický

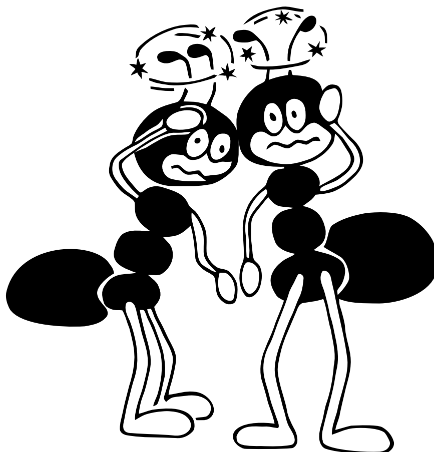
29-5-4 Rotující čepele

Příklad vyřešíme prohledáváním do hloubky. Pokud bychom se podívali na všechny bezpečné časy, budou tvořit sjednocení (byť možná prázdné) intervalů rychlostí. My budeme postupně přidávat čepele a při tom si přepočítávat interval B_k, což bude nejlevější interval, ve kterém bezpečně projedeme prvními k čepelemi a ve kterém by mohlo potenciálně ležet řešení.

Když přidáváme $(k + 1)$-ní čepel (tedy počítáme interval B_{k+1}), potřebujeme najít nejpomalejší bezpečný rychlostní interval $(k + 1)$-ní čepele, který má neprázdný průnik s B_k. Minimální rychlost intervalu získáme tak, že spočítáme,

kolikrát se daná čepel otočila do začátku B_k, toto číslo zaokrouhlíme nahoru a z tohoto čísla spočítáme čas, kdy se tak stalo.

Nejvyšší rychlost intervalu pak spočteme tak, že minimální rychlost přepočteme na čas, kdy bychom ke $(k+1)$-ní čepeli dojeli, přidáme polovinu její periody a tento čas přepočteme na odpovídající rychlost. Může se nám stát, že takový interval neexistuje. V takovém případě tudy cesta nevede a musíme se vrátit. Proto si budeme udržovat všechny intervaly na zásobníku (prohledáváme do hloubky). Budeme si tam pro každou čepel udržovat bezpečný interval pro prvních k čepelí i interval samotné k-té čepele.



Když odebereme k-tou čepel ze zásobníku, budeme chtít pokračovat prohledávání dalším intervalem k-té čepele. Ten můžeme jednoduše spočítat tak, že spočteme časy odpovídající krajním rychlostem intervalu, přičteme k nim periodu k-té čepele a to přepočítáme zpět na rychlosti. Pak spočítáme průnik tohoto intervalu s prohledávaným bezpečným intervalem prvních $k-1$ čepelí (ten najdeme na vrchu zásobníku). Může se stát, že i tento průnik bude prázdný, v kterémžto případě budeme pokračovat v odebírání ze zásobníku.

Na každý prvek na zásobníku nám stačí $\mathcal{O}(1)$ buněk paměti a zásobník je vysoký nejvýše n, kde n je počet čepelí. Potřebujeme tedy $\mathcal{O}(n)$ paměti. Libovolným intervalem libovolné čepele projdeme nejvýše jednou, takže celý algoritmus bude mít časovou složitost $\mathcal{O}(n \times č_{avg})$, kde $č_{avg}$ je průměrný počet intervalů čepele.

Kolik takový počet intervalů může být? Tuto hodnotu spočteme pro jednu čepel, $č_{avg}$ pak bude průměrem těchto hodnot. Nejdříve spočteme rozsah časů, ve kterých se umíme k čepeli dostat. Označme si minimální a maximální rychlost vozíku v_{min} a v_{max} respektive a d_i vzdálenost i-té čepele a p_i délku její periody. Pak bude interval, kdy bychom se uměli k i-té čepeli dostat (kdyby nebyly žádné

další čepele) $[v_{min} * d_i, v_{max} * d_i]$ a jeho délka je $v_{max} * d_i - v_{min} * d_i$. Čepel se za tuto dobu otočí $(v_{max} * d_i - v_{min} * d_i)/p_i$-krát a stejný bude i počet bezpečných intervalů této čepele.

Kuba Tětek

29-5-5 Zašifrovaný text

Nejprve obecněji k vašim řešením – častokrát se objevilo, že jste si označili délku věty např. písmenem V a složitost pak měřili vzhledem k V, nebo dokonce $\sqrt{V}$. Pozor, V může být asymptoticky stejně velké jako zašifrovaný text. Jeho délku si označíme jako N.

Občas jste si pak nerovnost ze zadání otočili – platí $K \leq \sqrt{V}$, ne naopak. Tedy speciálně neplatí, že $\mathcal{O}(\sqrt{V}) \subseteq \mathcal{O}(K)$, což jste se snažili občas použít. My se pokusíme vyjadřování složitosti vzhledem k V vyhnout, nicméně označení V pro délku klíče si vypůjčíme.

Nejprve se pro zjednodušení naučíme počítat se znaky. Aritmetické operace budeme totiž provádět rovnou s nimi. Prohlasme, že $\texttt{a} = 0, \texttt{b} = 1, \ldots, \texttt{z} = 25$, a všechny operace se chovají stejně, jako bychom je provedli s přiřazenými čísly – avšak modulo 26. Například $\texttt{a} - \texttt{b} = \texttt{z}$.

Nyní k věci. Napřed chvíli předpokládejme, že známe délku klíče K. Navíc předpokládejme, že $V > 1$, protože jinak by i K bylo rovno jedné a známé písmeno by šlo najít kdekoli.

Z definice Vigenèrovy šifry víme, že písmena vzdálená K od sebe jsou zašifrována stejným znakem klíče. To ovšem znamená, že rozdíl znaků v zašifrovaném textu, které jsou od sebe vzdálené právě K, na klíči vůbec nezávisí.

Přepočítáme si tedy jak vstupní text, tak známou větu tak, že od i-tého písmene odečteme $(i + K)$-té. Posledních K písmen zahodíme. Můžeme si to dovolit, K je asymptoticky menší než $\sqrt{V}$, tedy se nám velikosti vstupních dat nezmění řádově.

Nyní máme dvě posloupnosti rozdílů, které na klíči nezávisí. Speciálně to znamená, že upravený vstupní text obsahuje upravenou známou větu jako podposloupnost. Spustíme tedy obyčejný algoritmus na vyhledání jehly v seně, například KMP. Jeho výsledkem bude pozice p známé věty ve vstupním textu.

Když máme pozici, můžeme se zase vrátit k původním textům a známou větu na pozici p od textu odečíst. Vyjde nám nějaké opakování klíče. Pokud ale pozice p není zrovna dělitelná K, bude klíč nějak posunutý – to musíme napravit. Například tak, že vezmeme dva nejbližší násobky K vyšší nebo rovny p, a přečteme si klíč mezi těmito pozicemi.

Přepočítání i hledání pomocí KMP stihneme v čase lineárním k délce vstupu, tedy $\mathcal{O}(N)$. Předpokládáme, že známá věta není delší než vstup, protože pak by se zjevně nemohla v dešifrovaném vstupu vyskytovat.

Jak ale zjistit délku klíče, jejíž znalost jsme předpokládali? Prostě vyzkoušíme postupně všechny. Potom se samozřejmě může stát, že vyhledáváním větu nenajdeme, což ale znamená, že jsme K zatím netrefili. Nejpozději po K krocích nám hledání něco najde.

Proč nejpozději? Samotný klíč může být periodický, pak najdeme jen jeho periodu. Informaci o tom, jak skutečně vypadal původní klíč, už získat nemůžeme.

Ve výsledku tedy nejvýše K-krát opakujeme vyhledávání a náš algoritmus má dohromady časovou složitost $\mathcal{O}(NK)$.

Nejvíc paměti nám sebere vyhledávání v textu – potřebujeme si uložit vyhledávací automat. Na ten nám ale stále postačí $\mathcal{O}(N)$ paměti, ostatně stejně jako na uložení vstupu. Samotné upravené posloupnosti bychom ukládat nemuseli, dají se počítat „za letu“.

Mimochodem, pokud bychom algoritmu předhodili náhodná písmena místo textu, on by stejně nalezl nějaké řešení. Vždy totiž existuje klíč délky V, který text dešifruje tak, aby se v něm daná věta nacházela. Proti tomu není obrany, ale naštěstí po nás nikdo nechtěl, aby se algoritmus choval správně i pro nevalidní vstup.

Algoritmu navíc stačí (s drobnou modifikací získávání klíče), aby byla věta alespoň dvakrát tak dlouhá jako klíč – potom bude správný klíč jedním z nalezených. Zvyšujeme ale množství falešných klíčů, které v zašifrovaném textu najdou větu, přestože tam původně nebyla. Vzorový program je upraven tak, aby vypsal všechny nalezené klíče kratší než V.

Na závěr dodejme, že znalost části dešifrovaného textu je poměrně silný způsob, jak získat klíč šifry. Asi nejznámějším případem, který je možná nepravdivou historkou, je rozluštění šifer psaných na Enigmě. Němci totiž za války posílali každé ráno zprávy o počasí, které měly velice předvídatelný tvar, přičemž klíč se měnil pouze jednou denně.

Program (Python):

`http://ksp.mff.cuni.cz/viz/29-5-5.py`

Ondra Hlavatý

29-5-6 Nejsilnější kouzlo

Ačkoli zadání této úložky lze popsat třemi slovy: „najděte nejdelší palindrom“, možná byste nečekali, že řešení lze charakterizovat následující trojicí slov: „umíme to lineárně“. Než se ale dopracujeme k rychlému řešení, ukážeme si pro začátek dvě pomalejší. To druhé z nich se nám bude nesmírně hodit.

Asi každého napadlo řešení s časovou složitostí $\mathcal{O}(N^3)$, kde N je délka vstupního textu. Stačí vyzkoušet všechny možnosti. Vezmeme tedy každý možný začátek a každý možný konec. Tím získáme $\mathcal{O}(N^2)$ kandidátů. Teď už jen každého

kandidáta překontrolujeme jednoduchým průchodem v lineárním čase a nejdelší nalezený palindrom nakonec vypíšeme.

Kvadratické řešení

Často jste také vymysleli pěkné kvadratické řešení. Jeho myšlenka je jednoduchá. Nebudeme kontrolovat od krajů, ale od středu palindromů. Možných středů je totiž pouze lineárně s délkou vstupu. Střed palindromu je buď jeden znak (pro palindromy liché délky), nebo dvojice po sobě jdoucích znaků (pro sudé palindromy). Od každého tedy začneme kontrolovat postupně směrem ke krajům.

Pro jednoduchost můžeme hledat zvlášť nejdelší lichý palindrom a nejdelší sudý a na závěr si jen vybrat ten delší z nich. Pro liché palindromy tedy máme vždy pozici středu s, tj. index, na kterém je aktuálně testovaný střed palindromu ve vstupním poli.

V každém kroku porovnáme znaky na pozicích $s - i$ a $s + i$. Pokud jsou stejné, zvětšíme i o jedna. Pokud se však liší, znamená to, že nejdelší palindrom se středem v s jsme našli v předchozím kroku. Začíná tedy na znaku $s - i + 1$, končí na $s + i - 1$ a má délku $2i - 1$. V takovém případě můžeme přistoupit k dalšímu středu $(s = s + 1; i = 1)$.

Pro sudé palindromy to bude fungovat stejně, jen se na pár místech objeví navíc ± 1. Samozřejmě také musíme ošetřit to, abychom při kontrolování nevytekli s indexy mimo vstupní pole. Pokud vás zajímají detaily, podívejte se na ně do programu.

Umíme to lépe?

Na začátku jsme slibovali řešení s časovou složitostí $\mathcal{O}(N)$. Jak bychom mohli současné kvadratické řešení zrychlit? Nejprve řešení přidáme trochu paměti. V obyčejném poli si pro každý zkoumaný střed zapamatujeme délku nejdelšího možného palindromu.

Aby se nám s touto hodnotou lépe pracovalo, budeme si ve skutečnosti ukládat vzdálenost středu s od krajních znaků nejdelšího palindromu s tímto středem. Říkejme této vzdálenosti třeba *poloměr* a označme si ji $r[s]$. Pro palindrom délky 5 budeme mít uložený poloměr 2.

Představme si, že jsme právě našli nějaký relativně dlouhý palindrom se středem v s a krajními znaky $s - r[s]$ a $s + r[s]$.

V popsaném řešení nás nyní čeká to, že budeme postupně zkoušet hledat palindromy se středy $s + 1, s + 2, s + 3, \ldots$. Tato hledání budou vždy (alespoň ze začátku) probíhat uvnitř již nalezeného palindromu se středem v s. Tomuto palindromu se středem s říkejme *referenční*.

Co ale platí pro palindromy? Jsou přece symetrické! Takže pokud jsme našli nejdelší palindrom se středem v $s - j$, využijeme toho pro nalezení nejdelšího palindromu se středem v $s + j$.

Pokud palindrom se středem $s-j$ leží zcela uvnitř referenčního palindromu a ani oba nemají společný krajní znak, potom nemusíme poloměr palindromu se středem $s+j$ vůbec počítat. Rovnou přiřadíme již spočítaný poloměr $r[s+j] = r[s-j]$.

V opačném případě oba palindromy buď začínají na stejné pozici, nebo dokonce $s-j$ sahá mimo referenční palindrom. V obou případě víme o pozici $s+j$ pouze to, že na ní leží palindrom o poloměru alespoň $r[s]-j$. Jeho skutečný poloměr tedy budeme hledat až od této hodnoty.

Jakmile najdeme nejdelší palindrom se středem v $s+j$, začneme jej používat jako referenční palindrom. (V programu to je pouhé přiřazení do proměnné $s = s+j$).

Umíme to lineárně!

Je to vůbec funkční řešení? Všimněte si, že nově vzniklý algoritmus oproti předchozímu nekontroluje palindromy pouze na těch místech, o kterých jsme ukázali, že jsou symetrické s jinými, již zkontrolovanými místy.

Dobrá, tak jsme některé případy zrychlili, ale pomohli jsme si? Na první pohled ne. Středů stále procházíme lineárně a kontrola jednoho může trvat také až lineárně dlouho. Podívejme se na to ale z pohledu pravého okraje referenčního palindromu.

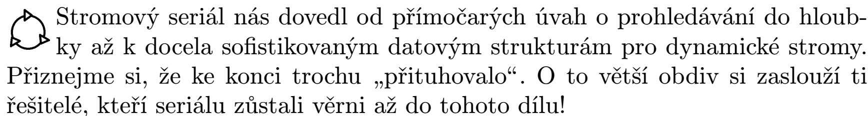
Při každém porovnání dvou znaků na vstupu buď posuneme pravý okraj o jedna doprava (pokud se znaky rovnají), nebo o jednom středu zjistíme poloměr jeho nejdelšího palindromu. Všimněte si, že pravý okraj referenčního palindromu se nikdy nepohne doleva. Dohromady to celé zabere nejvýše $2N$ porovnání pro liché palindromy a stejně tak pro sudé.

Program (Python):

`http://ksp.mff.cuni.cz/viz/29-5-6.py`

Jenda Hadrava

29-5-7 Stromy v pohybu



Stromový seriál nás dovedl od přímočarých úvah o prohledávání do hloubky až k docela sofistikovaným datovým strukturám pro dynamické stromy. Přiznejme si, že ke konci trochu „přituhovalo“. O to větší obdiv si zaslouží ti řešitelé, kteří seriálu zůstali věrni až do tohoto dílu!

Úkol 1: Následník uzlu

Rozcvička: máme BVS (totiž binární vyhledávací strom) a chceme najít následníka zadaného uzlu u. Využijeme toho, že prohledávání do hloubky navštěvuje uzly v rostoucím pořadí. Jak to vypadá z pohledu konkrétního uzlu? Nejprve do něj přijdeme shora a odejdeme do levého syna. Pak se vrátíme zleva,

vypíšeme aktuální uzel a odejdeme do pravého syna. Nakonec se vrátíme zprava, a hned poté odejdeme do otce.

Po vypsání u tedy pokračujeme do jeho pravého syna, má-li takového. Než ale vypíšeme příští uzel, půjdeme doleva, dokud to bude možné.

Pokud naopak u žádného pravého syna nemá, prohledávání se vrací z rekurze, a to tak dlouho, dokud se vrací z pravých synů. Jakmile se jednou vrátí z levého, vypíše aktuální uzel, což je opět hledaný následník.

Konečně se může stát, že se vrátíme z pravých synů až do kořene. Tehdy se prohledávání zastaví a žádný následník neexistuje. Není divu – nejpravější uzel v BVS je největší.

Hledání následníka tedy pracuje v čase nejvýše lineárním s hloubkou stromu.

Úkol 2: Následník ve splay stromu

Ve splay stromu můžeme samozřejmě následníka hledat stejně, ale když to uděláme trochu šikovněji, bude to (aspoň amortizovaně) rychlejší.

Začneme vysplayováním uzlu u. Tím se u dostane do kořene, takže pokud nebyl maximální, má určitě pravého syna. Takže stačí najít minimum z pravého podstromu: jít doprava a stále doleva, dokud to jde. A vzpomenout si na trik ze zadání, totiž nalezené minimum vysplayovat.

Tím zařídíme, že čas strávený hledáním minima je přímo úměrný času strávenému splayováním. A jelikož víme, že amortizovaná složitost splayování je $\mathcal{O}(\log n)$, musí být i amortizovaná složitost hledání minima $\mathcal{O}(\log n)$.

Úkol 3: Slepení cest za vrchol

Cesty A a B slepíme snadno: Nejprve nalezneme maximální uzel ve stromu cesty A, což je nějaký externí uzel reprezentující poslední vrchol cesty A. Tento uzel odstraníme a na jeho místo připojíme kořen stromu B a vysplayujeme ho. Po poslední hraně cesty A tedy bude v in-orderovém pořadí přirozeně následovat první vrchol cesty B. Časová složitost je amortizovaně logaritmická.

Úkol 4: Minimum cesty v cestě

Na počítání intervalových minim v posloupnosti se nám osvědčily intervalové stromy. Zde ovšem potřebujeme umět i spojovat posloupnosti a krájet je. Použijeme tedy podobnou myšlenku udržování minim podstromů, ale tentokrát to provedeme ve splay stromu.

Konkrétně každý uzel splay stromu – připomeňme, že reprezentuje hranu cesty – si zapamatuje jednak ohodnocení této hrany a jednak minimum z ohodnocení všech hran ve svém podstromu.

Udržuje se to snadno: kdykoliv změníme ohodnocení hrany, stačí přepočítat všechna minima na cestě do kořene splay stromu. A kdykoliv při splayování rotujeme, tak v uzlech, které se rotace účastnily, přepočítáme minima. V jednom uzlu umíme minimum přepočítat v konstantním čase z jeho ohodnocení a z minim uložených v jeho synech. Podobně můžeme minima aktualizovat při rozdělování a spojování splay stromů.

Zbývá popsat výpočet minima podcesty mezi danými dvěma vrcholy u a v. Mohli bychom se opět opičit po intervalových stromech (a v praxi by se to nejspíš vyplatilo), ale neodoláme a předvedeme jiný trik: pomocí *Split*(u) a *Split*(v) žádanou podcestu odsekneme od zbytku cesty. Pak se stačí podívat do kořene jejího splay stromu na minimum. A nakonec pomocí *Join*ů cesty poslepujeme zpět.

Toto vše trvá $\mathcal{O}(\log n)$ amortizovaně.

Úkol 5: Minimum cesty ve stromu

Nyní chceme cestová minima rozšířit na cesty v obecných stromech. Využijeme dekompozici na tlusté a tenké cesty. Každou tlustou cestu budeme reprezentovat splay stromem upraveným podle předchozího úkolu. Poslední vrchol tlusté cesty si bude pamatovat nejen tenkou hranu směrem ke kořeni stromu, ale i její ohodnocení.

Snadno upravíme *Expose*, aby při výměnách tenkých hran za tlusté a opačně správně přenášel ohodnocení. Jelikož přidáváme pouze konstantní množství práce na každou hranu, časová složitost *Expose* zůstane $\mathcal{O}(\log n)$ amortizovaně.

Změna ohodnocení hrany se týká pouze jedné tlusté cesty nebo jedné tenké hrany, takže ji stihneme v $\mathcal{O}(\log n)$ amortizovaně.

Nalezení minima cesty pak bude snadné: pomocí *Expose* z cesty uděláme tlustou cestu a pak se jenom podíváme do kořene jejího splay stromu. Opět vše amortizovaně logaritmické.

Úkol 6: Dynamická minimální kostra

Máme udržovat minimální kostru grafu, do nějž postupně přibývají hrany s daným ohodnocením (vahou). Graf nebude vždy souvislý, takže upřesněme, že nás zajímá minimální kostra každé komponenty souvislosti.

Budeme udržovat les koster jednotlivých komponent v podobě dynamického stromu upraveného podle předchozího úkolu, pouze s udržováním maxim místo minim. Na počátku v grafu nejsou žádné hrany, takže les obsahuje samé jednovrcholové stromy.

Uvažujme nyní přidání hrany uv do grafu. Nejprve se podíváme, zda u a v leží v různých stromech (k tomu stačí provést *Root*(u) a *Root*(v)). Pokud ano, pak leží i v různých komponentách souvislosti, takže nová hrana propojí dvě komponenty, a tedy se určitě nachází v každé kostře, čili i v té minimální. Tehdy operacemi *Evert* a *Link* hranu přidáme a jsme hotovi.

Dobrá, ale co když u i v leží v téže komponentě? Tehdy se podíváme na cestu (říkejme jí P), která spojuje u s v v minimální kostře této komponenty. Najdeme nejtěžší hranu f na této cestě (použijeme předchozí úkol). Pokud je uv těžší než f nebo stejně těžká, kostru ponecháme stejnou. Jinak hranu f odebereme a místo ní přidáme uv (to obnáší *Cut*, *Evert* a *Link*).

Dokažme, že tím vznikne správná minimální kostra. Uvažujme cyklus C vzniklý spojením konců cesty P hranou uv. Představme si, že hledáme mini-

mální kostru grafu s hranou uv Kruskalovým algoritmem, a uvažujme, jak se algoritmus chová k cyklu C. Pokud je uv těžší než f, narazí na uv algoritmus až v okamžiku, kdy se celá cesta P dostala do kostry, takže uv by již vytvořila cyklus a je zahozena. Pokud je naopak uv lehčí, pak hranu f předběhne a při přidávání f budou všechny vrcholy cesty P pospojované nějakou jinou cestou a f bude zahozena.

Minimální kostru tedy umíme po každém přidání hrany přepočítat v amortizovaně logaritmickém čase.

Příběh pokračuje

Svět dynamických grafových algoritmů je rozsáhlá džungle, dodnes ne zcela probádaná. V seriálu jsme navštívili její okrajové části, kde žijí dynamické stromy. Hluboko uvnitř se nacházejí mnohem divočejší algoritmy pracující s obecnými grafy a obecnými operacemi (například minimální kostra, která umí hrany nejen přidávat, ale i odebírat). Jsou to algoritmy jako tygr: oslnivě krásné, ale není vůbec snadné si je ochočit. Nechte si o nich vyprávět na podzimním soustředění…

Pěkné prázdniny (ehm, tedy už pěkný školní rok) přeje váš průvodce lesem

Martin „Medvěd“ Mareš

Pořadí řešitelů KSP

Pořadí	*Jméno*	*Škola*	*Ročník*	*Úloh*	*Bodů*
0.				35	300.0
1.	Tomáš Domes	MendelG_OP	4	26	222.3
2.	Lukáš Rozsypal	GÚstavníPH	4	23	199.4
3.	Peter Grajcar	GMetodovaBA	3	22	179.5
4.	Roman Bujdák	G JM Galanta	3	24	161.8
5.	Pavel Turek	GTomkovaOL	4	17	157.4
6.	Jakub Pelc	G UherBrod	3	16	147.7
7.	Matouš Bílek	GJŠkodyPŘ	2	16	136.2
8.	Richard Hladík	GOAMarLaz	4	13	124.1
9.	Martin Kurečka	GJarošeBO	3	12	112.3
10.	Pavel Turinský	G Brandýs	4	13	93.4
11.	Lukáš Caha	GZborovPH	3	13	75.0
12.	Kateřina Čížková	G_Rokycany	3	9	72.3
13.	Rajmund Hruška	GPošKošice	4	9	70.0
14.	Stanislav Lukeš	GPísnickáPH	4	8	69.8
15.	Jan Kaifer	GKepleraPH	1	12	68.9
16.	Filip Geib	G MMH LM	3	9	66.6
17.	Tomáš Konečný	GJirsíkaČB	4	8	64.4
18.	Michal Kodad	SPŠ_Smíchov	1	9	63.1
19.	Martin Picek	GJirsíkaČB	2	7	62.7
20.	Viktor Fukala	GKepleraPH	0	6	62.0
21.	František Deckert	GOpatovPHA	4	6	59.2
22.	František Kmječ	G Brandýs	1	7	56.2
23.	Jonáš Fiala	GJungmanLT	4	7	56.0
24.	Miroslav Hrabal	GTomkovaOL	3	7	53.6
25.	Jakub Pintera	SPŠ Prosek	4	6	51.4
26.	Zuzana Urbanová	GFXŠaldyLI	3	5	48.6
27.	Klára Tauchmanová	GOhradníPH	3	5	44.8
28.	Lenka Kopfová	MendelG_OP	2	5	44.6
29.	Matouš Mařík	G_Krumlov	4	5	40.7
30.	Ondřej Gonzor	G Brandýs	0	8	38.9
31.	Karel Balej	G_Rokycany	2	5	36.7
32.	Tomáš Raunig	GHlu	2	7	34.3
33.	Václav Pavlíček	SPSE_Pard	1	6	33.5
34.	Kryštof Mitka	ZŠUniverzum	0	4	31.2
35.	Jiří Löffelmann	GLitoměřPH	3	6	29.5
36.	Jindřich Dítě	VOSPŠŽďár	1	4	27.5
37.	Filip Masár	PiarGNitra	3	4	27.4
38.	Daniel Skýpala	GTomkovaOL	−1	4	27.2

Pořadí	*Jméno*	*Škola*	*Ročník*	*Úloh*	*Bodů*
39.	Petr Gebauer	GMělník	3	3	26.8
40.	Václav Šraier	GČeskoliPH	4	4	25.7
41.	Anna Řechtáčková	GJarošeBO	4	3	22.7
42.	Kristián Jacik	GSRandyJN	4	5	22.6
43.	Ondřej Krsička	GJarošeBO	1	6	22.0
44.	Kateřina Černá	GMilevsko	2	4	21.7
45.	Anna Hollmannová	GSRandyJN	0	7	21.5
46.	Jakub Suchánek	GOpatovPHA	3	2	19.0
47.	Radek Olšák	MensaG	2	2	18.4
48.	Daniela Hrbáčová	G Wicht	3	2	16.1
49.	Přemysl Šťastný	GŽamberk	4	4	15.6
50.	Ondřej Cach	SPSE_Pard	1	4	15.4
51.	Antonin Hejny	GLitoměřPH	0	2	13.3
52.–53.	Vojtěch Hudec	G_ČTřebová	3	3	12.1
	Josef Polášek	GKepleraPH	1	3	12.1
54.	Vojtěch Lengál	GZborovPH	3	1	11.0
55.	Štěpán Zapadlo	GJŠkodyPŘ	1	2	9.3
56.	Dalibor Kramář	G BO-Řeč	2	2	8.7
57.	Adam Dřínek	GNAlejíPH	3	1	8.0
58.	Vít Skalický	GPísnickáPH	−1	1	7.9
59.	Jan Neumann	GNAlejíPH	3	1	7.7
60.–61.	Jakub Dobrý	GMikulášPL	3	1	7.6
	Anna Šebestíková	GČeskáČB	2	1	7.6
62.	Michael Kozel	GZborovPH	3	1	7.5
63.	Jan Jeníček	GNAlejíPH	1	1	7.4
64.	Jakub Jirkal	GJungmanLT	2	1	7.2
65.	Jakub Spišák	G VBN Prie	4	3	7.0
66.	Michaela Bobeničová	GPošKošice	2	1	6.9
67.–68.	Erik Kučák	GHorMichal	4	2	6.7
	Martin Miller	GVoděraPH	3	1	6.7
69.	Michal Töpfer	G DrJPekMB	4	2	6.6
70.	Eliška Vlčinská	GHladnov	2	1	6.3
71.	Jan Bíl	GDašickáPA	4	1	4.0
72.	Jonáš Havelka	GJírovcČB	1	1	2.2

Jiří Setnička a kolektiv

Korespondenční seminář z programování XXIX. ročník

Vydal MatfyzPress
vydavatelství Matematicko-fyzikální fakulty Univerzity Karlovy
Sokolovská 83, 186 75 Praha 8
jako svou 549. publikaci.

Vytisklo Reprostředisko MFF UK
Sokolovská 83, 186 75 Praha 8.

Publikace neprošla recenzním ani lektorským řízením.
Nakladatelství neodpovídá za kvalitu a obsah textu.
Vydáno pro vnitřní potřebu MFF UK.
Publikace není určena k prodeji.

Autoři a opravující úloh:
Jana Bátoryová, Karolína Burešová, Marek Černý, Zuzana Drázdová
Jan Gocník, Jan Hadrava, Ondřej Hlavatý, Štěpán Hojdar, Petr Houška
Jan Knížek, Václav Končický, Martin Mareš, Jakub Maroušek
Jan Matějka, Jiří Sejkora, Jiří Setnička, Dominik Smrž, Zuzana Šimečková
Filip Štědronský, Michaela Štolová, Jakub Tětek, Jan Tománek
Kateřina Zákravská

Autoři příběhů v zadání:
Jiří Setnička, Karolína Burešová, Filip Štědronský

Autor seriálu:
Martin Mareš

TEX-ová makra pro sazbu ročenky vytvořili Martin Mareš, Jan Matějka, Radim Cajzl a Jiří Setnička.

S jejich pomocí ročenku vysázeli Jiří Setnička a Pavol Rohár.

Obrázek na obálce nakreslila Petra Pelikánová.

Sazba byla provedena písmem Computer Modern v programu TEX.

První vydání
Praha 2017

ISBN 978-80-7378-355-6

ISBN 978-80-7378-355-6