Korespondenční Seminář z Programování

29. ročník **KSP** **Květen 2017**

Milí řešitelé a řešitelky!

Zima se sice letos nevzdávala lehce a sníh nás překvapil ještě v dubnu, ale teď už snad bude teplota až do léta jenom růst. A společně s rostoucí teplotou nám na informatické zahrádce vyrostlo pár nových úloh, o které se s vámi chceme podělit.

Nachystejte si tedy své zahradnické náčiní a pojďte se s námi vrhnout na poslední sérii tohoto školního roku!

A připomínáme, že každému řešiteli, který v tomto ročníku z každé série dostane alespoň 5 bodů, darujeme KSP propisku, blok, placku. Pokud jste tedy vytrvali přes předchozí série, tak zachovejte pracovní nasazení i v této poslední letošní sérii!

Termín série: **Pondělí 29. května v 8:00**
Odevzdávání: Přes web na adrese *https://ksp.mff.cuni.cz/submit/*
Odměna série: **Sladkou odměnu** pošleme každému, kdo z této série získá alespoň **29 bodů**.

Pátá série dvacátého devátého ročníku KSP

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.

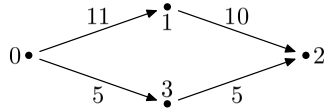
Formát výstupu: Na první řádek výstupu vypište délku cesty v hodinách, na druhém řádku pak vypište mezerami oddělenou posloupnost měst (včetně prvního a posledního), kterými tato nejkratší cesta vede.

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.

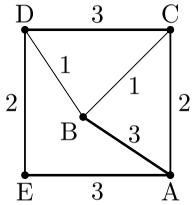
Podle pokynů z deníku je potřeba odpojit nějaké dráty tak, aby zanikly všechny cykly.

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:

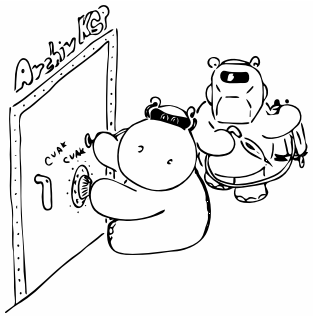


je správným řešením odebrat hrany např. v pořadí AC (nejkratší na $ACDEA$), BC, BD. Jiným správným pořadím může být BC, ED, BD.

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.

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.

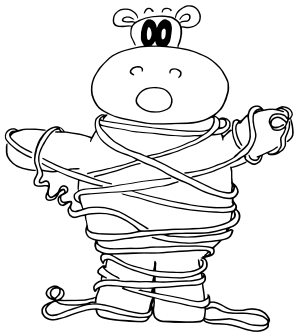
Kouzla létala vzduchem. Ani Rhea, ani Beliar neměli nad tím druhým navrch, ale Rhee rychle docházely síly. Držela svůj ochranný štít a pokoušela se sestavit z dostupných sil co nejsilnější kouzlo.

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

29-5-7 Stromy v pohybu — 15 bodů

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.[1]

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 $(x_n, +\infty)$.

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[2] v kapitole o amortizaci. Zde si vystačíme se základními principy.

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

[2] `http://pruvodce.ucw.cz/`

	řešitel	*škola*	*ročník*	*sérií*	*H4-1*	*H4-2*	*H4-3*	*H4-4*	*H4-5*	*H4-6*	*H4-7*	*série*	*celkem*
11.	Rajmund Hruška	GPošKošice	4	2								0,0	70,0
12.	Filip Geib	G MMH LM	3	5								0,0	66,6
13.	Kateřina Čížková	G_Rokycany	3	3	7	6			4			23,1	57,7
14.	Jonáš Fiala	GJungmanLT	4	7								0,0	56,0
15.	Stanislav Lukeš	GPísnickáPH	4	14	10	12			12			34,0	55,9
16.	Martin Picek	GJirsíkaČB	2	2								0,0	55,0
17.	Lukáš Caha	GZborovPH	3	3	7				4			15,8	54,5
18.	Jakub Pintera	SPŠ Prosek	4	2								0,0	51,4
19.	František Deckert	GOpatovPHA	4	2	10		11					21,0	50,0
20.	Viktor Fukala	GKepleraPH	0	2	10		11		12	11		44,0	44,0
21.	Miroslav Hrabal	GTomkovaOL	3	4								0,0	43,6
22.	Matouš Mařík	G_Krumlov	4	1								0,0	40,7
23.	Tomáš Konečný	GJirsíkaČB	4	1	10		11	1	4	6		40,2	40,2
24.	Jan Kaifer	GKepleraPH	1	5								0,0	34,5
25.	Tomáš Raunig	GHlu	2	2								0,0	34,3
26.	Michal Kodad	SPŠ_Smíchov	1	7	7							7,6	34,1
27.	Václav Pavlíček	SPSE_Pard	1	7	7							8,0	33,5
28.	František Kmječ	G Brandýs	1	4								0,0	33,3
29.	Ondřej Gonzor	G Brandýs	0	3	7							8,0	31,6
30.	Kryštof Mitka	ZŠUniverzum	0	3								0,0	31,2
31.	Jiří Löffelmann	GLitoměřPH	3	7			3					3,6	29,5
32.	Filip Masár	PiarGNitra	3	2								0,0	27,4
33.	Daniel Skýpala	GTomkovaOL	−1	3	7							7,6	27,2
34.	Petr Gebauer	GMělník	3	2								0,0	26,8
35.	Anna Řechtáčková	GJarošeBO	4	2								0,0	22,7
36.	Kristián Jacik	GSRandyJN	4	1								0,0	22,6
37.	Ondřej Krsička	GJarošeBO	1	2								0,0	22,0
38.	Anna Hollmannová	GSRandyJN	0	3								0,0	21,5
39.	Jakub Suchánek	GOpatovPHA	3	4			11					11,0	19,0
40.–41.	Radek Olšák	MensaG	2	1								0,0	18,4
	Václav Šraier	GČeskoliPH	4	11	7	7						12,7	18,4
42.	Jindřich Dítě	VOSPŠŽďár	1	2								0,0	17,2
43.	Přemysl Šťastný	GŽamberk	4	15								0,0	15,6
44.	Ondřej Cach	SPSE_Pard	1	1								0,0	15,4
45.	Karel Balej	G_Rokycany	2	1	7			4,5				14,5	14,5
46.	Antonin Hejny	GLitoměřPH	0	1	6				2			13,3	13,3
47.–48.	Vojtěch Hudec	G_ČTřebová	3	3								0,0	12,1
	Josef Polášek	GKepleraPH	1	1								0,0	12,1
49.	Vojtěch Lengál	GZborovPH	3	1								0,0	11,0
50.	Dalibor Kramář	G BO-Řeč	2	1								0,0	8,7
51.	Adam Dřínek	GNAlejíPH	3	1								0,0	8,0
52.	Vít Skalický	GPísnickáPH	−1	1	5							7,9	7,9
53.	Jan Neumann	GNAlejíPH	3	2								0,0	7,7
54.–55.	Jakub Dobrý	GMikulášPL	3	4								0,0	7,6
	Anna Šebestíková	GČeskáČB	2	2								0,0	7,6
56.	Michael Kozel	GZborovPH	3	1								0,0	7,5
57.	Jan Jeníček	GNAlejíPH	1	1								0,0	7,4
58.	Jakub Jirkal	GJungmanLT	2	1								0,0	7,2
59.	Jakub Spišák	G VBN Prie	4	1								0,0	7,0
60.	Michaela Bobeničová	GPošKošice	2	1	5							6,9	6,9
61.–62.	Erik Kučák	GHorMichal	4	1								0,0	6,7
	Martin Miller	GVoděraPH	3	1								0,0	6,7
63.	Michal Töpfer	G DrJPekMB	4	11								0,0	6,6
64.	Eliška Vlčinská	GHladnov	2	2								0,0	6,3
65.	Jan Bíl	GDašickáPA	4	1								0,0	4,0
66.	Jonáš Havelka	GJírovcČB	1	2								0,0	2,2

KSP pro vás připravují studenti Matematicko-fyzikální fakulty Univerzity Karlovy.

Webové stránky: *https://ksp.mff.cuni.cz/* **E-mail:** *ksp@mff.cuni.cz* **Diskusní fórum:** *https://ksp.mff.cuni.cz/forum/*

Chcete-li s námi komunikovat bezpečně, můžete si ověřit náš HTTPS certifikát – jeho SHA1 fingerprint je: `E9:DB:EE:C6:62:BC:14:DE:09:E4:E8:97:DC:36:0E:87:B3:50:B0:01`.

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 poščítat $\mathcal{O}(\log n)$ 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)[10] 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

Výsledková listina čtvrté série dvacátého devátého ročníku KSP

	řešitel	*škola*	*ročník*	*sérií*	*H4-1*	*H4-2*	*H4-3*	*H4-4*	*H4-5*	*H4-6*	*H4-7*	*série*	*celkem*
0.					10	12	11	8	12	11	15	61,0	241,0
1.	Lukáš Rozsypal	GÚstavníPH	4	7	7	4			12		4	31,2	171,4
2.	Tomáš Domes	MendelG_OP	4	5	9	12		8	4	11	14	54,6	166,9
3.	Pavel Turek	GTomkovaOL	4	8	10		11	8	12	11	13	57,6	157,4
4.	Peter Grajcar	GMetodovaBA	3	4	6	7	11		12	11		51,5	144,2
5.	Roman Bujdák	G JM Galanta	3	4	7	6	11	4,5	4	6		38,4	137,9
6.	Jakub Pelc	G UherBrod	3	9		12,1		8	4	9	10	37,1	137,7
7.	Richard Hladík	GOAMarLaz	4	23								0,0	121,6
8.	Martin Kurečka	GJarošeBO	3	2	10	11,5	5	8	12	11	12	59,0	112,3
9.	Matouš Bílek	GJŠkodyPŘ	2	2	7	7		3	4	9	10	48,8	89,8
10.	Pavel Turinský	G Brandýs	4	13	10			8				18,0	85,4

[10] http://ksp.mff.cuni.cz/viz/kucharky/minimalni-kostry

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 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.

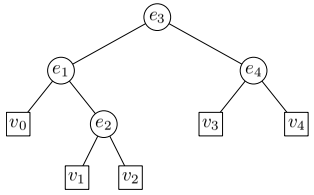
Rozdělování a spojování můžeme například použít ke vkládání a mazání hodnot. Tyto operace ale překvapivě nebudeme potřebovat.

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.

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.

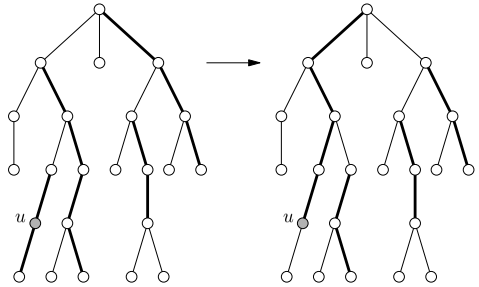
Každou tlustou cestu budeme reprezentovat již popsaným způsobem pomocí splay stromu. Poslední vrchol cesty w (v původním stromu leží nejvýše, ve splay stromu je to nejpravější externí uzel) si bude pamatovat informace o tenké hraně do nadřazené cesty: vrchol $tparent(w)$, do nějž tenká hrana vede. Vede-li 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.
- $Join(u, v)$ – je-li u kořen jednoho stromu a v libovolný vrchol jiného stromu, spojí oba stromu přidáním hranu 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“.

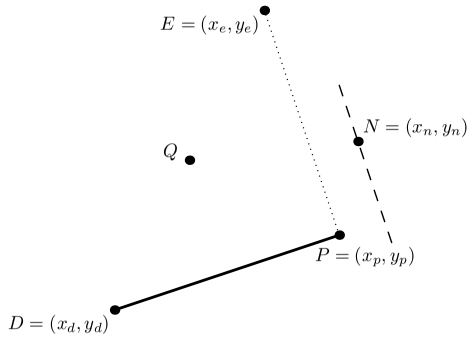
Úkol 6 [2b]: Navrhněte datovou strukturu pro inkrementální udržování minimální kostry. Na počátku máme graf bez hran a postupně přidáváme ohodnocené hrany. Po každém přidání chceme zjistit, jak se změnila minimální kostra.

Martin „Medvěd“ Mareš

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.



$$x_e = x_p - (y_p - y_d); \quad y_e = y_p + (x_p - x_d)$$

Ve výše uvedeném vztahu pro $v \cdot |DP|$ nahradíme body D a S za body E a N a dostaneme:[8]

$$v' \cdot |DP| = x_e(y_p - y_n) + x_p(y_n - y_e) + x_n(y_e - y_p)$$

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).

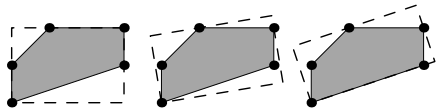
Pokud chceme počítat ultra přesně, uložíme si obě čísla zvlášť, tedy místo S si uložíme dvojici $(S \cdot |DP|^2, |DP|^2)$, a při hledání nejmenšího opsaného obdélníka pak můžeme porovnávat zlomky $S = \frac{S \cdot |DP|^2}{|DP|^2}$ algoritmem pro přesné porovnávání racionálních čísel.[9]

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“.



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

[8] Tiše též využíváme skutečnosti, že $|DP| = |EP|$.

[9] Platí, že $\frac{p}{q} > \frac{r}{s} \Leftrightarrow ps > rq$, pokud $q > 0$, $s > 0$.

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

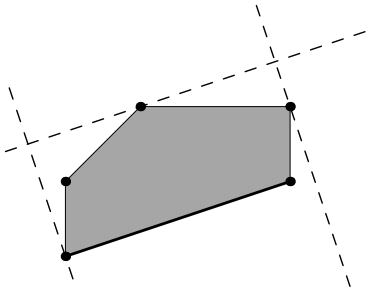
Václav Končický

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.

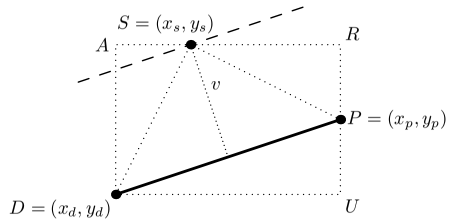
Ze všech takto nalezených opsaných obdélníků pak vybereme ten nejmenší. U mnohoúhelníka majícího $\mathcal{O}(M)$ hran bude celý algoritmus trvat $\mathcal{O}(M \cdot \phi(M))$, kde $\phi(M)$ je složitost vyhledání opsaného obdélníka.



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.

Recepty z programátorské kuchařky: Rekurzivní funkce a 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

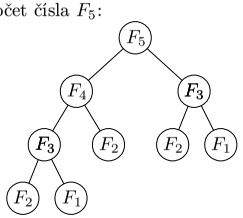
Budeme počítat n-té číslo Fibonacciho posloupnosti. To je posloupnost, jejíž nultý člen je nula, první je jednička ($F_0 = 0$, $F_1 = 1$) a každý další člen je součtem dvou předchozích ($F_n = F_{n-1} + F_{n-2}$ pro $n > 1$). Začíná takto:

0 1 1 2 3 5 8 13 21 34 55 89 ...

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.

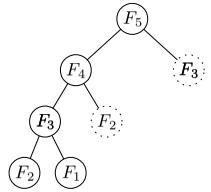
Jak najít efektivnější algoritmus? Všimneme si, že některé podstromy jsou shodné. Zřejmě to budou ty části, které reprezentují výpočet stejného Fibonacciho čísla – v našem příkladě třeba třetího. Tyto výpočty opakujeme stále dokola.

Nenabízí se proto nic snazšího, než si jejich výsledky uložit a pak je kdykoliv vytáhnout jako pověstného králíka z klobouku[3] 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:



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 $F_{1\ldots k}$ (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
```

[3] 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, ...

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.[4]

```
# 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.

[4] http://ksp.mff.cuni.cz/viz/kucharky/tezke-problemy

tek 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 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.

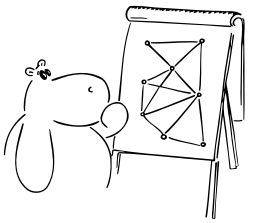
Od kořene k synům

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.

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 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.

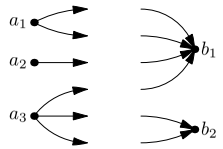
Co od takového rozdělení požadujeme? Protože každý dopis odeslaný někým z gangu A byl přijat někým z gangu B, musel gang B celkem přijmout přesně tolik dopisů, kolik jich gang A celkem odeslal. To samé musí platit v opačném směru. Takovému rozdělení budeme říkat *vyvážené*.

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.

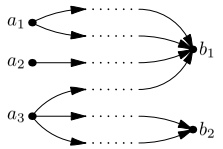
Představme si třeba následující situaci: gang A má tři členy, kteří poslali po řadě 2, 1 a 3 dopisy. Gang B má dva členy, kteří přijali 4 a 2 dopisy. Rozdělení je ve směru A→B vyvážené: tímto směrem bylo odesláno i přijato 6 dopisů.

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).[6]

K řešení použijeme obvyklý algoritmus pro batoh, který je popsán v naší kuchařce o dynamickém programování.[7]

Pokud dokážeme naplnit batoh předměty o celkové váze přesně M, jim odpovídající lidé tvoří např. gang B, zby-

[6] http://ksp.mff.cuni.cz/viz/kucharky/tezke-problemy
[7] http://ksp.mff.cuni.cz/viz/kucharky/dynamicke-programovani

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 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 $D[i][k] + D[k][j]$.

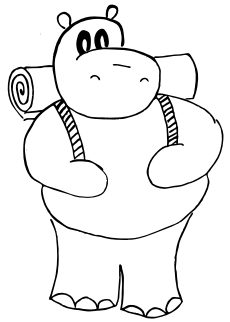
Pokud je součet $D[i][k]+D[k][j]$ menší než stávající hodnota $D[i][j]$, nahradíme hodnotu na pozici $D[i][j]$ tímto součtem, jinak ji ponecháme.

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 $A[1 \dots a]$ 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:

posloupnost:	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 čtvrté série dvacátého devátého ročníku KSP

29-4-1 Odevzdávání písemek

O třech slibných algoritmech

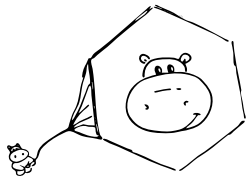
Za úkol máme rozdělit zadanou posloupnost na dvě rostoucí podposloupnosti: červenou a modrou. Nabízí se následné víceméně evidentní algoritmy. Všechny začnou se dvěma prázdnými posloupnostmi a postupně do nich budou přidávat jednotlivé prvky vstupu.

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

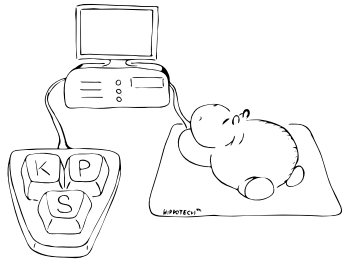
29-4-2 Hrací automat

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, budeme si vše ukládat do vyváženého vyhledávacího stromu.[5]

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

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