Korespondenční Seminář z Programování

29. ročník **KSP** **Prosinec 2016**

Milí řešitelé a řešitelky!

Přichází zima, doma pomalu ubývá zbytků cukroví a nový rok se chystá za rachotu výbušnin vpadnout do našich dveří. Zkuste ho zaskočit předsevzetím, že v KSPčku příští rok vyřešíte více úloh než letos! My vám k tomu pomůžeme třetí sérií. Přejeme vám co nejvíce úspěchů a doufáme, že nám zachováte přízeň i nadále :-)

A pokud stále váháte, připomínáme, že každému řešiteli, který v tomto ročníku z každé série dostane alespoň 5 bodů, darujeme KSP propisku, blok, placku a možná i další překvapení.

Za řešení KSP je také možné být přijat na MFF UK bez přijímacích zkoušek. Na získání osvědčení úspěšného řešitele je letos třeba získat v hlavní kategorii alespoň 150 bodů. Maturanti, kteří by chtěli osvědčení využít letos, jej musí získat nejpozději za čtvrtou sérii.

Termín série: **Pondělí 30. ledna 2017 v 8:00**
Odevzdávání: Přes web na adrese *https://ksp.mff.cuni.cz/submit/*
Odměna série: **Sladkou odměnu** pošleme každému, kdo získá **alespoň 42 bodů z celé série**.

Třetí série dvacátého devátého ročníku KSP

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:	*Ukázkový výstup:*

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

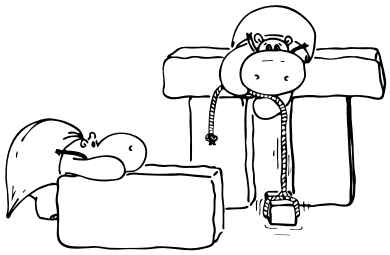
Zkusili několik sad závaží a po vyvážení se výtah konečně rozjel. Trpasličí ozubená kola se sice párkrát zadrhla, ale poté, co se z nich obrousila rez, už je výtah lehce dovezl několik set metrů do hloubky.

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.“

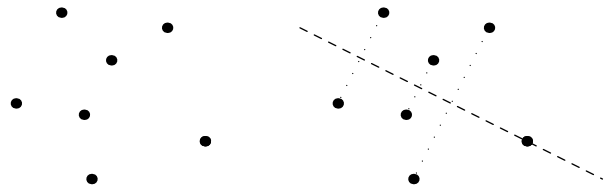


29-3-3 Skřetí věže — 11 bodů

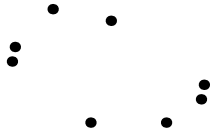
Průzkumníci se dostali nad skřetí ležení a zvláště je zaujaly jejich hlídkové věže. Vypadaly nezvykle symetricky, ale chtěli by ověřit, že jsou skutečně symetrické.

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.

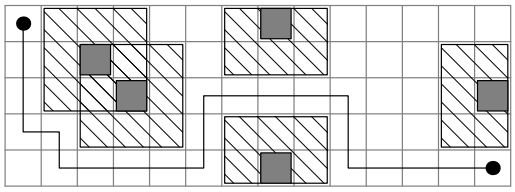
Pláň, na které jsou posazeny hlídky, si můžeme představit jako velikou čtvercovou síť (o velikosti $N \times M$) a hlídky jsou posazené na některých políčcích. Hlídek je řádově méně, než je počet políček pláně, a jejich pozice se mezi průchody jednotlivých bojovníků nemění.

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.

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.

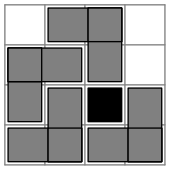
Gorf doběhl k truhlici a již si chystal paklíče, když tu mu došlo, že tady jeho paklíče budou k ničemu...

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

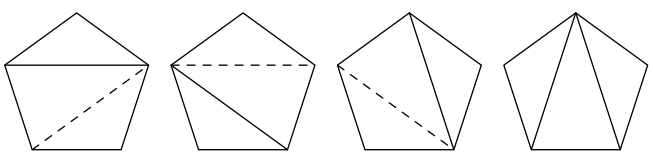
Drak je ovládán s pomocí obrazce na zemi vyskládaného v konvexním mnohoúhelníku. Mezi vrcholy tohoto mnohoúhelníku jsou položené dlouhé světélkující tyče a to tak, že se nekříží a celý obsah mnohoúhelníku je jimi rozdělen na trojúhelníky (informatici by řekli, že je *triangulován*).

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.



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

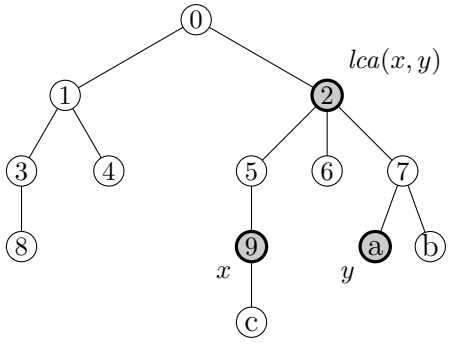
Jirka Setnička

29-3-7 Stromoví předci — 15 bodů

Náš seriál o stromech pokračuje dalším dílem, tentokrát o hledání (pra)předků vrcholů a užitečné technice zdvojování. Z předchozích dílů se nám bude hodit prohlédávání 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 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)^{k}předci a skočky

Na chvíli 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 $Pra(v, k)$ jako k-tý vrchol na cestě od v do kořene. Tedy $Pra(v, 0)$ je v sám, $Pra(v, 1)$ jeho otec, $Pra(v, 2)$ dědeček a obecně $Pra(v, k + 1) = P(Pra(v, k))$. Bude se nám též hodit, že platí $Pra(v, i + j) = Pra(Pra(v, i), j)$.

Počítat k-tého prapředka podle definice trvá $\mathcal{O}(k)$. Ukážeme zajímavý předvýpočet, s nímž to půjde rychleji.

Jistě si můžeme předpočítat všechna $Pra(v, k)$, 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 $S(v, i) = Pra(v, 2^i)$ pro $i = 0, \ldots, \lfloor \log_2 n \rfloor$. Jistě pro ni platí

$$\begin{aligned} S(v, 0) &= Pra(v, 1) = P(v), \\ S(v, i + 1) &= Pra(v, 2^{i+1}) = Pra(v, 2^i + 2^i) = \\ &= Pra(Pra(v, 2^i), 2^i) = S(S(v, i), i). \end{aligned}$$

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
$S(v,0)$	$\emptyset$	0	0	1	1	2	2	2	3	5	7	7	9
$S(v,1)$	$\emptyset$	$\emptyset$	$\emptyset$	0	0	0	0	0	1	2	2	2	5
$S(v,2)$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	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 $S(x, 0)$.

Vyzkoušejme 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.

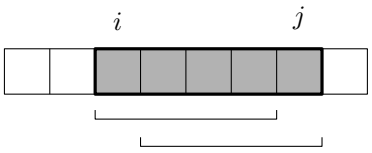
Jak na RMQ? Můžeme například použít intervalové stromy z minulého dílu. V čase $\mathcal{O}(n)$ vytvoříme pro naši poslupnost intervalový strom, jehož vnitřní vrcholy si budou pamatovat, kde v příslušném podstromu leží minimum. Minimum z obecného úseku pak vyhodnotíme v čase $\mathcal{O}(\log n)$.

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. $\quad\quad M(i,k) = \min(M(i,k-1), M(i+2^{k-1}, k-1))$

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.

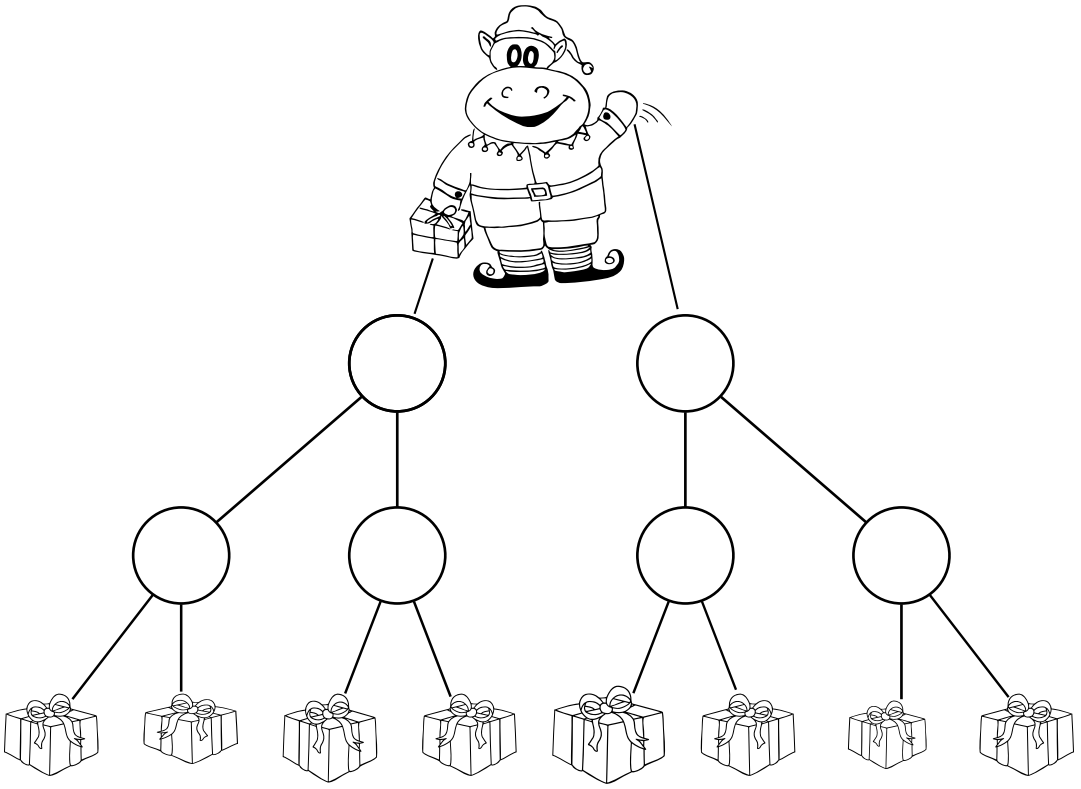
Tak získáme datovou strukturu pro RMQ, a tedy i LCA, v čase $\mathcal{O}(n \log n)/\mathcal{O}(1)$.

(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 ještě, 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,[1] v kapitole o dekompozici stromů.

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



Doufáme, že jste dostali aspoň tolik dárečků, kolik je pod naším vánočním binárním stromem :-)

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

Recepty z programátorské kuchařky: 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 $(N-1)/2 \pm 1$); přerovnávání v obou částech dohromady trvá opět $\mathcal{O}(N)$ a vzniknou tím části dlouhé nejvýše $N/4$. Zanoříme-li se v rekurzi do hloubky k, pracujeme s částmi dlouhými nejvýše $N/2^k$, které dohromady dají nejvýše N (všechny části dohromady dají prvky vstupní posloupnosti bez těch, které jsme si už zvolili jako pivoty). V hloubce $\lceil \log_2 N \rceil$ už jsou všechny části nejvýše jednoprvkové, takže se rekurze zastaví. Celkem tedy máme $\lceil \log_2 N \rceil$ hladin (hloubek) a na každé z nich trávíme lineární čas, dohromady $\mathcal{O}(N \log N)$.

V tomto důkazu jsme se ale dopustili jednoho podvodu: Zapomněli jsme na to, že také musíme medián umět najít. Jak z této nepříjemné situace ven?

- *Naučit se počítat medián.* Ale jak?
- *Spokojit se se „lžimediánem“:* Kdybychom si místo mediánu vybrali libovolný prvek, který bude v setříděné posloupnosti „v prostřední polovině“ (čili alespoň čtvrtina prvků bude větší a alespoň čtvrtina menší než on), získáme také složitost $\mathcal{O}(N \log N)$, neboť úsek délky N rozložíme na úseky, které budou mít délky nejvýše $(1-1/4)\cdot N$, takže na k-té hladině budou úseky délek nejvýše $(1-1/4)^k \cdot N$, čili hladin bude maximálně $\log_{1-1/4} N = \mathcal{O}(\log N)$. Místo 1/4 by fungovala i libovolná jiná konstanta mezi nulou a jedničkou, ale ani to nám nepomůže k tomu, abychom uměli lžimedián najít.
- *Recyklovat pravidlo* typu „vezmi poslední prvek“ a jen ho trochu vylepšit. To bohužel nebude fungovat, protože pokud budeme při výběru pivota hledět jenom na konstantní počet prvků, bude poměrně snadné přijít na vstup, pro který toto pravidlo bude dávat kvadratickou složitost, i když obvykle půjde dokázat, že takových vstupů je „málo“. (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 takovéhoto randomizovaného QS bude v průměru 2-krát větší, než lžimediánového QS, čili v průměru také $\mathcal{O}(N \log N)$. Jednoduše řečeno, zatímco fixní pravidlo nám dalo dobrý čas pro průměrný vstup, ale existovaly vstupy, na kterých bylo pomalé, randomizování nám dává dobrý průměrný čas pro všechny možné vstupy.

Hledání k-tého nejmenšího prvku

Nad QuickSortem jsme zvítězili, ale současně jsme při tom zjistili, že neumíme rychle najít medián. To tak nemůžeme nechat, a proto rovnou zkusíme vyřešit obecnější problém: najít k-tý nejmenší prvek (medián dostáváme pro $k = \lfloor N/2 \rfloor$).

První řešení této úlohy se nabízí samo. Načteme posloupnost do pole, prvky pole setřídíme nějakým rychlým algoritmem a kýžený k-tý nejmenší prvek nalezneme na k-té pozici v nyní již setříděném poli. Má to však jeden háček. Pokud prvky, které máme na vstupu, umíme pouze porovnat, pak nedosáhneme lepší časové složitosti (a to ani v průměrném případě) než $\mathcal{O}(N \log N)$ – rychleji prostě třídit nelze, důkaz můžete najít například v třídící kuchařce.

O něco rychlejší řešení je založeno na výše zmíněném algoritmu QuickSort (často se mu proto říká QuickSelect). Opět si vybereme pivota a posloupnost rozdělíme na prvky menší než pivot, pivota a prvky větší než pivot (pro jednoduchost budeme předpokládat, že žádné dva prvky posloupnosti nejsou stejné).

Pokud se pivot nalézá na k-té pozici, je to hledaný k-tý nejmenší prvek posloupnosti, protože právě $k-1$ prvků je menších. Zbývají dva případy, kdy tomu tak není. Pakliže je pozice pivota v posloupnosti větší než k, pak se hledaný prvek nalézá nalevo od pivota a postačí rekurzivně najít k-tý nejmenší prvek mezi prvky nalevo. V opačném případě, kdy je pozice pivota menší než k, je hledaný prvek v posloupnosti napravo od pivota. Mezi těmito prvky však nebudeme hledat k-tý nejmenší prvek, ale $(k-p)$-tý nejmenší prvek, kde p je pozice pivota v posloupnosti.

Časovou složitost rozebereme podobně jako u QuickSortu. Nešikovná volba pivota dává opět v nejhorším případě kvadratickou složitost. Pokud bychom naopak volili za pivota medián, budeme nejprve přerovnávat N prvků, pak jich zbude nejvýše $N/2$, pak nejvýše $N/4$ atd., což dohromady dává složitost $\mathcal{O}(N + N/2 + N/4 + \ldots + 1) = \mathcal{O}(N)$. Pro lžimedián dostaneme rovněž lineární složitost a opět stejně jako u QS můžeme nahlédnout, že náhodnou volbou pivota dostaneme v průměru stejný čas jako se lžimediánem.

Program bude velmi jednoduchý, využijeme-li přerovnávací proceduru od QS:

```
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
    medians = [0] * petic
    for i in range(0, pocet, 5):
        if (i + 5 > pocet):
            # Ignorujeme neúplnou pětici
            break
        quicksort(pole, L + i, L + i + 5)
        medians[i // 5] = pole[i + 2]

    # Nalezneme medián mediánů pětic
    median = kty(medians, 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 $7/10 \cdot N$.

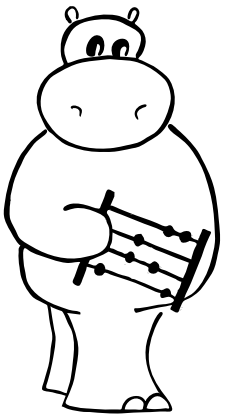
Rozdělení na pětice, hledání mediánů pětic a přerovnávání trvá lineárně, tedy nejvýše cN kroků pro nějakou konstantu $c > 0$. Pak už algoritmus pouze dvakrát rekurzivně volá sám sebe: nejprve pro $N/5$ mediánů pětic, pak pro $\leq 7/10 \cdot N$ prvků před/za pivotem. Pro celkovou časovou složitost $t(N)$ 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

Vzorová řešení druhé série dvacátého devátého ročníku KSP

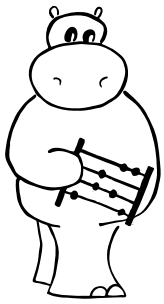
29-2-1 Cesty do školy

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 obslouží-me 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“) – zapamatování 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.[2]

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: $X \cdot X = X^2$, $X^2 \cdot X^2 = X^4$, …, 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`. 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.

[2] `http://ksp.mff.cuni.cz/viz/28-Z1-6/reseni`

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 Říša 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 přečíslování 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

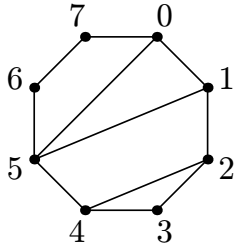
29-2-4 Nejsložitější záhon

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.

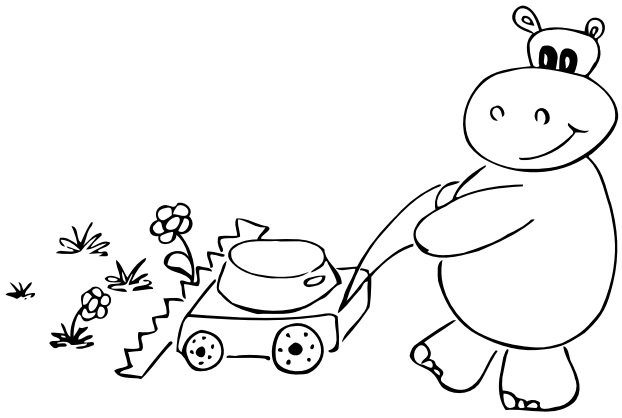
Každý chodníček nám bude na zásobníku záhonů zakládat nový záhon. Pokud ze stejného vrcholu vychází více chodníčků, tak chceme nejdříve založit ty záhony, které skončí později. Jinak řečeno na vršku zásobníku chceme ten ze záhonů, který skončí nejdříve – k tomuto záhonu budeme navíc ukládat vrcholy, 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.[3] 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.

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

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

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).

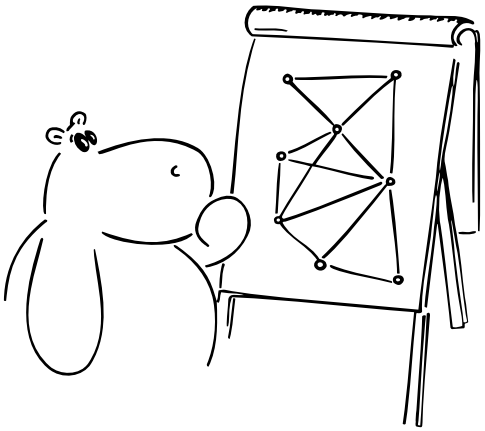
Podobně jednoduché to bude, i když nová oblast naváže na jednu oblast z předchozího řádku (nebo i více oblastí, ale všechny náležející té stejné nadoblasti), pak jen nové oblasti nastavíme stejné číslo nadoblasti a k nadoblasti připočteme velikost přidávané oblasti.

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,[4] 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.



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.

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

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 inů 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 iny 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 inů 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 inem a outem 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 inů 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 instrukci, 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 $\mathcal{O}((n+k)\log n)$.

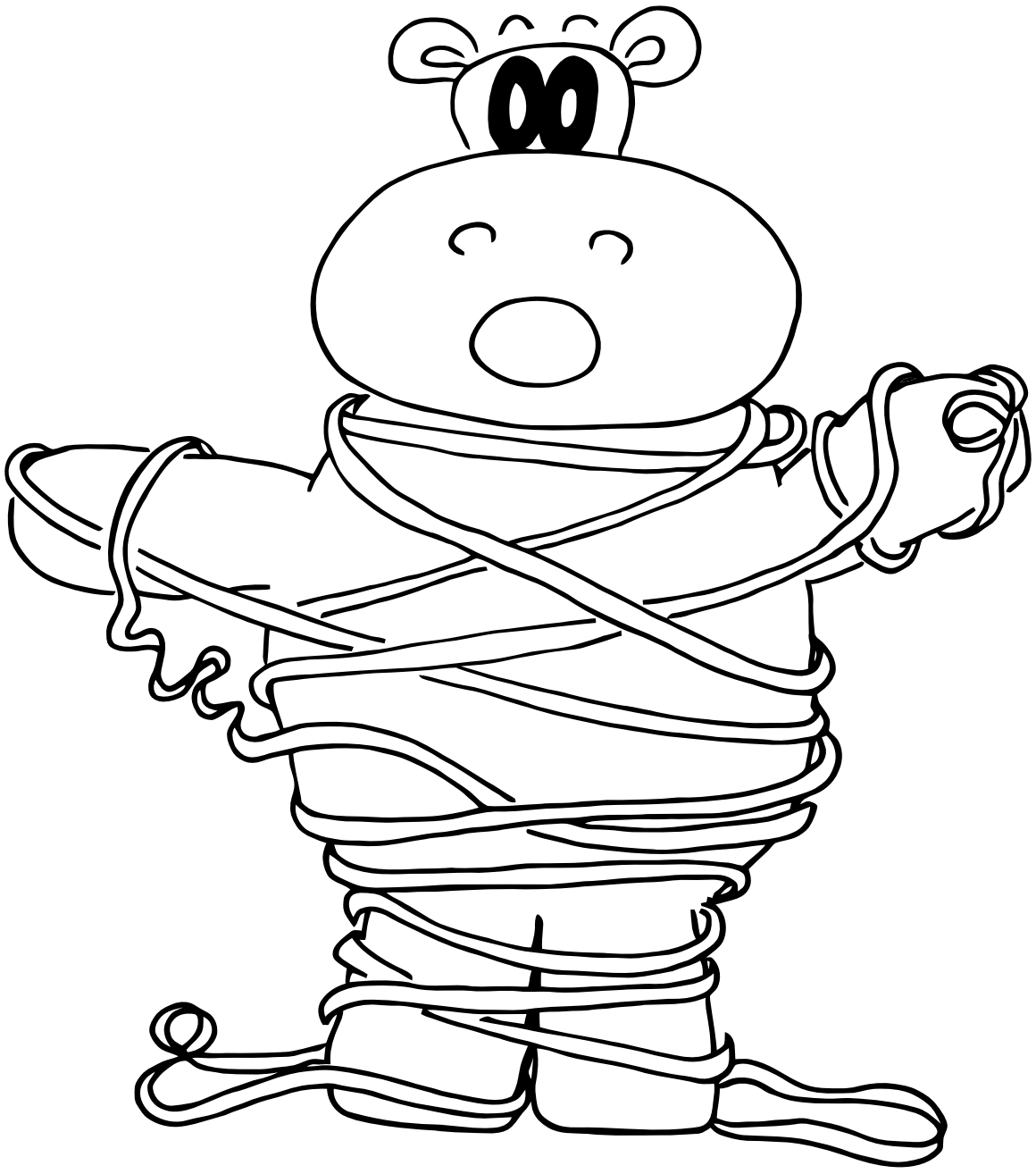
Úkol 4: Hrana mezi podstromy

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



Výsledková listina druhé série dvacátého devátého ročníku KSP

	řešitel	*škola*	*ročník*	*sérií*	*H2-1*	*H2-2*	*H2-3*	*H2-4*	*H2-5*	*H2-6*	*H2-7*	*série*	*celkem*
0.					10	13	13	9	10	11	15	62,0	120,0
1.	Richard Hladík	GOAMarLaz	4	22		13	13		10	11	15	62,0	113,6
2.	Lukáš Rozsypal	GÚstavníPH	4	5	4	12	7	9	6	1		44,9	93,2
3.	Tomáš Domes	MendelG_OP	4	3	10				9	1	5,5	29,1	73,2
4.	Roman Bujdák	G JM Galanta	3	2	10	5	6	2	7			36,8	68,3
5.	Peter Grajcar	GMetodovaBA	3	2	4	12	3					25,6	65,6
6.	Jonáš Fiala	GJungmanLT	4	7	8			9	9			27,2	56,0
7.	Jakub Pelc	G UherBrod	3	7		0					12	10,6	55,6
8.	Pavel Turek	GTomkovaOL	4	6								0,0	52,2
9.	Filip Geib	G MMH LM	3	4	4	6						15,2	52,1
10.	Martin Picek	GJirsíkaČB	2	1								0,0	47,0
11.	Jakub Pintera	SPŠ Prosek	4	1	4	6	5,5	9	5			43,4	43,4
12.	Rajmund Hruška	GPošKošice	4	1								0,0	43,0
13.	Matouš Mařík	G_Krumlov	4	1								0,0	40,7
14.	Pavel Turinský	G Brandýs	4	11	8	6				8	3	22,6	39,0
15.	Lukáš Caha	GZborovPH	3	2	4	2			5		0	19,0	38,7
16.	Tomáš Raunig	GHlu	2	2	4	5	2	1	1	2		26,7	34,3
17.	František Kmječ	G Brandýs	1	4		7			5		5	25,3	33,3
18.	Filip Masár	PiarGNitra	3	2				0				0,0	27,4
19.	Petr Gebauer	GMělník	3	2								0,0	26,8
20.	Michal Kodad	SPŠ_Smíchov	1	6			1				1	3,2	26,5
21.	Miroslav Hrabal	GTomkovaOL	3	3								0,0	25,6
22.	Václav Pavlíček	SPSE_Pard	1	6								0,0	25,5
23.	Anna Řechtáčková	GJarošeBO	4	2	4							6,8	22,7
24.	Kristián Jacik	GSRandyJN	4	1								0,0	22,6
25.	Ondřej Krsička	GJarošeBO	1	2	4			0				5,7	22,0
26.	Ondřej Gonzor	G Brandýs	0	1								0,0	18,8
27.	Anna Hollmannová	GSRandyJN	0	2	4							5,3	18,7
28.	Radek Olšák	MensaG	2	1								0,0	18,4
29.	Kryštof Mitka	ZŠUniverzum	0	2	10							10,0	17,4
30.	Jindřich Dítě	VOSPŠŽďár	1	1								0,0	15,6
31.	Ondřej Cach	SPSE_Pard	1	1								0,0	15,4
32.	Daniel Skýpala	GTomkovaOL	−1	1								0,0	12,5
33.	Vojtěch Hudec	G_ČTřebová	3	3								0,0	12,1
34.	Stanislav Lukeš	GPísnickáPH	4	12		2						1,6	11,9
35.	Vojtěch Lengál	GZborovPH	3	1								0,0	11,0
36.	Jan Kaifer	GKepleraPH	1	4				0				0,0	10,5
37.	Kateřina Čížková	G_Rokycany	3	1			5,5					8,8	8,8
38.	Adam Dřínek	GNAlejíPH	3	1								0,0	8,0
39.	Jiří Löffelmann	GLitoměřPH	3	5				0				0,0	7,9
40.	Jan Neumann	GNAlejíPH	3	2								0,0	7,7
41.–43.	Jakub Dobrý	GMikulášPL	3	4								0,0	7,6
	Anna Šebestíková	GČeskáČB	2	2								0,0	7,6
	Přemysl Šťastný	GŽamberk	4	14				1				0,5	7,6
44.	Michael Kozel	GZborovPH	3	1								0,0	7,5
45.	Jan Jeníček	GNAlejíPH	1	1								0,0	7,4
46.	Jakub Jirkal	GJungmanLT	2	1								0,0	7,2
47.	Jakub Spišák	G VBN Prie	4	1								0,0	7,0
48.–49.	Erik Kučák	GHorMichal	4	1								0,0	6,7
	Martin Miller	GVoděraPH	3	1				4				6,7	6,7
50.	Michal Töpfer	G DrJPekMB	4	11								0,0	6,6
51.	Eliška Vlčinská	GHladnov	2	2	4							6,3	6,3
52.	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`.