Výsledková listina druhé série dvacátého šestého ročníku KSP

	řešitel	*škola*	*ročník*	*sérií*	*2621*	*2622*	*2623*	*2624*	*2625*	*2626*	*2627*	*2628*	*série*	*celkem*
0.					7	8	10	10	12	12	13	14	61,0	113,0
1.	Martin Raszyk	G_Karvina	4	17			10	10	12		13	13,5	58,0	104,8
2.	Jan Špaček	G_Wicht	3	2	7	6	10	3	10	12		13,5	54,7	99,5
3.	Václav Rozhoň	GJirsíkaČB	3	3	6	8	10	10				14,5	49,2	95,9
4.	Matej Lieskovský	GOmskPha	4	12	6	8		8	12	12		13,5	53,2	91,2
5.	Marek Černý	G_Chrudim	3	2	6	3	10	9,5				12	45,3	87,6
6.	Michal Korbela	GJJesen	4	2	0		10	2		9	9		36,8	72,6
7.	Jakub Svoboda	GKomHavíř	4	7	6	8	0	6	1			7	32,1	70,5
8.	Michal Punčochář	GJírovcČB	4	12	6	3	10	10		12	13		50,9	66,9
9.	Richard Hladík	GOAMarLaz	1	7	6	6	2	10				5	32,9	65,8
10.	Aneta Šťastná	GOmskPha	4	8	6	8	0	6,5	12				33,7	65,1
11.	Štěpán Hojdar	GJírovcČB	4	7	6	2	10	7					27,3	60,1
12.	Jan Pokorný	G_Bučovice	2	3	7		0	10					17,0	59,1
13.	Jakub Zárybnický	GTomkovaOL	3	2	5	1	0	3		0		8	25,8	55,8
14.	Anna Steinhauserová	GDačice	4	2	3	6		7	1	0,5			25,1	55,6
15.	Štěpán Trčka	GSlavičín	3	9	3	3	3	7					18,2	54,3
16.	Filip Bialas	GOpatovPHA	1	2	2	8	2	10					26,1	50,4
17.	Antonín Češík	SPSE_Pard	4	2	7	5		10					23,9	49,2
18.	Jan-Sebastian Fabík	GJarošeBO	4	10	7	8	10	10				5	40,5	48,5
19.	Dorian Řehák	GCoubTábor	3	2		1		3		3		7,8	25,4	47,9
20.	Václav Volhejn	GKepleraPH	1	7	6,5		3	3,5				4,7	22,2	46,4
21.	Jan Knížek	G_Strakon	3	11	6	4						3,5	13,5	42,9
22.	Jonatan Matějka	SŠP_ČB	4	16	5,5		10						14,7	39,9
23.	Jakub Maroušek	G_Písek	4	6									0,0	33,5
24.	Lucie Studená	GKepleraPH	4	1	7	3		9,5	3	0	1	2	32,7	32,7
25.	Adam Španěl	ArcibisGPH	2	2	5	8		9					24,0	32,0
26.	Dominik Roháček	SPŠLegioJI	4	3	6								6,7	27,0
27.	Jan Pavlovský	GJiM	4	1									0,0	21,3
28.	Marek Dobranský	GHorMichal	4	6									0,0	20,3
29.	Ondřej Hübsch	GArabskáPH	4	20			10						10,0	18,0
30.	Dalimil Hájek	GKepleraPH	3	12			9						8,9	16,9
31.	Aneta K. Lesna	GZborovPH	1	1	2	2		3		0	1		16,8	16,8
32.	Michal Hloušek	GNadŠtolPH	1	1									0,0	16,3
33.	Antonín Teichmann	GJeronýmLI	4	1									0,0	15,2
34.	Petro Kostyuk	GEBenešeKL	4	2			3						5,7	12,1
35.	Radovan Švarc	G_ČTřebová	3	3									0,0	8,0
36.	Tadeas Friedrich	GOhradníPH	4	2	5								6,3	6,3
37.	Jan Horešovský	GMěl	4	2									0,0	6,2
38.	Michal Martinek	GHavPodl	3	1									0,0	6,0
39.	Marek Židek	GTomkovaOL	4	1	2								4,0	4,0
40.	Ladislav Tlapák	G_Břeclav	-1	1				1					2,5	2,5

Milí řešitelé, řešitelky a řešitelčata!

Vánoce už jsou za námi, pomalu začíná padat sníh, medvědi ulehli k zimnímu spánku a my vám přinášíme zadání třetí série letošního KSP. Těšit se můžete na pokračování napínavého příběhu i na další díl seriálu o výpočetních modelech. Také připomínáme, že se z každé série do celkového hodnocení započítává 5 nejlépe vyřešených úloh.

Nejprve ale chceme dát na vědomí, že před Vánocemi **vyšlo historicky první zadání KSP-Z**, neboli začátečnické kategorie KSP určené pro nováčky. Pokud tedy s hlavní kategorií KSP jen těžko zápasíte nebo máte kamarády, které by řešení úloh mohlo také bavit, můžete se na zadání KSP-Z podívat na našem webu. V nejbližších dnech bychom měli zveřejnit zadání druhé série.

Připomínáme, že každému řešiteli, který v tomto ročníku z každé série dostane alespoň 5 bodů, darujeme KSP propisku, blok a tužku. Dále se na vědomost dává, že **každému, kdo v této sérii z libovolných pěti úloh dostane alespoň polovinu možných bodů, pošleme čokoládu**.

Termín odevzdání třetí série je stanoven na **pondělí 3. února 2014 v 8:00 SEČ**. CodExová úloha má termín o den posunutý, opravuje ji totiž automat – odevzdejte ji do 4. února, 8:00 SEČ.

Řešení přijímáme elektronicky na stránce *https://ksp.mff.cuni.cz/submit/*. Chcete-li s námi komunikovat bezpečně, můžete si ověřit náš HTTPS certifikát – zde je jeho SHA1 fingerprint: `0E:D9:B6:E5:6F:B0:51:D9:66:EB:E9:29:E4:58:AB:5F:99:D6:FD:A3`.

Před tím ale vyplňte přihlášku na *http://ksp.mff.cuni.cz/* (a to i tehdy, když jste se KSP účastnili loni). Na tomtéž místě najdete i další informace o tom, jak KSP funguje. Na webu máme rovněž fórum, kde se můžete na cokoli zeptat. Také nám můžete napsat na e-mail *ksp@mff.cuni.cz*.

Třetí série dvacátého šestého ročníku KSP

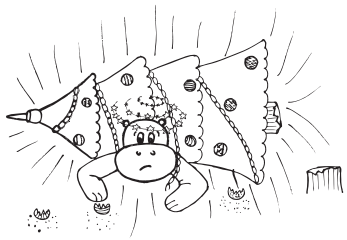
V předchozích dvou sériích jste mohli sledovat nouzové přistání vesmírné lodi Freya. *Jacob, její jediný přeživší, začal prozkoumávat neznámou planetu a pátrat po inteligentních formách života.*

Hrdina příběhu ve svém pátraní uspěl a objevil celý hlouček mimozemských bytostí. Neučinil to však příliš šikovně. Poté, co si jej jeden z tvorů všiml a upozornil ostatní, se rázem na Jacoba upínaly pohledy celého hloučku mimozemských bytostí . . .

* * *

Dva mimozemšťané vydali zvláštní vřískot, při kterém Jacobovi ztuhla krev v žilách. Rozhodl se nic neriskovat, prudce se otočil a přešel v trysk. Terén se zde prudce svažoval. Ozvaly se další vřískoty. Zněly tak hrozivě, že Jacob na chvíli přestal dávat pozor na své kroky a nohy se mu zamotaly do lián.

Udržet balanc se mu nepodařilo a pak už šlo vše velmi rychle. Jacob ještě stihl natáhnout ruce před sebe, z pádu se tak stal kotrmelec. První kotrmelec byl vystřídán druhým, mnohem rychlejším kotrmelcem. Nežli získal příležitost jakkoliv ovlivnit dráhu svého pohybu, už se řítil dolů ze zhruba desetimetrového srázu.



Po probuzení byl Jacob oslněn červeno-žlutými světýlky, odvrátil proto svůj pohled pryč. Viděl, že se nachází v místnosti se zelenými stěnami. V místnosti se mimo té, na níž ležel, nacházely další čtyři postele. Jednalo se vlastně spíše o lenošky nežli klasické postele, na jaké byl Jacob zvyklý. Jakmile Jacob pořádně zaostřil svůj zrak, zpozoroval, že neleží v klasické místnosti, nýbrž ve velmi honosném stanu.

Vrátil se pohledem ke stropu, teď už byl schopen si prohlédnout ona světýlka. Nebyla to světýlka, ale ozdobné drahokamy odrážející záři svící rozmístěných všude po stanu. Kromě svícnů se zde nacházely zdobené zlaté oštěpy, zlaté masky a totemy. Stan působil jako sídlo šamana.

26-3-1 Výklad z drahokamů — 11 bodů

Žluté a červené drahokamy se těšily velké oblibě a mimo ozdobných účelů se využívaly i k vykládání osudu.

Při takovém výkladu se drahokamy rozsypou na zem a uspořádají do obdélníkového tvaru. Protože všechny drahokamy mají zhruba stejné rozměry, vytvoří tak čtvercovou síť.

Následně se zkoumají všemožné pravidelnosti ve vzniklém obrazci. Zvláštní význam mají uspořádání, ve kterých se drahokamy střídají jako políčka na šachovnici. Vaším úkolem je proto najít největší takovéto uspořádání.

Nyní o něco formálněji: Je zadána tabulka znaků `.` a `#` o R řádcích a S sloupcích. Nalezněte největší čtvercovou oblast, ve které jsou znaky uspořádány jako na šachovnici. Tedy v této čtvercové oblasti nikdy nesousedí dva stejné znaky celou stranou, ale pouze rohem.

Příklad: Podívejme se na následující tabulku o šesti řádcích a sedmi sloupcích:

```
.#...#
#.##.#.
.#.#.##
#.#.#.#
...#.#.
#.#.#.#
```

Největší hledaná čtvercová oblast s šachovnicovým vzorem má stranu dlouhou čtyři a svůj levý horní roh má na třetím řádku a v třetím sloupci.

Teprve po prohlédnutí stanu si Jacob uvědomil, co se dělo před jeho probuzením. Radost z neporaněné hlavy a rukou vyhasla v momentu, kdy zjistil, že nohama nemůže nejen

pohybovat, ale dokonce je ani necítí. Nezbývalo než zůstat ležet a zaposlouchat se do venkovních zvuků.

Jacob věděl pouze to, že se nachází v mimozemském táboře a stará se zde o něj trojice léčitelů. Dobře jej živili a Jacob sílil. Pomocí posunků se s nimi i poměrně dobře dorozuměl.

Po zhruba měsíci donesli léčitelé Jacobovi berle. Vyrobeny byly z precizně leštěného dřeva. Dřevo připomínalo pozemský mahagon, jen bylo mechově zelené. Rukojeti byly omotány kožešinou a z boku berlí se nacházela spousta rytin vyobrazujících souboje mimozemšťanů s divou zvěří.

Pohyb s berlemi byl obtížný, protože Jacob nohy za sebou pouze vlačel. I tak měl ohromnou radost, když se mohl začít procházet mimozemským táborem. Při jedné z prvních procházek natrefil na stan zdejších švadlen.

26-3-2 Střih látky — 12 bodů

Jacob vypomáhá švadlenám v jejich práci. Jeho prvním úkolem je stříhání látky. Pozemský pomocník je však ukrutně nešikovný a švadlenám je pro smích.

Jacobovi se daří vést střih rovně, má však velmi špatný odhad. Látku prostě nikdy neustřihne tak, jak by si představoval. Pomozte mu!

Vašemu algoritmu bude předložen konvexní mnohoúhelník zadaný posloupností vrcholů (vrcholy budou zadány v pořadí, v jakém se vyskytují na obvodu mnohoúhelníka). Jeho úkolem pak je vybudovat si datovou strukturu, se kterou bude schopen efektivně odpovídat na střihové dotazy.

V rámci jednoho střihového dotazu bude zadána polopřímka, podél které bude veden střih. Úlohou datové struktury je určit průsečíky polopřímky s mnohoúhelníkem, pokud existují.

Následující čas v táboře ubíhal našemu hrdinovi jen pozvolna. Přes usilovnou práci léčitelů (nebo právě kvůli ní) trvalo dva roky, než byl Jacob schopen chodit bez berlí. I poté většina procházek po chvíli skončila ostrou bolestí. Až po zhruba třech a půl letech byl Jacob schopen chodit asi půl dne v kuse, než přišla ona bolest.

Jak Jacob celý ten čas trávil? Inu, zažil ještě vesmírné výpravy před kryogenickým spánkem, přečkat pár let s omezenými možnostmi pohybu a lidského kontaktu pro něj nebylo nic nového. Našel si celou paletu činností. Velké úsilí věnoval například luštění mimozemského jazyka a po zhruba třech letech byl schopen se s mimozemšťany běžně dorozumět. Zdálo se však, že čím lépe Jacob hovořil mimozemsky, tím menší chuť s ním hovořit mimozemšťané měli.

Vynechme drobné Jacobovy zážitky z těchto časů a přesuňme se raději do doby zhruba čtyř let od ztroskotání. Tehdy se Jacob poprvé odvážil vypravit pěšky až k vraku lodi UFC Freya*, kdysi věhlasné nákladní lodi Spojené federace. Loď zarůstala zelení, působila však poměrně celistvě. K výbuchu poškozeného hlavního reaktoru evidentně nedošlo.*

U trosek lodi potkal Jacob mimozemšťanku. Po krátké konverzaci se ukázalo, že toto není její první návštěva vraku lodi. Ada, jak se mimozemšťanka jmenovala, dle svých slov společně se svými mimozemskými kumpány získala velkou část lodní knihovny.

V luštění beletrie prý mimozemšťané příliš úspěšní nebyli. Na lodi se však nacházelo mnoho knih s informatickou a matematickou tématikou. Na základě obrázků pak odvodili význam většiny pozemské matematické notace.

Adě se prý obzvlášť líbila kniha o teorii grafů. Toho se Jacob rozhodl využít.

Při svém pobytu se bavil řešením spousty úloh, které si pamatoval ze Země, ale nikdy dřív na ně neměl dostatek času. Jednu úlohu řešil marně celé čtyři roky a třeba by Adu mohla napadnout právě ta myšlenka, která jemu unikala.

26-3-3 Grafová — 9 bodů

Nechť k je libovolné celé číslo větší než jedna. Uvažme graf, jehož všechny vrcholy mají stupeň alespoň k. Tím myslíme, že z každého vrcholu vede alespoň k hran.

Dokažte, že v takovém grafu existuje kružnice délky alespoň $k + 1$, a sestrojte algoritmus, který nějakou takovou kružnici najde.

Trvalo sice dlouho, než si vzájemně vyjasnili terminologii, nakonec se však dorozuměli a úlohu překvapivě svižně vyřešili.

Jacob byl setkáním s Adou doslova nadšen. Během pobytu v mimozemském táboře dávno přestal věřit, že by snad mohli existovat mimozemšťané se smyslem pro humor. Nebo nedej bože mimozemšťané, se kterými by se dalo jen tak volně povídat.

V průběhu roku se Jacob s Adou pravidelně scházeli. Zásadně vždy u onoho vraku. Tato setkání dávala Jacobovu životu na planetě smysl.

Jak se chýlil pátý rok soužití s mimozemšťany v táboře ke svému konci, dokázal Jacob pobíhat celý den po pralese, aniž by cokoliv cítil. Přišel čas zvažovat co dál. Rozhodně neměl chuť zůstávat u těchto prazvláštních mimozemšťanů. Sice se o něj pět let starali, nebyli však za celou dobu schopni si s ním rozumně promluvit a říci, co jsou zač. Na druhou stranu o Adě toho věděl ještě méně. Neměl ani to nejmenší tušení, kde a jak vlastně žije.

Během jedné noci tohoto období, kdy Jacob nevěděl kudy kam, jej ze spánku vyrušilo zašelestění. Nebral na něj žádný ohled a pouze se začal v posteli přetáčet na druhý bok. Uprostřed pohybu jej polekaly obrovské oranžové oči.

V úleku začal šmátrat po dýce, kterou měl vždy položenou u své postele. Než však pevně uchopil rukojeť dýky, rozpoznal v šepotu neznámé osoby Adin hlas.

Jacoba napadla celá řada dotazů. Než je stihl všechny vyslovit, Ada jej přerušila. „Není čas na dotazy," šeptala Ada, „Bratrstvo tě potřebuje."

Jacob se omezil na jediný dotaz. Jak se bez povšimnutí a bezpečně dostanou pryč z tábora? Kolem tábora byla totiž rozmístěna spousta pastí pro zneškodnění vetřelců neznalých terénu. V noci byly tyto pasti obzvláště zákeřné.

Na to měla Ada prostou odpověď. Ukázala Jacobovi pečlivě vypracovanou mapku rozmístění těchto pastí po okolí tábora.

26-3-4 Kladení pastí — 10 bodů

V této úloze budete navrhovat algoritmus pro hledání optimálního rozmístění pastí v pralese.

Prales je neprostupný a pohybovat se lze prakticky jen po vyšlapaných pěšinkách. K dispozici máte mapu terénu. Na mapě se nachází N křižovatek a M pěšinek. Každá pěšinka spojuje dvě křižovatky.

Pro každou dvojici křižovatek existuje právě jedna cesta (posloupnost pěšinek), po které se lze přemístit z první křižovatky na druhou. V informatické řeči bychom řekli, že mapa je *stromem*.

Pasti umisťujeme do křižovatek. Vyžadujeme, aby pro každou pěšinku platilo, že na alespoň jedné z křižovatek, které spojuje, leží past.

Spojením kopírovacího a porovnávacího programu (spojení můžeme provést, protože vstup obsahuje znak `#`, který druhý program nepoužívá) vznikne program řešící celou úlohu. Jelikož při přesouvání provedeme pro každý znak lineárně mnoho kroků vzhledem k délce vstupu a přesunů je lineárně mnoho, je výsledná složitost $\mathcal{O}(N^2)$, kde N je počet znaků na vstupu.

Úkol 4

Jelikož jsme měli Turingovy stroje v minulé sérii zadané tak, že vracely pouze výsledek ANO nebo NE (stavem, ve kterém skončily), tak se na tento výstup omezíme i u přepisovacích pravidel – na výstupu tedy zanecháme buď řetězec `ANO` nebo řetězec `NE`.

V případě, že by se řetězce `ANO` nebo `NE` vyskytovaly na vstupu nebo někde v průběhu výpočtu, odlišíme ty koncové tím, že pro ně použijeme speciální symboly.

Když jsme si odbyli technické detaily, pojďme se podívat na hlavní myšlenku převodu Turingova stroje na přepisovací pravidla. Následující krok Turingova stroje je vždy určený pozicí hlavy, stavem, ve kterém se stroj nachází, a aktuálním obsahem pásky. Tuto informaci budeme potřebovat nějak kódovat v řetězci.

Pozici hlavy tedy budeme reprezentovat speciálním znakem někde v řetězci. Přesněji více speciálními znaky – pro každý stav stroje jeden znak. Zaveďme si pozici hlavy tak, že tento speciální znak bude stát před znakem, na který by ukazovala hlava v Turingově stroji.

Každé pravidlo Turingova stroje má tyto části: vstupní stav, vstupní znak, výstupní stav, výstupní znak a výstupní posunutí. To ale můžeme velmi lehce přepsat do řeči přepisovacích pravidel. Například pravidlo pro stroj říkající „Pokud jsi ve stavu S a na pásce je x, přejdi do stavu T, zapiš na pásku y a posuň se doprava" bychom mohli napsat jako: `Sx` → `yT`.

Drobným problémem je posouvání doleva (vzhledem k tomu, že náš znak pro hlavu stojí před přepisovaným znakem). Pro posun hlavy doleva si musíme vyrobit jedno pravidlo pro každý znak abecedy. Když bude `#` zastupovat libovolný znak abecedy, vypadá pak stejné pravidlo, jen s posunem hlavy doleva, takto: `#Sx` → `T#y`.

Tím máme hlavní část převodu hotovou. Stačí vyřešit pouze tři drobné technické detaily:

- Turingův stroj počítá s tím, že má nekonečnou pásku plnou mezer, kdežto náš řetězec je omezený. To však vyřešíme jednoduše tím, že pokud se hlava dostane na začátek nebo na konec řetězce, tak přidáme mezeru. Pro všechny stavy `S` budeme tedy mít následující pravidla (tato pravidla musíme v programu umístit nad všechna ostatní):

  `^S` → `␣S`

  `S$` → `S␣`

- Druhým detailem je to, že musíme řetězec na konci výpočtu smazat a zanechat zde jen `ANO` nebo `NE`. To však lehce zařídíme nějakými mazacími pravidly.
- Poslední drobnost je, jak celý výpočet odstartovat, tedy jak na začátku umístit do řetězce symbol hlavy? Kdybychom měli jen pravidlo `^` → `S`, tak by nám (minimálně po skončení výstupu) program začal běžet znovu.

  Zabráníme tomu tak, že si zajistíme, že tato náhrada proběhne jen a pouze na začátku programu. Nejprve potřebujeme, aby se ani `ANO` ani `NE` nevyskytovaly na vstupu, ale to jsme si už ošetřili výše. Zkonstruujeme si tedy následující pravidla pro všechny znaky vstupní abecedy (kde `#` zastupuje libovolný znak abecedy na vstupu, `S` je počáteční stav a `@` je nějaký speciální znak, který není jinde použitý):

  `^#` → `@S#`

  `@ANO` → `ANO`

  `@NE` → `NE`

  První pravidlo nám zajistí to, že k přidání počáteční pozice hlavy už nikdy znovu nedojde (protože neexistuje přidávací pravidlo s `^@` na levé straně), a druhé a třetí pravidlo nám odstraní tento speciální znak z výstupu. Také je nutné upravit pravidlo na přidávání mezer na začátek řetězce, aby zachovávalo pozici `@`, ale to je už maličkost. Převod je tedy tímto hotov.

Úkol 4 – dodělávky

Zbývá zamyslet se nad složitostí. Na každou aplikaci pravidla Turingova stroje provedeme právě jednu aplikaci přepisovacího pravidla, takže by se mohlo zdát, že složitost obou bude stejná. Přepisovací programy ovšem musí na konci výpočtu smazat celý řetězec, takže pokaždé běží alespoň lineárně s délkou vstupu (kdežto Turingův stroj může skončit třeba po první aplikaci pravidla přechodem do koncového stavu).

Můžeme tedy prohlásit, že přepisovací program má stejnou složitost jako Turingův stroj, vždy ale nejméně lineární s velikostí vstupu.

Pokud bychom měli převod demonstrovat na příkladu s vyváženou posloupností z prvního dílu seriálu, tvořil by hlavní část programu pouze mechanický přepis pravidel Turingova stroje na pravidla přepisovacího programu. To jistě každý zvládne sám.

Mimo nich budeme potřebovat pravidla, která nám na začátku vloží do řetězce hlavu, a pravidla, která nám na konci vymažou zbylé znaky.

A to je milí přátelé z druhé série vše. Přeji vám hodně zdaru i v dalších dílech seriálu při potýkání se s dalšími zajímavými výpočetními modely.

Jirka Setnička

aby se přepis nemohl opakovat po doběhnutí druhého programu). K tomu nám stačí libovolný jeden znak na vstupu, který se nevyskytne v průběhu výpočtu druhého programu, ani na jeho výstupu. Pomocí něj provedeme úvodní přepis.

V praxi budeme chtít mít všechna překladová pravidla na začátku přepisovacího programu. Spouštět je budeme jedním přepisovacím pravidlem na správném místě programu, které nám přidá do řetězce speciální přepisovací znak (rozmyslete si, proč to děláme takto a proč nemůžeme všechna pravidla přesunout na místo spouštěcího pravidla).

Výsledné spojení programů pak vypadá takto:

- Pravidla pro převod abecedy $A \to A'$ využívající nějaký speciální znak ze vstupu
- Pravidla pro převod abecedy $A' \to A$ využívající znak α
- První program (s převedenou abecedou)
- Pravidlo přidávající α (zajistí překlad abecedy)
- Druhý program

Umíme tedy spojit libovolné dva programy do jednoho, pokud se na vstupu prvního vyskytuje alespoň jeden znak, který v druhém programu použitý není.

Úkol 3

Poznatek o spojování programů použijeme pro řešení třetí úlohy. Idea bude taková, že si vstup nafoukneme o zkopírované verze obou čísel. První číslo opíšeme normálně, druhé číslo pozadu (aby se nejnižší bity obou čísel dostaly k sobě).

Taky předpokládejme to, že čísla nejsou prefixována nulami. Pokud by byla, můžeme přebývající nuly snadno odmazat. Stačí si zavést dvě pomocná pravidla pro levé a pro pravé číslo.

Tedy ze vstupu ve tvaru „`1010#1100`“ vyrobíme přepisem „`1010_1010=0011_1100`“. S touto verzí pak provedeme porovnání (postupným odmazáváním nejmenších bitů) a zjistíme, jestli je větší pravé nebo levé číslo. Druhé pak vymažeme a jsme hotovi.

Nejprve se zabývejme **rozkopírováním**. Ukážeme pravidla pro kopírování pravé části pozpátku. Druhá sada pravidel pro kopírování levé části bude podobná.

Hlavní idea je, že si vyrobíme na koncích jakési zarážky a postupně budeme znak po znaku pomocí *vozíku* (znaku `v`) převážet zkopírované znaky na správné místo. Následující sada pravidel nám přepis odstartuje:

```
#  → y_=_Y
Y0 → 0Y
Y1 → 1Y
Y$ → X
```

Tím nám vznikne řetězec (zajímejme se teď jen o pravou stranu od `=`) ve tvaru `_Y...X`. Teď můžeme začít převážet znaky. Vždy nabereme jedno číslo u koncové značky, posuneme koncovou značku o jedna doleva a pomocí *vozíku* převezeme číslo těsně za `_`. Ve chvíli, kdy nemáme žádný vozík, si ho pořídíme a ve chvíli, kdy pravý ukazatel dojde až k `_`, ukazatel zahodíme.


```
0v0 → v00
1v0 → v01
0v1 → v10
1v1 → v11
_v0 → 0_
_v1 → 1_
 1X → v1X1
 0X → v0X0
 _X → _
```

Tím jsme si vytvořili přepisovací program, který nám předzpracuje vstup. Nyní si vytvořme program, který nám zajistí samotné **porovnání čísel**.

Budeme postupně odebírat nejméně významné bity z obou čísel a porovnávat je. Aktuální stav porovnání si budeme držet mezi oběma čísly – jako `=`/`>`/`<` podle toho, jestli jsou čísla zatím stejná, je větší levé, nebo je větší pravé.

Je jasné, že pokud budou na obou stranách stejná čísla, stav (ať bude jakýkoliv) se nám nezmění. Jinak se nahne na tu stranu, kde je aktuálně větší číslo (významnější bit převáží nad těmi méně významnými). Pro stejně dlouhá čísla tedy máme následující pravidla:

```
1=1 → =
0=0 → =
1=0 → >
0=1 → <
1>1 → >
0>0 → >
1>0 → >
0>1 → <
1<1 → <
0<0 → <
1<0 → >
0<1 → <
```

Samostatně vyřešíme případ, že je jedno z čísel kratší jak druhé – pak je to delší určitě větší (ukážeme pravidla pro `>`, pro `<` jsou analogická):

```
1>_ → >_
0>_ → >_
1<_ → >_
0<_ → >_
0=_ → >_
1=_ → >_
```

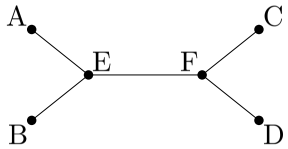
Uprostřed nám zůstal jeden znak udávající, které z čísel je větší. Nyní stačí to menší vymazat (opět ukážeme jen pro `>`):

```
_>_ → Q
 Q1 → Q
 Q0 → Q
 Q$ →
```

Umístit past na křižovatku nemusí vždy stát stejné úsilí. Pro každou křižovatku máte zadánu hodnotu U, která modeluje míru vynaloženého úsilí.

Navrhněte algoritmus, který nalezne rozmístění pastí do křižovatek tak, aby každá pěšinka měla na alespoň jednom ze svých konců past. Součet hodnot U všech křižovatek, na které algoritmus umístí past, musí být minimální možný.

Příklad: Představte si mapu cestiček jako na následujícím obrázku. V případě, že by umístění pastí stálo na všech křižovatkách stejné úsilí, bylo by nejlepší umístit pasti na křižovatky E a F, čímž bychom pokryli všechny cestičky.



Pokud by však ohodnocení úsilí vypadalo jako $U(A) = 1$, $U(B) = 1$, $U(C) = 2$, $U(D) = 3$, $U(E) = 3$, $U(F) = 4$, bylo by nejvýhodnější umístit pasti na křižovatky A, B a F za celkově 6 jednotek úsilí. Žádné jiné rozmístění pokrývající všechny cestičky nemá menší cenu.

Lehčí varianta (za 3 body): Vyřešte úlohu pro případ, kdy křižovatky budou tvořit jenom jednu nerozvětvenou cestu.

Ada s Jacobem pod hávem noci úspěšně opustili tábor a vydali se na cestu. Po cestě se Ada podělila o několik základních informací.

Ada je příslušnicí tajného společenstva s názvem Podzemní bratrstvo. Není mu však oprávněna v tuto chvíli prozrazovat cokoliv o poslaní Bratrstva. Bratrstvo má po okolí rozesetu síť základen. Všechny základny jsou řízeny z Hlavního štábu, kam právě směřují.

Hlavní štáb zabírá většinu rozsáhlého jeskynního komplexu, v řeči mimozemšťanů zvaného Ĝesrít íhgišpoř kiĝes sda ƥkuteraƥ. V posledních dnech se objevil problém, se kterým si Bratrstvo není schopno poradit. Chce proto požádat o pomoc Jacoba a využít jeho pozemských znalostí. Podrobnosti není Ada oprávněna sdělovat, žádost o pomoc chce vyslovit osobně Rada stařešinů.

Po poledni dorazili na pralesní mýtinku. Krom zpěvu ptáků neupoutalo nic Jacobovu pozornost. Ada na správném místě odhrnula listí a objevil se mřížový poklop. Poklop chránil úzký otvor, kterým se oba protáhli do jeskyně. Po průchodu takřka třísetmetrovou úzkou chodbou se objevili v ohromném dómu.

Ada pokynula hloučku mimozemšťanů, ať Jacobovi ukážou dóm. Sama zmizela v další chodbě vycházející z dómu hledajíc stařešiny. V dómu se nacházely desítky stanů, obydlí členů Bratrstva. Za stany se nacházel plácek sloužící jako shromaždiště. Zde byli každé ráno členové Bratrstva informováni o všem potřebném.

Dobrou čtvrtinu dómu zabírala mimozemská kovárna. Sestávala z asi tří výhní a dobrých dvou tuctů kovadlin. Mezi právě kovanými předměty Jacob zahlédl meč, lopatu, ba dokonce i svícen.

Jacob byl uchvácen sehraností mimozemšťanů, kladiva svými pravidelnými údery spíše než jako pracovní nástroje působila jako malý orchestr.

26-3-5 Rozvrh kovárny — 11 bodů

V této úloze se budete zabývat plánováním práce mimozemské kovárny vyrábějící všechny potřebné kovové nástroje.

Základny posílají kovárně své požadavky. Parametry požadavku jsou druh nástroje, priorita P a hodina H, do které je základna ochotna čekat. Než by dostala nástroj po hodině H, raději se zařídí jinak. Hodnoty P a H jsou kladná celá čísla.

Pro zjednodušení předpokládejme, že výroba jakéhokoli nástroje trvá jednu hodinu a během této doby nelze vyrábět nic jiného. Výroba začíná v hodinu 0, tedy na konci této hodiny může být vyroben první výrobek.

Na vstupu dostanete N požadavků, každý požadavek je určen dvojicí hodnot P a H. Protože výroba jakéhokoli nástroje trvá hodinu, typ nástroje se na vstupu vůbec neobjeví.

Váš algoritmus má za úkol sestavit optimální rozvrh výroby. To znamená pro každý požadavek určit hodinu, během které se požadovaný nástroj bude vyrábět, případně -1, pokud požadavek nebude splněn vůbec. Výrobek může být vyroben nejpozději v hodinu H.

Rozvrh je optimální, pokud součet priorit splněných požadavků je nejvyšší možný.

Příklad: Pro požadavky $(4, 4)$, $(1, 1)$, $(2, 2)$, $(2, 3)$, $(4, 4)$ (zadané v pořadí priorita, hodina nutného dokončení) je jednou ze správných odpovědí $3, -1, 1, 0, 2$. Tedy vyrobíme předměty s celkovou prioritou 12 a druhý předmět nevyrobíme vůbec.

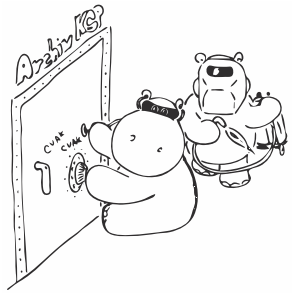
Z jedné z mnoha chodeb ústících v dómu se vynořila Ada následovaná trojicí mimozemšťanů. Nebylo pochyb o tom, že se jedná o stařešiny. Podsaditý mimozemšťan se představil jako Ubu, Tajemník Bratrstva a Vrchní stařešina. Přivítal Jacoba a pozval jej do svého stanu.

Jacob byl pohoštěn dobrým jídlem a dozvěděl se, že za hodinu bude oficiálně přivítán. Po hodině zdvořilé konverzace se rozezněly fanfáry. Když vyšli z Ubuova stanu, na shromaždišti už postávaly hloučky mimozemšťanů.

Ubu před davem představil Jacoba a oznámil, že Jacobovi bude jako výraz úcty Bratrstva předán dar. Asi dvacítka statných mimozemšťanů po těchto slovech donesla ohromný trezor.

Ke třem stařešinům, které už Jacob poznal, se přidalo dalších devět. Všichni stařešinové se shromáždili u trezoru.

Ke zdárnému otevření trezoru bylo nutno podniknout sérii přesně daných kroků. Nejprve bylo nutno ovládací páku přepnout do spodní polohy. Čtveřice mimozemšťanů poté asi minutu točila obrovitou klikou.



Stařešinové přistoupili k trezoru a každý vložil svůj klíč do otvoru v trezoru. Páka byla přepnuta do své horní polohy a čtveřice mimozemšťanů točila další dvě minuty klikou. Ozvalo se cvaknutí.

Další čtveřice mimozemšťanů otevřela těžké víko trezoru.

26-3-6 Trezor — 8 bodů

V této úloze budeme zkoumat mimozemský trezor. K otevření trezoru je potřeba do některých otvorů z vnějšku trezoru vložit sadu klíčů. Pokud je sada klíčů správná (správných sad může existovat více), trezor se po dostatečně dlouhém točení klikou otevře.

Princip trezoru tkví v tom, že všechny klíče odpovídají mocninám dvojky. Trezor má navíc ve svých útrobách skrytou druhou sadu klíčů. (Tvůrce trezorů totiž vyrábí všechny trezory stejně a až podle potřeb kupce navolí tyto vnitřní klíče.)

Během otáčení kliky vnitřní mechanická sčítačka sečte dohromady všechny klíče, jak vnitřní, tak ty zasunuté zvnějšku.

Je-li výsledný součet při zápisu v dvojkové soustavě tvořen samými jedničkami (na úvodní nuly se ohled nebere), trezor se otevře.

Vaším úkolem je pro zadanou N-tici vnitřních klíčů určit nejmenší počet vnějších klíčů potřebných k otevření trezoru. Klíče jsou zadány exponenty a nemusí být různé.

Poznámka: Možná jste se ještě nezabývali výpočetními modely a otázkou toho, o kterých operacích můžete prohlásit, že proběhnou v konstantním čase. Pak by vás mohlo napadnout umocnit dvojku na zadané exponenty a dále s nimi počítat. Vězte však, že výpočetní model, ve kterém bychom mohli v konstantním čase provádět aritmetiku nad libovolně velkými čísly, by byl absurdně silný.

Takový model by už neměl mnoho společného s tím, jak lze využívat skutečné počítače. Nejčastěji se proto uvažují modely, ve kterých lze v konstantním čase počítat pouze s čísly, která jsou polynomiálně velká vzhledem ke vstupním číslům. Takový model uvažujte i při řešení této úlohy.

Příklad: Pro seznam exponentů `0 1 0 1 0 2 4 4` lze trezor otevřít třeba pomocí pětice klíčů 2^1, 2^1, 2^3, 2^3 a 2^6.

To však není správná odpověď. Když použijeme pouze klíče o hodnotách 2^2 a 2^4, bude součet hodnot všech klíčů 63. To je ve dvojkovém zápise číslo `111111` a trezor se také otevře. Použít pouze jeden klíč pro zadaný příklad již nepostačuje.

Lehčí varianta (za 2 body): Vyřešte úlohu pro případ, kdy všechny exponenty vnitřních klíčů budou různé.

Ubu z trezoru vytáhl meč, došel k Jacobovi, poklekl a v natažených rukou mu nabídl tento dar. Za potlesku davu se stal Jacob majitelem mimozemského meče.

Po ceremoniálu se Jacob, tentokrát společně se všemi stařešiny, přesunul zpátky do Ubuova stanu. Ubu Jacobovi stručně povyprávěl o historii meče. Pak najednou zvážněl. Všichni stařešinové si sesedli blíž k Jacobovi. Zjevně nadešel čas poodhalit Jacobovi účel zdejší návštěvy.

„Drahý příteli," začal svou řeč Ubu, „skutečnost je prostá. Bratrstvu hrozí zkáza. Během posledního týdne se začal probouzet vulkán nedaleko Hlavního štábu. Pokud se potvrdí nejhrozivější obavy našich šamanů, láva z vulkánu zaplaví tento jeskynní systém. Nemáme dostatek jiných jeskyní, kam bychom přesunuli všechen náš lid, a na povrchu by jej stihla zkáza. Žádáme tě jménem celého našeho lidu o pomoc."

„Drazí přátelé," volil Jacob opatrně svá slova. „Učiním vše, co bude v mých silách. Obávám se však, že toho nebude mnoho. Má loď ztroskotala a nezdá se, že by jakákoli technologie z ní byla použitelná. Rád bych se však na onen vulkán alespoň podíval."

„Dobrá," odvětil trochu zklamaně Ubu a na chvíli se zamyslel. „Vyšlu s tebou své nejlepší lidi, budou ti ve všem nápomocni."

O pár hodin později stál Jacob spolu s Adou a dalšími čtyřmi mimozemšťany u úpatí sopky. Z jejího vrcholku pomalu stoupaly malé obláčky dýmu.

Jacob se zamyslel. Co od něj Bratrstvo vůbec očekává? Jistě, na Zemi dávno existovala technologie, která se s touto hrozbou vypořádala. Co si však lze počít zde? Zdálo se, že sopka lehce smete Bratrstvo. Neměl pocit, že by si to nechala jen tak vymluvit...

26-3-7 Sopečné pokrytí — 12 bodů

Pro zabezpečení sopky je na Zemi nutno podniknout dvě opatření – zahájit odčerpávání lávy ze sopouchu speciálním potrubím a pokrýt ústí sopky panely tak, aby neunikal sopečný popel ani gejzíry lávy. Vaším úkolem bude nalézt optimální pokrytí.

Okolí sopky si můžeme představit jako čtvercovou síť velikosti $R \times S$. Na každém políčku se nachází buď znak `Z`, nebo `K`. Pomocí `Z` je označena zem, kterou není potřeba pokrývat, naopak znaky `K` označují oblast sopečného kráteru, kterou je nutno pokrýt. Oblast kráteru je souvislá.

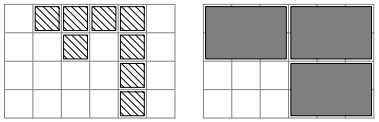
V celé úloze budeme za sousední políčka považovat ta, která se dotýkají celými stranami. Souvislou oblastí pak rozumíme množinu políček takovou, že kdykoli máme políčka A, B z této oblasti, existuje posloupnost políček z této oblasti s následujícími vlastnostmi: prvním políčkem je A, posledním B a pro každé políčko (mimo posledního) platí, že následující políčko je jeho sousedem.

Pokrývá se panely obdélníkového tvaru rozměrů $\varrho \times \sigma$. Pro zjednodušení úlohy budeme předpokládat, že panel lze položit pouze tak, aby pokryl obdélníkovou podoblast čtvercové sítě o ϱ řádcích a σ sloupcích.

Aby celé pokrytí fungovalo, musí panely do sebe zapadnout. To znamená, že pokud spolu dva panely sousedí nějakými políčky, pak spolu musí sousedit celými svými stranami. Panely mohou sahat i za hranice původní mapy.

Nalezněte panelové pokrytí, které pokryje všechna políčka se znakem `K` a využije k tomu nejmenší možný počet panelů.

Příklad: Na obrázku níže vidíte jedno z možných pokrytí kráterů vlevo pomocí panelů o rozměrech 2×3.



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

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

26-2-8 Továrna na přepisování

Druhou série sice neřešilo tak moc lidí, jako sérii první, ale i tak jsem byl potěšen tím, kolik vašich řešení přišlo.

Úkol 1

Na řešení první úlohy bylo možné jít několika směry. Ukážeme si zde přepisovací program, který postupně umazává stejná čísla a zvedá první z nich. Lehce nahlédneme, že nahrazením dvou stejných čísel vedle jedním (stejným) číslem nic nepokazíme. Neklesající posloupnost zůstane neklesající a klesající také nepřestane klesat.

Pak nám už stačí jen zvedat číslo na první pozici postupně až do devítky. Po každém zvýšení nám tak potenciálně vznikne nová dvojice stejných znaků na začátku, které srazíme na jeden. Pokud bude posloupnost neklesající, tak nám tento postup v průběhu vymaže všechna čísla až na poslední. Pokud se tak stane, zahlásíme úspěch přepsáním na `ANO`.

Pokud je však posloupnost někde klesající, tak první číslo zvýšíme až na devítku a ještě nám za ní něco zbude. V takovém případě se jen zbavíme zbytku vstupu a na výstupu zanecháme `NE`.

Takový přepisovací program je uveden níže. V každém kroku (až na konstantně mnoho zvýšení prvního čísla) jedno číslo umaže, tedy běží v lineárním čase k velikosti vstupu.

```
00 → 0
 ⋮
99 → 9
^0 → 1
 ⋮
^8 → 9
^9$ → ANO
^9 → X
X0 → X
 ⋮
X9 → X
 X → NE
```

Alternativním postupem může být naopak detekovat chyby. Neboli měli bychom pravidla pro jakoukoliv dvojici sousedních čísel ve špatném uspořádání (takových je 45) a tuto dvojici bychom přepsali na nějaký chybový znak. Potom bychom vymazali všechna čísla a pokud by nám na konci zůstal nějaký chybový znak, vypsali bychom `NE`. Tento postup má také lineární složitost, ale potřebuje o něco více přepisovacích pravidel.

Úkol 2

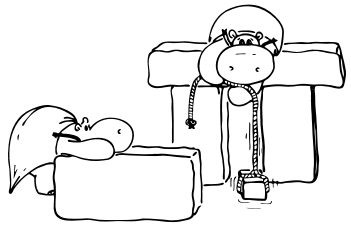
Budeme triviálně realizovat binární sčítačku. Budeme si ji držet vlevo od hvězdiček tak, aby hvězdičky přiléhaly k nejnižšímu bitu čísla. To nám umožní za každou hvězdičku k tomuto bitu přičíst +1.

Jak ale zajistit přenosy do vyšších řádů? Jednoduše, v momentě, kdy bude potřeba provést přenos, zapíšeme namísto přenosu nějaký symbol (z logiky sčítání se nabízí použít třeba symbol 2). A poté ho pomocí dalších pravidel posuneme do vyššího řádu (tedy z 1011+1 se stane 1012 a po přenosu nejprve 1020 a pak 1100).

Realizace s pomocí pravidel (a s ošetřením úvodní inicializace počítadla a prázdného vstupu) vypadá následovně:

```
^$ → 0
^* → 0*
0* → 1
1* → 2
^2 → 10
02 → 10
12 → 20
```

Kolik kroků provedeme? Odstranění hvězdičky provedeme právě tolikrát, kolikrát byla na vstupu. Mohlo by se však zdát, že nám složitost pokazí přenosy. Stačí si ovšem uvědomit, že přenos z posledního bitu provedeme v každém druhém kroku, přenos z předposledního bitu jen v každém čtvrtém kroku a tak dále. Když posčítáme všechny přenosy, vyjde nám, že jich provedeme maximálně $2N$, tedy složitost je stále lineární k délce vstupu. Pro podrobnější důkaz amortizace časové složitosti binární sčítačky nahlédněte na konec řešení minulého dílu seriálu.[11]



Skládání programů

Pro řešení třetího úkolu si nejdříve ukážeme, jak za sebe složit libovolné dva přepisovací programy. Mějme dvě sady přepisovacích pravidel, z kterých chceme vyrobit třetí sadu tak, že nám bude vracet stejný výsledek, jako kdybychom výstup prvního přepisovacího programu použili jako vstup pro druhý.

Co se stane, pokud oba programy jen napíšeme pod sebe? Jelikož aplikace pravidel probíhá postupně, tak dokud běží první přepisovací program, nemůže se provést žádné pravidlo z druhého přepisovacího programu. Problém ale nastane ve chvíli, kdy se výpočet přehoupne do druhého programu. Jak zajistit, aby se nepoužilo žádné pravidlo z prvního programu?

Kdyby programy používaly úplně jiné sady znaků, bylo by to jednoduché (metaznaky začátku a konce řetězce můžeme také odlišit a to třeba tak, že k nim vždy přilepíme ještě nějaký další speciální znak). Tak si to pojďme zařídit.

Všechna pravidla prvního programu přeložíme tak, aby používaly nějaké jiné znaky (třeba ty samé, ale s čárkou). Poté jen potřebujeme přidat překlad znaků před vstupem do druhého programu, který je přepíše zpět na původní znaky.

Druhou nutnou věcí je přepis vstupní abecedy na čárkovanou verzi před vstupem do prvního programu (ale tak,

[11] `http://ksp.mff.cuni.cz/viz/26-1-8/reseni`

můžeme vygenerovat vícekrát – ale všimněme si, že každou nejvýš čtyřikrát (např. `ab` ve slově `aabb`), navíc díky značkování vrcholů nás každé toto opakování stojí jen konstantní čas. Opakování se dá zbavit úplně, pokud si k redukovanému slovu zapamatujeme ještě něco navíc (snadné cvičení).

- Spárovaného souseda dané zkratky nalezneme prostým přečtením příslušné hodnoty ze slovníku.

Prostřednictvím těchto informací již dokážeme „simulovat“ náš graf s logaritmickým či průměrně konstantním zpomalením. Vzhledem k tomu, že žádné párování není větší než N, spotřebujeme $\mathcal{O}(N)$ paměti na pomocné slovníky, ale nesmíme zapomenout na $\mathcal{O}(S)$ pro vstup, tedy celkem $\mathcal{O}(S)$. Výsledný algoritmus poběží v čase $\mathcal{O}(S + N^3 \log N)$.

Hopcroft-Karp a velké abecedy

I pro velké abecedy bychom rádi použili Hopcroft-Karpův algoritmus. Ovšem jen pokud na něm dokážeme provést optimalizace obdobné těm výše – v původní podobě by byl pomalejší než předchozí algoritmus.

Hopcroft-Karp se skládá z *fází*: v každé najde v inkluzi maximální množinu disjunktních nejkratších zlepšujících cest a všechny použije ke zvětšení párování. Provádí to tak, že si nejdřív pomocí BFS rozdělí vrcholy grafu do vrstev podle délky nejkratší střídavé cesty vedoucí do nich z nějakého volného vrcholu levé partity a pak už jen pomocí DFS vysbírává nejkratší cesty a značkuje vrcholy, aby žádný nepoužil dvakrát. Obě prohledávání začínají ve volných vrcholech levé partity.

Kromě grafu samotného, na který si vystačíme se strukturami popsanými dříve, si navíc potřebujeme pamatovat ještě:

- BFS ohodnocení (rozdělení do vrstev)
- DFS značky (označující již navštívené vrcholy, abychom se do nich nevraceli)

Obojí budeme uchovávat ve slovnících a s každou částí se vypořádáme trochu jinak.

BFS začínáme v zajímavé části grafu. Jakmile poprvé navštívíme nezajímavý vrchol, prohledávání ukončíme. Tak určitě doběhne v $\mathcal{O}(N^2)$, jen některým vrcholům v ℓ-té (kde ℓ je délka nejkratší zlepšující cesty) vrstvě bude chybět ohodnocení (díky čemuž si též vystačíme s $\mathcal{O}(N)$ paměti). To ale vůbec nevadí. Prostě jen při DFS budeme ochotni navštěvovat neohodnocené sousedy vrcholů v $(\ell - 1)$-ní vrstvě a chovat se k nim, jako by ležely v ℓ-té vrstvě.

Nyní k DFS: Každá navštívená volná zkratka znamená novou nalezenou zlepšující cestu. Těch ovšem najdeme za jednu fázi nejvýše N, navštívíme tedy jen $\mathcal{O}(N)$ nezajímavých vrcholů a hran. To opět znamená, že si vystačíme s $\mathcal{O}(N^2)$ času na DFS část jedné fáze a $\mathcal{O}(N)$ paměti pro značky.

Hopcroft-Karpův algoritmus běží v čase $\mathcal{O}(T_f \sqrt{|M|})$, kde T_f je čas strávený jednou fází a $|M|$ je velikost výsledného párování. V předchozích odstavcích jsme ukázali, že $T_f = \mathcal{O}(N^2 \log N)$ (použijeme-li jako slovník vyvážené vyhledávací stromy), již dávno víme, že $|M| \leq N$ a ještě potřebujeme $\mathcal{O}(S)$ na načtení vstupu. Výsledná složitost tedy bude $\mathcal{O}(S + N^2\sqrt{N} \log N)$ a stále si vystačíme s $\mathcal{O}(S)$ paměti. V praxi by nejspíš bylo výhodnější reprezentovat slovníky jako hešovací tabulky.

Program (Python):
`http://ksp.mff.cuni.cz/viz/26-2-6-velke-abc.py`

Filip Štědronský

26-2-7 Čištění kmene

Nejdříve se podíváme na to, jakým způsobem může vypadat výsledný pohyb humanoidů. Nahlédneme, že existuje optimální řešení splňující tyto vlastnosti:

- Každý humanoid vyčistí nějaký souvislý interval.
- Každý humanoid změní směr pouze jednou. Tedy napřed půjde jen doprava a pak jen doleva, nebo naopak. Tedy dojde na jeden kraj svého intervalu, otočí se a zamíří ke druhému kraji, kde se zastaví.
- Z toho je hned vidět, že se nevyplatí humanoidy cíleně prohazovat. Pokud by se dva humanoidi měli prohodit, tak si můžeme situaci představit tak, že se od sebe odrazí a každý pokračuje plánovanou cestou toho druhého.

Zkusme nyní zodpovědět otázku, zda je možné kmen vyčistit, pokud budeme mít k dispozici k kroků. Postupovat budeme následovně: Pokud má nejlevější humanoid nějakou nečistotu nalevo od sebe, tak ji určitě musí vyčistit on. Tedy začátek jeho intervalu bude l kroků nalevo od něj – tam kde daná nečistota leží. Pravý konec jeho intervalu bude r kroků napravo od něj, přičemž chceme, aby r bylo co největší.

Humanoid má dvě možnosti:

- Buď půjde nejprve l kroků doleva, tam se otočí a vydá se $l + r$ kroků doprava.
- Nebo půjde nejdříve r kroků doprava, obrátí se a pak se vydá $l + r$ kroků doleva.

Celkově má udělat k kroků, takže bude platit buď $2l+r = k$, nebo $l + 2r = k$. Z těchto dvou rovnic vybereme tu, jejíž řešení má větší r. Pokud ani jedna rovnice nemá řešení s $r \geq 0$, tak nejlevější nečistotu za k kroků vyčistit nemůžeme, tedy pro tuto hodnotu k řešení neexistuje.

Pokud alespoň jedna z rovnic řešení pro r má, tak smažeme všechny nečistoty do l kroků nalevo od humanoida a do r kroků napravo od humanoida a stejným způsobem pokračujeme výpočtem u dalšího humanoida zleva. Pokud po zpracování posledního humanoida budeme mít vyčištěné všechny nečistoty, tak jsme našli řešení, které funguje pro k kroků.

Nyní si stačí všimnout, že pokud úloha má řešení pro k kroků, tak určitě má řešení i pro $k + 1$ kroků. Naopak pokud úloha nemá řešení pro k kroků, tak určitě nemá řešení ani pro $k-1$ kroků. Optimální k tedy můžeme nalézt binárním vyhledáváním.

Maximální počet kroků, který má smysl uvažovat, je dvakrát rozdíl nejmenší a největší souřadnice na vstupu, protože určitě nepůjdeme víc jak tam a zpátky. Označme tuto hodnotu X.

Pak binární vyhledávání udělá maximálně $\mathcal{O}(\log X)$ kroků; každý zabere čas $\mathcal{O}(H + M)$, kde H je počet humanoidů a M je počet nečistot. Poznamenejme ještě, že humanoidy i místa si na začátku musíme setřídit. Celková časová složitost tedy bude $\mathcal{O}((H + M) \log X)$.

Paměťová složitost je $\mathcal{O}(M + H)$. Můžete nahlédnout do vzorového programu.

Program (C++):
`http://ksp.mff.cuni.cz/viz/26-2-7.cpp`

Karel Tesař

Nezdálo se však, že by podobná technologie byla k dispozici zde. Jak si vlastně lidstvo umělo poradit se sopkami v dobách, kdy ještě neexistovaly materiály dost odolné k jejich usměrnění? Přeci by jim neustupovalo...

Z úvah Jacoba vytrhlo zapraskání. Znělo to skoro, jako by se sopka po probuzení potřebovala nejprve protáhnout. Po dalších dvou zapraskáních nastalo opět ticho. Nemělo však dlouhého trvání, bylo po chvilce prořízнuto ránou tak ohromnou, že si všichni instinktivně zacpali uši. Ze sopky se vyvalil oblak popela.

Šedivé kousky popela zvolna jako sníh dopadaly na Jacoba a mimozemšťany. Všichni stáli na místě jako přikovaní. Nikdo nebyl schopen slova. Začaly se objevovat první gejzíry lávy. První se vzpamatoval Jacob a zavelel všem k útěku.

Ozval se další výbuch. Tentokrát se už nejednalo o pouhý gejzír. Zformoval se celý proud lávy, který si jako horská bystřina razil svou cestu krajinou. Zřejmě se vydal stíhat prchající skupinku. Jacob se ohlédl a odhodil svůj batoh. Pochopil, že bude muset být opravdu rychlý.

Pokračování příště...

O Jacobových setkáních s mimozemšťany vyprávěl

Lukáš Folwarczný

26-3-8 Zdivočelá počítadla — 15 bodů

Seriál o výpočetních modelech pokračuje, tentokrát ve znamení minimalismu. V tomto dílu si ukážeme jeden z vůbec nejjednodušších teoretických strojů. Takové obyčejné kuličkové počítadlo, jen trochu programovatelné. Proto mu budeme říkat *počítadlový stroj*.

Definice stroje

Počítadlový stroj pracuje výhradně s přirozenými čísly. Ta mohou být libovolně velká a zahrnujeme mezi ně i nulu. Obvykle jim budeme říkat prostě *čísla*.

Stroj je vybaven libovolným konečným počtem *registrů*. Registry jsou očíslovány a každý z nich obsahuje jedno číslo.

Program stroje je tvořen konečnou posloupností *instrukcí*. Těch jsou k dispozici 3 druhy:

- `INC` x (increment) – zvýší registr x o jedna
- `DEC` x (decrement) – sníží registr x o jedna; pokud už byl nulový, nic se nestane.
- `JNZ` x, p (jump if non-zero) – pokud je hodnota v registru x nenulová, skočí na p-tou instrukci programu; pokud je hodnota nulová, nic se nestane.

Na počátku výpočtu je ve smluvených registrech vstup a ostatní registry obsahují nuly. Instrukce se vykonávají jedna po druhé, počínaje první instrukcí programu. Pouze instrukce skoku může způsobit, že se místo následující instrukce začne provádět nějaká jiná, od níž program opět pokračuje sekvenčně.

Pokud se ocitneme mimo program (ať už tím, že jsme tam skočili, nebo po provedení poslední instrukce programu), stroj se zastaví a ve smluvených registrech najdeme výstup. Časovou složitost bychom mohli zavést jako počet provedených instrukcí, paměťovou třeba jako velikost čísel, která se během výpočtu vyskytnou v registrech. U úloh v této sérii se nicméně nebudeme počítání složitosti věnovat, bude nás zajímat pouze počet použitých registrů. Ten se jako míra složitosti nechová, protože nezávisí na vstupu – je to spíš míra komplikovanosti programu.

Rozšíření instrukční sady

Instrukční sada našeho stroje je značně spartánská. Proto ji rovnou rozšíříme o několik zkratek pro běžné programátorské obraty.

- Zkratka `CLR` x (clear) bude sloužit k vynulování registru x:

```
A:  DEC x
    JNZ x, A
```

Druhý parametr instrukce `JNZ` by měl udávat pořadové číslo neboli adresu instrukce `DEC` v programu. Abychom si nemuseli adresy pamatovat, instrukci `DEC` si pojmenujeme *návěštím* `A` a kdekoliv se na její adresu potřebujeme odkázat, uvedeme místo ní návěští. Tuto konvenci budeme používat u všech instrukcí skoku.

- `JMP` p (jump) bude značit nepodmíněný skok na instrukci s adresou p. Jak si ho pořídit? Můžeme si například obstarat nějaký registr n a na začátku programu instrukcí `INC` n zařídit, aby zaručeně nebyl nulový. Kdykoliv pak použijeme `JNZ` n, p, stroj vždy skočí na adresu p.

Pokud vám přijde, že plýtváme registry, máte pravdu. Proto si raději místo nového registru n „vypůjčíme“ nějaký registr x, který už se v programu používá. Vždy ho na chvíli inkrementujeme, aby byl nenulový, a po použití zase vrátíme do původního stavu. Pokud by tedy program vypadal takto:

```
    (nějaké instrukce)
A:  (další instrukce)
    JMP A
```

mohli bychom ho přeložit na:

```
    (nějaké instrukce)
    INC x
A:  DEC x
    (další instrukce)
    INC x
    JNZ x,A
```

- `JZ` x, p (jump if zero) – opak instrukce `JNZ`, tedy skok, pokud je registr x nulový:

```
    JNZ x,Q
    JMP p
Q:
```

- `MOV` a, b (move) – zkopírování hodnoty z registru a do registru b. Pořídíme si pracovní registr t a provedeme:

```
    CLR b
    JZ  a,Z
    CLR t
X:  DEC a
    INC b
    INC t
    JNZ a,X
Y:  DEC t
    INC a
    JNZ t,Y
Z:
```

V prvním cyklu (návěští `X`) snižujeme a po jedné a zvyšujeme b. Tak přesuneme hodnotu z a do b, ale a si zničíme. Proto kromě b zvyšujeme i t a v druhém cyklu (`Y`) přelijeme t zpět do a. Tato konstrukce ovšem nefunguje, pokud $a = 0$, což vyřešíme výjimkou (`JZ` na počátku programu).

Úkol 1 [3b]: Navrhněte následující zkratky:

- `ADD` x, y, z (add) – sečte obsah registrů x a y a výsledek uloží do registru z.
- `SUB` x, y, z (subtract) – odečte od obsahu registru x obsah registru y a výsledek uloží do registru z. Pokud by mělo vyjít záporné číslo, uloží nulu.
- `MUL` x, y, z (multiply) – vynásobí obsah registrů x a y a výsledek uloží do registru z.

Zase ty závorky

Nemůžeme opomenout naši tradiční úlohu o závorkování. Potřebujeme ovšem vymyslet, jak řetězec závorek popsat číslem. Zakódujeme ho velice jednoduše: každou levou závorku v řetězci přepíšeme na číslici 1, každou pravou na 2 a výsledek přečteme jako číslo v desítkové soustavě.

Například řetězec `()(())` přeložíme na číslo `121122`, prázdný řetězec zakódujeme jako nulu.

Úkol 2 [4b]: Napište program pro počítadlový stroj, který v registru x dostane kód posloupnosti závorek a až doběhne, sdělí v registru y, zda posloupnost byla ($y = 1$) nebo nebyla ($y = 0$) správně uzávorkovaná.

V programu můžete používat všechny zkratky, které jsme definovali, nebo které jste vymysleli v předchozím úkolu.

O síle počítadel

Jak je vidět z předchozího příkladu, pomocí počítadlového stroje lze odpovídat na netriviální otázky. Ukážeme, že je dokonce stejně silný jako Turingův stroj (a tím pádem i jako přepisovací programy z předchozí série).

Simulace počítadel na Turingově stroji je triviální – stačí každému počítadlu vyhradit jednu pásku stroje.

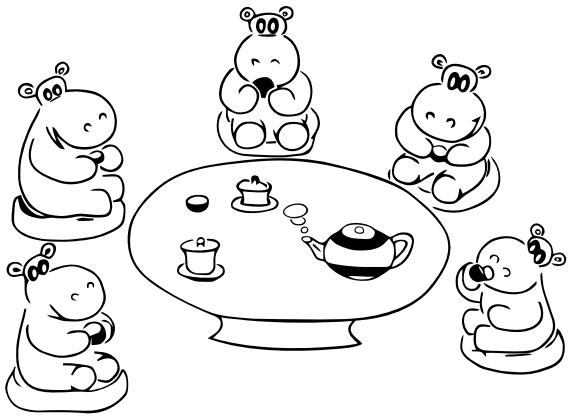
Úkol 3 [3b]: Vymyslete opačný převod: popište, jak k libovolnému Turingovu stroji sestrojit počítadlový stroj, který spočítá totéž. Zvolte vhodné kódování vstupu a výstupu Turingova stroje čísly.

Nabízí se také otázka, kolik počítadel doopravdy potřebujeme. U Turingova stroje víme, že si (za cenu zpomalení výpočtu) vystačí s jedinou páskou. Jak je to zde? Stačí pevný počet počítadel? A potřebujeme vůbec tolik instrukcí?

Úkol 4 [3b]: Pro co nejmenší konstantu k dokažte, že ke každému počítadlovému stroji existuje ekvivalentní stroj, který použije jen k registrů. Můžete předpokládat, že původní stroj pro vstup i výstup využívá jediný registr x. Dokázali byste k ještě snížit, pokud byste mohli zavést nějaké vlastní kódování vstupu?

Úkol 5 [2b]: Navrhněte ještě menší instrukční sadu, pomocí které půjdou vyjádřit všechny tři instrukce našeho počítadlového stroje. (Nová sada nemusí nutně být podmnožinou té původní.)

Martin „Medvěd“ Mareš



(kde $|E|$ je počet hran grafu a $|M|$ velikost výsledného párování), potrvá to $\mathcal{O}(N^2)$. Paměti spotřebujeme $\mathcal{O}(N)$ na uložení grafu.

Kdož jste zvyklí převádět párování na toky v sítích, vězte, že nejde o nic jiného než jinou formulaci Ford-Fulkersonova algoritmu pro párovací sítě. My zde používáme tuto verzi, protože další úpravy a optimalizace se na ní budou popisovat snáz než v tokové formulaci.

Program (Python):
`http://ksp.mff.cuni.cz/viz/26-2-6-male-abc.py`

Toto řešení je dostačující jak pro většinu běžných použití (anglická abeceda, ASCII), tak pro získání plného počtu bodů. Prostou výměnou párovacího algoritmu za Hopcroft-Karpův[7] zlepšíme čas na $\mathcal{O}(|E|\sqrt{|M|}) = \mathcal{O}(N\sqrt{N})$.

Zvláště zvědaví mohou pokračovat ve čtení a dozvědět se, co dělat v případě, že máme abecedu opravdu velkou.

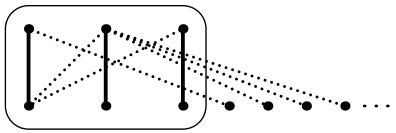
Velké abecedy

Pokud bychom předchozí postup chtěli zkusit např. s Unicode, náš graf by měl řádově 2^{32} vrcholů, do většiny z kterých by žádná hrana nevedla. Nabízí se vytvářet vrcholy pouze pro zkratky, které mohou z nějakého slova vzniknout. Kolik jich bude? Ze slova délky L lze vytvořit až řádově L^2 zkratek (pokud jsou všechna jeho písmena různá). Tedy náš graf může obsahovat až $\mathcal{O}(S^2)$ vrcholů a $\mathcal{O}(NS^2)$ hran.

To je pořád docela dost, jak pro časovou ($\mathcal{O}(N^2S^2)$), tak paměťovou ($\mathcal{O}(NS^2)$) náročnost našeho algoritmu. Obojí zkusíme zachránit drobnými úpravami párovacího algoritmu.

Pokud jej neznáte, teď je vhodná chvíle to napravit. Pokud ano, pro jistotu zopakujeme drobné shrnutí a trochu terminologie: *(ne)párovací hrana* je hrana grafu (ne)patřící do aktuálního párování, *volný vrchol* je takový, který není spárován (všechny hrany s ním incidentní jsou nepárovací), *střídavá cesta* je cesta v grafu, na které se střídají párovací a nepárovací hrany, a nakonec *zlepšující cesta*[8] (též *volná střídavá cesta*) je střídavá cesta začínající a končící volným vrcholem (a tedy nutně i nepárovací hranou). Algoritmus začne s prázdným párováním a v každém kroku najde zlepšující cestu, změní po celé její délce párovací hrany na nepárovací a naopak, čímž zachová korektnost párování a zvětší jej o jedničku. Pokud zlepšující cesta neexistuje, párování je maximální.

Klíčovým pro nás bude jedno pozorování: po celou dobu běhu algoritmu je většina grafu „nezajímavá“, totiž tvořená volnými zkratkami. „Zajímavých“ vrcholů je jen $\mathcal{O}(N)$ (N slov a nejvýše N spárovaných zkratek). Tedy v zajímavé části grafu je nejvýše $\mathcal{O}(N^2)$ hran. Graf si lze představit takto:



Nejprve napravíme čas. Všimneme si, že každá zlepšující cesta má lichou délku, a tudíž spojuje vrcholy z opačných partit. Stačí nám tedy hledat zlepšující cesty pouze z vrcholů v levé partitě. Dále si uvědomíme, že skoro celé hledání zlepšující cesty probíhá v zajímavé části grafu. Jakmile se dostaneme do nezajímavé části, narazili jsme na volný vrchol, a tedy konec zlepšující cesty. Při libovolném hledání zlepšující cesty tedy navštívíme libovolné množství vrcholů ze zajímavé části grafu, ale nejvýše jeden z nezajímavé. Takové hledání tedy bude trvat $\mathcal{O}(N^2)$. A použijeme-li odhad složitosti $\mathcal{O}$(čas na nalezení zlepšující cesty $\cdot\, |M|$), dostáváme čas $\mathcal{O}(N^3)$.

Nyní co s pamětí? Pomoci by nám mohl obvyklý trik na úlohy, které řešíme sestavením vhodného grafu: nebudeme jej v paměti vytvářet celý najednou. Místo toho, kdykoli se bude párovací algoritmus chtít podívat na nějakou část našeho grafu, tak mu ji „na požádání“ sestavíme. Co musíme umět s naším „grafem“ provádět, aby párovací algoritmus fungoval?

1. Pamatovat si, která hrana je aktuálně v párování a která ne, umět do párování hranu přidat/odebrat.
2. Umět vyjmenovat sousedy daného vrcholu.
3. Umět o vrcholu poznat, zda je volný.

Stačí nám si mimo samotného grafu pamatovat nějaké jiné datové struktury, které budou umět na takovéto dotazy rychle odpovídat. Pak prostě párovací algoritmus upravíme tak, aby kdekoli původně přistupoval přímo k paměťové reprezentaci grafu, místo toho použil tyto naše struktury.

Najít sousedy daného slova je jednoduché: prostě vyzkoušíme všechny možnosti volby prvního písmene a pro každou z nich všechny možnosti volby druhého písmene, dostávající všechny možné zkratky v čase $\mathcal{O}(1)$ na každou. Ovšem obráceně (najít sousedy zkratky, tedy všechna slova, ze kterých ji lze utvořit) to vůbec není jednoduché.

Naštěstí si všimneme, že to vůbec nepotřebujeme. Hledáme pouze volné střídavé cesty, a pokud začneme zleva, půjdeme na takové cestě doprava vždy po nepárovací hraně a doleva vždy po párovací. Stačí nám tedy umět hledat:

- Pro vrcholy z levé (slovní) partity sousedy po nepárovacích hranách.
- Pro vrcholy z pravé (zkratkové) partity souseda po párovací hraně, pokud existuje.

A to už zvládneme snadno. Naše reprezentace bude vypadat následovně:

- Vrcholy z levé partity (slova) si očíslujeme, vrcholy z pravé budeme prostě označovat dvěma písmeny.
- Párování si budeme uchovávat jako dvojici slovníků (hešovacích tabulek[9] či vyhledávacích stromů)[10] mapujících slova na aktuálně přiřazené zkratky a naopak. To nám umožní rychle hledat párovací hrany „z obou stran“.
- Volný vrchol poznáme prostě tak, že v příslušném slovníku není.
- Sousedy slova vyjmenujeme triviálně, jak bylo popsáno výše, z již předem redukovaných slov (zachován jen první a poslední výskyt libovolného písmene). Některé zkratky takto

[7] `http://ksp.mff.cuni.cz/encyklopedie/hopcroft-karp.html`
[8] Příznivci tokového párování: rozmyslete si, že takováto definice zlepšující cesty přesně odpovídá tokové zlepšující cestě v párovací síti – jen s useknutými hranami vedoucími ke zdroji a stoku (neb v našem grafu žádný zdroj a stok nemáme).
[9] `http://ksp.mff.cuni.cz/viz/kucharky/hesovani`
[10] `http://ksp.mff.cuni.cz/viz/kucharky/vyhledavaci-stromy`

Představme si nejprve, jak libovolný strom projít ve vzestupném pořadí hodnot bez použití rekurze. Začneme v kořeni a řídíme se následujícími pravidly:

- Pokud lze jít doleva, jdeme doleva.
- Pokud už nelze jít doleva, jdeme nahoru. Pokud jsme přišli zleva, vypíšeme aktuální hodnotu a pokračujeme doprava. Pokud jsme přišli zprava, pokračujeme nahoru.

Při konstrukci stromu si budeme počínat obdobně: vytvářený strom budeme takto „obcházet“ a místo abychom vrcholy vypisovali, tak je budeme zakládat.

To se snadno řekne, ale při implementaci nás čeká několik překážek.

Předně potřebujeme udržovat plánovanou velikost podstromu (proměnná k v logaritmickém řešení). Při kroku dolů ji prostě dělíme dvěma, leč při kroku nahoru nestačí násobit dvěma, neboť původní hodnota mohla být lichá. To vyřešíme tak, že si v každé úrovni rekurze zapamatujeme zbytek po dělení dvěma. To je jeden bit na každé z $\mathcal{O}(\log n)$ úrovní, takže se všechny dají poskládat do jediného čísla velkého řádově n.

Další potíž je, že občas potřebujeme přejít přes vrchol, který jsme ještě nevytvořili. To není těžké, ale při programování to řádně zamotá hlavu. Postačí, když budeme udržovat nejbližší vyšší vrchol, který skutečně existuje, a aktuální hloubku.

Co víc, k neexistujícímu vrcholu také občas potřebujeme něco připojit. Tak to připojíme k onomu nejbližšímu vyššímu existujícímu a navíc si zapamatujeme, ze které strany připojujeme (to je opět bit na úroveň).

Konečně si potřebujeme pro každou úroveň pamatovat, jestli aktuální vrchol na této úrovni ještě není vytvořen (tzn. zrovna jsme zalezlí kdesi v jeho levém podstromu), nebo už existuje (levý podstrom hotov, teď jsme někde v pravém).

Tímto způsobem také dosáhneme lineárního času a konstantní paměti.

Program (C) – alternativní řešení:
`http://ksp.mff.cuni.cz/viz/26-2-5-pruchod.c`

Dva pointery místo tří

Na závěr jedna pozoruhodná perlička: způsob uložení binárního stromu se dvěma pointery na vrchol, který umožňuje v konstantním čase zjistit levého syna, pravého syna i otce libovolného vrcholu.

První pointer (říkejme mu třeba P) bude ukazovat na levého syna. Druhý (řečený Q) bude v levém synovi ukazovat na jeho pravého sourozence, ale v pravém synovi na otce.

Levého syna vrcholu tedy zjistíme pomocí P. Pravého pomocí Q z levého syna. Pokud chceme znát otce, zkusíme přejít pomocí Q a podíváme se, zda jsme se dostali do vrcholu s menší hodnotou. Pokud ano, je to hledaný otec. Pokud ne, dostali jsme se do pravého sourozence, takže přejdeme ještě jednou po Q.

Pokud by vrchol měl jen jediného syna, bude na něj ukazovat P a jeho Q se odkáže zpět na otce. Porovnáním hodnot zjistíme, zda je to levý či pravý syn.

Martin „Medvěd“ Mareš & Dominik Macháček

[6] `http://ksp.mff.cuni.cz/encyklopedie/parovani.html`

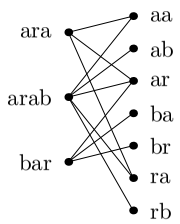
26-2-6 Zkratky míst

Všimněme si, že slova s různým počátečním písmenem nikdy nemůžou mít stejnou zkratku. Zkratka je totiž určená prvním písmenem slova a pak nějakým výběrem dalších dvou písmen z něj.

Můžeme tedy úlohu řešit samostatně pro každé počáteční písmeno a úloha se nám tak pro jednu „hromádku“ se stejným počátečním písmenem zjednodušuje na nalezení unikátních dvoupísmenných zkratek sestávajících se ze zbylých písmen slova (mimo prvního) ve správném pořadí.

Pokud jste již někdy slyšeli o problému hledání maximálního párování v bipartitním grafu, určitě vám jej tato úloha trochu připomíná. Pokud ne, doporučujeme vaší pozornosti příslušný článek v naší zbrusu nové programátorské encyklopedii.[6]

Jednu partitu (říkejme jí levá) tvoří slova, druhou (pravou) všechny možné zkratky. Hrany vedou z každého slova do všech zkratek, které z něj lze utvořit. Pak párování v tomto grafu odpovídá nějakému přiřazení zkratek slovům. Nyní stačí prostě najít párování maximální, tedy takové, které co nejvíce slovům přiřadí zkratku. Pro množinu slov `{ara, arab, bar}` by mohl graf vypadat takto:



Mohlo by vás ještě trochu překvapit, že si na úlohu bereme takto obecné kladivo. Naše zkratkové grafy přece mají velice speciální tvar, fakt, kterého obecný párovací algoritmus nijak nevyužije. To se občas hodí, když se náš problém podobá nějakému známému, na chvíli se tvářit, že jsou stejné. Nebojte, za chvíli se k alespoň některým zvláštnostem zkratkového grafu zase vrátíme.

Označme si N počet vstupních slov, S součet jejich délek a $|\Sigma|$ velikost abecedy. Než se pustíme do rozboru složitosti, ještě přidáme do algoritmu malý zlepšovák: při načítání vstupu upravíme slova tak, že v každém zachováme pouze první a poslední výskyt libovolného písmene. Snadno si rozmyslíte, že tím nijak nezměníme množinu zkratek, které lze ze slova vytvořit. V každém takto upraveném slově se libovolné písmeno abecedy vyskytuje nejvýše dvakrát, je tedy dlouhé $\mathcal{O}(|\Sigma|)$, a tudíž $S = \mathcal{O}(N \cdot |\Sigma|)$.

Jak bude náš graf velký? Možných zkratek existuje $|\Sigma|^2$, počet vrcholů tedy bude $N+|\Sigma|^2$. V nejhorším případě, kdy každé slovo obsahuje všechna písmena abecedy a půjde z něj vytvořit řádově $|\Sigma|^2$ zkratek, bude graf obsahovat $N|\Sigma|^2$ hran.

Pokud je abeceda malá, můžeme $|\Sigma|$ prohlásit za konstantu a dostaneme příjemný lineárně velký graf. Graf sestavíme v čase $\mathcal{O}(N \cdot |\Sigma|^2)$ (díky tomu, že každé slovo je nyní dlouhé $\mathcal{O}(|\Sigma|)$), tedy pro malou abecedu $\mathcal{O}(N)$.

Budeme-li, jako v odkazovaném článku, párovat postupným hledáním zlepšujících cest, které běží v čase $\mathcal{O}(|M| \cdot |E|)$

Dnešní povídání o algoritmech a datových strukturách se bude zabývat jedním z nejznámějších algoritmů: Dijkstrovým algoritmem pro hledání nejkratších cest v grafech. K tomu (a nejenom k tomu) se nám bude hodit šikovná datová strukturka zvaná halda, tak si předvedeme nejdříve ji.

Halda

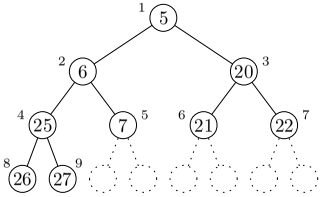
Halda je datová struktura pro uchovávání množiny čísel (či jakýchkoliv jiných prvků, na kterých máme definováno uspořádání, tj. umíme pro každou dvojici prvků říci, který z nich je menší). Tato datová struktura obvykle podporuje následující operace: Přidání nového prvku, nalezení nejmenšího prvku a odebrání nejmenšího prvku. My si ukážeme jednoduchou implementaci haldy, která bude při uložení N prvků potřebovat čas $\mathcal{O}(\log N)$ na přidání či odebrání jednoho prvku a $\mathcal{O}(1)$ (tj. konstantní) na zjištění hodnoty nejmenšího prvku.

Naše implementace bude vypadat následovně: Pokud halda obsahuje N prvků, uložíme její prvky do pole na pozice 1 až N. Prvek na pozici k bude mít dva *následníky,* a to prvky na pozicích $2k$ a $2k+1$; samozřejmě, pokud je k velké, a tedy např. $2k+1 > N$, má takový prvek jen jednoho či dokonce žádného následníka. Naopak prvek na pozici $\lfloor k/2 \rfloor$ nazveme *předchůdcem* prvku na pozici k. Ti z vás, kteří znají binární stromy, v tomto jistě rozpoznali způsob, jak v poli uchovávat úplné binární stromy (následníci jsou synové a předchůdci otcové v obvyklé stromové terminologii, prvek č. 1 je kořen stromu).

Prvky haldy však v poli neuchováváme v úplně libovolném pořadí. Chceme, aby platilo, že každý prvek je menší nebo roven všem svým následníkům. Naše halda tedy může vypadat např. takto:

1	2	3	4	5	6	7	8	9
5	6	20	25	7	21	22	26	27

Tomu odpovídá tento strom:



Z toho, co jsme si právě popsali, je jasné, že nejmenší prvek je uložen na pozici s indexem 1, a tedy můžeme snadno v konstantním čase zjistit jeho hodnotu. Ještě prozradíme, jak lze prvky do haldy rychle přidávat a odebírat:

Jestliže halda obsahuje N prvků, pak nový prvek, říkejme mu třeba x, přidáme na konec pole, tj. na pozici s indexem $N+1$. Nyní x porovnáme s jeho předchůdcem. Pokud je jeho předchůdce menší, je vše v pořádku a jsme hotovi. V opačném případě x s jeho předchůdcem prohodíme. Tím jsme problém napravili, ale nyní může být x menší než jeho nový předchůdce. To lze napravit dalším prohozením a tak budeme pokračovat dále, než se buďto dostaneme do situace, kdy už je x větší nebo rovno svému předchůdci, nebo „vybublá“ až do kořene haldy, kde už žádného předchůdce nemá. Protože se v každém kroku pozice, na níž se prvek x právě nachází, zmenší alespoň na polovinu, provedeme dohromady nejvýše $\mathcal{O}(\log N)$ výměn, a tedy spotřebujeme čas $\mathcal{O}(\log N)$.

Odebírání nejmenšího prvku probíhá podobně: Prvek z poslední pozice (tj. z pozice N) přesuneme na pozici 1, tedy místo minima. Místo s předchůdci jej však porovnáme s jeho následníky a v případě, že je větší než některý z jeho následníků, opět je prohodíme (pokud je větší než oba následníci, prohodíme ho s menším z nich). A protože se nám v každém kroku index „bublajícího“ prvku v poli alespoň zdvojnásobí, opět spotřebujeme čas $\mathcal{O}(\log N)$.

Jako cvičení si rozmyslete, že v čase $\mathcal{O}(\log N)$ lze z haldy smazat dokonce libovolný prvek, pokud si ovšem pamatujeme, kde se v haldě nachází. Také můžeme prvek ponechat a jen změnit jeho hodnotu.

Ještě si předvedeme program:

```
var halda: array[1..MAX] of integer;
    N: integer; { počet prvků v haldě }

function nejmensi: integer;
begin
  nejmensi:=halda[1]
end;

procedure vloz(prvek: integer);
var i, x: integer;
begin
  N:=N+1; i:=N;
  halda[i]:=prvek;
  while (i>1) and (halda[i div 2]>halda[i])
  do begin
    x:=halda[i div 2];
    halda[i div 2]:=halda[i];
    halda[i]:=x;
    i:=i div 2
  end
end;

procedure smaz_nejmensi;
var i, j, x: integer;
begin
  halda[1]:=halda[N];
  N:=N-1; i:=1;
  while 2*i<=N do begin
    j:=i;
    if halda[j]>halda[2*i] then j:=2*i;
    if (2*i+1<=N) and (halda[j]>halda[2*i+1])
       then j:=2*i+1;
    if i=j then break;
    x:=halda[i]; halda[i]:=halda[j];
    halda[j]:=x;
    i:=j
  end
end;
```

HeapSort

Když už máme k dispozici haldu, můžeme pomocí ní například snadno třídit čísla. Máme-li N čísel, která chceme setřídit, vytvoříme si z nich nejprve haldu o N prvcích (například postupným vkládáním do prázdné haldy), načež z ní budeme postupně N-krát odebírat nejmenší prvek. Tím získáme prvky původního pole v rostoucím pořadí. Celkově provedeme N vložení, N nalezení minima a N smazání. To vše dohromady stihneme v čase $\mathcal{O}(N \log N)$.

Než si ukážeme program, přidáme ještě dva triky, které nám implementaci značně usnadní. Předně si vše uložíme do jednoho pole – to bude při plnění haldy obsahovat na svém začátku haldu a na konci zbytek vstupního pole, přitom zbytek pole se bude postupně zmenšovat a uvolňovat tak místo haldě; naopak v druhé polovině algoritmu budeme zmenšovat haldu a do volného prostoru ukládat setříděné prvky. K tomu se nám bude hodit získávat prvky v opačném pořadí, proto si upravíme haldu tak, aby udržovala nikoliv minimum, nýbrž maximum.

Druhý trik spočívá v tom, že nebudeme haldu vytvářet postupným vkládáním, nýbrž naopak zabubláváním prvků (podobným, jako děláme při mazání minima) od konce. Všimněte si, že takto také získáme správné nerovnosti mezi prvky a jejich následníky, a dokonce tak zvládneme celou haldu vytvořit v lineárním čase (proč to tak je, si zkuste dokázat sami, stačí si uvědomit, kolikrát zabubláváme které prvky). Zbytek třídění bohužel nadále zůstává $\mathcal{O}(N \log N)$.

Tomuto algoritmu se obvykle říká *HeapSort* (čili třídění haldou) a je jedním z mála známých rychlých třídících algoritmů, které nepotřebují pomocnou paměť.

```
type Pole = array[1..MAXN] of Integer;

procedure HeapSort(var A: Pole);
var i, x: integer;
   procedure bublej(m, i: integer);
   { zabublání prvku: m je velikost haldy,
   i je index zabublávaného prvku }
   var j, x: integer;
   begin
      while 2*i<=m do begin
         j:=2*i;
         if (j<m) and (A[j+1]>A[j]) then j:=j+1;
         if A[i]>=A[j] then break;
         x:=A[i]; A[i]:=A[j]; A[j]:=x;
         i:=j;
      end;
   end;
begin
   for i:=N div 2 downto 1 do bublej(N,i);
   { vybírej maximum }
   for i:=N downto 2 do begin
      x:=A[1]; A[1]:=A[i]; A[i]:=x;
      bublej(i-1, 1);
   end;
end;
```

Dijkstrův algoritmus

Nyní již konečně k slíbenému *Dijkstrovu algoritmu*. Tento algoritmus dostane orientovaný graf s hranami ohodnocenými nezápornými čísly (viz kuchařka o grafech)[2] a nalezne v něm nejkratší cestu mezi dvěma zadanými vrcholy. Ve skutečnosti tento algoritmus dělá o malinko více: Najde totiž nejkratší cesty z jednoho zadaného vrcholu do všech ostatních.

Nechť v_0 je vrchol grafu, ze kterého chceme určit délky nejkratších cest. Budeme si udržovat pole délek zatím nalezených cest z vrcholu v_0 do všech ostatních vrcholů grafu. Navíc u některých vrcholů budeme mít poznamenáno, že cesta nalezená do nich je už ta nejkratší možná. Takovým vrcholům budeme říkat *definitivní*. Na začátku inicializujeme v poli všechny hodnoty na ∞ kromě hodnoty odpovídající vrcholu v_0, kterou inicializujeme na 0 (délka nejkratší cesty z v_0 do v_0 je 0). V každém *kroku* algoritmu pak provedeme následující: Vybereme vrchol w, který ještě není definitivní, a mezi všemi takovými vrcholy je délka zatím nalezené cesty do něj nejkratší možná. Vrchol w prohlásíme za definitivní. Dále otestujeme, zda pro nějaký vrchol v cesta z vrcholu v_0 do w a pak po hraně z w do v není kratší, než zatím nalezená cesta z v_0 do v, a je-li tomu tak, upravíme délku zatím nalezené cesty do v. Toto provedeme pro všechny takové vrcholy v. Celý algoritmus skončí, pokud jsou už všechny vrcholy definitivní nebo všechny vrcholy, co nejsou definitivní, mají délku cesty rovnou ∞ (v takovém případě se graf skládá z více nesouvislých částí).

Předtím než dokážeme, že právě představený algoritmus opravdu nalezne délky nejkratších cest z vrcholu v_0, se zamysleme nad jeho časovou složitostí.

Pro každý z N vrcholů si délku dosud nalezené cesty uchováváme v poli. Celý algoritmus provede nejvýše N kroků, protože v každém kroku nám přibyde jeden definitivní vrchol. Ten vybíráme jako minimum z délky aktuální cesty přes všechny dosud nedefinitivní vrcholy, kterých je $\mathcal{O}(N)$. V každém kroku musíme zkontrolovat tolik vrcholů v, kolik hran vede z vrcholu w. Počet takových změn pro všechny kroky dohromady je pak nejvýše $\mathcal{O}(M)$, kde M je počet hran vstupního grafu. Z toho vyjde časová složitost $\mathcal{O}(N^2 + M)$, čili $\mathcal{O}(N^2)$, jelikož M je nejvýše N^2. Tuto implementaci Dijkstrova algoritmu najdete na konci naší kuchařky.

K uchovávání délek dosud nalezených nejkratších cest můžeme ovšem použít haldu. Ta bude na začátku obsahovat N prvků a v každém kroku se počet jejích prvků sníží o jeden: Nalezneme a smažeme nejmenší prvek, to zvládneme v čase $\mathcal{O}(\log N)$, a případně upravíme délky nejkratších cest do sousedů právě zpracovávaného vrcholu. To pro každou hranu trvá rovněž $\mathcal{O}(\log N)$, celkově za všechny hrany tedy $\mathcal{O}(M \log N)$. Z toho vyjde celková časová složitost algoritmu $\mathcal{O}((N+M) \log N)$, a to je pro „řídké“ grafy (tedy grafy s $M \ll N^2$) výrazně lepší.

Vraťme se nyní k důkazu správnosti Dijkstrova algoritmu. Ukážeme, že po každém kroku algoritmu platí následující tvrzení: Nechť A je množina definitivních vrcholů. Pak délka dosud nalezené cesty z v_0 do v (v je libovolný vrchol grafu) je délka nejkratší cesty $v_0 v_1 \ldots v_k v$ takové, že všechny vrcholy $v_0, v_1, \ldots, v_k$ jsou v množině A. Tvrzení dokážeme indukcí dle počtu kroků algoritmu, které již proběhly. Tvrzení zřejmě platí před a po prvním kroku algoritmu. Nechť w je vrchol, který byl v předchozím kroku prohlášen za definitivní. Uvažme nejprve nějaký vrchol v, který je definitivní. Pokud $v = w$, tvrzení je triviální. V opačném případě ukážeme, že existuje nejkratší cesta z v_0 do v přes vrcholy z A, která nepoužívá vrchol w. Označme D délku cesty z v_0 do v přes vrcholy A bez vrcholu w. Protože v každém kroku vybíráme vrchol s nejmenším ohodnocením a ohodnocení vybraných vrcholů v jednotlivých krocích tvoří neklesající posloupnost (váhy hran jsou nezáporné!), tak délka cesty z v_0 do w přes vrcholy z A je alespoň D. Ale potom délka libovolné cesty z v_0 do v přes w používající vrcholy z A je alespoň D. Z volby D pak víme, že existuje nejkratší cesta z v_0 do v přes vrcholy z A, která nepoužívá vrchol w.

Nyní uvažme takový vrchol v, který není definitivní. Nechť

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

V pseudokódu bychom ji zapsali třeba takto:
Strom(S, k):

1. Je-li $k = 0$, vrátíme prázdný strom a seznam S.
2. $\ell \leftarrow \lfloor (k-1)/2 \rfloor$
3. $(L, S') \leftarrow$ Strom(S, ℓ)
4. $x \leftarrow$ odpojíme první prvek seznamu S'
5. $(P, S'') \leftarrow$ Strom$(S', k - \ell - 1)$
6. Vytvoříme strom T: kořen bude mít hodnotu x, jeho levým podstromem bude L a pravým P.
7. *Výstup:* Dvojice (T, S'').

Čas je opět lineární, paměti nám stačí $\mathcal{O}(\log n)$ na zásobník.

Konstantní prostor pro úplné stromy

Ani logaritmický prostor ovšem není nutný. Ukážeme řešení v konstantním prostoru, zatím ovšem pouze pro *úplné stromy*. Tak budeme říkat stromům, v nichž mají všechny vnitřní vrcholy právě 2 syny a všechny listy leží na téže hladině. Úplný strom hloubky h má tedy

$$2^0 + 2^1 + \ldots + 2^h = 2^{h+1} - 1$$

vrcholů. (Jiná možnost, jak úplné stromy definovat, je ještě zesílit definici dokonalé vyváženosti a požadovat, aby levý a pravý podstrom byly pokaždé přesně stejně velké. Proto se jim také někdy říká *perfektní stromy*.)

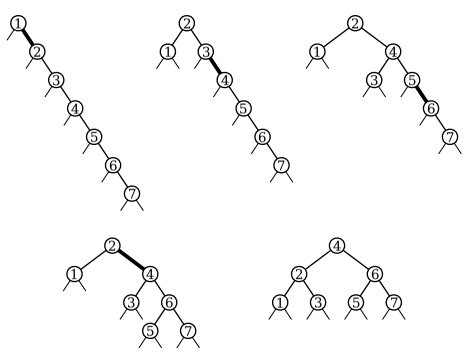
Dostaneme tedy seznam délky mocnina dvojky minus 1 a máme z něj vyrobit úplný strom.

Algoritmus bude jednoduchý, ale nečekaný. Jeho průběh sledujme na obrázcích níže (tučně je opět zvýrazněna hrana, kterou rotujeme; vlasové čáry pod vrcholy zatím ignorujte).

Půjdeme z kořene doprava a v každém kroku provedeme jednu rotaci doleva. Tím nám z cesty vznikne „hřeben“ – pravá cesta poloviční délky, z jejichž vrcholů vedou levé odbočky do listů.

Poté průchod z kořene doprava zopakujeme. Pravá cesta se opět dvakrát zkrátí a na levých odbočkách budou už viset „třešničky“ – úplné stromy hloubky 1 se třemi vrcholy. Další průchod vytvoří úplné stromy hloubky 2 se sedmi vrcholy a tak dále.

Nakonec celou pravou cestu rozebereme a zbude nám jeden úplný strom.



Času jsme spotřebovali lineárně: hran na pravé cestě je na počátku $n - 1$ a každá rotace jednu odstraní. Prostor potřebujeme pouze na konstantně mnoho pracovních proměnných.

Konstantní prostor pro všechny stromy

Dobrá, ale co když počet vrcholů není tak hezké číslo, aby šlo sestrojit úplný strom? Tehdy najdeme nejbližší menší číslo m tvaru $2^i - 1$, z tolika prvků seznamu sestrojíme úplný strom a zbylých $r = n - m$ vrcholů pověsíme pod listy úplného stromu.

Zařídí se to snadno: pokud k některým vrcholům počáteční liány přivěsíme levé syny a spustíme algoritmus pro úplné stromy, tak se tyto „přívěsky“ objeví právě pod listy úplného stromu. (To znázorňují ony vlasové čáry v obrázku, které jsme napoprvé přehlíželi.)

Náš obecnější algoritmus tedy projde liánu a celkem r-krát provede rotaci doleva, aby se z části vrcholů staly přívěsky a zbyla pravá cesta délky m, na kterou půjde spustit původní algoritmus.

Oněch r rotací se samozřejmě budeme snažit rozmístit co nejrovnoměrněji. Uděláme to takto: pořídíme si proměnnou, která bude na počátku nulová, v každém kroku k ní přičteme r/m a kdykoliv překročí 1, tak zrotujeme a odečteme 1.

Dokonce se můžeme zbavit dělení: místo původní proměnné si budeme pamatovat její m-násobek. Začneme tedy na nule, pokaždé přičteme r a pokud překročíme m, zrotujeme a odečteme m. (Mimochodem, přesně tento postup se používá při aproximování úseček pomocí pixelů na obrazovce.)

Zbývá dokázat, že skutečně vyjde dokonale vyvážený strom. Pro každý vrchol má platit, že rozdíl velikostí levého a pravého podstromu činí nejvýše 1. Jelikož v úplném stromu jsou oba podstromy stejně velké, stačí ukázat, že se počty přívěsků pod nimi liší nejvýše o 1.

Všimneme si, že vrcholy obou podstromů tvoří v liáně souvislé úseky, navíc stejně dlouhé. Ukážeme, že pro každé dva úseky stejné délky platí, že se počty přívěsků pod nimi liší nejvýše o 1.

Počítejme, kolik náš algoritmus mohl vygenerovat přívěsků pro úsek délky u. Nechť pomocná proměnná řídící přivěšování má na začátku úseku hodnotu h (víme, že $0 \le h < m$). Pak vytvoříme celkem $\lfloor (h + ur)/m \rfloor$ přívěsků. Tento výraz je nejmenší pro $h = 0$ a největší pro $h = m - 1$, přičemž obě krajní hodnoty se mohou lišit nejvýše o 1. (Neboť pro každé x platí $\lfloor x + 1 \rfloor = \lfloor x \rfloor + 1$.)

Výsledný strom je tedy dokonale vyvážený a na jeho vytvoření nám postačil lineární čas a konstantní paměť.

Program (C) – řešení pomocí rotací:
`http://ksp.mff.cuni.cz/viz/26-2-5-rotace.c`

Uvedený algoritmus pochází z článku *Tree Rebalancing in Optimal Time and Space* od Quentina F. Stouta a Bette L. Warrenové, na který nás upozornili řešitelé. Aneta Šťastná navíc navrhla jednodušší rozmisťování přívěsků, kterým jsme se také inspirovali.

Alternativní řešení

Když jsme úlohu zadávali, měli jsme vymyšlený úplně jiný způsob řešení. Jeho rozbor se všemi detaily je trochu pracnější, základní myšlenka ovšem také stojí za zmínku.

Upravíme logaritmické řešení, aby nepotřebovalo tolik paměti na zásobník. Budeme předpokládat, že každý vrchol si kromě ukazatelů na syny pamatuje i ukazatel na otce (časem se ukáže, že to není potřeba).

Nyní si můžeme všimnout, že při sčítání hmotností ostatních kostek sčítáme dokola téměř ta samá čísla. Lepší tedy bude si předem spočítat součet hmotností všech kostek na hromádce, označme jej třeba S. Do věže potom můžeme přidat kostky, pro které platí $S - w_i \leq \ell_i$, tedy ekvivalentně $S \leq \ell_i + w_i$.

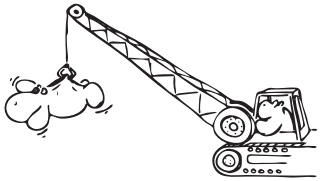
Když se nyní podíváme na vývoj S po jednotlivých krocích, zjistíme, že se číslo S pouze snižuje. Nic nám tedy nebrání kostky vybírat v pořadí od největší po nejmenší podle součtu $w_i + \ell_i$.

Algoritmus díky tomu můžeme velmi zjednodušit a urychlit. Kostky nejprve setřídíme podle součtu hmotnosti a nosnosti. A následně v lineárním čase ověříme, zda se kostky unesou. K tomu stačí, když půjdeme od špičky věže dolů a vždy porovnáme nosnost aktuální kostky se součtem hmotností kostek nad ní. Tento součet si v průběhu snadno spočítáme z předchozího pouhým přičtením hmotnosti další kostky. Celý algoritmus tak doběhne v čase $\mathcal{O}(K \log K)$ a spotřebuje při tom $\mathcal{O}(K)$ paměti.

Program (C) – lehčí varianta:
`http://ksp.mff.cuni.cz/viz/26-2-4-lehci.c`

Program (C++) – těžší varianta:
`http://ksp.mff.cuni.cz/viz/26-2-4-tezsi.cpp`

Vojta Sejkora & Jenda Hadrava



26-2-5 Vyvažování

Vyhledávací stromy lze vyvažovat mnoha různými algoritmy, které všechny pracují v lineárním čase, ovšem liší se množstvím potřebné paměti. Začneme tím nejobyčejnějším, který potřebuje lineární pracovní prostor, a postupně se propracujeme až ke konstantní paměti.

Rozebrat a složit: lineární prostor

Přetvářet nevyvážený strom na vyvážený pomocí lokálních úprav vypadá složitě. Co kdybychom ho prostě rozebrali a pak znovu poskládali?

Rozebírání proběhne rekurzivním průchodem stromu: vždy projdeme nejprve levý podstrom, pak kořen a nakonec pravý podstrom. Takto jednotlivé vrcholy navštívíme ve vzestupném pořadí. Stačí si je tedy průběžně ukládat do pole.

Ze setříděného pole posléze vyrobíme dokonale vyvážený strom: kořenem bude prvek, který v poli leží uprostřed (tedy medián). Hodnoty ležící v poli před ním patří do levého podstromu, hodnoty za ním do pravého. Oba podstromy sestrojíme rekurzivně a zavěsíme pod kořen. Do rekurze přitom stačí předávat počáteční a koncový index úseku pole, ze kterého zrovna stavíme strom.

Jak rozebrání starého stromu, tak postavení nového nás stojí $\mathcal{O}(n)$, kde n je počet vrcholů stromu. Na co všechno spotřebujeme paměť? Předně si musíme uložit lineárně velké pole hodnot. Nesmíme také zapomenout na zásobník od rekurze: na něm bude najednou tolik položek, kolik činí hloubka stromu. Ta může při rozebírání dosáhnout až n, během skládání pak pouze $\mathcal{O}(\log n)$. Prostoru tedy celkem zabereme $\mathcal{O}(n)$.

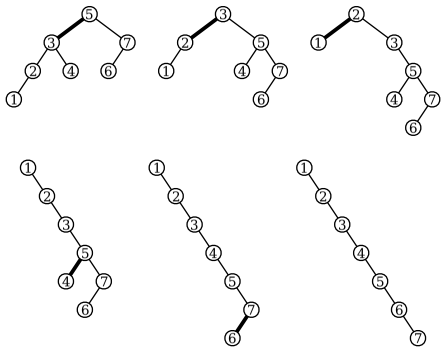
Rozebrání stromu na seznam

Pojďme se zbavit zbytečné kopie hodnot. Místo kopírování vrcholy zadaného stromu popřepojujeme, aby tvořily *liánu,* tedy strom, v němž nejsou žádní leví synové. Ukazatele na pravé syny nám tedy tvoří obyčejný spojový seznam, navíc setříděný podle hodnot.

K převodu stromu na liánu se budou hodit rotace (viz kuchařka).[5] Ty umožňují měnit tvar stromu, a přitom stále zachovávají uspořádání vrcholů (strom je tedy stále vyhledávací).

Budeme postupovat takto: dokud má kořen levého syna, provádíme v kořeni rotaci doprava. Když už levého syna nemá, sestoupíme do jeho pravého syna a tam algoritmus opakujeme.

Ukažme si, jak to vypadá pro jeden strom se 7 vrcholy (tučně je vždy vyznačena hrana, kterou se chystáme rotovat):



Proč to funguje? Stačí si všimnout, že mezi kořenem a aktuálním vrcholem leží nějaká už sestrojená liána. Rotace ji nepokazí a po konečně mnoha rotacích (každá zmenšuje velikost levého podstromu) učiníme jeden krok doprava, který liánu prodlouží. Algoritmus se tedy musí zastavit a v tu chvíli je celý strom liánou.

Kolik celkem strávíme času? Kroků doprava je lineárně. Abychom ukázali, že rotací také, stačí sledovat, jak se vyvíjí délka *pravé cesty.* To je cesta vedoucí z kořene doprava, dokud to jde. Každá rotace ji prodlouží o 1, ovšem délka cesty nikdy nepřekročí n, takže všech rotací je nejvýše n.

Logaritmické řešení

Nyní upravíme funkci na převod seznamu na strom, aby si vystačila se seznamem namísto pole.

Problematické místo je hledání prostředního prvku: v poli byl přístupný v konstantním čase, v seznamu bychom ho hledali lineárně dlouho, čímž bychom si pokazili celkovou časovou složitost.

Místo toho rekurzivní funkci navrhneme tak, aby dostala jako parametry seznam S a počet prvků k. Funkce odpojí prvních k prvků seznamu, vytvoří z nich strom T a vrátí jak tento strom, tak zbytek seznamu S''.

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

$v_0 v_1 \ldots v_k v$ je nejkratší cesta z v_0 do v taková, že všechny vrcholy $v_0, v_1, \ldots, v_k$ jsou v množině A. Pokud $v_k = w$, pak jsme ohodnocení v změnili na délku této cesty v právě proběhlém kroku. Pokud $v_k \neq w$, pak $v_0 v_1, \ldots, v_k$ je nejkratší cesta z v_0 do v_k přes vrcholy z množiny A a tedy můžeme předpokládat, že žádný z vrcholů $v_1, \ldots, v_k$ není w (podle toho, co jsme si rozmysleli na konci minulého odstavce). Potom se ale délka cesty do v rovnala správné hodnotě už před právě proběhlým krokem.

Vzhledem k tomu, že po posledním kroku množina A obsahuje právě ty vrcholy, do kterých existuje cesta z vrcholu v_0, dokázali jsme, že náš algoritmus funguje správně.

Na závěr ještě poznamenejme, že Dijkstrův algoritmus je možné snadno upravit tak, aby nám kromě určení délky nejkratší cesty i takovou cestu našel: U každého vrcholu si v okamžiku, kdy mu měníme ohodnocení, poznamenáme, ze kterého vrcholu do něj přicházíme. Nejkratší cestu do nějakého vrcholu pak zrekonstruujeme tak, že u posledního vrcholu této cesty zjistíme, který vrchol je předposlední, u předposledního, který je předpředposlední, atd.

Poznámka pro zvídavé: existují i jiné druhy hald, například k-regulární haldy, v nichž má každý prvek k následníků (rozmyslete si, jaká je v takové haldě časová složitost operací a jak nastavit k v závislosti na M a N, aby byl Dijkstrův algoritmus co nejrychlejší), nebo tzv. Fibonacciho halda, která dokáže upravit hodnotu prvku v konstantním čase. S tou pak umíme hledat nejkratší cesty v čase $\mathcal{O}(M + N \log N)$.

Dnešní menu Vám servírovali

Dan Král', Martin Mareš a Petr Škoda

Implementace Dijkstrova algoritmu

```
var N: word;                                  { počet vrcholů }
    vahy: array[1..MAX, 1..MAX] of integer; { váhy hran, -1 = hrana neexistuje }
    delky: array[1..MAX] of integer;          { délky zatím nalezených cest, -1 = nekonečno }
    def: array[1..MAX] of boolean;            { definitivní? }

procedure Dijkstra(odkud: word);
var i, w, v: word;
begin
  for i:=1 to N do begin
    def[i]:=false;  delky[i]:=-1;
  end;
  def[odkud]:=true;
  delky[odkud]:=0;
  repeat
    w:=0;
    for i:=1 to N do
      if not def[i] and ((w=0) or (delky[i]<delky[w])) then w:=i;
    if w<>0 then begin
      def[w]:=true;
      for i:=1 to N do
        if (vahy[w][i]<>-1) and (delky[w]+vahy[w][i]<delky[i]) then delky[i]:=delky[w]+vahy[w][i]
    end
  until w=0;
end;
```

26-2-1 Zamotané provazy

Prvním postřehem může být, že počet křížení musí být vždy sudý, jinak nemůže být provaz 2 nahoře na obou stromech. Dále si lze všimnout, že když je na dvou po sobě jdoucích kříženích nahoře stejný provaz, jde jen o přehozenou „vlnu“, a lze ji bez problémů rozmotat.

Pokud budeme postupně odstraňovat dvojice stejných křížení, dobereme se k jednomu ze dvou stavů. Pokud nám zbyly dva provazy bez křížení, jde je rozmotat. Jinak budou tvořit posloupnost, ve které se pouze střídají jedničky a dvojky, a kterou již rozmotat nelze.

Velmi se nabízí možnost implementovat program pomocí zásobníku. Každé nové překřížení porovnáme s vrcholem zásobníku a pokud se liší, přidáme nové křížení. Pokud jsou stejné, vrchol odebereme. Takto se určitě dostaneme ke každé dvojici, kterou bychom odebrat mohli. Už teď máme algoritmus lineární s počtem křížení a toto stačilo na zisk 6 bodů.

Lépe než lineárně to samozřejmě nepůjde, každé křížení ovlivňuje výsledek. Pomůžeme si ale s pamětí a to vynecháním zásobníku. Můžeme si to dovolit, protože se v něm po sobě jdoucí prvky vždy liší – jde vždy o střídající se jedničky a dvojky. Stačí si tedy pamatovat, čím zásobník končí a jak je hluboký.

Optimálním řešením je tedy výše popsaný algoritmus běžící v čase $\mathcal{O}(N)$ a využívající $\mathcal{O}(1)$ paměti. Za takové řešení jsme už dávali plný počet bodů.

Program (C):
`http://ksp.mff.cuni.cz/viz/26-2-1.c`

Ondra Hlavatý & Jirka Setnička

26-2-2 Barevný trojúhelník

V řešení se inspirujeme myšlenkou, kterou nám zaslal řešitel Matej Lieskovský. Předpokládejme pro spor, že v síti neexistuje souvislá oblast spojující všechny tři strany. Vezměme horní vrchol trojúhelníka a uvažme největší jednobarevnou souvislou oblast, ve které leží.

Tato oblast, řekněme bez újmy na obecnosti bílá, se dotýká minimálně dvou stěn, kterých se dotýká vrchol. Z předpokladu se však nedotýká třetí. Je tedy zespoda zcela ohraničena souvislým pásem černých šestiúhelníků, který navíc spojuje ty samé dvě stěny, které spojuje zvolená bílá oblast. To opět z předpokladu znamená, že tato černá oblast nezasahuje do třetí strany.

Nahlédněme, že když celou bílou oblast u horního vrcholu přebarvíme na černo, nevznikne tím souvislá oblast spojující tři strany – tato oblast spojuje stejné dvě strany jako její černá hranice, takže spojení se třetí stranou nevznikne. Tím jsme ovšem zvýšili aspoň o 1 počet černých šestiúhelníků a dostali jsme opět obrazec bez souvislé oblasti spojující všechny 3 strany.

Tuto operaci můžeme opakovat kolikrát chceme, ovšem po určitém počtu kroků musíme dojít do stavu, kdy už bude černý celý trojúhelník. To je ovšem obrazec, ve kterém už zřejmě existuje souvislá jednobarevná oblast spojující všechny tři strany, což je hledaný spor. Tím je důkaz hotov.

Alternativní interpretace stejné myšlenky by fungovala následovně: Pomocí pozorování výše snadno ukážeme, že v daném obarvení existuje souvislá oblast spojující všechny tři strany **právě tehdy, když** existuje taková oblast v obarvení, kde přebarvíme onu souvislou oblast při některém vrcholu.

Když tedy budeme dostatečně dlouho takto přebarvovat, opět dostaneme zcela černý trojúhelník, který už takovou všespojující oblast obsahuje. Z tohoto stavu ovšem můžeme do stavu, kde jsme začali, natáhnout řetěz výše popsaných ekvivalencí, takže existuje-li ve finálním stavu oblast spojující všechny tři strany, existuje i v původním obrazci.

Mark Karpilovskij

26-2-3 Plánování cesty

Úloha o plánování Jacobovy cesty nebyla ničím jiným, než hledáním nejkratší cesty v grafu. Přeformulujme si úlohu nejprve z řeči čtvercových políček do čisté řeči grafů.

Definujme orientovaný ohodnocený graf G. Vrcholy grafu budou políčka zadané oblasti. Hrany mezi dvěma políčky vedou právě tehdy, když spolu obě políčka sousedí stranou a obě jsou průchozí. Ohodnocení hrany z políčka A do políčka B bude odpovídat času potřebnému na zdolání políčka B.

Průchod do šířky

Graf G má RS vrcholů a $\mathcal{O}(RS)$ hran, kde $R \times S$ jsou rozměry oblasti. Řešením původní úlohy je délka nejkratší cesty v G ze startovního políčka do cílového políčka.

Prvním způsobem, jak úlohu řešit, je prohledat graf do šířky. Prohledávání do šířky (BFS) je popsáno v naší grafové kuchařce[3] a zde jej nebudeme opakovat.

Nesmíme ovšem zapomenout, že prohledávání do šířky funguje pouze v případech, kdy ohodnocení všech hran je jednotkové. Nejprve musíme ještě provést jednu úpravu grafu. Využijeme celočíselnosti ohodnocení hran a každou hranu nahradíme posloupností jednotkových hran. Konkrétně hranu s ohodnocením t nahradíme cestičkou složenou z t jednotkových hran. Takové operaci se také říká *podrozdělení*.

Pokud označíme jako T nejvyšší z ohodnocení všech hran, můžeme počet hran i vrcholů nového grafu omezit výrazem $\mathcal{O}(TRS)$. Nejkratší cesta v podrozděleném grafu odpovídá nejkratší cestě v původním grafu.

Na tomto grafu už použijeme prohledávání do šířky a získáme řešení s časovou složitostí $\mathcal{O}(TRS)$.

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

Dodejme ještě, že si můžeme vystačit s prostorem velikosti $\mathcal{O}(RS)$. Nově přidané vrcholy a hrany si totiž nemusíme nutně pamatovat. Stačí pracovat s původním G a akorát při vkládání vrcholu do fronty přidělit vrcholu jakési „zpoždění“, se kterým má být zpracován. Zpoždění nejprve nastavíme na ohodnocení hrany. Po vyjmutí vrcholu z fronty zpoždění snížíme o jedna. Je-li zpoždění stále větší než nula, vrchol nezpracujeme, ale znovu jej vložíme do fronty.

Dijkstrův algoritmus

S pomocí průchodu do šířky bylo možno zdárně vyřešit prvních pět vstupů, v dalších pěti testovacích vstupech již byla hodnota T příliš vysoká a bylo nutno využít sofistikovanější algoritmus.

Mezi takové algoritmy se například řadí Dijkstrův algoritmus, který je vlastně rozvinutím předchozí myšlenky o zpožděných vrcholech. Dijkstrův algoritmus je důkladně vyložen v další z našich kuchařek.[4] Kuchařka shodou okolností vychází letos zároveň se zadáním třetí série a bylo by zbytečné zde její obsah opakovat.

Použijeme-li při implementaci Dijkstrova algoritmu binární haldu, dosáhneme časové složitosti $\mathcal{O}((N+M)\log N)$, kde N je počet vrcholů a M počet hran grafu. V našem případě získáváme řešení s časovou složitostí $\mathcal{O}(RS \log RS)$. Paměťová složitost řešení bude $\mathcal{O}(RS)$.

Můžete si povšimnout, že použití k-regulární či Fibonacciho haldy by nám v našem konkrétním problému nepomohlo. Graf je totiž velmi řídký (má pouze lineární počet hran vzhledem k počtu vrcholů).

Program (C) – průchod do šířky:
`http://ksp.mff.cuni.cz/viz/26-2-3-bfs.c`

Program (C) – Dijkstrův algoritmus:
`http://ksp.mff.cuni.cz/viz/26-2-3-dijkstra.c`

Lukáš Folwarczný

26-2-4 Stavba věže

Lehčí varianta

Jakmile víme, že všech K kostek má stejnou váhu, můžeme si je setřídit podle jejich nosnosti. Poté je budeme postupně procházet od největší k nejmenší a budeme kontrolovat, že nosnost $\ell_i \geq K - i - 1$ (kde i je index v poli, které je číslováno od nuly).

Proč to můžeme udělat právě takto? Protože potřebujeme, aby každá kostka unesla všechny nad sebou. Ty váží právě $K - i - 1$ (pokud mají všechny jednotkovou váhu) a pokud bude tato nerovnost splněna pro každou kostku, tak takovou věž určitě postavit můžeme.

Pokud by naopak tato nerovnost na nějakém místě splněna nebyla, tak bychom aktuální kostku mohli prohodit jenom za nějakou kostku výše (protože kdybychom ji prohodili níž, musela by unést ještě víc). Jenže všechny kostky nad aktuální mají nosnost menší nebo rovnou aktuální, takže v takovém případě neexistuje způsob, jak by věž šla postavit.

Časová složitost takového řešení je $\mathcal{O}(K \log K)$, protože právě tak dlouho budeme třídit. Kontrola, že lze věž postavit, už proběhne jen v čase $\mathcal{O}(K)$. Taková řešení jsme ohodnotili slíbenými 3 body, tedy maximem za lehčí variantu.

[4] `http://ksp.mff.cuni.cz/viz/kucharky/halda-a-cesty`

Ale lehčí variantu lze vyřešit i v čase $\mathcal{O}(K)$. Uděláme to takto:

1. Vytvoříme pole o velikosti K (opět číslujeme od nuly).
2. Projdeme kostky a u těch, které mají $\ell_i \geq K$, nastavíme jejich nosnost na $K-1$ (protože větší nosnost nepotřebujeme, stačí nám, aby unesly maximálně $K-1$ kostek).
3. Postupně projdeme všechny kostky a do pole velikosti K si zapíšeme, kolik kostek má kterou nosnost (tedy budeme indexovat pole přímo nosností).
4. Poté si vybereme jednu kostku s nosností $K-1$. Pokud taková neexistuje, je jasné že věž postavit nelze a zahlásíme neúspěch. Pokud existuje, tak ji smažeme a spustíme náš algoritmus znovu pro K o jedna menší.

Ale počkat, to bude přece trvat až $\mathcal{O}(K^2)$! Zkusíme tedy vzít jenom hlavní myšlenku a vylepšíme ji. Rozmysleme si, jak se nám pole po smazaní jedné kostky ve 4. bodě změní.

Kostky, které měly předtím nosnost $\ell_i < K-1$, to nijak neovlivní, ty zůstanou na stejném místě. Jediné, co se změní, jsou kostky s nosností $\ell_i \geq K-1$, tyto (až na tu jednu smazanou) dostanou nosnost $\ell_i = K-2$.

Z toho nám vyplývá, jak naprogramovat řešení. První tři kroky uděláme přesně tak, jak je napsáno výše, ale 4. krok uděláme trochu chytřeji – uvědomíme si, že vlastně vůbec není nutné celé pole počítat pokaždé znova.

Stačí nám jednu kostku o nosnosti $K-1$ odečíst a všechny kostky o stejné nosnosti, které nám zbyly, přičíst ke kostkám s nosností $K-2$. Takto upravený 4. krok budeme opakovat, dokud se nedostaneme na $K=0$. Pokud jsme cestou nenarazili na žádné potíže, tak lze celá věž postavit.

Těžší varianta

V případě, že mají kostky rozdílné váhy, nám už nebude stačit obyčejné třídění podle nosnosti. Jako protipříklad použijeme dvě kostky: $w_1 = 1$, $\ell_1 = 3$ a $w_2 = 4$, $\ell_2 = 2$. Podle nosností bychom chtěli nejdříve umístit první, která už druhou neunese. Prohlásili bychom tedy, že věž nelze postavit. Druhá kostka však první unese, ale my bychom tuto možnost ani nevyzkoušeli.

Jak tedy úlohu vyřešit? Mnozí z vás správný postup vymysleli. Obtížnější však bylo dokázat, že řešení opravdu funguje. Ukážeme si kromě algoritmu a důkazu i to, jak můžeme na takové řešení přijít postupně.

Věž stavíme odspoda v jednotlivých krocích. Na začátku máme věž nulové výšky a *hromádku* všech kostek. V každém kroku z hromádky vybereme ty kostky, které unesou všechny zbývající kostky na hromádce. Vybrané kostky můžeme přidat v libovolném pořadí do věže.

Pokud se nám podaří tímto způsobem věž postavit ze všech kostek, tak máme vyhráno, protože jsme nikdy nepoužili takovou kostku, která by neunesla vše nad sebou. Pokud věž nesestavíme, tak jsme nalezli hromádku, ze které ani jedna kostka neunese všechny zbývající. Z této hromádky však nelze nikdy postavit věž – žádná kostka nemůže být základnou věže. Přidáním dalších kostek si celou konstrukci můžeme zatížit, nikdy ji však neodlehčíme. Věž tedy nejde postavit ani ze všech kostek.

Okamžitě tak dostáváme algoritmus, který úlohu řeší v čase $\mathcal{O}(K^3)$. Provedeme totiž nejvýše K kroků, kde v každém z nich spočítáme pro každou kostku součet hmotností všech zbývajících.