Milé řešitelky a milí řešitelé!

Zima už je dávno za námi, pomalu nastává jaro a už se opatrnými krůčky blíží i léto. To je přesně ten pravý čas na to, aby se objevila letošní poslední, pátá, série KSP, s termínem odevzdání tak, abychom vám ji stihli opravit, ještě než odjedete pryč na letní prázdniny.

Můžete se těšit opět na pokračování seriálu o výpočetních modelech, tentokrát si budeme hrát s grafovými automaty. Mimo seriál na vás čeká dalších sedm zajímavých úloh okořeněných závěrem napínavého Jacobova dobrodružství. A pokud chcete vědět, jak to s Jacobem nakonec dopadne, řešte, ať se dostanete na podzimní soustředění KSP, kde se k příběhu vrátíme.

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 z úloh této série získá dohromady alespoň 50 bodů, pošleme čokoládu**.

Termín odevzdání páté série je stanoven na **pondělí 26. května 2014 v 8:00 SELČ**. CodExová úloha má termín o den posunutý, opravuje ji totiž automat – odevzdejte ji do 27. května, 8:00 SELČ.

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*.

Pátá série dvacátého šestého ročníku KSP

V minulém díle se Jacob, zkoušený osudem již pádem hvězdné lodě, setkáním s mimozemšťany i výbuchem sopky, propadl do nějaké podzemní prostory. Teď v úžasu zíral na kovovou bednu se začouzeným nápisem „Made in China“.

Tohle je přeci lidská technika! A určitě to nejsou jen náhodně popadané trosky z Freyi*, protože tady je to celé propojené. To znamená… „Jsi v pořádku?“ přerušil náhle jeho myšlenky hlas shora. Jacob vzhlédl a spatřil jednoho mimozemšťana, jak klečí na okraji díry, kterou sem propadl.*

Ubu ho asi poslal, aby na něj dával pozor. No teď se to hodí, pomyslel si Jacob. Poslal mimozemšťana pro posily a za půl hodiny už táhli největší z kovových beden do tábora.

Jacob chtěl přijít na to, k čemu zařízení slouží. Jelikož bylo, zdá se, vyrobeno jako co nejlevnější a nejuniverzálněji použitelné zařízení, skládalo se jen z univerzální řídící jednotky, která byla naprogramována pomocí spousty relátek uspořádaných v pravidelné mřížce. Bohužel některá z nich byla spálená a bylo nutné je vyměnit.

26-5-1 Made in China — 9 bodů

Jacob má před sebou podivné zařízení a chtěl by ho spustit. Jenže předtím musí vyměnit několik spálených relé. Každé relé má své typové označení (přirozené číslo), které jde poznat po vyndání relé ze zařízení. Avšak spálená relé mají označení bohužel spečená k nepoznání.

Podle schématu na zařízení víme, že původně byla relé uspořádaná podle tabulky, kde v horní řádce i na levém okraji této tabulky byla relé označená postupně rostoucími přirozenými čísly začínajícími nulou $(0, 1, 2, 3, \ldots)$.

Zbytek tabulky byl vyplněn tak, že na pozici $[A, B]$ se nacházelo relé s nejmenším takovým číslem, které se doposud doleva ani nahoru od něj nevyskytlo (všechny tyto pozice již samozřejmě musí být osazeny). Prvních šest řádků a sloupců této tabulky tak bude vypadat takto:

```
0 1 2 3 4 5
1 0 3 2 5 4
2 3 0 1 6 7
3 2 1 0 7 6
4 5 6 7 0 1
5 4 7 6 1 0
```

Jacob potřebuje vyměnit několik spálených relé, ale nechce se mu kvůli tomu zkoumat všechna relé doleva a nahoru od těchto pozic. Proto by od vás potřeboval postup, který by mu rychle řekl, jaké relé má umístit na pozici $[A, B]$.

Po osazení všech relé se zařízení spustilo a zobrazilo na malé obrazovce několik informací. Po zběžném pročtení bylo Jacobovi vše jasné. Za pár minut už Ubu a několik dalších starších seděli v jednom ze stanů.

„Ctihodný Ubu, viděl někdy někdo z vašich lidí přímo někoho z vládnoucí kasty?“ zeptal se rovnou.

„Ne, podle našich zvědů nikdy nesundávají své blyštivé helmy, Jacobe.“

Jacob se nadechl, tohle by mohlo být ošemetné: „Ctihodný Ubu, rado starších, věc, kterou jsem vykopal, pravděpodobně spustila výbuch sopky. Je to asi nástroj, který sem umístil někdo z vládcových lidí. Ten nástroj ale poznávám,“ nadechl se, „je to technika mých lidí, nebešťanů.“

Viděl, jaké reakce to mezi radou starších vyvolalo, a tak rychle pokračoval: „O těch lidech nic nevím, možná to bude nějaký odštěpený klan. Musím se ale vypravit do královského města a promluvit si s nimi, přesvědčit je o tom, že dělají špatnou věc.“

Ještě chvíli diskutoval s radou, než se mu ji povedlo přesvědčit, že ho mají nechat jít samotného. Přízvuk už měl docela dobrý, tak dostal alespoň místní ekvivalent kutny, kterou tu občas nosili starší mimozemšťané a která mu zakrývala obličej i celé tělo. S ní by snad mohl splynout s místními. Mimo to si ještě vzal meč, který získal při rituálu, a dostal od Ady docela přesnou mapu pralesa. Rozloučil se s místními, minul místní ekvivalent kočky ležící na kraji tábora a vyrazil.

Jako první se vydal k vraku Freyi*. Tam si cestu pamatoval, a tak si během ní plánoval trasu od vraku směrem*

ke královskému městu. Bude to dlouhá cesta, vedoucí v první části skrz hustý prales, kde bude muset dávat pozor, aby v něm nepřehlédl žádnou odbočku.

26-5-2 Cesta pralesem — 9 bodů

Prales je místo, kde se velmi lehce přehlédne pokračování cesty. Máme mapu pralesa představovanou čtvercovou mřížkou o rozměrech $R \times S$ políček, dále máme zadané startovní políčko a cílové políčko.

Pohybovat se v mapě můžeme jen horizontálně nebo vertikálně, šikmo ne. Každé políčko má navíc dva *koeficienty přehlédnutelnosti*. Jeden pro cestu vedoucí přes něj rovně (shora dolů, zleva doprava nebo naopak) a druhý pro odbočení (tedy všechny zbylé průchody – zleva dolů, shora doleva, ...).

Protože chceme mít co největší naději, že se v pralese neztratíme, je vaším úkolem nalézt cestu ze startovního do cílového políčka, která bude mít v součtu přes všechna políčka cesty co nejmenší přehlédnutelnost (přehlédnutelnost ve startovním a cílovém políčku neuvažujte).

Než si Jacob naplánoval celou další cestu, už stál u boku UFC Freya. *Torzo lodě i po těch letech stále vypadalo majestátně, ale brázda v pralese i poničené okolí po nouzovém přistání už pomalu zarůstaly novou vegetací.*

Během svého několikaletého pobytu v mimozemském táboře se do vraku párkrát podíval, ale vždy se vydal jen k výstrojnímu skladu pro nové boty. Od ztroskotání nikdy nezašel dál, to místo na něj z nějakého důvodu působilo tísnivě a stejně tam nebylo nic zajímavého – nouzový signál nešel vyslat kvůli nějakému rušení v atmosféře a vody i jídla měl od mimozemšťanů dost.

Teď se skrz zprohýbaný koridor prodral hlouběji do lodi. Freya *jako původně válečná transportní loď měla v přídi několik zabezpečených skladů a Jacob je chtěl zkontrolovat. Zastavil se u zavřené přepážky, která nesla stopy po použití malého plazmového hořáku – někdo se chtěl propálit skrz, ale malým hořákem se mu to zdá se nepovedlo. To znamená, že se tady objevil nějaký jiný člověk, pravděpodobně někdo z těch, kteří teď vládnou domorodcům.*

Jacob odklopil ovládací panel a několika stisky probudil palubní počítač z jeho spánku. Chvíli trvalo, než naběhl, ale záložní baterie stále poskytovaly dostatek energie. Potom, co Jacob zadal svůj přístupový kód, se ale nic nestalo. Zkusil to znovu, a teprve pak si všiml, jakou paseku na kabelech neznámý návštěvník s plazmovým hořákem provedl. Ještě že ve skladu údržby byla celá krabice náhradních – jen rychle najít ten správný.

26-5-3 Náhradní kabel — 10 bodů

Máme před sebou bednu plnou náhradních kabelů a potřebujeme rychle poznat, jestli jsou nějaké dva kabely stejné. Nejsou to ale jen kabely se dvěma konci, tyto jsou rozvětvené a mohou obecně propojovat i více věcí dohromady.

Každý kabel si tedy můžeme představit jako spoustu uzlů pospojovaných jednotlivými dráty. Celý kabel je navíc pospojovaný tak, že v něm neexistují cykly – z každého uzlu do každého jiného se lze dostat jen jedinou cestou. Informatik by tedy takovýto rozvětvený kabel mohl nazvat *stromem*.

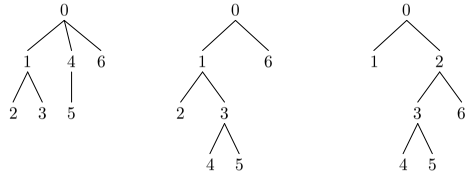
Jacob vždy náhodně uchopí dva kabely, a to tak, že je chytí za nějaký uzel a zbytek nechá viset dolů. Tomuto uzlu budeme říkat *kořen*. Díky tomu, že kabely jsou v uzlech spojeny v pevném pořadí, můžeme lehce popsat zbytek kabelu například tak, že kořen označíme indexem 0 a zbylé uzly vyjádříme jako N čísel označujících, pod kterým jiným uzlem je připojený i-tý uzel.

V případě více uzlů připojených pod ten samý je uvedeme třeba v pořadí proti směru hodinových ručiček (rozmyslete si, že je jedno, od jaké pozice začneme připojené uzly popisovat, když dodržíme jejich pořadí).

Jacoba by nyní zajímalo, jestli náhodou nejsou dva kabely, které drží v ruce, stejné. To znamená, že kdyby si je chytl oba za správný uzel, vypadaly by pak oba stejně (mimo přechycení za jiný uzel nesmíme pořadím kabelů v jednotlivých uzlech nijak rotovat).

Příklad: Pro kabely níže stačí, abychom prostřední kabel chytili za uzel s indexem 1 a správně otočili, pak budou levý a prostřední kabel stejné (všimněte si, že uzel 2 se sice zrotoval na druhou stranu, ale pořadí uzlů připojených k uzlu 1 se tím nijak nezměnilo). Nicméně pravý kabel se od nich liší (má jiné pořadí podstromů v uzlech).

Levý kabel	Prostřední kabel	Pravý kabel
1: 0	1: 0	1: 0
2: 1	2: 1	2: 0
3: 1	3: 1	3: 2
4: 0	4: 3	4: 3
5: 4	5: 3	5: 3
6: 0	6: 0	6: 2



Po zapojení správného kabelu zadal Jacob znovu svůj přístupový kód. Panel zezelenal a s hrozivým skřípotem se pancéřová přepážka otevřela asi do poloviny. Jacob prolezl skrz a ocitl se v sekci lodě, ve které od startu nebyl. Matně si pamatoval, že sem nakládali nějaké záhadné bedny s tajným obsahem – stále zde stály a vypadalo to, že jim nouzové přistání ani příliš neublížilo.

Jacoba ale zajímalo něco jiného. V malém skladu na konci oddělení se nalézala bedna, o níž vědělo jen několik členů bývalé posádky. Pomalu ji otevřel a vytáhl pár věcí ven. Pak zvedl plazmovou karabinu, zapřel si ji o rameno a pohledem skrz zaměřovač vyzkoušel, že vojenská zbraň stále funguje. Myslí mu proletěly vzpomínky na válku... jeho služba u elitní roty mariňáků, kamarádi v jednotce a jejich heslo Semper Fidelis – vždy věrní.

Zatřepal hlavou a zahnal vzpomínky. Je to už skoro dvacet let, co po konci války opustil armádu. Schoval masivní zbraň do batohu, sundal z helmy noční vidění a termovizi, k boku si připnul malou pistoli na uspávací šipky a pobral ještě několik drobností včetně sady nářadí. Pak vyrazil.

Podle mapy odhadl, že královské město bude ležet něco přes pět set kilometrů od místa, kde havaroval. Během svého putování minul několik mimozemských osad, ale vždy se držel mimo ně. Patnáctý den cesty, odhadem půlden cesty od královského města, mu však už došlo všechno jídlo. Chtěl si před setkáním raději pořádně odpočinout, a tak se rozhodl, že navštíví nejbližší vesnici, na kterou narazí.

Výsledková listina čtvrté série dvacátého šestého ročníku KSP

	řešitel	*škola*	*ročník*	*série*	*4-1*	*4-2*	*4-3*	*4-4*	*4-5*	*4-6*	*4-7*	*4-8*	*série*	*celkem*
0.					10	10	10	12	14	11	9	16	63,0	237,0
1.	Martin Raszyk	G_Karvina	4	19	10	10	10	12	3	3	9	15,5	56,8	218,5
2.	Jan Špaček	G_Wicht	3	4	10	10	6	11	10	11	9	15	60,4	215,6
3.	Marek Černý	G_Chrudim	3	4	4		8	12	10	11	9	11	58,1	206,1
4.	Václav Rozhoň	GJirsíkaČB	3	5	9,5	10	5				9	9	47,7	198,5
5.	Matej Lieskovský	GOmskPha	4	14	10	5	8		3			11	33,1	175,1
6.	Michal Punčochář	GJírovcČB	4	14	9,5	10	10		14		9		52,3	174,6
7.	Michal Korbela	GJJesen	4	4	9	6	8	4	2,5	3	8	2	42,0	166,7
8.	Jakub Svoboda	GKomHavíř	4	9	10		8	10		10		7	47,1	166,5
9.	Aneta Šťastná	GOmskPha	4	10	10		10	9	3			5	38,2	143,4
10.	Richard Hladík	GOAMarLaz	1	9		8					7	7	23,9	136,7
11.	Jakub Zárybnický	GTomkovaOL	3	4		10	6	1,5		10	3		36,6	127,7
12.	Jan-Sebastian Fabík	GJarošeBO	4	12							9		9,0	107,6
13.	Václav Volhejn	GKepleraPH	1	9	10		8		2,5		9		30,5	95,8
14.	Filip Bialas	GOpatovPHA	1	4							8		8,6	95,1
15.	Jan Knížek	G_Strakon	3	12	9	5	6	3,5	9	11	9	9	46,0	88,9
16.	Antonín Češík	SPSE_Pard	4	4							7		8,2	86,6
17.	Lucie Studená	GKepleraPH	4	3			8		0			2	13,2	83,8
18.	Jan Pokorný	G_Bučovice	2	4			8				9		18,2	77,3
19.	Jakub Maroušek	G_Písek	4	7									0,0	72,2
20.	Anna Steinhauserová	GDačice	4	3									0,0	71,7
21.	Štěpán Hojdar	GJírovcČB	4	8			3				4		8,8	68,9
22.	Dorian Řehák	GCoubTábor	3	3									0,0	67,5
23.	Ondřej Hübsch	GArabskáPH	4	22	10	9,5	8				9		32,9	62,9
24.	Štěpán Trčka	GSlavičín	3	9									0,0	54,3
25.	Jonatan Matějka	SŠP_ČB	4	17									0,0	50,9
26.	Dalimil Hájek	GKepleraPH	3	14							8		7,6	34,1
27.	Adam Španěl	ArcibisGPH	2	2									0,0	32,0
28.	Anna Gajdová	GFPValMez	3	1	10		8				9		28,5	28,5
29.	Dominik Roháček	SPŠLegioJI	4	3									0,0	27,0
30.	Antonín Teichmann	GJeronýmLI	4	2									0,0	22,6
31.	Jan Pavlovský	GJiM	4	1									0,0	21,3
32.	Marek Dobranský	GHorMichal	4	6									0,0	20,3
33.	Aneta K. Lesná	GZborovPH	1	1									0,0	16,8
34.	Michal Hloušek	GNadŠtolPH	1	1									0,0	16,3
35.	Přemysl Šťastný	GZamberk	0	3	4								6,0	16,0
36.	Petro Kostyuk	GEBeneše KL	4	2									0,0	12,1
37.	Radovan Švarc	G_ČTřebová	3	3									0,0	8,0
38.	Tadeas Friedrich	GOhradníPH	4	2									0,0	6,3
39.	Jan Horešovský	GMěl	4	2									0,0	6,2
40.	Michal Martinek	GHavPodl	3	1									0,0	6,0
41.	Josef Čech	GJMasar_JI	2	1							3		5,7	5,7
42.	Marek Židek	GTomkovaOL	4	1									0,0	4,0
43.	Ladislav Tlapák	G_Břeclav	−1	1									0,0	2,5
44.	Michal Kužela	GSlavičín	2	4	1					0			2,0	2,0

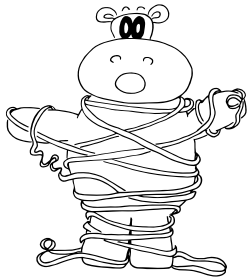
Pojďme se teď podívat na dláždění, která pro jednotlivé posloupnosti vzniknou. Na chvilku předpokládejme, že každé z nich má výšku právě V.

Pro $n/4$ prvních polovin máme $n/4$ druhých polovin, přičemž první polovině vždy odpovídá právě jedna ze druhých polovin. Na prostředním sloupci dláždění tedy musíme být schopní odlišit alespoň těchto $n/4$ různých možností, jinými slovy musí existovat $n/4$ možných obarvení prostředního sloupce.

Pokud označíme celkový počet barev v našem programu jako B, můžeme možná obarvení prostředního sloupce omezit také výrazem B^V. Tím dostáváme nerovnost $B^V \geq n/4$, úpravami získáme $V \geq \log_B n - \log_B 4$. To ovšem můžeme asymptoticky zapsat jako $V \in \Omega(\log n)$.

Rozhodnout správné závorkování tedy skutečně neumíme lépe než logaritmicky.

Pro úplnost ještě okomentujme náš drzý předpoklad, že dláždění pro každou z našich posloupností mělo výšku právě V. To samozřejmě nemusí být pravda, některá klidně mohla být jednořádková. Každé dláždění, které má výšku maximálně V ovšem umíme „nafouknout“ tak, aby mělo výšku právě V. Můžete si jako cvičení rozmyslet, jak toho dosáhnout.



Úkol 5 – Ekvivalence s Turingovými stroji

Začněme s asi snazším převodem, a to z dlaždičkových programů na Turingovy stroje. Necháme Turingův stroj simulovat stavbu dláždění, a to konkrétně po řádcích. Obarvení vodorovných hran budeme považovat za znaky na pásce vždy před provedením kroku a po něm, obarvení svislých hran pak za stavy.

Pro každou dlaždici (ℓ, h, p, d) tedy zavedeme pravidlo typu $(\ell, h) \to (p, d, \to)$. Tedy pokud se ve stavu ℓ na pásce nachází znak h, stroj zapíše znak d a přejde do stavu p, hlava se posune doprava.

Tímto umíme vybudovat jeden řádek dláždění. Řádků ale může být víc, potřebujeme se tedy umět vracet zpět. V případě, že aktuální stav odpovídá barvě pravého okraje zdi, řekněme P, a na vstupu se nachází mezera, přejde stroj do speciálního návratového stavu. V něm znaky vždy zapisuje zpět a posouvá se doleva, dokud opět nenarazí na mezeru.

Návrat nastartuje pravidlo $(P, \sqcup) \to (N, \sqcup, \leftarrow)$, během vracení potřebujeme pravidla typu $(N, x) \to (N, x, \leftarrow)$ pro všechna možná obarvení x a k ukončení návratu přidáme ještě pravidlo $(N, \sqcup) \to (L, \sqcup, \to)$.

Ještě musíme poznat, že je celé dláždění dokončené. Mohli bychom při návratu odlišovat dva stavy: jeden, kdy jsme dosud ze vstupu četli pouze znak odpovídající obarvení dolního okraje, a druhý, kdy se alespoň jeden znak lišil. My si ale představíme, že ve speciálním stavu vybudujeme ještě jeden řádek navíc.

Po skončení návratu umožníme přechod do speciálního stavu, $(N, \sqcup) \to (*, \sqcup, \to)$. V tomto stavu přijímáme pouze správný znak (tedy správné obarvení), a pouze pokud dojdeme až na konec vstupu, ukončíme výpočet. Zavedeme proto ještě dvě pravidla, $(*, D) \to (*, D, \to)$ a $(*, \sqcup) \to$ ANO.

Už víme, že dlaždičkové programy nejsou silnější než Turingovy stroje. Pojďme teď převést Turingův stroj na dlaždičkový program.

Tentokrát bude jeden řádek dláždění odpovídat jednomu kroku Turingova stroje. Musíme vědět, nad kterým znakem se právě nachází hlava a v kterém stavu se nachází celý stroj. Zároveň nám informaci o stavu stačí znát u znaku pod hlavou, ostatní znaky stejně výpočet ovlivnit nemohou.

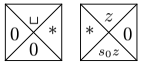
Pro každé pravidlo typu $(s_i, z_i) \to (s_j, z_j, \bullet)$ si pořídíme následující dlaždici:

Dále si pořídíme dlaždice pro pravidla, kdy se hlava posouvá, pokud bychom u předchozího pravidla měli posun hlavy doprava, potřebujeme následující dvojici „slepených“ dlaždic:

Nesmíme zapomenout, že většinu znaků ze vstupu v jednotlivých krocích nijak neměníme, ale dlaždice pro ně existovat musí. Taková dlaždice bude mít podobu

Koncového stavu dosáhneme při čtení nějakého konkrétního znaku, ale aby šlo dláždění dokončit, musí být správnou barvou obarvený celý poslední řádek. Pro jakékoli pravidlo typu $(s, z) \to$ ANO proto přidáme dlaždici (ANO, (s, z), ANO, ANO) a pro ostatní prvky abecedy dlaždice (ANO, z, ANO, ANO). Pouze pro okrajové mezery musí dlaždice vypadat jinak, (L, $\sqcup$, ANO, ANO), analogicky pro pravou stranu.

Zbývá si ještě rozmyslet, jak by měl vypadat začátek dláždění. Potřebujeme do vstupu dostat informaci o počátečním umístění hlavy a počátečním stavu. K tomu si zavedeme ještě jeden speciální inicializační stav. Pak nám budou stačit tyto dlaždice:



Tím je převod hotov, Turingův stroj umíme simulovat pomocí dlaždičkových programů, tedy tyto dva výpočetní modely jsou skutečně ekvivalentní.

Karolína „Karryanna“ Burešová

Jednu takovou objevil k večeru. Asi hodinu ji pozoroval, než se odvážil ukázat se. Zdálo se, že místní náčelník zrovna řešil nějaký problém s rozdělováním surovin na výrobu jídla.

26-5-4 Rozdělování jídla — 11 bodů

Vesnický náčelník má k dispozici K typů různých surovin a chce je co nejlépe využít, aby mu jich dohromady zůstalo co nejméně nepoužitých. Od každé suroviny má na začátku nějaké množství $m_1, \ldots, m_K$ ($m_i \geq 0$).

Obyvatelé vesnice mu předložili N receptů. Každý recept spotřebuje nějaké množství od každé suroviny, tedy je zadaný K čísly $a_1, \ldots, a_K$ ($a_i \geq 0$) a lze ho použít maximálně jednou (a to jen tehdy, pokud ještě od každé suroviny zbývá alespoň tolik, kolik recept spotřebuje).

Vaším úkolem je ze všech N receptů vybrat takové, které dohromady zanechají co nejmenší zbytek surovin (tedy pokud si zbytky označíme jako $z_1, \ldots, z_K$, tak $\sum_{i=1}^{K} z_i$ bude co nejmenší možné).

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.

Potom, co se s nimi Jacob najedl, poznal podle náhrdelníku s tajným heslem v jednom z domorodců člena Bratrstva. Počkal, než odejde stranou, a pak se za ním nenápadně vydal.

Ve chvíli, kdy si před ním Jacob sundal z hlavy kápi, se mimozemšťan zarazil, jako by ho někdo praštil palicí po hlavě. Pak ale jeho pohled sklouzl na náhrdelník Bratrstva, který Jacob dostal od Ady, a došlo mu to: „Ty jsi ten neznámý cizinec, který nám slíbil pomoc, že? Co děláš takhle daleko od hlavního tábora? A mimochodem, já jsem Lakus.“

Jacob mu všechno vysvětlil a také mu řekl, že by se potřeboval nenápadně dostat do královského města, aby si promluvil s králem. To, že jsou to taky pozemšťané jako on, raději nezmiňoval.

Mimozemšťan se zamyslel a pak pravil: „Pracuji se skupinkou řemeslníků, kteří do města dopravují kámen. Využíváme k tomu staré podzemní tunely, tudy by ses tam mohl dostat.“

* * *

Druhý den ráno už Jacob pomáhal tlačit velký povoz kamení naložený tak, že byl trojnásobně vyšší než on sám. Měl na sobě stále kápi, ale batoh s karabinou a největšími věcmi dal raději Lakusovi do úschovy, nechtěl aby se případně dostaly do rukou královým lidem.

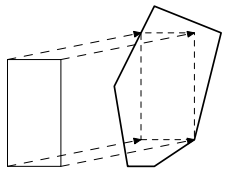
Právě se blížili k vstupu do tunelu, město se tyčilo na ostrohu nad nimi. Jacob si mimo jiné všiml pozemsky vyhlížející antény na největším z domů. Náhle je zastavilo zavolání jednoho z hlídačů u ústí do tunelu. Začal se hádat s předákem skupinky, jestli se takhle velký povoz do tunelu vejde, nebo jestli ho bude nutné složit tady a náklad odnosit ručně.

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

26-5-5 Průjezd tunelem — 12 bodů

Potřebujeme zjistit, jestli náš povoz projede tunelem. Povoz má průřez obdélníku rovnoběžného se souřadnými osami, průřez tunelu je představován konvexním mnohoúhelníkem (zadaným na vstupu třeba jako posloupnost vrcholů po směru hodinových ručiček).

Ptáme se, jestli se povoz vejde do tunelu, neboli jestli existuje nějaké posunutí obdélníku takové, že se celý obdélník vejde dovnitř konvexního mnohoúhelníku. Obdélník můžeme posouvat v libovolném směru (tedy klidně i nahoru a dolů), ale nesmíme ho otáčet. Pokud takové posunutí existuje, najděte ho.



Po proniknutí do města se Jacob oddělil od skupinky mimozemšťanů. Opatrně proklouzl okolo stráží a vyšplhal na vyvýšenou věž na okraji hradeb. Zde zalehl a vytáhl z kapsy dalekohled.

Pozorně si prohlédl antény, které už dříve spatřil. Podle toho, kam byly zaměřené, se pravděpodobně používaly jen k místní komunikaci. Jeho pohled upoutala vyčnívající kovová věc za královským palácem, ale ze své současné pozice tam neviděl. Rozhodl se počkat do noci a pak se přes střechy jednotlivých staveb opatrně přesunout blíže paláci.

Čekání na tmu využil k tomu, že si pro každou oblast města našel nejvyšší stavbu, kterou by při nočním přesunu mohl použít jako pozorovatelnu.

26-5-6 Nejvyšší stavby — 12 bodů

Chceme v mimozemském městě najít vždy lokálně nejvyšší stavbu. Plán mimozemského města je tvořen čtvercovou sítí o rozměrech $R \times S$, kde pro každé políčko známe výšku stavby, která se na něm nalézá. Dále máme zadaná čísla r a s (platí $r \leq R$ a $s \leq S$) – velikost oblasti, která nás zajímá.

Chtěli bychom pro každou oblast velikosti $r \times s$ zjistit, jakou nejvyšší stavbu obsahuje.

Výstupem programu by tedy měla být tabulka velikosti $(R - r + 1) \times (S - s + 1)$, kde políčko $[i, j]$ bude obsahovat maximální výšku stavby nalézající se v oblasti velikosti $r \times s$ s levým vrchním rohem v tomto políčku.

Ukázkový vstup:

```
r = 3, s = 3
1 4 2 2 0 2
0 2 2 2 1 1
2 1 3 3 0 1
8 1 0 5 0 6
```

Ukázkový výstup:

```
4 4 3 3
8 5 5 6
```

Lehčí varianta (za 6 bodů): Vyřešte úlohu jen pro jednorozměrný případ (tedy $R = r = 1$).

Padla noc. Jacob se plížil po střechách jen v černé kombinéze, pomáhaje si nočním viděním. Díky tomu, že si přesně zmapoval nejvyšší budovy ve městě, se dokázal snadno vyhýbat hlídkám. Metr po metru se blížil ke královskému paláci.

Seskočil z poslední střechy, obešel věžičku a naskytl se mu pohled na něco podivného. Samozřejmě tu věc poznával, za svoji kariéru ji viděl mnohokrát na spoustě hvězdných lodí. Nebo alespoň její modernější sourozence, tohle byl vážně starý model, ten už se skoro půlstoletí nepoužíval.

Co ho ale zaskočilo mnohem víc než shledání se starým záchranným raketoplánem Mark II, bylo to, že aktuálně vypadal jako vyvržený vorvaň. Jeho motory to asi odnesly při tvrdém přistání, aspoň podle škod na zádi. Plátování z celého okolí fúzního reaktoru bylo odstraněno a od odhalených součástek vedlo směrem do paláce několik tlustých svazků kabelů.

Další vedly z místa bývalého kokpitu a končily u soustavy antén na střeše vlevo od něj. Působilo to celé, jako kdyby se někdo rozhodl využít z raketoplánu každou použitelnou věc, začal ho přestavovat na obytný přívěs křížený s továrnou na boty a v půlce navíc ztratil výrobní plány.

Náhle ho vyrušil šoupavý zvuk po jeho pravici. Rychle vyskočil, otočil se a ztuhnul s rukou na šipkové pistoli. Z půl metru hleděl do zrcadlového hledí skafandru. Kriticky si uvědomil, že ho ten člověk má špatně nasazený, zámky helmy nebyly zacvaklé a navíc na sobě neměl rukavice. To mu však nijak nezabránilo v tom praštit Jacoba po hlavě kovovou tyčí.

⋆ ⋆ ⋆

Probudil se v temné místnosti a příšerně ho bolela hlava. Zahlédl vedle sebe nějakého člověka. Poprvé po pěti letech zase spatřil příslušníka lidské rasy!

„Omlouvám se za tu ránu na hlavě, můj syn je trochu zbrklý," začal člověk starým hlasem. „To víte, nespatřil nikdy nikoho jiného mimo nás, narodil se pět let po zroskotání."

„Jak jste tu dlouho?" zeptal se Jacob a mnul si bouli na hlavě.

„Naše loď, Hermes*, tu ztroskotala před asi 30 pozemskými roky, pokud počítám správně. Já jsem Cedric, nejvyšší přeživší důstojník," řekl stařec, „kapitán a většina ostatních důstojníků zemřeli, když naváděli loď někam do nejhlubšího moře... selhával nám reaktor a oni nechtěli zamořit místní ekosystém výbuchem."*

Hermes*, to mu něco říkalo. Nebyla to ta civilní loď, která se ztratila během války? Původně se myslelo, že byla v nějaké potyčce omylem zničena ale průzkum záznamů všech zúčastněných stran neukázal nic.*

„A od té doby tu využíváte domorodce?" přešel Jacob hned k jádru věci.

„Oni si myslí, že jsme bohové. A já nemám v úmyslu jim to vyvracet. Sem tam se sice objeví nějaký pochybovač, ale jak se říká, dostatečně pokročilou technologii nelze odlišit od magie – ty blázni se daj hrozně snadno přesvědčit. A když to nejde..." s těmi slovy si nadhodil v ruce původně Jacobovu šipkovou pistoli.

„Ty jsi tu spadl s tou velkou lodí před pár lety, ne? Pokoušeli jsme se do ní dostat, ale s našimi nástroji ze záchranného člunu jsme se nedokázali proříznout ani přes jednu přepážku. Ty ale určitě máš přístupové kódy, že?"

„Předpokládáte, že vám pomůžu?!?" zeptal se Jacob.

„Ty se nechceš nechat považovat těmihle tupci za boha?" podíval se Cedric na Jacobův znechucený výraz. „No dobrá, pár dnů na přemýšlení možná změní tvůj názor."

S těmito slovy odešel a nechal Jacoba samotného v cele. S vyhlídkou na dlouhý pobyt začal Jacob zkoumat své vězení. Na jedné zdi objevil dlouhý zápis jedné z oblíbených mimozemských her, asi si tu nějaký vězeň taky krátil dlouhé čekání. Hra se docela podobala pozemským piškvorkám a Jacoba by zajímalo, jak vlastně dopadla.

26-5-7 Partie piškvorek — 13 bodů

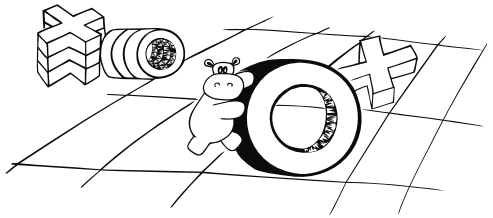
Jacob našel zápis jedné partie mimozemských piškvorek, která se hraje se na čtvercové síti o rozměrech $N \times N$. Zápis hry je tvořen třemi typy tahů:

- Na políčko $[A, B]$ umísti kolečko
- Na políčko $[A, B]$ umísti křížek
- Vymaž znak z políčka $[A, B]$

Po každém tahu by nás zajímalo, jak je dlouhá aktuálně nejdelší souvislá řada symbolů (v řádce, sloupci nebo na úhlopříčce). Vybudujte datovou strukturu držící aktuální stav hrací desky, která by navíc měla zvládat rychle zapisovat jednotlivé tahy a po každém tahu rychle vypsat aktuálně nejdelší řadu.

Předpokládejte, že tahů bude řádově stejně, jako je velikost hrací desky (tedy řádově N^2). Tahy budou vždy korektní, tedy nebude docházet k mazání prázdného políčka ani k umisťování znaku na neprázdné políčko.

⇧ **Lehčí varianta (za 5 bodů):** Vyřešte úlohu pro partii, v níž se znaky nebudou mazat (nastanou jen tahy prvních dvou typů).



Po zamotání se v partii piškvorek Jacob ulehl na tvrdou zem – ráno moudřejší večera, pomyslel si. Uprostřed noci ho ale probudil slabý hlas. Chvíli přemýšlel, jestli se mu to nezdálo, ale když Adu zaslechl znova, okamžitě vyskočil. Dívala se na něj skrz malé okénko a podávala mu jeho plný batoh.

Bez rozmýšlení ho popadl, vyndal z něj plazmovou karabinu, připjal si meč k boku a řekl Adě: „Radši ustup." Pozvedl zbraň, zacílil a několika přesnými výstřely zničil mříž. Pak se vzniklým otvorem protáhl k Adě. „Jak..." začal, ale Ada ho umlčela. Ukázala mu rukou a rozeběhla se do temnoty.

Nedostali se ale daleko. Na nádvoří, kus od raketoplánu, je obklíčili strážci s ostrými oštěpy. „Já je nechci zranit, Ado," sykl, „není tu nějaká jiná cesta?"

„Máš meč? Vytáhni ho, dělej!" řekla rychle Ada.

Jacob si spustil karabinu na popruhu dolů a tasil meč. Co se stalo dál, nechápal. Strážní upřeli pohled na blyštivou čepel zdobenou kameny, začali si něco mumlat a ustupovali. Zaslechl něco o proroctví a prorokovi. Co ho ale udivilo víc, bylo to, že jeden z kamenů na meči začal v blízkosti fúzního reaktoru raketoplánu nezvykle pulzovat. Vrhl rychlý pohled do změti kabelů a pak mu to došlo.

„Ado, chyť ten meč a drž je od nás dál!"

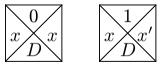
Ada opatrně uchopila meč, jako by se bála, že se o něj spálí. Jacob začal kuchat obnažené zařízení, než se mu po-

26-4-8 Dlaždičky

Úkol 1 – Počet jedniček dělitelný třemi

K řešení prvního úkolu nám bude stačit poměrně jednoduchá myšlenka. Budeme si zleva doprava propagovat dosavadní počet jedniček modulo 3 (úplně stejný trik bychom mohli použít pro libovolnou jinou konstantu).

Dlaždice si pořídíme dvojího typu, jedny budou mít nahoře nulu, druhé jedničku. Ty s nulou předávají stejnou hodnotu, tedy mají levou i pravou stranu obarvenou stejně. Naopak ty s jedničkou předávaný počet zvyšují, takže mají vlevo x a vpravo $(x + 1) \bmod 3$. Dole všechny dlaždice obarvíme barvou D. Naše dlaždice tedy vypadají takto:



pro $x' = (x + 1) \bmod 3$

Dolnímu okraji přidělíme barvu D, levému i pravému okraji pak barvu 0.

Pro vstup, který má počet jedniček dělitelný třemi, určitě umíme dláždění vytvořit (přesně podle myšlenky řešení jen počítáme jedničky ve vstupu modulo 3).

Dláždění buď vůbec nelze vytvořit, nebo je určené jednoznačně. Pro daný vstup máme právě jednu možnost, jakou dlaždici přiložit přímo k levému okraji, pak právě jednu možnost, jakou dlaždicí navázat na tuto první atd. Dláždění tak vždy počítá jedničky modulo 3, tedy pokud dláždění existuje, vstup skutečně podmínku splňoval.

Všimněte si, že máme levý i pravý okraj zdi obarvený stejnou barvou. Tím jsme si ušetřili řešení okrajových případů. Chceme-li ovšem takový trik provést, musíme si velmi dobře rozmyslet, jestli si tím nepokazíme správnost. Obecně tím totiž tvrdíme, že zřetězíme-li několik vstupů splňujících naši podmínku, výsledek ji bude splňovat také.

Úkol 2 – Snižování výšky dláždění o konstantu

Představme si, že máme dláždění výšky t, a podívejme se na libovolný sloupec tohoto dláždění. Obarvení horizontálních stran dlaždic, vyjma horní stěny nejvyšší a spodní stěny nejnižší dlaždice, nemá pro zbytek dláždění žádný význam, určuje pouze návaznosti v tomto sloupci.

Naopak obarvení vertikálních stran důležité je, a to po celé výšce sloupce, určuje totiž návaznosti v rámci jednotlivých řádků. Toto obarvení tedy chceme zachovat.

Abychom snížili výšku dláždění, nahradíme každý sloupec jedinou dlaždicí. Očíslujme si dlaždice ve sloupci od 0 do $t-1$ (dlaždice v nultém řádku přiléhají ke vstupu), obarvení každé z nich pak (ℓ_i, h_i, p_i, d_i). Nová dlaždice pak bude nahoře obarvena barvou h_0, dole barvou d_{t-1}. Pro boční stěny zavedeme nové barvy, které označíme jako uspořádanou t-tici $(\ell_0, \ldots, \ell_{t-1})$, resp. $(p_0, \ldots, p_{t-1})$.

Pro úplnost dodejme, že levý okraj přebarvíme z barvy ℓ na $(\ell, \ldots, \ell)$, podobně pravý.

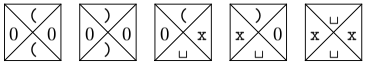
Sloupce k sobě přiléhají stále stejně, ovšem výšku dláždění jsme z t zmenšili na 1. Velikost množiny D, tedy množiny všech použitelných dlaždic, nepovažujeme za důležitou, přesto stojí za zmínku, že jsme ji tímto trikem exponenciálně zvětšili.

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

Úkol 3 – Dlaždičkové závorkování

Kdybychom chtěli správnost závorkování ověřovat přímočaře, mohli bychom postupně umazávat dvojice `()`, které určitě správně jsou. Místo opravdového smazání takových dvojic a následného posouvání zbytku řetězce se hodí „smazané" závorky jen něčím nahradit a vstup nechávat na původním místě.

Potřebujeme tedy vždy nahradit dvě závorky `()`, mezi kterými buď není vůbec nic, nebo jsou pouze smazané znaky. Ostatní závorky musíme poslat o řádek níž, na konci chceme mít smazaný celý vstup. Toho můžeme dosáhnout například s následující sadou dlaždiček:



Levý a pravý okraj obarvíme barvou 0, dolní barvou ␣.

V nejhorším případě budeme ovšem potřebovat $n/2$ kroků na smazání celého vstupu, program tedy pracuje v lineární časové složitosti. To přece musí jít lépe!

Závorkování rychleji

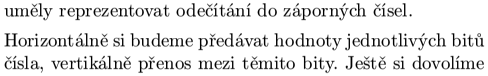
Abychom dosáhli lepší než lineární složitosti, vypůjčíme si trik z řešení 26-1-8.[12] Budeme počítat úroveň zanoření, neboli kolik jsme zatím potkali neuzavřených závorek. A protože máme jen konečnou množinu barev, budeme si tuto úroveň ukládat jako číslo ve dvojkovém zápisu.

Úroveň zanoření může být u správného závorkování nejvýše $n/2$, na reprezentaci takového čísla ve dvojkovém zápisu potřebujeme $\log n$ bitů. My si dvojkové zápisy budeme ještě doplňovat tak, aby vždy začínaly nulou, ale tím jsme přidali jeden řádek. Náš program tak bude pracovat v logaritmické časové složitosti.

Vždy, když potkáme otvírací závorku, úroveň vnoření o 1 zvýšíme, u zavíracích závorek ji zase o 1 snížíme. Zápis úrovně nám bude vznikat směrem dolů, jinak řečeno, při hotovém dláždění ho můžeme číst na pravých stranách dlaždic v daném sloupci. Navíc si nepořídíme dlaždice, které by uměly reprezentovat odečítání do záporných čísel.

Horizontálně si budeme předávat hodnoty jednotlivých bitů čísla, vertikálně přenos mezi těmito bity. Ještě si dovolíme jeden podlý trik, a to ten, že vertikální přenosy budeme kódovat opět závorkami. Díky tomu totiž nemusíme nijak odlišovat přikládání dlaždic ke vstupu a k předchozímu řádku dláždění.

Pro úplnost dodejme, že všechny tři okraje zdi obarvíme 0, a pak se konečně pojďme podívat na používané dlaždice:



Úkol 4 – Minimální složitost závorkování

Zkusme si pořídit různé posloupnosti závorek délky n. Řekněme, že budeme posloupnosti budovat z dvojic závorek, že nejprve vybudujeme první polovinu této posloupnosti a že v první polovině smíme použít pouze dvojice `()` a `((`.

Takto jistě můžeme vytvořit $n/4$ posloupností délky $n/2$ takových, že žádné dvě nemají stejný počet otevřených závorek. Nyní každou z nich doplňme pomocí dvojic `()` a `))` na správně uzávorkovanou posloupnost délky n.

26-4-6 Sněhová bitva

Nejjednodušší řešení by bylo nepředpočítávat si vůbec nic. Při dotazu, zda na polopřímce $\overrightarrow{AB}$ je ještě nějaký další bod, jednoduše spočítáme směr z bodu A do všech ostatních a porovnáme se směrem do B. Jak spočítat a reprezentovat směr rozebereme níže, prozatím předpokládejme, že jde směr spočítat i porovnat v konstantním čase. Takové řešení má dotaz za $\mathcal{O}(n)$ a potřebuje $\mathcal{O}(n)$ paměti na uložení vstupu. Ale za takto jednoduchou úlohu by asi organizátoři 11 bodů nenabízeli.

Tak co bychom si mohli předpočítat? Co třeba odpověď na každý možný dotaz? Protože odpověď na takový dotaz je jen ANO/NE a počet dvojic je $\mathcal{O}(n^2)$, bude nám stačit dvourozměrné pole. Ale předpočítat si ho pomocí výše zmíněného jednoduchého řešení by trvalo $\mathcal{O}(n^3)$. To zkusíme zrychlit.

Zkusme si spočítat naráz všechny odpovědi pro jednoho mířícího bojovníka A a všechny možné cíle C. Tedy odpovědi pro všechny polopřímky $\overrightarrow{AX}$. Uděláme to tak, že spočítáme směry pro všechny cíle a tyto směry setřídíme. Tím se nám stejné směry „sesypou“ k sobě. Nyní můžeme směry ještě jednou projít a najít úseky stejných směrů a jim odpovídající cíle označit, že leží na společné polopřímce, a tedy že pro ně bude odpověď ANO. Toto nám zabere $\mathcal{O}(n \cdot \log n)$ pro jednoho mířícího bojovníka, tedy $\mathcal{O}(n^2 \cdot \log n)$ celkem. Spotřebujeme $\mathcal{O}(n^2)$ paměti a odpovídat můžeme v konstantním čase.

Nyní, co s tím směrem? Nejpřirozenější reprezentace by asi byla počítat úhel. To s sebou ale nese jisté technické problémy (jako například trochu pomalejší výpočet trigonometrických funkcí či chlupatá iracionální čísla, co se kamarádí s π i pro rozumná vstupní data). Budeme tedy počítat poměr rozdílu v ose X a Y. To má stále jisté problémy. Jednak tento poměr bude pro protilehlé směry vycházet stejně – směr doleva dolů a směr doprava nahoru bychom nerozlišili. Tedy, směr bude dvousložková hodnota. První složka bude říkat, jestli je to směr nahoru, nebo dolů (resp. doleva, či doprava v případě vodorovných směrů, ať se vyhneme kladné a záporné nule). Druhým problémem je dělení nulou pro svislé směry. Ale v takovém případě si budeme nějakým způsobem reprezentovat nekonečno (např. čísla s plovoucí čárkou opravdu mají speciální hodnotu pro nekonečno).

Nakonec jedna poznámka o tom, že to může jít lépe. Pokud bychom použili hešování místo třídění pro shluknutí stejných směrů dohromady, dostali bychom se na očekávanou složitost $\mathcal{O}(n)$ (bohužel jen očekávanou, ne nejhorší případ) pro jednoho střelce, namísto $O(n \cdot \log n)$. To lze udělat proto, že nám vlastně vůbec nezáleží na pořadí, v jakém směry dostáváme, jen aby byly ty stejné pohromadě.

To má ale háček. Čísla s plovoucí čárkou mají v počítači tendenci akumulovat při operacích chybu. Proto je třeba je porovnávat s malou tolerancí, nikoliv na přesnou rovnost, a to s hešovací tabulkou nejde. Pokud bychom ale měli celočíselný (nebo alespoň racionální) vstup, můžeme směr ukládat jako zlomky a poté nám již hešovací tabulka pomůže. Ale převést zlomek na základní tvar nezvládneme v konstantním čase. Všimneme si však, že když jsou čísla na vstupu omezená hodnotou L, dva různé zlomky se budou lišit alespoň o $1/L^2$. Vynásobíme tedy všechny směry L^2 a budeme dělit celočíselně.

Michal „vorner“ Vaner

26-4-7 Královští špioni

Rozplést síť špionů většinou nebývá snadné. V této úloze jsme ji ale měli již pěkně naservírovanou.

Nejjednodušším, avšak pomalým řešením je odsimulovat putování zpráv postupně od každého špiona. K tomu nám stačí si vstup načíst do pole A. Pak už jen v jedné proměnné udržujeme počet kroků a ve druhé pozici zprávy, tj. číslo špiona, u kterého se zrovna nachází. Jedno přeposlání zprávy provedeme snadno přiřazením `pozice = A[pozice]`.

Jak poznáme, kdy se má předávání zpráv zastavit? Jednoduše si budeme v dalším poli pamatovat, kteří špioni tuto zprávu viděli. Uvedené řešení má časovou složitost $\mathcal{O}(N^2)$, kde N je počet špionů. Za takové CodEx uděloval tři body.

Problém s pomalostí spočívá v tom, že procházíme v síti špionů stále dokola ta stejná místa, přestože výsledek je vždy stejný. Při každém průchodu můžeme rovnou uložit počty kroků pro všechny špiony, přes které jsme zprávu posílali.

Všimněme si, že každá zpráva končí své putování nějakým *cyklem* – částí, ve které si špioni posílají informace v kruhu. Zpráva vyslaná libovolným špionem na cyklu bude předaná přesně tolikrát, kolik je v daném cyklu špionů. V úseku před cyklem roste doba putování postupně o jedna za každého špiona.

Asymptoticky jsme si zatím nepomohli. Nic nám totiž nezaručuje, že špiony budeme procházet ve správném pořadí od toho nejvzdálenějšího. Musíme začít využívat to, co jsme již spočítali. Když při simulaci dojdeme ke špionovi, pro nějž máme výsledek spočítaný, nebudeme pokračovat dál, protože bychom se stejně nic nového nedozvěděli. K počtu kroků jen přičteme výsledek pro aktuálního špiona.

Tím už pro každého špiona provedeme jenom konstantně mnoho operací. Celková časová složitost tak je $\mathcal{O}(N)$. Pro úplnost ještě dodejme, že s použitý trikem se můžeme setkat častěji. Dokonce si vysloužil vlastní název *memoizace*.

Program (C):
`http://ksp.mff.cuni.cz/viz/26-4-7.c`

Jenda Hadrava

vedlo uvolnit vlnový rezonátor montovaný na tyto staré reaktory. Okamžitě se spustil nouzový protokol a reaktor se odstavil, palác náhle potemněl.

Ada s Jacobem využili nastalý zmatek a rychle unikli z města. Tam na ně čekalo pár dalších členů Bratrstva s tvory vzdáleně podobnými koňům. Cesta nazpět k Freye *jim nezabrala ani tři dny.*

Během zastávek vyrazil Jacob z meče, ke zděšení ostatních, několik kamenů a něco s nimi a s ukradeným rezonátorem dělal. Co, to nechtěl říct, ale naléhal, že jako první musí k vraku lodě. Také se cestou věnoval sepisování nějakých věcí do elektronického zápisníku.

Když tam dorazili, vylezl po trupu nahoru až k troskám anténního pole. Vůbec neočekával, že by se na téhle planetě mohlo vyskytovat thallium v krystalické podobě. Pokud tohle vyjde, mohl by přes něj pomocí toho prastarého rezonátoru vyslat krátký subprostorový impulz, který by mohl proniknout vším tím rušením okolo.

Potom, co všechno správně pospojoval, usedl k jednomu z řídících panelů. Přehrál do paměti lodního počítače připravenou zprávu z elektronického zápisníku. Zhluboka se nadechl a potvrdil příkaz. Lodní počítač ve zlomku sekundy přetížil anténní systém a v efektním záblesku spálil krystal na prach...

* * *

Ve sledovacím centru vesmírného provozu na Zemi zrovna Bob pojídal sendvič, když vtom začal jeho počítač varovně pípat. Líně zamáčkl spínač poplachu a podíval se, co za planý poplach to je teď. Sendvič mu vypadl z ruky a o sekundu později sahal po interkomu: „Šéfe... Ano, vím, že jsou dvě v noci, ale tohle musíte vidět!“

Pokračování příběhu na podzimním soustředění...

Další část příběhu o Jacobovi pro vás sepsal

Jirka Setnička

26-5-8 Automatizovaný graf — 15 bodů

Ve všech čtyřech minulých dílech tohoto seriálu jsme se potulovali po pomyslné zoo výpočetních modelů a zastavovali se u zajímavých exemplářů. Dnes naši letošní pouť zakončíme u jednoho podivného výběhu, ve kterém se nepase jedno velké výpočetní zvířátko, ale spousta maličkých. Řeč bude o *grafových automatech*.

Úvod

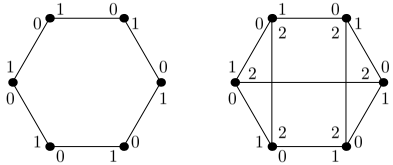
Grafový automat se skládá ze spousty stejných jednoduchých automatů (můžeme si je představit třeba jako malé programy, omezení viz níže), které jsou nějakým způsobem pospojovány a společně řeší nějaký složitější problém. Abychom si nepletli pojmy, budeme dále grafovému automatu jako celku říkat *grafomat* a pojmem *automat* budeme označovat jednotlivé malé automaty obsažené v grafomatu.

Řečeno formálně, grafomat si můžeme představit jako obyčejný neorientovaný graf[2] tvořený množinou *vrcholů* a množinou *hran* mezi nimi. V každém vrcholu sídlí jeden automat a hrany nám vyjadřují to, jak jsou automaty mezi sebou propojené. Pro účely tohoto seriálu si dovolíme pojmy vrchol a automat ve vrcholu zaměňovat.

Aby navíc mohl každý automat rozpoznat své sousedy, jsou v každém vrcholu všechny hrany, které z něj vychází, očíslovány navzájem různými čísly $0, 1, 2, \ldots$ Jedna hrana přitom na obou koncích může (ale nemusí) dostat různá čísla, takže spíše než hrany číslujeme konce hran. Pokud budeme mluvit o nějaké hraně z pohledu určitého automatu, budeme její konce nazývat *místní* a *protější*.

Navíc budeme pro jednoduchost předpokládat, že ze všech vrcholů vede stejný počet hran. Grafům s touto vlastností se říká *regulární*, nebo přesněji K-regulární, kde K je počet hran vycházejících z vrcholu. To mimochodem znamená, že v celém grafu se nachází právě $N \cdot K/2$ hran, neboť každá hrana má 2 konce a konců napočítáme $N \cdot K$.

Ukázku 2-regulárního a 3-regulárního grafomatu na 6 vrcholech s očíslovanými hranami můžete vidět níže.



Automaty a jejich paměť

Už víme, jak grafomat vypadá jako celek, ale ještě bychom si měli popsat, jak vypadají jednotlivé automaty ve vrcholech. Jak už jsme řekli výše, všechny automaty musí být stejné – navzájem se budou odlišovat jen tím, co který z nich dostane na vstupu, a tím, co uvidí okolo sebe.

Formálně vzato, budou to *konečné automaty*, o kterých jsme psali seriál ve 23. ročníku. Abychom je nemuseli precizně definovat, budeme si je raději představovat jako velmi jednoduché programy. V příkladech a v řešeních budeme programovat v Pythonu, svá řešení můžete psát ve svém oblíbeném jazyce, pokud si myslíte, že je na to vhodný. Zavedeme ale několik omezení, aby možnosti našich programů odpovídaly možnostem konečných automatů.

Předně omezíme paměť: Každý automat si může pamatovat jen konstantně mnoho bitů informace nezávisle na velikosti vstupu. Můžeme použít libovolný pevný počet proměnných nějakého libovolného, ovšem omezeného rozsahu. Smíme si například pamatovat číslo od 0 do 42, ale nemůžeme si pořídit proměnnou, ve které bychom si spočítali počet vrcholů grafu – taková proměnná by musela mít horní mez závislou na velikosti vstupu, což nemáme dovoleno.

Druhé omezení vyplývá z prvního: V programech jednotlivých automatů nemůžeme použít *rekurzi* a pro všechny cykly musí existovat konstantní horní mez na počet iterací.

Zbývá určit, jak spolu mohou automaty komunikovat. Kdykoliv jsou dva automaty propojeny hranou, vidí si navzájem do paměti. Mohou si do ní ovšem jen nahlížet, ne ji jeden druhému přepisovat. Pro přístup k paměti sousedů máme v každém automatu k dispozici dvě pole indexovaná místním číslem hrany (tedy od 0 do $K-1$):

- $P[i]$ obsahuje protější číslo hrany s místním číslem i.
- $S[i]$ přistupuje k proměnným souseda připojeného hranou s místním číslem i. Pomocí konstrukce $S[i].promenna$ přečteme libovolnou sousedovu proměnnou. Jen se nesmíme odkazovat na jeho pole P a S, tedy není možné se například zeptat na proměnnou sousedova souseda.

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

Průběh výpočtu a jeho ukončení

Jak bylo naznačeno výše, výpočet probíhá v taktech. V každém taktu se automat může podívat na stav proměnných svých sousedů a podle nich a svého vlastního stavu provést nějaký výpočet a modifikovat vlastní proměnné. (Pokud během taktu soused své proměnné změnil, stále vidíme jejich stav z počátku taktu.)

Když se automat rozhodne, že už pro něj veškerá práce skončila, může zavolat speciální instrukci `stop`. Od této chvíle až do konce běhu celého grafomatu už tento automat nevykoná žádnou akci. Jeho sousedé stále mohou číst jeho proměnné, ale už není žádný způsob, jak by jeho výpočet mohl být opět nastartován. Poté, co instrukci `stop` zavolají všechny automaty v grafu, končí celý výpočet.

Druhou možností ukončení výpočtu je *ustálení*. Tím se myslí, že se dva takty po sobě nezmění v žádném automatu hodnota jakékoliv proměnné (rozmyslete si, proč potřebujeme dva takty a nestačí nám jeden – souvisí to s tím, že automaty vidí stav proměnných souseda jakoby o tah nazpět).

Pokud se grafový automat nikdy celý nezastaví (ani instrukcí `stop` ani ustálením), je to špatně a takový program je chybný.

Vstup je realizován tak, že se před prvním taktem výpočtu objeví ve smluvených proměnných každého automatu vstupní hodnoty (jaké a v jakých proměnných, to záleží na úloze). Výstup je obdobný, po konci výpočtu by se ve všech automatech měla ve smluvených proměnných nacházet správná výstupní hodnota.

Při počítání složitosti nás bude zajímat jen počet taktů do zastavení celého grafového automatu (na rychlosti výpočtu jednotlivých automatů ve vrcholech nezáleží). Odhad počtu taktů stačí dělat jen asymptoticky (pomocí $\mathcal{O}$-notace), pokud to konkrétní úloha nebude vyžadovat jinak.

Příklad 1 – hledání dosažitelných vrcholů

Zadání: Mějme 5-regulární graf na N vrcholech a v každém vrcholu proměnnou a. Na začátku bude ve všech vrcholech $a = 0$ s jedinou výjimkou vrcholu A, kde bude $a = 1$. Po konci výpočtu by mělo být $a = 1$ ve všech vrcholech, kam lze z vrcholu A po hranách grafu dojít.

Řešení: Použijeme princip prohledávání grafu do šířky – každý vrchol bude sledovat své sousedy a ve chvíli, kdy se v nějakém z nich objeví $a = 1$, sám si také své a nastaví na 1. Až se hodnoty a ustálí, výpočet se přirozeně zastaví. Program pro jednotlivý automat bude vypadat následovně:

```
# Proměnné:
# a - rozsah 0..1
# i - rozsah 0..4, výchozí hodnota 0

for i in range(5):
        if S[i].a == 1:
                a = 1
```

Program vykoná nejvýše N taktů. Nejpomaleji poběží, pokud má graf tvar cesty. Jestliže bude graf „hustší“, program může doběhnout mnohem rychleji – například pro úplný graf vykonáme jen 3 takty: v prvním se všude nastaví $a = 1$ a zbylé dva slouží pro ustálení.

Příklad 2 – nalezení protějšího vrcholu

Zadání: Mějme 2-regulární graf na N vrcholech složený z jediného cyklu sudé délky (graf je tedy sudá kružnice délky N). Na začátku je jeden vrchol označen: má $x = 1$, zatímco ostatní $x = 0$. Chceme najít protější vrchol a také ho označit (nastavit mu $x = 1$).

Řešení: Pošleme si po kružnici signály směrem doleva i doprava a ve chvíli, kdy se potkají, tak víme, že jsme nalezli vrchol přesně naproti. Zde pro ukázku použijeme zastavení pomocí `stop`, i když by šlo napsat i verzi zastavující ustálením.

```
# Proměnné:
# x - rozsah 0..1
# signal - rozsah 0..1, výchozí hodnota 0

if x == 1:
        # Vyšleme úvodní signál na obě
        # strany a skončíme
        signal = 1
        stop

if S[0].signal and S[1].signal:
        # Dostali jsme signál z obou stran
        x = 1
        stop
elif S[0].signal or S[1].signal:
        # Signál přišel alespoň z jedné strany
        signal = 1
        stop
```

Celý výpočet poběží právě $N/2$ taktů.

Úkoly

Úkol 1 [3b]: Mějme 2-regulární graf na N vrcholech (N je dělitelné třemi) složený z jediného cyklu. Na začátku je jeden vrchol označený: má $x = 1$, zatímco ostatní vrcholy mají $x = 0$. Vaším úkolem je označit zbylé dva vrcholy ve třetinách kružnice. Tedy pokud si startovní vrchol označíme indexem 0, tak po konci běhu grafového automatu budou označeny právě vrcholy s indexy 0, $N/3$ a $2N/3$.

Pokud vám to pomůže, můžete předpokládat, že každý vrchol je spojený hranou s místním číslem 1 se sousedem po směru hodinových ručiček a hranou s místním číslem 0 s druhým sousedem (jako na obrázku v úvodu).

Úkol 2 [3b]: Mějme 5-regulární souvislý graf s jedním označeným vrcholem (bude mít na začátku proměnnou $a = 1$, ostatní vrcholy ji budou mít nulovou). Vaším úkolem bude najít nějakou *kostru* tohoto grafu, tedy nějakou podmnožinu hran takovou, že stále spojuje všechny vrcholy, ale neobsahuje žádný cyklus.

Pro výstup použijte pole proměnných *kostra*[i] obsahující pět prvků. Na začátku bude toto pole v každém vrcholu plné nul, na konci by v něm měly být jedničky právě na pozicích, které odpovídají místním číslům hran, které jsou v nalezené kostře (pozor, pamatujte na to, že místní a protější číslo hrany nemusejí být stejné a že je nutné nastavit pole *kostra*[i] na obou koncích hrany).

V předchozích dvou úkolech stačilo odhadovat počet taktů asymptoticky, nebylo by ale zajímavé zkusit si vyrobit program, o kterém budeme vědět úplně přesně, jak dlouho poběží?

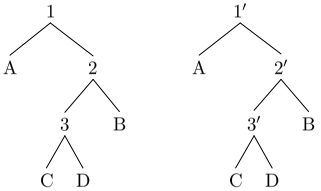
Úkol 3 [4b]: Vyrobte program, který bude na K-regulárním grafu na N vrcholech ($N \geq 1$) běžet přesně $C \cdot N$ kroků, kde C bude nějaká konstanta, která může záviset na velikosti K. Pokud se vám však povede C stanovit pevně (bez závislosti na K), bude to ještě lepší. Pokud vám to pomůže, můžete, jako v předchozích úkolech, předpokládat, že právě jeden vrchol bude nějakým způsobem označen.

Persistentní datová struktura

Největším problémem konstrukce našeho 2D-intervalového stromu bylo to, že stromy v druhé úrovni vypadaly pořád skoro stejně, ale pokaždé jsme je museli konstruovat celé znovu. Nedá se nějak využít již dříve zkonstruovaných stromů?

Dá, a takovému triku se říká *persistence*. V našich stromech vedou všechny hrany jen dolů (nepotřebujeme zpětné hrany do otce), a tak klidně můžeme udělat to, že si zkonstruujeme nový strom tak, že se bude odkazovat na části již postavených stromů. Stačí se držet základního pravidla persistence: Staré hodnoty nikdy neměním, když potřebuji něco změnit, vyrobím si od toho kopii.

Kdybychom například chtěli ve stromu níže změnit hodnotu ve vrcholu s číslem 3, stačí nám vyrobit si kopii tohoto vrcholu, kopii jeho otce (a tak dále, až ke kopii jeho kořenu) a ty odkázat na již existující podstromy. Když se nyní vydám z kořene $1'$, budu mít k dispozici nový strom, ale když se vydám z kořene 1, uvidím strom v původním stavu. Máme tedy dva stromy velikosti N, ale výrobou nového jsme museli strávit pouze čas $\mathcal{O}(\log N)$ (délku cesty z vrcholu do kořene).



V našem případě persistenci použijeme k vyrábění jednotlivých intervalových stromů v pásech. Mezi jednotlivými pásy dochází jen k tomu, že nějaký obdélník zmizí, nebo se jiný objeví, tedy potřebujeme přičíst nebo odečíst nějakou hodnotu od nějakého intervalu hodnot. Počkat, co když ale budeme muset modifikovat řádově N vrcholů oproti předchozímu stromu, nepokazí se nám časová složitost?

Kdybychom to dělali jako dosud (se zjednodušenými intervalovými stromy), tak by se pokazila. Teď už budeme potřebovat plné intervalové stromy (tak, jak jsou popsané v kuchařce), které nám umožňují i rychlou aktualizaci hodnot. Pokud chceme modifikovat nějaký celý interval, můžeme to rozložit na modifikaci $\mathcal{O}(\log N)$ vrcholů intervalového stromu (do vnitřních vrcholů si přičteme nějaké plus a mínus jedničky, přes které pak při dotazu projdeme a vezmeme je v potaz).

Modifikace každého vrcholu v persistentním intervalovém stromu nám trvá $\mathcal{O}(\log N)$, takže nám (kromě prvního) výroba každého z $\mathcal{O}(N)$ intervalových stromů trvá $\mathcal{O}(\log^2 N)$ a zabere maximálně stejně paměti. Vyhledávání v nich je pak stejné jako v minulém případě.

Dohromady tedy potřebujeme na výrobu celé struktury čas $\mathcal{O}(N \log^2 N)$ a na Q dotazů čas $\mathcal{O}(Q \log N)$. Paměti nám stačí $\mathcal{O}(N \log^2 N)$.

Program (C):
`http://ksp.mff.cuni.cz/viz/26-4-4.c`

Jirka Setnička

[11] O intervalových stromech se můžete dočíst v naší kuchařce.

Poznámka: Algoritmus můžeme ještě trochu vylepšit, pokud si všimneme, že při přidávání či odebírání obdélníka sice upravujeme až logaritmický počet vrcholů, ale všechny leží v blízkém okolí dvou cest z kořene do listu. Pokud persistenci naučíme zaznamenat všechny tyto změny najednou, zkopírujeme celkově jen $\mathcal{O}(\log N)$ vrcholů stromu, takže výroby struktury klesne na $\mathcal{O}(N \log N)$.

Dalšími triky by šlo snížit paměťové nároky na $\mathcal{O}(N \log N)$, ale to už se do tohoto textu nevejde. Kdybyste chtěli, příležitostně se zeptejte :)

Martin „Medvěd“ Mareš

26-4-5 Místo pro tábor

Úlohu budeme řešit jen pro těžší variantu. Řešení lehčí varianty je úplně stejné, akorát jen v jednom rozměru. Zadáním úlohy je najít obdélník velký $r \times s$, kde potřebujeme převézt co nejméně hlíny na jeho vyrovnání. Tedy takový, v kterém na políčkách pod průměrnou výškou obdélníka chybí co nejméně hlíny. Pro další část řešení zadefinujeme $n = rs$ a $N = RS$.

Pokud v daném obdélníku máme průměrnou výšku V a právě k políček s podprůměrnou výškou $p_1, \ldots, p_k$, musíme převézt právě

$$k \cdot V - \sum_{j=1}^{k} p_j$$

hlíny. Chtěli bychom tedy pro všechny možné obdélníky $r \times s$ tyto hodnoty spočítat.

Nejdříve spočítáme průměrné výšky. Pomocí dvourozměrných prefixových součtů snadno zvládneme spočítat součty výšek v obdélnících v čase $\mathcal{O}(RS)$ a tyto hodnoty vydělíme n. Průměrné výšky si setřídíme a označíme jako $X_1, X_2 \ldots, X_{(R-r+1)(S-s+1)}$. Zároveň zvolíme $X_0 = 0$.

Pro výpočet hodnot $\sum_{j=1}^{k} p_j$ se nám budou hodit dva dvourozměrné součtové intervalové stromy S a T.[11] Do stromu T budeme iterativně přidávat hodnoty výšek v původní mapě a do stromu S budeme vkládat jedničky na místa, kde se přidané prvky vyskytují.

V i-té iteraci do stromů vložíme všechny hodnoty z intervalu $\langle X_{i-1}, X_i)$. Pokud si navíc pro každé X_i budeme pamatovat, ke kterému obdélníku $r \times s$ patří, rovnou pro něj zvládneme spočítat hodnoty k a $\sum_{j=1}^{k} p_j$. Zde je pseudokód celé jedné iterace:

1. Do intervalových stromů přidáme všechna políčka, jejichž hodnota je v intervalu $\langle X_{i-1}, X_i)$.
2. Ze stromu S získáme hodnotu k pro obdélník patřící k X_i.
3. Ze stromu T získáme hodnotu $\sum_{j=1}^{k} p_j$ pro obdélník patřící k X_i.

A to je celé. Do každého intervalového stromu vložíme každou z N hodnot maximálně jednou, tedy dostáváme časovou složitost $\mathcal{O}(RS \cdot \log R \cdot \log S)$. Hodnoty výšek a průměrů máme setříděné a zpracováváme je ve vzestupném pořadí.

Posíláme gratulaci Michalu Punčochářovi, který tuto úlohu jako jediný vyřešil optimálně a zcela správně.

Karel Tesař

26-4-3 Obnovené spojení

V této úloze se nám bohužel vloudila chyba do zadání úlohy, kde mělo být uvedeno, že jeskyně, které mezi sebou propojujeme, si očíslujeme od 1 do n. Proto se řada z vás spokojila s triviálním řešením pomocí třídění nebo pomocí hešování. Chyba je ovšem na naší straně, takže jsem taková řešení jen mírně bodově odlišil od těch, která si chybějící předpoklad domyslela a byla zcela správně.

Optimálním řešením je dvojité užití přihrádkového třídění. Nejprve si každou dvojici jeskyň uspořádáme vzestupně. Nyní roztřídíme dvojice do přihrádek podle hodnoty druhé jeskyně. V praxi to můžeme reprezentovat třeba polem spojových seznamů délky n, kde ke každé hodnotě druhé jeskyně budeme mít spojový seznam hodnot první jeskyně. K druhému třídění využijeme výsledek prvního. Budeme postupně procházet každý spojový seznam od nejmenší hodnoty druhé jeskyně k největší a zatřiďovat do jiných přihrádek tentokrát podle první jeskyně. Všimněme si, že pokud k jedné první jeskyni existují dvě druhé, tímto průchodem jako první narazíme na tu s menším číslem. Z toho vyplývá, že čísla ve výsledných přihrádkách budou setříděná od nejmenšího k největšímu.

Tím jsme dostali všechny dvojice setříděné primárně podle první a sekundárně podle druhé jeskyně. Takto setříděná data už snadno projdeme a vyhneme se při vypisování duplikátům například tak, že si pamatujeme, co jsme naposledy vypsali.

Zacházeli jsme s $\mathcal{O}(n)$ přihrádkami a s $\mathcal{O}(m)$ jeskyněmi, přitom jsme na každou jeskyni spotřebovali konstantní čas při jejím zapracování do některého seznamu, výsledná časová i paměťová složitost tohoto řešení je tedy $\mathcal{O}(n+m)$.

Program (C++):
`http://ksp.mff.cuni.cz/viz/26-4-3.cpp`

Mark Karpilovskij

26-4-4 Skládání mapy

Častou chybou u úlohy bylo to, že jste předpokládali celočíselnost souřadnic a pokoušeli jste se obdélníky s mapou napasovat do nějaké tabulky, o celočíselnosti jsme však nic neslíbili. Ta by se sice dala zachránit nějakou kompresí souřadnic (seřadili bychom si neceločíselné souřadnice za sebe a očíslovali), ale větší problém byl v tom, že taková tabulka by mohla být obrovská – představte si dva malé kusy mapy vzdálené od sebe milion políček. Inicializace takové tabulky by nám zabrala neúměrně mnoho času, a proto bylo nutné vydat se jinudy.

Nejdříve se zkusíme zamyslet nad tím, jak by se úloha řešila jen v jednom rozměru, tedy kdybychom namísto obdélníků měli jen úsečky na přímce. Takový případ je přesně stvořený pro zjednodušený intervalový strom.[10]

Intervalový strom nám ve zkratce umožňuje provádět dotazy nad intervaly, tedy s kterými intervaly se protíná hledaný interval. My ale hledáme jen jeden bod a navíc nás zajímá jen počet obdélníků mapy, se kterými se protneme, proto nám stačí použít jednodušší verzi intervalového stromu.

Z původního intervalového stromu si vypůjčíme jen vyhledávání podle konců pokrytých intervalů ve vnitřních vrcholech, a protože naše dotazy půjdou vždy až do samotných listů intervalového stromu, stačí nám mít počty pokrytých intervalů jen v těchto listech. Později u dvourozměrné varianty budeme muset do vnitřních vrcholů přidat nějaké další hodnoty, ale u jednorozměrné verze bez nutnosti měnit intervaly za běhu si vystačíme s touto lehčí verzí.

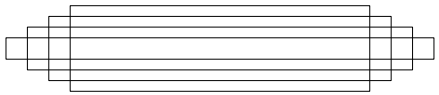
Takový strom si lehce postavíme v čase $\mathcal{O}(N \log N)$ (kde N je počet úseček) tak, že si nejdříve setřídíme souřadnice začátků a konců úseček, budeme je postupně procházet (za začátek přičteme jedničku k aktuálnímu počtu pokrytých úseček, za konec zase odečteme) a ukládat do listů úplného binárního stromu. Při obcházení stromu navíc ještě do vnitřních vrcholů uložíme konce intervalů (pro vyhledávání) a jsme hotovi. Vyhledání pak lehce zvládneme v čase $\mathcal{O}(\log N)$ na dotaz.

Dva rozměry

Jak postup výše zobecnit pro dva rozměry? Ano, použijeme dvourozměrné intervalové stromy. Nejdřív si vyrobíme intervalový strom pro jeden rozměr (jako kdybychom kusy mapy promítli jako úsečky třeba na osu x). Tím si rovinu rozdělíme na jednotlivé (v tomto případě svislé) pásy, kde máme jen spodní a vrchní okraje obdélníků mapy. Ještě však potřebujeme vyhledat správnou oblast uvnitř těchto pásů.

Proto si v každém listu prvního intervalového stromu ubytujeme další intervalový strom, který bude mít na starost pouze tento pás (každý pás si totiž můžeme zase představit jako úsečky v ose y odpovídající jednotlivým kusům mapy). Vyhledání je pak dvoustupňové: nejdříve v prvním stromě najdeme správný pás a v tomto pásu pomocí odpovídajícího stromu najdeme přesné místo, kde je umístěn bod, na který se ptáme. Takový dotaz zvládneme v $\mathcal{O}(\log N)$.

S dvourozměrnými intervalovými stromy se ale musí opatrně – spousta zdrojů sice uvádí, že se dají vybudovat v čase $\mathcal{O}(N \log^2 N)$, ale to jen, pokud se na ně uplatní nějaký trik. Ti z vás, kteří použili ve svých řešeních pojem 2D-intervalového stromu jako všemocný blackbox bez toho, aby se nad ním a jeho použitím a problémy zamysleli, nějaký ten bod bohužel ztratili.



Pokud by totiž obdélníky vypadaly třeba jako na obrázku výše, budeme mít řádově N intervalů v prvním (x-ovém) stromě a každý z y-ových stromů bude také držet řádově N intervalů. Pokud si budeme každý z menších stromů ukládat samostatně, dostaneme se na paměť $\mathcal{O}(N^2)$, a k jejich konstrukci tedy i na stejnou časovou složitost.

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

Možná už začínáte být nervózní, už je skoro konec seriálu a ještě nebylo ani slovo o tradičním uzávorkování. Nebojte se, zde přichází:

> **Úkol 4** [5b]: Vaším úkolem bude zkontrolovat správnost uzávorkování skládajícího se z levých a pravých závorek. Správné uzávorkování je takové, kde jsou závorky správně spárovány a páry se nekříží.
>
> V řeči grafomatu budou závorky zakódované hodnotami ve vrcholech 2-regulárního grafu (kružnice) na N vrcholech: v prvním vrcholu bude zapsána první závorka, ve druhém druhá, a tak dále. Poslední vrchol pak bude navíc hranou spojen s prvním, aby byla kružnice uzavřená.
>
> Závorky budou zapsány pomocí proměnné *zavorka* a to tak, že hodnota 1 bude odpovídat levé (otevírací) závorce a hodnota -1 pravé (uzavírací) závorce. Navíc bude první vrchol označen pomocí $prvni = 1$, aby šel jednoduše poznat.
>
> Na konci běhu by ve všech vrcholech měla být nastavená proměnná *vystup* na hodnotu 1, pokud bylo uzávorkování korektní, nebo na hodnotu 0, pokud nebylo.

[3] `http://mo.mff.cuni.cz/p/56/zadani-1.html#P-I-4`

[4] `http://en.wikipedia.org/wiki/Conway%27s_Game_of_Life`

Pár slov závěrem

S grafomaty jste se již mohli potkat při studování starých ročníků Matematické olympiády kategorie P. Naše grafomaty jsou velmi podobné (ale ne identické – třeba se liší v podmínkách zastavování), takže se pro inspiraci můžete podívat i na studijní texty a úlohy 56. ročníku olympiády.[3]

Možná vám grafomaty přijdou jako zajímavý teoretický model, který nemá s praxí pranic společného. Opak je však pravdou. Speciální verzí grafomatů jsou třeba takzvané *buněčné automaty* a nejznámější z nich je asi Conwayova hra Život.[4] Ta se svým fungováním blíží svému biologickému předobrazu – buňky v těle složitějších organismů jsou také často jedna jako druhá (alespoň v rámci stejné tkáně) a každá z nich reaguje jen na to, co „vidí“ okolo sebe.

Mimo to grafomat můžeme považovat za jeden z dosti realistických modelů paralelních počítačů. Z fyzikálních zákonů totiž plyne, že vzdálené procesory spolu nemohou komunikovat okamžitě (kvůli konečné rychlosti světla) a že každý procesor může komunikovat pouze s velmi omezeným počtem sousedů (kvůli omezené dimenzi prostoru). Obě omezení grafomat poměrně věrně zachycuje.

Jirka Setnička

Řetězec je v podstatě jakákoli posloupnost symbolů zapsaná za sebou a s nimi budeme v této kapitole pracovat. Každého napadne „vyhledávání v textu“ nebo „hledání jmen v telefonním seznamu“, ale řetězce najdeme i na nižších úrovních informatiky. Například celé číslo zakódované v binární soustavě, které dostaneme na vstupu programu, je také jen řetězec nul a jedniček.

Jiný příklad použití řetězců (a jejich algoritmů) najdeme v biologii. DNA není o mnoho více, než chytré uložení posloupnosti čtyř znaků/nukleových bazí – a chceme-li hledat vzory anebo konkrétní podposloupnosti, bude se nám hodit znalost základních algoritmů pro práci s řetězci.

Nemáme bohužel šanci vysvětlit *všechny* algoritmy s řetězci, protože je příliš mnoho možných věcí, co s řetězci dělat. Převáděním řetězců na čísla (hešováním) jsme se věnovali v jiné kuchařce, v této se budeme soustředit na algoritmy, které se objevují spíše v práci s textem.

Kromě úvodu popíšeme dva *stavební kameny* textových algoritmů, což bude jedna datová struktura pro slovníky (trie) a jedno vyhledání v textu s předzpracováním hledaného slova (a jeho rozšíření pro více slov). S jejich znalostí se pak mnohem snáze vymýšlí řešení složitějších, reálnějších problémů.

Jak řetězce chápat

Když programátor dělá první krůčky, často moc netuší, co s těmi řetězci vlastně může a nesmí dělat. V programovacím jazyce to je jasné – něco mu jazyk dovolí a na něco nejsou prostředky. Ale jak to je na úrovni ryze teoretické?

Jak jsme si řekli na začátku, řetězec bude posloupnost nějakých symbolů, kterým říkáme *znaky*. Tyto znaky jsou z nějaké množiny, které říkáme *abeceda*. Abeceda může být jen $\{0,1\}$ pro čísla v binárním zápisu, klasické {`A-Z`,`a-z`} pro anglickou abecedu anebo plný rozsah univerzální znakové sady Unicode, která má až 2^{31} znaků. Nezapomínejme, že nejenom písmena a číslice, ale i mezery a interpunkce jsou znaky!

Vidíme, že zanedbat velikost abecedy při odhadu složitosti by bylo příliš troufalé, a tak budeme velikost abecedy označovat $|\Sigma|$. Abeceda sama se v textech o řetězcích často značí řeckým Σ.

O znacích samotných předpokládáme, že jsou dostatečně malé, abychom s nimi mohli pracovat v konstantním čase, podobně jako s celými čísly v ostatních kapitolách.

Nyní hlavní otázka – máme chápat řetězec jako pole znaků, nebo jako spojový seznam? Šalamounská odpověď: můžeme s ním pracovat tak i tak. Když budeme potřebovat převést řetězec na spojový seznam (protože se nám hodí rychlé přepojování řetězců), tak si jej převedeme. Tento převod nás samozřejmě bude stát čas lineárně závislý na *délce* řetězce. Budeme ji dále značit L; časová složitost převodu bude $\mathcal{O}(L)$.

Standardně se ale počítá s tím, že řetězec je uložen v poli někde v paměti (již od začátku algoritmu), takže ke každému znaku můžeme přistupovat v konstantním čase.

Jelikož jsme řetězce definovali jako posloupnosti, nesmíme zapomínat ani na *prázdný řetězec* ε. A když už máme řetězec, určitě máme i *podřetězec* – souvislou podposloupnost znaků jiného řetězce. Například `BAR`, `RET`, ε i `KABARET` jsou podřetězce slova (řetězce) `KABARET`; `KAT` však podřetězcem není.

Často nás budou zajímat dva zvláštní druhy podřetězců. Pokud ze slova odstraníme nějaký souvislý úsek na konci, vznikne podřetězec, kterému říkáme *prefix* (česky předpona), a pokud odstraníme nějaký souvislý úsek ze začátku, dostaneme *suffix* neboli příponu. `RET` je suffix slova `KABARET`, `KABA` je zase jeho prefixem.

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

Pro některá použití řetězců je důležité, abychom je mohli porovnávat – když máme řetězce R a S, chceme rozhodnout, který je menší, a který je větší. Jaké přesně toto uspořádání bude, závisí na naší aplikaci, ale mnohdy se používá tzv. *lexikografické uspořádání*.

Pro lexikografické uspořádání potřebujeme nejprve zadané lineární uspořádání na znacích. Tím se myslí takové, kde jsou všechny prvky navzájem porovnatelné a v podstatě to znamená, že jsou uspořádány v jedné řadě za sebou (kromě binárního `0` < `1` se často používá „telefonní“ `A` = `a` < `B` = `b` < ... < `Z` = `z`, které je ovšem lineární až na velikost znaků).

Když máme zadané uspořádání na znacích, na všechny řetězce je rozšíříme následovně: Nejkratší je prázdný řetězec. Ostatní řetězce třídíme podle jejich prvního znaku. Jestliže se první znak shoduje, tak podle druhého znaku, atd. Pokud přitom znaky jednoho z řetězců dojdou dřív, prohlásíme tento řetězec za menší.

Platí tedy třeba $\varepsilon <$ `A` < `AUTO` < `AUTOBUS` < `AUTOGRAM` < `AUTOR` < `BAMBITKA` < `BARNABAS` < `Z`.

Adresář pomocí trie

Typický „textový“ problém je udržování množiny řetězců – můžete si představit třeba slovník. Slova ve slovníku si chceme šikovně předzpracovat, abychom pak mohli efektivně odpovídat na otázky typu: „Je slovo S obsaženo ve slovníku?“ Můžeme také po předzpracování chtít přidávat nové položky, nebo i odebírat staré.

Pokud bychom nemuseli odebírat slova, můžeme použít hešování, které je rychlé a účinné. Více o něm najdete v hešovací kuchařce.[5] Má však tu nevýhodu, že při velkém zaplnění se začne chovat pomaleji a mírně nepředvídatelně.

Ukážeme si jiné řešení, které je také asymptoticky rychlé a není ani příliš náročné na paměť. Využívá stromové struktury a říká se mu *trie* (vyslovujeme česky „tryje“ a anglicky jako část slova „retrieval“, z něhož slovo trie vzniklo). V češtině se občas používá také označení „písmenkový strom“.

Trie bude zakořeněný strom. V prvním patře se bude větvit podle prvního písmene slova, ve druhém podle dalšího, a tak dále.

Obrázek vydá za tisíc definic, pojďme se podívat, jak vypadá trie pro slova `AHOJ`, `AT`, `KSP`, `TRIE`, `TROUD`, `TYC`, `TYCKA`. Pro přehlednost písmena místo na hrany kreslíme do následujících vrcholů:

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

26-4-1 Kamínkový solitér

Mnoho řešitelů odeslalo správná řešení, to nás moc těší.

Lehčí varianta

Lehčí varianta byla, jak mnozí poznali, Euklidův algoritmus na zjišťování největšího společného dělitele (nsd). Viz kuchařka o teorii čísel.[7]

Správnou odpovědí tedy bylo, že nám na každé hromádce zbude $\mathrm{nsd}(a,b)$, kde a a b jsou počty kamínků na jedné a druhé hromádce.

Někteří řešitelé si povolili odčítat i stejně velké hromádky od sebe, to pak vede k tomu, že celkový počet zbylých kamínků bude $\mathrm{nsd}(a,b)$.

Těžší varianta

Jak se nám změní úloha pro více hromádek?

S hromádkami, které mají 0 kamínků, se vypořádáme tak, že je prostě zahodíme, ty nám nijak nemohou ovlivnit průběh hry ani celkový výsledek (protože odečtením této hromádky od jiné se stav hry nezmění).

Nyní musíme dokázat, že $\mathrm{nsd}(a,b,c) = \mathrm{nsd}(a,\mathrm{nsd}(b,c))$.

Víme, že $a>0, b>0, c>0$ (jinak bychom je neuvažovali mezi hromádky). Dále platí, že $a = g\cdot a'$, $b = g\cdot b'$, $c = g\cdot c'$. Navíc platí, že $\mathrm{nsd}(b,c) = g\cdot z'$, protože $\mathrm{nsd}(b,c)$ získáme odečítáním hromádek b a c od sebe. Vždy tu vyšší odčítáme od té nižší, a tím zůstane g zachováno. To platí proto, že $b-c = (g\cdot b') - (g\cdot c') = g\cdot(b'-c')$. Pak platí, že pokud $g = \mathrm{nsd}(a,b,c)$, pak i $\mathrm{nsd}(a, g\cdot z') = g$.

Tedy správná odpověď byla, že na každé hromádce zbude $\mathrm{nsd}(h_i)$, případně že zbude právě $\mathrm{nsd}(h_i)$ (pokud si řešitel povolil odčítat od sebe i stejně velké hromádky). Dokázané pozorování $\mathrm{nsd}(a,b,c) = \mathrm{nsd}(a,\mathrm{nsd}(b,c))$ využijeme pro náš algoritmus, který bude umět v $\mathcal{O}(1)$ poradit další tah hry. Algoritmus bude vypadat takto:

Nejprve zahodíme hromádky, které mají hodnotu 0. Nyní vezmeme 1. a 2. hromádku, odčítáme vždy od větší menší, dokud nemají stejnou hodnotu (která je, jak už víme, nsd původních hodnot těch dvou hromádek). Poté to samé provedeme s 2. a 3. (zde dostáváme nsd původních hodnot všech předchozích hromádek a té aktuální), pak se 3. a 4., ..., $(n-1)$-tou a n-tou. Tím jsme získali na n-té a $(n-1)$-té hromádce nsd všech hromádek. Nyní potřebujeme tento dělitel dostat k předchozím. Tedy provedeme odčítání hromádek zpětně (nejprve s hromádkou $n-1$ a $n-2$, pak s $n-2$ a $n-3$, ..., 1 a 2). Když skončíme, máme na všech nenulových hromádkách $\mathrm{nsd}(h_i)$ a 0 na nulových.

Vojta Sejkora

26-4-2 Výroba amuletu

Úloha se značně podobá známému problému hledání *nejdelší společné podposloupnosti*, popsanému např. na Wikipedii,[8] včetně velice přímočarého řešení. Zmiňujeme se o něm i v naší kuchařce o dynamickém programování,[9] ale algoritmus popsaný tam by se hůře upravoval pro potřeby naší úlohy.

Předpokládejme nejdříve pro jednoduchost, že ceny všech korálků jsou stejné, $c_R = c_G = c_B = 1$. Upravený amulet bude obsahovat tři druhy korálků: *nové* (které jsme vyrobili pro potřebu hesla), *recyklované* (které jsme použili z původního amuletu jako součást hesla) a *zbytečné* (které zbyly z původního amuletu, ale nejsou použity k vytvoření hesla).

My hledáme řešení s co nejméně novými korálky. Ale ježto nových a recyklovaných dohromady je vždy stejně (jako délka hesla), můžeme zrovna tak hledat řešení obsahující nejvíce recyklovaných korálků. Vzhledem k tomu, že recyklované korálky tvoří (z definice) společnou podposloupnost amuletu a hesla, můžeme použít algoritmus z odkazovaného článku pro nalezení jejich nejdelší společné podposloupnosti (NSP), a tak zjistit, které korálky chceme v optimálním řešení recyklovat.

Ukažme si to na příkladu pro amulet `RRRGGGBBB` a heslo `RGBRGB`:

```
Původní amulet            RRRG  GGBBB
Heslo                     R  GBR GB
Nejdelší spol. podposl.   R  G   GB
Vyrobeno                     BR
Nový amulet               RRRGBRGGBBB
Typ korálku               rzzrnnzrrzz
```

Z výstupu algoritmu kromě samotné podoby NSP (`RGGB`) snadno zjistíme, i na jakých pozicích v amuletu a hesle se tyto recyklované korálky budou nacházet. Jedním průchodem NSP si tedy můžeme ke každému recyklovanému korálku uložit jeho pozici v původním amuletu, a z toho už snadno dopočítáme, kam se vkládají korálky nové.

Tady je trochu problém s interpretací zadání, neboť přidáváním nových korálků se nám původní posouvají a není příliš jasné, co vlastně znamená „i-tý korálek“. V příkladu výše nejprve vypíšeme „vlož modrý korálek za čtvrtý“ a pak červený za ... čtvrtý, nebo pátý? Nejjednodušším způsobem, jak se problému elegantně zcela vyhnout, je vypisovat přidané korálky odzadu. Tak nám žádný přidaný korálek nemůže ovlivnit číslování míst, na která budeme přidávat později. V našem případě by to tedy znamenalo nejdříve vložit červený korálek za čtvrtý a pak modrý za čtvrtý (čímž se červený odsune o jednu pozici doprava).

Složitosti algoritmu dominuje hledání NSP, které zvládneme v čase a paměti $\mathcal{O}(|A|\cdot|H|)$, kde $|A|$ je délka původního amuletu a $|H|$ je délka hesla. Zbývá se vypořádat s tím, že výrobní ceny korálků nemusí být jednotkové. Ale není těžké si rozmyslet, že algoritmus NSP můžeme jednoduše upravit tak, aby místo nejdelší hledal *nejdražší* společnou podposloupnost. Prostě všude místo délek posloupností počítáme jejich ceny a při rozšiřování posloupnosti místo jedničky přičítáme cenu přidávaného korálku. Složitost tím nijak neovlivníme. Pro přesnou podobu upraveného algoritmu nahlédněte do programu.

Program (C):
`http://ksp.mff.cuni.cz/viz/26-4-2-nsp.c`

Filip Štědronský

[7] `http://ksp.mff.cuni.cz/viz/kucharky/teorie-cisel`

[8] `http://en.wikipedia.org/wiki/Longest_common_subsequence#Solution_for_two_sequences`

[9] `http://ksp.mff.cuni.cz/viz/kucharky/dynamicke-programovani`

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

1. c = poslední znak slova (znak stavu P, pro který hledáme zpětnou hranu);
2. přesuneme se do otce;
3. přesuneme se po zpětné hraně;
4. dokud neexistuje syn se znakem c nebo nejsme v kořeni, přesouváme se po zpětných hranách;
5. pokud existuje syn se znakem c, natáhneme do něj zpětnou hranu z P, jinak ji natáhneme do kořene.

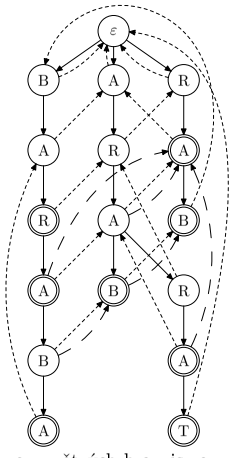
Automat je zkonstruován. Časová složitost konstrukce sestává z konstrukce trie v $\mathcal{O}(J \cdot |\Sigma|)$, resp. $\mathcal{O}(J \cdot \log|\Sigma|)$ (pokud použijeme binární strom ve vrcholech) a z předpočítání zpětných hran. Při předpočítávání uděláme nějaký konstantní počet operací pro každý vrchol (celkem tedy $\mathcal{O}(J)$) a také paralelně vyhledáváme všechny jehly z jehelníčku, jejichž vyhledání nás stojí $\mathcal{O}(J)$, resp. $\mathcal{O}(J \cdot \log|\Sigma|)$.

Tedy konstrukce trvá celkem $\mathcal{O}(J \cdot |\Sigma|)$, resp. $\mathcal{O}(J \cdot \log|\Sigma|)$, paměťová náročnost je stejná jako u trie – $\mathcal{O}(J \cdot |\Sigma|)$, resp. $\mathcal{O}(J)$, přidali jsme jen $\mathcal{O}(J)$ zpětných hran.

Projdeme tedy automatem text `BARABARARAT`. Ohlásí postupně nález slov `BAR`, `BARA`, `BARABA`, `BAR`, `BARA`, `ARARA` a `ARARAT`.

Nenalezl však všechno. Chybí mu např. `ARAB`, který začíná druhým znakem a končí pátým. Dále chybí několik výskytů `RA` a jeden `RAB`.

Když byl na pátém znaku, byl ve stavu `BARAB`, jehož suffixem je `ARAB`. Obecně na suffixy zapomínáme – narozdíl od KMP, kde suffix aktuálního stavu nikdy nebyl jehla, tady jehlou být může.



V každém stavu bychom tedy měli projít všechny suffixy a zkontrolovat je, jestli náhodou nejsou jehlami. Jak najdeme všechny suffixy? Projdeme postupně po zpětných hranách až do kořene. Má to jen jeden problém – je to pomalé.

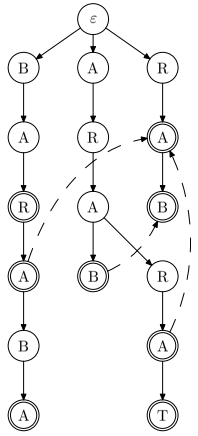
Představme si například slovník obsahující `A` a `AAAA...A` (délky $J-1$). Budeme-li jím vyhledávat v textu `AAAA...A` délky $S > J$, projdeme prakticky pro každý znak až $J-1$ zpětných hran, čímž složitost naroste až na nepoužitelných $\mathcal{O}(S \cdot J)$.

Všimněme si však, že většinou zpětných hran jsme prošli úplně zbytečně. Předpočítáme si tedy *zkratky* – z vrcholu vede zkratka do nejdelšího jeho suffixu, který je jehlou. Na obrázku jsou vyznačeny dlouze čárkovanými šipkami.

Předpočítání zpětných hran časovou složitost konstrukce automatu jistě nezhorší, neboť vyžaduje v nejhorším případě projít všechny zpětné hrany ještě jednou.

Potřebujeme-li ohlásit všechny výskyty slov včetně pozice, kde se nacházejí, jsme hotovi. Výsledná časová složitost prohledávání bude $\mathcal{O}(S+O)$, resp. $\mathcal{O}(S \cdot \log|\Sigma| + O)$, kde O je velikost výstupu – počet výskytů všech slov.

Celková časová složitost prohledávání včetně stavby automatu tedy bude $\mathcal{O}(O+S+J\cdot|\Sigma|)$, resp. $\mathcal{O}(O+(S+J)\cdot\log|\Sigma|)$.



Jak velký může být výstup? Obecně až S^2. Extrémně velký výstup je možné vygenerovat třeba slovníkem obsahujícím všechny prefixy slova `AAAA...A` délky S a senem taktéž `AAAA...A` délky S. Automat pak hlásí výskyt pro každé podslovo, kterých je řádově S^2.

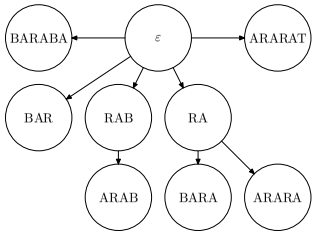
Pokud nám stačí u každého slova jen počet výskytů, nemusíme zoufat – závislost na počtu výskytů umíme odstranit.

Použijeme trik – na každé pozici započítáme pouze nejdelší jehlu, která tam končí (u každé jehly si budeme udržovat čítač). Nebudeme tedy v každém kroku poskakovat po zkratkách až do aleluja, ale maximálně jednou. Díky tomu nám z časové složitosti zmizí velikost výstupu.

V našem příkladu se senem `BARABARARAT` tedy na konci budeme mít uloženo, že `ARAB` se vyskytnul 1×, `ARARA` 1×, `ARARAT` 1×, `BAR` 2×, `BARA` 2× a `BARABA` 1×. `RA` a `RAB` nemají hlášený žádný výskyt.

Nyní si zkonstruujeme strom jenom ze zkratek a pro každý vrchol spočítáme součet celého jeho podstromu.

Tedy po přepočtu bude mít `RA` tři výskyty a `RAB` jeden výskyt; celkový počet výskytů pak bude 12.



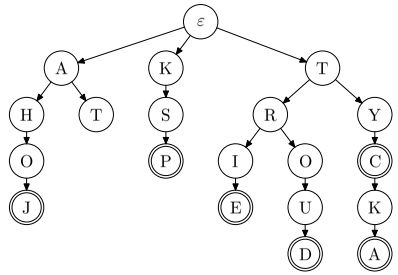
Poznámky

- Dalším krokem po KMP a Aho-Corasickové jsou konečné automaty a regulární výrazy, o kterých jsme měli seriál ve 23. ročníku.
- Není moc rozumné snažit se implementovat Aho-Corasickovou v rozumné době například při soutěži, pokud tento algoritmus nemáte opravdu pod kůží. Radši zkuste použít hešování, pokud budete něco takového potřebovat.

Cvičení

- Redukci o velikost výstupu můžeme provést i pro případ, kdy výstup nebudeme vypisovat, ale stačí nám mít jej uložený v paměti. Vymyslete vhodnou úpravu triku s čítačem.
- Zkuste si implementovat Aho-Corasickovou vlastnoručně ve svém oblíbeném jazyce, abyste si byli jisti, že doopravdy chápete všechny záludnosti tohoto algoritmu.

Martin Böhm, Jan Matějka, Martin Mareš a Petr Škoda



Všimněte si, že vrcholy v hloubce h (tedy v h-tém patře trie) odpovídají prefixům délky h zadaných slov. Například prefixy délky 2 jsou `AH`, `AT`, `KS`, `TR` a `TY`. Hrana mezi prefixy vede právě tehdy, lze-li jeden z druhého získat připsáním písmene na konec.

Jak bychom takovou trii postavili algoritmem? Přesně, jak jsme ji definovali: každé slovo ze slovníku budeme procházet znak po znaku a bude-li nějaká hrana chybět, tak ji vytvoříme a pokračujeme dále podle slova.

Z takto popsané trie bohužel nepoznáme, kde končí slovo ze slovníku a kde končí jen jeho prefix. Standardní způsoby, jak to vyřešit, jsou dva: buď si do každého vrcholu přidáme informaci o tom, je-li koncem celého slova nebo ne (jak je to naznačeno dvojitými kroužky v obrázku), anebo si rozšíříme abecedu o speciální znak, který se v ní předtím nevyskytoval – třeba `$` – a pak všem slovům přilepíme tento `$` na konec.

Budeme-li se později ptát, bylo-li slovo ve slovníku, po průchodu trií zkontrolujeme ještě, jestli z konečného vrcholu vede hrana odpovídající znaku `$`.

Ještě jsme si nerozmysleli, jak budeme v jednotlivých vrcholech trie reprezentovat hrany do delších prefixů. Abychom mohli vyhledávat skutečně lineárně, potřebovali bychom umět v konstantním čase odpovědět na otázku „má vrchol P potomka přes hranu se znakem c?“.

Abychom zajistili konstantní čas odpovědi, museli bychom mít v každém vrcholu pole indexované znaky abecedy. To ovšem znamená, že takové pole budeme muset vytvořit, a tedy alokovat $|\Sigma|$ políček v každém znaku.

To zvýší paměťovou náročnost trie (a časovou náročnost její stavby) na $\mathcal{O}(D \cdot |\Sigma|)$, kde D značí velikost vstupu, čili součet délek všech slov ve slovníku. To je naprosto přijatelné pro malé abecedy, ale už pro {`A-Z`, `a-z`} je tento faktor roven 52 a pro Unicode je taková alokace nemyslitelná.

Jak z toho ven? Můžeme oželet konstantní rychlost dotazu a použít namísto pole třeba binární vyhledávací strom nebo hešovací tabulku všech znaků, kterými aktuální prefix může pokračovat. Nebo můžeme každý znak velké abecedy zapsat pomocí několika znaků menší abecedy. Tou menší abecedou může být třeba {`0`, `1`}. Tehdy nahradíme každý znak původní abecedy $\lceil \log_2|\Sigma| \rceil$ novými (jeho zápisem ve dvojkové soustavě). Tím se časová slozitost konstrukce zlepší na $\mathcal{O}(D \cdot \log|\Sigma|)$ a časová složitost dotazu na slovo délky L zhorší na $\mathcal{O}(L \cdot \log|\Sigma|)$.

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

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

Poznámky

- Chcete-li algoritmus konstrukce trie vidět napsaný v Pascalu, podívejte se do knihy *Algoritmy a programovací techniky*.
- Triím se také říká *prefixové stromy*, což popisuje, že každý vrchol odpovídá prefixu některého slova ve slovníku.
- Kdybychom chtěli, mohli bychom pomocí trie vyhledávat v textu v lineárním čase. Můžeme přeci postavit slovník ze všech slov v daném textu, a pak procházet příslušnou trii. Má to ale pár háčků: jednak je často hledaný řetězec krátký, ale text se nevejde do paměti. Druhak, pokud bychom použili jako oddělovač mezery, mohli bychom hledat jen jednotlivá slova, a nikoli jejich konce nebo delší kusy věty.
- Asi se po poslední poznámce ptáte – existuje nějaká modifikace trie, která umí hledat libovolnou část textu? Ano, jmenuje se *suffixový strom* a jdou s ní dělat spousty krásných kousků. Říká se, že každou řetězcovou úlohu lze řešit v lineárním čase pomocí suffixových stromů. Víc se o nich dočtete třeba v knížce *Krajinou grafových algoritmů*.[6]

Cvičení

- Řekněme, že chceme slovník na vstupu setřídit v lexikografickém pořadí (definovaném v sekci „Jak řetězce chápat“). Problémem pro klasické třídící algoritmy je to, že porovnání dvou řetězců není bohužel konstantně rychlé. Vymyslete způsob, jak setřídit takový slovník rychle pomocí trie.
- *Komprese trie.* Co kdybychom chtěli odstranit přebytečné vrcholy trie, tedy ty, v nichž se slova nevětví? Rozmyslete si, jestli by něčemu vadilo místo takovýchto cest mít jen jednotlivé hrany. Zesložití se konstrukce nebo vyhledávání? Mimochodem, je celkem jasné, že takováto *komprimovaná trie* přinese jen konstantní zrychlení dotazů i prostoru, a tak na soutěžích apod. stačí použít základní variantu.

Vyhledávání v textu

Začátek situace je asi zřejmý – máme na vstupu zadán dlouhý text a krátké slovo. Slovo si můžeme nějak předzpracovat, načež projdeme co nejrychleji text a nahlásíme jeden nebo všechny výskyty slova. Zajímají nás při tom i výskyty, které se navzájem překrývají: v textu `NANANA` se slovo `NANA` vyskytuje dvakrát. Často se hovoří o „hledání jehly v kupce sena“, pročež se textu přezdívá *seno* a hledanému slovu *jehla*. Délku jehly označíme J a délku textu S.

Představme si nejdříve hledané slovo jako spojový seznam, třeba slovo `INSTINKT`:

Mohli bychom text začít procházet znak po znaku a kontrolovat, zda se text shoduje s naším slovem/spojovým seznamem. Pokud by si znaky odpovídaly, skočíme na další znak z textu a i na další znak v seznamu. Co když se ale neshodují? Pak nemůžeme jen skočit na další znak textu – co kdybychom v textu narazili na slovo `INSTINSTINKT`?

Musíme se tedy vrátit nejen na začátek spojového seznamu, ale i zpátky v textu na druhý znak, který jsme označili jako odpovídající, a zkoušet porovnávat s jehlou znovu od začátku. To už naznačuje, že takto získaný algoritmus nebude lineární, protože se musí vracet zpět v textu o délku jehly.

Sice je předchozí popis skutečně v nejhorším případě složitý $\mathcal{O}(S \cdot J)$, avšak stačí malá úprava a složitost přejde na

lineární $\mathcal{O}(S+J)$. Ve skutečnosti algoritmus nezpomalovalo vracení se – za špatnou složitost mohl fakt, že jsme se vraceli *příliš zpátky*.

Třeba v našem příkladu s textem `INSTINSTINKT` se nemusíme vracet ve spojovém seznamu na začátek, jakmile načteme `INSTINS`. Mohli jsme se vrátit jen na druhý znak, tedy do prvního `N`, a pak kontrolovat, jaký znak pokračuje dál. Když následuje `S` jako v našem případě, můžeme pokračovat dále v čtení a nevracíme se v textu. Kdyby text byl jiný, třeba `INSTINB`, vrátili bychom se po načtení `B` na začátek spojového seznamu a v textu bychom pokračovali dále bez zastavení.

Pro každý znak ve spojovém seznamu si tedy určíme políčko spojového seznamu, na které skočíme, pokud se následující znak v textu liší od toho očekávaného. Pořadové číslo tohoto políčka nám poradí tzv. *zpětná funkce* F, což bude funkce definovaná pomocí pole, kde $F[i]$ bude pořadové číslo políčka, na které se má skočit z políčka číslo i. Porovnávat pak budeme s následujícím znakem. Pokud $F[i] = 0$, znamená to, že máme začít porovnávat úplně od prvního znaku jehly.

Pokud máte rádi grafovou terminologii, můžete se na náš spojový seznam dívat jako na graf a hovořit o *zpětných hranách*.

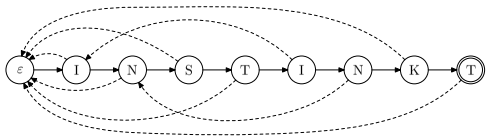
Zatím jsme ale přesně nepopsali, na které políčko přesně bude zpětná funkce ukazovat. Nechť chceme určit zpětné políčko pro druhé `N` ve slově `INSTINKT`. Pracujeme teď s prefixem `INSTIN`. Selsky řečeno, chceme najít „konec slova `INSTIN` takový, že je stejný, jako začátek slova `INSTIN`“.

Abychom náš požadavek upřesnili, zamyslíme se nad zpětným políčkem pro jiné slovo. Co kdyby jehlou bylo slovo `ABABABC` a my určovali zpětné políčko pro `ABABAB`? Kdybychom ukázali na první písmenko `B`, nebylo by to správně, protože pak bychom pro text `ABABABABC` nezahlásili výskyt jehly, což je jasná chyba. Musíme se vrátit už na `ABAB`!

Zajímá nás tedy ne libovolný suffix, který je stejný jako začátek, ale nejdelší takový konec/suffix. A ještě navíc ne jen ten nejdelší, ale nejdelší „netriviální“ – slovo `INSTIN` je samo sobě prefixem a suffixem, ale zpětná funkce pro `N` by se neměla cyklit, měla by vést zpátky.

Řekněme to tedy znova, zcela formálně: pokud bychom právě určovali hodnotu zpětné funkce pro znak číslo i, kterému odpovídá prefix P, pak její hodnota bude *délka nejdelšího vlastního suffixu* slova P, pro který ještě platí, že *je zároveň prefixem* P.

Pro slovo `INSTINKT` vypadá spojový seznam se zpětnou funkcí (zakreslenou pomocí ukazatelů) takto:



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

Poperme se nejdříve s tou první. Pro každý znak vstupního textu mohou nastat dva případy: Buď znak rozšiřuje aktuální prefix, nebo musíme použít zpětnou funkci. První případ má jasně konstantní složitost, druhý je horší, neboť zpětná funkce může být pro jeden znak volána až J-krát.

Při každém volání však klesne pořadové číslo aktuálního stavu (políčka) alespoň o jedna, zatímco kdykoliv stav prodlužujeme, roste jen o jeden znak. Proto všech zkrácení dohromady může být nejvýše tolik, kolik bylo všech prodloužení, čili kolik jsme přečetli znaků textu. Celkem je tedy počet kroků automatu lineární v délce textu, $\mathcal{O}(S)$.

Konstrukci zpětné funkce provedeme malým trikem. Všimněme si, že $F[i]$ je přesně číslo stavu, do nějž se dostaneme při spuštění našeho vyhledávacího algoritmu na řetězec, který tvoří prefix délky i z jehly bez prvního znaku.

Proč to tak je? Zpětná funkce říká, jaký je nejdelší vlastní suffix daného stavu, který je také stavem, zatímco políčko, ve kterém po i krocích skončíme, označuje nejdelší suffix textu, který je stavem. Tyto dvě věci se přeci liší jen v tom, že ta druhá připouští i nevlastní suffixy, a právě tomu zabráníme odstraněním prvního znaku.

Takže F získáme tak, že spustíme vyhledávání na část samotné jehly. Jenže k vyhledávání zase potřebujeme funkci F. Jak z toho ven? Budeme zpětnou funkci vytvářet postupně od nejkratších prefixů. Zřejmě $F[1] = 0$. Pokud již máme $F[i]$, pak výpočet $F[i+1]$ odpovídá spuštění automatu na slovo délky i a při tom budeme zpětnou funkci potřebovat jen pro stavy délky i nebo menší, pro které ji již máme hotovou.

Navíc nemusíme pro jednotlivé prefixy spouštět výpočet vždy znovu od začátku – $(i+1)$-ní prefix je přeci prodloužením i-tého prefixu o jeden znak. Stačí tedy spustit algoritmus na jehlu bez prvního znaku a sledovat, jakými stavy bude procházet – to budou přesně hodnoty zpětné funkce.

Vytvoření zpětné funkce se nám tak nakonec zredukovalo na jediné vyhledávání v textu o délce $J-1$, a proto poběží v čase $\mathcal{O}(J)$. Časová složitost celého algoritmu tedy bude $\mathcal{O}(S+J)$. Dodáme už jen, že tento algoritmus poprvé popsali pánové Knuth, Morris a Pratt a na jejich počest se mu říká KMP. Naprogramovaný bude vypadat následovně (čtení vstupu jsme si odpustili):

```
var
  Slovo: array[1..J] of char;        { jehla }
  Text: array[1..S] of char;          { seno }
  F: array[1..J] of integer;    { zpětná fce }
  I, J: integer;          { pomocné proměnné }

  function Krok(I: integer; C: char): integer;
  begin
    if (I < J) and (Slovo[I+1] = C) then
      Krok := I + 1
    else if I > 0 then
      Krok := Krok(F[I], C)
    else
      Krok := 0;
  end;

begin { konstrukce zpětné funkce }
  F[1] := 0;
  for I := 2 to J do
    F[I] := Krok(F[I-1], Slovo[I]);
  { procházení textu }
  J := 0;
  for I := 1 to S do begin
    J := Krok(J, Text[I]);
    if J = J then
      writeln(I);
  end;
end.
```


Poznámky

- Pro anglický nebo český text je použití takto sofistikovaného algoritmu skoro škoda, protože v obou jazycích se stává jen málokdy, že bychom měli několik slov spojených dohromady. Prakticky bude stačit i na začátku zmíněný naivní algoritmus. Na soutěžích a olympiádách ale pište raději algoritmus KMP.
- Hešování lze použít i na vyhledávání řetězce v textu. Obzvláště vhodné jsou na to *rolling hash functions* („okénkové hešovací funkce“), které umí v konstantním čase přepočítat heš, ubereme-li nějaký znak na začátku a přidáme-li jiný na konci – jako kdybychom se dívali na text skrz posouvající se okénko.

Cvičení

- Rozmyslete si, že když vyhledáváme více slov, ne jen jedno, a algoritmus musí vypsat všechny výskyty na výstup, můžeme se dobrat vyšší než lineární složitosti v závislosti na vstupu. Na čem potom taková časová složitost také záleží?
- Vymyslete nějakou vhodnou okénkovou hešovací funkci pro vyhledávání jedné jehly.

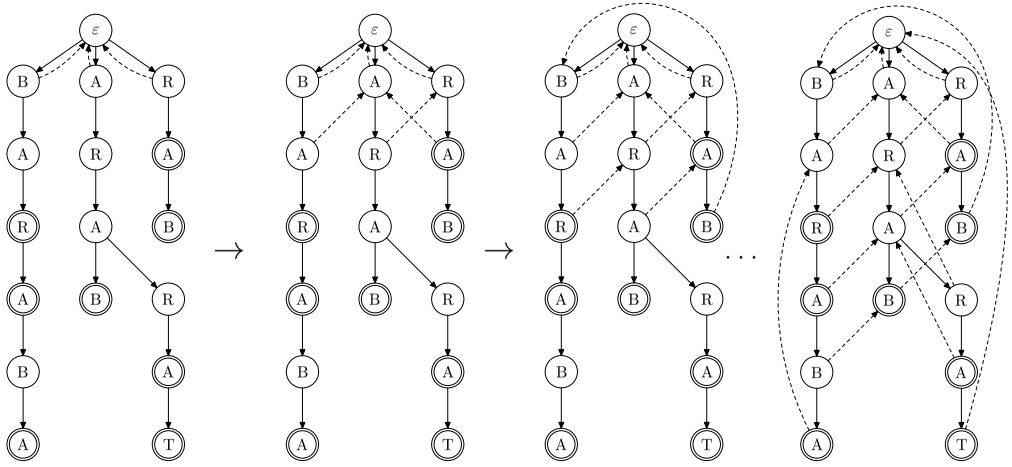
Vyhledávání jehelníčku

Co kdybychom neměli jen jednu jehlu/hledané slovo, ale celý jehelníček, čili seznam hledaných slov? I to lze řešit podobnou metodou, jakou jsme hledali jedno slovo. Tento algoritmus se nazývá po tvůrcích *algoritmus Aho-Corasickové* a spočívá v tom, že jednoduchý spojový seznam nahradíme trií a do trie opět přidáme zpětné hrany.

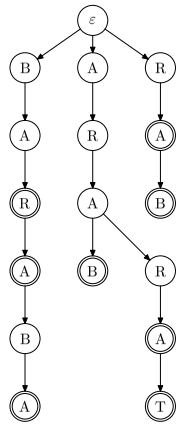
Budeme postupovat podobně jako u KMP. Nejprve naskládáme jehelníček do trie. Pro příklady v této kuchařce použijeme jehelníček `ARAB`, `ARARA`, `ARARAT`, `BAR`, `BARA`, `BARABA`, `RA` a `RAB`.

Dalším krokem v KMP bylo sestrojení zpětných hran. Nejprve jsme sestrojili zpětnou hranu pro první znak slova, pak pro druhý atd. Ve trii to bude o něco složitější.

Na první pohled se může zdát, že bychom mohli automat sestrojit tak, že bychom vyrobili KMP pro první slovo, pak KMP pro druhé slovo s využitím struktury prvního atd., ale to má háček.



Zpětné hrany totiž nemusí vést do předka. Například pro slovo `BARAB` povede zpětná hrana do slova `ARAB`, z toho do slova `RAB` a z toho do `B`. Kdybychom ale zkonstruovali automat výše popsaným způsobem (a začali slovem `BARAB`), nebude existovat v trii ani `ARAB`, ani `RAB`, takže bychom vedli zpětnou hranu chybně do `B`.



Můžeme se ale opřít o stejný trik, jako při konstrukci KMP. Budeme opět vyhledávat nejdelší vlastní suffix. Kam dojde výpočet po jeho vyhledání, tam povede zpětná hrana.

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

Rozdělíme si trii na *vrstvy* – první znaky slov budou první vrstva, druhé znaky budou tvořit druhou vrstvu atd., až i-té znaky slov budou tvořit i-tou vrstvu.

Zpětná hrana jistě povede do kratšího slova. Z i-té vrstvy tedy povede do vrstvy s nižším pořadovým číslem. Pokud tedy budeme zpětné hrany konstruovat po vrstvách, dojdeme kýženého výsledku.

Ještě zbývá otázka, jak konstruovat zpětné hrany efektivně, když je musíme vyrábět po vrstvách. Mohli bychom prostě vzít slovo, pro které hledáme zpětnou hranu, utrhnout mu první znak a vyhledat. Jenže to budeme dělat spoustu práce zbytečně.

Například pro slovo `BARABA` bychom mohli vyhledávat `ARABA` v již zkonstruované části automatu, ale proč to dělat celé, když jsme při konstrukci předchozí vrstvy vyhledávali `ARAB` při konstrukci zpětné hrany pro `BARAB`?