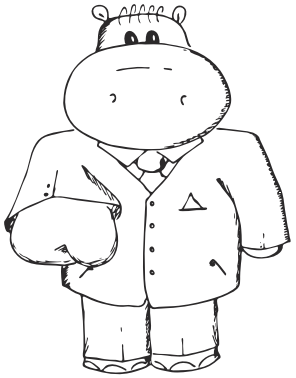
Milé řešitelky a milí řešitelé!



Vítejte u dalšího napínavého Jacobova dobrodružství na neznámé planetě, okořeněného spoustou propečených úložek. Pokud bažíte po vědomostech, můžete se podívat do kuchařky o intervalových stromech nebo si užít manuální práci při dlaždičkování zdi v naší seriálové zoo výpočetních modelů. A nedočkavci se mohou rovnou podívat, co zajímavého Jacob nalezl po pokoření všech úložek ...

Chtěli bychom vyzdvihnout, že každému řešiteli, který v tomto ročníku z každé série dostane alespoň 5 bodů, darujeme KSP propisku, blok a tužku. Dále se na vědomost dává, že **každému, kdo v této sérii získá alespoň dva body z každé úlohy, pošleme čokoládu**.

Také připomínáme, že za úspěšné řešení KSP je možno být přijat na MFF UK bez přijímacích zkoušek. Úspěšným řešitelem se stává ten, kdo získá za celý ročník alespoň 50 % z maxima bodů, tedy alespoň 150 bodů. Pokud se ale hlásíte na MFF letos a chcete prominutí přijímaček za tento ročník, máte čtvrtou sérií poslední možnost na zisk bodů. V takovém případě se nám ozvěte emailem a my se pokusíme vaše řešení opravit přednostně před ostatními.

Termín odevzdání čtvrté série je stanoven na **pondělí 31. března 2014 v 8:00 SELČ**. CodExová úloha má termín o den posunutý, opravuje ji totiž automat – odevzdejte ji do 1. dubna, 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.*

Čtvrtá série dvacátého šestého ročníku KSP

Jacob pomalu popadal dech. Právě se mu podařilo uniknout před proudem lávy valící se z rozběsněné sopky. Dokonce našel malý převis, který ho uchránil před deštěm kamení a žhavého sopečného popela.

Jeho myšlenkami letěly vzpomínky na události, které zažil od okamžiku, kdy ho ztroskotání vesmírné lodi UFC Freya *uvěznilo na této planetě. První opatrné zkoumání pralesa. Setkání s mimozemšťany. Polámané nohy. Dlouhé léčení. Tajuplná Ada. Podzemní bratrstvo. Darovaný meč. Probuzený vulkán ...*

Podařilo se aspoň části členů Bratrstva utéci do bezpečí, nebo všechny v jejich podzemním komplexu zaplavila láva? To se Jacob ještě nějaký čas nedozví – teď mu totiž nezbývá než vyčkat v úkrytu, dokud se sopka neuklidní. Aby zahnal nervozitu, zkouší hrát nejrůznější hry pro jednoho hráče neboli solitéry. Třeba s kamínky, těch je teď všude plno.

26-4-1 Kamínkový solitér — 10 bodů

Jacob hraje následující hru. Nejprve vytvoří n hromádek kamínků. Pak odebírá kamínky podle následujícího pravidla: Vždy si vybere dvojici různě velkých hromádek a z té větší odebere tolik kamínků, kolik jich je na té menší. Cílem hry je zbavit se co nejvíce kamínků.

Vymyslete algoritmus, který pro tuto hru najde optimální strategii. Na vstupu dostane počet hromádek n a počáteční počty kamínků $h_1, \ldots, h_n$. Výstupem má být posloupnost tahů typu „od h_i odečti h_j“ taková, že po provedení těchto tahů bude součet $h_1 + \ldots + h_n$ nejmenší možný.

⇧ **Lehčí varianta (za 4 body):** Rozmyslete si, jak úloha dopadne pro dvě hromádky. Své tvrzení dokažte.

Jacob se s trhnutím probudil. Co to bylo? Venku byla tma jako v pytli, hvězdy pohltil všudypřítomný sopečný prach. Opodál cosi šramotilo. „Psst, Jacobe, to jsi ty?“ špitl hlas nápadně podobný Adinu. Po chvilce oboustranného ujišťování se ze tmy vynořila Ada spolu s dalšími čtyřmi mimozemšťany, kteří s Jacobem zkoumali kráter sopky. Bylo to včera, ale zdálo se to jako věčnost ...

Přeci jen se jim podařilo uniknout potokům prskající lávy a schovat se ve stejných skalách, které poskytly azyl Jacobovi. Jacob si pomyslel, že jsou jistě celí žhaví zjistit, co se událo v okolí. Nejspíš i doslova. Každopádně jim nezbývá než zůstat v úkrytu, dokud se popel nerozptýlí. Zatím není vidět na krok a kdo ví, jak se erupcí změnila krajina.

Sedli si tedy společně na teplou zemi. Ze začátku tiše, ale po chvíli jeden z tvorů vytáhl jakési zvíře podobné bažantovi, které našel pod vrstvou rozpáleného popela. Zjevně dobře upečené. Po dobrém jídle nervozita opadla a Jacob využil příležitosti a zeptal se na pár věcí, které mu už delší dobu vrtaly hlavou.

Například proč mají všichni členové Bratrstva na krku podivný amulet – náhrdelník s řadou barevných korálků. Každý s jinou kombinací barev, ale přeci jen bylo možné vysledovat určité podobnosti. Ada usoudila, že teď už před Jacobem není potřeba nic tajit. Prozradila mu, že amulet slouží jako poznávací znamení členů Bratrstva a obsahuje heslo, které se čas od času mění.

26-4-2 Výroba amuletu — 10 bodů

Amulet definujeme jako posloupnost červených, zelených a modrých korálků. Heslo je také nějaká posloupnost korálků. Amulet obsahuje heslo, pokud lze z amuletu vypustit některé korálky tak, aby zbylo právě heslo. Matematik by tedy řekl, že heslo tvoří vybranou podposloupnost amuletu.

Ada zná nové heslo a chce svůj amulet upravit tak, aby toto heslo obsahoval. Upravovat ho může vkládáním korálků na libovolné místo. Ovšem výroba jednoho korálku trvá nějaký čas závislý na jeho barvě, tak by chtěla vymyslet, kam vložit který korálek, aby tím celkově strávila co nejméně času.

Vymyslete algoritmus, který jí v tom pomůže. Na vstupu dostane dva řetězce písmen `R`, `G` a `B`: *amulet* a *heslo*. Mimo to ještě dostane celá kladná čísla c_R, c_G a c_B udávající, kolik času trvá vyrobit korálek které barvy.

Výstupem algoritmu má být posloupnost operací „za i-tý korálek vlož korálek barvy b“, která zabere nejkratší možný čas.

Noc pokračuje. Skupinka se snaží usnout, ale ve stísněném prostoru pod převisem to jde jen obtížně. Polštáře tu nejsou, tak musel Jacob vzít zavděk jakýmsi kamenem. Nepříjemně tlačil do ucha a navíc se z něj ozývalo podivné zvonění, jako by v hlubinách planety nějaký skřítek mlátil kladivem do skály.

Zvonění je ale podivně pravidelné. Skoro jako by si ti skřítci posílali nějaké zprávy. Jacob místo počítání oveček přemýšlí, jak by takový přenos zpráv mohl fungovat. Snaží se vymýšlet různé způsoby a zkoušet pomocí nich zvonění dešifrovat. Evidentně to k ničemu nebude, ale aspoň svou mysl unaví a konečně usne.

„J... S... M... E... Z... A... S... Y... P... “ Cože??!!! Jacob ihned vzbudil ostatní a společně naslouchali skalám. Text byl poněkud zmatený, postupně však pochopili, že se jedná o víc překrývajících se zpráv. Posílají je skupinky členů Bratrstva uvězněné na různých místech v podzemí.

Do jeskynního systému nejspíš nenatekla žádná láva, ale výbuch na mnoha místech zavalil chodby. Ada hned začala do vrstvy popela zapisovat, které části podzemí zůstaly propojené, ale situaci znepřehledňovalo, že se zprávy o spojení jeskyní často opakovaly.

26-4-3 Obnovené spojení — 10 bodů

Mějme n jeskyní a m neuspořádaných dvojic $\{x_i, y_i\}$, které popisují, že jeskyně x_i je propojena s jeskyní y_i. Navrhněte co nejefektivnější algoritmus, který z tohoto seznamu odstraní opakující se dvojice.

Výstupem je tedy seznam navzájem různých dvojic, které se alespoň jednou vyskytly ve vstupním seznamu. Na pořadí dvojic nezáleží.

Ada dokreslila plán podzemí a ve tváři jí zazářila radost. Právě zjistila, že navzdory všem závalům stále existuje cesta, jak se do všech obydlených jeskyní dostat. Značně složitá, ale je tu.

Vzduch, který se mezitím trochu pročistil, ovšem odhalil, že okolní skaliska jsou zasypaná hromadami sopečného popela, tufu a kamení. Dříve důvěrně známé kopce se najednou proměnily v nepřehlednou krajinu plnou skrytých nebezpečí.

Skupina se rozdělila a každý dostal za úkol důkladně, ale velmi opatrně prozkoumat část okolí a pokusit se nakreslit mapu. Když se vrátili, zjistili, že mapy se poněkud překrývají. Některá místa nejsou zmapována vůbec, zatímco jiná velmi důkladně. Jak se v tom vyznat?

26-4-4 Skládání mapy — 12 bodů

V rovině je položeno několik kusů mapy. Kusy mají tvar obdélníků se stranami rovnoběžnými s osami souřadnic.

Naším úkolem je vytvořit datovou strukturu, která bude umět rychle odpovídat na dotazy typu „v kolika obdélnících leží zadaný bod?“

Důležitá je přitom jak časová složitost dotazů, tak čas potřebný na vybudování struktury.

Konečně se podařilo poskládat jednotlivé mapy do jednoho jakž takž použitelného celku a naplánovat záchrannou akci. Než se ale podaří zpustošené podzemí vrátit do obyvatelného stavu, bude potřeba vybudovat pro všechny provizorní tábor v horách.

Tiše přemýšleli, kde ve skalách vzít kousek rovné plochy. Naštěstí sopečné tufy a popel jsou lehké, takže menší nerovnosti půjde snadno srovnat. I tak by ale bylo milé si co nejvíc práce ušetřit. Sesedli se okolo mapy a uvažovali nad vhodným místem.

26-4-5 Místo pro tábor — 14 bodů

Je dána výšková mapa krajiny. Terén je rozdělen na $R \times S$ políček a pro každé z nich známe jeho nadmořskou výšku v mimozemských pídích.

Na nějakém místě chceme postavit tábor. To obnáší vybrat obdélníkovou část krajiny o rozměrech $r \times s$ políček a srovnat ji do roviny. Tedy přesunout mezi těmito políčky zeminu tak, aby všechna políčka byla stejně vysoko. Jednotky si zvolme tak, že zvýšení políčka o jednu mimozemskou píď vyžaduje přivezení jedné mimozemské kárky zeminy.

Vaším úkolem je pro zadanou výškovou mapu a velikost tábora najít takové místo pro tábor, abychom museli přesunout co nejméně zeminy.

Zemina je neomezeně dělitelná, ale lze ji pouze přesouvat. Není možné ani vytvořit zeminu z ničeho, ani ji zničit.

Lehčí varianta (za 10 bodů): Vyřešte pro jednorozměrnou krajinu ($S = s = 1$).

Tábor utěšeně rostl. Proudili do něj stále noví členové Bratrstva, vysvobození z čím dál vzdálenějších částí jeskynního systému. Na krajinu se snášela další noc, mnohem optimističtější než ty předchozí.

Středu tábora vévodilo veliké ohniště, u kterého právě odpočíval Ubu spolu s několika staršími mimozemšťany. Jacob si k nim přisedl. Chtěl totiž využít klidné chvilky a dozvědět se něco o tom, co je Bratrstvo zač a proč se tolik snaží svou existenci utajit.

A důvody k tajnostem skutečně existovaly: Bratrstvo organizovalo odboj proti místnímu králi. Členové královské dynastie byli sice považováni za potomky bohů (a dokonce prý vypadali trochu jinak než jejich poddaní), ale vládli velmi nevybíravě a nebývale krutě.

Spiklenci už dlouho připravovali plán na svržení panovníka. Podezírali ale krále, že o jejich úmyslech ví a že se celého Bratrstva pokusil zbavit výbuchem sopky vyvolaným magií. Jacob se tvářil značně nedůvěřivě – na kouzla nevěřil a ještě méně pravděpodobné bylo, že by kdokoliv na této planetě disponoval potřebnou technikou.

Ale dost už pochybností, dnes je den mnoha šťastných shledání a takový si zaslouží oslavu. Jacoba napadlo, že by ostatní mohl naučit nějakou pozemskou hru. Všiml si, že sopečný tuf je natolik měkký a lehký, že z něj jde nožem vyřezávat něco jako sněhové koule. Sice tolik nestudí, vlastně vůbec, ale házet jdou úplně stejně. Hej! Kryj se! Pal!

26-4-6 Sněhová bitva — 11 bodů

V rovině stojí n bojovníků se sněhovými koulemi v rukou. Za chvíli začne velká řež. Pokud bojovník A hodí kouli po bojovníkovi B, může trefit kohokoliv, kdo se nachází na polopřímce AB.

Na vstupu dostanete polohy bojovníků v rovině. Vaším úkolem je vytvořit datovou strukturu, pomocí které budete umět efektivně odpovídat na dotazy „Pokud A míří na B, může zasáhnout někoho dalšího?“. Jako odpověď stačí ANO nebo NE, není potřeba hledat, koho zasáhnete.

Do všeobecného veselí se vkrádaly stíny nedůvěry. Jak se mohl král o existenci Bratrstva dozvědět? Není mezi nimi nějaký špión, nebo dokonce víc takových? Královská tajná policie je přeci svými schopnostmi po celé říši proslulá.

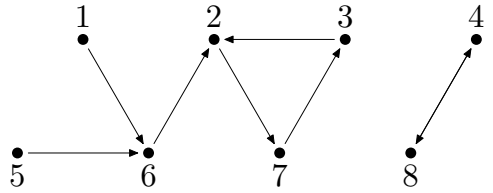
Jacoba napadlo, že to mohl být důvod, proč se k němu jeho někdejší léčitelé chovali tak uzavřeně a odmítali mu odpovídat na jeho otázky. Konec konců, královská rodina přeci má vypadat jinak než ostatní obyvatelé planety, tak není divu, že byl Jacob krajně podezřelý. O to víc si vážil důvěry Bratrstva.

Teď s Ubuem probírali různé hypotézy, jak by si mohli králové zvědové předávat informace.

26-4-7 Královští špioni — 9 bodů

Král má n špionů. Každý špion má pevně určeno, kterému jednomu špionovi předává všechny informace, jež zjistí.

Špionská síť s osmi špiony může například vypadat následovně:



Pokud špion dostane zprávu, kterou už zná, neposílá ji dál. Šíření každé zprávy se tedy po konečném počtu kroků zastaví.

Váš algoritmus dostane zadanou síť špionů. Jeho úkolem je pro každého špiona spočítat, jak dlouho bude v síti putovat zpráva, kterou tento špion vyšle.

V síti na obrázku jednotlivé zprávy urazí postupně v pořadí dle čísla vysílajícího špiona $5, 3, 3, 2, 5, 4, 3$ a 2 kroků.

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.

Jakmile v táboře přestala být potřeba každá pomocná ruka, Jacob dostal chuť projít se po okolí. Chtěl si zblízka prohlédnout ztuhlé potoky lávy, na kterých se utvořily zajímavé obrazce.

Zvolna kráčel mezi skalami a sledoval pustou krajinu. Připomněla mu povrch Měsíce. Posmutněl, když si uvědomil, že tam už se nejspíš nikdy nepodívá.

Najednou mu nohy uvázly v sopečném popelu. Když se je pokusil vyprostit, jenom se propadl o něco hlouběji. Zjevně narazil na jámu plnou popela. Marně se pokoušel rukama zachytit okraje. Sjížděl čím dál rychleji. „Už zase!“ blesklo mu hlavou.

Proletěl jakousi šikmou chodbou a přistál na podlaze nevelké jeskyně. Všude se válely podivné kovové krabice. Značně omšelé a propojené zašlými zkroucenými kabely. Na nejbližší z nich zahlédl kovový štítek s nápisem.

Stálo na něm: „Made in China.“ Uhhh...

Pokračování příště...

O Jacobových příhodách na vzdálené planetě vyprávěl

Martin „Medvěd“ Mareš

26-4-8 Dlaždičky — 16 bodů

V naší zoo výpočetních modelů jsme zatím potkávali volně pasoucí se exempláře. Dnes uvidíme první zdi. Není to ovšem proto, že by náš model potřeboval chránit před větry dešti, nýbrž proto, že tyto zdi bude náš program obkládat dlaždicemi.

Pojďme si nejprve říct, co je to taková dlaždice. *Dlaždice* představuje čtverec jednotkové velikosti, který má každou hranu obarvenou nějakou barvou. Obarvení jednotlivých hran může a nemusí být stejné, ale každá hrana musí být obarvena právě jednou barvou. Dohodněme se také, že barvy budeme označovat nějakými symboly, typicky čísly a písmeny.

Často budeme pro názornost dlaždice zobrazovat opravdu jako čtverce rozdělené na čtyři části, ale formálně dlaždici zavedeme jako uspořádanou čtveřici (ℓ, h, p, d), kde jednotlivé symboly ℓ, h, p, d označují po řadě obarvení levé, horní, pravé a dolní hrany dlaždice.

Z takových dlaždic můžeme skládat *dláždění*. Prostoru, který chceme dlaždičkovat, říkejme *zeď*. Ta je obdélníková a má rozměry $r \times s$ (přičemž jednotkou bude délka hrany dlaždice). Okraje zdi jsou rozděleny na úseky o jednotkové délce, a každý z těchto úseků má podobně jako hrany dlaždic nějaké obarvení.

Jako dláždění označujeme pokrytí zdi dlaždicemi, pokud toto pokrytí splňuje několik podmínek. V první řadě požadujeme, aby v každém z $r \times s$ čtverců byla umístěna právě jedna dlaždice. Dále každé dvě sousední dlaždice musí mít ty hrany, kterými se dotýkají, obarvené stejnou barvou. Požadavek na stejnou barvu máme i na okrajové dlaždice, tedy dlaždice, které přiléhají k okraji zdi, musí mít příslušnou hranu obarvenou stejnou barvou, jakou je obarvený příslušný úsek okraje. Poslední podmínka, dlaždice nesmíme při tvorbě dláždění otáčet. Dodejme ještě, že každou dlaždici smíme použít libovolně-krát.

Pomocí dláždění, resp. jeho existence nebo neexistence, můžeme snadno rozhodovat úlohy, na které se odpovídá ANO, nebo NE. Jak to uděláme? Musíme sestavit vhodnou množinu dlaždic, z kterých budeme smět vybírat při tvorbě dláždění. Horní okraj zdi obarvíme podle vstupu. Ještě potřebujeme obarvit ostatní okraje, s tím, že všechny jejich úseky obarvíme stejnou barvou (můžeme o tom tedy uvažovat jako o obarvení celého okraje jednou barvou). Dlaždice a obarvení vybíráme tak, aby dláždění existovalo, právě když odpověď na úlohu je ANO.

Možných dláždění (a jim příslušných množin dlaždic a obarvení okrajů) může existovat velmi mnoho, tak si je alespoň trochu omezme. Požadujeme, aby zeď byla vždy široká právě tak, jak dlouhý je vstup. Horní okraj tedy bude vstupu přesně odpovídat. Navíc chceme, aby výška zdi byla nejmenší možná.

To, co jsme před chvilkou popsali, je *dlaždicový program*. Ten se skládá z nějaké konečné množiny dlaždic a nějakého obarvení okrajů zdi. Formálně by se jednalo o uspořádanou čtveřici (D, ℓ, p, d), kde D je množina dlaždic a ℓ, p, d představují obarvení levého, pravého a dolního okraje.

Program na zadaný vstup odpoví ANO, pokud je možné vydláždit nějakou zeď dlaždicemi z množiny D tak, aby horní okraj byl obarven podle vstupu a zbývající okraje barvami ℓ, p a d. Neexistuje-li žádné takové dláždění, výstupem programu je NE.

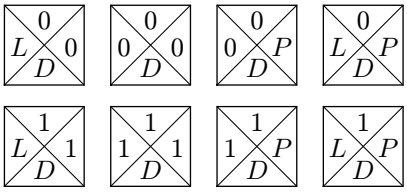
Dlaždicové programy jsou chráněná zvířátka, a tak jim budeme na vstupu předkládat pouze neprázdné řetězce.

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

Bývá hezké umět u programů ve výpočetním modelu určovat složitost. U dlaždicových programů to zvládneme jednoduše: za dobu výpočtu prohlásíme minimální výšku zdi, pro kterou existuje dláždění (časová složitost je pak maximum z dob výpočtu přes všechny vstupy dané délky), použitou paměť pak představuje plocha vydlážděné zdi. Vstupy, na něž je odpověď NE, takže žádné dláždění neexistuje, složitost neovlivní.

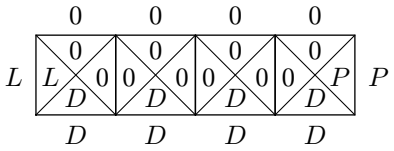
Dost bylo teoretizování, pojďme se podívat, jak se náš model návštěvníkům předvede.

Mějme na vstupu nějakou posloupnost nul a jedniček. Naším úkolem je rozhodnout, jestli je tato posloupnost konstantní, tedy zda obsahuje pouze nuly nebo pouze jedničky. Využijeme k tomu dlaždicový program s následujícími dlaždicemi. Můžeme je rozdělit do čtyř typů, každý typ existuje ve dvou barvách:



Levý okraj obarvíme L, pravý P, dolní D. Náš program určitě odpoví správně, a dokonce mu k tomu bude stačit jeden řádek. Takové odvážné tvrzení by se ale slušelo dokázat.

Jelikož všechny dlaždičky mají dolní hranu obarvenou D a žádná nemá barvou D obarvenou hranu horní, buď bude mít dláždění výšku 1, nebo vůbec nepůjde vytvořit. Pro konstantní posloupnost jistě dláždění existuje. V případě jednoprvkové posloupnosti použijeme příslušnou dlaždici čtvrtého typu, v případě posloupnosti delší pomocí vhodné dlaždice prvního typu „zvolíme číslo“ a následně ho „propagujeme“ až k pravému okraji:



Platí i to, že vše, pro co dláždění existuje, musí být konstantní posloupnost. K levému i k pravému okraji může přiléhat vždy jen jedna konkrétní dlaždice (podle hodnoty na vstupu), a k jejich spojení je potřeba „předávat“ stále stejné číslo.

Úkol 1 [3b]: Sestavte dlaždicový program, který o posloupnosti nul a jedniček na vstupu zjistí, zda je v ní počet jedniček dělitelný třemi.

Úkol 2 [2b]: Mějme nějaký dlaždicový program, který pracuje v čase t, kde t je konstanta. Dokažte, že existuje jiný dlaždicový program, který odpovídá na tutéž otázku, ale stačí mu čas 1.

Závorkování nás stále baví

V jednotlivých dílech seriálu jste si mohli zkoušet v různých výpočetních modelech ověřovat, že zadaná posloupnost je správně uzávorkovaná. Toho jsou schopné i dlaždicové programy, a na rozdíl od počítadlových strojů z minulého dílu jim ani nemusíme posloupnost nějak speciálně kódovat.

Úkol 3 [4b]: Sestavte dlaždicový program, který o posloupnosti otvíracích a zavíracích závorek na vstupu rozhodne, zda je správně uzávorkovaná.

Úkol 4 [3b]: Dokažte, že rozhodnutí uzávorkování nelze pomocí dlaždicových programů dosáhnout v lepší než logaritmické časové složitosti. Kdybyste si nevěděli rady, zkuste alespoň dokázat, že konstantní čas nestačí.

Příbuzní Turingových strojů?

Když jsme se na začátku seriálu zastavili u Turingových strojů, nepřečetli jsme si jednu ceduli o jejich příbuzných.

Připomeňme si, že stroj se v každém kroku výpočtu rozhoduje podle stavu, ve kterém se právě nachází, a podle znaku na aktuálním políčku pásky. A každé kombinaci stavu a znaku jeho program přiřadí instrukci, která se má provést. Instrukce říká, co má stroj dál udělat (na jaký znak přepsat aktuální část vstupu, kam se posunout, do jakého přejít stavu). Ke každé kombinaci stavu a znaku jsme měli právě jednu možnost.

Ale co kdyby těch možností bylo víc? Co kdybychom jedné kombinaci stavu a vstupu přiřadili hned několik možných reakcí? Přesně tak to totiž mají *nedeterministické Turingovy stroje*. Jak si ale takový stroj z možných reakcí vybere tu, kterou doopravdy provede?

Jedna možnost je představit si, že nedeterministický stroj umí *vracet* svůj výpočet. Pak můžeme říct, že nedeterministický stroj v každém kroku výpočtu vykoná první nevyzkoušenou možnou reakci (nevyzkoušenou v daném kroku) a pokračuje dál. Pokud se někdy dostane do stavu, kdy už nemá další možné reakce, nebo do koncového stavu NE, jednoduše vrací svůj výpočet až do toho kroku, kdy měl naposledy na výběr. Teprve v případě, že se vrátí do počátečního stavu a už nemá co vyzkoušet, zapíše NE.

Nebo si můžeme představit, že je stroj vybaven křišťálovou koulí (neboli orákulem), které mu pokaždé poradí takovou reakci, aby na konci výpočtu stroj odpověděl ANO. Jen pokud taková posloupnost rad neexistuje, odpověď zní NE.

Úplně mimo ale není ani představa, že v každém kroku se vesmír rozštěpí na tolik kopií, kolik má nedeterministický Turingův stroj právě možností, v každém ze vzniklých vesmírů se provede jedna reakce a výpočet pokračuje dál. Důležité pro nás je, jestli alespoň v jednom vesmíru dojde stroj do stavu ANO.

Úkol 5 [4b]: Dokažte, že dlaždicové programy jsou ekvivalentní nedeterministickým Turingovým strojům pracujícím v lineárním prostoru. Tedy máte za úkol dokázat, že jakýkoli nedeterministický Turingův stroj, který má lineární prostorovou složitost, lze reprezentovat jako dlaždicový program, a naopak, každý dlaždicový program (při našich omezeních na rozměry zdi) lze reprezentovat jako nedeterministický Turingův stroj pracující v lineárním prostoru.

Prozradíme vám malou nápovědu k předchozímu úkolu: tvrzení stačí dokázat o Turingových strojích pracujících v prostoru přesně n (kde n je velikost vstupu). Pokud totiž stroj používá prostor cn pro nějakou konstantu $c > 1$, můžeme podobně jako v úkolu 2 vytvořit jiný stroj, kterému bude stačit prostor n.

Karolína „Karryanna“ Burešová

Představme si, že máme posloupnost celých čísel

$$p_0, p_1, \ldots p_{N-1},$$

se kterou budeme průběžně provádět tyto dvě operace:

1. Změna jednoho čísla v posloupnosti.
2. Zjištění součtu čísel na nějakém intervalu $[a, b]$, tedy $p_a + p_{a+1} + \ldots + p_b$.

Nejdříve se zkusíme zamyslet, jak bychom úlohu řešili, kdybychom měli jen druhou operaci, tj. dotazy na součty na konkrétních intervalech. K řešení využijeme pole *prefixových součtů*.

Pole prefixových součtů je pole délky $N + 1$, ve kterém na indexu i leží součet prvků posloupnosti od indexu 0 až do indexu $i - 1$. Tedy

$$pref[i] = p[0] + \ldots + p[i-1], \quad pref[0] = 0$$

Není těžké si rozmyslet, že toto pole dokážeme jednoduše spočítat v čase $\mathcal{O}(N)$.

Nyní, když už známe všechny prefixové součty posloupnosti, umíme snadno spočítat součet na libovolném intervalu $[a, b]$:

$$s[a, b] = pref[b + 1] - pref[a]$$

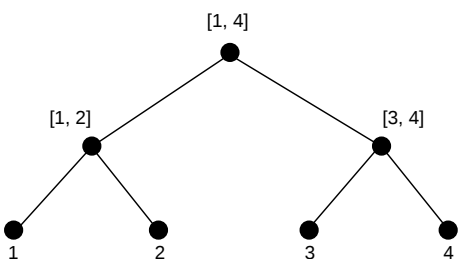
Každý dotaz dokážeme zodpovědět v konstantním čase. Celý algoritmus má tedy složitost $\mathcal{O}(N + D)$, kde N je délka posloupnosti a D je počet dotazů.

Když si do úlohy přidáme i operaci č. 1 (změna čísla v posloupnosti), tak se nám pokazí časová složitost. S prefixovými součty stále dokážeme dotaz č. 2 provádět v konstantním čase, ale při operaci č. 1 se nám může stát, že musíme změnit až všechny prefixové součty, takže složitost této operace je $\mathcal{O}(N)$ a celková složitost pro Z změn a D dotazů je v nejhorším případě $\mathcal{O}(NZ + D)$.

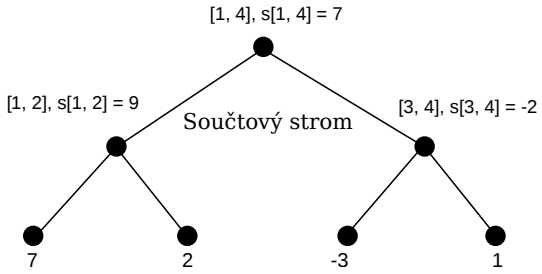
S touto složitostí se samozřejmě nespokojíme a budeme se snažit, abychom výsledné intervaly uměli co nejrychleji skládat z předpočítaných hodnot a abychom při změně posloupnosti museli změnit co nejméně hodnot. K tomu se nám bude hodit datová struktura jménem intervalový strom.

Zavedení intervalového stromu

Intervalový strom je dokonale vyvážený binární strom, jehož každý list představuje nějaký interval a všechny ostatní vrcholy reprezentují interval, který vznikne složením intervalů jejich synů. Zároveň intervaly vrcholů jedné hladiny na sebe navazují (vždy směrem zleva doprava). Z toho vyplývá, že složením intervalů z vrcholů jedné hladiny dostaneme interval, který si pamatujeme v kořeni.



Intervalových stromů existuje více druhů. Obvykle je rozlišujeme podle toho, jaké informace si v nich pamatujeme. Například ve stromě pro součty si každý vrchol pamatuje součet na svém intervalu, ve stromě pro maxima si pamatuje maximum na intervalu, apod. Můžeme ale klidně mít strom, který si pamatuje, jestli celý jeho interval obsahuje jen jednu hodnotu a pokud ano, tak jakou.



My se teď zaměříme na intervalový strom pro součty a pomocí něj vyřešíme úvodní úlohu.

Na začátku budeme chtít, aby v listech intervalového stromu byly hodnoty původní posloupnosti, přičemž první a poslední list stromu necháme volné, později uvidíme, proč. Zároveň ale chceme, aby tento strom byl dokonale vyvážený.

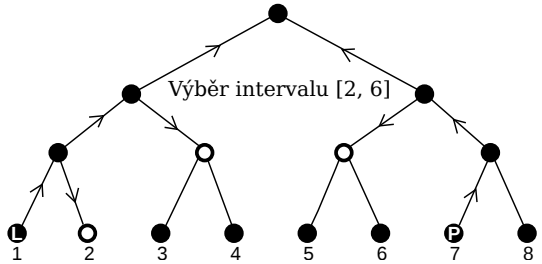
Posloupnost tedy prodloužíme tak, aby její velikost byla mocnina dvojky minus dva (na její konec přidáme nějaké prvky). Všimněte si, že tím jsme strom nezvětšili více než dvakrát a že nám nezáleží na tom, jaké prvky jsme do stromu přidali, protože s nimi nikdy nebudeme pracovat. Nyní k jednotlivým operacím.

Změnu čísla v posloupnosti uděláme jednoduše. Zjistíme, o kolik se hodnota prvku posloupnosti změní, najdeme odpovídající list a k tomuto listu a ke všem jeho předkům přičteme daný rozdíl. Tím jsme upravili všechny intervaly, do kterých tento prvek patří.

Nyní se podívejme, jak ze stromu zjistíme součet na nějakém intervalu $[a, b]$. Jinými slovy: potřebujeme ze stromu vybrat takové vrcholy, aby sjednocení jejich intervalů byl náš dotazovaný interval, a zároveň chceme, aby těchto vrcholů bylo co nejméně.

Součet intervalu $[a, b]$ zjistíme tak, že si ve stromě najdeme listy reprezentující pozice $a-1$ a $b+1$ posloupnosti a jejich nejbližšího společného předka p. Nyní budeme postupovat z listu od $a - 1$ až do p a vždy když do nějakého vrcholu přijdeme z levého syna, tak do výsledku přidáme interval pravého syna. Stejně tak postupujeme od $b + 1$ k p a pokud do vrcholu přijdeme z pravého syna, tak přidáme jeho levého syna.

Všimněte si, že při takovémto průchodu složíme celý interval. Vše je vidět na nasledujícím obrázku:



Způsobů, jak pracovat z intervalovým stromem a zjišťování informací z něj, je více. Toto byl jeden z nich.

Změna prvku posloupnosti má časovou složitost $\mathcal{O}(\log N)$, protože jsme na každé hladině změnili pouze jeden interval a strom má $\mathcal{O}(\log N)$ hladin. Zjištění součtu na intervalu

má také složitost $\mathcal{O}(\log N)$, jelikož jsme do výsledku přidali maximálně $2 \log N$ intervalů: nejvýše $\log N$ při cestě z listu $a-1$ a $\log N$ při cestě z $b+1$.

Implementace intervalového stromu

Při implementaci intervalového stromu využijeme jeho dokonalé vyváženosti a budeme jej implementovat v poli (stejně jako se do pole ukládá halda). Kořen stromu bude v poli na indexu 1, vrcholy z druhé hladiny budou mít postupně indexy 2 a 3, ..., až listy budou mít indexy $N, \ldots, 2N-1$. V této reprezentaci platí pro vrchol s indexem i následující pravidla:

1. $2i$ a $2i+1$ jsou jeho synové.
2. $\lfloor i/2 \rfloor$ je jeho předek (pro $i > 1$).
3. Pokud je i sudé, tak je vrchol levým synem, jinak pravým.
4. Pro sudé i je $i+1$ pravý bratr, pro liché i je $i-1$ levý bratr.

Nyní víme vše potřebné, tak se podívejme na samotnou implementaci v jazyce C:

```
int N = 100;    // velikost posloupnosti
int posl[100];  // posloupnost
int *strom;     // intervalový strom

// Deklarace funkcí
void inic(int N);
void pricti(int index, int hodnota);
int soucet(int A, int B);

// Inicializace intervalového stromu
// Pozor: prvky posloupnosti indexujeme 1, ..., N
void inic(int N) {
  // Najdeme nejbližší vyšší mocninu dvojky
  int listy = 1;
  while (listy<N+2) listy = listy*2;
  // Pro strom potřebujeme 2*(počet listů) vrcholů
  // (nepoužíváme strom[0])
  strom = (int*)malloc(sizeof(int)*2*listy);
  N = listy;
  for (int i=0; i<2*listy; i++) strom[i] = 0;
  // Na příslušná místa přičteme hodnoty posloupnosti
  for (int i=0; i<N; i++)
    pricti(i, posl[i]);
}

// Přičtení hodnoty na dané místo posloupnosti
void pricti(int index, int hodnota) {
  int k = N + index;
  while(k>0) {
    strom[k] = strom[k] + hodnota;
    k = k/2;
  }
}

// Zjištění součtu na intervalu
int soucet(int A, int B) {
  int souc = 0;
  int a = N + A - 1;
  int b = N + B + 1;
  while (a!=b) {
    // Pokud je a levý syn, tak přičti pravého bratra
    if (a%2==0) souc = souc + strom[a+1];
    // Pokud je b pravý syn, tak přičti levého bratra
    if (b%2==1) souc = souc + strom[b-1];
    // Přesun na otce
    a = a/2; b = b/2;
  }

  // Navíc jsme přičetli syny společného předka.
  souc = souc - strom[2*a] - strom[2*a+1];
  return souc;
}
```

V této implementaci jsme strom upravovali zdola směrem nahoru. Existuje ještě rekurzivní implementace, v níž se strom upravuje od kořene směrem dolů, ale tu si zde ukazovat nebudeme.

Cvičení

- Naprogramujte rekurzivní implementaci operací (strom se prochází shora dolů).
- Jak by vypadala implementace intervalového stromu pro maxima?

Použití intervalového stromu

Intervalový strom je silný nástroj, kterým se dá vyřešit spousta úloh. Ale než ho začnete používat, tak si vždy rozmyslete, zda úloha nelze řešit elegantněji bez intervalového stromu. Ne všechny druhy intervalových stromů se dobře implementují.

Intervalový strom obvykle použijeme, pokud potřebujme průběžně zjišťovat informace o intervalech a zároveň je i měnit. Pokud používáme jen jednu z těchto operací (a tu druhou jen zřídka), existuje často lepší řešení než intervalový strom – viz úvodní příklad.

Fenwickův strom

Fenwickův strom, někdy také nazývaný jako *finský strom,* je v podstatě jen strom reprezentovaný v poli. Jeho používání je podobné jako používání intervalového stromu pro součty. Rozdíl je jen v implementaci daných funkcí. My si Fenwickův strom opět ukážeme na úvodním příkladu. Zase tedy budeme potřebovat funkci pro změnu hodnoty v posloupnosti a funkci pro zjištění součtu na intervalu. (Ve skutečnosti zjistíme dva prefixové součty a z nich pak spočítáme výsledný interval.)

Fenwickův strom je poněkud magická datová struktura. Abychom si tuto magii mohli užít, zvolíme trochu netradiční způsob vysvětlování a nejdříve si ukážeme, jak se Fenwickův strom implementuje a teprve pak si vysvětlíme, jak to všechno funguje.

Fenwickův strom bude pole velikosti $N+1$, kde index 0 nebudeme používat. Používat budeme pouze prvky $1, \ldots, N$, které všechny na začátku nastavíme na 0. Pokud v posloupnosti změníme hodnotu, stejně jako u intervalového stromu, ve Fenwickově stromě na některá místa přičteme rozdíl oproti předchozí hodnotě.

```
void pricti(unsigned int index, int rozdil) {
  while (index<=N) {
    strom[index] += rozdil;
    index = index + (index & -index);
    // "&" značí bitový AND
  }
}
```

A zde je funkce pro zjištění prefixového součtu:

```
int pref_soucet(unsigned int index) {
  int soucet = 0;
  while (index>0) {
    soucet = soucet + strom[index];
    index = index & (index-1);
  }
  return soucet;
}
```

Toť celá implementace. No, nevypadá na první pohled magicky? Pokud chcete vědět, jak tohle celé funguje, tak čtěte dál.

Ve Fenwickově stromě je na indexu 1 uložen první prvek, na indexu 2 součet prvního a druhého, na indexu 3 třetí prvek na indexu 4 součet prvních čtyř, ... na indexu N je uložen součet posledních 2^K hodnot, kde K je pozice prvního jedničkového bitu v binárním zápise čísla N. Ve stromě máme tedy uloženou takovou pravidelnou strukturu intervalů.

Nyní se podíváme, co dělají naše magické funkce na posouvání ve stromě a pak najednou bude všechno jasné. Ve výrazu `index & (index-1)` z funkce `pref_soucet()` se neděje nic jiného než, že se vynuluje nejpravější jedničkový bit v indexu. Tím se dostaneme na první interval, který jsme ještě nepřičetli. V momentě, kdy se dostaneme na index 0, tak už máme dotazovaný interval kompletní a výpočet můžeme ukončit.

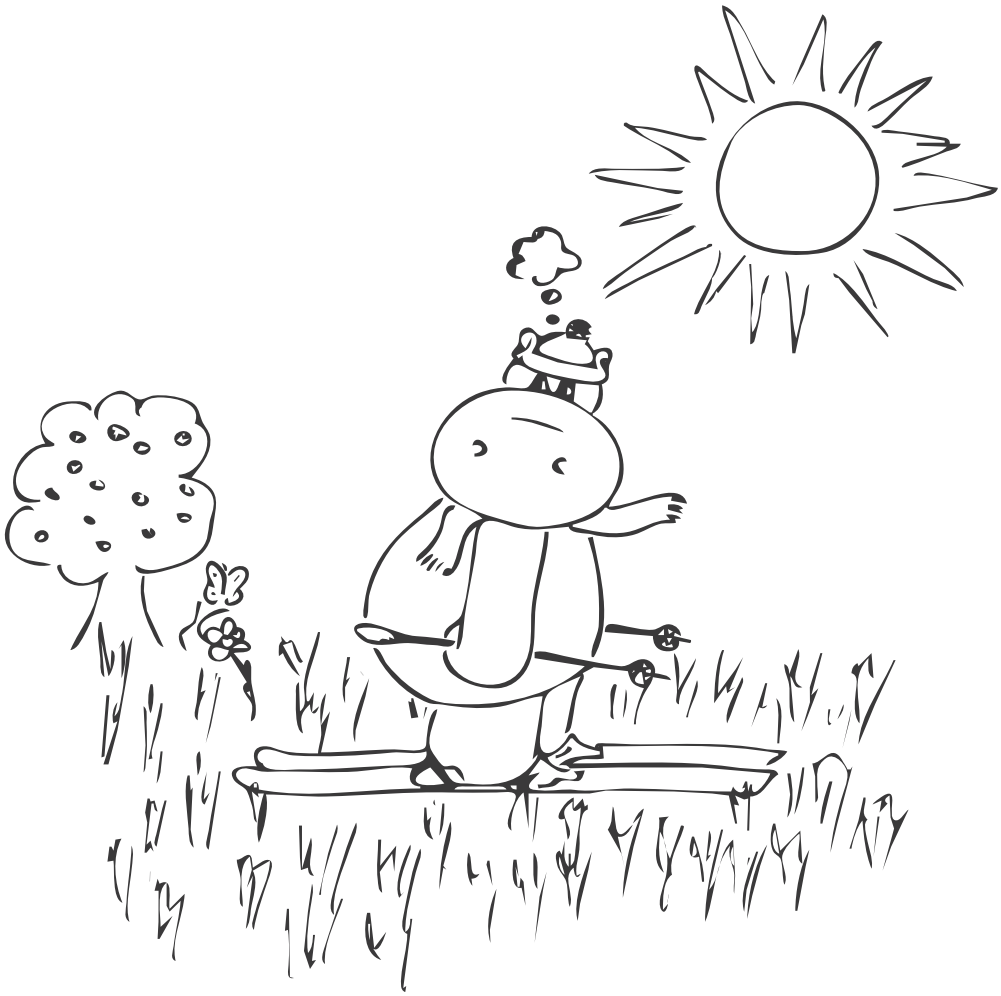
Výraz `index + (index & -index)` dělá to, že se v pomyslném stromě intervalů posune o úroveň výš.[2] Pokud jsme tedy v intervalu o velikosti 2, tak se dostaneme do intervalu velikosti 4, který daný interval obsahuje (tento interval je jednoznačný). Samotný výpočet dělá to, že v čísle `index` vezme nejpravější jedničku a znova ji přičte.

Fenwickův strom se používá hlavně kvůli jednoduchosti jeho naprogramování a také kvůli efektivitě samotného výpočtu a nevelké náročnosti na paměť. Při jeho implementaci doporučujeme dávat si pozor na správnost bitových funkcí.

Cvičení

- Rozmyslete si, že oba magické výpočty opravdu dělají to, co mají, a také, proč vše vlastně funguje.

Karel Tesař



[2] Magická operace `index & -index` funguje jen v případě, že se jako reprezentace záporného čísla používá tzv. dvojkový doplněk: `-k == ~k + 1`, neboli všechny bity čísla se znegují a pak se přičte jednička.

Čokolády

V této sérii jsme, jak už je naším zvykem, opět rozdávali čokolády. Tentokrát za to, pokud se vám podařilo získat z alespoň pěti úloh alespoň polovinu možných bodů.

To se podařilo celkem devíti z vás. Shodou okolností je to právě prvních devět řešitelů v pořadí zisku bodů za tuto sérii. Gratulujeme a užijte si sladkou odměnu.

26-3-1 Výklad z drahokamů

Protože věštění, byť z drahokamů, je komplikovaná věc, začneme jednodušší úlohou. Budeme pro začátek hledat co největší žlutý čtverec.

Podobné úlohy se vyskytují docela často a zkušenému řešiteli už něco našeptává „dynamika“.

Každý (i největší) jednobarevný čtverec má nějaký pravý dolní roh. My si tedy pro každou pozici spočítáme největší čtverec, který od něj vede doleva nahoru a potom jen vybereme nejlepší možnost.

Pokud je aktuální pozice červená, je zřejmě výsledek 0, neboť je to největší žlutý čtverec, který se zde nachází. Naopak, pokud je žlutá, podíváme se na pozici o jedna doleva, o jedna nahoru a o jedna šikmo doleva nahoru. Z nich vybereme minimum jejich čtverců a to zvětšíme o 1 – na čem se „zarazí“ čtverec odpovídající tomuto minimu, na tom se zarazí i náš čtverec v aktuální pozici. Naopak, tím jak se čtverce sousedních políček překrývají, tak nám dají k dispozici odpovídající žlutou plochu, kterou jen aktuální pozicí rozšíříme. Zkuste si to nakreslit.

Samozřejmě, potřebujeme počítat maximální čtverce v takovém pořadí, abychom všechny tři sousedy, do kterých nahlížíme, měli již spočítané. Například po řádcích zleva doprava, stejně jako se čte.

A jako malý technický trik, abychom nemuseli řešit vykukování z pole na levém a horním okraji, připravíme si řádek a sloupeček nul jako jakýsi rámeček.

Nyní tedy, jak na naši těžší úlohu? Všimneme si, že každý šachovnicový čtverec je podčtvercem nějaké velké šachovnice takových rozměrů, jaké má celá mřížka. A tyto velké šachovnice existují právě dvě – jedna, která má vlevo nahoře žlutý drahokam, a k ní inverzní, která má vlevo nahoře červený. Celou mřížku si tedy převedeme a budeme pokládat žluté drahokamy tam, kde barva na dané pozici odpovídá první šachovnici, a červené, kde druhé. Tím jsme úlohu převedli na nalezení největšího jednobarevného čtverce – což je buď žlutý, nebo červený. Žlutý už najít umíme a nalezení červeného bude ponecháno na čtenáři.

Jak paměťová, tak časová složitost jsou lineární s velikostí vstupu. Ke každému políčku vstupu si pamatujeme konstantně mnoho informací (jeho převedenou barvu a jeho maximální čtverec doleva nahoru). Obdobně, při převodu políčka uděláme konstantně mnoho práce, a při počítání minima také koukáme jen na tři okolní políčka.

Program (Python):
`http://ksp.mff.cuni.cz/viz/26-3-1.py`

Michal „vorner“ Vaner

26-3-2 Střih látky

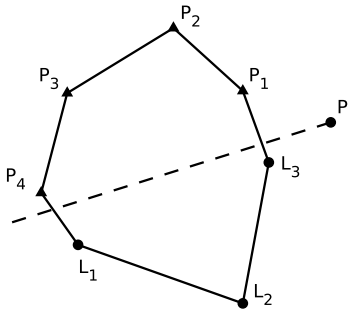
Označme si vrcholy mnohoúhelníka V_0 až V_{n-1} a střihovou polopřímku p (zadanou počátkem P a vektorem u). Dovolme si z úsporných důvodů zanedbat „ošklivé“ případy, kdy se polopřímka mnohoúhelníka dotýká v jednom bodě či splývá s některou jeho stranou. Nebude těžké ošetřit je dodatečně přidáním několika podmínek na vhodná místa.

Víceméně z definice konvexity plyne, že p bude mít s mnohoúhelníkem nejvýše dva průsečíky.

Všimněme si, že doopravdy vlastně chceme zjistit, se kterými stranami mnohoúhelníka se protíná. Dopočítat konkrétní souřadnice průsečíků už je pak jen drobné cvičení ze středoškolské geometrie. Pokud si s takovými úlohami nerozumíte, nastala správná chvíle přečíst si geometrickou kuchařku.[3]

V průběhu řešení budeme často chtít vědět, jestli bod od nějaké (polo)přímky leží napravo, nebo nalevo (při pohledu po směru polopřímky). Prozatím nám uvěřte, že to snadno zjistíme v konstantním čase.

Všimněme si, že nějaká hrana mnohoúhelníka kříží p právě tehdy, když jeden její konec je vlevo od p a druhý vpravo. Dále si všimněme, že vrcholy nalevo a napravo od p tvoří v mnohoúhelníku souvislý úsek (díky konvexitě). My tedy vlastně v posloupnosti vrcholů nehledáme nic než hranici mezi levými a pravými vrcholy (díky cykličnosti existují dvě). To svádí k tomu použít něco jako binární vyhledávání.



Zde nám ovšem komplikuje život fakt, že posloupnost je cyklická a záleží na tom, jaký vrchol vezmeme za počáteční. Uvažme např. posloupnosti `PLLPPPP` a `PPPPLLP`. Pouhým pohledem na prostřední prvek (v obou případech `P`) nepoznáme, ve které polovině máme pokračovat v hledání.

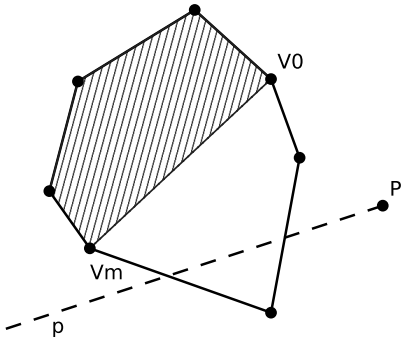
Pokud bychom znali alespoň jeden vrchol nalevo od přímky a jeden napravo, je to jednoduché: v těchto dvou vrcholech posloupnost rozstřihneme. Tím vzniknou dvě posloupnosti tvaru `LLLPPP` či `PPPLLL`, v obou z kterýchž najdeme hranici prachobyčejným binárním vyhledáváním.

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

Jak takové dva protilehlé vrcholy sehnat? Uvažme body V_0 a $V_m := V_{n/2}$. Mohou nastat dva případy:

a) V_0 a V_m leží na opačných stranách p. Vyhráno.

b) V_0 a V_m leží na stejné straně p. Pak p neprotíná úsečku V_0V_m. Ta však rozděluje mnohoúhelník na dvě poloviny – p tedy prochází jen jednou z nich. Tu druhou můžeme zahodit a pokračovat v hledání rekurzivně s mnohoúhelníkem o polovičním počtu bodů.

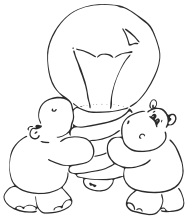


Algoritmus se tedy skládá ze dvou fází. V první hledáme dva body na opačných stranách p. Pokud takové dva body najdeme, přejdeme do druhé fáze. Jinak se zastavíme, až nám zbude trojúhelník, pro který už úlohu vyřešíme triviálně testem každé ze tří hran. V druhé fázi binárně hledáme hrany protínající se s p postupem popsaným výše.

V obou fázích nás každý krok stojí konstantní čas a zmenšíme jím počet zpracovávaných bodů na polovinu. Tím dosáhneme celkové složitosti $\mathcal{O}(\log n)$.

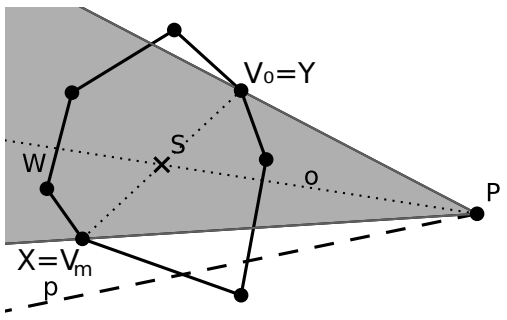
Nalezení správné poloviny bodů

Až dosud je algoritmus docela přímočarý a vymyslitelný. Zatajili jsme ovšem jeden na první pohled drobný detail: jak z polohy p poznáme, kterou polovinu bodů ($V_1, \ldots, V_{m-1}$, nebo $V_{m+1}, \ldots, V_{n-1}$) zahodit?



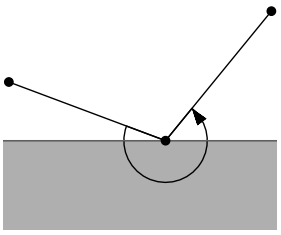
To překvapivě není vůbec zřejmé. Existuje spousta způsobů, jak to „umlátit“, např. počítáním nějakých vzdáleností a průsečíků. To jsou ale výpočty komplikované a nepřesné, zkusíme si proto ukázat hezčí, byť možná myšlenkově trochu náročnější řešení.

Představme si úhel V_0PV_m jako jakýsi „stín“. Víme, že p neleží ve stínu (jinak by se protínala s V_0V_m), tedy kdykoli se nějaký objekt nachází celý ve stínu, nemá průsečík s p. Označme si S střed V_0V_m a o polopřímku PS („pseudoosu“ stínu). Ta nám rozdělí rovinu na dvě poloroviny.



Označme si X ten z bodů V_0, V_m, který leží ve stejné polorovině jako p a Y ten opačný.

Podívejme se nyní na sousedy X. Alespoň jeden z nich leží ve stínu, neb kdyby oba ležely na světle, měl by mnohoúhelník vnitřní úhel větší než 180 °:



Označme si takového souseda W. Protože $X, W, \ldots, Y$ tvoří konvexní mnohoúhelník, musí jeho obvod „zatáčet“ pořád na stejnou stranu – na obrázku výše doprava (směrem k Y). Tedy pokud je W ve stínu, budou tam i všechny následující vrcholy (přinejmenším ty ve stejné polorovině). Tedy polovina našeho mnohoúhelníku tvořená body $X, W, \ldots, Y$ se nemůže protnout s p: část se jí nachází ve stínu a druhá část v opačné polorovině vůči o než p. Tyto body tedy můžeme (kromě krajních X, Y) zahodit.

Pokud jsou oba sousedé ve stínu, p nemůže mít s mnohoúhelníkem žádný průnik.

Tudíž potřebujeme umět zjistit, (1) ve které polorovině se nachází p, (2) jestli je daný soused X ve stínu.

To jsou ale jen další instance naší operace „poloha bodu vůči polopřímce“. Zvolíme si libovolný bod na p (třeba $P + u$) a podíváme se, na které straně je od o. Pak X je ten z V_0, V_m, který leží na stejné straně. A nějaký soused X je ve stínu, právě když leží od PX na opačné straně, než X od o.

Zbývá nám nějak zařídit onu mýtickou operaci zjištění polohy bodu vůči polopřímce: mějme nějaký bod B a polopřímku q s počátkem Q a směrovým vektorem w. Nyní se nám bude hodit vektorový součin.[4] Z pravidla pravé ruky si snadno rozmyslíte, že $w \times (B - Q)$ je kladný právě tehdy, když B je nalevo od q, záporný, když napravo, a nulový, když leží na q.

Další (byť méně elegantní) způsob je reprezentovat si přímku rovnicí $y = kx + q$. Detaily necháme čtenáři k rozmyšlení.

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

Filip Štědronský

26-3-3 Grafová

Požadovaný důkaz je ve své podstatě celkem jednoduchý, k dokázání použijeme *matematickou indukci*.

Nechť je dán graf s N vrcholy, M hranami a minimálním stupněm k. Důkaz provedeme konstrukcí, ze které nám rovnou vyplyne lineární algoritmus. Induktivně sestrojíme vhodnou posloupnost vrcholů:

1. V prvním kroku zvolíme libovolný vrchol v_1.
2. V i-tém kroku máme již cestu z $i-1$ vrcholů, chceme přidat i-tý. Vybereme tedy libovolného souseda v_{i-1}, který ještě nebyl vybrán, a zvolíme jej jako v_i. Pokud takový už neexistuje, skončíme.

Takto jsme dostali v zadaném grafu cestu $v_1 v_2 \ldots v_r$ pro nějaké $r \leq N$.

Nyní nahlédneme, že všichni sousedé vrcholu v_r musí ležet na sestrojené cestě, neboť jinak bychom mohli cestu

[4] `http://en.wikipedia.org/wiki/Cross_product`

prodloužit a provést další krok indukce. Vrchol v_r je tedy spojen hranou s aspoň k vrcholy na sestrojené cestě a tudíž určitě existuje vrchol v_s, který je sousedem v_r a pro který platí $s \leq r - k$. Nyní už můžeme jásat, neboť vrcholy $v_s v_{s+1} \ldots v_r$ nám tvoří kružnici délky alespoň $k + 1$.

Postup užitý v důkaze můžeme implementovat přímo jedním průchodem grafu do hloubky, což nám dává časovou i paměťovou složitost $\mathcal{O}(N + M)$.

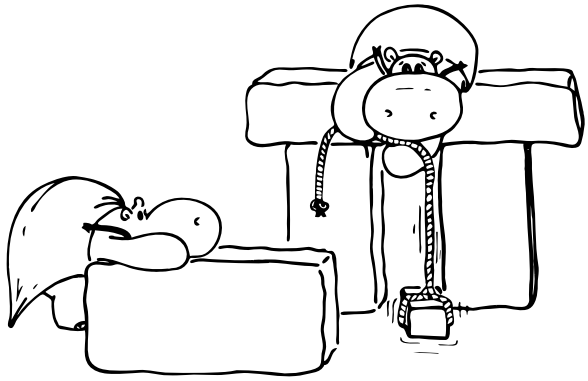
Program (C++):
`http://ksp.mff.cuni.cz/viz/26-3-3.cpp`

Mark Karpilovskij

26-3-4 Kladení pastí

Snažíme se rozmístit pasti v pralese. Přitom musíme dodržet pravidlo, že každá pěšinka spojující dvě křižovatky sousedí alespoň s jednou pastí. V řeči teorie grafů křižovatku nazveme *vrchol* a pěšinku *hrana*. Rozmístění pastí splňující výše popsanou podmínku se obvykle nazývá *vrcholové pokrytí*. Na stavbu pasti v každém vrcholu musíme vynaložit úsilí U_i (zaplatit cenu). Hledáme proto takzvané *minimální vážené vrcholové pokrytí*.

Podívejme se nejprve na řešení lehčí varianty, ve které je graf tvořen jedinou cestou. Postup potom dokážeme zobecnit i pro těžší verzi.



Lehčí varianta

Pro každou z křižovatek máme dvě možnosti. Buď na ni past umístíme, nebo neumístíme. Pro N vrcholů je tedy všech možných rozmístění pastí 2^N. Některá z těchto rozmístění samozřejmě nejsou vrcholová pokrytí. Každopádně ani tak není v našich silách vyzkoušet všechny možnosti.

Co kdyby nás pro začátek nezajímalo samotné rozložení pastí, ale pouze celková minimální cena? Zkusme ji počítat postupně pro jednotlivé počáteční úseky cesty délky i. (Délku budeme měřit třeba v počtu vrcholů.)

Zvlášť si zapamatujeme minimální cenu p_i tohoto počátečního úseku délky i za podmínky, že v posledním (i-tém) vrcholu nalíčíme past. Naopak n_i bude značit cenu začátečního úseku délky i v případě, že v i-tém vrcholu past není. Celkovou minimální cenu tohoto úseku cesty snadno spočteme jako $\min(p_i, n_i)$.

Pro první vrchol určíme minimální ceny jednoduše: $p_1 = U_1$ a $n_1 = 0$. Pro ostatní využijeme toho, co jsme spočítali dříve. Ceny tedy budeme počítat postupně pro všechna i od 1 do N. Pokud je v i-tém vrcholu past, tak minimální cenu p_i spočteme z celkové minimální ceny předchozího úseku jako $p_i = \min(p_{i-1}, n_{i-1}) + U_i$. V případě, že past do vrcholu i neumístíme, tak jsme ji museli umístit do předchozího vrcholu $i - 1$. Platí tedy, že $n_i = p_{i-1}$.

Tímto postupem snadno v lineárním čase $\mathcal{O}(N)$ spočteme minimální cenu potřebnou k rozmístění pastí. Původně jsme však chtěli minimální vrcholové pokrytí i najít. Zrekonstruujeme jej v opačném pořadí od N do 1.

Pokud se rozhodujeme, zda použít i-tý vrchol, pak jej vybereme tehdy, když se nám to vyplatí. Tedy když platí $p_i \leq n_i$. Pokud však vrchol nevybereme, nutně musíme vybrat vrchol $i - 1$. V tom případě se znovu rozhodujeme až u vrcholu $i - 2$.

Těžší varianta

Jak postupovat v případě, že cestičky a křižovatky tvoří strom? Minimální ceny chceme opět počítat postupně z předchozích mezivýsledků. Představme si celý strom jako zakořeněný (všechny vrcholy kromě kořene mají jednoznačně určeného otce). Minimální ceny p_i a n_i pak budeme počítat pro všechny podstromy, tj. pro vrchol i se všemi jeho potomky.

Potřebné pořadí pro postupné počítání cen získáme tak, že projdeme celý strom pomocí algoritmu prohledávání do hloubky.[5] Při opuštění vrcholu spočteme ohodnocení p_i a n_i jeho podstromu následovně:

- Pokud je vrchol i list, nastavíme minimální cenu s použitím daného vrcholu $p_i = U_i$.
- Cena bez umístění pasti do listu potom bude $n_i = 0$.
- Pro ostatní vrcholy určíme minimální cenu s použitím pasti ve vrcholu i jako součet ohodnocení daného vrcholu a celkové minimální ceny podstromů všech jeho synů:

$$p_i = U_i + \sum_{j \in Syn(i)} min(p_j, n_j),$$

  kde $Syn(i)$ značí množinu synů vrcholu i.
- Pokud do vrcholu i past neumístíme (dle předchozího bodu), musíme pasti nalíčit ve všech jeho synech:

$$n_i = \sum_{j \in Syn(i)} p_j.$$

Při opouštění posledního vrcholu tak spočítáme minimální cenu rozmístění pasti v celém stromě.

Stejně jako v lehčí variantě musíme ještě určit, ve kterých vrcholech máme pasti nalíčit, abychom dodrželi spočítanou minimální cenu. Projdeme tedy znovu náš strom. Na křižovatku i past umístíme, pokud opět platí $p_i \leq n_i$. Když však past do vrcholu neumístíme, musíme ji přidat do všech jeho synů.

Tím máme i pro těžší variantu celkem slušnou časovou a paměťovou složitost $\mathcal{O}(N)$. Strom pouze dvakrát projdeme. Přidané výpočty sice závisí na počtu synů daného vrcholu, dohromady je však synů stejně jako vrcholů.

Pro (NP-)úplnost ještě dodejme, že pro obecný graf není v současnosti známé žádné efektivní řešení.

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

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

Jenda Hadrava

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

26-3-5 Rozvrh kovárny

Až na pár originálních řešení, často fungujících a optimálních, zvolili všichni hladový přístup zezadu, čili hladový algoritmus, který si popíšeme. Stručně řečeno: seřadíme si požadavky dle termínu dokončení, bereme je od nejvyšší hodiny dokončení po nejmenší a dáváme do rozvrhu do nejvyšší volné hodiny tak, že vždy bereme nezařazený požadavek s nejvyšší prioritou, který lze v tu hodinu vyrábět.

Jak vybírat požadavek s nejvyšší prioritou? Řešení napovídal symbol kuchařky u zadání odkazující se na kuchařku o haldách.[6] Na požadavky použijeme haldu, konkrétně maximovou, tedy řazenou od největšího prvku po nejmenší. Náš algoritmus tedy bude takový hladový.

Trochu podrobněji: Nejprve si seřadíme požadavky dle termínu dokončení a zavedeme si proměnnou *hodina*, což bude poslední možná hodina, do níž můžeme rozvrhnout požadavek, a kterou zinicializujeme na maximální hodinu dokončení nějakého požadavku mínus jedna. Pak bereme požadavky sestupně dle hodiny dokončení.

V každé iteraci se nejprve podíváme, jestli mezi požadavky nejsou nějaké s termínem *hodina*+1 (plus jedna kvůli tomu, že s výrobou nástroje musíme začít hodinu před termínem). Všechny takové přidáme do haldy. Pak vybereme z haldy požadavek s nejvyšší prioritou, rozvrhneme vybraný požadavek na hodinu *hodina* a tuto proměnnou snížíme o jedna. Skončíme, když se *hodina* dostane pod nulu.

Počet iterací je roven největší hodině dokončení H mezi požadavky. H však není polynomiální v N, ani v délce vstupu, náš algoritmus tedy není ani polynomiální. Nejčastější chybou v řešeních právě bylo, že skončila s algoritmem, který závisel na H.

Oprava na polynomiální algoritmus je však nasnadě: pokud je halda prázdná, rovnou posuneme proměnnou *hodina* na nejvyšší hodinu dokončení nezpracovaného požadavku mínus jedna. Díky tomu v každé iteraci odebereme jeden prvek z haldy a přidáme ho do rozvrhu. Alternativně je možné na začátku inicializovat proměnnou *hodina* na N a všimnout si, že si tím nic nepokazíme, protože stejně nebudeme vyrábět více jak N hodin.

Seřazení dle hodin dokončení jistě zvládneme v $\mathcal{O}(N \log N)$, kde N je počet požadavků. Z kuchařky víme, že haldové operace jako přidávání, nalezení největšího prvku a jeho smazání trvají logaritmus z velikosti haldy, která bude v nejhorším případě obsahovat všech N požadavků.

V každé iteraci vybereme z haldy maximum (za $\mathcal{O}(\log N)$) a smažeme ho, můžeme ale do haldy přidat hodně požadavků. Každý požadavek však dáme do haldy jen jednou, což celkově dává $\mathcal{O}(N \log N)$ času na přidávání. Časovou složitost jsme tedy určili na $\mathcal{O}(N \log N)$. Co se týče paměti, vystačíme si jistě s prostorem $\mathcal{O}(N)$.

Důkaz správnosti

Zbývá už jen dokázat, že náš algoritmus dává optimální rozvrh, tedy že nelze udělat rozvrh s větším součtem priorit. Budeme postupovat sporem, tedy předpokládat, že pro nějaký seznam požadavků existuje lepší rozvrh než ten, jež našel náš algoritmus. Postupně dojdeme k nějaké zjevné nepravdě, což znamená, že lepší rozvrh neexistuje.

Rozvrh R_1 vytvořený naším algoritmem a lepší rozvrh R_2 se musí někde lišit, přičemž nás zajímají jen rozdíly v prioritách požadavků pro konkrétní hodiny. Jelikož R_2 má vyšší součet priorit, musí existovat hodina H, ve které má lepší rozvrh požadavek P s vyšší prioritou, než má požadavek zpracovávaný v hodině H v našem rozvrhu.

Provedeme výměnu a upravíme lepší rozvrh R_2 na podobný rozvrh se stejným součtem priorit. Požadavek P musíme vyrábět v našem rozvrhu R_1 v hodině H', kde $H' > H$, jinak bychom ho v hodině H vytáhli z haldy. Nyní zaměníme v rozvrhu R_2 požadavky v hodinách H a H' a všimneme si, že jsme si nepokazili rozvrh: požadavek z hodiny H' vyrábíme dříve a požadavek z hodiny H lze v našem rozvrhu vyrábět v hodině H', takže to musí jít i v rozvrhu R_2.

Jelikož upravený rozvrh R_2 má vyšší součet priorit než R_1, tak i po výměně musí existovat hodina, v níž se v R_2 vyrábí požadavek s vyšší prioritou než v R_1. Mohli bychom tedy takovéhle výměny provádět donekonečna, což však nejde, neboť po každé výměně vzroste alespoň o jedna počet hodin, kde se shodují rozvrhy R_1 a R_2, konkrétně o hodinu H'. Čili máme kýžený spor.

Tím jsme úspěšně vykovali algoritmus. Užijte si zbytek zimy ... tedy, chtěl jsem říct jara.

Program (C++):
`http://ksp.mff.cuni.cz/viz/26-3-5.cpp`

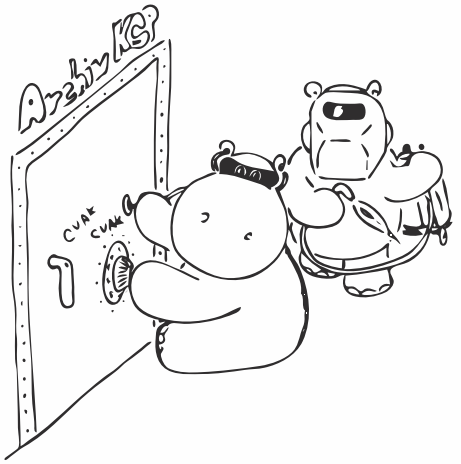
Pavel „Paulie“ Veselý

26-3-6 Trezor

Všichni, kdo se pokusili o tuto úlohu, se nezastavili u lehčí verze a rovnou se pustili do verze plné. Přesto se na chviličku u lehké verze zastavme, uvidíme, že se dá řešit pěkným trikem.

V lehčí verzi nám jednotlivé vnitřní klíče dají dohromady číslo ve dvojkovém zápise – sérii jedniček a nul – a my chceme všechny nuly změnit na jedničky. To určitě můžeme udělat tak, že použijeme právě klíče s exponenty odpovídající pozicím nul v tomto čísle. Nejde to však lépe, s menším počtem klíčů, než kolik je v čísle nul?

Nejde. Žádným způsobem totiž přidáním jednoho klíče nezměníme více než jednu nulu na jedničku. I když by došlo k „přetečení“ nějakého řádu, tak se nám na tomto místě objeví nula a zbytek čísla se chová stejně, jako kdybychom jedničku přičítali k o jedna většímu řádu. Zadání úlohy se tedy dá přeformulovat přímo na spočítání počtu nul ve dvojkovém zápisu součtu vnitřních klíčů.



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

Protože se nám v lehčí variantě stejné klíče neopakují, tak rovnou dostaneme číslo ve dvojkové soustavě, kde počet jedniček odpovídá počtu vnitřních klíčů. Stačí tedy v lineárním čase a konstantní paměti při čtení vstupu najít největší exponent vnitřního klíče a odečíst od něj celkový počet vnitřních klíčů, čímž rovnou dostaneme potřebný počet vnějších klíčů.

Při řešení těžší varianty ale již bude potřeba hodnoty klíčů nějak sčítat. Nemusíme operovat přímo s hodnotami, bohatě nám stačí sčítat exponenty. Označme si pomocí N počet exponentů (vnitřních klíčů) na vstupu a jako M maximální exponent, který se na vstupu může vyskytnout. Podle velikosti M můžeme úlohu řešit dvěma různými způsoby:

Pro malé M, pokud $M < N \log N$, je nejvýhodnější použít *přihrádkové třídění*. Maximální exponent, ke kterému se můžeme v průběhu výpočtu dostat, je $M + \log N$ (kdyby všech N vnitřních klíčů bylo maximálního exponentu), takže si vyrobíme takto velké pole přihrádek a pak v lineárním čase spočítáme počty jednotlivých exponentů na vstupu.

Pak nám stačí toto pole projít odzadu, počítat počet nul, a kdykoliv se nám v nějaké přihrádce vyskytne číslo větší jak jedna, tak ho necháme „přetéct“ – v aktuální přihrádce necháme zbytek po dělení dvěma (tedy buď 0 nebo 1) a do další přihrádky přičteme (celočíselnou) polovinu hodnoty z aktuální přihrádky. Tím úlohu vyřešíme v čase $\mathcal{O}(N+M)$ a s pamětí $\mathcal{O}(M + \log N)$.

Program (Python) – přihrádky:
`http://ksp.mff.cuni.cz/viz/26-3-6-prihradky.py`

Pro velká M (klidně řádově větší jak N) by se nám však již pole přihrádek do paměti vůbec nemuselo vejít, nemluvě o tom, že by v něm mohly být velké „díry“, ve kterých bychom trávili zbytečně moc času. Pak je výhodnější zvolit jiný postup.

Všech N vnitřních klíčů si můžeme na začátku programu v čase $\mathcal{O}(N \log N)$ seřadit, seřazené naskládat do spojového seznamu, a pak ho opět jako v předcházejícím případě projít od nejmenšího. Zde však bude drobný rozdíl v tom, že pokud ve spojovém seznamu ještě položka pro další exponent není, tak ji založíme. V tomto případě se dostáváme na časovou složitost $\mathcal{O}(N \log N)$ (nejdéle trvá úvodní setřídění) a paměťovou $\mathcal{O}(N)$.

Poznámka: V ukázkovém programu je použit navíc trik, díky kterému se bez spojáku nakonec obejdeme úplně. Stačí si při průchodu jen pamatovat, kolik stejných exponentů jsme v seřazeném poli už potkali. Kdo chcete, nahlédněte.

Program (C) – velké exponenty:
`http://ksp.mff.cuni.cz/viz/26-3-6-velke-exp.c`

Jirka Setnička

26-3-7 Sopečné pokrytí

Naším úkolem je pro mapu $R \times S$ najít posunutí mřížky s velikostí buňky $A \times B$ takové, že počet buněk obsahujících alespoň jeden sopečný kráter je minimální. K takovému výpočtu se nám bude hodit umět rychle zjišťovat, jestli nějaký konkrétní obdélník o velikosti $A \times B$ obsahuje sopečný kráter.

My si ukážeme, jak v čase $\mathcal{O}(RS)$ vyrobit strukturu, pomocí které v čase $O(1)$ dokážeme zjistit počet kráterů v libovolném podobdélníku, speciálně tedy i v podobdélnících $A \times B$.

Jedná se o takzvané *dvourozměrné prefixové součty*. Mohli jste je potkat například v kuchařce o základních algoritmech.[7] Pokud nepotkali, nevadí, myšlenku zde zopakujeme:

Pro každé políčko si předpočítáme, kolik kráterů obsahuje horní levý obdélník, který má pravý dolní roh právě v tomto políčku. Tedy políčko na pozici $[x, y]$ bude mít hodnotu počtu kráterů v obdélníku $([1, 1], [x, y])$. Tento výpočet si můžete sami rozmyslet, nebo se na něj podívat ve zdrojovém kódu – není to nic těžkého.

Dvourozměrné pole dvourozměrných prefixových součtů si označme P, pak počet kráterů v obdélníku $([a, b], [c, d])$ získáme jako:

$$K[a, b, c, d] = P[c, d] - P[c, b-1] - P[a-1, d] + P[a-1, b-1]$$

Když již máme vybudovanou pomocnou datovou strukturu, zkusíme každé možné posunutí větších buněk. Každé políčko se právě jednou stane nějakým pravým dolním rohem buňky (pro právě jedno posunutí).

V praktické realizaci tak pro každé políčko v mapě $R \times S$ zjistíme, jestli obdélník $A \times B$ mající pravý dolní roh v tomto políčku obsahuje kráter a pokud ano, tak k příslušnému posunutí mřížky přičteme jedničku. Tím jsme vlastně hotovi, už se jen stačí podívat, při jakém z $A \times B$ posunutí jsme potřebovali nejméně buněk mřížky.

Časová i paměťová složitost algoritmu je $\mathcal{O}(RS)$. Můžete se také podívat do vzorového kódu psaného v jazyce C++.

Program (C++):
`http://ksp.mff.cuni.cz/viz/26-3-7.cpp`

Karel Tesař

26-3-8 Zdivočelá počítadla

Úkol 1 – aritmetické instrukce

Začneme zkratkou `ADD` x, y, z pro uložení součtu $x + y$ do registru z. Nejprve zkopírujeme x do z a y do pomocného registru t. Poté budeme postupně dekrementovat t a inkrementovat z.

```
        MOV x,z
        MOV y,t
        JZ  t,Y
X:      INC z
        DEC t
        JNZ t,X
Y:
```

(Všimněte si, že naše zkratka nepotřebuje, aby registry x, y a z byly navzájem různé. O to se budeme snažit i u ostatních aritmetických operací. Jen musíme dodefinovat `MOV` x, x, aby neprovedl nic.)

[7] `http://ksp.mff.cuni.cz/viz/kucharky/zakladni-algoritmy`

S odčítáním `SUB` x, y, z si poradíme podobně. Nezapomeneme na případ, kdy $x < y$, na což máme podle zadání odpovědět nulou.

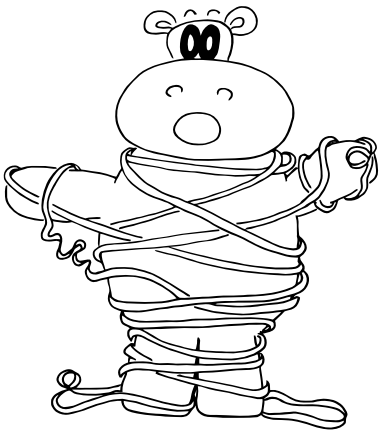
```
        MOV x,z
        JZ  y,Y
        MOV y,t
X:      DEC z
        DEC t
        JNZ t,X
Y:
```

Násobení `MUL` x, y, z pak definujeme jako opakované sčítání (s je další pomocný registr):

```
        CLR z
        JZ  y,Y
        MOV y,s
X:      ADD x,z,z
        DEC s
        JNZ s,X
Y:
```

Ještě se nám v následujících úkolech bude hodit dělení `DIV` x, y, p, q, které do p uloží celou část podílu x/y a do q zbytek po tomto dělení. Implementujeme jako opakované odčítání, jen pokaždé odečteme $y - 1$ a před odečtením zbývající jedničky zkontrolujeme, zda se dělenec už nevynuloval.

```
        MOV x,t
        CLR p
        MOV y,s
        DEC s       ; s = y-1
X:      MOV t,q
        SUB t,s,t
        JZ  t,Y
        DEC t
        INC p
        JMP X
Y:
```



Úkol 2 – tradiční závorky

Závorky na vstupu dostaneme zakódované do jednoho velkého čísla. To potřebujeme rozebrat na desítkové číslice, což se daleko snáz dělá od konce než od začátku: vydělením deseti odstraníme poslední číslici, zbytek nám řekne, jaká číslice to byla.

Na samotnou kontrolu uzávorkování použijeme jako obvykle počítadlo, ale jelikož vstup zpracováváme zprava doleva, bude tentokrát udávat, kolik závorek bylo uzavřeno, ale ještě ne otevřeno. Za každou `)` ho tedy zvýšíme o 1 a za každou `(` snížíme. Nesmí přitom nikdy klesnout pod nulu a na konci vstupu musí vyjít nulové.

S našimi aritmetickými instrukcemi je implementace hračka. V registru x očekáváme vstup, do y zapisujeme výstup, z nám bude počítat závorky, v d bude uložena konstanta 10 a r bude obsahovat právě odebranou číslici.

```
        CLR y         ; zatím špatně
        CLR z         ; počítadlo závorek
        CLR d
        INC d  (10x)  ; d=10
NEXT:   JZ  x,END
        DIV x,d,x,r   ; další závorka?
        DEC r
        JZ  r,OPEN
        INC z         ; zavírací
        JMP NEXT
OPEN:   JZ  z,DONE    ; otvírací
        DEC z
        JMP NEXT
END:    JNZ z,DONE    ; závěrečný test
        INC y         ; správně
DONE:
```

Úkol 3 – simulace Turingova stroje

Nejprve využijeme trik z 1. série, abychom daný Turingův stroj převedli na jednopáskový.

Nyní navrhneme, jak do registrů zakódovat konfiguraci stroje. Abecedu stroje očíslujeme od 0 do nějakého $A - 1$, přičemž 0 bude znamenat mezeru. Pásku rozdělíme v místě hlavy. Levou část uložíme do registru ℓ jako číslo v soustavě o základu A, číslice nejnižšího řádu bude odpovídat znaku těsně před hlavou (zde se hodí, že 0 je mezera, takže vlevo přirozeně vznikne nekonečně mnoho mezer). Znaky vpravo od hlavy zakódujeme obdobně do registru r, nejnižší řád bude odpovídat znaku bezprostředně za hlavou.

Zbývá nějak reprezentovat znak, na kterém právě stojí hlava, a aktuální stav řídicí jednotky stroje. To zakódujeme do pozice v počítadlovém programu, kde se zrovna nacházíme. Program bude tvořen *velikost abecedy* × *počet stavů* podobnými bloky, změna stavu nebo aktuálního znaku bude pouhý skok do jiného bloku.

Stačí tedy umět posouvat hlavu. Popišme posun doprava: Aktuální znak se přesune do levé části pásky, takže registr ℓ vynásobíme velikostí abecedy a přičteme aktuální znak. Naopak registr ℓ vydělíme velikostí abecedy a zbytek nám řekne, jaký znak se stal aktuálním. Posun doleva vyřešíme obdobně.

Úkol 4 – redukce počtu registrů

Kdyby nám stačilo dokázat, že stačí nějaký pevný počet registrů, můžeme k tomu použít předchozí úkol. Ze zadání víme, že každý počítadlový program lze přeložit na ekvivalentní Turingův stroj, v předchozím úkolu jsme se naučili převést ho zpět. Kombinací obou převodů získáme počítadlový program s pevným počtem registrů. Kdybychom promysleli detaily (zejména počty pracovních registrů v aritmetických instrukcích), dostali bychom se na něco jako 5 registrů. Ukážeme, že stačí méně.

Mějme program, který používá registry $r_1, \ldots, r_k$. Jejich stav dovedeme zakódovat do jediného čísla

$$r = p_1^{r_1} \cdot \ldots \cdot p_k^{r_k},$$

kde $p_1, \ldots, p_k$ jsou navzájem různá prvočísla. Z jednoznačnosti prvočíselného rozkladu plyne, že zakódovaný stav lze dekódovat jediným možným způsobem.

Instrukci `INC` r_i přeložíme na násobení registru r prvočíslem p_i. Ačkoliv bychom mohli použít zkratku `MUL`, raději si násobení naprogramujeme sami využívajíce toho, že p_i je konstanta. Bude nám stačit jediný pracovní registr t.

```
        CLR t
X:      INC t   (p_i-krát)
        DEC r
        JNZ r,X
Y:      INC r       ; přelijeme zpět do r
        DEC t
        JNZ t,Y
```

Podobně `DEC` r_i přeložíme na dělení registru r číslem p_i, ovšem musíme si dát pozor, abychom v případě, kdy r není dělitelné, vše vrátili do původního stavu.

```
        CLR t
        ; Opakovaně odčítáme p_i.
DIV:    JZ  r,OK
        DEC r
        JZ  r,R1
        DEC r
        JZ  r,R2
        (... dalších p_i-3 dvojic ...)
        DEC r
        INC t
        JMP DIV

        ; Nebylo to dělitelné,
        ; pozice v programu udává zbytek.
        (... p_i-3 inkrementů jako níže ...)
R2:     INC r
R1:     INC r
        ; Nakonec k r přičteme t * p_i.
T:      JZ  t,DONE
        INC r   (p_i-krát)
        DEC t
        JMP T

        ; Povedlo se, přelijeme zpět do r,
        ; které už je touto dobou nulové.
OK:     INC r
        DEC t
        JNZ t,OK
DONE:
```

Podmíněný skok `JNZ` vyřešíme obdobně: pokusíme se o `DEC`, pokud se povede, zvrátíme jeho účinek dalším `INC` a skočíme. Pakliže se nepovede, jen uvedeme registr r do původního stavu a pokračujeme v programu.

Vypadá to tedy, že každý program dokážeme upravit, aby mu postačily pouhé dva registry r a t. Jenže ouha: ještě musíme umět zakódovat vstup do našeho „exponenciálního“ kódování, a na konci programu zase dekódovat výstup. K tomu bohužel potřebujeme další registr.

Kódování vstupu bude probíhat tak, že na počátku položíme $r = 1$ a pak budeme dekrementovat vstupní registr a přitom inkrementovat jeho zakódovaný obraz v r. Podobně dekódování bude dekrementovat zakódovaný obraz výstupního registru a inkrementovat skutečný výstupní registr. Na obojí nám postačí tři registry.

Dodejme ještě, že je známo, že se dvěma registry takové kódování provést nelze. Důvod je prostý: nelze spočítat žádnou funkci, která roste exponenciálně. Dvojregistrový stroj tedy nemůže být univerzální. (Důkaz viz Rich Schroeppel: “A Two-counter Machine Cannot Calculate 2^N”, Massachusetts Institute of Technology, Artificial Intelligence Memo #257.)

Úkol 5 – minimální instrukční sada

Někteří řešitelé dokázali vymyslet jednoinstrukční sadu, ale pokaždé nějakým ošklivým trikem. Třeba instrukcí, jejíž součástí je konstanta, která instrukci řekne, jakou operaci má provést. Zde předvedeme také mírně podlé, ale snad o ždibec elegantnější řešení.

Naše instrukce se bude jmenovat `IDJNZ` x, y, p (increment, decrement and jump if not zero) a bude fungovat takto: Nejprve otestuje registr y na nulu. Pak inkrementuje registr x, načež dekrementuje registr y (pokud by vzniklo záporné číslo, zapíše nulu). Nakonec skočí na adresu p, pokud na začátku byl registr y nenulový. V opačném případě nikam neskáče.

`INC` x zapíšeme jako `IDJNZ` x, t, p, kde t je nějaký pracovní registr a p adresa těsně za instrukcí.

`DEC` x přeložíme analogicky na `IDJNZ` t, x, p.

`JNZ` x, p upravíme na `IDJNZ` x, x, p.

Martin „Medvěd“ Mareš

Výsledková listina třetí série dvacátého šestého ročníku KSP

	řešitel	*škola*	*ročník*	*sérií*	*3-1*	*3-2*	*3-3*	*3-4*	*3-5*	*3-6*	*3-7*	*3-8*	*série*	*celkem*
0.					11	12	9	10	11	8	12	15	61,0	174,0
1.	Martin Raszyk	G_Karvina	4	18	11		9	10	11	8	12	14	56,9	161,7
2.	Jan Špaček	G_Wicht	3	3	11		9	10	4	8	10	13	55,7	155,2
3.	Václav Rozhoň	GJirsíkaČB	3	4	11			9		7,8	12	13,5	54,9	150,8
4.	Marek Černý	G_Chrudim	3	3	11	11			11	7,6	12	14	60,4	148,0
5.	Matej Lieskovský	GOmskPha	4	13	11		9		11	7,9		12,5	50,8	142,0
6.	Michal Korbela	GJJesen	4	3	11	7,5			11	8	12		52,1	124,7
7.	Michal Punčochář	GJírovcČB	4	13	11	12	9	10	10,5		12		55,4	122,3
8.	Jakub Svoboda	GKomHavíř	4	8	9		9		11	8		10	48,9	119,4
9.	Richard Hladík	GOAMarLaz	1	8	10				11	7	12	5	47,0	112,8
10.	Aneta Šťastná	GOmskPha	4	9	8		9		7	7,7		6	40,1	105,2
11.	Jan-Sebastian Fabík	GJarošeBO	4	11	11		9	10			12	8	50,1	98,6
12.	Jakub Zárybnický	GTomkovaOL	3	3	5	6	0	2	2	7		3,5	35,3	91,1
13.	Filip Bialas	GOpatovPHA	1	3			7		6	7,7	10		36,1	86,5
14.	Antonín Češík	SPSE_Pard	4	3			6			7,7		11	29,2	78,4
15.	Jakub Maroušek	G_Písek	4	7	11		9	2	7	7			38,7	72,2
16.	Anna Steinhauserová	GDačice	4	3		6	5						16,1	71,7
17.	Lucie Studená	GKepleraPH	4	2	3	5	5		7	7,5			37,9	70,6
18.	Dorian Řehák	GCoubTábor	3	3					6	3		3	19,6	67,5
19.	Václav Volhejn	GKepleraPH	1	8	11				4			2	18,9	65,3
20.	Štěpán Hojdar	GJírovcČB	4	7									0,0	60,1
21.	Jan Pokorný	G_Bučovice	2	3									0,0	59,1
22.	Štěpán Trčka	GSlavičín	3	9									0,0	54,3
23.	Jonatan Matějka	SŠP_ČB	4	17	11								11,0	50,9
24.	Jan Knížek	G_Strakon	3	11									0,0	42,9
25.	Adam Španěl	ArcibisGPH	2	2									0,0	32,0
26.	Ondřej Hübsch	GArabskáPH	4	21							12		12,0	30,0
27.	Dominik Roháček	SPŠLegioJI	4	3									0,0	27,0
28.	Dalimil Hájek	GKepleraPH	3	13							10		9,6	26,5
29.	Antonín Teichmann	GJeronýmLI	4	2								4	7,4	22,6
30.	Jan Pavlovský	GJiM	4	1									0,0	21,3
31.	Marek Dobranský	GHorMichal	4	6									0,0	20,3
32.	Aneta K. Lesna	GZborovPH	1	1									0,0	16,8
33.	Michal Hloušek	GNadŠtolPH	1	1									0,0	16,3
34.	Petro Kostyuk	GEBenešeKL	4	2									0,0	12,1
35.	Přemysl Šťastný	GZamberk	0	2						5		1,5	10,0	10,0
36.	Radovan Švarc	G_ČTřebová	3	3									0,0	8,0
37.	Tadeas Friedrich	GOhradníPH	4	2									0,0	6,3
38.	Jan Horešovský	GMěl	4	2									0,0	6,2
39.	Michal Martinek	GHavPodl	3	1									0,0	6,0
40.	Marek Židek	GTomkovaOL	4	1									0,0	4,0
41.	Ladislav Tlapák	G_Břeclav	−1	1									0,0	2,5