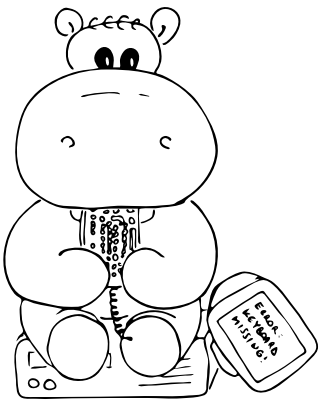
Milí řešitelé a řešitelky!



Spadané listí hraje všemi barvami, někdo hledá kaštany a vás si našel druhý leták 26. ročníku KSP. V něm vám přinášíme další sadu úložek a pokračování seriálu o výpočetních modelech.

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 % bodů. Slibujeme, že letos půjde získat maximálně 300 bodů, takže hranice pro úspěšné řešitele je 150.

Pro začátečníky chceme vyzdvihnout informaci, že v každé sérii jsou dvě lehčí úlohy určené právě jim. A časem pro ně možná budeme mít ještě víc. ☺

Připomínáme, že každému řešiteli, který v tomto ročníku z každé série dostane alespoň 5 bodů, darujeme KSP propisku, blok a tužku. Dále se na vědomost dává, že každému, kdo v této sérii **vyřeší alespoň tři libovolné úlohy na plný počet bodů, pošleme čokoládu**.

Všechny řešitele, stejně jako kohokoli dalšího se zájmem o studium na MFF UK, si navíc dovolujeme pozvat na **Den otevřených dveří MFF UK**, který proběhne ve **čtvrtek 28. listopadu**. Více informací naleznete na adrese *http://www.mff.cuni.cz/verejnost/dod*.

Termín odevzdání druhé série je stanoven na **pondělí 9. prosince v 8:00 SEČ**. CodExová úloha má termín o den posunutý, opravuje ji totiž automat – odevzdejte ji do 10. prosince, 8:00 SEČ.

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

Před tím ale vyplňte přihlášku na *http://ksp.mff.cuni.cz/* (a to i tehdy, když jste se KSPčka 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*.

Druhá série dvacátého šestého ročníku KSP

V předchozí sérii jste mohli sledovat nouzové přistání vesmírné lodi Freya. *Jacob, který přistání přežil, nalezl kus od lodi zdobený oštěp, známku inteligentního života. Vyrazil ho hledat. Najednou ho ale vyrušilo zapraskání za jeho zády . . .*

⋆⋆⋆

Mezi listy zlatě zasvítily dva velké kruhy. Byly to oči nějakého zvířete. K očím patřil i čumák a překvapivě malá hlava. A tělo a čtyři dlouhé nohy. Jacobovi vyskočilo srdce někam do krku.

Nedělal si iluze o tom, že zvíře přišlo, aby se stalo jeho průvodcem po neznámé planetě. Když jako jediný přežijete nouzové přistání vesmírné lodi, je to samo o sobě dost šílené a na další zázrak si obvykle musíte zase chvíli počkat.

Zvíře zatím Jacoba pozorovalo. Možná pozorovalo i kus jeho okolí. Rozhodně se zdálo, že pozorně naslouchá. Aspoň pokud to, co Jacob považoval za uši, skutečně byly uši. Něco tu nesedělo, ale strach nedovolil jeho mozku zabývat se nějakými nepodstatnými nesrovnalostmi. Místo toho se dal na útěk.

Běžel, co mu síly stačily, a litoval, že se nedá vylézt na okolní stromy. Netušil, jestli zvíře zůstalo stát na místě, nebo jestli běží za ním a on ho jen pro svůj vlastní dusot neslyší. Netroufal si ohlédnout se, a navíc, dokud zvíře neviděl, mohl doufat, že tam není.

Najednou vzduch prořízlo nějaké zasvištění. Ani tehdy se Jacob neohlédl, jen si uvědomoval, že ho nic neskolilo, že může běžet dál.

Zastavil se až za hodnou dobu, kdy začal opadávat jeho strach, a s tím mu začaly docházet síly. Našel odvahu ohlédnout se. Po zvířeti nebylo ani vidu. Jacob si nebyl úplně jistý, že trefí zpět, ale pro tuhle chvíli se rozhodl doufat, že cestu najde.

Začal se zase soustředit na svůj původní plán, totiž hledat místní inteligentní život. Pozorně se rozhlédl po okolí. A zděsil se.

Teprve teď si všiml, že kus od něj jsou volně natažené dva provazy, které se do sebe navíc poněkud zamotaly. Byl div, že se o ně při svém úprku nepřerazil. Jacob se k nim sehnul, aby je mohl prozkoumat blíž.

26-2-1 Zamotané provazy — 7 bodů

V pralese jsou volně natažené dva provazy, které se do sebe zamotaly. Provazy jsou pevně uvázané na dva stromy, první provaz níž a druhý výš. Žádný z nich se nikde nevrací. Může to tedy vypadat třeba takto:

O každém překřížení víme, který z provazů je vpředu. Chceme zjistit, jestli je možné provazy rozmotat bez toho, aby bylo nutné některý z nich odvazovat.

Program dostane na vstupu číslo N udávající počet křížení a N čísel 1 nebo 2 udávajících, který provaz je vpředu (na stromech je provaz 2 vždy uvázaný nad provazem 1). Odpovědět má `ano`, nebo `ne`, podle toho, zda je provazy možné rozmotat.

Příklad (odpovídá obrázku):

```
4
1 2 2 1
```

Odpovědí je `ano`, jelikož provazy rozmotat jdou.

Při zkoumání provazů Jacob najednou zahlédl mezi stromy jakýsi barevný záblesk. Pouhým pohledem se mu nedařilo zjistit víc, a tak se s patřičnou obezřetností vydal tím směrem. Jak se přibližoval, měnily se občasné záblesky v barevnou plochu prosvítající mezi stromy.

Konečně se Jacob dostal až k ní. Ocitl se na něčem, co snad mohlo připomínat mýtinku, ovšem po stromech ani trávě tu nebyly vůbec žádné stopy. Teď už Jacob viděl, že

trojúhelníkovou barevnou plochu tvoří zvláštní hexagonální kameny, některé červené, některé do žluta.

Zdálo se, že je kdosi naskládal do mnoha vrstev, kterými pečlivě vyplnil hlubokou jámu; několik dalších kamenů se válelo v nejbližším okolí. V jednom z nich byl vyrytý podivný nápis:

X̌eȓa trȷ́ineźr ĝaetȯ všpogiń
X̌eȓa giĝesȯr ĝaetȯ mànȷ́ine
Íhgišpop̌ kiĝes trǔćep̌ náp̌
matrȷ́inep̌ vàs heriřep̌ sdi náp̌

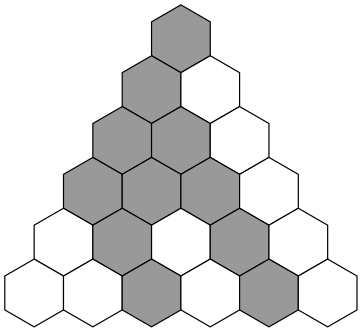
Jacob si všiml, že jedna z barev spojuje všechny tři strany trojúhelníku, a napadlo ho, jestlipak je to tak i ve vrstvách, které nevidí.

26-2-2 Barevný trojúhelník — 8 bodů

Máme rovnostranný trojúhelník o hraně N tvořený hexagonálními kameny dvou barev.

Dokažte, že v trojúhelníku existuje jednobarevná souvislá plocha, ve které se nachází alespoň jeden kámen z každé strany trojúhelníku. Kameny ve vrcholech patří do dvou stran zároveň.

Příklad takového trojúhelníku si můžete prohlédnout na obrázku. Barvou, která spojuje všechny tři strany, je zde šedá.



Protože se začalo připozdívat, rozhodl se Jacob brzy pokračovat dál. Kamenný trojúhelník mu nyní sloužil jako dobrý orientační bod, a on se vypravil směrem dál od jednoho z jeho vrcholů.

Po nějaké době došel k místu, kde se prales najednou začal rozestupovat. Vzniklý výhled odhalil krom jiného to, že tato část pralesa je mírně vyvýšená nad okolím – zdola musel být nahoru opravdu impozantní pohled.

O kus dál uviděl Jacob řeku. To ho potěšilo, neboť pokud se tu dá spoléhat na podobné věci jako na Zemi, mohl by ji sledovat a dojít po jejím proudu k nějaké civilizaci.

Zdálo se ovšem, že terén je všelijaký a některá místa by mohla cestu hodně zpomalit. Musel si ji proto dobře rozmyslet.

26-2-3 Plánování cesty — 10 bodů

Jacob se chce co nejrychleji dostat k řece. Terén, přes který musí projít, si lze představit jako čtvercovou síť o rozměrech $R \times S$. V některých oblastech je tento terén dost nepříjemný, a tak přes některá políčka trvá průchod delší dobu, některá jsou dokonce zcela neprůchozí.

Průchod přes oblast ležící na souřadnicích i, j trvá $t_{i,j}$. Přecházet mezi políčky je možné pouze vodorovně nebo svisle (šikmo ne).

Vaším úkolem je najít nejrychlejší cestu.

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.

Jacob cestu naplánoval, jak nejlépe uměl. Dál byl ovšem rozhodnutý vyrazit až ráno. Přeci jen není nejrozumnější pohybovat se unavený někde, odkud můžete spadnout. Teď si raději opatřil provizorní přístřešek. Chvíli se ještě podivoval nad tím, jak jasná je noc, a pak se uložil ke spánku.

⋆⋆⋆

Ráno se Jacob vzbudil brzy. Pobalil si svých několik věcí a odhodlaně vyrazil na cestu. Terén tu byl skutečně všelijaký, místy příjemně ušlapaná hlína, místy skoro bažiny, mezitím ledacos jiného, ovšem připravená trasa se vcelku osvědčila. Mohlo být krátce po poledni, když Jacob dorazil k řece.

V jednom kameni u břehu byl opět vyrytý podivný nápis:

Íhgišpop̌ kiĝes trǔćep̌ náp̌
lugi íhgišpop̌ kiĝes ńeríp̌ athĝep̌ sdi náp̌
Reho xeȓ ná sjo ĝesŕit
ASN sjo trȷ́ine

Ještě víc ovšem Jacoba zaujaly podivné kostky na protějším břehu. Vypadaly trochu jako dětské dřevěné kostičky, jen o mnoho větší. A zdálo se, že z nich někdo chce podobně jako z kostiček pro děti postavit velkou věž.

26-2-4 Stavba věže — 10 bodů

Někdo se z K kostek snaží postavit věž. Všechny kostky jsou stejně velké, ale každá kostka má svoji váhu w_i a nosnost ℓ_i. Nosnost ℓ_i znamená, že i-tá kostka unese na ní stojící kostky o maximální celkové váze ℓ_i, jinak se zbortí a věž spadne.

Zajímá nás, jestli lze postavit věž ze všech K kostek. Věž stavíme tak, že přímo na sebe pokládáme jednotlivé kostky (můžete si představit i to, že stavíme komín).

Lehčí varianta (za 3 body): Vyřešte úlohu za předpokladu, že všechny kostky mají váhu $w = 1$.

Stavba, ať už měla být v budoucnosti čímkoli, byla zrovna opuštěná. Jacob se tedy vydal dál po proudu řeky. Cestou občas potkal nějaký ten nápis na kameni, stále se mu v nich ale nedařilo odhalit žádný smysl.

Po několika nápisech a mnoha krocích se před Jacobem najednou objevila malá věžička. Nebo alespoň malá na místní poměry. Jak už Jacob očekával, stál před ní kámen s nápisem:

Ĝesŕit íhgišpop̌ kiĝes ńeríp̌ athĝep̌ sdi náp̌
ASN krees

Věžička neměla okna, měla ovšem dveře. Po troše váhání si Jacob dodal odvahy a zkusil dveře otevřít. Šlo to překvapivě lehce. Jako první zaznamenal směs různého hučení a bzučení. Pak teprve s jistou úlevou zjistil, že vevnitř nikdo není.

Původcem podivných zvuků se ukázaly být jakési krabičky. Byly na sobě různě zavěšené, celá konstrukce byla navíc uchycená ve věžičce. Jacob se ale nemohl ubránit dojmu, že konstrukce už nemůže dlouho vydržet. Ačkoli si nebyl jistý, jestli je to dobrý nápad, rozhodl se jí pomoci tím, že ji vyváží.

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

26-2-5 Vyvažování — 12 bodů

Očíslované krabičky jsou zavěšené ve věžičce, hrozí ale, že celá konstrukce spadne. Chceme ji proto vyvážit. Podle piktogramů na zdech víme, že je potřeba zachovat určité pořadí krabiček na konstrukci. Celou úlohu si tak můžeme představit jako vyvažování binárního vyhledávacího stromu.

Na vstupu máme obecný binární vyhledávací strom (jeho definici naleznete v kuchařce) a chceme z něj udělat dokonale vyvážený binární vyhledávací strom. Pro každý vrchol výsledného stromu tedy musí platit, že rozdíl velikostí jeho levého a pravého podstromu je nejvýše 1.

Konstrukce už drží jen silou vůle, takže požadujeme, aby váš algoritmus pracoval s lineární časovou složitostí. Počet bodů, které dostanete, bude záviset na prostorové (paměťové) složitosti vašeho řešení. Nějaké body dostanete i za lineární, na plný počet ale potřebujete konstantní složitost.

Po práci si Jacob udělal pauzu na svačinu. Bylo pomalu vhodné začít myslet na návrat, ale on chtěl ještě pokračovat ve svém pátrání. Usoudil, že alespoň do večera může jít dál, a pak se uvidí.

S plnějším žaludkem se pak Jacob vypravil opět po proudu řeky. Tentokrát ovšem potkal něco zajímavého mnohem dřív. Došel totiž k něčemu, co se snad dalo považovat za brod.

Tradičně tu potkal vyrytý nápis, tentokrát dost složitý. Po chvilce jeho zkoumání Jacob pojal podezření, že znaky, které už nějakou dobu potkával na kamenech u břehu, jsou zkratkou nějakého názvu. Studoval další slova na kameni a přemýšlel, která z nich by také mohla být názvy a mít nějakou svoji zkratku.

26-2-6 Zkratky míst — 12 bodů

Máme podezření, že pro názvy míst se používají třípísmenné zkratky, a máme také seznam slov, o kterých si myslíme, že by mohly být názvem nějakého místa.

Zkratka má první písmeno vždy stejné, jako je první písmeno názvu daného místa, a zbylá dvě písmena jsou libovolná dvě písmena z názvu místa ve správném pořadí. Název *Praha* tak lze zkrátit např. na *PRH* nebo *PHA*, ale už ne na *PHR* nebo *RHA*.

Vaším úkolem je zjistit, pro která všechna slova bychom skutečně uměli takovou zkratku najít. Zkratky míst musí být navzájem jedinečné, chceme jich ale najít co nejvíce. Je-li možných řešení se stejným počtem nalezených zkratek víc, vypište libovolné z nich.

Příklad:

Vstup:	Výstup:	
Pra	Pra	PRA
Praga	Praga	PAA
Prak	Prak	PAK
Prk	Prk	PRK
Prkg	Prkg	PRG
Pak	Prague	PGU
Prague		

Jacobovi se povedlo překonat řeku a dostat se na druhý břeh. Před ním začínalo něco, co silně připomínalo cestičku. Zhluboka se nadechl a pak po ní vyrazil.

Cestička chvíli vedla skoro podél řeky, pak se od ní ovšem prudce odklonila a zavedla Jacoba za hradbu stromů. Nebyly to stejné stromy jako v pralese, ale podobně jako ony byly vysokánské a měly o trochu jinou barvu, než by člověk čekal na Zemi.

Jacob se rozhlédl kolem sebe a bezděky zatajil dech. Našel, co hledal! Byl tu inteligentní život, byl tady, přímo před ním.

Tvorové byli vlastně podobní lidem. Byli vyšší, Jacob by vedle nich vypadal jako malé dítě. Podobně jako zvíře minulý den měli překvapivě malou hlavu a naopak nezvykle velké oči. Barvou kůže připomínali spíše pozemšťany, kterým zrovna není dobře od žaludku. Ale bez nejmenších pochyb to byli představitelé místního inteligentního života.

Skupinka tvorů se zrovna motala kolem obrovského kmene stromu, který zřejmě pocházel z pralesa.

26-2-7 Čištění kmene — 13 bodů

Humanoidní tvorové čistí kmen stromu. Celkem se práce účastní T tvorů. Na kmeni je M míst, která je potřeba očistit. Každé takové místo je na nějaké pozici m_i (to může být třeba vzdálenost od konkrétního konce kmene; pozice bude vždy celočíselná).

Na začátku stojí j-tý tvor na pozici t_j. Práce tvorů probíhá v jednotlivých krocích: v každém kroku se každý tvor (nezávisle na ostatních) může přesunout o jednu pozici vlevo, vpravo, nebo může zůstat stát na místě.

Očištění části kmene, nad kterou tvor stojí, je bleskové (proběhne ve stejném kroku, kdy tvor k této části kmene dojde). Speciální výjimkou je počáteční pozice každého tvora. Ta je očištěná ještě před tím, než tvor udělá první krok.

Zajímá nám nejmenší počet kroků potřebný k očištění celého kmene.

Při řešení můžete předpokládat $1 \leq M, T \leq 100\,000$ a $1 \leq m_i, t_j \leq 1\,000\,000\,000$.

Příklad: Mají-li se zkontrolovat místa $2, 5, 6$ a tvorové stojí na pozicích $3, 5$, zvládnou kmen očistit na 1 krok. Kdyby stáli na $3, 4$, budou kroky potřebovat 2.

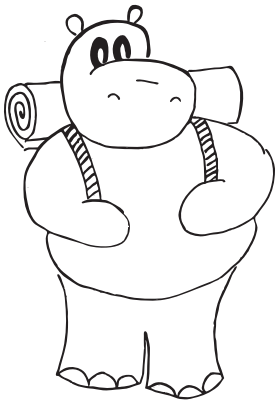
Když se před ním tvorové vynořili, nebyl si Jacob najednou vůbec jistý, co by vlastně měl udělat. Pozoroval skupinku, všiml si také kdesi za nimi ďolíků v zemi, všiml si, že v jednom z nich zmizel stejný tvor.

Najednou se ozval výkřik. Jacob se polekaně rozhlédl. Někdo si ho zřejmě všiml a upozornil na něj ostatní. V té chvíli se na něj upřely pohledy všech tvorů.

Pokračování příště...

První Jacobovy kroky po neznámé planetě s vámi sledovala

Karolína „Karryanna“ Burešová



26-2-8 Továrna na přepisování — 14 bodů

V prvním díle letošního seriálu jste se mohli seznámit s Turingovými stroji coby zajímavým teoretickým modelem počítače. Dnes v naší zoo výpočetních modelů popojdeme k sousednímu výběhu, kde se pasou *přepisovací pravidla.* Ne nadarmo jsou hned vedle – časem uvidíme, že si spolu tahle dvě zvířátka náramně rozumějí.

Abeceda a vstup s výstupem

Jak už název napovídá, přepisovací pravidla pracují nad nějakým zadaným řetězcem znaků (říkejme mu *vstup*) a postupně ho nějak přepisují, až dostanou nějaký další řetězec znaků (*výstup*). Než začneme mluvit o samotných pravidlech, pojďme tyto řetězce pořádně definovat.

Za *abecedu* budeme označovat konečnou množinu všech *znaků,* které se mohou vyskytnout v řetězci. *Vstup* a *výstup* jsou pak vždy nějaké konečné řetězce znaků z této abecedy.

Mimo to zavedeme dva metaznaky, kterými budeme označovat okraje řetězce: `^` na začátku a `$` na konci řetězce. Tyto znaky nebudou součástí abecedy, a tedy se nebudou vyskytovat nikde jinde než právě na okrajích řetězce.[2]

Abecedou mohou být například čísla `0-9`, písmena `a-z`, nebo třeba {♡, ♠, ♣, ♢}. Jednotlivé úlohy pak mohou navíc určit, že ve vstupu nebo výstupu se smí vyskytovat jen některé znaky abecedy.

Přepisovací pravidla

Každé *přepisovací pravidlo* má svou levou a pravou část: levé části budeme říkat *vzor* a pravé *přepis*. Zapisovat ho budeme takto:

$$\textit{vzor} \to \textit{přepis}$$

Ve vzoru se mohou kromě znaků abecedy vyskytovat i metaznaky začátku a konce řetězce (pak se vzor váže k nějakému z okrajů řetězce), v přepisu se už mohou vyskytovat jen znaky abecedy.

Aplikace konkrétního pravidla na nějaký řetězec pak vypadá následovně:

- Najde se první (nejlevější) výskyt vzoru v řetězci.
- Tento výskyt se vymaže a na jeho místo se vloží přepis.

Pravá strana (přepis) může být i prázdná. Takové pravidlo lze například využít k vymazání části vstupu. Vzor naopak prázdný být nemůže, vždy musí obsahovat alespoň jeden znak nebo metaznak.

Uvažme například následující pravidla:

$$\texttt{a} \to \texttt{b}$$
$$\texttt{a\$} \to \texttt{b}$$
$$\texttt{\^{}aa} \to \texttt{ccc}$$

Každé samostatně aplikujeme na vstup `baa`. První pravidlo ho přepíše na `bba`, druhé na `bab` a třetí pravidlo nic nepřepíše (takovýto vzor ve vstupu neexistuje, a tedy se nic neprovede).

Samostatnou aplikací pravidel ovšem dost silný výpočetní model nedostaneme. Na to bude potřeba poskládat více pravidel dohromady.

Přepisovací programy

Přepisovacím programem nazveme nějaký uspořádaný soubor přepisovacích pravidel, čili několik pravidel seřazených za sebe.

Výpočet přepisovacího programu probíhá v *krocích*: první krok přepisuje vstup programu, výstup prvního kroku se stane vstupem druhého, a tak stále dokola. Výpočet končí ve chvíli, kdy již mezi pravidly neexistuje žádné, jehož vzor by se vyskytoval ve vstupu. To se samozřejmě nemusí stát – jistě snadno vymyslíte přepisovací program, který se nikdy nezastaví.[3]

Výpočet každého kroku by se dal formálně popsat takto:

- Najde se první pravidlo, jehož vzor se vyskytuje někde ve vstupu.
- Toto pravidlo se provede (najde se první výskyt vzoru ve vstupu a na jeho místo se vloží přepis).
- Výstup tohoto pravidla se použije jako vstup dalšího kroku. Výpočet dalšího kroku začíná opět od prvního pravidla a od začátku řetězce.

Zbývá zavést nějakou míru efektivity – analogii časové složitosti normálních programů. Nám pro tyto účely bude sloužit počet provedených kroků přepisovacího programu. (Na zamyšlení: co by odpovídalo prostorové složitosti?)

Příklady

Pojďme vyzkoušet, co přepisovací programy dovedou.

První příklad bude jednoduchý. Představte si, že máme zadanou posloupnost nul a jedniček (třeba `01101`) a chceme ji setřídit. Jinými slovy chceme přesunout všechny nuly nalevo od jedniček.

To půjde překvapivě snadno. Sestrojíme program obsahující jediné pravidlo:

$$\texttt{10} \to \texttt{01}$$

Na našem ukázkovém vstupu bude výpočet probíhat takto: `01101`, `01011`, `00111`.

Povšimněme si, že dokud posloupnost není setříděná, existuje v ní aspoň jedna po sobě jdoucí dvojice `10`, takže program běží dál.

Doběhne někdy? Sledujme *inverze* v posloupnosti: tak budeme říkat dvojicím (ne nutně sousedních) čísel, která jsou ve špatném pořadí. Naše ukázková posloupnost obsahuje 2 takové dvojice. Každé použití pravidla sníží počet inverzí právě o jedna, takže se program po konečně mnoha krocích musí zastavit.

Jak dlouho náš program poběží? Ve vstupu délky n může být nejvýše $(n/2)^2$ inverzí (to odpovídá situaci, kdy na vstupu máme $n/2$ jedniček a za nimi $n/2$ nul) a program je odstraňuje po jedné, takže provede řádově n^2 kroků. (Mimochodem, pokud budeme provádět chytřejší operace než pouhé prohazování, je možné dosáhnout i lepší efektivity. Zkuste vymyslet, jak.)

Druhý příklad už bude záludnější. Možná si vzpomínáte na kontrolu správnosti uzávorkování z minulé série. Pojďme si ukázat, jak snadno se dá vyřešit pomocí přepisovacích pravidel.

[2] Podobnost s regulárními výrazy používanými v praxi vůbec není náhodná. Chcete vědět víc? Nahlédněte do seriálu 23. ročníku.

[3] Jednou z možností je třeba program skládající se z jediného pravidla `$` → `a`. Ten nikdy neskončí, protože vzor pravidla je obsažen v každém řetězci.

Na vstupu budeme mít posloupnost levých a pravých závorek a naším úkolem bude rozhodnout, jestli tvoří správné uzávorkování, nebo netvoří. Podle toho vrátíme na výstupu buď `ANO`, nebo `NE`. Pro připomenutí, správné uzávorkování je takové, ve kterém jsou závorky správně spárované a páry se nekříží.

Idea našeho přepisovacího programu bude takováto: Všimneme si, že se v každé neprázdné správně uzávorkované posloupnosti vyskytuje po sobě jdoucí dvojice `()`. Tu můžeme odstranit, čímž získáme další správně uzávorkovanou posloupnost, a tak dále, až nakonec dospějeme k prázdné posloupnosti. Funguje to i naopak: přidáním `()` do jakéhokoliv správného uzávorkování nelze získat nesprávné, takže se nemůže stát, že bychom na prázdný řetězec zredukovali nějaké nesprávné uzávorkování.

Program vypadá následovně:

$$\texttt{X\$} \to \texttt{NE}$$
$$\texttt{X(} \to \texttt{X}$$
$$\texttt{X)} \to \texttt{X}$$
$$\texttt{()} \to$$
$$\texttt{\^{}\$} \to \texttt{ANO}$$
$$\texttt{\^{}(} \to \texttt{X}$$
$$\texttt{\^{})} \to \texttt{X}$$

Dokud vše probíhá tak, jak má, čtvrté pravidlo umazává závorky. Pokud umaže všechny závorky, páté pravidlo vypíše `ANO`. Pokud ale již nebude možné umazat žádnou dvojici závorek, nastoupí jedno z posledních dvou pravidel a na scénu přijde `X`. To pak za pomoci prvních tří pravidel vymaže zbytek vstupu, vypíše `NE` a program se zastaví.

Rozmyslete si, proč namísto posledních dvou pravidel nemůžeme použít $\texttt{\^{}} \to \texttt{X}$.[4]

Úlohy

Vymyslete přepisovací programy na následující problémy. U všech programů odhadněte efektivitu (v počtu provedených kroků) a pokuste se o co nejlepší. Součástí řešení by měl být slovní popis a alespoň náznak zápisu použitých přepisovacích pravidel.

Úkol 1 [2b]: Vstup je tvořený posloupností čísel `0` až `9`. Vaším úkolem je zanechat na výstupu `ANO`, pokud je posloupnost čísel neklesající, a `NE` v opačném případě.

Úkol 2 [3b]: Na vstupu je posloupnost znaků `*`. Spočítejte, kolik jich je, a výsledek zanechte na výstupu jako číslo ve dvojkové soustavě.

Úkol 3 [5b]: Na vstupu jsou dvě čísla ve dvojkové soustavě oddělená křížkem `#`. Na výstupu nechť se objeví to větší z nich. Pozor, zápisy čísel nemusí být stejně dlouhé. Případ, kdy jsou si čísla rovna, nemusíte uvažovat.

Převod na Turingův stroj a naopak

Jak už jsme zmínili v úvodu, přepisovací pravidla a Turingovy stroje spolu úzce souvisí. Ukážeme, jak převést jakýkoli přepisovací program na Turingův stroj.

Nejdříve se zamyslíme, jak převést jedno přepisovací pravidlo do řeči Turingova stroje. Začneme s hlavou na levém konci pásky a budeme se postupně pokoušet najít výskyt vzoru (vyzkoušíme všechny začátky a vždy porovnáme se vzorem; stav stroje bude říkat, kolikátý znak vzoru porovnáváme).

Ve chvíli, kdy nalezneme první výskyt, přepneme se do dalšího stavu a pásku přepíšeme odpovídajícím přepisem (a případně odsuneme či přisuneme zbytek znaků na pásce, pokud přepis bude mít jinou délku než vzor). A nakonec, po dokončení přepisování, vrátíme hlavu opět na začátek pásky (a stejně tak v případě, že se nám nepovede najít žádný výskyt vzoru).

Tím jsme úspěšně naučili Turingův stroj zpracovat jedno pravidlo. K překladu celého přepisovacího programu už zbývá jen krůček. Každé pravidlo v programu bude mít svou vlastní sadu stavů (sady nechť jsou třeba očíslované). Po úspěšném provedení pravidla a návratu hlavy zpět na začátek pásky se přesuneme do první sady; po neúspěšném hledání vzoru se přesuneme do následující sady (zkusíme aplikovat další pravidlo v pořadí). Pokud budeme neúspěšní i u posledního pravidla, ukončíme výpočet.

Právě jsme dokázali, že každý přepisovací program lze simulovat Turingovým strojem. Pokud navíc ukážeme, že každý Turingův stroj lze převést na přepisovací program, bude jasné, že oba výpočetní modely jsou stejně silné (co jde spočítat v jednom, jde i ve druhém a opačně). To už ale bude na vás:

Úkol 4 [4b]: Dokažte, že pro každý jednopáskový Turingův stroj existuje přepisovací program, který počítá totéž. Přesněji řečeno, pokud se pro daný vstup Turingův stroj zastaví, přepisování se také zastaví a vydá stejný výsledek. Pokud se stroj nezastaví, přepisování se také nezastaví. Rozmyslete si, v jakém vztahu je časová složitost stroje a pravidel. Svůj přístup demonstrujte na stroji z příkladu v 1. sérii (vyvážená posloupnost).

Jirka Setnička & Martin „Medvěd“ Mareš

[4] Správná odpověď je, že pak by se nám program nikdy nezastavil, protože vzor takového pravidla by byl nalezen vždy.

V kuchařce první série jsme probrali základní způsoby ukládání dat v počítači, tzv. datové struktury, a také často používané programátorské techniky. Konkrétně jsme se naučili udržovat čísla nebo jiné objekty v poli, ve spojovém seznamu, v grafu nebo ve stromu. Ukázali jsme si rekurzi a její využití v backtrackingu (prostém zkoušení všech možností). Dále jsme nakouli pod pokličku dalším technikám: rozděl a panuj, dynamickému programování, hladovým algoritmům a pár dalším.

Nyní se podíváme podrobněji na binární vyhledávání, které bylo rovněž minule představeno, a pokusíme se ho vylepšit, abychom mohli průběžně měnit data, v nichž vyhledáváme. Zdá-li se vám to na jednu kuchařku málo, zvlášť v porovnání s tou minulou, věřte, že problém není jednoduchý, ale zajímavý a řešení jsou navíc v praxi často používána.

Nejprve však zopakujme binární vyhledávání.

Binární vyhledávání

Stejně jako minule máme obrovské pole setříděných záznamů, třeba identifikačních čísel studentů nejmenované univerzity (záznamy však nemusí být čísla, stačí, když jsou navzájem porovnatelné). Naším úkolem je najít záznam z v poli s N záznamy $x_1 < x_2 < \ldots < x_N$.

Při použití binárního vyhledávání neboli půlení intervalu se podíváme na prostřední záznam x_m a porovnáme s ním naše z. Pokud $z < x_m$, víme, že se z nemůže vyskytovat „napravo“ od x_m, protože tam jsou všechny záznamy větší než x_m a tím spíše než z. Analogicky pokud $z > x_m$, nemůže se z vyskytovat v první polovině pole. V obou případech nám zbude jedna polovina a v ní budeme pokračovat stejným způsobem. Tak budeme postupně půlit interval, ve kterém se z může nacházet, až buďto z najdeme nebo vyloučíme všechny prvky, kde by mohlo být.

Tento algoritmus můžeme naprogramovat buďto rekurzivně nebo pomocí cyklu, v němž si budeme udržovat interval $\langle l, r\rangle$, ve kterém se hledaný prvek ještě může nacházet. My si ukážeme v jazyce C přístup s cyklem:

```
int bin_najdi(int z) {
    int levy, pravy, median;
    // interval, ve kterém hledáme
    levy = 0; pravy = N;
    // dokud interval ještě není prázdný
    while (levy <= pravy) {
        median = (levy+pravy)/2;
        // hledaná hodnota je vlevo
        if (z < x[median])
            pravy = median-1;
        // je vpravo
        else if (z > x[median])
            levy = median+1;
        else // našli jsme ji
            return median;
    }
    return -1; // hledaná hodnota nebyla nikde
}
```

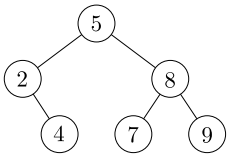
Samozřejmě bychom při vyhledávání záznamu mohli být ještě chytřejší. Víme-li třeba, že čísla jsou z rozsahu 1 až 1000 a dostaneme číslo 900, můžeme se napřed podívat do devíti desetin pole místo do poloviny. Obecně se tedy snažíme odhadovat, kde bude záznam v rámci pole podle jeho hodnoty. Tomuto přístupu se říká *interpolační vyhledávání* a v průměru je lepší než binární (průměrná časová složitost je $\mathcal{O}(\log\log N)$), byť v nejhorším případě je lineární.

Binární vyhledávání je velmi rychlé, pokud máme možnost si data předem setřídit. Jakmile ale potřebujeme za běhu programu přidávat a odebírat záznamy, se zlou se potážeme. Buďto budeme mít záznamy uložené v poli a pak nezbývá než při zatřiďování nového prvku ostatní „rozhrnout“, což může trvat až N kroků, anebo si je budeme udržovat v nějakém seznamu, do kterého dokážeme přidávat v konstantním čase, jenže pak pro změnu nebudeme při vyhledávání schopni najít tolik potřebnou polovinu.

Zkusme ale provést jednoduchý myšlenkový pokus:

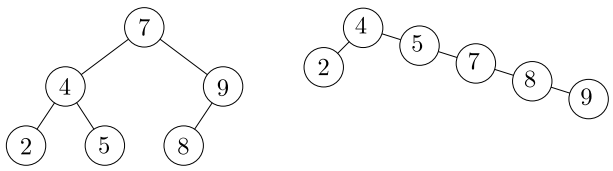
Vyhledávací stromy

Představme si, jakými všemi možnými cestami se může v našem poli binární vyhledávání ubírat. Na začátku porovnáváme s prostředním prvkem a podle výsledku se vydáme jednou ze dvou možných cest (nebo rovnou zjistíme, že se jedná o hledaný prvek, ale to není moc zajímavý případ). Na každé cestě nás zase čeká porovnání se středem příslušného intervalu a to nás opět pošle jednou ze dvou dalších cest atd. To si můžeme přehledně popsat pomocí stromu:



Jeden vrchol stromu prohlásíme za kořen a ten bude odpovídat celému poli (a jeho prostřednímu prvku). K němu budou připojené vrcholy obou polovin pole (opět obsahující příslušné prostřední prvky) a tak dále. Ovšem jakmile známe tento strom, můžeme náš půlící algoritmus provádět přímo podle stromu (ani k tomu nepotřebujeme vidět původní pole a umět v něm hledat poloviny): začneme v kořeni, porovnáme a podle výsledku se buďto přesuneme do levého nebo pravého podstromu, a tak dále. Každý průběh algoritmu bude tedy odpovídat nějaké cestě z kořene stromu do hledaného vrcholu.

Teď si ale všimněte, že aby hledání hodnoty podle stromu fungovalo, strom vůbec nemusel vzniknout půlením intervalu – stačilo, aby v každém vrcholu platilo, že všechny hodnoty v levém podstromu jsou menší než tento vrchol a naopak hodnoty v pravém podstromu větší. Hledání v témže poli by také popisovaly následující stromy (např.):



Hledací algoritmus podle jiných stromů samozřejmě už nemusí mít pěknou logaritmickou složitost (kdybychom hledali podle „degenerovaného“ stromu z pravého obrázku, trvalo by to dokonce lineárně). Důležité ale je, že takovéto stromy se dají poměrně snadno modifikovat a že je při troše šikovnosti můžeme udržet dostatečně podobné ideálnímu půlení intervalu. Pak bude hloubka stromu stále $\mathcal{O}(\log N)$, tím pádem i časová složitost hledání, a jak za chvilku uvidíme, i mnohých dalších operací.

Definice

Zkusme si tedy pořádně nadefinovat to, co jsme právě vymysleli:

Binární vyhledávací strom (podomácku BVS) je buď prázdná množina nebo *kořen* obsahující jednu hodnotu a mající dva *podstromy* (levý a pravý), což jsou opět BVS, ovšem takové, že všechny hodnoty uložené v levém podstromu jsou menší než hodnota v kořeni, a ta je naopak menší než všechny hodnoty uložené v pravém podstromu.

Úmluva: Pokud x je kořen a L_x a R_x jeho levý a pravý podstrom, pak kořenům těchto podstromů (pokud nejsou prázdné) budeme říkat *levý a pravý syn* vrcholu x a naopak vrcholu x budeme říkat *otec* těchto synů. Pokud je některý z podstromů prázdný, pak vrchol x příslušného syna nemá. Vrcholu, který nemá žádné syny, budeme říkat *list* vyhledávacího stromu. Všimněte si, že pokud x má jen jediného syna, musíme stále rozlišovat, je-li to syn levý nebo pravý, protože potřebujeme udržet správné uspořádání hodnot. Také si všimněte, že pokud známe syny každého vrcholu, můžeme již rekonstruovat všechny podstromy.

Každý BVS také můžeme popsat velmi jednoduchou strukturou v paměti:

```
struct vrchol {
    struct vrchol *levy, *pravy; // synové
    int x;                 // hodnota ve vrcholu
};
```

Pokud některý ze synů neexistuje, zapíšeme do příslušné položky hodnotu `NULL`.

Hledání

V řeči BVS můžeme přeformulovat náš vyhledávací algoritmus takto:

```
/* Dostane kořen stromu a hodnotu. Vrátí vrchol,
   kde se nachází, nebo NULL, není-li. */
struct vrchol *strom_najdi(struct vrchol *v,
                           int x)
{
    while (v != NULL && v->x != x) {
        if (x < v->x)
            v = v->levy;
        else
            v = v->pravy;
    }
    return v;
}
```

Funkce `strom_najdi` bude pracovat v čase $\mathcal{O}(h)$, kde h je hloubka stromu, protože začíná v kořeni a v každém průchodu cyklem postoupí o jednu hladinu níže.

Vkládání

Co kdybychom chtěli do stromu vložit novou hodnotu (aniž bychom se teď starali o to, zda tím strom nemůže degenerovat)? Stačí zkusit hodnotu najít a pokud tam ještě nebyla, určitě při hledání narazíme na odbočku, která je `NULL`. A přesně na toto místo připojíme nově vytvořený vrchol, aby byl správně uspořádán vzhledem k ostatním vrcholům (že tomu tak je, plyne z toho, že při hledání jsme postupně vyloučili všechna ostatní místa, kde nová hodnota být nemohla). Naprogramujeme opět snadno, tentokráte si ukážeme rekurzivní zacházení se stromy:

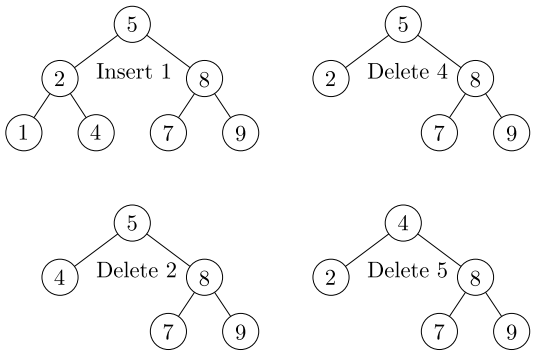
```
/* Dostane kořen stromu a hodnotu ke vložení,
   vrátí nový kořen. */
struct vrchol *strom_vloz(struct vrchol *v,
                          int x)
{
    if (v == NULL) { // prázdný strom
        // založíme nový kořen
        v = malloc(sizeof(struct vrchol));
        v->levy = v->pravy = NULL;
        v->x = x;
    } else if (x < v->x) // vkládáme vlevo
        v->levy = strom_vloz(v->levy, x);
    else if (x > v->x)  // vkládáme vpravo
        v->pravy = strom_vloz(v->pravy, x);
    return v;
}
```

Mazání

Mazání bude o kousíček pracnější, musíme totiž rozlišit tři případy: Pokud je mazaný vrchol list, stačí ho vyměnit za `NULL`. Pokud má právě jednoho syna, stačí náš vrchol v ze stromu odstranit a syna přepojit k otci v. A pokud má syny dva, najdeme největší hodnotu v levém podstromu (tu najdeme tak, že půjdeme jednou doleva a pak pořád doprava), umístíme ji do stromu namísto mazaného vrcholu a v levém podstromu ji pak smažeme (což už umíme, protože má 1 nebo 0 synů). Program následuje:

```
/* Parametry stejně jako strom_vloz */
struct vrchol *strom_vymaz(struct vrchol *v,
                           int x)
{
    struct vrchol *w, *ret = v;
    if (v == NULL) return v; // prázdný strom
    else if (x < v->x)  // hledáme x
        v->levy = strom_vymaz(v->levy, x);
    else if (x > v->x)
        v->pravy = strom_vymaz(v->pravy, x);
    else { // našli jsme x ... jaké má syny?
        if (v->levy == NULL
            && v->pravy == NULL) {  // žádné
            free(v);
            return NULL;
        } else if (v->levy == NULL) {
            // jen pravý syn
            ret = v->pravy;
            free(v);
            return ret;
        } else if (v->pravy == NULL) {
            // jen levý syn
            ret = v->levy;
            free(v);
            return ret;
        } else { // má oba dva syny
            w = v->levy; // hledáme max(L)
            while (w->pravy != NULL)
                w = w->pravy;
            v->x = w->x; // prohazujeme
            // mažeme původní max(L)
            v->levy = tree_del(v->levy, w->x);
        }
    }
    return v;
}
```

Když do stromu z našeho prvního obrázku zkusíme přidávat nebo z něj odebírat prvky, dopadne to takto:



Jak vkládání, tak mazání opět budou trvat $\mathcal{O}(h)$, kde h je hloubka stromu. Ale pozor, jejich používáním může h nekontrolovatelně růst (v závislosti na počtu prvků ve stromě).

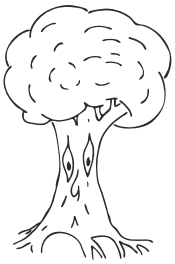
Cvičení

- Zkuste najít nějaký příklad, kdy h dosáhne až N – při postupném budování stromu operacemi vkládání i při mazání ze stromu hloubky $\mathcal{O}(\log N)$.

Procházení stromu

Pokud bychom chtěli všechny hodnoty ve stromu vypsat, stačí strom rekurzivně projít a sama definice uspořádání hodnot ve stromu nám zajistí, že hodnoty vypíšeme ve vzestupném pořadí: nejdříve levý podstrom, pak kořen a pak podstrom pravý. Časová složitost je, jak se snadno nahlédne, lineární, protože strávíme konstantní čas vypisováním každého prvku a prvků je právě N. Program bude opět přímočarý:

```
void strom_ukaz(struct vrchol *v){
    if (v == NULL)
        return; // není co dál dělat
    printf("(");
    strom_ukaz(v->levy);
    printf(")\%d(",v->x);
    strom_ukaz(v->pravy);
    printf(")");
}
```



Vyvážené stromy

S binárními stromy lze dělat všelijaká kouzla a prakticky všechny stromové algoritmy mají společné to, že jejich časová složitost je lineární v hloubce stromu. (Pravda, právě ten poslední je výjimka, leč všechny prvky rychleji než lineárně s N opravdu nevypíšeme.)

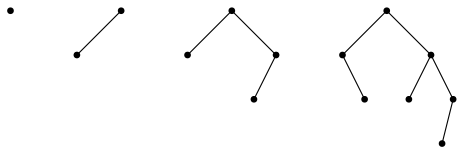
Jenže jak jsme viděli, neopatrným insertováním a deletováním prvků mohou snadno vznikat všelijaké degenerované stromy, které mají lineární hloubku. Abychom tomu zabránili, musíme stromy *vyvažovat.* To znamená definovat si nějaké šikovné omezení na tvar stromu, aby hloubka byla vždy $\mathcal{O}(\log N)$. Možností je mnoho, my uvedeme jen ty nejdůležitější:

Dokonale vyvážený budeme říkat takovému stromu, ve kterém pro každý vrchol platí, že počet vrcholů v jeho levém a pravém podstromu se liší nejvýše o jedničku. Takové stromy kopírují dělení na poloviny při binárním vyhledávání, a proto mají vždy logaritmickou hloubku. Jediné, čím se liší, je, že mohou zaokrouhlovat na obě strany, zatímco náš půlící algoritmus zaokrouhloval polovinu vždy dolů, takže levý podstrom nemohl být nikdy větší než pravý.

Z toho také plyne, že se snadnou modifikací půlícího algoritmu dá dokonale vyvážený BVS v lineárním čase vytvořit ze setříděného pole. Bohužel se ale při Insertu a Deletu nedá v logaritmickém čase strom znovu vyvážit.

AVL stromy

Zkusíme tedy vyvažovací podmínku trochu uvolnit a vyžadovat, aby se u každého vrcholu lišily o jedničku nikoliv velikosti podstromů, nýbrž pouze jejich hloubky. Takovým stromům se říká *AVL stromy* a mohou vypadat třeba takto:



Každý dokonale vyvážený strom je také AVL stromem, ale jak je vidět na předchozím obrázku, opačně to platit nemusí. To, že hloubka AVL stromů je také logaritmická, proto není úplně zřejmé a zaslouží si to trochu dokazování:

Věta: AVL strom o N vrcholech má hloubku $\mathcal{O}(\log N)$.

Důkaz: Označme A_d nejmenší možný počet vrcholů, jaký může být v AVL stromu hloubky d. Snadno zjistíme, že $A_1 = 1$, $A_2 = 2$, $A_3 = 4$ a $A_4 = 7$ (příslušné minimální stromy najdete na předchozím obrázku). Navíc platí, že $A_d = 1 + A_{d-1} + A_{d-2}$, protože každý minimální strom hloubky d musí mít kořen a 2 podstromy, které budou opět minimální, protože jinak bychom je mohli vyměnit za minimální a tím snížit počet vrcholů celého stromu. Navíc alespoň jeden z podstromů musí mít hloubku $d-1$ (protože jinak by hloubka celého stromu nebyla d) a druhý hloubku $d-2$ (podle definice AVL stromu může mít $d-1$ nebo $d-2$, ale s menší hloubkou bude mít evidentně méně vrcholů).

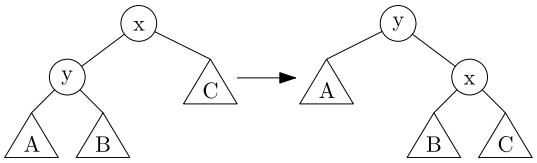
Spočítat, kolik přesně je A_d, není úplně snadné. Nám však postačí dokázat, že $A_d \geq 2^{d/2}$. To provedeme indukcí: Pro $d < 4$ to plyne z ručně spočítaných hodnot. Pro $d \geq 4$ je $A_d = 1 + A_{d-1} + A_{d-2} > 2^{(d-1)/2} + 2^{(d-2)/2} = 2^{d/2} \cdot (2^{-1/2} + 2^{-1}) > 2^{d/2}$ (součet čísel v závorce je ≈ 1.207).

Jakmile už víme, že A_d roste s d alespoň exponenciálně, tedy že $\exists c : A_d \geq c^d$, důkaz je u konce: Máme-li AVL strom T na N vrcholech, najdeme si nejmenší d takové, že $A_d \leq N$. Hloubka stromu T může být maximálně d, protože jinak by T musel mít alespoň A_{d+1} vrcholů, ale to je více než N. A jelikož A_d rostou exponenciálně, je $d \leq \log_c N$, čili $d = \mathcal{O}(\log N)$. *Q.E.D.*

AVL stromy tedy vypadají nadějně, jen stále nevíme, jak provádět Insert a Delete tak, strom zůstal vyvážený. Nemůžeme si totiž dovolit strukturu stromu měnit libovolně – stále musíme dodržovat správné uspořádání hodnot. K tomu se nám bude hodit zavést si nějakou množinu operací, o kterých dokážeme, že jsou korektní, a pak strukturu stromu měnit vždy jen pomocí těchto operací. Budou to rotace a dvojrotace.

Rotace

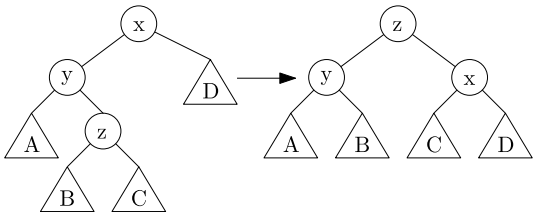
Rotací binárního stromu (respektive nějakého podstromu) nazveme jeho „překořenění“ za některého ze synů kořene. Místo formální definice ukažme raději obrázek (trojúhelník zastupuje podstrom, který může být v některých případech i prázdný):



Strom jsme překořenili za vrchol y a přepojili jednotlivé podstromy tak, aby byly vzhledem k x a y opět uspořádané správně (všimněte si, že je jen jediný způsob, jak to udělat). Jelikož se tím okolí vrcholu y „otočilo“ po směru hodinových ručiček, říká se takové operaci *rotace doprava.* Inverzní operaci (tj. překořenění za pravého syna kořene) se říká *rotace doleva* a na našem obrázku odpovídá přechodu zprava doleva.

Dvojrotace

Také si nakreslíme, jak to dopadne, když provedeme dvě rotace nad sebou lišící se směrem (tj. jednu levou a jednu pravou nebo opačně). Tomu se říká *dvojrotace* a jejím výsledkem je překořenění podstromu za vnuka kořene připojeného „cikcak“. Raději opět předvedeme na obrázku:



Znaménka

Při vyvažování se nám bude hodit pamatovat si u každého vrcholu, v jakém vztahu jsou hloubky jeho podstromů. Tomu budeme říkat *znaménko* vrcholu a bude buďto 0, jsou-li oba stejně hluboké, − pro levý podstrom hlubší a + pro pravý hlubší. V textu budeme znaménka, respektive vrcholy se znaménky značit $\odot$, $\ominus$ a $\oplus$.

Pokud celý strom zrcadlově obrátíme (prohodíme levou a pravou stranu), znaménka se změní na opačná ($\oplus$ a $\ominus$ se prohodí, $\odot$ zůstane). Toho budeme často využívat a ze dvou zrcadlově symetrických situací popíšeme jenom jednu s tím, že druhá se v algoritmu zpracuje symetricky.

Často také budeme potřebovat nalézt otce nějakého vrcholu. To můžeme zařídit buďto tak, že si do záznamů popisujících vrcholy stromu přidáme ještě ukazatele na otce a budeme ho ve všech operacích poctivě aktualizovat, anebo využijeme toho, že jsme do daného vrcholu museli někudy přijít z kořene, a celou cestu z kořene si zapamatujeme v nějakém zásobníku a postupně se budeme vracet.

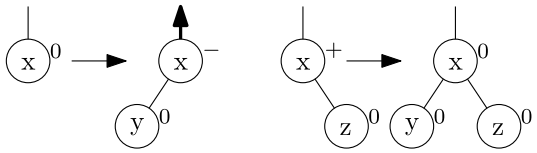
Tím jsme si připravili všechny potřebné ingredience, tož s chutí do toho.

Vyvažování po Insertu

Když provedeme Insert tak, jak jsme ho popisovali u obecných vyhledávacích stromů, přibude nám ve stromu list. Pokud se tím AVL vyváženost neporušila, stačí pouze opravit znaménka na cestě z nového listu do kořene (všude jinde zůstala zachována). Pakliže porušila, musíme s tím něco provést, konkrétně ji šikovně zvolenými rotacemi opravit.

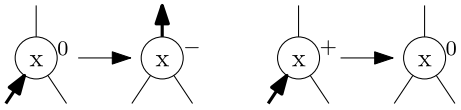
Popíšeme algoritmus, který bude postupovat od listu ke kořeni a vše potřebné zařídí.

Nejprve přidání listu samotné:

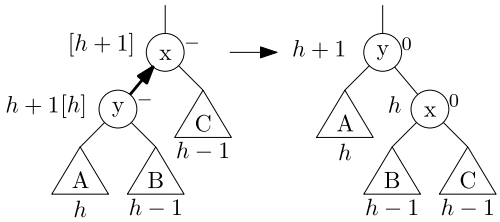


Pokud jsme přidali list (bez újmy na obecnosti levý, jinak vyřešíme zrcadlově) vrcholu se znaménkem $\odot$, změníme znaménko na $\ominus$ a pošleme o patro výš informaci o tom, že hloubka podstromu se zvýšila (to budeme značit šipkou). Přidali-li jsme list k $\oplus$, změní se na $\odot$ a hloubka podstromu se nemění, takže můžeme skončit.

Nyní rozebereme případy, které mohou nastat na vyšších hladinách, když nám z nějakého podstromu přijde šipka. Opět budeme předpokládat, že přišla zleva; pokud zprava, vyřešíme zrcadlově. Pokud přišla do $\oplus$ nebo $\odot$, ošetříme to stejně jako při přidání listu:

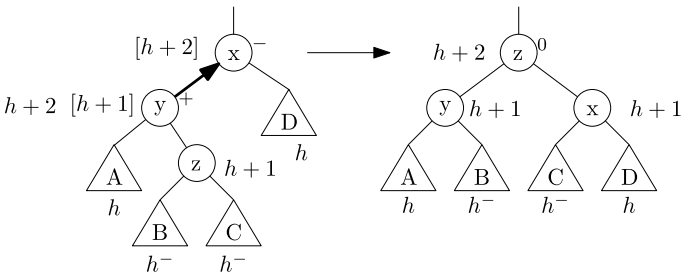


Pokud ale vrchol x má znaménko $\ominus$, nastanou potíže: levý podstrom má teď hloubku o 2 vyšší než pravý, takže musíme rotovat. Proto se podíváme o patro níž, jaké je znaménko vrcholu y pod šipkou, abychom věděli, jakou rotaci provést. Jedna možnost je tato (y je $\ominus$):



Tehdy provedeme jednoduchou rotaci vpravo. Jak to dopadne s hloubkami jsme přikreslili do obrázku – pokud si hloubku podstromu A označíme jako h, B musí mít hloubku $h-1$, protože y je $\ominus$, atd. Jen nesmíme zapomenout, že v x jsme ještě $\ominus$ nepřepočítali (vede tam přeci šipka), takže ve skutečnosti je jeho levý podstrom o 2 hladiny hlubší než pravý (původní hloubky jsme na obrázku naznačili [v závorkách]). Po zrotování vyjdou u x i y znaménka $\odot$ a celková hloubka se nezmění, takže jsme hotovi.

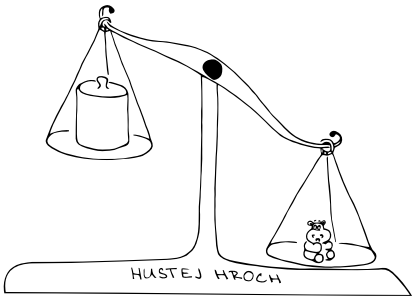
Další možnost je y jako $\oplus$:



Tehdy se podíváme ještě o hladinu níž a provedeme dvojrotaci. (Nemůže se nám stát, že by z neexistovalo, protože jinak by v y nebylo $\oplus$.) Hloubky opět najdete na obrázku. Jelikož z může mít libovolné znaménko, jsou hloubky podstromů B a C buďto h nebo $h-1$, což značíme h^-. Podle

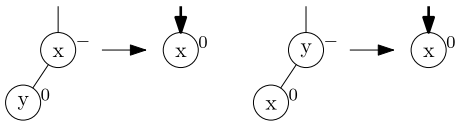
toho pak vyjdou znaménka vrcholů x a y po rotaci. Každopádně vrchol z vždy obdrží $\odot$ a celková hloubka se nemění, takže končíme.

Poslední možnost je, že by y byl $\odot$, ale tu vyřešíme velmi snadno: všimneme si, že nemůže nastat. Kdykoliv totiž posíláme šipku nahoru, není pod ní $\odot$. (Kontrolní otázka: jak to, že $\oplus$ může nastat?)

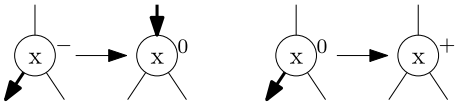


Vyvažování po Deletu

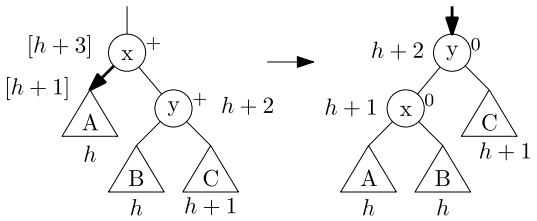
Vyvažování po Deletu je trochu obtížnější, ale také se dá popsat pár obrázky. Nejdříve opět rozebereme základní situace: odebíráme list (bez újmy na obecnosti (BÚNO) levý) nebo vnitřní vrchol s jediným synem (tehdy ale musí být jeho jediný syn listem, jinak by to nebyl AVL strom):



Šipkou dolů značíme, že o patro výš posíláme informaci o tom, že se hloubka podstromu snížila o 1. Pokud šipku dostane vrchol typu $\ominus$ nebo $\odot$, vyřešíme to snadno:

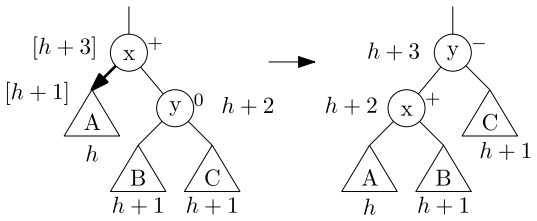


Problematické jsou tentokráte ty případy, kdy šipku dostane $\oplus$. Tehdy se musíme podívat na znaménko *opačného* syna a podle toho rotovat. První možnost je, že opačný syn má $\oplus$:



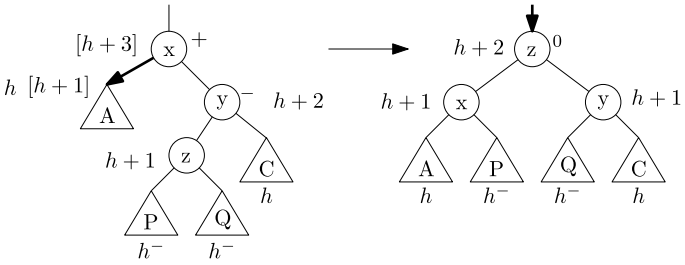
Tehdy provedeme rotaci vlevo, x i y získají nuly, ale celková hloubka stromu se sníží o hladinu, takže nezbývá, než poslat šipku o patro výš.

Pokud by y byl $\odot$:



Opět rotace vlevo, ale tentokráte se zastavíme, protože celková hloubka se nezměnila.

Poslední, nejkomplikovanější možnost je, že by y byl $\ominus$:



V tomto případě provedeme dvojrotaci (z určitě existuje, jelikož y je typu $\ominus$), vrcholy x a y obdrží znaménka v závislosti na původním znaménku vrcholu z a celý strom se snížil, takže pokračujeme o patro výš.

Happy end

Jak při Insertu, tak při Deletu se nám podařilo strom upravit tak, aby byl opět AVL stromem, a trvalo nám to lineárně s hloubkou stromu (konáme konstantní práci na každé hladině), čili stejně jako trvá Insert a Delete samotný. Jenže o AVL stromech jsme již dokázali, že mají hloubku vždy logaritmickou, takže jak hledání, tak Insert a Delete zvládneme v logaritmickém čase (vzhledem k aktuálnímu počtu prvků ve stromu).

Další typy stromů

AVL stromy samozřejmě nejsou jediný způsob, jak zavést stromovou datovou strukturu s logaritmicky rychlými operacemi. Jaké jsou další?

2-3-stromy nemají v jednom vrcholu uloženu jednu hodnotu, nýbrž jednu nebo dvě (a synové jsou pak 2 nebo 3, odtud název.) Přidáme navíc pravidlo, že všechny listy jsou na téže hladině. Hloubka vyjde logaritmická, vyvažování řešíme pomocí spojování a rozdělování vrcholů.

Červeno-černé stromy si místo znamének vrcholy barví. Každý je buďto červený nebo černý a platí, že nikdy nejsou dva červené vrcholy pod sebou a že na každé cestě z kořene do listu je stejný počet černých vrcholů. Hloubka je pak znovu logaritmická.

Po Insertu a Deletu barvy opravujeme rotováním a přebarvováním na cestě do kořene, jen je potřeba rozebrat podstatně více případů než u AVL stromů. (Za to jsme ale odměněni tím, že nikdy neděláme více než 2 rotace.) Počet případů k rozebrání lze omezit zpřísněním podmínek na umístění červených vrcholů – dvěma různým takovým zpřísněním se říká *AA-stromy* a *left-leaning červeno-černé stromy*.

Interpretujeme-li červené vrcholy jako rozšíření otcovského vrcholu o další hodnoty, pochopíme, že jsou červeno-černé stromy jen jiným způsobem záznamu 2-4-stromů. Proč se takový kryptický překlad dělá? S třemi potomky vrcholu a dvěma hodnotami se pracuje nešikovně.

V případě *splay stromů* nezavádíme žádnou vyvažovací podmínku, nýbrž definujeme, že kdykoliv pracujeme s nějakým vrcholem, vždy si jej vyrotujeme do kořene a pokud to jde, preferujeme dvojrotace. Takové operaci se říká Splay a dají se pomocí ní definovat operace ostatní: Find hodnotu najde a poté na ni zavolá Splay. Insert si nechá vysplayovat předchůdce nové hodnoty a vloží nový vrchol mezi předchůdce a jeho pravého syna. Delete vysplayuje mazaný prvek, pak uvnitř pravého podstromu vysplayuje minimum, takže bude

mít jen jednoho syna a můžeme jím tedy nahradit mazaný prvek v kořeni.

Jednotlivé operace samozřejmě mohou trvat až lineárně dlouho, ale dá se o nich dokázat, že jejich *amortizovaná* složitost je vždy $\mathcal{O}(\log N)$. Tím chceme říci, že provést t po sobě jdoucích operací začínajících prázdným stromem trvá $\mathcal{O}(t \cdot \log N)$ (některé operace mohou být pomalejší, ale to je vykoupeno větší rychlostí jiných).

To u většiny použití stačí – datovou strukturu obvykle používáte uvnitř nějakého algoritmu a zajímá vás, jak dlouho běží všechny operace dohromady – a navíc je Splay stromy daleko snazší naprogramovat než nějaké vyvažované stromy. Mimo to mají Splay stromy i jiné krásné vlastnosti: přizpůsobují svůj tvar četnostem hledání, takže často hledané prvky jsou pak blíž ke kořeni, snadno se dají rozdělovat a spojovat, atd.

Treapy jsou randomizovaně vyvažované stromy: něco mezi stromem (tree) a haldou (heap). Každému prvku přiřadíme *váhu,* což je náhodné číslo z intervalu $\langle 0, 1\rangle$. Strom pak udržujeme uspořádaný stromově podle hodnot a haldově podle vah (všimněte si, že tím je jeho tvar určen jednoznačně, pokud tedy jsou všechny váhy navzájem různé, což skoro jistě jsou). Insert a Delete opravují haldové uspořádání velmi jednoduše pomocí rotací. Časová složitost v průměrném případě je $\mathcal{O}(\log N)$.

BB-α stromy nabízí zobecnění dokonalé vyváženosti jiným směrem: zvolíme si vhodné číslo α a vyžadujeme, aby se velikost podstromů každého vrcholu lišila maximálně α-krát (prázdné podstromy nějak ošetříme, abychom nedělili nulou; dokonalá vyváženost odpovídá $\alpha = 1$ (až na zaokrouhlování)). V každém vrcholu si budeme pamatovat, kolik vrcholů obsahuje podstrom, jehož je kořenem, a po Insertu a Deletu přepočítáme tyto hodnoty na cestě zpět do kořene a zkontrolujeme, jestli je strom ještě stále α-vyvážený.

Pokud ne, najdeme nejvyšší místo, ve kterém se velikosti podstromů příliš liší, a vše od tohoto místa dolů znovu vytvoříme algoritmem na výrobu dokonale vyvážených stromů. Ten, pravda, běží v lineárním čase, ale čím větší podstrom přebudováváme, tím to děláme méně často, takže vyjde opět amortizovaně $\mathcal{O}(\log N)$ na operaci.

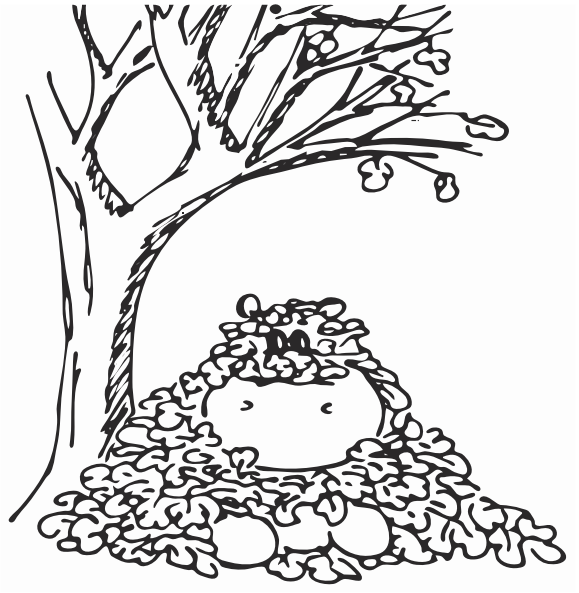
Cvičení

- Jak konstruovat dokonale vyvážené stromy?
- Jak pomocí toho naprogramovat BB-α stromy?
- Najděte algoritmus, který k prvku v obecném vyhledávacím stromu najde jeho následníka, což je prvek s nejbližší vyšší hodnotou (zde předpokládejte, že ke každému prvku máte uložený ukazatel na jeho otce).
- Jak vypsat celý strom tak, že začnete v minimu a budete postupně hledat následníky? (I když nalezení následníka může trvat až $\mathcal{O}(h)$, všimněte si, že projití celého stromu přes následníky bude lineární.)
- Jak do vrcholů stromu ukládat různé pomocné informace, jako třeba počet vrcholů v podstromu každého vrcholu, a jak tyto informace při operacích se stromem (při Insertu, Deletu, rotaci) udržovat?
- Ukažte, že lze libovolný interval $\langle a, b\rangle$ rozložit na logaritmicky mnoho intervalů odpovídajících podstromům vyváženého stromu.
- Ukažte, že zkombinováním předchozích dvou cvičení lze odpovídat i na otázky typu „kolik si strom pamatuje hodnot ze zadaného intervalu“ v logaritmickém čase ...

Poznámky

- Představte si, že budujete binární vyhledávací strom vkládáním prvků v náhodném pořadí. Obecně nemusí být vyvážený, ale v průměru v něm bude možné vyhledávat v čase $\mathcal{O}(\log N)$. Žádný div: Stromy, které nám vzniknou, odpovídají přesně možným průběhům QuickSortu, který má průměrnou časovou složitost $\mathcal{O}(N \log N)$.
- Pokud bychom připustili, že se mohou vyskytnout dva stejné záznamy, budou stromy stále fungovat, jen si musíme dát o něco větší pozor na to, co všechno při operacích se stromem může nastat.
- Jakpak přišly AVL stromy ke svému jménu? Podle Adeľsona-Veľského a Landise, kteří je objevili.
- Rekurenci $A_d = 1 + A_{d-1} + A_{d-2}$, $A_1 = 1$, $A_2 = 2$ pro velikosti minimálních AVL stromů je samozřejmě možné vyřešit i přesně. Žádné překvapení se nekoná, objeví se totiž stará známá Fibonacciho čísla: $A_n = F_{n+2} - 1$.

Martin „Medvěd“ Mareš & Tomáš Valla



26-1-1 Blokující signály

Řekněme, že dokážeme zastavit nějaký signál v uzlu X. Co to znamená? To znamená, že můžeme z hlavního počítače poslat blokující signál, který do uzlu X dorazí ve stejný moment jako ten vadný.

Všimněme si, že když můžeme poslat z hlavního počítače signál, který dorazí do uzlu X v čase T, tak můžeme poslat i signál, který dojde kdykoliv potom. Můžeme totiž signál z hlavního počítače jednoduše vyslat později.

Ke každému uzlu X tedy existuje nějaký minimální čas T_0 takový, že dokážeme zablokovat každý vadný signál, který do uzlu X dorazí nejdříve v čase T_0. Speciálně pro hlavní počítač je T_0 rovno nule: pokud vadný signál prochází hlavním počítačem, stačí si na něj počkat.

Na každý uzel X, který je nějak spojený s hlavním počítačem, můžeme nejdříve dosáhnout v čase, za který stihneme přejít nejkratší cestou z hlavního počítače do uzlu X. Stačí si tedy zjistit délku nejkratších cest z počátku do všech ostatních uzlů, čímž získáme všechny časy T_0.

Tahle úloha je (jak si jistě zkušenější řešitelé všimli na první pohled) grafová. Na zjištění délek všech nejkratších cest z nějakého daného vrcholu se v obecném grafu dá použít třeba Dijkstrův algoritmus, který jde naprogramovat tak, aby seběhl v čase $\mathcal{O}(M + N \log N)$ na grafu s N vrcholy a M hranami. Protože ale v této síti jsou všechny hrany (neboli spojení mezi počítači) stejně dlouhé, můžeme nejkratší cesty najít prostým prohledáváním do šířky, což se stihne za $\mathcal{O}(M + N)$. Prohledávání do šířky si můžete představit jako postupné „oloupávání“ sítě: nejdříve utrhneme hlavní počítač, pak všechny počítače na něj napojené (tedy ve vzdálenosti 1), pak všechny napojené na ně (2 kroky daleko), a tak dále. Na detaily implementace se můžete podívat ve zdrojáku vzorového řešení.

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

Michal Pokorný

26-1-2 Přeskládání nákladu

Problém rozdělení kontejnerů do skladišť můžeme převést na obarvování neorientovaného grafu. Kontejnery budou vrcholy, doporučení budou hrany mezi nimi a obarvení vrcholů dvěma barvami bude znázorňovat jejich rozdělení do skladišť.

Naším cílem je obarvit vrcholy grafu tak, aby hran vedoucích mezi vrcholy stejné barvy (tedy nesplněných doporučení) byla nejvýše polovina.

Ve speciálním případě, kdy bychom měli slíbeno, že graf je čistě bipartitní (tedy že se dá rozdělit na dvě množiny, kde hrany vedou jen mezi množinami a ne uvnitř), bude obarvení vrcholů dvěma barvami triviální.

Dokud budeme mít nějaký neobarvený vrchol, budeme opakovat toto: obarvíme ho první barvou, všechny jeho sousedy druhou, všechny sousedy sousedů opět první a tak dále. Protože každý vrchol sousedí pouze s vrcholy z opačné partity, povedlo by se nám takto splnit všechna doporučení u každého z vrcholů.

Pro bipartitní grafy je to tedy snadné. Jak to ale bude v obecném případě? Už se nám asi nepovede splnit všechna doporučení, ale můžeme se pokusit splnit jich alespoň polovinu.[5]

Náš algoritmus bude pracovat po krocích a v každém kroku se bude lokálně pokoušet splnit alespoň polovinu doporučení. Nejdříve se podíváme, jak bude vypadat jeden jeho krok, a potom si dokážeme, že tím splníme alespoň polovinu doporučení i globálně.

Krok algoritmu:

1. Vezmi libovolný neobarvený kontejner.
2. Spočítej, kolik má sousedů které barvy.
3. Pokud má sousedů jedné barvy méně, obarvi ho touto barvou. Jinak libovolně.
4. Dokud nejsou všechny kontejnery obarveny, pokračuj bodem 1.

Pro účely dokazování přiřadíme každé doporučení jen jednomu z dvojice kontejnerů – tomu obarvenému později (tím si určitě nic nepokazíme, neboť jednotlivá doporučení ani jejich počet se nijak nezmění).

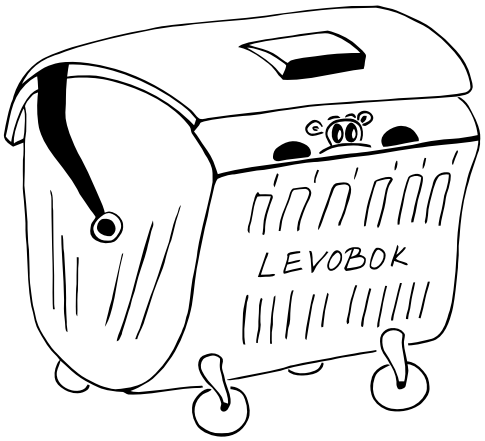
Každý z kontejnerů bude tedy mít svou vlastní množinu doporučení. Ale to jsou přesně ta doporučení, která jsme uvažovali v bodu 2 algoritmu a obarvením kontejneru jsme jich splnili alespoň polovinu. U každého kontejneru je tedy alespoň polovina doporučení splněna, takže v součtu přes všechny kontejnery musí být splněna také alespoň polovina doporučení. A tím je splněno i zadání.

Zbývá ještě časová a prostorová složitost. Vše, co si musíme pamatovat, je nějaký seznam vrcholů a ke každému vrcholu seznam jeho hran, takže paměťová složitost je $\mathcal{O}(N + M)$.

Časová složitost je mírně složitější. Provedeme sice N kroků algoritmu a v každém se můžeme podívat až na M hran (což by mohlo vést na $\mathcal{O}(MN)$), ale stačí si uvědomit, že celkově se za celý běh programu podíváme maximálně na $2M$ konců hran a tedy výsledná časová složitost bude jen $\mathcal{O}(N + M)$.

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

Jirka Setnička & Petra Pelikánová



[5] Pokud bychom jich však chtěli splnit co nejvíc, už by šlo o NP-úplnou úlohu.

26-1-3 Plynové kapsy

Jednoduché řešení

Pro každý dotaz prostě projdeme příslušný interval zleva doprava a pokud se aktuální znak bude shodovat s předchozím, přičteme jedničku. Toto řešení je ale pomalé.

Rychlé řešení

Připravíme si pomocné pole. Na pozici i budeme mít uložený počet bezpečných míst v intervalu $\langle 0, i\rangle$. Tomu se říká prefixový součet. Potom při dotazu na $\langle i, j\rangle$ stačí od j-té pozice odečíst $(i-1)$-tou. Tím od všech bezpečných pozic nalevo od pravého konce intervalu odečteme ty nalevo od levého konce intervalu, tedy zbudou nám jen ty pozice uvnitř intervalu.

A jak si toto pomocné pole spočítat? Velmi podobně tomu, jak jsme počítali výsledek při jednoduchém řešení. Vypravíme se od levého konce a pokaždé, když bude aktuální znak stejný, jako předchozí, zvětšíme si průběžný počet o jedna. A v každém kroku si aktuální průběžný součet uložíme.

Proč to funguje, je vidět z vysvětlení v druhém odstavci. Pomocné pole nám zabere lineární množství paměti s velikostí vstupní posloupnosti. Co se týče času, pak předvýpočet je lineární s délkou posloupnosti na vstupu (projdeme jej jednou zleva doprava a v každém políčku uděláme konstantní množství práce). Jeden dotaz zodpovíme v konstantním čase, protože jen odečteme dvě čísla.

Program (C++):
`http://ksp.mff.cuni.cz/viz/26-1-3.cpp`

Lucka Mohelníková & Michal „Vorner“ Vaner

26-1-4 Oprava databáze

Vyřešme nejprve jednodušší variantu, totiž dvojné prvky. Pro každý prvek p_k v zadané posloupnosti můžeme vyzkoušet všechny dvojice předchozích prvků p_i, p_j a ověřit, zda náhodou jejich součet není p_k. Tím dostaneme řešení s časovou složitostí $\mathcal{O}(n^3)$. To ale není nejlepší řešení této úlohy.

Můžeme si všimnout, že zbytečně několikrát provádíme stejné součty. Co kdybychom si místo toho dané součty systematicky pamatovali a pak v nich jen vyhledávali? To můžeme udělat například binárním vyhledávacím stromem.

V něm si budeme uchovávat všechny možné součty dvojic prvků před aktuálním prvkem p_k. Tedy v moment zpracovávání prvku p_k v něm bume mít hodnoty součtů dvojic p_i, p_j pro $i, j < k$. Takže jen ověříme, zda je hodnota p_k obsažená ve stromě a pokud ano, tak p_k je dvojným prvkem. Nakonec přidáme do stromu všechny součty $p_k + p_i$ pro $i < k$ a pokračujeme prvkem p_{k+1}.

Toto řešení má časovou složitost $\mathcal{O}(n \cdot (\log n + n \log n)) = \mathcal{O}(n^2 \log n)$ a paměťovou složitost $\mathcal{O}(n^2)$.

Nyní pojďme vyřešit úlohu pro trojné prvky. Postupovat budeme velmi podobně. Opět budeme mít binární vyhledávající strom uchovávající součty zatím potkaných dvojic, akorát budeme rozdílně zpracovávat prvek p_k. Pro něj budeme předpokládat, že je třetím prvkem v součtu a pro všechny $j > k$ ověříme, zda je možné pomocí součtu dvou prvků před p_k dostat hodnotu $p_j - p_k$.

Pokud ano, tak prvek p_j je trojným prvkem. To zjistíme dotazem na binární vyhledávací strom. Pak stejně jako předtím do stromu přidáme všechny součty tvořené prvkem p_k a některým předchozím prvkem a přesuneme se s výpočtem na další prvek posloupnosti.

Řešení má časovou složitost $\mathcal{O}(n^2 \log n)$, protože provádíme $\mathcal{O}(n^2)$ operací s binárním vyhledávacím stromem. Při programování použijeme knihovní implementaci binárního vyhledávacího stromu, například v jazyce C++ to je `set` z knihovny STL. Celá realizace řešení je pak kratší, než tento slovní popis. :-)

Program (C++):
`http://ksp.mff.cuni.cz/viz/26-1-4.cpp`

Karel Tesař & Mark Karpilovskij

Medvědí poznámky

Dvojné prvky jde hledat o něco rychleji. Postupně procházíme přes všechna j a zjišťujeme, zda je součet prvku p_j s nějakým prvkem nalevo od něj roven nějakému prvku napravo od něj. Za tímto účelem si budeme udržovat dva setříděné seznamy: L bude obsahovat hodnoty ležících nalevo od aktuálního prvku, P ty napravo.

Pro každé j spočítáme $S_j = L + p_j$ (seznam vzniklý přičtením p_j ke každému prvku z L). To je také setříděný seznam, takže jeho sléváním s P můžeme snadno zjistit, zda S_j a P mají nějaký společný průnik, čili dvojný prvek. Všechny tyto operace zvládneme pro jedno j provést v lineárním čase, celkově tedy v $\mathcal{O}(n^2)$. Paměti spotřebujeme pouze $\mathcal{O}(n)$.

Vyhledávací stromy jsou mocná zbraň, kterou je dobré ovládat, ale občas lze věci řešit i jednodušeji. Jako třeba zde. Pro hledání trojných prvků postačí předpočítat si všechny možné součty dvojic, setřídit si je a pro každou hodnotu součtu si zapamatovat její nejlevější výskyt. Pak můžeme namísto ve stromu binárně vyhledávat v této setříděné posloupnosti a podle pozice nejlevějšího výskytu snadno ověřit, zda součet leží před zkoumaným p_k, anebo až za ním.

Pokud bychom se ovšem spokojili s algoritmem, který je rychlý v průměru a ne nutně v nejhorším případě, hodí se místo stromu použít hešovací tabulku – ta pracuje v průměrně konstantním čase na operaci, čímž se časová složitost hledání trojných prvků sníží na $\mathcal{O}(n^2)$. Najdeme ji i v STL pod názvem `unordered_set`.

Martin „Medvěd“ Mareš

26-1-5 Senzory

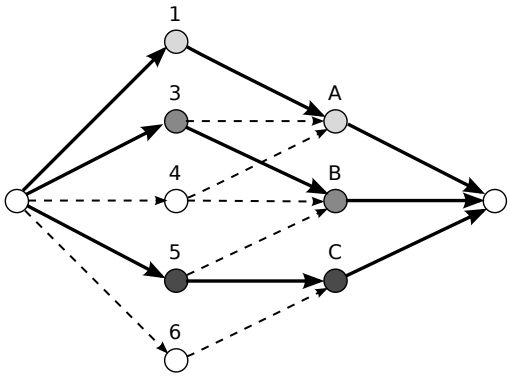
Zkoušení všech možností tady moc efektivní nebude. Pojďme úlohu trochu rozebrat, aby se na ni jednodušeji útočilo.

První učiněné pozorování bude následující: můžeme věže rozmístit do řádků a do sloupečků nezávisle. Když se totiž věže ohrožují, tak se ohrožují buď v řádku, nebo ve sloupečku, a naopak pokud je rozmístění věží správné v řádcích i ve sloupečcích, je správné i celkově. Zredukovali jsme si tedy zadání na jeho jednodušší verzi, ve které se věže staví do jednoho řádku, dvě nesmí stát ve stejném sloupci, a každá může stát jenom v nějakém vymezeném intervalu. Vyřešíme tyhle podúlohy pro sloupce a pro řádky zvlášť a výsledky spojíme.

Řešení pomocí systému různých reprezentantů

Jednodušší podúloha je speciální případ hledání takzvaného *systému různých reprezentantů*. Mějme třeba množiny $A = \{1, 3, 4\}, B = \{3, 4, 5\}, C = \{5, 6\}$. Systém různých reprezentantů je takové přiřazení prvků množin k jejich množinám, že všechny vybrané prvky jsou různé. Příklad systému různých reprezentantů pro tyto množiny A, B, C je třeba $A \to 1, B \to 3, C \to 5$ (ale třeba $A \to 3, B \to 3, C \to 6$

nebo $A \to 5, B \to 3, C \to 6$ už ne). Obecně se systémy různých reprezentantů dají hledat přes toky v sítích. Vytvoříme si bipartitní graf, ve kterém jedna partita budou prvky množin $(1, 3, 4, 5, 6)$ a druhá budou množiny (A, B, C) a natáhneme hrany s kapacitou 1 mezi prvkem x a množinou M tam, kde $x \in M$.



Potom si vytvoříme zdroj, ze kterého povedou hrany kapacity 1 do všech prvků, a stok, do kterého povedou hrany kapacity 1 ze všech množin. V této síti najdeme maximální tok například pomocí Fordova-Fulkersonova algoritmu. Maximální tok nám ukáže hledaný systém různých reprezentantů (pokud existuje). Kapacity hran a podmínky, které musí tok splňovat, zajišťují, že všechny prvky i všechny množiny budou použity maximálně jednou.

Je to zcela správný způsob řešení úlohy o rozmisťování věží. Podrobnosti hledání maximálního toku tu však nenajdete, protože úloha jde řešit jednodušeji a rychleji. Máte-li o ně zájem, můžete je najít v kuchařce o tocích.[6]

Jednodušší řešení

Zkusíme to takzvaně hladově: můžeme nejdříve vybrat věž, kterou umístíme na první sloupeček, pak ze zbylých vybrat tu, kterou umístíme na druhý, a tak dále (samozřejmě přeskakujeme sloupečky bez věží). Jak si ale budeme věže vybírat? Překvapivě tu bude fungovat metoda „do prvního sloupečku vyber tu věž, se kterou jde nejméně hýbat“ (pojmenujme ji třeba *minimální věž*). To, že takový algoritmus bude fungovat, nahlédneme indukcí.

Důkaz správnosti

Indukci začneme třeba od triviálního případu s jednou věží: tu můžeme umístit hned na první sloupeček, na který může přijít, takže tam algoritmus funguje.

Indukční krok bude schematicky takovýhle: „Když nějaká věž má prázdný interval, je jasné, že žádné řešení neexistuje. Řekněme teď bez újmy na obecnosti, že na první sloupeček jde umístit nějaká věž. Vybereme z věží, které jdou dát do prvního sloupečku, libovolnou minimální, a položíme ji tam. Tím si zmenšíme zadání o jednu věž a jeden sloupeček. Necháme si od indukce přihrát řešení menšího problému. Pokud existuje, přidáme k němu tuhle věž a máme výsledek. Pokud neexistuje, pak neexistuje ani řešení problému včetně minimální věže.“

Zbývá teď dokázat, že pokud budeme věže takhle umisťovat, tak o žádné řešení nepřijdeme. (Důkaz toho, že žádné řešení nepřidáme, je jednoduchý.) Jinými slovy: pokud jdou věže nějak rozmístit podle zadání, tak jdou rozmístit i tak, že v prvním sloupečku bude z věží, které tam šly umístit, libovolná minimální.

[6] `http://ksp.mff.cuni.cz/viz/kucharky/toky-v-sitich`

Vezměme si nějaké řešení a vyberme si z věží, které můžou dostat první sloupeček, nějakou minimální. Označíme ji třeba M. Nechť M nedostane první sloupeček, ale sloupeček S_M. Pokud je první sloupeček v řešení volný, můžeme do něj M přemístit, čímž dostaneme řešení, ve kterém první sloupeček drží vybraná minimální věž. Pokud není první sloupec volný, znamená to, že nějaká věž X jej drží. Protože M je minimální věž, tak interval věže X obsahuje mimo jiné sloupeček S_M. Můžeme tedy věže X a M beztrestně prohodit. Po prohození opět dostaneme řešení, které má v prvním sloupci minimální věž.

Implementace

Máme dokázáno, že to bude fungovat, a zbývá to „jenom“ implementovat. Budeme postupně ukrajovat věže a sloupečky. V proměnné si budeme držet poslední sloupeček, do kterého jsme už umístili věž (a další věže budeme umisťovat jenom za něj).

Věže chceme brát v takovém pořadí, abychom pokaždé umisťovali tu, která je v nezpracovaném prostoru minimální. Jak toho dosáhneme? Mohli bychom si v každém kroku najít minimální věž průchodem všech zbylých věží, ale to by nás stálo kvadratický čas. Existuje lepší řešení: ukládat si věže do haldy. Halda konkrétně bude minimová a budeme v ní třídit podle konce intervalu, do kterého můžeme věž umístit. Také do ní věže nebudeme přidávat všechny hned, ale teprve okamžikem, kdy narazíme na začátek jejich intervalu. Kdykoliv z téhle haldy tedy odebereme věž s nejmenším koncem intervalu, bude to právě ta minimální pro sloupeček, který budeme zabydlovat.

Jako první krok věže v $\mathcal{O}(N \log N)$ setřídíme vzestupně podle minima intervalu. Potom budeme postupně zleva zaplňovat sloupečky a přidávat věže do haldy.

Každou věž zpracujeme takto: jde-li umístit hned za poslední umístěnou věž, učiníme to. Nejde-li to, zkusíme ji umístit na začátek jejího intervalu. Pokud ani to nejde, znamená to, že poslední umístěná věž je za koncem intervalu právě zpracovávané věže, takže zpracovávaná věž nejde umístit nikam – řešení neexistuje. Nakonec nesmíme zapomenout zvýšit proměnnou s poslední umístěnou věží.

A co nás to bude stát? Paměť je jednoduchá: stačí $\mathcal{O}(N)$ na uložení vstupu a haldy. Co se času týče: setřídění věží zabere $\mathcal{O}(N \log N)$. Každou věž také jednou uložíme do haldy a jednou z ní vybereme, což obojí trvá $\mathcal{O}(\log N)$ za věž, tedy celkem $\mathcal{O}(N \log N)$. Všechno ostatní trvá kratší dobu.

Program (C++):
`http://ksp.mff.cuni.cz/viz/26-1-5.cpp`

Michal Pokorný

26-1-6 Hydroponie

Zahrada skýtá přihrádky na rostliny označené čísly 0 až $N-1$. K dispozici máme M nápajecích okruhů, i-tý okruh je intervalem $I_i = [a_i, b_i) = \{a_i, a_i + 1, \ldots, b_i - 1\}$. Délka intervalu v tomto značení je $b_i - a_i$.

Pokoušíme se určit počet všech možných osazení rostlin do přihrádek tak, aby každý interval obsahoval alespoň jednu rostlinu. Protože mohou být výsledné hodnoty extrémně vysoké, počítáme pouze zbytek po dělení této hodnoty číslem 1 000 000 007.

Rozklad na schodiště

Proveďme úvodní pozorování. Platí-li pro navzájem různá i a j inkluze $I_i \subseteq I_j$, můžeme interval I_j ze svých úvah vypustit. Podmínku na obsazení rostliny v tomto intervalu nám zajistí interval I_i.

Dále budeme uvažovat pouze intervaly, které v sobě neobsahují žádný celý interval. Intervaly si navíc uspořádáme vzestupně dle jejich počátku. Pro libovolné $i < j$ budou nyní platit nerovnosti $a_i < a_j$ a $b_i < b_j$. (První nerovnost je dána uspořádáním, druhá tím, že I_j nemůže celý ležet v I_i.)

Získali jsme jakousi schodovitou strukturu. Tuto strukturu si rozebereme na jednotlivá schodiště. Schodiště S_i bude množinou intervalů I_u až I_v takovou, že každé dva po sobě jdoucí intervaly mají neprázdný průnik, navíc S_i bude vždy největší takovou možnou množinou. Tedy I_{u-1} a I_{v+1}, pokud existují, jsou disjunktní s I_u a I_v. Počet schodišť označme jako T.

Povšimněme si, že úloha se nám nyní rozpadla na $T+1$ zcela nezávislých podúloh. Pro přihrádky neobsažené v žádném intervalu nejsme ničím vázáni, tedy máme 2^K možností jejich obsazení, je-li K počtem těchto přihrádek. Zbývajícími podúlohami jsou jednotlivá schodiště.

Výpočet pro jedno schodiště

Mějme jediné schodiště s intervaly $J_0, \ldots, J_{\ell-1}$. Nyní označujme $J_i = [a_i, b_i)$.

Zužitkujeme myšlenku dynamického programování a budeme si postupně počítat hodnoty $D(0)$ až $D(\ell-1)$, kde $D(i)$ udává počet možných rozmístění květin, umisťujeme-li je pouze do intervalů 0 až i.

Počáteční hodnota $D(0)$ je zřejmě $2^{|J_0|} - 1$, neb nám je zakázáno pouze to obsazení květin v intervalu J_0, při kterém neumístíme ani jednu květinu.

Předokládejme nyní, že hodnoty $D(0), \ldots, D(i-1)$ již byly spočteny, a my chceme určit $D(i)$.

Buď p nejmenší takové, že $b_p > a_i$. Interval J_i rozdělíme na podintervaly $[a_i, b_p)$, $[b_p, b_{p+1})$, $[b_{p+1}, b_{p+2}), \ldots, [b_{i-1}, b_i)$.

Možnosti osazení květinami si rozdělíme podle toho, do kterého z těchto podintervalů umístíme nejpravější květinu. Nejprve se podívejme na intervaly tvaru $[b_j, b_{j+1})$ pro nějaké j splňující $p \leq j < i$. Později se ještě podíváme na speciální interval $[a_i, b_p)$. Výsledky pro jednotlivé intervaly pak sečteme a získáme $D(i)$.

Máme interval $[b_j, b_{j+1})$, ve kterém bude umístěna alespoň jedna rostlina, a víme, že na pozicích od b_{j+1} dále už žádná květina nebude. Možností, jak umístit květiny do tohoto podintervalu, je $2^{b_{j+1}-b_j} - 1$. Kolika způsoby lze korektně osadit zbytek schodiště udává $D(j)$, což dává celkem $\left(2^{b_{j+1}-b_j} - 1\right) \cdot D(j)$ možností pro tento interval.

Zbývá nám interval $[a_i, b_p)$. Je-li $p = 0$, pak je mezivýsledkem hodnota $2^{a_0-a_i} \cdot (2^{a_i-b_0} - 1)$. Musili jsme obsadit alespoň jednu květinu do intervalu $[a_i, b_0)$, zbytek intervalu J_0 jsme mohli obsadit libovolně.

Pro $p > 1$ se ještě podíváme na interval J_{p-1}. Pro ten už platí $b_{p-1} \leq a_i$. Počet možných osazení intervalu $[a_i, b_p)$ je opět $2^{a_i-b_0} - 1$. Interval $[b_{p-1}, a_i)$ lze obsadit libovolně, tedy $2^{a_i-b_{p-1}}$ způsoby. Jak obsadit všechny pozice před tímto intervalem už udává $D(p-1)$. Celkem tedy $\left(2^{a_i-b_0} - 1\right) \cdot 2^{a_i-b_{p-1}} \cdot D(p-1)$.

Celkový počet možných osazení tohoto schodiště je $D(\ell-1)$.

Plody našeho snažení

Protože jsou umístění do jednotlivých T schodišť a do přihrádek mimo schodiště nezávislá, získáme výsledek jako součin počtu možností pro tyto jednotlivé případy.

Zbývá se zamyslet, s jakou složitostí umíme celé naše řešení implementovat. Abychom si intervaly správně uspořádali a zbavili se přebytečných, stačí je setřídit a vhodně projít. (Detaily si rozmyslete sami, nebo si je přečtěte ve vzorové implementaci.)

Asymptotická složitost algoritmu bude funkcí dvou proměnných, jmenovitě N a M. V takovém případě nemusí být jednoznačné, která časová složitost je tou optimální. Může to záviset na vztahu mezi těmito proměnnými. (Optimální algoritmus by asi měl podle vstupních hodnot N a M zvolit správnou variantu implementace).

Předvedeme si dvě možné složitosti řešení. V prvním případě intervaly setřídíme v čase $\mathcal{O}(M \log M)$, třeba pomocí Quicksortu.

Během počítání možností pro všechna schodiště mocníme dvojku a to exponentem z rozsahu 0 až N. Tuto operaci jsme schopni provést v čase $\mathcal{O}(\log N)$.

Při výpočtu možností pro daný interval ve schodišti sčítáme možnosti podintervalů. Všimněme si, že tento součet není potřeba počítat vždy od nuly, ale stačí jej aktualizovat při zvyšování i a p. To znamená lineární počet operací vzhledem k počtu intervalů. Nejdražší operací je už zmíněné mocnění dvojky.

První řešení má tedy časovou složitost $\mathcal{O}(M \log N)$ a paměťovou složitost $O(M)$. (Všimněte si, že $\mathcal{O}(\log M)$ a $\mathcal{O}(\log N)$ je totéž, protože M je nejvýše N^2.)

Protože okraje intervalů jsou z rozsahu 0 až $N-1$, můžeme je také setřídit přihrádkovým tříděním v čase $\mathcal{O}(N)$. Všechny mocniny dvojky si můžeme dopředu předpočítat a pak na dotaz odpovídat v konstantním čase. Tak získáváme druhé řešení s časovou i paměťovou složitostí $\mathcal{O}(N + M)$.

Při implementaci nesmíme zapomenout všechny hodnoty průběžně nahrazovat jejich zbytkem po dělení 1 000 000 007, abychom nedostávali ohromná čísla.

Vzorová implementace ukazuje řešení v čase $\mathcal{O}(M \log N)$.

Program (C):
`http://ksp.mff.cuni.cz/viz/26-1-6.c`

Lukáš Folwarczný

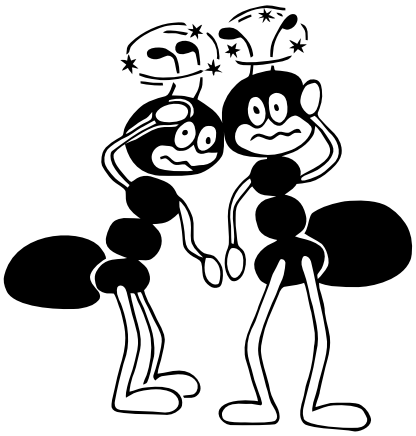
26-1-7 Mravenci

Pokusme se soustředit na srážku určitých dvou mravenců. Lze si všimnout, že pokud si je neoznačíme, vypadá situace tak, jako by se míjeli a k žádné srážce nedošlo. Mravenec, který spadne z klacku poslední, je tedy ten, který je nejvzdálenější od okraje klacku, ke kterému je otočený. Pozor, takový mravenec je vždy aspoň jeden, ale můžou být i dva!

U druhé části nám přibylo několik těžkostí. Nemůžeme už totiž zanedbat označení mravenců. Nahlédneme ale, že mravenci se na klacku nemohou přeskakovat. Speciálně je tedy prvních n mravenců spadlých z klacku na levé straně totožných s těmi n mravenci, kteří jsou na začátku nejblíže levému konci klacku. Pro pravou stranu platí analogické tvrzení.

Z první části již víme, do jakého směru je na začátku orientován mravenec, jenž spadne jako poslední (pro jednoduchost dále předpokládejme, že doleva, jinak analogicky). Nyní zjistíme počet mravenců, kteří na začátku míří doleva. Řekněme, že je jich k. Pak mravenec, který z klacku spadne jako poslední, je k-tý mravenec zleva v pořadí, v jakém stojí mravenci na začátku.

Jan Bok



26-1-8 Turingova strojovna

První tři úkoly byly snadné, čtvrtý trochu těžší. Ukázalo se ale, že všechny čtyři skrývají nečekané hlubiny. Začneme proto snadnými řešeními a pak se spolu vydáme na cestu do hlubin.

Úkol 1

Nejprve si všimneme, že každá neprázdná správně uzávorkovaná posloupnost obsahuje po sobě jdoucí dvojici `()`. Smazáním této dvojice vytvoříme jiné správné uzávorkování, v němž opět najdeme takovou dvojici, a tak dále, až posloupnost zredukujeme na prázdnou. Naopak začneme-li se špatným uzávorkováním, smazáním `()` z něj nikdy nevytvoříme správné. Zjistili jsme, že správná uzávorkování jsou právě ta, která lze zredukovat na prázdnou posloupnost.

Přesně o to se bude pokoušet náš stroj. Pracovat bude nad abecedou $\{$`(`, `)`, `*`$\}$ a jeho program bude vypadat následovně:

stav/znak	`(`	`)`	`*`	␣
S	$(\texttt{(}, \rightarrow, S)$	$(\texttt{*}, \leftarrow, P)$	$(\texttt{*}, \rightarrow, S)$	$(\text{␣}, \leftarrow, K)$
P	$(\texttt{*}, \rightarrow, S)$	—	$(\texttt{*}, \leftarrow, P)$	NE
K	NE	—	$(\texttt{*}, \leftarrow, K)$	ANO

Začne ve stavu S, bude procházet řetězcem zleva doprava a hledat první pravou závorku. Jakmile ji najde, přepíše ji na `*` a přepne se do stavu P, v němž se bude vracet zpátky a hledat nejbližší levou. Tu také vyhvězdičkuje, načež se přepne opět do stavu S (všimněte si, že vlevo od aktuální pozice už žádné pravé závorky nejsou, takže se k nim není potřeba vracet). Až vstupní řetězec dojde, stroj přejde do stavu K, v němž bude kontrolovat, zda nezbyly nějaké nespárované levé závorky.

Časová složitost tohoto stroje je $\mathcal{O}(n^2)$, jelikož až řádově n-krát potřebujeme najít párovou závorku, což trvá až řádově n kroků. Kvadraticky dlouho bude počítat například na vstupu `(((...)))`. Kromě prostoru na pásce, kde byl napsán vstup, nepotřebuje žádnou další paměť.

Úkol 2 s lineární pamětí

Sledujme, jak probíhá výpočet stroje z předchozího úkolu. Vyhvězdičkovaná políčka pásky můžeme ignorovat, ta stroj vždy přeskakuje. Pak platí, že vlevo od aktuální pozice jsou samé levé závorky – to jsou ty, ke kterým jsme ještě nenašli pravou závorku do páru. Každá další závorka je buďto levá (a pak ji přeskočíme, čímž se přesune do levé části), nebo pravá (a tehdy naopak z levé části jednu levou závorku smažeme).

Jinými slovy levou část používáme jako zásobník dosud neuzavřených závorek. Na vícepáskovém stroji si ho můžeme uložit na samostatnou pásku. Tím zařídíme, že další znak v zásobníku bude dostupný v konstantním čase.

Program stroje bude vypadat takto:

$$\begin{aligned}
(S, \texttt{(}, \text{␣}) &\rightarrow ((\texttt{(}, \rightarrow), (\texttt{(}, \rightarrow), S)\\
(S, \texttt{)}, \text{␣}) &\rightarrow ((\texttt{)}, \rightarrow), (\text{␣}, \leftarrow), P)\\
(S, \text{␣}, \text{␣}) &\rightarrow ((\text{␣}, \bullet), (\text{␣}, \leftarrow), K)\\
(P, \alpha, \texttt{(}) &\rightarrow ((\alpha, \bullet), (\text{␣}, \bullet), S)\\
(P, \alpha, \text{␣}) &\rightarrow \text{NE}\\
(K, \text{␣}, \text{␣}) &\rightarrow \text{ANO}\\
(K, \text{␣}, \texttt{(}) &\rightarrow \text{NE}
\end{aligned}$$

Stroj začne ve stavu S. Na první pásce se bude pohybovat zleva doprava a postupně číst závorky ze vstupu. Na druhé pásce si bude udržovat zásobník otevřených závorek a hlava se v klidovém stavu bude nacházet vpravo od poslední uložené závorky.

Pokud stroj přečte levou závorku, uloží ji na zásobník.

Pokud přečte pravou závorku, přepne se do stavu P a zkontroluje, jestli na zásobníku má nějakou levou. Pakliže ano, odstraní ji a vrátí se do počátečního stavu. Pokud ne, závorkování není správné.

Skončí-li vstup, je ještě potřeba zkontrolovat, že na zásobníku nezůstala žádná neuzavřená závorka. O to se stará stav K.

Tento stroj pracuje v lineárním čase s délkou vstupu (každý znak zpracuje v konstantním čase) a spotřebuje lineárně mnoho prostoru na pracovní pásce.

Úkol 2 s logaritmickou pamětí

Zadání vybízelo k co nejmenší spotřebě paměti. Vskutku, předchozí řešení prostorem vysloveně plýtvá. Do zásobníku ukládá samé levé závorky, takže by úplně stačilo pamatovat si jejich počet ve dvojkové soustavě. Uložíme ho jako posloupnost znaků `0` a `1` na pracovní pásce. Nejnižší řád se bude nacházet vpravo a v klidovém stavu bude hlava stát na mezeře napravo od něj.

Obsluha vstupní pásky bude vypadat následovně:

$$\begin{aligned}
(S, \texttt{(}, \text{␣}) &\rightarrow ((\texttt{(}, \rightarrow), (\text{␣}, \leftarrow), I)\\
(S, \texttt{)}, \text{␣}) &\rightarrow ((\texttt{)}, \rightarrow), (\text{␣}, \leftarrow), D)\\
(S, \text{␣}, \text{␣}) &\rightarrow ((\text{␣}, \bullet), (\text{␣}, \leftarrow), K)
\end{aligned}$$

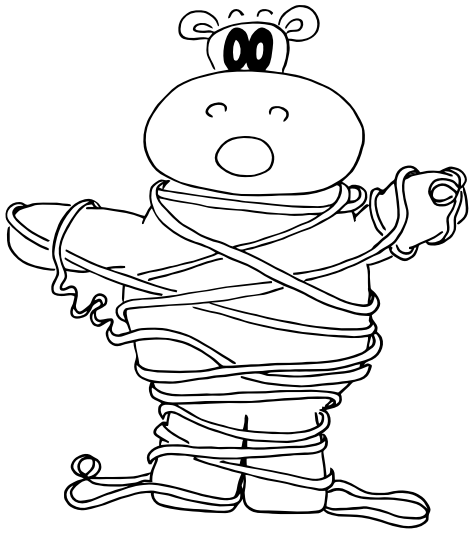
Stavy I, D a K znamenají „zvyš počítadlo o 1“ (inkrementace), „sniž počítadlo o 1“ (dekrementace) a „závěrečná kontrola, zda počítadlo je 0“. V nich už se budeme zabývat jen pracovní páskou s počítadlem, takže je popíšeme jako jednopáskový Turingův stroj. Navíc jsme přidali stav Z, který slouží k návratu hlavy doprava.

stav/znak	`0`	`1`	␣
I	$(\texttt{1}, \rightarrow, Z)$	$(\texttt{0}, \leftarrow, I)$	$(\texttt{1}, \rightarrow, Z)$
D	$(\texttt{1}, \leftarrow, D)$	$(\texttt{0}, \rightarrow, Z)$	NE
K	$(\texttt{0}, \leftarrow, K)$	NE	ANO
Z	$(\texttt{0}, \rightarrow, Z)$	$(\texttt{1}, \rightarrow, Z)$	$(\text{␣}, \bullet, S)$

Nyní spotřebujeme pouze logaritmické množství prostoru, neboť dvojkové číslo v rozsahu 0 až n zapíšeme pomocí $\lfloor \log_2 n \rfloor + 1$ bitů. Ovšem zvyšování a snižování počítadla trvá logaritmicky dlouho, takže jsme si časovou složitost pokazili na $\mathcal{O}(n \log n)$.

Úkol 3

Pokud si můžeme dovolit přidávat pásky, vystačíme si dokonce s jediným stavem. Rozšíříme abecedu o tolik symbolů, kolik měl původní stroj stavů. Přidáme novou pásku, z níž budeme používat jen jediné políčko. Na něm budeme udržovat informaci o tom, jaký stav původního stroje právě simulujeme. A aby se nový stroj správně rozběhl, určíme, že prázdné políčko odpovídá počátečnímu stavu původního stroje.



Úkol 4

Na jednopáskovém stroji se nabízí rozšířit abecedu na uspořádané dvojice (z, s), kde z je znak původní abecedy a s stav původního stroje. Jenže: pokud posuneme hlavu na sousední políčko, musíme tam přenést informaci o stavu, která byla zakódovaná do stávajícího políčka. To nejde udělat najednou (protože do stavu nového stroje zakódujeme jen 1 bit informace), ale s trochou šikovnosti poskytují dva stavy dost manévrovacího prostoru na to, abychom informaci přenesli po částech.

Stavy nového stroje nazveme X a Y. Abecedu rozšíříme na trojice (z, s, m), přičemž z a s budou odpovídat znaku a stavu původního stroje (stavy očíslujeme) a m bude *mód,* na němž bude záviset, co zrovna X a Y znamenají. Módů budeme rozeznávat pět: *klid, vysílání doprava, vysílání doleva, příjem zprava, příjem zleva.*

Představme si, že nový stroj právě odsimuloval jednu instrukci starého stroje. Na aktuálním políčku změnil s i z a teď se potřebuje posunout na sousední políčko, řekněme doprava. Udělá toto:

- Na aktuální políčko zapíše mód *vysílání doprava,* přepne se do stavu X a posune hlavu doprava.
- Sousední políčko bylo v módu *klid,* takže si stav X vyloží jako požadavek, který přišel zleva. Nastaví $s = 0$, přejde do módu *příjem zleva* a ve stavu X se posune zpět doleva.
- Nyní je předávání informací nastartováno. Vysílací políčko pokaždé sníží své s o 1 a dokud nevyjde 0, přesouvá se na přijímací políčko ve stavu X. Přijímací políčko zvýší své s o 1 a vrátí hlavu zpět (stále stav X). Pokračujeme v předávání.
- Jakmile vysílací políčko dopočítá do nuly, přepne svůj mód na *klid* a posune hlavu na přijímací políčko, tentokrát ve stavu Y. Podle toho přijímací políčko pozná, že přenos je u konce, a odsimuluje další instrukci původního stroje.

Pokud chceme přenášet informaci doleva místo doprava, postupujeme obdobně, jen v prvním kroku použijeme stav Y, podle čehož přijímací políčko pozná, že přenášíme zprava.

Každou instrukci původního stroje tedy umíme odsimulovat konstantně mnoha instrukcemi stroje nového.

Úkol 4: start výpočtu

Právě předvedené řešení 4. úkolu má jeden malý, leč podstatný háček: jak se vlastně celý výpočet rozběhne? Potřebujeme přeci, aby byl ve znaku na prvním políčku pásky zakódován počáteční stav S_0 původního stroje.

Jak to zařídit? Máme možnost určit pro každý znak původní abecedy, jaká trojice mu bude odpovídat v nové abecedě, a také si můžeme vybrat počáteční stav nového stroje.

Hned se nabízí zapisovat znaky původní abecedy jako trojice (z, S_0, klid). Jenže ani pro počáteční stav X, ani pro Y to nedopadne dobře: stroj se bude snažit kopírovat stav ze sousedního políčka, které na to vůbec není připraveno. Tak raději necháme nový stroj, ať svůj výpočet zahájí zapsáním stavu S_0. Jak ale pozná, kdy to má udělat?

Půjdeme na to menší oklikou. Nejprve si rozmyslíme, že každý Turingův stroj můžeme předělat tak, aby nikdy nevyužíval políčka pásky nalevo od počáteční polohy hlavy (tedy aby jeho páska byla jen jednostranně nekonečná). Zařídíme to „přeložením pásky napůl“. Políčka původní pásky si očíslujeme $\ldots, -3, -2, -1, 1, 2, 3, \ldots$ a na i-té políčko nové pásky uložíme uspořádanou dvojici znaků z původních políček i a $-i$.

Předělaný stroj bude simulovat instrukce původního stroje a navíc si bude ve svém stavu pamatovat, zda se nachází v kladné či záporné části původní pásky. Podle toho bude používat buď první, anebo druhou složku dvojice a případně obracet směr pohybu hlavy. Navíc si na políčko 1 umístíme značku, abychom poznali, že jsme přešli přes rozhraní kladné a záporné části.

Tato transformace zpomalí výpočet pouze konstanta-krát a má jeden příjemný důsledek, kterého vzápětí využijeme: vstoupíme-li na jakékoli políčko poprvé, je to vždy zleva.

Nyní se vraťme zpět k redukci počtu stavů. Každý znak z původní abecedy zakódujeme jako trojici (z, S_0, init). Nový mód *init* se chová stejně jako *klid* a navíc říká, že jsme na políčku poprvé. Proto víme, že během výpočtu nového stroje nemůžeme na takové políčko přijít ve stavu Y (ten by totiž znamenal „jdeme kopírovat zprava“). Y tedy prohlásíme za počáteční stav nového stroje a kombinaci mód *init* + stav Y využijeme k rozjezdu stroje.

Heuréka, problém vyřešen. Každý jednopáskový Turingův stroj umíme upravit tak, aby počítal totéž (až na změnu abecedy), zpomalil se jenom konstanta-krát a vystačil si přitom s pouhými dvěma stavy. (Přesněji řečeno, předvedli jsme to pro stroje, které odpovídají stavem ANO nebo NE. Pokud by výstup vydávaly na pásce, museli bychom ještě na závěr výpočtu pásku „vyčistit“ a překódovat zpět do vstupní abecedy. Ale to už je maličkost.)

Pro přehlednost ještě ukážeme tabulku, co který stav znamená ve kterém módu:

mód	**stav** X	**stav** Y
init	začne příjem zleva	start stroje
klid	začne příjem zleva	začne příjem zprava
vysílání	chci pokračování	—
příjem	zvýšit číslo stavu	konec vysílání

Poznámka: Dodejme ještě, že inicializaci lze provést i jinak. Využijeme toho, že jsme ve vysílacích módech nepotřebovali stav Y. Pojďme i tam stavem rozlišovat, zda jsme přišli zleva nebo zprava. Navíc máme možnost v každém módu poprohazovat, co znamená X a co Y. Pak zafunguje následující trik.

Na počáteční políčko přijdeme ve stavu X, takže si políčko myslí, že má přijímat zleva. Požádá proto políčko vlevo od sebe o pokračování vysílání. Jenže levé políčko o žádném vysílání neví, takže to interpretuje jako žádost o příjem zprava. Přejde tedy doprava s žádostí o vysílání. Můžeme zařídit, aby si pravé políčko tuto žádost vyložilo jako konec vysílání, čili přijalo stav 0. Ten zpracujeme speciálně: skočíme doleva a také předáme konec vysílání. I levé políčko ukončí příjem přijetím stavu 0, ale umí to rozlišit, protože přišel zprava, takže ho může zpracovat jinak speciálně: zahájením přenosu skutečného počátečního stavu stroje doprava. Kouzlo se zdařilo.

Zpět k úkolu 1: rychlejší řešení

Trik s počítadlem z úkolu 2 se dá využít i na jednopáskovém stroji. Jen si musíme dát pozor, kam počítadlo uložíme: pokud na začátek pásky (před vstup), budeme ke konci vstupu potřebovat spoustu kroků na přesuny mezi vstupem a počítadlem; počítadlo za koncem vstupu se bude chovat podobně špatně na začátku výpočtu. Kam tedy? Pořídíme si počítadlo stěhovavé: bude umístěno těsně před dosud nezpracovanou částí vstupu a po zpracování každé další závorky ho celé přestěhujeme o jednu pozici doprava.

Program stroje bude vypadat následovně:

stav/znak	`(`	`)`	`0`	`1`	␣
S	$(\texttt{(}, \leftarrow, I)$	$(\texttt{)}, \leftarrow, D)$	—	—	$(\sqcup, \leftarrow, K)$
I	—	—	$(\texttt{1}, \leftarrow, L)$	$(\texttt{0}, \leftarrow, I)$	$(\texttt{1}, \leftarrow, L)$
D	—	—	$(\texttt{1}, \leftarrow, D)$	$(\texttt{0}, \leftarrow, L)$	NE
L	—	—	$(\texttt{0}, \leftarrow, L)$	$(\texttt{1}, \leftarrow, L)$	$(\sqcup, \rightarrow, c_s)$
c_s	—	—	$(\sqcup, \rightarrow, c_0)$	$(\sqcup, \rightarrow, c_1)$	—
c_0	$(\texttt{0}, \rightarrow, S)$	$(\texttt{0}, \rightarrow, S)$	$(\texttt{0}, \rightarrow, c_0)$	$(\texttt{0}, \rightarrow, c_1)$	—
c_1	$(\texttt{1}, \rightarrow, S)$	$(\texttt{1}, \rightarrow, S)$	$(\texttt{1}, \rightarrow, c_0)$	$(\texttt{1}, \rightarrow, c_1)$	—
K	—	—	$(\texttt{0}, \leftarrow, K)$	NE	ANO

Stroj opět začíná ve stavu S. Jakmile zmerčí levou závorku, posune se těsně před aktuální pozici, kde je uloženo počítadlo, a začne ho inkrementovat. Přitom setrvává ve stavu I, dokud dochází k přenosu do vyšších řádů. Jakmile přenosy ustanou, přejde do L a pohybuje se směrem k levému okraji počítadla. Nakonec pomocí stavů c_0, c_1 a c_s celé počítadlo přesune o znak doprava a vrátí se zpět do S.

Podobně reaguje na pravou závorku, jen k tomu používá dekrementovací stav D. Na konci výpočtu jako obvykle pomocí stavu K zkontroluje, že počítadlo vyšlo nulové.

Časová složitost tohoto řešení je $\mathcal{O}(n \log n)$, protože pro každý znak vstupu strávíme $\mathcal{O}(\log n)$ kroků zvyšováním či snižováním a následně přesunem logaritmicky dlouhého počítadla. Paměti zabíráme stále $\mathcal{O}(n)$.

Zpět k úkolu 2: trocha naděje

Také vám vrtá hlavou, jestli by nešlo nějak zkřížit naše dvě řešení druhého úkolu a dosáhnout současně lineárního času a logaritmické paměti? Jistá naděje tu je. Pozorujme, jak se mění číslice počítadla, když ho opakovaně inkrementujeme:

$$0000 \to 000\mathbf{1} \to 00\mathbf{10} \to 001\mathbf{1} \to 0\mathbf{100} \to 010\mathbf{1} \to 01\mathbf{10} \to$$
$$\to 011\mathbf{1} \to \mathbf{1000} \to 100\mathbf{1} \to 10\mathbf{10} \to 101\mathbf{1} \to \dots$$

Pro přehlednost jsme změněné číslice vyznačili tučně.

Pokaždé se jedničky na konci čísla změní na nuly a nula před nimi na jedničku. Nebo jinak: jedna jednička vznikne a možná nějaké zaniknou. Pokud provedeme m inkrementů, vzniklo celkem m jedniček. Každá z nich zanikla nejvýš jednou, takže všech zániků dohromady je také nejvýš m. Všech m inkrementů tedy zabralo čas $\mathcal{O}(m)$.

Tehdy říkáme, že jeden inkrement má konstantní *amortizovanou časovou složitost.* Hezky se to popisuje pomocí *penízkové metody.* Jelikož čas jsou jak známo peníze, zavedeme mezi nimi směnný kurs. Jeden *penízek* bude představovat dostatek času na provedení jedné operace našeho stroje: tedy přepsání nuly na jedničku či opačně, včetně případného pohybu hlavy.

Nyní prohlásíme, že na provedení jednoho inkrementu po uživateli chceme 2 penízky. Jeden z nich použijeme na vytvoření jedničky, druhý dáme nově vzniklé jedničce do vínku a ona si z něj časem zaplatí své smazání. Za m inkrementů tedy zaplatíme $2m$ penízků a každou operaci stroje jsme naúčtovali některému z inkrementů.

Dokud tedy při kontrole závorek potkáváme jen ty levé, časová složitost stroje je lineární. Jenže… pravá závorka způsobí snížení počítadla a to nám celý elegantní amortizační argument zboří: počítadlo může libovolně dlouho střídat hodnoty `0111...1` a `1000...0`, což nutně zabere logaritmický čas.

Úkol 2: dvojková soustava vrací úder

Tak snadno se přeci nevzdáme. Dvojkovou soustavu rozšíříme, aby byla *vyvážená.* Vedle číslic 0 a 1 použijeme navíc -1 (pro zkrácení zápisu budeme místo 1 psát `+` a místo -1 prostě `-`). Váhy řádů budou stále mocniny dvojky, takže číslo $c_k c_{k-1} \dots c_1 c_0$ bude mít hodnotu $\sum_{i=0}^{k} 2^i \cdot c_i$.

Dvojkový zápis čísla funguje i ve vyvážené dvojkové soustavě, ale totéž číslo může mít i jiné zápisy. Například 6 (`110` dvojkově) se dá zapsat jako `++0`, ale i `+-+0` nebo `+-0-0`, jakož i mnoha dalšími způsoby.

Přesto z první číslice poznáme, zda je číslo kladné nebo záporné. Nechť první číslice je `+` a má váhu 2^k. Potom ani kdyby byly všechny ostatní číslice záporné, nepřispějí dohromady tolik, aby se celé číslo vynulovalo, nebo dokonce přehouplo do záporna. Platí totiž $2^0 + 2^1 + \dots + 2^{k-1} = 2^k - 1$. Podobně je-li první číslice `-`, musí být číslo nutně záporné.

Z toho speciálně plyne, že jediné možnosti, jak zapsat nulu, jsou řetězce číslic `0`. Cokoliv jiného je buď kladné, nebo záporné.

Inkrementování čísla provedeme takto: půjdeme zprava doleva. Pokud potkáme `0`, změníme ji na `+` a zastavíme se. Potkáme-li `-`, změníme ho na `0` a zastavíme se. Narazíme-li na `+`, přepíšeme ho na `0` a stejně jako v klasické dvojkové soustavě provedeme přenos do vyššího řádu (o číslici vlevo).

Dekrementování je symetrické: z `0` uděláme `-`, z `+` uděláme `0`, z `-` vytvoříme `0` a přenos.

Posloupnost inc, inc, inc, inc, dec, dec, inc tedy projde hodnotami `0`, `+`, `+0`, `++`, `+00`, `+0-`, `+-0`, `+-+`.

Nyní nahlédneme, že inkrementování i dekrementování má konstantní amortizovanou složitost. Penízky budeme tentokrát přiřazovat všem nenulovým číslicím. Inkrementování si nechá od uživatele zaplatit 2 penízky. Pokud přepíše `0` na `+`, zaplatí to z uživatelova penízku a ten druhý dá do vínku vzniklému `+`. Změní-li `+` na `0`, zaplatí to z penízku toho `+` a pokračuje ve výpočtu. A pokud přepíše `-` na `0`, zaplatí to z penízku toho `-` a dva uživatelovy penízky může prohýřit. Dekrementování se chová obdobně, též si vystačí se dvěma penízky.

K vyřešení úlohy použijeme Turingův stroj, který bude fungovat obdobně jako naše předchozí řešení s dvojkovým počítadlem, jen použijeme vyváženou dvojkovou soustavu. Popíšeme jen obsluhu počítadla, zbytek stroje zůstane stejný. V klidovém stavu budeme opět udržovat hlavu vpravo od počítadla. Navíc aby se nám snadno testovala nulovost počítadla, budeme nevýznamné nuly ze začátku čísla při každé příležitosti mazat.

Program stroje vypadá takto:

stav/znak	`0`	`+`	`-`	␣
I	$(+, \rightarrow, Z)$	$(0, \leftarrow, I)$	$(0, \leftarrow, N)$	$(+, \rightarrow, Z)$
D	$(-, \rightarrow, Z)$	$(0, \leftarrow, N)$	$(0, \leftarrow, D)$	NE
Z	$(0, \rightarrow, Z)$	$(+, \rightarrow, Z)$	$(-, \rightarrow, Z)$	$(␣, \bullet, S)$
N	$(0, \rightarrow, Z)$	$(+, \rightarrow, Z)$	$(-, \rightarrow, Z)$	$(␣, \rightarrow, N')$
N'	$(␣, \rightarrow, Z)$	—	—	—
K	NE	NE	NE	ANO

Stav I je jako obvykle inkrementovací, stav D dekrementovací. V obou stroj upravuje počítadlo tak dlouho, dokud dochází k přenosu. Pak se přepne do stavu Z, v němž se vrací na konec pásky. Pokud na pásku zapíše `0`, odskočí si ještě do stavu N, v němz zkontroluje, zda tato nula neleží na začátku čísla, a případně ji smaže. Stav K slouží k závěrečné kontrole nulovosti počítadla – jelikož nevýznamné nuly průběžně mažeme, postačí testovat prázdnost pásky.

Dosáhli jsme tedy lineární časové a logaritmické prostorové složitosti. Na závěr dodejme, že logaritmický prostor je skutečně zapotřebí, ale důkaz je trochu pracnější a do okraje této stránky by se nevešel ☺

Úkol 2: kočkopsí řešení

Ukážeme ještě jedno optimální řešení druhého úkolu. Je technicky pracnější, ale myšlenkově prostší: Šikovně zkřížíme první řešení (počítadlo v jedničkové soustavě, lineární čas, lineární paměť) s druhým (dvojková soustava, čas $\mathcal{O}(n \log n)$, paměť $\mathcal{O}(\log n)$).

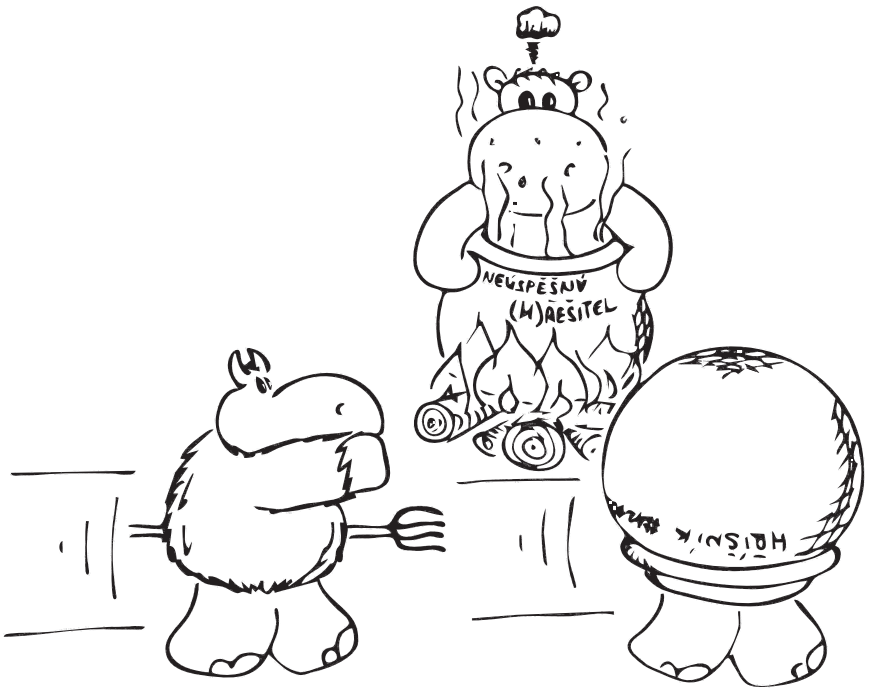
Opět budeme udržovat počítadlo rozdílu levých a pravých závorek. Počítadlo bude dvojkové, ale budeme ho aktualizovat po skocích. Nejprve spočítáme $\ell = \lceil \log_2 n \rceil$.

Vstup budeme zpracovávat po blocích velikosti ℓ. Pro každý blok budeme udržovat počítadlo v jedničkové soustavě. Zajímat nás bude jeho hodnota na konci bloku a také nejnižší záporná hodnota během bloku. To vše zjistíme v čase $\mathcal{O}(\ell)$ a prostoru taktéž $\mathcal{O}(\ell)$.

Hodnotu počítadla na konci bloku převedeme do dvojkové soustavy a přičteme ji ke globálnímu počítadlu. Před tím ještě ověříme, že nejnižší záporná hodnota nepřesáhla předchozí hodnotu globálního počítadla. Všechny tyto operace stihneme v $\mathcal{O}(\ell)$ – tolik bitů mají dvojková čísla, s nimiž pracujeme. Převod z jedničkové do dvojkové soustavy zařídíme postupným přičítáním jedničky, které, jak už víme, trvá $\mathcal{O}(1)$ amortizovaně.

Každý blok tedy zpracujeme v čase $\mathcal{O}(\ell)$ a prostoru $\mathcal{O}(\ell)$. Všech $\mathcal{O}(n/\ell)$ bloků v čase $\mathcal{O}(n)$ a prostoru opět $\mathcal{O}(\ell) = \mathcal{O}(\log n)$.

Martin „Medvěd“ Mareš



Snad se vám při čtení řešení neuvařila hlava ☺

Hodně štěstí i do dalších sérií!

Výsledková listina první série dvacátého šestého ročníku KSP

	řešitel	*škola*	*ročník*	*sérií*	*2611*	*2612*	*2613*	*2614*	*2615*	*2616*	*2617*	*2618*	*série*	*celkem*
0.					8	9	8	10	10	11	8	12	52,0	52,0
1.	Martin Raszyk	G_Karvina	4	16	8	9	8	7	10	11		10	46,8	46,8
2.	Václav Rozhoň	GJirsíkaČB	3	2	8	9	8		9			12	46,7	46,7
3.	Jan Špaček	G_Wicht	3	1	8	9	8	4			8	11	44,8	44,8
4.	Marek Černý	G_Chrudim	3	1	8	4	8	6,5				8	42,3	42,3
5.	Jan Pokorný	G_Bučovice	2	2	7	8	8				7	7	42,1	42,1
6.	Jakub Svoboda	GKomHavíř	4	6	8	6,5	0		8	0	3	8	38,4	38,4
7.	Matej Lieskovský	GOmskPha	4	11	8	7			7		8	8	38,0	38,0
8.	Štěpán Trčka	GSlavičín	3	8	8		8	10	3			5	36,1	36,1
9.	Michal Korbela	GJJesen	4	1	6		8	8		0,5		6	35,8	35,8
10.	Jakub Maroušek	G_Písek	4	6	8	9	8	3				2,5	33,5	33,5
11.	Richard Hladík	GOAMarLaz	1	6	8		8	8				6	32,9	32,9
12.	Štěpán Hojdar	GJírovcČB	4	6	8	8,5	8					6	32,8	32,8
13.	Aneta Šťastná	GOmskPha	4	7	5	9	8				3	3	31,4	31,4
14.	Anna Steinhauserová	GDačice	4	1	6	6			8		3		30,5	30,5
15.	Jakub Zárybnický	GTomkovaOL	3	1		2,5	4	2,5	2	1		5	30,0	30,0
16.	Jan Knížek	G_Strakon	3	10	6		8		4		8	2,5	29,4	29,4
17.	Antonín Češík	SPSE_Pard	4	1			8				7	6	25,3	25,3
18.	Jonatan Matějka	SŠP_ČB	4	15	7		8	8				5	25,2	25,2
19.	Filip Bialas	GOpatovPHA	1	1	7	7	8						24,3	24,3
20.	Václav Volhejn	GKepleraPH	1	6	8	7,5	8						24,2	24,2
21.	Dorian Řehák	GCoubTábor	3	1			5	3				6	22,5	22,5
22.	Jan Pavlovský	GJiM	4	1	6	5	4						21,3	21,3
23.–24.	Marek Dobranský	GHorMichal	4	6	4		8					5	20,3	20,3
	Dominik Roháček	SPŠLegioJI	4	2	3		0	3				6	20,3	20,3
25.	Michal Hloušek	GNadŠtolPH	1	1	4	1,5	4						16,3	16,3
26.	Michal Punčochář	GJírovcČB	4	11	8		8						16,0	16,0
27.	Antonín Teichmann	GJeronýmLI	4	1	4	8							15,2	15,2
28.–32.	Jan-Sebastian Fabík	GJarošeBO	4	9			8						8,0	8,0
	Dalimil Hájek	GKepleraPH	3	11			8						8,0	8,0
	Ondřej Hübsch	GArabskáPH	4	19			8						8,0	8,0
	Adam Španěl	ArcibisGPH	2	1							8		8,0	8,0
	Radovan Švarc	G_ČTřebová	3	3			8						8,0	8,0
33.	Petro Kostyuk	GEBenešeKL	4	1			4						6,4	6,4
34.	Jan Horešovský	GMěl	4	2			4						6,2	6,2
35.	Michal Martinek	GHavPodl	3	1				3					6,0	6,0