

Programátorské kuchařky



matfyzpress

VYDAVATELSTVÍ
MATEMATICKO-FYZIKÁLNÍ FAKULTY
UNIVERZITY KARLOVY V PRAZE

BÖHM, LÁNSKÝ, VESELÝ A KOLEKTIV

Programátorské kuchařky

matfyzpress

Praha 2011

Vydáno pro vnitřní potřebu fakulty.
Publikace není určena k prodeji.

ISBN 978-80-7378-181-1

Úvod


Kuchařka je krátký učební text, který *Korespondenční seminář z programování* (KSP) posledních zhruba deset let vydává. Má svým čtenářům, zpravidla středoškolákům dychtícím po programování, doplnit znalosti potřebné k vyřešení úloh, které seminář svým řešitelům pravidelně předkládá.

Jako taková si kuchařka neklade nároky na úplnost. Jejím účelem je srozumitelně nabídnout zajímavý, aplikovatelný a stravitelně veliký kus vědění. Postupem času se stalo, že kuchařky pokryly velkou část úvodního kurzu algoritmizace, stále však je potřeba jejich soubor, tuto knihu, chápat spíše jako čítanku než jako systematickou učebnici.

Abychom kuchařkám zachovali kontext, vybrali jsme z historie semináře vhodné úlohy a pod většinu kuchařek jednu či dvě umístili. Stejně jako naši řešitelé si tak můžete po přečtení studijního textu získané znalosti zkusit aplikovat. Vzorová řešení úloh od organizátorů KSP potom naleznete na konci knížky.

Najdete-li chybu, či napadne-li vás jakékoliv vylepšení uvedených textů, napište nám prosím na adresu ksp@mff.cuni.cz. Seznam dosud nalezených chyb a aktuální verze kuchařek je na stránce <http://ksp.mff.cuni.cz/kucharky/>.

Symbol

 V knize je systematicky využívána jediná značka, variace na klasický Bourbakiho symbol nebezpečné zatáčky. Varuje před matematicky obtížnější pasáží.

Literatura

V průběhu výkladu se občas odkážeme na další literaturu a zde předkládáme její přehled. Komentář necht' milému čtenáři poslouží k úvaze nad tím, co bude číst, až odloží *kuchařky*.

- Didaktější a učesanější přístup k úvodu do algoritmizace nabízí klasická publikace Pavla Töpfera *Algoritmy a programovací techniky*, kterou budeme značit odkazem [Töpfer].
- Jemný úvod do diskrétní (tj. nespojité) matematiky, která tvoří jeden z teoretických základů matematické informatiky, najdete ve známé a do mnoha jazyků překládané knize Jiřího Matouška a Jaroslava Nešetřila *Kapitoly z diskrétní matematiky*: [Kapitoly].
- Systematický přístup ke grafům nabízí kniha Jiřího Demela *Grafy a jejich aplikace*: [Demel].
- Mnohá témata této knihy jsou ryze vysokoškolská a zajímavé texty najdete v nejrozličnějších skriptech. Na adrese <http://mj.ucw.cz/vyuka/> jsou k dispozici zápisy matfyzáckých přednášek Martina Mareše k předmětům *Algoritmy a datové struktury I a II*: [ADS] a [ADS2]. Na stránce <http://mj.ucw.cz/vyuka/ga/> lze stáhnout skriptíčka k přednášce *Grafové algoritmy*: [GrafAlg].

Na adrese <http://kam.mff.cuni.cz/~valla/kg.html> objevíte *Skriptíčka z kombinatoriky* Tomáše Vally a Jiřího Matouška: [Skriptíčka]. Obsahují některá složitější kombinatorická témata, která [Kapitoly] nepokryly.

- Z anglických knížek zmiňme relativně útlou a novou publikaci *Algorithms* od Dasgupty, Papadimitriou a Vaziraniho: [Algo]. Je čtivým a promyšleným úvodem do algoritmizace, obsahuje také mnoho cvičení a můžete si ji zdarma stáhnout z adresy <http://www.cs.berkeley.edu/~vazirani/algorithms.html>.

Introduction to Algorithms od Cormena, Leisersona, Rivesta a Steina [IntroAlg] je pak jistým opakem, jde totiž o obsáhlou bichli, která encyklopedicky zpracovává všechna základní témata.

- Další úlohy k procvičení vymyšlení algoritmů naleznete na stránkách našeho semináře: <http://ksp.mff.cuni.cz/>. Na adrese <http://ksp.mff.cuni.cz/vyzvy.html> je pak seznam dalších nejen programátorských soutěží.

Obsah

Složitost	6
Třídění	12
Binární vyhledávání	22
Halda	25
Grafy	30
Dijkstrův algoritmus	43
Minimální kostra	47
Rozděl a panuj	55
Dynamické programování	63
Vyhledávací stromy	73
Hešování	86
Řetězce a vyhledávání v textu	92
Rovinné grafy	101
Eulerovské tahy	107
Toky v sítích	112
Intervalové stromy	118
Těžké problémy	124
Řešení úloh	131

Složitost

Pokud řešíme nějakou programátorskou úlohu, často nás napadne více různých řešení a potřebujeme se rozhodnout, které z nich je „nejlepší“. Abychom to mohli posoudit, potřebujeme si zavést měřítko, podle kterých budeme různé algoritmy porovnávat. Nás u každého algoritmu budou zajímat dvě vlastnosti: čas, po který algoritmus běží, a paměť, kterou při tom spotřebuje.

Čas nebudeme měřit v sekundách (protože stejný program na různých počítačích běží rozdílnou dobu), ale v počtu provedených operací. Pro jednoduchost budeme předpokládat, že aritmetické operace, přiřazování, porovnávání apod. nás stojí jednotkový čas. Ona to není úplná pravda, tyto operace se ve skutečnosti přeloží na procesorové instrukce, které se teprve zpracovávají. Ale nám postačí vědět, že těch instrukcí bude vždy konstantní počet. A později se dozvíme, proč nám na takové konstantě nezáleží.

Množství použité paměti můžeme zjistit tak, že prostě spočítáme, kolik bajtů paměti náš program použil. Nám obvykle bude stačit menší přesnost, takže všechna čísla budeme považovat za stejně velká a velikost jednoho prohlásíme za jednotku prostoru.

Jak čas, tak paměť se obvykle liší podle toho, jaký vstup náš program zrovna dostal – na velké vstupy spotřebuje více času i paměti než na ty malé. Budeme proto oba parametry algoritmu určovat v závislosti na velikosti vstupu a hledat funkci, která nám tuto závislost popíše. Takové funkci se odborně říká *časová (případně paměťová, někdy též prostorová) složitost* algoritmu/programu.

U výběru algoritmu tedy bereme v potaz čas a paměť. Který z těchto faktorů je pro nás důležitější, se musíme rozhodnout vždy u konkrétního příkladu. Často také platí, že čím více času se snažíme ušetřit, tím více paměti nás to pak stojí kvůli chytré reprezentaci dat v paměti a různým vyhledávacím strukturám, o kterých se můžete dočíst v našich dalších kuchařkách.

Nás u valné většiny algoritmů bude nejdříve zajímat časová složitost a až poté složitost paměťová. Paměti mají totiž dnešní počítače dost, a tak se málokdy stane, že vymyslíme algoritmus, který má dokonalý čas, ale nestačí nám na něj paměť. Ale přesto doporučujeme dávat si na paměťová omezení pozor.

Jednoduché počítání složitosti

Nyní si na příkladu ukážeme, jak se časová a paměťová složitost dá určovat intuitivně, a pak si vše podrobně vysvětlíme.

Představme si, že máme danou posloupnost N celých čísel, ze které chceme vybrat maximum. Použijeme algoritmus, který za maximum prohlásí nejprve první číslo posloupnosti. Pak toto maximum postupně porovnává s dalšími čísly posloupnosti, a pokud je některé větší, učiní z něj nové maximum. Zapsat bychom to mohli třeba takto:


```

posl[1...N] = vstup
max = posl[1]
Pro i = 2 až N:
    Jestliže posl[i] > max:
        max = posl[i]
Vypiš max

```

Není těžké nahlédnout, že algoritmus provede maximálně $N - 1$ porovnání. Intuitivně časová složitost bude lineárně záviset na N , protože porovnání dvou čísel nám zabere „jednotkový čas“ a paměťová složitost bude také na N záviset lineárně, protože každé číslo z posloupnosti budeme uchovávat v paměti. Pokud bychom si nepamatovali celou posloupnost, ale vždy jen poslední přečtený člen, stačilo by nám jen konstantně mnoho proměnných, takže paměťová složitost by klesla na konstantní (nezávislou na N) a časová by zůstala stejná.

Jiný příklad: Mějme dané číslo K . Naším úkolem je vypsát tabulku všech násobků čísel od 1 do K :

```

Pro i = 1 až K:
    Pro j = 1 až K:
        Vypiš i*j a mezeru
    Přejdi na nový řádek

```

Tabulka má velikost K^2 a na každém jejím políčku strávíme jen konstantní čas. Proto časová složitost bude záviset na čísle K kvadraticky, tedy bude K^2 . Paměťová složitost bude buď konstantní, pokud hodnoty budeme jen vypisovat, anebo kvadratická, pokud si tabulku budeme ukládat do paměti. Můžeme si také všimnout, že tabulku nemusíme vypisovat celou, ale bude nám stačit jen její dolní trojúhelníková část – i tak budeme muset spočítat $(K \cdot K - K)/2 + K = K^2/2 + K/2$ hodnot, což je stále řádově kvadratické vzhledem ke K .

Než se pustíme do podrobnějšího vysvětlování, ještě si ukážeme tzv. „metodu kouknu a vidím“, kterou můžeme použít na určování časové složitosti u těch nejjednodušších algoritmů. Spočívá jen v tom, že se podíváme, kolik nejvíc obsahuje náš program vnořených cyklů. Řekněme, že jich je k a že každý běží od 1 do N . Potom za časovou složitost prohlásíme N^k .

Vzhledem k čemu budeme složitosti určovat?

Složitosti obvykle určujeme vzhledem k velikosti vstupu (počet čísel, případně znaků na vstupu). Tento počet si označme N . Časovou i paměťovou složitost pak vyjádříme vzhledem k tomuto N . To je vidět třeba na výběru maxima v předchozím textu.

Pokud by existovalo několik vstupů stejné velikosti, pro které náš algoritmus běží různě dlouho, bude časová složitost popisovat ten nejhorší z nich (takový, na kterém algoritmus poběží nejpomaleji). Stejně tak pro paměťovou složitost použijeme ten ze vstupů délky N , na který spotřebujeme nejvíce paměti. Dostaneme tzv. složitosti v nejhorším případě. Podrobněji si o tom povíme později.

Někdy se nám hodí určit složitost v závislosti na více než jedné proměnné. Pokud bychom například chtěli vypisovat všechny dvojice podstatného a přídavného jména

ze zadaného slovníku, strávíme tím čas, který bude záviset nejen na celkové velikosti slovníku, ale i na tom, kolik obsahuje podstatných a kolik přídavných jmen. Rozmyslete si, jaká složitost vyjde, pokud víte, že velikost slovníku je S , podstatných jmen je A a přídavných jmen B .

Častým příkladem, kde si velikost vstupu potřebujeme rozdělit do více proměnných, jsou algoritmy pracující s grafy (viz grafová kuchařka). V případě grafů obvykle vyjadřujeme složitost pomocí proměnných N a M , kde N je počet vrcholů grafu a M je počet jeho hran.

Ne vždy ale určujeme složitosti v závislosti na velikosti vstupů. Například pokud je velikost vstupu konstantní, složitost určíme vzhledem k hodnotám proměnných na vstupu. Třeba u příkladu s tabulkou násobků jsme složitost určili vzhledem k velikosti tabulky, kterou jsme dostali na vstupu. Jiným příkladem může být vypsání všech prvočísel menších než dané N .

Asymptotická složitost

V této části textu se budeme věnovat pouze časové složitosti. Všechna pravidla, která si řekneme, pak budou platit i pro paměťovou složitost.

U určování časové složitosti nás bude především zajímat, jak se algoritmy chovají pro velké vstupy. Mějme například algoritmus A o časové složitosti $4N$ a algoritmus B o složitosti N^2 . Tehdy je sice pro $N = 1, 2, 3$ algoritmus B rychlejší než A , ale pro všechna větší N ho už algoritmus A předběhne. Takže pokud bychom si měli mezi těmito algoritmy zvolit, vybereme si algoritmus A .

U složitosti nás obvykle nebude zajímat, jak se chová na malých vstupech, protože na těch je rychlý téměř každý algoritmus. Rozhodující pro nás bude složitost na maximálních vstupech (pokud nějaké omezení existuje) anebo složitost pro „hodně velké vstupy“. Proto si zavedeme tzv. *asymptotickou časovou složitost*.

Představme si, že máme algoritmus se složitostí $N^2/4 + 6N + 12$. Pod asymptotikou si můžeme představit, že nás zajímá jen nejvýznamnější člen výrazu, podle kterého se pak pro velké vstupy chová celý výraz. To znamená, že:

- Konstanty u jednotlivých členů můžeme škrtnout (např. $6n$ se chová podobně jako n). Tím dostáváme $N^2 + N + 1$.
- Pro velká N je $N + 1$ oproti N^2 nevýznamné, tak ho můžeme také škrtnout. Dostáváme tak složitost N^2 . Obecně škrtnáme všechny členy, které jsou pro dost velké N menší než nějaký neškrtnutý člen.

Tahle pravidla sice většinou fungují, ale škrtnat ve výpočtech přece nemůžeme jen tak. Proto si nyní zavedeme operátor \mathcal{O} (velké O), díky kterému budeme umět popsat, co přesně naše „škrtnání“ znamená, a používat ho korektně.

Definice: Mějme funkce $f : \mathbb{N} \rightarrow \mathbb{R}_0^+$ a $g : \mathbb{N} \rightarrow \mathbb{R}_0^+$. Řekneme, že $f \in \mathcal{O}(g)$, pokud $\exists n_0 \in \mathbb{N}$ a $\exists c \in \mathbb{R}^+$ tak, že $\forall n \geq n_0$ platí $f(n) \leq c \cdot g(n)$.

Nyní slovy: Mějme funkce f a g funkce z přirozených do nezáporných reálných čísel. Řekneme, že funkce f patří do třídy $\mathcal{O}(g)$, pokud existují konstanty n_0 a c takové, že f je pro dost velká n (totiž pro $n \geq n_0$) menší než $c \cdot g(n)$.

Někdy také píšeme, že $f = \mathcal{O}(g)$ nebo říkáme, že program má složitost $\mathcal{O}(g)$.

A zde je použití: $n^2/4 + 6n + 12 \in \mathcal{O}(n^2)$, protože například pro $c = 10$ platí pro všechna $n > 1$ (tedy $n_0 = 2$):

$$n^2/4 + 6n + 12 \leq 10n^2.$$

Pokud vám tento způsob nevyhovuje a více se vám líbí metoda pomocí „škrtání“, tak ji klidně používejte, akorát všude pište $\mathcal{O}(\dots)$. Někdy také říkáme, že se konstanty a méně významné členy v \mathcal{O} ztrácí.

Poznámky

- Uvědomte si, že je notace nadefinovaná tak, že omezuje shora, takže nejen, že platí $n^2/2 \in \mathcal{O}(n^2)$, ale také třeba $n^2/2 \in \mathcal{O}(n^5)$. Na první pohled je to neintuitivní – můžeme tvrdit, že *QuickSort* běží v $\mathcal{O}(2^n)$ a mít pravdu. Copak by byl problém definovat tak, aby její význam nebyl „funkce až na konstanty shora omezená touto funkcí“, ale „funkce je až na (rozdílné) konstanty shora i zdola omezená touto funkcí“?

Samozřejmě to vyjádřit můžeme! Definovat lze skoro cokoliv a existuje dokonce zaběhnutá notace $f \in \Theta(g)$, která tuto skutečnost (omezení zdola i shora) vyjadřuje. Samostatné omezení zdola (až na konstantu) se značí Ω a jeho definice je velmi podobná definici \mathcal{O} .

Nejhorší a průměrný případ

Opět si vše vysvětlíme jen na časové složitosti.

Velká část algoritmů běží pro různé vstupy stejné velikosti různou dobu. U takových algoritmů pak můžeme rozlišovat složitost v nejhorším případě (tu už známe), v nejlepším případě a třeba i průměrnou časovou složitost.

Vše si ukážeme na algoritmu *BubbleSort* (bublínkovém třídění), o kterém se můžete dočíst v kuchařce o třídících algoritmech. Funguje tak, že se dívá na všechny dvojice sousedních prvků, a kdykoliv je dvojice ve špatném pořadí, tak ji prohodí. Zde je pseudokód algoritmu:

BubbleSort(pole, N):

Opakuj:

 setříděno = 1

 Pro $i = 1$ až $N-1$:

 Jestliže $\text{pole}[i] > \text{pole}[i+1]$:

$p = \text{pole}[i]$

$\text{pole}[i] = \text{pole}[i+1]$

$\text{pole}[i+1] = p$

 setříděno = 0

Skonči, až bude setříděno = 1

Časová složitost v nejhorším případě činí $\mathcal{O}(N^2)$ – v každém průchodu vnějším cyklem nám největší neseřazená hodnota „probublá“ na začátek hodnot, které jsou již na seřazené správném místě, a ostatní se posunou o jednu pozici doleva. Rozmyslete

si, proč. Průchodů je proto nejvýše N a každý z nich trvá $\mathcal{O}(N)$. Tento nejhorší případ může doopravdy nastat, pokud necháme setřídít klesající posloupnost. Tam provedeme přesně N průchodů (v posledním se jen ověří, zda je posloupnost seřazená).

Naopak v nejlepším případě bude časová složitost pouze $\mathcal{O}(N)$. To nastane, pokud na vstupu dostaneme už setříděnou posloupnost. U té algoritmus pouze zkontroluje všechny dvojice a pak se ihned zastaví.

Průměrná časová složitost nám udává, jak dlouho náš algoritmus běží průměrně. Co to ale znamená, není snadné definovat ani spočítat. U třídícího algoritmu bychom mohli počítat průměr přes všechny možnosti, jak mohou být prvky na vstupu zamíchané (tedy přes všechny jejich permutace). To nám někdy může dát přesnější odhad chování algoritmu.

Zrovna u BubbleSortu a mnoha jiných algoritmů vyjde průměrná časová složitost stejně jako složitost v nejhorším případě. Jedním z nejznámějších příkladů algoritmu, který je v průměru asymptoticky lepší, je třídící algoritmus QuickSort (opět viz třídící kuchařka). Jeho průměrná časová složitost činí $\mathcal{O}(N \cdot \log N)$, zatímco v nejhorším případě může běžet až kvadraticky dlouho.

Často používané složitosti

Na závěr si ukážeme často se vyskytující časové složitosti algoritmů (ty paměťové jsou obdobné). Seřadili jsme je od nejrychlejších a ke každé připsali příklad algoritmu.

$\mathcal{O}(1)$ – *konstantní* (třeba zjištění, jestli je číslo sudé)

$\mathcal{O}(\log N)$ – *logaritmická* (binární vyhledávání); všimněte si, že na základu logaritmu nezáleží, protože platí $\log_a n = \log_b n / \log_b a$, takže logaritmy o různých základech se liší jen konstanta-krát, což se „schová do \mathcal{O} -čka“.

$\mathcal{O}(N)$ – *lineární* (hledání maxima z N čísel)

$\mathcal{O}(N \cdot \log N)$ – *lineárně-logaritmická* (nejlepší algoritmy na třídění pomocí porovnávání)

$\mathcal{O}(N^2)$ – *kvadratická* (BubbleSort)

$\mathcal{O}(N^3)$ – *kubická* (násobení matic podle definice)

$\mathcal{O}(2^N)$ – *exponenciální* (nalezení všech posloupností délky N složených z nul a jedniček; pokud je chceme i vypsát, dostaneme $\mathcal{O}(N \cdot 2^N)$)

$\mathcal{O}(N!)$ – *faktoriálová*, $N! = 1 \cdot 2 \cdot 3 \cdot \dots \cdot N$ (nalezení všech permutací N prvků, tedy třeba všech přesmyček slova o N různých písmenech)

Složitosti ještě často rozdělujeme na *polynomiální* a *nepolynomiální*. Polynomiální říkáme těm, které patří do $\mathcal{O}(N^k)$ pro nějaké k . Naopak nepolynomiální jsou ty, pro něž žádné takové k neexistuje.

Do polynomiálních algoritmů patří například i algoritmus se složitostí $\mathcal{O}(\log N)$. A to proto, že $\mathcal{O}(\log N) \subset \mathcal{O}(N)$ (každý algoritmus, který sebehne v čase $\mathcal{O}(\log N)$, sebehne i v $\mathcal{O}(N)$).

Nepolynomiální jsou z naší tabulky třídy $\mathcal{O}(2^N)$ a $\mathcal{O}(N!)$. Takové algoritmy jsou extrémně pomalé a snažíme se jim co nejvíce vyhýbat.

Pro představu o tom, jak se složitost projevuje na opravdovém počítači, se podíváme, jak dlouho poběží algoritmy na počítači, který provede 10^9 (miliardu) operací za sekundu. Tento počítač je srovnatelný s těmi, které dnes běžně používáme. Podívejme se, jak dlouho na něm poběží algoritmy s následujícími složitostmi:

funkce / $n =$	10	20	50	100	1 000	10^6
$\log_2 n$	3.3 ns	4.3 ns	4.9 ns	6.6 ns	10.0 ns	19.9 ns
n	10 ns	20 ns	30 ns	100 ns	1 μ s	1 ms
$n \cdot \log_2 n$	33 ns	86 ns	282 ns	664 ns	10 μ s	20 ms
n^2	100 ns	400 ns	900 ns	100 μ s	1 ms	1 000 s
n^3	1 μ s	8 μ s	27 μ s	1 ms	1 s	10^9 s
2^n	1 μ s	1 ms	1 s	10^{21} s	10^{292} s	$\approx \infty$
$n!$	3 ms	10^9 s	10^{23} s	10^{149} s	10^{2558} s	$\approx \infty$

Pro představu: 1 000 s je asi tak čtvrt hodiny, 1 000 000 s je necelých 12 dní, 10^9 s je 31 let a 10^{18} s je asi tak stáří Vesmíru. Takže nepolynomiální algoritmy začnou být velmi brzy nepoužitelné.

Karel Tesař a Martin Mareš

Třídění

Pojem *třídění* je možná maličko nepřesný, často se však používá. Nehodláme data (čísla, řetězce a jiné) rozdělovat do nějakých *tříd*, ale přerovnat je do správného *pořadí*, od nejmenšího po největší – ať už pro nás „větší“ znamená jakékoli uspořádání.

Se seřazenými údaji se totiž mnohem lépe pracuje. Potřebujeme-li v nich kupříkladu vyhledávat, zvládneme to v uspořádaném stavu mnohem rychleji, jak dosvědčí další kuchařka a běžná zkušenost. Takové uspořádávání dat je proto denním chlebem každého programátora, a tak není divu, že třídící algoritmy jsou jedny z nejstudovanýchějších.

Obvykle třídíme exempláře datové struktury typu pascalského záznamu (v jiných jazycích struktury, třídy apod.). V takové datové struktuře bývá obsažena jedna význačná položka, *klíč*, podle které se záznamy řadí. Malinko si náš život zjednodušíme a budeme předpokládat, že třídíme záznamy obsahující pouze klíč, který je navíc celočíselný – budeme tedy třidit pole celých čísel. Vzhledem k počtu tříděných čísel N pak budeme vyjadřovat časovou (a paměťovou) složitost jednotlivých algoritmů, které si předvedeme.

Dodejme ještě, že se také studují případy, kdy je tříděných dat tolik, že se všechna naráz nevejdou do paměti. Tehdy by nastoupily takzvané *vnější* třídící algoritmy. Ty dovedou zacházet s daty uloženými třeba na disku a přizpůsobit své chování jeho vlastnostem. Zejména tomu, že diskům svědčí spíše sekvenční přístup k datům, zatímco neustálé přeskakování z jednoho konce souboru na druhý je pomalé. Ostatně, i u našeho *vnitřního* třídění na *lokalitě přístupů* díky existenci procesorové cache trochu záleží. Toto vše si ale v naší úvodní kuchařce odpustíme.

Přímé metody

Nejjednodušší třídící algoritmy patří do skupiny *přímých metod*. Jsou krátké, jednoduché a třídí přímo v poli (nepotřebujeme pole pomocné). Tyto algoritmy mají většinou časovou složitost $\mathcal{O}(N^2)$. Z toho vyplývá, že jsou použitelné tehdy, když tříděných dat není příliš mnoho. Na druhou stranu pokud je dat opravdu málo, je zbytečně složité používat některý z komplikovanějších algoritmů, které si předvedeme později.

Třídění přímým výběrem (SelectSort) je založeno na opakovaném vybírání nejmenšího čísla z dosud neseřazených čísel. Nalezené číslo prohodíme s prvkem na začátku pole a postup opakujeme, tentokrát s nejmenším číslem na indexech $2, \dots, N$, které prohodíme s druhým prvkem v poli. Poté postup opakujeme s prvky s indexy $3, \dots, N$ atd. Je snadné si uvědomit, že když takto postupně vybíráme minimum z menších a menších intervalů, setřídíme celé pole (v i -tém kroku nalezneme i -tý nejmenší prvek a zařadíme ho v poli na pozici s indexem i).

```

procedure SelectSort(var A: Pole);
var i, j, k, x: integer;
begin
  for i:=1 to N-1 do begin
    k:=i;
    for j:=i+1 to N do
      if A[j]<A[k] then k:=j;
    x:=A[k]; A[k]:=A[i]; A[i]:=x;
  end;
end;

```

Pro úplnost si ještě řekněme pár slov o časové složitosti právě popsaného algoritmu. V i -tém kroku musíme nalézt minimum z $N - i + 1$ čísel, na což spotřebujeme čas $\mathcal{O}(N - i + 1)$. Ve všech krocích dohromady tedy spotřebujeme čas $\mathcal{O}(N + (N - 1) + \dots + 3 + 2 + 1) = \mathcal{O}(N^2)$.

Třídění přímým vkládáním (InsertSort) funguje na podobném principu jako třídění přímým výběrem. Na začátku pole vytváříme správně utříděnou posloupnost, kterou postupně rozšiřujeme. Na začátku i -tého kroku má tato utříděná posloupnost délku $i - 1$. V i -tém kroku určíme pozici i -tého čísla v dosud utříděné posloupnosti a zařadíme ho do utříděné posloupnosti (zbytek utříděné posloupnosti se posune o jednu pozici doprava). Není těžké si rozmyslet, že každý krok lze provést v čase $\mathcal{O}(N)$. Protože počet kroků algoritmu je N , celková časová složitost právě popsaného algoritmu je opět $\mathcal{O}(N^2)$.

```

procedure InsertSort(var A: Pole);
var i, j, x: integer;
begin
  for i:=2 to N do begin
    x:=A[i];
    j:=i-1;
    while (j>0) and (x<A[j]) do begin
      A[j+1]:=A[j];
      j:=j-1;
    end;
    A[j+1]:=x;
  end;
end;

```

V uvedeném programu je využito *zkráceného vyhodnocování* v podmínce cyklu `while`. To znamená, že testování podmínky je ukončeno, jakmile $j \leq 0$, a nikdy nemůže dojít k situaci, že by program zjišťoval prvek na pozici 0 nebo menší v poli A .

Bublínkové třídění (BubbleSort) pracuje jinak než dva dříve popsané algoritmy. Algoritmu se říká „bublínkový“, protože podobně jako bublinky v limonádě „stoupají“ vysoká čísla v poli vzhůru. Postupně se porovnávají dvojice sousedních prvků, řekněme zleva doprava, a pokud v porovnávané dvojici následuje menší číslo po větším, tak se tato dvě čísla prohodí. Celý postup opakujeme, dokud probíhají nějaké vý-

měny. Protože algoritmus skončí, když nedojde k žádné výměně, je pole na konci algoritmu setříděné.

```
procedure BubbleSort(var A: Pole);
var i, x: integer;
    zmena: boolean;
begin
    repeat
        zmena:=false;
        for i:=1 to N-1 do
            if A[i] > A[i+1] then begin
                x:=A[i]; A[i]:=A[i+1]; A[i+1]:=x;
                zmena:=true;
            end;
        until not zmena;
    end;
```

Správnost algoritmu nahlédneme tak, že si uvědomíme, že po i průchodech cyklem `repeat` bude posledních i prvků obsahovat největších i prvků setříděných od nejmenšího po největší (rozmyslete si, proč tomu tak je). Popsaný algoritmus se tedy zastaví po nejvýše N průchodech a jeho celková časová složitost v nejhorsím případě je $\mathcal{O}(N^2)$, neboť na každý průchod spotřebuje čas $\mathcal{O}(N)$. Výhodou tohoto algoritmu oproti předchozím dvěma je, že je tím rychlejší, čím blíže bylo zadané pole k setříděnému stavu – pokud bylo úplně setříděné, tehdy algoritmus spotřebuje jen lineární čas $\mathcal{O}(N)$.

Rychlé metody

Sofistikovanější třídící algoritmy pracují v čase $\mathcal{O}(N \log N)$. Jedním z nich je *třídění sléváním* (*MergeSort*), založené na principu slévání (spojování) již setříděných posloupností dohromady. Představme si, že již máme dvě setříděné posloupnosti a chceme je spojit dohromady. Jednoduše stačí porovnávat nejmenší prvky obou posloupností a menší z těchto prvků vždy odstranit a přesunout do nové posloupnosti. Je zřejmé, že ke slití dvou posloupností potřebujeme čas úměrný součtu jejich délek.

My si zde popíšeme a předvedeme modifikaci algoritmu MergeSort, která používá pomocné pole. Algoritmus lze implementovat při zachování časové složitosti i bez pomocného pole, ale je to o dost pracnější. Existuje též modifikace algoritmu, která má počet fází (viz dále) v nejhorsím případě $\mathcal{O}(\log N)$, ale pokud je již pole na začátku setříděné, proběhne pouze jediná a v takovém případě má algoritmus časovou složitost $\mathcal{O}(N)$. My si však zatajíme i tuto variantu.

Algoritmus pracuje v několika *fázích*. Na začátku první fáze tvoří každý prvek jedno-prvkovou setříděnou posloupnost a obecně na začátku i -té fáze budou mít setříděné posloupnosti délky 2^{i-1} . V i -té fázi tedy vždy ze dvou sousedních 2^{i-1} -prvkových posloupností vytvoříme jedinou délky 2^i . Pokud N není násobkem 2^i , bude délka poslední posloupnosti zbytek po dělení N číslem 2^i . Zastavíme se, pokud $2^i \geq N$, tj. po $\lceil \log_2 N \rceil$ fázích.

Protože v i -té fázi slijeme $\lceil N/2^i \rceil$ dvojic nejvýše 2^{i-1} -prvkových posloupností, je časová složitost jedné fáze $\mathcal{O}(N)$. Celková časová složitost popsaného algoritmu je pak $\mathcal{O}(N \log N)$.

```

procedure MergeSort(var A: Pole);
var P: Pole;           { pomocné pole }
    delka: integer;   { délka setříděných posl. }
    i: integer;       { index do vytvářené posl. }
    i1, i2: integer; { index do slévavých posl. }
    k1, k2: integer; { konce slévavých posl. }
begin
    delka:=1;
    while delka<N do begin
        i1:=1; i2:=delka+1; i:=1;
        k1:=delka; k2:=2*delka;
        while i2<=N do begin
            { sléváme A[i1..k1] s A[i2..k2] }
            if k2>N then k2:=N;
            while (i1<=k1) or (i2<=k2) do
                if (i2>k2) or ((i1<=k1) and (A[i1]<=A[i2])) then begin
                    P[i]:=A[i1]; i:=i+1; i1:=i1+1;
                end
                else begin
                    P[i]:=A[i2]; i:=i+1; i2:=i2+1;
                end;
                i1:=k2+1; i2:=i1+delka;
                k1:=k2+delka; k2:=k2+2*delka;
            end;
            A:=P;
            delka:=2*delka;
        end;
    end;
end;

```

V čase $\mathcal{O}(N \log N)$ pracuje také algoritmus jménem *QuickSort*. Tento algoritmus je založen na metodě Rozděl a panuj. Nejprve si zvolíme nějaké číslo, kterému budeme říkat *pivot*. Více si o jeho volbě povíme později. Poté pole přeuspořádáme a rozdělíme je na dvě části tak, že žádný prvek v první části nebude větší než pivot a žádný prvek v druhé části naopak menší. Prvky v obou částech pak setřídíme rekurzivním zavoláním téhož algoritmu. Je zřejmé, že po skončení algoritmu bude pole setříděné.

Malá zrada spočívá ve volbě pivotu. Pro naše účely by se hodilo, aby levá i pravá část pole byly po přeházení přibližně stejně velké. Nejlepší volbou pivotu by tedy byl *medián* tříděného úseku, tj. prvek takový, jenž by byl v setříděném poli přesně uprostřed. Přeuspořádání jistě zvládneme v lineárním čase, a pokud by pivoty na všech úrovních byly mediány, pak by počet úrovní rekurze byl $\mathcal{O}(\log N)$. Protože je navíc na každé úrovni rekurze součet délek tříděných posloupností nejvýše N , bude celková časová složitost $\mathcal{O}(N \log N)$.

Ačkoli existuje algoritmus, který medián pole nalezne v čase $\mathcal{O}(N)$, v QuickSortu se obvykle nepoužívá, jelikož konstanta u členu N je příliš velká v porovnání s pravděpodobností, že náhodná volba pivota algoritmus příliš zpomalí. Většinou se pivot volí náhodně z dosud neseříděného úseku. Dá se ukázat, že takovýto algoritmus s velmi vysokou pravděpodobností poběží v čase $\mathcal{O}(N \log N)$.

Důkaz tohoto tvrzení je trošičku trikový a lze jej nalézt např. v knize Kapitoly z diskrétní matematiky od pánů Matouška a Nešetřila. Je však třeba si pamatovat, že pokud se pivot volí náhodně, může rekurze dosáhnout hloubky N a časová složitost algoritmu až $\mathcal{O}(N^2)$ – představme si, že se pivot v každém rekurzivním volání nešťastně zvolí jako největší prvek z tříděného úseku. V naší implementaci QuickSortu pro názornost nebudeme pivot volit náhodně, ale vždy jako pivot vybereme prostřední prvek tříděného úseku.

```

procedure QuickSort(var A: Pole; l, r: integer);
var i, j, k, x: integer;
begin
  i:=l; j:=r;
  k:=A[(i+j) div 2]; { volba pivota }
  repeat
    while A[i]<k do i:=i+1;
    while A[j]>k do j:=j-1;
    if i<=j then begin
      x:=A[i]; A[i]:=A[j]; A[j]:=x;
      i:=i+1;
      j:=j-1;
    end;
  until i >= j;
  if j>l then QuickSort(A, l, j);
  if i<r then QuickSort(A, i, r);
end;

```

Metody pro specifická data

Ještě si předvedeme dva třídící algoritmy, které jsou vhodné, pokud tříděné objekty mají některé další speciální vlastnosti. Prvním z nich je *třídění počítáním* (*CountSort*). To lze použít, pokud tříděné objekty obsahují pouze klíče a možných hodnot klíčů je málo. Tehdy si stačí spočítat, kolikrát se který klíč vyskytuje, a místo třídění vytvořit celé pole znovu na základě toho, kolik jednotlivých objektů obsahovalo pole původní. My si tento algoritmus předvedeme na příkladu třídění pole celých čísel z intervalu $\langle D, H \rangle$:

```

const D = 1;
      H = 10;
procedure CountSort(var A: Pole);
var C: array[D..H] of integer;
    i, j, k: integer;
begin

```

```

for i:=D to H do C[i]:=0;
for i:=1 to N do C[A[i]]:=C[A[i]] + 1;
k:=1;
for i:=D to H do
  for j:=1 to C[i] do begin
    A[k]:=i;
    k:=k+1;
  end;
end;
end;
```

Časová složitost takového algoritmu je lineární v N , ale nesmíme zapomenout přičíst ještě velikost intervalu, ve kterém se prvky nacházejí ($K = H - D + 1$), protože nějaký čas spotřebujeme i na inicializaci pole počítadel. Celkem tedy $\mathcal{O}(N + K)$.

Pokud by tříděné objekty obsahovaly vedle klíčů i nějaká data, můžeme je místo pouhého počítání rozdělovat do přihrádek podle hodnoty klíče a pak je z přihrádek vysbírat v rostoucím pořadí klíčů. Tomuto algoritmu se říká *přihrádkové třídění* (*BucketSort*) a my si popíšeme jeho *víceprůchodovou variantu* (*RadixSort*), která je vhodnější pro větší hodnoty K .

V první fázi si čísla rozdělíme do přihrádek (skupin) podle nejméně významné cifry a spojíme do jedné posloupnosti, v druhé fázi čísla roztřídíme podle druhé nejméně významné cifry a opět spojíme do jedné posloupnosti atd. Je důležité, aby se uvnitř každé přihrádky zachovalo pořadí čísel v posloupnosti na začátku fáze, tj. posloupnost uložená v každé přihrádce je vybranou podposloupností posloupnosti ze začátku fáze.

Tvrdíme, že na konci i -té fáze obsahuje výsledná posloupnost čísla utříděná podle i nejméně významných cifer. Zřejmě i -té nejméně významné cifry tvoří neklesající posloupnost, neboť podle nich jsme právě v této fázi rozdělovali čísla do přihrádek, a pokud dvě čísla mají tuto cifru stejnou, jsou uložena v pořadí dle jejich $i-1$ nejméně významných cifer, neboť v každé přihrádce jsme zachovali pořadí čísel z konce minulé fáze.

Na závěr poznamenejme, že místo čísel podle cifer lze do přihrádek rozdělovat též textové řetězce podle jejich znaků, atp.

Jak je to s časovou složitostí této varianty RadixSortu? Pokud třídíme celá čísla od 1 do K a v každém kroku je rozdělujeme do ℓ přihrádek, potřebujeme $\log_{\ell} K$ průchodů (tolik je cifer v zápisu čísla K v ℓ -kové soustavě). Každý průchod spotřebuje čas $\mathcal{O}(N + \ell)$, takže celý algoritmus běží v čase $\mathcal{O}((N + \ell) \log_{\ell} K)$. To je $\mathcal{O}(N)$, pokud K a ℓ jsou konstanty. My si předvedeme implementaci algoritmu pro $K = 255$ a $\ell = 2$ (čísla budeme rozhazovat do přihrádek podle bitů v jejich binárním zápisu).

```

const K=255;
procedure RadixSort(var A: Pole);
var P0, P1: Pole;
    k1, k2: integer;
    i: integer;
    bit: integer;
```


```

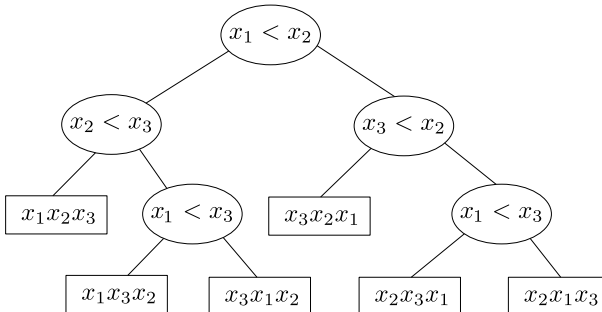
begin
  bit:=1;
  while bit<=K do begin
    k1:=0; k2:=0;
    for i:=1 to N do
      if (A[i] and bit)=0 then begin
        k1:=k1+1; P0[k1]:=A[i];
      end
      else begin
        k2:=k2+1; P1[k2]:=A[i];
      end;
    for i:=1 to k1 do A[i]:=P0[i];
    for i:=1 to k2 do A[k1+i]:=P1[i];
    bit:=bit * 2; { bitový posun o jedna vlevo }
  end;
end;

```

Dolní mez na rychlost třídění

Na závěr našeho povídání o třídících algoritmech si ukážeme, že třídít obecné údaje, se kterými neumíme provádět nic jiného než je navzájem porovnávat, rychleji než $\Theta(N \log N)$ nejen nikdo neumí, ale také ani umět nemůže. Libovolný třídící algoritmus založený na porovnávání a prohazování prvků totiž musí na některé vstupy vynaložit řádově alespoň $N \log N$ kroků. (RadixSort na první pohled tento výsledek porušuje, na druhý však už ne, když si uvědomíme, o jak speciální druh tříděných dat se jedná.)

 Třídící algoritmus v průběhu své činnosti nějak porovnává prvky a nějak je přehazuje. Provedeme myšlenkový experiment. Pozměníme algoritmus tak, že nejdříve bude pouze porovnávat, podle toho zjistí, jak jsou prvky v poli uspořádány, a když už si je jistý správným pořadím, prvky najednou přehází. Tím se algoritmus zpomalí nejvýše konstanta-krát. Také pro jednoduchost předpokládejme, že všechny tříděné údaje jsou navzájem různé. Porovnávací činnost algoritmu si pak můžeme popsat tzv. *rozhodovacím stromem*. Zde je příklad rozhodovacího stromu pro tříprvkové pole:



Každý vrchol obsahuje porovnání dvou prvků x a y , v levém podstromu daného vrcholu je činnost algoritmu pokud $x < y$, v pravém podstromu činnost při $x \geq y$. V listech je už jisté správné pořadí prvků.

Každému algoritmu odpovídá nějaký rozhodovací strom a každý průběh činnosti algoritmu odpovídá průchodu rozhodovacím stromem od kořene do nějakého listu. Naším cílem bude ukázat, že v libovolném rozhodovacím stromu (a tedy i libovolném odpovídajícím algoritmu) bude existovat cesta z kořene do nějakého listu (neboli výpočet algoritmu) délky $N \log N$.

Kolik maximálně hladin h , a tedy i jaká nejdelší cesta se v takovém stromu může vyskytnout? Náš strom má tolik listů, kolik je možných pořadí tříděných prvků, tedy právě $N!$. Různým pořadím totiž musí odpovídat různé listy, jinak by algoritmus netřídil (předpokládáme přeci, že to, jak má prvky prohazovat, může zjistit jenom jejich porovnáváním), a naopak každé pořadí prvků jednoznačně určuje cestu do příslušného listu. Na nulté hladině je jediný vrchol, na každé další hladině se oproti předchozí počet vrcholů nejvýše zdvojnásobí, takže na i -té hladině se nachází nejvýše 2^i vrcholů. Proto je listů stromu nejvýše 2^h (některé listy mohou být i výše, ale za každý takový určitě chybí jeden vrchol na h -té hladině). Z toho víme, že platí:

$$2^h \geq \text{počet listů} \geq N!,$$

a proto:

$$h \geq \log_2(N!).$$

Logaritmus faktoriálu se těžko počítá přesně, ale můžeme si ho zdola odhadnout pomocí následujícího pozorování:

$$\begin{aligned} n! &= \underbrace{n \cdot (n-1) \cdot \dots \cdot (n/2)}_{n/2 \text{ členů, každý } \geq n/2} \cdot \dots \geq \\ &\geq (n/2)^{(n/2)}. \end{aligned}$$

Dosažením získáme:

$$\begin{aligned} h &\geq \log_2(N!) \geq \log_2((N/2)^{N/2}) = \\ &= \frac{N}{2} \log_2(N/2) = \frac{1}{2} \cdot N(\log_2 N - 1) \geq \frac{1}{4} \cdot N \log_2 N. \end{aligned}$$

Vidíme tedy, že pro každý třídící algoritmus existuje vstup, na kterém se bude muset provést alespoň $c \cdot N \log N$ kroků, kde $c > 0$ je nějaká konstanta.

Poznámky

- Zkuste si též rozmyslet (drobnou modifikací předchozího důkazu), že ani *průměrná* časová složitost třídění nemůže být lepší než $N \log N$.
- Odvodit průměrnou složitost QuickSortu vlastně není zase tak těžké. Zkusme následující úvahu: Pokud by pivot nebyl přesně medián, ale alespoň se nacházel v prostřední třetině setříděného úseku, byla by složitost stále $\mathcal{O}(N \log N)$, jen by se zvýšila konstanta v \mathcal{O} -čku. Kdybychom pivota volili náhodně, ale po rozdělení prvků si zkontrolovali, jestli pivot padl do prostřední třetiny, a pokud ne (jeden z úseků by byl moc velký a druhý moc malý), volbu bychom opakovali, v průměru by nás to stálo konstantní počet pokusů (pokud čekáme na událost,

kteřá nastává náhodně s pravděpodobností p , stojí nás to v průměru $1/p$ pokusů; zde je $p = 1/3$), takže celková složitost by v průměru vzrostla jen konstantně.

Původní QuickSort sice žádné takové opakování volby neprovádí a rovnou se zavolá rekurzivně na velký i malý úsek, ale opět se po v průměru konstantně mnoha iteracích velký úsek zredukuje na nejvýše $2/3$ původní velikosti a třídění malých úseků jednotlivě nezabere víc času, než kdyby se třídily dohromady.

- Kdybychom u QuickSortu použili rekurzivní volání jen na menší interval, zatímco ten větší bychom obsloužili přenastavením proměnných a skokem na začátek právě prováděné procedury, zredukovali bychom paměťovou složitost na $\mathcal{O}(\log N)$ (nepočítaje samotné pole čísel), jelikož každé další rekurzivní volání zpracovává alespoň dvakrát menší úsek než to předchozí. Časové složitosti tím však nepomůžeme.
- Počet příhrádek u RadixSortu vůbec nemusí být konstanta – pokud např. chcete třídít N čísel v rozsahu $1 \dots N^k$, stačí si zvolit $\ell = N$ a fázi bude jenom k . Pro pevné k tak dosáhneme lineární časové složitosti.

Tomáš Valla, Martin Mareš a Dan Král

Úloha 18-2-4: Stavbyvedoucí

Stavbyvedoucím krokoběhu se stal Potrhlík a všichni ostatní zvířecí stavitelé si u něj mohli objednávat materiál. Když si konečně všichni nadiktovali, co chtěli, měl už Potrhlík pěkně dlouhý seznam. U každého předmětu ze seznamu si Potrhlík pamatuje N údajů a každý předmět má zapsaný v seznamu na jedné řádce. Celý seznam je tedy tabulka, která má N sloupců a tolik řádek, kolik je předmětů.

Všichni stavitelé si ovšem (stejně jako učitelé) myslí, že jejich předměty jsou ty nejdůležitější, a tak každý chce, aby byly všechny řádky tabulky setříděny podle jejich požadavku. Každý požadavek je číslo údaje (sloupce), podle kterého by se měly všechny řádky setřídít. (Neboli je to číslo sloupce, podle kterého bychom měli setřídít celou tabulku.)

Chudák Potrhlík nakonec obdržel M požadavků, tedy M žádostí o setřídění dle určitého sloupce. Rozhodl se, že řádky setřídí nejprve podle 1. požadavku, potom podle 2., ..., až M -tého požadavku. Navíc když budou v nějakém kroku dvě řádky podle zpracovávaného požadavku stejné, jejich vzájemné pořadí zůstane stejné jako před tímto tříděním.

Počet třídících požadavků je ale opravdu velký a často se v něm opakují čísla sloupců, takže Potrhlíka napadlo, že byste mu mohli pomoci jeho úkol zjednodušit. Zajímalo by ho, jestli by nemohl provést setřídění řádků podle menšího počtu třídících požadavků. Tato kratší posloupnost třídících požadavků by měla být s původní posloupností *ekvivalentní*, čili ať je seznam předmětů na začátku uspořádán libovolně, setřídění podle původní posloupnosti požadavků a podle kratší posloupnosti požadavků musí dát vždy stejné výsledky (stejně setříděný seznam).

Zkuste napsat program, který dostane N (počet sloupců seznamu), M (počet třídících požadavků) a jednotlivé třídící požadavky a najde nejkratší posloupnost tří-

dících požadavků, která je zadané posloupnosti ekvivalentní. Pokud je minimálních posloupností více, stačí vypsát libovolnou z nich.

Příklad: Pro $N = 3$ a $M = 7$ požadavků 3, 3, 1, 1, 2, 3, 3 je hledaná nejkratší posloupnost požadavků třeba 1, 2, 3.

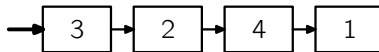
Úloha 23-3-5: Rozházené EWD

Chudák pan Richards má jen svou zapomnětlivou hlavu a pár papírů, tak budete muset vymyslet, jak setřídít EWD (číslované záznamy Edsgera Dijkstry) v konstantní paměti. To jest, že si může udělat třeba 1000 záznamů, ale ne pro každou z N EWD jeden. Vámi spotřebovaná paměť prostě na N vůbec nesmí záviset (a N může být libovolně velké – argument, že EWD je konečně mnoho, vám neprojde). Dávejte si pozor na rekurzi, spotřebovává tolik paměti, jak hluboko je zanořena.

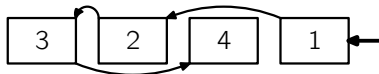
Přeházená EWD budeme reprezentovat jako spojový seznam. V programu dostanete ukazatel na první prvek spojového seznamu, kde je číslo EWD a ukazatel na další. Vaším úkolem je ho setřídít a vrátit ukazatel na první prvek (nejstarší EWD).

Spojový seznam už máte v paměti, vaším úkolem je přepojit jej do setříděného stavu.

Příklad před setříděním:



A po setřídění:



Binární vyhledávání

Představte si, že jste k narozeninám dostali obrovské pole setříděných záznamů (to je, pravda, trochu netradiční dárek, ale proč ne – může to být třeba telefonní seznam). Záznamy mohou vypadat libovolně a to, že jsou setříděné, znamená jen a pouze, že $x_1 < x_2 < \dots < x_N$, kde $<$ je nějaká relace, která nám řekne, který ze dvou záznamů je menší (pro jednoduchost předpokládáme, že žádné dva záznamy nejsou stejné).

Co si ale s takovým polem počneme? Zkusíme si v něm najít nějaký konkrétní záznam z . To můžeme udělat třeba tak, že si nalistujeme prostřední záznam (označíme si ho 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 zůstane jedna polovina a v ní budeme pokračovat stejným způsobem. Tak budeme postupně zmenšovat interval, ve kterém se z může nacházet, až buďto z najdeme nebo vyloučíme všechny prvky, kde by mohlo být.

Tomuto principu se obvykle říká *binární vyhledávání* nebo také *hledání půlením intervalu* a snadno ho naprogramujeme buďto rekurzivně nebo pomocí cyklu, v němž si budeme udržovat interval $\langle l, r \rangle$, ve kterém se hledaný prvek může nacházet:

```
function BinSearch(z : integer): integer;
var l, r, m : integer;
begin
  l := 1;    { interval, ve kterém hledáme }
  r := N;
  while l <= r do begin { ještě není prázdný }
    m := (l+r) div 2;   { střed intervalu }
    if z < x[m] then
      r := m-1         { je vlevo }
    else if z > x[m] then
      l := m+1         { je vpravo }
    else begin         { Bingo! }
      hledej := m; exit;
    end;
  end;
  hledej := -1;       { nebyl nikde }
end;
```

Všimněte si, že průchodů cyklem `while` může být nejvýše $\lceil \log_2 N \rceil$, protože interval $\langle l, r \rangle$ na počátku obsahuje N prvků a v každém průchodu jej zmenšíme na polovinu (ve skutečnosti ještě o jedničku, ale tím lépe pro nás). Proto po k průchodech bude interval obsahovat nejvýše $N/2^k$ prvků a jelikož pro $N/2^k < 1$ se algoritmus zastaví, může být k nejvýše $\log_2 N$. Proto je časová složitost binárního vyhledávání $\mathcal{O}(\log N)$.

Poznámky

- Algoritmus je nejjednodušším příkladem návrhového postupu zvaného *Rozděl a panuj*, kterému je v dalším textu věnována celá kapitola.
- Pokud záznamy můžeme jenom porovnávat, je binární vyhledávání nejlepší možné. Libovolné hledání založené na porovnávání lze totiž popsat binárním stromem a binární strom s N vrcholy musí mít vždy hloubku alespoň $\lfloor \log_2 N \rfloor$. Pokud můžeme použít *hešování* (viz další kapitoly), dostaneme se na průměrně konstantní složitost (ale v nejhorším případě lineární). Jeho nevýhodou ovšem je, že udržuje jenom množinu prvků, nikoliv uspořádání na ní, takže například nelze najít k zadanému prvku nejbližší vyšší.
- Když hledáme v telefonním seznamu, nepůlíme intervaly, ale hádáme, kam přibližně by ze zkoumaného intervalu hledaná hodnota mohla přijít – pokud hledáme Zemana, otevřeme seznam někde u konce. Na tom je založené *interpoláčnické hledání*, které v průměrném případě rovnoměrně rozložených dat najde výsledek v $\mathcal{O}(\log \log N)$.

Od výše uvedeného kódu se liší právě jenom určováním „středu“:

```
m := 1 + (z-x[l]) * (r-l) div (x[r]-x[l]); { střed intervalu }
```

Na nepravidelných datech takové hledání ale může trvat až lineárně dlouho. Můžeme to ošetřit tím, že budeme kroky binárního a interpoláčnického hledání střídát. Na špatných datech tak bude doba běhu omezená dvojnásobkem doby běhu hledání binárního a na dobrých dvojnásobkem časové složitosti interpoláčnického hledání.

Stojí nám to však za tu námahu? Asi jen v případě, kdy nás stojí čtení prvků pole nemalý čas, tedy pokud je uloženo na pevném disku, popř. aspoň vypadlo-li pro svou velikost z procesorových keší.

- Co když potřebujeme seznam rychle upravovat? Do pole novou hodnotu rychle vložit nejde. Normálně bychom použili spojový seznam, ale tím bychom zhoršili složitost binárního vyhledávání k nepoužitelnosti, protože je v něm nalezení prostřední hodnoty v $\mathcal{O}(N)$. Řešením jsou *vyhledávací stromy*, o kterých mluvíme v jedné z následujících kuchařek.

Martin Mareš, Tomáš Valla a Lukáš Lánský

Úloha 22-5-2: Strážce údolí

Údolí draků hlídají havrani rozmístění na přímce. Jenže někteří jsou moc blízko u sebe a mají tendenci se místo hlídání vybavovat, takže jsme se rozhodli počet stráží zredukovat. Nevíme však, které propustit.

Známe polohu všech N havranů na přímce, zadanou celočíselnými mezerami mezi nimi, a chceme jich vyřadit K , aby byli dva nejbližší havrani od sebe co nejdále. Potřebujeme tedy maximalizovat minimální vzdálenost mezi nimi. Dokážete pro nás rychle najít K havranů, jež pošleme do výslužby?

Například pro $N = 6$, $K = 3$ a mezery mezi havrany 4, 6, 2, 5, 7 je správným řešením propustit 2., 3. a 5. havrana (bráno zleva), takže zůstanou mezery 12, 12. Pro

$N = 14$, $K = 7$, mezery 5, 12, 6, 3, 8, 1, 4, 1, 1, 9, 15, 1, 16 vyhodíme 2., 4., 6., 7., 8., 9. a 12. (možností je tentokrát více).

Úloha 23-1-3: Jedna maticová

Na vstupu dostaneme matici, tj. dvojrozměrné pole celých čísel, která má navíc tu zvláštní vlastnost, že jsou čísla v každém jejím řádku a sloupci ostře rostoucí (liší se alespoň o 1). Potřebovali bychom rychle zjistit, zdali v ní neexistuje nějaké políčko v i -tém řádku a j -tém sloupci, které by mělo hodnotu přesně $i + j$.

Pokud hledaných políček existuje víc, můžete vypsat libovolné z nich. Pokuste se také vymyslet, jak rychle spočítat, kolik takových políček je.

Při zvažování časové složitosti nepočítejte dobu načítání: představujte si, že už máte matici v paměti. Zkuste zdůvodnit, proč nelze dosáhnout rychlejšího řešení.

Příklad vstupu:

```
-3  1  4
  4  5  6
  7  9 11
```

Odpovídající výstup: 1. řádek, 3. sloupec

Příklad vstupu:

```
 3  4  5
  4  5  6
  5  6  7
```

Odpovídající výstup: žádné takové políčko není

Halda

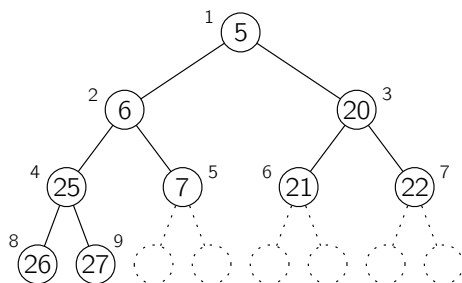
Halda je datová struktura pro uchovávání množiny čísel (či jakýchkoliv jiných prvků, na kterých máme definováno uspořádání, tj. umíme pro každou dvojici prvků říci, který z nich je menší). Tato datová struktura obvykle podporuje následující operace: přidání nového prvku, nalezení nejmenšího prvku a odebrání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

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

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

i	1	2	3	4	5	6	7	8	9
$h[i]$	5	6	20	25	7	21	22	26	27

Tomu odpovídá tento strom:



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

Jestliže halda obsahuje N prvků, pak nový prvek, řekněme mu třeba x , přidáme na konec pole, tj. na pozici s indexem $N + 1$. Nyní x porovnáme s jeho předchůdcem. Pokud je jeho předchůdce menší, je vše v pořádku a jsme hotovi. V opačném případě x s jeho předchůdcem prohodíme.

Tím jsme problém napravili, ale nyní může být x menší než jeho nový předchůdce. To lze napravit dalším prohozením a tak budeme pokračovat dále, než se buďto dostaneme do situace, kdy už je x větší nebo rovno svému předchůdci, nebo „vybublá“

až do kořene haldy, kde už žádného předchůdce nemá. Protože se v každém kroku pozice, na níž se prvek x právě nachází, zmenší alespoň na polovinu, provedeme dohromady nejvýše $\mathcal{O}(\log N)$ výměn, a tedy spotřebujeme čas $\mathcal{O}(\log N)$.

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

Poznámky

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

Zdrojový kód

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

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

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

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

```

    if i=j then break;
    x:=halda[i]; halda[i]:=halda[j]; halda[j]:=x;
    i:=j
end
end;
```

HeapSort

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

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

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

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

```

type Pole = array[1..MAXN] of Integer;
procedure HeapSort(var A: Pole);
var i, x: integer;
    procedure bubblej(m, i: integer); { "zabublání" prvku }
    { m je velikost haldy, i je index zabublávaného prvku }
    var j, x: integer;
    begin
        while 2*i<=m do begin
            j:=2*i;
            if (j<m) and (A[j+1]>A[j]) then j:=j+1;
            if A[i]>=A[j] then break;
            x:=A[i]; A[i]:=A[j]; A[j]:=x;
            i:=j;
        end;
    end;
end;
```

```

begin
  for i:=N div 2 downto 1 do bubblej(N,i); { bubblej }
  for i:=N downto 2 do begin { vybírej maximum }
    x:=A[1]; A[1]:=A[i]; A[i]:=x;
    bubblej(i-1, 1);
  end;
end;
end;

```

Dan Král, Martin Mareš a Petr Škoda

Úloha 19-2-3: Moneymaker

Na vstupu dostane váš program číslo N , což je počet úkolů ke zpracování. Zpracování každé úlohy zabere jednotkový čas. Dále pak N řádků, každý se dvěma čísly. První číslo znamená, do kdy je třeba úkol vykonat, a druhé číslo je odměna, kterou za splněný úkol dostaneme. V jednom čase mohou pracovat právě na jednom úkolu. Výstupem programu by pak mělo být takové pořadí úkolů, aby zisk byl maximální. Pokud je takových pořadí více, stačí libovolné z nich.

Příklad: Pro $N = 4$ a záznamy

```

    3  1
    1  3
    2  5
    2  4

```

je optimální pořadí 3, 4 a 1.

Úloha 20-4-4: Skupinky pro chytré

Budeme pracovat se záznamy lidí. Každý člověk má jméno a IQ (přičemž IQ je celé kladné číslo). Z lidí budeme vytvářet skupinky. Každá skupinka má unikátní ID (identifikační číslo), které je opět celé a kladné. Na počátku máme pouze jedinou prázdnou skupinku s $ID=1$.

Nad skupinkami chceme provozovat operace:

- **INSERT** – Vloží nového člověka.
- **FIND_BEST** – Nalezne člověka s nejvyšším IQ.
- **DELETE_BEST** – Člověk s nejvyšším IQ odchází za kariérou do zahraničí (odstraníme jej).

Výše uvedené operace dostanou vždy ID skupinky, nad kterou mají být provedeny. Žádná z operací nemodifikuje skupinku, ale místo toho vytvoří skupinku novou, ve které budou uloženy výsledky operace. ID nové skupinky bude nejmenší dosud nepoužité číslo. Pochopitelně operace **FIND_BEST** pouze vrací nalezeného člověka, takže nevytvoří novou skupinku. Skupinky nikdy nezanikají, takže je potřeba si je nějakým způsobem udržovat všechny.

Výsledek každé operace musíte oznámit ještě před tím, než začnete zpracovávat operaci další – tj. nesmíte si operace bufferovat a pak jich provést víc najednou.

Vaším úkolem je navrhnout a popsat vhodnou datovou reprezentaci a jak na ní budou probíhat požadované operace.

Příklad: Jak bylo řečeno, na začátku máme jen jednu prázdnou třídu s $ID=1$. Budeme provádět operace:

- `INSERT("Aleš", IQ=130)` do $ID=1$ vytvoří skupinku $ID=2$.
- `INSERT("Petr", IQ=110)` do $ID=2$ vytvoří skupinku $ID=3$.
- `INSERT("Jana", IQ=140)` do $ID=1$ vytvoří skupinku $ID=4$.
- `FIND_BEST` v $ID=2$ vrátí "Aleš".
- `FIND_BEST` v $ID=3$ vrátí také "Aleš".
- `FIND_BEST` v $ID=4$ vrátí ovšem "Jana".
- `DELETE_BEST` z $ID=3$ vytvoří skupinku $ID=5$.
- `FIND_BEST` v $ID=5$ vrátí "Petr", protože Aleše odstranila předchozí operace.

Grafy

Co mají společného následující úlohy?

- Na filmovém festivalu se sešlo šest tisíc lidí, některé dvojice se znají, některé ne. Jak najít největší skupinu lidí, ve které se všichni znají?
- Podnikatel sepisuje procesy, které se pravidelně opakují při tvorbě jeho produktu. Některé úkony závisí na dokončení jiných, a tak by rád věděl, jak je uspořádat a jaké možnosti má při jejich paralelizování.
- Jak najít nejkratší cestu z Prahy do Brna, máme-li zadánu silniční síť České republiky jako trojice (město, město, délka)?

Všechny problémy mají společné, že jejich vstup můžeme redukovat na dvě množiny: objekty a vztahy mezi dvojicemi těchto objektů. Objekty jsou po řadě lidé, úkony a města; vztahy jsou

- A se zná s B .
- A závisí na dokončení B .
- Mezi A a B existuje cesta dlouhá x .

Takovým zadáním říká moderní matematika *grafy*. Ten pojem nijak nesouvisí s grafy jakožto obrázky vývoje funkcí nebo vizualizacemi statistických dat. Graf je v jádru skutečně jen dvojice množin, které budeme v počítači často reprezentovat seznamy.

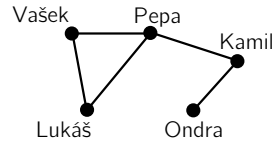
Na rozdíl od jiných částí nové matematiky jsou grafy názorné – krásně se kreslí. Ačkoliv se část teorie zabývá vlastnostmi takového kreslení a jedna z dalších kuchařek se věnuje grafům, které se dají dobře kreslit do roviny, nesmíme se nechat touto názorností zmást. Jako informatici si je kreslíme jen proto, že se nám o seznamech čísel špatně přemýšlí. To, jak jsme si je nakreslili, na seznamech čísel pranic nezmění.

Definování

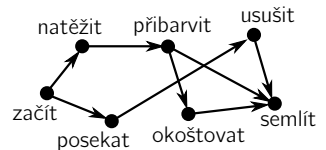
Naše tři příklady zachycují tři obvyklé situace vztahů mezi objekty:

- Nejjednodušší případ nastává na filmovém festivalu. Jediné, co si o vztahu pamatujeme, je, zda-li existuje. Mezi dvěma lidmi navíc nemůže existovat víc vztahů. Grafům, které modelují takové situace, říkáme *obyčejné* a kreslíme je tak, že označené vrcholy spojujeme bezejmennými čarami.
- Situace podnikatele je složitější, protože vztahy mají směr. Ačkoliv v uvedené úloze existence dvou protisměrných vztahů mezi dvojicí vrcholů znamená neexistenci řešení, v obecnosti nám takové situace nevadí. Grafy tohoto typu označujeme jako *orientované* a místo čar kreslíme šipky.
- Případ hledání nejkratší cesty pracuje na nejobecnějším grafu (v jistém smyslu), grafu *ohodnoceném*. Ten si u každého vztahu pamatuje, zdali existuje a pokud

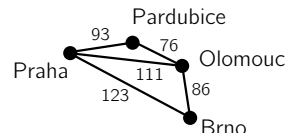
Obyčejný graf



Orientovaný graf



Ohodnocený graf



ano, jaké má ohodnocení, což může být libovolný údaj, zpravidla číslo. Při kreslení si toto ohodnocení připisujeme k čarám.

Nemá cenu dále zastírat obvyklou terminologii: objektům se říká *vrcholy* a vztahům *hrany*. Počátky teorie grafů se totiž vážou ke zkoumání pravidelných mnohostěnů.

Matematika definuje obyčejný graf jako uspořádanou dvojici (V, E) , kde V je obecná (*nosná*) množina vrcholů (**vertices**) a E množina neuspořádaných dvojic $\{u, v\}$ hran (**edges**). V případě grafu orientovaného jsou dvojice z E uspořádané, (u, v) . Zanést ohodnocení do definice můžeme prostě tím, že k (V, E) přidáme funkci $h : E \rightarrow M$, kde M je množina, ze které ohodnocení vybíráme. Ale jde to samozřejmě i jinak. Definice slouží jen jako kontrola, že nám intuice sloužila všem stejně.

Cvičení a poznámky

- Řešení všech třech úvodních úloh zde neuvádíme explicitně, ale v knize přítomná jsou. Zkuste je objevit.
- Můžeme ohodnotit graf orientovaný? Můžeme ohodnocovat vrcholy, ne hrany? Můžeme vést mezi dvěma vrcholy několik hran? Můžeme všechno! Existují dokonce i taková zobecnění, která jako hranu chápou množinu ne dvou, ale k vrcholů.

Programová reprezentace

Programátor si definuje obyčejný graf především podle možností svého programovacího jazyka. Zatímco součástí pythonistovy jazykové kultury jsou seznamy, díky kterým se nemusí starat o dynamické alokace paměti a může tak graf psát prostě jako seznam vrcholů, kde je vrchol seznamem sousedních vrcholů, pascalista takové řešení zpravidla nezvolí. Nechce totiž plýtvat pamětí statickou alokací, u níž musí předpokládat, že vrchol může mít tolik sousedů, kolik je v grafu vrcholů, ale ani se nechce párat se spojovými seznamy.

Pro mluvíci starších jazyků existuje standardní trik, jak neplýtvat pamětí a nemuset dynamicky alokovat. Uděláme si dvě pole, jedno bude velikosti N (tímto písmenkem se obvykle značí počet vrcholů), druhé velikosti M (počet hran). Do hranového pole budeme ukládat *cíle* hran, ve vrcholovém bude ke každému vrcholu uvedeno, kde v poli hran začínají hrany *z něj vycházející* – konec tohoto úseku je implicitně určen počátkem následujícího vrcholu. Pokud zrovna ukládáme graf neorientovaný, budeme každou hranu chápat jako dvojici hran orientovaných, protisměrných, což ostatně platí u každého způsobu uložení grafu *seznamem sousedů*, jak se probírané metodě říká.

V případě složitých grafových algoritmů, které za svého běhu graf extenzivně modifikují, se kvůli časové složitosti nevyhneme nutnosti použít spojový seznam pro seznamy hran vedoucích z jednotlivých vrcholů. Pokud si ke každé hraně zapamatujeme nejen, kam vede, ale i kde se nachází její *druhý konec* ve spojovém seznamu protějšího vrcholu, můžeme v konstantním čase hranu odebrat, což se nám v jedno-dušších reprezentacích bude dařit jen těžko.

V pokročilejších partiích teorie grafů se používají rozličné maticové reprezentace a máte-li rádi algebru, je doporučeníhodné se s nimi seznámit.

V následujících receptech budeme, vzhledem k použitému programovacímu jazyku, vždy používat seznamy sousedů uložené výše popsaným, trikovým způsobem. Hranovému poli budeme říkat *Sousedí*, poli vrcholovému *Zacatky* a nadeklarujeme si je takto:

```
var N, M: integer; { počet vrcholů a hran }
    Zacatky: array[1..MaxN+1] of integer;
    Sousedí: array[1..MaxM] of integer;
```

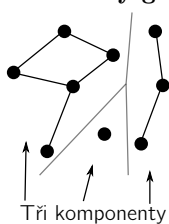
Souvislost

Grafová terminologie je bohatá a vtipná. Třeba *cesta* (délky $k-1$) je taková posloupnost různých vrcholů u_1, u_2, \dots, u_k , kde jsou všechny těsně po sobě jdoucí vrcholy u_i, u_{i+1} spojeny hranou. Tedy vskutku *cesta* v netechnickém významu toho slova, pokud graf chápeme jako mapu, po které se můžeme procházet.

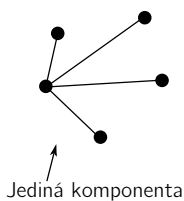
Nabízí se přemýšlet nad tím, kam až v takovém grafu/mapě můžeme z některého vrcholu po všech možných cestách dojít. Z Prahy se pěšky (suchou nohou) do Sydney nedostaneme, do Bratislavy ano.

Množině vrcholů grafu, pro které platí, že se z libovolného jejího vrcholu dostaneme po nějaké cestě do libovolného jiného vrcholu množiny, a která je maximální v tom smyslu, že už do ni nemůžeme přidat žádný jiný vrchol grafu, říkáme *komponenta souvislosti* grafu. Vrcholy každého grafu můžeme na komponenty beze zbytku rozdělit. Pokud má graf komponentu pouze jednu (z odkudkoliv se dostaneme kamkoliv), budeme jej označovat za graf *souvislý*.

Nesouvislý graf



Souvislý graf



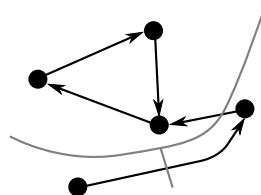
Uvědomte si, že fakt, že lze z každého vrcholu dojít do každého jiného *po cestě*, neznamená, že mezi každými dvěma vrcholy existuje hrana (cesty délky jedna). Kdyby mezi každými dvěma vrcholy vedla hrana, prohlásili bychom graf za *úplný* a značili ho K_n .

Jak najít komponenty souvislosti programově? Těžko vymyslet snazší úkol! Označíme si vrcholy příznakem *nenavštíveno* a pro libovolný nenavštívený vrchol spustíme rekurzivní funkci prohledávání, která nedělá nic jiného, než že odejme příznak nenavštívenosti vrcholu v argumentu a zavolá sama sebe na všechny sousedy, kteří příznak ještě mají.

Takový postup v čase lineárním vůči počtu hran komponenty objeví komponentu, ve které se nacházel vybraný vrchol, a můžeme ho použít znovu a znovu, dokud do komponent nerozbijeme celý graf. Systematicky se tímto prohledáním budeme zabývat v části *Prohledávání do hloubky*.

Souvislost orientovaného grafu je o dost složitější vlastnost. Existuje ve dvou variantách: definici *slabé souvislosti* získáme tak, že ze zkoumaného grafu „odmažeme šipky“ a na výsledném neorientovaném grafu identifikujeme komponenty souvislosti. *Silná souvislost* se definuje takřka stejně jako souvislost na obyčejném grafu, avšak využívá místo obyčejné cesty definici cesty orientované – je to to samé, akorát pořadí vrcholů musí respektovat orientaci šipek.

Silně souvislý graf je tedy i slabě souvislý, protože lze-li se z vrcholu do vrcholu dostat respektující směry šipek, jde to jistě, i když na ně zapomeneme. Naopak to ale neplatí, jak ukazuje zobrazený slabě souvislý graf s vyznačenými komponentami silné souvislosti.

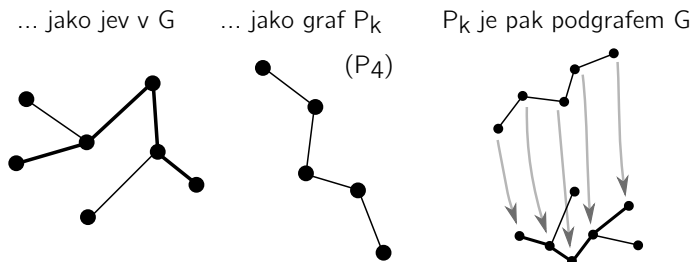


Slabě souvislý graf se třemi komponentami silné souvislosti.

Poznámky

- Vedle *cesty* existují ještě dva podobné pojmy související s procházkami – jde o *tah* a *sled*. Zatímco v cestě se nemohou opakovat vrcholy, a tedy ani hrany (to si rozmyslete), v tahu se nesmí opakovat pouze hrany a sled je obecná procházka po grafu, která se může sestávat z poskakování mezi dvěma vrcholy.
- Vypadá to, jako by pojem „cesta“ měl dva významy: je to jednak jev, který nastává v obecném grafu (tak jsme si jej definovali), druhak je cesta délky k graf sám o sobě označovaný jako P_k (z anglického *path*).

Cesta



Souvislost obou použití je v tom, že můžeme P_k prohlásit za *podgraf* libovolného grafu, v němž nastává v úvodu kapitoly popisovaná situace (existuje v něm cesta délky k). Graf G_1 je přitom podgrafem grafu G_2 tehdy, existuje-li v G_2 množina vrcholů V taková, že můžeme sestavit vzájemně jednoznačné přiřazení mezi vrcholy V a všemi vrcholy G_1 takové, že kdykoliv mezi dvěma vrcholy v G_1 vede hrana, existuje hrana i mezi příslušnými vrcholy v G_2 .

Totožná situace nastává u kružnic, o kterých mluví další část. Obecně ale tímto způsobem zaměřovat výskyt v grafu a graf sám nezní dostatečně odborně – vyskytne-li se kupříkladu úplný graf K_n v grafu H , začne se mu říkat *klika*.

Kružnice a stromy

Kružnice je, podobně jako cesta, posloupnost různých vrcholů u_1, u_2, \dots, u_k , kde platí, že mezi u_i a u_{i+1} vede hrana. Navíc ale musí vést hrana i mezi u_k a u_1 .

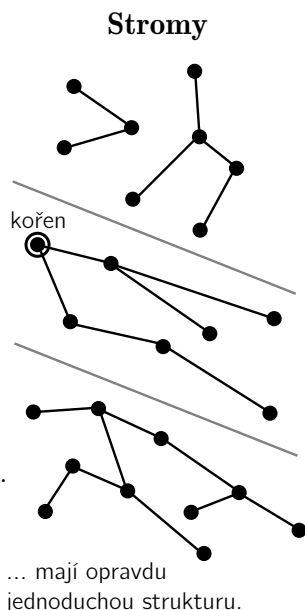
Jsou-li dva vrcholy v obyčejném grafu na kružnici, znamená to, že mezi nimi existují alespoň dvě cesty – jedna po prvním oblouku kružnice, druhá po druhém. Považujeme-li hrany za spojení (dopravní, komunikační), je to dobrá zpráva, protože výpadek jedné z hran kružnice nezpůsobí nedostupnost (graf zůstane souvislý – viz kapitola o hranové 2-souvislosti).

Neexistence kružnice má pro strukturu grafu silné důsledky. Platí, že právě pokud v souvislém grafu není kružnice, pak

- mezi každými dvěma vrcholy vede právě jedna cesta,
- odebrání libovolné hrany způsobí ztrátu souvislosti,
- přidání libovolné hrany vytvoří kružnici,
- vrcholů je v takovém grafu právě o jeden víc než hran.

Těmto souvislým grafům bez kružnic se říká *stromy* a je zábavným cvičením si dokázat ekvivalenci mezi čtyřmi uvedenými body.

Stromy jsou velmi oblíbené datové struktury – jen mezi kuchařkami najdete dvě ne náhodou pojmenované „vyhledávací stromy“ a „intervaldové stromy“. Protože je užitečné mít ve stromu zvláštní vrchol, ze kterého budeme strom prohledávat a skrze který na něj budeme v programu odkazovat, dostalo se mu zvláštního pojmenování – kořen.



Prohledávání do hloubky

Naše povídání o grafových algoritmech začneme dvěma základními způsoby procházení grafem. K tomu budeme potřebovat dvě podobné jednoduché datové struktury: *Fronta* je konečná posloupnost prvků, která má označený začátek a konec. Když do ní přidáváme nový prvek, přidáme ho na konec posloupnosti. Když z ní prvek odebíráme, odebereme ten na začátku. Proto se tato struktura anglicky nazývá *first in, first out*, zkráceně *FIFO*.

Zásobník je také konečná posloupnost prvků se začátkem a koncem, ale zatímco prvky přidáváme také na konec, odebíráme je z téhož konce. Anglický název je (překvapivě) *last in, first out*, čili *LIFO*.

Algoritmus *prohledávání grafu do hloubky*:

1. Na začátku máme v zásobníku pouze vstupní vrchol w . Dále si u každého vrcholu v pamatujeme značku z_v , která říká, zda jsme vrchol již navštívili. Vstupní vrchol je označený, ostatní vrcholy nikoliv.
2. Odebereme vrchol ze zásobníku, nazvěme ho u .
3. Každý neoznačený vrchol, do kterého vede hrana z u , přidáme do zásobníku a označíme.
4. Kroky 2 a 3 opakujeme, dokud není zásobník prázdný.

Na konci algoritmu budou označeny všechny vrcholy dosažitelné z vrcholu w , tedy v případě neorientovaného grafu celá komponenta souvislosti obsahující w .

To můžeme snadno dokázat sporem: Předpokládáme, že existuje vrchol x , který není označen, ale do kterého vede cesta z w . Pokud je takových vrcholů více, vezmeme si ten nejbližší k w . Označme si y předchůdce vrcholu x na nejkratší cestě z w ; y je určitě označený (jinak by x nebyl nejbližší neoznačený). Vrchol y se tedy musel někdy objevit na zásobníku, tím pádem jsme ho také museli ze zásobníku odebrat a v kroku 3 označit všechny jeho sousedy, tedy i vrchol x , což je ovšem spor.

To, že algoritmus někdy skončí, nahlédneme snadno: v kroku 3 na zásobník přidáváme pouze vrcholy, které dosud nejsou označeny, a hned je značíme. Proto se každý vrchol může na zásobníku objevit nejvýše jednou, a jelikož ve 2. kroku pokaždé odebereme jeden vrchol ze zásobníku, musí vrcholy někdy (konkrétně po nejvýše N opakováních cyklu) dojít. Ve 3. kroku probereme každou hranu grafu nejvýše dvakrát (v každém směru jednou).

Časová složitost celého algoritmu je tedy lineární v počtu vrcholů N a počtu hran M , čili $\mathcal{O}(N + M)$. Paměťová složitost je stejná, protože si tak jako tak musíme hrany a vrcholy pamatovat a zásobník není větší než paměť na vrcholy.

Prohledávání do hloubky implementujeme nejsnáze rekurzivní funkcí. Jako zásobník v tom případě používáme přímo zásobník programu, kde si program ukládá návratové adresy funkcí. Může to vypadat třeba následovně:

```
var Oznaceni: array[1..MaxN] of boolean;

procedure Projdi(V: integer);
var I: integer;
begin
  Oznaceni[V] := True;
  for I := Zacatky[V] to Zacatky[V+1]-1 do
    if not Oznaceni[Sousedci[I]] then
      Projdi(Sousedci[I]);
end;
```

Rozdělit neorientovaný graf na komponenty souvislosti je pak už jednoduché. Projdeme postupně všechny vrcholy grafu, a pokud nejsou v žádné z dosud označených komponent grafu, přidáme novou komponentu tak, že graf z tohoto vrcholu prohledáme do hloubky. Vrcholy značíme přímo číslem komponenty, do které patří. Protože prohledáváme do hloubky několik oddělených částí grafu, každou se složitostí $\mathcal{O}(N_i + M_i)$, kde N_i a M_i je počet vrcholů a hran komponenty, vyjde dohromady složitost $\mathcal{O}(N + M)$. Nic nového si ukládat nemusíme, a proto je paměťová složitost stále $\mathcal{O}(N + M)$.

```

var Komponenta: array[1..MaxN] of integer;
    NovaKomponenta: integer;
procedure Projdi(V: integer);
var I: integer;
begin
    Komponenta[V] := NovaKomponenta;
    for I := Zacatky[V] to Zacatky[V+1]-1 do
        if Komponenta[Sousedi[I]] = -1 then
            Projdi(Sousedi[I]);
end;
var I: integer;
begin
    ...
    for I := 1 to N do Komponenta[I] := -1;
    NovaKomponenta := 1;
    for I := 1 to N do
        if Komponenta[I] = -1 then begin
            Projdi(I);
            NovaKomponenta := NovaKomponenta + 1;
        end;
    ...
end.

```

Průběh prohledávání grafu do hloubky můžeme znázornit stromem (podle anglického názvu pro prohledávání do hloubky „Depth-First Search“ se mu říká *DFS strom*). Z počátečního vrcholu w učiníme kořen. Pak budeme graf procházet do hloubky a vrcholy zakreslovat jako syny vrcholů, ze kterých jsme přišli. Syny každého vrcholu si uspořádáme v pořadí, v němž jsme je navštívili; tomuto pořadí budeme říkat *zleva doprava* a také ho tak budeme kreslit.

Hranám mezi otci a syny budeme říkat *stromové hrany*. Protože jsme do žádného vrcholu nešli dvakrát, budou opravdu tvořit strom. Hrany, které vedou do již navštívených vrcholů na cestě, kterou jsme přišli z kořene, nazveme *zpětné hrany*. *Dopředné hrany* vedou naopak z vrcholu blíže kořeni do už označeného vrcholu dále od kořene. A konečně *příčné hrany* vedou mezi dvěma různými podstromy grafu.

Všimněte si, že při prohledávání neorientovaného grafu objevíme každou hranu dvakrát: buďto poprvé jako stromovou a podruhé jako zpětnou, anebo jednou jako zpětnou a podruhé jako dopřednou. Příčné hrany se objevit nemohou – pokud by příčná hrana vedla doprava, vedla by do dosud neoznačeného vrcholu, takže by se prohledávání vydalo touto hranou a nevznikl by oddělený podstrom; doleva rovněž vést nemůže: představme si stav prohledávání v okamžiku, kdy jsme opouštěli levý vrchol této hrany. Tehdy by naše hrana musela být příčnou vedoucí doprava, ale o té už víme, že neexistuje.

Pro orientované grafy je situace opět trochu složitější: stromové a dopředné hrany jsou orientované vždy ve stromě shora dolů, zpětné zdola nahoru a příčné hrany mohou existovat, ovšem vždy vedou zprava doleva, čili pouze do podstromů, které jsme již prošli (nahlédneme opět stejně).

Prohledávání do šířky

Prohledávání do šířky je založené na podobné myšlence jako prohledávání do hloubky, pouze místo zásobníku používá frontu:

1. Na začátku máme ve frontě pouze jeden prvek, a to zadaný vrchol w . Dále si u každého vrcholu x pamatujeme číslo $H[x]$. Všechny vrcholy budou mít na začátku $H[x] = -1$, jen $H[w] = 0$.
2. Odebereme vrchol z fronty, označme ho u .
3. Každý vrchol v , do kterého vede hrana z u a jeho $H[v] = -1$, přidáme do fronty a nastavíme jeho $H[v]$ na $H[u] + 1$.
4. Kroky 2 a 3 opakujeme, dokud není fronta prázdná.

Podobně jako u prohledávání do hloubky jsme se dostali právě do těch vrcholů, do kterých vede cesta z w (a označili jsme je nezápornými čísly). Rovněž je každému vrcholu přiřazeno nezáporné číslo maximálně jednou. To vše se dokazuje podobně, jako jsme dokázali správnost prohledávání do hloubky.

Vrcholy se stejným číslem tvoří ve frontě jeden souvislý celek, protože nejprve odebereme z fronty všechny vrcholy s číslem n , než začneme odebírat vrcholy s číslem $n + 1$. Navíc platí, že $H[v]$ udává délku nejkratší cesty z vrcholu w do v .

Neexistenci kratší cesty dokážeme sporem: Pokud existuje nějaký vrchol v , pro který $H[v]$ neodpovídá délce nejkratší cesty z w do v , čili vzdálenosti $D[v]$, vybereme si z takových v to, jehož $D[v]$ je nejmenší. Pak nalezneme nejkratší cestu z w do v a její předposlední vrchol z . Vrchol z je bližší než v , takže pro něj už musí být $D[z] = H[z]$. Ovšem když jsme z fronty vrchol z odebírali, museli jsme objevit i jeho souseda v , který ještě nemohl být označený, tudíž jsme mu museli přidělit $H[v] = H[z] + 1 = D[v]$ a to je spor.

Prohledávání do šířky má časovou složitost taktéž lineární s počtem hran a vrcholů. Na každou hranu se také ptáme dvakrát. Fronta má lineární velikost k počtu vrcholů, takže jsme si oproti prohledávání do hloubky nepohoršili a i paměťová složitost je $\mathcal{O}(N + M)$. Algoritmus implementujeme nejsnáze cyklem, který bude pracovat s vrcholy v poli představujícím frontu.

```
var Fronta, H: array[1..MaxN] of integer;  
    I, V, Prvni, Posledni: integer;  
    PocatecniVrchol: integer;  
begin  
    ...  
    for I := 1 to N do H[I] := -1;  
    Prvni := 1;  
    Posledni := 1;  
    Fronta[Prvni] := PocatecniVrchol;
```

```

H[PocatecniVrchol] := 0;
repeat
  V := Fronta[Prvni];
  for I := Zacatky[V] to Zacatky[V+1]-1 do
    if H[Sousedi[I]] < 0 then begin
      H[Sousedi[I]] := H[V]+1;
      Posledni := Posledni + 1;
      Fronta[Posledni] := Sousedi[I];
    end;
  Prvni := Prvni + 1;
until Prvni > Posledni; { Fronta je prázdná }
...
end.

```

Prohledávání do šířky lze také použít na hledání komponent souvislosti a hledání kostry grafu (viz speciální kuchařka).

Topologické uspořádání

Teď si vysvětlíme, co je *topologické uspořádání* grafu. Máme orientovaný graf G s N vrcholy a chceme očíslovat vrcholy čísly 1 až N tak, aby všechny hrany vedly z vrcholu s menším číslem do vrcholu s větším číslem, tedy aby pro každou hranu $e = (v_i, v_j)$ bylo $i < j$. Představme si to jako srovnání vrcholů grafu na přímku tak, aby „šipky“ vedly pouze zleva doprava.

Nejprve si ukážeme, že pro žádný orientovaný graf, který obsahuje cyklus, nelze takovéto topologické pořadí vytvořit. Označme vrcholy cyklu v_1, \dots, v_k tak, že hrana vede z vrcholu v_{i-1} do vrcholu v_i , resp. z v_k do v_1 . Pak vrchol v_2 musí dostat vyšší číslo než vrchol v_1 , v_3 než v_2, \dots, v_k než v_{k-1} . Ale vrchol v_1 musí mít zároveň vyšší číslo než v_k , což nelze splnit.

Cyklus je ovšem to jediné, co může existenci topologického uspořádání zabránit. Libovolný acyklický graf lze uspořádat následujícím algoritmem:

1. Na začátku máme orientovaný graf G a proměnnou $p = N$.
2. Najdeme takový vrchol v , ze kterého nevede žádná hrana (budeme mu říkat *stok*).
Pokud v grafu žádný stok není, výpočet končí, protože jsme našli cyklus.
3. Odebereme z grafu vrchol v a všechny hrany, které do něj vedou.
4. Přiřadíme vrcholu v číslo p .
5. Proměnnou p snížíme o 1.
6. Opakujeme kroky 2 až 5, dokud graf obsahuje alespoň jeden vrchol.

Proč tento algoritmus funguje? Pokud v grafu nalezneme stok, můžeme mu určitě přiřadit číslo větší než všem ostatním vrcholům, protože překážet by nám v tom mohly pouze hrany vedoucí ze stoku ven a ty neexistují. Jakmile stok očíslováme, můžeme jej z grafu odstranit a pokračovat číslováním ostatních vrcholů. Tento postup musí někdy skončit, jelikož v grafu je pouze konečně mnoho vrcholů.

Zbývá si uvědomit, že v neprázdném grafu, který neobsahuje cyklus, vždy existuje alespoň jeden stok: Vezmeme libovolný vrchol v_1 . Pokud z něj vede nějaká hrana, pokračujeme po ní do nějakého vrcholu v_2 , z něj do v_3 atd. Co se při tom může stát?

- Dostaneme se do vrcholu v_i , ze kterého nevede žádná hrana. Vyhráli jsme, máme stok.
- Narazíme na v_i , ve kterém jsme už jednou byli. To by ale znamenalo, že graf obsahuje cyklus, což, jak víme, není pravda.
- Budeme objevovat stále a nové a nové vrcholy. V konečném grafu nemožno.

Algoritmus můžeme navíc snadno upravit tak, aby netratil příliš času hledáním vrcholů, z nichž nic nevede – stačí si takové vrcholy pamatovat ve frontě, a kdykoliv nějaký takový vrchol odstraňujeme, zkontrolujeme si, zda jsme nějakému jinému vrcholu nezrušili poslední hranu, která z něj vedla, a pokud ano, přidat takový vrchol na konec fronty. Celé topologické třídění pak zvládneme v čase $\mathcal{O}(N + M)$.

Jiná možnost je prohledat graf do hloubky a všimnout si, že pořadí, ve kterém jsme se z vrcholů vraceli, je právě topologické pořadí. Pokud zrovna opouštíme nějaký vrchol a čísujeme ho dalším číslem v pořadí, rozmysleme si, jaké druhy hran z něj mohou vést: stromová nebo dopředná hrana vede do vrcholu, kterému jsme již přiřadili vyšší číslo, zpětná existovat nemůže (v grafu by byl cyklus) a příčné hrany vedou pouze zprava doleva, takže také do již očíslovaných vrcholů. Časová složitost je opět $\mathcal{O}(N + M)$.

```
var Ocislovani: array[1..MaxN] of integer;
    Posledni: integer;
    I: integer;

procedure Projdi(V: integer);
var I: integer;
begin
    Ocislovani[V] := 0; { zatím V jen označíme }
    for I := Zacatky[V] to Zacatky[V+1]-1 do
        if Ocislovani[Sousedi[I]] = -1 then
            Projdi(Sousedi[I]);
    Ocislovani[V] := Posledni;
    Posledni := Posledni - 1;
end;

begin
    ...
    for I := 1 to N do
        Ocislovani[I] := -1;
    Posledni := N;
    for I := 1 to N do
        if Ocislovani[I] = -1 then Projdi(I);
    ...
end.
```

Hranová a vrcholová 2-souvislost

Nyní se podíváme na trochu komplikovanější formu souvislosti. Říkáme, že neorientovaný graf je *hranově 2-souvislý*, když platí, že:

- má alespoň 3 vrcholy,
- je souvislý,
- zůstane souvislý po odebrání libovolné hrany.

Hranu, jejíž odebrání by způsobilo zvýšení počtu komponent souvislosti grafu, nazýváme *most*.

Na hledání mostů nám poslouží opět upravené prohledávání do hloubky a DFS strom. Všimněme si, že mostem může být jediné stromová hrana – každá jiná hrana totiž leží na nějaké kružnici. Odebráním mostu se graf rozpadne na část obsahující kořen DFS stromu a podstrom „visící“ pod touto hranou. Jediné, co tomu může zabránit, je existence nějaké další hrany mezi podstromem a hlavní částí, což musí být zpětná hrana, navíc taková, která není jenom stromovou hranou viděnou z druhé strany. Takovým hranám budeme říkat *ryzí zpětné hrany*.

Proto si pro každý vrchol spočítáme hladinu, ve které se nachází (kořen je na hladině 0, jeho synové na hladině 1, jejich synové 2, ...). Dále si pro každý vrchol v spočítáme, do jaké nejvyšší hladiny (s nejmenším číslem) vedou ryzí zpětné hrany z podstromu s kořenem v . To můžeme udělat přímo při procházení do hloubky, protože než se vrátíme z v , projdeme celý podstrom pod v .

Pokud všechny zpětné hrany vedou do hladiny stejné nebo větší než té, na které je v , pak odebráním hrany vedoucí do v z jeho otce vzniknou dvě komponenty souvislosti, čili tato hrana je mostem. V opačném případě jsme našli kružnici, na níž tato hrana leží, takže to most být nemůže. Výjimku tvoří kořen, který žádného otce nemá a nemusíme se o něj proto starat.

Algoritmus je tedy pouhou modifikací procházení do hloubky a má i stejnou časovou a paměťovou složitost $\mathcal{O}(N + M)$. Zde jsou důležité části programu:

```
var Hladina, Spojeno: array[1..MaxN] of integer;
    DvojSouvisle: Boolean;
    I: integer;

procedure Projdi(V, NovaHladina: integer);
var I, W: integer;
begin
    Hladina[V] := NovaHladina;
    Spojeno[V] := Hladina[V];

    for I := Zacatky[V] to Zacatky[V+1]-1 do begin
        W := Sousedi[I];
        if Hladina[W] = -1 then
            begin { stromová hrana }
                Projdi(W, NovaHladina + 1);
                if Spojeno[W] < Spojeno[V] then
```

```

        Spojeno[V] := Spojeno[W];
    if Spojeno[W] > Hladina[V] then
        DvojSouvisle := False; { máme most }
    end else { zpětná nebo dopředná hrana }
    if (Hladina[W] < NovaHladina-1) and
        (Hladina[W] < Spojeno[V]) then
        Spojeno[V] := Hladina[W];
    end;
end;
begin
    ...
    for I := 1 to N do
        Hladina[I] := -1;
        DvojSouvisle := True;
        Projdi(1, 0);
    ...
end.

```

Další formou souvislosti je *vrcholová souvislost*. Graf je *vrcholově 2-souvislý*, právě když:

- má alespoň 3 vrcholy,
- je souvislý,
- zůstane souvislý po odebrání libovolného vrcholu.

Artikulace je takový vrchol, který když odebereme, zvýší se počet komponent souvislosti grafu.

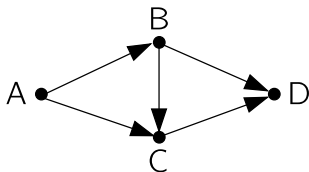
Algoritmus pro zjištění vrcholové 2-souvislosti grafu je velmi podobný algoritmu na zjišťování hranové 2-souvislosti, jen si musíme uvědomit, že odebíráme celý vrchol. Ze stromu procházení do hloubky může odebráním vrcholu vzniknout až několik podstromů, které všechny musí být spojeny zpětnou hranou s hlavním stromem. Proto musí zpětné hrany z podstromu určeného vrcholem v vést až *nad* vrchol v . Speciálně pro kořen nám vychází, že může mít pouze jednoho syna, jinak bychom ho mohli odebrat a vytvořit tak dvě nebo více komponent souvislosti. Algoritmus se od hledání hranové 2-souvislosti liší jedinou změnou ostré nerovnosti na neostrou, sami zkuste najít, které nerovnosti.

Martin Mareš, David Matoušek, Petr Škoda a Lukáš Lánský

Úloha 20-3-4: Orientace na mapě

Na vstupu váš program dostane popis orientovaného grafu znázorňujícího mapu. Víte, že tento graf neobsahuje žádný orientovaný cyklus, čili že neexistuje žádná orientovaná cesta délky alespoň 1, která by začínala a končila ve stejném vrcholu. Úkolem programu je vypsát dvojici vrcholů, mezi kterými vede nejvíce různých cest v celém grafu. Za různé jsou považovány libovolné dvě cesty, které se liší alespoň jednou hranou. Pokud je takových dvojic vrcholů více, stačí libovolná z nich.

Příklad: Pro tento graf je řešením dvojice vrcholů A a D :



Úloha 18-2-5: Krokoběh

Krokoběh se skládá z několika jezírek, ve kterých mohou krokodýli odpočívat, a kanálů mezi nimi. Kanály jsou obousměrné, vedou vždy mezi dvěma jezírky a žádné dva kanály se mimo jezírka neprotínají (mimoúrovňově ale mohou).

Nějaká jezírka a kanály jsou již postaveny. Pokud se ovšem stane, že se krokodýl nemůže dostat do nějakého jezírka (nevede k němu žádná cesta), je velmi nerudný a žere vše kolem. (*Kvák!*) Protože Potrhlík nechce dopadnout stejně jako Skrblikvák, chtěl by dostavět potřebné kanály tak, aby byli krokodýli spokojení. Ti budou spokojení, pokud i když jeden libovolný kanál vyschne, pořád se budou moci dostat z každého jezírka do kteréhokoliv jiného. A protože stavba krokoběhu Potrhlíka finančně velmi vyčerpala, chtěl by postavit nových kanálů co nejméně.

Váš program dostane na vstupu popis existujícího krokoběhu. Ten se skládá z $N > 2$ jezírek a M kanálů, každý kanál spojuje dvojici jezírek. Vaším cílem je zjistit, kolik nejméně kanálů je třeba přidat, aby i když libovolný jeden kanál vyschne, bylo pořád možné dostat se z každého jezírka do každého.

Příklad: Pro $N = 6$ jezírek a $M = 4$ kanály vedoucí mezi jezírky $(1, 2)$, $(2, 3)$, $(3, 1)$ a $(4, 5)$ je třeba postavit alespoň další 3 kanály. (Jsou to například $(1, 4)$, $(5, 6)$, $(6, 2)$.)

Dijkstrův algoritmus

Tento algoritmus dostane orientovaný graf s hranami ohodnocenými nezápornými čísly a nalezne v něm nejkratší cestu mezi dvěma zadanými vrcholy. Ve skutečnosti dělá o malinko více: najde totiž nejkratší cesty z jednoho zadaného vrcholu do všech ostatních.

Nechť v_0 je vrchol grafu, ze kterého chceme určit délky nejkratších cest. Budeme si udržovat pole délek zatím nalezených cest z vrcholu v_0 do všech ostatních vrcholů grafu. Navíc u některých vrcholů budeme mít poznamenáno, že cesta nalezená do nich je už ta nejkratší možná. Takovým vrcholům budeme říkat *definitivní*.

Na začátku inicializujeme v poli všechny hodnoty na ∞ kromě hodnoty odpovídající vrcholu v_0 , kterou inicializujeme na 0 (délka nejkratší cesty z v_0 do v_0 je 0). V každém kroku algoritmu pak provedeme následující: Vybereme vrchol w , který ještě není definitivní, a mezi všemi takovými vrcholy je délka zatím nalezené cesty do něj nejkratší možná (v první kroku tedy v_0). Vrchol w prohlásíme za definitivní. Dále otestujeme, zda pro nějaký sousední vrchol v vrcholu w cesta z vrcholu v_0 do w a pak po hraně z w do v není kratší, než zatím nalezená cesta z v_0 do v , a je-li tomu tak, upravíme délku zatím nalezené cesty do v . Toto provedeme pro všechny takové vrcholy v .


Celý algoritmus skončí, pokud jsou už všechny vrcholy definitivní nebo všechny vrcholy, co nejsou definitivní, mají délku cesty rovnou ∞ (v takovém případě se graf skládá z více nesouvislých částí).

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

Pro každý z N vrcholů si délku dosud nalezené cesty uchováme v poli. Celý algoritmus má nejvýše N kroků, protože v každém kroku nám přibude jeden definitivní vrchol. Ten vybíráme jako minimum z délky aktuální cesty přes všechny dosud ne-definitivní vrcholy, kterých je $\mathcal{O}(N)$.

V každému kroku musíme zkontrolovat tolik vrcholů v , kolik hran vede z vrcholu w . Počet takových změn pro všechny kroky dohromady je pak nejvýše $\mathcal{O}(M)$, kde M je počet hran vstupního grafu. Z toho vyjde časová složitost $\mathcal{O}(N^2 + M)$, čili $\mathcal{O}(N^2)$, jelikož M je nejvýše N^2 . Tuto implementaci Dijkstrova algoritmu najdete na konci naší kuchařky.

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

 Vraťme se nyní k důkazu správnosti Dijkstrova algoritmu. Ukážeme, že po každém kroku algoritmu platí následující tvrzení: nechtě A je množina definitivních vrcholů, pak délka dosud nalezené cesty z v_0 do v (v je libovolný vrchol grafu) je

délka nejkratší cesty $v_0v_1 \dots v_kv$ takové, že všechny vrcholy v_0, v_1, \dots, v_k jsou v množině A .

Tvrzení dokážeme indukcí dle počtu kroků algoritmu, které již proběhly. Zřejmě to platí před a po prvním kroku algoritmu (A je prázdná, potom obsahuje jen v_0), takže je třeba ukázat platnost tvrzení pro k -tý krok za předpokladu, že platí pro krok $k - 1$ a všechny předchozí.

Nechť w je vrchol, který byl v předchozím kroku prohlášen za definitivní. Uvažme nejprve nějaký vrchol v , který je definitivní. Pokud $v = w$, tvrzení je triviální. V opačném případě ukážeme, že existuje nejkratší cesta z v_0 do v přes vrcholy z A , která nepoužívá vrchol w .

Označme D délku cesty z v_0 do v přes vrcholy A bez vrcholu w . Protože v každém kroku vybíráme vrchol s nejmenším ohodnocením a ohodnocení vybraných vrcholů v jednotlivých krocích tvoří neklesající posloupnost (váhy hran jsou nezáporné!), tak délka cesty z v_0 do w přes vrcholy z A je alespoň D . Ale potom délka libovolné cesty z v_0 do v přes w používající vrcholy z A je alespoň D . Z volby D pak víme, že existuje nejkratší cesta z v_0 do v přes vrcholy z A , která nepoužívá vrchol w .

Nyní uvažme takový vrchol v , který není definitivní. Nechť $v_0v_1 \dots v_kv$ je nejkratší cesta z v_0 do v taková, že všechny vrcholy v_0, v_1, \dots, v_k jsou v množině A .

Pokud $v_k = w$, pak jsme ohodnocení v změnili na délku této cesty v právě proběhlém kroku. Pokud $v_k \neq w$, pak v_0v_1, \dots, v_k je nejkratší cesta z v_0 do v_k přes vrcholy z množiny A a tedy můžeme předpokládat, že žádný z vrcholů v_1, \dots, v_k není w (podle toho, co jsme si rozmysleli na konci minulého odstavce). Potom se ale délka cesty do v rovnala správné hodnotě už před právě proběhlým krokem.

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

Poznámky

- Dijkstrův algoritmus je možné snadno upravit tak, aby nám kromě délky nejkratší cesty i takovou cestu našel: u každého vrcholu si v okamžiku, kdy mu měníme ohodnocení, poznamenáme, ze kterého vrcholu do něj přicházíme. Nejkratší cestu do nějakého vrcholu pak zrekonstruujeme tak, že u posledního vrcholu této cesty zjistíme, který vrchol je předposlední, u předposledního, který je předpředposlední, atd.
- Existují i jiné než binární haldy, například k -regulární haldy, v nichž má každý prvek k následníků (rozmyslete si, jaká je v takové haldě časová složitost operací a jak nastavit k v závislosti na M a N , aby byl Dijkstrův algoritmus co nejrychlejší), nebo tzv. Fibonaccioho halda, která dokáže upravit hodnotu prvku v konstantním čase. S tou pak umíme hledat nejkratší cesty v čase $\mathcal{O}(M + N \log N)$.

Dan Král, Martin Mareš a Petr Škoda

Implementace Dijkstrova algoritmu

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

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

Úloha 18-3-4: Pochoutka pro prasátko

V lese sousedícím s poklidným rybníčkem našich hrošíků se objevilo hladem šilhající prasátko. Zaslýchlo totiž zvěsti o Velké Bukvici, která si tou dobou lebedila v podzemí lesíku. Začalo tedy bez rozmyslu rejdit mezi stromy, leč brzy mu došly síly – byl to už přeci jenom nějaký čas od poslední mňamky. Budete umět prasátku poradit?

Les si představme jako čtvercovou síť, v jednom políčku prasátko, v jiném bukvice. Aby to nebylo tak jednoduché, prasátko se v lese může pohybovat jen podle určitých pravidel a každé z nich stojí nějaké kladné množství námahy.

Konkrétně – na vstupu dostanete rozměry lesa a pozici prasátka a bukvice spolu s pravidly, podle kterých se prasátko může pohybovat. Každé pravidlo obsahuje trojici čísel $x y z$, kde x a y je povolený posun v mřížce (o kolik se změní pozice prasátka ve čtvercové síti), zatímco z je úsilí, které prasátko musí vynaložit pro daný přesun. Vaším úkolem je najít a vypsát cestu od prasátka k bukvici. Na své cestě nesmí prasátko opustit lesík. A aby milý čuník hlady nepošel, musí být vynaložené úsilí nejmenší možné.

Příklad: Les má rozměry 6×6 , poloha prasátka je $[3, 3]$ a poloha bukvice $[1, 5]$. Pohyb prasátka se řídí třemi pravidly $2 \ 2 \ 3, 1 \ 1 \ 1$ a $-4 \ 0 \ 5$. Potom je pro prasátko nejvýhodnější použít dvakrát pravidlo 2 ($\rightarrow [5, 5]$) a pak jednou pravidlo 3 ($\rightarrow [1, 5]$). Vynaložená námaha je pak $2 \cdot 1 + 5 = 7$.

Úloha 22-1-1: Alčina interpretace

Máme velký dům se spoustou pokojů, mezi některými z nich vedou schodiště: z jednoho pokoje do druhého se buď stoupá, nebo klesá. Jen tak z legrace hledáme cestu z jednoho pokoje do druhého tak, abychom co nejméně krát musili přestat vycházet schody a začít scházet, nebo naopak přestat scházet a začít vycházet.

Minimální kostra

Představme si následující problém: Chceme určit silnice, které se budou v zimě udržovat sjízdné, a to tak, abychom celkově udržovali co nejméně kilometrů silnic, a přesto žádné město od ostatních neodřízli.

Města a silnice si můžeme představit jako nám už dobře známý graf, o kterém nyní budeme předpokládat, že je souvislý. Kdyby nebyl, náš problém nijak vyřešit nelze. Výsledný podgraf/seznam silnic, který řeší náš problém se sněhem, nazývají matematici *minimální kostra grafu*.

Co se v souvislém grafu přesně myslí pod pojmem *kostra*? Nazveme jí libovolný podgraf, který obsahuje všechny vrcholy a zároveň je stromem. *Strom* jsme si definovali v kapitole o grafech; jsou to přesně ty grafy, které jsou souvislé (z každého vrcholu „dojedeme“ do každého jiného) a bez kružnice (takže nemáme v silniční síti žádné přebytečné cesty).

Pokud každou hranu grafu ohodnotíme nějakou *vahou*, což v našem případě bude vždy kladné číslo, dostaneme *ohodnocený graf*. V takových grafech pak obvykle hledáme mezi všemi kostrami *kostru minimální*, což je taková, pro kterou je součet vah jejích hran nejmenší možný. Graf může mít více minimálních koster – například jestliže jsou všechny váhy hran jedničky, všechny kostry mají stejnou váhu $n - 1$ (kde n je počet vrcholů grafu), a tedy jsou všechny minimální.

Pro vyřešení problému hledání minimální kostry se nám bude hodit datová struktura *Disjoint-Find-Union* (DFU). Ta umí pro dané disjunktní množiny (disjunktní znamená, že každé 2 množiny mají prázdný průnik neboli žádné společné prvky) rychle rozhodnout, jestli dva prvky patří do stejné množiny, a provádět operaci sjednocení dvou množin.

Algoritmus

Algoritmus na hledání minimální kostry, který si předvedeme, je typickou ukázkou tzv. hladového algoritmu. Nejprve setřídíme hrany vzestupně podle jejich váhy. Kostru budeme postupně vytvářet přidáváním hran od té s nejmenší vahou tak, že hranu do kostry přidáme právě tehdy, pokud spojuje dvě (prozatím) různé komponenty souvislosti vytvořeného podgrafu. Jinak řečeno, hranu do vytvářené kostry přidáme, pokud v ní zatím neexistuje cesta mezi vrcholy, které zkoumaná hrana spojuje.

Je zřejmé, že tímto postupem získáme kostru, tj. acyklický podgraf grafu, který je souvislý (pokud vstupní graf je souvislý, což mlčky předpokládáme). Než si ukážeme, že nalezená kostra je opravdu minimální, podívejme se na časovou složitost našeho algoritmu: Pokud vstupní graf má N vrcholů a M hran, tak úvodní setřídění hran vyžaduje čas $\mathcal{O}(M \log M)$ (použijeme některý z rychlých třídících algoritmů popsanych v jednom z minulých dílů kuchařky) a poté se pokusíme přidat každou z M hran.

V druhé části kuchařky si ukážeme datovou strukturu, s jejíž pomocí bude M testů toho, zda mezi dvěma vrcholy vede hrana, trvat nejvýše $\mathcal{O}(M \log N)$. Celková časová složitost našeho algoritmu je tedy $\mathcal{O}(M \log N)$ (všimněte si, že $\log M \leq \log N^2 = 2 \log N$). Paměťová složitost je lineární vzhledem k počtu hran, tj. $\mathcal{O}(M)$.

Důkaz správnosti

Zbývá dokázat, že nalezená kostra vstupního grafu je minimální. Bez újmy na obecnosti můžeme předpokládat, že váhy všech hran grafu jsou navzájem různé: Pokud tomu tak není již na začátku, přičteme k některým z hran, jejichž váhy jsou duplicitní, velmi malá kladná celá čísla tak, aby pořadí hran nalezené naším třídícím algoritmem zůstalo zachováno. Tím se kostra nalezená hladovým algoritmem nezmění a pokud bude tato kostra minimální s modifikovanými váhami, bude minimální i pro původní zadání.

Označme si nyní T_{alg} kostru nalezenou hladovým algoritmem a T_{min} nějakou minimální kostru. Co by se stalo, kdyby byly různé? Víme, že všechny kostry mají stejný počet hran, takže musí existovat alespoň jedna hrana e , která je v T_{alg} , ale není v T_{min} . Ze všech takových hran si vyberme tu, která má nejmenší váhu, tedy kterou algoritmus přidal jako první. Když se podíváme na stav algoritmu těsně před přidáním e , vidíme, že sestrojil nějakou částečnou kostru F , která je ještě součástí jak T_{min} , tak T_{alg} .

Přidejme nyní hranu e ke kostře T_{min} . Tím vznikl podgraf vstupního grafu, který zjevně obsahuje nějakou kružnici C – už před přidáním hrany e totiž T_{min} byla souvislá. Protože kostra T_{alg} neobsahuje žádnou kružnici, na kružnici C musí být alespoň jedna hrana e' , která není v T_{alg} .

Všimněme si, že hranu e' nemohl algoritmus zpracovat před hranou e : hrana e' neleží v T_{min} na žádném cyklu, takže tím spíše netvoří cyklus v F a kdyby ji algoritmus zpracoval, musel by ji přidat do F , což, jak víme, neučinil. Z toho plyne, že váha hrany e' je větší než váha hrany e . Když nyní z kostry T_{min} odebereme hranu e' a přidáme místo ní hranu e , musíme opět dostat souvislý podgraf (e a e' přeci ležely na společné kružnici), tudíž kostru vstupního grafu. Jenže tato kostra má celkově menší váhu než minimální kostra T_{min} , což není možné. Tím jsme došli ke sporu, a proto T_{min} a T_{alg} nemohou být různé.

Cvičení

- V důkazu jsme předpokládali, že váhy hran jsou různé (resp. jsme je různými udělali). Není potřeba i v samotném algoritmu přičítat velmi malá čísla k hranám se stejnou vahou?

Disjoint-Find-Union

Datová struktura *DFU* slouží k udržování rozkladu množiny na několik disjunktních podmnožin (čili takových, že žádné dvě nemají společný prvek). To znamená, že pomocí této struktury můžeme pro každé dva z uložených prvků říci, zda patří či nepatří do stejné podmnožiny rozkladu.

V algoritmu hledání minimální kostry budou prvky v *DFU* vrcholy zadaného grafu a budou náležet do stejné podmnožiny rozkladu, pokud mezi nimi v již vytvořené části kostry existuje cesta. Jinými slovy podmnožiny v *DFU* budou odpovídat komponentám souvislosti vytvářené kostry.

S reprezentovaným rozkladem umožňuje datová struktura *DFU* provádět následující dvě operace:

- **find:** Test, zda dva prvky leží ve stejné podmnožině rozkladu. Tato operace bude v případě našeho algoritmu odpovídat testu, zda dva vrcholy leží ve stejné komponentě souvislosti.
- **union:** Sloučení dvou podmnožin do jedné. Tuto operaci v našem algoritmu na hledání kostry provedeme vždy, když do vytvářené kostry přidáme hranu (tehdy spojíme dvě různé komponenty souvislosti dohromady).

Povězme si nejprve, jak budeme jednotlivé podmnožiny reprezentovat. Prvky obsažené v jedné podmnožině budou tvořit zakořeněný strom. V tomto stromě však povedou ukazatele (trochu nezvykle) od listů ke kořeni. Operaci *find* lze pak jednoduše implementovat tak, že pro oba zadané prvky nejprve nalezneme kořeny jejich stromů. Jsou-li tyto kořeny stejné, jsou prvky ve stejném stromě, a tedy i ve stejné podmnožině rozkladu. Naopak, jsou-li různé, jsou zadané prvky v různých stromech, a tedy jsou i v různých podmnožinách reprezentovaného rozkladu. Operaci *union* provedeme tak, že mezi kořeny stromů reprezentujících slučované podmnožiny přidáme ukazatel a tím tyto dva stromy spojíme dohromady.

Implementace dvou výše popsaných operací, jak jsme se ji právě popsali, následuje. Pro jednoduchost množina, jejíž rozklad reprezentujeme, bude množina čísel od 1 do N . Rodiče jednotlivých vrcholů stromu si pak pamatujeme v poli *parent*, kde 0 znamená, že prvek rodiče nemá, tj. že je kořenem svého stromu. Funkce *root(v)* vrátí kořen stromu, který obsahuje prvek v .

```
var parent: array[1..N] of integer;

procedure init;
var i: integer;
begin
  for i:=1 to N do parent[i]:=0;
end;

function root(v: integer):integer;
begin
  if parent[v]=0 then root:=v
  else root:=root(parent[v]);
end;

function find(v, w: integer):boolean;
begin
  find:=(root(v)=root(w));
end;

procedure union(v, w: integer);
begin
  v:=root(v); w:=root(w);
  if v<>w then parent[v]:=w;
end;
```

S právě předvedenou implementací operací *find* a *union* by se ale mohlo stát, že stromy odpovídající podmnožinám budou vypadat jako „hadi“ a pokud budou obsahovat N prvků, na nalezení kořene bude potřeba čas $\mathcal{O}(N)$.

Ke zrychlení práce DFU se používají dvě jednoduchá vylepšení:

- **union by rank:** Každý prvek má přiřazen *rank*. Na začátku jsou ranky všech prvků rovny nule. Při provádění operace *union* připojíme strom s kořenem menšího ranku ke kořeni stromu s větším rankem. Ranky kořenů stromů se v tomto případě nemění. Pokud kořeny obou stromů mají stejný rank, připojíme je libovolně, ale rank kořenu výsledného stromu zvětšíme o jedna.
- **path compression:** Ve funkci *root(v)* přepojíme všechny prvky na cestě od prvku v ke kořeni rovnou na kořen, tj. změníme jejich rodiče na kořen daného stromu.

Než si obě metody blíže rozebereme, podívejme se, jak se změní implementace funkcí *root* a *union*:

```

var parent: array[1..N] of integer;
    rank: array[1..N] of integer;

procedure init;
var i: integer;
begin
    for i:=1 to N do
        begin
            parent[i]:=0;
            rank[i]:=0;
        end;
end;

{změna path compression}
function root(v: integer): integer;
begin
    if parent[v]=0 then root:=v
    else begin
        parent[v]:=root(parent[v]);
        root:=parent[v];
    end;
end;

{stejná jako minule}
function find(v, w: integer):boolean;
begin
    find:=(root(v)=root(w));
end;

{změna kvůli union by rank}
procedure union(v, w: integer);
begin
    v:=root(v);

```

```

w:=root(w);
if v=w then exit;
if rank[v]=rank[w] then
  begin
    parent[v]:=w;
    rank[w]:=rank[w]+1;
  end
else if rank[v]<rank[w] then
  parent[v]:=w
else
  parent[w]:=v;
end;

```

Zaměříme se nyní blíže na metodu *union by rank*. Nejprve učiníme následující pozorování: Pokud je prvek v s rankem r kořenem stromu v datové struktuře DFU, pak tento strom obsahuje alespoň 2^r prvků. Naše pozorování dokážeme indukci podle r . Pro $r = 0$ tvrzení zřejmě platí. Nechť tedy $r > 0$. V okamžiku, kdy se rank prvku v mění z $r - 1$ na r , slučujeme dva stromy, jejichž kořeny mají rank $r - 1$. Každý z těchto dvou stromů má dle indukčního předpokladu alespoň 2^{r-1} prvků, a tedy výsledný strom má alespoň 2^r prvků, jak jsme požadovali. Z našeho pozorování ihned plyne, že rank každého prvku je nejvýše $\log_2 N$ a prvků s rankem r je nejvýše $N/2^r$ (všimněme si, že rank prvku v DFU se nemění po okamžiku, kdy daný prvek přestane být kořen nějakého stromu).

Když tedy provádíme jen *union by rank*, je hloubka každého stromu v DFU rovna ranku jeho kořene, protože rank kořene se mění právě tehdy, když zvětšujeme hloubku stromu o jedna. A protože rank každého prvku je nanejvýš $\log_2 N$, hloubka každého stromu v DFU je také nanejvýš $\log_2 N$. Potom ale procedura *root* spotřebuje čas nejvýše $\mathcal{O}(\log N)$, a tedy operace *find* a *union* stihneme v čase $\mathcal{O}(\log N)$.

Amortizovaná časová složitost

Abychom mohli pokračovat dále, musíme si vysvětlit, co je *amortizovaná* časová složitost. Řekneme, že nějaká operace pracuje v amortizovaném čase $\mathcal{O}(t)$, pakliže provedení libovolných k takových operací trvá nejvýše $\mathcal{O}(kt)$. Přitom provedení kterékoliv konkrétní z nich může vyžadovat čas větší. Tento větší čas je pak v součtu kompenzován kratším časem, který spotřebovaly některé předchozí operace.

Nejdříve si předvedme tento pojem na jednoduchém příkladě. Řekneme, že máme číslo zapsané ve dvojkové soustavě. Přičíst k tomuto číslu jedničku jistě netrvá konstantní čas, neboť záleží na tom, kolik jedniček se vyskytuje na konci zadaného čísla. Pokud se nám ale povede ukázat, že N přičtení jedničky k číslu, které je na počátku nula, zabere čas $\mathcal{O}(N)$, pak můžeme říci, že každé takové přičtení trvalo amortizovaně $\mathcal{O}(1)$.


Jak tedy ukážeme, že N přičtení jedničky k číslu zabere čas $\mathcal{O}(N)$? Použijeme k tomu „penízkovou metodu“. Každá operace nás bude stát jeden penízek, a pokud jich na N operací použijeme jen $\mathcal{O}(N)$, bude tvrzení dokázáno.

Každé jedničky, kterou chceme přičíst, dáme dva penízky. V průběhu celého přičítání bude platit, že každá jednička ve dvojkovém zápisu čísla má jeden penízek (když začneme jedničky přičítat k nule, tuto podmínku splníme). Přičítání bude probíhat tak, že přičítaná jednička se „podívá“ na nejnižší bit (tj. ve dvojkovém zápise na poslední cifru) zadaného čísla (to jí stojí jeden penízek). Pokud je to nula, změní ji na jedničku a dá jí svůj zbylý penízek. Pokud to je jednička, vezme si přičítaná jednička její penízek (čili už má zase dva), změní zkoumanou jedničku na nulu a pokračuje u dalšího bitu, atd.

Takto splníme podmínku, že každá jednička v dvojkovém zápisu čísla má jeden penízek. Tedy N přičítání nás stojí $2N$ penízků. Protože počet penízků utracených během jedné operace je úměrný spotřebovanému času, vidíme, že všech N přičtení proběhne v čase $\mathcal{O}(N)$. Není těžké si uvědomit, že přičtení některých jedniček může trvat až $\mathcal{O}(\log N)$, ale amortizovaná časová složitost přičtení jedné jedničky je konstantní.

Dokončení analýzy DFU

Pokud bychom prováděli pouze *path compression* a nikoliv *union by rank*, dalo by se dokázat, že každá z operací *find* a *union* vyžaduje amortizovaně čas $\mathcal{O}(\log N)$, kde N je počet prvků. Toto tvrzení nebudeme dokazovat, protože tím bychom si nijak oproti samotnému *union by rank* nepomohli. Proč tedy vlastně hovoříme o obou vylepšeních? Inu proto, že při použití obou metod současně dosáhneme mnohem lepšího amortizovaného času $\mathcal{O}(\alpha(N))$ na jednu operaci *find* nebo *union*, kde $\alpha(N)$ je inverzní Ackermannova funkce. Její definici můžete nalézt na konci kuchařky, zde jen poznamenejme, že hodnota inverzní Ackermannovy funkce $\alpha(N)$ je pro všechny praktické hodnoty N nejvýše čtyři. Čili dosáhneme v podstatě amortizovaně konstantní časovou složitost na jednu (libovolnou) operaci DFU.

 Dokázat výše zmíněný odhad časové složitosti funkcí $\alpha(N)$ je docela těžké, my si zde předvedeme poněkud horší, ale technicky výrazně jednodušší časový odhad $\mathcal{O}((N + L) \log^* N)$, kde L je počet provedených operací *find* nebo *union* a $\log^* N$ je tzv. iterovaný logaritmus, jehož definice následuje. Nejprve si definujeme funkci $2 \uparrow k$ rekurzivním předpisem:

$$2 \uparrow 0 = 1, \quad 2 \uparrow k = 2^{2 \uparrow (k-1)}.$$

Máme tedy $2 \uparrow 1 = 2$, $2 \uparrow 2 = 2^2 = 4$, $2 \uparrow 3 = 2^4 = 16$, $2 \uparrow 4 = 2^{16} = 65536$, $2 \uparrow 5 = 2^{65536}$, atd. A konečně, iterovaný logaritmus $\log^* N$ čísla N je nejmenší přirozené číslo k takové, že $N \leq 2 \uparrow k$. Jiná (ale ekvivalentní) definice iterovaného logaritmu je ta, že $\log^* N$ je nejmenší počet, kolikrát musíme číslo N opakovaně zlogaritmovat, než dostaneme hodnotu menší nebo rovnu jedné.

Zbývá provést slíbenou analýzu struktury DFU při současném použití obou metod *union by rank* a *path compression*. Prvky si rozdělíme do skupin podle jejich ranku: k -tá skupina prvků bude tvořena těmi prvky, jejichž rank je mezi $(2 \uparrow (k - 1)) + 1$ a $2 \uparrow k$. Např. třetí skupina obsahuje ty prvky, jejichž rank je mezi 5 a 16. Prvky jsou tedy rozděleny do $1 + \log^* \log N = \mathcal{O}(\log^* N)$ skupin. Odhadněme shora počet

prvků v k -té skupině:

$$\begin{aligned} \frac{N}{2^{2\uparrow(k-1)+1}} + \dots + \frac{N}{2^{2\uparrow k}} &= \frac{N}{2^{2\uparrow(k-1)}} \cdot \left(\sum_{i=1}^{2\uparrow k - 2\uparrow(k-1)} \frac{1}{2^i} \right) \leq \\ &\leq \frac{N}{2^{2\uparrow(k-1)}} \cdot 1 = \frac{N}{2\uparrow k}. \end{aligned}$$

Ted' můžeme provést časovou analýzu funkce $root(v)$. Čas, který spotřebuje funkce $root(v)$, je přímo úměrný délce cesty od prvku v ke kořeni stromu. Tato cesta je pak následně rozpojena a všechny prvky na ní jsou přepojeny přímo na kořen stromu. Rozdělíme rozpojené hrany této cesty na ty, které „naučtujeme“ tomuto volání funkce $root(v)$, a ty, které zahrneme do faktoru $\mathcal{O}(N \log^* N)$ v dokazovaném časovém odhadu. Do volání funkce $root(v)$ započítáme ty hrany cesty, které spojují dva prvky, které jsou v různých skupinách. Takových hran je zřejmě nejvýše $\mathcal{O}(\log^* N)$ (všimněte si, že ranky prvků na cestě z listu do kořene tvoří rostoucí posloupnost).

Uvažme prvek v v k -té skupině, který již není kořenem stromu. Při každém přepojení rank rodiče prvku v vzroste. Tedy po $2\uparrow k$ přepojeních je rodič prvku v v $(k+1)$ -ní nebo vyšší skupině. Pokud v je prvek v k -té skupině, pak hrana z něj na cestě do kořene nebude účtována volání funkce $root(v)$ nejvýše $(2\uparrow k)$ -krát. Protože k -tá skupina obsahuje nejvýše $N/(2\uparrow k)$ prvků, je počet takových hran pro všechny prvky této skupiny nejvýše N . A protože počet skupin je nejvýše $\mathcal{O}(\log^* N)$, je celkový počet hran, které nejsou započítány voláním funkce $root(v)$, nejvýše $\mathcal{O}(N \log^* N)$. Protože funkce $root(v)$ je volána $2L$ -krát, plyne časový odhad $\mathcal{O}((N+L) \log^* N)$ z právě dokázaných tvrzení.

Inverzní Ackermannova funkce $\alpha(N)$

Ackermannovu funkci lze definovat následující konstrukcí:

$$A_0(i) = i + 1, \quad A_{k+1}(i) = A_k^i(i) \text{ pro } k \geq 0,$$

kde výraz A_k^i zastupuje složení i funkcí A_k , např. $A_1(3) = A_0(A_0(A_0(3)))$. Platí tedy následující rovnosti:

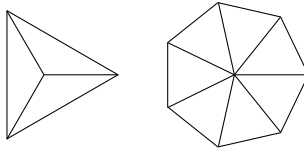
$$A_0(i) = i + 1, \quad A_1(i) = 2i, \quad A_2(i) = 2^i \cdot i.$$

Ackermannova funkce s jedním parametrem $A(k)$ je pak rovna hodnotě $A_k(2)$, čili $A(2) = A_2(2) = 8$, $A(3) = A_3(2) = 2^{11}$, $A(4) = A_4(2) \approx 2\uparrow 2048$ atd. . . Hodnota inverzní Ackermannovy funkce $\alpha(N)$ je tedy nejmenší přirozené číslo k takové, že $N \leq A(k) = A_k(2)$. Jak je vidět, ve všech reálných aplikacích platí, že $\alpha(N) \leq 4$.

Dan Král', Martin Mareš a Milan Straka

Úloha 20-1-4: Kormidlo

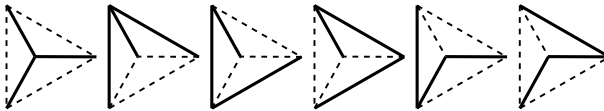
Správné námořnické kormidlo s N loukotěmi je v podstatě pravidelný N -úhelník, jehož střed je spojen s každým z N bodů na obvodu. Skládá se tedy z $2N$ rovných dílů. Kormidlo s třemi a sedmi loukotěmi si můžete prohlédnout na následujícím obrázku.



Vilda chce ale kormidlo opravit co nejdříve, a tak se rozhodl, že ho sestaví neúplně – pouze z N rovných dílů. Přitom chce, aby z každého z $N + 1$ vrcholů kormidla vedl alespoň jeden díl a všech N rovných dílů bylo navzájem spojeno (tj. kormidlo se nerozpadá na dva či více dílů).

Napište program, který dostane na vstupu $N \geq 3$. Výstupem vašeho programu by měl být počet způsobů, kterými může sestavit Vilda kormidlo s N loukotěmi z N dílů tak, aby z každého vrcholu neúplného kormidla vedl alespoň jeden díl a kormidlo se nerozpadalo na více částí.

Příklad: Pro $N = 3$ lze kormidlo sestavit 16-ti způsoby. Jsou to



posledních 5 ve 3 otočeních.

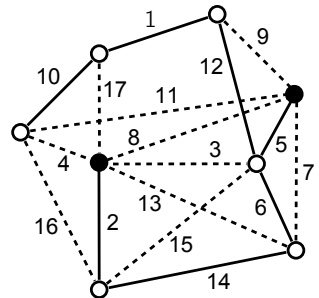
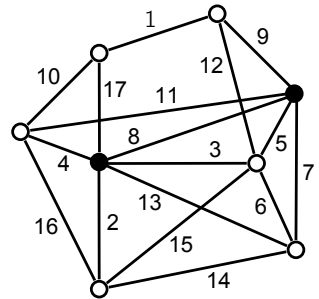
Úloha 20-5-4: Dračí chodbičky

Spleť dračích chodeb a jeskyní si představíme jako graf, kde vrcholy jsou jeskyně nebo křižovatky a hrany jsou tunely.

Drak by rád co nejvíc chodeb zasypal, ale zároveň chce, aby se dostal do všech jeskyní (vrcholů). Také vám dává seznam míst, ve kterých má část pokladu. K takovým místům by chtěl nechat pouze jednu přístupovou chodbu (tj. z těchto vrcholů mají být listy).

Navrhněte, které chodby by měl drak zachovat, aby součet délek zasypaných chodeb byl největší možný. Můžete předpokládat, že zadaný problém má řešení (tzn. z vrcholů s pokladem lze udělat listy, aniž by se graf rozpadl na více komponent).

Příklad: Vpravo nahoře je obrázek současného stavu tunelů v Ohnivé hoře (místa s pokladem jsou vyznačena černě). Dole pak vidíte výsledek (zasypané chodby jsou čárkované).



Rozděl a panuj

Často se setkáme s úlohami, které lze snadno rozdělit na nějaké menší úlohy a z jejich výsledků zase snadno složit výsledek původní velké úlohy. Přitom menší úlohy můžeme řešit opět tímž algoritmem (zavoláme si ho rekurzivně), leda by již byly tak maličké, že dokážeme odpovědět triviálně bez jakéhokoliv počítání. Zkrátka jak říkali staří římsí císaři: Divide et impera. Uvedme si pro začátek jeden staronový příklad:

Quicksort

QuickSort (alias QS) je algoritmus pro třídění posloupnosti prvků. Už o něm byla jednou řeč v „třídící kuchařce“. Tentokrát se na něj podíváme trochu podrobněji a navíc nám poslouží jako ingredience pro další algoritmy.

QS v každém svém kroku zvolí nějaký prvek (budeme mu říkat *pivot*) a přerovná prvky v posloupnosti tak, aby napravo od pivota byly pouze prvky větší než pivot a nalevo pouze menší. Pokud se vyskytnou prvky rovné pivotu, můžeme si dle libosti vybrat jak levou, tak pravou stranu posloupnosti, funkčnost algoritmu to nijak neovlivní. Tento postup pak rekurzivně zopakujeme zvlášť pro prvky nalevo a zvlášť pro prvky napravo od pivota, a tak získáme setříděnou posloupnost.

Implementací QS je mnoho a mimo jiné se liší způsobem volby pivota. My si předvedeme jinou, než jsme ukazovali v třídící kuchařce (hlavně proto, že se nám z ní pak budou snadno odvozovat další algoritmy) a pro jednoduchost budeme jako pivota volit poslední prvek zkoumaného úseku:

```
{ budeme třídít takováto pole }
type Pole=array[1..MaxN] of Integer;

{ přerovnávací procedura pro úsek a[l..r] }
function prerovnej(a: Pole; l, r: integer): integer;
var i, j, x, q: integer;
begin
  { pivotem se stane poslední prvek úseku }
  x:=a[r];           { hodnota pivota }
  i:=l-1;           { a[i] bude vždy poslední <= pivotovi }

  { samotné přerovnávání }
  for j:=l to r-1 do
    if a[j]<=x then  { právě probíraný prvek }
      begin        { menší/rovný hodnotě pivota }
        i:=i+1;    { pak zvyš ukazatel }
        q:=a[j];   { a proved' přerovnáání prvku }
        a[j]:=a[i];
        a[i]:=q;
      end;
  end;

  { nakonec přesuneme pivota za poslední <= }
  q:=a[r];
```

```

a[r]:=a[i+1];
a[i+1]:=q;
prerovnej:=i+1;      { vrátíme novou pozici pivota }
end;

{ hlavní třídící procedura, třídí a[l..r] }
procedure QuickSort(a: Pole; l, r: integer);
var m: integer;
begin
  if l<r then begin      { máme ještě co dělat? }
    m:=prerovnej(l,r);  { m pozice pivota }
    QuickSort(l,m-1);   { setříd' prvky napravo }
    QuickSort(m+1,r);   { setříd' prvky nalevo }
  end;
end;
end;
```

Bohužel volit pivota právě takto je docela nešikovné, protože se nám snadno může stát, že si vybereme nejmenší nebo největší prvek v úseku (rozmyslete si, jak by vypadala posloupnost, ve které to nastane pokaždé), takže dostaneme-li posloupnost délky N , rozdělíme ji na úseky délek $N-2$ a 1 (pivot se nepočítá), načež pokračujeme s úsekem délky $N-2$, ten rozdělíme na $N-4$ a 1 , atd. Přitom pokaždé na přerovnání spotřebujeme čas lineární s velikostí úseku, celkem tedy $\mathcal{O}(N + (N-2) + (N-4) + \dots + 1) = \mathcal{O}(N^2)$.

Na druhou stranu pokud bychom si za pivota vybrali vždy *medián* z právě probíraných prvků (tj. prvek, který by se v setříděné posloupnosti nacházel uprostřed; pro sudý počet prvků zvolíme libovolný z obou prostředních prvků), dosáhneme daleko lepší složitosti $\mathcal{O}(N \log N)$. To dokážeme snadno:

Přerovnávací část algoritmu běží v čase lineárním vůči počtu prvků, které máme přerovnat. V prvním kroku QS pracujeme s celou posloupností, čili přerovnáme celkem N prvků. Následuje rekurzivní volání pro levou a pravou část posloupnosti (obě dlouhé $(N-1)/2 \pm 1$); přerovnávání v obou částech dohromady trvá opět $\mathcal{O}(N)$ a vzniknou tím části dlouhé nejvýše $N/4$. Zanoříme-li se v rekurzi do hloubky k , pracujeme s částmi dlouhými nejvýše $N/2^k$, které dohromady dávají nejvýše N (všechny části dohromady dávají prvky vstupní posloupnosti bez těch, které jsme si už zvolili jako pivoty).

V hloubce $\lceil \log_2 N \rceil$ už jsou všechny části nejvýše jednoprvkové, takže se rekurze zastaví. Celkem tedy máme $\lceil \log_2 N \rceil$ hladin (hloubek) a na každé z nich trávíme lineární čas, dohromady $\mathcal{O}(N \log N)$.

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

- *Naučit se počítat medián. Ale jak?*
- *Spokojit se se „lžimediánem“:* Kdybychom si místo mediánu vybrali libovolný prvek, který bude v setříděné posloupnosti „v prostřední polovině“ (čili alespoň čtvrtina prvků bude větší a alespoň čtvrtina menší než on), získáme také složitost

$\mathcal{O}(N \log N)$, neboť úsek délky N rozložíme na úseky, které budou mít délky nejvýše $(1 - 1/4) \cdot N$, takže na k -té hladině budou úseky délek nejvýše $(3/4)^k \cdot N$, čili hladin bude maximálně $\log_{3/4} N = \mathcal{O}(\log N)$. Místo $1/4$ by fungovala i libovolná jiná konstanta mezi nulou a jedničkou, ale ani to nám nepomůže k tomu, abychom uměli lžimedián najít.

- *Recyklovat pravidlo* typu „vezmi poslední prvek“ a jen ho trochu vylepšit. To bohužel nebude fungovat, protože pokud budeme při výběru pivotu hledět jenom na konstantní počet prvků, bude poměrně snadné přijít na vstup, pro který toto pravidlo bude dávat kvadratickou složitost, i když obvykle půjde dokázat, že takových vstupů je „málo“. (Také se tak často QS implementuje.)
- *Volit pivota náhodně* ze všech prvků zkoumaného úseku. K náhodné volbě samozřejmě potřebujeme náhodný generátor a s těmi je to svízelné, ale zkusme na chvíli věřit, že jeden takový máme nebo alespoň že máme něco s podobnými vlastnostmi. Jak nám to pomůže? Náhodně zvolený pivot nebude sice přesně uprostřed, ale s pravděpodobností $1/2$ to bude lžimedián, takže po průměrně dvou hladinách se ke lžimediánu dopracujeme (rozmyslete si, proč, nebo nahlédněte do seriálu o pravděpodobnostních algoritmech v 16. ročníku KSP). Proto časová složitost takového randomizovaného QS bude v průměru 2-krát větší, než lžimediánového QS, čili v průměru také $\mathcal{O}(N \log N)$. Jednoduše řečeno, zatímco fixní pravidlo nám dalo dobrý čas pro průměrný vstup, ale existovaly vstupy, na kterých bylo pomalé, randomizování nám dává dobrý průměrný čas pro všechny možné vstupy.

Hledání k -tého nejmenšího prvku

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

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

O něco rychlejší řešení je založeno na výše zmíněném algoritmu QuickSort (často se mu proto říká *QuickSelect*). Opět si vybereme pivotu a posloupnost rozdělíme na prvky menší než pivot, pivotu a prvky větší než pivot (pro jednoduchost budeme předpokládat, že žádné dva prvky posloupnosti nejsou stejné). Pokud se pivot nalézá na k -té pozici, je to hledaný k -tý nejmenší prvek posloupnosti, protože právě $k - 1$ prvků je menších.

Zbývají dva případy, kdy tomu tak není. Pakliže je pozice pivotu v posloupnosti větší než k , pak se hledaný prvek nalézá nalevo od pivotu a postačí rekurzivně najít k -tý nejmenší prvek mezi prvky nalevo. V opačném případě, kdy je pozice pivotu menší než k , je hledaný prvek v posloupnosti napravo od pivotu. Mezi těmito prvky však nebudeme hledat k -tý nejmenší prvek, ale $(k - p)$ -tý nejmenší prvek, kde p je pozice pivotu v posloupnosti.

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

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

```
function kty(var a: Pole; l, r, k: integer): integer;
var x, z: integer;
begin
  x:=prerovnej(a,l,r);   { x je pozice pivotu }
  z:=x-1+1;             { pozice pivotu vzhledem k [l..r] }
  if k=z then
    kty:=a[x]           { k-tý nejmenší je pivot }
  else if k<z then
    kty:=kty(a,l,x-1,k) { k-tý nejmenší je nalevo }
  else
    kty:=kty(a,x+1,r,k-z); { napravo }
end;
```

***k*-tý nejmenší podruh, tentokrát lineárně a bez náhody**

Existuje však algoritmus, který řeší naši úlohu lineárně, a to i v nejhorším případě. Je založený na ďábelském triku: zvolit vhodného pivotu (jak ukážeme, bude to jeden ze lžimediánů) rekurzivním voláním téhož algoritmu. Zařídíme to takto:

- Pokud jsme dostali méně než 6 prvků, použijeme nějaký triviální algoritmus, například si posloupnost setřídíme a vrátíme k -tý prvek setříděné posloupnosti.
- Rozdělíme prvky posloupnosti na pětičky; pokud není počet prvků dělitelný pěti, poslední pětičky necháme nekompletní.
- Spočítáme medián každé pětičky. To můžeme provést například rekurzivním zavoláním celého našeho algoritmu, čili v důsledku třídění. (Také bychom si mohli pro 5 prvků zkonstruovat rozhodovací strom s nejmenším možným počtem porovnání, což je rychlejší, ale jednak pouze konstanta-krát, jednak je to daleko pracnější.)
- Máme tedy $N/5$ mediánů. V nich rekurzivně najdeme medián m (označíme mediány pětiček za novou posloupnost a na ní začneme opět od prvního bodu).
- Přerovnáme vstupní posloupnost po quicksortovsku a jako pivotu použijeme prvek m . Po přerovnání je pivot, podobně jako v předchozím algoritmu, na $(z+1)$ -ní pozici v posloupnosti, kde z je počet prvků s menší hodnotou, než má pivot.
- Podobně jako u předchozího algoritmu, pokud je $k = z + 1$, pak je pivot m k -tým nejmenším prvkem posloupnosti. Není-li tomu tak a $k < z + 1$, budeme hledat k -tý nejmenší prvek mezi prvními z členy, v opačném případě, kdy $k > z + 1$, vyhledáme $(k - z + 1)$ -tého nejmenšího mezi posledními $n - z - 1$ prvky.

Řečeno s panem Pascalem:

```
{ potřebujeme přerovnávací funkci, která
  dostane hodnotu pivota jako parametr }
function prerovnejp(var a: Pole; l, r, m: integer): integer;
var q, p: integer;
begin
  { nalezneme pozici pivota }
  p:=l;
  while a[p]<>m do
    p:=p+1;
  { pivota prohodíme s posledním prvkem }
  q:=a[p]; a[p]:=a[r]; a[r]:=q;
  { a zavoláme původní přerovnávací fci }
  prerovnejp := prerovnej(a,l,r);
end;

{ hledání k-tého nejmenšího prvku z a[l..r] }
function kth(var a: Pole; l, r, k: integer): integer;
var medp:Pole;          { pole pro mediány pětic }
    i, j, q, x, pocet, m, z: integer;
begin
  pocet:=r-l+1;          { s kolika prvky pracujeme }
  if pocet<=1 then      { pouze jeden prvek? }
    kth:=a[l]           { výsledek nemůže být jiný }
  else if pocet<6 then begin { méně než 6 prvků }
    QuickSort(a,l,r);
    kth:=a[l+k-1];
  end
  else begin            { mnoho prvků, jde to tuhého }
    { rozdělíme prvky do pětic }
    q:=1;              { zatím máme jednu pětici }
    i:=1;              { levý okraj první pětice }
    j:=i+4;           { pravý okraj první pětice }
    while j<=r do begin { procházíme celé pětice }
      QuickSort(a,i,j);
      medp[q]:=a[i+2]; { medián pětice }
      q:=q+1;          { zvyš počet pětic }
      i:=i+5;          { nastav levý okraj pětice }
      j:=j+5;          { nastav pravý okraj pětice }
    end;
    { případnou neúplnou pětici můžeme ignorovat }

    m:=kth(medp,1,q-1,q div 2); { najdeme medián mediánů pětic }
    x:=prerovnejp(a,l,r,m); { přerovnej a zjisti, kde skončil pivot }
    z:=x-1+1;           { pozice vzhledem k [l..r] }
  end;
end;
```

```

if k=z then
  kth:=m           { k-tý nejmenší je pivot }
else if k<z then
  kth:=kth(a,l,x-1,k) { k-tý nejmenší nalevo }
else
  kth:=kth(a,x+1,r,k-z); { napravo }
end;
end;

```

Zbývá dokázat, že tato dvojitá rekurze má slíbenou lineární složitost. Zkusme se proto podívat, kolik prvků poslopnosti po přerovnání je větších než prvek m . Všech pětic je $N/5$ a alespoň polovina z nich (tedy $N/10$) má medián menší než m . V každé takové pětici pak navíc najdeme dva prvky menší než medián pětice, takže celkem existuje alespoň $3/10 \cdot N$ prvků menších než m . Větších tedy může být maximálně $7/10 \cdot N$. Symetricky ukážeme, že i menších prvků může být nejvýše $7/10 \cdot N$.

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

$$t(N) \leq cN + t(N/5) + t(7/10 \cdot N).$$

Nyní zbývá tuto rekurzivní nerovnici vyřešit, což provedeme drobným úskokem: uhadneme, že výsledkem bude lineární funkce, tedy že $t(N) = dN$ pro nějaké $d > 0$. Dostaneme:

$$dN \leq (c + 1/5 \cdot d + 7/10 \cdot d) \cdot N.$$

To platí např. pro $d = 10c$, takže opravdu $t(N) = \mathcal{O}(N)$. (Lépe už to nepůjde, jelikož na každé číslo se musíme podívat alespoň jednou.)

Cvičení

- Při hledání k -tého nejmenšího prvku jsme předpokládali, že všechny prvky jsou různé. Prohlédněte si algoritmy pozorně a rozmyslete si, že budou fungovat i bez toho. Opravdu?
- Proč jsme zvolili zrovna pětice? Jak by to dopadlo pro trojice? A jak pro sedmice? Fungoval by takový algoritmus? Byl by také lineární?
- Ve výpočtu $t(N)$ jsme si nedali pozor na neúplné pětice a také jsme předpokládali, že pětic je sudý počet. Ono se totiž nic zlého nemůže stát. Jak se to snadno nahlédne? Proč nestačí na začátku doplnit vstup „nekonečný“ na délku, která je mocninou deseti?
- Kdybychom neuhodli, že $t(N)$ je lineární, jak by se na to dalo přijít?

Násobení dlouhých čísel

Dalším pěkným příkladem na rozdělování a panování je násobení dlouhých čísel – tak dlouhých, že se už nevejdou do integeru, takže s nimi musíme počítat po číslicích (ať už v jakékoliv soustavě – teď zvolíme desítkovou, často se hodí třeba 256-ková). Klasickým „školním“ algoritmem pro násobení na papíře to zvládneme na kvadratický počet operací, zde si předvedeme efektivnější způsob.

Libovolné $2N$ -ciferné číslo můžeme zapsat jako $10^N A + B$, kde A a B jsou N -ciferná. Součin dvou takových čísel pak bude $(10^N A + B) \cdot (10^N C + D) = (10^{2N} AC + 10^N(AD + BC) + BD)$. Sčítat dokážeme v lineárním čase, násobit mocninou deseti také (dopíšeme příslušný počet nul na konec čísla), N -ciferná čísla budeme násobit rekurzivním zavoláním téhož algoritmu. Pro časovou složitost tedy bude platit $t(N) = cN + 4t(N/2)$. Nyní tuto rovnici můžeme snadno vyřešit, ale ani to dělat nebudeme, neboť nám vyjde, že $t(N) \approx N^2$, čili jsme si oproti původnímu algoritmu vůbec nepomohli.

Přijde trik. Místo čtyř násobení čísel poloviční délky nám budou stačit jen tři: spočteme AC , BD a $(A+B) \cdot (C+D) = AC + AD + BC + BD$, přičemž pokud od posledního součinu odečteme AC a BD , dostaneme přesně $AD + BC$, které jsme předtím počítali dvěma násobeními. Časová složitost nyní bude $t(N) = c'N + 3t(N/2)$. (Konstanta c' je o něco větší než c , protože přibýlo sčítání a odčítání, ale stále je to konstanta. My si ovšem zvolíme jednotku času tak, aby bylo $c' = 1$, a ušetříme si tak spoustu psaní.)

Jak naši rovnici vyřešíme? Zkusíme ji dosadit do sebe samé a pozorovat, co se bude dít:

$$\begin{aligned} t(N) &= N + 3(N/2 + 3t(N/4)) = \\ &= N + 3/2 \cdot N + 9t(N/4) = \\ &= N + 3/2 \cdot N + 9/4 \cdot N + 27t(N/8) = \dots = \\ &= N + 3/2 \cdot N + \dots + 3^{k-1}/2^{k-1} \cdot N + 3^k t(N/2^k). \end{aligned}$$

Pokud zvolíme $k = \log_2 N$, vyjde $N/2^k = 1$, čili $t(N/2^k) = t(1) = d$, kde d je nějaká konstanta. To znamená, že:

$$t(N) = N \cdot (1 + 3/2 + 9/4 + \dots + (3/2)^{k-1}) + 3^k d.$$

Výraz v závorce je součet prvních k členů geometrické řady s kvocientem $3/2$, čili $((3/2)^k - 1)/(3/2 - 1) = \mathcal{O}((3/2)^k)$. Tato funkce však roste pomaleji než zbylý člen $3^k d$, takže ji klidně můžeme zanedbat a zabývat se pouze oním posledním členem: $3^k = 2^{k \log_2 3} = 2^{\log_2 N \cdot \log_2 3} = (2^{\log_2 N})^{\log_2 3} = N^{\log_2 3} \approx N^{1.58}$. Konstanta d se nám „schová do \mathcal{O} -čka“, takže algoritmus má časovou složitost přibližně $\mathcal{O}(N^{1.58})$. Existují i rychlejší algoritmy se složitostí až $\mathcal{O}(N)$, ale ty jsou mnohem ďábelštější a pro malá N se to sotva vyplatí.

Program si pro dnešek odпустíme, šetříme naše lesy (tedy alespoň trošku).

David Matoušek a Martin Mareš

Úloha 19-2-5: Hluboký les

Pomozte nám v hledání nejhlubšího lesa. Ten se nachází na místě, kde jsou dva stromy, které jsou u sebe nejbližší ze všech v lese. Váš program dostane na vstupu číslo N a dále N řádků s reálnými souřadnicemi jednotlivých stromů. V případě, že je dvojic nejbližších stromů víc, stačí vypsát libovolnou z nich.

Příklad: Pro $N = 4$ a stromy

```
1 3
2 1
3 1
4 3
```

by měl program vypsat: Stromy 2 a 3 jsou si k sobě nejbliže.

Úloha 19-5-5: Počet inverzí

Je dána posloupnost celých čísel P_1, P_2, \dots, P_N . Čísla P_i a P_j jsou v *inverzi*, pokud $i < j$ a zároveň $P_i > P_j$. Inverze je tedy porucha ve vzestupném uspořádání posloupnosti. Vaším úkolem je zjistit, kolik inverzí posloupnost obsahuje.

Na prvním řádku vstupu je číslo N , na druhém řádku následuje N celých čísel v desítkovém zápisu oddělených mezerami. Počet inverzí vypište na standardní výstup. Čísel v posloupnosti je maximálně 100 000.

Příklad: Pro vstup:

```
5
4 5 3 1 2
```

vypište na standardní výstup 8.

Dynamické programování

Rekurzivní funkce je taková funkce, která při svém běhu volá sama sebe, často i více než jednou, což v důsledku může vést na exponenciální algoritmus. Dynamické programování je technika, kterou jde z pomalého rekurzivního algoritmu vyrobit pěkný polynomiální (až na výjimečné případy). Ale nepředbíhejme, nejdříve se podíváme na jednoduchý příklad rekurze:

Fibonacciho čísla

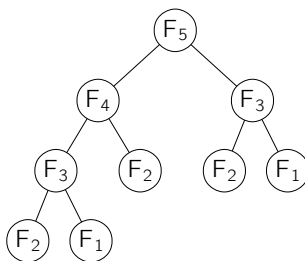
Budeme počítat n -té číslo Fibonacciho posloupnosti. To je posloupnost, jejíž první dva členy jsou jedničky a každý další člen je součtem dvou předchozích. Začíná takto:

1 1 2 3 5 8 13 21 34 55 89 ...

Pro nalezení n -tého členu (ten budeme značit F_n) si napíšeme rekurzivní funkci `Fibonacci(n)`, která bude postupovat přesně podle definice: zeptá se sama sebe rekurzivně, jaká jsou dvě předchozí čísla, a pak je sečte. Možná více řekne program:

```
function Fibonacci(n: integer): integer;
begin
  if n <= 2 then
    Fibonacci := 1
  else
    Fibonacci := Fibonacci(n-1) + Fibonacci(n-2)
end;
```

To, jak funkce volá sama sebe, si můžeme snadno nakreslit třeba pro výpočet čísla F_5 :



Vidíme, že program se rozvětjuje a tvoří strom volání. Všimněme si také, že některé podstromy jsou shodné. Zřejmě to budou ty části, které reprezentují výpočet stejného Fibonacciho čísla – v našem příkladě třeba třetího.

Pokusme se odhadnout časovou složitost T_n naší funkce. Pro $n = 1$ a $n = 2$ funkce skončí hned, tedy v konstantním (řekněme jednotkovém) čase. Pro vyšší n zavolá sama sebe pro dva předchozí členy plus ještě spotřebuje konstantní čas na sčítání:

$$T_n \geq T_{n-1} + T_{n-2} + \text{const}, \text{ a proto } T_n \geq F_n.$$

Tedy na spočítání n -tého Fibonacciho čísla spotřebujeme čas alespoň takový, kolik je ono číslo samo. Ale jak velké takové F_n vlastně je? Můžeme třeba využít toho, že:

$$F_n = F_{n-1} + F_{n-2} \geq 2 \cdot F_{n-2},$$

z čehož plyne:

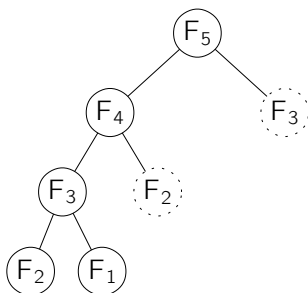
$$F_n \geq 2^{n/2}.$$

Funkce **Fibonacci** má tedy exponenciální časovou složitost, což není nic vítaného (ukázali jsme sice jen dolní odhad, není však těžké přijít na to, že i v nejhorším případě poběží exponenciálně pomalu). Ovšem jak jsme už řekli, některé výpočty opakujeme stále dokola. Nenabízí se proto nic snazšího, než si tyto mezivýsledky uložit a pak je vytáhnout jako pověstného králíka z klobouku s minimem námahy.

Bude nám k tomu stačit jednoduché pole P o n prvcích, na počátku inicializované nulami. Kdykoliv budeme chtít spočítat některý člen, nejdříve se podíváme do pole, zda jsme ho již jednou nespočetli. A naopak jakmile hodnotu spočítáme, hned si ji do pole poznamenejme:

```
var P: array[1..MaxN] of integer;
function Fibonacci(n: integer): integer;
begin
  if P[n] = 0 then
    begin
      if n <= 2 then
        P[n] := 1
      else
        P[n] := Fibonacci(n-1) + Fibonacci(n-2)
      end;
      Fibonacci := P[n]
    end;
end;
```

Podívejme se, jak vypadá strom volání nyní:



Na každý člen posloupnosti se tentokrát ptáme maximálně dvakrát – k výpočtu ho potřebují dva následující členy. To ale znamená, že funkci **Fibonacci** zavoláme maximálně $2n$ -krát, čili jsme touto jednoduchou úpravou zlepšili exponenciální složitost na lineární.

Zdálo by se, že abychom získali čas, museli jsme obětovat paměť, ale to není tak úplně pravda. V prvním příkladu sice nepoužíváme žádné pole, ale při volání funkce si musíme zapamatovat některé údaje, jako je třeba návratová adresa, parametry funkce a její lokální proměnné, a na to samotné potřebujeme určité paměť lineární s hloubkou vnoření, v našem případě tedy lineární s n .

Určitě vás už také napadlo, že n -té Fibonacciho číslo se dá snadno spočítat i bez rekurze. Stačí prvky našeho pole P plnit od začátku – kdykoliv známe $P[1] = F_1, \dots, F_k = P[k]$, dokážeme snadno spočítat i $P[k + 1] = F_{k+1}$:

```
function Fibonacci(n: integer): integer;
var
  P: array[1..MaxN] of integer;
  i: integer;
begin
  P[1] := 1;
  P[2] := 1;
  for i := 3 to n do
    P[i] := P[i-1] + P[i-2];
  Fibonacci := P[n]
end;
```

Zopakujme si, co jsme postupně udělali: nejprve jsme vymysleli pomalou rekurzivní funkci, tu jsme zrychlili zapamatováváním si mezivýsledků a nakonec jsme celou rekurzi „obrátili naruby“ a mezivýsledky počítali od nejmenšího k největšímu, aniž bychom se starali o to, jak se na ně původní rekurze ptala.

V případě Fibonacciho čísel je samozřejmě snadné přijít rovnou na nerekurzivní řešení (a dokonce si všimnout, že si stačí pamatovat jen poslední dvě hodnoty a pamětovou složitost tak zredukovat na konstantní), ale zmíněný obecný postup zrychlování rekurze nebo rovnou řešení úlohy od nejmenších podproblémů k těm největším – obvykle se mu říká *dynamické programování* – funguje i pro řadu složitějších úloh. Třeba na tuto:

Problém batohu

Je dáno N předmětů o hmotnostech m_1, \dots, m_N (celočíselných) a také číslo M (hmotnost batohu). Úkolem je vybrat některé z předmětů tak, aby součet jejich hmotností byl co největší, a přitom nepřekročil M . Předvedeme si algoritmus, který tento problém řeší v čase $\mathcal{O}(MN)$.

Náš algoritmus bude používat pomocné pole $A[0 \dots M]$ a jeho činnost bude rozdělena do N kroků. Na konci k -tého kroku bude prvek $A[i]$ nenulový právě tehdy, jestliže z prvních k předmětů lze vybrat předměty, jejichž součet hmotností je přesně i . Před prvním krokem (po nultém kroku), jsou všechny hodnoty $A[i]$ pro $i > 0$ nulové a $A[0]$ má nějakou nenulovou hodnotu, řekněme -1 . Všimněme si, jak kroky algoritmu odpovídají podúlohám, které řešíme: v prvním kroku vyřešíme podúlohu tvořenou jen prvním předmětem, ve druhém kroku prvními dvěma předměty, pak prvními třemi předměty, atd.

Popišme si nyní k -tý krok algoritmu. Pole A budeme procházet od konce, tj. od $i = M$. Pokud je hodnota $A[i]$ stále nulová, ale hodnota $A[i - m_k]$ je nenulová, změňme hodnotu uloženou v $A[i]$ na k (později si vysvětlíme, proč zrovna na k).

Nyní si rozmyslíme, že po provedení k -tého kroku odpovídají nenulové hodnoty v poli A hmotnostem podmnožin z prvních k předmětů (*podmnožina* je v podstatě jen

výběr nějaké části předmětů). Pokud je hodnota $A[i]$ nenulová, pak buď byla nenulová před k -tým krokem (a v tom případě odpovídá hmotnosti nějaké podmnožiny prvních $k-1$ předmětů) anebo se stala nenulovou v k -tém kroku. Potom ale hodnota $A[i-m_k]$ byla před k -tým krokem nenulová, a tedy existuje podmnožina prvních $k-1$ předmětů, jejíž hmotnost je $i-m_k$. Přidáním k -tého předmětu k této podmnožině vytvoříme podmnožinu předmětů hmotnosti přesně i .

Naopak, pokud lze vytvořit podmnožinu X hmotnosti i z prvních k předmětů, pak takovou podmnožinu X lze buď vytvořit jen z prvních $k-1$ předmětů, a tedy hodnota $A[i]$ je nenulová již před k -tým krokem, anebo k -tý předmět je obsažen v takové množině X . Potom ale hodnota $A[i-m_k]$ je nenulová před k -tým krokem (hmotnost podmnožiny X bez k -tého prvku je $i-m_k$) a hodnota $A[i]$ se stane nenulovou v k -tém kroku.

Po provedení všech N kroků odpovídají nenulové hodnoty $A[i]$ přesně hmotnostem podmnožin ze všech předmětů, co máme k dispozici. Speciálně největší index i_0 takový, že hodnota $A[i_0]$ je nenulová, odpovídá hmotnosti nejtěžší podmnožiny předmětů, která nepřekročí hmotnost M . Nalézt jednu množinu této hmotnosti také není obtížné: protože v k -tém kroku jsme měnili nulové hodnoty v poli A na hodnotu k , tak v $A[i_0]$ je uloženo číslo jednoho z předmětů nějaké takové množiny, v $A[i_0 - m_{A[i_0]}]$ číslo dalšího předmětu, atd. Zdrojový kód tohoto algoritmu lze nalézt níže.

Časová složitost algoritmu je $\mathcal{O}(NM)$, neboť se skládá z N kroků, z nichž každý vyžaduje čas $\mathcal{O}(M)$. Paměťová složitost činí $\mathcal{O}(N+M)$, což představuje paměť potřebnou pro uložení pomocného pole A a hmotností daných předmětů.

```

var N: integer; { počet předmětů }
    M: integer; { hmotnostní omezení }
    hmotnost: array[1..N] of integer; { hmotnosti daných předmětů }
    A: array[0..M] of integer;
    i, k: integer;
begin
  A[0] := -1;
  for i:=1 to M do A[i] := 0;
  for k:=1 to N do
    for i:=M downto hmotnost[k] do
      if (A[i-hmotnost[k]] <> 0) and (A[i]=0) then
        A[i] := k;
  i:=M; while A[i]=0 do i:=i-1;
  writeln('Maximální hmotnost: ', i);
  write('Předměty v množině:');
  while A[i] <> -1 do begin
    write(' ', A[i]);
    i:=i-hmotnost[A[i]];
  end;
  writeln;
end.

```

Cvičení a poznámky

- Proč pole A procházíme pozadu a ne popředu?
- Složitost algoritmu vypadá jako polynomiální, což ve skutečnosti není úplně pravda. Závisí totiž na hodnotě M , která má na vstupu délku $\log M$. Algoritmům, jež jsou polynomiální vůči hodnotám na vstupu (a tedy exponenciální vůči délce vstupu), se říká *pseudopolynomiální*. Podrobnosti jsou v kuchařce o těžkých úlohách.

Nejkratší cesty a Floydův-Warshallův algoritmus

Náš další příklad bude z oblasti grafových algoritmů, ale zkusíme si ho nejdříve říci bez grafů:

Bylo-nebylo-je N měst. Mezi některými dvojicemi měst vedou (obousměrné) *silnice*, jejichž (nezáporné) délky jsou dány na vstupu. Předpokládáme, že silnice se jinde než ve městech nepotkávají (pokud se kříží, tak mimoúrovňově). Úkolem je spočítat nejkratší vzdálenosti mezi všemi dvojicemi měst, tj. délky nejkratších cest mezi všemi dvojicemi měst.

Cestou rozumíme posloupnost měst takovou, že každá dvě po sobě následující města jsou spojené silnicí, a délka cesty je součet délek silnic, které tato města spojují. (V grafové terminologii tedy máme daný ohodnocený neorientovaný graf a chceme zjistit délky nejkratších cest mezi všemi dvojicemi jeho vrcholů.)

Půjdeme na to následovně: Vzdálenosti mezi městy jsou na začátku algoritmu uloženy ve dvourozměrném poli D , tj. $D[i][j]$ je vzdálenost z města i do města j . Pokud mezi městy i a j nevede žádná silnice, bude $D[i][j] = \infty$ (v programu bude tato hodnota rovna nějakému dostatečně velkému číslu). V průběhu výpočtu si budeme na pozici $D[i][j]$ udržovat délku nejkratší dosud nalezené cesty mezi městy i a j .

Algoritmus se skládá z N fází. Na konci k -té fáze bude v $D[i][j]$ uložena délka nejkratší cesty mezi městy i a j , která může procházet skrz libovolná z měst $1, \dots, k$. V průběhu k -té fáze tedy stačí vyzkoušet, zda je mezi městy i a j kratší stávající cesta přes města $1, \dots, k-1$, jejíž délka je uložena v $D[i][j]$, anebo nová cesta přes město k .

Pokud nejkratší cesta prochází přes město k , můžeme si ji rozdělit na nejkratší cestu z i do k a nejkratší cestu z k do j . Délka takové cesty je tedy rovna $D[i][k] + D[k][j]$. Takže pokud je součet $D[i][k] + D[k][j]$ menší než stávající hodnota $D[i][j]$, nahradíme hodnotu na pozici $D[i][j]$ tímto součtem, jinak ji ponecháme.

Z popisu algoritmu přímo plyne, že po N -té fázi je na pozici $D[i][j]$ uložena délka nejkratší cesty z města i do města j . Protože v každé z N fází algoritmu musíme vyzkoušet všechny dvojice i a j , vyžaduje každá fáze čas $\mathcal{O}(N^2)$. Celková časová složitost našeho algoritmu tedy je $\mathcal{O}(N^3)$. Co se paměti týče, vystačíme si s polem D a to má velikost $\mathcal{O}(N^2)$. Program bude vypadat následovně:

```

var N: integer; { počet měst }
    D: array[1..N] of array[1..N] of longint;
    { délky silnic mezi městy,
      D[i][i]=0, místo neexistujících je "nekonečno" }
    i, j, k: integer;
begin
    for k:=1 to N do
        for i:=1 to N do
            for j:=1 to N do
                if D[i][k]+D[k][j] < D[i][j] then
                    D[i][j]:=D[i][k] + D[k][j];
end.

```

Popišme si ještě, jak bychom postupovali, kdybychom kromě vzdáleností mezi městy chtěli nalézt i nejkratší cesty mezi nimi. To lze jednoduše vyřešit například tak, že si navíc budeme udržovat pomocné pole $E[i][j]$ a do něj při změně hodnoty $D[i][j]$ uložíme nejvyšší číslo města na cestě z i do j délky $D[i][j]$ (při změně v k -té fázi je to číslo k). Máme-li pak vypsát nejkratší cestu z i do j , vypíšeme nejprve cestu z i do $E[i][j]$ a pak cestu z $E[i][j]$ do j . Tyto cesty nalezneme stejným (rekurzivním) postupem.

Poznámky

- Popis algoritmu vysloveně svádí k „rejpnutí“: Jak víme, že spojením dvou cest, které provádíme, vznikne zase cesta (tj. že se na ní nemohou nějaké vrcholy opakovat)? To samozřejmě nevíme, ale všimněte si, že kdykoliv by to cesta nebyla, tak si ji nevybereme, protože původní cesta bez vrcholu k bude vždy kratší nebo alespoň stejně dlouhá ... tedy alespoň pokud se v naší zemi nevyskytuje cyklus záporné délky. (Což, pokud bychom chtěli být přesní, musíme přidat do předpokladů našeho algoritmu.)
- Pozor na pořadí cyklů – program vysloveně svádí k tomu, abychom psali cyklus pro k jako vnitřní ... jenže pak samozřejmě nebude fungovat.

Cvičení

- Jak by algoritmus fungoval, kdyby silnice byly jednosměrné?
- Na první pohled nejpřirozenější hodnota, kterou bychom mohli použít pro ∞ , je `maxint`. To ovšem nebude fungovat, protože $\infty + \infty$ přeteče. Stačí `maxint div 2`?
- Hodnoty v poli si sice přepisujeme pod rukama, takže by se nám mohly poplést hodnoty z předchozí fáze s těmi z fáze současné. Ale zachrání nás to, že čísla, o která jde, vyjdou v obou fázích stejně. Proč?

Nejdelší společná podposloupnost

Poslední příklad dynamického programování, který si předvedeme, se bude týkat posloupností. Mějme dvě posloupnosti čísel A a B . Chceme najít jejich nejdelší společnou podposloupnost, tedy takovou posloupnost, kterou můžeme získat z A i B odstraněním některých prvků. Například pro posloupnosti

$$A = 2\ 3\ 3\ 1\ 2\ 3\ 2\ 2\ 3\ 1\ 1\ 2$$

$$B = 3\ 2\ 2\ 1\ 3\ 1\ 2\ 2\ 3\ 3\ 1\ 2\ 2\ 3$$

je jednou z nejdelších společných podposloupností tato posloupnost:

$$C = 2\ 3\ 1\ 2\ 2\ 3\ 1\ 2.$$

Jakým způsobem můžeme takovou podposloupnost najít? Nejdříve nás asi napadne vygenerovat všechny podposloupnosti a ty pak porovnat. Jakmile si ale spočítáme, že všech podposloupností posloupnosti o délce n je 2^n (každý prvek nezávisle na ostatních buď použijeme, nebo ne), najdeme raději nějaké rychlejší řešení.

Zkusme využít následující myšlenku: vyřešíme tento problém pouze pro první prvek posloupnosti A . Pak najdeme řešení pro první dva prvky A , přičemž využijeme předchozích výsledků. Takto pokračujeme pro první tři, čtyři, ... až n prvků.

Nejprve si rozmyslíme, co všechno si musíme v každém kroku pamatovat, abychom z toho dokázali spočítat krok následující. Určitě nám nebude stačit pamatovat si pouze nejdelší podposloupnost, jenže množina všech společných podposloupností je už zase moc velká.

Podívejme se tedy detailněji, jak se změní tato množina při přidání dalšího prvku k A : Všechny podposloupnosti, které v množině byly, tam zůstanou a navíc přibude několik nových, končících právě přidaným prvkem. Ovšem my si podposloupnosti pamatujeme proto, abychom je časem rozšířili na nejdelší společnou podposloupnost, takže pokud známe nějaké dvě stejně dlouhé podposloupnosti P a Q končící nově přidaným prvkem v A a víme, že P končí v B dříve než Q , stačí si z nich pamatovat pouze P , jelikož v libovolném rozšíření Q -čka můžeme Q vyměnit za P a získat tím stejně dlouhou společnou podposloupnost.

Proto si stačí pro již zpracovaných a prvků posloupnosti A pamatovat pro každou délku l tu ze společných podposloupností $A[1 \dots a]$ a B délky l , která v B končí na nejlevějším možném místě, a dokonce nám bude stačit si místo celé podposloupnosti uložit jen pozici jejího konce v B . K tomu použijeme dvojrozměrné pole $D[a, l]$.

Ještě si dovolíme jedno malé pozorování: Koncové pozice uložené v poli D se zvětšují s rostoucí délkou podposloupnosti, čili $D[a, l] < D[a, l + 1]$, protože posloupnosti délky $l + 1$ nejsou ničím jiným než rozšířeními posloupností délky l o 1 prvek.

Teď již výpočet samotný: Pokud už známe celý a -tý řádek pole D , můžeme z něj získat $(a + 1)$ -ní řádek. Projdeme postupně posloupnost B . Když najdeme v B prvek $A[a + 1]$ (ten právě přidávaný do A), můžeme rozšířit všechny podposloupnosti končící před aktuální pozicí v B . Nás bude zajímat pouze ta nejdelší z nich, protože rozšířením všech kratších získáme posloupnost, jejíž koncová pozice je větší než koncová pozice některé posloupnosti, kterou již známe. Rozšíříme tedy tu nejdelší podposloupnost a uložíme ji místo původní podposloupnosti. Toto provedeme pro každý výskyt nového prvku v posloupnosti B . Všimněte si, že nemusíme procházet pole s podposloupnostmi stále od začátku, ale můžeme se v něm posouvat od nejmenší délky k největší.

Ukážeme si, jak vypadá zaplněné pole hodnotami při řešení problému s posloupnostmi z našeho příkladu. Řádky jsou pozice v A , sloupce délky podposloupností.

D	1	2	3	4	5	6	7	8	9	10	11	12
1	2	—	—	—	—	—	—	—	—	—	—	—
2	1	5	—	—	—	—	—	—	—	—	—	—
3	1	5	9	—	—	—	—	—	—	—	—	—
4	1	4	6	11	—	—	—	—	—	—	—	—
5	1	2	5	7	12	—	—	—	—	—	—	—
6	1	2	3	7	9	14	—	—	—	—	—	—
7	1	2	3	7	8	12	—	—	—	—	—	—
8	1	2	3	7	8	12	13	—	—	—	—	—
9	1	2	3	5	8	9	13	14	—	—	—	—
10	1	2	3	4	6	9	11	14	—	—	—	—
11	1	2	3	4	6	9	11	14	—	—	—	—
12	1	2	3	4	6	7	11	12	—	—	—	—

Zbývá popsat, jak z těchto dat zvládneme rekonstruovat hledanou nejdelší společnou podposloupnost (NSP). Ukážeme si to na našem příkladu: jelikož poslední nenulové číslo na posledním řádku je v 8. sloupci, má hledaná NSP délku 8. $D[12, 8] = 12$ říká, že poslední písmeno NSP je na pozici 12 v posloupnosti B . Jeho pozici v posloupnosti A určuje nejvyšší řádek, ve kterém se tato hodnota také vyskytuje, v našem případě je to řádek 12. Druhé písmeno tedy budeme určovat z $D[11, 7]$, třetí z $D[9, 6]$, atd. Jednou z hledaných podposloupností je:

```
poslupnost: 2 3 1 2 2 3 1 2
indexy v A: 1 2 4 5 7 9 10 12
indexy v B: 2 5 6 7 8 9 11 12
```

Ještě trochu konkrétněji:

```
program Podposlupnost;
var
  A, B, C: array[0..MaxN - 1] of Integer;
  LA, LB, LC: Integer; { Délky posloupností }
  D: array[0..MaxN, 1..MaxN] of Integer;
  I, J, L, MaxL, T: Integer;
begin
  ...
  if LA > LB then begin { A bude kratší z obou }
    C := A;
    A := B;
    B := C;
    T := LA;
    LA := LB;
    LB := T;
```



```

end;
for I := 1 to LA do
  D[0, I] := LB;
L := 0;
MaxL := 0;
for I := 1 to LA do begin
  for J := 1 to LA do
    D[I, J] := D[I - 1, J];
  L := 0;
  for J := 0 to LB - 1 do
    if B[J] = A[I - 1] then
      begin
        while (L = 0) or (D[I - 1, L] < J) do
          L := L + 1;
          if D[I, L] >= J then
            D[I, L] := J;
          end;
        if L > MaxL then MaxL := L;
      end;
    end;
  LC := MaxL;
  J := LA;
  for I := LC downto 1 do
    begin
      while D[J - 1, I] = D[J, I] do
        J := J - 1;
        C[I - 1] := A[J - 1];
        J := J - 1;
      end;
      ...
    end.

```

Již zbývá jen odhadnout složitost algoritmu. Časově nejnáročnější byl vlastní výpočet hodnot v poli, který se skládá ze dvou hlavních cyklů o délce $|A|$ a $|B|$, což jsou délky posloupností A a B . Vnořený cyklus `while` proběhne celkem maximálně $|A|$ -krát a časovou složitost nám nezhorší. Můžeme tedy říct, že časová složitost je $\mathcal{O}(|A| \cdot |B|)$. Posloupnosti jsme si prohodili tak, aby první byla ta kratší, protože pak je maximální délka společné podposloupnosti i počet kroků algoritmu roven délce kratší posloupnosti a tedy i velikost pole s daty je kvadrát této délky. Paměťovou složitost odhadneme $\mathcal{O}(N^2 + M)$, kde N je délka kratší posloupnosti a M té delší.

Cvičení

- Proč jsme si z více posloupností zapamatovali zrovna tu, která v B končí nejlevějším možným prvkem?

Martin Mareš a Petr Škoda

Úloha 22-1-3: Sazba

Na vstupu dostanete text a číslo N . Vaším úkolem je zarovnat ho do bloku tak, aby byl co nejhezčí. Protože krása je věc názoru, zadefinujeme si pro naše účely vhodné objektivní měřítko: pro každou posloupnost mezer délky k (oddělující slova) vezmeme číslo $(k - 1)^2$ a tato čísla sečteme přes všechny posloupnosti mezer ve vysázeném textu. No a sazba je nejhezčí, pokud je tento součet nejmenší.

Slova nelze dělit mezi řádky a máte zaručeno, že se v textu neobjeví slovo delší než N . Na výstup od vás nechceme vypisovat vytvořené zarovnání, ale pouze minimální výše popsaný součet pro daný text, tedy číslo.

Například pro text „This is the example you are actually considering.“ a $N = 28$ má program vypsát 12, protože optimální zarovnání je

```
This is the example you
are actually considering.
```

a ohodnocení $1 + 1 + 1 + 4 + 1 + 4 = 12$.

Úloha 23-2-1: Balíčky balíčků

Chcete poslat poštou petici za lehčí úlohy v KSP organizátorům a co nevidíte. Výhodné nabídky balíčků! Můžete poslat jeden o váze N kg, dva o váze $N - 1$ kg, tři o váze $N - 2$ kg, ..., N o váze 1 kg, kde N závisí na ročním období, denní hodině a sjízdnosti silnic. To vás nemusí trápit, N dostane váš program na vstupu.

Dále dostanete váhu H petice v celých kilogramech. Vaším úkolem bude vymyslet, které nabídky balíčků je třeba vybrat, aby se do nich dohromady vešlo H kg petice, ale zároveň aby jejich kapacita byla co nejlíže tomuto H .

Je třeba zdůraznit, že „3 balíčky, každý o váze $N - 2$ kg“ je jedna nabídka, kterou jako celek buď přijmete, nebo nepřijmete. Chcete-li poslat $3N - 6$ kg, je to ideální volba.

Chcete-li poslat $N - 2$ kg a N není úplně malé (třeba $N = 100$), je lepší zvolit nabídku „1 balíček o váze N kg“, přestože dva kilogramy nevyužijete. Stejně dobré řešení by pak bylo vybrat „ N balíčků o váze 1 kg“ a nám je jedno, které z takových dvou stejně dobrých řešení vypíšete.

Chcete-li poslat 100 kg a $N = 12$, můžete vybrat třeba kombinaci

$$3 \cdot (N - 2) + 3 \cdot (N - 2) + 5 \cdot (N - 4) = 3 \cdot 10 + 3 \cdot 10 + 5 \cdot 8 = 100$$

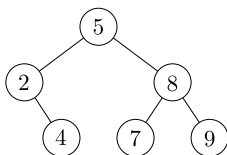
Vyhledávací stromy

Hledání půlením intervalu, které jsme si představili v třetí kuchařce, je velmi rychlé, pokud máme možnost si data předem seřadit. Jakmile ale potřebujeme za běhu programu přidávat a odebírat záznamy, se zlou se potážeme. Buďto budeme mít záznamy uložené v poli a pak nezbyvá než při zatřídování 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:

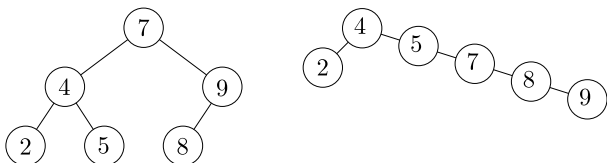
Vyhledávací stromy

Představme si, jakými všemi možnými cestami se může v našem poli binární vyhledávání ubírat. Na začátku porovnáme 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 chvíli uvidíme, i mnohých dalších operací.

Definice

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

Binární vyhledávací strom (podomácku BVS) je buďto 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:

```
type pvrchol = ^vrchol;
   vrchol = record
       l, r : pvrchol; { levý a pravý syn }
       x   : integer; { hodnota }
   end;
```

Pokud některý ze synů neexistuje, zapíšeme do příslušné položky hodnotu *nil*.

Find

V řeči BVS můžeme přeformulovat náš vyhledávací algoritmus takto:

```
function TreeFind(v :pvrchol; x: integer): pvrchol;
{ Dostane kořen stromu a hodnotu. Vrátil vrchol,
  kde se hodnota nachází, nebo nil, není-li. }
begin
  while (v<>nil) and (v^.x<>x) do begin
    if x<v^.x then
      v := v^.l
    else
      v := v^.r
    end;
  TreeFind := v;
end;
```

Funkce `TreeFind` 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.

Insert

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 *nil*. 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, tentokrát si ukážeme rekurzivní zacházení se stromy:

```
function TreeIns(v: pvrchol; x: integer): pvrchol;
{ Dostane kořen stromu a hodnotu ke vložení, vrátí nový kořen. }
begin
  if v=nil then begin
    { prázdný strom => založíme nový kořen }
    new(v);
    v^.l := nil;
    v^.r := nil;
    v^.x := x;
  end else if x<v^.x then { vkládáme vlevo }
    v^.l := TreeIns(v^.l, x)
  else if x>v^.x then    { vkládáme vpravo }
    v^.r := TreeIns(v^.r, x);
  TreeIns := v;
end;
```

Delete

Mazání bude o kousíček pracnější, musíme totiž rozlišit tři případy: Pokud je mazaný vrchol list, stačí ho vyměnit za *nil*. 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:

```
function TreeDel(v: pvrchol; x: integer): pvrchol;
{ Parametry stejně jako TreeIns }
var w:pvrchol;
begin
  TreeDel := v;
  if v=nil then exit          { prázdný strom }
  else if x<v^.x then
    v^.l := TreeDel(v^.l, x) { ještě hledáme x }
  else if x>v^.x then
    v^.r := TreeDel(v^.r, x)
  else begin                  { našli jsme }

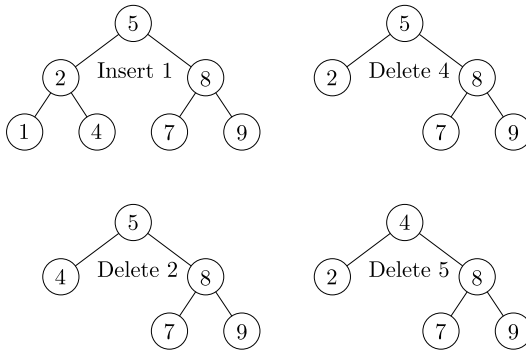
```

```

if (v^.l=nil) and (v^.r=nil) then begin
  TreeDel := nil;           { mažeme list }
  dispose(v);
end else if v^.l=nil then begin
  TreeDel := v^.r;         { jen pravý syn }
  dispose(v);
end else if v^.r=nil then begin
  TreeDel := v^.l;         { jen levý }
  dispose(v);
end else begin             { má oba syny }
  w := v^.l;               { hledáme max(L) }
  while w^.r<>nil do w := w^.r;
  v^.x := w^.x;            { prohazujeme }
  { a mažeme původní max(L) }
  v^.l := TreeDel(v^.l, w^.x);
end;
end;
end;

```

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:

```
procedure TreeShow(v: pvrchol);
begin
  if v=nil then exit; { není co dělat }
  TreeShow(v^.l);
  writeln(v^.x);
  TreeShow(v^.r);
end;
```

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 byl výjimka, leč všechny prvky rychleji než lineárně s N opravdu nevyvíšeme.)

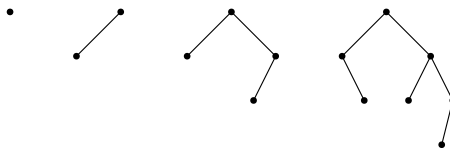
Jenže jak jsme viděli, neopatrným insertováním a deletováním prvků mohou snadno vznikat všelijaké degenerované stromy, které mají lineární hloubku. Abychom tomu zabránili, musíme stromy *vyvažovat*. To znamená definovat si nějaké šikovní omezení na tvar stromu, aby hloubka byla vždy $\mathcal{O}(\log N)$. 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 (jak jsme již dokázali) 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).

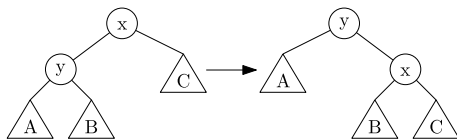
Spočítat, kolik přesně je A_d , není úplně snadné. Nám však postačí dokázat, že $A_d \geq 2^{d/2}$. To provedeme indukcí: Pro $d < 4$ to plyne z ručně spočítaných hodnot. Pro $d \geq 4$ je $A_d = 1 + A_{d-1} + A_{d-2} > 2^{(d-1)/2} + 2^{(d-2)/2} = 2^{d/2} \cdot (2^{-1/2} + 2^{-1}) > 2^{d/2}$ (součet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

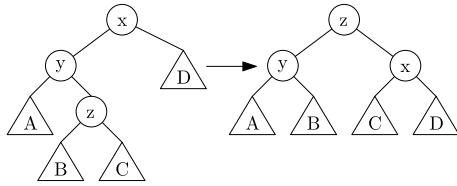
Rotací binárního stromu (respektive nějakého podstromu) nazveme jeho „překořnění“ za některého ze synů kořene. Místo formální definice ukažme raději obrázek:



Strom jsme překořnili 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. překořnění 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 překořnění podstromu za vnuka kořene připojeného „cickak“. 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 \ominus , \ominus a \oplus .

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

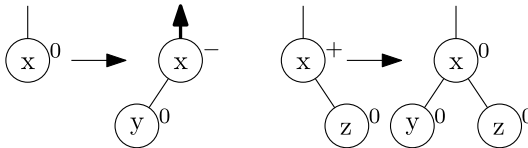
Často také budeme potřebovat nalézt otce nějakého vrcholu. To můžeme zařídit buďto tak, že si do záznamů popisujících vrcholy stromu přidáme ještě ukazatele na otce a budeme ho ve všech operacích poctivě aktualizovat, anebo využijeme toho, že jsme do daného vrcholu museli někudy přijít z kořene, a celou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

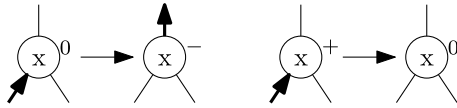
Když provedeme Insert tak, jak jsme ho popisovali u obecných vyhledávacích stromů, přibude nám ve stromu list. Pokud se tím AVL vyváženost neporušila, stačí pouze opravit znaménka na cestě z nového listu do kořene (všude jinde zůstala zachována). Pakliže porušila, musíme s tím něco provést, konkrétně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

Nejprve přidání listu samotné:

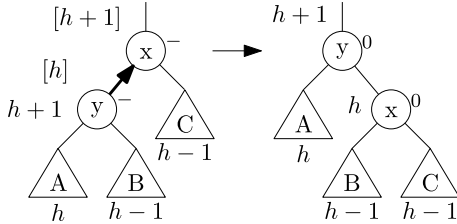


Pokud jsme přidali list (bez újmy na obecnosti levý, jinak vyřešíme zrcadlově) vrcholu se znaménkem \ominus , změniame znaménko na \ominus a pošleme o patro výš informaci o tom, že hloubka podstromu se zvýšila (to budeme značit šipkou). Přidali-li jsme list k \oplus , změni se na \ominus a hloubka podstromu 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 \ominus , ošetříme to stejně jako při přidání listu:

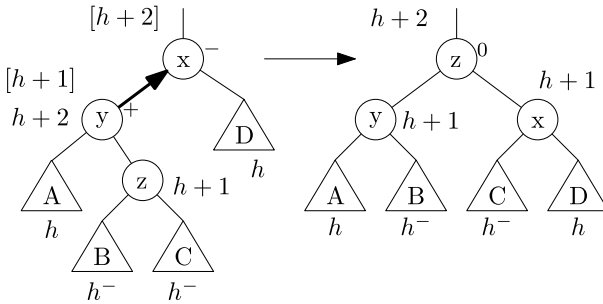


Pokud ale vrchol x má znaménko \ominus , nastanou potíže: levý podstrom má teď hloubku o 2 vyšší než pravý, takže musíme rotovat. Proto se podíváme o patro níž, jaké je znaménko vrcholu y pod šipkou, abychom věděli, jakou rotaci provést. Jedna možnost je tato (y je \ominus):



Tedy 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 \ominus a celková hloubka se nezmění, takže jsme hotovi.

Další možnost je y jako \oplus :

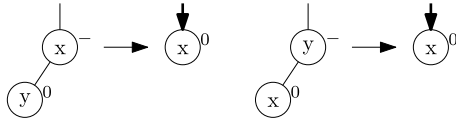


Tedy se podíváme ještě o hladinu níž a provedeme dvojrotaci. (Nemůže se nám stát, že by z neexistovalo, protože jinak by v y nebylo \oplus .) Hloubky opět najdete na obrázku. Jelikož z může mít libovolné znaménko, jsou hloubky podstromů B a C buďto h nebo $h - 1$, což značíme h^- . Podle toho pak vyjdou znaménka vrcholů x a y po rotaci. Každopádně vrchol z vždy obdrží \ominus a celková hloubka se nemění, takže končíme.

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

Vyvažování po Deletu

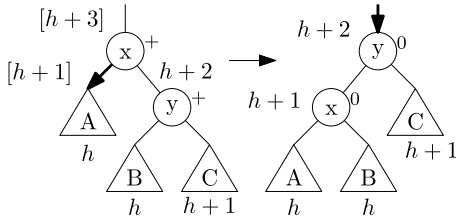
Vyvažování po Deletu je trochu obtížnější, ale také se dá popsat pár obrázky. Nejdříve opět rozebereme základní situace: odebíráme list (bez újmy na obecnosti (BÚNO) levý) nebo vnitřní vrchol stupně 2 (tehdy ale musí být jeho jediný syn listem, jinak by to nebyl AVL strom):



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:

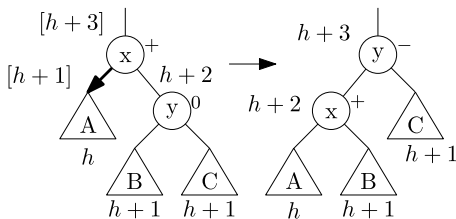


Problematické jsou tentokrát ty případy, kdy šipku dostane \oplus . Tehdy se musíme podívat na znaménko *opačného* syna a podle toho rotovat. První možnost je, že opačný syn má \oplus :



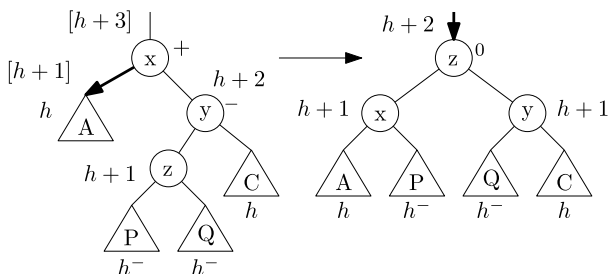
Tehdy provedeme rotaci vlevo, x i y získají nuly, ale celková hloubka stromu se snížila o hladinu, takže nezbyvá, než poslat šipku o patro výš.

Pokud by y byl \odot :



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

Poslední, nejkomplicovanější 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 přebarvováním na cestě do kořene a rotováním, 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 četností 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 $(0, 1)$. 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 vypsát 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 půjde 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 Adelsona-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 Mareš a Tomáš Valla

Úloha 16-4-5: Obchodníci s deštěm

Testujete generátor pseudonáhodných čísel a to takový, který bude generovat nelokální posloupnosti náhodných čísel. To jsou takové posloupnosti, jejichž členy jsou rozptýlené na celém používaném intervalu. Jinak řečeno, nejmenší vzdálenost mezi dvěma libovolnými prvky je pokud možno co největší.

Na vstupu dostanete N a K . Pak budete postupně načítat N různých náhodných čísel. *Hned* po načtení jednoho náhodného čísla (kromě prvního) vypíšete, jaký je nejmenší rozdíl mezi libovolnými různými dvěma z posledních K načtených náhodných čísel.

Příklad: Pro $N = 6$, $K = 3$ má vypadat vstup a výstup programu následovně:

<i>náhodné číslo</i>	<i>aktuální nejmenší rozdíl</i>
5	
7	2
4	1
15	3
6	2
20	5

Úloha 20-5-5: Roztržitý matematik

Jistý roztržitý matematik potřebuje udělat pořádek ve svých papírech, které má v řadě a jež jsou očíslované od 1 do N . Při své práci vždy nějaký vezme, podívá se na něj a poté ho zařadí na začátek řady (ostatní papíry se posunou). Na začátku matematikovy práce to šlo pěkně, neboť všechny papíry byly seřazeny podle čísel $(1, 2, \dots, N)$. Teď už jsou ale hodně přeházené a matematik nemůže najít ani svoji tramvajenku. Naštěstí si ještě pamatuje, kolikátý od začátku řady byl každý papír, se kterým pracoval. A v tomto okamžiku nastupujete do vzniklého chaosu vy, abyste matematika zachránili před jistou smrtí vyčerpáním.

Na vstupu jsou na prvním řádku dvě čísla N a K , kde N ($1 \leq N \leq 500\,000$) představuje počet papírů a K ($1 \leq K \leq 500\,000$) počet operací, které matematik udělal. Na druhém řádku je posloupnost K čísel, kde každé číslo x_i představuje i -tou operaci, při které matematik vzal x_i -tý papír od začátku řady a posunul ho na první místo. Před započítáním všech operací byly papíry seřazeny vzestupně od 1 do N .

Na prvním řádku výstupu bude N čísel představujících permutaci dokumentů po provedení všech K operací.

Příklad: Vstup:

8 3

5 1 4

Výstup:

3 5 1 2 4 6 7 8

Hešování

(V literatuře se také často setkáme s jinými přepisy tohoto anglicko-českého patvaru (*hashování*), či více či méně zdařilými pokusy se tomuto slovu zcela vyhnout a místo „heš“ používat například termín *asociativní pole*.)

Na *heš* se můžeme dívat jako na pole, které ale neindexujeme po sobě následujícími přirozenými čísly, ale hodnotami nějakého jiného typu (řetězci, velkými čísly, apod.). Hodnotě, kterou heš indexujeme, budeme říkat *klíč*. K čemu nám takové pole může být dobré?

- Aplikace typu slovník – máme zadán seznam slov a jejich významů a chceme k zadanému slovu rychle najít jeho význam. Vytvoříme si heš, kde klíče budou slova a hodnoty jim přiřazené budou překlady.
- Rozpoznávání klíčových slov (například v překladačích programovacích jazyků) – klíče budou klíčová slova, hodnoty jim přiřazené v tomto příkladě moc význam nemají, stačí nám vědět, zda dané slovo v heši je.
- V nějaké malé části programu si u objektů, se kterými pracujeme, potřebujeme pamatovat nějakou informaci navíc a nechceme kvůli tomu do objektu přidávat nové datové položky (třeba proto, aby nám zbytečně nezabíraly paměť v ostatních částech programu). Klíčem heše budou příslušné objekty.
- Potřebujeme najít v seznamu objekty, které jsou „stejně“ podle nějakého kritéria (například v seznamu osob ty, co se stejně jmenují). Klíčem heše je jméno. Postupně procházíme seznam a pro každou položku zjišťujeme, zda už je v heši uložena nějaká osoba se stejným jménem. Pokud není, aktuální položku přidáme do heše.

Potřebovali bychom tedy umět do heše přidávat nové hodnoty, najít hodnotu pro zadaný klíč a případně také umět z heše nějakou hodnotu smazat.

Samozřejmě používat jako klíč libovolný typ, o kterém nic nevíme (speciálně ani to, co znamená, že dva objekty toho typu jsou stejné), dost dobře nejde. Proto potřebujeme ještě *hešovací funkci* – funkci, která objektu přiřadí nějaké malé přirozené číslo $0 \leq x < K$, kde K je velikost heše (ta by měla odpovídat počtu objektů N , které v ní chceme uchovávat; v praxi bývá rozumné udělat si heš o velikosti zhruba $K = 2N$). Dále popsaný postup funguje pro libovolnou takovou funkci, nicméně aby také fungoval rychle, je potřeba, aby hešovací funkce byla dobře zvolena. K tomu, co to znamená, si něco řekneme níže, prozatím nám bude stačit představa, že tato funkce by měla rozdělovat klíče zhruba rovnoměrně, tedy že pravděpodobnost, že dvěma klíčům přiřadí stejnou hodnotu, by měla být zhruba $1/K$.

Ideální případ by nastal, kdyby se nám podařilo nalézt funkci, která by každým dvěma klíčům přiřazovala různou hodnotu (i to se může podařit, pokud množinu klíčů, které v heši budou, známe dopředu – viz třeba příklad s rozpoznáváním klíčových slov v překladačích). Pak nám stačí použít jednoduché pole velikosti K , jehož prvky budou obsahovat jednak hodnotu klíče, jednak jemu přiřazená data:


```

struct položka_heše {
    int obsazeno;
    typ_klíče klíč;
    typ_hodnoty hodnota;
} heš[K];

```

A operace naprogramujeme zřejmým způsobem:

```

void přidej (typ_klíče klíč, typ_hodnoty hodnota) {
    unsigned index = hešovací_funkce (klíč);
    // Kolize nejsou, čili heš[index].obsazeno=0.
    heš[index].obsazeno = 1;
    heš[index].klíč = klíč;
    heš[index].hodnota = hodnota;
}

int najdi (typ_klíče klíč, typ_hodnoty *hodnota) {
    unsigned index = hešovací_funkce (klíč);
    // Nic tu není nebo je tu něco jiného.
    if (!heš[index].obsazeno || !stejný(klíč, heš[index].hodnota))
        return 0;
    // Našel jsem.
    *hodnota = heš[index].hodnota;
    return 1;
}

```

Normálně samozřejmě takové štěstí mít nebudeme a vyskytnou se klíče, jimž hešovací funkce přiřadí stejnou hodnotu (říká se, že nastala *kolize*). Co potom?

Jedno z řešení je založit si pro každou hodnotu hešovací funkce seznam, do kterého si uložíme všechny prvky s touto hodnotou. Funkce pro vkládání pak bude v případě kolize přidávat do seznamu, vyhledávací funkce si vždy spočítá hodnotu hešovací funkce a projde celý seznam pro tuto hodnotu. Tomu se říká *hešování se separovanými řetězci*.

Jiná možnost je v případě kolize uložit kolidující hodnotu na první následující volné místo v poli (cyklicky, tj. dojdeme-li ke konci pole, pokračujeme na začátku). Samozřejmě pak musíme i příslušně upravit hledání – snadno si rozmyslíme, že musíme projít všechny položky od pozice, kterou nám poradí hešovací funkce, až po první nepoužitou položku (protože seznamy hodnot odpovídající různým hodnotám hešovací funkce se nám mohou spojit). Tento přístup se obvykle nazývá *hešování se srůstajícími řetězci*. Implementace pak vypadá takto:

```

void přidej (typ_klíče klíč, typ_hodnoty hodnota) {
    unsigned index = hešovací_funkce (klíč);
    while (heš[index].obsazeno) {
        index++;
        if (index == K) index = 0;
    }
}

```

```

heš[index].obsazeno = 1;
heš[index].klíč = klíč;
heš[index].hodnota = hodnota;
}

int najdi (typ_klíče klíč, typ_hodnoty *hodnota) {
    unsigned index = hešovací_funkce (klíč);
    while (heš[index].obsazeno) {
        if (stejný (klíč, heš[index].klíč)) {
            *hodnota = heš[index].hodnota;
            return 1;
        }
        // Něco tu je, ale ne to, co hledám.
        index++;
        if (index == K)
            index = 0;
    }
    // Nic tu není.
    return 0;
}

```

Jaká je časová složitost tohoto postupu? V nejhorším případě bude mít všech N objektů stejnou hodnotu hešovací funkce. Hledání může v nejhorším přeskakovat postupně všechny, čili složitost v nejhorším případě může být až $\mathcal{O}(NT + H)$, kde T je čas pro porovnání dvou klíčů a H je čas na spočtení hešovací funkce. Laicky řečeno, pro nalezení jednoho prvku budeme muset projít celou heš (v lineárním čase).

Nicméně tohle se nám obvykle nestane – pokud velikost pole bude dost velká (alespoň dvojnásobek prvků heše) a zvolili jsme dobrou hešovací funkci, pak v průměrném případě bude potřeba udělat pouze konstantně mnoho porovnání, tj. časová složitost hledání i přidávání bude jen $\mathcal{O}(T + H)$. A budeme-li schopni prvky hešovat i porovnávat v konstantním čase (což například pro čísla není problém), získáme konstantní časovou složitost obou operací.

Mazání prvků může působit menší problémy (rozmyslete si, proč nelze prostě nastavit u mazaného prvku „obsazeno“ na 0). Pokud to potřebujeme dělat, buď musíme použít separované řetězce (což se může hodit i z jiných důvodů, ale je o trochu pracnější), nebo použijeme následující figl: když budeme nějaký prvek mazat, najdeme ho a označíme jako smazaný. Nicméně při hledání nějakého jiného prvku se nemůžeme zastavit na tomto smazaném prvku, ale musíme hledat i za ním. Ovšem pokud nějaký prvek přidáváme, můžeme jím smazaný prvek přepsat.

A jakou hešovací funkci tedy použít? To je tak trochu magie a dobré hešovací funkce mají mimo jiné hlubokou souvislost s kryptografií a s generátory pseudonáhodných čísel. Obvykle se dělá to, že se hešovaný objekt rozloží na posloupnost čísel (třeba ASCII kódů písmen v řetězci), tato čísla se nějakou operací „slejí“ dohromady a výsledek se vezme modulo K . Operace na slévání se používají různé, od jednoduchého xoru až třeba po komplikované vzorce typu

```

#define mix(a,b,c) {
    a-=b; a-=c; a^=(c>>13);
    b-=c; b-=a; b^=(a<< 8);
    c-=a; c-=b; c^=((b&0xffffffff)>>13);
    a-=b; a-=c; a^=((c&0xffffffff)>>12);
    b-=c; b-=a; b =(b ^ (a<<16)) & 0xffffffff;
    c-=a; c-=b; c =(c ^ (b>> 5)) & 0xffffffff;
    a-=b; a-=c; a =(a ^ (c>> 3)) & 0xffffffff;
    b-=c; b-=a; b =(b ^ (a<<10)) & 0xffffffff;
    c-=a; c-=b; c =(c ^ (b>>15)) & 0xffffffff;
}

```

My se ale spokojíme s málem a ukážeme si jednoduchý způsob, jak hešovat čísla a řetězce. Pro čísla stačí zvolit za velikost tabulky vhodné prvočíslo a klíč vymodulit tímto prvočíslem. (S hledáním prvočísel si samozřejmě nemusíme dělat starosti, v praxi dobře poslouží tabulka několika prvočísel přímo uvedená v programu.)

Rozumná funkce pro hešování řetězců je třeba:

```

unsigned hash_string (unsigned char *str)
{
    unsigned r = 0;
    unsigned char c;

    while ((c = *str++) != 0)
        r = r * 67 + c - 113;

    return r;
}

```

Zde můžeme použít vcelku libovolnou velikost tabulky, která nebude dělitelná čísly 67 a 113. Šikovné je vybrat si například mocninu dvojky (což v příštím odstavci oceníme), ta bude s prvočísly 67 a 113 zaručeně nesoudělná. Jen si musíme dávat pozor, abychom nepoužili tak velkou hešovací tabulku, že by 67 umocněno na obvyklou délku řetězce bylo menší než velikost tabulky (čili by hešovací funkce časteji volila začátek heše než konec). Tehdy ale stačí místo našich čísel použít jiná, větší prvočísla.

A co když nestačí pevná velikost heše? Použijeme „nafukovací“ heš. Na začátku si zvolíme nějakou pevnou velikost, sledujeme počet vložených prvků a když se jich zaplní víc než polovina (nebo třeba třetina; menší číslo znamená méně kolizí a tedy větší rychlost, ale také větší paměťové plýtvání), vytvoříme nový heš dvojnásobné velikosti (případně zaokrouhlené na vyšší prvočíslo, pokud to naše hešovací funkce vyžaduje) a starý heš do něj prvek po prvků vložíme.

To na první pohled vypadá velice neefektivně, ale protože se po každém nafouknutí heš zvětší na dvojnásobek, musí mezi přehešováním na N prvků a na $2N$ přibýt alespoň N prvků, čili průměrně strávíme přehešováním konstantní čas na každý vložený prvek.

Pokud navíc používáme mazání prvků popsané výše (u prvku si pamatujeme, že je smazaný, ale stále zabírá místo v heši), nemůžeme při mazání takového prvku snížit počet prvků v heši, ale na druhou stranu při nafukování můžeme takové prvky opravdu smazat (a konečně je odečíst z počtu obsazených prvků).

Poznámky

- S hešováním se separovanými řetězci se zachází podobně, nafukování také funguje a navíc je snadno vidět, že po vložení N náhodných prvků bude v každé přihrádce (přihrádky odpovídají hodnotám hešovací funkce) průměrně N/K prvků, čili pro K velké řádově jako N konstantně mnoho. Pro srůstající řetězce to pravda být nemusí (protože jakmile jednou vznikne dlouhý řetězec, nově vložené prvky mají sklony „nalepovat se“ za něj), ale platí, že bude-li heš naplněna nejvýše na polovinu, průměrná délka kolizního řetízku bude omezená nějakou konstantou nezávislou na počtu prvků a velikosti heše. Důkaz si ovšem raději odpustíme, není úplně snadný.
- Bystrý čtenář si jistě všiml, že v případě prvočíselných velikostí heše jsme v důkazu časové složitosti nafukování trochu podváděli – z heše velikosti N přeci přehesováváme do heše velikosti větší než $2N$. Zachrání nás ale věta z teorie čísel, obvykle zvaná Bertrandův postulát, která říká, že mezi čísly t a $2t$ se vždy nachází alespoň jedno prvočíslo. Takže nová heš bude maximálně $4\times$ větší, a tedy počet přehesování na jedno vložení bude nadále omezen konstantou.

Zdeněk Dvořák

Úloha 17-2-1: Prasátko programátorem

Programy pašíka Kvašíka bývají úděsně pomalé a potřebují zrychlit. Lze si je představit jako posloupnosti přiřazení do proměnných, což jsou řetězce znaků složené z malých písmen, velkých písmen a podtržíték. Na pravé straně přiřazení může být buď proměnná nebo operace „+“ nebo „*“ aplikovaná na dvě proměnné. Tyto operace jsou komutativní, neboli $a + b = b + a$ a $a * b = b * a$.

Vášim úkolem je napsat program, který dostane Kvašíkův program skládající se z N přiřazení a má říci, jak moc ho lze zrychlit, čili říci, kolik nejméně operací „+“ a „*“ stačí k tomu, aby nový program přiřadil do všech proměnných stejnou hodnotu jako Kvašíkův. Formálně pro každých i prvních řádků Kvašíkova programu musí v novém programu existovat místo, kdy jsou hodnoty všech proměnných z Kvašíkova programu v obou programech shodné. Můžete využívat toho, že operace „+“ a „*“ jsou komutativní, ale jejich asociativita a distributivita se neberou v úvahu, čili $a + (b + c) \neq (a + b) + c$ a také $(a + b) * c \neq a * c + b * c$.

Příklad: Vlevo je Kvašíkův program, vpravo náš.

$a = b + c;$ $d = a + b;$ $e = c + b;$ $f = a * e;$ $a = d;$ $g = e * e;$	$t = b + c;$ $a = t;$ $d = a + b;$ $e = t;$ $s = a * e;$ $f = s;$ $a = d;$ $g = s;$
--	--

Zatímco Kvašíkův program potřeboval operací pět, náš si vystačí se třemi, takže výstup programu by měl být „3“.

Úloha 19-4-3: Naskakování na vlak

Naskakování na vlak není věc jednoduchá, při níž se může hodit vědět, jestli se podobný vagón (resp. posloupnost vagónů) vyskytuje ve vlaku víckrát. A tady je příležitost pro vás, abyste se zkoumáním vlaku pomohli.

Vlak si představte jako řetězec délky N , kde každé písmeno představuje jeden vagón (např. U je uhelný vagón, P je poštovní vůz atp.). Dále máte dáno číslo k ($k \leq N$) a máte zjistit, kolik navzájem různých podřetězců délky k se v řetězci (tedy ve vlaku) vyskytuje. Zároveň tyto podřetězce a počty jejich výskytů vypište. Pozor, vlak už se blíží, takže byste to měli spočítat pekelně rychle.

Příklad: Pro řetězec (vlak) UPDUPDUDUP a $k = 3$ jsou nalezené podřetězce

UPD	2×
PDU	2×
DUP	2×
DUD	1×
UDU	1×

Řetězce a vyhledávání v textu

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

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

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

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

Jak řetězce chápat

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

Jak jsme si řekli na začátku, řetězec bude posloupnost nějakých symbolů, kterým říkáme *znaky*. Tyto znaky jsou z nějaké množiny, které říkáme *abeceda*. Abeceda může být jen 01 pro čísla v binárním zápisu, klasické A-Za-z pro malou anglickou abecedu anebo plný rozsah univerzální znakové sady Unicode, která má až 2^{31} znaků. Nezapomínejme, že i mezery a interpunkce jsou znaky!

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

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

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

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

Jelikož jsme řetězce definovali jako posloupnosti, nesmíme zapomínat ani na *prázdný řetězec* ε . A když už máme řetězec, určitě máme i *podřetězec* – souvislou podposloupnost znaků jiného řetězce. Například **ret**, ε i **kabaret** jsou podřetězce slova (řetězce) **kabaret**.

Často nás budou zajímat dva zvláštní druhy podřetězců. Pokud ze slova usekneme nějaký souvislý úsek na konci, vznikne podřetězec, které říkáme *prefix* (česky předpona), a pokud usekneme nějaký souvislý úsek ze začátku, dostaneme *suffix* neboli příponu. **ret** je suffix slova **kabaret**, **kaba** je zase jeho prefixem.

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

Pro některá použití řetězců je důležité, abychom je mohli porovnávat – když máme řetězce A a B , tak rozhodnout, který je menší, a který je větší. Jaké přesně toto uspořádání bude, závisí na naší aplikaci, ale mnohdy se používá tzv. *lexikografické uspořádání*. Pro lexikografické uspořádání potřebujeme nejprve zadané (lineární) uspořádání na znacích (kromě binárního $0 < 1$ se často používá „telefonní“ $A = a < B = b < \dots < Z = z$, které je ovšem lineární až na velikost znaků).

Když máme zadané uspořádání na znacích, na všechny řetězce jej rozšíříme následovně: nejkratší je prázdný řetězec a ostatní řetězce třídíme podle znaků od začátku do konce. Zvláštnost je v tom, že řetězec je větší než jeho každá vlastní předpona (neboli *prefix*). Řetězec **a** tedy bude menší než **auto**, které samo bude menší než **autobus**.

Adresář pomoci trie

Typický problém v oblasti textu je, že máme seznam nějakých řetězců (často třeba jmenný adresář), můžeme si jej nějak předzpracovat, a pak bychom rádi efektivně odpovídali na otázku: „Je řetězec S obsažen v adresáři?“ Můžeme také po předzpracování chtít přidávat nové položky i odebírat staré.

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

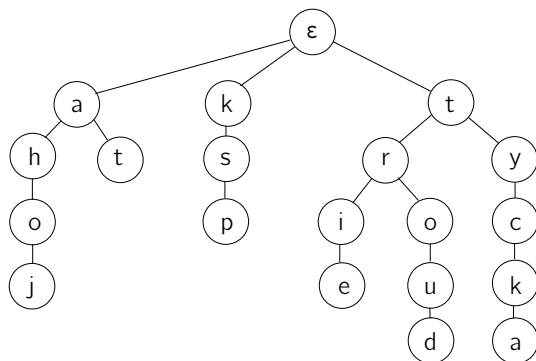
Ukážeme si jiné řešení, které je také asymptoticky rychlé a není ani příliš náročné na paměť. Využívá stromové struktury a říká se mu *trie* (vyslovujeme česky „tryje“ a anglicky jako část slova „retrieval“, z něhož slovo *trie* vzniklo). V češtině se občas používá také označení „písmenkový strom“.

Trie bude zakořeněný strom, budeme jej stavět pro nějaký adresář A . Kořen bude odpovídat prázdnému slovu ε . Každá hrana, která z něj povede, odpovídá jednomu ze znaků, kterým slovo z adresáře A začíná, a to bez opakování (tedy jsou-li v A čtyři slova začínající na **a**, hranu vedeme jen jednu).

Na koncích těchto hran z kořene nám vznikly vrcholy, které odpovídají všem jednoznakovým prefixům slov z A , a už je celkem jasné, jak struktura dále pokračuje –

z každého vrcholu odpovídajícímu prefixu P vede hrana se znakem c právě tehdy, když slovo $P + c$ (za P přilepíme znak c) je také prefixem některého slova z A .

Obrázek vydá za tisíc definic, zde je postavená trie pro slova *ahoj*, *at*, *ksp*, *trie*, *troud*, *tyc*, *tycka*:



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

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

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

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

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

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

Pokud tedy pracujeme s velkou abecedou, může se nám vyplatit oželeť konstantní rychlost dotazu a použít v každém vrcholu vlastní binární vyhledávací strom pro znaky, kterými aktuální prefix může pokračovat. To zmírní časovou složitost konstrukce na $\mathcal{O}((\log |\Sigma|) \cdot D)$ a zhorší časovou složitost dotazu na slovo délky l na $\mathcal{O}(l \cdot \log |\Sigma|)$.

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

Poznámky

- Chcete-li algoritmus konstrukce trie vidět napsaný v Pascalu, podívejte se do knihy [Töpfer].
- Triím se také říká *prefixové stromy*, což popisuje, že každý vrchol odpovídá prefixu některého slova v adresáři. (Slovo *prefixové* je však v matematice hodně nadužívané (prefixová notace, prefixové kódy), a tak to může vést ke zmatení).
- Kdybychom chtěli, mohli bychom pomocí trie vyhledávat v českém textu v lineárním čase. Můžeme přeci postavit adresář ze všech slov v daném textu, a pak procházet tu trii. Má to ale pár háčků: jednak je často hledaný řetězec krátký, ale text se nevejde do paměti. Druhak, pokud bychom použili jako oddělovač mezery, bychom mohli hledat jen jednotlivá slova, a nikoli jejich konce nebo delší kusy věty.
- Asi se po poslední poznámce ptáte – existuje nějaká modifikace trie, která umí hledat libovolnou část textu? Ano, jmenuje se *suffixový strom* a jdou s ní dělat spousty krásných kousků. Říká se, že každou řetězcovou úlohu lze řešit v lineárním čase pomocí suffixových stromů. Více se o nich dočtete třeba v [GrafAlg].

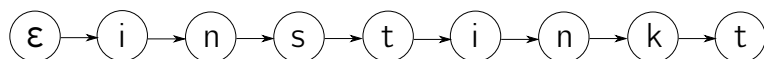
Cvičení

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

Vyhledávání v textu

Začátek situace je asi zřejmý – máme na vstupu zadán dlouhý text a krátké slovo. Chceme si slovo zpracovat, načež projdeme co nejrychleji text a zahlásíme jeden nebo všechny jeho výskyty. Často se hovoří o „hledání jehly v kupce sena“, a tedy se textu přezdívá *seno* a hledanému slovu *jehla*. Délku jehly označíme proměnnou j a délku textu n .

Představme si nejdříve hledané slovo jako spojový seznam, třeba slovo *instinkt*:



Mohli bychom text začít procházet písmenko po písmenko a kontrolovat, zda se text shoduje s naším slovem/spojovým seznamem. Pokud by si znaky odpovídaly,

skočíme na další písmenko z textu a i na další písmenko v seznamu. Co když se ale neshodují? Pak nemůžeme jen skočit na další písmeno textu – co kdybychom v textu narazili na slovo **instinstinkt**?

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

Sice je předchozí popis skutečně v nejhorším případě složitý $\mathcal{O}(nj)$, avšak stačí malá úprava a složitost přejde na lineární $\mathcal{O}(n + j)$. Ve skutečnosti nebylo vrácení se to, co algoritmus zpomalovalo, za špatnou složitost mohl fakt, že jsme se vraceli *příliš zpátky*.

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

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

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

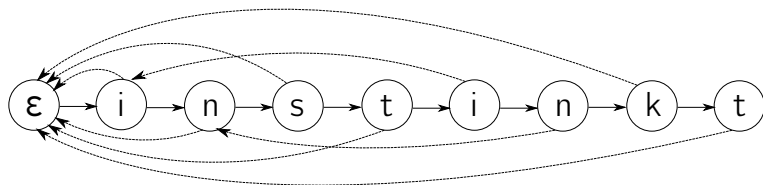
Zatím jsme ale přesně nepopsali, na které políčko přesně bude zpětná funkce ukazovat. Nechtě chceme určit zpětné políčko pro druhé **n** ve slově **instinkt**. Pracujeme teď s prefixem **instin**. Selsky řečeno, chceme najít „konec slova **instin** takový, že je stejný, jako začátek slova **instin**“.

Abychom náš požadavek upřesnili, zamysleme se nad zpětným políčkem pro jiné slovo. Co kdyby jehlou bylo slovo **abababc** a my určovali zpětné políčko pro **ababab**? Kdybychom ukázali na první písmenko **b**, nebylo by to správně, protože pak bychom pro text **ababababc** nezahlásili výskyt jehly, což je jasná chyba. Musíme se vrátit už na **abab**!

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

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

Pro slovo **instinkt** vypadá spojový seznam obohacený o zpětnou funkci (zakreslenou pomocí ukazatelů) takto:



Nyní vyvstávají dvě otázky: Jakou to má celé časovou složitost? Jak spočítat zpětnou funkci? Poperme se nejdříve s tou první. Pro každý znak vstupního textu mohou nastat dva případy: Buď znak rozšiřuje aktuální prefix, nebo musíme použít zpětnou funkci. První případ má jasně konstantní složitost, druhý je horší, neboť zpětná funkce může být pro jeden znak volána až j -krát.

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

Konstrukci zpětné funkce provedeme malým trikem. Všimněme si, že $F[i]$ je přesně číslo stavu, do něž se dostaneme při spuštění našeho vyhledávacího algoritmu na řetězec, který tvoří prvních i znaků jehly bez prvního písmenka.

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

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

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

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

```

var
  Slovo: array[1..P] of char;   { jehla }
  Text: array[1..N] of char;   { seno }
  
```

```

F: array[1..MaxS] of integer; { zpětná fce }
function Krok(I: integer; C: char): integer;
begin
  if (I < P) and (Slovo[I+1] = C) then
    Krok := I + 1
  else if I > 0 then
    Krok := Krok(F[I], C)
  else
    Krok := 0;
end;

var
  I, R: integer; { pomocné proměnné }
begin
  { konstrukce zpětné funkce }
  F[1] := 0;
  for I := 2 to P do
    F[I] := Krok(F[I-1], Slovo[I]);
  { procházení textu }
  R := 0;
  for I := 1 to N do begin
    R := Krok(R, Text[I]);
    if R = P then
      writeln(I);
  end;
end.

```

Poznámky

- Pro anglický nebo český text je použití takto sofistikovaného algoritmu skoro škoda, protože v obou jazycích se stává jen málokdy, že bychom měli několik slov spojených dohromady. Prakticky bude stačit i na začátku zmíněný naivní algoritmus. Na soutěžích a olympiádách ale pište raději algoritmus KMP.
- Hešování lze použít i na vyhledávání řetězce v textu. Obzvláště vhodné jsou na to *rolling hash functions* („okénkové hešovací funkce“), které umí v konstantním čase přepočítat heš, ubereme-li nějaké písmeno na začátku a přidáme-li ho na konci.
- Co kdybychom neměli jen jednu jehlu/hledané slovo, ale celý jehelníček, čili seznam hledaných slov? I to lze řešit podobnou metodou, jako jsme řešili jedno slovo. Tento algoritmus se nazývá po tvůrcích *algoritmus Aho-Corasicková* a spočívá v tom, že jednoduchý spojový seznam nahradíme trií a do trie opět přidáme zpětné hrany. Není to tak těžké vymyslet, pokud rozumíte tomu základnímu algoritmu.
- Algoritmus KMP je často zmiňován v souvislosti s *konečnými automaty*, protože náš postup skákání po spojovém seznamu se zpětnými hranami je vlastně jen

přechod konečným automatem. KSP ve 23. sérii vytvořilo seriál o regulárních výrazech, který teorii konečných automatů popisuje.

Cvičení

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

Martin Böhm, Martin Mareš a Petr Škoda

Úloha 18-5-4: Detektív

Slavný detektiv Šerlok Houmles je na stopě vážného zločinu. A to doslova a do písmene. S lupou až u země právě prohlíží stopy, které by ho měly dovést k pachateli. Pomůžete detektivovi s jeho případem?

Stopy jsou uspořádány do řady. Navíc každou stopu lze označit nějakým písmenem, nebo jiným znakem a těchto „typů“ stop není mnoho (desítky až stovky). Dále má Šerlok k dispozici Knihu Stopování Pachatelů, ve které jsou popsány všechny podezřelé výskyty stop.

Na vstupu dostanete všechny podezřelé sekvence stop a dále řetězec stop, které detektiv sleduje. Tento řetězec je velice dlouhý a nevejde se do operační paměti. Pro jednoduchost předpokládejte, že existuje funkce `GetFootprint`, která vrací právě přečtenou stopu (např. jako znak) a procedura `RewindFootprint`, která vrátí detektiva na začátek stop. Váš program by měl zjistit ke každé sekvenci podezřelých stop, kolikrát se vyskytla během stopování.

Zároveň si uvědomte, že času je málo, a tak by váš program měl pracovat ideálně v čase $\mathcal{O}(N + P)$ (N je délka stopovaného řetězce a P je součet délek všech podezřelých stop), bez ohledu na počet výskytů podezřelých sekvencí, přestože jejich počet může být až $\mathcal{O}(Nk)$, kde k je počet podezřelých sekvencí. Detektiv také nemůže stále běhat sem a tam, takže váš program by měl funkci `RewindFootprint` volat co nejméně (ideálně vůbec).

Nicméně i řešení v čase $\mathcal{O}(Nk + P)$ je daleko hodnotnější než řešení se složitostí $\mathcal{O}(NP)$.

Příklad: Podezřelé sekvence stop jsou LPBBLP, BBBBO, OSSO.

Prohledávané stopy budtež OSSOSSOLPBBLPBBLPBBLPBBBBO.

Výstup programu by měl být

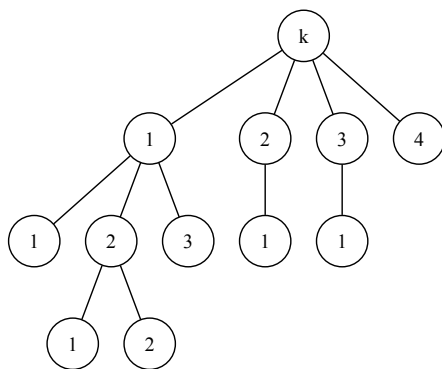
<i>podezřelý vzorek</i>	<i>počet jeho výskytů</i>
LPBBLP	2
BBBBO	1
OSSO	2

Úloha 22-4-4: Ořez stromu

V sadě jabloní se jako každé jaro ořezávají větve. Abychom si ověřili, že nám stromy nikdo neukradl a nevyměnil za nějaké atrapy, potřebujeme umět zjistit, jestli je-

den strom mohl vzniknout z druhého operací ořezávání, přičemž se nám pomíchaly informace a nevíme, který strom by měl vzniknout ořezáním z druhého.

V této úloze budeme za strom považovat souvislý graf bez kružnic s pevně daným kořenem. Navíc každý uzel má jasné uspořádání synů, takže podíváme-li se na některý vrchol, tak umíme vždy jasně říci, který syn je první, který je druhý, a tak dále.



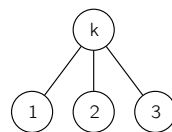
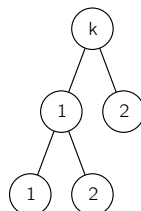
Jak probíhá takové ořezávání? Ze stromu se nejprve vybere vrchol (například první syn kořene z příkladu) a v tomto místě se ještě zvolí souvislý interval synů (např. druhý až třetí syn). Původně vybrané rozbočení se prohlásí za kořen, synové kořene budou jen ty vrcholy, které byly jeho syny ve vybraném intervalu, a uspořádání se zachová (druhý syn bude prvním synem nového kořene).

Spolu s tímto intervalem patří do ořezaného stromu také celé podstromy pod těmito syny. Vrcholy, které neležely v příslušném intervalu nebo byly jinde v původním stromě, v novém stromě prostě nebudou.

Na vstupu dostanete dva zakořeněné stromy a máte zjistit, jestli jeden mohl vzniknout ořezem druhého. Stromy mohou být zadány například takto: vrcholy si očíslovujeme od 1 do N , kořenem bude vrchol s číslem 1 a na vstupu dostaneme pole spojových seznamů. i -tý prvek pole je spojový seznam, který obsahuje číselná označení synů i -tého vrcholu, uspořádaná zleva doprava. V této reprezentaci můžeme zadat „osekaný strom“ z obrázku níže například takto:

- 1: 2 3
- 2: 4 5
- 3:
- 4:
- 5:

osekaný strom

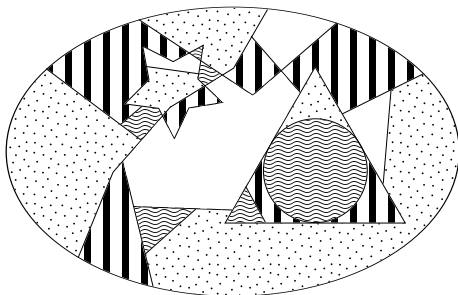


tento ale nemohl vzniknout

Rovinné grafy

Na úvod položíme otázku: „Stačí čtyři barvy na obarvení libovolné politické mapy, aby žádné dva sousedící státy neměly stejnou barvu?“ (Za sousedící státy se nepovažují ty, které sousedí jen v jediném bodě, ani v konečně mnoha.)

Na první pohled jednoduchá otázka, že? Matematici se s ní však trápili více jak století (od první formulace v roce 1852 do vyřešení v roce 1976) a nikdo nebyl schopen přijít s důkazem ani s protipříkladem, tedy mapou, na níž je potřeba pět barev. I třeba na tuto mapu jsou potřeba jen čtyři barvy:



Nebudeme dlouho tajit odpověď: čtyři barvy stačí. Důkaz se spoléhá na strojově probírání stovek případů (fragmentů rovinných grafů) a ve své době pro svou domnělou nematematickostí vzbudil velké pozdvižení. Dodnes se snaží mnoho vědců najít jednodušší důkaz, podobně jako u Velké fermatovy věty.

My si ukážeme, jak každou politickou mapu obarvit šesti barvami, a od toho přejdeme k pěti barvám. Nejdříve si však převedeme politické mapy na rovinné grafy a předvedeme si několik jejich užitečných vlastností.

Cvičení

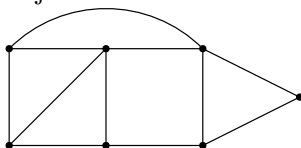
- U státu se v úvodní otázce tiše předpokládá, že je souvislý, tedy mezi každými dvěma místy v něm lze přejít bez přechodu do jiného státu. Rozmyslete si, jak by vypadala mapa s nespojitými státy, na niž je potřeba pět barev.

Rovinné grafy

Rovinný graf (někdy též nazývaný planární) je graf, který můžeme nakreslit do roviny bez křížení hran. To znamená, že vrcholům přiřadíme vhodné body a hrany nakreslíme jako křivky spojující příslušné body tak, že se žádné dvě křivky neprotínají mimo své krajní body.

Ne každý graf lze takto nakreslit – sami si rozmyslete, že například graf K_5 , což je 5 vrcholů spojených každý s každým, žádné rovinné nakreslení nemá. Na druhou stranu například každý strom určitě rovinný je.

Vezmeme si tedy nějaký graf a jeho rovinné nakreslení, například tento:



Hrany nakreslení dělí rovinu na několik oblastí, těm budeme říkat *stěny*. Náš graf má 6 stěn: jednu čtvercovou, čtyři „trojúhelníkové“ (tedy ohraničené třemi hranami, byť to nejsou vždy úsečky) a jednu 6-úhelníkovou (to je celý zbytek roviny okolo grafu, tzv. *vnější stěna*).

Například libovolné rovinné nakreslení stromu by mělo pouze jednu stěnu, a to tu vnější. Všimněte si, že pokud v grafu nejsou mosty ani artikulece, je každá stěna ohraničena nějakou kružnicí. (Pozor, to, jak vypadají stěny, závisí na konkrétním nakreslení do roviny!)

Barvení grafů

Jistě byste dokázali vymyslet převod politické mapy se státy na graf. Jen pro úplnost: samotné státy budou vrcholy a hrana povede mezi vrcholy právě tehdy, když odpovídající státy sousedí.

Podobně jako se barví politické mapy, lze barvit i grafy. Samozřejmě ne *doslova*, barvením se zpravila myslí přiřazování přirozených čísel jednotlivým vrcholům pod podmínkou, že sousední vrcholy nesmí mít stejnou barvu. Důležitou vlastností grafu je pak jeho barevnost, neboli nejmenší počet barev, kterými se dá obarvit. Úvodní problém tedy vlastně říká, že barevnost každého rovinného grafu je nejvýše čtyři.

V praxi má barvení grafů (nejen rovinných) velké využití: představte si například, že chováte spoustu psů, přičemž každý pes nesnáší několik jiných a nesmí s nimi být ve výběhu. Otázkou je, kolik nejméně výběhů je potřeba.

Cvičení

- Rozmyslete si, že graf vytvořený z politické mapy je skutečně rovinný, tzn. že tento graf lze zakreslit do roviny tak, že se nekříží jeho hrany.
- Představte si, že máte algoritmus, který obarví vstupní graf daným počtem barev, pokud to jde. Jak pomocí něj vyřešit sudoku?
- U grafů se studuje i barvení hran (hrany nesmí mít stejnou barvu, pokud sdílejí vrchol). Zkuste si rozmyslet, jak vrcholová i hranová barevnost souvisí s maximálním stupněm grafu (např. najít horní omezení v závislosti na maximálním stupni, u hranové i dolní).

Vlastnosti rovinných grafů

O rovinných grafech platí několik důležitých vět, které se často hodí při vytváření grafových algoritmů.

Je zřejmé, že každý strom je rovinný. Navíc pro každý strom platí, že má o jedna méně hran než vrcholů (tedy $e = v - 1$, kde v je počet vrcholů a e počet hran). Proč tomu tak je? Abychom tvrzení dokázali, použijeme *indukci* podle počtu vrcholů.

(Důkaz indukcí funguje tak, že ukážeme platnost tvrzení pro strom s jedním vrcholem a potom pro stromy s v vrcholy, pokud tvrzení platí pro všechny stromy s méně jak v vrcholy – tomu se říká *indukční předpoklad*).

Pro strom s jedním vrcholem formulka určitě platí. Strom s $v > 1$ vrcholy má jistě list, tak jej odtrhneme (poněkud vandalské, nicméně účinné), čímž získáme strom s menším počtem vrcholů, pro který podle indukčního předpokladu formulka platí, a opětovným přidáním listu platit nepřestane, protože k oběma stranám přičteme jedničku.

Vztah počtu vrcholů, hran a stěn

Počet stěn souvislého rovinného grafu je pevně určen počtem vrcholů a hran, aniž by záleželo na konkrétním nakreslení do roviny nebo dokonce na tom, mezi kterými vrcholy hrany vedou. Pro každý souvislý rovinný graf nakreslený do roviny totiž platí tzv. *Eulerova formule*: $v + f = e + 2$, kde v je počet vrcholů, e počet hran a f počet stěn.

Důkaz: Opět indukcí, tentokrát podle počtu hran. Každý souvislý graf má alespoň $v - 1$ hran a pokud jich má právě tolik, je to strom. (Kdyby ne, stačí se podívat na kostru grafu, což musí být strom a ty, jak už víme, mají právě tolik hran a náš graf měl hran více.) Jenže každé rovinné nakreslení stromu má právě jednu stěnu, takže Eulerova formule platí.

Pokud máme nakreslení grafu, který je souvislý a není to strom, znamená to, že obsahuje alespoň jednu kružnici. A každá hrana na kružnici jistě odděluje nějaké dvě stěny. Zvolme si tedy nějakou takovou hranu h a z grafu ji odeberme. Tím získáme graf s menším počtem hran (opět nakreslený do roviny), použijeme indukční předpoklad, Eulerova formule pro něj tedy již platí, a vrátíme hranu zpět. Levá strana rovnosti se tím zvětší o 1 (přidali jsme stěnu), pravá také (přidali jsme hranu), tedy rovnost stále platí.

Cvičení

- Dokažte, že Eulerova formule pro grafy s více komponentami je $v + f = e + k + 1$, kde k je počet komponent.

Omezení počtu hran

Intuicí snadno odhalíte, že velké rovinné grafy nemohou mít spoustu hran, protože by nešly nakreslit bez křížení. Hran je dokonce lineárně, protože platí následující nerovnost (pro grafy s alespoň třemi vrcholy): $e \leq 3v - 6$.

Nejprve jednoduchá aplikace: výše jsme zmínili, že K_5 (úplný graf na 5 vrcholech) není rovinný. Má totiž 10 hran, ale naše formulka mu dovoluje jen 9. Takto se dá o spoustě „hustých“ grafů dokázat, že nejsou rovinné, bohužel však tvrzení neplatí obráceně a existují grafy s $3v - 6$ vrcholy (nebo méně), které nejsou rovinné.

Jak tedy nerovnost dokázat? Zvolme si libovolné nakreslení grafu do roviny. Nejprve předpokládejme, že je to triangulace, čili že každá stěna je trojúhelník. V takovém grafu patří každá hrana k právě dvěma trojúhelníkovým stěnám, takže $e = f \cdot 3/2$,

čili $f = e \cdot 2/3$. Dosazením do Eulerovy formule získáme $v + (2/3)e = e + 2$, tedy $e = 3v - 6$.

Není-li náš graf triangulace, může to mít několik důvodů. Buďto není souvislý (pak ale stačí větu dokázat pro jednotlivé komponenty a nerovnosti sečíst), nebo je moc malý (má nejvýše dva vrcholy, ale tak malé grafy neuvažujeme) anebo obsahuje nějakou stěnu ohraničenou více než třemi hranami. Dovnitř takové stěny ovšem můžeme dokreslit další hrany a tím ji rozdělit na trojúhelníčky. Tím tedy dokážeme graf doplnit hranami na triangulaci, pro tu, jak už víme, platí dokonce rovnost, a když přidané hrany opět odebereme, snížíme pouze počet hran a uděláme tak z rovnosti nerovnost.

Cvičení

- Rovinné grafy bez trojúhelníkové stěny (tedy ty, co neobsahují K_3 jako podgraf) mají dokonce maximálně $2v - 4$ hran. Důkaz tentokrát ponecháme na vás.
- Lze pro každé v najít rovinný graf s v vrcholy a $3v - 6$ hranami?
- Naopak zkuste přijít na graf, který má $3v - 6$ nebo méně hran a není rovinný.

Vrchol nízkého stupně

V každém rovinném grafu existuje vrchol stupně maximálně 5. (Stupeň vrcholu je počet hran, které s vrcholem sousedí.) Proč tomu tak musí být? Kdyby všechny vrcholy měly stupeň alespoň 6, byl by součet stupňů alespoň $6v$. Jenže součet stupňů je přesně dvojnásobek počtu hran (každá hrana má dva konce), takže $e \geq 3v$, což je spor s předchozí větou.

Cvičení

- Najděte nejmenší rovinný graf, který má všechny stupně 5.
- Tvrzení o vrcholu nízkého stupně platí jen pro konečné grafy. Najděte nekonečný rovinný graf, jehož všechny vrcholy mají stupeň 6. Dokážete totéž i pro stupeň 42?

Natíráme 6 barvami

Konečně máme všechny potřebné ingredience (i se zdůvodněním, abyste nám věřili) a můžeme se pustit do barvení. Jelikož máme zaručený vrchol stupně maximálně 5, vezmeme ho, odebereme z grafu a zařadíme na zásobník. Odebráním se určitě neporušila rovinnost grafu, takže vezmeme další vrchol stupně maximálně 5. Takto pokračujeme rekurzivně, dokud nerozebereme celý graf a nenaskládáme všechny vrcholy na zásobník.

Jelikož jsme odebírali vrcholy stupně maximálně 5, má každý vrchol na zásobníku nad sebou nejvýše 5 sousedů v původním grafu. Z toho už je patrný barvicí algoritmus: budeme postupně odebírat z vrchu zásobníku a u každého vrcholu máme jistotu, že mezi jeho sousedy chybí jedna barva (některé sousední vrcholy jsou samozřejmě ještě neobarvené, ale obarvených je maximálně 5).

Co se týče implementace, jediným problémem je hledání vrcholů stupně 5 a méně. Řešení však není těžké na vymyšlení, ani na programování: stačí na začátku naskládat všechny vrcholy stupně maximálně 5 do nějaké datové struktury (hodí se třeba

fronta) a potom si uvědomit, že při odebrání vrcholu z grafu je potřeba se podívat jen na jeho sousedy, jestli jim neklesl stupeň na 5.

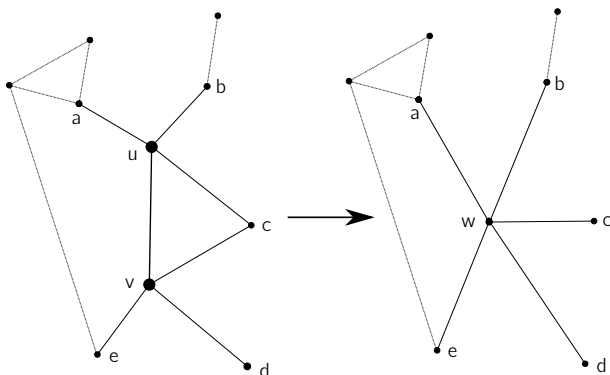
Cvičení

- S jakou nejlepší časovou a paměťovou složitostí lze náš algoritmus implementovat?

Třešnička na dortu: 5 barev

◊ Algoritmus na obarvení rovinného grafu 5 barvami má stejný průběh jako předchozí představený, tedy se postupně rozebírá graf (mimo jiné se odebírají vrcholy) a potom se barví vrcholy v obráceném pořadí, než se zpracovávaly. Jenže tentokrát nelze přímočaře obarvit vrcholy stupně 5 (s vrcholy s menšími stupni můžeme zacházet stejně).

K vyřešení problému se nám bude hodit operace *kontrakce hrany*, která jednu danou hranu odstraní a spojí její dva koncové vrcholy u, v do nového vrcholu w . Hrany vedoucí z u a v do jiných vrcholů nyní povedou z w , násobné hrany se smažou (tzn. pokud u a v měly společného souseda, z w do něj povede jen jedna hrana).



Prvně je důležité pozorování, že mezi sousedy vrcholu v stupně 5 v rovinném grafu, existuje alespoň jedna dvojice vrcholů, mezi kterými nevede hrana. (Kdyby tomu tak nebylo, obsahuje tento graf K_5 jako podgraf, a tudíž nemůže být rovinný.) Najdeme tedy dvojici x, y sousedů v , která není spojena hranou, a místo odstranění v provedeme kontrakci hran xv a yv do vrcholu w . Vrchol v přidáme na zásobník (při implementaci se hodí také uložit, jaké hrany se zkontrahovaly) a pokračujeme rekurzivně s kontrahovaným grafem.

Při samotném obarvování, narazíme-li na zkontrahovaný vrchol v , máme už obarvený vrchol w vzniklý kontrakcí (nechť má například barvu 1). Jelikož sousedi w zahrnují sousedy vrcholů x a y , dáme těmto dvou vrcholům barvu 1 a žádný jejich soused určitě už nedostal stejnou barvu. Vrcholu v pak přidělíme jinou barvu, kterou nemají jeho 3 zbývající sousedi, ani x (tedy ani y). Barev je 5, takže to akorát vyjde.

Tím jsme zakončili povídání o barvicích algoritmech, samotnou implementaci ponecháme čtenáři jako cvičení.

Poznámky

- Kdybychom definici rovinného nakreslení změnili a dovolili hrany kreslit pouze jako úsečky místo libovolných křivek, překvapivě se nic nezmění: každý rovinný graf má rovinné nakreslení, v němž jsou všechny hrany úsečky. Ale není to zrovna jednoduché dokázat.
- Stejně jako do roviny bychom mohli grafy kreslit třeba na povrch koule. Tím se také nic nezmění, zkuste sami vymyslet, jak z rovinného nakreslení udělat „kulové“ a naopak. Ale třeba anuloid (povrch pneumatiky) se už chová jinak, například zmíněný nerovinný graf K_5 se na anuloid dá nakreslit bez křížení hran.
- Jak poznat, jestli je daný graf rovinný nebo ne? Tak, že nalezneme jeho nakreslení v rovině, ale rychlý algoritmus není vůbec jednoduchý. Více o tomto problému se dočtete například na anglické Wikipedii pod heslem Planarity testing nebo v [GrafAlg].
- Při pohledu na mapu států lze vidět také jiný rovinný graf. Ten má jako vrcholy body, kde se střetávají hranice tří nebo více států, a hrany v rovinném nakreslení tohoto grafu vedou po hranicích. Vztah druhého rovinného grafu na mapě k prvnímu má svou abstrakci v teorii grafů: *duální graf* rovinného grafu má vrcholy odpovídající stěnám původního grafu a hrana mezi nimi vede právě, když v původním rovinném grafu spolu stěny sousedí. Sami si ověřte, že oba grafy jsou vůči sobě navzájem duální. Malé cvičení: jak vypadá duální graf duálního grafu nějakého rovinného grafu? (Tedy dvakrát uděláme z grafu duální graf.)
- Více informací o teorii (nejen rovinných) grafů najdete například v [Kapitoly] či [Demel].

Pavel Veselý, Martin Mareš a Petr Škoda

Úloha 18-5-5: Do vysokých kruhů

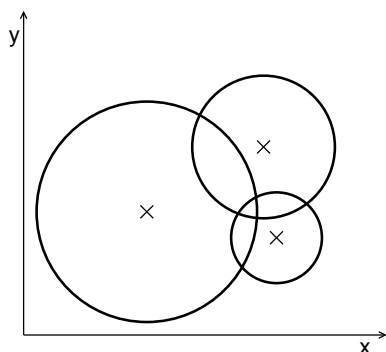
Napište program, který dostane na vstupu N kružnic zadaných souřadnicemi středu a poloměrem (souřadnice a poloměry jsou reálná čísla), a na výstup vypíše, na kolik částí dělí tyto kružnice rovinu. Můžete předpokládat, že se žádné tři kružnice neprotínají v jednom bodě. Rovina bez kružnic se považuje za jeden díl, jedna kružnice rozdělí rovinu na dva díly (vnitřek a vnějšek kružnice) atd.

Příklad:

Na vstupu jsou 3 kružnice:

x	y	r
1	1	0,9
2	0,8	0,4
1,9	1,5	0,6

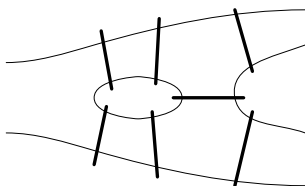
Rovina je pak rozdělena na 8 částí.



Eulerovské tahy

Historický problém

V roce 1735 se švýcarskému matematikovi Leonhardu Eulerovi na stůl dostal na první pohled jednoduchý problém, který mu předložil starosta města Královec (dnešní Kaliningrad). Královcem teče řeka Pregola, na ní je několik ostrovů a ostrovy jsou spojeny se zbytkem města mosty. Dobová ilustrace situaci vystihla takto (schématická kresba):



Pan starosta se pana matematika v dopise tázal, jestli je možné začít na některém z břehů (nebo ostrovů) a udělat si vycházku po městě tak, že se každým mostem projde právě jednou. Navíc chtěl procházku skončit na kusu suché země, ze kterého vyšel.

Profesor Euler jej nejprve chtěl poslat k šípku – problém jde snadno vyřešit rozбором případů, což by zvládli i tehdejší studenti střední školy (natož pak ti dnešní). Zachoval se ovšem jako pravý matematik – přišel na to, jak problém zobecnit, a mistrně vyřešil hádanku i pro všechna možná města, která kdy budou chtít pořádat podobné procházky.

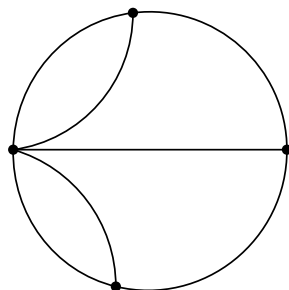
Eulerovský tah

Pojďme si nyní problém popsat abstraktně a tím si připomenout grafovou terminologii. Vrcholy našeho grafu jsou kusy pevniny, ať už to budou části města nebo ostrovy. Mezi dvěma vrcholy povede hrana, pokud jsou spojeny mostem, a onen most odpovídá hraně.

V tomto zadání má smysl uvážit, že mezi dvěma kusy pevniny povede mostů více – například v Praze jich vede tolik, že se na to ptají v lečkeré zeměpisné olympiádě. Graf, kde mezi vrcholy vede více hran, nazýváme *multigraf*, a pokud dvě hrany vedou mezi stejnými vrcholy, mluvíme o nich jako o *paralelních* hranách.

Obecná procházka v grafu z vrcholu A do vrcholu B (posloupnost hran taková, že cílový vrchol předchází hrany je počáteční vrchol hrany následující) se nazývá *sled* z A do B . Ve sledu se mohou opakovat jak hrany, tak vrcholy; sled tedy není řešením našeho problému (ve sledu je možné se vrátit po hraně, ze které jsme právě přišli).

Pro naši úlohu se hodí posloupnost hran taková, že vrcholy se opakovat mohou, ale hrany nikoli. Této posloupnosti se říká *tah* z A do B . Kdyby se neopakovaly ani vrcholy, pak posloupnost označujeme jako *cestu*. Tah (respektive sled) je *uzavřený*, pokud začíná v A a končí také v A .



Podíváme-li se tedy na mapu Královce jako na multigraf, ptáme se, zdali existuje uzavřený tah takový, že každou hranu navštíví právě jednou. Takovému tahu pak říkáme *uzavřený eulerovský*.

Mimochodem, tahu se „tah“ neříká jen tak náhodou. Děti se často ve školce překonávají v umění nakreslit obrázek jedním tahem, aby se tužkou nemuselo vracet po už nakreslené čáře. Pokud si obrázek představíme jako graf (čáry jsou hrany, místa jejich setkání vrcholy), pak eulerovský tah nalezneme jen v tom obrázku, který lze nakreslit jedním tahem. V uzavřeném eulerovském tahu se pak vrátíme i do místa, kde jsme začali.

Podmínky tahu

Je na čase poodhalit řešení našeho problému s eulerovským tahem. Půjdeme na to jako matematici – nejprve ukážeme *nutnou* a hned nato *postačující* podmínku. Nutná vlastnost grafu je taková, že bez ní eulerovský tah není možné najít; postačující vlastnost je ta, se kterou vždy eulerovský tah najít umíme. Jsou-li obě podmínky stejné, pak se jedná o ekvivalenci, a tak tomu bude i nyní.

Představme si, že jsme kouzlem nějaký uzavřený eulerovský tah našli, ať už je jakýkoli. Vždy, když se dostaneme do jednoho vrcholu (a není důležité, jestli už jsme v něm byli, nebo ne), tak z něj musíme hned také odejít, abychom tah uzavřeli. A protože tah je eulerovský, každou hranou projdeme jen jednou, takže tyto dvě hrany (tu příchozí a odchozí) už nepoužijeme. U každého vrcholu mimo výchozí tedy platí, že hrany tvoří dvojice – jedna, co vedla dovnitř, a jedna, která z něj vedla ven.

Podobná věc platí i pro startovní vrchol. Sice do něj nevstoupíme poprvé pomocí hrany, takže počet navštívených hran u něj bude stále lichý – ale jen do chvíle, než se do něj naposledy vrátíme a skončíme, protože skončením jsme použili poslední hranu, která bude tvořit dvojici s hranou první.

Jakou vlastnost grafu jsme odhalili? Neplatí, že graf má sudý počet hran (protože trojúhelník jedním tahem nakreslíme a přesto má 3 hrany), ale platí, že do každého vrcholu vede sudý počet hran, tedy že graf má *všechny stupně sudé*. Nezapomeňme také na to, že graf musí být souvislý – dva oddělené obrázky jedním tahem bez zvednutí tužky nenakreslíme. Máme nutné podmínky!

Nalezení tahu

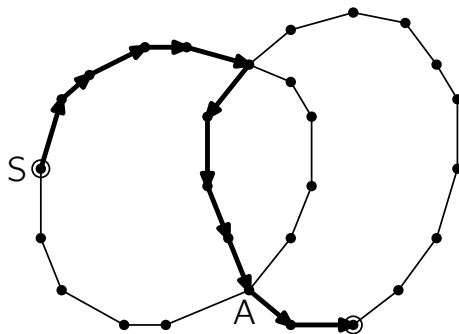
Zbývá tedy ověřit, že podmínky jsou i postačující. Mějme souvislý graf, který má všechny stupně sudé. Umíme v něm vždy najít uzavřený eulerovský tah? Ověřme to, jak se na informatiky patří – algoritmem.

Předložený algoritmus je založený na vylepšeném prohledávání do hloubky, tedy DFS, o kterém jste si mohli přečíst v první grafové kuchařce.

Vyberme si vrchol, v něm začneme. Náš algoritmus musí umět označovat hrany jako „probrané“, jako to dělá DFS. Vyberme si tedy jednu hranu a pokračujme dále, zatím bez vypisování.

Po nějakém tom procházení se jistě stane, že jsme se zastavili – vrchol už nemá žádné nepoužité hrany. Nutně to znamená, že to je ten vrchol, ve kterém jsme začínali.

V procházení do hloubky se vracíme zpět, ale my k tomu přidáme vypisování cesty – postupně pozpátku vypisujeme hrany, kterými se vracíme zpět v prohledávání.



Na obrázku výše je příklad právě probíhajícího algoritmu. Začal ve zvýrazněném vrcholu vlevo, procházel po šipkách až do bodu A , kde volil hrany tak, že hned skončil na začátku. Dále pokračoval vypisováním hran pozpátku, až došel zase do bodu A . Zde si vybral jednu ještě nepoužitou hranu a po ní prošel celou druhou kružnici – zbytek hran – zpět do bodu A . Nyní vypisuje hrany pozpátku od bodu A .

Buď tímto výpisem dojdeme až na začátek, nebo se dostaneme do vrcholu, který má ještě nějaké nepoužité hrany (situace může vypadat třeba jako na obrázku). Potom vypisování zastavíme a pokračujeme v prohledávání DFS přes nepoužitou hranu. I tam se to může zastavit (a zastaví), i tam začneme vypisovat pozpátku. Nakonec dojdeme do původního místa rozbočení, a budeme opět pozpátku vypisovat hrany, které nás nakonec dostanou až na počátek, kde skončíme.

Najde tento algoritmus opravdu korektní uzavřený eulerovský tah? Graf byl souvislý a o algoritmu DFS se ví, že v takovém případě navštíví každou hranu právě jednou. Algoritmus opravdu vypisuje cyklus – jen je u něj trochu zvláštní způsob, jak ho vypisuje. Když dojde na křižovatku s ještě nepoužitými hranami, tak výpis zastaví, tiše po nich kráčí, označuje si je a vypisuje, až když se po nich vrací. Ověříme si, že hrany opravdu navazují.

V duchu argumentů z předcházející části víme, že jediný vrchol grafu s lichým počtem nepoužitých hran je právě ona křižovatka – a algoritmus DFS prochází graf podobně, jako jsme ho procházeli v minulé sekci, takže právě do tohoto vrcholu algoritmus dojde, až se průchod touto částí grafu zastaví.

Jakmile sem program dojde (a nezbudou mu volné hrany), začne cestovat zpět a hrany vypisovat – a opravdu, pokračuje se tedy z místa, kde naposledy přestal, a program vskutku vypíše tah přes všechny hrany v grafu – uzavřený eulerovský tah.

Věta o eulerovském tahu v celé své kráse tedy zní: *(Multi)graf obsahuje uzavřený eulerovský tah právě tehdy, když má všechny stupně sudé a je souvislý.*

Je třeba podotknout, že složitost našeho algoritmu na bázi DFS je lineární vůči velikosti grafu (počtu vrcholů a hran). Existují i jiné algoritmy pro hledání eulerovského tahu, jedna varianta například prochází grafem a vybírá si na křižovatkách takové

hrany, které souvislost grafu pokud možno nepoškodí. Tyto algoritmy už nemusí mít nutně lineární časovou složitost.

Jiné druhy procházek

Nejen kreslením obrázků ze stejného bodu živ je člověk. Co kdybychom mohli začít a skončit v jiném místě, tedy ptali se po neuzavřených eulerovských tazích, změnilo by se něco? Není tomu tak, pouze nutné a postačující podmínky si vyžádají, aby všechny vrcholy měly sudý stupeň až na právě dva vrcholy, které mají lichý stupeň. Pokud nám to nevěříte, zkuste si to rozmyslet sami, opravdu to není těžké.

Smysl také dává zkusit najít ne uzavřený tah, ale uzavřenou cestu – uzavřenou cestu přes všechny vrcholy, která navštíví každý vrchol právě jednou (říká se jí „Hamiltonovská cesta“). Bohužel, ačkoli jsou problémy příbuzné, musíme vás zklamat – není znám žádný efektivní (polynomiální) algoritmus na tento problém, a kdyby jej někdo z vás našel, vyřešil by otázku „P vs. NP“, o níž se více dočtete v kuchařce o těžkých problémech.

V matematice se také někdy zmiňují „náhodné procházky“ po grafech – můžete si je představit tak, že se po mostech města Královce motá opilec, který si hází (opilou nebo spravedlivou) mincí a podle toho se rozhoduje, přes který most jít dál. Použití mají tyto modely hlavně v matematické teorii grafů a teorii pravděpodobnosti. O tom si můžeme povědět zase někdy jindy.

Martin Böhm

Úloha 23-2-3: Projížďka

Turing před projížďkou na kole pečlivě prozkoumal terén, který si reprezentoval jako seznam rozcestí a cest mezi nimi. Silnice jsou však nevyrovnané – některé jsou krátké a klidné, dokonce vedou z kopce; jiné jsou dlouhé, klikaté a strmé, takže velmi unavují. Proto každé z nich přidělil celé číslo vyjadřující tuhle subjektivní obtížnost – na kladně ohodnocených cestách si bude odpočívat a na záporně ohodnocených tuhle nashromážděnou energii vydá.

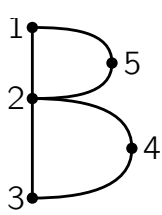
Teď by od vás chtěl, abyste napsali program, který mu najde takovou cestu (včetně začátku – a může začínat na libovolném rozcestí), po které jednak projede všechny silnice *právě jednou*, druhak bude na každém rozcestí součet všech Turingem do té chvíle projetych silnic *nezáporný* a navíc se vrátí na rozcestí, na kterém začal. Chcete-li a myslíte-li si, že vám to pomůže, předpokládejte klidně, že z každého rozcestí vychází sudý počet silnic.

Úloha 23-3-4: Psaní písmen

Každé písmeno se skládá z bodů a linií, které je spojují. V jednom bodě může začínat i končit více linií. Při psaní perem lze psát víc navazujících linií jedním tahem, nejde-li to, musí se pero zvednout a začít jinde. Kolikrát nejméně je potřeba pero zvednout? Na vstupu dostanete neorientovaný graf o N vrcholech a M hranách a vypíšete, kolika nejméně tahy lze nakreslit.

Samozřejmě šetříme, takže je zakázáno jakoukoli hranu nakreslit více než jednou. Jinak řečeno, nesmíte se vracet po již nakreslených liniích.

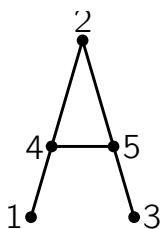
Příklad vstupu:



5 6
1 2
2 3
3 4
4 2
2 5
5 1

výstup: 1

Jiný příklad:



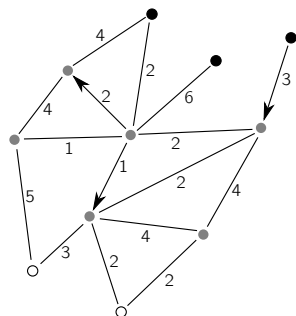
5 5
1 4
4 2
2 5
5 3
4 5

výstup: 2

Toky v sítích

Ruský petrobaron vlastní ropná naleziště na Sibiři a trubky vedoucí do Evropy. Trubky vedou mezi nalezišti, uzlovými body a koncovými body, kde ropu přebírají odběratelé. Každá trubka může a nemusí být definována, kterým směrem jí má téci ropa. Pro každou trubku zvlášť víme, kolik nejvýše jí za hodinu protlačíme.

Naleziště jsou bezedná a mohou posílat neomezená množství ropy. Odběratelé také dokáží neomezená množství ropy z koncových bodů odebírat. Petrobaron čelí problému, jak protlačit danou distribuční síť co nejvíce ropy za hodinu ze zdrojů k odběratelům.



Zapeklité je to hlavně proto, že v uzlových bodech nelze ropu hromadit, ani pálit – rozhodně tedy nejde bez rozmyslu přikázat, ať každou trubkou teče maximum, protože bychom poškodili cenná zařízení a v hnusu labutě zahubili.

Zmatematizování

V zadání vidíme graf, který obsahuje orientované i neorientované hrany, kde je nějaká podmnožina vrcholů označená jako zdroje a jiná jako ... řekněme tomu třeba stoky.

Abychom měli situaci jednodušší, zbavíme se hned na úvod mnohočetnosti zdrojů a stoků. Přikreslíme si dva nové vrcholy – z nadzdroje budeme posílat ropu do všech zdrojů, do nadstoku budeme posílat ropu ze všech stoků. Kapacitu přikreslených hran pak nastavíme na nekonečno.

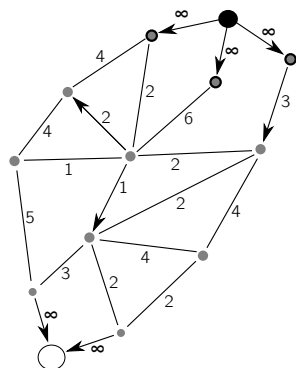
Teď nám stačí vymyslet algoritmus, který řeší problém s právě jedním zdrojem a právě jedním stokem. Každý vstup totiž popsaným způsobem převedeme, pošleme ho algoritmu a z výstupu prostě jen odstraníme dva přidané vrcholy a připojené hrany.

Podobně se zbavíme neorientovaných hran. Každou takovou hranu v každém zadání změním na dvojici protisměrných orientovaných hran se stejnou kapacitou. V algoritmu pak už můžeme počítat jen s hranami orientovanými.

Dostáváme se nyní k nejdůležitějšímu – podmínkám na hledaný tok.

Na vstupu dostáváme ohodnocení hran nezápornými čísly a naším úkolem je sestavit jiné ohodnocení těch samých (všech) hran. Je důležité, aby se nám to nepletlo – ohodnocení ze vstupu se říká *kapacita* a značí se $c(e)$, konstruované ohodnocení se jmenuje *tok* a říkáme mu $f(e)$.

Konstruované ohodnocení maximalizujeme, ale omezuje nás kapacita a *Kirchhoffův zákon*. Tak budeme říkat podmínce na to, že součet toku na hranách, které do vrcholu



vstupují, musí být stejný jako součet toku na hranách, které z vrcholu vystupují. Máte-li rádi fyziku nebo, důvod k takovému pojmenování jistě chápete.

Formálně ony dvě podmínky vypadají takto:

$$\forall e \in E : f(e) \leq c(e)$$

$$\forall v \in V \setminus \{z, s\} : \sum_{\overrightarrow{uv} \in E} f(\overrightarrow{uv}) = \sum_{\overrightarrow{v\bar{u}} \in E} f(\overrightarrow{v\bar{u}})$$

Kirchhoffova podmínka se samozřejmě netýká ani zdroje, ani stoku – tam nám naopak jde o to ji co nejvíce porušit. Velikost toku je nejsnazší měřit na nich. Budeme ji definovat jako rozdíl mezi součtem odtoků a součtem přítoků ve zdroji.

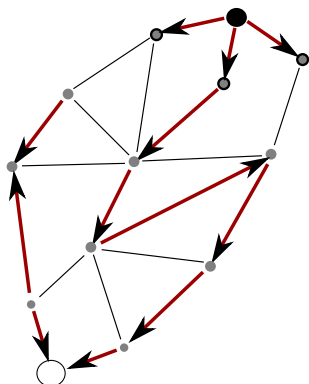
Cvičení

- Neorientované hrany, neboli obousměrné trubky, si zaslouží podrobnější rozbor, než jaký jsme jim věnovali v textu. Jak spolehlivě převedeme řešení algoritmu do původní sítě?
- Vymysleli jsme, jak vyřešit více zdrojů a stoků a jak ošetřit obousměrné trubky. Co kdyby bylo v zadání omezení na průtok vrcholy?
- Umíte dokázat, že je absolutní hodnota rozdílu přítoků a odtoků stejná na zdroji i na stoku? Tedy že bychom mohli velikost toku stejně tak dobře měřit i na stoku?

Řešení

Problém je velmi studovaný a k jeho řešení existují dva velké přístupy, které jsou humorně protikladné. Ten první vezme nulový tok a opatrně ho zlepšuje. Druhý si napíská veliké ohodnocení hran, které ani tokem není, a pak ho opravuje.

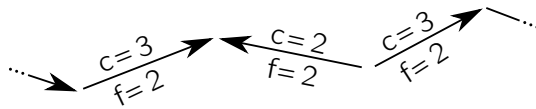
Předvedeme si onen první způsob a algoritmus, který se podle svých autorů jmenuje Fordův-Fulkersonův. Bude se nám odteď hodit tvářit se, že hrany mezi dvěma vrcholy buď vedou oběma směry, anebo žádným (tj. tam, kde hrana vedla jen jedním směrem, doplníme hranu druhým směrem). Přidaným hranám dáme nulovou kapacitu.



Představme si graf, na kterém počítáme tok a dejme tomu, že už nějaký tok máme – třeba prázdný. Představme si, že jsme ropný magnát a každý rozdíl mezi kapacitou potrubí a jejím využitím (tokem) nás stojí miliony dolarů. Už jsme se smířili s tím, že každá trubka nemůže být využita na maximum, ale ... zkusme si vyznačit ty hrany, kde $c(e) \neq f(e)$.

Co když existuje cesta z nadzdroje do nadstoku, která vede pouze po takových hranách? Můžeme vzít minimum z rozdílů na každé hraně a o toto číslo zvýšit tok na každé z nich! Ani kapacitní, ani Kirchhoffovu podmínku to jistě nepoškodí.

Pokud žádnou takovou cestu nevidíme, znamená to, že tok vylepšit nejde? Ne úplně. Představte si následující situaci:



Copak nejde zlepšit? Jde! Není na to první pohled úplně jasné, ale můžeme zlepšovat výsledný tok i tím, že ho na protisměrné části cesty snížíme. Samozřejmě však nesmíme nastavovat tok záporný.

(Je smutné, že si teď trochu kazíme grafovou terminologií – co je to za cestu v orientovaném grafu, která nemusí respektovat orientaci hran?)

Takže jaká je přesně podmínka pro „vyznačení“ hrany \overrightarrow{uv} ? Nastává $f(\overrightarrow{uv}) < c(\overrightarrow{uv})$ nebo $f(\overrightarrow{vu}) > 0$. Potom ji lze zlepšit o $c(\overrightarrow{uv}) - f(\overrightarrow{uv}) + f(\overrightarrow{vu})$.

Hledání všech vhodných („zlepšujících“) cest tedy můžeme dělat prostým prohledáváním do šířky přes vyznačené hrany. Budeme to dělat opakovaně znovu a znovu, až žádnou takovou nenajdeme, a pak vrátíme získaný tok jako výsledek.

Analýza algoritmu

Správnost

Zavolali jsme algoritmus na prázdný tok, ten ho zlepšil do situace, ve které neexistuje zlepšující cesta.

Znamená tato neexistence, že je výsledný tok maximální? Opačná implikace je jasná – maximální tok zlepšit žádným způsobem nepůjde, takže ani přes zlepšující cesty.

Když zkusíme algoritmus pustit na graf, kde už žádná taková cesta není, můžeme si poznamenat všechny vrcholy, kam jsme se pomocí prohledávání zlepšitelných hran ještě dostali. Tato množina bude jistě obsahovat zdroj (tam jsme začali) a jistě nebude obsahovat stok (to by existovala zlepšující cesta).

Na hranách mezi touto množinou a jejím doplňkem nemůžeme zlepšovat, jinak by se po nich náš program pustil dál a množinu vrcholů, kam se dostal, by rozšířil. Všechny hrany směřující ven tedy mají $f(e) = c(e)$, pro všechny hrany směřující dovnitř platí $f(e) = 0$.

Tyto hrany tvoří řez naším grafem. Dovolám se v tuto chvíli na vaši intuici (pro korektní důkaz viz [Skriptička]) – tok nemůže být větší než libovolný řez. Z toho už dostáváme, že náš algoritmus našel tok maximální, protože našel také řez, který zaručuje, že nemůže existovat tok větší.

Časová složitost

Je možné dobu běhu omezit počtem vrcholů a hran? Výše uvedeným postupem na grafu s celočíselnými kapacitami každou nalezenou cestou zvýšíme tok alespoň o jednotku, takže program nebude běžet déle, než je součet všech kapacit. Ale to není moc uspokojivý odhad, protože záleží na ohodnocení.

Pokud budeme hledat cesty skutečně prohledáváním do šířky, bude počet kroků v $\mathcal{O}(nm^2)$, protože se dá ukázat, že se hrany, které při zlepšování cesty tvoří minimum, postupně vzdalují od zdroje. Pak máme $\mathcal{O}(m)$ času k nalezení cesty a m hran,

kteřé se nejvřýše n -krát mohou vzdálit. řZe to tak skutečně je, je lehce zdlouhavé intelektuální cvičení (viz [IntroAlg]).

O vylepřění daného postupu a rozbor jednoho alternativního si můžete přečíst v [ADS2].

Cvičení

- Důležitou vlastností algoritmu je, řže když dostane celočíselné kapacity, vrátí celočíselný tok. Bude se nám to hodit v aplikacích. Umíte to dokázat?
- Rozdíl mezi Fordem-Fulkersonem, který hledá cesty obecným způsobem, a takovým, který to dělá prohledáváním do šířky, je ze složitostního hlediska docela velký, a proto se tomu druhému občas říká Edmondsův-Karpův. Najděte malý graf a nevhodnou posloupnost cest, která způsobí, řže F-F poběží skutečně v závislosti na velikosti kapacit.
- Můžete dokonce zkusit využít zlatého řezu k nalezení grafu s reálnými kapacitami, na kterém F-F pro danou (nešikovnou) posloupnost cest nikdy neskončí.
- Skončí algoritmus v konečném řese, jsou-li kapacity čísla racionální?

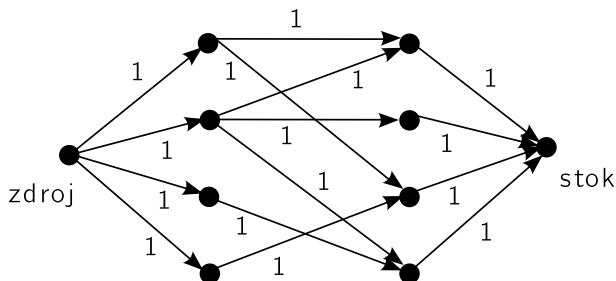
Užití

Párování v bipartitních grafech

Máme-li za úkol najít na plese co nejvříce tanečnicím tanečnřka, kterého znají, stojřme před zásadním a nelehkým úkolem.

K tomu se nám bude hodit znát *bipartitní graf*, v němř jsou vrcholy rozděleny na dvě skupiny (mohou být i prázdné) a hrany vedou jen mezi těmito skupinami. Pokud jsou tedy vrcholy u, v ve stejné skupině, nikdy mezi nimi nevede hrana, jinak tam být může, ale nemusí. Skupinám vrcholů se někdy říká *partity*.

Na základě známosti postavřme bipartitní graf mezi partitou tanečnřků a partitou tanečnic, přidáme zdroj za kluky a stok za holky. Oba nové vrcholy k nim připojřme hranami s jednotkovou kapacitou, hranám v bipartitním grafu také nastavřme jednotkové kapacity a nakonec všechno zorientujřme směřem do stoku.



Maximální celočíselný tok, který na tomto grafu získáme, nám hrany bipartitního grafu rozdělř na nevybrané s tokem 0 a vybrané s tokem 1. Můžou vybrané hrany sdřlet tanečnřka? Těžko, když do něj teče nejvřýše jednotkový tok a musí platit Kirchhoffův zákon. Podobně s tanečnicemi.

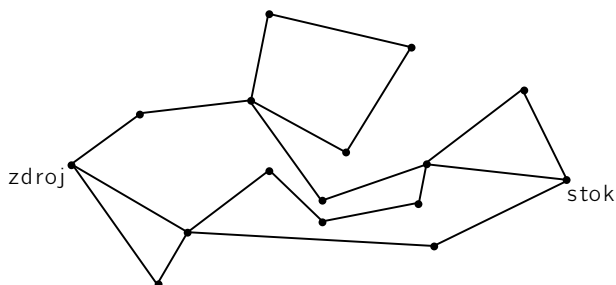
Vybrané hrany nám proto vytvoří párování. A protože jsme našli maximální tok, jde o párování největší. Kdyby existovalo párování větší, dokázali bychom podle něj zvětšit tok.

Hledání hranově a vrcholově disjunktčních cest

Chceme-li se v grafu G dostat z vrcholu u do vrcholu v , může nás zajímat (třeba kvůli spolehlivosti, s jakou se umíme dopravit do cíle), kolik mezi nimi existuje cest, které nesdílí hrany, nebo nesdílí vrcholy. (Druhá podmínka je silnější. Když dvě cesty nesdílí vrcholy, nesdílí hrany.)

Oba tyto problémy lze převést na hledání maximálního toku. V obou případech nastavíme u jako zdroj a v jako stok. V prvním případě nastavíme jednotkové kapacity všem hranám, v druhém navíc všem vrcholům (kapacitu vrcholu a přidělíme tak, že ho rozdělíme na dva vrcholy b a c , do b povedou všechny hrany vedoucí do a , z c budou vycházet všechny hrany původně vycházející z a a z b do c přidáme hranu o kapacitě vrcholu).

Ford-Fulkerson nastaví některým hranám jednotkový tok, některým nulový. Nulové nyní z grafu vyhodíme. Pokud jsme hledali hranově disjunktční cesty, můžeme nyní získat třeba takovýto graf:



Jak z něj vykresat kýžený výsledek? Začneme procházet ze zdroje zbylé hrany. Vždy, když se dostaneme do vrcholu, ve kterém už jsme v tom samém průchodu byli, vyhodíme z grafu všechny hrany cyklu, který jsme tímto objevili. (Hodnota toku se tím nezmění.)

Průchodem grafu se vždy můžeme dostat až do stoku (všude jinde budeme moci podle Kirchhoffova zákona jít dál – dost to připomíná úvahu o eulerovských tazích) a protože jsme mezitím horlivě odstraňovali cykly, dostali jsme cestu. Vrátime ji jako jeden výsledek, smažeme její hrany a pokud ještě tok není nulový, pokračujeme dál.

Počet cest je tedy velikost toku. Podle Mengerovy věty je navíc minimum počtu hranově/vrcholově disjunktčních cest pro všechny dvojice vrcholů roven stupni hranově/vrcholové souvislosti grafu – pokud máme dost času zavolat tokový algoritmus pro každou dvojici vrcholů našeho grafu, získáváme tak použitelný (polynomiální) postup, jak vypočítat souvislost grafu (graf je hranově/vrcholově k -souvislý, když zůstane souvislý po odebrání libovolných $k - 1$ hran/vrcholů).

Cvičení

- Úvaha nebyla naprosto přímočará kvůli cyklům v nalezeném toku. Říká se jim cirkulace. Je jasné, že v případě hledání hranově disjunktčních cest vzniknout mohou. Co v případě vrcholově disjunktčních, tedy v situaci, kdy jsme omezili tok vrcholy?
- Nepracuje náhodou Edmondsův-Karpův algoritmus rychleji, pokud je graf, jak jsme teď opakovaně viděli, ohodnocený toliko nulami a jedničkami?

Lukáš Lánský

Úloha 23-4-1: Studenti a profesori

Studenti na Stanfordově univerzitě se chtějí prosadit a napsat co nejvíce článků, přičemž každý z nich má vytipováno několik profesorů, pod jejichž vedením by chtěl článek psát. S jinými profesory spolupracovat nechce a nebude.

Studenti jsou schopni psát maximálně K článků najednou. Leč čas profesorů je omezený, každý z nich je totiž ochoten spolupracovat maximálně s K studenty, přičemž je jim jedno, kteří to budou.

Vášim úkolem je najít algoritmus, který zjistí, jestli je možné, aby každý student psal právě K článků a každý profesor spolupracoval právě s K studenty, a pokud ano, tak vypsát, který student bude spolupracovat s kterým profesorem.

Můžete předpokládat, že profesorů i studentů je stejně, totiž N , a nemusíte uvažovat situaci, že by student chtěl psát u jednoho profesora více článků.

Příklady: pro vstup $N = 4$, $K = 2$, student S1 chce psát článek s profesory P1 a P2, student S2 s P1, P2, P3, P4, student S3 s P2, P3, P4 a student S4 s profesory P3, P4, jsou řešením tyto páry student–profesor: S1–P1, S1–P2, S2–P1, S2–P3, S3–P2, S3–P4, S4–P3, S4–P4.

Pro vstup $N = 5$, $K = 2$, studenti S1 a S2 chtějí psát u profesorů P1, P2, P3, student S3 u P3, P4, P5 a studenti S4, S5 u P4, P5, řešení neexistuje. Existovalo by, kdyby bylo $K = 1$, ale to už je zase jiný vstup.

Úloha 23-5-6: Limity a grafy

Dostanete na vstupu orientovaný graf s kladně celočíselně ohodnocenými hranami. Dále tam bude pro každý vrchol dvojice kýžených limitů – minimální součet vstupních hran a maximální součet výstupních hran.

Vášim úkolem je najít nové nezáporné celočíselné ohodnocení každé hrany, které nebude větší než to původní a které bude dohromady se všemi ostatními novými ohodnoceními respektovat dané limity.

Intervalové stromy

Představme si, že máme posloupnost celých čísel p_0, p_1, \dots, p_{N-1} , se kterou budeme průběžně provádět tyto dvě operace:

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

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

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

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

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

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

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

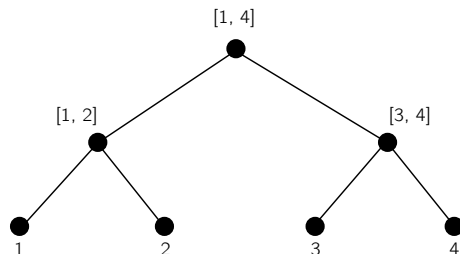
Každý dotaz dokážeme zodpovědět v konstantním čase. Celý algoritmus má tedy složitost $\mathcal{O}(N + D)$, kde N je délka posloupnosti a D je počet dotazů.

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

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

Zavedení intervalového stromu

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

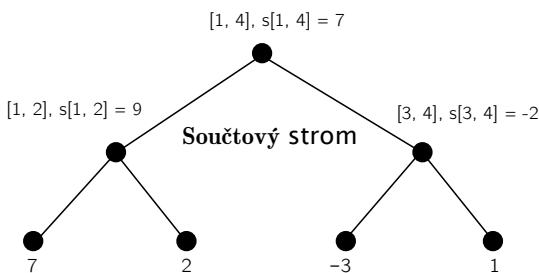


Intervalových stromů existuje více druhů. Obvykle je rozlišujeme podle toho, jaké informace si v nich pamatujeme. Například ve stromě pro součty si každý vrchol pamatuje součet na svém intervalu, ve stromě pro maxima si pamatuje maximum

na intervalu, apod. Můžeme ale klidně mít strom, který si pamatuje, jestli celý jeho interval obsahuje jen jednu hodnotu a pokud ano, tak jakou.

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

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

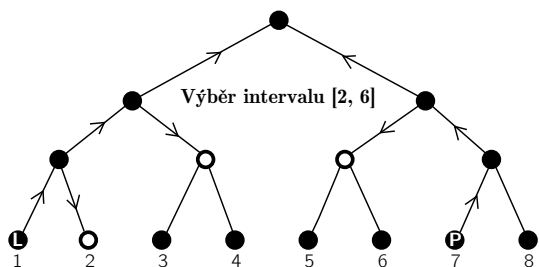


Posloupnost tedy prodloužíme tak, aby její velikost byla mocnina dvojky minus dva (na její konec přidáme nějaké prvky). Všimněte si, že tím jsme strom nezvětšili více než dvakrát a že nám nezáleží na tom, jaké prvky jsme do stromu přidali, protože s nimi nikdy nebudeme pracovat. Nyní k jednotlivým operacím.

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

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

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



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

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

Změna prvku posloupnosti má časovou složitost $\mathcal{O}(\log N)$, protože jsme na každé hladině změnili pouze jeden interval a strom má $\mathcal{O}(\log N)$ hladin. Zjištění součtu na intervalu má také složitost $\mathcal{O}(\log N)$, jelikož jsme do výsledku přidali maximálně $2 \log N$ intervalů: nejvýše $\log N$ při cestě z listu $a - 1$ a $\log N$ při cestě z $b + 1$.

Implementace intervalového stromu

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

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

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

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

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

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

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

```

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

```

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

Cvičení

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

Použití intervalového stromu

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

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

Fenwickův strom

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

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

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

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

A zde je funkce pro zjištění prefixového součtu:

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

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

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

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

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

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

Cvičení

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

Karel Tesář

Úloha 16-3-1: Fyzikova blecha

Newtoon trénuje svoji blechu na bleší turnaj. Ten probíhá na svislé stěně, na které jsou vodorovné plošinky. Cílem blechy je slézt ze startovací polohy co nejdříve na podlahu. Pohyb blechy závisí na tom, zda je blecha na nějaké plošince, či padá. Pokud padá, klesne za jednu blechovteřinu o jeden blechometr. Pokud je na plošince, posune se za jednu blechovteřinu o jeden blechometr vlevo či vpravo.

Závod tedy probíhá tak, že blecha padá, padá, až dopadne na plošinku. Pak se rozhodne (nebo jí její majitel přikáže), zda půjde doleva nebo doprava, a jde, dokud nedojde na konec plošinky. Z ní pak seskočí a zase padá, dokud se nedostane na podlahu. Vítězí blecha, která přistane na podlaze jako první. Ovšem je nutné, aby žádná blecha nespadla z větší výšky než v blechometrů, jinak se totiž po dopadu urazí a odmítne pokračovat v závodě.

Newtoon již vytrénoval svou blechu tak, že ho poslouchá na slovo. Problém je ten, že sám neví, jak blechu navigovat, aby prošla bludištěm nejkratší možnou cestou. Protože Vás ale zajímá, jak vypadá blecha, která poslouchá, rozhodli jste se Newtoonovi pomoci.

Na vstupu dostanete jednak počáteční souřadnice blechy (v celých blechometrech), v , což je největší výška, ze které může blecha spadnout, aby se neurazila, a N , což je počet plošinok na stěně. Dále dostanete popis N plošinok, u každé plošinky souřadnice jejího horního dolního rohu a její šířku (vše opět v celých blechometrech). Všechny plošinky jsou vysoké jeden blechometr a žádné dvě se nedotýkají. Blecha je na podlaze, pokud se nachází na souřadnicích $[x; 0]$, kde x je libovolné celé číslo.

Výstupem vašeho programu je nejmenší počet blechovteřin, které bude blecha potřebovat, aby se dostala na podlahu. Kromě tohoto počtu vypište i počet plošinok, na které blecha dopadne, a u každé plošinky (v pořadí, jak na ně blecha dopadá) rozhodněte, zda má blecha jít vlevo či vpravo. Pokud úloha nemá řešení (moc malé v), vypište odpovídající zprávu.

Příklad: Blecha se nachází na souřadnicích $[5; 12]$, $v = 4$, $N = 3$. Plošinky jsou $[3; 8]$; 5 , $[3; 4]$; 5 a $[7; 6]$; 3 ([souřadnice levého horního rohu]; délka). Nejkratší cesta trvá 17 blechovteřin, blecha navštíví všechny tři plošinky a půjde vpravo na první, vlevo na druhé a vpravo na třetí navštívené plošince.

Těžké problémy

Představme si, že jsme v bludišti a hledáme (naš algoritmus hledá) nejkratší cestu ven. Rychle nás napadne, že bychom mohli použít prohledávání do šířky a cestu najít v čase lineárním ku velikosti bludiště. To je asymptoticky nejlepší možné řešení, v nejhorším případě bude totiž bludiště jedna dlouhá nudle a i nejkratší cesta bude dlouhá lineárně vůči velikosti bludiště.

Ve skutečném životě však „kulišáci“ znají lepší řešení – podvádět! Prostě si od kamaráda půjčíme mapu bludiště s vyznačenou nejkratší cestou a pak poběžíme hned tou nejkratší cestou, aniž bychom kdekoli ztráceli čas.

V nudlovém bludišti (nejkratší cesta má zhruba stejně vrcholů jako celý graf) jsme si vůbec nepomohli (takže je řešení asymptoticky stejně dobré). V alespoň trochu spleťtém bludišti už budeme v cíli dříve než náš kamarád, který bloudí (prohledává) do šířky.

Existují tedy problémy, kde by se i v nejhorším možném případě vyplatilo podvádět pomocí taháku? Ano, zde je příklad – opět jsme v labyrintu, ale tentokrát jsou na všech stanovištích umístěny koláčky. Labyrint je to zvláštní, cesty se v něm nekříží, ale je tam plno nadchodů a podchodů.

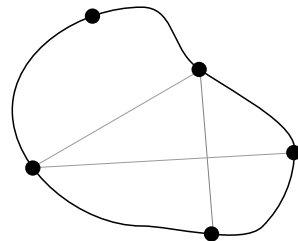
Naším cílem je najít okružní cestu ze startovního místa zpátky na start, abychom každé stanoviště s koláčkem prošli právě jednou (protože víc než jeden koláček nám nedají).

Kdybychom tady chtěli použít procházení do šířky, bylo by to opět možné – ale tentokrát bychom se museli mnohokrát vracet, protože posloupnost stanovišť (začátek, první, druhé) může být špatná, zatímco posloupnost (začátek, druhé, první) už může být dobrá.

Přesněji řečeno, už by neplatilo, že při prohledávání do šířky každé stanoviště navštívíme nejvýše jednou, ale každou *posloupnost* stanovišť navštívíme nejvýše jednou. Projít všechny nám potvrzuje, matematicky řečeno, exponenciálně mnoho času vůči velikosti bludiště.

Kamarád s tahákem je na tom pořád dobře – prostě si pořídí jiný plán, na kterém bude mít vyznačenou cestu, po které má jít, aby vyhrál.

Ta cesta má stejně křižovatek jako bludiště samo, a tak bude jeho nápověda lineárně velká vůči velikosti bludiště a průchod nápovězenou cestou bude trvat také lineárně. Podvodník tedy vyhrává i asymptoticky. Bídák!



Všechny by nás zajímalo, jestli by bylo možné najít tu nejlepší cestu bez podvádění v rozumně krátkém (řekněme polynomiálním) čase. Tato otázka je ekvivalentní známé otázce P vs. NP. Pojďme ta tajemná písmena přesně definovat.

Podvádíme s certifikáty

V teorii složitosti se často omezujeme jen na jeden typ problému, takzvaný *rozhodovací problém*. To je vlastně otázka, na kterou existují dvě možné odpovědi: ANO

a NE. Například „Existuje cesta z bludiště délky k ?“ nebo „Je součet čísel $8 + 3$ roven 5 ?“

Ve zbytku kuchařky už budeme pracovat jen s nimi – skoro vždy se rychlé řešení rozhodovacího problému dá převést na rychlé řešení příslušného vyhledávacího problému, jako *Nalezněte nejkratší cestu z bludiště*.

Rozhodovací problém (dále už jen problém) bude náležet do *třídy problémů P* (třída je zde jen pomocné označení pro nekonečnou množinu), pokud existuje polynomiální algoritmus, který pro zadaný vstup odpoví korektně ANO nebo NE.

Taháku z předchozí kapitoly se v literatuře říká *certifikát*. Formálně to je jen jakási polynomiálně velká informace. Můžeme si jej představit jako data, která náš program nalezne v „našeptávacím“ vstupním souboru, ke kterému program z třídy P nemá přístup.

Problém bude náležet do *třídy problémů NP* (nepoctivci), pokud existuje algoritmus a ke každé odpovědi ANO vhodný certifikát tak, že algoritmus je schopen pomocí certifikátu ověřit, že odpověď je skutečně ANO. Čili má-li ten program správný tahák, musí být schopen bludištěm projít rychle.

Zde si dejme pozor na to, že definice nedovoluje „podvádět na druhou“ – nemůžeme si do pomocného souboru prostě uložit ANO a pak jej vypsát. Tak by se pak dal řešit libovolně složitý problém, i problémy mimo třídu NP! Jen na okraj – takové opravdu existují.

Onen algoritmus musí být schopen řešení ověřit, tedy odpovědět ANO tehdy a jen tehdy, pokud mu to napověděl certifikát a odpověď je správná. Kdyby byla skutečná odpověď NE a certifikát chybně tvrdil, že ANO, algoritmus musí být napsán tak, aby oznámil NE.

Co přesně bude certifikát, záleží na zadané úloze – často to bývá právě ono nejlepší možné řešení, kterého se stačí držet a najdeme hledanou odpověď (nebo zjistíme, že úloha nemá řešení).

Asi vám bylo hned jasné, že každý program z P patří také do NP – jakmile známe polynomiální řešení bez nápovědy, certifikátem může být i třeba prázdný soubor! Horší je to s problémy, pro které potřebujeme pro polynomiální vyřešení nějaký certifikát a zatím to lépe neumíme.

Příkladem buď problém z povídání o bludišti. Říká se mu *Hamiltonovská kružnice*.

Název problému: Hamiltonovská kružnice

Vstup: Neorientovaný graf.

Problém: Existuje v zadaném grafu kružnice procházející všemi vrcholy právě jednou?

Certifikát: Posloupnost vrcholů hamiltonovské kružnice.

Ověření v polynomiálním čase s certifikátem: Projdeme postupně vrcholy a ověříme, že jsou opravdu zapojeny do kružnice a kružnice je správné délky. Vrátime NE, pokud tomu tak není.

Zatím nikdo nepřišel s řešením, které by nepoužívalo vůbec žádný certifikát. Dokonce zatím nikdo nenalezl problém, který by byl v NP, ale bez certifikátu už jej nelze řešit v polynomiálním čase. Kdyby takový neexistoval, třídy P a NP by se rovnaly. To je jádro otevřeného problému P vs. NP.

Převoditelnost a NP-úplnost

Když řešíme nějakou algoritmickou úlohu, obvykle přijdeme na nějaké přímé řešení využívající základních technik (prohledávání do šířky, dynamické programování, zametání přímkou). Vzácně se může i stát, že v problému rozpoznáme problém jiný – občas lze geometrický problém převést na třídění čísel nebo umíme popsat situaci nějakým vhodným grafem.

Ukazuje se, že se ve třídě NP často vyplatí problémy převádět, neboť přímá řešení neznáme. Dokonce tak můžeme i zjistit, do které z probíraných tříd problém patří.

Převodem budeme rozumět polynomiální algoritmus, který upraví vstup jednoho problému na vstup jiného problému. Musí navíc problémy převést tak, aby správná odpověď (ANO nebo NE) na vstup prvního problému byla tatáž, jako správná odpověď na vstup druhého problému.

Jednoduchým převodem je úprava problému *Existuje cesta z bludiště ze zadaného políčka délky d ?* na *Existuje cesta v grafu délky c začínající v zadaném vrcholu?*

Do výstupního grafu za každou křižovátku dáme vrchol, za každou cestu mezi křižovatkami hranu a ke hraně si poznamenejme, jak dlouhá byla. Hodnotu c pak můžeme nechat stejně velkou, jako d .

Pokud najdu správnou cestu v tomto grafu, pak nutně podobná cesta je i v bludišti, a pokud cesta v grafu není, pak není ani v bludišti. Převod je tedy korektní.

Nadefinujme si nyní pojem, který nám bude sloužit jako zkratka za to, že problém je ve třídě NP a je alespoň tak těžký jako ostatní problémy v NP. Nemůžeme jen tak ledabyle říci „je v NP a není v P“, protože to nevíme. To je právě ta slavná otázka.

Uděláme tedy krok stranou – budeme říkat, že problém je *NP-úplný*, pokud onen problém je v NP a zároveň jdou všechny ostatní problémy v NP převést na tento problém.

Všechny problémy v NP na něj jdou převést? Pokud tuto definici vidíte poprvé, asi to působí dost zvláště – je těžké si představit, že všechny grafové, geometrické, počítací problémy, o kterých víte, že jsou v P (a tedy i v NP) jdou převést na nějaký NP-úplný superproblém.

Ale je to správně, ba co víc, Cookova věta říká, že existuje alespoň jeden takový problém. (Samotná definice NP-úplného problému nezaručuje, že takový problém vůbec existuje.)

Ukazuje se však, že není sám, jsou jich stovky. Dokazovat existenci dalších NP-úplných problémů je však o dost lehčí, než dokázat Cookovu větu! Stačí totiž jen najít následující dva kroky:

- Dokázat, že problém je v NP – najít certifikát a polynomiální algoritmus, co jej využívá.

- Převést zadání libovolného NP-úplného problému na zadání našeho problému tak, že náš algoritmus vlastně vyřeší onen NP-úplný problém.

To postačí, protože pak libovolný jiný problém v NP nejprve převedeme na zvolený NP-úplný problém a pak pustíme námi vymyšlený převod. Zřetězení dvou polynomiálních algoritmů (převodů) je opět polynomiální algoritmus, takže podmínka převoditelnosti je splněna.

Ukážeme si důkaz NP-úplnosti jednoho problému na příkladu, pokud nám uvěříte, že již probíraný problém *Hamiltonovská kružnice* je NP-úplný. Nejprve zadefinueme jiný problém:

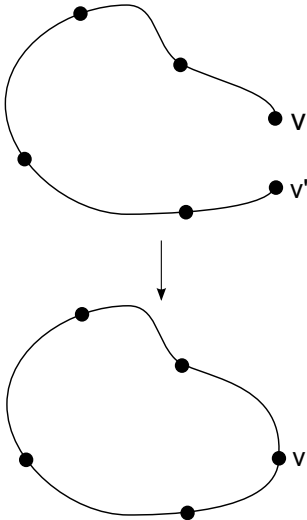
Název problému: Hamiltonovská cesta.

Vstup: Neorientovaný graf, dva speciální vrcholy x a y .

Problém: Existuje cesta z x do y (posloupnost vrcholů, ve které se žádné dva neopakují), která prochází každým vrcholem právě jednou?

Certifikát: Posloupnost vrcholů tvořící správnou cestu.

Řešení v NP: Projdeme cestu z certifikátu a ověříme, že vrcholy jdou za sebou, je jich správný počet a žádný jsme nevynechali.



Důkaz NP-úplnosti: Převedeme předchozí problém (hamiltonovskou kružnici) na hledání hamiltonovské cesty. Uvažme graf G , ve kterém chceme najít hamiltonovskou kružnici.

Vyberme si libovolný vrchol v a vytvořme vrchol v' , který bude kopií vrcholu v – do grafu přidáme hranu mezi u a v' , pokud už v něm je hrana mezi u a v .

Na upravený graf zavoláme řešení problému *Hamiltonovská cesta* mezi vrcholy v a v' . Pokud taková cesta existuje, tak nutně v původním grafu G existuje hamiltonovská kružnice.

Cesta z vrcholu v' přesně odpovídá pokračování kružnice poté, co přijde do vrcholu v .

Pseudopolynomiální algoritmy

Znáte problém batohu? Jeho varianty jsou oblíbené na programovacích soutěžích. Zadat se může třeba takto: mějme na vstupu seznam n dvojic kladných přirozených čísel, kde každá dvojice označuje váhu a cenu nějakého předmětu. Nakonec dostaneme na vstupu ještě číslo b , které udává nosnost našeho batohu.

Otázka zní: Jaký je nejčennější možný náklad, který přesto nepřesahuje váhový limit batohu?

Možná víte, že úloha jde řešit dynamickým programováním – vytvořím si pole `podbatoh[]` od 1 do b , kde `podbatoh[i]` je maximální hodnota, kterou bych si odnesl

v batohu o nosnosti i . Postupně od první věci do poslední pak projdu celé pole `podbatoh[]` „zprava doleva“ od b do 1 a zkusím, jestli je výhodnější do batohu vložit novou věc a volné místo doplnit starými (optimální volné místo pro předchozí věci máme napočítané), nebo si nechat jen ty staré. Tuto hodnotu pak zapíšeme jako aktuální pro váhu i na místo `podbatoh[i]`.

Po n průchodech tohoto pole dostaneme řešení pro všechny věci dohromady na políčku `podbatoh[b]`. Celková složitost je $\mathcal{O}(nb)$, to je polynom, algoritmus je tedy polynomiální.

Světě div se, toto řešení je ve skutečnosti exponenciální. Kde jsme v řešení udělali chybu? Nikde – naše složitost závisela na b , ovšem když se podíváme do vstupních dat, tak pokud jsou zapsána v binárním (nebo ternárním a vyšším) tvaru, tak zápis čísla b byl veliký $\mathcal{O}(\log_2 b)$, ale naše složitost závisela na $b = 2^{\log_2 b}$, tedy exponenciálně vůči velikosti vstupu.

Problém batohu, respektive jeho rozhodovací verze, je dokonce NP-úplný problém.

Algoritmům, které řeší nějakou úlohu a jsou polynomiální oproti *hodnotě* čísel na vstupu, ale exponenciální ve *velikosti zápisu* těchto čísel, říkáme *pseudopolynomiální algoritmy*. Některé další NP-úplné problémy mají pseudopolynomiální řešení (jako například *Dva loupežníci* níže), ale dá se dokázat, že na některé jiné problémy pseudopolynomiální algoritmus neexistuje (pokud $P \neq NP$).

Mimočodem: pokud bychom na vstupu zapisovali čísla v unárním zápisu, každý pseudopolynomiální problém by ležel v P.

Poznámky na závěr

Otázku „Je třída P rovna NP?“ se již snažilo rozlousknout mnoho matematiků a inamatiků. Tato teorie přinesla spoustu zajímavých výsledků, například už se podařilo dokázat, že některými technikami tuto domněnku nelze nikdy dokázat, ani vyvrátit.

Kdyby platilo $P = NP$, pak by mnoho lidí zajásalo – mnoho přirozených problémů, které nastávají i v reálném životě, by najednou byla řešitelná rychle. Navíc by krachlo dosavadní šifrování a bylo by možné najít rychle důkaz ke každému pravdivému tvrzení výrokové logiky.

Tato rovnost by se dala hypoteticky ukázat velice snadno – stačilo by najít jeden polynomiální algoritmus pro libovolný NP-úplný problém! Většina inamatiků studujících složitost se však domnívá, že se třídy nerovnájí.

To ale neznamená, že si to nemáte zkusit dokázat! Naopak, bojovat s NP-úplnými problémy je užitečné i v reálném světě – například jde mnohdy vymyslet dobrá aproximace NP-úplného problému.

Například nenajdeme hamiltonovskou kružnici v polynomiálním čase, ale nalezneme nějakou relativně dlouhou kružnici, která nám v praxi může stačit, pokud podle ní třeba chceme vést náročný cyklistický závod.

O aproximacích už toho bylo napsáno mnoho, viz například [ADS2]. O NP-složitosti můžete něco najít tamtéž, nebo zkuste [Algo].

Existují i problémy, které jsou mimo P i NP, a dokonce existuje spousta různých dalších tříd problémů. Je jich celá zoologická zahrada – můžete ji najít na internetu.

Seznam NP-úplných problémů

Sedíte-li nad zatím nevyřešenou úlohou, kterou jste našli jinde než v KSP, pak se klidně může stát, že bude NP-úplná. Abyste mohli mezi NP-úplnými úlohami převádět, tak je dobré znát jich aspoň hrstku, podle toho, je-li problém grafový, rovnicový a tak dále.

V následujícím seznamu najdete několik úloh, které jsou zaručeně NP-úplné. Převody se nám sem sice nevešly, ale mnoho z nich (ne-li všechny) zvládnete vymyslet sami – zkuste si to!

Název problému: Hamiltonovská kružnice

Vstup: Neorientovaný graf.

Problém: Existuje v zadaném grafu kružnice procházející všemi vrcholy právě jednou?

Název problému: Hamiltonovská cesta.

Vstup: Neorientovaný graf, dva speciální vrcholy x a y .

Problém: Existuje cesta z x do y (posloupnost vrcholů, ve které se žádné dva nepakují), která prochází každým vrcholem právě jednou?

Název problému: Splnitelnost

Vstup: Logická formule. Tu tvoří proměnné a logické spojky negace \neg , konjunkce \wedge a disjunkce \vee . Například

$$(x \wedge (\neg y)) \vee z.$$

Problém: Můžeme proměnným přiřadit hodnoty 0 nebo 1 tak, že výsledná vyhodnocená formule má hodnotu 1?

Název problému: Součet podmnožiny

Vstup: Seznam nezáporných celých čísel, speciální číslo k .

Problém: Existuje podmnožina čísel, jejíž součet je přesně k ?

Název problému: Batoh

Vstup: Seznam dvojic nezáporných čísel, kde dvojice označuje hodnotu a váhu předmětu. Přirozené číslo b – nosnost batohu, přirozené číslo k .

Problém: Umíme vložit do batohu předměty o hodnotě alespoň k , aniž bychom překročili limit váhy b ?

Název problému: Dva loupežníci

Vstup: Seznam nezáporných celých čísel.

Problém: Existuje rozdělení seznamu na dvě hromádky tak, že každé číslo bude v právě jedné hromádce a v každé hromádce bude stejný součet čísel?

Název problému: Klika

Vstup: Neorientovaný graf, číslo k .

Problém: Existuje v grafu úplný podgraf o velikosti k , tedy k vrcholů takových, že mezi každými dvěma z nich vede hrana?

Název problému: Nezávislá množina

Vstup: Neorientovaný graf, číslo k .

Problém: Existuje v grafu prázdný podgraf o velikosti k , tedy k vrcholů takových, že žádné dva z nich nejsou spojeny hranou?

Název problému: Trojbarvnost grafu

Vstup: Neorientovaný graf.

Problém: Lze vrcholy tohoto grafu obarvit třemi barvami tak, že každá hrana sousedí s vrcholy dvou různých barev?

Název problému: Rozparcelování roviny

Vstup: Seznam bodů v rovině, kde každý má navíc přiřazenu jednu z b barev, číslo k .

Problém: Umíme rozdělit rovinu pomocí k přímků tak, že v každé oblasti jsou jen body stejné barvy?

Název problému: 3D párování

Vstup: Seznam mužů, žen a zvířátek, následovaný seznamem kompatibilních trojic tvaru {muž, žena, zvířátko}. Tyto trojice říkají, která trojice muž, žena a zvířátko by se dohromady snesla.

Problém: Můžeme všechny muže, ženy a zvířátka z prvního seznamu rozdělit do trojic tak, že každá trojice je kompatibilní a každá bytost je právě v jedné trojici?

Martin Böhm

Úloha 23-5-5: NP-úplný metr

Následující problém si pojmenujeme Metr a vaším úkolem je dokázat, že je NP-úplný. Jistě znáte skládací metry. Mají typicky pět článků po dvaceti centimetrech. Mějme metr nepravidelný, jehož jednotlivé články jsou různě dlouhé. Tyto délky dostaneme v pořadí na vstupu, stejně tak délku pouzdra, do kterého bychom chtěli metr uložit. Podaří se nám to? Pro články délek 6, 3, 3 a pouzdro délky 6 odpověď jistě zní ANO, pro vstup 6, 3, 4 a stejně dlouhé pouzdro to už ale NEPŮJDE.

Řešení úloh

Třídění

18-2-4: Stavbyvedoucí

Ah, bezdůvodně čekáš, čtenáři drahý, dábelské figle, i když na obyčejný počátek prachobyčejného řešení stavbyvedoucího strastí tato věta vyznívá značně zvláštně. Demonstruje totiž jeden velice důležitý fakt: setřídít slova podle třetího písmenka, pak podle druhého (stabilně, tj. se zachováním původního pořadí, pokud jsou druhá písmenka nějakých dvou slov stejná) a nakonec podle prvního, dopadne stejně jako nejdříve je setřídít podle prvního, pak zvlášť každou skupinu začínající stejným písmenem setřídít podle druhého a skupinky mající stejné i druhé písmeno ještě podle třetího.

Totéž samozřejmě platí i pro třídění tabulek podle sloupečků podle Potrhlíkových požadavků: ponejprv řádky třídíme podle sloupečku daného posledním požadavkem, skupiny se stejnou hodnotou tohoto sloupečku pak podle předchozího požadavku a tak dále, až se propracujeme k začátku seznamu požadavků nebo skončíme se skupinkami o jednom řádku.

Z toho ovšem ihned plyne, že zabývat se tímto sloupečkem vícekrát je zhola zbytečné: pokud po prvním porovnání podle nějakého sloupečku zůstaly nějaké dva řádky v téže skupině, měly v příslušném sloupečku stejnou hodnotu, a proto nám je další porovnání musí ponechat ve stejném pořadí.

Pokud se tedy nějaký sloupeček v posloupnosti požadavků vyskytuje vícekrát, stačí ponechat jen jeho poslední výskyt. Tím určitě dostaneme posloupnost ekvivalentní se zadanou (takové budeme říkat *řešení*). Zbývá nám ještě dokázat, že žádné kratší řešení nemůže existovat.

Kdyby existovalo, vezmeme si nejkratší takové. Určitě se v něm nebudou opakovat sloupečky (jinak by se naším algoritmem dalo ještě zkrátit) a ani v něm nebude žádný sloupeček navíc (to by byl sloupeček, podle kterého se netřídilo ani v zadané posloupnosti, takže bychom ho mohli škrtnout). Tudíž v ní musí nějaký sloupeček z našeho řešení chybět.

Pak stačí vytvořit dva řádky, které se budou lišit pouze v chybějícím sloupečku, a takové musí obě řešení setřídít různě, což je evidentní podvod, totiž spor. Podobně můžeme dokázat i to, že naše řešení je nejen nejkratší, ale také jediné s touto délkou: jiné by se nutně lišilo pořadím nějakých dvou sloupečků i, j a mohli bychom sestrojít dva řádky podle i uspořádané opačně než podle j a jinak stejné a opět dojít ke sporu.

Zbývá si rozmyslet, jak naše řešení naprogramovat. Znalci Unixového shellu mohou navrhnout třeba toto:

```
n1 -s:|tac|sort -t: -suk2|sort -n|cut -d: -f2
```

My si předvedeme jednoduchý (a přiznejme, že daleko efektivnější) program v Pascalu. Bude číst vstupní posloupnost po jednotlivých prvcích a ve frontě si udržovat

řešení pro zatím přečtenou část vstupu. Přejde-li požadavek na třídění podle nějakého sloupečku, přidáme tento sloupeček na konec fronty a pokud se již ve frontě vyskytoval, předchozí výskyt odstraníme.

Abychom to zvládli rychle, budeme si frontu pamatovat jako obousměrný spojový seznam, tj. pro každý sloupeček si uložíme jeho předchůdce a následníka. Tak nám celá fronta zabere paměť lineární s počtem sloupečků a na každou operaci si vystačíme s konstantním časem, celkově tedy s časem $\mathcal{O}(M + N)$ (požadavků + sloupečků).

Tomáš Gavenčiak a Martin Mareš

23-3-5: Rozházené EWD

Úkolem bylo setřídít zadaný jednosměrný spojový seznam co nejrychleji, ale v konstantní paměti, což znamená jen s předem daným počtem proměnných, bez rekurze a dalších pomocných polí, tedy pouze přepojováním původního spojového seznamu.

Určitě bylo dobrým nápadem podívat se do naší (tradiční české) kuchařky o třídění. A co s tak malou pamětí? Bublínkové třídění (BubbleSort) bude zcela jistě fungovat, protože v průběhu algoritmu prohazujeme jen dva sousední prvky, což lze udělat jednoduše.

Bublínkové třídění má navíc pěknou vlastnost, že třídění již setříděných dat trvá pouze $\mathcal{O}(N)$. Jenže nejhůře a dokonce i průměrně vyjde asymptotická složitost $\mathcal{O}(N^2)$. Je to nejrychlejší možný výsledek za daných podmínek, nebo ne?

Než si řekneme řešení, uveďme si dolní odhad složitosti. Jelikož stárí záznamů EWD můžeme akorát tak porovnávat (nic o nich nevíme), platí důkaz uvedený na konci kuchařky o třídění, a tedy určitě nevymyslíme algoritmus s průměrnou složitostí lepší než $\mathcal{O}(N \log N)$.

Takový algoritmus skutečně existuje. My si ukážeme, jak modifikovat třídění sléváním (MergeSort) se zachováním složitosti v nejhorším případě i v průměru $\mathcal{O}(N \log N)$, na což přišlo i několik řešitelů. Nevylučuji však, že nepůjde upravit jiný algoritmus, i když třídění haldou ani QuickSort nejspíš převést na řešení úlohy nelze.

Jak funguje takový běžný MergeSort na třídění pole? Ten si nejprve rozdělí pole na dvě půlky, ty setřídí stejným algoritmem (zavolá se na každou rekurzivně) a pak je „slije“: odebírá vždy menší z prvků na začátku obou setříděných půlek pole a vkládá je do nového pole. Podrobnější popis opět v kuchařce.

Nyní upravíme MergeSort pro potřeby naší úlohy. Jelikož nesmíme použít rekurzi, nebudeme postupovat „odshora dolů“ (postupně půlíme data na co nejmenší části), ale „odspoda nahoru“ (spoustu malých setříděných částí sléváme postupně do jedné).

V prvním kroku se podíváme na všechny dvojice sousedních prvků (každý prvek je nejvýše v jedné dvojici), porovnáme prvky dvojice a případně je prohodíme, což v případě spojového seznamu znamená přepojení odkazů. V druhém kroku sléváme vždy dvě sousední dvojice prvků do setříděné čtveřice, v třetím dvě čtveřice do osmice . . .

Obecně v k -tém kroku slijeme dvě sousední části o 2^k prvcích. Až slijeme všechny prvky do jedné setříděné posloupnosti, máme vyhráno.

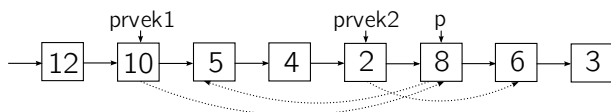
Často se může stát, že poslední slévání úsek v k -tém kroku nemusí mít 2^k prvků, ale to vůbec nevádí (jeden slévání úsek bude menší). Podobně lichý počet slévání úseků (nemůžeme je spárovat do dvojic) ošetříme prostým ignorováním posledního úseku. V nějakém pozdějším kroku musí být tento úsek slit se zbytkem, třeba pro $2^n + 1$ prvků se bude poslední prvek slévat až v posledním kroku.

Nyní pojďme na implementaci slévání dvou setříděných úseků ve spojovém seznamu (ne nutně stejné délky) s konstantní pomocnou pamětí. Budeme si pamatovat odkaz na prvek před prvním úsekem (tedy poslední prvek již slité části) v proměnné `prvek1` a odkaz na prvek před druhým úsekem v proměnné `prvek2`.

Na začátku slévání dvou úseků nejprve posuneme odkaz `prvek2` o délku prvního úseku za odkaz `prvek1`. Abychom mohli kontrolovat, jestli v nějakém úseku nedošly prvky, vytvoříme si dvě proměnné `delka1` a `delka2`, v nichž budou počty zbyvajících prvků v úsecích.

Pak postupně bereme prvky ze začátku obou úseků (následníky prvku `prvek1` a `prvek2`) a menší z nich přepojíme za prvek `prvek1`. Je-li to prvek z prvního seznamu, stačí posunout odkaz `prvek1` o jeden prvek dopředu, jinak je to následník `prvek2` (označme ho `p`), který přepojíme za `prvek1` takto: následníkem `p` bude následník `prvek1`, následníkem `prvek1` bude `p`, následníkem `prvek2` bude původní následník `p`.

Jestli vás předchozí odstavec zmátl, vůbec se nedivím a raději předkládám obrázek (tečkované šipky ukazují přepojení prvku `p`):



Je vidět, že potřebujeme jen konstantně mnoho pomocné paměti. Co se týče časové složitosti, bude pro jakákoliv data $\mathcal{O}(n \log n)$, kde n je počet prvků. V k -tém kroku totiž sléváme úseky o 2^k prvcích, a bude-li $2^k > n/2$, získáme po tomto kroku celý setříděný spojový seznam. Odtud zlogaritmováním dostaneme, že stačí $\log_2 n$ kroků, přičemž v každém provedeme $\mathcal{O}(n)$ operací.

Pavel Veselý

Binární vyhledávání

23-1-3: Jedna maticová

První řešení spočívá v prohledání celé matice řádek po řádku a kontrole každého prvku. Takové řešení samozřejmě funguje, dokonce funguje i pro obecné matice. A to je právě kámen úrazu.

Protože toto řešení nevyužívá vlastností matice, musí se podívat na každý prvek. Jeho složitost je tedy $\mathcal{O}(n \cdot m)$ pro matici velikosti $n \times m$. To ani zdaleka není to, co bychom chtěli.

Někteří si uvědomili, že když je posloupnost čísel v řádku ostře rostoucí, dalo by se využít binární vyhledávání. A tak jde zlepšit složitost z $\mathcal{O}(n \cdot m)$ na $\mathcal{O}(n \log m)$. Ale věřte tomu nebo ne, ani to nám nestačí.

Když nestačí použít na každém řádku binární vyhledávání, co ještě provést? Správné řešení používá binární vyhledávání na hlavní diagonále matice (tak se říká úhlopříčce vedoucí doprava dolů). Před uvedením algoritmu si musíme uvědomit, že platí dvě důležité věci:

- Pokud je v matici A na indexech i, j (označíme jako $A_{i,j}$) prvek, jehož hodnota je menší než $i + j$ ($A_{i,j} < i + j$), víme z uspořádání prvků v řádcích a sloupcích, že jsou menší i všechny prvky v matici, jejichž souřadnice jsou menší než i a j ($\forall k \leq i, l \leq j : A_{k,l} < k + l$).
 $A_{i,j}$ je alespoň o jedna menší než $i + j$, tedy i např. $A_{i-1,j}$ musí být alespoň o jedna menší než $A_{i,j}$, což znamená, že je alespoň o jedna menší než $i - 1 + j$. A takto tranzitivně dále.
- Pokud platí $A_{i,j} > i + j$, pak $\forall k \geq i, l \geq j : A_{k,l} > k + l$. Opět platí obdobně, $A_{i,j}$ je alespoň o jedna větší než $i + j$, takže i všechny následující prvky musí být alespoň o jedna vychýleny.

Z těchto dvou pozorování plyne, že pokud se podíváme na prvek uprostřed matice, tak mohou nastat tři možnosti. Mohli jsme narazit na správný prvek. To znamená, že můžeme skončit. Nebo je nalezený prvek větší než součet jeho souřadnic, pak můžeme zapomenout pravou dolní čtvrtinu matice, případně je prvek menší a zapomeneme levou horní čtvrtinu matice.

Takže budeme provádět binární vyhledávání na hlavní diagonále, buď najdeme správné řešení, nebo nám nakonec zůstane jen pravá horní a levá dolní čtvrtina matice. Na ty zavoláme rekurzivně stejný algoritmus. Právě tento způsob je použit ve vzorovém kódu.

Toto řešení nám přišlo několikrát, ovšem pouze jednou u něj byla uvedena správná časová složitost. Pojďme si ji tedy rozebrat detailně. Čas potřebný pro nalezení řešení je definován rekurzivně: $T(n^2) = 2T(n^2/4) + \log_2 n$ (pro jednoduchost předpokládáme čtvercovou matici).

Každý správný programátor je hlavně hrozný lenoch, využijeme tedy kuchařkovou metodu pro počítání složitosti rekurzivních algoritmů. Ta se jmenuje Master Theorem a řeší rekurzivní vztahy ve tvaru $T(N) = aT(N/b) + f(N)$, kde $a \geq 1, b > 1$. Dále tvrdí, že pokud $f(N) = \mathcal{O}(N^{\log_b(a) - \varepsilon})$ pro nějaké $\varepsilon > 0$, tak $T(N) = \Theta(N^{\log_b a})$.

Pro naši rekurenci tohle všechno platí:

$$a = 2, b = 4, \log_2 n = \mathcal{O}(n^{2 \log_4 2 - \varepsilon}),$$

takže výsledná složitost je $\Theta(n)$. Prostorová složitost je logaritmická, protože používáme zásobník.

Existuje i jednodušší řešení, které také vede k cíli. Pro něj si stačí uvědomit, že pokud se podíváme na prvek v levém dolním rohu, tak buď jsme našli správné řešení, nebo je větší než součet souřadnic, pak můžeme zahodit celý poslední řádek, nebo je menší

než součet souřadnic a můžeme zahodit celý první sloupec. Nakonec se posuneme buď nahoru nebo doprava, podle toho, čeho jsme se zbavili, a pokračujeme stejně.

Takto se v každém kroku zbavíme buď celého sloupce, nebo řádku. V nejhorším případě tedy provedeme $\mathcal{O}(n + m)$ operací. Prostorová složitost je zde konstantní.

Pokud bychom chtěli najít všechny prvky matice, které odpovídají zadání, tak je snadné uvedené dva algoritmy upravit, víme totiž, že pokud najdeme jedno řešení, budou s ním další sousedit, nebo budou v zatím neprozkoumané části matice.

David Marek a Karel Tesař

22-5-2: Stráže údolí

V této úloze sme mali zadané body na priamke, vedeli sme medzeru medzi každými dvoma susednými a chceli sme odstrániť maximálne K z nich, aby sme maximalizovali najkratšiu medzeru medzi tými bodmi, ktoré zostanú.

Táto úloha rovnako ako mnoho iných úloh má jednoduché, rýchle, ale pritom nesprávne greedy riešenie, ktoré je založené na postupnom odstraňovaní bodov susediacich s najkratšou medzerou (skúste si nájsť protipríklad).

Jednoduché korektné riešenie vieme naprogramovať pomocou dynamického programovania, kde stav výpočtu je dvojica (n, k) a pre každú dvojicu chceme spočítať optimálne riešenie, ak sme spracovali prvých n bodov a vyhodili sme práve k z nich. Takáto úvaha vedie na riešenie so zložitou $\mathcal{O}(N^2 K)$, ale stále má ďaleko od vzorového riešenia.

Naše vzorové riešenie využíva myšlienku, ktorá sa používa vo veľa problémoch, kde spočítať samotné riešenie problému je pomerne zložité, zato však overiť, či existuje riešenie s požadovanou vlastnosťou, je pomerne jednoduché.

V našom príklade vieme ľahko overiť, či existuje riešenie, ktoré odstráni maximálne K bodov z daných a má minimálnu vzdialenosť aspoň takú ako pevne dané M . Zodpovedanie tejto otázky vieme previesť na iný známy problém „plánovania intervalov“: Máme zadaných N intervalov v čase, pričom každý začína v čase a_i a má dĺžku b_i . Pričom z týchto intervalov chceme vybrať maximálny počet tak, že žiadne dva vybrané intervaly sa neprekrývajú.

Prevod je nasledovný: všetky čísla a_i sú rovné pozíciám bodov na priamke a všetky b_i sú rovné M . Ak nájdeme riešenie tohto problému, potom sme našli maximálnu množinu bodov (počiatky intervalov), ktoré sú od seba vzdialené aspoň M . Pričom keď sme našli takéto riešenie, ktoré maximalizovalo počet vybraných intervalov (bodov), potom súčasne toto riešenie minimalizuje počet intervalov (bodov), ktoré sme nevybrali. Teda po nájdení riešenia, vieme zodpovedať otázku, či existuje riešenie, ktoré má najmenšiu vzdialenosť aspoň M , podľa toho, či naše riešenie „plánovania intervalov“ nevybralo maximálne K intervalov.

Treba však vedieť riešiť samotný problém plánovania intervalov. Na tento problém je však známy jednoduchý greedy algoritmus: Na začiatku si utriedime body podľa času konca intervalu, následne začneme tieto intervaly prechádzať v tomto poradí a súčasne si budujeme riešenie (množinu vybraných intervalov) použitím jednoduché-

ho pravidla: pri prechádzaní, vždy keď môžeme práve spracovávaný interval pridať k budovanému riešeniu, tak ho tam pridáme.

Toto sa dá po usporiadaní intervalov vykonať v lineárnom čase od počtu intervalov, stačí si vždy len pamätať čas konca posledného intervalu v našom budovanom riešení. Navyiac v našom špeciálnom prípade majú všetky intervaly rovnakú dĺžku, takže stačí usporiadať intervaly podľa začiatku (na vstupe však už máme pozície utriedené a v našej úlohe nemusíme triedenie vôbec riešiť).

Teraz už vieme zodpovedať otázku, či existuje riešenie s danou minimálnou vzdialenosťou. K čomu nám to poslúži? Treba si všimnúť, že ak existuje riešenie, ktoré má minimálnu vzdialenosť aspoň M , potom existuje riešenie, ktoré má minimálnu vzdialenosť M' pre každé $M' \leq M$ (jednoducho ponecháme rovnakú množinu bodov). Inak povedané, existuje číslo M^* také, že pre všetky $M \leq M^*$ riešenie existuje a pre všetky $M > M^*$ riešenie neexistuje.

A práve číslo M^* hľadáme. Teda riešenie by sme mohli nájsť tak, že ak máme rozsah súradníc bodov z nejakého intervalu R , potom vieme postupným skúšaním existencie riešenia, ktoré má minimálnu vzdialenosť $R, R-1, R-2, \dots$, nájsť číslo R^* v čase $\mathcal{O}(RM)$. Avšak z vlastnosti hľadaného čísla M^* môžeme použiť binárne vyhľadávanie na intervale R . Keď si pre medián prehľadávaného intervalu riešenia zistíme, či existuje riešenie, pak sa na základe toho vieme rozhodnúť, v ktorej polovici prehľadávaného intervalu leží číslo M^* .

Takto vieme nájsť maximálnu minimálnu vzdialenosť medzi dvojicou bodov a body, ktoré máme odstrániť, sú počiatky nevybratých intervalov pri riešení príslušného podproblému plánovania intervalov.

Celková časová zložitosť je $\mathcal{O}(N \log R)$, kde pri binárnom vyhľadávaní na intervale dĺžky R vieme v lineárnom čase overiť existenciu riešenia. Pamäťová zložitosť je $\mathcal{O}(N)$.

Peter Ondrúška

Halda

19-2-3: Moneymaker

Asi nejjednoduchší riešenie této úlohy by se dalo popsat slovy když metoda *hrr* na ně nezabere, tak se stáhneme a zkusíme to zezadu. Až na jedno řešení využívající intervalové stromy skončili všichni řešitelé začínající od počátku kvadratickou, popř. ještě horší časovou složitostí. Nyní ale zpět k tomu, jak se úloha měla řešit.

Označme T termín nejméně spěchající zakázky. Budeme postupně, pro jednotlivé časy $t < T$, generovat pořadí plnění zakázek (označme je $A_t^t, A_{t+1}^t, \dots, A_T^t$), kterým maximalizujeme zisk v časovém úseku $\langle t; T \rangle$. Pokud zjistíme jak toto pořadí vypadá pro $t = 1$, tak máme hotovo.

Pro $t = T$ je to jednoduché. Mezi všemi zakázkami s termínem T vybereme tu, která je nejlépe placená. Nyní předpokládejme, že známe optimální pořadí zakázek od času $t + 1$ (tj. známe $A_{t+1}^{t+1}, A_{t+2}^{t+1}, \dots, A_T^{t+1}$). Pak tvrdím, že jedna z možných sekvencí zakázek s maximálním ziskem je:

- $A_i^t = A_i^{t+1}$ pro $i \geq t + 1$
- A_t^t nalezneme jako zakázku s maximální odměnou, která má termín t , nebo pozdější, a kterou jsme ještě nepoužili (tj. není mezi $\{A_i^{t+1}\}$).

Dokáže se to snadno. Pro spor předpokládejme, že známe nějaké pořadí $B_t^t, B_{t+1}^t, \dots, B_T^t$, které nám zajistí lepší zisk.

Zároveň ale víme, že odměna za úkoly $A_{t+1}^t, A_{t+2}^t, \dots, A_T^t$ je alespoň stejně velká jako za zakázky $B_{t+1}^t, B_{t+2}^t, \dots, B_T^t$ (z toho, že jsme předpokládali, že $\{A_i^{t+1}\}$ maximalizuje zisk na časovém intervalu $< t + 1; T >$). Z toho plyne, že odměna za B_t^t je větší než odměna za A_t^t . Jelikož ale A_t^t má maximální odměnu ze všech zakázek, které nebyly obsaženy v $\{A_i^{t+1}\}$, musí tedy existovat $j > t$ takové, že $A_j^{t+1} (= A_j^t) = B_t^t$, nebo jsme dostali spor. Prohodíme tedy v posloupnosti $\{B_i^t\}$ pozice úkolů B_t^t a B_j^t (to si můžeme dovolit, jelikož pak úkol B_j^t splníme dřív a zakázku B_t^t můžeme splnit až v čase j , protože se až tak pozdě vyskytovala v posloupnosti $\{A_i^{t+1}\}$, která termíny respektuje). Tím jsme zřejmě nezměníme celkovou odměnu za úkoly v $\{B_i^t\}$ a tedy celý tento odstavec můžeme použít na novou posloupnost $\{B_i^t\}$ úplně stejně.

To jsme ale ještě nic dokázali, jak si jistě čtenář všiml. Spor dostaneme, až když si uvědomíme, že výše uvedené nemůžeme opakovat donekonečna. Pokud budeme uvažovat počet úkolů, které jsou v $\{A_i^t\}$ a $\{B_i^t\}$ na stejném místě (tj. počet takových k , že $A_k^t = B_k^t$), tak v každém cyklu stoupne o 1 (úkol B_k^t se dostane na stejné místo jako je v posloupnosti $\{A_i^t\}$), tedy po několika opakováních výše uvedeného musíme někdy dostat spor.

A jak toto nejlépe implementovat? Nejdřív seřídíme úkoly dle termínu. Pak budeme odzadu generovat jednotlivé úkoly, které je třeba v daný čas t udělat. K tomu použijeme haldy. Budeme si v ní udržovat úkoly, které mají termín t či pozdější a které jsme zatím ještě nezařadili mezi zakázky, které splníme. Na začátku bude prázdná a v každém kroku do haldy přidáme všechny úkoly, které mají termín t (pozdější tam již máme z předchozích kroků) a odebereme maximum. Tím jsme skoro hotovi.

Kdybychom implementovali výše uvedené doslovně, tak čas běhu programu bude kromě velikosti vstupu záviset i na nejpozdějším termínu úkolu. Toho se ale je možno jednoduše zbavit. Pokud bude halda prázdná, můžeme rovnou posunout čas na nejbližší dřívější termín zakázky, čímž si ušetříme čas.

Seřídění pomocí rychlého třídícího algoritmu trvá $\mathcal{O}(N \log N)$. Přidání do haldy zabere $\mathcal{O}(\log N)$ a provádíme ho N -krát, tedy opět $\mathcal{O}(N \log N)$. Ještě z haldy odebíráme kořen, což uděláme také maximálně N -krát a trvá to $\mathcal{O}(\log N)$. Dohromady tedy $\mathcal{O}(N \log N)$.

V paměti máme vstup a haldy. Jejich velikost je přímo úměrná velikosti vstupu, tedy paměťová složitost je $\mathcal{O}(N)$.

Pavel Čížek

20-4-4: Skupinky pro chytré

Operace nad skupinkami přesně odpovídají operacím nad haldou a naše řešení bude opravdu vycházet z haldy. Protože nechceme hledat jedince s minimálním IQ (těch

je dost :-)) , ale s maximálním, bude zapotřebí otočit porovnávání. Větší rozdíl spočívá v tom, že operace potřebujeme provádět „nedestruktivně“ – nesmíme měnit již existující skupinky.

Na principu fungování haldy se nic nemění, ale data nebudeme moci ukládat do pole jako v kuchařce. Strom uložíme jako sadu prvků pospojovaných pointerů na levého a pravého syna. Operace nad haldou tak bude jednoduché dělat nedestruktivně: místo modifikace prvku naalokujeme nový prvek, zkopírujeme hodnoty ze starého prvku, a upravíme co je potřeba upravit. Při modifikaci syna budeme muset vždy vyrobit i nového otce a dál až ke kořeni.

Pro haldové operace potřebujeme efektivně umět pracovat s nejpravějším prvkem ve spodní hladině, a potřebujeme umět určit otce daného prvku. V poli je situace jednoduchá, požadovaný prvek je v poli tolikátý, kolik je prvků v haldě, a otec je na pozici $i/2$ (kde i je pozice syna).

Jednoduché řešení by bylo přidat do každého prvku ukazatel na jeho otce, a držet si ukazatel na nejpravější prvek ve spodní hladině; ale toto řešení použít nemůžeme, protože ukazatel na otce by nám neumožnil pracovat „nedestruktivně“.

Naštěstí je možné i -tý prvek najít pomocí bitového zápisu i , stačí postupovat od kořene a dle hodnoty bitu jít do levého nebo pravého podstromu. Pokud si do pomocného pole schováme prvky, které jsme prošli, odpadne též problém s hledáním předchůdců.

Časová složitost operací `insert` a `delete_best` je $\mathcal{O}(\log N)$, složitost `find_best` je $\mathcal{O}(1)$. Operace `find_best` nepotřebuje žádnou dodatečnou paměť, `insert` i `delete_best` naalokují $\mathcal{O}(\log N)$.

(S díky Peteru Ondrůškovi.)

Pavel Machek

Grafy

20-3-4: Orientace na mapě

Nejprve si nejspíš uvědomíme, že v acyklickém orientovaném grafu musí být alespoň jeden vrchol, do kterého nevede žádná hrana – zdroj. Z každého vrcholu (které není zdroj) můžeme cestou proti směru hran dojít do nějakého zdroje. Proto při hledání vrcholů, mezi nimiž vede nejvíce cest, můžeme předpokládat, že počáteční vrchol je zdroj – kdyby cesty vycházely z jiného vrcholu, můžeme všechny prodloužit až do nějakého zdroje. Tím se jejich počet určitě nezmenší. (Z podobného důvodu bychom také mohli hledat koncové vrcholy pouze ve stocích – vrcholech z nichž nevede žádná hrana.)

Vzápětí si uvědomíme, že zdrojů v grafu může být mnoho, takže nám tohle pozorování práci neušetří a algoritmus nezlepší, ale využít ho můžeme... Z každého zdroje tedy spočítáme cesty do jednotlivých vrcholů.

Máme-li pro nějaký vrchol v spočítat počet cest z určitého zdroje, lze to udělat tak, že sečteme počty cest do všech vrcholů, ze kterých vede hrana do v . K tomu ovšem musíme tyto počty cest znát. Proto je nutné počítat cesty do vrcholů ve správném

pořadí – v topologickém pořadí. Když máme spočítané cesty do všech vrcholů, zapamatujeme si maximální počet cest (a kam vedly) a prozkoumáme cesty z dalšího zdroje. Pak už stačí jenom vybrat zdroj, z něhož vede nejvíce cest.

A jak to všechno bude složité? Na jednotlivé průchody do hloubky potřebujeme $\mathcal{O}(N + M)$ času. Počet potřebných průchodů závisí na počtu zdrojů v grafu, může být až $\mathcal{O}(N)$. Celkem se dostáváme na časovou složitost $\mathcal{O}(N \cdot (N + M))$. Program lze implementovat s paměťovou složitostí $\mathcal{O}(N + M)$.

Tereza Klímošová

18-2-5: Krokoběh

O co tedy šlo. Zjistit, kolik nejméně hran je třeba přidat do grafu, aby se stal 2-souvislým. Hned na začátku si všimneme, že pokud najdeme v grafu komponentu, která je 2-souvislá, tak ji můžeme zkontrahovat (scvrknout) do jednoho vrcholu, aniž by se změnil počet potřebných hran. Takhle můžeme pokračovat tak dlouho, dokud se v grafu budou vyskytovat kružnice. Snadno se dá nahlédnout, že hrany tohoto zkontrahovaného grafu budou mosty v původním grafu (most není součástí žádné kružnice, proto nebude v žádném kroku zkontrahován, na druhou stranu pokud hrana není most, pak je součástí nějaké kružnice a proto bude dříve či později zkontrahována). Je také vidět, že takto zkontrahovaný graf bude les, jelikož neobsahuje kružnice. Dále budeme uvažovat tento les.

Nyní mohou nastat dvě situace. První, kterou dost řešitelů zapomnělo ve svých řešeních ošetřit, je ta, že graf byl na počátku 2-souvislý, tj. že se zkontrahoval do bodu. Pak není třeba nic přidávat.

Druhá je zbytek. Kolik bude třeba hran dodat? Jelikož v 2-souvislém grafu má každý vrchol stupeň alespoň 2 a hrana spojuje právě 2 vrcholy, musíme přidat alespoň $A + \lceil B/2 \rceil$ (*) hran, kde A je počet vrcholů stupně 0, B počet vrcholů stupně 1 (tj. listů) a zaokrouhluje se nahoru, jelikož v případě lichého B musíme tento lichý list také zapojit do nějaké kružnice, tedy tento lichý list zapojíme na libovolný vrchol.

Nyní indukci podle počtu vrcholů dokážeme, že tolik i stačí. Pro 2 vrcholy může les vypadat buď jako 2 vrcholy a pak je třeba přidat ještě 2 hrany (což splňuje vzorec (*)), nebo jsou tyto 2 vrcholy spojené hranou, a pak stačí přidat jednu (opět v souladu s (*)). Všimneme si také, že jsme v obou případech alespoň jednu hranu přidali.

Teď si uvědomíme, že libovolný les jde vyrobit z jednoho vrcholu pomocí operací:

- 1) přidej vrchol (a s ničím ho nespojuj)
- 2) přidej vrchol a spoj ho hranou s nějakým vrcholem, který už v lese je.

Nyní uvažujme, že pro N vrcholů máme již 2-souvislý les pomocí

- a) $A + B/2$ hran (pro sudé B)
- b) $A + (B - 1)/2$ hran (pro liché B ; 1 cesta nezesouvislena)

Přidejme vrchol X pomocí pravidla:

- 1) Vezmeme nějakou přidanou hranu (vedoucí $I \leftrightarrow J$), tu odstraníme a přidáme místo ní hrany $I \leftrightarrow X$ a $X \leftrightarrow J$. Tím nám stoupl počet přidaných hran do grafu o 1. Také A se zvětšilo o jedna, takže a), resp. b) stále platí.
- 2) Při připojování X mohou nastat tři situace:
 - α) Připojujeme ho hranou pod vrchol Y stupně 0. Pak ale od tohoto vrcholu vedou 2 přidané hrany. Vezmeme libovolnou z nich (nechť vede z I) a zrušíme ji. Místo ní zavedeme novou hranu $I \leftrightarrow X$. Touto operací se nám snížil počet vrcholů stupně 0 v grafu o 1, nicméně z X i Y se staly listy a proto je B o 2 větší. Tedy a) příp. b) je stále splněno.
 - β) Připojujeme ho hranou za list L . Pokud je B liché a list L je konec naší volné cesty, není třeba nic dělat a indukční předpoklady máme splněny. Jinak do tohoto listu vede nějaké přidaná hrana (z nějakého vrcholu I). Pak ale stačí zrušit hranu $I \leftrightarrow L$ a zavést novou hranu $I \leftrightarrow X$. Tím zůstane počet přidaných hran zachován. L přestal být po tomto kroku listem, nicméně objevil se nový list X , tudíž A i B zůstalo a tedy a), resp. b) stále platí.
 - γ) Připojujeme-li ho hranou za vrchol stupně alespoň 2, pak se nám zvýší B o 1, A zůstane stejné. Pokud B bylo sudé, není třeba nic dělat. Po tomto přidání bude B liché a vrchol X bude konec nezesouvislné cesty. Vzorec b) bude zřejmě platit. Nyní pokud je B liché, označíme si list na konci cesty Y . Pokud vrchol X napojujeme za vrchol, který nebyl součástí cesty, pak stačí přidat hranu $X \leftrightarrow Y$. Pokud napojujeme X na cestu, pak vezmeme libovolnou přidanou hranu $I \leftrightarrow J$, tu z grafu odstraníme a přidáme 2 nové $I \rightarrow X$ a $J \rightarrow Y$. V obou případech stoupne počet přidaných hran v do lesa o 1, což je v souladu s a).

A je to. Pro sudé B jsme dostali rovnou 2-souvislý graf, pro liché musíme ještě konec cesty napojit na libovolný vrchol, který do téhle cesty nepatří, abychom dostali 2-souvislý graf. Tím se ale dostaneme na $A + (B - 1)/2 + 1 = A + \lceil B/2 \rceil$ hran.

V zadaném grafu tedy najdeme mosty a pak v každé komponentě 2-souvislosti spočítáme, kolik mostů z ní vede. Nakonec spočteme hrany, které je třeba přidat, pomocí (*). Časová i paměťová náročnost algoritmu je $\mathcal{O}(M + N)$ (při každém průchodu do hloubky se algoritmus zřejmě na každou hranu podívá dvakrát).

Pavel Čížek

Dijkstrův algoritmus

18-3-4: Pochoutka pro prasátko

Máme šachovnici o rozměrech $X \times Y$ a sadu K pravidel, podle nichž se prasátko umí pohybovat s určitou námahou. Krátké pozorování odhalí, že každé políčko šachovnice je jeden vrchol grafu a že mezi vrcholy vede hrana právě tehdy, pokud existuje pravidlo převádějící prasátko z jednoho vrcholu na druhý. Pak je hrana samozřejmě ohodnocena příslušným množstvím námahy. A jelikož jsou hrany kladně ohodnocené a my hledáme nejkratší cestu ze startovní pozice hladovějícího pašíka na naleziště Velké Bukvice, máme úlohu jako dělanou (ve skutečnosti opravdu dělanou) pro použití kuchařkového Dijkstrova algoritmu s haldou.

Ukázalo se ale, že naprogramovat takový algoritmus nemusí být až tak jednoduché. Někteří těžce válčili s haldou, jiní v boji podlehli a zaslali jen slovní popis algoritmu.

První otázka je, jak si vyrobit onen graf zobrazující prostor lesa. Odpověď je jednoduchá. Žádný graf není třeba vyrábět, budeme pracovat přímo nad políčky lesa a hledané hrany si budeme konstruovat přímo v okamžiku, kdybychom se v Dijkstrově algoritmu dívali na sousedy aktuálně zkoumaného vrcholu. Postupně použijeme všechna možná pravidla pro pohyb z daného políčka a podíváme se, jestli jsme se nedostali mimo les.

Druhý, horší problém, vzniká u haldy. V okamžiku, kdy v Dijkstrově algoritmu najdeme lepší cestu a přepočítáváme vzdálenost nějakému vrcholu, mění se samozřejmě jeho pozice v haldě vrcholů a haldou musíme přeskládat. Jak na to?


Můžeme si někde bokem pamatovat, kde přesně se každý vrchol v haldě nachází, a pustit na něj bublání. Pak ale musíme při jakékoli operaci s haldou každému vrcholu přepočítávat tento jeho index v haldě a to je trochu zmatek.

Jiné, jednodušší řešení je haldou nijak nepředělávat, a když nějakému vrcholu přepočítáme vzdálenost, prostě jej do haldy strčit znovu. Tak se nám některé vrcholy mohou v haldě opakovat, ale my dokážeme v Dijkstrově algoritmu při vytahování minimálního prvku z haldy snadno rozeznat, jestli jej máme zpracovávat, nebo jestli je to jen zopakovaný prvek. Poznáme to podle toho, jestli už má trvalou hodnotu.

Za jednodušší řešení ale zaplatíme. Zatímco v těžším, „přepočítávacím“ řešení se každý prvek dostane do haldy nejvýš jednou, takže halda může zabírat jen tolik místa, jaký je počet vrcholů grafu, u druhého řešení se prvky mohou dostat do haldy víckrát, konkrétně halda může být veliká jako počet hran grafu.

Dijkstrův algoritmus z kuchařky trvá $\mathcal{O}((N + M) \cdot \log N)$, kde N je počet vrcholů, u nás $X \times Y$, a M počet hran, u nás XYK . Za každou operaci s haldou násobíme logaritmem velikosti haldy. Pokud tedy použijeme haldou s přepočítáváním, dostaneme časovou složitost $\mathcal{O}(XYK \cdot \log(XY))$. Jednodušší halda dá časovou složitost $\mathcal{O}((N + M) \cdot \log M) \leq \mathcal{O}((N + M) \cdot \log N^2) = \mathcal{O}((N + M) \cdot 2 \log N) = \mathcal{O}((N + M) \cdot \log N)$, takže vlastně tutéž.

Paměťová složitost je u haldy s přepočítáváním $\mathcal{O}(XY)$, protože si potřebujeme pamatovat jen les a haldou na vrcholy, ale u větší haldy až $\mathcal{O}(XYK)$.

 Jak si všiml Pepa Pihera, náš algoritmus jde ještě vylepšit. Malou úpravou dosáhneme toho, že v haldě bude vždy nejvýš K prvků, čímž stlačíme složitost na $\mathcal{O}(XYK \cdot \log K)$. (Platí $K \leq XY$, protože pokud by pravidel bylo více, na některé políčko by se dalo dostat pomocí více pravidel a my si můžeme nechat jenom to lepší z nich.)

V jednom kroku se Dijkstrův algoritmus pokouší najít vrchol s nejmenším dočasným ohodnocením. Jinak řečeno, hledá takový nezpracovaný vrchol spojený s už zpracovaným vrcholem, že součet ohodnocení zpracovaného vrcholu a hrany z něj vedoucí je co nejmenší. Navíc vrcholy zpracováváme (trvale ohodnocujeme) podle jejich vzdálenosti od výchozího místa, tedy v neklesajícím pořadí.

Zvolme si pro tento odstavec jediné pravidlo. Kromě krajních případů ho můžeme použít z každého vrcholu. Sledujme vrcholy, které pomocí tohoto pravidla dostanou trvale ohodnocení. V průběhu algoritmu je ohodnocení těchto zpracovávaných vrcholů neklesající. Protože jsme ho získali přičtením hodnoty pravidla k ohodnocení výchozímu vrcholu, je i ohodnocení vrcholů, ze kterých toto pravidlo používáme, neklesající. Toto jediné pravidlo tedy používáme na vrcholy v tom pořadí, v jakém je trvale ohodnocujeme.

Když víme, že každé pravidlo používáme na vrcholy v tom pořadí, v jakém je trvale ohodnocujeme, zapamatujeme si u každého pravidla, ze kterého vrcholu jsme ho naposledy použili. Když potom hledáme vrchol s nejnižším dočasným ohodnocením, každé pravidlo už má určený vrchol, ze kterého ho použijeme. Vybereme si tedy tu nejlepší kombinaci vrchol-pravidlo, tím jsme našli další vrchol s trvalým ohodnocením, a použité pravidlo „posuneme“ k dalšímu vrcholu. Tím myslíme, že příště ho budeme používat z vrcholu, který jsme v Dijkstroví trvale ohodnotili hned po tom vrcholu, ze kterého jsme teď pravidlo používali.

V každém kroku tedy potřebujeme najít minimum z K hodnot, toto minimum odstranit a přidat místo něj jinou hodnotu. K tomu je halda jako stvořená, všechny tyto operace zvládne v čase $\mathcal{O}(\log K)$. Navíc každou hranu zpracujeme právě jednou, čímž se dostáváme na slibovanou složitost $\mathcal{O}(XYK \log K)$.

Jana Kravalová

22-1-1: Alčina interpretace

Úlohou bylo najít cestu $P = (s = v_0, v_1, \dots, v_n = c)$, na které se nejméně mění značky $+$, $-$ na hranách.

Pro řešení je třeba modifikace algoritmu pro hledání nejkratší cesty. Chtěli bychom, aby se algoritmus ve fázi i rozlil do všech vrcholů, které jsou od počátečního vrcholu vzdáleny přesně i změn. To nám samotný algoritmus procházení do šířky nezaručí. Pokud ale v každé fázi provedeme procházení do hloubky po hranách se stejnou značkou, projdeme graf přesně tak, jak chceme.

Uděláme menší trik a rozdělíme si každý vrchol na dva, podle toho, kterou hranou jsme do něj přišli. U každého vrcholu si budeme pamatovat značku hrany, která do něj vedla, a jeho předka. Jako datová struktura pro naše prohledávání nám bude sloužit obousměrný seznam. Pokud budeme přidávat na hlavu seznamu, tak bude sloužit jako zásobník, pokud přidáme vrchol na konec seznamu, tak budeme mít frontu. Díky tomu nejprve projdeme všechny hrany se stejnou značkou a až pak teprve ty s jinou. Na začátku přidáme do fronty oba počáteční vrcholy $+s$ i $-s$. Nyní odebíráme vrcholy z hlavy seznamu, dokud není prázdný. Pro každý vrchol v se podíváme na všechny jeho sousedy, pokud jsme v nich ještě nebyli, tak jim nastavíme v jako předka, označíme je jako prošlé a zařadíme do seznamu podle toho, jestli jsme se do nich dostali po hraně stejné nebo různé značky jako do v . Ve chvíli, kdy ze seznamu vytáhneme cílový vrchol, známe nejkratší cestu k němu.

Nakonec už zbývá jen zrekonstruovat cestu. Tady nám hodně pomůže, že jsme si vrcholy rozdělili, protože tak jsou jejich předci jednoznačně určeni. Stačí jen postu-

povát od cílového vrcholu rekurzí po předcích, dokud nedorazíme do počátečního.

Vrcholů máme kvůli rozdělení dvakrát více, ale to nám složitost nepokazí. Každý z nich přidáme do seznamu jen jednou. Časová složitost našeho prohledávání bude tedy $\mathcal{O}(n + m)$. V grafových úlohách se často používá n pro počet vrcholů a m pro počet hran; tak je tomu i tentokrát. Paměťová složitost bude lineární. Kromě zadaného grafu potřebujeme v paměti jen frontu na ukládání vrcholů.

Bylo by možné použít i jiné algoritmy, např. Dijkstrův algoritmus. Ten jsme ovšem v podstatě použili, jen nepotřebujeme prioritní frontu, protože si dovedeme vrcholy uspořádat sami.

David Marek

Minimální kostra

20-1-4: Kormidlo

Zdá se, že tato úloha byla těžší, než se z počátku zdálo. Správných řešení přišlo pomálu, ty rychlé v podstatě žádné, takže Vildovi nezbylo než přibít místo kormidla jeden obdélníkový kus dřeva, který mu zbyl z opravy.

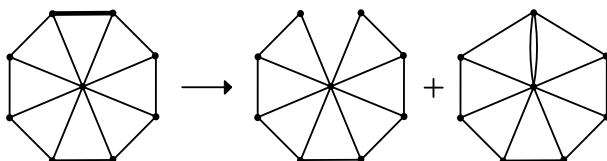
Úkolem je vlastně spočítat počet koster daného grafu. Vzorec na výpočet počtu koster úplného grafu nám nepomůže, protože kormidlo není úplný graf. Stejně tak postupy pro obecné grafy jsou trochu jako nukleární bomba na vrabce. Jde to jednodušeji.

Tedy, naší úlohou je najít počet koster určeného grafu. Úlohu si mírně zobecníme. V grafu je obvykle zakázané mít násobné hrany (více hran spojující stejnou dvojici vrcholů). My toto zakazovat nebudeme, čímž dostaneme multigraf. K čemu nám to bude dobré, si povíme později.

Máme tedy multigraf M . Vyberme si jednu multihranu (multihrana jsou všechny „normální“ hrany, které spojují stejné 2 vrcholy). Rozdělíme si množinu koster grafu M podle této multihrany na dvě (disjunktní) podmnožiny.

První podmnožina bude obsahovat všechny kostry, které neobsahují žádnou hranu z této multihrany. Velikost takové množiny je zjevně stejná, jako velikost množiny všech koster grafu M^- , který vznikne z M odebráním celé této multihrany.

Druhá podmnožina je ten zbytek, tedy všechny kostry, kde použijeme právě jednu hranu z této multihrany (více jich vést nemůže, to by nebyla kostra). Kdyby naše multihrana nebyla násobná (byla by to jen obyčejná hrana), velikost této podmnožiny by byla stejná jako počet koster na grafu $M^{\rightarrow\leftarrow}$, který z M vznikne odstraněním této multihrany a sloučením vrcholů touto multihranou spojených do jednoho (toto je proč celou dobu pracujeme s multigrafy – tady mohou vznikat multihrany).



Jak to ale bude vypadat, když námi vybraná hrana bude h -násobná? Úplně stejně, jako s jednoduchou, jen použitou hranu můžeme vybrat h způsoby, tedy výsledkem bude $h \cdot M^{\Rightarrow\Leftarrow}$.

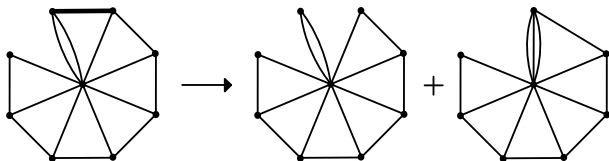
Protože jsou tyto dvě podmnožiny disjunktní a dohromady dávají celou množinu koster (nic jiného, než že tam hrana je a že tam není, se stát nemůže), můžeme velikosti těchto dvou podmnožin jednoduše sečíst.

Tímto převedeme problém počtu koster na multigrafu na dva stejné problémy, ale na menších multigrafech (čímž jsme mimochodem dokázali, že algoritmus je konečný, neboť počet koster jednovrcholového grafu je roven jedné a počet koster nesouvislého grafu je nula). Nyní stačí už jen využít toho, že vstupní graf není jen tak ledajaký, ale že je to naše pěkné kormidlo.

Podívejme se, na co se rozloží kormidlo velikosti N . Vybereme si jednu hranu na jeho obvodu. Když hranu vynecháme, vznikne něco, co by se dalo nazvat vějířem (viz obrázek). Když hranu použijeme, vznikne skoro totéž, jako kormidlo velikosti $N - 1$, jen s tím rozdílem, že jedna hrana do středu je dvojitá.

Kormidlo velikosti N s jednou k -násobnou hranou se rozloží na vějíř velikosti N s jednou $(k + 1)$ -násobnou hranou na kraji (vybereme si opět hranu sousedící s onou k -násobnou hranou) a jedno kormidlo velikosti $N - 1$ s jednou $(k + 1)$ -násobnou hranou.

Co uděláme s vějířem velikosti N a k -násobnou krajní hranou? Vybereme si vnější hranu, která sousedí s tou k -násobnou. Když ji použijeme, dostaneme vějíř velikosti $N - 1$ s jednou $k + 1$ -násobnou hranou. Když ji nepoužijeme, dostaneme vějíř velikosti $N - 1$ na násobné stopce. Protože do toho vrcholu na konci stopky vede už jen tato multihrana, musíme ji použít a počet koster takové kostry bude stejný jako počet koster vějíře velikosti N vynásobeným k (máme k způsobů, jak připojit stopkový vrchol).



Nyní, kdy toho necháme? Vějíře se jednou stáhnou až do jedné k -násobné hrany (která má k různých koster). Když nám nebude vadit myšlenka existence kormidla velikosti 1 s jednou k -násobnou hranou, všimneme si, že je to opět hrana samotná (spojující „krajní“ se „středovým“ vrcholem).

Nyní trocha počtů. Označme μ_N^k počet koster kormidla o velikosti N s jednou k -násobnou hranou. Stejně tak V_N^k budiž počet koster vějíře velikosti N s jednou k -násobnou hranou. Pomocí našeho rozkladacího pravidla si vyjádříme, že $V_N^k = V_{N-1}^{k+1} + k \cdot V_{N-1}^1$. Stejně tak $\mu_N^k = V_N^k + \mu_{N-1}^{k+1}$. Toto je jen prepis výše zmíněných rozkladů na menší podproblémy.

Kdybychom nyní iterovali přes všechna potřebná N a k (všimneme si, že k bude nejvýše $N - 1$ až na nějaké malé konstanty okolo), tak se zajisté dobereme k výsledku.

Když si budeme mezivýsledky ukládat (některé budeme potřebovat vícekrát), tak se dostaneme na časovou složitost $\mathcal{O}(N^2)$.

Mohlo by se stát, že se nám taková časová složitost nelíbí. V takovém případě se pokusíme zbavit počítání multigrafů s násobnými hranami tím, že přepíšeme vzorečky, aby používaly pouze V_N^1 a μ_N^1 . Postupně budeme rozkládat vše, co má horní index různý od 1. Tedy, $V_N^k = k \cdot V_{N-1}^1 + V_{N-1}^{k+1} = k \cdot V_{N-1}^1 + (k+1) \cdot V_{N-2}^1 + V_{N-3}^{k+2} = \dots$ Zastavíme se, až budeme mít V_1^{k+N-1} , což je, jak jsme si rozmysleli výše, $k+N-1$. Obdobně to uděláme pro μ_N^k . Doporučuji si to napřed rozepsat třeba pro $N=4$, je z toho hezky vidět, co vyjde.

Protože již k nepotřebujeme, pro zkrácení si označme V_N jako ekvivalent V_N^1 . Obdobně pro μ_N a μ_N^k . Až práci s tužkou a papírem dokončíme, vyjde nám, že $V_N = 1 \cdot V_{N-1} + 2 \cdot V_{N-2} + \dots + (N-1) \cdot V_1 + N$. Pro celá kormidla to vyjde $\mu_N = 1^2 \cdot V_{N-1} + 2^2 \cdot V_{N-2} + \dots + (N-1)^2 \cdot V_1 + N^2$.


Kdybychom nám někdo dal všechna V_1, \dots, V_{N-1} , není problém v lineárním čase spočítat μ_N sečtením všech sčítanců.

Zbývá tedy spočítat všechny vějíře, pokud možno také v lineárním čase. Kdybychom měli čísla $S_l := 1 + \sum_{i=1}^l V_i$ a $V_l = l + \sum_{i=1}^{l-1} (l-i) \cdot V_i$, jejich sečtením získáme V_{l+1} (čtenář si může ověřit sečtením). S_{l+1} získáme tak, že k S_l přičteme V_{l+1} (které již nyní máme také). Stačí doplnit startovní hodnoty. V_1 je jedna (vějířek s jedním krajním bodem je jen hrana), S_1 spočteme na 2. Všechny tedy zvládneme spočítat v $\mathcal{O}(N)$.

Nyní si už stačí jen všimnout, že každé V_l potřebujeme jen k přičtení k celkovému výsledku (samozřejmě vynásobené správným číslem). Toto přičtení můžeme udělat okamžitě, tudíž ho již příště nepotřebujeme a není třeba uchovávat pole se všemi. Tím k lineární časové složitosti získáme jako bonus konstantní paměťovou.

Program si můžeme zjednodušit dopočítáním V_0 a S_0 (na 0 a 1), čímž zjednodušíme chování cyklu a celkový součet můžeme přepočítat už po spočtení V_1 .

Michal „Vorner“ Vaner

 Každý pravověrný matematik samozřejmě věří, že na libovolný „počítací“ problém existuje chytrý vzoreček. Někdy je i hezký :) Pokud na formulky pro μ_N z našeho vzorového řešení použijete techniku zvanou metoda vytvářejících funkcí (ta je moc pěkně popsána ve starých dobrých Kapitolách z diskretní matematiky), dostanete následující pěkný vztah (časem – ono dá docela dost práce se tím vším propočítat, takže detaily si pro tentokrát odpustíme):

$$\mu_N = \alpha^N + \beta^N - 2,$$

kde α a β jsou konstanty definované takto:

$$\alpha = \frac{3 + \sqrt{5}}{2}, \quad \beta = \frac{3 - \sqrt{5}}{2}.$$

Pro počítání v programu to žádná velká výhra není, protože stěží dovedeme iracionální odmocniny z pěti reprezentovat dost přesně. Můžeme si ale pomoci drobným úskokem: Podobně jako se počítá s komplexními čísly jako s výrazy typu $a + b\sqrt{-1}$,

my budeme počítat s dvousložkovými čísly ve tvaru $a + b\sqrt{5}$, kde a a b jsou racionální. Jelikož součet, rozdíl i součin takových čísel je opět číslo v tomto tvaru, můžeme vše počítat v nich a na konci pouze vypsat první složku. (Víme totiž, že výsledek je přirozené číslo, a tak musí být druhá složka nulová. Navíc díky symetrii bude první složka u α^N stejně jako u β^N , takže stačí počítat jen jednu z nich). Ještě si vzpomeneme na trik na rychlé umocňování (viz třeba řešení úlohy 18-4-1) a vyloupne se následující program, který μ_N spočítá v čase $\mathcal{O}(\log N)$.

```
/* Dvousložková čísla a jejich násobení */
typedef struct { int i, j; } num;
num mul(num x, num y)
{ return (num){ x.i*y.i + 5*x.j*y.j, x.i*y.j + x.j*y.i }; }
int M(int n)
{
    num x={3,1}, y={1,0};    // x=2*alfa
    for (int i=n; i; i/=2)    // počítáme y=x^n
        {
            if (i%2)
                y = mul(y,x);
            x = mul(x,x);
        }
    return ((2*y.i) >> n) - 2;
}
```

Martin Mareš

20-5-4: Dračí chodbičky

Napřed jak bude algoritmus fungovat. Nejdříve bude ignorovat veškeré jeskyně s pokladem a na tom zbytku spočítá minimální kostru, například algoritmem popsáním v kuchařce. Poté vezme každou jeskyni s pokladem a připojí ji k nejbližší jeskyni bez pokladu. Jediné, na co si je třeba dát pozor, je speciální případ, pouze dvě jeskyně, obě s pokladem.

Proč to funguje? Kdyby byly dvě jeskyně s pokladem spojeny přímo a žádná další cesta z nich nevedla, pak utvoří zcela samostatnou komponentu. Tedy každá taková musí být připojena k některé bez pokladu. Je jedno, ke které, neboť zbylé jeskyně musí být navzájem propojené (a lze nahlédnout, že přes jeskyně s pokladem to nelze). Tedy vybereme si tu, ke které vede nejkratší chodba.

Zbýlý kus musí být navzájem propojený a mít minimální možný součet hran. Toto přesně počítá algoritmus minimální kostry a jeho zdůvodnění správnosti lze nalézt ve zmíněné kuchařce.

Zbývá ještě časová a paměťová složitost. Paměťová je jednoduchá, pamatujeme si každý vrchol (jeskyni) a hranu (chodbu), tedy $\mathcal{O}(N + M)$. V časové bude jednak figurovat tvorba minimální kostry, který je $\mathcal{O}(N + M \log M)$. Při připojování pokladů projdeme každý vrchol a každou hranu nejvýše jednou, takže zde máme složitost $\mathcal{O}(N + M)$. Celková tedy bude $\mathcal{O}(N + M \log M)$.

A jedna implementační poznámka na závěr. Obě fáze jsou na sobě zcela nezávislé. Proto je možné tyto dvě fáze prolínat a udělat je obě na jeden průchod seřazenými hranami, jen si u hran dáme pozor, aby maximálně jeden z konců byl s pokladem a nebyl již připojen jinam.

Michal „Vornier“ Vaner

Rozděl a panuj

19-2-5: Hluboký les

Je zajisté triviální nalézt nehlubší les zkoumáním vzdáleností všech dvojic stromů, ale tak úlohu s tak lehkým řešením bychom sem nedali, protože je to cca desetiřádkový program s ošklivou kvadratickou složitostí. Zkrátka to, čemu se říkává dřevorubecké řešení. Pojdme se raději zakoukat do hladiny křišťálové studánky, jestli nám neporadí, jak na to jít lépe (třeba od lesa):

Stromy si představme jako body v rovině, x -ová souřadnice bude odpovídat směru zleva doprava, y -ová shora dolů. Vzdálenost stromů $S_1 = (x_1, y_1)$ a $S_2 = (x_2, y_2)$ bude činit:

$$d(S_1, S_2) = \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}.$$

Kdo jste tento vzoreček ještě nepotkali, vzpomeňte si na pana Pythagora a jeho větu – chceme změřit přeponu pravouhlého trojúhelníka S_1TS_2 s pravým úhlem u vrcholu $T = (x_2, y_1)$. Místo vzdáleností budeme ale raději porovnávat jejich druhé mocniny, což jsou pro celočíselné souřadnice bodů také celá čísla. Tak si ušetříme starosti se zaokrouhlovacími chybami a program bude nadále fungovat, jelikož $x < y$ platí právě tehdy, když $x^2 < y^2$, tedy aspoň pro nezáporná čísla, což výraz pod odmocninou bezpochyby je.

Ještě si všimněme jednoho zajímavého faktu: pokud chceme do čtverce velikosti $d \times d$ umístit body tak, aby vzdálenost každých dvou byla alespoň d , vejdou se tam maximálně čtyři (třeba do vrcholů čtverce). Dokázat to můžeme například tak, že čtverec rozřežeme na čtyři menší čtverce velikosti $d/2 \times d/2$, které budou mít společné hrany, a nahlédneme, že do každého z nich můžeme umístit nejvýše jeden bod. Nejvzdálenější body v malém čtverci jsou totiž jeho protilehlé vrcholy a ty mají vzdálenost $d\sqrt{2}/2 < d$.

Jak onehdy naznačili jistí programátorští kuchaři, hodit by se mohla metoda Rozděl a panuj. Ta by se pro hledání nejbližší dvojice bodů dala použít zhruba následovně:

- Rozděl všechny body vodorovnou přímkou do dvou stejně velkých množin X_1 a X_2 .
- Rekurzivním zavoláním algoritmu najdi minimální vzdálenost d_1 dvojic bodů v X_1 a d_2 v X_2 .
- Doplní dvojice sahající přes hraniční přímkou: zajímavají nás jen takové dvojice, které mohou změnit výsledek, čili jejich vzdálenost je menší než $d = \min(d_1, d_2)$. Proto stačí uvážit body vzdálené od hraniční přímky méně než d (ostatní body mají moc daleko k hraniční přímce, natož k bodům v druhé množině). Projdeme všechny dvojice takových bodů a označíme d_3 minimum z jejich vzdáleností.

- Vrať jako výsledek $\min(d_1, d_2, d_3)$.

Pokud by první a třetí krok algoritmu běžely v lineárním čase, choval by se celý algoritmus podobně jako QuickSort s rovnoměrným dělením, který jsme ukazovali v kuchařce, a tedy by jeho časová složitost byla $\mathcal{O}(N \log N)$ a paměťová $\mathcal{O}(N)$. Stručně: Na vstup délky N spotřebujeme čas $\mathcal{O}(N)$ plus ho rozložíme na dva vstupy délky $N/2$. Pro ty potřebujeme dohromady také čas $\mathcal{O}(N)$ plus je rozdělíme na čtyři vstupy délky $N/4$, a tak dále, až se po $\log_2 N$ krocích dostaneme ke vstupům délky 1 a celkem tedy spotřebujeme čas $\mathcal{O}(N \log N)$. To je velmi lákavá představa, jen zatím poněkud efemérní, jelikož není vůbec jasné, jak první a třetí krok provést.

Rozdělování bodů: Nabízí se vybrat souřadnici rozdělovací přímky náhodně (podobně jako u QuickSortu bychom se tak dostali na průměrně rovnoměrné rozdělení) nebo si vzpomenout na lineární algoritmus pro výpočet mediánu uvedený v kuchařce. Oba přístupy ale mají společný háček: pokud většina stromů leží na jedné vodorovné přímce, vybereme nejspíš tuto přímku a body rozdělíme nerovnoměrně. Tomu by se dalo odpomoci dělením na tři části – body ležící na dělicí přímce bychom zpracovali úplně zvlášť, beztak padnou do pásu, ve kterém dvojice kontrolujeme explicitně.

Mnohem jednodušší je na začátku algoritmu setřídít všechny body podle svislé souřadnice a rozdělit je prostě na prvních $\lfloor N/2 \rfloor$ a zbylých $\lceil N/2 \rceil$. Různé body na dělicí přímce sice mohou padnout do různých polovin, ale to není nikterak na škodu, stejně je následně všechny probereme. Třídění nám časovou složitost nepokazí a rozdělování pak dokonce zvládneme v konstantním čase.

Porovnávání hraničních dvojic: Dvojic může být až kvadraticky mnoho (představte si všechny body ležící na dvou vodorovných přímkách), takže je musíme probírat šikovně. Kdybychom je měli setříděné zleva doprava, stačilo by pro každý bod B prozkoumat jen několik bodů od něj doprava – jakmile x -ová vzdálenost překročí d , nemá smysl dál hledat. Zajímají nás tedy body z X_1 ležící ve čtverečku $d \times d$ bezprostředně nad přímkou a body z X_2 ve stejně velkém čtverečku pod přímkou. A my už víme, že v každém z těchto dvou čtverečků mohou ležet nejvýše 4 zajímavé body (každé dva body ležící v téže množině jsou přeci vzdálené aspoň d a použijeme pozorování o umístování do čtverečků). To je celkem 8 bodů, navíc jedním z nich je náš bod B , čili pro každý bod B zbývá prozkoumat jen 7 následníků. To snadno stihneme v lineárním čase.

Předpokládali jsme ale, že prvky máme setříděné. To skutečně máme, jenže podle druhé souřadnice, než potřebujeme. Jak z toho ven? Jistě můžeme body na počátku setřídít podle každé souřadnice zvlášť a při rozdělování udržovat obě poloviny také setříděné oběma způsoby, ale opět bychom se dostali do potíží s mnoha body na jedné přímce. Proto se uchýlíme k drobnému úskoku: zabudujeme do naší funkce třídění sléváním: funkce na vstupu dostane body setříděné podle y a vrátí je setříděné podle x . To půjde snadno, jelikož z rekurzivních volání dostane každou polovinu správně setříděnou, a tak je jen v lineárním čase slije.

Pár poznámek na závěr:

- Sedmička je trochu přemrštěný odhad: zajímají nás pouze ty dvojice, jejichž vzdálenost je *ostře* menší než d , takže čtverce, ve kterých body mohou ležet, jsou

o maličko menší než $d \times d$ a do takových se už vejdou jen tři body (zkuste si dokázat). Správná konstanta je tedy 5.

- Také bychom mohli zkoumat na švu body z X_1 a hledat k nim do páru body z X_2 . Pro každý bod z X_1 leží kandidáti z X_2 v obdélníku $2d \times d$ a do něj se vejde nejvýše 6 bodů, což Marek Nečada pěkně dokázal rozřezáním na 6 kousků velikosti $2d/3 \times d/2$ s úhlopříčkou délky $5d/6$.
- Algoritmus, který jsme použili pro zkoumání dvojic ležících na švu, by bylo možné použít i na celou úlohu: body setřídíme podle jedné ze souřadnic a pro každý bod zkusíme do dvojice jen ty, které jsou v této souřadnici vzdálené maximálně tolik, kolik činí zatím nejmenší nalezená vzdálenost. To může být v nejhorším případě také kvadratické, ale v průměru se dostaneme na $\mathcal{O}(N \cdot \sqrt{N})$. Idea důkazu (podle Zbyňka Konečného): leží-li všechny body v obdélníku $a \times b$ a minimální vzdálenost činí d , nesmí se kruhy o poloměru $d/2$ se středy v zadaných bodech protnout, takže součet jejich obsahů $N\pi d^2/4$ smí být maximálně $(a + 2d)(b + 2d)$ (kruhy mohou na krajích z obdélníků přechuhovat až o d). Dostaneme kvadratickou nerovnici pro d a z ní po pár úpravách $d = \mathcal{O}(\min(a, b)/\sqrt{N})$.

Martin Mareš

19-5-5: Počet inverzí

Jak už to tak v životě bývá, způsobů řešení této úlohy je více. Zde si popíšeme jeden velice jednoduchý a naznačíme některé další možné. Posadte se, prosím, na svá místa, připoutejte se a během startu nekuřte.

Náš postup je založen na známém třídícím algoritmu MergeSort, neboli třídění pomocí slévání. Tento algoritmus pracuje na principu Rozděl a panuj. Tříděnou posloupnost rozdělí na dvě poloviny (tedy podúlohy menšího rozsahu), které setřídí rekurzivním použitím stejného algoritmu. Setříděné poloviny následně slijí do jedné posloupnosti.

Pro lepší pochopení našeho algoritmu si ještě raději zopakujeme průběh slévání. Řekněme, že máme dvě vzestupně setříděné posloupnosti (uložené jako pole) A a B a chceme je slít do jedné opět vzestupně setříděné posloupnosti C . Vytvoříme si indexy a , b a c , které inicializujeme tak, aby ukazovaly na první prvky jednotlivých posloupností (tj. a ukazuje na první prvek A atd.). Dokud se index a , nebo b nedostane mimo rozsah jeho posloupnosti, budeme provádět následující krok: Porovnáme prvky $A[a]$ a $B[b]$, menší z nich zkopírujeme do $C[c]$ a posuneme index v poli s menším prvkem na další prvek v posloupnosti. Rovněž posuneme c na další volné místo ve výsledné posloupnosti. Když některý z indexů (a , nebo b) dojde za konec své posloupnosti, algoritmus končí, avšak ještě je třeba dokopírovat zbyývající prvky z druhé posloupnosti (z té, která ještě nebyla zpracována celá). Např. pokud a dojde za konec A , musí se ještě zpracovat zbytek posloupnosti B .

Nyní zbývá rozmyslet, jak nám tento algoritmus pomůže při počítání inverzí. Celkový počet inverzí v posloupnosti lze spočítat jako součet počtu inverzí v obou polovinách (tj. v obou menších podproblémech) plus počet inverzí, které objevíme při slévání těchto polovin. Z principu fungování algoritmu je jasné, že nám stačí počítat pouze inverze objevené sléváním (o ostatní se postará rekurze).

Máme tedy algoritmus na slévání dvou posloupností popsany výše. Jako A si označíme první polovinu tříděné posloupnosti a jako B polovinu druhou. Pokud by bylo uspořádání správné (tj. neobsahovalo by žádné inverze), budou všechny prvky z A menší než prvky B . V každém kroku algoritmu nastává právě jedna z možností:

- $A[a] \leq B[b]$ – prvek v první posloupnosti je menší nebo roven prvku ve druhé posloupnosti, takže je vše v pořádku a žádnou inverzi jsme neobjevili.
- $A[a] > B[b]$ – prvek v první posloupnosti je větší než prvek ve druhé posloupnosti. To znamená, že $B[b]$ bude ve výsledku zařazen před všechny zbývající prvky v A , což je rozhodně porucha v uspořádání. Každý zbývající prvek v A je tím pádem v inverzi s prvkem $B[b]$, takže nám stačí přičíst k celkovému počtu inverzí počet zbývajících prvků v A .

Časová složitost tohoto algoritmu je stejná jako časová složitost MergeSortu, tzn. $\mathcal{O}(N \log N)$. Paměťová složitost je při vhodné implementaci pouze $\mathcal{O}(N)$, neboť nám stačí jedno pole na načtené prvky a jedno pomocné pole na slévání.

Závěrem bych ještě zmínil další možné způsoby řešení. Prvním způsobem je použít jiné třídící algoritmy místo MergeSortu. Problém je v tom, že ne každý algoritmus nám bude vyhovovat. Např. QuickSort použít nemůžeme, neboť přehazuje prvky mezi oběma polovinami tříděných dat, a tak nám může během třídění vytvářet inverze, které v původní posloupnosti nebyly. Druhou možností je použít vhodně upravené binární vyhledávací stromy, avšak detailnější popis by si vyžádal poměrně velké množství dalšího textu, a tak si jej dovolím vynechat.

Martin „Bobřík“ Kruliš

Dynamické programování

22-1-3: Sazba

Rozložení slov do bloku je velice pravidelné, díky tomu umíme v konstantním čase spočítat krásu jednoho řádku, máme-li načtené délky slov. Pak už si stačilo jen rozmyslet, jak počítat minimální krásu (logicky správně spíše minimální ošklivost) pro $K + 1$ slov, pokud už známe všechna minima pro K slov a méně.

Postupně budeme zkoušet, kolik se nám s aktuálním slovem vejde předcházejících slov na ten samý řádek. Pro každý takový počet slov P spočítáme krásu řádku a tu sečteme s minimální krásou pro $K - P + 1$ slov, kterou již známe. Najdeme-li minimum ze všech těchto součtů, získáme minimální krásu pro $K + 1$ slov.

Typičtější úlohu na dynamické programování aby člověk pohledal! Časová složitost pro N slov bude $\mathcal{O}(N^2)$ (pro K slov počítáme K minim, a $\sum_{K=1}^N K = (N \cdot (N+1))/2$) a paměťová $\mathcal{O}(N)$.

Martin Böhm a Martin „Bobřík“ Kruliš

23-2-1: Balíčky balíčků

Naše úloha se docela podobá problému batohu, takže by nás mohlo napadnout použít modifikovanou verzi algoritmu, kterým se řeší.

Postupně procházíme celá čísla od nuly vzhůru a pokud jsme právě na hodnotě, kam se umíme dostat, tak projdeme všechny nabídky a pro každou z nich si poznačíme, že se umíme dostat na hodnotu, která je součtem této nabídky a hodnoty, na které právě jsme. Na začátku víme jenom to, že se umíme dostat do čísla nula. Takhle postupujeme, dokud se nedostaneme do čísla, které je větší nebo rovno H , a máme řešení.

Tenhle postup sice funguje, ale dosti pomalu. K rychlejšímu algoritmu dojdeme, když si uvědomíme, co to znamená, že každou nabídku můžeme použít, kolikrát chceme – to, že kdykoliv umíme poslat x kg, tak umíme poslat i $x + kN$ kg pro jakékoliv nezáporné celé číslo k (N kg je totiž hmotnost nejmenší nabídky).

Díky tomu si můžeme pole hmotností přeuspořádat do tabulky o N sloupcích. Políčko na i -tém řádku j -tého sloupce pak představuje $(i \cdot N + j)$ kg.

K vyplňování této tabulky bychom mohli použít stejný postup jako před chvílí, ale my si ho upravíme tak, že když jsme na nějakém políčku a umíme se dostat do nějakého políčka nad ním (číslo sloupce je stejné, číslo řádku menší), tak si poznačíme, že se umíme dostat i do aktuálního políčka, ale už nemusíme zjišťovat, kam se odsud můžeme dostat s použitím různých nabídek.

To proto, že pokud se na nějaké políčko umíme dostat z aktuálního použitím nabídky x kg, tak se tam umíme dostat i ze zmíněného políčka nad ním. A to nejdříve použitím nabídky x kg a následně několikanásobným použitím nabídky N kg.

Tuto tabulku si ale nemusíme pamatovat celou. Stačí si pro každý sloupec pamatovat, který je první řádek v tomto sloupci, na který se umíme dostat.

Tento seznam sloupců pak procházíme dokola podobně, jako jsme předtím procházeli celou tabulku – jeden průchod seznamem odpovídá průchodu jedním řádkem v tabulce.

Navíc ani nemusíme procházet seznamem sloupců tolikrát, kolik řádků bychom prošli v tabulce. Jakmile se jednou umíme dostat do sloupce, který obsahuje cílové políčko, tak víme, že se umíme dostat až tam.

Pro určení výsledné kombinace balíčků si musíme pro každý sloupec zapamatovat, s použitím jakého balíčku jsme se tam dostali.

Samotnou výslednou kombinaci určíme tak, že nejdříve započítáme nabídku N kg tolikrát, kolik řádků by činil rozdíl v tabulce mezi cílovým políčkem a políčkem, kam se umíme dostat. Následně procházíme sloupce podle toho, pomocí kterého balíčku jsme se do něj dostali, dokud se nedostaneme do multého sloupce. Všechny balíčky, které jsme na této cestě použili, započítáme také a máme kýžený výsledek.

Jakou má tento algoritmus složitost? Paměťová je $\mathcal{O}(N)$ – nejvíce zabírá seznam sloupců a těch je N .

S časovou složitostí je to složitější. Procházení nabídek provádíme nejvýše jedenkrát pro každý sloupec, což nám dává $\mathcal{O}(N^2)$. Protože se ale může stát, že budeme procházet seznamem opakovaně, dokud se neumíme dostat do všech sloupců, potřebujeme zjistit, kolikrát nejvýše to uděláme.

Stačí se podívat na jedinou nabídku: $2 \cdot (N - 1)$. Pokud budeme používat jenom tuto nabídku, tak se v případě lichého N po N krocích dostaneme do každého sloupce. Došli jsme tedy až do čísla $N \cdot 2 \cdot (N - 1)$, a počet průchodů seznamem je tedy $2 \cdot (N - 1) = \mathcal{O}(N)$.

V případě sudého N se do lichých sloupců nedá dostat žádným způsobem a použitím stejné nabídky jako v předchozím případě se po $N/2$ krocích dostaneme do všech dostupných sloupců. Prošli jsme tedy seznamem opět $\mathcal{O}(N)$ -krát.

V obou případech tedy musíme projít v nejhorším případě $\mathcal{O}(N^2)$ políček. Zpětný průchod pro zjištění výsledku projde každým sloupcem nejvýše jednou a složitost nám tedy nezhorší. Celková časová složitost tedy je $\mathcal{O}(N^2 + N^2 + N) = \mathcal{O}(N^2)$.

Petr Onderka

Vyhledávací stromy

16-4-5: Obchodníci s deštěm

První věci, které si všimneme, je to, že čas potřebný na jednu odpověď (vypsání aktuálního rozdílu po přečtení jednoho čísla) by neměl být závislý na N , ale jenom na K .

Nejjednodušší řešení je po načtení další hodnoty spočítat všechny vzdálenosti dvojic posledních K vrcholů a z nich si vybrat tu nejmenší. To určitě zvládneme v $\mathcal{O}(N \cdot K^2)$.

Vylepšit to můžeme například tak, že si všimneme, že pokud bychom měli posledních K hodnot setříděných, nemusíme zkoumat $\mathcal{O}(K^2)$ vzdáleností, stačí nám spočítat vzdálenosti mezi dvěma sousedními prvky (sousedí v *setříděném* poli). Těch už je jenom $K - 1$, nicméně třídění nás stojí zase $\mathcal{O}(K \log K)$. Celkem vylepšení na $\mathcal{O}(NK \log K)$.

Další pozorování je, že po načtení jednoho čísla se pole posledních K čísel moc nezmění – určitě nemá cenu ho třídít vždy znova. Pokud máme setříděné pole posledních K čísel a načítáme další, stačí to nejstarší z pole vyhodit ($\mathcal{O}(K)$) a nové přidat ($\mathcal{O}(K)$) tak, aby pole zůstalo uspořádané. Pak stačí v jednom průchodu nad polem spočítat vzdálenosti sousedních prvků a vypsát nejmenší. Tím jsme na $\mathcal{O}(NK)$.

Vylepšovat ale jde dále. Ukážeme si dvě možná řešení se složitostí $\mathcal{O}(N \log K)$. První z nich je založeno na tomto pozorování: pokud uvažujeme o postupu s lineárním časem, tak počet dvojic, jejichž vzdálenost počítáme, se při načtení jednoho čísla mění velmi málo. Můžeme tedy mít všechny vzdálenosti sousedních prvků (sousedních v setříděném poli) v haldě.

Při načtení nového čísla ho zatřídíme do nějaké struktury (použijeme např. AVL stromy), která nám řekne jeho sousedy (většího a menšího) v setříděném poli. Pokud už je máme, z haldy odebereme vzdálenost těchto dvou sousedů a naopak do ní vložíme vzdálenost aktuálního prvku od menšího a vzdálenost aktuálního prvku od většího souseda. Při mazání čísla uděláme podobnou úpravu – zase si najdeme sousedy mazaného prvku, z haldy odebereme dvě hodnoty a dáme tam místo nich jednu (vzdálenost sousedů mazaného prvku).

Pokud použijeme ke zjišťování sousedů nějaký druh vyvážených stromů (třeba AVL :-)), můžeme hledání sousedů, vkládání a mazání provádět v čase $\mathcal{O}(\log K)$. Stejnou složitost mají i operace s haldou – a protože všeho tohoto děláme konstantní počet, máme řešení se složitostí $\mathcal{O}(N \log K)$.

To bylo jedno řešení, slíbili jsme ještě druhé: opět použijeme nějaký vyvážený binární strom. Každý jeho vrchol bude odpovídat jednomu z posledních K čísel, nicméně ve vrcholu si kromě hodnoty budeme pamatovat ještě tyto údaje:


- *min* – minimum hodnot v tomto podstromě.
- *max* – maximum hodnot v tomto podstromě.
- *delta* – nejmenší vzdálenost hodnot v tomto podstromě.

Pokud máme vrchol a známe tyto hodnoty u obou jeho synů, můžeme si spočítat i jeho hodnoty v konstantním čase:

- *min* – vezmeme minimum od levého syna.
- *max* – vezmeme maximum od pravého syna.
- *delta* – vezmeme minimum z delt levého a pravého syna, dále ze vzdálenosti hodnoty aktuálního vrcholu od maxima levého syna a ještě rozdíl hodnoty aktuálního vrcholu a minima pravého syna.

Můžeme tedy načtené hodnoty vložit do stromu, přepočítat popsané hodnoty a vypsát deltu kořene. Přepočítání hodnot můžeme provádět tak, že po vložení/smazání prvku budeme stromem procházet od vloženého/smazaného prvku směrem ke kořeni a po cestě upravovat popsané hodnoty. Pokud bude strom opravdu vyvážený, bude mít logaritmickou hloubku a tedy popsané operace budou mít složitost $\mathcal{O}(\log K)$ a celé řešení tedy $\mathcal{O}(N \log K)$.

Ve vzorovém řešení jsme schválně nepoužili AVL stromy, ty už znáte. Použili jsme tzv. BB- α stromy, které mají logaritmickou složitost pouze amortizovaně. To nám ale vůbec nevadí, protože nás zajímá složitost N operací a ne jedné.

 BB- α strom je normální binární vyhledávací strom takový, že v každém vrcholu platí podmínka, že počet vrcholů v levém a pravém podstromě se liší nanejvíc α -krát. Takový strom má vždy logaritmickou hloubku, protože podstrom nějakého stromu má nanejvýš $\alpha/(\alpha + 1)$ vrcholů – počet vrcholů v podstromu tak klesá geometrickou řadou a maximální možná výška stromu je tak $\log_{(\alpha+1)/\alpha} N$.

A jak takovou podmínku dodržet? U každého vrcholu si budeme udržovat počet vrcholů v levém a pravém podstromu. Pokud kdykoliv zjistíme, že se liší více než α -krát, celý podstrom odpojíme, vytvoříme z něj vyvážený strom a vrátíme zpátky. Takové „vybalancování“ určitě trvá lineárně vzhledem k počtu vrcholů ve vybalancovaném stromečku.

Předpokládejme nyní, že $\alpha = 2$. Kolik stojí jedno vkládání či mazání? Na to, aby se nějaké vybalancování spustilo, se musí lišit hodnoty v levém a pravém podstromu dvakrát, čili od minulého rebalancování muselo dojít k řádově tolika vkládáním a mazáním, kolik je vrcholů ve zkoumaném stromečku. Čili stačilo, aby každé vklá-

dání a mazání přispělo aktuálnímu vrcholu konstantním časem (jedním penízkem), ze kterého se pak vybalancování „uplatí“.

Každé vkládání a mazání musí přispět na rebalancování všem vrcholům, přes které projde. Těch je ale nanejvíc tolik, jaká je výška stromu – a ta je logaritmická. Čili amortizovaná složitost vkládání nebo mazání prvku je $\mathcal{O}(\log K)$ (amortizovaná znamená, že i když nevíme, jak dlouho bude jedna operace doopravdy trvat, N operací bude trvat nejdéle $\mathcal{O}(N \cdot K)$).

Milan Straka

20-5-5: Roztržitý matematik

Milí řešitelé a řešitelky, připravil jsem si tu pro vás nástin řešení, abyste si udělali alespoň hrubou představu o tom, jak to u nás chodí a na co si dávat pozor. Na úvod bych rád zdůraznil, že s papíry se to nemá tak jednoduše, jak by se mohlo na první pohled zdát. Jakmile na papír cokoli napíšete, začne žít vlastním životem a sám od sebe se přesunuje. Má tendenci se schovávat pod jiné papíry, když ho právě potřebujete, a naopak ležet na vrchu a překážet, pokud zrovna hledáte něco jiného.

Ale to jsem trochu odbočil . . . ach ano – to řešení. Někde jsem ho tu měl připravené. Kam se asi mohlo schovat? V zásadě teď může být kdekoli. Věřili byste, že jsem jednou našel svůj článek dokonce až pod automatem na kávu? Opravdu netuším, jak se tam dostal, protože automat je na chodbě poměrně daleko od mého kabinetu . . .

Ale abych se vrátil – problém, se kterým se každý den potýkám, se nazývá move-to-front transformace. Můj kolega z informatiky tvrdí, že se používá také při kompresi, ale to mi příliš nepomůže. Jádro problému spočívá v rychlém nalezení a odebrání i -tého papíru v pořadí a jeho vložení na začátek tak, aby se správně posunuly ostatní papíry.

Půjdeme-li na to přímo, nenarazíme na žádné potíže. Všechny papíry si uložíme do pole tak, že i -tý papír se nachází na indexu i . Nalezení papíru máme zadarmo v konstantním čase. Papír odebereme a všechny papíry, které jsou před ním, posuneme o jednu pozici. Tím se nám vzniklá díra zaplní a naopak vytvoříme díru na první pozici. Nyní na začátek vložíme odebraný papír a máme hotovo.

Tohle řešení má lineární časovou složitost na každou operaci (tzn. celkem $\mathcal{O}(N \cdot k)$, kde N je počet papírů a k počet operací), takže se hodí k přerovnávání několika papírků na stole mého pořádkumilovného kolegy, ale prohledání celého mého kabinetu by zabralo věčnost . . .

Dlouho jsem si s tím lámal hlavu, až mi kolega informatik poradil lepší řešení. Jak jsem se dozvěděl, klíčem jsou stromy – tím nemyslím to, co mi roste pod okny, ale binární stromy. Je vhodné použít nějakou variantu vyvážených stromů (AVL, červeno-černé, . . .), protože jinak vaše řešení rychle zdegeneruje na lineární spojový seznam. Sám se ve stromech příliš nevyznám, takže pokud vás zajímají detaily, nahlédněte do kuchařky.

V každém vrcholu u bude uložen počet prvků (označme jej $c(u)$) v podstromě, který má u jako kořen, a také číslo papíru, který je v tomto vrcholu uložen.

Takový strom postavíme jednoduše. Na začátku víme, že papíry jsou seřazeny od 1 do N . Kořen našeho stromu bude reprezentovat prostřední papír z daného intervalu. Levý a pravý podstrom pak vygenerujeme rekurzivně. Počet prvků v každém podstromě spočítáme také snadno: stačí v každém vrcholu sečíst:

$$c(\text{levého podstromu}) + c(\text{pravého podstromu}) + 1.$$

Nyní se podívejme, jak rychle nalézt, co hledáme. Řekněme, že jsme ve vrcholu u a pátráme po i -tém papíru (oproti zadání je budeme číslovat od nuly, to vyjde elegantněji). Podíváme se na počet prvků v levém podstromě $\ell = c(\text{levý syn } u)$. Pokud je $i < \ell$, víme, že se hledaný prvek nachází v levém podstromu, je-li $i = \ell$, hledaným prvkem je u sám, a konečně v posledním případě ($i > \ell$) se hledaný papír nachází v pravém stromu. Samozřejmě si musíme dát pozor, když přecházíme do pravého podstromu. Tam už nehledáme i -tý papír, ale papír s indexem $i - \ell - 1$.

Odebrání samotného papíru pak probíhá podle pravidel mazání z binárního vyhledávacího stromu (viz kuchařka). Stejně tak musíme po mazání provést vyvážení stromu, které závisí na tom, jaký typ stromu jsme použili (opět viz kuchařka). Po mazání je nezbytné ještě opravit všechny hodnoty $c(u)$ ve vrcholech, které ležely po cestě k hledanému papíru.

Odebraný papír vložíme do stromu na nejlevější pozici (tedy na první místo). Opět dodržíme pravidla pro vkládání do stromu, opravíme všechny hodnoty $c(u)$ po cestě a provedeme vyvážení.

Nakonec potřebujeme ještě vypsát konečnou permutaci dokumentů. Stačí pouze projít a vypsát náš strom v pořadí in-order (tzn. když dojde algoritmus výpisu do nějakého vrcholu, nejprve se pustí rekurzivně na levý podstrom, potom vypíše hodnotu vrcholu a pak vypíše pravý podstrom).

Časová složitost uvedeného algoritmu je $\mathcal{O}(\log N)$ na jednu operaci, protože hledání, mazání i vkládání trvá u vyváženého binárního stromu logaritmičtě dlouho. Paměťová složitost se nám přitom nezhoršila. Sice spotřebujeme několikrát víc paměti, ale asymptoticky zůstáváme stále na příjemné složitosti $\mathcal{O}(N)$.

Jeden student mi ještě tvrdil, že zná řešení v čase $\mathcal{O}(k\sqrt{N})$, ale vůbec si nejsem jistý, jak by takové řešení mělo fungovat, takže si můžete zkusit takové řešení napsat za domácí cvičení.

Náš čas na konzultaci bohužel vypršel a já se s vámi musím rozloučit. Někde jsem tu měl papír se seznamem dalších schůzek – ale kam jsem si ho sakra založil . . . ?

Martin „Bobřík“ Kruliš

Hešování

17-2-1: Prasátko programátorem

Nejprve si povšimněme, že se po nás chce pouze spočítat počet výrazů v programu, jejichž hodnota je různá – výrazy se stejnou hodnotou bychom nevyhodnocovali dvakrát, ale poprvé uložili do pomocné proměnné a podruhé použili tuto uloženou hodnotu. Upřesněme si ještě, co to znamená „mít stejnou hodnotu“. Představme si,

že bychom za proměnné ve výrazech postupně dosazovali jejich definice tak dlouho, dokud by alespoň jedna proměnná neměla svou počáteční hodnotu. Pak dva výrazy E_1 a E_2 jsou si rovny, pokud

- $E_1 = E_2 = v$, kde v je nějaká proměnná, nebo
- $E_1 = E'_1 \text{ op } E''_1$, $E_2 = E'_2 \text{ op } E''_2$, kde op je buď $+$ nebo $*$ a buď
 - E'_1 je rovno E'_2 a E''_1 je rovno E''_2 , nebo
 - E'_1 je rovno E''_2 a E''_1 je rovno E'_2 .

Samozřejmě ověřovat rovnost přímo podle této definice je nevhodné (už proto, že takto rozexpandované výrazy mohou mít i exponenciální velikost). Místo toho každému výrazu přiřadíme číslo, které bude reprezentovat jeho hodnotu – tj. dva výrazy dostanou stejné číslo právě tehdy, pokud jsou si rovny, jinak dostanou různá čísla.

První hešovací tabulka A bude jménu proměnné přiřazovat číslo hodnoty, která je aktuálně v této proměnné uložena. Ve druhé hešovací tabulce B si pak budeme pamatovat čísla hodnot výrazů, které se v programu vyskytují – klíčem této tabulky budou trojice (operátor, číslo hodnoty levého operandu, číslo hodnoty pravého operandu), a jim bude přiřazeno číslo hodnoty tohoto výrazu. Na konci stačí vypsát počet různých čísel hodnot v tabulce B , protože to bude právě počet různých hodnot výrazů v programu.

Čísla hodnot výrazů určíme takto:

- Když zpracováváme nějakou proměnnou poprvé, přiřadíme jí nové číslo hodnoty.
- Když zpracováváme přiřazení $var_1 = var_2$, pak proměnné var_1 přiřadíme stejné číslo hodnoty, jaké má proměnná var_2 .
- Když zpracováváme přiřazení $var_1 = var_2 \text{ op } var_3$, pak si nejprve zjistíme čísla hodnot v proměnných var_2 a var_3 – nechť to jsou n_2 a n_3 . Pak se podíváme do hešovací tabulky B , zda v ní je uložen výraz (op, n_2, n_3). Je-li tomu tak, pak jeho číslo hodnoty přiřadíme proměnné var_1 . Jinak tento výraz přidáme do tabulky B s novým číslem hodnoty, a toto číslo přiřadíme proměnné var_1 .

Zbývá si rozmyslet, jak ošetřit komutativitu operací. To je ale snadné – před prací s tabulkou B stačí čísla hodnot v trojici seřadit tak, aby druhé z nich bylo menší nebo rovno třetímu.

Časová složitost na operaci s tabulkou A je v průměrném případě $\mathcal{O}(k)$, kde k je délka názvu proměnné. Protože pro každý výskyt proměnné v programu provedeme právě jednu operaci s touto tabulkou, dohromady bude časová složitost pro práci s ní $\mathcal{O}(n)$, kde n je délka vstupu. Časová složitost pro práci s tabulkou B je $\mathcal{O}(1)$ na operaci, a počet operací s ní je roven počtu přiřazení ve vstupu, tj. celková časová složitost je $\mathcal{O}(n)$ – toto je složitost v průměrném případě, v nejhorším případě, kdy by docházelo ke všem možným kolizím, by časová složitost byla $\mathcal{O}(n^2)$. Paměťová složitost je zřejmě $\mathcal{O}(n)$.

Poznámka na závěr – zde popsaná metoda identifikace redundantních výpočtů se s mírnými vylepšeními skutečně používá v kompilátorech. Anglický název je Value Numbering.

19-4-3: Naskakování na vlak

Hned na začátek si neodpustím jednu poznámku: ve všech algoritmech budeme zkoumat pouze složitost, se kterou algoritmus řešení nalezne. Časovou složitost na jeho vypsání v odhadech počítat nebudeme. Vyniknou tak lépe rozdíly mezi jednotlivými algoritmy. Pokud by to někomu připadalo nefér, tak si může ke všem složitostem přičíst $\mathcal{O}(v \cdot k)$, kde v je počet navzájem různých podřetězců délky k .

Nyní již k samotné úloze. Mnoho řešitelů využilo nápovědu v zadání úlohy, a tak drtivá většina řešení využívala hešování. Ale už jenom drobná hrstka objevila, že úplně přímočaré použití kuchařky k rychlému řešení nepovede.

Základní algoritmus, který se na první pohled nabízel, byl ten, že jsme postupně brali jednotlivé podřetězce délky k , ty jsme zahešovali, a pak jsme si v nějaké tabulce (po ošetření kolizí) ukládali počet výskytů jednotlivých podřetězců. Takové řešení má v průměrném případě časovou složitost $\mathcal{O}(n \cdot k)$

Předchozí metoda měla tu nevýhodu, že jsme pro každý podřetězec museli spočítat znovu celou hešovací funkci a to zabere čas $\mathcal{O}(k)$. Co kdybychom ale našli takovou funkci, která by dokázala využít toho, že její hodnotu známe již pro předchozí podřetězec? Zde je:

$$h(i) = \sum_{j=0}^{k-1} A_i[j] \cdot P^{k-j-1}.$$

Zápis $A_i[j]$ je totéž co $A[i+j]$, tedy j -tý znak od i -tého znaku v řetězci a P je nějaké číslo, které je řádově tak velké, jako velikost abecedy.

Pokud chceme přejít na následující podřetězec provedeme tyto operace: celou sumu vynásobíme P , škrtneme první písmeno z předchozího slova a přičteme poslední písmeno z následujícího. Matematicky zapsáno:

$$\begin{aligned} P \cdot \sum_{j=0}^{k-1} (A_i[j] \cdot P^{k-j-1}) - A_i[0] \cdot P^k + A_i[k] &= \\ \sum_{j=0}^{k-1} (A_i[j] \cdot P^{k-j}) - A_i[0] \cdot P^k + A_i[k] &= \\ \sum_{j=1}^k A_i[j] \cdot P^{k-j} = \sum_{j=0}^{k-1} A_{i+1}[j] \cdot P^{k-j-1} &= h(i+1). \end{aligned}$$

Takže jsme použili konstantně mnoha kroků (nezávisle na k) a získali jsme hodnotu hešovací funkce pro řetězec, který začíná na pozici $i+1$, a to je přesně to, co jsme chtěli.

Zbývá dorešit několik technických detailů. V běžných programovacích jazycích máme proměnné omezeného rozsahu, takže pro velké k nemůžeme spočítat celou sumu. Ale

můžeme si pomoci. Stačí všechny operace provádět modulo nějaké prvočíslo. A jako ono prvočíslo můžeme použít třeba rovnou velikost hešovací tabulky.

Za poznámku stojí, že ono prvočíslo musí být opravdu prvočíslo, jinak bychom se dostali do problému. Odpověď na otázku „Proč?“ by asi nebyla nejstručnější, zájemci si ale mohou přečíst nějaké povídání o konečných tělesech.

Jak ale takové prvočíslo najít? Dle teorie čísel je pravděpodobnost toho, že libovolné přirozené číslo n je prvočíslem, je zhruba $1/\ln n$, a ověření toho, že n je prvočíslo lze základním algoritmem provést v čase $\mathcal{O}(\sqrt{n})$. Takže prvočíslo větší než nějaké n lze najít v čase $\mathcal{O}(\sqrt{n} \cdot \ln n)$, což je méně než $\mathcal{O}(n)$. Takže problémy s hledáním prvočísla mít nebudeme.

A jak to bude s paměťovou složitostí? Mnoho řešitelů si pro každou položku v hešovací tabulce pamatovalo celý podřetězec. To je ale zbytečné a paměťová složitost se tím zhorší. Stačí si přeci pamatovat pouze index, kde daný podřetězec ve vstupním řetězci začíná, což zlepšuje časovou složitost na $\mathcal{O}(n)$.

Takže jsme našli algoritmus, který v průměrném případě poběží v čase $\mathcal{O}(n+v \cdot k)$, kde v je opět počet různých podřetězců. V nejhorším případě pak v čase $\mathcal{O}(n^2 \cdot k)$.

Poznámka na úplný závěr: Pokud bychom chtěli dosáhnout času $\mathcal{O}(n+v \cdot k)$ i v nejhorším případě, mohli bychom použít sufixové stromy. Povídání o této datové struktuře a i návod, jak pomocí ní vyřešit tuto úlohu, lze nalézt v [GrafAlg].

Zbyněk Falt

Vyhledávání v textu

18-5-4: Detektýv

S důkladností takřka šerlokovskou prozkoumáme několik možných řešení, až usvědčíme to nejrychlejší. Označme si (věrní písmenkům ze zadání) N délku stopovaného řetězce, k počet podezřelých sekvencí, p_1, \dots, p_k délky těchto sekvencí a $P = p_1 + \dots + p_k$ jejich celkovou délku.

0. pokus (jak by ho vymyslel strážník Vopička): Budeme hledat každou sekvenci zvlášť, a to tak, že si po vstupu „pojedeme okénkem“ délky p_i a vždy porovnáme, jestli se okénko rovná i -té sekvenci. Kdybychom si okénko ukládali jako cyklické pole, zvládli bychom ho posunout v konstantním čase, ale stejně nás nemine čas $\mathcal{O}(p_i)$ na porovnání. Celkově trvá $\mathcal{O}(Np_1 + \dots + Np_k) = \mathcal{O}(NP)$ a navíc potřebujeme k -krát volat rewind.

1. pokus (inspektor Neverley): Damned, na hledání výskytů jednoho řetězce přeci můžeme použít algoritmus KMP z té vaší cookbook, takže jeden průchod zvládneme v času $\mathcal{O}(N + p_i)$, celkově tedy $\mathcal{O}(Nk + P)$ s k rewindy. That's it.

2. pokus (policejní rada Žák): V kuchařce je přeci i algoritmus A-McC na hledání výskytů více slov najednou. Stačí, když hlášení výskytu nahradíme připočtením jedničky k počítadlu. (Na to praktikant Hlaváček:) Dobrý plán, pane rado, ale má jedno háčisko jak na sumce: jelikož se sekvence mohou překrývat, může jich v jednom místě končit až k , takže jsme opět na $\mathcal{O}(Nk + P)$, i když tentokrát bez rewindů.

3. *pokus* (Šerlok osobně): Postavíme si vyhledávací automat jako v minulém pokusu, ale místo abychom počítali rovnou výskyty, budeme si pamatovat jen to, kolikrát jsme prošli kterým stavem, a pak z toho výskyty dopočítáme. Well, ale jak?

Pokud máme nějaký stav α (o kterém víme, že je prefixem některého z vyhledávaných slov, takže mimo jiné mezi stavy najdeme všechny sekvence stop, které počítáme) a chceme zjistit, kolikrát se slovo α v textu vyskytlo, stačí sečíst počet průchodů tímto stavem a všemi dalšími stavy, které končí na α , což jsou přesně ty, ze kterých se do α lze dostat pomocí zpětné funkce (případně zavolané vícekrát).

Stačí tedy projít automat v opačném pořadí, než ve kterém jsme vytvářeli zpětnou funkci (nejlepší bude si během konstrukce automatu toto pořadí zapamatovat, třeba v poli, v němž jsme měli uloženu frontu). Pro každý stav α pak přičteme počítadlu stavu, do něž vede z α zpětná funkce. (To se pak přičte podle další zpětné funkce atd., takže počítadlo stavu α se opravdu postupně popřičítá ke všem rozšířením stavu α .)

To vše zvládneme v čase $\mathcal{O}(P + N + P)$ (konstrukce automatu + průchod textem + dopočítání), čili $\mathcal{O}(P + N)$, a v paměti $\mathcal{O}(P + N)$, bez jediného zavolání rewindu.

It's a lemon tree, my dear Watson!

Martin Mareš

22-4-4: Ořez stromu

Označme si mateřský strom A , odvozený B . Začneme drobným pozorováním: Pokud ve stromě A najdeme posloupnost bratrských podstromů, která odpovídá podstromům synů kořene B , potvrdili jsme odvození B od A . Je-li x kořenem stromu X , jeho bratrským podstromem přirozeně rozumíme podstrom s kořenem y , kde y je bratrem x . Jaký strom zvolit jako mateřský? Zřejmě ten, který obsahuje více vrcholů. Každé „osekání“ pouze vrcholy odebírá. Pokud jich mají po „osekání“ stejně, musí být stromy identické a uvedené pozorování nadále platí.

Jak efektivně hledat posloupnost podstromů synů kořene B v A ? Uděláme cimrmanovský krok stranou, vyhneme se znovuobjevování kola a převedeme problém na hledání podřetězce v řetězci. Ano, kuchařku jste si měli přečíst ...

Zbývá najít vhodnou reprezentaci stromu pomocí řetězce. Odpověď je triviální – použijeme uzávorkované výrazy. List je reprezentovaný pomocí $()$. Každý jiný vrchol (včetně kořene) pak jako $($ *reprezentace 1. syna*, *2. syna*, ... *reprezentace posledního syna* $)$. Dva malé stromečky ze zadání této úlohy jsou pak reprezentovány například takto: $((()())())$ a $((()())())$.

Zřejmě každý strom má nějakou reprezentaci. Platí také, že je reprezentací strom jednoznačně určen? To snadno dokážete pomocí indukce. Pro list to platí a dále postupně podle složitosti vrcholu ... Zkuste si to rozmyslet. Také platí, že každý správně uzávorkovaný výraz (v běžném slova smyslu) reprezentuje nějaký strom. Pokud tedy vezmeme několik správně uzávorkovaných výrazů a „slepíme“ je za sebe do řetězce q , reprezentují posloupnost nějakých stromů Y_1, Y_2, \dots, Y_n . Pokud se navíc q vyskytuje v reprezentaci nějakého stromu X , našli jsme uvnitř X interval sousedících bratrských podstromů Y_1, \dots, Y_n .

Ať to tedy uzavřeme: Vezměme reprezentaci B a odštípněme vnější závorky (tj. získáme „slepenec“ reprezentací podstromů jeho synů), označme jako q . Pokud nalezneme q v reprezentaci stromu A , platí, že B je odvozený od A , v opačném případě nemůže být B od A odvozen.

Cože? Ještě jste si tu kuchařku nepřčetli a nevíte jak najít q v reprezentaci A ? Přece pomoci vynálezu pánů Knutha, Morrise a Pratta ... algoritmem KMP.

Čas, paměť? Trvání výroby řetězcové reprezentace stromu a její velikost jsou lineární vzhledem k počtu vrcholů stromu. KMP běží v lineárním čase se součtem délek řetězců (jehly i kupky sena :o). Časová i prostorová složitost algoritmu je tedy $O(N)$, kde N budiž součtem počtu vrcholů obou stromů.

Pepa Pihera

Rovinné grafy

18-5-5: Do vysokých kruhů

Nejprve bylo potřeba oblasti převést na objekty, se kterými umíme manipulovat rozumněji než s obecnými množinami bodů v rovině. Velmi užitečné je představit si protínající se kružnice jako graf s průsečíky a dotyky kružnic jako vrcholy. Hrany budou oblouky mezi sousedními vrcholy. Tento graf je vlastně multigraf, což je graf, ve kterém může mezi dvěma vrcholy vést více než jedna hrana a z jednoho vrcholu do toho samého může vést více než jedna smyčka. Takový graf je určitě jednoznačně zadán polohami a poloměry kružnic a je rovinný (původní rozmístění kružnic je jeho rovinné nakreslení). Bohužel se nám do něj nijak nepromítnou izolované kružnice, ty je třeba ošetřit jinak.

Nyní se nám z na první pohled neuchopitelného problému stal problém mnohem jednodušší – spočítat stěny rovinného grafu. K tomu se ideálně hodí Eulerova věta z kuchařky:

$$V + F = K + E + 1$$

Toto je vztah mezi počtem vrcholů (V), stěn (včetně té vnější) (F), komponent souvislosti (K) a hran (E). Tato věta platí pro rovinné grafy a platí i pro multigrafy, pokud si zvolím, že mezi „rovnoběžnými“ násobnými hranami jsou také stěny a že smyčka přidává jednu stěnu. Toto rozšíření přesně odpovídá naší představě toho, jak kružnice dělí rovinu na oblasti.

Stačilo by tedy spočítat počet komponent, hran a průsečíků. Víme, že na každé kružnici je stejně vrcholů a hran. Vrchol je ale sdílen mezi dvěma kružnicemi, zatímco hrana patří právě jedné. Jinak řečeno je stupeň každého vrcholu 4. Z toho plyne, že $E = 2V$. Tedy:

$$F = K + 2V + 1 - V = K + V + 1.$$

Tento vzorec nám navíc zahrne i izolované kružnice, počítáme-li je jako jednu komponentu bez průsečíků. To nám trochu zjednoduší algoritmus.

Stačí tedy spočítat počet komponent a průsečíků, obojí zvládneme v čase $O(N^2)$ průchodem do hloubky (s hledáním sousedů vyzkoušením všech) a vyzkoušením všech dvojic. Zkoušení dvojic navíc zahrneme do toho průchodu.

Toto řešení má časovou složitost $\mathcal{O}(N^2)$, paměťovou $\mathcal{O}(N)$. Existuje ještě jiné o dost složitější řešení používající *zametací přímkou* k dosažení složitosti $\mathcal{O}((N+V)\log N)$, což je lepší než naše $\mathcal{O}(N^2)$, pokud je počet průsečíků $V < N^2/\log N$, tedy pro dost „řídké“ konfigurace kružnic. Pro $V = \mathcal{O}(N^2)$ má ale časovou složitost až $\mathcal{O}(N^2 \log N)$. Paměťová složitost tohoto algoritmu je $\mathcal{O}(N)$. Jeho popis by ale byl dost komplikovaný, a proto ho neuvádím.

Tomáš Gavenčíak

Eulerovské tahy

23-2-3: Projížďka

Milý čtenář mi jistě pro jednu odpustíš, pokud si zahraji na kouzelníka a vytáhnu jednoho králíka z klobouku.

Napřed, zadání šlo chápat různými způsoby, avšak příliš neměnilo podstatu řešení. Předpokládejme tedy například, že všechny cesty jsou jednosměrky a že „z rozcestí vychází sudý počet cest“ znamená, že právě polovina tohoto sudého počtu je v příchozím a právě polovina v odchozím směru.

◊ Opravdu nám stačí taková podmínka pro orientovaný graf. V neorientovaném Σ jsme potřebovali sudý počet, protože kdykoliv jsme vešli do vrcholu, také z něj někudy musíme odejít. Stejně to funguje pro orientovaný, jen musíme přijít po vstupní hraně a odejít po výstupní. Že jde o podmínku postačující, lze nahlédnout také zcela stejně jako v neorientovaném grafu. Jediné, na co si musíme dát pozor, je, že při vypisování dostáváme hrany pozpátku.

Na grafu na vstupu (rozcestí jsou vrcholy a cesty jsou hrany) si najdeme uzavřený eulerovský tah (to již za nás vyřešila kuchařka). Nyní jej projdeme a budeme si udržovat průběžný součet prošlých hran (říkejme tomu součtu odpočatost). Rozeberme dva případy.

Jako první případ vezmeme situaci, kdy po projití celého tahu dostaneme záporné číslo. Potom je součet všech hran záporný a takový zůstane, ať je vezmeme v libovolném pořadí. Proto úloha nemá řešení.

Pokud průšvih popsany v minulém případě nenastane, vezmeme místo v tahu, kde se nachází minimum ze všech odpočatostí (místem v tahu není myšlen jen vrchol, ale i který průchod tímto vrcholem máme na mysli, neboť při různých průchodech můžeme mít různé hodnoty odpočatosti). V tomto místě v tahu začneme (jakoby jej pootočíme).

Máme tedy hezké lineární řešení (jak pamětí, tak časem), neboť již kuchařka nám ukázala, že eulerovský tah v dané složitosti zvládneme najít, a přidali jsme jen dva průchody vzniklým cyklem (jeden na průběžné počítání, druhý na výpis „pootočené“ verze).

Nyní už jen zbývá zdůvodnit, proč tento algoritmus vlastně počítá, co má. První případ je nezajímavý (neboť jsme jej již zdůvodnili výše). Dále tedy předpokládejme, že nám nastal druhý případ. Protože máme uzavřený eulerovský tah, projedeme

každou cestou právě jednou. Zbývá dokázat, že odpočetost v pootočeném tahu nikde neklesne do záporných čísel.

Předpokládejme tedy, že v místě s na tahu máme zápornou odpočetost. Minimum máme v místě m . Pokud by v původním neotočeném tahu bylo s až za m , pak by muselo být také s menším číslem než m a m by tedy nebylo minimum. Tento případ tedy nenastal.

Takže s je před m . Představme si, že jsme prošli tahem dvakrát místo jednou, tedy při druhém průchodu s jsme na nižším čísle, než při prvním průchodu m (proto nám po pootočení v s vyšlo něco záporného). Ale protože druhý průchod nezačíná od nuly, ale od něčeho nezáporného, odpočetost druhého průchodu s je alespoň tak velká, jako první. Tedy i při prvním průchodu s jsme měli nižší číslo než u m , což je opět ve sporu s výběrem minima.

Jak na to přijít? Můžeme si představit, že jsme řešení již našli a koukat na jeho vlastnosti. To, že je to uzavřený eulerovský tah, je vidět celkem jednoduše. Dále si všimneme, že vybráním jiného začátku se nám všechna čísla posouvají jen nahoru a dolů, rozdíly zůstávají stejné (s výjimkou rozpojeného konce – začátku). No a dále víme, že nejmenší číslo je 0 a to je na počátku.

Michal „Vorner“ Vaner

23-3-4: Psaní písmen

To, že jde obrázek nakreslit jedním tahem, znamená, že obsahuje uzavřený či otevřený eulerovský tah, o němž se dočtete v kuchařce. Z ní se nám bude hodit následující věta: Pokud souvislý graf obsahuje pouze vrcholy sudého stupně, je v něm možno nalézt uzavřený eulerovský tah.

Co se stane, pokud neobsahuje pouze vrcholy sudého stupně? Mezi dvojici lichých vrcholů přidáme hranu (opakujeme, dokud máme vrcholy lichého stupně), takto postupně dostaneme graf, ve kterém jsou všechny vrcholy sudého stupně, tedy obsahuje uzavřený eulerovský tah.

Nyní odebereme hrany, které jsme přidali, a tento eulerovský tah se nám rozpadne na několik hranově disjunktních tahů, které vždy začínají a končí v nějakém vrcholu lichého stupně (jeden počáteční lichý vrchol a jeden koncový lichý vrchol pro každý tah), tudíž celkový počet těchto tahů je počet lichých vrcholů děleno dvěma.

Žádný vrchol lichého stupně nemůže být uprostřed tahu, tudíž tahů nemůže být méně, než jsme našli. Stačí nám vědět, kolik takových tahů potřebujeme, není tedy potřeba je konstruovat, stačí nám určit počet lichých vrcholů (a dát si pozor na grafy bez lichých vrcholů).

Samotné řešení úlohy provedeme pro každou komponentu souvislosti samostatně: Potřebujeme pole délky n (počet vrcholů), při načítání si v něm udržujeme stupně jednotlivých vrcholů. Po načtení projdeme toto pole a určíme počet lichých vrcholů, který vydělíme 2. Dostaneme, kolikrát musíme zvednout pero při kreslení grafu.

Paměťová složitost je $\mathcal{O}(n)$, časová $\mathcal{O}(m + n)$, kde m je počet hran grafu.

Martin Böhm, Lucie Mohelníková

Toky v sítích

23-4-1: Studenti a profesori

Je docela jasné, že si budeme uzpůsobovat první ze dvou aplikací hledání maximálního toku, o nichž se píše v kuchařce. Tato aplikace říká, jak pomocí toku najít maximální párování. Postavíme si ze zadání bipartitní graf, zorientujeme v něm hrany k profesorům, vrcholy studentů a profesorů pak napojíme na studentský zdroj a profesorský stok.

Protože chceme, aby měl student právě K profesorů, nastavíme váhu každé z hran ze studentského zdroje na K – to samé uděláme hranám do profesorského stoku, to aby měl každý profesor právě K studentů. Hranám uvnitř někdejšího bipartitního grafu nastavíme jedničky.

Povšimněme si tu, že kdyby zadání nezakazovalo, aby si některý student vybral profesora pro několik svých prací, vyrovnali bychom se s tím jednoduše – hraně, která by mezi příslušnými vrcholy vedla, bychom nastavili kapacitu na povolenou maximální násobnost.

Samozřejmě by ani nebyl problém mít rozdílný počet profesorů a studentů, či dokonce zavést individuální požadavky na počet vedených prací. Zadání bylo tak jednoduché předně proto, aby neděsilo.

Vraťme se k původní úloze. Na popsáný graf pustíme tokový algoritmus zachovávající celočíselnost a získáme z něj výsledek. Pokud není nalezený tok velký právě NK , řešení, které by každého plně uspokojilo, není. Pokud ano, vypíšeme páry profesor-student, jejichž hrana má jednotkový tok.

Důvod, že postup funguje, můžeme načrtnout třeba skrze fakt, že tok větší než NK v grafu existovat nemůže. Svědčí o tom řez na hranách mezi studentským zdrojem a studentskými vrcholy, kde je N hran, každá o kapacitě K .

Z toho vidíme, že pokud nám algoritmus vrátí takto velký tok, musí vést z každého studentského vrcholu k profesorům K jednotkových hran (a podobně ze strany profesorů), tedy jde o skutečné řešení našeho původního problému.

Zároveň se nemůže stát, aby postup řešení (maximální tok) nenašel a ono by existovalo – vzhledem k tomu, že z každého řešení sestavíme tok o maximální velikosti.

Co časová složitost? Smířit se s tím, že má Edmondsův-Karpův algoritmus složitost $\mathcal{O}(M^2N)$, je přístup lenivý. Nicméně si můžeme všimnout, že zlepšili-li každá cesta výsledek alespoň o jednotku, nenajdeme takových cest víc než KN .

Z toho plyne složitost $\mathcal{O}(KMN)$, což je lepší, protože pro $K > N$ úloha zřejmě není zajímavá.

Vysloveně akční přístup je začít se poohlížet po nekuchařkovém algoritmu. (To ale k získání maximálního počtu bodů potřeba nebylo.) Můžeme buď přemýšlet o tom, jestli není možné vzít Dinice či Goldberga a vzhledem k jisté speciálnosti našeho grafu vylepšit odhady časové složitosti, nebo zkusit najít specializovaný postup.

Vtip tkví v tom, že při zkoumání druhé možnosti nejspíše narazíme na Hopcroftův-Karpův algoritmus pro nalezení maximálního párování v bipartitním grafu běžící

v čase $\mathcal{O}(M\sqrt{N})$, který je však jen dobře odhadnutý a přeříkaný Dinic.

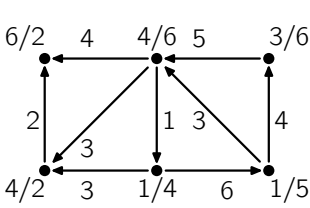
My tu sice nechceme bipartitní párování, leč každé naše řešení (K -regulární bipartitní podgraf) se skládá z K takových disjunktních množin hran (1-regulárních bipartitních podgrafů). To není úplně vidět, ale je to hezká a užitečná pravda.

Můžeme tedy K -krát spustit Hopcrofta-Karpa a pokud nějaké řešení existuje, získáme ho v čase $\mathcal{O}(KM\sqrt{N})$. Pořád tak netrůfneme škálu rozličných moderních algoritmů pro hledání maximálního toku na obecném grafu, jde však o celkem srozumitelné a snadno naprogramovatelné řešení.

Lukáš Lánský

23-5-6: Limity a grafy

Největší problém celé úlohy je poznat, že se jedná o toky v sítích. My si nyní tipneme, že se jedná o nějaký tok, a budeme se jej tam snažit najít. Jak na to?



Vstupní hrany a výstupní hrany jsou na sobě nezávislé v rámci vrcholu. Tak si každý vrchol rozdělíme na 2 nové vrcholy, levý a pravý.

Levý nám bude reprezentovat výstupní část (z této části povedou všechny hrany) a pravý bude reprezentovat vstupní část (do tohoto vrcholu naopak povedou všechny hrany).

Není těžké nahlédnout, že jsme takto vytvořili orientovaný bipartitní graf, kde všechny hrany vedou z levé partity do pravé.

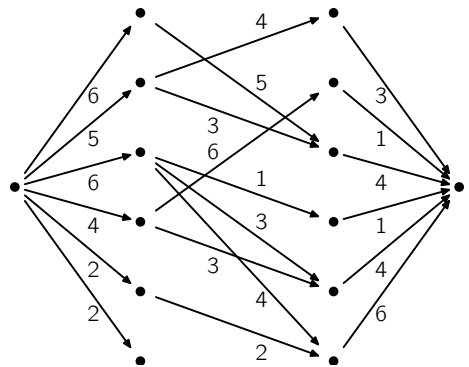
Nyní ještě potřebujeme zohlednit maximální vstupní součet a minimální výstupní součet. To uděláme tak, že do grafu přidáme další 2 vrcholy.

Jeden pojmenujeme zdroj a povede z něj hrana do každého vrcholu levé partity. Tyto hrany budou ohodnoceny maximálním výstupním součtem příslušných vrcholů.

Druhý pojmenujeme stok a z každého vrcholu pravé partity do něj povede hrana. Tyto hrany budou ohodnoceny minimálním vstupním součtem příslušných vrcholů.

Nyní máme ohodnocený orientovaný graf se zdrojem, stokem a celočíselnými kapacitami hran. Zavoláme tedy některý z algoritmů na hledání maximálního (celočíselného) toku, například Fordův-Fulkersonův algoritmus s hledáním zlepšujících cest pomocí prohlédávání do šířky (viz kuchařka).

Pokud se velikost maximálního toku bude rovnat sumě minimálních výstupních součtů, tak jsme našli příslušné ohodnocení. Pokud ne, tak neexistuje žádné řešení.



Proč to funguje? Hrany ze zdroje do levé partity nám zajišťují, že se do grafu nikdy nedostanou takové hrany, které by porušovaly podmínku maximálního výstupního součtu. Hrany mezi partitami jsou přesně ty samé hrany jako hrany v původním grafu.

Hrany vedoucí z pravé partity do stoku nám obstarávají minimální vstupní součty a jejich kapacity jsou právě tyto hodnoty. Kdybychom totiž měli řešení, ve kterém by některý vstupní součet byl větší než daný minimální, pak můžeme tok hran vedoucích dovnitř libovolně snížit tak, aby jejich součet byl roven minimálnímu vstupnímu součtu a všechny podmínky zůstanou zachovány.

Hrany z pravé partity do stoku tvoří v grafu řez. Problém má řešení, právě když tyto hrany jsou naplněny na maximum. Hrany mezi partitami nám také tvoří řez, takže vše, co proteče ze zdroje do stoku, proteče i hranami mezi partitami. A hrany mezi partitami reprezentují hrany původního grafu, takže tok na nich je naším řešením.

Nyní k časové složitosti. Časová složitost převodu na nový graf je $\mathcal{O}(n + m)$, kde n je počet vrcholů v původním grafu a m je počet hran.

Každý vrchol zdvojíme, na každou hranu se podíváme jen jednou a přidáváme jen 2 nové vrcholy a s nimi dohromady $2n$ hran. Zbytek časové složitosti závisí na použitém algoritmu pro zjištění maximálního toku. V našem případě, kdy jsme použili Forda-Fulkersona s procházením do šířky, je to $\mathcal{O}(nm^2)$.

Karel Tesař

Intervalové stromy

16-3-1: Fyzikova blecha

Jak tento bleší problém vyřešíme? Začneme tím, že si plošinky utřídíme podle y -ové souřadnice. Předpokládejme, že u konců každé plošinky víme, na jakou jinou plošinku z tohoto konce blecha spadne. Budeme probírat plošinky podle stoupající y -ové souřadnice a u každé plošinky si budeme u obou konců počítat nejkratší cestu na podlahu. To provedeme tak, že zkusíme ze zpracovávaného konce plošinky spadnout na nižší plošinku (víme, na kterou). Protože plošinka, na kterou dopadneme, je níž než zpracovávaná, už u ní známe nejkratší cestu z obou konců – vybereme si, zda jít doleva nebo doprava, aby byla cesta co nejkratší.

Celý tento postup zvládneme v čase $\mathcal{O}(N)$, protože u každé plošinky uděláme jen konstantně mnoho operací (zjistíme, na kterou plošinku spadneme, jak bude dlouhá cesta, když po dopadu zahneme nalevo, jak bude dlouhá cesta, když po dopadu zahneme napravo, vybereme minimum).


Jak tedy budeme u plošinky určovat, na jakou nižší blecha z jejího konce spadne? Použijeme k tomu *intervalový strom*. To je struktura, která si pro každý prvek s indexem 1 až P pamatuje nějaké číslo, přičemž P musí být pevně po celou dobu běhu programu. Intervalový strom umí dvě operace: *zjistí hodnotu prvku i a nastaví hodnotu prvků v intervalu $i \dots j$ na co , obě v čase $\mathcal{O}(\log N)$.*

Předpokládejme, že už takovou strukturu známe. Použijeme ji tímto způsobem: Jednotlivé prvky intervalového stromu budou použité x -ové souřadnice plošinek (je jich

nanejvýš $2N$) a hodnota prvku i (i -tá nejmenší x -ová souřadnice) je číslo nejvýše umístěné plošinky, která se na této x -ové souřadnici vyskytuje. Abychom mohli x -ové souřadnice očíslovat, musíme si je za začátku opět setřídít.

Na začátku dáme do intervalového stromu jen podlahu. Probereme si plošinky opět podle vzrůstající y -ové souřadnice a u každého konce (souřadnice $left_x, right_x$; předpokládejme, že po očíslování mají indexy $left_i, right_i$) se intervalového stromu zeptáme, jaká je hodnota prvku $left_i$ a $right_i$ (0 znamená podlahu, jiné číslo je pořadové číslo plošinky). Tím jsme zjistili, na kterou plošinku spadne blecha z levého a pravého konce plošinky. Poté zpracovávanou plošinku „přidáme“ do intervalového stromu, čili (pokud zpracováváme i -ou odspoda) do intervalového stromu zapíšeme hodnotu i do prvků v intervalu $left_i \dots right_i$.

Pokud tedy zvládneme implementovat popsany intervalový strom, máme řešení s časovou složitostí $\mathcal{O}(N \log N)$, protože třídění nás stojí $\mathcal{O}(N \log N)$ a dále zpracováváme N plošinek a každou v čase $\mathcal{O}(\log N)$. Paměťová složitost je jako obvykle $\mathcal{O}(N)$.

 *Intervalový strom* (pozor, malinko jiný než v kuchařce) si můžeme představit jako dokonale vyvážený binární strom. Jednotlivé vrcholy odpovídají intervalům z rozmezí 1 až P tak, že listy tohoto stromu jsou jednotlivé prvky (odpovídají intervalům $i \dots i$) a každý vnitřní vrchol odpovídá intervalu, který je roven sjednocení intervalů synů tohoto vrcholu. Čili vrchol celého stromu odpovídá intervalu $1 \dots P$, jeho levý syn intervalu $1 \dots \lfloor P/2 \rfloor$ a pravý syn intervalu $(\lfloor P/2 \rfloor + 1) \dots P$.

U každého vrcholu si budeme pamatovat jednak hodnotu h_i a jednak informaci p_i , zda hodnota h_i odpovídá všem prvkům na intervalu, který tento vrchol reprezentuje (u listů je to vždy *true*). Zjištění hodnoty nějakého prvku potom provedeme následovně: začneme ve vrcholu. Pokud je p_v *true*, vrátíme hodnotu h_v . Jinak si vybereme levého nebo pravého syna (podle indexu prvku, jehož hodnotu zjišťujeme), a rekurzíme (určitě se zastavíme, listy mají p_i na *true*).

Jak dopadne nastavení hodnoty prvků na intervalu $i \dots j$? Opět začneme ve vrcholu. Pokud interval, který zkoumaný vrchol v pokrývá, je podinterval $i \dots j$, nastavíme p_v na *true* a h_v na nastavovanou hodnotu. Jinak se spustíme na toho syna (případně na oba), jehož interval má neprázdný průnik s intervalem $i \dots j$. (Pozor: Bylo-li p_v *true*, je třeba nejprve rozdělit vrcholem reprezentovaný interval synům.)

Protože strom je dokonale vyvážený, má logaritmickou výšku. Obě operace závisí na výšce stromu (u nastavování intervalu je si to třeba rozmyslet – někdy se sice spustíme na pravého i levého syna, ale když to nastane, jednoho syna pokryjeme celého – nebudeme se z něj spouštět níže), mají tedy logaritmickou složitost.

Tomáš Vyskočil a Milan Straka

Těžké problémy

23-5-5: NP-úplný metr

Naším úkolem je dokázat, že úloha Metr je NP-úplná. Jak nám kuchařka radila, je příliš pracné dokazovat úplnost tak, že převedeme na Metr všechny úlohy z NP. Raději tedy dokážeme, že lze jednu NP-úplnou úlohu vyřešit pomocí Metru.

Nejtěžší v NP-úplnostních převodech bývá rozpoznat, která úloha se nám bude převádět nejsnáze.

Na Metru stojí za všimnutí, že překládání samotného metru do pouzdra nám v jistém smyslu rozděluje úseky na dva typy – pokud jde metr uložit, tak jeden typ úseku je přeložen na jednu stranu (řekněme zprava doleva) a druhý je přeložený nazpátek (zleva doprava). Navíc je metr zadán jako posloupnost čísel.

Když se podíváme do seznamu NP-úplných úloh, najdeme tam úlohu Dva loupežníci, která také rozděluje čísla na dvě hromádky. Zkusme tedy pomocí Metru řešit Loupežníky.

Připomeňme si zadání Dvou loupežníků z kuchařky:

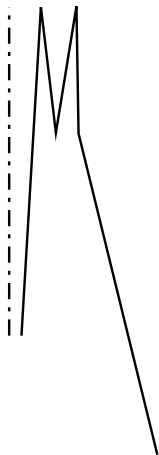
Název problému: Dva loupežníci

Vstup: Seznam nezáporných celých čísel.

Problém: Existuje rozdělení seznamu na dvě hromádky tak, že každé číslo bude v právě jedné hromádce a v každé hromádce bude stejný součet čísel?

Začneme tedy převádět vstup Dvou loupežníků na vstup Metru. Vstup Loupežníků nám nijak neurčuje, jak velké má být pouzdro metru – to si tedy můžeme zvolit sami, aby se nám snáz převádělo.

Dopředu není úplně jasné, jaká velikost by se nám hodila. Bude nám stačit součet všech předmětů (označujme ho σ), nebo velikost jednoho lupu, $\sigma/2$? Méně než $\sigma/2$ nedává příliš smysl, ale více by mohlo ...



Jak jsme diskutovali výše, mohlo by nám stačit označit ty části metru (tedy tu část kořisti), které jdou zleva doprava, jako lup pro loupežníka A a ty, které jdou zprava doleva, přiřadíme loupežníku B .

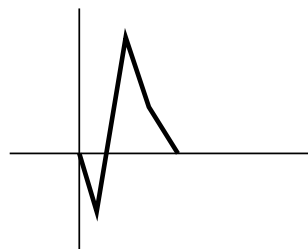
Nyní se zamysleme nad vstupy, které by nám mohly dělat neplechu. Například seznam předmětů $1\ 1\ 1$ by se do pouzdra velikosti alespoň 1.5 snadno vešel, ale my musíme odpovědět NE, protože jej rozdělit pro dva loupežníky nelze.

Mohli bychom tedy zkusit nastavit, aby začátek i konec lupu končil ve stejném bodě metru – například tak, že na začátek i konec přidáme úsek dlouhý jako celé pouzdro.

Tím by určitě odpadl případ $1\ 1\ 1$. Jak by taková úprava vstupu vypadala, vidíte na obrázku. Bohužel nám po chvíli úvah dojde, že by nám také odpadl případ $1\ 3\ 1\ 1$, který ovšem rozdělit jde.

Podívejme se na vstup $1\ 3\ 1\ 1$ a zamysleme se, jak naši úvahu vylepšit. Na dalším obrázku jsme jej zakreslili tak, aby se uložení metru podobalo grafu funkce, který začíná a končí v nule.

Každé rozdělitelné zadání Dvou loupežníků jde takto nakreslit – prostě jednu část kresleme jako rostoucí úsečky a druhou jako klesající.



Můžeme tedy vhodnou úpravou našeho vstupu pro Loupežníky zajistit, aby řešení Metru přesně odpovídalo grafu takovéto funkce?

Ano, stačí jen trochu upravit nápad, který jsme měli před pár odstavci. Potřebujeme totiž v Metru povolit, abychom mohli vstoupit na grafu i do „záporných hodnot“.

Na začátek metru tedy vložíme úsek o velikosti k , což bude také velikost pouzdra. Ten se dá do pouzdra vložit jen tak, že jeho konec bude na okraji pouzdra. Další úsek si tedy také zvolme – tentokrát jako $k/2$. Z okraje pouzdra jsme se tedy dostali přesně doprostřed. To bude náš počátek grafu.

Dále už pokládejme úseky o velikosti stejné, jako byly hodnoty na vstupu Dvou loupežníků, a ve stejném pořadí. Abychom se ujistili, že na konci opravdu naše funkce skončí v nule, přidejme ještě jeden úsek délky $k/2$ a za něj úsek délky k .

Nyní už víme, co od k chceme – abychom neřekli zbytečné NE, pokud bychom neměli dostatečný rozsah na jejich poskládání. Bude nám stačit nastavit $k = \sigma$, ale klidně bychom mohli mít pouzdro i větší.

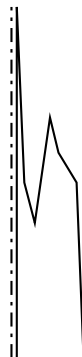
Převod je dokonán, pojďme si tedy ukázat, že je korektní.

Už během rozboru jsme si rozmysleli, že řešení Dvou loupežníků existuje právě tehdy, když existuje nakreslení lupu jako grafu funkce tak, že graf začíná i končí v nule.

V naší konstrukci platí, že metr lze vložit právě tehdy, když část odpovídající lupu loupežníků začíná a končí uprostřed pouzdra – a to platí právě tehdy, když existuje onen graf funkce začínající a končící v počátku.

Složením těchto ekvivalencí dostaneme, že náš převod odpoví ANO na Metr právě tehdy, když problém Dva loupežníci šel vyřešit, a tedy je vše v pořádku – Metr je NP-těžký.

Pro formální správnost si ještě povězme, že rozdělení metru (informace o tom, kde metr začíná a v jakém směru jej zlomit) je polynomiálně velkým certifikátem k našemu problému, a Metr je tedy v NP. Obě tvrzení spojíme dohromady a dostáváme, že Metr je NP-úplný.



Martin Böhm

Rejstřík

abeceda	92	fronta	34
Ackermannova funkce	53	funkce	
inverzní	53	hešovací	86
algoritmus		graf	30
Aho-Corasicková	98	bipartitní	115
Dijkstrův	43	obyčejný	30
Edmondsův-Karpův	115	ohodnocený	30
Floydův-Warshallův	67	orientovaný	30
Fordův-Fulkersonův	113	rovinný	101
Knuth, Morris, Pratt	97	souvislý	32
nalezení eulerovského tahu	108	úplný	32
pseudopolynomiální	67, 128	heš	86
aproximace	128	nafukovací	89
asymptotika	8	se separovanými řetězci	87
barvení grafu	102	se srůstajícími řetězci	87
BubbleSort	13	hrana	31
BucketSort	17	paralelní	107
certifikát	125	InsertSort	13
cesta	32	iterovaný logaritmus	52
hamiltonovská	110, 127	Kirchhoffův zákon	112
zlepšující	114	klíč	12
CountSort	16	klika	33
DFS strom	36	kolize	87
diff	68	komponenta souvislosti	32
Disjoint-Find-Union	48	kořen	34
důkaz		kostra	47
indukcí	44, 102	minimální	47
dvojrotace	78	Královec	107
dynamické programování	63	kružnice	33
Fibonacciho čísla	63	hamiltonovská	125
FIFO	34	lexikografické uspořádání	93
formule		LIFO	34
Eulerova	103	lokalita přístupů	12

medián	15	sled	33
MergeSort	14	uzavřený	107
most	40	složitost	6
multigraf	107	amortizovaná	51
náhodná procházka	110	asymptotická	8
NP-úplnost	126	časová, paměťová	6
óčková notace	8	kvadratická	10
párování	115	lineární	10
partita	115	polynomiální	10
path compression	50	v nejhorším/průměrném případě ..	9
penízková metoda	51	souvislost	32
pivot	15, 55	slabá, silná	32
podgraf	33	2-souvislost	40
pole prefixových součtů	118	splnitelnost	129
prefix	93	stěna	102
problém		vnější	102
batohu	65, 129	stok	38
čtyř barev	101	strom	34
existence k -kliky	129	AA	82
rozparcelování roviny	130	AVL	77
trojbarevnosti grafu	130	BB- α	83
3D párování	130	červeno-černý	82
prohledávání do hloubky	34	degenerovaný	73
prohledávání do šířky	37	dokonale vyvážený	77
QuickSelect	57	Fenwickův	121
QuickSort	15, 55	intervalový	118
RadixSort	17	left-leaning červeno-černý	82
rotace	78	prefixový	95
rozhodovací problém	124	rozhodovací	18
řazení	12	splay	82
řetězec	92	suffixový	95
prázdný	92	vyhledávací	74
řez	114	2-3	82
SelectSort	12	stupeň	104

suffix	93	přímým vkládáním	13
tah	33	přímým výběrem	12
eulerovský	107	sléváním	14
uzavřený	107	vnitřní a vnější	12
tok	112	union by rank	50
topologické uspořádání	38	věta	
treap	83	Betrandův postulát	90
triangulace	103	o čtyřech barvách	101
trie	93	o eulerovském tahu	109
komprimovaná	95	vrchol	31
třída NP	125	vyhledávací problém	125
třída P	125	vyvažování stromu	77
třídění	12	zásobník	34
bublínkové	13	zkrácené vyhodnocování	13
počítáním	16	znak	92
přihrádkové	17		

Böhm, Lánský, Veselý a kolektiv Programátorské kuchařky

Editori:

Martin Böhm, Lukáš Lánský a Pavel Veselý

Autoři textů kuchařek:

Martin Böhm, Zdeněk Dvořák, Dan Král, Lukáš Lánský,
Martin Mareš, David Matoušek, Milan Straka, Petr Škoda,
Karel Tesař, Tomáš Valla a Pavel Veselý

Vydal MATFYZPRESS

vydavatelství Matematicko-fyzikální fakulty Univerzity Karlovy v Praze
Sokolovská 83, 186 75 Praha 8
jako svou 375. publikaci.

Ilustrace vytvořili autoři kuchařek, autorem ilustrace na obálce je Martin Kruliš.
Sazba byla provedena písmem Computer Modern v programu \TeX .

Vytisklo Reprošředisko UK MFF.

Dotisk 1. vydání, 172 stran
Náklad dotisku 32 výtisků
Praha 2014

Publikace byla vydána pro vnitřní potřebu fakulty a není určena k prodeji.
Můžete si ji ale zdarma stáhnout na stránkách <http://ksp.mff.cuni.cz/>.
Všechny texty jsou navíc pod licencí Creative Common BY-NC-SA 3.0. :-)

ISBN 978-80-7378-181-1

ISBN 978-80-7378-181-1

