Řešení úloh

Třídění

18-2-4: Stavbyvedoucí

Ah, bezdůvodně čekáš, čtenáři drahý, ďábelské fígle, i když na obyčejný počátek prachobyčejného řešení stavbyvedoucího strastí tato věta vyznívá značně zvláštně. Demonstruje totiž jeden velice důležitý fakt: setřídit 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 setřídit podle prvního, pak zvlášť každou skupinu začínající stejným písmenem setřídit 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 tímtéž 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 setřídit 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 sestrojit 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:

```
nl -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. Přijde-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 setřídit 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? Bublinkové třídění (BubbleSort) bude zcela jistě fungovat, protože v průběhu algoritmu prohazujeme jen dva sousední prvky, což lze udělat jednoduše.

Bublinkové 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 stáří 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 slévání (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.

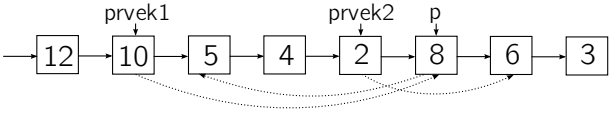
Často se může stát, že poslední slévaný úsek v k-tém kroku nemusí mít 2^k prvků, ale to vůbec nevadí (jeden slévaný úsek bude menší). Podobně lichý počet slévaných úseků (nemůžeme je spárovat do dvojic) ošetříme prostým ignorováním posledního úseku. V nějakém pozdějším kroku musí být tento úsek slit 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 zbývajících 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 tejto úlohe 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 zložitosťou $\mathcal{O}(N^2K)$, 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 vybraté intervaly sa neprekrývajú.

Prevod je nasledovný: všetky čísla a_i sú rovné pozíciam 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 vybratých 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 vybratých 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. Naviac 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 čislo 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,$R-1$,$R-2$,..., 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 riešení 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 nejjednodušší řešení 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 $A_t^t, A_{t+1}^t, \ldots, A_T^t$), 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 $A_{t+1}^{t+1}, A_{t+2}^{t+1}, \ldots, A_T^{t+1}$). 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 $B_t^t, B_{t+1}^t, \ldots, B_T^t$, které nám zajistí lepší zisk.

Zároveň ale víme, že odměna za úkoly $A_{t+1}^t, A_{t+2}^t, \ldots, A_T^t$ je alespoň stejně velká jako za zakázky $B_{t+1}^t, B_{t+2}^t, \ldots, B_T^t$ (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_t^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 setřídíme úkoly dle termínu. Pak budeme odzadu generovat jednotlivé úkoly, které je třeba v daný čas t udělat. K tomu použijeme haldu. 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.

Setřídění 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 haldu. 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 pointery 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 Ondrúškovi.)

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 spočíst 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 Klimošová

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 řešení 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 indukcí 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 $(*)$). Všimněme 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 vetší. 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 nezesouvislněné 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 $I \to X$ a $J \to Y$. 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 haldu 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 trošku zmatek.

Jiné, jednodušší řešení je haldu 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 haldu s přepočítáváním, dostaneme časovou složitost $\mathcal{O}(XYK\cdot\log(XY))$. Jednodušší halda dá časovou složitost $\mathcal{O}((N+M)\cdot\log M)\le\mathcal{O}((N+M)\cdot\log N^2)=\mathcal{O}((N+M)\cdot 2\log N)=\mathcal{O}((N+M)\cdot\log N)$, takže vlastně tutéž.

Paměťová složitost je u haldy s přepočítáváním $\mathcal{O}(XY)$, protože si potřebujeme pamatovat jen les a haldu 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 $K\le XY$, 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 trvalé 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 Dijkstrovi 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 $P = (s = v_0, v_1, \ldots, v_n = c)$, 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-

povat 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.

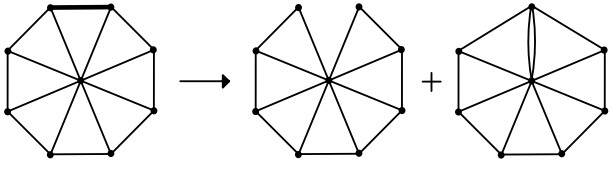
Úkolem je vlastně spočítat počet koster daného grafu. Vzorec na výpočet počtu koster úplného grafu nám nepomůže, protože kormidlo není úplný graf. Stejně tak postupy pro obecné grafy jsou trochu jako nukleární bomba na vrabce. Jde to jednodušeji.

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 $h \cdot M^{\Rightarrow\Leftarrow}$.

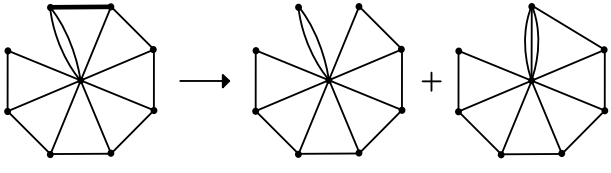
Protože jsou tyto dvě podmnožiny disjunktní a dohromady dávají celou množinu koster (nic jiného, než že tam hrana je a že tam není, se stát nemůže), můžeme velikosti těchto dvou podmnožin jednoduše sečíst.

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 korμdla 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 rozkládacího 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 přepis výše zmíněných rozkladů na menší podproblémy.

Kdybychom nyní iterovali přes všechna potřebná N a k (všimněme si, že k bude nejvýše N – 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, $V_N^k = k \cdot V_{N-1}^1 + V_{N-1}^{k+1} = k \cdot V_{N-1}^1 + (k+1) \cdot V_{N-2}^1 + V_{N-3}^{k+2} = \ldots$. 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 $V_N = 1 \cdot V_{N-1} + 2 \cdot V_{N-2} + \ldots + (N-1) \cdot V_1 + N$. Pro celá kormidla to vyjde $\mu_N = 1^2 \cdot V_{N-1} + 2^2 \cdot V_{N-2} + \ldots + (N-1)^2 \cdot V_1 + N^2$.

Kdybychom nám někdo dal všechna $V_1, \ldots, V_{N-1}$, 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 vytvořujících funkcí (ta je moc pěkně popsaná ve starých dobrých Kapitolách z diskrétní 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 stejné 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 $\mathcal{O}(\log N)$.

```
/* Dvojsložková 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 popsaným 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 připojená 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.

Zbylý 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 prolnout 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 „Vorner“ Vaner

Rozděl a panuj

19-2-5: Hluboký les

Je zajisté triviální nalézt nejhlubší 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. Pojďme 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 pravoúhlého 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 $d \times d$ umisťovat 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.
- Doplň dvojice sahající přes hraniční přímku: zajímají nás jen takové dvojice, které mohou změnit výsledek, čili jejichž 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 dělící přímce bychom zpracovali úplně zvlášť, beztak padnou do pásu, ve kterém dvojice kontrolujeme explicitně.

Mnohem jednodušší je na začátku algoritmu setřídit všechny body podle svislé souřadnice a rozdělit je prostě na prvních $\lfloor N/2 \rfloor$ a zbylých $\lceil N/2 \rceil$. Různé body na dělící 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 umisťování 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 setřídit 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 zkoušíme 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 přečuhovat 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. Posaďte se, prosím, na svá místa, připoutejte se a během startu nekuřte.

Náš postup je založen na známém třídicím 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 slije 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) dojede za konec své posloupnosti, algoritmus končí, avšak ještě je třeba dokopírovat zbývající prvky z druhé posloupnosti (z té, která ještě nebyla zpracována celá). Např. pokud a dojede 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 slévání (o ostatní se postará rekurze).

Máme tedy algoritmus na slévání dvou posloupností popsaný 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 třídicí 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 nultého 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 třídit 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 vypsat 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 vypsat 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 vybalancovávaném 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 nejvýš $\mathcal{O}(N \cdot K)$).

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:

$$c(\textit{levého podstromu}) + c(\textit{pravého podstromu}) + 1.$$

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 $\ell = c(\textit{levý syn } u)$. 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 mazaní z binárního vyhledávacího stromu (viz kuchařka). Stejně tak musíme po mazaní 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 vypsat konečnou permutaci dokumentů. Stačí pouze projít a vypsat 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 logaritmicky 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
- $E_1 = E_1' \; op \; E_1''$, $E_2 = E_2' \; op \; E_2''$, 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 vypsat počet různých čísel hodnot v tabulce B, protože to bude právě počet různých hodnot výrazů v programu.

Čísla hodnot výrazů určujeme 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 $var_1 = var_2 \; op \; var_3$, 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 dořešit 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 zlepší časovou složitost na $\mathcal{O}(n)$.

Takže jsme nalezli 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 (věrni písmenkům ze zadání) N délku stopovaného řetězce, k počet podezřelých sekvencí, $p_1,\ldots,p_k$ délky těchto sekvencí a $P = p_1+\ldots+p_k$ 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 $\mathcal{O}(Np_1+\ldots+Np_k) = O(NP)$ 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 timu $\mathcal{O}(N+p_i)$, 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 (Šérlok 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 počítadlo odpovídající tomuto stavu k počítadlu stavu, do nějž 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 $Y_1, Y_2, \ldots, Y_n$. Pokud se navíc q vyskytuje v reprezentaci nějakého stromu X, našli jsme uvnitř X interval sousedících bratrských podstromů $Y_1, \ldots, Y_n$.

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 nepřečetli a nevíte jak najít q v reprezentaci A? Přece pomocí 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 $\mathcal{O}(N)$, 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 $\mathcal{O}(N^2)$ 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 přímku* 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 Gavenčiak

Eulerovské tahy

23-2-3: Projížďka

Milý čtenář mi jistě pro jednou odpustí, 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 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 popsaný v minulém případu 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 odpočatost v pootočeném tahu nikde neklesne do záporných čísel.

Předpokládejme tedy, že v místě $š$ na tahu máme zápornou odpočatost. Minimum máme v místě m. Pokud by v původním neotočeném tahu bylo $š$ až za m, pak by muselo být také s menším číslem než m a m by tedy nebylo minimum. Tento případ tedy nenastal.

Takže $š$ je před m. Představme si, že jsme prošli tahem dvakrát místo jednou, tedy při druhém průchodu $š$ jsme na nižším čísle, než při prvním průchodu m (proto nám po pootočení v $š$ vyšlo něco záporného). Ale protože druhý průchod nezačíná od nuly, ale od něčeho nezáporného, odpočatost druhého průchodu $š$ je alespoň tak velká, jako první. Tedy i při prvním průchodu $š$ 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

23-4-1: Studenti a profesoři

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 popsaný 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 – vždyť 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 zlepší-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.

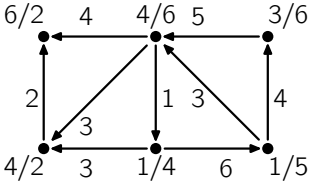
My tu sice nechceme bipartitní párování, leč každé naše řešení (K-regulární bipartitní podgraf) se skládá z K takových disjunktních množin hran (1-regulárních bipartitních podgrafů). To není úplně vidět, ale je to hezká a užitečná pravda.

Můžeme tedy K-krát spustit Hopcrofta-Karpa a pokud nějaké řešení existuje, získáme ho v čase $\mathcal{O}(KM\sqrt{N})$. Pořád tak netrumfneme 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.

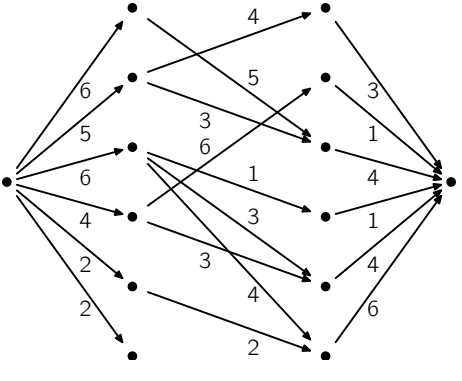
Levý nám bude reprezentovat výstupní část (z této části povedou všechny hrany) a pravý bude reprezentovat vstupní část (do tohoto vrcholu naopak povedou všechny hrany).

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 prohledávání 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 pevné po celou dobu běhu programu. Intervalový strom umí dvě operace: *zjisti hodnotu prvku* i a *nastav hodnotu prvků* v intervalu $i \ldots j$ 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 setřídit.

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 podlaha, 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 $left_i \ldots right_i$.

Pokud tedy zvládneme implementovat popsaný 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 $i \ldots i$) a každý vnitřní vrchol odpovídá intervalu, který je roven sjednocení intervalů synů tohoto vrcholu. Čili vrchol celého stromu odpovídá intervalu $1 \ldots P$, jeho levý syn intervalu $1 \ldots \lfloor P/2 \rfloor$ a pravý syn intervalu $(\lfloor P/2 \rfloor + 1) \ldots P$.

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 $i \ldots j$? Opět začneme ve vrcholu. Pokud interval, který zkoumaný vrchol v pokrývá, je podinterval $i \ldots j$, 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 $i \ldots j$. (*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

Našim 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:

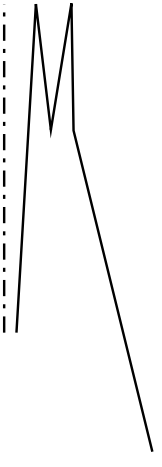
Název problému: Dva loupežníci

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?

Začněme 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 ...



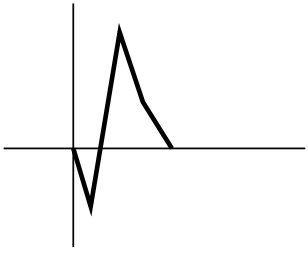
Jak jsme diskutovali výše, mohlo by nám stačit označit ty části metru (tedy tu část kořisti), které jdou zleva doprava, jako lup pro loupežníka A a ty, které jdou zprava doleva, přiřadíme loupežníku B.

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 vložme 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