INFORMATIKA

Informatické hádanky

PETR OSIČKA, PETR KRAJČA

Přírodovědecká fakulta UP, Olomouc

Hádanky jsou již dlouho součástí matematiky i informatiky. Některé z nich se již stali součástí folkloru dané oblasti, jiné se objevují při pracovních pohovorech (například v Google) či v odborných magazínech. Řešení hádanek je příjemnou zábavou, při které lze dobře procvičit myšlení, u hádanek informatických myšlení algoritmické.

1. Dělení parlamentu

Víme, že každý poslanec má v parlamentu přesně tři nepřátele. Přitom předpokládáme, že nepřátelství je symetrické. Tedy považuje-li poslanec A poslance B za svého nepřítele, pak i poslanec B považuje poslance A za svého nepřítele. Lze parlament rozdělit na dva kluby tak, aby každý poslanec měl klubu, do kterého patří, nejvýše jednoho nepřítele? Zápornou odpověď odůvodněte důkazem, u kladné odpovědi uveďte, jak by se takové rozdělení zkonstruovalo.

Odpověď na otázku je ano. Rozdělení poslanců na kluby lze totiž zkonstruovat následujícím způsobem.

1. Libovolně rozdělíme poslance do klubů. Nemáme žádné další požadavky, jeden z klubů může být dokonce prázdný.
2. Dokud existuje poslanec, který má více než jednoho nepřítele ve svém klubu, přesuneme jej do opačného klubu.

Vidíme, že pokud algoritmus skončí, rozdělí poslance do klubů podle našich požadavků. Ty totiž přesně odpovídají podmínce z kroku 2 algoritmu. Stačí tedy ukázat, že algoritmus končí.

Na první pohled se zdá, že můžeme poslance přesouvat libovolně krát. Může se totiž stát, že přesuneme poslance do opačného klubu a později za ním přesuneme i některé jeho nepřátele. Nemůžeme na první pohled vyloučit situaci, že pokud původního poslance přesuneme zpět, budeme muset později přesunout zpět i jeho nepřátele. Celá situace by se pak mohla opakovat a algoritmus by nikdy neskončil.

Ve skutečnosti se dá dokázat, že přesunů mezi kluby se provede pouze konečně mnoho. Označme p počet nepřátelských dvojic poslanců, kteří se nachází ve stejném klubu. A nyní klíčová myšlenka:

Přesunem poslance v kroku 2 se p zmenší.

Je to proto, že před přesunem má poslanec ve svém klubu více nepřátel než po přesunu. Současně je ale číslo p omezeno, triviálně například vidíme

$$0 \leq p \leq n^2.$$

Řešení dnešní hádanky je ve skutečnosti algoritmem pro optimalizační úlohu nalezení maximálního řezu grafu. V této úloze hledáme rozklad množiny vrcholů grafu na dvě množiny tak, aby byl počet hran mezi uzly z různých množin maximální.

2. Nerozbitné skleničky

Východoněmecký podnik vyrábí velmi odolné skleničky. Pro účely propagace potřebuje zjistit nejvyšší patro své 100 patrové budovy, ze kterého se z okna vyhozená sklenička dole na ulici nerozbije. K dispozici nám dá 2 skleničky a přístup do budovy, takže můžeme skleničky vyhodit z námi zvoleného patra. Vyhození jedné skleničky a zjištění toho, jestli se rozbila, chápeme jako jeden pokus. Jaký je minimální počet pokusů, které v nejhorším případě potřebujeme, chceme-li úlohu vyřešit?

Úloha je jednodušší, pokud máme k dispozici jenom jednu skleničku. Abychom nalezli požadované patro, musíme skleničku postupně vyhazovat z $1., 2., 3., \ldots, 100.$ patra. Rozbije-li se v patře k, je hledané patro $k-1$. Odpověď na naši otázku je 100, protože v nejhorším případě se sklenička nerozbije ani ve stém patře.

Druhou skleničku můžeme využít k urychlení hledání. Po krátkém zamyšlení zjistíme, že nemůžeme při hledání našeho patra použít analogie půlení intervalů ani jiný jednoduchý trik. Pomůže nám až pohled z druhé strany. Označíme si $H(k)$ maximální výšku budovy, pro kterou zvládneme požadované patro nalézt s použitím k pokusů. Máme-li k dispozici obě

skleničky, provedeme pokus v patře k. Pokud se sklenička rozbije, zbude nám $k-1$ pokusů pro patra $1, 2, \ldots, k-1$, tak jako v případě s jednou skleničkou. Pokud se naopak sklenička nerozbije a zůstává nám ještě $k-1$ pokusů i obě skleničky, můžeme prozkoumat dalších $H(k-1)$ pater. Pro $H(k)$ tedy dostaneme rekurenci

$$H(k) = k + H(k-1); \quad H(1) = 1,$$

jejímž rozbalením dostaneme uzavřenou formu

$$\begin{aligned} H(k) &= k + H(k-1) \\ &= k + (k-1) + (k-2) + \cdots + 1 \\ &= k(k+1)/2. \end{aligned}$$

Nyní zbývá nalézt nejmenší k takové, aby platilo

$$100 \leq \frac{k(k+1)}{2}.$$

Snadno ověříme, že hledané k je 14.

Dokud máme obě skleničky, provádíme pokusy postupně v patrech

$$14,\ 27,\ 39,\ 50,\ 60,\ 69,\ 77,\ 84,\ 90,\ 95,\ 99,\ 100.$$

Po rozbití skleničky pokračujeme od patra nad tím, ve kterém jsme provedli poslední pokus bez rozbití.

3. Sto tisíc telefonních čísel

Představte si, že jste zaměstnanec nebo zaměstnankyně telekomunikačního operátora a máte soubor obsahující sto tisíc šestimístných telefonních čísel, která jsou přiřazena jednotlivým zákazníkům. Vaším úkolem je čísla seřadit a vypsat. Ale aby tento úkol nebyl úplně jednoduchý, předpokládejte, že z důvodů úspor má váš nejlepší počítač jen 256 KB[1)] paměti RAM.[2)]

[1)] Předpokládáme, že 1 KB = 1024 B.

[2)] V relativně nedávné minulosti, říkejme jí třeba osmdesátá léta, počítače s takto velkou pamětí nebyly nic neobvyklého.

3.1. Přímočaré řešení

Ze zadání víme, že jednotlivá telefonní čísla budou šestimístná. Můžeme tedy uvažovat, že každé telefonní číslo je celé číslo z intervalu 0 až 999.999 (včetně). Nabízí se proto pro uložení telefonních čísel použít třicetidvoubitový typ `int`. Stačí tedy načíst jednotlivá čísla do pole hodnot typu `int` a setřídit nějakým vhodným algoritmem jako je *QuickSort*.

Toto řešení je sice přímočaré a jednoduché, ale v námi nastavených podmínkách nebude fungovat. Do 256 kB paměti se nám vejde jen 65535 hodnot typu `int`, a to nám nezbude ani místo pro samotný program, ani pro operační systém.

Další možností by bylo řadit data s využitím vnější paměti. Takové řešení by již fungovalo, ale nebylo by patrně příliš rychlé, ani elegantní, a proto se jím nebudeme dále zabývat.

3.2. Lepší řešení

Při hledání vhodnějšího řešení můžeme využít toho, že v seznamu čísel bude každé telefonní číslo nanejvýš jednou. Můžeme si proto udělat seznam úplně všech čísel z intervalu [0;999.999] a poznačit si, která čísla jsou již zákazníkům přiřazena. Když tento seznam následně projdeme od začátku do konce a vypíšeme jen přiřazená čísla, získáme seřazenou posloupnost přesně, jak potřebujeme.

Jak ale vytvořit takový seznam? Opět můžeme využít skutečnost, že každé číslo na vstupu bude nanejvýš jednou a pro reprezentaci seznamu můžeme použít bitmapu. V této bitmapě každý bit bude představovat příznak, zda dané telefonní číslo bylo přiřazeno, nebo ne.

Výsledné řešení by mohlo vypadat následovně. Nejdříve si vytvoříme prázdné pole, které bude v paměti zabírat přesně jeden milion bitů, tj. 125 tis. bytů (nebo 31.250 slov o velikosti 32 bitů). Toto pole inicializujeme nulami.

```
1  int32_t numbers[1000000 / 32];
2  for (int i = 0; i < 1000000 / 32; i++)
3      numbers[i] = 0;
```

Všimněme si, že v tomto případě nám 256 kB paměti bude dostačovat. Výhodou i nevýhodou může být, že bez ohledu na to, kolik hodnot je na vstupu, vždy alokujeme stejné množství paměti.

Dále předpokládejme, že máme funkci `get_number`, která postupně na-

čítá a vrací hodnoty ze vstupu. V momentě, kdy narazí na konec vstupu, vrátí zápornou hodnotu. Zpracování vstupu bude v takovém případě vypadat následovně.

```
int32_t n;
while ((n = get_number()) >= 0) {
    numbers[n / 32] |= 1 << (n % 32);
}
```

Nejdříve se načte do proměnné `n` hodnota ze vstupu, a pokud je nezáporná, vyznačí se příslušný bit v poli `numbers`. Hodnota `numbers[n / 32]` říká, ve kterém slově budeme nastavovat příslušný bit, výraz `1 << (n % 32)` udává, který bit se v daném slově nastaví na jedna a operace `|=` se postará o samotné nastavení příslušného bitu.

Posledním krokem je vypsání seřazené posloupnosti čísel. To provedeme v podobném duchu. Projdeme všechna čísla od 0 do 999.999 (včetně) a podíváme se, zda pro ně byl v poli `numbers` nastaven odpovídající bit na 1.

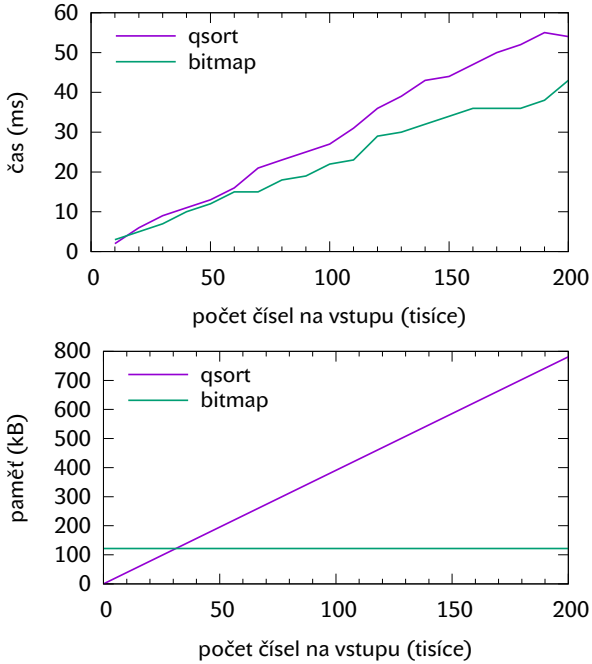
```
for (int i = 0; i < 1000000; i++) {
     if (numbers[i / 32] & (1 << (i % 32)))
         printf("%i\n", i);
}
```

Všimněme si, že pro výběr bitů používáme podobný postup jako v případě jejich nastavení v předchozím kroku. K testování, zda je příslušný bit nastaven, slouží operátor `&`, který představuje bitový součin dvou hodnot. Jinými slovy, pokud je příslušný bit nastaven na jedna, bude hodnota výrazu `numbers[i / 32] & (1 << (i % 32))` nenulová, a tudíž to bude interpretováno jako splnění podmínky. V opačném případě nebude podmínka splněna.

3.3. Srovnání s přímočarým řešením

Dalo by se namítnout, že s dnešními počítači, které mají mnoho gigabytů RAM a k tomu virtuální paměť, omezení na množství paměti, které v této hádance máme, je příliš vykonstruované. Udělali jsme proto experiment a srovnali jsme čas nutný k seřazení posloupnosti čísel pomocí QuickSortu a pomocí bitmapy. Výsledky ukazuje obr. 1 (nahoře). Pro menší množství čísel je mírně lepší QuickSort, ale s rostoucím počtem čísel postupně ztrácí na řešení vycházející z bitmapy.

Druhá námitka by mohla směřovat k tomu, že navržené řešení má tendenci plýtvat s pamětí. Alokovat za všech okolností bitmapu o velikosti 1.000.000 bitů může vypadat jako rozmařilost. Je ale potřeba brát v potaz, že uložení celých čísel má také svou nezanedbatelnou režii. Jak vypadají nároky na paměť u obou algoritmů ukazuje obr. 1 (dole).



Obr. 1 Srovnání náročnosti obou přístupů z pohledu rychlosti (nahoře) a nároků na paměť (dole).

Jednoduchou úvahou se dá dobrat k tomu, že pokud počet řazených hodnot k je větší než $\frac{1}{32}H$, kde H je počet všech možných hodnot, je z pohledu spotřebované paměti výhodnější bitmapa.

Z toho se dá usuzovat, že běžné řešení (např. QuickSort) nemusí být za všech okolností to nejefektivnější.

Zbylé hádanky jsou převzaty z knihy [1].

Literatura

[1] *Levitin, A., Levitin, M.*: Algorithmic puzzles. Oxford University Press, New York, 2011.