1. Úvodní příklady, definice RAM

Příklad: Reportáž

Novinář má za úkol za rok napsat reportáž o pracovních podmínkách v jedné nejmenované firmě. Musí tedy vyzkoušet co nejvíce pracovních pozic. Chce ale, aby se mu neustále zvyšoval plat. Firma v různých časech vypisuje pracovní místa.

Řečeno matematicky, máme zadánu posloupnost $p_1, \ldots, p_n$ reálných čísel a hledáme v ní nejdelší ostře rostoucí vybranou podposloupnost.

Jak můžeme takový problém řešit?

Podle definice: budeme generovat všechny podposloupnosti a testovat, jestli jsou rostoucí. Podposloupnost můžeme popsat charakteristickým vektorem, což je posloupnost nul a jedniček, kde na i-té pozici je 1, právě když podposloupnost obsahuje i-tý člen původní posloupnosti. Charakteristické vektory odpovídají binárním zápisům čísel 1 až 2^n kde n je počet vypsaných prací.

Charakteristické vektory můžeme generovat například tak, že si cifry binárního čísla budeme udržovat v poli a budeme přičítat 1. Po hlubších úvahách (zvídaví hledejte pojem amortizovaná časová složitost) zjistíme, že na jedno přičtení jedničky potřebujeme průměrně konstantně mnoho operací.

Nyní nás bude zajímat kolik řádově provedeme kroků. Všech charakteristických vektorů je 2^n. Pro každý zkontrolujeme, jestli je podposloupnost rostoucí, což zabere n kroků. Celkem tedy provedeme řádově $2^n \cdot n$ kroků.

Rekurzívně: vytvořme funkci, která dostane začátek posloupnosti a najde všechna rozšíření na rostoucí podposloupnost. Zajímá nás ale jen nejdelší podposloupnost, položme tedy $f(i_1, \ldots, i_k) :=$ maximání délka rostoucí podposloupnosti navazující na $x_{i_1}, \ldots, x_{i_k}$.

Probereme všechna j od $i_k + 1$ do n a pro každé j takové, že $x_j > x_{i_k}$, nastavíme maximum $m \leftarrow \max(m, f(i_1, \ldots, i_k, j) + 1)$. Jako výsledek funkce vrátí m. Na začátek posloupnosti přidáme $-\infty$ a zavoláme $f(0)$.

1. Pro $j = i_k + 1$ to n
2. Když $x_j > x_{i_k}$
3. $m \leftarrow \max(m, f(i_1, \ldots, i_k, j) + 1)$
4. Vrať m

Nejhorším případem je rostoucí posloupnost, na které naše funkce vykoná řádově 2^n kroků.

Zamysleme se, jestli potřebujeme prvních $k-1$ parametrů. Pokračování podposloupnosti může ovlivnit poze poslední parametr funkce f. Zjednodušíme tedy volání funkce a místo $f(i_1, \ldots, i_k)$ budeme volat $f(i_k)$.

Rekurze s blbenkou: $f(i)$ bude volána mnohokrát pro stejné i. Nejlépe je to vidět na příkladu rostoucí posloupnosti, kde je $f(i)$ volána po každém zavolání $f(j)$ kde $j < i$.

V poli X si pamatujeme výsledky funkce f pro jednotlivá i, tedy pole X obsahuje na pozici i hodnotu $f(i)$.

Cvičení: ukažte, že algoritmus vykoná řádově n^2 operací a spotřebuje n buněk paměti.

Bez rekurze: všimněme si, že spočítat $f(n)$ je velmi snadné ($f(n) = 0$).

1. $f(n) = 0$
2. $k = n - 1 \ldots 0$
3. $\quad f(k) = 0$
4. $\quad j = k + 1 \ldots n$
5. $\qquad$ Když $x_j > x_k$
6. $\qquad\quad f(k) = \max(f(k), f(j) + 1)$

Rychlost jsme nezvýšili, dokonce ani paměť jsme neušetřili, ale zbavili jsme se rekurze.

Převést úlohu na grafovou je standardní informatický trik. Vrcholy jsou čísla $V := \{1, \ldots, n\}$, hrana $(i, j) \in E \equiv i < j \ \& \ x_i < x_j$. Cesty v tomto grafu odpovídají vybraným rostoucím posloupnostem a my hledáme nejdelší cestu v acyklickém grafu, což umíme (budeme umět) lineárně s velikostí grafu. Hran může být až $|E| = \binom{n}{2} \approx n^2$. Čímž jsme dostali další kvadratický algoritmus.

Datová struktura: během semestru poznáme šikovnou datovou strukturu, která obsahuje uspořádané dvojice reálných čísel (x, y), kde x je klíč a y hodnota. Po této struktuře budeme chtít aby uměla vložit dvojici *Insert*(x, y) a dotaz *Query*: *Query*(t):= $\max\{y \mid \exists x \geq t : (x, y)$ je ve struktuře$\}$.

Postupujeme podobně jako v algoritmu *Bez rekurze* s tím rozdílem, že kroky 4 až 6 za nás udělá datová struktura. Pro každé k zavoláme *Insert*$(x_j, f(j))$ a *Query*(x_{k+1}). Obě trvají řádově $\log n$, struktura nám vrátí největší hodnotu $f(j)$ pro dané x_{k+1}.

Provedeme tedy řádově $n \cdot \log n$ kroků, což je nejlepší známé řešení.

Algoritmus

Na příštích přednáškách budeme studovat algoritmy a jejich vlastnosti. Co ale algoritmus doopravdy je? Jak ho definovat? Žádná pořádná definice algoritmu neexistuje. Pro nás bude algoritmus program v nějakém jazyce na nějakém výpočetním stroji (viz definice RAM).

Churchova teze: všechny definice algoritmů jsou ekvivalentní. Toto není opravdová věta, spíš vyjadřuje, že všechny rozumné definice algoritmu definují v podstatě to samé.

Model RAM

V předchozí části jsme mluvili o výpočetním modelu, pojďme tedy nějaký nadefinovat. Výpočetních modelů je více, my vybereme jeden poměrně blízký skutečným počítačům.

Definice: Random Access Machine (RAM)

RAM počítá jen s celými čísly (dále jen *čísla*). Znaky, stringy a podobně reprezentujeme čísly, jejich posloupnostmi atd. Paměť je tvořena buňkami, které obsahují čísla. Paměťové buňky jsou adresované taktéž čísly. Program samotný je konečná posloupnost instrukcí (také opatřených adresami) následujících druhů:

(kde X, Y jsou nějaké operandy)

- Datové přesuny X `<-` Y
- Aritmetické, logické a bitové: X `<-` $Y \oplus Z$

  $\oplus \in \{+, -, *, \texttt{div}, \texttt{mod}, \&, |, <<, >>\}$ kde &, | znamenají logické and a or, <<, >> znamenají bitový posun vlevo a vpravo.
- Řídící: skok `goto` Z, podmíněný skok `Když` X `<` Y `goto` Z, zastavení programu `halt`.

Operandy:

- Konstanty $(1, 2, \ldots)$
- Adresované přímo – `[`*konst.*`]` – budeme používat písmena `A-Z` jako aliasy pro buňky paměti -1 až -26 (tedy A=`[-1]`), které nazýváme *registry* a budou nám sloužit jako proměnné (samozřejmně nejen ony).
- Adresované nepřímo – `[[`*konst.*`]]`

  Můžeme se chtít podívat na adresu, kterou máme uloženou v nějaké buňce, podobně jako pointery v C.

Samotný výpočet probíhá takto:

1. Do smluvených buněk umístíme vstup, obsah zbylých paměťových buněk není definován.
2. Provádíme program po instrukcích, dokud nedojdeme k `haltu` nebo konci programu.
3. Pokud se program nezacyklil, tedy pokud skončil, ze smluvených buněk přečteme výstup.

Míry složitosti

1. *RAM s jednotkovou cenou*: čas = # instrukcí při daném vstupu, prostor = # buněk do kterých algoritmus aspoň jednou zapsal během výpočtu.

   Toto není moc dobrý nápad, protože není nijak penalizována například práce s velmi dlouhými čísly – pořád je to jedna instrukce, takže cena je stejná, ale počítače se tak přece nechovají. Velikost čísel ale konstantou (třeba 32 bitů) omezit nesmíme, protože bychom omezili paměť (čísly ji adresujeme) a co hůř i možnou velikost vstupu.
2. *RAM s logaritmickou cenou*: cena instrukce = # bitů zpracovávaných čísel, prostor = # bitů všech použitých buněk. To je teoreticky přesné, ale dost nepraktické (ve všech složitostech by byly spousty logaritmů).
3. *RAM s omezenými čísly*: jednotková cena instrukcí, ale čísla omezíme nějakým polynomem $P(n)$, kde n je velikost vstupu. Tím zmizí

paradoxy prvního modelu, ale můžeme adresovat jen polynomiální prostor (to nám ovšem obvykle nevadí).

Nadále budeme předpokládat třetí zmíněný model.

Definice:

- *Čas běhu algoritmu* $t(x)$ pro vstup x měříme jako sumu časů instrukcí, které program provedl při zpracování vstupu x. Pokud se pro daný vstup program nezastaví berme $t(x) = +\infty$.
- *Prostor běhu algoritmu* $s(x)$ je analogicky počet paměťových buněk použitých při výpočtu se vstupem x.

Chceme zavést míru časové a prostorové náročnosti programů zvanou složitost. Složitost je maximum délky běhu přes všechny vstupy určité délky.

- *Množina možných vstupů* X
- *Délka vstupu* je funkce $l : X \to \mathbb{N}$
- *Časová složitost* (v nejhorším případě) je:

$$T(n) := \max\{t(x) \mid x \text{ je vstup délky } n\}.$$

- *Prostorová složitost* (v nejhorším případě) je:

$$S(n) := \max\{s(x) \mid x \text{ je vstup délky } n\}.$$

Podobně můžeme zavést i složitost v nejlepším a průměrném případě, ale ty budeme používat jen zřídka.