Metoda rozděl a panuj

Základy algoritmizace – 6. cvičení

Zuzana Petříčková

24. března 2020

Metoda rozděl a panuj (divide and conquer, divide et impera)

- jedna ze základních metod tvorby algoritmů
- aplikace metody shora dolů:
  1. **ROZDĚL**: rozdělíme úlohu na několik menších podúloh stejného typu (jako původní úloha)
  2. **VYŘEŠ**: podúlohy vyřešíme
  3. **SPOJ**: spojíme řešení podúloh → odvodíme výsledné řešení

→ vede na rekurzivní metodu se složitostí tvaru:

$$T(n) = aT(\tfrac{n}{b}) + f(n),$$
$$T(1) = f(1)$$, kde

- a … úlohu dělíme na a podúloh
- $\frac{n}{b}$ … velikost jedné podúlohy
- $f(n) = O(n^c)$ … časová složitost fází **ROZDĚL** a **SPOJ**

Metoda rozděl a panuj (divide and conquer, divide et impera)

Samostatná práce

- nastudujte si skripta, kapitola 3.1, strana 80-82
- ve skriptech se zaměřte na příklad 3.1 a funkci BinarniVyhledavani a zkuste jí porozumět

Binární vyhledávání (vyhledání půlením intervalů)

- máme pole délky n a v něm jsou hodnoty (klíče) uspořádané podle velikosti (od nejmenšího po největší)
- chceme v poli rychle vyhledat daný klíč (na jakém je indexu?)

Jak na to:

- počáteční interval indexů v poli, kde hledáme, je $[l, r] = [0, n - 1]$, podíváme se na prostřední prvek v poli (na indexu $s = \frac{l+r}{2}$)
  - je roven klíči? $\rightarrow$ hurá, našli jsme ho (vrátíme s)
  - je větší než klíč? $\rightarrow$ budeme hledat dál v intervalu $[l_1, r_1] = [l, s - 1]$
  - je menší než klíč? $\rightarrow$ budeme hledat dál v intervalu $[l_1, r_1] = [s + 1, r]$

$\rightarrow$ algoritmus vlastně imituje vyhledávání v binárním vyhledávacím stromě (o optimální hloubce) uloženém v poli.

Pokud BVS nemusíme měnit a jen v něm vyhledáváme, je jeho implementace v poli docela efektivní

Binární vyhledávání (vyhledání půlením intervalů)

Jak můžeme binární vyhledávání implementovat?

1. pomocí rekurzivní funkce (viz skripta)
2. bez rekurze (to zkusíme my)

Jaká je časová složitost binárního vyhledávání?

- bude stejná jako složitost vyhledávání v BVS optimální hloubky (odvození viz skripta)
- můžeme spočítat i metodou zvanou ,,kuchařka" (ukážeme si za chvíli)

$T(n) = T(\frac{n}{2}) + 0(1),$
$T(1) = 0(1)$
$\rightarrow \boldsymbol{T(n) = O(log_2 n)}$

Postavení binárního vyhledávacího stromu (BVS) z uspořádaného pole

- máme pole délky n a v něm jsou hodnoty (klíče) uspořádané podle velikosti (od nejmenšího po největší)
- chceme na základě pole postavit BVS o optimální hloubce

Jak na to:

- chceme postavit strom z hodnot na indexech $\{l, ..., r\} = \{0, .., n-1\}$:
  - do kořene vytvářeného stromu dáme prostřední prvek pole, ten je na indexu $s = \frac{l+r}{2}$
  - jeho levý podstrom rekurzivně postavíme z ,,podpole'' na indexech $\{l_L, ..., r_L\} = \{l, ..., s-1\}$
  - jeho pravý podstrom rekurzivně postavíme z ,,podpole'' na indexech $\{l_R, ..., r_R\} = \{s+1, ..., r\}$

Postavení binárního vyhledávacího stromu (BVS) z uspořádaného pole

Jak můžeme metodu implementovat?

1. pomocí rekurzivní metody
2. nebo bez rekurze s využitím zásobníku

Jaká je časová složitost metody?

- můžeme spočítat metodou zvanou ,,kuchařka" (ukážeme si za chvíli)

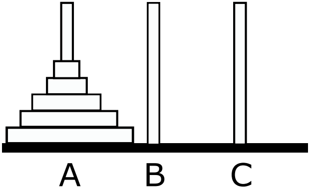
$$T(n) = 2T(\tfrac{n}{2}) + 0(1),$$
$$T(1) = 0(1)$$
$$\rightarrow \boldsymbol{T(n) = O(n)}$$

Hanojské věže (známý hlavolam)

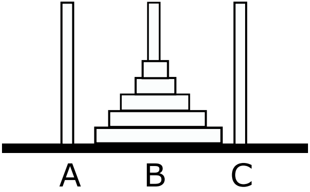
Popis úlohy

- máme 3 kolíky a n kotoučů o různých poloměrech
- **Začátek:** všechny kotouče jsou umístěné na prvním kolíku v pořadí od nějvětšího po nejměnší
- **Cíl:** všechny kotouče jsou umístěné na druhém kolíku v pořadí od nějvětšího po nejměnší

start:



cíl:



Hanojské věže (známý hlavolam)

Popis úlohy

- máme 3 kolíky a n kotoučů o různých poloměrech
- **Začátek:** všechny kotouče jsou umístěné na prvním kolíku v pořadí od nějvětšího po nejměnší
- **Cíl:** všechny kotouče jsou umístěné na druhém kolíku v pořadí od nějvětšího po nejměnší

Přesouvání kotoučů se řídí následujícími pravidly:

- vždy přesouváme jen jeden kotouč
- není povoleno odložit kotouč mimo kolíky
- není povoleno položit větší kotouč na menší.

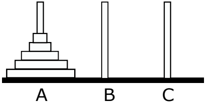
$\rightarrow$ úlohu budeme řešit metodou rozděl a panuj

Hanojské věže

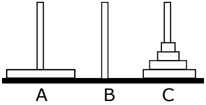
Úvaha

- V průběhu řešení musí nastat bezprostředně po sobě následující situace:
  - situace 1
  - situace 2
- Chceme přenést n kotoučů z A na B (C je pomocný):
  start →
  1. přeneseme $(n-1)$ kotoučů z A na C (B je pomocný) → situace 1
  2. přeneseme 1 kotouč z A na B → situace 2
  3. přeneseme $(n-1)$ kotoučů z C na B (A je pomocný) → cíl

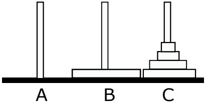
start:



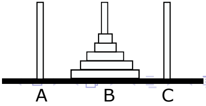
situace 1:



situace 2:



cíl:



Hanojské věže

Jak můžeme implementovat Hanojské veže?

- Každý kolík (věž) je vlastně zásobník pevné velikosti (n) $\rightarrow$ můžeme ho reprezentovat pomocí (dynamického) pole nebo spojového seznamu

Algoritmus ... rekurzivní funkce:

prenes(int n, věž Z, věž Do, Pomocná věž)
  prenes(n-1, věž Z, Pomocná věž, věž Do)
  přenes jeden kotouč z věže Z na věž Do
  prenes(n-1, Pomocná věž, věž Do, věž Z)

Jaká bude časová složitost algoritmu?

- můžeme spočítat (ukážeme si za chvíli)

$T(n) = 2T(n-1) + 0(1)$,
$T(1) = 0(1)$
$\rightarrow \boldsymbol{T(n) = O(2^n - 1) = O(2^n)}$

Master Theorem (metoda kuchařka)

- jednoduchý postup pro výpočet složitosti rekurentních metod s exponenciálním krokem,
  tj. pro implementace algoritmů typu ,,rozděl a panuj"

Samostudium

- skripta, věta 1.1 a její důkaz (strana 21-22)
- aplikace věty: velikost podmnožin (kapitola 3.1.1, strana 82-83)

Master Theorem (metoda kuchařka) (podle skript)

Nechť $a, b, c \in N$ a $f : N \rightarrow N$ je funkce, pro kterou platí $f(n) = O(n^c)$. $T(n)$ je neklesající posloupnost taková, že $\forall n : n = b^k, k \in N$ platí:

$$T(n) \leq aT(\frac{n}{b}) + f(n)$$

$$T(1) = f(1)$$

Potom

- je-li $a < b^c$, $\rightarrow T(n) = O(n^c)$ ... **typ A**
- je-li $a = b^c$, $\rightarrow T(n) = O(n^c log_b n)$ ... **typ B**
- je-li $a > b^c$, $\rightarrow T(n) = O(n^{log_b a})$ ... **typ C**

Master Theorem (metoda kuchařka) (oproti skriptům rozvolněné předpoklady na a, b, c)

Nechť $a, b, c \in R, a \geq 1, b > 1, c \geq 0$ a $f : N \to N$ je funkce, pro kterou platí $f(n) = O(n^c)$. $T(n)$ je neklesající posloupnost taková, že platí:

$$T(n) \leq aT(\frac{n}{b}) + f(n)$$

$$T(1) = f(1)$$

Potom

- je-li $a < b^c$, $\to$ $T(n) = O(n^c)$ ... **typ A**
- je-li $a = b^c$, $\to$ $T(n) = O(n^c log_b n)$ ... **typ B**
- je-li $a > b^c$, $\to$ $T(n) = O(n^{log_b a})$ ... **typ C**

Příklad I: Časová složitost binárního vyhledávání

Složitost můžeme vyjádřit následujícím rekurentním vztahem:

$$T(n) = T(\tfrac{n}{2}) + 0(1),$$
$$T(1) = 0(1)$$

Určíme parametry kuchařky a, b, c:

$$a = 1,$$
$$b = 2,$$
$$c = 0$$

Porovnáme a a b^c:

$$a = 1$$
$$b^c = 2^0 = 1$$
$$\rightarrow a = b^c, \text{ je to typ B}$$

Aplikuje kuchařku, spočteme výsledek:

$$T(n) = 0(n^c log_b n) = O(n^0 log_2 n) = O(log_2 n)$$

Příklad II: Časová složitost postavení BVS

Složitost můžeme vyjádřit následujícím rekurentním vztahem:

$T(n) = 2T(\frac{n}{2}) + 0(1)$,

$T(1) = 0(1)$

Určíme parametry kuchařky a, b, c:

$a = 2$,

$b = 2$,

$c = 0$

Porovnáme a a b^c:

$a = 2$

$b^c = 2^0 = 1$

$\rightarrow a > b^c$, je to typ C

Aplikuje kuchařku, spočteme výsledek:

$T(n) = 0(n^{log_b a}) = O(n^{log_2 2}) = O(n^1) = O(n)$

Příklady pro Vás

Spočítejte za pomoci kuchařky (Master Theorem):

1. $T(n) = 2T(\frac{n}{2}) + n$,
2. $T(n) = 9T(\frac{n}{3}) + 5$,
3. $T(n) = 3T(\frac{n}{4}) + 4n^2 - 3n - 6$,
4. $T(n) = T(\frac{2n}{3}) + \sqrt{n}$,
5. $T(n) = T(\frac{n}{4}) + 2$,

Pro všechny příklady předpokládáme $T(1) = O(1)$

Alternativy ke kuchařce

Substituční metoda

1. uhádneme řešení
2. dokážeme matematickou indukcí, že je správné

Metoda rekurzivního stromu

- spočítáme složitost celého rekurzivního stromu

Příklad III: Časová složitost algoritmu Hanojských věží

Složitost můžeme vyjádřit následujícím rekurentním vztahem:

$$T(n) = 2T(n-1) + 0(1),$$
$$T(1) = 0(1)$$

- Rekurentní krok není exponenciální, proto bohužel nemůžeme použít kuchařku
- Použijeme proto substituční metodu:
  1. uhádneme řešení
  2. dokážeme matematickou indukcí, že je správné
- Uvidíme, že se jedná o velmi časově náročný algoritmus ($T(n) = O(2^{n-1})$)
  $\rightarrow$ pro větší n je úloha těžko řešitelná v reálném čase.

Příklad III: Časová složitost algoritmu Hanojských věží

Složitost můžeme vyjádřit následujícím rekurentním vztahem:

$$T(n) = 2T(n-1) + 1,$$
$$T(1) = 1$$

(bez újmy na obecnosti jsme nahradili $O(1)$ za 1)

1. Uhádneme řešení:

$$T(1) = 1 = 2^1 - 1$$
$$T(2) = 2T(1) + 1 = 3 = 2^2 - 1,$$
$$T(3) = 2T(2) + 1 = 2 * 3 + 1 = 7 = 2^3 - 1,$$
$$T(4) = 2T(3) + 1 = 2 * 7 + 1 = 15 = 2^4 - 1,$$
$$\rightarrow \; T(n) = 2^{n-1} - 1,$$

2. Dokážeme matematickou indukcí, že $T(n) = 2^{n-1} - 1$:

$T(1) = 1 = 2^1 - 1$ ... OK

indukční předpoklad: $T(n-1) = 2^{n-2} - 1$

potom: $T(n) = 2T(n-1) + 1 = 2(2^{n-2} - 1) + 1 = 2^{n-1} - 1$

... hotovo

Příklad pro Vás

Zkuste některý z příkladů pro kuchařku (slide 17) vyřešit pomocí substituční metody