Dynamické programování

Základy algoritmizace – 11. cvičení

Zuzana Petříčková

1. května 2020

1. Rekurzivní algoritmy

- na předchozích cvičeních jsme se seznámili s celou řadou rekurzivních algoritmů i s mechanismy jak se rekurze zbavit:
  - iterativním výpočtem, pomocí while-cyklu (pokud má rekurze „jen jednu větev")
  - pomocí zásobníku nebo fronty (pokud se rekurze „více větví")

Samostudium: skripta, kapitola 4

- najdete zde formální definici rekurzivního algoritmu
- ilustrativní příklady (faktoriál, fibonacci, Ackermannova funkce)
- pak také obecný postup jak rekurzi odstranit

Jedná se o oblíbené téma u zkoušky, doporučuji zejména kapitolu 4.2 pořádně nastudovat

1. Rekurzivní algoritmy

- pokud vycházíme z rekurzivních definic nebo pracujeme s rekurzivními datovými strukturami (spojový seznam, strom), často nás to přirozeně navede k rekurzivnímu algoritmu řešení
- ale rekurzivní algoritmus často není nejleším řešením
- nyní si to demonstrujeme na několika triviálních příkladech

Příklad 1 ... faktoriál

Rekurentní vztah:

- $f(0) = f(1) = 1$
- $f(n) = n * f(n-1), n \geq 1$

Řešení s využitím rekurze

- přímá implementace rekurentního vztahu

Řešení bez rekurze (iterativně)

- použijeme pomocnou proměnnou pro uchování mezivýsledků a for-cyklus

Příklad 1 ... faktoriál

```
unsigned long long fact(unsigned int n)
{
   if (n > 1)
      return n * fact(n - 1);
   else
      return 1;
}

unsigned long long fact(unsigned int n)
{
   unsigned long long f = 1; // pomocna promenna
   for ( ; n > 1; n--)
      f *=  n;
   return f;
}
```

Příklad 1 ... faktoriál

Časová a prostorová složitost obou algoritmů

- **Řešení s využitím rekurze**
  - čas $T(n) = O(n)$, prostor $S(n) = O(n)$ (hloubka rekurze)
- **Řešení bez rekurze (iterativně)**
  - čas $T(n) = O(n)$, prostor $S(n) = O(1)$ (jedna pomocná proměnná)

$\rightarrow$ asymptoticky se algoritmy liší hlavně prostorovou složitostí

- pokud si oba programy pustíte na počítači, snadno ověříte, že pro velká n bude rekurzivní program pomalejší a mimo to spadne pro ,,nedostatek paměti na zásobníku'' pro poměrně nízké hodnoty n, se kterými si druhý program snadno poradí.

Příklad 2 ... Euklidův algoritmus

Zadání:

- máme dvě celá nezáporná čísla **a** a **b**
- chceme spočítat největšího společného dělitele čísel **a** a **b**

Euklidův algoritmus (myšlenka):

- odečteme menší číslo od většího
- opakujeme předchozí krok, dokud není jedno z čísel rovno 0
- jakmile je jedno z čísel rovno 0, je výsledkem druhé z čísel

Příklad:

12	20
12	8
4	8
4	4
4	0

Příklad 2 ... Euklidův algoritmus

Rekurentní vztah 1:

- $0 < a \leq b \rightarrow nsd(a, b) = nsd(a, b - a)$
- $0 < b < a \rightarrow nsd(a, b) = nsd(a - b, b)$
- $0 = a \leq b \rightarrow nsd(a, b) = b$
- $0 = b < a \rightarrow nsd(a, b) = a$

Rekurentní vztah 2 (vylepšení: odečítání můžeme nahradit modulem, %):

- $0 < a \leq b \rightarrow nsd(a, b) = nsd(a, b\%a)$
- $0 < b < a \rightarrow nsd(a, b) = nsd(a\%b, b)$
- $0 = a \leq b \rightarrow nsd(a, b) = b$
- $0 = b < a \rightarrow nsd(a, b) = a$

Implementace pomocí rekurze a iterativně

- na rozmyšlenou

Příklad 2 ... Euklidův algoritmus

Časová a prostorová složitost obou algoritmů
Řešení s využitím rekurze

- 1) čas $T(n) = O(a + b)$, prostor $O(a + b)$ (rekurze)

Řešení bez rekurze (iterativně)

- 1) čas $T(n) = O(a + b)$, prostor $O(1)$
- 2) čas $T(n) = O(log_2 ab)$, prostor $O(1)$

$\rightarrow$ asymptoticky se algoritmy liší hlavně prostorovou složitostí (podobně jako u pvního příkladu)

Příklad 3 ... Fibonacciho posloupnost

Rekurentní vztah:

- $f(0) = 0$
- $f(1) = 1$
- $f(n) = f(n-1) + f(n-2), n > 1$

Řešení s využitím rekurze

- přímá implementace rekurentního vztahu

Řešení bez rekurze (iterativně)

- použijeme tři pomocné proměnné pro uchování posledních tří členů posloupnosti (alternativně: použijeme pole pro uchování všech prvků posloupnosti až do n-tého prvku)

Řešení pomocí známého vzorce (Bernoulli)

- $f(n) = \frac{1}{\sqrt{5}}(\frac{1+\sqrt{5}}{2})^n - \frac{1}{\sqrt{5}}(\frac{1-\sqrt{5}}{2})^n$

Příklad 3 ... Fibonacciho posloupnost

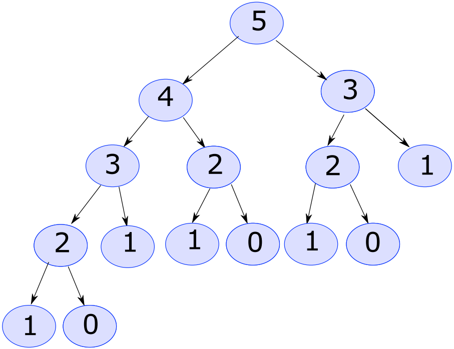
```
unsigned long long fib(unsigned int n)
{
    if (n <= 1)
        return n;
    else
        return fib(n - 1) + fib(n - 2);
}

unsigned long long fib(unsigned int n)
{
    if (n <= 1)
        return n;
    else {
        unsigned long long a = 0, b = 1, c;
        for ( ; n > 1; n--) {
            c = a + b;
            a = b;
            b = c;
        }
        return c;
    }
}
```

Příklad 3 ... Fibonacciho posloupnost

Příklad: Strom rekurze pro f(5)

- je binární
- spousta hodnot se zbytečně počítá opakovaně:



$T(n) = T(n-1) + T(n-2) + 1 > 2T(n-2) > 4T(n-4) > 8T(n-6) > \ldots > 2^k T(n-2k) = (\text{pro } k = n/2)\ 2^{n/2}$

Příklad 3 ... Fibonacciho posloupnost

Časová a prostorová složitost:

- **Řešení s využitím rekurze**
  - čas $T(n) = \Omega(2^{n/2})$, prostor $O(n)$ (hloubka rekurze)
- **Řešení bez rekurze (iterativně)**
  - čas $T(n) = O(n)$, prostor $O(1)/O(n)$ (tři pomocné proměnné/pomocné pole)

$\rightarrow$ tentokrát se algoritmy zásadně liší i časovou složitostí

- pokud si programy pustíte na počítači, snadno ověříte, že je rekurzivní program extrémně neefektivní a mimo to spadne pro ,,nedostatek paměti na zásobníku'' již pro nízké hodnoty n

Dynamické programování

- ani u jednoho z předchozích příkladů nebyla rekurze dobrým řešením
  $\rightarrow$ efektivnějším řešením byl iterativní algoritmus

- uvedené iterativní algoritmy byly triviálními příklady obecného postupu, kterému říkáme **dynamické programování**
- za chvíli si předvedeme několik typických příkladů ,,pokročilejšího" dynamického programování

Princip dynamického programování

1. máme rekurzivní algoritmus
   (typicky s exponenciální časovou složitostí)
2. zjistíme, že opakovaně voláme to samé $\rightarrow$ zbytečně
3. nahradíme rekurzi iterativním algoritmem
   (typicky s lineární nebo kvadratickou časovou složitostí),
   - využijeme pomocnou paměť (CACHE) pro uchování mezivýsledků

Princip dynamického programování

Samostudium: skripta

- **Kapitola 3.3** je zde vysvětlený princip dynamického programování na úloze hledání nejkratší cesty v síťovém grafu
- **Kapitola 6.2.2** konstrukce optimálního binárního vyhledávacího stromu s využitím dynamického programování

Příklad 4 ... automat na mince

Zadání:

- v automatu na mince je m typů mincí, např. $\{1, 3, 4\}$ (od každé mince neomezený počet)
- máme částku $n \in N$, kterou chceme rozměnit pomocí co nejméně mincí

Jak úlohu řešit?

1. hladový algoritmus:
   - použijeme vždy co největší minci, která je menší nebo rovna aktuální cílové částce, od cílové částky odečteme použitou minci
   - bude dobře fungovat pro běžně používané mince $\{1, 2, 5, 10, 20, 50\}$
   - obecně nefunguje, př. mince $\{1, 3, 4\}$ a cílová částka 6 (hladový algoritmus najde 4,1,1, optimální řešení je 3,3)
2. rekurzivní algoritmus
3. dynamické programování

Příklad 4 ... automat na mince (rekurzivní algoritmus

Značení

- $r(i)$ ... minimální počet mincí, který je potřeba na rozměnění částky i

Základní myšlenka: backtracking

- krok dopředu: zkusíme zaplatit jednou z mincí, snížíme o její hodnotu cílovou částku
- krok zpět: pokud nevede cesta k řešení, vezmeme poslední provedenou akci zpět a zkusíme jinou minci (kterou jsme ještě nezkoušeli)

Rekurentní vztah:

- $r(0) = 0$
- $r(i) = min_{m \in mince, i-m \geq 0}[r(i-m)+1], \quad 1 \leq i \leq n$
- $\quad = \infty$ (pokud částku nelze rozměnit)

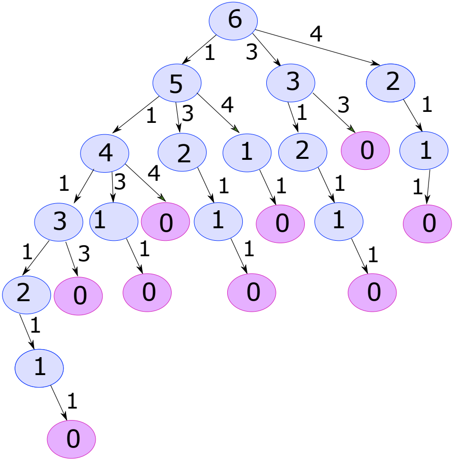
Řešení s využitím rekurze

- přímá implementace rekurentního vztahu

Příklad 4 ... automat na mince

Příklad: Strom rekurze pro mince $\{1, 3, 4\}$ a cílovou částku $n = 6$

- je ternární
- spousta hodnot se zbytečně počítá opakovaně



Příklad 4 ... automat na mince

Řešení s využitím dynamického programování

- rekurzi nahradíme iterativním výpočtem
- budeme postupně počítat hodnoty $r(i)$ pro $i = 0, \dots, n$, tyto hodnoty (mezivýsledky) budeme ukládat („cachovat") v pomocné tabulce (v poli) **a**

$$a(i) = \begin{cases} \text{minimální počet mincí potřebný pro rozměnění částky i} \\ \infty \text{ (popř. } -1\text{), pokud částku nelze rozměnit} \end{cases},$$

- v dalším poli **p** si budeme pamatovat naposledy použitou minci (abychom mohli snadno zrekonstruovat řešení)

Příklad 4 ... automat na mince

Řešení s využitím dynamického programování - příklad

- **Zadání:** mince $\{1, 3, 4\}$ a cílová částka $n = 8$
- **Řešení:**

n	0	1	2	3	4	5	6	7	8
a	0	1	2	1	1	2	2	2	2
p	-1	1	1	3	4	4	3	4	4

- **Výsledek: 2 mince : 4,4**

Příklad 4 ... automat na mince

Časová a prostorová složitost

- **Řešení s využitím rekurze**
  - čas $T(n) = O(m^n)$ (strom rekurze je m-ární)
  - prostor $S(n) = O(n)$ (hloubka rekurze)
- **Řešení s využitím dynamického programování (iterativně)**
  - čas $T(n) = O(nm)$ (při výpočtu hodnoty v poli na indexu i se díváme až na m dříve vyplněných hodnot v poli)
  - prostor $S(n) = O(n)$ (pomocná pole)

Příklad 5 ... nejdelší rostoucí podposloupnost

Zadání:

- máme pole x delky n (posloupnost čísel)
- hledáme nejdelší vybranou rostoucí podposloupnost čísel v poli

Příklad:

n	0	1	2	3	4	5	6	7	8	9
x	3	**1**	**2**	-5	**10**	8	20	**12**	**17**	13

Jak úlohu řešit?

1. hladový algoritmus:
   - procházíme prvky jeden po druhém a přidáváme je do vybrané podosloupnosti, pokud jsou větší než posledně přidaný prvek
   - nemusí najít optimální řešení (např. pro předchozí příklad algoritmus vybere podposloupnost: 3,10,20)
2. rekurzivní algoritmus
3. dynamické programování

Příklad 5 ... nejdelší rostoucí podposloupnost

Značení:

- $r(i)$ ... délka nejdelší rostoucí podposloupnosti končící **přesně** na indexu i
- nrp ... délka nejdelší rostoucí podposloupnosti končící na libovolném indexu pole

Základní myšlenka

1. spočítáme $r(i)$ pro všechna $i = 0, ..., n$
2. spočítáme $nrp = max_{j;0 \leq j < n} r(j)$

Rekurentní vztah :

$$r(i) = \begin{cases} max_{j;0 \leq j < i, a[i] > a[j]}(r(j) + 1) \\ 1 \text{ (pokud nelze prodloužit žádnou předponu)} \end{cases},$$

Řešení s využitím rekurze

- přímá implementace rekurentního vztahu

Příklad 5 ... nejdelší rostoucí podposloupnost

Řešení s využitím dynamického programování

- rekurzi nahradíme iterativním výpočtem
- budeme postupně počítat hodnoty $r(i)$ pro $i = 0, \ldots, n$, tyto hodnoty (mezivýsledky) budeme ukládat („cachovat") v pomocném poli **a**
  $a(i) =$ délka nejdelší rostoucí podposloupnosti končící na indexu i
- v dalším poli **p** si budeme pamatovat index předchůdce prvku (abychom mohli snadno zrekonstruovat řešení)
- *nrp* spočítáme stejně jako u rekurzivního algoritmu

Příklad 5 ... nejdelší rostoucí podposloupnost

Řešení s využitím dynamického programování - příklad:

- **Zadání:**

n	0	1	2	3	4	5	6	7	8	9
x	3	**1**	**2**	-5	**10**	8	20	**12**	**17**	13

- **Řešení:**

n	0	1	2	3	4	5	6	7	8	9
a	1	1	2	1	3	3	4	4	5	5
p	-1	-1	1	-1	2	2	4	4	7	7

- **Výsledek: vybraná podposloupnost má 5 prvků : 1,2,10,12,17**

(nejedná se o jedinné optimální řešení úlohy)

Příklad 5 ... nejdelší rostoucí podposloupnost

Časová a prostorová složitost:

- **Řešení s využitím rekurze**
  - přímá implementace rekurentního vztahu
  - čas $T(n) = O(2^n)$ (strom rekurze je binární)
  - prostor $S(n) = O(n)$ (hloubka rekurze)
- **Řešení s využitím dynamického programování (iterativně)**
  - čas $T(n) = O(n^2)$ (při výpočtu hodnoty v poli na indexu i se díváme na všechny předchozí hodnoty v poli)
  - prostor $S(n) = O(n)$ (pomocná pole)

Příklad 6 ... optimální binární vyhledávací strom

- u řady úloh dynamického programování si nevystačíme s jednorozměrným polem pro uchování mezivýsledků, ale je potřeba vícerozměrné pole (např. matice)
- příkladem takové úlohy je úloha konstrukce optimálního binárního vyhledávacího stromu (detaily viz skripta, kapitola 6.6.2)

Příklad 6 ... optimální binární vyhledávací strom

= BVS, který v průměru potřebuje k vyhledání klíče nejmenší počet kroků

```
Vrchol* BinarniStrom::najdiPrvek(int klic)
{
    Vrchol *vrchol = koren;
    while (vrchol != nullptr && vrchol->klic != klic)
    {
        if (vrchol->klic < klic)
            vrchol = vrchol->pravy;
        else
            vrchol = vrchol->levy;
    }
    return vrchol;
}
```

Příklad 6 ... optimální binární vyhledávací strom

Máme:

- hodnoty klíčů $x_0 < x_1 < \ldots < x_{n-1}$
- pravděpodobnosti / četnosti dotazů na klíče:
  $p_i = P(x = x_i), \quad 0 \le i < n$
  $q_i = P(x_{i-1} < x < x_i), \quad 0 \le i \le n, \; x_{-1} = -\infty, x_n = \infty$

Hledáme optimální BVS:

- BVS, který v průměru potřebuje k vyhledání klíče nejmenší počet kroků
  = BVS s nejnižší „cenou":

$$C = C_{0,n-1} = \sum_{i=0}^{n-1} p_i h(x_i) + \sum_{i=0}^{n} q_i h(e_i)$$

e_i je list stromu odpovídající intervalu (x_{i-1}, x_i), $h(v)$ je hloubka vrcholu v

Příklad 6 ... optimální binární vyhledávací strom

Základní myšlenka rekurzivního algoritmu

- rekurzivní funkce vrátí optimální strom pro hodnoty $x_i < x_{i+1} < \ldots < x_{j-1}$, vrátí zároveň jeho cenu C_{ij}

1. zvolíme kořen $x_k, i \leq k < j$ (postupně zkusíme všechny možnosti, každá z hodnot může být v kořeni)
2. zavoláme rekurzivně na levého a pravého syna
3. pro každý z kořenů x_k spočteme (s využitím rekurzivně spočítaných cen levého a pravého podstromu) cenu celého stromu s kořenem x_k
4. vrátíme strom s nejnižší cenou (a jeho cenu)

Příklad 6 ... optimální binární vyhledávací strom

Odvození rekurentní vztahu:

- chceme spočítat cenu optimálního stromu pro úsek $x_i < x_{i+1} < \ldots < x_{j-1}$:

$$C_{i,j} = \sum_{l=i}^{j-1} p_l h(x_l) + \sum_{l=i}^{j} q_l h(e_l)$$

- spočítáme cenu $C^k_{i,j}$ stromu s kořenem $x_k, i \leq k < j$:

$$C^k_{i,j} = C_{i,k} + C_{(k+1),j} + v_{i,j}$$

(cena levého podstromu + cena pravého podstromu + cena navíc)

$$v_{i,j} = \sum_{k=i}^{j-1} p_k + \sum_{k=i}^{j} q_k$$

(každý vrchol nebo list vlevo i vpravo klesl o jedničku + je zde navíc x_k v hloubce 1)

Příklad 6 ... optimální binární vyhledávací strom

Odvození rekurentní vztahu (pokračování):

- spočítáme cenu optimálního stromu:
  $C_{i,j} = min_{\{k, i \leq k < j\}} C^k_{i,j} = min_{\{k, i \leq k < j\}} (C_{i,k} + C_{(k+1),j} + v_{i,j})$,

$\rightarrow$

Rekurentní vztah:

- $C_{i,i} = v_{i,i} = q_i, \qquad 0 \leq i \leq n$
- $C_{i,j} = v_{i,j} + min_{\{k, i \leq k < j\}} (C_{i,k} + C_{k+1,j}), \qquad 0 \leq i < j \leq n$
- $v_{i,j} = \sum_{i=0}^{n-1} p_i + \sum_{i=0}^{n} q_i$

Příklad 6 ... optimální binární vyhledávací strom

Řešení s využitím rekurze

- přímá implementace rekurentního vztahu

Zjednodušeně (pokud by nám stačilo rekurzivně spočítat pouze cenu stromu):

```
double spoctiC (int i, int j)
{
    if (i > j)
        return 0;
    if (i == j)
        return q[i];
    double v = vaha(i,j);
    return "minimum pres k pro i<=k<j"
           (spoctiC(i,k)+spoctiC(k+1,j)+v);
}
```

Příklad 6 ... optimální binární vyhledávací strom

Řešení s využitím dynamického programování (iterativně)

- pro uložení hodnot C(i,j) využijeme pomocnou matici C o rozměrech $(n+1)x(n+1)$ (spodní část matice pod hlavní diagonálou zůstane nevyplněna)
- matici C postupně zaplňujeme s využitím odvozeného vzorce po diagonálách (od hlavní diagonály nahoru)
- cenu optimálního stromu přečteme jako hodnotu $C[0][n]$
- v další pomocné matici P o rozměrech $(n+1)x(n+1)$ si budeme pamatovat pro každou dvojici (i,j) index kořene optimálního podstromu pro daný úsek (abychom mohli snadno zrekonstruovat řešení)

Příklad 6 ... optimální binární vyhledávací strom

Časová a prostorová složitost:

- **Řešení s využitím rekurze**
  - čas $T(n) = \Omega(2^n)$
  - prostor $S(n) = O(n)$ (hloubka rekurze)
- **Řešení s využitím dynamického programování (iterativně)**
  - čas $T(n) = O(n^3)$
  - prostor $S(n) = O(n^2)$ (pomocná matice)

Příklad 6 ... optimální binární vyhledávací strom

Detaily:

- v případě, že by vám tato úloha nebyla moc jasná, doporučuji nahlédnout do skript (je tam vysvětlena dopodrobna)
- úlohu tento týden nedávám za domácí úkol, přesto ke zkoušce doporučuji si ji ve skriptech nastudovat