Seminár z algoritmizácie a programovania 1

Martin Bobák
Ústav informatiky
Slovenská akadémia vied

Obsah prednášky

Dynamické programovanie

Spätná väzba:
https://forms.gle/iKbuLdF6xDtNSEDP8

Dynamické programovanie

Úloha:

- riešenie náročného problému tým, že ho zjednodušíme rozdelením na menšie podproblémy, ktorých vyriešením sa dostaneme k riešeniu pôvodného problému.
- musí existovať (rekurzívny) vzťah medzi vstupným problémom a podproblémami.
- podproblémy sa prelínajú (opakujú sa -> rozdeľ a panuj generuje v každom kroku nové podproblémy)
- obmedzenia: neexistuje vzťah medzi vstupným problémom a jeho podproblémami, vysoký počet podproblémov...

Prečo dynamické programovanie?

- v 50. rokoch, keď Richard Bellman prišiel s týmto konceptom, pojem programovanie znamenal plánovanie
- dynamické programovanie označovalo metódu pre optimálne plánovanie úloh
- "vypĺňanie matice/tabuľky riešení podproblémov"

Motivácia

Zrejmé aplikácie:

- matematická optimalizácia (plánovanie, ekonomika...)
- vyhľadávanie v reťazcoch
- grafové algoritmy

Pokročilé aplikácie:

- bioinformatika (zarovnanie sekvencií)
- Markovove modely (spracovanie prirodzeného jazyka)
- počítačová grafika
- teoretické informatika

Rozdeľ a panuj

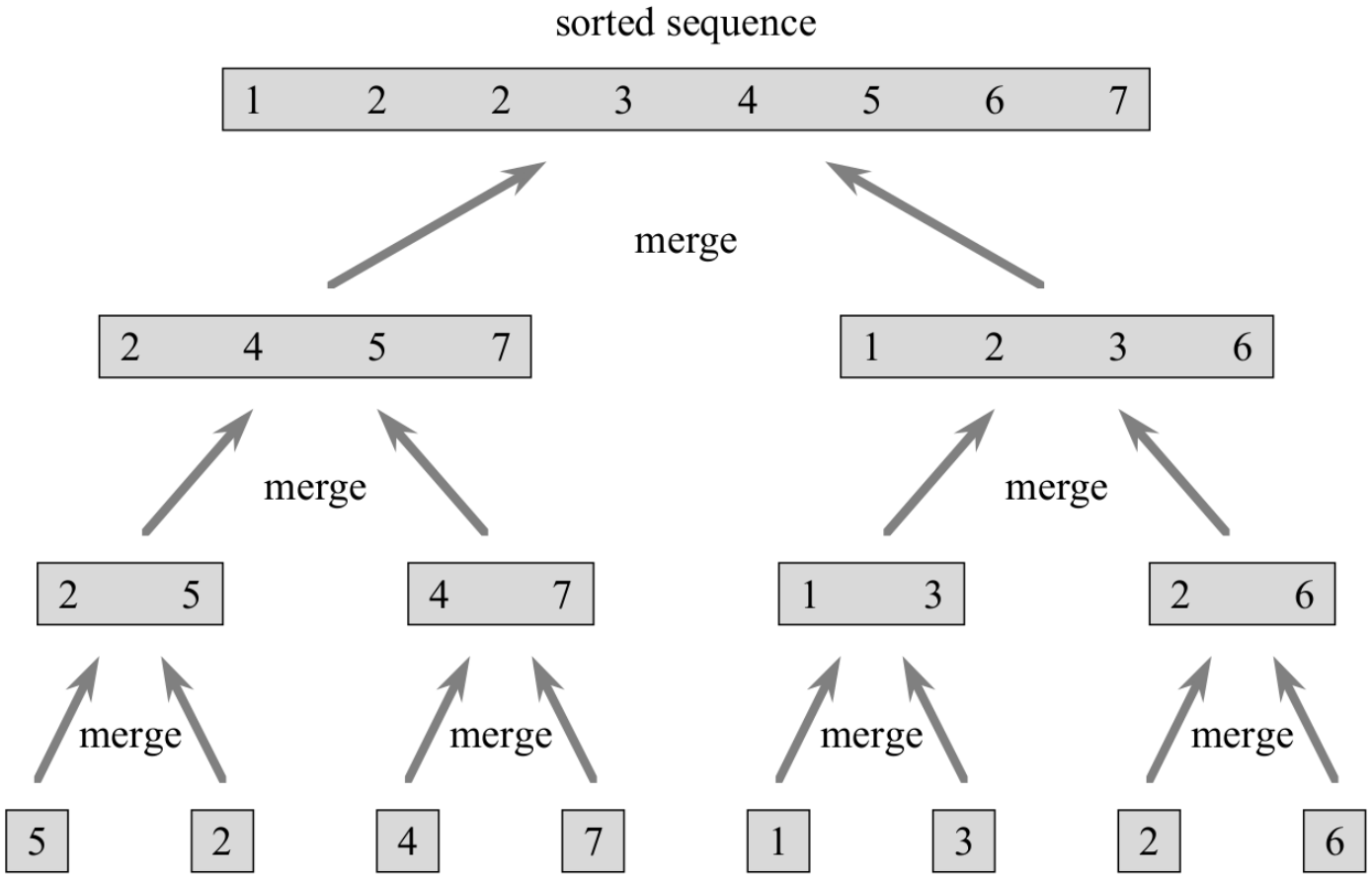
Pripomína prístup rozdeľ a panuj:

- vstupný problém rozdelíme na **disjunktné** (nezávislé, samostatné) podproblémy
- vyriešime podproblémy
- pomocou riešení podproblémov vyriešime vstupný problém

Príklady:

- triedenie (merge sort, quick sort)

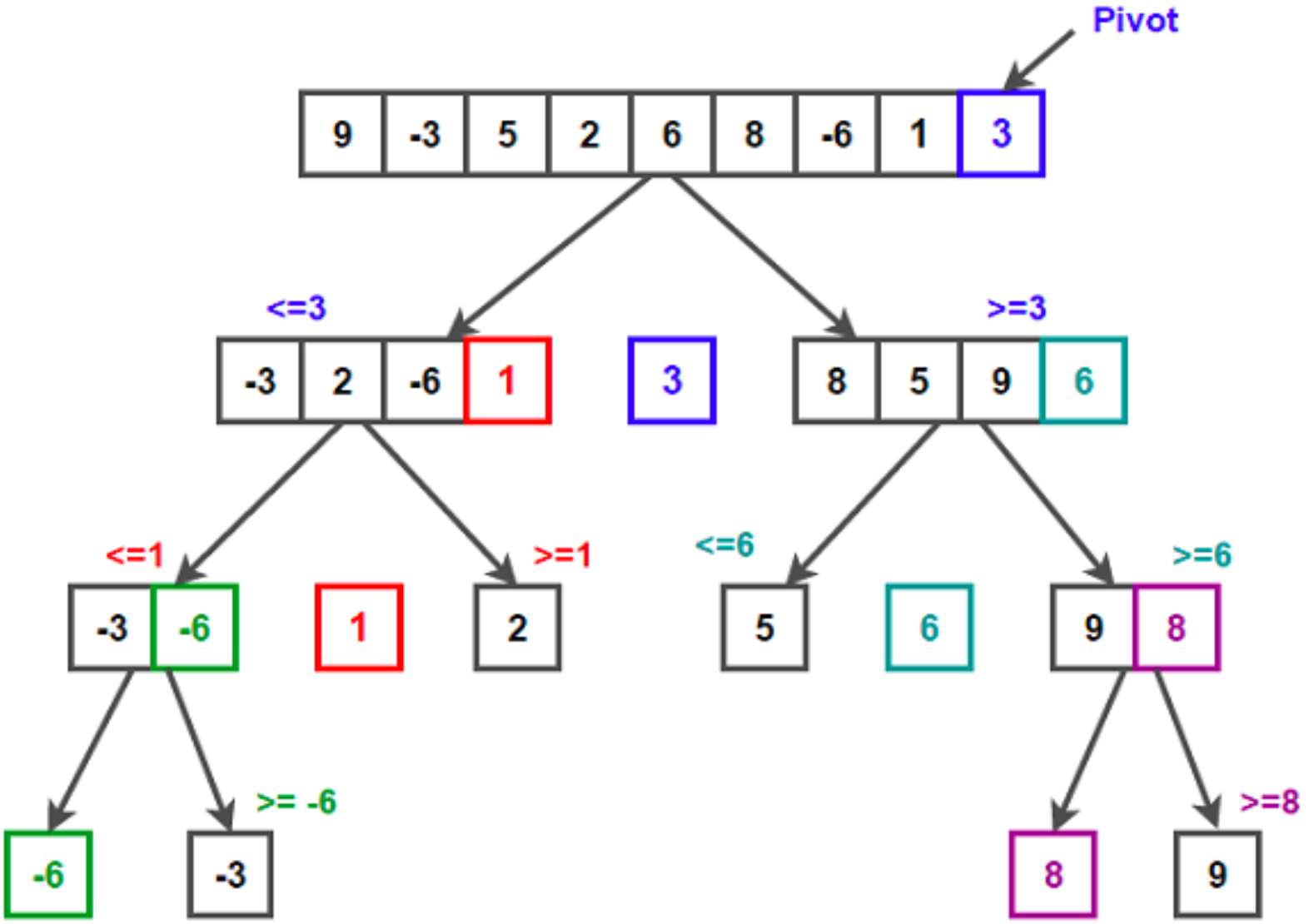
Merge sort



Zdroj: Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2009). Introduction to algorithms. MIT press.

a = {5, 2, 4, 7, 1, 3, 2, 6}

Quick sort



https://afteracademy.com/blog/quick-sort

Fibonacciho čísla

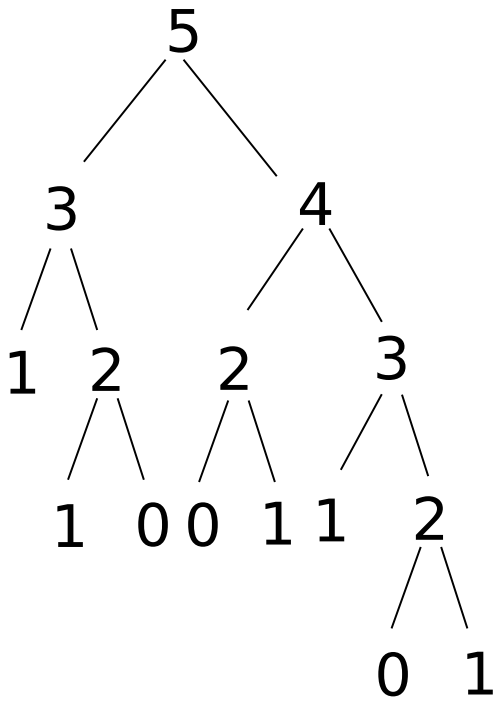
0, 1, 1, 2, 3, 5, 8, 13, 21, ...

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

- čísla v postupnosti sú súčtom dvoch predchádzajúcich

Fibonacciho čísla
Rekurzívne

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



neefektívne, pretože sa veľakrát vypočítavajú tie isté čísla – časová zložitosť: **O(2^n)**

Technika pamätania

- systematicky riešime všetky podproblémy
- riešenia už predchádzajúcich podproblémov si zapamätáme a využívame ich pri riešení nových podproblémov
  - rozdiel medzi technikou rozdeľ a panuj, zbytočne si budeme ukladať medzivýsledky, keď s nimi následne nepracujeme

Poznáme dva prístupy:

- memoizácia
- tabulácia

Memoizácia

Zhora nadol:

- problém riešime ako pri rekurzii, avšak predtým ako ideme vyriešiť podproblém, pozrieme sa do pomocnej tabuľky, či sme sa s nám už nestretli

časová zložitosť: **O(n)**
pamäťová zložitosť: **O(n)**

```
int fib(int n, int *fib_tab){
    if (fib_tab[n] == NIL){
        if (n <= 1)
            fib_tab[n] = n;
        else
            fib_tab[n] = fib(n-1, fib_tab) + fib(n-2, fib_tab);
    }

    return fib_tab[n];
}
```

Tabulácia

Zdola nahor:

- začíname od najjednoduchších podproblémov, pomocou ktorých riešime/zostavujeme komplikovanejšie varianty.
- nerekurzívne riešenie

časová zložitosť: **O(n)**
pamäťová zložitosť: **O(n)**

```
int fib(int n)
{
    int f[n+1];
    int i;
    f[0] = 0; f[1] = 1;
    for (i = 2; i <= n; i++)
        f[i] = f[i-1] + f[i-2];

    return f[n];
}
```

Fibonacciho čísla iteratívne

```
int fib(int n)
{
   if (n <= 1)
      return n;
   else {
      int n_1, n_2, i, sucet;

      n_1 = 0;
      n_2 = 1;

      for(i=2; i<=n; i++) {
         sucet = n_1 + n_2;
         n_1 = n_2;
         n_2 = sucet;
      }
      return sucet;
   }
}
```

niekedy si nepotrebujeme pamätať celú tabuľku, ale iba jej časť (napr. posledný riadok, niektoré hodnoty)

časová zložitosť: **O(n)**

pamäťová zložitosť: **O(1)**

Zhrnutie

- pri memoizácii uschovávame podproblémy, na ktoré narazíme počas riešenia vstupného problému
  - niektoré podproblémy vieme preskočiť
  - rekurzívne riešenie si vyžaduje minimálne zásahy
- pri tabulácii začíname so základnými podproblémami, ktoré vieme vyriešiť a postupne/systematicky riešime všetky podproblémy.
  - je potrebné pretransformovať rekurzívne riešenie na nerekurzívne riešenie

Najdlhšia spoločná podpostupnosť

- podobnosť reťazcov
  - diff, história zmien (Git)
  - bioinformatika

Old revision	New revision
2010-10-31 17:10:03 by admin	2010-10-31 17:31:03 by admin
this is some text that will be changed	this is the changed text

- X = ABCD
  Y = ACBAD
  - AB, AC, AD, BD
  - ABD, ACD

Najdlhšia spoločná podpostupnosť

Vstup: dva reťazce (polia znakov)

$$X = (X_1, X_2...X_m), Y = (Y_1, Y_2...Y_n)$$

Výstup: reťazec reprezentujúci najdlhšiu spoločnú podpostupnosť

Podproblém: C[i,j] = najdlhšia spoločná podpostupnosť reťazcov $X = (X_1, X_2...X_i)$, $Y = (Y_1, Y_2...Y_j)$.

C[m,n] = riešenie pre vstupné reťazce

Najdlhšia spoločná podpostupnosť

$$LCS(X_i, Y_j) = \begin{cases} \emptyset & \text{if } i = 0 \text{ or } j = 0 \\ LCS(X_{i-1}, Y_{j-1}) \hat{} x_i & \text{if } i, j > 0 \text{ and } x_i = y_j \\ \max\{LCS(X_i, Y_{j-1}), LCS(X_{i-1}, Y_j)\} & \text{if } i, j > 0 \text{ and } x_i \neq y_j. \end{cases}$$

- okraje tabuľky inicializujeme na 0
- postupne vypĺňame tabuľku podľa okolitých hodnôt
  - C[i, j] určíme podľa C[i−1, j], C[i, j−1] a C[i−1, j−1]

Najdlhšia spoločná podpostupnosť

LCS Strings

	Ø	A	G	C	A	T
Ø	Ø	Ø	Ø	Ø	Ø	Ø
G	Ø					
A	Ø					
C	Ø					

Najdlhšia spoločná podpostupnosť

"G" Row Completed

	Ø	A	G	C	A	T
Ø	Ø	Ø	Ø	Ø	Ø	Ø
G	Ø	↑ ←Ø	↖(G)	←(G)	←(G)	←(G)
A	Ø					
C	Ø					

Najdlhšia spoločná podpostupnosť

"G" & "A" Rows Completed

	Ø	A	G	C	A	T
Ø	Ø	Ø	Ø	Ø	Ø	Ø
G	Ø	↑←Ø	↖(G)	←(G)	←(G)	←(G)
A	Ø	↖(A)	↑←(A) & (G)	↑←(A) & (G)	↖(GA)	←(GA)
C	Ø					

Najdlhšia spoločná podpostupnosť

Completed LCS Table

	Ø	A	G	C	A	T
Ø	Ø	Ø	Ø	Ø	Ø	Ø
G	Ø	↑←Ø	↖(G)	←(G)	←(G)	←(G)
A	Ø	↖(A)	↑←(A) & (G)	↑←(A) & (G)	↖(GA)	←(GA)
C	Ø	↑(A)	↑←(A) & (G)	↖(AC) & (GC)	↑←(AC) & (GC) & (GA)	↑←(AC) & (GC) & (GA)

Najdlhšia spoločná podpostupnosť

Storing length, rather than sequences

	Ø	A	G	C	A	T
Ø	0	0	0	0	0	0
G	0	↑←0	↖1	←1	←1	←1
A	0	↖1	↑←1	↑←1	↖2	←2
C	0	↑1	↑←1	↖2	↑←2	↑←2

Najdlhšia spoločná podpostupnosť

Traceback example

	Ø	A	G	C	A	T
Ø	0	**0**	0	0	0	0
G	0	↑←0	↖**1**	←**1**	←1	←1
A	0	↖1	↑←1	↑←1	↖**2**	←**2**
C	0	↑1	↑←1	↖2	↑←2	↑←**2**

Najdlhšia spoločná podpostupnosť

```
int LCS(char* X, char* Y, int m, int n){
    int C[m + 1][n + 1];
    int result = 0;

    for (int i = 0; i <= m; i++) {
        for (int j = 0; j <= n; j++) {
            if (i == 0 || j == 0)
                C[i][j] = 0;

            else if (X[i - 1] == Y[j - 1]) {
                C[i][j] = C[i - 1][j - 1] + 1;
                result = max(result, C[i][j]);
            }
            else
                C[i][j] = 0;}}
    return result;}
```

Námety na semestrálnu prácu

- nájdite si úlohu vhodnú pre dynamické programovanie a porovnajte zložitosti jednotlivých prístupov

Zdroje

Dynamické programovanie:

https://www.geeksforgeeks.org/overlapping-subproblems-property-in-dynamic-programming-dp-1/

Najdlhšia spoločná podpostupnosť:

https://en.wikipedia.org/wiki/Longest_common_subsequence_problem

Ďakujem vám za pozornosť!

Spätná väzba:
https://forms.gle/iKbuLdF6xDtNSEDP8