Dynamické programovanie

- dynamické programovanie je ďalšia technika návrhu efektívnych algoritmov
- pomocou neho vieme častokrát riešiť úlohy na prvý pohľad exponenciálnej zložitosti v polynomiálnom čase
- algoritmy používajúce dynamické programovanie sú obyčajne pomerne jednoduché, nevyžadujú veľa riadkov kódu
- primárne sa používa na optimalizačné problémy
- slovo 'programovanie' v názve zaviedol autor metódy Richard Bellman, neznamená písanie kódu (v roku 1950 bola informatika v plienkach), ale znamená (optimálne) plánovanie

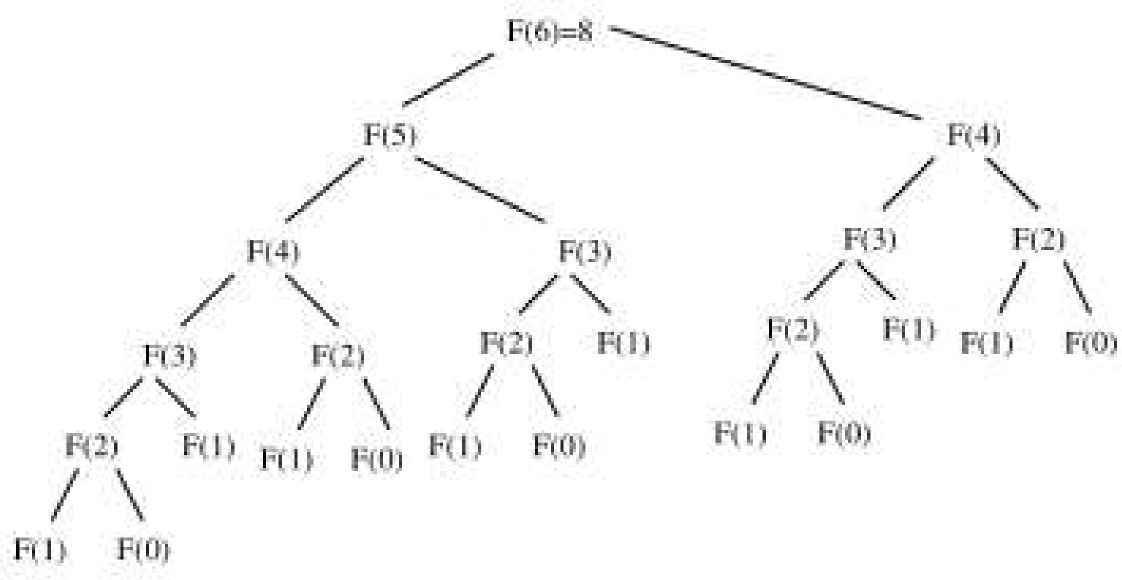
Dynamické programovanie

Podobne ako pri metóde *divide and conquer*, aj tu riešime problém vyriešením niekoľkých podproblémov menšieho rozsahu, z ktorých vieme dostať riešenie celého problému. DP je možné použiť na výpočtový problém, ak tento spĺňa :

- **jednoduché podproblémy :** podproblémy sú rovnakej štruktúry (ale menšieho rozsahu), obvykle sa dajú definovať pomocou zopár indexov
- **optimalita podproblémov :** optimálne riešenie globálneho problému dostaneme kompozíciou z optimálnych riešení podproblémov
- **prekrývanie podproblémov :** obvykle je potrebné použiť riešenie podproblému viackrát, preto sa tieto riešenia odkladajú do tabuľky a v prípade potreby odtiaľ vezmú (počítajú sa len raz) - **memoizácia** - vymieňame čas za priestor

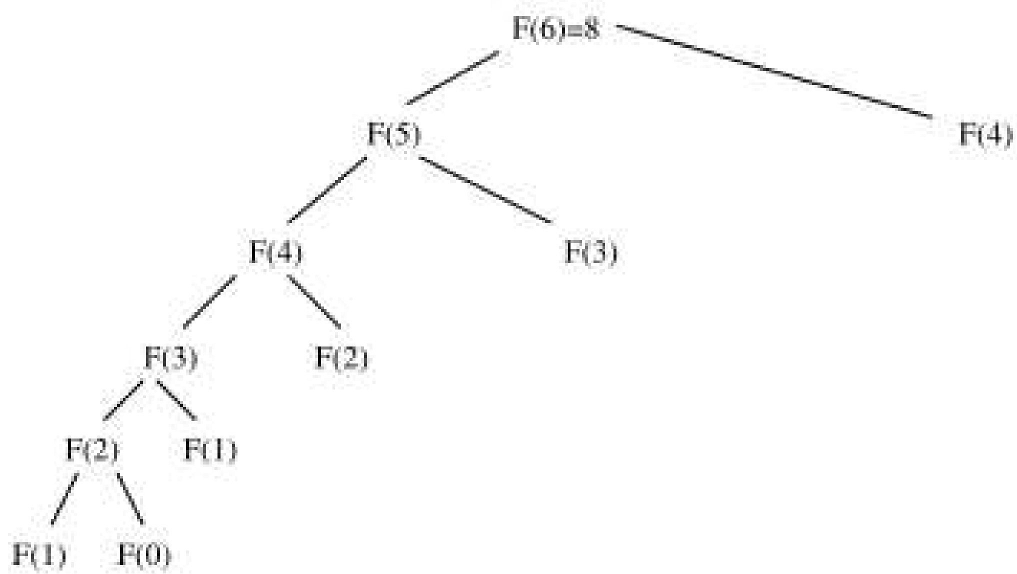
Príklad - rekurzívny Fibonacci

```
long fibonacci( int n) {
  if ( n==0 ) return 0;
  if ( n==1 ) return 1;
  return fibonacci(n–1) + fibonacci(n–2);
}
```



Príklad - rekurzívny Fibonacci - memoizácia(caching)

```
long F[MAXN+1];        // F[0]=0, F[1]=1, F[i]=UNKNOWN i>1
long fibonacci(int n) {
   if ( F[n] == UNKNOWN ) F[n] = fibonacci(n−1) + fibonacci(n−2);
  return F[n];
}
```



Obrázok: Pre každú hodnotu n sa volá fibonacci(n) presne 2x

Príklad - Fibonacci - DP zdola nahor

Fibonacci iteračne zdola nahor :

```
long fibonacci(int n) {
  int i;
  long F[MAXN+1];
  F[0] = 0;
  F[1] = 1;
  for (i=2; i<=n; i++)
    F[i] = F(i-1) + F(i-2);
  return F[n];
}
```

Hoci výpočet Fibonacciho čísel nie je ideálnym príkladom použitia DP (nie je to optimalizačná úloha),
pre jednoduchosť problému sa dá na ňom dobre demonštrovať aplikácia DP.

4 kroky pri návrhu DP algoritmu

- charakterizovanie štruktúry optimálneho riešenia
- rekurzívne definovanie hodnôt optimálneho riešenia
- výpočet hodnôt optimálneho riešenia
- konštrukcia optimálneho riešenia z vypočítanej informácie

Najdlhšia rastúca podpostupnosť

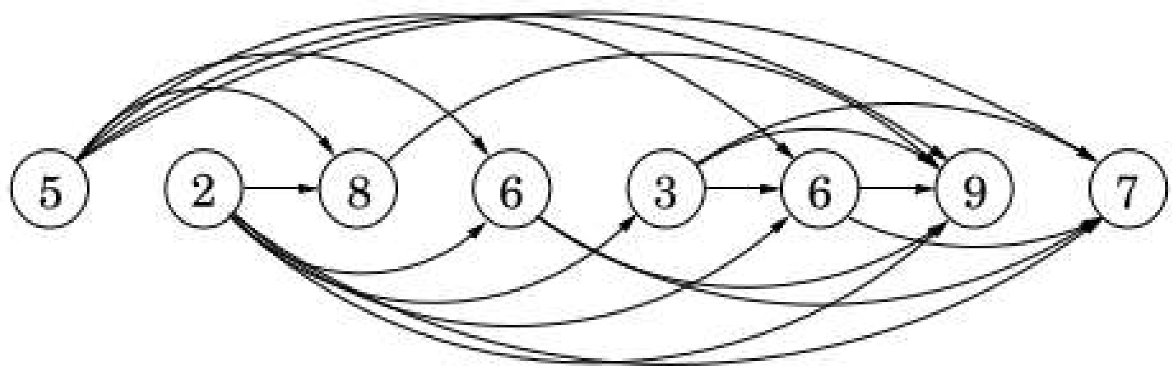
Je daná postupnosť čísel $a_1, a_2, \ldots, a_n$. Jej podpostupnosť je $a_{i_1}, a_{i_2}, \ldots, a_{i_k}$, kde platí $1 \leq i_1 < i_2 < \cdots < i_k \leq n$.
V rastúcej postupnosti platí, že $a_i < a_j$ ak $i < j$.
Úlohou je nájsť najdlhšiu rastúcu podpostupnosť danej postupnosti.
Hrubou silou : zobrať všetky podpostupnosti, ktorých je 2^n, každú testovať na monotónnosť a pamätať si maximum.

Pre ľahšie pochopenie DP algoritmu zostrojme takýto linearizovaný DAG :

- vrcholy v rade reprezentujú prvky postupnosti
- prvý vrchol má označenie 1 a posledný označenie n
- orientovaná hrana z vrchola i do vrcholu j ($i < j$) existuje práve vtedy, ak $a_i < a_j$

Rekurencia

Postupnosť : 5, 2, 8, 6, 3, 6, 9, 7.



Obrázok: DAG z postupnosti, pre názornosť vo vrchole i je a_i

Nech $L(j)$ je dĺžka najdlhšej rastúcej podpostupnosti končiacej prvkom a_j. Potom platí :

$$L(j) = 1 + \max_{(i,j) \in E} \{ L(i) \}$$

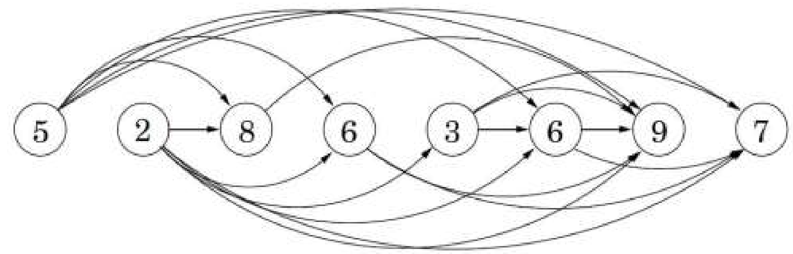
Algoritmus

```
// max{ ∅ } = 0
for j=1 to V do
   L(j) = 1 + max{ L(i) : (i,j) ∈ E }
   P(j) = m // L(m) = max{ L(i) }
return max{ L(j) : 1 ≤ j ≤ n }
```

Zložitosť : $\sum_{j=1}^{n} \sum_{i=1}^{j-1} \Theta(1) = \mathcal{O}(n^2)$

- prechádzame cez všetky vrcholy j, $1 \leq j \leq n$
- v každom vrchole musíme prejsť cez všetky predchádzajúce a testovať, pre ktoré i, $1 \leq i < j$ je $a_i < a_j$
- tieto zobrať do úvahy pre výpočet maxima
- záverečné maximum je $\mathcal{O}(n)$
- z hodnôt $P(j)$ vieme skonštruovať riešenie

Riešenie príkladu



- 5 **2** 8 6 **3** **6** **9** 7
- 5 **2** 8 6 **3** **6** 9 **7**

for $j = 1, 2, \ldots, n$:
$L(j) = 1 + \max\{L(i) : (i,j) \in E\}$
return $\max_j L(j)$

j		Predchodca/vrchol	L(j)	
1	5		1	
2	2		1	
3	8	1,2	2	1+Max{1,1}
4	6	1,2	2	1+Max{1,1}
5	3	2	2	1+Max{1}
6	6	1,2,5	3	1+Max{1,1,2}
7	9	1,2,3,4,5,6	4	1+Max{1,1,2,2,2,3}
8	7	1,2,4,5,6	4	1+Max{1,1,2,2,3}

Problém ruksaku

- ruksak má kapacitu W ($W \in \mathcal{N}$)
- do ruksaku si vyberáme z n druhov predmetov, pričom i-ty druh predmetu má váhu $w_i \in \mathcal{N}$ a hodnotu v_i
- úlohou je zaplniť ruksak tak, aby celková hodnota predmetov v ňom bola čo najväčšia, pričom sa ale neprekročí jeho kapacita (váha vecí v ňom)
- hrubou silou : $\mathcal{O}(2^n)$
- problém je NP-ťažký, ale existuje pseudo-polynomiálny DP algoritmus zložitosti $\mathcal{O}(n\,W)$
- sú 2 základné varianty problému :
  - problém ruksaku bez opakovania (každý druh je v ruksaku najviac raz)
  - problém ruksaku s opakovaním

0-1 knapsack : ruksak bez opakovania

- z každého druhu môžme dať do ruksaku najviac jeden
- nech $K(j, w)$ je maximálna hodnota vecí v ruksaku, ak vyberáme z prvých j druhov predmetov a kapacita ruksaku je w, $\quad (0 \le j \le n,\ 0 \le w \le W)$
- cieľom je nájsť hodnotu $K(n, W)$
- j-ty druh predmetu do ruksaku buď dáme alebo nedáme, vyberieme si lepšiu z týchto 2 možností

$$K(j, w) = \max\{K(j-1, w),\ K(j-1, w-w_j) + v_j\}$$

$$K(0, w) = K(j, 0) = 0 \qquad K(j, w) = -\infty \text{ ak } w < 0$$

Riešenie príkladu

Nech dvojice (v_j, w_j) postupne sú :
$(1,1),\ (2,2),\ (4,3),\ (4,4),\ (5,5).$
$W = 10,\ n = 5$

$$K(j,w) = \max\{K(j-1,w),\ K(j-1,w-w_j)+v_j\}$$

j / w	0	1	2	3	4	5	6	7	8	9	10
0	0	0	0	0	0	0	0	0	0	0	0
1	0	1	1	1	1	1	1	1	1	1	1
2	0	1	2	3	3	3	3	3	3	3	3
3	0	1	2	4	5	6	7	7	7	7	7
4	0	1	2	4	5	6	7	8	9	10	11
5	0	1	2	4	5	6	7	8	9	10	11

V optimálnom riešení vyberieme predmety : 2, 3, 5 alebo 1, 2, 3, 4.

Analýza zložitosti

- inicializácia $K(j, w)$: nultý riadok aj stĺpec sú nuly
- vypĺňame tabuľku po riadkoch, zľava doprava
- každé okienko tabuľky nás stojí $\Theta(1)$
- keďže počet okienok na vyplnenie je $n\,W$, celková zložitosť je $\mathcal{O}(n\,W)$ - pseudo-polynomiálny algoritmus
- pseudo-polynomiálny algoritmus - čas výpočtu závisí polynomiálne od veľkosti čísla na vstupe, nie polynomiálne od veľkosti vstupu ($2n + 1$ čísel, štandardne binárne kódovaných)
- napríklad ak by $W = 2^n$, máme horší algoritmus ako je metódou hrubej sily

Ruksak s opakovaním

- máme potenciálne neobmedzené množstvo predmetov každého druhu
- je dovolené vziať viac ako jeden rovnaký
- problém sa dá riešiť podobne ako verzia bez opakovania
- $K(j, w)$ je slovne definované rovnako, len môže byť viac predmetov rovnakého druhu, cieľ : $K(n, W)$
- j-ty druh predmetu v ruksaku buď nie je alebo je, vyberieme si lepšiu z týchto 2 možností

$$K(j, w) = \max\{K(j-1, w),\ K(j, w - w_j) + v_j\}$$

$$K(0, w) = K(j, 0) = 0 \qquad K(j, w) = -\infty \text{ ak } w < 0$$

Ruksak s opakovaním - iné riešenie

Iný spôsob riešenia problému ruksaku :

- nech $K(w)$ je maximálna hodnota vecí v ruksaku, ak je kapacita ruksaku w $(0 \leq w \leq W)$
- chceme poznať $K(W)$
- ak je kapacita ruksaku w, vyskúšajme dať doňho postupne ako prvé všetky druhy predmetov (pokiaľ sa tam zmestia), zvyšnú kapacitu ruksaku zaplňme optimálne a vyberme z týchto možností najlepšiu

$$K(w) = \max_{1 \leq i \leq n,\ w \geq w_i} \{v_i + K(w - w_i)\}$$

$$K(0) = 0$$

Zložitosť : $\mathcal{O}(n\,W)$.

Príklad

Nech dvojice (v_j, w_j) sú postupne :
$(30, 6),\ (14, 3),\ (16, 4),\ (9, 2)$.
$W = 10,\ n = 4$

$$K(w) = \max_{1 \leq i \leq n,\ w \geq w_i} \{v_i + K(w - w_i)\}$$

0	1	2	3	4	5	6	7	8	9	10
0	0	9	14	18	23	30	32	39	44	**48**

Maximálna cena je 48, vyberieme 2x štvrtý a 1x prvý.

Na určenie, ktoré predmety sú koľkokrát v ruksaku si treba pamätať (pre každé w) pre ktoré i dostávame maximálnu hodnotu. Táto verzia je pamäťovo efektívnejšia.

Ruksak s opakovaním - top-down vs bottom-up

Prístup zhora nadol :

```
// K(0) = 0 je vypocitane
function knapsack_td(w)
  if ( K(w) je vypocitane )
    return K(w)
  K(w) = max {knapsack_td(w − w_j) + v_j : w_j ≤ w}
  return K(w)
```

Prístup zdola nahor :

```
function knapsack_bu(w)
  K(0) = 0
  for w=1 to W do
    K(w) = max {K(w − w_j) + v_j : w_j ≤ w}
  return K(W)
```

Porovnanie prístupov

- zložitosť oboch verzií je $\mathcal{O}(n\,W)$
- konštanta $\mathcal{O}(n\,W)$ v top-down verzii môže byť väčšia vzhľadom na náklady na rekurziu
- bottom-up býva rýchlejší aj z dôvodu rýchleho prístupu k relatívne blízkym hodnotám v tabuľke
- výhodou rekurzívnej verzie je, že sa počítajú len skutočne možné váhy predmetov v ruksaku (v bottom-up verzii sa vždy vypĺňa celá tabuľka)
- rekurzívny prístup môže byť ľahšie predstaviteľný/programovateľný
- avšak pre veľké n môže nastať stack overflow