

1

Técnicas de Concepção de Algoritmos (1ª parte): Programação Dinâmica

R. Rossetti, A. P. Rocha, L. Ferreira, J. P. Fernandes, F. Ramos, G. Leão
CAL, MIEIC, FEUP

Técnicas de Concepção de Algoritmos, CAL - MIEIC/FEUP

2

Programação dinâmica (*dynamic programming*)

Técnicas de Concepção de Algoritmos, CAL - MIEIC/FEUP

3

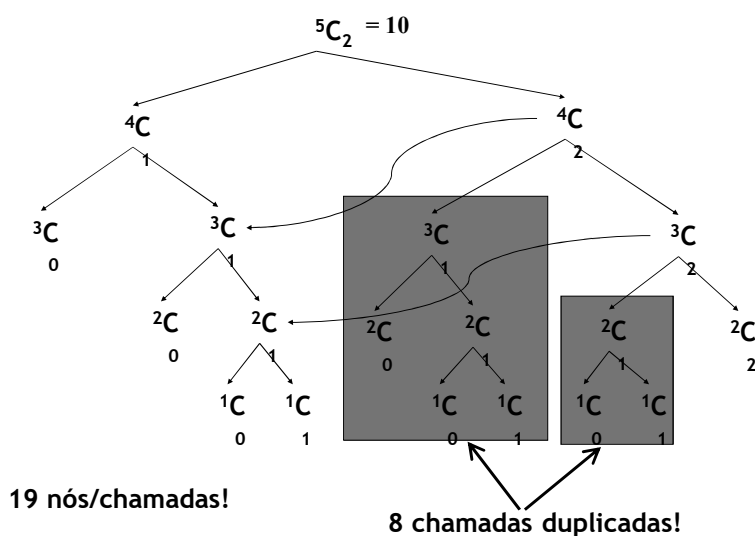
Aplicabilidade e abordagem

- ◆ Problemas resolúveis recursivamente (solução é uma combinação de soluções de subproblemas similares)
- ◆ ... Mas, em que a resolução recursiva directa duplicaria trabalho (resolução repetida do mesmo subproblema)
- ◆ Abordagem:
 - 1º) Economizar tempo (evitar repetir trabalho), memorizando as soluções parciais dos subproblemas (gastando memória!)
 - 2º) Economizar memória, resolvendo subproblemas por ordem que minimiza nº de soluções parciais a memorizar (*bottom-up*, começando pelos casos base)
- ◆ Termo “Programação” vem da Investigação Operacional, no sentido de “formular restrições ao problema que o tornam num método aplicável” e autocontido, de decisão.

Técnicas de Concepção de Algoritmos, CAL - MIEIC/FEUP

4

nC_k - repetição de trabalho



Técnicas de Concepção de Algoritmos, CAL - MIEIC/FEUP

5

Exemplo: nC_k , versão recursiva

```
int comb(int n, int k) {
    if (k == 0 || k == n)
        return 1;
    else
        return comb(n-1, k) + comb(n-1, k-1);
}
```

- Executa ${}^nC_{k-1}$ vezes (nº de somas a efectuar é nº de parcelas -1)
- Executa nC_k vezes (nº de 1's / parcelas que é preciso somar ...)
- Executa $2{}^nC_k - 1$ vezes para calcular nC_k !!

Pode-se melhorar muito, evitando repetição de trabalho (cálculos intermédios nC_j)

Técnicas de Concepção de Algoritmos, CAL - MIEIC/FEUP

6

nC_k - Programação dinâmica

Memorização de soluções parciais:

nC_k	k=0	k=1	K=2	K=3	K=4	k=5
n=0	1					
n=1	1	1				
n=2	1	2	1			
n=3	1	3	3	1		
n=4	1	4	6	4	1	
n=5	1	5	10	10	5	1

Técnicas de Concepção de Algoritmos, CAL - MIEIC/FEUP

7

Implementação

Guardar apenas uma coluna, e calcular da esq. para dir.
(também se podia guardar 1 linha e calc. cima p/ baixo):

```
static final int MAXN = 50;
static int c[] = new int[MAXN+1];

int comb(int n, int k) {
    int maxj = n - k;
    for (int j = 0; j <= maxj; j++)
        c[j] = 1;
    for (int i = 1; i <= k; i++)
        for (int j = i; j <= maxj; j++)
            c[j] += c[j-1];
    return c[maxj];
}
```

 $T(n,k) = O(k(n-k))$
 $S(n,k) = O(n-k)$
 $\rightarrow n-k+1$ vezes

 $\rightarrow k(n-k)$ vezes

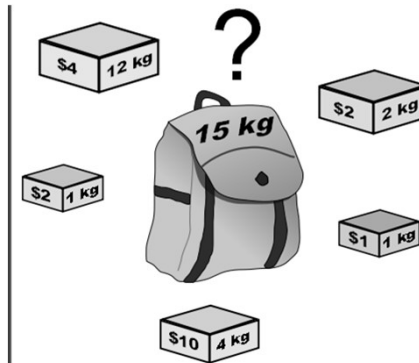
Técnicas de Concepção de Algoritmos, CAL - MIEIC/FEUP

8

Problema da mochila

- ◆ Um ladrão encontra o cofre cheio de itens de vários tamanhos e valores, mas tem apenas uma mochila de capacidade limitada; qual a combinação de itens que deve levar para maximizar o valor do roubo?

- Tamanhos e capacidades inteiros
- Vamos assumir n° ilimitado de itens de cada tipo



Técnicas de Concepção de Algoritmos, CAL - MIEIC/FEUP

9

Exemplo

Itens

Tamanho	3	4	7	8	9
Valor	4	5	10	11	13
Nome	A	B	C	D	E

Capacidade da mochila: 17

Uma
solução
óptima:

3

+

7

+

7

4 10 10

A C C

Outra
solução
óptima:

8

+

9

11 13

D E

Técnicas de Concepção de Algoritmos, CAL - MIEIC/FEUP

10

Estratégia de prog. dinâmica

- ◆ Calcular a melhor combinação para todas as mochilas de capacidade 1 até M (capacidade pretendida)
- ◆ Começar por considerar que só se pode usar o item 1, depois os itens 1 e 2, etc., e finalmente todos os itens de 1 a N (N = nº de itens)
- ◆ Cálculo é eficiente em tempo e espaço se efectuado pela ordem apropriada

Técnicas de Concepção de Algoritmos, CAL - MIEIC/FEUP

11

Dados

- ◆ Entradas:
 - N - nº de itens (com nº de cópias ilimitado de cada item)
 - $\text{size}[i]$ ($1 \leq i \leq N$) - tamanho (inteiro) do item i
 - $\text{val}[i]$ ($1 \leq i \leq N$) - valor do item i
 - M - capacidade da mochila (inteiro)
- ◆ Dados de trabalho, no final de cada iteração i (de 0 a N)
 - $\text{cost}[k]$ ($1 \leq k \leq M$) - melhor valor que se consegue com mochila de capacidade k , usando apenas itens de 1 a i
 - $\text{best}[k]$ ($1 \leq k \leq M$) - último item seleccionado p/ obter melhor valor com mochila de capac. k , usando apenas itens de 1 a i
- ◆ Dados de saída:
 - $\text{cost}[M]$ - melhor valor que se consegue c/ mochila de cap. M
 - $\text{best}[M]$, $\text{best}[M - \text{size}[\text{best}[M]]]$, etc. - itens seleccionados

Técnicas de Concepção de Algoritmos, CAL - MIEIC/FEUP

12

Formulação recursiva

- ◆ Caso base ($i = 0$; $k = 1, \dots, M$):

$\text{cost}[k]^{(0)} = 0$
 $\text{best}[k]^{(0)} = 0$

- ◆ Caso recursivo ($i = 1, \dots, N$; $k = 1, \dots, M$) :

$$\text{cost}[k]^{(i)} = \begin{cases} \text{val}[i] + \text{cost}[k - \text{size}[i]]^{(i)}, & \text{se } \begin{cases} \text{size}[i] \leq k \\ \text{val}[i] + \text{cost}[k - \text{size}[i]]^{(i)} > \text{cost}[k]^{(i-1)} \end{cases} \\ \text{cost}[k]^{(i-1)}, & \text{no caso contrário} \end{cases}$$

$$\text{best}[k]^{(i)} = \begin{cases} i, & \text{no primeiro caso acima (usa o item } i) \\ \text{best}[k]^{(i-1)}, & \text{no segundo caso acima (não usa o item } i) \end{cases}$$

Encher o resto

 Permite usar repetidamente o item i
 (senão, escrevíamos $i-1$)

Técnicas de Concepção de Algoritmos, CAL - MIEIC/FEUP

13

Evolução dos dados de trabalho

i	size	val
0	-	-
1	3	4
2	4	5
3	7	10
4	8	11
5	9	13

Informação sobre os elementos a incluir na mochila

i é o label dado a cada elemento

i = 0 é utilizado para inicializar o procedimento

Técnicas de Concepção de Algoritmos, CAL - MIEIC/FEUP

14

Evolução dos dados de trabalho

i	size	val	k	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	
0	-	-	cost[k]	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	
			best[k]	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	
1	3	4	<p>Iteração:</p> <p>i = 0;</p> <p>0 <= k <= M;</p>																			
2	4	5																				
3	7	10																				
4	8	11																				
5	9	13																				

Técnicas de Concepção de Algoritmos, CAL - MIEIC/FEUP

15

Evolução dos dados de trabalho

i	size	val	k	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
0	-	-	cost[k]	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
			best[k]	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	3	4	cost[k]	0	0	0	4	4	4	8	8	8	12	12	12	16	16	16	20	20	20
			best[k]	0	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
2	4	5	Iteração: i = 1; size[i] <= k <= M; size[i] <= k AND val[i] + cost[k-i] > cost[k]? THEN best[k] = i; cost[k] = val[i] + cost[k-i];																		
3	7	10																			
4	8	11																			
5	9	13																			

Técnicas de Concepção de Algoritmos, CAL - MIEIC/FEUP

16

Evolução dos dados de trabalho

i	size	val	k	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
0	-	-	cost[k]	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
			best[k]	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	3	4	cost[k]	0	0	0	4	4	4	8	8	8	12	12	12	16	16	16	20	20	20
			best[k]	0	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
2	4	5	cost[k]	0	0	0	4	5	5	8	9	10	12	13	14	16	17	18	20	21	22
			best[k]	0	0	0	1	2	2	1	2	2	1	2	2	1	2	2	1	2	2
3	7	10	cost[k]	0	0	0	4	5	5	8	10	10	12	14	15	16	18	20	20	22	24
			best[k]	0	0	0	1	2	2	1	3	2	1	3	3	1	3	3	1	3	3
4	8	11	cost[k]	0	0	0	4	5	5	8	10	11	12	14	15	16	18	20	21	22	24
			best[k]	0	0	0	1	2	2	1	3	4	1	3	3	1	3	3	4	3	3
5	9	13	cost[k]	0	0	0	4	5	5	8	10	11	13	14	15	17	18	20	21	23	24
			best[k]	0	0	0	<u>1</u>	2	2	1	3	4	5	<u>3</u>	3	5	3	3	4	5	<u>3</u>

Técnicas de Concepção de Algoritmos, CAL - MIEIC/FEUP

17

Codificação

Tempo: $T(N,M) = O(NM)$ Espaço: $S(N,M) = O(M)$

```
int[] cost = new int[M+1]; // iniciado c/ 0's
int[] best = new int[M+1]; // iniciado c/ 0's

for (int i = 1; i <= N; i++ )
    for (int k = size[i]; k <= M; k++)
        if (val[i] + cost[k-size[i]] > cost[k]) {
            cost[k] = val[i] + cost[k-size[i]];
            best[k] = i;
        }

// impressão de resultados (valor e itens)
print(cost[M]);
for (int k = M; k > 0; k -= size[best[k]])
    print(best[k]);
```

Como k é percorrido por ordem crescente $cost[k-size[i]]$ já tem o valor da iteração i

Técnicas de Concepção de Algoritmos, CAL - MIEIC/FEUP

18

Números de Fibonacci

- ◆ $F = \{0, 1, 1, 2, 3, 5, 8, 13, 21, 34, \dots\}$
- ◆ Fórmula de recorrência: $F(n) = F(n-1) + F(n-2)$, $n > 1$
 - $F(0) = 0$
 - $F(1) = 1$
- ◆ Para calcular $F(n)$, basta memorizar os dois últimos elementos da sequência para calcular o seguinte:

```
int Fib(int n) {
    int a = 1, b = 0; // F(1), F(0)
    for (int i=1; i <= n; i++) {int t = a; a = b; b += t;}
    return b;
}
```

Técnicas de Concepção de Algoritmos, CAL - MIEIC/FEUP

19

Subsequência crescente mais comprida

◆ Exemplo:

- Sequência $S = (9, 5, 2, 8, 7, 3, 1, 6, 4)$
- Subsequência crescente mais comprida (elem's não necessariamente contíguos): $(2, 3, 4)$ ou $(2, 3, 6)$

◆ Formulação:

- s_1, \dots, s_n - sequência
- l_i - compr. da maior subseq. crescente de (s_1, \dots, s_i)
- p_i - predecessor de s_i nessa subsequência crescente
- $l_i = 1 + \max \{ l_k \mid 0 < k < i \wedge s_k < s_i \}$ ($\max\{\} = 0$)
- p_i = valor de k escolhido para o máx. na expr. de l_i
- Comprimento final: $\max(l_i)$

Técnicas de Conceção de Algoritmos, CAL - MIEIC/FEUP

20

Cálculo para o exemplo dado

	i	1	2	3	4	5	6	7	8	9
Sequência	s_i	9	5	<u>2</u>	8	7	<u>3</u>	1	<u>6</u>	4
Tamanho	l_i	1	1	1	2	2	2	1	<u>3</u>	3
Predecessor	p_i	-	-	-	2	2	<u>3</u>	-	<u>6</u>	6

Resposta: (2, 3, 6)

Técnicas de Conceção de Algoritmos, CAL - MIEIC/FEUP

Referências

- ◆ Mark Allen Weiss. Data Structures & Algorithm Analysis in Java. Addison-Wesley, 1999
- ◆ Steven S. Skiena. The Algorithm Design Manual. Springer 1998
- ◆ Robert Sedgewick. Algorithms in C++. Addison-Wesley, 1992