1

Técnicas de Concepção de Algoritmos (1ª parte): programação dinâmica

R. Rossetti, A.P. Rocha
CAL, MIEIC, FEUP
Fevereiro de 2016

Técnicas de Concepção de Algoritmos, CAL - MIEIC/FEUP (2015-2016)

2

Programação dinâmica (*dynamic programming*)

Técnicas de Concepção de Algoritmos, CAL - MIEIC/FEUP (2015-2016)

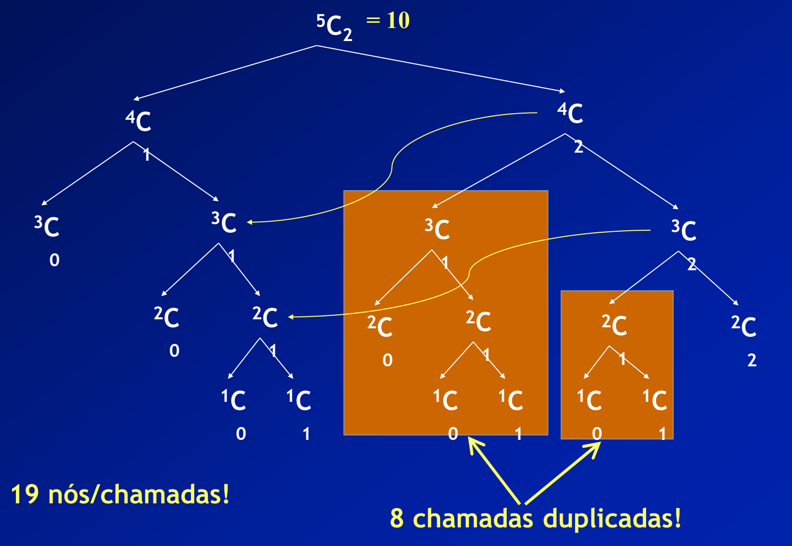
3

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.

4

${}^{n}C_{k}$ - repetição de trabalho



Exemplo: ${}^{n}C_{k}$, 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 ${}^{n}C_{k}$-1 vezes (nº de somas a efectuar é nº de parcelas -1)

Executa ${}^{n}C_{k}$ vezes (nº de 1's / parcelas que é preciso somar ...)

Executa $2{}^{n}C_{k}$-1 vezes para calcular ${}^{n}C_{k}$!!

Pode-se melhorar muito, evitando repetição de trabalho (cálculos intermédios ${}^{i}C_{j}$)

${}^{n}C_{k}$ - Programação dinâmica

Memorização de soluções parciais:

${}^{n}C_{k}$	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

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];
}
```

$T(n,k) = O(k(n-k))$
$S(n,k) = O(n-k)$

n-k+1 vezes

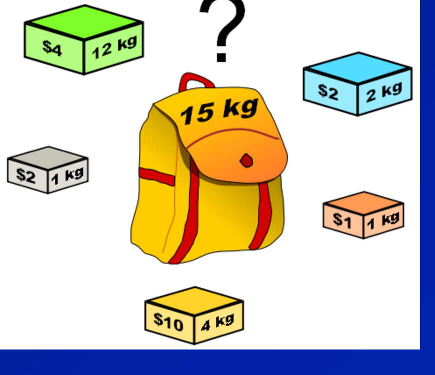
k(n-k) vezes

Técnicas de Concepção de Algoritmos, CAL - MIEIC/FEUP (2015-2016)

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 nº ilimitado de itens de cada tipo



Técnicas de Concepção de Algoritmos, CAL - MIEIC/FEUP (2015-2016)

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:

Tamanho	3	7	7
Valor	4	10	10
Nome	A	C	C

Outra solução óptima:

Tamanho	8	9
Valor	11	13
Nome	D	E

Técnicas de Concepção de Algoritmos, CAL - MIEIC/FEUP (2015-2016)

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 (2015-2016)

Dados

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

Formulação recursiva

- Caso base (i = 0; k = 1, ..., M):

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

- Caso recursivo (i = 1, ..., N; k = 1, ..., M) :

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

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

Encher o resto

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

13

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	1	2	2	1	3	4	5	3	3	5	3	3	4	5	3

14

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

* Números de Fibonacci

- Formulação recursiva: {0, 1, 1, 2, 3, 5, 8, 13, 21, 34, ...}
  - F(0) = 0
  - F(1) = 1
  - F(n) = F(n-1) + F(n-2), n > 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;
}
```


* 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:
  - $s_1, \ldots, s_n$ - sequência
  - l_i - compr. da maior subseq. crescente de $(s_1, \ldots, s_i)$
  - 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: $\max(l_i)$

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
- Slides de Maria Cristina Ribeiro