



Notas de Aula 11 - Algoritmos e Estrutura de Dados 2 (AE43CP) — Técnicas e Análise de Algoritmos: programação dinâmica Prof. Jefferson T. Oliva

Nas aulas anteriores, estudamos duas técnicas para projeto de algoritmos:

Força-bruta: também conhecida como tentativa-e-erro, esses algoritmos geram todas as soluções possíveis, incluindo as erradas, e no final, retornar a melhor solução (ótima).

Métodos gulosos: técnicas que consideram melhores objetos locais como parte da solução global. Como vocês viram, métodos gulosos são geralmente simples de serem implementados, mas nem sempre geram a solução ótima.

Divisão e conquista: dividir, conquistar e combinar. Os métodos também são geralmente simples de serem implementados, muitas vezes são rápidos e simplificam problemas complexos. No entanto, podem haver repetições de subproblemas e é necessária uma quantidade extra de memória.

Hoje veremos uma outra técnica para a solução de problemas: programação dinâmica.

Programação Dinâmica

Essa técnica divide o problema em vários subproblemas mais simples (similar à divisão e conquista). Esses subproblemas são resolvidos e as soluções são armazenadas em uma tabela para serem reusadas para a solução de subproblemas pouco maiores. Isso é feito até que o problema original seja resolvido. Como o problema é resolvido em partes, dos menores aos maiores subproblemas, a solução é feita iterativamente. Nessa abordagem, como a solução de subproblemas menores é utilizada em problemas maiores, o recálculo pode ser evitado, economizando operações computacionais.

A ideia básica da programação dinâmica é construir a solução por etapas.

Primeiro, o problema é decomposto em subproblemas de tamanho mínimo.

Durante a solução do problema, os resultados são combinados para a geração da resposta para subproblemas maiores.

Isso é feito até que o problema original seja resolvido.

Cada passo é feito uma única vez, mas as soluções podem ser reusadas quando for necessário.

Com o reuso de resultados, o número de verificações é reduzido drasticamente. Para isso, durante a aplicação da programação dinâmica, as soluções que não são ótimas são descartadas em cada passo.

Consequentemente, a solução final é ótima.

A programação dinâmica tem duas desvantagens: o desenvolvimento de uma solução pode ser mais complexa e pode ser necessário uso de grande espaço de memória (para armazenar resultados).







Existem algoritmos exponenciais que podem ter a sua complexidade reduzida por meio da programação dinâmica. Um exemplo é a sequência de Fibonacci, cuja solução recursiva tem complexidade O(2^n) e a iterativa tem o custo de tempo na ordem de O(n). Na programação dinâmica, pode ser utilizado um vetor de tamanho n + 1 para o armazenamento da sequência, em que os resultados subproblemas menores estão armazenados nas primeiras posições do vetor. Quanto maior for o índice do vetor, maior é o subproblema e o problema fica na posição n. Dessa forma o curso de espaço é de Theta(n).

```
Fib1(n)
1: if f[n] != ∞ then
2: return f [n]
3: end if
4: if n ≤ 1 then
5: return f [n] = n
6: end if
7: return f [n] = Fib1(n − 1) + Fib1(n − 2)
```

Assim, o vetor é inciado com valores infinitos, nas posições zero e um são armazenados 0 e 1 respectivamente. Nos demais campos do vetor, são armazenados os resultados da soma entre v[i - 2] e v[i - 1].

Em diversos problemas podem ser aplicada a programação dinâmica, dos quais abordaremos nessa aula: multiplicação de cadeias de matrizes e mochila binária

Multiplicação de Cadeias de Matrizes

Cálculo de multiplicação de cadeias de matrizes: minimizar as operações escalares

Considere o produto: $M = M_{1}[10,20] \times M_{2}[20,50] \times M_{3}[50,1] \times M_{4}[1,100]$

 $M = M_{1} \times (M_{2} \times (M_{3} \times M_{4}))$ requer 125.000 operações

 $M = (M_{1} \times (M_{2} \times M_{3})) \times M_{4}$ requer 2.200 operações

A figura abaixo é o slide 9 da aula "Programação Dinâmica"

```
Seja m<sub>i,j</sub> o menor custo para calcular o produto M<sub>i</sub> × M<sub>i+1</sub>... × M<sub>j</sub> para 1 ≤ i ≤ j ≤ n
m<sub>i,j</sub> = 0 se i = j
m<sub>i,j</sub> = Min<sub>i≤k≤j</sub>(m<sub>i,k</sub> + m<sub>k+1,j</sub> + b<sub>i-1</sub>b<sub>k</sub>b<sub>j</sub>) se j > i
onde:
m<sub>i,k</sub> representa o custo mínimo para calcular M' = M<sub>i</sub> × M<sub>i+1</sub> × ... × M<sub>k</sub>
m<sub>k+1,j</sub> representa o custo mínimo para calcular calcular M" = M<sub>k+1</sub> × M<sub>k+2</sub> × ... × M<sub>j</sub>
b<sub>i-1</sub>b<sub>k</sub>b<sub>j</sub> é o custo para multiplicar M'[b<sub>i-1</sub>, b<sub>k</sub>] por M"[b<sub>k</sub>, b<sub>j</sub>]
b<sub>i-1</sub> × b<sub>i</sub> são as dimensões da matriz M<sub>i</sub>
m<sub>i,j</sub>, j > i, representa o custo mínimo de todos os valores
```

possíveis de k entre i e j-1 da soma dos três termos







m[i,j] é um elemento de uma tabela onde serão armazenados os custos mínimos para multiplicação de cadeias de matrizes. Essa tabela é uma matriz quadrática onde a quantidade de linhas e colunas são diretamente proporcionais à quantidade de matrizes consideradas na multiplicação.

Na tabela (ou matriz) de custos, por exemplo, o elemento m[1,3] representa o custo mínimo para multiplicar M1 x M2 x M3.

Nessa matriz, os elementos da diagonal principal são iniciados com o valor 0, pois i = j. Quando j > i, deve ser considerada a minimização apresentada no slide 9 (figura na página anterior). Se j - i for igual a 1, significa que m[i,j] é a quantidade de operações para calcular Mi x Mj, ou seja, não é necessário incluir parênteses para minimizar a multiplicação das matrizes. Caso, j - i > 1, deve-se definir onde deverá haver a parentesiação (por isso, k entra na equação). Por exemplo, a m[1,3] é o menor custo entre (M1 x M2) x M3 e M1 x (M2 x M3).

A figura abaixo é o

• Considerando o exemplo anterior: $M = M_1[10, 20] \times M_2[20, 50] \times M_3[50, 1] \times M_4[1, 100]$. Como minimizar as operações escalares?

$m_{11} = ?$	$m_{12} = ?$	$m_{13} = ?$	$m_{14} = ?$
	$m_{22} = ?$	$m_{23} = ?$	$m_{24} = ?$
		$m_{33} = ?$	$m_{34} = ?$
			$m_{44} = ?$

$$b_0 = 10 \mid b_1 = 20 \mid b_2 = 50 \mid b_3 = 1 \mid b_4 = 100$$

- Relembrando:
 - $m_{i,j} = 0$ se i = j
 - $m_{i,j} = Min_{i < k < j} (m_{i,k} + m_{k+1,j} + b_{i-1}b_kb_j)$ se j > i

No exemplo apresentado em aula

$$m[1,2] = m[1,1] + m[2,2] + b[0] * b[1] * b[2] => M1 x M2$$

$$m[2,3] = m[2,2] + m[3,3] + b[1] * b[2] * b[3] => M2 x M3$$

$$m[3,4] = m[3,3] + m[4,4] + b[2] * b[3] * b[4] => M3 x M4$$

$$m[1,3] = m[1,1] + m[2,3] + b[0] * b[1] * b[3] => 0 + M2 x M3 + (M1 x (M2 x M3)) => A$$

$$m[1,2] + m[3,3] + b[0] * b[2] * b[3] => M1 x M2 + 0 + ((M1 x M2) x M3) => B$$

$$M1 x M2 x M3$$

$$m[2,4] = m[2,2] + m[3,4] + b[1] * b[2] * b[4] => 0 + M3 x M4 + (M2 x (M3 x M4)) => C$$

$$m[2,3] + m[4,4] + b[1] * b[3] * b[4] => M2 x M3 + 0 + ((M2 x M3) x M4) => D$$

$$M2 x M3 x M4$$







```
m[1,4] = m[1,1] + m[2,4] + b[0] * b[1] * b[4] => 0 + (C \text{ ou } D) + (M1 \text{ x } (C \text{ ou } D))
m[1,2] + m[3,4] + b[0] * b[2] * b[4] => M1 \text{ x } M2 + M3 \text{ x } M4 + ((M1 \text{ x } M2) \text{ x } (M3 \text{ x } M4))
m[1,3] + m[4,4] + b[0] * b[3] * b[4] => (A \text{ ou } B) + 0 + ((A \text{ ou } B) \text{ x } M4)
```

A resolução do exemplo acima é apresentada entre os slides 11 e 15.

Implementação da solução em C

```
#include <stdio.h>
#include <stdlib.h>
#define inf 2147483647
int** create_matrix(int l, int c){
  int i = 0;
  int** mat = (int**) malloc(sizeof(int*) * l);
  for (i = 0; i < l; i++)
     mat[i] = (int*) malloc(sizeof(int) * c);
  return mat;
}
int multiplicacao_minima(int** m, int n, int b[]){
  int h, i, j, k, aux;
  for (i = 0; i < n; i++)
     m[i][i] = 0;
  for (h = 1; h < n; h++) { // indica qual diagonal após a principal será processada. Por exemplo: se h = 1,
a diagonal processada é logo após a principal
     for (i = 1; i \le n - h; i++) // n - h é a quantidade de elementos analisados na diagonal
       j = i + h; // i = > linhas; j = > colunas
       m[i - 1][j - 1] = INT\_MAX;
       for (k = i; k \le j - 1; k++)
          aux = m[i-1][k-1] + m[k][j-1] + b[i-1] * b[k] * b[j];
          if (aux < m[i - 1][j - 1])
            m[i - 1][j - 1] = aux;
    }
  }
  return m[0][n - 1];
```







```
int main(){
   int b[] = {10, 20, 50, 1, 100};
   int** m = create_matrix(4, 4);
   int min = multiplicacao_minima(m, 4, b);
   printf("Operacoes minimas: %d", min);
   free(m);
   return 0;
}
```

Como determinar a parentesiação?

Solução: utilizar uma outra tabela auxiliar para determinar onde vai a parentesiação em m[i,j]

$m_{11} = 0$	$m_{12} = 10.000$	$m_{13} = 1.200$	$m_{14} = 2.200$
	$m_{22} = 0$	$m_{23} = 1.000$	$m_{24} = 3.000$
		$m_{33} = 0$	$m_{34} = 5.000$
			$m_{44} = 0$

$$b_0 = 10 \mid b_1 = 20 \mid b_2 = 50 \mid b_3 = 1 \mid b_4 = 100$$

0	1	1 (M1 x (M2 x M3))	3 ((M1 x (M2 x M3)) x M4)
	0	2	3 ((M2 x M3) x M4)
		0	3
			0

Na matriz auxiliar deve ser colocado o valor de k da escolha

Primeiro, olhar na posição 1,4 da matriz, no qual indica que a separação deve ser na matriz 3, ou seja, de M1 x M2 x M3 x M4, agora será (M1 x M2 x M3) x (M4). O próximo passo será olhar o campo da matriz na posição 1,3 (já que ainda falta colocar parênteses entre as matrizes de 1 a 3).

Na posição 1,3, é indicado que a separação deve ser na matriz 1, ou seja, de $M1 \times M2 \times M3$, passa para (M1) x ($M2 \times M3$). Assim, no problema original passa a ser de ($M1 \times M2 \times M3$) x (M4) para ((M1) x ($M2 \times M3$)) x (M4).

Complexidade de tempo: Theta(n^3)

Complexidade de espaço (espaço extra): Theta(n^2)







Problema da Mochila Binária

Considere **n** itens a serem levados para uma viagem, dentro de uma mochila de capacidade **b**

Cada item x[i] tem um peso a[i] e um valor c[i]. Quais itens escolher, que modo que o valor total dos itens levados seja o maior possível?

As figuras a seguir são os slides 18--21.

 O problema da mochila é definido em programação matemática:

$$P: z = \max \sum_{i=1}^{n} c_i x_i$$

sujeito a

$$\sum_{i=1}^{n} a_{i} x_{j} \leq b, \ x_{j} \in \{0, 1\}$$

Princípio para aplicação de programação dinâmica:

imagine que o lado direito da desigualdade assume um valor λ que varia de 0 até b e as soluções ótimas são $x_0, ..., x_k, ..., x_b$

Isto nos leva a definir o problema $P_k(\lambda)$, a sua respectiva solução ótima $x_k(\lambda)$ e o seu respectivo valor objetivo $f_k(\lambda)$

$$Pk(\lambda): f_k(\lambda) = \max \sum_{i=1}^k c_i x_i$$

sujeito a

$$\sum_{j=1}^{k} a_j x_j \le \lambda, \ x_j \in \{0,1\}, \ \{a\}_{j=1}^n, \ j=1,\ldots,k$$

 $P_k(\lambda)$ é o problema da mochila restrito aos k primeiros itens e uma mochila de capacidade λ

Resta-nos obter um procedimento que permita calcular $f_k(\lambda)$ em termos dos valores $f_s(u)$ com $s \leq k$ e $u \leq \lambda$

O que podemos dizer sobre a solução ótima x* para o problema $P_k(\lambda)$ com valor $f_k(\lambda)$?

•
$$x_k^* = 0$$
 ou $x_k^* = 1$







Considerando cada caso, temos:

- Se $x_k^* = 0$, então a solução ótima satisfaz $f_k(\lambda) = f_{k-1}(\lambda)$
- Se $x_k^* = 1$, então a solução ótima satisfaz $f_k(\lambda) = c_k + f_{k-1}(\lambda - a_k)$

Combinando os casos (1) e (2), obtêm-se a recorrência abaixo:

$$f_k(\lambda) = \max\{f_{k-1}(\lambda), c_k + f_{k-1}(\lambda - a_k)\}\$$
 (1)

Definindo-se os valores iniciais como $f_0(\lambda)=0$ para $\lambda\geq 0$, pode-se utilizar a recorrência 1 para calcular sucessivamente os valores de $f_1,f_2,...,f_n$ para todos os valores inteiros de $\lambda\in\{0,\ldots,b\}$

Quando x_{k}^{*} é igual a zero, significa que o item k não é considerado.

O que a Equação 1 quer dizer? Se o item x_k não for selecionado, então o valor de $f_k(lambda)$ será o mesmo de $f_k(lambda)$. Se o item x_k for selecionado, então o valor de $f_k(lambda)$ será o custo do item k mais o valor de $f_k(lambda - peso do item <math>k$), ou seja, o custo do item k mais mais o maior custo para a inclusão de outros itens antes de k

A questão que resta é como encontrar a solução ótima associada ao valor ótimo.

Exemplo: considere a instância do problema da mochila

$$-z = max 10x_1 + 7x_2 + 25x_3 + 24x_4$$
, sujeito a

$$-2x_1 + x_2 + 6x_3 + 5x_4 \le 7$$

O que isso quer dizer? Que temos 4 itens (x1, x2, x3 e x4), cada um com determinado custo (10, 7, 25 e 24) e peso (2, 1, 6, e 5). Desejamos colocar esses itens em uma mochila com capacidade 7,

λ	f0	f1	f2	f3	f4
0	0	0	0	0	0
1	0	0	7	7	7
2	0	10	10	10	10
3	0	10	17	17	17
4	0	10	17	17	17
5	0	10	17	17	24
6	0	10	17	25	31
7	0	10	17	32	34

$$a1 = 2$$
, $a2 = 1$, $a3 = 6$, $a4 = 5$, $c1 = 10$, $c2 = 7$, $c3 = 25$, $c4 = 24$, $b = 7$







$$f_k(\lambda) = \max\{f_{k-1}(\lambda), c_k + f_{k-1}(\lambda - a_k)\}\$$

```
f1(0) = max(f0(0); c1 + f0(0 - a1)) = max(0; 10 + f0(0 - 2)) = 0 (não existe f0(-2)) – Extrapola peso
f1(1) = max(f0(1), c1 + f0(1 - a1)) = max(0; 10 + f0(1 - 2)) = 0 (não existe f0(-1)) – Extrapola peso
f1(2) = max(f0(2), c1 + f0(2 - a1)) = max(0; 10 + f0(2 - 2)) = 10 // Neste caso, c1 faz parte da solução
f1(3) = max(f0(3), c1 + f0(3 - a1)) = max(0; 10 + f0(3 - 2)) = 10 // Neste caso, c1 faz parte da solução
f1(4) = max(f0(4), c1 + f0(4 - a1)) = max(0; 10 + f0(4 - 2)) = 10 // Neste caso, c1 faz parte da solução
f1(5) = max(f0(5), c1 + f0(5 - a1)) = max(0; 10 + f0(5 - 2)) = 10 // Neste caso, c1 faz parte da solução
f1(6) = max(f0(6), c1 + f0(6 - a1)) = max(0; 10 + f0(6 - 2)) = 10 // Neste caso, c1 faz parte da solução
f1(7) = max(f0(7), c1 + f0(7 - a1)) = max(0; 10 + f0(7 - 2)) = 10 // Neste caso, c1 faz parte da solução
f2(0) = \max\{f1(0); c2 + f1(0 - a2)\} = \max\{0; 7 + f1(0 - 1)\} = 0 (não existe f0(-1)) – Extrapola peso
f2(1) = max\{f1(1); c2 + f1(1 - a2)\} = max\{0; 7 + f1(1 - 1)\} = 7 // Neste caso, c2 faz parte da solução
f2(2) = max\{f1(2); c2 + f1(2 - a2)\} = max\{10; 7 + f1(2 - 1)\} = 10 \text{ // Neste caso, c1 faz parte da solução}
f2(3) = max\{f1(3); c2 + f1(3 - a2)\} = max\{10; 7 + f1(3 - 1)\} = 17 // Neste caso, c1 e c2 fazem parte da solução
f2(4) = max\{f1(4); c2 + f1(4 - a2)\} = max\{10; 7 + f1(4 - 1)\} = 17 // Neste caso, c1 e c2 fazem parte da solução
f2(5) = max\{f1(5); c2 + f1(5 - a2)\} = max\{10; 7 + f1(5 - 1)\} = 17 // Neste caso, c1 e c2 fazem parte da solução
f2(6) = max\{f1(6); c2 + f1(6 - a2)\} = max\{10; 7 + f1(6 - 1)\} = 17 // Neste caso, c1 e c2 fazem parte da solução
f2(6) = max\{f1(7); c2 + f1(7 - a2)\} = max\{10; 7 + f1(7 - 1)\} = 17 // Neste caso, c1 e c2 fazem parte da solução
f3(0) = max\{f2(0); c3 + f2(0 - a3)\} = max\{0; 25 + f2(0 - 6)\} = 0 (não existe f0(-6))
f3(1) = max\{f2(1); c3 + f2(1 - a3)\} = max\{7; 25 + f2(1 - 6)\} = 7 \text{ (não existe } f0(-5)\text{)}
f3(2) = max\{f2(2); c3 + f2(2 - a3)\} = max\{10; 25 + f2(2 - 6)\} = 10 (não existe f0(-4))
f3(3) = max\{f2(3); c3 + f2(3 - a3)\} = max\{17; 25 + f2(3 - 6)\} = 17 \text{ (não existe f0(-3))}
f3(4) = max\{f2(4); c3 + f2(4 - a3)\} = max\{17; 25 + f2(4 - 6)\} = 17  (não existe f0(-2))
f3(5) = max\{f2(5); c3 + f2(5 - a3)\} = max\{17; 25 + f2(5 - 6)\} = 17 \text{ (não existe f0(-1))}
f3(6) = max\{f2(6); c3 + f2(6 - a3)\} = max\{17; 25 + f2(6 - 6)\} = 25 // Neste caso, c1 e c3 fazem parte da solução
f3(7) = max\{f2(7); c3 + f2(7 - a3)\} = max\{17; 25 + f2(7 - 6)\} = 32 // Neste caso, c2 e c3 fazem parte da solução
f(4(0)) = \max\{f(3(0)); c(4) + f(3(0 - a(4))\} = \max\{0; 24 + f(2(0 - 5))\} = 0 \text{ (não existe } f(0(-5))\}
f4(1) = max\{f3(1); c4 + f3(1 - a4)\} = max\{7; 24 + f2(1 - 5)\} = 7 \text{ (não existe } f0(-4))
f4(2) = max\{f3(2); c4 + f3(2 - a4)\} = max\{10; 24 + f2(2 - 5)\} = 10 (não existe f0(-3))
f4(3) = max\{f3(3); c4 + f3(3 - a4)\} = max\{17; 24 + f2(3 - 5)\} = 17  (não existe f0(-2))
f4(4) = max\{f3(4); c4 + f3(4 - a4)\} = max\{17; 25 + f3(4 - 5)\} = 17  (não existe f0(-1))
f4(5) = max\{f3(5); c4 + f3(5 - a4)\} = max\{17; 24 + f3(5 - 5)\} = 24 // Neste caso, c3 e c4 fazem parte da solução
f4(6) = max\{f3(6); c4 + f3(6 - a4)\} = max\{25; 24 + f3(6 - 5)\} = 31 // Neste caso, c1, c2 e c3 fazem parte da solução
f4(7) = max\{f3(7); c4 + f3(7-a4)\} = max\{32; 24 + f3(7-5)\} = 34 // Neste caso, c1 e c4 fazem parte da solução
```

Complexidade de tempo: Theta(b * n) Complexidade de espaço: Theta(b * n)







Programação Dinâmica

Vantagens:

- Explora todas as alternativa de maneira eficiente
- A cada iteração, a decisão pode ser reconsiderada
- Tem prova de correção simples

Desvantagens:

- Solução pode ser lenta, pois a complexidade pode ser exponencial
- Necessita de maior espaço de memória

Outros exemplos de aplicação:

- Algoritmo de Dijkstra
- Subsequência crescente máxima
- Sequência de Fibonacci
- Entre outras

Referências

Cormen, T. H.; Leiserson, C. E.; Rivest, R. L.; Clifford, S. Algoritmos: teoria e prática. Elsevier, 2012.

Horowitz, E., Sahni, S. Rajasekaran, S. Computer Algorithms. Computer Science Press, 1998.

Szwarcfiter, J.; Markenzon, L. Estruturas de Dados e Seus Algoritmos. LTC, 2010.

Ziviani, M. Projetos de Algoritmos: com implementações em Pascal e C. Thomson, 2004.

