

MC448 — Análise de Algoritmos I

Cid Carvalho de Souza Cândia Nunes da Silva
Orlando Lee

22 de outubro de 2009

Programação Dinâmica

Programação Dinâmica: Conceitos Básicos

- Tipicamente o paradigma de programação dinâmica aplica-se a problemas de **otimização**.
- Podemos utilizar programação dinâmica em problemas onde há:
 - **Subestrutura Ótima**: As soluções ótimas do problema incluem soluções ótimas de subproblemas.
 - **Sobreposição de Subproblemas**: O cálculo da solução através de recursão implica no recálculo de subproblemas.

Programação Dinâmica: Conceitos Básicos (Cont.)

- A técnica de **programação dinâmica** visa evitar o recálculo desnecessário das soluções dos subproblemas.
- Para isso, soluções de subproblemas são armazenadas em **tabelas**.
- Logo, para que o algoritmo de programação dinâmica seja eficiente, é preciso que o número total de subproblemas que devem ser resolvidos seja pequeno (polinomial no tamanho da entrada).

Multiplicação de Cadeias de Matrizes

Problema: Multiplicação de Matrizes

Calcular o número mínimo de operações de multiplicação (escalar) necessários para computar a matriz M dada por:

$$M = M_1 \times M_2 \times \dots \times M_i \dots \times M_n$$

onde M_i é uma matriz de b_{i-1} linhas e b_i colunas, para todo $i \in \{1, \dots, n\}$.

- Matrizes são multiplicadas aos pares sempre. Então, é preciso encontrar uma parentização (agrupamento) ótimo para a cadeia de matrizes.
- Para calcular a matriz M' dada por $M_i \times M_{i+1}$ são necessárias $b_{i-1} * b_i * b_{i+1}$ multiplicações entre os elementos de M_i e M_{i+1} .

Multiplicação de Cadeias de Matrizes (Cont.)

- **Exemplo:** Qual é o mínimo de multiplicações escalares necessárias para computar $M = M_1 \times M_2 \times M_3 \times M_4$ com $b = \{200, 2, 30, 20, 5\}$?
- As possibilidades de parentização são:

$$\begin{aligned} M &= (M_1 \times (M_2 \times (M_3 \times M_4))) \rightarrow 5.300 \text{ multiplicações} \\ M &= (M_1 \times ((M_2 \times M_3) \times M_4)) \rightarrow 3.400 \text{ multiplicações} \\ M &= ((M_1 \times M_2) \times (M_3 \times M_4)) \rightarrow 45.000 \text{ multiplicações} \\ M &= ((M_1 \times (M_2 \times M_3)) \times M_4) \rightarrow 29.200 \text{ multiplicações} \\ M &= (((M_1 \times M_2) \times M_3) \times M_4) \rightarrow 152.000 \text{ multiplicações} \end{aligned}$$

- A ordem das multiplicações faz **muita** diferença !

Multiplicação de Cadeias de Matrizes (Cont.)

- Poderíamos calcular o número de multiplicações para todas as possíveis parentizações.
- O número de possíveis parentizações é dado pela recorrência:

$$P(n) = \begin{cases} 1, & n = 1 \\ \sum_{k=1}^{n-1} P(k) \cdot P(n-k) & n > 1, \end{cases}$$

- Mas $P(n) \in \Omega(4^n/n^2)$, a estratégia de força bruta é **impraticável** !

Multiplicação de Cadeias de Matrizes (Cont.)

- Inicialmente, para todo (i, j) tal que $1 \leq i \leq j \leq n$, vamos definir as seguintes matrizes:

$$M_{i,j} = M_i \times M_{i+1} \times \dots \times M_j.$$

- Agora, dada uma parentização ótima, existem dois pares de parênteses que identificam o último par de matrizes que serão multiplicadas.
Ou seja, existe k tal que $M = M_{1,k} \times M_{k+1,n}$.
- Como a parentização de M é ótima, as parentizações no cálculo de $M_{1,k}$ e $M_{k+1,n}$ devem ser ótimas também, caso contrário, seria possível obter uma parentização de M ainda melhor!
- Eis a **subestrutura ótima** do problema: a parentização ótima de M inclui a parentização ótima de $M_{1,k}$ e $M_{k+1,n}$.

Multiplicação de Cadeias de Matrizes (Cont.)

- De forma geral, se $m[i, j]$ é número mínimo de multiplicações que deve ser efetuado para computar $M_i \times M_{i+1} \times \dots \times M_j$, então $m[i, j]$ é dado por:

$$m[i, j] := \min_{i \leq k < j} \{m[i, k] + m[k + 1, j] + b_{i-1} * b_k * b_j\}.$$

- Podemos então projetar um algoritmo recursivo (**indutivo**) para resolver o problema.

Efetuando a Multiplicação Ótima

- É muito fácil efetuar a multiplicação da cadeia de matrizes com o número mínimo de multiplicações escalares usando a tabela s , que registra os índices ótimos de divisão em subcadeias.

MultiplicaMatrizes(M, s, i, j)

▷ **Entrada:** Cadeia de matrizes M , a tabela s e os índices i e j que delimitam a subcadeia a ser multiplicada.

▷ **Saída:** A matriz resultante da multiplicação da subcadeia entre i e j , efetuando o mínimo de multiplicações escalares.

- se** $i < j$ **então**
- $X := \text{MultiplicaMatrizes}(M, s, i, s[i, j])$
- $Y := \text{MultiplicaMatrizes}(M, s, s[i, j] + 1, j)$
- devolva** $\text{Multiplica}(X, Y, b[i - 1], b[s[i, j]], b[j])$
- senão devolva** M_i ;

Multiplicação de Matrizes - Algoritmo Recursivo

MinimoMultiplicacoesRecursivo(b, i, j)

▷ **Entrada:** Vetor b com as dimensões das matrizes e os índices i e j que delimitam o início e término da subcadeia.

▷ **Saída:** O número mínimo de multiplicações escalares necessário para computar a multiplicação da subcadeia. Esse valor é registrado em uma tabela $(m[i, j])$, bem como o índice da divisão em subcadeias ótimas $(s[i, j])$.

- se** $i = j$ **então devolva** 0
- $m[i, j] := \infty$
- para** $k := i$ **até** $j - 1$ **faça**
- $q := \text{MinimoMultiplicacoesRecursivo}(b, i, k) +$
 $\text{MinimoMultiplicacoesRecursivo}(b, k + 1, j) +$
 $b[i - 1] * b[k] * b[j]$
- se** $m[i, j] > q$ **então**
- $m[i, j] := q$; $s[i, j] := k$
- devolva** $m[i, j]$.

Algoritmo Recursivo - Complexidade

- O número mínimo de operações feita pelo algoritmo recursivo é dada pela recorrência:

$$T(n) \geq \begin{cases} 1, & n = 1 \\ 1 + \sum_{k=1}^{n-1} [T(k) + T(n - k) + 1] & n > 1, \end{cases}$$

- Portanto, $T(n) \geq 2 \sum_{k=1}^{n-1} T(k) + n$, para $n > 1$.
- É possível provar (por substituição) que $T(n) \geq 2^{n-1}$, ou seja, o algoritmo recursivo tem complexidade $\Omega(2^n)$, ainda **impraticável!**

Algoritmo Recursivo - Complexidade

- A ineficiência do algoritmo recursivo deve-se à **sobreposição de subproblemas**: o cálculo do mesmo $m[i, j]$ pode ser requerido em vários subproblemas.
- Por exemplo, para $n = 4$, $m[1, 2]$, $m[2, 3]$ e $m[3, 4]$ são computados duas vezes.
- O número de total de $m[i, j]$'s calculados é apenas $O(n^2)$!
- Portanto, podemos obter um algoritmo mais eficiente se evitarmos recálculos de subproblemas.

Memorização x Programação Dinâmica

- Existem duas técnicas para evitar o recálculo de subproblemas:
 - **Memorização**: Consiste em manter a estrutura recursiva do algoritmo, registrando em uma tabela o valor ótimo para subproblemas já computados e verificando, antes de cada chamada recursiva, se o subproblema a ser resolvido já foi computado.
 - **Programação Dinâmica**: Consiste em preencher uma tabela que registra o valor ótimo para cada subproblema de forma apropriada, isto é, a computação do valor ótimo de cada subproblema depende somente de subproblemas já previamente computados. Elimina completamente a recursão.

Algoritmo de Memorização

MinimoMultiplicacoesMemorizado(b, n)

1. **para** $i := 1$ **até** n **faça**
2. **para** $j := 1$ **até** n **faça**
3. $m[i, j] := \infty$
4. **devolva** $Memorizacao(b, 1, n)$

Memorizacao(b, i, j)

1. **se** $m[i, j] < \infty$ **então devolva** $m[i, j]$
2. **se** $i = j$ **então** $m[i, j] := 0$
3. **senão**
4. **para** $k := i$ **até** $j - 1$ **faça**
5. $q := Memorizacao(b, i, k) +$
 $Memorizacao(b, k + 1, j) + b[i - 1] * b[k] * b[j];$
6. **se** $m[i, j] > q$ **então** $m[i, j] := q$; $s[i, j] := k$
7. **devolva** $m[i, j]$

Algoritmo de Programação Dinâmica

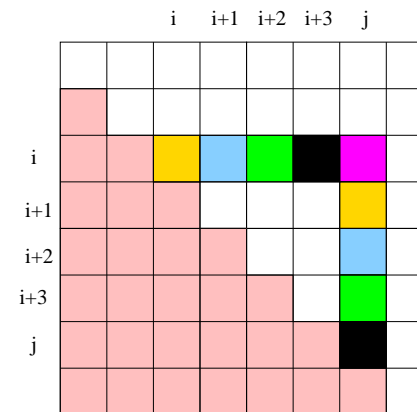
- O uso de programação dinâmica é preferível pois elimina completamente o uso de recursão.
- O algoritmo de programação dinâmica para o problema da multiplicação de matrizes é o seguinte: inicialmente a diagonal principal da tabela é preenchida com zeros.
- A partir daí as entradas das diagonais “à direita” são calculadas usando a recorrência. Note que as entradas da diagonal começando em $m[1, j]$ correspondem aos subproblemas com cadeias de j matrizes.

Algoritmo de Programação Dinâmica

MinimoMultiplicacoes(b)

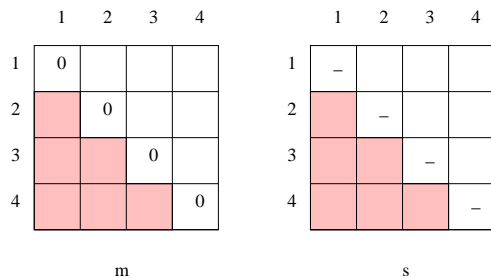
- ▷ **Entrada:** Vetor b com as dimensões das matrizes.
 ▷ **Saída:** As tabelas m e s preenchidas.
1. **para** $i = 1$ **até** n **faça** $m[i, i] := 0$
 - ▷ calcula o mínimo de todas sub-cadeias de tamanho $u + 1$
 2. **para** $u = 1$ **até** $n - 1$ **faça**
 3. **para** $i = 1$ **até** $n - u$ **faça**
 4. $j := i + u; m[i, j] := \infty$
 5. **para** $k = i$ **até** $j - 1$ **faça**
 6. $q := m[i, k] + m[k + 1, j] + b[i - 1] * b[k] * b[j]$
 7. **se** $q < m[i, j]$ **então**
 8. $m[i, j] := q; s[i, j] := k$
 9. **devolva** (m, s)

Algoritmo de Programação Dinâmica - Exemplo

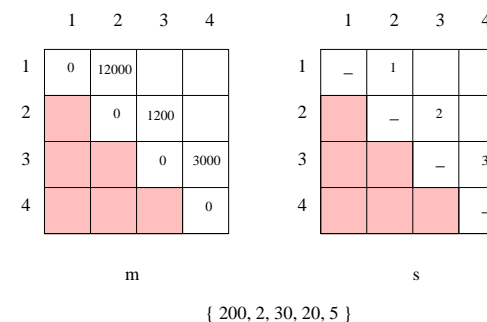


Note que para calcular $m[i, j]$ é preciso conhecer apenas as entradas da tabela m que estão nas diagonais anteriores.

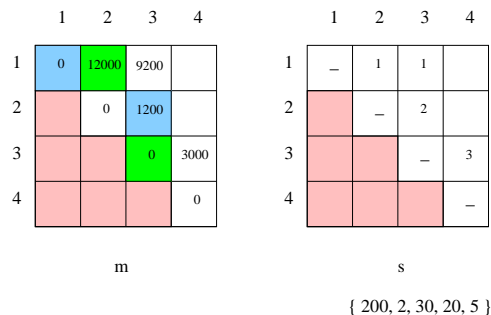
Algoritmo de Programação Dinâmica - Exemplo



Algoritmo de Programação Dinâmica - Exemplo



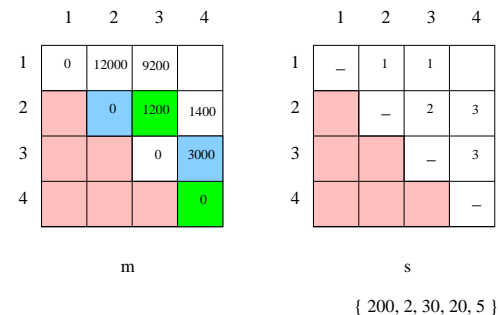
Algoritmo de Programação Dinâmica - Exemplo



$$b_0 * b_1 * b_3 = 200 * 2 * 20 = 8000$$

$$b_0 * b_2 * b_3 = 200 * 30 * 20 = 120000$$

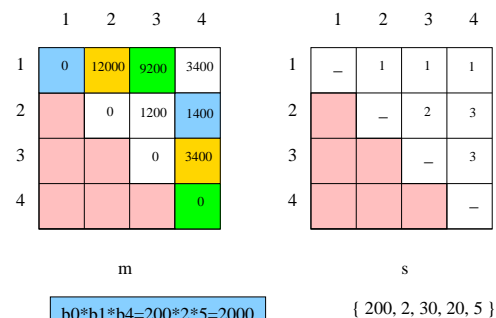
Algoritmo de Programação Dinâmica - Exemplo



$$b_1 * b_2 * b_4 = 2 * 30 * 5 = 300$$

$$b_1 * b_3 * b_4 = 2 * 20 * 5 = 200$$

Algoritmo de Programação Dinâmica - Exemplo

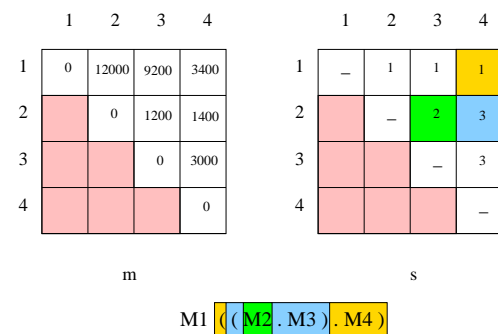


$$b_0 * b_1 * b_4 = 200 * 2 * 5 = 2000$$

$$b_0 * b_2 * b_4 = 200 * 30 * 5 = 30000$$

$$b_0 * b_3 * b_4 = 200 * 20 * 5 = 20000$$

Algoritmo de Programação Dinâmica - Exemplo



Algoritmo de Programação Dinâmica - Complexidade

- A complexidade de tempo do algoritmo é dada por:

$$\begin{aligned} T(n) &= \sum_{u=1}^{n-1} \sum_{i=1}^{n-u} \sum_{k=i}^{n-u+i-1} \Theta(1) \\ &= \sum_{u=1}^{n-1} \sum_{i=1}^{n-u} u \Theta(1) \\ &= \sum_{u=1}^{n-1} u(n-u) \Theta(1) \\ &= \sum_{u=1}^{n-1} (nu - u^2) \Theta(1). \end{aligned}$$

O Problema Binário da Mochila

O Problema da Mochila

Dada uma mochila de capacidade W (inteiro) e um conjunto de n itens com tamanho w_i (inteiro) e valor c_i associado a cada item i , queremos determinar quais itens devem ser colocados na mochila de modo a **maximizar** o valor total transportado, respeitando sua capacidade.

- Podemos fazer as seguintes suposições:
 - $\sum_{i=1}^n w_i > W$;
 - $0 < w_i \leq W$, para todo $i = 1, \dots, n$.

Algoritmo de Programação Dinâmica - Complexidade

- Como

$$\sum_{u=1}^{n-1} nu = n^3/2 - n^2/2$$

e

$$\sum_{u=1}^{n-1} u^2 = n^3/3 - n^2/2 + n/6.$$

Então,

$$T(n) = (n^3/6 - n/6) \Theta(1).$$

- A complexidade de tempo do algoritmo é $\Theta(n^3)$.
- A complexidade de espaço é $\Theta(n^2)$, já que é necessário armazenar a matriz com os valores ótimos dos subproblemas.

O Problema Binário da Mochila

- Podemos formular o problema da mochila com **Programação Linear Inteira**:

- Criamos uma variável x_i para cada item: $x_i = 1$ se o item i estiver na solução ótima e $x_i = 0$ caso contrário.
- A modelagem do problema é simples:

$$\max \sum_{i=1}^n c_i x_i \quad (1)$$

$$\sum_{i=1}^n w_i x_i \leq W \quad (2)$$

$$x_i \in \{0, 1\} \quad (3)$$

- (1) é a **função objetivo** e (2-3) o **conjunto de restrições**.

O Problema Binário da Mochila

- Como podemos projetar um algoritmo para resolver o problema?
- Existem 2^n possíveis subconjuntos de itens: um algoritmo de força bruta é **impraticável!**
- É um problema de otimização. **Será que tem subestrutura ótima?**
- Se o item n estiver na solução ótima, o valor desta solução será c_n mais o valor da melhor solução do problema da mochila com capacidade $W - w_n$ considerando-se só os $n - 1$ primeiros itens.
- Se o item n não estiver na solução ótima, o valor ótimo será dado pelo valor da melhor solução do problema da mochila com capacidade W considerando-se só os $n - 1$ primeiros itens.

O Problema Binário da Mochila

- Seja $z[k, d]$ o valor ótimo para o problema da mochila considerando os k primeiros itens (de tamanho w_1, \dots, w_k) e uma mochila de capacidade d .
- A fórmula de recorrência para computar $z[k, d]$ para todo valor de d e k é:

$$z[0, d] = 0$$

$$z[k, 0] = 0$$

$$z[k, d] = \begin{cases} z[k-1, d], & \text{se } w_k > d \\ \max\{z[k-1, d], z[k-1, d-w_k] + c_k\}, & \text{se } w_k \leq d \end{cases}$$

O Problema Binário da Mochila - Complexidade Recursão

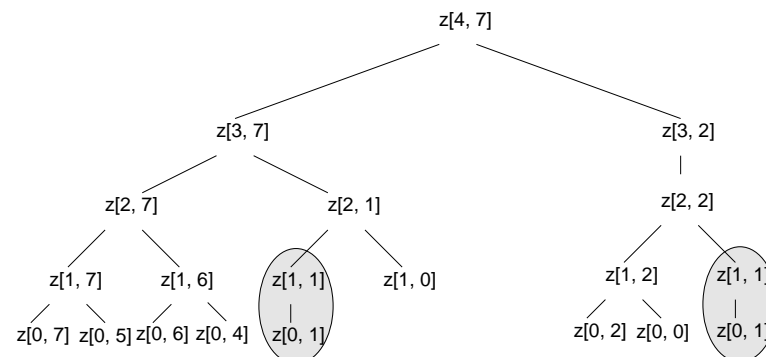
- A complexidade do algoritmo recursivo para este problema no **pior caso** é dada pela recorrência:

$$T(k, d) = \begin{cases} 1, & k = 0 \text{ ou } d = 0 \\ T(k-1, d) + T(k-1, d-w_k) + 1 & k > 0 \text{ e } d > 0. \end{cases}$$

- Portanto, no **pior caso**, o algoritmo recursivo tem complexidade $\Omega(2^n)$. É impraticável!
- Mas há **sobreposição de subproblemas**: o recálculo de subproblemas pode ser evitado!

Mochila - Sobreposição de Subproblemas

- Considere vetor de tamanhos $w = \{2, 1, 6, 5\}$ e capacidade da mochila $W = 7$. A árvore de recursão seria:

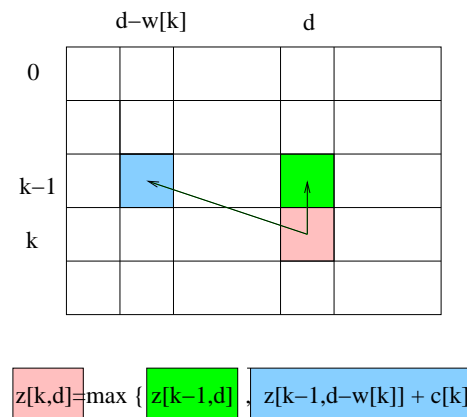


- O subproblema $z[1, 1]$ é computado duas vezes.

Mochila - Programação Dinâmica

- O número total máximo de subproblemas a serem computados é nW .
- Isso porque tanto o tamanho dos itens quanto a capacidade da mochila são **inteiros!**
- Podemos então usar programação dinâmica para evitar o recálculo de subproblemas.
- Como o cálculo de $z[k, d]$ depende de $z[k-1, d]$ e $z[k-1, d-w_k]$, preenchemos a tabela linha a linha, da esquerda para a direita.

Mochila



O Problema Binário da Mochila - Algoritmo

Mochila(c, w, W, n)

▷ **Entrada:** Vetores c e w com valor e tamanho de cada item, capacidade W da mochila e número de itens n .

▷ **Saída:** O valor máximo do total de itens colocados na mochila.

1. **para** $d := 0$ **até** W **faça** $z[0, d] := 0$
2. **para** $k := 1$ **até** n **faça** $z[k, 0] := 0$
3. **para** $k := 1$ **até** n **faça**
4. **para** $d := 1$ **até** W **faça**
5. $z[k, d] := z[k-1, d]$
6. **se** $w_k \leq d$ **e** $c_k + z[k-1, d-w_k] > z[k, d]$ **então**
7. $z[k, d] := c_k + z[k-1, d-w_k]$
8. **devolva** ($z[n, W]$)

Mochila - Exemplo

- Exemplo: $c = \{10, 7, 25, 24\}$, $w = \{2, 1, 6, 5\}$ e $W = 7$.

	d	0	1	2	3	4	5	6	7
k	0	0	0	0	0	0	0	0	0
1	0								
2	0								
3	0								
4	0								

Mochila - Exemplo

- Exemplo: $c = \{10, 7, 25, 24\}$, $w = \{2, 1, 6, 5\}$ e $W = 7$.

k \ d	0	1	2	3	4	5	6	7
0	0	0	0	0	0	0	0	0
1	0	0	10	10	10	10	10	10
2	0							
3	0							
4	0							

Mochila - Exemplo

- Exemplo: $c = \{10, 7, 25, 24\}$, $w = \{2, 1, 6, 5\}$ e $W = 7$.

k \ d	0	1	2	3	4	5	6	7
0	0	0	0	0	0	0	0	0
1	0	0	10	10	10	10	10	10
2	0	7	10	17	17	17	17	17
3	0							
4	0							

Mochila - Exemplo

- Exemplo: $c = \{10, 7, 25, 24\}$, $w = \{2, 1, 6, 5\}$ e $W = 7$.

k \ d	0	1	2	3	4	5	6	7
0	0	0	0	0	0	0	0	0
1	0	0	10	10	10	10	10	10
2	0	7	10	17	17	17	17	17
3	0	7	10	17	17	17	25	32
4	0							

Mochila - Exemplo

- Exemplo: $c = \{10, 7, 25, 24\}$, $w = \{2, 1, 6, 5\}$ e $W = 7$.

k \ d	0	1	2	3	4	5	6	7
0	0	0	0	0	0	0	0	0
1	0	0	10	10	10	10	10	10
2	0	7	10	17	17	17	17	17
3	0	7	10	17	17	17	25	32
4	0	7	10	17	17	24	31	34

Mochila - Exemplo

- Exemplo: $c = \{10, 7, 25, 24\}$, $w = \{2, 1, 6, 5\}$ e $W = 7$.

k \ d	0	1	2	3	4	5	6	7
0	0	0	0	0	0	0	0	0
1	0	0	10	10	10	10	10	10
2	0	7	10	17	17	17	17	17
3	0	7	10	17	17	17	25	32
4	0	7	10	17	17	24	31	34

Programação dinâmica versus memorização

- Para casos particulares, uma implementação com memorização pode ser muito mais eficiente (embora não seja polinomial ainda).
- Por exemplo, suponha que os pesos dos itens são múltiplos de 7. Os valores dos subproblemas podem ser armazenados em uma tabela de hash.

Mochila - Complexidade

- Claramente, a complexidade do algoritmo de programação dinâmica para o problema da mochila é $O(nW)$.
- É um algoritmo **pseudo-polinomial**: sua complexidade depende do **valor** de W , parte da entrada do problema.
- O algoritmo não devolve o subconjunto de valor total máximo, apenas o valor máximo.
- Veremos daqui a pouco que é fácil recuperar o subconjunto a partir da tabela z preenchida.

Mochila - Recuperação da Solução

MochilaSolucao(z, n, W)

- Entrada: Tabela z preenchida, capacidade W da mochila e número de itens n .
- Saída: O vetor x que indica os itens colocados na mochila.
para $i := 1$ **até** n **faça** $x[i] := 0$
MochilaSolucaoAux(x, z, n, W)
devolva (x)

MochilaSolucaoAux(x, z, k, d)

- se** $k \neq 0$ **então**
se $z[k, d] = z[k - 1, d]$ **então**
 $x[k] := 0$; *MochilaSolucaoAux*($x, z, k - 1, d$)
senão
 $x[k] := 1$; *MochilaSolucaoAux*($x, z, k - 1, d - w_k$)

Mochila - Exemplo

- Exemplo: $c = \{10, 7, 25, 24\}$, $w = \{2, 1, 6, 5\}$ e $W = 7$.

k \ d	0	1	2	3	4	5	6	7
0	0	0	0	0	0	0	0	0
1	0	0	10	10	10	10	10	10
2	0	7	10	17	17	17	17	17
3	0	7	10	17	17	17	25	32
4	0	7	10	17	17	24	31	34

$x[4] = ?$

Mochila - Exemplo

- Exemplo: $c = \{10, 7, 25, 24\}$, $w = \{2, 1, 6, 5\}$ e $W = 7$.

k \ d	0	1	2	3	4	5	6	7
0	0	0	0	0	0	0	0	0
1	0	0	10	10	10	10	10	10
2	0	7	10	17	17	17	17	17
3	0	7	10	17	17	17	25	32
4	0	7	10	17	17	24	31	34

$x[3] = ?$ $x[4] = 1$

Mochila - Exemplo

- Exemplo: $c = \{10, 7, 25, 24\}$, $w = \{2, 1, 6, 5\}$ e $W = 7$.

k \ d	0	1	2	3	4	5	6	7
0	0	0	0	0	0	0	0	0
1	0	0	10	10	10	10	10	10
2	0	7	10	17	17	17	17	17
3	0	7	10	17	17	17	25	32
4	0	7	10	17	17	24	31	34

$x[2] = ?$ $x[3] = 0$ $x[4] = 1$

Mochila - Exemplo

- Exemplo: $c = \{10, 7, 25, 24\}$, $w = \{2, 1, 6, 5\}$ e $W = 7$.

k \ d	0	1	2	3	4	5	6	7
0	0	0	0	0	0	0	0	0
1	0	0	10	10	10	10	10	10
2	0	7	10	17	17	17	17	17
3	0	7	10	17	17	17	25	32
4	0	7	10	17	17	24	31	34

$x[1] = ?$ $x[2] = 0$ $x[3] = 0$ $x[4] = 1$

Mochila - Exemplo

- Exemplo: $c = \{10, 7, 25, 24\}$, $w = \{2, 1, 6, 5\}$ e $W = 7$.

d \ k	0	1	2	3	4	5	6	7
0	0	0	0	0	0	0	0	0
1	0	0	10	10	10	10	10	10
2	0	7	10	17	17	17	17	17
3	0	7	10	17	17	17	25	32
4	0	7	10	17	17	24	31	34

$$x[1] = 1 \quad x[2] = 0 \quad x[3] = 0 \quad x[4] = 1$$

Mochila - Complexidade

- O algoritmo de recuperação da solução tem complexidade $O(n)$.
- Portanto, a complexidade de tempo e de espaço do algoritmo de programação dinâmica para o problema da mochila é $O(nW)$.
- É possível economizar memória, registrando duas linhas: a que está sendo preenchida e a anterior. Mas isso inviabiliza a recuperação da solução.

Subcadeia comum máxima

Definição: Subcadeia

Dada uma cadeia $S = a_1 \dots a_n$, dizemos que $S' = b_1 \dots b_p$ é uma *subcadeia* de S se S' pode ser obtido a partir de S apagando-se alguns caracteres.

- Exemplo:** considere a cadeia $S = ABCDEFG$.
 $ADFG$, B , $ABCDEFG$ são subcadeias de S .
 IJK , $ACGE$, $AADE$ **não** são subcadeias de S .

Problema da Subcadeia Comum Máxima

Dadas duas cadeias de caracteres X e Y de um alfabeto Σ , determinar a maior subcadeia comum de X e Y

Subcadeia comum máxima (cont.)

- É um problema de otimização. **Será que tem subestrutura ótima?**
- Notação:** Seja S uma cadeia de tamanho n . Para todo $i = 1, \dots, n$, o prefixo de tamanho i de S será denotado por S_i .
- Exemplo:** Para $S = ABCDEFG$, $S_2 = AB$ e $S_4 = ABCD$.
- Definição:** $c[i, j]$ é o tamanho da subcadeia comum máxima dos prefixos X_i e Y_j . Logo, se $|X| = m$ e $|Y| = n$, $c[m, n]$ é o valor ótimo.

Subcadeia comum máxima (cont.)

- **Teorema (subestrutura ótima):** Seja $Z = z_1 \dots z_k$ a subcadeia comum máxima de $X = x_1 \dots x_m$ e $Y = y_1 \dots y_n$, denotado por $Z = \text{SCM}(X, Y)$.
 - 1 Se $x_m = y_n$ então $z_k = x_m = y_n$ e $Z_{k-1} = \text{SCM}(X_{m-1}, Y_{n-1})$.
 - 2 Se $x_m \neq y_n$ então $z_k \neq x_m$ implica que $Z = \text{SCM}(X_{m-1}, Y)$.
 - 3 Se $x_m \neq y_n$ então $z_k \neq y_n$ implica que $Z = \text{SCM}(X, Y_{n-1})$.

Fórmula de Recorrência:

$$c[i, j] = \begin{cases} 0 & \text{se } i = 0 \text{ ou } j = 0 \\ c[i-1, j-1] + 1 & \text{se } i, j > 0 \text{ e } x_i = y_j \\ \max\{c[i-1, j], c[i, j-1]\} & \text{se } i, j > 0 \text{ e } x_i \neq y_j \end{cases}$$

Subcadeia comum máxima (cont.)

SCM(X, m, Y, n, c, b)

01. para $i = 0$ até m faça $c[i, 0] := 0$
02. para $j = 1$ até n faça $c[0, j] := 0$
03. para $i = 1$ até m faça
04. para $j = 1$ até n faça
05. se $x_i = y_j$ então
06. $c[i, j] := c[i-1, j-1] + 1$; $b[i, j] := "\diagdown"$
07. senão
08. se $c[i, j-1] > c[i-1, j]$ então
09. $c[i, j] := c[i, j-1]$; $b[i, j] := "\leftarrow"$
10. senão
11. $c[i, j] := c[i-1, j]$; $b[i, j] := "\uparrow"$;
12. devolva $(c[m, n], b)$.

Subcadeia comum máxima - Exemplo

- **Exemplo:** $X = abcb$ e $Y = bdcab$, $m = 4$ e $n = 5$.

X \ Y	b	d	c	a	b
0	0	0	0	0	0
a 1	0	0	0	1	1
b 2	0	1	1	1	2
c 3	0	1	1	2	2
b 4	0	1	1	2	3

X \ Y	b	d	c	a	b
0					
a 1		↑	↑	↘	←
b 2		↘	←	←	↑
c 3		↑	↑	↘	↑
b 4		↘	↑	↑	↘

Subcadeia comum máxima - Complexidade

- Claramente, a complexidade do algoritmo é $O(mn)$.
- O algoritmo não encontra a subcadeia comum de tamanho máximo, apenas seu tamanho.
- Com a tabela b preenchida, é fácil encontrar a subcadeia comum máxima.

Subcadeia comum máxima (cont.)

- Para recuperar a solução, basta chamar *Recupera_MSC*(*b*, *X*, *m*, *n*).

Recupera_SCM(*b*, *X*, *i*, *j*)

1. **se** $i = 0$ e $j = 0$ **então** **retorne**
2. **se** $b[i, j] = "\backslash"$ **então**
3. *Recupera_MSC*(*b*, *X*, $i - 1$, $j - 1$); **imprima** x_i
4. **senão**
5. **se** $b[i, j] = "\uparrow"$ **então**
6. *Recupera_MSC*(*b*, *X*, $i - 1$, *j*)
7. **senão**
8. *Recupera_MSC*(*b*, *X*, *i*, $j - 1$)

Subcadeia comum máxima - Complexidade

- Dada *b* podemos determinar a subcadeia comum máxima em tempo $O(m + n)$ no **pior caso**.
- Portanto, a complexidade de tempo e de espaço do algoritmo de programação dinâmica para o problema da subcadeia comum máxima é $O(mn)$.
- Note que a tabela *b* é dispensável, podemos economizar memória recuperando a solução a partir da tabela *c*. Ainda assim, o gasto de memória seria $O(mn)$.
- Caso não haja interesse em determinar a subcadeia comum máxima, mas apenas seu tamanho, é possível reduzir o gasto de memória para $O(\min\{n, m\})$: basta registrar apenas a linha da tabela sendo preenchida e a anterior.