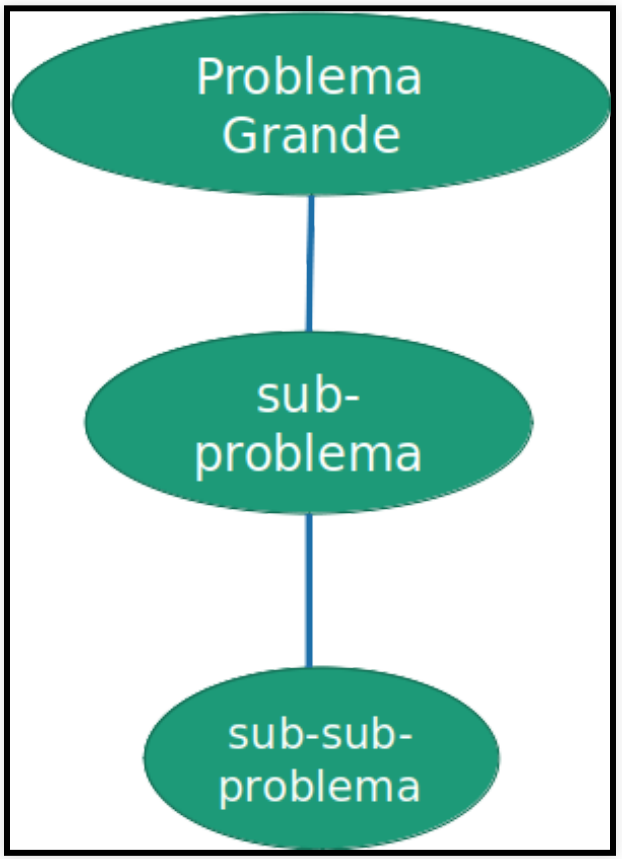
Análise de Algoritmos

Ronaldo Cristiano Prati

Bloco A, sala 513-2

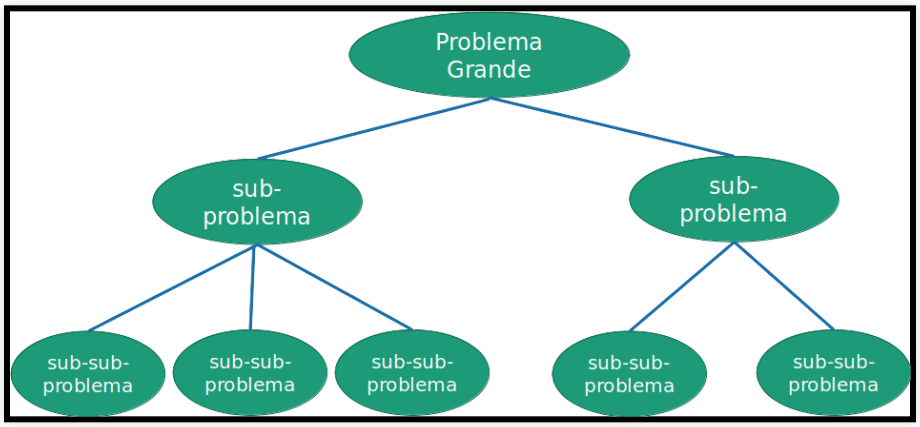
ronaldo.prati@ufabc.edu.br

Abordagem Gulosa



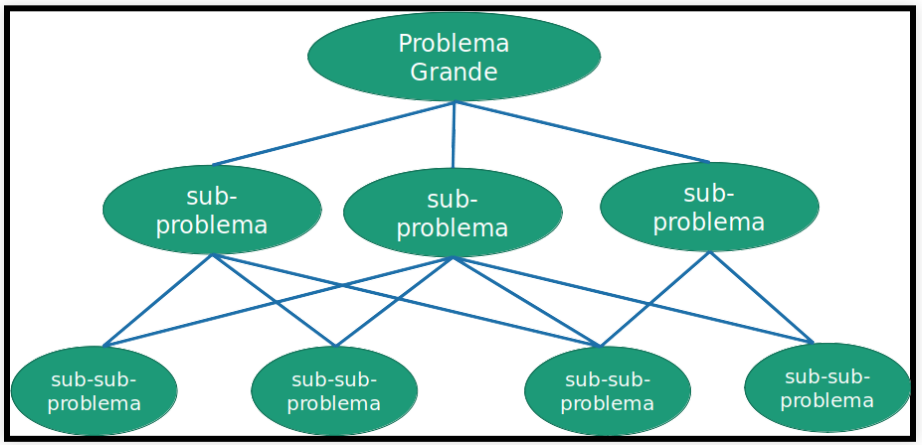
- Gulosamente toma uma decisão local para reduzir o problema
- Essa decisão faz parte da solução ótima
- Problema tem uma sub-estrutura ótima
- Resolução do sub-problema só depende dele mesmo

Abordagem dividir-e-conquistar



- Divide o problema grande em problemas menores
- (Recursivamente) resolve os problemas menores
- Junta as soluções dos problemas menores para resolver o problema maior

Programação dinâmica



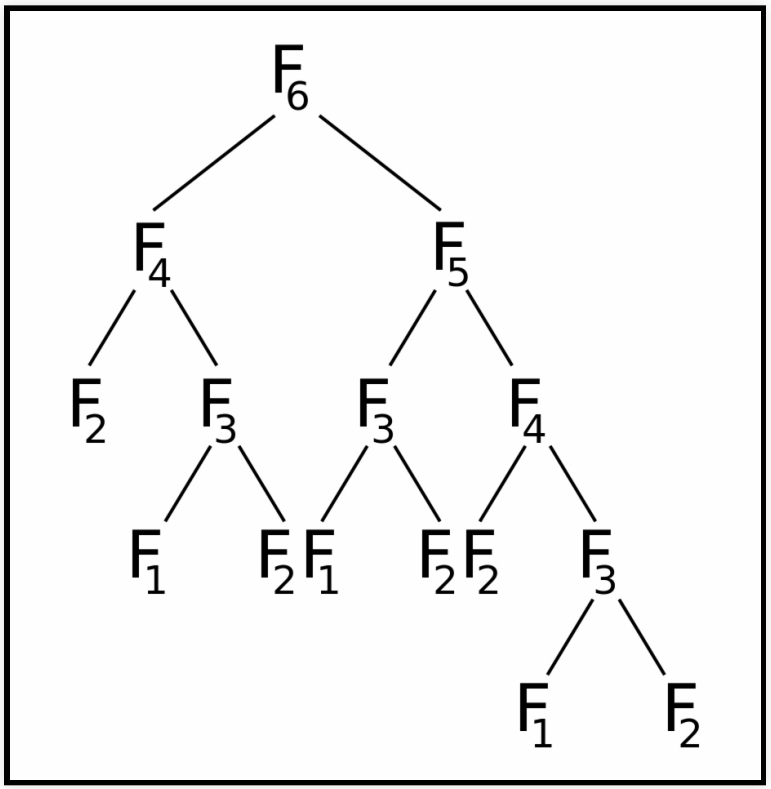
- Divide o problema grande em problemas menores
- A solução do problema grande pode ser descrita em termos das soluções menores
- Os subproblemas tem **sobreposição**
- Armazena as soluções intermediárias para não precisar recalcular

Sequência de Fibonacci

- Por exemplo, considere a sequência de Fibonacci:

$$F_n = \begin{cases} 1 & \text{se } n = 1 \\ 1 & \text{se } n = 2 \\ F_{n-1} + F_{n-2} & \text{se } n > 2 \end{cases}$$

Sequência de Fibonacci



Fibonacci recursivo

Algoritmo 43: FIBONACCIRECURSIVO(n)

```
1 se n ≤ 2 então
2     retorna 1
3 retorna FIBONACCIRECURSIVO(n − 1) + FIBONACCIRECURSIVO(n − 2)
```

- Cuja complexidade é dada pela equação de recorrência:

$$T(n) = T(n-1) + T(n-2) + 1$$

Qual é a complexidade?

- Vamos usar o método da substituição para mostrar que $\Omega\left(\left(\frac{1+\sqrt{5}}{2}\right)^n\right)$
- Inicialmente, vamos mostrar que $T(n) \geq x^n$ para $x > 1$.
- Seja $T(1) = 1$, e $T(2) = 3$, e $n \geq 2$
- Suponha que $T(m) \geq x^n$, para $2 \leq m \leq n - 1$. Assim, aplicando a $T(n)$, temos:

$$\begin{aligned} T(n) &= T(n-1) + T(n-2) + 1 \\ &\geq x^{n-1} + x^{n-2} \\ &\geq x^{n-2}(1 + x) \end{aligned}$$

Qual é a complexidade?

- Note que $1 + x \geq x^2$ para $(1 - \sqrt{5})/2 \leq x \leq (1 + \sqrt{5})/2$.
- Portanto, fazendo $x = (1 + \sqrt{5})/2$ e substituindo em $T(n)$, temos

$$\begin{aligned} T(n) &\geq \left(\frac{1+\sqrt{5}}{2}\right)^{n-2} \left(1 + \left(\frac{1+\sqrt{5}}{2}\right)\right) \\ &\geq \left(\frac{1+\sqrt{5}}{2}\right)^{n-2} \left(\frac{1+\sqrt{5}}{2}\right)^{2} \\ &= \left(\frac{1+\sqrt{5}}{2}\right)^{n} \end{aligned}$$

Dá pra fazer melhor?

- Observe que na abordagem puramente recursiva, precisamos resolver o mesmo problema várias vezes.
- Podemos criar uma memória auxiliar para armzenar valores intermediários.
  - Criar um vetor de tamanho n
  - A medida que formos calculando F_i, armanezenamos esse valor para consultar posteriormente.

Fibonacci com memória

Algoritmo 44: FIBONACCI-TOPDOWN(n)

```
1 Cria vetor F[1..n] global
2 para i = 1 até n faça
3     F[i] = −1
4 retorna FIBONACCIRECURSIVO-TOPDOWN(n)
```

Algoritmo 45: FIBONACCIRECURSIVO-TOPDOWN(n)

```
1 se n ≤ 2 então
2     retorna 1
3 se F[n] ≥ 0 então
4     retorna F[n]
5 F[n] = FIBONACCIRECURSIVO-TOPDOWN(n − 1) +
    FIBONACCIRECURSIVO-TOPDOWN(n − 2)
6 retorna F[n]
```

Fibonacci com memória

- Uma outra abordagem é preencher o vetor em uma abordagem bottom-up

Algoritmo 46: FIBONACCI-BOTTOMUP(n)

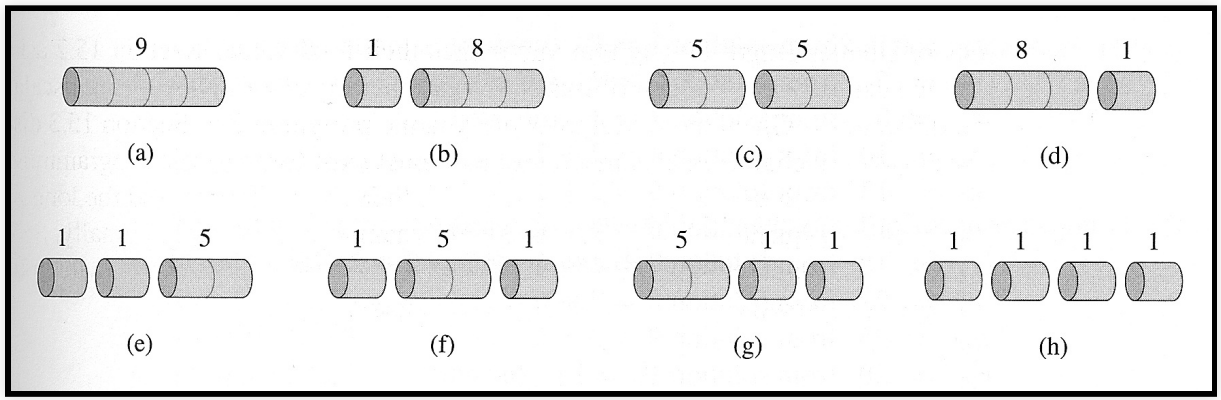
```
1 se i ≤ 2 então
2     retorna 1
3 Seja F[1..n] um vetor de tamanho n
4 F[1] = 1
5 F[2] = 1
6 para i = 3 até n faça
7     F[i] = F[i − 1] + F[i − 2]
8 retorna F[n]
```

Fibonacci com memória

- Esse é um exemplo de programação dinâmica!
- É fácil ver que a complexidade do algoritmo é $O(n)$ já que a consulta ao vetor pode ser feita em $O(1)$
- Entretanto, precisamos de uma memória auxiliar para armazenar os subproblemas já solucionados

Corte da barra de ferro

- Considere um problema em que temos uma barra de ferro de tamanho n, e queremos dividí-la em pedaços, cujos tamanho também são inteiros



- Suponha também que os pedaços tem preços diferentes

p_1	p_2	p_3	p_4	p_5	p_6	p_7	p_8	p_9	p_{10}
1	5	8	9	10	17	17	20	24	30

Corte da barra de ferro

Tamanho	Lucro	Peças
1	1	1 (sem corte)
2	5	2 (sem corte)
3	8	3 (sem corte)
4	10	2 e 2
5	13	2 e 3
6	17	6 (sem corte)
7	18	1 e 6 ou 2, 2 e 3
8	22	2 e 6

Tamanho	Lucro	Peças
9	25	3 e 6
10	30	10

Corte da barra de ferro

- Quantas combinações de corte existem?
- Cada ponto de corte é uma variável aleatória binária (0 ou 1)
- Temos $n-1$ possíveis pontos de corte
- O total de possibilidades é 2^{n-1}

Corte da barra de ferro

- Uma abordagem simplista consiste em gerar todas as possíveis combinações
- Calcular o lucro de cada uma delas
- Selecionar aquela que maximiza o lucro
- Claramente essa abordagem tem uma complexidade $O(2^n)$, já que temos 2^{n-1} combinações

Corte da barra de ferro

- Uma outra possibilidade é um algoritmo recursivo
- Um algoritmo recursivo para o problema do corte da barra escolhe uma posição p_i para fazer o corte e faz uma chamada recursiva
- Como não sabemos qual o tamanho do primeiro corte, testamos todas as possibilidades de 1 a n

Corte da barra de ferro recursivo

Algoritmo 47: CORTEBARRAS(n, p)

```
1 se n == 0 então
2     retorna 0
3 lucro = −1
4 para i = 1 até n faça
5     valor = p_i + CORTEBARRAS(n − i, p)
6     se valor > lucro então
7         lucro = valor
8 retorna lucro
```

Corte da barra de ferro recursivo

- A equação de recorrência desse algoritmo é

$$T(n) = 1 + \sum_{i=1}^{n} T(n-i)$$

- Podemos usar o método da substituição para mostrar que esse algoritmo é $O(2^n)$. Suponha que $T(m) \geq 2^m$ para $0 \leq m \leq n-1$

$$\begin{aligned} T(n) &= 1 + T(0) + T(1) + \cdots + T(n-1) \\ &\geq 1 + \left(2^0 + 2^1 + \cdots + 2^{n-1}\right) \\ &= 2^n \end{aligned}$$

Corte da barra de ferro recursivo

Corte da barra de ferro

- O problema do corte da barra de ferro é um problema de otimização
- Ele tem a propriedade de subestrutura ótima
  - Você precisa saber o corte ótimo do sub-problema para calcular o corte ótimo
- Tem uma solução recursiva exponencial

Corte da barra de ferro

- Vamos usar um vetor B de n posições para armezar o custo de um sub-problema já resolvido
- Além disso, um vetor S em que a posição $S[j]$ indica a posição em que a barra de tamanho j deve ser cortada

Corte da barra de ferro

Algoritmo 48: CorteBarras-TopDown(n, p)

```
1 Cria vetores B[0..n] e S[0..n] globais
2 B[0] = 0
3 para i = 1 até n faça
4 |   B[i] = −1
5 retorna CorteBarrasRecursivo-TopDown(n, p)
```

Algoritmo 49: CorteBarrasRecursivo-TopDown(m, p)

```
1 se B[m] == −1 então
2 |   lucro = −1
3 |   para i = 1 até m faça
4 |   |   valor = p_i + CorteBarrasRecursivo-TopDown(m − i, p)
5 |   |   se valor > lucro então
6 |   |   |   lucro = valor
7 |   |   |   S[m] = i
8 |   B[m] = lucro
9 retorna B[m]
```

Corte da barra de ferro

- Considerando os preços

p_1	p_2	p_3	p_4	p_5	p_6	p_7	p_8	p_9	p_{10}
1	5	8	9	10	17	17	20	24	30

- O algoritmo calcula

i	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
$B[i]$	0	1	5	8	10	13	17	18	22	25	30
$S[i]$	0	1	2	3	2	2	6	1	2	3	10

Corte da barra de ferro

- O primeiro passo do algoritmo é verificar se o subproblema já foi resolvido.
- Caso não esteja resolvido, ele faz isso de uma maneira parecida com o algoritmo `CorteBarra`.
- A diferença é que o primeiro local pra fazer o corte de uma barra de tamanho i é armazenada em $S[i]$, e o maior lucro em $B[i]$.

Complexidade

- Observe que a cada chamada de `CorteBarraRecursivoTopDown`, um subproblema que já foi resolvido retorna imediatamente.
- As demais linhas são executadas em tempo constante.
- Como armazenamos o resultado sempre que resolvemos um subproblema, cada subproblema é resolvido uma vez.
- O tempo de execução é então

$$T(m) = 1 + 2 + \cdots + m = \Theta\left(m^2\right)$$

Impressão dos Cortes

- Para saber os pontos de corte, podemos percorrer o vertor $S[i]$ para encontrar os pontos de corte.
- Observe que em $S[n]$, temos o tamanho do primeiro corte.
- Após esse corte, a barra tem tamanho $n - S[-]$

Algoritmo 50: ImprimeCortes(n, S)

```
1 enquanto n > 0 faça
2     Imprime S[n]
3     n = n − S[n]
```

Abordagem BottomUP

- Também podemos ter uma abordagem BottomUp

Algoritmo 51: CORTEBARRAS-BOTTOMUP(n, p)

```
1  Cria vetores B[0..n] e S[0..n]
2  B[0] = 0
3  para i = 1 até n faça
4      lucro = −1
5      para j = 1 até i faça
6          se p_j + B[i − j] > lucro então
7              lucro = p_j + B[i − j]
8              S[i] = j
9      B[i] = lucro
10 retorna B[n]
```

Mochila inteira

- Dado um conjunto $I = \{1, 2, \cdots, n\}$ contendo n itens
- Cada item $i \in I$ tem um peso w_i e um valor v_i
- Como podemos selecionar um subconjutno de $S \subseteq I$ uma mochila com capacidade W tal que:
  - O peso dos itens selecionados cabe na mochila: $\sum_{i \in S} w_i \leq W$
  - O valor dos itens selecionados é maximo: $\max \sum_{i \in S} v_i$

Mochila inteira

- Por exemplo, dado o seguinte conjunto de itens

item	**1**	**2**	**3**	**4**	**5**
peso	6	2	4	3	11
valor	20	8	14	13	35

- Como podemos selecionar itens para preencher uma mochila de tamanho 10?

Mochila inteira

- Podemos resolver o problema por força bruta enumerando todos os subconjuntos possíveis
- Verificar quais subconjuntos cabem na mochila
- Calcular o valor total dos suconjuntos que cabem e armazenar o maior deles

Quantos subconjuntos existem?

- Para cada item, temos a opção de colocá-lo ou não
- Temos um total de 2^n subconjuntos
- Para cada conjunto, levamos $O(n)$ para verificar se os itens cabem na mochila e qual é o seu valor total
- Ou seja, esse algoritmo leva temp $O(n2^n)$

Subestrutura do problema

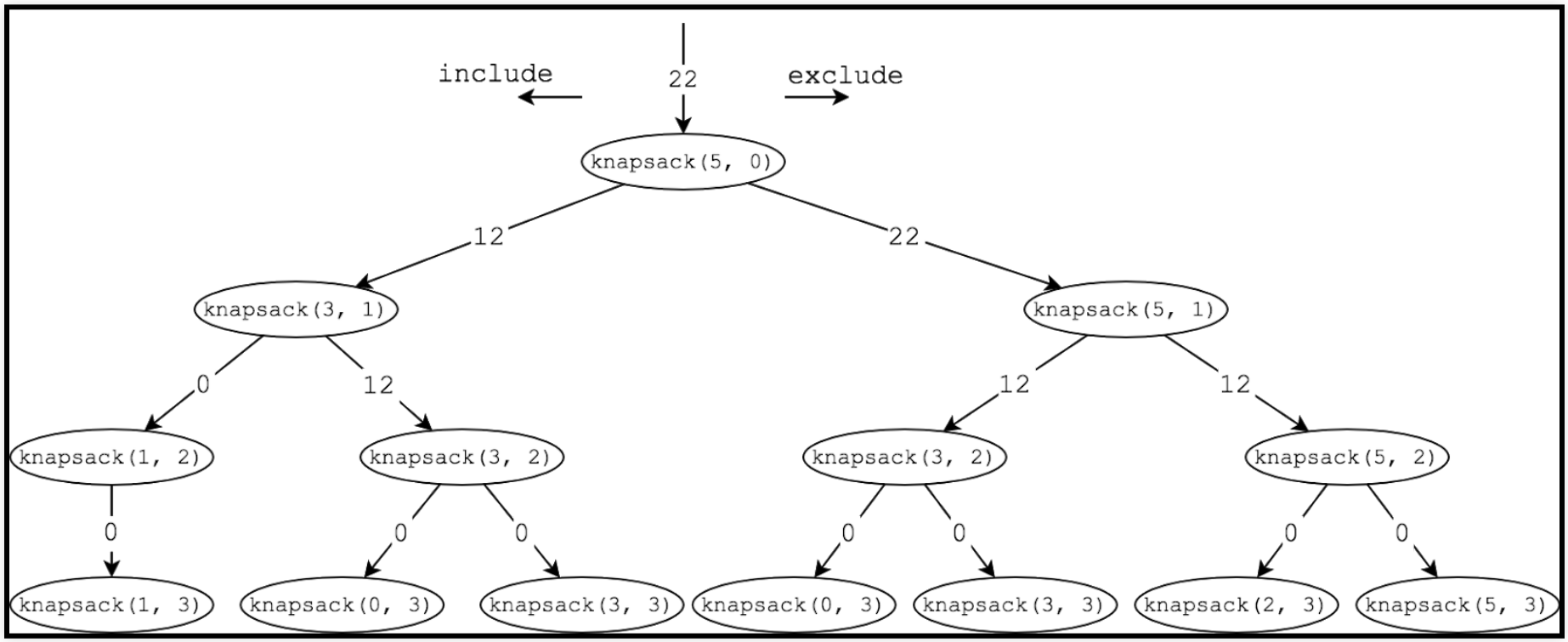
- Seja uma instância do problema por $K\ (I_n, v, wW)$, em que:
  - $I_n = \{1, 2, \cdots, n\}$ um conjunto de itens
  - v é uma lista dos valores dos itens
  - w é uma lista dos pesos dos itens
  - W é a capacidade da mochila

Subestrutura do problema

- S^* é uma solução ótima para $K(I_n, v, w, W)$, e $S^* \subseteq, I_n$ com um valor total ótimo $V_{n,W} = \sum_{i \in S^*} v_i$
  - Se $n \notin S$, então S^* também é uma solução para $K\left(I_{n-1}, W\right)$
  - Se $n \in S$, então S^* também é uma solução para $K\left(I_{n-1}, W - w_n\right)$
- Como saber se n está em S^*?
  - Máximo entre selecionar ou não o item n, se ele cabe na mochila
  - Valor do problema desconsiderando o item n, se ele não cabe na mochila

$$V_{n,W} = \begin{cases} \max\left\{V_{n-1,W}, V_{n-1,W-w_n} + v_n\right\} & \text{se } w_n \leq W \\ V_{n-1,W} & \text{se } w_n > W \end{cases}$$

Mochila inteira



Mochila inteira

- Uma solução direta recursiva baseada nessa regra:

Algoritmo 52: MochilaInteira(n, v, w, W)

```
1 se n == 0 então
2     retorna 0
3 se w_n > W então
4     retorna MochilaInteira(n − 1, v, w, W)
5 senão
6     usa = v_n MochilaInteira(n − 1, v, w, W − w_n)
7     naousa = MochilaInteira(n − 1, v, w, W)
8     retorna max{usa, naousa}
```

Mochila inteira

- A equação de recorrência para esse algoritmo é $T(n) = 2(Tn - 1)$, cuja solução é $O(2^n)$
- Olhando a árvore de recursão, é fácil perceber que esse algoritmo resolve o mesmo subproblema várias vezes.
- Observe que os subproblemas dependem tanto do número de itens n, quanto da capacidade W do subproblema.
  - Poderíamos armazenar as soluções usando um vetor de tamanho nW
  - Entretanto, usar uma matriz $n \times W$ facilita a indexação

Mochila inteira

Algoritmo 53: MochilaInteira-TopDown(n, v, w, W)

```
1 Seja M[0..n][0..W] uma matriz global
2 para x = 0 até W faça
3 |   M[0][x] = 0 para j = 1 até n faça
4 |   |   M[j][x] = −1
5 retorna MochilaInteiraRecursivo-TopDown(n, v, w, W)
```

Algoritmo 54: MochilaInteiraRecursivo-TopDown(n, v, w, W)

```
1 se M[n][W] == −1 então
2 |   se w_n > W então
3 |   |   M[n][W] = MochilaInteira(n − 1, v, w, W)
4 |   senão
5 |   |   usa = v_n MochilaInteira(n − 1, v, w, W − w_n)
6 |   |   naousa = MochilaInteira(n − 1, v, w, W)
7 |   |   M[n][W] = max{usa, naousa}
8 retorna M[n][W]
```

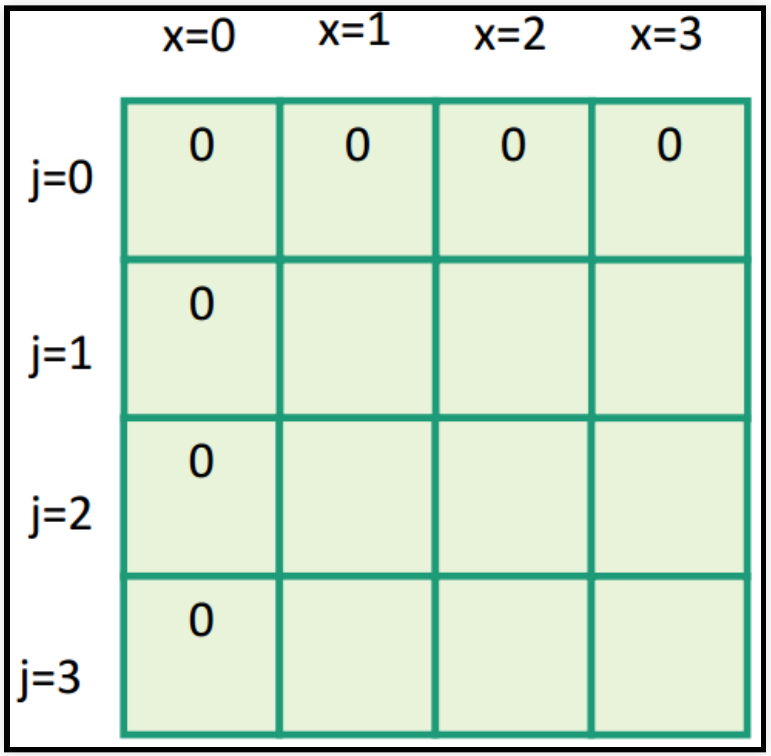
Mochila inteira

Algoritmo 55: MochilaInteira-BottomUp(n, v, w, W)

```
1  Seja M[0..n][0..W] uma matriz
2  para x = 0 até W faça
3      M[0][x] = 0
4  para j = 1 até n faça
5      para x = 0 até W faça
6          se w_n > W então
7              M[j][x] = M[j − 1][x]
8          senão
9              usa = v_j + M[j − 1][x − w_j]
10             naousa = M[j − 1][x]
11             M[j][x] = max{usa, naousa}
12 retorna M[n][W]
```

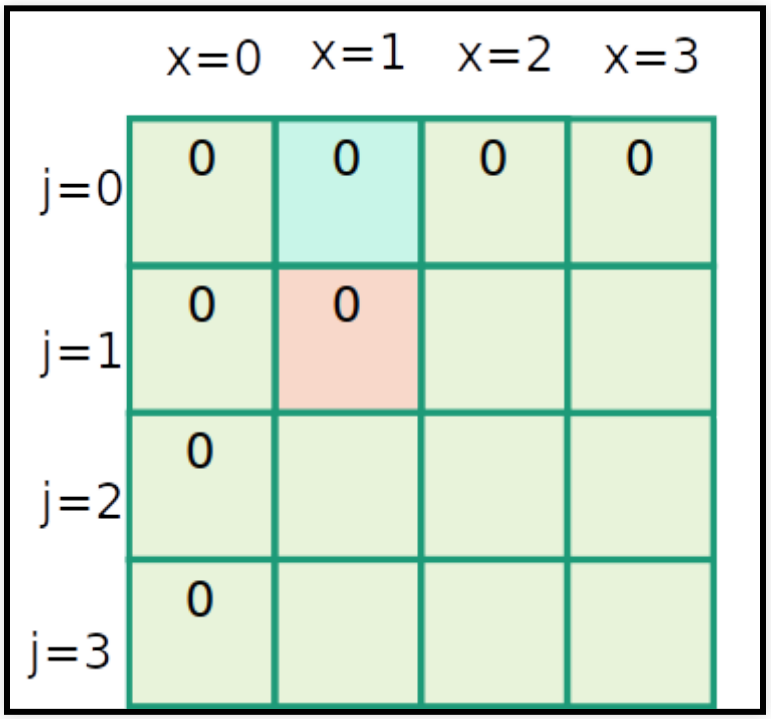
Mochila inteira

item	1	2	3
peso	1	2	3
valor	1	4	6



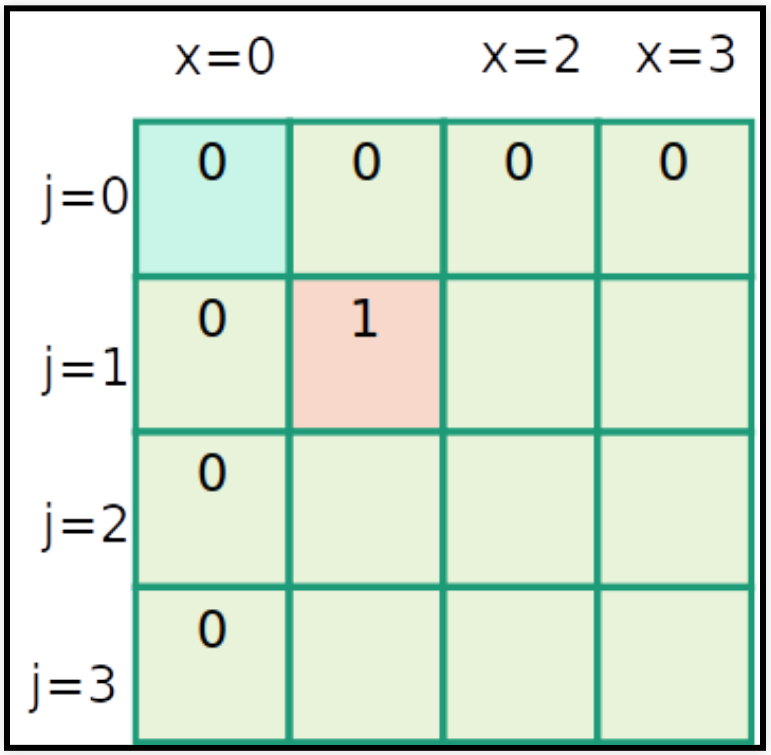
Mochila inteira

item	1	2	3
peso	1	2	3
valor	1	4	6



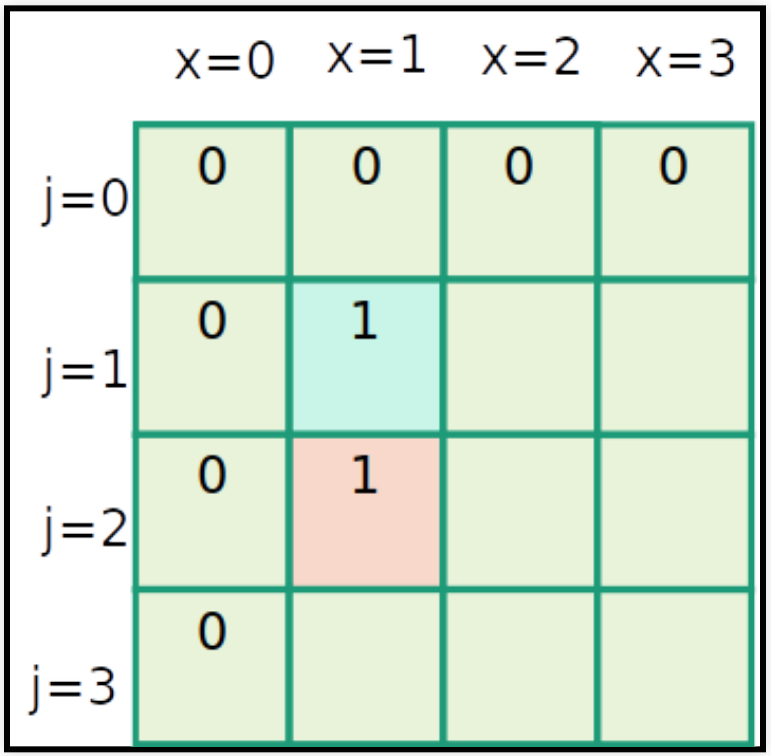
Mochila inteira

item	1	2	3
peso	1	2	3
valor	1	4	6



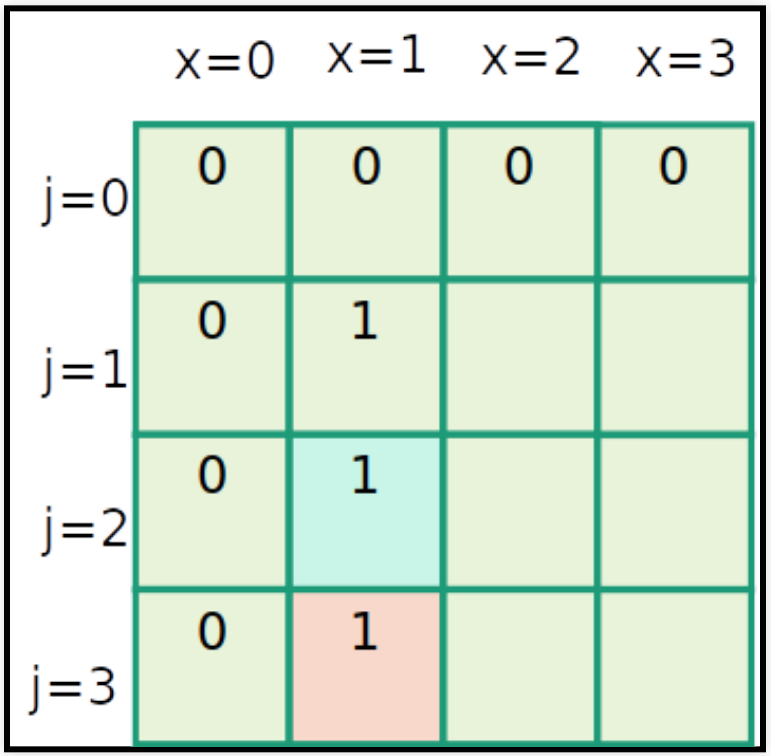
Mochila inteira

item	1	2	3
peso	1	2	3
valor	1	4	6



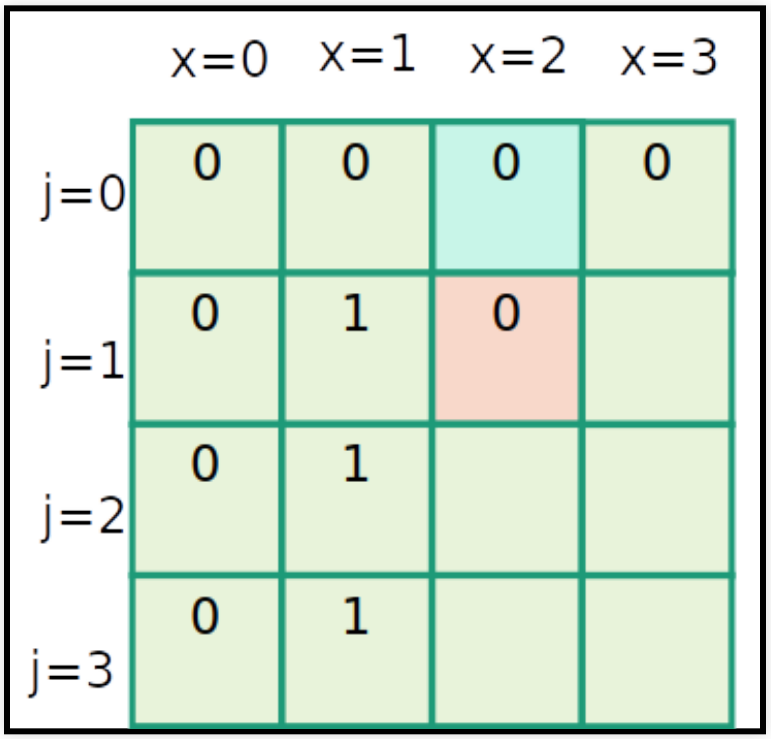
Mochila inteira

item	1	2	3
peso	1	2	3
valor	1	4	6



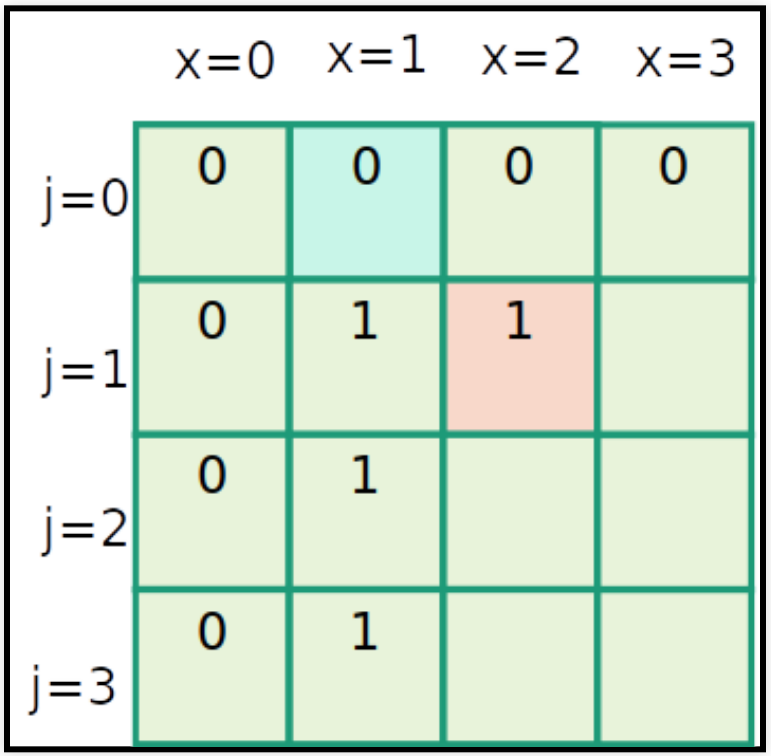
Mochila inteira

item	1	2	3
peso	1	2	3
valor	1	4	6



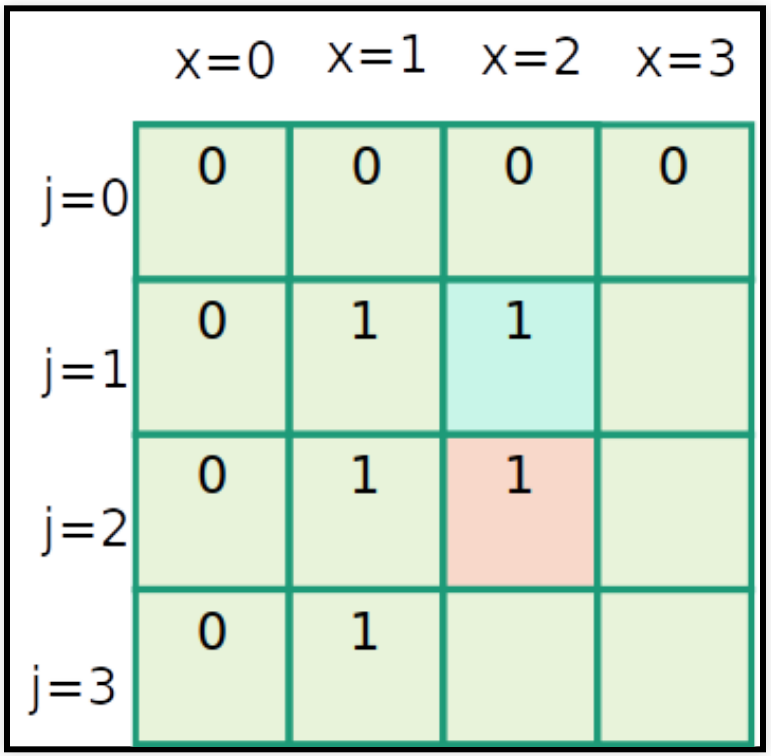
Mochila inteira

item	1	2	3
peso	1	2	3
valor	1	4	6



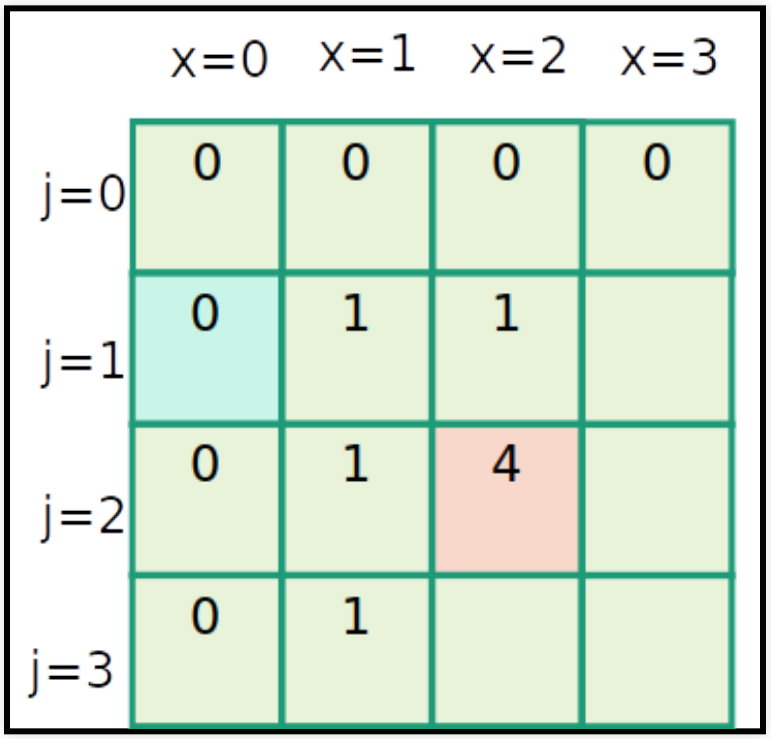
Mochila inteira

item	1	2	3
peso	1	2	3
valor	1	4	6



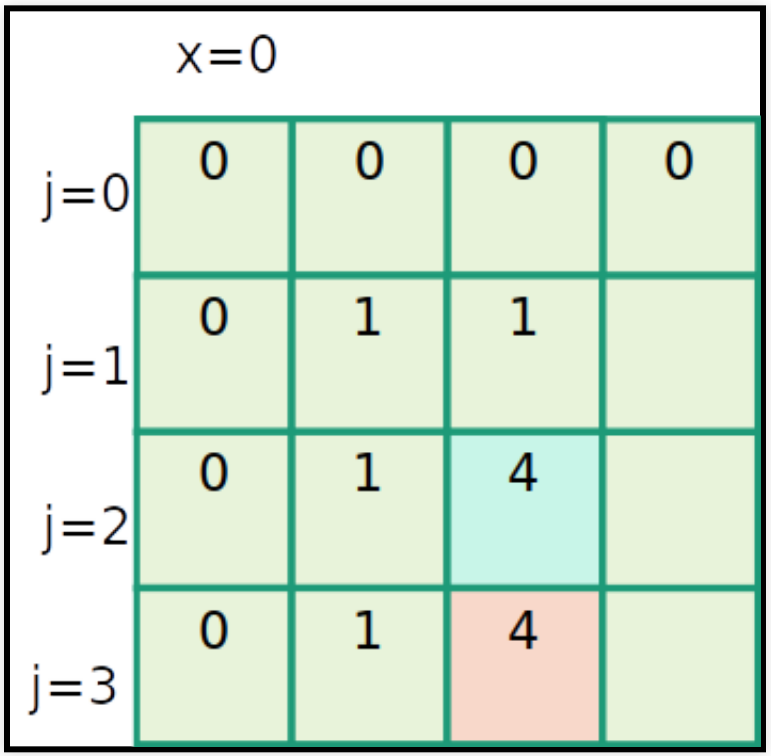
Mochila inteira

item	1	2	3
peso	1	2	3
valor	1	4	6



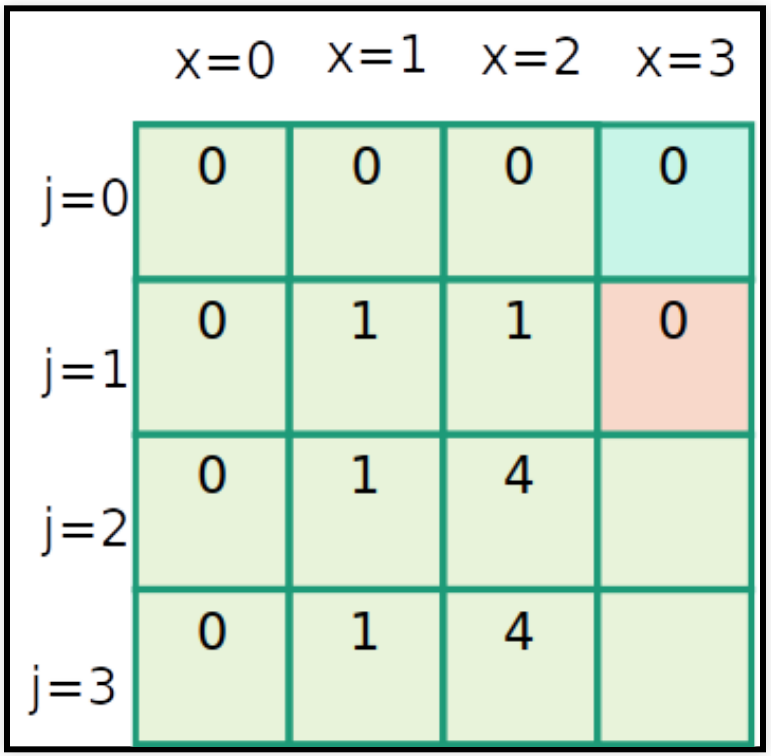
Mochila inteira

item	1	2	3
peso	1	2	3
valor	1	4	6



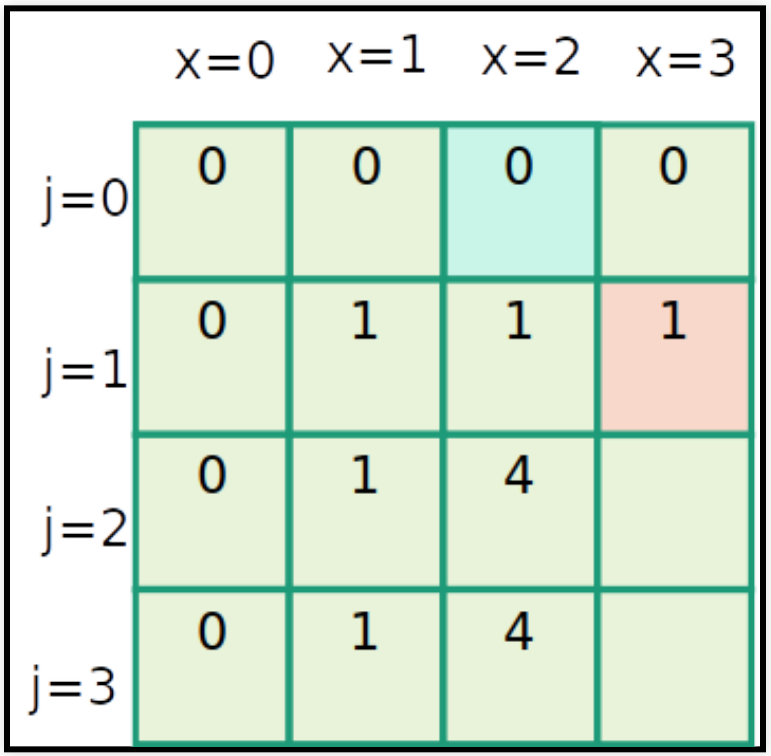
Mochila inteira

item	1	2	3
peso	1	2	3
valor	1	4	6



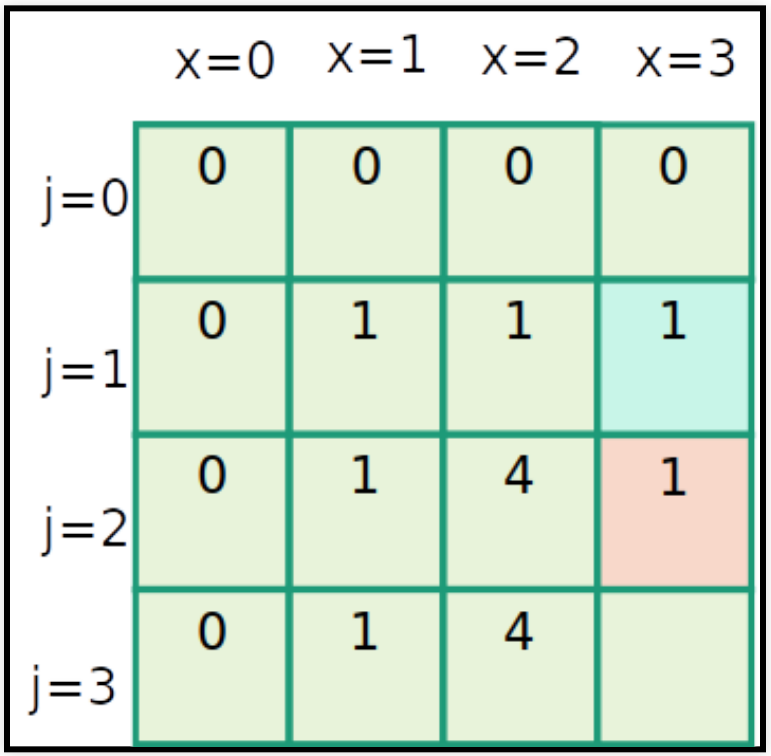
Mochila inteira

item	1	2	3
peso	1	2	3
valor	1	4	6



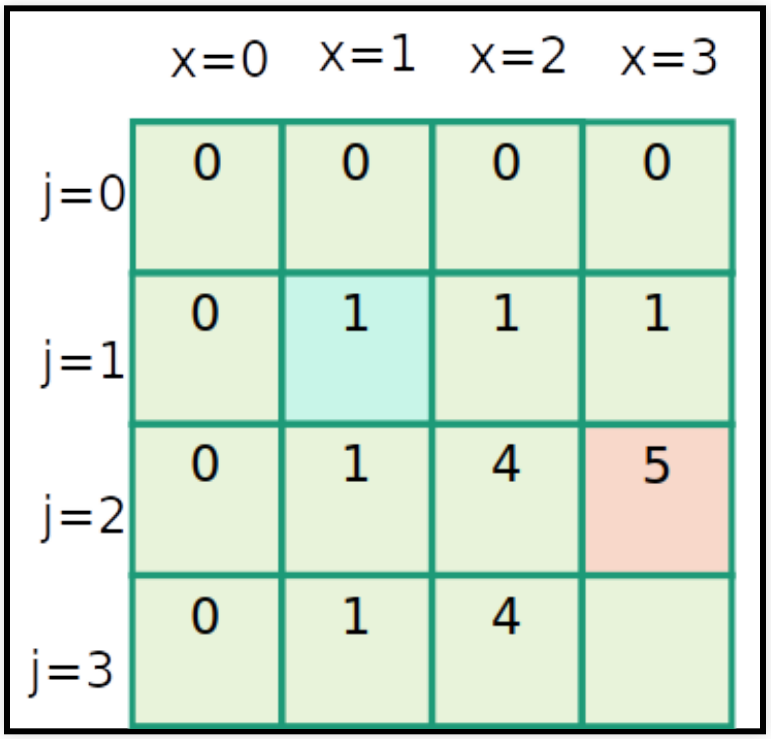
Mochila inteira

item	1	2	3
peso	1	2	3
valor	1	4	6



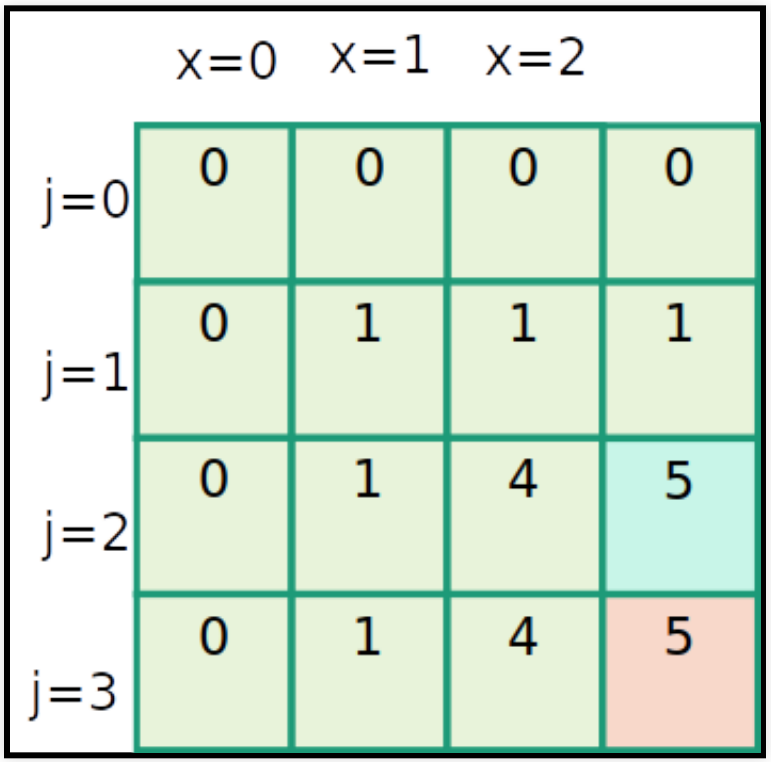
Mochila inteira

item	1	2	3
peso	1	2	3
valor	1	4	6



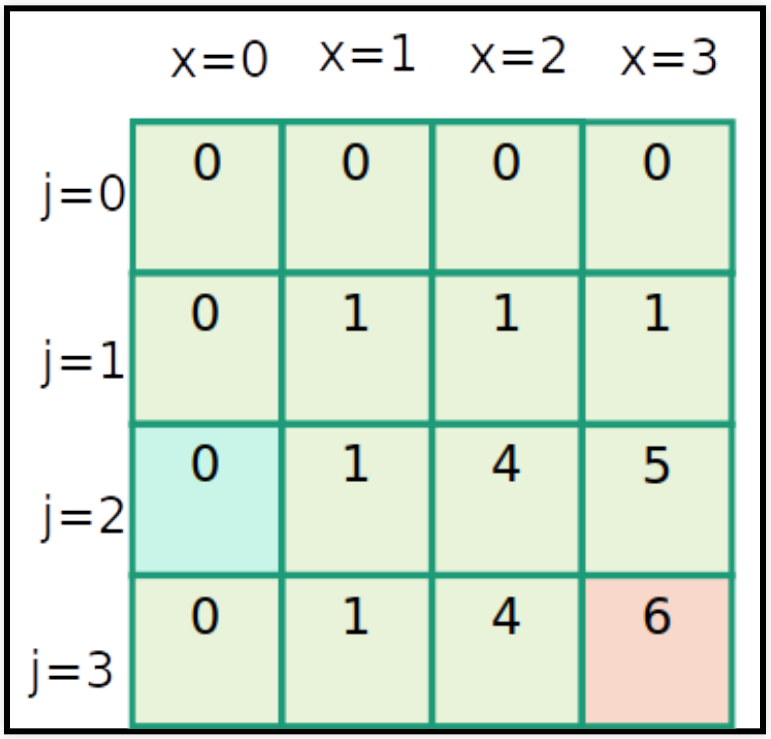
Mochila inteira

item	1	2	3
peso	1	2	3
valor	1	4	6



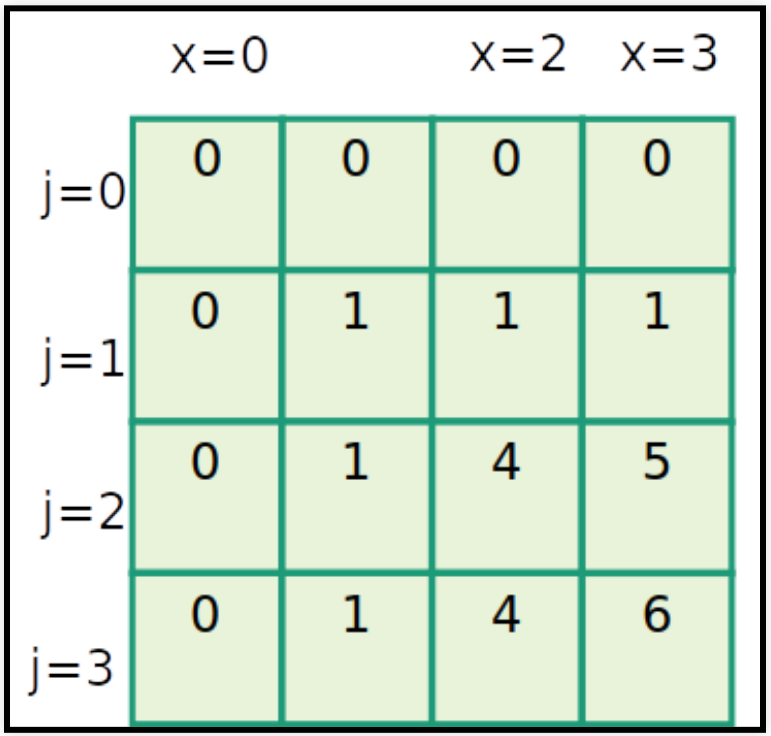
Mochila inteira

item	1	2	3
peso	1	2	3
valor	1	4	6



Mochila inteira

item	1	2	3
peso	1	2	3
valor	1	4	6



Imprimindo a mochila

- Na última célula da matriz $M[n][W]$ contém o valor ótimo.
- Mas não sabemos quais itens devem estar na mochila
- No entanto, a maneira como a matriz foi preenchida nos permite obter quais são esses elementos

Imprimindo a mochila

Algoritmo 56: ConstroiMochila(n, v, w, W, M)

```
1 S = ∅
2 x = W
3 j = n
4 enquanto i ≥ 1 faça
5 |   se M[j][x] == M[j − 1][x − w_j] + v_j então
6 |   |   S = S ∪ {i}
7 |   |   x = x − w_j
8 |   j = j − 1
9 retorna S
```

Complexidade

- É fácil perceber que a complexidade do algoritmo é proporcioal ao tamanho da matriz, ou seja $O(nW)$
- No entanto, observe que precisamos de uma coluna na tabela para cada possível tamanho da mochila dos subproblemas. Esse número, no entanto, não é polinomial no tamanho do da entrada
- Considerando pesos inteiros, teríamos $O(2^{\log W})$ possíveis valores para W, e a complexidade do algoritmo é $O(n2^{\log W})$
- Esse algoritmo é o que chamamos de *pseudo-polinomal*

Programação dinâmica

1. Identificar subestrutura ótima
2. Encontrar uma formulação recursiva
3. Usar programação dinâmica para encontrar o valor da função ótima
4. Se necessário, armazenar informação adicional de tal maneira que no passo 3 podemos encontrar a solução ótima