

Curso Estruturas de Dados e Algoritmos Expert

Prof. Nelio Alves

Programação dinâmica (parte 2)



1

Problemas Clássicos PD - Problema do Troco

<https://devsuperior.com.br>

Prof. Dr. Nelio Alves

2

Problema do Troco (Coin Change)

- Dado um valor V e um conjunto de N moedas, ache o número **mínimo** de moedas que representa V .

Exemplo:

$V = 37$
 $C = \{25, 10, 5, 1\}$



Moedas disponíveis

Solução



$$37 = 25 + 10 + 1 + 1$$

4 moedas

3

Problema do Troco (Coin Change)

Como desenvolver um algoritmo para o problema?

Abordagem 1: Algoritmo Guloso

Solução: Escolha a maior moeda possível a cada passo.

$V = 37$
 $C = \{25, 10, 5, 1\}$



Passo 1
 $37 - 25 = 12$

Passo 2
 $12 - 10 = 2$

Passo 3
 $2 - 1 = 1$

Passo 4
 $1 - 1 = 0$



$$37 = 25 + 10 + 1 + 1$$

4 moedas

4

Problema do Troco (Coin Change)

Como desenvolver um algoritmo para o problema?

Abordagem 1: Algoritmo Guloso

Solução: Escolha a maior moeda possível a cada passo.

Problema: não funciona para qualquer conjunto de moedas!

Caso 1

$$V = 8$$
$$C = \{10, 7, 1\}$$

Solução
 $8 = 7 + 1$
2 moedas

Caso 2

$$V = 14$$
$$C = \{10, 7, 1\}$$

Solução
 $14 = 10 + 1 + 1 + 1 + 1$
5 moedas

Errado!

Solução correta
 $14 = 7 + 7$
2 moedas

5

Problema do Troco (Coin Change)

Como desenvolver um algoritmo para o problema?

Abordagem 2: Força Bruta

Solução: Testar todos os conjuntos de moedas possíveis.
Pode ser implementado recursivamente.

- 1. Formulação de Estado:** definimos a função $f(V)$ como o mínimo de moedas para formar o valor v .
- 2. Casos Base:** se $V = 0$, então $f(0) = 0$
- 3. Transições entre Estados:** em um conjunto C de moedas possíveis, consideramos cada $c_i \in C$, e a relação recursiva é:

$$f(V) = 1 + \min\{f(V - c_1), f(V - c_2), \dots, f(V - c_n)\}, \text{ se } (V - c_i) \geq 0$$

Ou seja, tentamos adicionar cada uma das moedas à solução se o resultado da subtração não for negativo, e adicionamos 1 para contar as moedas.

6

Problema do Troco (Coin Change)

1. **Estado:** $f(V)$ = mínimo de moedas para formar o valor V .
2. **Casos Base:** se $V = 0$, então $f(0) = 0$
3. **Transições entre Estados:**
$$f(V) = 1 + \min\{f(V - c_1), f(V - c_2), \dots, f(V - c_n)\}, \text{ se } (V - c_i) \geq 0$$

```
min_coins(v, c)
  if v == 0
    return 0

  result = Infinity
  for coin in c
    if (v - coin) >= 0
      result = min(result, 1 + min_coins(v - coin, c))

  return result
```

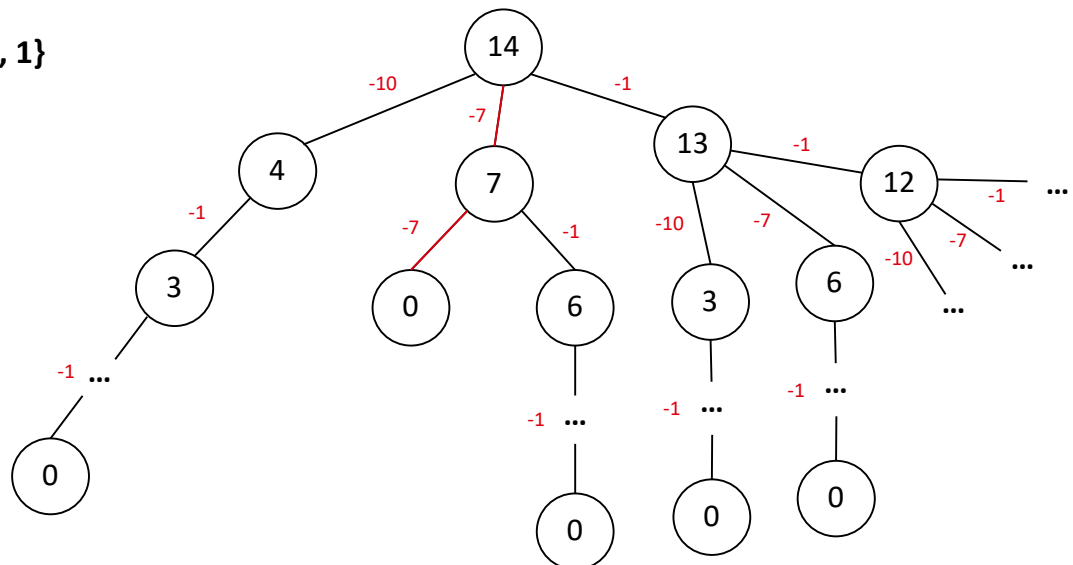
7

Exemplo

$V = 14$

$C = \{10, 7, 1\}$

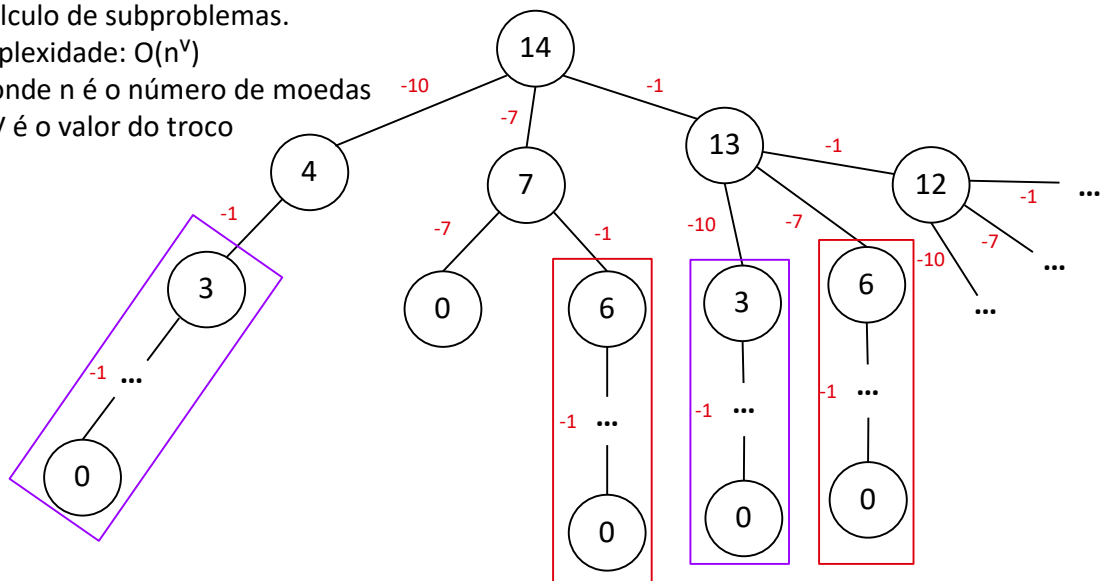
$f(14) = 2$



8

Por que a abordagem atual está lenta?

- Recálculo de subproblemas.
- Complexidade: $O(n^V)$
 - onde n é o número de moedas
 - V é o valor do troco



9

Melhorando a solução com memoização (Abordagem Top-Down)

```
memo = []
min_coins(v, c)
    if memo[v] not null
        return memo[v]

    if v == 0
        return 0

    result = Infinity
    for coin in c
        if (v - coin) >= 0
            result = min(result, 1 + min_coins(v - coin, c))

    memo[v] = result
    return memo[v]
```

Complexidade: $O(n * v)$

10

Abordagem Bottom-Up

```
min_coins(v, c)
    memo = [Infinity] * (v + 1)

    memo[0] = 0
    for i=1 to v
        for coin in c
            if (v - coin) >= 0
                memo[i] = min(memo[i], memo[v - coin] + 1)

    return memo[v]
```

11

Simulação Abordagem Bottom-Up

v = 13

c = {11, 6, 5, 2, 1}

v	0	1	2	3	4	5	6	7	8	9	10	11	12	13
memo	0	INF	INF	INF	INF	INF	INF	INF	INF	INF	INF	INF	INF	INF

```
memo = [Infinity] * (v + 1)
memo[0] = 0
```

- Inicializamos as posições não conhecidas como INF e o caso base como 0.

12

Simulação Abordagem Bottom-Up

$v = 13$

$c = \{11, 6, 5, 2, 1\}$

v	0	1	2	3	4	5	6	7	8	9	10	11	12	13
memo	0	INF	INF	INF	INF	INF	INF	INF	INF	INF	INF	INF	INF	INF

```
for i=1 to v
  for coin in c
    if (v - coin) >= 0
      memo[i] = min(memo[i], memo[v - coin] + 1)
```


- Percorrer cada posição v e tentar minimizar esse subproblema com base nas soluções anteriores

13

Simulação Abordagem Bottom-Up

$v = 13$

$c = \{11, 6, 5, 2, 1\}$



v	0	1	2	3	4	5	6	7	8	9	10	11	12	13
memo	0	1	INF	INF	INF	INF	INF	INF	INF	INF	INF	INF	INF	INF


```
for i=1 to v
  for coin in c
    if (v - coin) >= 0
      memo[i] = min(memo[i], memo[v - coin] + 1)
```

14

Simulação Abordagem Bottom-Up

v = 13

c = {11, 6, 5, 2, 1}



v	0	1	2	3	4	5	6	7	8	9	10	11	12	13
memo	0	1	1	INF	INF	INF	INF	INF	INF	INF	INF	INF	INF	INF


```
for i=1 to v
  for coin in c
    if (v - coin) >= 0
      memo[i] = min(memo[i], memo[v - coin] + 1)
```

15

Simulação Abordagem Bottom-Up

v = 13

c = {11, 6, 5, 2, 1}



v	0	1	2	3	4	5	6	7	8	9	10	11	12	13
memo	0	1	1	2	INF	INF	INF	INF	INF	INF	INF	INF	INF	INF


```
for i=1 to v
  for coin in c
    if (v - coin) >= 0
      memo[i] = min(memo[i], memo[v - coin] + 1)
```

16

Simulação Abordagem Bottom-Up

v = 13

c = {11, 6, 5, 2, 1}



v	0	1	2	3	4	5	6	7	8	9	10	11	12	13
memo	0	1	1	2	2	INF	INF	INF	INF	INF	INF	INF	INF	INF


```
for i=1 to v
  for coin in c
    if (v - coin) >= 0
      memo[i] = min(memo[i], memo[v - coin] + 1)
```

17

Simulação Abordagem Bottom-Up

v = 13

c = {11, 6, 5, 2, 1}



v	0	1	2	3	4	5	6	7	8	9	10	11	12	13
memo	0	1	1	2	2	1	INF	INF	INF	INF	INF	INF	INF	INF

```
for i=1 to v
  for coin in c
    if (v - coin) >= 0
      memo[i] = min(memo[i], memo[v - coin] + 1)
```

18

Simulação Abordagem Bottom-Up

$v = 13$

$c = \{11, 6, 5, 2, 1\}$

v	0	1	2	3	4	5	6	7	8	9	10	11	12	13
memo	0	1	1	2	2	1	1	INF	INF	INF	INF	INF	INF	INF

```
for i=1 to v
  for coin in c
    if (v - coin) >= 0
      memo[i] = min(memo[i], memo[v - coin] + 1)
```

19

Simulação Abordagem Bottom-Up

$v = 13$

$c = \{11, 6, 5, 2, 1\}$

v	0	1	2	3	4	5	6	7	8	9	10	11	12	13
memo	0	1	1	2	2	1	1	2	2	3	2	1	2	2

```
for i=1 to v
  for coin in c
    if (v - coin) >= 0
      memo[i] = min(memo[i], memo[v - coin] + 1)
```

Para cada valor v , passamos cada um dos n tipos de moeda, logo:

Complexidade: $O(v * n)$

20

Problemas Clássicos PD - Caminhos no Grid

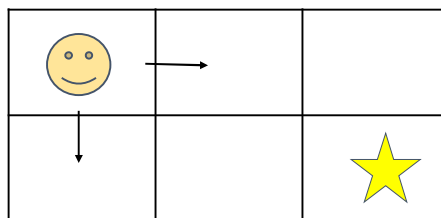
<https://devsuperior.com.br>

Prof. Dr. Nelio Alves

21

Contando Caminhos Únicos em um Grid

- Dado um grid $N \times M$, **de quantas formas** podemos chegar do canto superior esquerdo ao canto inferior direito, sendo que só podemos nos mover para baixo e para a direita?

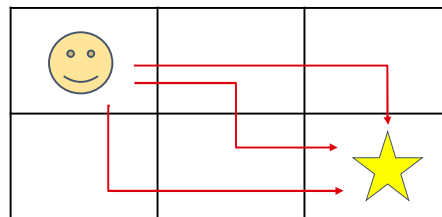


$N = 2, M = 3$

22

Contando Caminhos Únicos em um Grid

- Dado um grid $N \times M$, **de quantas formas** podemos chegar do canto superior esquerdo ao canto inferior direito, sendo que só podemos nos mover para baixo e para a direita?



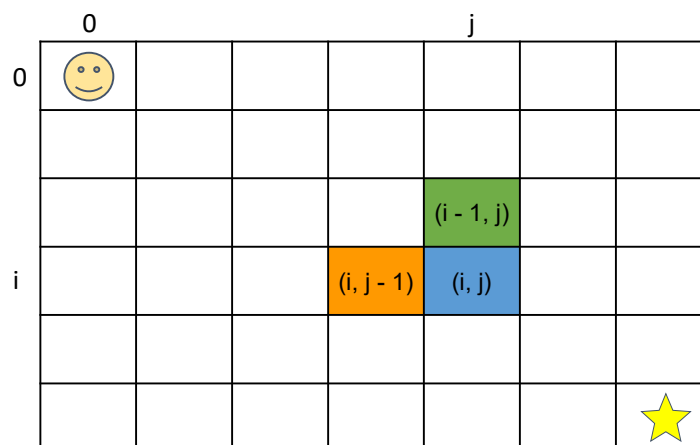
$N = 2, M = 3$

Nesse grid, existem 3 caminhos!

23

Contando Caminhos Únicos em um Grid

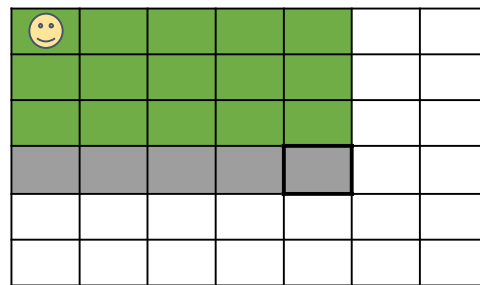
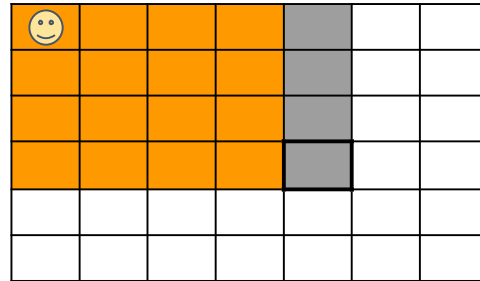
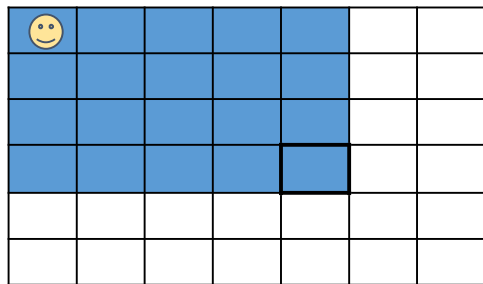
- Identificando subproblemas
 - Para saber de quantas maneiras podemos chegar a uma posição (i, j) , devemos saber de quantas maneiras chegar a $(i - 1, j)$ e $(i, j - 1)$.



24

Contando Caminhos Únicos em um Grid

Contar os caminhos de um grid NxM, envolve contar os caminhos de seus subgrids!



25

Contando Caminhos Únicos em um Grid

Casos triviais

1. Apenas uma coluna



Solução

Em uma coluna, só **existe um caminho** para todas as células (para baixo)

2. Apenas uma linha



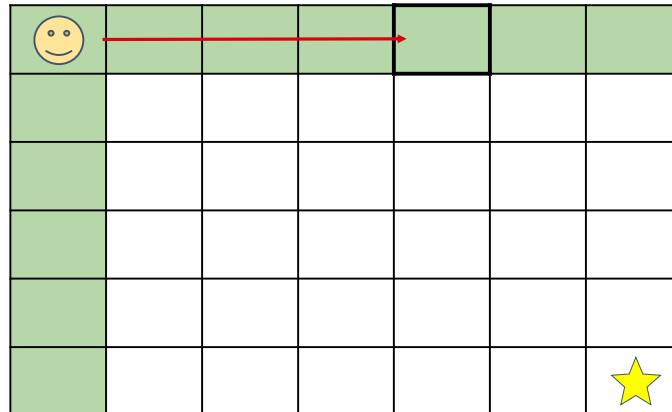
Solução

Em uma linha, só **existe um caminho** para todas as células (para a direita)

26

Contando Caminhos Únicos em um Grid

- Identificando casos base
 - Para a primeira linha, só podemos nos mover para a direita
 - Para a primeira coluna, só podemos nos mover para baixo
 - Existe somente um caminho para células na primeira coluna e na primeira linha



27

Contando Caminhos Únicos em um Grid

- Identificando casos base
 - Existe somente um caminho para células na primeira coluna e na primeira linha
 - Logo, se $i = 0$ ou $j = 0$, a resposta é 1



0	😊	1	1	1	1	1	1
1							
1							
1							
1							
1							★

28

Contando Caminhos Únicos em um Grid

- Juntando as pontas
 - Sabemos que para $i = 0$ ou $j = 0$, a resposta é 1
 - Sabemos que para saber a resposta de (i, j) , precisamos de $(i - 1, j)$ e $(i, j - 1)$

0



0		1	1	1	1	1	1
	1						
	1						
	1						
	1						
	1						

29

Contando Caminhos Únicos em um Grid

- Juntando as pontas
 - se $i = 0$ ou $j = 0$, retorne 1
 - $\text{paths}(i, j) = \text{paths}(i - 1, j) + \text{paths}(i, j - 1)$



0

0		1	1	1	1	1	1
	1						
	1						
	1						
	1						
	1						

30

Contando Caminhos Únicos em um Grid

$$\text{paths}(1, 1) = \text{paths}(0, 1) + \text{paths}(1, 0)$$

0							
0		1	1	1	1	1	1
	1	2					
	1						
	1						
	1						
	1						

31

Escrevendo solução recursiva

- **Casos base**
 - se $i = 0$ ou $j = 0$, retorne 1
- **Relação recursiva**
 - $\text{paths}(i, j) = \text{paths}(i - 1, j) + \text{paths}(i, j - 1)$



```
countPaths(i, j)
  if i == 0 || j == 0
    return 1

  return countPaths(i - 1, j) + countPaths(i, j - 1)
```

Complexidade: $O(2^{(n+m)})$

32

Otimizando código com memoização (Abordagem Top-Down)

```
countPaths(i, j)
  if i == 0 || j == 0
    return 1

  return countPaths(i - 1, j) + countPaths(i, j - 1)
```

Complexidade:
 $O(2^{(n+m)})$



```
memo = [array (n*m)]
countPaths(i, j)
  if memo[i][j] not null
    return memo[i][j]
  if i == 0 || j == 0
    return 1

  memo[i][j] = countPaths(i - 1, j) + countPaths(i, j - 1)
  return memo[i][j]
```

Complexidade:
 $O(N * M)$

33

Otimizando código com tabulação (Abordagem Bottom-Up)



```
countPaths(n, m)
  grid = [array (n*m) inicializado com 0]
  for i=0 to n - 1
    grid[i][0] = 1

  for j=0 to m - 1
    grid[0][j] = 1

  for i=1 to n - 1
    for j=1 to m - 1
      grid[i][j] = grid[i - 1][j] + grid[i][j - 1]
```

34

Otimizando código com tabulação (Abordagem Bottom-Up)



	1	1	1	1
1				
1				

N = 3, M = 5

```
grid = [array (nxm) inicializado com 0]
for i=0 to n - 1
  grid[i][0] = 1
for j=0 to m - 1
  grid[0][j] = 1
```

35

Otimizando código com tabulação (Abordagem Bottom-Up)



	1	1	1	1
1	2			
1				

N = 3, M = 5

```
for i=1 to n - 1
  for j=1 to m - 1
    grid[i][j] = grid[i - 1][j] + grid[i][j - 1]
```

36



Otimizando código com tabulação (Abordagem Bottom-Up)

	1	1	1	1
1	2	3		
1				

$N = 3, M = 5$

37



Otimizando código com tabulação (Abordagem Bottom-Up)

	1	1	1	1
1	2	3	4	
1				

$N = 3, M = 5$

38



Otimizando código com tabulação (Abordagem Bottom-Up)

	1	1	1	1
1	2	3	4	5
1				

$N = 3, M = 5$

39



Otimizando código com tabulação (Abordagem Bottom-Up)

	1	1	1	1
1	2	3	4	5
1	3			

$N = 3, M = 5$

40



Otimizando código com tabulação (Abordagem Bottom-Up)

	1	1	1	1
1	2	3	4	5
1	3	6		

$N = 3, M = 5$

41

Otimizando código com tabulação (Abordagem Bottom-Up)

	1	1	1	1
1	2	3	4	5
1	3	6	10	

$N = 3, M = 5$

42

Otimizando código com tabulação (Abordagem Bottom-Up)

	1	1	1	1
1	2	3	4	5
1	3	6	10	15

$N = 3, M = 5$

$\text{paths}(N, M) = 15$

43

Otimizando código com tabulação (Abordagem Bottom-Up)

	1	1	1	1
1	2	3	4	5
1	3	6	10	15

$N = 3, M = 5$

Passamos por cada célula uma vez, logo:

Complexidade: $O(N * M)$

44

Conclusões

- Nem sempre precisamos começar da abordagem top-down para chegar à bottom-up, pode ser que forma iterativa seja mais intuitiva em alguns casos
- Pode ser que precisemos de mais de uma variável para definir um estado
 - DP em duas dimensões (2D), três dimensões (3D), etc...
 - Escolher o menor número de variáveis possível para representar estado
 - Nesse caso, é intuitivo usar as coordenadas (x, y) como estado, mas em outros problemas temos de ser mais criativos

45

Problemas Clássicos PD - Soma Contígua Máxima

<https://devsuperior.com.br>

Prof. Dr. Nelio Alves

46

Soma Contígua Máxima de um Array

- Dado um array de inteiros não-nulos, ache o maior valor possível de ser obter com uma soma contígua de elementos de s.
 - Soma contígua: soma de todos os elementos entre os índices i e j

0	1	2	3	4	5	6	7	8
5	-10	2	3	6	-5	7	-20	10

Exemplo: soma contígua de 1 até 7

- $s(1, 7) = -10 + 2 + 3 + 6 + (-5) + 7 + (-20) = -17$

47

Soma Contígua Máxima de um Array

- Nesse array, qual a maior soma contígua?

0	1	2	3	4	5	6	7	8
5	-10	2	3	6	-5	7	-20	10

Resposta:

0	1	2	3	4	5	6	7	8
5	-10	2	3	6	-5	7	-20	10

$$s(2, 6) = 2 + 3 + 6 + (-5) + 7 = 13$$

48

Formulando um algoritmo

Abordagem 1: Força Bruta

- Testar todos os possíveis subarrays e verificar suas somas
- Para o índice i , calcular a soma de todos os subarrays que começam em $V[i]$

0	1	2	3	4	5	6	7	8
5	-10	2	3	6	-5	7	-20	10

49

Formulando um algoritmo - Força Bruta

- Gerando todos os subarrays que começam em $V[0]$

5	-10	2	3	6	-5	7	-20	10
---	-----	---	---	---	----	---	-----	----

5

5	-10
---	-----

...

5	-10	2	3	6	-5	7	-20
---	-----	---	---	---	----	---	-----

5	-10	2	3	6	-5	7	-20	10
---	-----	---	---	---	----	---	-----	----

- Gerando todos os subarrays que começam em $V[1]$

5	-10	2	3	6	-5	7	-20	10
---	-----	---	---	---	----	---	-----	----

-10

-10	2
-----	---

...

-10	2	3	6	-5	7	-20
-----	---	---	---	----	---	-----

-10	2	3	6	-5	7	-20	10
-----	---	---	---	----	---	-----	----

Para cada um dos subarrays gerados, calcular a soma e manter melhor solução.

50

Formulando um algoritmo - Força Bruta

Problema com a solução: Ineficiente!

- Passamos muitas vezes pelas mesmas posições

```
for(i = 0; i < n; i++)  
  for(j = i; j < n; j++)  
    for(k = i; k < j; k++)
```

← 1. Escolher índice inicial - $O(n)$
← 2. Gerar subarrays iniciando em i - $O(n)$
← 3. Calcular soma do subarray (i, j) - $O(n)$

Complexidade: $O(n^3)$

Pergunta: Será que precisamos realmente percorrer de novo os subarrays para obter a soma? Como fazer passo 3 melhor?

51

Formulando um algoritmo

Abordagem 2: Força Bruta com Janela Deslizante (Sliding Window)

- Testar todos os possíveis subarrays, mantendo a soma de forma inteligente
 - A cada novo começo, manter a soma acumulada

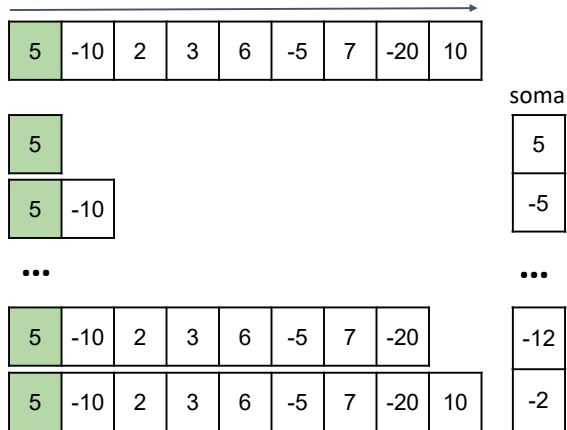
0	1	2	3	4	5	6	7	8
5	-10	2	3	6	-5	7	-20	10

Obs.: essa otimização pode ser feita também com a técnica de soma de prefixos

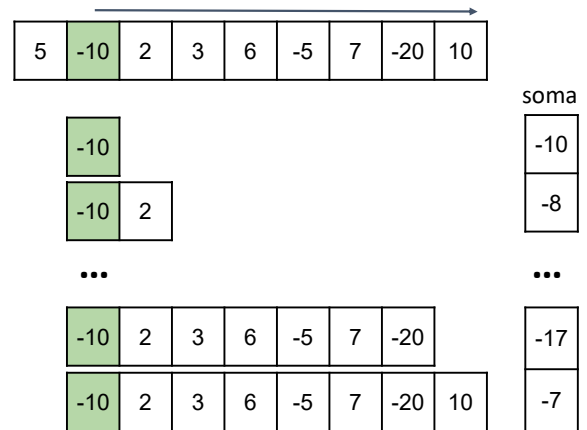
52

Formulando um algoritmo - Força Bruta com Janela Deslizante

- Gerando todos os subarrays que começam em $V[0]$



- Gerando todos os subarrays que começam em $V[1]$



Acumular soma a cada elemento novo adicionado ao subarray

53

Formulando um algoritmo - Força Bruta com Janela Deslizante

- Melhor que a solução força bruta simples! Agora obtemos a soma de maneira eficiente.

```
maxSum = 0;
for(i = 0; i < n; i++){
    currentSum = 0;
    for(j = i; j < n; j++){
        currentSum += v[j];
        maxSum = max(maxSum, currentSum)
    }
}
```

← 1. Escolher índice inicial - $O(n)$

← 2. Gerar subarrays iniciando em i - $O(n)$
 - Mantemos a soma acumulada e testamos soma máxima - $O(1)$

Complexidade: $O(n^2)$

54

Formulando um algoritmo

Abordagem 3: Algoritmo de Kadane

- Será que é possível achar a resposta do problema em apenas uma passada?
- Vamos voltar à abordagem força bruta, mas começaremos de trás para frente
- Para o i , começando em $n - 1$, calcular a soma de todos os subarrays que terminam em $V[i]$

0	1	2	3	4	5	6	7	8
5	-10	2	3	6	-5	7	-20	10

55

Formulando um algoritmo - Algoritmo de Kadane

- Gerando todos os subarrays que terminam em $V[8]$

5	-10	2	3	6	-5	7	-20	10	
								10	
							-20	10	
...									
		-10	2	3	6	-5	7	-20	10
5	-10	2	3	6	-5	7	-20	10	

- Gerando todos os subarrays que terminam em $V[7]$

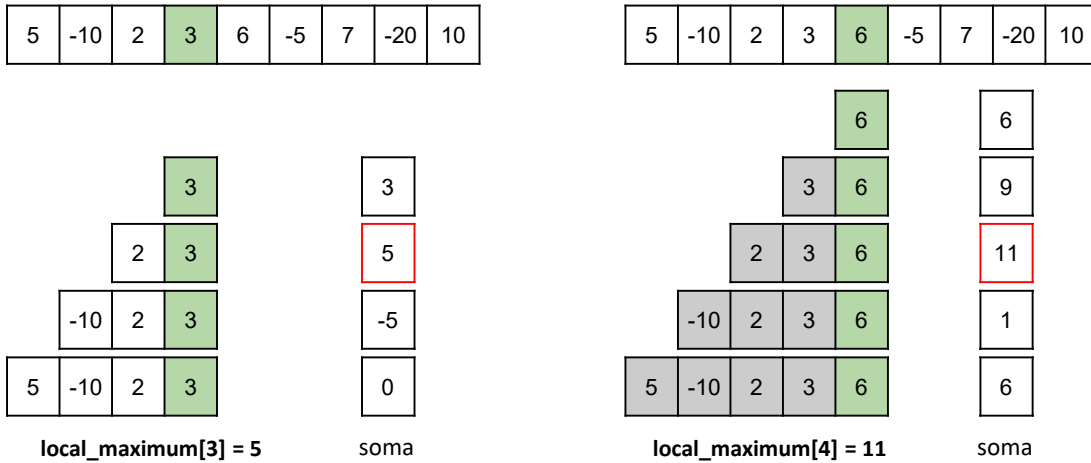
5	-10	2	3	6	-5	7	-20	10
							-20	
						7	-20	
								...
		-10	2	3	6	-5	7	-20
5	-10	2	3	6	-5	7	-20	

Acumular soma a cada elemento novo adicionado ao subarray

56

Formulando um algoritmo - Algoritmo de Kadane

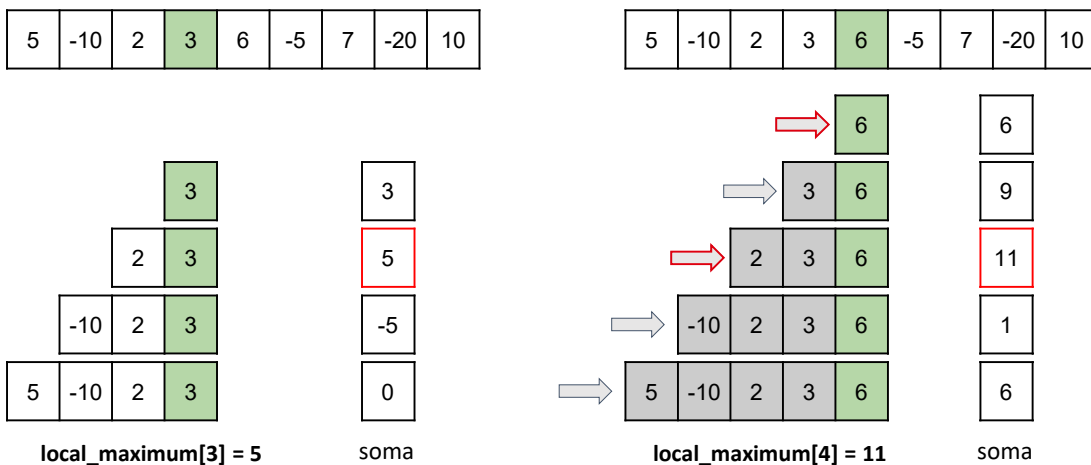
- Se $\text{local_maximum}[i] = \text{soma}$ do maior array que termina no índice i



57

Formulando um algoritmo - Algoritmo de Kadane

- Se $\text{local_maximum}[i] = \text{soma}$ do maior array que termina no índice i

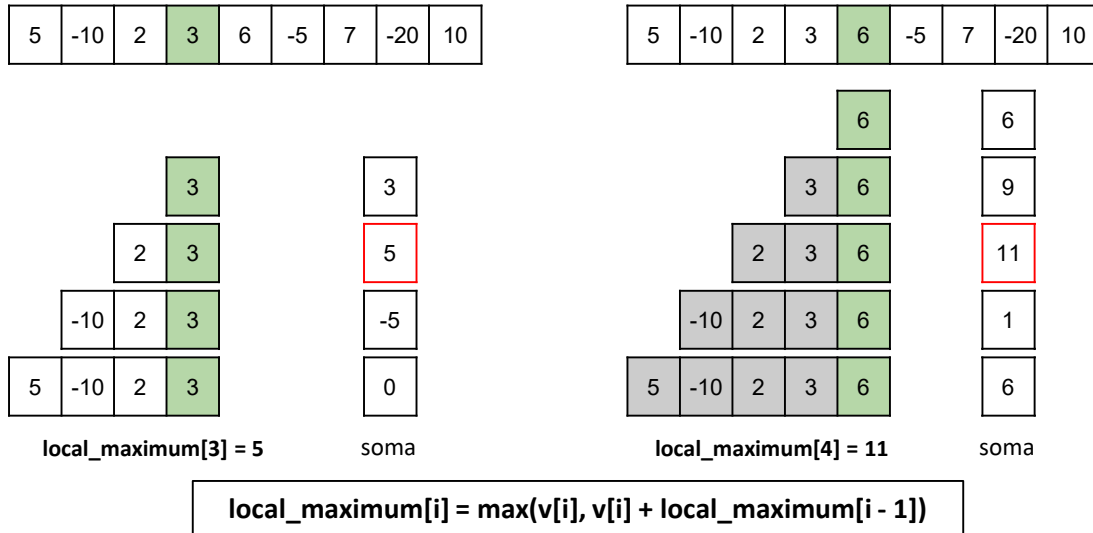


Tudo o que está em cinza já foi analisado anteriormente, e concluímos que o maior array é o de soma 5. Logo, podemos continuar esse subarray, ou começar um novo.

58

Formulando um algoritmo - Algoritmo de Kadane

- Se $\text{local_maximum}[i]$ = soma do maior array que termina no índice i



59

Algoritmo de Kadane

- O problema se resume em, a cada índice i , achar o máximo entre dois números
 - $v[i]$ e $(v[i] + \text{local_maximum}[i - 1])$
- Utilizamos o subproblema $\text{local_maximum}[i - 1]$ para determinar $\text{local_maximum}[i]$

```
local_maximum = []
local_maximum[0] = v[0];
maxSum = v[0];

for(i = 0; i < n; i++){
    local_maximum[i] = max(v[i], v[i] + local_maximum[i-1])
    if(local_maximum[i] > maxSum){
        maxSum = local_maximum[i];
    }
}
return maxSum
```

Complexidade Temporal: $O(n)$

Complexidade Espacial: $O(n)$

60

Algoritmo de Kadane

- Como o subproblema atual só depende do anterior, não precisamos do vetor memória o tempo inteiro

```
maxSum = v[0];  
for(i = 0; i < n; i++){  
    currentSum = max(v[i], v[i] + currentSum)  
    if(currentSum > maxSum){  
        maxSum = currentSum;  
    }  
}
```

Complexidade Temporal: $O(n)$

Complexidade Espacial: $O(1)$

61

Problemas Clássicos PD - Problema da Mochila 0/1

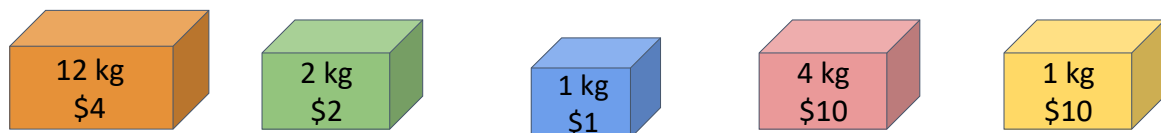
<https://devsuperior.com.br>

Prof. Dr. Nelio Alves

62

O Dilema do Caçador de Tesouros

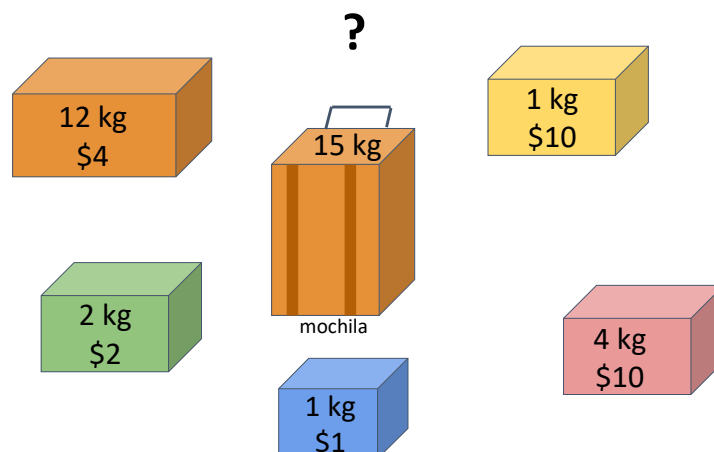
- Um caçador de tesouros chega à sala do tesouro e se depara com um dilema. Diante dele, há N itens, cada um seu próprio peso e valor.



63

O Dilema do Caçador de Tesouros

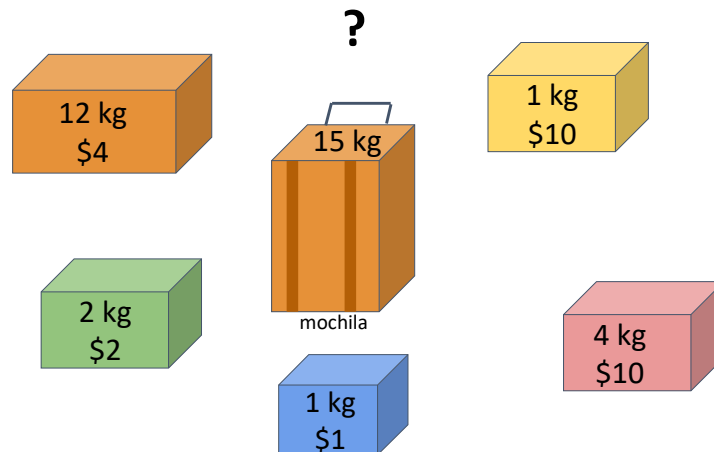
- No entanto, ele se depara com uma limitação: sua mochila não pode carregar todos os itens, pois tem um **peso máximo** que pode suportar. Então, ele precisa decidir cuidadosamente que itens levar para **maximizar o valor total** que levará consigo. Que itens ele deve escolher?



64

O Problema da Mochila 0/1

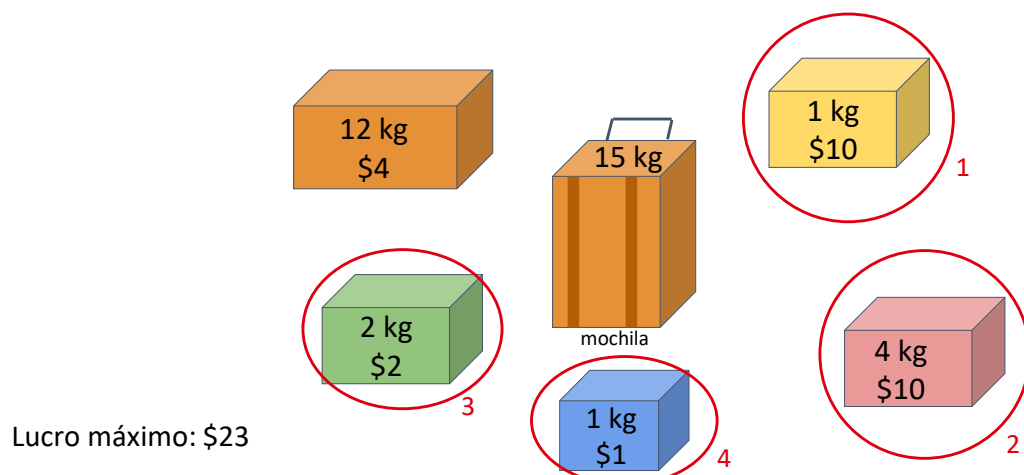
- Essa situação é conhecida como o **problema da mochila**. O objetivo é selecionar itens da melhor maneira (lucro máximo), levando em consideração as restrições de peso.



65

O Problema da Mochila 0/1

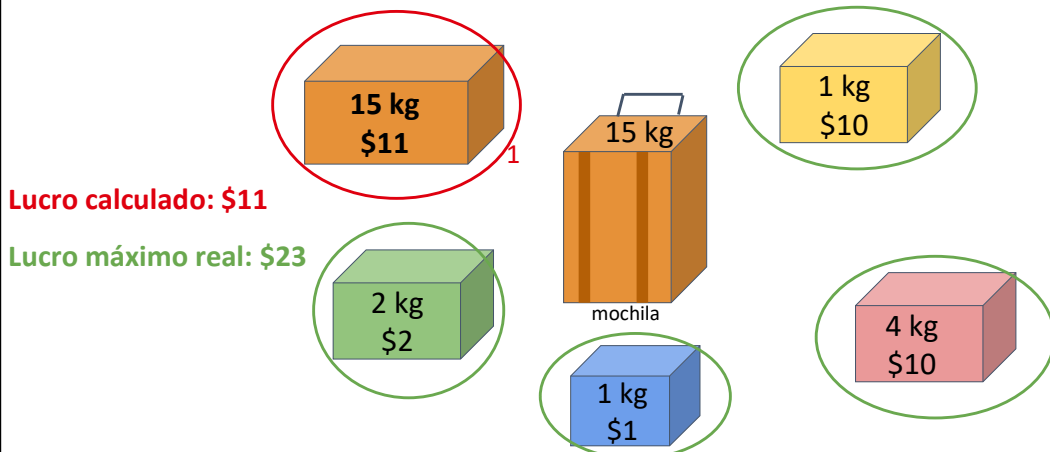
- Abordagem gulosa:** se escolhermos o **item mais valioso** até encher a mochila, conseguimos solução ótima?
 - Para esse caso específico funciona, mas será que vale para todos?



66

O Problema da Mochila 0/1

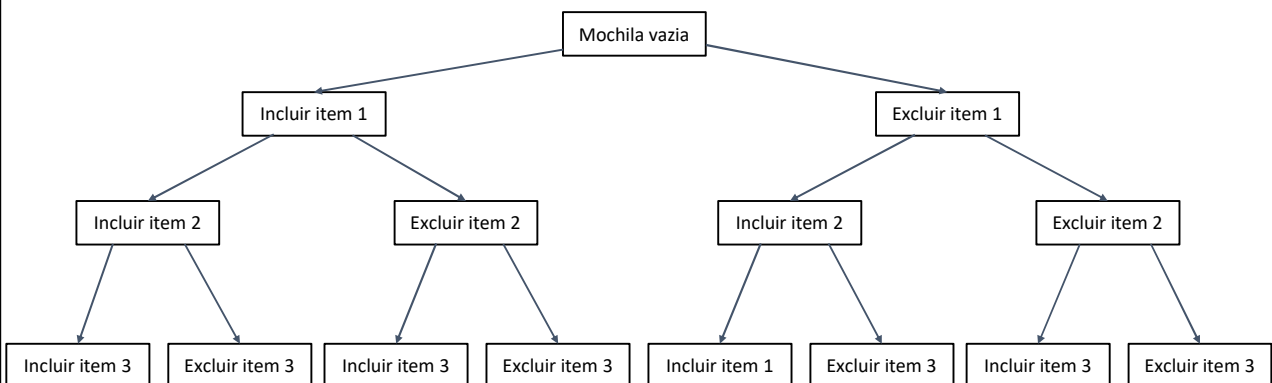
- **Abordagem gulosa:** se escolhermos o **item mais valioso** até encher a mochila, conseguimos solução ótima?
 - Para esse caso específico funciona, mas será que vale para todos? **Não!**
 - Também não funciona escolhendo item mais leve primeiro...



67

O Problema da Mochila 0/1

- **Abordagem Força Bruta:** para garantir a corretude, podemos **testar todas as combinações** possíveis. Isto é, **para cada item** da mochila, **podemos o escolher ou não**. Com 3 itens, por exemplo, isso resulta na seguinte árvore de decisões.



Em cada nível d, existem 2^d escolhas. Se existem N itens, a **complexidade de tempo é $O(2^n)$**

68

O Problema da Mochila 0/1

Definição de estados

- Apesar de lenta, a abordagem Força Bruta nos ajudará a **definir estados** para o problema.
 - Nem sempre será possível incluir um item → Limite de peso da mochila
 - Assim um estado pode ser definido pela escolha de um **item i** e o **peso restante**

$\text{knapsack}(i, w)$ = valor máximo considerando até o item i, com capacidade atual w

69

O Problema da Mochila 0/1

Exemplo: pesos = [1, 2, 3]
valores = [6, 10, 12]
w = 5

Estado inicial, ainda não
consideramos nenhum item,
capacidade total.

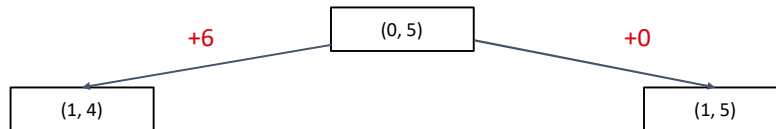
(0, 5)

70

O Problema da Mochila 0/1

Exemplo: pesos = [1, 2, 3]
valores = [6, 10, 12]
w = 5

Consideramos o item 1

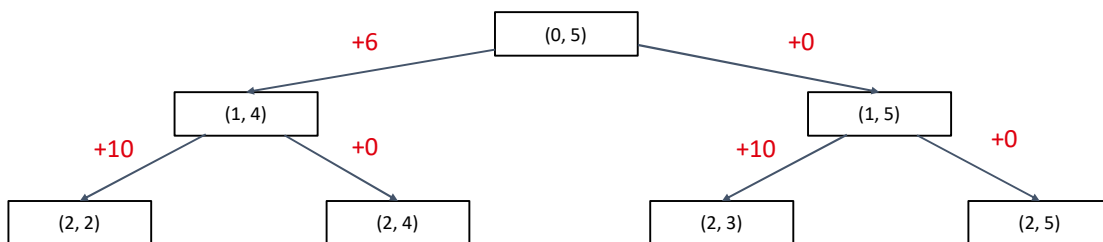


71

O Problema da Mochila 0/1

Exemplo: pesos = [1, 2, 3]
valores = [6, 10, 12]
w = 5

Consideramos o item 2

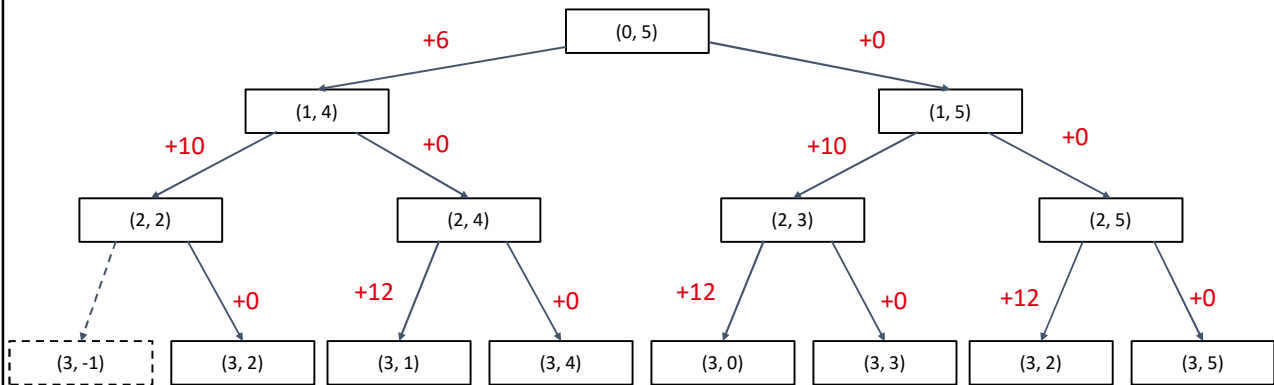


72

O Problema da Mochila 0/1

Exemplo: pesos = [1, 2, 3]
valores = [6, 10, 12]
w = 5

Consideramos o item 3

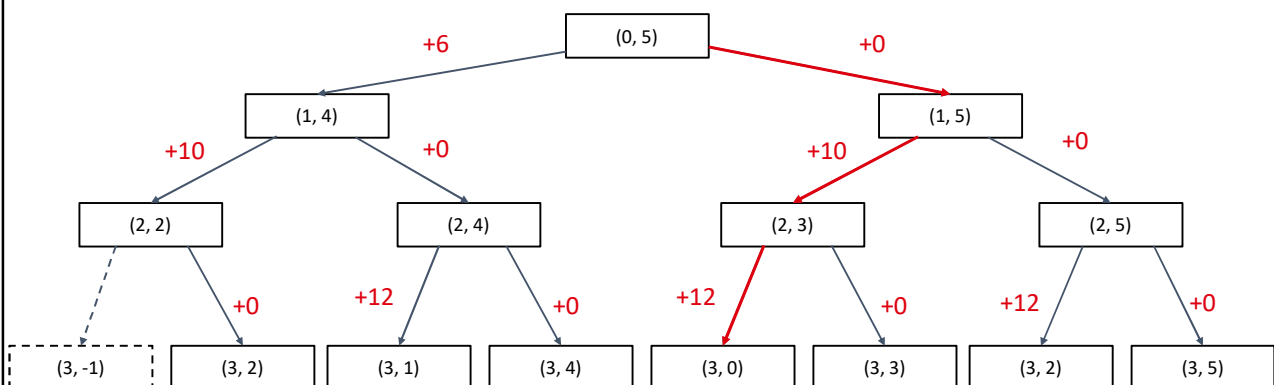


73

O Problema da Mochila 0/1

Exemplo: pesos = [1, 2, 3]
valores = [6, 10, 12]
w = 5

Qual é a combinação de maior valor?



Lucro máximo: 22

74

O Problema da Mochila 0/1

- **Problema:** achar valor máximo dentre n itens, com capacidade w
- **Definição de estado**
 - $\text{knapsack}(n, w)$ = valor máximo considerando até o item n , com capacidade atual w na mochila

Subproblemas:

Incluir o item $n - 1$	Excluir o item $n - 1$
$\text{knapsack}(n - 1, w - \text{pesos}[n - 1]) + \text{valor}[n - 1]$	$\text{knapsack}(n - 1, w)$
Diminui a capacidade atual, aumenta o valor	Mantemos a capacidade atual e o valor

75

O Problema da Mochila 0/1

Se queremos maximizar o lucro, a resposta será a escolha que o maximiza, logo:

- **Relação recursiva**

$$\text{knapsack}(n, w) = \max(\text{knapsack}(n - 1, w - \text{pesos}[n - 1]) + \text{valor}[n - 1], \text{knapsack}(n - 1, w))$$

Incluir o item $n - 1$

Excluir o item $n - 1$

76

O Problema da Mochila 0/1

Se queremos maximizar o lucro, a resposta será a escolha que o maximiza, logo:

- **Casos base**

- Se a capacidade é 0, não podemos incluir mais nada, logo o lucro é 0
- Se não temos mais itens para considerar, o lucro é 0
- Se $n == 0$ ou $w == 0$, retorne 0

- **Relação recursiva**

```
knapsack(n, w) = max(knapsack(n - 1, w - pesos[n - 1]) + valor[n - 1], knapsack(n - 1, w))
```

77

O Problema da Mochila 0/1

```
knapsack(n, w)
    if n == 0 || w == 0
        return 0

    // Se próximo item excede capacidade, pula ele
    if weights[n - 1] > w
        return knapsack(n - 1, w)

    else
        return max(knapsack(n - 1, w - weights[n - 1]) + values[n - 1], knapsack(n - 1, w))
```

78

O Problema da Mochila 0/1

Problemas com a implementação atual: Recálculo de subproblemas repetidos. Ainda estamos fazendo solução força bruta!

Como otimizar? Memoização.

- Iremos utilizar uma matriz NxW para armazenar as soluções dos subproblemas definido por n (número de itens) e w (peso).

79

O Problema da Mochila 0/1 - Abordagem DP (Top-Down)

```
memo = [array (n*w)]
knapsack(n, w)
    if memo[n][w] not null
        return memo[n][w]

    if n == 0 || w == 0
        return 0

    // Se próximo item excede capacidade, pula ele
    if weights[n - 1] > w
        return knapsack(n - 1, w)

    else
        memo[n][w] = max(knapsack(n - 1, w - weights[n - 1]) + values[n - 1], knapsack(n - 1, w))
        return memo[n][w]
```

Complexidade: $O(N * W)$

80

O Problema da Mochila 0/1 - Abordagem Bottom-Up

A partir da abordagem top-down, obtemos essa relação, que podemos usar para calcular cada item iterativamente.

```
memo[i][w] = max(memo[i - 1][w], values[i - 1] + memo[i - 1][w - weights[i - 1]])
```

81

O Problema da Mochila 0/1 - Abordagem Bottom-Up

		capacidade					
		0	1	2	3	4	5
vazio	0						
(v ₁ =6, w ₁ =1)	1						
(v ₂ =10, w ₂ =2)	2						
(v ₃ =12, w ₃ =3)	3						

82

O Problema da Mochila 0/1 - Abordagem Bottom-Up

		capacidade					
		0	1	2	3	4	5
vazio	0	0	0	0	0	0	0
$(v_1=6, w_1=1)$	1						
$(v_2=10, w_2=2)$	2						
$(v_3=12, w_3=3)$	3						

83

O Problema da Mochila 0/1 - Abordagem Bottom-Up

		capacidade					
		0	1	2	3	4	5
vazio	0	0	0	0	0	0	0
$(v_1=6, w_1=1)$	1	0					
$(v_2=10, w_2=2)$	2						
$(v_3=12, w_3=3)$	3						

`memo[i][w] = max(memo[i - 1][w], values[i - 1] + memo[i - 1][w - weights[i - 1]])`

84

O Problema da Mochila 0/1 - Abordagem Bottom-Up

		capacidade					
		0	1	2	3	4	5
vazio	0	0	0	0	0	0	0
$(v_1=6, w_1=1)$	1	0	6				
$(v_2=10, w_2=2)$	2						
$(v_3=12, w_3=3)$	3						

$memo[i][w] = \max(memo[i-1][w], values[i-1] + memo[i-1][w - weights[i-1]])$

85

O Problema da Mochila 0/1 - Abordagem Bottom-Up

		capacidade					
		0	1	2	3	4	5
vazio	0	0	0	0	0	0	0
$(v_1=6, w_1=1)$	1	0	6	6			
$(v_2=10, w_2=2)$	2						
$(v_3=12, w_3=3)$	3						

86

O Problema da Mochila 0/1 - Abordagem Bottom-Up

		capacidade					
		0	1	2	3	4	5
vazio	0	0	0	0	0	0	0
$(v_1=6, w_1=1)$	1	0	6	6	6		
$(v_2=10, w_2=2)$	2						
$(v_3=12, w_3=3)$	3						

87

O Problema da Mochila 0/1 - Abordagem Bottom-Up

		capacidade					
		0	1	2	3	4	5
vazio	0	0	0	0	0	0	0
$(v_1=6, w_1=1)$	1	0	6	6	6	6	
$(v_2=10, w_2=2)$	2						
$(v_3=12, w_3=3)$	3						

88

O Problema da Mochila 0/1 - Abordagem Bottom-Up

		capacidade					
		0	1	2	3	4	5
vazio	0	0	0	0	0	0	0
$(v_1=6, w_1=1)$	1	0	6	6	6	6	6
$(v_2=10, w_2=2)$	2						
$(v_3=12, w_3=3)$	3						

89

O Problema da Mochila 0/1 - Abordagem Bottom-Up

		capacidade					
		0	1	2	3	4	5
vazio	0	0	0	0	0	0	0
$(v_1=6, w_1=1)$	1	0	6	6	6	6	6
$(v_2=10, w_2=2)$	2	0					
$(v_3=12, w_3=3)$	3						

90

O Problema da Mochila 0/1 - Abordagem Bottom-Up

		capacidade					
		0	1	2	3	4	5
vazio	0	0	0	0	0	0	0
$(v_1=6, w_1=1)$	1	0	6	6	6	6	6
$(v_2=10, w_2=2)$	2	0	6				
$(v_3=12, w_3=3)$	3						

91

O Problema da Mochila 0/1 - Abordagem Bottom-Up

		capacidade					
		0	1	2	3	4	5
vazio	0	0	0	0	0	0	0
$(v_1=6, w_1=1)$	1	0	6	6	6	6	6
$(v_2=10, w_2=2)$	2	0	6	10			
$(v_3=12, w_3=3)$	3						

92

O Problema da Mochila 0/1 - Abordagem Bottom-Up

		capacidade					
		0	1	2	3	4	5
vazio	0	0	0	0	0	0	0
$(v_1=6, w_1=1)$	1	0	6	6	6	6	6
$(v_2=10, w_2=2)$	2	0	6	10	16		
$(v_3=12, w_3=3)$	3						

93

O Problema da Mochila 0/1 - Abordagem Bottom-Up

		capacidade					
		0	1	2	3	4	5
vazio	0	0	0	0	0	0	0
$(v_1=6, w_1=1)$	1	0	6	6	6	6	6
$(v_2=10, w_2=2)$	2	0	6	10	16	16	
$(v_3=12, w_3=3)$	3						

94

O Problema da Mochila 0/1 - Abordagem Bottom-Up

		capacidade					
		0	1	2	3	4	5
vazio	0	0	0	0	0	0	0
$(v_1=6, w_1=1)$	1	0	6	6	6	6	6
$(v_2=10, w_2=2)$	2	0	6	10	16	16	16
$(v_3=12, w_3=3)$	3						

95

O Problema da Mochila 0/1 - Abordagem Bottom-Up

		capacidade					
		0	1	2	3	4	5
vazio	0	0	0	0	0	0	0
$(v_1=6, w_1=1)$	1	0	6	6	6	6	6
$(v_2=10, w_2=2)$	2	0	6	10	16	16	16
$(v_3=12, w_3=3)$	3	0					

96

O Problema da Mochila 0/1 - Abordagem Bottom-Up

		capacidade					
		0	1	2	3	4	5
vazio	0	0	0	0	0	0	0
$(v_1=6, w_1=1)$	1	0	6	6	6	6	6
$(v_2=10, w_2=2)$	2	0	6	10	16	16	16
$(v_3=12, w_3=3)$	3	0	6				

97

O Problema da Mochila 0/1 - Abordagem Bottom-Up

		capacidade					
		0	1	2	3	4	5
vazio	0	0	0	0	0	0	0
$(v_1=6, w_1=1)$	1	0	6	6	6	6	6
$(v_2=10, w_2=2)$	2	0	6	10	16	16	16
$(v_3=12, w_3=3)$	3	0	6	10			

98

O Problema da Mochila 0/1 - Abordagem Bottom-Up

		capacidade					
		0	1	2	3	4	5
vazio	0	0	0	0	0	0	0
$(v_1=6, w_1=1)$	1	0	6	6	6	6	6
$(v_2=10, w_2=2)$	2	0	6	10	16	16	16
$(v_3=12, w_3=3)$	3	0	6	10	16		

99

O Problema da Mochila 0/1 - Abordagem Bottom-Up

		capacidade					
		0	1	2	3	4	5
vazio	0	0	0	0	0	0	0
$(v_1=6, w_1=1)$	1	0	6	6	6	6	6
$(v_2=10, w_2=2)$	2	0	6	10	16	16	16
$(v_3=12, w_3=3)$	3	0	6	10	16	18	

100

O Problema da Mochila 0/1 - Abordagem Bottom-Up

		capacidade					
		0	1	2	3	4	5
vazio	0	0	0	0	0	0	0
$(v_1=6, w_1=1)$	1	0	6	6	6	6	6
$(v_2=10, w_2=2)$	2	0	6	10	16	16	16
$(v_3=12, w_3=3)$	3	0	6	10	16	18	22

101

O Problema da Mochila 0/1 - Abordagem Bottom-Up

		capacidade					
		0	1	2	3	4	5
vazio	0	0	0	0	0	0	0
$(v_1=6, w_1=1)$	1	0	6	6	6	6	6
$(v_2=10, w_2=2)$	2	0	6	10	16	16	16
$(v_3=12, w_3=3)$	3	0	6	10	16	18	22

Lucro máximo: 22

102

O Problema da Mochila 0/1 - Abordagem DP (Bottom-Up)

```
knapsack(n, W)
    memo = [array (n+1)][array (W+1)]

    for i=1 to n
        for w=1 to W
            if weights[i - 1] <= w
                memo[i][w] = max(memo[i - 1][w],
                                values[i - 1] + memo[i - 1][w - weights[i - 1]])

            else memo[i][w] = memo[i - 1][w]

    return memo[n][w]
```

Complexidade: $O(N * W)$