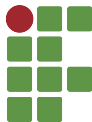
Programação Dinâmica – Tópicos Especiais em Algoritmos

Tópicos Especiais em Algoritmos - Ciência da Computação



INSTITUTO FEDERAL
Brasília

Prof. Daniel Saad Nogueira Nunes

IFB – Instituto Federal de Brasília, Campus Taguatinga

Sumário

1 Introdução

2 Programação Dinâmica

3 Considerações Finais

Sumário

1 Introdução

Introdução

- A programação dinâmica consiste em armazenar a solução de subproblemas para que elas possam ajudar a computar mais rapidamente a solução dos problemas maiores.
- Esta estratégia é extremamente efetiva quando há uma grande sobreposição de subproblemas.
- Em outras palavras, quando um mesmo estado é revisitado com frequência, não precisamos gastar tempo computando a resposta, pois ela já está armazenada.
- Troca tempo por espaço.

Programação Dinâmica

UVa 11450

- Vamos considerar o problema UVa 11450 para discorrer sobre o conceito de programação dinâmica.
- Neste problema existe um montante inicial, vários tipos de peças de roupa diferentes e, para cada tipo, um conjunto de peças daquele tipo, cada uma com um determinado preço.
- O objetivo é comprar uma peça de cada tipo de modo que o custo total seja o maior possível, mas sem ultrapassar o valor do montante inicial.

Programação Dinâmica

UVa 11450: Entrada

- $1 \leq m \leq 200$: montante inicial disponível.
- $1 \leq c \leq 20$: tipos diferentes de peças.
- $Q = (q_0, \ldots, q_{c-1})$: a quantidade de peças diferentes para cada tipo, com $1 \leq q_i \leq 20$.
- Para cada tipo de peça de roupa i, temos uma sequência de valores $V_i = (v_{i_0}, \ldots, v_{i_{q_i - 1}})$.

UVa 11450: Saída

- O valor máximo a ser gasto ao comprar uma peça de cada tipo sem exceder o valor m ou a mensagem `no solution` caso não haja solução.

Programação Dinâmica

UVa 11450: exemplo

Para entender melhor o problema, suponha que o montante inicial é $m = 20$ existam $c = 3$ tipos de peças de roupa: camisas, gravatas e sapatos.

- Existem três camisas de custo $V_0 = (6, 4, 8)$.
- Duas gravatas de custo $V_1 = (5, 10)$.
- Três sapatos de custo $V_2 = (1, 5, 3)$.

Uma solução é comprar uma camisa de custo 8, uma gravata de custo 10 e um sapato de custo 1, totalizando 19.
Outra solução é obter uma camisa de custo 6, uma gravata de custo 10 e um sapato de custo 3.

Programação Dinâmica

UVa 11450: exemplo

- Uma solução é comprar uma camisa de custo 8, uma gravata de custo 10 e um sapato de custo 1, totalizando 19.
- Outra solução é obter uma camisa de custo 6, uma gravata de custo 10 e um sapato de custo 3.
- Não existe solução com custo total de 20.

Programação Dinâmica

UVa 11450: exemplo

Considerando os mesmos valores de itens, mas com um montante inicial $m = 9$, mesmo comprando os itens mais baratos de cada tipo, o montante inicial seria ultrapassado.

- `no solution`.

UVa 11450

- Discutiremos agora uma série de abordagens para tratar esse problema.

UVa 11450: abordagens

Abordagem 1: algoritmo guloso (WA)

- Uma possível abordagem gulosa é sempre pegar o item mais caro de cada tipo.
- Utilizando o exemplo anterior com $m = 12$, o algoritmo guloso selecionaria:
  - Camisa de custo 8.
- Com um montante de $m' = 4$ não conseguimos mais comprar nenhuma peça.
- A solução ótima envolve uma camisa de custo 4, uma gravata de custo 5 e um sapato de custo 3 ($4 + 5 + 3 = 12$).

UVa 11450: abordagens

Abordagem 2: divisão e conquista (WA)

- O paradigma de divisão e conquista não se aplica a este problema.
- Os subproblemas nem sempre são independentes um dos outros.
- O problema tem propriedade de subestrutura ótima.

UVa 11450: abordagens

Abordagem 3: busca completa (TLE)

- O paradigma de busca completa sempre fornece a solução ótima. Mas gasta muito tempo.
- Como $c \leq 20$ e cada $q_i \leq 20$, temos, no pior caso, 20^{20} combinações de itens.
- É demais!

UVa 11450: busca completa

Algorithm 1: $\textsc{Backtracking}(V, m, current, i)$

Input: $V, m, current, i$

Output: valor máximo comprando um item de cada tipo

1 **if**($current < 0$)
2   **return** $-\infty$
3 **if**($i = c \wedge current \geq 0$)
4   **return** $m - current$
5 $ans \leftarrow -\infty$
6 **for**($j \leftarrow 0; j < V_i.\textsc{size}(); j++$)
7     $ans \leftarrow \max(ans, \textsc{Backtracking}(current - V_i[j], i + 1))$
8 **return** ans

- Chamada inicial: $\textsc{Backtracking}(V, m, m, 0)$.

UVa 11450

Podemos perceber duas coisas:

1. O problema tem subestrutura ótima. Se a j-ésima peça do i-ésimo tipo está na solução, então a solução do subproblema sem a peça escolhida tem que ser ótima.
2. Existem sobreposições entre subproblemas. O espaço de busca de tamanho 20^{20} tem muitas sobreposições.

UVa 11450

- Os dois itens são chaves para proposição de soluções baseadas em programação dinâmica.
- Sempre que um problema for resolvido, sua solução é armazenada.
- Se ele ocorrer novamente, já saberemos a resposta.

UVa 11450

Abordagem 4: Programação Dinâmica Top-down (AC)

- Podemos utilizar a nossa solução de busca completa e adicionar uma tabela para lembrar os estados (subproblemas) que já foram computados.
- Esta técnica é conhecida como *memoization*.

UVa 11450: Programação Dinâmica Top-down

Algorithm 2: DP-TOPDOWN$(V, DP, m, current, i)$

Input: $V[0, k-1], DP[0, c][0, m], m, current, i$

Output: valor máximo comprando um item de cada tipo

```
1  if( current < 0 )
2      return −∞
3  if( i = c ∧ current ≥ 0 )
4      return m − current
5  if( D[i][current] ≠ ⊥ )  return D[i][current]
6  ans ← −∞
7  for( j ← 0; j < V_i.SIZE(); j++ )
8      ans ← max(ans, DP-TOPDOWN(V, DP, m, current − V_i[j], i + 1))
9  DP[i][current] ← ans
10 return ans
```

- Chamada inicial: DP-TOPDOWN$(V, DP, m, m, 0)$.

UVa 11450: Programação Dinâmica Bottom-up

- Onde a resposta está?

UVa 11450: Programação Dinâmica Bottom-up

- Onde a resposta está?
- Na célula $DP[0][m]$.

UVa 11450: Programação Dinâmica Top-down

Complexidade de Pior-caso

- Quando a solução já foi computada ou caímos no caso base, a chamada recursiva gasta $\Theta(1)$.
- No caso geral, gastamos tempo $O(k)$, em que k corresponde a maior quantidade de peças de um determinado tipo.
- Só precisamos saber o número máximo de estados, que é mc
- Custo total: $O(mck)$.

Programação Dinâmica Bottom-up

- Uma solução baseada em programação dinâmica não necessariamente precisa ser recursiva.
- Abordagem bottom-up: computa as soluções dos problemas menores para utilizá-las na resolução dos problemas maiores.
- Abordagem iterativa.

UVa 11450

Abordagem 5: Programação Dinâmica Bottom-up (AC)

- Podemos utilizar uma solução Bottom-up iterativa para computar a tabela de programação dinâmica.

UVa 11450: Programação Dinâmica Bottom-up

- Seja $T(i, j)$ um valor *booleano* que indica se conseguimos, utilizando as i primeiras peças ficar com um montante restante de valor j.
- Claramente, $T(0, j) =$ **true** se $j = m$ e **false** se $0 \leq j < m$.
- Como computar $T(i, j)$ sabendo que as soluções de $T(a, b)$, $0 \leq a < i$ e $0 \leq b \leq m$ já foram computadas?

UVa 11450: Programação Dinâmica Bottom-up

- Para cada um dos custos v considerando o i-ésimo tipo de peça, basta verificar se $T(i-1, j+v)$ é verdadeiro.
- Em caso afirmativo, é possível incluir a peça de valor v de tipo i na solução e obter um montante igual a j.

UVa 11450: Programação Dinâmica Bottom-up

$$T(i,j) = \begin{cases} \textbf{true}, & i = 0 \land j = m \\ \textbf{false}, & i = 0 \land 0 \leq j < m \\ \textbf{true} \text{ se } T(i-1, j+v) = \textbf{true}, \text{com } v \in V_{i-1} \text{ e } j + v \leq m, \\ \quad i \geq 0 \land j \leq m \\ \textbf{false} \text{ caso contrário} \end{cases}$$

UVa 11450: Programação Dinâmica Bottom-up

Algorithm 3: DP-BOTTOM-UP(V, c, m)

Input: $V[0, k-1], c, m$

Output: Valor máximo comprando um item de cada tipo

```
1 DP[0][m] ← true
2 for( j ← 0; j < m; j++ ) DP[0][j] ← false
3 for( i ← 1; i ≤ c; j++ )
4     for( j ← 0; j ≤ m; j++ )
5         for( l ← 0; l < V_{i-1}.SIZE(); l++ )
6             if( j + V_{i-1}[l] ≤ m ∧ DP[i-1][j + V_{i-1}[l]] )
7                 DP[i][j] ← True
```

UVa 11450: Programação Dinâmica Bottom-up

- Onde a resposta está?

UVa 11450: Programação Dinâmica Bottom-up

- Onde a resposta está?
- Na célula $DP[c][j] =$ **true** com menor valor de j possível.

UVa 11450: Programação Dinâmica Bottom-up

Complexidade da Solução

- $O(mck)$, em que k corresponde a maior quantidade de peças de um determinado tipo.

UVa 11450: Programação Dinâmica Bottom-up

Economizando Espaço

- Como cada linha da matriz de programação dinâmica só depende da anterior, só precisamos manter em memória 2 linhas.
- Em vez de usar uma matriz, podemos usar dois vetores de tamanho $m + 1$.

UVa 11450: Programação Dinâmica Bottom-up

Algorithm 4: DP-BOTTOM-UP(V, c, m)

Input: $V[0, k-1], c, m$

Output: Valor máximo comprando um item de cada tipo

```
1  DP[0][m] ← true
2  cur ← 1
3  prev ← 0
4  for( j ← 0; j < m; j++ ) DP[0][j] ← false
5  for( i ← 1; i ≤ c; j++ )
6      for( j ← 0; j ≤ m; j++ )
7          for( k ← 0; k ≤ m; k++ ) DP[cur][j] ← false
8          for( k ← 0; k < V_{i-1}.SIZE(); k++ )
9              if( j + V_{i-1}[k] ≤ m ∧ DP[prev][j + V_{i-1}[k]] )
10                 DP[cur][j] ← True
11     SWAP(cur, prev)
```

UVa 11450: Programação Dinâmica Bottom-up

- Onde a resposta está?

UVa 11450: Programação Dinâmica Bottom-up

- Onde a resposta está?
- Na célula $DP[prev][j] =$ **true** com menor valor de j possível.

Sumário

2 Programação Dinâmica

Programação Dinâmica

- Ilustraremos agora uma série de problemas clássicos nos quais a técnica de programação dinâmica é aplicável.

Sumário

2 Programação Dinâmica
- 2D-Range Sum
- LIS
- Problema do troco

Programação Dinâmica: 2D-Range Sum

2D-Range Sum

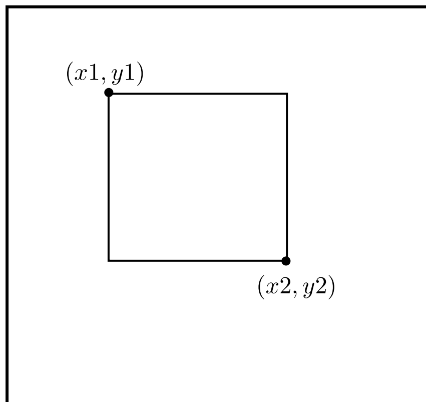
Dada uma matriz $M_{n \times m}$ de números inteiros, verificar qual a submatriz (retângulo) de M com maior soma.

2D-Range Sum: Solução Força-Bruta

Solução Força-Bruta (TLE)

- Uma submatriz pode ser vista como um retângulo definido por duas coordenadas (x_1, y_1) e (x_2, y_2) correspondendo, respectivamente, aos cantos superior esquerdo e inferior direito.
- Para cada retângulo, verifique se a soma dos elementos contidos nele é maior que a soma global. Em caso positivo, atualize a soma global.
- Complexidade: $\Theta(n^3 m^3)$.

2D-Range Sum: Solução Força-Bruta



2D-Range Sum: Solução Força-Bruta

Algorithm 5: BRUTE-FORCE-2D-RANGE-SUM(M)

Input: $M[0, n-1][0, m-1]$

Output: Maior soma dentre todas as submatrizes de M.

```
1 ans ← −∞
2 for( i ← 0; i < n; i++ )
3     for( j ← 0; j < m; j++ )
4         for( ii ← i; i < n; i++ )
5             for( jj ← j; j < m; j++ )
6                 ans ← max(ans, SUM(M, i, j, ii, jj))
7 return ans
```

2D-Range Sum: Solução Força-Bruta

Algorithm 6: $\text{SUM}(M, i, j, ii, jj)$

Input: $M[0, n-1][0, m-1], i, j, ii, jj$

Output: Maior soma dentre todas as submatrizes de M.

1 $sum \leftarrow 0$

2 **for**($k \leftarrow i; k \leq ii; k++$)

3   **for**($l \leftarrow j; l \leq jj; l++$)

4         $sum \leftarrow sum + M[k][l]$

5 **return** ans

2D-Range Sum: Programação Dinâmica

Solução em Programação Dinâmica (AC)

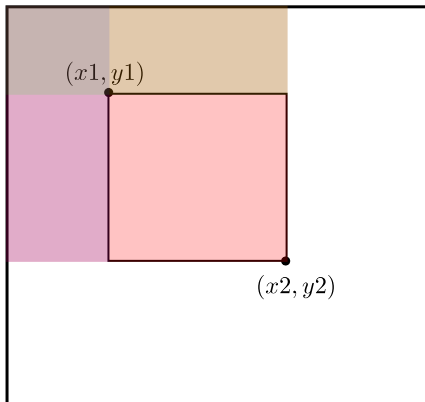
- Se conseguirmos responder em tempo constante quanto vale a soma dos elementos de uma determinada submatriz conseguimos reduzir a complexidade do algoritmo.
- $\Theta(n^2m^2)$.

2D-Range Sum: Programação Dinâmica

- Suponha que tenhamos uma matriz $S[0, n-1][0, m-1]$ que, para cada entrada $S[i][j]$ contenha a soma do retângulo com coordenadas $(0, 0)$ e (i, j).
- A partir de S, podemos computar a soma de qualquer retângulo com coordenadas (x_1, y_1) e (x_2, y_2) da seguinte maneira:

$$S[y_2][x_2] - S[y_1 - 1][x_2] - S[y_2][x_1 - 1] + S[y_1 - 1][x_1 - 1]$$

2D-Range Sum: Programação Dinâmica



2D-Range Sum: Programação Dinâmica

Algorithm 7: SUM(S, i, j, ii, jj)

Input: $S[0, n-1][0, m-1], i, j, ii, jj$

Output: Soma do retângulo em M com coordenadas (j, i) e (jj, ii)

1 $a \leftarrow S[ii][jj]$

2 $b \leftarrow S[i-1][jj]$

3 $c \leftarrow S[ii][j-1]$

4 $d \leftarrow S[i-1][j-1]$

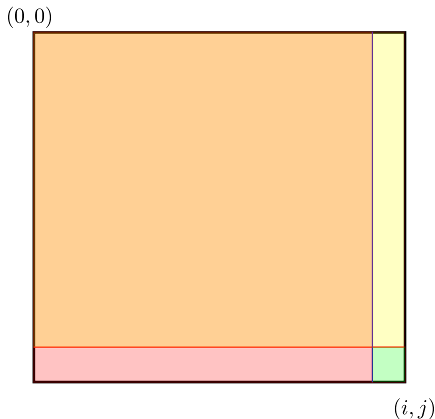
5 **return** $a - b - c + d$

2D-Range Sum

- A pergunta que fica: como computar S eficientemente?
- De uma maneira muito similar a que utilizamos para responder a soma de um retângulo em tempo constante:

$$S(i,j) = \begin{cases} M[0][0], & i = 0 \land j = 0 \\ S(i-1,0) + M[i][0], & i > 0 \land j = 0 \\ S(0,j-1) + M[0][j], & i = 0 \land j > 0 \\ S(i-1,j) + S(i,j-1) - S(i-1,j-1) + M[i][j], & c.c \end{cases}$$

2D-Range sum



2D-Range Sum

Algorithm 8: PREPROCESS(M)

Input: $M[0, n-1][0, m-1]$

Output: $S[0, n-1][0, m-1]$

1 $S[0][0] \leftarrow M[0][0]$

2 **for**($i \leftarrow 1; i < n; i++$) $S[i][0] \leftarrow S[i-1][0] + M[i][0]$

3 **for**($j \leftarrow 1; j < m; j++$) $S[j][0] \leftarrow S[j-1][0] + M[0][j]$

4 **for**($i \leftarrow 1; i < n; i++$)

5   **for**($j \leftarrow 1; j < m; j++$)

6         $S[i][j] \leftarrow S[i-1][j] + S[i][j-1] - S[i-1][j-1] + M[i][j]$

7 **return** S

2D-Range Sum

- Para computar S nós gastamos tempo $\Theta(nm)$, pois cada entrada $S[i][j]$ leva tempo $\Theta(1)$ para ser computada.

Sumário

2 Programação Dinâmica
- 2D-Range Sum
- LIS
- Problema do troco

Longest Increasing Subsequence

Definição

Subsequência Seja $V = (v_0, \ldots, v_{n-1})$ uma sequência. Uma subsequência S de V se $S = (v_{i_0}, v_{i_1}, \ldots, v_{i_{k-1}})$ com $i_0 < i_1 < \ldots < i_{k-1}$.

Em outras palavras, S pode ser formada a partir de elementos de V quando lido da esquerda para a direita.

Longest Increasing Subsequence

Exemplo

$S = (-7, 9, 2, 3, 1)$ é uma subsequência de
$V = (-7, 10, 9, 2, 3, 8, 8, 1)$.
Especificamente, temos que $S = (v_0, v_2, v_3, v_4, v_7)$

Longest Increasing Subsequence

Longest Increasing Subsequence

O problema da subsequência crescente mais longa (Longest Increasing Subsequence, ou LIS) consiste em, dado uma sequência V, informar o tamanho da mais subsequência estritamente crescente de V, isto é, informar a subsequência $S = (s_0, \ldots, s_{k-1})$ de V que:

- É estritamente crescente, isto é, $S[i] < S[i+1], 0 < i < k-1$.
- Seja a maior possível.

Longest Increasing Subsequence

Exemplo

Para $V = (-7, 10, 9, 2, 3, 8, 8, 1)$, a maior subsequência crescente de V é:

$$S = (-7, 2, 3, 8)$$

a qual possui tamanho 4.

Longest Increasing Subsequence

Abordagem Gulosa (WA)

- Está claro que a abordagem gulosa, que pega o próximo elemento de V maior que o anterior não funciona.
- Para $V = (-7, 10, 9, 2, 3, 8, 8, 1)$, S incluiria o valor -7 e em seguida o valor de 10. Após a inclusão do último seria impossível incluir

Longest Increasing Subsequence

- Vamos tentar projetar uma solução via Programação Dinâmica.

Longest Increasing Subsequence

- Seja $L(k)$ o tamanho da maior subsequência de V que termine com v_k. Suponha que tenhamos $L(k)$ computado para qualquer $0 < k < i$, como podemos computar $L(i)$?

$$L(i) = \begin{cases} 1, & i = 0 \\ 1 + \max_{0<k<i}\{L(k) | v_k < v_i\}, & i > 0 \end{cases}$$

A resposta será o maior valor de $L(i)$ possível, para $0 \leq i < n$

Longest Increasing Subsequence

- Onde está a resposta?

Longest Increasing Subsequence

- Onde está a resposta?
- No maior valor de $DP[i]$ para $0 \leq i < n$.

Longest Increasing Subsequence

Algorithm 9: $\textsc{top-down-lis}(V, DP, i)$

Input: $V[0, n-1], DP[0, n-1], i$
Output: O tamanho da maior subsequência crescente de V

```
1 if( i = 0 )
2     return 1
3 if( DP[i] ≠ ⊥ )
4     return DP[i]
5 for( j ← i − 1; j ≥ 0; j − − )
6     DP[i] ← 1
7     if( V[i] > V[j] )
8         DP[i] ← max(DP[i], 1 + TOP-DOWN-LIS(V, DP, j))
9 return DP[i]
```

- Chamada inicial: $\textsc{top-down-lis}(V, DP, n-1)$

Longest Increasing Subsequence

Algorithm 10: BOTTOM-UP-LIS(V)

Input: $V[0, n-1]$

Output: O tamanho da maior subsequência crescente de V

```
1 ans ← 1
2 for( i ← 0; i < n; i++ ) DP[i] ← 1
3 for( i ← 1; i < n; i++ )
4     for( j ← 0; j < i; j++ )
5         if( V[i] > V[j] )
6             DP[i] ← max(DP[i], 1 + DP[j])
7     ans ← max(ans, DP[i])
8 return ans
```

Longest Increasing Subsequence

Complexidade de Pior-caso

- $\Theta(n^2)$.

Sumário

2 Programação Dinâmica
- 2D-Range Sum
- LIS
- Problema do troco

Problema do Troco

- Vimos que para sistemas canônicos de moeda, o problema do troco admite uma solução gulosa.
- Quando o sistema não é canônico, a estratégia gulosa pode não funcionar.
- Sera que é possível elaborar uma solução, baseada em Programação Dinâmica, que resolva o problema do troco para qualquer sistema de moedas?

Problema do Troco

Problema do Troco

- Entrada, um sistema de moedas $C = (c_0, \ldots, c_{n-1})$ e um valor W a ser pago.
- Saída: o menor número de moedas de C que paga V.

Observação: não há restrição na quantidade de moedas de cada valor.

Problema do Troco

- Seja $T(i, j)$ o número mínimo de moedas utilizadas considerando as i primeiras moedas e com valor j a ser pago.
- $T(0, 0)$ é 0, pois não precisamos de nenhuma moeda para pagar o troco de valor 0.
- É impossível pagar um valor de troco superior a 0 sem utilizar qualquer moeda, então $T(0, j) = \infty$ para qualquer $j > 0$.
- Supondo que T esteja computado para qualquer valor $T(a, b)$ com $0 \leq a < i$ e $0 \leq b \leq W$, como computar $T(i, j)$?

Problema do Troco

Existem duas possibilidades:

- A i-ésima moeda não está na solução. Então a solução de $T(i, j)$ vem de $T(i-1, j)$.
- A i-ésima moeda está na solução. Como podemos utilizá-la múltiplas vezes, a resposta de $T(i, j)$ pode vir de $T(i, j - C[i-1])$.
- No segundo caso pegamos o mínimo entre a estratégia que não considera a i-ésima moeda e a estratégia que a considera: $\min(T(i-1, j), 1 + T(i, j - C[i-1]))$.

Problema do Troco

- Vamos caracterizar a solução ótima recursivamente:

$$T(i,j) = \begin{cases} 0, & i = 0 \land j = 0 \\ \infty, & i = 0 \land j > 0 \\ \min(T(i-1,j), 1 + T(i, j - C[i-1])), & i > 0 \land j \geq C[i-1] \\ T(i-1,j), & \text{caso contrário} \end{cases}$$

- Se $T(i,j) = \infty$ significa que é impossível pagar o valor de j utilizando as i primeiras moedas.
- A resposta está em $T(n, W)$, a quantidade de moedas que paga W considerando as n moedas.

Problema do Troco

Algorithm 11: TOP-DOWN-COIN-CHANGE(C, DP, i, j)

Input: $C[0, n-1], DP[0, n][0, W], i, j$
Output: Número mínimo de moedas que paga W.

```
1  if( i = 0 ∧ j = 0 )
2      return 0
3  else if( i = 0 ∧ j > 0 )
4      return ∞
5  else if( DP[i][j] ≠ ⊥ )
6      return DP[i][j]
7  DP[i][j] ← TOP-DOWN-COIN-CHANGE(C, DP, i − 1, j)
8  if( C[i − 1] ≤ j )
9      DP[i][j] ← min(DP[i][j], 1 + TOP-DOWN-COIN-CHANGE(C, DP, i, j − C[i − 1]))
10 return DP[i][j]
```

- Chamada inicial: TOP-DOWN-COIN-CHANGE(C, DP, n, W)

Problema do Troco

Algorithm 12: BOTTOM-UP-COIN-CHANGE(C,W)

Input: $C[0, n-1], W$

Output: Número mínimo de moedas que paga W.

```
1 DP[0][0] ← true
2 for( j ← 1; j ≤ W; j++ ) DP[0][j] ← ∞
3 for( i ← 1; i ≤ n; i++ )
4     for( j ← 0; j ≤ W; j++ )
5         DP[i][j] ← DP[i−1][j]
6         if( C[i−1] ≤ j )
7             DP[i][j] ← min(DP[i][j], 1 + DP[i][j − C[i−1]])
8 return DP[n][W]
```

Problema do Troco

- É possível salvar mais espaço armazenando apenas 2 linhas da matriz de programação dinâmica: a atual e a anterior.

Problema do Troco

Complexidade de Pior-caso

- $\Theta(n \cdot W)$.

Sumário

3 Considerações Finais

Considerações

- Programação dinâmica: uma forma eficiente de resolver problemas quem possuem a propriedade de subestrutura ótima e sobreposição de subproblemas.
- Os problemas podem ser resolvidos de maneira *top-down* ou *bottom-up*.

Considerações

Abordagem *top-down*: Vantagens

- Solução é obtida diretamente da relação de recorrência.
- Mais eficiente se os mesmos estados são revisitados frequentemente.

Considerações

Abordagem *top-down*: desvantagens

- Mais lenta se temos vários estados diferentes sendo visitados.
- Se existem M estados, é necessário uma tabela de tamanho M, o que pode levar a um MLE. Contudo, é possível mitigar isso com um mapeamento.

Considerações

Abordagem *bottom-up*: Vantagens

- Eficiente se vários estados diferentes são alcançados
- Podemos economizar memória guardando apenas as informações realmente necessárias na tabela de PD.

Considerações

Abordagem *bottom-up*: Desvantagens

- Não é natural para quem está acostumado com projeto de algoritmos usando recursividade.
- Pode ser mais lenta do que a solução *top-down* se vários estados são visitados muitas vezes.