UNIVERSIDADE FEDERAL DE OURO PRETO
PCC104 - Projeto e Análise de Algoritmos
Prof. Rodrigo Silva

Programação Dinâmica

Sara Câmara

29 de maio de 2023

3 Questão Extra

Problema: Implemente um algoritmo para o problema do troco (*Change-making problem* (Seção 8.1)) utilizando programação dinâmica.

Para cada implementação, apresentar a análise de complexidade de tempo do algoritmo.

Solução:

a) Implementação:

```
def change_making(change, coins):
    if change == 0:
        return 0
    if change == 1:
        return 1
    memory = [0] + [float('inf')] * change
    for coin in coins:
        count = coin
        while count <= change:
            memory[count] = min(memory[count], memory[count - coin] + 1)
            count += 1
    if memory[change] != float('inf'):
        return memory[change]
    else:
        return -1
```

b) Operação básica: Comparação para encontrar a menor quantidade de moedas para o troco.

```
min(memory[count], memory[count - coin] + 1)
```

c) Equação de custo:

Os valores de moedas podem ser modelados por um conjunto de n valores inteiros positivos distintos (números inteiros), organizados em ordem crescente de w_1 a w_n. O problema é: dada uma quantidade W, também um inteiro positivo, encontrar um conjunto de inteiros não negativos (positivos ou nulos) $x_1, x_2, ..., x_n$, com cada x_j representando quantas vezes a moeda com valor w_j é usado, o que minimiza o número total de moedas $f(W)$.

$$F(W) = \sum_{j=1}^{n} x_j \quad sujeito\ a \quad \sum_{j=1}^{n} w_j x_j = W$$
$$F(0) = 0$$

d) Cálculo da equação de recorrência:

$$F(n) = min_{j:n \geq d_j}\{F(n - d_j)\} + 1 \quad \forall \ \ n > 0,$$
$$F(0) = 0$$

Nessa relação de recorrência, $F(n)$ representa o número mínimo de moedas necessárias para fazer o troco no valor n usando as moedas disponíveis.

- Se n for igual a 0, significa que nenhuma mudança é necessária, portanto, o número mínimo de moedas é 0.
- Para n maior que 1, é considerado cada moeda e calculado o número mínimo de moedas necessário para fazer o troco por n, calculando recursivamente o número mínimo de moedas necessário para $(n - coin) + 1$.

A aplicação do algoritmo para quantidade n = 8 e denominações $1, 5, 10, 25, 50, 100$:

$F(0) = 0$
$F(1) = min\{F(1-1)\} + 1 = 1$
$F(2) = min\{F(2-1)\} + 1 = 2$
$F(3) = min\{F(3-1)\} + 1 = 3$
$F(4) = min\{F(4-1)\} + 1 = 4$
$F(5) = min\{F(5-1), F(5-5)\} + 1 = 1$
$F(6) = min\{F(6-1), F(6-5)\} + 1 = 2$
$F(7) = min\{F(7-1), F(7-5)\} + 1 = 3$
$F(8) = min\{F(8-1), F(8-5)\} + 1 = 4$

n	0	1	2	3	4	5	6	7	8
F	0	1	2	3	4	1	2	3	**4**

A resposta que esse algoritmo produz para $change = 8$ é 4 moedas.

e) Eficiência (O ou Θ):

Eficiência de tempo: $O(nm)$
Eficiência de espaço: $\Theta(n)$

- Melhor caso: Solução ótima em somente uma operação, por exemplo, troco = 1 com 1 moeda.
- Pior caso: Pior solução, por exemplo, troco = 4 com 4 moedas.
- Caso Médio: Analisado anteriormente.

4 Questão

Problema: Implemente um algoritmo para o problema de coleta de moedas (*Coin-collecting problem* (Seção 8.1)) utilizando programação dinâmica.

Para cada implementação, apresentar a análise de complexidade de tempo do algoritmo.

Solução:

a) Implementação:

```
def coin_collecting(coins):
    n = len(coins)
    m = len(coins[0])

    max_coins = [[0] * m for _ in range(n)]

    for i in range(n):
        for j in range(m):
            if i == 0 and j == 0:
                max_coins[i][j] = coins[i][j]
            elif i == 0:
                max_coins[i][j] = max_coins[i][j - 1] + coins[i][j]
            elif j == 0:
                max_coins[i][j] = max_coins[i - 1][j] + coins[i][j]
            else:
                max_coins[i][j] = max(max_coins[i - 1][j],
                    max_coins[i][j - 1]) + coins[i][j]

    return max_coins[n - 1][m - 1]
```

b) Operação básica: Comparação entre as posições próximas da quantidade de moedas a serem coletadas para saber qual é o número que representa o máximo.

```
    max_coins[i][j] = max_coins[i][j - 1] + coins[i][j]
```

ou

```
    max_coins[i][j] = max(max_coins[i - 1][j],
                          max_coins[i][j - 1]) + coins[i][j]
```

ou

```
    max_coins[i][j] = max_coins[i][j - 1] + coins[i][j]
```

c) Equação de recorrência:

$$F(i,\ j) = max\{F(i-1,\ j),\ F(i,\ j-1)\} + c_{ij} \quad \forall \ \ 1 \leq i \leq n, \ \ 1 \leq j \leq m$$
$$F(0,\ j) = 0 \quad \forall \ \ 1 \leq j \leq m \quad e \quad F(i,\ 0) = 0 \quad \forall \ \ 1 \leq i \leq n,$$

onde $c_{ij} = 1$ se houver uma moeda na célula (i, j), e $cij = 0$ caso contrário.

d) Eficiência (O ou Θ):

Eficiência de tempo: $\Theta(nm)$
Eficiência de espaço: $\Theta(nm)$

5 Questão

Problema: Implemente um algoritmo para o problema de coleta de moedas (*Coin-collecting problem* (Seção 8.1)) sem utilizar programação dinâmica.

Para cada implementação, apresentar a análise de complexidade de tempo do algoritmo.

Solução:

a) Implementação:

```
def coin_collecting(coins):
    n = len(coins)
    m = len(coins[0])

    max_coins = 0

    def dfs(i, j, collected_coins):
        nonlocal max_coins

        if i == n - 1 and j == m - 1:
            max_coins = max(max_coins, collected_coins)
            return

        if i < n - 1:
            dfs(i + 1, j, collected_coins + coins[i + 1][j])
        if j < m - 1:
            dfs(i, j + 1, collected_coins + coins[i][j + 1])


    dfs(0, 0, coins[0][0])

    return max_coins
```

b) Operação básica: Comparação entre as posições próximas da quantidade de moedas a serem coletadas para saber qual é o número que representa o máximo.

```
max_coins = max(max_coins, collected_coins)
```

c) Equação de recorrência:

$$F(i,\ j) = max\{F(i-1,\ j),\ F(i,\ j-1)\} + c_{ij} \quad \forall \;\ 1 \leq i \leq n, \;\ 1 \leq j \leq m$$
$$F(0,\ j) = 0 \quad \forall \;\ 1 \leq j \leq m \quad e \quad F(i,\ 0) = 0 \quad \forall \;\ 1 \leq i \leq n,$$

onde $c_{ij} = 1$ se houver uma moeda na célula (i, j), e $cij = 0$ caso contrário.

d) Eficiência (O ou Θ):

Eficiência de tempo: $\Theta(nm)$
Eficiência de espaço: $\Theta(nm)$

6 Questão

Problema: Implemente um algoritmo baseado em função de memória (*memory function*) para solução do problema da mochila (*knapsack problem*).

Para cada implementação, apresentar a análise de complexidade de tempo do algoritmo.

Solução:

a) Implementação:

```
def knapsack(weights, values, capacity):
    n = len(weights)

    max_value = [[-1] * (capacity + 1) for _ in range(n + 1)]

    def knapsack_memoization(i, j):
        if max_value[i][j] >= 0:
            return max_value[i][j]

        if i == 0 or j == 0:
            value = 0
        elif j < weights[i - 1]:
            value = knapsack_memoization(i - 1, j)
        else:
            value = max(knapsack_memoization(i - 1, j), values[i - 1] +
                        knapsack_memoization(i - 1, j - weights[i - 1]))

        max_value[i][j] = value
        return value

    return knapsack_memoization(n, capacity)
```

b) Operação básica: Comparação entre o item atual (avaliado) e o histórico de itens (já foi avaliado) para saber qual possui o valor máximo.

```
max(knapsack_memoization(i - 1, j), values[i - 1] +
                        knapsack_memoization(i - 1, j - weights[i - 1]))
```

c) Equação de custo:

O problema mais comum a ser resolvido é o problema knapsack (ou mochila 0-1), que restringe o número x_i de cópias de cada tipo de item a zero ou um. Dado um conjunto de n itens numerados de 1 a n, cada um com um peso w_i e um valor v_i, juntamente com uma capacidade máxima de peso W,

$$max \sum_{i=1}^{n} v_i x_i \quad sujeito\ a \quad \sum_{i=1}^{n} w_i x_i \leq W \quad e \quad x_i \in 0,\ 1$$

Aqui x_i representa o número de instâncias do item i a serem incluídas na mochila. Informalmente, o problema consiste em maximizar a soma dos valores dos itens da mochila de forma que a soma dos pesos seja menor ou igual à capacidade da mochila.

d) Cálculo da equação de recorrência:

$$F(i,\ j) = \begin{cases} max\{F(i-1,\ j), v_i + F(i-1,\ j-w_i)\} & \text{se } j - w_i \geq 0, \\ F(i-1,\ j) & \text{se } j - w_i < 0. \end{cases}$$

Definidas as condições iniciais da seguinte forma:

$$F(0,\ j) = 0 \ \ para \ \ j \geq 0 \quad e \quad F(i,\ 0) = 0 \ \ para \ \ i \geq 0$$

Exemplo:

- $Pesos : \{3, 2, 4, 1\}$
- $Valores : \{8, 3, 9, 6\}$
- $W = 5$

i	1	2	3	4
w_i	3	2	4	1
v_i	8	3	9	6

$\rightarrow$	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	0	8	8	8
$\downarrow$ 2	0	0	3	8	8	11
3	0	0	3	8	9	11
4	0	6	6	9	14	**15**

$Para\ i = 1$
$F(1,\ 1), w_1 = 3, j = 1, w_1 \leq j$
$\Rightarrow F(i,\ j) = F(i-1, j) \Rightarrow F(1,\ 1) = F(0,\ 1) = 0$

$Analogamente : F(1,\ 2) = F(0,\ 2) = 0$

$F(1,\ 3), w_1 = 3, j = 3, w_1 \leq j$
$\Rightarrow F(i,\ j) = max\{F(i-1, j), F(i-1, j-w_i) + v_i\}$
$\Rightarrow F(1,\ 1) = max\{F(0, 3), F(0, 0) + 8\} = 8$

$Analogamente :$
$F(1,\ 4) = max\{F(0, 4), F(0, 1) + 8\} = 8$
$F(1,\ 5) = max\{F(0, 5), F(0, 2) + 8\} = 8$

$Para\ i = 2$
$F(2,\ 1) = F(1,\ 1) = 0$
$F(2,\ 2) = max\{F(1, 2), F(1, 0) + 3\} = 3$
$F(2,\ 3) = max\{F(1, 3), F(1, 1) + 3\} = 8$
$\vdots$

$Para\ i = 4\ e\ j = 5$
$F(4,\ 5) = max\{F(4, 5), F(4, 2) + 6\} = 15$

e) Eficiência (O ou Θ):

O consumo de tempo do algoritmo é proporcional ao tamanho da tabela t. Portanto o algoritmo consome $\Theta(nW)$ unidades de tempo.

Observando o desempenho da programação dinâmica para o problema *SubsetSUM*, é melhor dizer que o algoritmo consome $\Theta(n2^{\log w})$ unidades de tempo e deixar claro que o algoritmo é exponencial.

SubsetSUM: Cada execução da linha do algoritmo que preenche uma casa da tabela $t[0..n, 0..w]$ consome uma quantidade constante de tempo. Logo, o consumo total de tempo do algoritmo é proporcional ao tamanho da tabela. Como a tabela tem $n+1$ linhas e $w+1$ colunas, o consumo de tempo está em $\Theta(nW)$, Como exemplo, para colocar em foco o efeito do alvo w, o alvo e os pesos são multiplicados por 1000.

Esta operação é uma mera mudança de escala ou de unidades de medida (como mudar de kilogramas para gramas), e portanto a nova instância é conceitualmente idêntica à original e também $\Theta(nW)$. Porém,

deve ser analisado o tamanho de w e não o seu valor. O tamanho de w, ou seja, o número de dígitos de w, é cerca de $\log w$ (O tamanho de 100000, por exemplo, é 6.) O consumo de tempo deveria, portanto, ser escrito como $\Theta(n10^{\log w})$ ou, equivalentemente, $\Theta(n2^{\log w})$

- Eficiência de tempo: $\Theta(nW)$ ou $\Theta(n2^{\log w})$

- Eficiência de espaço: $\Theta(nW)$

- **Melhor caso**: eficiência de tempo em $O(n)$.