

# Resolução – Lista 4 (Projeto e Análise de Algoritmos)

Fevereiro - 2023 / Leticia Bossatto Marchezi – 791003

## Questão 1

---

Explique o que é e como funciona a programação dinâmica

### Resolução:

A programação dinâmica é uma abordagem para solucionar problemas de forma a combinar resoluções de problemas menores. Diferente da estratégia dividir para conquistar, que lida com subproblemas que não possuem sobreposição, na programação dinâmica a resolução de suas partes pequenas dependem uma da outra, necessitando uma abordagem diferente e usando de estratégias para otimizar tais procedimentos, seja partindo de um caso base e avançando na resolução, ou iniciando do caso geral e armazenando os subcálculos para evitar repetição. □

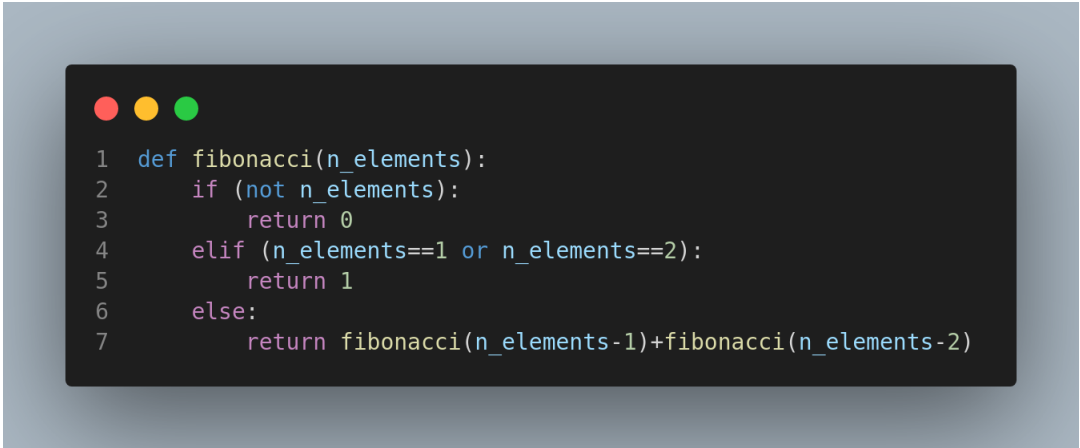
## Questão 2

---

Escreva uma função recursiva para calcular a o n-ésimo termo da série de Fibonacci. Calcule a complexidade dessa função e explique porque ela não é eficiente.

### Resolução:

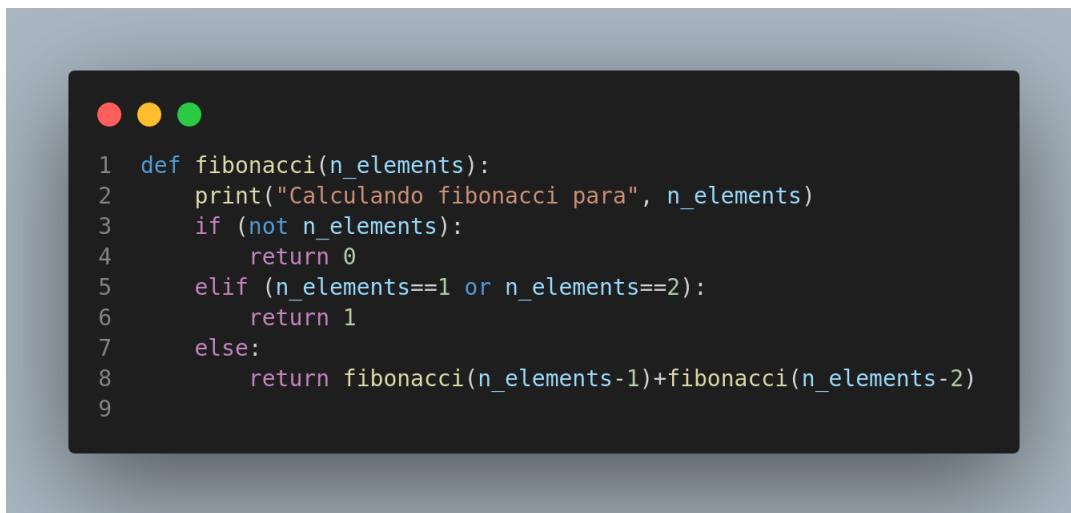
A seguinte função recursiva na linguagem Python retorna o n-ésimo termo da série de Fibonacci.



```
1 def fibonacci(n_elements):
2     if (not n_elements):
3         return 0
4     elif (n_elements==1 or n_elements==2):
5         return 1
6     else:
7         return fibonacci(n_elements-1)+fibonacci(n_elements-2)
```

Figure 1: Questão 2

Entretanto, esta abordagem não é ideal pois para cada termo será calculado todos seus antecessores. Ao adicionar um comando de output na primeira linha da função, é possível observar que há uma sequência de recursões repetidas e que são necessariamente refeitas para cada termo.

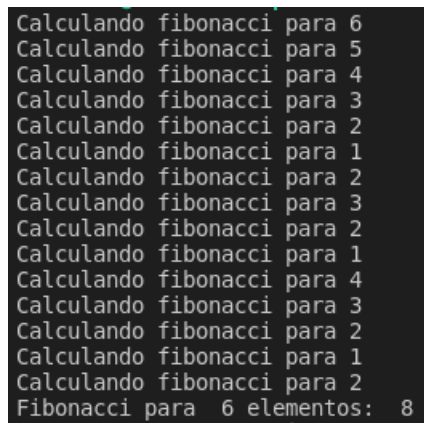


```

1 def fibonacci(n_elements):
2     print("Calculando fibonacci para", n_elements)
3     if (not n_elements):
4         return 0
5     elif (n_elements==1 or n_elements==2):
6         return 1
7     else:
8         return fibonacci(n_elements-1)+fibonacci(n_elements-2)
9

```

Figure 2: Questão 2 - prints



```

Calculando fibonacci para 6
Calculando fibonacci para 5
Calculando fibonacci para 4
Calculando fibonacci para 3
Calculando fibonacci para 2
Calculando fibonacci para 1
Calculando fibonacci para 2
Calculando fibonacci para 3
Calculando fibonacci para 2
Calculando fibonacci para 1
Calculando fibonacci para 4
Calculando fibonacci para 3
Calculando fibonacci para 2
Calculando fibonacci para 1
Calculando fibonacci para 2
Fibonacci para 6 elementos: 8

```

Figure 3: Questão 2 - outputs

Assim, tem-se a seguinte função de recorrência:

$$T(n) = T(n-1) + T(n-2) + O(1) \quad (2.1)$$

Aproximando  $n$  suficientemente grande,  $T(n-1)$  é aproximadamente  $T(n-2)$ . Resultando em:

$$T(n) = 2T(n-1) + O(1) \quad (2.2)$$

E recursivamente:

$$T(n) = 2^2 T(n-2) + O(1) \quad (2.3)$$

$$T(n) = 2^3 T(n-3) + O(1) \quad (2.4)$$

Então, para o  $k$ -ésimo termo a função de recorrência é:

$$T(n) = 2^k T(n-k) + O(1) \quad (2.5)$$

Como a parada da função é obtida para  $k=n$ , assim:

$$T(n) = 2^n T(0) + O(1) \quad (2.6)$$

Logo, como  $2^n$  é o termo dominante, conclui-se que a complexidade da função é  $O(2^n)$

□

## Questão 3

Explique como podemos utilizar a programação dinâmica para desenvolver algoritmos mais eficientes para a sequência de Fibonacci:

### Resolução:

- a) Abordagem Top-Down (memorização) A abordagem Top-Down para o problema da sequência de Fibonacci é constituída pelo armazenamento de cálculos dos menores termos e os utilizando na soma dos próximos termos, evitando o recálculo e criação de chamadas recursivas redundantes.
- b) Abordagem Botton-Up (reversão) Não utiliza chamadas recursivas, mas resolve o problema partindo do caso base ( $n=0$ ,  $n=1$  ou  $n=2$ ) calculando crescentemente os termos de forma a somar o termo atual com o anterior, obtendo o próximo termo.

□

## Questão 4

Projete um algoritmo recursivo para o problema da sequência de cédulas, sem a utilização de programação dinâmica. Calcule a complexidade da sua função e explique se ela é eficiente.

### Resolução:

O algoritmo recursivo que resolve o problema da sequência de cédulas é:



```
1 def pegaCedulas(tam):
2     if not tam:
3         return c[0]
4     elif tam == 1:
5         return c[1]
6     else:
7         return max(c[tam]+pegaCedulas(tam-2), pegaCedulas(tam-1))
```

Figure 4: Questão 4

E sua função de recorrência é similar à de Fibonacci, criando 2 chamadas recursivas no caso geral e mais uma operação de complexidade  $O(1)$  de comparação do valor máximo.

Assim, a função de recorrência é:

$$T(n) = T(n-1) + T(n-2) + O(1) \quad (4.1)$$

Aproximando  $n$  suficientemente grande,  $T(n-1)$  é aproximadamente  $T(n-2)$ . Resultando em:

$$T(n) = 2T(n-1) + O(1) \quad (4.2)$$

e por consequência:

$$T(n) = 2^n T(0) + O(1) \quad (4.3)$$

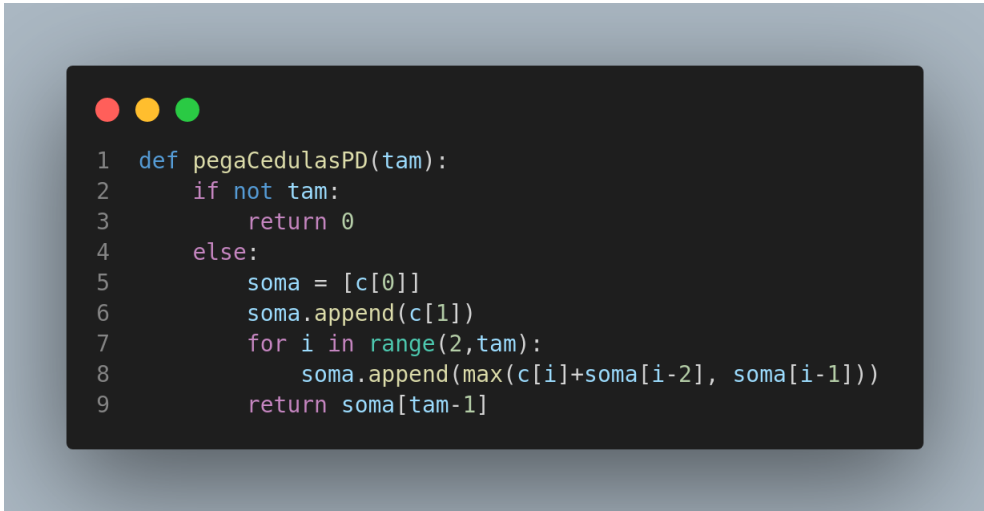
Logo, como  $2^n$  é o termo dominante, conclui-se que a complexidade da função é  $O(2^n)$

Por fim, pode-se concluir que a função não é eficiente pois pode ser aprimorada para combinar a resolução dos subproblemas. O cálculo da soma das sequências de cédulas coletadas é feito de forma redundante à medida que a quantidade de cédulas aumenta, enquanto seus valores poderiam ser armazenados e apenas somados à uma nova cédula, ou resolver o problema partindo do caso base até o caso geral.  $\square$

## Questão 5

Ainda sobre o problema da sequência de cédulas, utilize programação dinâmica com uma abordagem Bottom-Up (não recursiva) para desenvolver uma função mais eficiente. Calcule a complexidade da nova função e compare com a função recursiva.

**Resolução:**



```
1 def pegaCedulasPD(tam):
2     if not tam:
3         return 0
4     else:
5         soma = [c[0]]
6         soma.append(c[1])
7         for i in range(2,tam):
8             soma.append(max(c[i]+soma[i-2], soma[i-1]))
9         return soma[tam-1]
```

Figure 5: Questão 5

A complexidade da função `pegaCedulasPD` é  $O(n)$  pois há um loop de custo  $O(n)$  e uma operação de custo constante  $O(1)$ . Assim, essa função tem desempenho superior em comparação à versão não otimizada, que cresce exponencialmente na base 2 ( $O(2^n)$ )

$\square$

## Questão 6

Considere a sequência de  $n = 12$  cédulas a seguir:  $C = [2, 5, 5, 2, 10, 50, 100, 50, 20, 20, 50, 100]$  Sabendo que  $F$  é um vetor em que  $F[0] = 0$  e  $F[1] = c_1$ , execute manualmente o algoritmo desenvolvido no exercício 5 para obter a solução do problema, ou seja, o máximo valor de dinheiro que pode ser coletado sem que 2 cédulas vizinhas sejam obtidas. Você deve gerar todos os valores de  $F[i]$  para  $i$  iniciando em 2 e terminando em  $n$ .

**Resolução:**

$$F[0] = 2$$

$$F[1] = 5$$

$F[2] = \max(7, 5) = 7$   
 $F[3] = \max(7, 7) = 7$   
 $F[4] = \max(17, 7) = 17$   
 $F[5] = \max(57, 17) = 57$   
 $F[6] = \max(117, 57) = 117$   
 $F[7] = \max(107, 117) = 117$   
 $F[8] = \max(137, 117) = 137$   
 $F[9] = \max(137, 137) = 137$   
 $F[10] = \max(187, 137) = 187$   
 $F[11] = \max(237, 187) = 237$   
Resultado: 237



## Questão 7

Projete um algoritmo para o problema do robô coletor de moedas usando programação dinâmica. Calcule a complexidade da sua função e explique se ela é eficiente.

**Resolução:**



## Questão 8

Considere o seguinte tabuleiro de entrada para o problema do robô coletor de moedas. Execute o algoritmo desenvolvido no exercício 7 para solucionar essa instância do problema do robô coletor de moedas. Calcule todos os valores de  $F$ , ou seja, preencha o quadro da direita. Qual é o trajeto que deve ser percorrido pelo robô iniciando na posição  $(1, 1)$ ?

**Resolução:**

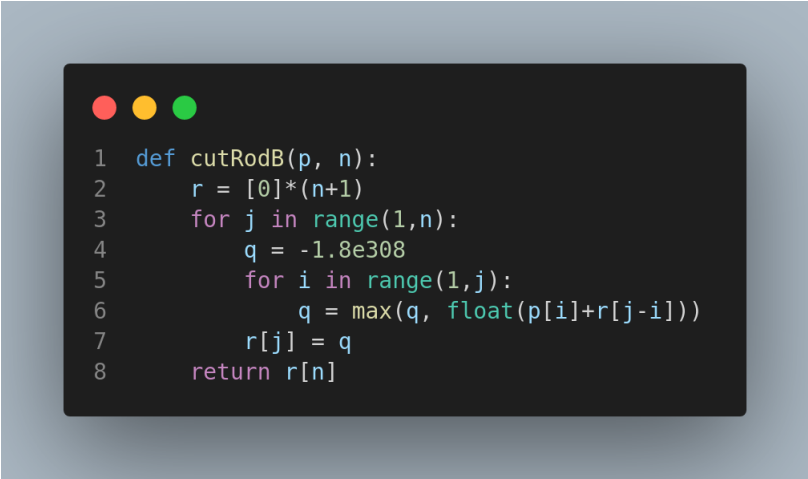


## Questão 9

O problema do corte da haste nos diz que dada uma haste de comprimento  $n$  e uma tabela de preços para cada possível pedaço da haste, devemos maximizar o ganho. Projete um algoritmo recursivo para o problema do corte da haste usando programação dinâmica. Calcule a complexidade da sua função e explique se ela é eficiente.

**Resolução:**

O código abaixo soluciona o problema da haste de comprimento com complexidade  $O(n^2)$  devido ao aninhamento de 2 laços de repetição com custo  $O(n)$  cada, e uma operação de cálculo de valor máximo de  $O(1)$ . Assim, essa versão é mais eficiente do que a versão sem a programação dinâmica, que possui complexidade  $O(2^n)$



```
1 def cutRodB(p, n):  
2     r = [0]*(n+1)  
3     for j in range(1,n):  
4         q = -1.8e308  
5         for i in range(1,j):  
6             q = max(q, float(p[i]+r[j-i]))  
7         r[j] = q  
8     return r[n]
```

Figure 6: Questão 9

