MATA52 - Exercícios da Semana 08

- Grupo: Paladio
- Autores:
    - Monique Santos da Silva (responsável)
        - Respondi a quarta questão.
    - Lucas dos Santos Lima
        - Respondi a ? questão.
    - Elis Marcela de Souza Alcantara
        - Resolvi a primeira questão.
    - Bruno de Lucas Santos Barbosa
        - Resolvi a segunda questão

Instruções (não apagar)

1. **Responsável**: Após copiar este notebook, altere o nome do notebook/documentação incluindo o nome do seu grupo. Por exemplo, se você é do grupo Ouro, altere o nome do notebook para "MATA53-Semana02-Ouro.ipynb"
2. **Responsável**: Compartilhe este documento com todos os membros do grupo (para participarem da elaboração deste documento). É importante que o notebook utilizado seja o mesmo compartilhado para que os registros de participação e colaboração fiquem salvos no histórico. Sugira uma divisão justa e defina um prazo aceitável para a inserção das soluções no Colab.
3. **Responsável**: Ao concluir a atividade, compartilhe o notebook com januario.ufba@gmail.com (dando permisão para edição) e deixando o aviso de notificação marcado, para que eu receba o seu e-mail. Identificar o nome do grupo na mensagem de compartilhamento.
4. **Cada membro**: Incluia o *seu próprio nome completo* na lista de autores que auxiliaram na elaboração deste notebook. Relate brevemente a sua contribuição na solução desta lista. O responsável aparece como sendo o(a) primeiro(a) autor(a).
5. **Cada membro**: Utilize os recursos de blocos de texto e de código para adicionar as suas respostas, sem alterar os blocos de texto e código existente. Não economize, esses blocos são de graça.

Exercícios

Para cada um dos seguintes problemas:

- Defina os subproblemas
- Reconheça e resolva os casos base
- Apresente a recorrência que resolva os subproblemas
- Apresente uma implementação ingênua
- Apresente uma solução utilizando PD "top-down" ou "bottom-up"
- Analise a complexidade da solução utilizando PD

1. Dado um número inteiro n, determine o número de formas diferentes de se escrever n como a soma dos números 1, 3 e 4.

S_n (solução de n) é considerado o a quantidade de jeitos diversos que podemos escrever um inteiro n utilizando o conjunto $1, 3, 4$. Como por exemplo, o número 6 pode ser representado dessa forma:

1+1+1+1+1+1

3+1+1+1

1+3+1+1

1+1+3+1

1+1+1+3

3+3

4+1+1

1+4+1

1+1+4

Então 6 pode ser representado de 9 formas, $S_6 = 9$.

Os casos base podem ser definidos como:

Solução de n	Ocorrência
$S_n = 0$	quando $n = 0$
$S_n = 1$	quando $n = 2$, acontece apenas com $1 + 1$
$S_n = 2$	quando $n = 3$, acontece com $1 + 1 + 1$ e 3
$S_n = 1$	quando $n = 1$, acontece apenas com 1

Logo, chegamos a conclusão de que inteiros n além dos contemplados pelos casos base podem ser solucionados com a seguinte recorrência:

$$S_n = S_{n-1} + S_{n-3} + S_{n-4}$$

Poderiamos implementar de forma ingênua já que sabemos a recorrência usando recursividade:

```
def formasDeEscreverNcomoSomaDeUmTresQuatro(n):
    if n == 0:
        return 1

    if n < 0:
        return 0

    return formasDeEscreverNcomoSomaDeUmTresQuatro(n - 1) + formasDeEscreverNcomoS

print(formasDeEscreverNcomoSomaDeUmTresQuatro(6))
```

```
9
```

Porém, sabemos que em linguagens como Python utilizar recursão é caro e como temos também os casos base podemos implementar utilizando memoization (Top Down):

```
import numpy as np

def formasDeEscreverNcomoSomaDeUmTresQuatro(n):
    array_range = n + 1
    Sn = np.empty(array_range, dtype=object)

    Sn[0] = Sn[1] = Sn[2] = 1 # Casos base
    Sn[3] = 2 # Caso base


    for i in range(4, array_range):
        Sn[i] = Sn[i - 1] + Sn[i - 3] + Sn[i - 4] # Recorrência

    return Sn[n]

print(formasDeEscreverNcomoSomaDeUmTresQuatro(6))
```

```
9
```

Utilizamos memoization então nunca iremos calcular uma combinação mais de uma vez. O trabalho realizado por cada chamada basicamente está no loop `for i in range(4, array_range):`, `array_range` sendo `n + 1`, dependemos da entrada, logo, a complexidade é O(n).

2. Dada uma matriz formada por apenas 0s e 1s, encontre a maior submatriz quadrada formada apenas por 1s.

Olhando para os algoritmos possíveis, vemos que os subproblemas não são tão complexos, é preciso fazer a leitura da matriz e guardar os valores a cada índice desde que possuam o valor de 1, além de armazenar sem que haja perca de índice e torne a nova matriz errada.

```
matriz = [[0,1,0,1], [1,0,1,0], [1,0,1,0], [1,0,1,0]]
# 0 , 1 , 0 , 1
# 1 , 0 , 1 , 0
# 1 , 0 , 1 , 0
# 1 , 0 , 1 , 0
# 1 , 0 , 1 , 0
novaMatriz = [];
for l in range(len(matriz)):
    for c in range(len(matriz)):
      if (matriz[l][c] == 1):
        novaMatriz.append(1);
print(novaMatriz)
```

A partir desse código base, teríamos implementações mais "brutas" para resolver os sub-problemas, como o problema pede para que seja a maior sub-quadrada, a inserção de condicionais para medir o tamanho que não ultrapasse o número máximo de linhas e colunas.

3. Dada uma matriz formada por apenas 0s e 1s, encontre a maior submatriz formada apenas por 1s. Muito próximo ao problema anterior, porém sem a restrição de ser quadrada.

4. Encontre um subvetor com a maior soma possivel. Por exemplo, para a seguinte sequência de valores $[-2,1,-3,4,-1,2,1,-5,4]$$[-2,1,-3,4,-1,2,1,-5,4]$ o segmento de soma máxima é $[4,-1,2,1]$$[4,-1,2,1]$ cuja soma é 6.

Podemos encontrar o subvetor com a maior soma possível mantendo a soma máxima do subvetor que termina em cada índice do vetor. Utilizando o algoritmo de Kadane é possível encontrar a soma máxima de um subvetor contíguo de um vetor com tempo de execução linear $O(n)$$O(n)$ e espaço O(1).

```
def maiorSubvetor(sequencia):
    dp = maximo = sequencia[0]
    for i in range(1, len(sequencia)):
```

```
        dp = max(sequencia[i], dp + sequencia[i])
        if dp < 0:
            dp = 0
        maximo = max(dp, maximo)
    return maximo

print("A soma do maior subvetor é:", maiorSubvetor([-2, 1, -3, 4, -1, 2, 1, -5, 4]
```

```
A soma do maior subvetor é: 6
```

Para o vetor acima, o subvetor com a maior soma é [4, -1, 2, 1], cuja soma resulta em 6.

Análise

Pior caso:

Complexidade: $O(n^2)$$O(n^2)$

O pior caso consiste em calcular a soma de todos os subvetores possíveis, o máximo deles seria a solução.

Exemplo:

Dada a sequência dos valores $[−2, 1, −3, 4, −1, 2, 1, −5, 4]$$[−2, 1, −3, 4, −1, 2, 1, −5, 4]$, podemos definir seus subvetores:

$[−2, 1, −3, 4, −1, 2, 1, −5, 4]$$[−2, 1, −3, 4, −1, 2, 1, −5, 4]$,

$[1, -3, 4, -1, 2, 1, −5, 4]$$[1, -3, 4, -1, 2, 1, −5, 4]$,

$[-3, 4, -1, 2, 1, −5, 4]$$[-3, 4, -1, 2, 1, −5, 4]$,

$[4, -1, 2, 1, −5, 4]$$[4, -1, 2, 1, −5, 4]$,

$[-1, 2, 1, −5, 4]$$[-1, 2, 1, −5, 4]$,

$[2, 1, −5, 4]$$[2, 1, −5, 4]$,

$[1, −5, 4]$$[1, −5, 4]$,

$[−5, 4]$$[−5, 4]$,

$[4]$$[4]$

O maior subvetor é [4, -1, 2, 1, −5, 4], cuja soma resulta em 5.

Retirando o último elemento obtemos: $[−2, 1, −3, 4, −1, 2, 1, −5]$$[−2, 1, −3, 4, −1, 2, 1, −5]$, podemos definir seus subvetores:

$[−2, 1, −3, 4, −1, 2, 1, −5]$$[−2, 1, −3, 4, −1, 2, 1, −5]$,

$[1, -3, 4, -1, 2, 1, −5]$$[1, -3, 4, -1, 2, 1, −5]$,

$[-3, 4, -1, 2, 1, −5]$$[-3, 4, -1, 2, 1, −5]$,

$[4, -1, 2, 1, −5]$$[4, -1, 2, 1, −5]$,

$[-1, 2, 1, −5]$$[-1, 2, 1, −5]$,

$[2, 1, −5]$$[2, 1, −5]$,

$[1, −5]$$[1, −5]$,

$[−5]$$[−5]$,

O maior subvetor é [4, -1, 2, 1, −5], cuja soma resulta em 1.

Retirando novamente o último elemento obtemos: $[−2, 1, −3, 4, −1, 2, 1]$$[−2, 1, −3, 4, −1, 2, 1]$, podemos definir seus subvetores:

$[−2, 1, −3, 4, −1, 2, 1]$$[−2, 1, −3, 4, −1, 2, 1]$,

$[1, -3, 4, -1, 2, 1]$$[1, -3, 4, -1, 2, 1]$,

$[-3, 4, -1, 2, 1]$$[-3, 4, -1, 2, 1]$,

$[4, -1, 2, 1]$$[4, -1, 2, 1]$,

$[-1, 2, 1]$$[-1, 2, 1]$,

$[2, 1]$$[2, 1]$,

$[1]$$[1]$,

O maior subvetor é $[4, -1, 2, 1]$$[4, -1, 2, 1]$, cuja soma resulta em 6. Encontramos assim o subvetor da questão.

O cenário descrito seria o pior caso, visto que teriamos que percorrer todos os subvetores, à medida que o tamanho do vetor aumenta, o número de subvetores possíveis aumenta rapidamente. Se o tamanho do vetor for n, então a complexidade de tempo desta solução é $O(n^2)$$O(n^2)$.

Melhor caso:

Complexidade: $O(n)$$O(n)$

Com a finalidade de obter o melhor caso, $O(n)$$O(n)$, foi ultilizado o algoritmo de Kadane para evitar cálculos repetidos. O algoritmo procura todos os segmentos contíguos positivos e acompanhar o subvetor contíguo de soma máxima entre todos os segmentos positivos.

Primeiro, vamos considerar dois elementos, um que armazena a extremidade máxima do subvetores e outro que armazena a soma máxima até o momento.

Sejam essas duas variáveis `dp e maximo`, respectivamente. Cada vez que obtemos uma soma positiva, comparamos com `maximo` e atualizamos `maximo` se for o maior valor.

A cada índice i, o problema se resume a encontrar o máximo de apenas dois números, `dp` e `maximo`. Assim, o problema do subvetor máximo pode ser resolvido resolvendo esses subproblemas de encontrar `dp` recursivamente.

Analisando a complexidade, precisamos percorrer o array apenas uma vez, então a complexidade do tempo é O(n) e a complexidade do espaço é de duas variáveis, ou seja, O(2) =

O(1).

Subproblemas

Os subproblemas sobrepostos no algoritmo de Kadane são:

`dp = max(sequencia[i], dp + sequencia[i])`

`maximo = max(dp, maximo)`

dp é avaliado duas vezes para cada i. O algoritmo de Kadane memoriza isso, transformando-o de O(n2) para O(n)

Recorrência: dp = max(sequencia[i], dp + sequencia[i])

Tipo de solução: Bottom up