Lista 3 - Fábio Alves de Freitas

```
import numpy as np
```

Utils 1º)

4 células ocultas

1º)

Dê o pseudo-código para reconstruir uma LCS (sequência comum mais longa possível) a partir de tabela c já computada e as sequências originais $X =< x_1, x_2, \ldots, x_m >$ e $Y =< y_1, y_2, \ldots, y_n >$ com um custo de tempo de $O(m + n)$, sem utilizar a tabela b.

resposta:

```
def print_lcs_new_aux(X, Y, c, i, j):
  if c[i, j] == 0:
    return None
  if X[i-1] == Y[j-1]:
    print_lcs_new_aux(X, Y, c, i-1, j-1)
    print(X[i-1],end='')
  elif c[i-1, j] >= c[i, j-1]:
    print_lcs_new_aux(X, Y, c, i-1, j)
  else:
    print_lcs_new_aux(X, Y, c, i, j-1)

def print_lcs_new(X,Y,c):
  i = len(X)
  j = len(Y)
  print_lcs_new_aux(X,Y,c,i,j)


x = list('ABCBDAB')
y = list('BDCABA')
c, b = lcs_length(x, y)

print('matriz c:')
print_memoization_lcs(x, y, c)
print('\nlongest common sequence:')
print_lcs_new(x, y, c)
```

```
     matriz c:
                  B       D       C       A       B       A
            0       0       0       0       0       0       0
```

```
A       0       0       0       0       1       1       1
B       0       1       1       1       1       2       2
C       0       1       1       2       2       2       2
B       0       1       1       2       2       3       3
D       0       1       2       2       2       3       3
A       0       1       2       2       3       3       4
B       0       1       2       2       3       4       4

longest common sequence:
BCBA
```

Utils 2º)

[ ] ↳ *2 células ocultas*

2º)

Considere uma modificação no problema de seleção de atividades já vistos. Agora, cada atividade a_i tem, em adição ao seu tempo de início e tempo de término, um valor monetário de v_i. O objetivo agora não é mais maximizar o número de atividades agendadas, mas sim maximizar o valor monetário das atividades agendadas. Ou seja, deseja-se encontrar um conjunto A de atividades compativeis tal que o somatóro abaixo seja maximizado. Dê um algoritmo com custo polinomial no tempo.

$$\sum_{a_k \in A} v_k$$

resposta:

```
# Dado um vetor de atividades de tamanho i busco o
# indice da primeira atividade que se encaixa com a
# última atividade desse vetor se não houver uma,
# retorno -1, se sim retoro seu indice
def latest_non_conflict(s, f, i):
  for j in reversed(range(i)):
    if f[j] <= s[i-1]:
      return j
  return -1



def activity_selector_max_profit_aux(s, f, v, n, m):
  # caso base
  if n == 1:
    return v[n-1]
  # memoization - verifica se subproblema ja foi resolvido
  elif m[n-1] > float('-inf'):
    return m[n-1]
  # caso geral do problema
```

```
  # caso geral do problema
  else:
    # verificando valor final com o valor da atividade n-1
    profit1 = v[n-1]
    i = latest_non_conflict(s, f, n)
    if i != -1:
      profit1 = profit1 + activity_selector_max_profit_aux(s, f, v, i+1, m)

    # verificando valor final sem o valor da atividade n-1
    profit2 = activity_selector_max_profit_aux(s, f, v, n-1, m)

    # adiciona maior caso no memoization e o retorna
    m[n-1] = max(profit1, profit2)
    return m[n-1]
```

```
def activity_selector_max_profit(start, finish, value):
  num_activities = len(start)
  memoization = np.full(num_activities, float('-inf'))
  memoization[0] = value[0]
  return activity_selector_max_profit_aux(start, finish, value, num_activities, memoizatio
```

```
# assume-se que as atividades estao ordenadas pelo tempo de finalização
start =  [1, 3, 0, 5, 3, 5, 6, 8, 8, 2, 12]
finish = [4, 5, 6, 7, 9, 9, 10, 11, 12, 14, 16]
value =  [4, 5, 5, 3, 5, 6, 6, 7, 9, 9, 0]

print('max: ')
print(activity_selector_max_profit(start, finish, value))
```

```
max:
17.0
```

3º)

Suponha que lhe são dados dois conjuntos A e B, cada qual contendo n inteiros positivos. Você pode escolher reordenar os conjuntos de qualquer forma. Após o reordenamento, seja a_i o i-ésimo elemento do conjunto A, e seja b_i o i-ésimo elemento do conjunto B. Você irá receber um pagamento dado ela produtório abaixo. Dê um algorimto que maximiza o seu pagamento e mostre qual é o seu custo em tempo.

$$\prod_{i=1}^{n} a_i^{b_i}$$

resposta:

Para maximizar o pagamento, devemos calcular a exponenciação com ambos os arrays ordenados de corma crescente, de modo que sempre teremos as maiores bases elevadas aos maiores expoentes. Para o caso de existir 0 na lista A, então o custo do algoritmo será $\Omega(n)$. Caso contrário, devemos ordenar os dois arrays e em seguida realizar o cálculo do produtório. Esse custo pode ser calculado somando os custos do caso $else$, descrito no algoritmo abaixo.

$$O(n * log(n) + n * log(n) + n) = O(n * log(n))$$

Desta forma, o custo final será de:

$$\text{Custo} = \begin{cases} \Omega(n), \text{ se } 0 \in A \\ O(n * log(n)) \end{cases}$$

```
def calcular_salario(A, B):
  if 0 in A:                          # n          custo de buscar o 0 num array de tam
    return 0                          # 1          constante
  else:                               # 1          constante
    A = np.sort(A, kind='mergesort')  # n*log(n)   custo do mergesort é n*log(n)
    B = np.sort(B, kind='mergesort')  # n*log(n)   custo do mergesort é n*log(n)
    total = 1                         # 1          constante
    for i in range(len(A)):           # n          loop com as n iterações do produtóri
      total = total * A[i]**B[i]      # 1          constante
    return total                      # 1          constante
```

```
num = 6
A = np.random.randint(low=1,high=num+1, size=num) # lista de números aleatorios entre 1 e 
B = np.random.randint(low=1,high=num+1, size=num) # lista de números aleatorios entre 1 e 
print("A = {}\nB = {}\nTotal = {}".format(A,B,calcular_salario(A,B)))
```

```
A = [3 2 5 2 5 5]
B = [2 1 1 4 3 3]
Total = 351562500
```

✓ 0s conclusão: 20:08