Problem set 3
Esercizio 1

Alessandro Straziota

Esercizio 1. Si consideri il seguente algoritmo *greedy* per il problema *chain matrix multiplication*

Algorithm 1 `Alg`$(M_1, \dots, M_n)$

INPUT: Matrici $M_1, \dots, M_n$, dove M_i è una matrice $a_{i-1} \times a_i$, con $a_0, a_1, \dots, a_n \in \mathbb{N}$
OUTPUT: La matrice prodotto $M = M_1 \times \dots \times M_n$

```
if n = 2 then
    return M_1 × M_2
k = arg min{a_{i-1} a_i a_{i+1} : i = 1, ..., n − 1}
P = M_k × M_{k+1}
return Alg(M_1, ..., M_{k-1}, P, M_{k+2}, ..., M_n)
```

---

1. Fornire un controesempio che mostri come il numero totale di moltiplicazioni numeriche eseguite dall'algoritmo per calcolare $M_1 \times \dots \times M_n$ non è ottimo.
2. Implementare `Alg` e l'algoritmo di programmazione dinamica visto a lezione (si veda il Cap. 6.5 in [1]) in un linguaggio di programmazione a piacere.
3. Scrivere un programma che prenda in input n numeri $a_0, a_1, \dots, a_n \in \mathbb{N}$ e calcoli quante moltiplicazioni farebbe l'algoritmo greedy *Alg* e quante ne farebbe l'algoritmo di programmazione dinamica, con input matrici M_i di dimensioni $a_{i-1} \times a_i$, per $i = 1, \dots, n$.
4. Eseguire più volte il programma del punto precedente con numeri $a_0, a_1, \dots, a_n \in \mathbb{N}$ scelti a caso e valutare in media quante operazioni si risparmiano con l'algoritmo di programmazione dinamica rispetto all'algoritmo greedy.

Punto 1 Un controesempio in cui l'algoritmo greedy appena descritto è il seguente:

$$\underset{(3\times 4)}{A} \times \underset{(4\times 6)}{B} \times \underset{(6\times 5)}{C} \times \underset{(5\times 3)}{D}$$

L'algoritmo greedy eseguirebbe le moltiplicazioni in questa sequenza

$$\big((A \times B) \times C\big) \times D$$

con un numero complessivo di moltiplicazioni pari a

$$\begin{aligned} X\Big(\big((A \times B) \times C\big) \times D\Big) &= (3 \times 4 \times 6) + X\big((AB \times C) \times D\big) \\ &= 72 + (3 \times 6 \times 5) + X\big(ABC \times D\big) \\ &= 72 + 90 + (3 \times 5 \times 3) \\ &= 162 + 45 = 207 \end{aligned} \tag{1}$$

Invece una sequenza che consente di effettuare un minor numero di moltiplicazioni è la seguente

$$A \times \big(B \times (C \times D)\big)$$

con un numero complessivo di moltiplicazioni pari a

$$\begin{aligned} X\Big(A \times \big(B \times (C \times D)\big)\Big) &= (6 \times 5 \times 3) + X\Big(A \times \big(B \times CD\big)\Big) \\ &= 90 + (4 \times 6 \times 3) + X\big(A \times BCD\big) \\ &= 90 + 72 + (3 \times 4 \times 3) \\ &= 162 + 36 = 198 \end{aligned} \tag{2}$$

Punto 2 I due algoritmi sono stati implementati in linguaggio Python
Greedy Alg.

```python
import numpy as np

def GREEDY_CHAIN_MULT(A):

    if len(A) == 2:
        return A[0]*A[1]

    k = min( list( map( lambda k:
   (A[k].shape[0]*A[k].shape[1]*A[k+1].shape[1],k),
   list(range(len(A)-1)) ) ), key=lambda tup: tup[0])[1]

    P = A[k]*A[k+1]
    A[k] = P
    A.pop(k+1)

    return GREEDY_CHAIN_MULT(A)
```

Dynamic Alg.

```
import numpy as np

def OPT(A):

    n = len(A)
    C = [ [0]*n for _ in range(n) ]

    for s in range(1,n):
        for i in range(n-s):
            j = i + s
            C[i][j] = min( list(map(lambda k: A[i].shape[0]
    * A[k].shape[1] * A[j].shape[1] + C[i][k] + C[k+1][j],
    list(range(i,j)))) )

    return C

def CHAIN_MULT(A):

    global O
    O = OPT(A)

    return procedura_ricorsiva(A,0,len(A)-1)

def procedura_ricorsiva(A,i,j):

    if i == j:
        return A[i]

    k = i
    while O[i][j] !=
    A[i].shape[0]*A[k].shape[1]*A[j].shape[1] + O[i][k] +
    O[k+1][j]:
        k += 1

    X = procedura_ricorsiva(A,i,k)
    Y = procedura_ricorsiva(A,k+1,j)

    return X*Y
```

Punto 3 Il seguente programma è stato scritto in linguaggio Python

```python
from copy import deepcopy

'''
a deve essere una lista di interi positivi. Verranno eseguiti dei controlli per verificare se a
rappresenta effettivamente le dimensioni di una sequenza ordinata di matrici
'''
def COUNT_MULT(a):

    if len(a)%2 != 0:
        raise Exception("Il numero di elementi nella sequnza deve essere necessariamente pari")

    dim = []
    for i in range(0,len(a),2):
        dim.append((a[i],a[i+1]))


    for i in range(len(dim)-1):
        if dim[i][1] != dim[i+1][0]:
            raise Exception("Dimensioni incoerenti")

    greedy = GREEDY_COST(deepcopy(dim))
    dynamic = DYNAMIC_COST(deepcopy(dim))

    #print("GREEDY : " + str(greedy))
    #print("DYNAMIC : " + str(dynamic))

    return (greedy, dynamic)


def GREEDY_COST(a):

    if len(a) == 2:
        return a[0][0]*a[0][1]*a[1][1]

    k = min( list( map( lambda k: (a[k][0]*a[k][1]*a[k+1][1],k), list(range(len(a)-1)) ) ), key=lambda tup: tup[0])[1]

    ret = a[k][0]*a[k][1]*a[k+1][1]

    a[k] = (a[k][0],a[k+1][1])
```

```
    a.pop(k+1)

    return ret + GREEDY_COST(a)


def DYNAMIC_COST(a):

    n = len(a)
    C = [ [0]*n for _ in range(n) ]

    for s in range(1,n):
        for i in range(n-s):
            j = i + s
            C[i][j] = min( list(map(lambda k: a[i][0] *
    a[k][1] * a[j][1] + C[i][k] + C[k+1][j],
    list(range(i,j)))) )

    return C[0][len(a)-1]
```

Punto 4 In seguito verranno riportati i risultati di diversi test. Ogni test esegue m moltiplicazioni di catene di matrici, dove ogni catena è lunga n

m	n	*dynamic/greedy*
1000	3	1.0
10000	3	1.0
20000	3	0.9634330537304684
1000	20	0.9992772021914211
10000	20	0.9999540240736369
20000	20	0.8550020432137697
200	100	0.815460755557939
1000	100	0.8160108822623396
10000	100	0.8153916921267482
20000	100	0.8207176372556815
1000	500	0.8160108822623396