Mestrado em Matemática e Aplicações

Especialização em Matemática Financeira

2019/2020, 1st semester

Computational Methods

Test #1 – 20 December 2019

Duration: 2 hours

Close Book (no consulting materials are allowed)

Student nº _____ Name: _____

1. (1 pt) What is the value of variable **x** at the end of the following program

```
i = 1;
x = 0;
while x < 25:
   x = x + i;
   i = 2 * i;
```

Answer: x = 31

2. (1 pt) What integer value should **k** take so that, at the end of the program below, variable **s** takes value **13**.

```
M = [[2, -4, k], [-2, k, 6], [3, -1, k]]
s = 0;
for i in range(len(M)):
   for j in range(len(M[0])):
        s = s + M[i][j]
```

Answer: k = 3

3. (1 pt) After running the sequence of instructions below, what is the value of the variable **s**?

```
M = [[2,3,7,9,6,0], [1,5,4,5,1,3]]
s = 0
for i = range(len(M[0])):
   s = s + (M[0][i]- M[1][i])**2
s = math.sqrt(s)
```

Answer: x = 8

4. **(1.5 pt)** Given the text file with name **`xpto.txt`** containing the text below

This is a file with 3 lines
This line is the second of these lines.
And this is the last line.

what is the value returned by the call **`check_chars ("xpto.txt","a", 3)`**?

```
def check_chars(fname, ch, k):
    "documentation omitted"
    c = 0
    fid = open(fname, "r")
    lines = fid.readlines()
    fid.close()
    for line in lines:
        for x in line:
            if x == ch:
                c = c + 1;
    return c < k
```

Answer: **`True`**

5. **(1.5 pt)** What is the approximate value that you expect from the execution of the function below when the call **`dice_extremes(900)`** is made.

```
def dice_extremes(n):
    c = 0;
    for k in range(n):
        d = 1 + math.floor(6*random.random())
        if d == 1 or d == 6:
            c = c + 1;
    return c
```

Answer: **`300`**

6. **(1.5 pt)** What is the final value **`Q`** computed by the program below?

```
Q = []
for i in range(1,4):
    L = []
    for j in range(1,6):
        L.append(i*j)
    Q.append(L)
```

Answer: **`Q =`**

```
[
 [1, 2, 3, 4, 5],
 [2, 4, 6, 8,10],
 [3, 6, 9,12,15]
]
```

7. **(2 pt)** Complete the specification of the function below (including its documentation) so that, given a list of lists LL, it returns a pair (L, P), where L is the index of the list that contains the smallest element, and P its position in that list. For example, for

```
LL = [[4,9,7,2], [3,5,0], [1,6], [8]]
```

The function should return pair `(1,2)` since the smallest element (`0`) is in position `2` of list `1`.

Note 1: The position assumes that in Python the first element of a list has index `0`.

Note 2: in case where more than one element is the smallest, the function may return the position of any of these elements.

`def position_smallest(LL):`

```
    """Returns the indices of the smallest element of
    the list of lists LL"""
    lowest = math.inf
    for i in range(len(LL)):
        for j in range(len(LL[i])):
            if LL[i][j] < lowest:
                lowest = LL[i][j]
                (p,q) = (i,j)
    return (p,q)
```

8. **(2 pt)** Complete the specification of the function below (including its documentation) so that it returns the number of times an element of list L is greater than the previous one. For example, for **`L = [1,3,2,6,4,9,5,8]`**, the function should return **`4`** (as **`3>1, 6>2 , 9>4, 8>5`**).

`def increasing(L):`

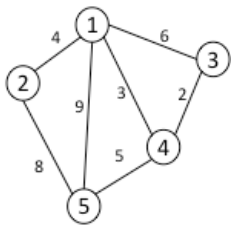
```
    """counts the number of times an element is greater then the
    previous element in list L"""
    c = 0
    for i in range(len(L)-1):
        if L[i+1]> L[i]:
            c = c + 1
    return c
```

9. **(2.0 pt)** As you will recognise, for a list of numbers given as input, the function below returns the list of numbers sorted in increasing order (by the bubble sort algorithm).

```
def bubble_sort(V):
    """ returns the (bubble) sorted version of vector V"""
    n = len(V)
    for k in range(n-1,0,-1):
        for i in range(0,k):
            if V[i] > V[i+1]:
                x = V[i]
                V[i] = V[i+1]
                V[i+1] = x
    return V
```

graph_x.txt

```
5; 7
1; 2; 4
1; 3; 6
1; 4; 3
1; 5; 9
2; 5; 8
3; 4; 2
4; 5; 5
```



Adapt the above algorithm to sort the list of arcs forgiven as input, but in decreasing order of their weights. For example, for the graph shown the function may get as input the list

```
A = [(1,2,4), (1,3,6), (1,4,3), (1,5,9), (2,5,8), (3,4,2), (4,5,5)]
```

(the arcs may be presented in a different order) and should return list

```
[(1,5,9), (2,5,8), (1,3,6), (4,5,5), (1,2,4), (1,4,3), (3,4,2)]
```

```
def sorted_ arcs(A):
    """Returns a graph's arcs A in decreasing order of their weights"""
```

```
    n = len(A)
    for k in range(n-1,0,-1
        for i in range(0,k):
            if A[i][2] < A[i+1][2]:
                x = A[i]
                A[i] = A[i+1]
                A[i+1] = x
    return A
```

10. (3.5 pt) In many cases, a random choice is made to assign an element of a group of people for some task, taking into account its "weigth". Assume that you are given a list of triples (age, name, weigth). Given such a list, specify a function that returns the first and last name of an element chosen randomly, but proportionally to its weigth. For example, for the list of triples

```
[ (45, "Ana Marta da Silva Lopes", 8),
  (23, "Pedro Miguel Costa Matos", 4),
  (39, "Clara Isabel Campos e Cunha Teles", 1),
  (56, "Carlos Jorge Santos Abreu Pinto",2)
]
```

the possible names returned are

"Ana Lopes", **"Pedro Matos"**, **"Clara Teles"** or **"Carlos Pinto"**,

where the first is returned with twice the probability of the second, which is returned with twice the probability of the fourth, which is returned with twice the probability of the third.

Note: Alternative, return any of them with equal probability (with a penalty of 1.5 pts).

```
def random_choice(L):
    """Returns the first and last name of a randomly chosen triple of T"""
    n = len(L)
    t = 0
    for i in range(n):
        t = t + L[i][2]
    # create a list of probability bounds
    s = 0
    P = []
    for i in range(n):
        s = s + L[i][2]
        P.append(s)

    # get a random number within the bounds
    r = t * random.random()

    # check in which bounds r lies
    i = 0
    while r > P[i]:
        i = i + 1

    # obtain the names from the selected person
    name = L[i][1]
    names = name.split(" ")
    return names[0] +" "+ names[-1]
```

11. Assume that you are given a set of locations in a village to be connected by some water supply. The locations may be connected through different streets, and the cost of these connections are represented by an undirected weighted graph, specified in DIMACS format in a file with name **`fname`**. Unfortunately, there is an extra node not included in the file, but whose direct distances to all the other nodes are given in a list **`D`**, and must be considered for the water distribution infrastructure.

In these conditions, specify a function that returns a triple **`(T, p, d)`** where **`T`** is the adjacency matrix of the (extended) graph that contains only the arcs to be considered to the infrastructure, and **`d`** represents the distance (according to the adopted infrastructure **`T`**) between the extra node and some node **`p`** that is farthest apart from the extra node.

- **Suggestion 1**: You may use auxiliary functions (e.g. those in the annex).
- **Suggestion 2**: Specify an auxiliary function, **`add_node(G,D)`**, that returns the original adjacency matrix **`G`** extended with a node whose distances to the other nodes are in list **`D`**. (Note that you may use this function even if you do not implement it - penalty 1 pt)

```
def cheapest_water_distribution (G):
    """ returns the cheapest water distribution facility, when a node
    is added with distances D to all nodes of the original graph G as
    well as the distance to the farthest node """
```

```
    G = dimacs_read(fname)
    R = add_node(G,D)
    T = prim(R)
    S = floyd(R)
    farther = 0
    n = len(R)
    for i in range(n):
        if S[n-1][i] > farther:
            farther, q) = (S[n-1][i], i)
    return (T, (q, farther))



def add_node(G,D):
    """ adds a node to a graph, represented by its adkjacency
    matrix G, where the distances to all the graph's nodes are
    indicated in list D"""
    E = D.copy()
    n = len(G)
    H = [G[i] for i in range(n)]
    for i in range(n):
        H[i].append(E[i])
    E.append(0)
    H.append(E)
    return H
```

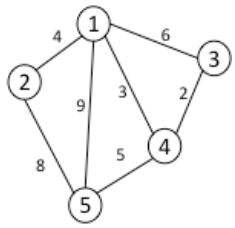
Annex

In question 11, you should consider the functions that were studied in the classes regarding weighted undirected graphs where the weights may be interpreted as distances between vertices of the graph. If no edge exists between two vertices of a graph G = <V, E>, a virtual edge with value -1 is assumed in the adjacency matrix G of the graph.

- **`def dimacs_read(filename)`**
  - returns the adjacency matrix, **`G`**, of a weighted undirected graph specified in file with **`filename`**. The first line of the graph contains the number of vertices and arcs, and the subsequent lines the triple **`<n1, n2, w>`**, where **`w`** is the weight of the edge connecting the vertices **`n1`** and **`n2`** (**`n1 < n2`**). The integers **`n1`**, **`n1`** and **`w`** are separated by semicolons (“**`;`**”).

graph_x.txt

```
5; 7
1; 2; 4
1; 3; 6
1; 4; 3
1; 5; 9
2; 5; 8
3; 4; 2
4; 5; 5
```




```
G =
   0    4    6    3    9
   4    0   -1   -1    8
   6   -1    0    2   -1
   3   -1    2    0    5
   9    8   -1    5    0
```

- **`def write_graph(G, filename)`**
  - writes a graph, given by its adjacency matrix **`G`**, in file with name **`filename`**, with the format specified above.

- **`def prim(G)`**
  - Given an undirected weighted graph **`G`**, it returns a minimum spanning tree, **`T`**. Both **`T`** and **`G`** are represented by the corresponding adjacency matrices. For the graph above, **`prim(G)`** returns

```
T =
   0    4   -1    3   -1
   4    0   -1   -1   -1
  -1   -1    0    2   -1
   3   -1    2    0    5
  -1   -1   -1    5    0
```

- **`def floyd(G)`**
  - returns a matrix **`S`** with the shortest distances between any two vertices of the graph specified by its adjacency matrix, **`G`**. For the graph above **`floyd(G)`** returns

```
S =
   0    4    5    3    8
   4    0    9    7    8
   5    9    0    2    7
   3    7    2    0    5
   8    8    7    5    0
```