Homework 1

Algorithm Design
Università di Roma "La Sapienza"

Francesco Bovi 1803762
George Ciprian Telinoiu 1796513

5 dicembre 2020

1 Exercise 1

1.1

```
s_rev=reverse(s)                    #compute the reverse of the original string
for i in range(0,length(s)):        #iterate over s
    aux=""
    count=0
    for j in range(i,length(s)):    #consider sub-strings of s
        aux+=s[i]                   #construct partial sub-strings of s
        if(aux==s_rev[length(s)-1-j:length-i]): count+=1
            if(count>max_length):
                max_length=count    #max_lenght  to be initialized
                max_palindr=aux     #max_palindr to be initialized
return max_palindr
```

The cost of the algorithm is $O(n)+O(n^2)$ since we initially compute the reverse of the input string and then we use 2 nested FOR loops in order to compare them. Since the $O(n^2)$ dominates over $O(n)$, we have indeed a time cost of $O(n^2)$

1.2

```
getlength(input_string,0,N-1,matrix):           #matrix to be initialized to NxN
    if(i==j): matrix[i][j]=1  return 1          #base case
    if(i>j) :                 return 0
    if(matrix[i][j]==0):
        if(input_string[i]==input_string[j]):
            matrix[i][j]=2 + getlength(input_string,i+1,j-1,matrix)
        else:
            matrix[i][j]=max(getlength(input_string,i+1,j  ,matrix),
                             getlength(input_string,i  ,j-1,matrix))
    return matrix[i][j]

getstring(input_string,0,N-1,matrix,max_palindr):#matrix to be initialized to NxN
    if(matrix[i][j]==1): return input_string[i]  #base case
    if(x>j):             return ""               #to ignore empty part of matrix
    if(s[i]==s[j]):
        max_palindr+=s[i]+getstring(input_string,i+1,j-1,matrix,max_palindr)+s[j]
    elif(matrix[i+1][j]>=matrix[i][j-1]):
        max_palindr+=getstring(input_string,i+1,j  ,matrix,max_palindr)
    else:
        max_palindr+=getstring(input_string,i  ,j-1,matrix,max_palindr)
    return max_palindr
```

The costs of our functions are as following: getlenght() being that it calls itself until it finds 2 matching characters, in the worst case, that being a string that has all different characters, will have to do so for all possible sub-strings, therefore filling the matrix with values of 1. This results in a time complexity of $O(n^2)$. On the other hand, getstring() at worst will have to traverse the whole column or row of the matrix, therefore returning a running time of $O(n)$.

2 Exercise 2

Given a set of investors and founders with a list of good pairs (I,F), define a network flow N whose max-flow contains only good pairs. Given this input, construct a network flow N such that:

1. Create a node for each investor and denote it with I_i;
2. Create a forward edge (SI_{edge}) from the source **s** to every investor's node (I_i) and set it's capacity to 2;
3. Create a node for each founder and denote it with F_i;
4. Create a forward edge (IF_{edge}) from I_i to F_i for each element of P (I_i, F_i) and set it's capacity to 1. After exhausting all elements of P, create a forward edge (IF_{edge}) from I_{n-i} to all remaining F_{n-i} in order to have a fully connected sub-graph;
5. Create a forward edge (FT_{edge}) from every F_i to the sink and set it's capacity to 2.

The constructed network flow N will have as solution a max-flow iff all of these conditions are met (to find it we could run the Ford-Fulkerson until no augmenting path is possible):

- **2|I|=2|F|**: Because we know that each investor must have 2 founders as neighbours (and vice versa), so with a number of founders greater than the number of investors is not possible respect such restriction. The number of founders and investors must be at least 2 because we know that in each table there must be at least 4 people.
- **Capacity of SI_{edge} is 2**: Because each investor can have at most 2 neighbours, in fact setting the capacity to more than 2 could let the algorithm assign more than 2 preferences for some investor, contradicting the fact that it can have at most 2 founders as neighbours.
- **Capacity FT_{edge} is 2**: Because each founder can be chosen at most by 2 investors, so with a capacity higher than 2 a founder could chosen by more than 2 investors, negating the neighbours restriction;
- **Capacity IF_{edge} is 1**: Because each founder can be chosen only once by a specific investor, otherwise it would result that a single founder is both the left and the right neighbours of a certain investor.
- **$|IF_{edge}|$ must be at least 2 for each investor**: Given the previous point, it wouldn't be possible to respect the neighbours restriction, since with less than 2 edges the algorithm could never assign 2 founders as neighbours.
- **$|IF_{edge}|$ must be at least 2 for each founder**: Following again from the previous point, if it would have less than 2 edges, it could never be the neighbour of at least 2 investors.

Following all the conditions and the steps reported above, we can clearly see that in the network flow N there are 2 sub-graphs: one composed by all the pairs in P and the other (fully connected) with the remaining investors and founders. This way we can construct at least 2 different tables, the former respecting all the preferences in P, and the latter, being that it just needs to respect the neighbours restriction, will be composed by any arrangement of the remaining guests. In this configuration therefore we always satisfy (if all conditions are met) at least a table arrangement with *only* good pairs if max-flow is returned. If that kind of arrangement would need to be the only one, we would just exclude all the vertices out of P, having a Network Flow N that still solves the problem with max-flow.

3 Exercise 3

Given an initial credit score C, a set of projects P:[*((Cp1, Bp1),...(Cpn, Bpn))*], the following algorithm returns if all projects can be done in $O(n \cdot logn)$.

```
for i in projects:
    if (i[1] is positive):
        positive_project.add(i) #local variable positive_projects = {}
    else:
        negative_projects[0].add(i) #local variable negative_project = {}
        negative_projects[1].add(i[0] - |i[1]|) #(Cpi - |Bpi|)
sorting.merge(positive_projects) #based on Cp
for i in positive_projects:
    if (C >= i[0]):
        C+ = i[1]
    else:
        exit()
(sorting.merge(negative_projects[1])).reverse() #based on Cpi - |Bpi|
for i in negative_project[0]:
    if(C >= i[0]):
        C+ = i[1]
    else:
        exit()
```

Our algorithm takes $3 \cdot O(n) + 2 \cdot O(n \cdot logn)$; respectively *O(n)* is referred to the iterations that we have to do in order to select projects with positive and negative values of Bp, plus the 2 iterations that we do in order to work on each project. On the other hand $O(n \cdot logn)$ is due to the application of *merge sort*. Being that $O(n \cdot logn)$ dominates over *O(n)* and we don't consider constants, we have indeed a running time that is $O(n \cdot logn)$.

To prove it's correctness, we analyze the 2 sub-vectors individually:

1. Having ordered the positive sub-vector, we know for sure that the first project is the one with the lowest Cp required. If that project can't be done, which means that $C < Cp_i$, for sure not even the next one can be attempted being that we know that $Cp_i \geq Cp_{i+1}$. Based on those considerations, we will be able to do all positive projects iff we reach the end of the sub-vector. Negative Projects should not be considered at this time.

2. After exhausting the positive vector, our final C is for sure at his maximum. To proceed we need to have $C \geq Cp_i$, with Cp_i being the Cp linked to the highest number in negative_projects[1]. If $C \geq Cp_i$ is TRUE, then also $C + Bp_i \geq Cp_i + Bp_i$ must be TRUE, given that Bp_i is surely negative and that we can always assume that $Cp_i + Bp_i \geq 0$.
   But we also know that $Cp_i + Bp_i$ is greater than $Cp_{i+1} + Bp_{i+1}$ given negative_projects[1] order, and that we can proceed iff $C + Bp_i \geq Cp_{i+1}$. Given that to exhaust all projects, we must always satisfy this condition, we'll know that if we reached the end with a positive score, we for sure have completed all projects in P.

4 Exercise 4

4.1

```
d=d_init/2                          #start considering d_init/2
cures=[c1,...,cn]
ub=d_init                           #initial upper bound is known to be d_init
lb=1                                #lowest possible bound could be 1
while((ub-lb)!=1):                  #while we have possible values to test
    check=0
    for index in len(list_cure):    #we need to test every cure
        result=test(cures[index],d) #test i-th cure with dose 'd'
        if(result==OK):
            check=1
            ub=d                    #set new upper bound since test OK
    if(check==1):                   #focus only on lower values of d
        d=d-((ub-lb)/2)             #decrease d based on current up & lb
    else:                           #focus only on higher values of d
        lb=d                        #set new lower bound since all test not OK
        d=d+((ub-lb)/2)             #increase d based on current up & lb
return best_cure
```

The cost of the algorithm is $O(nlogd)$ because the outer while cycle is executed $logd$ times and an inner for loop that tests all n cures for each dose d proposed. $Logd$ is guaranteed by the fact that we iteratively split the search space in half until we isolate one item, mathematically it results: $d/2^x = 1$ and so $x = logd$, that is indeed the number of iteration.

4.2

Compared to 4.1, this time we must find a way to reduce the number of tested cures c_i, while also keeping low the number of doses a_i considered. We start from considering all possible cures c_i which we know will be successful if tested with our maximum d_init dose. Assuming that if we pick a random c_i cure, the probability that we've hit a cure with minimum a_i is at least 1/n, we don't have a deterministic way to identify a superior set of cures. Therefore we could split the set of possible cures c_i in half and randomly select one of them, for example by tossing a coin. After we have selected a set, we will perform the test on all of them in order to identify the positive ones to the value *d/2*. Now we find ourselves in the same situation as before, but with a remaining number of tests to perform equal to C' = n/2 - (negative_c_i). We should underline at this moment that the only way to know if the best cure is in the other half would be to get a negative result on all tested cures , which is definitely unlikely when working under average assumptions. Therefore we should continue to apply the same process to the new set C', until we are able to isolate at most 2 cures c_i and c_j with a certain a_i. Let's say we choose c_i. If testing with a_i is positive, we need to keep lowering a_i for c_i until we reach 2 possible outcomes: testing with $a_i = 1$ is positive, in that case we have finished, or testing with a_i gives back a negative result. If negative, we move testing a_i on c_j. The same 2 previous outcomes are presented to us at this moment, so we either succeed with 1 or we need to move to a new set of cures. Given the analysis based on the average case, we should be able to find our optimum between those 2 cures, since we would keep on . If we would want to extend to a worst-case analysis, we know for sure that this algorithm would return a solution since all we would have to do is try the first value a_i that failed for c_j on the first subset that we had excluded and follow the exact same pattern as we did here, in that case though we would have the guarantee that if we reach only 1 cure c_i, that would be the best one.