HOMEWORK 1 Students VALERIO VALENTE 1894954 - GABRIELE DE STRADIS 1947850

To solve 1.1 and 1.2 we adopted a dynamic programming approach building a 2-dim array which helped us to not exceed the running time. In 1.1 we also use a matrix P in which storing the various palindrome substrings, whereas in the 1.2 we created the subsequence step by step through the algorithm.

Exercise 1.1

```
LongestPalindromeSubstring(W,n)  // n is the length of the word
P is another 2-dimensional array in which we store the palindromic substrings collected
Initialize P[1..n,1..n]
Palindrome(1,n,P,W)
max = 0
For i=0 to n
        For j=0 to n
                If P[i][j] is not empty and P[i][j].length > max
                        max = P[i][j].length
                        word = P[i][j]
Return word

Palindrome(i,j,P,W)
n = W.length
Initialize M[1..n,1..n]
If M[i][j] is empty then
        If i>j then return 0   //if this situation happens we "let win" the other element into
                               //the max function because the substring is over
        If i=j then
                M[i][j] = 1
        Else If w[i] == w[j] then
                M[i][j] = j-i+1
                P[i][j] = w[i] to w[j]
        Else
                M[i][j] = max{ Palindrome(i,j-1,P), Palindrome(i+1,j,P)}
Return M[i][j]
```

Giving as input i = 1 and j = n you will get the longest palindrome substring. We have to fill the M and P n x n table simultaneously, hence we have $O(n^2)$. In the end, we iterate over P searching the longest word $\rightarrow O(n^2) + O(n^2) = O(n^2)$

Exercise 1.2

```
LongestPalindromeSubsequence(W, n)
Initialize M[1..n,1..n] and S[] //S is a list which will contain the indices of the substring
added = True
For i = 0 to n
        M[i][i] = 1
For sub_len = 2 to n+1 //sub_len is the length of the considered subsrtring at the current iter.
        For i = 0 to n+sub_len-1
                j = i + cl -1
                If w[i] == w[j] and sub_len == 2
                        M[i][j] = 2
                        S.append(i)
                        S.append(j)
                Else If seq[i] == seq[j]
                        L[i][j] = L[i+1][j-1] + 2
                                If L[i][j] > len(S)
                                        If i not in S
                                                S.append(i)
                                        If j not in S
                                                S.append(j)
                                        If i+1 not in S //consider this only when sub_len is even
                                                S.append(i+1)
                                Else If sub_len.isOdd()
                                        If i+1 not in S  and not added //condider this once and only when
                                                                       //sub_len is odd
                                                S.append(i+1)
                                                added = True
                Else
                        L[i][j] = max(L[i][j-1], L[i+1][j])

S.sort()
Return S
```

The complexity is given by the number of instances needed to fill the matrix, at most $O(n^2)$. We need a loop at the end to iterate over the indices and print the substring but this won't affect the runtime, resulting $O(n^2) + O(n) = O(n)$

HOMEWORK 1 Students VALERIO VALENTE 1894954 - GABRIELE DE STRADIS 1947850

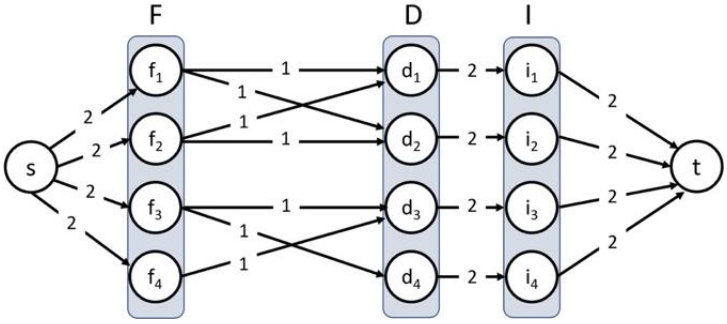
Exercise 2

In order to model the problem, we developed a graph based on the bipartite matching problem, which can be solved as a maximum flow problem. Therefore we decided to design the following set of nodes F, D, I U {s,t} = V. F represents all the founders, I represents all the investors, D is a particular subset of nodes which helped us to model the following constraints:

- Let i_k be an investor, he can have at most only two founder f_i and f_j seated close to him;
- Let f_k be a founder, he can have at most only two investor i_i and i_j seated close to him.

So we got an edge (f_i,d_j) when a founder wants to be seated close to an investor i_j.
The founder can seat close to such investor iff the edge (d_i,i_i) is not already filled, which means that the investor i_i has the two seats close to him occupied.
Each edge (s,f_i) has capacity 2, allowing a founder to seat by at most 2 investors.
Each edge (i_i,t) has capacity 2 which means that 3 people are willing to seat together.

To show an example, given the list $P \subseteq I \times F$ of good pairs (i_1,f_1), (i_1,f_2), (i_2,f_1), (i_2,f_2), (i_3,f_3), (i_3,f_4), (i_4,f_3), this is the resulting flow network N.



The numbers on each edge represents the maximum capacity, which helped us to model the fact of having at least three people seated at one table respecting the above constraints. Exploring this network with a maximum flow algorithm will solve the problem of respecting as much good pairings as possible in finding a seating arrangement.

PROOF OF CORRECTNESS

Def: Given an undirected graph *G = (V, E)* a subset of edges $M \subseteq E$ is a matching if each node of *F* and *D* appears in at most two edges in *M*.
We want to prove that the max cardinality of the matching in *G* is equal to the max flow in *G'* (the directed graph designed as shown above)

Proof ($\Leftarrow$)

- Given a max matching *M* of cardinality *k*.
- Consider flow *f* that sends 1 unit along each of *k* paths (because 1 is the bottleneck).
- *f* is a flow, and has value *k*.

Proof ($\Rightarrow$)

- Let *f* be the max flow in *G'* of value *k*
- From the *Integrality Theorem* we know that *k* is integral and $\forall\ e \in E$ we can either have $f(e) = 0, 1$ or 2
- Consider *M* as the set of edges from *F* to *D*. Each edge can have at most $f(e) = 1$
  1. Each node in *F* and *D* participates in at most 2 edges of *M.* But since the flow entering into each of the F nodes can be at most 2, we have a relationship like #edges of the cut $(s \cup F, D \cup I \cup t)$ = flow into F nodes
  2. Hence $|M| = k$, considering such cut.

HOMEWORK 1 Students VALERIO VALENTE 1894954 - GABRIELE DE STRADIS 1947850

Exercise 3

In order to solve this problem, our interpretation on the solution considered a Greedy approach. Thus, respecting the constraint of the running time O(n log n), we decided to sort all the projects so that the algorithm considers all the projects at most once. The sorting criterion has been the core of our solution. We tried many approaches before coming up with the optimal one. Just to mention a bunch of them, we tried to perform the sorting based on highest/lowest b_p and c_p or even the ratio (c_p / b_p) but for all of them we found a disproving counterexample. In the end, what we considered the best approach is the following one.

IMPLEMENTED SOLUTION

Given a set P of n projects p, first we scan the whole set and put into an array P_+ the projects with positive b_p and into another one, P_-, those with negative b_p. Then, we perform two merge sorts over the two arrays with different approaches. For P_+, sorting is made by an ascending delta Δ, calculated as $c_p - C$, where C is the initial score. For P_-, sorting is made by a descending value *v* calculated as $c_p + b_p$. Subsequently, the sequence composed by $P_+ \cup P_-$ will give the optimal order of projects. In order to solve the problem of whether it is possible or not to complete all n projects starting from a given score C, we need to effectively iterate over the final sorted sequence by summing up b_p to the value C. While $C \geq c_p$ for each project p_i, we will consider the provided sequence as a feasible one. Otherwise, the algorithm will terminate and return the impossibility to perform all the n projects.

RUNNING TIME

Given this model, running time is calculated as follows: the overall initial scan is computed in O(n). The sorting of both arrays is computed in O(n log n) each. Finally, we rescan all the sorted n projects in O(n). Having O(n) + O(n log n) + O(n log n)+ O(n), the overall running time is O(n log n).

PROOF OF CORRECTNESS (by contradiction)

The core condition of the whole program is based on the condition that, given project $p \in P$, the overall score C must be $\geq c_p$.

For what concerns the first subset P_+, we clearly respect this constraint since it is sorted basing on ascending $\Delta = c_p - C$. Hence, we have for a given $k \in P_+$ that $c_{pk} - C \geq c_{pk-1} - C$, having all $b_p > 0$. The optimal solution encompasses the condition that $c_{pk} \geq c_{pk-1}$ and C_k (at iteration k) is greater or equal than C_{pk}. By absurd, if $C_k = C_{k-1} + b_{k-1}$ is $< c_{pk}$, we don't respect the core condition and the solution is no longer optimal. So, we must have $C_k \geq c_{pk.}$

As for the second subset P_-, we respect the core constraint since it is sorted basing on descending $v = c_p + b_p$. Hence, we have for a given $k \in P_-$ that $c_{pk} + b_{pk} \geq c_{pk-1} + b_{pk-1}$, having all $b_p < 0$. Thus, $C_k < C_{k-1}$. This means that if $c_{pk} = c_{pk-1}$ we must have $b_{pk} > b_{pk-1}$. If it wasn't the case, we would have an absurd where $b_{pk} < b_{pk-1}$. Having $C_k = C_{k-1} + b_{k-1}$ and assuming $b_{pk-1} < c_{pk} - C_{k-1}$, we would have $C_k < c_{pk}$, which contradicts the core condition of correctness.

COUNTEREXAMPLE CORRECTNESS SUBSET P_- (choosing inverse b_{pi} when equals c_{pi})

Consider two projects P1 and P2 $\in$ P as following: $c_{p1} = 10$ $b_{p1} = -1$; $c_{p2} = 10$ $b_{p2} = -10$. Consider having $C_{k-1} = 11$ in the iteration k-1.

- Optimal case: having $c_{p1} == c_{p2}$, the optimal order will be P_1 -> P_2, leading to $C_k = 11 - 1 = 10$, where $C_k \geq c_{p2}$.
- Absurd case: we order by inverse P2 -> P1, leading to $C_k = 11 - 10 = 1$, where $C_k < c_{p1}$. Hence, the core condition would not be respected.

HOMEWORK 1 Students VALERIO VALENTE 1894954 - GABRIELE DE STRADIS 1947850

Exercise 4.1

In order to solve this problem, we had to think about an algorithm that, for each cure $c_i \in \{c_1,\dots,c_n\}$, performs a certain number of tests. The main goal of our implementation was to find a criterion to perform efficiently these tests without exceeding the given running time. We finally achieved this through an approach based on a binary search. The algorithm can be described as follows.
For each cure c_i we have to find its a_i value i.e., the minimum number of units that kill the virus. So, assuming *d* as an upper bound for all the a_i. Thus, the binary search starts looking at *d*/2 units. If we get that this number of units killed the virus, then we update the upper bound and continue iterating the test between 1 and the upper bound value units. If we get that the virus has not been killed, then we update the lower bound and continue iterating with the same approach between the lower bound value and *d* units. In the end, the updated lower and upper bounds will collide on the same number, that is the minimum number of units a_i. These operations will result in a running time of O(log n). Given the facts that we iterate on all n cures and constantly update the index of the cure with the minimum a_i, we will get the most effective cure in O(n log d).

Exercise 4.2

In order to solve this exercise, we had to think on a randomized approach that implements the algorithm. We want to find the optimal cure c among the set $C = \{c_1,\dots,c_n\}$ performing $O(n + \log n \cdot \log d)$ tests. Our idea is based on a *Las Vegas* algorithm, which iterates some tests over smaller subsets, by decreasing a parameter each time. In our case, we iterate over smaller subsets of C, by decreasing the number of units to test each cure c_i with. At each iteration, the number of units will be chosen randomly.
The algorithm starts with a random value $h_1 < d$ and tests all the cures, giving $\theta(n)$ tests. For some of them, the number of units h_1 will not kill the virus, but for others it will. In the next step, we will pick up a random $h_2 < h_1$ and perform the tests on the subset of cures that in the previous step returned that virus was killed. We will end up by defining a minimum value for the units, that will correspond to one or more optimal cures. For these repeated steps, the overall amount of tested cures is O(log n) as we repeatedly splitted the subsets into the effective and non-effective cures with the provided h_i. For each subset, we repeatedly decrement the number of units of cure tested, so that the overall possible tests are O(log d). This will result in an overall $O(n + \log n \cdot \log d)$.