1 Exercise 1

Algorithm 1: Exercise 1.1

```
Input: str
Output: the logest palindrome substring of str
$n \leftarrow$ length of str
$M[n][n] \leftarrow$ matrix of $n \times n$ elements
start ← 0, end ← 0
for ($i = 0$; $i \leq n$; $i$++) do
    M[i][i] = 1;
for ($i = 0$; $i \leq n - 1$; $i$++) do
    if str[i] == str[i + 1] then
        M[i][i+1] = 1
        start = i
        end = i+1
    else
        M[i][i+1] = 0
for (curr_len = 3; curr_len ≤ n; curr_len++) do
    for ($i = 0$; $i < n -$ curr_len $+ 1$; $i$++) do
        j ← i + curr_len - 1
        if str[i] == str[j] and M[i+1][j-1] == 1 then
            M[i][j] = 1
            start = i
            end = j
        else
            M[i][j] = 0
return getSubstring(str, start, end)
```

1.1 Exercise 1.1

This problem can be solved by using dynamic programming with memoization. Given the string of length n, in order to solve the problem we can define an $n \times n$ matrix in which, in the generic $M[i][j]$ cell we can store if the substring that starts from index i and ends in index j is palindrome (1) or not (0). We start from the first base case, i.e by filling the elements of the matrix that corresponds to a single character, that are the ones on the diagonal because they are the elements in which $i = j$ (i.e that corresponds to the substrings that starts in i and ends in i), and they are of course palindrome. Then we can fill the elements that corresponds to the second base case, that is the case in which we consider the substrings of $length = 2$, and they are palindrome if the two characters are equal. Then, for the substrings of $length \geq 3$ we can generalize by saying that they are palindrome if the character in position i is equal to the character in position j and if the "inner" substring is also palindrome, and the inner substring is the one that starts at $i+1$ and ends in $j-1$, so the information will be stored in the cell $M[i+1][j-1]$. Since the algorithm considers all the possible substrings in increasing order of length, in order to return the longest palindrome substring we simply need to store the *start* and *end* indexes every time that we find a new substring that is palindrome, given that its length will be greater or equal than the previous considered one.
Since there are two nested for loops that depend on the size of the input string, the time complexity of the algorthm is $O(n^2)$.

Algorithm 2: Exercise 1.2

```
Input: str
Output: the longest palindrome subsequence of str
$n \leftarrow$ length of str
$M[n][n] \leftarrow$ matrix of $n \times n$ elements
subseq = ""
subseq_reverse = ""
for ($i = 0$; $i \leq n$; $i$++) do
    M[i][i] = 1;
for (curr_len = 2; curr_len ≤ n; curr_len++) do
    for ($i = 0$; $i < n -$ curr_len $+ 1$; $i$++) do
        j ← i + curr_len - 1
        if str[i] == str[j] then
            if curr_len == 2 then
                M[i][j] = 2
            else
                M[i][j] = M[i+1][j-1] + 2
            if M[i+1][j-1] == 1 then
                subseq = str[i+1] + subseq
            subseq = str[i] + subseq
            subseq_reverse = subseq_reverse + str[j]
        else
            M[i][j] = max(M[i+1][j], M[i][j-1])
return subseq + subseq_reverse
```

1.2 Exercise 1.2

For this exercise i used again the dynamic programming approach with memoization, storing this time in each cell $M[i][j]$ the length of the longest palindrome subsequence (lps) of the substring with starting character i and ending character j. Again, we start with the base case filling the cells in the diagonal with the length 1. Then we consider all the possible substrings of *str* of $length \geq 2$ and we can generalize by saying that the length of the longest palindrome subsequence between the i^{th} and the j^{th} characthers will be the length of the lps in the inner substring plus 2 if str[i] and str[j] are the same (because we found two more characters that can be added to the lps), otherwise it will be the maximum of the length of the lps of the current substring in which we do not consider the i^{th} character ($M[i+1][j]$) and the one in which we do not consider the j^{th} character ($M[i][j-1]$). In order to save the lps found at each step, i defined two string variables, *subseq* and *subseq_reverse* that will contain the two "halves" of the lps, and they will be modified every time that we find out that the starting and ending characters of the current substring are equal and they need to be added to the lps, concatenating the new character respectively at the beginning and at the end of *subseq* and *subseq_reverse*, in such a way that the final lps returned will simply be the concatenation of the two.

Since in the algorithm we have two nested loops that depend on the size n of the input string, the time complexity of the algorithm is $O(n^2)$, which is also $O(n^3)$.

2 Exercise 2

The goal of the exercise is to find if it exists a seating arrangement such that every person forms a good pair with both of its neighbors. This means that the goal of the problem is to find whether it is possible to assign to every $i \in I$ and to every $f \in F$ exactly two neighbors that are in their interests or not. In order find the solution to the problem, we can construct a flow network with the following constraints:

- To each investor $i \in I$ it is possible to assign at most 2 founders that will be its neighbors.
- To each founder $f \in F$ it is possible to assign at most 2 investors that will be its neighbors.
- Each assigned pair of neighbors (i_i, f_j) can appear in the assignment at most once.

So, from these constraints that can be derived from the definition of the problem, the definition of the network flow N that solves the problem is the following.

2.1 Formal definition of the flow network N

Given I, F and $P \subseteq I \times F$ that are, respectively, the set of all the investors, the set of all the founders and the set of all the good pairs in the form (i, f) given in input to the problem, the flow network N can be defined in the following way:

- Create a node s (the source).
- Create a node t (the sink).
- For each investor $i_i \in I$ create a node i_i and an edge (s, i_i) directed from s to i_i, with capacity 2.
- For each founder $f_i \in F$ create a node f_i and an edge (f_i, t) directed from f_i to t, with capacity 2.
- For each good pair $(i_i, f_j) \in P$ create an edge (i_i, f_j) directed from i_i to f_j, with capacity 1.

Having defined this flow network N, it is possible to prove that if we find the maximum flow of N (for example with Ford-Fulkerson), the maximum flow corresponds to the maximum number of different good pairs that we can have in the seating arrangement that satisfie the above constraints, and if the maximum flow is such that $maxFlow = 2|I| = 2|F|$, then it is possible to have an arrangement with only good pairs, otherwise it is not possible.

2.2 Proof of correctness

Let's first prove that the maximum flow in N corresponds to the maximum number of good pairs that we can have in the arrangement. So, in order to prove it, let's first suppose that there is an arrangement in which there are k good pairs (of course satisfying the above constraints) $(i_{i_1}, f_{i_1}), ..., (i_{i_k}, f_{i_k})$. Then, if we consider the flow that sends one unit along each path $s \to i_{i_j} \to f_{i_j} \to t$, then this is an $s - t$ flow of value k. On the other hand, suppose now that we have a flow in N with value k. Then, by the integrality theorem and by the fact that each intermediate edge (i.e the edges from I to F) has capacity 1, the value of the flow on these edges is either equal to 1 or to 0. So, if we now consider a set A as the set of the edges of the form (i, f) in which the value of the flow is equal to 1, then we can prove that:

- A contains exactly k edges, because if we consider the cut $C = \{s\} \cup I$, then the value of the flow is the total flow leaving C (and this is exactly the cardinality of A) minus the total flow entering in C (and this is 0 because there are no edges entering in C), thus, since the value of the flow is k, A contains exactly k edges.
- Each node $i \in I$ ($f \in F$) is the tail (head) of at most two edges in A, because suppose that this is not true and that i (f) is the tail (head) of at least three edges in A, then this would mean that at least three units of flow leave from (enter into) i (f). But by the conservation of the flow this requires that at least three units of flow would have enter into (leave from) i (f), and this is never possible because only a single edge of capacity 2 enters in (leave from) i (f). Thus i (f) is the tail (head) of at most two edges in A.

From these facts, we can see that if we view A as the set of good pairs of neghbors that we have in the seating arrangement, then this is an arrangement that satisfie the above constraints (each $i \in I$ and each $f \in F$ can be assigned with at most two neigbors) and in particular is an arrangement that contains k good pairs of neigbors.

So we proved in both directions that having an arrangement that contains k good pairs of neighbors corresponds to having a flow of value k in N. Then from this directly follows that the value of the maximum flow in N corresponds to the maximum number of good pairs of neigbors that we can have in a seating arrangement, and the pairs in such arrangement are exactly the pairs (i_i, f_j) for which the value of the flow is equal to 1 on the edge from i_i to f_j in N.

Let's now prove that if $maxFlow = 2|I| = 2|F|$ then we can have an arrangement with only good pairs, that is to say that everyone is assigned to two neighbors that forms a good pair with himself. Then, each investor $i \in I$ needs to appear in exactly two good pairs, so the maximum number of different good pairs that we can have in the arrangement, that is also equal to the value of the maximum flow of N, must be equal to $2|I|$, beacause i cannot have more than $2|I|$ different good pairs since this would mean that some i appears in more than two good pairs, and this is not admissible by construction. So from here we find that we should have that $maxFlow = 2|I|$. But the exact same reasoning can be applied also to the founders, because also for the founders we should have that they all need to appear in exactly two different good pairs, so from this we find that the maximum flow should also be $maxFlow = 2|F|$. So, in order to have a seating arrangement with only good pairs, both of these conditions need to be verified, and so we should have that the condition that needs to be verified is exactly that $maxFlow(N) = 2|I| = 2|F|$.

3 Exercise 3

```
Algorithm 3: Exercise 3
   Input: $C$, $P = [(c_1, b_1), ..., (c_n, b_n)]$
   Output: true if it's possible do all the projects,
           false otherwise
 1 currentCS ← C
 2 positive_bp ← [ ]
 3 negative_bp ← [ ]
 4 foreach project $p \in P$ do
 5     if $p$ has $b_p \geq 0$ then
 6         add the project $p$ to positive_bp list
 7     else
 8         add the project $p$ to negative_bp list
 9 Sort projects in positive_bp in increasing order of
   their $c_p$
   Sort projects in negative_cp in decreasing order of
   their $c_p$, by giving precedence to the projects that
   has greater $b_p$
10 foreach project $p \in$ positive_bp do
11     if $p$ has $c_p \leq currentCS$ then
12         currentCS += $b_p$
13     else
14         return false
15 foreach project $p \in$ negative_bp do
16     if $p$ has $c_p \leq currentCS$ then
17         currentCS += $b_p$
18     else
19         return false
20 return true
```

As a first step, the algorithm simply checks, for each project, if its corresponding value of b_p is positive or negative, and depending on this it inserts the project in the list of projects that has $b_p \geq 0$ or in the list of projects that has $b_p < 0$. Then these two lists are treated separately: the list of projects with $b_p \geq 0$ is sorted in increasing order of their c_p, while the list of projects with $b_p < 0$ is sorted in decreasing order of their c_p by giving precedence to the projects that has a greater b_p. After the sorting steps, the algorithm simply needs to consider in order each project (first the projects in the positive_bp array and second the ones in the negative_bp array) and if at some point it finds that the current credit score (that is updated every time that a project can be done by adding its corresponding b_p) is not sufficient to do the current project, then the algorithm simply returns false, because there is no other possibility that the current project can be done and so this means that it is not possible to do all the projects, otherwise it returns true. Since the for loops run in $O(n)$, the running time of the algorithm is simply given by the running time of the sorting algorithm: we can use MergeSort, and since we need to order a total of n projects, the running time is $O(nlogn)$.

3.1 Proof of Correctness

The proof consists in proving that the ordering performed by the presented algorithm is optimal for the given problem, that is to say that either it finds that all projects can be done, or when it finds that the *currentCS* is not sufficent to do some project, it can conclude that this is not possible. Let's split the proof by dividing between the two lists of projects with $b_p \geq 0$ and $b_p < 0$.

For both cases, let's assume that $S = \{p_1, ..., p_k\}$ is the sequence of choices done by the presented algorithm and let's suppose that this is not the optimal sequence of choices. Then it will exists another sequence of choices that instead is optimal, and let's denote it by $O = \{p'_1, ..., p'_k\}$. Since we are supposing that $S \neq O$, let p'_i be the first element in the sequence of choices of O that differs from the the element p_i in the same i^{th} position of S.

-**Case of positive** b_p: in this case, since the projects in S are ordered in increasing order of c_p, this means that for sure $c_{p_i} \leq c_{p'_i}$. Now, let's assume that when we consider p'_i the *currentCS* is such that $c_{p_i} \leq currentCS \leq c_{p'_i}$ (this is the only interesting case because in the other two cases in which $c_{p_i} \leq c_{p'_i} \leq currentCS$ and $currentCS \leq c_{p_i} \leq c_{p'_i}$ either we can do both or none of the two, so the result of the algorithm at that same step will not change). Then in this case we are not able to do p'_i, but it turns out that if we choose p_i first instead, then we are able to do it with the *currentCS* and, since its $b_p \geq 0$, it will also increase the *currentCS* and be able to do also other projects with an higher c_p than the *currentCS*.
- **Case of negative** b_p: Since the projects in S are ordered in decreasing order of their c_p, then we have that $c_{p_i} \geq c_{p'_i}$. Now, let's assume that when we consider p'_i the *currentCS* is such that $currentCS \geq c_{p_i} \geq c_{p'_i}$ (this is the only interesting case because in the other two cases in which $c_{p_i} \geq c_{p'_i} \geq currentCS$ and $c_{p_i} \geq currentCS \geq c_{p'_i}$ we have that either in the first case i'm not able to do any of the two, or i'm only able to do p'_i, but since since $b_p < 0$ for every project, the *currentCS* cannot be improved in any way and so i will eventually find that is not possible to do all the projects anyway). Then in this case i can do both with the *currentCS*, but if i first do p'_i, then it could happen that $currentCS + b_{p'_i} \leq c_{p_i}$, and in this case we would find that there is no way that i will be able to do p_i later, because all the projects has $b_p < 0$ and there is no way of incrementing the value of *currentCS*, while if i would choose p_i before p'_i i would be able to do p_i for sure and it could be possible to do also p'_i, since it required less credit score than p_i.

So, in both cases if we define O^* as $O^* = \{p'_1, ..., p'_{i-1}, p_i, p'_{i+1} ..., p'_k\}$ we obtain a solution that is even better than O, because we have just proved that considering all the possible cases, by considering the project p_i of S instead of p'_i the solution can only be improved, and this means that O is not optimal, so this contraddicst the intial assumption that O was optimal, and so the sequence of choices S made by the presented algorithm is optimal.

4 Exercise 4

Algorithm 4: Exercise 4.1

```
Input: cures = $[c_1, ..., c_n]$, $d$
Output: returns the best cure and its $a_i$
1  best_cure ← $c_1$
2  ai_min ← $d$
3  foreach cure $c_i$ in cures do
4      $L \leftarrow 1$
5      $R \leftarrow d$
6      curr_ai ← $d$
7      while ($L \leq R$) do
8          mid ← $\lceil \frac{L+R}{2} \rceil$
9          if mid units of $c_i$ kill the virus then
10             curr_ai ← mid
11             R ← mid−1
12         else
13             L ← mid+1
14     if curr_ai < ai_min then
15         ai_min ← curr_ai
16         best_cure ← $c_i$
17 return (best_cure, ai_min)
```

4.1 Exercise 4.1

The algorthm picks in order every cure from the first to the n^{th} and for each cure c_i it finds its minimum number of units (a_i) that are needed in order to kill the virus, by performing a binary search on the numbers of units from 1 to d. In particular, the binary search is done in such a way that at each step, it tests the cure with the number of units in the middle of the current "range" and: if that number of units do kill the virus, then this is stored in *curr_ai* and the search continues only on the number of units less than *mid*, because the current a_i of c_i found could not be actually the minimum, so the search cannot terminate yet, while if that *mid* number of units do not kill the virus, then the search continues on the number of units that are greater than *mid*, because more units of c_i are needed to kill the virus. Then, when the binary search for the a_i of c_i terminates, the algorithm simply checks if the a_i of c_i is less than the *ai_min* found so far, in which case *ai_min* will be replaced by a_i and the cure c_i stored as the best cure found so far. Since the algorithm, for each cure n performs a binary search on the dose quantities from 1 to d, testing at each iteration the number of dose quantities in the middle, the number of tests that will be done by the algorithm is $O(nlog(d))$.

Algorithm 5: Exercise 4.2

```
Input: cures = $[c_1, ..., c_n]$, $d$
Output: returns the best cure and its $a_i$
1  best_cure ← $c_1$
2  ai_min ← $d$
3  for i from 0 to n do
4      Randomly select a cure $c_i$ from cures that has
        not yet been selected before
5      if ai_min − 1 units of $c_i$ kill the virus then
6          $L \leftarrow 1$
7          $R \leftarrow$ ai_min
8          curr_ai ← ai_min
9          while ($L \leq R$) do
10             ...                      // exactly as before
11         ai_min ← curr_ai
12         best_cure ← $c_i$
13 return (best_cure, ai_min)
```

4.2 Exercise 4.2

In the second part of the exercise we need to introduce the randomization, and this is done in the way of picking the next cure that needs to be tested: instead of picking each cure in order from 1 to n, the cure is selected uniformly at random between all the cures that has not yet been selected before. Then for each cure c_i considered, instead of performing the binary search on the number of dose quantity from 1 to d as before, we first test if the cure kills the virus with *ai_min*-1 dose quantities of the best cure found so far: if it doesn't kill the virus, then there is no need to find the a_i of c_i, because it will be for sure greater or equal than the one of the better cure found so far, so we can simply go on testing the next cure. Otherwise, if *ai_min* -1 units of c_i do kill the virus, then this means that c_i is better than the best cure found so far and so we need to find its a_i with the binary search, that will be between 1 and *ai_min* -1. Now, let X_i be the random variable that is 1 if the cure c_i is the better so far and i need to find its a_i, and 0 otherwise.

Then $X = \sum_{i=1}^{n} X_i$ is the number of times in which we find a better cure and we need to find its a_i and $E[X_i] = P(c_i\ is\ better\ than\ all\ the\ other\ cures\ tested\ so\ far)$. Since we pick the cures in a random order, without considering the ones that are already been tested before, if we consider the first i cures, each of them has equal probability to be the better so far, so the probability that the cure i is better of any of the other cures from 1 to $i-1$ is simply $1/i$. So we have that $E[X_i] = 1/i$ and $E[X] = \sum_{i=1}^{n} 1/i$, but this is equal to $log(n)$ because is the Harmonic series $H(n)$. Then the expected number of cures for which we need to find their a_i with the binary search is $log(n)$ in average. This means that the algorithm will always do n tests, because for each cure it needs to test if that cure kills the virus with the a_i found so far, and then in average it will also perform $log(d)$ tests (that are the number of tests needed to find the a_i of a single cure) for $log(n)$ cures, so the number of tests done by the algorithm will be in average $O(n + log(n)log(d))$. This result makes sense if we can prove that the probability that X is far from its expectation is very low, and this can be done by applying the Chernoff Bound: $P(X > (1+\delta)\mu)) \leq e^{-\frac{\delta^2 \mu}{3}}$ with $0 \leq \delta \leq 1$ and $\mu = E[X] = log(n)$. If i choose for example $\delta = \frac{1}{2}$, then i have that $P(X > (1 + \frac{3}{2}log(n)) \leq e^{-\frac{log(n)}{12}} = \frac{1}{\sqrt[12]{n}}$, so the probability to find more than $log(n)$ better cures for which we need to find their a_i with the binary search goes asymptotically to 0 as far as n goes to ∞.