# Exercise 1

## 1.1

To solve this point we can use *dynamic programming*. We build an $n \times n$ matrix starting from the main diagonal (given the input string $s$, $n = len(s)$) and we keep building the matrix toward the top-right corner. Since we have to build and populate an $n \times n$ matrix, the cost is $O(n^2)$.

**Fun: LongestPalindrome(s)**
**Initialize M** $n \times n$ *with all 0*
$(start, end) \leftarrow (0, 0)$
**for** $c \in [0, len(s)]$ **do**
  **for** $i \in [0, len(s) - c]$ **do**
    j = i + c
    **if** i == j **then**
      $M[i][j] \leftarrow 1$
    **if** $i == j - 1 \wedge s[i] == s[j]$ **then**
      $M[i][j] \leftarrow 1$
      $(start, end) \leftarrow (i, j)$
    **if** $M[i+1][j-1] == 1 \wedge s[i] == s[j]$ **then**
      $M[i][j] \leftarrow 1$
      $(start, end) \leftarrow (i, j)$
**Return s[start: end]** *//end included*

## 1.2

Also in this case we use dynamic programming and we build an $n \times n$ matrix. To get the solution we use the function *OPT(0, len(s))*. Since, also in this case, we have built and populated an $n \times n$ matrix and the function OPT(0, len(s)) only does O(n) operations, the cost is $O(n^2)$.

**Fun: BuildMatrix(s)**
**Initialize M** $n \times n$ *with all 0*
**for** $c \in [0, len(s)]$ **do**
  **for** $i \in [0, len(s) - c]$ **do**
    j = i + c
    **if** i == j **then**
      $M[i][j] \leftarrow 1$
    **else if** $s[i] == s[j]$ **then**
      $M[i][j] \leftarrow 2 + M[i+1][j-1]$
    **else**
      $M[i][j] \leftarrow max(M[i+1][j], M[i][j-1])$

$$OPT(i,j) = \begin{cases} null & i < j \\ s[i] & i = j \\ s[i:j] & \text{i = j-1 } \wedge \text{s[i] = s[j]} \\ s[i] + OPT(i+1, j-1) + s[j] & s[i] = s[j] \\ OPT(ArgMax(M[i][j-1], M[i+1][j])) & else \end{cases}$$

With the function *ArgMax(M[i][j],M[a][b])* we mean a function that returns the couple (i,j) iff (M[i][j]≥M[a][b]) or the couple (a,b) otherwise.

## Exercise 2

To solve the problem of finding a seating arrangement such that neighbours are good pairs, we design a flow network such that the graph G is defined in the following way: G = (I ∪ F ∪ $\{s, t\}$, P), where I is the set of investors, F a set of founders and $P \subseteq I \times$F the set of good pairings. An edge from $i_a \in I$ to $f_a \in F$ exists iff $(i_a, f_a) \in P$, it is directed from $i_a$ to $f_a$ and has *capacity* = 1. Finally, we add the source $s$ and the sink $t$, connecting $s$ with edges of *capacity = 2* from s to each node in I, and $t$ with edges of *capacity = 2* from each node in F to t. Designing the network in this way guarantees that every $i_a \in$ I will seat near $f_a$ if and only if there is a flow in the edge $(i_a, f_a) \in$ P, i.e. $i_a$ and $f_a$ are neighbours iff $flow(i_a, f_a) = 1$. The problem is solvable iff the max flow of the graph G is equal to $m = 2\,|I| = 2\,|F|$ (if $|I| \neq |F|$ there will not be enough founders for every investor, or vice versa).

**Thesys: max flow = m ⇔ problem is solvable**

**Proof:**

1. $Maxflow = m \implies$ problem is solvable.
   With $n$ a node of the graph, we define $node\_flow(n)$ as the sum of the flow of the edges going out of $n$. If the max flow is $m$ then $node\_flow(f_i) = 2\ \forall\ f_i \in F \implies \exists\ i_a, i_b \in I \mid flow(i_a, f_i) = flow(i_b, f_i) = 1$. If *node_flow($f_i$)* = 2 $\forall\ f_i \in$ F $\implies$ *node_flow($i_a$)* = 2 $\forall i_a \in$ I. So we obtain that $\forall$ *node* $\in I \cup F$, $node\_flow(node) = 2$ and this implies that every node of the graph has two "good" neighbours.

2. *Problem is solvable* $\implies$ *max flow = m.*
   Using reductio ad absurdum, we check what happens if the solvability of the problem implies that the max flow is different from m:

   - max flow is < m $\implies$ $\exists\ f_i \in$ F such that *flow($f_i$, t)* < 2 $\implies$ $f_i$ has only one neighbour from the good pairings.
   - max flow is > m $\implies$ $\exists\ f_i \in$ F such that *flow($f_i$, t)* > 2 $\implies$ $flow(f_i, t) >$ *capacity*$(f_i, t)$ which is impossible.

## Exercise 3

*C initial credits*
*p array of projects*
*c array of credits needed by each project*
*b array of credit score's modifiers*
**Fun: FindOrdering(C, p, c, b)**
**Initialize pos** *as an empty array*
**Initialize neg** *as an empty array*
**for** $i \in [0, len(p)]$ **do**
    **if** $b[i] \geq 0$ **then**
        $pos.append(i)$
    **else**
        $neg.append(i)$
Sort(pos, c) *//sort in ascending order the positive projects over the credits they give*
Sort(neg, c + b) *//sort in descending order over the sum* $c_i + b_i$
*// pos.concat(neg) = concatenation of the arrays pos and neg*
**for** $i \in pos.concat(neg)$ **do**
    **if** $C \geq c[i]$ **then**
        $C \leftarrow C + b[i]$
    **else**
        **return** *false*
**return** *true*

**Proof and running time**
**Running time:** the most expensive operations are the two orderings that can be done in $O(nlog(n))$, with $n = number\ of\ projects$.
**Proof of correctness:** the problem is to find an ordering (if it exists) such that at each step $C_i \geq c_i$, where $C_n = C + \sum_{j=0}^{n} b_j$ and $c_i$ and $b_i$ are respectively the credits needed to work on project $i$ and the credits' modifier. To prove that our ordering is the best one, we divide the proof in two parts, the first one in which we consider only the positive values of $b_i$ and the second one in which we consider the negative values.
**1)** $b_i \geq 0$ **:** we order these values in ascending order of $c_i$; in this part the total value of *credit score* that we have is increasing due to the fact that $b_i \geq 0$. If we are not able to choose any of the projects from this first group (i.e. the set of projects with $b_i \geq 0$) at step $n$ because the total *credit score* is too low, we are not going to find an ordering for the problem (the rest of the projects have negative $b_i$ and won't help to increase our credit). After this part is finished, the total *credit score* is at the highest achievable value (since we have completed all the projects that increase it).
**2)** $b_i < 0$ **:** we define $R_n$ as the set of the remaining projects at step $n$. In this part we want to prove that if the problem is solvable, with our ordering at every step $n$, $C_n \geq c_i \ \forall \ i \ \in \ R_n$ (i.e. at step $n$ we have enough *credit score* to do any remaining projects, otherwise there will be at least one project that we will not be able to do since the *credit score* is only going to decrease). We proceed by induction:
Step 0: $C_0 \geq c_j \ \forall \ j \ | b_j < 0$ otherwise the problem is not solvable.
Inductive step: we assume that, at step $n$, $C_n = C + \sum_{i=0}^{n} b_i > c_j \ \forall \ j$ and we know that if the problem is solvable $\exists \ i \mid C_n + b_i \geq c_j \ \forall \ j \neq i, \ j \in \ R_n$. Our claim is that, if some projects with that property exist, then one of them is project $k \mid c_k + b_k \geq c_j + b_j \ \forall \ j \in \ R_n$ (note that $c_k + b_k \geq c_j + b_j \implies c_k - b_j \geq c_j - b_k$). We can prove this by contradiction: assuming that the problem is solvable even if $\exists \ j \mid C_n + b_k < c_j \implies C_n < c_j - b_k \leq c_k - b_j \implies C_n + b_j < c_k$. Therefore, with $c_k + b_k \geq c_j + b_j$, if by doing $k$ first we are not able to do $j$ later (because $C_n + b_k < c_j$), we have that we are not even able to do $k$ after $j$ ($C_n + b_j < c_k$). If $k$ cannot be done neither before nor later $j$, the problem is not solvable, which contradicts our assumptions; thus, if the problem is solvable, at step $n$, if we pick a project $k$ such that $c_k + b_k \geq c_j + b_j \ \forall \ j \in \ R_n$ we have that $C_{n+1} \geq c_i \ \forall \ i \ \in \ R_{n+1}$ (i.e. at step *n+1* we have enough *credit score* to do any project).

# Exercise 4

## 4.1

In this algorithm we start searching for the best cure considering a starting value of $v = d/2$ and at each iteration we decrease or increase $v$ by $d/2^n$ where $n$ is the number of iterations (we decrease $v$ if there are cures that requires less, otherwise we increase). To achieve this goal we do at most a number of iterations which is logarithmic on $d$. At each iteration over the value of the cure we do at most $n$ other iterations over the list to calculate the cost of the cures; thus the cost is $O(nlog(n))$.

```
Fun: FindCure(C)
Initialize v as d/2
Initialize D as the array storing the indexes for the list of cures C
Initialize low,high as two empty lists
Initialize n as 2
while n ≤ ceil(log2(d) + 1) do
    for c ∈ D do
        if do_test(c, v) then
            low.append(c)
        else
            high.append(c)
        if len(low)! = 0 then
            v ← v − d/(2^n)
            D ← low
        else
            v ← v + d/(2^n)
            D ← high
        high ← [ ]
        low ← [ ]
        n ← n + 1
Return d[0]
```

## 4.2

To solve the previous problem with a randomized algorithm, at each iteration we pick randomly a cure, compute its minimum amount needed to be effective and delete from the list of cures the ones for which this amount is too low. In the next iteration we consider only the remaining cures. On average, at each iteration, half of the cures will be deleted since the randomly picked cure will be, on average, in the middle of the list (the list of the remaining cures, ordered by their $a_i$); so, considering that the cost of finding the minimum amount needed by a cure is $O(log(d))$, we obtain a total cost of $O(log(d)log(n) + n)$.

```
Fun: get_ai(v)
Initialize aux as d/2
Initialize n as 2
while n ≤ ceil(log2(d) + 1) do
    if do_test(v, aux) then
        aux ← aux − d/(2^n)
    else
        aux ← aux + d/(2^n)
    n ← n + 1
Return aux
Fun: FindCure(C)
Initialize v as null
while len(C) > 0 do
    v ← PickRandomCure(C)
    C ← C − v
    a_i ← get_a_i(v)
    for c ∈ C − v do
        if not do_test(c, a_i − 1) then
            C ← C − c
Return v
```