1 Exercise 1

Bianchini - Evangelisti

1.1 The problem of finding the longest palindrome string in `W` could be solved using dynamic programming. We defined the length of the optimal solution `L[i][j]` as it follows:

$$L[i][j] = \begin{cases} 0, & \text{if } i > j \\ 1, & \text{if } i = j \\ L[i+1][j-1] + 2, & \text{if } i < j \wedge W[i] = W[j] \wedge L[i+1][j-1] = j - i - 1 \\ max\{L[i+1][j], L[i][j-1]\}, & \text{otherwise} \end{cases}$$

```
longestPalindrome(L,W,i,j,index)
    if(L[i][j] == NULL)
        if(i>j)
            L[i][j] = 0
        else if(i == j)
            L[i][j] = 1
            index = i
        else if(i<j AND W[i] == W[j] AND
        longestPalindrome(L,W,i+1,j-1,index) == j-i-1)
            L[i][j] = j-i+1
            index = i
        else
            idx1, idx2
            L[i][j] = max(longestPalindrome(L,W,i+1,j,idx1),
            longestPalindrome(L,W,i,j-1,idx2))
            if(L[i][j]== L[i+1][j])
                index = idx1
            else
                index = idx2
    return L[i][j]
```

Defining `max_len` as the result of the function `longestPalindrome`, called with `index=0`, `i=0` and `j=n-1`, where `n` is the length of `W`, the longest palindrome string is found taking the substring of `W` starting from `index` to `index+max_len-1`. Note that this function has a side-effect on the variable `index`.

The computational time is in $O(n^2)$, in fact this is the time needed to build the `L` matrix which has size `nxn` and all values initially set to `null`. During the execution each cell will be filled at most once and every following access has cost $O(1)$ because the function will only return the value `L[i][j]` contained in the matrix.

1.2 The problem of finding the longest palindrome not continuous subsequence of `W` could be solved modifying the previous solution. We defined the length of the optimal solution `L[i][j]` as it follows:

$$L[i][j] = \begin{cases} 0, & \text{if } i > j \\ 1, & \text{if } i = j \\ L[i+1][j-1] + 2, & \text{if } i < j \wedge W[i] = W[j] \\ max\{L[i+1][j], L[i][j-1]\}, & \text{otherwise} \end{cases}$$

```
longestPalindrome_NC(W,i,j,S,L)
    if(L[i][j] == NULL)
        if(i>j)
            L[i][j] = 0
            S[i][j] = ""
        else if(i == j)
            L[i][j] = 1
            S[i][j] = W[i]
        else if(i<j AND W[i] == W[j])
            res=longestPalindrome_NC(W,i+1,j-1,S,L)
            S[i][j] = W[i].append(res[1]).append(W[i])
            L[i][j] = 2 + res[0]
        else
            res1 = longestPalindrome_NC(W,i+1,j,S,L)
            res2 = longestPalindrome_NC(W,i,j-1,S,L)
            L[i][j] = max(res1[0], res2[0], res3[0])
            if(L[i][j] == L[i+1][j])
                S[i][j].append(res1[1])
            else
                S[i][j].append(res2[1])
    return L[i][j],S[i][j]
```

The matrix `S` is used to store in each cell the substring which represents the partial result. The function `longestPalindrome_NC`[1] (called with `i=0` and `j=n-1`) will return the length and the string representing the final solution, which are respectively stored in `L[0][n-1]` and `S[0][n-1]`.

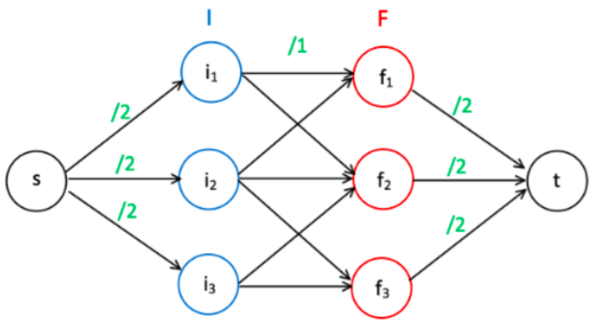
The computational time is in $O(n^3)$: the time needed to build the `L` and `S` matrices, which have size `nxn`, is $O(n^2)$ and the computational cost of the operations done in the `if(L[i][j]==NULL)` block is $O(n)$ (due to the append operation). During the execution each cell will enter this block at most once so the cost is $O(n^3)$. Every following access has cost $O(1)$ because the function will only return the values contained in the matrices.

[1] Note that in the pseudocode `res` represents the values returned by the function: `res[0]` is `L[i][j]` and `res[1]` is `S[i][j]`.

2 Exercise 2

Bianchini - Evangelisti

Given a set of investors `I`, a set of founders `F` and a list `P` $\subseteq$ `I x F` of good pairs representing the preferences of the investors, we have to find an arrangement of round tables, of at least three people each, in order to have only good pairings as neighbors. Each investor `i` $\in$ `I` has to have only founders `f` $\in$ `F` as neighbors and vice versa. In order to know if such an arrangement is possible or not we defined a flow network as it follows:



Flow network N=(V,E):
- vertex set V = I $\bigcup$ F + s + t, where:
  - each $i_i \in$ I represents an investor;
  - each $f_i \in$ F represents a founder;
  - s represents the source of the flow network;
  - t represents the sink of the flow network;

- each edge (i_i, f_i) in the edge set E represents a good pair in the list P.
- the out-edges of s and the in-edges of t are inserted by construction.

In order to have a solution in which each investor `i` has to have only founders `f` as neighbors and vice versa (respecting the good pairs), each `i` has to be connected to at least two `f` (out edges of `i` in the graph) and each `f` has to be connected at least by two `i` (in edges of `f` in the graph). Since $f(e)$ of each edge $e = (i_i, f_i)$ represents whether i_i and f_i are neighbors or not, it's capacity is set to 1 so that the flow $f(e)$ could be 0 or 1 (in order to respect the capacity constraint of the flow network). Considering that we're equipping the room with round tables each person has two neighbors so we set the capacity of the edges from s and into t to 2.

Each investor i_i sits near to the founders f_j, f_k if $f((i_i, f_i)) = 1$ and $f((i_i, f_k)) = 1$ and each founder f_i sits near to the investors i_j, i_k if $f((i_j, f_i)) = 1$ and $f((i_k, f_i)) = 1$.
This solution exists if and only if **maximum flow = C**, where $C = \sum_{\text{e out of s}} \text{c(e)} = \sum_{\text{e into t}} \text{c(e)}$.
We prove this by contradiction:

- We suppose `max_flow < C` and that does exist a solution, so we can have two cases:
  - $\exists$ at least an edge e from s with $f(e) < 2 \Rightarrow \exists$ a vertex i_i with at most one out edge with $f(e) = 1$ and all the others with $f(e) = 0$, that means that i_i investor could sit at most near to one founder, so there doesn't exist a solution because we can't have a table of 2 [$\lightning$].
  - $\exists$ at least an edge e into t with $f(e) < 2 \Rightarrow \exists$ a vertex f_i with at most one in edge with $f(e) = 1$ and all the others with $f(e) = 0$, that means that f_i founder could sit at most near to one investor, so there doesn't exist a solution because we can't have a table of 2 [$\lightning$].

- We suppose that `max_flow = C` and that doesn't exist a solution, so:
  $\Rightarrow$ each i_i has one in edge with $f(e) = 2$ so, as a consequence of flow conservation[2] it will also have two out edges with $f(e) = 1$, so it means that this investor can sit near to 2 *good* founders.
  $\Rightarrow$ each f_i has two in edges with $f(e) = 1$ and all the others with $f(e) = 0$ so it means that this founders can sit near to 2 *good* investors.
  $\Rightarrow$ it does exist a solution [$\lightning$].

If there exists a solution then $C = 2 \times |I| = 2 \times |F|$, and so $|I| = |F|$. It is also important to note that, in order to respect this property, at each table must be accommodated the same number of investors and founders. In fact the only way in which they can sit at the table to respect the problems' constraints it's alternating one founder to one investor. We can prove this by contradiction supposing in one table we have $|F| > |I|$, then if $|I| = k$, the first 1,...,k founders and investors can sit with an alternation of f and i but the remaining $|F| - k$ founders would be one near to the other violating the previous constraint, by the same reasoning we can prove the same if we have $|I| > |F|$.

[2]for each v $\in$ V - {s,t}: $\sum_{\text{e in to v}} f(e) = \sum_{\text{e out of v}} f(e)$

3 Exercise 3

We have `n` projects `p` ∈ `P`. Each `p` is characterized by `(cp,bp)`, in order to do project `p` your current score `C` has to be ≥ `cp`. After doing p_i, the new score $C = C_{i-1}$ `+ bp`. Assuming that for each `p`, `cp + bp` ≥ `0`, the algorithm, in $O(nlog(n))$, returns `True` if you can do all the projects and `False` if not.

```
do_proj(P,C)
    P1, P2
    for each p in P
        if(p.bp >= 0)
            P1.add(p)
        else
            P2.add(p)
    if(do_p1(P1,C)==False)
        return False
    if(do_p2(P2,C)==False)
        return False
    return True
```

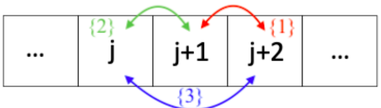
```
do_p1(P,C)
    P <- sorted P by increasing cp
        such that: cp1<cp2<...<cpn

    for each p in P
        if(C >= p.cp)
            C = C + p.bp
        else
            return False
    return True
```

```
do_p2(P,C)
    P <- sorted P by decreasing bp+cp
        such that: bp1+cp1<...<bpn+cpn
    .
    for each p in P
        if(C >= p.cp)
            C = C + p.cp
        else
            return False
    return True
```

As seen in the pseudo-code, P is initially split into 2 subsets: P1 containing the projects with `bp` ≥ `0` and P2 contains the remaining ones. P1 and P2 are solved respectively by `do_p1` and `do_p2`, both greedy algorithms. `do_p1` takes first the projects with lower `cp`, `do_p2` takes first the projects with higher value of `cp+bp`, if one project it's not doable, then the algorithm returns False because we can't do all `n` projects. Note that `do_p1` has a side effect on the value of `C` that will be the input of `do_p2`. If both `do_p1` and `do_p2` return True, then we can do all `n` projects and the algorithm returns True. The **computational cost** of this algorithm is in $O(nlog(n))$, in fact we spent $O(n)$ to split P into P1 and P2, we spent $O(nlog(n))$ to sort in `do_p1` and $O(n)$ when iterating on P. In `do_p2` we spent $O(nlog(n))$ to sort and $O(n)$ when iterating. So $O(n + nlog(n) + nlog(n) + n)$ is in $O(nlog(n))$.

To **prove the correctness** of the algorithm we prove that both `do_p1` and `do_p2` are correct.
In **do_p1** we proceed by induction. We assume we've done j projects: if we can't do project $j+1$ because $C < c_{j+1}$ then we can't do all the remaining ones $p_{j+2} \ \ldots \ \ p_n$ because we sorted by increasing `cp`, so $cp_1 \leq ... \leq cp_n$ and it doesn't exist another sorting in which all projects are done because $bp > 0$ so if we can't do the project p at iteration $j+1$ we can't do it neither in other iterations. In fact the current score at j+1 is $C_{(j+1)} = C_{(0)} + \sum^j bp$ would be the same even if we change the order in which the projects 1,...,j are done.



To prove **do_p2** we assume that we can't do the project j+1 at iteration j+1 and we prove that we can't do it in any other order.
{1} If we can't do project $j+1$ at that iteration, since $C < c_{pj+1}$, we can't switch it with none of the following projects because, having all $b_p < 0$, the value of C at each subsequent iteration will be decreased and so the condition $C < c_{pj+1}$ will be still true.
{2} We assume that we can do the project $j+1$ at the iteration j and the project j at iteration $j+1$:
• having C at iteration $j+1$, at iteration j we have $C' = C - b_{pj}$, and we assume that $C' \geq c_{pj+1}$. After doing project $j+1$, $C'' = C' + b_{pj+1}$.
• to do project j, we have $C'' \geq c_{pj} \Rightarrow C' + b_{pj+1} \geq c_{pj} \Rightarrow C - b_{pj} + b_{pj+1} \geq c_{pj} \Rightarrow C + b_{pj+1} \geq c_{pj} + b_{pj}$, but considering $C < c_{pj+1}$ and given our order $c_{pj} + b_{pj} \geq c_{pj+1} + b_{pj+1}$, we have a contradiction.[ϟ]
So, project $j+1$ can't be exchanged neither with the previous {2} nor the following ones {1}.
{3} We try to exchange p_j with p_{j+2}. If $c_{pj+2} \geq c_{pj} \Rightarrow b_{pj+2} \leq b_{pj}$ (considering our sorting of projects), even if we could exchange these projects we can't do p_{j+1}, C would be smaller. So we exchange p_j and p_{j+2} considering $c_{pj+2} < c_{pj}$:
• having C at iteration $j+1$, at iteration j we have $C' = C - b_{pj}$, and we assume that $C' \geq c_{pj+2}$. After doing project $j+2$, $C'' = C' + b_{pj+2}$.
• to do project $j+1$, we have $C'' \geq c_{pj+1} \Rightarrow C' + b_{pj+2} \geq c_{pj+1} \Rightarrow ... \Rightarrow C + b_{pj+2} \geq c_{pj+1} + b_{pj}$, but considering $C < c_{pj+1} \Rightarrow c_{pj+1} + b_{pj+2} > c_{pj+1} + b_{pj} \Rightarrow b_{pj+2} > b_{pj}$, so we assume we do $p_{j+1} \Rightarrow C''' = C'' + b_{pj+1}$.
• to do project j we have $C''' \geq c_{pj} \Rightarrow C'' + b_{pj+1} \geq c_{pj} \Rightarrow ... \Rightarrow C + b_{pj+2} + b_{pj+1} \geq c_{pj} + b_{pj}$, considering $C < c_{pj+1}$ and given our order $c_{pj} + b_{pj} \geq c_{pj+1} + b_{pj+1}$, if we add to $C_{pj+1} + b_{pj+1}$ a negative value, it can't be greater than $c_{pj} + b_{pj}$, so we can't do project j.[ϟ]
So, there not exist any other ordering of the projects in which we can do all the projects.

4 Exercise 4

Bianchini - Evangelisti

We have to find the COVID-19 cure between n candidates: `c1,...,cn` which requires the minimum amount a_i to kill the virus, knowing that each cure with `d` units would work. We need to find the best cure,such that a_i is minimal, using as few tests as possible.
We call C={c1,...,cn} the set containing the n cures.

4.1 The deterministic algorithm that solves the problem in $O(nlog(d))$ is represented by the following pseudo-code.

```
find_cure(C,d)
    best_c
    min_a
    for each c in C
        a=find_dose(c,0,d,d)
        if(a < min_a)
            min_a=a
            best_c=c
    return best_c
```

```
find_dose(c,i,j,d)
    if(i>j OR j==0)
        return d
    mid= (j+i)/2
    if(test(c,mid)==True)
        return min(mid, find_dose(c,i,mid-1,d))
    else
        return find_dose(c,mid+1,j,d)
```

The cost is in $O(nlog(d))$ because we do an iteration on all the `n` cures and for each one we do a binary research to find the right amount $0 <$ `a` $\leq$ d which as a cost in $O(log(d))$.

4.2 The randomized algorithm that solves the problem, in the average case, with $O(n + log(n)log(d))$ test is represented by the following pseudo-code.

```
find_cure_random(C,d)
    tested={}
    while(C-tested is not empty)
        c_i = random(C - tested) [choose a random c_i from C-tested]
        tested = tested U c_i
        a_i = find_dose(c_i,0,d,d)
        for each c_j in C-{c_i}
            if(!test(c_j, a_i))
                C.remove(c_j)
    return c_i
```

The function `find_dose` called in `find_cure_random` is the one defined in **4.1** with computational cost in $O(log(d))$.
At each iteration we chose a random cure from the ones in the set C and we find the minimum dose that this cure needs to kill the virus in $O(log(d))$. Then we test if the other cures in the set work with this amount: if they don't, they won't work even with smaller amounts so we discard them, this operation at the first iteration will cost $O(n)$.
The probability of an amount a_i to be the minimum and so, the best cure is at least $\frac{1}{n}$. In that case we'd remove all the other cures (which are not the best ones) from C and the algorithm would stop.
We performed an empirical analysis implementing the algorithm and doing tests on different sets of cures and we found out that, even if at some iteration we'll discard just one cure and at some other we'll discard more than half of the cures, on average, at each iteration the number of cures to be considered is halved. If at each step n is the half of the previous one, in $log(n)$ steps we have n=1.
So in the average case the number of iterations will be $log(n)$ and the cost is: $log(d)log(n) + (n + \frac{n}{2} + \frac{n}{4} + \frac{n}{8} + ...) = log(d)log(n) + \sum_{i=0}^{log(n)} \frac{n}{2^i} = log(d)log(n) + \frac{1-(\frac{1}{2})^{log(n)+1}}{1-\frac{1}{2}} \simeq log(d)log(n) + 2n$. So the average computational cost is in $O(n + log(n)log(d))$.

We can also verify that the average number of iterations is $O(log(n))$ analyzing the expectation: we consider $X_i = 1$ if we find the best cure at the i^{th} iteration, 0 otherwise. The probability is equal to $P[X_i] = \frac{1}{n-(i-1)}$. From linearity of expectation we can say that E[X]=$\sum_{i=1}^{n} \frac{1}{n-i+1} = \frac{1}{n} + \frac{1}{n-1} + ... + \frac{1}{2} + 1 = H(n)$. So we expect to have $O(log(n))$ iterations.