1 Exercise 1

In this exercise we want to find the expected optimal reward, which requires to generate a $(n+1)\cdot k$ matrix where in each cell $M[r][b]$ is stored the expected reward of the box b when the current reward is r. Considering that, the expected reward of each box depends on the subsequent boxes, we use the backward approach for the calculus, indeed only the last box does not depend on the subsequent draws. Once arrived at the first box the whole matrix is completed and the optimal expected value is stored in $M[0][0]$.

1.1 $\mathbf{O(n^2 \cdot k)}$ solution

- E = expected value, cost[k] = costs vector, M = $(n+1)\cdot k$ matrix.

```
1  for (b from k-1 to 0):
2    for (r from 0 to n+1):
3        E = 0
4        for (j from 0 to n+1):
5            if (j < r):
6                if (b == k-1): E += r/(n+1)
7                else: E += (M[r][b+1])/(n+1)
8            else:
9                if (b == k-1): E += j/(n+1)
10               else: E += (M[j][b+1])/(n+1)
11       E -= cost[b]
12       M[r][b] = max(r, E)
```

The $O(n^2 \cdot k)$ complexity is justified by the fact that we have to compute the sum of n numbers for n expected values. For each cell, the summation is different, because it depends on both the current box b, and the current reward r. For this reason we can't do only one sum but necessarily n sums.

1.2 $\mathbf{O(n \cdot k)}$ improvement

To improve the first algorithm we noticed that is possible to use the *dynamic programming with memoization* to avoid to repeatedly sum up the n values, indeed each summation can be proceeds from the results we have already saved in the matrix. We distinguish two main cases:

- Last box: the expected value of the first cell is equivalent to the sum of the first n natural numbers, divided by $n+1$, so we can use the known math series. After that, we can derive from it the other cells of the column, by adding a normalization factor which depends on the current reward r, finally we subtract the cost.

$$M[r][k-1] = max\left(r,\ \frac{1}{(n+1)} \cdot \left[r^2\ +\ \frac{n\cdot(n+1)}{2}\ -\ \frac{(r-1)\cdot r}{2}\right]\ -\ C[k-1]\right)$$

- Other boxes: first of all we use a variable called *previous_box_sum* in which we store the sum of the previous column, then we set another variable called $diff$ in this way:

```
if (r == 0): diff = 0
else:
  diff = (M[r][b+1] - M[r-1][b+1])*r + previous_diff
  previous_diff = diff
```

Where $previous\_diff$ is a variable created to save the calculated difference in the upper cell. Now we compute the result:

$$M[r][b] = max\left(r,\ \frac{previous\_box\_sum + diff}{(n+1)}\ -\ C[b]\right)$$

2 Exercise 2

In this exercise, given a weighted tree T with n nodes, we want to find the complete graph G of minimum weight such that $T \subseteq G$ and T is the unique minimum spanning tree of G.

Both solutions are based on the Kruskal's algorithm, indeed they start by creating a separate set for each node and then sort all the edges in ascending order. The edges are stored in a priority queue and at each step of the first cycle, the edge e with the lower weight, is extracted. At this point the algorithm finds the sets A, B containing the nodes u, v corresponding to the edge e.

2.1 Complete Graph from weighted Tree - $O(n^2)$

In the first case the algorithm iterate trough all the nodes of the sets A, B and create a new edge every time there is a node of the set A not connected to a node of the set B. This will have a weight corresponding to weight(e) + 1, which guarantees the uniqueness of the mst T by the *cut property*.

```
1 INPUT: mst T (n nodes, m = n-1 edges), OUTPUT: graph G
2 graph G = ∅
3 MakeUnionFind(T.vertex)
4 priority queue Q = sort_ascending_order(T.edges)
5 while Q.is_not_empty():
6         edge e(u, v) = Q.dequeue_min()
7         set A, B = Find(set(u)), Find(set(v))
8         for each w ∈ A:
8                 for each z ∈ B:
10                        if e(w, z) != e(u, v):
11                                weight(e(w, z)) = weight(e(u, v)) + 1
12                                G.add_edge(e(w, z))
13        G.add_edge(e(u, v))
14        Union(A, B)
15 return G
```

2.2 Total weight sum of complete Graph - $O(n \cdot \log(n))$

In the second case, instead, the algorithm does not need to iterate trough all the nodes of the sets. It virtually enumerate all the possible missing edges and sum them with the weight = weight(e) + 1.

```
1 INPUT: mst T (n nodes, m = n-1 edges), OUTPUT: int total_weight
2 graph G = ∅
3 int total_weight = 0
4 MakeUnionFind(T.vertex)
5 priority queue Q = sort_ascending_order(T.edges)
6 while Q.is_not_empty():
7         edge e(u, v) = Q.dequeue_min()
8         set A, B = Find(set(u)), Find(set(v))
9         total_weight += (((size(A) * size(B)) - 1) * (weight(e(u, v)) + 1))
                          + weight(e(u, v))
10        G.add_edge(e(u, v))
11        Union(A, B)
12 return total_weight
```

Considering a pointer-based implementation we assume that *Union* takes O(1), *MakeUnionFind* takes O(n) and *Find* takes O(logn). Due to the introduction of the cycles on the sets (with cost size(A)*size(B)), the actual cost of the first algorithm is not immediate, but analyzing it at running time, it is possible to verify that the total number of iterations is equal to the number of edges of the complete graph n(n-1)/2. Indeed it does never iterate 2 times on the same couple of nodes.

The termination of the algorithm is guaranteed by the limited number of edges in T.

3 Exercise 3

3.1 Basic flow model

In this exercise we want to model the chocolates distribution network of Federico with a network flow, which can be generated as follows:

Source s = Federico
Sink t = Customers
for each $f_i \in F$:
    node W_{f_i} = create_node()
    node F_i = create_node()
    edge e = add_edge(W_{f_i}, F_i).set_capacity(n_{f_i})
for each F_i that Federico meets in person:
    edge e = add_edge(s, W_{f_i}).set_capacity(n_{f_i})
for each F_i that can sell directly:
    edge e = add_edge(F_i, t).set_capacity(s_{f_i})
for each pair $(F_a, F_b) \in F$ that can meet in person:
    edge e_1 = add_edge(F_a, W_{f_b}).set_capacity(n_{f_a})
    edge e_2 = add_edge(F_b, W_{f_a}).set_capacity(n_{f_b})

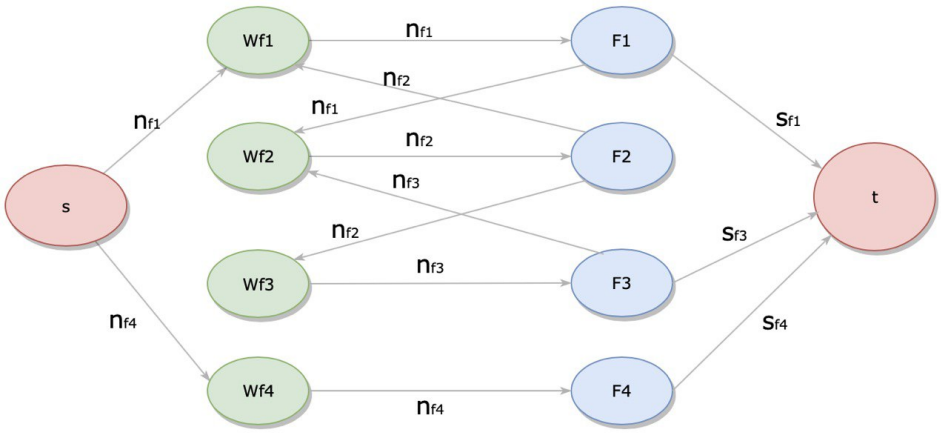

Figure 1: example with $|F| = 4$

In the example we have that F_3 is able to sell but he/she can not directly takes chocolates from Federico, moreover, he/she can't meet F1, who directly meets Federico. However, we know the existence of the pairs (F_1, F_2) and (F_2, F_3), so F_2 can be used as exchange if $s_{f_1} < n_{f_1}$ to give at most n_{f_2} chocolates to F_3.

To permit the double exchange we introduced the W_f nodes, so that we do not break the assumption: if $(u, v) \in E$ exist, then $(v, u) \notin E$.

F_4 has no problems to meet Federico, so he/she can takes an amount of chocolates limited to n_{f_4} and sell s_{f_4} (those will be equal). The exact amount of flow on each edge can be calculated by the Ford-Fulkerson algorithm.

3.2 Introduction of buildings

With the introduction of the university buildings, there is a possible restriction c_b on the amount of chocolates that each friend belonging to a particular building $b \in B$ can sell. This requires to delete the 3^{rd} "for each" and replace it with:

for each F_i that can sell directly in the building $b_j \in B$:
    edge e = add_edge(F_i, b_j).set_capacity(s_{f_i})
for each $b_j \in B$
    edge e = add_edge(b_j, t).set_capacity(c_{b_j})

To notice that this does not change the relationships between friends.

4 Exercise 4

In this exercise we want to demonstrate that our Scheduling problem is *NP−complete*. Given n tasks in the task set J, we have an earliest time s_j to start task j, a deadline d_j when it has to be finished, and the length $l_j \in \mathrm{N}$ that specifies how long it will take a single person to do task j. We also know that $\sum_{j \in J} l_j \leq k$, and, once started, tasks connot be paused and then finished later.

First of all, we turn our problem into a *decision problem*: can we schedule all the jobs so that each of them can be completed within its deadline and before k?

We can prove that this problem is *NP−Complete* by the following steps:

1. Prove that *Scheduling problem* $\in$ *NP* :

   Set a specific starting time s_j for each job j and then check that each job runs in the intervall $[s_j, d_j]$. Thereafter check that all of them can be completed in time k. In this way we can certify that this instance of the problem can be solved in polynomial time and therefore that the problem is in NP. $\square$

2. Choose a problem that is known to be NP-complete: *Subset Sum Problem*

3. Prove that *Subset Sum Problem* $\leq_p$ *Scheduling problem* :

   Given $W = \{w_1, ..., w_n\}$, a set of positive integers such that $X = \sum_{i=1}^{n} w_i$, and a subset $W' \subseteq W$ such that its elements add up to Z, we have to put inside a solvable instance of our problem. Therefore define a set of jobs $J = \{j_1, ..., j_n\}$ such that:

   $$s_{j_i} = 0, \qquad d_{j_i} = X + 1, \qquad l_{j_i} = w_i \qquad\qquad \forall\, j_i \in J$$

   start time, *deadline* and *duration* of the job respectively. This instance solves the scheduling problem because you can arrange the jobs in any order such that they will always meet their deadline, and, for the initial condition, they finish before k.
   Viceversa, suppose that we start with a solvable instance of the Scheduling problem. Let's take the set $\{j_1, ..., j_n, j_{n+1}\}$ where the first n jobs are defined as before and job j_{n+1} as follows:

   $$s_{j_{n+1}} = Z, \qquad d_{j_{n+1}} = Z + 1, \qquad l_{j_{n+1}} = 1$$

   The $(n+1)^{st}$ job can be run only in $[Z, Z+1]$, but this means that there are X units of time available in the interval $[0, X+1]$. Now since this instance is solvable, we have a subset of jobs $W' = \{j_{i_1}, ..., j_{i_m}\}$ such that $\sum_{j \in W'} l_j = Z$ and for this reason we can schedule each $j \in W'$ before job j_{n+1}, and the remaining after job j_{n+1}. $\square$

With these three simple steps we have demonstrated that our problem is *NP-Complete*.