Lorenzo Romagnoli 1975517, Andrea Morelli 1845525

Exercise 1

1.a

To solve the problem we thought about using a dynamic programming approach.
rij=jth parent at distance i
yi = minimum price allowed to ask the relatives in distance i.
oi = gifts from parents in distance i.

We created a recursive function that for every distance *i* of the parents tries all possibles *yi* till the value of *yi-1* (in the range *0,yi-1*) that is passed through the function and asks all the relatives in *i* a value between *yi* and *yi-1* taking into account their *v*: if *v>=yi-1* we ask him a value p=*yi-1*, if *yi<=v<yi-1* then we ask him a value p=*v*, otherwise if *v<yi* we ask him for a value p=*yi* so he will not gift us anything(so that in the next distances we are not limited by his gift value*)*. Now if they can afford our suggestions (*v<=p*) they will give us a gift of the value of p otherwise 0.

We sum all the gifts received in the current i, given the current value of yi, to the value returned by the recursion to which we pass the next i and the current yi. When all the recursive calls for the current i returns, we select the global maximum value of the sum of all gifts taken(gifts from i and all r in dist>i) given i and yi, and we return this value, storing it in dp[i,yi]. Since it is stored, if we enter in a recursion call with the same values i and yi , we don't need to recalculate it since the global maximum value has already been computed for these values and we can return instantly the value stored.

Given the recursive function described above , we start a call with the values i=0,y0=max(*vr*).

-the function is called with max(vr) as y0 to take into account the possibility of having a relative in later distances with a high vr.
-when we enter the function if we arrived at a distance i that is after the last distance possible ,we return 0;
-if the current i and yi has been previously stored then we do not need to reevaluate the function but we return instantly the value stored.
-otherwise try all possible recursions with all the values yi and return the maximum of those and store the result in dp[i,yi] as described in the description of the function.

proof of correctness:
The function returns one of the optimal solutions to the problem because it analyzes and takes into account all the possible suggested prices for every relative and distance starting from a minimum ,in a smart way, taking advantage of the memorization. At each call to a recursion, so at all the distances, the algorithm saves the optimal solution, so evaluate the maximum value of all the recursions + the gifts in the current i, given a specific i and yi. Trying all the possible values of gifts at each distance and for each relative, we arrive at a global solution that is one of the optimal ones.
Calculating the max(giftsAtDistance*i*+ric(i+1,yi)) for each yi in {0...yi-1}, we know that at a given distance *i* and given a value *yi-1* we get the optimal solution. Repeating this for each distance *i* and for each value of *yi* allow us to obtain a global optimal solution for *i=0* and *y0=max(vr)*

analysis of the runtime:
The runtime of the algorithm is O(C* i *max(vr))
Where C is a constant, i is the number of distances and max(vr) is the maximum of all the v between every relative.
In the worst case scenario *i* is equal to r and max(vr)in the worst case is evaluated in every iteration ,so the cost is O(n*max(vr)) that is the cost to fill the entire hashmap,multiplied by a constant value c for every evaluation of the function.
In practise it is also multiplied by a factor to describe the granularity of the result(float numbers or approximations).

1.b

```
relatives=[[15.2,9.4,11.1],
      [9.7,12.2,7.5,10.8],
      [5.2,6.3,5.6,10.2],
      [4.9,2,3.5,2.9],
      [1,6.4,4.6,2.1]]
dp={}
def recursive(dist,pr):
   if dist>=len(relatives):
         return 0
   maximumGift=0
   if (dist,pr)in dp:
         return dp[(dist,pr)]
   for i in range(pr+1):
      totalValueAtRow=0
      for j in range(len(relatives[dist])):
         if relatives[dist][j]>=pr:
            totalValueAtRow+=pr
         elif i<=relatives[dist][j]:
            totalValueAtRow+=relatives[dist][j]
      val=totalValueAtRow+recursive(dist+1,i)
      if val>maximumGift:
         maximumGift=val
   dp[(dist,pr)]=maximumGift
   return maximumGift

if __name__ == "__main__":
   for i in range(len(relatives)):
      for j in range(len(relatives[i])):
         relatives[i][j]=int(relatives[i][j]*10)
   maximumVr=0
   for i in range(len(relatives)):
      for j in range(len(relatives[i])):
         if relatives[i][j]>maximumVr:
            maximumVr=relatives[i][j]
   result=recursive(0,maximumVr)/10
   print(result)#out=116.3
```

We multiply initially every v by 10 to take into account the decimals and then the result will be divided by 10.

The result of the execution of the code with the input of the problem 2.b is: 116.3.

EXERCISE 2

2.a

The network *G(N,E)* modelled is composed as follows: on the left we have the nodes representing the rows, where a row is identified by *ai*, given $ai \in A$, $A \in N$. On the right we have the nodes representing the columns where a column is identified by *bj*, given $bj \in N$, $B \in N$. Each node *ai* is connected with an edge to every node *bj*, and each edge *e*(*ai,bj*) have capacity 1. The idea is that, after the execution of the Ford-Fulkerson algorithm, each *e(ai,bj)* with saturated capacity will indicate that there will be a shop in the city in row *i* and column *j*. Then we placed the source node *S*, which is connected with an edge with every node *ai* and each of these edges *e(S,ai)* have capacity *ri* to limit the flow as stated in the problem, so limit the number of shops that can be opened given a row. Then we added the sink node *t* to which all the nodes *bj* are connected with edges *e(bj,t)* with capacity *cj* to put the limit also on the shops that can be opened given a column, limiting flow passing through that edge.

The scope of the problem is to find the maximum number of shops that can be opened in the city and this can be achieved by applying the Ford-Fulkerson algorithm. The resulting max-flow will be the number of shops that can be opened given the constraints of the problem.

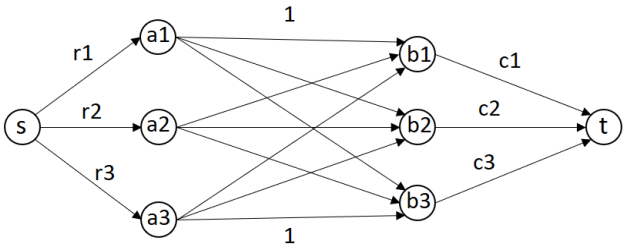

Figure 1: the network flow modelled

2.b

To solve this problem first we modeled a new network flow as shown in the picture below. We added, in the previous graph, some nodes *a* and *b*, respectively representing rows and columns to have 10 of both. Then we added two new nodes *x,y* given $x \in X$, $X \in N$ and $y \in Y$, $Y \in N$. The source node is connected to each node x with an edge *e(s, xz, 1, ∞)* which will assure a minimum of 1 unit of flow passing through it. Then each node *xw* is connected with an edge to 5 nodes *ai*, so with the previously mentioned edge *e(s, xz, 1, ∞)* we will assure that at least one flow unit will pass through a node row given a group of 5 rows as the problem stated. Then each node *y* is connected to the sink with an edge *e(yw, t, 1, ∞)* that will assure that at least one unit of flow will pass through a node column given a group of 5 columns.

To check if it is possible to solve the problem with the new given constraints we reduced it firstly to a circulation problem with demands and lower bounds||: for every edge *e(s,xz,l,c)* with a lower bound, we updated the edge to *e(s,xz,0,c-l)*, we increased the demand of *s* by *l* and we decreased the demand of *xz* by *l*. We made then the same for the edges *e(yw,t,l,c)*, updating those edges to *e(yw,t,0,c-l)* and increasing the demand of each *yw* by *l* and decreasing the demand of *t* by *l*. So we modeled a new graph *G'* from *G*, adding also a "super-source" connected with outgoing edges to the nodes with negative demands and a "super-sink" connected with ingoing edges to the nodes with positive demands.|| From there we had to check if in our new network exists a feasible circulation following the constraints given, so if for every node $n \in N$ and given the demand of each one *{dn}* we have $\Sigma n(dn) = 0$ while also satisfying the capacity and

demand conditions. Then we know that “There is a feasible circulation in *G* if and only if there is a feasible circulation in *G’* ”[1] , so after checking that in *G’* a feasible circulation exists, we know that a feasible circulation exists also in G. From that the problem can be solved by our network flow, given the constraints of the problem.

After having checked the feasible circulation, we translated our graph G’ back to G. Doing this we had edges with some flow passing through them, because of the previous operation, but this is not yet the flow representing the max flow. The previous operations assured us that the constraints on the lower bounds can be fulfilled so from there we have just to calculate the max-flow to get the final answer. To do this we have to execute The Ford-Fulkerson algorithm and, once we get the max-flow of the network flow, we know the maximum number of shops that can be opened in the city with the new constraints of the problem since the number of shops is equal to the value of the max-flow resulting from the algorithm.

---

[1] Kleinberg J, Tardos E, Algorithm Design, ‘proof 7.52’ pp. 384

-Nella trasformazione in G’ non sono presenti tutti i passi. Vedere se necessario descrivere la trasformazione.
-Vedere se necessario definizione formale di tutti i G.

Lorenzo Romagnoli 1975517, Andrea Morelli 1845525

Excercise 3

In order to solve the problem we decided to use a dynamic programming approach:

We created a recursive function makeSelection that for every school tries all possible combinations of boys and girls given k and each combination will be used to continue the recursion in the following schools.

For every possible combination allowed, we call the recursive function to the next school (so the school *i+1*) ,passing to the function the total number of boys chosen until that moment plus the number of boys chosen in the current combination and, the total number of girls chosen until that moment plus the number of girl chosen in the current combination. We then return the minimum of all those recursive calls and save that value(the best for the current boys and girls) .

In the next iterations ,if a recursive function has already calculated a best solution for the same number of boys and girls ,then we return that value without re-evaluating the recursive function, since it will end up with the same best solution.

Since we take into account all possible combinations of boys and girls for every school, we will end up with a global optimal solution for the problem: at every boysChoosen and girlsChoosen we return the max(makeSelection(i+i,boysChoosen+ba,girlsChoosen+ga)for every ba,ga in allValidCombinations(bi,gi,k).

So if we have a maximum for every boysChoosen and girlsChoosen, then makeSelection of (0,0,0) will return the global maximum .

Starting from i=0, boysChoosen= 0, girlsChoosen= 0. Given n= list of schools, i = index of schools, dp = hashmap for memorization

```
function makeSelection ( i, boysChoosen, girlsChoosen):
    if i>=len(n):
        return |boysChoosen - girlsChoosen|
    if  (boysChoosen,girlsChoosen) in dp :
        return dp[(i,boysChoosen,girlsChoosen)]
    l=allValidCombinations(bi,gi,k)
    bestSolution = ∞
    for bi , gi in l:
        possibleSolution= makeSelection(i+1,boysChoosen+bi ,girlsChoosen +gi)
        if possibleSolution < bestSolution:
            bestSolution = possibleSolution
    dp[(i,boysChoosen,girlsChoosen)]=bestSolution
    return bestSolution
function allValidCombinations(bi,gi,k):
    return a list of all possible combinations of gi and bi given k.
```

The cost of the algorithm is O(C* max(bi)*n *max(gi)*n) ≃ O(C*n^2*max(bi)*max(gi))≃O(C*n^2)≃ C*O(n^2) with C that is a constant.Thats because the cost of this problem is the total number of iterations needed to fill the hashmap multiplied by a constant value for every evaluation.

Since we are using a hashmap instead of a matrix in the average case we will have C=O(n^2) but in the worst case scenario it will be O(n^3). Anyway, in every case, the algorithm will run in polynomial time.

Lorenzo Romagnoli 1975517, Andrea Morelli 1845525

Exercise 4:

To prove that the given problem is NP-complete we have to:
-prove that the problem is in np
-there exists a polynomial time reduction from a known NP-complete problem to this one.

To prove the first we have to give a polynomial certificate verifier that is an algorithm that:
for each child checks if the chosen bags for him of the solution contain all the sweets he needs and that no more than k bags of the same types were chosen.
This is clearly a polynomial certifier.

To prove the second point we start from a modified version of SAT where every clause has either all positive or all negative literals, that is so a NP-complete problem.
All the clauses are in this form (x1 or x2 or ...) and (-x1 or -x2 or ...)
From the modified SAT create the following instance of the santa problem:

C=set of clauses
K1={i: ∀ ci ∈ C : if ci in the form of (xj ∨...) }
K2={i: ∀ ci ∈ C : if ci in the form of (-xj ∨...) }
B={Bj : ∀ ci ∈ C in the form of (xj ∨ ...)or(-xj ∨...) : i ∈ Bj }
k=1

The first child will have for each positive clause a sweet i (index of the clause).
The second child will have from each negative clause a sweet i (index of the clause).
We will have for each literal j(index of the literal) present in the clauses a bag j.
For each clause i and for each literal of that clause j , we will have a candy i(index of the clause) in the bag j (index of the literal).
The instance of the problem will have a k=1 so that the grinch will not be able to get more than 1 bag of the same kind.

To get a result for the modified SAT problem from the reduction we analyze the chosen bags by the grinch:
if for the first child the grinch chose the bag i then the xi has to be a 1.
if for the second child the grinch chose the bag i then xi has to be a 0.
if a bag i has not been chosen then xi can be 0 or 1 (we do not care about his value)
if the grinch is not able to fulfill all the two children's needs then there is no solution for the modified SAT.

We now have a polynomial reduction from the modified SAT to the santa problem, that means that the santa's problem is NP-complete.

example:
(x1 ∨ x3 ∨ x4)∧(-x3 ∨ -x4 ∨ -x5)∧(-x3 ∨ -x1)∧(x1 ∨ x5)∧ (-x1 ∨ -x3 ∨ -x4)
we construct the instance of the santa problem:
k=2,K1={1,4} K2={2,3,5}
B1={1,4,5} B2={} B3={1,2,3,5} B4={1,2,5} B5={2,3}
the grinch will choose for example: $\widehat{B}(1)$={1} $\widehat{B}(2)$={3,5}
the the solution for the sat modified problem is x1=1,x3=0,x5=0,x2=(0 or 1),x4=(0 or 1)
Since the grinch was able to find a solution, there exists a solution for the modified SAT problem.

Lorenzo Romagnoli 1975517, Andrea Morelli 1845525