Exercise 1

1.a)

To solve the problem we thought about using a dynamic programming approach.
r_{ij}= j-th parent at distance i
y_i = minimum price allowed to ask to the relatives in distance i. and in each recursive call y_{i-1} is used instead as maximum price allowed
V_i=list of v at distance i

We created a recursive function that for every distance i of the parents tries all possibles $y_i \in \{y_{i-1} \cup V_i\}$ (y_{i-1} was passed to the function in the previous iteration)and asks all the relatives in i a value for the current y_i only if $y_i \leq y_{i-1}$,taking into account their v_r: if $v_r \geq y_{i-1}$ we ask him a value p=y_{i-1} , if $y_i \leq v_r < y_{i-1}$ then we ask him a value p=v_r, otherwise if $v_r < y_i$ we ask him for a value p=y_i 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 (so they have $v_r \geq p$) they will give us a gift of the value of p proposed, otherwise 0.

We sum all the gifts received in the current i, given the current value of y_i , to the value returned by the recursions to which we pass the next i and the current y_i. When all the recursive calls for the current i and y_i returns, we select the global maximum value of the sum of all gifts taken(gifts from all parents in $dist \geq i$) given i and y_i, and we return this value, storing it in $dp[i,y_i]$. Since it is stored, if we enter in a recursive call with the same values i and y_i , 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, $y_o = max(v_r)$.

-the function is called with $max(v_r)$ as y_o to take into account the possibility of having a relative in later distances with a high v_r.
-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 y_i has been previously stored then we do not need to reevaluate the function and we return instantly the value stored.
-otherwise try all possible recursions with all the values y_i in $\{y_{i-1} \cup V_i\}$,return the maximum of those recursions and store the result in $dp[i,y_i]$ as described in the description of the function above.

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 in v_r for every relative and distance ,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 it evaluates the maximum value of all the recursions summed to the gifts in the current i, given a specific i and y_i. Trying all the possible price proposals at each distance and for each relative, we arrive at a global solution that is one of the optimal ones.
Calculating the $max(giftsAtDistancei + recursive(i+1,y_i))$ for each y_i in $\{y_{i-1} \cup V_i\}$, we know that at a given distance i and given a value y_{i-1} we get the optimal solution. Repeating this for each distance i and for each value of y_i allows us to obtain a global optimal solution for i=0 and $y_o = max(v_r)$ *(*so the initial function call*)*.

runtime analysis:
The runtime of the algorithm is $O(C^* r^2) \Rightarrow C^*O(r^2)$
Where C is a constant and r is the number of parents. That's because the cost of the algorithm consists in filling the entire hashmap used for memorization (i^*r) and, for every price proposed, iterating over the parents at distance i. So the total number of iterations are r^*r where the first r identifies the total number of parents and the second one identifies the total number of prices suggested.

1.b)

With reference to the description of the algorithm above, here in the code we have: pr ⇒ y_{i-1} , dist ⇒ i, yi ⇒y_i, relatives ⇒ r_{ij}, maximumvr⇒$max(vr)$, minimumPrices⇒$\{y_{i-1} \cup V_i\}$

```
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):
    dp={}
    if dist>=len(relatives):
        return 0
    maximumGift=0
    if (dist,pr)in dp:
            return dp[(dist,pr)]
    minimumPrices=[pr]+relatives[dist]
    for yi in minimumPrices:
        totalValueAtRow=0
        if yi<=pr:
            for j in range(len(relatives[dist])):
                if relatives[dist][j]>=pr:
                    totalValueAtRow+=pr
                elif yi<=relatives[dist][j]:
                    totalValueAtRow+=relatives[dist][j]
            val=totalValueAtRow+recursive(dist+1,yi)
            if val>maximumGift:
                maximumGift=val
    dp[(dist,pr)]=maximumGift
    return maximumGift
if __name__ == "__main__":
    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)
    print(round(result,1)) #out=116.3
```

The result of the execution of the code with the input of the problem 1.b is: 116.3.

In the implementation, due to the way the floating numbers are represented, we have to round the solution at some decimals: in our case to the first decimal.

Exercise 2:

2a) The network *G(V,E)* is modeled as follows:
Vertices V: $s,t,A=\{a_i, \forall i: i \leq n\}$, $B=\{b_j, \forall j: j \leq n\}$, $V=\{s \cup t \cup A \cup B\}$

Edges E: $E_1=\{(s,a_i), \forall i: i \leq n\}$, $E_2=\{(a_i,b_j) , \forall a_i \text{ and } \forall b_j\}$, $E_3=\{(b_j,t), \forall j: j \leq n\}$, $E=\{E_1 \cup E_2 \cup E_3\}$
Capacities: $\{\forall e \in E_1, c(e)=r_i ; \forall e \in E_2, c(e)=1 ; \forall e \in E_3, c(e)=c_j\}$.
To get the answer for the problem we have to use Ford Fulkerson on the network with source *s* and sink *t*. We used the capacities on the edges $e \in E_1$, $e \in E_3$ to describe the fact that we can't put more than r_i shops in the *i-th* row and more than c_j shops in the *j*-th column. If there is a unit of flow in the edge $e(a_i,b_j)$ so it is saturated ,then there exists a shop in row *i* and column *j*.
Proof of correctness: Given the capacities of *G*, setted to put a limit on the number of shops that can be opened given a specific row, column, or box of the grid representing the city, having a possible number of opened shops greater than the max-flow would end in a flow in the network that will exceed the capacity of at least one of the previous edges, so without respecting the limits given by the problem. The optimum number of shops opened in the city is thus obtained when the number of them is equal to the max flow.
2b)
The network *G(V,E)* is modeled as follows: with *p*=number of groups of 5 rows or columns each:$p=n/5$
Vertices: $s,t,X=\{x_z, \forall z: z \leq p\}$, $A=\{a_i, \forall i: i \leq n\}$, $B=\{b_j, \forall j: j \leq n\}$,$Y=\{y_w , \forall w: w \leq p\}$,$V=\{s \cup t \cup X \cup Y \cup A \cup B\}$

Edges:$E_1=\{(s,x_z) , \forall z: z \leq p\}$, $E_2=\{(x_z,a_i), \forall z: z \leq p \text{ and } \forall i: ((z-1) * 5 + 1) \leq i \leq z * 5\}$,
$E_3=\{(a_i,b_j) , \forall a_i \text{ and } \forall b_j\}$, $E_4=\{(b_j,y_w), \forall w: w \leq p \text{ and } \forall j: ((w-1) * 5 + 1) \leq j \leq w * 5\}$,
$E_5=\{(y_w,t), \forall w: w \leq p\}$, $E=\{E_1 \cup E_2 \cup E_3 \cup E_4 \cup E_5\}$

Capacities: $\{\forall e \in E_1, c(e)=\infty; \forall e \in E_2, c(e)=r_i ; \forall e \in E_3, c(e)=1; \forall e \in E_4, c(e)=c_j; \forall e \in E_5, c(e)=\infty\}$

Lower Bounds= $\{\forall e \in E_1, l(e)=1 ; \forall e \in E_5, l(e)=1\}$

To check if it is possible to solve the problem with the new given constraints (to see if exists a yes instance),we construct a new graph *G'=(V',E')* from *G*. We add new source and sink vertices s' and t', adding edges from *s'* to each vertex $x \in X$ and from s' to *t*. We add edges from each $y \in Y$ to *t'* and from s to t', and finally adding an edge from *t* to *s*. We then define the capacity *c'(e')* of each edge $e' \in E'$ as follows:

For each vertex $x \in X$, we set c'(s', x)=*l(s, x)=1,* for each vertex $y \in Y$, we set *c'(y, t')=l(y, t)=1* ,
we set c'(s', t)=1**p,* c'(s, t')=*1*p* , and at the end we set *c'(t, s)* =∞

We now run the normal Ford Fulkerson and see if *G'* is saturated once the algorithm finish its execution: we check that for every outgoing edges from *s'* and for every ingoing edges to *t'*, the flow that passes through them is equal to the maximum capacities of the edges, saturating so the capacities. If that's the case then we can satisfy the lower bounds in the original network *G* so we have a 'yes' instance , in the other case, the problem can't be solved given the constraints. In case of a 'yes' instance we go back to the original *G* network where for each edge its flow is the sum of its lower bound and the flow found in *G'* ,so *f(e)=f(e')+l(e).* We then construct the residual network with forward edges of capacity *c(e)=c(e)-f(e)* and backward edges of capacity *c(e)=f(e)-l(e)*, in this way we assure that the lower bounds will continue to be satisfied, since eventual augmenting paths will not be able to violate the lower bound on the edges. Then we restart another Ford-Fulkerson on G using the residual network. After that we can get from the algorithm the correct max-flow in G, that, as in point 2a) is the maximum number of shops that can be opened and they are placed in row *i* and column *j* if there is a unit flow passing through the edge $e((a_i,b_j)) \in E_3$.

Exercise 3: The algorithm designed uses a greedy approach: we start taking as much boys as we can given the constraints on k ,b_i and g_i ,and for every school we save how many students we can move to girls.Then for every school, if the total number of males chosen≤the total number of girls chosen +1 we stop since we are at the optimum, otherwise we swap as many students as we can given how many swaps for each school are available and given the difference between the number of males and females chosen.

$\hat{b}$, $\hat{g}$, swapsAvailable all initialized to 0 with size n, sumOfChosenBoys=0,sumOfChosenGirls=0

```
def solver(b,g,k,n):
    for i in range(n):
        if b[i]>=k:
            b̂[i]=k
            sumOfChosenBoys+=k
        else:
            b̂[i]=b[i]
            ĝ[i]=k-b[i]
            sumOfChosenBoys+=b[i]
            sumOfChosenGirls+=k-b[i]
        swapsAvailable[i]= min(g[i]-ĝ[i],b̂[i])
    for i in range(n):
        if sumOfChosenBoys<=sumOfChosenGirls+1:
            break
        val= min(swapsAvailable[i],int((sumOfChosenBoys-sumOfChosenGirls)/2))
        sumOfChosenBoys-=val
        sumOfChosenGirls+=val
        b̂[i]-=val
        ĝ[i]+=val
    return b̂,ĝ
```

proof of correctness:

Given $\left| \sum_{i=1}^{n} \hat{b}_i - \sum_{i=1}^{n} \hat{g}_i \right|$=A , where A is the initial solution obtained maximizing $\hat{b}_i$ given k,b_i and g_i ,

now for every school that we reiterate to optimize A, one of the two of the following conditions will hold:

1) if $\sum_{i=1}^{n} \hat{b}_i \leq (\sum_{i=1}^{n} \hat{g}_i) + 1$ then if we swap one boy with a girl we have that $\left| \sum_{i=1}^{n} \hat{b}_i - \sum_{i=1}^{n} \hat{g}_i \right|$=A or A+2

2) if $\sum_{i=1}^{n} \hat{b}_i > (\sum_{i=1}^{n} \hat{g}_i) + 1$ then if we swap one boy with a girl we have that $\left| \sum_{i=1}^{n} \hat{b}_i - \sum_{i=1}^{n} \hat{g}_i \right|$=A-2

In the first case we will get a worse or equal solution if we swap one student from boys to girls in a school, in the second case the solution instead will be improved by a value of 2. Once we maximized the $\hat{b}_i$ given k, b_i and g_i if chosenBoys<chosenGirls then there is no way to improve the solution since we cannot take another boy because of b_i and k. If chosenBoys>chosenGirls +1 and there are no more swaps available then we cannot improve the solution because of k and g_i .If instead, there are swaps available and they can improve the solution we use them as in the algorithm and, if we can't get a better solution, since we are in the first condition, continuing to swap will increase the absolute difference, so we stop at the optimum.

runtime analysis:The algorithm has a linear cost O(n) because in the worst case scenario we cycle 2 times on the schools, one for maximizing the boys and one for improving the solution till the optimum.

Exercise 4:

To prove that the given problem is NP-complete we have to:
-prove that the problem is in np
-prove that 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 have been 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, which is a NP-complete problem.
All the clauses are in this form $(x_1 \vee x_2 \vee ...)$ or $(\bar{x}_1 \vee \bar{x}_2 \vee ...)$

From the modified SAT create the following instance of the santa problem:
C=set of clauses , $S=\{i: \forall c_i \in C\}$

$K_1=\{i: \forall c_i \in C$, if c_i in the form of $(x_j \vee ...)\}$, $K_2=\{i: \forall c_i \in C$, if c_i in the form of $(\bar{x}_j \vee ...)\}$

$B_j=\{ i : \forall c_i \in C$ such that $x_j \in c_i$ or $\bar{x}_j \in c_i\}$

k=1

The first child will have for each positive clause a sweet i(index of the clause)in its set of preferred sweets.
The second child will have for each negative clause a sweet i(index of the clause)in its preferred sweets set.
We will have a bag B_j for each literal j(index of the literal) present in the clauses.
For each clause $c_i \in C$ and for each literal of that clause $x_j \in c_i$, we will have a candy i (index of the clause) in the bag B_j (index of the literal in that clause).
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 bags chosen by the grinch:
if for the first child the grinch chose the bag i then the literal x_i will have value 1.

if for the second child the grinch chose the bag i then the literal x_i will have value 0.

if a bag i has not been chosen then the literal x_i can be either 0 or 1 (we do not care about its 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 and a certificate verifier, that means that the santa's problem is NP-complete.
example:
$(x_1 \vee x_3 \vee x_4) \wedge (\bar{x}_3 \vee \bar{x}_4 \vee \bar{x}_5) \wedge (\bar{x}_3 \vee \bar{x}_1) \wedge (x_1 \vee x_5) \wedge (\bar{x}_1 \vee \bar{x}_3 \vee \bar{x}_4)$
we construct the instance of the santa problem:
k=1,$K_1=\{1,4\}$ $K_2=\{2,3,5\}$

$B_1=\{1,3,4,5\}$ $B_2=\{\}$ $B_3=\{1,2,3,5\}$ $B_4=\{1,2,5\}$ $B_5=\{2,4\}$

the grinch will choose for example the certificate: $\widehat{B}(1)=\{B_1\}$ $\widehat{B}(2)=\{B_3,B_5\}$
the certificate for the sat modified problem is then x_1=1,x_3=0,x_5=0,x_2=(0 or 1),x_4=(0 or 1)
Since the grinch was able to find a solution, there exists a solution for the modified SAT problem.