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 the relatives in distance i.

We created a recursive function that for every distance i of the parents tries all possibles y_i from 0 *to* y_{i-1} (that was passed to the function in the previous iteration) and asks all the relatives in i a value between the current y_i and y_{i-1} , taking into account their v: if $v \geq y_{i-1}$ we ask him a value p=y_{i-1} , if $y_i \leq v < y_{i-1}$ then we ask him a value p=v, otherwise if $v < y_i$ 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 $v \geq p$) 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 y_i , to the value returned by the recursion 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 recursion 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 but we return instantly the value stored.
-otherwise try all possible recursions with all the values y_i ,return the maximum of those and store the result in $dp[i,y_i]$ 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 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 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 $max(giftsAtDistancei+recursive(i+1,y_i))$ for each y_i in $\{0...y_{i-1}\}$, 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 $y_o = max(v_r)$

analysis of the runtime:
The runtime of the algorithm is $O(C^* r * max(v_r))$
Where C is a constant, r is the number of parents and $max(v_r)$ is the maximum of all the v of the relatives.
In the worst case scenario $max(v_r)$ is evaluated in every iteration, so the cost is $C^*O(r*max(v_r))$ that is the cost to fill the entire hashmap and evaluate all the gifts for every i and y_i, multiplied by a constant value c for every evaluation of the function.
In the code implementation it is also multiplied by a factor (10 in our case) that describes the granularity of the result (float numbers or approximations).

1.b

With reference to the description of the algorithm above, here in the code we have: pr = y_{i-1} , dist is i, i is y_i, relatives is r_{ij}

```
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_r 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 1.b is: 116.3.

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, so without respecting the limits given by the problem. The optimum number of shops opened in the city is then 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:$E_1=\{(s,x_z)\ \forall z: z \leq p\}$, $E_2=\{(x_z,a_i)\ \forall z: z \leq p \text{ and } \forall i: ((z-1)*5+1) \leq i \leq z*5\}$,

$E_3=\{(a_i,b_j)\ \forall a_i \text{and } \forall b_j\}$, $E_4=\{(b_j,y_w)\ \forall w: w \leq p \text{ and } \forall i: ((w-1)*5+1) \leq i \leq w*5\}$,

$E_5=\{(y_w,t)\ \forall w: w \leq p\}$, $E=\{E_1 \cup E_2 \cup E_3 \cup E_4 \cup E_5\}$

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

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* by adding new source and sink vertices s' and t', adding edges from *s'* to each vertex $x \in X$ and from s' to *t,* adding 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 $c'(s' \to x)=l(s \to x)=1$, for each vertex $y \in Y$, we set $c'(y \to t')=l(y \to t)=1$,
we set $c'(s' \to t)=1*p$, $c'(s \to t')=1*p$, and at the end we set $c'(t \to s)=\infty$

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*, 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 we sum its lower bound to its flow found in the previous step 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 be satisfied, since eventual augmenting paths could not use that flow that will be preserved for the edge. Then we restart another Ford-Fulkerson on the residual network and, once the FF finished, for each edge we update the flow value as $f(e)=f(e)+l(e)$. After that we can get from the algorithm the correct max-flow, that, as in point 2a) will be equal to the number of shops that will be opened in row *i* and column *j* if there is a unit flow passing through the edge $e((a_i,b_j)) \in E_3$.

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) that is the min of *|boysChosen - girlsChosen|*.
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.
Starting from i=0, boysChosen= 0, girlsChosen= 0. Given n= list of schools, i = index of schools, dp = hashmap for memorization

```
function makeSelection ( i, boysChosen, girlsChosen):
    if i>=len(n):
        return |boysChosen - girlsChosen|
    if  (boysChosen,girlsChosen) in dp :
        return dp[(boysChosen,girlsChosen)]
    l=allValidCombinations(bi,gi,k)
    bestSolution = ∞
    for bi , gi in l:
        possibleSolution= makeSelection(i+1,boysChosen+bi ,girlsChosen +gi)
        if possibleSolution < bestSolution:
            bestSolution = possibleSolution
        dp[(boysChosen,girlsChosen)]=bestSolution
        return bestSolution

function allCombinations(bi,gi,k):
    return a list of all possible combinations of gi and bi given k.  #(there are at most k combinations)
```

proof: 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 boysChosen and girlsChosen we return the max(makeSelection(i+1,boysChosen+b_a,girlsChosen+g_a)for every b_a,g_a in *allCombinations*(b_i,g_i,*k*).
So if we have a maximum value for every boysChosen and girlsChosen, then makeSelection of (0,0,0) will return the global maximum .The memorization works because in a iteration, if a recursive function has already calculated a best solution for the same number of boys and girls till that moment,then will return that value without re-evaluating the recursive function, since it will end up with the same best solution. Moreover we didn't add *i* to the memorization since for each *i* and *i+1* we will have a total number of students chosen that will increase by *k* so there is no possibility that at two different schools we will end up with the same number of girls chosen and boys chosen .

The cost of the algorithm is $O(C*k* n *k*n)=O(C * n^2 * k^2)=C*O(n^2 * k^2)$ with C that is a constant.That is 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 $C=O(n^2 * k^2)$ but in the worst case scenario it will be $O(n^3 * k^2)$. Anyway, in every case, the algorithm will run in polynomial time.

Exercise 4:

To prove that the given problem is NP-complete we have to:
-prove that the problem is in np
-prove that 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, 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
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={B_j : $\forall\, c_i \in C$ in the form of $(x_j \vee ...)$or in the form of $(\bar{x}_j \vee ...) : i \in B_j$ }
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 chosen bags by the grinch:
if for the first child the grinch chose the bag i then the x_i has to be a 1.
if for the second child the grinch chose the bag i then x_i has to be a 0.
if a bag i has not been chosen then 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.