We solved this first exercise with a Dynamic Programming algorithm; in fact, we broke the problem into sub-problems and stored the result of sub-problems to avoid the computing same results again. The core is the recursive function *OPT* (shown below) which takes as parameter the "layer" (which is composed by all the relatives at a certain distance), and an upper bound for the values to be taken of all subsequent layers. The algorithm sorts the nodes (relatives) in the layers by crescent values v_r, then it invokes the *OPT* $(layer_1, v_{max})$, where v_{max} is the highest v_r (highest possible upper bound). The recursive calls and the use of the "max" function, allow the algorithm to compute the maximum possible accumulated value.

$OPT(layer_i,\ currentMin) = \{$

$0 \qquad if(i > numLayers)$

$currentMin * numNodesLayer_i + OPT(layer_{i+1}, currentMin) \qquad if(currentMin < v_{i,1})$

$max(v_{i,1} + currentMin * (numNodesLayer_i - 1) + OPT(layer_{i+1}, v_{i,1}),$
$\quad currentMin * (numNodesLayer_i - 1) + OPT(layer_{i+1}, currentMin)) \qquad if(currentMin < v_{i,2})$

$\dots\dots\dots\dots\dots \qquad \dots\dots\dots\dots\dots$

$max\ (\dots\dots\dots\dots\dots\dots\dots\dots\dots\dots\dots\dots\dots\dots\dots\dots\dots\dots\dots\dots\dots\dots\dots\dots) \qquad if(currentMin < v_{i,numNodesLayer_i})$

$max\ (v_{i,1} + v_{i,2} + \dots + v_{i,numNodesLayer_i} + OPT(layer_{i+1}, v_{i,1}), \qquad if(currentMin \geq v_{i,numNodesLayer_i})$

$\quad v_{i,2} + \dots + v_{i,numNodesLayer_i} + OPT(layer_i + 1, v_{i,2}),$

$\quad \dots,$

$\quad v_{i,numNodesLayer_i} + OPT(layer_i + 1, v_{i,numNodesLayer_i}),$

$\quad OPT(layer_i + 1, currentMin))$

$\}$

Correctness:

Theorem: The algorithm returns the maximum possible accumulated value

Proof: The algorithm is based on the recursive function *OPT* which is invoked on a specific layer i and with a $currentMin$ upper bound for the values. The function returns the maximum accumulated value between all the possible combinations for the given layer and upper bound. Each of these combinations leads to another call of the function of the subsequent layer (top-down approach) with the respective upper bound. The base case (when $i > numLayers$) guarantees the termination by returning 0. Therefore, by first invoking the function on the first layer with the highest possible bound, this recursion leads to every layer/bound combination feasible for the current instance. Finally, by returning always the best possible choice (with the "max" function) for each layer, the return of the first call will be the maximum possible accumulated value.

Running Time:

The n nodes (relatives) are distributed in the layers (levels of distance) as $n/f(i)$, with i as number of layers. For simplicity let's consider $f(i) = i$, which means that the nodes are equally distributed in the layers. When first calling *OPT* function on the closest layer, it does $\left(\frac{n}{i}\right) + 1$ calls of itself on the next layer. Then each of these calls does, in the worst case (when the bound is higher than the highest value in the layer), $(\left(\frac{n}{i}\right) + 1) + (\left(\frac{n}{i}\right) + 1)$ calls of itself on the next layer, thanks to the *memoization*. This recursion continues until the "next layer" doesn't exists. By analysing the complexity of each call and multiplying it for the max number of calls, we can get the complexity of the algorithm (the sorting cost is negligible). Each call cost is dominated, in the worst case, by a cycle over n/i followed by another cycle over n/i (constants irrelevant for the complexity has been removed for legibility). Regards the calls with the layer after the last as input, their cost is constant. Therefore, the total complexity is:

$$\left(\sum_{k=1}^{i}((\left(\left(\frac{n}{i}\right) + 1\right) * k) * \left(\frac{n}{i} * \frac{n}{i}\right))\right) + (\left(\left(\frac{n}{i}\right) + 1\right) * (i + 1))$$

Which is:

$$O\left(\left(\frac{n}{i}\right)^3\right)$$

Python code in the next sheet

```python
import numpy as np
import functools

layerList = [[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]] #list nodes for each layer

def getMaxValue(): #get the max possibile value
  maxValue = 0
  for i in range(len(layerList)):
    maxValue = max(maxValue, layerList[i][len(layerList[i])-1])
  return maxValue

def sortNodes(): #sort nodes in ascending order for each layer
  for i in range(len(layerList)):
    layerList[i] = np.sort(layerList[i], kind="mergesort")

@functools.cache #using a cache to store results
def maximiseGifts(layer, currentMin):
  layer_result = [] #layer local results
  flag = False
  if layer < len(layerList):
    for i in range(len(layerList[layer])): #for all nodes in the layer
      if (not flag):
        if (currentMin < layerList[layer][i]): #find an element that is bigger than currentMin
          for k in range(i+1): #from pos 0 to i
            result = 0
            if (k < i):
              for j in range(k, i): #nodes from pos k to pos i-1
                result = result + layerList[layer][j]
              layer_result.append(result + (currentMin*(len(layerList[layer])-(i)) + maximiseGifts(layer + 1, layerList[layer][k])))
            else:
              layer_result.append(currentMin*(len(layerList[layer])-(i)) + maximiseGifts(layer + 1, currentMin))
            flag = True #exit the cicle after the first satisfied element
        elif (currentMin >= layerList[layer][len(layerList[layer])-1]): #currentMin bigger than all other layer nodes
          for k in range(len(layerList[layer])+1): #from pos 0 to last + 1
            result = 0
            if (k <= (len(layerList[layer])-1)):
              for j in range(k, len(layerList[layer])): #from pos k to last
                result = result + layerList[layer][j]
              layer_result.append(result + maximiseGifts(layer + 1, layerList[layer][k]))
            else:
              layer_result.append(maximiseGifts(layer + 1, currentMin)) #refuse all smaller nodes in the layer
            flag = True #exit the cicle after the first satisfied element
    print(' layer: ' + str(layer) + ' currentMin: ' + str(currentMin) + ' result: ' + str(layer_result))
    return max(layer_result) #return the max aggregated value
  else:
    return 0

if __name__ == "__main__":
  sortNodes()
  max_gift = getMaxValue()
  print(' max possible accumulated value: ' + "{:.1f}".format(maximiseGifts(0, max_gift)))
  print(maximiseGifts.cache_info()) #print cache info
```

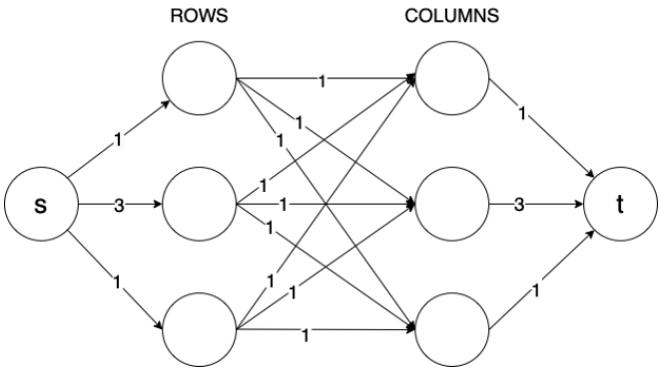
```
 layer: 4 currentMin: 2.0 result: [7.0, 6.0]
 layer: 4 currentMin: 2.9 result: [8.9, 7.9, 5.8]
 layer: 4 currentMin: 3.5 result: [10.1, 9.1, 7.0]
 layer: 4 currentMin: 4.9 result: [12.6, 11.6, 9.5, 4.9]
 layer: 4 currentMin: 5.2 result: [12.899999999999999, 11.899999999999999, 9.8, 5.2]
 layer: 3 currentMin: 5.2 result: [20.3, 20.200000000000003, 18.5, 17.5, 12.899999999999999]
 layer: 4 currentMin: 5.6 result: [13.299999999999999, 12.299999999999999, 10.2, 5.6]
 layer: 3 currentMin: 5.6 result: [20.3, 20.200000000000003, 18.5, 17.5, 13.299999999999999]
 layer: 4 currentMin: 6.3 result: [14.0, 13.0, 10.899999999999999, 6.3]
 layer: 3 currentMin: 6.3 result: [20.3, 20.200000000000003, 18.5, 17.5, 14.0]
 layer: 4 currentMin: 7.5 result: [14.1, 13.1, 11.0, 6.4, 0]
 layer: 3 currentMin: 7.5 result: [20.3, 20.200000000000003, 18.5, 17.5, 14.1]
 layer: 2 currentMin: 7.5 result: [44.900000000000006, 39.7, 34.1, 27.8]
 layer: 4 currentMin: 9.4 result: [14.1, 13.1, 11.0, 6.4, 0]
 layer: 3 currentMin: 9.4 result: [20.3, 20.200000000000003, 18.5, 17.5, 14.1]
 layer: 2 currentMin: 9.4 result: [46.800000000000004, 41.6, 36.0, 29.700000000000003]
 layer: 1 currentMin: 9.4 result: [80.60000000000001, 75.0]
 layer: 4 currentMin: 9.7 result: [14.1, 13.1, 11.0, 6.4, 0]
 layer: 3 currentMin: 9.7 result: [20.3, 20.200000000000003, 18.5, 17.5, 14.1]
 layer: 2 currentMin: 9.7 result: [47.1, 41.9, 36.3, 30.0]
 layer: 4 currentMin: 10.2 result: [14.1, 13.1, 11.0, 6.4, 0]
 layer: 3 currentMin: 10.2 result: [20.3, 20.200000000000003, 18.5, 17.5, 14.1]
 layer: 4 currentMin: 10.8 result: [14.1, 13.1, 11.0, 6.4, 0]
 layer: 3 currentMin: 10.8 result: [20.3, 20.200000000000003, 18.5, 17.5, 14.1]
 layer: 2 currentMin: 10.8 result: [47.6, 42.4, 36.8, 30.5, 20.3]
 layer: 4 currentMin: 11.1 result: [14.1, 13.1, 11.0, 6.4, 0]
 layer: 3 currentMin: 11.1 result: [20.3, 20.200000000000003, 18.5, 17.5, 14.1]
 layer: 2 currentMin: 11.1 result: [47.6, 42.4, 36.8, 30.5, 20.3]
 layer: 1 currentMin: 11.1 result: [84.0, 78.7, 69.5, 58.7]
 layer: 4 currentMin: 12.2 result: [14.1, 13.1, 11.0, 6.4, 0]
 layer: 3 currentMin: 12.2 result: [20.3, 20.200000000000003, 18.5, 17.5, 14.1]
 layer: 2 currentMin: 12.2 result: [47.6, 42.4, 36.8, 30.5, 20.3]
 layer: 4 currentMin: 15.2 result: [14.1, 13.1, 11.0, 6.4, 0]
 layer: 3 currentMin: 15.2 result: [20.3, 20.200000000000003, 18.5, 17.5, 14.1]
 layer: 2 currentMin: 15.2 result: [47.6, 42.4, 36.8, 30.5, 20.3]
 layer: 1 currentMin: 15.2 result: [85.10000000000001, 79.80000000000001, 70.6, 59.8, 47.6]
 layer: 0 currentMin: 15.2 result: [116.30000000000001, 110.3, 100.30000000000001, 85.10000000000001]
 max possible accumulated value: 116.3
CacheInfo(hits=111, misses=56, maxsize=None, currsize=56)
```

a)

To solve this point, we defined a flow network based on a directed graph (definition shown below), and applied Ford Fulkerson algorithm on it. To ensure the constraint on the rows and columns, we limited the capacities of the edges from *s* to the rows with the respective r_i; and the capacities of the edges from columns to *t* with the respective c_i. Of course, we set to 1 the capacities of the internal edges since there could be only one shop in a specific block of the grid.

Model definition:

- create a node ROW_i for each row of blocks i
- create a node COL_j for each column of blocks j
- add source s and sink t
- add directed edge (s, ROW_i) from node s to each node ROW_i with capacity equal to r_i
- add directed edge (ROW_i, COL_j) from each node ROW_i to each node COL_j with capacity 1
- add directed edge (COL_j ,t) from each node COL_j to node t with capacity equal to c_j



Example instance

Correctness:

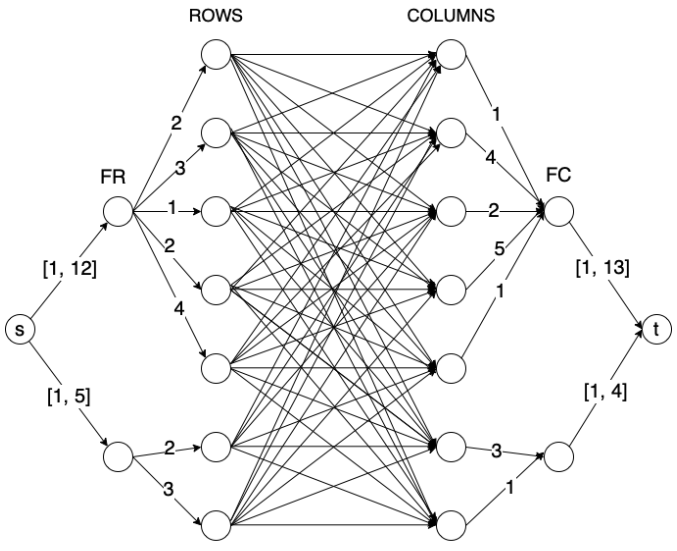
Ford Fulkerson applied to the previous model guarantees the max flow in the network; while capacities of the model guarantees that the constraints of the problem are respected. If we see the flow from *s* to a row i as the number of shops in that row, and the flow from a column *j* to *t* as the number of shops in that column, maximizing the network flow correspond to maximizing the number of shops in the city. In fact, a path that goes through row i and column *j* represents a shop in position (i, j) of the grid.

b)

To solve this point, we modified the previous graph by adding some nodes and edges. These changes (shown in the graph definition below) allowed us to model the new constraint over rows and columns. Moreover, we introduced a variation in the Ford Fulkerson algorithm to guarantee the correctness of the new decision problem. The variation is about the way in which Ford Fulkerson algorithm choose the augmenting paths, indeed we forced it to prioritize, in the choice, paths which have two edges with a lower bound and no flow (one from the rows side, one from the columns side). If our new algorithm doesn't find at least X paths with those characteristics, with X equal to $n/5$ rounded to the next integer, returns "No", otherwise proceed with the regular Ford Fulkerson until termination, without the possibility to choose reverse edges from an FR_k to s and from t to an FC_h, then returns "Yes".

Model definition:

- create a node ROW_i for each row of blocks i
- create a node COL_j for each column of blocks j
- create a node FR_k for each existing (even partially) subgroup of rows {k, 1+k, 2+k, 3+k, 4+k} with k that starts from 1 and increase by 5 for each node FR_k
- create a node FC_h for each existing (even partially) subgroup of columns {h, 1+h, 2+h, 3+h, 4+h} with h that starts from 1 and increase by 5 for each node FC_h
- add source s and sink t
- add directed edge (FR_k, ROW_i) from each node FR_k to each existing node ROW_i, with i from k to k+4, and with capacity equal to r_i
- add directed edge (ROW_i, COL_j) from each node ROW_i to each node COL_j with capacity 1
- add directed edge (COL_j, FC_h) from each existing node COL_j to each node FC_h, with *j* from h to h+4, and with capacity equal to c_j
- add directed edge (s, FR_k) from node *s* to each node FR_k, with capacity equal to $[1, \sum_{i=k}^{k+4} r_i]$
- add directed edge (FC_h, t) from each node FC_h to node *t*, with capacity equal to $[1, \sum_{j=h}^{h+4} c_j]$



Example instance

Correctness:

The algorithm should return "No" when is not possible in any way to respect the new constraint on rows and columns. This happens when iff there is at least one subgroup of rows or columns, connected to an FR_k or an FC_h, which has all zero capacities. Indeed, in that case our Ford Fulkerson variation terminates and returns "No". Otherwise, the algorithm proceeds with regular Ford Fulkerson, without the possibility to choose reverse edges from an FR_k to s and from t to an FC_h; therefore, the new constraint couldn't be lost once respected. All the considerations done for the point 2.a) about Ford Fulkerson still holds, thus, in this case the algorithm would maximize the number of shops and then return "Yes".

Our algorithm, that computes the $b_i^{\hat{}}$ and $g_i^{\hat{}}$ for each school *i,* is based on two main cycles over the given schools. The goal of the first cycle is to select *k* students from each school; the selection could be done in any way (randomly, balancing …), what is important is to store which is the total number of boys and girls selected. In the pseudocode, the first cycle gives the priority to the boys for the selection. After the first cycle, the algorithm checks which is the gender in advantage up to this point and computes the difference (in module) between the two totals. The second cycle is the most important. It scans the schools until the difference between the totals is less than 1, or until the schools are over. In each iteration the difference is reduced, because the students selected in the first cycle are changed in favour of the disadvantaged gender. At the end of the second cycle, the $b_i^{\hat{}}$ and $g_i^{\hat{}}$ for each school *i,* succeed in minimizing $|\sum_{i=1}^{n} g_i^{\hat{}} - \sum_{i=1}^{n} b_i^{\hat{}}|$.

Pseudocode:

$Boys = 0$
$Girls = 0$
$\boldsymbol{for\ each}\ i \in Schools\ \mathbf{do}$:
    $\boldsymbol{if}\ b_i \geq k\ \boldsymbol{do}$:
        $b_i^{\hat{}} = k$
        $g_i^{\hat{}} = 0$
    $\boldsymbol{else}$:
        $b_i^{\hat{}} = b_i$
        $g_i^{\hat{}} = k - b_i^{\hat{}}$
    $Boys = Boys + b_i^{\hat{}}$
    $Girls = Girls + g_i^{\hat{}}$

$\boldsymbol{if}\ Boys \geq Girls\ \boldsymbol{do}$:
    $MalesAdvantage = true$
$Diff = |Boys - Girls|$

$\boldsymbol{for\ each}\ i \in Schools\ \mathbf{do}$:
    $\boldsymbol{if}\ Diff > 1\ \boldsymbol{do}$:
        $\boldsymbol{if}(MalesAdvantage == true)\ \boldsymbol{do}$:
            $MaxTrade = \boldsymbol{min}(b_i^{\hat{}}, (g_i - g_i^{\hat{}}), Diff/2)$ #Diff/2 is rounded down
            $b_i^{\hat{}} = b_i^{\hat{}} - MaxTrade$
            $g_i^{\hat{}} = g_i^{\hat{}} + MaxTrade$
        $\boldsymbol{else}$:
            $MaxTrade = \boldsymbol{min}(g_i^{\hat{}}, (b_i - b_i^{\hat{}}), Diff/2)$ #Diff/2 is rounded down
            $b_i^{\hat{}} = b_i^{\hat{}} + MaxTrade$
            $g_i^{\hat{}} = g_i^{\hat{}} - MaxTrade$
        $Diff = Diff - (MaxTrade * 2)$
    $\boldsymbol{else}$:
        $\boldsymbol{break}\ out\ of\ the\ for\ each$

Correctness:

(Notation: "Diff" = "gender difference w.r.t all schools")

Let's analyse all the cases in which the algorithm terminates after the second cycle:

- if Diff = 0, the solution is optimal because k students have been selected from each school and there is no difference between the genders.
- if Diff = 1, the solution is optimal because k students have been selected from each school and there is no possibility for this instance to get difference 0 by switching one or more couples of students.
- if Diff > 1 (which means that the second cycle is terminated because it iterated over all the schools), the solution is optimal:

By contradiction, if we say that it is not optimal, it means that there is another optimal solution which leads to a lesser Diff. This other optimal solution should differ from our solution by at least one couple $(b_i^{\hat{}}, g_i^{\hat{}})$. Although, considering that in the second cycle all the $(b_i^{\hat{}}, g_i^{\hat{}})$ selections were made to reduce the Diff as much as possible; changing one or more couples $(b_i^{\hat{}}, g_i^{\hat{}})$ would lead to a Diff greater or equal to the one computed by our algorithm, therefore our solution is the optimal.

Running Time:

The analysis is extremely simple because the core of the algorithm are the two cycles, each with *n* iterations, in which only constant time operation are made. The total complexity is $O(n)$ for the first cycle where we scan all the n schools plus $O(n)$ for the second cycle where we scan again all the n schools to minimize the difference. Therefore, the complexity is linear: $O(n)$.

To prove that our problem X is NP-Complete, we must start by proving that it is NP; it is indeed a decision problem for which we can find:
- a *certificate* polynomial in the size of the problem instance, defined by the array of $\mathrm{B}^{\wedge}(j) \subseteq \{B_1, B_2, \dots, B_n\}$, with one element for each kid *j*.
- a *certifier* that verifies a certificate in time polynomial in the size of the problem instance. The certifier algorithm takes the problem instance and a certificate as input, then check, with a simple scan, if each $\mathrm{B}^{\wedge}(j)$ satisfies the respective constraint on the K_j; moreover, while scanning, the algorithm counts the occurrences of the bags B_i to check the constraint on *k.* If a constraint is not respected the certifier returns "No", otherwise if the scan ends and the constraints are respected, it returns "Yes".

Now that we have proved that our problem $X \in$ NP, we must consider an NP-complete problem Y and prove that $Y \leq_p X$. After that we will conclude that also X is NP-complete.

For the NP-complete reduction we have chosen, as problem Y, a version of SAT where every clause has either all positive or all negative literals, which is NP-complete indeed.

Therefore, given an instance ϕ of Y, we must build an ad hoc instance of X, which has solution if and only if ϕ is satisfiable:
- Consider a sweet S_h for each clause C_h of the SAT formula
- Consider a bag B_i for each literal x_i of the SAT formula
- Consider only one kid, with K_1 = set of sweets corresponding to the *positive* clauses (with all positive literals)
- Consider k=1

Proof:

(=>) Given a solution of our problem, let's show that ϕ is satisfiable:
The solution to our problem would be the $B^{\wedge}(1)$, which is the set of bags associated to the kid. Let's set to True all the literals x_i that corresponds to the bags B_i that are contained in $B^{\wedge}(1)$. Then let's set to False all the remaining literals (that corresponds to the remaining bags). This assignment satisfies the ϕ.

(<=) Given an assignment of ϕ, let's show that our problem admit solution:
An assignment of ϕ is defined by the Boolean values associated with the literals. The solution that our problem admits will be the $B^{\wedge}(1)$ composed by the B_i associated to the True literals.

Example:

$\phi = (x_1 \vee x_2) \wedge (\bar{x}_1 \vee \bar{x}_3) \wedge (x_1)$ (satif. If $(x_1 = true,\ x_2 = true,\ x_3 = false)$ or if $(x_1 = true,\ x_2 = false,\ x_3 = false)$)
- Construction:
  $S = \{S_1, S_2, S_3\}$
  $B_1 = \{S_1, S_2, S_3\}$
  $B_2 = \{S_1\}$
  $B_3 = \{S_2\}$
  k = 1
  $K_1 = \{S_1, S_3\}$
- Solution:
  $B^{\wedge}(1) = \{B_1, B_2\}$ or $B^{\wedge}(1) = \{B_1\}$