Algorithm Design
Homework 1

Brunetti Jacopo 1856271
Carmignani Federico 1845479

December 5, 2021

1 Exercise 1

Brunetti, Carmignani

1.a) The algorithm

The problem requires the construction of an algorithm which, given the values that each relative is willing to give, maximizes the sum of the revenues received, keeping into account the given constraints. In order to solve the problem, it was implemented an algorithm using the approach known as **dynamic programming** by which the problem is divided into easier problems, incrementally constructed, that lead to the final optimal solution of the whole problem. In order to implement the algorithm, it was important to define a structure to store the optimal values found. This structure is an $l \cdot s$ matrix, where l are the degrees of kinship of the relatives, each of these are represented by a row in the matrix, from the highest to the lowest one, whereas s defines the cardinality of the set S composed by all the different values in all the levels of kinship, representable by a list of lists v, for each of these a column is created, from the lowest to the highest one. The optimal value *OPT(lev, v_r)* is defined as the max value that is possible to find at the level *lev* imposing as roof value v_r, meaning it as the max value obtainable in *lev*. The dynamic trick is that, in order to calculate this optimal value, the algorithm exploits the values of the previous row, trying in this way all the possible floor values, considering a floor value as the min value for a certain level, together with the same roof value defined by the column, at level *lev*, and memorizing the max value obtained from all the possible sums. So, the best possible choice is considered, supposing to have on the level *lev* the upper bound v_r, and it is repeated for each possible upper bound at that level, that is every element in S. This procedure is finally repeated for each level of kinship, constructing incrementally the final solution as the max of the last row of the matrix.

$$\textbf{OPT (lev, } \mathbf{v_r}\textbf{)} = \begin{cases} find\_opt(v[lev], 0, v_r) & \text{if } lev = \text{len}(v)\text{-1} \\ max_{\forall f \in S \leq v_r}(OPT((lev+1), f) + find\_opt(v[lev], f, v_r)) & else \end{cases} \tag{1}$$

$$find\_opt(v[lev], floor, roof) = \sum_{i \in v[lev]} \begin{cases} 0 & if\ floor > i \\ i & if\ floor \leq i \text{ AND } i \leq roof \\ roof & if\ floor \leq i \text{ AND } i > roof \end{cases} \tag{2}$$

Algorithm 1

Require: $v \leftarrow$ *list of lists containing the max values that each relative is willing to give for each level of kinship.*

procedure Ex_1(v):

Step 1: in order to build the *matrix[l][s]*, it is necessary to create the ordered set S containing all the different values that all the relatives are willing to give, without repetitions.

Step 2: in order to fill the first row of the matrix, each element *matrix[0][j]* is the return value of the *find_opt* procedure, passing as parameters the last list of *v*, the floor value as *0* and the roof value as *S[j]*.

Step 3: for each element that is not in the first row of the matrix, the algorithm calculates the optimal value using the previous matrix row. In case of the element *matrix[i][j]*, it calculates the max value that is possible to obtain summing each element of the previous row, whose column is less or equal than *j*, with the highest obtainable value, returned by *find_opt*, in the level i given as roof value *j* and as floor value the column of the element of the previous row considered.

Require: $v[lev] \leftarrow$ *the list of the level lev*, $floor\_value$, $roof\_value$.

procedure FIND_OPT($v[lev]$, $floor\_value$, $roof\_value$):

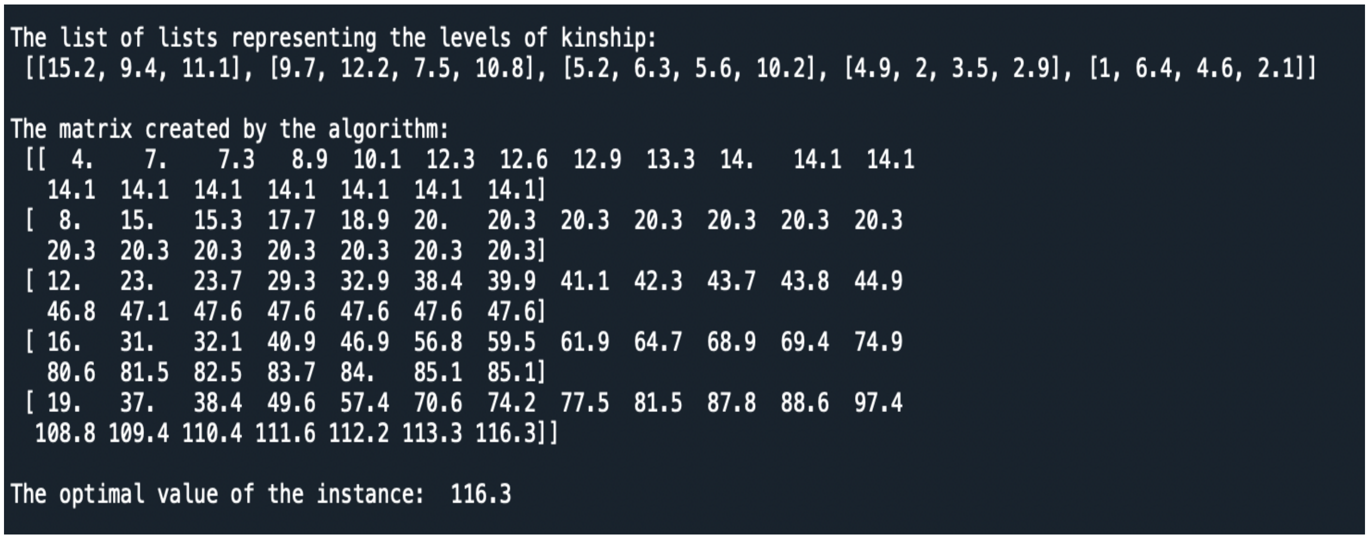
It returns the max value that can be obtained as sum in the level *lev*, taking for each element *el* in *v[lev]*: 0 if *el* is less than the floor value, *el* if it is between the the floor and the roof value, else taking the roof value if the value *el* is higher than the roof value.

1.a) The proof of correctness and the running time

The algorithm so first computes the first row of the matrix, considering the last level of kinship, and so trying to take the highest value obtainable in this level having all possible roof values and a floor values as 0. Then it computes the second row trying to understand which is the highest sum that is obtainable from the last and second last levels, having in second last a fixed roof value, equal to the column of the element to fill, and one of the possible lower bounds, which would be the roof values of the last level, and so considering the elements of the previous row in the matrix whose columns represent this lower bound; it is done adding, to the max value obtainable in the second last with this constraints, the value of the element considered each time in the previous row. Therefore, at the end of this iteration, the second row of the matrix will contain the max obtainable values imposing at the second last level every possible upper bound; repeating this until the first level, the last row of the matrix will contain the max values obtainable imposing at the first level every possible upper bound. The max of the last row will be necessarily, **by induction**, the **optimal solution** for the problem. Since the lower bound of a level is the upper bound of the next one, and they are transferred from the last level, the floor values and the roof values are taken from the same set, which is S. Each element of the $l \cdot s$ elements of the matrix, where l is the number of levels, is computed using all the elements of the previous row, therefore it takes a cost of *O($l \cdot s \cdot s$)*. But, the *find_opt* procedure, which is called for each element, costs *O(n)*, since it checks all the values in a level. So, the **running time** of the algorithm is *O($l \cdot s \cdot s \cdot n$)*, or rather *O($l \cdot s^2 \cdot n$)*, and knowing that s is *O(n)*, then it is *O($l \cdot n^3$)*. Since l is *O(n)*, the algorithm is **polynomial** in the size of the input n.

```
import numpy as nup #library to create the matrix

def find_opt(l, floor, roof): #it finds the max value obtainable at a level given floor and roof values
    mx=0
    for el in l:
        if (el>=floor):
            if (el<=roof):
                mx+=el
            else:
                mx+=roof
        else:
            mx+=0
    return mx

def ex_1(v): #v is the list of lists representing all the levels of kinship
    print("\nThe list of lists representing the levels of kinship: \n", v)
    #step 1
    l=[]
    for el in v:
      l+=el
    sett=set(l) #the set of the elements of all the levels, without repetitions, is created
    l=list(sett)
    l.sort() #l is the sorted set of elements
    n=len(l) #n is the cardinality of l
    num_lvl=len(v) #num_lvl is the number of levels
    matrix=nup.zeros((num_lvl,n)) #the matrix[num_lvl][n]

    #step 2
    for j in range(0,n): #filling the first row (related to last level)
        matrix[0][j]=find_opt(v[num_lvl-1],0,l[j])

    #step 3
    for i in range(1,num_lvl): #filling the other rows (related to the other levels going up from the last one)
        actual_lvl = num_lvl-1-i
        for k in range(0,n): #considering the element matrix[i][k]
            tmp=[]
            for f in range(0,k+1): #using the previous row to compute all the sums for element matrix[i][k]
               tmp.append(matrix[i-1][f]+find_opt(v[actual_lvl ],l[f],l[k]))
            matrix[i][k]=max(tmp) #setting it with the maximum of the sums

    print("\nThe matrix created by the algorithm: \n", matrix)
    print("\nThe optimal value of the instance: ", round(max(matrix[num_lvl-1]),2))

#solving a particular instance of the problem
ex_1([[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]])
```

1.b) The output of a particular instance



```
The list of lists representing the levels of kinship:
 [[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]]

The matrix created by the algorithm:
 [[  4.    7.    7.3   8.9  10.1  12.3  12.6  12.9  13.3  14.   14.1  14.1
   14.1  14.1  14.1  14.1  14.1  14.1  14.1]
 [  8.   15.   15.3  17.7  18.9  20.   20.3  20.3  20.3  20.3  20.3  20.3
   20.3  20.3  20.3  20.3  20.3  20.3  20.3]
 [ 12.   23.   23.7  29.3  32.9  38.4  39.9  41.1  42.3  43.7  43.8  44.9
   46.8  47.1  47.6  47.6  47.6  47.6  47.6]
 [ 16.   31.   32.1  40.9  46.9  56.8  59.5  61.9  64.7  68.9  69.4  74.9
   80.6  81.5  82.5  83.7  84.   85.1  85.1]
 [ 19.   37.   38.4  49.6  57.4  70.6  74.2  77.5  81.5  87.8  88.6  97.4
  108.8 109.4 110.4 111.6 112.2 113.3 116.3]]

The optimal value of the instance:  116.3
```

2 Exercise 2

Brunetti, Carmignani

2.a) The flow network, the algorithm and the proof of correctness

In order to solve the problem, it is possible to model the city through an *n x n* matrix in which each cell *matrix[i][j]* represents a block in which it is allowed to open a shop. The goal is to find the max number of shops that can be legally opened in the city, and legally means that the constraints of the problem are satisfied. The problem can be modeled as a **flow network** F:

$$\mathbf{F} = \{\mathbf{V}, \mathbf{E}, \mathbf{c}: \text{E} \rightarrow \mathbb{R}\}$$

In particular, the set of nodes V, the set of edges E and a function c to add capacities to the edges:

- **V** = {s, t, row$_i$ $\forall$ i $\in$ [1,n], col$_j$ $\forall$ j $\in$ [1,n]}
- **E** = {(s, row$_i$) $\forall$ i $\in$ [1,n], (col$_j$, t) $\forall$ j $\in$ [1,n], (row$_i$, col$_j$) $\forall$ i $\in$ [1,n] $\forall$ j $\in$ [1,n]}
- **c(E)** = {r$_i$ $\forall$ e $\in$ E s.t. e = (s, row$_i$) $\forall$ i $\in$ [1,n], 1 $\forall$ e $\in$ E s.t e = (row$_i$, col$_j$) $\forall$ i $\in$ [1,n] $\forall$ j $\in$ [1,n], c$_j$ $\forall$ e $\in$ E s.t. e = (col$_j$, t) $\forall$ j $\in$ [1,n]}

Then, applying the **Ford-Fulkerson algorithm** to the flow network F, it will find the max feasible flow in F, from s to t. The max feasible flow is equal to the max number of shops which can be legally opened in the city since every augmenting path adds one unit of flow representing the opening of a shop and, by construction, F models every constraint of the problem:

- In each block it is possible to open 0 or 1 shop $\Rightarrow$ each block is represented by an edge (row$_i$, col$_j$), for each i $\in$ [1,n], for each j $\in$ [1,n] whose flow can be equal to 0 or 1.
- For each row i there is a max number of shops r_i $\Rightarrow$ each edge (s, row$_i$), for each i $\in$ [1,n], has a max flow equal to r_i.
- For each column j there is a max number of shops c_j $\Rightarrow$ each edge (col$_j$, t), for each j $\in$ [1,n], has a max flow equal to c_j.

Therefore, it proves the correctness of the construction of the flow network and that it achieves the **optimal solution** asked by the problem.

2.b) The variation of the flow network, of the algorithm and the proof of correctness

The new constraint of the problem is that there is an obligation to open at least one shop for each group of rows {1, ..., 5}, {6, ..., 10}, etc. and similarly for the columns. In order to satisfy this new constraint the flow network has to be changed in **F'={V', E' , c', l}**, this can be done from **F** deleting all the edges (s, row$_i$), for each i $\in$ [1,n], and (col$_j$, t), for each j $\in$ [1,n], then adding a node for each group of 5 row$_i$ nodes and inserting an edge from s to these nodes and from them to the respective "row nodes" row$_i$ of the group with capacity r$_i$, in case n is not divisible by 5 the last group will be less than 5 rows. The same has to be done with columns, adding a node for each group of 5 columns and inserting an edge from these nodes to t and from the respective "column nodes" col$_j$ to them with a capacity of c$_j$. Now it must be introduced a new function l: E $\Rightarrow$ $\mathbb{R}$, which represents the lower bounds of the edges. So, every flow will be such that $l(e) \leq f(e) \leq c(e)$, for each e $\in$ E. In order to find the max number of shops with this new constraint, it would seem possible to find a max flow using Ford-Fulkerson through the network adding also the constraint given by **2.b** to the constraints described in **2.a**:

- For each group of rows gr $\in$ {1, ..., 5}, {6, ..., 10}, etc. $\Rightarrow$ $\exists$ a node gr s.t $l(s, gr) = 1$.
- For each group of columns gr $\in$ {1, ..., 5}, {6, ..., 10}, etc. $\Rightarrow$ $\exists$ a node gc s.t $l(gc, t) = 1$.

However, Ford-Fulkerson cannot be applied with lower bounds, so it is needed to find an equivalent network flow without lower bounds. The new equivalent network flow **F"** is obtained from *F'* adding a new source *s"* and a new sink *t"*. *F"* is obtained by replacing each edge (u,v) of *F'* with three edges: an edge (u,v) with capacity c(u,v)-l(u,v), an edge (s",v) with capacity l(u,v) and an edge (u,t") with capacity l(u,v). If this construction produces multiple edges from *s'* to the same vertex *v*, the graph can be modified deleting this edges and adding a single edge with the sum of the capacities. If *F"* contains multiple edges from *s"* to the same vertex *x* or from *x* to *t"*, it is necessary to unify them into a single edge with capacity the sum of the capacities of the edges. On the *F"* graph it is now possible to apply the **Ford-Fulkerson algorithm**, if it returns a max flow equal to the sum C of the capacities of the edges exiting from *s"* then *F'* has a feasible flow, otherwise it is a **no-instance** because it is impossible to satisfy the lower bounds. It is possible to demonstrate that the ***F'* has a feasible flow if and only if *F"* has a max flow equal to *C***, in fact if *F'* has a feasible flow f then *F"* has a max flow equal to C just considering that the flow on every edge in *F"*, also existing in *F'*, is equal to the difference between the flow on the edge of *F'* and the lower bound of this edge, the flow for each edge exiting by *s"* and entering in a node v is the sum of the lower bounds of all the edges entering in v in *F'*, the flow for each edge entering in *t"* and exiting by a node v is the sum of the lower bounds of all the edges exiting by v in *F'* and the flow on the edge between *t'* to *s'* is equal to f; so, the max flow in *F"* is equal to C because, by construction, in this way every edge out of *s"* or into *t"* is saturated. On the other hand, the max flow of *F"* equal to C implies that F' has a feasible flow f because f would be equal to C+ x, where x represents a number which allows to not exceed the capacities of the edges in F'. Therefore, there are two possibilities: **if the graph F" has a max flow less than C, then F' has not a feasible flow (no-instance), otherwise if the graph F" has a max flow equal to C, then F' has a feasible flow (yes-instance)**. **In case of a yes-instance** the feasible flow is equal to C, but it is not the max feasible flow in F'. In order to reach **an optimal solution**, which **will be the max number of openable shops** satisfying the imposed lower bounds, it is needed to try to improve the flow, applying a **variation of Ford-Fulkerson algorithm**, which **differs from the original one to keep into account the lower bounds; so, for each augmenting path iteration, the residual graph is updated setting the capacities of the backward edges equal to the difference between the flow of the forward edges and the value of their lower bounds**.

3 Exercise 3

Brunetti, Carmignani

3.a) The algorithm

The solution proposed for this problem is based on the **greedy** approach, so it tries to make each step the best choice to get closer to the optimum. The optimum is intended to be the smallest value achievable in terms of the gap between the total number of girls and boys chosen, therefore the algorithm tries every time to improve this gap, reducing it, as long as it is possible. Besides, it is useful to define some important concepts used in the algorithm:

Definition 3.1 (Gap) *The gap is defined as the absolute value of the difference between the sum of the girls and the sum of the boys chosen among all the schools:* $\Delta = |\sum_{i=1}^{\infty} \hat{g}_i - \sum_{i=1}^{\infty} \hat{b}_i|$.

Definition 3.2 (Possible pair) *Given any school i with* (g_i, b_i)*, respectively the number of girls and boys in the school,* $(\hat{g}_i, \hat{b}_i)$ *is a possible pair if* $\hat{g}_i + \hat{b}_i = k$, $0 \leq \hat{g}_i \leq g_i$ *and* $0 \leq \hat{b}_i \leq b_i$.

Definition 3.3 (Improvement) *Given any pair* (g_i, b_i)*, where* g_i *are the girls present in the school and* b_i *the boys, in a certain school i, given* Δ *as the actual gap between girls and boys in absolute value summing all the choices for each school, and supposing that a possible pair* $(\hat{g}_i, \hat{b}_i)$ *is chosen at a certain point of the algorithm for the school i, an improvement is defined as a new possible pair* $(\bar{g}_i, \bar{b}_i)$ *chosen for the school i such that the gap is then improved, or rather reduced in absolute value. It is called* ***unit improvement*** *the one in which both the numbers of girls and boys in the pair chosen for a school has changed by one, producing in this way a variation of* Δ *by two.*

Algorithm 2

Require: $n \leftarrow$ *number of schools* , $k \leftarrow$ *students to choose for each school* , $(g, b) \leftarrow$ *arrays with the numbers of girls and boys for each school.*

```
procedure Ex_3($n$, $k$, $(g, b)$):
    Step 1: define an empty list $l$ and fill it with random possible pairs $(\hat{g}_i, \hat{b}_i)$ from each school.
    Step 2: in case of $\Delta = 0$, it returns $l$.
    Step 3: for each school $i$, with $i$ from 0 to n, take the possible pair returned by find_best_improvement((b[i],g[i]), l[i], $\Delta$, k) in order to update l[i] and to improve $\Delta$.
    Step 4: it returns $l$.
```

Require: $(b[j], g[j]) \leftarrow$ *the number of girls and boys in the school j* , $l[j] \leftarrow$ *the actual possible pair chosen for school j*, $\Delta \leftarrow$ *the actual gap*, $k \leftarrow$ *students to choose for each school.*

```
procedure FIND_BEST_IMPROVEMENT($(g[j], b[j])$, $l[j]$, $\Delta$, $k$)):
    In case of $\Delta > 0$, it computes the minimum min between $\Delta/2$, the maximum number of unit improvements that are needed to reach directly $\Delta = 0$, and so to stop the algorithm, in case of an even $\Delta$, or $\Delta = 1$ in case of an odd $\Delta$, and the number of the possible increases by one in the boys of school $j$. So, if $(\hat{g}_i, \hat{b}_i)$ is the actual possible pair chosen, it returns $(\hat{g}_i - min, \hat{b}_i + min)$ if $\hat{g}_i - min$ is $\geq 0$, otherwise it returns (0,k), and in the first case the $\Delta$ is improved of a factor equal to $2 \cdot min$, in the other of a factor equal to $2 \cdot (k - \hat{b}_i)$. Similarly, in case of $\Delta < 0$, it computes the minimum min between $\Delta/2$, the maximum number of unit improvements that are needed to reach directly $\Delta = 0$, and so to stop the algorithm, in case of an even $\Delta$, or $\Delta = 1$ in case of an odd $\Delta$, and the number of the possible increases by one in the girls of school $j$. So, if $(\hat{g}_i, \hat{b}_i)$ is the actual possible pair chosen, it returns $(\hat{g}_i + min, \hat{b}_i - min)$ if $\hat{b}_i - min$ is $\geq 0$, otherwise it returns (k,0), and in the first case the $\Delta$ is improved of a factor equal to $2 \cdot min$, in the other of a factor equal to $2 \cdot (k - \hat{g}_i)$.
```

3.a) The proof of correctness and the running time

Ad absurdum, we assume that the algorithm arrives at a solution x, a series of pairs $(\hat{g}_i, \hat{b}_i)$, which is not optimal, with a gap $\Delta = \Delta_x$. If it is not optimal, it means that it will exist at least a pair in the solution x which is improvable, meaning that it can be substituted with another possible pair which would lead to a minor gap $\Delta = \Delta_* < \Delta_x$, but it is absurd because the algorithm, by construction, stops finding all pairs that are not improvable anymore, since the algorithm starts from an initial list of possible pairs taken at random with a certain Δ and tries all possible best improvements to reduce it until there are not pairs to improve anymore.
The size of the inputs, considering the size in number of bits of n and k, being numbers given to the algorithm, and the size of the arrays b and g, is $\sim log(n) + log(k) + n + n \sim 2 \cdot n + log(k) = D$. The procedure *find_best_improvement* has a cost of $O(1)$ since it finds the best improvement, considering the case of $\Delta > 0$, computing the minimum of $\Delta/2$, the maximum number of improvements possible to reach directly $\Delta = 0$ in case of an even Δ, or $\Delta = 1$ in case of an odd Δ, and the number of the possible increases by one in the boys of each school, keeping into account that if the number of girls becomes less or equal to 0 the best choice will be *(0, k)*; similarly in case of $\Delta < 0$. So, the dominating part of the algorithm is the third one, where it makes $O(n)$ steps to achieve a condition in which no improvements are possible anymore. The algorithm has a **time complexity** of $O(n) < O(D)$ and, therefore, it is sublinear, and so **subpolynomial** in the size of the input.

4 Exercise 4

Brunetti, Carmignani

4.a) The concept of NP-completeness

In order to claim that the Grinch problem is an NP-complete problem, it is necessary to carry out the following steps:

- It shall be proved that the problem is an NP problem.
- It must be selected an NP problem G'.
- An algorithm f which transforms the problem G' in the Grinch problem, or rather through a **polynomial reduction** has to be defined → **G' $\leq_p$ Grinch problem**. The polynomial reduction function maps a generic instance *g'* of the problem G' to an instance *grinch* of the Grinch problem ⇒ ***g'* is a yes-instance for G' $\iff$ *grinch* is a yes instance for Grinch problem**.
- It shall be demonstrated the correctness of the algorithm f.
- It shall be demonstrated that algorithm f works in polynomial time.

In order to solve the problem, the **SAT problem where every clause has either all positive or all negative literals, which is an NP-complete problem, represents the problem G'**.

4.a) The proof that the problem is NP

At first, in order to prove that the Grinch problem is NP, it must be defined a **certificate** composed by a set of sets defined in this way: $(child_i, K_i, \widehat{B}(i))$, $\forall$ child i. After that a polynomial certificate has been defined, it is built an algorithm that defines in polynomial time if the certificate is a **yes-instance** or a **no-instance** for the problem.

Algorithm 3

Require: : $B \leftarrow$ set of bags, $K_i \leftarrow$ preferred set of sweets by child i, $\widehat{B}(i) \leftarrow$ set of bags that satisfies the preferences of the child i.

```
procedure CERTIFIER($B$, $K_i$, $\widehat{B}(i)$):
    Step 1: It is checked that for each bag $i$: the occurrences of $B_i$ in $\cup_i \widehat{B}(i) \leq k$. If it is true for each bag go to Step 2, otherwise
  it is a no-instance.
    Step 2: It it checked that, for each child i, $K_i \subseteq \widehat{B}(i)$: for each child and for each set of sweets that he prefers, the algorithm
  checks that all the favorite sweets are contained in child's bags. If it is true for all the children it is a yes-instance, otherwise it is a
  no-instance.
```

Both the steps are polynomial in the number of sweets, children and bags, so the problem is an NP problem.

4.a) How to prove that the problem is NP-complete

In order to prove that the problem Grinch is NP-complete, after seeing that the problem is NP, it is necessary to verify that each NP problem x is such that **x $\leq_p$ Grinch problem**. The used NP-complete problem to verify the NP-completeness of Grinch problem is the version of SAT described above. SAT defines if the expression is satisfiable or not, this is satisfiable if it exists an assignment of boolean literals that makes it 'true'. What is needed to do is to **reduce the known problem NP-complete (SAT) to the problem (Grinch)**, so given an arbitrary input i for SAT, it must to be created a particular input j for the Grinch problem such that the instance i is a yes-instance for SAT if and only if j is a yes-instance for Grinch. What is asked to show is that **SAT $\leq_p$ Grinch problem**, but, knowing that SAT is NP-complete, then for each problem x in NP it is true that **SAT $\leq_p$ x**, so it is possible to say that **x $\leq_p$ SAT $\leq_p$ Grinch problem ⇒ x $\leq_p$ Grinch problem**.
One way to approach the problem can be to build on every time a particular case of the Grinch problem, in fact it is trivial that **particular Grinch problem $\leq_p$ Grinch problem** and in this case the particular Grinch problem can be such that each bag can be stolen at most one time, so defining k = 1.
In order to prove that any arbitrary instance of the SAT problem can be mapped into a particular instance of the Grinch problem with k=1 such that the first is a yes-instance if and only if the second is a yes-instance, it is needed to map every constraint of the SAT to a constraint of the problem, in this way if the SAT has a solution also the Grinch will have it and viceversa.
Therefore, given a SAT formula, in the variation told before, it can be possible to consider only one child in the Grinch problem and to map a clause with a sweet. The positive clauses represent the sweets that the child prefers, whereas the negative ones represent sweets that are not favorite. Each literal x_i is a bag B_i where the sweet related to the clause is contained. In case of negative clauses with only a negative literal, it means that the bag related to this literal cannot be taken by the child, but it is admitted since it would contain a sweet not preferred by the child. In case of negative clauses with more than one negative literal, it means that the bags related to the literals cannot be taken together, so the reduction will add sweets in order to make them the same bag: since k=1, the Grinch won't take them together.