Optimization Problem - Summative Assessment

Pietro Zafferani

BSc in Genomics, University of Bologna

May 2, 2021

1 Introduction

The problem we were asked to solve consisted in a so called *minimization problem*, in other words the goal was to find, among all the possible solutions, one for which the unknown's value was minimum.

Supposing we are researches, we have to move across the ocean all our equipment. To do so we need containers in which locate each piece of our equipment, however these containers have a maximum weight capability that must be respected. Since shipping across the ocean each one of the putative containers has a cost, we are asked by the problem to use as few containers as possible.

The restrictions that have been imposed in the solving process were the following:

- The capacity C of each container is equal to square root of the total number of elements that must be shipped.
- Each element's weight is drawn randomly and uniformly in a range between 0 and 1.

In this paper we propose two different approaches to tackle the problem: the first approach is heuristic, it does not guarantee to obtain the optimal solution for every single instance and we will refer to it as *Greedy algorithm*; instead the second approach implements an *Exhaustive search* algorithm and it will return the optimized solution every single time. Moreover, it will be defined for each of the two algorithms the upper bound to the worst-case

computation time, in big O notation. Such predictions will be compared with empirical data.

Eventually, we will define also an approximation ratio of the two proposed algorithms and we will show how our model fits the experimental data.

2 Combinatorial Problem

Abstracting the problem at the combinatorial level is mandatory, since to design a reliable algorithm we need to start from the clear and precise definition of inputs and outputs:

Note that either the input and the output are maintained for both algorithms.

INPUT: a list $L = [i_1, \ldots, i_n]$ of n non repeated elements, where $n \in N$, each of them is associated with a weight w generated randomly in range (0,1].

OUTPUT: a list $S = [l_1, \ldots, l_m]$ with $m \in N$ and $l_1, \ldots, l_m$ be a set of non empty lists each one with $\sum w_i \leq C$, such that m is minimum for any given input L. The total sum of the elements contained in $l_1, \ldots, l_m$ must be equal to n.

As you can notice, one parameter has been directly omitted during the formulation of the combinatorial settings. We are referring to the shipping cost p for each container; since it remains the same no matter the container's degree of filling, it does not have any relevance for the solution of the problem and thus can be simply left apart.

3 Designing the Greedy algorithm

3.1 The concept

The choice for the functioning of the Greedy algorithm comes directly from the real-life perspective on the given problem. Imagine to be a very busy researcher, your goal would still be to place all the objects in the fewest number of containers possible, however the time that you can dedicate to such task is limited.

Eventually, the strategy adopted consists in allocating the elements following a descending order with respect to the weight, in other words the first element placed is the heaviest one, then we proceed with the second heaviest and so on. When the remaining capacity of a container is insufficient for the allocation of the next element, a new container starts to be filled.

3.2 The Design

Below it is shown the pseudocode proposed for the Greedy algorithm, its functioning reflects the description given in section 3.1:

$Greedy(L)$
$S \leftarrow emptyList$
$C \leftarrow \sqrt{len(L)}$
$i \leftarrow len(L) - 1$
$OrderedElements \leftarrow Sorting(L)$
while $i \geq 0$ **do**
  $RemainingCapacity \leftarrow C$
  $l \leftarrow emptyList$
  while $RemainingCapacity \geq I^{th}OrderedElementWeight$ **do**
    $RemainingCapacity = RemainingCapacity - I^{th}OrderedElementWeight$
    $l.append(I^{th}OrderedElementName)$
    $i \leftarrow i - 1$
    if $i \leq 0$ **then**
      $STOP$
    end if
  end while
  $S.append(l)$
end while
return S

The first essential feature needed for the proper functioning of the algorithm is $OrderedElements \leftarrow Sorting(L)$, this line shows the usage of an external function named *Sorting* (implementation shown in section 3.3) which takes a list as argument and returns the same list sorted ascendantly by a key feature, in this case it corresponds to the weight of each element.

The activity of the first **while** loop depends on the length of the list L denominated as i, the same variable is used also as an index to iterate over the very list L that has been already ordered. In this way, the first element of the list to be processed by the algorithm is the element with highest index and thus with heaviest weight, subsequently the second heaviest will be considered and so forth.

At this point a second **while** loop checks whether there is enough capacity left in the current container to add the i^{th} element of the ordered list, if this is the case then the element is added and its weight is subtracted from the capacity left. When the container cannot take any more elements, it is added

to S, then the second loop stops to be executed and a new container starts to be filled.

The **If** $i \leq 0$ makes sure that the condition imposed in the first **while** loop is not broken while inside the scope of the second loop. After all the elements have been allocated in the containers, namely when the index i reaches 0, the final list S is returned.

3.3 The Implementation

In this section it shown the the Python implementation that we proposed for the Greedy algorithm, it follows faithfully the pseudocode described in section 3.2. Moreover we characterized also the auxiliary functions needed for the execution of the program:

```
def create_equipment(n: int) -> list:
    #creates an unordered list of n elements
    #each element is a tuple formed by an ID and a weight
   generated randomly
    L = [(i, round(random.random(), 3)) for i in range(1, n +
    1)]
    return L
```

This first function simply creates the list L of n elements, each element has associated a weight that was generated by exploiting the library *random*.

```
#Example: create a list of 5 objects
print(create_equipment(5))

>>[(1, 0.045), (2, 0.02), (3, 0.784), (4, 0.062), (5, 0.98)]
```

The following function instead is fundamental for the Greedy algorithm as mentioned in section 3.2 since it orders L by each elements weight:

```
def Sorting(List:list)->list:
    # the key function works by sorting each element
   according to the second position in each tuple, namely the
    weight
    # reverse = False defines the ascendent order
    return sorted(List, key=lambda i: i[1], reverse=False)

#Example of Sorting() functioning
```

```
Sorting([(1, 0.045),(2, 0.02),(3, 0.784),(4, 0.062),(5, 0.98)
   ])

>> [(2, 0.02), (1, 0.045), (4, 0.062), (3, 0.784), (5, 0.98)]
```

The piece of code below represents the core function of the Greedy algorithm, it satisfies the impositions defined in section 2 and its implementation was based on the pseudocode of section 3.2:

```
1  import random
2  import numpy
3
4  def Greedy(L: list, check=True) -> list:
5      # check set by default on True
6      if check:
7      # the list is ordered by increasing weight
8          objects = Sorting(L)
9      else:
10     # the list's order is not modified
11         objects = L
12
13     # max capacity of the container
14     max_weight = round(numpy.sqrt(len(objects)), 2)
15     S = []
16     # index to search through the input list
17     i = len(objects) - 1
18
19     #loop executed until each element has been placed in the
       containers
20
21     while i >= 0:
22
23         container = []
24         max_container = max_weight
25
26         # check whether there is enough capacity in the
       container
27         while max_container >= objects[i][1]:
28             # subtract the newly-added element's weight to
       the remaining capacity
29             max_container -= objects[i][1]
```

```
30              # add element's ID to container list
31              container.append(objects[i][0])
32              i -= 1
33              # check whether there are elements left to be
    processed
34              if i < 0:
35              # if not the case exit the scope of the loop (
    line 25)
36                  break
37
38          # the container is full so it is inserted in S
39          S.append(container)
40
41      # eventually return the final list
42      return S
```

The purpose of Lines 3 to 6 will be better explained in section 4.3 since it is connected to the Exhaustive search implementation, for what concerns the Greedy algorithm, these lines make sure that the input list is always ordered ascendantly (**check** = $True$) by weight since this is a crucial passage for the proper functioning of the algorithm.

The remaining part of the function is instead a faithful implementation of the pseudocode as described in section 3.2.

3.4 The Complexity

After analyzing both the pseudocode and the Python implementation, we are now able to define the upper boundary to the worst-computationally case in terms of big O notation.

The following list explains which features of the Greedy algorithm contribute the most to its overall complexity when the input's size increases towards infinity:

Note that the reference lines come from the Python implementation of section 3.3.

- **line 8**: the Sorting() function exploit the built-in method *.sorted()*, and it is known that its internal complexity is $n * log_2(n)$.
- **line 19 to 34**: both *while* loops are executed until there are no more elements left to be allocated, hence they are linearly related to n.

All the other lines in the Python implementation are negligible since they are either constant or do not depend strictly on the input's size n. By joining each single complexity present in the Greedy algorithm we obtain the formula:

$$n * log_2(n) + n + c$$

c stands for the constant operations, hence the overall complexity considered in the worst case can be represented as

$$O(n * log_2(n))$$

for n increasing asymptotically.

3.5 The Approximation Ratio

Since the Greedy algorithm belongs to the heuristic family, it does not offer certainties in yielding always the optimal solution; for many instances it could give approximated solutions. In this section we will define the Approximation Ratio for the Greedy algorithm, namely the worst-case scenario of just how far off the heuristic's output can be from the optimal solution produced by the Exhaustive Search.

If for a given instance x of a *minimization problem*, we set as $A(x)$ the solution produced by Greedy algorithm and as $OPT(x)$ the optimal solution, then the approximation ratio $AR(x)$ is defined as

$$AR(x) = \frac{A(x)}{OPT(x)}$$

We will demonstrate how to calculate the upper bound of the approximation ratio for the Greedy algorithm:

First of all, we can assert that the algorithm does not allow the presence of two half-filled containers; in fact if for instance we take a set of ordered elements whose weights sum is equal to half of the container's capacity, the set would have been added to the container which is already half full instead of opening a new one. So we can conclude that the algorithm allows at most one half-full container.

By defining as S* the set of all the containers that are at least half-full:

$$S^* = \{l_j | \sum_{i \in l_j} w_j > \frac{1}{2}\}$$

We obtain that the sum of all the elements in all the containers present in S^* is bigger than $\frac{1}{2}$ multiplied by the same number of containers:

$$\sum_{l_j \in S^*} \sum_{i \in l_j} w_i > \sum_{l_j \in S^*} \frac{1}{2}$$

but $\sum_{l_j \in S^*}$ correspond to $A(x) - 1$, in fact it is the total number of containers that are at least half-full minus at most one less than half-full. Moreover the total sum of all elements weights might be bigger than the total sum of the elements weights in S^*:

$$\sum_i w_i \geq \sum_{l_j \in S^*} \sum_{i \in l_j} w_i$$

The optimal solution, logically, has at least enough containers to allocate all the elements. For instance if $\sum w_i = 4$ and each container has $C = 2$, the optimal solution will consists in exactly 2 containers:

$$OPT(x) \geq \sum w_i$$

By merging the previous inequalities we obtain that:

$$OPT(x) > \frac{A(x) - 1}{2}$$

We can erase -1 from the equation by converting the $>$ to $\geq$, hence:

$$2 \geq \frac{A(x)}{OPT(x)}$$

According to this result, the approximated solution given by our Greedy algorithm in the worst case can at most double the optimal solution produced by the Exhaustive Search algorithm. In section 5.1 we will check whether the theoretical upper bound holds true with respect to empirical tests.

4 Designing the Exhaustive Search algorithm

4.1 The Concept

As we did in section 3.1, the logic underlying the functioning of an algorithm can be obtained by a real-scenario perspective.

Supposing to be a meticulous researcher, this time the strategy chosen to fill the containers explores all the possible solutions present in the search space, in this way the researcher is sure that eventually, the best solution to its problem will be found.

4.2 The Design

Below it is shown the pseudocode for the Exhaustive Search algorithm with a comprehensive description of its features:

$ExhaustiveSearch(L)$
$S \leftarrow emptyList$
$BestLenght \leftarrow len(L)$
for all $Permutation\ in\ Permutation(L)$ **do**
  $Result \leftarrow Greedy(Permutation)$
  if $len(Result) < BestLength$ **then**
    $BestLength \leftarrow len(Result)$
    $S \leftarrow Result$
  end if
end for
return S

The for loop introduces the iteration over the function $Permutation(L)$ (implementation described in section 4.3), the role of this function is to generate all the possible permutation of the elements within a given list. We must iterate over it because of its particular implementation that takes advantage of Python generators; in other words every time the function is called it yields a single permutation, however it keeps track of the ones previously yielded so that it continues from there. The loop will not stop its execution until every single permutation of L will have been processed by the algorithm.

After the for loop we declared $Result \leftarrow Greedy(Permutation)$, this happens because we apply $Greedy()$ to every single permutation of L. As explained in section 3.2, $Greedy()$ returns the list of containers filled according to the order of the input list, however in this case the functioning is slightly different from the one one seen in 3.2 and the explanation will be given in section 4.3.

The *If* condition checks whether the number of containers obtained in the previous step are smaller than the current best solution, if that is the case we substitute the current best solution with the one just obtained and consequently, also the shortest number of containers ($BestLength$) is updated with the new one. The usage of the operator '$<$' and not '$\leq$' has a specific purpose in the algorithm, in fact we do not need to discriminate between results with equal number of containers, but only when there is one with fewer containers and so we can avoid doing useless operations that will not produce a better result.

Note also that in the second line of the pseudocode, $BestLength$ was

set equal to the total numbers of elements, this was necessary for allowing the execution of the *If* condition since there exist for sure results with fewer container than n, in fact that is the worst strategy of filling.

After all the possible permutations of L have been processed and checked by the algorithm, the best result S is eventually returned.

4.3 The Implementation

In this section we propose the Python implementation for the Exhaustvie Search algorithm as designed in section 4.2, together with the implementation of its auxiliary function:

```
1  def permutation(List):
2      # base case of recursion
3      if List == []:
4          # generate an empty list
5          yield []
6
7      #repeat the recursion for every elements of the input
       list
8      for Object in List:
9          # we create a new list without the 'object' element
10         new_l = [y for y in List if not y == Object]
11
12         #repeat recursion for every elements of the newly-
       truncated list inside the upper loop
13         for p in permutation(new_l):
14             # generate one of the permuted list at a time
15             yield ([Object] + p)
```

The choice of using generators for producing all the possible permutation of L was due to the fact that with high numbers of elements the computational burden becomes unbearable for the machine memory, so it was unfeasible to return the whole set of permutations at the same time, whereas generators solve this problem by creating them sequentially and thus decreasing the memory usage.

```
#Example: print all possible permutations of [(1, 0.58),(2,
   0.3),(3, 0.22)]

for p in permutation([(1, 0.58),(2, 0.3),(3, 0.22)]):
    print(p)
```

```
>>[(1, 0.58), (2, 0.3), (3, 0.22)]
[(1, 0.58), (3, 0.22), (2, 0.3)]
[(2, 0.3), (1, 0.58), (3, 0.22)]
[(2, 0.3), (3, 0.22), (1, 0.58)]
[(3, 0.22), (1, 0.58), (2, 0.3)]
[(3, 0.22), (2, 0.3), (1, 0.58)]
```

Below it is shown the code for the core function of the Exhaustive Search algorithm:

```
1  def ExhaustiveSerach(L: list) -> list:
2
3      #number of containers that would be used in the worst
       case
4      best = len(L)
5      #S is set by default as empty
6      S = []
7      #generator of all possible permutations of L
8      search_space = permutation(L)
9
10     #go through every single permutation
11     for single in search_space:
12         # the list is given to Greedy() and a set of
           containers is returned
13         # check=False means that the list is not sorted
           before being processed
14         current_single = Greedy(single, check=False)
15
16         #if all the elements fit in only one container we
           found the best solution possible
17         if len(current_single) == 1:
18             return current_single
19
20         #if a solution with fewer container is found, it is
           substituted to the former best one
21         if len(current_single) < best:
22             best = len(current_single)
23             S = current_single
24
25     return S
```

The main features of this Python implementation have been already described in section 4.2. Only few parts were not included in the pseudocode: in **line 14** $Greedy()$ has the parameter $check = False$, this condition forces the function to skip the ordering of the list's elements according to their weight. This adjustment is necessary since by default $Greedy()$ would order the input list and by doing so, exploring all the possible permutations would become useless since the only processed list would be always the one ordered according to the weight.

Lines 17-18 handle the rare case in which all the elements fit inside a single container and since there cannot be a better solution than that, the output list is directly returned.

4.4 The Complexity

Exhaustive Search algorithms are also known as *exact algorithms* since they explore the whole search space before returning the optimal solution. The drawback of employing such a strategy is that they have an intrinsically high computational complexity that cannot be lowered significantly.

Below are listed the algorithm's features defining its asymptotically complexity in terms of big O notation:

Note that the reference lines come from the Python implementation of section 4.3.

- **Line 8:** the generation of all possible permutations of n elements by definition has $n!$ time-complexity.
- **Lines 11- 14:** the for loop iterates overs $n!$ permutations and for each of them we apply the Greedy algorithm whose complexity is $n * log_2(n)$, as described in section 3.4.

All the other lines in the Python implementation can be summarized by the constant c since they do not depend strictly on the input's size n. By joining the internal complexities, we obtain the formula:

$$n * log_2(n) * n! + c$$

So we can conclude that the upper bound to the Exhaustive Search algorithm's complexity is:

$$O(n * log_2(n) * n!)$$

for n increasing asymptotically.

5 Testing the algorithms

In this section we will focus on testing both algorithms on a set of instances, moreover we will comment their functioning and how their output is produced.

The first instance we are gonna test consist in $n = 4$ and $C = 2$:

We start by using the Greedy algorithm. We proceed to generate the elements thanks to the auxiliary function:

```
#implementation:
print(create_equipment(4))

>> (1, 0.534), (2, 0.639), (3, 0.494), (4, 0.903)
```

Then we invoke the the core function that will take the list, it will sort it ascendantly by weight and it will fill the containers starting from the heaviest element. The output consists in a list showing the elements IDs allocated in each container.

Note that Greedy() automatically sets C as requested by the problem.

```
#implementation Greedy algorithm:
print(Greedy([(1, 0.534), (2, 0.639), (3, 0.494), (4, 0.903)
   ]))

>> [[4, 2], [1, 3]]
```

It is trivial to confirm that in this instance the Greedy algorithm has returned the optimal solution, or better to say, one of the optimal solutions, since other possible combinations such as: $[[2, 3], [4, 1]]$, $[[3, 1, 4], [2]]$ and many others are all equally acceptable. In fact all of them have exactly 2 containers.

Now we will proceed to test the same instance in the Exhaustive Search algorithm. It starts by taking the list of elements and generating all the permutations in this way:

```
#implementation:
for p inpermutation([(1, 0.534), (2, 0.639), (3, 0.494), (4,
   0.903)]):
print(p)

#4! = 24 possibilities

>>[(1, 0.534), (2, 0.639), (3,
   0.494), (4, 0.903)]
[(1, 0.534), (2, 0.639), (4,
   0.903), (3, 0.494)]
[(1, 0.534), (3, 0.494), (2,
   0.639), (4, 0.903)]
[(1, 0.534), (3, 0.494), (4,
   0.903), (2, 0.639)]
```

```
[(1, 0.534), (4, 0.903), (2,
   0.639), (3, 0.494)]
[(1, 0.534), (4, 0.903), (3,
   0.494), (2, 0.639)]
[(2, 0.639), (1, 0.534), (3,
   0.494), (4, 0.903)]
[(2, 0.639), (1, 0.534), (4,
   0.903), (3, 0.494)]
[(2, 0.639), (3, 0.494), (1,
   0.534), (4, 0.903)]
[(2, 0.639), (3, 0.494), (4,
   0.903), (1, 0.534)]
[(2, 0.639), (4, 0.903), (1,
   0.534), (3, 0.494)]
[(2, 0.639), (4, 0.903), (3,
   0.494), (1, 0.534)]
[(3, 0.494), (1, 0.534), (2,
   0.639), (4, 0.903)]
[(3, 0.494), (1, 0.534), (4,
   0.903), (2, 0.639)]
[(3, 0.494), (2, 0.639), (1,
   0.534), (4, 0.903)]
[(3, 0.494), (2, 0.639), (4,
   0.903), (1, 0.534)]
[(3, 0.494), (4, 0.903), (1,
   0.534), (2, 0.639)]
[(3, 0.494), (4, 0.903), (2,
   0.639), (1, 0.534)]
[(4, 0.903), (1, 0.534), (2,
   0.639), (3, 0.494)]
[(4, 0.903), (1, 0.534), (3,
   0.494), (2, 0.639)]
[(4, 0.903), (2, 0.639), (1,
   0.534), (3, 0.494)]
[(4, 0.903), (2, 0.639), (3,
   0.494), (1, 0.534)]
[(4, 0.903), (3, 0.494), (1,
   0.534), (2, 0.639)]
[(4, 0.903), (3, 0.494), (2,
   0.639), (1, 0.534)]
```

Each one of the above lists is then processed through $Greedy()$, skipping the ordering process thanks to $check = False$), and among all the possible results only the one with the fewest containers is returned. Also with this approach more than one optimal result is found, however the first one encountered is the one returned, as explained in section 4.2.

```
#implementation Exhaustive Search

print(ExhaustiveSearch([(1, 0.534), (2, 0.639), (3, 0.494),
   (4, 0.903)]))

>> [[4, 3], [2, 1]]
```

As we can notice, the orders of the elements in the containers in the optimal solution reflects the order of the elements in the input list (start allocating from the last element), this occurred because that was the first list encountered by the algorithm and it returned immediately the optimal result.

The next instance we are gonna test consists in $n = 6$ and $C = 2.5$, the procedure applied is the same used for the previous instance.

```
#creation of the 6 elements
```

```
elements = create_equipment(6)
print(elements)

>>[(1, 0.775), (2, 0.935), (3, 0.945), (4, 0.863), (5, 0.259)
   , (6, 0.601)]

#implementation Greedy algorithm
print(Greedy(elements))

>> [[3, 2], [4, 1, 6], [5]]

#implementation Exhaustive Search
print(ExhaustiveSearch(elements))

# 6! = 720 possibilities

>> [[6, 4, 3], [5, 2, 1]]
```

With this particular set of elements the Greedy algorithm returned a non-optimal solution since the numbers of containers was 3, whereas the Exhaustive search returned the optimal solution of 2 containers.

We propose also an automated routine to test the performance of both algorithms for $n = 4, 6$.

```
#implementation of tesitng routine
def routine(List) -> print:
    for n in List:
        elements=create_equipment(n)
        greedy=Greedy(elements)
        ex = ExhaustiveSearch(elements)

        print('List of elements: '+ str(elements))
        print('Greedy result: ' + str(greedy))
        print('Exhaustive result: ' + str(ex)+'\n')

        if len(greedy) != len(ex):
            print('Greedy algorithm did not perform well'+'\n
   ')

        else:
            print('Greedy algorithm performed well'+'\n')
```

```
print(routine([4,6]))

>> List of elements: [(1, 0.537), (2, 0.414), (3, 0.087), (4,
    0.807)]
Greedy result: [[4, 1, 2, 3]]
Exhaustive result: [[4, 3, 2, 1]]

Greedy algorithm performed well

List of elements: [(1, 0.134), (2, 0.9), (3, 0.92), (4,
   0.598), (5, 0.162), (6, 0.402)]
Greedy result: [[3, 2, 4], [6, 5, 1]]
Exhaustive result: [[6, 5, 4, 3], [2, 1]]

Greedy algorithm performed well

#Both algorithms gave the optimal solutions for each of the
   instances.
```

Below it is shown a benchmarking routine that takes as argument either of the two algorithms and runs it on randomly generated inputs of size 4, 6, 8, 10, 12, 14, 16, 18, 20.

```
#implementation of benchmarking routine
def benchmarking(function) -> print:
    for n in [4, 6, 8, 10, 12, 14, 16, 18, 20]:
        elements = create_equipment(n)

        if function == 'Greedy':
            result = Greedy(elements)
            print(result)

        elif function == 'Exhaustive':
            result = ExhaustiveSearch(elements)
            print(result)


#to test the Greedy algorithm:
benchmarking('Greedy')

#to test the Exhaustive Search
benchmarking('Exhaustive')
```

```
#Example: testing routine for Greedy()
benchmarking('Greedy')

>> [[2, 1, 4, 3]]
[[4, 1, 6], [5, 2, 3]]
[[5, 1, 7], [3, 8, 6, 4, 2]]
[[8, 1, 5], [4, 2, 10, 7, 3], [9, 6]]
[[5, 8, 2, 7, 11], [1, 3, 10, 4, 9, 12, 6]]
[[5, 3, 8, 14], [7, 2, 10, 9, 6, 1, 13, 4, 12, 11]]
[[2, 6, 1, 12, 4], [16, 11, 5, 8, 10, 9, 14, 15, 13, 7, 3]]
[[14, 12, 9, 5], [11, 2, 7, 13, 3], [4, 18, 16, 1, 6, 10, 8,
   15, 17]]
[[14, 3, 7, 17, 8], [1, 10, 20, 19, 5, 11, 15, 12, 13, 16,
   18, 4, 2, 6, 9]]
```

5.1 Evaluating the Approximation Ratio

In section 3.5 we demonstrated how the upper bound to the Approximation ratio of the Greedy algorithm cannot exceed the value of 2. Now we will evaluate if the empirical data supported our conclusion.

If we consider any instance I of 4 elements, we can demonstrate that $AR(I) = 1$. In fact, if all 4 elements fit inside a unique container, both algorithms will produce the optimal result since in that case the order of the elements will not matter. In the case one container is not sufficient, since $\sum_1^4 w \leq 4$ because of the constrains of the constrains imposed in section 1, exactly 2 containers will be required for both algorithms.

Instead if we consider an instance I of 6 elements as we have seen in section 5, (precisely $[(1, 0.775), (2, 0.935), (3, 0.945), (4, 0.863), (5, 0.259), (6, 0.601)]$) the Greedy algorithm produced a result different from the optimal one. When we calculate the the approximation ratio for this specific instance we find that:

$$AR(I) = \frac{3}{2}$$

so the theoretical bound of $AR(I) \leq 2$ is still valid.

The same result can be obtained by testing instances of 9 elements, for example the list $[(1, 0.786), (2, 0.67), (3, 0.241), (4, 0.149), (5, 0.597), (6, 0.77), (7, 0.842),$ $(8, 0.945), (9, 0.633)]$ produced two different results:

```
elements = [(1, 0.786), (2, 0.67), (3, 0.241), (4, 0.149),
   (5, 0.597), (6, 0.77), (7, 0.842), (8, 0.945), (9, 0.633)]

print(Greedy(elements))
print(ExhaustiveSearch(elements))

>> [[8, 7, 1], [6, 2, 9, 5, 3], [4]]
[[9, 8, 6, 5], [7, 4, 3, 2, 1]]
```

Obviously $AR(I) = \frac{3}{2}$

6 Running-time complexities

In this section we will evaluate through empirical data the running-time of each algorithm tested for different input sizes, then we will compare our findings with the predicted time complexities proposed in sections 3.4 and 4.4.

To evaluate the average running time for a given input size we implemented functions exploiting the *codetiming* library.

6.1 Running the Greedy algorithm

The first function proposed is used to return the average running time of the Greedy algorithm for a defined number of elements. The average is calculated over 1000 repetitions of the experiment.

```
def TestGreedy(n: int, repetitions=1000) -> float:
    # number of trials
    for i in range(repetitions):
        # create the list of elements
        Input = create_equipment(n)

        # record the running time
        with Timer(name='GreedyAlgorithm', logger=None):
            Greedy(Input)

    # return the average
    return round(Timer.timers.mean('GreedyAlgorithm'), 9)

#test the running time(s) of Greedy() with 6 elements
print(TestGreedy(6))
```

```
>> 4e-05
```

Then to test an array of inputs we implemented another function, the results will be used to produce a set of coordinates needed for the graphical representation of the algorithm's behaviour.

```
def Time_setsGreedy(input_array: list) -> list:
    GreedyRes = list()
    # iterating over the list of input sizes
    for Input in input_array:
        # test the running time for each input size
        result = TestGreedy(Input)
        GreedyRes.append(result)
    # return the an array of running times
    return GreedyRes

#recording the running times for
   [4,6,8,10,20,30,40,50,60,70,80,90,100] input sizes
print(Time_setsGreedy([4,6,8,10,20,30,40,50,60,70,80,90,100])
   )
```

```
>> [8.5321e-05, 6.1387e-05, 4.9774e-05, 4.4544e-05, 5.4515e
   -05, 5.6888e-05, 5.8269e-05, 6.0034e-05, 7.0486e-05,
   7.3534e-05, 8.0375e-05, 9.1957e-05, 9.8907e-05]
```

The set of coordinates is instead produced by another auxiliary function.

```
def printPairs(Set, time_data) -> print:
    for (n, t) in zip(Set, time_data):
        print((n, t), end=' ')

#pass to the function the inputs size and their respective
   running times previously produced

PrintPairs([4,6,8,10,20,30,40,50,60,70,80,90,100], [8.5321e
   -05, 6.1387e-05, 4.9774e-05, 4.4544e-05, 5.4515e-05,
   5.6888e-05, 5.8269e-05, 6.0034e-05, 7.0486e-05, 7.3534e
   -05, 8.0375e-05, 9.1957e-05, 9.8907e-05])
```

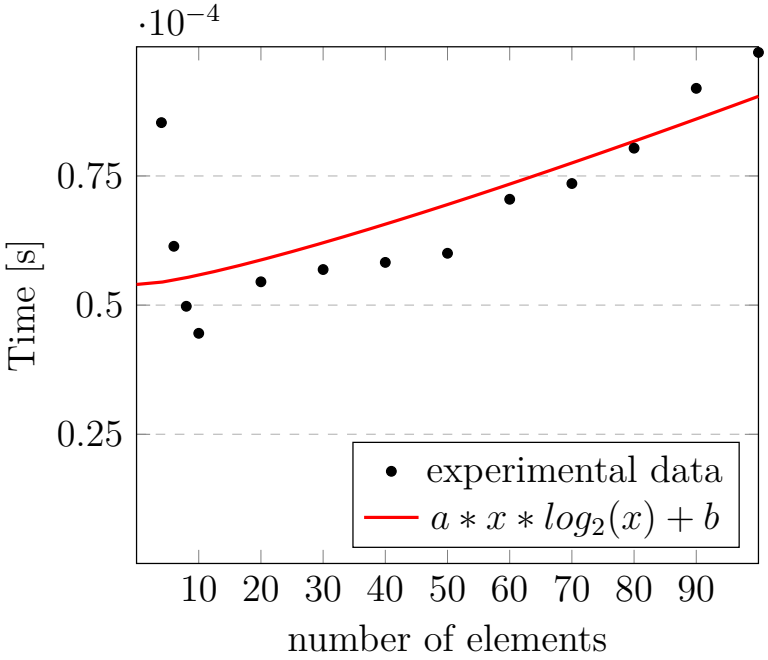
```
>> (4, 8.5321e-05) (6, 6.1387e-05) (8, 4.9774e-05) (10,
   4.4544e-05) (20, 5.4515e-05) (30, 5.6888e-05) (40, 5.8269e
```

```
-05) (50, 6.0034e-05) (60, 7.0486e-05) (70, 7.3534e-05)
(80, 8.0375e-05) (90, 9.1957e-05) (100, 9.8907e-05)
```

In section 3.4 we explained that the upper bound to the worst case complexity is $O(n * log_2(n))$, now we will use the coordinates just found above to graphically represent the algorithm's behaviour in time, moreover we will plot a mathematical curve that follows the big O notation trend. Our goal is to check whether our prediction is supported by empirical data. The *scipy* library's methods have been exploited in order to perform the operations just mentioned.

```
def fit_greedy(Set,times):

    #define the mathematical function
    def curve(x, a, b):
        return a * x * numpy.log2(x) + b

    #return the coefficients that shape the curve in order to
    fit the given data
    return optimization.curve_fit(curve, Set, times)

#returns a,b for Y = a*x* log_2(x) + b
print(fit_greedy([4,6,8,10,20,30,40,50,60,70,80,90,100],
   [8.5321e-05, 6.1387e-05, 4.9774e-05, 4.4544e-05, 5.4515e
   -05, 5.6888e-05, 5.8269e-05, 6.0034e-05, 7.0486e-05,
   7.3534e-05, 8.0375e-05, 9.1957e-05, 9.8907e-05]))

>> (array([5.48396555e-08, 5.40327265e-05]), array([[
   2.08907826e-16, -5.37907789e-14],[-5.37907789e-14,
   2.41743458e-11]]))

#a = 5.48396555e-08
#b = 5.40327265e-05 -> b is needed to make the curve more
   realistic
```

The plot below was possible only because of all the steps we just performed:



Time [s]	8.53e-05	6.13e-05	4.97e-05	4.45e-05	5.45e-05	5.68e-05	5.82e-05
Elements	4	6	8	10	20	30	40

Time [s]	6e-05	7.04e-05	7.35e-05	8.03e-05	9.19e-05	9.89e-05
Elements	50	60	70	80	90	100

6.2 Running the Exhaustive Search algorithm

The functions designed for testing the Exhaustive Search running time are very similar to the ones seen in section 6.1 so their description will be briefer, however we will comment their implementation.

The first function proposed is used to return the average running time of the Exhaustive Search for a defined number of elements. The average is calculated over 1000 repetitions of the experiment.

```
def TestExhaustive(n: int, repetitions=1000) -> float:
    # number of trials
    for i in range(repetitions):
        # create the list of elements
        Input = create_equipment(n)

        # record the running time
        with Timer(name='ExhaustiveSearch', logger=None):
            ExhaustiveSerach(Input)
```

```
    # return the average
    return round(Timer.timers.mean('ExhaustiveSearch'), 5)

#test the running time (in seconds) of ExhaustiveSearch()
   with 6 elements
print(TestExhaustive(6))

>> 0.01065
```

Then to test an array of inputs we implemented another function, the results will be used to produce a set of coordinates needed for the graphical representation of the algorithm's behaviour.

```
def Time_setsExhaustive(input_array: list) -> list:
    EXRes = list()
    # iterating over the list of input sizes

    for Input in input_array:
        # test the running time for each input size
        result =TestExhaustive(Input)
        EXRes.append(result)

    # return the an array of running times
    return EXRes

#recording the running times for [3,4,5,6,7,8,9,10] input
   sizes
print(Time_setsExhaustive([3,4,5,6,7,8,9,10]))

>> [7e-05, 0.00013, 0.00054, 0.00223, 0.01906, 0.10204,
   1.25208, 10.9534]
```

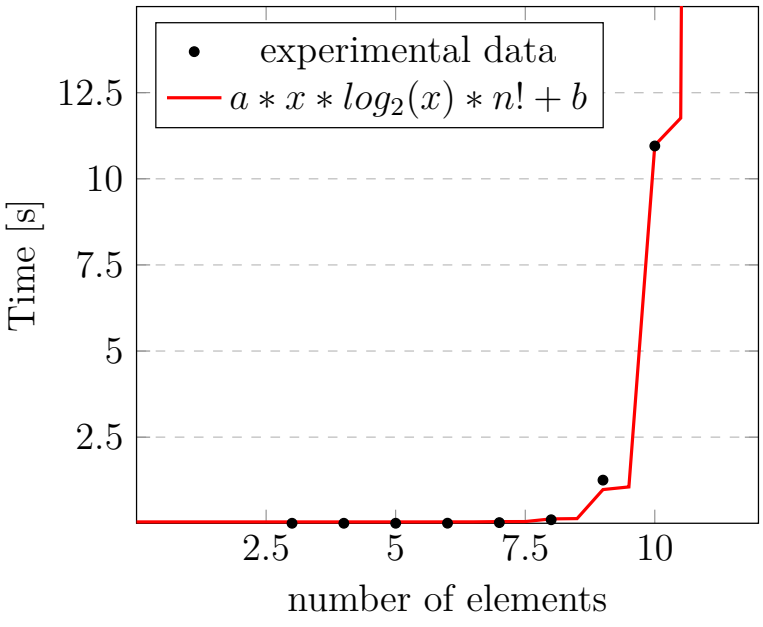
The set of coordinates is produced by $PrintPairs()$ as in section 6.1 and they are listed below:

```
>> (3, 7e-05) (4, 0.00013) (5, 0.00054) (6, 0.00223) (7,
   0.01906) (8, 0.10204) (9, 1.25208) (10, 10.9534)
```

For the Exhaustive Search algorithm, the upper bound to the worst case complexity predicted in section 4.4 was $O(n * log_2(n) * n!)$, thanks to an auxiliary function we will fit the experimental data around a mathematical curve shaped as the the big O notation of the algorithm. Our goal is to check whether our prediction is supported by empirical data.

```
def fit_exhaustive(Set, times):

    #define the mathematical function
    def factorial(x, a, b):
        return a * x * numpy.log2(x) * scipy.special.
   factorial(x) + b

    #returns the coefficients of the curve
    return optimization.curve_fit(factorial, Set, times)

#returns a,b for Y = a*x* log_2(x)*n! + b
print(fit_exhaustive([3,4,5,6,7,8,9,10],[7e-05, 0.00013,
   0.00054, 0.00223, 0.01906, 0.10204, 1.25208, 10.9534]))

>> (array([9.07302444e-08, 3.94623893e-02]), array([[
   9.48867906e-19, -1.39144347e-11], [-1.39144347e-11,
   1.54343891e-03]]))

#a = 9.07e-08
#b = 3.94e-02
```

Below we propose the graphical representation of the outcomes:



Time [s]	7e-05	0.00013	0.00054	0.00223	0.01906	0.10204	1.25208	10.9534
Elements	3	4	5	6	7	8	9	10