Assignment:
Exhaustive search algorithm Vs Greedy algorithm

Worked by: Amelia Kurti

May 2021

1 Introduction

The aim of this report is to compare the solution of a given combinatoric problem, using two different approach, Exhaustive search algorithm and Greedy algorithm. Both algorithm codes were tested, data is collected and compared to determine the efficiency and complexity of each of the algorithms.

2 The problem to be solved

Given a number of objects n, each of them having a certain weight (a float number from 0 to 1), one has to allocate them on the minimal number of containers. To do so, must be taken into account that each container has a maximal weight capacity, C such that C = $\sqrt[2]{n}$.

3 Different approaches

3.1 The Greedy algorithm

Greedy algorithm is a heuristic problem solving technique of making the locally optimal choice in every step of execution, which provides a solution in a reasonable amount of time, however not ensured to be always the optimal one. So in our study case, the greedy solution will provide us with an option of objects allocation, yet it will not always give the combination involving the smallest number of containers.

Input: The list containing the weights of objects.

Output: List of containers, each of them storing some of the objects, whose total weight is smaller or equal to C. In each container are given the weights of the objects instead of the number that identifies the object.

$C \leftarrow \sqrt[2]{No.objects}$
$positioning \leftarrow []$
$i \leftarrow 0$
while $i \leq len(weights)$ **do**
  $Container \leftarrow []$
  $capacity \leftarrow C$
  while $weights[i] \leq C$ **do**
    $capacity- = weights[i]$
    $Container.append(weights[i])$
    $i \leftarrow i + 1$
    if $i == len(weights)$ **then**
      break
    end if
    $positioning.append(Container)$
  end while
end while
return *positioning*

3.1.1 Description of algorithm's design

The execution of the function which applies the Greedy approach, comes after some other computational steps, which we will see to be valid and executed for the Exhaustive search as well. For the total program to be run, an integer is needed as the only input. This integer that for explanation simplicity we will call ***n***, is given as an input to the function ***generate()***.

First it will create a list of ***n*** integers, logically containing numbers from 1 to ***n***, by exploiting Python's ***random*** module. The elements of this list will serve as the **keys** of a dictionary, which stores the number associated to each object, and its respective weight. To generate the weights, which are a float number from 0 to 1, ***random*** module is used. In the end, it returns the created dictionary.

```
def generate_d(n):#create dictionary of object:weight pairs
    l = input_number(n)
    dict1= {}
    for key in l:
        if key not in dict1:
            dict1[key]= random.random()
    return dict1
```

Why do we need to complicate the program by using a dictionary?

The need of the dictionary arises from the fact that every object has a specific weight. On the constrains of the given problem, we saw that containers do have a maximal weight. Consequently to put the objects in the containers, we must proceed by considering the weight of the object, not its associated num-

ber. However in the end, we are interested to know which object goes in a given container, and undoubtedly the whole program needs to return the numbers instead of the weights. Later on two functions that perform this task will be shown.

```
def allocation(list):
    C = np.sqrt(len(list))
    final = []
    i = 0
    while i<len(list):
        container = []
        capacity = C
        while lis[i]<=capacity:
            capacity-=lis[i]
            container.append(lis[i])
            i+=1
            if i == len(list):
                break
        final.append(container)
    return final
```

The represented Greedy function allocates the objects in the containers making local optimal choices. Overall, this function fills a container at a time, considering every object's weight, and putting it inside a container, always checking whether the container is out of mass capacity (C - the weight of object to be inserted).

C is calculated as the square root of the number of objects to be moved. In this case, since the input of the function is the list with all $\boldsymbol{n}$ objects' weights, its length is equal to $\boldsymbol{n}$. So C = $\sqrt[2]{len(list)}$.

Set $\boldsymbol{i} = 0$, which will be used as an index to iterate over the elements of the list. Starting from $\boldsymbol{i}$=0 (the very first list element), till the very last list element, the following steps are taken:

- Define the maximum weight capacity as a variable which will be updated every time an object is inserted into a container.
- An empty container is given.
- For every element of the list, it is checked whether its weight exceeds the maximal capacity of container.
- If capacity is not exceeded (C not 0) then these elements are appended, placed in this container.
- The capacity of the container now becomes ***initial capacity - weight of the inserted element***

- If the object being considered cannot be inserted into the container, this container will be appended to the final list, and anew empty one will be created.
- ***i+=1***, meaning that we are indexing the next element of the list, and proceed the same way as for the previous one.
- This flow will stop when the ***i*** reaches the length of the list, theoretically the program would have been indexing outside of the list.
- All filled containers are appended to ***final***, first an empty list, which symbolises the final allocation.This is returned by the function, A list of lists which contain objects' weights.

3.1.2 The Theoretical Approximation Ratio Calculation

Before in this paper was mentioned that the Greedy approach does not always yield the optimal solutions, but for certain cases provides approximated ones. The Approximation Ratio is an estimator of an algorithm's provided solutions, which measures how bad the approximate solution is distinguished with the optimal solution.

In a minimization problem similar to the one we are considering, for any instance$x$, we define $A(x)$ the solution produced by Greedy algorithm and $OPT(x)$ the optimal solution (in our case given by the Exhaustive Search), then the approximation ratio $AR(x)$ is:

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

The structure of the Greedy algorithm, ensures that we cannot have two containers that are less than half full. Because each time a container has enough weight capacity to accommodate another object, the flow of the algorithm will definitely put this object in this container instead of an empty one. Hence, instead of having two half full containers, the algorithm returns only one full container.

Consequently we can define as a property of the Greedy algorithm: The Greedy algorithm returns an objects' allocation in which at most one container is less than half full.

Let H* be the set of all the containers that whose capacity is consumed at least at 50

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

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

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

This to demonstrate than all these objects cannot be put in this given number of half full containers.

but $\sum_{l_j \in H^*}$ correspond to $A(x) - 1$, the total number of containers needed to allocate the objects, which are at more than or half-full, minus at most one less than half-full.

So, since we have at least one container which is less than half full, the sum of all elements weights might be bigger than the total sum of the elements weights in S^*:

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

On the other hand, the optimal solution, logically, has at least enough containers to allocate all the elements. So the weight capacity of all containers present in the optimal solution is at least equal, sometimes even bigger that the total weight of all the objects.

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

By considering the previous inequalities we obtain that:

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

We can erase -1 from the equation, implying that:

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

From the mathematical logical analysis, we can accept that the Greedy algorithm worst case solution, involves twice the number of containers which are given in the optimal solution.

3.2 The Exhaustive search algorithm

Exhaustive search is general problem-solving technique that explained by our current problem, consists of systematically enumerating all possible permutations of objects in the minimal number of containers, always respecting the maximal weight capacity C. In the end, returns the combination which uses the smallest number of containers, to position every given object.

Input: The list containing the weights of objects.

Output: List of the minimal number of containers needed to transport all the objects. In each container are given the weights of the objects instead of the number that identifies the object.

$maxContainersNo \leftarrow len(weights.list)$
$totalPermutations \leftarrow permutations(weights.list)$
$best.permutation \leftarrow []$
for all $permuation \in totalPermutations$ **do**
  $allocationPosibility \leftarrow allocation(permutation)$
  if $len(allocationPosibility) \leq maxContainersNo$ **then**
    $maxContainersNo \leftarrow len(allocationPosibility)$
    $best.permutation \leftarrow allocationPosibility$
  end if
end for
return $best.permutation$

3.2.1 Description of algorithm's design

The implementation of the Exhaustive search depends directly on the greedy approach program as well as on a function, ***total-p(list)***, to compute all the possible permutations of a list recursively.

```
def total_p(random_list):
    if len(random_list) <=1:
        yield random_list
    else:
        for permutation in total_p(random_list[1:]):
            for i in range(len(random_list)):
                yield permutation[:i]+random_list[0:1]+permutation[i:]
```

Since in ***total-p(list)*** generators are used, in the exhaustive search function, iteration over all the possible permutations in performed.

```
def ex_search(list):
    max_containers_no = len(list)
    result = total_p(list)
    best_allocation = []
    for permutation in result:
        if len(allocation(permutation))<max_containers_no:
            max_containers_no = len(allocation(permutation))
            best_allocation = allocation(permutation)
    return best_allocation
```

Generally, the function considers all the permutations, in which it applies the ***allocation(list)***. Then it chooses and returns the allocation option which involves the smallest number of containers needed to put the objects (or one of them, in case there are several options with the same containers number).

max-containers-no coincides to the length of the objects' weights list, which is given as a parameter to the function. This to take in consideration that the worse case scenario one might need one container to transport one object.

result is named the list with all the possible permutations formed by the execution of ***total-p(weights.list)***. Every permutation sorts out differently the elements of the list.

Why do we need all the possible weights sorting?

Extremely important as a step, because for every permutation, ***allocation(permutation)*** is run. We already saw that this function considers and puts in a container the list elements one by one. So in the end, applying this function creates all different possible combinations of objects in containers. All these cases need to be analysed by the exhaustive search, in order for the best one to be chosen and returned.

After the ***allocation(permutation)*** is computed, its length compared to the maximal number of containers that can be used. If the allocation being compared involves fewer containers, it will be set as the new ***max-containers-no***, and the next allocation will be compared to it. This goes on until there is no other allocation whose length is smaller than the ***max-containers-no***.

In this cases, the allocation used to evaluate ***max-containers-no***, is assigned as the best-allocation, and it is returned as the function's result.

4 Programs' execution

As previously mentioned, the program of Exhaustive search as well as the Greedy one, when executed will print the containers in which, the inserted objects are presented by their weight. Our interest is to clarify and ameliorate the understanding of what the programs offer us as a solution, the final representation should be given by the number of objects, rather than their weights.

$GivenOutput : [[w_1, w_3], [w_2], [w_4]]$

$NeededOutput : [[1, 3], [2], [4]]$

In order for the programs to return the Needed Output, two additional functions are used, ***objects(list, dictionary)***, with performs the transformation from weights list to objects' numbers list, and ***find(list, dictionary)*** that works as auxiliary function to complete the activity of ***objects(list, dictionary)***.

```
def objects(list, dictionary):
    positioning = []
    for container in list:
        new_container = find(container, dictionary)
        positioning.append(new_container)
    return positioning
```

```
def find(list, dictionary):
    new_l = []
    for element in list:
        for key in dictionary:
            if dictionary[key] == element:
                new_l.append(key)
    return new_l
```

The flow of execution starts with creating an empty list, ***positioning*** which will be our new container representation. ***objects(list, dictionary)*** takes as input the resulting list of either Exhaustive Search or Greedy Function, and the dictionary with *object:weight*. For every container of the result, ***find(list, dictionary)*** function is called. ***find(list, dictionary)*** considers the elements of the container, and search on the dictionary for a key, whose value corresponds to the weight being considered. All the keys are appended on ***new-l*** list, the new representation of container.

After every container is converted, it is appended to ***positioning***.

5 Testing

In order to understand the difference between the two algorithmic approaches we have used to solve this combinatorial problem, testing gives us a clearer insight regarding the correctness of the programs, the efficiency and optimal solution provision.

5.1 Testing two instances

In the first step of testing, both The Exhaustive Search program and the Greedy program are tested manually, by two instances, n=4 and n=6. What facilitates the process, is the identical interface of the programs. The chosen **n** is used as the input in the call of ***generate-d(n)*** function, which instantiates a dictionary of n key:value pairs, corresponding to **object number:object weight**. The values of the dictionary are returned forming the weight list, which directly is used by the greedy and the exhaustive search functions.

After the respective functions have performed the computation and returned the container allocation, the ***objects(allocation list, dictionary)*** function, takes care to return the allocation whose containers are filled by the numbers of the objects, rather than by their weights.

Note: To have a better understanding and evaluation of the returned result by both programs, not only their result is printed, but also the dictionary containing the number:weight pairs. So, since the inputs are considerably small, we can easily judge if the program has made the optimal selection.

5.1.1 Testing to define the correctness of Greedy

As described before, the list of weights is given as the input to the function of the exhaustive search. Directly the length of this list is saved as a variable **max-containers-no** and will be further used to evaluate if the allocation of objects we are considering is better than the last one. Moreover, all the permutation of the weights list will be computed by ***total-p(weights-list)*** and the function starts an iteration among them. For each permutation the allocation of objects in container will be computed and the purpose is to find the shortest one.

Why the shortest?

Because every allocation is returned as a list of containers. The shortest is the list, the fewer containers it contains.

```
def ex_s(weights-list):
    max_containers_no = len(weights-list)
    result = total_p(weights-list)
    best_allocation = []
        for permutation in result:
        if len(allocation(permutation))<max_containers_no

        ...
```

Taking as input the weights list, the greedy function directly computes the allocation. The capacity, corresponding to the square root of the list length is set as a variable to control the container filling, while the function iterates through all the elements in the list, following their index order. If the weight of element we are considering exceeds the left capacity, this container is considered to be full and consequently appended to the final allocation array. then the next element of the list is considered. For the element which could not fit, we create another container to place it. Once this element is placed, the function moves to the next element of the list and the procedure follows the same steps.

In case of Exhaustive Search, for input both $\mathbf{n = 4}$ and $\mathbf{n = 6}$, the results usually consist on allocation involving only one container, however based on the randomly generated weight values, it occurs to have allocation with two containers. By looking at the weights, we can clearly check that this is the optimal solution, remembering that the sum of the weights inside a container should not exceed C = $\sqrt[2]{n}$

For a better comparison, the greedy has to be tested for the list of weights ran in the exhaustive search, to see if the allocation result will be the same.

Exhaustive Search

```
>>n = 4

>>{4: 0.5309096527510525, 1: 0.762213124607858, 2: 0.8561751016879173,
3: 0.6379177722827676}
```

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

>>{2: 0.9315519786194371, 1: 0.13982240520591838, 3: 0.1286926976904842,
4: 0.07833802870425188}
>>[[2, 1, 3, 4]]
```

Greedy

```
>>n = 4

>>{4: 0.5309096527510525, 1: 0.762213124607858, 2: 0.8561751016879173,
3: 0.6379177722827676}
>>[[4, 1], [2, 3]]

>>{2: 0.9315519786194371, 1: 0.13982240520591838, 3: 0.1286926976904842,
4: 0.07833802870425188}
>>[[2, 1, 3, 4]]
```

For input **n=4** it is clear that the results obtained by both functions are the same. This is because the capacity of the container is quite small, only 2 , and since the length of the list is 4, the number of permutations considered by the exhaustive search is small. This increases the likelihood that the exhaustive and greedy display the same result.

Exhaustive Search

```
>>n = 6

>>{4: 0.014046794823393882, 3: 0.09788387796552112, 5: 0.27720854040566456,
2: 0.1466625710110686, 6: 0.7448443773459372, 1: 0.9245782153721317}
>>[[4, 3, 5, 2, 6, 1]]

>>{4: 0.7106289000988407, 3: 0.775341045501294, 2: 0.9719702254579078,
1: 0.8923647363642258, 6: 0.25421987330870366, 5: 0.5364601811587754}
>>[[3, 2], [4, 1, 6, 5]]
```

Greedy

```
>>n = 6

>>{4: 0.014046794823393882, 3: 0.09788387796552112, 5: 0.27720854040566456,
2: 0.1466625710110686, 6: 0.7448443773459372, 1: 0.9245782153721317}
>>[[4, 3, 5, 2, 6, 1]]

>>{4: 0.7106289000988407, 3: 0.775341045501294, 2: 0.9719702254579078,
1: 0.8923647363642258, 6: 0.25421987330870366, 5: 0.5364601811587754}
```

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

For input **n=6**, for certain high value weights which are randomly generated, the result of the greedy is not optimal. In the case of the exhaustive search, the permutation function allows the formation of different elements ordering, which further are to be considered and compared by the exhaustive search mechanism, to return the best allocation which involves the smallest number of containers. So the very last testing case, proves that the greedy approach will not always provide us with the optimal solution.

5.1.2 The automated testing routine

To simplify the comparison, an automated testing routine can be implemented. This function takes as an input a list of values in which both exhaustive search and the greedy are executed on the same time. The value is needed for the ***generate-d()*** function to generate a dictionary, whose list of values serve as input for the greedy and exhaustive search.

```
def automated_testing(input:list):
    for input_option in input:
        generate_dictionary = generate_d(input_option)
        generate_list = list(generate_dictionary.values())
        exhaustive = objects(ex_s(generate_list), generate_dictionary)
        greedy = objects(allocation(generate_list), generate_dictionary)
        if len(exhaustive)==len(greedy):
            print('For input', input_option, 'the given results by
            both functions are appropriate:')
            print('Result of the exhaustive search: ', exhaustive)
            print('Result of the greedy function: ', greedy)
        else:
            print('For input', input_option, 'the given result by
            Greedy approach is not the optimal one:')
            print('Result of the exhaustive search: ', exhaustive)
            print('Result of the greedy function: ', greedy)
```

The result for each value of the testing list are compared between greedy and exhaustive search. In the end, by evaluating the length of allocation option provided by each of approaches, if the length of outputs from different methods correspond, it returns the pair of solution and the message that the results are appropriate. Otherwise it prints that the greedy proposed solution is not optimal.

This can be used to test different values, including those from the previous test. Here is one example obtained after several of testing rounds of ***automated-***

testing([4,6]) in which the result given by the greedy isn't the optimal one.

```
>>For input 4 the given results by both functions are appropriate:
>>Result of the exhaustive search:  [[1, 2, 3, 4]]
>>Result of the greedy function:  [[1, 2, 3, 4]]
>>For input 6 the given result by Greedy approach is not the optimal one:
>>Result of the exhaustive search:  [[4, 6, 5], [2, 3, 1]]
>>Result of the greedy function:  [[2, 4], [6, 5, 3], [1]]
```

5.2 The benchmarking routine

The benchmarking routine is a function that depending on the input provided by the user, test either the Exhaustive search or the Greedy, on a given set of input values, printing the allocation result for every value.

Based on the function name given by the user, the benchmarking function decides whether to proceed with the exhaustive search function of with the greedy one. After this decision is taken, for every input number of the list, it computes the dictionary with n pairs, subtracts the list of its values and perform the allocation.

```
def benchmarking(function, inputs_numbers):
    if function == 'Exhaustive_Search':
        for element in inputs_numbers:
            generate_dictionary = generate_d(element)
            generate_list = list(generate_dictionary.values())
            print(objects(ex_s(generate_list), generate_dictionary), end='\n')
    if function == 'Greedy':
        for element in inputs_numbers:
            generate_dictionary = generate_d(element)
            generate_list = list(generate_dictionary.values())
            print(objects(allocation(generate_list), generate_dictionary),
            end='\n')
```

The only issue is that the size of inputs for which each of either greedy or exhaustive search can run, is not the same. The complexity if the exhaustive search, as it will be presented later in this document, is too high due to the permutation calculation, and it is almost impossible to run for an input larger than 10. So one must consider this fact when gives the input to the benchmarking function, otherwise the function will run for hours.

Results of benchmarking applied on Greedy and Exhaustive Search, for values [2, 3, 4, 5, 6, 7, 8]

```
Greedy:
[[1, 2]]
```

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

Exhaustive Search:
[[2, 1]]
[[1, 2], [3]]
[[3, 2], [4, 1]]
[[2, 1, 5, 4], [3]]
[[3, 4, 1, 6], [2, 5]]
[[3, 1, 4, 6, 7, 5], [2]]
[[8, 5, 7], [6, 4, 3, 2, 1]]
```

Why does it look like the Greedy's solutions are the optimal ones compared to the Exhaustive search?

This is due to the fact that although the function that generates the needed dictionary is run on the same values, the values of the dictionary are generated randomly. Consequently, values lists taken as input form the Greedy and the Exhaustive search are completely different.

5.3 Evaluation of Approximation Ratio of Greedy Approach

To evaluate the worse case approximation ratio given by the greedy program, the solutions provided by each program for a given value are compared. To do so, the before mentioned ***automated-testing(input:list)*** is modified to fulfill our purpose. Instead of providing a list with different values as an input, in this case we provide a list with the same integer repeated several time, so the programs are tested several times for the same input. We just compare the results of both algorithmic approaches, and then the function prints only the cases when it notices that the solution provided by the greedy involves more container than the exhaustive search one.

```
def automated_testing(input:list):
    for input_option in input:
        generate_dictionary = generate_d(input_option)
        generate_list = list(generate_dictionary.values())
        exhaustive = objects(ex_s(generate_list), generate_dictionary)
        greedy = objects(allocation(generate_list), generate_dictionary)
        if len(exhaustive)!=len(greedy):
            print('For input', input_option, 'the given results by
            the Greedy is not the optimal one:')
            print('Result of the exhaustive search: ', exhaustive)
```

```
print('Result of the greedy function: ', greedy)
```

The following results were obtained by running the

automated-testing(input:list) for a maximal number of objects 10, since this is the limit of acceptable time one can wait to get an output from the exhaustive search. By looking at the results, usually the solution provided by the Exhaustive Search program, involves 1 or 2 containers, due to the rise of containers' maximal weight capacity while the number of objects gets higher, yet the weights of the objects still vary from 0 to 1kg. For the Greedy program, the obtained solution sometimes involves a container more than the optimal solution.

```
For input 6 the given results by the Greedy is not the optimal one:
Result of the exhaustive search:  [[6, 5], [4, 3, 2, 1]]
Result of the greedy function:  [[4, 6], [5, 3, 2], [1]]

For input 7 the given results by the Greedy is not the optimal one:
Result of the exhaustive search:  [[2, 4, 1, 7], [3, 6, 5]]
Result of the greedy function:  [[3, 2, 4], [1, 7, 6], [5]]

For input 8 the given results by the Greedy is not the optimal one:
Result of the exhaustive search:  [[7, 2, 6, 8, 4], [1, 3, 5]]
Result of the greedy function:  [[1, 7, 2, 6], [8, 4, 3], [5]]

For input 10 the given results by the Greedy is not the optimal one:
Result of the exhaustive search:  [[9, 8, 4, 6, 3], [1, 10, 7, 5, 2]]
Result of the greedy function:  [[1, 10, 7, 9, 8], [5, 4, 6, 3], [2]]
```

For the tested instances, we can conclude that the approximation obtained from testing is:

$$\frac{A(x)}{OPT(x)} = \frac{GreedySolution}{ExhaustiveSearchSolution} = \frac{3}{2} \leq 2$$

The worst case solution which the Greedy algorithm can give, involved at max 2 times more containers than the optimal solution.

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

6 The complexity of each approach

We recently compared the solutions provided respectively by both approaches and it is easily noticeable that the Greedy does not always give the optimal solution. But is this the only difference between two algorithmic approaches?

To answer to this question, we need to look at the complexity of each program, evaluating so the worst case computation time. By looking at the code and

explanation described before on this paper, the complexity of the Exhaustive Search program should be larger than the Greedy one, for as long as Exhaustive exploits also the Greedy during its run.

6.1 The predicted complexity

6.1.1 Greedy algorithm implementation

Taking a look at the structure of the Greedy function, what eyes notice are two ***while*** loops. Theoretically, loops represent iteration through all the elements of a list, in our case. Consequently, for a list of size ***n***, considering each of its elements two times, corresponds to a complexity of $O(n^2)$.

However, in the provided greedy function, this is not true. For the computations performed by both loops, we index the list only once, to subtract a certain element. Every element of the list is considered, because for as long as they belong to the list, their index is smaller than the length of the list, fulfilling so the condition of the first ***while*** loop. Once the first element is put into a container, we subtract the next element of the list.

In this way, going from one element of a list of length n, to the other, the complexity is linear, depending only in the value of n, $O(n)$.

6.1.2 Exhaustive search algorithm implementation

The implementation of the Exhaustive Search approach, exploits not only the greedy function, but the function computing all possible permutations as well. What the ***ex-s(list)*** does, is that it iterates through all the possible permutations, to which it applies the greedy function, called ***allocation(list)***. So for a list of n elements, there are n! possible permutations, to each of it is applied the greedy function, which considers all the n elements of this permutation one by one to insert them in a container. The complexity of permutation function is ***O(n!)***, the greedy has a complexity of ***O(n)***, so to total complexity of the exhaustive search program is ***O(n*n!)***.

The code used to obtain the results to evaluate the complexity

For both programs the same format of testing functions were used, differing only from the naming of the program to be tested.

```
def testES(n: int, repetitions=10) -> tuple:
    for i in range(repetitions):
        generate-input = generate-d(n)
        input = list(generate-input.values())
        with Timer(name='exhaustive-s', logger=None):
            ex_s(input)
    return round(Timer.timers.mean('exhaustive-s'), 6)


def Time-setsES(input-list:list)-> list:
    time-list=list()
```

```
    for input in input_list:
        result=testES(input)
        time-list.append(result)
    return time-list
```

The first function ***test()*** takes an integer which will be used to call ***generated(n)*** to create a dictionary with n pairs, then generate the weights-list to call the exhaustive search in the presented case (keep in mind that the same functions are used for the Greedy as well). This procedure is repeated a certain time, defined by the given number of the repetitions, all the times needed for the program to provide a solution will be stored and an average time for this computation to be performed will be given by ***codetiming*** module.

The second function instead, ***Time-sets(inputs-list)*** takes a list of integers, each of them is passed to the ***test()***, the average time for the execution of each integer value, will be stored in the ***time-list***, which is returned by the ***Time-sets(inputs-list)***.

To simplify the insertion of our programs result in the graphs, a function which creates pairs (input, running time) is implemented.

```
def print_pairs(time_list, value_list):
    for (n, t) in zip(time_list, value_list):
        print((n,t), end='')
```

It takes as a parameter the result of and the input given to the ***Time-sets(inputs-list)*** in the form of lists. Iterating in parallel in both lists, it creates the pairs and returns them as individual tuples.

6.2 Graphical representation

To evaluate the correctness of complexity prediction, the pairs provided by ***print-pairs(time-list, value-list)*** for each of the functions, are represented on a graph.

A crucial component of this analysis step is to build a graph which represents the predicted time complexity. How the parameters of these functions are calculated, will be represented in the functions' respective sessions.

6.2.1 Greddy function

The results and the input given to ***Time-setsG(inputs-list)***, identical to ***Time-sets(inputs-list)*** previously presented, computing the average running time of the ***allocation()*** greedy function, are used by the following function ***curveG(x, a, b)*** and ***scipy*** module, to give an approximation of the coefficients values, such that the obtained running time values fit to the predicted time complexity curve.

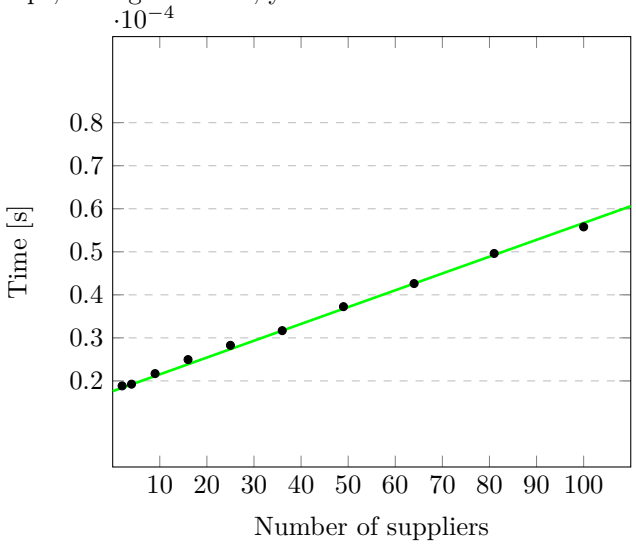
Running time for each input size in tuple format as given by the print-pairs(time-list, value-list)

```
>>(2, 1.8841e-05)(4, 1.924e-05)(9, 2.1696e-05)
(16, 2.4953e-05)(25, 2.8266e-05)(36, 3.1678e-05)
(49, 3.725e-05)(64, 4.2619e-05)(81, 4.9609e-05)
(100, 5.5786e-05)

def curveG(x, a, b):
    return a * x + b

print(optimization.curve_fit(curveG, inputs-list,
Time-setsG(inputs-list))
```

The result of curveG(x, a, b), coefficients of the linear function, predicted by the results of the running time of our greedy function.

```
>>array([3.90766950e-07, 1.76101957e-05])
```

Although the predicted time complexity is $O(n)$, the computed curve is displaced by ***b***, yet still linear. This happens because, although the input of the whole function can be 0 objects to be placed, the program will execute each of its steps, taking a certain, yet short amount of time.



In the graph one can notice that the dots which represents the values obtained by the testing, align almost perfectly to the linear function with coordinates provided by Python implemented function **curveG(x, a, b)** represented on a green motif. Hence we verify that the complexity of the Greedy function is $O(n)$.

6.2.2 Exhaustive search function

Exactly the same is proceeded for the Exhaustive search program. The results and the input given to ***Time-setsES(inputs-list)***, identical to ***Time-sets(inputs-list)*** previously presented, computing the average running time

of the $ex_s()$ exhaustive search function, are used by the following function $curveES(x, a)$ and ***scipy*** module, to give an approximation of the coefficients values, such that the obtained running time values fit to the predicted time complexity curve.

However, since the complexity of the exhaustive search is much more higher compared to the greedy program, due to usage of permutation, we cannot test it for the same huge values. By manual testing, for an input 10, the required time is 2 minutes. For an input 12, the time needed would be

12 * 11 * 2 minutes = 4,4 hours.

```
>>(2, 0.000051)(3, 0.000118)(4, 0.000208)
(5, 0.000718)(6, 0.003177)(7, 0.019108)
(8, 0.132226)
```

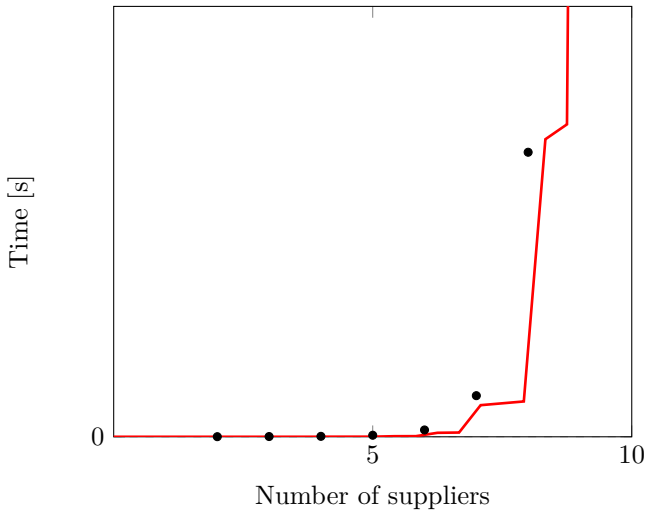
```
def curveES(x, a):
    return a * x * scipy.special.factorial(x)
```

```
print(optimization.curve_fit(curveES, inputs-list,
Time-setsES(inputs-list))
```

The result of curveES(x, a), coefficients of the linear function, predicted by the results of the running time of our greedy function.

```
>>(array([3.78143447e-07])
```

One must take into consideration that the factorial graph weird motif, occurs because the visual representation is stretched out only on the X axis.



The graph representation shows an almost perfect alignment of the dots built based on the **print-pairs(time-list, value-list)** result and the factorial graph

whose coefficient is determined by **curveES(x, a)**. Hence we prove that the time complexity of the exhaustive search function is ***O(n*n!)***.