Second Assignment
- Algorithms and Data Structures in Biology -

Monica Nicolau

1 Combinatorial optimization problem

In the following table, there is the model I propose for the combinatorial problem presented in the second assignment that can be seen as an instance of the Bin Packing problem.

"Bin Packing" combinatorial problem

Given a set of n items with their respective weights and an unlimited number of bins with capacity C, assign each item to one bin so that the total weight of the items in each bin does not exceed C and the number of used bins is minimum.

Input: The number of items n, the list of weights $W = (w_1,w_2,...,w_n)$, where w_i is the weight of the i^{th} item and the capacity C.

Output: The minimum number of bins strictly necessary to pack all the items and the packaging that derives.

2 Exhaustive search algorithm

To solve the previously described combinatorial problem, I have designed first an exhaustive search algorithm. The algorithm is able to find the optimal way to pack all the items, between all the valid ones, that minimizes the number of containers needed to ship all the items to their final destination. One sub-routine has been realised to check the validity of a bin.

Exhaustive Search Algorithm to obtain the optimal solution for the Bin Packing problem

```
1:  function BRUTE FORCE PACKING(number of items, weights, capacity)
2:      optimal number of bins ← number of items
3:      optimal packing ← the worst packing with each item in a different bin
4:      for each choice in GENERATE ALL VALID CHOICES(weights, capacity) do
5:          if length of current choice ≤ optimal number of bins then
6:              optimal packing ← choice
7:              optimal number of bins ← length of current choice
8:          end if
9:      end for
10:     return optimal packing and optimal number of bins
11: end function
```

The Brute Force Packing function takes as arguments the number of items, the list with the weight of each item, the fixed space capacity of the bins and returns the optimal packing in order to pack all the n items in the bins and the corresponding number of bins, which is the minimum that could ever be found.
Since this is a **minimization** problem, the variables *optimal number of bins* and *optimal packing* are first initialized to the worst case solutions that are the total number of items and the packaging in which each item is in a different bin, respectively. In this way the variables will always store the lowest values encountered so far during the iteration of the `for` loop.

2.1 Exhaustive search algorithm's complexity

In this section I will analyse the time complexity of the algorithm previously described. I will use a Big O (also known as asymptotic) notation that defines a relevant upper bound to the worst-case computation time in function of the size of the input n (i.e. number of items).

The first algorithm to be performed during the execution of Brute Force Packing is the one that generates all the valid bins given the list of *weights* and the *capacity*. Both the time and the space complexity of Generate All Valid Choices should be considered to have a more accurate analysis of the overall main algorithm complexity. I decided to introduce the Python code of the latter function and of the Check Validity one in order to analyse its complexity unambiguously.

```
1   def check_validity(box, weights, capacity):
2      weight_sum = 0.
3      for item in box:
4         weight_sum += weights[item]
5      if weight_sum <= capacity:
6         return True
7      return False

8   def generate_all_valid_choices(weights, capacity):
9      items = list(range(len(weights)))
10     number_of_items = len(weights)

11     def sub_generator():
12        nonlocal items, number_of_items
13        if len(items) == 1:
14           yield [items]
15        else:
16           first = items[0]
17           items = items[1:]
18           for partition in sub_generator():
19              for i, subset in enumerate(partition):
20                 if check_validity([first]+subset,weights,capacity):
21                    yield partition[:i]+[[first]+subset]+partition[i+1:]

22              if len([[first]] + partition) < number_of_items:
23                 yield [[first]] + partition
24           items = items[:]

25     yield from sub_generator()
```

The biggest contribution to the complexity of the generator function comes from Sub Generator, that is a recursive function. The complexity function of a recursive algorithm can be expressed by means of linear recurrence relations with constant coefficients.

Let n be the size of the input of Sub Generator, the amount of space $S_s(n)$ generated by the function is equal to the space when the size of the input is $n-1$, $S_s(n-1)$, since Sub Generator is recalled within itself with $n-1$ (because of the slicing performed on items at line 17), multiplied by n because the nested for at line 19 will iterate at most n times (since partition in the worst case is n). Hence the *space complexity* $S_s(n)$ can be defined as follows:

$$S_s(n) = \begin{cases} 1 & \text{if } n = 1 \text{ (base case)} \\ n \cdot S_s(n-1) & \text{if } n > 1 \end{cases} \tag{1}$$

Let's now analyse the time complexity of Sub Generator, that can be defined in a recursive way too, as explicated in system (2). The relevant instructions contained in the function are:

- the for loop (line 18) which iterate n^n times;
- the nested for loop (line 19) which iterates at most n times;

- and the nested `if` statement (line 20) that has complexity $O(n)$ because of the Check Validity function it calls, that in turn contains a `for` loop that iterates at most $n-1$ times (since each bin –one of the arguments of the function– can contain at most $n-1$ items).

$$T_s(n) = \begin{cases} 1 & \text{if } n = 1 \text{ (base case)} \\ T_s(n-1) + n^2 \cdot S_s(n) & \text{if } n > 1 \end{cases} \tag{2}$$

From system (1) can be inferred that $S_s(n) = O(n^2)$. If this value is substituted in (2), we obtain $T_s(n) = O(n^2 \cdot n^n)$ that is equal to $T_s(n) = O(n^{n+2})$. So, the resulting time complexity of Sub Generator is $T_s(n) = O(n^{n+2})$.

Going back to the Brute Force Packing time complexity (that from now on will be referred to as $T(n)$) analysis, it is evident that the `for` loop at line 4 of the Brute Force Packing algorithm, iterates over the space generated by the Sub Generator function. The complexity of the operations within the before mentioned `for` loop is $O(n)$, because the assignment of *optimal packing* to *choice* cost $O(n)$ (since it is making a copy of an array). So, $T(n)$ will be equal to n times the space complexity of Generate All Valid Choices plus the time it takes to actually generate all the valid choices, that can be translated in mathematical terms as $T(n) = n \cdot S_s(n) + T_s(n)$. Then $T(n)$ can be computed as follows:

$$\begin{aligned} T(n) &= n \cdot S_s(n) + T_s(n) \\ &= O(n \cdot n^n + n^2 \cdot n^n) \\ &= O(n^2 \cdot n^n) \\ &= O(n^{(n+2)}) \end{aligned}$$

The move from the second line to the third one holds since $O(n \cdot n^n)$ is considerably outperformed by $O(n^2 \cdot n^n)$. So in the end, the overall theoretical time complexity of the Brute Force Packing algorithm in Big O notation is $O(n^{n+2})$.

3 Greedy algorithm

To solve the combinatorial problem described in **section 1**, I have designed also a greedy algorithm. The function First Fit takes as arguments the total number of items n, the list with the *weights* of each item and the space capacity of the bins and returns the minimum number of bins to be used in order to pack all the n items and the corresponding packaging. Here follows the pseudocode of the First Fit algorithm.

Greedy Algorithm for the Bin Packing problem

```
1: function First Fit(n, weights, capacity)
2:     ($bin_1$ ... $bin_n$) ← (∅ ... ∅)
3:     $number\ of\ bins$ ← 0
4:     ($spaceinbin_1$ ... $spaceinbin_n$) ← ($capacity$ ... $capacity$)
5:     for each item ← 0 to n do
6:         for each box ← 0 to $number\ of\ bins$ do
7:             if $spaceinbin_{box} \geq weights_{item}$ then
8:                 add item to $bin_{box}$
9:                 $spaceinbin_{box} \leftarrow spaceinbin_{box} - weights_{item}$
                                        ▷ update the remaining space in the current box
10:            end if
11:        end for
12:        if item has not already been used then
13:            add item to a new bin
14:            space in the new bin ← $capacity - weights_{item}$
15:            increase $number\ of\ bins$ by 1
16:        end if
17:    end for
18:    return prefix of ($bin_1$ ... $bin_n$) consisting of non-empty subsets and $number\ of\ bins$
19: end function
```

3.1 Greedy algorithm's complexity

In this section I will analyse the time complexity of the algorithm previously described and as in **section 2.1** I will use the Big O notation. For each relevant instruction in the First Fit algorithm, I will indicate the number of times it is executed. The operations contained in the main `for` loop, starting at line 5, are executed n times. Then, it is possible to iterate over the nested `for` loop at most n times, since in a worst-case scenario each item will be packed in its own bin (which correspond to a total of n bins). The membership check at line 12 could have cost $O(n)$, but since in the `Python` implementation *already_used_items* is a set, the complexity of this search (in the hash table) approaches $O(1)$ as the input size increases and do not affect the final time complexity. So, the overall time complexity of First Fit is quadratic with $O(n^2)$.

3.2 Approximation ratio

The theoretical approximation ratio of First Fit has been proved to be less or equal 2 in one of our classes (by means of two lemmas). More accurate calculations (that I have found in the literature) have pointed out that a slightly more precise upper bound to First Fit approximation ratio is 1.7. In the **Conclusion section** of this document I will show the empirical results I have obtained.

4 Automated Testing routine

In making up two instances of the Bin Packing problem (the first with $n = 4, C = 2$ and the second with $n = 6, C = 2.5$), I chose to draw the weights in such a way that $w_i \in [0, C]$ and not to [0, 1], so it will be easier for me to choose an example where the greedy algorithm fails (i.e. returns a *sub-optimal* solution) as the first instance that follows.

Instance	Capacity (C)	Weights
First	2	[1.0, 1.4, 0.6, 1.0]
Second	2.5	[0.50, 1.25, 1.00, 1.75, 0.75, 1.50]

Suppose the list of the items' identities of the first instance is [0, 1, 2, 3] and for the second one is [0, 1, 2, 3, 4, 5]. As expected, the two algorithms are able to return the same solutions that I have computed by hand.
The optimal solution for the first instance is made by two bins and they are [0, 3] and [1, 2] – where the square brackets surround the content of a bin. The brute force algorithm returns exactly this solution, while the greedy one returns a sub-optimal solution that include three bins ([0, 2], [1], [3]). The optimal solution for the second instance is made by three bins and both the algorithms returned a packaging with the optimal number of bins. The brute force algorithm returned the following packaging: [0, 3], [1, 2], [4, 5], while the greedy one returned [0, 1, 4], [2, 5], [3] as my hand-solved solutions forecasted.

5 Benchmarking routine

For the benchmarking routine, I have used the `random` module.
I have defined two functions for this purpose that I have chosen to insert in this report.

```
items = [4, 6, 8, 10, 12, 14, 16, 18, 20]
def benchmarking_routine(bpp_function, n):
   C = n**(1/2)
   weights = [random.uniform(0,1) for i in range(n)]
   while sum(weights) <= C:
      weights = [random.uniform(0,1) for i in range(n)]
   return bpp_function(n, weights, C)

def print_benchmarking_routine(items):
   for n in items:
      print('Number␣of␣items:',n)
      if n <= 13:
            print('The␣BRUTE␣packing␣and␣the␣optimal␣number␣of␣bins␣are:␣',
            benchmarking_routine(brute_force_bin_packing, n))
        else:
            print('The␣selected␣number␣of␣items␣is␣too␣high␣for␣the␣brute␣force␣solution')
        print('The␣GREEDY␣packing␣and␣the␣greedy␣number␣of␣bins␣are:␣',end='␣')
        print(benchmarking_routine(first_fit, n))
```

I have designed the interface of both the exhaustive search function and the greedy one in the same way, in order to have an easier implementation of the `benchmarking routine` function. Both my functions (Brute Force Packing and First Fit) expect three arguments as input: the number of items n, the list with their *weights* and the space *capacity*. As suggested in the text of the assignment, I have setted the capacity to the square root of n and I have uniformly drawn the *weight* values from the interval $[0, 1]$. To be sure that the condition for which "C (*capacity*) is typically smaller than $\sum_{i=1}^{n} w_i$" has been fulfilled, I have inserted a `while` loop that will provide the list of *weights* only if their total sum is greater than the computed *capacity*. In human words, it cannot happen that all the items can be packed in only one bin, otherwise it would be pointless (and actually trivial) to solve the bin packing problem.
Inside the `for` loop of the print benchmarking routine function I have inserted an `if` statement that will compute the brute force solution only if the number of items is less or equal to 13. This was essential for making the brute force function executable in a reasonable amount of time. Fortunately, the greedy function is faster and can return a result basically instantaneously even for inputs in the order of magnitude of 10^4.

6 Graphical presentation of the results

Finally, I have experimentally test the runtime of my two programs using the `timeit` module. I have designed a function `runtime` that takes as arguments the desired function that solve the Bin Packing Problem – `bpp_function` – and the number of items n I want to test the function for. Then, I have performed the `timeit` analysis recalling the Benchmarking Routine function that generates random instances of weights, given n. I have set the `number` parameter in the `.timeit()` function to 2 for the brute force function in order to have the results in a reasonable amount of time and to 100 for the greedy one, that is much faster.
Then I have stored the outputs of the `runtime` function in a list generated by list comprehension, `brute_data` for the brute force function and `greedy_data` for the greedy one. For each of these lists, I have zipped it with the list containing the number of items' instances in order to get the coordinates in an understandable way for `pfgplots`. The so obtained data points are represented in the graphs in Figures 1 and 2 as blue dots.
As the reader can observe, in the graph representing the brute force function runtime performance, the data points' trend follows an exponentiation one, typical of the exhaustive search algorithms, while in the graph representing the greedy function runtime performance, the data points' trend follows a quadratic one, as predicted by the theoretical time complexity (of the First Fit function).
Then, I have used the `scipy.optimize` module to compute the parameters of the two best-fit curves that match my algorithms' complexity. I have also defined two functions (`exponentiation` and `quadratic`) which return the equations of the complexity that will be the inputs of the `scipy.optimize.curve_fit()` function. In this way I have found the parametric relationships that best fit the obtained data points and they are:

$$\text{Model for the Brute Force function: } t = 1.09 \cdot 10^{-7} \cdot (n^{(5.18 \cdot 10^{-1}) \cdot (n+2)}) + 8.46 \cdot 10^{-3}$$

$$\text{Model for the Greedy function: } t = (7.23 \cdot 10^{-9}) \cdot n^2 + (1.43 \cdot 10^{-6}) \cdot n + 2.24 \cdot 10^{-5}$$

where n is the number of items, and t is the time (in seconds) it takes to solve the problem. The models are graphically represented in the Figures 1 and 2 as brown plots.

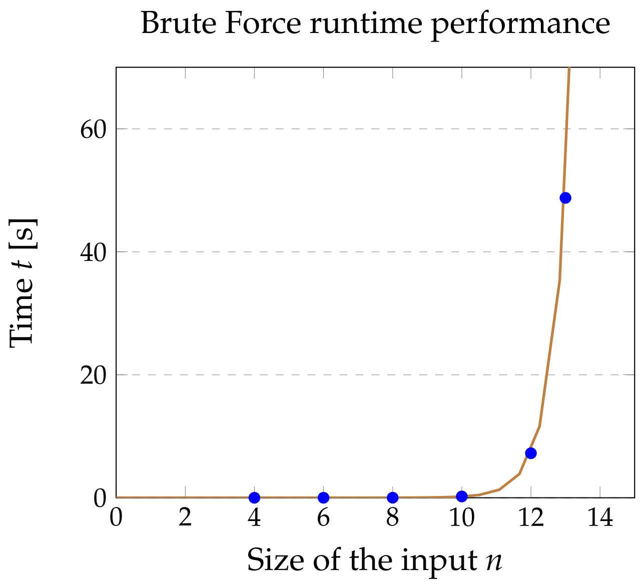

Figure 1: Time t to compute `brute_force_packing()`.

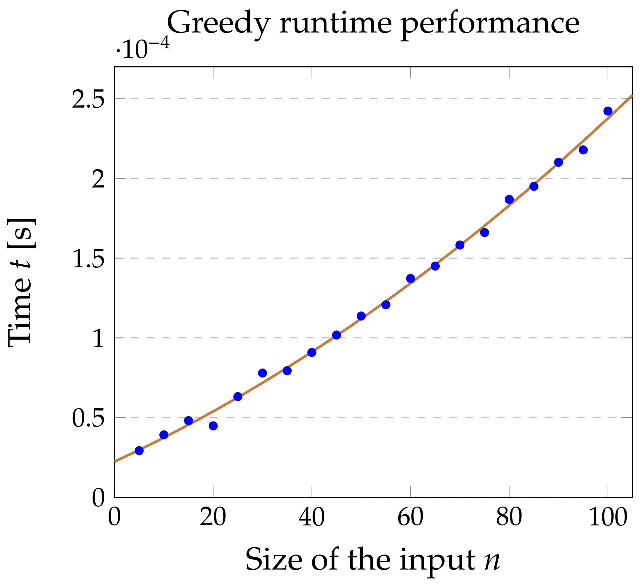

Figure 2: Time t to compute `first_fit()`.

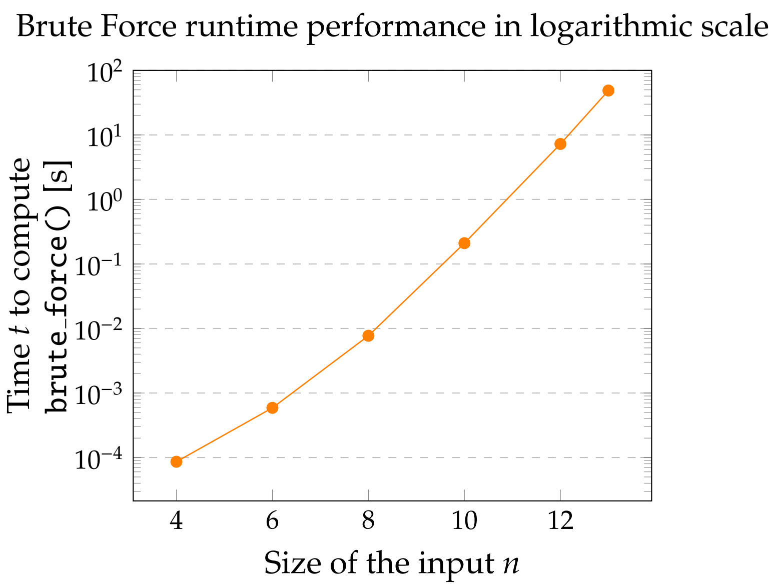

Figure 3: Time t to compute `brute_force_packing()`.

A version of the Brute Force runtime performance graph (Figure 1) has been realised (in Figure 3) by applying a logarithmic scale on the vertical axis in order to highlight its super-exponential trend.

7 Conclusion

The models suggested by the theoretical complexity analysis fit the actual, measured computation times substantially well, which suggests that the complexity analysis has been done in the proper way for both the algorithms. Moreover, I have designed an Approximation Ratio Routine function that will compute the ratio between the number of bins used by the greedy solution and the number of bins used by the optimal (so, the brute force) solution, whenever the First Fit function returns a sub-optimal solution.

Observation: If the capacity is $\sqrt{n}$ as the input size n increases, the greedy function tends to perform very well, i.e to return the same number of bins as the optimal solution. This is because the *weights* are still drawn from [0, 1], while the *capacity* grows. By fixing the *capacity* value to 1, I am able to analyse better the approximation ratio's behaviour.
With this constraint, I have observed an approximation ratio of 1.5 (in the worst-case scenario), that is the highest ratio I obtained experimentally. This result is consistent with the theoretical bounds established in the **section 3.2** since it has been proven that the approximation ratio of First Fit can be at most 1.7 (and $1.5 \leq 1.7$).

As an additional study, I propose a comparison between the runtime performances (as functions of the size of the input n) of the two algorithms designed by me on randomly generated inputs. This investigation is useful to verify if it is actually advantageous to use the greedy algorithm instead of the brute force one, especially for very large values of n.

A logarithmic scale has been applied on the vertical axis of the graph in Figure 4 to allow an easier confrontation and to enhance the wide difference between the runtime performance of the two algorithms. A quick look to the plots is enough to notice that the Brute Force Packing algorithm requires a high amount of seconds to return the optimal solution even for inputs where the number of items is less than 20 (i.e. its time complexity function grows very fast), while the First Fit algorithm is able to return an output, which most of the times is an optimal one, (almost) instantaneously even when there are 100 items to be packed (of course, respecting all the constraints defined before in the paper).

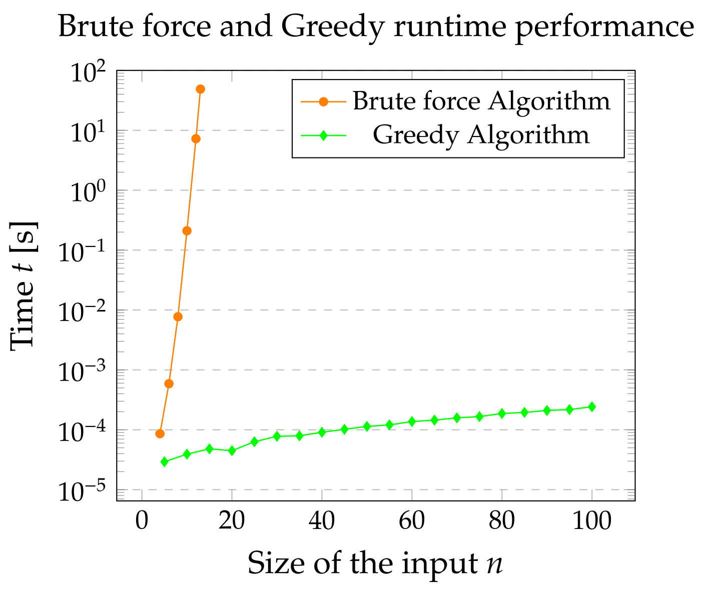

Figure 4: Comparison between the time to compute `brute_force_packing()` and the time to compute `first_fit()`.

Thank you for your attention!