# 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 assignment.

**"Pharmaceutical company PURCHASE PLAN"** combinatorial problem

*Given the list of suppliers along with the weight each one can provide and the list of incompatibilities for each supplier, determine the optimal purchase plan.*

**Input:** The list of suppliers $S = (s_1, s_2, ..., s_n)$, the list of weights $W = (w_1, w_2, ..., w_n)$, where $w_i$ is the weight of the supplier $s_i$, and the list of their corresponding incompatible suppliers $L = (L_1, L_2, ..., L_n)$

**Output:** The list of suppliers to be selected such that $\sum_{i=1}^{k} w_i$ is maximized.

## 2 Exhaustive search algorithm

To solve the previously described combinatorial problem, I have designed an exhaustive search algorithm. I have divided the algorithm in two parts: one checking the compatibility between the suppliers and one that find the list of suppliers that ensures the purchase of the maximum amout of the chemical substance, observing the defined incompatibilities. Here follow the first part:

**Algorithm** to verify compatibility of suppliers

```
function COMPATIBLE(choiches, L)
    for each supplier in choices do
        for each incompatible present in the list of L corresponding to supplier do
            if incompatible in choices then
                return False
            end if
        end for
    end for
    return True
end function
```

The function COMPATIBLE takes as arguments `choices`, that is the set of suppliers currently under analysis, and `L` which contains the incompatibilty list for each supplier. Then, for each supplier in the set choices, the function checks if any incompatible element is present by iterating over the incompatibilty list $L_i$ of the currently selected supplier; if a conflict is found the function will return the `False` boolean, otherwise it will return `True`.

The second part of the algorithm select the optimal list of suppliers that maximizes the amount of chemical substance purchased.

**Algorithm** to select the optimal purchase plan

```
1: function OPTIMAL_PURCHASE_PLAN(suppliers, weights, incompatibilities)
2:     best_combination ← empty list
3:     max weight ← 0                              ▷ set to 0 since I want to maximize this parameter
4:     for each combination in ALL_POSSIBLE_COMBINATIONS(suppliers, weights) do
5:         if weight of combination is greater than max weight then
6:             if COMPATIBLE(combination, incompatibilities) then
                                   ▷ alias if COMPATIBLE(combination, incompatibilities)==True
7:                 best_combination ← combination
8:                 max weight ← weight of combination
9:             end if
10:        end if
11:    end for
12:    return best_combination
13: end function
```

The OPTIMAL_PURCHASE_PLAN function takes as arguments the list of suppliers, their weights and the list containing the incompatibility list for each supplier. To have an easier implementation I though of ALL_POSSIBLE_COMBINATIONS as a function that generates all the possible subsets of suppliers associated to the sum of the weight they offer, e.g. suppose $S = (s_1, s_2, s_3, s_4, s_5)$ is the list of suppliers my company has access to and $W = (w_1, w_2, w_3, w_4, w_5)$ contains the respective grams of substance they can provide, then one of the possible elements generated by the function will be $((s_1, s_3, s_5), (w_1 + w_3 + w_5))$, where $(w_1 + w_3 + w_5)$ is obviously a positive float number. Since this is a **maximization** problem, the variable *max weight* is first initialized to 0 in such a way that it will always store the greatest value encountered so far during the iteration of the `for` loop.

In the appendix of this document, the reader can find the algorithm for ALL_POSSIBLE_COMBINATIONS, that I have used to compute all the possible subsets given the set of suppliers.

## 3 Algorithm's complexity

In this section I will analyse the complexity of the main algorithm described previously. I will use a big O notation that defines a relevant upper bound to the worst-case computation time in function of the size of input $n$, which is the total number of suppliers. The first algorithm to be performed during the execution of OPTIMAL_PURCHASE_PLAN is the one that generates all the possible subsets of a given set $S$. If $S$ has length $n$, the complexity of ALL_POSSIBLE_COMBINATIONS will be $O(n \cdot 2^n)$ and its output will have size $2^n$.
Therefore, the instructions contained in the for loop starting at line 4 of the main algorithm are repeated $2^n$ times. They are two `if` statements that contribute to the time complexity in the measure of what their statments demand. The second `if` (line 6) recalls the COMPATIBLE function, so let's analyse in detail the time complexity of this algorithm.

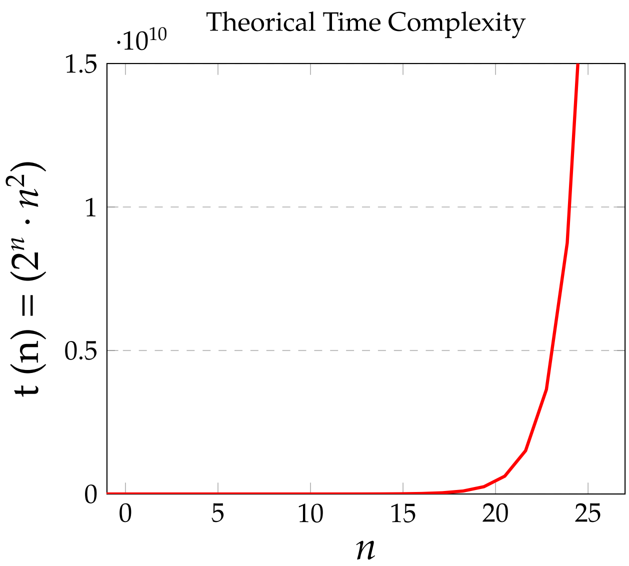
Considering the worst-case, the size of the input could be at most $n$. A quick look to the `Python` implementation of this algorithm would be helpful to better understand how I have computed its complexity.

```python
def compatible(choices, L):
   set_of_choices=set(choices)
   for supplier in choices:
      for incompatible in L[supplier]:
         if incompatible in set_of choices:
            return False
   return True
```

Before doing anything else, the function converts the input choices (that comes in the form of a list) into a set. This instruction takes $O(n)$. The conversion has actually a strong impact on its time complexity, because in order to find an element in a set, a hash lookup is used. This makes the "in" membership operator a lot more efficient for sets than lists.

The first for loop is executed $n$ times. The algorithm now checks for each element if it is present in list of incompatibles of all the others that are long at most $(n-1)$ and for each component of the $(n-1)$ checks if this component is present in the set_of_choices or not. Thanks to the conversion to set, the complexity of this search (in the hash table) approaches $O(1)$ as the input size increases. In the end, we have $T(n) = n \cdot (n-1) \cdot 1$ and the $O(n)$ deriving from the initial conversion, which is overwhelmed by the complexity of the rest of the algorithm. Indeed if I sum up $T(n) = n \cdot (n-1) \cdot 1$, which leads to $O(n^2)$, to $O(n)$ it results into $O(n^2+n)$, that is equal to $O(n \cdot (n+1))$. Since in a Big-O runtime analysis all the constant factors must be removed, the global time complexity of the COMPATIBLE algorithm is $O(n^2)$.

To sum up, the overall time complexity of OPTIMAL_PURCHASE_PLAN is $O(2^n \cdot n^2)$.
Below, I have inserted a graphical representation of how the theorical time complexity will grow as a function of the input size $n$.




## 4 Testing routine

For the testing routine, I have used the random module. I have defined a function for this purpose that I have choosen to insert in this report.

```
lengths=[4,8,10,12,14,16,18,20,22,24]
def testing_routine(n):
    suppliers=list(range(n))
    weights, incompatibilities = dict(), dict()
    for s_i in suppliers:
        weights[s_i] = random.uniform(0,1)
        tc_suppliers = set(suppliers)
        tc_suppliers.discard(s_i)
        incompatibles_for_s_i = set(random.sample(tc_suppliers,n//2))
        incompatibilities[s_i] = incompatibles_for_s_i
    return optimal_purchase_plan(suppliers, weights, incompatibilities)

for n in lengths:
    print('Number␣of␣suppliers:',n)
    print('The␣optimal␣purchase␣plan␣to␣be␣selected␣is:␣',end='␣')
    print(testing_routine(n))
```

My OPTIMAL_PURCHASE_PLAN function expect as weights and incompatibilities inputs two dictionaries. So first I have inizitialized the two variables to empty dictionaries and exploiting a for loop I have assigned to each supplier (which identifier is the key of the dictionary) its weight (weights dictionary) and its list of incompatible suppliers (incompatibilities dictionary).

I chose to use the `.sample()` inbuilt function of `random` module to generate the incompatibilities lists $L_i$ , since it allows me to return a particular length (in this case $\frac{n}{2}$) list of items – without replacement – chosen from a temporary set of suppliers from which I have removed the current one in order to be consistent with the concept of incompatibility (for which a subject cannot be incompatible with himself).

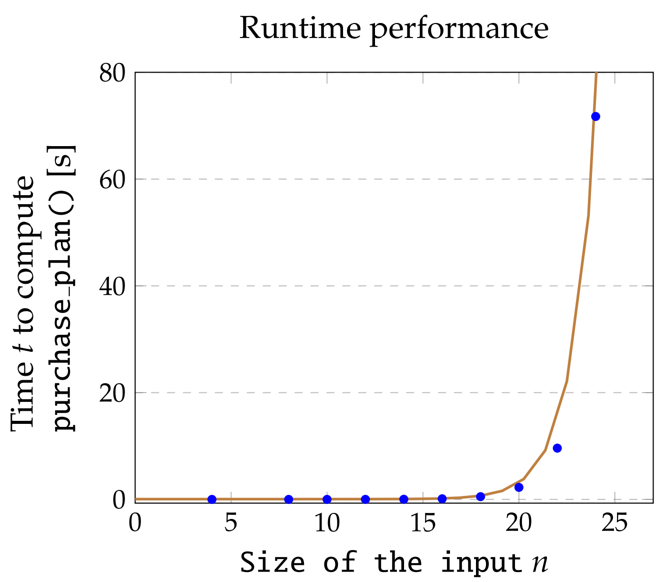
I've chosen smaller $n$ values than the ones suggested, to make the code executable in a reasonable amount of time. Since I wanted an empirical proof of this, I have left the code running overnight and after 13 hours there's still no sign of the results for the $n = 32$ case. This behaviour might be due to the small computational power of my computer, but I sincerly would blame the nature of the code, that is a Python implementation of an **Exhaustive search** algorithm whose cost tends to grow very quickly as the size of the input increases, so it is legit to infer that is the reason why I wasn't able to compute the optimal solution for $n = 32$ and therefore neither for $n = 40$.

## 5 Graphical presentation of the results

Finally, I have experimentally test the runtime of my program using the `timeit` module.
I have designed a function `runtime` that takes as arguments the `testing_routine` function and the list containing all the $n$ values I want to test the program for. I have set the `number` parameter in the `.timeit()` function to 10 in order to have the results in a reasonable period of time. To clarify any doubts, I explicit that the `.timeit()` function returns an output in seconds.
Then I have stored the outputs of the `runtime` function in a list and I have zipped the list with the $n$ values and the beforementioned one in order to have the coordinates in an understandable way for `pfgplots`. The so obtained data points are represented in the graph inserted below as blue dots. As the reader can observe, their trend follows an exponetial one, typical of the exhaustive search algorithms.
Then, I have used the `scipy.optimize` module to compute the parameters of the best-fit curve that matches my algorithm's complexity. I have also defined a function which returns the equation of the complexity that will be the input of the `scipy.optimize.curve_fit()` function. In this way I have found the parametric relationship that best fits the obtained data points and it is:

$$t = 7.37^{-9} \cdot (2^n \cdot n^2) + 5.49^{-2}$$

and it is graphically represented below as the brown plot.




Thank you for your attention!

# 6 Appendix

**Algorithm** to find all possible subsets of suppliers with respective sum of weights

```
function ALL_POSSIBLE_COMBINATIONS(suppliers, weights)
    records ← (empty list, 0)
    for supplier in suppliers do
        n ← length of records
        for i ← 0 to n do
            new_list ← copy of the first element of the i^th tuple in records    ▷ copy of records[i][0]
            new_list ← new_list extendend by supplier
            records ← records extended by (new_list, weight of element in position i + weight of supplier)
        end for
    end for
    return records
end function
```