Algorithms and Data Structures for Biology
5 may 2019 — Assignment number 1

Daniele Traversa; ID:0000830457

1 The Problem

Suppose you are a biologist who will soon move to the franco-italian Antartic Station1. The Station is not covered by broadband Internet, and so most of the needed data must be brought with you. You can bring with you only one hard disk with a capacity of n gigabytes. Inside the hard disk you need to fit as many of your databases as possible. You are currently working with m different databases, call them $1.....m$, with database j having a size equal to S_j megabytes. Please notice that S_j is much bigger than n gigabtyes How should you decide which ones of the databases you should bring with you? You first attribute to each database a measure U_j of the utility the database has for you. Then, you realise you want to bring with you a set of databases with maximal total quality among those which fits into your hard disk.

2 Description of the combinatorial problem

Given a set of objects $1.....m$ and an integer number n. To each object j of $1.....m$, we assign two integer numbers S_j and U_j. Consider that: $\sum_{j\in 1.....m} S_j > n$. Find the subset D of $1....m$ such that:

$$\sum_{i\in D} U_i \text{ is maximal and } \sum_{i\in D} S_i \leq n \text{ where } D \subseteq 1.....m.$$

3 The Algorithm-Exhaustive Search

`REQUIRE`: An integer number m an other integer number n and two arrays S and U containing integer numbers such that both S and U have size equal to m.

`ENSURE`: Return the subset D of integer numbers of $1.....m$ such that: $\sum_{i\in D} U_i$ is maximal and $\sum_{i\in D} S_i \leq n$ where $D \subseteq 1.....m$.

The pseudo code that has been constructed for the development of the algorithm, is composed of two functions:

- the first function `setOfPossibleCombinations({1.....m})` is fundamental since it defines the search space, so the environment where the algorithm has to search the subset D of $1.....m$ that maximise the $\sum_{i\in D} U_i$ and has $\sum_{i\in D} S_i \leq n$. So this function is responsible of building all possible subsets of $1.....m$ having size from 0 up to m, which represents all possible combinations through which the elements in $1.....m$ can be combined. The function starts from all possible subsets of size 1 and from these, it builds all possible combinations of subsets of size increased by one. The results are collected in two arrays: one called `result` that at the end of the execution will contain the entire search space; another one called `volPermu` which temporary becomes the starting point for computing the subsets of higher size. The function keep on computing the different combinations until the last created one has a size equal to the set $1.....m$,meaning that it has generated all possible subsets of all sizes. The function avoids to compute redundant subsets meaning that, if we consider the subset $1, 2, 3$ has the same meaning of the subset $2, 1, 3$;

- the second function `ExhaustiveSearchAlgorithm(n,m,S,U)` analyzes each element of the search space. As it analyzes the current subset, it evaluates the $\sum_{i \in subset} S_i$ and the $\sum_{i \in subset} U_i$ keeping on track of just those subsets where $\sum_{i \in subset} S_i \leq n$ and $\sum_{i \in subset} U_i$ being maximal. So at the end, the function will return the best possible solution.

SETSOFPOSSIBLECOMBINATIONS($\{1.....\text{M}\}$)
  $result \leftarrow$ array of all possible subsets of length 1 and 0
  $permu \leftarrow$ array of all possible subsets of length 1
  while $|lastelementinresult| \neq |1.....m|$ **do**
    $volPermu \leftarrow$ empty array
    for all elements e **in** $permu$ **do**
      $start \leftarrow$position of the last character of e in $1.....m$
      $i \leftarrow$ start $+1$
      while $i \neq |1.....m|$ **do**
        $vol \leftarrow e + m_i$
        $volPermu \leftarrow volPermu + vol$
        $i \leftarrow i + 1$
      end while
      $result \leftarrow result + volPermu$
      $permu \leftarrow volPermu$
    end for
  end while
  return $result$

How does this function generates the search space? Basically, it starts considering all possible subsets of $1.....m$ having size equal to one. Then,per each subset under analysis, the function uses a for loop and a nested while loop to generate all possible subsets of size being increased by one. The for loop selects the starting subset and then the while loop creates all different new subsets of size increased by one by adding one element of $1.....m$ that is not already present in the subset under analysis. In this way, the function avoids to create redundant combinations. Once the for loop has evaluated all subsets of the current size, it starts again considering those that has been just generated, so those having a size being increased by one. The process continues until all subsets of all different sizes has been generated. In fact, the overall iteration process is controlled by a while loop that, once new subsets has been generated, checks their size. If the size of the last element added in the power set is equal to the one of $1.....m$,its guard fails meaning that all possible subsets of all possible sizes has been created and collected in the search space.

EXHAUSTIVESEARCHALGORITHM(N,M,S,U)
  $databases \leftarrow \{1.....\text{m}\}$
  $sets \leftarrow$ setsOfPossibleCombinations(databases)
  $databases \leftarrow$ empty array
  $maxU \leftarrow 0$
  for all elements e in sets **do**
    $volS \leftarrow 0$
    $volU \leftarrow 0$
    for all elements i in e **do**
      $volS \leftarrow \sum_{i \in e} S_i$
      $volU \leftarrow \sum_{i \in e} U_i$

end for
if $volS \leq n$ and $volU > maxU$ **then**
$maxU \leftarrow$ volU
$databases \leftarrow$ e
end if
end for
return $databases$

This function is responsible to evaluate each element in the search space. The latter is generated by calling the function `setsOfPossibleCombinations({1.....m})`. Once the search space is available, the search for the subset D of $1.....m$ that has $\sum_{i \in D} S_i \leq n$ and $\sum_{i \in D} U_i$ maximal is performed by two for loops, nested one inside the other. The first considers the subsets in the power set, each by each. The second, calculates the $\sum_{i \in subset} S_i$ and the $\sum_{i \in subset} U_i$ of each element considered in the first for loop. Then, these summations are validated by an if statement that keeps on track of best subset from $1.....m$ having $\sum_{i \in D} S_i \leq n$ and $\sum_{i \in D} U_i$ maximal. In the final output, the best combination is reported.

4 The Implementation-Exhaustive Search

The algorithm has been developed in `python` language `(version(3.7))`. The program is structured of two python functions, each performing a specific role.

The following Python function `setsPos(m)` implements the above algorithm for the creation of all possible subsets of $1.....m$ of different sizes representing all possible combinations through which one can organize elements of the set $1.....m$. So this python function is responsible of constructing the search space.

```
def setsPos(m):
        result = [[],] + [[y,] for y in m]
        permu = [[x,] for x in m]
        while len(result[-1]) != len(m):
                volPermu = []
                for e in permu:
                        end = m.index(e[-1])
                        i = end+1
                        while i < len(m):
                                vol = []
                                vol = e + [m[i],]
                                volPermu += [vol,]
                                i+= 1
                result += volPermu
                permu = volPermu

        return result
```

The following python function `ExhaustiveSearch(n,m,S,U)` implements the above algorithm for searching among all possible subsets of $1.....m$, the one D having $\sum_{i \in D} U_i$ is maximal and $\sum_{i \in D} S_i \leq n$

```
def ExhaustiveSearch(n,m,S,U):

    databases = [z for z in range(m)]
    sets = setsPos(databases)
    maxU = 0
    database = []
```

```
for e in sets:
    volS = 0
    volU = 0
    for i in e:
        search = databases.index(i)
        volS += S[search]
        volU += U[search]
    if volS <= n and volU > maxU:
        maxU = volU
        database = e

return database
```

5 Proof Time Complexity-Exhaustive Search

We want to calculate the time complexity of the algorithm for any value m. As the value of m increases, the time needed by the algorithm to execute all the instructions increases exponentially. Considering the function `setsOfPossibleCombinations(m)`, we observe that the output of this function has a size equal to 2^m. So it produces an array carrying such number of elements which basically represents our search space. We can say that the output of this function is an array counting 2^m elements simply because it calculates the `power set`, that considers all kinds of possible subsets, avoiding to compute redundant elements. Therefore, we can conclude that the total number of elements obtained by this function increases exponentially as the value of m becomes higher. Considering the pseudo-code that has been constructed, the structures that rise the number of instructions are two `while` loops and one `for` loop that are nested in a hierarchical and precise way.
Since we want to count the number of instructions performed by each of them, we proceed analysing the first `while` loop found in the pseudo-code. Basically, the guard is evaluated `m` times. This is due to the fact that, every time the array *result* is extended with all subsets having the same size, the value of the size under the analysis of the guard of the `while` loop, increases. Therefore, considering that the power set is made by a number of elements having length that can take values from 1 up to m and that the `while` loop is evaluated only once all subsets of the same size are created, the guard will be checked for a number of times equal to the number of possible value of size available in the power set, so `m` times.
Now we can move to the nested `for` loop and we can notice that at the end of the execution, the variable e present in the guard, is gradually assigned to all values in the search space along the execution. Basically, the guard of the `for` loop under analysis considers an array called `permu` a total number of `m` times since its execution is ruled by a `while` loop whose guard, as we said before, is evaluated `m` times. So the `for` loop evaluates a total number of `m` guards. At each run, the variable e in the current guard of the `for` loop is assigned to all subsets having same size present in the array `permu`. At the end of the loop, the array on which the `for` loop iterates is updated with all subsets having size increased by one. So, since this process continue up the failure of the guard of the first `while` loop, marking that all subsets of all size from 1 up to m have been constructed, the variable e in the `for` loop, will be assigned to 2^m values, so the entire power set. Then, considering the last `while` loop, nested in the previously analyzed `for` loop, we can see that, in the worst case, the guard is evaluated `m` times since the variable `i` has to consider all elements of the set 1.....m to built all subsets of size increased by one, each being different from the others. So, to conclude, the total number of instructions performed by the function `setsOfPossibleCombinations(m)` is equal to:

number of basic instructions $= m * 2^m * m = m^2\ 2^m$
number of basic instructions $= m^2\ 2^m$

Now, we can consider the second function of the algorithm `ExhaustiveSearchAlgorithm(n,m,S,U)`. Basically, in this case, the number of instructions is dictated by two `for` loops.
Considering the first one, the variable e is assigned 2^m times since it has to analyze all possible elements of the search space, that, as we have already discussed, counts a total of 2^m subsets in it.
The second `for` loop, in the worst case, evaluates the variable i a total number of `m` times since it has to calculate the $\sum_{i \in subset} S_i$ and the $\sum_{i \in subset} U_i$ per each subset in the power set. Therefore, considering that the search space is formed of elements of size from 1 up to m, when the variable e of the first `for` loop considers a set of size m the variable i of the second `for` loop (nested in the first one) is evaluated m times since it considers all values of the set under analysis. So the total number of instructions performed by the function`ExhaustiveSearchAlgorithm(n,m,S,U)` is equal to:

number of basic instructions = m * 2^m
number of basic instructions =$m\ 2^m$

To conclude, the time complexity of the algorithm under analysis can be evaluated summing the number of basic instructions performed by the two functions that we have just explained.

sum of basic instructions = $m^2\ 2^m + m\ 2^m$
sum of basic instructions = $m^2\ 2^m$

So the complexity of the algorithm is approximately $O(m^2\ 2^m)$. Therefore we can observe that the number of basic instructions grows exponentially as the value of m increases.

6 The algorithm-Branch And Bound

In this section, we analyze the possible improvement of the `Exhaustive Search` algorithm. Basically, one way that can be used to make the algorithm faster, even if the complexity remains exponential, is to consider only subsets where $\sum_{i \in subset} S_i$ does not exceed the threshold n. In the moment in which the program meets a subset that is not favorable to be taken into consideration, there is no need to calculate its relative, so the algorithm skips them and proceed into evaluating other subsets. Otherwise, when a subset is in a favorable condition, meaning that $\sum_{i \in subset} S_i < n$, the program explores its relatives which are subsets of higher length. The algorithm has been developed following the `Branch And Bound` methodology in which each internal node represents a possible subset of size lower then m.

The pseudo-code that has been constructed is formed of four functions, each performing a specific functionality:

- the function `OverPass(node,m)`is responsible of bypassing a node, which represents one possible combination, in the moment in which the $\sum_{i \in node} S_i > n$;
- the function `NextNode(node,m)` is responsible of building a node of size increased by one, so a possible subset of 1.....m, originated from the current node under analysis. So it allows the exploration of novel nodes of higher size;
- the function `ComputeSums(node,S,U)` is responsible to calculate the $\sum_{i \in node} S_i$ and the $\sum_{i \in node} U_i$ of the current node so of the current subset of 1.....m;
- the function `BranchAndBound(n,m,S,U)` is responsible of evaluating the validity of the subset of 1.....m under study. The control is performed analysing the $\sum_{i \in node} S_i$ and the $\sum_{i \in node} U_i$, where *node* is the current subset of 1.....m.

```
OverPass(node,m)
  if a subset from which no further combinations are available then
    node ← remove the last element of the current node {go back to the vertex}
    node ← increase the last value of node by 1 {go to the node on the right}
    return node
  else
    node ← add 1 to the last element of the current node {bypass the current node}
    return node
  end if

NextNode(node,m)
  if a subset from which no further combinations are available then
    node ← remove the last element of the current node {go back to the vertex}
    node ← increase the last value of node by 1 {go to right node}
    return node
  else
    node ← build node of size increased by 1 {explore lower level in the tree}
    return node
  end if

ComputeSums(node,S,U)
  capacity ← 0
  utility ← 0
  for all elements e in node do
    capacity ← ∑_{e∈node} S_e
    utility ← ∑_{e∈node} U_e
  end for
  return capacity,utility

BranchAndBound(n,m,S,U)
  finalQual ← 0
  output ← empty array
  for i ← 1 to m do
    node ← array containing i
    capacity ← array containing the corresponding element at position i in S
    utility ← array containing the corresponding element at position i in U
    while forever do
      if The current node is the last leaf of this section of the tree then
        stop while
      end if
      if capacity > n then
        node ← OverPass(node,m)
        newSums ← ComputeSums(node,S,U)
        capacity ← first element of newSums
        utility ← second element of newSums
      else
        if utility > finalQual then
          finalQual ← utility
          output ←the current node to output
        end if
        node ← NextNode(node,m)
        newSums ← ComputeSums(node,S,U)
```

capacity ← first element of newSums
utility ← second element of newSums
end if
end while
end for
if number of subsets = 1 **then**
if $S_{node} < n$ **then**
return m
else
return no subsets, respecting the condition, available
end if
return last element collected in *output*

7 The implementation-Branch And Bound

The following python function `OverPass(node,m)` is the implementation of the first function represented in the pseudo-code. It is entitled to bypass a node of the tree in the moment in which the value of the $\sum_{i \in node} S_i$ exceed the threshold n in this way, the algorithm avoids to perform useful computations.

```
def OverPass(node,m):
        if node[-1] == m-1:
                back = node[:-1]
                back[-1] += 1
                return back
        else:
                node[-1] += 1
                return node
```

The following python function `NextNode(node,m)` is the implementation of the second function provided in the pseudo-code. It is responsible to explore the internal nodes of the tree in the moment in which the $\sum_{i \in node} S_i$ is lower or equal then the threshold n.

```
def NextNode(node,m):
        if node[-1] == m-1:
                back = node[:-1]
                back[-1] += 1
                return back
        else:
                node = node + [node[-1] +1]
                return node
```

The following python function `ComputeSums(node,S,U)` id the implementation of the third function provided by the pseudo-code. It is responsible of calculating the $\sum_{i \in node} S_i$ and the $\sum_{i \in node} U_i$ of the current *node* under analysis.

```
def ComputeSums(node, S, U):
        capacity = 0
        utility = 0
        for e in node:
                capacity += S[e]
                utility += U[e]
        return capacity,utility
```

The following python function `BrancAndBound(n,m,S,U)` is the implementation of the fourth function provided by the pseudo-code. It is responsible of evaluating each node provided by the other functions of the algorithm, starting from the root node. As a node has a $\sum_{i \in node} S_i$ greater then the threshold n, it induce the algorithm to bypass it and its relatives. Otherwise, it explores them keeping on track of the max $\sum_{i \in node} U_i$.

```
def BranchAndBound(n,m,S,U):
    finalQual = 0
    output = []
    for x in range(m-1):
        node = [x]
        capacity = S[x]
        utility = U[x]
        while True:
            if node == [x,m-1] or node == [m-1]:
                break
            if capacity > n:
                node = OverPass(node,m)
                newSums = ComputeSums(node,S,U)
                capacity = newSums[0]
                utility = newSums[1]
            else:
                if utility > finalQual:
                    finalQual = utility
                    output += [node[:],]

                node = NextNode(node,m)
                newSums = ComputeSums(node,S,U)
                capacity = newSums[0]
                utility = newSums[1]

    if len(output)==0:    #special case when m=1
        if S[0] < n:
            return [m-1,]
        else:
            return None

    return output[-1]
```

8 Testing Routine

This section is dedicated to present a possible function that can be used to test the two algorithms `Exhaustive Search` and `Branch And Bound`. This function does not take any argument. As the function is executed, it takes one random value of m among 5,10,15,20,25. It builds two arrays S and U having both size being equal to m and the values inside are selected randomly from 1 to 10. Then the function tests both algorithms for the given value of m, S, U and n where the latter is evaluated for two values: $m * 7$ and $m * 3$.
The following function is the implementation of such test:

```
import random

def test():
    testNumber = [5,10,15,20,25]
    m = random.choice(testNumber)
```

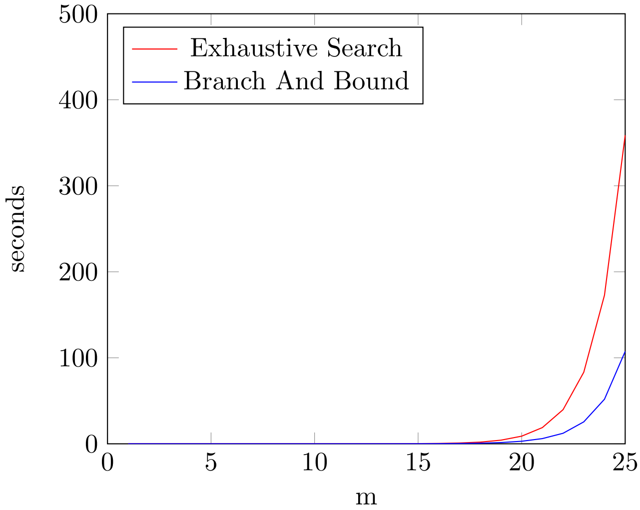
```
    S = [random.randint(1,10) for x in range(1,m+1)]
    U = [random.randint(1,10) for y in range(1,m+1)]
    print('the␣number␣of␣databases␣is:',m)
    print("The␣array␣S␣contains␣the␣following␣sizes:",S)
    print("The␣array␣U␣contains␣the␣following␣utilities:",U)
    for e in (3*m,7*m):
        n = e
        print('The␣total␣amount␣of␣mamory␣available␣is:',n)
        print('EXHAUSTIVE␣SEARCH')
        ES = ExhaustiveSearch(n,m,S,U)
        print(ES)
        print('BRANCH␣AND␣BOUND')
        BB = BranchAndBound(n,m,S,U)
        print(BB)
        print('Are␣they␣equal?', ES==BB)

    return ''
```

The values inside S and U are provided randomly importing the dedicated python module `import random`, that must be inserted before the function definition.
As we can noticed, the algorithms are evaluated for different values of n each: the first time with $n = m * 7$ and the second with $n = m * 3$. The test for the second value of n is performed because considering that the values of S are taken randomly in a range between 1 and 10, the total $\sum_i S_i$ will mostly result to be lower then n when $n = m * 7$. So the final solution of the algorithm will be the overall set 1.....m
In the last part of the test, the function evaluates if the result of the two algorithms is the same. If such happens, the function will return the Boolean value `TRUE`. Otherwise, the output will be `FALSE`.
Performing 10 tests for values of n equal to $m*7$ and $m*3$ for each value of m among $(5, 10, 15, 20, 25)$, a total number of 100 test has been done. Among these, 10 provided `FALSE` as result. This is due to the fact that, values inside S and U are randomly taken from 1 to 10. So it can happen that there are different elements of 1.....m having same same S or U. Therefore, at the end, we could find subsets of 1.....m being different but still having same $\sum_{i \in subset} S_i$ and same $\sum_{i \in subset} U_i$. The ratio of `FALSE` as result in the test routine increases as the value of m becomes bigger since the number of possible elements of 1.....m having different S or U associated to them increases.

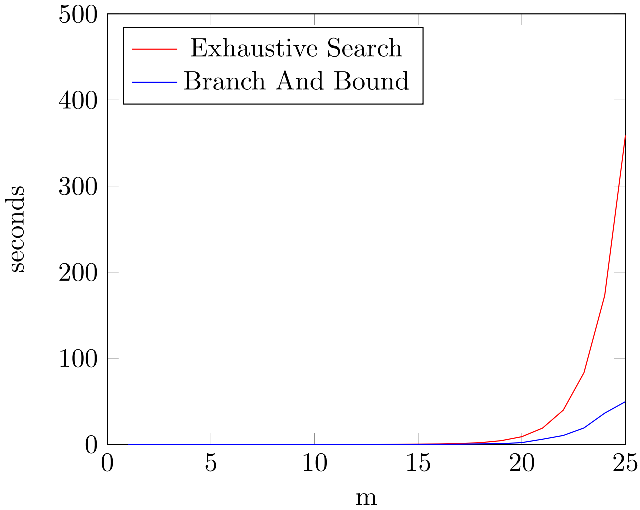
9 Performance *ExhaustiveSearch* and *BranchAndBound* algorithms

The following graphs represent the comparison of the performance. The latter has been evaluated considering how the time needed by the algorithms to be executed changes as the value of m increases. Time has been measured in seconds. So two different graphs have been created. The first one considers a value of n being equal to $m*7$. While in the second, the value of n is equal to $m*3$. In both graph we find the values of m from 1 up to 25 on the `x-axis`; while seconds are reported in the `y-axis` where the value considered go from 1 up to 500. The data cornering time, has been extrapolated by testing the algorithms with different values of m using the python package `cProfile`.

The following graph shows how the performance, expressed as number of seconds, changes as the value of m increases when the value of n is equal to $m * 7$. From the graph we can observe that as m becomes bigger and bigger, the time grows exponentially.



The following shows how the performance, expressed as number of seconds, changes as the value of m increases when the value of n is equal to $m * 3$. From the graph we can observe that as m becomes bigger and bigger, the time grows exponentially but in the case of the `Branch And Bound` algorithm, the amount of seconds needed to provide a result is lower for high values of m. This is mainly due to the fact the `Branch And Bound` algorithm has been developed to avoid not useful computations in the moment in which a the value of the $\sum_{i \in node} S_i$ exceed the threshold n. Therefore, when n is equal to $m * 3$, it lowers the value of the threshold and considering that there are more probable sets where the $\sum_{i \in set} S_i > n$, the algorithm will skip them, reducing the number of computations and so the amount of seconds of the overall execution.



We can notice from both graphs that, for both values of n, the `Exhaustive Search` algorithm has the same behaviour. This is mainly due to the fact that the algorithm follows a `Brute Force` structure, so, independently from the value of n, it has to evaluate all possible solutions rising the number of computations each time.