LABORATOY PAPER

PLOAE DARIUS-ADONIS-IONUȚ

CEN 1.2B

Faculty of Automatics, Computer Science and Electronics

-

May 2024

-

CLICK HERE FOR THE SOURCE CODE :

https://github.com/PLOAE-DARIUS-ADONIS-IONUT/ADHomework

1) PROBLEM STATEMENT

A fisherman is exploring a coastal region rich in lobsters, each with its own size (in centimeters) and value (in gold coins). The fisher- man's net has a limited capacity, expressed in the total number of centimeters it can hold. Given a detailed list of the sizes and values of the lobsters available in that region, your task is to develop a strategy for the fisherman to select lobsters in such a way as to maximize the total value of his catch while adhering to the net's capacity limit. You need to decide which lobsters to include in the net and which to leave behind so that the sum of the values of the selected lobsters is as high as possible, without the sum of their sizes exceeding the capacity of the net.

Imagine a scenario where a fisherman is given the opportunity to choose from a selection of lobsters, each with a specified size and value, to fill his net which has a maximum capacity. The fisherman's goal is to maximize the total value of his catch without exceeding the net's size limit. Here's an example:

- Lobster A: Size = 4 cm, Value = 20 gold coins
- Lobster B: Size = 3 cm, Value = 15 gold coins
- Lobster C: Size = 2 cm, Value = 10 gold coins
- Lobster D: Size = 5 cm, Value = 25 gold coins

Net capacity: 10 cm

The challenge is to select the combination of lobsters that maximizes the total value without exceeding a total size of 10 cm. One possible solution would involve choosing Lobsters A and C, giving us a total size of 6 cm (4 cm + 2 cm) and a total value of 30 gold coins (20 + 10). However, a better solution would be to choose Lobsters B, C, and D, which together have a total size of 10 cm (3 cm + 2 cm + 5 cm) and offer a higher total value of 50 gold coins (15 + 10 + 25). This combination exactly fills the net's capacity and maximizes the catch's value.

The task involves evaluating a detailed inventory of available lobsters, considering both their sizes and values, to determine the optimal subset of lobsters to include in the net. This requires a methodical approach to ensure that the sum of the selected lobsters' sizes does not exceed the net's capacity, while the sum of their values is maximized.

One possible approach to solve this problem involves employing combinatorial optimization techniques, such as the Knapsack problem algorithm, to determine the optimal selection of lobsters. Here, each lobster can either be included in the net or excluded, forming a binary decision-making process that evaluates all possible combinations.

The challenge of maximizing the value of the fisherman's catch within a limited net capacity can be effectively addressed through algorithmic optimization. By analyzing all possible combinations of available lobsters and their respective sizes and values, one can identify the optimal subset that maximizes the total value without exceeding the net's capacity. This combinatorial problem exemplifies practical applications of optimization techniques in resource management and decision-making processes in constrained environments.

Input:

– C - the maximum capacity of the net (a positive constant).

– n - the number of available lobsters.

– For each lobster i (i = 1, 2, . . . , n):

∼ name_i - the name of lobster i (a string of characters).

∼ size_i - the size of lobster i (a positive real number).

∼ value_i - the value of lobster i (a positive real number).

• Output:

– A subset of selected lobsters such that:

∼ The sum of the sizes of the selected lobsters is less than or equal to C.

∼ The sum of the values of the selected lobsters is maximum.

∼ The selected lobsters are enumerated.

2) ALGORITHMS

2.1 Knapsack algorithm

In order to get the maximum possible value that can fit in the net I used the knapsack dynamic programming algorithm. The time complexity is O(n * c) where n is the number of lobsters and c the maximum capacity.

Algorithm 1 (Knapsack)

```
Let a table $K[0..n][0..C]$
for $i = 0$ to $n$ do
  for $cap = 0$ to $C$ do
    if $i == 0$ or $cap == 0$ then
        $K[i][cap] = 0$
    else if $w[i] <= cap$ then
        $K[i][cap] = \max(v[i] + K[i-1][cap - w[i]], K[i-1][cap])$
    else
        $K[i][cap] = K[i-1][cap]$
    end if
  end for
end for
return $K[n][C]$
```

2.2 EXPLANATION OF THE KNAPSACK ALGORITHM

Let’s detail the steps of the algorithm:

• We define a table K where K[i][cap] represents the maximum value that can be obtained using the first i objects and having the capacity cap available in the knapsack.

• We initialize the table K with zero for base cases where either the number of objects is zero, or the knapsack capacity is zero.

• For each object i and each capacity cap:

– If the size of object i is less than or equal to the capacity cap, then we have two options:

∼ Include object i and add its value to the optimal value of the knapsack with remaining capacity cap − w[i].

∼ Do not include object i and take the optimal value of the knapsack without this object.

– We choose the option that offers the maximum value.

– If the size of object i is greater than the capacity cap, we cannot include the object, so we keep the optimal value without this object.

• The final result, i.e., the maximum value that can be obtained with capacity C using all the objects, is found in K[n][C]

2.3 ALGORITHM FOR ENUMERATING SELECTED ITEMS

To determine which objects have been selected to achieve the maximum value, we can traverse the table K in reverse order, starting from K[n][C]:

Algorithm 2 Algorithm for Enumerating Selected Items

```
1: Initialize the empty list selectedItems
2:   cap = C
3:     for i = n to 1 with step -1 do
4:         if K[i][cap] != K[i-1][cap] then
5:         Add object i to selectedItems
6:          cap = cap − w[i]
7:       end if
8:     end for
9: return selectedItems
```

2.4 Explanation of the Algorithm for Enumerating Selected Items

The algorithm for enumerating selected items utilizes the table K to identify the objects included in the optimal solution:

• We start from position K[n][C] and check if the current value differs from the one above it K[i − 1][cap].

• If the values are different, it means object i was included in the knapsack. We add the object to the list of selected items and decrease the available capacity cap by the size of object i − 1.

- We continue this process until we reach the first object.
- The resulting list, selectedItems, contains the objects that were included to achieve the maximum value.

The complexity of the enumeration algorithm is O(number of objects), so it does not affect the overall program complexity.

3) EXPERIMENTAL DATA

The program was ran on ten test files. Each test file was randomly generated using the "test-generator" program. The data had lobsters with names less than 100 charachter, with sizes and values in between 1 and 1000. The first test file has 1000 lobsters and 1000 maximum bag capacity. It then grows by 1000 for every test file. The parametres used for generating the files can be found in the params folder. The same test files were used for testing both the C and Python implementation. Both implementations produced the same results.

Test no. 1:

Input:1000

Value in the knapsack: 22813

C Time Elapsed: 0.019800 seconds

Python Time Elapsed

Test no.2 :

Input: 2000

Value in the knapsack: 46314

C Time Elapsed : 0.082938 seconds

Python Time Elapsed : 4.108994007110596 seconds

Test no.3 :

Input: 3000

Value in the knapsack: 76067

C Time Elapsed : 0.190251 seconds

Python Time Elapsed : 9.936467409133911 seconds

Test no.4 :

Input: 4000

Value in the knapsack: 102317

C Time Elapsed : 0.337445 seconds

Python Time Elapsed : 18.381574869155884 seconds

Test no.5 :

Input: 5000

Value in the knapsack: 132882

C Time Elapsed : 0.548055 seconds

Python Time Elapsed : 29.041252613067627 seconds

Test no.6 :

Input: 6000

Value in the knapsack: 160288

C Time Elapsed : 0.797755 seconds

Python Time Elapsed : 42.897181034088135 seconds

Test no.7 :

Input: 7000

Value in the knapsack: 185679

C Time Elapsed : 1.063027 seconds

Python Time Elapsed : 58.98277187347412 seconds

Test no.8 :

Input: 8000

Value in the knapsack: 215871

C Time Elapsed : 1.430579 seconds

Python Time Elapsed : 77.308030128479 seconds

Test no.9 :

Input: 9000

Value in the knapsack: 240475

C Time Elapsed : 1.775200 seconds

Python Time Elapsed : 97.88844871520996 seconds

Test no.10 :

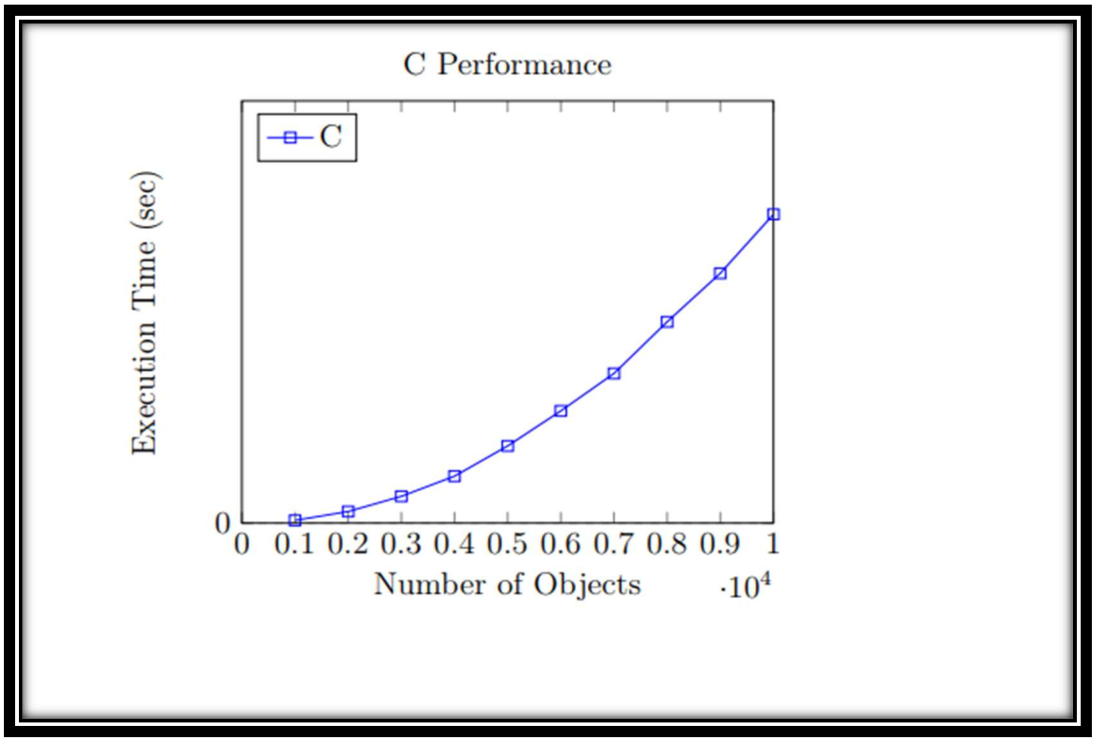
Input: 10000

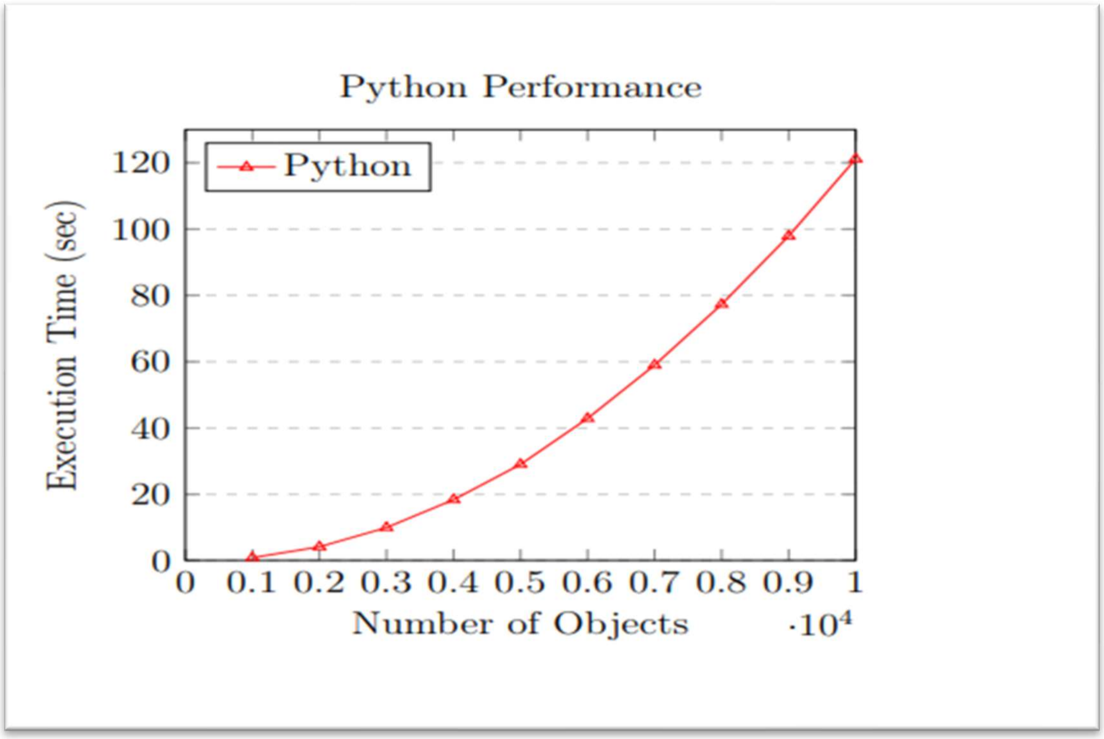
Value in the knapsack: 263073

C Time Elapsed : 2.194263 seconds

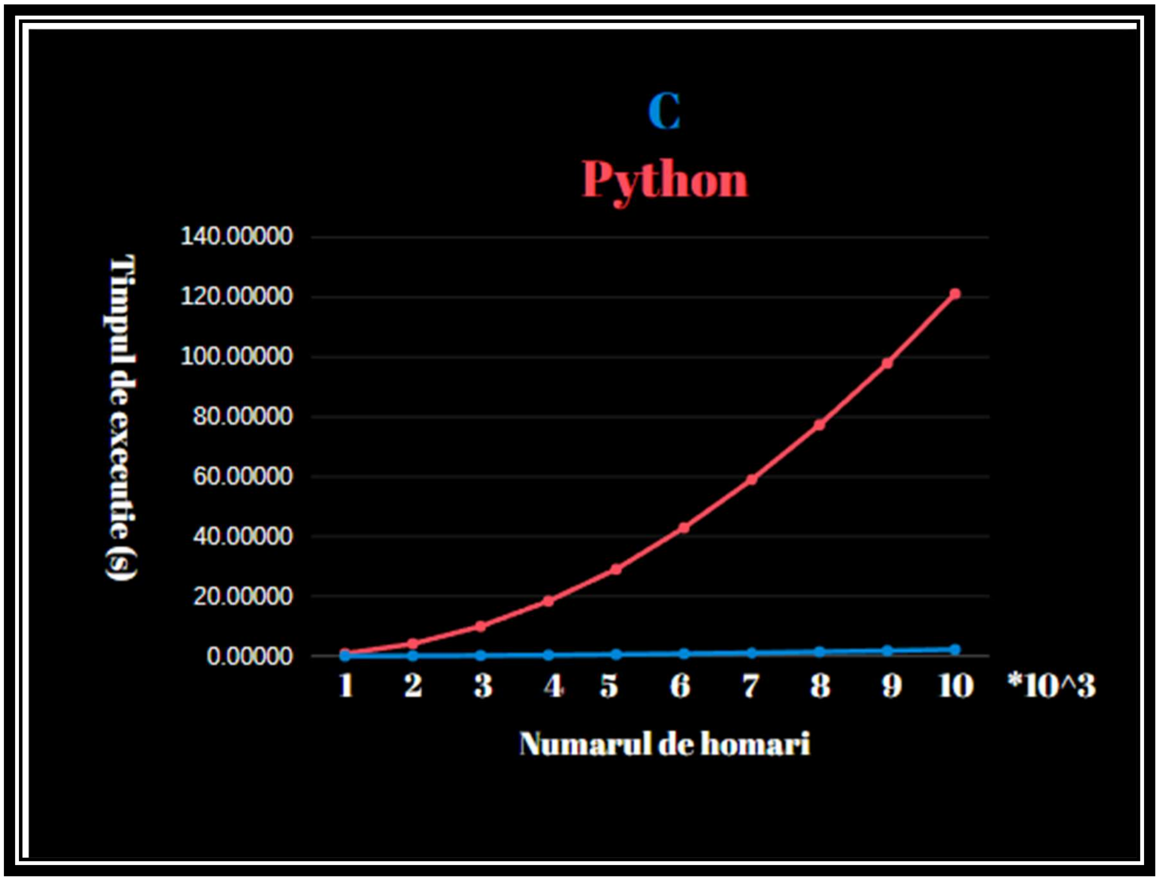
Python Time Elapsed : 121.14253902435303 seconds

Here is a graph mapping the C implementation execution time to the number of objects in each test case.





THE NEXT GRAPH COMPARES THE C AND PYTHON IMPLEMENTATIONS.



The execution time grows exponentially, because the time complexity is O(n*c), and here c is equal to n. The same is true of the Python implementation, but it is much slower (50x).

4) RESULTS AND CONCLUSIONS

My results show that the algorithm is right in every circumstance that I investigated, proving its dependability and resilience in generating reliable answers. Important insights were obtained from a thorough comparison of the algorithm's Python and C implementations. The C version, however

Although its stricter syntax and lower level nature made it more difficult and time-consuming to code, it executed far more quickly. C is the best option for applications that require high performance because of its speed advantage. On the other hand, rapid development and prototyping, when development time is more important than execution speed, are better suited for the Python implementation, which is distinguished by its simplicity and ease of writing, even though it executes more slowly.

Free and Open Source Software (FOSS) tools were found to be superior in terms of flexibility, cost-effectiveness, and community support, according to the software tool review. These tools are preferred for both development and deployment since they offer extensive functionality along with simpler integration and customization.

Using a functional programming language like Haskell to build the method could be a potential next step for this research. This would offer a chance to contrast Python's object-oriented methodology with C's imperative structure and the functional paradigm, maybe revealing fresh perspectives on the effectiveness and expressiveness of various programming paradigms.

To sum up, this paper has proven the accuracy of the method and offered a thorough analysis of several programming languages and development tools. The knowledge acquired highlights the benefits of FOSS tools and the trade-offs between Python and C coding simplicity and speed of execution.