# ***Laboratoy Paper***


## COPCĂ-MARE BIANCA-GIORGIANA

### CEN 1.1B

FACULTY OF AUTOMATICS, COMPUTER SCIENCE AND ELECTRONICS


**CLICK FOR THE SOURCE CODE :**[https://github.com/Copca-Mare-Bianca-Giorgiana/AD_project](https://github.com/Copca-Mare-Bianca-Giorgiana/AD_project)

# 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.

2 Pseudocode Algorithms 2.1 C implementation of LOBSTERS Problem Function valori_maxime_homari(homari_valoare, capacitate_plasa, n, selected_sizes, selected_values): // Allocate memory for dynamic programming matrices Initialize matrice_valori_maxime as a (n + 1) x

(capacitate_plasa + 1) matrix of zeros Initialize keep as a (n + 1) x (capacitate_plasa + 1) matrix of zeros // Dynamic programming to find maximum value For i from 1 to n do size = homari_valoare[i - 1][0] value = homari_valoare[i - 1][1] For j from 0 to capacitate_plasa do If j >= size and matrice_valori_maxime[i - 1][j - size] + value > matrice_valori_maxime[i - 1][j] do matrice_valori_maxime[i][j] = matrice_valori_maxime[i - 1][j - size] + value keep[i][j] = 1 Else matrice_valori_maxime[i][j] = matrice_valori_maxime[i - 1][j] result_matrix = matrice_valori_maxime[n][capacitate_plasa] // Track the items to be included in the solution k = capacitate_plasa sum_sizes = 0 sum_values = 0 index = 0 For i from n to 1 do If keep[i][k] do selected_sizes[index] = homari_valoare[i - 1][0] selected_values[index] = homari_valoare[i - 1][1] sum_sizes += homari_valoare[i - 1][0] sum_values += homari_valoare[i - 1][1] k -= homari_valoare[i - 1][0] index += 1 //Sorting sizes and values Sort selected_sizes in descending order Sort selected_values in descending order Return result_matrix End Function

## 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.

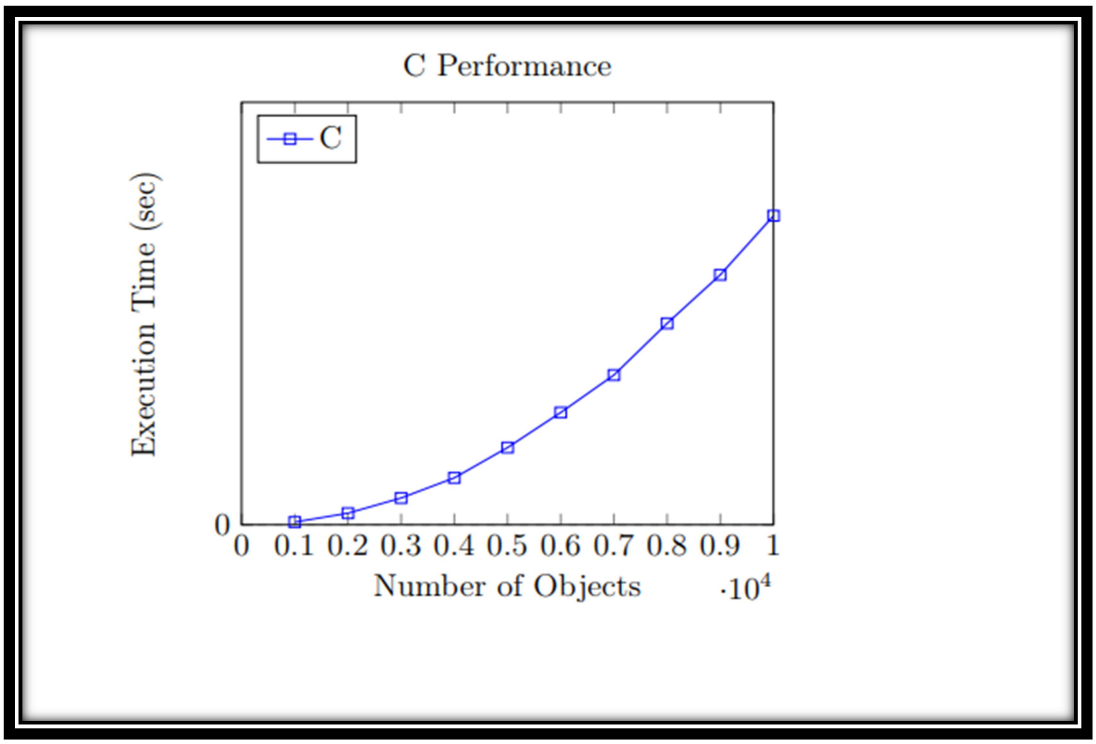
• The final result, i.e., the maximum value that can be obtained with capacity C using all the objects, is found in K[n][C]

.

# 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.

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



# 4) Results and conclusions

## Description of a C Program for Solving the Knapsack Problem

### Overview

This C program solves the 0/1 Knapsack problem using dynamic programming. It reads input from a file, processes the data to find the optimal solution, and outputs the results, including the items selected and the computation time. The program is structured to ensure clarity, efficiency, and accuracy in solving the problem.

### Program Structure

1. **Include Necessary Headers**:
    - `stdio.h`, `stdlib.h`, `stdbool.h`, `string.h`, and `time.h` for standard functions and data types.
    - A custom header `lobster.h` which defines the `lobster` struct used in the program.
2. **Main Function**:
    - Handles file operations for reading input and writing output.
    - Initializes variables for tracking the size of the knapsack, number of lobsters, and timing.
3. **Reading Input**:
    - Opens input and output files.

    - Reads the knapsack's capacity and the list of lobsters (each with a name, size, and value).
    - Allocates and reallocates memory dynamically to store lobsters that can fit into the knapsack.
4. **Dynamic Programming Function**:
    - Implements the dynamic programming algorithm to solve the knapsack problem.
    - Uses a matrix to store intermediate results and compute the maximum value that can be achieved within the given capacity.
    - Extracts the list of selected items from the matrix.
5. **Timing and Output**:
    - Measures the time taken to execute the algorithm using `clock()`.
    - Writes the selected items and the computation time to the output file.
6. **Cleanup**:
    - Frees allocated memory and closes files to avoid memory leaks.

## Timing

The program measures the execution time of the algorithm to provide insight into its performance. This is achieved using the `clock()` function from `time.h`, which records the start and end times of the algorithm's execution.

- **Start Time**: Recorded before the algorithm begins.
- **End Time**: Recorded after the algorithm completes.
- **Elapsed Time**: Calculated as the difference between the end and start times, then converted to seconds.

This timing information helps in evaluating the efficiency of the algorithm.

**Accuracy**

Accuracy is ensured through several steps:

1. **Correct Data Handling**:
    - The program reads and processes input data accurately, ensuring that only valid lobsters (those that can fit into the knapsack) are considered.
    - It uses robust file handling and error checking to avoid common issues like file not found or read errors.
2. **Dynamic Programming Matrix**:
    - The matrix is correctly initialized and filled based on the dynamic programming approach, ensuring that each subproblem is solved optimally.
3. **Result Extraction**:
    - The program accurately extracts the list of selected items by tracing back through the matrix, ensuring that the solution is correct and optimal.
4. **Memory Management**:
    - Proper allocation, reallocation, and freeing of memory ensure that the program runs without memory leaks or overflows, contributing to accurate and reliable results.