# Homework 1

**(100 points)**

**Assigned:** Monday, January 12, 2026
**Due:** Thursday, January 22, 2026

## Objective

The purpose of this assignment is to practice designing and implementing a brute-force algorithm to solve a variant of the classic knapsack problem, and to analyze both theoretical time complexity and actual running time.

## Liao-Knapsack problem

The Liao-Knapsack Problem (two-dimensional, 0-1, full-Knapsack) is defined as follows:

Given a knapsack of weight capacity $W$ and size capacity $S$, and $n$ items with weights $w_1$, $w_2$, ..., $w_n$, sizes $s_1$, $s_2$, ..., $s_3$, and values $v_1$, $v_2$, ..., $v_n$, where all $W$, $S$, $w_i$, $s_i$, and $v_i$ are positive real numbers, find a *full* packing of the knapsack. That is, choose a subset of the $n$ items such that the total weight and total size of the chosen items are exactly $W$ and $S$, respectively, and the total value of the selected items is maximized. Only one copy of each item is available, and items cannot be divided (i.e., you must either select the entire item or not select it at all).

## Part 1. Algorithm (20 pts)

Design and write an exhaustive search (brute-force) algorithm to solve the Liao–Knapsack Problem described above. Recall that an algorithm is a concise, step-by-step, high-level description of how to solve a problem; it is not a repetition of your programming code. Your algorithm must follow a formal algorithm format. You may describe your algorithm either in pseudocode (see the example on textbook page 295, the MFKnapsack algorithm) or in plain English (see the example on textbook page 455, the greedy algorithm for the continuous knapsack problem).

In addition, analyze the algorithm's time complexity in Big-O notation.

## Part 2. Implementation (80 pts)

Implement the algorithm you designed above in `Java`. Specifically, define an `Item` class containing *weight*, *size*, and *value*, and a `FullKnapsack` class to find an optimal packing, if one exists. To generate all possible packings, you may find the provided code[1] for generating all subsets helpful. Test your program on various positive real numbers (not just integers) to ensure correctness. The program should either output the actual packing or report that no valid packing exists.

[1] Download the code (`FindSubsets.java`) from Blackboard

Note: Submitting a program that solves a different type of knapsack problem or does not use a brute-force approach will receive zero points.

## Part 3. Performance Evaluation (Extra Credit: 25 pts)

You must have a valid brute-force implementation from Part 2 in order to earn extra credit. Test your program using either manually generated or randomly generated inputs with 15 to 30 items (or until the program takes an impractically long time to finish). Record the actual running time[2] of your program, and plot the running time as a function of the number of input items. Document your system configuration, including CPU specifications, RAM, disk, and operating system.

## Sample Program Input and Output

### Input

Your program should read a text file (`input.txt`[3]), which resides in the same working directory as the program, as the input. The first row contains the knapsack's weight and size capacities. Each subsequent row contains the information for one item: the item's name, weight, size, and value. All values are separated by spaces.

For example, the following is a sample input file with 4 items:

```
15 20
A 5 10 2.5
B 10 10 10
C 2 3 1
D 3 7 2
```

### Output

The following is an example of program output:

```
Enter the input file name: input.txt
Reading input.txt ....
Knapsack weight: 15
Knapsack size: 20
Number of items: 4
Item A (weight, size, value): 5 10 2.5
Item B (weight, size, value): 10 10 10
Item C (weight, size, value): 2 3 1
Item D (weight, size, value): 3 7 2
Finding optimal packing ....
Found a packing!
Total weight: 15
Total size: 20
Total value: [value]
```

[2] E.g., using `System.currentTimeMillis()`
[3] Download from Blackboard

```
Packing:  Item A, Item B, ..., etc.
Total running time: [second] seconds

or

Finding optimal packing ....
No packing is found!
Total running time: [second] seconds
```

## Academic Integrity

Please review the academic integrity policy outlined in the course syllabus. All homework assignments are individual work. Obtaining solutions or code from any other person, the Internet, or other sources—including generative artificial intelligence (AI)—constitutes a violation of the academic integrity policy and may result in serious consequences. AI may be used for appropriate purposes, such as a learning aid or for English grammar checks; however, you may not use AI to directly generate solutions to homework assignments. Any use of AI must be properly cited, including the prompts used and the corresponding outputs.

## What to Submit

Parts 1 and 3: Submit your algorithm, analysis, plots, and report as a single PDF file.

Part 2: Submit your well-documented and properly formatted (indented) `Java` source files (i.e., `Item.java` and `FullKnapsack.java`). Do not submit the entire Eclipse project or package.

Part 3 (Data): Also submit the input files used for your experiments.

Submit all materials on Blackboard before the due date.

Remember: Each homework assignment is non-trivial. Begin early to ensure sufficient time for completion.