Department of Computer Engineering and Science,
Florida Institute of Technology, Melbourne, Florida

0-1 Knapsack Problem

By

Sheetal Ghodake (903971055)

Supervisor

Dr. William Shoaff

CSE5211 Analysis of Algorithms

Fall 2021

TABLE OF CONTENTS

1. Problem Statement

The knapsack problem is a combinatorial optimization problem, which means that given a limited collection of items, the method will identify the most optimum one. Knapsack's problem is basically organizing items in an efficient way and picking the most optimal items to fulfil the required constraint. Assume you have a collection of objects with a weight and a value, and your task is to select items that are less than or equal to the weight limit while still retaining the maximum value. Issue frequently arises in resource allocation, as decision makers must pick among a collection of non-divisible projects or activities while working under a strict budget or time restriction and when you have to pack your bags with the most valuable items within a certain weight limit. The Knapsack problem can be solved using multiple approaches. The first simple solution that comes to my mind is Brute force approach. However, there are better approaches like Greedy and Dynamic Programming. Here, I will be using the dynamic programming approach.

2. First Solution

Using recursive Brute force approach is the most obvious answer to this problem. This is a brute-force solution because it calculates the total weight and value of all potential subsets, then chooses the one with the highest value while staying under the weight limit.

"Time complexity: *O(2^n) due to the number of calls with overlapping sub-calls.*

Auxiliary space: *O(1), no additional storage is needed."*[6]

While this is a viable approach, it is not ideal due to the exponential time complexity. Therefore, I will be using Dynamic programming approach to solve this problem.

3. Better Solution (Dynamic Programming)

Formal Definition

"*There is a knapsack of capacity c > 0 and N items.*

Each item has value vi > 0 and weight wi > 0.

Find the selection of items (δi = 1 if selected, 0 if not) that fit, ∑i=1N δiwi ≤ c, and the total value, ∑i=1N δivi, is maximized." [5]

4. Applications

The knapsack problem used in various fields like

- Computer science
- Combinatorics
- Applied mathematics
- Complex theories

Also, the knapsack problem is used in real-world decision-making processes like

- Determining the most efficient way to cut raw materials
- Investing/Portfolio management
- Assets for asset-backed securitization selection
- Other cryptosystems in a backpack

5. Algorithm (Pseudocode)

The following is the pseudocode for the 0-1 Knapsack Problem using dynamic programming. [1]

```
// Values (stored in array v)
// Weights (stored in array w)
// Number of distinct items (n)
// Knapsack capacity (W)

     for j from 0 to W do:
          m[0, j] := 0

      for i from 1 to n do:
          for j from 0 to W do:
              if w[i] > j then:
                  m[i, j] := m[i-1, j]
              else:
                  m[i, j] := max(m[i-1, j], m[i-1, j-w[i]] + v[i])
```

6. Algorithm Explanation

1. First and foremost, all weights and expenses (value) must be positive.
2. The most important criterion here is that the total of the chosen weights should be smaller than W. (total desired weight).
3. The time it takes to solve the 0-1 Knapsack problem is pseudo-polynomial.
4. Recursively perform the following three operations until the best optimum items are selected:
   a) m[0,w] = 0
   b) m[i,w] = m[i-1,w], if w_i > w (if the new chosen item is more than the weight limit)
   c) m[i, j] := max(m[i-1, j], m[i-1, j-w[i]] + v[i]), if w_i <= w

5. Calculate m[n,W] to obtain the final solution.
6. For increased efficiency, we utilize a table to store this from the bottom up.

7. Analysis:

Time Complexity: O(N*W).

where 'N' is the number of weight element(Input) and 'W' is total weight of knapsack.

As for every weight element we traverse through all weight capacities 1<=w<=W.

Auxiliary Space: O(N*W).

The use of 2-D array of size 'N*W'.

8. Statistics

I used JProfiler tool to analyze the code and calculate the time consumed and memory usage.

8.1 Comparison Tables

Time consumed

Input	Knapsack Dynamic Programming
10	359 µs
10^2	428 µs
10^3	1241 µs
10^4	7763 µs
10^5	93361 µs
10^6	543,000 µs

Table 1: Measuring the time for different input sizes for knapsack problem

Memory Usage

Input	Knapsack Dynamic Programming
10	7.17 MB
10^2	7.17 MB
10^3	7.17 MB
10^4	11.23 MB
10^5	54.61 MB
10^6	560 MB

Table 2: Measuring memory usage for different input sizes for knapsack problem

8.2 JProfiler Screenshots

Input size:10

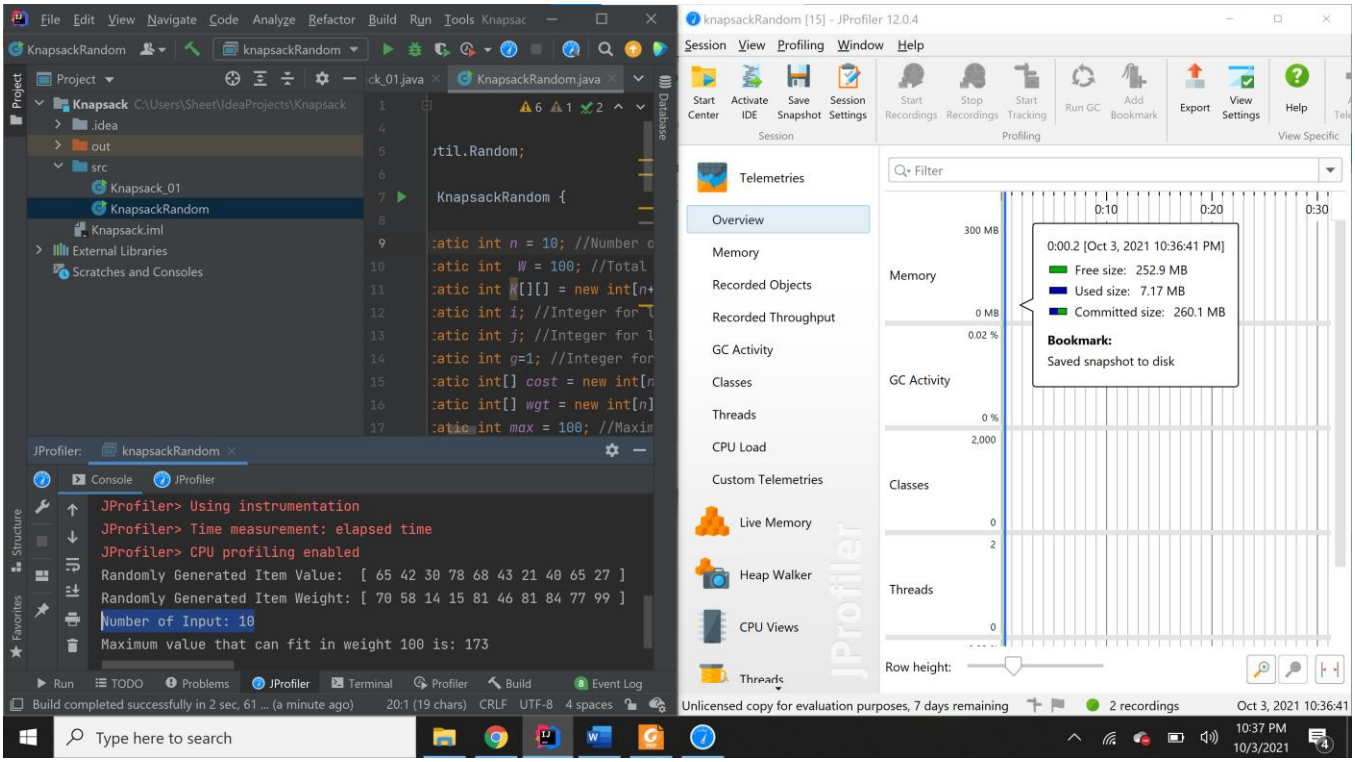


Figure 1: Memory used for input size 10[1]

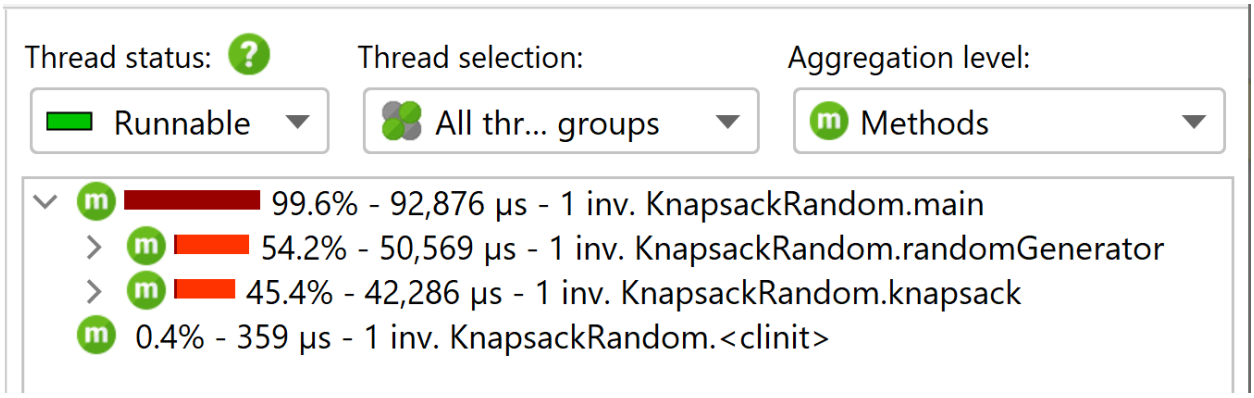


Figure 2: Time Consumed for input size 10[1]

Input Size: 10^2

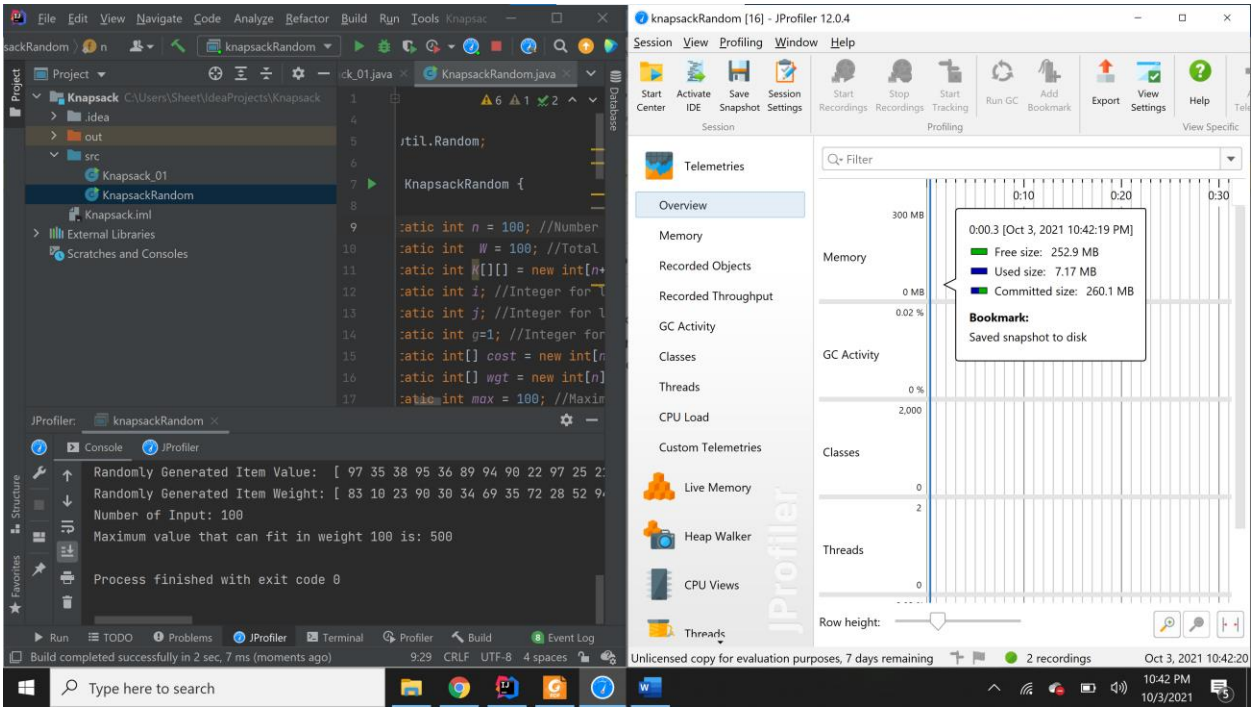


Figure 3: Memory used for input size 10^2

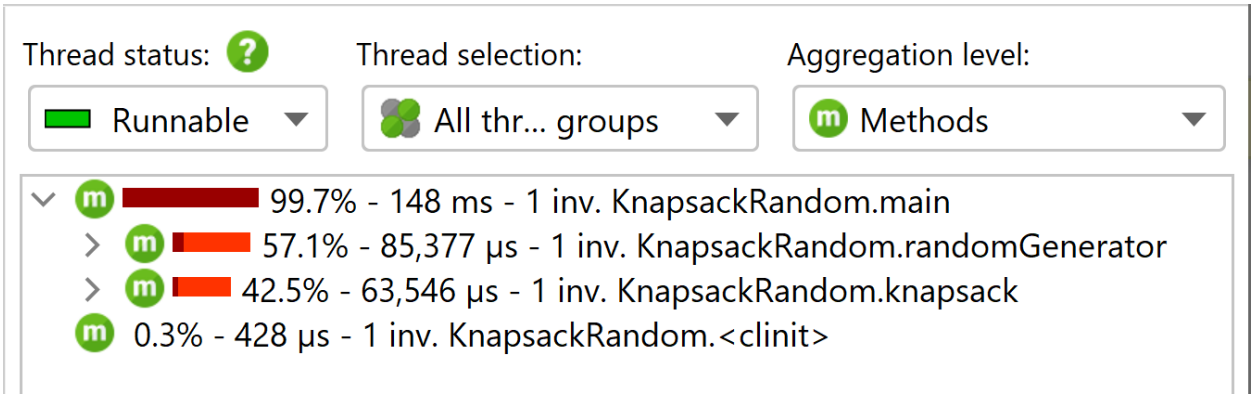


Figure 4: Time Consumed for input size 10^2

Input Size: 10^3

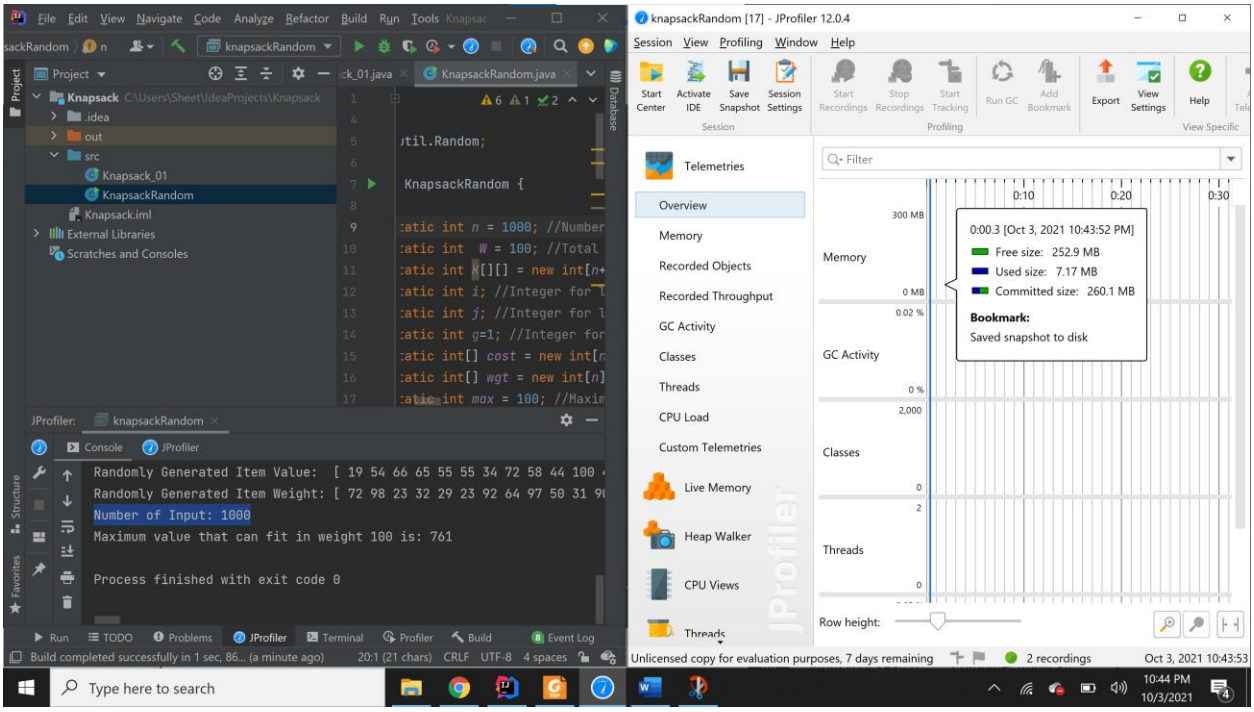


Figure 5: Memory used for input size 10^3

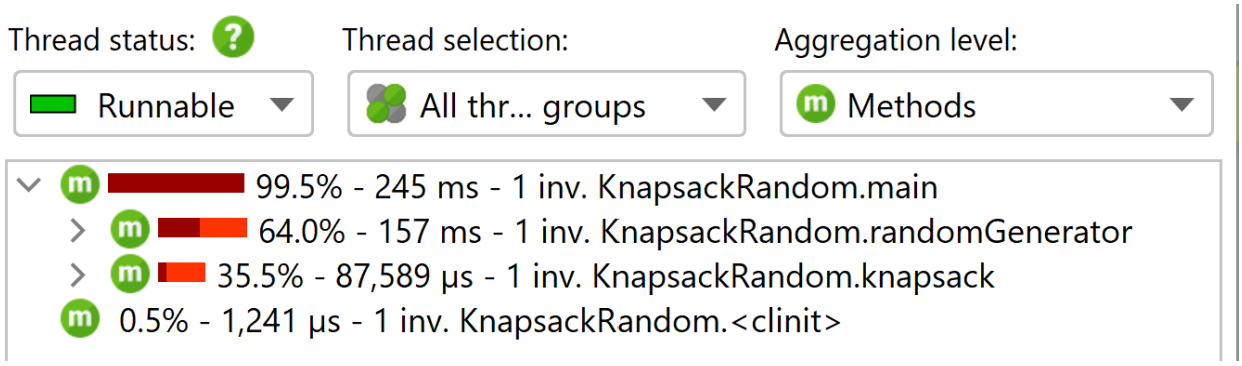


Figure 6: Time Consumed for input size 10^3

Input Size: 10^4

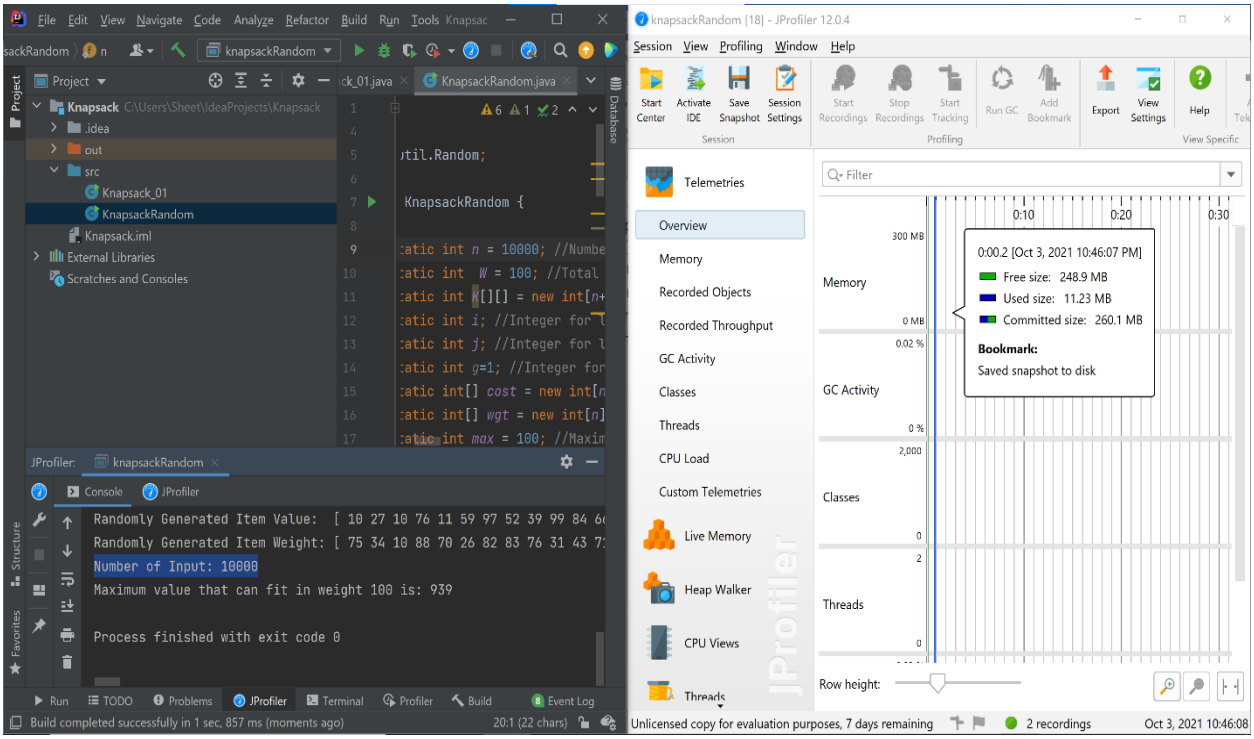


Figure 7: Memory used for input size 10^4

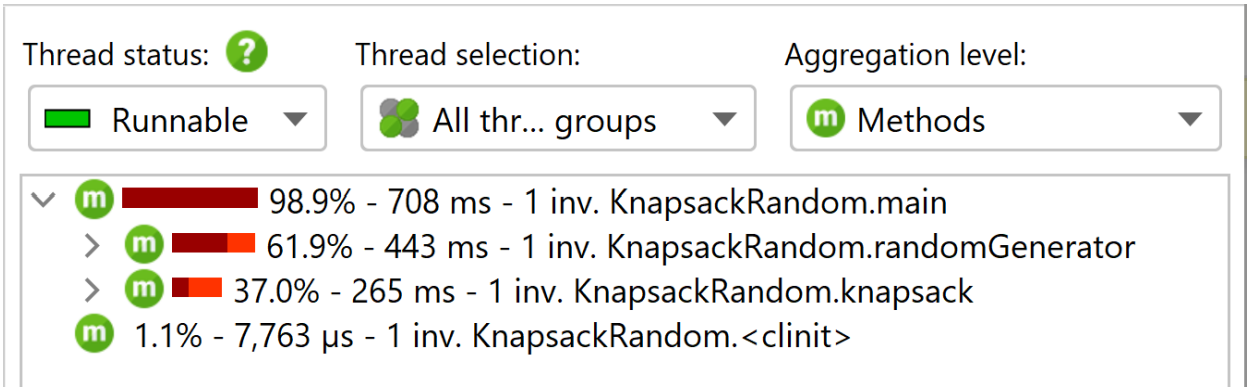


Figure 8: Time Consumed for input size 10^4

Input Size: 10^5

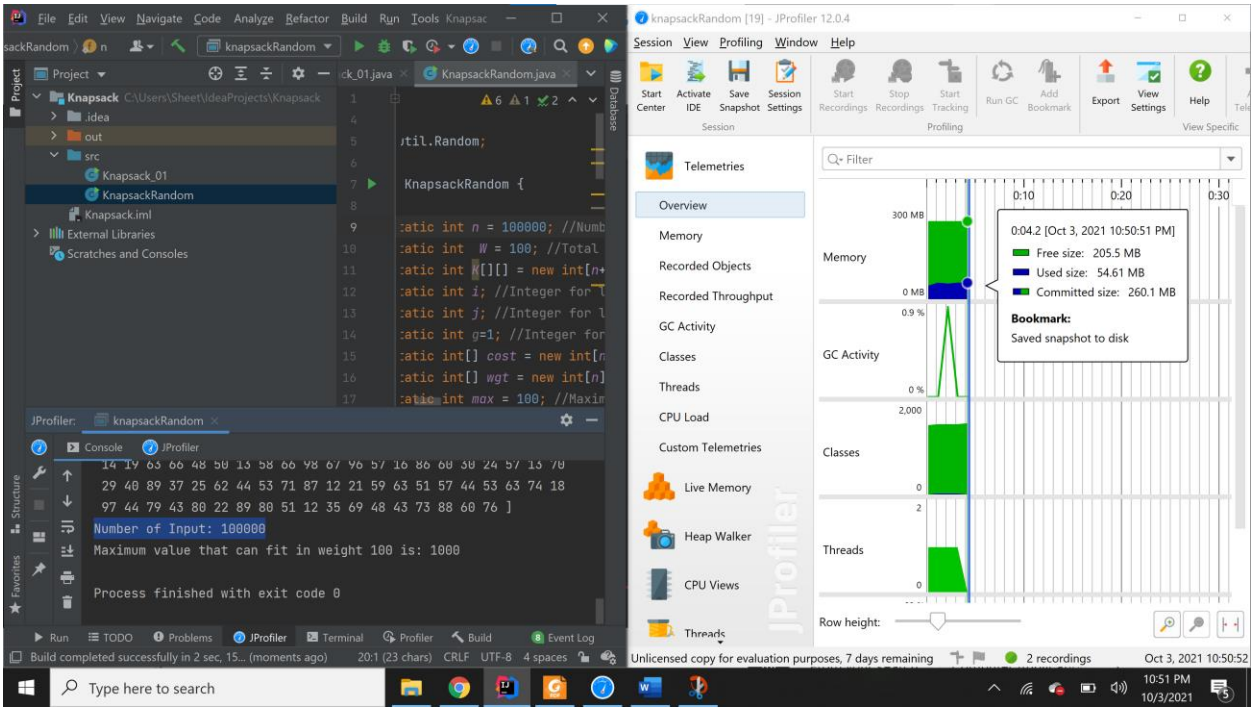


Figure 9: Memory used for input size 10^5

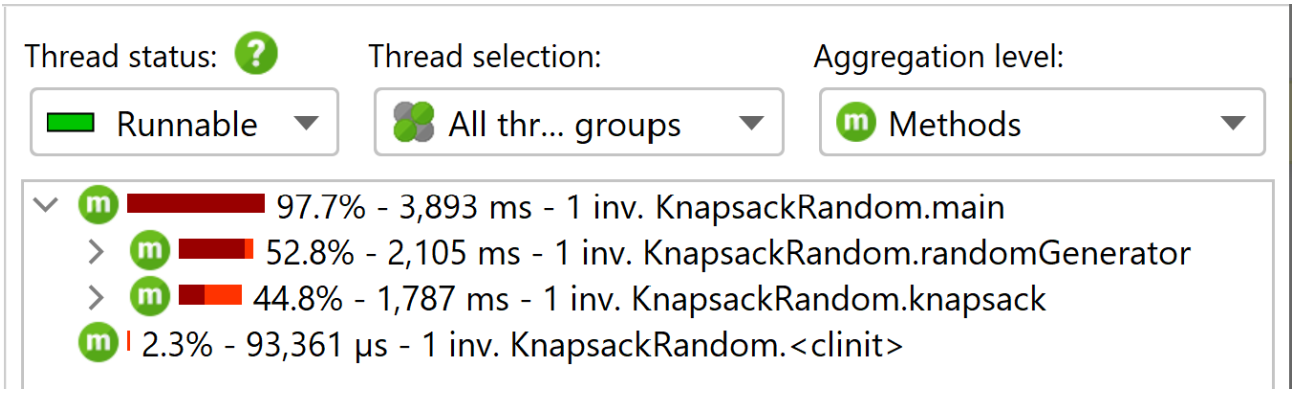


Figure 10: Time Consumed for input size 10^5

Input Size: 10^6

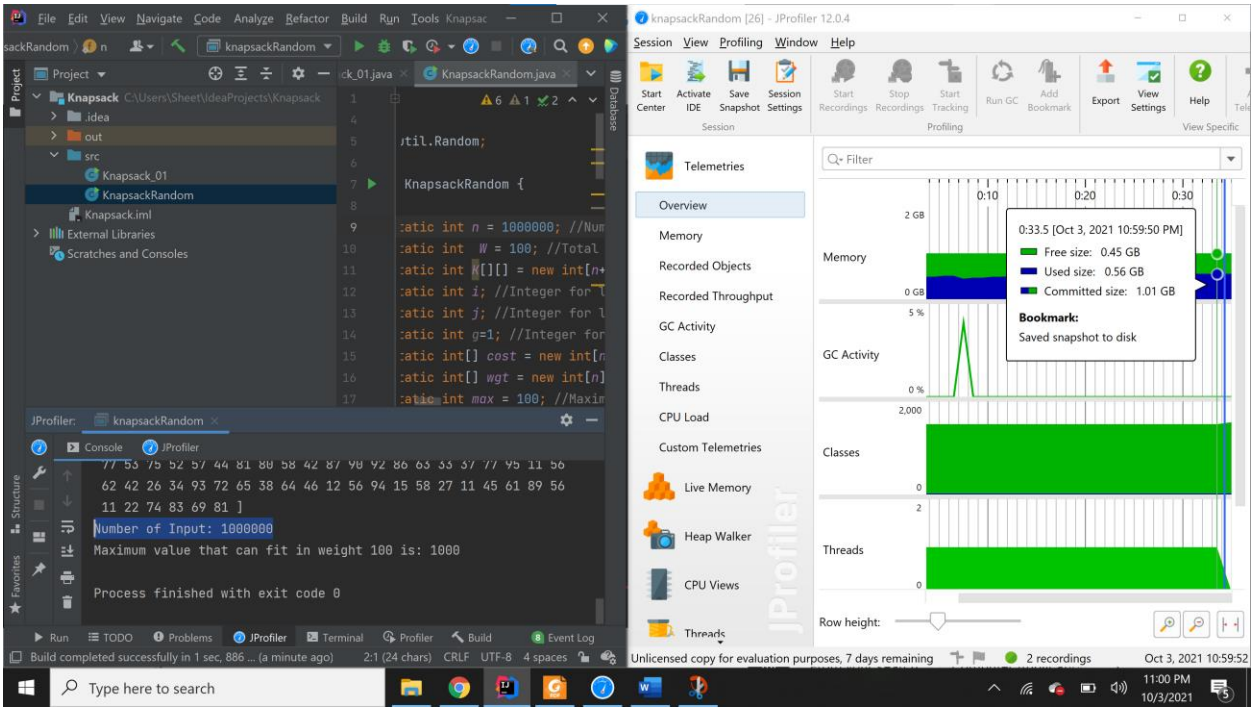


Figure 11: Memory used for input size 10^6

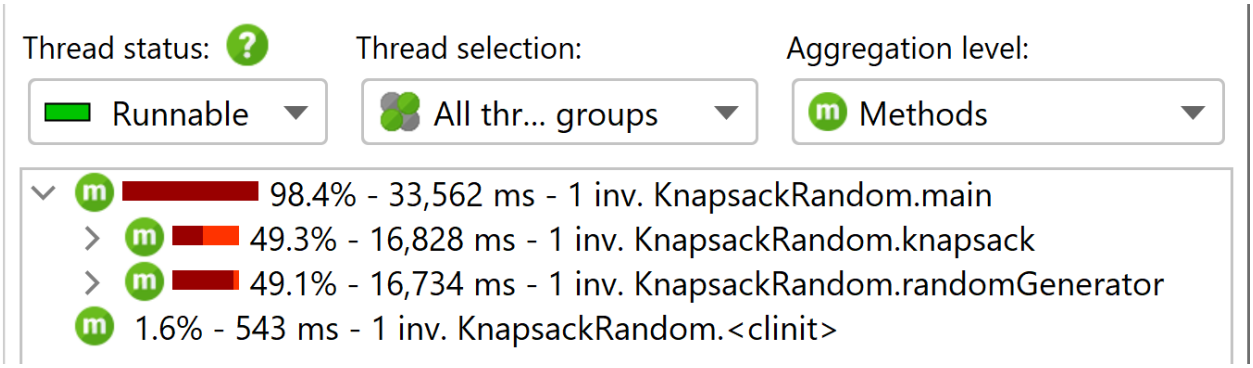
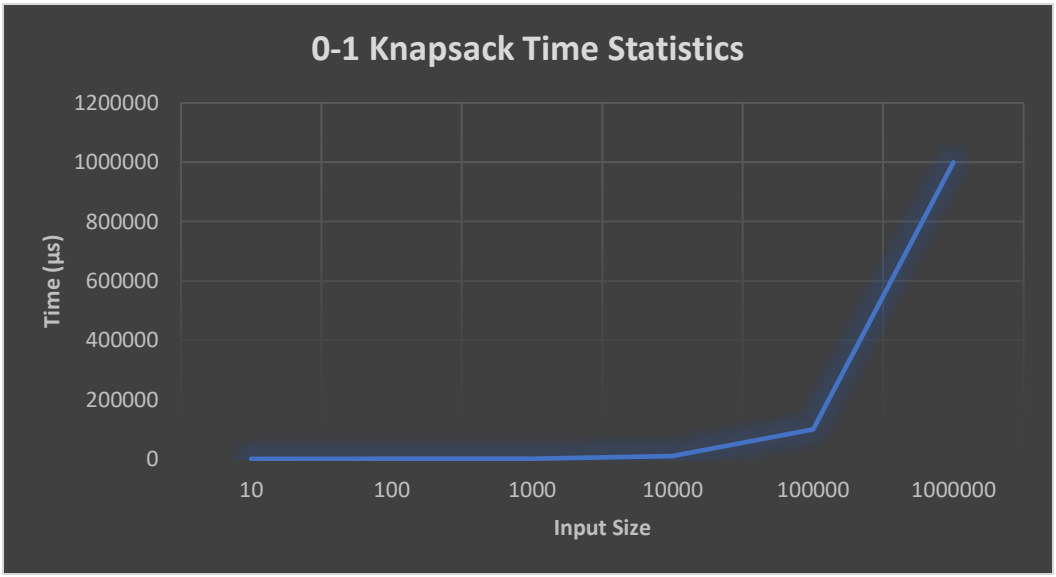
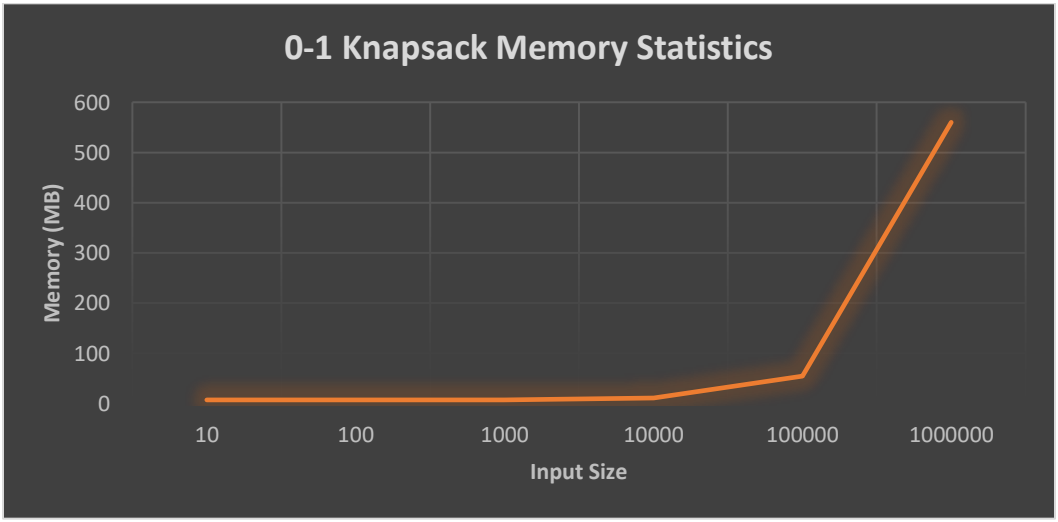


Figure 12: Time Consumed for input size 10^6

8.3 Comparison Graphs



As the amount of input increases, so does the amount of time spent on it. This is very close to linear time complexity, which is why inputs bigger than 10^5 require so much more time.



The memory usage is also directly related to the input size. However, for inputs less than 1000, the memory usage is pretty much the same. It only changes drastically when we go for inputs more than 10^4.

9. Conclusion

The knapsack problem arises in real-world decision-making in a variety of domains, such as determining the least wasteful way to chop raw materials. This problem has been proven to be an NP-complete problem, meaning that finding an optimal solution is difficult. This project describes the problem, formulas, its solution, the method, and their applications. I looked at the simple approach (Brute Force) but it is not an effective solution as I mentioned earlier. Therefore, I analyzed and implemented the dynamic programming solution to solve this problem. Using JProfiler, I also analyzed the time and memory it utilizes and showed them using graphs for input size versus time/memory consumption. The dynamic programming algorithm achieves the optimal solution. The dynamic programming algorithm is the best choice both in running time and accuracy of the solution. The time and space complexity of Dynamic Programming is also more efficient than the others.

10. References

[1] CSCE 310J Data Structures & Algorithms. (n.d.). Retrieved October 5, 2021, from http://cse.unl.edu/~goddard/Courses/CSCE310J/Lectures/Lecture8-DynamicProgramming.pdf.

[2] Saini, K. (n.d.). Knapsack problem in analysis and design of algorithms. C# Corner. Retrieved October 5, 2021, from https://www.c-sharpcorner.com/article/knapsack-problem-in-analysis-and-design-of-algorithms/.

[3] Wikimedia Foundation. (2021, October 4). Knapsack problem. Wikipedia. Retrieved October 5, 2021, from https://en.wikipedia.org/wiki/Knapsack_problem

[4] 0-1 Knapsack problem: DP-10. GeeksforGeeks. (2021, July 19). Retrieved October 5, 2021, from https://www.geeksforgeeks.org/0-1-knapsack-problem-dp-10/.

[5] Knapsack problem. (n.d.). Retrieved October 5, 2021, from https://xlinux.nist.gov/dads/HTML/knapsackProblem.html.

[6] Demystifying the 0-1 Knapsack problem: Top solutions explained. Educative. (n.d.). Retrieved October 5, 2021, from https://www.educative.io/blog/0-1-knapsack-problem-dynamic-solution#recursive.

11. Appendix

I have attached two source codes in zip file.

1) 0-1 Knapsack with User-Input (*Knapsack_01_User_Input.java*)

This is code for user input where the user can enter inputs manually like Number of items, values, and weights of each item and the total capacity of the Knapsack. Therefore, according to user inputs, code will get executed and give suitable output.

2) 0-1 Knapsack with Randomly generated inputs (*KnapsackRandom.java*)

This is code where inputs are randomly generated to analysis time complexity and memory usage. Input sizes are customizable as variables and for this report, we have kept the input size to be 10^1, 10^2, 10^3, 10^4, 10^5, and 10^6.