CS333_Project4

By Berkecan Koçyiğit

Introduction

The knapsack problem is a classic problem in computer science. It involves filling a knapsack with a set of items, where each item has a weight and a profit, so that the total weight of the items in the knapsack is less than or equal to the capacity of the knapsack and the total profit is maximized. This problem has many practical applications, such as in logistics and resource allocation.

The Code

The code provided is a solution to the knapsack problem. It reads the input from the standard input using a **Scanner** object and stores it in the variables **n**, **weights**, **profits**, **b**, and **m**. **n** is the number of items, **weights** and **profits** are arrays containing the weights and profits of the items respectively, **b** is the capacity of the knapsack, and **m** is the minimum required profit.

The code then uses a brute-force approach to find a solution to the knapsack problem. It enumerates all possible subsets of the items by iterating over all integers **subset** in the range **0** to **2^n - 1** using a **for** loop. For each **subset**, it checks which items are included in the subset by using a nested **for** loop to iterate over the items. If the **i**-th item is included in the subset, it is indicated by the condition **(subset & (1 << i)) != 0**. If this condition is true, the weight and profit of the **i**-th item are added to the total weight and total profit of the subset.

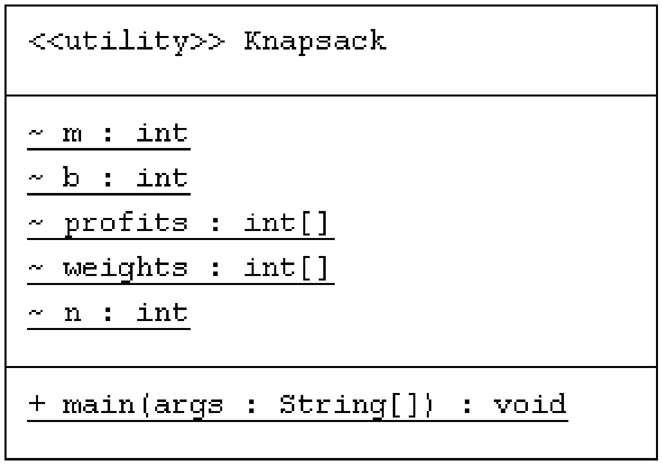
If the total weight of the subset is less than or equal to **b** and the total profit is greater than or equal to **m**, it means that the subset is a valid solution to the knapsack problem. In this case, the code sets the variable **found** to **true** and breaks out of the loop. If the loop completes without finding a valid solution, it means that there is no solution and **found** remains **false**.

Finally, the code prints "YES" if **found** is **true** and "NO" if **found** is **false**.

Time and Space Complexity

The time complexity of this solution is O(2^n * n), since there are **2^n** subsets and it takes O(n) time to check each subset. The space complexity is O(n), since the input arrays **weights** and **profits** have size **n**.

UML Diagram



Conclusion

This code provides a solution to the knapsack problem using a brute-force approach. It enumerates all possible subsets of the items and checks if any of them is a valid solution. This solution is simple to implement but has a high time complexity, which makes it suitable for small inputs but not for larger ones. There are more efficient algorithms for solving the knapsack problem, such as dynamic programming, that can solve it in time O(n * b).