MARMARA UNIVERSITY
FACULTY OF ENGINEERING
COMPUTER ENGINEERING

CSE 246
Analysis of Algorithms

Homework 3
Multiple Knapsack Problem

Hale ŞAHİN -150116841

1. Problem Definition

Knapsack problem aims to have items into knapsack where the total value of items are maximum and the total weight of them will be less than or equal to knapsack capacity. This is a kind of optimization problem to maximize profit. There are some algorithms to solve this problem.

2. Methodology

First, I have read the files in appropriate format and create value and weights array from them, and take the information of number of inputs and total capacity of knapsack from the first line. Then, I have implemented a solution for Knapsack Problem which is based on dynamic programming. First, it worked properly for input sample1a, and sample1b. But when I tried to run my code with the sample1c, sample1d and knapsack_big, it didn't work and gave segmentation fault, because of very big space complexity. So, I changed my code in order to use much less space. I decided to use greedy approach, it will work for large spaces. I have created my greedy code from the algorithm I inspire from textbook, section 12.[1]

Greedy algorithm for the discrete knapsack problem

Step 1 Compute the value-to-weight ratios $r_i = v_i/w_i$, $i = 1, \ldots, n$, for the items given.

Step 2 Sort the items in nonincreasing order of the ratios computed in Step 1. (Ties can be broken arbitrarily.)

Step 3 Repeat the following operation until no item is left in the sorted list: if the current item on the list fits into the knapsack, place it in the knapsack and proceed to the next item; otherwise, just proceed to the next item.

Figure 1. Algorithm for greedy approach to solve knapsack problem

I calculated ratio values for all data and I kept them in a float array that I called ratio[]. Then, I used quick-sort algorithm that I prepared last homework in order to sort ratios decreasingly. While I do that I sort the values and weights with the ratio, cause I need them sorted too. Then starting from the biggest ratio value, my code adds all the values that their weight is available for knapsack. If the weight is bigger than the knapsack's remained capacity, then algorithm goes for the next one and check again until there is no input left. Greedy approach didn't give the best optimal solution, but the result are very close to the optimal. Here, the result of my greedy code and the optimal result for corresponding inputs.

Input / Output	Solution of My Code	Optimal Solution
Sample1a.dat	35	35
Sample1b.dat	2375	2397
Sample1c.dat	54386	54503
Sample1d.dat	90199	90204

Table 1.Results for greedy approach

Even the result are very close, I have tried to improve my code for the best. If the input items are dividable that means problem is continuous, then I can adjust the code. When the item i is not suitable, then take the part of it as much as the remaining capacity takes. The result for this approach would be better than the given optimal values but I thought that we need to consider our input as discrete inputs because in our problem we consider only 0 or 1, included in knapsack or not included.

I made a research for new improvement and I come across with an article [2] about comparison of different algorithms on knapsack problem. They used dynamic programming, greedy approach, brute force, branch and bound and genetic algorithm. When I examine the output of their result, brute force, branch and bound, genetic algorithm and dynamic programming are not usable for large inputs. However, dynamic programming always yields the optimal solution where greedy is failing to give. In our task, we have been asked to run the algorithm for big inputs and we are trying to find optimal value. I couldn't find a way too improve dynamic approach to execute it with very large input, so, I have decided to use greedy approach method.

For the multiple knapsack problem, I used same basic idea where I had in the single knapsack problem. I used greedy approach, I sorted items according to the value to weight ratio non-increasing order. I filled knapsacks in order and I checked never to use same item twice or more.

Then I created output files as asked format. I have calculated the time passed while running each input. Time 1 is for all program and Time 2 is for algorithm. Here below time values in millisecond for each input.

Problem 1	Test1a	Test1b	Test1c	Test1d
Time 1	2	5	20	3
Time 2	0	1	3	1

Table 2. Time in milliseconds for test inputs of Problem 1

Time efficiency for my algorithm is consists of knapsack code and quick sort code. For the quick sort, it is $\Theta(n\log n)$ where n is the number of items. For the knapsack code of problem 1, it is O(n) because I am doing at most n comparison to include it or not. In total time efficiency for problem 1 is $\Theta(n\log n)$. For the test input 1c, we have 10.000 inputs and it should take about 4 milliseconds to operate. For the reading I am doing n times read and I am having another stuff going on in total the time reaches 20. I choose greedy approach because of its less time efficiency and less space efficiency and I sacrificed best optimality for

time and space. Space complexity for my algorithm is n because it keeps n values for ratio values and n values for array to say included or not.

Problem 2	Test2a	Test2b	Test2c	Test2d	Test2e
Time 1	2	1	3	2	16
Time 2	0	0	0	0	3

Table 3. Time in milliseconds for test inputs of Problem 2

There is no much to say about multi-knapsack time efficiency because it is nearly 0. Difference of this algorithm from the problem 1 is that I am spending m times more time where m keeps the number of knapsacks. I am spending n+m times space for keeping total capacity of m knapsacks in an array.

3. References

[1] Anany Levitin, Introduction to design and analysis of algorithms, 3rd edition.
[2] Different Approaches to Solve the 0/1 Knapsack Problem,Access Date May 28,2019.
http://www.micsymposium.org/mics_2005/papers/paper102.pdf