

In this exercise we study greedy optimization algorithms in case of knapsack problem.

Knapsack problems are planning problems involving the optimal selection of differently-sized items, which must be placed in a container of limited capacity. These problems are common in logistics (where items must fit into cargo bays, aeroplanes, trucks), in the buying/selling of advertising space (e.g. online) and in cutting problems (e.g. how best to cut up a board or strip into component pieces.)

Often, knapsack problems are multi-dimensional and may involve multiple knapsacks (containers). However, we will consider three simple 1-D knapsack problems here, as follows.

Excercise 6a. (Implement brute force 0/1 knapsack resolving algorithm, 1p)

In an 0/1 knapsack problem an instance consists of a set of N items with weights w_i and values v_i , for $i=1$ to N , and an integer, C , the carrying capacity of the knapsack. A solution to the problem is a subset of items having a total weight less than or equal to C that is maximal in total value (among all such subsets). The problem above is "0/1" because we either do carry an item: "1"; or we don't: "0".

First task: Input/Output

Write a program that reads in a knapsack problem instance text file written in the form

```
N
1  v1  w1
2  v2  w2
.    .
.    .
N  vN  wN
C
```

Test your code by reading in the knapsack problem instance easy20.txt (available from Tuubi) and then write the instance to the screen for visual inspection.

Second task: Full Enumeration (brute force search)

The 0/1 problem can be solved by a full enumeration of the search space. Every binary string of length N represents a valid candidate solution.

Adding to your code from the previous part, write a program that loops through every possible binary value of length N and evaluates the value and weight of each of these solutions. Note that your solution should work with an arbitrary large value of N (e.g. 200).

Output upon termination, the program should output the following:

```
Optimal solution found:  <value> <weight>
<item> <item> <item> ...
```

where the items are the indexes of the items to be placed in the knapsack, listed in ascending index order.

You could also get the program to output something to show its progress, i.e. the fraction of the search space it has enumerated. E.g. every 1,000,000 solutions evaluated it could output a line like "Y % of the search space enumerated so far".

Run your code on easy20.txt. You should get the answer in very short time even your program tried $2^{20}=1048576$ different combinations of items. The correct answer (optimal solution) is 726 value units and combined weight is 519.

Then run your code on hard33.txt. Now there are $2^{33}=8589934592$ combinations ($2^{13}=8192$ times more). Running time (under Netbeans) is about 14-15 minutes with Intel i5-6600K 3.5 GHz, compiled with fully optimization).

You may be tempted to try easy200.txt. Running brute force search algorithm requires about $6,1 \cdot 10^{45}$ years to complete (the remaining time for the universe is estimated to be $26 \cdot 10^9$ years).

Excercise 6b. (Optional, Implement Greedy 0/1 knapsack resolving algorithm, 0,25p)

Greedy search can be applied to the 0/1 Knapsack problem, but it does not guarantee to give an optimal solution.

Remember that a greedy search has the form:

```
While solution not constructed
  Select best remaining element and try adding it in
```

What is the sense of "best" here? I.e. what is the criterion by which the items should be ordered?

Write a program that implements the greedy method for the 0/1 problem. In your implementation, first sort the items by your criterion so that you don't have to search through all of them each step. One quite useful criterion could be value density v_i/w_i (so we will be greedy on value density).

Output upon termination, the program should output the following:

```
Feasible solution (not necessarily optimal) found:  <value> <weight>
<item> <item> <item> ...
```

Run your greedy algorithm code on the two problem instances, easy.20.txt and hard33.txt. Compare the results of brute force search for hard33.txt (optimal solution for hard33.txt

should be 22486 value units, and greedy solution should have 19388 value units; about 14% less than the optimal).

Excercise 6c. (Optional, Improve greedy algorithm with a heuristic, 0,25p)

We can modify the greedy algorithm to include heuristic to provide solutions within x percent of optimal for $x < 100$. First we place a subset of at most k objects into the knapsack. If this subset has weight greater than c , we discard it. Otherwise, the remaining capacity is filled by considering the remaining objects in decreasing order of v_i/w_i . The best solution obtained considering all possible subsets with at most k objects is the solution generated by the heuristic.

The solution produced by this modified greedy algorithm is k -optimal. That is, if we remove k objects from the solution and put back up to k , the new solution is no better than the original. Further, the value of a solution obtained in this manner comes within $100/(k+1)$ percent of optimal. When $k=1$, the solutions are guaranteed to have value within 50% of optimal; when $k=2$, they are guaranteed to have value within 33,33% of optimal; and so on. The run time of the heuristic increases with k . The number of subsets to be tried is $O(n^k)$, and $O(n)$ time is spent on each. Also, $O(n \log n)$ time is needed to order the objects by v_i/w_i . So the total time taken is $O(n^{k+1})$ when $k > 0$.

Implement this heuristic when $k=1$. In this case you put one item first at time to the knapsack, and after that run the original greedy algorithm for the remaining items. Then finally you select the best solution of those previous selections. Compare the performance of this heuristic to the original greedy algorithm (hard33.txt file should give 22340 value units which is 0,65% less than the optimal solution, 22486 value units, a substantial improvement to the normal greedy algorithm; hard200.txt file should give the value of 136724 with normal greedy algorithm, and 136972 with this heuristic algorithm).

You may use the following main() function to test all those previous tasks.

```
public class Lab06 {
    private final static String FILE = "H: (your path here)/easy20.txt";
    private final static int ALG = 1;

    /**
     * @param args the command line arguments
     */
    public static void main(String[] args) {
        KnapSack ks = new KnapSack();

        ks.readKnapSack(new File(FILE));
        ks.printKnapSack();

        long tic = System.currentTimeMillis();
        switch (ALG) {
            case 1:
                ks.bruteForce();
                break;

            case 2:
                ks.greedy();
                break;

            case 3:
```

¹ You can observe this using operating system tool for checking the processor utilization, e.g. in Windows using Task Manager

With this parallel computation technique you can speed up the search process. If you have four threads (in four core system), the execution time of the search is one quarter of the original time².

² And the CPU utilization is now 100%