Extended Essay in Computer Science

Comparing the efficiency of different algorithms for solving the knapsack problem

---

To what extent are the brute force, dynamic programming, greedy approximation and branch and bound appropriate approaches for solving the 0/1 Knapsack problem in terms of runtime and finding the optimal solution?

word count: 3845

Table of Contents

1. Introduction

One of many interesting patterns that are present in nature is that we move from making things work to making things work efficiently. From making a fire with two stones, to making a fire with a pull of a match; from writing an algorithm that produces the desired output, our first programs, to an algorithm doing so in much less time. The knapsack problem, goal of which is to find the most optimal subset from a set of items based on properties of the items (for instance value and weight), captures the essence of this pattern of finding a more efficient way; it reflects the real world in a way.

Even if the knapsack problem may look abstract in nature, the opposite is the truth. The application of the problem includes a broad spectrum of practical problems. Firstly, its role is essential in multiple parts of the manufacturing process – capital budgeting (constraint being the budget itself), cargo loading (limited volume), cutting stock, or distribution of products in the warehouse (the most requested products are chosen to be in the most convenient area). Secondly, the knapsack problem illustrates distribution of money among multiple investments, when the optimal solution leads to the best possible combination of investments (Chhajed and Lowe, 2008).

Be it a producer in need of filling the shipping containers with his products in the most optimal way, or a manufacturer of an optic cable searching for the most optimal lengths of cable to maximise his profit, they both strive for the most efficient way of employment of their resources. The knapsack problem gives them the opportunity to become more efficient in their work.

Furthermore, the knapsack problem can figure as a subproblem in more complex problems, such as two processor scheduling problem, fault tolerance problem or vehicle routing problems (Fidanova, 2005).

However, it is important to denote, that when dealing with large quantities as in cargo shipping, it is not always the most optimal solution that is needed, since that might require

simply too much time to compute. Likewise, regularly changing parameters in investments could render the optimal solution irrelevant with days of recomputing. For that reason, it is the balance between the best possible solution and the duration of producing such solution that is crucial.

With many different algorithms available for solving the knapsack problem, I deem it vital to determine, which one offers the best trade-off between their "quality" (success rate of computing the most optimal solution) and the time it takes them to do so. Therefore, the essay will be dedicated to examination of: ***to what extent are the brute force, dynamic programming, greedy approximation algorithms and branch and bound appropriate approaches for solving the knapsack problem in terms of time and finding the optimal solution?*** The type of the knapsack problem examined will the 0-1 knapsack problem. An experimental comparison will be conducted on 2 datasets designed to help distinguish the benefits of algorithms. Data about the duration of runtimes and the optimal solutions produced by the algorithms will be collected and evaluated.

2. Theoretical Background

2.1 Definition of Knapsack Problem

Knapsack problem is most commonly illustrated on the example of a burglar, who tries to fill his knapsack with items of known value and weight so that he will achieve a maximum net benefit of objects without exceeding the capacity of the knapsack.

In order to give the problem a mathematical format, we number the items from 1 to n. A vector of binary variables x_j (j = 1, …, n) denotes whether an item is selected and therefore present ($x_j = 1$) or not ($x_j = 0$). The profit (value) of an item j is given by p_j, its weight (volume) by w_j, and c is for the capacity of the container (knapsack), with a condition that profits, weights and the capacity are positive integers.

The most optimal solution to knapsack problem is the one that maximizes the function:

$$\sum_{j=1}^{n} p_j x_j \qquad \text{while} \quad \sum_{j=1}^{n} w_j x_j \leq c$$

The knapsack problem includes various formulations and types of the problem itself. In multiple knapsack problem there is more than one container available. In fractional knapsack problem can be the available items broken down into smaller parts, fractions, with the objective being to maximize:

$$\sum_{j=1}^{n} p_j f_j \qquad \text{while} \quad \sum_{j=1}^{n} w_j f_j \leq c \quad \text{and} \quad 0 \leq f_j \leq 1$$

In the essay the 0-1 knapsack problem will be examined, as it is the most general from the group, without the too specific characteristics of the rest. Furthermore, it does not increase the requirements for the dataset (available items for the knapsacks) as for example the multiple knapsack problem would. Therefore the 0-1 knapsack problem was deemed as the most eligible for the experiment from all types.

The problem belongs to combinatorial optimization problems, an area concerned with finding an optimal or close to optimal solution among a finite collection of possibilities. As it is a NP-complete problem, meaning it is nondeterministic in polynomial time, the time required to solve the problem increases quickly as the size of the problem grows. Therefore, in case of large problems, the quality of the solution may be partially disregarded as the optimal solution can be approximated, up from a certain point may become impossible to compute in a real time.

2.2 Brute force approach

For the first algorithm will be used brute force approach which involves no optimization. The essence of the approach lies in running through all the possibilities of knapsack composition to find the most optimal one. The knapsack can be represented as a bit string of length of n, in which 1 on the i^{th} position means that i^{th} item is chosen, while 0 on the i^{th} positions means

that the i^{th} item was not chosen. When choosing from n items, there are 2^n possible combinations of items chosen for the knapsack.

2.3 Dynamic programming

Dynamic programming is an optimization approach that transforms complex problems into sequences of smaller ones, solves each subproblem just once, stores the result and when the subproblem arises again, it takes the already computed solution (Bradley, 1977). A matrix can be used for storing solutions to subproblems.

With one row for each item (also one for no item) and one column for each possible integer weight capacity, the matrix represents the solution space.

For a knapsack of capacity w = 6 and 4 available items with values {1, 3, 2, 2} and weights {1, 2, 2, 4}, the matrix has a following look:

				immediate capacity w						
			i \ j	0	1	2	3	4	5	6
items	no items		0							
	val[i]=1	wt[i]=1	1							
	val[i]=3	wt[i]=2	2							
	val[i]=2	wt[i]=2	3							
	val[i]=2	wt[i]=4	4							

Figure 2.3.1 A not-filled up table illustrating dynamic programming approach

To fill it up, for each cell with i available items and weight capacity w is attributed the higher value of either *maximum value obtained by i-1 items* at the capacity of w (when we do not take the n item), or *value of the item i plus the maximum value obtained by the i-1 available items at remaining capacity* (in case that we include the item i), with the condition that the weight of the item i is smaller than the remaining capacity of the knapsack. Following piece of code represents the just mentioned procedure of filling a cell of the table:

```
int f1 = f[i - 1][w] ;
int f2 = val[i] + f[i - 1][w - wt[i]];
f[i][w] = Math.max(f1, f2);
```

Figure 2.3.2 code example

				immediate capacity w						
			i \ j	0	1	2	3	4	5	6
items	no item		0	0	0	0	0	0	0	0
	val[i]=1	wt[i]=1	1	0	1	1	1	1	1	1
	val[i]=3	wt[i]=2	2	0	1	1	4	5	5	5
	val[i]=2	wt[i]=2	3	0	1	1	4	5	6	6
	val[i]=2	wt[i]=4	4	0	1	1	4	5	7	8

Figure 2.3.3 Filled up table illustrating dynamic programming approach

The value of the optimal solution lies in the rightmost bottom cell, which is in the example matrix (*Figure 2.3.3*) equal to 8.

2.4 Greedy approximation approach

Greedy approximation approach is often employed in optimisation problems, Knapsack problem being one of them. Being based on making locally optimal choices, after series of which it is supposed to get to the globally optimal solution, it does not return to evaluate previously castaway, locally not optimal, possibilities (Sharma, 2016).

There are several strategies of greedy approximation approach for solving the 0/1 knapsack problem:

Filling the knapsack with the most precious item from the remaining ones, filling the knapsack with the items with the smallest weight from the remaining items, or filling the knapsack with items based on the ratio of the item's value and weight. The literature (Hristakeva, 2018) shows that the third strategy, working with ratios of the value and weight of items, is the most effective from those three. For that reason, it will be implemented in the experiment as a

representative of the greedy approximation approach, being further referred to as a greedy approximation algorithm.

2.5 Branch and bound approach

Branch and bound approach, rather than evaluating each possible solution, constructs candidate solutions one component at a time and then evaluates these partly constructed solutions. If there is a possibility to move to a next component with keeping problem's constraints unviolated, the next option for the next component is taken. Otherwise, with no remaining option for further development of the solution, the algorithm backtracks and takes a next option at the last component of the impartial solution.

With the items sorted in the non-increasing order based on their value-to-weight ratios, we can construct a binary tree where the nodes represent subsets of n items, the candidate solutions for the optimal knapsack composition. Having left path meaning that an item is included and right one that it is omitted, nodes on the level i illustrate the subsets of the first i items.

The bound in the name of the approach refers to a maximum value at the node that can be obtained by further development of the partially constructed solution. Therefore, if the bound value at a node is smaller than the value of the current best solution, the node does not have to be examined further as we already know that no better solution can be obtained from it. Such bound value *b* can be calculated as:

$$b = v + (W - w)(v_{i+1}/w_{i+1}),$$

where v refers to the value of items already included in the knapsack, (W-w) marks the remaining capacity, to be multiplied by the maximum value-to-weight ratio, which, since the items are ordered in a non-increasing way, will belong to the next item (Levitin, 2012).

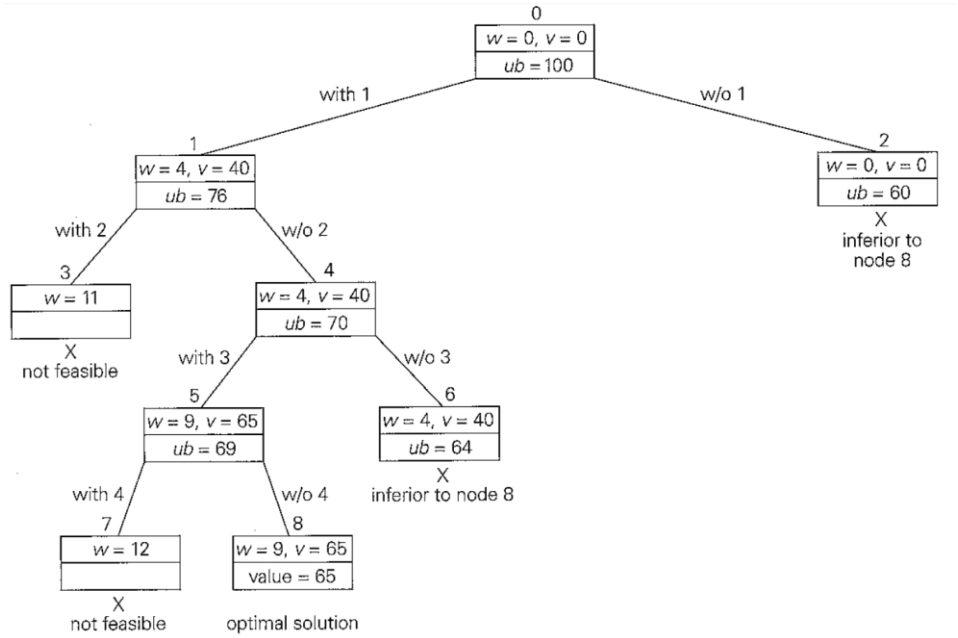

Figure 2.5. (Levitin, 2012)

3. Experiment

3.1 Method

As the aim of the extended essay is to evaluate which of the chosen algorithms are the most efficient in finding the most optimal subset from a set of items with a known value and weight, I will run the algorithms on precomputed datasets. My independent variable will be the algorithms themselves as well as the size of the problem expressed in number of available items that will be given as inputs to the algorithms. Data obtained by running the algorithms will consist of time durations of the runs and the optimal solutions produced as the output.

For the evaluation of the optimal solutions, the method of determining the quality of an approximation was selected as the most appropriate way. The method, in the maximization problems such as Knapsack, measures the quality of solutions in terms of the ratio of the optimal value to approximation, also called the approximation factor (Berg, 2013). For the purposes of the essay, I will use my altered definition, where approximation factor f is the ratio of the optimal solution for the input I to a solution given by an algorithm for the input *I*.

$$f = \frac{optimal\ solution(I)}{generated\ solution(I)}$$

I believe such strategy to clearly show to what extent do the not-optimal solutions deviate from the optimal ones. The greater the approximation factor, the more the generated solution diverges, while the correct solution is given by an algorithm when the factor f is equal to 1.

As for the runtimes, to get the most relevant idea of the time it takes the algorithms to compute the optimal solution, they will be executed each time in 10 repetitions in order to minimise the variation caused by the operation system running other processes, which cannot be considered constant. The precomputed input parameters for every repetition will be different to avoid any random failures of algorithms on certain extreme input values, which does not reflect the total efficiency of the algorithms. From two ways of finding the average number of runtime for one size of the problem – measuring the time it takes to run all 10 repetitions and dividing that number by 10 or measuring the individual runtimes for each of the 10 repetitions and taking the average of those; the second way was selected. Otherwise, such average runtime would include the time it takes using Arrays.copyOfRange() to prepare arrays with values (weights and prices) specific and exclusive to each repetition, which should rather not be counted into the runtimes of algorithms as it increases the values, even if only linearly.

The data will be displayed in form of graphs with the use of standard deviation for error bars for each point on the graph. The standard deviation will show how much are the individual values of the repetitions (that add up together the average value which is marked on the graph) spread. With low standard deviation indicating low spread of the individual values and high standard deviation indicating that the individual values differ greatly from the average (Illowsky, 2011), the magnitude of the standard deviations the will enable me to determine whether the difference between values for different algorithms is significant or not.

The experiment will consist of:

- implementation of brute force, dynamic programming, greedy approximation and branch and bound algorithms in Java

- generating datasets
- executing the algorithms on a Windows 10 64-bit computer with 8GB of RAM, Intel® Core™ i5-4460 CPU clocked at 3.20 GHz
- processing of obtained data
- evaluation of the collected data

3.2 Datasets

For the first dataset would be used only smalls sets of available items, the size being in the range from 1 to 30, the focus being on the success rate of finding the most optimal solution. All algorithms are able to run in relatively good time for these relatively small sizes of the problem. The 10 repetitions for each size per algorithm will include various values of the inputs, all in the defined range of values.

Furthermore, the algorithms will be run on the second dataset which will contain high numbers of items to choose from. In practice, that means that brute force algorithm will no longer be able to give any results in relevant time, as was shown in a test run. For that reason, it will be excluded from this part. The rest of the algorithms will be once again run in 10 repetitions. The goal of this part will be to distinguish the efficiency of the algorithms in situations closer to reality, when huge amounts of data is supplied to the algorithms, needed to be quickly evaluated.

3.3 Hypothesis

My hypothesis is that brute force algorithm will prove highly inefficient in terms of runtime in comparisons to the other algorithms, as it does not use any optimization. In this regard, greedy approximation algorithm should prove the fastest, since it only needs to sort the items according to the decreasing value-to-weight ratio and choose the first ones that fit into the knapsack.

However, as it will not produce every time the correct optimum solution, it will be important to evaluate and distinguish between dynamic programming and branch and bound algorithms.

3.4 Data collection and evaluation

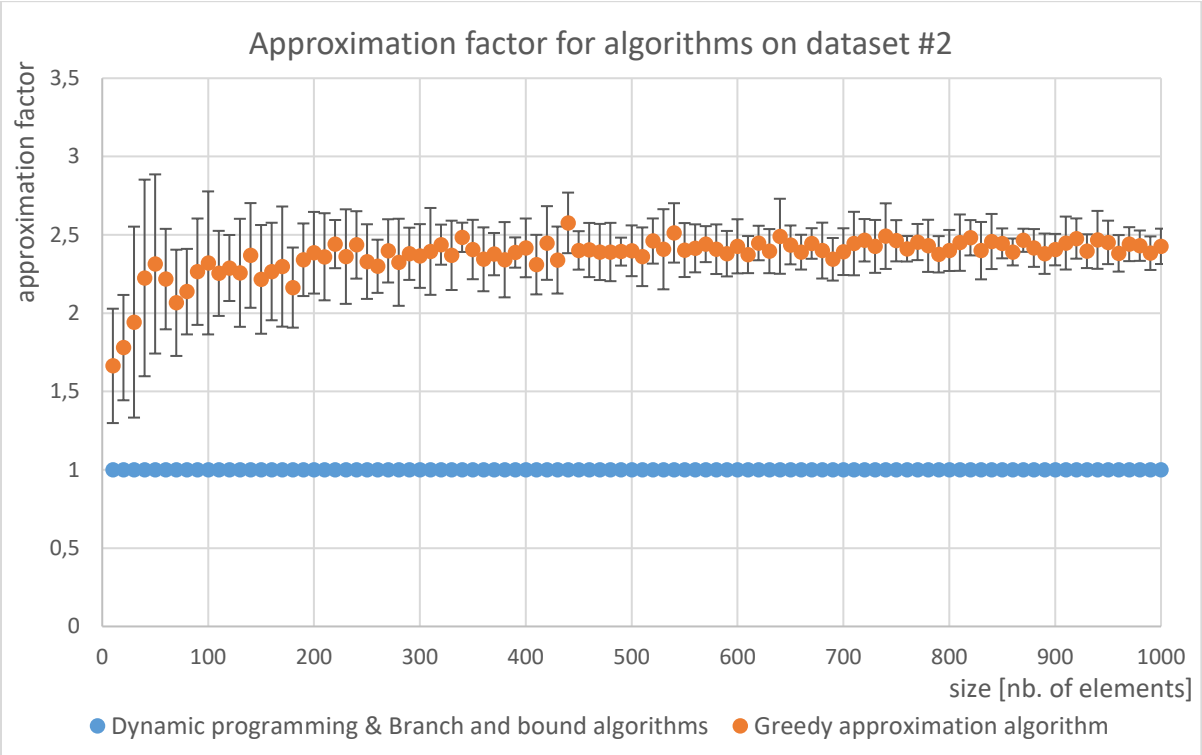

Figure 3.4.1 Approximation factor for solutions produced by dynamic programming, branch and bound, and greedy approximation algorithms

Regarding the success in producing the correct optimal solution, brute force, dynamic programming, and branch and bound algorithms accomplished to produce the correct solution in each and every execution, for both the first and second dataset. As can be seen on figure 3.4.1, the greedy approximation algorithm did not accomplish to generate a correct optimal solution in average in any case (although if looked at the data in a form of a table, it reaches the correct optimal solution a couple of times for small sizes of the problem), the error bars indicating the standard deviation in the repetitions of the algorithm.

As all algorithms (brute force, dynamic programming and branch and bound), but the greedy approximation algorithm, consistently achieve an approximation factor of 1, the experiment confirmed that the three algorithms can produce the correct optimal solution on both small and large scales. Such result was expected as the three algorithms do not use any approximation and mainly differ only in how efficiently they reuse already computed data.

More intriguingly, the pattern of the approximation factor for the greedy approximation algorithm seems to converge to the number 2.42, with the error bars indicating more consistent results with the increasing size of the inputs.

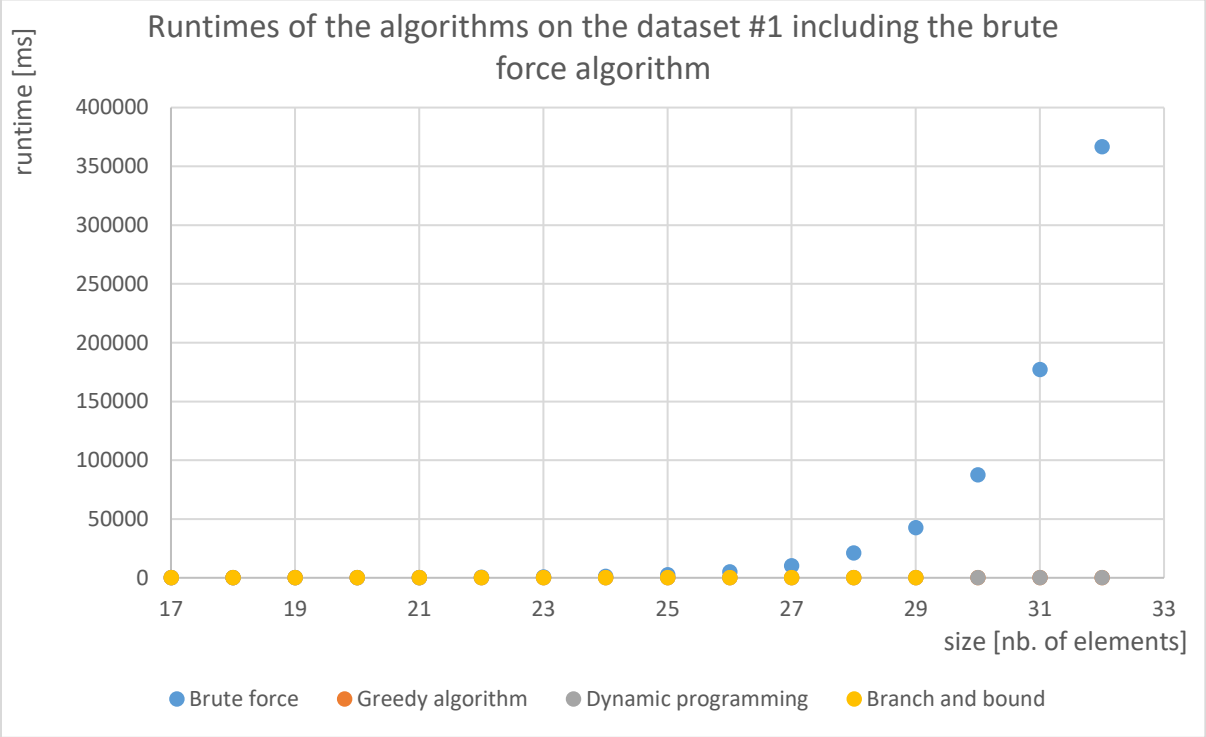

Figure 3.4.2 Runtimes of the brute force, dynamic programming, branch and bound, and greedy approximation algorithms on the dataset #1

The figure 3.4.2 showing the runtimes of the algorithms on the smaller dataset displays a sharp exponential behaviour of the brute force algorithm. Even the small dataset is enough to show how ineffective evaluating every possible solution in order to find the most optimal one, loosing ability to compete with other algorithms as soon as when having the input of 9 items.

As for the dataset #2, the runtime of the greedy approximation algorithm increases at the smallest rate, as it performs a sorting and goes only once though the array of sorted price-to-weight ratios to choose those with the biggest values. However, while the dynamic programming takes at the number of 1000 items 400x more time to compute the solution than the greedy algorithm, the branch and bound algorithm manages to do so in only 8x more time than the greedy one, and therefore coming as a close second concerning the time duration.

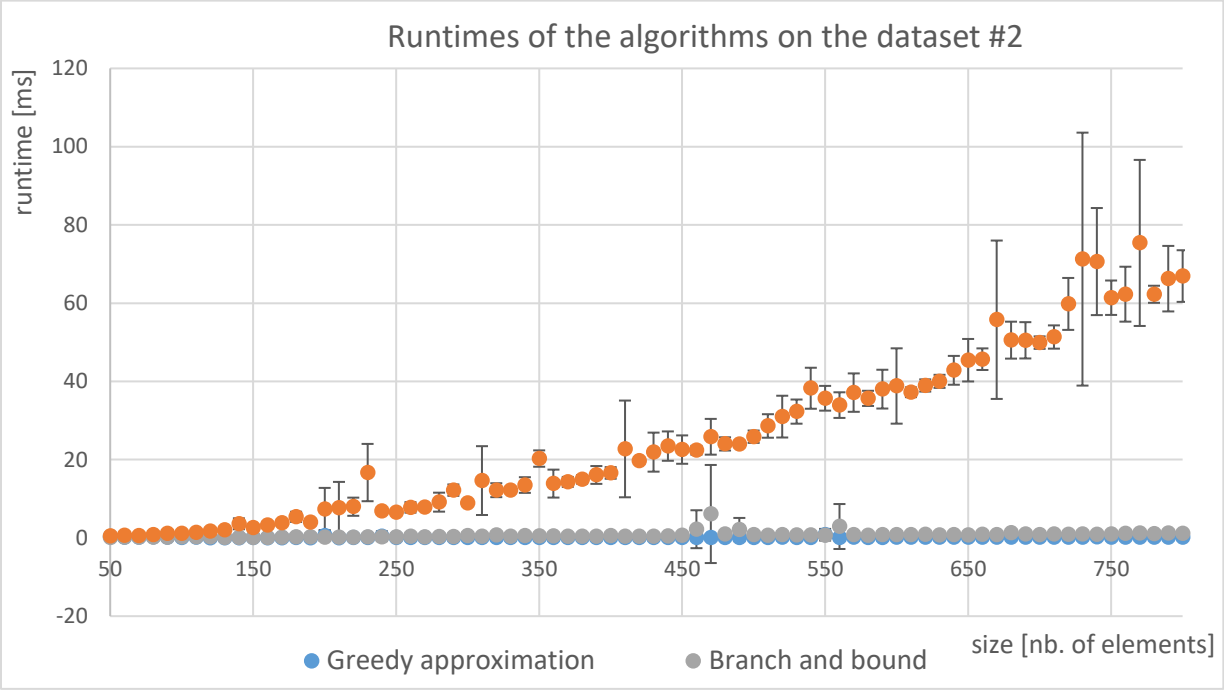

Figure 3.4.3 Runtimes of the dynamic programming, branch and bound, and greedy approximation algorithms on the dataset #2

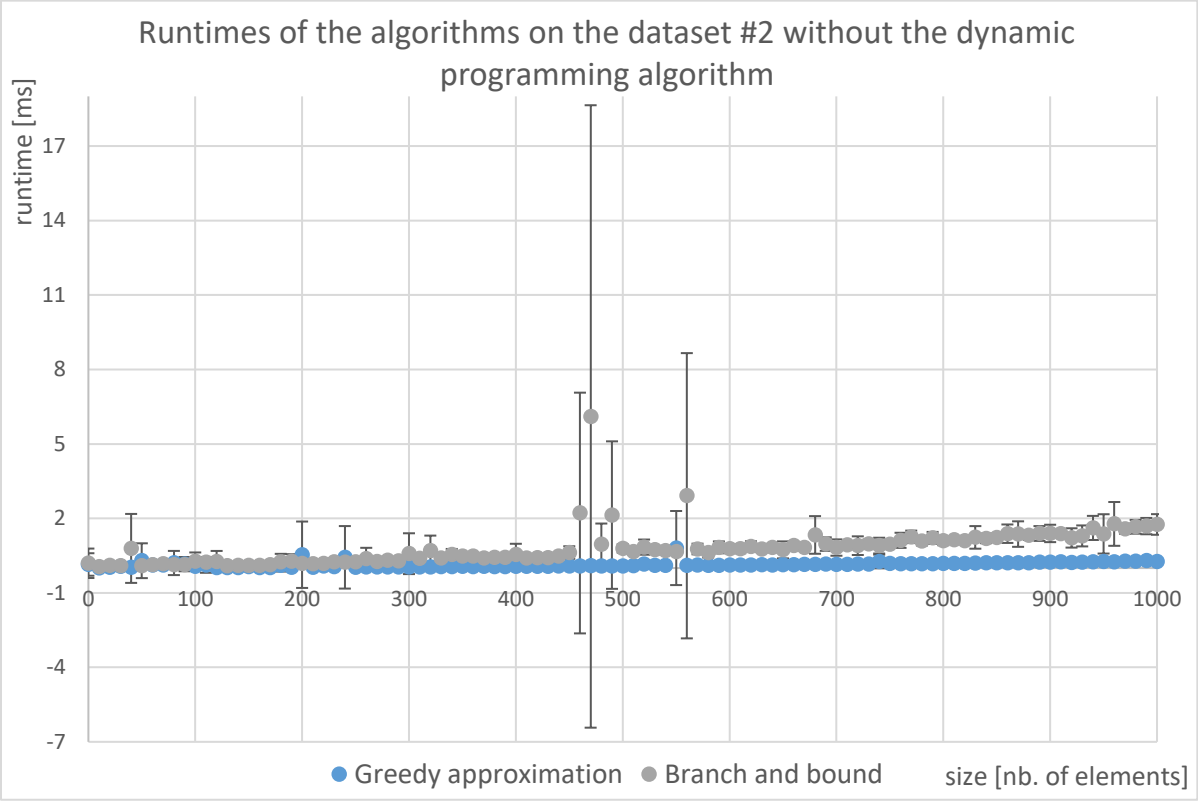

Figure #3.4.3 Runtimes of the branch and bound, and greedy approximation algorithms on the dataset #2

3.5 Limitations

The method of the experiment includes several limitations that need to be accounted in order to draw conclusions from the obtained data.

To begin with, since the operating system was executing other tasks unrelated to my program while the algorithms computing the knapsack problem were run, these other processes added up to occasional and irregular spurts of the runtimes for individual sizes of the inputs. Especially in the last graph (*Figure #3.4.3.*), for the branch and bound algorithm there are several values of runtimes that diverge highly from the pattern that the rest of the points follows, especially around the size of 500 items. To determine the cause of these values that stick out I ran the algorithms branch and bound and greedy approximation algorithm once again, on the dataset number 2 consisting of the same unchanged values that were used for the *Figure #3.4.3* one more time. As the values that stuck out in this second repetition (*see Appendices, Figure 7.1*) were for other number sizes of the input than in the original case, it can be concluded that these values were not caused by the algorithms failing on certain inputs, but rather have a different cause, in my view most possibly by the random spurts of the operation system.

Linked to the first limitation, another one is a low number of repetitions. The points on the *Figure #3.4.3* that stick out are accompanied by substantially bigger span of error bars. To account for the random values out of the patterns, caused for instance by the operating system, firstly, repetitions were conducted for each case, and secondly, standard deviation was used to also show to what degree was the distorting effect of the random values reduced. From the data shown in the figure *3.5.1 below* can be deduced, that the off values were of enough magnitude to significantly increase both the average value of the repetitions and the standard deviation.

			repetitions									
number of items	average of repet.	standard deviation	#1	#2	#3	#4	#5	#6	#7	#8	#9	#10
560	2.9128	5.745	**19.142**	0.965	0.640	1.300	0.615	2.607	0.600	0.610	0.608	2.039
570	0.7684	0.215	0.678	1.184	0.635	0.828	0.585	0.629	0.630	0.613	0.789	1.113

Figure #3.5.1 Individual runtimes of each repetition of the branch and bound on the dataset #2 for number of items 560 and 570

More repetitions would decrease the overall effect of such off values. Therefore, it can be concluded that 10 repetitions were too few and more should have been done.

Furthermore, it is extremely important to mention that for the reason of the limited length of the essay, only one algorithm was selected for each approach, even though there are various ways of implementing the chosen approaches to find out the optimal solution to the knapsack problem.

Lastly, brute force algorithm was tested only on the dataset #1, running over only a small range of number of inputs, since, based on the exponential growth on the first graph as well as on the theory, it would take days to run the algorithm on the dataset #2.

4. Conclusion

The most optimal approaches to finding to optimal solution to a 0/1 knapsack problem in terms of actually succeeding in computing exactly the optimal solution are all three brute force, dynamic programming and branch and bound approaches. We cannot further distinguish between them in this regard, as they all accomplished to produce correct optimal solutions at all times. On the other hand, greedy approximation algorithm did not output optimal solutions in none but for the smallest knapsack sizes. It can be further concluded, that the greedy approximation algorithm is not fit to calculate the knapsack problem in any case, as even though it manages to reach some optical solutions correct for very small sizes, it is unreliable even on such range. In addition, the solutions that the algorithm produces on broader range are too different to the correct ones, managing to fill the knapsack with items

of overall value two times smaller, which was deemed as a too big a difference even for an approximation.

The most optimal approaches to finding to optimal solution to a 0/1 knapsack problem in terms of the time are branch and bound and greedy approximation approaches. The runtime of the brute force algorithm was increasing exponentially, no competition for the rest three algorithms. As for the dynamic programming algorithm, it still took several hundreds more times to compute a solution to it than to the greedy approximation algorithm, from which the branch and bound algorithm exhibits only slightly more steep increases in the runtime over increasing size of the problem.

The fact that the branch and bound algorithm performed better than the dynamic programming algorithm implies that only a small portion of the constructed tree is needed to cover in practice, as while the algorithm shares the idea of reusing partially computed subproblems with dynamic programming approach, the two approaches differ in that the branch and bound algorithm does not compute certain subproblems once they are deemed infeasible solutions and the dynamic programming always counts the subproblem value for every place in the table.

To conclude, the overall best performance belongs to the branch and bound algorithm.

Even with mentioned limitations, the drawn conclusions are considered significant as there is from none to only unsubstantial overlapping of the error bars for the points on the graph.

5. Future research

While working on the essay, multiple directions to which the research of the Knapsack problem could be taken occurred to me, many of them associated with including more of real life features, as for instance taking as input more parameters of an item than only price and weight, or binding certain items together, when one item would depreciate without the presence of another item. How would have to be algorithms that I used modified and to what extent would such changes affect their efficiency? In such cases the research becomes greatly

narrowed down. One of the ideas for the further research relevant to more situations could be focusing on different implementations of another – genetic approach, or increasing the success rate of approximation algorithms, as the one examined was found to be the quite efficient in terms of time, if not in any other regard, and their research could increase their potential.

6. Bibliography

Berg, M. d. (2013) *Advanced Algorithms (2IL45)*. [online] p. 30 Available at: http://www.win.tue.nl/~mdberg/Onderwijs/AdvAlg_Material/course-notes-AA-2013.pdf [Accessed 10 Nov. 2018]

Bradley, S. P. (1977) *Applied Mathematical Programming*. United States: Addison-Wesley Publishing Company, p. 320

Chhajed, D. Lowe, T. J. (2008*) Building Intuition: Insights from Basic Operations Management Models and Principles*. Springer, p. 29

Fidanova, S. (2005) *Heuristics For Multiple Knapsack Problem*. p. 1 [online] Available at: https://pdfs.semanticscholar.org/2dbc/6dff3ce74a906ec5f1af4a4868a68313c7a8.pdf [Accessed 7 Nov. 2018]

Hristakeva, M. (2018) *Different Approaches to Solve the 0/1 Knapsack Problem*. p. 5. [online] Available at: http://www.micsymposium.org/mics_2005/papers/paper102.pdf [Accessed 7 Nov. 2018]

Illowsky, B. and Dean, S. (2011) *Collaborative Statistics. Eleven Learning*, p. 81

Levitin, A. (2012) *Introduction To Design And Analysis Of Algorithms*. 3rd ed. England: Pearson Education, pp.423-437

Sharma, A. (2016) *Basics of Greedy Algorithms*. [online] Available at: https://www.hackerearth.com/practice/algorithms/greedy/basics-of-greedy-algorithms/tutorial/ [Accessed 11 Nov. 2018]

7. Appendices

7.1 Graphs

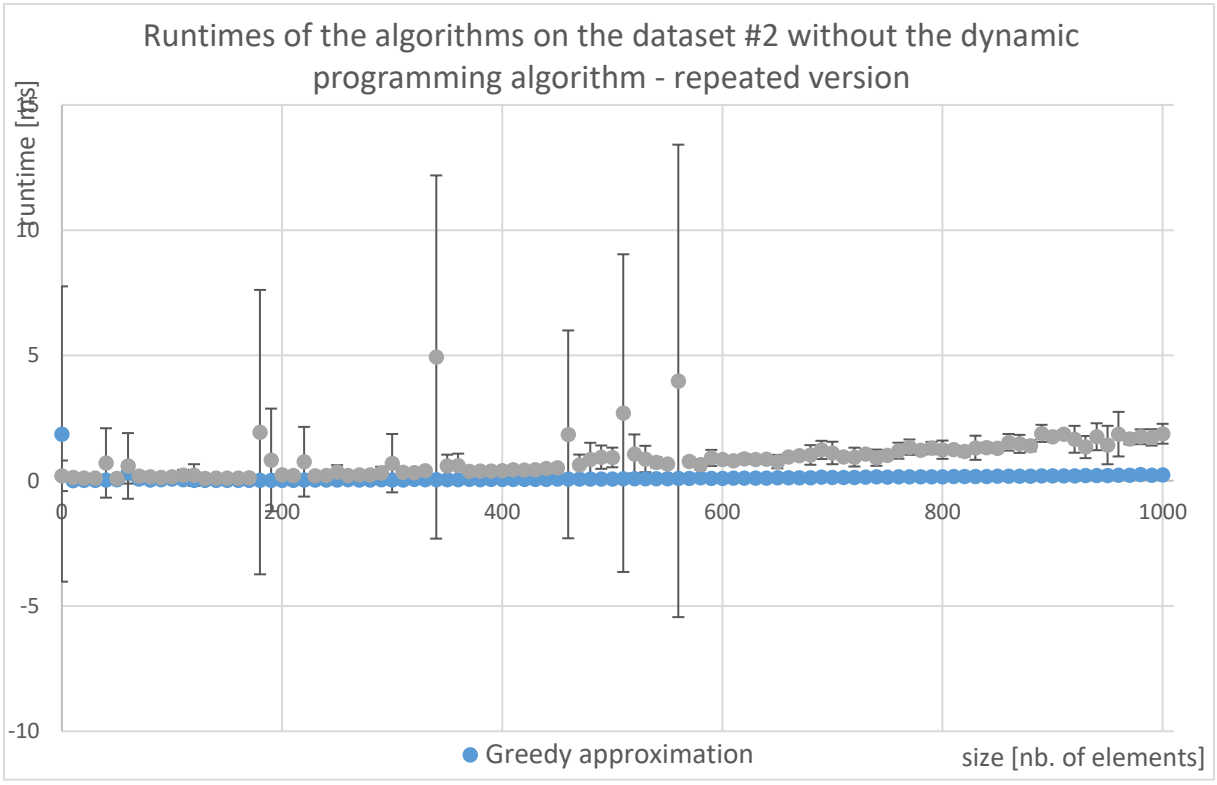

Figure #7.1 Runtimes of the branch and bound, and greedy approximation algorithms on the dataset #2, repeated version

7.2 Brute force algorithm

Author of the code to the brute force algorithm: Melany Diaz.

The code was modified by the author of the essay.

```
package com.company;/*
   Class: Bruteforce

   Author: Melany Diaz
   with assistance from: Gerry Howser
```

```
   Creation date:  3/5/2016

   Modifications:
        Date        Name        reason
       03/09/2016   Melany Diaz Implemented Binary Counter
*/

import java.util.ArrayList;

/**
 This class will implement the Brute force solution: Try all possible
 combinations of xi and take the maximum value that fits within the
 given capacity.
 */
public  class BruteForce{

    //instance variables
     private static int capacity = 0;
     private static int[] prices;
     private static Integer[] weight;
     private static int numItems = 0;

     private static int currentValue = 0;
     private static int maxValue = 0;
     private static int currentWeight = 0;

    //the knapsack
    public static Integer[] permutations = new Integer[numItems];

    public static boolean overflow = false;

    //Constructors
    public BruteForce()
    {
        capacity = 0;
        prices = new int[0];
        weight = new Integer[0];
        numItems = 0;
        currentValue = 0;
        maxValue = 0;
```

```
        currentWeight = 0;
    }

    // Methods

    /**
     * finds the solution to the 0/1 knapsack problem
     *
     * Pre-condition:  Array has a length > 0, the integers in the price
and weight array are positive, and the
     *                      prices array has the same number of items as
the weights array
     * Post condition: The returned array is filled with integer "0" and
     *                      overflow is set to "false".
     *   @returns the value of the best combination
     **/
    public static int bruteForce(int capacity, int[] prices, int[] weight,
int numItems) {
        //checking preconditions
        assert (numItems > 0);
        assert (prices.length == weight.length);
        for (int i = 0; i < numItems; i++){
            assert (prices[i] >= 0);
            assert (weight[i] >= 0);

        }

        int[]permutable = new int[numItems];
        permutable = Reset(permutable);   //resets permutable to all 0's

        while (!overflow) {
            //go through the array of permutations
            for (int i = 0; i < permutable.length; i++){

                //if there is an item the thief is taking, add it's weight
                //to the weight the thief is considering taking
                if(permutable[i]  == 1) {
                    currentWeight += weight[i];

                    //if that weight fits in the knapsack, add the
                    //values of the prices
```

```
                    if (currentWeight <= capacity) {
                        currentValue += prices[i];
                        //find the most optimal value for the thief
                        if(currentValue > maxValue)
                            maxValue = currentValue;
                    }
                    //if the weight was too much for the knapsack's
capacity, value is 0
                    else
                        currentValue = 0;
                }
            }

//             //to print all of the combinations of the knapsack considered
//             //the last one printed is the most optimal
//             if(currentValue == maxValue)
//                 System.out.println("the permutation is now:  " +
toString(permutable) + " Weight: " + currentWeight + " value: " +
currentValue + "\t");

            currentWeight = 0;
            currentValue = 0;

            if (!overflow) {
                permutable = Bump(permutable);
            }
        }

        return maxValue;
    }


    public long TimeToFind(int capacity, int[] prices, int[] weight, int
numItems){
        long start = System.nanoTime();
        int value = bruteForce(capacity, prices, weight, numItems);
        long end = System.nanoTime();
        long duration =  end- start;
        System.out.println("Value for brute force: " + value);
        return duration;
    }
```

```
    public String[] bruteForceSolTime(int capacity, int[] prices, int[]
weight, int numItems){
        String[] solTime = new String[2];
        long start = System.nanoTime();
        int value = bruteForce(capacity, prices, weight, numItems);
        long end = System.nanoTime();
        long duration =  end- start;
        //System.out.println("Value for brute force: " + value);
        solTime[0] = Integer.toString(value);
        solTime[1] = Long.toString(duration);
        return solTime;
    }

    /**resets the permutation array to all 0's.
     * Pre-condition:  Array has a length > 0
     * Post condition:  The returned array is filled with integer "0" and
     * overflow is set to "false".
     */
    public static int[] Reset(int[] x)
    {
        assert (x.length > 0);
        int i = 0;
        overflow = false;
        while (i < x.length)
        {
            x[i] = 0;
            i++;
        }
        return x;
    }


    /**returns a permutation of all possible combinations of "1"s and "0"s
that an array of
     * size n can have
     *
     * Pre-condition:  Array contains only "0" and "1" and length > 0
     * Post condition:  The returned array is "bumpped" by 1 as a binary
counter
     *                If the binary counter overflows, overflow is set to
```

```
     *                "true" otherwise overflow is set to "false"
     */
    public static int[] Bump(int[] x)
    {
        assert (x.length > 0);
        assert (isBinary(x));
        int i = x.length - 1;
        overflow = true;
        while ((i >= 0) && (overflow))
        {
            if (x[i] == 1)
            {
                x[i] = 0;
            }
            else
            {
                x[i] = 1;
                overflow = false;
            }
            i--;
        }
        return x;
    }

    //takes the array of permutaitons and transforms it to a printable
string
    public static String toString(int[] x)
    {
        String result = " ";
        int i = 0;
        while (i < x.length)
        {
            result = result + x[i];
            i++;
        }
        return result;
    }

    //Prints out the different permutations of a binary array
    public void printPermutations(int[] permutable) {
        permutable = this.Reset(permutable);
```

```
        while (!this.overflow) {
            System.out.println("the permutation is now:  " +
this.toString(permutable) + "\t");
            if (!this.overflow) {
                permutable = this.Bump(permutable);
            }
        }
    }

    //assert methods
    //confirms that the integers in the permutation array are just 0s and
1s
    public static boolean isBinary(int[] x)
    {
        boolean result = true;
        int i = 0;
        while ((i <= x.length) && (result))
        {
            if ((x[i] != 0) && (x[i] != 1))
            {
                result = false;
            }
            i++;
        }
        return result;
    }
}
```

7.3 Dynamic programming algorithm

Author of the code to the brute force algorithm: Melany Diaz.

The code was modified by the author of the essay.

```
package com.company;/*
   Class: Dynamic

   Author: Melany Diaz
   with assistance from cs.princeton.edu and Gerry Howser
```

```
   Creation date:  3/5/2016

   Modifications:
        Date    Name        reason
        3/10/16 Melany Diaz enhansed
*/

import java.util.ArrayList;
import java.util.Arrays;

/**

 This class will implement the Dynamic Programming solution.

 */
public class Dynamic {

    //instance variables
    private static int capacity;
    private static int[] prices;
    private static Integer[] weight;
    private static int numItems;

    private static int maxValue;

    //Constructors
    public Dynamic()
    {
    }

    // Methods

    /**
     * finds the solution to the 0/1 knapsack problem
     *
     * Pre-condition:  Array has a length > 0, the integers in the price
and weight array are positive, the
     *                  prices array has the same number of items as the
weights array, and the weights are integers
     *
```

```
     * Post condition: The return value is the max value with the capacity
allotted.
     **/
    public static int dynamic(int capacity, int[] prices, int[] weight, int
numItems) {
        maxValue = 0;
        //checking preconditions
        assert (numItems > 0);
        assert (prices.length == weight.length);
        for (int i = 0; i < numItems; i++){
            assert (prices[i] >= 0);
            assert (weight[i] >= 0);
        }


        int[][] f = new int[numItems + 1][capacity + 1];
        boolean[][] knapsack = new boolean[numItems + 1][capacity + 1];

        //Build table k[][] in bottom up manner
        for (int i = 1; i <= numItems; i++) {
            for (int w = 1; w <= capacity; w++) {
                //don't take the item
                int f1 = f[i - 1][w];

                //take it
                int f2 = Integer.MIN_VALUE;
                if (weight[i] <= w) {
                    f2 = prices[i] + f[i - 1][w - weight[i]];
                }
                    //select the better of two options
                    f[i][w] = Math.max(f1, f2);
                    knapsack[i][w] = (f2 > f1);

            }
        }

            //determine which items to take
            boolean[] take = new boolean[numItems + 1];
            for (int n = numItems, w = capacity; n > 0; n--) {
                if (knapsack[n][w]) {
                    take[n] = true;
```

```
                    w = w- weight[n];
                } else
                    take[n] = false;
            }

//            //print results
//            System.out.println("item" + "\t" + "profit" + "\t" + "weight"
+ "\t" + "take");
//            for (int n = 1; n < numItems+1; n++)
//                System.out.println(n + "\t" + prices[n] + "\t" +
weight[n] + "\t" + take[n]);

            //finds the max value of the most optimal knapsack the thief
can fill
            for(int i = 0; i < take.length; i++){
                if(take[i]) {
                    maxValue += prices[i];
                }
            }
            return maxValue;
        }


    //used to time how long it takes to find the solution using dynamic
programming
    //returns the value of the knapsack stolen
    public long TimeToFind(int capacity, int[] prices, int[] weight, int
numItems){
        long start = System.nanoTime();
        int value = dynamic(capacity, prices, weight, numItems);
        long end = System.nanoTime();
        long duration =  end- start;
        System.out.println("Value for dynamic programming: " + value);
        return duration;
    }

    public String[] dynamicSolTime(int capacity, int[] prices, int[]
weight, int numItems){
        String[] solTime = new String[2];
        long start = System.nanoTime();
        int value = dynamic(capacity, prices, weight, numItems);
```

```
        long end = System.nanoTime();
        long duration =  end- start;
        //System.out.println("Value for dynamic programming: " + value);
        solTime[0] = Integer.toString(value);
        solTime[1] = Long.toString(duration);
        return solTime;
    }

}
```

7.4 Greedy approximation algorithm

Author of the code to the brute force algorithm: Melany Diaz.

The code was modified by the author of the essay.

```
package com.company;
/*
   Class: Greedy

   Author: Melany Diaz
   with assistance from: Gerry Howser

   Creation date:  3/5/2016

   Modifications:
        Date        Name        reason
        3/10/2016   Melany Diaz Debugging
*/

import java.lang.reflect.Array;
import java.util.ArrayList;
import java.util.Arrays;

/**
 This class will implement the Greedy solution: Form a price density array,
sort it in non-ascending order,
 and use a greedy strategy to fit the greatest value into the knapsack.
```

```
 */
public class Greedy {

    //instance variables
    private static int capacity;
    private static int[] prices;
    private static int[] weight;
    private static int numItems;

    private static int maxValue;
    private static int currentWeight;

    //rearranged weight array
    public static int[] orderedWeights;


    //Constructors
    public Greedy(int capacity, int[] prices, int[] weight, int numItems)
    {
        this.capacity = capacity;
        this.numItems = numItems;
        this.prices = prices;
        this.weight = weight;

        this.orderedWeights = new int[numItems];
        maxValue = 0;
        currentWeight = 0;

    }

    public Greedy() {
    }

    // Methods

    /**
     * finds the solution to the 0/1 knapsack problem
     *
     * Pre-condition:  Array has a length > 0, the integers in the price
and weight array are positive, and the
     *                  prices array has the same number of items as the
```

```
weights array
     *
     * Post condition: The return value is the max value with the capacity
allotted.
     **/
    public static int greedy(int capacity, int[] prices, int[] weight, int
numItems) {
        maxValue = 0;
        //checking preconditions
        assert (numItems > 0);
        assert (prices.length == weight.length);
        for (int i = 0; i < numItems; i++){
            assert (prices[i] >= 0);
            assert (weight[i] >= 0);
        }


        //sort price list in non-ascending order
        Heapsort h = new Heapsort();
        int[] OGPrices = Arrays.copyOf(prices, numItems);

        h.sort(prices);
        //System.out.println(Arrays.toString(prices));

        //rearrange weight list accordingly
        orderedWeights = newWeights(prices, OGPrices, weight, numItems);
        //System.out.println(Arrays.toString(orderedWeights));

        //fill the knapsack using the greedy idea
        currentWeight = 0;
        int w =0;
        while(currentWeight <= capacity && w < numItems){
            if(orderedWeights[w] <= (capacity - currentWeight)) {
//                    System.out.print(prices[w] + " ");
                currentWeight += orderedWeights[w];
                maxValue += prices[w];
                //System.out.println("MV:  " + maxValue + "  n: " +
numItems + "  cap:" + capacity + "   currentWeight:"  + currentWeight);
            }
            w++;
        }
```

```
        return maxValue;
    }

    //used to rearrange weight array to match new, ordered, price array
    public static int[] newWeights(int[] newPrices, int[] OGPrices, int[]
weight, int numItems){
        for(int j = 0; j < numItems; j++){
            int i = 0;
            boolean found = false;
            while (!found && i <numItems){
                if(newPrices[j] == OGPrices[i]){
                    OGPrices [i] = Integer.MAX_VALUE;
                    orderedWeights[j] = weight[i];
                    found = true;
                }
                i++;
            }
        }
        return orderedWeights;
    }


    //used to time how long it takes to find the solution using greedy
algorithm
    //returns the value of the knapscack stolen
    public long TimeToFind(int capacity, int[] prices, int[] weight, int
numItems){
        long start = System.nanoTime();
        int value = greedy(capacity, prices, weight, numItems);
        long end = System.nanoTime();
        long duration =  end- start;
        System.out.println("Value for greedy algorithm: " + value);
        return duration;
    }

    public String[] greedySolTime(int capacity, int[] prices, int[] weight,
int numItems){
        String[] solTime = new String[2];
        long start = System.nanoTime();
        int value = greedy(capacity, prices, weight, numItems);
        long end = System.nanoTime();
```

```
        long duration =  end- start;
        //System.out.println("Value for greedy algorithm: " + value);
        solTime[0] = Integer.toString(value);
        solTime[1] = Long.toString(duration);
        return solTime;
    }

}
```

7.5 Branch and bound algorithm

Author of the code to the brute force algorithm: Patrick Herrmann.
The code was modified by the author of the essay.

ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

```
package com.company;

/* ============================================
 * jORLib : a free Java OR library
 * ============================================
 *
 * Project Info:  https://github.com/jkinable/jorlib
 * Project Creator:  Joris Kinable (https://github.com/jkinable)
 *
 * (C) Copyright 2015, by Joris Kinable and Contributors.
 *
 * This program and the accompanying materials are licensed under LGPLv2.1
 *
 */
/* -----------------
 * Knapsack.java
 * -----------------
 * (C) Copyright 2015, by Joris Kinable and Contributors.
 *
 * Original Author:  Joris Kinable
 * Contributor(s):   -
 *
 * $Id$
 *
 * Changes
 * -------
 *
 */
        import java.io.*;
        import java.util.*;

/**
 * Memory efficient Branch and bound implementation of knapsack.
 *
 * Solves the problem:<br>
```

```
 * {@code max \sum_i c_i x_i}<br>`
 * {@code s.t. \sum_i a_i x_i <= b}<br>
 * {@code x_i binary}<br>
 *
 * The implementation is memory efficient: it does not rely on large
matrices.
 * The knapsack problem is solved as a binary tree problem. Each level of
the tree corresponds to a specific item. Each time we branch on a
particular node, two child-nodes
 * are created, reflecting whether the item at the level of the child nodes
is selected, or not. As a result, at most 2^n nodes, where n is the number
of items, are created.
 * In practice, the number of generated nodes is significantly smaller, as
the number of items one can choose depends on the knapsack weight.
Furthermore, the search tree is pruned using
 * bounds.<p>
 * Consider replacing this implementation by a faster one such as
MT2<br><br>
 *
 * NOTE: All item weights, as well as the maxKnapsackWeight have to be
integers. The item weights can be fractional, both positive and negative.
Obviously, since this is a
 * maximization problem, items with a value smaller or equal to 0 are never
selected.
 *
 *
 * @author Joris Kinable
 * @since April 8, 2015
 */

public class Branch_and_Bound  {      //implements KnapsackAlgorithm

    //Define the knapsack parameters
    private int nrItems; //number of items in the knapsack
    private int maxKnapsackWeight; //max allowed wait of the knapsack
    private double[] itemValues; //Values of the knapsack items
    private int[] itemWeights; //Weights of the knapsack items

    //Solution
    private double knapsackValue=0;
    private int knapsackWeight=0;
```

```
    private boolean[] knapsackItems;

    /**
     * Calculates a greedy solution for the knapsack problem. This solution
is a valid lower bound and is used for pruning.
     * @param itemOrder Order in which the items are considered by the
greedy algorithm. The items are sorted ascending, based on their
value/weight ratio
     */

    private void getGreedyKnapsackSolution(Integer[] itemOrder){
        double value=0;
        int remainingWeight=maxKnapsackWeight;
        boolean[] selectedItems=new boolean[nrItems];
        //Greedily take a single item until the knapsack is full
        for(int i=0; i<nrItems; i++){
            if(itemWeights[itemOrder[i]]<=remainingWeight){
                value+=itemValues[itemOrder[i]];
                remainingWeight-=itemWeights[itemOrder[i]];
                selectedItems[itemOrder[i]]=true;
            }
        }
        this.knapsackValue=value;
        this.knapsackWeight=maxKnapsackWeight-remainingWeight;
        this.knapsackItems=selectedItems;
    }


    /**
     * Solve the knapsack problem.
     * //@param nrItems nr of items in the knapsack
     * //@param maxKnapsackWeight max size/weight of the knapsack
     * //@param itemValues item values
     * //@param itemWeights item weights
     * //@return The value of the knapsack solution
     */

    public static void main(String[] args) throws FileNotFoundException {

        String inputFileBase = "./inputsHigherMore/inputs_dataset#";
        String inputFile, timesFile =
```

```
"./outputs/VH4/times_BB_output_file_VS.txt", outputFile =
"./outputs/VH4/BB_output_file_VS.txt";
        String timeBBFile = "./outputs/VH4/mid_times_BB.txt";
        int numItems, value, capacity, c_coef = 30, all, repetitions = 10,
step = 10;
        File file;
        //int numberOfItems[] = {10, 20, 30, 40, 50, 60, 70, 80, 90, 100,
200, 300};
        int numberOfItems[] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13,
14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29};
        double solutions[] = new double[repetitions];
        long times[] = new long[repetitions];

        long start, end, startHelp, endHelp;
        double duration = 0;

        //numItems = numberOfItems[w];
        double[] prices; double[] pricesN; int[] weight; int[] weightN;

        //clear output file
        PrintWriter pw1 = new PrintWriter(outputFile); PrintWriter pw2 =
new PrintWriter(timesFile);
        PrintWriter pw3 = new PrintWriter(timeBBFile);
        pw1.close(); pw2.close(); pw3.close();

        //for (int w = 0; w < numberOfItems.length; w++) { //for cycle
start
        for (int w = 0; w <= 100; w++) {
            //numItems = numberOfItems[w];
            numItems = w*step;        //w*10
            capacity = numItems * c_coef;
            all = numItems * repetitions;
            prices = new double[all];
            weight = new int[all];

            //inputFile = inputFileBase + Integer.toString(w + 1) + ".txt";
// w+1
            inputFile = inputFileBase + Integer.toString(((w+1) % 10)) +
".txt";       // w+1
            file = new File(inputFile);
            Scanner sc = new Scanner(file);
```

```
            for (int j = 0; j < all && sc.hasNextLine(); j++) {
                prices[j] = Double.parseDouble(sc.nextLine());
                weight[j] = Integer.parseInt(sc.nextLine());
                //System.out.println("price:  " + prices[j] + "  weight: "
+ weight[j]);
            }
            sc.close();

            start = System.nanoTime();
            for (int i = 0; i < repetitions; i++) {
                pricesN = Arrays.copyOfRange(prices, i * numItems, (i + 1)
* numItems);
                weightN = Arrays.copyOfRange(weight, i * numItems, (i + 1)
* numItems);
                //run the algorithm
                startHelp = System.nanoTime();
                Branch_and_Bound solver = new Branch_and_Bound();
                solutions[i] = solver.solveKnapsackProblem(numItems,
capacity, pricesN, weightN);
                times[i] = System.nanoTime() - startHelp;

            }
            end = System.nanoTime();
            duration = (end - start) / (double) repetitions;

            //write the solutions and durations into the output file
            writeToFileTime(Double.toString(duration), numItems);
            writeMidTimes(numItems, times, repetitions, timeBBFile);
            writeToFileSols(solutions, w*10, repetitions);
//writeToFileSols(solutions, numberOfItems[w], repetitions);

        }

    }

    public static void writeToFileTime(String time, int n) {
        try {
            FileWriter out = new FileWriter(new
File("./outputs/VH4/times_BB_output_file_VS.txt"), true);
            //out.write("n:" + n + " ");
            out.write(time + "\n");
```

```
            out.close();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }

    public static void writeMidTimes(int n, long[] array, int repetitions,
String fileName) {
        long sum = 0;
        try {
            FileWriter out = new FileWriter(new File(fileName), true);
            out.write("n:" + n + " ");
            for(int i = 0; i < repetitions; i++) {
                out.write(Long.toString(array[i]) + " ");
                sum += array[i];
            }
            out.write("sum: " + Long.toString(sum) + "\n");
            out.close();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }

    public static void writeToFileSols(double sols[], int w, int
repetitions) {
        try {
            FileWriter out = new FileWriter(new
File("./outputs/VH4/BB_output_file_VS.txt"), true);
            out.write( w + " ");
            for (int i = 0; i < repetitions; i++) {
                out.write(Double.toString(sols[i]) + " ");
            }
            out.write("\n");
            out.close();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }

    public double solveKnapsackProblem(int nrItems, int maxKnapsackWeight,
double[] itemValues, int[] itemWeights){
```

```
        //Initialize
        this.nrItems=nrItems;
        this.maxKnapsackWeight=maxKnapsackWeight;
        this.itemValues=itemValues;
        this.itemWeights=itemWeights;

        this.knapsackValue=0;
        this.knapsackWeight=0;

        Queue<KnapsackNode> queue=new PriorityQueue<>();

        //Define the order in which items will be processed. The items are
sorted based on their value/weight ratio, thereby considering
proportionally more valuable items first.
        Integer[] itemOrder=new Integer[nrItems];
        for(int i=0; i<nrItems; i++) itemOrder[i]=i;
        //Sort the times in ascending order, based on their value/weigth
ratio
        Arrays.sort(itemOrder, new Comparator<Integer>() {
            @Override
            public int compare(Integer item1, Integer item2) {
                return -
1*Double.compare(itemValues[item1]/itemWeights[item1],
itemValues[item2]/itemWeights[item2]);
            }
        });


        //Create initial node
        KnapsackNode kn=new KnapsackNode(nrItems);
        kn.bound=calcBound(itemOrder, kn.level+1, maxKnapsackWeight-
kn.weight, kn.value);
        queue.add(kn);

        //Get initial greedy solution
        this.getGreedyKnapsackSolution(itemOrder);

        //Maintain a reference to the best node
        double bestValue=this.knapsackValue;
        KnapsackNode bestNode=null;
```

```
        while(!queue.isEmpty()){
            kn=queue.poll();
            if(kn.bound>bestValue && kn.level<nrItems-1){
                kn.level++;
                //Create 2 new nodes, one where item <itemToAdd> is used,
and one where item <itemToAdd> is skipped.
                int itemToAdd=itemOrder[kn.level];
                if(kn.weight+itemWeights[itemToAdd]<=maxKnapsackWeight &&
itemValues[itemToAdd]>0){ //Check whether we can add the next item and
whether its value is positive
                    KnapsackNode knCopy=kn.copy();
                    knCopy.addItem(itemToAdd, itemWeights[itemToAdd],
itemValues[itemToAdd]);
                    knCopy.bound=calcBound(itemOrder, knCopy.level+1,
maxKnapsackWeight-knCopy.weight, knCopy.value);

                    if(knCopy.value>bestValue){
                        bestValue=knCopy.value;
                        bestNode=knCopy;
                    }
                    queue.add(knCopy);
                }
                //Dont use item[kn.level]
                kn.bound=calcBound(itemOrder, kn.level+1,
maxKnapsackWeight-kn.weight, kn.value);
                queue.add(kn);
            }
        }

        //Update the results based on the best solution found
        if(bestNode!=null){
            this.knapsackValue=bestNode.value;
            this.knapsackWeight=bestNode.weight;
            this.knapsackItems=bestNode.selectedItems;
        }

        return this.knapsackValue;
    }

    /**
     * Calculate a bound on the best solution attainable for a given
```

```
partial solution.
     * @return bound on the best value attainable  by the partially filled
knapsack
     */
    private double calcBound(Integer[] itemOrder, int level, int
remainingSize, double value){
        double bound=value;
        while(level<nrItems && remainingSize-
itemWeights[itemOrder[level]]>=0){
            remainingSize-=itemWeights[itemOrder[level]];
            bound+=itemValues[itemOrder[level]];
            level++;
        }
        if(level<nrItems){

bound+=itemValues[itemOrder[level]]*(remainingSize/(double)itemWeights[item
Order[level]]);
        }
        return bound;
    }

    /**
     * Get the value of the knapsack
     * @return Get the value of the knapsack
     */
    public double getKnapsackValue(){
        return knapsackValue;
    }
    /**
     * Get the total weight of the knapsack
     * @return Get the total weight of the knapsack
     */
    public int getKnapsackWeight(){
        return knapsackWeight;
    }
    /**
     * Get the items in the knapsack
     * @return Get the items in the knapsack
     */
    public boolean[] getKnapsackItems(){
        return knapsackItems;
```

```
    }

    /**
     * Knapsack nodes represent partial solutions for the knapsack problem.
A subset of the variables, starting from the root node of the tree up to
<level> level,
     * have been fixed.
     * @author jkinable
     *
     */
    private final class KnapsackNode implements Comparable<KnapsackNode>{
        public final int nrItems; //Max number of items in the problem
        public final boolean[] selectedItems; //Selected items
        public int level; //Depth of the knapsack node in the search tree;
Each level of the search tree corresponds with a single item.
        public double bound; //Bound on the optimum value atainable by this
node
        public double value; //Total value of the items in this knapsack
        public int weight; //Total weight of the items in this knapsack

        public KnapsackNode(int nrItems){
            this.nrItems=nrItems;
            selectedItems=new boolean[nrItems];
            this.bound=0;
            this.value=0;
            this.weight=0;
            this.level=-1;
        }

        public KnapsackNode(int nrItems, int level, double bound, double
value, int weight, boolean[] selectedItems) {
            this.nrItems=nrItems;
            this.level=level;
            this.bound=bound;
            this.value=value;
            this.weight=weight;
            this.selectedItems=selectedItems;
        }

        public KnapsackNode copy(){
            boolean[] selectedItemsCopy=new boolean[nrItems];
```

```
            System.arraycopy(selectedItems, 0, selectedItemsCopy, 0,
nrItems);
            return new KnapsackNode(nrItems, this.level, this.bound,
this.value, this.weight, selectedItemsCopy);
        }

        public void addItem(int itemID, int itemWeight, double itemValue){
            selectedItems[itemID]=true;
            weight+=itemWeight;
            value+=itemValue;
        }

        @Override
        public int compareTo(KnapsackNode otherNode) {
            if(this.bound==otherNode.bound)
                return 0;
            else if(this.bound>otherNode.bound)
                return -1;
            else
                return 1;
        }

        public String toString(){
            return "Level: "+level+" bound: "+bound+" value: "+value+"
weight: "+weight+" items: "+Arrays.toString(selectedItems)+" \n";
        }
    }
}
```

7.6 Dataset generator

```
package com.company;
import java.io.File;
import java.io.FileNotFoundException;
import java.io.FileWriter;
import java.io.IOException;
import java.util.Random;

public class Input {
```

```
    public static void main(String[] args) {

        Random generator = new Random();
        int max_nb_items = 40100;
        int value, weight;
        int min = 1, max = 100;
        String filename;
        int numberOfDatasets = 50;

        for (int j = 0; j <= numberOfDatasets; j++) {
            filename = "./inputsHigherMore/inputs_dataset#" +
Integer.toString(j) + ".txt";
            try {
                FileWriter out = new FileWriter(new File(filename));
                for (int i = 0; i < max_nb_items; i++) {
                    value = generator.nextInt(max) + min;
                    weight = generator.nextInt(max) + min;
                    System.out.println(i + ". prices: " + value + "
weight: " + weight);
                    out.write(Integer.toString(value) + "\n" +
Integer.toString(weight) + "\n");
                }
                out.close();
            } catch (FileNotFoundException e) {
                System.out.println("Problem with input file");
            } catch (IOException e) {
                e.printStackTrace();
            }
        }

    }

}
```

7.7 Writing outputs to text files

Author of the code to the brute force algorithm: Melany Diaz

The code was modified by the author of the essay.

```
package com.company;

import java.io.*;
import java.util.Arrays;
import java.util.Scanner;

public class Knapsack_updated {

    public static void main(String[] args) throws FileNotFoundException {

        //the number of items for the robber to take
        int numItems = 10, maxItems = 400, repetitions = 10, step = 1;
//10100
        String inputFile, inputFileBase =
"./inputsHigherMore/inputs_dataset#";
        String timesFile = "./outputs/VH1/times_output_file_VS.txt",
solFile = "./outputs/VH1/outputDataset_VS.txt";
        String timeDPFile = "./outputs/VH1/mid_times_DP.txt", timeGFile =
"./outputs/VH1/mid_times_Greedy.txt", timeBFFile =
"./outputs/VH1/mid_times_BF.txt";

        int time, value, all = 0;
        int solutions[] = new int[repetitions];
        long times[] = new long[repetitions];
        File file;
        long start, end, startHelp, endHelp;
        double duration = 0;

        int[] pricesbf; int[] weightbf; int[] pricesg2; int[] weightg2;
int[] pricesd2; int[] weightd2;
        int[] prices = new int[maxItems]; int[] weight = new int[maxItems];
        //BruteForce bf = new BruteForce();
        Dynamic dp = new Dynamic();

        //capacity of the Knapsack (to remain constant)
        int c_coef = 30;
        int capacity = numItems * c_coef;

        //clear output file
        PrintWriter pw1 = new PrintWriter(solFile);
```

```
        PrintWriter pw2 = new PrintWriter(timesFile);
        PrintWriter pw3 = new PrintWriter(timeDPFile); PrintWriter pw4 =
new PrintWriter(timeBFFile); PrintWriter pw5 = new PrintWriter(timeGFile);
        pw1.close(); pw2.close(); pw3.close(); pw4.close(); pw5.close();


        for (int w = 0; w < numberOfItems.length; w++) { //for cycle start
        //for (int w = 0; w <= 100; w++) { //for cycle start
            System.out.println(w);
            numItems = w*step;        //w*10
            capacity = numItems * c_coef;
            all = numItems * repetitions;
            inputFile = inputFileBase + Integer.toString(((w+1) % 10)) +
".txt";        // w+1
            //inputFile = inputFileBase + Integer.toString(w + 1) + ".txt";
// w+1
            file = new File(inputFile);
            Scanner sc = new Scanner(file);
            for (int j = 0; j < all && sc.hasNextLine(); j++) {
                prices[j] = Integer.parseInt(sc.nextLine());
                weight[j] = Integer.parseInt(sc.nextLine());
                //System.out.println("price:  " + prices[j] + "  weight: "
+ weight[j]);
            }
            sc.close();

            pricesbf = new int[numItems]; weightbf = new int[numItems];
pricesg2 = new int[numItems]; weightg2 = new int[numItems];
            pricesd2 = new int[numItems + 1]; weightd2 = new int[numItems +
1]; pricesd2[0] = 0; weightd2[0] = 0;
            for (int s = 0; s < 3; s++) {
                switch (s) {
                    case 0:  //  BRUTE FORCE
                        if (numItems < 36) {
                            start = System.nanoTime();
                            for (int i = 0; i < repetitions; i++) {
                                pricesbf = Arrays.copyOfRange(prices, i *
numItems, (i + 1) * numItems);
                                weightbf = Arrays.copyOfRange(weight, i *
numItems, (i + 1) * numItems);
                                //run the algorithm
```

```
                                startHelp = System.nanoTime();
                                BruteForce bf = new BruteForce();
                                solutions[i] =
bf.bruteForceSolTime2(capacity, pricesbf, weightbf, numItems);
                                times[i] = System.nanoTime() - startHelp;
                            }
                            end = System.nanoTime();
                            duration = (end - start) / (double)
repetitions;
                        }
                        break;
                    case 1:  //  GREEDY ALGORITHM
                    start = System.nanoTime();
                    for(int i = 0; i < repetitions; i++) {
                        // fill the price and weight arrays
                        pricesg2 = Arrays.copyOfRange(prices, i * numItems,
(i + 1) * numItems);
                        weightg2 = Arrays.copyOfRange(weight, i * numItems,
(i + 1) * numItems);
                        //run the algorithm
                        startHelp = System.nanoTime();
                        Greedy greedy2 = new Greedy(capacity, pricesg2,
weightg2, numItems);
                        solutions[i] = greedy2.greedySolTime2(capacity,
pricesg2, weightg2, numItems);
                        times[i] = System.nanoTime() - startHelp;
                        //System.out.println("GA sol: " + solutions[i] +
"\n\n");
                    }
                    end = System.nanoTime();
                    duration =  (end - start) / (double) repetitions;
                        break;
                    case 2:  //  DYNAMIC PROGRAMMING
                        start = System.nanoTime();
                        for (int i = 0; i < repetitions; i++) {
                            //price array for dynamic, must be one relevant
with the 0 index equal to 0
                            pricesbf = Arrays.copyOfRange(prices, i *
numItems, (i + 1) * numItems);
                            weightbf = Arrays.copyOfRange(weight, i *
numItems, (i + 1) * numItems);
```

```
                            startHelp = System.nanoTime();
                            for (int j = 1; j < numItems + 1; j++) {
                                pricesd2[j] = pricesbf[j - 1];
                                weightd2[j] = weightbf[j - 1];
                            }
                            //run the algorithm
                            solutions[i] = dp.dynamicSolTime2(capacity,
pricesd2, weightd2, numItems);
                            times[i] = System.nanoTime() - startHelp;
                        }
                        end = System.nanoTime();
                        duration = (end - start) / (double) repetitions;
                        break;
                    case 3:
                        break;
                }

                switch(s) {
                    case(0):
                        if (numItems < 36 ) {
                        writeToFileTime(Double.toString(duration), s,
numItems, timesFile);
                        //writeToFileSols(solutions, s, numItems, solFile);
                        //writeMidTimes(numItems, times, repetitions,
timeBFFile);
                        }
                        break;
                    case (1):
                        writeToFileTime(Double.toString(duration), s,
numItems, timesFile);
                        //writeToFileSols(solutions, s, numItems, solFile);
                        //writeMidTimes(numItems, times, repetitions,
timeGFile);

                        break;
                    case(2):
                        writeToFileTime(Double.toString(duration), s,
numItems, timesFile);
                        //writeToFileSols(solutions, s, numItems, solFile);
                        //writeMidTimes(numItems, times, repetitions,
timeDPFile);
```

```
                    break;

                }
            }
        }
    }

    public int solveBruteForce(int n, int i, int[] prices, int[] weight) {
        int sol = 0;
        // fill the price and weight arrays
        int[] pricesbf = Arrays.copyOfRange(prices, i*10, (i+1)*10);
        int[] weightbf = Arrays.copyOfRange(weight, i*10, (i+1)*10);

        //run the algorithm
        BruteForce bf = new BruteForce();
        sol = bf.bruteForceSolTime2(100, pricesbf, weightbf, n);
        //System.out.println("sol: " + sol + "\n");
        return sol;
    }

    public static void writeToFileTime(String time, int s, int n, String
fileName) {
        try {
            FileWriter out = new FileWriter(new File(fileName), true);
            if (s == 0) {
                out.write("n:" + n + " ");
            }
            out.write(time + " ");
            if (s == 2) {
                out.write("\n");
            }
            out.close();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }

    public static void writeMidTimes(int n, long[] array, int repetitions,
String fileName) {
        long sum = 0;
        try {
```

```
            FileWriter out = new FileWriter(new File(fileName), true);
            out.write("n:" + n + " ");
            for(int i = 0; i < repetitions; i++) {
                out.write(Long.toString(array[i]) + " ");
                sum += array[i];
            }
            out.write("sum: " + Long.toString(sum) + "\n");
            out.close();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }

    public static void writeToFileSols(int sols[], int s, int w, String
fileName) {
        try {
            FileWriter out = new FileWriter(new File(fileName), true);
            out.write(Integer.toString(w)+ "    ");
            for (int i = 0; i < 10; i++) {
                out.write(Integer.toString(sols[i]) + " ");
            }
            out.write("\n");
            out.close();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }

}
```