Comparison of Effectiveness of Algorithms Solving Knapsack Problem

Mateusz Tabaszewski

Poznań, May 2022

Introduction and Methods of Research

This project is meant to compare the time and effectiveness of different algorithms used to solve the knapsack problem. The aforementioned algorithms include Brute Force(BF), Greedy Algorithm, and Dynamic Programming(DP). Program used for collecting the data was written in C++ language and time was measured using GetProcessTimes(), GetCurrentProcess() functions included via <processthreadapi.h> header. Items to be assigned to the knapsack were created with random "weight" and "value" parameters except for data shown in Figure 9 and Figure 10, where data was created artificially to show the types of cases, for which the Greedy Algorithm may make mistakes. Experiments have also been conducted with different values of both n-number of possible items to choose from, and different values of b-capacity of the knapsack, i. e. how much weight it can store. Most of the experiments were performed twice for small n values(up to 30 or 35 depending on the experiment) due to the time-consuming nature of BF, for the second run n was equal to up to 1200 or 2000(also depending on the experiment), however, only BF and Greedy were tested in such cases.

Brute Force Algorithm was implemented using binary representation, which means each item was represented using either 0 or 1 value and the set of all possible items was represented using a value consisting of 0s and 1s and to test all permutations, a binary value of 1 was added. Of course, only sets of items whose sum did not exceed the maximum possible weight of the knapsack were accepted as a solution.

A Greedy Algorithm requires the knowledge of the proportion of value to weight and a sorting algorithm. For these purposes Quick Sort was chosen mostly because barring certain outlier cases it is the fastest sorting algorithm out of the ones tested during previous experiments(aside from Counting Sort, however that algorithm would not be suitable due to double values being used to represent the ratio of value and weight of each item). Quick Sort is also relatively easy to implement for purposes of creating the Greedy Algorithm.

Dynamic Programming was implemented with n(number of items) and b (capacity of the knapsack**) representing rows and columns of the table used for calculations.

Experiments were performed for b dependent on total weight of the items generated for given n, as well as for b changing within a certain range, to create 3D plots.

**There are times in this paper, that b is referred to as knapsack's capacity, etc. and on other times it is called max weight, etc. it of course refers to the same thing, i.e. what the total weight of items inside a knapsack is acceptable.

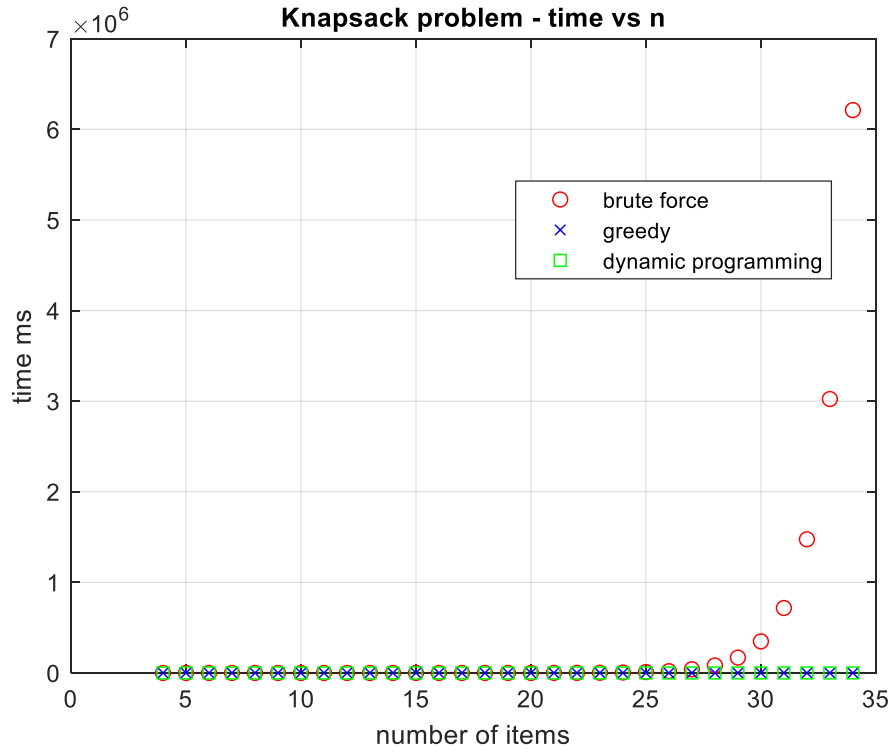

Fig.1. Time comparison between BF, Greedy and DP for constant b(knapsack's capacity).

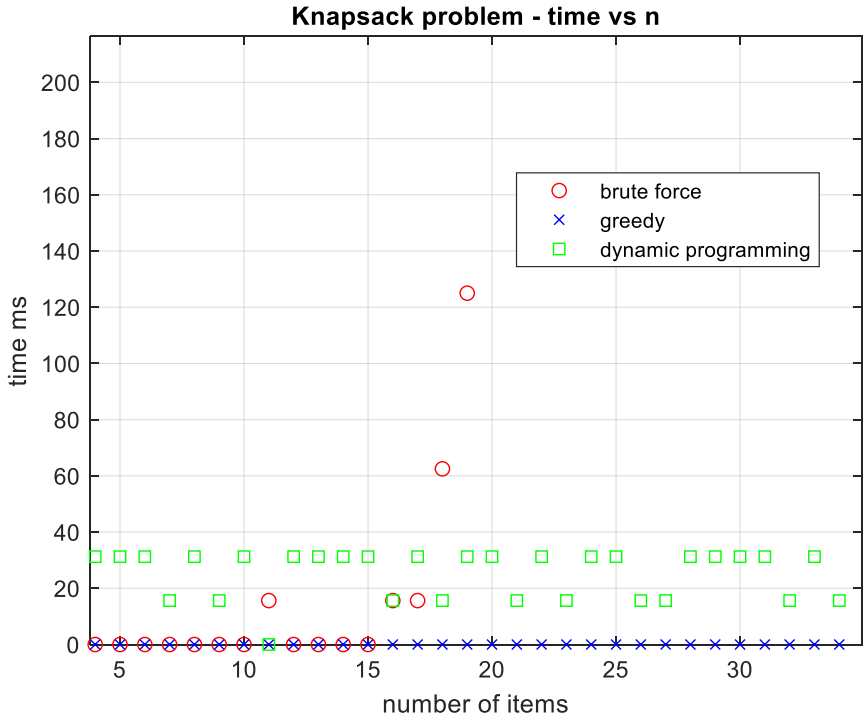

Fig.2. Time comparison between BF, Greedy and DP for constant b(knapsack's capacity) with Y-axis cut-off point of ~200ms.

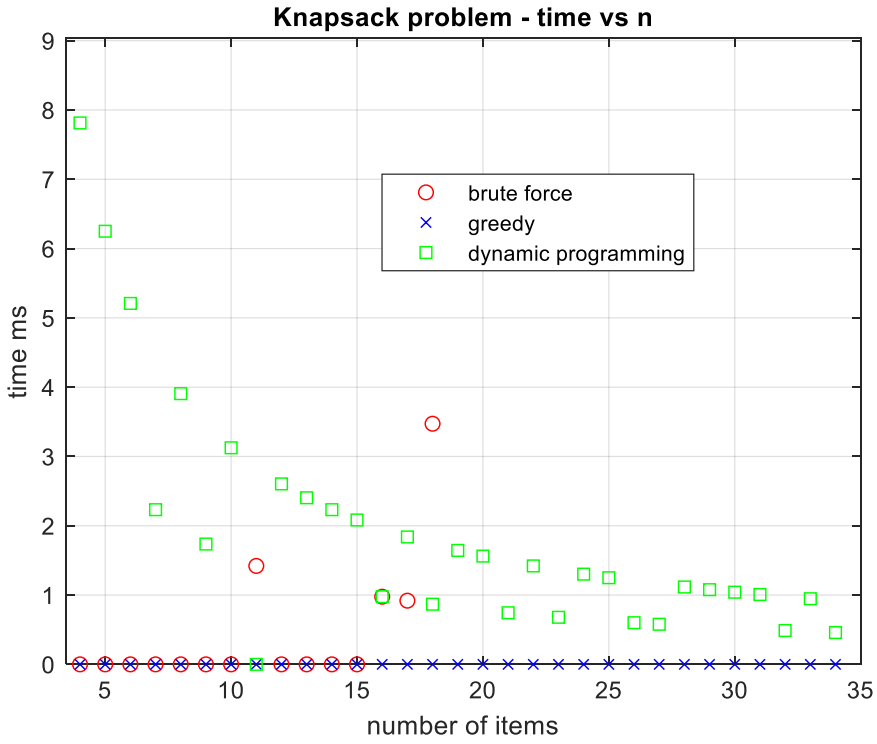

Fig. 3. Time comparison between BF, Greedy and DP for a single item(results of Figure 2 divided by n) with constant b(knapsack's capacity) with Y-axis cut-off point of 9ms.

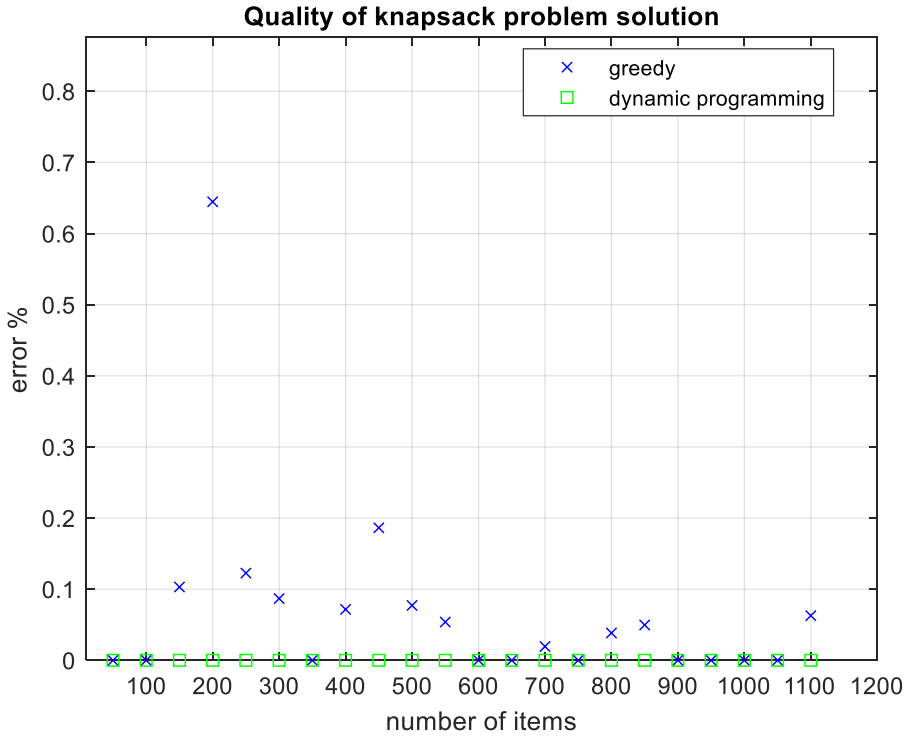

Fig. 4(a). Percentage of Error(how far the solution is from the optimum) for DP and Greedy versus number of items for randomly generated data.

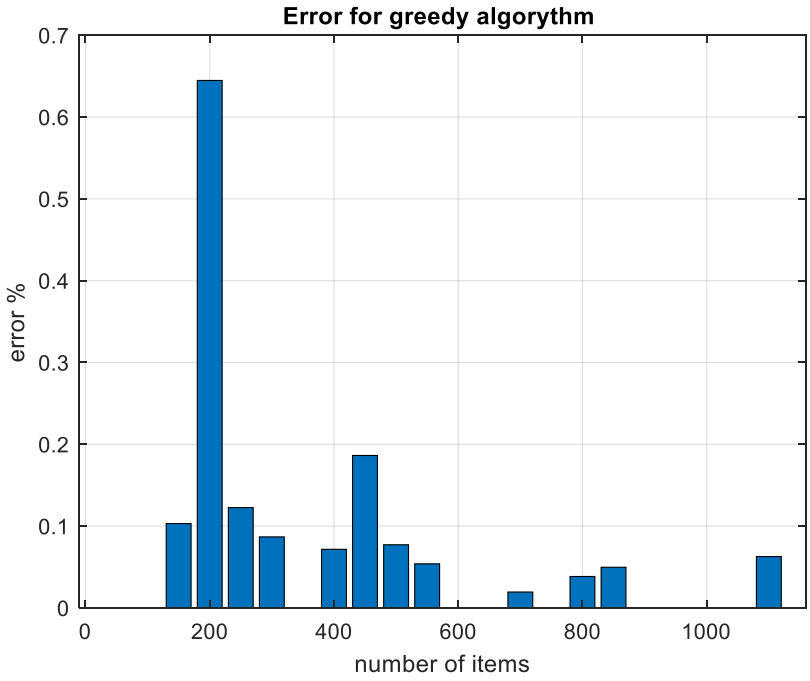

Fig. 4(b). Percentage of error per number of items for randomly generated data- bar chart form-Greedy Algorithm.

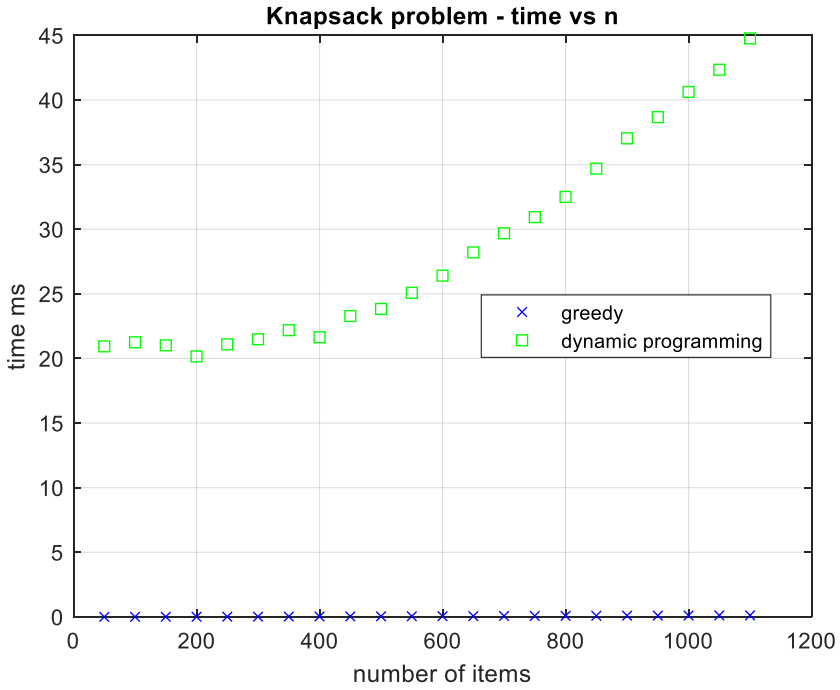

Fig. 5. Time comparison between Greedy and DP for constant b(knapsack's capacity) with larger n.

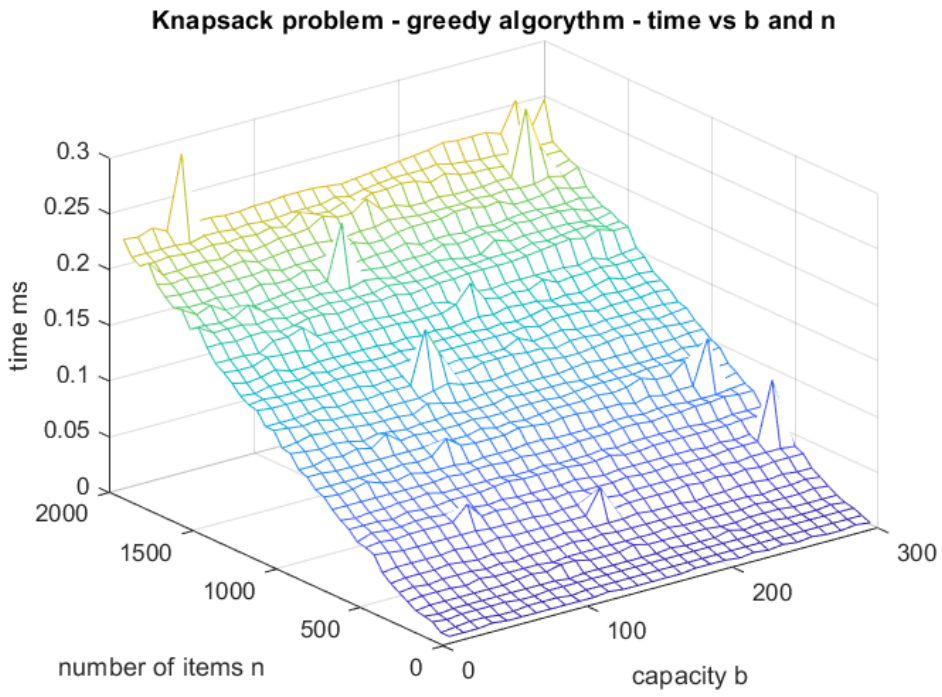

Fig. 6. Time of finding a solution for Greedy Algorithm based on number of items n and capacity b.

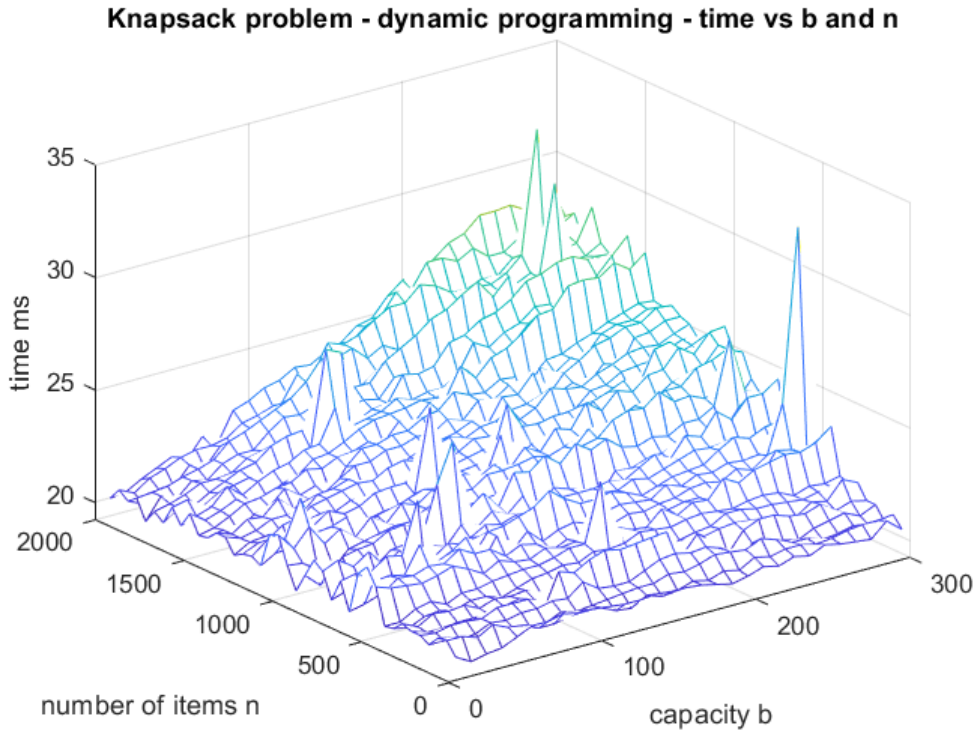

Fig. 7. Time of finding a solution for Dynamic Programming based on number of items n and capacity b.

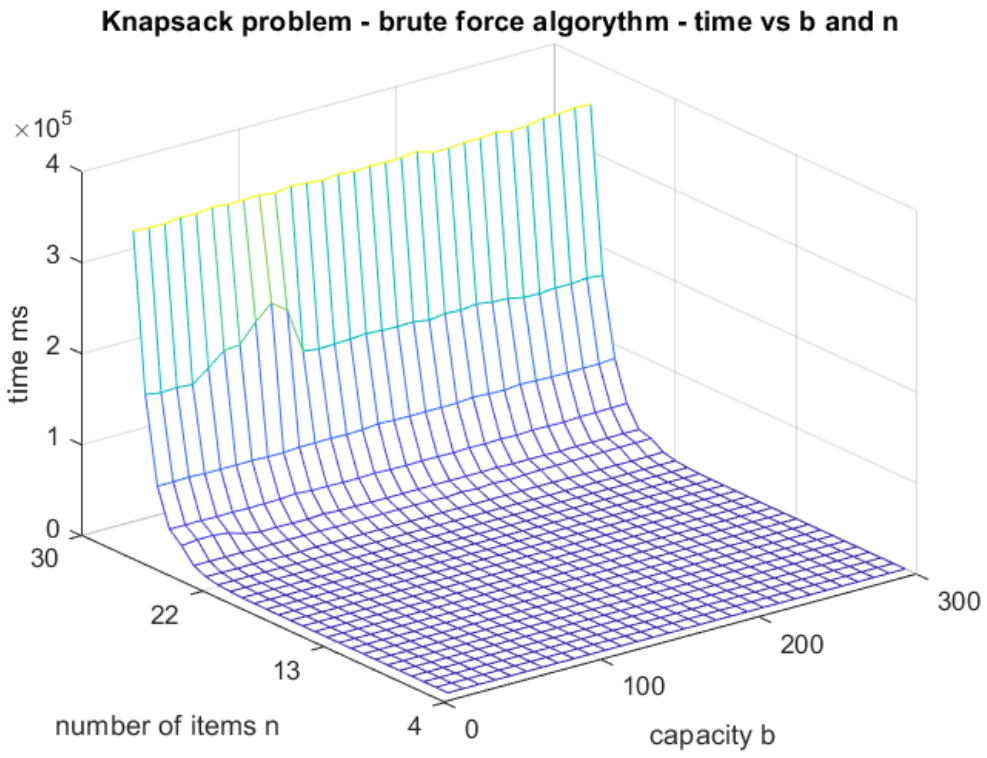

Fig. 8. Time of finding a solution for Brute Force Algorithm based on number of items n and capacity b.

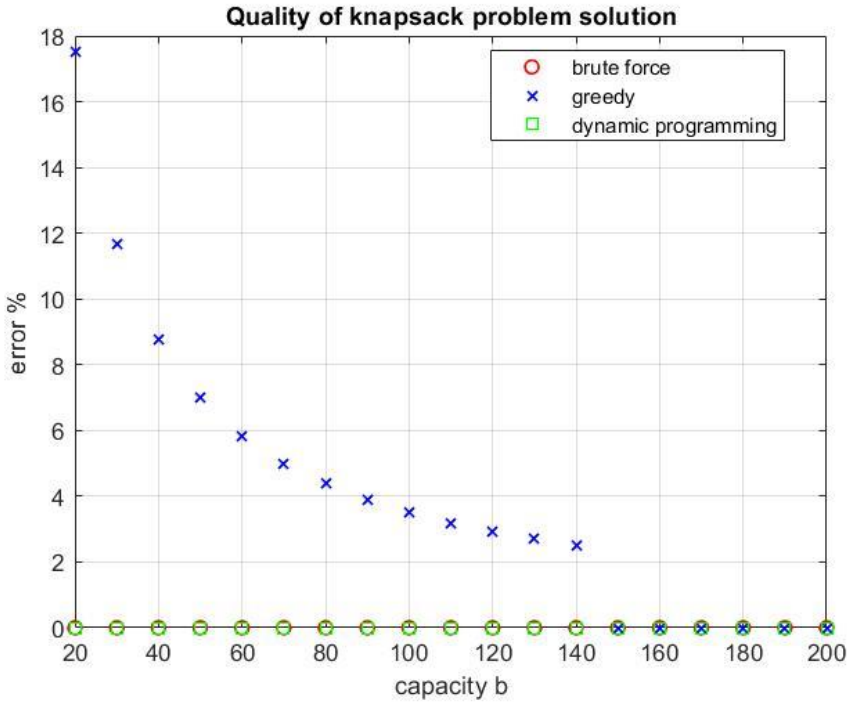

Fig. 9. Percentage of Error(quality of solution) for BF, DP and Greedy for “artificially generated data”*-see observations and conclusions.

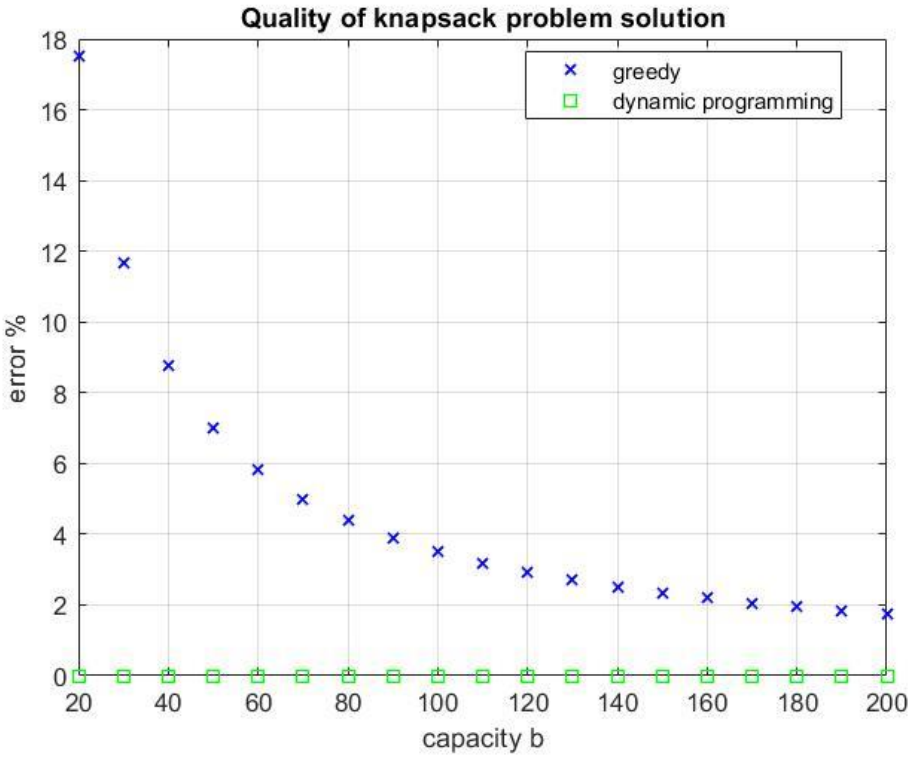

Fig. 10. Percentage of Error(quality of solution) for DP and Greedy for “artificially generated data”* for larger n.

Tab 1. Table of errors comparing BF, DF and Greedy algorithms for randomly generated data.

Number of items	Error %		
	brute force	greedy	dynamic programming
4	0	0	0
5	0	0	0
6	0	0	0
7	0	0	0
8	0	0	0
9	0	0	0
10	0	13.376	0
11	0	0	0
12	0	0.746	0
13	0	0	0
14	0	0	0
15	0	0.526	0
16	0	5.505	0
17	0	0	0
18	0	3.83	0
19	0	0	0
20	0	0.784	0
21	0	5.517	0
22	0	0	0
23	0	2.154	0
24	0	1.342	0
25	0	0	0
26	0	3.846	0
27	0	0	0
28	0	1.212	0
29	0	0	0
30	0	0	0

Observations and Conclusions

Based on Figures 1,2 and 3 it is easily noticeable that the Brute Force algorithm's time complexity is much greater than the other two algorithms. It is unsurprising as an exhaustive solution requires the computer to test all possible combinations of items. One benefit to using this algorithm is the fact that it will always return the optimal value. The time necessary to reach the optimum seems independent of the capacity of the knapsack-Figure 8. Brute Force as an algorithm is almost impossible to use for large n as the time necessary to find a solution grows too quickly. My implementation of the algorithm was based on binary numbers with 0s representing items outside of the knapsack and 1s representing items inside a knapsack, as such the complexity of the Brute Force Algorithm can be described using the 2^n function.

As can be seen in Figures 1,2,3 and 5 Greedy is the fastest out of the tested algorithms. It is caused by the fact, that this algorithm's complexity mostly depends on which sorting algorithm is chosen. The choice of quicksort means that in the average case the time complexity of this algorithm can be seen as a function of nlogn, however, in the worst case it may end up being characterized by the function of n^2. Based on Figure 6 it seems that the algorithm's time for finding a solution is independent of the capacity of the knapsack. However, it is important to note that the Greedy solution does not guarantee to find an optimum. As such, during the experiment error was measured by comparing the solution found by the algorithm compared to the optimum. As such, based on Figure 4, it seems that for randomly generated data the percentage of error is also going to be mostly random(dependent on data of course, but due to it being random it is hard to draw any conclusions). Although the overall trend seems to be decreasing as n increases but it is still hard to draw any concrete conclusions. As such, another experiment was conducted with "artificially generated data"- it was named that way because data was specifically created to illustrate the cases for which Greedy may make errors. "Artificially created data" contained 1 item with a value equal to 23 and weight equal to 11 and n-1 items with values equal to 10 and weights equal to 5. Such an approach was taken, because for this experiment knapsack's max weight was always a multiple of 10, which means that Greedy would prioritize putting the 23/11 value first. This means that for max weight=20 it would only put in 2 items(23/11 and 10/5) but the best solution requires an algorithm to put 4 items inside a knapsack(4 x 10/5). As such, the error will decrease for growing n but will be present until b(max knapsack capacity/max knapsack weight) is big enough so that the algorithm can fit all the items inside the knapsack-as seen on Figures 9 and 10. This experiment was meant to show that the error in the Greedy Algorithm originates from the fact, that prioritization of items with the greatest value to weight ratio may cause the knapsack to not be full and as such, may cause the algorithm to not find the optimum value.

Dynamic Programming managed to find the solution for the knapsack problem much faster than the exhaustive search(BF)-Figures 1,2,3. However, it seems that this algorithm performed finished its operation slower than Greedy- Figures 2,5. This algorithm uses n by b sized table in order as it breaks down the problem into smaller ones. As such its time complexity can be described as O(n*b) with b=max possible weight of the knapsack. This implementation of the algorithm also uses an additional O(n*b) sized array, meaning that it also has a memory complexity of n*b. This is further backed up based on Figure 7, as the time necessary for completing the task increases with both n and b. Based on Figures 4,9,10 it can be noted that this algorithm returns an optimal solution, meaning that the knapsack problem can be classified as an NP-complete problem.

In conclusion, the Brute Force algorithm will always give the optimum solution, however, the algorithm's time necessary to find the solution means that its application for the knapsack problem is quite limited, mostly only to relatively small n. Max weight also seems to not affect the working time of the algorithm which means it can be used on sets of data with large b, provided value n is relatively small. The Greedy Algorithm is very fast, in comparison to the other two, which means that it can be used in cases where obtaining the optimum is not the goal and a heuristic solution may suffice. Dynamic Programming obtains the optimum while also achieving that much faster than the BF algorithm, as such DP is best used for cases where an exact solution is needed, even if n is relatively large. However, depending on hardware used and values of both n and b. If either or one of these values is too large, limitations caused by computer's limited memory may occur. However, that can be partially mitigated with use of different implementations of DP.