Mateusz Małecki - 148265
Nikodem Seidel - 148269

Knapsack problem - how to solve one issue in many different ways.

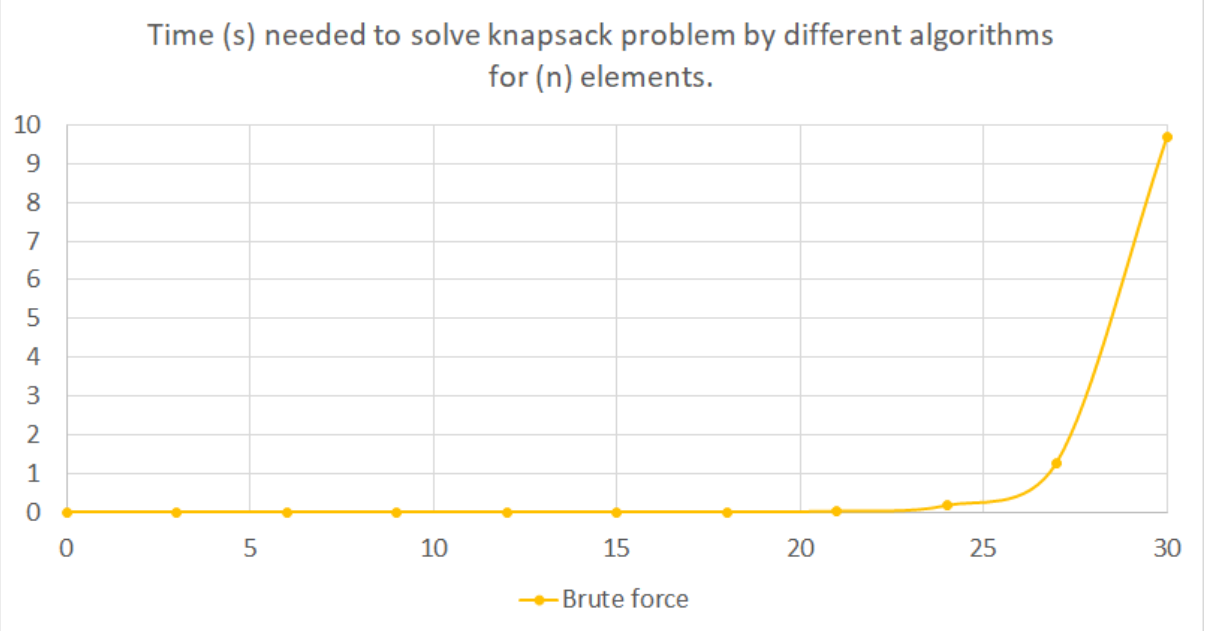
The knapsack is one of the most famous problems in the IT world. Because of that, there exist many different solutions. In this report, we've considered 4 approaches:

- recursive brute force
- greedy like,
- iterative dynamic programming
- recursive dynamic programming

Every approach has pros and cons (different precision or time and space complexity) and the main aim of this experiment was to show these distinctions.
In this project, we've created test data in such a way that the values and the weights of items are independent and completely random.

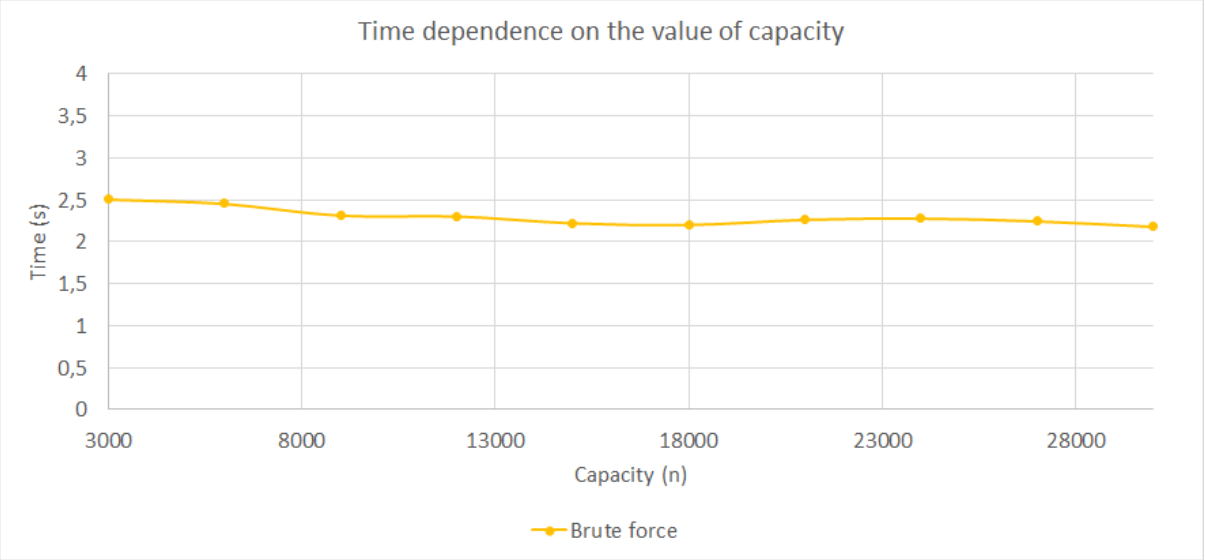
1. Time and space complexity considerations.

- **Recursive brute force** is the easiest and definitely the slowest one. This approach checks every possible solution that exists. It can be compared to checking all possible permutations of a binary sequence of length n, therefore its time complexity is equal to $O(2^n)$, where n is a number of all items we can put into the knapsack. It is highly non-optimal and uses recursion, which makes it even slower.
  Computer memory is used purely - only for keeping all items i.e. two arrays with values and weights. It makes its memory complexity equal to O(n) where n is equal to a number of elements.

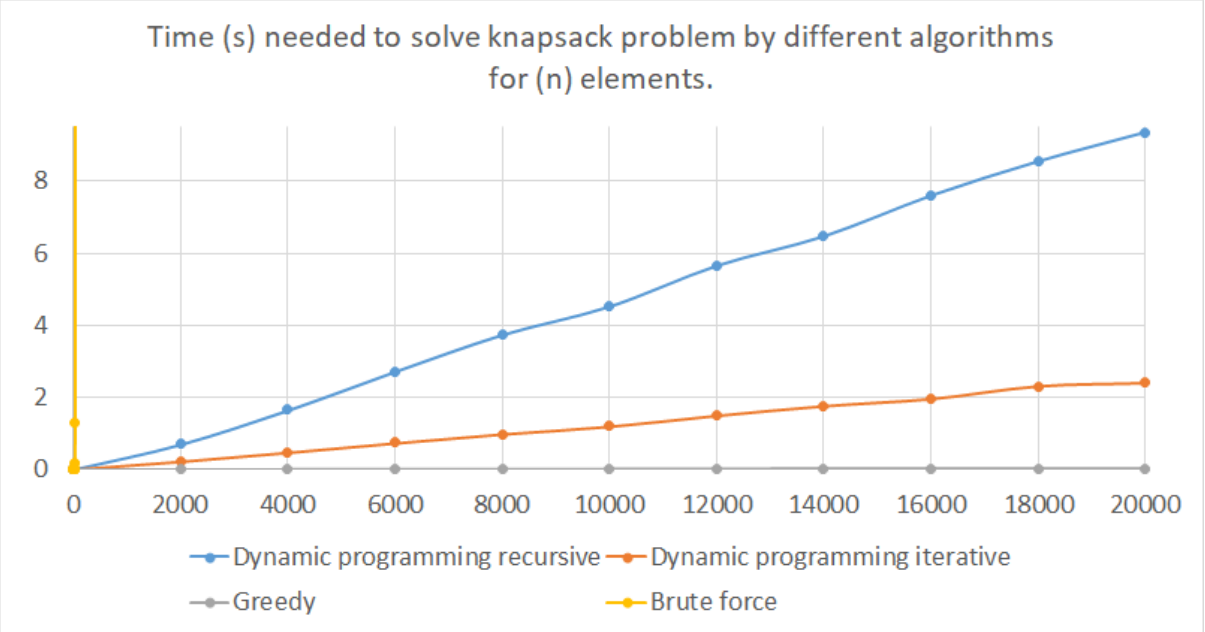


What is also worth mentioning is the fact that the BF approach is capacity insensitive. This means that manipulation of the capacity size for a constant number of elements does not influence the time complexity since it always checks every possibility. This can be clearly

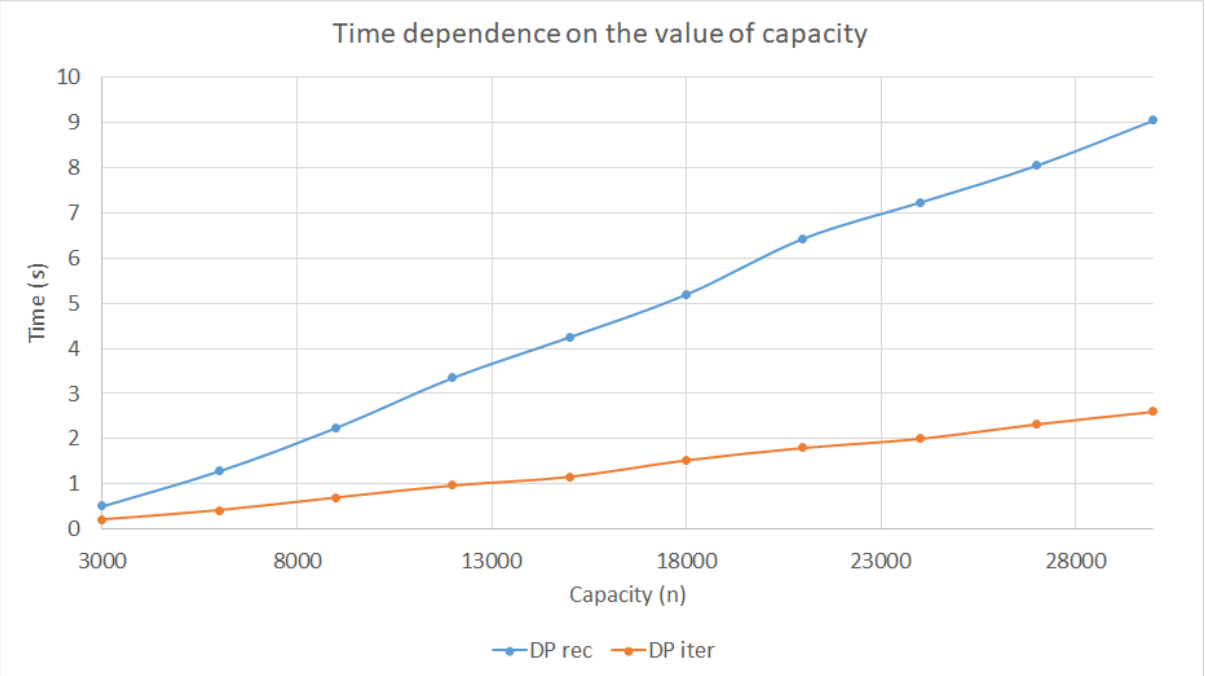
seen on this plot:



- **Dynamic programming (DP)**, both (iterative and recursive) approaches are using the same idea. The brute force algorithm calculates the same things many times so to avoid repeating processes we can calculate it only once and store the result in the 2D array. It speeds up the algorithm a lot and that is exactly what DP approaches do. From the graph below we can conclude that time complexity is O(n * m), where n is the number of elements and m is the weight capacity of the knapsack. The reduction of complexity from the exponential to the polynomial makes a big difference. For example, when capacity is equal to 12000 we can perform a 10 seconds DP algorithm for nearly 20,000 elements or BF for only 30 elements. This shows how important it is to write correctly optimized programs. Of course, brute force can be faster but only in extreme cases, where backpack capacity is irrationally big and the number of items is very small. The space complexity of this program is the same as the time complexity (n*m). It's undoubtedly one of the biggest cons. The difference between iterative and recursive algorithms is that computers better deal with iterations than with recursion so the first one is a little bit faster. (time complexity coefficient is smaller).



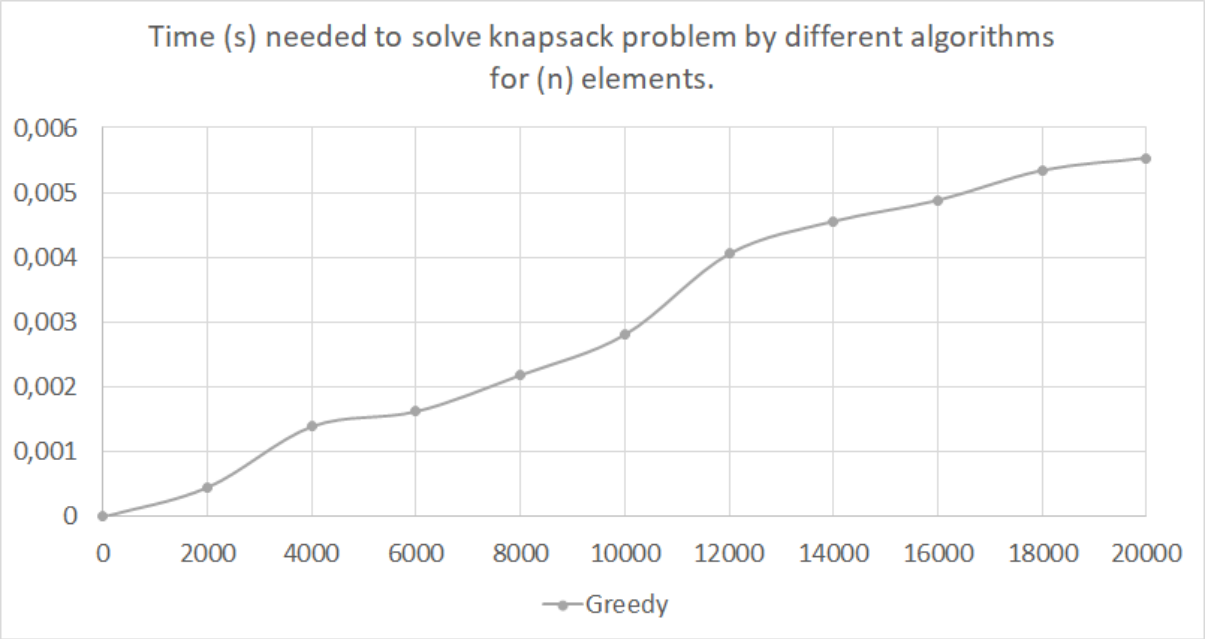
DP approaches are capacity sensitive. It means that an increase in capacity causes a linear increase in time and space needed to make calculations.



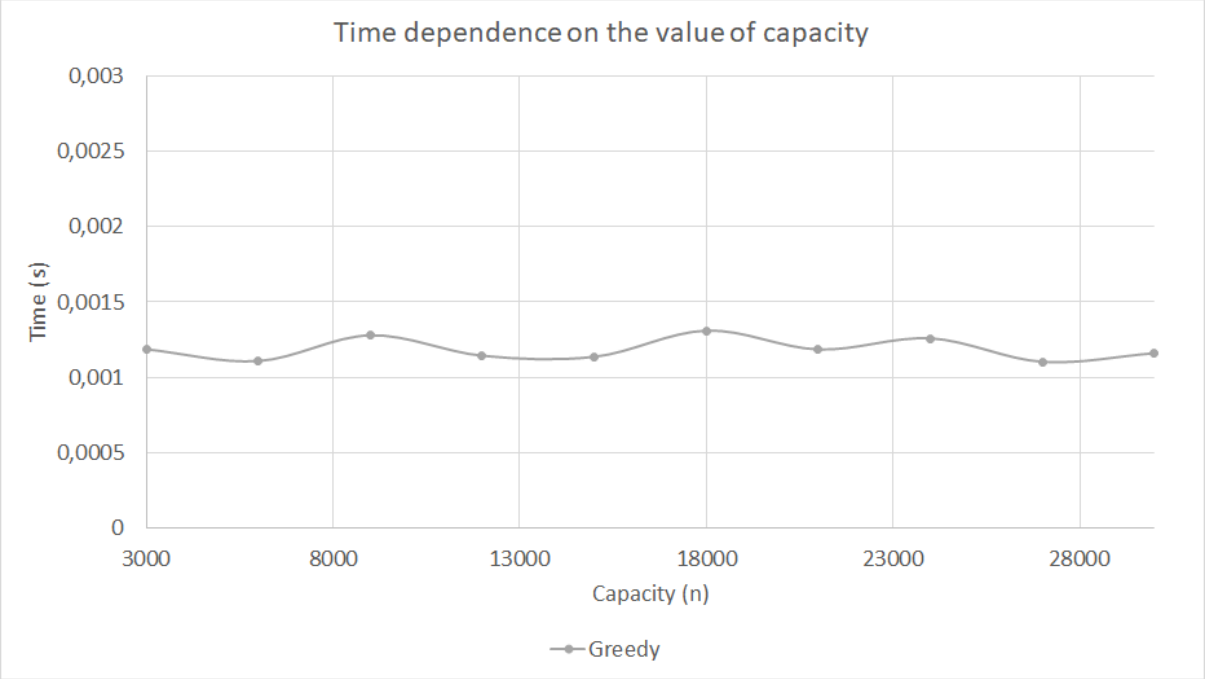
What can be easily concluded from the above statements is the fact, that in the case when capacity and no. of elements are increasing at the same moment the plot would have a parabolic shape (since linear function times linear function equals quadratic function).

- **The greedy algorithm** is the fastest one. It's also very simple. The main rule is to pick items with the best value/weight ratio as long as there is any capacity left. It starts with creating a new array of length n and filling it with calculated ratios for every single item. The next step is to sort this array, so the algorithm could easily

iterate through it and find the highest ratios in the fastest possible time. We used Quick Sort, therefore time complexity is equal to $O(n\log_2 n + n) = O(n\log_2 n)$ where n is the number of the elements.
The algorithm uses memory to keep information about values and weights of all elements and also to keep all item's ratios. Therefore memory complexity is equal to $O(3n) = O(n)$.



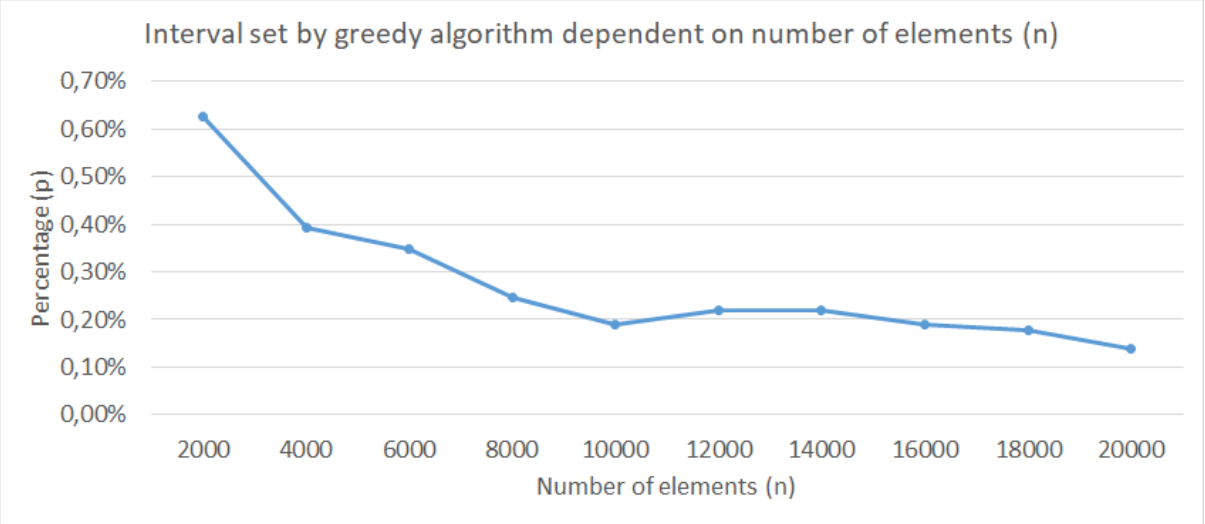
This algorithm is capacity insensitive. This means that manipulation of the capacity size for a constant number of elements (8 000) does not influence the time complexity since it always iterates over n (number of elements in the problem) element arrays.



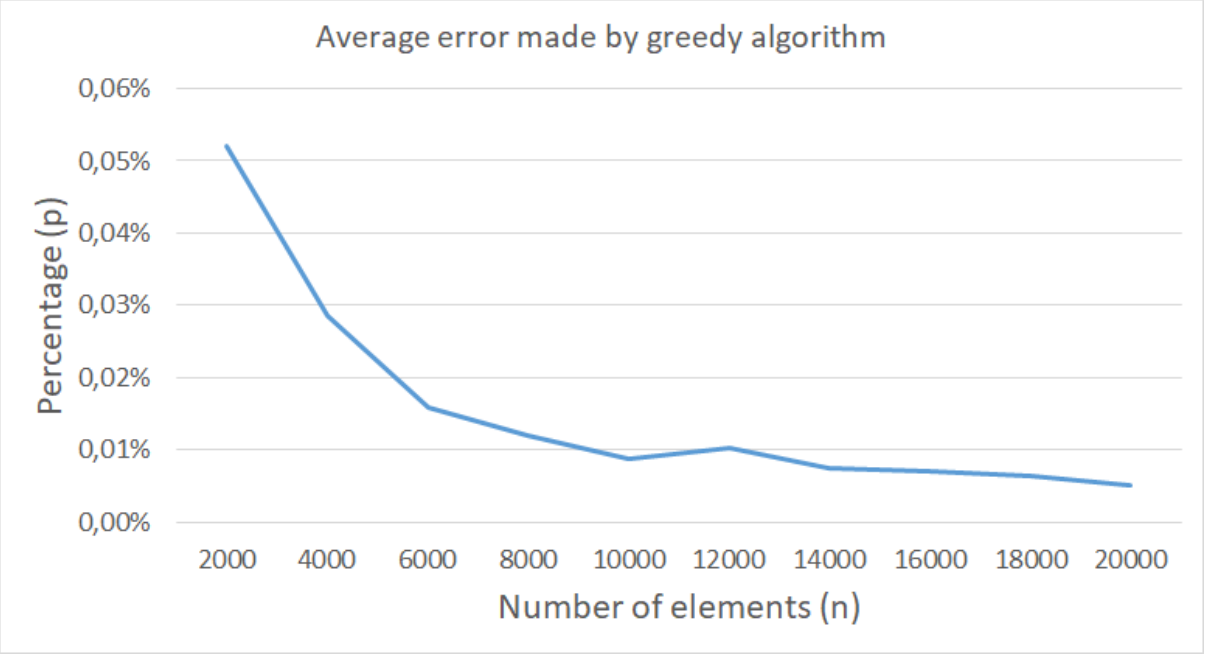
2. Quality of the solutions:

The first three approaches are unerring, which means that they always find the optimal solutions for the knapsack problem. This cannot be said about the greedy approach. Our

algorithm is capable of setting lower and upper bound for the optimal solution. To our surprise, it appears to be pretty accurate. On the below plot we can see, that for fixed capacity, increasing the number of elements also increases the accuracy of set intervals for the greedy algorithm, which was definitely expected since because of the way how test data is created the optimal solution consists of relatively light and valuable items. Of course, a bigger dataset causes that we can find more items like that so the interval will be smaller. In this case, the error is pretty small but there exist instances where it can be high.



In the next plot, we showed the percentage of the real solution by which the greedy algorithm miscue. Similarly like in the previous example, increasing the number of elements in the problem also increased accuracy for our algorithm.



To sum up, Knapsack Problem is quite a complex problem, especially without the help of DP. What's is interesting is the fact, that this problem still remains NP-Hard, because there is no algorithm that can solve it in polynomial time. Nevertheless, there is a commonly known

pseudo-polynomial algorithm based on DP, which allows finding the optimal solution but it is not only dependent on the number of input data but also on the size of the number of bits required to represent it.