**Algorithms and computability**

Szymon Majorek

Basic project level 2

Final report

**Table of contents**

1. Description of the application
   1. GUI
   2. Code behind
2. Time complexity
3. Experiments
4. Distance between results
5. Conclusions

**1. Description of the application**

**a. GUI**

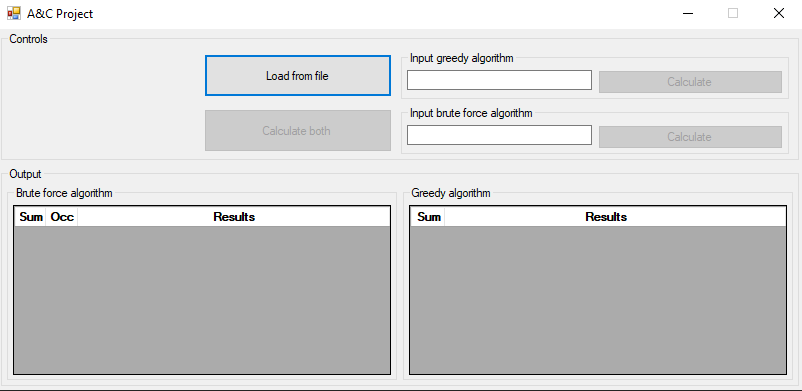


Figure 1 - GUI Overview

The GUI of the application is rather simple, it allows user to input numbers for both Greedy algorithm as well as the Brute force. We can load input from file, via the ‘Load from file’ button. The file has to be .txt, however it has to be formatted in csv format, with a single comma (,) being the separator. The application auto checks the input for every change in the input. If the input is incorrect (or empty) the activation buttons are disabled.

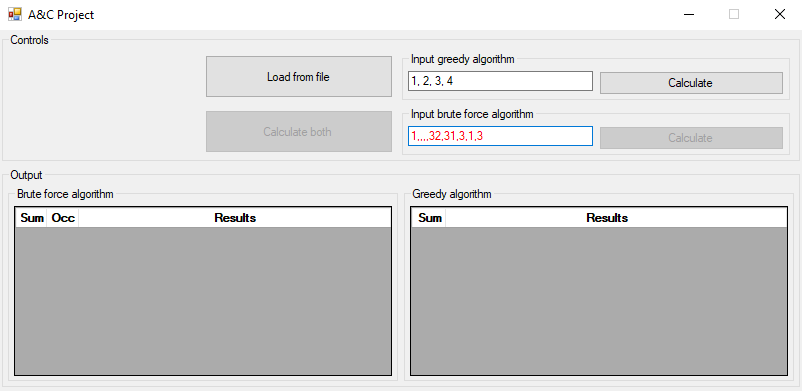


Figure 2 - Input validation

As we can see on the screenshot the application validates the input, incorrect input is marked as red. The ‘Calculate both’ button is only active if the both inputs are correct and no calculations are being currently made.

After calculations, the results get displayed in the two DataGridViews in ‘Output’ group box.

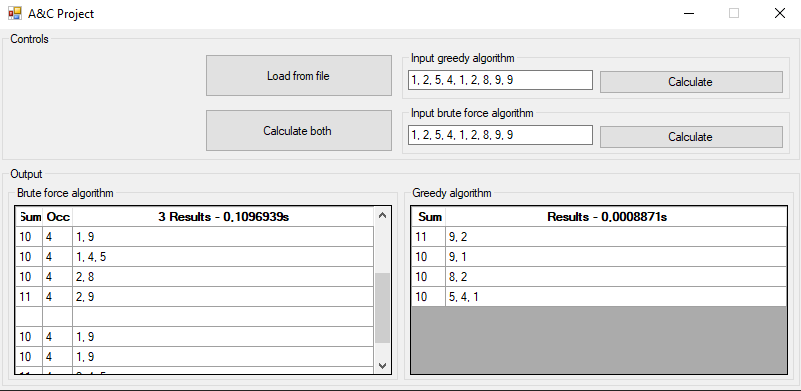


Figure 3 – Results View

The tables differ only by a little. Since the greedy algorithm always finds only one set of subsets it doesn’t need to ‘Occ’ column, which indicates how many occurrences of given set of subsets were found. Additionally we can see the calculation time for both of the algorithms, as well as the sum for each individual subset. For brute force we can additionally see how many different (in theory) subsets were found.

**b. Code behind**

Since I created a GUI, I am using multithreading in the application so as not to block the GUI when waiting for results for one of the algorithms. After starting calculations, the user can still interact with GUI, for example scrolling through the results of BF algorithm. Once user presses the ‘Calculate’ button, the input get’s parsed and then asynchronous function (background worker to be exact) starts and calculates our results. Once it ends, the results are being read into the data view for given algorithm.

The number of occurrences is calculated in rather naïve way. I use MyDictionary class for that, in which I have a dictionary and a MyEquals method which is able to compare two List<List<int>> objects. I hope that sorting first by sum, and then sorting each sublist will result in the same output, then I can go through each element and compare them to see if lists contain the same sublists and elements in them. However this method doesn’t always yield 100% correct results, though it get’s rid of most of the duplicates.

**2. Time complexity**

The time complexity seems to be absolutely huge for the brute force algorithm and rather decent for greedy one. Brute force complexity is connected to the Bell number, meaning the number of possible partitions of a set. The first part of the algorithm is to find all the partitions. Then, from them all we select the ones that have exactly 4 subsets (because we want a partition into set of 4 subpartitions). That requires only one pass through the List<List<List<int>>>. After that we enter the function responsible for finding the most optimal partitions.

There we proceed mostly according to the pseudocode, though I made few very small optimizations so as not to store too much useless data. Another small change is that we store indexes in exact\_solutions as well. The optimization is that we only add new entry in the dictionary if we don’t have any exact solutions (no point in adding anything in the dictionary, if we are sure that we’ve already found better solution) and if currently calculated difference between Max and Min sum is smaller or equal to the smallest one currently being stored. The complexity of this depends on the amount of partitions with 4 subsets found. We also have to find the sum of each subset (4 passes). Then we mostly have ifs and .Where from LINQ which is a single pass through the list.

With C# not being the fastest language, and the code not being that well optimized, the brute force takes huge amounts of time even for small inputs. For example input of length 20 seems to run almost indefinitely, which is pretty bad.

The greedy algorithm on the other hand is extremely fast and handles large inputs rather well (in time complexity sense at least). For a randomized input of size 500, it took only 0.000165 seconds to calculate.

**3. Experiments**

The fact that the brute force algorithm becomes virtually useless for inputs of length greater than 15 heavily limits the experimentation that we can do. Because of that, I will split the experiments into two parts. The first part will be about brute force, and comparing the two on smaller inputs whereas the second part I will play around with inputs of greater sizes for the greedy algorithm.

In order to produce my inputs I will use simple python script.



Figure 4 - Python script to generate input

It generates numbers from normal distribution where I provide mean and variance, however since technically I am dealing with weights, I cast them to integers to avoid floats and then shift the whole thing to the right by the absolute value of the smallest element +1 so as to avoid negative weights. Because of the fact that I’m manipulating obtained results, they won’t have exactly the mean and variance that I provide, especially for the first part where we will be using small number of elements in the input, however I believe that it is still a good starting point, a better one than coming up with the sets myself.

I will be running each experiment for a number of times and the calculating average as well as producing boxplots (provided high enough number of runs.

We will start with a small input of, say 8 elements and high variance.

**Experiment 1**

Input: 36, 1, 333, 614, 25, 635, 199, 400

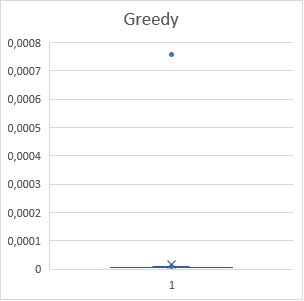
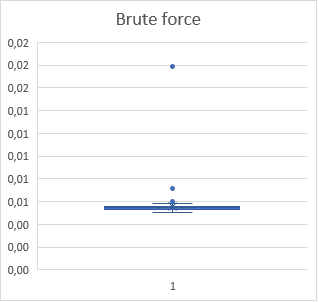
Variance: 57968.984375

Average time greedy: 1,43E-05 s

Average time brute force: 5,61E-03 s

MaxMin difference: 173 (both)

Number of runs: 89



As we can see on the boxplots the results are very consistent. For small number of elements in the input we get extremely small running time with singular outliers for both brute force and greedy algorithms. Even at this point we can see just how much faster the greedy algorithm is than the brute force. On average it runs 391,61 times faster than brute force.

The obtained results are obviously the same for each run:

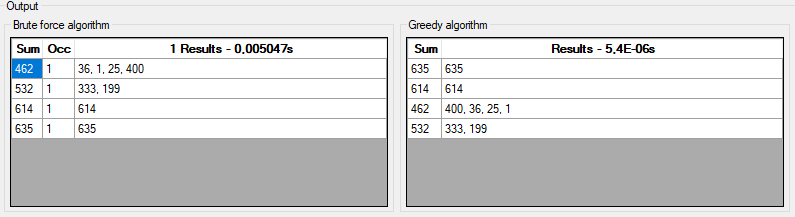


Figure 5 - experiment 1 results

In this case, both algorithms actually produced the same results.

**Experiment 2**

Similar experiment, but here we will try smaller variance, as small as possible.

Input: 18, 19, 19, 17, 18, 18, 19, 17

Variance: 0.609375

Average time greedy: 1,30E-05 s

Average time brute force: 2,62E-02 s

MaxMin difference: 1 (both)

Number of runs: 100

Here we can see that the results for greedy are almost the same, however ones for brute force are much worse (time needed wise). For this input, it ran almost 10 times slower on average than in previous case.

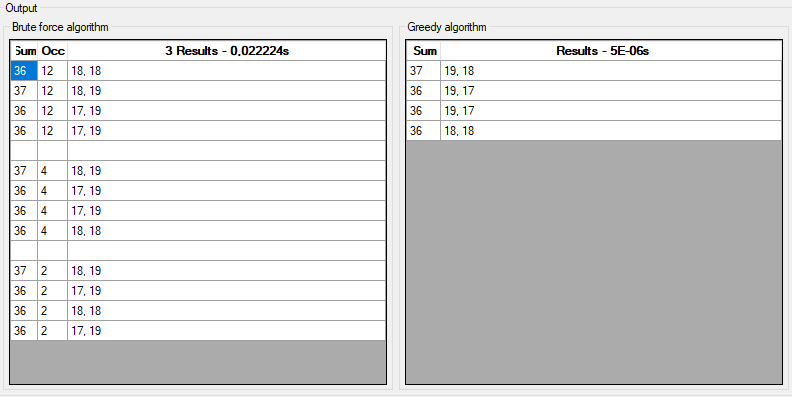


Figure 6 - Experiment 2 results

In this case we can actually see that my way of determining whether 2 given results for brute force are the same in some cases fails to classify them as such. We can see that have 3 different result, however they are actually all the same. Again the resulting subsets for both greedy and brute force is the same.

**Experiment 3**

Let’s try to increase the number of elements in the input.

Input: 16, 15, 13, 15, 17, 26, 21, 16, 18, 16, 13

Variance: 12.809917355371901

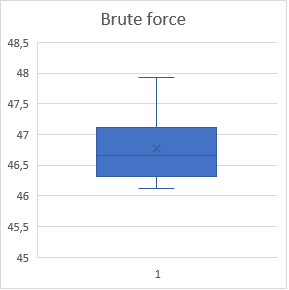
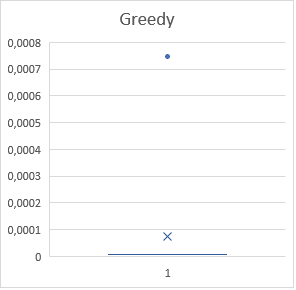
Average time greedy: 7,48364E-05 s

Average time brute force: 46,77182 s

MinMax difference greedy: 9

MinMax difference brute force: 1

Number of runs: 11

Now we can actually see that the brute force becomes much, much better, though also much slower.

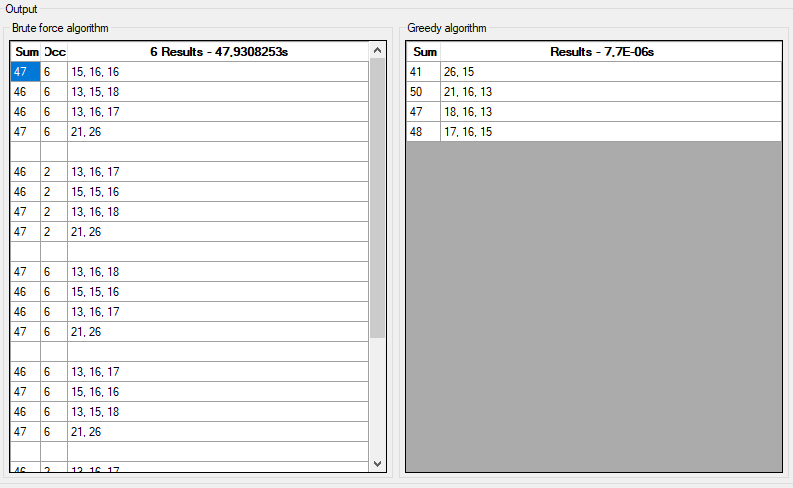


Figure 7 - Experiment 3 results

The difference in time is enormous, greedy runs, on average, over 600000 faster than brute force (provided we do a very small number of runs for this experiment). However brute force got also much better results, as we can see on the output, the difference between min and max sum was equal to 1, compared to 9 for greedy for a set with not that high variance.

**Experiment 4**

Let’s try to increase the number of elements, but also to increase the variance of our input.

Input: 53043, 106464, 68688, 79487, 69877, 1, 31190, 146375, 112131, 121562, 32447, 131310, 86305

Variance: 1733775458.4378698

Average time greedy: 8,24E-06 s

Average time brute force: 171 s

MinMax difference greedy: 6522

MinMax difference brute force: 27855

Number of runs: 5

Since the number of runs is so low, there’s not much of a use for boxplots. The interesting part is that the brute force became even better when compared to the greedy algorithm.

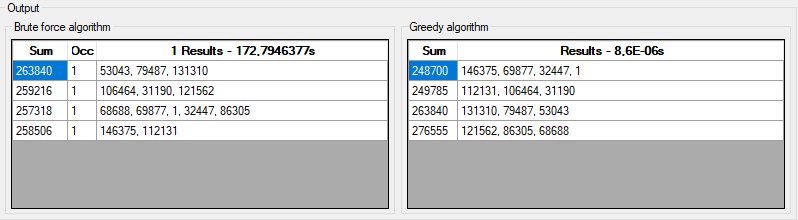


Figure 8 - Experiment 4 results

As we can see on the output view, the MaxMin difference for brute force is more than 4 times smaller than the one for greedy.

**Experiment 5**

Input: for input I used a set of 500 numbers calculated using the abovementioned python script, I used variance=10 and mean=10.

Variance: 10.196399999999999

Average time greedy: 0,000933 s

Number of runs: 100

This experiment is meant only for greedy, as the brute force would run way too long.

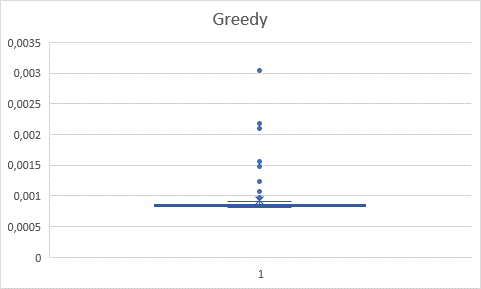


Figure 9 - Eperiment 5 boxplot

As we can see on the boxplot, for such a large input we get slightly more outliers, however the algorithm still is extremely quick. For such a big inputs, using brute force would be impossible.

**4. Distance between results**

We expect to see a growing difference in the quality of obtained results for the two algorithms. For smaller inputs, we expect both algorithms to be similarly bad, as there aren’t many choices to choose from for the brute force, however as the set size grows bigger, the brute force should be getting closer to exact solution (difference equal 0) whereas the heuristic approach might fail to get close. The difference should be the most visible for sets with high variance.

First we will consider sets with rather low variance. For each input I ran around 5-10 experiments.

|  |  |  |
| --- | --- | --- |
| Set size | BF | Greedy |
| 6 | 16 | 16 |
| 7 | 17 | 17 |
| 8 | 3 | 3 |
| 9 | 10 | 17 |
| 10 | 1 | 8 |
| 11 | 6 | 12 |
| 12 | 1 | 2,50 |

Figure 10 – figure 11 data

The values for a better overview.

Figure 11

As we can see the algorithms perform pretty much the same for low input size, they consistently how the same difference between max and min sums, however as we increase the input size, the brute force appears to be pushing clearly ahead in terms of getting smaller differences. The results vary heavily on the input that we provide, as we can also see, even for input of size 12, it may happen that the algorithms perform similarly well.

Figure 12

Here we can see the graph for sets of higher variance. Even though on the first glance it looks rather similar to the first one, it paints slightly different picture. The algorithms perform in a similar fashion for small inputs again, however as we move on the brute force pulls very clearly ahead.

Figure 13 - figur 12 zoomed in

|  |  |  |
| --- | --- | --- |
| size | BF | Greedy |
| 6 | 52 | 52 |
| 7 | 46,8 | 46,8 |
| 8 | 25,4 | 25,4 |
| 9 | 18,8 | 21 |
| 10 | 10,5 | 14 |
| 11 | 4,333333 | 16 |
| 12 | 5,5 | 20,25 |

Figure 14 - figure 13 data

Looking at the latter part of the graph, when the algorithms actually start to differ, we can easily see that brute force is much better and the trend is much clearer compared to the low variance sets. Again, for each data point I ran multiple experiments and took the average result.

**5. Conclusions**

Judging by the few experiments I was able to perform, I would say that brute force algorithm is much better for inputs with few elements and very high variance. The higher the variance was, the bigger the difference between the two algorithms was, with brute force coming on top of course. However for larger inputs, it’s virtually impossible to use the brute force algorithm, it simply take too much time to be useful in any way. As long as we have low variance in our input, the greedy algorithm seems to be fine, though for inputs with higher variance, the difference between Max and Min sums becomes very big.