Comparing Knapsack Experiment

Algorithm Analysis Report
CSB 302 Winter 2024

Team Friendsheep

Robin Batingan, Mika Kitazumi, Nguyen Giang, Tamara Slone

Experiment Introduction

The basis of our experiment is the knapsack problem. The basics of the knapsack problem are you have certain items with a designated weight and value. The knapsack has a certain capacity that can only hold a certain amount of weight. We want to maximize the knapsack and pick the items that will give us the most profit value while keeping the weight within the capacity of the knapsack. There are 2 types of knapsacks that we can do this with, a Fractional Knapsack, And a 01 Knapsack

A Fractional Knapsack is a knapsack that can break the items apart to fill the capacity to the max. For a Fractional Knapsack, we used a **Brute Force** algorithm, and a **Greedy** Algorithm to find the max profit of the fractional knapsack.

A 01 Knapsack is a knapsack that can not break the values of the items, therefore the max value does not necessarily mean that the knapsack will be at max capacity. For a 01 knapsack, we can find the max profit using a **Brute Force** algorithm, a **Greedy** algorithm, and a **Dynamic Programming** algorithm.

Our experiment is comparing the runtimes of these different algorithms. We want to see the difference each implementation will have, and compare the theoretical time complexity to the practical one we get.

To achieve this, we made a program that implements the different algorithms at the same time and displays the runtime of each. The input for the algorithms is 6 different CVS input files that contain the weights and values of different items, as well as the capacity of the knapsack. We converted the input files into usable arrays by making a **knapsack object** and then passing those values into the various methods. The methods then return the maximum profit of the items given from the different inputs, as well as the runtime of each.

Deep dive into each algorithm for both knapsacks

Fractional Knapsack

Brute Force: Using a **Brute Force** algorithm works by evaluating each value of the knapsack and calculating the total profit available. Although this algorithm is useful and can be used to find the optimal solution, the problem lies in the average runtime for Brute Force methods. The average time complexity for a brute-force algorithm can run at a big O of $O(2^n)$. This can severely affect the amount of time for a program with larger number sizes to run.

For our implementation of the brute force algorithm, we calculated each item with different weights and values and updated the max value variable if a higher one is reached. After the algorithms find the highest value, if there is any remaining capacity, it will take fractions of each item, and find the items that will make five the most value.

Greedy: The **Greedy-O(nW)** algorithm works by dividing the original problem into sub-problems. So it is going to decide the best choice based on the ratio. Then for each sub problem, it's going to take the best choice. If the best choice can't be fit into the capacity, The Greedy is going to take the one with the highest ration left and do the fraction and then multiply the fraction with the value to fill up the knapsack. The Greedy is working better than Brute Force because it does not have to go through all possible choices.

When implementing the Fractional Knapsack Greedy algorithm into our experiment there were two classes that were created: FKnapsack and Greedy. Our Greedy class contains the run method that implements the Greedy algorithm. This run method initializes the arrays imported from our Input CVS files (the arrays being weight, value, and an integer containing the capacity of our knapsack). Once the arrays are loaded into the method, a for loop iterates through our arrays and divides each weight element by the value element to get our ratios. These ratios are then implemented into their own ratio array.

Once everything is initialized, all of the elements from the arrays will be placed into an object known as FKnapsack. The FKnapsack object contains each individual item information such as an item name, weight, value, and ratios. Once an FKnapsack object is created it is then sent to an ArrayList "totalArrays" containing all FKnapsack objects. These objects are then sorted by descending order based on their ratios. Which will then be compared within a while loop that will calculate the total profit.

The dynamic algorithm was omitted from the program. Although it is possible to use a dynamic algorithm for Fractional Knapsack, it is not typically used. Dynamic algorithms are best suited for 01 Knapsack implementation.

01 Knapsack

Brute Force: The brute force method is the easiest to implement for a 01 knapsack. To calculate the max value, the algorithm takes each and every combination of the weights and values and finds the highest combination of those to find the max value. The runtime should be $O(2^n)$ where n is the number of items. This is because brute force is iterating through all possible subsets to maximize the output.

Updated by Nguyen

Greedy algorithm: The greedy algorithm for the 0/1 knapsack problem is not guaranteed to produce an optimal solution because it makes locally optimal choices at each step without considering the overall context. The problem has overlapping subproblems and the optimal solution may involve a combination of elements that do not fit the greedy criterion. For example, we have several items with ratios. The Greedy Method is going to pick the item greatest ratio and then go with 2nd greatest. When the second item does not fit the capacity of the knapsack, then it's going to skip it and stop the loop, returning the profit. This cause the knapsack capacity sometimes is not filled up which not is not optimized for maximum profit.

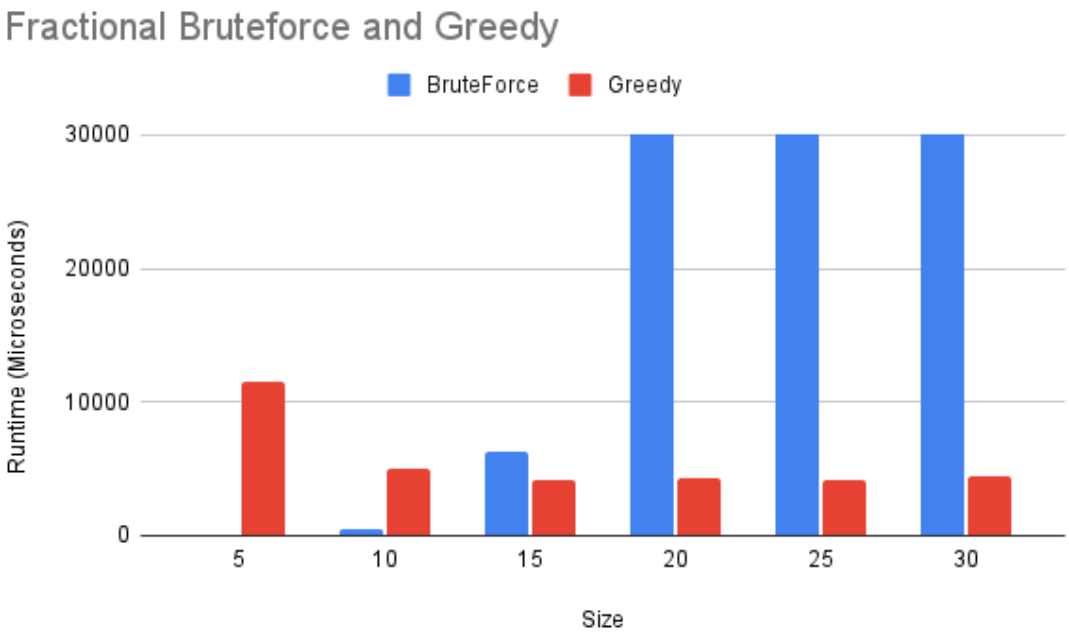
Dynamic Programming: The dynamic programming solution is very different from both the brute force solution and the greedy solution. To find the solution using a dynamic programming approach, you would find the overall solution of the problem by finding the solution to the overlapping subproblems. For the runtime: it should be O(nW), where n is the number of items given and W is the capacity of the knapsack.

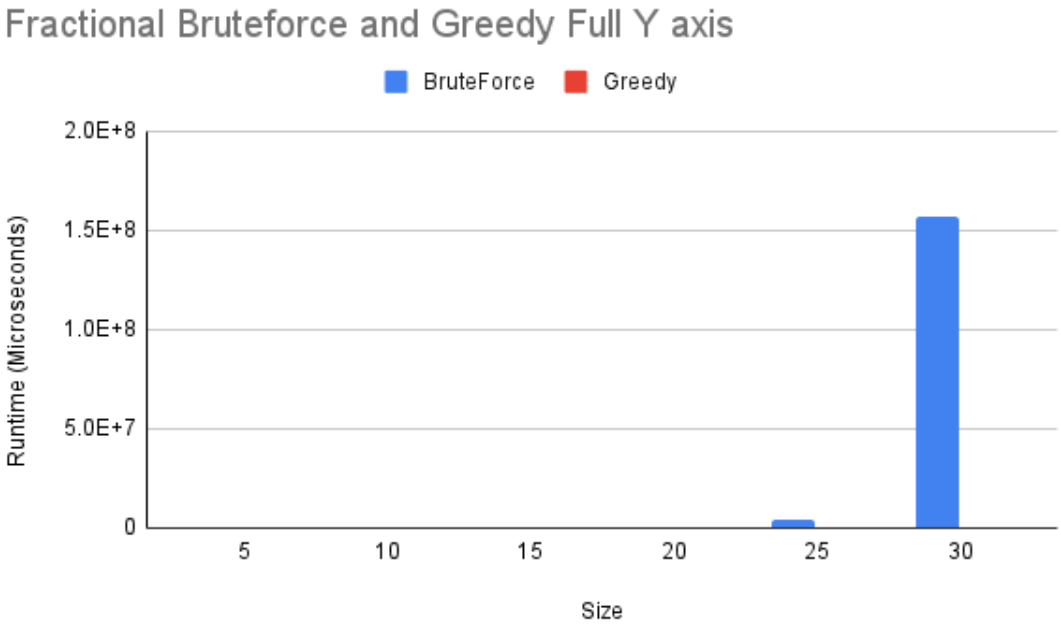
In the case of the 01 knapsack problem, our program will find the weight and value of n items. You want to find the highest value of the subset of items, without going over the capacity. We then create a 2d table, that represents the highest value subset of items that is within the capacity of the knapsack. We then fill the table, then iterate backward through the table to add the items that give the max value of the knapsack.

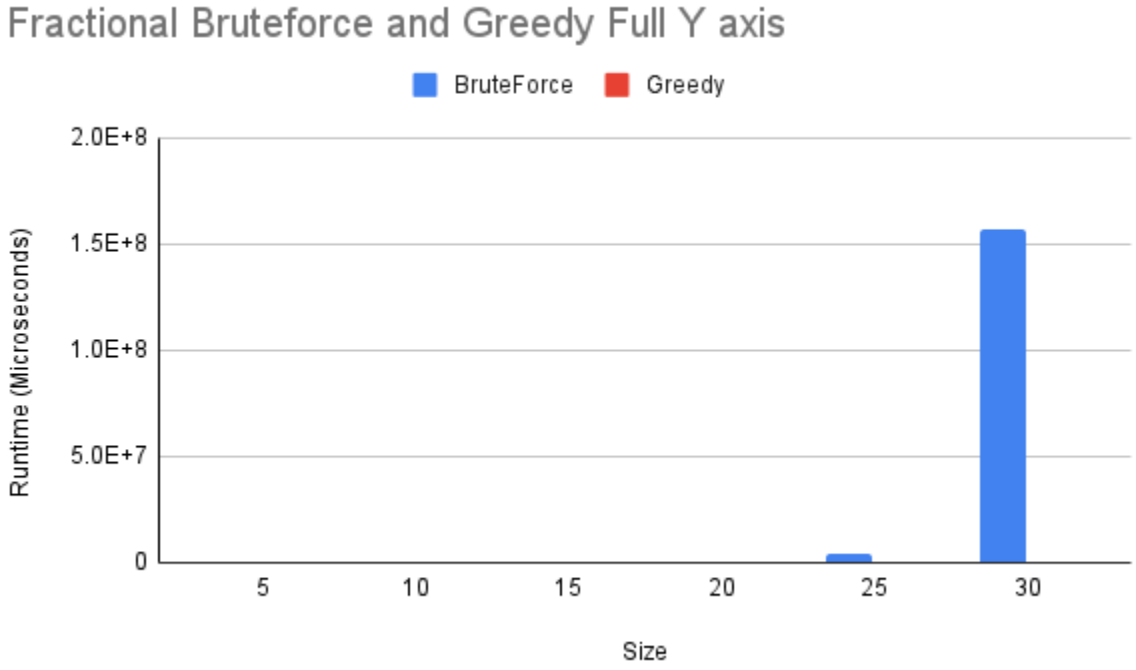
The time complexity for this is O(nW), which is significantly less than the brute force method. It is different from the greedy method because a greedy solution might not be the optimal solution for a 01 knapsack.

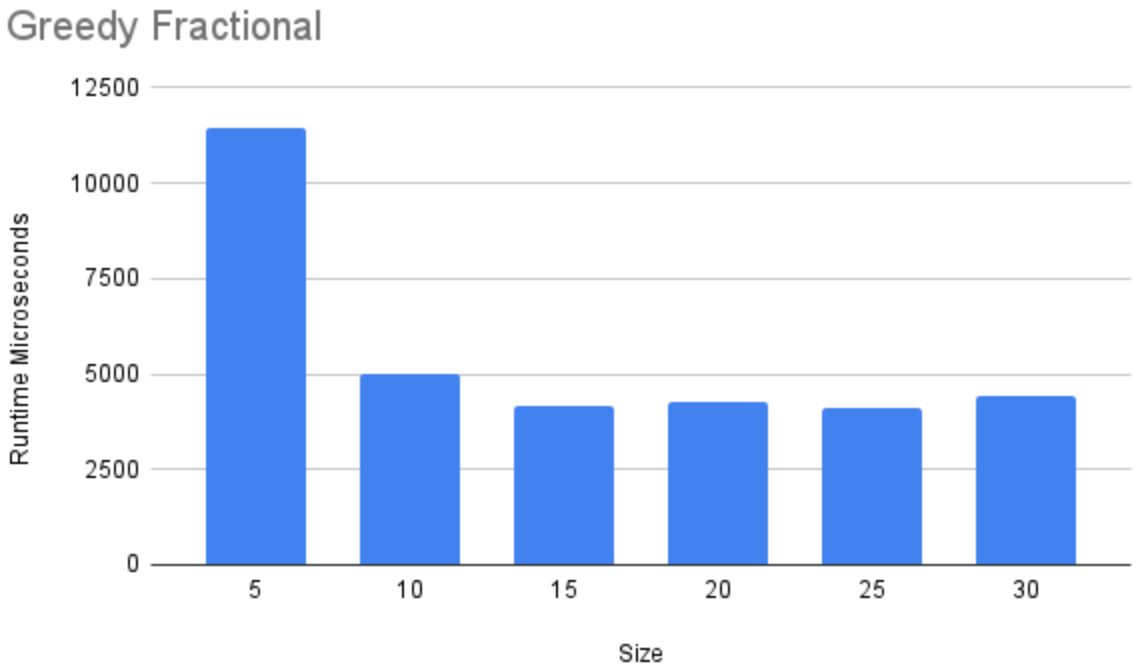
Results of Experiment

Fractional Knapsack Results:




Fractional Bruteforce and Greedy Full Y axis
BruteForce
Greedy
2.0E+8
1.5E+8
1.0E+8
5.0E+7
0
Runtime (Microseconds)
5
10
15
20
25
30
Size

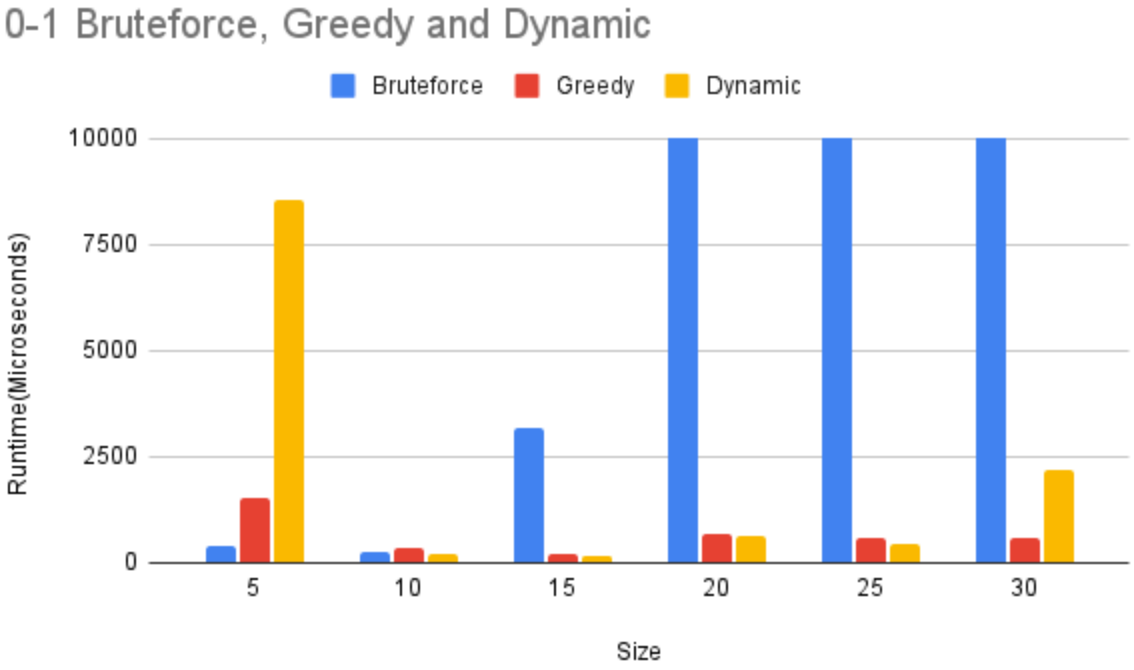

Fractional Bruteforce and Greedy Full Y axis
BruteForce
Greedy
2.0E+8
1.5E+8
1.0E+8
5.0E+7
0
Runtime (Microseconds)
5
10
15
20
25
30
Size

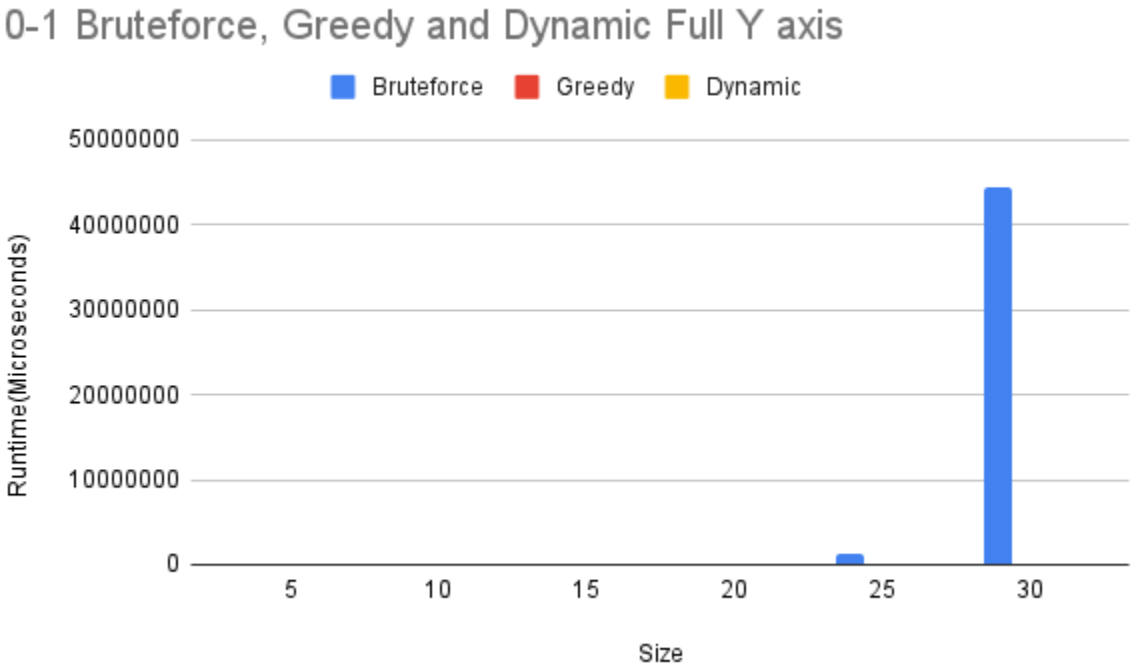


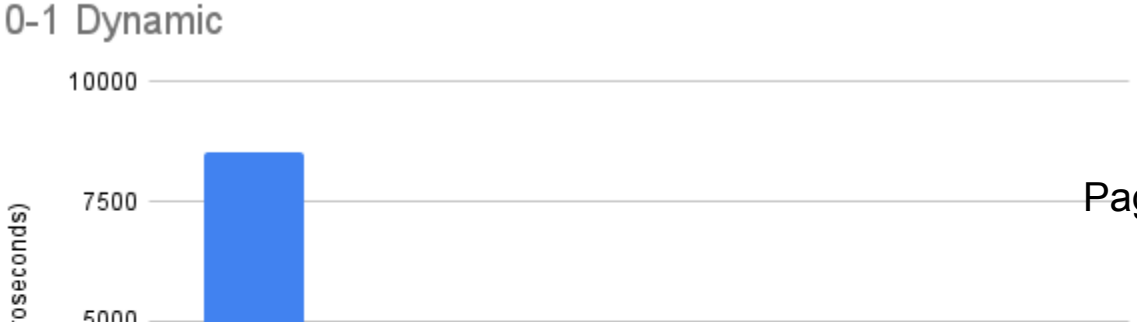
Posted above are the charts of our Fractional Knapsack results. One of the more obvious visual results is that the Fractional Bruteforce algorithm takes an exponentially larger amount of time to process in microseconds than the Fractional Greedy algorithm. This aligns with the theoretic assumption that Brute Force algorithms take a longer runtime (O(2^n)) than our typical Fractional Greedy algorithm ($O(nlogn)$).

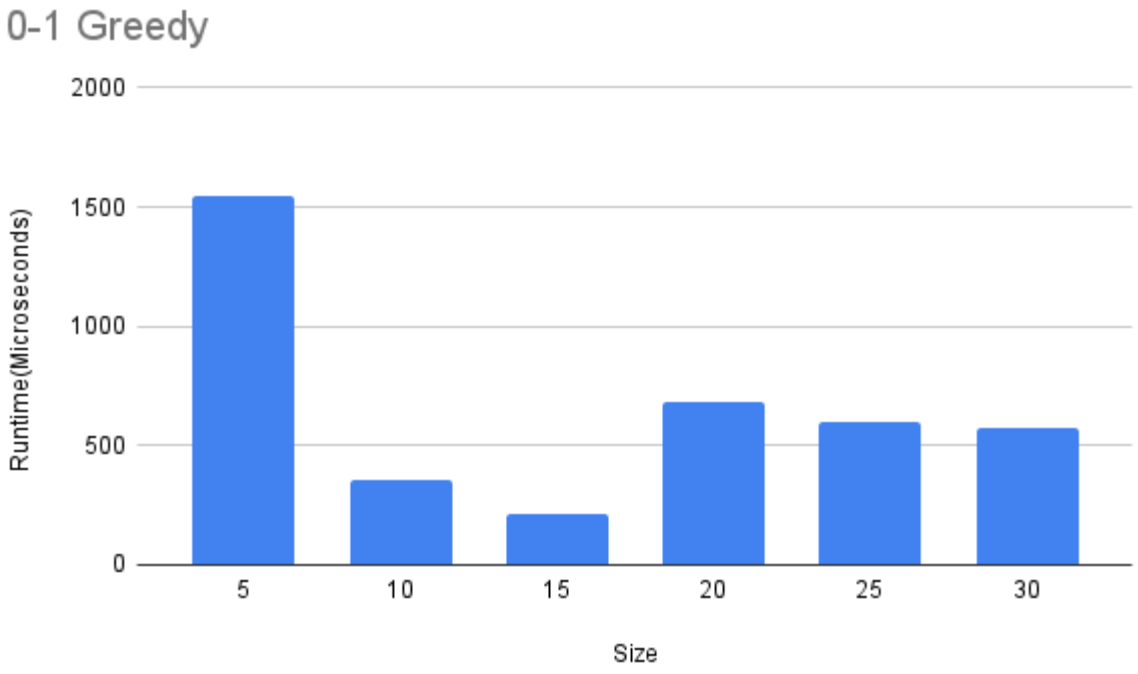
However, some results show outside the typical theoretical assumptions. For example the result of Input 1 (size 5 on the graph) shows the Greedy algorithm with a higher runtime at 11453 microseconds and our Brute Force algorithm with a runtime of only 49 microseconds. The reason for this discrepancy could possibly stem from the program initializing memory when the program starts running, considering in our main class our Greedy algorithm is the first test we run. It is also possible that the reason for the first few iterations of the Greedy algorithm being larger than normal could be due to the fact that within the Greedy method a display method is also called within. Which could possibly be increasing the run time and giving unexpected results.

0-1 Knapsack Results:

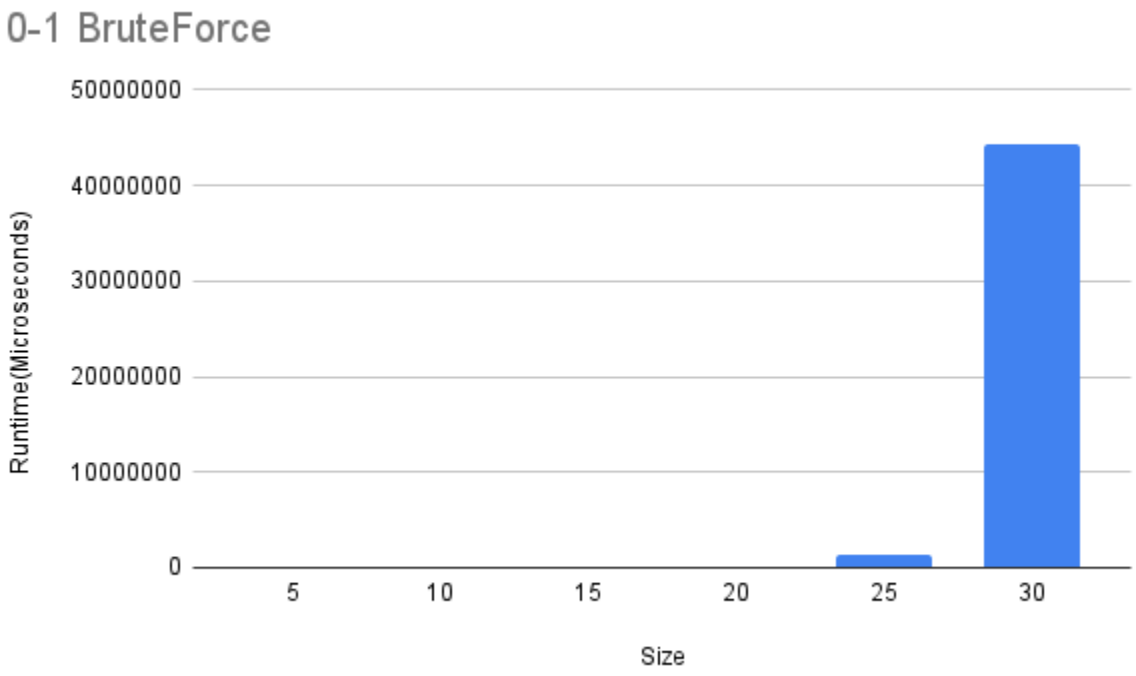

0-1 Bruteforce, Greedy and Dynamic
Bruteforce
Greedy
Dynamic
10000
7500
5000
2500
0
Runtime(Microseconds)
5
10
15
20
25
30
Size


0-1 Bruteforce, Greedy and Dynamic Full Y axis
Bruteforce
Greedy
Dynamic
50000000
40000000
30000000
20000000
10000000
0
Runtime(Microseconds)
5
10
15
20
25
30
Size


0-1 Dynamic
10000
7500
5000



as you can see, the runtime for the smaller size is large, but the faster with larger sizes



The Brute Force 0 - 1, we can see that the bigger the size we have for the input, the more time it needs to process the data. This is the worst way to solve a knapsack problem which figures out the best choices.

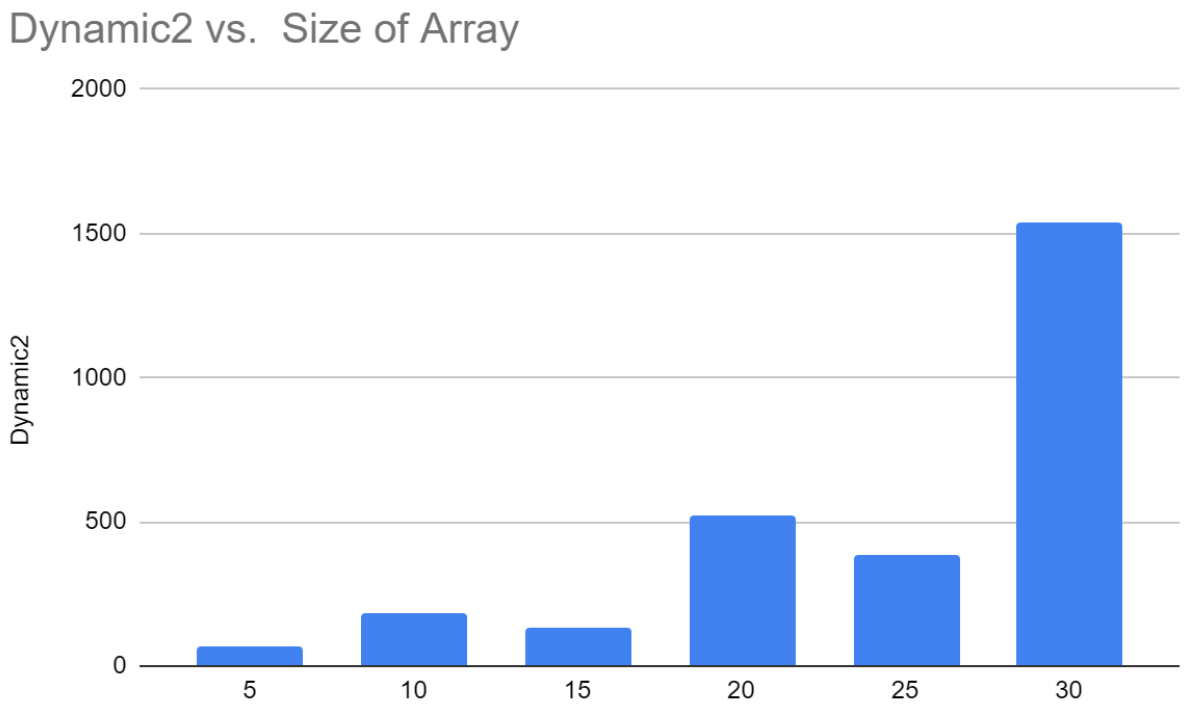
These are the graphs of our 01 knapsack experiment. In general, the results follow the theoretical runtime of the algorithms, with some exceptions. First, you can see that a brute force approach grows exponentially with the size of the input. It starts as the lowest runtime but grows bigger exponentially and eventually, the runtime is significantly larger than the other methods.

For the greedy implementation, we see a similar growth pattern with the array sizes. They stay pretty constant and are even the most efficient method for some of the knapsack sizes. This is the same theoretically because the greedy method is logarithmic while the other methods are not.

The last method is the dynamic programming approach. Like a lot of the other methods, the first input had one of the higher runtimes. This could be for a variety of factors, including program initialization, or wrongful implementation. The input size for the first and last follows the theoretical implementation, it is slower than the greedy approach but faster than the brute force method.

Updated by Robin

After further experimentation, the error in the first runtime may have been caused by a issue in the code initialization. To fix this, I ran the implementation of the dynamic programming solution twice, and found the results for the second block of input 1 solution to produce a runtime more in line of what was expected. Here is the chart for the updated experiment.



Overall, we found results that align with what should theoretically be. Our exception is with the first input with the smallest input size. It should be the smallest, but for a lot of the algorithms, it was the largest runtime or second largest. As stated previously, this can be due to a variety of factors, including program initialization or wrong implementation of the runtime. Other than that, we can see our results mostly align.

DID NOT IMPLEMENT DYNAMIC PROGRAMMING APPROACH FOR FRACTIONAL KNAPSACK