**Skewed By Toothpick Results on Comparing knapsack Experiment**

**Abstract:**  SBT created a program to test different algorithms for finding the optimal solution for filling a knapsack with various weights and values. To ensure accuracy of the experiment The program will use the same knapsack for each experiment and compare the run times. To account for various specifications among computer systems, the run times will only be compared with the system that the program was run on. The experiment will test two different types of knapsacks, the o1 knapsack that will only take the whole item, and the fractional knapsack that will accept partial items.

**Introduction:** Jackson, Max, and Justin have created a program to test the different decision algorithms that try and find the optimal solution for the two different kinds of knapsack problem. The first problem deals with the 01knapsack issues. This is when you are not allowed to split up the items or only take a part of an item. These items all have a unique weight and value. You can either take the items or leave the item. The second test is with the fractional knapsack. Unlike the 01knapsack we can split up the items to ensure we can achieve the most optimal solution.

**Methods:**  To implement the test, we created an array of knapsack problem objects that hold the items, max capacity, and test number. From there we would pass in the knapsack problem objects to our algorithms, first one being the brute force. This approach will iterate through each possible combination to find the most optimal solution. There is the possibility that this test will take a large amount of time for a large number of items. The second approach is the greedy algorithm which will retrieve the ratio of the value to weight and then take the highest ratio first followed by the second highest ratio and so on until there is no more weight. Our third and final method is a dynamic program that will continually compare the value of an item with the sack to see if it’s the most optimal. Details on each algorithm are listed in the results section.

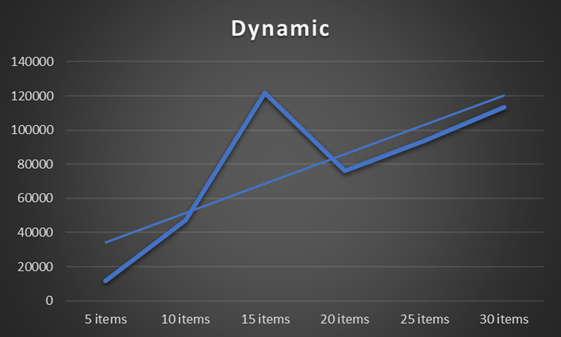
**Team SBT Roles and Responsibilites:**

1. **Jackson Kettel - Team Lead, Design Lead**
   1. **Created:** AlgorithmParent, TestResult, AlgtorithmTester, BruteForce01Knapsack, Greedy01KnapSack
2. **Max Cifuentes - Repository Administrator**
   1. **Created:** Fractional BruteForce, FractionalGreedy, Item, Knapsack, CSVOutput
3. **Justin Jemison - Quality Control**
   1. **Created**: DynamicKnapSack(includes both frictional and 01), KnapsackGenMain, TerminalOutput, Excel

**RESULTS**

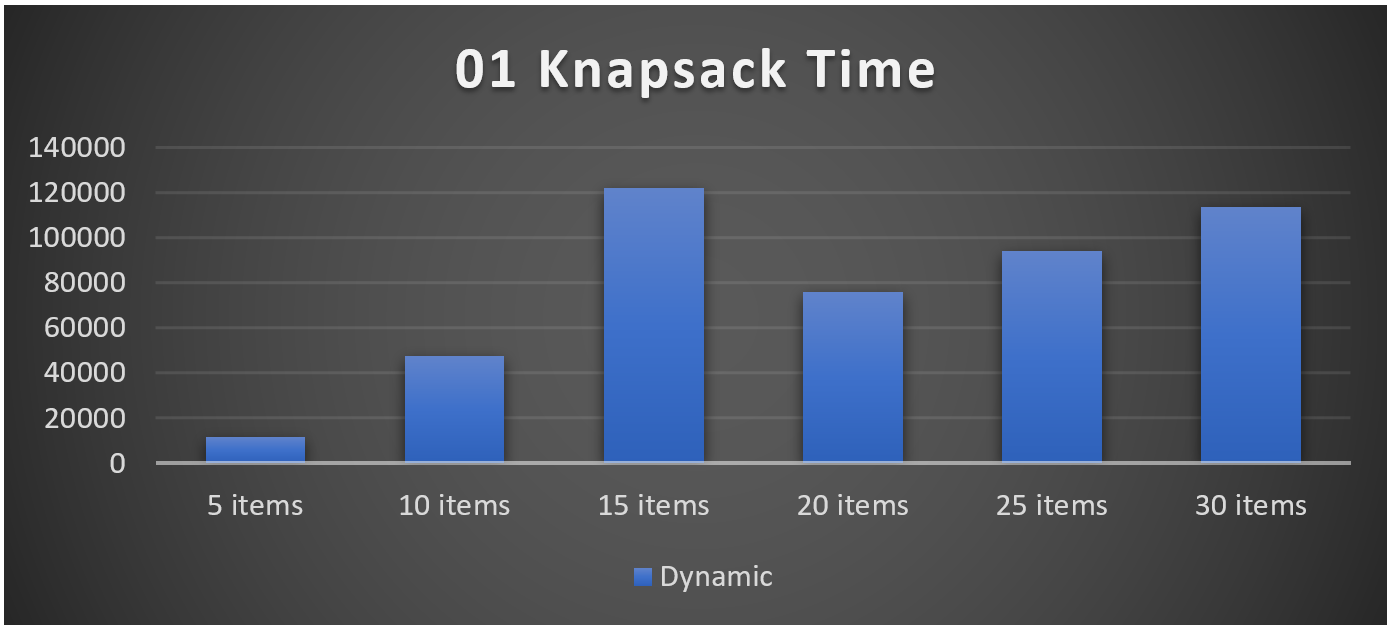
**Justin's Findings**:

Objective: I was responsible for creating the dynamic program(s) and testing them against the rest of the algorithms that would find the optimal solution to the knapsack problems.

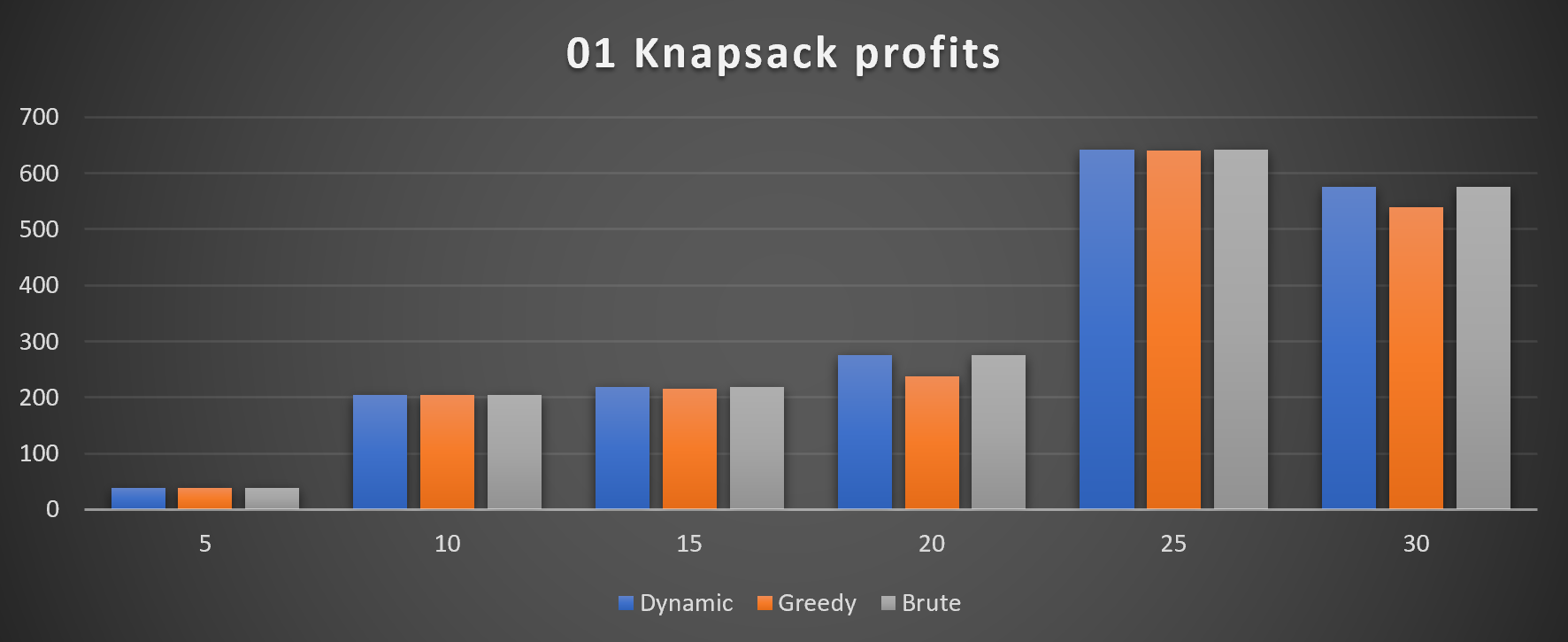
Test Environment details: The computer system that I tested and recorded results for my portion of the analysis were generated on a MSI laptop with 32 GB of RAM. The processor is an Intel i9-9880H 9th generation CPU with 8 cores running at 2.3GHz. All programs are created and run through JetBrains IntelliJ IDE with no other application running at that time. I ensured the power cable was plugged in to avoid battery saving optimization. 

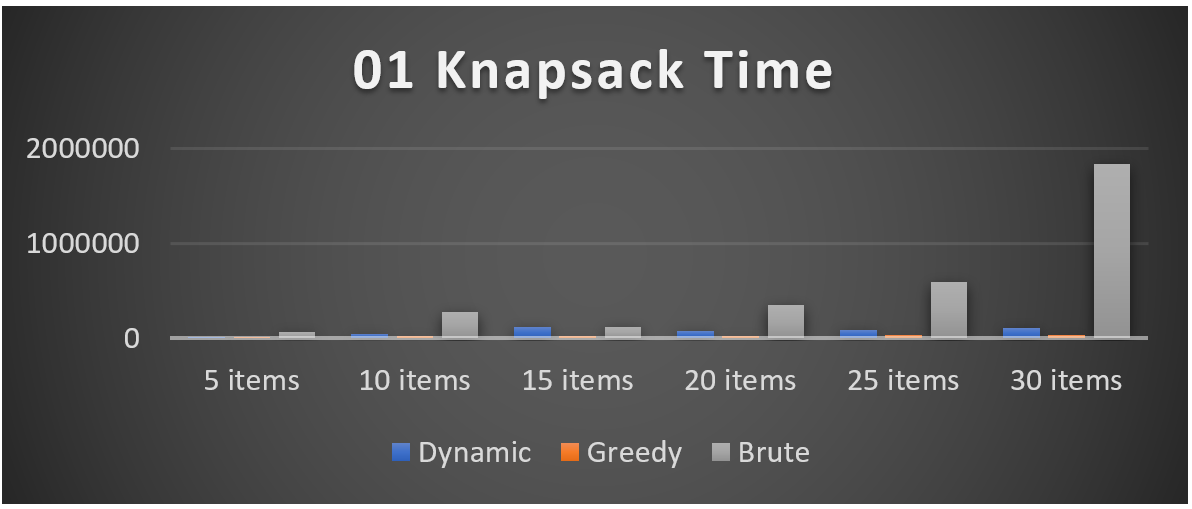
**Results:**

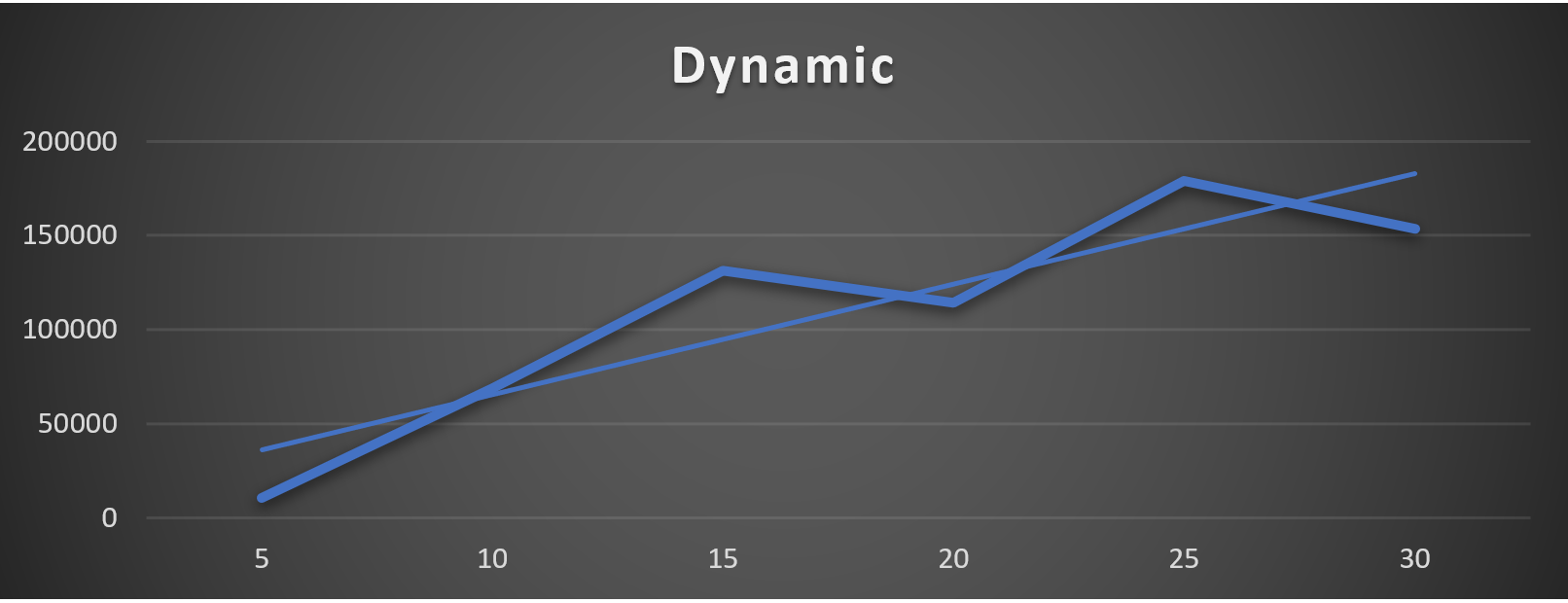
**Dynamic 01 Knapsack:**

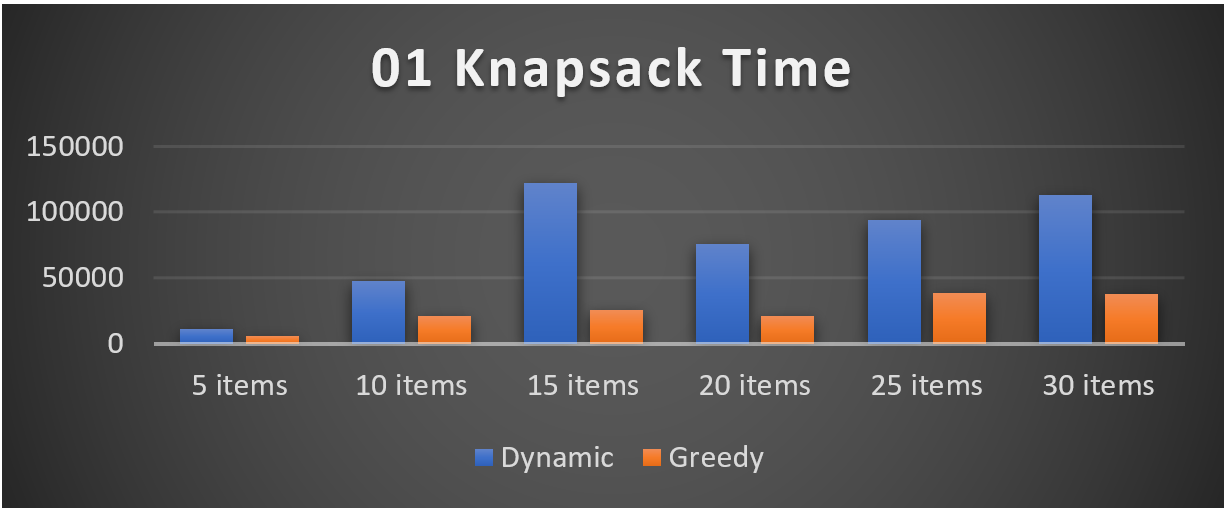
Below is the bar graph of the O1 knapsack time results. In the previous test, the 5 items were > 100000 microseconds. Then we added the “primer” to allow the test to instantiate the class and not skew the test results. As seen below the test results are now more expected. One thing to note is in one of the test, test # 3 with 15 items spiked due to some unknown factor. These random fluctuations in our test results make it difficult to actually compare algorithms. In the future, a test with averages would better represent the comparison of algorithms.

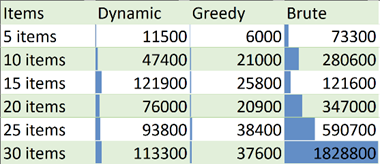
Running the test multiple times shows the same results with the same data since the first three tests are always the same.

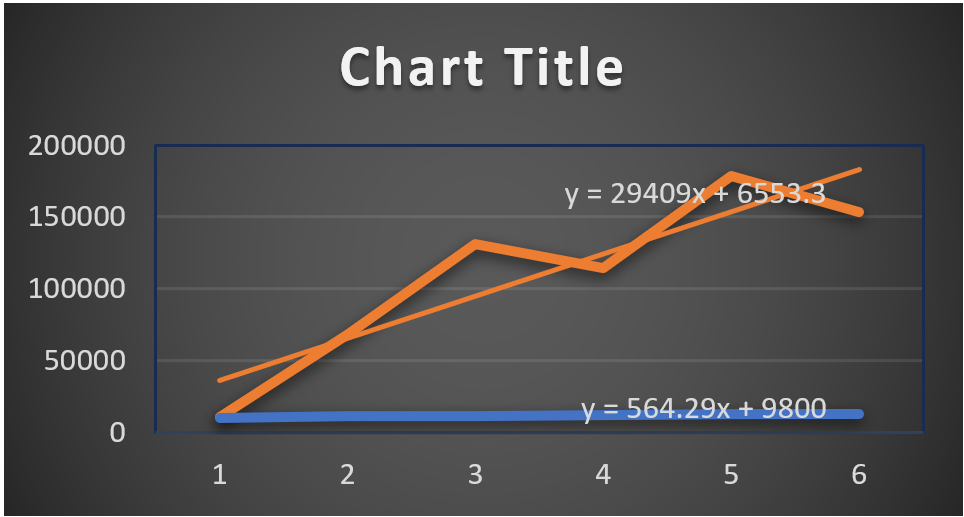
It was not until a few hours later that the test was reconducted and the results for the anomaly for item 3 went away. 

The dynamic algorithm shows a near linear progression as expected. The dynamic Big O is (n \* m) where (n) is the number of items and (m) represents the capacity of the knapsack. The current algorithm class “Dynamic” is actually (n^2) due to setting up the data for computation by the dynamic algorithm. This is due to the date being individual objects instead of an array format. 

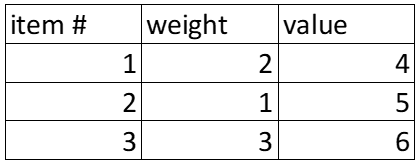
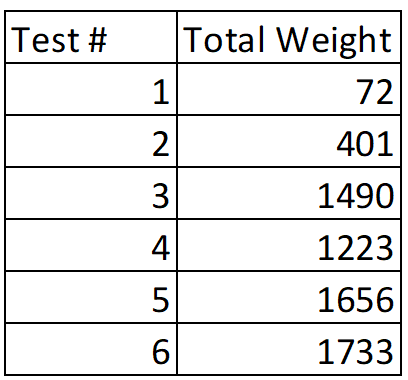


Comparing the dynamic program to the other programs we can see that it is far faster than the brute force approach. When compared to the greed algorithm we can see the greedy was consistently twice as fast as the dynamic program. The drawback is the greedy did not always choose the most effective combination of items. As seen in the 01 Knapsack profits table, the value for this set of tables was very close and for the 30 items it chooses the exact same items as the dynamic program in less than half the time. But there have been other trends with different sets of inputs that have shown a trend with more items the more inaccurate the greed program becomes. With that said. The accuracy vs process time is something to consider when creating a program that has a 01knapsack type of problem.

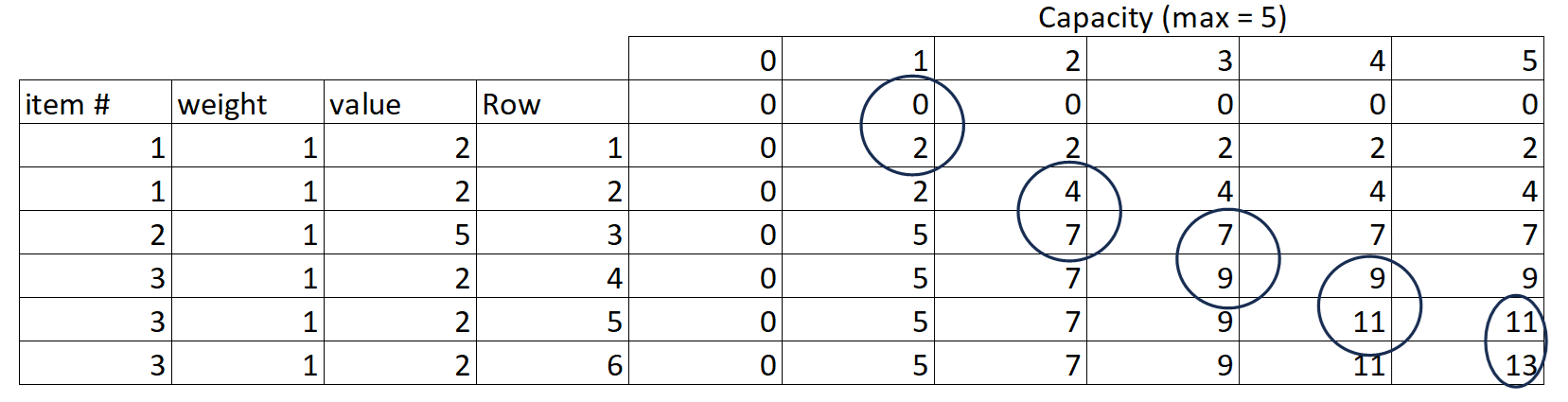
Creating the dynamic algorithm, we used two arrays to hold the values and weights and the (weight + 1) X (items # + 1) matrix. Then used the computation of x[row, column] = max{x[row – 1, column], x[row – 1, column – weight[row]] + value[row]}. In other words, we took the highest value from the row above, or from the current value + the value from the row above – the weight. Once the matrix was complete, we then iterated backwards from the bottom right value moving up trying to find where the value changes. Once we found that value, we then added it to the Items used. Then we would move left the amount of weight of the value we took and start the process over again. We would stop when we arrive at a 0, create the test result object and conclude the process. Note that the timer only captured the algorithm and the identification of the used items. The time to set up the test was not accounted for.

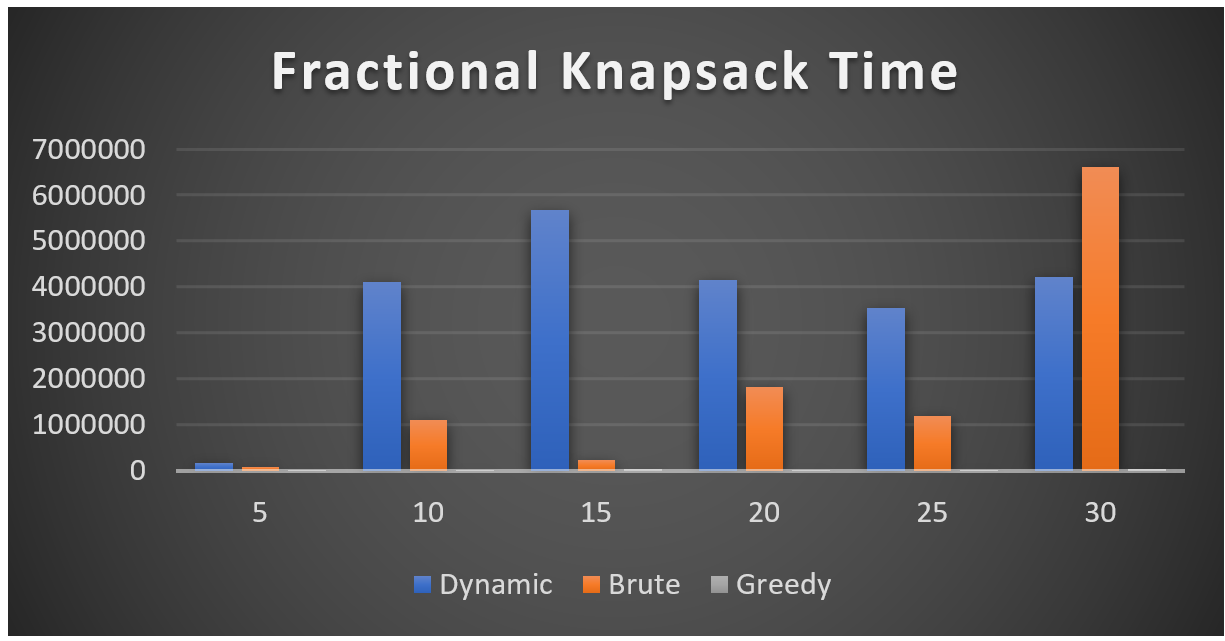
**Conclusion:** The dynamic program performed as expected against the other algorithms with it being exponentially faster than the brute force algorithm yet slower than the inaccurate greedy algorithm. Taking the weight and item numbers I computed the amount of matrix elements that would have been computed and plotted a line to compare with the actual line and we can see that they are extremely different. The ideal line has a slope of 564 while the actual line has a slope of 17209. As for the reason for this discrepancy I believe some of it has to do with the building of the matrix itself. 

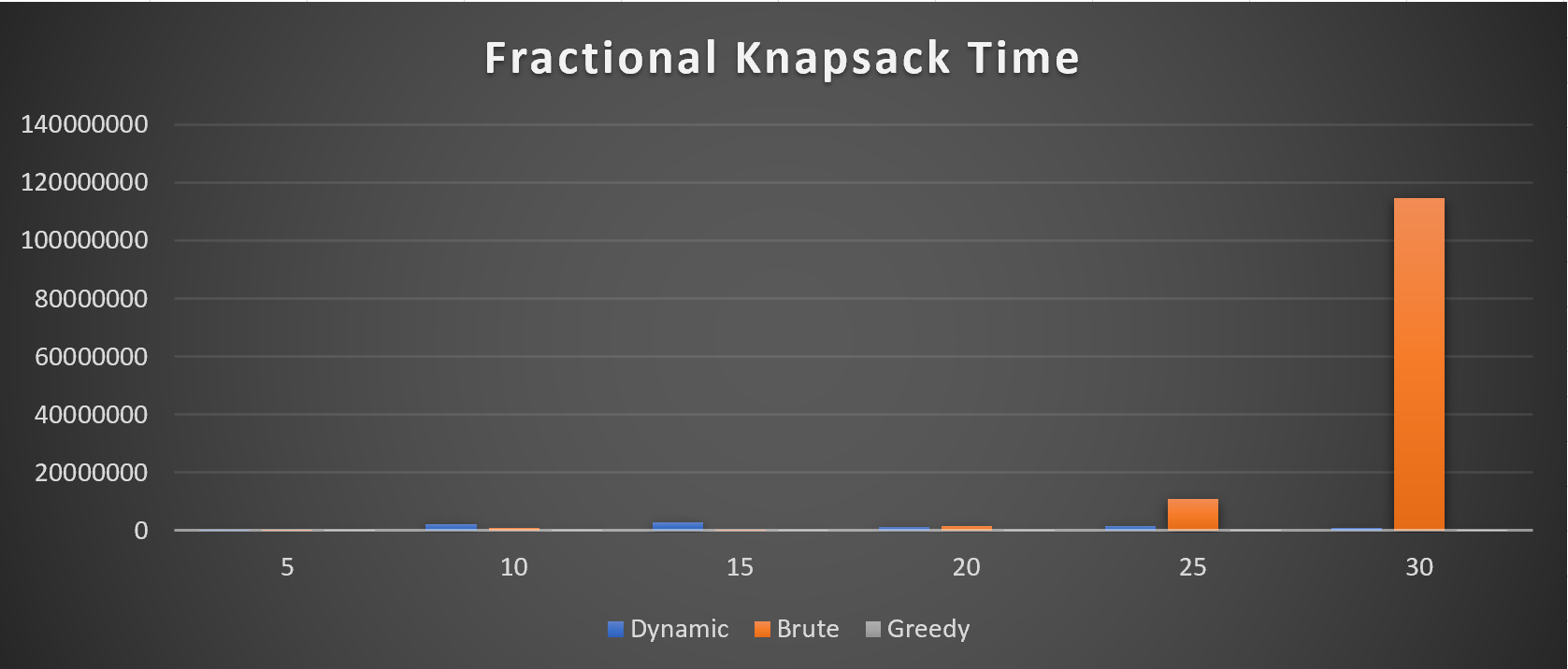
**Dynamic Fractional Programming**:

The dynamic fractional program utilizes the normal dynamic class and is activated with a “true” Boolean value when creating the class in the constructor parameter. The difference between the dynamic 01 knapsack algorithm and the fractional algorithm is the way it views the items. The fractional program will break down each item by its weight. This means that the matrix will be (combine items weights + 1 by capacity + 1). Reasoning behind this strategy is to find the most optimal combination of individual items to fit in the knapsack. Since each item can at most be broken down to a weight of 1, we can then conclude that by comparing all items with the weight of 1, we will find the fractional combination with the most profit. For instance, if we have the following items in the table to the left. To use the dynamic fractional algorithm, we can make each item the weight of 1 there by expanding the table's rows. To keep the value of the table the same we take the value/weight. Once we have the arrays, we can now conduct a dynamic 01 knapsack algorithm on this table. Only issue is to try and keep the item numbers straight. This is done by using a hash table that will account for which item is in what row. 

Our final dynamic table would look like this filled out. The final result is taking Items 3, 2, and only 1 of item 1.



As for performance, the fractional dynamic knapsack was not very consonant in linear performance. Also, the algorithm’s performance against the brute force algorithm was surprisingly poor. As seen in the graph below, the only time the algorithm was under 200 microseconds was the first test with only 5 items. This is due to the fact that the range of weight expands the matrix to a mixed range of numbers. Hence the reason that test 4 was faster than test 3. As for test 5 and 6, that is unknown. The table to the right is the total weight of all the items and the growth is not very consistent. This accounts for the lack of the linear progression of the program. The time Asymptotic equation for this program would be Big O ( n \* m) where n = the total weight and m represents the capacity of the knapsack.

Comparing this knapsack to the other algorithms, the greedy algorithm is far faster. As for the brute force. It is suspected that the way the brute force was implemented it ran far more efficiently than what was originally expected. This could account for the poor performance of the dynamic program against the brute.

As for the greedy algorithm, there is no comparison; it is barely visible in the chart due to scaling. This shows that doing a fractional dynamic algorithm is not very practical.

After a few adjustments to the brute force algorithm, it is starting to show a more accurate representation when the number of items increases, the time for processing increases exponentially.

**Conclusion:** The Dynamic algorithm for solving the fractional knapsack program was fairly successful but impractical since the greedy algorithm is far superior and produces the same results. It was surprising the unique way the algorithm progressed as the number of items increased. Since the random number generator was creating the weights between 1 and 101, the average weight would become the linear progression of the algorithm. Big θ(n \* (average number generated) \* capacity). With the worst case being Big O(n \* (101) \* capacity). That is still far better than the brute force (well the real brute force with the time complexity of 2^n). And lastly, Big Ω(n \* (1) \* capacity) where each item only had a single weight. At that point, Dynamic may compete with the greedy algorithm for speed.

**Max’s Analysis:**

**Fractional Greedy Algorithm-**

The implementation that I chose for my fractional greedy algorithm was fairly standard, and highly efficient for the fractional knapsack problem. The standard components of the greedy algorithm were followed, so the ArrayList of item’s were sorted in descending order from best ratio to worst ratio. Then, it was a simple matter of taking as many whole items as we can to fill our knapsack up as close to the max capacity as possible, filling in the remaining space with the remaining next-best ratio item. This method of solving the fractional knapsack was exceptionally fast, especially when compared to the Fractional Dynamic programming and even more so with the Brute Force Algorithm making it by far the best of all algorithms compared for the fractional problem.

When looking at the actual empirical results, the algorithm performed as expected by doing better than Fractional Dynamic or Brute Force. This is likely due to the timeliest part of the Greedy Method being the actual sorting of the items in the knapsack by the ratio. I used Java’s Comparator which uses a TimSort whose big O is n log , and also sorts in place rather than creating a new data structure. Since this is the costliest part of the algorithm, we can expect it to run at same Big O, which makes it much quicker than Brute Force’s O(2^n) or Fractional Dynamic’s Big θ(n \* (average number generated) \* capacity) which is still worse than O(n log n) that Greedy boasts. Taken together, and as our data shows, even as the size of the knapsack items increased, the difference in time that the fractional greedy took to run was for all intents and purposes, unnoticeable at time of execution due to its efficiency.

Additionally, a Greedy Method always returns the optimal solution because of the way it finds its solution. So consistently, across all three algorithms, it always returns the optimal solution. Therefore, this algorithm was extremely efficient with a low runtime and a guaranteed optimal solution in the fractional knapsack problem.

Lastly, there are additional algorithm-types to solve a fractional knapsack, such as linear programming where you set up the fractions of each item as variable, and work towards creating a linear equation to return the most optimal solution. Another strategy would be to use a divide-and-conquer approach that breaks apart the items into smaller successive groups until you build them back up by a form of item sorting like the greedy method until you can choose the items that best meet the constraints while returning the best possible solution. Additionally, as shown below, a brute force algorithm can also be implemented that searches through all possible combinations to find the best. Although all three of these are are alternatives to the greedy method, they’re all often considered worse since linear programming can go as far as an O(n^3), while the divide-and-conquer big O depends on the exact implementation of what sorting(s) are used (like merge sort) and what algorithm is implemented to sort the item themselves and brute force has an exponential growth rate with a big O(2^n) which our findings support since we did implement it ourselves.

*Author’s Note:* keeping track of what items were chosen by adding them in the dictionary we used to keep track of them for each algorithm would have also extended the runtime (a bit) because of the extra time required to perform those operations.

**Fractional Brute Force Algorithm**-

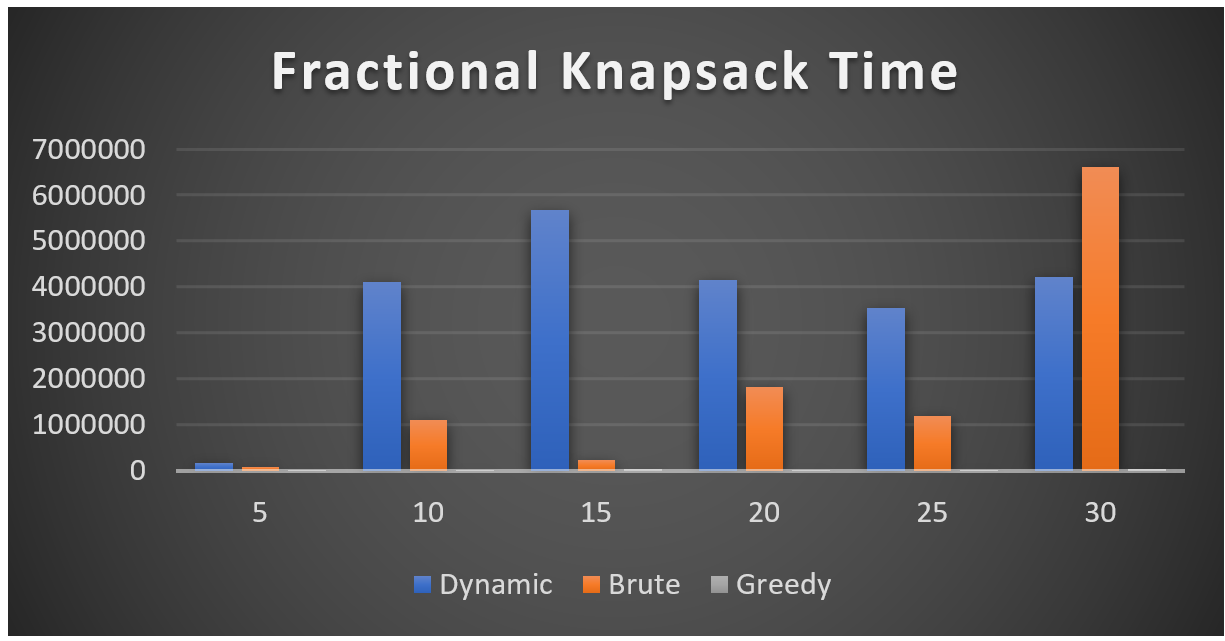
The implementation that I chose for this algorithm resulted in this author’s opinion, a lot of headache. There were not many (if any) pseudocode/writings that I saw for a brute force, and opinions differed on whether there could exist a fraction of an item (other than the last) or not. While I recognized that a fractional brute force algorithm would see us exploring every possible option of combinations, when coding, it took a lot for me to *not* take methods that would take a short approach. My first, and the approach I stuck for the longest with, was a recursive method that would consider different avenues for adding in an item, and considering which would be the best to add in. My algorithm wouldn’t consider options that were *greater* than the max capacity, which was an optimization that prevented a true 2^n runtime, since my algorithm would have cut out possible combinations that wouldn’t even work based on the constraints.

In the end, I did come up with another implementation after focusing on creating an implementation closer to 2^n, but it did not account for breaking individual items into fractional components, and then bringing them back together as mentioned in the Slack announcement. Without doing this part, it would become a pseudo-brute force/greedy since my approach only explored the last possible item in each combination as a fraction, while finding the different possible whole item combination) using bit masking/bit manipulation (Cool stuff! Since each item is either in or not, it can be represented as a 1 or 0 to signify whether a given item is in a particular combination, itself based on bits!), which was necessary since creating and saving the different combinations caused memory errors for me, requiring me to research alternatives. However, it is worth nothing that even this approach alone did take quite a bit of time, as creating and comparing the different items as 2^n requires a lot of different computations. In one of the charts attached below, it becomes extremely evident, even with this slightly-off approach, the exponential growth rate becomes heavily apparent as the size of the items in the knapsack increases, shooting past dynamic and greedy in runtime.

While the runtime was initially comparable to dynamic programming at smaller sizes, such as 5 and 10, it became exponentially worse the greater the size of the possible items in the knapsack. This is directly in line to what would be expected since an exponential algorithm growth rate would rise extremely fast, and deviate quickly from other algorithms as the size n increases. The Brute Force O(2^n) was far worse than the Greedy Method’s O(n log n) because the Brute Force had to constantly find new combinations, and update them with the best one as they came in.

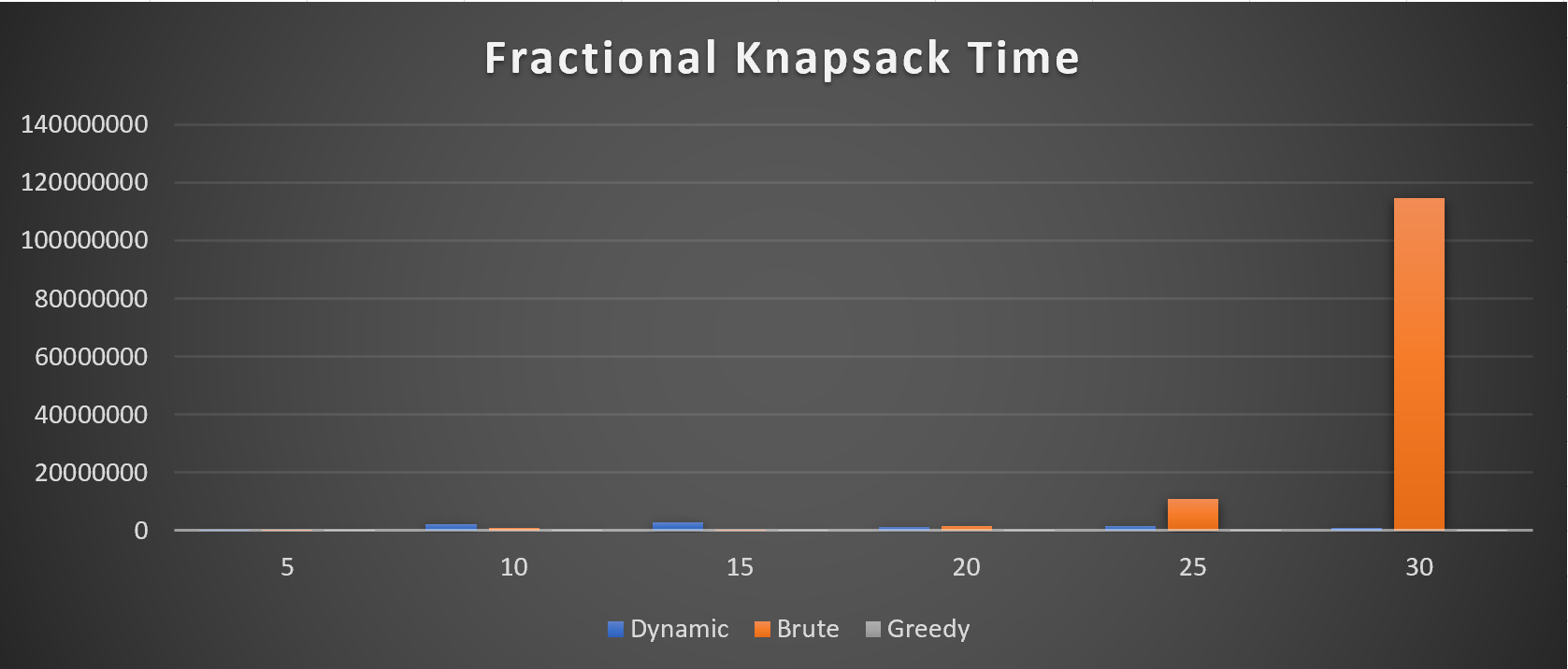
Although the brute force algorithm does take quite a bit of time, we would also expect it to similarly return the optimal solution just as the greedy method does. This is because although it is searching through many different combinations, it is constantly looking for the best combination and updating them accordingly. Since it is searching through all combinations, it should theoretically eventually arrive at the same combination that the Greedy Method would have also deemed the best, thus returning the same profit albeit at a much worse runtime. To best showcase how much worse the Brute Force Algorithm would be when compared to Greedy/Dynamic, the iterative implementation (closer to 2^n) that was ultimately merged into the repository, and provides data much closer to what could theoretically be expected, including by having the same profit values across as the other algorithms. I did leave my recursive approach available in the code even though it wasn’t actually used, because we did some of our initial write ups on that as well.

*Author’s Note:* Although one might think that the 01 Brute Force would have a similar runtime to the Fractional knapsack, the additional aspect of having to constantly look through remaining items that aren’t in a given combination and find which item when taken partially returns the best value for that combination adds an additional layer of complexity that drives this algorithm’s runtime right up. In a correct implementation, you would have to “build back up” an item, in a manner not too unlike a dynamic implementation.



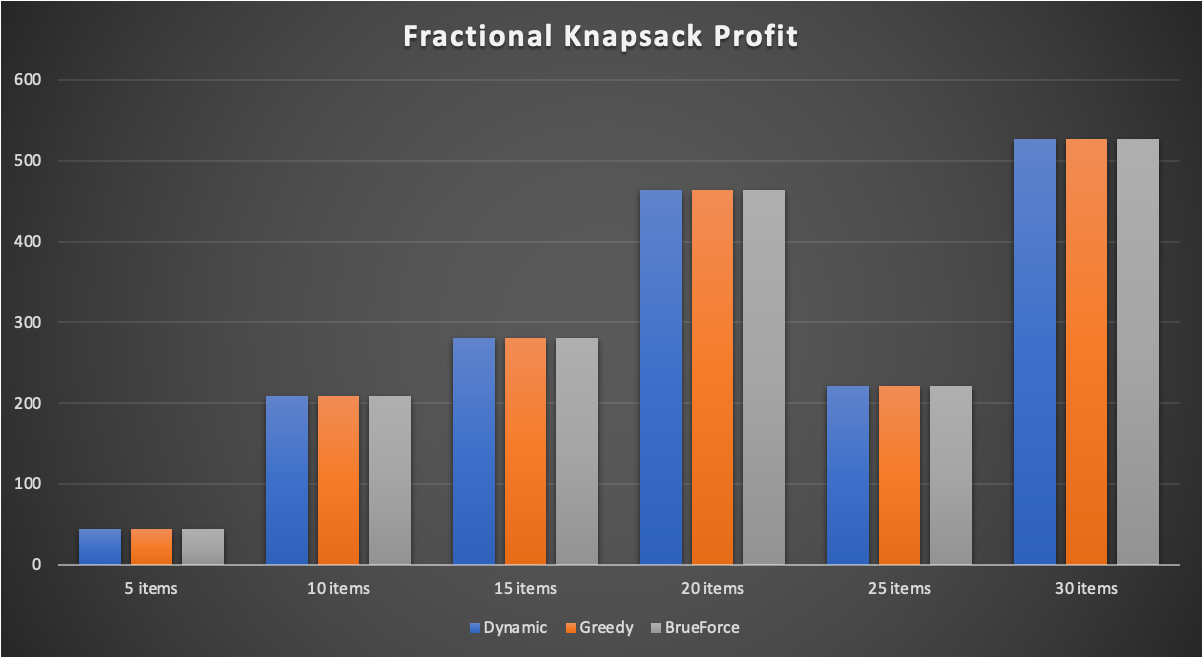
Graph depicting the originally implemented recursive method

Note: Brute Force got much worse at size 30



Graph depicting the bit-iterative approach, closer to what you would theoretically expect

Note: Brute Force is shown to perform extremely worse in runtime with higher sizes



Graphs depicts the total profit/value of each algorithm

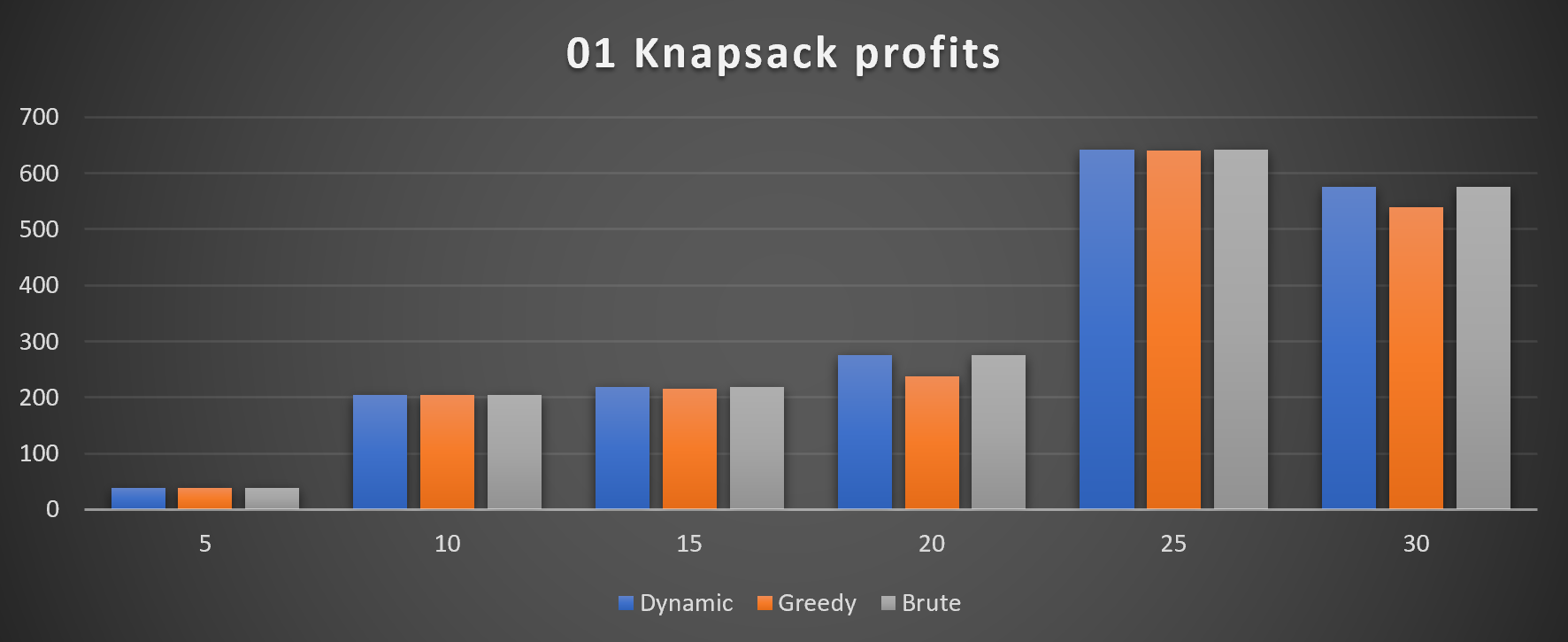
Note: despite the different runtimes, all fractional algorithms reach the same profit as expected

**Jackson’s Analysis**

**Knapsack 01 Greedy Algorithm-**

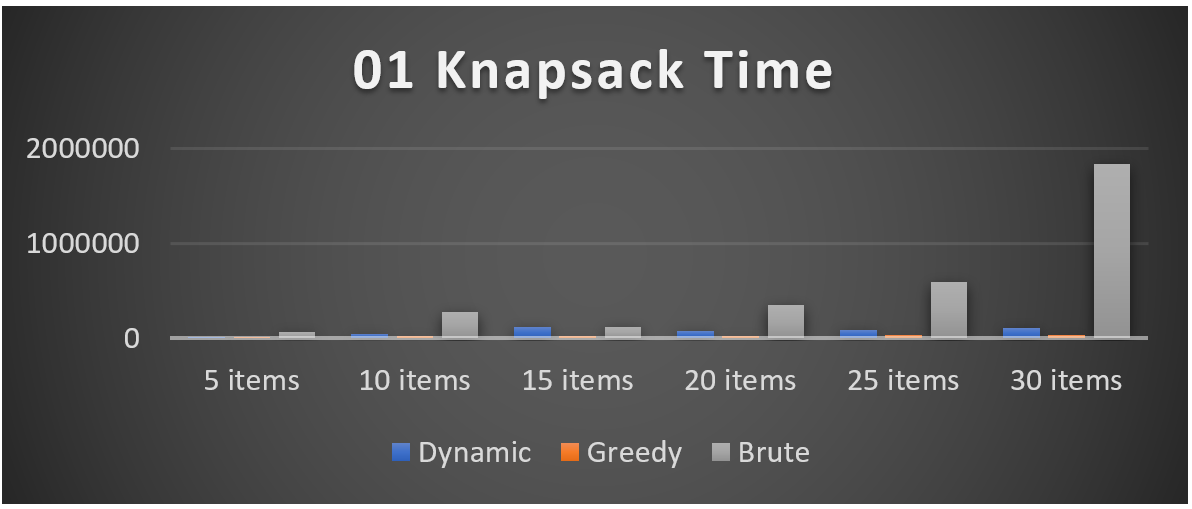
This section details the standard implementation of the greedy algorithm for the knapsack 01 problem. What it involves is taking all the items in the knapsack, sorting them based on their value to weight ratio. By doing this, you order the items in a way that the best value items are placed at the front of the list, and the worst value items are placed at the end. We use a Comparator to sort the items in the ArrayList in O(n log n) time. After the ArrayList is sorted, you then proceed to add each element into the result ArrayList until no more items can be added without exceeding the maximum capacity of the knapsack. This method, while incredibly fast, does not always yield the best results. This is because some items that are not of the higher value to weight ratio may generate more value for the knapsack despite not having a high ratio.

From a theoretical standpoint, the greedy algorithm for this should be O(n log n). This is because in order to sort the ArrayList of Item objects, you must use Collections.sort() method in Java, which utilizes quick sort. This sorting algorithm should take O(n log n) time to perform, as well as adding the items to the knapsack, which is another O(n) operation. Compared to the two other sorting algorithms (brute force and dynamic programming), this algorithm should be the quickest in practice as it scales well with large amounts of items. In the empirical results we got from our experiment, we see that in fact the greedy algorithm outperforms the other two algorithms in all input sizes. The results also show the time increases linearly, unlike the brute force algorithm. The empirical results we got line up exactly with what we expected.

However, while the performance of the greedy algorithm was better than that of the other sorting algorithms, the total value generated using the greedy algorithm did not always match the optimal item selection. On a few of the runs, it happened to get the correct answer, but on roughly half of the trials it gave a result less than the result of the other two sorting algorithms. This lines up with what was expected, because the greedy algorithm will not always produce the optimal solution. This is because it sorts the items in the ArrayList by value to weight ratio, then adds them in that order. While this might initially seem like the optimal solution, there can be items excluded from the knapsack that would have maximized the value of the knapsack. For example, there could be one pound left in the knapsack, but if the next item to be added exceeds one pound, the item is not added even if there exists a one pound item further down the list.

**Knapsack 01 Brute Force Algorithm-**

Implementing a brute force solution for this knapsack problem was tricky, as it needed to exhaust all possible item combinations. In order to achieve this, I utilized a recursive approach in order to be able to check every single possible combination, and return the value it produced. As such I would need to use a private recursive method to perform this.

For the private recursive method, I passed in the knapsack object, the maximum weight of the knapsack, the number of items in the knapsack, and an empty ArrayList of items to store items that would be added to the knapsack. In the base case, I check to see if the number of items is 0 or the weight capacity is 0. Both of these imply that you have no more items to check in this branch, or the capacity of the knapsack has been reached. In this case, you return a 0. If neither of these conditions are met, you pull out the item based on the number passed in by the recursive call. If the item is not able to fit in the bag, you make a recursive call to check the next item to see if it will fit. If the item can fit, then you create two ArrayLists: One with the selected items and the current item, and one with the selected items but not the current item. The program then makes a recursive call to see which of these ArrayLists returns the maximum value. If one of them produces a greater value than the other, we know that combination of items produces a greater value, and we update our selected items ArrayList to reflect that.

Through this process, the optimal value and items selected will be returned, as all possible combinations will have been compared. This implementation, while accurate in its results, is very inefficient in terms of time complexity. Because we must check every combination, the time complexity of this algorithm would be O(2^n). As the number of items increases, the amount of time the algorithm takes to complete should increase exponentially. In the empirical evidence we collected from our tests, we can see that this is the case. Compared to the other two algorithms, this one performed the worst in terms of time, and as the input increased in size, the time it took for the algorithm to execute increased exponentially. This goes to show that this is not the optimal solution for solving this problem.

The total value produced by the brute force algorithm matched the expected maximum value. This is to be expected, because after checking all possible combinations, it should theoretically find the best solution. While not the most efficient way to solve the problem, this algorithm still came up with the correct solution.

**Design-**

In designing a class structure for our project, I wanted to make it so the flow of information was well encapsulated and accounted for. Since this project had quite a few similarities in terms of scope to our previous project, I thought keeping a similar design philosophy would be smart. As a result, I divided the main function of the program into three foundational classes: KnapsackGenMain, AlgorithmTester, and the two Output Classes.

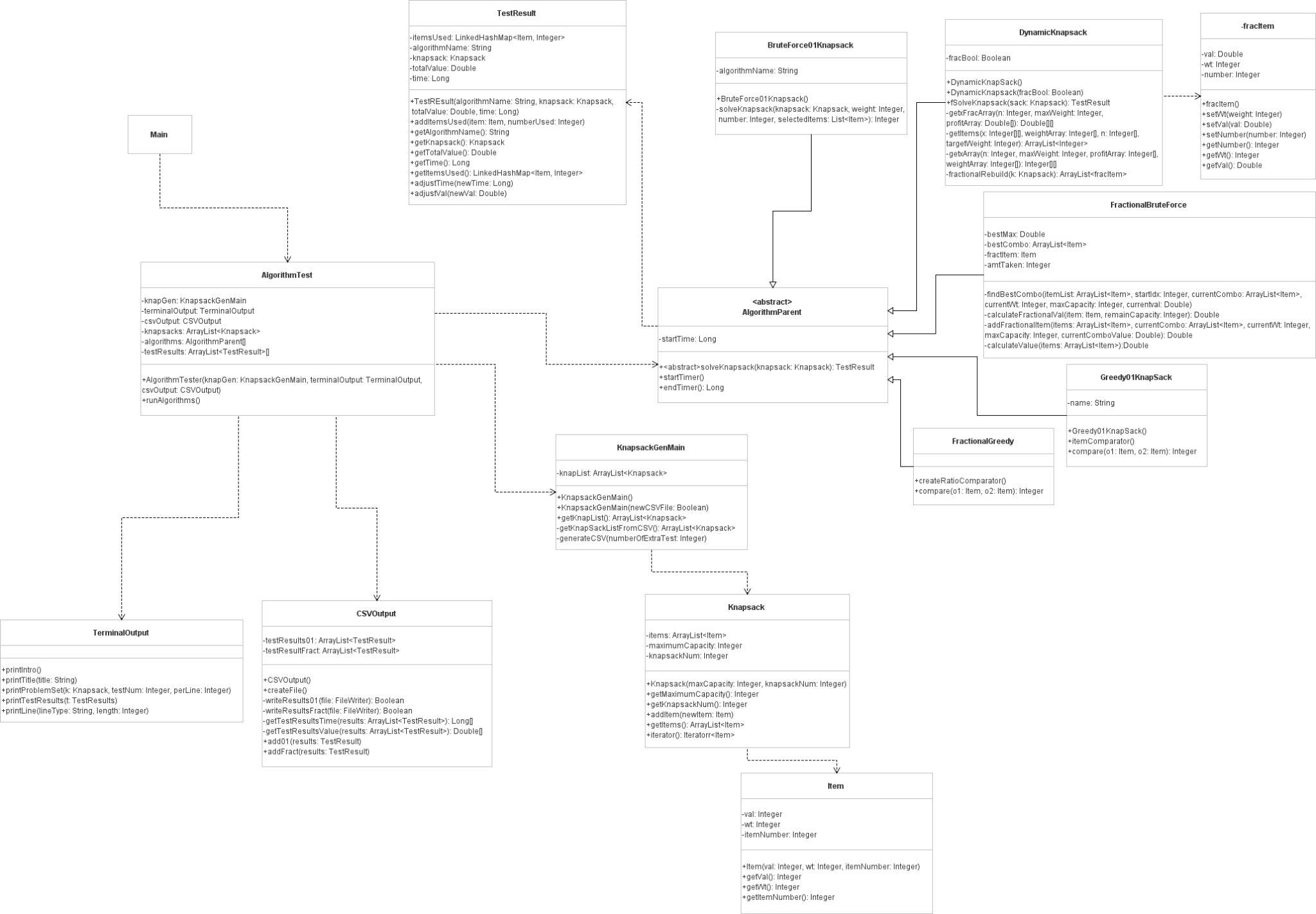
The KnapsackGenMain class serves to generate knapsack objects containing items from excel spreadsheets. Encapsulating this function into one main class helps to organize the flow of data in the program. I decided to make the knapsack and its items into their own classes, because this helps to encapsulate the data, and makes passing it from class to class more intuitive. A numbered knapsack object holding an ArrayList of items is much easier to deal with than passing multiple different variables and data structures separately and would prevent data from being improperly represented.

The AlgorithmTester class is responsible for performing tests with algorithms, storing the results, and requesting those results to be output. This class controls the flow of data and ensures that the program performs all necessary functions. First, it must get the knapsack objects from KnapsackGenMain. After it gets these knapsack objects, it will then request for each algorithm class to perform an algorithm on the knapsack and return the results. Each algorithm is encapsulated into its own class, and each algorithm inherits from the abstract class AlgorithmParent. We decided to make this class abstract because it will not be instantiated, and it has some methods that all its children should inherit. These methods include an abstract solveKnapsack method that must be implemented in each child class to return the results, as well as timer methods for recording the runtime of the algorithm. When the algorithm classes are finished running, they will return a TestResult object that stores all the data from the run. One of the issues we had in our last project that caused some issues with the results had to do with data not being properly stored and labeled, so it required us to do lots of debugging to find out which piece of data came from which sorting algorithm. Using this method of encapsulation, we were able to pass results with metadata and ensure the correct output.

Lastly, the two output classes (TerminalOutput and CSVOutput) are responsible for outputting the results of the algorithm tests. Partitioning these responsibilities to separate classes felt appropriate, as they are both trying to accomplish different tasks. The TestResult objects generated by AlgorithmTester are passed through methods into both output classes, and then processed to present the data either in the terminal, or in .csv format. In the terminal output, the data is presented in order of what type of algorithm it is (01 or fractional), whereas in the .csv file, the data is presented in a table and in graphical output. Both representations allow us to get a good understanding of what our results were.

Through breaking this program apart into three main groups of functionalities, it made the implementation of the program smoother, as well as making it easier to understand conceptually. While the details of each class can be a bit complicated, having a good idea of how information would flow when beginning our project would allow for a smooth development process. Focusing on encapsulating data, and labeling it accordingly also was helpful in making sure the results were accurate and debugging was easier.

**UML**

****