**PROJECT 2 FINAL REPORT**

**Team Maroon 4**

Project Manager: Edale Miguel   
Design Manager: Rojee Koju   
Development Manager: Kelvin Rajbhandari   
Version Control Manager: Rory Hackney

**01 Knapsack Problem**

Brute Force : Kelvin

Greedy : Kelvin

Dynamic Programming : Edale

**Fractional Knapsack Problem**

Brute Force : Rory

Greedy : Rojee

**UML :** Rojee

**Data/File Reading :** Rory

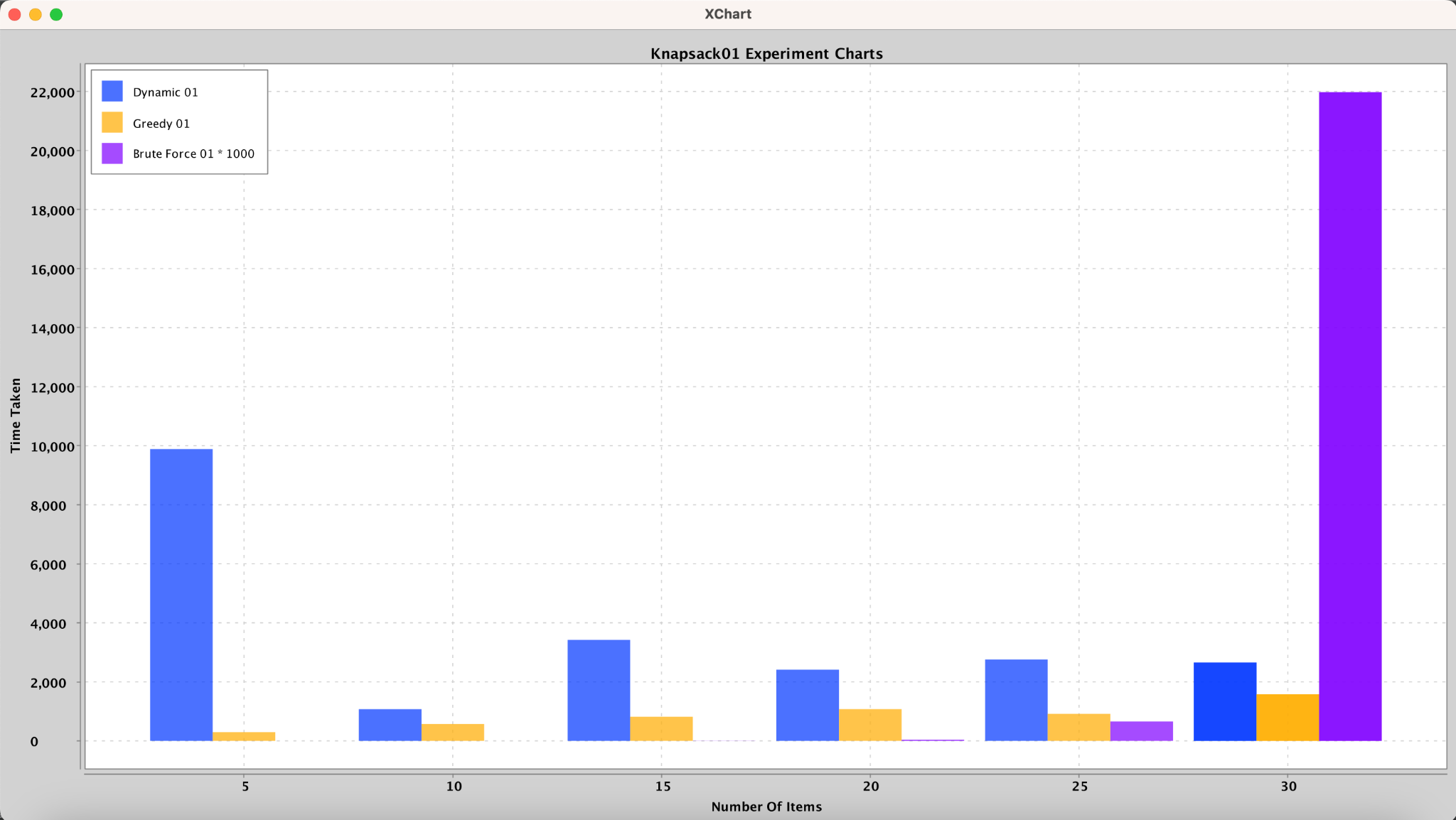
**Charts :** Edale

To run the charts, you need xchart. Download the jar file from <https://knowm.org/open-source/XChart/>.

Then, in IntelliJ, go to Project Structure > Libraries

Click the + icon and search for knowm.xchart, select the most recent version of xchart.

You should now be able to run the charts

****

**01 Knapsack Problem**

**Brute Force**  
Time complexity for Brute Force is О(n x 2n).  
Results are as expected, Brute Force is the algorithm/approach that should take the longest to iterate through as ALL possible combinations are tried. As the knapsack size increased, runtime increased exponentially and knapsack 6 took way too long, we had to stop the run.  
So when first gathering thoughts on how to implement the Brute Force for the 01 Knapsack Problem, it seemed fairly straightforward. Initial idea was to just go ahead and use nested loops which sounded simple enough. However when implementing in code, things changed real quick as the simple-minded approach of just using nested loops deemed complicated using excessive amounts of auxiliary data structures. That was quite a struggle. Decided to use an auxiliary array of bits, 1 for item added and 0 for not added, which would indicate if a new best combo was found and replaced the previous best result as the algorithm ran.

**Greedy**Time complexity for the Greedy Method is О(n x log(n)).  
Empirical results were not exactly aligned with theoretical results which was quite interesting to find. The process for Knapsack 01 Problem’s Greedy Method was fairly straightforward. Calculated the ratios as needed and used those values to sort the array of Items in the Knapsack in decreasing order. Given that both Knapsack Problems have Greedy Methods, we then moved the ratio calculation out to the Item class by implementing comparable. We could then leverage the compareTo( ) method as needed in both Greedy Methods for 01 and Fractional Knapsacks to sort the array. Using the now sorted in descending order according to the ratio array, items are added into the Knapsack until we face an item which would exceed the Knapsack’s capacity if added so we stop there. Now we have our Knapsack filled as much as possible with items of the better ratios.

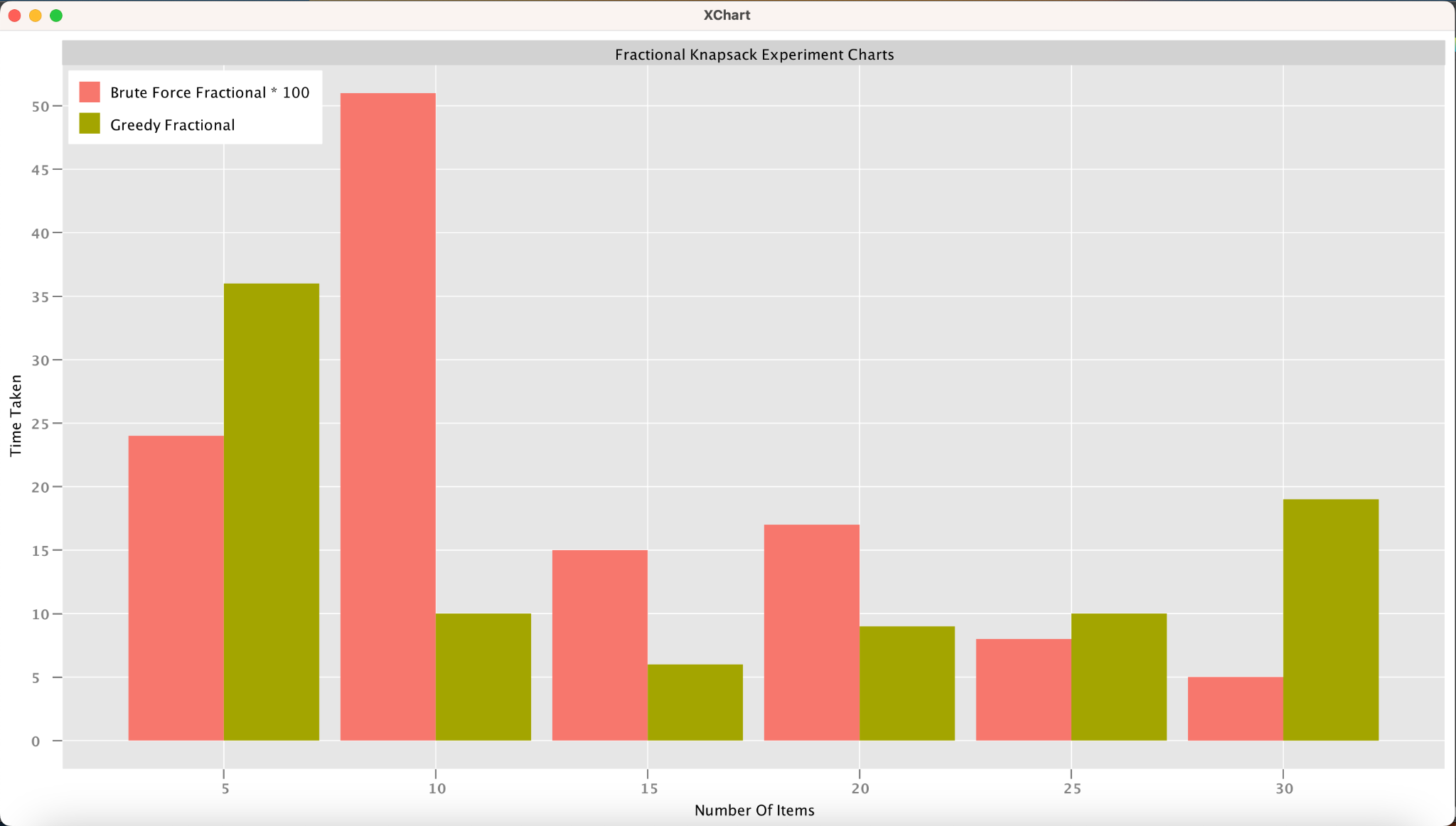
**Dynamic Programming**

When I was assigned the task to implement the Knapsack01 Dynamic algorithm, I felt overwhelmed and unsure where to begin. I struggled to grasp how it works and felt lost. However, through extensive research and watching instructional videos on dynamic programming, I started to understand and have more clarity on how to implement it. Dynamic programming involves breaking down complex problems into smaller, more manageable subproblems, which was the approach I needed to take. The algorithm aims to maximize the total value of selected items while not exceeding a specified weight limit, or knapsack capacity.

The process starts by initializing a matrix with zeros and then systematically filling it in, considering various combinations of items and capacities. At each step, the algorithm calculates the maximum value achievable with or without including the current item in the knapsack. The decision to include or exclude an item depends on which option yields a higher total value. By evaluating these options for each item and capacity, the algorithm determines the optimal combination of items to maximize value while staying within the knapsack's weight limit.

In terms of performance, the efficiency of this algorithm relies on the number of items and the knapsack's capacity. The time complexity is denoted as O(n \* W), where 'n' represents the number of items and 'X' signifies the capacity of the knapsack. Also, the space complexity is O (n \* x) due to the 2D matrix used to store intermediate values.

From running tests and theoretical understanding, Dynamic Programming is the approach which would be the most efficient out of all the algorithms. Especially when it comes to large sizes of input/data. If input was of smaller sizes, the Greedy Method would be ideal. Brute Force should never be used in real industry practices!!!



**Fractional Knapsack Problem**

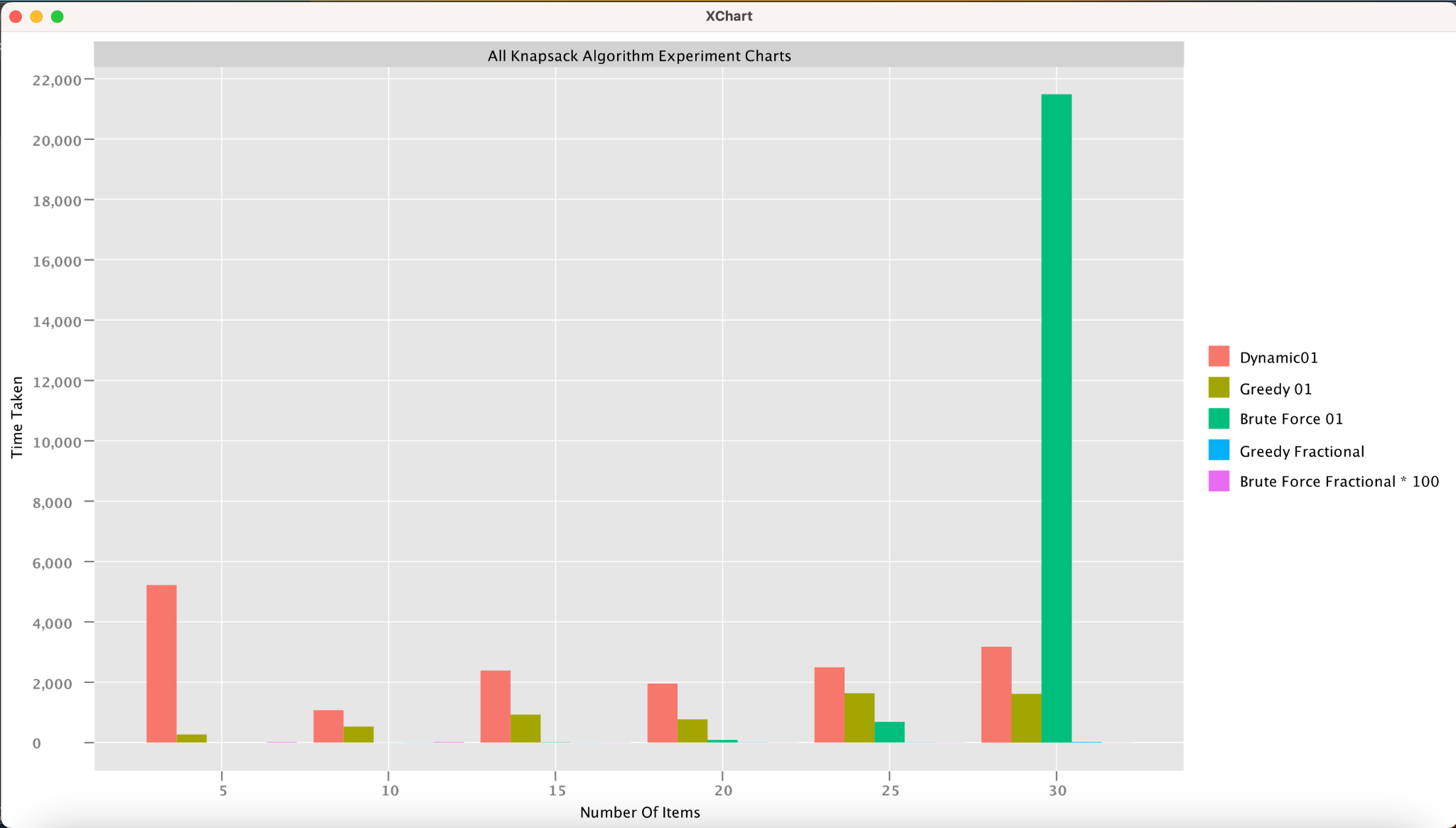
**Brute Force**

The brute force approach was more complicated than we thought. We had to compare all possible fractions of every item to find the maximum profit, and went through several attempts before a last minute hint helped us figure out how to approach the fractional portion. Brute force must compare all possible subsets instead of using a greedy approach to speed things up, so this took the longest time to run which can be seen on graph. We used a recursive function that explores all possible combinations of fractions of items to find the optimal solutions. The base case checks if the current index is negative or if adding the current item exceeds the knapsack capaity. If so, the current profit is returned. If not in the base case, the function considers excluding the current item and moving to the next index, compared to adding various fractions of the current item to the knapsack and recursively exploring the consequences. The loop runs from 1 to 100 representing the possibility of adding parts out of 100. The final result is the maximum profit achievable by considering all possible combinations of items and their fractions.The time complexity for Brute force is O(2^n). From both theoretical understanding and results (brute force taking several minutes per knapsack), the brute force solution is much slower than greedy, and should not be used if possible.

**Greedy**

The greedy approach for the fractional knapsack problem involves selecting items with the highest total benefit, ensuring that the combined weight does not exceed the knapsack capacity, W. To achieve this, we calculate the value-to-weight ratio (value/weight) for each item and arrange the items in descending order based on this ratio. Subsequently, we add items to the knapsack starting with the highest ratio until the capacity is fully utilized, incorporating whole items whenever possible.If an item can fit entirely, it is added whole but if the knapsack is not completely filled, a fraction of the item with the highest remaining ratio is added to maximize the overall benefit. This strategy optimally utilizes the knapsack capacity to obtain the maximum total value.The time complexity for this knapsack is O(n log n). This is the fastest algorithm that can run because it does not have to go through all possible choices.

**Graph for all Algorithm**



Creating the chart was really challenging. Since we are not using an interface, We had issues with how to implement it and how to connect it to each package having 2 mains.

After collaboration with the team, since we needed to compare the chart for the Fractional Knapsack Algorithm separately with the chart for Knapsack01 Algorithm, we created a class chart for each knapsack package, then call it in the main, and it worked. We also applied the same idea to the Charts package, so we could also print out all 5 algorithms.

After that, we had issues with the bar chart printouts. Since some of the algorithms take 0 ms and some algorithms are slow, the runtime isn't shown in the bar graph. We tried to implement annotation in XCharts, so the run-time would show on top of the bars. However, it didn't work. So to make the chart look better, we made adjustments to the units in run time for the Brute Force Algorithm and indicated the change in the bar chart legend.