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/ and place it into the project folder.
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

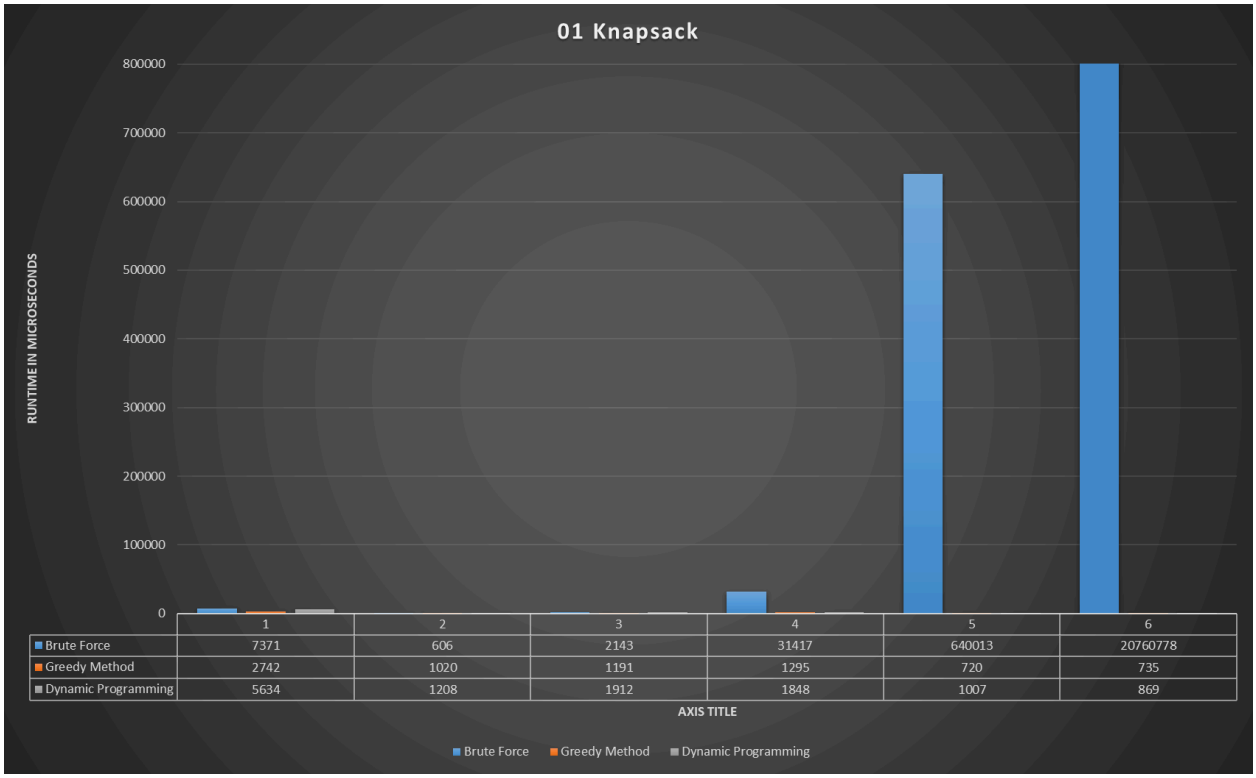
ATTRIBUTION FOR AMENDMENTS MADE, 2ND OPP

Rory : Removed old commented code for Fractional Knapsack's Brute Force & polished up final report's Fractional Brute Force.

Rojee : Cleaned up UML & polished up final report's Fractional Greedy Method.

Edale : Polished up final report's 01 Dynamic Programming.

Kelvin : Fixed 01 Knapsack's Brute Force, removed old commented code and polished up final report's 01 (Brute Force + Greedy Method).



	1	2	3	4	5	6
Brute Force	7371	606	2143	31417	640013	20760778
Greedy Method	2742	1020	1191	1295	720	735
Dynamic Programming	5634	1208	1912	1848	1007	869

01 Knapsack Problem

Brute Force

Time complexity for Brute Force is $O(n \times 2^n)$.

Brute Force is the algorithm/approach that should take the longest to iterate through as ALL possible combinations are tried. This exploration also guarantees that the optimal solution is found.

Empirical results are almost as expected. As the knapsack's size increased, runtime increased exponentially as knapsack 5 took significantly longer than knapsack 4 and same with 6 compared with 5. However there was an anomaly in the empirical findings between knapsacks 1 - 3. 1 took the most time out of the 3 despite being the smallest knapsack. Runtime data produced for knapsacks 4,5 and 6 were as expected, making a significant leap from the smaller knapsack to the bigger one. I would expect the trend to remain as such if there were more knapsacks to solve with increasing sizes. The advantage for using Brute Force would be that it guarantees an optimal solution. Disadvantage is that using Brute Force is unsuitable for large problem sizes due to exponential time complexity which makes it impractical for real-industry applications. Not scalable.

When implementing the Brute Force for the 01 Knapsack, recursion was the approach to explore all possible combinations and update an auxiliary data structure called bestChoice. It keeps track of the best combination which would be replaced by a better combo (if found) as the recursion progressed. In the recursion, an array of bits called itemInKnapsack exists to indicate if an item was to be included in the current combination being worked on. 1 for inclusion and 0 exclusion. Once the recursion was complete and the bit representation for the best combination

was located, items were added into a List of Items called the bestCombo in accordance with bestChoice which signals if an item is to be added into the optimal combination. We are now left with the optimal solution to the knapsack.

Greedy

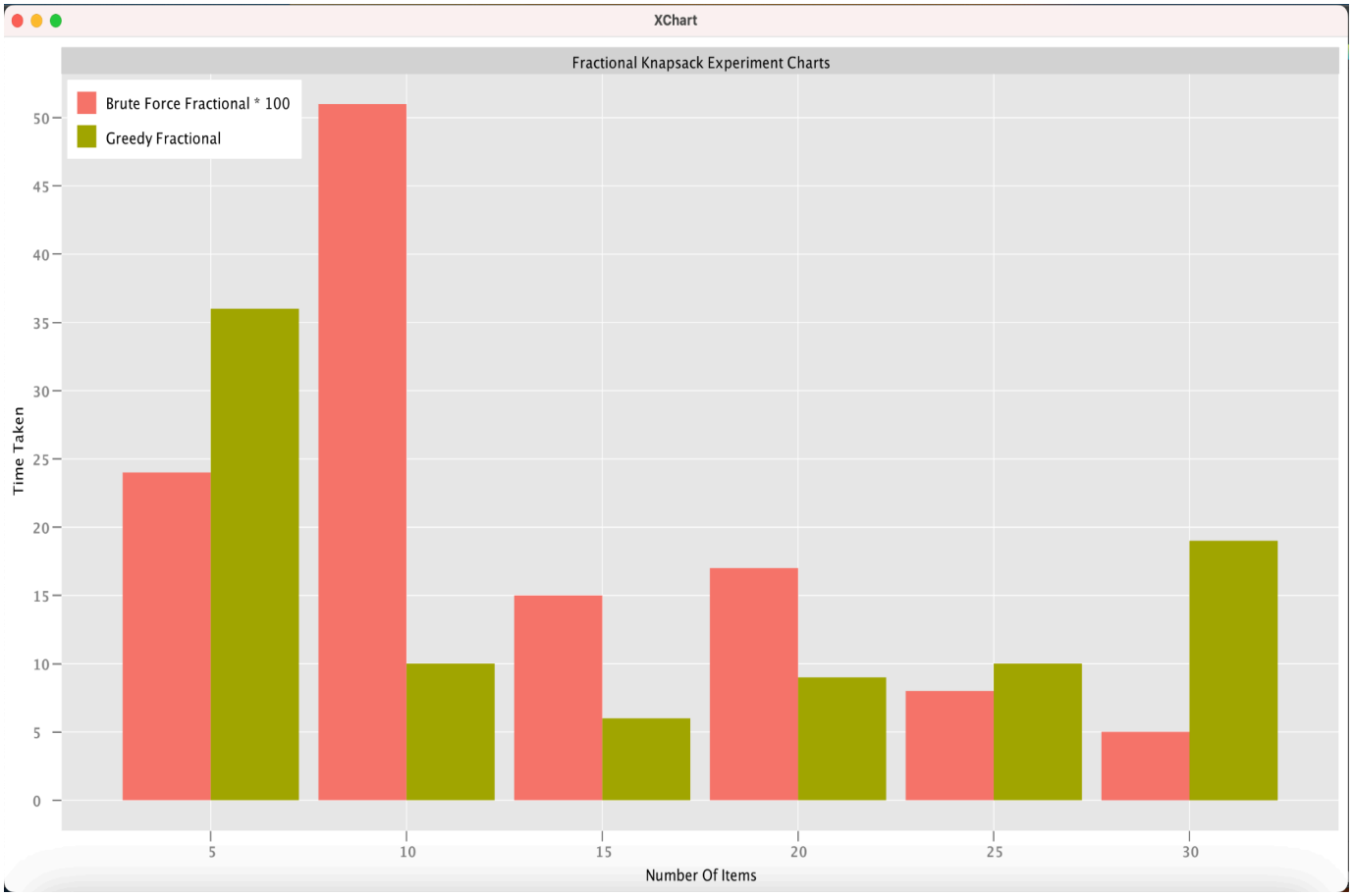
Time complexity for the Greedy Method is O(n x log(n)).
Greedy algorithms make locally optimal choices at each step which aims for a solution that seems like the best solution **in the short term**. However, it means that this approach does not always lead to an optimal solution.
Empirical results were not aligned with theoretical results which was quite interesting to find. I was expecting the runtimes to increase as the sizes of the knapsack increased, however empirical findings show that it fluctuated at certain points. The smallest knapsack 1 was the slowest and knapsack 5 was the fastest. Not entirely sure if hardware might be a cause to this, but my educated analysis on this finding is that the capacity of the knapsacks would be the culprit in this trend. However, utilizing the greedy method is definitely scalable.
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. We would then traverse the rest of the items available to see if anything else might fit and continue with this approach until the capacity is hit or we are out of items. Now we have our Knapsack filled as much as possible with items of the better ratios.

Dynamic Programming

The dynamic 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.

Time complexity of Dynamic Knapsack 01 is O(nW) where n is the number of items and W is the capacity of the knapsack. As the number of items and the capacity of the knapsack increase, the runtime increases, but not exponentially. In the experiment, it generally takes more time compared to the Greedy approach, but less time compared to the Brute Force approach. The Dynamic Knapsack 01 approach strikes a balance between optimality and runtime efficiency, making it suitable for solving the Knapsack problem in practice.

The Greedy approach is fast but may not always give the best solution, while the Brute Force approach is slow and inefficient, especially for larger problem sets.



Fractional Knapsack Problem

Brute Force

The brute force approach of the fractional knapsack problem not only compares all possible combinations of items / subsets of the set of usable items, but also all possible fractions of each item, to find the maximum profit. Because brute force cannot use a greedy approach to decide which items to include, it must compare every possible combination, so it is much slower than the greedy and dynamic algorithms, which can be seen on the graph, other than the last knapsack brute force which took so long we were unable to graph it.

We used a recursive function that explores all possible combinations of fractions of items to find the optimal solution by returning the profit for that combination and comparing the result. Once the current attempt runs out of items to process or remaining capacity in the knapsack, the profit is returned (base case). Otherwise, the function considers excluding the current item and moving to the next index, compared to adding various fractions of the item (which we limited to hundredths n / 100 using a loop in order to limit how long the algorithm took to complete) to the

knapsack and recursively exploring the consequences. The final result is the maximum profit achievable by considering all possible combinations of fractions of items.The time complexity for brute force is O(2^n).

From both the theoretical understanding of brute force and real results in microseconds (brute force took several minutes to execute per knapsack, when it completed at all), 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). From the graph, we can see that greedy is the fastest algorithm that can run because it does not have to go through all possible choices.

In the Fractional Knapsack experiment, the comparison between the brute force and greedy algorithms generally aligns with the theoretical expectations, where the brute force algorithm exhibits a longer runtime compared to the greedy algorithm. This is consistent with the understanding that the brute force approach has an exponential time complexity of O(2^n), while the greedy algorithm has a polynomial time complexity of O(n log n).

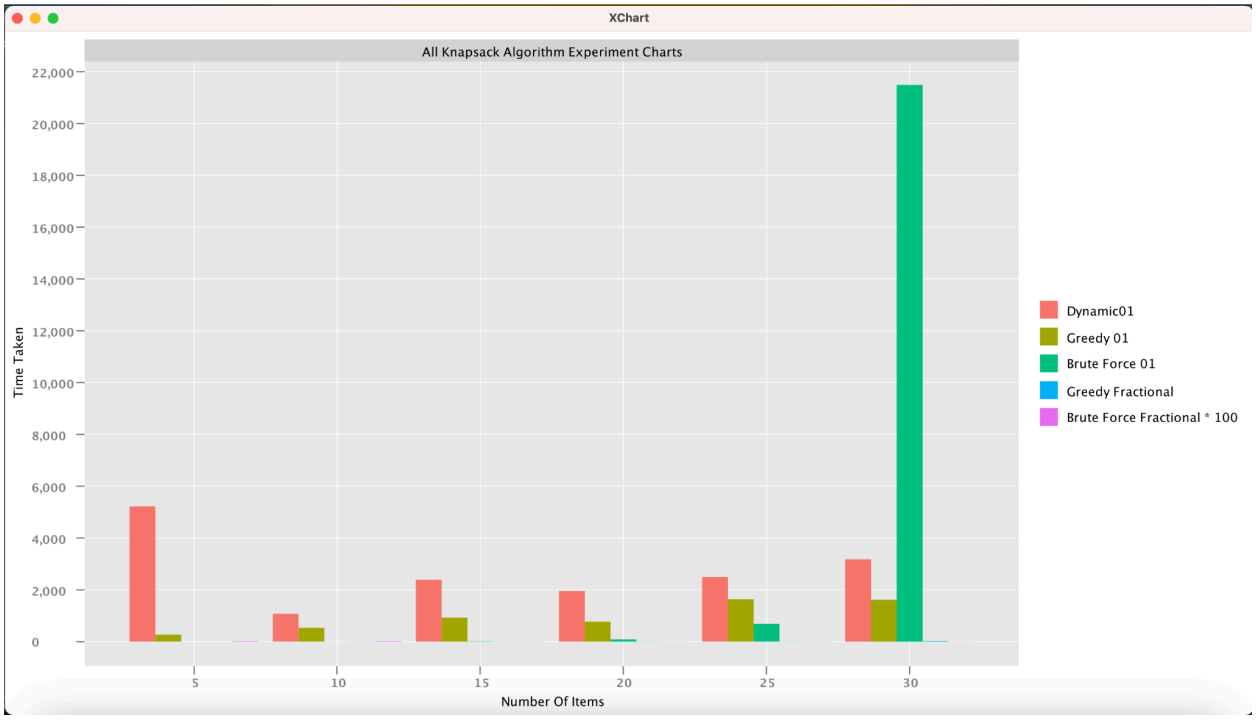
In 01 Knapsack, Brute force, where we check every possible combination of items, takes the longest time. As we add more items, it gets even slower. This is what we'd expect because checking all combinations becomes much harder as there are more items to consider.

The greedy algorithm, which makes the best choice at each step, is faster. Its speed grows more slowly as we add more items. So, even though it might not always give the best solution, it's quick and often works well enough.

Dynamic programming falls in between. It's slower than the greedy approach but faster than brute force. As we add more items, it takes longer to run, but not as much as brute force. It's like finding a balance between speed and accuracy by solving smaller problems first and then combining their solutions to find the best overall solution.

Overall, the choice of algorithm for the knapsack problem depends on the specific requirements of the problem instance. Greedy algorithms offer speed but may sacrifice optimality, while dynamic programming provides a balance between speed and accuracy. Brute force remains the most reliable option for ensuring the optimal solution but may become impractical for larger problem sizes due to its exponential runtime growth.

Graph for all Algorithm



Made adjustments to the units in run time for the Brute Force Algorithm and indicated the change in the bar chart legend.

Analytical and Theoretical Analysis

The Knapsack 01 Experiment Chart indicates that the dynamic 01 knapsack is more efficient than brute force but slower than greedy algorithms. However, we know that the greedy method

does not guarantee an optimal solution.
In the Fractional Knapsack Experiment Chart, the greedy program shows it's faster than brute force, and the results indicate that Fractional Knapsack tends to be faster than Knapsack01.

Dynamic programming for 0/1 knapsack has a time complexity of O(nW), where n is the number of items and W is the capacity of the knapsack. Greedy algorithms generally have a time complexity of O(n log n), and brute force for 0/1 knapsack has an exponential time complexity, typically O(n * 2^n). For fractional knapsack algorithms, greedy usually has a time complexity of O(n log n), and brute force is O(2^n).

The actual runtimes align with the expected trends based on theoretical analysis. Dynamic programming and greedy algorithms are significantly faster than brute force, confirming their expected efficiencies. Brute force, while conceptually simple, becomes impractical due to its exponential time complexity. Greedy algorithms provide fast solutions but may not always guarantee the optimal solution for 0/1 knapsack problem. The dynamic approach to the knapsack algorithm provides optimal solutions for 0/1 but has relatively slower computation time for fractional, compared to greedy.

In summary, the choice of the algorithm depends on various factors, including the size of the problem instance, the type of problem (0/1 vs fractional), the need for optimality, and available computational resources. Dynamic programming is preferable for larger instances requiring optimal solutions, while greedy algorithms offer fast solutions with acceptable optimality guarantees in certain scenarios. Brute force is usually impractical except for very small instances due to its exponential time complexity.