Comparing Knapsacks Experiment

Final Report

By: Team Toast

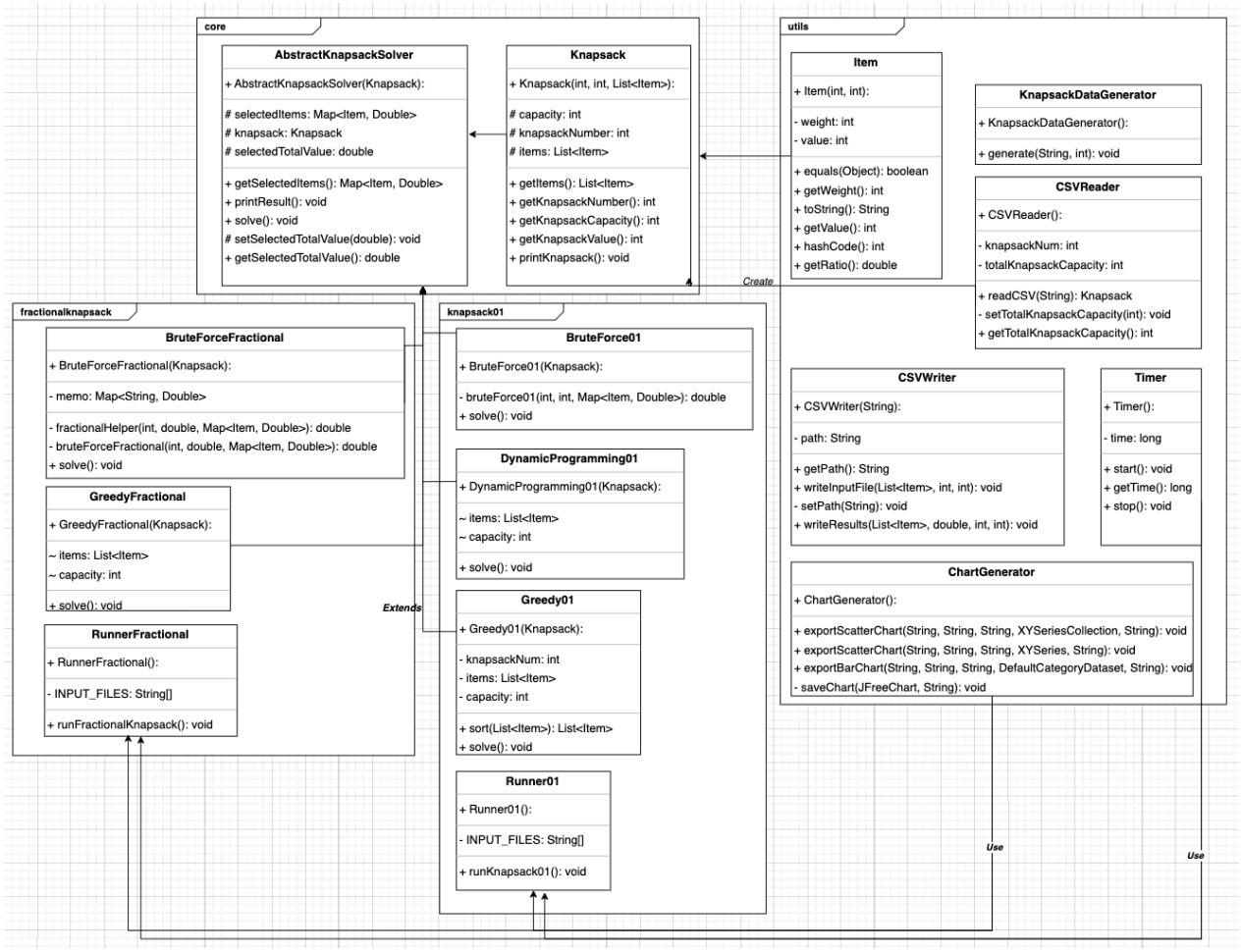
Alice Li, Skye Hobbs, Henry Stiff, Tinisha Davis

1. Design

1.1 Experiment Design Overview

The Comparing Knapsacks Experiment is a Java-based project aimed at implementing and analyzing different approaches to solving the 0/1 Knapsack Problem and the Fractional Knapsack Problem. The project requires the development of Brute Force, Greedy, and Dynamic Programming algorithms, comparing their execution times and efficiency. The program processes input data from CSV files, executes the algorithms, and generates performance metrics such as execution time and profit. Results are visualized using plot charts to evaluate the effectiveness of each algorithm. Additionally, a final report is required to document the design, results, and theoretical analysis, including comparisons between empirical and theoretical performance. The project follows OOP principles, unit testing, and GitHub version control practices, ensuring a structured and well-documented implementation.

1.2 UML Diagram



1.3 Metrics for Evaluation

Execution Time: Measures algorithm runtime in microseconds to compare efficiency across different input sizes.

Knapsack Profit: Records the maximum total value obtained for each algorithm to evaluate solution quality. Fractional Knapsack is limited to increments of 0.1.

Number of Items vs. Execution Time: Analyzes how input size impacts performance for each algorithm.

Number of Items vs. Profit: Compares the profitability of different algorithms for varying knapsack capacities.

Algorithm Complexity: Evaluates theoretical time complexity and compares it with observed performance.

Scalability: Identifies the breaking point where Brute Force becomes infeasible due to exponential growth in runtime.

Visualization: Generates bar and scatter plots using JFreeChart to illustrate execution time and profit comparisons.

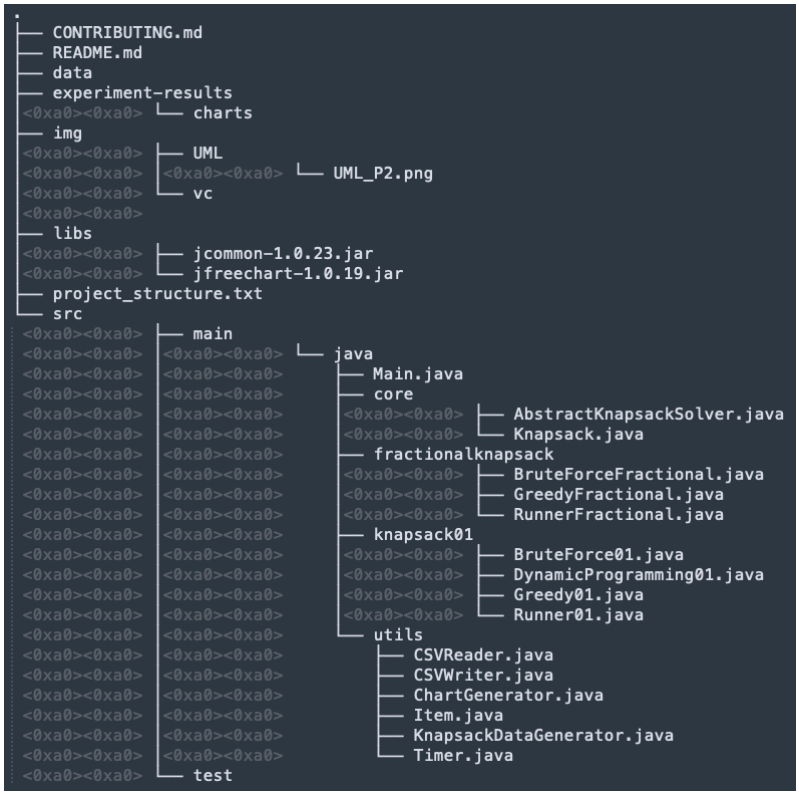
1.4 Results Display Approach

Console Output: Displays knapsack details, algorithm results, and execution times for verification. This includes knapsack capacity, item values, item weights, and computed profits for each algorithm.

Chart Visualization (JFreeChart):

- Scatter Plots: Show execution time vs. number of items to compare algorithm efficiency.
- Bar Charts: Compare knapsack profits across different algorithms.
- Combined Charts: Display all algorithm performances together for better insights.

1.5 Code Organization



Encapsulation and Abstraction (core/):

- Uses an abstract base class (AbstractKnapsackSolver) to enforce a standard interface across knapsack solvers.
- Encapsulates data within Knapsack and Item, restricting direct access and ensuring controlled data handling.

Modular Package Structure:

- Separates 0/1 Knapsack (knapsack01/) and Fractional Knapsack (fractionalknapsack/) implementations for clarity and maintainability.
- Runner classes (RunnerFractional, Runner01) provide dedicated entry points, keeping execution logic independent.

Utility Classes (utils/):

- Encapsulates data processing (CSVReader, CSVWriter), execution timing (Timer), and visualization (ChartGenerator), keeping algorithms clean and reusable.

Performance Tracking and Visualization:

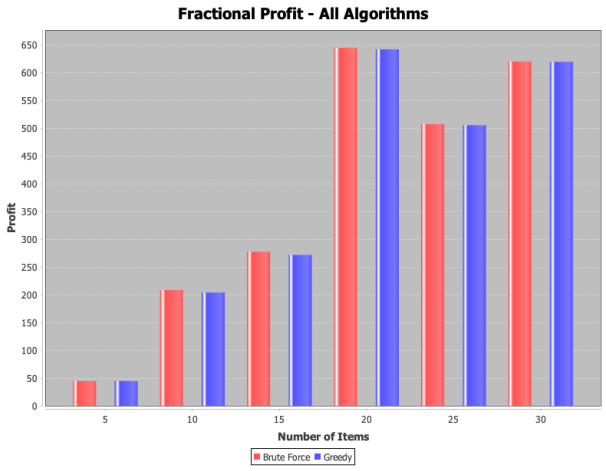
- Uses JFreeChart for execution time and profit comparison, encapsulating visualization logic within ChartGenerator.

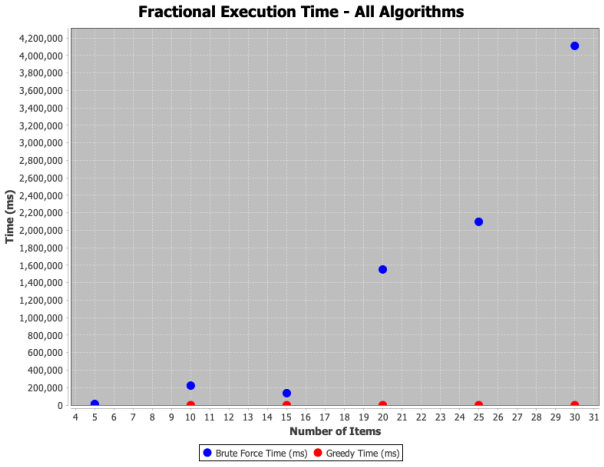
Testing and Validation (src/test/):

- Implements unit tests to achieve 75%+ method coverage, ensuring correctness without exposing internal algorithm details.
- This structure follows OOP principles, promoting encapsulation, modularity, and code reusability, making debugging, testing, and future improvements efficient.

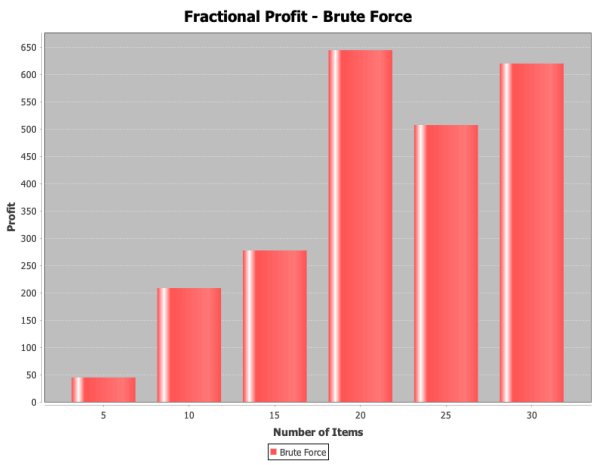
2. Experimental Results and Observations

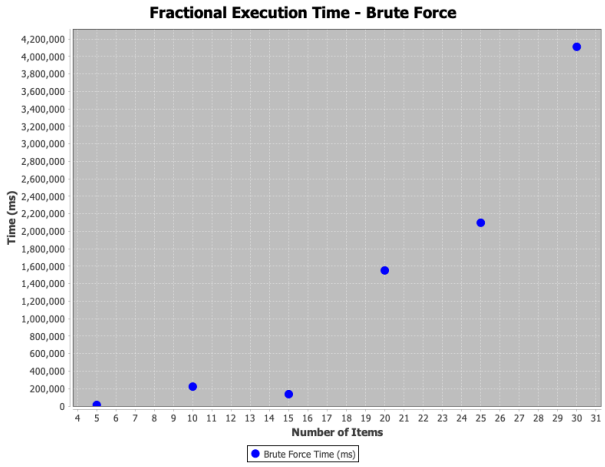
2.1 Fractional Knapsack





Fractional Knapsack Brute Force





The brute-force fractional knapsack algorithm is designed to find the optimal solution through trial and error. It does this by recursively evaluating every possible combination of items, selecting full items until no more can fit, and then taking a fractional portion of the final item to maximize the knapsack's capacity. This exhaustive process leads to an exponential time complexity of O(2^n)

Empirical vs. Theoretical Results

Theoretically, the runtime of the brute-force fractional knapsack algorithm should increase exponentially as the number of items grows. This pattern is generally reflected in the empirical results, where the graph trends upward exponentially. However, there is one exception: the runtime at an input size of 15 is smaller than at an input size of 10. Factors such as system-level optimizations, memory allocation differences, or input characteristics may contribute to these irregularities. Despite these small fluctuations, the overall trend demonstrated the expected exponential growth.

Profit Analysis and Performance

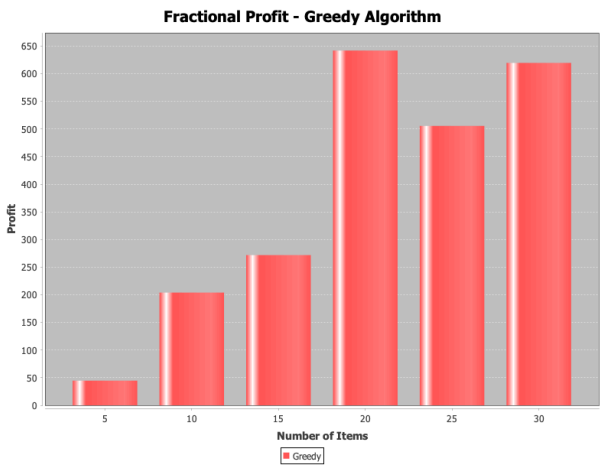
The brute-force fractional approach will always yield the highest-value result. Additionally, it will always provide a value greater than or equal to that of the brute-force 0/1 approach. However, the trade-off for this accuracy is the algorithm's time complexity. It consistently requires more time to execute its methods than any other knapsack algorithm.

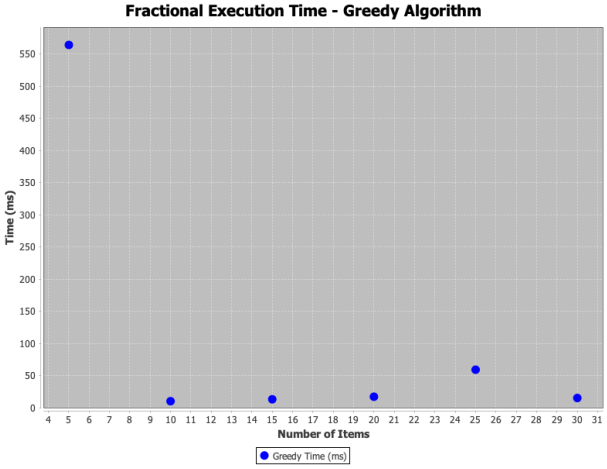
Development Process and Methodology

Implementing the brute-force fractional knapsack algorithm presented unique challenges due to its exhaustive nature. A recursive approach was chosen as the optimal way to evaluate the many possible combinations. This is achieved by recursively comparing the total value when including a given item versus excluding it.

For the fractional implementation, a helper method is used to calculate the fractional value of an item when the remaining capacity cannot accommodate its full weight. This ensures that the algorithm accurately accounts for partial item inclusion, maintaining the correctness of the solution while still exploring all possible selections.

Fractional Knapsack Greedy





The Greedy Fractional Knapsack algorithm is designed to select items based on the highest value-to-weight ratio, ensuring an optimal solution for the Fractional Knapsack Problem. The algorithm follows a two-step process: 1. Sorting items in descending order based on their value-to-weight ratio (O(n log n) complexity). 2. Iterating through the sorted list, adding whole or fractional parts of items until the knapsack reaches capacity (O(n) complexity).

Empirical vs. Theoretical Results

Empirical results revealed an unexpected runtime spike for input size 5, despite theoretical expectations of O(n log n) scaling smoothly. This anomaly likely arises from sorting overhead and JVM optimizations. Java's Collections.sort() incurs fixed setup costs, which disproportionately impact small datasets, where the overhead outweighs the actual sorting work. Additionally, since the JVM compiles code dynamically, the first few runs may be slower due to just-in-time (JIT) compilation. A practical solution is to add a warm-up phase before measuring execution time, allowing JIT optimizations to stabilize and reducing variability in results. For larger input sizes, the algorithm performed as expected, maintaining consistent efficiency compared to Brute Force.

Profit Analysis and Performance

In terms of total profit, the Greedy Fractional approach consistently produced high-value results, often very close to or matching the Brute Force solution. Since the fractional approach guarantees an optimal selection, the total value obtained was always expected to be higher than or equal to the 0/1 Knapsack solutions. Additionally, the 0.1 granularity constraint was handled effectively, ensuring realistic fractional selection while maintaining precision in profit calculations.

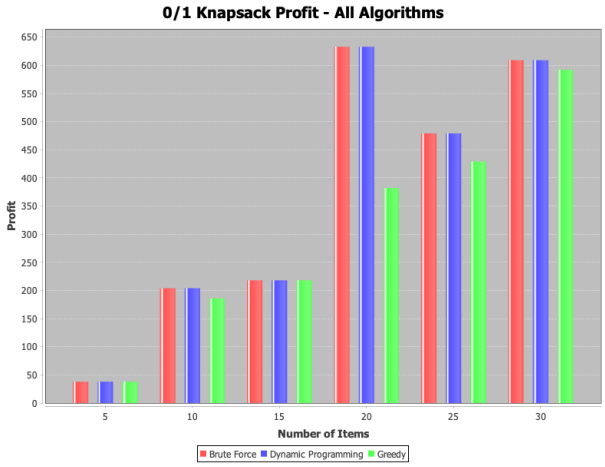
Development Process and Methodology

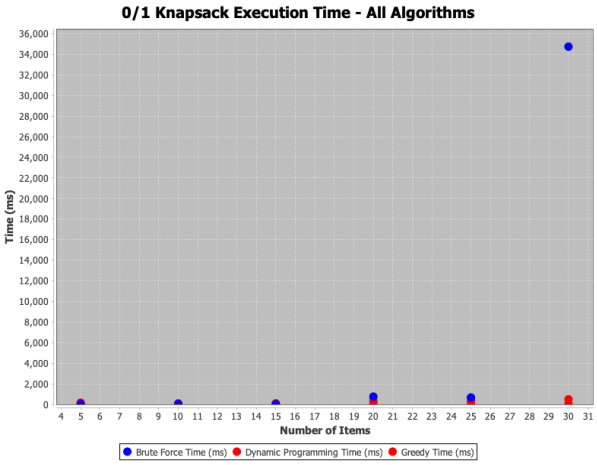
The algorithm was implemented within a modular OOP structure, reusing the AbstractKnapsackSolver class to maintain consistency across different solvers. A comparative sorting approach was used to prioritize items efficiently. Empirical validation was conducted by benchmarking results, analyzing runtimes, and visualizing execution time vs. input size trends.
Despite minor runtime variations in smaller inputs, the Greedy Fractional Knapsack algorithm remained highly efficient, reinforcing its suitability for Fractional Knapsack problems where optimal selection is achievable through local greedy decisions.

Fractional Knapsack Dynamic Programming

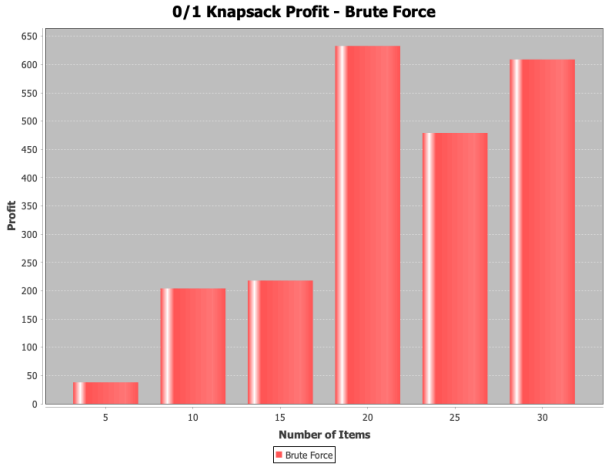
Dynamic programming for solving fractional knapsacks is complicated and difficult to implement. When implementing dynamic programming for a knapsack, a two dimensional array is used in which the cells represent individual subproblems. The rows of the table represent the items that are considered, and the columns represent the capacity considered when choosing what to put in the knapsack. This means that the two dimensional array will be of W * n size, where W is the capacity of the knapsack and n is the number of items in the problem. For 01 knapsacks, the subproblems are solved for every integer value from 0 to W. For example, using dynamic programming to solve an 01 knapsack problem of capacity 5 would mean solving 6 subproblems in each row where each column considers 1 more capacity. If there were 3 potential items in this case, there would be 0 to 3 rows representing the items, and in total there would be 4 * 6 = 24 subproblems needed to find the optimal solution. When solving a fractional knapsack problem, since any amount of an item can be added to the knapsack, fractional values between 0 and W must be considered, meaning that the amount of columns in the array will increase exponentially for every decimal place. This already large amount of columns is multiplied by the number of rows or items considered in the problem. There is also the case of infinite decimals, like pi, making it impossible to use dynamic programming and create an array which can represent the subproblems over infinite decimal weight. A possible workaround to this is only considering fractions up to a certain granularity (e.g. only considering columns in increments of 0.10), however without considering every possible subproblem the algorithm is prone to giving a suboptimal solution. Even if you are willing to compromise on using a certain granularity, the runtime will still be exponentially slower if even large decimal increments are chosen for granularity.

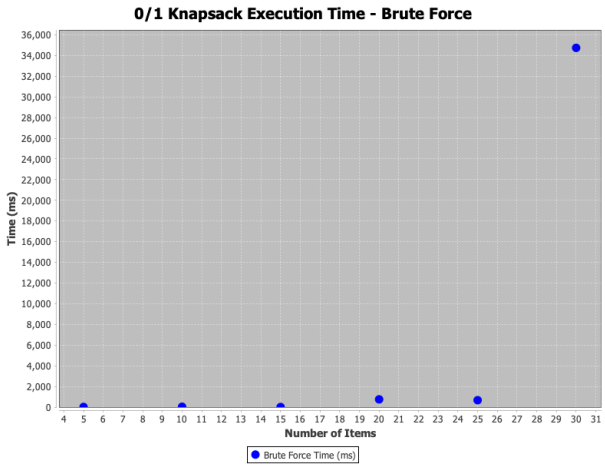
2.2 01 Knapsack





01 Knapsack Brute Force





Similar to the brute-force fractional knapsack algorithm, the brute-force 0/1 approach finds the optimal solution through trial and error. It recursively evaluates every possible combination of items but does not take fractional values to fill the remaining capacity. As a result, the knapsack may not always be fully utilized. However, this exhaustive approach still leads to an exponential time complexity of O(2^n).

Empirical vs. Theoretical Results

Theoretically, the runtime of the brute-force fractional knapsack algorithm should increase exponentially as the number of items grows. However, this pattern is not clearly reflected in the empirical data. Until the input size reaches 30, there is little noticeable exponential growth, suggesting that either the spike at size 30 is an anomaly or that exponential growth would become more apparent beyond this point.

Despite these variations, the empirical results consistently show that the brute-force 0/1 knapsack algorithm has a lower runtime than the brute-force fractional approach, which aligns with expectations. This difference arises because the 0/1 approach does not need to handle fractional values, reducing computational overhead and making it comparatively more efficient.

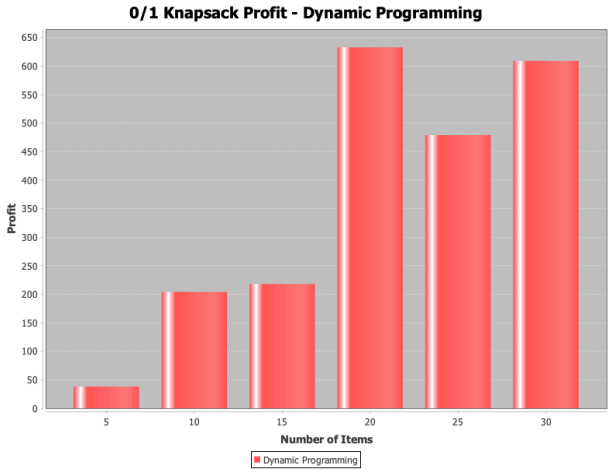
Profit Analysis and Performance

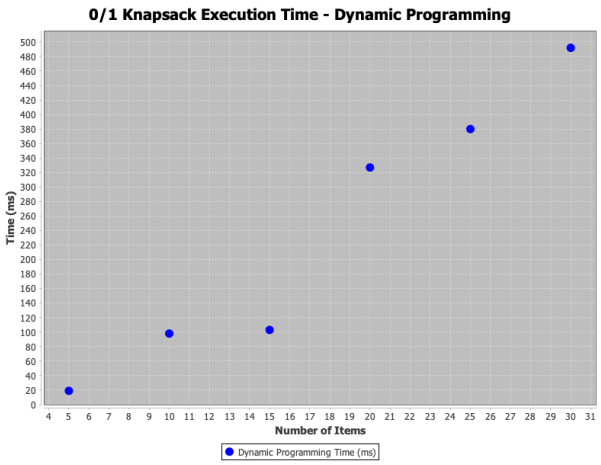
The brute-force 0/1 approach will always yield the highest-value result among all 0/1 knapsack approaches. Additionally, it will always provide a value less than or equal to that of the brute-force fractional approach. As previously mentioned, the exhaustive brute-force method comes with the trade-off of greater time complexity. However, compared to other algorithms, its performance remains relatively similar until the input size reaches 30, at which point more noticeable differences in runtime emerge.

Development Process and Methodology

The development of the brute-force 0/1 knapsack algorithm is similar to its fractional counterpart in its recursive implementation. The primary difference is that there is no helper function to calculate fractional values when the recursive method reaches capacity. Instead, the 0/1 implementation strictly follows a binary choice: an item is either taken in its entirety or excluded from the solution.

01 Knapsack Dynamic Programming

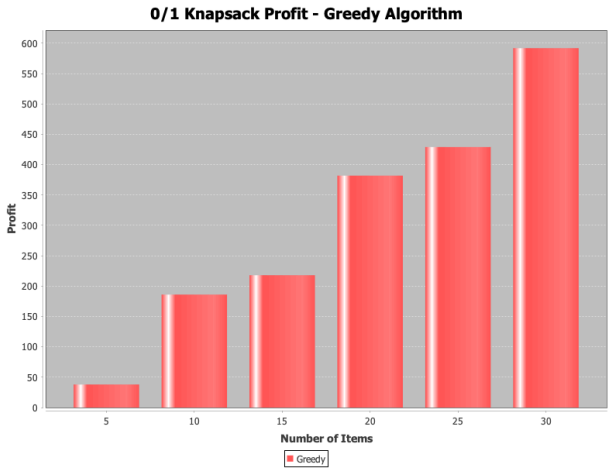


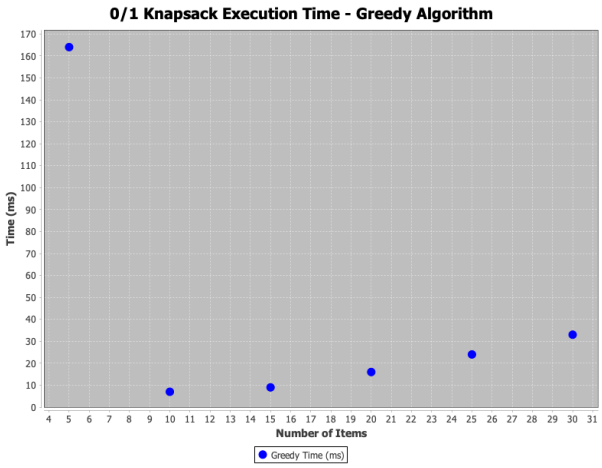


The dynamic programming algorithm for 01 knapsack problems uses the fact that subproblems overlap to optimize time complexity. There are multiple approaches one can take when implementing a dynamic programming algorithm, but for this project, the tabulation approach was chosen because of its efficiency. Another approach that was not chosen is the recursive implementation which should theoretically have about the same runtime as brute force. The tabulation implementation runs in O(n x W) time, where W is the capacity of the given knapsack. There is some variance in the dynamic programming results

which is likely due to the factor W in the time complexity. The knapsack capacity can vary in each case which will affect the overall runtime of the algorithm. The reason that this variable has a significant effect on runtime is that the tabulation implementation creates a two-dimensional array, where the columns range from zero to the knapsack capacity and the rows represent the items in the knapsack. Each column represents a max weight to consider when finding the max value that can be reached with the items that are considered in that row. Since this approach uses a two-dimensional array, increasing the capacity means increasing the amount of columns, which multiplies the amount of subproblems that must be completed to reach the final value. In the results of this experiment, it can be seen that dynamic programming is significantly faster than the brute force approach, but slightly slower than the greedy approach. By comparison, however, the greedy approach does not always find the most optimal value for the problem. Overall, dynamic programming is the most efficient algorithm for 01 knapsacks when you must guarantee the result is the most optimal value.

01 Knapsack Greedy





Empirical vs Theoretical Results

In our experiment with the Greedy 0/1 knapsack algorithm, we observed the empirical results closely to compare them with the theoretical expectations. The Greedy algorithm sorts the items based on their values in descending order and it selects the highest valued item, as long as it fits within the remaining capacity of the knapsack. Theoretically, this approach is efficient with a time complexity of O(n log n) due to sorting, but it does not always guarantee an optimal solution. Empirically, the results showed that the Greedy algorithm performed well for smaller knapsacks. However, when the dataset contains varied item sets it can miss optimal solutions. These findings align with theoretical expectations, confirming that while the Greedy algorithm is fast, it may not always yield the optimal solution.

Profit Analysis and Performance

The profit analysis of the Greedy 0/1 knapsack algorithm revealed that it efficiently maximized the total value of the knapsack but with some limitations. For item sets with homogeneous values, the algorithm performed admirably, achieving near-optimal solutions. However, in scenarios with diverse datasets, the heuristic nature of the Greedy algorithm sometimes misses the optimal combination of items, resulting in lower total values. Despite these occasional shortcomings, the algorithm's speed and simplicity make it a value option for large datasets where computational efficiency is a priority. The performance of the Greedy algorithm demonstrated its practical utility in specific contexts, especially when compared to more computationally intensive methods.

Development Process and Methodology

The development of the Greedy 0/1 knapsack algorithm involved several steps. First, the algorithm was designed to sort items in descending order based on their values. This sorting step ensures that the items with the highest value are considered first. Next, the algorithm iteratively adds the highest-valued item to the knapsack, if it fits within the remaining capacity. This methodology uses a heuristic approach to quickly build a solution, prioritizing efficiency over optimality. The primary goal was to achieve a balance between computation speed and solution quality, making the algorithm suitable for scenarios where quick decision-making is essential.

3. Discussion and Conclusions

3.1 Trade-offs and Applications

Brute Force Fractional Trade-offs

1. Efficiency vs Optimality
    - Efficiency: The brute-force fractional algorithm is extremely slow, with a time complexity of O(2^n). This inefficiency arises from its exhaustive approach, as it evaluates every possible combination of items, including fractional selections.
    - Optimality: Despite its inefficiency, the brute-force fractional approach guarantees the highest possible total value by considering both full and partial item selections, ensuring the optimal solution.
2. Simplicity vs Flexibility
    - Simplicity**:** Implementing brute force is straightforward once its time-inefficient nature is understood, as it follows a trial-and-error approach to evaluate all possible selections. However,

the fractional implementation introduces additional complexity due to the need for a helper method to calculate fractional values when full items cannot fit.

- Flexibility**:** The exhaustive nature of brute force ensures that it will always identify the best possible solution for any given data set, making it the most accurate approach despite its inefficiencies.

3. Speed vs Accuracy
    - Speed: The brute-force fractional algorithm is slower than any other algorithm in the knapsack experiment, consistently performing worse across all data sets. The added complexity of retrieving fractional values further contributes to its long runtime.
    - Accuracy: Despite its slow execution, the brute-force fractional approach always provides the most accurate solution, guaranteeing the highest possible total value compared to any other algorithm.

Greedy Fractional Trade-offs

1. Efficiency vs Optimality

- Efficiency: The Greedy Fractional Knapsack algorithm is highly efficient, with a time complexity of O(n log n) due to the sorting step. Once sorted, the algorithm runs in O(n) time, making it suitable for handling large datasets where computational speed is a priority.
- Optimality: Unlike the 0/1 Knapsack problem, the Fractional Knapsack problem guarantees an optimal solution when using the greedy approach. By selecting items based on their value-to-weight ratio and allowing fractional selections, the algorithm ensures that the total value is maximized within the weight constraint.

2. Simplicity vs Flexibility

- Simplicity: The implementation of the Greedy Fractional Knapsack algorithm is straightforward. Sorting items by their value-to-weight ratio and greedily adding them to the knapsack (including fractions when needed) is an intuitive and efficient process.

- Flexibility: While the greedy approach is optimal for the Fractional Knapsack problem, its reliance on sorting and direct selection means it lacks flexibility for variations of the knapsack problem, such as the 0/1 Knapsack or multi-constrained knapsack problems, where fractional selections are not allowed.

3. Speed vs Accuracy

- Speed: The algorithm is fast and can quickly determine the maximum possible value, making it ideal for scenarios where decisions need to be made in real time.
- Accuracy: Unlike the 0/1 Greedy Knapsack algorithm, the Fractional Knapsack algorithm is always optimal. Since it allows fractional items, it can efficiently utilize the available weight capacity without missing potential value, ensuring an exact solution rather than an approximation.

Greedy Fractional Applications

1. Resource Allocation
    - In scenarios where resources can be divided, such as budget allocation, fluid distribution, or bandwidth allocation, the Greedy Fractional algorithm ensures an optimal way to maximize utility. For example, in cloud computing, allocating resources like CPU time or memory to different tasks based on priority follows a similar approach.
2. Investment Strategy
    - When distributing funds among multiple investments, the Greedy Fractional approach helps in maximizing returns by allocating capital based on the highest return-to-cost ratio. This is useful in portfolio management, where partial investments can be made in different assets to achieve the best financial

Greedy 0/1 Trade-offs

1. Efficiency vs Optimality
    - Efficiency: The Greedy algorithm is highly efficient with a time complexity of O(n log n) due to sorting. This makes it suitable for large datasets where computational speed is a priority.
    - Optimality: While the algorithm is efficient, it doesn't always guarantee an optimal solution. By prioritizing items based solely on their value, the algorithm may miss combinations that offer a

higher total value within the given weight capacity. This is particularly evident when items have varied weights.

2. Simplicity vs Flexibility
    - Simplicity: The implementation of the Greedy algorithm is straightforward and easy to understand. Sorting items by value and iteratively adding them to the knapsack is a simple and intuitive approach.
    - Flexibility: The simplicity of the algorithm comes at the cost of flexibility. The value based sorting heuristic is rigid and may not adapt well to diverse datasets. This lack of flexibility can result in suboptimal solutions for more complex problems.
3. Speed vs Accuracy
    - Speed: The Greedy algorithm is fast and can quickly provide a solution, making it ideal for real-time applications where speed is crucial.
    - Accuracy: The speed of the algorithm is achieved at the expense of accuracy. The heuristic approach may produce suboptimal solutions, especially for datasets with a wide range of values and weights.

Greedy 0/1 Applications

1. Real-time decision making:
    - The Greedy algorithm's speed makes it well suited for applications that require real-time decision making. For example, in an online shopping platform that recommends items to a user based on their price within a given budget.
2. Simplified optimization problems:
    - In cases where the knapsack problem has a relatively small number of items or the items have similar values, the Greedy algorithm can provide near optimal solutions efficiency. For example, in situations involving load bearing where the objective is to distribute tasks or resources effectively.

3. Quick approximation:
    - When a quick approximation is needed, and an exact solution is not critical, this algorithm can be used to provide a reasonable solution in a short amount of time. This can be useful in scenarios like scheduling or packing problems.

Brute Force 0/1 Trade-offs

4. Efficiency vs Optimality
    - Efficiency: The brute-force 0/1 algorithm is extremely slow, with a time complexity of O(2^n). This inefficiency stems from its exhaustive problem-solving approach, as it evaluates every possible item combination.
    - Optimality: Despite its inefficiency, the brute-force 0/1 algorithm guarantees the highest possible total value among all 0/1 approaches.
5. Simplicity vs Flexibility
    - Simplicity: Implementing brute force is simple once you wrap your mind around its time-inefficient nature, requiring a trial-and-error approach to evaluate all possible selections.
    - Flexibility: The exhaustive nature of brute force ensures that it will always identify the best possible solution for any given data set.
6. Speed vs Accuracy
    - Speed: The trial-and-error approach significantly increases execution time as the input size grows, making brute force impractical for large data sets.
    - Accuracy: Given its slow speed it will always provide the most accurate solution within the 0/1 algorithms.

Brute Force 0/1 Applications

1. Guaranteed Optimal Solution:
    - Brute force will always result in the most optimal solution no matter the data set. If time is no concern it is always accurate.

Dynamic 0/1 Tradeoffs

1. Efficiency vs Optimality
    - Efficiency: Because dynamic programming takes advantage of the previously calculated values for the subproblems for the knapsack it is one of the most efficient solutions to the 0/1 knapsack problem.
    - Optimality: The dynamic programming algorithm guarantees an optimal value from the 0/1 knapsack problem. By following the values in the table back, one can also determine what items were selected in order to reach the optimal value.
2. Simplicity vs Flexibility
    - Simplicity: Dynamic programming may seem complicated at first, however the same steps are followed for every problem making it simple once you recognize the pattern.
    - Flexibility: Because the dynamic programming table size depends on the capacity of the knapsack, some problems will cause the algorithm to slow down.
3. Speed vs Accuracy
    - Speed: Reusing previously calculated values allows dynamic programming to solve 0/1 knapsack problems very quickly.
    - Accuracy: The final value in a dynamic programming table is always the optimal value that the knapsack can have.

Dynamic 0/1 Applications

Dynamic programming should be used when it is necessary to obtain the optimal value for an 0/1 knapsack problem since it is the most efficient of the solutions that guarantee that the optimal value will be found. Dynamic programming is simple to implement in code, however, if you only need to solve a 0/1 knapsack problem one time it may be faster to use a simpler algorithm to get it done.

3.2 Conclusion

The Comparing Knapsacks Experiment provided valuable insights into the performance and applicability of Brute Force, Greedy, and Dynamic Programming algorithms for solving the 0/1 and Fractional Knapsack Problems.

Each algorithm demonstrated unique strengths and trade-offs. The Brute Force algorithm consistently produced optimal solutions for both knapsack problems due to its exhaustive search of all possible combinations. However, its exponential time complexity made it inefficient for large datasets. The Greedy algorithm excelled in the Fractional Knapsack Problem, where its heuristic of prioritizing items based on their value-to-weight ratio aligned perfectly with the problem's requirements, yielding optimal solutions efficiently. Conversely, the Greedy algorithm struggled with the 0/1 Knapsack Problem, occasionally missing the optimal solution due to its reliance on a heuristic of prioritizing items based solely on their value. Dynamic Programming emerged as the most reliable and efficient solution for the 0/1 Knapsack Problem, consistently delivering optimal results with a manageable polynomial time complexity, particularly for larger datasets.

Furthermore, the project highlighted the potential for exploring other algorithmic approaches, such as divide-and-conquer methods. These approaches could break the problem into smaller subproblems, solve them independently, and combine results. While divide-and-conquer methods might offer improvements in certain scenarios, they may not always guarantee optimal solutions or outperform the Dynamic Programming approach for the 0/1 Knapsack Problem.

Overall, the experiment emphasized the importance of selecting the appropriate algorithm based on the problem type, dataset size, and required solution quality. By systematically analyzing empirical and theoretical results, the study reinforced the fundamental principles of algorithm design and evaluation in addressing complex computational challenges.