Final Project

Created time	@November 26, 2024 10:36 AM
Status	In progress
Date	@December 1, 2024
Person	Jakob
Email	jbalkovec@seattleu.edu

CPSC 4100 Design and Analysis of Algorithms

Final Project Writeup

Introduction

The 0-1 Knapsack Problem is a classical combinatorial problem with applications in fields like resource allocation, financial modeling, and logistics. This project explores four algorithmic approaches to solving the problem, analyzing three. It includes two brute-force methods, which guarantee an optimal solution at the cost of exponential runtime, and two Algorithm-based methods that prioritize efficiency and scalability over precision.

Implemented in C# using Visual Studio, the project examines the trade-offs between algorithmic complexity and practical performance. A variety of data sets were generated to test the scalability, run-times, and effectiveness of each method. The results provide insights into how input size and knapsack capacity impact performance, offering a comprehensive analysis of each approach's strengths and limitations.

The recursive approach utilizing threading was not thoroughly analyzed, as it provided only marginal performance improvements. The overhead associated with thread management outweighed the potential runtime benefits, rendering its impact negligible.

Project Overview

1.1 Objectives

The objectives of this project are outlined as follows:

Algorithm Design and Implementation:

- To develop a recursive brute-force solution to the 0-1 Knapsack Problem to guarantee optimal results despite high computational cost.
- To implement a DP based approach to balance performance and scalability while exploring trade-offs in accuracy.
- To implement a Memoization approach that balances simplicity of implementation and scalability, while again, exploring trade-offs in accuracy.

Performance Analysis and Evaluation:

- Compare the runtime and scalability of all algorithms across data sets of varying sizes.
- Assess the theoretical and practical complexities of the implementations.

Data Preparation:

- Generate randomized, diverse data sets tailored to the knapsack problem, while ensuring these data sets provide a robust testing ground for analyzing algorithmic performance.

Scalability Assessment:

- Measure how well each algorithm scales with increasing data size and varying knapsack capacities.

Documentation and Reporting:

- Present findings in a clear, well-organized report that highlights insights into algorithmic performance, complexity trade-offs, and real-world implications.

1.2 Scope

The scope of this project includes the design, implementation, and analysis of three approaches to solving the 0-1 Knapsack Problem. First, a recursive brute-force method is implemented to guarantee optimal solutions despite its exponential computational complexity. Secondly, a Algorithm-based (DP) solution is developed, balancing efficiency and scalability while sacrificing guaranteed optimality. Lastly, another Algorithm-based (Memoization) solution that balances simplicity of implementation and scalability, but sacrifices optimality.

The project involves generating diverse, randomized data sets of varying sizes using a custom Python script, structured to test both algorithms effectively. Performance is analyzed by running both implementations on identical data sets, measuring outcomes, execution times, and scalability. Furthermore, the theoretical and observed computational complexities are compared, along with trade-offs in code simplicity and performance. Finally, findings are documented in a comprehensive report, including detailed explanations, clear tables, and execution logs to highlight the algorithms' practical implications.

1.3 Data

The data was obtained using a python script `data_factory.py`. More on this in the *"Data Set Construction"* section.

1.4 Hardware

Before delving into the details of the project, I think it is important to provide an overview of the specifications of the machine used for testing

```
# Obtained using: system_profiler SPHardwareDataType
Hardware:

    Hardware Overview:

      Model Name: MacBook Pro
      Chip: Apple M2
      Total Number of Cores: 8 (4 performance and 4 efficiency)
      Memory: 8 GB
            .
            .
            .

# Obtained using: uname -m
arm64
```

The project was executed on a MacBook Pro using an Apple M2 chip with 8 cores (4 performance and 4 efficiency), 8 GB of memory, and running an *ARM64* architecture.

---

2. Algorithm Implementations

2.1 Optimal Algorithm

Description

The 0-1 Knapsack Problem can be solved optimally using a recursive approach, where the problem is broken down into subproblems. The algorithm considers two possibilities for each item: either include it in the knapsack (if it fits) or exclude it. The maximum value of the knapsack is the highest value obtainable by either including or excluding each item recursively. This recursive approach guarantees an optimal solution but has exponential time

complexity of $O(2^n)$where n is the number of items. The theoretical complexity arises because, for each item, the algorithm explores two possibilities (include or exclude), resulting in a tree-like structure with 2^n possible combinations of items.

C# Implementation

The recursive implementation of the knapsack problem is divided into two methods. The main function `KnapsackRecursive` initializes the recursive process, while the helper function `KnapsackRecursiveHelper` performs the recursive calculations.

1. **Base Case:** If there are not items or the knapsack has no remaining capacity, the function returns 0, representing a value of 0 for the knapsack.
2. **Item Exclusion:** If the current item exceeds the remaining capacity of the knapsack, it is excluded, and the function recurses with one less item.
3. **Item Inclusion and Exclusion:** If the current item fits, the function computes two values:
    - **Include Item:** The value of the item is added to the result of the subproblem with the reduced capacity.
    - **Exclude Item:** The subproblem is solved without the item.
4. **Result:** The function returns the maximum value between including or excluding the item, thereby selecting the best option

---

Raw Code

The code is available via the GitHub link provided in the references section, with the following path: `CPSC4100/Knapsack.cs`,

Lines

- **259-261**
- **271-286**

Key Takeaways

The recursive algorithm offers an optimal solution but at the cost of exponential time complexity, making it impractical for larger datasets. The next sections explore alternative approaches, such as dynamic programming, that aim to address these performance limitations while balancing accuracy and scalability.

2.2 Dynamic Programming Algorithm

Description

The Dynamic Programming (DP) approach to the 0-1 Knapsack Problem aims to solve the problem efficiently by breaking it into smaller subproblems and storing the results of these subproblems to avoid redundant calculations/computations. In contrast to the recursive approach, DP builds a solution incrementally by considering one item at a time and its possible inclusion or exclusion based on the remaining capacity of the knapsack. he time complexity for the approach is $O(n * W)$, where n is the number of items and W is the capacity of the knapsack. This is significantly more efficient than the exponential time complexity of the recursive approach, especially for large n

The DP solution builds a table, `dp[i][w]`, where each entry represents the maximum value that can be obtained with the first `i` items and a knapsack capacity of `w`. For each item, the algorithm decides whether to include it (if it fits) or exclude it, and the table is updated accordingly. The final solution is found in the last cell of the table, `dp[n][capacity]`.

C# Implementation

The DP solution is implemented by filling a 2D array `dp`, where each entry stores the maximum value for a given subset of items and knapsack capacity.

1. **Initialization:** In theory, the first row and column of the table should be initialized to 0 to represent the base case where no items are considered or the knapsack capacity is zero. However, in **C#**, this step is unnecessary, as arrays are automatically zero-initialized by default.

2. **Iterative Filling of the Table**: For each item and each capacity, the algorithm checks whether the item can be included in the knapsack. If it can, the algorithm compares two values:
    - The value of including the item (its value plus the maximum value for the remaining capacity).
    - The value of excluding the item (the maximum value without this item).
3. **Updating the Table:** The table is updated with the maximum value between including or excluding each item.
4. **Final Result**: The final maximum value is stored in `dp[n, capacity]`, which gives the solution for all items and the full capacity.

Raw Code

The code is available via the GitHub link provided in the references section, with the following path: `CPSC4100/Knapsack.cs`

Lines

- **345-369.**

Key Takeaways

The Dynamic Programming approach significantly improves the time complexity over the recursive method, making it a practical solution for larger datasets. It efficiently handles both small and large knapsack problems by storing intermediate results and avoiding redundant calculations. However, it still requires **O(n * W)** space and time complexity, which can be a limitation for very large problem instances. In the next sections, we will explore additional methods such as memoization to further optimize performance.

2.3 Memoization Algorithm

Description

Memoization is a technique that enhances the recursive approach to the 0-1 Knapsack Problem by storing the results of previously computed subproblems, effectively eliminating redundant calculations. It is a form of dynamic programming that uses a cache to remember the results of recursive calls and reuses these results when needed again. This approach maintains the simplicity of recursion while improving performance by ensuring each subproblem is solved only once. The time complexity of memoization is the same as dynamic programming, $O(n * W)$, but it can be more memory-efficient, as it avoids building a full 2D table as in DP. Instead, memoization stores only the specific results required for the recursive calls.

The key idea is to check if a result for a specific subproblem (defined by the remaining capacity and the index of the current item) is already computed. If it is, the algorithm directly retrieves the result from the cache (a dictionary). If not, it computes the result recursively, stores it in the cache, and returns it.

C# Implementation

The memoized solution starts by initializing the cache (a static dictionary `memo`) and then calls a helper method to solve the problem recursively.

1. **Base Case**: If the current index exceeds the number of items or if the remaining capacity is zero, the function returns 0, indicating no items can be selected.
2. **Cache Check**: Before computing the result for a specific subproblem, the function checks if the result is already stored in the dictionary `memo`. If the result exists, it returns the cached value.
3. **Recursive Calls**: If the result is not cached, the function recursively computes the maximum value achievable by excluding the current item and including the current item (if the item fits in the knapsack).
4. **Cache the Result**: After computing the result, the value is stored in the `memo` dictionary for future use.
5. **Clearing the Cache**: A precaution used to avoid having false data in the dictionary.

The recursive helper function compares the two possible outcomes, excluding or including the current item, and returns the maximum value. The result is then cached for the given state defined by the remaining capacity and the current index.

Raw Code

The code is available via the GitHub link provided in the references section, with the following path: `CPSC4100/Knapsack.cs`

Lines

- **374**
- **382-386**
- **402-434**

Key Takeaways

Memoization optimizes the recursive solution by caching the results of subproblems, which helps to avoid redundant recalculations. It strikes a balance between the simplicity of recursion and the efficiency of dynamic programming, providing a more scalable solution compared to the basic recursive approach. The time complexity remains **O(n * W)**, and the space complexity is also proportional to the number of unique subproblem states stored in the cache.

---

3. Data Set

3.1 Construction

As mentioned above, the data was obtained using a python script `data_factory.py`. This script produces randomized data sets tailored to the problem's requirements. This includes generating items with an associated weight and a value **(1)**. On top of that, it also generates the knapsack capacities **(2)**.

Items are generated as *JSON* objects with random weights and values within predefined bounds, while capacities are selected from a separate range, ensuring variety and scalability. Data sets are categorized by size, ranging from *extra small* to *extra large*, with the number of items determined by a `SIZE_MAP` dictionary. The generated data sets are saved as *JSON* files in the directory.

Items Legend

Name	Number of Items
XS	50
S	100
M	250
L	500
XL	1000

Table 1

```
// [1] Example of an "item" object
[
    {
        "Weight": 7,
        "Value": 25
    },
]

// [2] Example of a "capacity" object
[
    {
        "Capacity": 99
    },
]
```

The item weights and values range from **6 to 37**, while the capacities vary between **22 and 100**. These numbers were obtained by observing various data sets on Kaggle. For each iteration of the program (`Main()`), datasets

were randomly generated, ensuring that the same dataset was used for all algorithm iterations (For clarification, look at the **Raw Code** section below). This approach guarantees both complete randomness and fair comparison across all algorithms.

3.2 Scalability and Statistical Properties

The datasets are designed to reflect realistic variations in problem size and complexity. Small datasets focus on computational efficiency, while larger datasets test algorithm scalability. The average weight-to-value ratio is randomized to ensure diverse distributions, challenging the algorithms' optimization capabilities

3.3 Capacity Distribution By Algorithm Type

Using `R`, I created a plot illustrating how the capacity was evenly distributed across all algorithms.

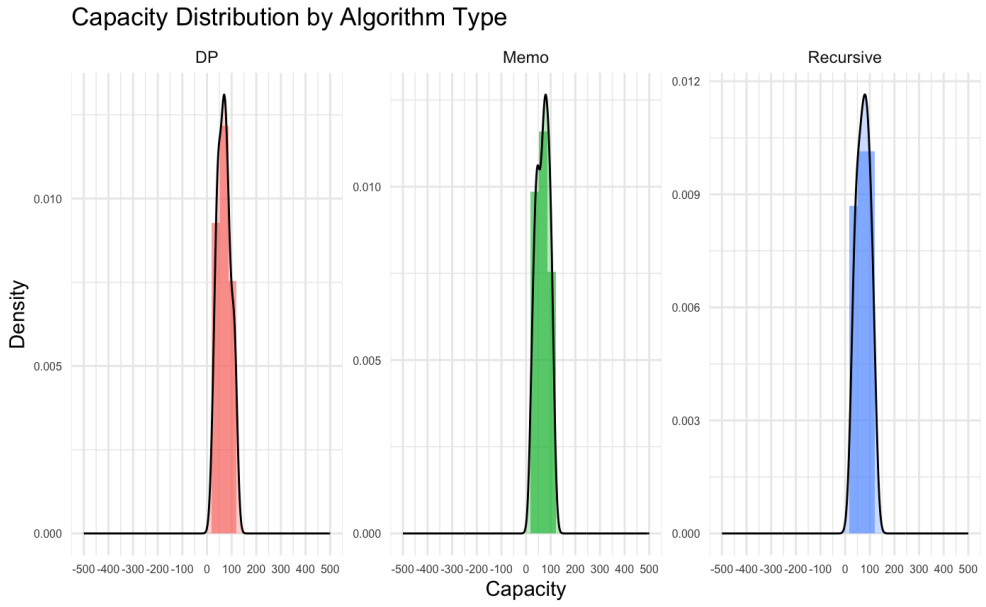

Figure 1

Raw Code

The code is available via the GitHub link provided in the references section, with the following path: `CPSC4100/data`

Data Factory

- Python script that generates data
- starting from **line 23**.

Data Sets (Items)

- Can be found at: `CPSC4100/data/items/`
- **5 files**

Data Set (Capacities)

- Can be found at: `CPSC4100/data/capacity/`
- **1 file**

Program

- Can be found at: `CPSC4100/Knapsack.cs`
- Includes the `Main()` function

---

4. Experimental Results

4.1 Setup

Measurement Process

The execution times for the knapsack algorithms (Recursive, DP, Memo) were measured as follows:

1. **Stopwatch Initialization**:
   - A `Stopwatch` object was used to measure the execution time with millisecond precision.
2. **Timing the Algorithms**:
   - The stopwatch was started immediately before invoking the selected algorithm and stopped upon completion.
   - Execution time was then calculated as the total elapsed time in milliseconds (`stopwatch.Elapsed.TotalMilliseconds`).
3. **Data Logging**:
   - The execution times, along with other relevant test details (method name, item size, capacity, solution value), were logged using a simple custom `Logger` class **(3)**.

```
// [3] Example of a log entry

2024-11-22 17:15:49 - [INFO]:  *** [TEST #1] ***

2024-11-22 17:15:49 - [INFO]: Method: Recursive
2024-11-22 17:15:49 - [TIME]: 0.7393
2024-11-22 17:15:49 - [INFO]: FOR: [ITEMS]: XS, [CAPACITY]: 50
2024-11-22 17:15:49 - [SOLUTION]: 146
----------------------------------------
```

System and Environment Details

The tests were conducted in the following `.NET` environment:

- **C#** `.NET 7.0.308`

Test Design and Constraints

Each algorithm was tested on the same dataset (randomly generated per test iteration) to ensure fairness. The Recursive and Recursive + Threading methods were intentionally skipped for larger datasets (M, L, XL) due to excessive execution time. These **skips were logged** with an execution time of `1`.

Reproducibility

All measurements, including skipped tests, were logged to the file `execution_log.log` for traceability.

Raw Code

The code is available via the GitHub link provided in the references section, with the following path: `CPSC4100/execution_log.log`

- Not to be confused with the file `FP_log.log` which was used for debugging, testing and predicting the algorithm runtime.

Potential Sources of Variability

I made every effort to ensure the environment was as *fair* as possible for all algorithms. However, it's important to acknowledge that certain factors beyond my control could influence performance, such as background processes and thermal throttling. These external factors might introduce slight variations in the measurements. To address this, each test was repeated five times, and the average execution time was analyzed to ensure consistency.

4.2 Runtime Comparison

The runtime data was recorded and organized into an Excel table for clarity. I then utilized `R` to create graphs and visual aids to better illustrate the results. As mentioned earlier, to minimize variability, the mean execution time was used for most comparisons.

Table

Mean Execution Time (ms)	Recursive	Memoization	Dynamic Programming
XS (50 items)	`17.3922`	`0.7113`	`0.1241`
S (100 items)	`44785.4585`	`1.0384`	`0.2890`
M (250 items)	`-1`	`2.0703`	`0.0660`
L (500 items)	`-1`	`3.8688`	`0.1939`
XL (1000 items)	`-1`	`10.2506`	`0.2532`

Table 2

The table shows the mean runtime for each of the algorithms (5 runs). As mentioned, the larger data sets were omitted for the recursive algorithm, due to exhaustion of my stack space.

Graph

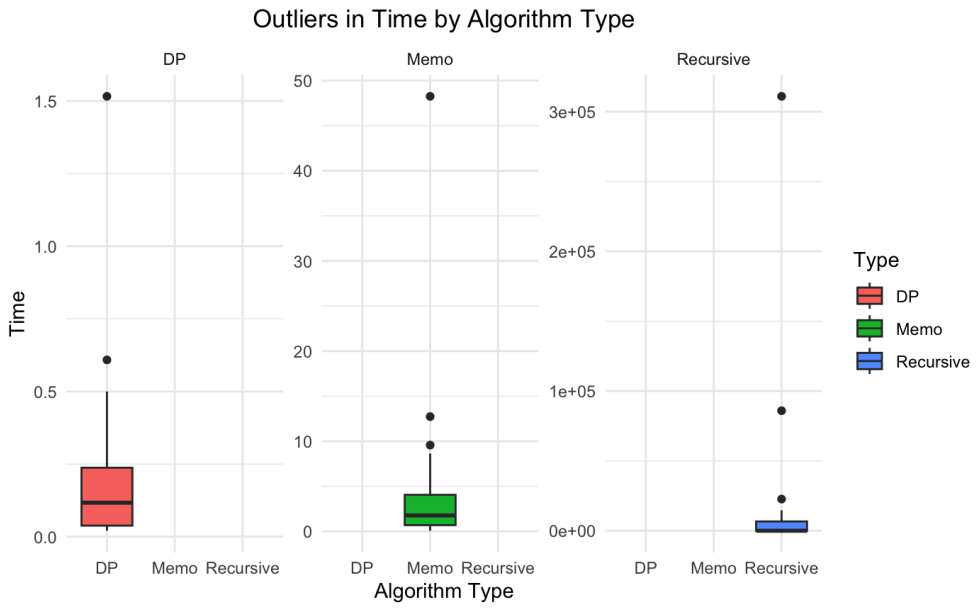

Figure 2 ⇒ Execution time for each method

The graph consists of three plots, one for each algorithm. It highlights the outliers, which may be influenced by external factors. Each plot has a distinct unit on the y-axis, reflecting the different measures for each algorithm.

Analysis of the Results

The recursive algorithm runs significantly slower than the Algorithm approaches, primarily due to its exponential time complexity of $O(2^n)$. This inefficiency led to the algorithm running so slowly that I had to abandon the execution for the `M` dataset, as it took over 26 hours to complete.

In contrast, the Dynamic Programming (DP) and Memoization approaches exhibit similar performance, as they share a comparable structure (using tables for storage) and both have a time complexity of $O(n * W)$. However, Memoization tends to be slower due to the added overhead of recursion, which involves additional stack space management and repeated function calls, making it less efficient compared to the iterative nature of DP. Despite this, both methods are still significantly faster than the recursive approach, demonstrating the benefit of optimizing recursive calls through memoization or transitioning to an iterative solution like DP.

Research Question

Modern compilers (excluding .NET) often optimize tail recursion by converting it into a loop through Tail Call Optimization (TCO). It could be worthwhile to explore whether using a language like C++, which supports this

optimization, offers any performance benefits.

Raw Data

The file is available via the GitHub link provided in the references section, with the following path: `CPSC4100/Results.xslx`

- **File Contents:**
  - Raw Data (`Sheet: Raw`)
  - Visually Enhanced Data (`Sheet: Visual`)
  - Unit Test Results (`Sheet: Unit Tests`)
  - Performance estimates (`Sheet: Estimates`)

Raw Code

The code is available via the GitHub link provided in the references section, with the following path: `CPSC4100/analysis/FP_Analysis.Rmd`

4.2 Solution Quality

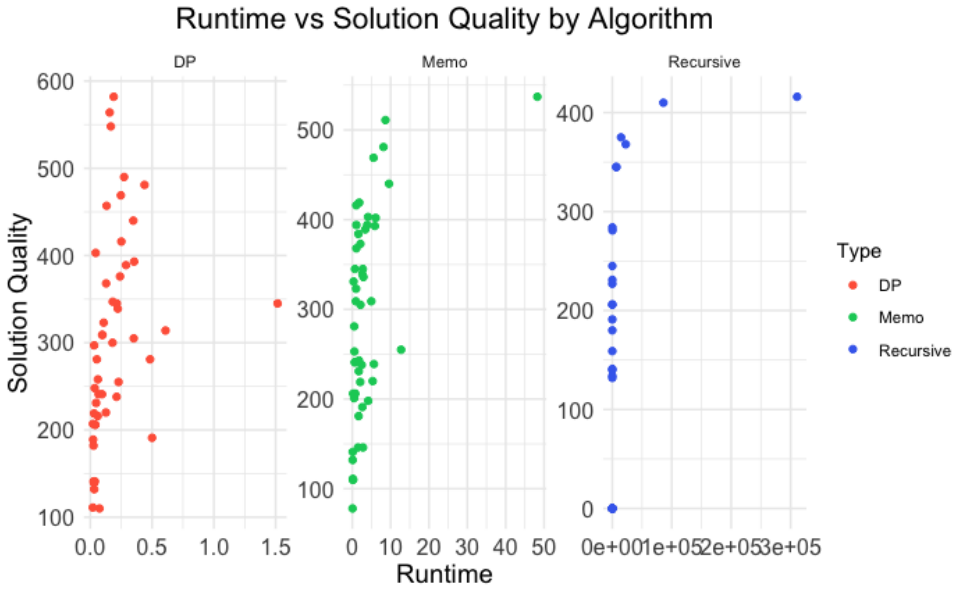

Figure 3 ⇒ Runtime vs Solution Quality

4.3 Expected Performance vs Observed Performance

Recursive Algorithm

The Recursive algorithm was expected to exhibit exponential growth in runtime with increasing input size, as it has a theoretical time complexity of $O(2^n)$. This was confirmed in practice, where the algorithm performed adequately on smaller datasets but became prohibitively slow for larger datasets. The performance observed was consistent with the theoretical prediction of exponential time complexity.

Dynamic Programming and Memoization

The Dynamic Programming and Memoization approaches both have a time complexity of $O(n * W)$, and were expected to handle larger datasets much more efficiently than the Recursive approach. In practice, they both showed a much more linear and predictable growth in runtime as the problem size increased. However, Memoization, while theoretically equivalent to DP in terms of time complexity, performed slightly worse in practice. This is likely due to the recursive overhead, as each function call incurs additional stack management costs and repeated function calls, which is not accounted for in the theoretical time complexity.

4.4 Scalability Expectations

Recursive Algorithm

The Recursive algorithm, as expected, did not scale well with larger problem sizes. It quickly became impractical for medium to large datasets, which aligns with its theoretical inefficiency. The exponential growth of runtime made the algorithm unsuitable for any problem size beyond a 100 items (estimate).

Dynamic Programming and Memoization

Dynamic Programming and Memoization, on the other hand, scaled much better, as anticipated by their $O(n * W)$ time complexity. These two algorithms were able to handle increasing problem sizes with far more efficiency, and while Memoization underperformed slightly due to recursion overhead, it was still much faster than the Recursive approach.

4.5 Edge Cases

For edge cases with small problem sizes, all three algorithms performed as expected. The Recursive algorithm handled small inputs quickly, and both DP and Memoization Algorithms showed their efficiency in terms of time.

The Memoization approach, although theoretically comparable to DP, faced practical challenges that were not fully captured by my theoretical analysis. The recursive calls added overhead that slowed down the algorithm, and the stack space used in recursion also added a layer of inefficiency that affected performance, especially in cases with a high depth of recursion.

4.6 Runtime Trends

The runtime growth trends largely matched the theoretical expectations, but with some deviations.
The Recursive algorithm's runtime increase was more dramatic than expected due to the high overhead of recalculating subproblems. For the DP and Memoization Algorithms, while the time complexity held up well, Memoization's additional recursion overhead was not anticipated in my theoretical analysis, leading to a very slight but still noticeable performance decrease compared to DP.

---

5. Algorithmic Complexity Analysis

5.1.1 Recursive Algorithm Time Complexity Analysis

Recursive Algorithm ⇒ $O(2^n)$

The time complexity of the recursive knapsack algorithm is $O(2^n)$ because the algorithm explores every possible subset of the items to determine the maximum value that can fit within the given capacity. For each item, the algorithm has two options: include the item in the knapsack or exclude it. These choices create a binary decision tree with n levels, where n is the total number of items. At each level, the number of recursive calls doubles, leading to 2^n total calls. This exponential growth occurs because the algorithm examines all possible combinations of items, making $O(2^n)$ the overall time complexity.

If the first item's weight equals the capacity, and its value is higher than any other combination, the recursion may terminate early when exploring the subset that includes the first item. However, due to the lack of optimization, the algorithm will still explore all subsets, leading to

$O(2^n)$ in the best, average, and worst cases.

5.1.2 Dynamic Programming Algorithm Time Complexity Analysis

Dynamic Programming Algorithm ⇒ $O(n * W)$

The time complexity of the dynamic programming knapsack algorithm is $O(n * W)$ because it builds a table to compute the optimal solution by iterating through all items and capacities. The table has n rows, corresponding to the items, and W columns, representing capacities from 0 to the maximum capacity. For each cell in the table, the algorithm computes the maximum value achievable by either including or excluding the current item, which is a constant-time operation. Since the algorithm fills the entire table with $n * W$ entries and performs constant-time computations for each, the overall time complexity is $O(n * W)$ also known as pseudo-polynomial.

Even when the first item matches the capacity, the DP table is still filled completely, as the algorithm iterates over all n items and capacities from 0 to W.

5.1.3 Memoization Heursitc Time Complexity Analysis

Memoization Algorithm ⇒ $O(n * W)$

The time complexity of the memoized knapsack algorithm is $O(n * W)$ because it solves each unique subproblem exactly once. The subproblems are defined by two parameters: the current item index n, which can range from 0 to $n - 1$, and the remaining capacity W, which can range from 0 to W. This creates at most $n * W$ unique states. The memoization ensures that any specific state, represented by `(remainingCapacity,currentIndex)`, is computed only once and stored for future reference. Each recursive call involves constant-time operations, such as checking the memo dictionary and computing the maximum of two values. As a result, the total computation is proportional to the number of unique states, leading to an overall time complexity of $O(n * W)$ also known as pseudo-polynomial.

Even if the first item's weight matches the capacity, the algorithm still checks other possible states to ensure no better solution exists. The memoization avoids redundant calculations, so the time complexity is bounded by the number of unique states.

5.2 Time Complexity Table

Approach	Best Case	Average Case	Worst Case
Recursive	$O(2^n)$	$O(2^n)$	$O(2^n)$
Dynamic Programming	$O(n * W)$	$O(n * W)$	$O(n * W)$
Memoization	$O(n * W)$	$O(n * W)$	$O(n * W)$

Table 3

6. Code Complexity Analysis

6.1.1 Recursive Algorithm Code Complexity Analysis

Code Length

The recursive implementation is relatively short and straightforward. It involves a base case check and two recursive calls (one including the current item and one excluding it). The structure is clean and easy to follow, but it does not include any optimizations for overlapping subproblems, leading to redundant calculations.

Maintainability

The recursive solution is easier to understand, because it closely follows the definition of the knapsack problem. It is more maintainable if the problem does not involve large inputs or require optimization. However, due to the exponential time complexity, the recursive approach becomes inefficient for larger inputs, which can make maintenance problematic when scaling up.

Tradeoffs in Complexity

The recursive approach has an exponential time complexity of $O(2^n)$ because it explores all possible subsets of items. This makes it highly inefficient for large inputs, and the time taken can grow exponentially. Its simplicity, however, makes it a good choice for small inputs or as a conceptual starting point for understanding the knapsack problem.

Conclusion

Simple but inefficient for large problems, making it less maintainable for performance-critical applications.

6.1.2 Dynamic Programming Algorithm Code Complexity Analysis

Code Length

The DP implementation is longer due to the necessity of building and filling a 2D table to store the results of subproblems. The code requires nested loops to fill the table for every item and every possible capacity. Although

this approach introduces more lines of code compared to the recursive solution, it ensures that the problem is solved in pseudo-polynomial time.

Maintainability

The DP solution is more maintainable for larger problems because it ensures efficient computation by storing previously computed results. Although the code is longer, it is more scalable and can handle larger inputs without significant performance degradation. The iterative nature of the DP approach also makes it easier to debug (one stack frame) and modify, as all results are stored in a single table, and there is no need to worry about recursion depth or stack overflows.

Tradeoffs in Complexity

DP improves the time complexity to $O(n * W)$ by storing the results of subproblems in a table, ensuring each subproblem is solved only once. This approach is significantly more efficient than the recursive approach for larger inputs.

Conclusion

More complex, but the most efficient in terms of time for solving the problem, especially for large inputs. Best for scalability and maintainability.

6.1.3 Memoization Algorithm Code Complexity Analysis

Code Length

The memoized approach is sort of the *middle ground*. It retains the recursive structure, but with added complexity for storing and retrieving results from the `memo` dictionary (cache). This reduces the number of recursive calls, but still requires an additional dictionary to store intermediate results, which adds complexity in terms of code length.

Maintainability

Memoization is a compromise between recursion and the iterative DP approach. While it still uses recursion, the results are cached to prevent redundant computations. The use of a dictionary for memoization introduces complexity, especially if the data structures are not managed properly. Maintaining the dictionary and clearing it between calls can add complexity, but the memoized approach is generally easy to modify since it keeps the recursive structure intact.

Tradeoffs in Complexity

Memoization also reduces the time complexity to $O(n * W)$ by storing results of subproblems in a dictionary. While it still uses recursion, the dictionary prevents repeated computation of the same state, improving efficiency over the recursive approach. The main tradeoff here is the overhead of managing the dictionary and ensuring that cache hits and misses are handled correctly, which can increase the complexity of the code and memory usage compared to the DP approach.

Conclusion

A compromise between recursion and DP. While it retains the elegance of recursion, it requires additional management of the cache and introduces some overhead. Suitable for cases where recursion is *preferred* but performance optimization is still necessary.

---

7. Data Analysis

To begin, I implemented a function that reads the JSON data and encapsulates it into individual objects stored within a list. While this object-oriented approach introduced some additional overhead—such as object construction and data fetching, before running the algorithm, it offered advantages in terms of clarity and organization. An alternative approach would have been to generate the data directly within C# and store it as a raw array of integer values. However, I opted for the OOP method because it promotes better readability and conciseness in the code.

Each algorithm read in a list of objects (`items`), and an unsigned integer (`capacity`). From there the data analysis within the code was conducted differently for each algorithm to performance, correctness, and intermediate computations (DP, Memo).

Recursive Algorithm

In the recursive algorithm, data was analyzed by examining the depth of recursion and logging the decisions made at each step, such as including or excluding an item, to verify correctness and understand its exponential $O(2^n)$ time complexity and its enormous runtime for large data sets.

Dynamic Programming

For the dynamic programming solution, a 2D array (`dp`) was employed to systematically track the maximum values obtainable for each combination of items and capacities, allowing intermediate states to be analyzed iteratively and ensuring that edge cases, such as zero capacity or no items, were handled correctly. This approach demonstrated $O(n * W)$ complexity and provided insights into how the use of intermediate results helps us getting to the optimal solution.

Memoization

In the memoization algorithm, a dictionary (`memo`) was used to cache results for subproblems, enabling performance analysis through the observation of cache hits and misses. Recursive calls were tracked to ensure efficiency and correctness, with particular attention paid to reducing redundant computations. Debugging statements were used to see how caching works (cache hit and cache miss for key k)

---

8. Bottlenecks

Recursive

While simple and intuitive, the recursive approach has severe performance bottlenecks for large datasets due to redundant calculations and exponential time complexity. Its main limitation is its inability to handle large problem sizes efficiently.

Dynamic Programming

DP handles large datasets effectively by using an iterative approach and storing solutions to subproblems. Its primary bottleneck is space complexity due to the $n * W$ table, which may become prohibitive with large capacities.

Memoization

Memoization offers a good balance between recursion and dynamic programming. It reduces redundant calculations by caching intermediate results, but it introduces overhead with dictionary lookups and the potential for high memory usage. The approach still has the problem of deep recursion for large datasets, though it is more efficient than the pure recursive approach.

---

9. Impact of Knapsack Capacity

Note for the reader: The units on the `x-axis` differ for each plot

Recursive Algorithm

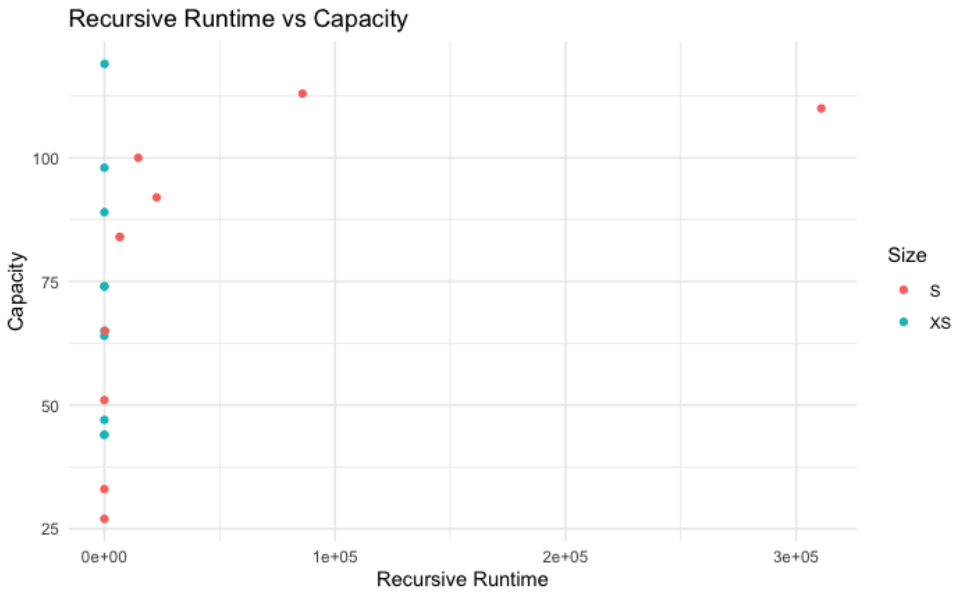

Figure 4 ⇒ Runtime vs Capacity

From the plotted data, it can be concluded that capacity (`W`) had minimal to no impact on the runtime of the algorithm for the `XS` dataset. In contrast, for the `S` dataset, capacity had a noticeable effect, which aligns with expectations. As capacity increases, more items must be considered for inclusion in the knapsack, resulting in a longer runtime. These observations are pretty consistent with my findings.

Dynamic Programming Algorithm

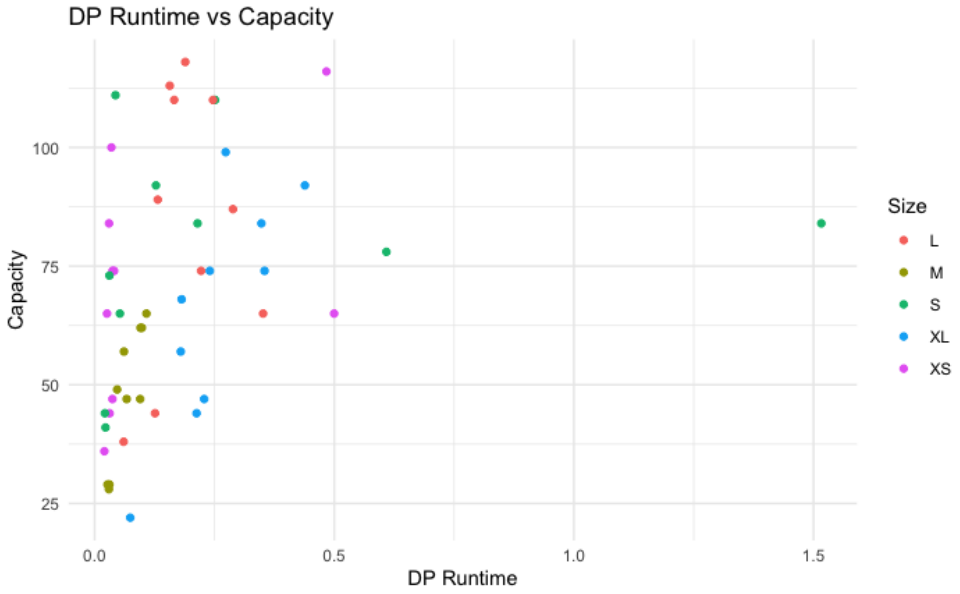

Figure 5 ⇒ Runtime vs Capacity

From the plotted data, we cannot definitively conclude that capacity (`W`) significantly impacts the runtime of the dynamic programming (DP) Algorithm. Across all datasets, the Algorithm demonstrated relatively consistent performance, with only a few outliers. This stability in runtime can be attributed to the structure of the DP approach: it precomputes results for all possible subproblems up to the given capacity in a systematic manner. As a result, the runtime is determined more by the number of items and the maximum capacity itself, rather than variations in the specific capacity values used in individual test cases.

Memoization Algorithm

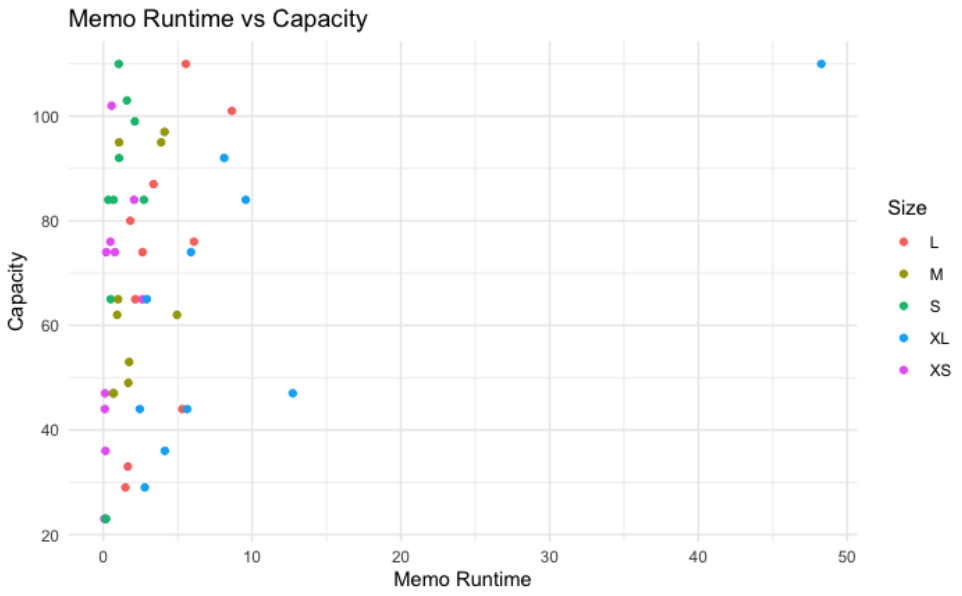

Figure 6 ⇒ Runtime vs Capacity

Similar to the DP Algorithm, the plotted data does not provide sufficient evidence to conclude that capacity significantly impacts the runtime of the Memoization Algorithm. The Algorithm demonstrates consistent performance across all datasets, which can be attributed to the fact that DP and Memoization are closely related in structure (*DP Table and Cache*). The primary difference between the two lies in the runtime increase, as Memoization uses recursion and stores previously computed results to avoid redundant calculations, whereas DP explicitly computes and stores all subproblem solutions in a table.

---

10. Conclusion

This project was loads of fun. I explored and compared different approaches to solving the Knapsack problem, specifically focusing on the Recursive Algortihm, Dynamic Programming (DP), and Memoization Algorithms. Through the analysis of runtime data across various capacity and item sets, I found that while the capacity had minimal impact on the runtime of the DP and Memoization Algorithms, where as the number of items had a notable impact. The runtime behavior of the Recursive approach was more sensitive to the problem size. Notably, both DP and Memoization Algorithms exhibited consistent performance, which is expected given their shared structural similarities. The key difference between the two lies in Memoization's use of caching, which helps avoid redundant calculations but leads to a slight increase in runtime compared to DP.

From my results, I can conclude that for larger datasets, the DP and Memoization approaches are way more efficient than the Recursive method, especially when the problem size and capacity increase. However, the consistent performance of both DP and Memoization across varying capacities highlights the efficiency of these methods in handling large numbers of items with minimal variation in runtime. I think that, ultimately, this project reinforces the importance of selecting the right algorithm based on the size and characteristics of the problem at hand.

I don't have much experience in the industry or field, but from what I've seen, algorithms are not always the go-to solution. I think the key question to ask is: Can we afford to sacrifice some resources and be off by a little bit at each step if it means a big performance boost? Or is getting the perfect solution every time more important than performance? If we are working on high-stakes projects like planes or rockets, we'd likely prioritize accuracy every time. But in cases like a shipping company packing containers or trucks, being a little off might not matter as long as it's within a reasonable threshold. This rabbit hole is deep! In order to really figure it out, we would need input from different departments within a company. So, I think we'll leave it at that for now `:)`.

---

11. References

Wikipedia contributors. (2024, November 29). *Knapsack problem: 0-1 knapsack problem*. **Wikipedia**. Retrieved from here

GeeksforGeeks. (2023, September 5). *0-1 knapsack problem using dynamic programming (DP)*. **GeeksforGeeks**. Retrieved from here

Warcoder. (2021, February 1). *Knapsack problem*. **Kaggle**. Retrieved from here

Colorado State University. (2018). *Knapsack problem and dynamic programming*. Retrieved from here (Need access with a key)

Vaccaro, P. (2005). *Theoretical and practical applications of the knapsack problem*. **Operations Research Letters**, 33(6), 621-630. Retrieved from here

Wippler, A., & Zimmerman, C. (2021). *Knapsack problem algorithms: Approaches and applications*. **Springer**. Retrieved from here

Stack Overflow contributors. (2010, April 12). *Why is the knapsack problem pseudo-polynomial?* **Stack Overflow**. Retrieved from here

---

Appendix

A. Utility Functions and Classes

- **Logger Class**

```
public class Logger { ... }
```

The `Logger` class serves two key functions. First, it logs essential information such as runtime, capacity, solution, and the type of algorithm used. Second, it records errors, warnings, and other debugging-related messages to aid in troubleshooting. To achieve this, the class establishes two distinct log paths: one for performance data and another for error tracking.

- `FPexecution_log.log` Logs all the necessary information about the algorithm/Algorithm.
- `FP_log.log` Logs all warnings, errors, informational messages that might pop up.

This class was created because the built in MS Logging library was simply above my needs for this project.

- **Maps Class**

```
public static class UtilityMaps { ... }
```

The `UtilityMaps` class functions as a central repo for the `Paths` and `Sizes` dictionaries. It was designed with the goal of maintaining clarity and enhancing readability throughout the codebase. By encapsulating these dictionaries within the class, the structure is kept organized, making it easier for you and future me to understand and manage the mapping of paths and sizes.

- **Knapsack Item Class**

```
public class KnapsackItem { ... }
```

The `KnapsackItem` class represents an item with its associated value and weight. It provides a constructor to initialize these properties, ensuring that the weight and value remain unchanged throughout the program's execution. This immutability helps maintain data integrity, preventing accidental modification of item attributes. By encapsulating the weight and value in this class, the design promotes clarity and modularity, making it easier to manage and work with individual knapsack items within the larger algorithm.

- **Capacity Item Class**

```
public class CapacityItem { ... }
```

The `CapacityItem` class represents a single capacity value and was designed to maintain consistency within the overall structure of the program. It follows the same design pattern as the `KnapsackItem` class, ensuring a uniform approach to managing data across the different components of the algorithm.

- **Knapsack Class**

```
public class Knapsack { ... }
```

The `Knapsack` class contains all the available methods used to solve the knapsack problem. Additionally, the class includes utility functions that aid in debugging and tracking the behavior of each algorithm throughout the process. This structured approach ensures that the class is versatile and capable of handling various algorithmic solutions while providing insights into their performance.

- `ReadItemsFromJsonFile`

```
public static List<T>? ReadItemsFromJsonFile<T>(string filePath) { ... }
```

This function is responsible for reading the JSON data into lists, which are then used later in the program. By parsing the JSON input, it ensures that the data is organized and ready for processing.

- `PrintItems` & `PrintCapacities`

```
public static void PrintItems(List<KnapsackItem>? items) { ... }
public static void PrintCapacities(List<CapacityItem>? capacities) { ... }
```

These functions were essential for debugging by ensuring the data was correctly parsed and extracted into the appropriate variables. They helped verify that the JSON data was accurately converted into usable objects and lists.

- **Stopwatch**

```
 public static void Stopwatch(CancellationToken cancellationToken) { ... }
```

The purpose of this function is to start the stopwatch before running all three methods five times and stop the stopwatch once the execution is complete, providing the total execution time for 15 method executions (the total elapsed time). This approach allowed me to monitor the algorithm's runtime directly within the program, without relying on any external applications or websites.

- `YieldRandomCapacity`

```
public static CapacityItem? YieldRandomCapacity(List<CapacityItem>? capacities) {...}
```

This function randomly selects a capacity from a list of 10, ensuring that only one capacity is chosen before executing all three algorithms. This guarantees complete randomness in the selection process before the algorithms run.

- `LogResults`

```
private static void LogResults(...) { ... }
```

This function formats the logging output and records the results using the provided parameters. It ensures the data is logged in a consistent format for easy tracking and analysis throughout the program.

- **Data Collection**

```
public static void CollectData(Logger logger) { ... }
```

This function is the core of the program. It measures and logs the results for each method, ensuring that all relevant data is captured for analysis. The full implementation is available on GitHub with the following path: `CPSC4100/Knapsack.cs`

Lines

- **568 - 632**

Pseudocode

```
Function CollectData(logger):
    Initialize stopwatch
    Define constants: CapacityFile, testNumber
    Define methods array with algorithm names and functions

    For each method in methods:
        For each itemKey in Paths.Keys excluding CapacityFile:
            Read items and capacities from respective JSON files

            Yield random capacity from capacities list
            If method is "Recursive" and itemKey is "XL", "L", or "M":
                Log default values (no execution)
                Increment testNumber
                Continue to next iteration

            Start stopwatch
            Execute algorithm with items and unpacked capacity
            Stop stopwatch
            Measure execution time

            Log results (execution time, solution)
            Increment testNumber
```

B. Additional Graphs

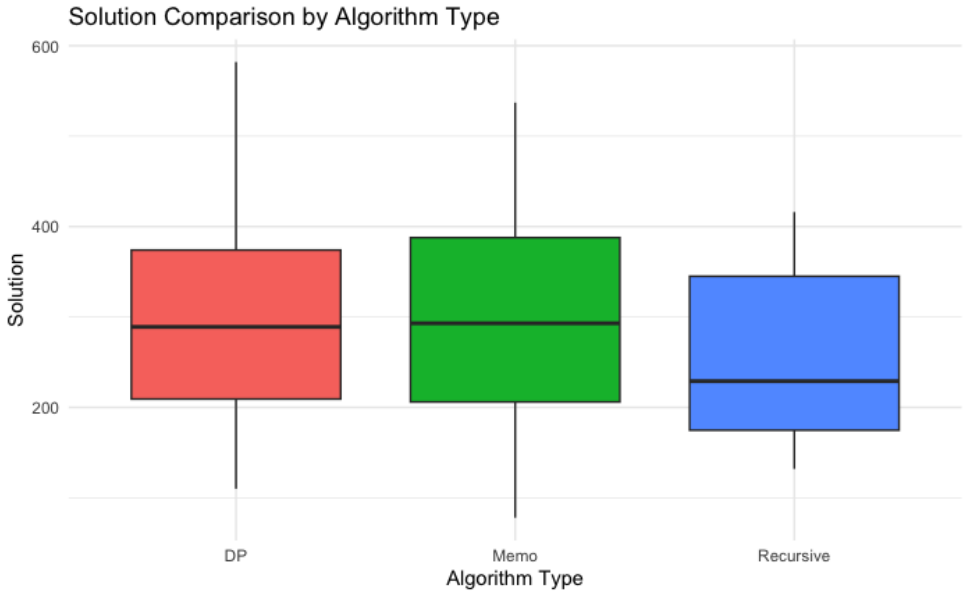

Figure 7 ⇒ Solution Comparison

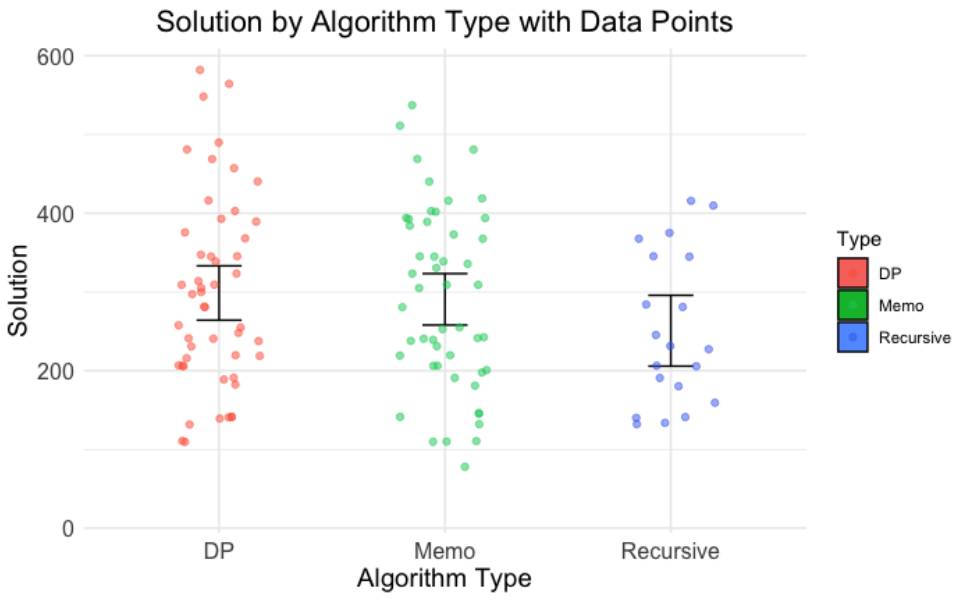

Figure 8 ⇒ Data Points

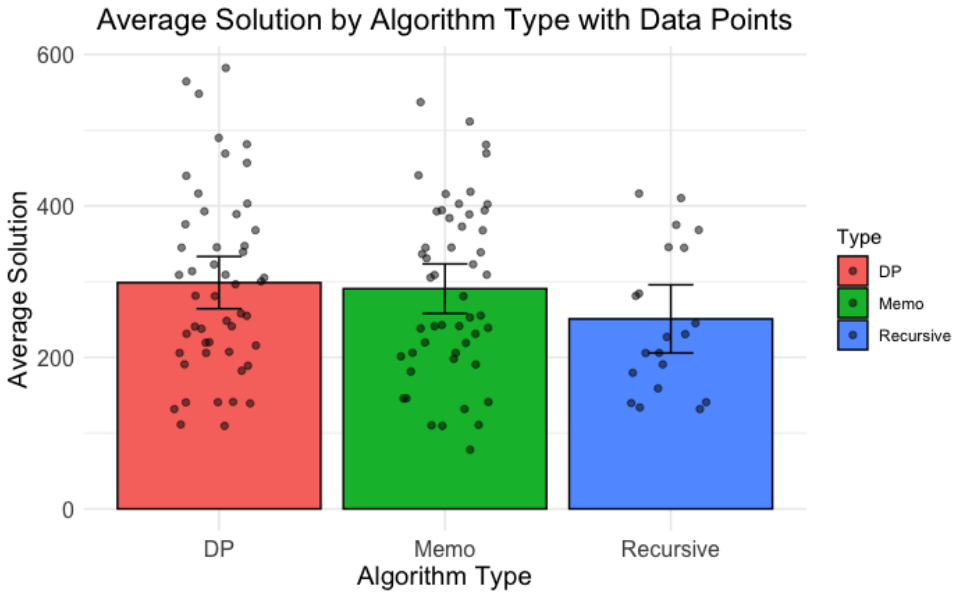

Figure 9 ⇒ Mean Solution with respect to Method Type

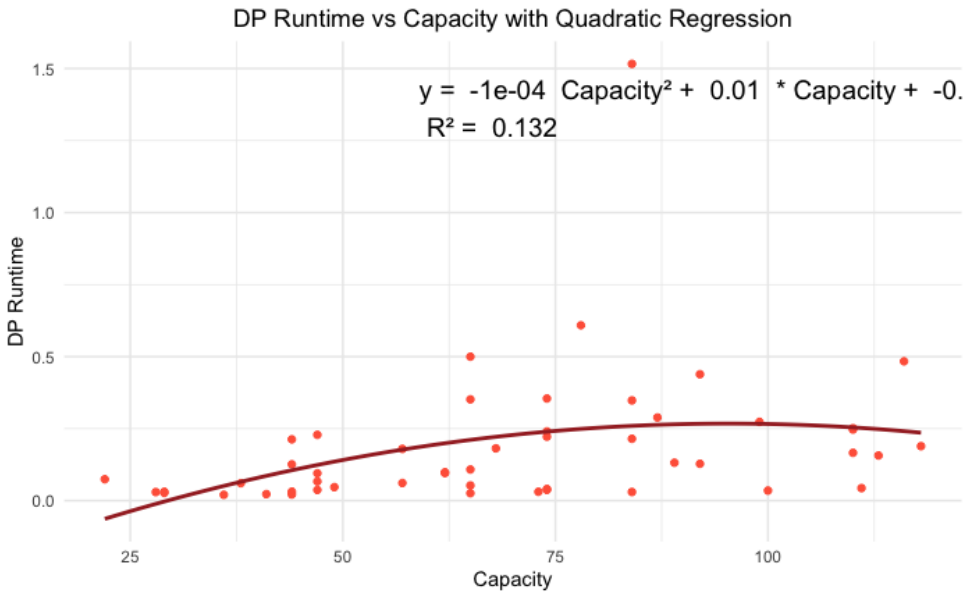

Figure 10 ⇒ Runtime ~ Capacity

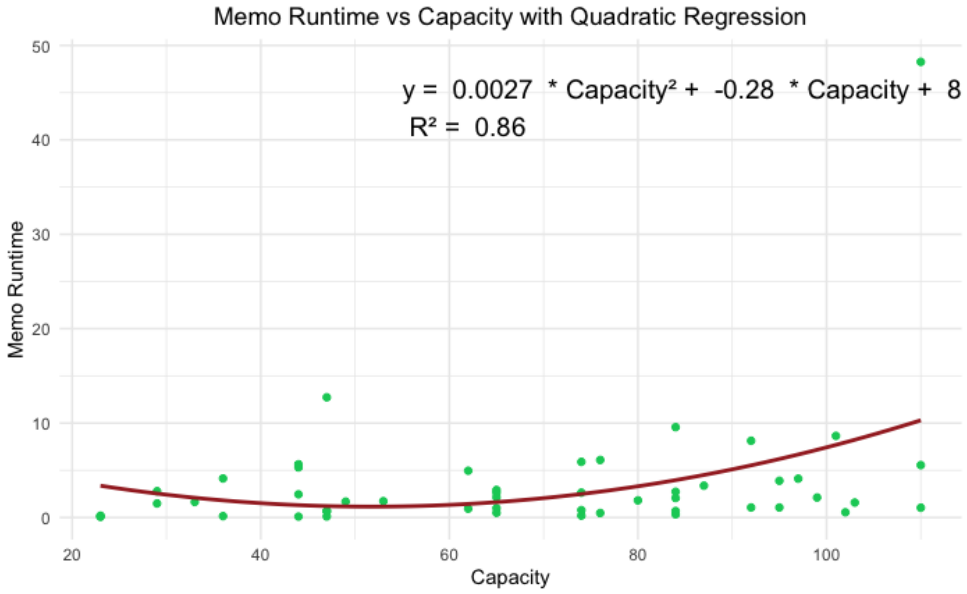

Figure 11 ⇒ Runtime ~ Capacity

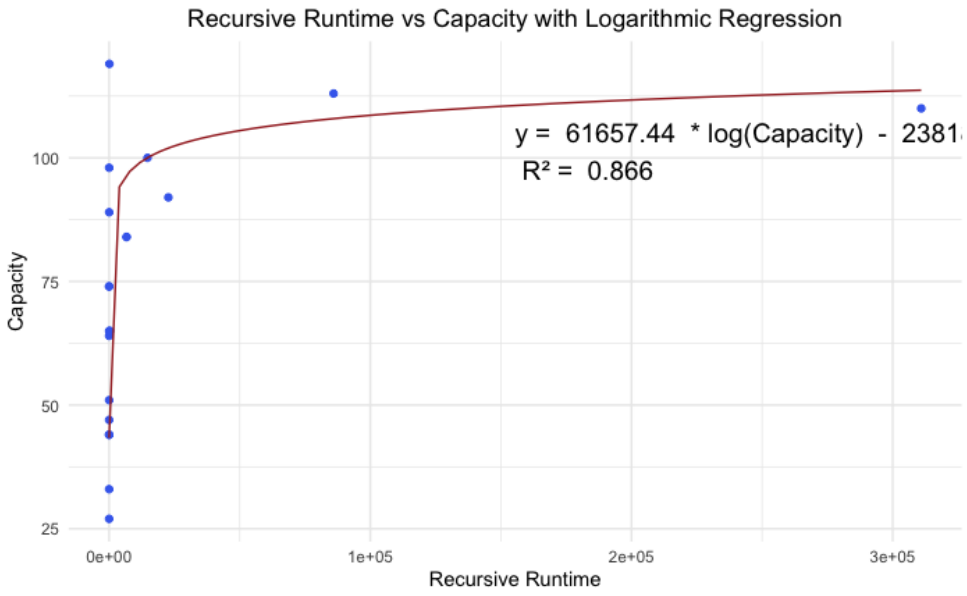

Figure 12 ⇒ Runtime ~ Capacity

C. Unit Testing

Unit testing was performed using the MSTest library, with results available in the "Unit Tests" sheet of the `Results.xlsx` file. To review the specific unit tests, visit the `FPUnitTests` submodule in the GitHub repository.

D. GitHub

Link to WebPage https://github.com/j-balkovec/CPSC4100

Jakob Balkovec (Seattle University)