CMSC142 - Design and Analysis of Algorithms Machine Problem 2: An Exploration on Dynamic Programming and Greedy Methods to the 0/1 Knapsack Problem

Katrina Ang [*] and Elaine Pajarillo [*]

Department of Mathematics and Computer Science, University of the Philippines - Baguio
[*]Corresponding author: knang@up.edu.ph
[*]Corresponding author: etpajarillo@up.edu.ph

Abstract

In this paper, the computational efficiency of five different dynamic programming and greedy algorithms (Bottom-Up, Top Down, Largest Value, Smallest Size, and Greatest Worth Ratio) for solving the classic combinatorial optimization challenge, the 0/1 Knapsack Problem is evaluated. The experiment aimed to empirically analyze how execution time may scale with increasing item counts for algorithms with faster time complexities than the traditional brute-force approach. Using Python-based implementations of each code, the optimal solutions for a number of items from the range of 100 to 100,000 were obtained. Random item weights of 100 to $1,500$ and values of 100 to 500 were considered with a fixed knapsack capacity of 1,000. The experimental findings reveal that Bottom-Up Dynamic Programming is preferred over Top-Down as n increases, and Greedy Algorithms are preferred for faster runtime execution than Dynamic Programming, with a trade-off of possible less optimal solutions for the problem. The analysis also revealed that while some algorithms share the same theoretical time complexities, the practical runtime performances yield significant differences, mainly due to their structure. This conclusion pushes the notion that the evaluation of theoretical and practical trade-offs between different algorithms should be considered when making informed decisions for designing solutions to optimization problems. Further improvements in dynamic programming structures or use of more advanced algorithms and approximation schemes such as the Branch-and-Bound technique and FPTAS scheme may be considered for even more efficient executions for solutions to optimization problems such as these.

1 Introduction

Optimization problems that involve logically looking for the most efficient solution based on a situation often play crucial roles in a wide array of fields in computer science and mathematics, often requiring either the maximization or minimization of particular attributes or involving concepts such as permutations, combinations, and/or subsets (Levitin, 2012). Such problems are useful not only in theoretical considerations, but in real-life extensions such as resource allocation or decision making.

As explored in the previous machine problem, a systematic finite approach to solving such problems are exhaustive search or brute force methods. However, while they can run and solve the given problems, the results are almost never cost-efficient in regards to runtime allocations as the number of items n grows larger. To overcome this, other algorithmic approaches can be identified to solve optimization problems in a more efficient time range. Of these different approaches, we aim to explore the general performance and efficiency of dynamic programming and greedy algorithm approaches — namely: dynamic programming using the bottom-up and top-down methods, and greedy algorithms that take the item of largest value first, the item of smallest size first, and lastly one which takes the item of greatest worth ratio first (approximation algorithm).

To evaluate the efficiencies of the different algorithms, we are to use the same well-known NP-hard optimization problem: the 0/1 Knapsack problem. This optimization problem prompts the user to select a subset of items $x_1, x_2, ..., x_n$ in such a way that the total weight w_n of each of the n items with differing values $v_1, v_2, ..., v_n$ do not exceed the knapsack's capacity W while maximizing the profit margin that may be gained for the subset. Describing this formally,

$$
\begin{aligned}
&\text{maximize} \quad \sum_{i=1}^{n} v_i x_i \\
&\text{subject to} \quad \sum_{i=1}^{n} w_i x_i \leq W \qquad (1)\\
&x_i \in \{0,1\}, \quad \forall i \in \{1,2,\ldots,n\}
\end{aligned}
$$

In the case of the 0/1 knapsack problem, the values of x are limited to 0 or 1 only, with 0 indicating you leave the entire item or take the entire item inside the knapsack, as there is no option to take only a fraction of the item in the case of this problem (Jooken et al., 2023).

Dynamic programming addresses the 0/1 Knapsack problem by considering smaller, more manageable sub-instances of the original problem (Levitin, 2012). This method of solving optimization problems aim to efficiently avoid the redundant calculations, often seen in brute-force approaches such as in the exhaustive search method, theoretically leading to more efficient run-times, especially in larger data sets. In particular, there are two primary approaches in the implementation of dynamic programming in solving the 0/1 knapsack problem: the bottom-up approach, and the top-down approach.

Dynamic programming implemented through the bottom-up approach involves iteratively constructing a table representing all possible sub-problems. Starting from the simplest cases and working up to the optimal solution to the knapsack problem, bottom-up dynamic programming builds upon the idea that the solution of any subproblem depends on the solution to smaller sub-problems (Gautam, 2023).The bottom-up approach then sorts the sub-problems by input size and solves iteratively form the smallest value. In terms of its algorithmic process in solving the 0/1 knapsack problem, a two-dimensional array is created with rows representing the weights and items. Each cell is used to store the maximum value possible based on the capacity of the knapsack. The table is then iteratively filled by determining whether to include the item through comparing its value with the capacity and maximum achievable value (note that if an item's weight exceeds the current capacity, the algorithm carries forward the value from the previous rows). After the table has been fully populated, the maximum achievable value based on the set knapsack capacity should be found (GeeksforGeeks, 2024).

Conversely, dynamic programming implemented through the top-down approach involves breaking the problem down into its smaller sub-problems recursively. It utilizes a concept called memoization to create a table that stores previously computed results. The recursive function takes into account this table, the current item index, and the remaining knapsack capacity. The top-down approach first identifies any base cases, or the simplest instance of the knapsack problem that can be solved directly. Building upon this, the algorithm evaluates two options: including the item being evaluated or excluding it. Before returning the result, the function stores it in the memoization table. This is done so that if the same subproblem is encountered during recursion, the function will retrieve it from the table instead of recomputing the value. This process continues until all items and their inclusion/exclusion have been considered.

For the greedy algorithms considered in this paper, the general strategy is to make a choice that will seem the "best" or a "greedy" grab at that stage of evaluation from a sequence of steps, aiming to find the global optimal solution from these local optimum choices. Note that each greedy grab identified must be feasible, locally optimal, and irrevocable (Levitin, 2012).

There are many different approaches to implement the greedy algorithm in solving the 0/1 knapsack problem. Among these approaches, the first considered in this paper is the greedy approach that prioritizes taking the item of largest value first. The items to be considered for the knapsack are sorted first in descending order, that is, the highest value comes first. The algorithm iterates through this sorted list and checks if each item can fit into the knapsack's remaining capacity, skipping those items that cannot fit into the knapsack. This greedy algorithm stops only when no more items can be added into the knapsack, or if all available items have been considered.

The next greedy algorithm considered is one that selects items based on the smallest weight first. This approach aims to prioritize the items taking up the least space in the knapsack. To do so, the items are sorted by weight in ascending order (the lightest items are first in line). This sorted list is then viewed one item at a time, where the item is included based on the remaining capacity of the knapsack. Similar to the previous approach, the algorithm stops only when no more items can be added or if all items have already been considered.

The last greedy algorithm to be considered is the backpack approximation algorithm, which aims to solve the 0/1 knapsack problem through prioritization of items that offer the highest value-to-weight ratio. This ratio is defined as (value/weight). This approaches the problem by computing the ratio for each item and sorting the results in descending order, where the highest ratios are first in the list to be considered. The algorithm then iterates through this list and follows the other two greedy approaches in determining whether the item is included or not, exiting only when the knapsack is full or if all items ratios have been considered.

This paper aims to investigate the efficiency of these algorithms in solving optimization problems such as the 0/1 Knapsack problem. To do so, an empirical analysis on the different algorithms described above will be implemented on different input sizes n for the knapsack problem. The average runtime of each algorithm in obtaining the optimal solution for each n amount of items across three trials will be then be taken into account. For the purpose of this analysis, the knapsack's capacity W will be fixed to 1000, the weight w of the items are random positive integers ranging 100 to 1,500, the value v of the items are set to random positive integers from 100 to 500, and the number of items n are from 100 to 100000.

The following portion of the paper will discuss the exact specifications, methods, and processes used by the students to achieve the analysis of each algorithm, based on the knapsack specifications stated above. This will include the algorithms used, the experiment setup, the generation of inputs, and the tools and functions used to measure runtime.

2 Methodology

This section details the approach taken to empirically analyze the time efficiencies of the different algorithms for solving the 0/1 Knapsack problem with the given initial parameters.

2.1 Algorithm Overviews

2.1.1 Dynamic Programming: Bottom-Up

The algorithm using dynamic programming through a bottom-up approach aims to obtain a solution to the 0/1 Knapsack problem as detailed in the introduction section of the paper. Python version 2.6 was utilized in applying and implementing the solutions to the 0/1 Knapsack problem. the following functions were used in the code's implementation. Additionally, the basic functions of the knapsack algorithm were taken from Andrew G. Evan's github account and modified to fit the specifics of the knapsack problem evaluated in this paper.

The `item_save_function` serves to randomly generate weights and values for the items to be considered for the knapsack. Note that the `seed` package is utilized to ensure that the randomly generated numbers are the same for every trial, every time that the code runs. The items are stored in a list, and each trial's generated data is saved into a file using the `pickle` library. Saving the data into a file allows the same data set to be reused across all trials.

The pre-saved item data is loaded from the corresponding `.pkl` file and the implementation of the knapsack dynamic programming begins. The `knapSack` function uses a two-dimensional table V where `V[i][w]` stores the maximum obtainable value using the first `i` items. In algorithmic terms, the recurrence relation observable in the code can be seen as

$$V[i][w] = \begin{cases} V[i-1][w] & \text{if weight}[i-1] > w \\ \max(\text{value}[i-1] + V[i-1][w - \text{weight}[i-1]], V[i-1][w]) & \text{if weight}[i-1] \leq w \end{cases} \tag{2}$$

Next, the runtime measurement `measure_runtime` is a function that will remain relatively the same throughout all algorithms. For a given trial and number of items, the measure runtime function measures the time taken to solve the knapsack problem using the `time.time` python function and return the maximum value and runtime. Note that this method does not consider the time taken to generate the item data used in the knapsack problem. After this, the number of n values is computed and saved for all three trials, keeping note of the runtime for each number of items. The average runtime across all trials is computed for each n value, and in order to aid in visualization, a graph of this function is plotted using the `matplotlib` python library.

Python Code for DP Bottom-Up Approach for the 0/1 Knapsack Problem Algorithm

```python
import random
import time
import pickle
import matplotlib.pyplot as plt

#seed fcn - same random inputs generated
random.seed(42)

W = 1000 #knapsack capacity

#generate random weights and values (save for reuse)
def item_save_function(n, num_trials=3):
    for trial in range(1, num_trials + 1):
        random.seed(42 + trial)  #diff seed per trial
        weights = [random.randint(100, 1500) for _ in range(n)]
        values = [random.randint(100, 500) for _ in range(n)]
        items = list(zip(weights, values))
        filename = f"items_trial{trial}.pkl"  #save data (for reuse in other algos)
        with open(filename, "wb") as f: pickle.dump(items, f)
```

Python Code for DP Bottom-Up Approach for the 0/1 Knapsack Problem Algorithm (continued)

```python
#load pre-saved items for each trial
def load_items(trial):
    filename = f"items_trial{trial}.pkl"
    with open(filename, "rb") as f:
        items = pickle.load(f)
    return items

def knapSack(W, wt, val, n):
    # Create dp table
    V = [[0 for _ in range(W + 1)] for _ in range(n + 1)]

    for i in range(1, n + 1):
        for w in range(W + 1):
            if wt[i - 1] <= w:
                V[i][w] = max(val[i - 1] + V[i - 1][w - wt[i - 1]], V[i - 1][w])
            else:
                V[i][w] = V[i - 1][w]

    #return max val
    return V[n][W]

#runtime
def measure_runtime(trial, n):
    items = load_items(trial)
    wt, val = zip(*items)

    start_time = time.time()
    result = knapSack(W, wt, val, n)
    end_time = time.time()

    runtime = end_time - start_time
    return result, runtime

#different n values
n_values = [100, 1000, 10000, 50000, 100000]
num_trials = 3

for n in n_values:
    item_save_function(n, num_trials)

average_runtimes = {n: [] for n in n_values}

for trial in range(1, num_trials + 1):
    print(f"Trial {trial}:")
    for n in n_values:
      result, runtime = measure_runtime(trial, n)
      average_runtimes[n].append(runtime)
      print(f"n = {n}, Maximum Value = {result}, Runtime = {runtime:.4f} seconds")

#averaging and plotting
average_runtimes = {n: sum(times) / len(times) for n, times in
average_runtimes.items()}
```

Python Code for DP Bottom-Up Approach for the 0/1 Knapsack Problem Algorithm (continued)

```
plt.figure(figsize=(10, 6))
plt.plot(list(average_runtimes.keys()),
list(average_runtimes.values()), marker='o')
plt.title('Average Running Time vs Number of Items')
plt.xlabel('Number of Items (n)')
plt.ylabel('Average Runtime (seconds)')
plt.xscale('log')
plt.grid(True)
plt.xticks(list(average_runtimes.keys()))
plt.show()
```

2.1.2 Dynamic Programming: Top-Down

The algorithm using dynamic programming through a top-down approach aims to obtain a solution to the 0/1 Knapsack problem as detailed in the introduction section of the paper. The general algorithm for the top-down approach was taken from Javier Ribera's github account and further modified to fit the constraints of the paper's problem.

Similar to the bottom-up approach, the algorithm for the top-down follows the same generation of inputs using the `seed` function and saving using the `pickle` package. The measurement of runtime also follows the bottom-up approach using `time.time`. The differences between the approaches lie in the `knap_sack_top_down` function. This function solves the 0/1 knapsack problem through memoization, with the base case being when there are no items left (value 0). A 2D array is created to store the results for the different subproblems, and with each iteration of the item list, the memoization table is first checked. If the result has previously been computed, the stored value is directly returned. Following this, the weight and the current capacity is checked before choosing to include or exclude the item. The result of this is then stored in a separate "store" table. Lastly, at the end of the algorithm, the maximum value is returned.

Additionally, another difference between the two dynamic programming approaches would be the importation of the `sys` library in order to increase recursion depth. This is used in order to accommodate the recursion errors that occur when the number of n items increase. Recursion errors occur in Python codes when a function exceeds the maximum recursion depth allowed by its interpreter (StudySmarter, n.d.). This error was observed to occur using larger values of n. Although measures were enforced to bypass this, the large number of items n combined with the recursive nature of the algorithm still lead to such errors. As such, the estimated runtime values for $n = 100000$ will be extrapolated through Google Sheets.

For the extrapolation of these data points, the following steps were performed: Firstly, the average runtimes across the three trials and their corresponding n item values from the range $n = 100$ to $n = 10000$ were imported from Google Colab into Google Sheets. The method of extrapolation for the data points given by the Python code through Google Sheets follows that detailed by Stuart Shumway (2019). The existing data is first visualized into a graph and the resulting chart is used to extract a trend formula, which will then be used for the extrapolation of data for $n = 100000$.

Python Code for DP Top-Down Approach for the 0/1 Knapsack Problem Algorithm

```
import random
import time
import pickle
import matplotlib.pyplot as plt
import sys

sys.setrecursionlimit(1000000)

random.seed(42)
```

Python Code for DP Top-Down Approach for the 0/1 Knapsack Problem Algorithm (continued)

```python
#knapsack capacity
W = 1000

def save_items_for_trials(n, num_trials=3):
    for trial in range(1, num_trials + 1):
        random.seed(42 + trial)  # Different seed for each trial
        weights = [random.randint(100, 1500) for _ in range(n)]
        values = [random.randint(100, 500) for _ in range(n)]
        items = list(zip(weights, values))
        filename = f"items_trial{trial}.pkl"
        with open(filename, "wb") as f:
            pickle.dump(items, f)

def load_items(trial):
    filename = f"items_trial{trial}.pkl"
    with open(filename, "rb") as f:
        items = pickle.load(f)
    return items

#knapsack dp top-down approach
def knap_sack_top_down(w, weights, values, n, store):
    if n == 0 or w == 0:
        return 0
    if store[n][w] is not None:
        return store[n][w]

    if weights[n - 1] > w:
        result = knap_sack_top_down(w, weights, values, n - 1, store)
    else:
        result = max(values[n - 1] + knap_sack_top_down(w - weights[n - 1],
        weights, values, n - 1, store),
                     knap_sack_top_down(w, weights, values, n - 1, store))

    store[n][w] = result
    return result

#runtime
def measure_runtime_top_down(trial, n):
    items = load_items(trial)
    weights, values = zip(*items)

    #memoization table
    store = [[None for _ in range(W + 1)] for _ in range(n + 1)]

    start_time = time.time()
    result = knap_sack_top_down(W, weights, values, n, store)
    end_time = time.time()
    runtime = end_time - start_time
    return result, runtime

#plotting
def plot_average_runtimes(n_values, avg_runtimes, title):
    plt.figure(figsize=(10, 6))
    plt.plot(n_values, avg_runtimes, marker='o', label='Average Runtime')
    plt.xlabel('Number of Items (n)')
    plt.ylabel('Average Runtime (seconds)')
```

Python Code for DP Top-Down Approach for the 0/1 Knapsack Problem Algorithm (continued)

```python
    plt.title(title)
    plt.legend()
    plt.grid(True)
    plt.xticks(n_values, labels=[f"$10^{int(len(str(n))-1)}$" for n in n_values])
    plt.show()

#trials
n_values = [100, 1000, 10000]
num_trials = 3

for n in n_values:
    save_items_for_trials(n, num_trials)

runtimes = {n: [] for n in n_values}
for trial in range(1, num_trials + 1):
    print(f"Trial {trial}:")
    for n in n_values:
        result, runtime = measure_runtime_top_down(trial, n)
        runtimes[n].append(runtime)
        print(f"n = {n}, Maximum Value = {result}, Runtime = {runtime:.4f} seconds")

#averaging runtimes
avg_runtimes = [sum(runtimes[n]) / num_trials for n in n_values]

#plotting
plot_average_runtimes(n_values, avg_runtimes, "Average Runtime vs Number of Items
(Top-Down Knapsack)")
```

2.1.3 Greedy Algorithm

The algorithm using the greedy approach aims to provide an efficient solution to the problem by making locally optimal choices at each step to arrive at a globally optimal solution. The general structure of the greedy algorithm was adapted from Javier Ibera's github account and further customized to align with the specific constraints and objectives outlined in the problem statement of this paper.

Three variations of the greedy approach were designed based on the problem description, with each variation differing in how items are prioritized and sorted for inclusion in the knapsack. The first variation, **Greedy Algorithm 1**, prioritizes items with the largest value, sorting them in descending order by value. This approach iteratively selects items from the sorted list, ensuring that the total weight does not exceed the knapsack's capacity, aiming to maximize the total value within the given weight constraint. The second variation, **Greedy Algorithm 2**, focuses on the smallest-sized items first by sorting them in ascending order by weight. This method seeks to fit as many items as possible into the knapsack, favoring scenarios where the quantity of items is more important than their individual value. The third variation, is the **Backpack Approximation Algorithm**, that calculates the worth-to-weight ratio for each item and sorts them in descending order by this ratio. By prioritizing items with the highest relative worth, this approach balances value and weight efficiency, often producing a closer approximation of the optimal solution. These three variations were implemented and evaluated using randomly generated datasets to analyze their performance in terms of total value achieved and computational efficiency, highlighting the adaptability of the greedy approach to various problem contexts and constraints.

The code used for the greedy approach in solving the given problem are the same for all of the three variations, with the exception of one line for the sorting of the items in the knapsack. The code shown below is the code for **Greedy Algorithm 1** that sorts the weights in ascending order and reverses it.

Python Code for Greedy Approach 1 for the 0/1 Knapsack Problem Algorithm

```python
import random
import time
import matplotlib.pyplot as plt

def knapsack_greedy_weight(items, max_weight):
    items_sorted = sorted(items, key=lambda x: x[1], reverse=True)
        # Sort by weight first then reverse
    knapsack, total_weight, total_value = [], 0, 0

    for item in items_sorted:
        if total_weight + item[0] <= max_weight:
            knapsack.append(item)
            total_weight += item[0]
            total_value += item[1]

    return knapsack, total_weight, total_value

# Experiment with 3 trials each
def run_experiment_value(item_counts, max_weight=1000):
    results = []
    for trial in range(3):
        trial_results = []
        for n in item_counts:
            items = [(random.randint(100, 1500), random.randint(100, 500))
            for _ in range(n)]
            start_time = time.time()
            _, _, max_value = knapsack_greedy_weight(items, max_weight)
            runtime = time.time() - start_time
            trial_results.append((n, max_value, runtime))
        results.append(trial_results)
    print_experiment_results(results)
    return results

# Print results
def print_experiment_results(results):
    for trial_index, trial_results in enumerate(results):
        print(f"Trial {trial_index + 1}:")
        for n, max_value, runtime in trial_results:
            print(f"n = {n}, Maximum Value = {max_value},
            Runtime = {runtime:.4f} seconds")

# Test parameters
item_counts = [100, 1000, 10000, 100000]
results_value = run_experiment_value(item_counts)

# Visualization
plt.plot(item_counts, [sum(trial[i][2] for trial in results_value)
    / len(results_value) for i in range(len(item_counts))],
         marker='o', label="Largest Value First")
plt.xlabel("Number of Items")
plt.ylabel("Average Running Time (seconds)")
plt.title("Greedy Algorithm 1: Largest Value First")
plt.legend()
plt.xscale("log")
plt.grid()
plt.show()
```

The code for the **Greedy Algorithm 2** changes the way the items are sorted by changing a single line in the specific function below and uses the same exact code as the **Greedy Algorithm 1** above, the change is shown below.

```
...

def knapsack_greedy_weight(items, max_weight):
    items_sorted = sorted(items, key=lambda x: x[0])
            # Sort by weight (smallest first)
    knapsack, total_weight, total_value = [], 0, 0
     ...
    return knapsack, total_weight, total_value

...
```

The **Backpack Approximation Algorithm** also employs the same change in that specific line to follow this variation's constraints while also using the same code as the **Greedy Algorithm 1** above, the change is shown below.

```
...

def knapsack_greedy_weight(items, max_weight):
    items_sorted = sorted(items, key=lambda x: x[1] / x[0]
      if x[0] > 0 else 0, reverse=True)
            # Sort by value/weight
    knapsack, total_weight, total_value = [], 0, 0
     ...
    return knapsack, total_weight, total_value

...
```

2.2 Experiment Setup

Similar to Machine Problem 1, the experimental setup involved establishing the parameters for the knapsack problem and configuring the conditions for running the algorithm. The knapsack capacity was fixed at 1000 which represents the maximum capacity it could hold. However, given the more agreeable nature of the algorithms up for evaluation for this paper to larger test data, the weight and values of the items have significantly grown. In specific, the weight of the items considered are now random positive integers ranging from 100 to 1,500, and the values for these items are now random positive integers from 100 to 500. Additionally, the number of items n to choose from are now $n = 100$ and $n = 100000$.

The experiment's test trials were performed and run on Google Colab, an online compiler hosted by the Jupyter Notebook service. This online service allows a significant execution speed for the given code above, as it provides more powerful cloud-based hardware with higher computing capabilities than previously specified locally (Google Colaboration, n.d.). In specific, the Google Colab runtime environment ran the algorithm using Python 3 with system RAM identified as 12.7 GB and a 107.7 GB disk storage.

The experiment aimed to test the algorithm's run-time performance using different input sizes among different algorithms, specifically selecting subsets of items ranging from 100 to 100,000. This approach allows for a detailed analysis of how the execution time scales with the number of items considered in the knapsack problem.

2.3 Tools & Methods in Measuring Runtime

The execution time of the algorithm was measured using the `time` module in Python, which provides a simple way to track how long the algorithm takes to run. The process involved capturing the time immediately before starting the execution of the algorithm and then capturing it again immediately after the algorithm completed its execution. The difference between these two timestamps gives the code's total execution time in seconds.

This approach was executed using these specific lines of code.

```python
start_time = time.time()
end_time = time.time()

execution_time = end_time - start_time
```

In addition to these methods, the `matplotlib` Python library was used in plotting the averages of the trials. In particular, the `pyplot` sub module was imported and served this function under the `plt` alias in order to create the graphs and their plotting areas. This function also enables the legends and labeling of all graphs seen under section 3.1 of the results and discussion (3).

The section of code used for the plotting for most of the graphs are as follows, with the titles and scales being edited to fit each algorithm's parameters.

```python
plt.figure(figsize=(10, 6))
plt.plot(list(average_runtimes.keys()), list(average_runtimes.values()), marker='o')
plt.title('Average Running Time vs Number of Items')
plt.xlabel('Number of Items (n)')
plt.ylabel('Average Runtime (seconds)')
plt.xscale('log')
plt.grid(True)
plt.xticks(list(average_runtimes.keys()))
plt.show()
```

In addition to this, the use of the `plt.xscale('log')` function was utilized in order ensure that the x-axis scales are shown logarithmically. This was done in order to accommodate the logarithmic range that was chosen for the test cases for experimentation, which will be further explored in the next subsection.

Note that the plotting of the graph for the dynamic programming through top-down approach is found in a separate code section than the main algorithm, in order to account for the extrapolated value done in Google Sheets. In order to ensure consistency between the graphing program, the extrapolated value from Google Sheets was imported into the Python environment and graphed using the following code

```python
#CODE FOR PLOTTING FULL DP TOPDOWN GRAPH
import matplotlib.pyplot as plt

n_values = [100, 1000, 10000, 100000]
avg_runtimes = [0.0314, 0.7025, 15.4043, 961.6658]

def plot_average_runtimes(n_values, avg_runtimes, title):
    plt.figure(figsize=(10, 6))
    plt.plot(n_values, avg_runtimes, marker='o', label='Average Runtime')
    plt.xlabel('Number of Items (n)')
    plt.ylabel('Average Runtime (seconds)')
    plt.title(title)
    plt.legend()
    plt.grid(True)
    plt.xticks(
        n_values,
        labels=[f"$10^{int(len(str(n))-1)}$" for n in n_values]
    )
    plt.xscale("log")
    plt.show()

plot_average_runtimes(n_values, avg_runtimes, "Average Runtime vs Number of Items (Top-Down Knapsack)")
```

2.4 Recording of Results

The results from each test case were recorded for analysis by taking into consideration the nature of the algorithms handled throughout this paper being more agreeable to larger test cases than that of the exhaustive search algorithm explored in machine problem 1. Specifically,the current test case ns for this experiment were identified to be $n = 100, n = 1000, n = 10000$, and $n = 100000$ or, in logarithmic terms, $n = 10^2, n = 10^3, n = 10^4$, and $n = 10^5$. Each algorithm was run on three trials for each test case to account for the variability in execution time due to any external factors such as system load, random number generation, and other hidden processes in the system.

The logarithmic nature of the range adapted for experimentation was chosen based from Korolivska's 2019 evaluation that logarithmic scales are often better to utilize to resolve any issues with data visualization when considering large datasets, naturally better enabling the identification and evaluation of patterns and relationships between data.

After running each algorithm, the run time it took for each test case was collected. These results were tabulated for each test size and each repetition, allowing for an average execution time to be calculated to provide a clearer picture of how the algorithm's performance scales with increasing input sizes.

3 Results and Discussion

This chapter presents the analysis of the data. Similar to machine problem 1, all gathered data from the experiments were presented and interpreted through visual media, using tables and graphs.

3.1 Tables and Figures

That being said, the following section is devoted to describing the exact results extracted from the various algorithm's test cases.

3.1.1 Bottom-Up Dynamic Programming

n	Trial 1	Trial 2	Trial 3	Average
100	0.0308	0.0313	0.0332	0.03143333333
1000	0.3733	0.3532	0.3471	0.3578666667
10000	3.6924	4.4920	4.8813	4.3552333333
100000	41.5751	40.9862	41.8182	41.4598333333

Table 1: Time in Seconds s of the Dynamic Programming Bottom-Up Method Runtime for Solving 0/1 Knapsack Problem

Table 1 describes the runtime in seconds s of running the dynamic programming bottom-up approach for solving the 0/1 Knapsack Problem from $n = 100$ to $n = 100000$. Alongside this is the graph generated from the data produced in each each trial is seen as follows. The x-axis represents the number of items from the aforementioned ranges, and the y-axis represents the execution time measured in seconds. Note that the x-axis values are scaled logarithmically to better show the trend in runtime as the number of items considered grow.

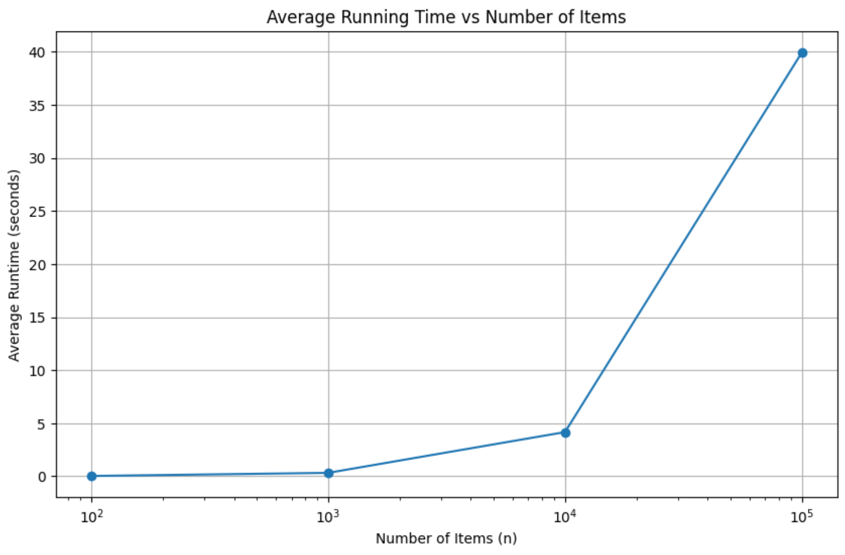

Figure 1: Graph for the Averages of Execution Time in Dynamic Programming Bottom-Up Method for Solving the Knapsack Problem

3.1.2 Top-Up Dynamic Programming

n	Trial 1	Trial 2	Trial 3	Average
100	0.0609	0.0156	0.0176	0.0314
1000	0.9859	0.5718	0.5497	0.7025
10000	14.9485	15.6926	15.5719	15.4043

Table 2: Time in Seconds s of the Dynamic Programming Top-Down Method Runtime for Solving 0/1 Knapsack Problem

n	Extrapolated Average
100000	961.6658

Table 3: Time in Seconds s of Extrapolated Data Runtime from Dynamic Programming Top-Down Method for Solving 0/1 Knapsack Problem

The next two tables, table 2 and 3 describe the runtime in seconds of the three trials among different value ns in solving the 0/1 Knapsack problem through the dynamic programming top-down method, and the extrapolated average runtimes for the value $n = 100000$ respectively.

Following this, figure 2 seen below represents the visualization of the points described within the two tables above. Mimicking the graph for the bottom-up approach, the x-axis describes the number of items n and the y-axis is used to reference the average runtimes in seconds between each trial as well as the extrapolated average runtimes for item $n = 100000$. Note that the identified trend formula used based from Google Sheets' software was '$-0.0346 + 6.47E - 04x + 8.97E - 08^2$'. This was used in order to extrapolate the other data point as denoted in table 3.

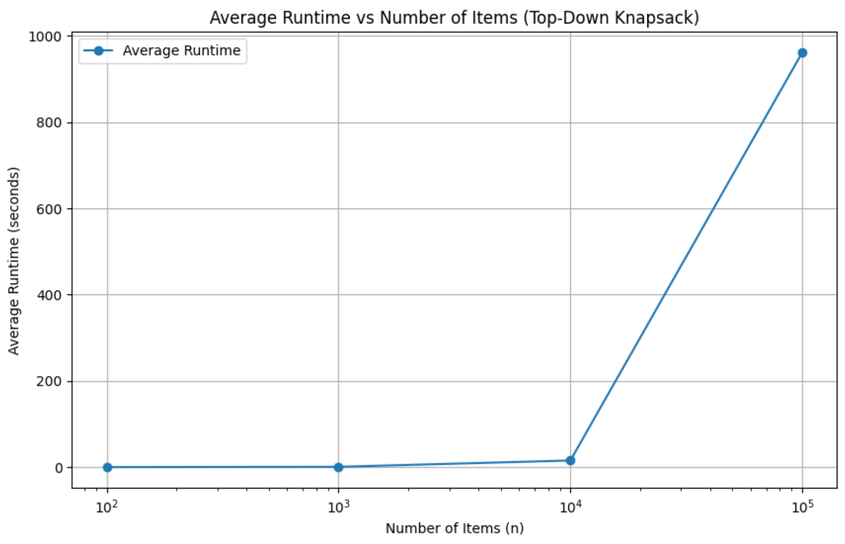

Figure 2: Graph for the Averages of Execution Time in Dynamic Programming Top-Down Method for Solving the Knapsack Problem

Lastly, figure 3 depicts the difference in the two dynamic programming graphs in terms of average runtime and number of items by combining the two graphs seen in 1 and 2.

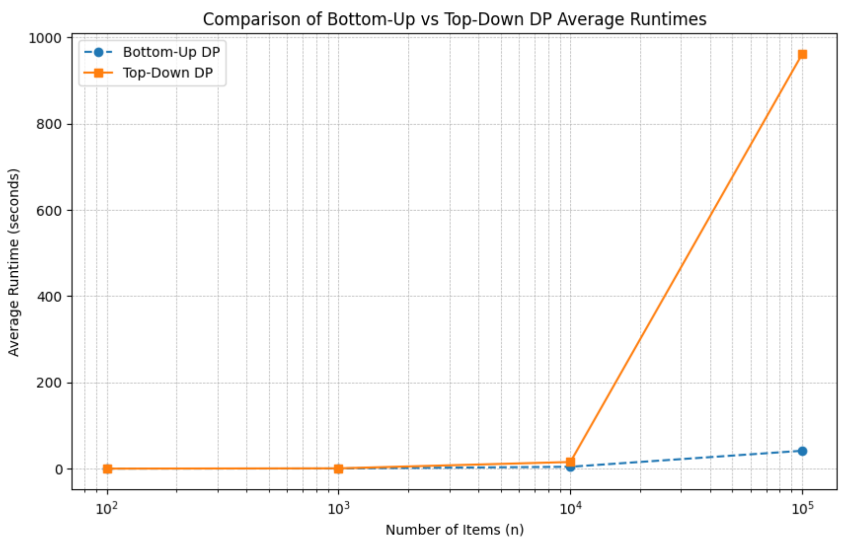

Figure 3: Graph for the Averages of Execution Time in Dynamic Programming Top-Down Method for Solving the Knapsack Problem

3.1.3 Greedy Approach

n	Trial 1	Trial 2	Trial 3	Average
100	0.0000	0.0001	0.0001	0.00007
1000	0.0003	0.0005	0.0003	0.00037
10000	0.0049	0.0036	0.0036	0.00403
100000	0.0552	0.0560	0.0564	0.05587

Table 4: Time in Seconds s of the Greedy Method 1 Runtime for Solving 0/1 Knapsack Problem

n	Trial 1	Trial 2	Trial 3	Average
100	0.0001	0.0002	0.0006	0.00030
1000	0.0005	0.0003	0.0003	0.00037
10000	0.0045	0.0037	0.0040	0.00407
100000	0.0619	0.0630	0.0610	0.06197

Table 5: Time in Seconds s of the Greedy Method 2 Runtime for Solving 0/1 Knapsack Problem

n	Trial 1	Trial 2	Trial 3	Average
100	0.0001	0.0002	0.0001	0.00013
1000	0.0007	0.0007	0.0007	0.00070
10000	0.0082	0.0079	0.0091	0.00840
100000	0.1196	0.1106	0.1238	0.11800

Table 6: Time in Seconds s of the Greedy Method 3 Runtime for Solving 0/1 Knapsack Problem

Tables 4 to 6 describes the recorded runtime in seconds s of running the Greedy Algorithm with its three different variations of how items are to be prioritized and included the Knapsack. In solving the problem, the students opted to use n that are the powers of 10 starting from 10^2 to 10^5. This range of values was chosen in order to provide a sufficient amount of n values for proper data plotting for proper analysis to be made.

The following figures are the graphs taken from the average data produced in the trials of each greedy algorithm. These following graphs illustrate the execution time for solving the 0/1 Knapsack Problem using the three different greedy algorithms as the number of n items increase. The x-axis represents the number of items from the range of $n = 100$ to $n = 100000$ and the y-axis illustrates the average execution time measured in seconds.

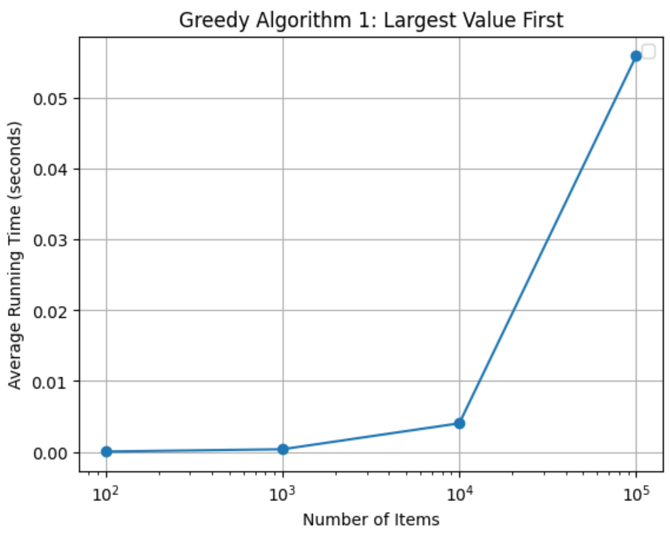

Figure 4: Graph for the Average Execution Time of Greedy Algorithm 1 for Solving the Knapsack Problem

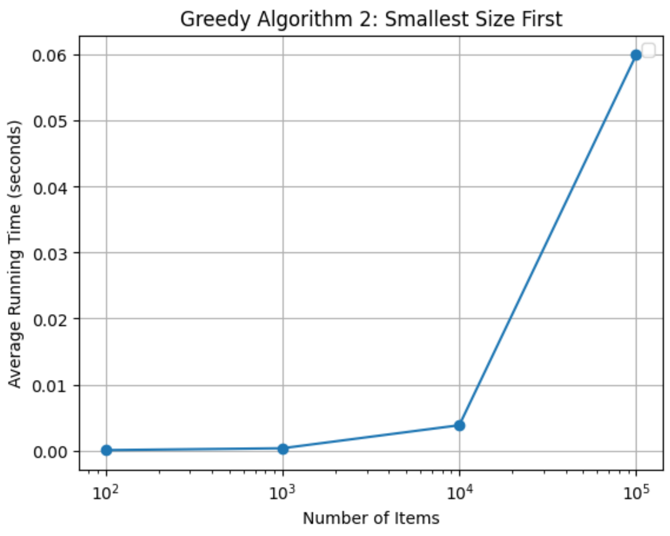

Figure 5: Graph for the Average Execution Time of Greedy Algorithm 2 for Solving the Knapsack Problem

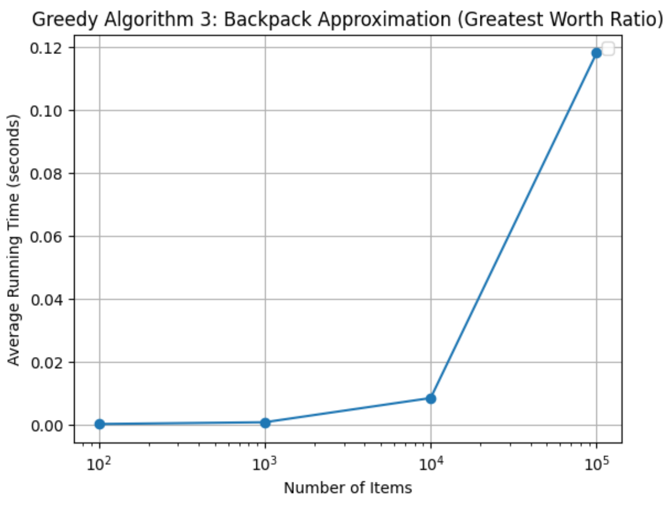

Figure 6: Graph for the Average Execution Time of Greedy Algorithm 3 for Solving the Knapsack Problem

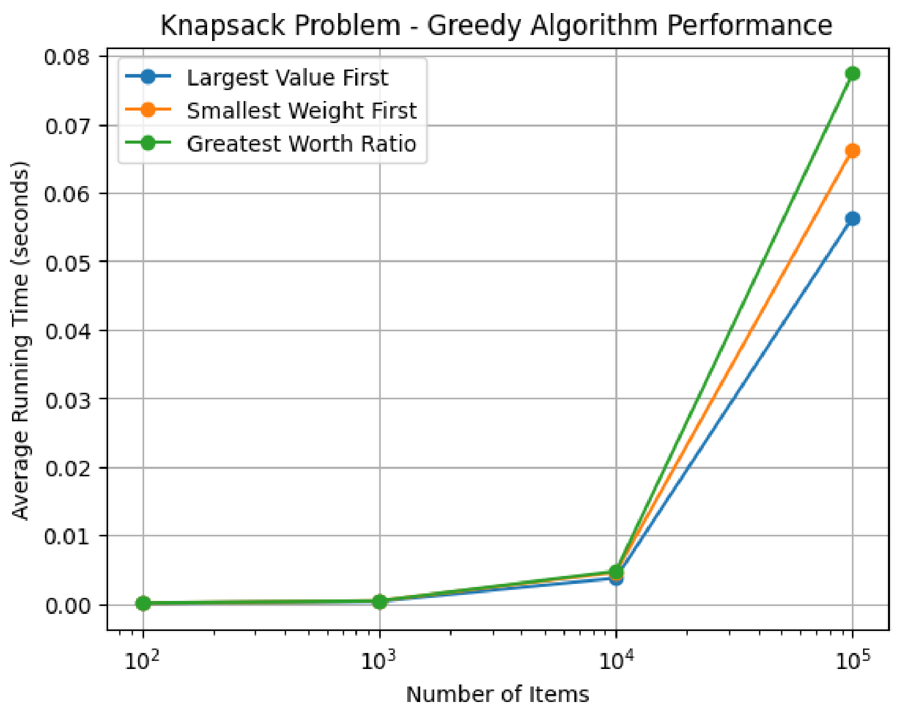

Figure 7: Graph of the Average Running Times of the Different Greedy Algorithms

3.2 Algorithm Analysis

3.2.1 Dynamic Programming

The dynamic programming approaches to solving the 0/1 knapsack problem's efficiency in terms of run-time performance was analyzed through observing the average runtime among three separate trials on different problem sizes. In order to strengthen the empirical analysis of these approaches, the theoretical time complexities of each part of the algorithms (i.e., the knapsack functions, the backtracking capabilities, etc.) were also compared with the practical runtime performances in looking for an optimal solution.

From figure 3, we see that there is a clear difference in the performances of the bottom up dynamic programming approach and the top-down dynamic programming approach to solving the the 0/1 Knapsack problem. In particular, it is observable that in smaller instances of knapsack items (from $n = 100$ to $n = 1000$) the differences between the two algorithm's performances differ only slightly, with both approaches being able to solve the knapsack problem in fractions of a second. The averages in runtime begins to deviate from each other around the $n = 10000$ mark, with the runtime performance of the top-down approach being slightly worse/higher than that of the runtime for the bottom-up approach. That deviation then skyrockets when considering n items past $n = 10000$ in that the top-down approach takes upwards of 900 seconds to return an optimal solution based on its algorithm, while the bottom-down approach only takes around 40 seconds.

Viewing this trend through the numerical tables detailing the exact values, we see that the general trend observable in the graph comparison is further strengthened. From the tables above, we see that the values in runtime for both algorithms vary only slightly from each other, especially in considering n values of 100 items. They start to deviate from each other with greater percentage from n items 1000 and above, with the largest difference in percentages occurring when considering $n = 100000$, where the percentage difference between the two shoots up to 183.47% as it takes the top-down approach an estimated 961.6658 seconds compared to the 41.460 seconds it takes for the bottom-up approach to complete it at that number of items.

Analyzing the differences in runtimes between the two algorithms, we may infer that the top-down dynamic programming approach is relatively better in terms of runtime speed than the bottom-up with smaller item data ns by around 0.1% while the bottom-up approach to solving the knapsack problem generally performs better than the top-down approach when the number of items being considered grows larger, outperforming the top-down approach by around 181%.

This practical observation differs from the theoretical, with both algorithms having a time complexity of $O(nW)$ where n is the amount of items being considered and W is the knapsack's capacity (Levitin, 2012). The divergence from the theoretical complexity compared to its practical counterpart may be attributed to the natures of the top-down and bottom-up algorithms; The bottom-up technique is iterative in nature, systematically filling its array based on previously computed values to obtain the solution to the knapsack problem, while the top-down approach utilizes recursion calls in order to find the optimal solution. The recursive nature of the technique may be one attributing factor to the faster execution times of the bottom-up approach, as the presence of recursion errors that occur when large datasets are considered may lead to slower execution times.

In addition to the runtime differences, the theoretical memory usage of the two approaches also differ from each other in practical settings. That is, the bottom-up approach traditionally allocates a 2D array of size $O(nW)$ to store the results for the item and weight combinations, becoming costly for larger values of n. On the other hand, the top-down approach uses recursive calls with memoization, also requiring $O(nW)$. However, due to these recursive calls, the top-down approach may introduce additional memory consumption, growing with the depth of the recursion. As such the top-down approach practically performed worse due the large amount of items, experiencing stack overflows and recursion errors past $n = 10000$.

Furthermore, the presence of backtracking can be seen in both the top-down and bottom-up approaches in determining the composition of the optimal solution for the knapsack problem. This concept is used by both algorithms to reconstruct optimal solutions after the computations halt. In general, the time complexity of backtracking depends on the specific problem and their constraints, but they usually fall into exponential time. In terms of the top down approach, backtracking occurs in evaluating which items are included based on the memoization table while the bottom-up approach utilizes backtracking through its filled array to determine the item sequence for the knapsack. Although backtracking algorithms typically fall into exponential time complexities for various kinds of problems, the time complexity of the backtracking exhibited by both dynamic programming approaches fall into the linear time $O(n)$, as both approaches check each previously computed item only once when looking back for the optimal solution. Thus, the time complexity of both algorithms fall within their theoretical limit of $O(nW)$ despite deviating from one another in terms of run time.

3.2.2 Greedy Algorithm

The efficiency and effectiveness of the greedy algorithm for solving the 0/1 knapsack problem were analyzed by varying the problem sizes n and observing the runtime performance across three different greedy approaches. These approaches were also compared against the theoretical expectations of their time complexity and their approximation to the optimal solution.

The three variations of the greedy algorithm used in this analysis are as follows:
Greedy Method 1: Prioritizes items with the largest value (v_i).

- Theoretical Basis: This method assumes that selecting items with the highest value will maximize the total value of the knapsack. However, it does not consider the weight of the items, which can lead to suboptimal solutions when high-value items are disproportionately heavy.
- Time Complexity: Sorting items by value has a complexity of $O(n \log n)$, and selecting items sequentially is $O(n)$. Thus, the overall complexity is $O(n \log n)$.

Greedy Method 2: Prioritizes items with the smallest value (v_i).

- Theoretical Basis: This approach is counterintuitive for optimization but was included for comparative purposes. It serves as a baseline to show how prioritizing the smallest values adversely affects the solution quality while still maintaining the same time complexity as the other methods.
- Time Complexity: Similarly, $O(n \log n)$ due to sorting, plus $O(n)$ for selection. Thus, the overall complexity is $O(n \log n)$.

Greedy Method 3: Prioritizes items based on the descending value-to-weight ratio (v_i/w_i).

- Theoretical Basis: This method aligns closely with the optimal fractional knapsack strategy and is expected to approximate the optimal solution better than the other two methods. By selecting items with the highest worth per unit weight, it balances both value and weight considerations.
- Time Complexity: Sorting by the value-to-weight ratio has a complexity of $O(n \log n)$, and selection remains $O(n)$, making the overall complexity $O(n \log n)$.

As seen in the data from tables 4 to 6, visualized through figures 4 to 6, and further reiterated with the graph of the averages among the greedy algorithm performance in figure 7, there is a clear trend in regards to the running time of each of the greedy algorithm implementation. For smaller values of n, such as 100 and 1,000, the runtimes were negligible, averaging less than 0.001 seconds. However, as n is increased to 10,000 and 100,000, the runtimes scaled predictably, with larger problem sizes resulting in longer execution times.

The Greedy Method 1 with the largest value priority, has an average runtime ranging from 0.00007 seconds to 0.05587 seconds, its simplicity and focus on high-value items contribute to its efficiency. The Greedy Method 2 with the smallest value priority has slightly higher runtimes, peaking at 0.06197 seconds, the strategy of prioritizing smaller values likely adds computational overhead without improving solution quality. Lastly, Greedy Method 3 that utilizes the descending worth-to-weight ratio as the priority had the highest average runtime, with an average of 0.11800 seconds which is to be expected due to the additional computation required for sorting items by their value-to-weight ratio.

Similar to that of the Dynamic Programming analysis, these results confirm that while all three greedy methods share the same theoretical time complexity, differences in implementation and computational overhead can affect actual performance. Among the three methods, the Greedy Method 3 provides solutions closest to the optimal because it accounts for both value and weight. This aligns with theoretical expectations that prioritize efficiency in resource allocation. Greedy Method 1 achieves a reasonable trade-off between runtime performance and solution quality. While it does not consider weight, it can still yield acceptable results for many practical cases. While the Greedy Method 2 consistently underperforms in terms of solution quality due to its flawed prioritization strategy, emphasizing its use only as a comparative baseline.

While these algorithms both theoretically and practically outperform traditional brute-force algorithms in obtaining solutions to optimization problems such as the 0/1 Knapsack problem, each algorithm mentioned in this paper still comes with their own set of caveats in obtaining optimal solutions. Aside from the sources of error previously mentioned in machine problem 1, such as factors considering hardware and compilation tools used to obtain the results, some possible sources of error involving the use of dynamic and greedy algorithms may arise from the computational limitations of the algorithms itself. That is, dynamic programming may face memory complications due to the space allocations needed for storing computations in the search for an optimal solution. Aside from this, potential arithmetic errors may arise due to the recursive depth element in one of the dynamic programming approaches.

For greedy algorithm approaches, a possible source of error may be the trade off observed for faster runtime executions. While offering faster runtimes than dynamic programming, suboptimal solutions may be traded off for this result. That is, the solutions obtained for the knapsack problem may be suboptimal, as local heuristic results may overlook the true globally optimal answer to the problem.

4 Conclusion

This paper has explored the efficiency of the dynamic programming of both bottom-up and top-down approaches as well as the three different variations of using the greedy algorithms in solving the 0/1 Knapsack Problem, focusing on the correlation between item count and execution time. The analysis reveals key insights into the computational performance and applicability of these algorithms across varying input sizes.

Dynamic programming approaches exhibit a number of trade offs between the bottom-up and top-down approaches. In comparing the two, these trade offs become more identifiable, reflecting key differences in their practical implementation and performances. In particular, the bottom-up approach is generally more efficient in terms of runtime performance and memory usage in situations with larger datasets, even though it typically uses more memory for storing all the possible states of the problem while the top-down approach is viable and, in some situations, the more desirable approach in small datasets as it theoretically would be more efficient in memory usage and runtime performance.

The empirical evaluation of the algorithms' practical performances demonstrate that although theoretical complexities may provide good insight in understanding and deciding the proper algorithm to use in different situations, practical behavior may still depend on the structure of said algorithms. Thus, we see that the choice between top-down or bottom-up dynamic programming approaches is dependent on factors such as problem size, memory, and recursion constraints.

On the other hand, the greedy algorithms demonstrate a trade-off between computational efficiency and solution optimality. Among the three variations, the Greedy Algorithm 3 or the backpack approximation algorithm, which is based on the value-to-weight ratio, achieves results closest to the optimal solution, making it a reliable choice when computational resources are limited. The Greedy Algorithm 1 or the "largest value first" and Greedy Algorithm 2 or the "smallest weight first" strategies, while computationally efficient and performing with less running time, often fail to account for the optimization of value and weight, leading to substandard results. Notably, all greedy algorithms outperform dynamic programming methods in runtime as input size increases, making them suitable for scenarios where near-optimal solutions suffice.

A comparative analysis of running times highlights that dynamic programming methods scale poorly with larger datasets due to their $O(nW)$ time complexity. In contrast, greedy algorithms, with their $O(n\log(n))$ time complexity, provide significant runtime advantages, especially for datasets with $n > 10,000$. For instance, in the bottom-up dynamic programming approach, while taking mere fractions of a second for $n = 1,000$ it requires substantially more time to compute for $n = 100,000$. Greedy algorithms complete the same task in orders of magnitude using less time, albeit with a compromise in solution quality.

These findings shows the importance of algorithm selection based on the problem's constraints and requirements. For scenarios demanding exact solutions and involving moderate input sizes, the use of dynamic programming is the better approach. In contrast, for large-scale problems or time-sensitive applications, greedy algorithms provide a more practical alternative, particularly when the approximation quality of the backpack algorithm is acceptable and can be implemented.

In summary, this study highlights the versatility and limitations of dynamic programming and greedy algorithms in solving the 0/1 Knapsack problem, both in the theoretical and practical sense. The results contribute to a deeper understanding of algorithmic trade-offs and provide practical guidelines for selecting appropriate algorithm approach based on the problem characteristics.

Further improvements may be done to the structure of some algorithms discussed above, such as the use of a 1D array rather than the 2D array used in bottom-up dynamic programming to reduce storage costs as well as exploring hybrid approaches that combine the strengths of dynamic programming and greedy methods to balance runtime efficiency and solution quality. Aside from this, other more advanced algorithms beyond dynamic programming and greedy algorithms may be considered such as branch and bound techniques and other approximation schemes such as Fully Polynomial-Time Approximation Schemes (FPTAS) may be applied for better runtime functionalities, especially in consideration of larger datasets.

5 References

[1] Anany Levitin. (2012). *Introduction to the design and analysis of algorithms* (3rd ed.). Pearson

[2] CodeVault. (2018, June 27). Recursion explained with examples [Video]. YouTube. https://www.youtube.com/watch?v=rCkeRxVjOEo

[3] DataClarity Corporation. (n.d.). The power of logarithmic scale. DataClarity Corporation. Retrieved December 28, 2024, from https://dataclaritycorp.com/the-power-of-logarithmic-scale/

[4] Digital Ocean. (n.d.). Python Bitwise Operators. https://www.digitalocean.com/community/tutorials/python-bitwise-operators

[5] Evans, A. G. (n.d.). Knapsack [Python code]. GitHub. Retrieved December 28, 2024, from https://github.com/AndrewGEvans95/Knapsack

[6] Gautam, S. (2023, September 4). Top-down vs bottom-up approach in Dynamic Programming. Medium. https://medium.com/enjoy-algorithm/top-down-vs-bottom-up-approach-in-dynamic-programming-53b917bfbe0

[7] GeeksforGeeks. (2024, December 4). 0/1 Knapsack problem. GeeksforGeeks. https://www.geeksforgeeks.org/0/1-knapsack-problem-dp-10/

[8] Google. (n.d.). *Colaboratory – Google*. Research.google.com; Google. Retrieved October 31, 2024, from https://research.google.com/colaboratory/faq.html

[9] Harish. (n.d.). Python program to demonstrate queue implementation using lists [Python code]. GitHub Gist. Retrieved December 28, 2024, from https://gist.github.com/harishv7/1bb7fa7acba8e741fbfbe5075106926b

[10] Jooken, J., Leyman, P., & De Causmaecker, P. (2023). Features for the 0/1 knapsack problem based on inclusionwise maximal solutions. *European Journal of Operational Research.* https://doi.org/10.1016 /j.ejor.2023.04.023

[11] Kumar, A. (2022, July 25). What is XOR in Python? Scaler Topics. https://www.scaler.com/topics/xor-in-python/

[12] Ribera, J. (n.d.). greedy.py [Python code]. GitHub. Retrieved December 28, 2024, from https://github.com/javiribera/greedy-knapsack/blob/master/greedy.py

[13] StudySmarter. (n.d.). Recursion in programming. StudySmarter. Retrieved December 28, 2024, from https://www.studysmarter.co.uk/explanations/computer-science/functional-programming/recursion-programming/

[14] UCD Professional Academy. (2024). *Why Do Data Analysts Use Python?.* https://www.ucd.ie/professionalacademy/resources/why-do-data-analysts-use-python/