CS 5720 Project 3: Knapsack Problem Analysis

Targol Bakhtiarvand

November 18, 2024

Introduction

This report explores the performance of two dynamic programming algorithms for solving the Knapsack problem: the bottom-up approach and the top-down approach with memoization. Performance comparisons are provided under various conditions, including random and low-weight inputs, and an illustration of the pseudopolynomial-time complexity.

1 Knapsack Problem: Bottom-Up and Top-Down Implementations

Bottom-Up Approach

The bottom-up dynamic programming approach constructs a DP table iteratively. Below is the implementation in Python:

Listing 1: Knapsack Bottom-Up Implementation

```python
def knapsack_bottom_up(values, weights, capacity):
    n = len(values)
    dp = [[0] * (capacity + 1) for _ in range(n + 1)]

    # Fill DP table
    for i in range(1, n + 1):
        for w in range(capacity + 1):
            if weights[i - 1] <= w:
                # Max of including or excluding the item
                dp[i][w] = max(dp[i - 1][w], dp[i - 1][w - weights[i - 1]] + values[i - 1])
            else:
                dp[i][w] = dp[i - 1][w]

    return dp[n][capacity]
```

Explanation

- **Inputs**:
  - `values`: List of item values.
  - `weights`: List of item weights.
  - `capacity`: Maximum capacity of the knapsack.
- **DP Table**: A 2D table `dp` is initialized, where `dp[i][w]` holds the maximum value for the first `i` items with capacity `w`.

- **Logic**:
  - If the current item's weight fits within the capacity, decide whether to include it or not.
  - Update `dp[i][w]` with the maximum value of including or excluding the item.
- **Output**: `dp[n][capacity]` holds the optimal solution.

Top-Down Approach with Memoization

The top-down dynamic programming approach uses recursion and memoization to solve subproblems efficiently. Below is the implementation in Python:

Listing 2: Knapsack Top-Down Implementation

```python
def knapsack_top_down(values, weights, capacity):
    n = len(values)
    memo = {}

    def helper(i, w):
        if i == 0 or w == 0:
            return 0

        if (i, w) in memo:
            return memo[(i, w)]

        if weights[i - 1] > w:
            memo[(i, w)] = helper(i - 1, w)
        else:
            memo[(i, w)] = max(
                helper(i - 1, w),
                helper(i - 1, w - weights[i - 1]) + values[i - 1]
            )
        return memo[(i, w)]

    return helper(n, capacity)
```

Explanation

- **Inputs**: Same as the bottom-up approach.
- **Helper Function**:
  - A recursive function `helper(i, w)` computes the maximum value for the first `i` items with capacity `w`.
  - **Base Case**: If no items are left or capacity is zero, return 0.
- **Memoization**:
  - Use a dictionary `memo` to store already computed subproblem results to avoid redundant calculations.
- **Logic**:
  - If the current item's weight exceeds the capacity, exclude the item.
  - Otherwise, decide whether to include or exclude the item and store the result in `memo`.
- **Output**: `helper(n, capacity)` gives the optimal solution.

Verification of Correctness

To verify correctness, test the algorithms with the following example:

```
values = [60, 100, 120]
weights = [10, 20, 30]
capacity = 50

# Bottom-Up
print(knapsack_bottom_up(values, weights, capacity))  # Output: 220

# Top-Down
print(knapsack_top_down(values, weights, capacity))   # Output: 220
```

Both functions should return 220, which is the optimal solution for this problem instance.

2 Deliverable 2: Performance Comparison on Random Inputs

Experiment Design

The experiments were designed as follows:

- **Scenario 1: Varying** n
  - Fix the knapsack capacity at $W = 500$.
  - Vary the number of items (n) from 10 to 100.
- **Scenario 2: Varying** W
  - Fix the number of items at $n = 50$.
  - Vary the knapsack capacity (W) from 100 to 1000.
- For both scenarios, random weights and values were generated, and the runtime of both algorithms was measured using Python's `time` module.

Results

Runtime vs. Number of Items (n)

The runtime for both algorithms was measured with a fixed knapsack capacity ($W = 500$) and varying numbers of items (n). The results are shown in Figure 1.

Observation: The bottom-up approach exhibits consistent scaling with n, while the top-down approach shows fluctuations due to recursive overhead and memoization structure.

Runtime vs. Knapsack Capacity (W)

The runtime for both algorithms was measured with a fixed number of items ($n = 50$) and varying knapsack capacities (W). The results are shown in Figure 2.

Observation: Both algorithms scale linearly with W. The bottom-up approach shows stable behavior, while the top-down approach exhibits variability due to recursive function calls.

Discussion

The results highlight the following:

- Both approaches have pseudopolynomial time complexity $O(nW)$.
- **Bottom-Up Approach:** Stable and predictable as all subproblems are solved iteratively.
- **Top-Down Approach:** Faster for small inputs due to selective computation but degrades for larger inputs due to recursion overhead.

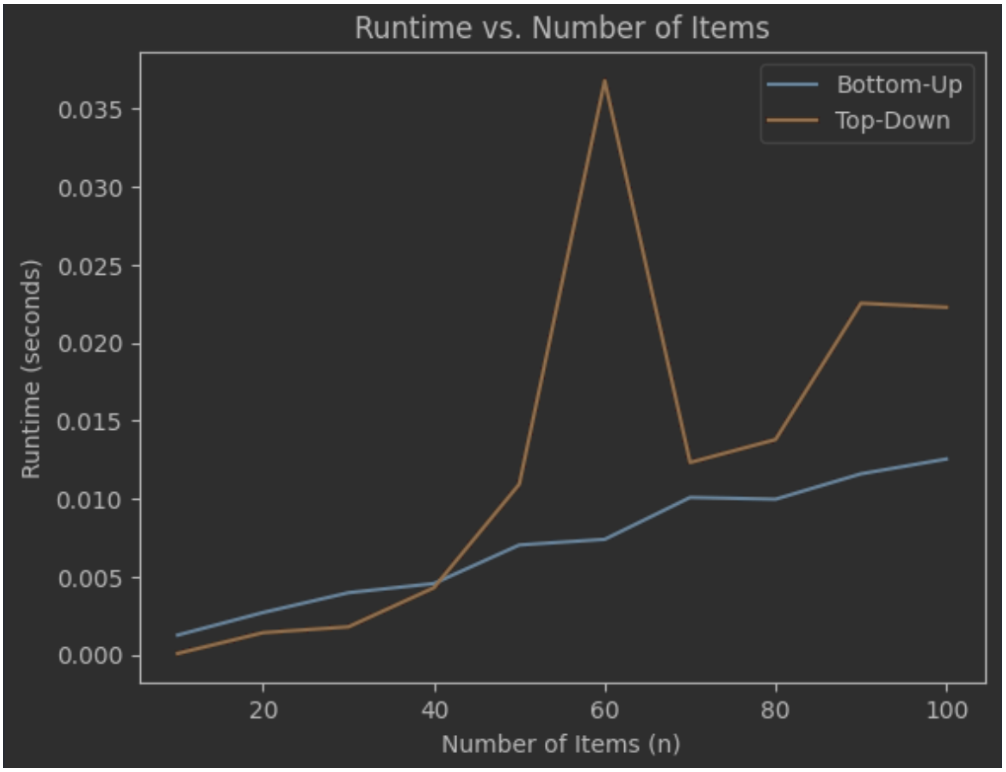

Figure 1: Runtime vs. Number of Items (n) for Bottom-Up and Top-Down Approaches.

3 Deliverable 3: Performance Comparison on Special Inputs

Experiment Design

The experiments were conducted as follows:

- **Scenario 1: Varying** n
  - Fix the knapsack capacity at $W = 500$.
  - Vary the number of items (n) from 10 to 100.
- **Scenario 2: Varying** W
  - Fix the number of items at $n = 50$.
  - Vary the knapsack capacity (W) from 100 to 1000.
- Runtime was measured for both algorithms.

Results

Runtime vs. Number of Items (n)

The runtime for both algorithms with restricted weights and $W = 500$ is shown in Figure 3.

Runtime vs. Knapsack Capacity (W)

The runtime for both algorithms with restricted weights and $n = 50$ is shown in Figure 4.

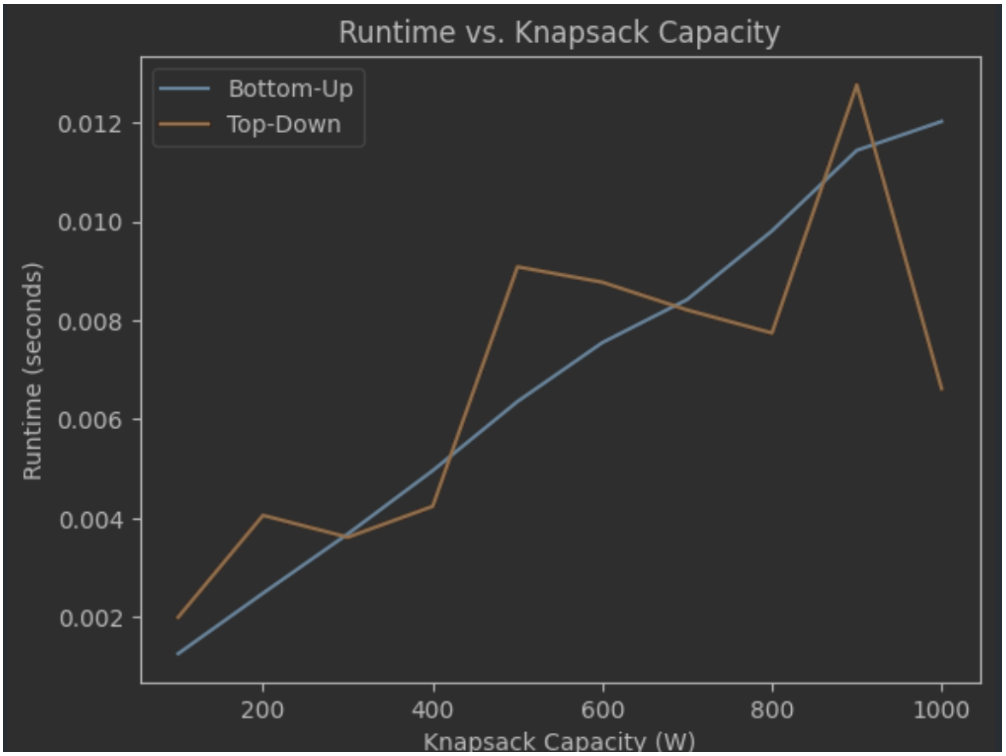

Figure 2: Runtime vs. Knapsack Capacity (W) for Bottom-Up and Top-Down Approaches.

Discussion

The results show:

- **Bottom-Up Approach:** Performs consistently, similar to general inputs, with predictable scaling.
- **Top-Down Approach:** Gains a marginal advantage due to solving fewer subproblems, as the low weights allow many capacities to remain unused.

4 Deliverable 4: Illustration of Pseudopolynomial Time Complexity

Introduction

The dynamic programming algorithms for the Knapsack problem have a time complexity of $O(nW)$, which is pseudopolynomial. This complexity is polynomial in the value of W but exponential in the size of W's binary representation ($\lceil \log_2 W \rceil$). This section demonstrates this characteristic through experimental results.

Experiment Design

The experiment was designed as follows:

- Fix the number of items at $n = 50$.
- Vary the knapsack capacity (W) exponentially: $W = 2^5, 2^6, \ldots, 2^{15}$.
- Measure the runtime of both bottom-up and top-down approaches.
- Plot runtime against the size of W's representation ($\lceil \log_2 W \rceil$).

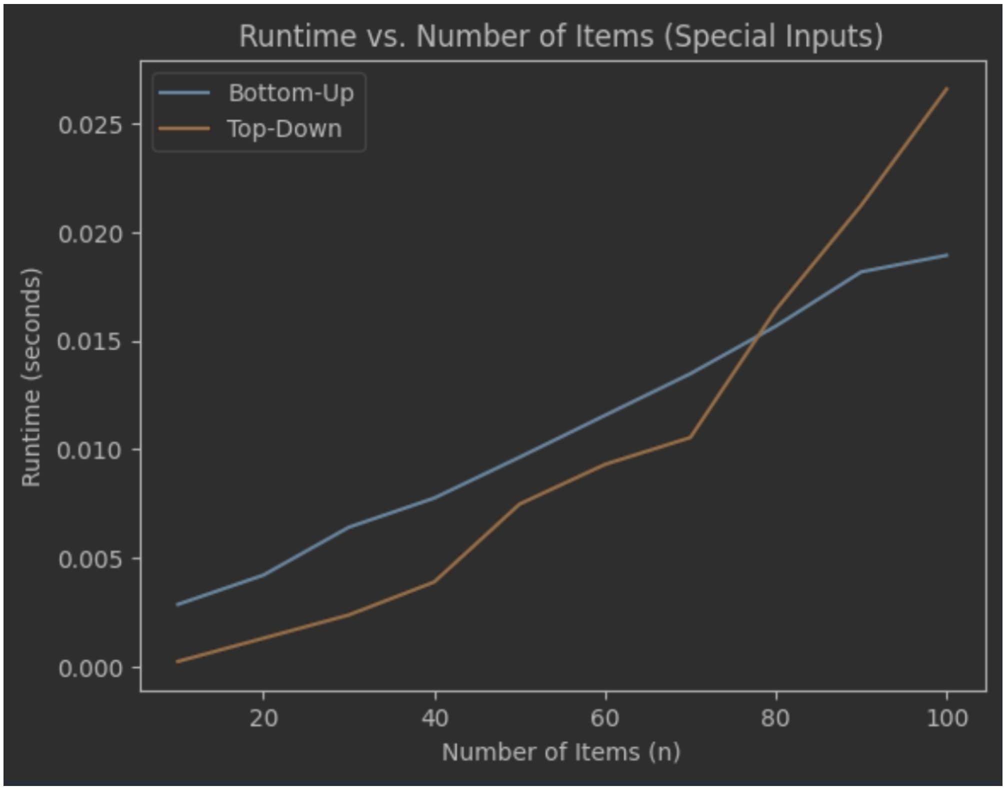

Figure 3: Runtime vs. Number of Items (n) for Bottom-Up and Top-Down Approaches (Special Inputs).

Results

The relationship between runtime and the size of W's binary representation is shown in Figure 5.

Discussion

The results confirm the pseudopolynomial nature of the algorithms:

- The runtime increases linearly with W.
- The runtime grows exponentially with the size of W's representation ($\lceil \log_2 W \rceil$).

Conclusion

The experiment illustrates that the algorithms' time complexity is pseudopolynomial. While polynomial in W, the complexity becomes exponential with respect to the size of W's binary representation.

Conclusion

This project successfully compared two dynamic programming approaches to the Knapsack problem under various input scenarios, highlighting differences in execution time and the implications of pseudopolynomial complexity.

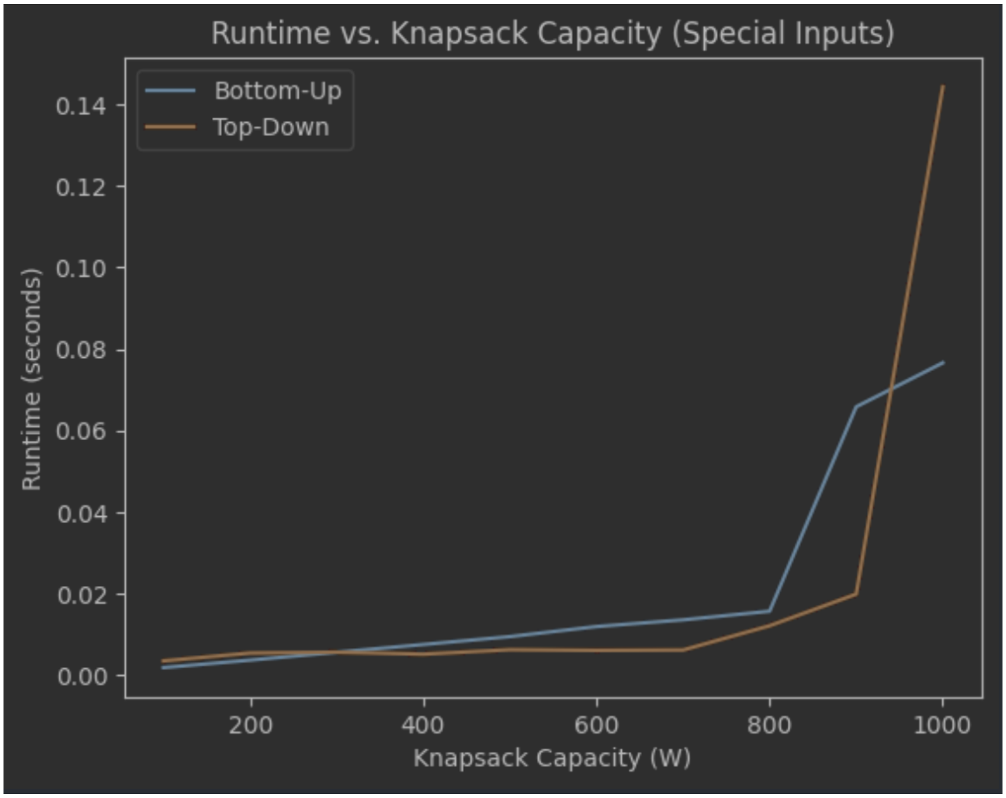

Figure 4: Runtime vs. Knapsack Capacity (W) for Bottom-Up and Top-Down Approaches (Special Inputs).

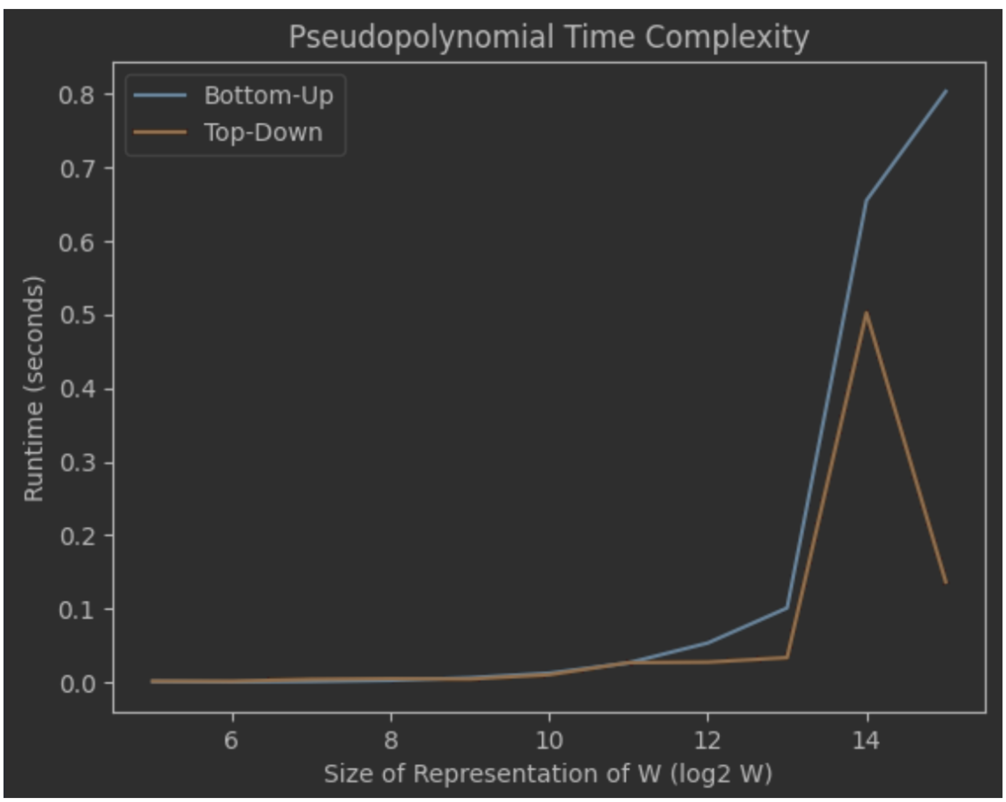

Figure 5: Runtime vs. Size of Representation of W ($\log_2 W$) for Bottom-Up and Top-Down Approaches.