Lab 6. Dynamic Programming

0/1 Knapsack Problem

In many real-world applications, we often need to select items to maximize value within a given limit. The 0/1 Knapsack Problem is a classical problem.
Given a set of items

$$X = \{x_1, x_2, \ldots, x_n\} \tag{1}$$

and a knapsack with capacity W. Each item $x_i \in X$ has an associated weight w_i and value v_i, where $1 \leq i \leq n$.

The goal is to find a subset $Y \subseteq X$ such that the total weight doesn't exceed the capacity.

$$\sum_{x_i \in Y} w_i \leq W \tag{2}$$

The total value is maximized:

$$\sum_{x_i \in Y} v_i \tag{3}$$

Here, each item can be selected at onced.

The 0/1 knapsack recursive transition function:

- $dp[i][w]$: The maximum total value achievable by selecting from the first i items with a knapsack capacity of w.
- w: The current capacity of the knapsack.
- w_i: The weight of the i-th item.
- v_i: The value of the i-th item.

$$dp[i][w] = \begin{cases} 0, & \text{if } i = 0 \text{ or } w = 0 \\ dp[i-1][w], & \text{if } w_i > w \\ max(dp[i-1][w], dp[i-1][w-w_i] + v_i), & \text{if } w_i \leq w \end{cases} \tag{4}$$

The 0/1 Knapsack Algorithm:

```
// n: number of items
// W: capacity of the knapsack
// weights[i]: weight of the i-th item
// values[i]: value of the i-th item
int knapsack(int n, int W, int weights[], int values[]) {
    int dp[n + 1][W + 1];

    // Initialize DP table
    for (int i = 0; i <= n; i++) {
        for (int w = 0; w <= W; w++) {
```

```
            if (i == 0 || w == 0) {
                dp[i][w] = 0;  // Base case: no items or zero capacity
            } else if (weights[i - 1] <= w) {
                // Case 1: take the item
                // Case 2: don't take 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];  // Item doesn't fit
            }
        }
    }

    return dp[n][W];  // Maximum value with n items and capacity W
}
```

Question 1. Analyze the time and space complexity of the above 0/1 Knapsack Algorithm.

Question 2. Please explain why the above 0/1 Knapsack Algorithm is a pseudo polynomial time algorithm, not a polynomial time algorithm.

Question 3. The 0/1 Knapsack Algorithm typically uses an $n \times W$ table to store subproblem solutions. We can improve the space complexity by reducing the size of the table. Please refer to the above 0/1 Knapsack Algorithm and modify ito achieve a space complexity of $O(W)$.

Unbounded Knapsack Problem

The unbounded knapsack problem (also known as complete knapsack problem) is similar to the 0/1 knapsack problem. Given a set of items and the capacity of knapsack, the goal is to maximize the total value of the items you can put into the knapsack. The only difference is whether we restrict the number of copies of each item. In the 0/1 knapsack problem, we can only decide whether to take an item or not. In the unbounded knapsack problem, we don't restrict the number of copies of each item. In other words, you can take each item any non-negative number of items.

You can design the transition function similarly to the 0/1 knapsack problem. Given a knapsack capacity W and an item (v_i, w_i), it is theoretically possible to take up to $\lceil W/w_i \rceil$ copies, depending on the remaining capacity.

Question 4. We can solve the above problem using dynamic programming. What is the recursive transition function? Refer to the 0/1 knapsack recursive transition function and modify it accordingly.

Question 5. Please modify the 0/1 Knapsack Algorithm into an Unbounded Knapsack Algorithm with space complexity $O(W)$ and design an algorithm to reconstruct the combination of items with $O(W)$.

Bounded Knapsack Problem

The bounded knapsack problem is a generalization of the 0/1 knapsack problem, **where each item can be selected multiple times but only up to a fixed limit.**
Given a set of items

$$X = \{x_1, x_2, \ldots, x_n\} \tag{5}$$

and a knapsack with capacity W, each item $x_i \in X$, a weight w_i, a value v_i, a quantity limit c_i, which specifies the maximum number of copies of item x_i that can be selected, where $1 \leq i \leq n$.

The goal is to determine how many copies of item to take, such that the total weight doesn't exceed the capacity:

$$\sum_{i=1}^{n} k_i \times w_i \leq W, 0 \leq k_i \leq c_i \tag{6}$$

and the total value is maximized:

$$\sum_{i=1}^{n} k_i \times v_i \tag{7}$$

Here, k_i is the number of copies of item x_i selected (non-negative integers).

The Bounded Knapsack Algorithm:

```
// n: number of items
// W: capacity of the knapsack
// weights[i]: weight of the i-th item
// values[i]: value of the i-th item
// counts[i]: maximum number of times that item i can be selected
int boundedKnapsack(int n, int W, int weights[], int values[], int counts[]) {
    int dp[n + 1][W + 1];

    for (int i = 1; i <= n; ++i) {
        int w = weights[i - 1];
        int v = values[i - 1];
        int c = counts[i - 1];
        for (int j = 0; j <= W; ++j) {

            dp[i][j] = dp[i - 1][j]; // Case 1: don't take item i

            // Case 2: try take k copies of item i (1 ≤ k ≤ c), if it fits
            for (int k = 1; k <= c; ++k) {
                if (j >= k * w) {
                    dp[i][j] = max(dp[i][j], dp[i - 1][j - k * w] + k * v);
                } else {
                    break;
                }
            }
        }
    }

    return dp[n][W];
```

```
}
```

The time complexity of the Bounded Knapsack Algorithm is $O(n \cdot W \cdot c_{max})$, where n is the number of items, W is the capacity of the knapsack, and c_{max} denotes the maximum copy limit among all items.

Question 6. Please refer to the above Bounded Knapsack Algorithm and modify it using binary optimization to achieve a time complexity of $O(n \cdot W \cdot \log c_{max})$, where c_{max} denotes the maximum copy limit among all items.