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

The given 0/1 Knapsack algorithm uses a dynamic programming (DP) approach with a 2D DP table to store intermediate results.

Time Complexity:

The algorithm consists of two nested loops:

```
for (int i = 0; i <= n; i++) {
    for (int w = 0; w <= W; w++) {
        // DP transition logic
    }
}
```

- The outer loop runs `n + 1` times (from `i = 0` to `i = n`).
- The inner loop runs `W + 1` times (from `w = 0` to `w = W`).

Thus, the total number of iterations is: **O(n * W)**

Each iteration does constant time work (max, addition, comparison), so the total time complexity is:

> **Time Complexity = O(n * W)**

Space Complexity:

The algorithm defines a 2D array `dp[n + 1][W + 1]`, which stores the maximum value for each subproblem defined by the first `i` items and weight limit `w`.

Thus, the space required is: **O(n * W)**

> **Space Complexity = O(n * W)**

Summary:

Metric	Complexity
Time Complexity	O(n * W)
Space Complexity	O(n * W)

Question 2: Why is the 0/1 Knapsack Algorithm a pseudo-polynomial time algorithm, not a polynomial time algorithm?

The time complexity of the 0/1 Knapsack algorithm is:

> **O(n * W)**
> Where:

- `n` is the number of items.
- `W` is the capacity of the knapsack (a numeric value, not the size of the input).

Key Insight:

Although `O(n * W)` appears to be polynomial, it depends on the **numeric value** `W`, not the **number of bits** required to represent `W`.

Example:

- Suppose `W = 1,000,000`, then `log(W) ≈ 20` (i.e., the input size of `W` is about 20 bits).
- The algorithm may still run up to `n * 1,000,000` iterations, which is **exponential in the size of the input**.

Definition:

An algorithm is said to run in **pseudo-polynomial time** if:

> Its running time is polynomial in the numeric value of the input, but **not** in the length of the input (i.e., the number of bits required to encode it).

Conclusion:

The 0/1 Knapsack algorithm is **not a true polynomial-time algorithm** because it scales with the **magnitude** of the capacity `W`, not its binary input size.

> **Therefore, it is considered a pseudo-polynomial time algorithm.**

Question 3: Modify the 0/1 Knapsack Algorithm to achieve a space complexity of O(W)

The original 0/1 Knapsack Algorithm uses a 2D DP array `dp[n + 1][W + 1]`, which results in a space complexity of **O(n * W)**.

However, we only need the values from the previous row at each step, so we can optimize the space by using a 1D array `dp[W + 1]` and update it **in reverse order** to prevent overwriting needed values.

Optimized 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[W + 1] = {0};  // Initialize with zeros

    for (int i = 0; i < n; i++) {
        for (int w = W; w >= weights[i]; w--) {
            dp[w] = max(dp[w], dp[w - weights[i]] + values[i]);
        }
    }

    return dp[W];
}
```

Why reverse order in the inner loop?

We use `w` from `W` to `weights[i]` (in reverse) to ensure that the value `dp[w - weights[i]]` used for computing `dp[w]` is from the **previous iteration of** `i`, not overwritten in the current iteration. This preserves the 0/1 nature (each item can only be used once).

Time and Space Complexity:

- **Time:** O(n * W)
- **Space:** O(W) ← improved from O(n * W)

Summary:

By using a single 1D array and updating it in reverse, we reduce the space usage without changing the correctness of the 0/1 Knapsack algorithm.

Question 4: What is the recursive transition function for the Unbounded Knapsack Problem?

The **Unbounded Knapsack Problem** allows taking **unlimited copies** of each item. Compared to the 0/1 Knapsack, where each item can be taken at most once, the recursive transition function must reflect the possibility of reusing items.

Notation:

- Let `dp[i][w]` be the maximum total value using the first `i` items to achieve capacity `w`.
- `w_i` is the weight of the i-th item
- `v_i` is the value of the i-th item

Recursive Transition Function:

For unbounded knapsack, we can take item `i` multiple times, so:

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

Key Difference from 0/1 Knapsack:

- In 0/1 Knapsack: `dp[i][w - w_i] + v_i` uses `dp[i - 1][...]` — use each item once.
- In **Unbounded Knapsack**: `dp[i][w - w_i] + v_i` uses `dp[i][...]` — allow multiple uses of the same item.

Explanation:

- `dp[i - 1][w]` → Skip the i-th item.
- `dp[i][w - w_i] + v_i` → Take the i-th item (and possibly take it again).

This captures the essence of the unbounded problem, allowing repeated item selection.

Question 5: 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).

To solve the **Unbounded Knapsack Problem** using **O(W)** space, we can use a 1D DP array. Since each item can be chosen multiple times, we update the array in **increasing order of weight**.

Optimized Unbounded Knapsack Algorithm (O(W) space):

```
int unboundedKnapsack(int n, int W, int weights[], int values[]) {
    vector<int> dp(W + 1, 0);

    for (int i = 0; i < n; i++) {
        for (int w = weights[i]; w <= W; w++) {
            dp[w] = max(dp[w], dp[w - weights[i]] + values[i]);
        }
    }

    return dp[W];
}
```

Why forward iteration (w++)?

- We allow multiple usage of the same item, so each updated `dp[w]` can use the current value of `dp[w - weights[i]]` including the current item.

Item Reconstruction with O(W):

To track the items used, we maintain a parallel `choice` array to store which item was last used to achieve each `dp[w]` value.

```
vector<int> reconstructItems(int n, int W, int weights[], int values[]) {
    vector<int> dp(W + 1, 0);
    vector<int> choice(W + 1, -1);

    for (int i = 0; i < n; i++) {
        for (int w = weights[i]; w <= W; w++) {
            int newValue = dp[w - weights[i]] + values[i];
            if (newValue > dp[w]) {
                dp[w] = newValue;
                choice[w] = i; // Store item index
            }
        }
    }

    // Backtrack items
    vector<int> resultCount(n, 0);
    int w = W;
    while (w > 0 && choice[w] != -1) {
        int item = choice[w];
        resultCount[item]++;
        w -= weights[item];
    }

    return resultCount;
}
```

Summary:

- The unbounded knapsack can be solved using O(W) space.
- An auxiliary array allows us to reconstruct which items were used.
- The reconstruction process also runs in O(W) time.

Question 6: Modify the Bounded Knapsack Algorithm using binary optimization to achieve a time complexity of O(n * W * log c_max)

The original Bounded Knapsack algorithm handles each item with a copy limit `c_i` using nested loops over all possible copy counts, resulting in time complexity **O(n * W * c_max)**.

We can improve this by using **binary optimization** (also known as binary decomposition), which breaks down `c_i` copies into a sum of powers of 2. This reduces the number of virtual items per item to **$\log_2$(c_i)**.

Idea:

Any integer `c_i` can be expressed as a sum of powers of 2:

Example:

```
13 = 1 + 2 + 4 + 6
```

We treat each power segment as a separate "virtual item" with:

- `weight = weight[i] * amount`
- `value  = value[i]  * amount`

Optimized Algorithm (C++):

```
int boundedKnapsack(int n, int W, int weights[], int values[], int counts[]) {
    vector<int> dp(W + 1, 0);

    for (int i = 0; i < n; ++i) {
        int count = counts[i];
        int weight = weights[i];
        int value = values[i];

        for (int k = 1; count > 0; k <<= 1) {
            int actual = min(k, count);
            int w = actual * weight;
            int v = actual * value;

            for (int j = W; j >= w; --j) {
                dp[j] = max(dp[j], dp[j - w] + v);
            }
            count -= actual;
        }
    }

    return dp[W];
}
```

Time Complexity:

- Each item produces up to **$\log_2(c_i)$** virtual items.
- Each virtual item processed in `O(W)` time.
- So total complexity: **O(n * W * log c_max)**

Summary:

By breaking copy limits into binary chunks, we reduce the number of iterations per item from `c_i` to `log₂(c_i)`, significantly improving performance while maintaining correctness.