Day 20:

Assignment 1: Knapsack Problem

Write a function int Knapsack(int W, int[] weights, int[] values) in Java that determines the maximum value of items that can fit into a knapsack with a capacity W. The function should handle up to 100 items. Find the optimal way to fill the knapsack with the given items to achieve the maximum total value. You must consider that you cannot break items, but have to include them whole.

A)

I will provide you with a solution for the 0/1 Knapsack problem in Java. This problem can be efficiently solved using dynamic programming. Here's the function that determines the maximum value of items that can fit into a knapsack with a capacity \(W \):

java

```
public class Knapsack {

    public static int knapsack(int W, int[] weights, int[] values) {
        int n = weights.length;
        int[][] dp = new int[n + 1][W + 1];

        // Build table dp[][] in bottom up manner
        for (int i = 0; i <= n; i++) {
            for (int w = 0; w <= W; w++) {
                if (i == 0 || w == 0)
                    dp[i][w] = 0;
                else if (weights[i - 1] <= w)
                    dp[i][w] = Math.max(values[i - 1] + dp[i - 1][w - weights[i - 1]], dp[i - 1][w]);
                else
                    dp[i][w] = dp[i - 1][w];
            }
        }

        return dp[n][W];
    }

    public static void main(String[] args) {
        int W = 50; // Example capacity
        int[] weights = {10, 20, 30}; // Example weights
        int[] values = {60, 100, 120}; // Example values

        System.out.println("Maximum value in Knapsack = " + knapsack(W, weights, values));
    }
}
```

Explanation:

1. Initialization:

- dp[i][w] will hold the maximum value that can be attained with weight w using the first i items.

- Initialize dp[0][w] = 0 for all w because with zero items, the maximum value is 0.

- Initialize dp[i][0] = 0 for all i because with zero capacity, the maximum value is 0.

2. Filling the DP Table:

- Iterate over each item (from 1 to n).

- For each item, iterate over each possible weight (from 1 to W).

- If the weight of the current item weights[i - 1] is less than or equal to the current weight w, then we have two choices:

- Include the current item and add its value to the result of the remaining capacity w - weights[i - 1].

- Exclude the current item and take the result without this item.

- Take the maximum of these two choices.

- If the weight of the current item is greater than w, we cannot include this item, so the result is the same as without this item.

3. Result:

- The result will be in dp[n][W] which represents the maximum value with n items and capacity W.

This code should handle up to 100 items as required. The complexity is \(O(n \times W)\), which is efficient for the given constraints.