.vscode\Practical4.java

```java
// 0-1 Knapsack Problem Using Dynamic Programming

public class KnapsackDP {

    // Function to solve 0-1 Knapsack problem using Dynamic Programming
    public static int knapSack(int capacity, int weights[], int values[], int n) {
        int[][] dp = new int[n + 1][capacity + 1];

        // Build the table dp[][] in bottom-up manner
        for (int i = 0; i <= n; i++) {
            for (int w = 0; w <= capacity; w++) {
                if (i == 0 || w == 0) {
                    dp[i][w] = 0; // Base case
                } 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][capacity]; // Maximum value in the knapsack
    }

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

        int maxValue = knapSack(capacity, weights, values, n);
        System.out.printf("Maximum value in the knapsack = %d\n", maxValue);
    }
}



// Output - Maximum value in the knapsack = 220


// 2. 0-1 Knapsack Problem Using Branch and Bound

import java.util.Arrays;

class Item {
    int value, weight;

```

```java
    public Item(int value, int weight) {
        this.value = value;
        this.weight = weight;
    }
}

class Node {
    int level; // Level of the node in the decision tree
    int profit; // Profit of the node
    int bound; // Upper bound of the profit
    int weight; // Weight of the node
}

public class KnapsackBranchBound {

    // Function to calculate the upper bound on profit
    public static int bound(Node node, int n, int capacity, Item[] items) {
        if (node.weight >= capacity) {
            return 0;
        }
        int profitBound = node.profit;
        int j = node.level + 1;
        int totalWeight = node.weight;

        while (j < n && totalWeight + items[j].weight <= capacity) {
            totalWeight += items[j].weight;
            profitBound += items[j].value;
            j++;
        }

        if (j < n) {
            profitBound += (capacity - totalWeight) * items[j].value / items[j].weight;
        }
        return profitBound;
    }

    // Function to solve 0-1 Knapsack problem using Branch and Bound
    public static int knapSack(int capacity, Item[] items, int n) {
        Arrays.sort(items, (a, b) -> (b.value * 100 / b.weight) - (a.value * 100 / a.weight));
        Node root = new Node();
        root.level = -1;
        root.profit = 0;
        root.weight = 0;
        root.bound = bound(root, n, capacity, items);

        int maxProfit = 0;
        java.util.LinkedList<Node> queue = new java.util.LinkedList<>();
        queue.add(root);

        while (!queue.isEmpty()) {
```

```java
            Node node = queue.poll();

            if (node.bound > maxProfit) {
                // Explore the left child (including the item)
                Node left = new Node();
                left.level = node.level + 1;
                left.weight = node.weight + items[left.level].weight;
                left.profit = node.profit + items[left.level].value;
                left.bound = bound(left, n, capacity, items);
                if (left.weight <= capacity && left.profit > maxProfit) {
                    maxProfit = left.profit;
                }
                if (left.bound > maxProfit) {
                    queue.add(left);
                }

                // Explore the right child (excluding the item)
                Node right = new Node();
                right.level = node.level + 1;
                right.weight = node.weight;
                right.profit = node.profit;
                right.bound = bound(right, n, capacity, items);
                if (right.bound > maxProfit) {
                    queue.add(right);
                }
            }
        }
        return maxProfit;
    }

    public static void main(String[] args) {
        Item[] items = { new Item(60, 10), new Item(100, 20), new Item(120, 30) };
        int capacity = 50;
        int n = items.length;

        int maxValue = knapSack(capacity, items, n);
        System.out.printf("Maximum value in the knapsack = %d\n", maxValue);
    }
}


//Output-Maximum value in the knapsack = 220
```