Bharatiya Vidya Bhavan's
SARDAR PATEL INSTITUTE OF TECHNOLOGY
(Autonomous Institute Affiliated to University of Mumbai)
Munshi Nagar, Andheri (W), Mumbai – 400 058.
Department of Computer Engineering

Experiment	8
Aim	To understand and implement branch and bound Approach
Objective	1) Write Pseudocode for given problems and understanding the implementation of branch and bound approach 2) Implementing Branch and bound algorithm for 15-puzzle problem. 3) Calculating time complexity of the given problems
Name	Anurag Nair
UCID	2022600035
Class	CSE-AIML
Batch	C
Date of Submission	25-03-24

Algorithm and Explanation of the technique used	1. Input: - Read the capacity (cap) of the knapsack. - Read the number of items (itemCount). - Read the weight and value for each item. 2. Initialize Comparator: - Define a comparator based on the ratio of value to weight (comparator). 3. Calculate Bound: - Define a method bound(Node u, int itemCount, int cap, Item[] items) to calculate the potential profit bound for a given node. - If the current weight (cw) of the node is greater than or equal to the capacity (cap), return 0. - Initialize potential profit (pb) with node's profit (p). - Iterate through the remaining items to check if they can be added to the knapsack: - If the total weight (tw) including the current item's weight exceeds the capacity, break the loop. - Add the current item's weight and value to tw and pb respectively. - If there are still items left, calculate the profit by adding a fraction of the next item's value based on remaining capacity. - Return the potential profit (pb). 4. Knapsack Algorithm: - Sort the items array using the comparator. - Initialize a priority queue (pq) to store nodes, prioritized by bound.

	- Create an initial node (cur) with level -1, profit 0, and current weight 0. - Offer the initial node to the priority queue. - Initialize maximum profit (mp) to 0. - While the priority queue is not empty: - Poll the current node (cur) from the priority queue. - Create two new nodes (nextNode) to represent: - Including the current item (if possible). - Excluding the current item. - Update the weight and profit of nextNode based on the chosen action. - If nextNode's weight is within the capacity and its profit is greater than mp, update mp. - Calculate bound for nextNode and check if it's greater than mp. - Offer nextNode to the priority queue if its bound is greater than mp. - Return the maximum profit (mp).
Program(Code)	

```
import java.util.*;

class Item {
    float w;
    int v;

    Item(float w, int v) {
        this.w = w;
        this.v = v;
    }
}

class Node {
    int l, p, b;
    float cw;

    Node(int l, int p, float cw) {
        this.l = l;
        this.p = p;
        this.cw = cw;
    }
}

public class Main {
    static Comparator<Item> comparator = (a, b) -> {
        double r1 = (double) a.v / a.w;
        double r2 = (double) b.v / b.w;
        return Double.compare(r2, r1);
    };

    static int bound(Node u, int itemCount, int cap, Item[]
items) {
        if (u.cw >= cap)
            return 0;

        int pb = u.p;
        int j = u.l + 1;
        float tw = u.cw;
```

```
        while (j < itemCount && tw + items[j].w <= cap) {
            tw += items[j].w;
            pb += items[j].v;
            j++;
        }

        if (j < itemCount)
            pb += (int) ((cap - tw) * items[j].v /
items[j].w);

        return pb;
    }

    static int knapsack(int cap, Item[] items, int
itemCount) {
        Arrays.sort(items, comparator);
        PriorityQueue<Node> pq =
                new PriorityQueue<>((a, b) ->
Integer.compare(b.b, a.b));
        Node cur, nextNode;

        cur = new Node(-1, 0, 0);
        pq.offer(cur);

        int mp = 0;

        while (!pq.isEmpty()) {
            cur = pq.poll();

            if (cur.l == -1)
                nextNode = new Node(0, 0, 0);
            else if (cur.l == itemCount - 1)
                continue;
            else
                nextNode = new Node(cur.l + 1, cur.p,
cur.cw);

            nextNode.cw += items[nextNode.l].w;
            nextNode.p += items[nextNode.l].v;

            if (nextNode.cw <= cap && nextNode.p > mp)
                mp = nextNode.p;

            nextNode.b = bound(nextNode, itemCount, cap,
items);

            if (nextNode.b > mp)
                pq.offer(nextNode);

            nextNode = new Node(cur.l + 1, cur.p, cur.cw);
            nextNode.b = bound(nextNode, itemCount, cap,
items);

            if (nextNode.b > mp)
                pq.offer(nextNode);
        }

        return mp;
    }
```

```
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);

        System.out.print("Enter the capacity of the
knapsack: ");
        int cap = scanner.nextInt();

        System.out.print("Enter the number of items: ");
        int itemCount = scanner.nextInt();

        Item[] items = new Item[itemCount];
        for (int i = 0; i < itemCount; i++) {
            System.out.println("Enter weight and value for
item " + (i + 1) + ":");
            float weight = scanner.nextFloat();
            int value = scanner.nextInt();
            items[i] = new Item(weight, value);
        }

        int optimalProfit = knapsack(cap, items, itemCount);
        System.out.println("Optimal profit = " +
optimalProfit);
    }
}
```

Output

```
Enter the capacity of the knapsack: 10
Enter the number of items: 5
Enter weight and value for item 1:
2 40
Enter weight and value for item 2:
3.14 50
Enter weight and value for item 3:
1.98 100
Enter weight and value for item 4:
5 95
Enter weight and value for item 5:
3 30
Optimal profit = 235
```

Justification of the complexity calculated

Item Sorting:
- To prioritize the most valuable items by weight, they are sorted.
- Time Complexity: O(n log n) due to the customized sorting.

Exploring Branches in the Algorithm:
- The algorithm delves into two potential paths at each decision point: one with the item included and one without.
- This leads to a combinatorial explosion of 2^n potential configurations, where n is the number of items.

	Evaluating Bounds: - At every step of the exploration, an upper limit on potential profit is assessed to prune less promising paths. - Calculating this bound involves iterating through the items and has a time complexity of O(n). Total Computational Cost: - Sorting: O(n log n) - Exploring Branches: O(2^n) - Evaluating Bounds: O(n) - The cumulative time complexity of the algorithm stands can be written as O(2^n *n).
Conclusion	In this experiment, I learned about the branch and bound method in coding. It helps make decisions by checking different options and picking the best one. The code we looked at skips bad choices to work faster. This method can be used in many ways, like deciding how to use money or resources, picking the best investments, or planning what products to make in a business.