```
/** Fourth algorithm - Branch & Bound */
public void BBAlgorithm(){
    Node solution = null;
    item[] copyItems = new item[totalItem]; // get the copy array of items
    for (int i = 0; i < totalItem; i ++)
        copyItems[i] = itemArray[i];
    Arrays.sort(copyItems); // sort the array of item in order of descending ratio of value/weight

    PriorityQueue<Node> myQ = new PriorityQueue<Node>();
    // Constructing the tree with original root Node
    Node root = new Node(0, 0, upperBound(copyItems, ""));
    int treeHeight = 0;
    myQ.add(root);
    // Keep building tree until tree's height = total number of items, or found a solution
    while (treeHeight < totalItem) {
        solution = myQ.remove(); // pop the first node in the queue which has highest upper bound.
        treeHeight = solution.partialSol.length();
        // check if we get to the leaf of tree, means we found a solution
        if (treeHeight == totalItem)
            break;
        // not found solution yet, keep building left & right child if possible, then add them to the priority queue
        if(makeLeftNode(solution, copyItems, treeHeight))
            myQ.add(solution.left);
        makeRightNode(solution, copyItems, treeHeight);
        myQ.add(solution.right);
    }

    // Trace back solution using the String partialSolution stored in the Node
    int[] resultIndex = new int[totalItem];
    int totalWeightSolution = 0;
    for (int i = 0; i < solution.partialSol.length(); i++)
        if (solution.partialSol.charAt(i) == '1') {
            resultIndex[i] = copyItems[i].index;
            totalWeightSolution += copyItems[i].weight;
        }
    Arrays.sort(resultIndex);
    // Print result
    System.out.println("Using Branch and Bound the best feasible solution found: "
                       + solution.totalV + " " + totalWeightSolution);
    for (int i: resultIndex)
        if (i != 0)
            System.out.print(i + " " );
    System.out.println();
}
/** Helper method to build the left child of a Node, return false if making this child would overfill the sack*/
private boolean makeLeftNode(Node parent, item[] items, int itemIndex) {
    if (parent.totalW + items[itemIndex].weight <= capacity) {
        String partialSol = parent.partialSol + "1";
        parent.left = new Node(parent.totalV + items[itemIndex].value,
                        parent.totalW + items[itemIndex].weight,
                        upperBound(items, partialSol));
```

```
            parent.left.partialSol = partialSol;
            return true;
        }
        return false;
    }

    /** Helper method to build the right child of a Node */
    private void makeRightNode(Node parent, item[] items, int itemIndex) {
        String partialSol = parent.partialSol + "0";
        parent.right = new Node(parent.totalV,
                                parent.totalW, upperBound(items, partialSol));
        parent.right.partialSol = parent.partialSol + "0";
    }

    /** Helper method to calculate upperbound */
    private double upperBound(item[] items, String partialSolution){
        double upperBound = 0;
        int totalWeight = 0;
        int index = 0;
        // Sum up all possible whole-items, according to the partialSolution
        for (index = 0; index < partialSolution.length(); index++)
            if (partialSolution.charAt(index) == '1') {
                upperBound += (double)(items[index].value);
                totalWeight += items[index].weight;
            }
        // Add all whole-items after partialSolution
        while (index < items.length && totalWeight + items[index].weight <= capacity) {
            upperBound += (double)(items[index].value);
            totalWeight += items[index].weight;
            index++;
        }
        // Adding the fractional item,then return it
        if (index < items.length)
            upperBound += (double)((capacity-totalWeight) *
                          items[index].value / items[index].weight);
        return upperBound;
    }

    /* Inner class Node to construct the tree when doing Branch & Bound algorithm */
    public class Node implements Comparable {
        int totalV; int totalW; double upperBound; String partialSol = "";
        Node left; Node right;
        public Node (int value, int weight, double bound) {
            totalV = value; totalW = weight; upperBound = bound;
        }
        // Comparing the upperBound of this Node with other Node for the purpose of using the Priority Queue
        public int compareTo(Object other) {
            return (this.upperBound > ((Node)other).upperBound) ? -1 :
                   (this.upperBound < ((Node)other).upperBound) ? 1 : 0;
        }
    }
```