SARDAR PATEL INSTITUTE OF TECHNOLOGY

Name: Darshit Bhagtani

2021700006

CSE DS D1

Exp. 8: 1/0 Knapsack problem

AIM: To implement 0/1 Knapsack problem using Branch and Bound.

THEORY:

The 0/1 knapsack problem is a classic optimization problem in computer science, which involves choosing a subset of items with maximum total value, subject to a constraint on the total weight of the items.

The name "0/1" knapsack refers to the fact that each item can either be included in the knapsack (assigned a value of 1) or not (assigned a value of 0), hence the binary decision variable.

The 0/1 knapsack problem is NP-hard, which means that there is no known polynomial-time algorithm that can solve all instances of the problem. However, there are various approximation algorithms and heuristic methods that can provide good solutions in reasonable time for many practical instances of the problem.

One common approach to solving the 0/1 knapsack problem is dynamic programming, which involves building a table of subproblems and using this table to solve larger subproblems until the full problem is solved. Another approach is the branch and bound algorithm, which involves generating a search tree of potential solutions and pruning branches that are guaranteed to be suboptimal.

PROGRAM:

```
#include <stdio.h>

#include <stdlib.h>


#define N 100
```

```
// Item struct representing an item
typedef struct {
    int weight;
    int value;
} Item;

// Node struct representing a node in the search tree
typedef struct {
    int level;
    int value;
    int weight;
    int bound;
} Node;

// Global variables
int n, capacity;
Item items[N];
Node queue[N];
int front = 0, rear = 0;
int max_profit = 0;

// Function prototypes
void enqueue(Node node);
Node dequeue();
int bound(Node node);
void knapsack();
```

```
int main() {
    // Read input
    printf("Enter the number of items: ");
    scanf("%d", &n);
    printf("Enter the capacity of the knapsack: ");
    scanf("%d", &capacity);
    printf("Enter the weight and value of each item:\n");
    for (int i = 0; i < n; i++) {
        printf("Item %d: ", i + 1);
        scanf("%d %d", &items[i].weight, &items[i].value);
    }

    // Solve knapsack problem using branch and bound algorithm
    knapsack();

    // Output result
    printf("Maximum profit: %d\n", max_profit);

    return 0;
}

// Enqueue a node into the queue
void enqueue(Node node) {
    queue[rear++] = node;
}

// Dequeue a node from the queue
```

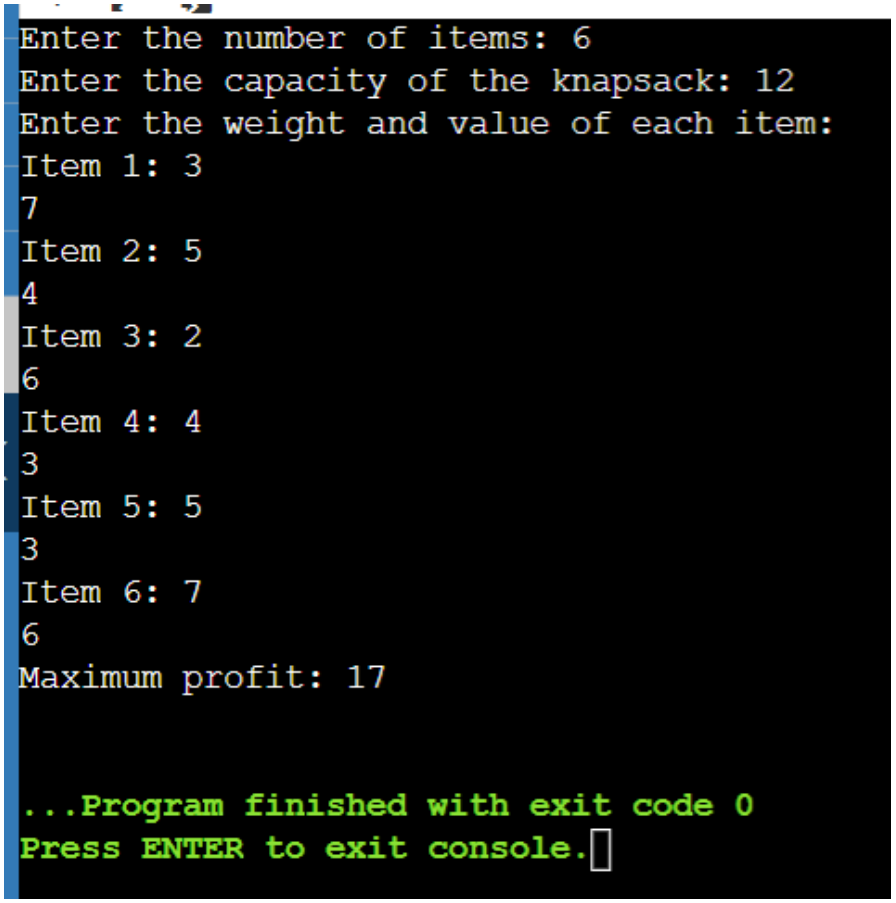
```
Node dequeue() {
    return queue[front++];
}

// Compute the upper bound of a node
int bound(Node node) {
    if (node.weight > capacity) {
        return 0;
    }
    int profit_bound = node.value;
    int j = node.level + 1;
    int total_weight = node.weight;
    while (j < n && total_weight + items[j].weight <= capacity) {
        total_weight += items[j].weight;
        profit_bound += items[j].value;
        j++;
    }
    if (j < n) {
        profit_bound += (capacity - total_weight) * (items[j].value / (double)
items[j].weight);
    }
    return profit_bound;
}

// Solve knapsack problem using branch and bound algorithm
void knapsack() {
    Node root = {0, 0, 0, 0};
    root.bound = bound(root);
```

```
    enqueue(root);
    while (front != rear) {
        Node node = dequeue();
        if (node.bound > max_profit) {
            Node left = {node.level + 1, node.value + items[node.level].value,
node.weight + items[node.level].weight, 0};
            left.bound = bound(left);
            if (left.weight <= capacity && left.bound > max_profit) {
                max_profit = left.value;
            }
            if (left.bound > max_profit) {
                enqueue(left);
            }
            Node right = {node.level + 1, node.value, node.weight, 0};
            right.bound = bound(right);
            if (right.weight <= capacity && right.bound > max_profit) {
                max_profit = right.value;
            }
            if (right.bound > max_profit) {
                enqueue(right);
            }
        }
    }
}
```

OUTPUT:



CONCLUSION:

In this experiment, I implemented 1/0 Knapsack Problem using branch and bound algorithm.