Experiment No. **Branch And Bound**

Date:

Aim: Write a C program to implement the following program using Branch and Bound strategy

a. 0/1 Knapsack

THEORY:

- The term branch-and-bound refers to all state space search methods in which all children of the E-node are generated before any other live node can become the E-node.
- We have already seen two graph search strategies, BFS and D-search, in which the exploration of a new node cannot begin until the node currently being explored is fully explored.
- Both of these generalize to branch and bound strategies. In branch and bound terminology, a BFS-like state space search will be called FIFO (First In First Out) searchas the list of live nodes is a first in first out list (or queue). A D-search like state space search will be called LIFO (Last In First Out) searchas the list of live nodesis a last in first outlist (or stack).
- Branch and bound is an optimization algorithm that efficiently explores the solution space by dividing it into smaller subproblems, known as branches. It uses a bounding technique to eliminate unpromising branches and prioritize the search for the most optimal solution. This iterative process continues until the best solution is found or a stopping criterion is met.
- Branch and bound is widely used in various fields such as operations research, mathematical programming, and computer science for solving optimization problems. It is particularly effective in problems that involve finding the best solution among a large number of possibilities, such as traveling salesman problems, knapsack problems etc.
- Advantages: Efficiently reduces search space and guarantees finding optimal solutions for optimization problems.
- Disadvantages: Performance heavily relies on the quality of bounding and branching strategies chosen.

a. 0/1 Knapsack

Date:

Branch and bound efficiently solves the **0/1 knapsack** problem by dividing the search space into smaller subproblems, using bounding techniques to eliminate unpromising branches and find the optimal solution.

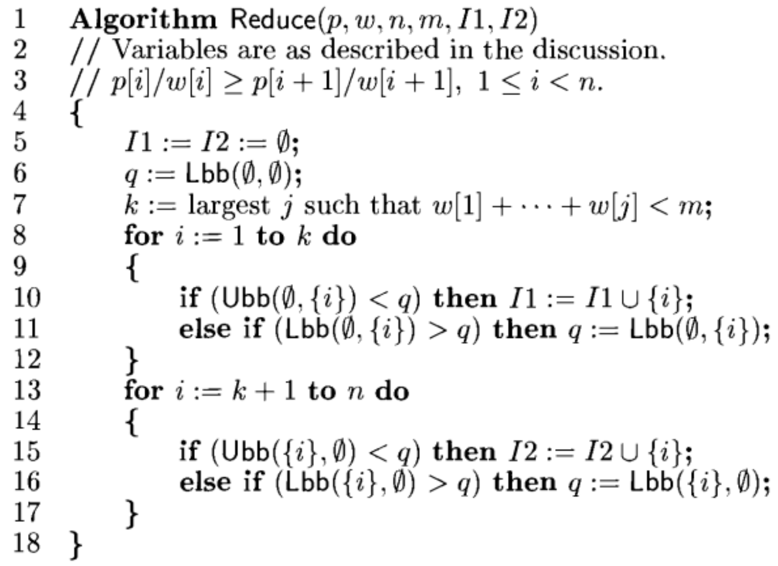
Problem Statement:

Write a c program to implement 0/1 Knapsack Using Branch and Bound

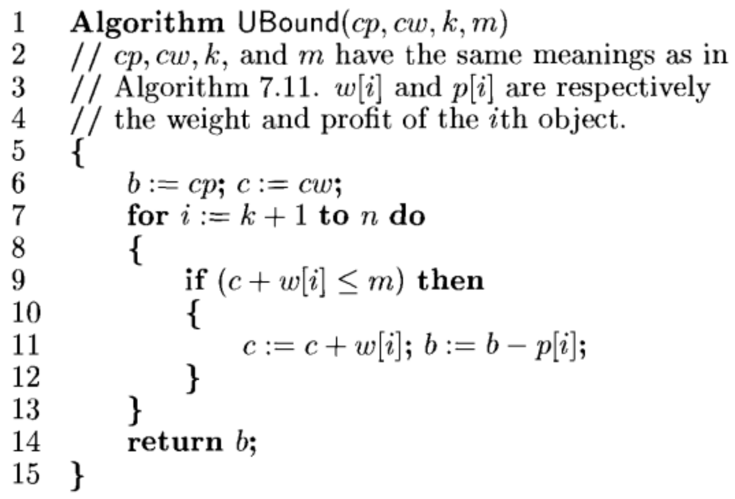
M = 8

Weight	Profit
2	3
3	5
4	6
5	10

Algorithm



```
Algorithm Reduce($p, w, n, m, I1, I2$)
// Variables are as described in the discussion.
// $p[i]/w[i] \geq p[i+1]/w[i+1],\ 1 \leq i < n$.
{
    $I1 := I2 := \emptyset$;
    $q :=$ Lbb($\emptyset, \emptyset$);
    $k :=$ largest $j$ such that $w[1] + \cdots + w[j] < m$;
    for $i := 1$ to $k$ do
    {
        if (Ubb($\emptyset, \{i\}$) $< q$) then $I1 := I1 \cup \{i\}$;
        else if (Lbb($\emptyset, \{i\}$) $> q$) then $q :=$ Lbb($\emptyset, \{i\}$);
    }
    for $i := k + 1$ to $n$ do
    {
        if (Ubb($\{i\}, \emptyset$) $< q$) then $I2 := I2 \cup \{i\}$;
        else if (Lbb($\{i\}, \emptyset$) $> q$) then $q :=$ Lbb($\{i\}, \emptyset$);
    }
}
```



```
Algorithm UBound($cp, cw, k, m$)
// $cp, cw, k$, and $m$ have the same meanings as in
// Algorithm 7.11. $w[i]$ and $p[i]$ are respectively
// the weight and profit of the $i$th object.
{
    $b := cp$; $c := cw$;
    for $i := k + 1$ to $n$ do
    {
        if $(c + w[i] \leq m)$ then
        {
            $c := c + w[i]$; $b := b - p[i]$;
        }
    }
    return $b$;
}
```

Time Complexity & Space Complexity

The Time complexity of the 0/1 knapsack problem using branch and bound is exponential, specifically O(2^n), where n is the number of items. This is because the algorithm explores all possible subsets of items in the search space. However, the actual runtime can be reduced by employing bounding techniques and heuristics.

The space complexity of branch and bound for the 0/1 knapsack problem is also exponential in the worst-case scenario. This is because the algorithm needs to store information about the active nodes in the search tree.

Code

```
#include <stdio.h>
int maxProfit = 0; // Global variable to store the maximum profit
int maxWeight = 0; // Global variable to store the maximum weight

void AlgorithmUBound(int cp, int cw, int k, int m, int w[], int p[], int n) {
int b = cp;
int c = cw;
for (int i = k + 1; i < n; i++) {
if (c + w[i] < m) {
c += w[i];
b -= p[i];
}
}
if (b > maxProfit && c <= m) {
maxProfit = b;
maxWeight = c;
}}

void Knapsack(int cp, int cw, int k, int m, int w[], int p[], int n) {
if (k == n - 1) {
if (cp > maxProfit && cw <= m) {
maxProfit = cp;
maxWeight = cw;
}
return;
}
AlgorithmUBound(cp, cw, k, m, w, p, n);
if (cp + p[k+1] > maxProfit && cw + w[k+1] <= m) {
Knapsack(cp + p[k+1], cw + w[k+1], k + 1, m, w, p, n);
}
Knapsack(cp, cw, k + 1, m, w, p, n);
}
int main() {
int n; // Number of items
printf("Enter the number of items: ");
scanf("%d", &n);
int w[n], p[n]; // Arrays to store weights and profits
printf("Enter the weights of the items: ");
for (int i = 0; i < n; i++) {
scanf("%d", &w[i]);
}

printf("Enter the profits of the items: ");
for (int i = 0; i < n; i++) {
scanf("%d", &p[i]);
}
int m; // Maximum capacity of the knapsack
printf("Enter the maximum capacity of the knapsack: ");
scanf("%d", &m);
Knapsack(0, 0, -1, m, w, p, n);
printf("Maximum Profit: %d\n", maxProfit);
printf("Maximum Weight: %d\n", maxWeight);
return 0;
}
```

<u>Output</u>

```
Enter the number of items: 4

Enter the weights of the items: 5 3 2 4

Enter the profits of the items: 10 5 3 6

Enter the maximum capacity of the knapsack: 8

Maximum Profit: 15

Maximum Weight: 8
```

CONCLUSION:

Branch and bound was studied. The programs for (a) 0/1 knapsack using branch and bound were studied and implemented successfully.