Name	Siddhesh Sonar
UID no.	2021700063
Experiment No.	8

AIM:	To implement 0/1 Knapsack problem using Branch and Bound
Program	
PROBLEM STATEMENT :	To implement 0/1 Knapsack problem using Branch and Bound
ALGORITHM/ THEORY:	**0/1 Knapsack Problem** We are given N items where each item has some weight and profit associated with it. We are also given a bag with capacity W, [i.e., the bag can hold at most W weight in it]. The target is to put the items into the bag such that the sum of profits associated with them is the maximum possible. Note: The constraint here is we can either put an item completely into the bag or cannot put it at all [It is not possible to put a part of an item into the bag]. Branch and bound is an algorithm design paradigm which is generally used for solving combinatorial optimization problems. These problems typically exponential in terms of time complexity and may require exploring all possible permutations in worst case. Branch and Bound solve these problems relatively quickly.

PROGRAM:

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <stdbool.h>

int num;
int pf[100];
int wt[100];
bool temp[100];

int cap = 0;
int maxval = 0;

void getWeightAndProfit(bool inKnap[num], int *weight, int
*profit)
{
    int i, w = 0, v = 0;
    for (i = 0; i < num; ++i)
    {
        if (inKnap[i])
        {
            w += wt[i];
            v += pf[i];
        }
    }
    *weight = w;
    *profit = v;
}

void printProfit(bool inKnap[num])
{
    int val = 0;
    for (int i = 0; i < num; ++i)
    {
        if (inKnap[i])
        {
            val += pf[i];
        }
    }
    printf("\nMax Profit = %d", val);
}

void knapSack(bool inKnap[num], int i)
{
```

```
    int currwt, currval;
    getWeightAndProfit(inKnap, &currwt, &currval);
    if (currwt <= cap)
    {
        if (currval > maxval)
        {
            memcpy(temp,inKnap,sizeof(temp));
            maxval = currval;
        }
    }
    if (i == num || currwt >= cap)
    {
        return;
    }
    int x = wt[i];
    bool use[num], waste[num];
    memcpy(use, inKnap, sizeof(use));
    memcpy(waste, inKnap, sizeof(waste));
    use[i] = true;
    waste[i] = false;
    knapSack(use, i + 1);
    knapSack(waste, i + 1);
}

void printTable(int num, int wt[], int pf[])
{
    printf("|\tItem\t|\tWeight\t|\tProfit\t|\n");
    printf("-------------------------------------------------
\n");
    for (int i = 0; i < num; i++)
    {
        printf("|\t%d\t|\t%d\t|\t%d\t|\n",i+1,wt[i],pf[i]);
    }
}

int main()
{
    printf("Enter the number of items: ");
    scanf(" %d", &num);
    bool inKnap[num];
    int i;
    for (i = 0; i < num; ++i)
    {
        printf("\nWeight of item %d: ", i + 1);
```

```
        scanf("%d", &wt[i]);
        printf("\nProfit of item %d: ", i + 1);
        scanf("%d", &pf[i]);
        inKnap[i] = false;
    }
    printf("\nEnter capacity of Knapsack: ");
    scanf(" %d", &cap);
    printf("\nInput Table:\n\n");
    printTable(num,wt,pf);
    knapSack(inKnap, 0);
    printProfit(temp);
    return 0;
}
```

RESULT:

```
Weight of item 3: 9

Profit of item 3: 20

Weight of item 4: 8

Profit of item 4: 15

Enter capacity of Knapsack: 15

Input Table:

|       Item    |       Weight  |       Profit  |
-------------------------------------------------
|       1       |       2       |       12      |
|       2       |       2       |       10      |
|       3       |       9       |       20      |
|       4       |       8       |       15      |

Max Profit = 42
c:\Siddhesh\Github\DAA\DAA_Exp_8>
```

Item	Weight (lbs)	Value ($)
1	2	12
2	1	10
3	3	20
4	2	15

In this particular example, the optimal solution to the 0/1 Knapsack problem is to include items 2, 3, and 4 in the knapsack, for a total weight of 6 pounds and a total profit of $45.

CONCLUSION:	Successfully understood 0/1 Knapsack Problem and implemented it in C program to calculate maximum profit.