DAA EXP-8

Name: Deepika Trivedi
UID: 2021700069
Batch: Data Science (D-4)

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

Theory:
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.

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.

CODE:

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

typedef enum { NO, YES } BOOL;

int N;
int vals[100];
int wts[100];

int cap = 0;
int mval = 0;

void getWeightAndValue (BOOL incl[N], int *weight, int *value) {
    int i, w = 0, v = 0;
    for (i = 0; i < N; ++i) {
        if (incl[i]) {
            w += wts[i];
            v += vals[i];
        }
    }
    *weight = w;
    *value = v;
}

void printSubset (BOOL incl[N]) {
    int i;
    int val = 0;
    printf("Included = { ");
    for (i = 0; i < N; ++i) {
        if (incl[i]) {
            printf("%d ", wts[i]);
            val += vals[i];
        }
    }
    printf("}; Total value = %d\n", val);
}

void findKnapsack (BOOL incl[N], int i) {
    int cwt, cval;
    getWeightAndValue(incl, &cwt, &cval);
    if (cwt <= cap) {
        if (cval > mval) {
            printSubset(incl);
            mval = cval;
        }
```

```c
        if (cval > mval) {
            printSubset(incl);
            mval = cval;
        }
    }
    if (i == N || cwt >= cap) {
        return;
    }
    int x = wts[i];
    BOOL use[N], nouse[N];
    memcpy(use, incl, sizeof(use));
    memcpy(nouse, incl, sizeof(nouse));
    use[i] = YES;
    nouse[i] = NO;
    findKnapsack(use, i+1);
    findKnapsack(nouse, i+1);
}

int main(int argc, char const * argv[]) {
    printf("Enter the number of elements: ");
    scanf(" %d", &N);
    BOOL incl[N];
    int i;
    for (i = 0; i < N; ++i) {
        printf("Enter weight and value for element %d: ", i+1);
        scanf(" %d %d", &wts[i], &vals[i]);
        incl[i] = NO;
    }
    printf("Enter knapsack capacity: ");
    scanf(" %d", &cap);
    findKnapsack(incl, 0);
    return 0;
}
```

RESULT:

```
input
Enter the number of elements: 4
Enter weight and value for element 1: 1 15
Enter weight and value for element 2: 5 10
Enter weight and value for element 3: 3 9
Enter weight and value for element 4: 4 5
Enter knapsack capacity: 8
Included = { 1 }; Total value = 15
Included = { 1 5 }; Total value = 25
Included = { 1 3 4 }; Total value = 29


...Program finished with exit code 0
Press ENTER to exit console.
```

CONCLUSION: We understood the working of Knapsack problem and found out that:

Time Complexity: O(N), as only one path through the tree will have to be traversed in the beat case and its worst time complexity is still given as O(2N) .