NAME: SADIQ ALI
FACULTY NO. : 18COB204
ENROLLMENT NO. : GI0206

QUESTION:

Part1: Write a program in C/C++ to solve the 0/1 knapsack problem using (i) Dynamic Programming based algorithm and (ii) Branch and Bound Search based algorithm Go through the related text and implement each of these algorithms using the efficient data structure. Show the results of different steps of these algorithms for an instance of the 0/1 Knapsack problem with number of items, n=4. The capacity of the knapsack, weights and profits of the items may be generated randomly with the condition that the capacity of the knapsack is such that all items can not be accommodated in the knapsack. But, at the same time, at least one item can be accommodated in the knapsack.

Part2: Analyze the complexity of these algorithms (Run each of the two algorithms for a set of ten randomly generated 0/1 knapsack instances (with n=4) and compute the time taken by the selected implementation in each run. Compute average time taken by each of these two algorithms.

SOLUTION:

PART 1:

A) 0/1 knapsack problem using dynamic programming.

For time calculation i used, a library named <chrono> and by using two variables start and end to calculate the time (in microseconds) at the start of the program and end of the program. By calculating these two variables we can calculate the time taken by the program in microseconds.

C++ Implementaion of the algorithm is:

1. Importing Important Libraries:

```cpp
#include <iostream>
#include <ctime>
#include <chrono>
using namespace std;
using namespace std::chrono;

```

2. Function which returns maximum value:

```cpp

int max(int a, int b)
{
    return (a > b) ? a : b;
}

```

3. Main algorithm for knapsack problem:

```cpp

int knapSack(int W, int wt[], int pr[], int n)
{
    int i, w;
    int K[n + 1][W + 1];

    // Build table K[][] in bottom up manner
    for (i = 0; i <= n; i++) {
        for (w = 0; w <= W; w++) {
            if (i == 0 || w == 0)
                K[i][w] = 0;
            else if (wt[i - 1] <= w)
                K[i][w] = max(
                    pr[i - 1] + K[i - 1][w - wt[i - 1]],
                    K[i - 1][w]);
            else
                K[i][w] = K[i - 1][w];
        }
    }

    return K[n][W];
}
```

4. In Main function i am generating random integer values for profit and weights of items and max. Possible capacity of the knapsack. And also calculating the time taken by the algorithm.

```
int main()
{
    srand(time(0));
    int n  = 4;
    int pr[n];
    int W=(rand()%1000) + 1;

    cout << "Weight of the bag:" << " " <<W << endl;

    for(int i = 0; i < n; i++){
        pr[i] = rand()%100 + 1;
    }
    cout << "printing the profit of itmes"<< endl;

    for(int i = 0; i < n; i++){
        cout << pr[i] << " " ;
    }
    int wt[items];

    for(int i = 0; i < n; i++){
        wt[i] = (rand()%(W/2));
    }
    cout << endl;
    cout << "printing the weight of itmes"<< endl;

    for(int i = 0; i < n; i++){
        cout << wt[i] << " " ;
    }
    cout << endl;

    auto start = high_resolution_clock::now();

    cout << "Maximum profit of the items is: "<< endl;
    cout << knapSack(W, wt, val, items);
    // printf("%d", knapSack(W, wt, val, items));
    auto end = high_resolution_clock::now();

    cout << endl;
    auto duration = duration_cast<microseconds>(end - start);

    cout << "Time taken by function: "<< duration.count() << " microseconds'

    return 0;
```

B) 0/1 Knapsack problem using branch and bound method.

For time calculation i used, a library named <chrono> and by using two variables start and end to calculate the time (in microseconds) at the start of the program and end of the program. By calculating these two variables we can calculate the time taken by the program in microseconds.

C++ Implementaion of the algorithm is:

1. Importing Important Libraries.

```
#include <iostream>
#include <bits/stdc++.h>
#include <chrono>
using namespace std;
using namespace std::chrono;

```

2. Defining structure Item which conatins weight and profit. Another structure named Node which contains level, profit, bound and weight.

```
struct Item
{
    float weight;
    int value;
};

struct Node
{
    int level, profit, bound;
    float weight;
};
```

3. Function to compare two Items.

```
bool cmp(Item a, Item b)
{
    double r1 = (double)a.value / a.weight;
    double r2 = (double)b.value / b.weight;
    return r1 > r2;
}
```

4. Bound Function

```
int bound(Node u, int n, int W, Item arr[])
{
    // if weight overcomes the knapsack capacity, return
    // 0 as expected bound
    if (u.weight >= W)
        return 0;

    // initialize bound on profit by current profit
    int profit_bound = u.profit;

    // start including items from index 1 more to current
    // item index
    int j = u.level + 1;
    int totweight = u.weight;

    // checking index condition and knapsack capacity
    // condition
    while ((j < n) && (totweight + arr[j].weight <= W))
    {
        totweight += arr[j].weight;
        profit_bound += arr[j].value;
        j++;
    }

    // If k is not n, include last item partially for
    // upper bound on profit
    if (j < n)
        profit_bound += (W - totweight) * arr[j].value /
                                         arr[j].weight;

    return profit_bound;
}
```

5. Knapsack function which will returns the maximum possible profit.

```
// Returns maximum profit we can get with capacity W
int knapsack(int W, Item arr[], int n)
{
    sort(arr, arr + n, cmp);
    queue<Node> Q;
    Node u, v;

    u.level = -1;
    u.profit = u.weight = 0;
    Q.push(u);

    int maxProfit = 0;
    while (!Q.empty())
    {
        u = Q.front();
        Q.pop();

        if (u.level == -1)
            v.level = 0;

        if (u.level == n-1)
            continue;

        v.level = u.level + 1;

        v.weight = u.weight + arr[v.level].weight;
        v.profit = u.profit + arr[v.level].value;

        if (v.weight <= W && v.profit > maxProfit)
            maxProfit = v.profit;

        v.bound = bound(v, n, W, arr);

        if (v.bound > maxProfit)
            Q.push(v);
        v.weight = u.weight;
        v.profit = u.profit;
        v.bound = bound(v, n, W, arr);
        if (v.bound > maxProfit)
            Q.push(v);
    }
    return maxProfit;
}
```

6. Main function in which i am randomly generated Items, and also printing them. Main function also print the time taken by the algorithm in microseconds.

```cpp

// driver program to test above function
int main()
{
    srand((unsigned int)time(0));
    int W = rand()%100 + 1; // Weight of knapsack
    cout << "Weight of the knapsack: " << W << endl;
    Item arr[4];

    for(int i = 0; i < 4; i++){
        float wt = (float (rand()%(W/2)) + 1);
        int p = rand()%100 + 1;
        cout << "wt and p is: " << "{ " << wt << ", " << p << "}" << endl;
        arr[i] = {wt, p};
    }

    int n = sizeof(arr) / sizeof(arr[0]);

    high_resolution_clock::time_point start = high_resolution_clock::now();
    cout << "Maximum possible profit = "<< knapsack(W, arr, n);
    auto stop = high_resolution_clock::now();
    cout << endl;
    high_resolution_clock::time_point end = high_resolution_clock::now();
    auto timetaken = duration_cast<microseconds>(end - start).count();

    cout << "Time taken by function: "<< timetaken<< " microseconds" << endl;

    return 0;
}

```

PART 2: Computing average time taken by the two algorithms above:

A) Time taken by solving the problem by dynamic programming.

Few of the examples of the input and output are:

```
sadiq@sadiqali:~/Downloads$ ./a.out
Weight of the bag: 276
printing the profit of itmes
18 50 22 93
printing the weight of itmes
8 18 106 5
Maximum profit of the items is:
183
Time taken by function: 54 microseconds
sadiq@sadiqali:~/Downloads$ ./a.out
Weight of the bag: 281
printing the profit of itmes
38 100 70 15
printing the weight of itmes
35 114 67 136
Maximum profit of the items is:
208
Time taken by function: 34 microseconds
sadiq@sadiqali:~/Downloads$ ./a.out
Weight of the bag: 459
printing the profit of itmes
76 50 69 21
printing the weight of itmes
93 58 116 62
Maximum profit of the items is:
216
Time taken by function: 90 microseconds
sadiq@sadiqali:~/Downloads$ ./a.out
Weight of the bag: 40
printing the profit of itmes
65 21 73 86
printing the weight of itmes
12 11 8 7
Maximum profit of the items is:
245
Time taken by function: 12 microseconds
sadiq@sadiqali:~/Downloads$ ./a.out
Weight of the bag: 525
printing the profit of itmes
18 54 45 62
printing the weight of itmes
235 232 159 61
Maximum profit of the items is:
161
Time taken by function: 65 microseconds
```

B) Time taken by solving the problem by Branch and Bound Method.

Few of the examples of the input and output are:

```
sadiq@sadiqali:~/Downloads$ g++ Branch.cpp
sadiq@sadiqali:~/Downloads$ ./a.out
Weight of the knapsack: 66
wt and p is: { 8, 28}
wt and p is: { 21, 95}
wt and p is: { 3, 3}
wt and p is: { 28, 19}
Maximum possible profit = 145
Time taken by function: 21 microseconds
sadiq@sadiqali:~/Downloads$ ./a.out
Weight of the knapsack: 88
wt and p is: { 41, 12}
wt and p is: { 18, 48}
wt and p is: { 41, 19}
wt and p is: { 8, 47}
Maximum possible profit = 114
Time taken by function: 23 microseconds
sadiq@sadiqali:~/Downloads$ ./a.out
Weight of the knapsack: 58
wt and p is: { 8, 95}
wt and p is: { 1, 35}
wt and p is: { 14, 89}
wt and p is: { 21, 99}
Maximum possible profit = 318
Time taken by function: 31 microseconds
sadiq@sadiqali:~/Downloads$ ./a.out
Weight of the knapsack: 6
wt and p is: { 1, 13}
wt and p is: { 3, 49}
wt and p is: { 2, 15}
wt and p is: { 2, 93}
Maximum possible profit = 155
Time taken by function: 26 microseconds
sadiq@sadiqali:~/Downloads$
```

Table for Time Comparison of both the methods :

S.No	Time for Dynamic prog (µs).	Time for Branch and Bound(µs).
1.	74	21
2.	69	23
3.	56	26
4.	99	20
5.	50	27
6.	58	30
7.	59	29
8.	82	23
9.	76	21
10.	69	29

Average time taken :

1) By Dynamic Programming = **69.2 µs**

2) By Branch and Bound Method = **24.9 µs**

Observation :

From the above table and calculated average time we can see that the time taken for solving the knapsack problem by dynamic programming is much greater than the time taken for solving the problem by Branch and Bound method.