Indian Institute of Information Technology
Sonepat

DESIGN AND ANALYSIS OF ALGORITHMS LAB

CSL-307

Submitted to:

Mr. Md. Arqam.

Submitted by:

Shivansh Joshi

Branch - CSE

Roll No: 12011042.

EXPERIMENT 10

AIM:

Write a program to implement fractional knapsack and 0/1 knapsack Algorithm. Also do the complexity analysis for both.

Source Code:

1. Fractional Knapsack Problem.

```
#include <stdio.h>

#include <stdlib.h>

#include <math.h>



void heapify(int no[], float ratio[], int size, int i)

{

    int largest = i;

    int left = 2 * i + 1;

    int right = 2 * i + 2;

    float temp;



    if (left < size && ratio[left] > ratio[largest])

        largest = left;



    if (right < size && ratio[right] > ratio[largest])
```

```
        largest = right;


    if (largest != i)
    {
        temp = ratio[i];
        ratio[i] = ratio[largest];
        ratio[largest] = temp;
        temp = no[i];
        no[i] = no[largest];
        no[largest] = temp;
        heapify(no, ratio, size, largest);
    }
}


void heapSort(int no[], float ratio[], int size)
{
    int i;
    float temp;
    for (i = size / 2 - 1; i >= 0; i--)
        heapify(no, ratio, size, i);
    for (i = size - 1; i >= 0; i--)
    {
        temp = ratio[0];
        ratio[0] = ratio[i];
```

```
        ratio[i] = temp;

        temp = no[0];

        no[0] = no[i];

        no[i] = temp;

        heapify(no, ratio, i, 0);

    }

}


void fractional_knapsack(float quantity[], float w[], float p[], int no[],
float W, int n)

{

    float weight = 0;

    for (int a = n - 1; a >= 0; a--)

    {

        if ((weight + w[no[a] - 1]) <= W)

        {

            quantity[n - a - 1] = 1;

            weight = weight + w[no[a] - 1];

        }

        else

        {

            quantity[n - a - 1] = (W - weight) / w[no[a] - 1];

            weight = W;

            break;
```

```
        }
    }
}

int main()
{
    int n;
    printf("Enter the number of items: ");
    scanf("%d", &n);
    float *w = (float *)malloc(n * sizeof(float));
    float *p = (float *)malloc(n * sizeof(float));
    int *no = (int *)malloc(n * sizeof(int));
    float *quantity = (float *)malloc(n * sizeof(float));
    float *ratio = (float *)malloc(n * sizeof(float));

    printf("Enter weights: \n");
    for (int a = 0; a < n; a++)
    {
        printf("%d. ", a + 1);
        scanf("%f", &w[a]);
    }

    printf("Enter profits: \n");
    for (int a = 0; a < n; a++)
```

```
    {
        printf("%d. ", a + 1);
        scanf("%f", &p[a]);
    }
    float W;

    printf("Enter the capacity of the knapsack: ");
    scanf("%f", &W);

    for (int a = 1; a <= n; a++)
    {
        no[a - 1] = a;
        ratio[a - 1] = p[a - 1] / w[a - 1];
        quantity[a - 1] = 0;
    }
    heapSort(no, ratio, n);
    fractional_knapsack(quantity, w, p, no, W, n);
    float profit = 0;
    printf("Quantities of item chosen: \n");
    for (int a = 1; a <= n; a++)
    {
        printf("Item %d: %f\n", no[n - a], quantity[a - 1]);
        profit = profit + quantity[a - 1] * p[n - a];
    }
```

```
    printf("Total profit is: %f", profit);

    return 0;

}
```

Output:

```
Enter the number of items: 6
Enter weights:
1. 12
2. 23
3. 15
4. 18
5. 11
6. 20
Enter profits:
1. 120
2. 150
3. 140
4. 200
5. 210
6. 180
Enter the capacity of the knapsack: 60
Quantities of item chosen:
Item 5: 1.000000
Item 4: 1.000000
Item 1: 1.000000
Item 3: 1.000000
Item 6: 0.200000
Item 2: 0.000000
Total profit is: 760.000000

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

```
Enter the number of items: 10
Enter weights:
1. 12
2. 23
3. 11
4. 24
5. 25
6. 10
7. 27
8. 37
9. 33
10. 43
Enter profits:
1. 120
2. 100
3. 140
4. 160
5. 210
6. 220
7. 200
8. 190
9. 150
10. 140
Enter the capacity of the knapsack: 70
Quantities of item chosen:
Item 6: 1.000000
Item 3: 1.000000
Item 1: 1.000000
Item 5: 1.000000
Item 7: 0.444444
Item 4: 0.000000
Item 8: 0.000000
Item 9: 0.000000
Item 2: 0.000000
Item 10: 0.000000
Total profit is: 777.777771

...Program finished with exit code 0
```

Complexity Analysis:

The fractional knapsack problem is a perfect example of the greedy approach.

We fill the knapsack with the items carrying the highest profits while also considering the weight of the item. So the value we need to maximise is the profit to weight ratio.

$$r_i = \frac{p_i}{w_i}$$

where p_i is the profit and w_i is the weight of the i^{th} item.

Then following the greedy technique, we select the item with the highest profit to weight ratio and put it in the knapsack. In this way we fill the knapsack with items in a decreasing order of profit to weight ratio. When the knapsack can no longer accommodate the whole of the next item, the knapsack is filled with the fraction of the item which can be placed in the knapsack. In this way the profit is maximised.

First of all we need to arrange the profit to weight ratios in a decreasing order. This can achieved in $O(nlgn)$ time using heapsort.

Next the items are placed one by one in the knapsack until its full using the following function.

```
float weight = 0;

for(int a =n-1; a>=0; a--)

{
```

```
    if((weight + w[no[a]-1])<=W)
    {
        quantity[n-a-1] = 1;
        weight = weight + w[no[a]-1];
    }
    else
    {
        quantity[n-a-1] = (W - weight) / w[no[a]-1];
        weight = W;
        break;
    }
}
```

The given for loop runs n times, where n is the number of available items.

The array no[] stores the item number in decreasing order of their profit to weight ratio. As the ratios are stored in an ascending order, the loop runs backwards.

The loop checks if the incoming item can be accommodated in the knapsack as a whole or not. If yes, then the item is added and if not then the fraction of item which can be contained is added and the loop ends as the knapsack is full.

The maximum number of times the loop can run is n(the number of items) when all of the items can be accomodated in the knapsack.

At the least, it runs for one time, when the item with the highest profit to weight ratio cannot fit into the knapsack.

So time complexity of fractional knapsack function:

Best Case: $T(n) = O(1)$.

Worst Case: $T(n) = O(n)$.

On an average, it runs for somewhere between one to n times. So averge case is also $O(n)$.

Total time complexity using heap sort:

$$T(n) = O(nlgn) + O(1) = O(nlgn). \ (Best\ case.)$$

$$T(n) = O(nlgn) + O(n) = O(nlgn). \ (Worst\ or\ average\ case.)$$

However for best case, time complexity can be reduced using insertion sort as it has time complexity $O(n)$ in best case.

Total time complexity using Insertion sort:

$$T(n) = O(n) + O(1) = O(n). \ (Best\ case.)$$

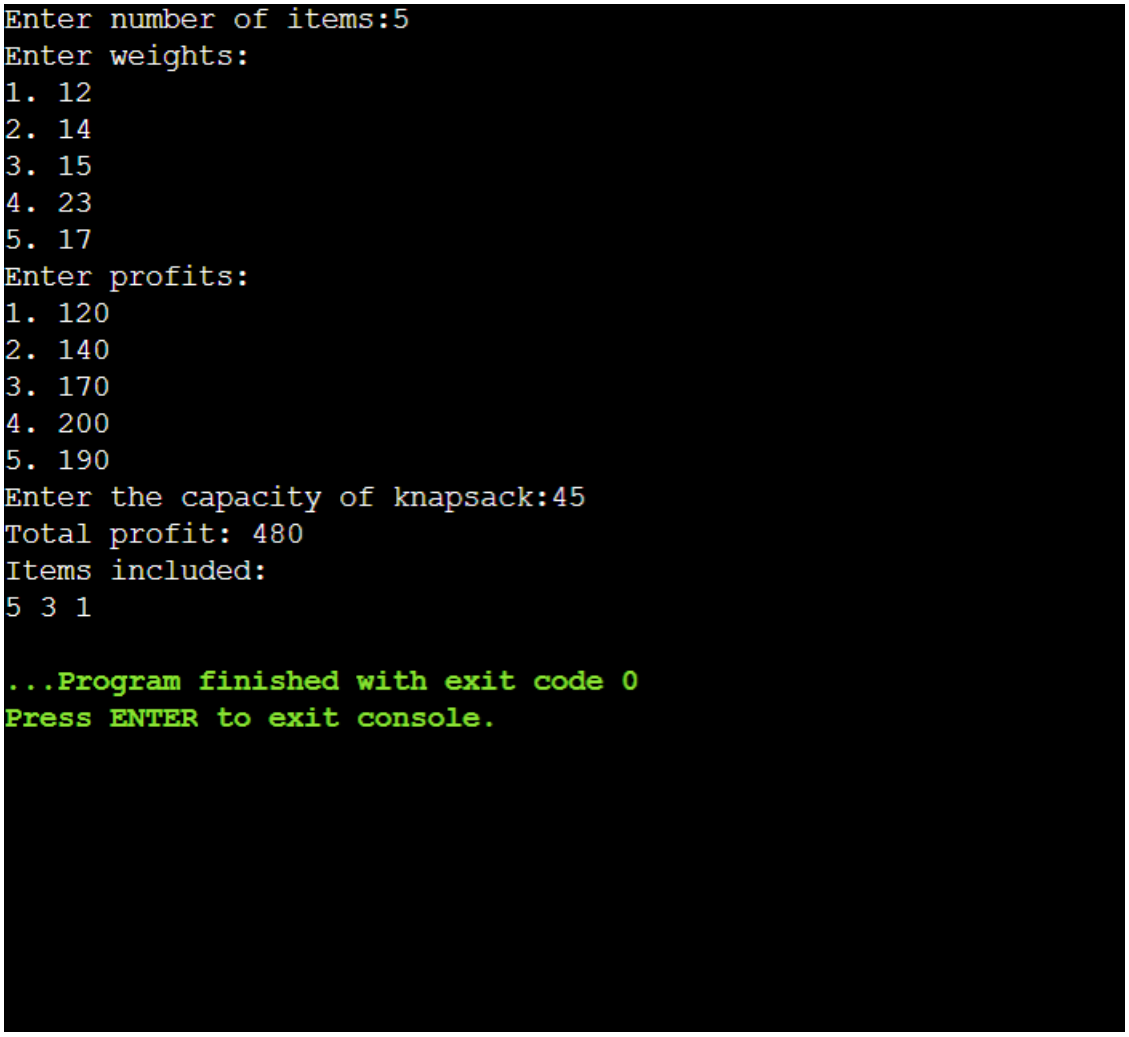
$$T(n) = O\left(n^2\right) + O(n) = O\left(n^2\right). \ (Worst\ or\ average\ case.)$$

2. 0/1 Knapsack problem.

```
#include <stdio.h>

#include <stdlib.h>

int max(int a, int b)

{

    return (a > b) ? a : b;

}

void knapSack(int W, int wt[], int p[], int n)

{

    int i, w;

    int K[n + 1][W + 1];

    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(p[i - 1] + K[i - 1][w - wt[i - 1]], K[i -
1][w]);

            else

                K[i][w] = K[i - 1][w];

        }

    }
```

```
    int res = K[n][W];
    printf("Total profit: %d\n", res);
    printf("Items included:\n");

    w = W;
    for (i = n; i > 0 && res > 0; i--)
    {
        if (res == K[i - 1][w])
            continue;
        else
        {
            printf("%d ", i);
            res = res - p[i - 1];
            w = w - wt[i - 1];
        }
    }
}
int main()
{
    int i, n, W;

    printf("Enter number of items:");
    scanf("%d", &n);
    int *p = (int *)malloc(n * sizeof(int));
```

```
    int *wt = (int *)malloc(n * sizeof(int));


    printf("Enter weights: \n");
    for (int a = 0; a < n; a++)
    {
        printf("%d. ", a + 1);
        scanf("%d", &wt[a]);
    }


    printf("Enter profits: \n");
    for (int a = 0; a < n; a++)
    {
        printf("%d. ", a + 1);
        scanf("%d", &p[a]);
    }


    printf("Enter the capacity of knapsack:");
    scanf("%d", &W);
    knapSack(W, wt, p, n);
    return 0;
}
```


Enter number of items:5
Enter weights:
1. 12
2. 14
3. 15
4. 23
5. 17
Enter profits:
1. 120
2. 140
3. 170
4. 200
5. 190
Enter the capacity of knapsack:45
Total profit: 480
Items included:
5 3 1
...Program finished with exit code 0
Press ENTER to exit console.

OUTPUT:

```
Enter number of items:10
Enter weights:
1. 12
2. 23
3. 15
4. 23
5. 21
6. 22
7. 30
8. 31
9. 18
10. 26
Enter profits:
1. 120
2. 150
3. 210
4. 220
5. 190
6. 230
7. 300
8. 110
9. 160
10. 90
Enter the capacity of knapsack:70
Total profit: 750
Items included:
6 5 3 1
```

Complexity Analysis:

The 0/1 knapsack problem can be solved efficiently using the dynamic programming approach. Unlike the fractional knapsack problem, items cannot be added to the knapsack in fractional amounts. This is the reason why greedy approach fails in this problem. The DP approach calculates all the possibilities in which the knapsack can be filled maximising the profit while in the constraint of weight.

The function maintains an array K[][] with number of rows as the number of items and number of columns as the capacity of the knapsack, which stores the items to be included in the knapsack according to its weight and profit. The schema for the filling of the knapsack is:

$$T(i,j) = \left\{T(i\ -\ 1, j)\ ,\ p_{_i} + T\left(i\ -\ 1, j\ -\ wt_{_i}\right)\right\}.$$

Therefore, the fill the whole array of n+1 rows and W+1 columns, total number of iterations will be $(n\ +\ 1) \times (W\ +\ 1)$.

Time taken to fill one entry of the table $=\ O(1)$.

Total time taken to fill the matrix
$=\ (n\ +\ 1) \times (W\ +\ 1) \times O(1) =\ O(n \times W)$.

The resultant profit is stored in the last row last column of the array, the retrieval of which takes $O(1)$ time.

Time taken to find total profit $=\ O(n \times W)$.

Now to find the items included in the knapsack requires backtracking of the array to find out which path did the overall optimum solution followed. A for loop is used to traverse the array

K row wise from bottom to up. It takes n iterations and each iteration takes $O(1)$ time.

Time taken to find items included $= O(n)$.

Therefore total time taken to solve the 0/1 knapsack problem with n items and capacity of the knapsack as W is,

$$T(n, W) = O(n \times W) + O(n) = O(n \times W).$$