Name: Chirag Rana
Class: SE Comps
Batch: C
UID: 2018130043

EXPERIMENT – 5

AIM: To perform experiments based on Knapsack Problem (C, B) using a greedy strategy.

THEORY:

The Greedy algorithm could be understood very well with a well-known problem referred to as Knapsack problem. Although the same problem could be solved by employing other algorithmic approaches, Greedy approach solves Fractional Knapsack problem reasonably in a good time. Let us discuss the Knapsack problem in detail.

Knapsack Problem

Given a set of items, each with a weight and a value, determine a subset of items to include in a collection so that the total weight is less than or equal to a given limit and the total value is as large as possible.

The knapsack problem is in combinatorial optimization problem. It appears as a subproblem in many, more complex mathematical models of real-world problems. One general approach to difficult problems is to identify the most restrictive constraint, ignore the others, solve a knapsack problem, and somehow adjust the solution to satisfy the ignored constraints.

Applications

In many cases of resource allocation along with some constraint, the problem can be derived in a similar way of Knapsack problem. Following is a set of example.

- Finding the least wasteful way to cut raw materials
- portfolio optimization
- Cutting stock problems

Problem Scenario

A thief is robbing a store and can carry a maximal weight of W into his knapsack. There are n items available in the store and weight of i^{th} item is w_i and its profit is p_i. What items should the thief take?

In this context, the items should be selected in such a way that the thief will carry those items for which he will gain maximum profit. Hence, the objective of the thief is to maximize the profit.

Based on the nature of the items, Knapsack problems are categorized as

- Fractional Knapsack

- Knapsack

Fractional Knapsack

In this case, items can be broken into smaller pieces, hence the thief can select fractions of items.

According to the problem statement, □ There are n items

in the store

- Weight of i^{th} item wi>0.
- Profit for i^{th} item pi>0 and
- Capacity of the Knapsack is W

In this version of Knapsack problem, items can be broken into smaller pieces. So, the thief may take only a fraction x_i of i^{th} item.

$0 \leqslant xi \leqslant 1$

The i^{th} item contributes the weight xi.wixi.wi to the total weight in the knapsack and profit xi.pixi.pi to the total profit. Hence, the objective of this algorithm is to

maximize$\sum(n=1 \text{ to } n)(xi.pi)$

subject to constraint,

$\sum(n=1 \text{ to } n)(xi.wi) \leqslant W$

It is clear that an optimal solution must fill the knapsack exactly, otherwise we could add a fraction of one of the remaining items and increase the overall profit.

Thus, an optimal solution can be obtained by:

$\sum(n=1 \text{ to } n)(xi.wi)=W\sum n=1n(xi.wi)=W$

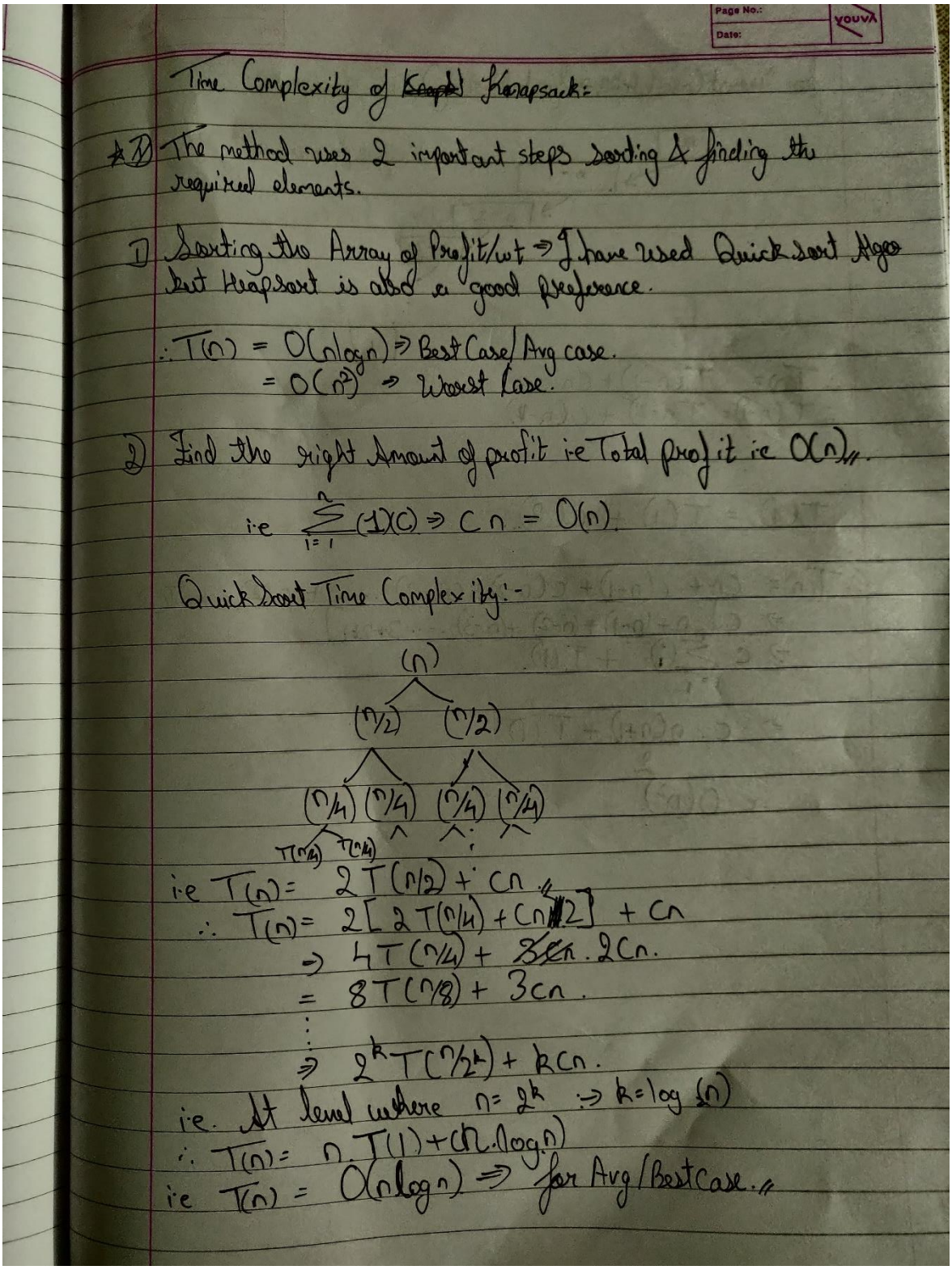
In this context, first we need to sort those items according to the value of pi/wi , so that pi+1/wi+1≤ pi/wi.

ALGORITHM:

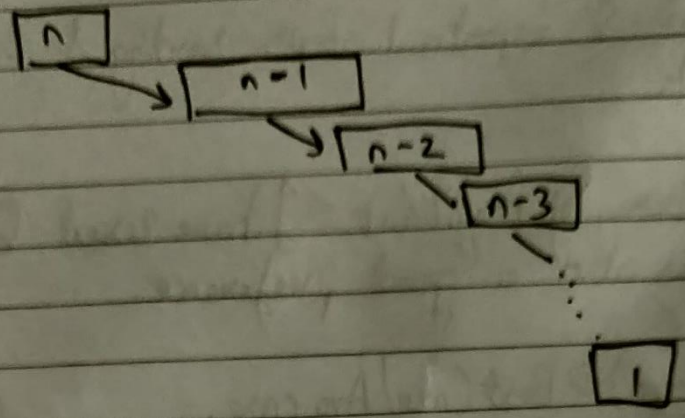
```
Fractional Knapsack (Array Wt, Array profit, int knapsack_capacity)
1.      for i <- 1 to size (V)
2.      calculate cost[i] <- profit[i] / Wt[i]
3.      Sort-Descending (cost)
4.      i ← 0
5.      while (i <= size(profit)-1)
6.      if  Wt[i] <= knapsack_capacity
7.      knapcapacity ← knapcapacity – Wt[i]
8.      total_profit ← total_profit + profit[i];
```

```
9.          i←i+1
10.         else if  Wt[i] > knapsack_capacity
11.         total_profittotal_← profit+ knapsack_cap*cost[i];
12.         i ← i+1
```

TIME COMPLEXITY:



for Worst Case (A.P.a ⇒ Sorted Array).



$\therefore T(n) = T(n-1) + cn$

$\therefore T(n-1) = T(n-2) + c(n-1)$

$T(2) = T(1) + c.(2)$...

$\therefore T(n) = cn + c(n-1) + c(n-2) + c(n-3) \ldots + c(2) + T(1)$

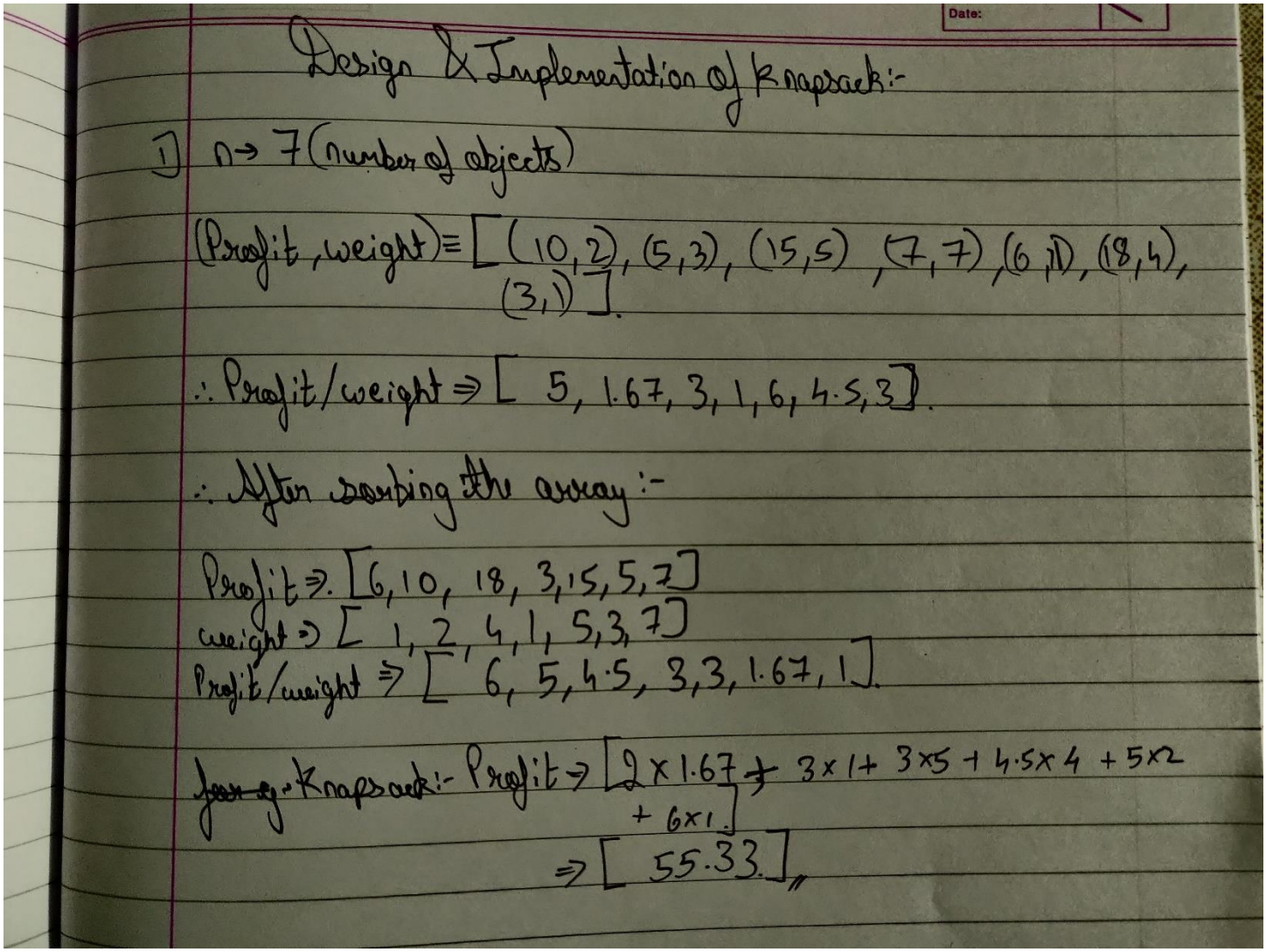
$\Rightarrow c[n + (n-1) + (n-2) + (n-3) \ldots 3+2+1]$

$\Rightarrow c \sum_{i=1}^{n} (i) + T(1).$

$\Rightarrow c. \frac{n(n+1)}{2} + T(1)$

i.e. $O(n^2)$.

DESIGN AND IMPLEMENTATION:



CODE:

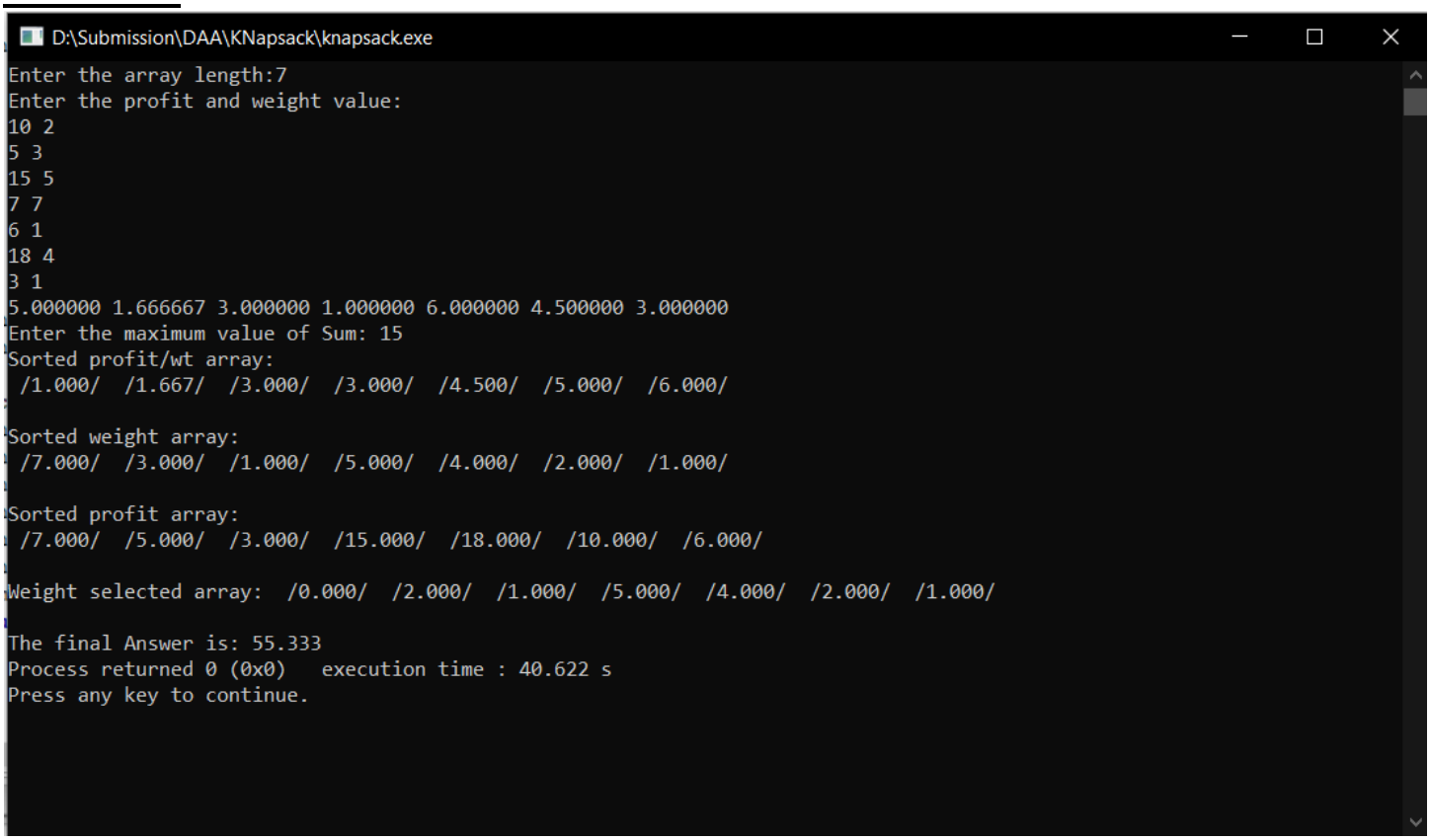
```
 #include<stdio.h>
#include<stdlib.h>
void swap(float* a, float* b)
{
	float t = *a;
	*a = *b;
	*b = t;
}

int partition (float arr[],float arrwe[],float arrpr[], int low, int high)
{
	float pivot = arr[high];
	int i = (low - 1);
```

```
        for (int j = low; j <= high- 1; j++)
        {
                if (arr[j] < pivot)
                {
                        //printf("The pivot: %f",pivot);
                        i++;
                        swap(&arr[i], &arr[j]);
                        swap(&arrwe[i], &arrwe[j]);
                        swap(&arrpr[i], &arrpr[j]);
                }
        }
        swap(&arr[i + 1], &arr[high]);
        swap(&arrwe[i + 1], &arrwe[high]);
        swap(&arrpr[i + 1], &arrpr[high]);

        return (i + 1);
}

void quickSort(float arr[],float arrwe[],float arrpr[], int low, int high)
{
        if (low < high)
        {
                int pi = partition(arr,arrwe,arrpr, low, high);
                quickSort(arr,arrwe,arrpr, low, pi - 1);
                quickSort(arr,arrwe,arrpr, pi + 1, high);
        }
}

void printArray(float arr[], int size)
{
        for (int i=0; i < size; i++)
                printf(" /%.3f/ ", arr[i]);
        printf("\n\n");
}

void submitsolution(float arrra[],float arrwe[],float arrpr[],int sum,int length){
   int temp = 0;
```

```
    float finalval = 0;
    float* arr = (float*)malloc(length*sizeof(float));
    for (int i = length-1;i>=0; i --){
            if(temp < sum){
                if((sum-temp) >= arrwe[i]){
                    temp += arrwe[i];
                    arr[i] = arrwe[i];
                    finalval += arr[i]*arrra[i];
                }else{
                    arr[i] = sum-temp;
                    finalval += arr[i]*arrra[i];
                    temp = sum;
                    break;
                }
            }
    }
    printf("Weight selected array: ");
    printArray(arr,length);
    printf("The final Answer is: %.3f",finalval);
}
int main()
{
    int n,sum;

    float *arrpr,*arrwe;
    float *arrra;
    printf("Enter the array length:");
        scanf("%d",&n);

        arrpr =  (float*)malloc(n*sizeof(float));
        arrwe =  (float*)malloc(n*sizeof(float));
        arrra =  (float*)malloc(n*sizeof(float));

        printf("Enter the profit and weight value:");

    for (int i = 0; i< n;i++){
       scanf("%f",&arrpr[i]);
       scanf("%f",&arrwe[i]);
```

```
        arrra[i] = arrpr[i]/arrwe[i];
    }
        for (int i = 0; i< n;i++){
                printf("%f ",arrra[i]);
        }
    printf(" \nEnter the maximum value of Sum: ");
    scanf("%d",&sum);

        quickSort(arrra,arrwe,arrpr,0, n-1);
        printf("Sorted profit/wt array: \n");
        printArray(arrra, n);
        printf("Sorted weight array: \n");
        printArray(arrwe, n);
        printf("Sorted profit array: \n");
        printArray(arrpr, n);
    submitsolution(arrra,arrwe,arrpr,sum,n);
        return 0;
}
// 10 2 5 3 15 5 7 7 6 1 18 4 3 1
```

OUTPUT:



CONCLUSION:

1. I implemented Fractional Knapsack Problem with the Quicksort algorithm.
2. I understood how to sort three array parallelly.