Practical 1

**Aim: -** Implement and analyze algorithms given below:

**1.1 Factorial (Iterative and recursive)**

**Program: -**

#include <iostream> using namespace std; int counter=1;

int fact(int no){ if(no==1){

return 1; counter++;

}

else{

counter++;

return no\*fact(no-1);

}

}

int main()

{ int n;

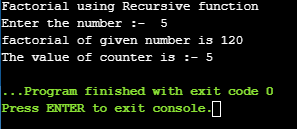
cout<<"Factorial using Recursive function" <<endl; cout<<"Enter the number :- ";

cin>>n;

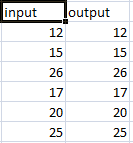
cout<<"factorial of given number is "<<fact(n); cout<<"\nThe value of counter is :- "<< counter; return 0;

}

**Output:-**



# Analysis Table:-



**Graph:**



30

25

20

15

Series1

Series2

10

5

0

1

2

3

4

5

6

**Aim: -** Implement and analyze algorithms given below Factorial (**Iterative**).

**Program: -**

#include <iostream> using namespace std; int counter=0;

int main()

{ int n,no=1;

cout<<"Factorial using Iterative method" <<endl; cout<<"Enter the number :- ";

cin>>n;

for(int i=1;i<=n;++i){ no\*=i;

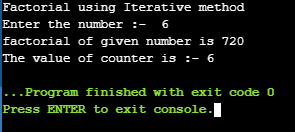
counter++;

}

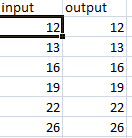
cout<<"factorial of given number is "<<no; cout<<"\nThe value of counter is :- "<< counter; return 0;

}

**Output:-**



# Analysis Table:-



**Graph:**

30

25

20

15

Series1

Series2

10

5

0

1

2

3

4

5

6

**Conclusion:-**

From this practical, I have learnt how to analyze the algorithms using graphs.

**1.2 Euclidean Algorithm**

**PROGRAM:-**

#include<stdio.h>

int counter = 0; int gcd(int x,int y)

{

while(y!=0)

{

counter++; int temp = x%y;

x = y; y = temp;

}

return x;

}

int main() {

int n1,n2,result;

printf("------Euclidean Algorithm \n");

printf("Enter First Integer: "); scanf("%d",&n1);

printf("\nEnter Second Integer: "); scanf("%d",&n2);

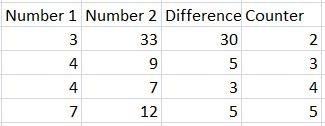
result = gcd(n1,n2);

printf("\nGCD of %d and %d is %d",n1,n2,result); printf("\nThe counter is %d",counter);

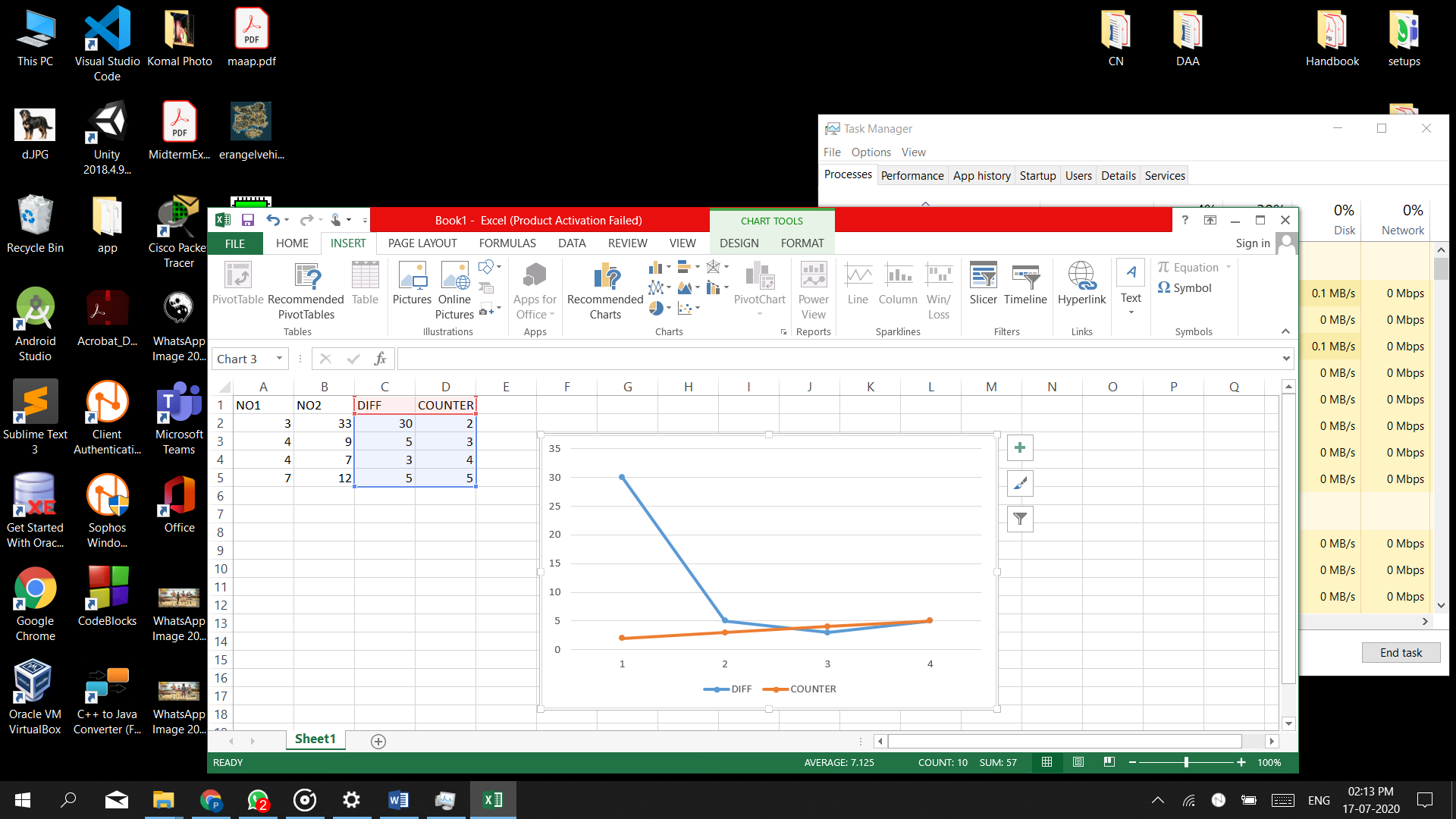
}

# OUTPUT:-

**ANALYSIS TABLE:-**



**GRAPH:-**

****

**CONCLUSION:-**

From this practical, I learnt how to code GCD for Euclidean algorithm and find its count it requires to find the solution. I was also able to do analysis and develop a graph for the same.

**1.3 (1) Matrix Addition**

**PROGRAM:-**

#include <iostream>

using namespace std;

int main()

{

int r, c, a[100][100], b[100][100], sum[100][100], i, j,counter=0;

cout << "Enter number of rows (between 1 and 100): ";

cin >> r;

cout << "Enter number of columns (between 1 and 100): ";

cin >> c;

cout << endl << "Enter elements of 1st matrix: " << endl;

// Storing elements of first matrix entered by user.

for(i = 0; i < r; ++i)

for(j = 0; j < c; ++j)

{

cout << "Enter element a" << i + 1 << j + 1 << " : ";

cin >> a[i][j];

}

// Storing elements of second matrix entered by user.

cout << endl << "Enter elements of 2nd matrix: " << endl;

for(i = 0; i < r; ++i)

for(j = 0; j < c; ++j)

{

cout << "Enter element b" << i + 1 << j + 1 << " : ";

cin >> b[i][j];

}

// Adding Two matrices

for(i = 0; i < r; ++i){

for(j = 0; j < c; ++j){

sum[i][j] = a[i][j] + b[i][j];

counter++;

}

}

cout<<"\nCounter: "<<counter;

// Displaying the resultant sum matrix.

cout << endl << "Sum of two matrix is: " << endl;

for(i = 0; i < r; ++i)

for(j = 0; j < c; ++j)

{

cout << sum[i][j] << " ";

if(j == c - 1)

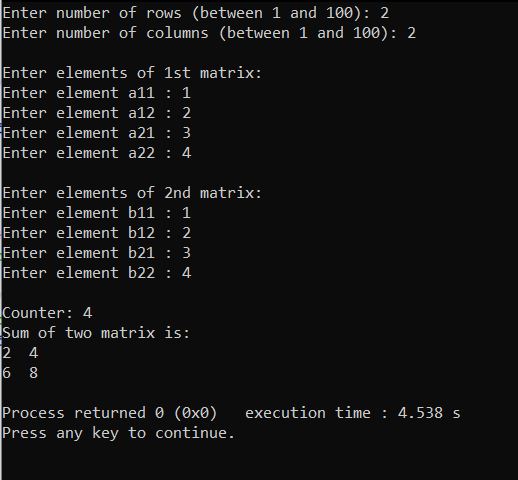
cout << endl;

}

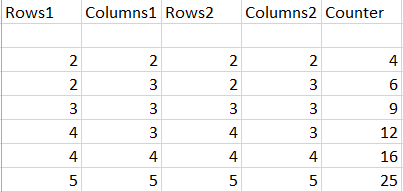
return 0;

}

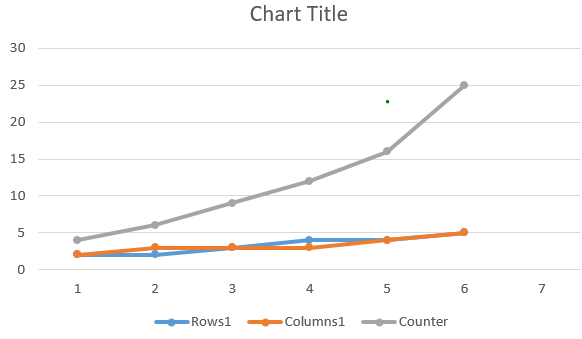
**OUTPUT:-**

****

**ANALYSIS TABLE:-**



**GRAPH:-**



**CONCLUSION:-**

From this practical, I learnt how to code matrix addition algorithm and find its count it requires to find the solution. I was also able to do analysis and develop a graph for the same.

**(2) Matrix Multiplication**

**PROGRAM:-**

#include <iostream>

using namespace std;

int main()

{

int a[10][10], b[10][10], mult[10][10], r1, c1, r2, c2, i, j, k, counter = 0;

cout << "Enter rows and columns for first matrix: ";

cin >> r1 >> c1;

cout << "Enter rows and columns for second matrix: ";

cin >> r2 >> c2;

// If column of first matrix in not equal to row of second matrix,

// ask the user to enter the size of matrix again.

while (c1!=r2)

{

cout << "Error! column of first matrix not equal to row of second.";

cout << "Enter rows and columns for first matrix: ";

cin >> r1 >> c1;

cout << "Enter rows and columns for second matrix: ";

cin >> r2 >> c2;

}

// Storing elements of first matrix.

cout << endl << "Enter elements of matrix 1:" << endl;

for(i = 0; i < r1; ++i)

for(j = 0; j < c1; ++j)

{

cout << "Enter element a" << i + 1 << j + 1 << " : ";

cin >> a[i][j];

}

// Storing elements of second matrix.

cout << endl << "Enter elements of matrix 2:" << endl;

for(i = 0; i < r2; ++i)

for(j = 0; j < c2; ++j)

{

cout << "Enter element b" << i + 1 << j + 1 << " : ";

cin >> b[i][j];

}

// Initializing elements of matrix mult to 0.

for(i = 0; i < r1; ++i)

for(j = 0; j < c2; ++j)

{

mult[i][j]=0;

}

// Multiplying matrix a and b and storing in array mult.

for(i = 0; i < r1; ++i){

for(j = 0; j < c2; ++j){

for(k = 0; k < c1; ++k){

mult[i][j] += a[i][k] \* b[k][j];

counter++;

}

}

}

cout<<"\nCounter:"<<counter;

// Displaying the multiplication of two matrix.

cout << endl << "Output Matrix: " << endl;

for(i = 0; i < r1; ++i)

for(j = 0; j < c2; ++j)

{

cout << " " << mult[i][j];

if(j == c2-1)

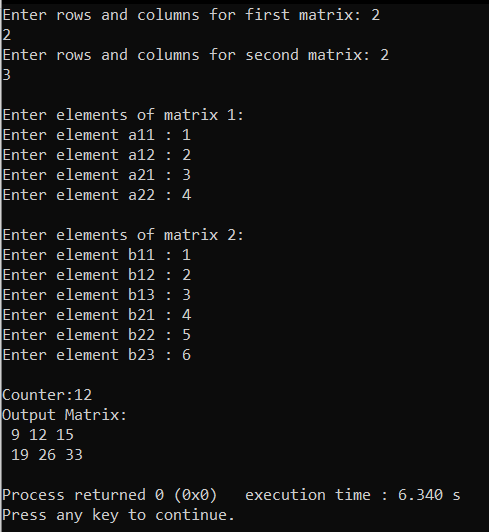
cout << endl;

}

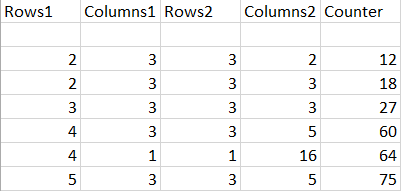
return 0;

}

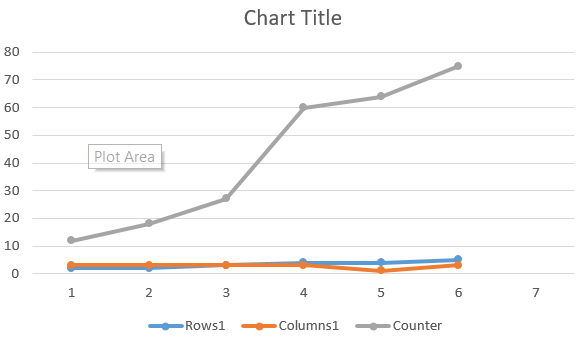
**OUTPUT:-**



**ANALYSIS TABLE:-**



**GRAPH:-**



**CONCLUSION:-**

From this practical, I learnt how to code matrix multiplication algorithm and find its count it requires to find the solution. I was also able to do analysis and develop a graph for the same.

**1.4 (1) Recursive Linear Search**

**PROGRAM:-**

#include<iostream>

using namespace std;

int counter=0;

int RecursiveLS(int arr[], int value, int index, int n)

{

counter++;

int pos = 0;

if(index >= n)

return 0;

else if (arr[index] == value)

{

pos = index + 1;

return pos;

}

else

return RecursiveLS(arr, value, index+1, n);

return pos;

}

int main()

{

int n, value, pos, m = 0, arr[100];

cout<<"\nEnter the total elements in the array : ";

cin>>n;

cout<<"\nEnter the array elements : \n";

for (int i = 0; i < n; i++)

cin>>arr[i];

cout<<"\nEnter the element to search :";

cin>>value;

pos = RecursiveLS(arr, value, 0, n);

if (pos != 0)

cout<<"\nElement found at pos "<< pos;

else

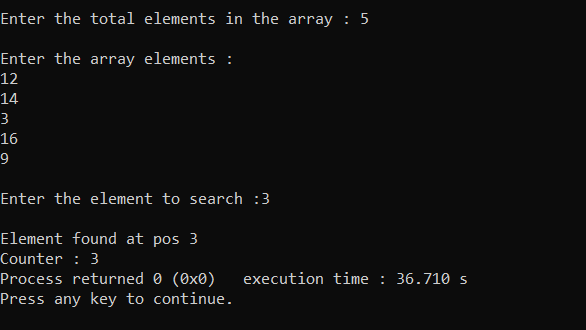
cout<<"\nElement not found";

cout<<"\nCounter : "<<counter;

return 0;

}

**OUTPUT:-**



**(Best Case)**

**ANALYSIS TABLE:**

|  |  |  |
| --- | --- | --- |
| **Total Elements** | **Pos of element to be searched** | **Counter** |
| 5 | 1 | 1 |
| 6 | 1 | 1 |
| 10 | 1 | 1 |
| 15 | 1 | 1 |
| 17 | 1 | 1 |

**GRAPH:**

**(Worst case)**

**ANALYSIS TABLE:**

|  |  |  |
| --- | --- | --- |
| **Total Elements** | **Pos of element to be searched** | **Counter** |
| 5 | 5 | 5 |
| 7 | 7 | 7 |
| 13 | 13 | 13 |
| 20 | 20 | 20 |
| 27 | 27 | 27 |

**GRAPH:**

**1.4 (2) Binary Search**

**PROGRAM:-**

#include<iostream>

using namespace std;

int counter=0;

int binarySearch(int A[], int low, int high, int x)

{

counter++;

if (low > high)

return -1;

int mid = (low + high)/2;

// int mid = low + (high - low)/2;

if (x == A[mid])

return mid;

else if (x < A[mid])

return binarySearch(A, low, mid - 1, x);

else

return binarySearch(A, mid + 1, high, x);

}

int main()

{

int nums,target;

cout<<"\nEnter the total elements in the array : ";

cin>>nums;

int arr[nums];

cout<<"\nEnter the array elements : \n";

for (int i = 0; i < nums; i++)

cin>>arr[i];

cout<<"\nEnter the element to search ";

cin>>target;

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

int low = 0, high = n - 1;

int index = binarySearch(arr, low, high, target);

if (index != -1)

cout<<"\nElement found at index "<<index;

else

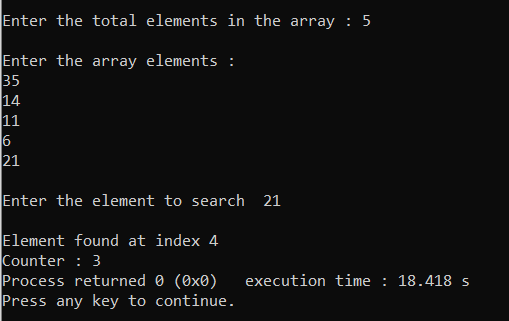
cout<<"\nElement not found in the array";

cout<<"\nCounter : "<<counter;

return 0;

}

**OUTPUT:-**



**(Best Case)**

**ANALYSIS TABLE:**

|  |  |  |
| --- | --- | --- |
| **Total Elements** | **Pos of element to be searched** | **Counter** |
| 5 | 3 | 1 |
| 7 | 4 | 1 |
| 12 | 6 | 1 |
| 20 | 10 | 1 |
| 27 | 13 | 1 |

**GRAPH:**

**(Worst Case)**

**ANALYSIS TABLE:**

|  |  |  |
| --- | --- | --- |
| **Total Elements** | **Pos of element to be searched** | **Counter** |
| 10 | 1 | 3 |
| 20 | 20 | 5 |
| 35 | 35 | 6 |
| 200 | 200 | 8 |
| 5555 | 1 | 12 |

**GRAPH:**

**CONCLUSION:**

Here we concluded by implementing and analyzing the program of linear and binary search the time complexity of linear search best case is O(1) and worst case is O(n) whereas in binary search the best case is O(1) and the worst case is also O(logn) by analyzing the graph of linear search in best case is constant and of worst case is linear whereas that of binary search the graph of best case is constant and of worst case is non-linear.

**1.5**

Find a subset of a given set S = {s1,s2,.....,sn} of n positive integers whose sum is equal to a given positive integer d. For example, if S= {1, 2, 5, 6, 8} and d = 9 there are two solutions {1,2,6} and {1,8}. A suitable message is to be displayed if the given problem instance doesn't have a solution.

**PROGRAM:**

#include<stdio.h>

#include<conio.h>

int s[10],d,n,set[10],count=0;

void Display(int);

int flag=0, Counter=0;

void main()

{

int subset(int,int);

int i;

printf("Enter the number of elements in set : \n");

scanf("%d",&n);

printf("Enter the set values : \n");

for(i=0; i<n; ++i)

scanf("%d",&s[i]);

printf("Enter the sum : \n");

scanf("%d",&d);

printf("The program output is : \n");

subset(0,0);

if(flag==0)

printf("there is no solution");

printf("\nCounter : %d",Counter);

getch();

}

int subset(int sum,int i)

{

if(sum==d)

{

flag=1;

Display(count);

return;

}

else if(sum>d||i>=n)

{

return;

}

else

{

set[count]=s[i];

count++;

subset(sum+s[i],i+1);

count--;

subset(sum,i+1);

}

Counter++;

}

void Display(int count)

{

int i;

printf("{");

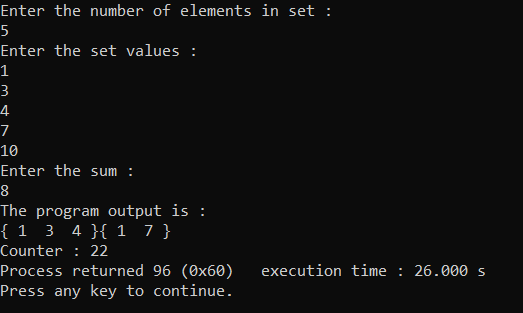
for(i=0; i<count; i++)

printf(" %d ",set[i]);

printf("}");

}

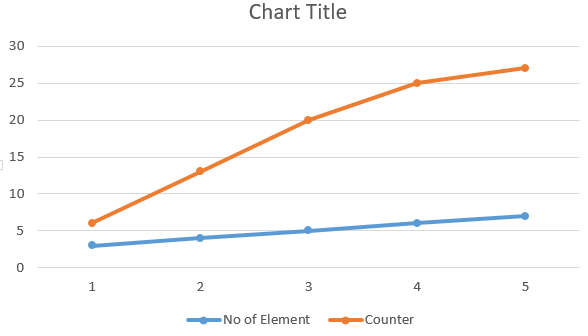
**OUTPUT:**



**ANALYSIS TABLE:**

|  |  |
| --- | --- |
| No of Element | Counter |
| 3 | 6 |
| 4 | 13 |
| 5 | 20 |
| 6 | 25 |
| 7 | 27 |

**GRAPH:**



**CONCLUSION:**

Here we concluded by implementing and analyzing the program of subset sum the time complexity of subset sum is O(2^n) and by analyzing the graph is non-linear.

Practical 2

|  |  |
| --- | --- |
| **AIM:** Implement and analyse algorithms given below:   |  | | --- | | **2.1) Bubble Sort** | |

**PROGRAM:**

#include<iostream>

using namespace std;

int main()

{

int siz,counter=0;

cout<<"Enter the size of array:- ";

cin>>siz;

int arr[siz];

cout<<"\n Enter the elements of array :- ";

for(int i=0; i<siz; ++i)

{

cin>>arr[i];

}

for(int i=0; i<siz-1; ++i)

{

int temp=0;

for(int j=0; j<siz-i-1; ++j)

{

if(arr[j]>arr[j+1])

{

counter++;

temp=arr[j];

arr[j]=arr[j+1];

arr[j+1]=temp;

}

}

}

cout<<"\n Sorted array is : - ";

for(int i=0; i<siz; ++i)

{

cout<<arr[i]<<" ";

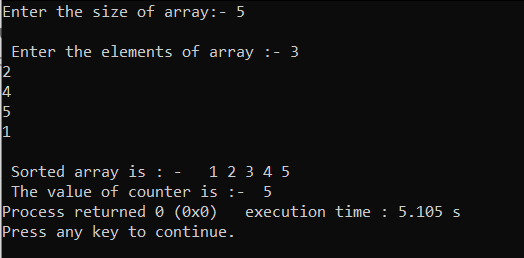
}

cout<<"\n The value of counter is :- "<<counter;

return 0;

}

**OUTPUT:**

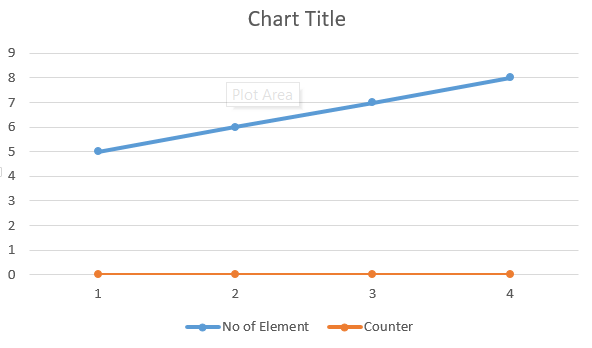


**1) Best Case:**

**ANALYSIS TABLE:**

|  |  |
| --- | --- |
| No of Element | Counter |
| 5 | 0 |
| 6 | 0 |
| 7 | 0 |
| 8 | 0 |

**GRAPH:**

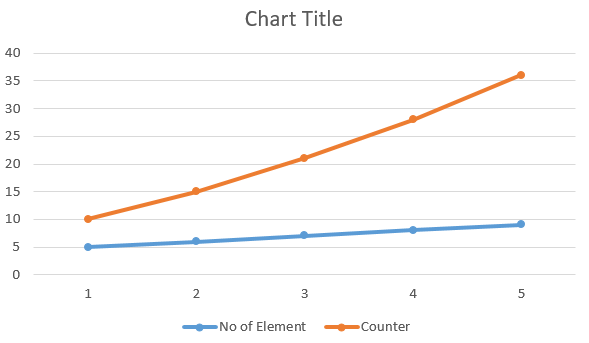


**2) Worst Case:**

**ANALYSIS TABLE:**

|  |  |
| --- | --- |
| No of Element | Counter |
| 5 | 10 |
| 6 | 15 |
| 7 | 21 |
| 8 | 28 |
| 9 | 36 |

**GRAPH:**



**CONCLUSION:**

Here we concluded by implementing and analyzing the program of bubble sort, the time complexity of algorithm by analyzing the graph is constant for best case i.e. 0 but for worst case it increases parabolically.

**2.2**

**PROGRAM:**

#include<iostream>

using namespace std;

int main(){

int n,i,minimum,j,temp,c = 0;

cout<<"Enter the length of array:";

cin>>n;

int arr[n];

cout<<"\nEnter the element of the array:";

for(i=0; i<n; i++){

cin>>arr[i];

}

for(i=0;i<=n-1;i++){

minimum = i;

for(j = i+1;j<=n;j++){

if(arr[j]<arr[minimum]){

minimum = j;

c++;

}

}

temp = arr[minimum];

arr[minimum] = arr[i];

arr[i] = temp;

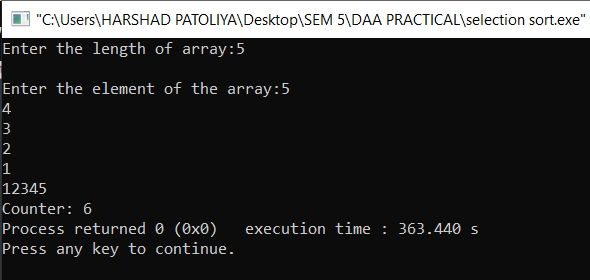
cout<<arr[i];

}

cout<< "\nCounter: "<< c;

}

**OUTPUT:**



**1) Best Case:**

**ANALYSIS TABLE:**

|  |  |
| --- | --- |
| No of Element | Counter |
| 4 | 0 |
| 5 | 0 |
| 6 | 0 |
| 7 | 0 |
| 8 | 0 |

**GRAPH:**

**2) Worst Case:**

**ANALYSIS TABLE:**

|  |  |
| --- | --- |
| No of Element | Counter |
| 3 | 2 |
| 4 | 4 |
| 5 | 6 |
| 6 | 9 |
| 7 | 12 |

**GRAPH:**

**CONCLUSION:**

Here I have learnt about analysis of selection Sort for the best case and worst case.

**2.3 Insertion Sort**

**PROGRAM:**

#include<iostream>

using namespace std;

int main(){

int n,i,key,j,c = 0;

cout<<"Enter the length of array:";

cin>>n;

int arr[n];

cout<<"\nEnter the element of the array:";

for(i=0; i<n; i++){

cin>>arr[i];

}

for (i = 1; i < n; i++){

key = arr[i];

j = i - 1;

while (j >= 0 && arr[j] > key){

arr[j + 1] = arr[j];

j = j - 1;

c++;

}

arr[j + 1] = key;

}

for (i = 0; i < n; i++){

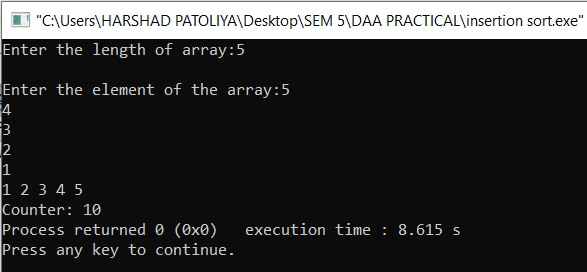
cout << arr[i] << " ";

}

cout<<"\nCounter: "<<c;

}

**OUTPUT:**



**1) Best Case:**

**ANALYSIS TABLE:**

|  |  |
| --- | --- |
| No of Element | Counter |
| 4 | 0 |
| 5 | 0 |
| 6 | 0 |
| 7 | 0 |
| 8 | 0 |

**GRAPH:**

**2) Worst Case:**

**ANALYSIS TABLE:**

|  |  |
| --- | --- |
| No of Element | Counter |
| 4 | 6 |
| 5 | 10 |
| 7 | 21 |
| 6 | 15 |
| 8 | 28 |

**GRAPH:**

**CONCLUSION:**

Here I have learnt about analysis of Insertion Sort for the best case and worst case.

**Practical 3.1**

**AIM:** Implement and perform analysis of worst case of Merge Sort and Quick sort. Compare

both algorithms.

**1) MERGE SORT:**

**PROGRAM:**

import java.util.\*;

class Mergesort

{

static int counter=0;

void merge(int arr[],int l,int r,int m)

{

int i=0,j=0,k=l,l1=m-l+1,l2=r-m;

int left[]=new int[l1];

int right[]=new int[l2];

for(int c=0;c<l1;c++){left[c]=arr[l+c];}

for(int c=0;c<l2;c++){right[c]=arr[m+1+c];}

while(i<l1 && j<l2)

{

counter++;

if(left[i]<=right[j])

{

arr[k]=left[i];

i++;

k++;

}

else

{

arr[k]=right[j];

j++;

k++;

}

}

while (i < l1)

{

counter++;

arr[k] = left[i];

i++;

k++;

}

while (j < l2)

{

counter++;

arr[k] = right[j];

j++;

k++;

}

}

void mergesrt(int arr[],int l,int r)

{

int m;

if(r==l)

return;

else

{

m=(l+r)/2;

mergesrt(arr,l,m);

mergesrt(arr,m+1,r);

merge(arr,l,r,m);

}

}

public static void main(String args[])

{

Mergesort m1=new Mergesort ();

int n,A[];

Scanner s=new Scanner(System.in);

System.out.print("Enter no of elements: ");

n=s.nextInt();

A=new int[n];

System.out.print("Enter the elements: ");

for(int i=0;i<n;i++){

A[i]=s.nextInt();

}

m1.mergesrt(A,0,n-1);

System.out.println("\nSorted Array is:");

for(int i=0;i<n;i++)

{

System.out.print(A[i]+" ");

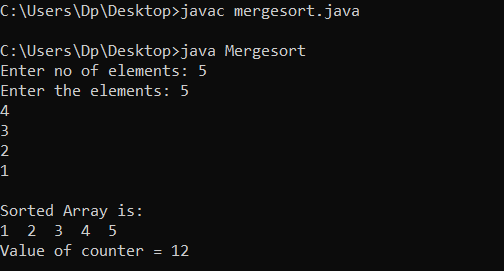
}

System.out.println("\nValue of counter = "+ Mergesort.counter);

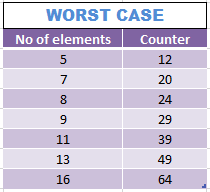
}

}

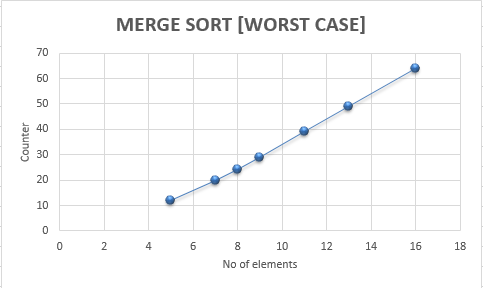
**OUTPUT:**



**ANALYSIS TABLE:**



**GRAPH:**



**2) QUICK SORT:**

**PROGRAM:**

import java.util.\*;

class Quicksort

{static int counter=0;

int partition(int arr[], int low, int high)

{

int pivot = arr[high];

int i = (low-1);

for (int j=low; j<high; j++)

{

counter++;

if (arr[j] < pivot)

{

i++;

int temp = arr[i];

arr[i] = arr[j];

arr[j] = temp;

}

}

int temp = arr[i+1];

arr[i+1] = arr[high];

arr[high] = temp;

return i+1;

}

void sort(int arr[], int low, int high)

{

if (low < high)

{

int p = partition(arr, low, high);

sort(arr, low, p-1);

sort(arr, p+1, high);

}

}

public static void main(String ae[])

{

int n,A[];

Scanner s=new Scanner(System.in);

System.out.print("Enter no of elements: ");

n=s.nextInt();

A=new int[n];

System.out.print("Enter the elements: ");

for(int i=0;i<n;i++){A[i]=s.nextInt();}

Quicksort a1=new Quicksort ();

a1.sort(A,0,n-1);

System.out.println("\nSorted array: ");

for(int i=0;i<n;i++)

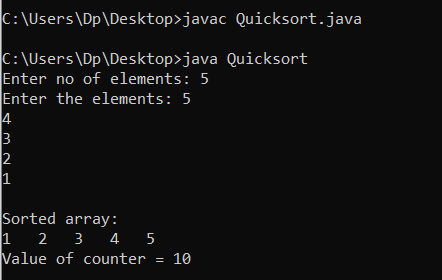
{System.out.print(A[i]+" "); }

System.out.println("\nValue of counter = "+Quicksort.counter);

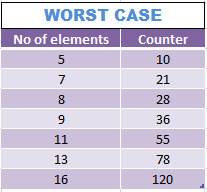
}

}

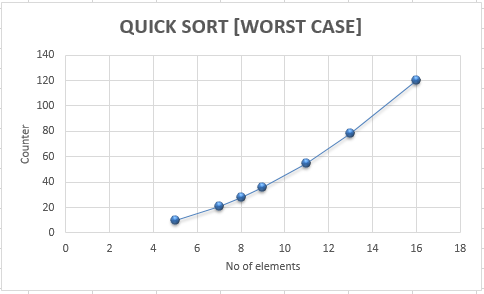
**OUTPUT:**



**ANALYSIS TABLE:**



**GRAPH:**



**CONCLUSION:**

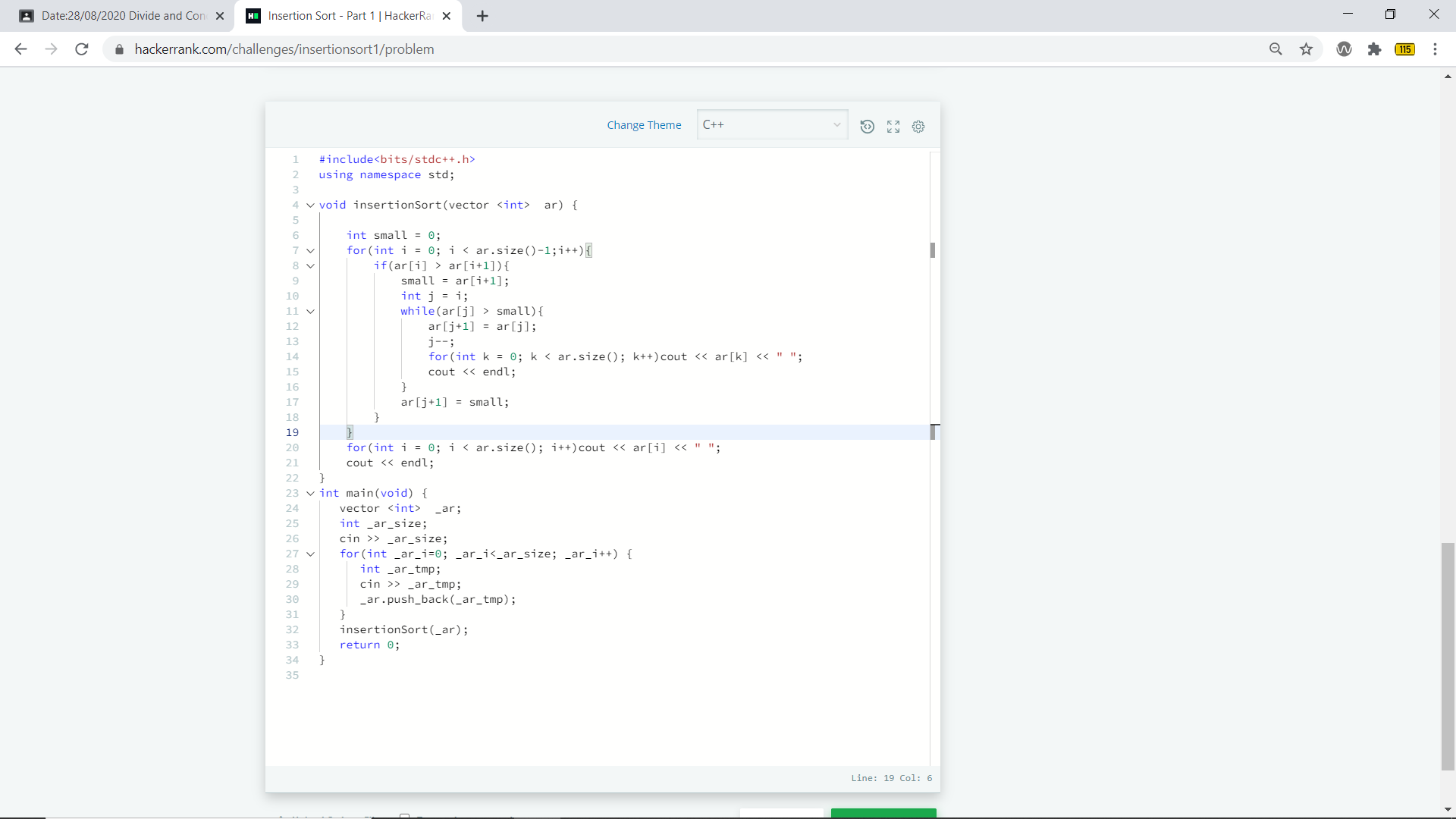
From this practical we analyzed the worst case of **Merge sort** and **Quick sort** algorithm. The graph was non-linear and counter is increasing with the number of elements for both the algorithms.

**Hacker Rank : INSERTION SORT**

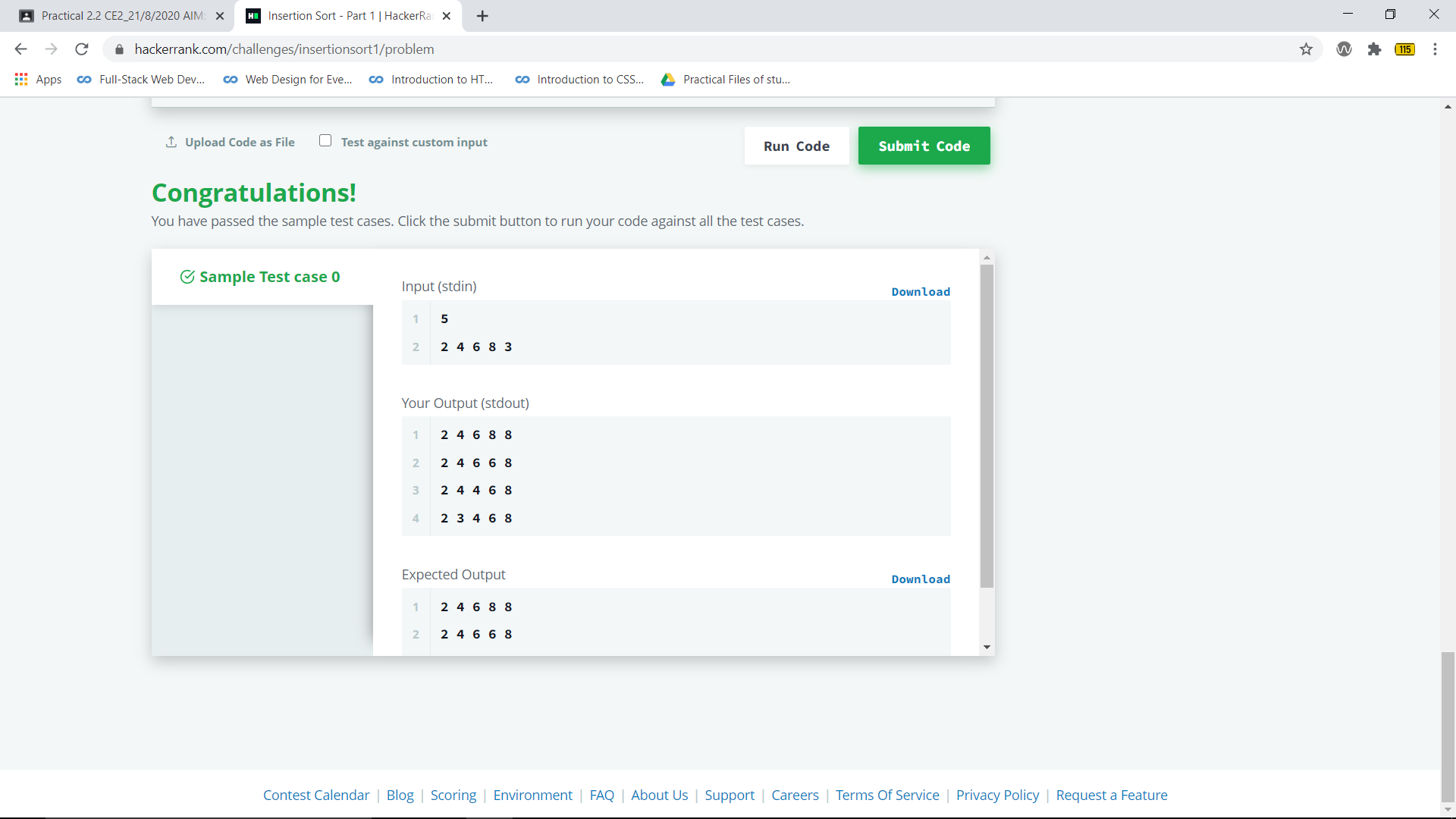
**PART –I**

**AIM:** Given a sorted list with an unsorted number in the rightmost cell, can you write some simple code to insert into the array so that it remains sorted.

**PROGRAM:**



**OUTPUT:**

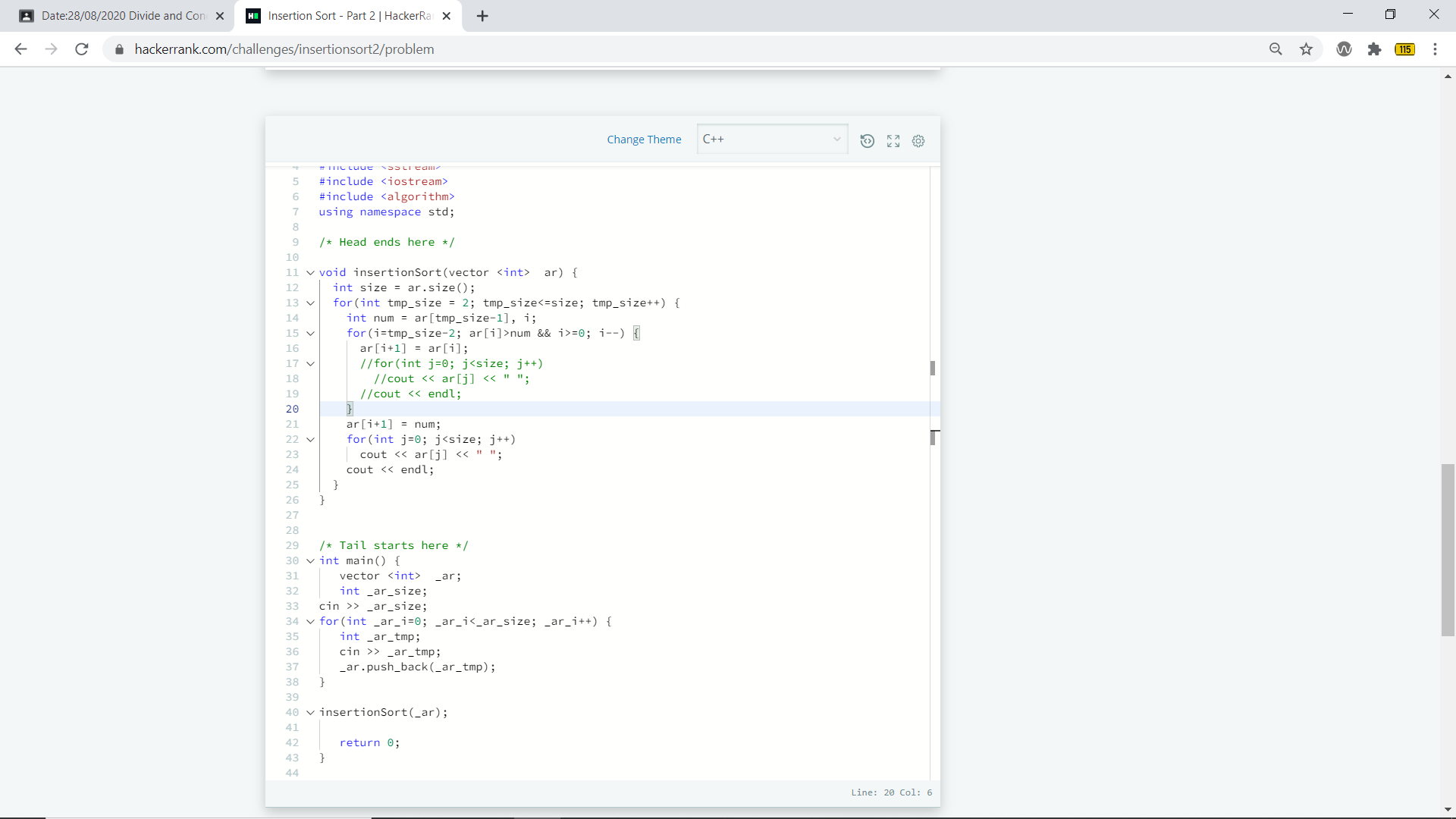


**Hacker Rank : INSERTION SORT**

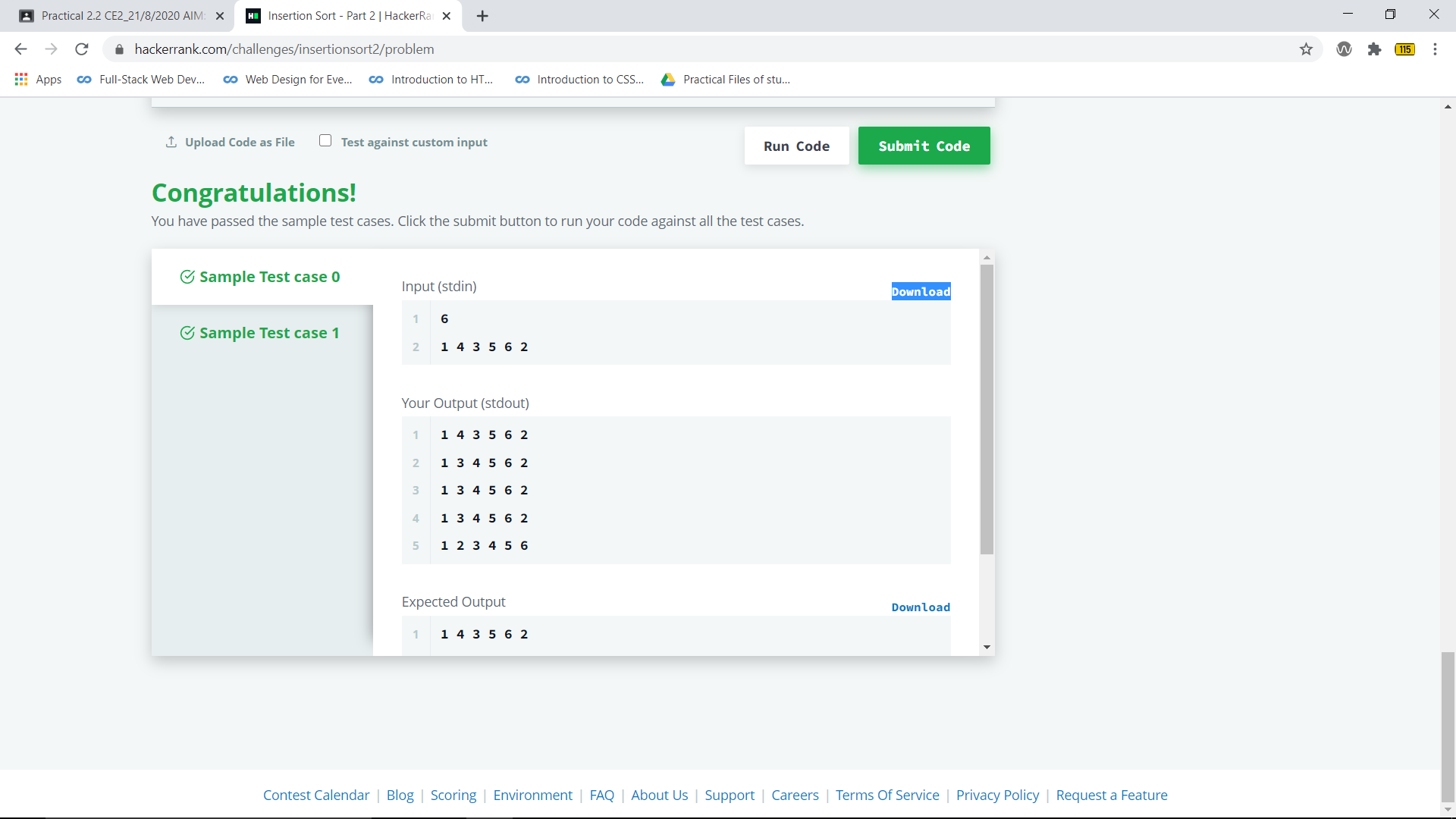
**PART –II**

**AIM:** In Insertion Sort Part 1, you inserted one element into an array at its correct sorted position. Using the same approach repeatedly, can you sort an entire array?

**PROGRAM:**



**OUTPUT:**



**Practical 3.2**

**AIM:** Implement the program to find X^Y using divide and conquer strategy and print the total number of multiplications required to find X^Y. Test the program for following test cases:

|  |  |  |
| --- | --- | --- |
| **Test** | **X2** | **Y2** |
| 1 | 2 | 6 |
| 2 | 7 | 25 |
| 3 | 5 | 34 |

**PROGRAM:**

#include<iostream>

using namespace std;

int counter=0;

long int power(unsigned long long int x,unsigned long long int y)

{

counter++;

if(y==0)

return 1;

else if(y%2==0)

return power(x,y/2)\*power(x,y/2);

else

return x\*power(x,y/2)\*power(x,y/2);

}

int main()

{

unsigned long long int x,y,result;

cout<<"X to the power Y....!!\n";

cout<<"\nEnter the value of x : ";

cin>>x;

cout<<"Enter the value of y : ";

cin>>y;

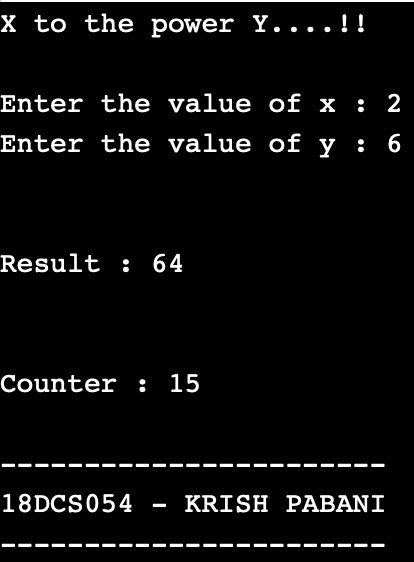
result=power(x,y);

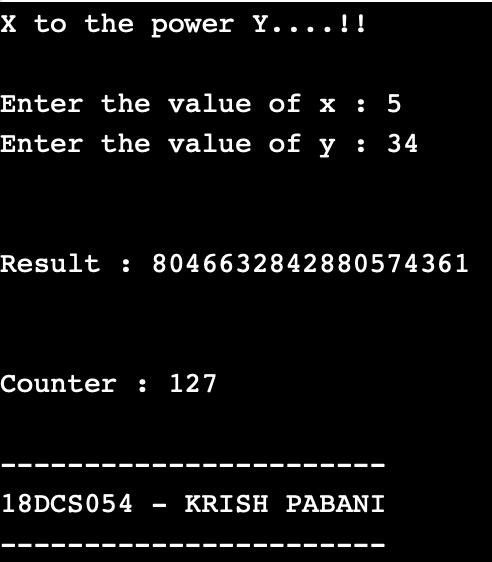
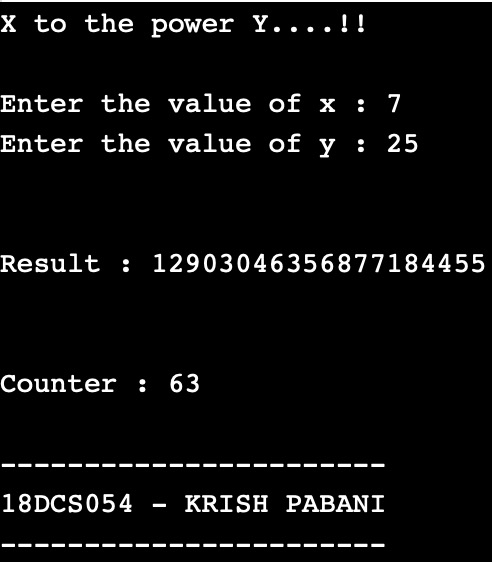
cout<<"\n\nResult : "<<result<<endl;

cout<<"\n\nCounter : "<<counter<<endl;

return 0;

}

**OUTPUT:**

****

**CONCLUSION:**

In this practical we have learnt the basics of divide and conquer strategy. We have also tried to optimize the power of element problem using divide and conquer strategy.

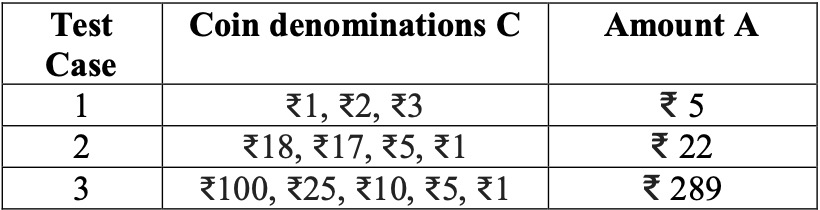
**Practical: 4**

**Greedy Approach**

**4.1**

**AIM:** A cashier at any mall needs to give change of an amount to customers many times in a day. Cashier has multiple number of coins available with different denominations which is described by a set C. Implement the program for a cashier to find the minimum number of coins required to find a change of a particular amount A. Output should be the total number of coins required of given denominations.

Check the program for following test cases:



**PROGRAM:**

#include <bits/stdc++.h>

using namespace std;

// All denominations of I ndian Currency

int deno[] = {1, 5, 10, 25, 100};

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

// Driver program

void findMin(int V)

{

// Initialize result

vector<int> ans;

// Traverse through all denomination

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

{// Find denominations

while (V >= deno[i])

{

V -= deno[i];

ans.push\_back(deno[i]);

}

}

// Print result

for (int i = 0; i < ans.size(); i++)

{

cout << ans[i] << " ";

}

}

// Driver program

int main()

{

int m ;

cout<<"enter the amount:";

cin>>m;

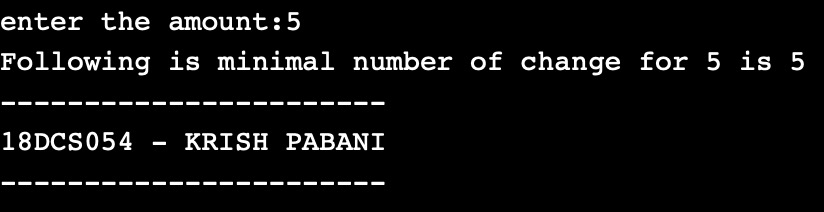
cout << "Following is minimal number of change for " << m << " is ";

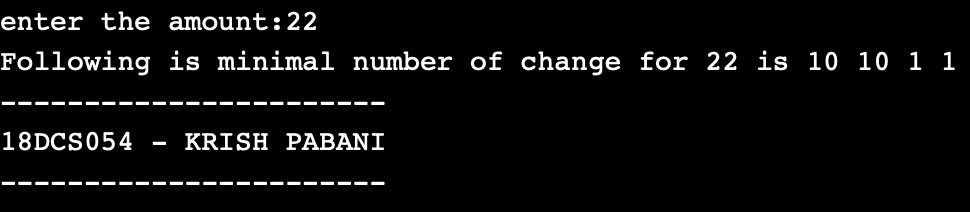
findMin(m);

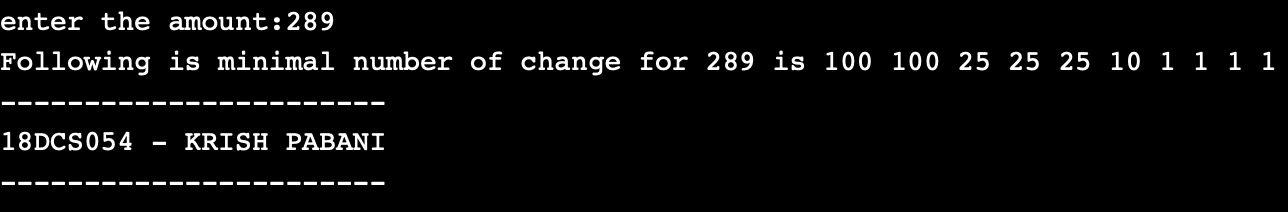
return 0;

}

**OUTPUT:**

****

****

****

**OBSERVATION:**

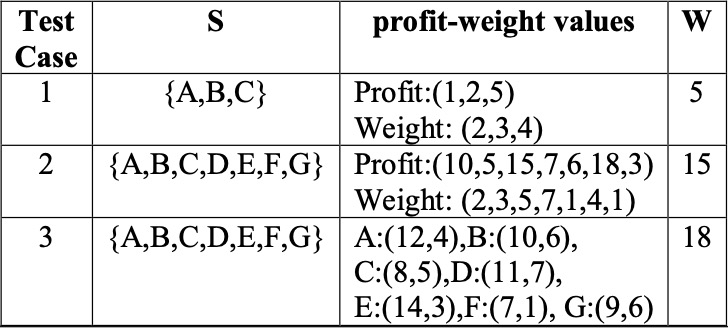
From the observation we can say that the output no 2 is not optimal because the output for change of 22rs is one 18rs coin and four 1rs coin. Rather the one 17rs coin and one 5rs is more optimal because the total number of coins are reduced to one.

**CONCLUSION:**

This problem is a variation of 'Coin Change Problem'. Worst case: When the only coin present is Rs. 1 Coin. So in that case, time complexity becomes O (A) where A is the amount to be paid.

**4.2**

**AIM:** Let S be a collection of objects with profit-weight values. Implement the fractional Knap sack problem for S assuming we have a sack that can hold objects with total weight W. Check the program for following test cases:



**PROGRAM:**

#include<iostream>

#include<math.h>

#include <bits/stdc++.h>

using namespace std;

int main(){

int n, weight;

cout << "enter the number of element : ";

cin >> n ;

float p[n],w[n];

float pw[n];

cout << "enter the Profit for all " << n << " elements." <<endl;

for(int i=0;i<n;i++){

cout << "enter the profit of "<< i+1 <<"th element : " ;

cin >> p[i];

cout << "enter the weight of "<< i+1 <<"th element : " ;

cin >> w[i];

pw[i] = p[i]/w[i];

}

cout<<endl << "Enter the maximum weight : ";

cin >> weight;

for(int i=0;i<n;i++){

for(int j=0;j<n-1;j++){

if(pw[j] < pw[j+1]){

float temp = pw[j];

pw[j] = pw[j+1];

pw[j+1] = temp;

temp = p[j];

p[j] = p[j+1];

p[j+1] = temp;

temp = w[j];

w[j] = w[j+1];

w[j+1] = temp;

}

}

}

cout <<endl<< "Considering weight of : " <<endl;

float ans\_p =0;

int ans = 0, i = 0;

while((ans+w[i]) <= weight){

ans = ans + w[i];

ans\_p = ans\_p + p[i];

cout << w[i]<< " " << p[i] << " "<< pw[i] <<" "<<endl;

i++;

}

if(ans <= weight){

float temp1 = weight-ans;

float temp2 = temp1/w[i];

ans\_p = ans\_p + (p[i]\*temp2);

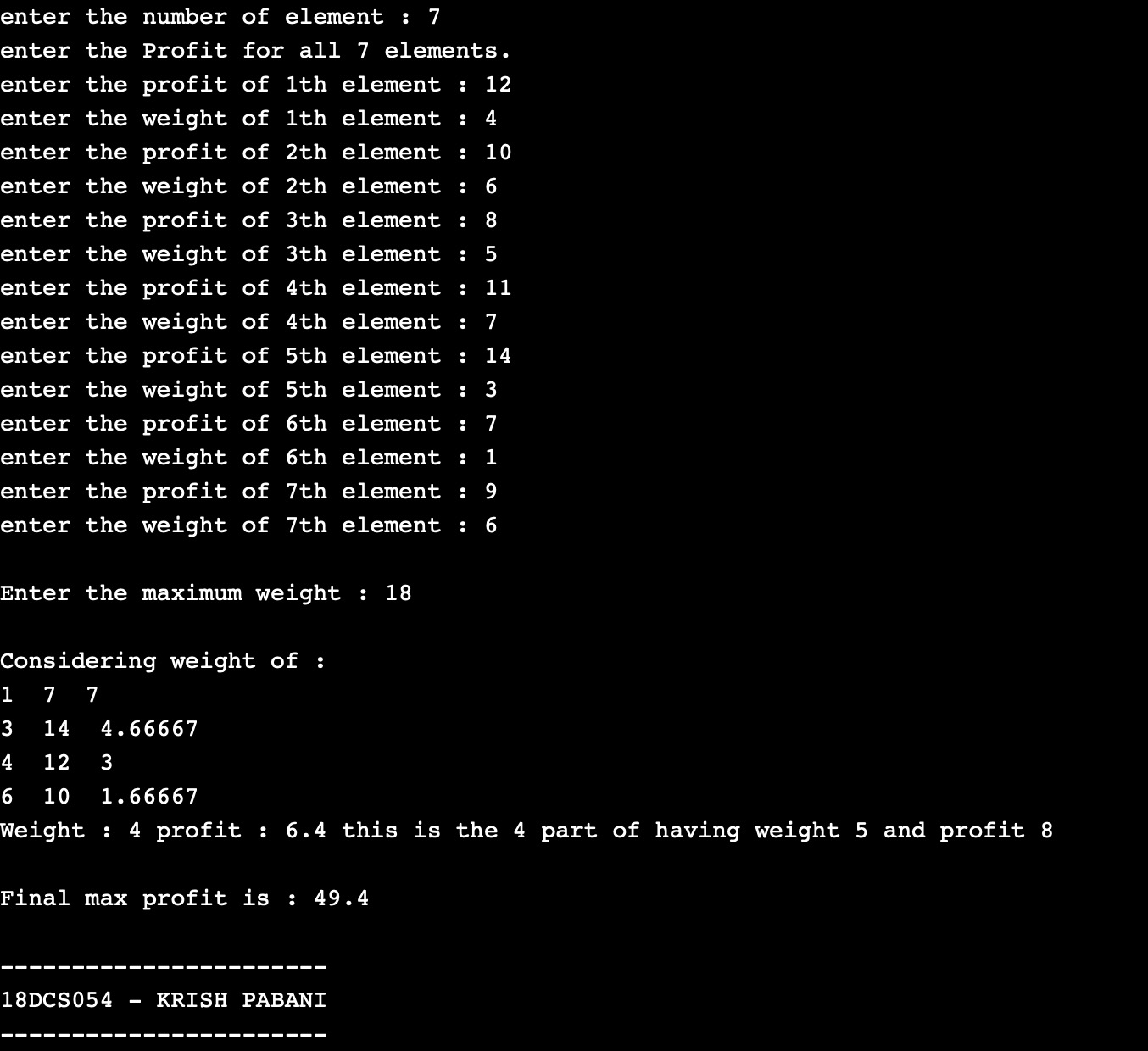
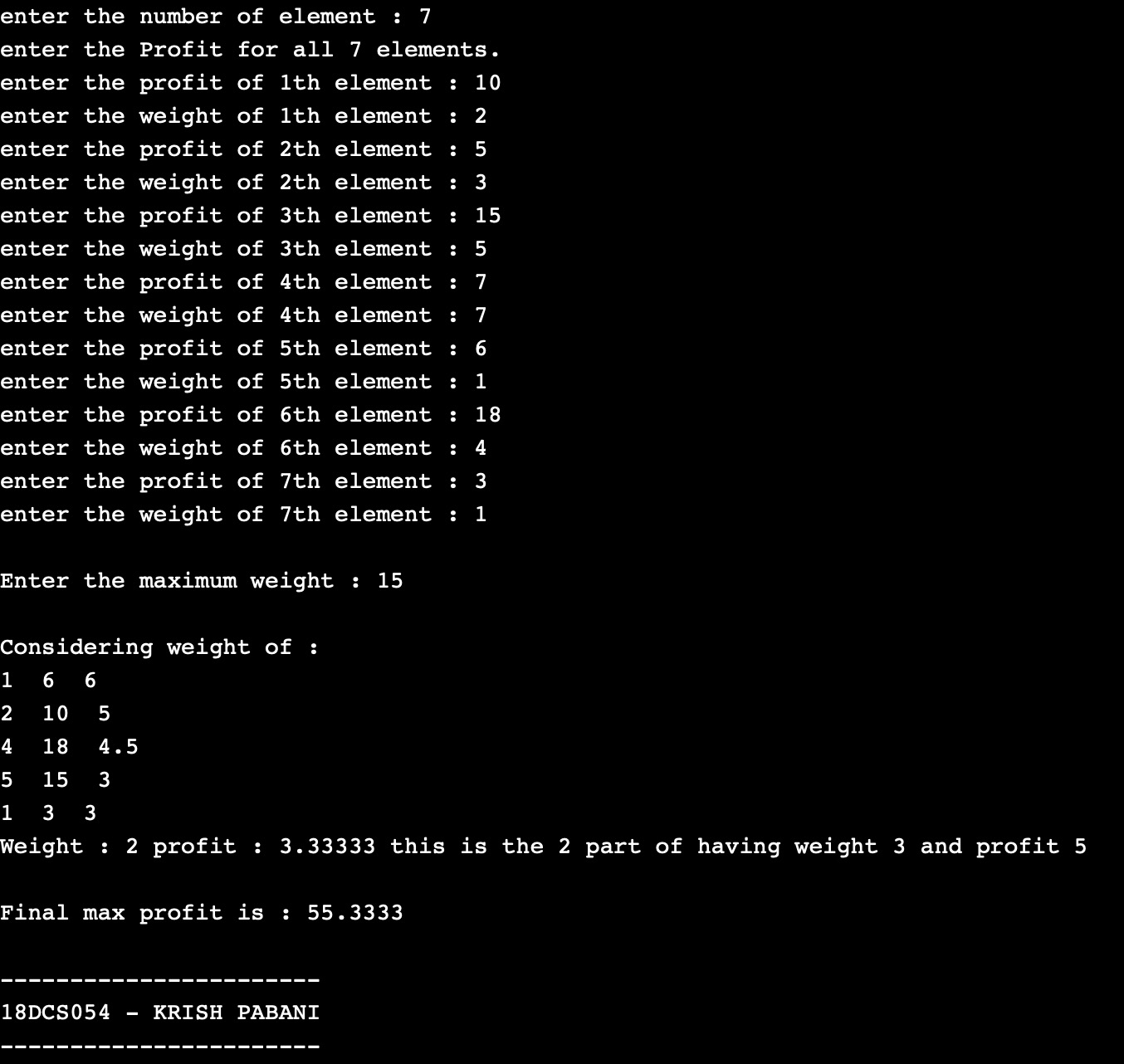
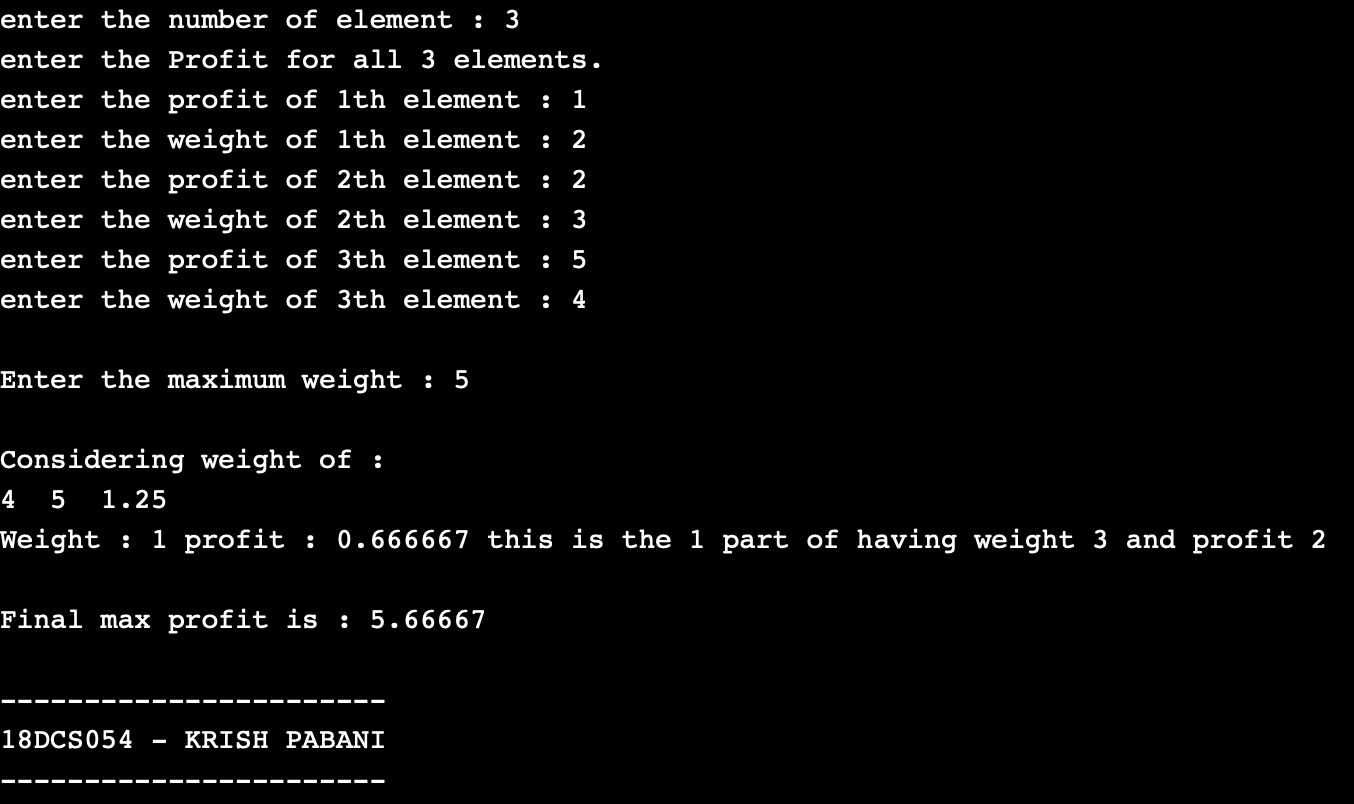
cout <<"Weight : "<< temp1 << " profit : "<< (p[i]\*temp2) <<" this is the " <<temp1<<" part of having weight " <<w[i]<<" and profit " << p[i]<<endl;

}

cout<<endl<< "Final max profit is : "<<ans\_p<<endl;

}

**OUTPUT:**

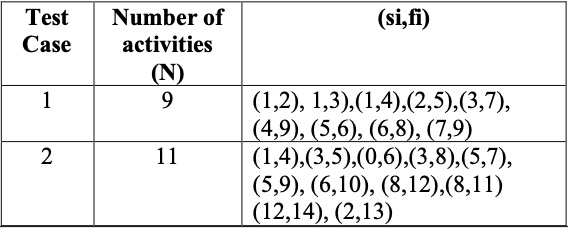
****

**CONCLUSION:**

In this practical we implemented the concept of knapsack problem.

**4.3**

**AIM:** Suppose you want to schedule N activities in a Seminar Hall. Start time and Finish time of activities are given by pair of (si,fi) for ith activity. Implement the program to maximize the utilization of Seminar Hall. (Maximum activities should be selected.)



**PROGRAM:**

#include <iostream>

#include<math.h>

#include <bits/stdc++.h>

using namespace std;

struct Activity

{

int start, finish;

};

// A utility function that is used for sorting

bool activityCompare(Activity s1, Activity s2)

{

return (s1.finish < s2.finish);

}

void printMaxActivities(Activity arr[], int n)

{

sort(arr, arr+n, activityCompare);

cout << "Following activities are selected: ";

int i = 0;

cout << "(" << arr[i].start << ", " << arr[i].finish << ") ";

for (int j = 1; j < n; j++)

{

if (arr[j].start >= arr[i].finish)

{

cout << "(" << arr[j].start << ", "

<< arr[j].finish << ") ";

i = j;

}

}

}

int main()

{

Activity arr[] = {{1,2}, {1,3}, {1,4}, {2,5}, {3,7}, {4,9}, {5,6}, {6,8}, {7,9}};

// Activity arr[] = {{1,4},{3,5},{0,6},{3,8},{5,7},{5,9}, {6,10}, {8,12}, {8,11}, {12,14}, {2,13}};

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

cout <<"\nTotal number of activities: " << n <<"\n";

cout << "Values of activities are:";

for (int j = 0; j < n; j++)

{

cout << "(" << arr[j].start << ", "

<< arr[j].finish << ") ";

}

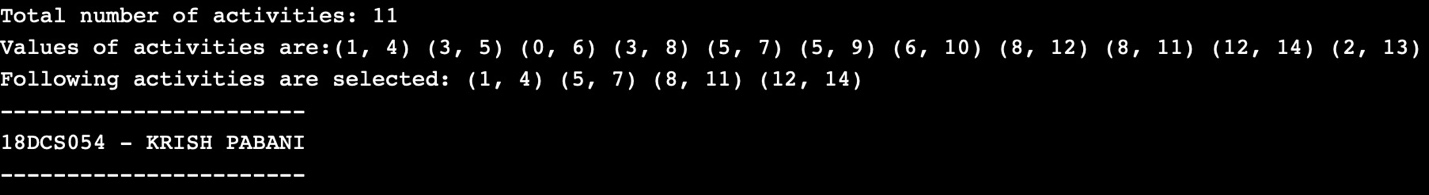
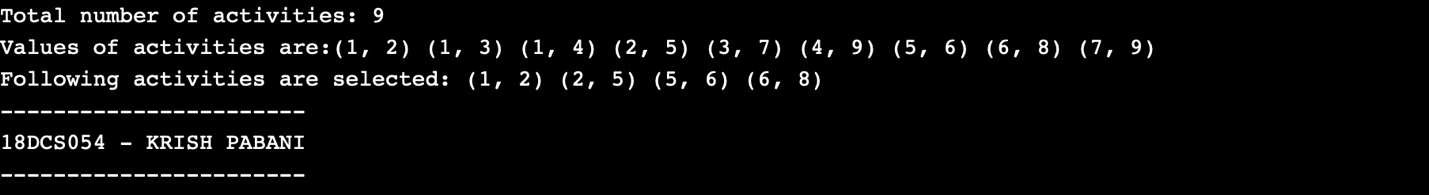
cout << "\n";

printMaxActivities(arr, n);

return 0;

}

**OUTPUT:**

****

**CONCLUSION:**

In this practical I have learnt and implemented the concept of Activity Selection Problem.

**Practical 5**

**Dynamic Programming**

**5.1**

**AIM:** Implement a program which has BNMCOEF() function that takes two parameters n and k and returns the value of Binomial Coefficient C(n, k). Compare the dynamic programming implementation with recursive implementation of BNMCOEF(). (In output, entire table should be displayed.)

|  |  |  |
| --- | --- | --- |
| **Test Case** | **n** | **k** |
| 1 | 5 | 2 |
| 2 | 11 | 6 |
| 3 | 12 | 5 |

**PROGRAM:**

#include<iostream>

using namespace std;

int binomialCoeff(int n, int k)

{

// Base Cases

if (k == 0 || k == n)

return 1;

// Recur

return binomialCoeff(n - 1, k - 1) +

binomialCoeff(n - 1, k);

}

int main()

{

int n,k;

cout<<"...BINOMIAL CO-EFFICIENT...\n";

cout<<"Enter the value of n : ";

cin>>n;

cout<<"Enter the value of k : ";

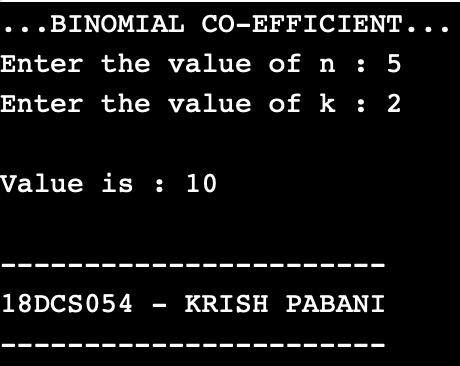
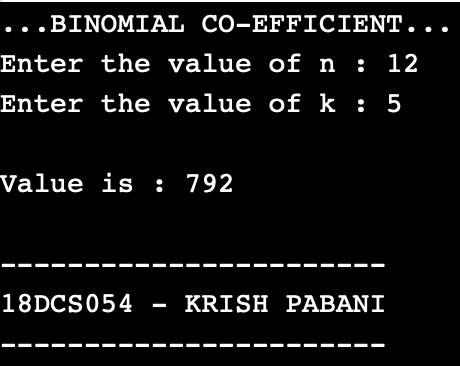
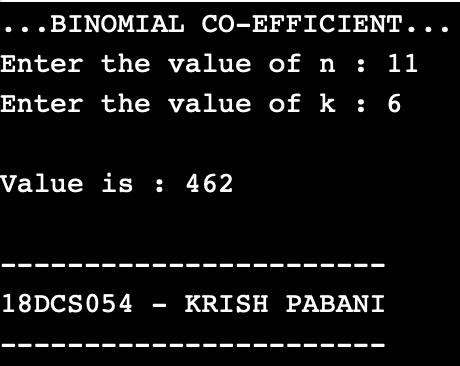
cin>>k;

cout<<"\nValue is : "<<binomialCoeff(n,k)<<endl;

return 0;

}

**OUTPUT:**

****

**CONCLUSION:** In this practical we implemented Binomial Coefficient using Dynamic Programming.

**5.2**

**AIM:** Implement the program 4.2 using Dynamic Programing. Compare Greedy and Dynamic approach.

**PROGRAM:**

#include<iostream>

using namespace std;

int maximum(int x,int y)

{

if(x>y)

return x;

else

return y;

}

int knapsack(int bag\_capacity,int weight[],int profit[],int number)

{

int matrix[number+1][bag\_capacity+1];

for(int i=0;i<number+1;i++)

for(int j=0;j<bag\_capacity+1;j++)

{

if(i==0 || j==0)

matrix[i][j]=0;

else if (j>=weight[i-1])

matrix[i][j]=maximum(matrix[i-1][j],profit[i-1]+matrix[i-1][j-weight[i-1]]);

else

matrix[i][j]=matrix[i-1][j];

}

return matrix[number][bag\_capacity];

}

int main()

{

int number,bag\_capacity;

cout<<".....BINARY KNAPSACK PROBLEM.....";

cout<<"\nEnter the size of arrays : ";

cin>>number;

int weight[number],profit[number];

cout<<"\nEnter the weights :";

for(int i=0;i<number;i++)

cin>>weight[i];

cout<<"Enter the profits :";

for(int i=0;i<number;i++)

cin>>profit[i];

cout<<"Enter bag capacity : ";

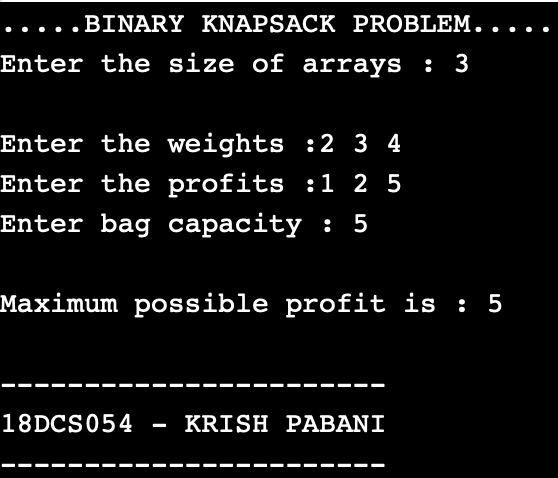
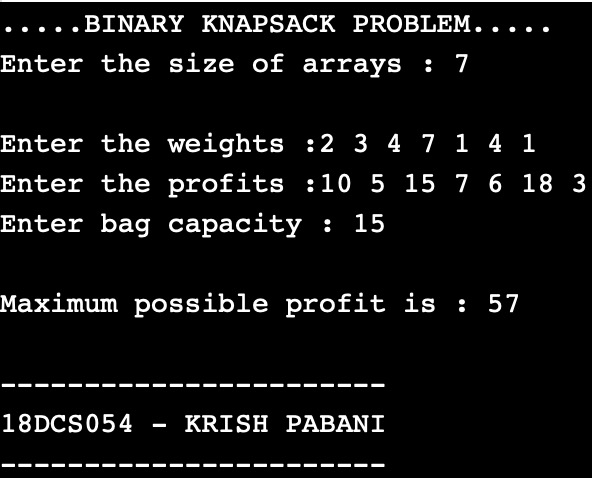
cin>>bag\_capacity;

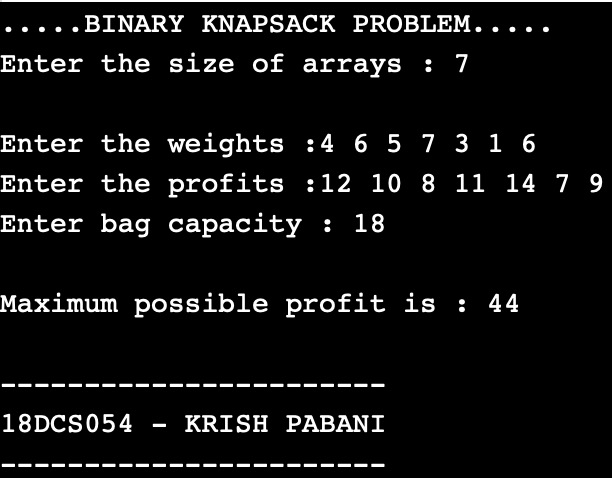
cout<<"\nMaximum possible profit is: " << knapsack(bag\_capacity,weight,profit,number) <<endl;

return 0;

}

**OUTPUT:**

** **

****

**CONCLUSION:**

In this practical I have learnt how to implement the 4.2 program using Dynamic Programing.

**5.3**

**AIM:** Given a chain < A1, A2,...,An> of n matrices, where for i=1,2,...,n matrix Ai with dimensions. Implement the program to fully parenthesize the product A1,A2,...,An in a way that minimizes the number of scalar multiplications. Also calculate the number of scalar multiplications for all possible combinations of matrices.

|  |  |  |
| --- | --- | --- |
| **Test Case** | **n** | **Matrices with dimensions** |
| 1 | 3 | A1:3\*5,A2:5\*6,A3:6\*4 |
| 2 | 6 | A1: 30\*35, A2: 35\*15, A3: 15\*5, A4: 5\*10, A5: 10\*20, A6: 20\*25 |

**PROGRAM:**

#include<bits/stdc++.h>

using namespace std;

int MatrixMultiplication(int p[], int n)

{

int m[n][n];

int i, j, k, L, q;

for (i = 1; i < n; i++)

m[i][i] = 0;

// L is chain length.

for (L = 2; L < n; L++)

{

for (i = 1; i < n - L + 1; i++)

{

j = i + L - 1;

m[i][j] = INT\_MAX;

for (k = i; k <= j - 1; k++)

{

// q = cost/scalar multiplications

q = m[i][k] + m[k + 1][j] +

p[i - 1] \* p[k] \* p[j];

if (q < m[i][j])

m[i][j] = q;

}

}

}

return m[1][n - 1];

}

int main()

{

int n;

cout<<"...MATRIX CHAIN MULTIPLICATION...\n\n";

cout<<"Enter total number of dimension values : ";

cin>>n;

int arr[n];

for(int i=0;i<n;i++)

{

cout<<"Enter P"<<i<<" : ";

cin>>arr[i];

}

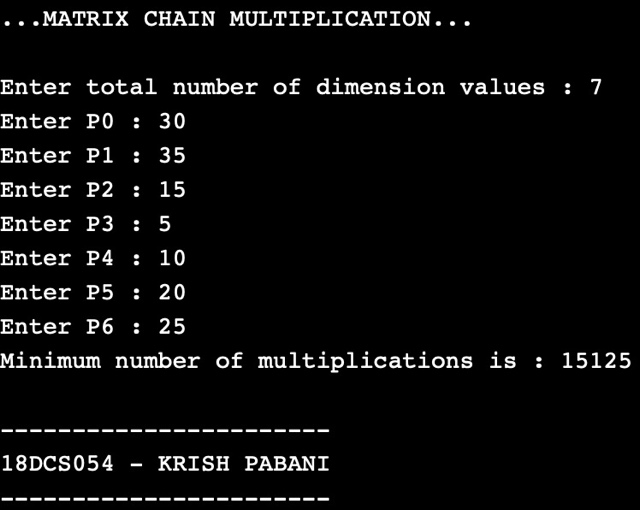
int length=sizeof(arr)/sizeof(arr[0]);

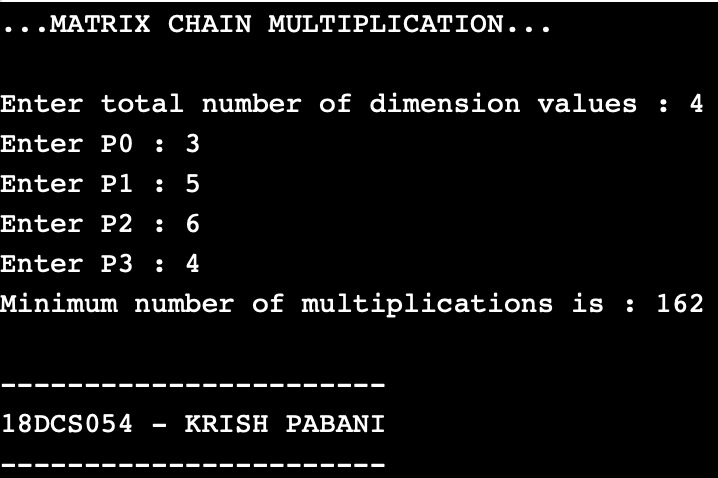
cout << "Minimum number of multiplications is : "<<MatrixMultiplication(arr,length)<<endl;

return 0;

}

**OUTPUT:**

****

****

**CONCLUSION:**

In this practical I have learnt how to implement scalar multiplications.

**5.4**

**AIM:** Implement a program to print the longest common subsequence for the following strings:

Check the program for following test cases:

|  |  |  |
| --- | --- | --- |
| **Test Case** | **String1** | **String2** |
| 1 | ABCDAB | BDCABA |
| 2 | EXPONENTIAL | POLYNOMIAL |
| 3 | LOGARITHM | ALGORITHM |

**PROGRAM:**

#include<iostream>

#include<string.h>

using namespace std;

int max(int a, int b);

int lcs( char \*X, char \*Y, int m, int n )

{ if (m == 0 || n == 0)

return 0;

if (X[m-1] == Y[n-1])

return 1 + lcs(X, Y, m-1, n-1);

else

return max(lcs(X, Y, m, n-1), lcs(X, Y, m-1, n));} int max(int a, int b)

{

return (a > b)? a : b;

}

int main()

{ cout<<"...LONGEST COMMON SUBSEQUENCE..."<<endl;

char X[100],Y[100];

cout<<"Enter 1st string sequence : ";cin>>X;

cout<<"Enter 2nd string sequence : ";cin>>Y;

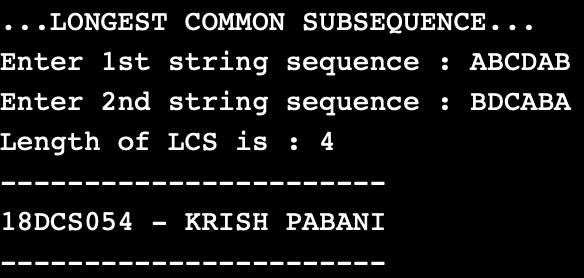
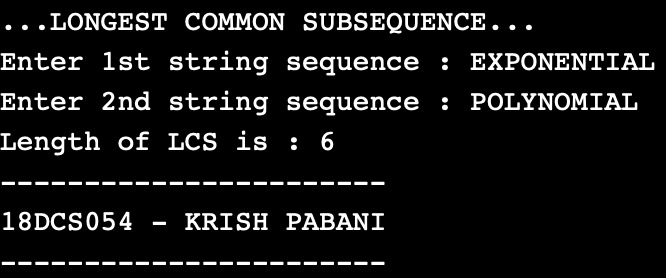
int m = strlen(X);int n = strlen(Y);

cout<<"Length of LCS is : "<<lcs(X,Y,m,n);

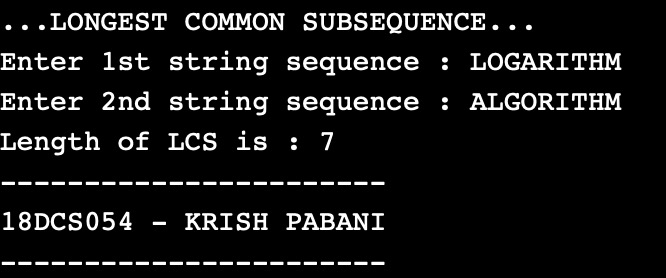
return 0;

}

**OUTPUT:**

****

**CONCLUSION:**

****

**CONCLUSION:**

* Time Complexity of the above implementation is O(mn) where m and n is length of string1 and string2.
* And, it is observed to be much better than the worst-case time complexity of Naive Recursive implementation.

**Practical 6: Graph**

**6.1**

**AIM:** Write a program to detect cycles in a directed graph.

**PROGRAM:**

#include<iostream>

using namespace std;

void DFS(int);

int G[20][20],visited[20],n;

bool flag=false;

main()

{

int i,j;

cout<<"Enter number of vertices : ";

cin>>n;

cout<<"\nEnteradjecency matrix of the graph \n";

for(i=0;i<n;i++)

{

cout<<"For "<<i<<" : ";

for(j=0;j<n;j++)

cin>>G[i][j];

}

for(i=0;i<n;i++)

visited[i]=0;

DFS(0);

if(flag)

cout<<"\n\nLoop is present in the graph.";

else

cout<<"\n\nLoop is not present in the graph.";

}

void DFS(int i)

{

int j;

if(visited[i]==1)

{

flag=true;

return;

}

visited[i]=1;

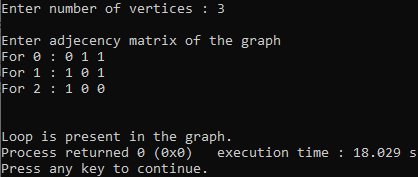
for(j=0;j<n;j++)

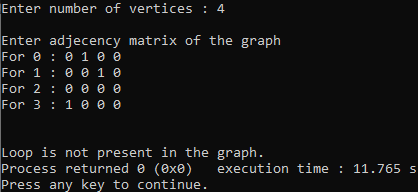
if(G[i][j]==1)

DFS(j);

}

**OUTPUT:**





**CONCLUSION:**

* Depth First Traversal can be used to detect a cycle in a Graph. DFS for a connected graph produces a tree.
* Time Complexity: O(V + E)

**6.2**

**AIM:** From a given vertex in a weighted graph, implement a program to find shortest paths to other vertices using Dijkstra’s algorithm.

**PROGRAM:**

#include<stdio.h>

#include<conio.h>

#define INFINITY 9999

#define MAX 10

void dijkstra(int G[MAX][MAX],int n,int startnode);

int main()

{ int G[MAX][MAX],i,j,n,u;

printf("Enter no. of vertices:");

scanf("%d",&n);

printf("\nEnter the adjacency matrix:\n");

for(i=0;i<n;i++)

for(j=0;j<n;j++)

scanf("%d",&G[i][j]);

printf("\nEnter the starting node:");

scanf("%d",&u);

dijkstra(G,n,u);

return 0;

}

void dijkstra(int G[MAX][MAX],int n,int startnode)

{ int cost[MAX][MAX],distance[MAX],pred[MAX];

int visited[MAX],count,mindistance,nextnode,i,j;

for(i=0;i<n;i++)

for(j=0;j<n;j++)

if(G[i][j]==0)

cost[i][j]=INFINITY;

else

cost[i][j]=G[i][j];

for(i=0;i<n;i++)

{ distance[i]=cost[startnode][i];

pred[i]=startnode;

visited[i]=0; }

distance[startnode]=0;

visited[startnode]=1;

count=1;

while(count<n-1)

{

mindistance=INFINITY;

for(i=0;i<n;i++)

if(distance[i]<mindistance&&!visited[i])

{

mindistance=distance[i];

nextnode=i;

}

visited[nextnode]=1;

for(i=0;i<n;i++)

if(!visited[i])

if(mindistance+cost[nextnode][i]<distance[i])

{ distance[i]=mindistance+cost[nextnode][i];

pred[i]=nextnode;

}

count++;

}

for(i=0;i<n;i++)

if(i!=startnode)

{ printf("\nDistance of node%d=%d",i,distance[i]);

printf("\nPath=%d",i);

j=i;

do

{ j=pred[j];

printf("<-%d",j);

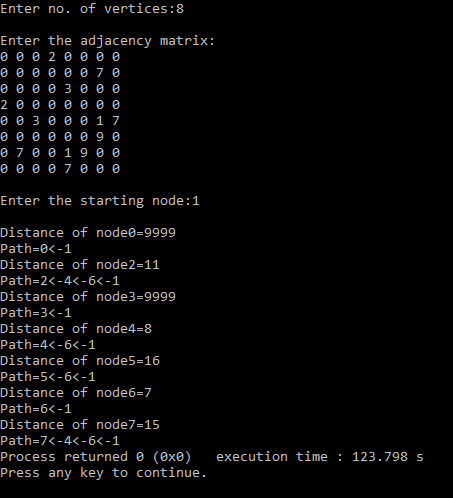
}while(j!=startnode);

}

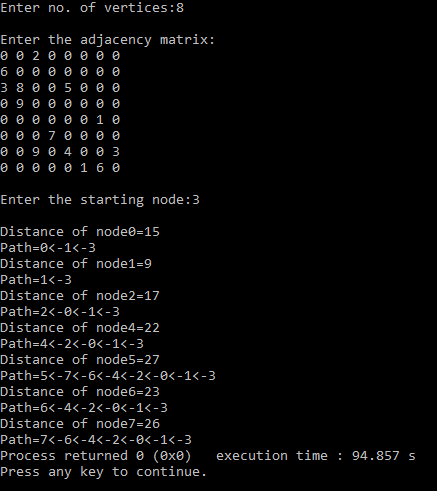
}

**OUTPUT:**

**1.**



**2.**



**CONCLUSION:**

* Dijkstra’s algorithm is very similar to [Prim’s algorithm for minimum spanning tree](https://www.geeksforgeeks.org/prims-minimum-spanning-tree-mst-greedy-algo-5/).
* Dijkstra's algorithm is an algorithm for finding the shortest paths between nodes in a graph, which may represent, for example, road networks.
* Time Complexity: O(ElogV)

**6.3**

**AIM:** Find Minimum Cost spanning tree of a given undirected graph using Prim’s algorithm.

**PROGRAM:**

#include<stdio.h>

#include<stdlib.h>

#define infinity 9999

#define MAX 20

int G[MAX][MAX],spanning[MAX][MAX],n;

int prims();

int main()

{ int i,j,total\_cost;

printf("Enter no. of vertices:");

scanf("%d",&n);

printf("\nEnter the adjacency matrix:\n");

for(i=0;i<n;i++)

for(j=0;j<n;j++)

scanf("%d",&G[i][j]);

total\_cost=prims();

printf("\nspanning tree matrix:\n");

for(i=0;i<n;i++)

{

printf("\n");

for(j=0;j<n;j++)

printf("%d\t",spanning[i][j]); }

printf("\n\nTotal cost of spanning tree=%d",total\_cost);

return 0; }

int prims()

{ int cost[MAX][MAX];

int u,v,min\_distance,distance[MAX],from[MAX];

int visited[MAX],no\_of\_edges,i,min\_cost,j;

for(i=0;i<n;i++)

for(j=0;j<n;j++)

{ if(G[i][j]==0)

cost[i][j]=infinity;

else

cost[i][j]=G[i][j];

spanning[i][j]=0;

}

distance[0]=0;

visited[0]=1;

for(i=1;i<n;i++)

{ distance[i]=cost[0][i];

from[i]=0;

visited[i]=0; }

min\_cost=0;

no\_of\_edges=n-1;

while(no\_of\_edges>0)

{ min\_distance=infinity;

for(i=1;i<n;i++)

if(visited[i]==0&&distance[i]<min\_distance)

{ v=i;

min\_distance=distance[i]; }

u=from[v];

spanning[u][v]=distance[v];

spanning[v][u]=distance[v];

no\_of\_edges--;

visited[v]=1;

for(i=1;i<n;i++)

if(visited[i]==0&&cost[i][v]<distance[i])

{ distance[i]=cost[i][v];

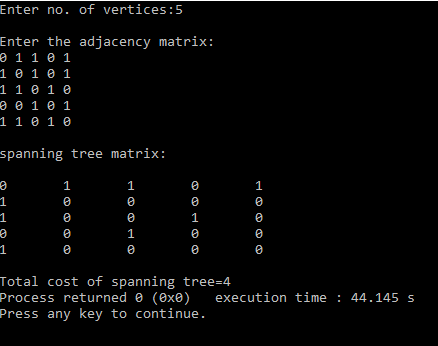
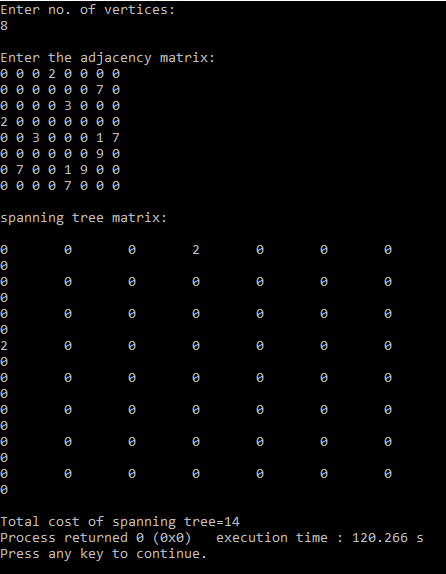
from[i]=v; }

min\_cost=min\_cost+cost[u][v]; }

return(min\_cost);

}

**OUTPUT:**



**CONCLUSION:**

* Prim's algorithm is very similar to Kruskal's: whereas Kruskal's "grows" a forest of trees, Prim's algorithm grows a single tree until it becomes the minimum spanning tree.
* Time Complexity: O(ElogV)

**Practical 7: Backtracking**

**7.1**

**AIM:** Implement a program to print all permutations of a given string.

**PROGRAM:**

#include <stdio.h>

#include <string.h>

void swap(char \*x, char \*y)

{

char temp;

temp = \*x;

\*x = \*y;

\*y = temp;

}

void permute(char \*a, int l, int r)

{

int i;

if (l == r)

printf("%s\n", a);

else

{

for (i = l; i <= r; i++)

{

swap((a+l), (a+i));

permute(a, l+1, r);

swap((a+l), (a+i)); //backtrack

} } }

int main()

{ char str[] = "NOTE";

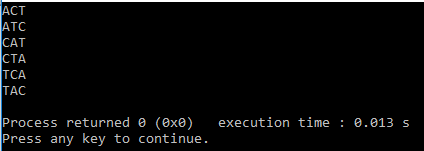
int n = strlen(str);

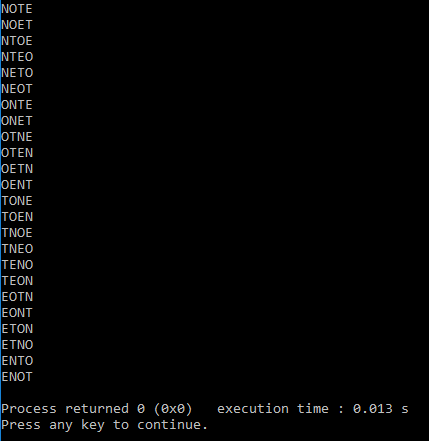
permute(str, 0, n-1);

return 0;

}

**OUTPUT:**





**CONCLUSION:**

* Backtracking is an algorithmic-technique for solving problems recursively by trying to build a solution incrementally, one piece at a time, removing those solutions that fail to satisfy the constraints of the problem at any point of time.
* A permutation, also called an “arrangement number” or “order,” is a rearrangement of the elements of an ordered list S into a one-to-one correspondence with S itself.
* **Time Complexity:**O(n\*n!).

**Practical 8: String Matching Algorithm**

**8.1**

**AIM:** Suppose you are given a source string S[0 ..n − 1] of length n, consisting of symbols a and b. Suppose that you are given a pattern string P[0 ..m − 1] of length m < n, consisting of symbols a, b, and \*, representing a pattern to be found in string S. The symbol \* is a “wild card” symbol, which matches a single symbol, either a or b. The other symbols must match exactly. The problem is to output a sorted list M of valid “match positions”, which are positions j in S such that pattern P matches the substring S [j..j + |P|− 1]. For example, if S = ababbab and P = ab\*, then the output M should be [0, 2]. Implement a straightforward, naive algorithm to solve the problem.

**PROGRAM:**

#include<stdio.h>

void main()

{ char t[100],p[100];

int tn,pn,shift[20]={0},s=0,i,j=0,count=0,m=0;

printf("\n Enter The Text : ");

scanf("%s",t);

fflush(stdin);

printf("\n Enter The Pattern : ");

scanf("%s",p);

tn = strlen(t);

pn = strlen(p);

while(s!=(tn-pn+1))

{ j=0;

for(i=s;i<pn+s;i++)

{ if(p[j]==t[i])

{ count++;

if(count==pn)

{ count=0;

shift[m]=s;

m++;

}

}

else

{ count=0;

break;

}

j++;

}

s++; }

if(m>0)

{ printf("\n\n Valid Shifts : ");

for(i=0;i<m;i++) printf("%d ",shift[i]);

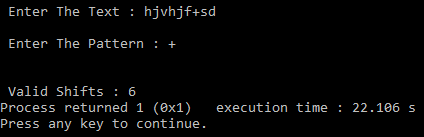
}

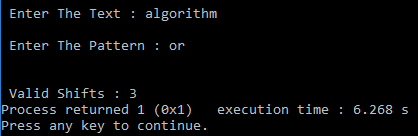
else

{ printf("\n\n No Valid Shifts."); }

}

**OUTPUT:**



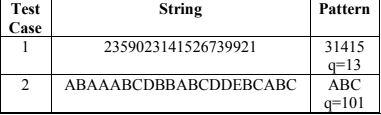


**CONCLUSION:**

* Naive pattern searching is the simplest method among other pattern searching algorithms.
* It checks for all character of the main string to the pattern.
* This algorithm is helpful for smaller texts. It does not need any pre-processing phases.
* The time complexity is O(m\*n). The m is the size of pattern and n is the size of the main string.

**8.2**

**AIM:** Implement Rabin karp algorithm and test it on the following test cases:



**PROGRAM:**

#include<stdio.h>

#include<string.h>

#define d 256

void search(char pat[], char txt[], int q)

{ int M = strlen(pat);

int N = strlen(txt);

int i, j;

int p = 0; // hash value for pattern

int t = 0; // hash value for txt

int h = 1;

for (i = 0; i < M-1; i++)

h = (h\*d)%q;

for (i = 0; i < M; i++)

{ p = (d\*p + pat[i])%q;

t = (d\*t + txt[i])%q;

}

for (i = 0; i <= N - M; i++)

{ if ( p == t )

{ for (j = 0; j < M; j++)

{ if (txt[i+j] != pat[j])

break; }

if (j == M)

printf("Pattern found at index %d \n", i);

}

if ( i < N-M )

{ t = (d\*(t - txt[i]\*h) + txt[i+M])%q;

if (t < 0)

t = (t + q); } } }

int main()

{ char txt[] = "ABAAABCDBBABCDDEBCABC";

char pat[] = "ABC";

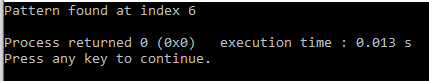
int q = 101; // A prime number

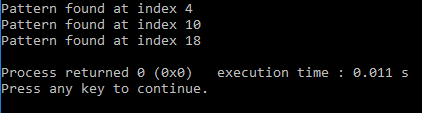
search(pat, txt, q);

return 0;

}

**OUTPUT:**





**CONCLUSION:**

* The Rabin–Karp algorithm is a form of [rolling hash](https://en.wikipedia.org/wiki/Rolling_hash) used in string searching [[1]](https://en.wikipedia.org/wiki/Rabin%E2%80%93Karp_algorithm#cite_note-1) to find any one of a set of pattern strings in a text.
* For text of length *n* and *p* patterns of combined length *m*, its average and best case time complexity is [O](https://en.wikipedia.org/wiki/Big-O_notation)(*n*+*m*) , but its worst-case time complexity is O(*nm*).