NETAJI SUBHAS UNIVERSITY OF TECHNOLOGY

A STATE UNIVERSITY UNDER DELHI ACT 06 OF 2018,GOVERNMENT OF NCT OF DELHI

AZAD HIND FAUZ MARG,SECTOR 3,DWARKA,NEW DELHI-11078

LABORATORY FILE

DESIGN AND ANALYSIS OF ALGORITHM

COCSC06

SUBMITTED BY:

NAME -> PRIKSHIT

ROLL NO. ->2021UCS1672

BRANCH -> CSE SECTION-3 ;SEMESTER 3

DECLARATION

I have done this assignment on my own. I have not copied any code from another student or any online source. I understand if my code is found similar to somebody else's code, my case can be sent to the Disciplinary Committee of the institute for appropriate action.

PRACTICAL 1:

Dr. Johnson has collected the blood pressure rate of a tuberculosis infected patient. He has stored the collected data for observing the blood pressure behaviour of a given patient. During the operation of the patient one of the doctors has asked Dr. Johnson to tell the maximum and minimum blood pressure of the patient. So, this is the routine work of Dr. Johnson. So, he wishes to have a divide conquer based approach which takes lesser comparison to achieve the desire results. Perform the following sorting algorithms-Merge sort, Quick sort, Bubble sort, insertion sort and selection sort.

QUICK SORT:

ALGORITHM:

I. TAKING INPUT ARRAY.

II. //PARTITION FUNCTION

III. CHOOSE FIRST ELEMENT AS APIVOT ELEMENT

IV. PLACE PIVOT AT ITS CORRECT POSITION BY ANALYSING COUNT OF LESSER ELEMENTS THAN PIVOT AND SWAPPING ARR[ST]WITH ARR[ST+COUNT].

V. ITERATING LEFT PART OF PIVOT AND RIGHT PART SIMULTANEOUSLY AND MAKING ALL LEFT ELEMENTS SMALLER THAN PIVOT AND RIGHT ELEMENTS GREATER THAN PIVOT BY SWAPPING WRONG POSITIONED ELEMENTS.

VI. AFTER THIS APPLY QUICK SORT RECURSIVELY ON LEFT AND RIGHT PART OF PIVOT ELEMENT.

VII. PRINT THE SORTED ARRAY.

TIME COMPLEXITY: $O(N^2)$

INPUT: INPUT ARRAY IS: 10, 56, 32, 1, 96

EXPECTED OUTPUT:

array after sorting will be:

1 ,10 ,32 ,56 ,96

CODE:

```
int partition(int input[],int st,int end)

{


  int pivot=input[st];   // STARTING AS A PIVOT ELEMENT

   int size=end-st+1;

   int count =0;

   for(int i=st+1;i<size;i++)

   {

       if(input[i]<pivot)

       {

           count++;          //SEARCHING FOR CORRECT POSITION OF PIVOT

       }

   }

   int temp=input[st];

   input[st]=input[st+count];      //PLACING PIVOT AT RGHT POSITION BY SWAPPING

   input[st+count]=temp;

   int j=st;

   int k=end;

   while(j<count&&k>count)     //MAKING ALL LEFT ELEMENTS LESS THAN PIVOT AND ALL
RIGHT ELEMENTS GREATER THEN PIVOT.
```

```
    {
        while(input[j]<pivot)
            j++;
        while(input[k]>=pivot)
            k--;
        if(input[j]>pivot&&input[k]<pivot)
        {
            int t=input[j];
            input[j]=input[k];
            input[k]=t;
            j++;
            k--;
        }
    }
    return count;       //RETURNING INDEX OF PIVOT ELEMENT



}

void sorting(int input[],int st,int end)
{

    if(st>=end)
        return;       // IF SIZE==0||SIZE==1 THEN SIMPLY RETURN
    int key=partition(input,st,end);   //CALLING PARTITION FUNCTION
    sorting(input,st,key-1);      //APPLY SORTING AT LEFT PART OF PIVOT ELEMENT
    sorting(input,key+1,end);      //APPLY SORTING AT RIGHT PART OF PIVOT ELEMENT
```

```
}
void quickSort(int input[], int size) {

    sorting(input,0,size-1);

}
#include<iostream>

using namespace std;


int main(){
    int n;
    cin >> n;   //SIZE OF ARRAY

    int *input = new int[n]; //DYNAMICALLY CREATE INPUT ARRAY

    for(int i = 0; i < n; i++) {
        cin >> input[i];   //TAKING INPUT
    }

    quickSort(input, n);
    cout<<"array after sorting will be : "<<endl;
    for(int i = 0; i < n; i++) {
        cout << input[i] << " ";  //PRINTING SORTED ARRAY
    }

    delete [] input;        //DELETE DYNAMICALLY ALLOCATED MEMEORY
```

}

OUTPUT:

```
enter the size of array
5
10 56 32 1 96
array after sorting will be:
1 10 32 56 96

Process returned 0 (0x0)   execution time : 10.121 s
Press any key to continue.
```

MERGE SORT:

ALGORITHM:

I. TAKE INPUT ARRAY.

II. MID=(ST+END)/2;APPLY MERGE SORT RECURSIVELY ON ARR(0.MID)AND ON ARR(MID+1,END) TILL WE GET SORTED LIST.

III. THEN MERGE SORTED LISTS TO GET FULLY SORTED LIST.

IV. PRINT THE OUTPUT LIST.

TIME COMPLEXITY: O(NLOG(N)).

INPUT: INPUT ARRAY IS: 10, 56, 32, 1, 96

EXPECTED OUTPUT:

array after sorting will be:

1 ,10 ,32 ,56 ,96

CODE:

```
void merge(int input[],int st,int mid,int end)
{
  int i=st;
  int j=mid+1;
```

```
    int tempsize=end-st+1;
    int *temp=new int[tempsize];    //output array
    int k=0;
    while((i<=mid)&&(j<=end))
    {
      if(input[i]<input[j])
      {                                 //filling output array till any two sorted array be finish
         temp[k++]=input[i++];
      }
       else
       {
          temp[k++]=input[j++];
       }
    }
    while(i<=mid)
    {
       temp[k++]=input[i++];           //if 2nd half array is finished
    }
    while(j<=end)
    {
       temp[k++]=input[j++];            //if 1st half array is finished
    }
    for(int l=0;l<tempsize;l++)
    {                                 //copying elements of output array in input array
       input[st++]=temp[l];
    }
    delete [] temp;
}
```

```
void sorting(int input[],int st,int end)

{

    if(st>=end)       //base case

        return;


    int mid=(st+end)/2;

    sorting(input,st,mid);       //merge sort on first half part

    sorting(input,mid+1,end);    //merge sort on 2nd half

    merge(input,st,mid,end);       //merge function to merge 2 sorted list

}

void mergeSort(int input[], int size){

    int st=0;

    int end=size-1;

                                        sorting(input,0,size-1);


}

#include<iostream>

using namespace std;

int main(){

    int n;

    cin >> n;   //SIZE OF ARRAY


    int *input = new int[n]; //DYNAMICALLY CREATE INPUT ARRAY


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

        cin >> input[i];   //TAKING INPUT

    }
```

```
    mergeSort(input, n);

    cout<<"array after sorting will be : "<<endl;

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

        cout << input[i] << " ";  //PRINTING SORTED ARRAY

    }


    delete [] input;        //DELETE DYNAMICALLY ALLOCATED MEMEORY


}
```

OUTPUT:

```
enter the size of array
5
10 56 32 1 96
array after sorting will be:
1 10 32 56 96

Process returned 0 (0x0)   execution time : 10.121 s
Press any key to continue.
```

BUBBLE SORT:

ALGORITHM:

I. Run a nested for loop to traverse the input array using two variables **i** and **j**, such that 0 ≤ j < n-1 and 0 ≤ i< n-j-1
II. If **arr[i]** is greater than **arr[i+1]** then swap these adjacent elements, else move on
III. Print the sorted array.

TIME COMPLEXITY: $O(N^2)$

INPUT: INPUT ARRAY IS: 10, 56, 32, 1, 96

EXPECTED OUTPUT:

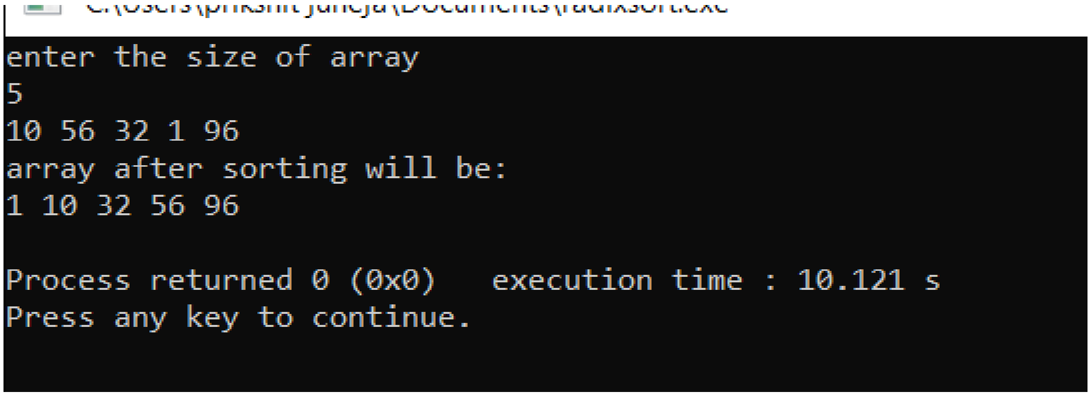
array after sorting will be:

1 ,10 ,32 ,56 ,96

CODE:

```
#include<iostream>
using namespace std;
void BubbleSort(int *input, int size)
{
  int j,i,temp;
   for(j=0;j<size-1;j++)     //for check condition for i
   {
      for(i=0;i<size-j-1;i++)     //array size virtually decreased by 1 from end
      {
         if(input[i]>input[i+1])
         {
            temp=input[i];              //swapping elements
            input[i]=input[i+1];
            input[i+1]=temp;
         }
      }
   }
}
int main(){
   int n;
   cin >> n;   //SIZE OF ARRAY

   int *input = new int[n]; //DYNAMICALLY CREATE INPUT ARRAY

   for(int i = 0; i < n; i++) {
      cin >> input[i];   //TAKING INPUT
   }

   BubbleSort(input, n);
   cout<<"array after sorting will be : "<<endl;
   for(int i = 0; i < n; i++) {
      cout << input[i] << " ";  //PRINTING SORTED ARRAY
   }

   delete [] input;       //DELETE DYNAMICALLY ALLOCATED MEMEORY

}
```

OUTPUT:



SELECTION SORT:

ALGORITHM:

I. Initialize minimum value(**min_idx**) to location 0.
II. Traverse the array to find the minimum element in the array.
III. While traversing if any element smaller than **min_idx** is found then swap both the values.
IV. Then, increment **min_idx** to point to the next element.
V. Repeat until the array is sorted.

TIME COMPLEXITY: $O(N^2)$

INPUT: INPUT ARRAY IS: 10, 56, 32, 1, 96

EXPECTED OUTPUT:

array after sorting will be:

1 ,10 ,32 ,56 ,96

CODE:

```
#include <iostream>
using namespace std;
void selectionSort(int input[], int n) {
for(int i = 0; i < n-1; i++ ) {
// Find min element in the array
int min = input[i], minIndex = i;
for(int j = i+1; j < n; j++) {
if(input[j] < min) {
min = input[j];
minIndex = j;
}
}
// Swap
int temp = input[i];
input[i] = input[minIndex];
input[minIndex] = temp;
}
}
```

```
int main(){
   int n;
   cin >> n;   //SIZE OF ARRAY

   int *input = new int[n]; //DYNAMICALLY CREATE INPUT ARRAY

   for(int i = 0; i < n; i++) {
      cin >> input[i];   //TAKING INPUT
   }

   selectionSort(input, n);
   cout<<"array after sorting will be : "<<endl;
   for(int i = 0; i < n; i++) {
      cout << input[i] << " ";  //PRINTING SORTED ARRAY
   }

   delete [] input;       //DELETE DYNAMICALLY ALLOCATED MEMEORY

}
```

<u>OUTPUT:</u>

```
enter the size of array
5
10 56 32 1 96
array after sorting will be:
1 10 32 56 96

Process returned 0 (0x0)   execution time : 10.121 s
Press any key to continue.
```

INSERTION SORT:

<u>ALGORITHM:</u>

I. ITERATE INDEX 1 TO LAST OF ARRAY.
II. COMPARE CURRENT ELEMENT WITH ITS PREDECESSOR.
III. IF CURR ELEMENT SMALLER THAN PREDECESSOR, COMPARE IT WITH ALL PREV ELEMENTS. MOVE THE GREATER ELEMNT ONE POS UP TO MAKE SPACE FOR SWAPPED ELEMENT.

<u>TIME COMPLEXITY:</u> $O(N^2)$

<u>INPUT:</u> INPUT ARRAY IS: 10, 56, 32, 1, 96

<u>EXPECTED OUTPUT:</u>

array after sorting will be:

1 ,10 ,32 ,56 ,96

CODE:

```
#include <iostream>
using namespace std;
void insertionSort(int *input, int size)
{
   int i,j,current;
   for(i=1;i<size;i++)
   {
      current=input[i];
      for(j=i-1;j>=0;j--)
      {
         if(input[j]>current)
         {
            input[j+1]=input[j];
         }
         else
            break;

      }
      input[j+1]=current;
   }
}
int main(){
   int n;
   cin >> n;   //SIZE OF ARRAY

   int *input = new int[n]; //DYNAMICALLY CREATE INPUT ARRAY

   for(int i = 0; i < n; i++) {
      cin >> input[i];   //TAKING INPUT
   }

   insertionSort(input, n);
   cout<<"array after sorting will be : "<<endl;
   for(int i = 0; i < n; i++) {
      cout << input[i] << " ";  //PRINTING SORTED ARRAY
   }

   delete [] input;       //DELETE DYNAMICALLY ALLOCATED MEMEORY

}
```

OUTPUT:

```
enter the size of array
5
10 56 32 1 96
array after sorting will be:
1 10 32 56 96

Process returned 0 (0x0)   execution time : 10.121 s
Press any key to continue.
```

PRACTICAL 2:

Perform the following sorting algorithms-Radix sort, Count sort, Bucket sort, and Shell sort.

RADIX SORT:

ALGORITHM:

I. BASICALLY THE RADIX SORT IS BASED ON COUNT SORT.IT IS APPLYING COUNT SORT DIGIT BY DIGIT.

II. FIRST WE FIND THE MAX AMONG ARRAY ELEMENTS.THEN CALL COUNT SORT TILL MAX/POS>0 WHERE POS INITIALLY=1 AND INCREMENTED BY 10 TIMES.

III. MAKE A COUNT ARRAY HAVING SIZE 10 AS UNIQUE NO.OF DIGITS ARE 0-9 OR BASE IS DECIMAL.INITIALISE ALL ELMENTS WITH 0.

IV. UPDATE COUNT ARRAY WITH FREQUENCY OF UNIT DIGIT COUNT i.e.count[ARR[I]/POS)%10]++.FINALLY UPDATE COUNT[I]=COUNT[I]+COUNT[I-1].

V. MAKE OUTPUT ARRAY HAVING SIZE SAME AS INPUT ARRAY AND FILL THIS OUTPUT[--COUNT[(ARR[I]/POS)%10]]=ARR[I]

VI. COPY CONTENTS OF OUTPUT ARR IN INPUT ARRAY.NOW APPLY COUNT SORT AGAIN ON TENS DIGIT TAKING NEW ARRAY AS INPUT

ARR.

TIME COMPLEXITY: O((n+b) * $\log_b$(k)) //B->BASE AS IN HERE B=10,K->MAX POSSIBLE VALUE

INPUT: INPUT ARRAY IS: 10, 56, 32, 1, 96

EXPECTED OUTPUT:

array after sorting will be:

1 ,10 ,32 ,56 ,96

CODE:

```
#include<iostream>

#include<algorithm>

using namespace std;


void count_sort(int*arr,int n,int pos)

{


//initialising count array...size of count is fixed as number of digits are 0-9.

   int count_arr[10]={0};

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

   {

      count_arr[(arr[i]/pos)%10]++;

//(arr[i]/pos)%10 is used to decided digit upon which count sort have to apply

  }

   //updating count array

   for(int i=1;i<=10;i++)

   {

      count_arr[i]=count_arr[i]+count_arr[i-1];


   }
```

```
//creating and updating output array
    int output[n];
    for(int i=n-1;i>=0;i--)
    {
        output[--count_arr[(arr[i]/pos)%10]]=arr[i];
        }
    //copying output array in input array
    for(int i=0;i<n;i++)
    {
        arr[i]=output[i];
    }
}

void radix_sort(int *arr,int n)
{
   //finding maximum element
    int maximum=arr[0];
    for(int i=1; i<n; i++)
    {
        if(arr[i]>maximum)
            maximum=arr[i];
    }
    //deciding frequency of count sort
    for(int pos=1;maximum/pos>0;pos*=10)
    {
        count_sort(arr,n,pos);
    }
```

```
}

int main()
{
    int n;
    cout<<"enter the size of array"<<endl;
    cin>>n;
    int arr[n];
    //taking input
    for(int i=0; i<n; i++)
    {
        cin>>arr[i];
    }
    radix_sort(arr,n);
  cout<<"array after sorting will be:"<<endl;
//printing array
    for(int i=0;i<n;i++)
    {
        cout<<arr[i]<<' ';
 }
    cout<<endl;
}
```

OUTPUT:

```
enter the size of array
5
10 56 32 1 96
array after sorting will be:
1 10 32 56 96

Process returned 0 (0x0)   execution time : 10.121 s
Press any key to continue.
```

COUNT SORT:

ALGORITHM:

I. TAKE THE INPUT ARRAY OF SIZE N .IN COUNT SORT FIRST WE FIND THE MAX ELEMENT TO FIND RANGE OF NUMBERS i.e.K.

II. MAKE A COUNT ARRAY OF SIZE K+1 AND INITILAISE ALL ELEMENTS WITH 0.

III. AFTER THIS UPDATE COUNT i.e.COUNT [ARR[I]]++.

IV. FINALLY UPDATE COUNT ARRAY AS COUNT[I]=COUNT[I]+COUNT[I-1].

V. MAKE OUTPUT ARRAY OF SIZE SAME AS INPUT AND FILL IT AS OUTPUT[--COUNT[ARR[I]]]=ARR[I].

VI. NOW COPY THE OUTPUT ARRAY IN INPUT ARRAY.

TIME COMPLEXITY: O(N + K) //N->SIZE OF ARRAY,K->MAX ELEMENT OF ARRAY.

INPUT: INPUT ARRAY IS: 10, 56, 32, 1, 96

EXPECTED OUTPUT:

array after sorting will be:

1 ,10 ,32 ,56 ,96

CODE:

```
#include<iostream>

#include<algorithm>

using namespace std;


void count_sort(int*arr,int n)

{

   //finding maximum element

   int maximum=arr[0];

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

   {

      if(arr[i]>maximum)

         maximum=arr[i];

   }

   int k=maximum;

//initialising count array

   int count_arr[k+1]={0};

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

   {

      count_arr[arr[i]]++;


   }

   //updating count array

   for(int i=1;i<=k;i++)

   {

      count_arr[i]=count_arr[i]+count_arr[i-1];


   }
```

```
//creating and updating output array
    int output[n];
    for(int i=n-1;i>=0;i--)
    {
        output[--count_arr[arr[i]]]=arr[i];
        }
    //copying output array in input array
    for(int i=0;i<n;i++)
    {
        arr[i]=output[i];
    }
}
int main()
{
    int n;
    cout<<"enter the size of array"<<endl;
    cin>>n;
    int arr[n];
    //taking input
    for(int i=0; i<n; i++)
    {
        cin>>arr[i];
    }
    count_sort(arr,n);
    cout<<"array after sorting will be:"<<endl;

//printing array
    for(int i=0;i<n;i++)
```

```
  {
      cout<<arr[i]<<' ';
}
  cout<<endl;
}
```

OUTPUT:

```
enter the size of array
5
10 56 32 1 96
array after sorting will be:
1 10 32 56 96

Process returned 0 (0x0)   execution time : 25.536 s
Press any key to continue.
```

SHELL SORT:

ALGORITHM:

I. TAKE INPUT ARRAY.

II. INITIALISE GAP WITH N/2 WHICH IS DECREMENTED BY HALF OF ITS VALUE.

III. DIVIDE LIST INTO SMALLER SUBPARTS HAVING EQUAL INTERVALS OF GAP.

IV. SORT THESE LIST USING INSERTION SORT.

V. REPEAT STEP 2 TILL LIST IS SORTED.

VI. PRINT THE SORTED LIST.

TIME COMPLEXITY:

WORST CASE->$O(N^2)$

BEST CASE->O(N)

INPUT: INPUT ARRAY IS: 10, 56, 32, 1, 96

EXPECTED OUTPUT:

array after sorting will be:

1 ,10 ,32 ,56 ,96

CODE:

```
#include<iostream>

#include<algorithm>

using namespace std;


void shell_sort(int*arr,int n)

{

   for(int gap=n/2;gap>=1;gap/=2)  //GAP IS TO DECIDE WHICH ELEMENTS TO BE COMPARED.

                                  // IF GAP=1 THEN BASICALLY INSERTION SORT.

   {

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

       {

           for(int i=j-gap;i>=0;i-=gap) //I=I-GAP AS TO CHECK PRECEDED ELEMENTS INITIALLY I
WILL BE 0.

                                      //PERFORMING GAPWISE COMPARISION

           {

               if(arr[i+gap]>arr[i])

                   break;

               else

               {

                   int t=arr[i+gap];

                   arr[i+gap]=arr[i];

                   arr[i]=t;

               }
```

```
          }
        }
    }
}

int main()
{
    int n;
    cout<<"enter the size of array"<<endl;
    cin>>n;
    int arr[n];
    //taking input
    for(int i=0; i<n; i++)
    {
        cin>>arr[i];
    }
    shell_sort(arr,n);

    cout<<"array after sorting will be:"<<endl;

//printing array
    for(int i=0;i<n;i++)
    {
        cout<<arr[i]<<' ';

    }
    cout<<endl;
}
```

OUTPUT:

```
enter the size of array
5
10 56 32 1 96
array after sorting will be:
1 10 32 56 96

Process returned 0 (0x0)   execution time : 25.536 s
Press any key to continue.
```

BUCKET SORT:

ALGORITHM:

I. Create n empty buckets (Or lists).

II. Do following for every array element arr[i].

III. ....a) Insert arr[i] into bucket[n*array[i]]

IV. Sort individual buckets using insertion sort.

V. Concatenate all sorted buckets.

VI. END.

INPUT: size of array :

5

0.1, 0.6, 0.9, 0.2, 0.3

EXPECTED OUTPUT:

array after sorting will be:

0.1, 0.2, 0.3 ,0.6, 0.9

CODE:

#include <iostream>

```
#include <vector>

using namespace std;

void insertionSort (vector<float> &arr, int n) {

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

    for (int j = i; j > 0; j--) {

      if (arr[j] < arr[j - 1]) {

        float temp = arr[j];

        arr[j] = arr[j - 1];

        arr[j - 1] = temp;

      }

      else {

        break;

      }

    }

  }

  return;

}

void bucketSort(float a[], int n) {

  vector<float> v[n];

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

    v[(int) ((n * a[i]) / 1.0)].push_back(a[i]);        //FILLING THE BUCKET

  }

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

    if (v[i].size()) {

      insertionSort(v[i], v[i].size());             //IF SIZE!=0 APPLYING INSERTION SORT

    }

  }
```

```
  int k = 0;
  for (int i = 0; i < n; i++) {
    for (int j = 0; j < v[i].size(); j++) {
      a[k++] = v[i][j];                    //COPYING ELEMENTS IN INPUT ARRAY
    }
  }
  return;
}
int main() {
  int n = 0;
  cout << "enter the size of array : "<<endl;
  cin >> n;
  float arr[n];

  for (int i = 0; i < n; i++) {
  cin >> arr[i];
  }
  bucketSort(arr, n);
  cout << "array after sorting will be: "<<endl ;
  for (int i = 0; i < n; i++) {
  cout << arr[i] << " ";
  }
  return 0;
}
```

OUTPUT:

```
enter the size of array :
5
0.1 0.6 0.9 0.2 0.3
array after sorting will be:
0.1 0.2 0.3 0.6 0.9
Process returned 0 (0x0)   execution time : 26.277 s
Press any key to continue.
```

PRACTICAL 3:

Perform searching algorithm-Linear and Binary search.

LINEAR SEARCH:

ALGORITHM:

I. START ITERATING FROM INDEX 0 TO LAST INDEX.

II. IF ARR[I]=KEY VALUE THEN RETURN THE INDEX I.e . I AND BREAK THE LOOP.

III. IF I=N THEN RETURN -1.

TIME COMPLEXITY: O(N)

INPUT:

ARRAY: 5,6,9,3,1,2

KEY VALUE :9

EXPECTED OUPUT:

2

CODE:

```
#include<iostream>

using namespace std;

int linearSearch(int arr[], int n, int x)

{


   int c=-1;

   //cin>>x;
```

```
    for(int i=0;i<n;i++)
    {
     if(arr[i]==x)        //CHECKING ARR[I] WITH X
        {
           c=i ;
            break;
        }



    }
    return c;

}
int main(){
    int n;
    cout<<"size ? "<<endl;
    cin >> n;   //SIZE OF ARRAY

    int *input = new int[n]; //DYNAMICALLY CREATE INPUT ARRAY

    for(int i = 0; i < n; i++) {
        cin >> input[i];   //TAKING INPUT
    }
    int key;
    cout<<"enter the key value : "<<endl;
    cin>>key;

    cout<<"index would be : "<<linearSearch(input,n,key);
    delete [] input;       //DELETE DYNAMICALLY ALLOCATED MEMEORY
```

}

OUTPUT:

```
size ?
6
5 6 9 3 1 2
enter the key value :
9
index would be : 2
Process returned 0 (0x0)   execution time : 12.455 s
Press any key to continue.
```

BINARY SEARCH:

ALGORITHM:

I. START CHECKING THE KEY VALUE FROM MID OF INPUT ARRAY.IF ARR[I]=KEY THEN RETURN I.

II. ELSE IF ARR[I]>KEY THEN SEARCH IN LEFT HALF PORTION .

III. ELSE SEARCH IN RIGHT HALF PORTION.

IV. REPEAT THIS UNTIL ELEMENT IS FOUND OR INTERVAL IS 0. IF ELEMENT IS NOT FOUND AND INTERVAL BECOMES 0 THEN RETURN -1.

TIME COMPLEXITY: O(LOG(N))

INPUT:

ARRAY: 5,6,9,3,1,2

KEY VALUE :9

EXPECTED OUPUT:

2

CODE:

```
#include<iostream>
using namespace std;
int binarySearch(int *input, int n, int val)
{
   int l=0;
   int r=n-1;
   bool result=false;

   int c=-1;

   while(l<=r)

   {
      int mid=(l+r)/2;
      if(input[mid]<val)
         l=mid+1;                    //SEARCH IN 2ND HALF
      else if(input[mid]>val)
         r=mid-1;//SEARCH IN 1ST HALF
      else
         c=mid;      //ELEMENT FOUND
         result=true;
         return c;              //PROGRAM COMPLETED
         break;
  }
   }
   if(result==false)
   return c;

}
```

```
int main(){
    int n;
    cout<<"size ? "<<endl;
    cin >> n;   //SIZE OF ARRAY

    int *input = new int[n]; //DYNAMICALLY CREATE INPUT ARRAY

    for(int i = 0; i < n; i++) {
        cin >> input[i];   //TAKING INPUT
    }
    int key;
    cout<<"enter the key value : "<<endl;
    cin>>key;

    cout<<"index would be : "<<binarySearch(input,n,key);
    delete [] input;       //DELETE DYNAMICALLY ALLOCATED MEMEORY

}
```

OUTPUT:

```
size ?
6
5 6 9 3 1 2
enter the key value :
9
index would be : 2
Process returned 0 (0x0)   execution time : 12.455 s
Press any key to continue.
```

PRACTICAL 4:

Perform tower of Hanoi.

ALGORITHM:

I. IF(N=1)THEN MOVE 1 PLATE A->PLATE C. //BASE CASE

II. MOVE N-1 PLATES FROM A TO B USING C USING RECURSION.

III. MOVE LAST PLATE FROM A TO C.

IV. MOVE N-1 PLATES FROM B TO C USING A USING RECURSION.

V. END.

TIME COMPLEXITY: $O(2^N)$

INPUT:

NUMBER OF PLATES =3

EXPECTED OUTPUT:

a c

a b

c b

a c

b a

b c

a c

CODE:

```
#include <iostream>
using namespace std;
void towerOfHanoi(int n, char source, char auxiliary, char destination) {

   if(n==0)
   {
      return;          //BASE CASE
   }
   if(n==1){
```

```
    cout<<source<<' '<<destination<<endl;
        return;



    }
   towerOfHanoi(n-1,source,destination,auxiliary);          // N-1 PLATES FROM A->B USING C
    cout<<source<<' '<<destination<<endl;                // REMAINING LAST PLATE FROM A->C
    towerOfHanoi(n-1,auxiliary,source,destination);     //N-1 PLATES FROM B->C USING A
}
int main() {
    cout<<"number of plates ? "<<endl;
    int n;
    cin >> n;
    towerOfHanoi(n, 'a', 'b', 'c');
}
```

OUTPUT:

```
number of plates ?
3
a c
a b
c b
a c
b a
b c
a c

Process returned 0 (0x0)   execution time : 3.543 s
Press any key to continue.
```

PRACTICAL 5:

Write a program for inserting elements in: i. AVL tree ii. Red-Black Tree iii. BST

AVL:

ALGORITHM:

I. DO THE INSERTION SAME AS BST .DURING INSERTION TAKE CARE OF BALANCING FACTOR(BF).

II. BF SHOULD BE IN RANGE OF -1 TO 1.

III. IF IT IS NOT IN RANGR THRN WE HAVE TO BALANCE THE TREE.

a) IF ADDED NODE IS IN RIGHT SUBTREE OF RIGHT CHILD OF THE UNBALANCED NODE,PERFORM RR ROTATION.

b) IF ADDED NODE IS IN LEFT SUBTREE OF LEFT CHILD OF THE UNBALANCED NODE,PERFORM LL ROTATION.

c) IF ADDED NODE IS IN RIGHT SUBTREE OF LEFT CHILD OF THE UNBALANCED NODE,PERFORM RL ROTATION.

d) IF ADDED NODE IS IN LEFT SUBTREE OF RIGHT CHILD OF THE UNBALANCED NODE,PERFORM LR ROTATION.

TIME COMPLEXITY: O(LOG(N)) //N->NUMBER OF NODES

INPUT:

NODES: 2,6,5,10,69

EXPECTED OUTPUT:

2

Inorder: 2

Postorder: 2

6

Inorder: 2 6

Postorder: 6 2

5

Inorder: 2 5 6

Postorder: 2 6 5

10

Inorder: 2 5 6 10

Postorder: 2 10 6 5

69

Inorder: 2 5 6 10 69

Postorder: 2 6 69 10 5

CODE:

```
#include <iostream>
using namespace std;
//NODE CLASS
class node {
public:
  int data;
  node *left;
  node *right;
 node *parent;
//PARAMATERISED CONSTRUCTOR
  node(int d) {
    data = d;
    left = NULL;
    right = NULL;
    parent = NULL;
  }
//DEFAULT CONSTRUCTOR
  node() {
    left = NULL;
    right = NULL;
    parent = NULL;
  }
};
class avl {
public:
  node *root;
//DEFAULT CONSTRUCTOR
  avl() {
    root = NULL;
```

```
}
//FINDING HEIGHT OF TREE
int get_height(node *root) {
  if (root == NULL) {
    return 0;
  }
  return 1 + max(height(root->right), height(root->left));
}
void rrRotation(node *prt) {
// right right rotation
  node *temp = prt->right;
  if (prt->parent) {
    if (prt == prt->parent->right) {
      prt->parent->right = prt->right;
    }
    else {
      prt->parent->left = prt->right;
    }
  }
  prt->right->parent = prt->parent;
  prt->right = prt->right->left;
  if (temp->left) {
    temp->left->parent = prt;
  }
  temp->left = prt;
  prt->parent = temp;
  if (temp->parent == NULL) {
    root = temp;
  }
  return;
```

```
  }
  void llRotation(node *prt) {   //left left rotation
    node *temp = prt->left;
    if (prt->parent) {
      if (prt->parent->left == prt) {
        prt->parent->left = prt->left;
      }
      else {
        prt->parent->right = prt->left;
      }
    }
    prt->left->parent = prt->parent;
    prt->left = prt->left->right;
    if (temp->right) {
      temp->right->parent = prt;
    }
    temp->right = prt;
    prt->parent = temp;
    if (temp->parent == NULL) {
      root = temp;
    }
    return;
  }
  void rlRotation(node *prt) {
// right left rotation
    llRotation(prt->right);
    rrRotation(prt);
    return;
  }
  void lrRotation(node *prt) {
```

```
// left right rotation
      rrRotation(prt->left);
      llRotation(prt);
      return;
   }
//BALANCING THE NODE
   void balance(node *ans) {
      node *prt = ans->parent;
      node *prev = ans;
      while (prt) {
         int hl = height(prt->left);
         int hr = height(prt->right);
         if (ans(hl - hr) > 1) {
            if (hr > hl) {
               if (prev == prt->right and ans == prev->right) {
                  rrRotation(prt);    // PERFORM RR ROTATION WRT PRT NODE
               }
               else {
                  rlRotation(prt);    //PERFORM RL ROTATION WRT PRT NODE
               }
            }
            else {
               if (prev == prt->left and ans == prev->left) {
                  llRotation(prt);
               }
               else {
                  lrRotation(prt);
               }
            }
         }
```

```
      ans = prev;
      prev = prt;
      prt = prt->parent;
    }
    return;
  }
void insert(int d) {
    if (root == NULL) {
      root = new node(d);
      return;
    }
    node *ptr = root;
    node *newNode = new node(d);
    while (1) {
      if (ptr->data > d) {
        if (ptr->left) {
          ptr = ptr->left;
        }
        else {
          ptr->left = newNode;
          newNode->parent = ptr;
          break;
        }
      }
      else {
        if (ptr->right) {
          ptr = ptr->right;
        }
        else {
          ptr->right = newNode;
```

```
          newNode->parent = ptr;
          break;
        }
      }
    }
    balance(newNode);
    return;
  }
  void inorder(node *root) {
    if (root == NULL) {
      return;
    }
    inorder(root->left);
    cout << root->data << " ";
    inorder(root->right);
    return;
  }
  void postorder(node *root) {
    if (root == NULL) {
      return;
    }
    postorder(root->left);
    postorder(root->right);
    cout << root->data << " ";
    return;
  }
};
int main() {
  int d = 0;
  avl b;
```

```
  cin >> d;
  while (d != -1) {
    b.insert(d);
    cout << "Inorder: ";
    b.inorder(b.root);
    cout << endl;
    cout << "Postorder: ";
    b.postorder(b.root);
    cout << endl;
    cin >> d;
  }
  return 0;
}
```

OUTPUT:

```
2
Inorder: 2
Postorder: 2
6
Inorder: 2 6
Postorder: 6 2
5
Inorder: 2 5 6
Postorder: 2 6 5
10
Inorder: 2 5 6 10
Postorder: 2 10 6 5
69
Inorder: 2 5 6 10 69
Postorder: 2 6 69 10 5
-1

Process returned 0 (0x0)   execution time : 16.567 s
Press any key to continue.
```

RED BLACK TREE:

ALGORITHM:

I. INSERT THE NODE AS SAME AS IN BST. IF TREE FOLLOWS ALL THE PROPERTIES OF RED BLACK TREE THEN DO THE FURTHER INSERTION IN TREE.

II. IF TREE VIOLATES THE PROPERTY OF RED BLACK TREE THEN PERFORM COLORATION IN TREE.

a. IF THE COLOR OF BOTH -THE NEW NODE PARENT AND ITS UNCLE IS RED ,WE COLOR BOTH THESE NODES BLACK.AFTER THIS APPLY RECOLOURING PROPERTY TO NEWNODE GRANDPARENT .

b. IF PARENT IS RED AND UNCLE IS BLACK ,WE CHECK IF OUR NEWNODE IS RIGHT CHILD OF ITS OPARENT.IF IT IS WE PERFORM EXTRA ROTATION STEP OF LEFT ROTATING THE TREE AT ITS PARENT.

c. _CHANGE COLOR OF PARENT TO BLACK AND GRANDPARENT TO RED.RIGHT ROTATE THE TREE AT NEW NODES GRANDPARENT.

TIME COMPLEXITY: O(LOGN)

INPUT: NODES ARE: 5,4,6,1,2

EXPECTED OUTPUT:

Inorder: 1R 2B 4R 5B 6B

Postorder: 1R 4R 2B 6B 5B

CODE:

```
#include <iostream>

using namespace std;

class Treenode {

public:

  int data;

  bool color;      //0 - red, 1 - black

  Treenode *left;

  Treenode *right;

  Treenode *parent;

  Treenode(int d) {

    data = d;

    left = NULL;

    right = NULL;
```

```
    parent = NULL;
    color = 0;
  }
};
class rbtree {
public:
  Treenode *root;
  rbtree() {
    root = NULL;
  }
  int blackHt(Treenode *root) {
    if (root == NULL) {
      return 0;
    }
    int add = ((root->color == 0) ? 1 : 0);
    return add + max(blackHt(root->right), blackHt(root->left));
  }
  void leftRotation(node *prt) {
    node *temp = prt->right;
    if (prt->parent) {
      if (prt == prt->parent->right) {
        prt->parent->right = prt->right;
      }
      else {
        prt->parent->left = prt->right;
      }
    }
    prt->right->parent = prt->parent;
    prt->right = prt->right->left;
    if (temp->left) {
```

```
    temp->left->parent = prt;
  }
  temp->left = prt;
  prt->parent = temp;
  if (temp->parent == NULL) {
    root = temp;
  }
  return;
}
void rightRotation(node *prt) {
  node *temp = prt->left;
  if (prt->parent) {
    if (prt->parent->left == prt) {
      prt->parent->left = prt->left;
    }
    else {
      prt->parent->right = prt->left;
    }
  }
  prt->left->parent = prt->parent;
  prt->left = prt->left->right;
  if (temp->right) {
    temp->right->parent = prt;
  }
  temp->right = prt;
  prt->parent = temp;
  if (temp->parent == NULL) {
    root = temp;
  }
  return;
```

```
}
void recolor(node *ans) {
  while (!ans->parent->color) {
    if (ans->parent == ans->parent->parent->left) {
      node* uncle = ans->parent->parent->right;
      if (uncle and uncle->color == 0) {
        ans->parent->color = 1;
        uncle->color = 1;
        ans->parent->parent->color = 0;
        ans = ans->parent->parent;
      }
      else {
        if (ans == ans->parent->right) {
          ans = ans->parent;
          leftRotation(ans);
        }
        ans->parent->color = 1;
        ans->parent->parent->color = 0;
        rightRotation(ans->parent->parent);
      }
    }
    else {
      node* uncle = ans->parent->parent->left;
      if (uncle and uncle->color == 0) {
        ans->parent->color = 1;
        uncle->color = 1;
        ans->parent->parent->color = 0;
        ans = ans->parent->parent;
      }
      else {
```

```
        if (ans == ans->parent->left) {
          ans = ans->parent;
          rightRotation(ans);
        }
        ans->parent->color = 1;
        ans->parent->parent->color = 0;
        leftRotation(ans->parent->parent);
      }
    }
    if (ans == root) {
      break;
    }
  }
  root->color = 1;
  return;
}
void insert(int d) {
  if (root == NULL) {
    root = new node(d);
    root->color = 1;
    return;
  }
  node *ptr = root;
  node *newNode = new node(d);
  while (1) {
    if (ptr->data > d) {
      if (ptr->left) {
        ptr = ptr->left;
      }
      else {
```

```
          ptr->left = newNode;
          newNode->parent = ptr;
          break;
        }
      }
      else {
        if (ptr->right) {
          ptr = ptr->right;
        }
        else {
          ptr->right = newNode;
          newNode->parent = ptr;
          break;
        }
      }
    }
    recolor(newNode);
    return;
}
void inorder(node *root) {
    if (root == NULL) {
      return;
    }
    inorder(root->left);
    cout << root->data << (root->color ? "B" : "R") << " ";
    inorder(root->right);
    return;
}
void postorder(node *root) {
    if (root == NULL) {
```

```
            return;
        }
        postorder(root->left);
        postorder(root->right);
        cout << root->data << (root->color ? "B" : "R") << " ";
        return;
    }
};
int main() {
    int d = 0;
    rbtree b;
    cin >> d;
    while (d != -1) {
        b.insert(d);
        cin >> d;
    }
    cout << "Inorder: ";
    b.inorder(b.root);
    cout << endl;
    cout << "Postorder: ";
    b.postorder(b.root);
    cout << endl;
    return 0;
}
```

OUTPUT:

```
Inorder: 1R 2B 4R 5B 6B
Postorder: 1R 4R 2B 6B 5B

Process returned 0 (0x0)   execution time : 15.437 s
Press any key to continue.
```

BST :

ALGORITHM:

I. CONSTRUCT A CLASS TREENODE HAVING PUBLIC PROPRTIES DATA AND POINTERS TO LEFT AND RIGHT CHILD.

II. CREATE A PARAMATERISED CONSRTUCTOR WHICH CREATES A NODE WITH DATA.

III. CREATE A NEWNODE DYNAMICALLY. TREENODE *NEWNODE=NEW TREENODE(INT DATA).

IV. IF (ROOT=NULL) THEN NEWNODE=ROOT AND AS ROOT IS PASSED AS REFERENCE CHANGES ARE REFLECTED DIRECTLY.

V. IF(ROOT->DATA>DATA)THEN INSERT _NODE(ROOT->LEFT,DATA).

VI. IF(ROOT->DATA<DATA)THEN INSERT_NODE(ROOT->RIGHT,DATA).

VII. END.

TIME COMPLEXITY:

GENERAL CASE : O(H) // H IS HEIGHT OF TREE.

WORST CASE: O(N) // N IS NO. OF NODES.

INPUT:

NODES WANT TO INSERT:30,15,40,10

EXPECTED OUTPUT:

preorder of tree is:

30 15 10 40

CODE:

```
#include<iostream>

using namespace std;

#include<cstddef>

class TreeNode

{

public:

   int data;
```

```
    TreeNode *left,*right;

    TreeNode(int data)
    {
        this->data=data;
        left=NULL;
        right=NULL;
    }
};
void insert_node(TreeNode*&root,int value)
{
    TreeNode *newnode=new TreeNode(value);
    if(root==NULL)
    {
        root=newnode;
        return;
    }
    if(root->data>value)
    {
        insert_node(root->left,value);
    }
    else if(root->data<value)
    {
        insert_node(root->right,value);
    }
    return;
}
```

```
void print_tree(TreeNode *root)
{
    //preorder

    if(root==NULL)
        return;

    cout<<root->data<<' ';
    print_tree(root->left);
    print_tree(root->right);

    return;
}
int main()
{
    TreeNode*root=NULL;
    insert_node(root,30);
    insert_node(root,15);
    insert_node(root,40);
    insert_node(root,10);
    cout<<"preorder of tree is:"<<endl;
    print_tree(root);
    Return 0;
}
```

OUTPUT:

```
preorder of tree is:
30 15 10 40
Process returned 0 (0x0)   execution time : 0.074 s
Press any key to continue.
```

PRACTICAL 6:

Write a program for deleting elements in: i. AVL tree ii. Red-Black Tree iii. BST

AVL:

ALGORITHM:

I. THE CURR NODE MUST BE ONE OF THE ANCESTORS OF PREVIOUS NODE.AFTER THIS UPDATE THE HEIGHT OF NODE.

II. FIND THE BALANCE FACTOR OF CURR NODE.

III. IF AFTER DELETION BALANCE FACOR IS NOT IN RANGE OF -1 TO 1 THEN REBALANCE THE TREE.

IV. FOUR CASES ARISE IN REBALANCING:

a. LL CASE

b. RR CASE

c. LR CASE

d. RL CASE

V. IF BALANCE FACTOR IS GREATER THAN 1 THEN IT CAN BE EITHER LL OR LR CASE.IF BALANCE FACTOR OF LEFT SUBTREE IS GREATER THAN OR EQUAL TO 0 THEN IT IS LL CASE ELSE IT IS LR CASE.

VI. IF BALNCE FACTOR IS LESS THAN -1 THEN IT IS EITHER RR OR RL CASE . IF BALANCE FACTOR OF RIGHT SUBTREE IS SMALLER THAN OR EQUAL TO 0 THEN IT IS RR ELSE IT IS RL CASE.

Time COMPLEXITY: O(LOGN)

INPUT:

INPUT TREE:

8,4,1,-1,5,7,9,10

<u>EXPECTED OUTPUT:</u>

PREORDER AFTER DELETION:4,1,-1,8,5,7,10

<u>CODE:</u>

```
#include<iostream>
Using namespace std;

class Node
{
   int key;
    Node *left;
    Node *right;
           int height;
           Node(int data)
{
         this->key= data;
   left   = NULL;
   right  = NULL;
   height = 1;
   }
 };
int height(Node *n)
{
   if (n == NULL)
       return 0;
   return n->height;
}
```

```
int max(int a, int b)
{
    if(a>b)
    return a;
    return b;
}

Node *RotateRight( Node *n)
{
    Node *x = n->left;
    Node *T2 = x->right;
     x->right = n;
    n->left = T2;
     n->height = max(height(n->left), height(n->right))+1;
        x->height = max(height(x->left), height(x->right))+1;
        return x;
}
 Node *RotateLeft(Node *n)
{
    Node *y = n->right;
    Node *T2 = y->left;
     y->left = n;
    n->right = T2;
     n->height = max(height(n->left), height(n->right))+1;
    y->height = max(height(y->left), height(y->right))+1;
        return y;
}
```

```
int Balance(Node *n)
{
    if (n == NULL)
        return 0;
    return height(n->left) - height(n->right);
}

Node* insert(Node* node, int key)
{
    if (node == NULL)
        return(newnode(key));

    if (key < node->key)
        node->left  = insert(node->left, key);
    else if (key > node->key)
        node->right = insert(node->right, key);
    else
        return node;
    node->height = 1 + max(height(node->left),height(node->right));
    int b = Balance(node);
   if (b > 1 && key < node->left->key)
        return RotateRight(node);
    if (b < -1 && key > node->right->key)
        return RotateLeft(node);
    if (b > 1 && key > node->left->key)
    {
        node->left = RotateLeft(node->left);
```

```
        return RotateRight(node);
    }
    if (b < -1 && key < node->right->key)
    {
        node->right = RotateRight(node->right);
        return RotateLeft(node);
    }
    return node;
}
 Node * minNode(Node* n)
{
    Node* current = n;
    while (current->left != NULL)
        current = current->left;

    return current;
}
Node* deleteNode(Node* root, int key)
{
    if (root == NULL)
        return root;
    if ( key < root->key )
        root->left = deleteNode(root->left, key);
    else if( key > root->key )
        root->right = deleteNode(root->right, key);
    else
    {
```

```
    if( (root->left == NULL) || (root->right == NULL) )
    {
        Node *temp = root->left ? root->left : root->right;
        if (temp == NULL)
        {
            temp = root;
            root = NULL;
        }
        else
         *root = *temp;
        delete(temp);
    }
    else
    {
         Node* temp = minNode(root->right);
        root->key = temp->key;
        root->right = deleteNode(root->right, temp->key);
    }
}
if (root == NULL)
  return root;
root->height = 1 + max(height(root->left),height(root->right));
int b = Balance(root);
if (b > 1 && Balance(root->left) >= 0)
    return RotateRight(root);
if (b > 1 && Balance(root->left) < 0)
{
```

```
        root->left = RotateLeft(root->left);
        return RotateRight(root);
    }
    if (b < -1 && Balance(root->right) <= 0)
        return RotateLeft(root);
    if (b < -1 && Balance(root->right) > 0)
    {
        root->right = RotateRight(root->right);
        return RotateLeft(root);
    }
   return root;
}
void preOrder(struct Node *root)
{
    if(root != NULL)
    {
        Cout<<root->key;
        preOrder(root->left);
        preOrder(root->right);
    }
}

int main()
{
  Node *root = NULL;
         root = insert(root, 8);
    root = insert(root, 4);
```

```
    root = insert(root, 9);

    root = insert(root, 1);

    root = insert(root, 5);

    root = insert(root, 10);

    root = insert(root, -1);

    root = insert(root, 1);

    root = insert(root, 7);


    Cout<<"Preorder traversal before deletion:"<<endl;

    preOrder(root);

    root = deleteNode(root, 9);

    Cout<<"Preorder traversal after deletion of 9:"<<endl;

    preOrder(root);

    return 0;

}
```

OUTPUT:

"C:\C++ tutorials\sa.exe"

```
Preorder traversal before deletion:
8 4 1 -1 5 7 9 10
Preorder traversal after deletion of 9:
4 1 -1 8 5 7 10
Process returned 0 (0x0)   execution time : 0.028 s
Press any key to continue.
```

RED BLACK:

ALGORITHM:

I.

BST:

ALGORITHM:

I. IF(ROOT->DATA>DATA) THEN RECURSION ON LEFT PART i.e. ROOT->LEFT=DLETE_NODE(ROOT->LEFT,DATA).RETURN ROOT.

II. IF(ROOT->DATA<DATA) THEN RECURSION ON RIGHT PART i.e. ROOT->RIGHT=DLETE_NODE(ROOT->RIGHT,DATA).RETURN ROOT.

III. ELSE ROOT NODE TO BE DELETED.

1. IF(ROOT->LEFT&&ROOT->RIGHT)BOTH ARE NULL THEN RETURN NULL.
2. ELSE IF(ROOT->LEFT=NULL)THEN PRESERVE ROOT->RIGHT IN TEMP. DELETE ROOT.RETURN TEMP AS A ROOT.
3. ELSE IF(ROOT->RIGHT=NULL)THEN PRESERVE ROOT->LEFT IN TEMP. DELETE ROOT.RETURN TEMP AS A ROOT.
4. ELSE REPLACE ROOT WITH INORDER SUCCESSOR.TEMP ->INORDER SUCCESSOR .SWAP TEMP->DATA WITH ROOT ->DATA.root->right=delete_node(root->right,temp->data); //recursion delete_node(root ->right,temp->data)
5. return root;
6. END.

TIME COMPLEXITY:

GENERAL: O(H) //H IS HEIGHT OF TREE

WORST:O(N) // N IS NUMBER OF NODES

INPUT:

NODE HAVING VALUE 40 TO BE DELETED.

EXPECTED OUTPUT:

preorder of tree is:

30 15 10 40

tree after deleting node having value 40 will be:

30 15 10

CODE:

```
#include<iostream>
```

```
using namespace std;
#include<cstddef>
class TreeNode
{
public:
    int data;
    TreeNode *left,*right;

    TreeNode(int data)
    {
        this->data=data;
        left=NULL;
        right=NULL;
    }
};
void insert_node(TreeNode*&root,int value)
{
    TreeNode *newnode=new TreeNode(value);
    if(root==NULL)
    {
        root=newnode;
        return;
    }
    if(root->data>value)
    {
        insert_node(root->left,value);
    }
```

```
    else if(root->data<value)
    {
        insert_node(root->right,value);
    }
    return;
}
void print_tree(TreeNode *root)
{
    //preorder

    if(root==NULL)
        return;

    cout<<root->data<<' ';
    print_tree(root->left);
    print_tree(root->right);

    return;
}
TreeNode*minimum(TreeNode*node)
{
    //to get inorder successor
    if(node->left==NULL)
        return node;
    TreeNode*small=minimum(node->left);
    if(node->data>small->data)
        return small;
```

```
   else
      return node;
}
TreeNode*delete_node(TreeNode*root,int data)
{
   cout<<endl;
   if(root->data>data)
   {
      root->left= delete_node(root->left,data);   //changes occur in left part
      return root;
}
   else if(root->data<data)
   {
      root->right=delete_node(root->right,data);   //changes occur in right part
      return root;
   }
   else
   {
      //if root node has to delete
      if(root->left==NULL&&root->right==NULL)   //after deletion tree becomes empty
root=null
         return NULL;

      else if(root->left==NULL)
      {

         TreeNode*temp=root->right;        //preserving root->right
         delete root;                    //deleting root
```

```
            return temp;                      //temp will return as a new root
        }
        else if(root->right==NULL)
        {
            TreeNode*temp=root->left;         //preserving root->left
            delete root;                      //deleting root
            return temp;                      //temp will return as a new root
        }
        else
        {
            TreeNode*temp=minimum(root->right);//finding inorder successor
            root->data=temp->data;
            root->right=delete_node(root->right,temp->data);  //recursion delete_node(root
->right,temp->data)
            return root;
        }
    }


}


int main()
{
    TreeNode*root=NULL;
    insert_node(root,30);
    insert_node(root,15);
    insert_node(root,40);
    insert_node(root,10);
    cout<<"preorder of tree is:"<<endl;
```

```
    print_tree(root);


    root=delete_node(root,40);

    cout<<"tree after deleting node having value 40 will be:"<<endl;

    print_tree(root);

    /*root=delete_node(root,30);

    print_tree(root);*/


    return 0;



}
```

<u>OUTPUT:</u>

```
preorder of tree is:
30 15 10 40

tree after deleting node having value 40 will be:
30 15 10
Process returned 0 (0x0)   execution time : 0.154 s
Press any key to continue.
```

<u>PRACTICAL 7:</u>

Given the root of a binary tree, return the maximum height of the tree. A binary tree's maximum height is the number of nodes along the longest path from the root node down to the farthest leaf node.

<u>ALGORITHM:</u>

I. IF(ROOT=NULL)THEN RETURN 0

II. HEIGHT LEFT=GET_HEIGHT(ROOT->LEFT)+1;

III. HEIGHT RIGHT=GET_HEIGHT(ROOT->RIGHT)+1;

IV. RETURN MAXIMUM OF THEM.

<u>TIME COMPLEXITY:</u>

O(N)

INPUT TREE: 30,15,40,10 NODES OF TREE

EXPECTED OUTPUT: HEIGHT OF TREE WILL BE 3.

CODE:

```
#include<iostream>
using namespace std;
#include<cstddef>
class TreeNode
{
public:
   int data;
   TreeNode *left,*right;

   TreeNode(int data)
   {
      this->data=data;
      left=NULL;
      right=NULL;
   }
};
void insert_node(TreeNode*&root,int value)    //making tree
{
   TreeNode *newnode=new TreeNode(value);
   if(root==NULL)
   {
      root=newnode;
      return;
```

```
    }
    if(root->data>value)
    {
        insert_node(root->left,value);
    }
    else if(root->data<value)
    {
        insert_node(root->right,value);
    }
    return;
}
int get_height(TreeNode*root)
{
    if(root==NULL)   // to avid runtime eror.
        return 0;
    int heightleft=get_height(root->left)+1;    //heigt of left subtree
    int heightright=get_height(root->right)+1;   //height of right subtree
    return (heightleft>heightright?heightleft:heightright);   //return max of both height
}
int main()
{
    TreeNode*root=NULL;
    insert_node(root,30);
    insert_node(root,15);
    insert_node(root,40);
        insert_node(root,10);
        cout<<"height of tree will be:"<<get_height(root)<<endl;
```

return 0;

}

OUTPUT:

```
height of tree will be:3

Process returned 0 (0x0)   execution time : 2.387 s
Press any key to continue.
```

PRACTICAL 8:

Find maximum and minimum of array using the dynamic programing.

ALGORITHM:

I. TAKE AN INPUT ARRAY OF SIZE AN FROM THE USER.

II. INITIALISE 2 ARRAYS THAT STORE MAX AND MIN VLAUE TILL INDEX.

III. INITIALISE FIRST ELEMENT OF BOT ARRAYS WITH FIRST ELEMENT OF INPUT ARRAY.

IV. FOR I=1 TO N DO

IF ARR[I]>MAX_ARR[I-1] THEN MAX_ARR[I]=ARR[I]

ELSE MAX_ARR[I]=MAX_ARR[I-1].

V. IF ARR[I]<MIN_ARR[I-1] THEN MIN_ARR[I]=ARR[I]

ELSE MIN_ARR[I]=MIN_ARR[I-1].

VI. MAX AND MIN ELEMENTS ARE FOUND AT LAST INDEX OF MAX_ARRAY AND MIN_ARRAY RESPECTIVELY.

TIME COMPLEXITY: O(N)

INPUT: ARRAY IS 1 45 54 71 76 12

EXPECTED OUTPUT: Array: 1 45 54 71 76 12

Min Element = 1

Max Element = 76

CODE:

```
#include<iostream>
#include<vector>

using namespace std;

int main(){
    int n,i;
    cout<<"enter number of elements: ";
    cin>>n;

    int input[n];

    cout<<"enter the elements:"<<endl;
    for(i=0;i<n;i++)
    {
        cin>>input[i];
    }

    int max_arr[n];
    int min_arr[n];

    max_arr[0]=input[0];
    min_arr[0]=input[0];

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

```
        if(input[i]>max_arr[i-1])
        {
            max_arr[i]=input[i];
        }
        else
        {
            max_arr[i]=max_arr[i-1];
        }

        if(input[i]<min_arr[i-1])
        {
            min_arr[i]=input[i];
        }
        else
        {
            min_arr[i]=min_arr[i-1];
        }
    }
    cout<<"minimum number of array : "<<min_arr[n-1]<<endl;
    cout<<"maximum number of array : "<<max_arr[n-1]<<endl;

    return 0;
}
```

OUTPUT:

```
Array: 1 45 54 71 76 12
Min Element = 1
Max Element = 76
```

PRACTICAL 9:

Given a set of N jobs where each job i has a deadline and profit associated with it. Each job takes 1 unit of time to complete and only one job can be scheduled at a time. We earn the profit associated with the job if and only if the job is completed by its deadline. Find the number of jobs done and the maximum profit.

ALGORITHM:

I. SORT THE JOBS ACC TO DEADLINES.

II. CALCULATE MAX DEADLINE AMONG ALL DEADLINES AND CREATE ARRAY TIMESLOT OF SIZE MAXDEADLINE AND INITIALISE IT WITH -1.

III. USING LOOPS FIND THE JOB HAVING MIN DEADLINE AND MAX PROFIT.AFTER THIS PUT THE INDEX OF THAT JOB IN TIMESLOT ARRAY.RUN LOOPS UNTIL ALL SENITEL VALUES HAVE BEEN CHANGED.

IV. ADD PROFIT OF ALL JOBS AND GET THE FINAL RESULT.

TIME COMPLEXITY :$O(N^2)$

INPUT: job j[5] = {

```
    {"j1",2,60},
    {"j2",1,100},
    {"j3",3,20},
    {"j4",2,40},
    {"j5",1,20}
    };
```

EXPECTED OUTPUT: NO. OF JOBES =3 AND PROFIT =180

CODE:

```
#include<iostream>
Using namespace std;
#define MAX 50

class job
{
   char id[5];
   int dead_line;
   int profit;
};
void job_arrange(job j[],int);
int min(int x,int y)
```

```
{
    if(x>y)
    return y;
    return x;
}

int main()
{
    class job j[5] = {
        {"j1",2,60},
        {"j2",1,100},
        {"j3",3,20},
        {"j4",2,40},
        {"j5",1,20}
    };
    int n=5;
     Job* temp;
    for (int i = 0; i < n; i++)
    {
        for (int k = 0; k < n-i-1; k++)
        {
            if(j[k+1].profit>j[k].profit)
            {
                temp = j[k+1];
                j[k+1] = j[k];
                j[k] = temp;
            }
        }
    }
   Cout<< "Job"<<','<< "Deadline"<<','<< "Profit";
    for(int i = 0; i < n; i++) {
   Cout<< j[i].id<<','<< j[i].dead_line<<','<< j[i].profit;
    }
    job_arrange(j,n);
}
void job_arrange(struct job j[],int n)
{
    int i,k,max_profit;
    int TimeSlot[MAX];

    int FilledSlot = 0;
    int dmax = 0;
    for ( i = 0; i < n; i++)
    {
        if(j[i].dead_line>dmax)
```

```
        dmax=j[i].dead_line;
    }
    for ( i = 1; i <= dmax ; i++)
    {
        TimeSlot[i] = -1;
    }
    Cout<<endl;
    for ( i = 1; i <= n ; i++)
    {
        k=min(dmax,j[i-1].dead_line);
        while(k>=1)
        {
            if(TimeSlot[k]==-1)
            {
              TimeSlot[k]=i-1;
              FilledSlot++;
              break;
            }
            k--;
        }
        if(FilledSlot==dmax)
        break;
    }
    Cout<<"Required Jobs: ";
    for(i = 1; i <= dmax; i++) {
    Cout<< j[TimeSlot[i]].id);

    if(i < dmax) {
      Cout<<",";
    }
  }
  max_profit=0;
  for ( i = 1; i <= dmax; i++)
  {
     max_profit += j[TimeSlot[i]].profit;
  }
  Cout<<"maximum profit ="<<max_profit;
}
```

OUTPUT:

"C:\C++ tutorials\sa.exe"

```
      Job        Deadline      Profit
       j2           1          100
       j1           2           60
       j4           2           40
       j3           3           20
       j5           1           20

Required Jobs: j2->j1->j3
maximum profit = 180
Process returned 0 (0x0)   execution time : 0.025 s
Press any key to continue.
```

PRACTICAL 10:

Given weights and values of N items, we need to put these items in a knapsack of capacity W to get the maximum total value in the knapsack. Note: Unlike 0/1 knapsack, you are allowed to break the item

ALGORITHM:

I. MAKE MATRIX OF SIZE OUTPUT[N+1][W+1].
II. START FILLING THE ARRAY.
III. IF(I==0||J==0) THEN OUTPUT[I][J]=0;
IV. ELSE IF WT[I-1]<=W THEN OUTPUT[I][W]=MAXIMUM OF (VAL[I-1]+OUTPUT[I-1][W-WT[I-1]],OUTPUT[I-1][W]).
V. ELSE
VI. OUTPUT[I][W]=OUTPUT[I-1][W];
VII. FINAL RESULT IS STORED AT OUTPUT[N][W],
VIII. END;

INPUT: VAL[]={60,10,50,45);WT[]={10,13,45,63},W=60
EXPECTED OUTPUT: 110

TIME COMPLEXITY: $O(2^N)$
CODE:

```
#include <iostream>
using namespace std;
int max(int a,int b)
{
   if(a>b)
   return a;
   else
   return b;
```

```
}

int knapsack(int W,int wt[],int val[],int n)
{
   int output[n+1][W+1];
    for (int i = 0; i <= n; i++)
   {
      for (int w = 0; w <= W; w++)
      {
         if (i == 0 || w == 0)
            output[i][w] = 0;
         else if (wt[i - 1] <= w)
            K[i][w] = max(val[i - 1]
                    + output[i - 1][w - wt[i - 1]],
                    output[i - 1][w]);
         else
            output[i][w] = output[i - 1][w];
      }
   }
   return output[n][W];
}

int main() {

   int val[] = {  60, 10, 50,45};
   int wt[] = {10,13,45,63 };
   int W = 60;
   int n = sizeof(val) / sizeof(val[0]);

   cout<<knapsack(W, wt, val, n);


   return 0;
}
```
OUTPUT:

"C:\Users\prikshit juneja\Documents\file.exe"

```
110
Process returned 0 (0x0)   execution time : 0.232 s
Press any key to continue.
```

PRACTICAL 11:

Write a program for the fractional and dynamic knapsack problem.

ALGORITHM:

I. RECURSIVELY ITERATE THROUGH THE ARRAY
II. EITHER TAKE THE ELEMENT OR SKIP ,

TIME COMPLEXITY: $O(2^N)$

INPUT: VAL[]={60,10,50,45);WT[]={10,13,45,63},W=60
EXPECTED OUTPUT: 110

CODE:

```
#include <iostream>
using namespace std;
int knapsack(int value[], int wt[], int cap, int n) {
  if (cap == 0 || n == 0) {
    return 0;
  }
  if (cap - wt[n - 1] < 0) {
    return knapsack(value, wt, cap, n - 1);
  }
  else {
    return max(value[n - 1] + knapsack(value, wt, cap - wt[n - 1], n - 1), knapsack(value,  wt,
cap, n - 1));
  }
}
int main() {

  int value[n], wt[n];
      int val[] = {  60, 10, 50,45};
        int wt[] = {10,13,45,63 };
```

```
    int W = 60;
    int n = sizeof(val) / sizeof(val[0]);



  cout <<  knapsack(value, wt, cap, n)  << "." << endl;
  return 0;
}
```

OUTPUT:

```
"C:\Users\prikshit juneja\Documents\file.exe"
110
Process returned 0 (0x0)   execution time : 0.232 s
Press any key to continue.
```

PRACTICAL 12:

Implement Minimum Spanning trees: Prim's algorithm and Kruskal's algorithm Input a directed graph G = (V, E) where vertices V are represented as alphabetical numbers. Run the DFS-based topological ordering algorithm on the graph. Whenever you have a choice of vertices to explore, always pick the one that is alphabetically first. Let G = (V, E) be a directed graph. Vertices u and v are strongly connected if there are u v and v u paths in G. A strongly connected component is a set of vertices C ⊆ V such that u, v is strongly connected for all u, v ∈ C (and no other vertices are strongly connected to a vertex u ∈ C.) Design an algorithm to identify all strongly connected components OF G.

PRIMS ALGORITHM:

I. FIRST OF ALL REMOVE THE LOOP EDGES .AFTER THAT REMOVE THE PARALLEL EDGES HAVING MAX WT.

II. SELECT ANY VERTICES AS A STARING VERTEX.AFTER THAT CHOOSE THE VERTEX HAVING MIN WEIGHT .

III. AFTER THAT FROM THE NEW VERTEX CHOOS ETHE

VERTEX HAVING MIN WT .OPTIONS MUST INCLUDE THE EDGES LEFT FROM ALL THE PREVIOUS VISITED NODE.

IV. REPEAT THE PROCESS UNTIL ALL VERTICES GET PASSED.

V. FINALLY THE OBTAINED TREE IS THE MIN SPANNING TREE.

TIME COMPLEXITY: O((V+E)LOGV).//V->VERTICES,E->EDGES.

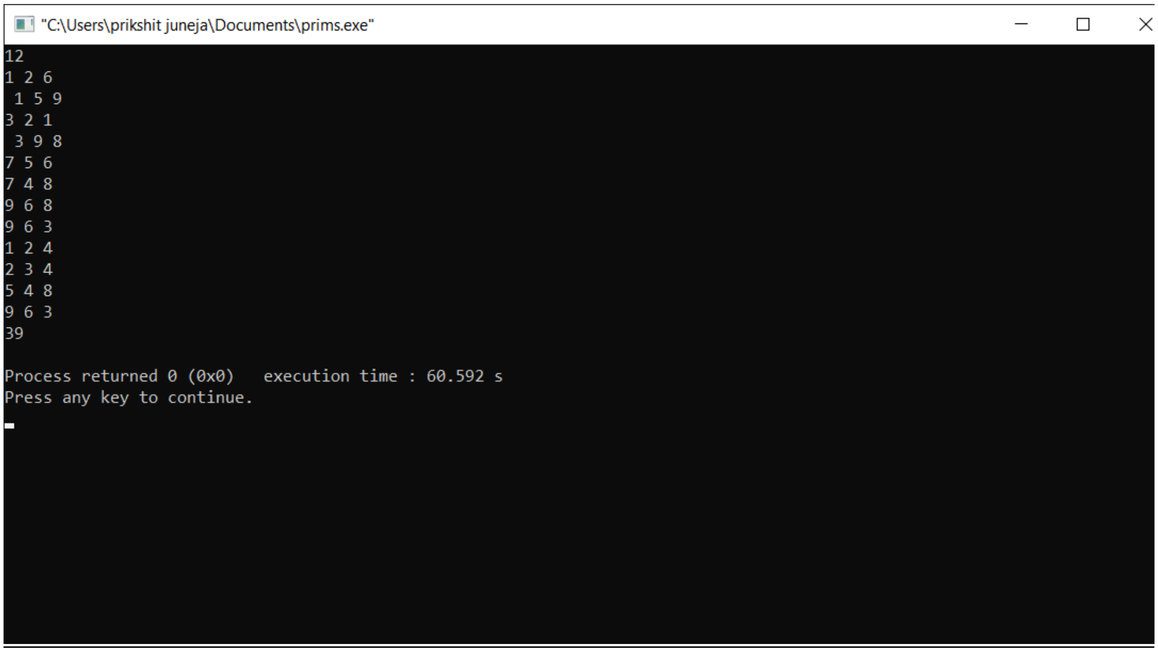
CODE:

```
#include <iostream>

#include <vector>

#include <queue>

using namespace std;

class Graph {

  vector<pair<int, int>> *l;

  int v;

  public:

  Graph(int n) {

     v = n;

     l = new vector<pair<int, int>> [n];

  }

  void addEdge (int x, int y, int w) {

     l[x].push_back({y, w});

     l[y].push_back({x, w});

  }
```

```
int prim_mst() {
    //Initialise a min heap
    priority_queue<pair<int, int>, vector<pair<int, int>>, greater<pair<int, int>>> q;
    //visited array that denotes whether a node has been included in MST or not
    bool *vis = new bool[v]{0};
    int ans = 0;
    q.push({0, 1});     //weight, node
    while (!q.empty()) {
        //pick out the edge with min weught
        auto best = q.top();
        q.pop();
        int to = best.second;
        int weight = best.first;
        if (vis[to]) {
            //discard the edge and continue
            continue;
        }
        //otherwise take the current edge
        ans += weight;
        vis[to] = 1;
        //add the new edges into the queue
        for (auto x: l[to]) {
            if (vis[x.first] == 0) {
```

```
                q.push({x.second, x.first});
            }
        }
    }
    return ans;
  }
};
int main() {
    int v = 0;
    cin >> v;
    Graph g(v);
    for (int i = 0; i < v; i++) {
        int x = 0, y = 0, w = 0;
        cin >> x >> y >> w;
        g.addEdge(x, y, w);
    }
    cout << g.prim_mst() << endl;
}
```

OUTPUT:



KRUSKAL ALGORITHM:

I. REMOVE ALL THE LOOPS EDGES AND PARALLEL EDGES HAVING MORE WT.

II. SORT THE EDGES IN INCREASING ODER ACC TO THEIR EDGE WT.

III. FIRST CHOOSE THE EDGE HAVING MIN WT.AN D REPEAT THIS .IF CYCLE IS FORMED BY INCLUDING ANY EDGE THEN SKIP THIS EDGE.

IV. KEEP TRAVERSING UNTIL ALL THE NODES ARE VISITED ONCE.

V. END.

CODE:

```
#include <iostream>

#include <algorithm>

#include <vector>
```

```
using namespace std;
class DSU {
  int *parent;
  int *rank;
  public:
  DSU (int n) {
    parent = new int[n];
    rank = new int[n];
    for (int i = 0; i < n; i++) {
       parent[i] = -1;
       rank[i] = 1;
    }
  }
  int find(int i) {
    if (parent[i] == -1) {            //base case
      return i;
    }
    return parent[i] = find(parent[i]);
  }
  void unite(int x, int y) {
    int s1 = find(x);
    int s2 = find(y);
    if (s2 != s2) {
```

```
            if (rank[s1] < rank[s2]) {
              parent[s1] = s2;
              rank[s2] += rank[s1];
            }
            else {
              parent[s2] = s1;
              rank[s1] += rank[s2];
            }
          }
        }
      };
      class Graph {
        vector<vector<int>> edgelist;        //weight, x, y
        int v;
        public:
        Graph(int n) {
           v = n;
        }
        void addEdge (int x, int y, int w) {
           edgelist.push_back({w, x, y});
        }
        int kruskals_mst() {
          //Sort all edges based on weight
```

```
    sort(edgelist.begin(), edgelist.end());
    DSU s(v);
    int ans = 0;
    for (auto edge : edgelist) {
      int w = edge[0];
      int x = edge[1];
      int y = edge[2];
      //take that edge in MST if it doesn't form a cycle
      if (s.find(x) != s.find(y)) {
        s.unite(x, y);
        ans += w;
      }
    }
    return ans;
  }
};
int main() {
  Graph g(4);
  g.addEdge(0, 1, 10);
  g.addEdge(1, 3, 15);
  g.addEdge(2, 3, 4);
  g.addEdge(2, 0, 6);
  g.addEdge(0, 3, 5);
```

```
  cout << g.kruskals_mst() << endl;

}
```

OUTPUT:

```
19
```

PRACTICAL 13:

Given a sequence of matrices, find the most efficient way to multiply these matrices together. The efficient way is the one that involves the least number of multiplications. The dimensions of the matrices are given in an array arr[] of size N (such that N = number of matrices + 1) where the ith matrix has the dimensions (arr[i-1] x arr[i]).

Input:

6 (Number of matrices, followed by matrix size)

2 4

4 3

3 6

6 5

5 2

2 1

EXPECTED OUTPUT:

Number of min operations are:78

ALGORITHM:

I. Create a recursive function that takes i and j as parameters that determines the range of a group.

II. Iterate from k = i to j to partition the given range into two groups.

III. Call the recursive function for these groups.

IV. Return the minimum value among all the partitions as the required minimum number of multiplications to multiply all the matrices of this group.

V. The minimum value returned for the range 0 to N-1 is the required answer.

TIME COMPLEXITY: The time complexity of the solution is exponential

Auxiliary Space: O(1)

CODE:

```
#include<iostream>

#include<bits/stdc++.h>

using namespace std;

int tdp[100][100];//dynamic top down approach

int chainmatrixmult(int *matrix,int i,int j)

{

  if(i==j)

    {

      tdp[i][j]=0;   //means single matrix left in array base case for recursion

      return 0;

    }

    if(tdp[i][j]!=-1)

    {

      return tdp[i][j]; // to manage overlapping case

    }

  int ans=INT_MAX;

  for(int k=i;k<j;k++)   // k is a position where we break down the pbm
```

```
  {
     // chainmatrixmult(matrix,i,k) calling recursion for first part
     // chainmatrixmult(matrix,k+1,j) calling recursion for second part
     //matrix[i-1]*matrix[j]*matrix[k] is for calculating steps in product of matrices
     int temp=chainmatrixmult(matrix,i,k)+chainmatrixmult(matrix,k+1,j)+matrix[i-
1]*matrix[j]*matrix[k];
      ans=min(ans,temp);
  }
  tdp[i][j]=ans; //putting no. of steps in table
  return ans;
  }
int main()
{
  int matrix[]={2,4,3,6,5,2,1};
  memset(tdp,-1,sizeof tdp);    //setting all enteries to -1 in table
  int n=sizeof(matrix)/sizeof(int);
  cout<<"number of min operations are : "<<chainmatrixmult(matrix,1,n-1)<<endl;
}
```

output:

```
"C:\Users\prikshit juneja\Documents\matrixchain.exe"
number of min operations are : 78

Process returned 0 (0x0)   execution time : 3.619 s
Press any key to continue.
```

<u>**PRACTICAL 14:**</u>

Write a program to implement Strassen's Matrix Multiplication.

Implement Matrix chain multiplication (MCM) using dynamic

programming, you need to estimate the minimum number of operations

and assign the parentheses for multiplying multiple matrices.

Input:

6 (Number of matrices, followed by matrix size)

2 4

4 3

3 6

6 5

5 2

2 1

Output:

Number of min operations are:78

ALGORITHM:

I. Create a recursive function that takes i and j as parameters that determines the range of a group.

II. Iterate from k = i to j to partition the given range into two groups.

III. Call the recursive function for these groups.

IV. Return the minimum value among all the partitions as the required minimum number of multiplications to multiply all the matrices of this group.

V. The minimum value returned for the range 0 to N-1 is the required answer.

TIME COMPLEXITY: The time complexity of the solution is exponential

Auxiliary Space: O(1)

CODE:

```
#include<iostream>
```

```
#include<bits/stdc++.h>

using namespace std;

int tdp[100][100];//dynamic top down approach

int chainmatrixmult(int *matrix,int i,int j)

{

    if(i==j)

        {

            tdp[i][j]=0;   //means single matrix left in array base case for recursion

            return 0;

        }

        if(tdp[i][j]!=-1)

        {

            return tdp[i][j]; // to manage overlapping case

        }

    int ans=INT_MAX;

    for(int k=i;k<j;k++)   // k is a position where we break down the pbm

    {

        // chainmatrixmult(matrix,i,k) calling recursion for first part

        // chainmatrixmult(matrix,k+1,j) calling recursion for second part

        //matrix[i-1]*matrix[j]*matrix[k] is for calculating steps in product of matrices

        int temp=chainmatrixmult(matrix,i,k)+chainmatrixmult(matrix,k+1,j)+matrix[i-
1]*matrix[j]*matrix[k];

         ans=min(ans,temp);

    }

    tdp[i][j]=ans; //putting no. of steps in table

   return ans;
```

```
  }
int main()
{
   int matrix[]={2,4,3,6,5,2,1};
   memset(tdp,-1,sizeof tdp);    //setting all enteries to -1 in table
   int n=sizeof(matrix)/sizeof(int);
   cout<<"number of min operations are : "<<chainmatrixmult(matrix,1,n-1)<<endl;
}
```

output:

```
"C:\Users\prikshit juneja\Documents\matrixchain.exe"
number of min operations are : 78

Process returned 0 (0x0)   execution time : 3.619 s
Press any key to continue.
```

PRACTICAL 15:

Write a program to implement Longest Common Subsequence.

INPUT:

STRING FIRST: AGGTAB

STRING SECOND: GXTXAYB

EXPECTED OUTPUT:

GTAB

ALGORITHM:

I. Construct arr[len_first+1][len_sec+1] and find the length of longest common subsequence.

II. The value arr[len_first][len_sec+1] contains length of LCS. Create a character array output[] of length equal to the length of lcs plus 1 (one extra to store \0).

III. Traverse the 2D array starting from arr[len_first][len_sec]. Do following for every cell arr[i][j]

IV. If characters (in first and second) corresponding to arr[i][j] are same (Or first[i-1] == second[j-1]), then include this character as part of LCS.

V. Else compare values of arr[i-1][j] and arr[i][j-1] and go in direction of greater value.

CODE:

```
#include<iostream>

#include<algorithm>

#include<cstring>

#include<cstdlib>

using namespace std;

void leastcommonsubs(string first,string second,int len_first,int len_sec)

{

   int arr[len_first+1][len_sec+1];   //MAKING 2D ARRAY

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

       {

       for(int j=0;j<=len_sec;j++)  //FILLING THE  2D ARRAY  i FOR FIRST STRING AND J
FOR SECOND STRING

      {

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

            arr[i][j]=0;        //FIRST ROW AND COLUMN WILL BE 0

         else if(first[i-1]==second[j-1])

            arr[i][j]=arr[i-1][j-1]+1;  //IF CHAR AT I POS=CHAR AT J POS THEN DIAGONALLY
ELEMENT WILL BE TAKEN

         else

            arr[i][j]=max(arr[i-1][j],arr[i][j-1]); //ELSE MAX OF CHAR AT I & J WILL BE TAKEN

      }

   }
```

```
   int subs_len=arr[len_first][len_sec];   //LENGTH OF LONGESTCOMMON
SUBSEQUENCE

        int length=subs_len;

        char output[subs_len+1];   //MAKING OUTPUT ARRAY FOR LONGEST COMMON
SUBS

        output[subs_len]='\0';   //FILLING FROM THE LAST

        int i=len_first;

        int j=len_sec;

         while(i>0&&j>0)

        {

         if(first[i-1]==second[j-1])

        {

output[subs_len-1]=first[i-1];  // CASE WHERE WE GO DIAGONALLY

           i--;

           j--;

           subs_len--;

        }

_else if(arr[i-1][j]>arr[i][j-1])

           i--;

        else

           j--;

    }


_  //DISPLAY OUTPUT

   cout<<"least common subsequence having length : "<<length<<" is : "<<output;


}
```

```
int main()
{
    string first;
    string second;
    cout<<"enter your first string:"<<endl;
    getline(cin,first);    //TAKING INPUT FOR FIRST STRING
    cout<<"enter your second string:"<<endl;
    getline(cin,second);   //TAKING INPUT FOR SECOND STRING

    int len_first=first.length();    //FINDING FIRST LENGTH
    int len_sec=second.length();     //FINDING SECOND LENGTH
    leastcommonsubs(first,second,len_first,len_sec);   //CALLING FUNCTION

    return 0;
}
```

OUTPUT:

```
enter your first string:
AGGTAB
enter your second string:
GXTXAYB
least common subsequence having length : 4 is : GTAB
Process returned 0 (0x0)   execution time : 64.151 s
Press any key to continue.
```

PRACTICAL 16:

Implement Travelling Salesman Problem.

INPUT:

NO.OF CITIES =4

DISTANCE IN MATRIX FORM WILL

BE:{{0,20,42,25},{20,0,30,34},{42,30,0,10},{25,34,10,0}}

EXPECTED OUTPUT:

shortest distance for salesman is : 85

ALGORITHM:

I. SET MASK AS 1 AND PASS IT IN FUNCTION ALONG WITH POSITION=0.MASK=1 INDICATES THAT FIRST CITY IS ALREADY VISITED.

II. IF MASK=7 MEANS THAT ALL CITIES ARE VISITED AND WE HAVE TO RETURN DISTANCE FROM CURRENT CITY TO STARTING CITY HAVING POS 0.THIS IS BASE CASE FOR RECURSION.

III. TO AVOID OVERLAPPING SUBPARTS APPLY CHECK ON dp[mask][pos].IF IT IS NOT EQUAL TO -1 MEANS CITY IS ALREADY VISITED AND RETURN dp[mask][pos].

IV. IF CITY IS NOT VISITED THEN NEW ANS CALCULATED BY APPLYING RECURSION TO NEXT CITY PLUS DIST[POS][CITY].

V. THEN FINALLY RETURN THE SMALLEST OF ALL ANSWERS AND IT IS OUR FINAL ANSWER.

TIME COMPLEXITY: $O(n^{2}*2_{n})$.

CODE:

```
#include<iostream>

#include<algorithm>

using namespace std;

#define INT_MAX 999999

int n=4;

int dp[16][4];

int dist[10][10]={{0,20,42,25},{20,0,30,34},{42,30,0,10},{25,34,10,0}};//matrix
defines the graph


int visited_all=(1<<n)-1;//if all cities have been visited i.e. setting mask


//travelling salesman pbm function
```

```
int tsp(int mask,int pos)//mask tells us about cities visited and pos tell abour
current city

{

   if(mask==visited_all)

      return dist[pos][0];   //base case to hit recursion


      if(dp[mask][pos]!=-1)

         return dp[mask][pos];  //if node already visited


   int ans=INT_MAX;


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

   {

      if((mask&(1<<city))==0) //city is not visited

      {

                                  //setting mask

         int new_ans=dist[pos][city]+tsp(mask|(1<<city),city);

         ans= min(ans,new_ans);

      }

   }

   return  dp[mask][pos]=ans;

}



int main()

{

   for(int i=0;i<(1<<n);i++)
```

```
    {
        for(int j=0;j<n;j++)
        {
            dp[i][j]=-1;
        }
    }
    cout<<"shortest distance for salesman is : "<<tsp(1,0)<<endl;
    return 0;

}
```

OUTPUT:

```
shortest distance for salesman is : 85

Process returned 0 (0x0)   execution time : 0.644 s
Press any key to continue.
```