1. AIM: To implement following algorithm using as a data structure and analyse their time complexity.

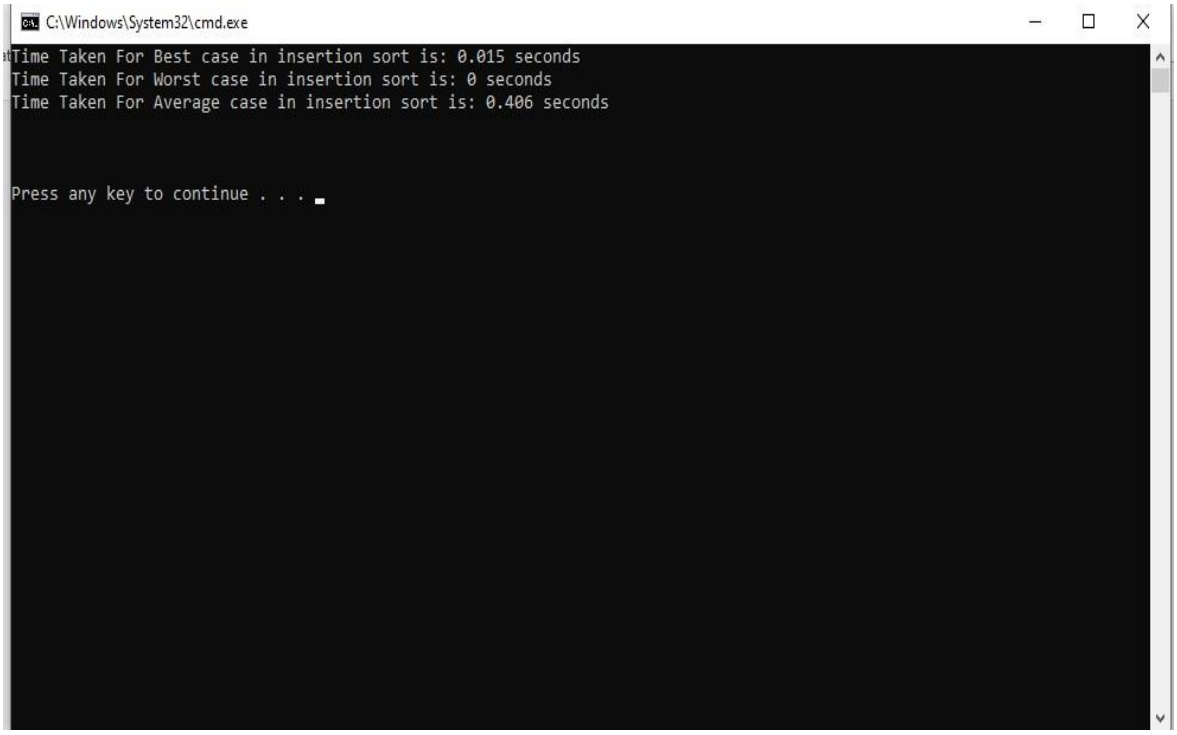
A) INSERTION SORT

```
#include <iostream>
#include <algorithm>
#include <bits/stdc++.h>
using namespace std; int
mycompare(int a,int b){
return a>b;
}
void MakeCases(int *BestCase,int *WorstCase,int *AverageCase,int n){
for (int i = 0; i < n; ++i)
{
//average case consist of random variables
AverageCase[i]=rand();
}
for (int i = 0; i < n; ++i)
{
//average case consist of random variables
BestCase[i]=AverageCase[i];
WorstCase[i]=AverageCase[i];
}
//sorted in ascending order
sort(BestCase,BestCase+n); //sorted in
descending order
```

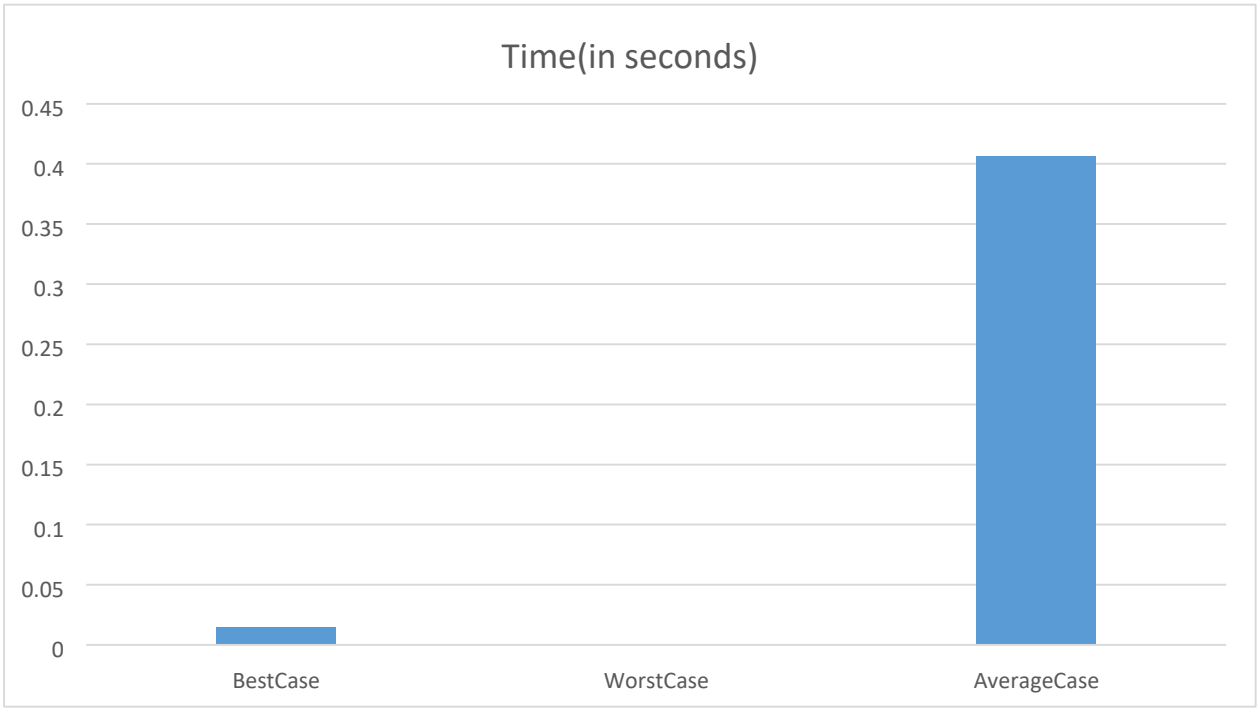
```
sort(WorstCase,WorstCase+n,mycompare);
}

void InsertionSort(int arr[], int n)
{ int i, key, j; for (i
= 1; i < n; i++)
{
key = arr[i]; j
= i - 1;

/* Move elements of arr[0..i-1], that are
greater than key, to one position ahead
of their current position */ while (j >= 0
&& arr[j] > key)
{ arr[j + 1] =
arr[j]; j = j - 1;
}
arr[j + 1] = key;
}
}
int main(int argc, char const *argv[])
{

int n=10000; int
*BestCase=new int [n]; int
*WorstCase=new int [n]; int
*AverageCase=new int [n];
```

```
MakeCases(BestCase,WorstCase,AverageCase,n); clock_t
time_req; time_req = clock(); InsertionSort(BestCase,n);
cout<<"Time Taken For Best case in insertion sort is:
"<<(float)time_req/CLOCKS_PER_SEC << " seconds" << endl;
time_req = clock() - time_req; InsertionSort(WorstCase,n);
cout<<"Time Taken For Worst case in insertion sort is:
"<<(float)time_req/CLOCKS_PER_SEC << " seconds" << endl;
time_req = clock() - time_req; InsertionSort(AverageCase,n);
cout<<"Time Taken For Average case in insertion sort is:
"<<(float)time_req/CLOCKS_PER_SEC << " seconds" << endl;;
cout<<endl; return 0;
}
```

OUTPUT:



Graph:



B) SELECTION SORT

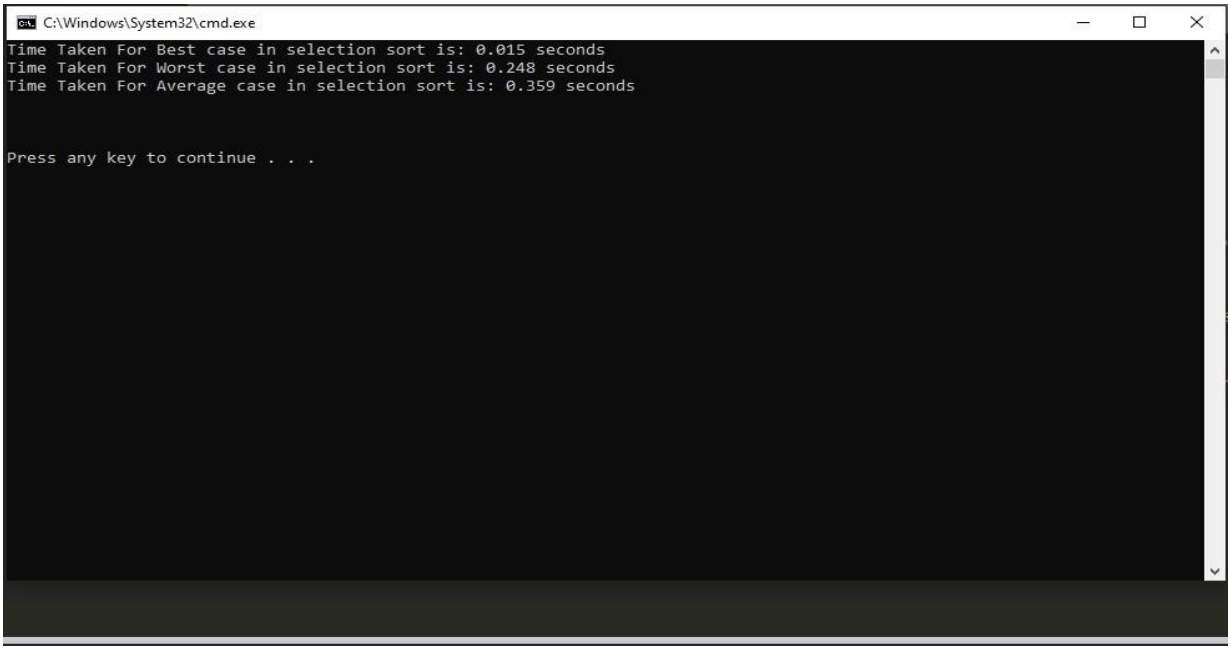
#include <iostream>

```
#include <algorithm>
#include <bits/stdc++.h>
using namespace std; int
mycompare(int a,int b){
        return a>b;
}
void MakeCases(int *BestCase,int *WorstCase,int *AverageCase,int n){
for (int i = 0; i < n; ++i)
        {
                //average case consist of random variables
                AverageCase[i]=rand();
        }
        for (int i = 0; i < n; ++i)
        {
                //average case consist of random variables
                BestCase[i]=AverageCase[i];
                WorstCase[i]=AverageCase[i];
        }
        //sorted in ascending order
sort(BestCase,BestCase+n);      //sorted in
descending order
sort(WorstCase,WorstCase+n,mycompare); }

void swap(int *xp, int *yp)
{
```

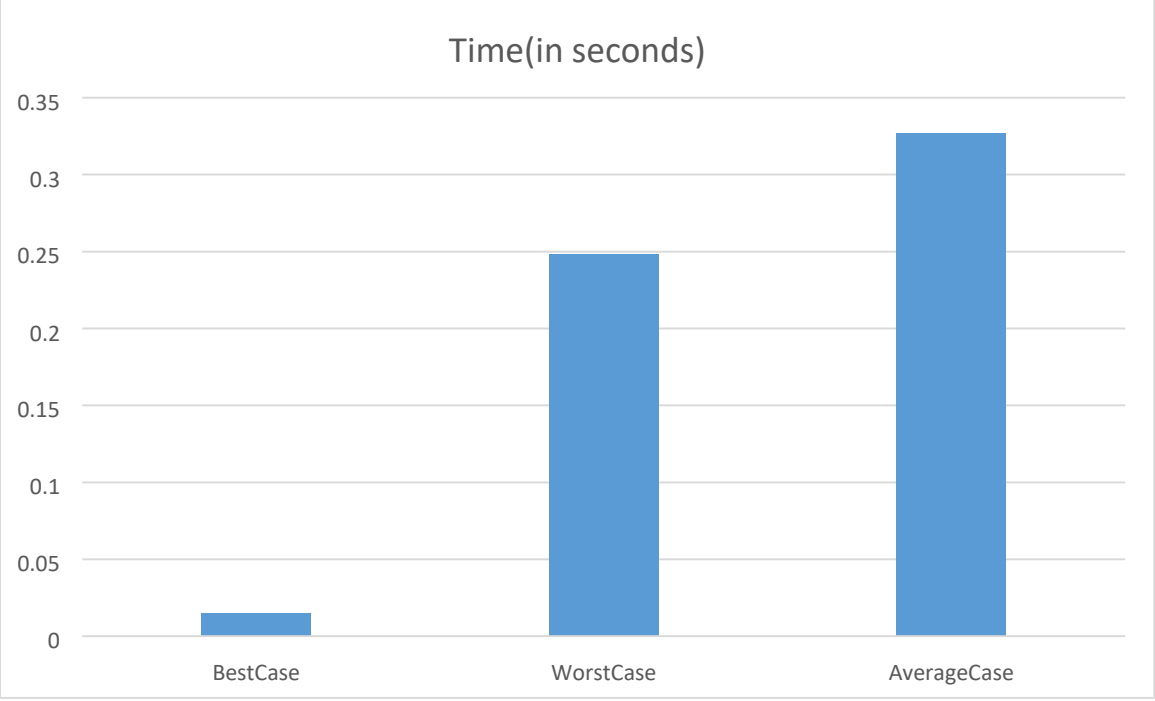
```
    int temp = *xp;
*xp = *yp;
    *yp = temp;
}

void SelectionSort(int arr[], int n)
{
    int i, j, min_idx;

    // One by one move boundary of unsorted subarray
    for (i = 0; i < n-1; i++)
    {
        // Find the minimum element in unsorted array
min_idx = i;          for (j = i+1; j < n; j++)          if
(arr[j] < arr[min_idx])              min_idx = j;

        // Swap the found minimum element with the first element
swap(&arr[min_idx], &arr[i]);
    }
}
int main(int argc, char const *argv[])
{

          int n=10000;          int
*BestCase=new int [n];    int
```

```
*WorstCase=new int [n];          int
*AverageCase=new int [n];


        MakeCases(BestCase,WorstCase,AverageCase,n);
clock_t time_req;           time_req = clock();
SelectionSort(BestCase,n);       cout<<"Time Taken For Best
case in insertion sort is: "<<(float)time_req/CLOCKS_PER_SEC
<< " seconds" << endl;     time_req = clock() - time_req;
SelectionSort(WorstCase,n);      cout<<"Time Taken For Worst
case in insertion sort is: "<<(float)time_req/CLOCKS_PER_SEC
<< " seconds" << endl;     time_req = clock() - time_req;
SelectionSort(AverageCase,n);  cout<<"Time Taken For
Average case in insertion sort is:
"<<(float)time_req/CLOCKS_PER_SEC << " seconds" << endl;;
        cout<<endl;
return 0;
}
```

OUTPUT:



Graph:



C) BUBBLE SORT

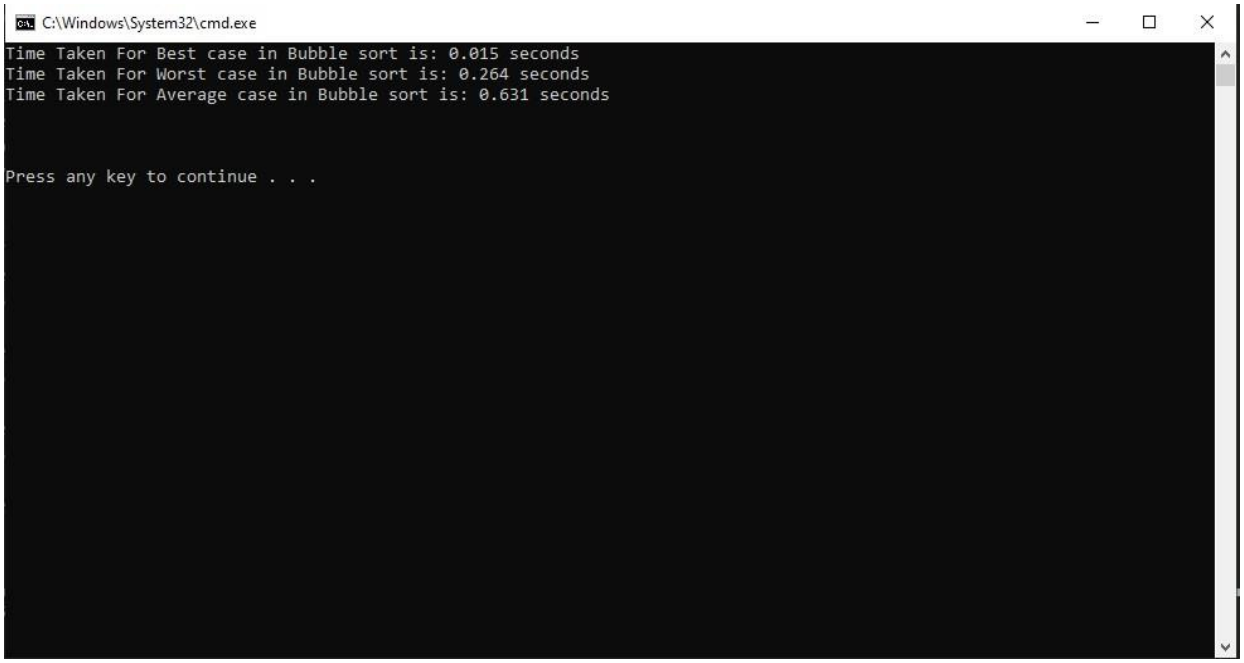
```
#include <iostream>
```

```
#include <algorithm>
#include <bits/stdc++.h>
using namespace std;  int
mycompare(int a,int b){
        return a>b;
}
void MakeCases(int *BestCase,int *WorstCase,int *AverageCase,int n){
for (int i = 0; i < n; ++i)
        {
                //average case consist of random variables
                AverageCase[i]=rand();
        }
        for (int i = 0; i < n; ++i)
        {
                //average case consist of random variables
                BestCase[i]=AverageCase[i];
                WorstCase[i]=AverageCase[i];
        }
        //sorted in ascending order
sort(BestCase,BestCase+n);       //sorted in
descending order
sort(WorstCase,WorstCase+n,mycompare);
}
void swap(int *xp, int *yp)
{
```

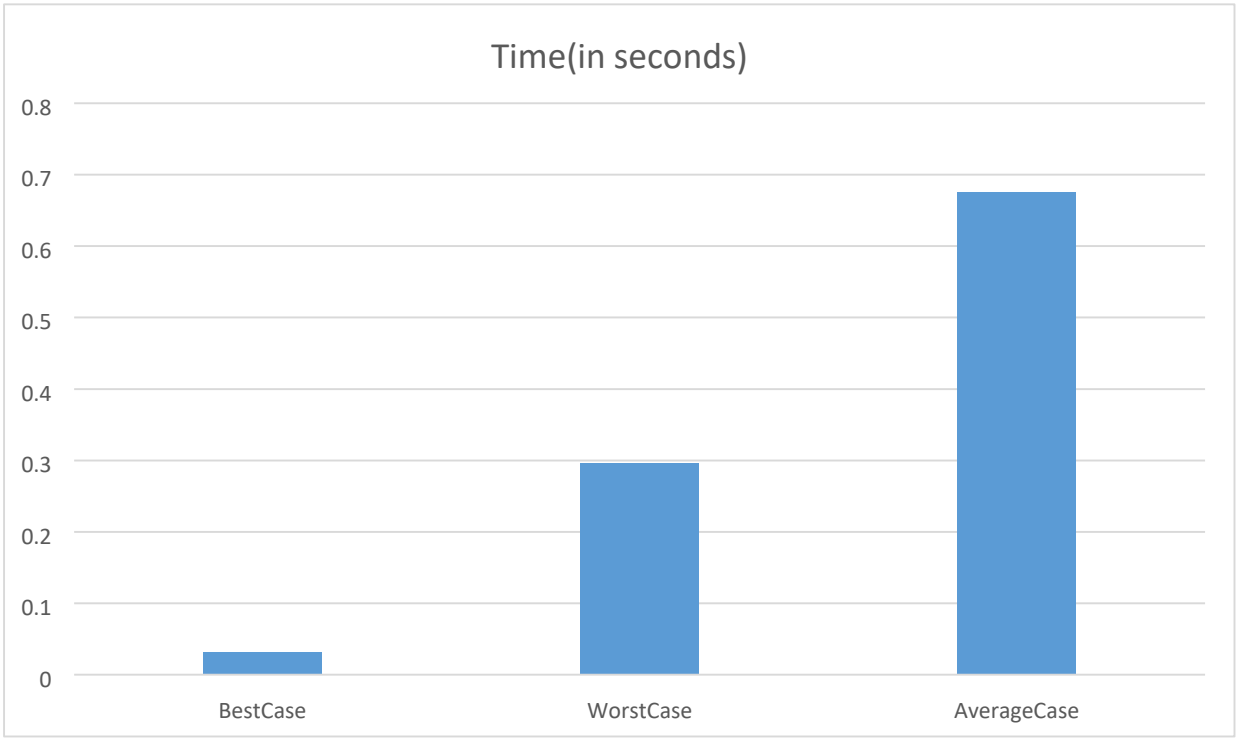
```
   int temp = *xp;
*xp = *yp;
   *yp = temp;
}

// A function to implement bubble sort   void
BubbleSort(int arr[], int n)
{      int i, j;      for (i =
0; i < n-1; i++)

   // Last i elements are already in place
for (j = 0; j < n-i-1; j++)         if (arr[j] >
arr[j+1])            swap(&arr[j],
&arr[j+1]);
}
int main(int argc, char const *argv[])
{

         int n=10000;          int
*BestCase=new int [n];    int
*WorstCase=new int [n];          int
*AverageCase=new int [n];
MakeCases(BestCase,WorstCase,Av
erageCase,n);        clock_t
time_req;     time_req = clock();
BubbleSort(BestCase,n);
```

```
cout<<"Time Taken For Best case in
insertion sort is:
"<<(float)time_req/CLOCKS_PER_SE
C << " seconds" << endl;  time_req =
clock() - time_req;
BubbleSort(WorstCase,n);
cout<<"Time Taken For Worst case
in insertion sort is:
"<<(float)time_req/CLOCKS_PER_SE
C << " seconds" << endl;  time_req =
clock() - time_req;
BubbleSort(AverageCase,n);
cout<<"Time Taken For Average
case in insertion sort is:
"<<(float)time_req/CLOCKS_PER_SEC << " seconds" << endl;;
        cout<<endl;
return 0;
}
```

OUTPUT:



Graph:



D) RADIX SORT

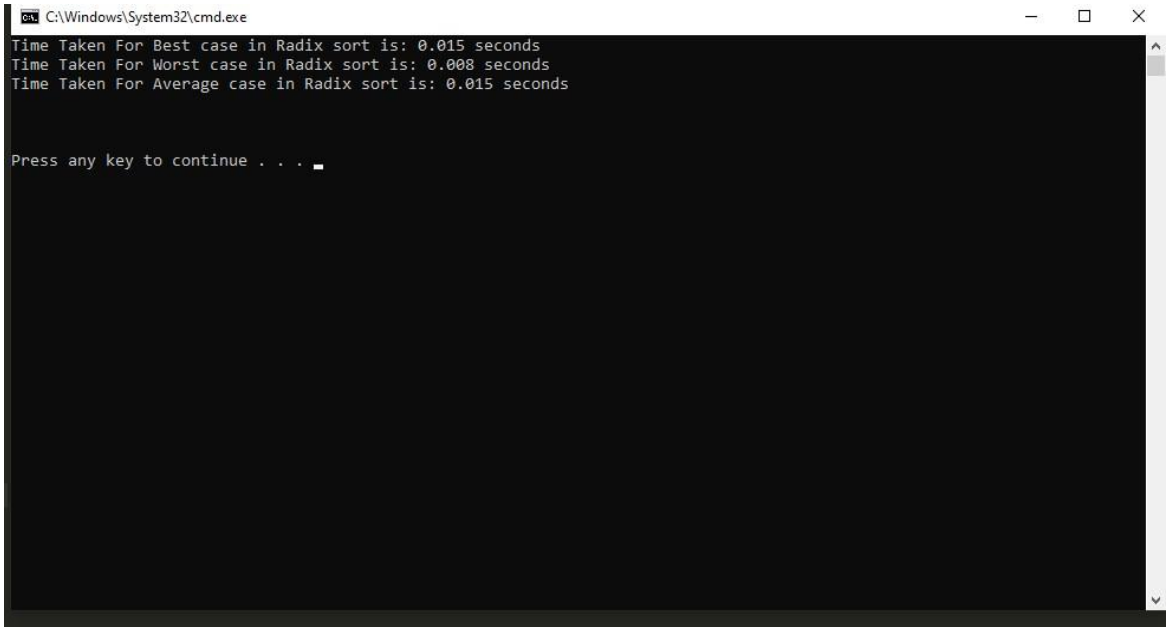
```
#include <iostream>
#include <algorithm>
#include <bits/stdc++.h>
using namespace std; int
mycompare(int a,int b){
        return a>b;
}
void MakeCases(int *BestCase,int *WorstCase,int *AverageCase,int n){
for (int i = 0; i < n; ++i)
        {
                //average case consist of random variables
                AverageCase[i]=rand();
        }
        for (int i = 0; i < n; ++i)
        {
                //average case consist of random variables
                BestCase[i]=AverageCase[i];
                WorstCase[i]=AverageCase[i];
        }
        //sorted in ascending order
sort(BestCase,BestCase+n);      //sorted in
descending order
sort(WorstCase,WorstCase+n,mycompare); }

// A utility function to get maximum value in arr[]  int
getMax(int arr[], int n)
```

```
{
   int mx = arr[0];     for
(int i = 1; i < n; i++)
if (arr[i] > mx)
mx = arr[i];     return
mx;
}
void countSort(int arr[], int n, int exp)
{
   int output[n]; // output array
int i, count[10] = {0};

   // Store count of occurrences in count[]
for (i = 0; i < n; i++)        count[
(arr[i]/exp)%10 ]++;

   // Change count[i] so that count[i] now contains actual
   //  position of this digit in output[]
for (i = 1; i < 10; i++)        count[i] +=
count[i - 1];

   // Build the output array
for (i = n - 1; i >= 0; i--)
   {
      output[count[ (arr[i]/exp)%10 ] - 1] = arr[i];
count[ (arr[i]/exp)%10 ]--;
```

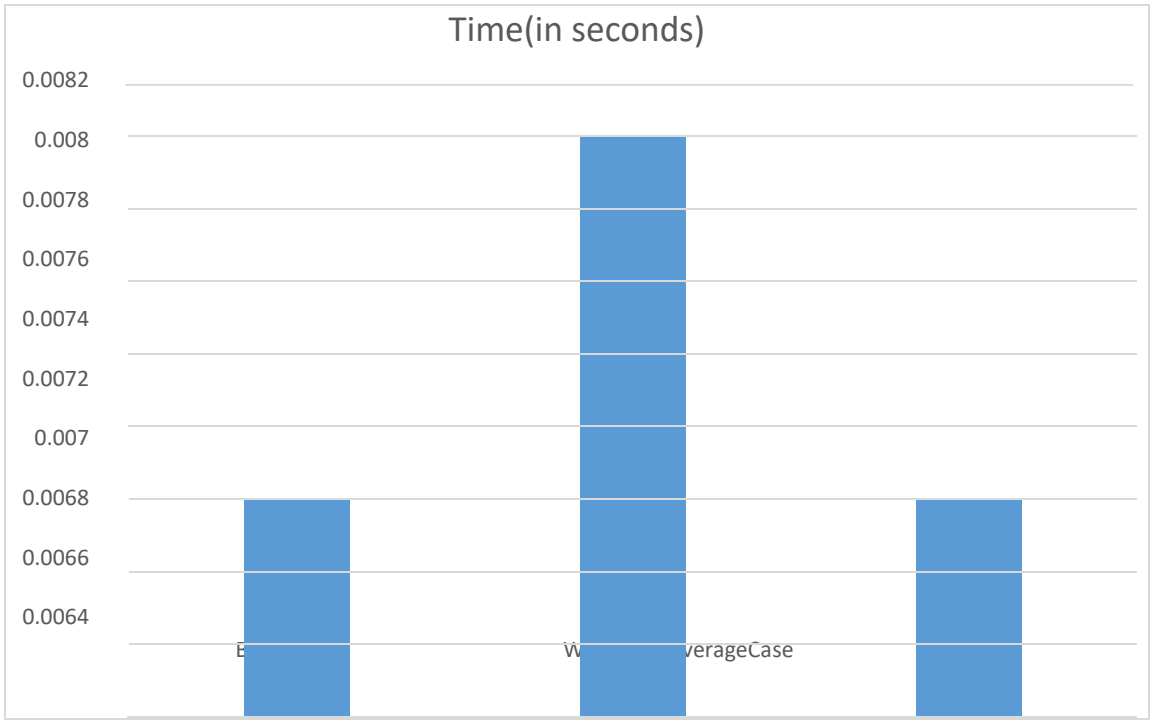
```
    }

    // Copy the output array to arr[], so that arr[] now
// contains sorted numbers according to current digit
for (i = 0; i < n; i++)         arr[i] = output[i];
}
// The main function to that sorts arr[] of size n using
// Radix Sort  void
RadixSort(int arr[], int n)
{
    // Find the maximum number to know number of digits
int m = getMax(arr, n);

    // Do counting sort for every digit. Note that instead
    // of passing digit number, exp is passed. exp is 10^i
    // where i is current digit number
for (int exp = 1; m/exp > 0; exp *= 10)
countSort(arr, n, exp);
}
int main(int argc, char const *argv[])
{

          int n=10000;          int
*BestCase=new int [n];    int
*WorstCase=new int [n];           int
*AverageCase=new int [n];
```

```
        MakeCases(BestCase,WorstCase,AverageCase,n);
clock_t time_req;           time_req = clock();
RadixSort(BestCase,n);    cout<<"Time Taken For Best case in
insertion sort is: "<<(float)time_req/CLOCKS_PER_SEC << "
seconds" << endl;  time_req = clock() - time_req;
RadixSort(WorstCase,n);          cout<<"Time Taken For Worst
case in insertion sort is: "<<(float)time_req/CLOCKS_PER_SEC
<< " seconds" << endl;     time_req = clock() - time_req;
RadixSort(AverageCase,n);       cout<<"Time Taken For
Average case in insertion sort is:
"<<(float)time_req/CLOCKS_PER_SEC << " seconds" << endl;;
        cout<<endl;
return 0;
}
```

OUTPUT:


C:\Windows\System32\cmd.exe
Time Taken For Best case in Radix sort is: 0.015 seconds
Time Taken For Worst case in Radix sort is: 0.008 seconds
Time Taken For Average case in Radix sort is: 0.015 seconds
Press any key to continue . . .

Graph:


Time(in seconds)
0.0082
0.008
0.0078
0.0076
0.0074
0.0072
0.007
0.0068
0.0066
0.0064
verageCase

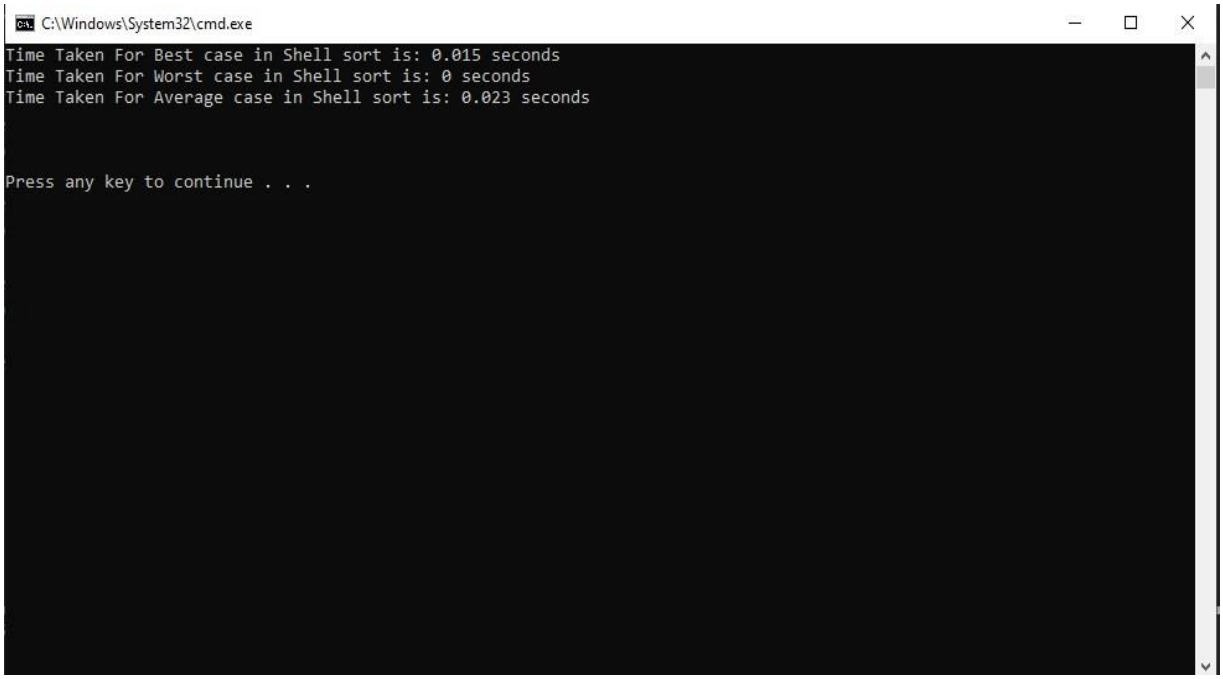
E) SHELL SORT

```
#include <iostream>
#include <algorithm>
#include <bits/stdc++.h>
using namespace std; int
mycompare(int a,int b){
        return a>b;
}
void MakeCases(int *BestCase,int *WorstCase,int *AverageCase,int n){
for (int i = 0; i < n; ++i)
        {
                //average case consist of random variables
                AverageCase[i]=rand();
        }
        for (int i = 0; i < n; ++i)
        {
                //average case consist of random variables
                BestCase[i]=AverageCase[i];
                WorstCase[i]=AverageCase[i];
        }
        //sorted in ascending order
sort(BestCase,BestCase+n);      //sorted in
descending order
sort(WorstCase,WorstCase+n,mycompare); int
ShellSort(int arr[], int n)
```

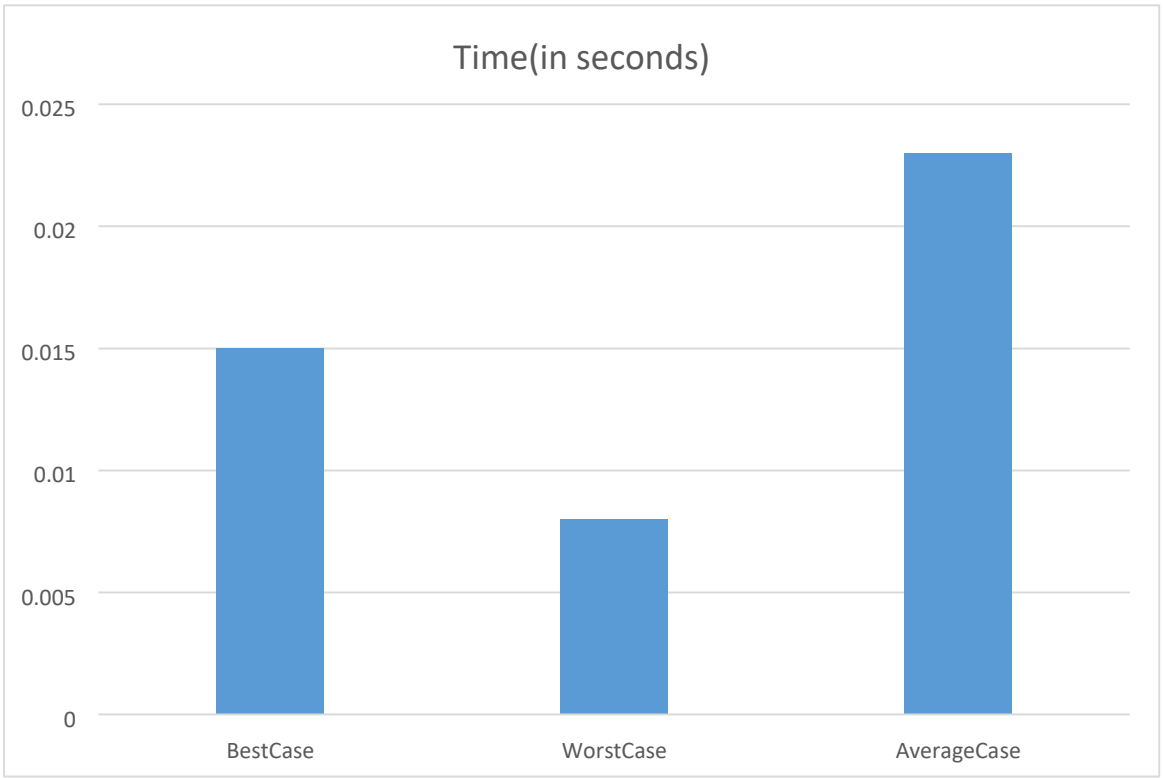
```
{
   // Start with a big gap, then reduce the gap
for (int gap = n/2; gap > 0; gap /= 2)
   {
      // Do a gapped insertion sort for this gap size.
      // The first gap elements a[0..gap-1] are already in gapped order
      // keep adding one more element until the entire array is
      // gap sorted
      for (int i = gap; i < n; i += 1)
      {
         // add a[i] to the elements that have been gap sorted
// save a[i] in temp and make a hole at position i            int
temp = arr[i];

         // shift earlier gap-sorted elements up until the correct
         // location for a[i] is found
         int j;
         for (j = i; j >= gap && arr[j - gap] > temp; j -= gap)
arr[j] = arr[j - gap];

         //  put temp (the original a[i]) in its correct location
arr[j] = temp;
      }
   }
   return 0;
}
```

```
int main(int argc, char const *argv[])
{

        int n=10000;        int
*BestCase=new int [n];   int
*WorstCase=new int [n];         int
*AverageCase=new int [n];

        MakeCases(BestCase,WorstCase,AverageCase,n);
clock_t time_req;           time_req = clock();
ShellSort(BestCase,n);     cout<<"Time Taken For Best case in
insertion sort is: "<<(float)time_req/CLOCKS_PER_SEC << "
seconds" << endl;  time_req = clock() - time_req;
ShellSort(WorstCase,n);   cout<<"Time Taken For Worst case
in insertion sort is: "<<(float)time_req/CLOCKS_PER_SEC << "
seconds" << endl;  time_req = clock() - time_req;
ShellSort(AverageCase,n);        cout<<"Time Taken For
Average case in insertion sort is:
"<<(float)time_req/CLOCKS_PER_SEC << " seconds" << endl;;
cout<<endl;          return 0;
```

OUTPUT:



Graph:



F) HEAP SORT

```
#include <iostream>
```

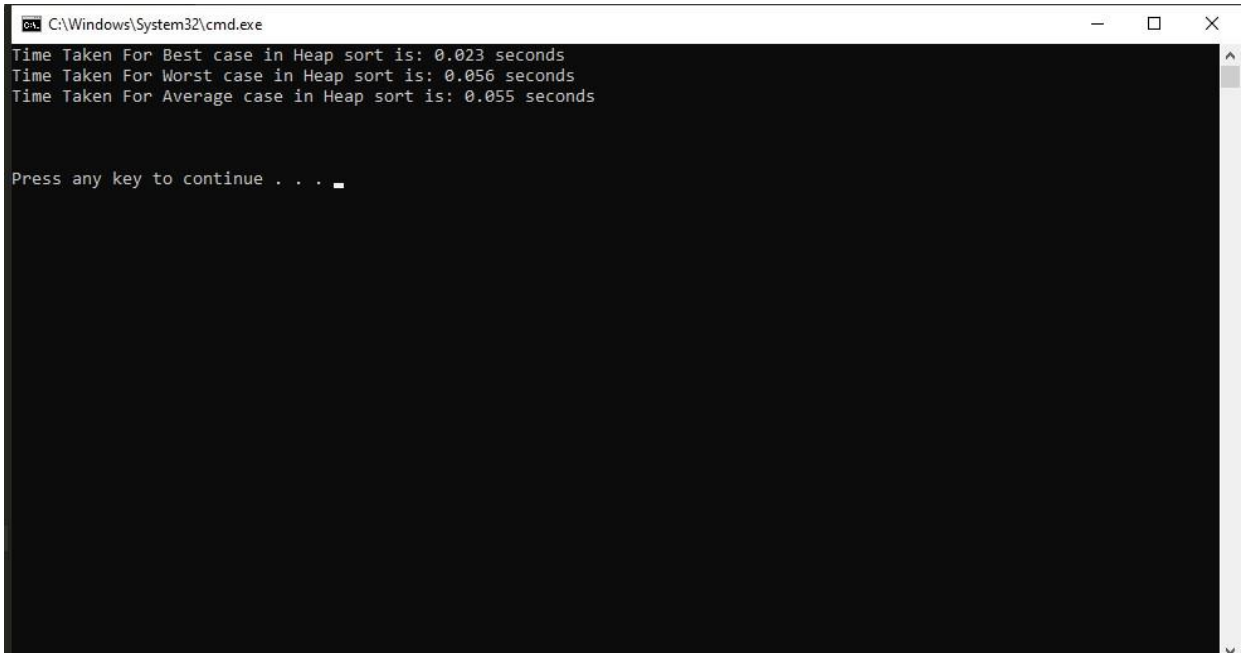
```
#include <algorithm>
#include <bits/stdc++.h>
using namespace std; int
mycompare(int a,int b){
        return a>b;
}
void MakeCases(int *BestCase,int *WorstCase,int *AverageCase,int n){
for (int i = 0; i < n; ++i)
        {
                //average case consist of random variables
                AverageCase[i]=rand();
        }
        for (int i = 0; i < n; ++i)
        {
                //average case consist of random variables
                BestCase[i]=AverageCase[i];
                WorstCase[i]=AverageCase[i];
        }
        //sorted in ascending order
sort(BestCase,BestCase+n);      //sorted in
descending order
sort(WorstCase,WorstCase+n,mycompare); void
heapify(int arr[], int n, int i)
{
```

```
   int largest = i; // Initialize largest as root
int l = 2*i + 1; // left = 2*i + 1      int r = 2*i
+ 2; // right = 2*i + 2

   // If left child is larger than root
if (l < n && arr[l] > arr[largest])
largest = l;

   // If right child is larger than largest so far
if (r < n && arr[r] > arr[largest])         largest
= r;

   // If largest is not root
if (largest != i)
   {
      swap(arr[i], arr[largest]);

      // Recursively heapify the affected sub-tree
heapify(arr, n, largest);
   }
}

// main function to do heap sort  void
HeapSort(int arr[], int n)
```

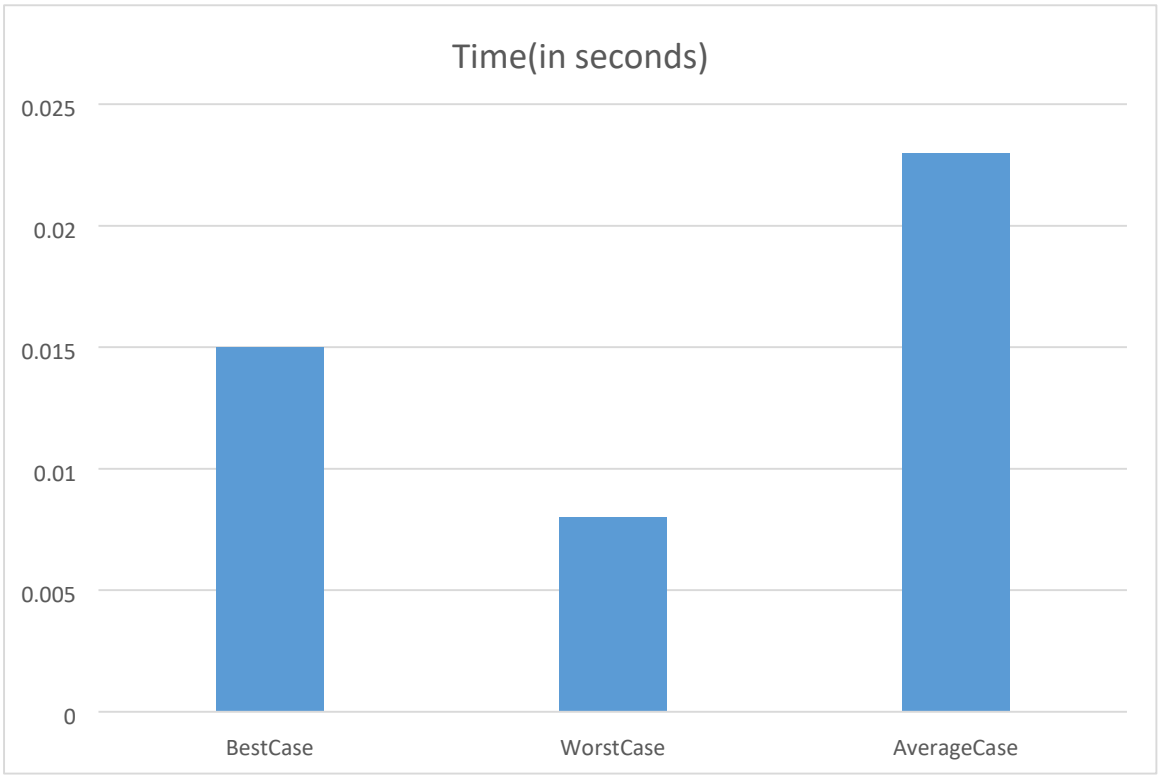
```
{
   // Build heap (rearrange array)
for (int i = n / 2 - 1; i >= 0; i--)
heapify(arr, n, i);

   // One by one extract an element from heap
   for (int i=n-1; i>0; i--)
   {
      // Move current root to end
swap(arr[0], arr[i]);

      // call max heapify on the reduced heap
heapify(arr, i, 0);
   }
}

int main(int argc, char const *argv[])
{

         int n=10000;          int
*BestCase=new int [n];    int
*WorstCase=new int [n];          int
*AverageCase=new int [n];
```

```
        MakeCases(BestCase,WorstCase,AverageCase,n); clock_t
        time_req;

        time_req = clock();          HeapSort(BestCase,n);
cout<<"Time Taken For Best case in insertion sort is:
"<<(float)time_req/CLOCKS_PER_SEC << " seconds" << endl;
time_req = clock() - time_req;    HeapSort(WorstCase,n);
cout<<"Time Taken For Worst case in insertion sort is:
"<<(float)time_req/CLOCKS_PER_SEC << " seconds" << endl;
time_req = clock() - time_req;    HeapSort(AverageCase,n);
cout<<"Time Taken For Average case in insertion sort is:
"<<(float)time_req/CLOCKS_PER_SEC << " seconds" << endl;;
        cout<<endl;
return 0;
}
```

OUTPUT:



Graph:



G) BUCKET SORT

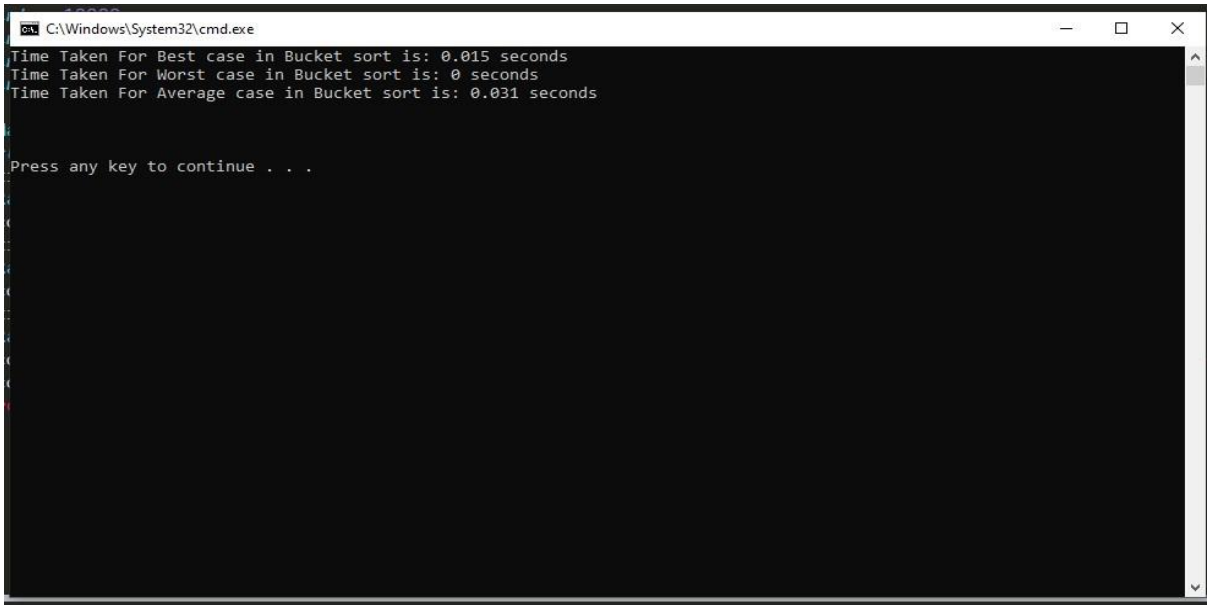
```
#include <iostream>
```

```
#include <algorithm>
#include <bits/stdc++.h>
#include <vector> using
namespace std; int
mycompare(int a,int b){
        return a>b;
}
void MakeCases(int *BestCase,int *WorstCase,int *AverageCase,int n){
for (int i = 0; i < n; ++i)
        {
                //average case consist of random variables
                AverageCase[i]=rand();
        }
        for (int i = 0; i < n; ++i)
        {
                //average case consist of random variables
                BestCase[i]=AverageCase[i];
                WorstCase[i]=AverageCase[i];
        }
        //sorted in ascending order
sort(BestCase,BestCase+n);      //sorted in
descending order
sort(WorstCase,WorstCase+n,mycompare);
}
```

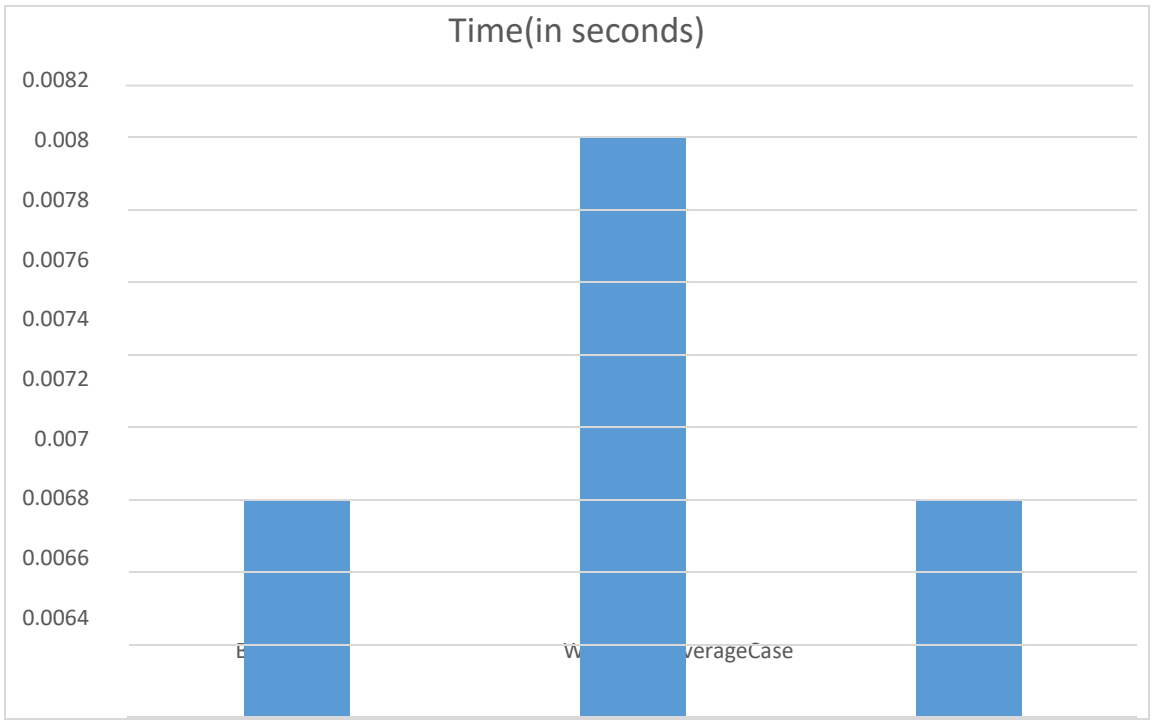
```
void BucketSort(int *array, int size) {
vector<float> bucket[size];
  for(int i = 0; i<size; i++) { //put elements into different buckets
bucket[int(size*array[i])].push_back(array[i]);
  }

  for(int i = 0; i<size; i++) {
    sort(bucket[i].begin(), bucket[i].end()); //sort individual vectors
  }

  int index = 0;    for(int i = 0; i<size; i++) {
while(!bucket[i].empty()) {
array[index++] = *(bucket[i].begin());
bucket[i].erase(bucket[i].begin());
    }
  }
}
int main(int argc, char const *argv[])
{

        int n=10000;          int
*BestCase=new int [n];    int
*WorstCase=new int [n];          int
*AverageCase=new int [n];
MakeCases(BestCase,WorstCase,Av
```

```
erageCase,n);        clock_t
time_req;    time_req = clock();
BucketSort(BestCase,n);
cout<<"Time Taken For Best case in
insertion sort is:
"<<(float)time_req/CLOCKS_PER_SE
C << " seconds" << endl;  time_req =
clock() - time_req;
BucketSort(WorstCase,n);
cout<<"Time Taken For Worst case
in insertion sort is:
"<<(float)time_req/CLOCKS_PER_SE
C << " seconds" << endl;  time_req =
clock() - time_req;
BucketSort(AverageCase,n);
cout<<"Time Taken For Average
case in insertion sort is:
"<<(float)time_req/CLOCKS_PER_SEC << " seconds" << endl;;
        cout<<endl;
return 0;
}
```

OUTPUT:



Graph:



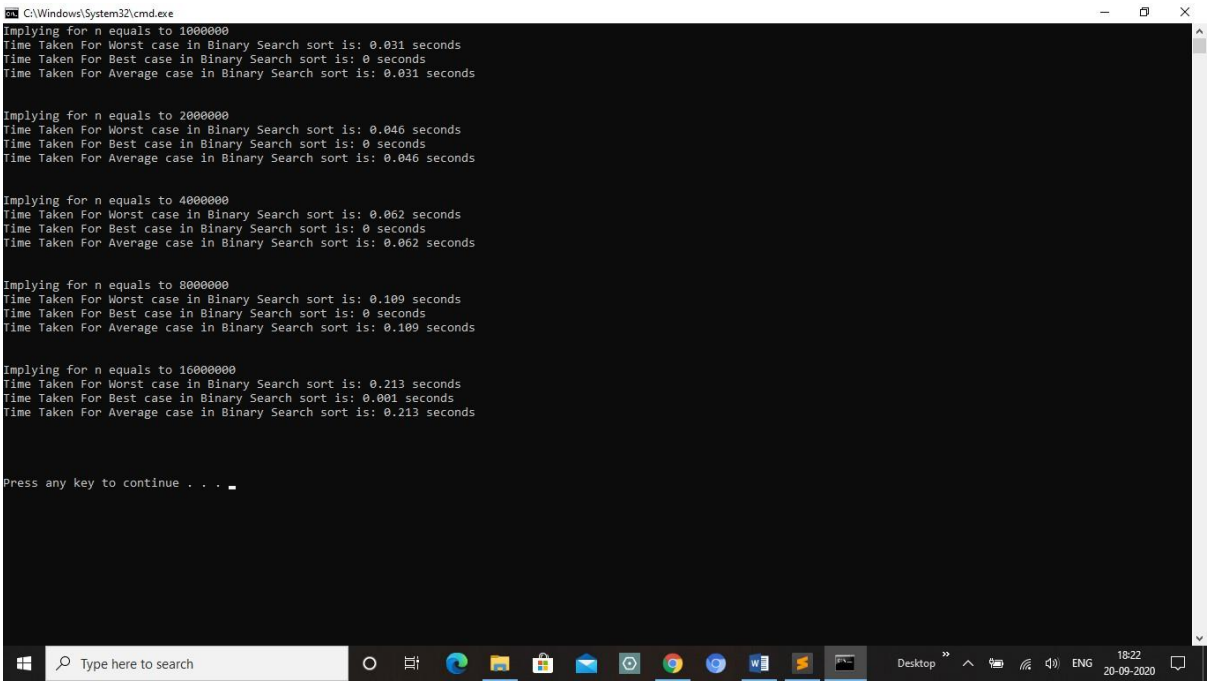
Aim: To implement Binary Search algorithm and analyse it's complexity.

Code:

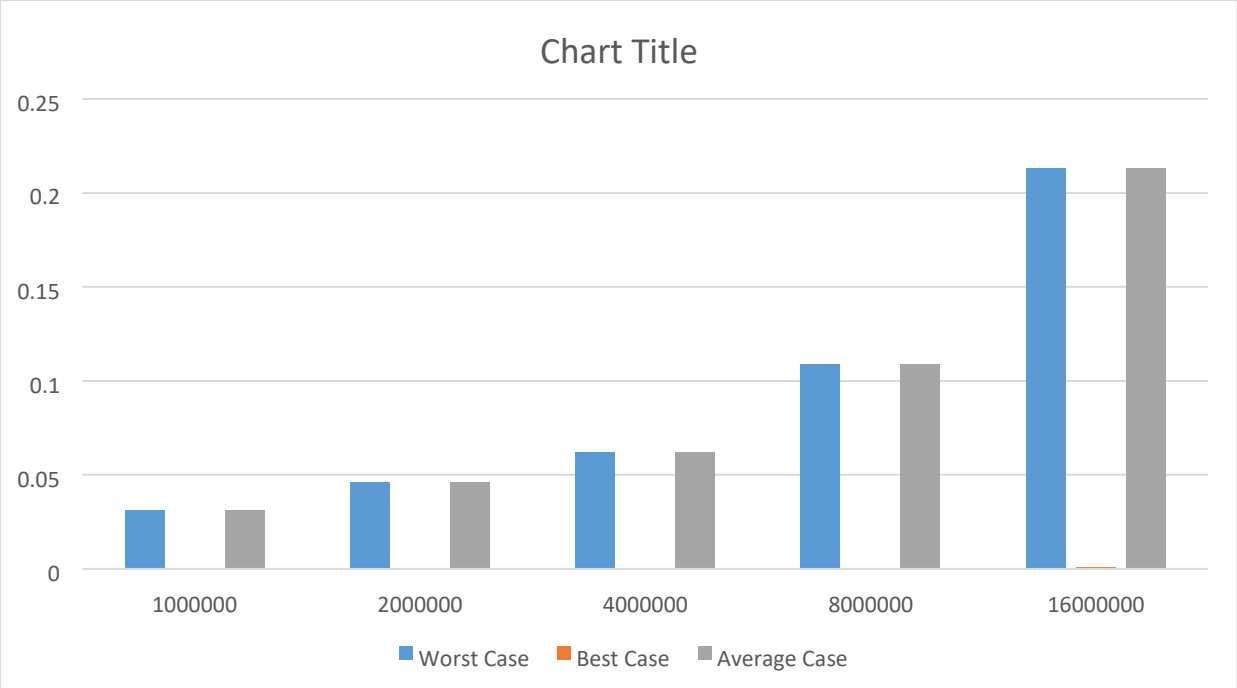
```
#include <iostream>
#include <algorithm> #include
<bits/stdc++.h> using
namespace std;
int BinarySearchPos(int *a, int start, int end, int key){
while(start<=end){          int mid = (start+end)/2;
if(a[mid]==key){
                return mid+1;
        }
        if(a[mid]<key){
                start=mid+1;
        }
        else{
end=mid-1;
        }
 }
 return -1;
}
int main(int argc, char const *argv[])
{  int t=5;  int
n=1000000;
while(t--){
```

```
        cout<<"Implying for n equals to
"<<n<<endl;          int *a = new int [n];         for
(int i = 0; i < n; ++i)
        {
                a[i]=i;
        }
        clock_t time_req;
time_req = clock();

        int key = n+100;
        BinarySearchPos(a,0,n-1,key);   cout<<"Time Taken For
Worst case in Binary Search sort is:
"<<(float)time_req/CLOCKS_PER_SEC << " seconds" << endl;
time_req = clock() - time_req;   key=a[(n-1)/2];
        BinarySearchPos(a,0,n-1,key);   cout<<"Time Taken For
Best case in Binary Search sort is:
"<<(float)time_req/CLOCKS_PER_SEC << " seconds" << endl;
time_req = clock() - time_req;   key=rand()%n;
        BinarySearchPos(a,0,n-1,key);   cout<<"Time Taken For
Average case in Binary Search sort is:
"<<(float)time_req/CLOCKS_PER_SEC << " seconds" << endl;
n+=n;
        cout<<endl<<endl;
 }
 return 0;
}
```

Output:



Graph:



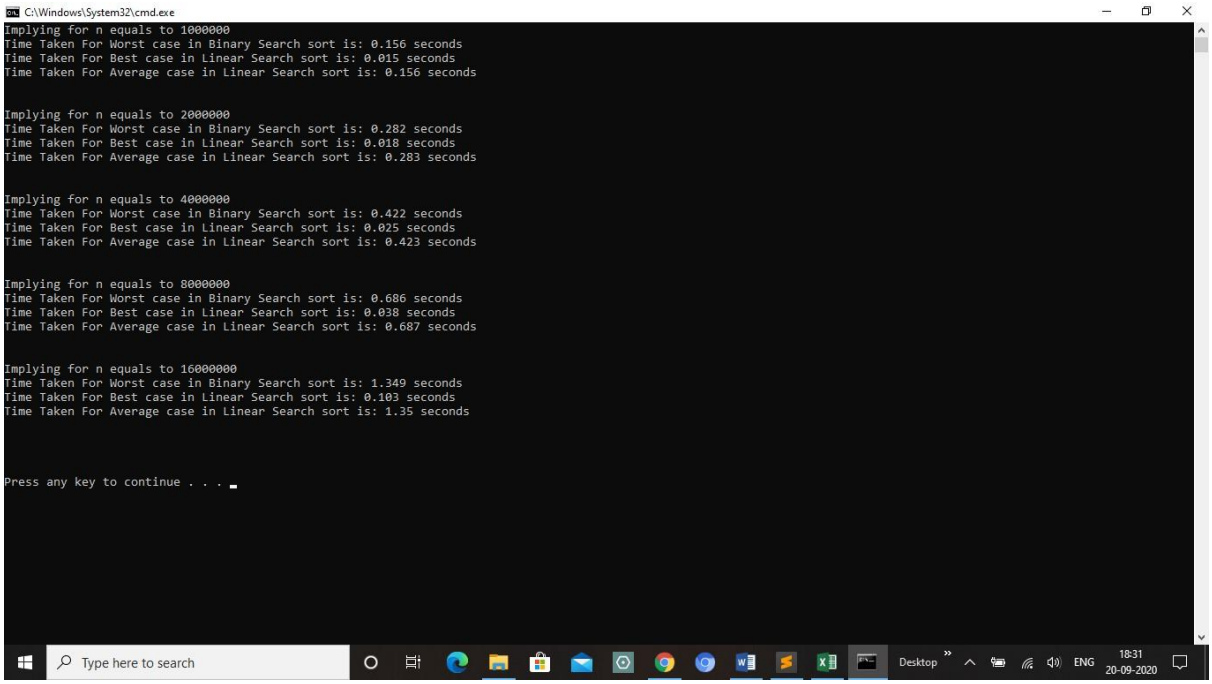
Aim: To implement Linear Search algorithm and analyse it's complexity.

Code:

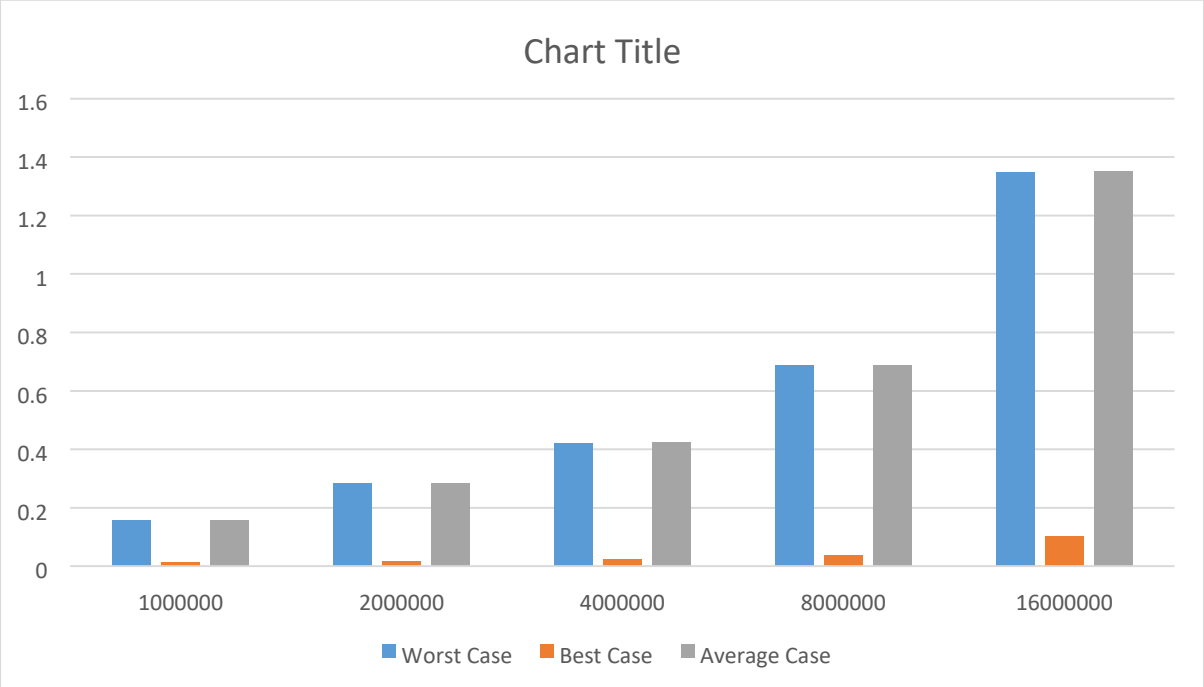
```
#include <iostream>
#include <algorithm> #include
<bits/stdc++.h> using
namespace std;
int LinearSearchPos(int *a,int start,int end,int key){
for (int i = start; i <= end; ++i)
 {
        if(a[i]==key){
                return i+1;
        }
 }
 return -1;
}

int main(int argc, char const *argv[])
{  int t=5;  int
n=1000000;
while(t--){
        cout<<"Implying for n equals to
"<<n<<endl;          int *a = new int [n];         for
(int i = 0; i < n; ++i)
        {
                a[i]=rand()%n;
        }
        clock_t time_req;
time_req = clock();
```

```
        int key = n+100;
        LinearSearchPos(a,0,n-1,key);    cout<<"Time Taken For
Worst case in Binary Search sort is:
"<<(float)time_req/CLOCKS_PER_SEC << " seconds" << endl;
time_req = clock() - time_req;    key=a[0];
        LinearSearchPos(a,0,n-1,key);    cout<<"Time Taken For
Best case in Linear Search sort is:
"<<(float)time_req/CLOCKS_PER_SEC << " seconds" << endl;
time_req = clock() - time_req;    key=rand()%n;
        LinearSearchPos(a,0,n-1,key);    cout<<"Time Taken For
Average case in Linear Search sort is:
"<<(float)time_req/CLOCKS_PER_SEC << " seconds" << endl;
n+=n;
        cout<<endl<<endl;
 }
 return 0;
}
```

Output:



Graph:



Aim: To implement Merge Sort algorithm and analyse it's complexity.

Code:

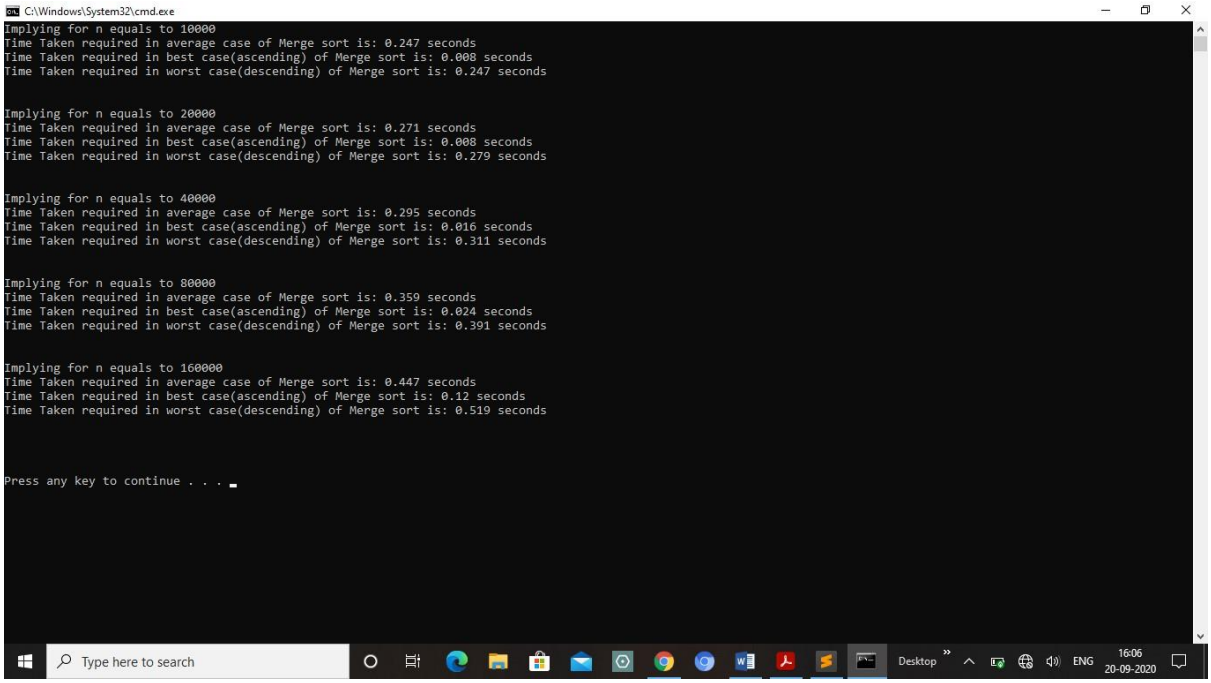
```
#include <iostream> #include
<bits/stdc++.h> using
namespace std;
void merge(int arr[], int l, int m, int r)
{     int i, j, k;      int
n1 = m - l + 1;
int n2 = r - m;

   /* create temp arrays */
int L[n1], R[n2];

   /* Copy data to temp arrays L[] and R[] */
for (i = 0; i < n1; i++)         L[i] = arr[l + i];
for (j = 0; j < n2; j++)
       R[j] = arr[m + 1 + j];

   /* Merge the temp arrays back into arr[l..r]*/
i = 0; // Initial index of first subarray      j = 0; //
Initial index of second subarray      k = l; //
Initial index of merged subarray      while (i < n1
&& j < n2) {
       if (L[i] <= R[j]) {
arr[k] = L[i];
i++;         }
else {            arr[k]
```

```
= R[j];            j++;
}        k++;
   }

   /* Copy the remaining elements of L[], if there
are any */     while (i < n1) {         arr[k] = L[i];
i++;         k++;
   }

   /* Copy the remaining elements of R[], if there
are any */     while (j < n2) {         arr[k] = R[j];
j++;         k++;
   }
}

/* l is for left index and r is right index of the
sub-array of arr to be sorted */ void
mergeSort(int arr[], int l, int r)
{     if (l <
r) {
       // Same as (l+r)/2, but avoids overflow for
       // large l and h
int m = l + (r - l) / 2;
```

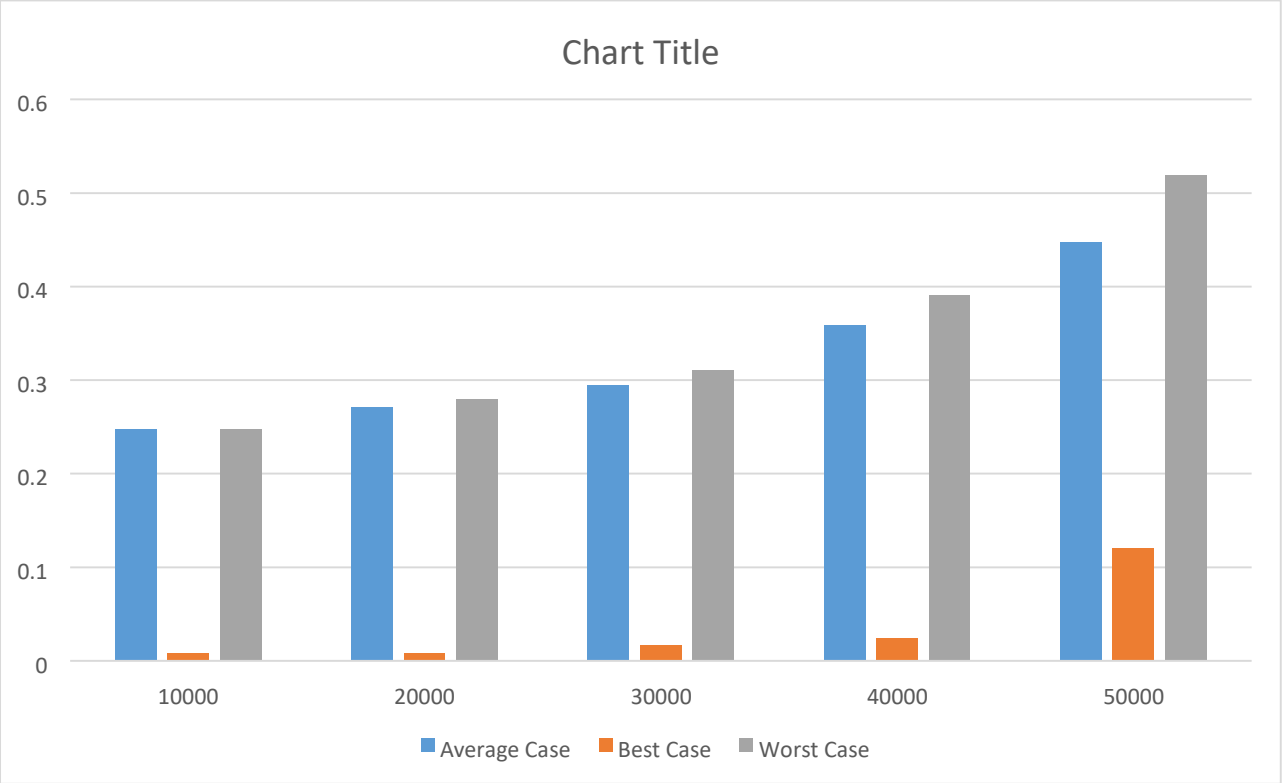
```
        // Sort first and second halves
mergeSort(arr, l, m);          mergeSort(arr,
m + 1, r);


        merge(arr, l, m, r);
    }
}
int main() {
int t=5;     int
n=10000;
while(t--){
        cout<<"Implying for n equals to "<<n<<endl;
int *a = new int [n];         int *b = new int [n];
int *c = new int [n];

        for (int i = 0; i < n; ++i)
        {
           a[i]=rand()%n;
c[i]=(n-i);             if(i<n/2){
              b[i]=rand()%(n/2);
           }
else{
              b[i]=rand()%n+n/2;
           }
        }
        clock_t time_req;         time_req = clock();         mergeSort(a,0,n);
cout<<"Time Taken required in average case of Merge sort is:
```

```
"<<(float)time_req/CLOCKS_PER_SEC << " seconds" << endl;
time_req = clock() - time_req;        mergeSort(b,0,n);        cout<<"Time
Taken required in best case(ascending) of Merge sort is:
"<<(float)time_req/CLOCKS_PER_SEC << " seconds" << endl;        time_req =
clock() - time_req;        mergeSort(c,0,n);        cout<<"Time Taken required
in worst case(descending) of Merge sort is:
"<<(float)time_req/CLOCKS_PER_SEC << " seconds" << endl;
cout<<endl<<endl;        n+=n;
   }
   return 0;
}
```

Output:



Graph:



Aim: To implement Quick Sort algorithm and analyse it's complexity.

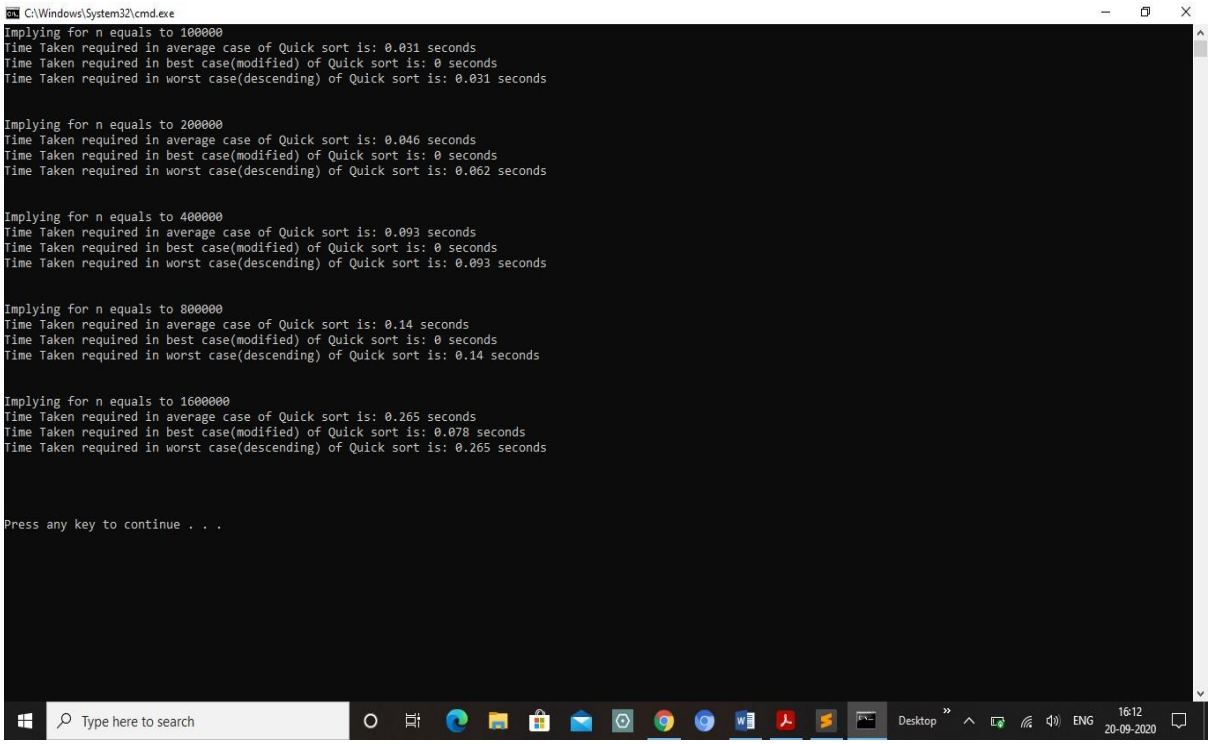
Code:

```
#include <iostream> #include
<bits/stdc++.h> using
namespace std;
int partition(int *a,int n,int s){
int i=s-1;    int j=s;    int
pivot=a[n];    for(j=s;j<n;j++){
if(a[j]<pivot){          i++;
swap(a[i],a[j]);
      }
   }
   swap(a[n],a[i+1]);
return i+1;
}
```

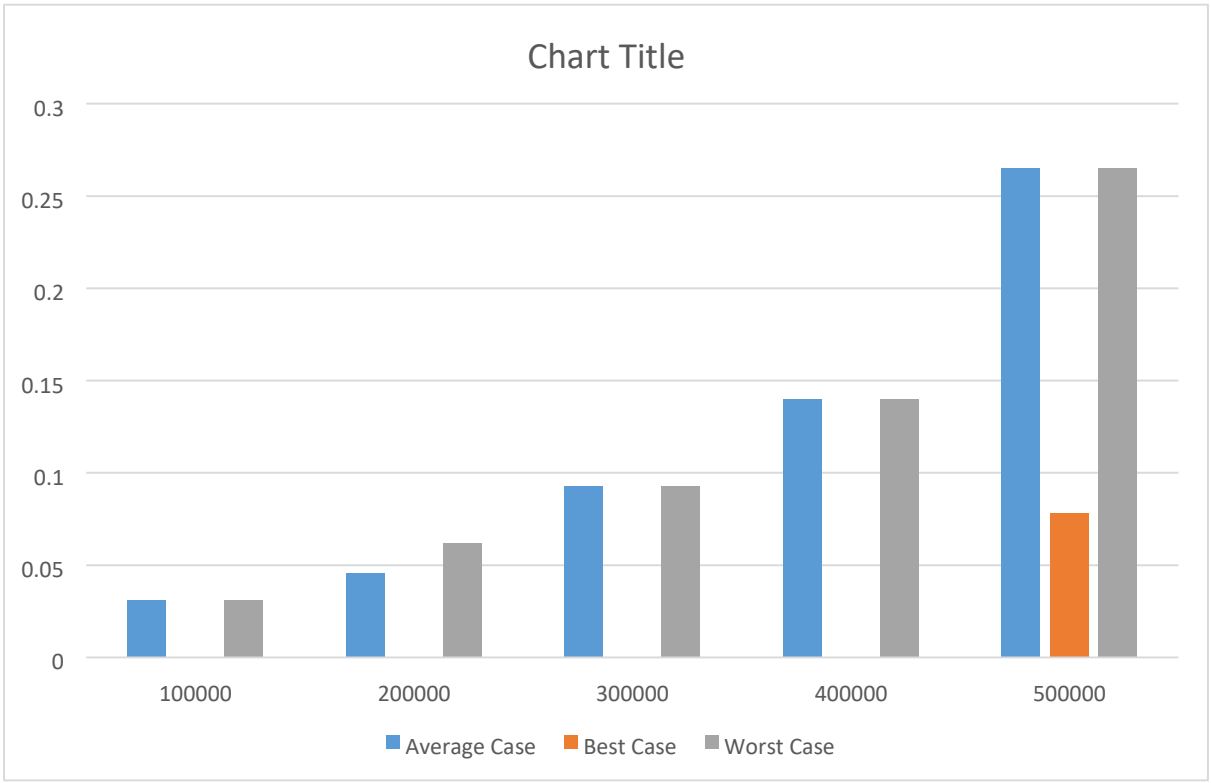
```
void quicksort(int *a,int n,int s){
int p;     if(s>=n)     return;
else
   {
     p=partition(a,n,s);
     quicksort(a,p-1,s);
quicksort(a,n,p+1);
   }
   return;
}
int main() {    int
t=5;     int
n=100000;
while(t--){
       cout<<"Implying for n equals to "<<n<<endl;
int *a = new int [n];        int *b = new int [n];
int *c = new int [n];        for (int i = 0; i < n; ++i)
       {
          a[i]=rand()%n;
c[i]=(n-i);            if(i<n/2){
b[i]=rand()%(n/2);
          }
          else if(i>(n-2) && i>(n/2)){
b[i]=rand()%n+n/2;
```

```
        }
else{
b[i]=n/2;
        }
      }
      clock_t time_req;       time_req = clock();       quicksort(a,0,n);
cout<<"Time Taken required in average case of Quick sort is:
"<<(float)time_req/CLOCKS_PER_SEC << " seconds" << endl;
time_req = clock() - time_req;        quicksort(b,0,n);       cout<<"Time
Taken required in best case(modified) of Quick sort is:
"<<(float)time_req/CLOCKS_PER_SEC << " seconds" << endl;       time_req
= clock() - time_req;        quicksort(c,0,n);       cout<<"Time Taken required
in worst case(descending) of Quick sort is:
"<<(float)time_req/CLOCKS_PER_SEC << " seconds" << endl;
cout<<endl<<endl;       n+=n;
   }
   return 0;
}
```

Output:



Graph:



Aim: To implement Strassen's Matrix Multiplication Algorithm and analyse it's complexity.

Code:

```
#include <stdio.h>
```

```
#include <stdlib.h>
#include <math.h>
#include <bits/stdc++.h>
#define SIZE 32

using namespace std;

void ReadMatrix(double**,int); void
WriteMatrix(double**,int);

void MatrixAdd(double **A, double **B, double **Result, int N); void
MatrixSubtrac(double **A, double **B, double **Result, int N);

void StrassenAlgorithm(double **A, double **B, double **C, int N);


/* For taking input from standard input(keyboard)*/ void
ReadMatrix(double A[][SIZE],int N)
{    int
i,j;

   for(i=0; i<N; i++)
   {
      for(j=0; j<N; j++)
      {
         A[i][j]=rand();
```

```
        }
    }
}

/*For printing the matrix in standard output(console)*/ void
WriteMatrix(double A[][SIZE], int N)
{    int i,
j;

    for(i=0; i<N; i++)
    {
        for(j=0; j<N; j++)
        {
            printf("%0.1lf \t", A[i][j]);
        }
        printf("\n");
    }
}

/*This function will add two square matrix*/
void MatrixAdd(double A[][SIZE], double B[][SIZE], double Result[][SIZE], int N)
{
    int i, j;

    for(i=0; i< N; i++)
    {
```

```
      for(j=0; j<N; j++)
      {
         Result[i][j] = A[i][j] + B[i][j];
      }
   }

}

/*This function will subtract one  square matrix from another*/
void MatrixSubtrac(double A[][SIZE], double B[][SIZE], double Result[][SIZE], int N)
{    int i,
j;

   for(i=0; i< N; i++)
   {
      for(j=0; j<N; j++)
      {
         Result[i][j] = A[i][j] - B[i][j];
      }
   }
}


/*This is the strassen algorithm. (Divide and Conqure)*/
void StrassenAlgorithm(double A[][SIZE], double B[][SIZE], double C[][SIZE], int N)
{
   // trivial case: when the matrice is 1 X 1:
```

```
   if(N == 1)
   {
      C[0][0] = A[0][0] * B[0][0];
return;
   }

   // other cases are treated here:
else
   {
      int Divide  = (int)(N/2);

      double A11[SIZE][SIZE], A12[SIZE][SIZE], A21[SIZE][SIZE], A22[SIZE][SIZE];
double B11[SIZE][SIZE], B12[SIZE][SIZE], B21[SIZE][SIZE], B22[SIZE][SIZE];
double C11[SIZE][SIZE], C12[SIZE][SIZE], C21[SIZE][SIZE], C22[SIZE][SIZE];
double P1[SIZE][SIZE], P2[SIZE][SIZE], P3[SIZE][SIZE], P4[SIZE][SIZE],
P5[SIZE][SIZE], P6[SIZE][SIZE], P7[SIZE][SIZE];       double AResult[SIZE][SIZE],
BResult[SIZE][SIZE];

      int i, j;

      //dividing the matrices in 4 sub-matrices:
      for (i = 0; i < Divide; i++)
      {
         for (j = 0; j < Divide; j++)
         {
            A11[i][j] = A[i][j];
```

```
            A12[i][j] = A[i][j + Divide];
A21[i][j] = A[i + Divide][j];
            A22[i][j] = A[i + Divide][j + Divide];

            B11[i][j] = B[i][j];
            B12[i][j] = B[i][j + Divide];
B21[i][j] = B[i + Divide][j];
            B22[i][j] = B[i + Divide][j + Divide];
          }
       }

       // Calculating p1 to p7:
       /*For details -- Introduction to Algorithms 3rd Edition by CLRS*/

       MatrixAdd(A11, A22, AResult, Divide);   // a11 + a22
       MatrixAdd(B11, B22, BResult, Divide);   // b11 + b22
       StrassenAlgorithm(AResult, BResult, P1, Divide);  // p1 = (a11+a22) *
(b11+b22)

       MatrixAdd(A21, A22, AResult, Divide);   // a21 + a22
       StrassenAlgorithm(AResult, B11, P2, Divide); // p2 = (a21+a22) * (b11)
MatrixSubtrac(B12, B22, BResult, Divide); // b12 - b22
       StrassenAlgorithm(A11, BResult, P3, Divide); // p3 = (a11) * (b12 - b22)

       MatrixSubtrac(B21, B11, BResult, Divide); // b21 - b11
       StrassenAlgorithm(A22, BResult, P4, Divide); // p4 = (a22) * (b21 - b11)
```

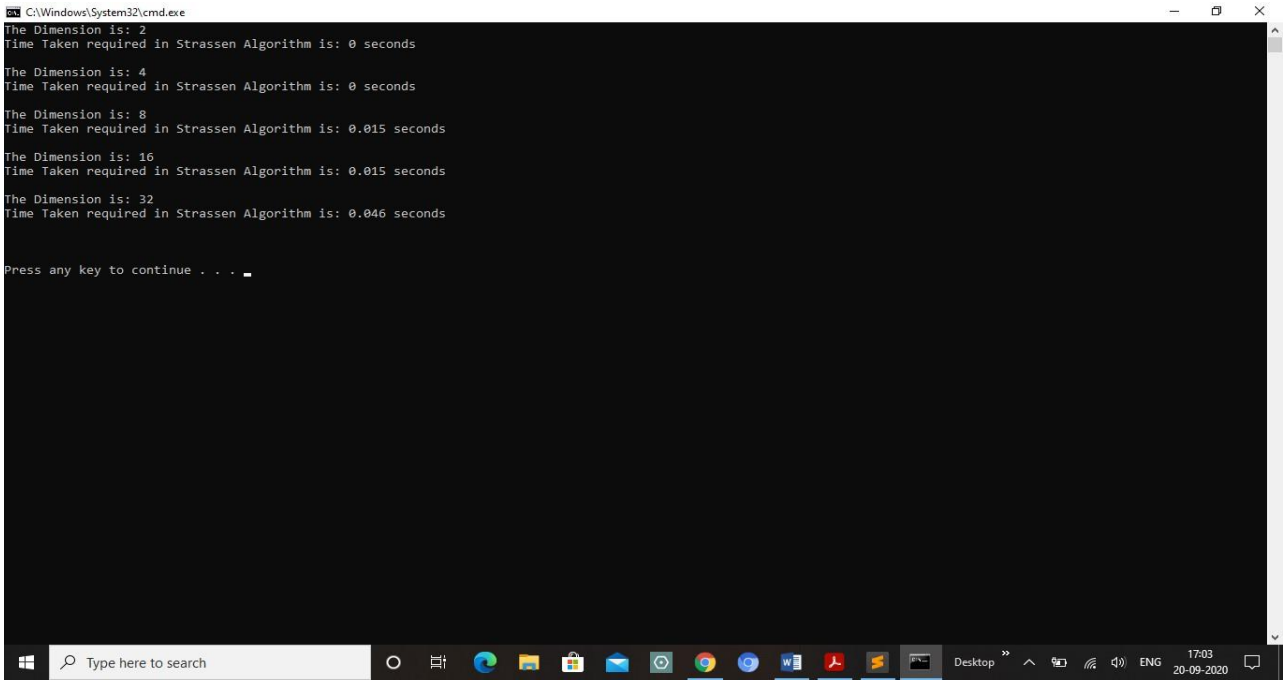
```
        MatrixAdd(A11, A12, AResult, Divide); // a11 + a12
        StrassenAlgorithm(AResult, B22, P5, Divide); // p5 = (a11+a12) * (b22)

        MatrixSubtrac(A21, A11, AResult, Divide); // a21 - a11
        MatrixAdd(B11, B12, BResult, Divide); // b11 + b12
        StrassenAlgorithm(AResult, BResult, P6, Divide); // p6 = (a21-a11) *
(b11+b12)

        MatrixSubtrac(A12, A22, AResult, Divide); // a12 - a22
        MatrixAdd(B21, B22, BResult, Divide); // b21 + b22
        StrassenAlgorithm(AResult, BResult, P7, Divide); // p7 = (a12-a22) *
(b21+b22)

        // calculating c21, c21, c11 e c22:

        MatrixAdd(P3, P5, C12, Divide); // c12 = p3 + p5
MatrixAdd(P2, P4, C21, Divide); // c21 = p2 + p4

        MatrixAdd(P1, P4, AResult, Divide); // p1 + p4
        MatrixAdd(AResult, P7, BResult, Divide); // p1 + p4 + p7
        MatrixSubtrac(BResult, P5, C11, Divide); // c11 = p1 + p4 - p5 + p7
MatrixAdd(P1, P3, AResult, Divide); // p1 + p3
        MatrixAdd(AResult, P6, BResult, Divide); // p1 + p3 + p6
        MatrixSubtrac(BResult, P2, C22, Divide); // c22 = p1 + p3 - p2 + p6


        // Grouping the results obtained in a single matrice:
```

```
        for (i = 0; i < Divide ; i++)
        {
           for (j = 0 ; j < Divide ; j++)
           {
              C[i][j] = C11[i][j];
              C[i][j + Divide] = C12[i][j];
              C[i + Divide][j] = C21[i][j];
              C[i + Divide][j + Divide] = C22[i][j];
           }
        }

     }

}

/*The main function*/ int
main()
{
   int t=5;     int
fixValue=2;
while(t--){
        double A[SIZE][SIZE], B[SIZE][SIZE], C[SIZE][SIZE];
int i,j;
        int N=fixValue,M,Count = 0;
```

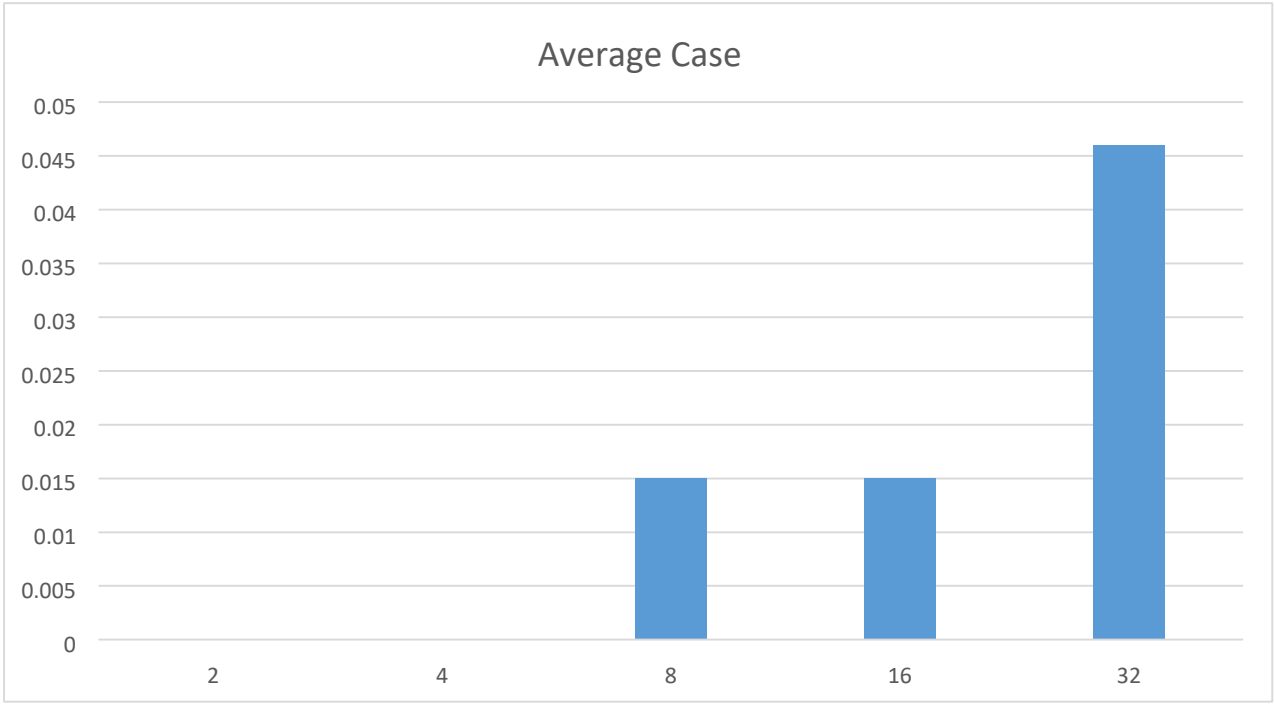
```
        printf("The Dimension is: %d\n",fixValue);
        M = N;

        ReadMatrix(A,M);
        ReadMatrix(B,M);

        if(M > 1)
        {

           while(M>=2)
           {
              M/=2;
              Count++;
           }

M = N;

           if(M != (pow(2.0,Count)))
           {
N  = pow(2.0,Count+1);

              for(i=0; i<N; i++)
              {
                 for(j=0; j<N; j++)
                 {
                    if((i>=M) || (j>=M))
```

```
                {
                    A[i][j] = 0.0;
                    B[i][j] = 0.0;
                }
              }
            }
          }
      }
      clock_t time_req;
time_req = clock();

      StrassenAlgorithm(A,B,C,N); // StrassenAlgorithm called here

      cout<<"Time Taken required in Strassen Algorithm is:
"<<(float)time_req/CLOCKS_PER_SEC << " seconds" << endl<<endl;
time_req = clock() - time_req;         fixValue*=2;
   }
   return 0;
}
```

Output:



Graph:



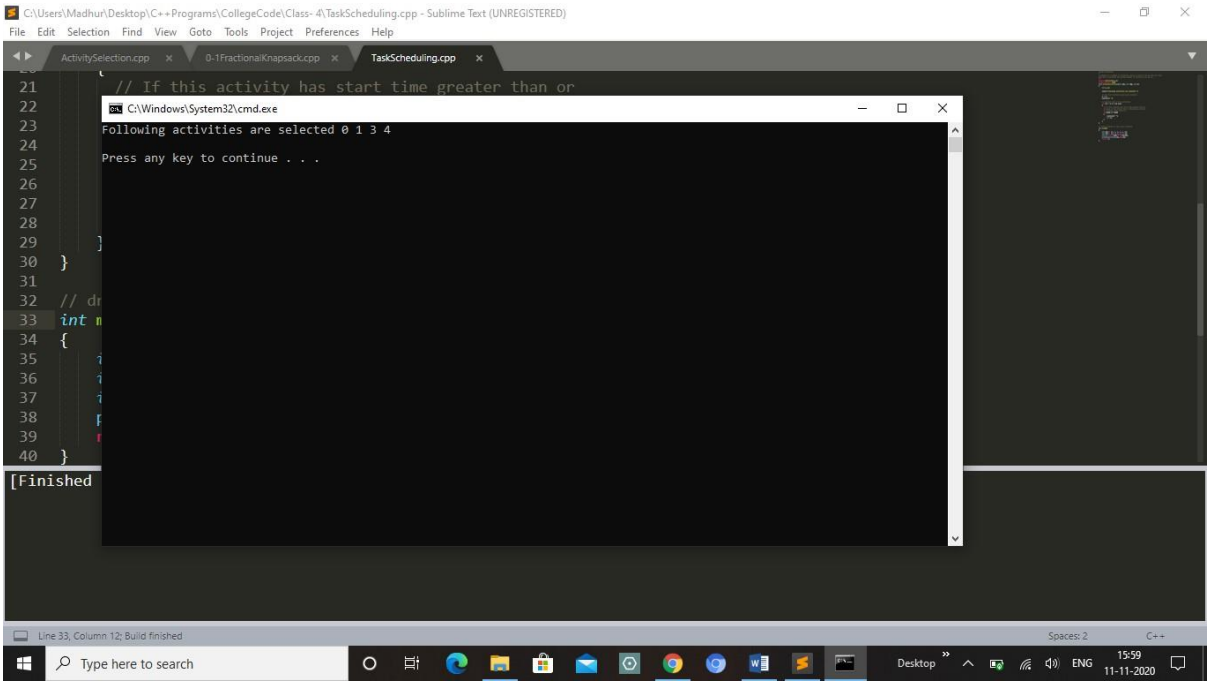
Aim: Task Scheduling Program (GREEDY APPROACH) Code:

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

```
void printMaxActivities(int s[], int f[], int n)
{     int i,
j;

   cout<<"Following activities are selected ";

   // The first activity always gets selected
i = 0;
   cout<<i<<" ";

   // Consider rest of the activities
for (j = 1; j < n; j++)
   {
     // If this activity has start time greater than or
     // equal to the finish time of previously selected
     // activity, then select it
if (s[j] >= f[i])
     {
        cout<<j<<" ";
        i = j;
     }
   }
}

// driver program to test above function  int
main()
{
```

```
    int s[] =  {1, 3, 0, 5, 8, 5};
int f[] =  {2, 4, 6, 7, 9, 9};      int
n = sizeof(s)/sizeof(s[0]);
printMaxActivities(s, f, n);
return 0;
}
```

Output:



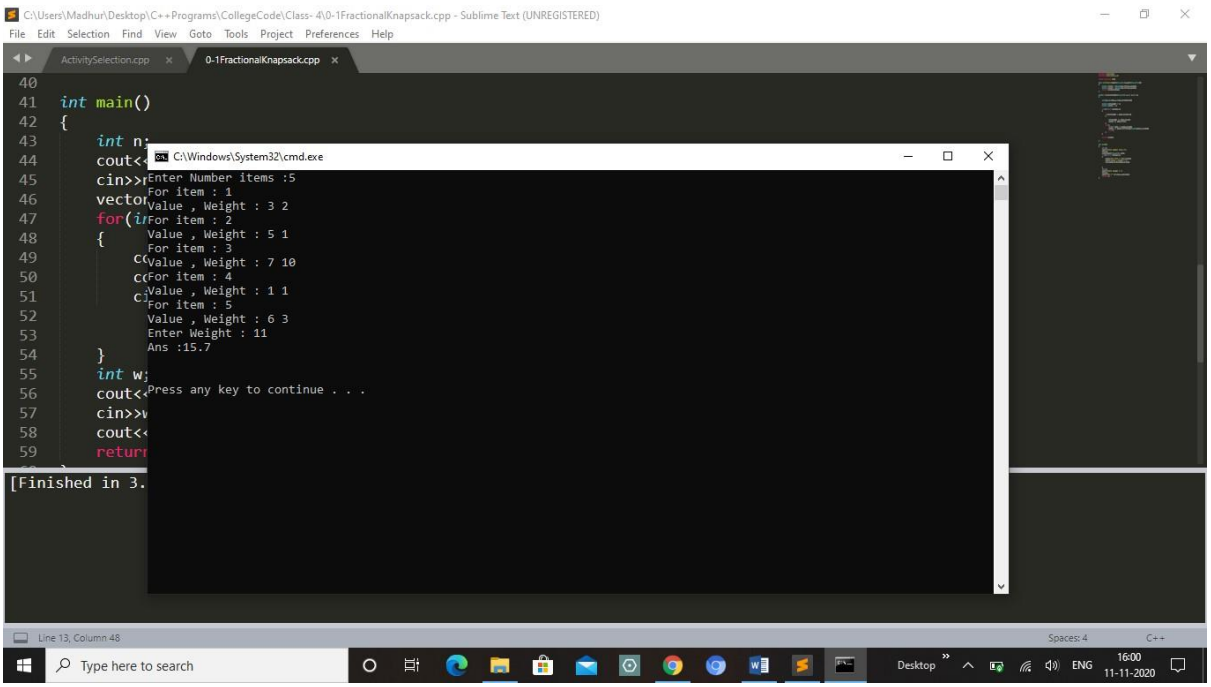
Aim: To implement 0-1 fractional knapsack (GREEDY APPROACH) Code:

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

using namespace std;

bool sortByRatio(pair<int,int> &a,pair<int,int> &b)
{
```

```
   double ratio1 = (double)a.first/a.second;
double ratio2 = (double)b.first/b.second;     return
ratio1>ratio2;
}

double fun(vector<pair<int,int>> a,int n,int w)
{

   sort(a.begin(),a.end(),sortByRatio);

   double Currweight = 0;
double valAns = 0;

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

      if(Currweight + a[i].second<=w)
      {

         Currweight += a[i].second;
valAns += a[i].first;
      }
      else{
         double temp = w-a[i].second;           valAns +=
a[i].first*(temp/double(a[i].second));           break;
      }
   }

   return valAns;

}

int main()
```

```
{    int
n;
   cout<<"Enter Number items :";
cin>>n;    vector<pair<int,int>>
a(n);    for(int i = 0;i<n;i++)
   {
      cout<<"For item : "<<i+1<<endl;
cout<<"Value , Weight : ";
cin>>a[i].first>>a[i].second;



   }
   int w;    cout<<"Enter Weight : ";
cin>>w;    cout<<"Ans :"
<<fun(a,n,w)<<endl;    return 0;
}
```

Output:

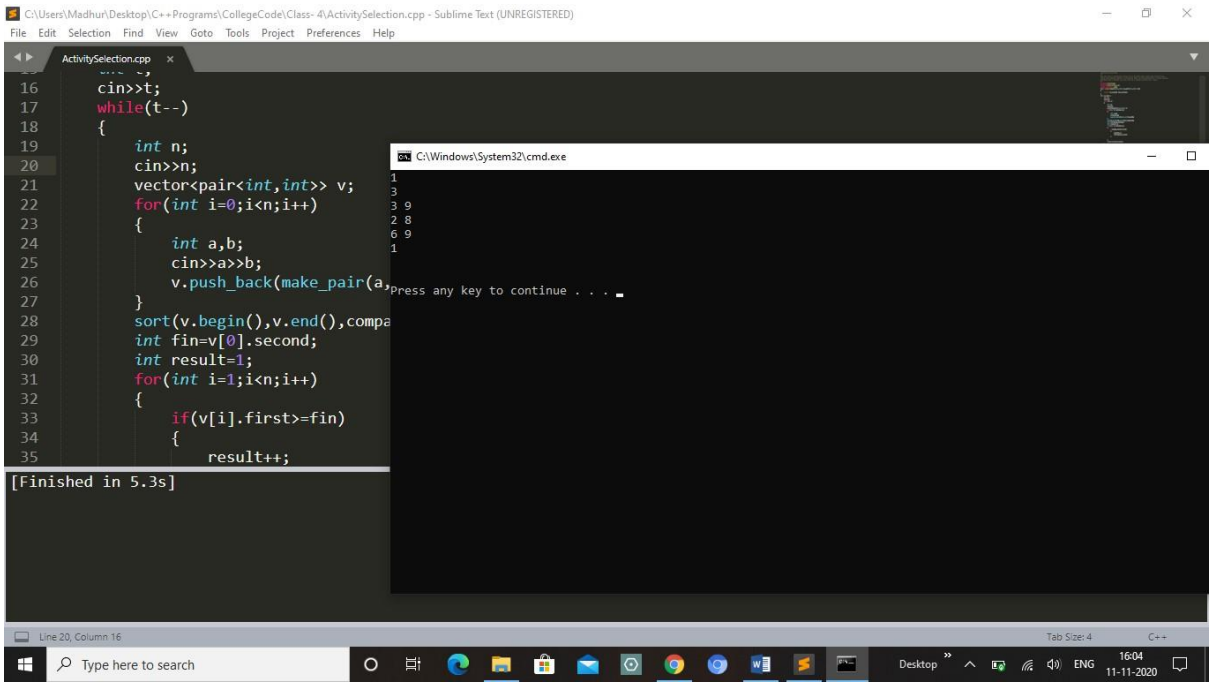


Aim: To implement Activity Selection Problem (GREEDY APPROACH) CODE:

```
#include<iostream>
#include<bits/stdc++.h> using
namespace std;
bool compare(pair<int,int> v1,pair<int,int> v2)
{
        return v1.second <v2.second;
}
int main() {   int
t;      cin>>t;
while(t--)
        {
                int n;
cin>>n;
                vector<pair<int,int>> v;
for(int i=0;i<n;i++)
                {
                        int a,b;
        cin>>a>>b;
                        v.push_back(make_pair(a,b));
                }
```

```
sort(v.begin(),v.end(),compare);
int fin=v[0].second;                    int result=1;
                for(int i=1;i<n;i++)
                {
                        if(v[i].first>=fin)
                        {
                                result++;
                                fin=v[i].second;
                        }

                }
                cout<<result<<endl;
        }
        return 0;
}
```

OUTPUT:



Aim: Longest Common Subsequence using Dynamic Programming

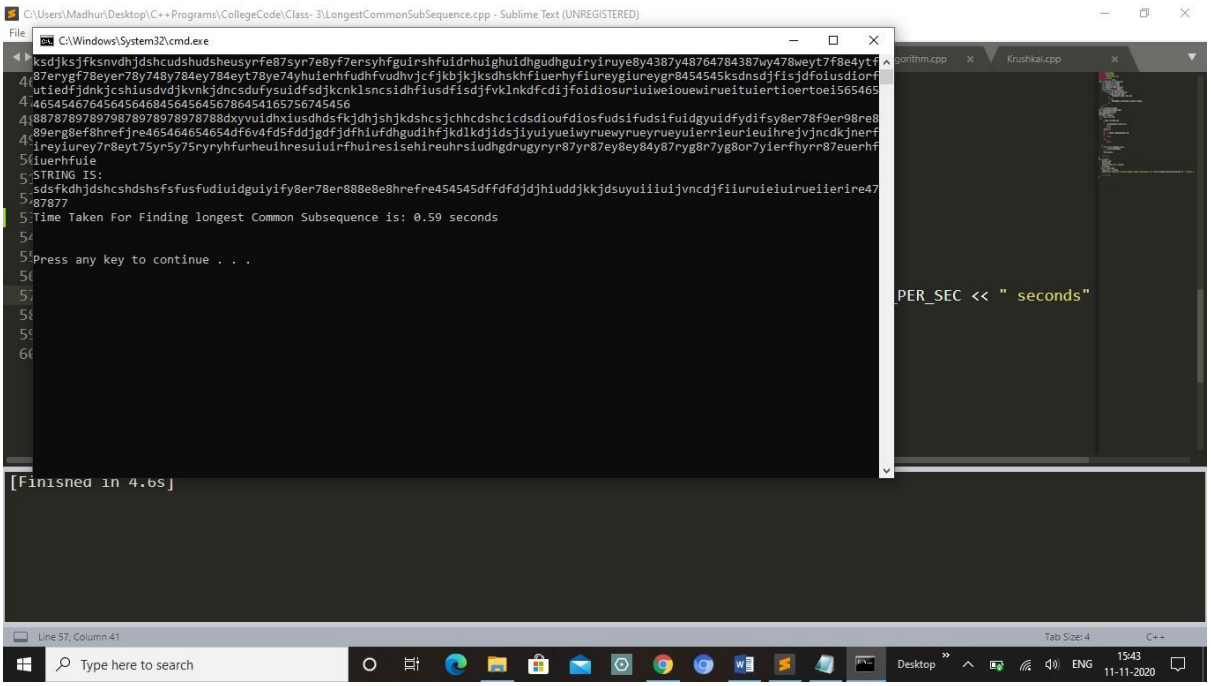
Code:

```
#include <iostream>
#include<algorithm>
#include<string> using namespace
std; void LCS(string a, string b){
int sizeA=a.length();        int
sizeB=b.length();    int
DP[sizeA+1][sizeB+1];
memset(DP,0,sizeof(DP));
for(int i=1;i<=sizeA;i++){
for(int j=1;j<=sizeB;j++){
if(a[i-1]==b[j-1]){
                        DP[i][j]=1+DP[i-1][j-1];
                    }
```

```
                else{
                        DP[i][j]=max(DP[i][j-1],DP[i-1][j]);
                }
            }
        }
  int i=sizeA,j=sizeB; int
index=DP[sizeA][sizeB];
char common[index+1];
common[index]='\0'; int
temp = index;
  while(i>0 && j>0)
  {
        if(a[i-1]==b[j-1])
        {
                common[index-1]=a[i-1];
        i--;            j--;
     index--;
        }
        else if(DP[i-1][j]>DP[i][j-1])
        {
  i--;  }
        else
                j--;

   }
   for(int k=0; k<temp; k++){
```

```
                cout<<common[k];
            }
    cout<<endl;

        return ;
}
int main() {
        string a,b;
        cin>>a>>b;
LCS(a,b);
  clock_t time_req;
        time_req = clock();          cout<<"Time Taken For Finding longest
Common Subsequence is: "<<(float)time_req/CLOCKS_PER_SEC << "
seconds" << endl;

        return 0;
}
```

Output:



GRAPHS ALGORITHMS

Aim: To implement Prim's Algorithm

Code:

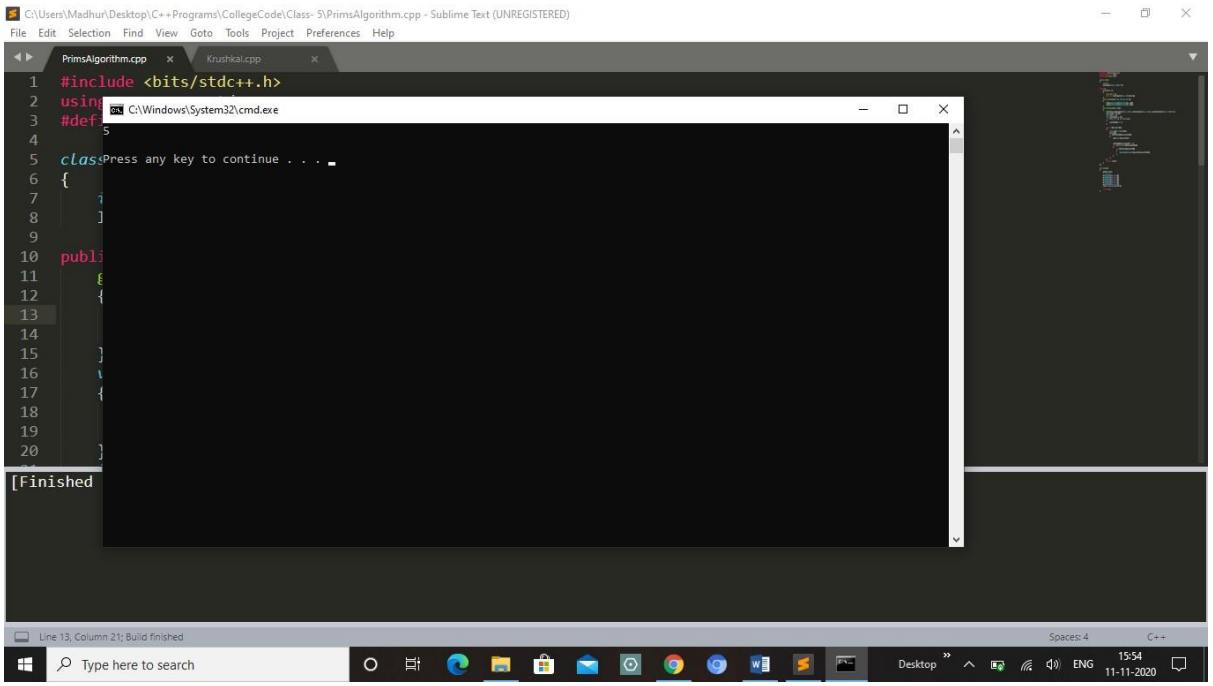
```
#include <iostream>  #include
<bits/stdc++.h> using
namespace std;
#define ll long long


class graph
{    int v;
list<pair<int, int>> *l;
```

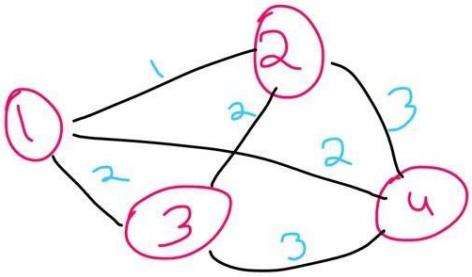
```
public:
graph(int v)
   {
      this->v = v;
      l = new list<pair<int, int>>[v+1];
   }
   void addedge(int x, int y, int w)
   {
      l[x].push_back(make_pair(y, w));
l[y].push_back(make_pair(x, w));
   }
   int primsmsp(int src)
   {
      priority_queue<pair<int, int>, vector<pair<int, int>>, greater<pair<int,
int>>> h;
      h.push({0, src});
int cost = 0;        int
visited[v + 1];        for (int i
= 0; i <= v; i++)
      {
         visited[i] = 0;
      }

      while (!h.empty())
      {
         auto best = h.top();
         h.pop();
```

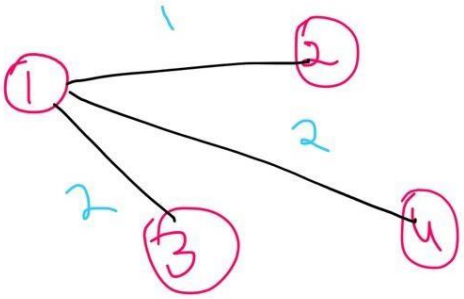
```
        if (!visited[best.second])
        {
            cost += best.first;


            visited[best.second] = 1;
for (auto p : l[best.second])
            {
                if (!visited[p.first])
                {
                    h.push(make_pair(p.second,p.first));
                }
            }
        }
      }
      return cost;
   }
};

int main()
{
   graph g(4);
   // input 1
   g.addedge(1, 2, 1);
   g.addedge(1, 3, 2);
   g.addedge(1, 4, 2);
```

```
    g.addedge(2, 3, 2);
    g.addedge(2, 4, 3);
    g.addedge(3, 4, 3);
cout << g.primsmsp(1);

    return 0;
}
```

Output:







Aim: To implement Krushkal Algorithm

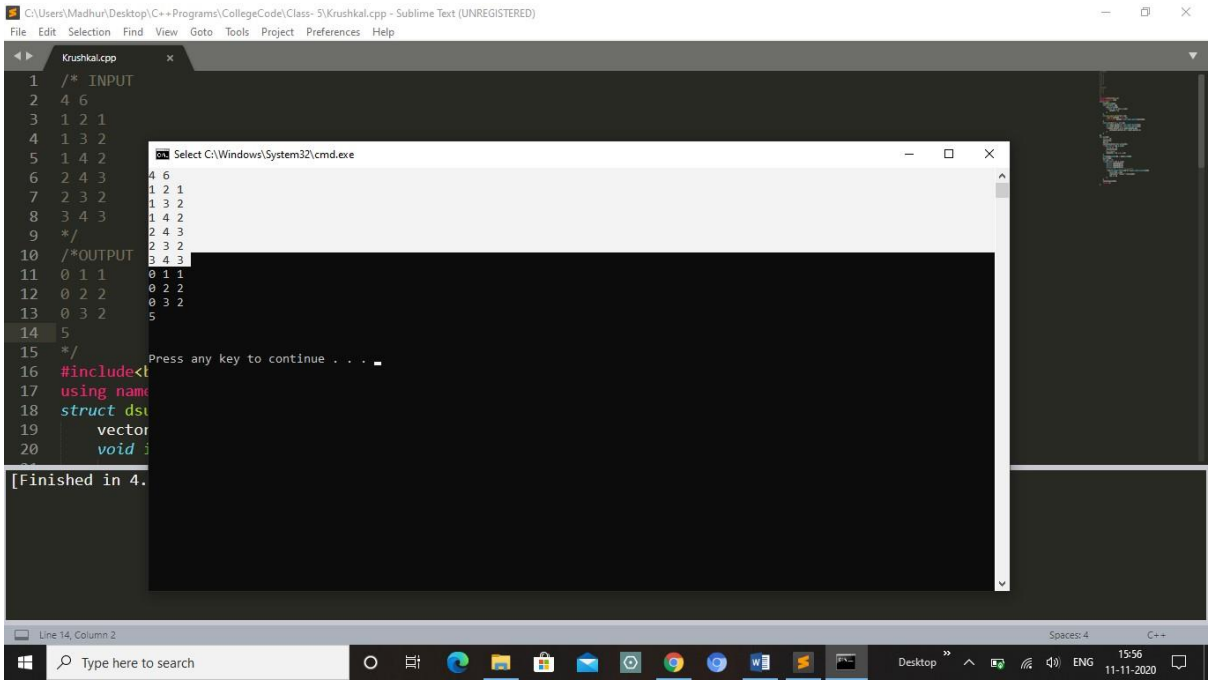
Code:

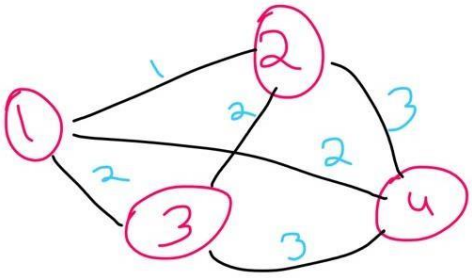
```
#include <iostream>
#include<bits/stdc++.h>
using namespace std; struct
dsu{    vector<int>par;
void init(int n){
```

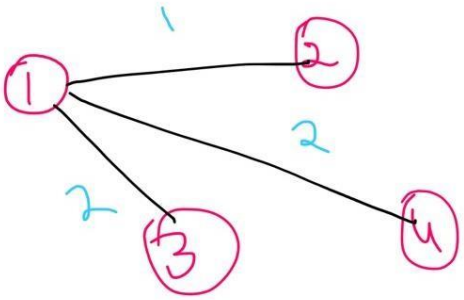
```
par.resize(n);        for(int
i=0 ; i<n ; i++){            par[i]
= i;
        }
    }
    int get_super_parent(int x){
if(x == par[x]) return x;
        else return par[x] = get_super_parent(par[x]);
    }
    void unite(int x , int y){
        int super_parent_x = get_super_parent(x);
int super_parent_y = get_super_parent(y);
if(super_parent_x != super_parent_y){
par[super_parent_x] = super_parent_y;
        }
    }
};
int main(){
int n , m;
cin>>n>>m;
dsu d;
    vector<vector<int>> edges(m);
for(int i=0 ; i<m ; i++){        int x,
y,  w;
        cin>>x>>y>>w;
```

```
      x-- ; y--;
edges[i] = {w , x , y};
   }
   sort(edges.begin() , edges.end());
   d.init(n);    int ans =
0;    for(int i=0 ; i<m ;
i++){       int w =
edges[i][0];       int x =
edges[i][1];       int y =
edges[i][2];
      //cout<<w<<" "<<x<<" "<<y<<endl;
if(d.get_super_parent(x) != d.get_super_parent(y)){
         d.unite(x , y);
         cout<<x<<" "<<y<<" "<<w<<endl;
ans += w;
      }
   }
   cout<<ans<<endl;
return 0;
}
```

Output:







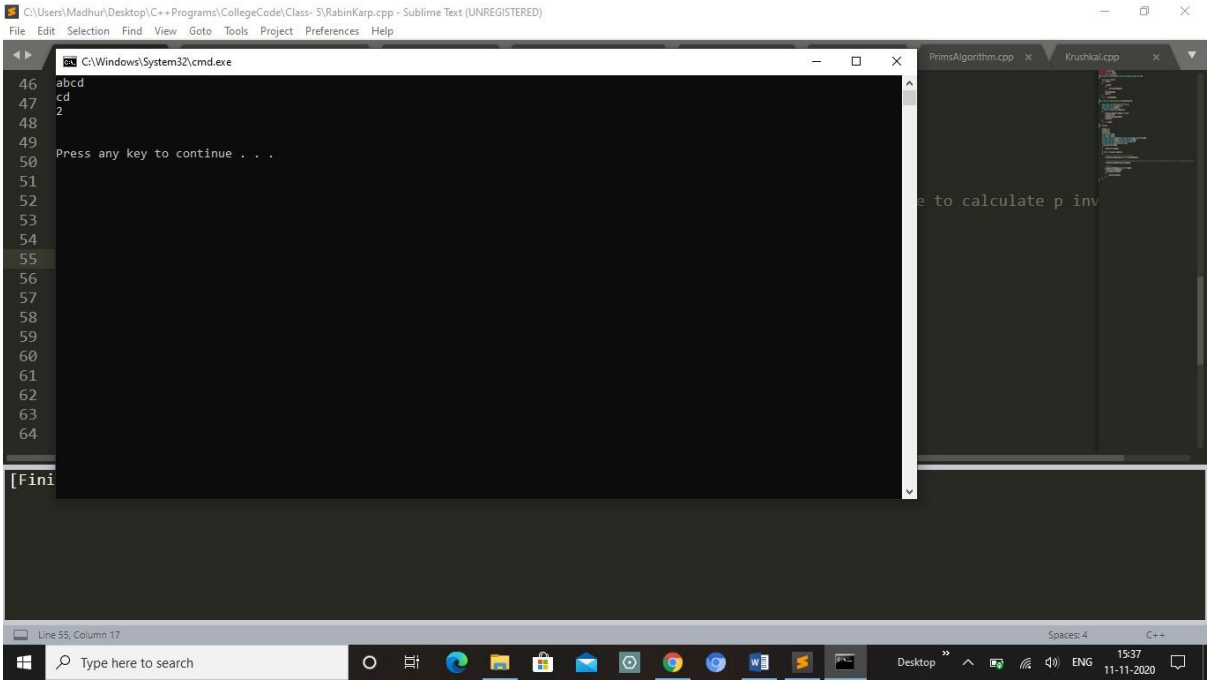
Aim: Rabin Karp Algorithm implementation Code:

```
#include <iostream>  using
namespace std; #define
mod 1000000007
long long int power(long long int a,long long int b)
{
   long long res=1;
while(b>0)
   {
      if(b&1)
      {
         res=(res*a)%mod;
      }
      a=(a*a)%mod;
b=b/2;
   }
   return (res)%mod;
}
long long int poly_hash_string(string n)
{
   long long int p=31,m=1000000007;
long long int p_power=1;     long long
int hash=0;     for(int
i=0;i<n.size();i++)
   {
      hash+=p_power* (n[i]-'a'+1);
p_power*=p;
p_power=(p_power)%m;        hash%=m;
```

```
    }
    return hash;
}
int main()
{    string a;
string b;
cin>>a>>b;    int
n=a.size();    int
m=b.size();
    long long int a_hash=poly_hash_string(a.substr(0,m));
long long int b_hash=poly_hash_string(b);    long long
int inverse_p=power(31,mod-2);    long long int
p=power(31,m-1);    if(a_hash==b_hash)
    {
        cout<<"0"<<endl;
    }
    for(int i=1;i+m-1<n;i++)
    {
        //Step 1 Delete The first window        a_hash=(a_hash-(a[i-1]-
'a'+1)+mod)%mod;        //Step 2 Divide by B that is nothing but hash/p that
is equal o hash*(p^-1) so we have to calculate p inverse
a_hash=(a_hash*inverse_p)%mod;

        //Step 3
        a_hash+=(p*((a[i+m-1]-'a'+1)));
a_hash=(a_hash)%mod;        if(a_hash==b_hash)
```

```
    {
        cout<<i<<endl;
    }
  }
}
```

Output:



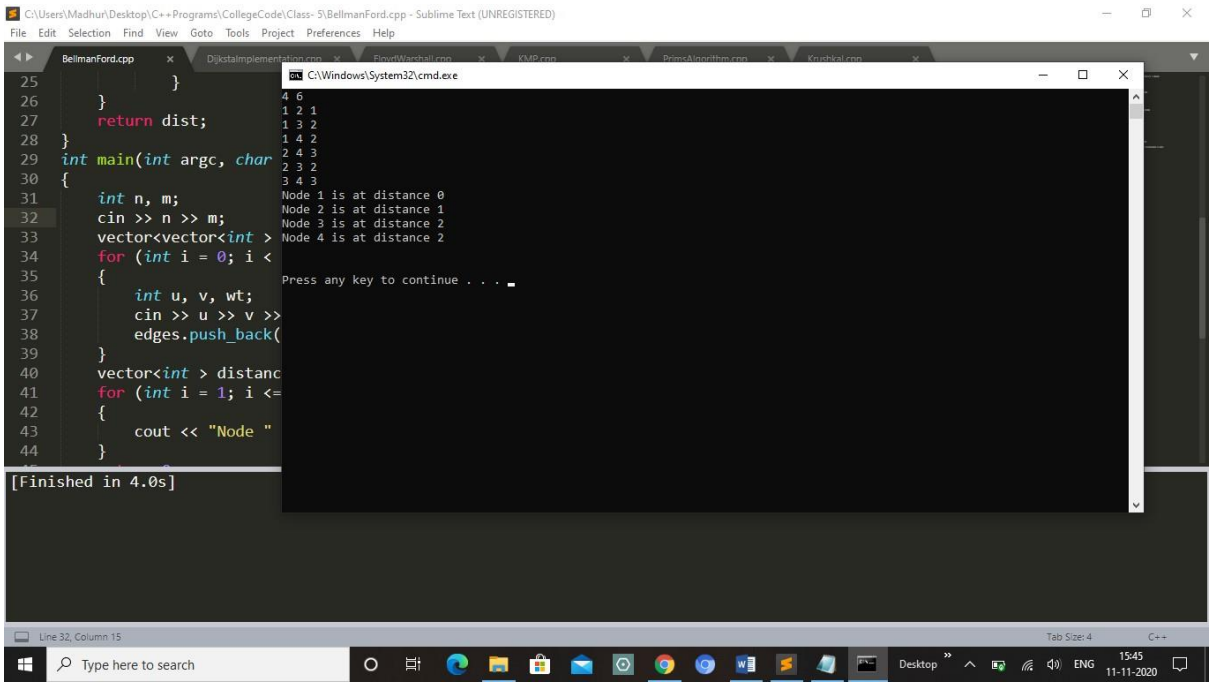
Aim: To implement Bellman Fords Algorithm Code:

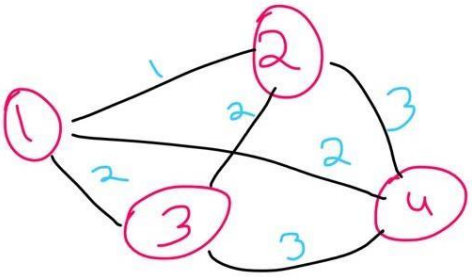
```
#include <iostream>
#include<bits/stdc++.h> using
namespace std;
vector<int > Bellman_Ford(int V, int src, vector<vector<int> > edges){
vector<int> dist(V+1, INT_MAX);          dist[src] =0;
        for (int i = 0; i < V-1; ++i)
        {
```

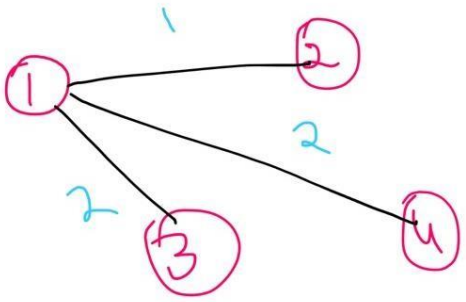
```
                for(auto edge : edges){
                int u = edge[0];
        int v = edge[1];
int wt = edge[2];
                        if(dist[u]!=INT_MAX and dist[u]+wt<dist[v]){
                dist[v] = dist[u]+wt;
                        }
                }
        }
        //negative cycle     for(auto
edge : edges ){                 int u =
edge[0];                        int v =
edge[1];                        int wt =
edge[2];
                        if (dist[u]!=INT_MAX and dist[u]+wt < dist[v]){
                cout<<"Negative weight cycle found";
exit(0);
                        }
        }
        return dist;
}
int main(int argc, char const *argv[])
{
        int n, m;       cin >> n >> m;
vector<vector<int > > edges;     for
(int i = 0; i < m; ++i)
```

```
        {
                int u, v, wt;           cin >> u
>> v >> wt;            edges.push_back( {u,
v, wt} );
        }
        vector<int > distances = Bellman_Ford(n, 1, edges);
for (int i = 1; i <= n; ++i)
        {
                cout << "Node " << i << " is at distance " << distances[i] <<
endl;
        }
        return 0;
}
```

Output:







Aim: To Implement Dijkstra Algorithm

Code:

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

template<typename T> class
Graph{
   unordered_map<T, list<pair<T,int> > > m;

public:

   void addEdge(T u,T v,int dist,bool bidir=true){
m[u].push_back(make_pair(v,dist));
```

```
        if(bidir){
            m[v].push_back(make_pair(u,dist));
        }

    }
    void printAdj(){
        //Let try to print the adj list
        //Iterate over all the key value pairs in the map
for(auto j:m){

            cout<<j.first<<"->";

            //Iterater over the list of cities
for(auto l: j.second){
                    cout<<"("<<l.first<<","<<l.second<<")";

            }
            cout<<endl;
        }

}
    void dijsktraSSSP(T src){

        unordered_map<T,int> dist;
```

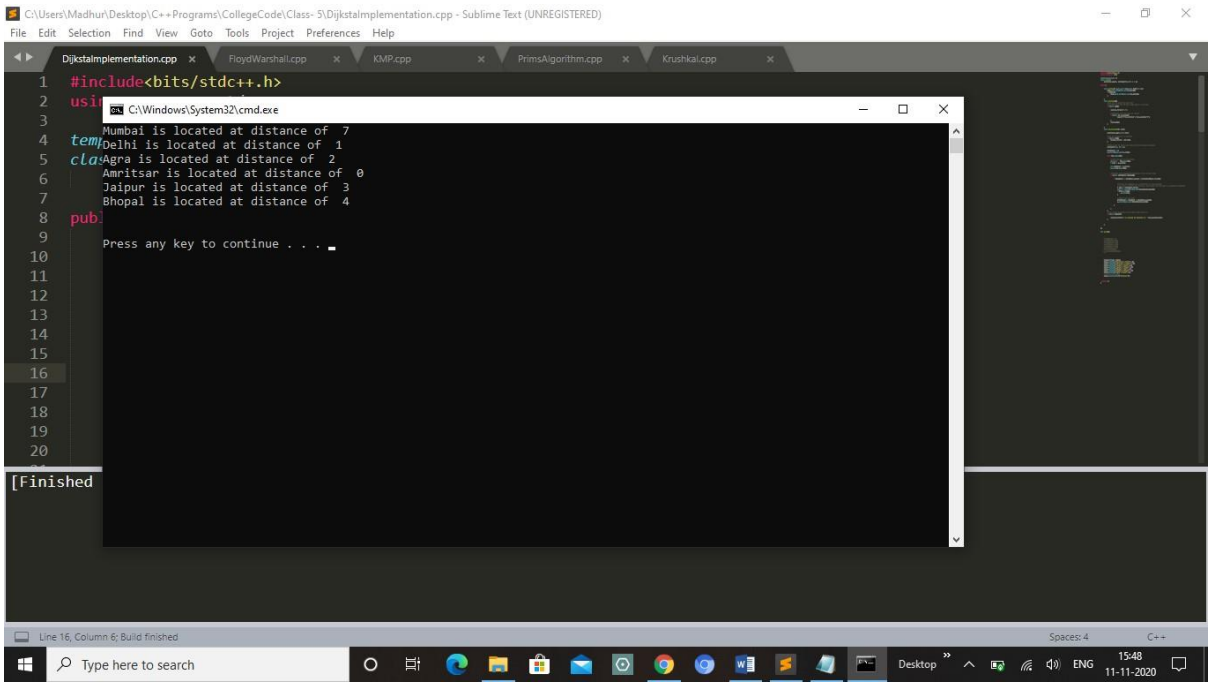
```
        //Set all distance to infinity
for(auto j:m){          dist[j.first]
= INT_MAX;
        }

        //Make a set to find a out node with the minimum distance
set<pair<int, T> > s;

        dist[src] = 0;
        s.insert(make_pair(0,src));

        while(!s.empty()){

            //Find the pair at the front.
            auto p =   *(s.begin());
            T node = p.second;

            int nodeDist = p.first;
            s.erase(s.begin());


            //Iterate over neighbours/children of the current node
for(auto childPair: m[node]){

                if(nodeDist + childPair.second < dist[childPair.first]){
```

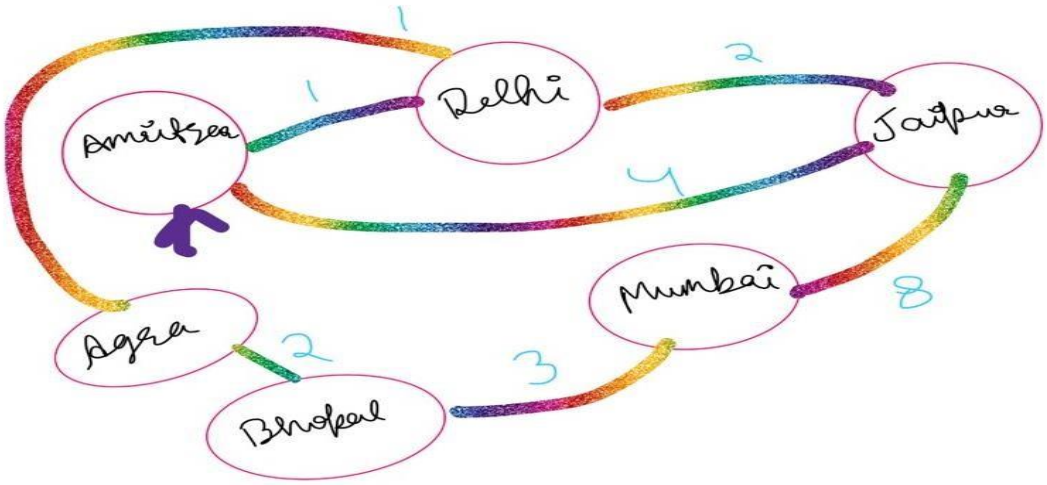
```
            //In the set updation of a particular is not possible
            // we have to remove the old pair, and insert the new pair to
simulation updation
            T dest = childPair.first;
            auto f = s.find( make_pair(dist[dest],dest));
if(f!=s.end()){
               s.erase(f);
            }

            //Insert the new pair
            dist[dest] = nodeDist + childPair.second;
            s.insert(make_pair(dist[dest],dest));

          }

        }
      }
      //Lets print distance to all other node from src
for(auto d:dist){

        cout<<d.first<<" is located at distance of  "<<d.second<<endl;
      }


   }

};
```

```
int main(){



    /*Graph<int> g;
    g.addEdge(1,2,1);
    g.addEdge(1,3,4);
    g.addEdge(2,3,1);
    g.addEdge(3,4,2);
    g.addEdge(1,4,7);
//g.printAdj();
   // g.dijsktraSSSP(1);
    */

    Graph<string> india;
india.addEdge("Amritsar","Delhi",1);
india.addEdge("Amritsar","Jaipur",4);
india.addEdge("Jaipur","Delhi",2);
india.addEdge("Jaipur","Mumbai",8);
india.addEdge("Bhopal","Agra",2);
india.addEdge("Mumbai","Bhopal",3);
india.addEdge("Agra","Delhi",1);
//india.printAdj();     india.dijsktraSSSP("Amritsar");
```

```
 return 0;
}
```

Output:



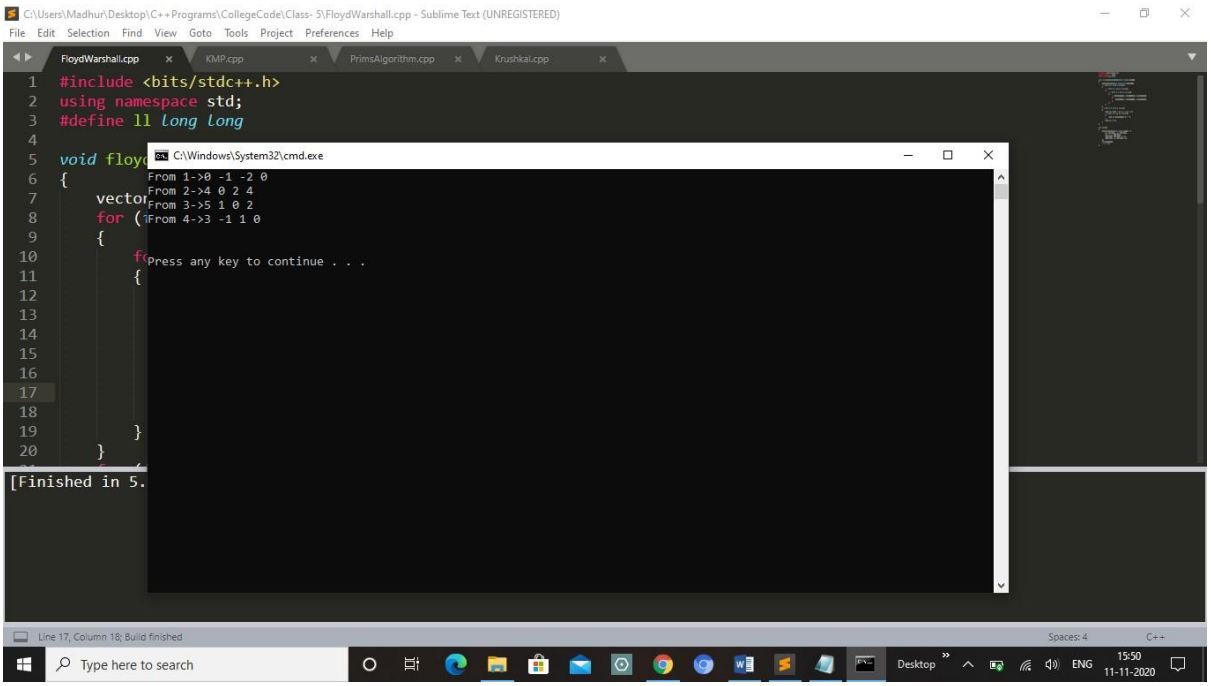


Aim: To Implement Floyd Warshall Code:

```
#include <iostream>  #include
<bits/stdc++.h> using
namespace std;
#define ll long long

void floyd(vector<vector<long long>> graph)
{
   vector<vector<long long>> dist(graph);
for (int k = 0; k < 4; k++)
   {
      for (int u = 0; u < 4; u++)
      {
         for (int v = 0; v < 4; v++)
         {
            if (dist[u][k] + dist[k][v] < dist[u][v])
            {
               dist[u][v] = dist[u][k] + dist[k][v];
            }
         }
      }
   }
   for (int i = 0; i < 4; i++)
   {
      cout << "From " << i + 1 << "->";
      for (int j = 0; j < 4; j++)
      {
```

```
            cout << dist[i][j] << " ";
        }
        cout << "\n";
    }
}

int main()
{
    vector<vector<long long>> graph = {
        {0, INT_MAX, -2, INT_MAX},
        {4, 0, 3, INT_MAX},
        {INT_MAX, INT_MAX, 0, 2},
        {INT_MAX, -1, INT_MAX, 0},
    };
    floyd(graph);
return 0;
}
```

Output:



STRING MATCHING ALGORITHMS

Aim: To Implement KMP algorithm

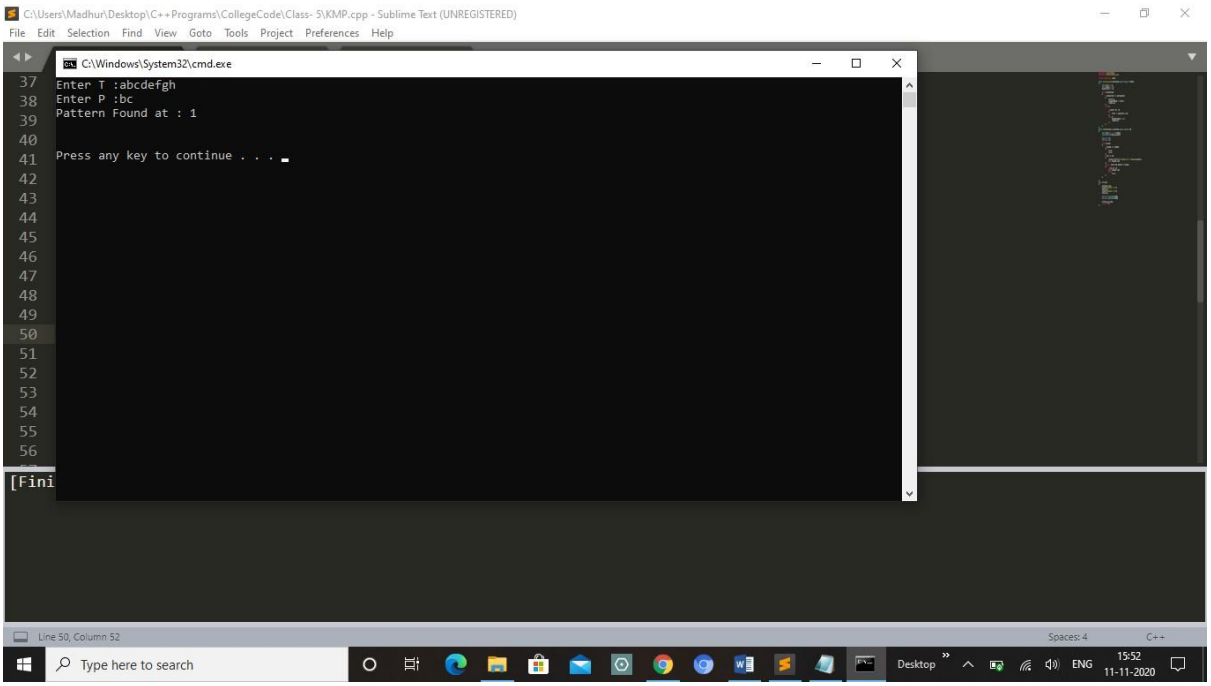
Code:

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

using namespace std;

void calculateLPS(string p,int m,int *lps)
{    int left = 0;
int right = 1;
lps[left] = 0;

   while(right<m)
   {
      if(p[left] == p[right])
      {          left++;
lps[right] = left;
right++;
      }
else{

         if(left != 0)
         {              left =
lps[left-1];
```

```
        }           else{
lps[right] = 0;
right++;
        }
      }
   }
}
void KMP(string t,string p,int n,int m)
{
   int *lps = new int[m];
calculateLPS(p,m,lps);

   int i = 0;
int j = 0;

   while(i<n)
   {       if(p[j] ==
t[i])
      {
i++;
j++;
      }
      if(j == m)
      {
         cout<<"Pattern Found at : "<<i-j<<endl;
         j = lps[j-1];
      }
```

```
        else if(i<n && p[j] != t[i])
        {           if(j
!= 0)           j =
lps[j-1];
else                i++;
        }
    }
 }
int main()
{    string t,p;
cout<<"Enter T :";
cin>>t;
cout<<"Enter P :";
cin>>p;

    int n = t.length();
int m = p.length();

    KMP(t,p,n,m);
    return 0;
}
```

Output:


C:\Users\Madhur\Desktop\C++Programs\CollegeCode\Class- 5\KMP.cpp - Sublime Text (UNREGISTERED)
File Edit Selection Find View Goto Tools Project Preferences Help
C:\Windows\System32\cmd.exe
Enter T :abcdefgh
Enter P :bc
Pattern Found at : 1
Press any key to continue . . .
Line 50, Column 52
Spaces: 4
C++
Type here to search
Desktop
ENG
15:52
11-11-2020