Assignment:-#2

Insertion Sort:-

Algorithm:-
Input:- An array of n elements A[n].
output:-An array of Sorted elements.

```
INSERTION SORT(A,n):
    FOR i=2 to n
        insert element A[i] into already sorted subarray A[1 to i-1]
        by pairwise element swaps down to its right position.
```

Analysis of Algorithm:-

- T(n)=T(n-1)+n-1
- Worst case:- when an array is reverse sorted.
  $N^2/4+O(N)$ comparisons and $N^2/4+O(N)$ swaps.
- Best case:-When an array is already sorted.
  N-1 comparisons and no swaps.

Code:-

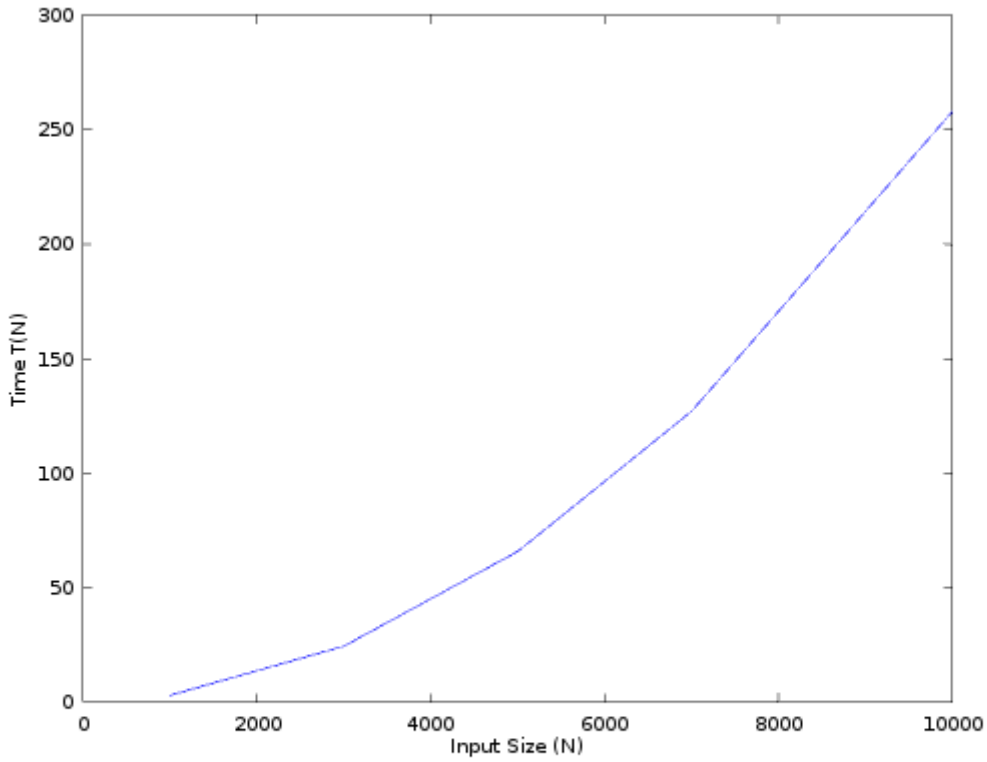
```
#include <iostream>
#include <vector>
#include <stdlib.h>
#include <map>
using namespace std;

int main(){
    int n;
    cin>>n;
    int arr[n];
    for(int i=0;i<n;i++){
        arr[i]=rand()%10000;
    }

    for(int i=0;i<n;i++)
        cout<<arr[i]<<" ";


    for(int i=1;i<n;i++){
       if(arr[i]<arr[i-1]){
        int temp=arr[i];
        for(int j=i-1;j>=0;j--){
            if(arr[j]>temp){
                arr[j+1]=arr[j];
                arr[j]=temp;
                temp=arr[j];
            }
        }
      }
//    for(int i=0;i<n;i++)
  //      cout<<arr[i]<<" ";
   //  cout<<endl;
```

```
}
      for(int i=0;i<n;i++)
            cout<<arr[i]<<" ";
         return 0;
}
```

Input/Output:-

Input n	Output T(n)
1000	2.9280
3000	23.9130
5000	68.3430
7000	126.5890
10000	260.9180

Graph:-



Merge Sort:-

Algorithm:-

Input:- An array of n elements A[n].

output:-An array of Sorted elements.

MERGE SORT(A,n):

IF n==1

done(nothing to sort)

ELSE

recursively sort A[1 to n/2] and A[n/2+1 to n]

MERGE the two sorted subarray.

Code:-

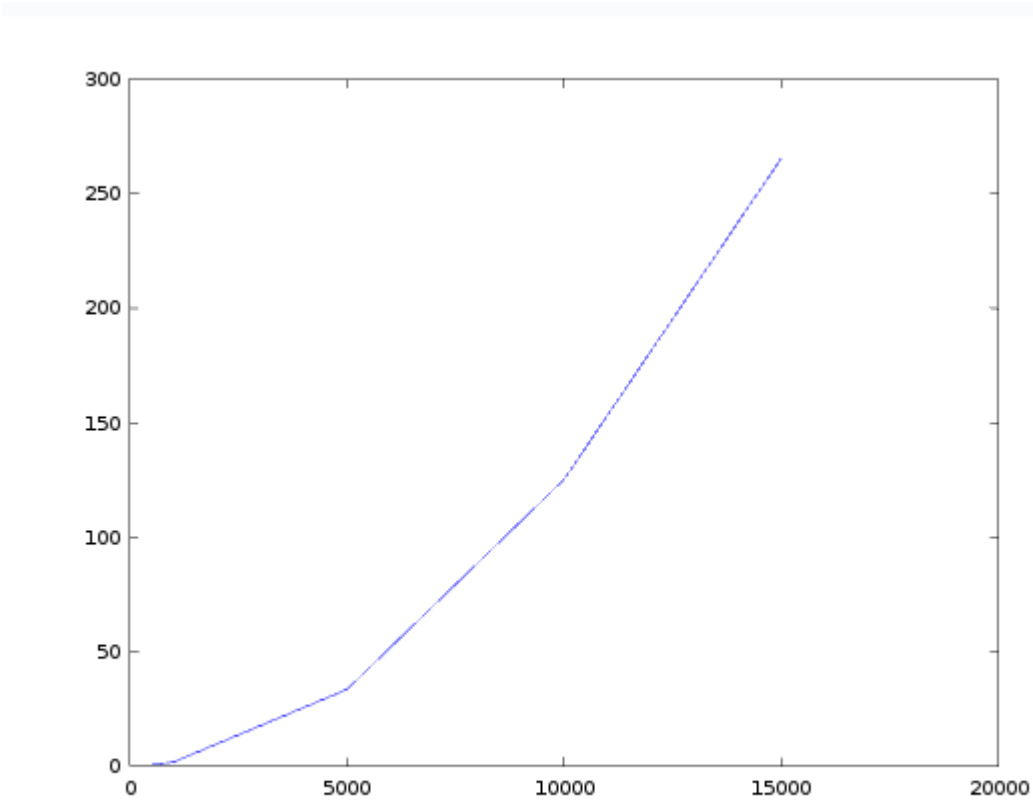
```
#include <iostream>
#include <stdlib.h>
#include <vector>
#include <ctime>
using namespace std;
int comp,swap;

int combine(vector<int>a,int l,int h,int m){
   int i=l,j=m+1,k=l,c[100000];
   while(i<=m && j<=h){
      if(a[i]<a[j]){
         c[k]=a[i];
         k++,i++;
      }
      else{
         c[k]=a[j];
         k++,j++;
      }
   }
   while(i<=m){
      c[k]=a[i];
      k++,i++;
   }
   while(j<=h){
      c[k]=a[j];
      k++,j++;
   }
   for(i=l;i<k;i++){
      a[i]=c[i];
   }
}

void partition(vector<int> v,int low,int high){
      if(low>=high)
           return;
      int mid=(low+high)/2;
      partition(v,low,mid);
      partition(v,mid+1,high);
      combine(v,low,high,mid);
}
int main(){
      int n;
      cin>>n;
      vector<int>v;
```

```
      for(int i=0;i<n;i++){
            v.push_back(rand()%100000);
       }
       for(int i=0;i<n;i++){
            cout<<v[i]<<" ";
       }
       cout<<endl;
 int start_s=clock();
    partition(v,0,n-1);
for(int i=0;i<n;i++)
        cout<<v[i]<<" ";
 int stop_s=clock();
cout << "time: " << (stop_s-start_s)/double(CLOCKS_PER_SEC)*1000 << endl;
      return 0;
    }
```

Analysis of Algorithm:-

- T(n)= O(1) if n==1,
  2T(n/2)+ Cn if n>1.
  = O(n*log(n)).
- Merging the subarray involves log(n) passes.On each pass,each subarray element is used in at most one comparison.So the number of comparisons per pass is n.Hence the number of comparison for Merge sort is O(n*log(n)).
- Merge sort require O(n) additional storage for the subarray.

Graph:-



Input/Output:-

Input n	Output T(n)
500	0.70000
1000	1.72500
5000	33.50300
10000	125.19900
15000	265.45400

Bubble Sort:-

Algorithm:-
Input:- An array of n elements A[n].
output:-An array of Sorted elements.
BUBBLE SORT(A,n):

```
FOR i=n-1 to 1
    FOR  j=0 to i
        IF A[j] > A[j+1]
            Swap A[j] and A[j+1]
        END IF
    END FOR
END FOR
```

Code:-

```
#include <iostream>
#include <stdlib.h>
#include <vector>
#include <ctime>
using namespace std;
int comp,swap;
 int main(){
    int n;
    cin>>n;
    vector<int> v;
    for(int i=0;i<n;i++){
        v.push_back(rand()%10000);
    }
    int start_s=clock();
    for(int i=n-1;i>=1;i--){
        for(int j=0;j<i;j++){
            if(v[j]>v[j+1]){
                int temp=v[j];
                v[j]=v[j+1];
                v[j+1]=temp;
            }
        }
       }
    for(int i=0;i<n;i++){
        cout<<v[i]<<" ";
    }
int stop_s=clock();
cout << "time: " << (stop_s-start_s)/double(CLOCKS_PER_SEC)*1000 << endl;
    return 0;
```

```
}
```

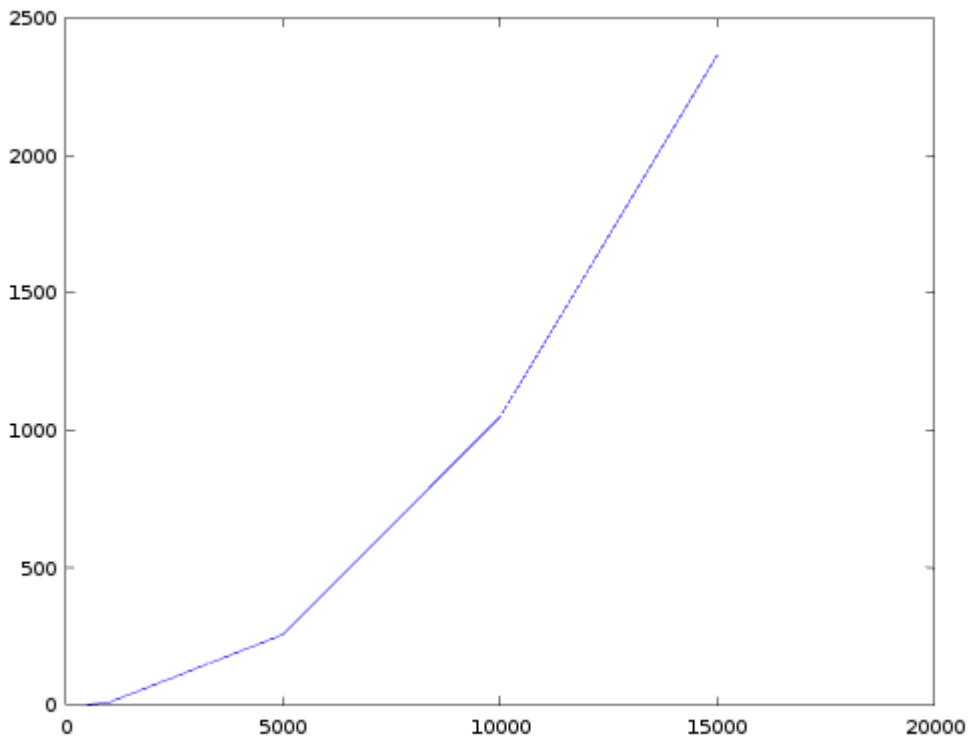
Analysis of Algorithm:-

- Best case :- O(n)
  This time complexity can occur if the array is already sorted, and that means that no swap occurred and only 1 iteration of n elements
- Worst case:- $O(n^2)$.
  The worst case is if the array is already sorted but in descending order. This means that in the first iteration it would have to look at n elements, then after that it would look n-1 elements (since the biggest integer is at the end) and so on and so forth till 1 comparison occurs.
  $O(n)=n+n-1+n-2$ .....$+1=(n*(n+1))/2=O(n^2)$

Input/Output:-

Input n	Output T(n)
500	2.4280
1000	10.2140
5000	256.7070
10000	1049.3700
15000	2365.3000

Graph:-



Heap Sort:-

Algorithm:-
Input:- An array of n elements A[n].
output:-An array of Sorted elements.

Assumptions:-
root of a tree:-first element of an array corresponds to i=1.
parent(i)=i/2 : return the index of a node's parent.
left child(i)=2*i :returns the index of the node's left child.
right child(i)=2*i+1: return the index of the node's right child.

```
MAXHEAPIFY(A,i,n):
    l=left child(i),r=right child(i),temp=A[i]
    WHILE l<=n
        IF l<n && A[r]>A[l]
            l=l+1
        IF temp>A[l]
            BREAK
        ELSE IF temp<=A[l]
            A[l/2]=A[l]
            l=2*l
        END IF
    END WHILE
    A[l/2]=temp
    RETURN


BUILDMAXHEAP(A,n)
    FOR i=n/2 to 1
        MAXHEAPIFY(A,i,n)
    END FOR
```

HEAPSORT(A,n)

- Build Max Heap from an unsorted array.
- Find maximum element A[1]
- Swap elements A[n] and A[1]
    - Now max element is at the end of the array.
- Discard node n from heap and decrement the heap size variable.
- Go to step 2 unless heap is empty.

Code:-

```
#include <iostream>
#include <stdlib.h>
#include <vector>
using namespace std;
int comp,swap;
```

```
void maxheapify(int a[],int i,int n){
   int l,temp=a[i];
   l=2*i;
   while(l<=n){
      if(l<n && a[l+1]>a[l])
         l=l+1;
      if(temp>a[l])
         break;
      else if(temp<=a[l]){

         a[l/2]=a[l];
         l=2*l;
      }
   }
   a[l/2]=temp;
   return;
}

void heapsort(int a[], int n){
   int temp;
   for(int i=n;i>=2;i--){
      temp=a[i];
      a[i]=a[1];
      a[1]=temp;
      maxheapify(a,1,i-1);
   }
}

void heapbuild(int a[],int n){
   for(int i=n/2;i>=1;i--){
      maxheapify(a,i,n);
   }
}

int main()
{
   int n,x;
   cin>>n;
   int a[100000];
      for(int i=1;i<=n;i++){
         a[i]=rand()%1000;
      }


      for(int i=1;i<=n;i++){
         cout<<a[i]<<" ";
      }
      cout<<endl;
   heapbuild(a,n);
   heapsort(a,n);
   for(int i=1;i<=n;i++)
      cout<<a[i]<<" ";
   return 0;
}
```

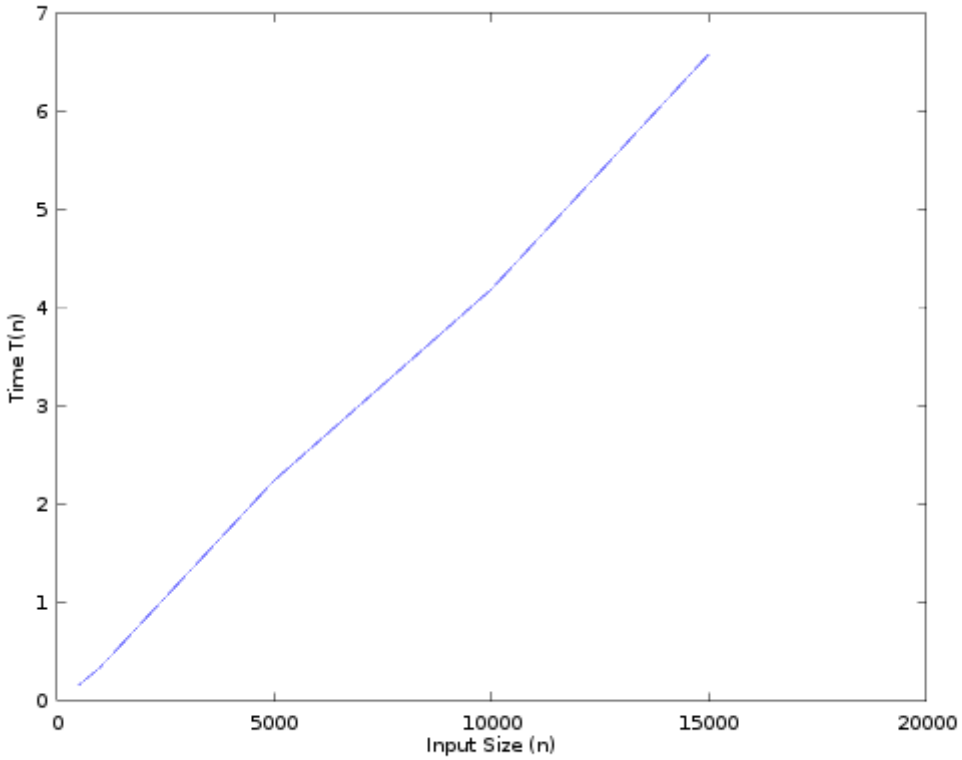
Analysis of Algorithm:-

- Max heapify takes log(n) time to modify heap as per the heap property.
- After n iterations the Heap is empty. Every iterator involves a swap and a max_heapify operation ,hence it takes O(log(n)) time.

- Therefore ,overall it takes O(n*log(n)) time.

In case if we have to find the k^{th} largest element then heap sort is best way to finding it and it takes O(k*log(n)) time.

Graph:-



Input/Output:-

Input n	Output T(n)
500	0.15800
1000	0.34500
5000	2.24500
10000	4.19300
15000	6.58400

Counting Sort:-

Algorithm:-

Input:- An array of n elements A[n].
output:-An array ans[n] of Sorted elements.
Assumptions:-
an array freq[MAX] for count of an element.

```
COUNTINGSORT(A,n):
    MAX=max(A[0],A[1].....,A[n-1])
    FOR i=0 to n
        freq[A[i]]++
    END FOR
    FOR i=1 to MAX
        freq[i]=freq[i]+freq[i-1]
    END FOR
    FOR i=0 to n
        x=A[i]
        ans[freq[x]]=x
    END FOR
```

Code:-

```
#include <iostream>
#include <stdlib.h>
#include <vector>
#include <ctime>
using namespace std;
int comp,swap;

int main()
{
   int n,x;
   cin>>n;
   int arr[n+1],ans[n+1],ma=0;
   for(int i=0;i<n;i++){
        arr[i]=rand()%10000;
        ma=max(ma,arr[i]);
      }
   for(int i=0;i<n;i++){
      cout<<arr[i]<<" ";
   }
   cout<<endl;
   int start_s=clock();
   int freq[ma+1]={0};
   for(int i=0;i<n;i++){
      freq[arr[i]]++;
   }
   for(int i=1;i<=ma+1;i++){
      freq[i]+=freq[i-1];
   }
   for(int i=0;i<n;i++){
      int x=arr[i];
```

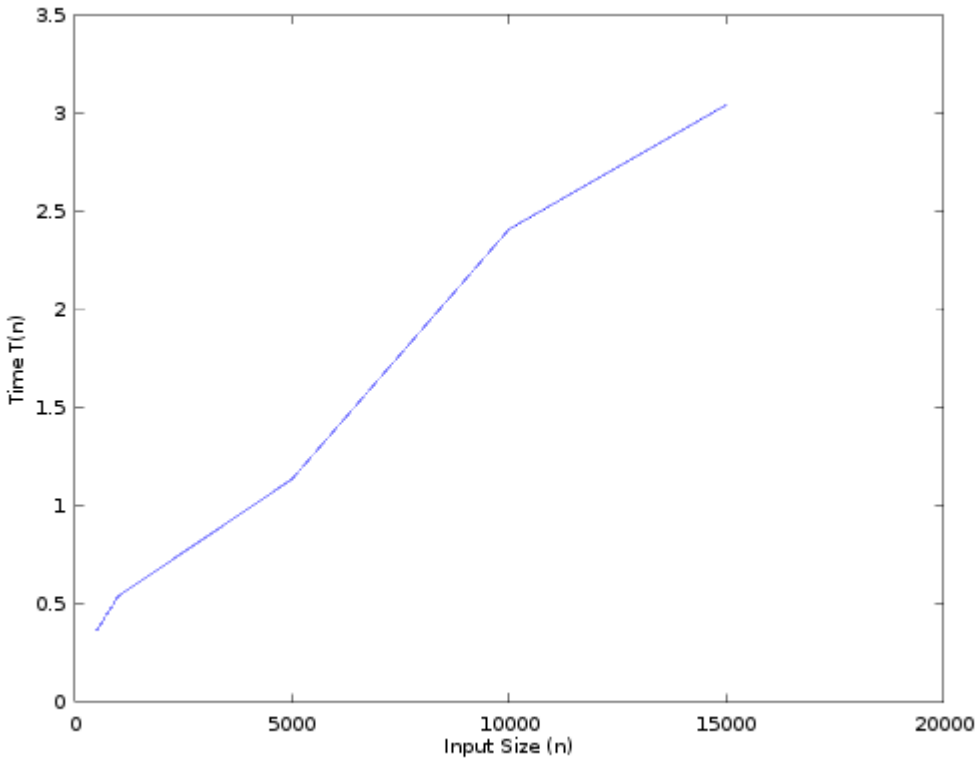
```
        ans[freq[x]]=x;
        freq[x]--;
    }
    for(int i=1;i<n+1;i++){
        cout<<ans[i]<<" ";
    }
 int stop_s=clock();
cout << "time: " << (stop_s-start_s)/double(CLOCKS_PER_SEC)*1000 << endl;
    return 0;
}
```

Analysis of Algorithm:-

- Time Complexity: O(n+k) where n is the number of elements in input array and k is the range of input.
- Auxiliary Space: O(n+k)
- Counting sort is efficient if the range of input data is not greater than the number of elements to be sorted
- It is not a comparison based sorting. It running time complexity is O(n) with space proportional to the range of data.
- ***What if the elements are in range from 1 to n^2?***

  We can't use counting sort because counting sort will take $O(n^2)$ which is worse than comparison based sorting algorithms.

Graph:-



Input/Output:-

Input n	Output T(n)
500	0.36200
1000	0.54000
5000	1.13600
10000	2.41000
15000	3.04300

Radix Sort:-

Algorithm:-
Input:- An array of n elements A[n].
output:-An array ans[n] of Sorted elements.
Assumptions:-
An array freq[MAX] for count of an element.

```
RADIXSORT(A,n):
    MAX=max(A[0],A[1].....,A[n-1])
    i=1
    WHILE  MAX/i > 0
        COUNTSORT(A,i,n)
        i=i*10
    END WHILE
```

Code:-

```
#include <iostream>
#include <stdlib.h>
#include <vector>
#include <ctime>
using namespace std;
int comp,swap;

int countSort(int arr[],int mod,int n){
      int freq[10]={0},ans[n];
       for(int i=0;i<n;i++){
          freq[(arr[i]/mod)%10]++;
        }
   for(int i=1;i<=9;i++){
      freq[i]+=freq[i-1];
   }
   for(int i=n-1;i>=0;i--){
      int x=(arr[i]/mod)%10;
      ans[freq[x]]=arr[i];
      freq[x]--;
   }
   for(int i=1;i<=n;i++){
      arr[i-1]=ans[i];
    //   cout<<ans[i]<<" ";
   }
   //cout<<endl;
```

```
}

int main()
{
   int n,x;
   cin>>n;
   int arr[n+1],ans[n+1],ma=0;
   for(int i=0;i<n;i++){
         arr[i]=rand()%10000;
         ma=max(ma,arr[i]);
      }
   for(int i=0;i<n;i++){
      cout<<arr[i]<<" ";
   }
   cout<<endl;
    int start_s=clock();

   for(int i=1;ma/i>0;i*=10){
      countSort(arr,i,n);
//      cout<<ans[i]<<" ";
   }
   for(int i=0;i<n;i++){
      cout<<arr[i]<<" ";
   }

 int stop_s=clock();
cout << "time: " << (stop_s-start_s)/double(CLOCKS_PER_SEC)*1000 << endl;
   return 0;
}
```

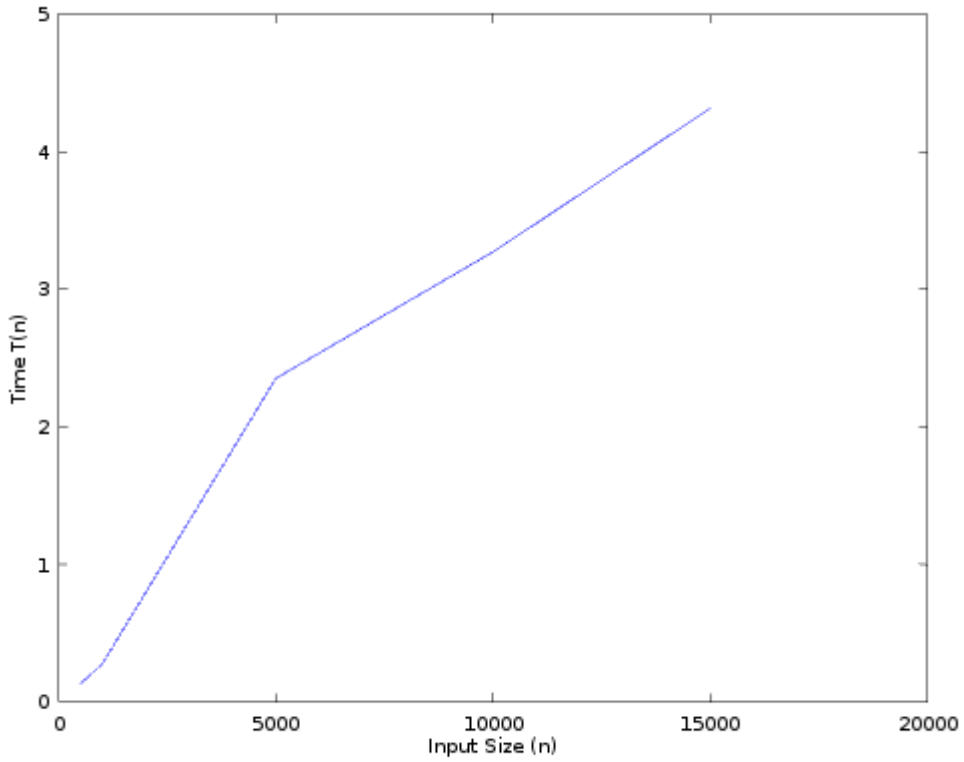
Analysis of Algorithm:-

- If every element lies in {0,1, . . . ,n},we need roughly $\log_{10}n$ digits in the decimal system. Each call to Counting sort takes time O(n+10)=O(n) because each digit has k=10 possibilities. So the total running time is. $O(n\log_{10}n)=O(n\lg n)$ even worse than Counting sort.
- If we have $\log_2 n$ bits for every digit, the running time of Radix appears to be better than Quick Sort for a wide range of inputs.

Input/Output:-

Input n	Output T(n)
500	0.13200
1000	0.27500
5000	2.35300
10000	3.27300
15000	4.31600

Graph:-



Bucket Sort:-

Algorithm:-
Input:- An array of n elements A[n].
output:-An array of Sorted elements.
Assumptions:-
input is uniformly distributed over a range.
A large set of floating point numbers which are in range from 0.0 to 1.0.
A vector B[n] for n buckets.

```
BUCKETSORT(A,n):
    FOR i=0 to n
        insert A[i] into bucket B[ceil(n*A[i])]
    END FOR
    FOR i=0 to n
        SORT(B[i])
        PRINT(B[i])
    END FOR
```

Code:-

```
#include <iostream>
#include <stdlib.h>
#include <vector>
#include <algorithm>
using namespace std;

int main()
{
   int n,l,temp,min;
   cin>>n;
   vector<double>v,B[20000];
      for(int i=0;i<n;i++){
        int d=rand();
       double x;
            if (d%100==0)
                x=1.0/(d%99);
            else
                x=1.0/(d%100);
          v.push_back(x);
      }
   for(int i=0;i<n;i++){
      cout<<v[i]<<" ";
   }

   cout<<endl;
   int start_s=clock();
   for(int i=0;i<n;i++){
      B[(int)(n*v[i])].push_back(v[i]);
   }

   for(int i=0;i<n;i++){
      sort(B[i].begin(),B[i].end());
   }
   for(int i=0;i<n;i++){
      l=B[i].size();
      for(int j=0;j<l;j++)
         cout<<B[i][j]<<" ";
   }
    int stop_s=clock();
cout << "time: " << (stop_s-start_s)/double(CLOCKS_PER_SEC)*1000 << endl;

   return 0;
}
```

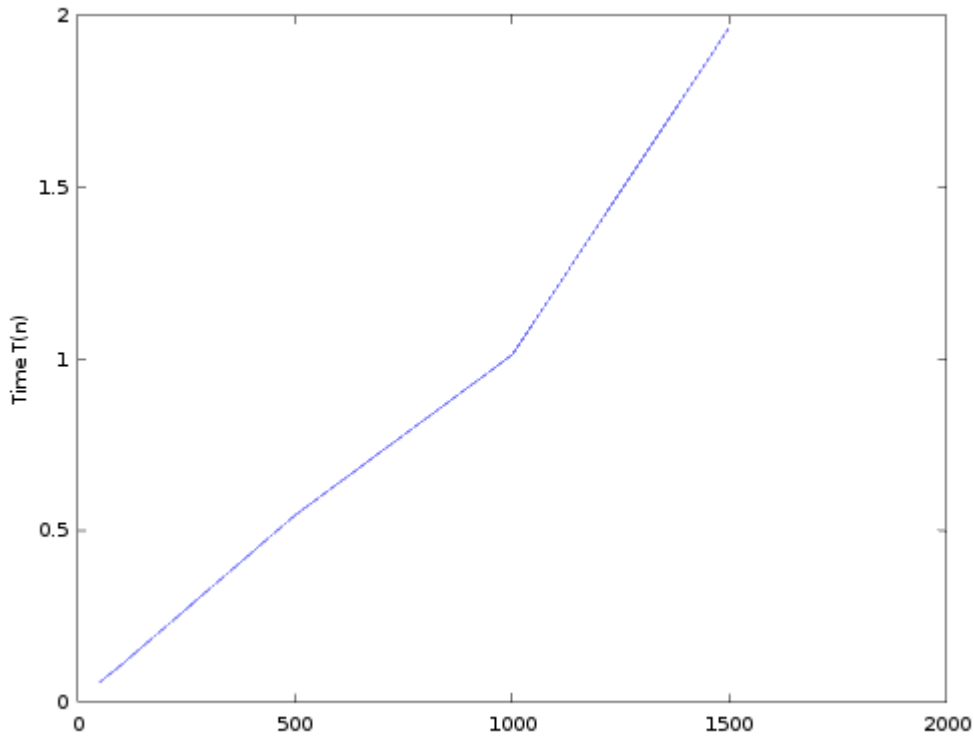
Analysis of Algorithm:-

- let n i be the random variable denoting the number of elements placed in bucket B[i]. Since worst case complexity of any sorting algorithm is $O(n^2)$ then running time of bucket sort is
  $T(n)=O(n)+Summation(n_i^2)$
  $T(n)=O(n)+n*O((n-1)/n)=O(n)$

Input/Output:-

Input n	Output T(n)
50	0.058000
100	0.108000
500	0.544000
1000	1.011000
5000	1.964000

Graph:-



Quick Sort:-

Algorithm:-
Input:- An array of n elements A[n].
output:-An array of Sorted elements.

```
PARTITION(A,l,r)
    p=A[r]
    i=l-1
```

```
        FOR j=l to r-1
                IF A[j]<=p
                        i++
                        SWAP(A[i],A[j])
                END IF
        END FOR
        SWAP(A[i+1],A[r])
        RETURN i+1


QUICKSORT(A,l,r):
        IF l<r
                mid=PARTITION(A,l,r)
                QUICKSORT(A,l,mid-1)
                QUICKSORT(A,mid+1,r)
        END IF
```

Code:-

```
#include <iostream>
#include <stdlib.h>
#include <vector>
#include <ctime>
using namespace std;
int comp,swap;

int partition(int arr[],int l,int r){
   int p=arr[r],i=l-1,j=l;
   //cout<<p<<endl;
   for(;j<r;j++){
      if(arr[j]<=p){
         i++;
         int temp=arr[i];
            arr[i]=arr[j];
            arr[j]=temp;
      }
//       cout<<arr[j]<<" ";
   }
  //  cout<<endl;
   int temp=arr[i+1];
      arr[i+1]=arr[r];
      arr[r]=temp;
   return i+1;
}

void quickSort(int arr[],int l,int r){
   if(l<r){
      int mid=partition(arr,l,r);
   //    cout<<mid<<endl;
      quickSort(arr,l,mid-1);
      quickSort(arr,mid+1,r);
   }
}
```

```
int main()
{
   int n,x;
   cin>>n;
   int arr[n+1];
   for(int i=0;i<n;i++){
         arr[i]=rand()%10000;
      }
   for(int i=0;i<n;i++){
      cout<<arr[i]<<" ";
   }
   cout<<endl;
    int start_s=clock();

    quickSort(arr,0,n-1);
//     cout<<ans[i]<<" ";
   for(int i=0;i<n;i++){
      cout<<arr[i]<<" ";
   }
 int stop_s=clock();
cout << "time: " << (stop_s-start_s)/double(CLOCKS_PER_SEC)*1000 << endl;
   return 0;
}
```

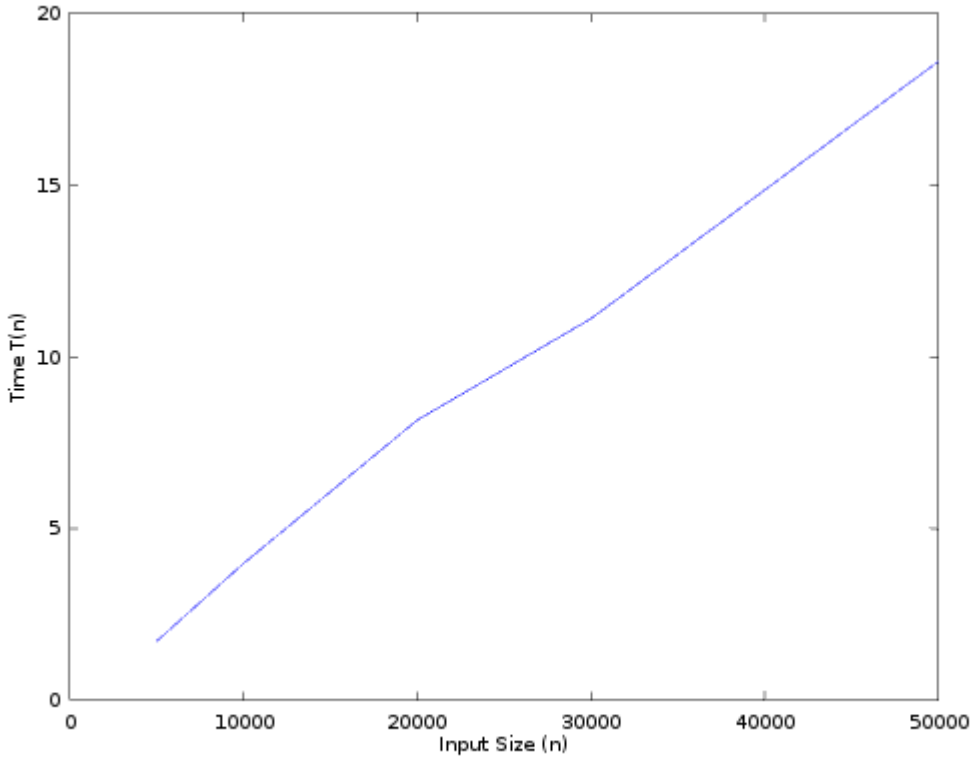
Analysis of Algorithm:-

- The running time of quicksort depends on whether the partitioning is balanced or unbalanced, which in turn depends on which elements are used for partitioning.If the partitioning is balanced, the algorithm runs asymptotically as fast as merge sort. If the partitioning is unbalanced, however, it can run asymptotically as slowly as insertion sort.
- Worst case:-The worst-case behavior for quicksort occurs when the partitioning routine produces one subproblem with n-1 elements and one with 0 elements.
    - $T(n)=T(n-1)+T(0)+O(n)=O(n^2)$
- Best case:- PARTITION produces two subproblems, each of size no more than n=2, since one is of size n/2 and one of size n/2-1
    - $T(n)=2T(n)+O(n)=O(n*\log(n))$
- We can get an idea of average case by considering the case when partition puts O(n/9) elements in one set and O(9n/10) elements in other set.
    - $T(n)=T(n/10)+T(9n/10)+O(n)=O(n*\log(n))$

Input/Output:-

Input n	Output T(n)
5000	1.7160
10000	3.9780
20000	8.1530
30000	11.1070
50000	18.5970

Graph : -



Selection Sort:-

Algorithm:-
Input:- An array of n elements A[n].
output:-An array of Sorted elements.

```
SELECTIONSORT(A,n)
    FOR i=0 to n
        MIN=A[i]
        l=i
        FOR j=i+1 to n
            IF MIN>A[j]
                MIN=A[j]
                l=j
            END IF
        END FOR
        SWAP(A[i],A[l])
    END FOR
```

Code:-

```
#include <iostream>
#include <stdlib.h>
#include <vector>
using namespace std;
int comp,swap;

int main()
{
   int n,l,temp,min;
   cin>>n;
   vector<int>v;
      for(int i=0;i<n;i++){
         v.push_back(rand()%100000);
      }
   for(int i=0;i<n;i++){
      cout<<v[i]<<" ";
   }
   for(int i=0;i<n-1;i++){
      min=v[i];
      l=i;
      for(int j=i+1;j<n;j++){
         if(min>v[j]){
            min=v[j];
            l=j;
         }
      }
      temp=v[i];
      v[i]=v[l];
      v[l]=temp;
   }

   cout<<endl;
   for(int i=0;i<n;i++){
      cout<<v[i]<<" ";
   }
   return 0;
}
```

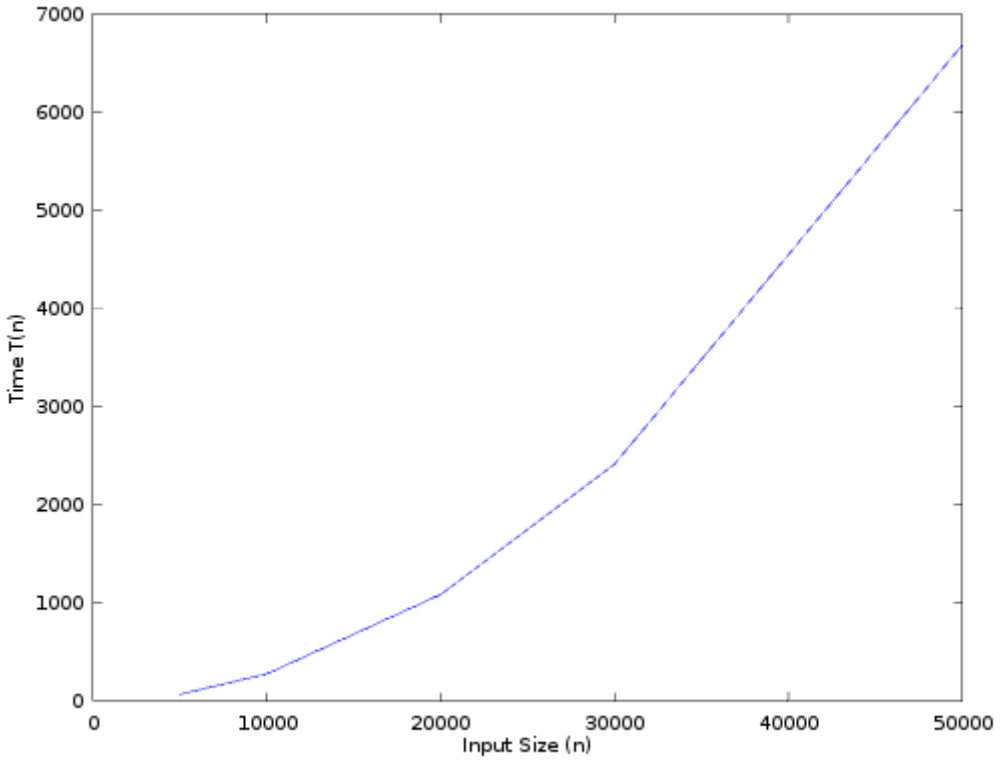
Analysis of Algorithm:-

- **Time Complexity:** $O(n^2)$ as there are two nested loops.
- **Auxiliary Space:** O(1)
- The good thing about selection sort is it never makes more than O(n) swaps and can be useful when memory write is a costly operation.

Input/Output:-

Input n	Output T(n)
5000	69.728
10000	277.361
20000	1087.100
30000	2414.770
50000	6686.090

Graph:-


7000
6000
5000
4000
3000
2000
1000
0
Time T(n)
0
10000
20000
30000
40000
50000
Input Size (n)