SORTING ALGORITHMS

1. Bubble Sort

Approach - Bubble sort is based on the idea of repeatedly comparing pairs of adjacent elements and then swapping their positions if they exist in the wrong order. This is the most time consuming sorting Algorithms as the time complexity of the Bubble sort goes like this :

Worst Case - $O(n^2)$

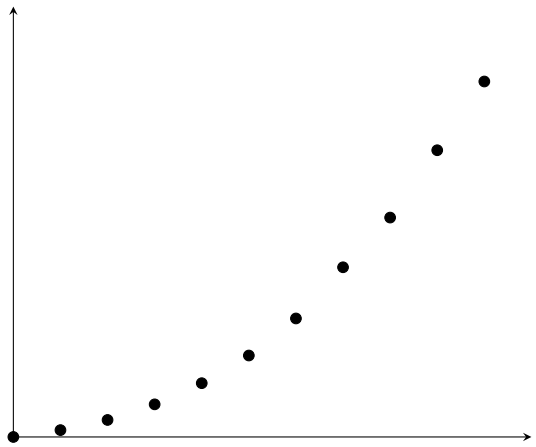
Average Case - $O(n^2)$

Best Case - $O(n)$

Program -

```
void bubble_sort(int *a,int n)
{
    for (int i=0; i<n;i++)
    {
        for(int j=0; j<n-i-1;i++)
        {
            if(a[j]>a[j+1])
            {
                swap(a[i],a[j]);
            }
        }
    }
}
```

Plot -



2. Selection Sort

Approach - The Selection sort algorithm is based on the idea of finding the minimum or maximum element in an unsorted array and then putting it in its correct position in a sorted array. Time complexity of the Bubble sort goes like this :

Worst Case - $O(n^2)$

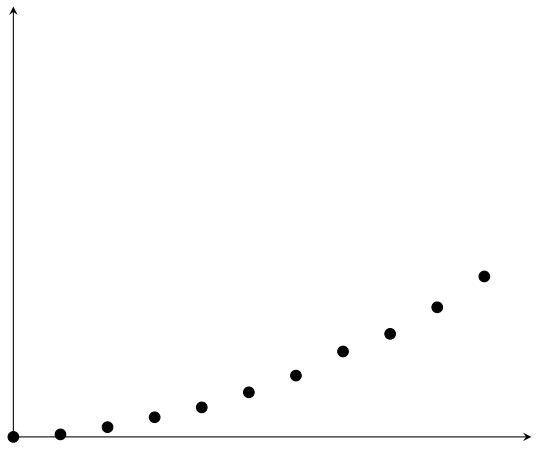
Average Case - $O(n^2)$

Best Case - $O(n^2)$

Program -

```
void selection_sort(int *a,int n)
{
    for(int i=0;i<n-1;i++)
    {
        imin=i;
        for(int j=i+1;j<n;j++)
        {
            if(a[j]<a[imin])
                imin=j;
        }
        swap(a[imin],a[i])
    }
}
```

Plot -



3. Insertion Sort

Approach - Insertion sort is based on the idea that one element from the input elements is consumed in each iteration to find its correct position i.e, the position to which it belongs in a sorted array. Time complexity of the Bubble sort goes like this :

Worst Case - $O(n^2)$

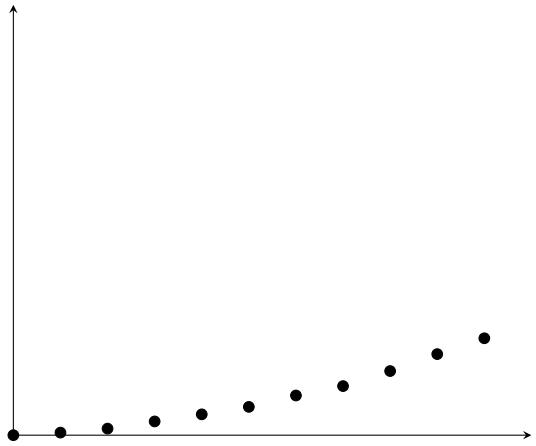
Average Case - $O(n^2)$

Best Case - $O(n)$

Program -

```
void insertion_sort(int *a,int n)
{
    for(int i=1;i<n;i++){
        value = a[i];
        hole =i;
        while(hole>0&&a[hole-1]>value){
            a[hole] = a[hole-1];
            hole--;}
        a[hole] =value;
    }
}
```

Plot -



4. Merge Sort

Approach - Merge sort is a divide-and-conquer algorithm based on the idea of breaking down a list into several sub-lists until each sublist consists of a single element and merging those sublists in a manner that results into a sorted list. Time complexity of the Bubble sort goes like this :

Worst Case - $O(nlogn)$

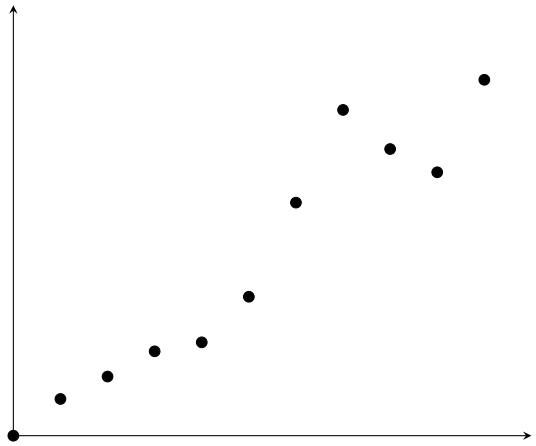
Average Case - $O(nlogn)$

Best Case - $O(nlogn)$

Program -

```
void merge_sort(int *a,int n)
{
    int mid,*l,*r;
    if(size==1) { return 1;}
    mid = size/2;
    l = new int [mid];
    r = new int [size-mid];
    for(int i=0;i<mid;i++) l[i] = a[i];
    for(int i=mid;i<size;i++) r[i-mid] = a[i];
    MergeSort(l,mid);
    MergeSort(r,size-mid);
    Merge(l,r,a,mid,size-mid);
}
```

Plot -



5. Quick Sort

Approach - Quick sort is based on the divide-and-conquer approach based on the idea of choosing one element as a pivot element and partitioning the array around it such that: Left side of pivot contains all the elements that are less than the pivot element Right side contains all elements greater than the pivot Time complexity of the Bubble sort goes like this :

Worst Case - $O(n^2)$

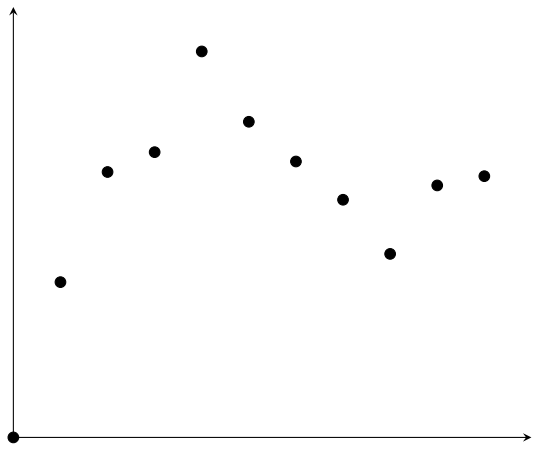
Average Case - $O(nlogn)$

Best Case - $O(nlogn)$

Program -

```
void quick_sort(int *a,int start,int end)
{
    int pIndex;
    if(start<end)
    {
        pIndex=partition(a,start,end);
        QuickSort(a,start,pIndex-1);
        QuickSort(a,pIndex+1,end);
    }
}
```

Plot -



6. Counting Sort

Approach - In Counting sort, the frequencies of distinct elements of the array to be sorted is counted and stored in an auxiliary array, by mapping its value as an index of the auxiliary array. Time complexity of the Bubble sort goes like this :

Worst Case - $O(n+k)$

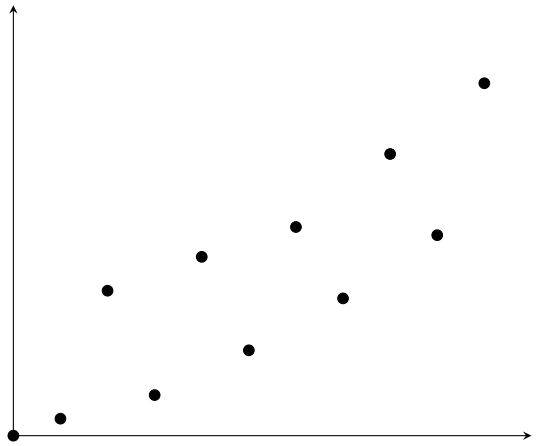
Average Case - $O(n+k)$

Best Case - $O(n+k)$

Program -

```
void counting_sort(int *arr,int n,int RANGE)
{
    int count[RANGE]={0};
    int out[n];
    for(i=0;i<n;i++) ++count[arr[i]];
    for(i=1;i<RANGE;i++) count[i]+=count[i-1];
    for(i=n-1;i>=0;i--){
    out[count[arr[i]]-1]=arr[i];
    --count[arr[i]];
    }
    for(i=0;i<n;i++) arr[i]=out[i];
}
```

Plot -



7. Heap Sort

Approach - Heaps can be used in sorting an array. In max-heaps, maximum element will always be at the root. Heap Sort uses this property of heap to sort the array.
Consider an array *Arr* which is to be sorted using Heap Sort.

1) Initially build a max heap of elements in Arr.
2) The root element, that is $Arr[1]$, will contain maximum element of *Arr*. After that, swap this element with the last element of *Arr* and heapify the max heap excluding the last element which is already in its correct position and then decrease the length of heap by one.
3) Repeat the step 2, until all the elements are in their correct position. Time complexity of the Bubble sort goes like this :

Worst Case - $O(nlogn)$
Average Case - $O(nlogn)$
Best Case - $O(nlogn)$

Program -

```
void heap_sort(int *a,int n)
{
    for (int i = n / 2 - 1; i >= 0; i--)
    heapify(arr, n, i);
    for (int i=n-1; i>=0; i--)
    {
        swap(arr[0], arr[i]);
        heapify(arr, i, 0);
    }
}
```

Plot -