Name	Best Case	Average Case	Worst Case	Memory	Stable	Method Used
Quick Sort	$nlogn$	$nlogn$	n^2	$logn$	No	Partitioning
Merge Sort	$nlogn$	$nlogn$	$nlogn$	n	Yes	Merging
Heap Sort	$nlogn$	$nlogn$	$nlogn$	1	No	Selection
Insertion Sort	n	n^2	n^2	1	Yes	Insertion
Tim Sort	n	$nlogn$	$nlogn$	n	Yes	Insertion & Merging
Selection Sort	n^2	n^2	n^2	1	No	Selection
Shell Sort	$nlogn$	$n^{4/3}$	$n^{3/2}$	1	No	Insertion
Bubble Sort	n	n^2	n^2	1	Yes	Exchanging

1. BUBBLE SORT

```
void bubbleSort(int arr[], int n)
{
    for(int i=0;i<n;i++)
    {
        bool flag=false;
        for(int j=0;j<n-i-1;j++)
        {
            if(arr[j]>arr[j+1])
            {
                flag=true;
                swap(arr[j],arr[j+1]);
            }
        }

        if(flag==false)
        {
            break;
        }
    }
}
```

2. INSERTION SORT

```
void insertionSort(int arr[], int n)
{

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

        while(j>=0 and arr[j]>key)
        {
            arr[j+1]=arr[j];
            j--;
        }

        arr[j+1]=key;
    }
}
```

3. Selection Sort

Input:

N = 5

arr[] = {4, 1, 3, 9, 7}

Output:

1 3 4 7 9

Explanation:

Maintain sorted (in bold) and unsorted subarrays.

Select 1. Array becomes **1** 4 3 9 7.

Select 3. Array becomes **1 3** 4 9 7.

Select 4. Array becomes **1 3 4** 9 7.

Select 7. Array becomes **1 3 4 7** 9.

Select 9. Array becomes **1 3 4 7 9**.

```
int select(int arr[], int i,int n)
{
    int min_index=i;

    for(int j=i+1;j<n;j++)
    {
        if(arr[j]<arr[min_index])
        {
            min_index=j;
        }
    }

    return min_index;
}

void selectionSort(int arr[], int n)
{
   for(int i=0;i<n;i++)
   {
       int j=select(arr,i,n);
       swap(arr[i],arr[j]);
   }
}
```

4. Quick Sort

Input:

N = 5

arr[] = { 4, 1, 3, 9, 7}

Output:

1 3 4 7 9

```
public:
void quickSort(int arr[], int low, int high)
{

    if(low<high)
    {
        int q=partition(arr,low,high);
        quickSort(arr,low,q-1);
        quickSort(arr,q+1,high);
    }

}

public:
int partition (int arr[], int low, int high)
{

   int pivot=arr[high];
   int i=low-1;

   for(int j=low;j<=(high-1);j++)
   {
       if(arr[j]<=pivot)
       {
           i++;
           swap(arr[i],arr[j]);
       }
   }

   swap(arr[i+1],arr[high]);

   return i+1;
}
```

5. Merge Sort

Given an array arr[], its starting position l and its ending position r. Sort the array using merge sort algorithm.

Example 1:

Input:

N = 5

arr[] = {4 1 3 9 7}

Output:

1 3 4 7 9

```
public:
void mergeSort(int arr[], int l, int r)
{
    if(l<r)
    {
        int m=l+(r-l)/2;
        mergeSort(arr,l,m);
        mergeSort(arr,m+1,r);
        merge(arr,l,m,r);
    }
}
```

```
void merge(int arr[], int l, int m, int r)
{
    int n1=m-l+1;
    int n2=r-m;

    int left[n1+2];
    int right[n2+2];

    for(int i=1;i<=n1;i++)
    {
        left[i]=arr[l+i-1];
    }

    for(int i=1;i<=n2;i++)
    {
        right[i]=arr[m+i];
    }

    left[n1+1]=INT_MAX;
    right[n2+1]=INT_MAX;

    int i=1,j=1;

    for(int k=l;k<=r;k++)
    {
        if(left[i]<=right[j])
        {
            arr[k]=left[i];
            i++;
        }
        else
        {
            arr[k]=right[j];
            j++;
        }
    }
}
```

HEAP SORT

```
//Heapify function to maintain heap property.
void heapify(int arr[], int n, int i)
{
  int largest=i;
  int left=2*i+1;
  int right=2*i+2;

  if(left<n and arr[largest]<arr[left])
  {
      largest=left;
  }

  if(right<n and arr[largest]<arr[right])
  {
      largest=right;
  }

  if(largest!=i)
  {
      swap(arr[largest],arr[i]);
      heapify(arr,n,largest);
  }
}

public:
//Function to build a Heap from array.
void buildHeap(int arr[], int n)
{
   for(int i=(n/2)-1;i>=0;i--)
   {
       heapify(arr,n,i);
   }
}
```

```
public:
//Function to sort an array using Heap Sort.
void heapSort(int arr[], int n)
{
    buildHeap(arr,n);

    for(int i=n-1;i>=0;i--)
    {
        swap(arr[i],arr[0]);
        heapify(arr,i,0);
    }
}
```

COUNTING SORT & Radix sort

Counting sort complexities

Case	Time
Best Case	O(n + k)
Average Case	O(n + k)
Worst Case	O(n + k)

Best Case Complexity - It occurs when there is no sorting required, i.e. the array is already sorted. The best-case time complexity of counting sort is **O(n + k)**.

Average Case Complexity - It occurs when the array elements are in jumbled order that is not properly ascending and not properly descending. The average case time complexity of counting sort is **O(n + k)**.

Worst Case Complexity - It occurs when the array elements are required to be sorted in reverse order. That means suppose you have to sort the array elements in ascending order, but its elements are in descending order. The worst-case time complexity of counting sort is **O(n + k)**.

Complexity Analysis of Radix Sort:

Time Complexity:

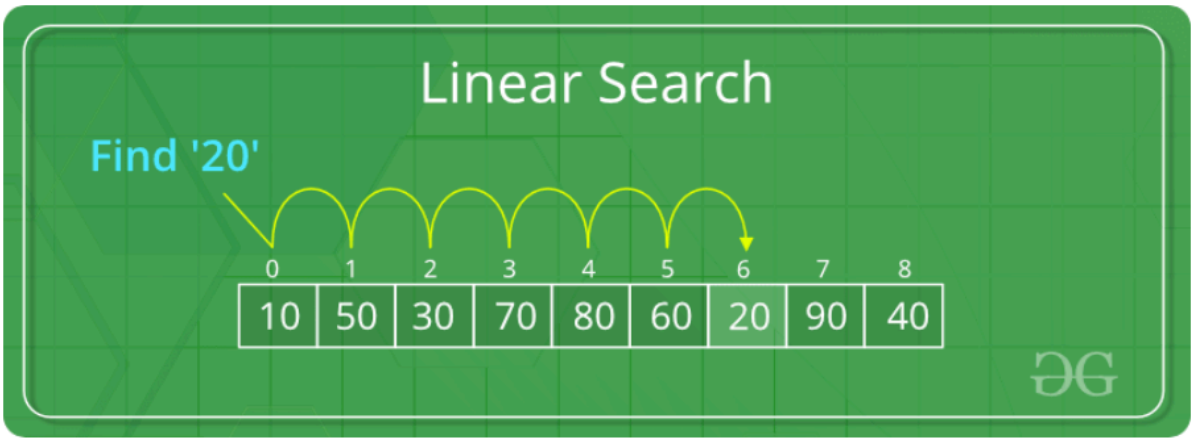
- Radix sort is a non-comparative integer sorting algorithm that sorts data with integer keys by grouping the keys by the individual digits which share the same significant position and value. It has a time complexity of **O(d * (n + b))**, where d is the number of digits, n is the number of elements, and b is the base of the number system being used.
- In practical implementations, radix sort is often faster than other comparison-based sorting algorithms, such as quicksort or merge sort, for large datasets, especially when the keys have many digits. However, its time complexity grows linearly with the number of digits, and so it is not as efficient for small datasets.

Auxiliary Space:

- Radix sort also has a space complexity of **O(n + b)**, where n is the number of elements and b is the base of the number system. This space complexity comes from the need to create buckets for each digit value and to copy the elements back to the original array after each digit has been sorted.

Linear Search

Linear Search to find the element “20” in a given list of numbers

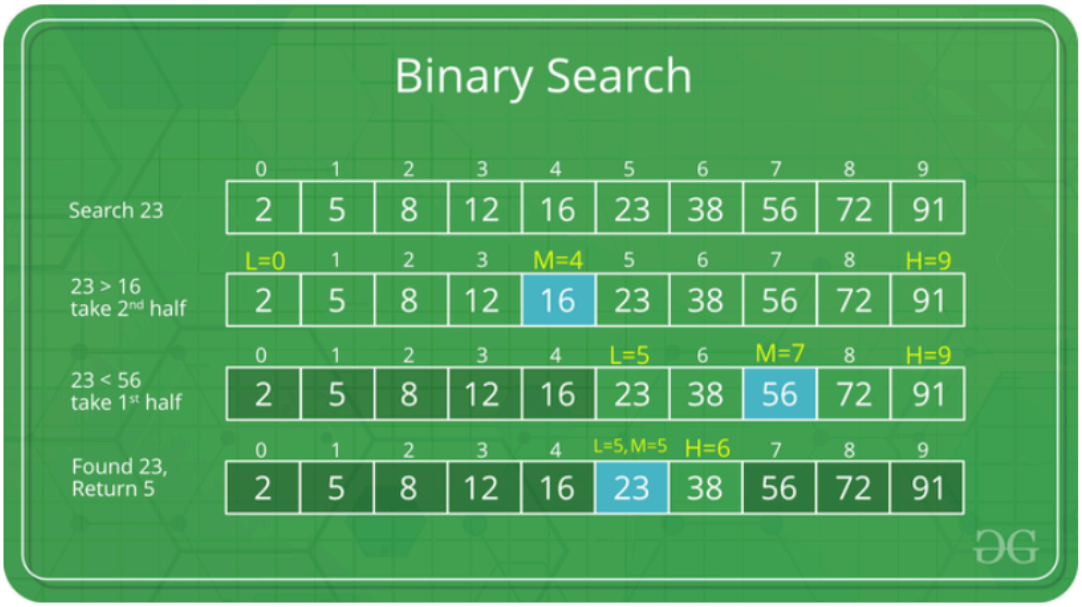


Linear-Search

```
int search(int arr[], int N, int X)
{
    for(int i=0;i<N;i++)
    {
        if(arr[i]==X)
        {
            return i;
        }
    }
    return -1;
}
```

Binary Search

Binary Search to find the element “23” in a given list of numbers




```c
int binarysearch(int arr[], int n, int k) {
    int low=0;
    int high=n-1;

    while(low<=high)
    {
        int mid=low+(high-low)/2;
        if(arr[mid]==k)
        {
            return mid;
        }
        else if(arr[mid]>k)
        {
            high=mid-1;
        }
        else
        {
            low=mid+1;
        }
    }

    return -1;
}
```