Bubble Sort

In Bubble sort, Each element of the array is compared with its adjacent element. The algorithm processes the list in passes. A list with n elements requires n-1 passes for sorting. Consider an array A of n elements whose elements are to be sorted by using Bubble sort. The algorithm processes like following.

1. In Pass 1, A[0] is compared with A[1], A[1] is compared with A[2], A[2] is compared with A[3] and so on. At the end of pass 1, the largest element of the list is placed at the highest index of the list.
2. In Pass 2, A[0] is compared with A[1], A[1] is compared with A[2] and so on. At the end of Pass 2 the second largest element of the list is placed at the second highest index of the list.
3. In pass n-1, A[0] is compared with A[1], A[1] is compared with A[2] and so on. At the end of this pass. The smallest element of the list is placed at the first index of the list.

Algorithm :

- **Step 1**: Repeat Step 2 For i = 0 to N-1
- **Step 2**: Repeat For J = i + 1 to N - I
- **Step 3**: IF A[J] > A[i]
  SWAP A[J] and A[i]
  [END OF INNER LOOP]
  [END OF OUTER LOOP
- **Step 4**: EXIT

Complexity

Scenario	Complexity
Space	O(1)
Worst case running time	$O(n^2)$
Average case running time	O(n)
Best case running time	$O(n^2)$

C Program

```
#include<stdio.h>
void main ()
{
   int i, j,temp;
   int a[10] = { 10, 9, 7, 101, 23, 44, 12, 78, 34, 23};
   for(i = 0; i<10; i++)
   {
      for(j = i+1; j<10; j++)
      {
         if(a[j] > a[i])
         {
            temp = a[i];
            a[i] = a[j];
            a[j] = temp;
         }
      }
   }
   printf("Printing Sorted Element List ...\n");
   for(i = 0; i<10; i++)
   {
      printf("%d\n",a[i]);
```

```
  }
}
```

Output:

```
Printing Sorted Element List . . .
7
9
10
12
23
34
34
44
78
101
```

Java Program

```
public class BubbleSort {
   public static void main(String[] args) {
   int[] a = {10, 9, 7, 101, 23, 44, 12, 78, 34, 23};
   for(int i=0;i<10;i++)
   {
      for (int j=0;j<10;j++)
      {
         if(a[i]<a[j])
         {
            int temp = a[i];
            a[i]=a[j];
            a[j] = temp;
         }
      }
   }
   System.out.println("Printing Sorted List ...");
   for(int i=0;i<10;i++)
   {
      System.out.println(a[i]);
   }
}
}
```

Output:

```
Printing Sorted List . . .
7
9
10
12
23
34
34
44
78
101
```

Python Program

```
a=[10, 9, 7, 101, 23, 44, 12, 78, 34, 23]
for i in range(0,len(a)):
    for j in range(i+1,len(a)):
        if a[j]<a[i]:
            temp = a[j]
            a[j]=a[i]
            a[i]=temp
print("Printing Sorted Element List...")
for i in a:
    print(i)
```

Output:

```
Printing Sorted Element List . . .
7
9
10
12
23
34
34
44
78
101
```

Bucket Sort

Bucket sort is also known as bin sort. It works by distributing the element into the array also called buckets. Buckets are sorted individually by using different sorting algorithm.

Complexity of Bucket Sort

Algorithm	Complexity
Space	O(1)
Worst Case	$O(n^2)$
Best Case	Ω(n+k)
Average Case	θ(n+k)

Algorithm

- Step 1 START
- Step 2 Set up an array of initially empty "buckets".
- Step 3 Scatter: Go over the original array, putting each object in its bucket.
- Step 4 Sort each non-empty bucket.
- Step 5 Gather: Visit the buckets in order and put all elements back into the original array.
- Step 6 STOP

Program

```
#include <stdio.h>
void Bucket_Sort(int array[], int n)
{
   int i, j;
   int count[n];
   for (i = 0; i < n; i++)
      count[i] = 0;

   for (i = 0; i < n; i++)
      (count[array[i]])++;

   for (i = 0, j = 0; i < n; i++)
      for(; count[i] > 0; (count[i])--)
         array[j++] = i;
}
/* End of Bucket_Sort() */

/* The main() begins */
int main()
{
   int array[100], i, num;

   printf("Enter the size of array : ");
   scanf("%d", &num);
   printf("Enter the %d elements to be sorted:\n",num);
   for (i = 0; i < num; i++)
      scanf("%d", &array[i]);
   printf("\nThe array of elements before sorting : \n");
   for (i = 0; i < num; i++)
      printf("%d ", array[i]);
   printf("\nThe array of elements after sorting : \n");
   Bucket_Sort(array, num);
   for (i = 0; i < num; i++)
      printf("%d ", array[i]);
   printf("\n");
   return 0;
}
```

COMB SORT

Comb Sort is the advance form of Bubble Sort. Bubble Sort compares all the adjacent values while comb sort removes all the turtle values or small values near the end of the list.

Factors affecting comb sort are:

- It improves on bubble sort by using gap of size more than 1.
- Gap starts with large value and shrinks by the factor of 1.3.
- Gap shrinks till value reaches 1.

Complexity

Algorithm	Complexity
Worst Case Complexity	$O(n^2)$
Best Case Complexity	$\theta(n \log n)$

Average Case Complexity	$\Omega(n^2/2^p)$ where p is number of increments.
Worst Case Space Complexity	O(1)

Algorithm

- STEP 1 START
- STEP 2 Calculate the gap value if gap value==1 goto step 5 else goto step 3
- STEP 3 Iterate over data set and compare each item with gap item then goto step 4.
- STEP 4 Swap the element if require else goto step 2
- STEP 5 Print the sorted array.
- STEP 6 STOP

Program

```
#include <stdio.h>
#include <stdlib.h>
 int newgap(int gap)
{
   gap = (gap * 10) / 13;
   if (gap == 9 || gap == 10)
      gap = 11;
   if (gap < 1)
      gap = 1;
   return gap;
}

void combsort(int a[], int aSize)
{
   int gap = aSize;
   int temp, i;
   for (;;)
   {
      gap = newgap(gap);
      int swapped = 0;
      for (i = 0; i < aSize - gap; i++)
      {
         int j = i + gap;
         if (a[i] > a[j])
         {
            temp = a[i];
            a[i] = a[j];
            a[j] = temp;
            swapped  =  1;
         }
      }
      if (gap  ==  1 && !swapped)
         break;
   }
}
int main ()
{
   int n, i;
   int *a;
   printf("Please insert the number of elements to be sorted: ");
   scanf("%d", &n);       // The total number of elements
   a  =  (int *)calloc(n, sizeof(int));
   for (i = 0;i< n;i++)
   {
      printf("Input element %d :", i);
      scanf("%d", &a[i]); // Adding the elements to the array
   }
```

```
   printf("unsorted list");       // Displaying the unsorted array
   for(i = 0;i < n;i++)
   {
       printf("%d", a[i]);
   }
   combsort(a, n);
   printf("Sorted list:\n"); // Display the sorted array
   for(i = 0;i < n;i++)
   {
      printf("%d ", (a[i]));
   }
   return 0;
}
```

Counting Sort

It is a sorting technique based on the keys i.e. objects are collected according to keys which are small integers. Counting sort calculates the number of occurrence of objects and stores its key values. New array is formed by adding previous key elements and assigning to objects.

Complexity

Time Complexity: O(n+k) is worst case where n is the number of element and k is the range of input.

Space Complexity: O(k) k is the range of input.

Complexity	Best Case	Average Case	Worst Case
Time Complexity	Ω(n+k)	θ(n+k)	O(n+k)
Space Complexity			O(k)

Limitation of Counting Sort

- It is effective when range is not greater than number of object.
- It is not comparison based complexity.
- It is also used as sub-algorithm for different algorithm.
- It uses partial hashing technique to count the occurrence.
- It is also used for negative inputs.

Algorithm

- STEP 1 START
- STEP 2 Store the input array
- STEP 3 Count the key values by number of occurrence of object
- STEP 4 Update the array by adding previous key elements and assigning to objects
- STEP 5 Sort by replacing the object into new array and key= key-1
- STEP 6 STOP

Program

```
#include <stdio.h>
void counting_sort(int A[], int k, int n)
{
   int i, j;
   int B[15], C[100];
   for (i = 0; i <= k; i++)
```

```
      C[i] = 0;
   for (j = 1; j <= n; j++)
      C[A[j]] = C[A[j]] + 1;
   for (i = 1; i <= k; i++)
      C[i] = C[i] + C[i-1];
   for (j = n; j >= 1; j--)
   {
      B[C[A[j]]] = A[j];
      C[A[j]] = C[A[j]] - 1;
   }
   printf("The Sorted array is : ");
   for (i = 1; i <= n; i++)
      printf("%d ", B[i]);
}
/*  End of counting_sort()  */

/*  The main() begins  */
int main()
{
   int n, k = 0, A[15], i;
   printf("Enter the number of input : ");
   scanf("%d", &n);
   printf("\nEnter the elements to be sorted :\n");
   for (i = 1; i <= n; i++)
   {
      scanf("%d", &A[i]);
      if (A[i] > k) {
         k = A[i];
      }
   }
   counting_sort(A, k, n);
   printf("\n");
   return 0;
}
```

Heap Sort

Heap sort processes the elements by creating the min heap or max heap using the elements of the given array. Min heap or max heap represents the ordering of the array in which root element represents the minimum or maximum element of the array. At each step, the root element of the heap gets deleted and stored into the sorted array and the heap will again be heapified.

The heap sort basically recursively performs two main operations.

- Build a heap H, using the elements of ARR.
- Repeatedly delete the root element of the heap formed in phase 1.

Complexity

Complexity	Best Case	Average Case	Worst case
Time Complexity	Ω(n log (n))	θ(n log (n))	O(n log (n))
Space Complexity			O(1)

Algorithm

HEAP_SORT(ARR, N)

- **Step 1**: [Build Heap H]
  Repeat for i=0 to N-1
  CALL INSERT_HEAP(ARR, N, ARR[i])
  [END OF LOOP]
- **Step 2**: Repeatedly Delete the root element
  Repeat while N > 0
  CALL Delete_Heap(ARR,N,VAL)
  SET N = N+1
  [END OF LOOP]
- **Step 3**: END

C Program

```
#include<stdio.h>
int temp;

void heapify(int arr[], int size, int i)
{
int largest = i;
int left = 2*i + 1;
int right = 2*i + 2;

if (left < size && arr[left] >arr[largest])
largest = left;

if (right < size && arr[right] > arr[largest])
largest = right;

if (largest != i)
{
temp = arr[i];
   arr[i]= arr[largest];
   arr[largest] = temp;
heapify(arr, size, largest);
}
}

void heapSort(int arr[], int size)
{
int i;
for (i = size / 2 - 1; i >= 0; i--)
heapify(arr, size, i);
for (i=size-1; i>=0; i--)
{
temp = arr[0];
   arr[0]= arr[i];
   arr[i] = temp;
heapify(arr, i, 0);
}
}

void main()
{
int arr[] = {1, 10, 2, 3, 4, 1, 2, 100,23, 2};
int i;
int size = sizeof(arr)/sizeof(arr[0]);

heapSort(arr, size);
```

```
printf("printing sorted elements\n");
for (i=0; i<size; ++i)
printf("%d\n",arr[i]);
}
```

Output:

```
printing sorted elements

1
1
2
2
2
3
4
10
23
100
```

Java Program

```
#include<stdio.h>
int temp;

void heapify(int arr[], int size, int i)
{
int largest = i;
int left = 2*i + 1;
int right = 2*i + 2;

if (left < size && arr[left] >arr[largest])
largest = left;

if (right < size && arr[right] > arr[largest])
largest = right;

if (largest != i)
{
temp = arr[i];
   arr[i]= arr[largest];
   arr[largest] = temp;
heapify(arr, size, largest);
}
}

void heapSort(int arr[], int size)
{
int i;
for (i = size / 2 - 1; i >= 0; i--)
heapify(arr, size, i);
for (i=size-1; i>=0; i--)
{
temp = arr[0];
   arr[0]= arr[i];
   arr[i] = temp;
heapify(arr, i, 0);
}
}
```

```
void main()
{
int arr[] = {1, 10, 2, 3, 4, 1, 2, 100, 23, 2};
int i;
int size = sizeof(arr)/sizeof(arr[0]);

heapSort(arr, size);

printf("printing sorted elements\n");
for (i=0; i<size; ++i)
printf("%d\n",arr[i]);
}
```

Output:

```
printing sorted elements

1
1
2
2
2
3
4
10
23
100
```

Insertion Sort

Insertion sort is the simple sorting algorithm which is commonly used in the daily lives while ordering a deck of cards. In this algorithm, we insert each element onto its proper place in the sorted array. This is less efficient than the other sort algorithms like quick sort, merge sort, etc.

Technique

Consider an array A whose elements are to be sorted. Initially, A[0] is the only element on the sorted set. In pass 1, A[1] is placed at its proper index in the array.

In pass 2, A[2] is placed at its proper index in the array. Likewise, in pass n-1, A[n-1] is placed at its proper index into the array.

To insert an element A[k] to its proper index, we must compare it with all other elements i.e. A[k-1], A[k-2], and so on until we find an element A[j] such that, A[j]<=A[k].

All the elements from A[k-1] to A[j] need to be shifted and A[k] will be moved to A[j+1].

Complexity

Complexity	Best Case	Average Case	Worst Case
Time	$\Omega(n)$	$\theta(n^2)$	$o(n^2)$
Space			o(1)

Algorithm

- **Step 1**: Repeat Steps 2 to 5 for K = 1 to N-1
- **Step 2**: SET TEMP = ARR[K]
- **Step 3**: SET J = K - 1
- **Step 4**: Repeat while TEMP <=ARR[J]
  SET ARR[J + 1] = ARR[J]
  SET J = J - 1
  [END OF INNER LOOP]
- **Step 5**: SET ARR[J + 1] = TEMP
  [END OF LOOP]
- **Step 6**: EXIT

C Program

```
#include<stdio.h>
void main ()
{
   int i,j, k,temp;
   int a[10] = { 10, 9, 7, 101, 23, 44, 12, 78, 34, 23};
   printf("\nprinting sorted elements...\n");
   for(k=1; k<10; k++)
   {
      temp = a[k];
      j= k-1;
      while(j>=0 && temp <= a[j])
      {
         a[j+1] = a[j];
         j = j-1;
      }
      a[j+1] = temp;
   }
   for(i=0;i<10;i++)
   {
      printf("\n%d\n",a[i]);
   }
}
```

Output:

```
Printing Sorted Elements . . .
7
9
10
12
23
23
34
44
78
101
```

Java Program

```
public class InsertionSort {
public static void main(String[] args) {
   int[] a = {10, 9, 7, 101, 23, 44, 12, 78, 34, 23};
   for(int k=1; k<10; k++)
   {
      int temp = a[k];
      int j= k-1;
      while(j>=0 && temp <= a[j])
      {
         a[j+1] = a[j];
         j = j-1;
      }
      a[j+1] = temp;
   }
   System.out.println("printing sorted elements ...");
   for(int i=0;i<10;i++)
   {
      System.out.println(a[i]);
   }
}
}
```

Output:

```
Printing sorted elements . . .
7
9
10
12
23
23
34
44
78
101
```

Python Program

```
a=[10, 9, 7, 101, 23, 44, 12, 78, 34, 23]
for k in range(1,10):
   temp = a[k]
   j = k-1
   while j>=0 and temp <=a[j]:
      a[j+1]=a[j]
      j = j-1
   a[j+1] = temp
print("printing sorted elements...")
for i in a:
   print(i)
```

Output:

```
printing sorted elements . . .
7
9
10
12
23
```

```
23
34
44
78
101
```

Merge sort

Merge sort is the algorithm which follows divide and conquer approach. Consider an array A of n number of elements. The algorithm processes the elements in 3 steps.

1. If A Contains 0 or 1 elements then it is already sorted, otherwise, Divide A into two sub-array of equal number of elements.
2. Conquer means sort the two sub-arrays recursively using the merge sort.
3. Combine the sub-arrays to form a single final sorted array maintaining the ordering of the array.

The main idea behind merge sort is that, the short list takes less time to be sorted.

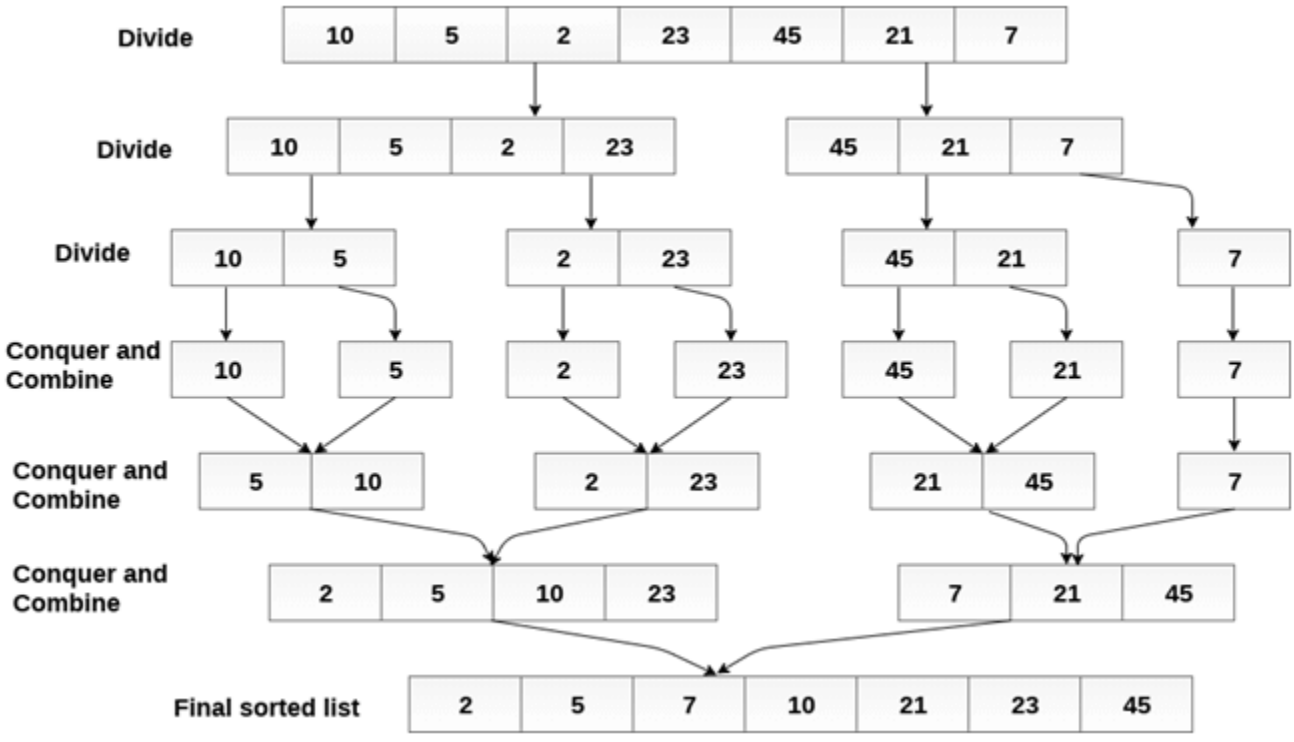
Complexity

Complexity	Best case	Average Case	Worst Case
Time Complexity	O(n log n)	O(n log n)	O(n log n)
Space Complexity			O(n)

Example :

Consider the following array of 7 elements. Sort the array by using merge sort.

A = {10, 5, 2, 23, 45, 21, 7}



Algorithm

- **Step 1**: [INITIALIZE] SET I = BEG, J = MID + 1, INDEX = 0
- **Step 2**: Repeat while (I <= MID) AND (J<=END)
  IF ARR[I] < ARR[J]
  SET TEMP[INDEX] = ARR[I]
  SET I = I + 1
  ELSE
  SET TEMP[INDEX] = ARR[J]
  SET J = J + 1
  [END OF IF]
  SET INDEX = INDEX + 1
  [END OF LOOP]
  Step 3: [Copy the remaining
  elements of right sub-array, if
  any]
  IF I > MID
  Repeat while J <= END
  SET TEMP[INDEX] = ARR[J]
  SET INDEX = INDEX + 1, SET J = J + 1
  [END OF LOOP]
  [Copy the remaining elements of
  left sub-array, if any]
  ELSE
  Repeat while I <= MID
  SET TEMP[INDEX] = ARR[I]
  SET INDEX = INDEX + 1, SET I = I + 1
  [END OF LOOP]
  [END OF IF]
- **Step 4**: [Copy the contents of TEMP back to ARR] SET K = 0
- **Step 5**: Repeat while K < INDEX
  SET ARR[K] = TEMP[K]
  SET K = K + 1
  [END OF LOOP]
- **Step 6**: Exit

MERGE_SORT(ARR, BEG, END)

- **Step 1**: IF BEG < END
  SET MID = (BEG + END)/2
  CALL MERGE_SORT (ARR, BEG, MID)
  CALL MERGE_SORT (ARR, MID + 1, END)
  MERGE (ARR, BEG, MID, END)
  [END OF IF]
- **Step 2**: END

C Program

```
#include<stdio.h>
void mergeSort(int[],int,int);
void merge(int[],int,int,int);
void main ()
{
   int a[10]= {10, 9, 7, 101, 23, 44, 12, 78, 34, 23};
   int i;
   mergeSort(a,0,9);
   printf("printing the sorted elements");
   for(i=0;i<10;i++)
   {
      printf("\n%d\n",a[i]);
   }
```

```
}
void mergeSort(int a[], int beg, int end)
{
   int mid;
   if(beg<end)
   {
      mid = (beg+end)/2;
      mergeSort(a,beg,mid);
      mergeSort(a,mid+1,end);
      merge(a,beg,mid,end);
   }
}
void merge(int a[], int beg, int mid, int end)
{
   int i=beg,j=mid+1,k,index = beg;
   int temp[10];
   while(i<=mid && j<=end)
   {
      if(a[i]<a[j])
      {
         temp[index] = a[i];
         i = i+1;
      }
      else
      {
         temp[index] = a[j];
         j = j+1;
      }
      index++;
   }
   if(i>mid)
   {
      while(j<=end)
      {
         temp[index] = a[j];
         index++;
         j++;
      }
   }
   else
   {
      while(i<=mid)
      {
         temp[index] = a[i];
         index++;
         i++;
      }
   }
   k = beg;
   while(k<index)
   {
      a[k]=temp[k];
      k++;
   }
}
```

Output:

```
printing the sorted elements

7
9
10
12
```

```
23
23
34
44
78
101
```

Java Program

```
public class MyMergeSort
{
void merge(int arr[], int beg, int mid, int end)
{

int l = mid - beg + 1;
int r = end - mid;

intLeftArray[] = new int [l];
intRightArray[] = new int [r];

for (int i=0; i<l; ++i)
LeftArray[i] = arr[beg + i];

for (int j=0; j<r; ++j)
RightArray[j] = arr[mid + 1+ j];


int i = 0, j = 0;
int k = beg;
while (i<l&&j<r)
{
if (LeftArray[i] <= RightArray[j])
{
arr[k] = LeftArray[i];
i++;
}
else
{
arr[k] = RightArray[j];
j++;
}
k++;
}
while (i<l)
{
arr[k] = LeftArray[i];
i++;
k++;
}

while (j<r)
{
arr[k] = RightArray[j];
j++;
k++;
}
}

void sort(int arr[], int beg, int end)
{
if (beg<end)
{
int mid = (beg+end)/2;
sort(arr, beg, mid);
```

```
sort(arr , mid+1, end);
merge(arr, beg, mid, end);
}
}
public static void main(String args[])
{
intarr[] = {90,23,101,45,65,23,67,89,34,23};
MyMergeSort ob = new MyMergeSort();
ob.sort(arr, 0, arr.length-1);

System.out.println("\nSorted array");
for(int i =0; i<arr.length;i++)
{
   System.out.println(arr[i]+"");
}
}
}
```

Output:

```
Sorted array
23
23
23
34
45
65
67
89
90
```

Quick Sort

Quick sort is the widely used sorting algorithm that makes n log n comparisons in average case for sorting of an array of n elements. This algorithm follows divide and conquer approach. The algorithm processes the array in the following way.

1. Set the first index of the array to left and loc variable. Set the last index of the array to right variable. i.e. left = 0, loc = 0, en d = n - 1, where n is the length of the array.
2. Start from the right of the array and scan the complete array from right to beginning comparing each element of the array with the element pointed by loc.

   Ensure that, a[loc] is less than a[right].

   1. If this is the case, then continue with the comparison until right becomes equal to the loc.
   2. If a[loc] > a[right], then swap the two values. And go to step 3.
   3. Set, loc = right

1. start from element pointed by left and compare each element in its way with the element pointed by the variable loc. Ensure that a[loc] > a[left]
   1. if this is the case, then continue with the comparison until loc becomes equal to left.
   2. [loc] < a[right], then swap the two values and go to step 2.
   3. Set, loc = left.

Complexity

Complexity	Best Case	Average Case	Worst Case
Time Complexity	O(n) for 3 way partition or O(n log n) simple partition	O(n log n)	$O(n^2)$
Space Complexity			O(log n)

Algorithm

PARTITION (ARR, BEG, END, LOC)

- **Step 1**: [INITIALIZE] SET LEFT = BEG, RIGHT = END, LOC = BEG, FLAG =
- **Step 2**: Repeat Steps 3 to 6 while FLAG =
- **Step 3**: Repeat while ARR[LOC] <=ARR[RIGHT]
  AND LOC != RIGHT
  SET RIGHT = RIGHT - 1
  [END OF LOOP]
- **Step 4**: IF LOC = RIGHT
  SET FLAG = 1
  ELSE IF ARR[LOC] > ARR[RIGHT]
  SWAP ARR[LOC] with ARR[RIGHT]
  SET LOC = RIGHT
  [END OF IF]
- **Step 5**: IF FLAG = 0
  Repeat while ARR[LOC] >= ARR[LEFT] AND LOC != LEFT
  SET LEFT = LEFT + 1
  [END OF LOOP]
- **Step 6**:IF LOC = LEFT
  SET FLAG = 1
  ELSE IF ARR[LOC] < ARR[LEFT]
  SWAP ARR[LOC] with ARR[LEFT]
  SET LOC = LEFT
  [END OF IF]
  [END OF IF]
- **Step 7**: [END OF LOOP]
- **Step 8**: END

QUICK_SORT (ARR, BEG, END)

- **Step 1**: IF (BEG < END)
  CALL PARTITION (ARR, BEG, END, LOC)
  CALL QUICKSORT(ARR, BEG, LOC - 1)
  CALL QUICKSORT(ARR, LOC + 1, END)
  [END OF IF]
- **Step 2**: END

C Program

```
1.  #include <stdio.h>
2.  int partition(int a[], int beg, int end);
3.  void quickSort(int a[], int beg, int end);
4.  void main()
5.  {
6.     int i;
7.     int arr[10]={90,23,101,45,65,23,67,89,34,23};
8.     quickSort(arr, 0, 9);
9.     printf("\n The sorted array is: \n");
10.    for(i=0;i<10;i++)
11.    printf(" %d\t", arr[i]);
```

```
}
int partition(int a[], int beg, int end)
{

   int left, right, temp, loc, flag;
   loc = left = beg;
   right = end;
   flag = 0;
   while(flag != 1)
   {
      while((a[loc] <= a[right]) && (loc!=right))
      right--;
      if(loc==right)
      flag =1;
      else if(a[loc]>a[right])
      {
         temp = a[loc];
         a[loc] = a[right];
         a[right] = temp;
         loc = right;
      }
      if(flag!=1)
      {
         while((a[loc] >= a[left]) && (loc!=left))
         left++;
         if(loc==left)
         flag =1;
         else if(a[loc] <a[left])
         {
            temp = a[loc];
            a[loc] = a[left];
            a[left] = temp;
            loc = left;
         }
      }
   }
   return loc;
}
void quickSort(int a[], int beg, int end)
{

   int loc;
   if(beg<end)
   {
      loc = partition(a, beg, end);
      quickSort(a, beg, loc-1);
      quickSort(a, loc+1, end);
   }
}
```

Output:

```
The sorted array is:
23
23
23
34
45
65
67
89
90
101
```

Java Program

```
public class QuickSort {
public static void main(String[] args) {
      int i;
      int[] arr={90,23,101,45,65,23,67,89,34,23};
      quickSort(arr, 0, 9);
      System.out.println("\n The sorted array is: \n");
      for(i=0;i<10;i++)
      System.out.println(arr[i]);
   }
   public static int partition(int a[], int beg, int end)
   {

      int left, right, temp, loc, flag;
      loc = left = beg;
      right = end;
      flag = 0;
      while(flag != 1)
      {
         while((a[loc] <= a[right]) && (loc!=right))
         right--;
         if(loc==right)
         flag =1;
         elseif(a[loc]>a[right])
         {
            temp = a[loc];
            a[loc] = a[right];
            a[right] = temp;
            loc = right;
         }
         if(flag!=1)
         {
            while((a[loc] >= a[left]) && (loc!=left))
            left++;
            if(loc==left)
            flag =1;
            elseif(a[loc] <a[left])
            {
               temp = a[loc];
               a[loc] = a[left];
               a[left] = temp;
               loc = left;
            }
         }
      }
      returnloc;
   }
   static void quickSort(int a[], int beg, int end)
   {

      int loc;
      if(beg<end)
      {
         loc = partition(a, beg, end);
         quickSort(a, beg, loc-1);
         quickSort(a, loc+1, end);
      }
   }
}
```

Output:

```
The sorted array is:
```

```
23
23
23
34
45
65
67
89
90
101
```

Radix Sort

Radix sort processes the elements the same way in which the names of the students are sorted according to their alphabetical order. There are 26 radix in that case due to the fact that, there are 26 alphabets in English. In the first pass, the names are grouped according to the ascending order of the first letter of names.

In the second pass, the names are grouped according to the ascending order of the second letter. The same process continues until we find the sorted list of names. The bucket are used to store the names produced in each pass. The number of passes depends upon the length of the name with the maximum letter.

In the case of integers, radix sort sorts the numbers according to their digits. The comparisons are made among the digits of the number from LSB to MSB. The number of passes depend upon the length of the number with the most number of digits.

Complexity

Complexity	Best Case	Average Case	Worst Case
Time Complexity	Ω(n+k)	θ(nk)	O(nk)
Space Complexity			O(n+k)

Example

Consider the array of length 6 given below. Sort the array by using Radix sort.

A = {10, 2, 901, 803, 1024}

Pass 1: (Sort the list according to the digits at 0's place)

10, 901, 2, 803, 1024.

Pass 2: (Sort the list according to the digits at 10's place)

02, 10, 901, 803, 1024

Pass 3: (Sort the list according to the digits at 100's place)

02, 10, 1024, 803, 901.

Pass 4: (Sort the list according to the digits at 1000's place)

02, 10, 803, 901, 1024

Therefore, the list generated in the step 4 is the sorted list, arranged from radix sort.

Algorithm

- **Step 1**:Find the largest number in ARR as LARGE
- **Step 2**: [INITIALIZE] SET NOP = Number of digits in LARGE
- **Step 3**: SET PASS =0
- **Step 4**: Repeat Step 5 while PASS <= NOP-1
- **Step 5**: SET I = 0 and INITIALIZE buckets
- **Step 6**:Repeat Steps 7 to 9 while I<n-1< li="""></n-1<>
- **Step 7**: SET DIGIT = digit at PASSth place in A[I]
- **Step 8**: Add A[I] to the bucket numbered DIGIT
- **Step 9**: INCREMENT bucket count for bucket numbered DIGIT
  [END OF LOOP]
- **Step 10**: Collect the numbers in the bucket
  [END OF LOOP]
- **Step 11**: END

C Program

```
#include <stdio.h>
int largest(int a[]);
void radix_sort(int a[]);
void main()
{
   int i;
   int a[10]={90,23,101,45,65,23,67,89,34,23};
   radix_sort(a);
   printf("\n The sorted array is: \n");
   for(i=0;i<10;i++)
      printf(" %d\t", a[i]);
}

int largest(int a[])
{
   int larger=a[0], i;
   for(i=1;i<10;i++)
   {
      if(a[i]>larger)
      larger = a[i];
   }
   return larger;
}
void radix_sort(int a[])
{
   int bucket[10][10], bucket_count[10];
   int i, j, k, remainder, NOP=0, divisor=1, larger, pass;
   larger = largest(a);
   while(larger>0)
   {
      NOP++;
      larger/=10;
   }
   for(pass=0;pass<NOP;pass++) // Initialize the buckets
   {
      for(i=0;i<10;i++)
      bucket_count[i]=0;
      for(i=0;i<10;i++)
      {
         // sort the numbers according to the digit at passth place
         remainder = (a[i]/divisor)%10;
```

```
            bucket[remainder][bucket_count[remainder]] = a[i];
            bucket_count[remainder] += 1;
        }
        // collect the numbers after PASS pass
        i=0;
        for(k=0;k<10;k++)
        {
            for(j=0;j<bucket_count[k];j++)
            {
                a[i] = bucket[k][j];
                i++;
            }
        }
        divisor *= 10;

}
}
```

Output:

```
The sorted array is:
23
23
23
34
45
65
67
89
90
101
```

Java Program

```
public class Radix_Sort {
public static void main(String[] args) {
        int i;
        Scanner sc = new Scanner(System.in);
        int[] a = {90,23,101,45,65,23,67,89,34,23};
        radix_sort(a);
        System.out.println("\n The sorted array is: \n");
        for(i=0;i<10;i++)
            System.out.println(a[i]);
    }

    static int largest(inta[])
    {
        int larger=a[0], i;
        for(i=1;i<10;i++)
        {
            if(a[i]>larger)
            larger = a[i];
        }
        returnlarger;
    }
    static void radix_sort(inta[])
    {
        int bucket[][]=newint[10][10];
        int bucket_count[]=newint[10];
        int i, j, k, remainder, NOP=0, divisor=1, larger, pass;
        larger = largest(a);
        while(larger>0)
        {
```

```
        NOP++;
        larger/=10;
      }
      for(pass=0;pass<NOP;pass++) // Initialize the buckets
      {
        for(i=0;i<10;i++)
        bucket_count[i]=0;
        for(i=0;i<10;i++)
        {
          // sort the numbers according to the digit at passth place
          remainder = (a[i]/divisor)%10;
          bucket[remainder][bucket_count[remainder]] = a[i];
          bucket_count[remainder] += 1;
        }
        // collect the numbers after PASS pass
        i=0;
        for(k=0;k<10;k++)
        {
          for(j=0;j<bucket_count[k];j++)
          {
            a[i] = bucket[k][j];
            i++;
          }
        }
        divisor *= 10;
      }
    }
}
```

Output:

```
The sorted array is:
23
23
23
34
45
65
67
89
90
101
```

Selection Sort

In selection sort, the smallest value among the unsorted elements of the array is selected in every pass and inserted to its appropriate position into the array.

First, find the smallest element of the array and place it on the first position. Then, find the second smallest element of the array and place it on the second position. The process continues until we get the sorted array.

The array with n elements is sorted by using n-1 pass of selection sort algorithm.

- In 1st pass, smallest element of the array is to be found along with its index **pos**. then, swap A[0] and A[pos]. Thus A[0] is sorted, we now have n -1 elements which are to be sorted.
- In 2nd pas, position pos of the smallest element present in the sub-array A[n-1] is found. Then, swap, A[1] and A[pos]. Thus A[0] and A[1] are sorted, we now left with n-2 unsorted elements.
- In n-1th pass, position pos of the smaller element between A[n-1] and A[n-2] is to be found. Then, swap, A[pos] and A[n-1].

Therefore, by following the above explained process, the elements A[0], A[1], A[2],...., A[n-1] are sorted.

Example

Consider the following array with 6 elements. Sort the elements of the array by using selection sort.

A = {10, 2, 3, 90, 43, 56}.

Pass	Pos	A[0]	A[1]	A[2]	A[3]	A[4]	A[5]
1	1	2	10	3	90	43	56
2	2	2	3	10	90	43	56
3	2	2	3	10	90	43	56
4	4	2	3	10	43	90	56
5	5	**2**	**3**	**10**	**43**	**56**	**90**

Sorted A = {2, 3, 10, 43, 56, 90}

Complexity

Complexity	Best Case	Average Case	Worst Case
Time	Ω(n)	θ(n^2)	o(n^2)
Space			o(1)

Algorithm

SELECTION SORT(ARR, N)

- **Step 1**: Repeat Steps 2 and 3 for K = 1 to N-1
- **Step 2**: CALL SMALLEST(ARR, K, N, POS)
- **Step 3**: SWAP A[K] with ARR[POS]
  [END OF LOOP]
- **Step 4**: EXIT

SMALLEST (ARR, K, N, POS)

- **Step 1**: [INITIALIZE] SET SMALL = ARR[K]
- **Step 2**: [INITIALIZE] SET POS = K
- **Step 3**: Repeat for J = K+1 to N -1
  IF SMALL > ARR[J]
  SET SMALL = ARR[J]
  SET POS = J
  [END OF IF]
  [END OF LOOP]
- **Step 4**: RETURN POS

```
#include<stdio.h>
int smallest(int[],int,int);
void main ()
{
   int a[10] = {10, 9, 7, 101, 23, 44, 12, 78, 34, 23};
   int i,j,k,pos,temp;
   for(i=0;i<10;i++)
   {
      pos = smallest(a,10,i);
      temp = a[i];
```

```
            a[i]=a[pos];
            a[pos] = temp;
        }
        printf("\nprinting sorted elements...\n");
        for(i=0;i<10;i++)
        {
            printf("%d\n",a[i]);
        }
    }
    int smallest(int a[], int n, int i)
    {
        int small,pos,j;
        small = a[i];
        pos = i;
        for(j=i+1;j<10;j++)
        {
            if(a[j]<small)
            {
                small = a[j];
                pos=j;
            }
        }
        return pos;
    }
```

Output:

```
printing sorted elements...
7
9
10
12
23
23
34
44
78
101
```

Java

```
public class SelectionSort {
public static void main(String[] args) {
    int[] a = {10, 9, 7, 101, 23, 44, 12, 78, 34, 23};
    int i,j,k,pos,temp;
    for(i=0;i<10;i++)
    {
        pos = smallest(a,10,i);
        temp = a[i];
        a[i]=a[pos];
        a[pos] = temp;
    }
    System.out.println("\nprinting sorted elements...\n");
    for(i=0;i<10;i++)
    {
        System.out.println(a[i]);
    }
}
public static int smallest(int a[], int n, int i)
{
    int small,pos,j;
    small = a[i];
    pos = i;
    for(j=i+1;j<10;j++)
    {
```

```
      if(a[j]<small)
      {
         small = a[j];
         pos=j;
      }
   }
   return pos;
}
}
```

Output:

```
printing sorted elements...
7
9
10
12
23
23
34
44
78
101
```

Python Program

```
def smallest(a,i):
   small = a[i]
   pos=i
   for j in range(i+1,10):
      if a[j] < small:
         small = a[j]
         pos = j
   return pos

a=[10, 9, 7, 101, 23, 44, 12, 78, 34, 23]
for i in range(0,10):
   pos = smallest(a,i)
   temp = a[i]
   a[i]=a[pos]
   a[pos]=temp
print("printing sorted elements...")
for i in a:
   print(i)
```

Output:

```
printing sorted elements...
7
9
10
12
23
23
34
44
78
101
```