

# Unit 6-Sorting and Searching

## Searching

Searching is the process of finding some particular element in the list. If the element is present in the list, then the process is called successful and the process returns the location of that element, otherwise the search is called unsuccessful.

There are two popular search methods that are widely used in order to search some item into the list. However, choice of the algorithm depends upon the arrangement of the list.

- Linear Search
- Binary Search

## Linear Search

Linear search is the simplest search algorithm and often called sequential search. In this type of searching, we simply traverse the list completely and match each element of the list with the item whose location is to be found. If the match found then location of the item is returned otherwise the algorithm return NULL.

Linear search is mostly used to search an unordered list in which the items are not sorted. The algorithm of linear search is given as follows.

## Algorithm

- LINEAR\_SEARCH(A, N, VAL)
- **Step 1:** SET POS = 0
- **Step 2:** SET I = 0
- **Step 3:** Repeat Step 4 while I < N
- **Step 4:** IF A[I] = VAL  
SET POS = I+1  
PRINT POS  
Go to Step 6  
[END OF IF]  
SET I = I + 1  
[END OF LOOP]
- **Step 5:** IF POS = 0  
PRINT " VALUE IS NOT PRESENTIN THE ARRAY "  
[END OF IF]

- **Step 6:** EXIT

## Complexity of algorithm

Complexity	Best Case	Average Case	Worst Case
Time	O(1)	O(n)	O(n)
Space			O(1)

```

1. #include<stdio.h>
2. void main ()
3. {
4.     int a[10] = {10, 23, 40, 1, 2, 0, 14, 13, 50, 9};
5.     int item, i, flag=0;
6.     printf("\nEnter Item which is to be searched\n");
7.     scanf("%d",&item);
8.     for (i = 0; i < 10; i++)
9.     {
10.         if(a[i] == item)
11.         {
12.             flag = i+1;
13.             break;
14.         }
15.         else
16.             flag = 0;
17.     }
18.     if(flag != 0)
19.     {
20.         printf("\nItem found at location %d\n",flag);
21.     }
22.     else
23.     {
24.         printf("\nItem not found\n");
25.     }

```

## Binary Search

Binary search is the search technique which works efficiently on the sorted lists. Hence, in order to search an element into some list by using binary search technique, we must ensure that the list is sorted.

Binary search follows divide and conquer approach in which, the list is divided into two halves and the item is compared with the middle element of the list. If the match is found then, the location of middle element is returned otherwise, we search into either of the halves depending upon the result produced through the match.

Binary search algorithm is given below.

### **BINARY\_SEARCH(A, lower\_bound, upper\_bound, VAL)**

- **Step 1:** [INITIALIZE] SET BEG = lower\_bound  
END = upper\_bound, POS = - 1
- **Step 2:** Repeat Steps 3 and 4 while BEG <=END
- **Step 3:** SET MID = (BEG + END)/2
- **Step 4:** IF A[MID] = VAL  
SET POS = MID+1  
PRINT POS  
Go to Step 6  
ELSE IF A[MID] > VAL  
SET END = MID - 1  
ELSE IF A[MID]<VAL  
SET BEG = MID + 1  
[END OF IF]  
[END OF LOOP]
- **Step 5:** IF POS = -1  
PRINT "VALUE IS NOT PRESENT IN THE ARRAY"  
[END OF IF]
- **Step 6:** EXIT

## Complexity

SN	Performance	Complexity
1	Worst case	$O(\log n)$
2	Best case	$O(1)$
3	Average Case	$O(\log n)$
4	Worst case space complexity	$O(1)$

## Example

Let us consider an array  $arr = \{1, 5, 7, 8, 13, 19, 20, 23, 29\}$ . Find the location of the item 23 in the array.

### In 1<sup>st</sup> step :

1.  $BEG = 0$
2.  $END = 8$
3.  $MID = 4$
4.  $a[mid] = a[4] = 13 < 23$ , therefore

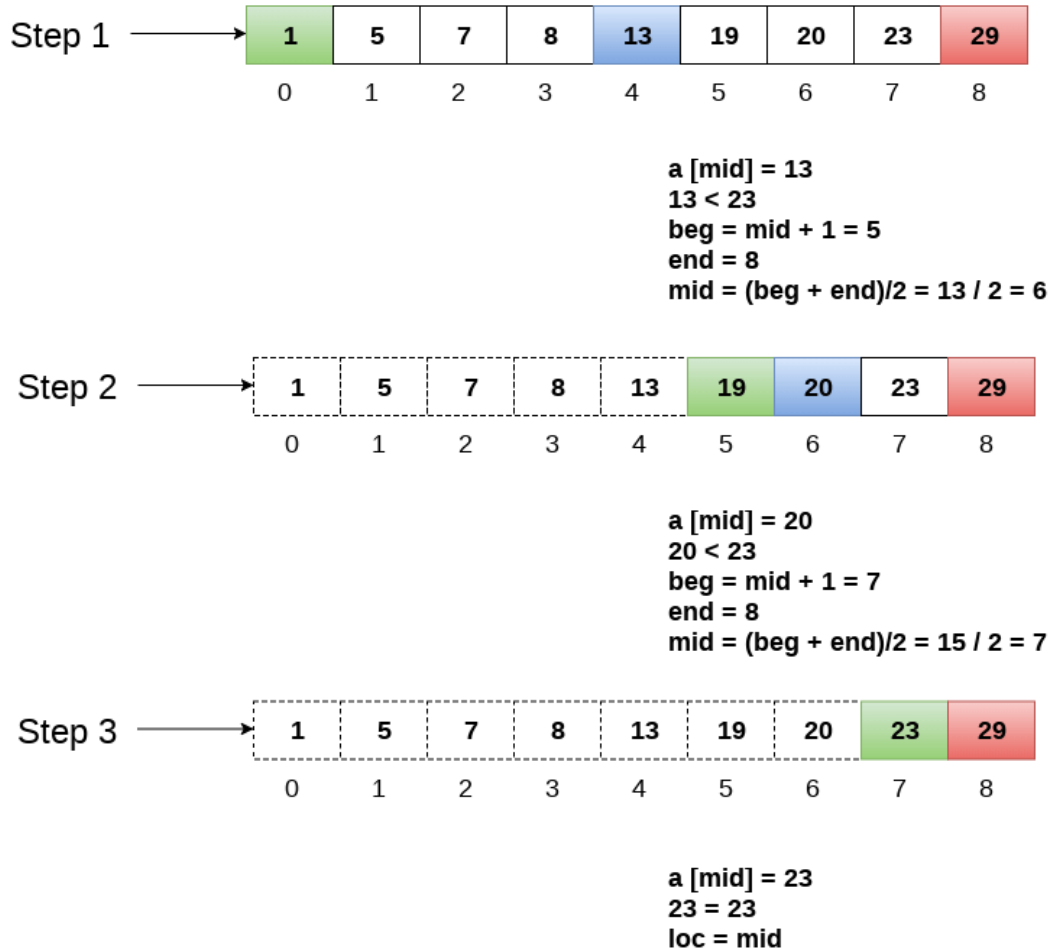
### in Second step:

1.  $Beg = mid + 1 = 5$
2.  $End = 8$
3.  $mid = 13/2 = 6$
4.  $a[mid] = a[6] = 20 < 23$ , therefore;

### in third step:

1.  $beg = mid + 1 = 7$
2.  $End = 8$
3.  $mid = 15/2 = 7$
4.  $a[mid] = a[7]$
5.  $a[7] = 23 = \text{item};$
6. therefore, set location = mid;
7. The location of the item will be 7.

Item to be searched = 23



Return location 7

## Binary Search Program using Recursion

### C program

1. `#include<stdio.h>`
2. `int binarySearch(int[], int, int, int);`
3. `void main ()`
4. `{`
5. `int arr[10] = {16, 19, 20, 23, 45, 56, 78, 90, 96, 100};`
6. `int item, location=-1;`
7. `printf("Enter the item which you want to search ");`
8. `scanf("%d",&item);`
9. `location = binarySearch(arr, 0, 9, item);`
10. `if(location != -1)`

```

11. {
12.     printf("Item found at location %d",location);
13. }
14. else
15. {
16.     printf("Item not found");
17. }
18.}
19.int binarySearch(int a[], int beg, int end, int item)
20.{
21.    int mid;
22.    if(end >= beg)
23.    {
24.        mid = (beg + end)/2;
25.        if(a[mid] == item)
26.        {
27.            return mid+1;
28.        }
29.        else if(a[mid] < item)
30.        {
31.            return binarySearch(a,mid+1,end,item);
32.        }
33.        else
34.        {
35.            return binarySearch(a,beg,mid-1,item);
36.        }
37.
38.    }
39.    return -1;
40.}

```

## Sorting Algorithms

Sorting is the process of arranging the elements of an array so that they can be placed either in ascending or descending order. For example, consider an array  $A = \{A_1, A_2, A_3, A_4, \dots, A_n\}$ , the array is called to be in ascending order if element of  $A$  are arranged like  $A_1 < A_2 < A_3 < A_4 < A_5 < \dots < A_n$ .

**Consider an array;**

int A[10] = { 5, 4, 10, 2, 30, 45, 34, 14, 18, 9 }

**The Array sorted in ascending order will be given as;**

A[] = { 2, 4, 5, 9, 10, 14, 18, 30, 34, 45 }

There are many techniques by using which, sorting can be performed.

## 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 - 1
- **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

Scenario22.....gt6q13vcxq2 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

```

1. #include<stdio.h>
2. void main ()
3. {
4.     int i, j,temp;
5.     int a[10] = { 10, 9, 7, 101, 23, 44, 12, 78, 34, 23};
6.     for(i = 0; i<10; i++)
7.     {
8.         for(j = i+1; j<10; j++)
9.         {
10.            if(a[i] > a[j])
11.            {
12.                temp = a[i];
13.                a[i] = a[j];
14.                a[j] = temp;
15.            }
16.        }
17.    }
18.    printf("Printing Sorted Element List ...\n");
19.    for(i = 0; i<10; i++)
20.    {
21.        printf("%d\n",a[i]);
22.    }
23.}

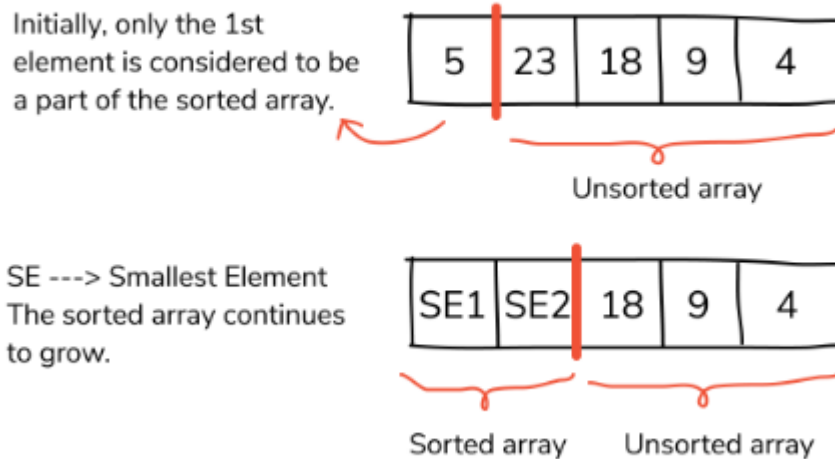
```

## Insertion Sort

**it compares the current element with the elements on the left-hand side (sorted array).** If the current element is greater than all the elements on its left hand side, then it leaves the



element in its place and moves on to the next element. Else it finds its correct position and moves it to that position by shifting all the elements, which are larger than the current element, in the sorted array to one position ahead.



### Insertion Sort Example & Working

Let us consider an array with 5 elements,  $A = [7, 1, 23, 4, 19]$ . **Below is how insertion sort works for this array.**

Given unsorted array →

7	1	23	4	19
---	---	----	---	----

Consider the first element as sorted as there are no other elements on its left-hand side


7	1	23	4	19
---	---	----	---	----

**Look at the next unsorted element (1) & compare it with the sorted elements (7)**

$1 < 7$  ?

Yes, so swap

7	1	23	4	19
---	---	----	---	----




$23 < 7$  ? ---> No

$23 < 1$  ? ---> No

So no swaps in this iteration

1	7	23	4	19
---	---	----	---	----




$4 < 23$  ? ---> Yes, so check the next number

$4 < 7$  ? ---> Yes, so check the next number

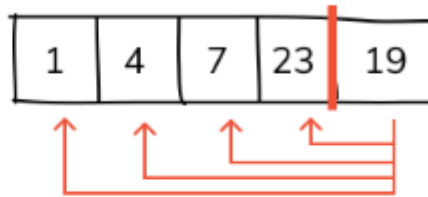
$4 < 1$  ? ---> No

1	7	23	4	19
---	---	----	---	----

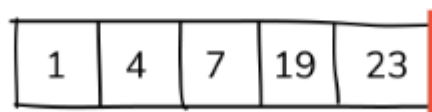


So the correct position of 4 is the current position of 7. **So shift all the elements from 7 to one position ahead.**

19 < 23 ? ---> Yes, so check  
the next number  
19 < 7 ? ---> No



So the correct position of 19 is the current position of 23. **So shift the element 23 one position ahead.**



The entire array is now sorted.

### Another Example:

12, 11, 13, 5, 6

Let us loop for  $i = 1$  (second element of the array) to 4 (last element of the array)

$i = 1$ . Since 11 is smaller than 12, move 12 and insert 11 before 12

11, 12, 13, 5, 6

$i = 2$ . 13 will remain at its position as all elements in  $A[0..i-1]$  are smaller than 13

11, 12, 13, 5, 6

$i = 3$ . 5 will move to the beginning and all other elements from 11 to 13 will move one position ahead of their current position.

5, 11, 12, 13, 6

$i = 4$ . 6 will move to position after 5, and elements from 11 to 13 will move one position ahead of their current position.

5, 6, 11, 12, 13

## 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
- Best case time-complexity:  **$O(n)$** . This occurs when the given array is already sorted.

```

1. {
2.     temp = a[k];
3.     j = k-1;
4.     while(j >= 0 && temp <= a[j])
5.     {
6.         a[j+1] = a[j];
7.         j = j-1;
8.     }
9.     a[j+1] = temp;
10. }
11. for(i=0; i<10; i++)
12. {
13.     printf("\n%d\n", a[i]);
14. }
15. }

```

## 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 pass, 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  $n-1$ th 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.

Initial unsorted array of elements

14	16	3	6	50
----	----	---	---	----

Take the first element and compare it with every other element. **If you find any element to be smaller than the 1st element, then swap both of them.**

{ 14 > 16 -----> No  
14 > 3 -----> Yes

14	16	3	6	50
----	----	---	---	----

↑

{ 3 > 6 -----> No  
3 > 50 -----> No

3	16	14	6	50
---	----	----	---	----

↑

3 is the smallest and is a part of sorted array

3	16	14	6	50
---	----	----	---	----

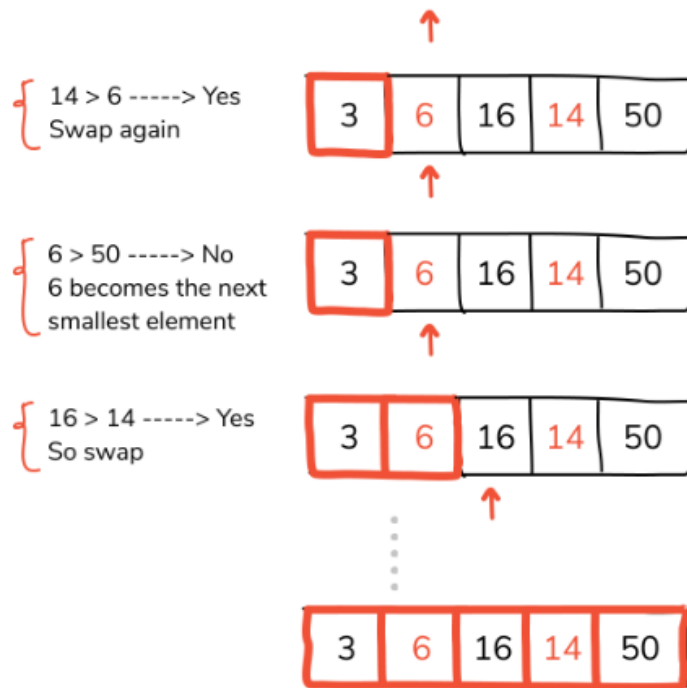
{ 16 > 14 -----> Yes.  
So swap

3	16	14	6	50
---	----	----	---	----

↑

{ 14 > 6 -----> Yes  
Swap again

3	14	16	6	50
---	----	----	---	----



## 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

```
1. #include<stdio.h>
2. int smallest(int[],int,int);
3. void main ()
4. {
5.     int a[10] = {10, 9, 7, 101, 23, 44, 12, 78, 34, 23};
6.     int i,j,k,pos,temp;
7.     for(i=0;i<10;i++)
8.     {
9.         pos = smallest(a,10,i);
10.        temp = a[i];
11.        a[i]=a[pos];
12.        a[pos] = temp;
13.    }
14.    printf("\nprinting sorted elements...\n");
15.    for(i=0;i<10;i++)
16.    {
17.        printf("%d\n",a[i]);
18.    }
19.}
20.int smallest(int a[], int n, int i)
21.{
22.    int small,pos,j;
23.    small = a[i];
24.    pos = i;
25.    for(j=i+1;j<10;j++)
26.    {
27.        if(a[j]<small)
28.        {
29.            small = a[j];
30.            pos=j;
31.        }
32.    }
33.    return pos;
34.}
```

## Quick Sort Algorithm in Brief

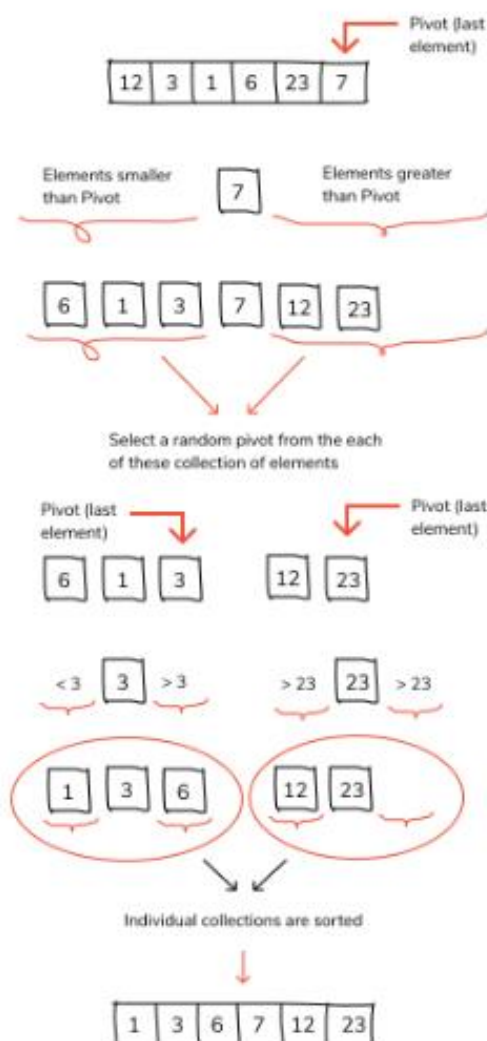
Quicksort is the quickest and one of the most efficient sorting algorithm. Similar to [Merge sort](#), Quick sort follows Divide and conquer algorithm to sort the given array.

The quicksort algorithm is a sorting algorithm that sorts an array by choosing a pivot element, and partition the array around the pivot such that.

**Elements smaller than the pivot are moved to the left of pivot, and elements larger than the pivot are moved to the right of pivot.**

It continues to choose a pivot element and break down the array into single-element array, before combing them back together to form a single sorted array.

**Elements smaller than the pivot are moved to the left of pivot, and elements**





**Note:** Don't get confused as to how to determine a pivot. Simply **choose the first/last/ideally the middle element as pivot without any confusion.**

## 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  $ARR[LOC] \leq ARR[RIGHT]$   
AND  $LOC \neq RIGHT$   
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  $ARR[LOC] \geq ARR[LEFT]$  AND  $LOC \neq LEFT$   
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

Or

Algorithm:➔

- **Step 1** - Consider the first element of the list as **pivot** (i.e., Element at first position in the list).
- **Step 2** - Define two variables i and j. Set i and j to first and last elements of the list respectively.
- **Step 3** - Increment i until list[i] > pivot then stop.
- **Step 4** - Decrement j until list[j] < pivot then stop.
- **Step 5** - If i < j then exchange list[i] and list[j].
- **Step 6** - Repeat steps 3,4 & 5 until i > j.
- **Step 7** - Exchange the pivot element with list[j] element.

## 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]);
12.}
13.int partition(int a[], int beg, int end)
14.{
15.
16.    int left, right, temp, loc, flag;
```

```
17. loc = left = beg;
18. right = end;
19. flag = 0;
20. while(flag != 1)
21. {
22.     while((a[loc] <= a[right]) && (loc!=right))
23.         right--;
24.     if(loc==right)
25.         flag = 1;
26.     else if(a[loc]>a[right])
27.     {
28.         temp = a[loc];
29.         a[loc] = a[right];
30.         a[right] = temp;
31.         loc = right;
32.     }
33.     if(flag!=1)
34.     {
35.         while((a[loc] >= a[left]) && (loc!=left))
36.             left++;
37.         if(loc==left)
38.             flag = 1;
39.         else if(a[loc] <a[left])
40.         {
41.             temp = a[loc];
42.             a[loc] = a[left];
43.             a[left] = temp;
44.             loc = left;
45.         }
46.     }
47. }
48. return loc;
49.}
50. void quickSort(int a[], int beg, int end)
51. {
52.
53.     int loc;
```

```
54.  if(beg<end)
55.  {
56.      loc = partition(a, beg, end);
57.      quickSort(a, beg, loc-1);
58.      quickSort(a, loc+1, end);
59.  }
60.}
```

## Merge Sort: ➔ 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.

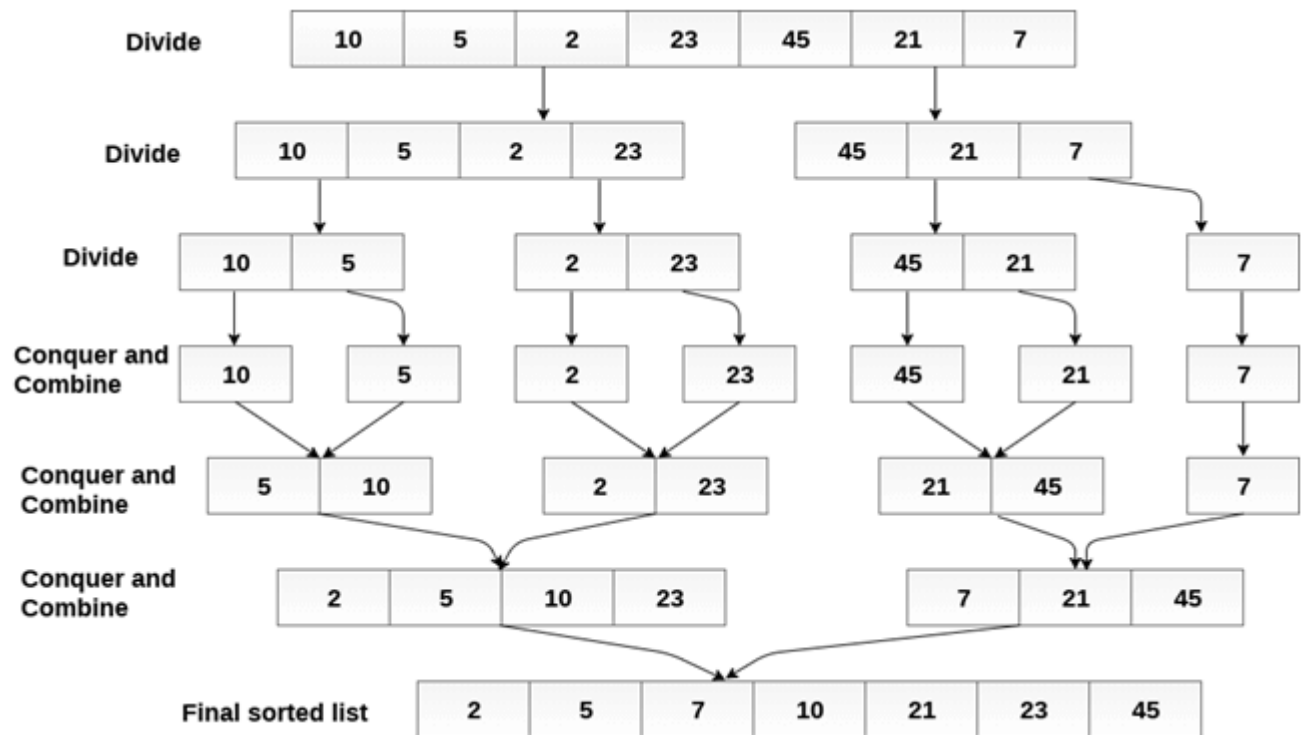
The main idea behind merge sort is that, the short list takes less time to be sorted.

### Example :

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

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

**A simple visualization of this is shown below.**



## Complexity

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

## Algorithm

- **Step 1:** [INITIALIZE] SET  $I = \text{BEG}$ ,  $J = \text{MID} + 1$ ,  $\text{INDEX} = 0$
- **Step 2:** Repeat while  $(I \leq \text{MID})$  AND  $(J \leq \text{END})$ 
  - IF  $\text{ARR}[I] < \text{ARR}[J]$
  - SET  $\text{TEMP}[\text{INDEX}] = \text{ARR}[I]$
  - SET  $I = I + 1$
  - ELSE
  - SET  $\text{TEMP}[\text{INDEX}] = \text{ARR}[J]$

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  $I > \text{MID}$

Repeat while  $J \leq \text{END}$

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  $I \leq \text{MID}$

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  $K < \text{INDEX}$   
SET ARR[K] = TEMP[K]  
SET  $K = K + 1$   
[END OF LOOP]
- **Step 6:** Exit

### **MERGE\_SORT(ARR, BEG, END)**

- **Step 1:** IF  $\text{BEG} < \text{END}$   
SET  $\text{MID} = (\text{BEG} + \text{END})/2$   
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

```
1. #include<stdio.h>
2. void mergeSort(int[],int,int);
3. void merge(int[],int,int,int);
4. void main ()
5. {
6.     int a[10]= {10, 9, 7, 101, 23, 44, 12, 78, 34, 23};
7.     int i;
8.     mergeSort(a,0,9);
9.     printf("printing the sorted elements");
10.    for(i=0;i<10;i++)
11.    {
12.        printf("\n%d\n",a[i]);
13.    }
14.
15. }
16. void mergeSort(int a[], int beg, int end)
17. {
18.     int mid;
19.     if(beg<end)
20.     {
21.         mid = (beg+end)/2;
22.         mergeSort(a,beg,mid);
23.         mergeSort(a,mid+1,end);
24.         merge(a,beg,mid,end);
25.     }
26. }
27. void merge(int a[], int beg, int mid, int end)
28. {
29.     int i=beg,j=mid+1,k,index = beg;
30.     int temp[10];
31.     while(i<=mid && j<=end)
32.     {
33.         if(a[i]<a[j])
34.         {
35.             temp[index] = a[i];
36.             i = i+1;
```

```
37.     }
38.     else
39.     {
40.         temp[index] = a[j];
41.         j = j+1;
42.     }
43.     index++;
44. }
45. if(i>mid)
46. {
47.     while(j<=end)
48.     {
49.         temp[index] = a[j];
50.         index++;
51.         j++;
52.     }
53. }
54. else
55. {
56.     while(i<=mid)
57.     {
58.         temp[index] = a[i];
59.         index++;
60.         i++;
61.     }
62. }
63. k = beg;
64. while(k<index)
65. {
66.     a[k]=temp[k];
67.     k++;
68. }
69. }
```



## Heap Sort →

Heap is a special kind of binary tree in which elements are stored in a hierarchical manner. Heap has some additional rules it must always have a heap structure, where all the levels of the binary tree are filled up from left to right. Second, it must either be ordered as a max heap or a min-heap.

- Max heap Parent node is greater than or equal to the value of its children node
- Min heap Parent node is less than or equal to the value of its children node

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.

### **How Heap Sort Works?**

Heap sort algorithm is divided into two basic parts:

- Creating a Heap of the unsorted list/array.
- Then a sorted array is created by repeatedly removing the largest/smallest element from the heap, and inserting it into the array. The heap is reconstructed after each removal.

- Heap sort is a comparison based sorting algorithm.
- It is a special tree-based data structure.
- Heap sort is similar to selection sort. The only difference is, it finds largest element and places the it at the end.
- This sort is not a stable sort. It requires a constant space for sorting a list.
- It is very fast and widely used for sorting.

## Complexity

Complexity	Best Case	Average Case	Worst case
Time Complexity	$\Omega(n \log (n))$	$\theta(n \log (n))$	$O(n \log (n))$
Space Complexity			$O(1)$

## Algorithm

### Step by Step Process

The Heap sort algorithm to arrange a list of elements in ascending order is performed using following steps...

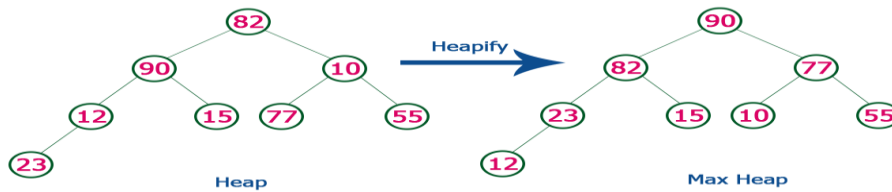
- **Step 1** - Construct a **Binary Tree** with given list of Elements.
- **Step 2** - Transform the Binary Tree into **Min Heap**.
- **Step 3** - Delete the root element from Min Heap using **Heapify** method.
- **Step 4** - Put the deleted element into the Sorted list.
- **Step 5** - Repeat the same until Min Heap becomes empty.
- **Step 6** - Display the sorted list.

## Example

Consider the following list of unsorted numbers which are to be sort using Heap Sort

**82, 90, 10, 12, 15, 77, 55, 23**

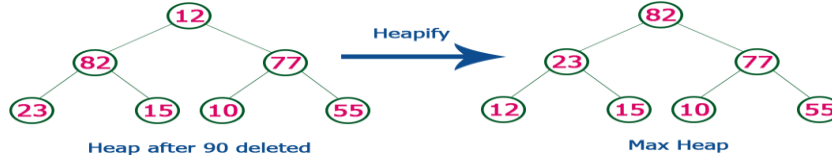
**Step 1 -** Construct a Heap with given list of unsorted numbers and convert to Max Heap



list of numbers after heap converted to Max Heap

**90, 82, 77, 23, 15, 10, 55, 12**

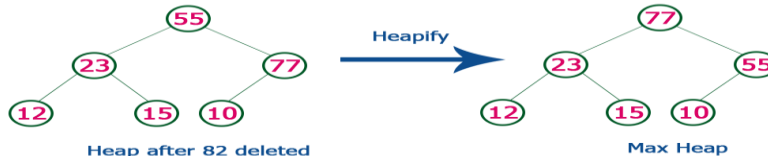
**Step 2 -** Delete root (**90**) from the Max Heap. To delete root node it needs to be swapped with last node (**12**). After delete tree needs to be heapify to make it Max Heap.



list of numbers after swapping 90 with 12.

**12, 82, 77, 23, 15, 10, 55, 90**

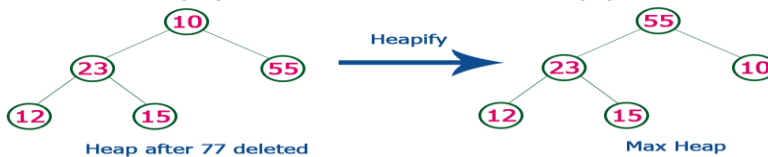
**Step 3 -** Delete root (**82**) from the Max Heap. To delete root node it needs to be swapped with last node (**55**). After delete tree needs to be heapify to make it Max Heap.



list of numbers after swapping 82 with 55.

**12, 55, 77, 23, 15, 10, 82, 90**

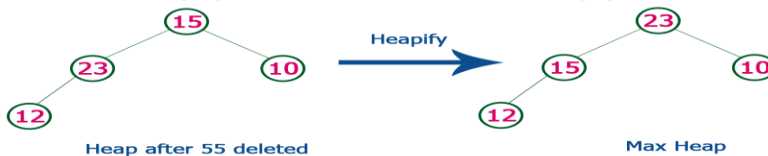
**Step 4 -** Delete root (**77**) from the Max Heap. To delete root node it needs to be swapped with last node (**10**). After delete tree needs to be heapify to make it Max Heap.



list of numbers after swapping 77 with 10.

**12, 55, 10, 23, 15, 77, 82, 90**

**Step 5 -** Delete root (**55**) from the Max Heap. To delete root node it needs to be swapped with last node (**15**). After delete tree needs to be heapify to make it Max Heap.



list of numbers after swapping 55 with 15.

**12, 15, 10, 23, 55, 77, 82, 90**

**Step 6 -** Delete root (**23**) from the Max Heap. To delete root node it needs to be swapped with last node (**12**). After delete tree needs to be heapify to make it Max Heap.



list of numbers after swapping 23 with 12.

**12, 15, 10, 23, 55, 77, 82, 90**

**Step 7 -** Delete root (**15**) from the Max Heap. To delete root node it needs to be swapped with last node (**10**). After delete tree needs to be heapify to make it Max Heap.



list of numbers after Deleting 15, 12 & 10 from the Max Heap.

**10, 12, 15, 23, 55, 77, 82, 90**

Whenever Max Heap becomes Empty, the list get sorted in Ascending order

## C Program

```
1. #include<stdio.h>
2. int temp;
3.
4. void heapify(int arr[], int size, int i)
5. {
6.     int largest = i;
7.     int left = 2*i + 1;
8.     int right = 2*i + 2;
9.
10.    if (left < size && arr[left] > arr[largest])
11.        largest = left;
12.
13.    if (right < size && arr[right] > arr[largest])
14.        largest = right;
15.
16.    if (largest != i)
17.    {
18.        temp = arr[i];
19.        arr[i] = arr[largest];
20.        arr[largest] = temp;
21.        heapify(arr, size, largest);
22.    }
23. }
24.
25. void heapSort(int arr[], int size)
26. {
27.     int i;
28.     for (i = size / 2 - 1; i >= 0; i--)
29.         heapify(arr, size, i);
30.     for (i = size - 1; i >= 0; i--)
31.     {
32.         temp = arr[0];
33.         arr[0] = arr[i];
34.         arr[i] = temp;
35.         heapify(arr, i, 0);
36.     }
```

```

37. }
38.
39. void main()
40. {
41. int arr[] = {1, 10, 2, 3, 4, 1, 2, 100,23, 2};
42. int i;
43. int size = sizeof(arr)/sizeof(arr[0]);
44.
45. heapSort(arr, size);
46.
47. printf("printing sorted elements\n");
48. for (i=0; i<size; ++i)
49. printf("%d\n",arr[i]);
50. }

```

Sorting Method	Time Complexity Worst Case	Time Complexity Average Case	Time Complexity Best Case	Space Complexity
Bubble Sort	$n(n-1)/2 = O(n^2)$	$n(n-1)/2 = O(n^2)$	$n(n-1)/2 = O(n^2)$	$O(1)$
Insertion Sort	$n(n-1)/2 = O(n^2)$	$n(n-1)/4 = O(n^2)$	$O(n)$	$O(1)$
Selection Sort	$n(n-1)/2 = O(n^2)$	$n(n-1)/2 = O(n^2)$	$n(n-1)/2 = O(n^2)$	$O(1)$
Quick Sort	$n(n+3)/2 = O(n^2)$	$O(n \log n)$	$O(n \log n)$	$O(N \log N)$
Heap Sort	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	$O(1)$
Merge Sort	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	$O(N)$