**SEARCHING AND SORTING**

**What is Searching in Data Structure?**

**The process of finding the desired information from the set of items stored in the form of elements in the computer memory is referred to as ‘searching in data structure’. These sets of items are in various forms, such as an array, tree, graph, or linked list.**

**There are two searching techniques-**

1. **Linear Search or sequential search**
2. **Binary Search**

### **Linear Search**

### **It is the most simple search algorithm in data structure and checks each item in the set of elements until it matches the search element til the end of data collection. When data is unsorted, a linear search algorithm is preferred.**

**Binary Search**

**In computer science, binary search, also known as half-interval search , is a search algorithm that finds the position of a target value within a sorted array. Binary search compares the target value to the middle element of the array.**

**Difference between sequential search and binary search**

| **Sequential Search** | **Binary Search** |
| --- | --- |
| **Time complexity is O(n)** | **Time complexity is O(log n)** |
| **Finds the key present at first position.** | **Finds the key present at center position.** |
| **Sequence of elements in the container does not affect.** | **The elements must be sorted in the container** |
| **Arrays and linked lists can be used to implement this** | **It cannot be implemented directly into the linked list. We need to change the basic rules of the list to implement this** |
| **Algorithm is iterative in nature** | **Algorithm technique is Divide and Conquer.** |
| **Algorithm is easy to implement, and requires less amount of code.** | **Algorithm is slightly complex. It takes more amount of code to implement.** |

**Sorting**

**The arrangement of data in a preferred order is called sorting in the data structure. Sorting is used to put elements in certain order. By sorting data, it is easier to search through it quickly and easily. The simplest example of sorting is a dictionary.**

# **Insertion Sort**

**Insertion sort works similar to the sorting of playing cards in hands. It is assumed that the first card is already sorted in the card game, and then we select an unsorted card. If the selected unsorted card is greater than the first card, it will be placed at the right side; otherwise, it will be placed at the left side. Similarly, all unsorted cards are taken and put in their exact place.**

**Insertion sort has various advantages such as -Competitive questions on Structures**

* **Simple implementation**
* **Efficient for small data sets**
* **Adaptive.**

## Algorithm

**The simple steps of achieving the insertion sort are listed as follows -**

**Step 1 - If the element is the first element, assume that it is already sorted. Return 1.**

**Step2 - Pick the next element, and store it separately in a key.**

**Step3 - Now, compare the key with all elements in the sorted array.**

**Step 4 - If the element in the sorted array is smaller than the current element, then move to the next element. Else, shift greater elements in the array towards the right.**

**Step 5 - Insert the value.**

**Step 6 - Repeat until the array is sorted.**

## Working of Insertion sort Algorithm

**Now, let's see the working of the insertion sort Algorithm.**

**To understand the working of the insertion sort algorithm, let's take an unsorted array. It will be easier to understand the insertion sort via an example.**

**Let the elements of array are -**

**Insertion Sort Algorithm**

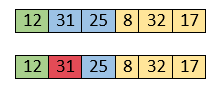
**Initially, the first two elements are compared in insertion sort.**

**Insertion Sort Algorithm**

**Here, 31 is greater than 12. That means both elements are already in ascending order. So, for now, 12 is stored in a sorted sub-array.**

**Insertion Sort Algorithm**

**Now, move to the next two elements and compare them.**

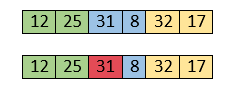
****

**Here, 25 is smaller than 31. So, 31 is not at correct position. Now, swap 31 with 25. Along with swapping, insertion sort will also check it with all elements in the sorted array.**

**For now, the sorted array has only one element, i.e. 12. So, 25 is greater than 12. Hence, the sorted array remains sorted after swapping.**

**Insertion Sort Algorithm**

**Now, two elements in the sorted array are 12 and 25. Move forward to the next elements that are 31 and 8.**

****

**Both 31 and 8 are not sorted. So, swap them.**

**Insertion Sort Algorithm**

**After swapping, elements 25 and 8 are unsorted.**

**Insertion Sort Algorithm**

**So, swap them.**

**Insertion Sort Algorithm**

**Now, elements 12 and 8 are unsorted.**

**Insertion Sort Algorithm**

**So, swap them too.**

**Insertion Sort Algorithm**

**Now, the sorted array has three items that are 8, 12 and 25. Move to the next items that are 31 and 32.**

**Insertion Sort Algorithm**

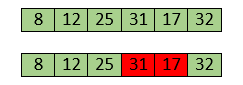
**Hence, they are already sorted. Now, the sorted array includes 8, 12, 25 and 31.**

**Insertion Sort Algorithm**

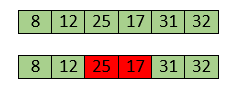
**Move to the next elements that are 32 and 17.**

**Insertion Sort Algorithm**

**17 is smaller than 32. So, swap them.**

****

**Swapping makes 31 and 17 unsorted. So, swap them too.**

****

**Now, swapping makes 25 and 17 unsorted. So, perform swapping again.**

**Insertion Sort Algorithm**

**Now, the array is completely sorted.**

## Insertion sort complexity

**Now, let's see the time complexity of insertion sort in best case, average case, and in worst case. We will also see the space complexity of insertion sort.**

### **Time Complexity**

|  |  |
| --- | --- |
| **Case** | **Time Complexity** |
| **Best Case** | **O(n)** |
| **Average Case** | **O(n2)** |
| **Worst Case** | **O(n2)** |

* **Best Case Complexity - It occurs when there is no sorting required, i.e. the array is already sorted. The best-case time complexity of insertion sort is O(n).**
* **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 insertion sort is O(n2).**
* **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 insertion sort is O(n2).**

|  |  |
| --- | --- |
| **Space Complexity** | **O(1)** |
| **Stable** | **YES** |

* **The space complexity of insertion sort is O(1). It is because, in insertion sort, an extra variable is required for swapping.**

# **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. It is also the simplest algorithm. It is an in-place comparison sorting algorithm. In this algorithm, the array is divided into two parts, first is sorted part, and another one is the unsorted part. Initially, the sorted part of the array is empty, and unsorted part is the given array. Sorted part is placed at the left, while the unsorted part is placed at the right.**

**In selection sort, the first smallest element is selected from the unsorted array and placed at the first position. After that second smallest element is selected and placed in the second position. The process continues until the array is entirely sorted.**

**The average and worst-case complexity of selection sort is O(n2), where n is the number of items. Due to this, it is not suitable for large data sets.**

**Selection sort is generally used when -**

* **A small array is to be sorted**
* **Swapping cost doesn't matter**
* **It is compulsory to check all elements**

**Now, let's see the algorithm of selection sort.**

## Algorithm

1. **SELECTION SORT(arr, n)**
2. **Step 1: Repeat Steps 2 and 3 for i = 0 to n-1**
3. **Step 2: CALL SMALLEST(arr, i, n, pos)**
4. **Step 3: SWAP arr[i] with arr[pos]**
5. **[END OF LOOP]**
6. **Step 4: EXIT**

1. **SMALLEST (arr, i, n, pos)**
2. **Step 1: [INITIALIZE] SET SMALL = arr[i]**
3. **Step 2: [INITIALIZE] SET pos = i**
4. **Step 3: Repeat for j = i+1 to n**
5. **if (SMALL > arr[j])**
6. **SET SMALL = arr[j]**
7. **SET pos = j**
8. **[END OF if]**
9. **[END OF LOOP]**
10. **Step 4: RETURN pos**

## Working of Selection sort Algorithm

**Now, let's see the working of the Selection sort Algorithm.**

**To understand the working of the Selection sort algorithm, let's take an unsorted array. It will be easier to understand the Selection sort via an example.**

**Let the elements of array are -**

**selection Sort Algorithm**

**Now, for the first position in the sorted array, the entire array is to be scanned sequentially.**

**At present, 12 is stored at the first position, after searching the entire array, it is found that 8 is the smallest value.**

**selection Sort Algorithm**

**So, swap 12 with 8. After the first iteration, 8 will appear at the first position in the sorted array.**

**selection Sort Algorithm**

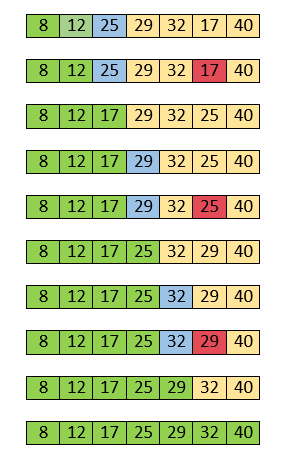
**For the second position, where 29 is stored presently, we again sequentially scan the rest of the items of unsorted array. After scanning, we find that 12 is the second lowest element in the array that should be appeared at second position.**

**selection Sort Algorithm**

**Now, swap 29 with 12. After the second iteration, 12 will appear at the second position in the sorted array. So, after two iterations, the two smallest values are placed at the beginning in a sorted way.**

**selection Sort Algorithm**

**The same process is applied to the rest of the array elements. Now, we are showing a pictorial representation of the entire sorting process.**

****

**Now, the array is completely sorted.**

## Selection sort complexity

**Now, let's see the time complexity of selection sort in best case, average case, and in worst case. We will also see the space complexity of the selection sort.**

**Time Complexity**

|  |  |
| --- | --- |
| **Case** | **Time Complexity** |
| **Best Case** | **O(n2)** |
| **Average Case** | **O(n2)** |
| **Worst Case** | **O(n2)** |

* **Best Case Complexity - It occurs when there is no sorting required, i.e. the array is already sorted. The best-case time complexity of selection sort is O(n2).**
* **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 selection sort is O(n2).**
* **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 selection sort is O(n2).**

### **Space Complexity**

|  |  |
| --- | --- |
| **Space Complexity** | **O(1)** |
| **Stable** | **YES** |

* **The space complexity of selection sort is O(1). It is because, in selection sort, an extra variable is required for swapping.**

# **Bubble sort**

**Bubble sort works on the repeatedly swapping of adjacent elements until they are not in the intended order. It is called bubble sort because the movement of array elements is just like the movement of air bubbles in the water. Bubbles in water rise up to the surface; similarly, the array elements in bubble sort move to the end in each iteration.**

**Although it is simple to use, it is primarily used as an educational tool because the performance of bubble sort is poor in the real world. It is not suitable for large data sets. The average and worst-case complexity of Bubble sort is O(n2), where n is a number of items.**

**Bubble short is majorly used where -**

**complexity does not matter**

**simple and shortcode is preferred**

## Algorithm

**In the algorithm given below, suppose arr is an array of n elements. The assumed swap function in the algorithm will swap the values of given array elements.**

1. **begin BubbleSort(arr)**
2. **for all array elements**
3. **if arr[i] > arr[i+1]**
4. **swap(arr[i], arr[i+1])**
5. **end if**
6. **end for**
7. **return arr**
8. **end BubbleSort**

## Working of Bubble sort Algorithm

**Now, let's see the working of Bubble sort Algorithm.**

**To understand the working of bubble sort algorithm, let's take an unsorted array. We are taking a short and accurate array, as we know the complexity of bubble sort is O(n2).**

**Let the elements of array are -**

**Bubble sort Algorithm**

### **First Pass**

**Sorting will start from the initial two elements. Let compare them to check which is greater.**

**Bubble sort Algorithm**

**Here, 32 is greater than 13 (32 > 13), so it is already sorted. Now, compare 32 with 26.**

**Bubble sort Algorithm**

**Here, 26 is smaller than 36. So, swapping is required. After swapping new array will look like -**

**Bubble sort Algorithm**

**Now, compare 32 and 35.**

**Bubble sort Algorithm**

**Here, 35 is greater than 32. So, there is no swapping required as they are already sorted.**

**Now, the comparison will be in between 35 and 10.**

**Bubble sort Algorithm**

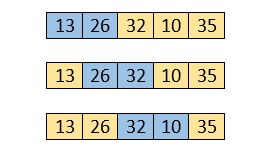
**Here, 10 is smaller than 35 that are not sorted. So, swapping is required. Now, we reach at the end of the array. After first pass, the array will be -**

**Bubble sort Algorithm**

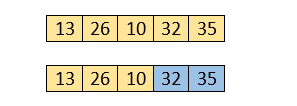
**Now, move to the second iteration.**

### **Second Pass**

**The same process will be followed for second iteration.**

****

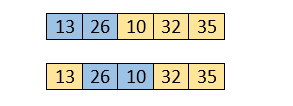
**Here, 10 is smaller than 32. So, swapping is required. After swapping, the array will be -**

****

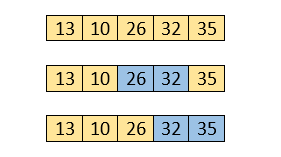
**Now, move to the third iteration.**

### **Third Pass**

**The same process will be followed for third iteration.**

****

**Here, 10 is smaller than 26. So, swapping is required. After swapping, the array will be -**

****

**Now, move to the fourth iteration.**

### **Fourth pass**

**Similarly, after the fourth iteration, the array will be -**

**Bubble sort Algorithm**

**Hence, there is no swapping required, so the array is completely sorted.**

## Bubble sort complexity

**Now, let's see the time complexity of bubble sort in the best case, average case, and worst case. We will also see the space complexity of bubble sort.**

### **Time Complexity**

|  |  |
| --- | --- |
| **Case** | **Time Complexity** |
| **Best Case** | **O(n)** |
| **Average Case** | **O(n2)** |
| **Worst Case** | **O(n2)** |

* **Best Case Complexity - It occurs when there is no sorting required, i.e. the array is already sorted. The best-case time complexity of bubble sort is O(n).**
* **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 bubble sort is O(n2).**
* **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 bubble sort is O(n2).**

### **Space Complexity**

|  |  |
| --- | --- |
| **Space Complexity** | **O(1)** |
| **Stable** | **YES** |

* **The space complexity of bubble sort is O(1). It is because, in bubble sort, an extra variable is required for swapping.**
* **The space complexity of optimized bubble sort is O(2). It is because two extra variables are required in optimized bubble sort.**

# **Quick Sort**

**Sorting is a way of arranging items in a systematic manner. Quicksort is the widely used sorting algorithm that makes n log n comparisons in average case for sorting an array of n elements. It is a faster and highly efficient sorting algorithm. This algorithm follows the divide and conquer approach. Divide and conquer is a technique of breaking down the algorithms into subproblems, then solving the subproblems, and combining the results back together to solve the original problem.**

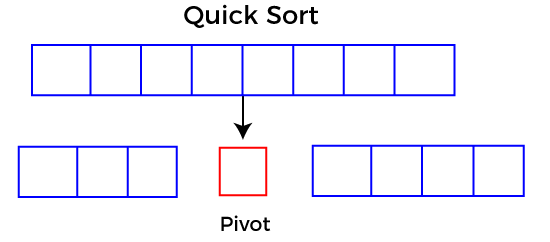
**Divide: In Divide, first pick a pivot element. After that, partition or rearrange the array into two sub-arrays such that each element in the left sub-array is less than or equal to the pivot element and each element in the right sub-array is larger than the pivot element.**

**Conquer: Recursively, sort two subarrays with Quicksort.**

**Combine: Combine the already sorted array.**

**Quicksort picks an element as pivot, and then it partitions the given array around the picked pivot element. In quick sort, a large array is divided into two arrays in which one holds values that are smaller than the specified value (Pivot), and another array holds the values that are greater than the pivot.**

**After that, left and right sub-arrays are also partitioned using the same approach. It will continue until the single element remains in the sub-array.**

****

## Choosing the pivot

**Picking a good pivot is necessary for the fast implementation of quicksort. However, it is typical to determine a good pivot. Some of the ways of choosing a pivot are as follows -**

* **Pivot can be random, i.e. select the random pivot from the given array.**
* **Pivot can either be the rightmost element of the leftmost element of the given array.**
* **Select median as the pivot element.**

**Algorithm:**

1. **QUICKSORT (array A, start, end)**
2. **{**
3. **1 if (start < end)**
4. **2 {**
5. **3 p = partition(A, start, end)**
6. **4 QUICKSORT (A, start, p - 1)**
7. **5 QUICKSORT (A, p + 1, end)**
8. **6 }**
9. **}**

**Partition Algorithm:**

**The partition algorithm rearranges the sub-arrays in a place.**

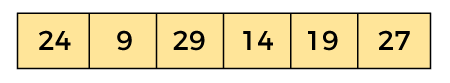
1. **PARTITION (array A, start, end)**
2. **{**
3. **1 pivot ? A[end]**
4. **2 i ? start-1**
5. **3 for j ? start to end -1 {**
6. **4 do if (A[j] < pivot) {**
7. **5 then i ? i + 1**
8. **6 swap A[i] with A[j]**
9. **7  }}**
10. **8 swap A[i+1] with A[end]**
11. **9 return i+1**
12. **}**

## Working of Quick Sort Algorithm

**Now, let's see the working of the Quicksort Algorithm.**

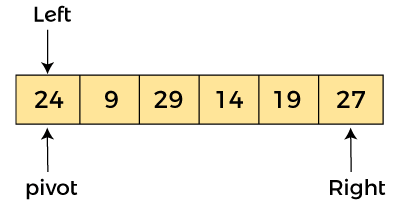
**To understand the working of quick sort, let's take an unsorted array. It will make the concept more clear and understandable.**

**Let the elements of array are -**

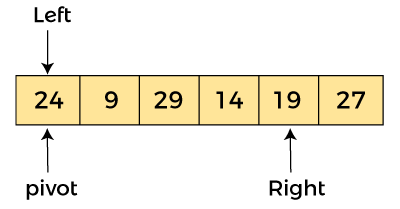
****

**In the given array, we consider the leftmost element as pivot. So, in this case, a[left] = 24, a[right] = 27 and a[pivot] = 24.**

**Since, pivot is at left, so algorithm starts from right and move towards left.**

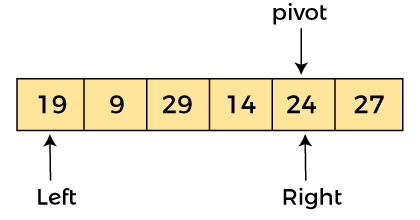
****

**Now, a[pivot] < a[right], so algorithm moves forward one position towards left, i.e. -**

****

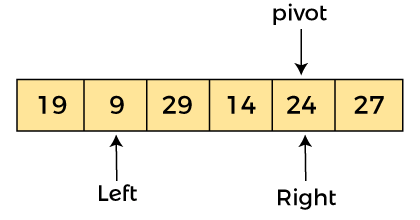
**Now, a[left] = 24, a[right] = 19, and a[pivot] = 24.**

**Because, a[pivot] > a[right], so, algorithm will swap a[pivot] with a[right], and pivot moves to right, as -**

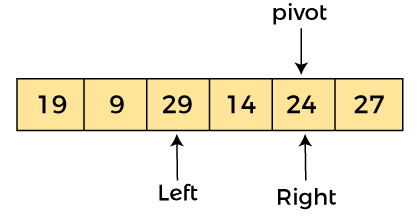
****

**Now, a[left] = 19, a[right] = 24, and a[pivot] = 24. Since, pivot is at right, so algorithm starts from left and moves to right.**

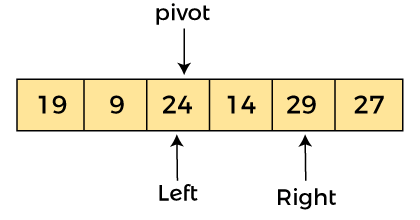
**As a[pivot] > a[left], so algorithm moves one position to right as -**

****

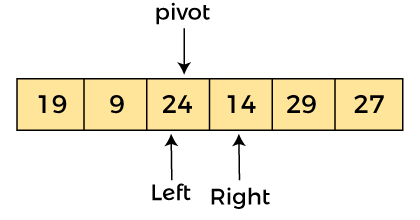
**Now, a[left] = 9, a[right] = 24, and a[pivot] = 24. As a[pivot] > a[left], so algorithm moves one position to right as -**

****

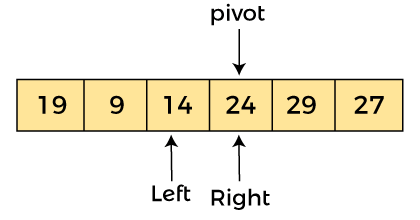
**Now, a[left] = 29, a[right] = 24, and a[pivot] = 24. As a[pivot] < a[left], so, swap a[pivot] and a[left], now pivot is at left, i.e. -**

****

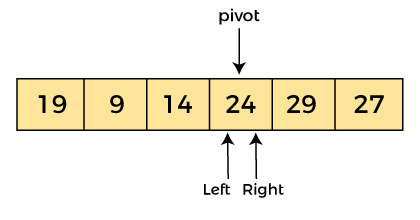
**Since, pivot is at left, so algorithm starts from right, and move to left. Now, a[left] = 24, a[right] = 29, and a[pivot] = 24. As a[pivot] < a[right], so algorithm moves one position to left, as -**

****

**Now, a[pivot] = 24, a[left] = 24, and a[right] = 14. As a[pivot] > a[right], so, swap a[pivot] and a[right], now pivot is at right, i.e. -**

****

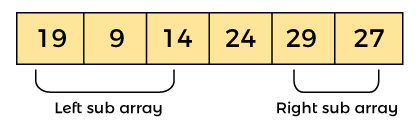
**Now, a[pivot] = 24, a[left] = 14, and a[right] = 24. Pivot is at right, so the algorithm starts from left and move to right.**

****

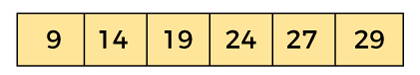
**Now, a[pivot] = 24, a[left] = 24, and a[right] = 24. So, pivot, left and right are pointing the same element. It represents the termination of procedure.**

**Element 24, which is the pivot element is placed at its exact position.**

**Elements that are right side of element 24 are greater than it, and the elements that are left side of element 24 are smaller than it.**

****

**Now, in a similar manner, quick sort algorithm is separately applied to the left and right sub-arrays. After sorting gets done, the array will be -**

****

## Quicksort complexity

## Now, let's see the time complexity of quicksort in best case, average case, and in worst case. We will also see the space complexity of quicksort.

### **1. Time Complexity**

|  |  |
| --- | --- |
| **Case** | **Time Complexity** |
| **Best Case** | **O(n\*logn)** |
| **Average Case** | **O(n\*logn)** |
| **Worst Case** | **O(n2)** |

* **Best Case Complexity - In Quicksort, the best-case occurs when the pivot element is the middle element or near to the middle element. The best-case time complexity of quicksort is O(n\*logn).**
* **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 quicksort is O(n\*logn).**
* **Worst Case Complexity - In quick sort, worst case occurs when the pivot element is either greatest or smallest element. Suppose, if the pivot element is always the last element of the array, the worst case would occur when the given array is sorted already in ascending or descending order. The worst-case time complexity of quicksort is O(n2).**

**Though the worst-case complexity of quicksort is more than other sorting algorithms such as Merge sort and Heap sort, still it is faster in practice. Worst case in quick sort rarely occurs because by changing the choice of pivot, it can be implemented in different ways. Worst case in quicksort can be avoided by choosing the right pivot element.**

### **2. Space Complexity**

|  |  |
| --- | --- |
| **Space Complexity** | **O(n\*logn)** |
| **Stable** | **NO** |

* **The space complexity of quicksort is O(n\*logn).**

# **Merge Sort**

**Merge sort is similar to the quick sort algorithm as it uses the divide and conquer approach to sort the elements. It is one of the most popular and efficient sorting algorithm. It divides the given list into two equal halves, calls itself for the two halves and then merges the two sorted halves. We have to define the merge() function to perform the merging.**

**The sub-lists are divided again and again into halves until the list cannot be divided further. Then we combine the pair of one element lists into two-element lists, sorting them in the process. The sorted two-element pairs is merged into the four-element lists, and so on until we get the sorted list.**

**Now, let's see the algorithm of merge sort.**

**Algorithm**

**In the following algorithm, arr is the given array, beg is the starting element, and end is the last element of the array.**

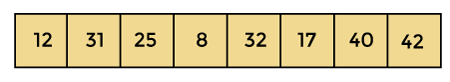
1. **MERGE\_SORT(arr, beg, end)**
3. **if beg < end**
4. **set mid = (beg + end)/2**
5. **MERGE\_SORT(arr, beg, mid)**
6. **MERGE\_SORT(arr, mid + 1, end)**
7. **MERGE (arr, beg, mid, end)**
8. **end of if**
10. **END MERGE\_SORT**

## Working of Merge sort Algorithm

**Now, let's see the working of merge sort Algorithm.**

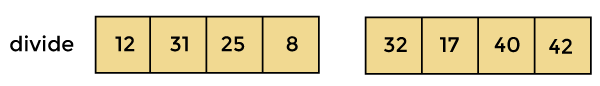
**To understand the working of the merge sort algorithm, let's take an unsorted array. It will be easier to understand the merge sort via an example.**

**Let the elements of array are -**

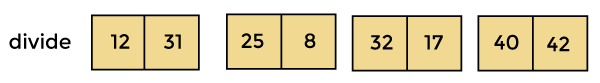
****

**According to the merge sort, first divide the given array into two equal halves. Merge sort keeps dividing the list into equal parts until it cannot be further divided.**

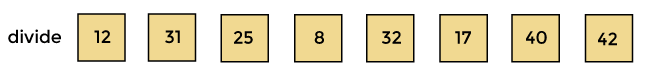
**As there are eight elements in the given array, so it is divided into two arrays of size 4.**

****

**Now, again divide these two arrays into halves. As they are of size 4, so divide them into new arrays of size 2.**

****

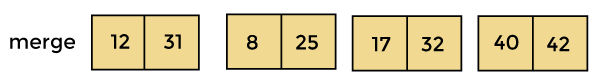
**Now, again divide these arrays to get the atomic value that cannot be further divided.**

****

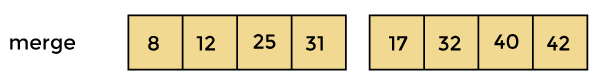
**Now, combine them in the same manner they were broken.**

**In combining, first compare the element of each array and then combine them into another array in sorted order.**

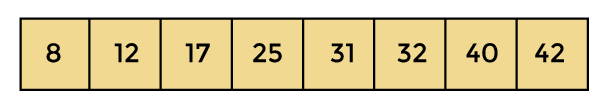
**So, first compare 12 and 31, both are in sorted positions. Then compare 25 and 8, and in the list of two values, put 8 first followed by 25. Then compare 32 and 17, sort them and put 17 first followed by 32. After that, compare 40 and 42, and place them sequentially.**

****

**In the next iteration of combining, now compare the arrays with two data values and merge them into an array of found values in sorted order.**

****

**Now, there is a final merging of the arrays. After the final merging of above arrays, the array will look like -**

****

**Now, the array is completely sorted.**

## Merge sort complexity

**Now, let's see the time complexity of merge sort in best case, average case, and in worst case. We will also see the space complexity of the merge sort.**

### **Time Complexity**

|  |  |
| --- | --- |
| **Case** | **Time Complexity** |
| **Best Case** | **O(n\*logn)** |
| **Average Case** | **O(n\*logn)** |
| **Worst Case** | **O(n\*logn)** |

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

### **Space Complexity**

|  |  |
| --- | --- |
| **Space Complexity** | **O(n)** |
| **Stable** | **YES** |

* **The space complexity of merge sort is O(n). It is because, in merge sort, an extra variable is required for swapping.**

# **Heap Sort Algorithm**

**Heap sort basically recursively performs two main operations -**

* **Build a heap H, using the elements of array.**
* **Repeatedly delete the root element of the heap formed in 1st phase.**

**Before knowing more about the heap sort, let's first see a brief description of Heap.**

### **What is a heap?**

**A heap is a complete binary tree, and the binary tree is a tree in which the node can have the utmost two children. A complete binary tree is a binary tree in which all the levels except the last level, i.e., leaf node, should be completely filled, and all the nodes should be left-justified.**

### **What is heap sort?**

**Heapsort is a popular and efficient sorting algorithm. The concept of heap sort is to eliminate the elements one by one from the heap part of the list, and then insert them into the sorted part of the list.**

**Heapsort is the in-place sorting algorithm.**

**Now, let's see the algorithm of heap sort.**

## Algorithm

1. **HeapSort(arr)**
2. **BuildMaxHeap(arr)**
3. **for i = length(arr) to 2**
4. **swap arr[1] with arr[i]**
5. **heap\_size[arr] = heap\_size[arr] ? 1**
6. **MaxHeapify(arr,1)**
7. **End**

**BuildMaxHeap(arr)**

1. **BuildMaxHeap(arr)**
2. **heap\_size(arr) = length(arr)**
3. **for i = length(arr)/2 to 1**
4. **MaxHeapify(arr,i)**
5. **End**

**MaxHeapify(arr,i)**

1. **MaxHeapify(arr,i)**
2. **L = left(i)**
3. **R = right(i)**
4. **if L ? heap\_size[arr] and arr[L] > arr[i]**
5. **largest = L**
6. **else**
7. **largest = i**
8. **if R ? heap\_size[arr] and arr[R] > arr[largest]**
9. **largest = R**
10. **if largest != i**
11. **swap arr[i] with arr[largest]**
12. **MaxHeapify(arr,largest)**
13. **End**

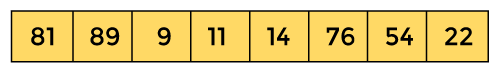
## Working of Heap sort Algorithm

**Now, let's see the working of the Heapsort Algorithm.**

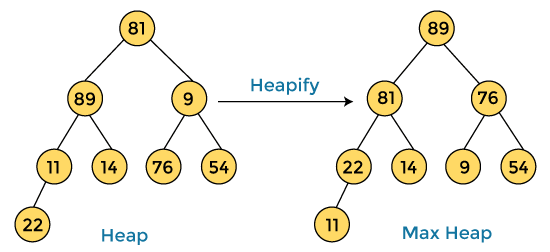
**In heap sort, basically, there are two phases involved in the sorting of elements. By using the heap sort algorithm, they are as follows -**

* **The first step includes the creation of a heap by adjusting the elements of the array.**
* **After the creation of heap, now remove the root element of the heap repeatedly by shifting it to the end of the array, and then store the heap structure with the remaining elements.**

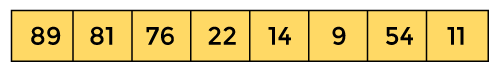
**Now let's see the working of heap sort in detail by using an example. To understand it more clearly, let's take an unsorted array and try to sort it using heap sort. It will make the explanation clearer and easier.**

****

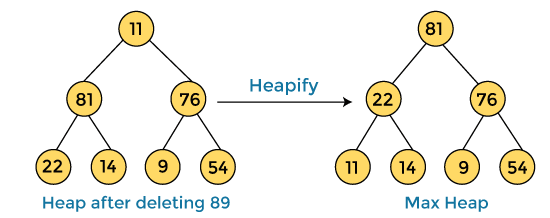
**First, we have to construct a heap from the given array and convert it into max heap.**

****

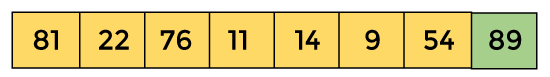
**After converting the given heap into max heap, the array elements are -**

****

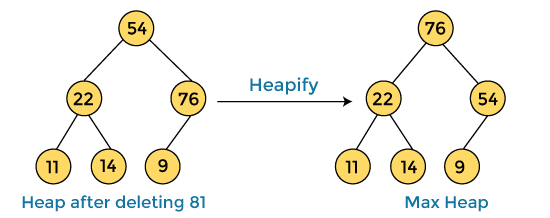
**Next, we have to delete the root element (89) from the max heap. To delete this node, we have to swap it with the last node, i.e. (11). After deleting the root element, we again have to heapify it to convert it into max heap.**

****

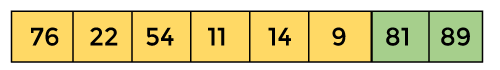
**After swapping the array element 89 with 11, and converting the heap into max-heap, the elements of array are -**

****

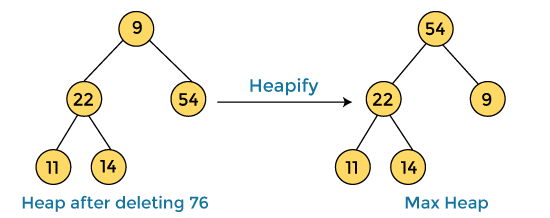
**In the next step, again, we have to delete the root element (81) from the max heap. To delete this node, we have to swap it with the last node, i.e. (54). After deleting the root element, we again have to heapify it to convert it into max heap.**

****

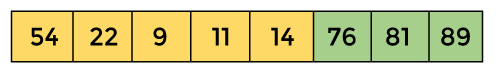
**After swapping the array element 81 with 54 and converting the heap into max-heap, the elements of array are -**

****

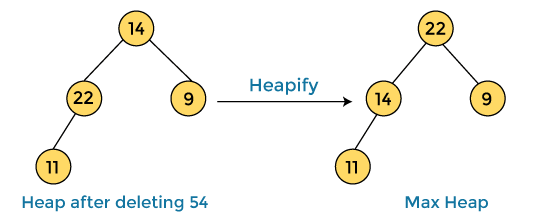
**In the next step, we have to delete the root element (76) from the max heap again. To delete this node, we have to swap it with the last node, i.e. (9). After deleting the root element, we again have to heapify it to convert it into max heap.**

****

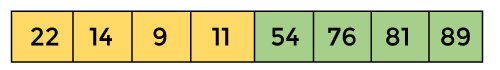
**After swapping the array element 76 with 9 and converting the heap into max-heap, the elements of array are -**

****

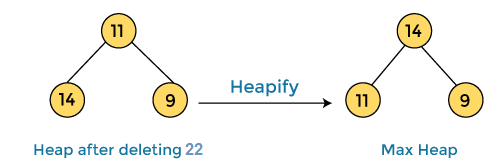
**In the next step, again we have to delete the root element (54) from the max heap. To delete this node, we have to swap it with the last node, i.e. (14). After deleting the root element, we again have to heapify it to convert it into max heap.**

****

**After swapping the array element 54 with 14 and converting the heap into max-heap, the elements of array are -**

****

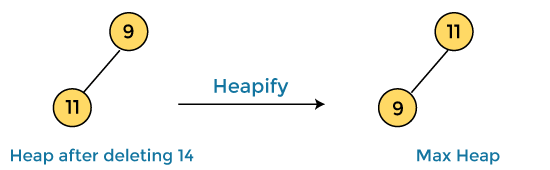
**In the next step, again we have to delete the root element (22) from the max heap. To delete this node, we have to swap it with the last node, i.e. (11). After deleting the root element, we again have to heapify it to convert it into max heap.**

****

**After swapping the array element 22 with 11 and converting the heap into max-heap, the elements of array are -**

****

**In the next step, again we have to delete the root element (14) from the max heap. To delete this node, we have to swap it with the last node, i.e. (9). After deleting the root element, we again have to heapify it to convert it into max heap.**

****

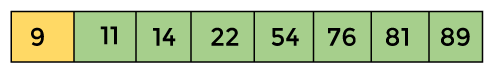
**After swapping the array element 14 with 9 and converting the heap into max-heap, the elements of array are -**

****

**In the next step, again we have to delete the root element (11) from the max heap. To delete this node, we have to swap it with the last node, i.e. (9). After deleting the root element, we again have to heapify it to convert it into max heap.**

****

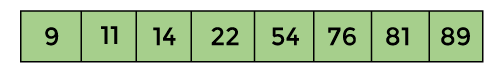
**After swapping the array element 11 with 9, the elements of array are -**

****

**Now, heap has only one element left. After deleting it, heap will be empty.**

****

**After completion of sorting, the array elements are -**

****

**Now, the array is completely sorted.**

## Heap sort complexity

**Now, let's see the time complexity of Heap sort in the best case, average case, and worst case. We will also see the space complexity of Heapsort.**

### **Time Complexity**

|  |  |
| --- | --- |
| **Case** | **Time Complexity** |
| **Best Case** | **O(n logn)** |
| **Average Case** | **O(n log n)** |
| **Worst Case** | **O(n log n)** |

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

**The time complexity of heap sort is O(n logn) in all three cases (best case, average case, and worst case). The height of a complete binary tree having n elements is logn.**

### **Space Complexity**

|  |  |
| --- | --- |
| **Space Complexity** | **O(1)** |
| **Stable** | **N0** |

* **The space complexity of Heap sort is O(1).**

# **Radix Sort**

**The process of radix sort works similar to the sorting of students names, according to the alphabetical order. In this case, there are 26 radix formed due to the 26 alphabets in English. In the first pass, the names of students are grouped according to the ascending order of the first letter of their names. After that, in the second pass, their names are grouped according to the ascending order of the second letter of their name. And the process continues until we find the sorted list.**

**Now, let's see the algorithm of Radix sort.**

## Algorithm

1. **radixSort(arr)**
2. **max = largest element in the given array**
3. **d = number of digits in the largest element (or, max)**
4. **Now, create d buckets of size 0 - 9**
5. **for i -> 0 to d**
6. **sort the array elements using counting sort (or any stable sort) according to the digits at**
7. **the ith place**

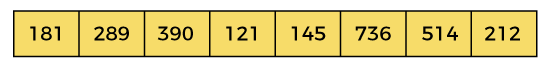
## Working of Radix sort Algorithm

**Now, let's see the working of Radix sort Algorithm.**

**The steps used in the sorting of radix sort are listed as follows -**

* **First, we have to find the largest element (suppose max) from the given array. Suppose 'x' be the number of digits in max. The 'x' is calculated because we need to go through the significant places of all elements.**
* **After that, go through one by one each significant place. Here, we have to use any stable sorting algorithm to sort the digits of each significant place.**

**Now let's see the working of radix sort in detail by using an example. To understand it more clearly, let's take an unsorted array and try to sort it using radix sort. It will make the explanation clearer and easier.**

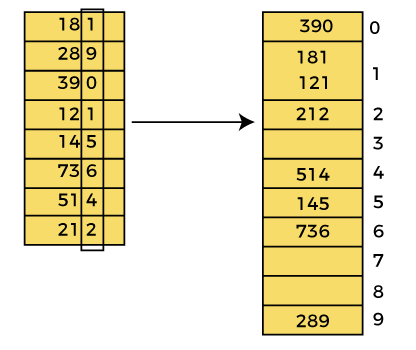
****

**In the given array, the largest element is 736 that have 3 digits in it. So, the loop will run up to three times (i.e., to the hundreds place). That means three passes are required to sort the array.**

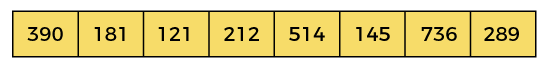
**Now, first sort the elements on the basis of unit place digits (i.e., x = 0). Here, we are using the counting sort algorithm to sort the elements.**

### **Pass 1:**

**In the first pass, the list is sorted on the basis of the digits at 0's place.**

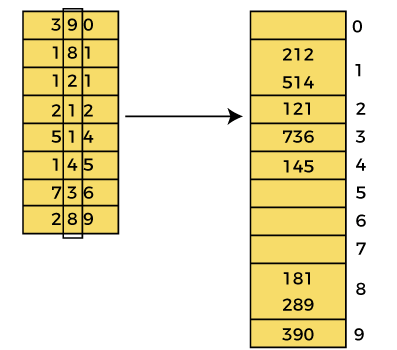
****

**After the first pass, the array elements are -**

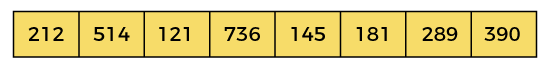
****

### **Pass 2:**

**In this pass, the list is sorted on the basis of the next significant digits (i.e., digits at 10th place).**

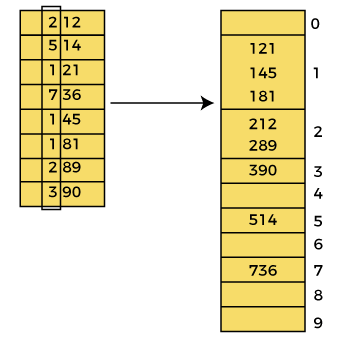
****

**After the second pass, the array elements are -**

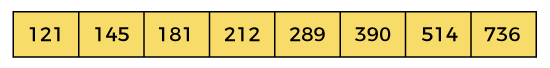
****

### **Pass 3:**

**In this pass, the list is sorted on the basis of the next significant digits (i.e., digits at 100th place).**

****

**After the third pass, the array elements are –**

****

**Now, the array is sorted in ascending order.**

## Radix sort complexity

**Now, let's see the time complexity of Radix sort in best case, average case, and worst case. We will also see the space complexity of Radix sort.**

### **1. Time Complexity**

|  |  |
| --- | --- |
| **Case** | **Time Complexity** |
| **Best Case** | **Ω(n+k)** |
| **Average Case** | **θ(nk)** |
| **Worst Case** | **O(nk)** |

* **Best Case Complexity - It occurs when there is no sorting required, i.e. the array is already sorted. The best-case time complexity of Radix sort is Ω(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 Radix sort is θ(nk).**
* **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 Radix sort is O(nk).**

**Radix sort is a non-comparative sorting algorithm that is better than the comparative sorting algorithms. It has linear time complexity that is better than the comparative algorithms with complexity O(n logn).**

### **2. Space Complexity**

|  |  |
| --- | --- |
| **Space Complexity** | **O(n + k)** |
| **Stable** | **YES** |

* **The space complexity of Radix sort is O(n + k).**