Chapter 2

Searching and Sorting

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.
- Two popular search methods are
- **Linear Search**
- **Binary Search.**

Linear Search Algorithm

- Linear search is also called as sequential search algorithm.
- It is the simplest searching algorithm.
- In Linear search, 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 is found, then the location of the item is returned; otherwise, the algorithm returns NULL.

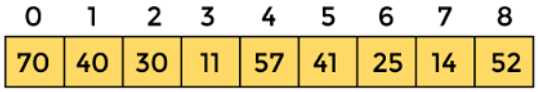
Algorithm of linear search:

```
Linear_Search(a, n, val) // 'a' is the given array, 'n' is the size of given array, 'val' is the value to search
Step 1: set pos = -1
Step 2: set i = 1
Step 3: repeat step 4 while i <= n
Step 4: if a[i] == val
set pos = i
print pos
go to step 6
[end of if]
set ii = i + 1
[end of loop]
Step 5: if pos = -1
print "value is not present in the array "
[end of if]
Step 6: exit
```

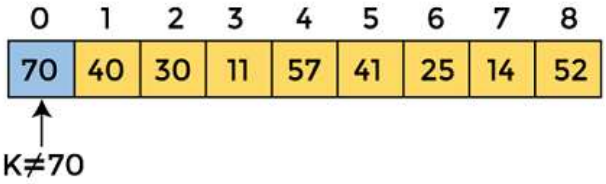
Working of Linear search

Now, let's see the working of the linear search Algorithm.
To understand the working of linear search algorithm, let's take an unsorted array. It will be easy to understand the working of linear search with an example.

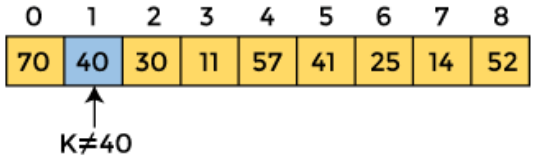
Let the elements of array are -

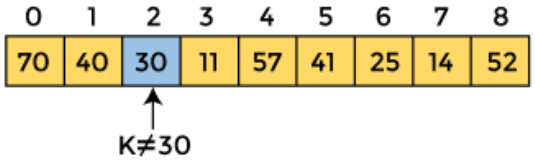


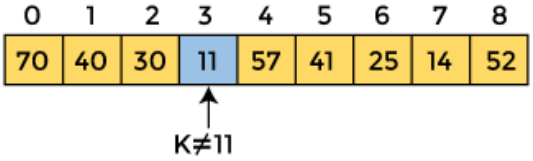
Let the element to be searched is **K = 41**
Now, start from the first element and compare **K** with each element of the array.

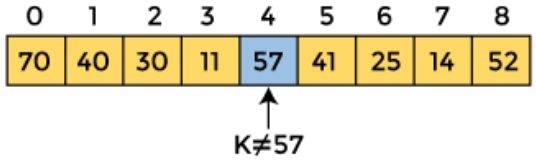


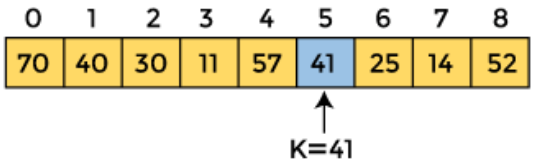
The value of **K,** i.e., **41,** is not matched with the first element of the array. So, move to the next element. And follow the same process until the respective element is found.











Now, the element to be searched is found. So algorithm will return the index of the element matched.

Binary Search

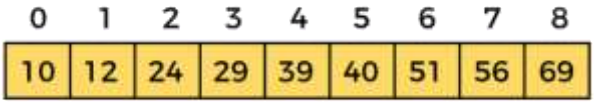
Binary search is the search technique that works efficiently on sorted lists. Hence, to search an element into some list using the binary search technique, we must ensure that the list is sorted.
Binary search follows the divide and conquer approach in which the list is divided into two halves, and the item is compared with the middle element of the list. If the match is found then, the location of the middle element is returned. Otherwise, we search into either of the halves depending upon the result produced through the match.
There are two methods to implement the binary search algorithm -

- Iterative method
- Recursive method

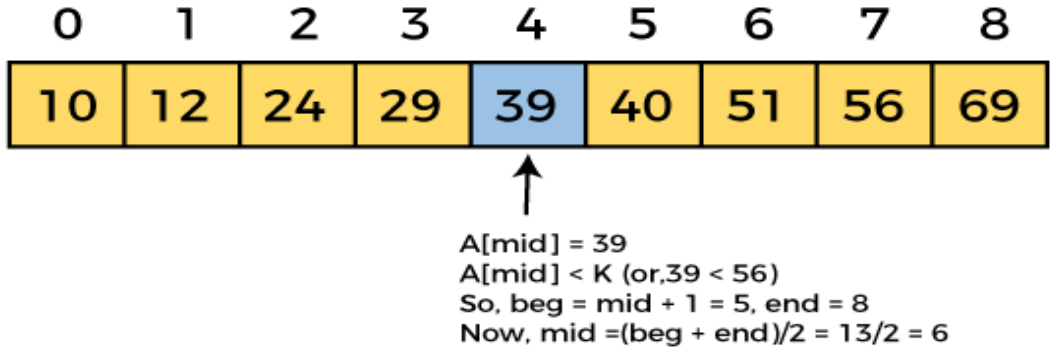
The recursive method of binary search follows the divide and conquer approach.

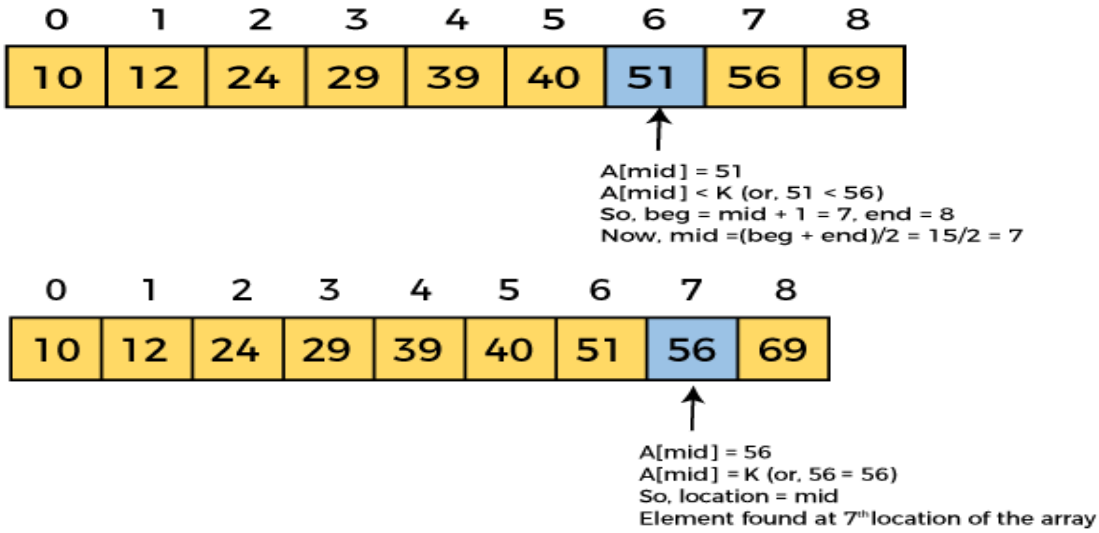
Working of Binary search:

Let the elements of array are -



Let the element to search is, **K = 56**
We have to use the below formula to calculate the **mid** of the array –
mid = (beg + end)/2
So, in the given array –
beg = 0 **end** = 8 **mid** = (0 + 8)/2 = 4.
So, 4 is the mid of the array.





Now, the element to search is found. So algorithm will return the index of the element matched.

Binary Search complexity

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

1. Time Complexity

Case	Time Complexity
Best Case	O(1)
Average Case	O(logn)
Worst Case	O(logn)

2. Space Complexity

Space Complexity	O(1)

- The space complexity of binary search is O(1).

Algorithm:

Binary_Search (a, lower_bound, upper_bound, val)

// 'a' is the given array,

'lower_bound' is the index of the first array element,

'upper_bound' is the index of the last array element,

'val' is the value to search

Step 1: 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

print pos

go to step 6

else if a[mid] > val

set end = mid - 1

else

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

Sorting Techniques:

A Sorting Algorithm is used to rearrange a given array or list elements according to a comparison operator on the elements.
The comparison operator is used to decide the new order of elements in the respective data structure.
Following are sorting techniques:
Selection sort
Bubble sort
Insertion sort
Quick sort
Radix sort

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(n^2)**, 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

Algorithm:

SELECTION SORT(arr, n)

Step 1: Repeat Steps 2 **and** 3 **for** i = 0 to n-1

Step 2: CALL SMALLEST(arr, i, n, pos)

Step 3: SWAP arr[i] with arr[pos]

[END OF LOOP]

Step 4: EXIT

SMALLEST (arr, i, n, pos)

Step 1: [INITIALIZE] SET SMALL = arr[i]

Step 2: [INITIALIZE] SET pos = i

Step 3: Repeat **for** j = i+1 to n

if (SMALL > arr[j])

SET SMALL = arr[j]

SET pos = j

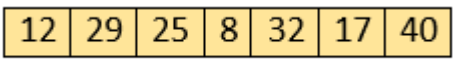
[END OF **if**]

[END OF LOOP]

Step 4: RETURN pos

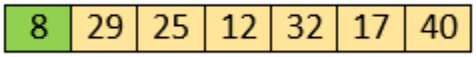
Working of Selection sort Algorithm

Now, let's see the working of the Selection sort Algorithm.
Let the elements of array are -



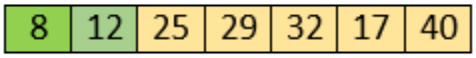
Now, for the first position in the sorted array, the entire array is to be scanned sequentially.
At present, **12** is stored at the first position, after searching the entire array, it is found that **8** is the smallest value.

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

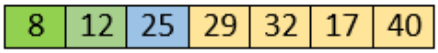


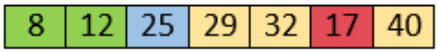
For the second position, where 29 is stored presently, we again sequentially scan the rest of the items of unsorted array. After scanning, we find that 12 is the second lowest element in the array that should be appeared at second position.

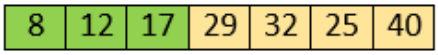
Now, swap 29 with 12. After the second iteration, 12 will appear at the second position in the sorted array. So, after two iterations, the two smallest values are placed at the beginning in a sorted way.

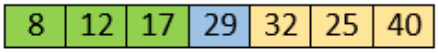


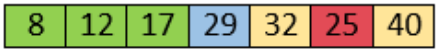
The same process is applied to the rest of the array elements. Now, we are showing a pictorial representation of the entire sorting process.











Now, the array is completely sorted.

Time Complexity

Case	Time Complexity
Best Case	$O(n^2)$
Average Case	$O(n^2)$
Worst Case	$O(n^2)$

Space Complexity O(1)

Advantages:

1. It performs very well on small lists
2. It is an in-place algorithm. It does not require a lot of space for sorting. Only one extra space is required for holding the temporal variable.
3. It performs well on items that have already been sorted.

Disadvantages:

1. It performs poorly when working on huge lists.
2. The number of iterations made during the sorting is n-squared, where n is the total number of elements in the list.
3. Other algorithms, such as quicksort, have better performance compared to the selection sort.

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 **$O(n^2)$,** where **n** is a number of items.
- Bubble short is majorly used where -
  - complexity does not matter
  - simple and shortcode is preferred

Algorithm

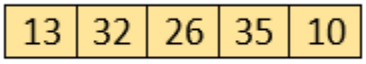
In the algorithm given below, suppose **arr** is an array of **n** elements. The assumed **swap** function in the algorithm will swap the values of given array elements.

```
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
```

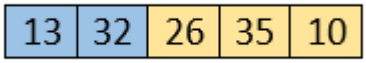
Working of Bubble sort Algorithm

Let the elements of array are -

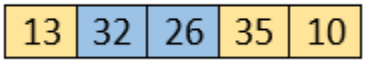


First Pass

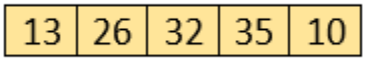
Sorting will start from the initial two elements. Let compare them to check which is greater.



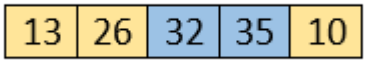
Here, 32 is greater than 13 (32 > 13), so it is already sorted. Now, compare 32 with 26.



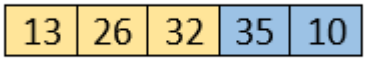
Here, 26 is smaller than 36. So, swapping is required. After swapping new array will look like -



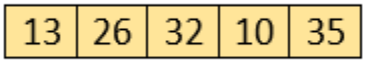
Now, compare 32 and 35.



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.



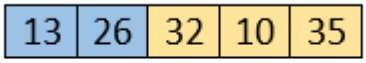
Here, 10 is smaller than 35 that are not sorted. So, swapping is required. Now, we reach at the end of the array. After first pass, the array will be -

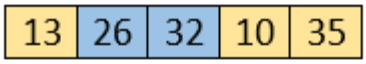


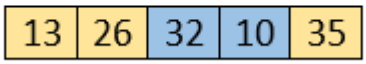
Now, move to the second iteration.

Second Pass

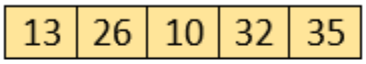
The same process will be followed for second iteration.

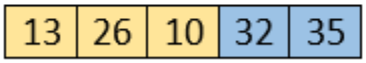






Here, 10 is smaller than 32. So, swapping is required. After swapping, the array will be -

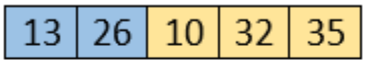


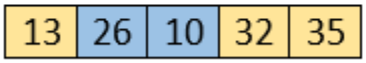


Now, move to the third iteration.

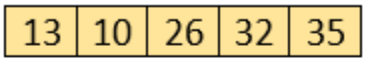
Third Pass

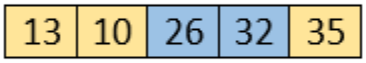
The same process will be followed for third iteration.

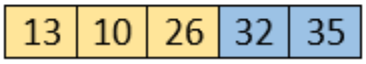




Here, 10 is smaller than 26. So, swapping is required. After swapping, the array will be -



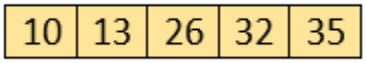




Now, move to the fourth iteration.

Fourth pass

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



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.

1. Time Complexity

Case	Time Complexity
Best Case	O(n)
Average Case	$O(n^2)$
Worst Case	$O(n^2)$

2. Space Complexity

Space Complexity	O(1)

Insertion Sort

Insertion sort is a simple sorting algorithm that works similar to the way you sort playing cards in your hands. The array is virtually split into a sorted and an unsorted part. Values from the unsorted part are picked and placed at the correct position in the sorted part.

Characteristics of Insertion Sort:

- This algorithm is one of the simplest algorithm with simple implementation
- Basically, Insertion sort is efficient for small data values
- Insertion sort is adaptive in nature, i.e. it is appropriate for data sets which are already partially sorted.

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.

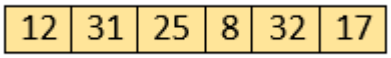
Step 6 - Repeat until the array is sorted.

Working of Insertion sort Algorithm

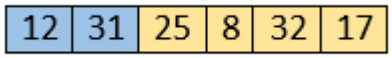
Now, let's see the working of the insertion sort Algorithm.

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.

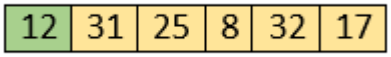
Let the elements of array are -



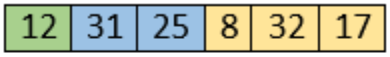
Initially, the first two elements are compared in insertion sort.

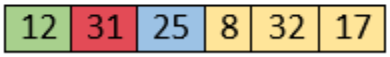


Here, 31 is greater than 12. That means both elements are already in ascending order. So, for now, 12 is stored in a sorted sub-array.



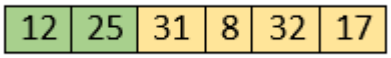
Now, move to the next two elements and compare them.



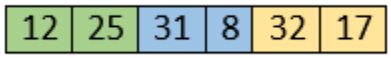


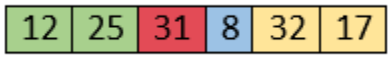
Here, 25 is smaller than 31. So, 31 is not at correct position. Now, swap 31 with 25. Along with swapping, insertion sort will also check it with all elements in the sorted array.

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.

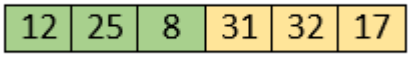


Now, two elements in the sorted array are 12 and 25. Move forward to the next elements that are 31 and 8.

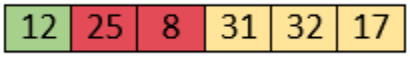




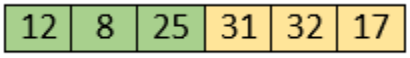
Both 31 and 8 are not sorted. So, swap them.



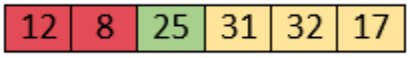
After swapping, elements 25 and 8 are unsorted.



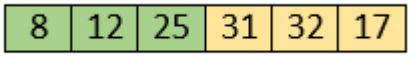
So, swap them.



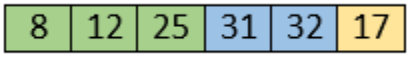
Now, elements 12 and 8 are unsorted.



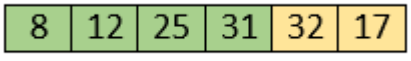
So, swap them too.



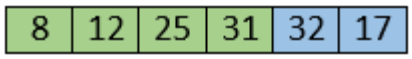
Now, the sorted array has three items that are 8, 12 and 25. Move to the next items that are 31 and 32.



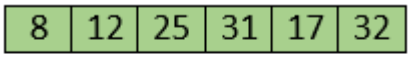
Hence, they are already sorted. Now, the sorted array includes 8, 12, 25 and 31.



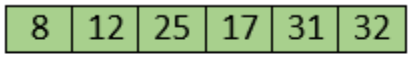
Move to the next elements that are 32 and 17.



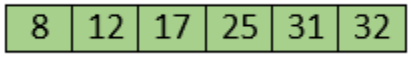
17 is smaller than 32. So, swap them.



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



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



Now, the array is completely sorted.

Insertion sort complexity

Time & space Complexity:

Case	Time Complexity
Best Case	O(n)
Average Case	$O(n^2)$
Worst Case	$O(n^2)$

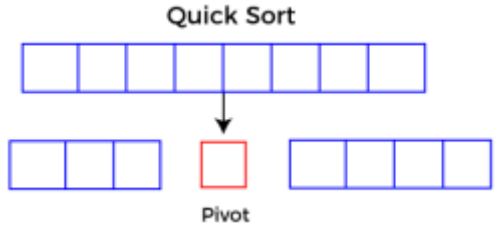
Space Complexity O(1)

Quick Sort

- 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

```
QUICKSORT (array A, start, end)
{
 if (start < end)
 {
p = partition(A, start, end)
QUICKSORT (A, start, p - 1)
QUICKSORT (A, p + 1, end)
 }
}
```

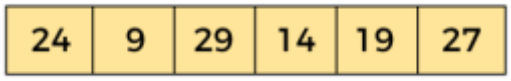
Partition Algorithm:

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

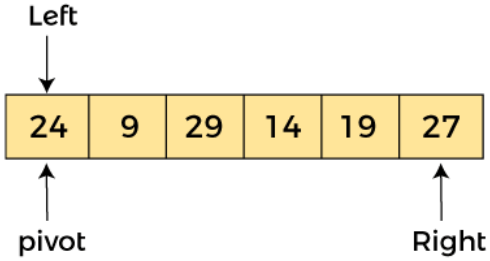
```
PARTITION (array A, start, end)
{
 pivot ? A[end]
 i ? start-1
 for j ? start to end -1 {
 do if (A[j] < pivot) {
 then i ? i + 1
 swap A[i] with A[j]
 }}
 swap A[i+1] with A[end]
 return i+1
}
```

Working of Quick Sort Algorithm

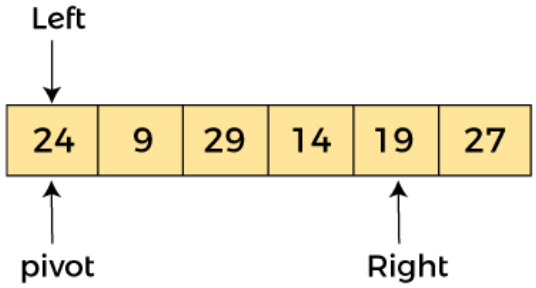
Now, let's see the working of the Quicksort Algorithm.
To understand the working of quick sort, let's take an unsorted array. It will make the concept more clear and understandable.
Let the elements of array are -



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.

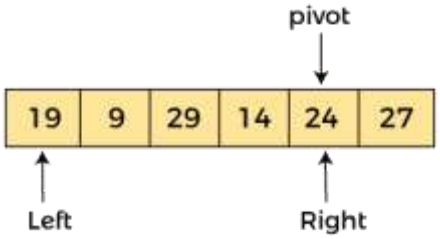


Now, a[pivot] < a[right], so algorithm moves forward one position towards left, i.e. -



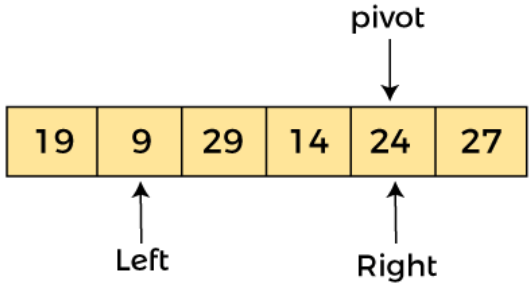
Now, a[left] = 24, a[right] = 19, and a[pivot] = 24.

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

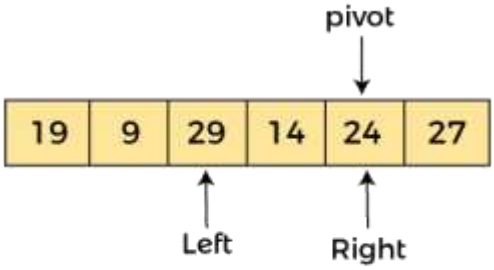


Now, a[left] = 19, a[right] = 24, and a[pivot] = 24. Since, pivot is at right, so algorithm starts from left and moves to right.

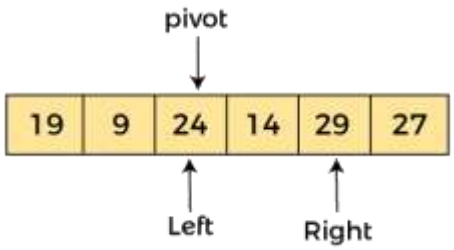
As a[pivot] > a[left], so algorithm moves one position to right as -



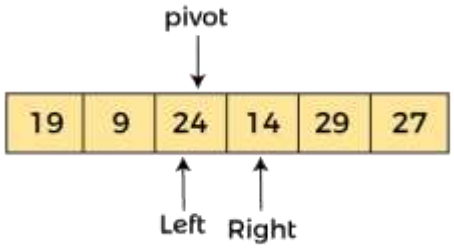
Now, a[left] = 9, a[right] = 24, and a[pivot] = 24. As a[pivot] > a[left], so algorithm moves one position to right as -



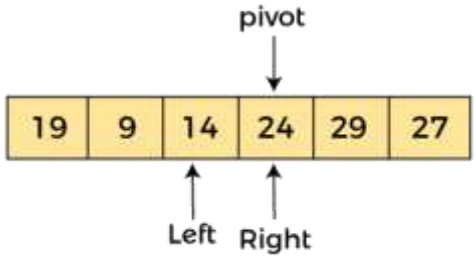
Now, a[left] = 29, a[right] = 24, and a[pivot] = 24. As a[pivot] < a[left], so, swap a[pivot] and a[left], now pivot is at left, i.e. -



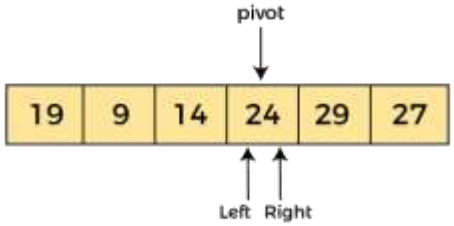
Since, pivot is at left, so algorithm starts from right, and move to left. Now, a[left] = 24, a[right] = 29, and a[pivot] = 24. As a[pivot] < a[right], so algorithm moves one position to left, as -



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. -



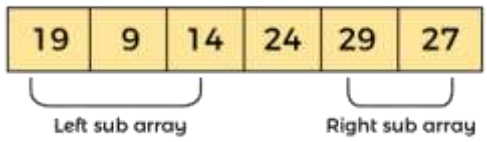
Now, a[pivot] = 24, a[left] = 14, and a[right] = 24. Pivot is at right, so the algorithm starts from left and move to right.



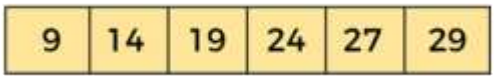
Now, a[pivot] = 24, a[left] = 24, and a[right] = 24. So, pivot, left and right are pointing the same element. It represents the termination of procedure.

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 -



Time & space Complexity:

Case	Time Complexity
Best Case	O(n*logn)
Average Case	O(n*logn)
Worst Case	$O(n^2)$
Space Complexity	**O(n*logn)**

Radix Sort Algorithm

In this article, we will discuss the Radix sort Algorithm. Radix sort is the linear sorting algorithm that is used for integers. In Radix sort, there is digit by digit sorting is performed that is started from the least significant digit to the most significant digit.

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.

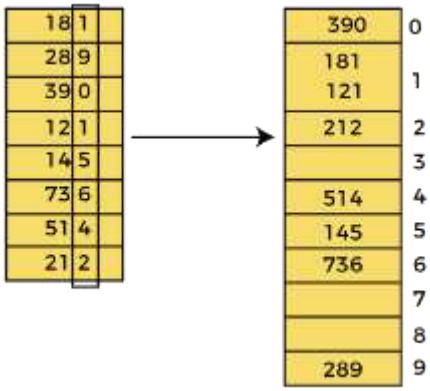
181	289	390	121	145	736	514	212

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.

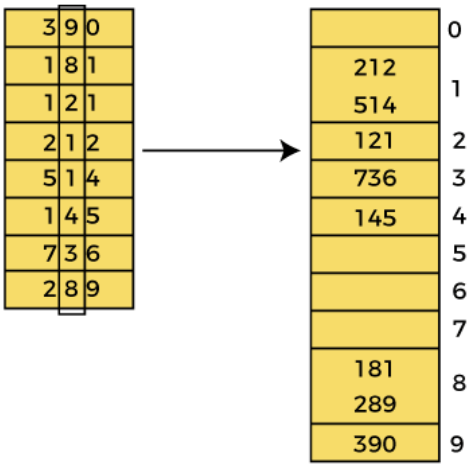


After the first pass, the array elements are -

390	181	121	212	514	145	736	289

Pass 2:

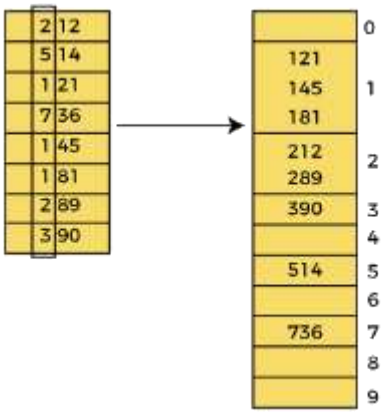
In this pass, the list is sorted on the basis of the next significant digits (i.e., digits at 10^{th} place).



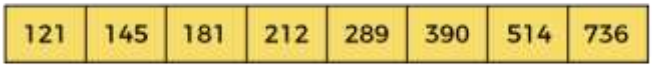
After the second pass, the array elements are -

Pass 3:

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

2. Space Complexity

Space Complexity	O(n + k)
Stable	YES

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

Comparison of Bubble sort and selection sort:

BASIS FOR COMPARISON	BUBBLE SORT	SELECTION SORT
Basic	Adjacent element is compared and swapped	Largest element is selected and swapped with the last element (in case of ascending order).
Best case time complexity	O(n)	$O(n^2)$
Efficiency	Inefficient	Improved efficiency as compared to bubble sort
Stable	Yes	No
Method	Exchanging	Selection
Speed	Slow	Fast as compared to bubble sort

Comparison of Insertion sort and Selection sort:

BASIS FOR COMPARISON	INSERTION SORT	SELECTION SORT
Basic	The data is sorted by inserting the data into an existing sorted file.	The data is sorted by selecting and placing the consecutive elements in sorted location.
Nature	Stable	Unstable
Process to be followed	Elements are known beforehand while location to place them is searched.	Location is previously known while elements are searched.
Immediate data	Insertion sort is live sorting technique which can deal with immediate data.	It can not deal with immediate data, it needs to be present at the beginning.
Best case complexity	O(n)	$O(n^2)$

Comparison of Quick sort and Radix sort:

Nos.	Quick Sort	Radix Sort
1	It is based on the divide and conquer principle and then merge.	It is based on the sorting of elements as per their unit,tens,hundered,...digits
2	According to the pivot element the array is divided into two sub array.	Numbers are sorted according to their digit position.
3	Choice of pivot element is important or else performance might degrade.	No such pivot element is used
4	Quick sort compares the item values against each other	Radix sort simply sorts by the binary representation of data. The items(as a whole) are never compared
5	The running time is O(n log n)	The running time is O(kn) where k is the number of digits in the largest number which is very fast
6	It is massively recursive	It has very less of recursion.
7	Comparatively requires more memory	Comparatively requires less memory
8	Slower than radix sort	Faster than quick sort as there is not even single comparison involved