Lesson:

Divide and Conquer

Pre Requisites:

- Basic JAVA syntax
- Recursion concepts

List of concepts involved :

- Introduction to divide and conquer strategy
- Merge sort Algorithm
- Working of merge sort algorithm
- time and space complexity analysis of merge sort algorithm
- Quick sort Algorithm
- working of Quick sort algorithm
- Time and space complexity analysis of quick sort algorithm
- Randomized quick sort algorithm

A divide and conquer algorithm is a strategy of solving a large problem by

- Breaking the problem into smaller sub-problems
- Solving the sub-problems, and
- Combining them to get the desired output.
- To use the divide and conquer algorithm, recursion is used.

Here are the steps involved:

- Divide: Divide the given problem into subproblems using recursion.
- Conquer: Solve the smaller subproblems recursively. If the subproblem is small enough, then solve it directly.
- Combine: Combine the solutions of the sub-problems that are part of the recursive process to solve the actual problem.

Merge sort :

Merge sort uses the divide and conquer approach to sort the elements. It is one of the most popular and efficient sorting algorithms. 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 are merged into the four-element lists, and so on until we get the sorted list.

Algorithm of merge sort :

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

```
MERGE_SORT(array, begin, end)
if begin < end
set mid = begin + (end-begin)/2
MERGE_SORT(array, begin, mid)
MERGE_SORT(array, mid + 1, end)
MERGE (array, begin, mid, end)
end of if
END MERGE_SORT
```

working of merge sort algorithm :

Now, let's look at the working of merge sort algorithm on an example :

let the given array be,

4	3	5	2	9	8	7	1

To sort this array divide,

4	3	5	2	9	8	7	1

Again divide it until the size does not become 1 for each smaller part.

4	3	5	2	9	8	7	1

divide again,

4	3	5	2	9	8	7	1

now that we have each subarray of size 1. we will now merge them into one single array.

3	4	2	5	8	9	1	7

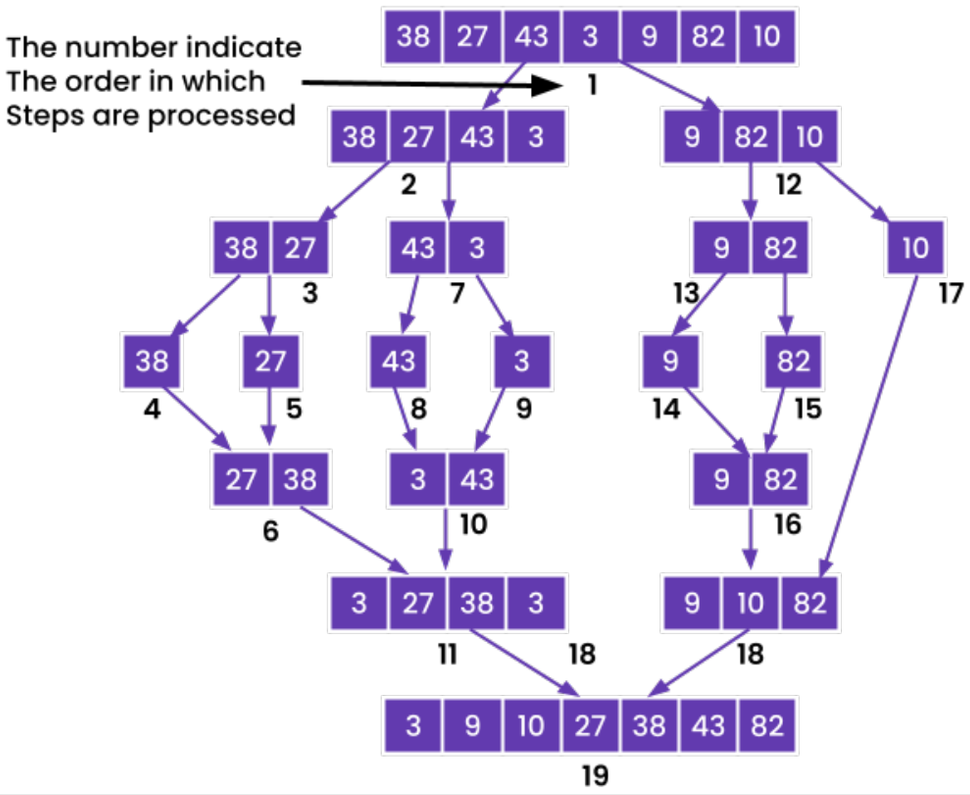
Merge again,

2	3	4	5	1	7	8	9

merge one last time ,

1	2	3	4	5	7	8	9

Another example and tree diagram for merge sort



Merge Sort Time and space complexity analysis:

Now, let us calculate time complexity with the steps. our very own first step was to divide the input into two halves which comprised us of a logarithmic time complexity ie. log(N) where N is the number of elements in the array.

Our second step was to merge back the array into a single array, so if we observe it in all the number of elements to be merged N, and to merge back we use a simple loop which runs over all the N elements giving a time complexity of O(N).
finally, total time complexity will be - step -1 + step-2
$T(n) = 2T(n/2) + O(n)$
The solution of the above recurrence can be written as O(nLogn).
The array of size N is divided into a max of Logn parts, and the merging of all subarrays into a single array takes O(N) time, the worst-case run time of this algorithm is O(nLogn).
The time complexity of MergeSort is O(n*Log n) in all the 3 cases (worst, average and best) as the mergesort always divides the array into two halves and takes linear time to merge two halves.
Now let's look at how we can calculate the space complexity for merge sort.

Let us take an example again.

Once we call merge sort for the entire array (N = 6), the array is divided into two parts, (each of size N / 2 = 3).

However, we must note that function calls are not running in parallel. Everytime during the divide phase, we are making a single function call, first with the left part and then waiting for its return (with sorted array in place) to call for the right part.

The same happens for the next function calls, till we get a single element, in which case we return.
So, the first time we encounter a single element in a function call, it would be as follows:
At any point, in the call stack, we would be having a maximum of O(logN) function calls.

Now, once we are returned from the left and the right call, the merge phase for that function call would begin.
So, if the size of the subarray for that function call is N with its left and right part of size N/2, we would be creating a new array of size N to do the merge.
Also note that once we return from any function, the extra space we take to merge the two sorted subarrays is also freed. So, we need not sum up the extra space taken in each function call since maximum space taken at any point is something we are concerned about.

Hence, during the divide phase, the maximum space we take in the recursion call stack is O(logN) and while merging, out of all the function calls, the maximum space we take is O(N) when we merge two subarrays of the entire array. Since O(N) is dominating, we would consider O(N) to be the space complexity of the merge sort.

Code : LP_Code1.java

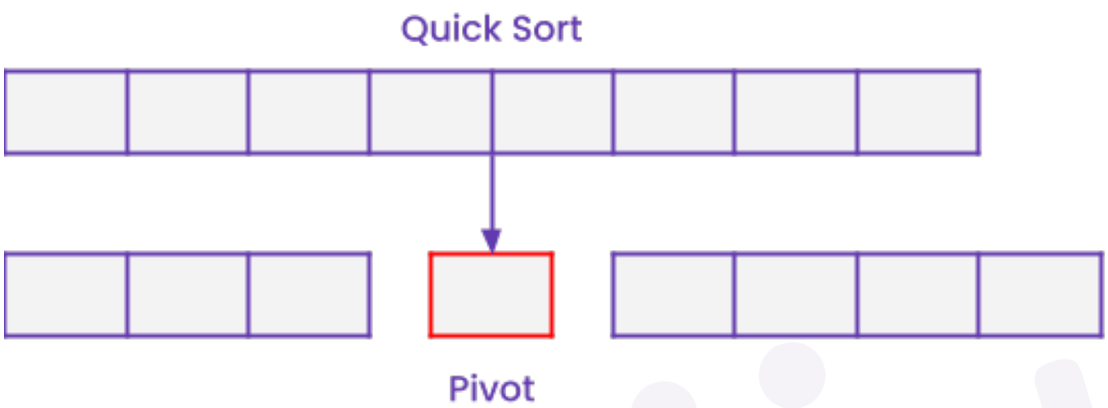
Output :

```
Before sorting array elements are -
6 4 2 7 9 11 12 5
After sorting array elements are -
2 4 5 6 7 9 11 12
```

Quick sort Algorithm :

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)
- end if

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

}

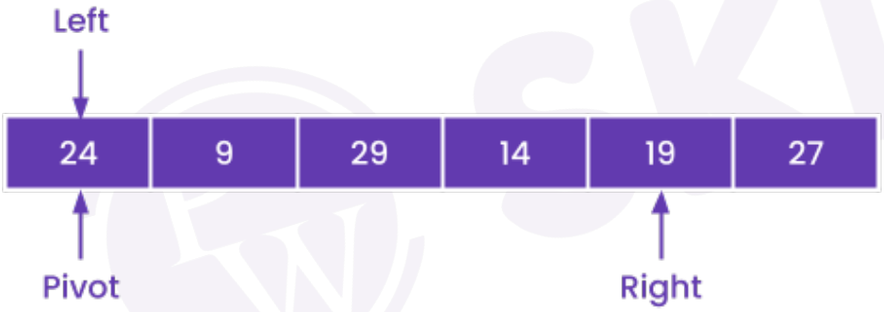
Working of quick sort algorithm :

let's understand the working of quicksort algorithm with the help of an example :

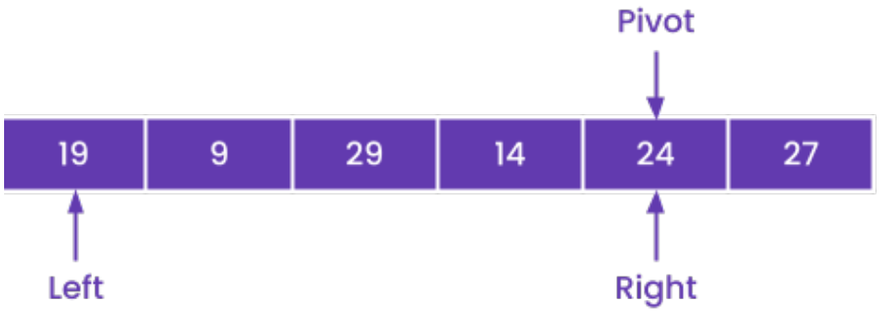
Let the elements of array are -

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.
The pivot is at left, so the algorithm starts from right and moves towards left.

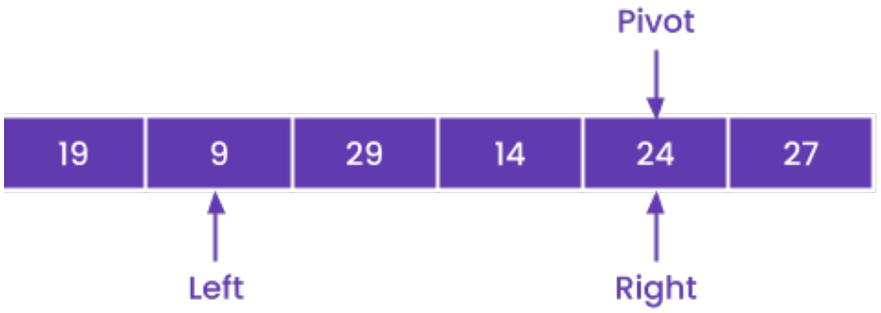
Now, a[pivot] < a[right], so algorithm moves forward one position towards left, i.e. -



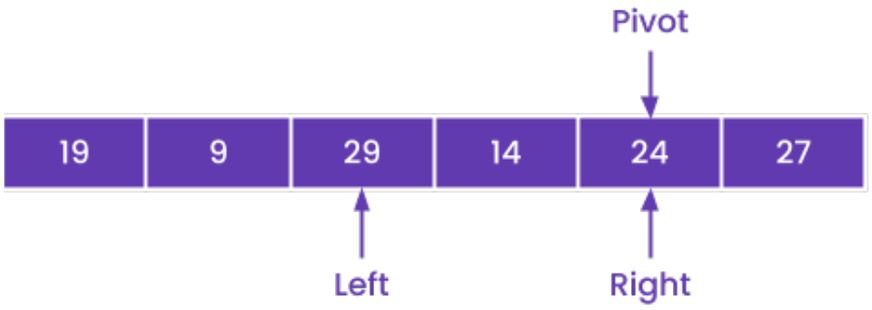
Now, a[left] = 24, a[right] = 19, and a[pivot] = 24.
Because, a[pivot] > a[right], so, algorithm will swap a[pivot] with a[right], and pivot moves to right, as -



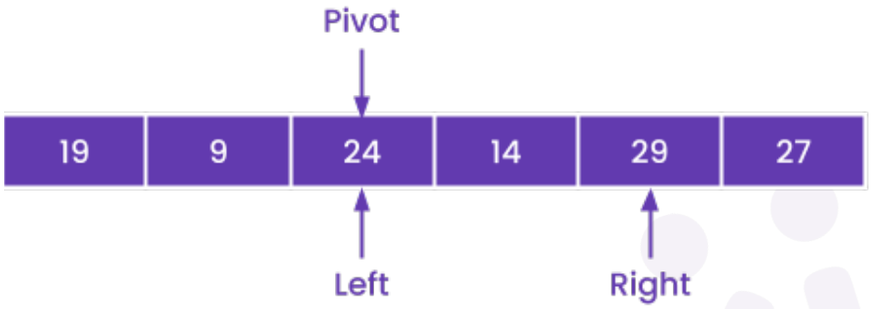
Now, a[left] = 19, a[right] = 24, and a[pivot] = 24. Since, the pivot is at right, so the algorithm starts from left and moves to right.
As a[pivot] > a[left], so algorithm moves one position to right as -



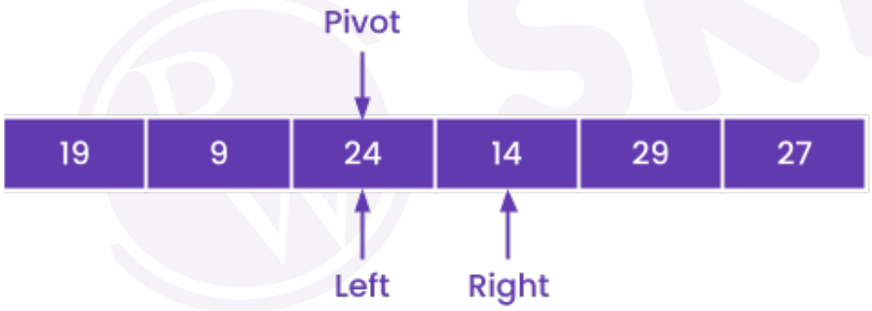
Now, a[left] = 9, a[right] = 24, and a[pivot] = 24. As a[pivot] > a[left], so algorithm moves one position to right as



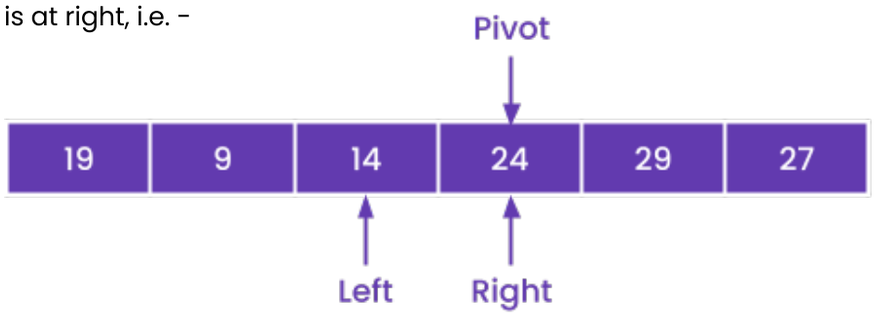
Now, a[left] = 29, a[right] = 24, and a[pivot] = 24. As a[pivot] < a[left], so, swap a[pivot] and a[left], now pivot is at left, i.e. -



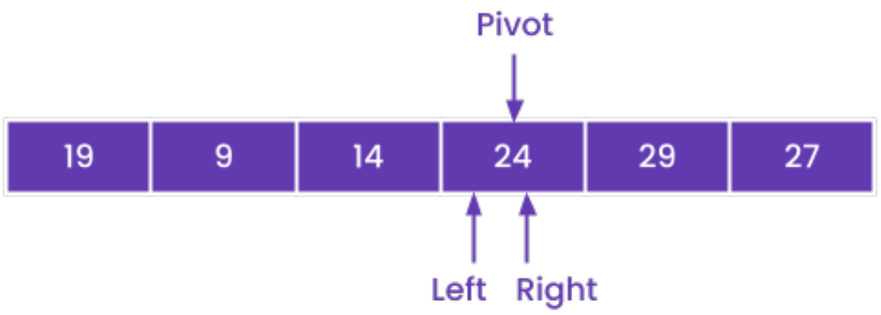
Since, the pivot is at left, so the algorithm starts from right, and moves to left. Now, a[left] = 24, a[right] = 29, and a[pivot] = 24. As a[pivot] < a[right], so algorithm moves one position to left, as -



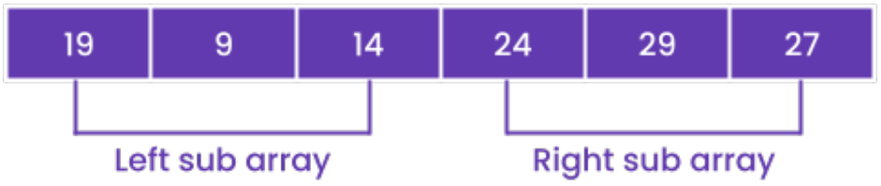
Now, a[pivot] = 24, a[left] = 24, and a[right] = 14. As a[pivot] > a[right], so, swap a[pivot] and a[right], now pivot is at right, i.e. -



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



Now, a[pivot] = 24, a[left] = 24, and a[right] = 24. So, pivot, left and right are pointing to 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 on the right side of element 24 are greater than it, and the elements that are on the 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 -

Quick sort time and space complexity analysis :

- **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 quicksort, the worst case occurs when the pivot element is either the 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).

NOTE : Worst case in quicksort can be avoided by choosing the right pivot element.
The space complexity of quicksort is O(n*logn).

Code : LP_Code2.java

output :

```
Before sorting array elements are -
2 1 5 3 8 9
After sorting array elements are -
1 2 3 5 8 9
```

Randomized quicksort algorithm :

The worst case time complexity comes out to be O(n2) which happens because of wrongly chosen pivot elements in the worst case scenario, which fails to partition the array in a balanced way. If we somehow choose a pivot element that tends to keep the array partitioning balanced, then we can claim the worst case time complexity to be O(n * log n). So we try to randomly choose our pivot element.
We have a built-in pseudo random number method in java which would help in selecting the random index, which gives us the element about which the partition is to be done.

Code : LP_Code3.java

output :

```
1 2 3 5 8 9
```

Next class teaser:

- Greedy Algorithms