SORTING ALGORITHMS

TEAM CRYSTALS

Anuja Ajay

Aysha Fathima

Nimisha R S

Sankari Suresh

SORTING :

Sorting is the process of arranging a collection of data or elements in a specific order, such as numerical or alphabetical. Sorting algorithms are used to implement this process in computer programs.

There are many types of sorting algorithms, including:

1. Bubble Sort:
   The simplest sorting algorithm that repeatedly steps through the list and compares adjacent elements, swapping them if they are in the wrong order.

2. Selection Sort:
   This algorithm sorts an array by repeatedly finding the minimum element from the unsorted part of the array and putting it at the beginning.

3. Insertion Sort:
   This algorithm builds the final sorted array one item at a time, by inserting each new element into its correct position in the already sorted array.

4. Merge Sort:
   A divide-and-conquer algorithm that divides an array into two halves, sorts each half, and then merges the two sorted halves into one.

5. Quick Sort:
   Another divide-and-conquer algorithm that selects a pivot element and partitions the array around the pivot, with the elements smaller than the pivot on one side and larger on the other. It then recursively sorts the two partitions.

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

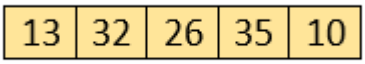
In the algorithm, suppose arr is an array of n elements. The assumed swap function in the algorithm will swap the values of given array elements.

```
begin BubbleSort(arr)
  for all array elements
    if arr[i] > arr[i+1]
      swap(arr[i], arr[i+1])
    end if
  end for
  return arr
end BubbleSort
```

Working of Bubble sort Algorithm

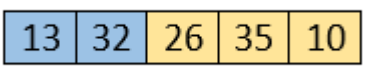
To understand the working of bubble sort algorithm, let's take an unsorted array. We are taking a short and accurate array, as we know the complexity of bubble sort is **O(n^2).**

Let the elements of array are

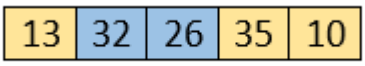


First Pass

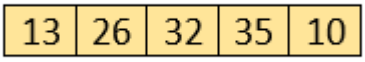
Sorting will start from the initial two elements. Let compare them to check which is greater.



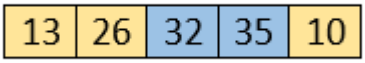
Here, 32 is greater than 13 (32 > 13), so it is already sorted. Now, compare 32 with 26.



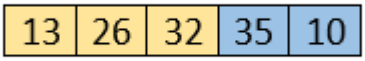
Here, 26 is smaller than 36. So, swapping is required. After swapping new array will look like



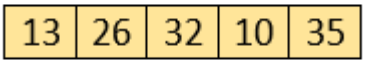
Now, compare 32 and 35.



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.



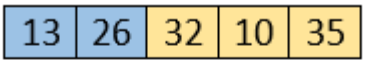
Here, 10 is smaller than 35 that are not sorted. So, swapping is required. Now, we reach at the end of the array. After first pass, the array will be

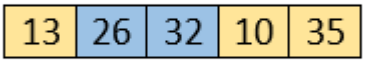


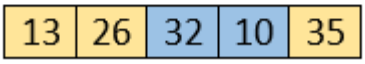
Now, move to the second iteration.

Second Pass

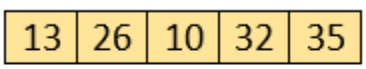
The same process will be followed for second iteration.

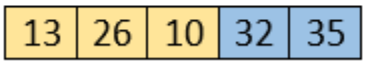






Here, 10 is smaller than 32. So, swapping is required. After swapping, the array will be

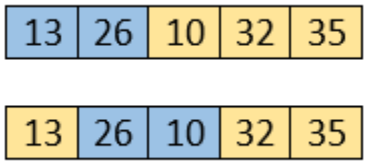




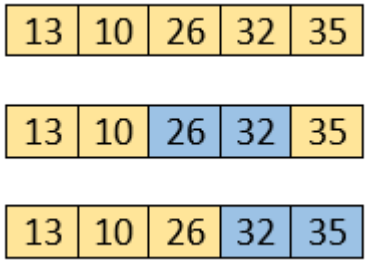
Now, move to the third iteration.

Third Pass

The same process will be followed for third iteration.



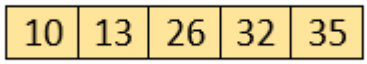
Here, 10 is smaller than 26. So, swapping is required. After swapping, the array will be



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

1. Time Complexity

Case Time Complexity

- Best Case : O(n)
- Average Case: $O(n^2)$
- Worst Case: $O(n^2)$

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 $O(n^2)$.

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 $O(n^2)$.

2. Space Complexity

Space Complexity: O(1)

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

program to implement bubble sort in Java

```
public class Bubble {
    static void print (int a[] ) //function to print array elements
  {
    int n = a.length;
    int i;
    for (i = 0; i < n; i++)
    {
      System.out.print(a[i] + " ");
    }
  }
  static void bubbleSort (int a[])    // function to implement bubble sort
  {
    int n = a.length;
    int i, j, temp;
    for (i = 0; i < n; i++)
    {
      for (j = i + 1; j < n; j++)
      {
        if (a[j] < a[i])
        {
          temp = a[i];
          a[i] = a[j];
          a[j] = temp;
        }
      }
```

```
        }
    }
    public static void main(String[] args) {
    int a[] = {35, 10, 31, 11, 26};
    Bubble b1 = new Bubble();
    System.out.println("Before sorting array elements are - ");
    b1.print(a);
    b1.bubbleSort(a);
    System.out.println();
    System.out.println("After sorting array elements are - ");
    b1.print(a);
}
}
```

SELECTION SORT

Selection sort is a simple sorting algorithm that sorts an array by repeatedly finding the minimum element from the unsorted part of the array and putting it at the beginning. The algorithm has a time complexity of $O(n^2)$ in the worst and average case, where n is the number of elements in the array and the space complexity is O(1).

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

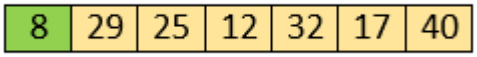
Working of selection algorithm

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 –

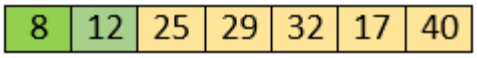
Now, for the first position in the sorted array, the entire array is to be scanned sequentially.
At present, 12 is stored at the first position, after searching the entire array, it is found that 8 is the smallest value.

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

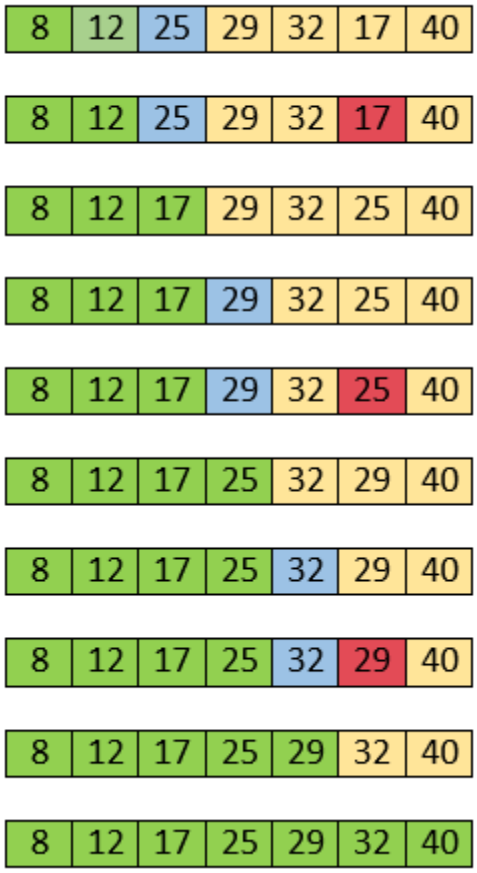


For the second position, where 29 is stored presently, we again sequentially scan the rest of the items of unsorted array. After scanning, we find that 12 is the second lowest element in the array that should be appeared at second position.

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.

Selection sort complexity

1. Time Complexity

Case Time Complexity

1. Best Case :$O(n^2)$
2. Average Case :$O(n^2)$
3. Worst Case :$O(n^2)$

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 $O(n^2)$.

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 $O(n^2)$.

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 $O(n^2)$.

2. Space Complexity

Space Complexity: O(1)

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

Program to implement selection sort in Java.

```
public class Selection
{
   void selection(int a[]) /* function to sort an array with selection sort */
{
   int i, j, small;
   int n = a.length;
   for (i = 0; i < n-1; i++)
   {
      small = i; //minimum element in unsorted array

      for (j = i+1; j < n; j++)
      if (a[j] < a[small])
         small = j;
// Swap the minimum element with the first element
   int temp = a[small];
   a[small] = a[i];
   a[i] = temp;
   }

}
void printArr(int a[]) /* function to print the array */
{
   int i;
   int n = a.length;
   for (i = 0; i < n; i++)
   System.out.print(a[i] + " ");
}
   public static void main(String[] args) {
```

```
        int a[] = { 91, 49, 4, 19, 10, 21 };
        Selection i1 = new Selection();
        System.out.println("\nBefore sorting array elements are - ");
        i1.printArr(a);
        i1.selection(a);
        System.out.println("\nAfter sorting array elements are - ");
        i1.printArr(a);
        System.out.println();
    }   }
```

INSERTION SORT

Insertion sort is a simple sorting algorithm that works by building the final sorted array one item at a time. The algorithm iterates over an array, considering one element at a time, and inserting it into its correct position among the sorted elements.

Although it is simple to use, it is not appropriate for large data sets as the time complexity of insertion sort in the average case and worst case is $O(n^2)$, where n is the number of items. Insertion sort is less efficient than the other sorting algorithms like heap sort, quick sort, merge sort, etc.

Insertion sort has various advantages such as -

- Simple implementation
- Efficient for small data sets
- Adaptive, i.e., it is appropriate for data sets that are already substantially sorted.

Algorithm

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.

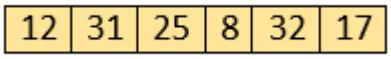
Step 5 : Insert the value.

Step 6 : Repeat until the array is sorted.

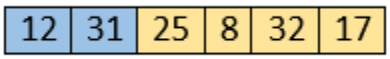
Working of Insertion sort Algorithm

To understand the working of the insertion sort algorithm, let's take an unsorted array.

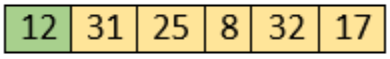
Let the elements of array are -



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

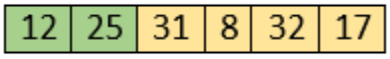


Here, 31 is greater than 12. That means both elements are already in ascending order. So, for now, 12 is stored in a sorted sub-array.

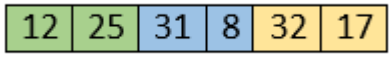


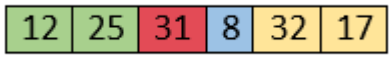
Now, move to the next two elements and compare them.

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.

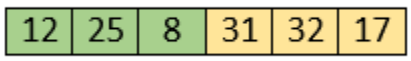


Now, two elements in the sorted array are 12 and 25. Move forward to the next elements that are 31 and 8.

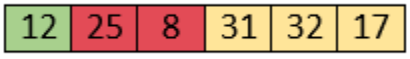




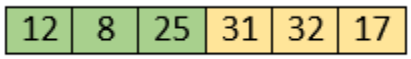
Both 31 and 8 are not sorted. So, swap them.



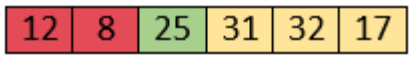
After swapping, elements 25 and 8 are unsorted.



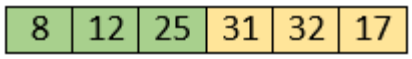
So, swap them.



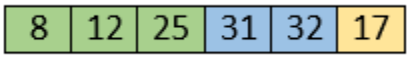
Now, elements 12 and 8 are unsorted.



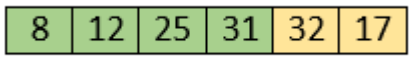
So, swap them too.



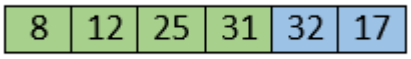
Now, the sorted array has three items that are 8, 12 and 25. Move to the next items that are 31 and 32.



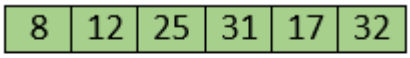
Hence, they are already sorted. Now, the sorted array includes 8, 12, 25 and 31.



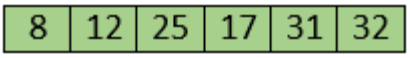
Move to the next elements that are 32 and 17.



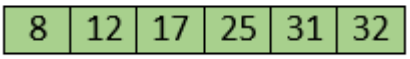
17 is smaller than 32. So, swap them.



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

1. 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).

2. Space Complexity

Space Complexity: O(1)

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

Program to implement insertion sort in Java.

```
public class Insert
{
   void insert(int a[]) /* function to sort an aay with insertion sort */
{
   int i, j, temp;
   int n = a.length;
   for (i = 1; i < n; i++) {
      temp = a[i];
      j = i - 1;
while(j>=0 && temp <= a[j])  /* Move the elements greater than temp to one position ahead from their current position*/
```

```
            {
                a[j+1] = a[j];
                j = j-1;
            }
            a[j+1] = temp;
        }
    }
    void printArr(int a[]) /* function to print the array */
    {
        int i;
        int n = a.length;
        for (i = 0; i < n; i++)
        System.out.print(a[i] + " ");
    }

        public static void main(String[] args) {
        int a[] = { 92, 50, 5, 20, 11, 22 };
        Insert i1 = new Insert();
        System.out.println("\nBefore sorting array elements are - ");
        i1.printArr(a);
        i1.insert(a);
        System.out.println("\n\nAfter sorting array elements are - ");
        i1.printArr(a);
        System.out.println();
        }
}
```

MERGE SORT

Merge sort is the sorting technique that follows the divide and conquer approach. t 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.

Algorithm

MERGE_SORT(arr, beg, end)

if beg < end

set mid = (beg + end)/2

MERGE_SORT(arr, beg, mid)

MERGE_SORT(arr, mid + 1, end)

MERGE (arr, beg, mid, end)

end of if

END MERGE_SORT

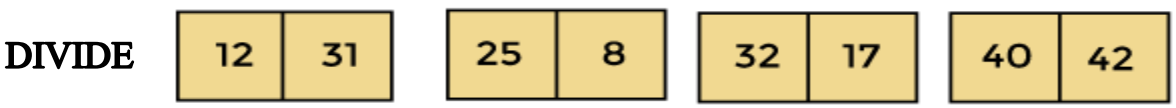
Working of Merge sort Algorithm

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.

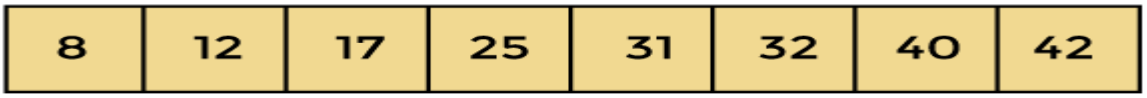
Now, combine them in the same manner they were broken.

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.

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).

2. Space Complexity

Space Complexity: O(n)

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

Program to implement merge sort in Java.

```
class Merge {
/* Function to merge the subarrays of a[] */
void merge(int a[], int beg, int mid, int end)
{
   int i, j, k;
   int n1 = mid - beg + 1;
   int n2 = end - mid;
      int LeftArray[] = new int[n1];     /* temporary Arrays */
      int RightArray[] = new int[n2];

   for (i = 0; i < n1; i++)        /* copy data to temp arrays */
   LeftArray[i] = a[beg + i];
   for (j = 0; j < n2; j++)
   RightArray[j] = a[mid + 1 + j];

   i = 0; /* initial index of first sub-array */
   j = 0; /* initial index of second sub-array */
   k = beg;  /* initial index of merged sub-array */

   while (i < n1 && j < n2)
   {
```

```
        if(LeftArray[i] <= RightArray[j])
        {
            a[k] = LeftArray[i];
            i++;
        }
        else
        {
            a[k] = RightArray[j];
            j++;
        }
        k++;
    }
    while (i<n1)
    {
        a[k] = LeftArray[i];
        i++;
        k++;
    }

    while (j<n2)
    {
        a[k] = RightArray[j];
        j++;
        k++;
    }
}

void mergeSort(int a[], int beg, int end)
{
    if (beg < end)
    {
        int mid = (beg + end) / 2;
        mergeSort(a, beg, mid);
        mergeSort(a, mid + 1, end);
```

```
            merge(a, beg, mid, end);
        }
    }
    void printArray(int a[], int n)   /* Function to print the array */
    {
        int i;
        for (i = 0; i < n; i++)
            System.out.print(a[i] + " ");
    }

    public static void main(String args[])
    {
        int a[] = { 11, 30, 24, 7, 31, 16, 39, 41 };
        int n = a.length;
        Merge m1 = new Merge();
        System.out.println("\nBefore sorting array elements are - ");
        m1.printArray(a, n);
        m1.mergeSort(a, 0, n - 1);
        System.out.println("\nAfter sorting array elements are - ");
        m1.printArray(a, n);
        System.out.println("");
    }
}
```

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.

Algorithm

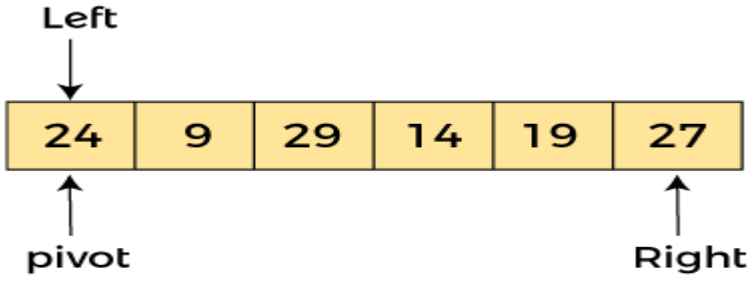
```
QUICKSORT (array A, start, end)
{
 if (start < end)
 {
 p = partition(A, start, end)
QUICKSORT (A, start, p - 1)
QUICKSORT (A, p + 1, end)
}
}
```

Working of Quick Sort Algorithm

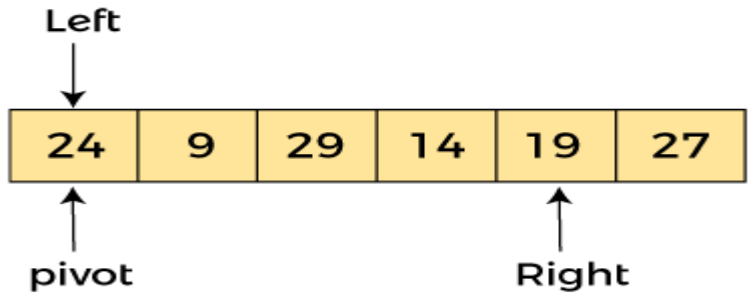
Let the elements of array are -

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.

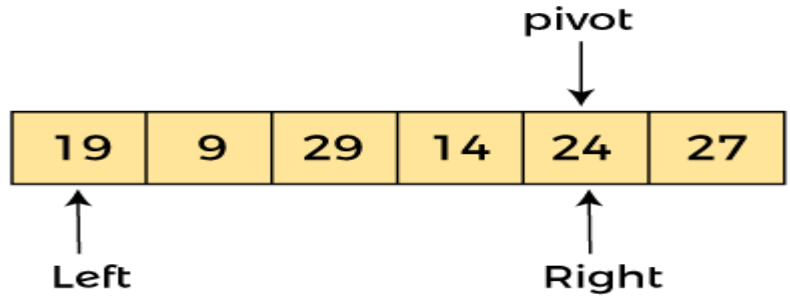


Now, a[pivot] < a[right], so algorithm moves forward one position towards left, i.e. -



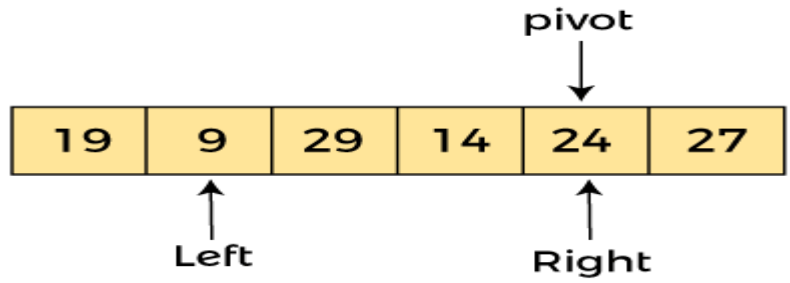
Now, a[left] = 24, a[right] = 19, and a[pivot] = 24.

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

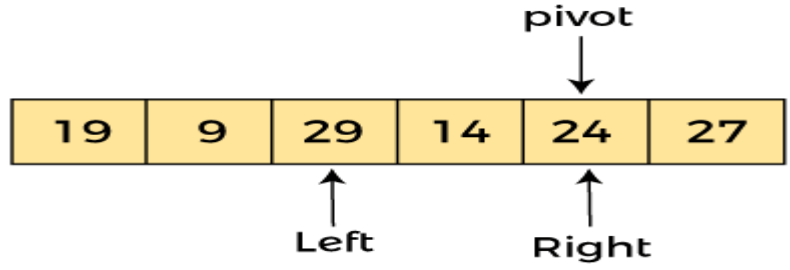


Now, a[left] = 19, a[right] = 24, and a[pivot] = 24. Since, pivot is at right, so algorithm starts from left and moves to right.

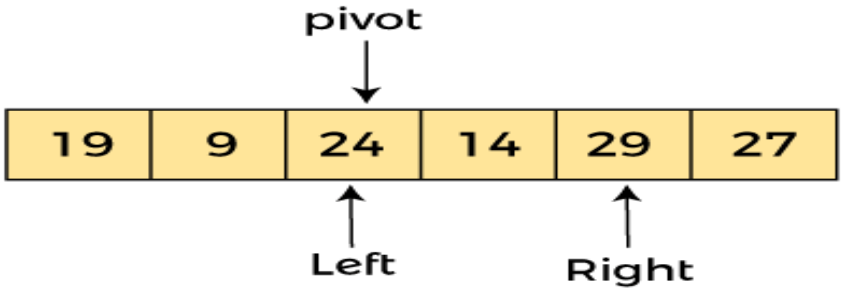
As a[pivot] > a[left], so algorithm moves one position to right as -



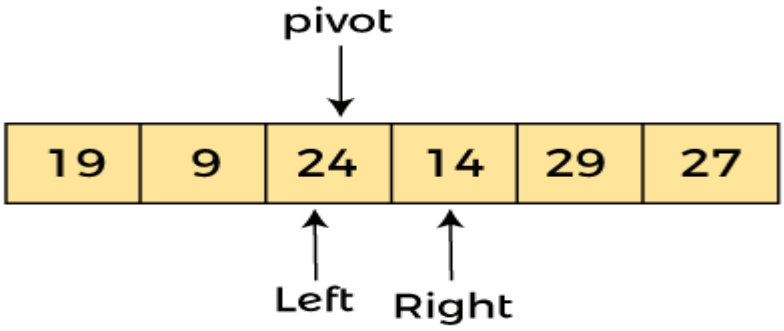
Now a[left] = 9, a[right] = 24, and a[pivot] = 24. As a[pivot] > a[left], so algorithm moves one position to right as -



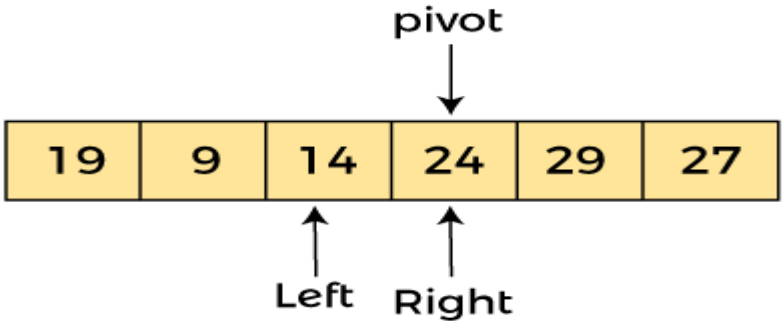
Now, a[left] = 29, a[right] = 24, and a[pivot] = 24. As a[pivot] < a[left], so, swap a[pivot] and a[left], now pivot is at left, i.e. -



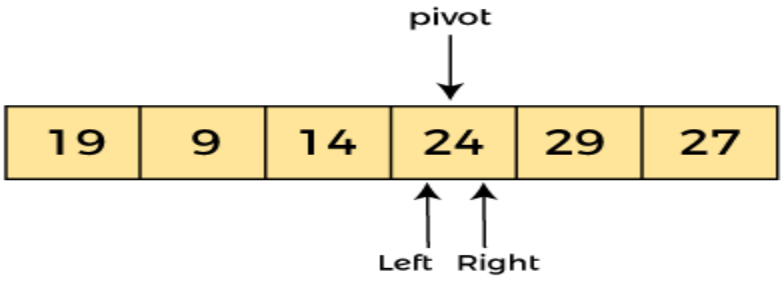
Since, pivot is at left, so algorithm starts from right, and move to left. Now, a[left] = 24, a[right] = 29, and a[pivot] = 24. As a[pivot] < a[right], so algorithm moves one position to left, as -



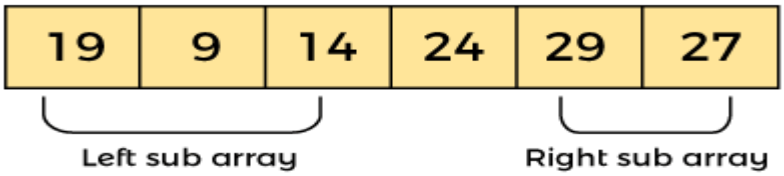
Now, a[pivot] = 24, a[left] = 24, and a[right] = 14. As a[pivot] > a[right], so, swap a[pivot] and a[right], now pivot is at right, i.e. -



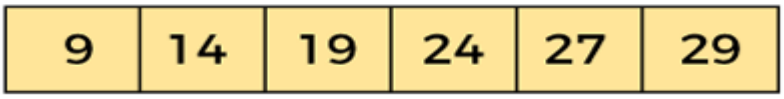
Now, a[pivot] = 24, a[left] = 14, and a[right] = 24. Pivot is at right, so the algorithm starts from left and move to right.



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

1. Time Complexity

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

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 $O(n^2)$.

2. Space Complexity

Space Complexity: O(n*logn)

Program to implement merge sort in Java

```
 public class Quick
{
int partition (int a[], int start, int end)
{
   int pivot = a[end]; // pivot element
   int i = (start - 1);
   for (int j = start; j <= end - 1; j++)
   {
      if (a[j] < pivot)         // If current element is smaller than the pivot
      {
         i++; // increment index of smaller element
         int t = a[i];
         a[i] = a[j];
         a[j] = t;
      }
   }
   int t = a[i+1];
   a[i+1] = a[end];
   a[end] = t;
   return (i + 1);
}
/* function to implement quick sort */
void quick(int a[], int start, int end)
{
   if (start < end)
   {
      int p = partition(a, start, end);  //p is partitioning index
      quick(a, start, p - 1);
      quick(a, p + 1, end);
   }
}
```

```
void printArr(int a[], int n)  /* function to print an array */
{
   int i;
   for (i = 0; i < n; i++)
      System.out.print(a[i] + " ");
}
   public static void main(String[] args) {
   int a[] = { 13, 18, 27, 2, 19, 25 };
   int n = a.length;
   System.out.println("\nBefore sorting array elements are - ");
   Quick q1 = new Quick();
   q1.printArr(a, n);
   q1.quick(a, 0, n - 1);
   System.out.println("\nAfter sorting array elements are - ");
   q1.printArr(a, n);
   System.out.println();
   }
}
```