Geeks Man

Algorithms

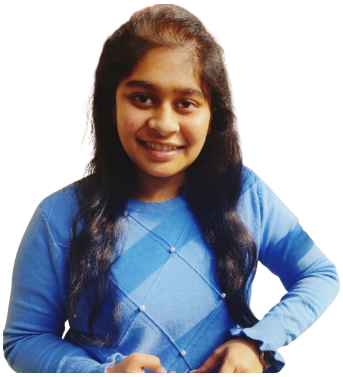
Lesson 6

Trilok Kaushik

Founder of Geeksman

Aditya Aggarwal

Aditya Gupta



Suhani Mittal

Team Coordinators

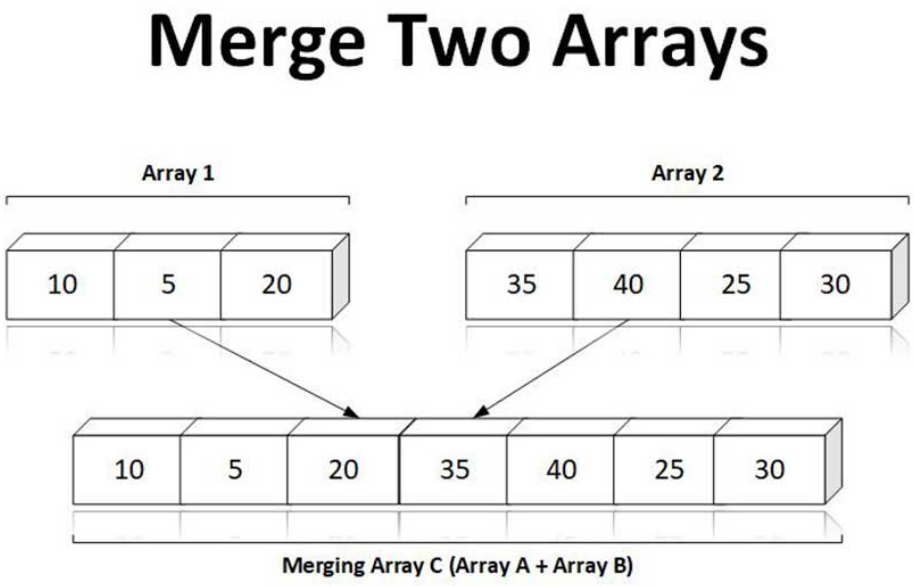
Sorting (Part=II)

4. Merge Sort

Characteristics :

1. Better than Insertion and Quick Sort
2. Works on Divide and conquer Technique.
3. **The merge() function** is used for merging two halves. The merge(arr, l, m, r) is a key process that assumes that arr[l..m] and arr[m+1..r] are sorted and merges the two sorted subarrays into one.
4. More efficient than Quick sort in terms of large input size.Therefore, this algo can work with large input Files.

Merging Arrays :



Code :

```
1.void merge(int arr[], int low, int mid, int high)
```

```
2.{
3.    int n1 = mid +1 - low;
4.    int n2 = end - mid;
5.
6.    // Create temp arrays
7.    int L[n1], R[n2];
8.
9.    // Copy data to temp arrays L[] and R[]
10.     for(int i = 0; i < n1; i++)
11.         L[i] = arr[l + i];
12.     for(int j = 0; j < n2; j++)
13.         R[j] = arr[m + 1 + j];
14.
15.      // copying the elements
16.     int i=0,j=0;
17. for (int k=low; k<high && i<n1 && j<n2; k++)
18.     {
19.         if (L[i] <= R[j])
20.         {
21.             arr[k] = L[i];
22.             i++;
23.         }
24.         else
25.         {
26.             arr[k] = R[j];
27.             j++;
28.         }
29.     }
30.
31.     // Copy the remaining elements of
```

```
    // L[], if there are any
    while (i < n1)
    {
        arr[k] = L[i];
        i++;
        k++;
    }

    // Copy the remaining elements of
    // R[], if there are any
    while (j < n2)
    {
        arr[k] = R[j];
        j++;
        k++;
    }
}
```

Operations performed in merging :

1. Copying = n
2. Comparison = n

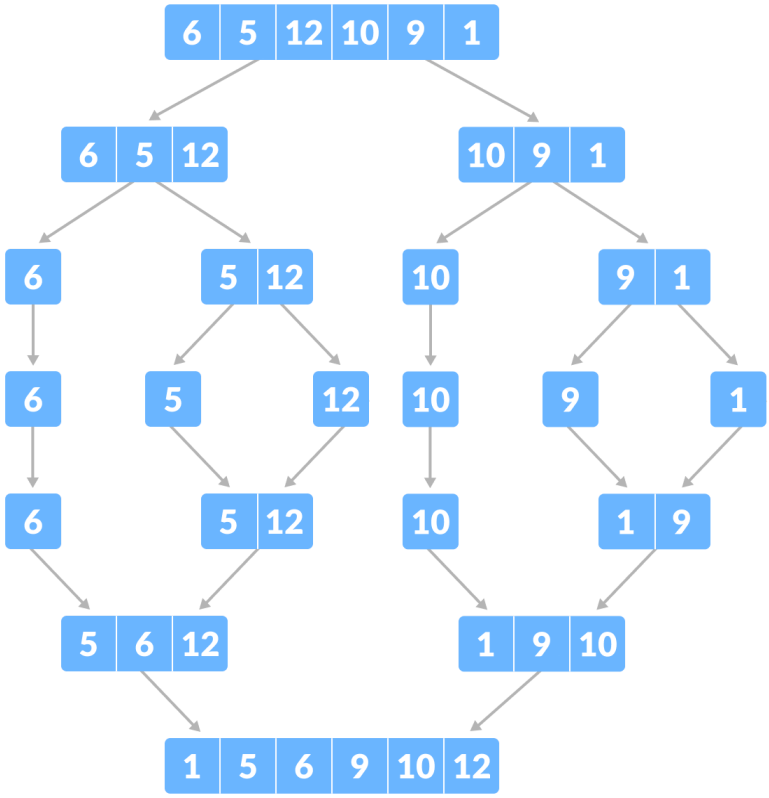
   Therefore, total time= O(n+n) = O(n)

   Space = O(n+n)=O(n)
3. If one sorted array size = n , other sorted array size= m

   Then -> Time = O(n+m)

   Space = O(n+m)
4. To perform merging we need extra space.of O(n)

Merge Sort :



Code:

```
1.void mergeSort(int arr[], int beg, int end)
2.{
3.    if (beg < end)
4.    {
5.
6.        // Same as (beg+end)/2, but avoids
7.        // overflow for large l and h
8.        int mid = beg + (end - beg) / 2;
9.
10.         // Sort first and second halves
11.         mergeSort(arr, beg, mid);
12.         mergeSort(arr, mid + 1, end);
13.
```

```
14.         merge(arr, beg, mid, end);
15.     }
16. }
```

Time Complexity :

T(n) = 2 T(n/2) + O(n)
Here, $T(n) = aT(n/b) + n^k \log^p n$
Therefore a=2, b=2, k=1 where $a=b^k$

-> T(n) = O(nlogn)
-> Best Case = O(nlogn)
-> Worst case = O(nlogn)

Space Complexity :

Stack Space = O(logn) (mergeSort)
Creating array = O(n) (merge)

Total Space = O(n+logn) = O(n)

5. Quick Sort

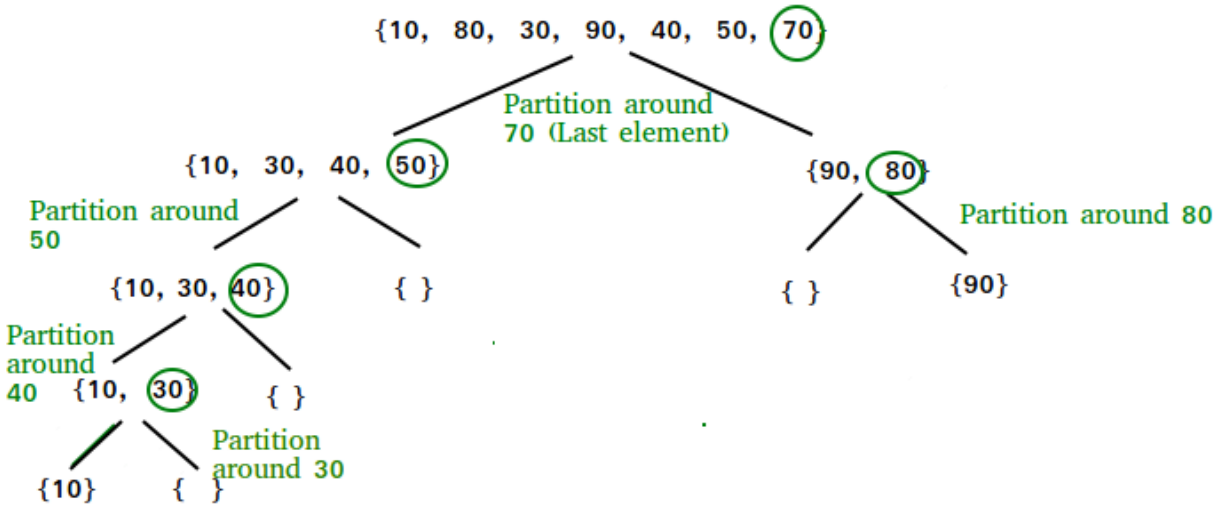
Characteristics:

1. Works on divide and conquer technique
2. The key process in quickSort is **partition().** Target of partitions is, given an array and an element x of array as pivot, put x at its correct position in sorted array and put all smaller elements (smaller than x) before x, and put all greater elements (greater than x) after x. All this should be done in linear time.
3. It picks an element as pivot and partitions the given array around the picked pivot. There are many different versions of quickSort that pick pivot in different ways.

- ➔ Always pick first element as pivot.
- ➔ Always pick last element as pivot
- ➔ Pick a random element as pivot.
- ➔ Pick median as pivot.

4. For Smaller Inputs : Quick Sort is better than Merge Sort

Partition :



Algorithm :

1. Choose the pivot element where you want to start the partition.
2. There would be 2 pointers for keeping track on smaller and larger elements as compared to the pivot element.
3. Increment j until it reaches to the pivot element
4. Increment i only if the element at j is smaller than the pivot element.
5. After incrementing i swap the i and j elements

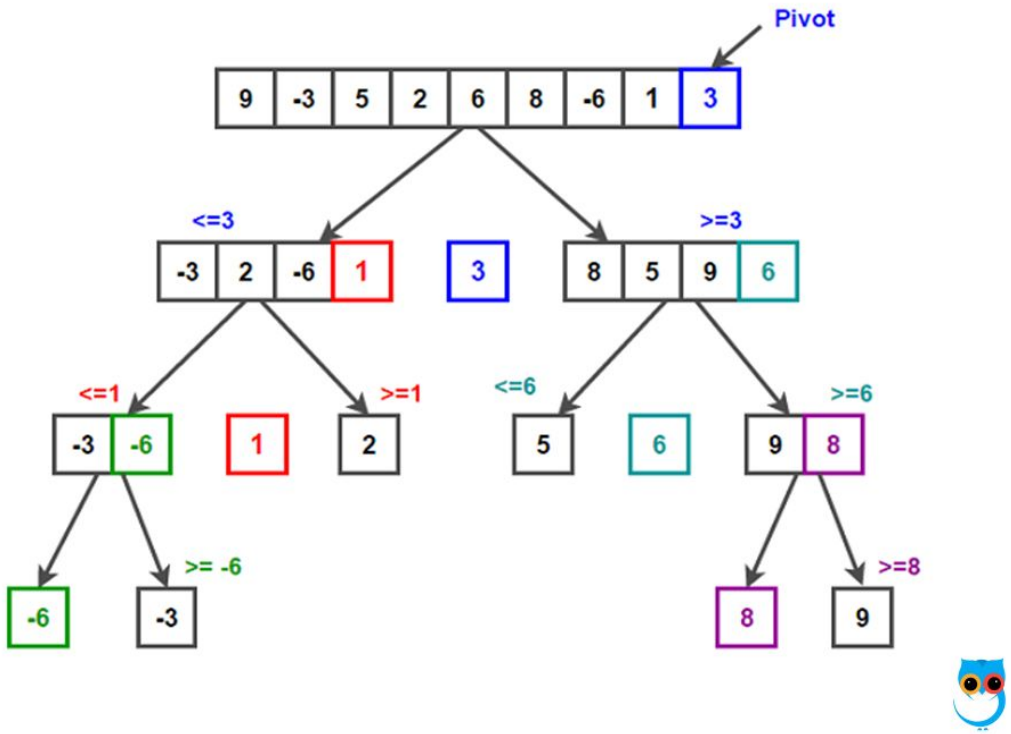
Code :

```
1. void swap(int* a, int* b)
2. {
```

```
3.    int t = *a;
4.    *a = *b;
5.    *b = t;
6. }
7.
8. /* This function takes last element as pivot, places
9. the pivot element at its correct position in sorted
10.  array, and places all smaller (smaller than pivot)
11.  to left of pivot and all greater elements to right
12.  of pivot */
13.  int partition (int arr[], int low, int high)
14.  {
15.     int pivot = arr[high]; // pivot
16.     int i = (low - 1); // Index of smaller element
17.
18.     for (int j = low; j < high; j++)
19.     {
20.         // If current element is smaller than the pivot
21.         if (arr[j] < pivot)
22.         {
23.             i++; // increment index of smaller element
24.             swap(&arr[i], &arr[j]);
25.         }
26.     }
27.     swap(&arr[i + 1], &arr[high]);
28.     return (i + 1);
29.  }
```

1. Partition :- Putting one element in its correct position
   Time Complexity **=** O(n)
   Space Complexity = O(1)

QuickSort :



Code:

```
30.
31.  void quickSort(int arr[], int low, int high)
32.  {
33.      if (low < high)
34.      {
35.          /* pi is partitioning index, arr[p] is now
36.           at right place */
37.          int pi = partition(arr, low, high);
38.
39.          // Separately sort elements before
40.          // partition and after partition
41.          quickSort(arr, low, pi - 1);
42.          quickSort(arr, pi + 1, high);
43.      }
```

```
44.   }
```

Time Complexity :

1. Best case : If partition is at the middle
    i. T(n) = 2 T(n/2) + O(n)
    T(n) = O(nlogn)

2. Worst Case : If partition is unbalanced (array is sorted)
    i. T(n) = 2 T(n-1) + O(n)
    T(n) = $O(n^2)$

Space Complexity :

1. Best Case : O(logn)
    a. If partition is at the middle then no. of levels of tree is minimum.

2. Worst Case : O(n)
    a. If partition is unbalanced (array is sorted) then no. of levels of tree increases to n.

In Merge Sort - Partition is done in the middle point.
In Quick Sort - Partition is done at any point.