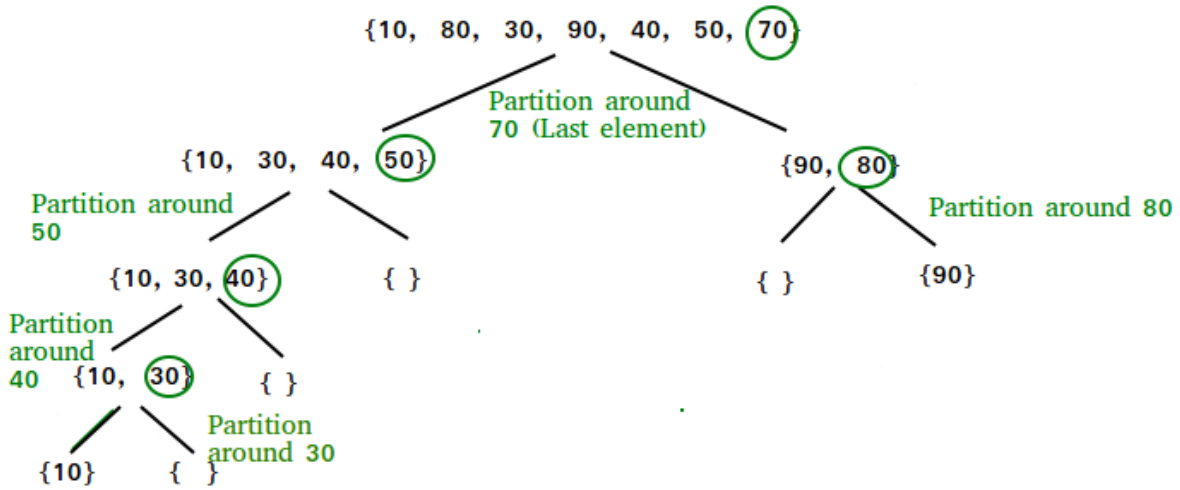
- *Create an empty sorted (or result) list*
- *Traverse the given list, do following for every node.*
    - *Insert current node in sorted way in sorted or result list.*
- *Change head of given linked list to head of sorted (or result) list.*

Quick Sort:

QuickSort is a Divide and Conquer algorithm. It picks an element as a 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 the first element as a pivot.
- Always pick the last element as a pivot (implemented below)
- Pick a random element as a pivot.
- Pick median as the pivot.

The key process in **quickSort** is a partition(). The target of partitions is, given an array and an element x of an array as the pivot, put x at its correct position in a 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.



Partition Algorithm:

There can be many ways to do partition, following pseudo-code adopts the method given in the CLRS book. The logic is simple, we start from the leftmost element and keep track of the index of smaller (or equal to) elements as i. While traversing, if we find a smaller element, we swap the current element with arr[i]. Otherwise, we ignore the current element.

Pseudo Code for recursive QuickSort function:

```
/* low  –> Starting index,  high  –> Ending index */

quickSort(arr[], low, high) {

    if (low < high) {

        /* pi is partitioning index, arr[pi] is now at right place */

        pi = partition(arr, low, high);

        quickSort(arr, low, pi – 1);  // Before pi

        quickSort(arr, pi + 1, high); // After pi
```

```
    }

}
```

Pseudo code for partition()

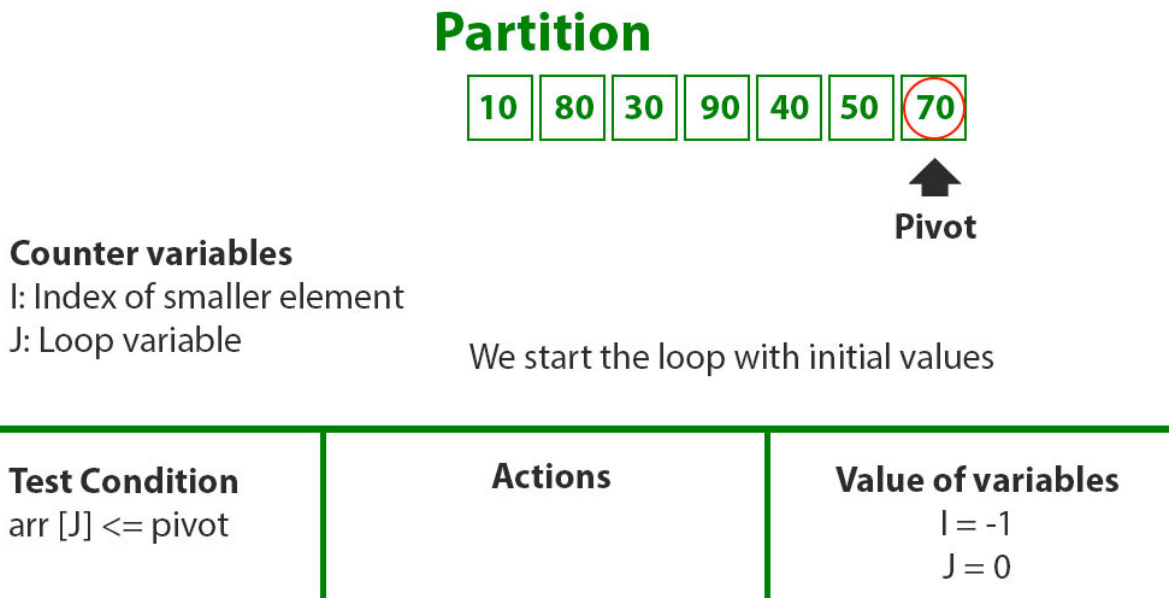
```
/* This function takes last element as pivot, places the pivot element at its correct position in sorted array, and places all smaller (smaller than pivot) to left of pivot and all greater elements to right of pivot */

partition (arr[], low, high)

{

    // pivot (Element to be placed at right position)

    pivot = arr[high];

    i = (low – 1)  // Index of smaller element and indicates the

    // right position of pivot found so far

    for (j = low; j <= high- 1; j++){

        // If current element is smaller than the pivot

        if (arr[j] < pivot){

            i++;    // increment index of smaller element
```

```
            swap arr[i] and arr[j]

        }

    }

    swap arr[i + 1] and arr[high])

    return (i + 1)

}
```

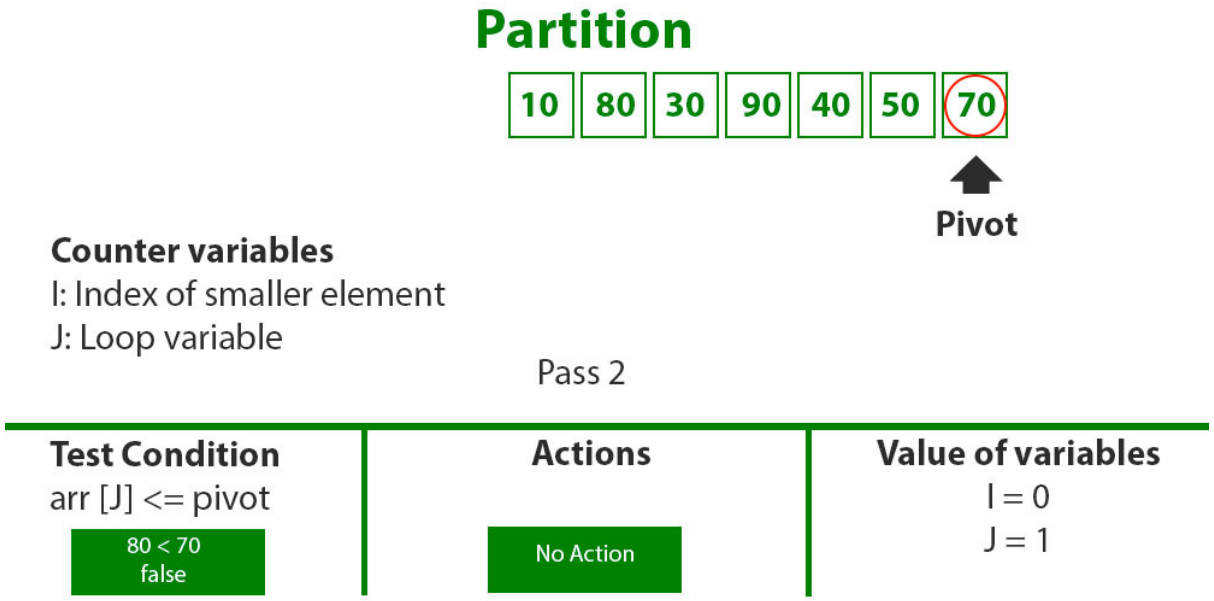
Illustration of partition() :

Consider: arr[] = {10, 80, 30, 90, 40, 50, 70}

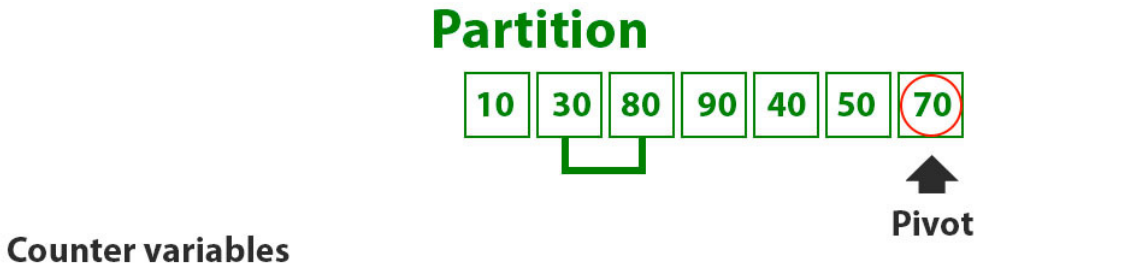
- *Indexes: 0 1 2 3 4 5 6*
- *low = 0, high = 6, pivot = arr[h] = 70*
- *Initialize index of smaller element,* ***i = -1***



- *Traverse elements from j = low to high-1*
    - ***j = 0**: Since arr[j] <= pivot, do i++ and swap(arr[i], arr[j])*
    - ***i = 0***
- *arr[] = {10, 80, 30, 90, 40, 50, 70} // No change as i and j are same*
- ***j = 1**: Since arr[j] > pivot, do nothing*



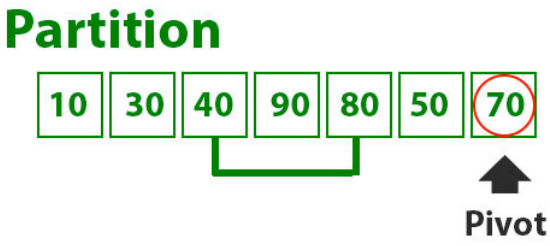
- ***j = 2** : Since arr[j] <= pivot, do i++ and swap(arr[i], arr[j])*
- ***i = 1***
- *arr[] = {10, 30, 80, 90, 40, 50, 70} // We swap 80 and 30*



Counter variables
I: Index of smaller element
J: Loop variable

Test Condition	Actions	Value of variables
arr [J] <= pivot 30 < 70 true	i++ Swap(arr[i],arr[j])	I = 1 J = 2

- ***j = 3*** *: Since arr[j] > pivot, do nothing // No change in i and arr[]*
- ***j = 4*** *: Since arr[j] <= pivot, do i++ and swap(arr[i], arr[j])*
- ***i = 2***
- *arr[] = {10, 30, 40, 90, 80, 50, 70} // 80 and 40 Swapped*

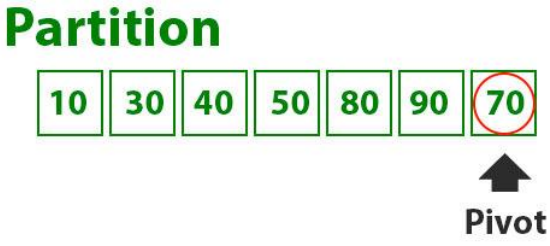


Counter variables
I: Index of smaller element
J: Loop variable

Pass 5

Test Condition	Actions	Value of variables
arr [J] <= pivot 40 < 70 true	i++ Swap(arr[i],arr[j])	I = 2 J = 4

- ***j = 5*** *: Since arr[j] <= pivot, do i++ and swap arr[i] with arr[j]*
- ***i = 3***
- *arr[] = {10, 30, 40, 50, 80, 90, 70} // 90 and 50 Swapped*

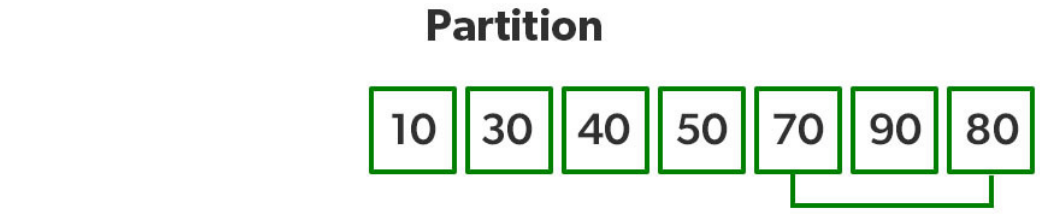


Counter variables
I: Index of smaller element
J: Loop variable

Before Pass 7, J becomes 6
so we come out of the loop

Test Condition	Actions	Value of variables
arr [J] <= pivot		I = 3 J = 6

- *We come out of loop because j is now equal to high-1.*
- ***Finally we place pivot at correct position by swapping arr[i+1] and arr[high] (or pivot)***
- *arr[] = {10, 30, 40, 50, 70, 90, 80} // 80 and 70 Swapped*



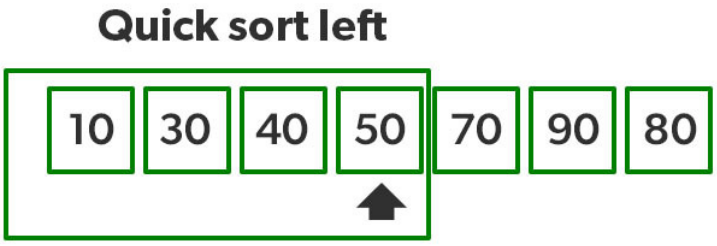
Counter Variable

I : Index of smaller element
J : Loop variable

We know swap arr[i+1] and pivot

I = 3

- *Now 70 is at its correct place. All elements smaller than 70 are before it and all elements greater than 70 are after it.*
- *Since quick sort is a recursive function, we call the partition function again at left and right partitions*

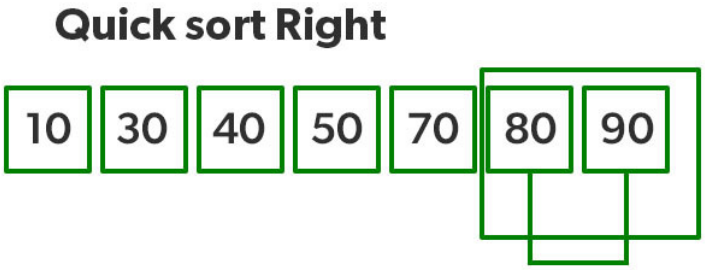


Since quick sort is a recursion function,
wecall the Partition function again

First 50 is the pivot.

As it is already at its correct position
we call the quicksort function again on the left part.

- *Again call function at right part and swap 80 and 90*



80 is the Pivot

80 and 90 are swapped to bring pivot
to correct position

Implementation:
Following are the implementations of QuickSort:

- C++14
- Java
- Python3
- C#
- Javascript

```
/* C++ implementation of QuickSort */
#include <bits/stdc++.h>
using namespace std;

// A utility function to swap two elements
void swap(int* a, int* b)
{
    int t = *a;
    *a = *b;
    *b = t;
}

/* This function takes last element as pivot, places
the pivot element at its correct position in sorted
array, and places all smaller (smaller than pivot)
to left of pivot and all greater elements to right
of pivot */
int partition(int arr[], int low, int high)
{
    int pivot = arr[high]; // pivot
```

```
    int i
        = (low
          - 1); // Index of smaller element and indicates
                // the right position of pivot found so far

    for (int j = low; j <= high - 1; j++) {
        // If current element is smaller than the pivot
        if (arr[j] < pivot) {
            i++; // increment index of smaller element
            swap(&arr[i], &arr[j]);
        }
    }
    swap(&arr[i + 1], &arr[high]);
    return (i + 1);
}

/* The main function that implements QuickSort
arr[] --> Array to be sorted,
low --> Starting index,
high --> Ending index */
```

```
void quickSort(int arr[], int low, int high)
{
    if (low < high) {
        /* pi is partitioning index, arr[p] is now
        at right place */
        int pi = partition(arr, low, high);

        // Separately sort elements before
        // partition and after partition
        quickSort(arr, low, pi - 1);
        quickSort(arr, pi + 1, high);
    }
}

/* Function to print an array */
void printArray(int arr[], int size)
{
    int i;
    for (i = 0; i < size; i++)
        cout << arr[i] << " ";
```

```
    cout << endl;
}

// Driver Code
int main()
{
    int arr[] = { 10, 7, 8, 9, 1, 5 };
    int n = sizeof(arr) / sizeof(arr[0]);
    quickSort(arr, 0, n - 1);
    cout << "Sorted array: \n";
    printArray(arr, n);
    return 0;
}

// This code is contributed by rathbhupendra
```

Output

Sorted array:

1 5 7 8 9 10

Hoare's vs Lomuto Partition

Please note that the above implementation is Lomuto Partition. A more optimized implementation of QuickSort is Hoare's partition which is more efficient than Lomuto's partition scheme because it does three times less swaps on average.

How to pick any element as pivot?

With one minor change to the above code, we can pick any element as pivot. For example, to make the first element as pivot, we can simply swap the first and last elements and then use the same code. Same thing can be done to pick any random element as a pivot

Analysis of QuickSort

Time taken by QuickSort, in general, can be written as follows.

$$T(n) = T(k) + T(n-k-1) + (n)$$

The first two terms are for two recursive calls, the last term is for the partition process. k is the number of elements that are smaller than the pivot.

The time taken by QuickSort depends upon the input array and partition strategy. Following are three cases.

Worst Case:

The worst case occurs when the partition process always picks the greatest or smallest element as the pivot. If we consider the above partition strategy where the last element is always picked as a pivot, the worst case would occur when the array is already sorted in increasing or decreasing order. Following is recurrence for the worst case.

T(n) = T(0) + T(n-1) + (n)which is equivalent to T(n) = T(n-1) + (n)

The solution to the above recurrence is (n2).

Best Case:

The best case occurs when the partition process always picks the middle element as the pivot. The following is recurrence for the best case.

T(n) = 2T(n/2) + (n)

The solution for the above recurrence is (nLogn). It can be solved using case 2 of Master Theorem.

Average Case:

To do average case analysis, we need to consider all possible permutation of array and calculate time taken by every permutation which doesn't look easy.

We can get an idea of average case by considering the case when partition puts O(n/9) elements in one set and O(9n/10) elements in other set. Following is recurrence for this case.

T(n) = T(n/9) + T(9n/10) + (n)

The solution of above recurrence is also O(nLogn):

Although the worst case time complexity of QuickSort is O(n2) which is more than many other sorting algorithms like Merge Sort and Heap Sort, QuickSort is faster in practice, because its inner loop can be efficiently implemented on most architectures, and in most real-world data. QuickSort can be implemented in different ways by changing the choice of pivot, so that the worst case rarely occurs for a given type of data. However, merge sort is generally considered better when data is huge and stored in external storage.

Is QuickSort stable?

The default implementation is not stable. However any sorting algorithm can be made stable by considering indexes as comparison parameter.

Is QuickSort In-place?

As per the broad definition of in-place algorithm it qualifies as an in-place sorting algorithm as it uses extra space only for storing recursive function calls but not for manipulating the input.

What is 3-Way QuickSort?

In simple QuickSort algorithm, we select an element as pivot, partition the array around pivot and recur for subarrays on left and right of pivot.

Consider an array which has many redundant elements. For example, {1, 4, 2, 4, 2, 4, 1, 2, 4, 1, 2, 2, 2, 2, 4, 1, 4, 4, 4}. If 4 is picked as pivot in Simple QuickSort, we fix only one 4 and recursively process remaining occurrences. In 3 Way QuickSort, an array arr[l..r] is divided in 3 parts:

- arr[l..i] elements less than pivot.
- arr[i+1..j-1] elements equal to pivot.
- arr[j..r] elements greater than pivot.

See this for implementation.

How to implement QuickSort for Linked Lists?

QuickSort on Singly Linked List

QuickSort on Doubly Linked List

Can we implement QuickSort Iteratively?

Yes, please refer Iterative Quick Sort.

Why Quick Sort is preferred over MergeSort for sorting Arrays ?

Quick Sort in its general form is an in-place sort (i.e. it doesn't require any extra storage) whereas merge sort requires O(N)

extra storage, N denoting the array size which may be quite expensive. Allocating and de-allocating the extra space used for merge sort increases the running time of the algorithm. Comparing average complexity we find that both type of sorts have O(NlogN) average complexity but the constants differ. For arrays, merge sort loses due to the use of extra O(N) storage space.

Most practical implementations of Quick Sort use randomized version. The randomized version has expected time complexity of O(nLogn). The worst case is possible in randomized version also, but worst case doesn't occur for a particular pattern (like sorted array) and randomized Quick Sort works well in practice.

Quick Sort is also a cache friendly sorting algorithm as it has good locality of reference when used for arrays.

Quick Sort is also tail recursive, therefore tail call optimizations is done.

Why MergeSort is preferred over QuickSort for Linked Lists ?

In case of linked lists the case is different mainly due to difference in memory allocation of arrays and linked lists. Unlike arrays, linked list nodes may not be adjacent in memory. Unlike array, in linked list, we can insert items in the middle in O(1) extra space and O(1) time. Therefore merge operation of merge sort can be implemented without extra space for linked lists.

In arrays, we can do random access as elements are continuous in memory. Let us say we have an integer (4-byte) array A and let the address of A[0] be x then to access A[i], we can directly access the memory at (x + i*4). Unlike arrays, we can not do random access in linked list. Quick Sort requires a lot of this kind of access. In linked list to access i'th index, we have to travel each and every node from the head to i'th node as we

don’t have continuous block of memory. Therefore, the overhead increases for quick sort. Merge sort accesses data sequentially and the need of random access is low.

Merge Sort:

The **Merge Sort** algorithm is a sorting algorithm that is based on the **Divide and Conquer** paradigm. In this algorithm, the array is initially divided into two equal halves and then they are combined in a sorted manner.

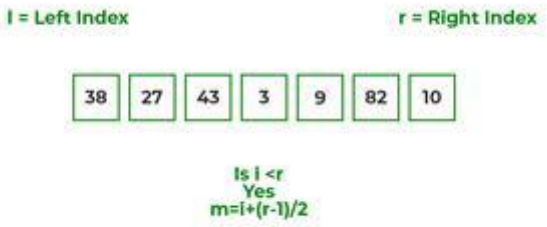
Merge Sort Working Process:

Think of it as a recursive algorithm continuously splits the array in half until it cannot be further divided. This means that if the array becomes empty or has only one element left, the dividing will stop, i.e. it is the base case to stop the recursion. If the array has multiple elements, split the array into halves and recursively invoke the merge sort on each of the halves. Finally, when both halves are sorted, the merge operation is applied. Merge operation is the process of taking two smaller sorted arrays and combining them to eventually make a larger one.

Illustration:

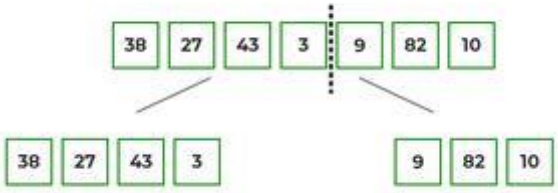
To know the functioning of merge sort, lets consider an array arr[] = {38, 27, 43, 3, 9, 82, 10}

- *At first, check if the left index of array is less than the right index, if yes then calculate its mid point*

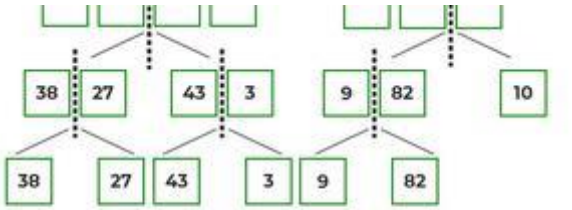


- *Now, as we already know that merge sort first divides the whole array iteratively into equal halves, unless the atomic values are achieved.*
- *Here, we see that an array of 7 items is divided into two arrays of size 4 and 3 respectively.*

- *Now, again find that is left index is less than the right index for both arrays, if found yes, then again calculate mid points for both the arrays.*

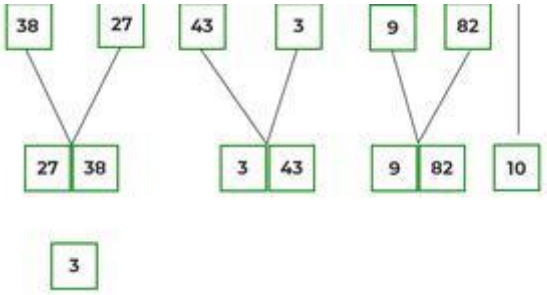


- *Now, further divide these two arrays into further halves, until the atomic units of the array is reached and further division is not possible.*

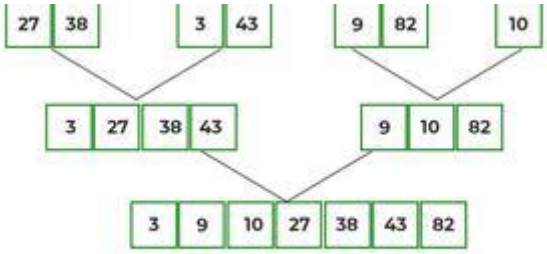


After dividing the array into smallest units merging starts, based on comparison of elements.

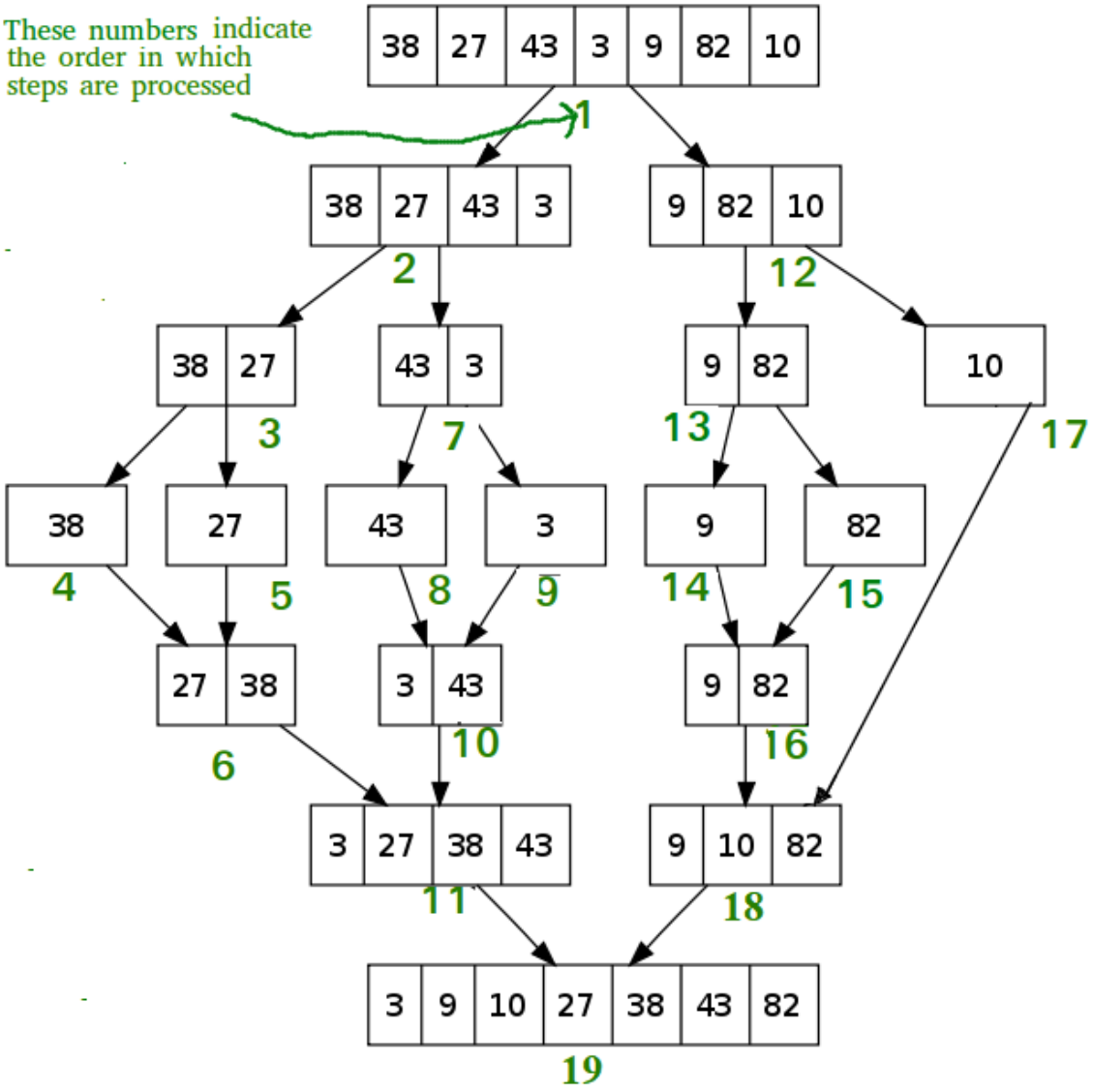
- *After dividing the array into smallest units, start merging the elements again based on comparison of size of elements*
- *Firstly, compare the element for each list and then combine them into another list in a sorted manner.*



- *After the final merging, the list looks like this:*



The following diagram shows the complete merge sort process for an example array {38, 27, 43, 3, 9, 82, 10}.
If we take a closer look at the diagram, we can see that the array is recursively divided into two halves till the size becomes 1. Once the size becomes 1, the merge processes come into action and start merging arrays back till the complete array is merged.



Recursive steps of merge sort

Algorithm:

step 1: start

step 2: declare array and left, right, mid variable

step 3: perform merge function.

if left > right

return

mid= (left+right)/2

mergesort(array, left, mid)

mergesort(array, mid+1, right)

merge(array, left, mid, right)

step 4: Stop

Follow the steps below the solve the problem:
MergeSort(arr[], l, r)
If r > l

- Find the middle point to divide the array into two halves:
    - middle m = l + (r – l)/2
- Call mergeSort for first half:
    - Call mergeSort(arr, l, m)
- Call mergeSort for second half:
    - Call mergeSort(arr, m + 1, r)
- Merge the two halves sorted in steps 2 and 3:
    - Call merge(arr, l, m, r)

Below is the implementation of the above approach:

- C++
- C
- Java

- Python3
- C#
- Javascript
- PHP

```
// C++ program for Merge Sort
#include <iostream>
using namespace std;

// Merges two subarrays of array[].
// First subarray is arr[begin..mid]
// Second subarray is arr[mid+1..end]
void merge(int array[], int const left, int const mid,
           int const right)
{
    auto const subArrayOne = mid - left + 1;
    auto const subArrayTwo = right - mid;

    // Create temp arrays
    auto *leftArray = new int[subArrayOne],
         *rightArray = new int[subArrayTwo];

    // Copy data to temp arrays leftArray[] and rightArray[]
    for (auto i = 0; i < subArrayOne; i++)
        leftArray[i] = array[left + i];
```

```
for (auto j = 0; j < subArrayTwo; j++)
    rightArray[j] = array[mid + 1 + j];

auto indexOfSubArrayOne
    = 0, // Initial index of first sub-array
    indexOfSubArrayTwo
    = 0; // Initial index of second sub-array
int indexOfMergedArray
    = left; // Initial index of merged array

// Merge the temp arrays back into array[left..right]
while (indexOfSubArrayOne < subArrayOne
       && indexOfSubArrayTwo < subArrayTwo) {
    if (leftArray[indexOfSubArrayOne]
        <= rightArray[indexOfSubArrayTwo]) {
        array[indexOfMergedArray]
            = leftArray[indexOfSubArrayOne];
        indexOfSubArrayOne++;
    }
    else {
```

```
            array[indexOfMergedArray]
                = rightArray[indexOfSubArrayTwo];
            indexOfSubArrayTwo++;
        }
        indexOfMergedArray++;
    }
    // Copy the remaining elements of
    // left[], if there are any
    while (indexOfSubArrayOne < subArrayOne) {
        array[indexOfMergedArray]
            = leftArray[indexOfSubArrayOne];
        indexOfSubArrayOne++;
        indexOfMergedArray++;
    }
    // Copy the remaining elements of
    // right[], if there are any
    while (indexOfSubArrayTwo < subArrayTwo) {
        array[indexOfMergedArray]
            = rightArray[indexOfSubArrayTwo];
        indexOfSubArrayTwo++;
```

```
        indexOfMergedArray++;
    }
    delete[] leftArray;
    delete[] rightArray;
}

// begin is for left index and end is
// right index of the sub-array
// of arr to be sorted */
void mergeSort(int array[], int const begin, int const end)
{
    if (begin >= end)
        return; // Returns recursively

    auto mid = begin + (end - begin) / 2;
    mergeSort(array, begin, mid);
    mergeSort(array, mid + 1, end);
    merge(array, begin, mid, end);
}
```

```
// UTILITY FUNCTIONS
// Function to print an array
void printArray(int A[], int size)
{
    for (auto i = 0; i < size; i++)
        cout << A[i] << " ";
}

// Driver code
int main()
{
    int arr[] = { 12, 11, 13, 5, 6, 7 };
    auto arr_size = sizeof(arr) / sizeof(arr[0]);

    cout << "Given array is \n";
    printArray(arr, arr_size);

    mergeSort(arr, 0, arr_size - 1);

    cout << "\nSorted array is \n";
```

```
    printArray(arr, arr_size);
    return 0;
}

// This code is contributed by Mayank Tyagi
// This code was revised by Joshua Estes
```

Output

Given array is

12 11 13 5 6 7

Sorted array is

5 6 7 11 12 13

Time Complexity: O(N log(N)), Sorting arrays on different machines. Merge Sort is a recursive algorithm and time complexity can be expressed as following recurrence relation.

$T(n) = 2T(n/2) + \theta(n)$

The above recurrence can be solved either using the Recurrence Tree method or the Master method. It falls in case II of the Master Method and the solution of the recurrence is θ(Nlog(N)). The time complexity of Merge Sort isθ(Nlog(N)) in all 3 cases (worst, average, and best) as merge sort always divides the array into two halves and takes linear time to merge two halves.

Auxiliary Space: O(n), In merge sort all elements are copied into an auxiliary array. So N auxiliary space is required for merge sort.

Is Merge sort In Place?

No, In merge sort the merging step requires extra space to store the elements.

Is Merge sort Stable?

Yes, merge sort is stable.

How can we make Merge sort more efficient?

Merge sort can be made more efficient by replacing recursive calls with Insertion sort for smaller array sizes, where the size of the remaining array is less or equal to 43 as the number of operations required to sort an array of max size 43 will be less

in Insertion sort as compared to the number of operations required in Merge sort.

Analysis of Merge Sort:

A merge sort consists of several passes over the input. The first pass merges segments of size 1, the second merges segments of size 2, and the pass merges segments of size *2i-1*. Thus, the total number of passes is [*log2n*]. As merge showed, we can merge two sorted segments in linear time, which means that each pass takes *O(n)* time. Since there are [*log2n*] passes, the total computing time is ***O(nlogn)***.

Applications of Merge Sort:

- Merge Sort is useful for sorting linked lists in O(N log N) time. In the case of linked lists, the case is different mainly due to the difference in memory allocation of arrays and linked lists. Unlike arrays, linked list nodes may not be adjacent in memory. Unlike an array, in the linked list, we can insert items in the middle in O(1) extra space and O(1) time. Therefore, the merge operation of merge sort can be implemented without extra space for linked lists.

  In arrays, we can do random access as elements are contiguous in memory. Let us say we have an integer (4-byte) array A and let the address of A[0] be x then to access A[i], we can directly access the memory at (x + i*4). Unlike arrays, we can not do random access in the linked list. Quick Sort requires a lot of this kind of access. In a linked list to access i'th index, we have to travel each and every node from the head to i'th node as we don't have a contiguous block of memory. Therefore, the overhead increases for quicksort. Merge sort accesses data sequentially and the need of random access is low.
- Inversion Count Problem