geeksforgeeks.org

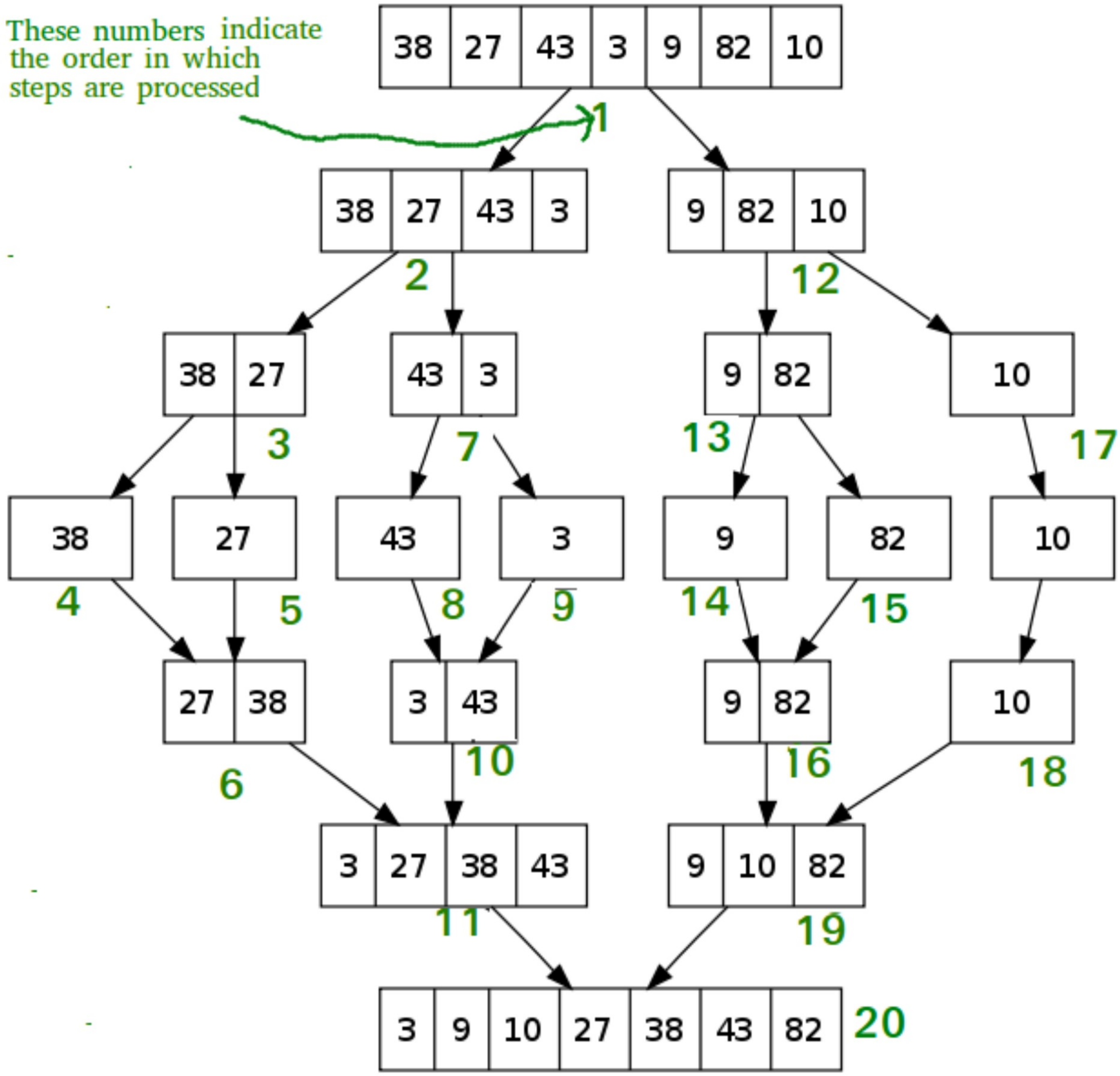
Merge Sort - GeeksforGeeks

7-9 minutes

Like QuickSort, Merge Sort is a Divide and Conquer algorithm. It divides input array in two halves, calls itself for the two halves and then merges the two sorted halves. **The merge() function** is used for merging two halves. The merge(arr, l, m, r) is key process that assumes that arr[l..m] and arr[m+1..r] are sorted and merges the two sorted sub-arrays into one. See following C implementation for details.

```
MergeSort(arr[], l,  r)
If r > l
     1. Find the middle point to divide the array
into two halves:
             middle m = (l+r)/2
     2. Call mergeSort for first half:
             Call mergeSort(arr, l, m)
     3. Call mergeSort for second half:
             Call mergeSort(arr, m+1, r)
     4. Merge the two halves sorted in step 2 and
3:
             Call merge(arr, l, m, r)
```

The following diagram from wikipedia 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 in two halves till the size becomes 1. Once the size becomes 1, the merge processes comes into action and starts merging arrays back till the complete array is merged.



- C/C++
- Java
- Python

C/C++

```
#include<stdlib.h>

#include<stdio.h>

void merge(int arr[], int l, int m, int r)
```

```
{
    int i, j, k;
    int n1 = m - l + 1;
    int n2 =  r - m;
    int L[n1], R[n2];
    for (i = 0; i < n1; i++)
        L[i] = arr[l + i];
    for (j = 0; j < n2; j++)
        R[j] = arr[m + 1+ j];
    i = 0;
    j = 0;
    k = l;
    while (i < n1 && j < n2)
    {
        if (L[i] <= R[j])
        {
            arr[k] = L[i];
            i++;
        }
        else
        {
            arr[k] = R[j];
            j++;
        }
```

```
        k++;
    }
    while (i < n1)
    {
        arr[k] = L[i];
        i++;
        k++;
    }
    while (j < n2)
    {
        arr[k] = R[j];
        j++;
        k++;
    }
}
void mergeSort(int arr[], int l, int r)
{
    if (l < r)
    {
        int m = l+(r-l)/2;
        mergeSort(arr, l, m);
        mergeSort(arr, m+1, r);
        merge(arr, l, m, r);
    }
```

```
}
void printArray(int A[], int size)
{
    int i;
    for (i=0; i < size; i++)
        printf("%d ", A[i]);
    printf("\n");
}
int main()
{
    int arr[] = {12, 11, 13, 5, 6, 7};
    int arr_size = sizeof(arr)/sizeof(arr[0]);
    printf("Given array is \n");
    printArray(arr, arr_size);
    mergeSort(arr, 0, arr_size - 1);
    printf("\nSorted array is \n");
    printArray(arr, arr_size);
    return 0;
}
```

Java

```
class MergeSort
{
    void merge(int arr[], int l, int m, int r)
```

```
    {
        int n1 = m - l + 1;
        int n2 = r - m;
        int L[] = new int [n1];
        int R[] = new int [n2];
        for (int i=0; i<n1; ++i)
            L[i] = arr[l + i];
        for (int j=0; j<n2; ++j)
            R[j] = arr[m + 1+ j];
        int i = 0, j = 0;
        int k = l;
        while (i < n1 && j < n2)
        {
            if (L[i] <= R[j])
            {
                arr[k] = L[i];
                i++;
            }
            else
            {
                arr[k] = R[j];
                j++;
            }
            k++;
```

```
        }
        while (i < n1)
        {
            arr[k] = L[i];
            i++;
            k++;
        }
        while (j < n2)
        {
            arr[k] = R[j];
            j++;
            k++;
        }
    }
    void sort(int arr[], int l, int r)
    {
        if (l < r)
        {
            int m = (l+r)/2;
            sort(arr, l, m);
            sort(arr , m+1, r);
            merge(arr, l, m, r);
        }
    }
```

```
    static void printArray(int arr[])
    {
        int n = arr.length;
        for (int i=0; i<n; ++i)
            System.out.print(arr[i] + " ");
        System.out.println();
    }
    public static void main(String args[])
    {
        int arr[] = {12, 11, 13, 5, 6, 7};
        System.out.println("Given Array");
        printArray(arr);
        MergeSort ob = new MergeSort();
        ob.sort(arr, 0, arr.length-1);
        System.out.println("\nSorted array");
        printArray(arr);
    }
}
```

Python

```
def merge(arr, l, m, r):
    n1 = m - l + 1
    n2 = r- m
    L = [0] * (n1)
```

```
R = [0] * (n2)

for i in range(0 , n1):

    L[i] = arr[l + i]

for j in range(0 , n2):

    R[j] = arr[m + 1 + j]

i = 0

j = 0

k = l

while i < n1 and j < n2 :

    if L[i] <= R[j]:

        arr[k] = L[i]

        i += 1

    else:

        arr[k] = R[j]

        j += 1

    k += 1

while i < n1:

    arr[k] = L[i]

    i += 1

    k += 1

while j < n2:

    arr[k] = R[j]

    j += 1

    k += 1
```

```
def mergeSort(arr,l,r):
    if l < r:
        m = (l+(r-1))/2
        mergeSort(arr, l, m)
        mergeSort(arr, m+1, r)
        merge(arr, l, m, r)
arr = [12, 11, 13, 5, 6, 7]
n = len(arr)
print ("Given array is")
for i in range(n):
    print ("%d" %arr[i]),
mergeSort(arr,0,n-1)
print ("\n\nSorted array is")
for i in range(n):
    print ("%d" %arr[i]),
```

Output:

```
Given array is
12 11 13 5 6 7

Sorted array is
5 6 7 11 12 13
```

Time Complexity: 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 Recurrence Tree method or Master method. It falls in case II of Master Method and

solution of the recurrence is $\Theta(nLogn)$.
Time complexity of Merge Sort is $\Theta(nLogn)$ in all 3 cases (worst, average and best) as merge sort always divides the array in two halves and take linear time to merge two halves.

Auxiliary Space: O(n)

Algorithmic Paradigm: Divide and Conquer

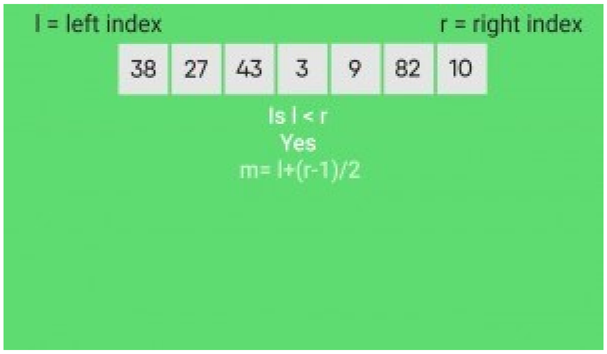
Sorting In Place: No in a typical implementation

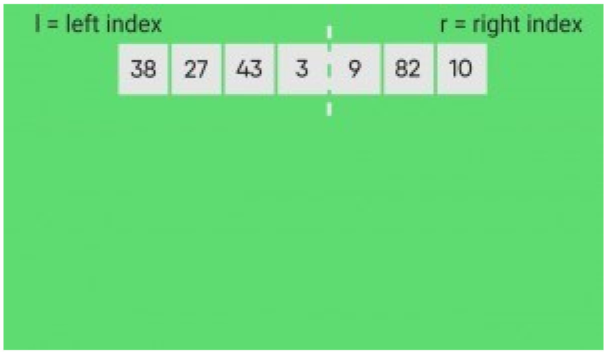
Stable: Yes

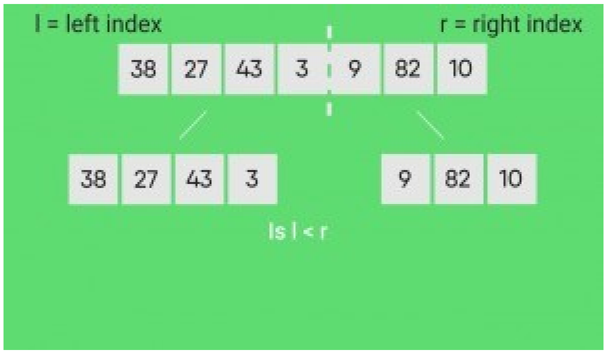
Applications of Merge Sort

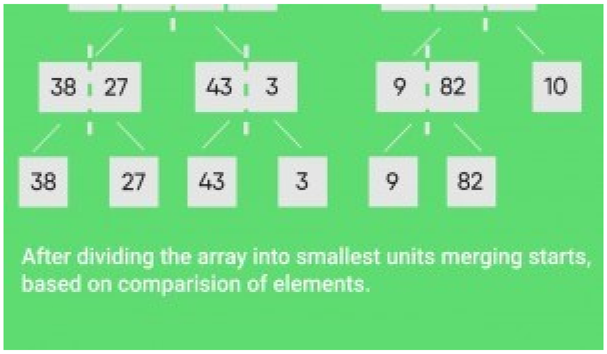
1. Merge Sort is useful for sorting linked lists in O(nLogn) time.In case of linked lists the case is different mainly due to difference in memory allocation of arrays and linked lists. Unlike arrays, linked list nodes may not be adjacent in memory. Unlike array, in linked list, we can insert items in 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.
2. Inversion Count Problem
3. Used in External Sorting

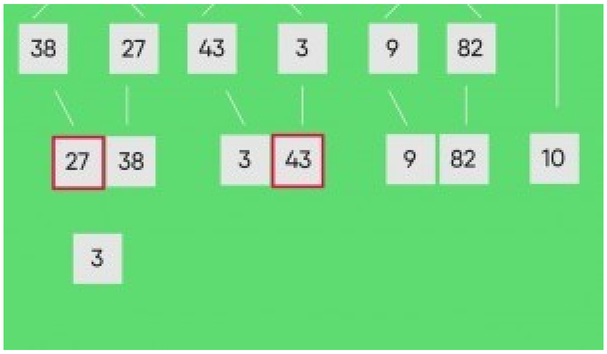
Snapshots:

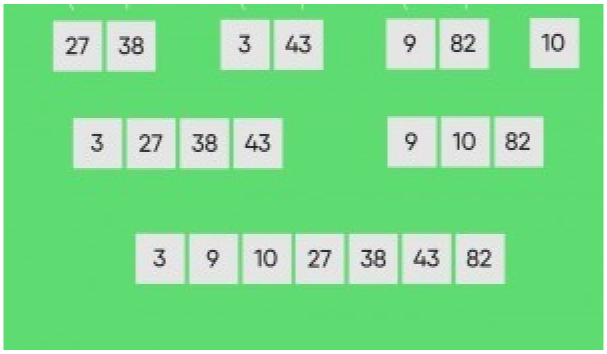

l = left index
r = right index
38 27 43 3 9 82 10
Is l < r
Yes
m= l+(r-1)/2


l = left index
r = right index
38 27 43 3 9 82 10


l = left index
r = right index
38 27 43 3 9 82 10
38 27 43 3
9 82 10
Is l < r


38 27
43 3
9 82
10
38 27 43 3 9 82
After dividing the array into smallest units merging starts, based on comparision of elements.


38 27 43 3 9 82
27 38
3 43
9 82
10
3



- Recent Articles on Merge Sort
- Coding practice for sorting.
- Quiz on Merge Sort

Other Sorting Algorithms on GeeksforGeeks:
3-way Merge Sort, Selection Sort, Bubble Sort, Insertion Sort, Merge Sort, Heap Sort, QuickSort, Radix Sort, Counting Sort, Bucket Sort, ShellSort, Comb Sort

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.