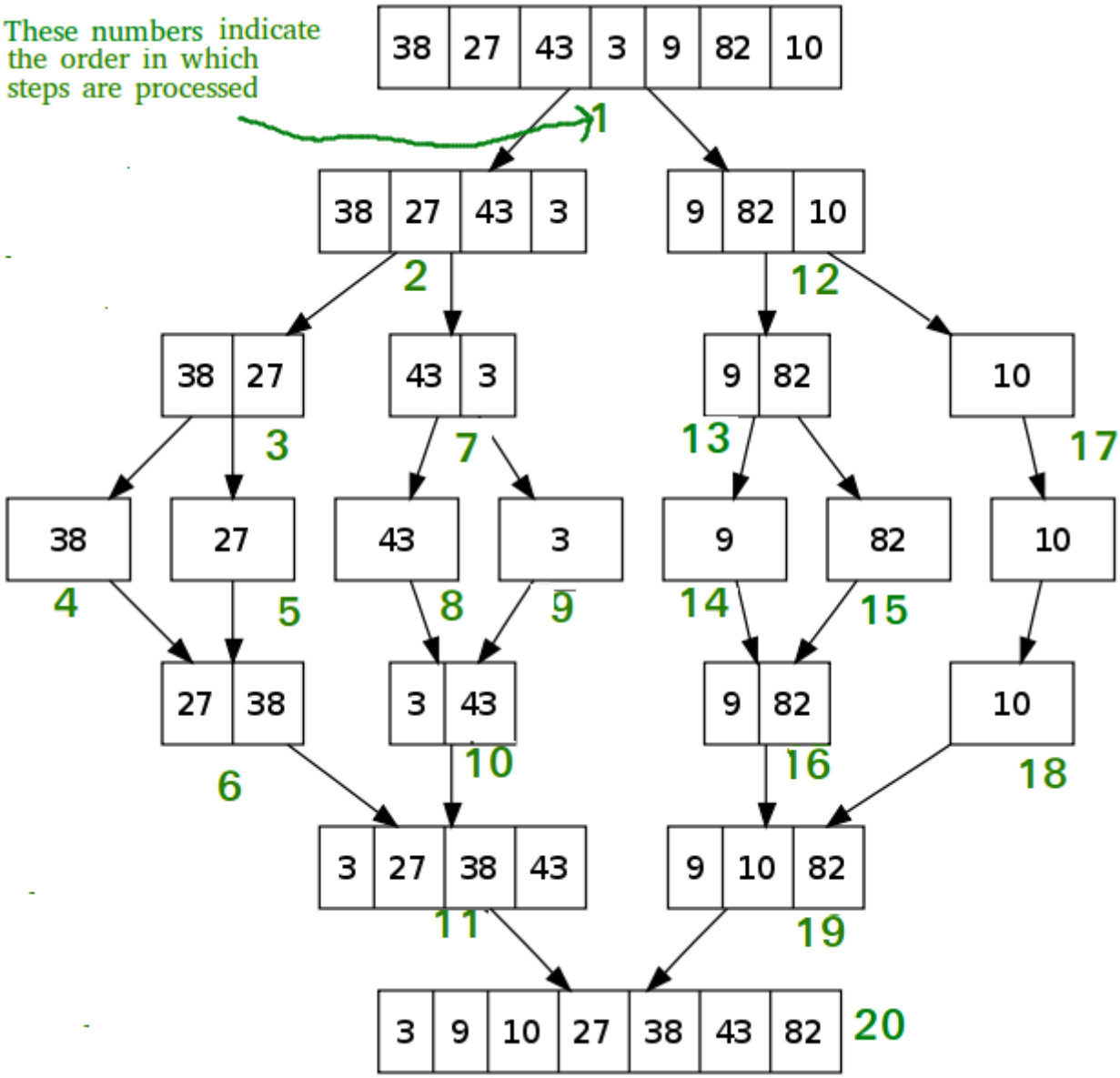
Merge Sort

Like QuickSort, Merge Sort is a Divide and Conquer algorithm. It divides the input array into 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 a 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 the 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-l)/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)




```
def mergeSort(arr,lo,hi):
    if lo==hi:
        return [arr[lo]]
    mid = (lo+hi)//2
    lefthalf = mergeSort(arr,lo,mid)
    righthalf = mergeSort(arr,mid+1,hi)

    sortedHalf = mergeTwoSortedArrays(lefthalf,righthalf)
    return sortedHalf


def mergeTwoSortedArrays(arr1,arr2):
    n = len(arr1)
    m = len(arr2)
    res = [0]*(n+m)
    i = 0
```

```
    j = 0
    k = 0
    while i<n and j<m:
        if arr1[i]<arr2[j]:
            res[k] = arr1[i]
            k = k+1
            i = i+1
        elif arr1[i]>arr2[j]:
            res[k] = arr2[j]
            k = k+1
            j = j+1
        else:
            res[k] = arr1[i]
            k = k+1
            res[k] = arr2[j]
            i = i+1
            j = j+1
            k = k+1

    while i<n:
        res[k] = arr1[i]
        i = i+1
        k = k+1
    while j<m:
        res[k] = arr2[j]
        j = j+1
        k = k+1
    return res
```

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 the Recurrence Tree method or the Master method. It falls in case II of Master Method and the 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 into two halves and takes 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 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 continuous block of memory. Therefore, the overhead increases for quicksort. Merge sort accesses data sequentially and the need of random access is low.
2. Inversion Count Problem
3. Used in External Sorting

Drawbacks of Merge Sort

- Slower comparative to the other sort algorithms for smaller tasks.
- Merge sort algorithm requires additional memory space of 0(n) for the temporary array .
- It goes through the whole process even if the array is sorted.

You have to sort 1 GB of data with only 100 MB of available main memory. Which sorting technique will be most appropriate?

The data can be sorted using external sorting which uses merging technique. This can be done as follows:

1. Divide the data into 10 groups each of size 100.

2. Sort each group and write them to disk.

3. Load 10 items from each group into main memory.

4. Output the smallest item from the main memory to disk. Load the next item from the group whose item was chosen.

5. Loop step #4 until all items are not outputted. The step 3-5 is called as merging technique.

What is the best sorting algorithm to use for the elements in array are more than 1 million in general?

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.