Lab3 Task6

# **Merge Sort**

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, left, mid, right) is a key process that assumes that arr[left..mid] and arr[mid+1..right] are sorted and merges the two sorted sub-arrays into one. As the following pseduecode (C implementation) explains:

**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)

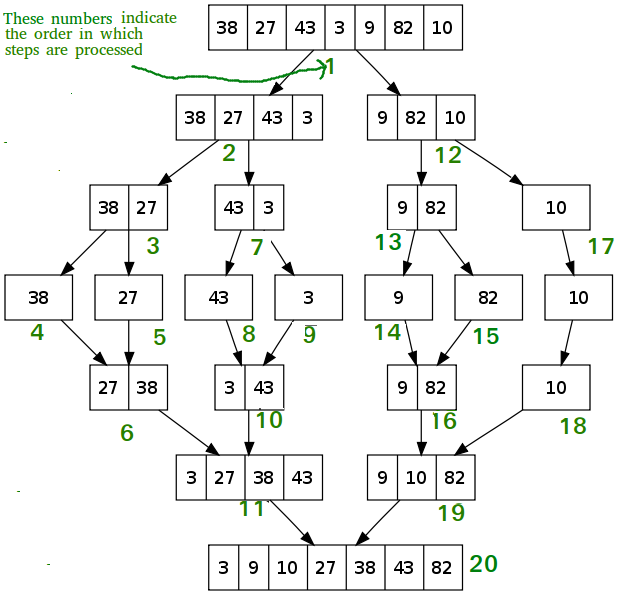
**For example**, The following diagram shows the complete merge sort process for an example array {38, 27, 43, 3, 9, 82, 10}.

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.

**Time Complexity** can be expressed as following relation: T(n) = 2T(n/2) + θ(n)

**Drawbacks** of Merge Sort includes the fact that is it:

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



Merge Sort could be 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.

### **Inversion Count Problem:**

It is how far (or close) the array is from being sorted. If the array is already sorted, then the inversion count is 0, but if the array is sorted in the reverse order, the inversion count is the maximum.

### **Used in External Sorting:**

It is a class of sorting algorithms that can handle massive amounts of data. External sorting is required when the data being sorted do not fit into the main memory of a computing device (usually RAM) and instead they must reside in the slower external memory, usually a hard disk drive.

# **Quick Sort**

QuickSort is a Divide and Conquer algorithm. 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.

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.

**For example**, The following diagram shows the complete Quick sort (pick last element as pivot) process for an example array {10, 80, 30, 90, 40, 50, 70}.



**Time Complexity**, can be written as following: T(n) = T(k) + T(n-k-1) + \0(n)

The first two terms are for two recursive calls, the last term is for the partition process. k is the number of elements which are smaller than 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 greatest or smallest element as pivot: 0(n2).

### **Best Case:**

The best case occurs when the partition process always picks the middle element as pivot: 0(nLogn).

### **Average Case: 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.

**Benefit of Quick Sort includes the fact that is it:**

* 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.
* 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.

**Drawbacks** of Quick Sort includes the fact that is it:

* The default implementation is not stable.
* It uses extra space only for storing recursive function calls but not for manipulating the input.