University of Engineering and Technology

**Analysis Of Algorithms**

**Submitted By:**

(2019-CS-127) Muhammad Awais

**Submitted To:**

**Dr. Sadaf Hina**

Comparison-Based Algorithms

# Merge Sort:

A merge sort is recursively sorting algorithm and it is the classic example of divide and conquer rule.

In this algorithm, first we divide the whole array/problem until we reach the n=1.Then we start sorting the each part of divided array. Finally, we merge them into a single array. Let suppose if you want to sort the elements of array A[1 to n].It basically follow three steps .

if n=1 we are done. Because sorting of one element is already sorted. And we can also say it is the base case of our recursive algorithm. Otherwise what we do we recursively sort array A from 1 up to the [n/2] and [n/2+1] up to the n. So its means we sort the two halves of array. And then, we take those two lists that we have done and we merge them. To do that we use merge subroutine. Here we use the term key subroutine merge . It work like this. After the second step, we have two lists [1 ..n/2] and [n/2+1….n] that we have sorted. And now merge these two what we have to do is produce a sorted list out of both of them. What we do is first we observe where is the smallest element of any two lists which we are already sorted. It maybe the head of first list or head of the second list.

Then what we do is the output in our output array we put the smaller of two lists. Compare and put the elements in our final sorted array. To compare the each element and put them into the array it will take the linear time O(n) and its proportional to the size of input. And what is that exactly mean , here we perform n operation. Each operation will take constant time so n operation will take O(n) time.

## Pseudocode:

* Declare left and right variable.
* Left will be assigned to 0 and right will be assigned to n-1
* Find mid = (left +right)/2
* Call merge Sort on (left, mid) and (mid+1,rear)
* Above will continue till left<right
* Then we will call merge on the 2 subproblems

## Algorithm:

mergeSort (array, left, right):

if left > right

return

mid = (left + right)/2

mergeSort(array, left, mid)

mergeSort(array, mid+1, right)

merge(array, left, mid, right)

end

## Time Complexity:

First step and base condition will take constant amount of time . if(n==1)

**O(1)**

Second step we divide the whole array into two halves it will perform recursion so complexity of this step is

**2T(n/2)**

Final step to merge these two halves is to take

**O(n)**

## Recurrence Relation of Merge Sort:

T(n) = O(1) if n=1 & (2T(n/2)+n) if n>1

Inductive hypothesis:

T(n) <= n-1 + 2T(n/2)

T(n) <= n-1 + 2(n/2)log(n/2)

T(n) =n-1+ n log(n/2)

T(n) <n log n

It will return us the time taken to perform merge sort is **O(n log n)** .

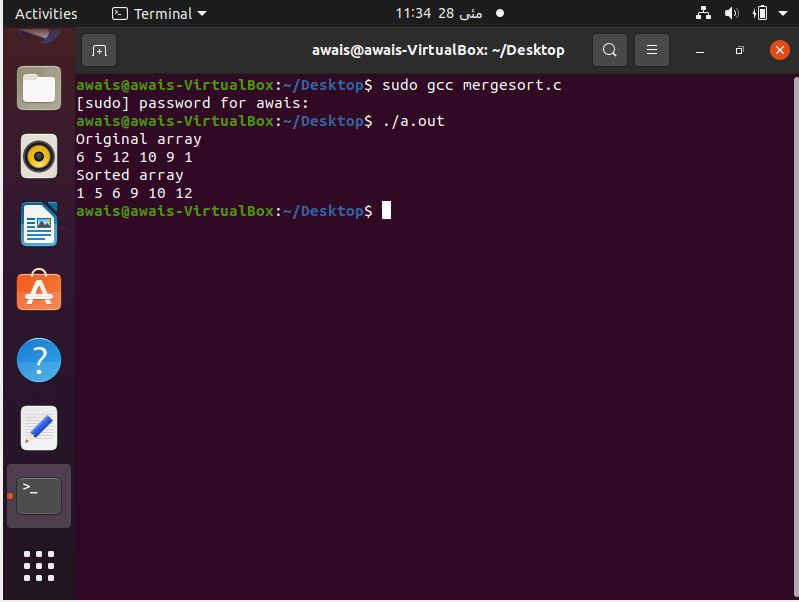
## Strengths:

1. It is quicker for larger lists because unlike insertion and bubble sort.
2. it doesn’t go through the whole list several times.
3. It has a consistent running time, carries out different bits with similar times in a stage.

## Weakness:

1. Slower comparative to the other sort algorithms for smaller tasks.
2. goes through the whole process even the list is sorted (just like insertion and bubble sort?)
3. uses more memory space to store the sub elements of the initial split list.

## Dry Run of Code:



# Quick Sort:

Quick sort is another divide and conquer algorithm. It’s also a in place algorithm , meaning that its just rearranging the elements where they are (like insertion sort). So that it is fairly efficient in use of storage. It’s very practical algorithm if we do tuning. It is also called partition-exchange sort. This algorithm divides the list into three main parts. Elements less than the Pivot element. Pivot element(Central element) and Elements greater than the pivot element. Pivot element can be any element from the array, it can be the first element, the last element or any random element. We will take the rightmost element or the last element as pivot

To perform quick sort , we first divide the whole array into the two subarrays around an element called pivot(x) such that the elements in lower subarrays is less than equal to x and less then equal to the elements in the upper sub array. And the elements in upper subarray is greater than x. In the next conquer step, we just recursively sort the element in two subarrays. First we select the pivot as first element of our array and set the invariant of our loop execution. 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.

And then combine is just trivial because once we sorted the element less than equal to x then sorted the things greater then equal to x the whole thing is sorted. The key step in quick sort is partition step. Now we can easily see quick sort is just recursively partitioning.

## Pseudocode:

1. Make the right-most index value pivot
2. Partition the array using pivot value
3. QuicK Sort left partition recursively.
4. Quick sort right partition recursively.

## Algorithm :

Partition(A ,p, q) // A[p…q]

x<- A[p] // select the pivot element

i<-p

for j<-p+1 to q

do if A[j] <= x

then i =i+1

A[i] <->A[j]

A[p] <-> A[j]

Return i

Quicksort(A,p,q)

If p<q

Then r <- Partition(A,p,q)

Quicksort(A,p,r-1)

Quicksort (A,r+1,q)

Initial call Quicksort(A,1,n)

## Time Complexity:

Assume all elements are distinct.

### T(n) – Worst case running time of Algorithm.

Input array is already sorted or reverse sorted in this case all elements are greater than pivot or less than pivot.

Recurrence Relation made for this condition is

T(n)=T(0) +T(n-1)+O(n)

T(n)=O(1) +T(n-1)+O(n)

T(n)=T(n-1)+O(n)

T(n)=O(n2) +O(n) (Arithmetic series)

### T(n) – Best case running time of Algorithm.

Partition splits the array in middle n/2: n/2

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

T(n) <= n-1 + 2(n/2)log(n/2)

T(n) =n-1+ n log(n/2)

T(n) = O(n log n) (same as merge sort)

### T(n) – Average case running time of Algorithm.

Put together the best and worst case.

T(n)=2T(n/2) +T(n-1)+O(n)

T(n)=2T(n/2-1) +O(n)

T(n) =O(n log n) (in average case).

## Strengths :

1. The quick sort is regarded as the best sorting algorithm.
2. It is able to deal well with a huge list of items.
3. Because it sorts in place, no additional storage is required as well

## Weakness:

1. The slight disadvantage of quick sort is that its worst-case performance is similar to average performances of the bubble, insertion or selections sorts.
2. If the list is already sorted than bubble sort is much more efficient than quick sort.
3. If the sorting element is integers than radix sort is more efficient than quick sort.

## Dry Run of code:

