Merge Sort

Friday, September 8, 2023 10:08 PM

Assume that you have two sorted sub arrays namely (A and B) of size (n1 and n2) respectively and your task is to merge
single array (Arr) of size (N) in sorted order.
here (N = n1+n2). It is purely a merging problem.
=>The idea is to create three iterators namely (i, j, and k) for the arrays (A, B, and Arr) respectively and initialize them with
=>Now compare the data of the arrays A and B and store the smallest in resultant array Arr and update the correspondin

```
=> if(A[i] <= B[j])
       Arr[k++] = A[i++]
   else
       Arr[k++] = B[j++]
```

This condition will be a part of loop, we will look into it completely later in merge sort. However this task can be done in a
can say that if we have two sorted sub arrays then we can merge them into a single sorted array in linear time. Now all w
about is to make the two sorted sub arrays and we will merge them in linear time.
The Question is how we can achieve two sorted sub arrays and how much time will be required for doing this task?
==> This can be done using merge sort.

Main idea of merge sort is to divide the original problem i.e.,(Array of N size) into sub problems until the base case will o
with single unit array. We know that the single unit array is already sorted. So we will solve the sub problems and then m
these sub problems to get the solution of original problem.

Key Points:
=> Divide the array into 2 equal halved sub arrays. **How?**
=> Calculate the mid-point and then make recursive calls to the function for data ranging from **start to mid** and for **mid+1**
way we will be able to get two sorted sub arrays when the control will be returned by the base cases (**dry Run**)
=> call the merge procedure to merge the sorted sub arrays into sorted original array.
=> merge procedure will create two temporary arrays and fetch the data from original arrays ranging from **(start to mid)** a
into these arrays. (Reason behind temp arrays?)
=> Since we are utilizing extra memory in merge procedure, that is why it is **not an in-place** sorting algorithm.
=> **Similar to post order traversal technique i.e.,(Solve the problem in left sub tree, then right sub tree and finally sort th**

Algorithm of Merge Sort

```
Merge_Sort(Arr[], left, right )
{
    If(left < right)                     ==> $O(1)$
    {
        M = (left + right)/2             ==> $O(1)$
        Merge_Sort(Arr, left, M)         ==> $T(N/2)$    recursive call with input size N/2   // logical division of array b
        boundary)
        Merge_Sort(Arr, M+1, right)      ==> $T(N/2)$
        Merge (Arr, left, M, right)      ==> $O(N)$     merge will take linear time to merge two sorted sub-Arrays i.e.,
        from Mid+1
                                                         to end) into single sorted array.
    }
}
```

Main Steps of Merge 1: calculate Size, **2:** Create temp Arrays, **3:** Copy data in temp Arrays, **4:** comparison of temp Arrays
pending elements 6: delete temp arrays

Merge (Arr[], left, M, right)

```
{
    n1 = M - left +1      //size of sub array1
    n2 = right - M        //size of sub array2
    A[n1]                 // temp Arrays definitely on heap
    B[n2]
      copy (Arr, A, left, M)          // By coping the data in temporary arrays, the original input space will get free and we c
the data
                                       inside original array
      copy (Arr, B, M+1 right)
    i = j = 1             // iterators for temp Arrays
    k = left              //   iterator for original array
    while (i < n1 AND j < n2)
    {
        If(A[i] <= B[j])
            Arr[k++] = A[i++]
        Else
            Arr[k++] = B[j++]
    }
```

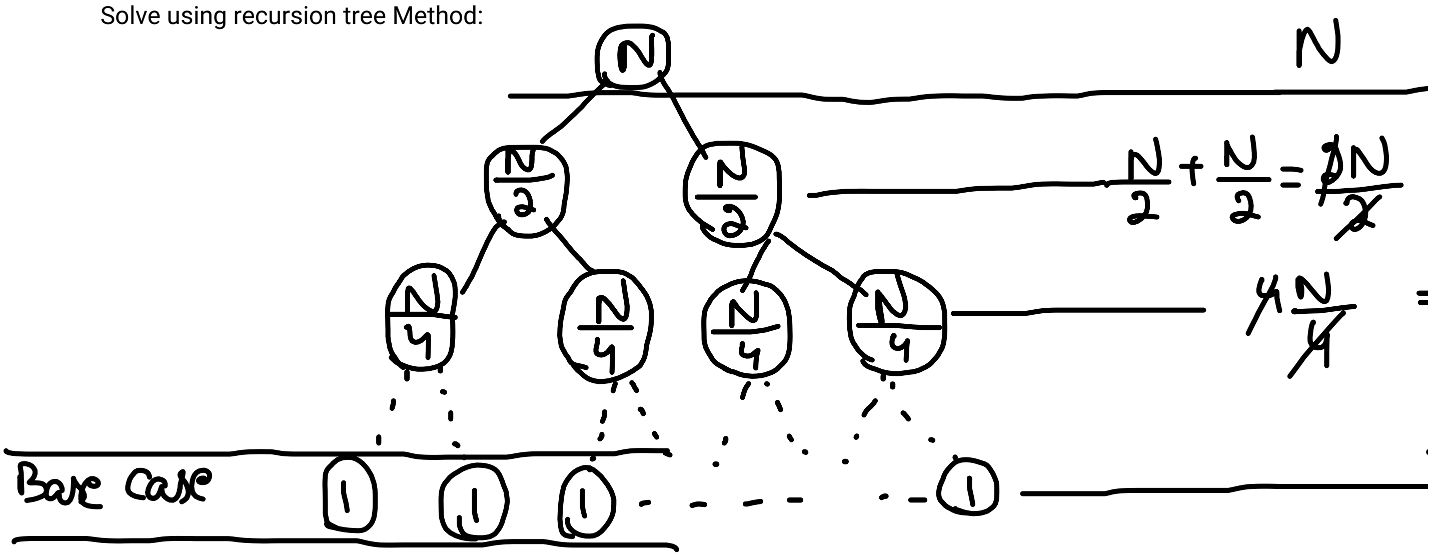
```
        //Either array A or B will get exhausted so we need to copy the pending elements in sorted order in the original array
        while (i < n1)
            Arr[k++] = A[i++]
        while (j < n2)
            Arr[k++] = B[j++]

        Delete A and B
        Since n1+ n2 = N so the overall complexity of this function will be O(N)
}
```

Time complexity Analysis:
Recurrence of Merge Sort: There are 2 recursive calls with input size N/2 and a merge function with linear time O(N)

$$T(N) = T(N/2) + T(N/2) + N$$
$$T(N) = 2T(N/2) + N$$

Solve using recursion tree Method:

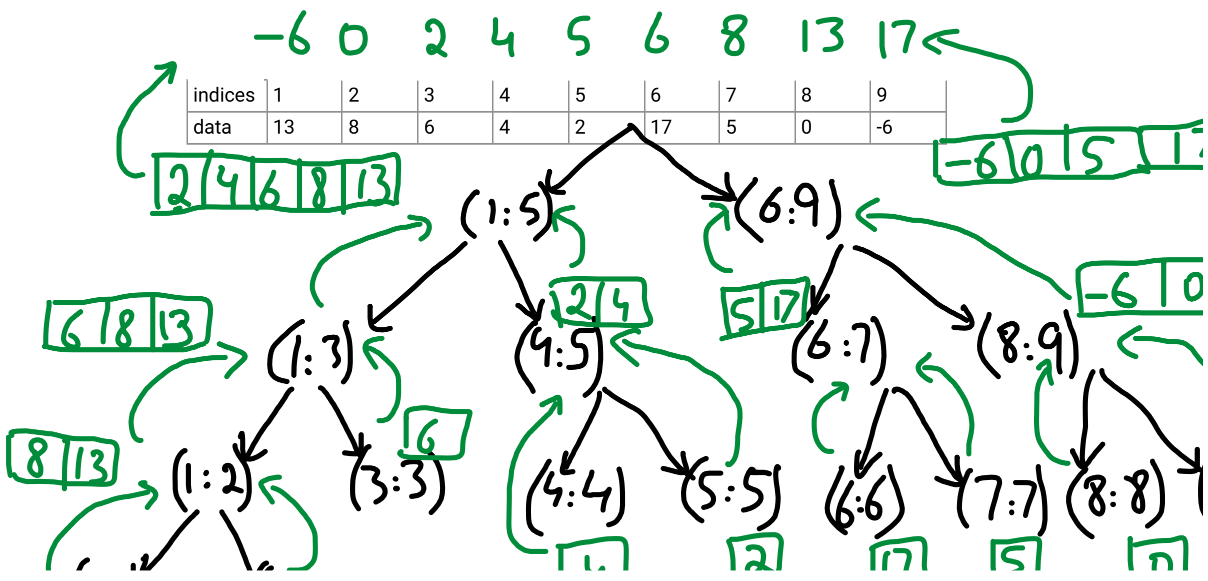


Accumulate horizontally and then vertically.
Horizontal Accumulation: The time required to solve sub problems at each level of tree is same i.e., linear O(N)
Vertical Accumulation: summation of each level cost i.e., ($N + N + \ldots + N$). In such cases where cost at eac level is same we just need to determine the height of tree because overall time complexity will be equals to **height of tree * cost at each level**
Since the dividing factor is 2, so height of tree will be $log_2 N$. Overall time complexity will be $N * log_2 N$

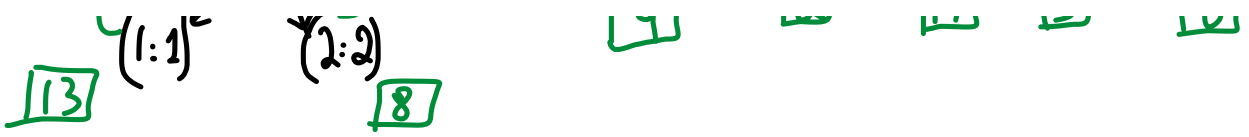
Cost at each level = N
Total # of levels (Height of tree) = $log_2 N$

$$\boxed{\Theta(N \log N)}$$

Dry Run:



indices	1	2	3	4	5	6	7	8	9
data	13	8	6	4	2	17	5	0	-6



*Arrows going downwards are showing the logical division of array by shifting the boundaries e.g., (1:5) means data r
index 1 to 5