CMSC 351: MergeSort

Justin Wyss-Gallifent

October 1, 2024

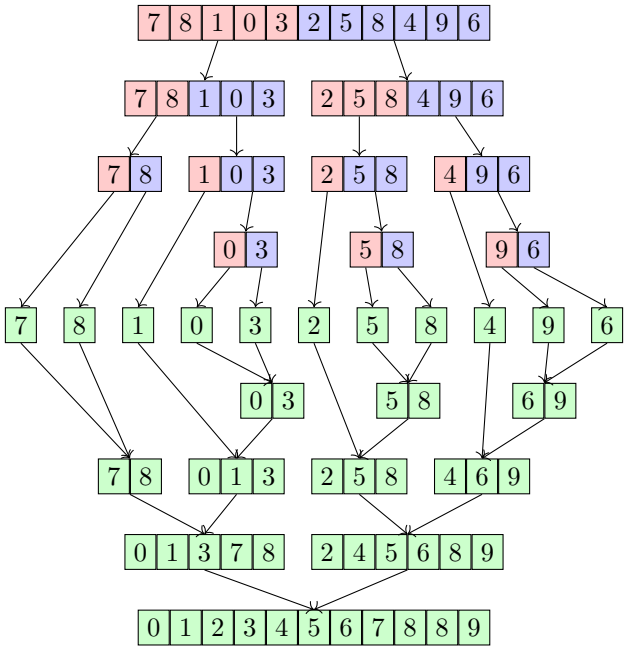
1 What it Does

Sorts a list of elements on which there is a total order. Think of integers or real numbers.

2 How it Works

Merge sort is a divide-and-conquer algorithm whereby the list is subdivided repeatedly in half. Each half is then divided in half again and so on until each sublist has size 1 and is obviously sorted. Pairs of sublists are then merged to preserve the sort.

Here is a visual representation. The red/blue divisions are illustrating how each (sub)list is divided in half. Any green element or group of elements are sorted. All the action above the center line is the recursive deconstruction while all the action below the center line is the re-merging of the sublists.



3 Pseudocode:

Here is the pseudocode for Merge Sort. It's not necessary to write two separate functions but it might help clarify what's going on.

```
# Here is the MergeSort code.
function MergeSort(arr,start,end)
    if start < end
        # Find the middle.
        middle = (start+end) // 2
        # Apply MergeSort to each half.
        MergeSort(arr,start,middle)
        MergeSort(arr,middle+1,end)
        # Merge the two halves back on top of arr.
        Merge(arr,start,middle,middle+1,end)
    end if
end function
# Here is the Merge function that merges two parts of a list.
function Merge(arr,start1,end1,start2,end2)
    temp = array of same size as arr
    i = start1; j = start2; k = start1
    while i <= end1 and j <= end2
        if arr[i] <= arr[j]
            temp[k] = arr[i]; i++; k++
        else
            temp[k] = arr[j]; j++; k++
        end if
    end while
    while i <= end1
        temp[k] = arr[i]; i++; k++;
    end while
    while j <= end2
        temp[k] = arr[j]; j++; k++
    end while
    # Copy temp on top of arr.
    for i = start1 to end2 inclusive
        arr[i] = temp[i]
    end for
end function
```

4 Pseudocode Time Complexity:

For a list of length n:

- The `MergeSort` function does a constant-time operation and then makes two recursive calls to lists of length essentially $n/2$ and then makes a call to `Merge`. It does all of this only if the length of the list is more than 1, so this length 1 would be the base case; If the list is length 1 then it just runs the conditional check at $\Theta(1)$.
- The `Merge` function itself operates on pairs of sublists which add to the total length n. It runs at $\Theta(n)$.

It follows that the asymptotic time complexity on an input of size n therefore satisfies the recurrence relation:

$$T(n) = 2T(n/2) + f(n) \text{ with } T(1) \text{ constant and } f(n) = \Theta(n)$$

This recurrence relation can be solved either by digging down, with a recurrence tree, or with the Master Theorem, resulting in $T(n) = \Theta(n \lg n)$.

Note that this is best, worst, and average-case. This is because MergeSort breaks down the list and puts it back together no matter what, even if the list is sorted at the start. Moreover the process of sorting the recursive parts during the reconstruction process is no quicker whether the parts are sorted or not.

5 Pseudocode Auxiliary Space

One fun fact is that we can analyze the auxiliary space using a recurrence relation. For a list of length n:

- The `MergeSort` function requires `middle`, which is $\Theta(a)$. It then makes two recursive calls to lists of length essentially $n/2$ and then makes a call to `Merge`. However the two recursive calls are in series - one after the other, meaning that any auxiliary space taken by the first call is released before the second call.
- The `Merge` function itself requires the creation of `temp` which has length n and also a few index counters. This is then $\Theta(n)$ auxiliary space on its own.

It follows that the asymptotic space complexity on an input of size n therefore satisfies the recurrence relation:

$$S(n) = S(n/2) + f(n) \text{ with } S(1) \text{ constant and } f(n) = \Theta(n)$$

This recurrence relation can be solved either by digging down, with a recurrence tree, or with the Master Theorem, resulting in $S(n) = \Theta(n)$.

6 Stability

Our MergeSort psuedocode is stable.

7 In-Place

Our MergeSort pseudocode is not in-place.

8 Notes

MergeSort is not interative in any sense which lends itself to an easy analysis of what any intermediate steps look like.