CMSC 351: MergeSort

Justin Wyss-Gallifent

October 4, 2023

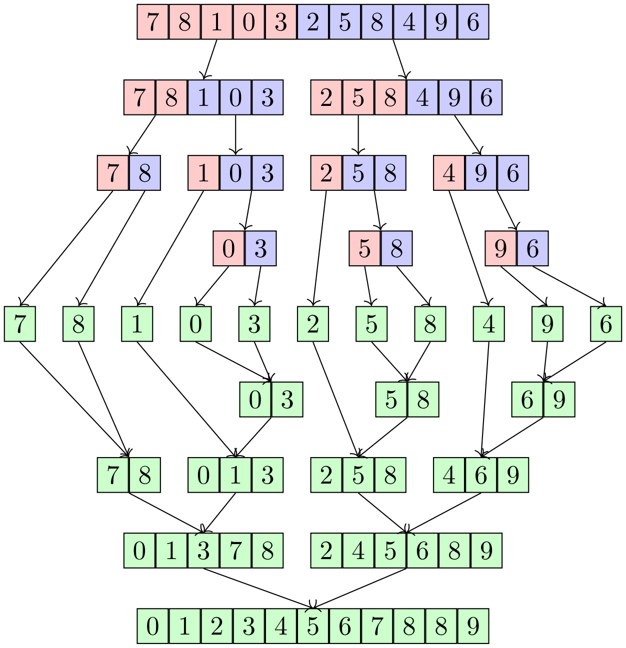
1 What it Does

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:

Nothing is assumed to be global here.

```
\\ PRE: A is a list of integers.
function mergesort(A)
    if len(A) > 1
        m = len(A) // 2
        L = A[0,...,m-1]
        R = A[m,...,len(A)-1]
        L = mergesort(L)
        R = mergesort(R)
        \\ Merge L and R back on top of A.
        Lind = 0
        Rind = 0
        Aind = 0
        while Lind < len(L) and Rind < len(R)
            if L[Lind] <= R[Rind]:
                A[Aind] = L[Lind]
                Lind ++
                Aind ++
            else:
                A[Aind] = R[Rind]
                Rind ++
                Aind ++
        end
        while Lind < len(L)
            A[Aind] = L[Lind]
            Lind ++
            Aind ++
        end
        while Rind < len(R)
            A[Aind] = R[Rind]
            Rind ++
            Aind ++
        end
    end
    return(A)
end
\\ POST: A is sorted.
```

A comment on the merging of `L` and `R`: we initialize indices for each of these and, while there are elements left in both, copy the smaller one off the corresponding list and overwrite it onto `A`. Once one has no elements remaining we simply copy all the elements in the other, one-by-one, and overwrite them onto `A`.

4 Pseudocode Time Complexity:

Observe that other than the two recursive calls to `mergesort` there are constant time calculations and three `while` loops.

However observe that the three `while` loops together result in a total of n iterations because together they just merge `L` and `R` back together and since `L` and `R` together form `A`, the claim follows.

Together then, other than the two recursive calls, $\Theta(n)$ time is required. It follows that the 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 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 Auxiliary Space

5.1 Auxiliary Space without the Master Theorem 1

The auxiliary space is a little confusing so let's step through it carefully. Here is a really simple pseudocode simplification of `mergesort`.

```
\\ PRE: A is a list of integers.
function mergesort(A)
    if len(A) > 1
        split A to L and R
        L = mergesort(L)
        R = mergesort(R)
        A = merge L and R
    end
    return(A)
end
\\ POST: A is sorted.
```

Consider a list of length $n = 16$. Our first call to Mergesort (recursion level 0) requires auxiliary space of 16 to do `split A to L and R`. In other words this is simply to allocate L and R.

The call `L = mergesort(L)` will subsequently require 8 more but this call ends and that memory is released before the call `R = mergesort(R)`. Consequently recursion level 1 only requires 8 more in total. Think of these 8 as being used first to manage L and then to manage R.

Now then, recursion level 1 makes four calls down to recursion level 2 but again these are done in series (first half of L then second half of L then first half of R then second half of R) so at that level only 4 more are needed in total. Think of these 4 as being used four times at that level,

This continues until we have lists of lenght 1 which require no space as they are not split, just returned. Consequently in the $n = 16$ case the total auxiliary memory required is:

$$16 + 8 + 4 + 2$$

Similarly simplicity suppose we have a list of length $n = 2^k$. Then the total auxiliary memory required is:

$$\begin{aligned}
n+\frac{n}{2}+\frac{n}{4}+...+\frac{n}{2^{k-1}} &= n\left(\left(\frac{1}{2}\right)^0+\left(\frac{1}{2}\right)^1+\left(\frac{1}{2}\right)^2+...\left(\frac{1}{2}\right)^{k-1}\right)\\
&= n\left(\frac{\left(\frac{1}{2}\right)^k-1}{\frac{1}{2}-1}\right)\\
&= 2n\left(1-\left(\frac{1}{2}\right)^k\right)\\
&= 2n\left(1-\frac{1}{n}\right)\\
&= \Theta(n)
\end{aligned}$$

5.2 Auxiliary Space without the Master Theorem 2

If that explanation is confusing, think from smaller lists to larger.

A list of length 2 splits to $1+1$, requiring 2.

A list of length 4 splits to $2+2$, requiring 4. It then runs Mergesort on the first half, requiring an additional 2, but that call is done and over and the memory released before it runs Mergesort on the second half, meaning the 2 for the second half is not additional. We thus have a total of $4+2$.

A list of length 8 splits to $4+4$, requiring 8. It then runs Mergesort on the first half, requiring an additional $4+2$, but that call is done and over and the memory released before it runs Mergesort on the second half, meaning the $4+2$ for the second half is not additional. We thus have a total of $8+4+2$.

In general a list of length 2^k splits to $2^{k-1}+2^{k-1}$, requiring 2^k. It then runs Mergesort on the first half, requiring an additional $2^{k-1}+...+4+2$, but that call is done and over and the memory released before it runs Mergesort on the second half, meaning the $2^{k-1}+...+4+2$ for the second half is not additional. We thus have a total of:

$$2^k+2^{k-1}+...+4+2$$

If $n=2^k$ this is exactly the same sum as the previous subsection calculation.

5.3 Auxiliary Space with the Master Theorem

The Master Theorem can be applied to the auxiliary space of MergeSort if we are particularly careful. We might be tempted to think that the auxiliary space $S(n)$ satisfies the recurrence relation $S(n)=2S(n/2)+f(n)$ where $f(n)=\Theta(n)$ but this is false. The reason for this is what we saw above, that the two adjacent recursive calls to MergeSort are called in series and the auxiliary space for the first call is released before the second call. The recurrence relation above suggests that we are adding the auxiliary space of the two adjacent calls, much as we would add the time requirement, and this is not true, rather we should only include it once.

Thus instead the recurrence relation satisfied by the auxiliary space is $S(n) = S(n/2) + f(n)$ with $f(n) = \Theta(n)$.

Since $\Theta(n) = \Theta(n^1) = \Theta(n^c)$ with $c = 1$ and since $\log_b a = \log_2 1 = 0 < 1 = c$ the third case of the Master Theorem is satisfied and $S(n) = \Theta(n^1) = \Theta(n)$.

5.4 Auxiliary Space Commentary

It it reasonable to ask: Instead of creating new lists, applying Merge Sort to those, then re-merging them on top of A, why don't we just apply Merge Sort directly to the left and right sublists of A, keeping them in-place, and then merge them back on top of A? The same idea could be reprased in terms of using references to A (thereby not creating new lists).

This does not work. The issue can be illustrated with the following example. Suppose $A = [5, 8, 6, 7, 4, 3, 2, 1]$. If we split this into $[5, 8, 6, 7]$ and $[4, 3, 2, 1]$ (using the same A or just use references to these sublists) and if we sort these we get $[5, 6, 7, 8]$ and $[1, 2, 3, 4]$.

At this point we have $A = [5, 6, 7, 8, 1, 2, 3, 4]$. If we now try to merge the left and right halves and if we do this on top of A, the first thing we do is pick out the 1 (it's the smallest first element in the two halves) but then we plop it down on top of $A[0]$ so now $A = [1, 6, 7, 8, 1, 2, 3, 4]$. Now we have lost the 5.

The practical upshot is that we are forced to put the left and right halves in new lists so that we can merge them from those new lists back on top of A.

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.

9 Thoughts, Problems, Ideas

1. Diagram the action of MergeSort on the list: `5,0,7,10,3,8,10,4,1`.
2. Suppose that in the recurrence relation:

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

   we had $f(n) = 5n + 2$ and $T(1) = 1$. Use a recurrence tree to calculate the time requirement and show that the result is still $\Theta(n \lg n)$.
3. Suppose for the sake of argument that Merge Sort took $T_M(n) = 7n \lg n + 10n$ and another sort you had available called `supersort(A)` took $T_S(n) = \frac{2}{3}n^2 + n$. For which n is `supersort` actually faster and how could you combine the two algorithms to produce a best-of-both-worlds result?
4. Rewrite the pseudocode of Merge Sort so that it does a three-way split instead of a two-way split.
5. Explain why MergeSort is stable.

10 Python Test

Code:

```
import random
def mergesort(A,indent):
    print(indent * ’_’ + ’Mergesort:’ + str(A))
    if len(A) > 1:
        m = len(A) // 2
        L = A[:m]
        R = A[m:]
        mergesort(L,indent+2)
        mergesort(R,indent+2)
        i = 0
        j = 0
        k = 0
        while i < len(L) and j < len(R):
            if L[i] < R[j]:
                A[k] = L[i]
                i += 1
            else:
                A[k] = R[j]
                j += 1
            k += 1
        while i < len(L):
            A[k] = L[i]
            i += 1
            k += 1
        while j < len(R):
            A[k] = R[j]
            j += 1
            k += 1
        print(indent * ’_’ + ’Merge:’ + str(L) + ’ and ’ + str(R))
        print(indent * ’_’ + ’Result: ’ + str(A))
A = []
for i in range(0,11):
    A.append(random.randint(0,100))
print(A)
mergesort(A,0)
print(A)
```

Output:

```
[93, 95, 44, 67, 11, 89, 10, 71, 11, 88, 50]
Mergesort:[93, 95, 44, 67, 11, 89, 10, 71, 11, 88, 50]
__Mergesort:[93, 95, 44, 67, 11]
____Mergesort:[93, 95]
______Mergesort:[93]
______Mergesort:[95]
____Merge:[93] and [95]
____Result: [93, 95]
____Mergesort:[44, 67, 11]
______Mergesort:[44]
______Mergesort:[67, 11]
________Mergesort:[67]
________Mergesort:[11]
______Merge:[67] and [11]
______Result: [11, 67]
____Merge:[44] and [11, 67]
____Result: [11, 44, 67]
__Merge:[93, 95] and [11, 44, 67]
__Result: [11, 44, 67, 93, 95]
__Mergesort:[89, 10, 71, 11, 88, 50]
____Mergesort:[89, 10, 71]
______Mergesort:[89]
______Mergesort:[10, 71]
________Mergesort:[10]
________Mergesort:[71]
______Merge:[10] and [71]
______Result: [10, 71]
____Merge:[89] and [10, 71]
____Result: [10, 71, 89]
____Mergesort:[11, 88, 50]
______Mergesort:[11]
______Mergesort:[88, 50]
________Mergesort:[88]
________Mergesort:[50]
______Merge:[88] and [50]
______Result: [50, 88]
____Merge:[11] and [50, 88]
____Result: [11, 50, 88]
__Merge:[10, 71, 89] and [11, 50, 88]
__Result: [10, 11, 50, 71, 88, 89]
Merge:[11, 44, 67, 93, 95] and [10, 11, 50, 71, 88, 89]
Result: [10, 11, 11, 44, 50, 67, 71, 88, 89, 93, 95]
[10, 11, 11, 44, 50, 67, 71, 88, 89, 93, 95]
```