CMSC 351: MergeSort

Justin Wyss-Gallifent

April 8, 2021

1. What it Does ............................................. 2
2. How it Works ............................................. 2
3. Pseudocode ............................................. 3
4. Pseudocode Time Complexity: .......................... 4
5. Auxiliary Space ........................................... 5
   5.1 Auxiliary Space without the Master Theorem .... 5
   5.2 Auxiliary Space with the Master Theorem ....... 7
6. Stability .................................................. 7
7. In-Place .................................................. 7
8. Notes ...................................................... 7
9. Thoughts, Problems, Ideas ................................ 8
10. Python Test ............................................ 9
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.

```plaintext
// 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.
        i = 0
        j = 0
        k = 0
        while i < len(L) and j < len(R)
            if L[i] <= R[j]:
                A[k] = L[i]
                i = i + 1
                k = k + 1
            else:
                A[k] = R[j]
                j = j + 1
                k = k + 1
        end
        while i < len(L)
            A[k] = L[i]
            i = i + 1
            k = k + 1
        end
        while j < len(R)
            A[k] = R[j]
            j = j + 1
            k = k + 1
        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 \texttt{mergesort} there are constant time calculations and three \texttt{while} loops. However observe that the three \texttt{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:

$$T(n) = 2T(n/2) + f(n)$$ with $T(1)$ 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

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.
```

The line `split A to L and R` takes time $n c_1$. and the line `A = merge L and R` takes time $n c_2$. Both of these follow from the fact that they’re essentially just moving $n$ elements. Let’s say the `return` takes time $c_3$.

As an example, consider the list $A = [4,3,2,1]$. The flow of the algorithm follows:

```
mergesort([4,3,2,1])
    split [4,3,2,1] to [4,3] and [2,1]
    mergesort([4,3])
        mergesort([4])
        mergesort([3])
            A = [3]
        A = [3,4]
    mergesort([2,1])
        split [2,1] to [2] and [1]
        mergesort([2])
            A = [2]
        mergesort([1])
            A = [1]
        A = [1,2]
    A = [1,2,3,4]
end
```

Notice because the two adjacent calls to `mergesort` are called in series the auxiliary space for the first call is released before the auxiliary space for the second call is needed. This means that in accounting for the auxiliary space we only need to look at how deep the calls go along a single branch.
The deepest that the calls levels go is three levels; The first call with four elements, the second call with two elements, and the third call with one element.

Counting the call levels separately the auxiliary space required for the first call is $4c_1 + 4c_2 + c_3$, the auxiliary space required for the second call is $2c_1 + 2c_2 + c_3$, and the auxiliary space required for the third call is $c_3$ only.

The total time is then as follows. We’ve written it this way because it makes it easier to see the generalization later:

$$T(4) = 4(c_1 + c_2) + 2(c_1 + c_2) + 3c_3$$

Consider the case where $n = 2^k$. The deepest that the call levels go is $k + 1$ levels; the first call with $2^k$ elements, the second call with $2^{k-1}$ elements, ..., the $k^{th}$ call with $2^{k-(k-1)} = 2^1 = 2$ elements, and the $(k+1)^{th}$ call with $2^{k-k} = 1$ element.

Counting the call levels separately the auxiliary space required for the first call is $2^k(c_1 + c_2) + c_3$, the auxiliary space required for the second call is $2^{k-1}(c_1 + c_2) + c_3$, ..., the auxiliary space required for the $k^{th}$ call is $2^1(c_1 + c_2) + c_3$, and the auxiliary space required for the $(k+1)^{th}$ call is $c_3$ only.

The total time is then:

$$T(n) = \left[ \sum_{i=1}^{k} 2^i(c_1 + c_2) + c_3 \right] + c_3$$

$$= (c_1 + c_2)(2^1 + 2^2 + ... + 2^k) + c_3(k + 1)$$

$$= (c_1 + c_2)(2^{k+1} - 2) + c_3(k + 1)$$

$$= (c_1 + c_2)(2n - 2) + c_3(1 + \log n)$$

$$= \Theta(n)$$

A similar but messier argument holds for when $n$ is not a power of 2.
5.2 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 $S(n) = \Theta(n^1) = \Theta(n)$.

6 Stability

Our MergeSort pseudocode is stable.

7 In-Place

Our MergeSort pseudocode is not in-place.

8 Notes

MergeSort is not iterative 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:

```python
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)
```

A = []
for i in range(0, 11):
    A.append(random.randint(0, 100))
print(A)
mergesort(A, 0)
print(A)
Output:

```plaintext
[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:[44]
________Mergesort:[67, 11]
_________Mergesort:[67]
_________Mergesort:[11]
______Merge:[67] and [11]
____Result: [11, 67]
____Mergesort:[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:[89]
________Mergesort:[10, 71]
_________Mergesort:[10]
_________Mergesort:[71]
______Merge:[10] and [71]
____Result: [10, 71]
____Mergesort:[89] and [10, 71]
____Result: [10, 10, 71, 89]
___Mergesort:[11, 88, 50]
____Mergesort:[11]
____Mergesort:[11]
____Mergesort:[88, 50]
______Mergesort:[88]
______Mergesort:[50]
_____Merge:[88] and [50]
_____Result: [50, 88]
_____Mergesort:[11] and [50, 88]
_____Result: [11, 50, 88]
__Merge:[10, 71, 89] and [11, 50, 88]
__Result: [10, 11, 50, 71, 88, 89]
Mergesort:[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]
```

10