Divide and Conquer Design & MergeSort

Suhas Arehalli

COMP221 - Spring 2024

1 Divide and Conquer

- A new algorithmic design strategy!
- **Idea**: We keep seeing that small instances of a problem are almost trivially solvable
  - This is what we see in the base case of a lot of our proofs by induction!
  - **What if we can transform large instances of a problem into multiple smaller, easier-to-solve instances of that problem**. We can keep breaking down that large problem (recursively) until we arrive at trivial version of that problem!
- If we can find a way to break a problem down into problems of half the size, it only takes a *logarithmic* amount of steps to get to a trivial problem size (i.e., n some small constant like 0 or 1).
- Divide and Conquer is a very powerful design principle, and we'll spend an entire unit (After the exam!) talking about it! Consider this a taste!

2 Designing Mergesort

MergeSort is one application of the divide and conquer design strategy to the problem of sorting. It can be captured with one s

> **Idea**: Split the array into two halves recursively, and then merge the two into one sorted list.

This works because we're able to write a fast algorithm Merge that combines those two arrays in $\Theta(n)$ time. Intuitively, you can think of this as a special case of a SelectionSort: At each step, you find the minimum remaining elements, but since we have two sorted arrays L and R to combine, the smallest element is either the smallest (i.e., leftmost remaining) element of L or the smallest (leftmost remaining) element of R, which we can find in constant time. Rinse, repeat, and we get the Merge algorithm below!

Algorithm 1 Pseudocode for MergeSort

```
function MergeSort(Array $A[1 \dots N]$)
    if $N \leq 1$ then
        return A
    end if
    $midpoint \leftarrow \lceil \frac{N}{2} \rceil$
    $L \leftarrow$ MergeSort($A[1...midpoint]$)
    $R \leftarrow$ MergeSort($A[midpoint + 1...N]$)
    return Merge($L$, $R$)
end function

function Merge(Array $L[1 \dots n]$, Array $R[1 \dots m]$)
    $N \leftarrow n + m$
    $C \leftarrow Array[1 \dots N]$
    $i \leftarrow 1$
    $j \leftarrow 1$
    for $k \leftarrow 1$ to $N$ do
        if $L[i] < R[j]$ or $j$ out-of-bounds in $R$ then
            $C[k] \leftarrow L[i]$
            $i \leftarrow i + 1$
        else (or $i$ out-of-bounds in $L$)
            $C[k] \leftarrow R[j]$
            $j \leftarrow j + 1$
        end if
    end for
    return C
end function
```

3 Proof of Correctness

We'll proceed in two steps: First, proving that our Merge algorithm is correct, and then using that to prove MergeSort is correct.

3.1 Merge

Since this is an iterative algorithm, we'll use a loop invariant to help us prove correctness by induction. I recommend attempting to prove the algorithm's correctness yourself before reading the full section so you can check your work!

This algorithm's goal is to take in two *sorted* arrays and merge them into a single sorted array, so keep in mind that we can always assume that L and R are sorted (we'll simply observe that L and R are not manipulated, so they will always remain sorted).

Problem Statement (Merge).
Input: Sorted Arrays $L[1 \dots n], R[1 \dots m]$
Output: Sorted Array $C[1 \dots n+m]$ with the elements of L and R.

First, we have to determine our loop invariant. We should first observe that after every iteration, the loop will find the minimum element in $L[i \dots]$ and $R[j \dots]$ — the "unsorted" part of our Array, if we think about this like our $\Theta(n^2)$ sorts. This means the contents of C should always be smaller than the unsorted portion! And, of course, our goal is to slowly construct a sorted C, and since we add 1 element to C at $C[k]$ each iteration, we probably also want to say something about $C[1 \dots k]$ being sorted. Thus...

Statement 1 (Merge Loop Invariant).
After the iteration lth iteration, for $e_1 \in C[1 \dots l]$ and $e_2 \in L[i \dots] \bigcup R[j \dots]$, $e_1 \leq e_2$, and $C[1 \dots l]$ is sorted.

Proof. Proceed by induction over the iterations of the loop.

Base Case: Before the first iteration, $l = 0$ and thus both claims are trivial, as $C[1 \dots 0] = \emptyset$.

Inductive Step: By our inductive hypothesis, we can assume that after the lth iteration, elements in $C[1 \dots l]$ are less than or equal to elements in $L[i \dots] \bigcup R[j \dots]$ and $C[1 \dots l]$ is sorted. We must show that after the $l+1$st iteration, elements in $C[1 \dots l+1]$ are less than or equal to elements in $L[i \dots] \bigcup R[j \dots]$ and $C[1 \dots l]$ and $C[1 \dots l+1]$ are sorted.

Note that, for the sake of not shuffling too many variable/constant names around, we'll refer to the values of i and j at the end of the lth iteration as i_l, j_l and the values of the variable after the $l+1$st iteration as i_{l+1}, j_{l+1}.

Due to the if-else conditional, We will claim (and show!) that $C[l+1] = min(L[i_l], R[j_l])$ (assuming we choose the existing value if one index is out-of-bounds). To see this, we just assess the cases corresponding to the if-else:

Case 1: We enter the if-statement, and we know that either $R[j_l]$ doesn't exist (j_l is out of bounds) or $L[i_l] < R[j_l]$. In this case, we assign $L[i_l]$ to $C[l+1]$, as desired.

Case 2: We enter the else, which means that either $L[i_l]$ doesn't exist or $R[j_l] \leq L[i_l]$. Either way, we want $C[i+1] = R[j_l]$, which is exactly what we do!

This means that...

$$C[l+1] \leq L[i_l] \leq L[i_l+1] \leq \cdots \leq L[N/2]$$
$$C[l+1] \leq R[j_l] \leq L[j_l+1] \leq \cdots \leq R[N/2]$$

The first inequality in each sequence comes from our minimum claim above, and the rest come from the fact that L and R are sorted by assumption!

Those inequalities together tell us that $C[l+1] \leq e_2$, $\forall e_2 \in L[i_l \ldots] \bigcup R[j_l \ldots]$. Now we simply note that $C[1 \ldots l]$ has not been changed in the loop, so combine this with the IH to find that

$$e_1 \leq e_2, \forall e_1 \in C[1 \ldots l+1], e_2 \in L[i_l \ldots] \bigcup R[j_l \ldots]$$

which can get us to the slightly weaker

$$e_1 \leq e_2, \forall e_1 \in C[1 \ldots l+1], e_2 \in L[i_{l+1} \ldots] \bigcup R[j_{l+1} \ldots]$$

as either $i_{l+1} \geq i_l$ and $j_{l+1} \geq j_l$.[1]

What remains is to show that $C[1 \ldots l+1]$ is sorted. However, since only $C[l+1]$ changed, we need only show that $C[l] \leq C[l+1]$. We first observe again that $C[l+1] = min(L[i_l], R[j_l])$, and then note that, by our inductive assumption, elements in $C[1 \ldots l]$ are less than or equal to elements in $L[i_l \ldots] \bigcup R[j_l \ldots]$. Since $C[l]$ is in $C[1 \ldots l]$ and both $L[i_l]$ and $R[i_l]$ are in $L[i_l \ldots] \bigcup R[j_l \ldots]$, we know that $C[l] \leq C[l+1]$, as desired! Thus we can conclude that $C[1 \ldots l+1]$ is sorted, and we have the other half of our loop invariant. □

We then can conclude that after the final Nth iteration, we have that $C[1 \ldots N] = C$ is sorted. As per usual, I'm going to be a little sloppy by not proving that C contains all and only the values in L and R, but that should be fairly straightforward to show — the sorted-ness is the tricky part!

3.2 MergeSort

Note that this is a recursive algorithm, so we can proceed directly by induction. Like BINARY-SEARCH, we can see that the recursive calls are half the size of the original call, so we need to use *strong* induction.

Proof. Proceed by strong induction over the length of the input array N.

Base Case: Suppose $N = 0$ or 1. In this case, we enter the first conditional and return A. Since every array of size 0 or 1 is sorted, we're done.

Recursive Case: Our inductive hypothesis will let us assume that MERGESORT(A) is correct for all A with $0 \leq N < K$. We must show that MERGESORT(A) is correct for A with $N = k$.

Because of our inductive hypothesis, we know that L and R contain all of the elements of $A[1 \dots \mathit{midpoint}]$ and $A[\mathit{midpoint} + 1 \dots N]$ respectively in sorted order. Since we proved the correctness of Merge in the previous section, we can assume that what we return is an array that contains all and only the elements of L and R in sorted order, which is exactly all and only the elements of A in sorted order. This is exactly our definition of correctness for sorting! □

4 Runtime Analysis

- The general strategy for proving the time complexities of recursive functions like MergeSort is through finding *recurrence relations* — equations that tell us how the time complexity of one call to the function can be writen in terms of the time complexities of it's recursive calls.
- In the next few weeks, we'll be seeing the *Master Theorem*, a mathematical result that can solve these recurrence relations effectively. But we'll hold off on that!
- For now, let's just consider two more informal ways of looking at the time complexity of MergeSort.

4.1 How many Merges?

We can gain some intution for the $\Theta(n \log n)$ behavior of MergeSort by realizing that the dominant time sink in every call to MergeSort is the Merge. Thus, if we track the total asymptotic cost of Merge through all of our recursive calls, we have a good picture of the cost of MergeSort.

It should be fairly straightforward to show that Merge is $\Theta(n)$, so I'll leave that as an exercise.

In our initial call, we will Merge the full array of size n. In each of our two recursive calls, we will Merge two subarrays of size $\frac{n}{2}$. In the recursive calls from those recursive calls (let's call them depth 2 recursive calls!), we will Merge four subarrays of size $\frac{n}{4}$. This pattern continues! recursive calls of depth k will consist of 2^k Merges of size $\frac{n}{2^k}$. If we look carefully, this is about $\Theta(n)$ worth of work at each depth! How deep do we need to recurse before we hit the base case? Since, at each level we half the size of each recursive call, there will only be $\Theta(\log n)$ levels of depth! Thus our time is $\Theta(n \log n)$. Figure 4.4 of Skiena is a helpful visual!

4.2 Analyzing a Recurrence

Let $T(n)$ be the number of assignments to the array C in Merge during MergeSort for an input of length n. There are n in the call to Merge, plus the two recursive calls on inputs of size $\frac{n}{2}$. That means

$$\begin{aligned}
T(n) &= 2T(\frac{n}{2}) + n \\
&= 2(2T(\frac{n}{4}) + \frac{n}{2}) + n \\
&= 4T(\frac{n}{4}) + n + n \\
&= 4T(\frac{n}{4}) + 2n
\end{aligned}$$

A little bit of fiddling with substitutions might let us see that this is, after k substitutions,

$$T(n) = 2^k T(\frac{n}{2^k}) + kn$$

So let $k = \log n$ and we get

$$T(n) = nT(1) + n \log n$$

And since the base case of recursion runs in constant time, we can see that this is $\Theta(n \log n)$!

We'll return to this approach with a bit more formality when we discuss the Master Theorem.