Lecture 16: MergeSort proof of correctness, and running time

Doina Precup
With many thanks to Prakash Panagaden and Mathieu Blanchette

February 10, 2014

1 Correctness of Merge

We would like to prove that MergeSort works correctly. To do this, let us first look at the correctness of the merge function, as this is iterative, and we have done proofs like this before. Recall that in such cases we want to find a **loop invariant** which is a condition that holds every time the internal loop (or loops) is executed, and helps us prove correctness. In this case, let us focus on the tmp array. At iteration k, suppose that the indices in the two parts of the array a are i and j. Then, our loop invariant will be: $tmp[k] \leq a[l], \forall l \in \{i, \ldots m\}$ and $tmp[k] \leq a[l], \forall l \in \{j, \ldots q\}$. In other words, the element we just copy at position k is the *minimum* of the remaining elements. Why does this condition help us? Since the tmp array is filled from left to right, all the elements left in a will be put in later, at positions $tmp[k+1], \ldots tmp[q-p+1]$. Hence, this condition means that in the end of the merge, $tmp[k] \leq tmp[l], \forall k < l$, which means that after copying, a will be correctly sorted between p and q.

combine

To see why the loop invariant is correct, recall that we assume that when merge is called, the two parts of a between p and m and between $m+1$ and q are already sorted. Hence:

a[p, q]
p = 5
q = 10
k = index of temp array, 1

$a[p,q]$ $[p, m]$
$$a[i] \leq a[l], \forall l \in \{i+1, \ldots m\}$$
merge

and

[0, 1] => sorted
[2, 6] == remaining positions to fill?

$[m+1, q]$
$$a[j] \leq a[l], \forall l \in \{j+1, \ldots q\}.$$
Correctness

So $a[i]$ and $a[j]$ are the smallest remaining elements. If $a[i] \leq a[j]$, then $tmp[k]$ will get value $a[i]$, and we also have that $a[i] \leq a[l], \forall l \in \{j, \ldots q\}$, The similar reasoning holds in the other case.

2 Correctness of MergeSort

Now that we know Merge works correctly, we will show that the entire algorithm works correctly, using a **proof by induction**. For the base case, consider an array of 1 element (which is the base case of the algorithm). Such an array is already sorted, so the base case is correct.

For the induction step, suppose that MergeSort will correctly sort any array of length less than n. Suppose we call MergeSort on an array of size n. It will recursively call MergeSort on two arrays of size $n/2$. By the induction hypothesis, these calls will sort these arrays correctly. Hence, after the recursive calls, array a will be sorted between indices $p, \ldots m$ and $m+1, \ldots q$ respectively. We have already showed that merge works correctly, hence after executing it, a will be correctly sorted between p and q. This concludes our proof.

Recall that when you design recursive algorithms, you have to "put your faith" in the recursion, assume it will work, then specify the processing that follows it. Induction basically gives you the mathematical tool to prove that your "faith leap" is indeed justified.

3 Time and space complexity of Merge

The Merge function goes sequentially on the part of the array that it receives, and then copies it over. So the complexity of this step is $O(q-p+1)$. To see this, note that either i or j *must* increase by 1 every time the loop is visited, so each element will be "touched exactly once in the loop. Note that the space complexity is also $O(q-p+1)$, as Merge has to allocate the temporary array tmp, in which elements get copied. This is actually a big disadvantage of the MergeSort algorithm: if the array to be sorted is very big (e.g. the 10,000,000 customers of some company), the memory cost becomes prohibitive.

4 Complexity of MergeSort

Let us think intuitively what the complexity of MergeSort might be. As seen, the Merge function goes sequentially on the part of the array that it receives, and then copies it over. So the complexity of this step is $O(p-q+1)$. MergeSort does two recursive calls, each on an array which is half the size of the original.

Length after merge

In order to get a better handle of what the resulting complexity might be, suppose that we denoted by $T(n)$ the amount of time that MergeSort uses on an array of size n. Recall that executing a *sequence* of instructions will cause us to *add* the running time. Hence, the running time will obey the following equation:

$$\boxed{T(n) = T\left(\frac{n}{2}\right) + T\left(\frac{n}{2}\right) + T_{merge}(n)} = 2T\left(\frac{n}{2}\right) + cn$$

where c is a constant, reflecting the actual number of basic operations (comparisons, tests, arithmetic operations, assignments) in the merge routine. An equation like the one above, where we have a function $T(n)$ defined based on values of T at other points $k < n$, is called a **recurrence**. Note that recurrences are naturally generated by recursive algorithms. to compute the running time, we would like to get a closed formula for T, or at least figure out what its fastest-growing term is (so that we can figure out $O()$ for the algorithm).

To get a better grip on the problem, let us unwind T for a couple more steps:

$$\begin{aligned} T(n) &= 2T\left(\frac{n}{2}\right) + cn \\ &= 2\underbrace{\left(2T\left(\frac{n}{4}\right) + c\frac{n}{2}\right)}_{T(n/2)} + cn = \boxed{2^2T\left(\frac{n}{4}\right) + 2cn} \end{aligned}$$

$$= 2^2 \underbrace{\left(2T\left(\frac{n}{8}\right) + c\frac{n}{4}\right)}_{T(n/4)} + 2cn = \boxed{2^3 T\left(\frac{n}{8}\right) + 3cn}$$

How many times can we continue this expansion? Until we get to $T(1)$ which is 1 (this corresponds to the base case of running on an array of size 1). Since in each step we halve n, we will reach $\boxed{T(1) \text{ is } \log_2 n \text{ steps.}}$ At that point, we will have:

$$T(n) = \boxed{2^{\log_2 n} T(1) + cn \log_2 n}$$

The first term above is $O(n)$, the second is $O(n \log n)$, so the whole algorithm is $O(n \log n)$.

Note that the "unrolling" is meant to give us an idea of the running time. To really establish it, we would need to prove now by induction that the recurrence indeed holds.

As a second example (done by request in class), consider the recursive algorithm for finding the maximum in an array (discussed in a previous lecture). Here, the recurrence is:

$$\begin{aligned} T(n) &= T(n-1) + c \\ &= T(n-2) + c + c = T(n-2) + 2c) \\ &= T(n-3) + c + 2c = T(n-3) + 3c \\ &= \ldots \\ &= T(1) + c(n-1) \end{aligned}$$

which yields $O(n)$ (as we know already).