Guide to Divide-and-Conquer

Thanks to Julie Tibshirani for helping with this handout!

This handout contains a sample divide-and-conquer problem and a complete solution so that you can get a better sense for what we're expecting on the problem set. As you'll see, the correctness proofs of divide-and-conquer algorithms tend to be proofs by induction, and runtime analyses often cite the Master Theorem.

A Sample Problem: Finding the Non-Duplicate

This question has been floating around as a tricky job interview question for about a year now. I apologize if it doesn't have much practical relevance beyond helping you get a high-paying job with lots of nice perks. ☺

> You are given a sorted array of numbers where every value except one appears exactly twice; the remaining value appears only once. Design an efficient algorithm for finding which value appears only once.

Here are some example inputs to the problem:

```
1 1 2 2 3 4 4 5 5 6 6 7 7 8 8

10 10 17 17 18 18 19 19 21 21 23

1 3 3 5 5 7 7 8 8 9 9 10 10
```

Clearly, this problem can be solved in O(n) time, where n is the number of elements in the array, by just scanning across the elements of the array one at a time and looking for one that isn't paired with the next or previous array element. But can we do better?

Since we know that the array is sorted, we might want to think about approaching this problem using some sort of binary-search-like algorithm. We can't use an exact copy of binary search to solve this problem, though, because we don't know what value we're looking for.

The key insight needed to solve this problem is the following: suppose you look at the pairs of elements at positions (0, 1), (2, 3), (4, 5), etc. All pairs that appear before the singleton element must consist of two copies of the same value. If we look at the pair containing the singleton value, then the first and second element of the pair will be different, with the singleton element at the first position. Importantly, if we look at any pair *after* the singleton element, the values in that pair will be different, since the pair containing the singleton will have "stolen" an element from the pair that appears after it, shifting everything down by one position.

Let's see how to formalize this into an algorithm. We can pick a pair of elements close to the middle of the the array and check whether the two elements there are equal or different. If they're equal, we can discard that pair and all pairs that come before it, since the singleton element must come after it. If they're different, we can discard the second element of that pair and everything that comes after it, since we know that the second element is part of a pair and that the singleton is either the first element of the pair or comes before it. Finally, when we get down to one element left, we know that element will be the singleton. We'd expect this to run in time O(log n), since we're discarding about half the elements on each iteration. Let's begin by formalizing the algorithm:

> **Algorithm:** Let A be our array and let its length be $n = 2k + 1$ (since it consists of k pairs and one singleton). If $n = 1$, return A[0]. If $n > 1$, compare A[$2\lfloor k / 2 \rfloor$] and A[$2\lfloor k / 2 \rfloor + 1$] (using zero-indexing). If the elements are equal, recursively apply this algorithm to the subarray beginning at position $2\lfloor k / 2 \rfloor + 2$ and extending to the end. Otherwise, recursively apply this algorithm to the subarray starting at the beginning of the array and extending to $2\lfloor k / 2 \rfloor$, inclusive.

Now that we have a formal version of the algorithm, we need to prove that the algorithm works correctly. This is a lot trickier than it might initially appear to be. In order to show correctness, we need to show that we can determine in which half of the array the singleton element appears simply by looking at the elements at positions $2\lfloor k / 2 \rfloor$ and $2\lfloor k / 2 \rfloor + 1$. Intuitively:

1. If the singleton appears after positions $2\lfloor k / 2 \rfloor$ and $2\lfloor k / 2 \rfloor + 1$, then all the elements before the singleton would be paired up correctly. Therefore, A[$2\lfloor k / 2 \rfloor$] = A[$2\lfloor k / 2 \rfloor + 1$], so we can discard the first half of the array.
2. If the singleton appears at or before position $2\lfloor k / 2 \rfloor$, then all the elements after the singleton would be mismatched. Therefore A[$2\lfloor k / 2 \rfloor$] ≠ A[$2\lfloor k / 2 \rfloor + 1$], so we can discard the second half of the array (but not A[$2\lfloor k / 2 \rfloor$], since it might be the singleton).

Based on these observations, we might want to try to prove the following lemma:

Lemma: A[$2m$] ≠ A[$2m + 1$] iff the position of the singleton (denoted s) satisfies $s \leq 2m$.

For now, let's hold off on proving this statement (we're pretty sure that it's true). Instead, let's see if we can prove that the algorithm is correct under the assumption that this lemma holds. If it doesn't, then we probably shouldn't waste our time trying to prove it!

> **Useful Tip #1:** Before proving any lemmas, confirm that you can prove that your algorithm is correct under the assumption that those lemmas are true. If so, go back and prove the lemmas. If not, discard or modify the lemmas.

Since our algorithm is inherently recursive, we are probably best off trying to prove it correct with induction, showing that the correctness on smaller inputs guarantees correctness on larger inputs. The algorithm is supposed to find the singleton element, so we should prove this is so:

Theorem: Given an array of size $2k + 1$, the algorithm returns the singleton element.

Proof: By induction on k. As a base case, when $k = 0$, the array has length 1 and the algorithm will return the only element, which must be the singleton. For the inductive step, assume that for some k that the claim holds for all $0 \leq k' < k$. We will prove it holds for k.

If A[$2\lfloor k / 2 \rfloor$] = A[$2\lfloor k / 2 \rfloor + 1$], then by our lemma, we have $s \geq 2\lfloor k / 2 \rfloor + 1$. Because A[$2\lfloor k / 2 \rfloor + 1$] is duplicated, this means $s \geq 2\lfloor k / 2 \rfloor + 2$. In this case, our algorithm recursively invokes itself on the subarray starting at position $2\lfloor k / 2 \rfloor + 2$. This subarray contains $2k + 1 - 2\lfloor k / 2 \rfloor - 2 = 2(k - \lfloor k / 2 \rfloor - 1) + 1$ elements. Therefore, by the inductive hypothesis, the recursive call returns the singleton element, so the algorithm returns the singleton element.

Otherwise, A[$2\lfloor k / 2 \rfloor$] ≠ A[$2\lfloor k / 2 \rfloor + 1$], so by the lemma, $s \leq 2\lfloor k / 2 \rfloor$. The algorithm then recursively invokes itself in the subarray ending at position $2\lfloor k / 2 \rfloor$, which has $2\lfloor k / 2 \rfloor + 1$ elements in it. Therefore, by the inductive hypothesis, the recursive call returns the singleton element, so the algorithm returns the singleton element. This completes the induction. ■

Notice how this proof worked via strong induction – we knew that we're going to make a recursive call to some smaller problem, but we weren't sure how small that problem would be.

> **Useful Tip #2:** Use strong induction (also called *complete induction*) to prove divide-and-conquer algorithms are correct.

Now that we have this theorem in tow, we should come back to prove that the lemma is true. As a reminder, our lemma is

Lemma: $A[2m] \neq A[2m + 1]$ iff the position of the singleton (denoted s) satisfies $s \leq 2m$.

This is a biconditional, so we can try thinking about how we'd show each direction of implication.

Let's start with the forward direction: if $A[2m] \neq A[2m + 1]$, then $s \leq 2m$. Why is this true? Intuitively, if these elements aren't paired up, then the singleton element has to be before them or equal to one of these two elements. In the latter case, this is easy to show. In the former case, we would somehow need to argue that the singleton had to have come earlier in order to offset the elements. This is true, but difficult to prove.

Let's try a different tactic: the contrapositive of this statement is

$$\text{If } s > 2m, \text{ then } A[2m] = A[2m + 1]$$

In other words, if the singleton comes after position $2m$, then $A[2m]$ and $A[2m + 1]$ would have to be paired with one another. The intuition here is a bit easier to work with: if the singleton appears after position $2m$, then everything before it must be paired up. We could easily prove that this is true using induction, since

- The first two elements must be paired up, since neither is the singleton.
- If the first $2r$ elements (where $2r \leq 2m < s$) are paired up and the element at position at $2r$ isn't the singleton, it has to be paired with the element at position $2r + 1$ or $2r – 1$. By the IH, the element at position $2r – 1$ is already paired up, so it has to be paired up with the element at position $2r + 1$.

So now we have to think about the reverse direction: if the singleton appears at or before position $2m$, then $A[2m] \neq A[2m + 1]$. We can prove this in more or less the same way that we proved the previous part: use induction to show that if the singleton is at some position $2t$, then all pairs after position $2t$ will be mismatched. Based on this discussion, we arrive at this formal proof:

Proof: We prove both directions of implication.

> (⇒) By contrapositive; we will show that if $s > 2m$, then $A[2m] = A[2m + 1]$. To do this, we prove the stronger claim that $A[2r] = A[2r + 1]$ for all $0 \leq r \leq m$ by induction on r. As a base case, when $r = 0$, $A[0]$ can't be the singleton ($0 \leq 2m < s$), so $A[0] = A[1]$. Now suppose that for some r that the claim is true for all r' satisfying $0 \leq r' < r \leq m$. We will prove the claim is true for r. We know $A[2r]$ is not the singleton, since then $s = 2r \leq 2m$, contradicting that $s > 2m$. Therefore, $A[2r]$ must be equal to an adjacent array element. We cannot have $A[2r] = A[2r – 1]$ or then $A[2(r – 1)] \neq A[2(r – 1) + 1]$, so we must have $A[2r] = A[2r + 1]$. Thus the claim holds for r, completing the induction.
>
> (⇐) Suppose $s \leq 2m$. We claim that s must be even; if s were odd (say, $s = 2t + 1$), then we have $A[2t] \neq A[2t + 1]$, and by (⇒) this would mean $s \leq 2t < 2t + 1$, a contradiction. So s is even; let $s = 2t$. Then since all elements after the singleton must be paired, a quick inductive argument shows that $A[2t + 2k] \neq A[2t + 2k + 1]$ for all $k \geq 0$. Setting $k = m – t$ completes the proof. ■

All that's left to do at this point is to analyze the runtime. We've spent a lot of time talking about how to solve recurrence relations, so we can try to set up a recurrence relation for this problem.

There's one catch – the algorithm works by writing $n = 2k + 1$ and then chooses a subproblem whose size is determined from k rather than from n. This makes writing out a precise recurrence relation in terms of n tricky. But that's nothing to worry about: since $n = 2k + 1$, if we write out a recurrence relation in terms of k and find that $T(k) = O(f(k))$ we can conclude that the runtime as a function of n is $O(f((n - 1) / 2))$, which, if f is well-behaved, should be easy to work with.

> **Useful Tip #3:** It's sometimes useful to define recurrence relations in terms of variables other than n. Just make sure you explain how to use the solution to the recurrence to learn something about the runtime as a function of n.

This gives the following runtime analysis:

Runtime: Our algorithm's runtime is given by the following recurrence in terms of k:

$$T(0) = \Theta(1)$$
$$T(k) \leq T(\lfloor k / 2 \rfloor) + \Theta(1)$$

By the Master Theorem, this solves to $T(k) = O(\log k)$. Since $n = 2k + 1$, this means that the runtime as a function of n is $O(\log n)$.