Divide and Conquer

Hello, welcome to competitive programming.

One of the most iconic problem solving techniques in computer science is divide and conquer.

Objectives

Your objectives are to understand how to keep hand-written quicksort implementations from exploding on bad input, and how to use binary search in more interesting ways than simply searching an array.

Binary Search

The classic idea of divide an conquer is that you break a problem into smaller pieces. There are two common forms this takes. The first is when you process two halves of the problem space and combine those solutions to solve the original problem. Classic examples include algorithms like quicksort and mergesort.

Another form this takes is when you divide the problem and prune some of the subsets. These are typically search problems, like binary search.

Sorting Considerations

Here are a few things to think about when sorting. You probably know that quicksort gives you bad behavior if you pick the first element of the array as a pivot. This is

because data that happens to be sorted already is fairly common. Picking a random pivot is better, but there is still a trick that problem setters can throw at you if you are not careful.

One horrible trick is to have an array where maybe 90 % of the elements are the same. In this case, a normally written quicksort will still give you O(n^2) behavior. So if you really need to write your own quicksort, you will need to partition the space into three partitions to be efficient.

If you are using C++, note that the algorithm include provides `sort`. It can handle already sorted data and data with all the same value just fine, so you should really use that if you can.

The biggest reason you might not want to use that is if you need a stable sort. You can modify quicksort to be stable if you have the memory to allocate a second array.

Other binary search

You all know how to use binary search to find an element in a sorted array, but there's another application of binary search. Sometimes there is a function you cannot compute directly, so you will need to make a guess and refine the guesses to converge on the proper value.

Newton's method has this pattern. The example problem you will get to solve for class also has this property. This is commonly used when you have to find a floating point number as your answer. The technique is to set a parameter Epsilon to something very small. When the next iteration of your answer is less than Epsilon from the previous answer, then you can consider it to be close enough. You never want to ask if two floating point numbers are exactly equal; round-off error will make it so even a correct solution gets missed.

Here is some general code for searching for a solution using binary search. The assumption is we want the minimum value of `f` that satisfies the requirement. Set

`lo` and `hi` to be your search range, and repeatedly call your simulation function until you converge on the value.

Next
Transcript — Dynamic Programming →