Divide & Conquer—Monday, February 10/Tuesday, February 11

Readings

- [Lecture Notes Chapter 7: Divide & Conquer and Recurrence Relations](#)

Review: Divide & Conquer and Recurrences

What does it mean to have a “Divide & Conquer” algorithm?

- **Divide** the problem into a number of subproblems that are smaller instances of the same problem.
- **Conquer** the subproblems by solving them recursively.
- **Combine** the solutions to the subproblems into the solution for the original problem.

How do you recognize situations where “Divide and Conquer” might work? A natural first question is “Can I break this down into subproblems equivalent to the original problem?” You can then ask “How can I solve these problems and combine them to reach a solution for my original problem?” Usually if you can solve each subproblem and combine them, it involves some sort of recursion. In order to better understand the “Divide & Conquer” paradigm, we will do an in depth study on a familiar algorithm: **Mergesort**.

Mergesort and Divide & Conquer

We can apply the principles of Divide and Conquer when thinking about approaching this problem.

- **Divide** Can we divide this into equivalent subproblems? Yes, we can divide this array into two halves each with \( \frac{n}{2} \) elements. Thus, each is an equivalent subproblem.
- **Conquer** How can we recursively sort the two halves? That’s easy! Since we already broke it into subproblems, we will recurse using Mergesort on the two halves until we hit the base case of a singleton element (We trivially know that a singleton element is sorted).
- **Combine** Once we have two sorted arrays we can combine them in \( O(n) \) time by interleaving the two halves!

How long does Mergesort take to run?

Let \( T(n) \) represent the time the algorithm takes for an input of size \( n \). Since the two halves are sorted recursively by the same algorithm, but with inputs that are each half the size of the original, each half should take time \( T(\frac{n}{2}) \). The merging takes linear time. So we can write \( T(n) = 2 \cdot T(\frac{n}{2}) + cn \) for some constant \( c \). From lecture, we know this is \( \Theta(n \log n) \). As you have seen in class, **recurrences** are equations that can help us describe the running time of a recursive algorithm.
Problem 1. Local Maxima

You are given an integer array $A[1..n]$ with the following properties:

- Integers in adjacent positions are different


Example: You have an array $[0, 1, 5, 3, 6, 3, 2]$. There are multiple local maxes at 5 and 6.

Propose an efficient algorithm that will find a local maximum and return its index.

Solution.

A naive solution to this problem is to simply iterate through the array and check each element with its neighbors until we find a possible solution, which runs in linear time. Instead, we aim for a more efficient way of solving this problem.

The first thing to notice is that given the properties of the array, there must be a local maximum. A simple proof of this is to consider a maximum element in the array. Since neighbors are distinct and since the endpoints cannot be maximum, we are guaranteed that this element will be a local maximum.

If we want to approach this problem with the Divide & Conquer methodology, we need to figure out how we can cut the problem into subproblem(s). We do this by cutting the problem in half every time.

We let our base case for this algorithm be when the array is of size 3, in which case we return the middle element index. Otherwise, consider the middle element of the array, say at index $m$. If that element meets the requirements of a local maximum, then we return $m$. If not, notice that this element must have a larger neighbor, WLOG say the right neighbor is larger.

Observe that the subarray $A[m...n]$ also satisfies the given property of the input array—the endpoints are smaller than their neighbors and every pair of adjacent elements are distinct. Therefore, by our claim above, the subarray must contain a local maximum. Since we are strictly reducing this problem at every recursive call, we must eventually hit our base case (which we know to be correct) or terminate early by finding a local maximum.

The running time of our algorithm is thus $T(n) = T(\frac{n}{2}) + O(1)$. Solving this recurrence yields $O(\log n)$.
Problem 2. Element Index Matching

You are given a sorted array of \( n \) distinct integers \( A[1...n] \). Design an \( O(\log n) \) time algorithm that either outputs an index \( i \) such that \( A[i] = i \) or correctly states that no such index \( i \) exists.

Solution.

A naive solution would be to iterate through the array, checking if \( A[i] = i \) at every stage. This runs in \( O(n) \), but we can probably do better. Since we have a sorted array and an \( O(\log n) \) runtime constraint, a modified binary search seems to be a good choice.

Notice that at any index \( i \), if \( A[i] \neq i \), we can narrow down the possibilities of where a potential candidate lies. More specifically, if \( A[i] < i \), since integers are distinct and sorted, it’s impossible to find an index \( m < i \) such that \( A[m] = m \). We can see this as follows:

\[
A[i - k] \leq A[i] - k < i - k
\]

The same reasoning applies to the right side when \( A[i] > i \). Therefore, letting \( i \) be the middle index, we can be sure that the specified half will never contains an index-matching element, and instead recurse on the other.

For each stage of our algorithm, we are performing a constant time comparison, and then dividing our problem in half. Thus, our recurrence is \( T(n) = T(n/2) + O(1) \), which we know is \( O(\log n) \) from binary search.

Problem 3. Largest Subarray Sum

Given an integer array (containing both positive and negative values), return the sum of the largest contiguous subarray which has the largest sum.

Solution.

One naive way to solve this problem is to use two for-loops. The outer loop runs through the elements in the array, while the inner loop finds the maximum contiguous subarray sum starting at the current outer loop element. If this sum is bigger than the best running maximum, we update and continue through the process. This runs in \( O(n^2) \).

A better solution is to apply our knowledge of the Divide & Conquer approach, and see if we can find a more efficient solution to this problem!

One thing that we can intuitively notice is that the optimal sub-sequence either lies in the left half of the array, the right half of the array, or runs along the center of the array and cuts through the middle element. Logically, these are the only three options we have. Thus, we can compute all three of these values and the maximum of them will be our solution!

To do this, we must recursively divide the array into two halves and find the maximum subarray sum in both halves. This can be done easily with two recursive calls. Lastly, we need to efficiently compute the maximum cross sum. This can be done in \( O(n) \) time by starting at the middle element and calculating the maximum sum to the left of the middle element, doing the same with the right and combining the two!

To calculate the run time of our algorithm, we notice that it breaks down the work into two sub problems, each with half the size as input and then checks the cross sum in linear time. Thus, we get the following recurrence: \( T(n) = 2T(n/2) + O(n) \).

Does this look familiar? It should! It’s the same recurrence you saw for the running time analysis of Mergesort! This evaluates to \( O(n \log n) \).