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.
Problems

Problem 1. Local Maxima
You are given an integer array \( A[1..n] \) with the following properties:

- Integers in adjacent positions are different

A position \( i \) is referred to as a local maximum if \( A[i−1] < A[i] \) and \( A[i] > A[i+1] \). You may assume \( n > 2 \).

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.

Problem 2. Element Index Matching
You are given a sorted array of \( n \) distinct integers \( A[1...n] \). Design an \( O(\lg n) \) time algorithm that either outputs an index \( i \) such that \( A[i] = i \) or correctly states that no such index \( i \) exists.

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.