CIS 1210—Data Structures and Algorithms—Fall 2023

Divide & Conquer—Tuesday, September 19th / Wednesday, September 20th

Readings

- Lecture Notes Chapter 7: Divide & Conquer and Recurrence Relations

Review: Divide & Conquer and Recurrence Relations

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

> **Divide** the original problem into subproblems that are smaller-sized instances of the same problem.
> **Conquer** the subproblems by solving them with recursion.
> **Combine** the solutions to the subproblems into the solution for the original problem.

How do you recognize problems where "Divide & 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 subproblems and *combine* these solutions to reach a solution for my original problem?" In order to better illustrate the "Divide & Conquer" paradigm, we will do an in-depth study on a familiar algorithm: **Mergesort**.

Applying Divide & Conquer to Mergesort

We can apply the principles of Divide & Conquer when thinking about the problem of sorting an array:

Divide Can we divide this problem into equivalent subproblems? Yes, we can divide the array into two halves, each half a subarray of size $\frac{n}{2}$. Thus, each half of the array is an equivalent subproblem.

Conquer How can we solve our subproblems? That is, how can we sort both halves of the array? Since each half of the array is an equivalent subproblem, we can sort each half by recursively calling Mergesort on each half of the array until we hit our base case of a subarray with a single element, which we know by definition is already sorted. Note that when applying "Divide & Conquer," our base case must also be an equivalent subproblem.

Combine How can I combine the solutions to my subproblems into a solution for the original problem? After Mergesort has finished recursing on both halves of the array, we have two sorted subarrays of size $\frac{n}{2}$. We can leverage how both halves of the array are already sorted, combining them into a sorted array to solve the original problem by interleaving the two halves in $O(n)$ time.

Runtime Analysis of Mergesort

As you have seen in lecture, **recurrence relations** are equations used to describe the running time of a recursive algorithm. To derive recurrence relations for "Divide & Conquer" algorithms, we usually need to consider the size of subproblem(s) recursed on; the number of recursive calls made; the amount of work done per "level" to combine the solutions to our subproblems; and the amount of work done at the base case.

Let $T(n)$ represent the time Mergesort takes on an input of size n. The algorithm divides the original problem into 2 subproblems of size $\frac{n}{2}$ — each of which takes $T(\frac{n}{2})$ time to solve since each is half the size of the original problem — and makes 2 recursive calls to Mergesort, one call to solve each subproblem. At each level, we perform linear work by merging the two sorted halves of the array; at the base case, we perform

constant work by returning a singleton array. Thus, the recurrence relation for Mergesort can be expressed as

$$T(n) = \begin{cases} 2T(\frac{n}{2}) + cn & n \geq 2 \\ 1 & n = 1 \end{cases}$$

Solving this yields a running time of $\Theta(n \log n)$.

Problems

Problem 1: 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 2: Local Maximum

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

- Integers in adjacent positions are different
- $A[1] < A[2]$
- $A[n-1] > A[n]$

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.

Design an $O(\lg n)$ algorithm to find a local maximum and return its index.

Problem 3: Largest Continuous Sum

You are given an integer array, with both positive and negative elements. Design an $O(n \lg n)$ algorithm to return the sum of the continuous subarray with the maximum sum.