# CIS 121—Data Structures and Algorithms with Java—Spring 2018

**Divide** & **Conquer**—Monday, February 5th / Tuesday, February 6th

## Learning Goals

During this lab, you will:

- Develop intuition for the divide and conquer paradigm
- Review the mergesort algorithm

## Divide and Conquer: An Overview

What does it mean to have a "divide and 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 and conquer" paradigm, we will do an in depth study on a familiar algorithm: Mergesort.

### Mergesort

Problem statement: Given an array of length $n$, sort it in ascending order (could also be descending). You cannot assume anything about the contents of the array.

We can apply the three principles of divide and conquer when thinking about approaching this problem.

*Divide*: First we ask ourselves: Can we divide this into equivalent subproblems? Yes, sorting two halves of the array of size $\frac{n}{2}$ each is an equivalent subproblem.

*Conquer*: How can I recursively sort the two halves of the array? That's easy—since I already broke it up into subproblems, I will recurse using mergesort on the two halves, until I hit the base case. The base case of a singleton element means the array is sorted.

*Combine*: Once I have two sorted arrays, I can combine them into a larger array by interleaving them.

I encourage you to review your notes on the pseudocode of the algorithm and identify which parts of the code correspond to these categories. Many algorithms of this sort have a similar structure, so intimately understanding this one will help you build your own divide and conquer algorithms later.

Proof of running time: We have a lot of practice with this! This is a straightforward recurrence relation. With an array of size $n$, you solve it by recursing on both halves of the array (size $\frac{n}{2}$. In order to combine it, you interleave those two sorted arrays, which takes $O(n)$ time. Therefore, the recurrence relation is $T(n) = 2T\left(\frac{n}{2}\right) + n$. Employ whatever method of solving recurrences that you are comfortable with, and you will get $O(n \lg n)$.

Proof of correctness:

Author's Note: I highly recommend reading the CLRS section 2.3.1 for an excellent overview of this topic.

## Problems

### Local maximum

**Problem 1.** You are given an integer array with the following properties:

- Integers in adjacent positions are different
- $arr[0] < arr[1]$
- $arr[arr.length - 2] > arr[arr.length - 1]$

A position $i$ is referred to as a local maximum if $arr[i] > arr[i-1]$ and $arr[i] > arr[i+1]$.

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.* While we can solve this problem quite easily in linear time by running through the array and checking each element with its neighbors until we find a possible solution, there is a more efficient way of solving this problem.

The first thing to notice is that there *must* be at least one local maximum in the array. Why? Every array has a maximum element, and by definition, this element is also a local max.

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). Does this remind you of any technique? (Hint: binary search.) Of course, we need to apply a slight modification to the original binary search algorithm.

Let's take the middle element of the array. If that element meets the requirements for a local maximum, then great, we are done. We can rejoice and go sleep soundly. However, this may not be the case. What can we do? We can compare the element to its rightmost neighbor (let's call it $x$). If $x$ is bigger than our current element, what do we know? Well, there must be a solution in the right half of the array! Why? Either $x$ is a local max or it is not. If it is not, then there must be a larger element in the half of the array containing $x$. Since every array must have at least one local max, this half array must also have at least one local max, and we can focus our search on that half. The reasoning is symmetric if the element left to $x$ is larger.

The running time of our algorithm is thus $T(n) = T(\frac{n}{2}) + O(1)$. Solving this recurrence yields $O(\lg n)$ (it is binary search after all!). □

### Maximum subarray sum

**Problem 2.** Given an integer array (contains positive and negative values), return the sum of the largest contiguous subarray which has the largest sum.

*Solution.* One way to solve this problem (naive method), is to use two loops. The outer loop runs through the elements in the array, while the inner loop finds the maximum sum given 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 median, doing the same with the right and combining the two!

Let us look at finding the maximum sum to the right of the median more closely. We can start by keeping a current sum, and a current max sum. Increment a pointer from the median to the right. At every element, add it to our current sum, and if it is greater than the current max sum, update it.

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: $T(n) = 2T(\frac{n}{2}) + O(n)$.

Does this look familiar? It should! It's the same recurrence you saw for the running time analysis of MergeSort! This evaluates to $O(n \lg n)$. □

## Element index matching

**Problem 3.** 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.

*Solution.* We can modify binary search to solve this problem. Note that for a given array $A$ and middle index $m$, if $A[m] > m$, then the matching element index, if it exists, must be in the left half of the array. This is because if $A[m] > m$, every successive index must have values at least one greater than the previous (distinct and sorted integers), and so no element can satisfy our condition. The reasoning is symmetric if $A[m] < m$, and the matching element index, if it exists, must be in the right half of the array.

This algorithm has the same running time as binary search, $O(\lg n)$. □

## Counting duplicates

**Problem 4.** Given an unsorted array of $n$ integers, design an algorithm to remove all the duplicate elements.

*Solution.* We can base our algorithm off of mergesort, by modifying the merge step. We run mergesort as normal, but during the merge step, we check to see if the two values we are comparing are equal. If they are, then the first time we see this element, we add it to our final list. We then increment the pointers to the two subarrays we are merging until we encounter a new element.

This can also be done by sorting the list, and doing one pass on the sorting list, removing elements that are equal to its neighbor. □

## Maximum Profit

**Problem 5.** Given, in an array, the prices of a stock for each day for $n$ days, what is the maximum profit you can make with exactly one buy transaction and one sell transaction?

Design a divide and conquer algorithm to solve this problem.

*Solution.* We can divide the problem into subproblems by splitting the array into two equal subarrays. The three scenarios are as follows. Both the buy date and sell date occur in the first half, both the buy date and sell date occur in the second half, and the buy date occurs in the first half while the sell date occurs in the second half.

To complete the conquer step, we can recursively solve the problem for the two halves. Now, all is left is to complete the combine step. We still have to account for the third case, in which the buy and sell dates occur in separate halves. Note that the maximum profit that can be made if the two sell dates are in separate halves is if we buy during the minimum price of the first half and sell during the maximum price of the second half. This can be found in a single pass through each of the halves. Finally, we compare the maximum profit from the three cases, and return the largest of the three values.

Now let us analyze the runtime of this algorithm. The combine step takes time proportional to $n$. We have two subproblems of size $\frac{n}{2}$. Thus our recurrence is $T(n) = 2T(n/2) + cn$. This is the same recurrence as mergesort - which we know simplifies to $O(n \lg n)$.

Note that this isn't the most optimal solution to this problem - it is actually possible to solve it in $O(n)$ time! We keep track of two things - the minimum element seen so far, and the current max profit so far. We iterate through the array from left to right. For each element, we check the current profit by finding the difference between the current element and our current min. If it is greater than our current stored max profit, we will update it. Then, if the element we're currently at is less than our stored min, we update that. By the time we reach the end of the array, our current max profit is the solution. □