Topic Notes: Divide and Conquer

Divide-and-Conquer is a very common and very powerful algorithm design technique. The general idea:

1. Divide the complete instance of problem into two (sometimes more) subproblems that are smaller instances of the original.
2. Solve the subproblems (recursively).
3. Combine the subproblem solutions into a solution to the complete (original) instance.

While the most common case is that the problem of size n is divided into 2 subproblems of size $\frac{n}{2}$. But in general, we can divide the problem into b subproblems of size $\frac{n}{b}$, where a of those subproblems need to be solved.

This leads to a general recurrence for divide-and-conquer problems:

$$T(n) = aT(n/b) + f(n), \text{ where } f(n) \in \Theta(n^d), d \geq 0.$$

When we encounter a recurrence of this form, we can use the *master theorem* to determine the efficiency class of T:

$$T(n) = \begin{cases} \Theta(n^d) & \text{if } a < b^d \\ \Theta(n^d \log n) & \text{if } a = b^d \\ \Theta(n^{\log_b a}) & \text{if } a > b^d \end{cases}$$

Application of this theorem will often allow us to do a quick analysis of many divide-and-conquer algorithms without having to solve the recurrence in detail. We will leave the mathematical proof of the master theorem to the mathemeticians.

Mergesort

Each sorting procedure we have considered so far is an "in-place" sort. They require only $\Theta(1)$ extra space for temporary storage.

Next, we consider *merge sort*, a divide-and-conquer procedure that uses $\Theta(n)$ extra space in the form of a second array.

It's based on the idea that if you're given two sorted arrays, you can merge them into a third in $\Theta(n)$ time. Each comparison will lead to one more item being placed into its final location, limiting the number of comparisons to $n - 1$.

In the general case, however, this doesn't do anything for our efforts to sort the original array. We have completely unsorted data, not two sorted arrays to merge.

But we can create two arrays to merge if we split the array in half, sort each half independently, and then merge them together (hence the need for the extra $\Theta(n)$ space).

If we keep doing this recursively, we can reduce the "sort half of the array" problem to the trivial cases.

This approach was invented by John von Neumann in 1945.

We will work through an example, look at the algorithm, and analyze it on the mergesort handout.

Quicksort

Another very popular divide and conquer sorting algorithm is the *quicksort*. This was developed by C. A. R. Hoare in 1962.

Unlike merge sort, quicksort is an in-place sort.

While merge sort divided the array in half at each step, sorted each half, and then merged (where all work is in the merge), quicksort works in the opposite order.

That is, quicksort splits the array (which takes lots of work) into parts consisting of the "smaller" elements and of the "larger" elements, sorts each part, and then puts them back together (trivially).

It proceeds by picking a *pivot* element, moving all elements to the correct side of the pivot, resulting in the pivot being in its final location, and two subproblems remaining that can be solved recursively.

We will work through an example, look at the algorithm, and analyze it on the mergesort handout.

Quicksort is often the method of choice for general purpose sorting with large data sizes.

Computational Geometry

We return to two familiar problems from computational geometry to explore divide-and-conquer solutions that are more efficient than the brute force approaches considered previously.

Divide-and-Conquer Closest Pairs

Our problem is to find among a set of points in the plane the two points that are closest together.

We begin by assuming that the points in the set are ordered by increasing x coordinate values. If this is not the case, the points can certainly be sorted in $\Theta(n \log n)$ time as a preprocessing step. As long as the rest of our procedure is at most $\Theta(n \log n)$, it will not increase the overall complexity.

We then divide the points into two subsets, S_1 and S_2, each of which contains $\frac{n}{2}$ points (which is easy since the points are sorted by x coordinate).

We then recursively find the closest pair of points in each subset S_1 and S_2. If the distance between the closest pair in $S_1 = d_1$ and the distance between the closest pair in $S_2 = d_2$. We then know that $d = \min\{d_1, d_2\}$ is an upper bound on the minimum distance between any pair, but we still need to make sure we check for shorter distances where one point is in S_1 and the other is in S_2.

The only points which might be closer together than distance d are those within a strip of width d from the dividing line between the subsets. For each point within that strip and within one subset, we potentially need to consider all points from within the strip within the other subset. That still sounds like a lot of work. The key observation is that for each point on one side, we have to consider points whose y coordinates are within d. This will mean at most 6 points from the other side, since if there are more points than that within d in the y coordinate on the other side, at least one pair from among that point would be closer than distance d to each other.

So how do we find those points to check? They can be found quickly if we also keep the points sorted in order by y coordinate. Still, this seems difficult but it can be done efficiently (see the text's description for details). Bottom line, we have a constant bound on the number of points to check (we can fairly easily show that each point need only be compared to up to 7 others) for each point in the boundary strip, so the "combine" operation ends up being linear time.

We end up with a recurrence:

$$T(n) = 2T(n/2) + O(n)$$

which given an overall time of $T(n) \in \Theta(n \log n)$.

Quickhull

The other computational geometry problem discussed in the text is called *quickhull* – an algorithm for finding the convex hull of a set of points. We will discuss the ideas (but not the details) and go through an example in class, but it is worth reading the section for all of the details.