16.4 Introduction to Divide-and-Conquer Algorithms

In the previous two sections, you learned about two different iterative sorting algorithms: selection sort and insertion sort. These two shared a similar structure, using a loop to build up a "sorted sublist" in its input list one element at a time. Both of them had a worst-case running time of $\Theta(n^2)$,[1] and so you might be wondering: are there sorting algorithms that

[1] Though remember, selection sort's running time is *always* $\Theta(n^2)$.

are faster than $\Theta(n^2)$?

The answer is yes! We'll now explore a different approach to designing sorting algorithms that uses recursion, and that have the potential to sort lists more efficiently than either selection sort or insertion sort.

A divide-and-conquer approach

Suppose you're tasked with sorting a very large collection of papers. Rather than sort them all yourself, you ask a friend to help you. When you get started, you divide up the collection in half, so that you and your friend each take one half to sort separately. Then when you're done, you combine your sorted halves together to result in the final sorted collection.

Intuitively, if we are given a large task to do, splitting that task up into subtasks can make it easier to manage, even if there's a bit of extra work at the end to combine the results of each subtask together. In computer science, we formalize this idea into a paradigm for algorithm design known as **divide-and-conquer** algorithms. The divide-and-conquer approach to algorithm design is the following:

1. Given the problem input, split it up into two or more smaller subparts with the same structure.
2. Recursively run the algorithm on each subpart separately.
3. Combine the results of each recursive call into a single result, solving the original problem.

That might seem a bit abstract, so let's phrase this in the context of sorting algorithms. A *divide-and-conquer* sorting algorithm consists of the following steps:

1. Given a list to sort, *split* it up into two or more smaller lists.
2. Recursively run the sorting algorithm on each smaller list separately.
3. *Combine* the sorted results of each recursive call into a single sorted list.

Note that the above steps work in the cases where we want the sorting algorithm to return a new sorted list (like `sorted`), or whether we want it to mutate its input (like `list.sort`).

In the next two sections, we'll explore two famous divide-and-conquer sorting algorithms: *mergesort* and *quicksort*. We'll see how these algorithms share the same structure, but differ in the complexity of each of "split" and "combine" steps described above.[2] These two algorithms are among the most famous recursive algorithms in computer science, and we're excited that you'll be learning about them!

[2] Notice the parallel to selection sort and insertion sort, where they differ in how complex the "selection" and "insertion" steps are.