CS 42—Sorting (and more analysis)

Tuesday, November 20, 2018

Summary

Today, we'll talk again about algorithmic strategies (e.g,. use-it-or-lose-it and tabulation) and analysis (e.g., recurrence relations and summations). Our goal will be to practice and become (even more) comfortable with these techniques. Then, we'll apply our analysis techniques to some algorithms in an exercise called "Sort Detective".

Sorting algorithms

A sorting algorithm takes as input a sequence of values whose elements are comparable (i.e., it's possible to compare any two elements to determine which one is less than the other or whether the two elements are equal). The algorithm produces as output the same sequence of elements, arranged in order.

When thinking about sorting, there are several things to take into consideration:

One problem and many solutions. There are many sorting algorithms, and each solves the problem of sorting in a different way. Additionally, for each algorithm, there might be many *implementations* of that algorithm (e.g., in different languages). These differences affect performance.

Is the algorithm in-place? An *in-place* algorithm uses minimal extra space. Instead, it modifies the input, shuffling the elements around until they're in order. An algorithm that is not in-place often creates and returns a new sequence of elements, leaving the original unchanged.

Is the algorithm adaptive? An *adaptive* sorting algorithm performs better on data that is almost sorted. An algorithm that is not adaptive performs the same (or worse) on almost-sorted data as it does on other kinds of data. In general, we should think about how sensitive an algorithm is to variations in its input.

Performance. How do we measure the performance of an algorithm, in theory and in practice? When coming up with a theoretical model, are we trying to model time, or space, or energy, or something else? If modeling time, are we counting comparisons, or accesses, or something else?

Next time: break (no class)