CSE250 Week 10: Trees (mostly...)

...But first, finishing the conceptual content of Chapter 15

QuickSort

The text soft-pedals what other texts highlight as two conceptual novelties of this algorithm:

1. Analysis of a deterministic algorithm based on random distributions of data.
2. Randomizing the algorithm itself, to take "bad-luck data" out of the mix.

First, the basic D&C format of the algorithm: It selects an element $p$ called the pivot from the array and then the partition step first rearranges the elements to look like this:

Here $p$ is the array index of the pivot element in the ultimate sorted order, which is what the partition step computes. The elements to its right and left are not necessarily sorted already, but they can be sorted in two recursive calls of quickSort: this is the D&C part. In the bad-luck case where $p$ happened to be chosen as the least element, we get $p=0$ so $p-1$ does not exist, whereupon we get just one recursive call on indices $p+1$ to $hi=n-1$. (The indices here are inclusive on both ends.) In case $p$ is unluckily picked as the largest element, things are similar in mirror image. At least progress to termination is guaranteed, but the edge-cases have no "D&C advantage."

Overall, the execution has the following characteristics:

- It sorts an array in-place via swaps, without wholesale copying.
- Alternately, it can sort a (linked-)list via cell rearrangements, thus avoiding the appearance of data copying, but this puts you in the simplest case of situation 1.
- It recurses on contiguous sub-ranges of the array (or rearranged pieces of the list), which again helps it work in-place.
- The $\Theta(n)$-time combination step happens before the recursive calls. So it is "D&C tail recursive", which means that all the work happens on the way "going down the stack" (text, p484 middle) rather than on the way up as with MergeSort.
- The recursive calls are not guaranteed to be on $\frac{n}{c}$ sized subsets, where $c > 1$ is a fixed
constant, let alone $c = 2$ as with MergeSort. But $c > \frac{3}{2}$ can be argued to hold "most of the time", which is enough to give overall running time $\Theta(n \log n)$ in most cases. "Bad luck", however, can yield cases where one of the two subsets steps down only to size $n - 1$ or $n - 2$ or so, which raises the absolute worst-case complexity to $O(n^2)$.

The first four points make the $\Theta(n \log n)$ time in the (good case of the) fifth point come with a low principal constant in practice. This is what puts the "Quick" into QuickSort. [And is how Sir C.A.R. "Tony" Hoare more than made up for his "billion dollar blunder" of inventing the null reference.]

The partition step is prefaced by selecting one element $p$ of the array as the pivot. We want $p$ to be as close to the median of the array as possible---but nobody knows how to compute the median in $O(n)$ time to begin with. So we resort to various levels of guessing-and-hoping:

a) Just pick $p$ to be the first element in the array (or list).

b) Pick $p$ to be the middle element of the array. (Not as friendly to lists.)

c) Pick $p$ at random from the array. This is what creates a randomized algorithm.

d) Pick $p$ to be the middle of three elements: usually the first, last, and middle.

Policy d) is called "Median-of-Three QuickSort" and is close to the most common policy. Policy a) is the quickest when it doesn't run afoul of bad data---weirdly, data that is already almost-sorted counts as a bad case here.

[Second half of lecture was a demo of code for MergeSort and QuickSort applied to the Fallows1898.txt dictionary. Some highlights---and lowlights:

- The non-tail recursive code "merge1" for merge causes a StackOverflow, just as the text warns. This is even though the array being sorted (after conversion to a List) has only 6,175 entries. It is not an infinite-recursion error; my lecture actually showed that it was making progress just before it bombed.
- The tail-recursive version "merge2"---as noted, basically identical to the text's code---works fine.
- QuickSort is about 4x quicker---when the pivot element is selected randomly.
- When the first element is used as the pivot, its time was similar to MergeSort. This is because Fallows1898.txt is an example of data that is already nearly-sorted. Only some words, notably "Vole", are out of place.
- Picking the middle element should have worked better, but didn't...hmmm...

The last bullet exemplifies why playing with code is a never-ending time sink unto itself. :-]