CIS 575. Introduction to Algorithm Analysis
Material for February 28, 2024

Selection by Divide & Conquer

The topic of this note is covered in *Cormen*'s Section 9.3.

1 Selection By Divide & Conquer

Recall the *selection* problem: given n numbers, find the kth smallest. Or to be more precise:

> Given n records, find a record whose key y is such that
> there are $< k$ records with key $< y$, but
> there are $\geq k$ records with key $\leq y$.

As our running example, we shall consider the 25 numbers:

37, 22, 42, 11, 17, 48, 12, 16, 20, 45, 61, 24, 47, 53, 33, 44, 35, 19, 10, 50, 13, 16, 30, 54, 23

and assume we want to find the 17th smallest element.
One way to do so is to apply one of our sorting algorithms; this will give

10, 11, 12, 13, 16, 16, 17, 19, 20, 22, 23, 24, 30, 33, 35, 37, 42, 44, 45, 47, 48, 50, 53, 54, 61

and we can find the 17th smallest number as the 17th element of that list (array): **42**
But the sorting algorithms we have seen so far all run in time $\Omega(n \lg(n))$, and it turns out (as we shall see later) that we cannot hope for a sorting algorithm (at least not one that is based on comparisons) with better asymptotic behavior. With the goal of getting a *linear*-time algorithm for selection, let us try to apply the *divide & conquer* paradigm.

1.1 Naive Divide & Conquer

The immediate approach is do to as in the *Quicksort* algorithm, except we need to explore at most *one* of the partitions. To be precise, we

1. choose a pivot p
2. apply the Dutch National Flag algorithm (DNF) to partition the numbers into
   (a) those that are $< p$ (called "Red" by DNF)
   (b) those that are $= p$ (called "White" by DNF)

(c) those that are $> p$ (called "Blue" by DNF)

Then determine in which partition the kth smallest element is, and do a suitable recursive call on that partition (unless it's the middle one in which case we know that p is the kth smallest element).

In our example, suppose we choose 23 as our pivot. Then the Dutch National Flag algorithm will give us 3 partitions:

1. the 10 elements that are < 23
2. the 1 element that is $= 23$
3. the 14 elements that are > 23.

To find the 17th smallest element of the given numbers, we thus need to make a recursive call to find the 6th smallest element of the third partition.

Running time

- If we are unlucky to always pick the smallest element as the pivot, we get the recurrence

$$T(n) \in T(n-1) + \Theta(n)$$

  and thus $T(n) \in \Theta(n^2)$, just as could happen for Quicksort in the worst case.

- If we pick the pivot *randomly*, one can prove by probabilistic analysis (as we may do in the graduate version of this course) that the *expected* running time is in $\Theta(n)$.

- If we pick the pivot as the *median*, we get the recurrence

$$T(n) \in T(\frac{n}{2}) + \Theta(n) + f(n)$$

  where $f(n)$ is the cost of finding the median. *If that can be done in linear time* we thus have the recurrence

$$T(n) \in T(\frac{n}{2}) + \Theta(n)$$

  and we get $T(n) \in \Theta(n)$, as desired! Before we get too impressed by ourselves, however, observe that finding the median is a special case of the selection problem: find the $\frac{n}{2}$th smallest element. So what we have really showed is that

  if we can do selection in linear time
  then we can do selection in linear time

  which is a much less impressive result...

In the following, we shall restrict ourselves to deterministic algorithms (thus not rely on any random choices), and shall indeed aim for the pivot to be the median. But rather than aiming for the pivot to be the exact median, which as we saw would be a kind of "chicken-and-egg" problem, we shall be less ambitious (since often the *perfect* is the enemy of the *good*) and go for a "pseudo-median" such that the partitions are not "too imbalanced". In the next note, we shall concretize this loose idea.