5.1 Introduction and Problem Definition

We saw last lecture a way to sort in time $O(n \log n)$: Randomized Quicksort. There are also other sorting algorithms with similar time bounds, most notably Mergesort and Heapsort (you should all know both of these already). In this lecture we will discuss a related problem with some surprisingly efficient algorithms: median-finding, or more generally, selection.

The median problem is the following: given an unsorted array, find and return the median element. In other words, given an array of length n, find and return the $(n/2)$nd smallest element. The selection problem is only slightly more general: given an array of length n and a value $k \leq n$, find and return the kth smallest element. From now on we'll mostly talk about selection.

It is obvious that selection can be done in time $O(n \log n)$: we can sort the array (using, e.g., mergesort), and then return the kth smallest element. Can we do any better?

It turns out that the answer is yes! We can do selection in $O(n)$ time, both randomized (worst-case expected time) and deterministic.

There are a few easy cases, which we can do to warm up. For example, suppose $k = 1$. Then we are trying to find the smallest element, which we can do by simply scanning the array in $O(n)$ time and keeping track of the smallest. Similarly, if $k = n$ a simple scan also suffices. In general, this strategy works whenever $k = O(1)$ or $k = n - O(1)$, since we can just keep track of the k smallest/largest elements we see while we do a scan.

This doesn't work for $k = n/2$, though. If we kept track of the k smallest elements, then when considering a new element in the scan we would have to figure out its place in the smallest k, which takes time $\Theta(\log k) = \Theta(\log n)$ (upper bound via binary search, lower bound something we'll see next week). So the total time would be $\Theta(n \log k) = \Theta(n \log n)$.

5.2 Randomized Quickselect

The idea here is to use randomized quicksort, but instead of recursing on both sides we only recurse on the side which has the desired element. Slightly more formally, suppose we are given an array A of length n and an integer $k \leq n$. Then Randomized Quickselect does the following:

1. If $n = 1$, return the element.
2. Pick a pivot element p uniformly at random from A.
3. Compare each element of A to p, creating subarrays L of elements less than p and G of elements greater than p.
4. (a) If $|L| = k - 1$ then return p.

(b) if $|L| > k-1$ then return Quickselect(L, k).

(c) If $|L| < k-1$ then return Quickselect$(G, k-|L|-1)$.

It's easy to argue correctness by arguing inductively that on every call to Quickselect(X, a), the original element we were looking for (the k'th smallest of A) must be the a'th smallest of X (do at home!). To argue running time, first note that the same intuition from quicksort continues to hold. We expect that our pivot splits the array approximately in half. This means that after $O(\log n)$ iterations we will find the element we are looking for. This might seem like it would give a bound of $n \log n$, but in each iteration the number of comparisons we make also goes down by a factor of (approximately) 2, and thus the total number of comparisons is only $O(n)$.

Let's make this a little more formal. Let $T(n)$ be the expected running time of Quickselect on an array of length n. As with quicksort, splitting the array around a pivot takes $n-1$ comparisons. Each possible split is equally likely, i.e. $|L|$ is uniformly distributed between 0 and $n-1$ (and same with $|G|$). Note that $T(n) \leq T(n+1)$ for all n. So whether we recurse in G or L depends on k and on the split, but since we are trying to provide an upper bound we can assume that we recurse on whichever has more elements (since that will make our algorithm take longer).

Thus we can write the following recurrence relation:

$$\begin{aligned} T(n) &\leq (n-1) + \sum_{i=1}^{n-1} \frac{1}{n} \max(T(i), T(n-i-1)) \\ &\leq (n-1) + \sum_{i=0}^{n/2-1} \frac{1}{n} T(n-i-1) + \sum_{i=n/2}^{n-1} \frac{1}{n} T(i) = (n-1) + \frac{2}{n} \sum_{i=n/2}^{n-1} T(i) \end{aligned}$$

Now let's use our guess-and-check method, with the guess $T(n) \leq 4n$.

$$\begin{aligned} T(n) &\leq (n-1) + \frac{2}{n} \sum_{i=n/2}^{n-1} 4i = (n-1) + 4 \cdot \frac{2}{n} \sum_{i=n/2}^{n-1} i \\ &= (n-1) + 4 \cdot \frac{2}{n} \left(\sum_{i=1}^{n-1} i - \sum_{i=1}^{n/2-1} i \right) \\ &= (n-1) + 4 \cdot \frac{2}{n} \left(\frac{n(n-1)}{2} - \frac{(n/2)(n/2-1)}{2} \right) \\ &\leq (n-1) + 4 \cdot \left((n-1) - \frac{n/2-1}{2} \right) \\ &\leq (n-1) + 4 \left(\frac{3n}{4} \right) \leq 4n. \end{aligned}$$

5.3 Deterministic Algorithm

What if we want a deterministic algorithm? Somewhat amazingly, this turns out to be possible. The basic idea is to deterministically find a pivot that will result in a more-or-less even split, and

use quickselect with this pivot. At first glance this seems pointless – in order to find a pivot which results in a good split we need to find one near the center, but finding an element at a specific place in the order is exactly what we're trying to solve! And it's true that we won't be able to quickly find the element that exactly splits the array, i.e., the median. But we will be able to find an element that is *close* to the median.

We will use a trick known as "median-of-medians". This is the key to the whole algorithm. We start by splitting the array A into $n/5$ groups of 5 (we do this arbitrarily, say by just grouping consecutive subarrays of length 5 in the unsorted input A). We then compute the median of each group, and then the median of the medians, and let this be our pivot p.

Let's start by proving that p, the "median-of-medians", is a good pivot: both L and G are reasonably small.

Lemma 5.3.1 *$|L|$ and $|G|$ are both at most $7n/10$ (when the pivot p is the median-of-medians).*

Proof: Let B be a group, and suppose that the median m of B is less than p. Then clearly m and the two elements less than it in B are all less than p, and thus in L. On the other hand, suppose that m is larger than p. Then m and the two elements larger than it in B are all in G.

Now note that exactly half of the groups (i.e. $n/10$ groups) have their median less than p, and half have their median larger than p (since p is the median-of-medians). Hence $|L| \geq \frac{n}{10} \cdot 3 = 3n/10$, and similarly $|G| \geq 3n/10$. Since every element other than p is in either L or G, this clearly implies that $|L|$ and $|G|$ are both at most $7n/10$. ■

So now we have a plan: compute the median-of-medians, and use that as the pivot. But computing the median is exactly what we're trying to solve in the first place! Fortunately the size of the input has shrunk, so we can do this recursively. Putting it all together, we get the following algorithm, which I'll call BPFRT (since it was invented by Blum-Pratt-Floyd-Rivest-Tarjan). As before, we are given an array A of size n and an integer $k \leq n$.

BPFRT(A, k):

1. Group A into $n/5$ groups of 5, and let A' be an array of size $n/5$ containing the median of each group.
2. Let $p = \text{BPFRT}(A', n/10)$, i.e., recursively find the median p of A' (the median-of-the-medians).
3. Split A using p as a pivot into L and G.
4. Recurse on the appropriate piece:
   (a) if $|L| = k - 1$ then return p.
   (b) if $|L| > k - 1$ then return $\text{BPFRT}(L, k)$.
   (c) if $|L| < k - 1$ then return $\text{BPFRT}(G, k - |L| - 1)$.

Let $T(n)$ be the running time on an array of size n (where as before we just use the worst case over possible k's). Then step 1 takes time $O(n)$, step 2 takes time $T(n/5)$, step 3 takes time $O(n)$, and

step 4 takes time at most $7n/10$ (by Lemma **??**). So the total running time is

$$T(n) \leq T(7n/10) + T(n/5) + cn.$$

It's a good exercise to draw out the recursion tree to see what's going on, but we can also solve by guess-and-check. Then we will guess that $T(n) \leq 10cn$. When we check this, we get that

$$T(n) \leq 10c(7n/10) + 10c(n/5) + cn = 9cn + cn = 10cn.$$

5.4 Deterministic Quicksort

We can now use Quickselect to get a deterministic version of Quicksort which only uses $O(n \log n)$ comparisons in the worst case (recall that traditional Quicksort uses $\Theta(n^2)$ comparisons in the worst case, while randomized Quicksort uses $O(n \log n)$ in expectation). The algorithm is simple: when deciding on a pivot, use Quickselect to find the median, and then use that as a pivot. Clearly this splits the input in half, so the total number of comparisons is

$$T(n) = 2T(n/2) + cn = O(n \log n),$$

where the cn term is the number of comparisons used for Quickselect plus the number used to split the array on the pivot.