Lecture 4

Selection (deterministic & randomized): finding the median in linear time

4.1 Overview

Given an unsorted array, how quickly can one find the median element? Can one do it more quickly than by sorting? This was solved affirmatively in 1972 by (Manuel) Blum, Floyd, Pratt, Rivest, and Tarjan. In this lecture we describe two linear-time algorithms for this problem: one randomized and one deterministic. More generally, we give linear-time algorithms for the problem of finding the kth smallest out of an unsorted array of n elements.

4.2 Lower bounds for comparison-based sorting

We saw in the last lecture that Randomized Quicksort will sort any array of size n with only $O(n \log n)$ comparisons in expectation. Mergesort and Heapsort are two other algorithms that will sort any array of size n with only $O(n \log n)$ comparisons (and these are deterministic algorithms, so there is no "in expectation"). Can one hope to sort with fewer, i.e., with $o(n \log n)$ comparisons? Let us quickly see why the answer is "no", at least for deterministic algorithms (we will analyze lower bounds for randomized algorithms later).

To be clear, we are considering *comparison-based sorting* algorithms that only operate on the input array by comparing pairs of elements and moving elements around based on the results of these comparisons. Sometimes it is helpful to view such algorithms as providing instructions to a data-holder ("move this element over here", "compare this element with that element and tell me which is larger"). The two key properties of such algorithms are:

1. The output must be a permutation of the input.
2. The permutation it outputs is solely a function of the series of answers it receives (any two inputs yielding the same series of answers will cause the same permutation to be output).

Using these key properties, we can show the following theorem.

Theorem 4.1 *For any deterministic comparison-based sorting algorithm $\mathcal{A}$, for all $n \geq 2$ there exists an input I of size n such that $\mathcal{A}$ makes at least $\log_2(n!) = \Omega(n \log n)$ comparisons to sort I.*

Proof: Suppose for contradiction that tha $\mathcal{A}$ is able to sort every array of size n using at most $k < \log_2(n!)$ comparisons. Notice there are at most 2^k different sequences of answers to these comparisons it can possibly receive. Therefore, by property (2) above, there are at most $2^k < n!$ different permutations of its input that it can possibly output. So there is at least one permutation of its input that it will never output. This means that $\mathcal{A}$ cannot be a correct sorting algorithm since for any permutation π, there is some ordering of $\{1, 2, \ldots, n\}$ for which π is the only correct answer. ■

Question: Suppose we consider the problem: "order the input array so that the smallest $n/2$ come before the largest $n/2$"? Does our lower bound still hold for that problem, or where does it break down?

Answer: No, the proof does not still hold. It breaks down because any given input will have multiple correct answers. E.g., for input `[3 4 2 1]`, we could output any of $[1, 2, 3, 4]$, $[2, 1, 3, 4]$, $[1, 2, 4, 3]$, or $[2, 1, 4, 3]$. So, even if there are some permutations the algorithm never outputs, it can still be correct. In fact, not only does the lower bound break down, but we will be able to solve this problem in linear time, by solving the selection problem we turn to now.

4.3 The selection problem and a randomized solution

A related problem to sorting is the problem of finding the kth smallest element in an unsorted array. (Let's say all elements are distinct to avoid the question of what we mean by the kth smallest when we have equalities). One way to solve this problem is to sort and then output the kth element. Is there something faster – a linear-time algorithm? The answer is yes. We will explore both a simple randomized solution and a more complicated deterministic one. Note that using $k = n/2$ (to find the median) and then re-scanning the array comparing every element to the median, will solve the "put the smallest $n/2$ before the largest $n/2$" problem we discussed above.

The idea for the randomized algorithm is to notice that in Randomized-Quicksort, after the partitioning step we can tell which subarray has the item we are looking for, just by looking at their sizes. So, we only need to recursively examine one subarray, not two. For instance, if we are looking for the 87th-smallest element in our array, and after partitioning the "LESS" subarray (of elements less than the pivot) has size 200, then we just need to find the 87th smallest element in LESS. On the other hand, if the "LESS" subarray has size 40, then we just need to find the $87-40-1 = 46$th smallest element in GREATER. (And if the "LESS" subarray has size exactly 86 then we just return the pivot). One might at first think that allowing the algorithm to only recurse on one subarray rather than two would just cut down time by a factor of 2. However, since this is occuring recursively, it compounds the savings and we end up with $\Theta(n)$ rather than $\Theta(n \log n)$ time. This algorithm is often called Randomized-Select, or QuickSelect.

QuickSelect: Given array A of size n and integer $k \leq n$,

1. Pick a pivot element p at random from A.

2. Split A into subarrays LESS and GREATER by comparing each element to p as in Quicksort. While we are at it, count the number L of elements going in to LESS.
3. (a) If $L = k-1$, then output p.
   (b) If $L > k-1$, output QuickSelect(LESS, k).
   (c) If $L < k-1$, output QuickSelect(GREATER, $k-L-1$)

Theorem 4.2 *The expected number of comparisons for QuickSelect is $O(n)$.*

Before giving a formal proof, let's first get some intuition. If we split a candy bar at random into two pieces, then the expected size of the larger piece is 3/4 of the bar. If the size of the larger subarray after our partition was always 3/4 of the array, then we would have a recurrence $T(n) \leq (n-1) + T(3n/4)$ which solves to $T(n) < 4n$. Now, this is not quite the case for our algorithm because $3n/4$ is only the *expected* size of the larger piece. That is, if i is the size of the larger piece, our expected cost to go is really $\mathbf{E}[T(i)]$ rather than $T(\mathbf{E}[i])$. However, because the answer is linear in n, the average of the $T(i)$'s turns out to be the same as T(average of the i's). Let's now see this a bit more formally.

Proof (Theorem 4.2): Let $T(n,k)$ denote the expected time to find the kth smallest in an array of size n, and let $T(n) = \max_k T(n,k)$. We will show that $T(n) < 4n$.

First of all, it takes $n-1$ comparisons to split into the array into two pieces in Step 2. These pieces are equally likely to have size 0 and $n-1$, or 1 and $n-2$, or 2 and $n-3$, and so on up to $n-1$ and 0. The piece we recurse on will depend on k, but since we are only giving an upper bound, we can imagine that we always recurse on the larger piece. Therefore we have:

$$\begin{aligned} T(n) &\leq (n-1) + \frac{2}{n}\sum_{i=n/2}^{n-1} T(i) \\ &= (n-1) + \text{avg}\left[T(n/2), \ldots, T(n-1)\right]. \end{aligned}$$

We can solve this using the "guess and check" method based on our intuition above. Assume inductively that $T(i) \leq 4i$ for $i < n$. Then,

$$\begin{aligned} T(n) &\leq (n-1) + \text{avg}\left[4(n/2), 4(n/2+1), \ldots, 4(n-1)\right] \\ &\leq (n-1) + 4(3n/4) \\ &< 4n, \end{aligned}$$

and we have verified our guess. ■

4.4 A deterministic linear-time algorithm

What about a deterministic linear-time algorithm? For a long time it was thought this was impossible – that there was no method faster than first sorting the array. In the process of trying to prove this claim it was discovered that this thinking was incorrect, and in 1972 a deterministic linear time algorithm was developed.

The idea of the algorithm is that one would like to pick a pivot deterministically in a way that produces a good split. Ideally, we would like the pivot to be the median element so that the two sides are the same size. But, this is the same problem we are trying to solve in the first place! So, instead, we will give ourselves leeway by allowing the pivot to be any element that is "roughly" in the middle: at least 3/10 of the array below the pivot and at least 3/10 of the array above. The algorithm is as follows:

DeterministicSelect: Given array A of size n and integer $k \leq n$,

1. Group the array into $n/5$ groups of size 5 and find the median of each group. (For simplicity, we will ignore integrality issues.)
2. Recursively, find the true median of the medians. Call this p.
3. Use p as a pivot to split the array into subarrays LESS and GREATER.
4. Recurse on the appropriate piece.

Theorem 4.3 *DeterministicSelect makes $O(n)$ comparisons to find the kth smallest in an array of size n.*

Proof: Let $T(n,k)$ denote the worst-case time to find the kth smallest out of n, and $T(n) = \max_k T(n,k)$ as before.

Step 1 takes time $O(n)$, since it takes just constant time to find the median of 5 elements. Step 2 takes time at most $T(n/5)$. Step 3 again takes time $O(n)$. Now, we claim that at least 3/10 of the array is $\leq p$, and at least 3/10 of the array is $\geq p$. Assuming for the moment that this claim is true, Step 4 takes time at most $T(7n/10)$, and we have the recurrence:

$$T(n) \quad \leq \quad cn + T(n/5) + T(7n/10), \tag{4.1}$$

for some constant c. Before solving this recurrence, lets prove the claim we made that the pivot will be roughly near the middle of the array. So, the question is: how bad can the median of medians be?

Let's first do an example. Suppose the array has 15 elements and breaks down into three groups of 5 like this:

$$\{1,2,3,10,11\}, \quad \{4,5,6,12,13\}, \quad \{7,8,9,14,15\}.$$

In this case, the medians are 3, 6, and 9, and the median of the medians p is 6. There are five elements less than p and nine elements greater.

In general, what is the worst case? If there are $g = n/5$ groups, then we know that in at least $\lceil g/2 \rceil$ of them (those groups whose median is $\leq p$) at least three of the five elements are $\leq p$. Therefore, the total number of elements $\leq p$ is at least $3\lceil g/2 \rceil \geq 3n/10$. Similarly, the total number of elements $\geq p$ is also at least $3\lceil g/2 \rceil \geq 3n/10$.

Now, finally, let's solve the recurrence. We have been solving a lot of recurrences by the "guess and check" method, which works here too, but how could we just stare at this and *know* that the answer is linear in n? One way to do that is to consider the "stack of bricks" view of the recursion tree discussed in Lecture 2.

In particular, let's build the recursion tree for the recurrence (4.1), making each node as wide as the quantity inside it:

Notice that even if this stack-of-bricks continues downward forever, the total sum is at most

$$cn(1 + (9/10) + (9/10)^2 + (9/10)^3 + \ldots),$$

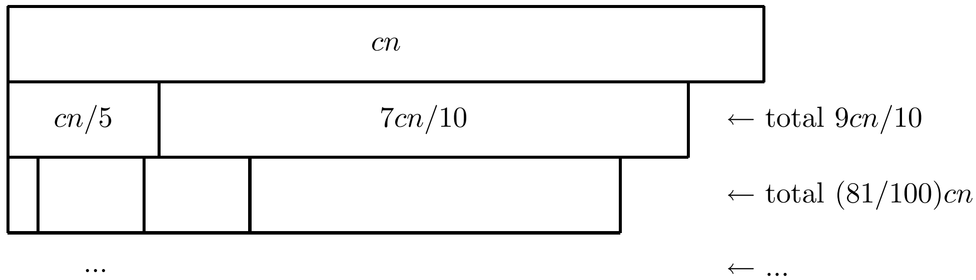

Figure 4.1: Stack of bricks view of recursions tree for recurrence 4.1.

which is at most $10cn$. This proves the theorem. ■

Notice that in our analysis of the recurrence (4.1) the key property we used was that $n/5 + 7n/10 < n$. More generally, we see here that if we have a problem of size n that we can solve by performing recursive calls on pieces whose total size is at most $(1 - \epsilon)n$ for some constant $\epsilon > 0$ (plus some additional $O(n)$ work), then the total time spent will be just linear in n. This gives us a nice extension to our "Master theorem" from Lecture 2.

Theorem 4.4 *For constants c and $a_1, \ldots, a_k$ such that $a_1 + \ldots a_k < 1$, the recurrence*

$$T(n) \quad \leq \quad T(a_1 n) + T(a_2 n) + \ldots T(a_k n) + cn$$

solves to $T(n) = \Theta(n)$.