CS161, Lecture 4 Median and Selection
Scribe: Albert Chen (2015) **Date:** April 7, 2016

1 Introduction

Today, we introduce the problem of finding the kth smallest element in an unsorted array. First, we show it can be done in $O(n \log n)$ time via sorting and that any correct algorithm must run in $\Omega(n)$ time. However, it is not obvious that a linear-time selection algorithm exists. We present a linear-time select algorithm, with an intuition for why it has the desired properties to achieve $O(n)$ running time.

2 Selection

Now for main part of this lecture. The selection problem is to find the kth smallest number in an array A.

Input: array A of n numbers, and an integer $k \in \{1, \cdots, n\}$.

Output: the k-th smallest number in A.

One approach is to sort the numbers in ascending order, and then return the kth number in the sorted list. This takes $O(n \log n)$ time, since it takes $O(n \log n)$ time for the sort (e.g. by mergesort) and $O(1)$ time to return kth number.

2.1 Minimum Element

As always, we ask if we can do better (i.e. faster in Big-Oh complexity). In the special case where $k = 1$, selection is the problem of finding the minimum element. We can do this in $O(n)$ time by scanning through the array and keeping track of the minimum element so far. If the current element is smaller than the minimum so far, we update the minimum.

Algorithm 1: SelectMin(A)

```
$m \leftarrow \infty$;
$n \leftarrow$ length($A$);
for $i = 1$ to $n$ do
    if $A(i) < m$ then
        $m \leftarrow A(i)$;
return $m$;
```

In fact, this is the best running time we could hope for.

Definition 2.1. *A deterministic algorithm is one which, given a fixed input, always performs the same operations (as opposed to an algorithm which uses randomness).*

Claim 1. *Any deterministic algorithm for finding the minimum has runtime $\Omega(n)$.*

Proof of Claim **??**. Intuitively, the claim holds because any algorithm for the minimum must look at all the elements, each of which could be the minumum. Suppose a correct deterministic algorithm does not look at $A(i)$ for some i. Then the output cannot depend on $A(i)$, so the algorithm returns the same value whether $A(i)$ is the minimum element or the maximum element. Therefore the algorithm is not always correct, a contradiction. So there is no sublinear deterministic algorithm for finding the minimum. □

So for $k = 1$, we have an algorithm which achives the best running time possible. By similar reasoning, this lower bound of $\Omega(n)$ applies to the general selection problem. So ideally we would like to have a linear-time selection algorithm in the general case.

2.2 Wishful Selection

In fact, a linear-time selection algorithm does exist. Before showing the linear time selection algorithm, it's helpful to build some intuition on how to approach the problem. Suppose we have a black-box algorithm MagicMedian that can find the median in linear time (the median is the $k = \lceil n/2 \rceil$th smallest element). Then we could use MagicMedian as a subroutine to develop a linear-time selection algorithm. For clarity we'll assume all elements are distinct from now on, but the idea generalizes easily.

Algorithm 2: WishfulSelect(A, k)

```
$m \leftarrow$ MagicMedian$(A)$;
$n \leftarrow$ length$(A)$;
if $k = \lceil n/2 \rceil$ then
    return $m$;
$A_< \leftarrow \{A(i) \mid A(i) < m\}$;
$A_> \leftarrow \{A(i) \mid A(i) > m\}$;
if $k < \lceil n/2 \rceil$ then
    return WishfulSelect$(A_<, k)$;
else
    return WishfulSelect$(A_>, k - \lceil n/2 \rceil)$;
```

Each iteration, we use the median to partition the array into two halves, into all elements smaller than the median and all elements larger. Since the median is the $\lceil n/2 \rceil$th smallest element, we know which half contains the kth smallest element. If it's the smaller half, the kth element is still the kth element; for the larger half we need to subtract $\lceil n/2 \rceil$ because each element in the larger half is greater than the $\lceil n/2 \rceil$ elements which are removed. We repeat, recursively working on a problem of smaller and smaller size until we find the kth element.

Claim 2. *If* MagicMedian *runs in $O(n)$ time, then* WishfulSelect *runs in $O(n)$ time.*

Proof. Intuitively, the running time is good because we remove half of the elements from consideration each iteration. It takes linear time for MagicMedian, linear time to create $A_<$ and $A_>$, and constant time for the rest. Recusive calls are made on inputs of half the size. As a recurrence relation, $T(n) \leq cn + T(n/2)$. Expanding this, the runtime is $T(n) \leq cn + cn/2 + cn/4 + ... + c \leq 2cn$, which is $O(n)$. □

Unfortunately, however, we do not know how to construct procedure MagicMedian without solving the original selection problem. Because of this, the WishfulSelect procedure doesn't actually make sense.

2.3 Linear-Time Selection

Given a linear-time median algorithm, we can solve the selection problem in linear time (and vice versa). Unfortunately, we don't have MagicMedian. But notice that as far as correctness goes, there was nothing special about partitioning around the median. We could use this same idea of partitioning and recursing on a smaller problem even if we partition around an arbitrary element. To get a good runtime, however, we need to guarantee that the subproblems get smaller quickly. In 1973, Blum, Floyd, Pratt, Rivest, and Tarjan came up with the Median of Medians algorithm. It is similar to the previous algorithm, but rather than partitioning around the exact median, uses a surrogate "median of medians". The recursive calls are updated accordingly.

Algorithm 3: SELECT(A, k) // Median of Medians

```
$n \leftarrow$ length$(A)$;
if $n = 1$ then
    return $A[1]$;
Split $A$ into $g = \lceil n/5 \rceil$ groups, where $g - 1$ have 5 elements each, and one has the remaining $\leq 5$ elements.;
for $i = 1$ to $g$ do
    $p_i \leftarrow$ BRUTEFORCEMEDIAN$(g_i)$ ;    compute the median of 5 numbers in constant time, say via sorting
$q \leftarrow$ SELECT$([p_1, \cdots, p_g], \lceil g/2 \rceil)$ ;    find median of medians
$A_< \leftarrow \{A(i) \mid A(i) < q\}$;
$A_> \leftarrow \{A(i) \mid A(i) > q\}$;
$a \leftarrow$ length$(A_<)$;
if $k \leq a$ then
    return SELECT$(A_<, k)$;
else if $k = a + 1$ then
    return $q$;
else
    return SELECT$(A_>, k - (a + 1))$;
```

2.4 Analysis of Select

What is this algorithm doing? First it divides A into segments of size 5. Within each group, it finds the median by brute force, for example by first sorting the elements. This takes constant time for each group, because each group has a constant number of elements. Then it makes a recursive call to SELECT to find the median q of all these group medians. Intuitively, by partitioning around q, we are able to find something that is close to the true median for partitioning, yet is 'easier' to compute, because it is the median of $g = \lceil n/5 \rceil$ elements rather than n. The last part is as before: once we have our pivot element q, we split the array and recurse on the proper subproblem, or halt if we found our answer.

We have devised a slightly complicated method to determine which element to partition around, but the algorithm remains correct for the same reasons as before. So what is its running time? As before, we're going to show this by examining the size of the recursive problems. As it turns out, by taking the median of medians approach, we have a guarantee on how much smaller the problem gets each iteration. The guarantee is good enough to achieve $O(n)$ runtime.

2.4.1 Running Time

Lemma 2.1. $|A_<| \leq 7n/10 + 5$ *and* $|A_>| \leq 7n/10 + 5$.

Proof of Lemma **??**.

q is the median of $p_1, \cdots, p_g$. Because q is the median of $g = \lceil n/5 \rceil$ elements, $\lceil g/2 \rceil - 1$ of the $p_i's$ are smaller than q. By transitivity, if q is larger than a group median, it is larger than at least three elements in that group (the median and the two numbers less than it). This applies to all groups except the remainder group, which might have fewer than 5 elements. Accounting for the remainder group, q is larger than at least $3 \cdot (\lceil g/2 \rceil - 2)$ elements of A. By symmetry, q is smaller than at least the same number.

Now,

$$\begin{aligned} |A_>| &= \text{\# of elements greater than } q \\ &= (n-1) - \text{\# elements smaller than } q \\ &\le (n-1) - 3 \cdot (\lceil g/2 \rceil - 2) \\ &= n + 5 - 3 \cdot \lceil g/2 \rceil \\ &\le n - 3n/10 + 5 \\ &= 7n/10 + 5. \end{aligned} \tag{1}$$

By symmetry, $|A_<| \le 7n/10 + 5$ as well. □

The recursive call used to find the median of medians has input of size $\lceil n/5 \rceil \le n/5 + 1$. The other work in the algorithm takes linear time: constant time on each of $\lceil n/5 \rceil$ groups for BruteForceMedian $\Longrightarrow$ linear time in total, $O(n)$ time scanning A to make $A_<$ and $A_>$, and $O(n)$ time for finding the length of A and $A_<$. The sum of this constant amount of linear steps is still linear.

Thus, we can write the full recurrence for the runtime,

$$T(n) \le \begin{cases} c_1 n + T(n/5 + 1) + T(7n/10 + 5) & \text{if } n > 5 \\ c_2 & \text{if } n \le 5. \end{cases}$$

This seems good at first glance. Each step does linear work in terms of its input size, and the total amount of linear work propagated to the next level is about $n/5 + 7n/10 = 9n/10$, so we expect to see an exponential decay in the runtime of each layer of the recursion tree, just as in WishfulSelect. In fact, in the last lecture we solved exactly the above recurrence and obtained $T(n) = O(n)$ as claimed.