Divide-and-Conquer: Finding The Median

Selection Problems

Selection problem. A selection problem is the problem of given an array of n numbers finding the ith largest (or smallest) number in the array.

Finding the largest, the smallest, the second largest number in an array are all instances of a selection problem.

If i is constant, then $T_{Select(i)}(n) = O(n)$, in fact, we can find the ith element in less than $i \cdot n$ comparisons.[1]

Finding Median

Problem. Finding a median. Given an array of n elements, find its median.

This problem can be reduced to solving one or two selection problems. Indeed, if n is odd, then finding a median is a selection problem with $i = \lfloor n/2 \rfloor + 1$. Ifr n is even, then finding a median can be reduced to two selection problems for values $i = n/2$ and $i = n/2 + 1$.

Naïve Algorithm. Using our traditional approach to selection, finding a median median will yield an algorithm with $T(n) = O(n^2)$.

Sort-based Algorithm. A simple improvement over the naïve algorithm is a sort-based algorithm:

- Sort input array A using any $O(n \log(n))$ algorithm.
- Return $A[\frac{n}{2}]$ if n is odd, or $\frac{A[\frac{n}{2}]+A[\frac{n}{2}+1]}{2}$ if n is even.

[1] We actually know that tighter bounds exist, since the second largest element can be found using $n - 1 + log_2(n) - 1$ comparisons.

This algorithm has the complexity $O(n\log(n))$.

Linear Algorithm. Can we do better?

We discuss the general SELECT(A[1..n], n, i algorithm, which uses divide-and-conquer strategy to find ith smallest element in the array. If we can build a linear selection algorithm, the linear algorithm for median will follow.

Idea #1. Pick an element x from the array. Compare all other elements to it, and split the array into two parts: one that contains all numbers smaller than x, and the other, containing all elements greater than or equal to x. Determine, in which of the two subarrays, the ith smallest element will lie. Recursively find this element in the subarray.

Problem with Idea # 1. We can pick x which is **really bad** for us. (e.g., looking for a median, we pick x with is the largest element in the array).

Idea #2. We would like to run **Idea #1**, but with a guarantee, that the pivot number x we pick is *not too bad*. I.e., we want a guarantee, that at least a certain number of array elements will be on either side of x. We also would like to establish this we *reasonably few comparison operations*.

We can do this using the following algorithm:

1. Divide input array A into $n/5$ groups of 5 elements in each (the last group can have fewer elements).
2. Find the median of each group of 5 elements using insertion-sort and then taking the third element. Let $b_1, \ldots b_k$, where $k = n/5$ be the list of medians.
3. **Recursively** find the median of $b_1, \ldots, b_k$. Let c be the median. of $b_1, \ldots, b_k$.
4. Partition input array A around c. Let $d_1, \ldots d_m$ be all elements of A that are less than c, and $e_1, \ldots, e_t$ are all elements of A that are greater than or equal to c. $m + t = n$.
5. If $m \geq i$, then the ith smallest element is the low partition. Call SELECT$((d_1, \ldots, d_m), m, i)$.
6. If $m < i$, then, the ith element of A is the $i - m$th element of the upper partition. Call SELECT$(e_1, \ldots, e_t), t, i - m)$.

Algorithm Analysis. We need to show that SELECT(A,n,i) has linear running time. We will look at the number of comparisons that SELECT makes.

Step 1. How many elements are **guaranteed** to be in each partition. (a.k.a., there was a reason we chose the median of medians).

How many elements are guaranteed to be greater than c? c is greater than $\frac{1}{2} \cdot \frac{n}{2} - 1$ other group medians. This means that in those groups, at least 3 elements are greater than c (except for the last group, which may contain fewer than 5 elements). This means that we have at least

$$3\left(\lceil\frac{1}{2}\lceil\frac{n}{5}\rceil\rceil - 2\right) = \frac{3n}{10} - 6$$

array elements that are greater than c. Similarly, $\frac{3n}{10} - 6$ elements are less than c.

Step 2. The largest possible size of a partition (either lower or upper) is

$$n - \left(\frac{3n}{10} - 6\right) = \frac{7n}{10} + 6$$

elements.

Step 3. On Step 3 of the algorithm we make a recursive call to SELECT with the input array size of $n/5$.

On Steps 5/6 of the algorithm we will make one recursive call to SELECT with the input array of size at most $\frac{7n}{10} + 6$.

Steps 1,2 and 4 take $O(n)$ time.

Our recurrence is thus:

$$T(n) = T\left(\lceil\frac{n}{5}\rceil\right) + T\left(\frac{7n}{10} + 6\right) + O(n)$$

We also assume that $T(n) = O(1)$ for $n \leq 140$.

To solve this recurrence, assume $T(n) \leq cn$ for some $c > 0$ and $n \leq 140$. (given that $T(n) = O(1)$ for $n \leq 140$, this will be true for large enough c).

Also, let $a > 0$ be such that the $O(n)$ term is the recurrence is bound by an, i.e., let

$T(n) \leq T\left(\lceil\frac{n}{5}\rceil\right) + T\left(\frac{7n}{10} + 6\right) + an$

Then

$$\begin{aligned}
T(n) &\leq c\lceil\frac{n}{5}\rceil + c\left(\frac{7n}{10} + 6\right) + an \\
&\leq \frac{cn}{5} + c + \frac{7cn}{10} + 6c + an \\
&= \frac{9cn}{10} + 7c + an \\
&= cn + \left(-\frac{cn}{10} + 7c + an\right)
\end{aligned}$$

If $-\frac{cn}{10} + 7c + an \leq 0$, then $T(n) \leq cn$.

Because $n > 140$, $\frac{n}{n-70} \leq 2$. In this case, for $c \leq 20a$,

$-\frac{cn}{10} + 7c + an \leq 0$