Deterministic Linear Time Selection

Revised October 7, 2014

Problem. Given a sequence of numbers $a_1, \ldots, a_n$ and an integer $i \in [1, n]$, find the ith smallest element. When $i = \lceil n/2 \rceil$, this is called the *median problem*.

A trivial $O(n \log n)$ algorithm is to first sort the numbers in $O(n \log n)$ time and then find the ith number in either $O(1)$ time or $O(n)$ time depending on whether the sorted sequence is stored in an array or linked list. We already saw an $O(n)$ *expected time* randomized algorithm for this problem. In this lecture we develop an $O(n)$ worst-case time algorithm.

Partition. Recall the partition subroutine that was already used in Quicksort and randomized selection. Given an array $A[1..n]$ of distinct numbers, two indices $p \leq r$ and $x = A[i]$ for some $i \in [p, r]$, rearrange the subarray $A[p..r]$ into two (possibly empty) subarrays $A[p..q-1]$ and $A[q+1..r]$ where now $x = A[q]$ and

$$A[k] \leq A[q] < A[l]$$

for any $p \leq k \leq q-1$ and $q+1 \leq l \leq r$. The value in $A[q]$ is called the *pivot*. Recall that Quicksort worked by first calling partition and then recursively sorting the two subarrays $A[p..q-1]$ and $A[q+1, r]$, while randomized selction worked by calling partition and then recursing on either the left or right subarray as appropriate.

In deterministic Quicksort we always chose $A[r]$ as the pivot. In randomized Quicksort and selection we chose a random item as the pivot.

A first attempt. A straightforward application of partition in a divide-and-conquer algorithm for the selection problem would work by always choosing $A[r]$ as the pivot. That is, suppose that we want to select the ith smallest number in the subarray $A[p, r]$.

- First run partition on $A[p..r]$ and let $A[p..q-1]$ and $A[q+1..r]$ be the rearranged subarrays returned. The rank of the entry $A[q]$ within $A[p..r]$ is $q - p + 1$.
- If $i = q - p + 1$, the pivot $A[q]$ is the answer; return it and stop.
- If $i < q - p + 1$, then $A[p..q-1]$ contains the requested entry, which is also the ith smallest number in $A[p..q-1]$. So we can recurse on $A[p..q-1]$.
- If $i > q - p + 1$, we recurse on $A[q+1, r]$ to look for the $(i - q + p - 1)$th smallest number in $A[q+1, r]$.

The disadvantage of this method is that it runs in $O(n^2)$ time in the worst case on $A[1..n]$. Consider the following example. The array $A[1..n]$ contains the numbers in increasing order and we are looking for the smallest number in $A[1..n]$. The first call of partition does nothing and then we recurse on the subarray $A[1..n-1]$. The second call of partition also does nothing and we recurse on $A[1..n-2]$ afterward. This gives a recurrence $T(n) = T(n-1) + O(n)$, which solves to $T(n) = O(n^2)$. The quadratic running time stems from the imbalanced division in the worst case.

An improvement. The intuition is to find a pivot so that the resulting division is roughly balanced, i.e., there exists a constant $\alpha \in (0, 1)$ such that at least αn numbers are less than the pivot and at least αn numbers are greater than the pivot. The algorithm finds the ith smallest number within $A[p..q]$ as follows.

1. Divide the $n = r - p + 1$ items into $\lceil n/5 \rceil$ sets in which each, except possibly the last, contains 5 items.
2. Find the *median* of each of the $\lceil n/5 \rceil$ sets. This can be done by, e.g., insertion sorting each set.
3. Take these $\lceil n/5 \rceil$ medians and put them in another array. Recursively calculate the *median* of these medians. Call this x.
4. Partition the original array using x as the pivot. Let q be index of x.

5. If $i = q - p + 1$, return x. If $i < q - p + 1$, recursively find the ith smallest number in $A[p..q-1]$. If $i > k$, recursively find the $(i - q + p - 1)$th smallest number in $A[q+1..r]$.

To visualize the algorithm's steps, it helps to form a matrix with each of the $n/5$ sets as columns, each column sorted in increasing order from top to bottom. The median of each column is the middle element in the column. The median x of the medians of the $n/5$ sets is the "middle" element of the matrix. If we remove the row and the column containing this "middle" element x , the remaining parts form four quadrants. The numbers in the upper left quadrant are less than x and the numbers in the lower right qudrant are greater than x. Hence, we roughly ensures that at least $n/4$ numbers are less than x and at least $n/4$ are greater than x. We give a more precise analysis below.

Lemma 1 *At least* $3n/10 - 6$ *numbers are greater (less) than the pivot.*

Proof. At least half of the $\lceil n/5 \rceil$ medians in step 2 are greater than or equal to to x. Ignoring the group to which x belongs and the final group (containing possibly fewer than 5 numbers), we have $\frac{1}{2}\lceil n/5 \rceil - 2$ groups whose medians are greater than x. Each such group has at least three numbers greater than x. Hence, at least $\frac{3}{2}\lceil n/5 \rceil - 6 \geq 3n/10 - 6$ are greater than x. A symmetric argument shows that at least $3n/10 - 6$ numbers are less than x. □

The above results implies that the algorithm recurses on at most $7n/10 + 6$ numbers.

To construct the recurrence first note that step 2 can be implemented using $\leq a'n$ comparisons for some constant a'. If, for example, we use Insertion sort, $a' = 2$ (why?).

Step 4 requires $\leq n$ comparisons so, setting $a = a' + 1$, gives the following recurrence:

$$\begin{aligned} T(1) &= 0. \\ T(n) &\leq T(\lceil n/5 \rceil) + T(7n/10 + 6) + an. \end{aligned}$$

We verify by induction that $T(n) \leq cn$ for some constant c as follows.

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

To prove $T(n) \leq cn$ by induction we need to find $N \geq 0$ and a condition on c such that,
for all $n > 0$, $-cn/10 + 7c + an < 0$ or, equivalently,

$$\forall n > N, \quad c \geq 10a\frac{n}{n - 70}.$$

If we set $N = 70$ this will be true if we set $c \geq 710a$.
If we set $N = 140$ this will be true if we set $c \geq 20a$.

Note that this only guarantees that the induction will work. We still need to set the initial conditions by having $T(n) \leq c(n)$ for *all* $n > 0$. This can be satisfied by also requiring that $c \geq \max_{i \leq N} \frac{T(n)}{n}$. That is, we can set

$$c = \max\left(\max_{n \leq 70} \frac{T(n)}{n},\ 710a\right)$$

or

$$c = \max\left(\max_{n \leq 140} \frac{T(n)}{n},\ 20a\right).$$

More generally, any c of the form

$$c \geq \max\left(\max_{n \leq N} \frac{T(n)}{n},\ 10\alpha\frac{N}{N - 69}\right).$$

with $N \geq 70$ would work.

Final Notes. Note that this algorithm is a very different type of divide-and-conquer recursion. In all of the previous divide and conquer algorithms we solved subproblems and built the larger solution directly out of the subproblem solutions. The solution to this problem added a new twist. It first solved a small subproblem to find the pivot x. Only after that (using the x), did it recurse to solve a subproblem whose solution would give us the real solution.

Historically, whether selection could be solved (deterministically) in less than *sorting* time was a big open question. The publication of this algorithm in 1971 [1] was an eye-opener for algorithm design.

References

[1] Manuel Blum, Robert W. Floyd, Vaughan Pratt, Ronald L. Rivest, and Robert E. Tarjan, *Time bounds for selection,* Journal of computer and system sciences, **7**(4), (1973), pp. 448–461

Note: This document was written by M J. Golin, revised from an original by S.W. Cheng, for COMP3711H, HKUST.