Notes on Linear-Time Selection, and a Sorting Lower Bound

1 The Problem

In the Selection problem, we are given an array A containing n distinct numbers and an integer $i \in \{1, 2, \ldots, n\}$. We need to output the *ith order statistic*, defined as the ith smallest number in A. So for example, if n is odd, the $[(n+1)/2]$th order statistic is the median of A; if n is even, we define the $(n/2)$th order statistic to be the median.

We can solve the Selection problem in $O(n \log n)$ time by sorting the array (e.g. using MergeSort) in $O(n \log n)$ time, and then returning the ith element of the sorted array. In these notes we give a sophisticated divide and conquer algorithm, due to Blum, Floyd, Pratt, Rivest, and Tarjan, that solves the Selection problem in linear ($O(n)$) time.

2 The Algorithm

A key subroutine in the algorithm is to partition an array around a *pivot element* (pivots are called *splitters* in Kleinberg & Tardos). This subroutine picks an element p of the array A (more on how to do this later), and rearranges the elements of A so that all array elements earlier than p in A are less than p and all array elements subsequent to p in A are greater than p. So if the initial array is $[3, 2, 5, 7, 6, 1, 8]$ and 3 is chosen as the pivot element, then the subroutine could output the rearranged array $[2, 1, 3, 5, 7, 8, 6]$. Note that we don't care about the relative order of elements that are on the same side of the pivot element. This subroutine should be familiar to all of you who have previously studied the QuickSort algorithm.

Partitioning around a pivot is useful for two reasons: it reduces the problem size, and it can be implemented in linear time. First we show that, given the pivot element, this partitioning can be done in linear time. One easy way to accomplish this is to do a linear scan through A and copy its elements into a second array B from both ends. In more detail, initialize pointers j and k to 1 and n, respectively. For $i = 1$ up to n, copy the element $A[i]$ over to $B[j]$ (and advance j) if $A[i]$ is less than the pivot, and copy it to $B[k]$ (and decrement k) if it is greater than the pivot. (The pivot element is copied over to B last, in the final remaining slot.)

There is also a slick way to partition around a pivot in place (i.e., without introducing the second array B), via repeated swaps. See any programming textbook's description of QuickSort for pseudocode. It is this in-place version that makes partitioning such an attractive subroutine in practice.

Our (underspecified) selection algorithm is as follows.

```
Select($A$,$n$,$i$)
  (0) If $n = 1$ return $A[1]$.
  (1) $p$ = ChoosePivot($A$, $n$)
  (2) $B$ = Partition($A$, $n$, $p$)
  (3) Suppose $p$ is the $j$th element of $B$ (i.e., the $j$th order statistic of $A$). Let the "first part of $B$" denote
      its first $j - 1$ elements and the "second part" its last $n - j$ elements.
      (3a) If $i = j$, return $p$.
      (3b) If $i < j$, return Select(1st part of $B$, $j$, $i$).
      (3c) Else return Select(2nd part of $B$, $n - j$, $i - j$).
```

Make sure you understand the rationale behind the recursive calls (e.g., think about the case where $i = j+1$). Now matter how the ChoosePivot subroutine is implemented, this algorithm correctly solves the selection problem; we can prove this formally using the following straightforward induction on the array length n. The base case $n = 1$ is clear (presumably $i = 1$, and the array's only occupant is the correct solution). Consider an input of length $n > 1$, and suppose the chosen pivot p is the jth order statistic of A, while we were looking for the ith order statistic. If $i = j$ then by definition we are correct in returning the pivot. If $i < j$ then the pivot is bigger than the element we seek. The ith smallest element of A is thus the ith smallest element of B (which has length $j - 1$). By the inductive hypothesis, the recursive call correctly computes this. Similarly, if $i > j$ then the sought element is larger than the pivot (and all of the first part of B); thus it is the $(i-j)$th order statistic of the 2nd part of B (which has length $n - j$). Again, the inductive hypothesis implies that the recursive call correctly solves this smaller subproblem.

The running time of the algorithm depends on the quality of the pivot chosen by ChoosePivot. For example, if ChoosePivot always returns the first element of the array and the input array A is already sorted, then the subproblem size decreases by only one array element with each recursive call. It follows that for this implementation of ChoosePivot, the worst-case running time of Select is $\Omega(n^2)$. (Do you see why?)

The key is thus to implement ChoosePivot so that it always returns a good pivot element—a pivot that gives a "balanced" partition. Of course, the best-possible pivot is the median, which might well be what we're trying to find in the first place, and we seem to have returned to square one.

The main idea of the algorithm is to implement ChoosePivot so that it finds a pretty good pivot very quickly. Our pretty good pivot will be a "median of medians", in the following sense. ChoosePivot takes the input array A and logically breaks it into $\lceil n/5 \rceil$ groups of 5 elements (plus one group with at most 5 elements). (We use the notation $\lceil x \rceil$ to mean x rounded up to the nearest integer, and similarly $\lfloor x \rfloor$ for x rounded down.) It then sorts each group independently. Using e.g. MergeSort, this requires at most 120 operations for each group, and hence at most $120 \times \lceil n/5 \rceil = O(n)$ operations in all. Copy the $\lceil n/5 \rceil$ "middle elements" (i.e., the medians of these groups of 5) into an auxiliary array M, and recursively call Select to find the median element of M. ChoosePivot returns the median of M as the pivot.

3 The Recurrence

Select is a more sophisticated divide-and-conquer algorithm than any of the ones we've seen so far. Note that it makes two recursive calls (the obvious one in Step (3), but also a second one buried in the ChoosePivot subroutine in Step (1)). These recursive calls are for different purposes and on problems of different size. For this reason, we cannot use the Master Method to analyze the running time of Select.

Let $T(n)$ denote the worst-case running time of Select on an input array of n elements. We have $T(1) = 1$. For $n > 1$, we have a recurrence

$$T(n) \leq O(n) + T(\lceil n/5 \rceil) + T(?),$$

where the $O(n)$ term is for the work done by Select not including the recursive calls (mostly the Partition subroutine of Step (2) and also the sorting in the ChoosePivot subroutine), the $T(\lceil n/5 \rceil)$ is for the recursive call to Select within the ChoosePivot subroutine, and the $T(?)$ term is for the recursive call to Select in Step (3), which we don't fully understand yet.

The key claim of the analysis is this: given that ChoosePivot returns the median of medians, the second recursive call to Select (in Step (3)) is passed an array of size at most $\approx 7n/10$. Thus our clever choice of a pivot element ensures that $\approx 30\%$ of the array gets thrown out before we recurse. (If we chose the median for the pivot, we could throw out 50% of the array before recursing.)

We now prove the key claim. Let $k = \lceil n/5 \rceil$. Suppose for simplicity that k is even (the argument when k is odd is the same). Recall that M denotes the array of k medians from the groups of (at most) 5. Let m_i denote the ith smallest element of M. Thus $m_{k/2}$ is the median of M and the pivot returned by ChoosePivot. The "almost correct" argument is as follows. First, $m_{k/2}$ is bigger than $m_1, \ldots, m_{(k/2)-1}$ and is smaller than $m_{(k/2)+1}, \ldots, m_k$. Second, for $i \leq k/2$, m_i is bigger than two of the other four elements in its group of 5 (recall m_i is the median of its group of 5). For $i \geq k/2$, m_i is smaller than two of the other four elements in its group of 5. Thus $m_{k/2}$ is bigger than at least 60% of the elements in at least (roughly) 50% of the groups—the largest three elements in each of the groups containing $m_1, \ldots, m_{k/2-1}$—and is thus bigger than at least 30% of the elements of A. Similarly, $m_{k/2}$ is smaller than at least 30% of the elements of A. Thus if

$m_{k/2}$ is used as a pivot element, both sides of the resulting partition contain at least 30% of the elements, and hence each side contains at most 70% of the elements.

This almost correct argument gives the following "almost correct" recurrence for the worst-case running time $T(n)$ of Select:

Almost Correct Recurrence

- $T(1) = 1$;
- there is a constant $c > 0$ such that $T(n) \leq T(\lfloor n/5 \rfloor) + T(\lfloor 7n/10 \rfloor) + cn$ for all $n > 1$.

The above argument and recurrence are only "almost correct" because they neglect a couple of niggling additive errors. (We have also cheated by rounding the $n/5$ down instead of up. We will fix this shortly.) The main one is that one of the "groups of 5" might have less than 5 elements in it. This potentially causes an additive loss of 2 in our argument (do you see why?). A second detail is that, when k is odd, we can't really say "50% of the groups", and have to round $k/2$ down to the nearest integer. The above argument thus rigorously shows the following: if the pivot is chosen as the median of medians, then both sides of the resulting partition contain at least $3 \times \lfloor k/2 \rfloor - 2 \geq 3k/2 - 5$ elements. Since $k \geq n/5$, this is at least $3n/10 - 5$. Substituting this into the almost correct recurrence, and also remembering that the $T(\lfloor n/5 \rfloor)$ should actually be $T(\lceil n/5 \rceil)$, we get:

A Revised Recurrence

- $T(1) = 1$;
- there is a constant $c > 0$ such that $T(n) \leq T(\lceil n/5 \rceil) + T(\lfloor 7n/10 \rfloor + 5) + cn$ for all $n > 1$.

Now the annoying problem is that the second part of the recurrence doesn't even make sense when n is small (note that $\lfloor 7n/10 \rfloor + 5$ is no smaller than n unless n is at least 20 or so). To fix this, we will invoke the following rule of thumb about the substitution ("guess and check") method: *if your argument only has problems for small n, enlarge the base case.* Using this rule, we state our final, correct recurrence that upper bounds the running time of Select:

Final Recurrence

- there is a constant $d > 0$ such that $T(n) \leq d$ for all $n \leq 100$ [new base case];
- there is a constant $c > 0$ such that $T(n) \leq T(\lceil n/5 \rceil) + T(\lfloor 7n/10 \rfloor + 5) + cn$ for all $n > 100$.

The choice of 100 here is arbitrary (any sufficiently large constant will do).

4 Solving the Recurrence

We now solve the final recurrence by the substitution method. Since we are aiming for a linear-time algorithm, our guess should be clear.

Guess: for a sufficiently large constant $a > 0$, $T(n) \leq an$ for every $n \geq 1$.

If the guess is correct, then clearly $T(n) = O(n)$. We warm up with the easier task of verifying the guess for the almost correct recurrence.

Verification for the Almost Correct Recurrence: We assume that the almost correct recurrence holds, and prove that there is a constant $a > 0$ such that $T(n) \leq an$ for all $n \geq 1$. We proceed by induction.

For the base case, suppose $n = 1$. We are given that $T(1) = 1$. Thus $T(1) \leq a \cdot 1$ as long as a is at least 1.

For the inductive step, suppose $n > 1$. We are given that $T(n) \leq T(\lfloor n/5 \rfloor) + T(\lfloor 7n/10 \rfloor) + cn$ for some constant $c > 0$. By the inductive hypothesis, $T(\lfloor n/5 \rfloor) \leq a\lfloor n/5 \rfloor$ and $T(\lfloor 7n/10 \rfloor) \leq a\lfloor 7n/10 \rfloor$. Thus

$$T(n) \leq a\left(\left\lfloor \frac{n}{5} \right\rfloor + \left\lfloor \frac{7n}{10} \right\rfloor\right) + cn \leq n\left(\frac{9}{10}a + c\right).$$

Thus the inductive step holds as long as we choose $a \geq 10c$. This completes the verification that there is a constant $a > 0$ such that $T(n) \leq an$ for all $n \geq 1$ [specifically, the choice $a = \max\{1, 10c\}$ works].

Verification for the Final Recurrence: We now verify the guess for the final (correct) recurrence. We proceed by induction. For all $n \leq 100$, we are given that $T(n) \leq d$ for a constant d. Thus $T(n) \leq an$ for all $n \leq 100$ as long as $a \geq d$. For the inductive step, suppose that $n > 100$. We are given that $T(n) \leq T(\lceil n/5 \rceil) + T(\lfloor 7n/10 \rfloor + 5) + cn$. By the inductive hypothesis, $T(\lceil n/5 \rceil) \leq a\lceil n/5 \rceil$ and $T(\lfloor 7n/10 \rfloor + 5) \leq a(\lfloor 7n/10 \rfloor + 5)$. Thus

$$T(n) \leq a\left(\frac{n}{5} + 1 + \frac{7n}{10} + 5\right) + cn.$$

Since $n > 100$,

$$T(n) \leq a\frac{96}{100}n + cn = n\left(\frac{24}{25}a + c\right).$$

Thus the inductive step holds as long as we choose $a \geq 25c$.

Choosing $a = \max\{d, 25c\}$, we have $T(n) \leq an$ for all $n \geq 1$, and the proof is complete.

5 A Lower Bound for Comparison-Based Sorting Algorithms

Can the above linear-time bound be extended to the more general problem of sorting an array? We next show that the answer is "no" for *comparison-based* sorting algorithms.

By a comparison-based sorting algorithm, we mean an algorithm that learns about the elements of the input array A only by comparing them. Non-examples include radix sort, which needs to access the bits of A's elements, as well as bucket and counting sorts, which need to know the precise values of A's elements. (These algorithms run in linear time in certain cases, such as when all array elements are small integers.) Examples of comparison-based sorting algorithms include MergeSort, HeapSort, and QuickSort. Such algorithms are useful in practice because they are general-purpose and make no assumptions about what kind of elements you want to sort (e.g., the UNIX `qsort` function requires only a pointer to the data to be sorted and a function pointer to a user-specified comparison subroutine).

Consider a comparison-based sorting algorithm that always makes at most k comparisons to correctly sort an input array of length n. By the definition of being comparison-based, the only knowledge the algorithm has about the relative order of the input elements is what can be inferred from the results of its comparisons (and the transitive closure of these results). Using at most k comparisons yields at most 2^k distinct comparison results, so the algorithm can infer at most 2^k distinct relative orderings. Since the algorithm needs to distinguish the $n!$ different possible relative orderings of A's elements, we have $2^k \geq n!$. Since $n! \geq (n/2)^{n/2}$ — or much sharper estimates follow from Stirling's approximation, if you like — we can take logarithms base 2 to obtain $k \geq \frac{n}{2}\log_2 \frac{n}{2} = \Omega(n \log n)$.