EECS 3101 York University Instructor: Andy Mirzaian

A LINEAR TIME SELECTION ALGORITHM

A problem closely related to, but simpler than sorting is that of the selection (also referred to as the *order statistics*) problem:

> **The Selection problem:** Given a sequence $S = (a_1 , a_2 , \cdots , a_n)$ of n elements on which a linear ordering is defined, and an integer k, $1 \le k \le n$, find the k-th smallest element in the sequence.

Strictly speaking, the k-th smallest of the sequence is an element b in the sequence such that there are at most $k-1$ values for i for which $a_i < b$, and at least k values of i for which $a_i \le b$. For example, 4 is the second and third smallest element of the sequence 7, 4, 2, 4 .

One obvious solution is to sort the sequence into nondecreasing order and then locate the element at the k-th position. We already know that this will take $\Omega(n \log n)$ time both in the average case and the worst case. For some values of k it is easy to solve the problem more efficiently. For example, $k = 1$ and $k = n$ correspond, respectively, to finding the minimum and the maximum element, and these can be done in $O(n)$ time. Also, when k is close to 1, we can construct a min-heap (i.e., a heap with the minimum at the root) of the given elements in $O(n)$ time, then do k deletemin operations in $O(k \log n)$ time for a total time of $O(n + k \log n)$. Note that this is $O(n)$ when k is $O(n/ \log n)$. Symmetrically, if k is close to n we can use a max-heap and deletemax operations. An important special case occurs when $k = \lceil n/2 \rceil$, in which case we are interested in finding the median. Can we find the median in linear time? Below, we will describe two algorithms to solve the general selection problem, one has $O(n)$ average time complexity and the other has $O(n)$ worst-case time complexity.

Selection in Average linear time

Algorithm *QuickSelect* shown below is a divide-and-conquer algorithm based on a variation of QuickSort. We select a pivot element at random and partition the elements of the sequence around the pivot. By comparing k with the sizes of the two blocks of the partition, we can determine which side contains the k-th smallest, and we recur on that side. Note that the major difference of *QuickSelect* and *QuickSort* is that the former recurs on one subproblem, while the latter on two.

```
function QuickSelect ( $S$ , $k$ );    (*finds the $k$-th smallest element of $S$*)
begin
    $m \leftarrow$ a random element of $S$;         (*the pivot element*)
    $S_1 \leftarrow \{ a \in S \mid a < m \}$;
    $S_2 \leftarrow \{ a \in S \mid a > m \}$;
    if $|S_1| \ge k$ then return QuickSelect ( $S_1$ , $k$ )
      else if $|S| - |S_2| \ge k$ then return $m$
         else return QuickSelect ( $S_2$ , $k - |S| + |S_2|$ )
end
```

An inplace implementation of this algorithm (i.e., without the use of any extra lists) using the *partition* subroutine of *QuickSort* shown in ,e.g., [CLR]. In the worst-case this algorithm takes $O(n^2)$ time, and hence is as bad as *QuickSort*. The reason is that by a poor choice of the pivot element, the subproblem that we have to recur on may contain almost all the elements of S. However, things are better for *QuickSelect* on average. Let $\bar{T}(n)$ denote the average time complexity of *QuickSelect* on a sequence of n elements. Now we can set up a recurrence relation for $\bar{T}(n)$ as follows. Excluding the recursive calls, the rest (including the partitioning) can be done in linear time, say in cn steps, for some constant $c > 0$. Let i be the size of the subproblem that we have to recur on. This size can range anywhere between 0 and $n-1$, all with equal probability $1/n$. Thus, the recurrence is

$$\bar{T}(n) = cn + \frac{1}{n} \sum_{i=0}^{n-1} \bar{T}(i) \,.$$

We have seen a similar recurrence in Lecture Note 2. We can use the methods developed there to find an exact solution to the above recurrence. However, let us take a different approach. We claim that $\bar{T}(n) \le 2cn$. We can prove this by a simple mathematical induction on n. Since $\bar{T}(0) = 0$ and $\bar{T}(1) = c$, the claim holds for $n = 0,1$. Now assume $n \ge 2$, and by induction hypothesis that $\bar{T}(i) \le 2ci$ for all $i \le n-1$. Then we have $\bar{T}(n) = cn + 1/n \sum_{i=0}^{n} \bar{T}(i) \le cn + 1/n \sum_{i=0}^{n-1} 2ci = 2cn - c \le 2cn$. □

Remark: After the randomized partitioning, depending on the rank of the element we are seeking, we might end up solving either of the two subproblems. If we let the adversary pick the larger of the two subproblems to solve, the recurrence above is modified as

$$\bar{T}(n) = cn + \frac{1}{n} \sum_{i=0}^{n-1} \max\{\, \bar{T}(i), \bar{T}(n-i-1) \,\} \,.$$

Show that the solution to this recurrence is still $O(n)$.

Selection in worst-case linear time

Can we suitably modify the *QuickSelect* algorithm described above to prevent bad worst-cases? One way to ensure this is to select the pivot element more carefully so that none of the two possible subproblems are too big. For instance, if the pivot element happens to be the median (and if this holds for all recursive subproblems), then the subproblem we have to solve will be at most half the size of the main problem. This will result in a recurrence of the form $T(n) = T(n/2) + O(n)$ which has the solution $O(n)$. The hole in the above argument is how do we guarantee that the pivot is the median; finding the median is a selection problem itself! The trick is we do not have to find the exact median. It is sufficient to guarantee that the size of none of the two subproblems is more than some fixed fraction of the main problem. The intuition comes from the fact that the solution to any recurrence of the form $T(n) = T(\alpha n) + cn$, where $0 < \alpha < 1$ is a constant, is also $O(n)$.

In *QuickSelect* we select a pivot element randomly. Here we use a *sampling technique*. We select a sample fraction of the n elements, then recursively find the median of that sample. We use the sample-median as a good approximation for the true median to carry out our divide-and-conquer algorithm. But what sample should we take? For instance, suppose we take a 20% sample (i.e., $\lfloor n/5 \rfloor$ of the elements). and find the median of the sample recursively in $T(n/5)$ time. Then use that sample-median as pivot to partition the original n elements in $O(n)$ time. The largest subproblem after the partitioning could be as large as 90% (i.e., 9n/10). (For instance, this would happen if the 20% sample turned out to be the 20% smallest elements of S.) So, if the divide-and-conquer routine is forced to solve that subproblem, that will cost $T(9n/10)$ time. So, the recurrence would be $T(n) = T(n/5) + T(9n/10) + O(n)$. The solution to this recurrence is super-linear.

What can we do? We will not take the 20% sample just arbitrarily. We will spend $O(n)$ time to select the 20% sample set (called M in the algorithm below) more carefully, so that its median is closer to the true median of set S. One such algorithm is procedure *Select* shown below.

```
function Select ( $S$ , $k$ );
(* finds the $k$-th smallest element of sequence $S$ *)
begin
 1)   if $|S| < 50$ then begin
 2)       sort $S$;
 3)       return the $k$-th element in the sorted sequence $S$
 4)   end
 5)   else begin
        (* find a good pivot element $m$ *)
 6)     $g \leftarrow \lfloor |S|/5 \rfloor$;
 7)     divide $S$ into $g$ sequences $M_1, M_2, \cdots, M_g$
            of 5 elements each with up to four leftover elements;
 8)     for $i \leftarrow 1..g$ do sort $M_i$;
 9)     for $i \leftarrow 1..g$ do let $m_i$ be the median of $M_i$;
10)      $M \leftarrow \{ m_1, m_2, \cdots, m_g \}$;
11)      $m \leftarrow Select ( M , \lceil g/2 \rceil )$,        (* the median of $M$ *)

        (* partition and recur by using $m$ as the pivot element *)
12)      $S_1 \leftarrow \{ a \in S \mid a < m \}$;
13)      $S_2 \leftarrow \{ a \in S \mid a > m \}$;
14)      if $|S_1| \geq k$ then return $Select ( S_1 , k )$;
15)         else if $|S| - |S_2| \geq k$ then return $m$;
16)            else return $Select ( S_2 , k - |S| + |S_2| )$
17)   end
end
```

Let us see how *Select* works. Its difference with *QuickSelect* is in choosing the pivot element. To find the pivot element, we divide S into $\lfloor n/5 \rfloor$ groups M_i of 5-elements each. Each group is then sorted and its median m_i is selected. (Note that sorting a 5-element group takes constant time.) Then, at line 11 we invoke a recursive call to *Select* to find the median m of the $\lfloor n/5 \rfloor$ elements, m_i's. Now m is our pivot element. We also notice that if S is sufficiently small (less than 50 elements), we solve the problem by sorting in $O(1)$ time.

The way the pivot element m is selected guarantees that at least one-fourth of the elements of S are less than or equal to m and at least one-fourth of the elements are greater than or equal to m. This is illustrated in *Figure 1*.

The question arises, why the ‘‘magic number’’ 5? The answer is that there are two recursive calls of *Select* (one on line 11, and one on line 14 or 16), each on a sequence a fraction of the size of S. The lengths of the two sequences must sum to less than $|S|$ to make the algorithm work in linear time. Numbers other than 5 will work, but for certain numbers sorting the subsequences

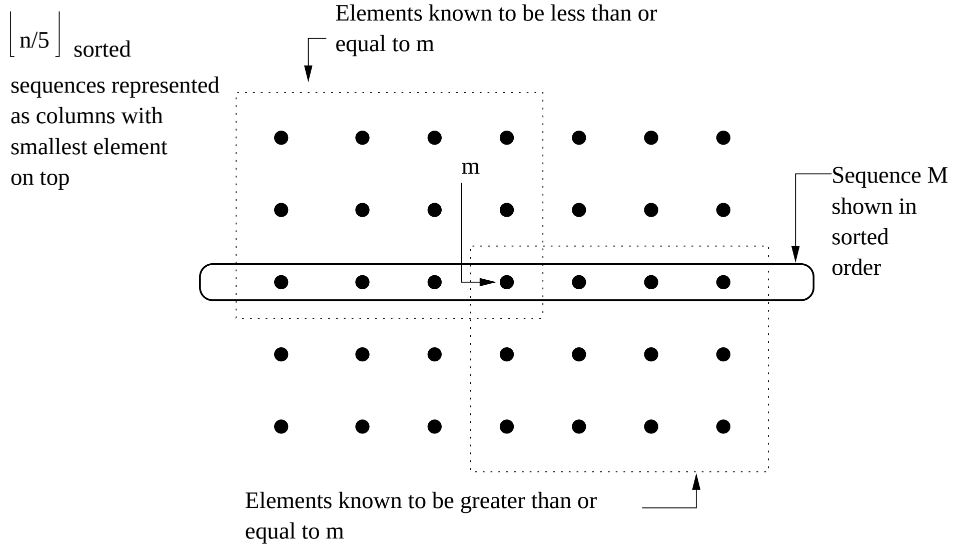

Figure 1: The median of group-medians in the algorithm.

will become expensive. We leave it as an exercise to determine which numbers are appropriate in place of 5.

Theorem. Algorithm *Select* finds the k-th smallest element in a sequence S of n elements in worst-case time $O(n)$.

Proof: The correctness of the algorithm is a straightforward induction on the size of S, and this part of the proof is left for an exercise. Let $T(n)$ be the worst-case time required by algorithm *Select* to select the k-th smallest element from a sequence of size n. The sequence of medians M is of size at most $n/5$ and thus the recursive call at line 11 requires at most $T(n/5)$ time.

Sequences S_1 and S_2 are each of size at most $3n/4$. To see this note that at least $\lfloor n/10 \rfloor$ elements of M are greater than or equal to m, and for each of these elements m_i there are two more elements in M_i, hence in S, which are at least as large. Thus S_1 is of size at most $n - 3\lfloor n/10 \rfloor$, which for $n \geq 50$ is less than $3n/4$. A symmetric argument applies to S_2. Thus the recurrence call at line 14 or 16 requires at most $T(3n/4)$ time. All other statements require at most $O(n)$ time. Thus, for some constant c, we have

$$\begin{aligned} T(n) &\leq cn, && \text{for} \quad n \leq 49\,, \\ T(n) &\leq T(n/5) + T(3n/4) + cn && \text{for} \quad n \geq 50\,. \end{aligned}$$

From the above recurrence we can prove by induction on n that $T(n) \leq 20cn$.

Basis. ($n \leq 49$): In this case clearly $T(n) \leq cn \leq 20cn$.

Inductive Step. ($n \geq 50$): Now assume, by the induction hypothesis, that for all $i < n$ we have $T(i) \leq 20ci$. Then we have $T(n) \leq T(n/5) + T(3n/4) + cn \leq 20c \cdot n/5 + 20c \cdot 3n/4 + cn = 20cn$. The proof is complete. □

Concluding remarks

The average-case linear time algorithm *QuickSelect* is from [C.A.R. Hoare, "*Quicksort,*" Computer Journal 5, pp. 10-15, 1962]. The worst-case linear time algorithm *Select* is from [M. Blum, R.W. Floyd, V.R. Pratt, R.L. Rivest, R.E. Tarjan, "*Time Bounds for Selection,*" Journal of

Computer and System Sciences 7, pp. 448-461, 1972]. You may also consult, for example, pages 126-130 of [Sed88], and chapter 10 of [CLR]. A low level description of algorithm *Select* may be found on page 288 of [A.V. Aho, J.E. Hopcroft, J.D. Ullman, *"Data Structures and Algorithms,"* Addison-Wesley, 1983].

The idea behind the worst-case linear time selection algorithm was subsequently generalized as an algorithmic paradigm by N. Meggido. Efficient solutions to several problems are known using this method which is now known as the *prune-and-search* method. A general description is worth mentioning here:

Prune-and-Search: The idea of this algorithmic paradigm is to perform just enough computation to detect at least a constant fraction of the data as irrelevant and to remove it. After deleting the constant fraction from the data, the algorithm simply recurs. The total amount of time spent by the algorithm turns out to be asymptotically the same as the amount of time for one iteration. This is because the amount of data involved in the sequence of iterations decreases geometrically. It is clear from this description that only problems that produce little output, as opposed to creating an elaborate output structure, qualify as candidates for the application of the prune-and-search paradigm.

Exercises:

(1) **The Facility Location Problem (FLP):** We are given n points in the d dimensional space. We want to compute the location of the optimum facility point whose sum of distances to the given data points is minimized. Imagine we want to establish a communication center (the facility location) and run communication lines from that center to the users (the given points). The objective is to minimize the total length of the communication lines between the center and the users.

(a) Show the one dimentional version of FLP ($d = 1$) can be solved in $O(n)$ time.

(b) How would you solve the problem for $d = 2$ (the planar case) if the communication lines are only allowed to go along horizontal and vertical segments (suppose the communication lines are supposed to follow along the streets which are all North-South or East-West).

(2) **The Majority Problem (MP):** We are given a set S of n elements. A majority value, if it exists, is one that appears more than $n/2$ times in the input set. The problem is to determine if S has a majority element, and if yes, find it.

(a) Suppose we are allowed to compare pairs of elements of S using the comparisons from the set $\{=, \neq, <, \leq, >, \geq\}$. Within this model we can solve MP in $O(n\ lg\ n)$ worst-case time by first sorting S. Describe the rest of the process.

(b) Within the same comparison based model as in part (a), show the problem can be solved in $O(n)$ worst-case time using selection.

(c) Now suppose the only comparisons we are allowed to make are from the set $\{=, \neq\}$. So, we cannot sort. Show how to solve MP in worst-case $O(n\ lg\ n)$ time in this model using divide-and-conquer.

(d) Within the same comparison based model as in part (c), show the problem can be solved in $O(n)$ worst-case time.