Analysis of Algorithms I: Selection and Comparison Lower Bound

Xi Chen

Columbia University

Consider the following problem:

> **Selection**: *Given a sequence* $A[1 \ldots n]$ *and an integer* $i : 1 \leq i \leq n$, *find the* i*th smallest integer in* A.

For $i = 1$ or n, the problem is to find the smallest or the largest element. This clearly can be done in linear time by making a pass over all the n elements and keeping track of the smallest or largest element seen so far. For general i, we can solve the problem by sorting the whole sequence A first (e.g., using Mergesort) and then output $A[i]$. This takes time $O(n \lg n)$. Can we do better?

Here is a divide-and-conquer framework for Selection: (For simplicity we assume that all the elements are distinct.)

1. **Divide**: Pick a pivot x from the sequence. Use it to partition the sequence so that it becomes:

$$A[1], \ldots, A[k-1], A[k] = x, A[k+1], \ldots, A[n]$$

for some k. All elements on the left of $A[k]$ are smaller than x and all elements on the right are larger than x.

2. **Conquer**: Compare i with k. Three cases:
   1. If $i = k$, output x because x is the kth smallest.
   2. If $i < k$, make a recursive call to find the ith smallest in
      $$\text{the lower subsequence } \Big(A[1], \ldots, A[k-1]\Big)$$
   3. If $i > k$, make a recursive call to find the $(i-k)$th smallest in
      $$\text{the upper subsequence } \Big(A[k+1], \ldots, A[n]\Big)$$
3. **Combine**: Trivial.

Using induction, one can prove the correctness of this algorithm (no matter how we pick a pivot in the Divide step).

From what we have learnt from Quicksort, it is not surprising that the performance of this algorithm crucially depends on the way we pick pivots. Again, if we always use the first element in the sequence, then the worst-case running time is quadratic (consider sequences that are already sorted). And again, randomization helps. We show that if pivots are picked uniformly at random (just like in Randomized Quicksort), then the worst-case expected running time is linear. We will refer to this randomized selection algorithm as Randomized-Select.

Theorem

Randomized-Select has worst-case expected running time $O(n)$.

Similar to the analysis of Randomized Quicksort, there are two approaches to prove the theorem above. We present a proof similar to the analysis of Randomized Quicksort described in Exercise 7-3. A different proof using indicator random variables and linearity of expectations can be found in Exercise 9-4.

We use $T(n)$ to denote the worst-case expected running time of Randomized-Select over all sequences of length n. Note that $T(n)$ here is a number instead of a random variable:

$$T(n) = \max_{A \text{ of length } n \text{ and } i \in [n]} E\Big[\text{running time on } (A, i)\Big]$$

where we use $[n]$ to denote $\{1, 2, \ldots, n\}$ for convenience.

Now let $z_1 < z_2 < \cdots < z_n$ denote a reordering of the input sequence. First, the partition step takes time $\Theta(n)$ no matter which element we pick to be the pivot. So we have

$$T(n) \leq \Theta(n) + \text{expected running time of the recursive call}$$

We discuss the following cases:

1. If $x = z_1$, then the expected running time of the recursive call is no more than $T(n-1)$ (this happens if $i > 1$).
2. ...
3. If $x = z_{n/2}$, then the expected running time of the recursive call is no more than $T(n/2)$ (this happens if $i > n/2$).
4. If $x = z_{n/2+1}$, then the expected running time of the recursive call is no more than $T(n/2)$ (this happens if $i \leq n/2$).
5. ...
6. If $x = z_n$, then the expected running time of the recursive call is no more than $T(n-1)$ (this happens if $i < n$).

To summarize, when $x = z_k$, where $k \in [n]$, the expected running time of the recursive call to Randomized-Select is no more than

$$T\big(\max(k-1, n-k)\big) = \begin{cases} T(n-k) & \text{when } k \leq n/2 \\ T(k-1) & \text{when } k > n/2 \end{cases}$$

Because the probability that we pick $x = z_k$, for each $k \in [n]$, is $1/n$, the expected running time is bounded by:

$$T(n) \leq \Theta(n) + \sum_{k=1}^{n} (1/n) \cdot T\big(\max(k-1, n-k)\big)$$

$$\sum_{k=1}^{n} T\big(\max(k-1, n-k)\big)$$

$$= \sum_{k=1}^{n/2} T(n-k) + \sum_{k=n/2+1}^{n} T(k-1)$$

$$= \big(T(n-1) + T(n-2) + \cdots + T(n/2)\big)$$
$$+\big(T(n/2) + T(n/2+1) + \cdots + T(n-1)\big)$$

$$= 2\sum_{k=n/2}^{n-1} T(k)$$

Finally we use the substitution method to show that $T(n) \leq an$ for some constant $a > 0$ to be specified later. First, the basis is trivial if we set a to be large enough so that $a > T(1)$. To prove the induction step, by the recurrence and the inductive hypothesis:

$$\begin{aligned} T(n) &\leq cn + (2/n) \sum_{k=n/2}^{n-1} T(k) \\ &\leq cn + (2/n) \sum_{k=n/2}^{n-1} ak < cn + 3an/4 \end{aligned}$$

It is clear that if we pick $a > 4c$, $T(n) < cn + 3an/4 < an$. This shows that $T(n) = O(n)$, and we finish the proof of Theorem 1.

While Randomized-Quicksort has expected running time $O(n \lg n)$, we know Mergesort is deterministic and also has running time $O(n \lg n)$. So a natural question is, does there exist a deterministic selection algorithm that has linear worst-case running time? The answer is affirmative.

Here is a deterministic method to pick a pivot from $A[1 \ldots n]$ (for simplicity, we assume n is a multiple of 5):

1. Divide A into groups of 5 elements each: for each $i \in [n/5]$, Group i has $A[5(i-1)+1], A[5(i-1)+2], \ldots, A[5i]$.
2. Find the median of each group directly. We use C_i, $i \in [n/5]$, to denote the median of Group i.
3. Recursively select the median x out of $(C_1, \ldots, C_{n/5})$.
4. Partition A using x as a pivot.
5. Recurse on the lower or upper subsequence.

We show that this deterministic selection algorithm has worst-case running time $O(n)$. The key here is that every time we pick an x using the method described in the last slide (Step 1-3), it results in two balanced subsequences. More exactly, we show that if x is the pivot picked, then both the number of elements smaller than x and the number of elements larger than x are at least $3n/10$. This immediately implies (why) that the length of the lower and upper subsequences are both no more than $7n/10$.

To see this, check Figure 9.1 on Page 221, in which we place all the groups with its median smaller than x to the left of x; and all the groups with its median larger than x to the right of x. Because x is the median of the medians, we know that there are roughly speaking $n/10$ groups to the left of x and $n/10$ groups to the right of x. Then all elements in the shadow are larger than x: each of them is larger than the median of its own group, and the median of its group is larger than x. Now how many elements we have in the shadow? $3n/10$ roughly speaking (the shadow is a rectangle with height 3 and width $n/10$).

This gives us the following recurrence:

$$T(n) \leq \Theta(n) + T(n/5) + \Theta(n) + T(7n/10)$$

The first term $\Theta(n)$ is the time we need to find the $n/5$ medians, because for each group it takes constant many steps. The second term $T(n/5)$ is the time we need to recursively find the median x of the $n/5$ medians $(C_1, \ldots, C_{n/5})$. The third term $\Theta(n)$ is the running time of Partition. The last term $T(7n/10)$ is the running time of the recursive call on one of the subsequences, because we have shown that both of them have length at most $7n/10$.

Use the substitution method to solve this recurrence, and we get $T(n) = O(n)$. Question: How about dividing the input sequence into groups of 3 (why 5?) elements each? Doew the worst-case running time remain linear?

Next we come back to sorting. We have seen several sorting algorithms. The best deterministic sorting algorithm we have seen so far, e.g., Mergesort and Heapsort (We will not cover Heapsort in class. Check Chapter 6 if you are not familiar with Heaps.) has worst-case running time $\Theta(n \lg n)$. A natural question is then, can we do better? does every sorting algorithm take time $\Omega(n \lg n)$?

It depends. We show that if an algorithm only uses comparisons to get order information about the input sequence, then $\Theta(n \lg n)$ is the best one can hope. In particular, this bound is achieved, e.g., by Mergesort and Heapsort. In the next class, we will see that if all the input integers fall in a reasonably small range, then there is a linear-time algorithm using no comparison at all! (This contrasts with comparison sorts, e.g., Mergesort and Heapsort, where the running time does not depend on how large the input elements are and we never make any assumption on the range of the input elements. As long as the RAM model is considered, it always takes one step to compare two input elements.)

A comparison sorting algorithm only uses comparisons between elements to gain order information about an input sequence. Such an algorithm behaves like the questioner in 20 questions:

en.wikipedia.org/wiki/Twenty_Questions

In each round the algorithm can pick two elements a_i and a_j and ask which element is larger. Depending on the answer (and all answers to previous questions as well), the algorithm picks again two elements, compare them, and repeat. The algorithm stops until the comparison results so far are sufficient to determine the correct order of the input sequence.

Consider Insertion sort, Mergesort, Heapsort and Quicksort. All of them are comparison sorting algorithms because in any of these algorithms, we only gain order information about the input sequence by comparisons. (Between any two comparisons, these algorithms may reorder the input sequence or make copies of input elements, but gain no order information at all from these operations.)

Theorem

Any deterministic comparison sorting algorithm must make $\Omega(n \lg n)$ comparisons in the worst case.

In the proof, we assume that all the input elements are distinct and thus, every comparison is of the form: "$a_i < a_j$" or "$a_i > a_j$". We show that, over distinct elements, any comparison sorting algorithm makes $\Omega(n \lg n)$ comparisons. Theorem 2 then follows. (If an algorithm makes $\Omega(n \lg n)$ comparisons in the special case of distinct elements, then of course it makes $\Omega(n \lg n)$ comparisons over general sequences that may have equal elements. Note that we are proving a lower bound here.)

An abstract way to describe a comparison sorting algorithm is to use decision trees. (Consider a decision tree as a game plan prepared in advance for 20 questions.) See Figure 8.1 on Page 192. A decision tree for sort n elements is a binary tree in which:

1. every internal node is labeled by $i : j$, where $i \neq j \in [n]$; and
2. every leaf is labeled by a permutation $(\pi_1, \pi_2, \ldots, \pi_n)$ of $[n]$.

Given such a decision tree, we can use it to sort any sequence of n distinct elements $(a_1, \ldots, a_n)$ as follows:

1. Start at the root;
2. If the current node is internal and is labeled $i : j$, we compare a_i with a_j. If $a_i < a_j$, move to the left child; if $a_i > a_j$, move to the right child. Repeat this step until a leaf is reached;
3. If the leaf is labeled with permutation $(\pi_1, \ldots, \pi_n)$, output

$$\left(a_{\pi(1)}, a_{\pi(2)}, \ldots, a_{\pi(n)}\right) \tag{1}$$

We say a decision tree is correct if it always outputs the correct order for any sequence of n distinct elements $(a_1, \ldots, a_n)$.

Every (correct) comparison sorting algorithm can be used to construct, for each $n \geq 1$, a correct decision tree for sorting n elements. Question: How to construct decision trees from a comparison sorting algorithm? and why they are correct? For example, Figure 8.1 is the decision tree of Insertion sort operating on 3 elements. Question: What is the decision tree of Mergesort over 4 elements? Also note that given any comparison sorting algorithm, the worst-case number of comparisons it uses over sequences of length n is exactly the height of its decision tree for sorting n elements. For example, from Figure 8.1, we know that in the worst case, Insertion sort uses 3 comparisons over sequences of length 3.

As a result, if we can show that any correct decision tree for sorting n elements has height $\Omega(n \lg n)$, then any comparison sorting algorithm makes $\Omega(n \lg n)$ comparisons in the worst case.

Theorem

Any correct decision tree for sorting n elements must have height $\Omega(n \lg n)$.

Proof.

The tree must have $\geq n!$ leaves. This is because there are $n!$ permutations over $[n]$. Every such permutation must be labeled at one of the leaves. (If one of the permutation, say $(1, 2, \ldots, n)$, is missing, then the output of the decision tree cannot be correct when the input satisfies $a_1 < a_2 < \cdots < a_n$.) On the other hand, the number of leaves is $\leq 2^h$, where h denote the height. Thus,

$$h \geq \lg(n!) = \Omega(n \lg n).$$

The last step follows from Stirling's approximation on Page 57. □

Stirling's formula gives us a very good approximation for $n!$. But if we only want to show that $\lg(n!) = \Omega(n \lg n)$, here is a proof:

$$\begin{aligned}
\lg n! &= \lg n + \lg(n-1) + \cdots + \lg 1 \\
&> \lg n + \lg(n-1) + \cdots + \lg(n/2) \\
&> (n/2)\lg(n/2) \\
&= (n/2)(\lg n - 1) \\
&= \Omega(n \lg n).
\end{aligned}$$