CS 161 (Stanford, Winter 2026)　　　　Lecture 4

Median and Selection

1　Introduction

So far we have covered the master theorem, which can be used for recurrences of a certain form. Recall that if we have a recurrence $T(n) = aT\left(\frac{n}{b}\right) + O(n^d)$ where $a \geq 1, b > 1$, then

$$
T(n) = \begin{cases} O(n^d \log n) & \text{if } a = b^d \\ O(n^d) & \text{if } a < b^d \\ O(n^{\log_b a}) & \text{if } a > b^d \end{cases}
$$

Many algorithms that result from the divide-and-conquer paradigm yield recurrence relations for their runtimes that have the above form − namely algorithms that divide the problem into equal-sized sub-pieces at each recursion.

Today, we will introduce a problem where the master theorem cannot be applied: the problem of finding the k-th smallest element in an unsorted array. First, we show it can be done in $O(n \log n)$ time via sorting and that any correct algorithm must run in $\Omega(n)$ time. However, it is not obvious that a linear-time selection algorithm exists. We present a linear-time selection algorithm, with an intuition for why it has the desired properties to achieve $O(n)$ running time. The two high-level goals of this lecture are 1) to cover a really cool and surprising algorithm, and 2) illustrate that some divide-and-conquer algorithms yield recurrence relations that cannot be analyzed via the "Master Method/Theorem", yet one can (often) still successfully analyze them.

2　Selection

The selection problem is to find the k-th smallest number in an array A.

Input: array A of n numbers, and an integer $k \in \{1, \ldots, n\}$.

Output: the k-th smallest number in A.

One approach is to sort the numbers in ascending order, and then return the k-th number in the sorted list. This takes $O(n \log n)$ time, since it takes $O(n \log n)$ time for the sort (e.g. by MergeSort) and $O(1)$ time to return k-th number.

2.1 Minimum Element

As always, we ask if we can do better (i.e., faster in big-O terms). In the special case where $k = 1$, selection is the problem of finding the minimum element. We can do this in $O(n)$ time by scanning through the array and keeping track of the minimum element so far. If the current element is smaller than the minimum so far, we update the minimum.

Algorithm 1: SelectMin(A)

$m \leftarrow \infty$
$n \leftarrow$ length(A)
for $i = 1$ *to* n **do**
    if $A[i] < m$ **then**
        $m \leftarrow A[i]$
return m

In fact, this is the best running time we could hope for.

Definition 1. A deterministic algorithm is one which, given a fixed input, always performs the same operations (as opposed to an algorithm which uses randomness).

Proposition 2. *Any deterministic algorithm for finding the minimum has runtime* $\Omega(n)$.

Proof. Intuitively, the claim holds because any algorithm for the minimum must look at all the elements, each of which could be the minimum. Suppose a correct deterministic algorithm does not look at $A[i]$ for some i. Then the output cannot depend on $A[i]$, so the algorithm returns the same value whether $A[i]$ is the minimum element or the maximum element. Therefore the algorithm is not always correct, which is a contradiction. So there is no sublinear deterministic algorithm for finding the minimum. ☐

So for $k = 1$, we have an algorithm which achieves the best running time possible. By similar reasoning, this lower bound of $\Omega(n)$ applies to the general selection problem. So ideally we would like to have a linear-time selection algorithm in the general case.

3 Linear-Time Selection

In fact, a linear-time selection algorithm does exist. Before showing the linear time selection algorithm, it's helpful to build some intuition on how to approach the problem. The high-level idea will be to try to do a Binary Search over an unsorted input. At each step, we hope to divide the input into two parts, the subset of smaller elements of A, and the subset of larger elements of A. We will then determine whether the k-th smallest element lies in the first part (with the "smaller" elements) or the part with larger elements, and recurse on exactly one of those two parts.

How do we decide how to partition the array into these two pieces? Suppose we have a black-box algorithm ChoosePivot that chooses some element in the array A, and we use this pivot to define the two sets–any $A[i]$ less than the pivot is in the set of "smaller" values, and any $A[i]$ greater than the pivot is in the other part. We will figure out precisely how to specify this subroutine ChoosePivot a bit later, after specifying the high-level algorithm structure. The algorithm ChoosePivot does not affect the *correctness* of the algorithm as we will see in Section 3.6. Rather, it only affects the runtime.

For clarity we'll assume all elements are distinct from now on, but the idea generalizes easily. Let n be the size of the array and assume we are trying to find the k-th element.

Algorithm 2: Select(A, n, k)

---

if $n = 1$ **then**
   | return $A[1]$

$p \leftarrow$ ChoosePivot(A, n)
$A_< \leftarrow \{A(i) \mid A(i) < p\}$
$A_> \leftarrow \{A(i) \mid A(i) > p\}$
if $|A_<| = k - 1$ **then**
   | return p
else if $|A_<| > k - 1$ **then**
   | return Select($A_<, |A_<|, k$)
else if $|A_<| < k - 1$ **then**
   | return Select($A_>, |A_>|, k - |A_<| - 1$)

At each iteration, we use the element p to partition the array into two parts: all elements smaller than the pivot and all elements larger than the pivot, which we denote $A_<$ and $A_>$, respectively.

Depending on what the size of the resulting sub-arrays are, the runtime can be different. For example, if one of these sub-arrays is of size $n - 1$, at each iteration, we only decreased the size of the problem by 1, resulting in total running time $O(n^2)$. If the array is split into two equal parts, then the size of the problem at iteration reduces by half, resulting in a linear time solution. (We assume ChoosePivot runs in $O(n)$.)

Proposition 3. *If the pivot p is chosen to be the minimum or maximum element, then* Select *runs in $\Theta(n^2)$ time.*

Proof. At each iteration, the number of elements decreases by 1. Since running ChoosePivot and creating $A_<$ and $A_>$ takes linear time, the recurrence for the runtime is $T(n) = T(n - 1) + \Theta(n)$. Expanding this,

$$T(n) \leq c_1 n + c_1(n - 1) + c_1(n - 2) + ... + c_1 = c_1 n(n + 1)/2$$

and
$$T(n) \geq c_2 n + c_2(n-1) + c_2(n-2) + \ldots + c_2 = c_2 n(n+1)/2.$$
We conclude that $T(n) = \Theta(n^2)$. □

Proposition 4. *If the pivot p is chosen to be the median element, then* Select *runs in $O(n)$ time.*

Proof. Intuitively, the running time is linear since we remove half of the elements from consideration each iteration. Formally, each recursive call is made on inputs of half the size, namely, $T(n) \leq T(n/2) + cn$. Expanding this, the runtime is $T(n) \leq cn + cn/2 + cn/4 + \ldots + c \leq 2cn$, which is $O(n)$. □

So how do we design ChoosePivot that chooses a pivot in linear time? In the following, we describe three ideas.

3.1 Idea #1: Choose a random pivot

As we saw earlier, depending on the pivot chosen, the worst-case runtime can be $O(n^2)$ if we are unlucky in the choice of the pivot at every iteration. As you might expect, it is extremely unlikely to be this unlucky, and one can prove that the *expected* runtime is $O(n)$ provided the pivot is chosen uniformly at random from the set of elements of A. In practice, this randomized algorithm is what is implemented, and the hidden constant in the $O(n)$ runtime is very small.

3.2 Idea #2: Choose a pivot that creates the most "balanced" split

Consider ChoosePivot that returns the pivot that creates the most "balanced" split, which would be the median of the array. However, this is exactly selection problem we are trying to solve, with $k = n/2$! As long as we do not know how to find the median in linear time, we cannot use this procedure as ChoosePivot.

3.3 Idea #3: Find a pivot "close enough" to the median

Given a linear-time median algorithm, we can solve the selection problem in linear time (and vice versa). Although ideally we would want to find the median, notice that as far as correctness goes, there was nothing special about partitioning around the median. We could use this same idea of partitioning and recursing on a smaller problem even if we partition around an arbitrary element. To get a good runtime, however, we need to guarantee that the subproblems get smaller quickly. In 1973, Blum, Floyd, Pratt, Rivest, and Tarjan came up with the Median of Medians algorithm. It is similar to the previous algorithm, but rather than partitioning around the exact median, uses a surrogate "median of medians". We update ChoosePivot accordingly.

Algorithm 3: ChoosePivot(A, n)

Split A into $g = \lceil n/5 \rceil$ groups $p_1, \ldots, p_g$
for $i = 1$ *to* g **do**
    $p_i \leftarrow$ MergeSort(p_i)
$C \leftarrow \{$median of $p_i \mid i = 1, \ldots, g\}$
$p \leftarrow$ Select($C, g, g/2$)
return p

What is this algorithm doing? First it divides A into segments of size 5. Within each group, it finds the median by first sorting the elements with MergeSort. Recall that MergeSort sorts in $O(n \log n)$ time. However, since each group has a constant number of elements, it takes constant time to sort. Then it makes a recursive call to Select to find the median of C, the median of medians. Intuitively, by partitioning around this value, we are able to find something that is close to the true median for partitioning, yet is 'easier' to compute, because it is the median of $g = \lceil n/5 \rceil$ elements rather than n. The last part is as before: once we have our pivot element p, we split the array and recurse on the proper subproblem, or halt if we found our answer.

We have devised a slightly complicated method to determine which element to partition around, but the algorithm remains correct for the same reasons as before. So what is its running time? As before, we're going to show this by examining the size of the recursive subproblems. As it turns out, by taking the median of medians approach, we have a guarantee on how much smaller the problem gets each iteration. The guarantee is good enough to achieve $O(n)$ runtime.

3.3.1 Running Time

Lemma 5. $|A_<| \leq 7n/10 + 5$ *and* $|A_>| \leq 7n/10 + 5$.

Proof. p is the median of $p_1, \cdots, p_g$. Because p is the median of $g = \lceil n/5 \rceil$ elements, the medians of $\lceil g/2 \rceil - 1$ groups p_i are smaller than p. If p is larger than a group median, it is larger than at least three elements in that group (the median and the smaller two numbers). This applies to all groups except the remainder group, which might have fewer than 5 elements. Accounting for the remainder group, p is greater than at least $3 \cdot (\lceil g/2 \rceil - 2)$ elements of A. By symmetry, p is less than at least the same number of elements.

Now,

$$
\begin{aligned}
|A_>| &= \# \text{ of elements greater than } p \\
&\leq (n - 1) - 3 \cdot (\lceil g/2 \rceil - 2) \\
&= n + 5 - 3 \cdot \lceil g/2 \rceil \\
&\leq n - 3n/10 + 5 \\
&= 7n/10 + 5.
\end{aligned}
\tag{1}
$$

By symmetry, $|A_<| \leq 7n/10 + 5$ as well.

Intuitively, we know that 60% of half of the groups are less than the pivot, which is 30% of the total number of elements, n. Therefore, at most 70% of the elements are greater than the pivot. Hence, $|A_>| \approx 7n/10$. We can make the same argument for $|A_<|$. □

The recursive call used to find the median of medians has input of size $\lceil n/5 \rceil \leq n/5 + 1$. The other work in the algorithm takes linear time: constant time on each of $\lceil n/5 \rceil$ groups for MergeSort (linear time total for that part), $O(n)$ time scanning A to make $A_<$ and $A_>$.

Thus, we can write the full recurrence for the runtime,

$$T(n) \leq \begin{cases} c_1 n + T(n/5 + 1) + T(7n/10 + 5) & \text{if } n > 5 \\ c_2 & \text{if } n \leq 5. \end{cases}$$

How do we prove that $T(n) = O(n)$? The master theorem does not apply here. Instead, we will prove this using the substitution method.

3.4   Solving the Recurrence of Select Using the Substitution Method

For simplicity, we consider the recurrence $T(n) \leq T(n/5) + T(7n/10) + cn$ instead of the exact recurrence of Select.

To prove that $T(n) = O(n)$, we guess:

$$T(n) \leq \begin{cases} d \cdot n_0 & \text{if } n = n_0 \\ d \cdot n & \text{if } n > n_0 \end{cases}$$

For the base case, we pick $n_0 = 1$ and use the standard assumption that $T(1) = 1 \leq d$. For the inductive hypothesis, we assume that our guess is correct for any $n < k$, and we prove our guess for k. That is, consider d such that for all $n_0 \leq n < k$, $T(n) \leq dn$.

To prove for $n = k$, we solve the following equation:

$$T(k) \leq T(k/5) + T(7k/10) + ck \leq dk/5 + 7dk/10 + ck \leq dk$$

$$9/10d + c \leq d$$

$$c \leq d/10$$

$$d \geq 10c$$

Therefore, we can choose $d = max(1, 10c)$, which is a constant factor. The induction is completed. By the definition of big-O, the recurrence runs in $O(n)$ time.

3.5 Issues When Using the Substitution Method

Now we will try out an example where our guess is incorrect. Consider the recurrence $T(n) = 2T\left(\frac{n}{2}\right) + n$ (similar to MergeSort). We will guess that the algorithm is linear.

$$T(n) \leq \begin{cases} d \cdot n_0 & \text{if } n = n_0 \\ d \cdot n & \text{if } n > n_0 \end{cases}$$

We try the inductive step. We try to pick some d such that for all $n \geq n_0$,

$$n + \sum_{i=1}^{k} dg(n_i) \leq d \cdot g(n)$$

$$n + 2 \cdot d \cdot \frac{n}{2} \leq dn$$
$$n(1 + d) \leq dn$$
$$n + dn \leq dn$$
$$n < 0,$$

However, the above can never be true, and there is no choice of d that works! Thus our guess was incorrect.

This time the guess was incorrect since MergeSort takes superlinear time. Sometimes, however, the guess can be asymptotically correct but the induction might not work out. Consider for instance $T(n) \leq 2T(n/2) + 1$.

We know that the runtime is $O(n)$ so let's try to prove it with the substitution method. Let's guess that $T(n) \leq cn$ for all $n \geq n_0$.

First we do the induction step: We assume that $T(n/2) \leq cn/2$ and consider $T(n)$. We want that $2 \cdot cn/2 + 1 \leq cn$, that is, $cn + 1 \leq cn$. However, this is impossible.

This doesn't mean that $T(n)$ is not $O(n)$, but in this case we chose the wrong linear function. We could guess instead that $T(n) \leq cn - 1$. Now for the induction we get $2 \cdot (cn/2 - 1) + 1 = cn - 1$ which is true for all c. We can then choose the base case $T(1) = 1$.

3.6 Correctness of the Algorithm

Recall that the choice of pivot only affects the runtime, and not the correctness of the algorithm. Here, we prove formally, by induction, that Select is correct. We will use *strong induction*. That is, our inductive step will assume that the inductive hypothesis holds for *all n* between 1 and $i - 1$, and then we'll show that it holds for $n = i$.

Remark 6. You can also do this using regular induction with a slightly more complicated inductive hypothesis; either way is fine.

Inductive Hypothesis (for n). When run on an array A of size n and an integer $k \in \{1, \ldots, n\}$, Select returns the k-th smallest element of A.

Base Case ($n = 1$). When $n = 1$, the requirement $k \in \{1, \ldots, n\}$ means that $k = 1$; that is, Select(A, k) is supposed to return the smallest element of A. This is precisely what the pseudocode above does when $|A| = 1$, so this establishes the Inductive Hypothesis for $n = 1$.

Inductive Step. Let $i \geq 2$, and suppose that the inductive hypothesis holds for all n with $1 \leq n < i$. Our goal is to show that it holds for $n = i$. That is, we would like to show that

> When run on an array A of size i and an integer $k \in \{1, \ldots, i\}$, Select(A, k) returns the k-th smallest element of A.

Informally, we want to show that assuming that Select "works" on smaller arrays, then it "works" on an array of length n.

We do this below:

Suppose that $1 \leq k \leq i$, and that A is an array of length i. There are three cases to consider, depending on $p = $ ChoosePivot(A, i). Notice that in the pseudocode above, p is a value from A, not an index. Let $A_<, A_>, p$ be as in the pseudocode above.

- **Case 1.** Suppose that $|A_<| = k - 1$. Then by the definition of $A_<$, there are $k - 1$ elements of A that are smaller than p, so p must be the k-th smallest. In this case, we return p, which is indeed the k-th smallest.

- **Case 2.** Suppose that $|A_<| > k - 1$. Then there are more than $k - 1$ elements of A that are smaller than p, and so in particular the k-th smallest element of A is the same as the k-th smallest element of L. Next we will use the inductive hypothesis for $n = |A_<|$, which holds since $|A_<| < i$. Since $1 \leq k \leq |A_<|$, the inductive hypothesis implies that Select$(A_<, k)$ returns the k-th smallest element of $A_<$. Thus, by returning this we are also returning the k-th smallest element of A, as desired.

- **Case 3.** Suppose that $|A_<| < k - 1$. Then there are fewer than $k - 1$ elements that are less than p, which means that the k-th smallest element of A must be greater than p; that is, it shows up in $A_>$. Now, the k-th smallest element in A is the same as the $(k - |A_<| - 1)$-st element in $A_>$. To see this, notice that there are $|A_<| + 1$ elements smaller than the k-th that do *not* show up in $A_>$. Thus there are $k - (|A_<| + 1) = k - |A_<| - 1$ elements in $A_>$ that are smaller than or equal to the k-th element. Now we want to apply the inductive hypothesis for $n = |A_>|$, which we can do since $|A_>| < i$. Notice that we have $1 \leq k - |A_<| - 1 \leq |A_>|$; the first inequality holds because $k > |A_<| + 1$ by the definition of Case 3, and the second inequality holds because it is the same as $k \leq |A_<| + |A_>| + 1 = n$, which is true by assumption. Thus, the inductive hypothesis implies that Select$(A_>, k - |A_<| - 1)$ returns the $(k - |A_<| - 1)$-st element of $A_>$. Thus, by returning this we are also returning the k-th smallest element of A, as desired.

Thus, in each of the three cases, Select(A, k) returns the k-th smallest element of A. This establishes the inductive hypothesis for $n = i$.

Conclusion. By induction, the inductive hypothesis holds for *all* $n \geq 1$. Thus, we conclude that Select(A, k) returns the k-th smallest element of A on any array A, provided that $k \in \{1, \ldots, |A|\}$. That is, Select is correct, which is what we wanted to show.