COMP 285 (NC A&T, Spr '22) Lecture 8

Median of Medians and Randomized Algorithms

1 Introduction

We ended last lecture with a few ideas on how we could solve the kSelect problem. The only difference between the ideas involved how we chose the 'pivot'! In this lecture, we'll cover a deterministic mechanism for chosing a pivot. As a refresher, here is the pseudo-code for the divide-and-conquer approach to the kSelect problem.

Algorithm 1: Select(A, n, k)

```
if $n = 1$ then
  return $A[1]$
end if
$p \leftarrow$ 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$(A_>, |A_>|, k - |A_<| - 1)$
end if
```

Lastly, we'll introduce *randomized algorithms* and discuss a simple one used only for teaching purposes called `BogoSort`.

2 Choose 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 algorithm we covered in the last lecture, but rather than partitioning around the exact median, uses a surrogate "median of medians". We update `ChoosePivot`accordingly.

Algorithm 2: ChoosePivot(A, n)

```
Split $A$ into $g = \lceil n/5 \rceil$ groups $p_1, \cdots, p_g$
for $i = 1$ to $g$ do
    $p_i \leftarrow$ MergeSort($p_i$)
end for
$C \leftarrow \{$median of $p_i \mid i = 1, \cdots, g\}$
$g \leftarrow$ 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.

2.1 Running Time

Lemma. $|A_<| \leq 7n/10 + 5$ 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,

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

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.

2.2 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:

$$\begin{aligned} T(k) &\leq T(k/5) + T(7k/10) + ck \\ &\leq dk/5 + 7dk/10 + ck \\ \implies 9/10d + c &\leq dk \\ \implies c &\leq d/10 \\ \implies d &\geq 10c \end{aligned}$$

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

2.3 Isssues When Using the Substitution Method

Now we will try out an example where our guess is incorrect. Consider the recurrence $T(n) = 2T(n/2) + n$ (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 $n \geq n_0$,

$$\begin{aligned} n + \Sigma_{i=1}^{k} dg(n_i) &\leq d \cdot g(n) \\ n + 2 \cdot d \cdot \frac{n}{2} &\leq dn \\ n(1+d) &\leq dn \\ n + dn &\leq dn \\ n &< 0 \end{aligned}$$

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 $T(1) = 1$.

2.4 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. 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 $k \in \{1, \cdots, n\}$, Select returns the k-th smallest element of A.

Base Case ($n = 1$). When $n = 1$, the requirement $k \in \{1, \cdots, n\}$ 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 $k \in \{1, \cdots, i\}$, 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 $k \in \{1, \cdots, |A|\}$. That is, `Select` is correct, which is what we wanted to show.

3 Randomized Algorithms

We have already seen and discussed one option to selecting the pivot, which involves selecting it at random. This can be done in $O(1)$ time, and rather simply, leading to an algorithm that is almost always faster than even our complex version above.

This idea of using randomization to improve the performence of algorithms in practice occurs often in algorithms. Frequently, introducing randomization makes the algorithm both (1) faster in practice and (2) simplifies the implementation. The only thing we trade-off is how confident we are in whether the algorithm is correct.

Oftentime, randomized algorithms require a high-level of analysis to determine correctness, and also requires a significant amount of dedication to figure out their running times.

3.1 What happens when you introduce randomness?

There are two types of randomized algorithms. The first type is an algorithm that uses randomness internally, but always return the correct answer. The algorithm above using a random pivot is an example of this type of algorithm. The randomness itself does not effect the correctness of the algorithm, and instead affects the runtime of the algorithm.

Another type of randomized algorithm, which we will cover later in the class, uses randomness internally but guarantees a particular runtime. These algorithms often sacrifice correctness (they're not correct all of the time), but their running time is not random.