COMP 285 (NC A&T, Spr '22) Lecture 7

Adapted From Virginia Williams' lecture notes. Additional credits: J. Su, W. Yang, Gregory Valiant, Mary Wootters, Aviad Rubinstein, Sami Alsheikh.

k-Select Problem & Median Selection

1 Introduction

In the last lecture, we wrapped up with the subtitution method for solving recurrences and introduced the selection problem. In this lecture, we find an $O(n)$ algorithm to solve the selection problem.

2 Selection Problem

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

Input: array A of n numbers, and an integer $k \in \{1, \cdot, n\}$. **Output**: the k-th smallest number in A.

One approach is to sort the numbers in ascending order, and then return the kth 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 kth 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 ← ∞
n ← length(A)
for i = 1 to n do
  if A[i] < m then
    m ← A[i]
  end if
end for
```

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

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

Claim. *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 2. 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.

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

Algorithm 2: Select(A, n, k)

```
if n = 1 then
  return A[1]
end if
p ← ChoosePivot(A, n)
A_< ← {A[i] | A[i] < p}
A_> ← {A[i] | 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
```

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. *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

$$T(n) \geq c_2 n + c_2(n-1) + c_2(n-2) + ... + c_2 = c_2 n(n+1)/2.$$

We conclude that $T(n) = \Theta(n^2)$.

Proposition. *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 $T(n) \leq cn + cn/2 + cn/4 + ... + c \leq 2cn$, 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`.