COMP 285 (NC A&T, Spr '22) Lecture 6

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

Substitution Method & k-Select Problem

1 Introduction

In the last lecture, we covered the Master Theorem to solve **recurrence relations**. This method works well in cases where the sub-problems are of equal size (which is most of the time). In this lecture, we'll cover a more advanced method for solving more complicated recurrence relations.

This method is required to understand and proof the running time of the k-Select Problem, which we will also introduce.

2 The Subtitution Method

Recurrence trees can get quite messy when attempting to solve complex recurrences. With the substitution method, we can guess what the runtime is, plug it in to the recurrence and see if it works out.

Given a recurrence $T(n) \leq f(n)+\Sigma_{i=1}^{k} T(n_i)$, we can guess that the solution to the recurrence is

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

for some constants $d > 0$ and $n_0 \geq 1$ and a function $g(n)$. We are essentially guessing that $T(n) \leq O(g(n))$.

For our base case we must show that you can pick some d such that $T(n_0) \leq d \cdot g(n_0)$. For example, this can follow from our standard assumption that $T(1) = 1$.

Next we assume that our guess is correct for everything smaller than n, meaning $T(n') \leq d \cdot g(n')$ for all $n' < n$. Using the inductive hypothesis, we prove the guess for n. We must pick some d such that

$$f(n) + \sum_{i=1}^{k} d \cdot g(n_i) \leq d \cdot g(n), \text{ whenever } n \geq n_0$$

Typically the way this works is that you first try to prove the inductive step starting from the inductive hypothesis, and then from this obtain a condition that d needs to obey. Using this condition you try to figure out the base case, i.e., what n_0 should be.

3 Selection

The selection problem is to find the kth 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.

3.1 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.

3.2 Linear-Time Selection

In fact, a linear-time selection algorithm does exist, and we will cover it in next lecture.