Algorithms

Margaret M. Fleck

17 March 2010

This lecture starts the discussion of analyzing the running time of algorithms. This material is in sections 3.1 and 3.3 of Rosen. This was a half-lecture due to the quiz.

1 Introduction

The main reason for studying big-O notation and solving recurrences is so that we can predict how fast different computer programs will run. This allows us to choose good designs for different practical applications. We will normally do these analyses only up to big-O, i.e. ignoring multiplicative constants and considering only the terms that dominate the running time.

Over the next three lectures, we'll look at basic searching and sorting algorithms, plus one or more other algorithms with similar structure (e.g. the Towers of Hanoi solver).

When analyzing these algorithms, we'll see how to focus in on the parts of the code that determine the big-O running time and where we can cut corners to keep our analysis simple. Such an approach may seem like sloppiness right now, but it becomes critical as you analyze more and more complex algorithms in later CS classes.

2 Linear Search

Suppose that we have an array of real numbers a_1, a_2, $\ldots a_n$ and an input number b. The goal of a search algorithm is to find out whether b is in the list and, if so, at which array index it resides. Let's assume, for simplicity, that the list contains no duplicates or (if it does) that our goal is to return

the first array position containing the value b. The values in our input array might be in any random order. For example, if our input array contains (in order) 7, 3, 17, -2 and $b = 17$, we would return 3. If $b = 27$, we would return some special value to indicate that b isn't in the array. I'll pick zero for our special value.

The input "array" in our pseudocode might actually be implemented as either an array or as a linked list in a real programming language. It's often an array because these tend to be more efficient

The pseudo-code for linear search looks like:

1: linearsearch(a_1,..., a_n: reals, b real)

2: i = 1

3: while ($a_i \neq$ b and i < n)

4: i = i + 1

5: if (a_i = b) return i

6: else return 0

Notice that the input array starts with position 1. This is common in pseudocode, though most programming languages have array indices that start with zero. Also notice that the length of the array is given somewhat implicitly as the subscript on a_n.

The main loop in lines 3-4 walks through the array from start to end, looking for a value that matches b. The loop stops when it finds such a value ($a_i = b$) or when we are about to go past the end of the input array ($i \geq n$).

To analyze this code in big-O terms, first notice that the start-up code in lines 1-2 and the ending code in lines 5-6 takes the same amount of time regardless of the input size n. So we'll say that it takes "constant time" or $O(1)$ time. The big-O running time of linear search depends entirely on the behavior of the loop in lines 3-4, because the number of loop iterations does depend on n. In each iteration of the loop, we do a constant amount of work. So we just have to figure out how many times the loop runs, as a function of n.

3 Worst case and average case analysis

So the big-O analysis would be easy except for the fact that the number of time the loop runs depends on how quickly we find a matching value in the array. If b is near the start of the array, the code stops quickly. If b is near the end or not in the array at all, the code runs for n or close to n iterations.

This sort of variable running time is quite common in computer algorithms. There are two ways to handle it: worst case analysis and average case analysis.

In a worst-case analysis, we pick some random value for the input size n and ask what is the worst possible input of that size. That is, which input of size n will make the program run slowest. For linear search, it's an input where b is at the end of the array or not in the array. So, in the worst case, linear search requires n loop iterations. So its worst-case running time is $O(n)$.

In an average-case analysis, we again fix n but then work out the average behavior of the code on all inputs of size n. To do this, we need to model how often we expect to see different types of inputs.

First, let's suppose that n is in the array. For linear search, it's normally reasonable to assume that different orderings of the array occur equally often. So b is as likely to be at position 37 as at position 8. In this case, the average number of times our loop runs is

$$\frac{1+2+\ldots+n}{n} = \frac{1}{n}\sum_{k=1}^{n} k = \frac{n(n+1)}{2n} = \frac{n+1}{2} = O(n)$$

If b might not be in the array, we'd need to add in a term for this possibility, weighted by how frequently we expect b to be missing. This is very application-dependent, but the running time will still be $O(n)$ because the loop runs n times whenever b is not found.

If we expected some particular ordering to our input array, then the average running time might be very different. For example, in some applications, it's common to build the array by adding new items onto the front and then soon afterwards search the array for new copies of the same item. In this case, the average search time might be substantially less, perhaps even not $O(n)$. But such applications are unusual.

4 Binary search

We briefly saw the idea behind binary search. When looking for a value in a sorted array, compare it to the value in the middle position in the array. You can then restrict your search to either the first half or the second half of the array. More details next class.