Algorithms

Margaret M. Fleck

19 October 2009

This lecture starts the discussion of analyzing the running time of algorithms. This material is in sections 3.1 and 3.3 of Rosen.

1 Announcements

There's a quiz coming on Wednesday (21 October). Study materials are available on the web.

2 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.

3 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,$\ldots$, 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.

4 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 thise

case, the average search time might be substantially less, perhaps even not $O(n)$. But such applications are unusual.

5 Binary search

In many applications, it's sensible to assume that the input array is sorted. That is $a_1 \leq a_2 \leq \ldots \leq a_n$. We'll see that it would take $O(n \log n)$ time to sort an unsorted array. But sometimes an array of data comes pre-sorted. And sometimes an application will do many searches on an array of data that's constant or changes only very slowly, e.g. the registrar's big database of student records. In such cases, the cost of sorting the array may be insignificant compared to the cost of doing the searches.

When our input array is sorted, we can use a much more efficient algorithms: binary search. Let's assume that our array is sorted with the smallest element at the start. (The same idea works, with small code changes, if the array is sorted in reverse order.)

```
1: binarysearch(a_1,..., a_n: sorted array of reals, b real)
    2: i = 1
    3: j = n
    4: while (i < j)
        5: m = ⌊(i + j)/2⌋
        6: if (b > a_m)
            7: then i = m+1
            8: else j = m
    9: if (a_i = b) return i
    10: else return 0
```

The two variables i and j define a range of array positions. At the start of the function, this range is all of the input array. In each iteration of the loop, we shrink the range in half, either by making i larger or by making j smaller. The loop terminates when the range contains only a single array position i.

To do the shrinking, we examine the value at the middle position in the current range. If it's smaller than b, we know that b must live in the upper half of the range (if it's in the array at all). Otherwise, b must live in the lower half of the range (again, if it's in the array).

6 Analysis of binary search

To do a big-O analysis of binary search's running time, notice that the code in lines 1-3 and 9-10 takes constant time, i.e. the same amount of time regardless of the input array size. So it's $O(1)$ time and doesn't matter to our big-O analysis. The code inside the loop (lines 5-8) also takes constant time. So the running time of the code (as a function of n) is determined by how many times the loop runs.

To figure out how many times the loop runs, first suppose that n is a power of two, e.g. $n = 2^k$. Since the range starts at length n and is cut in half in each iteration, it will take k iterations before the range has length 1 and the loop stops. If you have to do such an analysis really fast, e.g. on an exam, you might be uncertain about whether the loop really runs k times or perhaps $k + 1$ or $k - 1$ times. This difference doesn't matter: it's still $O(k)$ times. Since $k = \log_2 n$, this means that the loop runs $O(\log n)$ times.

If n isn't a power of two, we can get an upper bound on the running time. The code run on an array of size n will clearly take no more time than if it were run on an array of size 2^k where 2^k is the next power of two larger than n. So, the loop still runs $O(\log n)$ times.

So, the overall running time of binary search is $O(\log n)$, where n is the length of the input array. In general, if an algorithm works by reducing the size of its problem by a factor of two (or three or four ...), it probably takes $O(\log n)$ time.

7 Bubble sort

A sorting algorithm takes an input array of real numbers and puts them into sorted order, e.g. smallest number first. Sorting algorithms are not actually restricted to numbers. With some minor changes, they can operate on any sort of data where you can precisely define an ordering. For example, they can be used to sort strings (e.g. names) into dictionary order. We'll stick to

real numbers for simplicity.

The bubble sort algorithm is an especially stupid sorting method. So stupid, in fact, that you're unlikely to see it outside of theory classes. But it is a useful starting point for looking at the running times of sorting algorithms.

1: bubblesort(a_1,..., a_n: array of reals)

2: for j = 1 to $n-1$

3: for k = 1 to $n-j$

4: if ($a_k > a_{k+1}$)

· 5: swap a_k and a_{k+1}

This function doesn't return anything explicitly. Rather, it rearranges the values within its input array, so the input array will be nicely sorted after the function returns. (Rather like organizing someone's closet for them.)

The swap command in line 5 interchanges two values in the array. To implement this in most programming languages, you'd need to put one of the values into a temporary variable e.g. $t = a_k$, then move the other value into the first one's e.g. $a_k = a_{k+1}$, then put in the new second value e.g. $a_{k+1} = t$. Normally we hide this gory detail when writing pseudocode.

Bubblesort walks repeatedly through the input array, fixing local problems in the ordering. Some versions of bubblesort just keep sweeping through the entire array repeatedly until they do a complete sweep without finding any more local problems.

This version of bubblesort is a bit smarter. After the first time through the outer loop (line 2), we know that the largest value must have worked its way all the way down to the last array position. So this last position has its final value and so we can ignore it in subsequent iterations. After the second time through the outer loop. the last two positions are ok and can be left alone. This is why the top index in the inner loop (line 3) keeps decreasing.

The running time of this algorithm is proportional to how many times the code inside both loops (lines 4-5) runs. This is

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

When reversing a summation like this, it would be easy to get something off by one, e.g. the top or bottom bound. Writing out the terms of the summation is helpful for checking this.