Algorithms

Margaret M. Fleck

18 October 2010

These notes cover how to analyze the running time of algorithms (sections 3.1, 3.3, 4.4, and 7.1 of Rosen).

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.

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.

In lecture, we typically do demos of how these algorithms work. Similar demos are easy to find on the web, if you aren't sure you understand how the pseudocode works.

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 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.

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

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 algorithm: 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).

5 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.

6 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.

7 Insertion sort

Like bubble sort, insertion sort works by side-effect: it doesn't return a value but, instead, rearranges the numbers within an array. The idea behind

insertion sort is to divide the input array (conceptually) into two pieces: a portion that has been sorted and a portion that isn't sorted yet. At the start, the sorted portion contains only a single number: the original first number in the array. We add the other numbers one by one, inserting each one into its proper place in the growing sorted list.

When we add some new number a_j to the sorted part of the array, we first remove it from the array. Then we first need to find its proper position in the sorted portion. Then we usually need to shove some of the sorted values to the right to make a space to insert a_j. Then we actually insert a_j.

Our pseudo-code looks like

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

2: for j = 2 to n

{

3: t = a_j

4: i = 1

5: while ($a_i < t$)

6: i = i + 1

7: for k = j down to i + 1

8: $a_k = a_{k-1}$

9: $a_i = t$

}

The inside of the big loop moves the value a_j from the unsorted part of the array into the sorted part. Lines 4-6 locate the place where a_j belongs in the sorted portion, which will be right before the value a_i.

Lines 7-8 then shove the sorted values starting with a_i over to the right to make a blank space, into which line 9 puts a_j.

Notice that the loop index in line 7 moves downwards rather than upwards. In some programming languages, you can just do this directly. In others, you (annoyingly) need to use an upward-moving loop variable and do some algebra on the index values to derive the downward-moving indices.

8 Running time of insertion sort

Suppose that $T(n)$ is the worst-case running time for insertion sort. That is, for each input array length n, $T(n)$ is the maximum time that insertion sort might take on any array of that length. Notice that we now have two closely-related functions floating around: insertionsort and its running time T.

Insertion sort has two nested loops. This should immediately suggest to you: $O(n^2)$ time. So let's check out the details of the loops to see if that's right.

The inner while loop (line 5-6) runs no more than j times. If it hasn't already halted earlier, it definitely has to halt when $i = j$. To be really precise, the test for this loop (line 5) gets executed no more than j times and the material inside the loop (line 6) no more than $j - 1$ times. Similarly, the work inside the for loop (lines 7-8) takes at most j time.

So, the code inside the loops runs no more than $\sum_{j=2}^{n} j = \sum_{j=1}^{n-1}(j+1) = (n-1) + \frac{n(n-1)}{2} = O(n^2)$

Other parts of the code run less often. E.g. the loop index computation and testing in line 2 takes $O(n)$ time. And there's probably some constant-time work at the start and end of the function. But this work is dominated by the $O(n^2)$ work in the inner loop.

So, $T(n) = O(n^2)$.

9 Linked Lists

So far, all of our sorting and searching algorithms have stored their data in arrays. The next algorithm, merge sort, can be implemented using arrays but is more often described using linked lists. Like an array, a linked list stores an ordered sequence of objects (e.g. numbers). However, its length is flexible and it's easy to add and remove objects.

When you take the data structures course, you'll learn how to implement several varieties of linked lists. In these notes, we'll ignore the implementation details. To keep things simple, we'll assume a generic and fairly flexible type of linked list.

A linked list starts with its *head* and ends with its *tail*. For example, if our list is $(1, 7, 3, 4, 7, 19)$, then the value at the head end is 1 and the value at the tail end is 19. Adding or removing a single element at either end of the

list takes constant (aka $O(1)$) time. [1] Elements in the middle of the list are accessed by starting at the head and walking down the list. So retrieving or changing the value in the nth list position takes $O(n)$ time, as does deleting the nth item from the list.

10 Merge sort

Mergesort takes an input linked list of numbers and returns a new linked list containing the sorted numbers. This is somewhat different from bubble and insertion sort, which rearrange the values within a single array (and don't return anything).

Merge sort is a so-called divide-and-conquer algorithm. That is, it takes a big problem (sorting a list of length n) and divides it into two smaller problems (sorting two lists of length $n/2$). We divide up the list repeatedly until we have a large number of very short lists, of length 1 or 2 (depending on the preferences of the code writer). A length-1 list is necessarily sorted. A length 2 list can be sorted in constant time.

Then, we take all these small sorted lists and merge them together in pairs, gradually building up longer and longer sorted lists until we have one sorted list containing all of our original input numbers.

The big trick here is merging two sorted lists, $a_1, \ldots, a_n$ and $b_1, \ldots, b_m$. To do this, we make a new third list to contain our merged output. Then we examine the first elements of the two input lists and move the smaller value onto our output list. We keep looking at the first elements of both lists until one list is empty. We then copy over the rest of the non-empty list.

This merger process is very efficient, because each merger step only examines the two values at the heads of the lists. And, in fact, the running time of merge sort is $O(n \log n)$, which is the best possible big-O running time for a sorting algorithm.

11 Pseudo-code for mergesort

The pseudocode for mergesort might look as follows:

[1] If you do know something about data structures, this is only strictly true for doubly-linked lists. However, many algorithms, including merge sort, can also be implemented with singly-linked lists, either by keeping a pointer to the tail of the list or by strategic reversal of the list order.

1. mergesort($L = a_1, a_2, \ldots, a_n$: list of real numbers)
    2. if ($n = 1$) then return L
    3. else
        4. $m = \lfloor n/2 \rfloor$
        5. $L_1 = (a_1, a_2, \ldots, a_m)$
        6. $L_2 = (a_{m+1}, a_{m+2}, \ldots, a_n)$
        7. return merge(mergesort(L_1),mergesort(L_2))

And the pseudocode for the merge function might look like the following, assuming that head(L) returns the value at the head of a list L.

1. merge(L_1,L_2: sorted lists of real numbers)
    2. O = emptylist
    3. while (L_1 is not empty or L_2 is not empty)
        4. if (L_1 is empty)
            5. move head(L_2) to the tail of O
        6. else if (L_2 is empty)
            7. move head(L_1) to the tail of O
        8. else if (head(L_1) <= head(L_2))
            9. move head(L_1) to the tail of O
        10. else move head(L_2) to the tail of O
    11. return O

12 Analysis of mergesort

Now, let's do a big-O analysis of mergesort's running time. We need to start by looking at the merge function. For merge, a good way to measure the size of the input n is that n is the sum of the lengths of the two input arrays. Or, equivalently, n is the length of the output array.

The merge function has one big loop. Since the operations within the loop all take constant time, we just need to figure out how many times the loop runs. Each time the loop runs, we move one number from L_1 or L_2 onto

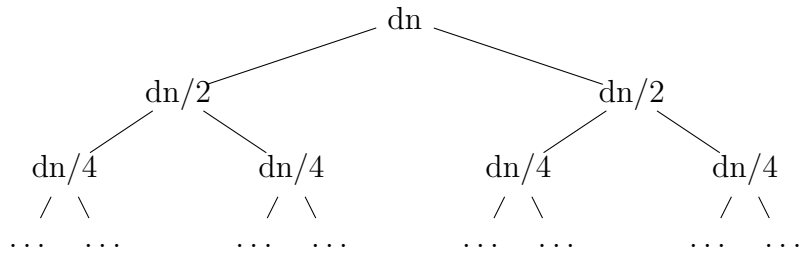
the output list O. n numbers have to be moved, in total. So the loop must run n times. So merge takes $O(n)$ (aka linear) time.

In this analysis of merge, we have a resource: the numbers in the list. This resource has a known size and is being consumed at some known rate. For this algorithm, analyzing the consumption of the resource is the simplest way to figure out how many times the critical loop will execute.

Now, let's analyze the running time of mergesort on an input of length n. Mergesort makes two recursive calls to itself. It also does $O(n)$ work dividing the list in half, because it must walk, element by element, from the head of the list down to the middle position. And it does $O(n)$ work merging the two results. So if the running time of mergesort is $T(n)$, we can write the following recurrence for $T(n)$, where c and d are constants.

- $T(1) = c$
- $T(n) = 2T(n/2) + dn$

This recurrence has the following recursion tree:



The tree has $O(\log n)$ non-leaf levels and the work at each level sums up to dn. So the work from the non-leaf nodes sums up to $O(n \log n)$. In addition, there are n leaf nodes (aka base cases for the recursive function), each of which involves c work. So the total running time is $O(n \log n) + cn$ which is just $O(n \log n)$.

13 The practical truth for sorting algorithms

The worst-case big-O running time of merge sort is as good as you can do. However, mergesort's constants are only so-so. A better practical choice in most circumstances is a different $O(n \log n)$ algorithm called heap sort.

On small to moderately-sized sets of data, the algorithm of choice is actually quick sort, which we won't discuss in this class because it requires a more complicated, statistical analysis. Quicksort has a $O(n^2)$ worst-case running time, but a $O(n \log n)$ average running time and good constants. Most built-in "sort" functions use quicksort.

Aside from making an nice example in theory classes, mergesort is used for certain specialized sorting tasks. These include applications where data is already stored in a linked list and applications requiring a "stable sort" in which elements with identical value maintain their relative positions. Because the key merge step accesses its three lists in storage order, variations of this technique are used when access to the data is very slow. In the Bad Old Days of ancient computers, "slow storage" usually involved medium-sized datasets stored on magnetic tapes. These days, slow access typically involves data that is distributed across several processors or across the internet, e.g. extremely large datasets.

14 Tower of Hanoi

The Tower of Hanoi puzzle was invented by the French mathematician Edouard Lucas in 1883. It consists of three pegs and a set of disks of graduated size that fit on them. The disks start out in order on one peg. The goal is to rebuild the ordered tower on another peg without ever placing a disk on top of a smaller disk.

Show the game and its solution using the animation at:

`http://www.mazeworks.com/hanoi/`

The best way to understand the solution is recursively. Suppose that we know how to move k disks from one peg to another peg, using a third temporary-storage peg. To move $k+1$ disks from peg A to peg B using a third peg C, we first move the top k disks from A to C using B as temporary storage. Then we move the biggest disk from A to B. Then we move the other k disks from C to B, using A as temporary storage.

So our recursive solver looks like

1. hanoi(A,B,C: pegs, $d_1, d_2 \ldots d_n$: disks)
    2. if $(n = 1)$ move $d_1 = d_n$ from A to B.
    3. else
        4. hanoi(A,C,B,$d_1, d_2, \ldots d_{n-1}$)
        5. move d_n from A to B.
        6. hanoi(C,B,A,$d_1, d_2, \ldots d_{n-1}$)

The function hanoi breaks up a problem of size n into two problems of size $n-1$. Alert! Warning bells! This can't be good: the sub-problems aren't much smaller than the original problem!

Anyway, hanoi breaks up a problem of size n into two problems of size $n-1$. Other than the two recursive calls, it does only a constant amount of work. So the running time $T(n)$ for the function hanoi would be given by the recurrence (where c and d are constants):

- $T(1) = c$
- $T(n) = 2T(n-1) + d$

If we unroll this recurrence, we get

$$\begin{aligned} T(n) &= 2T(n-1) + d \\ &= 2 \cdot 2(T(n-2) + d) + d \\ &= 2 \cdot 2(2(T(n-3) + d) + d) + d \\ &= 2^3 T(n-3) + 2^2 d + 2d + d \\ &= 2^k T(n-k) + d \sum_{i=0}^{k-1} 2^i \end{aligned}$$

We'll hit the base case when $k = n - 1$. So

$$\begin{aligned} T(n) &= 2^k T(n-k) + d\sum_{i=0}^{k-1} 2^i \\ &= 2^{n-1}c + d\sum_{i=0}^{n-2} 2^i \\ &= 2^{n-1}c + d(2^{n-1}-1) \\ &= 2^{n-1}c + 2^{n-1}d - d) \\ &= O(2^n) \end{aligned}$$

15 Multiplying big integers

Suppose we want to multiply two integers. Back in the Bad Old Days of slow computers, we would need to multiply moderate-sized integers digit-by-digit. These days, we typically have a CPU that can multiply two moderate-sized numbers (e.g. 16-bit, 32-bit) as a single operation. But some applications (e.g. in cryptography) involve multiplying very, very large integers. Each very long integer must then be broken up into a sequence of 16-bit or 32-bit integers.

Let's suppose that we are breaking up each integer into individual digits, and that we're working in base 2. The recursive multiplication algorithm divides each n-digit input number into two $n/2$-digit halves. Specifically, suppose that our inputs numbers are x and y and they each have $2m$ digits. We can then divide them up as

$$x = x_1 2^m + x_0$$
$$y = y_1 2^m + y_0$$

If we multiply x by y in the obvious way, we get

$$xy = A2^{2m} + B2^m + C$$

where $A = x_1 y_1$, $B = x_0 y_1 + x_1 y_0$, and $C = x_0 y_0$. Set up this way, computing xy requires multiplying four numbers with half the number of digits.

So, if you just count the number of multiplications, the running time of this naive method has a recurrence

$$T(n) = 4T(n/2) + O(n)$$

If you solve this recurrence with $T(1)$ being a constant, you find that $T(n) = O(n^2)$. (Unrolling works well for finding this solution.)

In this analysis, we worry primarily about the multiplications and treat other arithmetic as being fast, aka $O(n)$. To understand why this is reasonable, first notice that adding two numbers requires time proportional to the number of digits in them. Multiplying by 2^m is also fast, because it just requires left-shifting the bits in the numbers. If you haven't done much with binary numbers yet (and perhaps don't know what it means to shift them), then it's just like multiplying by 10^m when working with (normal) base-10 numbers. To multiply by 10^m, you just have to add m zeros to the end of the number.

However, we can rewrite our algebra for computing B as follows

$$B = (x_1 + x_0)(y_1 + y_0) - A - C$$

So we can compute B with only one multiplication. So, if we use this formula for B, the running time of multiplication has the recurrence:

$$P(n) = 3P(n/2) + O(n)$$

It's not obvious that we've gained anything substantial, but we have. If we build a recursion tree for P, we discover that the kth level of the tree contains 3^k problems, each involving $n\frac{1}{2^k}$ work. The tree height is $\log_2(n)$.

So each level requires $n(\frac{3}{2})^k$ work. If you add up this up for all levels, you get a summation that is dominated by the term for the bottom level of the tree: $n(\frac{3}{2})^{\log_2 n}$ work. If you mess with this expression a bit, using facts about logarithms, you find that it's $O(n^{\log_2 3})$ which is approximately $O(n^{1.585})$.

So this trick, due to Anatolii Karatsuba, has improved our algorithm's speed from $O(n^2)$ to $O(n^{1.585})$ with essentially no change in the constants. If $n = 2^10 = 1024$, then the naive algorithm requires $(2^10)^2 = 1,048,576$ multiplications, whereas Katatsuba's method requires $3^{10} = 59,049$ multiplications. So this is a noticable improvement and the difference will widen as n increases.

There are actually other integer multiplication algorithms with even faster running times, e.g. Schoöhage-Strassen's method takes $O(n \log n \log\log n)$ time. But these methods are more involved.