CS 228: Introduction to Data Structures
Lecture 9
Monday, February 2, 2015

Array Equality, Continued

The next page shows the pseudocode that we saw at the end of last lecture, annotated with the worst case number of times each step is executed.

	#Times performed	
`i = 0`	1	
`while i < n`	n + 1	
`    found = false`	n	
`    j = 0`	n	
`    while j < n`	n × (n + 1)	*at most*
`        if a[i] == b[j]`	n × n	*at most*
`            found = true`	n × 1	
`            break`	n × 1	
`        ++j`	n × n	*at most*
`    if !found`	n	
`        return false`	0	
`    ++i`	n	
`return true`	1	
Total	$3n^2 + 8n + 3$	*at most*

The total number of steps is $T(n) = 3n^2 + 8n + 3$. Intuitively, when n is large, 8n + 3 is negligible compared to n^2, so T(n) is proportional to n^2. Formally:

Claim. $T(n) = O(n^2)$.

Proof: Choose c = 14 (= 3 + 8 + 3) and observe that as long as $n \geq 1$,

$$3n^2 + 8n + 3 \leq 3n^2 + 8n^2 + 3n^2 = 14n^2.$$

More generally, ***every*** quadratic function is $O(n^2)$. $O(n^2)$ is the class of all functions that asymptotically grow no faster than quadratic functions.

Notice that $3n + 3$ —the running time of sequential search — is also $O(n^2)$. This is because big-O gives ***upper bounds***. But we are most interested in describing an algorithm using the ***smallest*** (slowest growing) big-O class that we can identify. So, while we can say that $3n + 3$ is $O(n^2)$, this is not as precise as saying that it is $O(n)$.

Algorithm Analysis in Practice

Algorithm analysis is seldom done at the level of detail at which we have done it thus far. Indeed, it is asymptotically (that is, in terms of big-O) just as accurate to focus on the parts of the algorithm that depend most heavily on input size. For our algorithm to test for array equality would be analyzed like this:

- The inner loop executes at most n times and its body does not depend on n (so it does not change the big-

O bound). Thus, the inner loop is O(n).

- Everything else in the main loop body does not depend on n, so the main loop body is O(n) .

- The main loop executes at most n times, so the total for the loop is $O(n^2)$.

- Adding the extra constant-time steps outside the main loop does not add to the big-O complexity.

Binary Search

Here is another solution to the search problem. It requires the array A to be sorted in nondecreasing order.

```
BinarySearch(A, v)  // A must be sorted
    n = A.size
    left = 0
    right = n - 1
    while left ≤ right
        mid = (left + right)/2
        if A[mid] == v
            return true
        if v < A[mid]
            right = mid - 1
        else
            left = mid + 1
    return false
```

The body of the loop only takes O(1) time, and all steps outside the loop take O(1) time. Thus, the running time is

$$\#\text{iterations} \times O(1) + O(1) = O(\#\text{iterations}).$$

That is, the constant factor and the constant additive term are “absorbed” in the big-O. Now, lets bound the number of iterations.

Each iteration divides the search range `[left..right]` by 2. The loop terminates when we find what we are

looking for or there are no more elements in the search range. Thus, the number of iterations, call it k, is bounded by the number of times we can divide n by 2 before we get 1. That is, $n/2^k = 1$ or, equivalently, $2^k = n$ This number k is known as the ***logarithm base 2*** of n, and is denoted by "$\log_2 n$" or "lg n".

Logarithms. For any real number $b > 1$, $\log_b n$ is the number x such that $b^x = n$; b is called the ***base*** of the logarithm. In Computer Science, the most common base for logarithms is b = 2. Logarithms appear in the time complexity of an algorithm when some aspect of the problem size repeatedly decreases by a constant factor. For instance, in binary search, the search range is reduced by a factor of 2 at each iteration. Something similar happens in the next piece of code.

```
void foo(int n)
{
  while (n > 1) { n = n / 3;}
}
```

Suppose n = 243. Since $243 = 3 * 3 * 3 * 3 * 3 = 3^5$, it will take 5 iterations to get down to 1. For arbitrary n, the number of iterations equals the number x of times we can

divide n by half so that we get 1. Thus, x is the exponent for which $n(1/3)^x = 1$. Equivalently, x is the number such that $3^x = n$; i.e., $x = O(\log_3 n)$. Thus, `foo(n)` takes $O(\log_3 n)$ time. In general, x will not be a whole number but is never more than 1 away from the number of iterations.

Logarithmic functions grow slowly. E.g., $\log_2 1{,}000{,}000$ is less than 20. In terms of big-O, we don't usually bother with the base, because all log functions are proportional to each other. E.g., for any base b,

$$\log_2 n = (\log_b n)/(\log_b 2).$$

I.e, you can get the log base 2 from the log to any other base by multiplying by the ***constant*** $1/(\log_b 2)$.

Rule of thumb: If the problem size decreases by a ***constant factor*** at each iteration, then the number of iterations is a logarithmic function. For example, the loop

```
while (n > 1) { n = n * .99; }
```

Is done $\log_{1/.99} n$ times. This is O(lg n), although the constant is not that small (≈ 69).

Subset Sum

Consider the following problem.

SUBSET SUM

Input: An array A with n elements and a number K.

Question: Does A contain a subset of elements that adds up to exactly K?

The obvious algorithm for this problem is ***exhaustive enumeration***.

Enumerate all subsets of the elements of A. For each subset, see if its elements add up to K.

There are 2^n subsets to enumerate. The reason is that, for each `i`, we can either include or exclude `A[i]`, so the number of choices is $2 \times 2 \times 2 \times \cdots \times 2$ (n times). Therefore, the algorithm takes $O(n \cdot 2^n)$ time. While this is slow, it might essentially be the best we can do. Subset Sum is ***NP-complete***, which means that it is likely ***not*** to have an efficient algorithm. If you could find such an algorithm or show that none exists, you could win a million

dollars, as this would solve one of the Clay Mathematics Institute's seven Millennium Prize problems[1].

Practical Implications of Time Complexity

A big-O bound on worst-case time complexity is only an estimate of the actual running time of an algorithm; still, this bound has a practical impact. Suppose, optimistically, that each step takes one CPU clock cycle, and that our CPU runs at 1 GHz. The table on the next page some execution times for various input sizes. Notice the big difference between a O(n log n) algorithm and a $O(n^2)$ algorithm, and the huge difference between an $O(n^2)$ algorithm and a $O(2^n)$ algorithm. Even if you get a machine that is a hundred times faster, it won't make much of a dent on the gap between a quadratic algorithm and an exponential one.

[1] http://www.claymath.org/millennium-problems/millennium-prize-problems. The specific question here is the "P vs NP Problem" — see http://www.claymath.org/millenium-problems/p-vs-np-problem.

Clock rate:	1,000,000,000
seconds/day	86400
seconds/year	31536000

size	log n	n	n log n	n^2	n^3	2^n
10	3 ns	0.00001 ms	0.00003 ms	0.0001 ms	0.0010 ms	0.00102 ms
20	4 ns	0.00002 ms	0.00009 ms	0.0004 ms	0.0080 ms	1.04858 ms
30	5 ns	0.00003 ms	0.00015 ms	0.0009 ms	0.0270 ms	1.0737 s
50	6 ns	0.00005 ms	0.00028 ms	0.0025 ms	0.1250 ms	13.0312 days
100	7 ns	0.00010 ms	0.00066 ms	0.0100 ms	1.0000 ms	4.0E+13 years
1000	10 ns	0.00100 ms	0.00997 ms	1.0000 ms	1000.0000 ms	3.4E+284 years
10000	13 ns	0.01000 ms	0.13288 ms	0.1000 s	1000.0000 s	#NUM!
100000	17 ns	0.10000 ms	1.66096 ms	10.0000 s	11.5741 days	#NUM!
1000000	20 ns	1.00000 ms	19.93157 ms	1000.0000 s	31.7098 years	#NUM!