

CS231 Fall 2013 Lecture 18

Stephanie R Taylor

October 16, 2013

1 Administrative Topics

- Return quizzes.
- There will be a quiz this week. You want to be rewarded for practicing recursion, right?

2 Refining our terminology

We have been using Big-Oh notation to describe the time-complexity of algorithms as n grows very large. Technically, Big-Oh notation gives an *upper bound* on the growth rate of the runtime.

Consider the code snippet

```
int x = 0;
for (int i = 0; i < n; i++)
    x++;
```

This code runs in time proportional to n . So we say that it is $O(n)$. But, technically, it is also $O(n^2)$ because the run-time of the code is no greater than a constant times n^2 .

Likewise, there is terminology for a lower bound on the growth rate. The term for that is Ω (big-Omega). The above code is $\Omega(1)$ because the time to run the program will grow at least as fast as a constant. It is also $\Omega(\log(n))$

because its run-time will grow at least as fast as $\log(n)$. Likewise, we can say it is $\Omega(n)$ because its run-time will grow at least as fast as n .

Notice that the same code is both $\Omega(n)$ and $O(n)$ (i.e. the upper and lower bounds on the runtime are the same within a constant factor). When the upper and lower bounds are the same within a constant factor, we indicate this by using Θ (big-Theta) notation. So, the most precise way to describe the complexity of an algorithm is to use Θ notation.

In all of the examples we have used so far, we could have used the Θ notation and had the same answers. From now on, let's use Θ notation.

For more reading, check out Chapter 3 of <http://people.cs.vt.edu/~shaffer/Book/index.php> and our good friend, Wikipedia <http://en.wikipedia.org/wiki/Big-oh>.

3 Recursion and time complexity

Here are the most common kinds of recursive methods and their time complexities.

- If your recursive method is a straightforward rewriting of a non-recursive method, then they should both have the same time complexity, so you can analyze either one for that complexity.
- If your method is of the form

```
foo(n) {  
    if (n == 0)  
        do a const number of steps  
    else {  
        do a const number of steps  
        foo(n-1)  
    }  
}
```

then the time complexity is $\Theta(n)$.

- If your method is of the form

```
foo(n) {
```

```

    if (n == 0)
        do f(n) number of steps
    else {
        do g(n) number of steps
        foo(n-1)
    }
}

```

then the time complexity is $\Theta(f(n) + n * g(n))$.

- If your method is of the form

```

foo(n) {
    if (n == 0)
        do a const number of steps
    else {
        do a const number of steps
        foo(n-1)
        foo(n-1)
    }
}

```

then the time complexity is $\Theta(2^n)$.

- If your method is of the form

```

foo(n) {
    if (n == 0)
        do a const number of steps
    else {
        do a const number of steps
        foo(n/2)
    }
}

```

then the time complexity is $\Theta(\log_2(n))$. (Example: binary search)

- If your method is of the form

```

foo(n) {
    if (n == 0)
        do a const number of steps
    else {
        do a const number of steps
        foo(n/2)
        foo(n/2)
    }
}

```

then the time complexity is $\Theta(n)$.

- If your method is of the form

```
foo(n) {  
    if (n == 0)  
        do a const number of steps  
    else {  
        do a multiple of n steps  
        foo(n/2)  
        foo(n/2)  
    }  
}
```

then the time complexity is $\Theta(n * \log(n))$. (Example: merge sort)

4 Binary search

Suppose you have a sorted list of integers. How could you find a particular integer in that list? What about a binary search? The way a binary search works is you look at the middle element and determine whether the item would be in the first half or the second half of the list. Then search the appropriate half, recursively.

Let's write the code:

```
private static boolean binarySearch( int[] a, int start, int stop, int goal ) {  
    if (start == stop) {  
        return a[start] == goal;  
    }  
  
    int half_dist = (stop-start)/2;  
    if (a[start+half_dist] >= goal)  
        return binarySearch( a, start, start+half_dist, goal );  
    else  
        return binarySearch( a, start+half_dist+1, stop, goal );  
}  
  
public static boolean binarySearch( int[] a, int goal ) {  
    if (a.length == 0)  
        return false;  
    return binarySearch( a, 0, a.length-1, goal );  
}
```

}

Note that if we use `start == stop` to detect an array of length 1, then we have to use another way to detect an array of length 0. In the above code, I chose to put that test for an empty array in the public method.