CMPSCI 187: Programming With Data Structures

Lecture 5: Analysis of Algorithms Overview
16 September 2011

Analysis of Algorithms Overview

- What is Analysis of Algorithms?
- L&C's Dishwashing Example
- Being Usefully Vague About Functions
- Important Classes of Growth Functions
- Determining Time Complexity From Code

What is Analysis of Algorithms?

- We want to talk about the resources, usually time, used by an algorithm, as a function of the input size.

- The time may be different for different inputs of the same size -- we take the **worst-case** time because we want to make a guarantee to the user.

- The **time complexity** of an algorithm is a function with the number of input bits as its input, and the worst-case running time (in seconds, say, or in clock cycles) as the output.

- But such a function is very hard to work with. We need to develop a better mathematical way of talking about such functions, called **asymptotic analysis** or **big-O notation**.

L&C's Dishwashing Example

- Let $f(n)$ be the time (in seconds) that it takes to wash n dishes.
- Individual dishes may be cleaner or dirtier, but say that the worst take 30 seconds.
- In the Good Method, where the washing of later dishes doesn't affect the earlier ones, we have that $f(n)$ is at most 30n seconds. In the worst case where every dish is horribly dirty, we take exactly 30n seconds.
- In the Bad Method, washing the i'th dish soils the first i-1 dishes. For fun, let's say that we can rewash a dish in 10 seconds. Now we have $f(1) = 30$, $f(2) = 30 + (30 + 10) = 70$, $f(3) = 30 + (30 + 10) + (30 + 10 + 10) = 120$, and $f(4) = 30 + 40 + 50 + 60 = 180$. We have that $f(n)$ is the sum for i from 1 to n of (30 + 10(i-1), which evaluates to $30n + 10n(n-1)/2 = 5n^2 + 25n$ seconds.

Being Usefully Vague About Functions

- We went through some effort to get this function 5n^2 + 25n, but the most important thing about this function is that it is **quadratic**, a polynomial in n of degree 2. The first function, 30n, is **linear**, meaning a polynomial of degree 1.

- The growth behavior of a polynomial in n, as n increases, depends primarily on the degree of the polynomial rather than the leading constant or the low-order terms. On page 16 of L&C is a table showing values of the functions 15n^2, 45n, and 15n^2 + 45n. Their point is that the first gets so much bigger than the second that the first and third are practically identical.

- If we graphed 0.0001n^2 against 10000n, the linear function would be larger for a long time, but the quadratic one would eventually catch up (in this case at n = 10^8. *Any* quadratic with positive leading coefficient will eventually beat *any* linear. So the linear term in a quadratic eventually does not matter.

Important Classes of Growth Functions

- There are a number of classes of growth functions that often occur in the analysis of algorithms.

- The first and perhaps more important is the class of **constant** functions, also called **O(1)** functions. These don't always have the same value, but they are *bounded above* by some constant. For example, washing a dish in our example took O(1) time because it was never more than 30 seconds. It is often much easier to see that a process takes O(1) time than to find the actual constant. We need to know that the time is **independent** of the input size.

- The other functions that L&C list on page 17 are **logarithmic**, **linear**, **n log n**, **quadratic**, **cubic**, and **exponential**. They have some graphs.

- In general "O(f)" means "grows proportionally with f(n)".

More on Classes of Growth Functions

- Many important behaviors of a function depend only on the growth class.

- Look at how doubling the input size affects the running time in each case. For a constant function, there is no change. For a linear function, the running time doubles. For a quadratic function, it multiplies by four. For an exponential function, it goes way up -- for 2^n it *squares*.

- Similarly, L&C look at how a fixed speedup affects the maximum size you can handle in a given time. A speedup of 10 means that a linear-time algorithm can handle 10 times as much input. A quadratic-time algorithm can handle about 3 times as much. A 2^n time algorithm can handle *three or four more inputs* than it could before -- the speedup matters hardly at all.

Determining Time Complexity From Code

- It's generally not too hard to tell when a piece of code takes O(1) or constant time. You need to be sure that the behavior does not depend on the input size at all.

- If we have a loop like `for (int i=0; i < n; i++) whatever()`, where n is the input size, then we will execute `whatever()` at most n times. We will also have some other steps to control the loop, but only a constant number for each time through.

- Arithmetic with big-O is fun: We have (O(n) times O(1)) + (O(n) times O(1)), which is O(n) + O(n) = O(n). (We'll play with this a bit on HW#1.)

Determining Complexity From Code

- Another code example with nested loops: if again the method call `whatever()` takes O(1) time, then the j-loop takes O(n) and the total loop takes O(n^2). You might think that we get an advantage from not always taking n times through in the inner loop, but it's only about half the time we would take from saying `j < n` instead.

```
for  (int i = 0, i < n, i++)
  for (int j = 0, j < i, j++)
     whatever( );
```

Running Time of Searches

- Suppose we have n elements in an array and need to find a particular value if it is there.

- If there are n values and each has its own address, we just check that address. This takes O(1) time because we just need a few indirect addressing steps.

- If the list is unsorted, we do a **linear search** from the beginning until or unless we find it. In the worst case we spend O(1) time on each location for O(n) total.

- If the list is sorted, and we do a **binary search**, we take O(log n) time, enormously better than O(1).