CMPSCI 187: Programming With Data Structures

Lecture 5: Analysis of Algorithms Overview
14 September 2012

Analysis of Algorithms Overview

- What is Analysis of Algorithms?
- Example: Summing Consecutive Integers
- Example: Finding a Number in a Phone Book
- 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**.

Example: Summing Consecutive Integers

- Suppose that we are asked to write a method that takes a parameter n (which must be a non-negative integer) and returns the sum of the numbers from 1 through n. (CMPSCI 250 preview: What is the correct output if n = 0?)

- The first method below will take a number of steps **proportional to** the input n. The actual function giving the number of steps will be something of the form an+b, a **linear function**. But the second method will take the *same number* of operations no matter what value n has.

```
public int firstMethod (int n){
   int sum = 0;
   for (int i = 1; i <= n; i++)
      sum += i;
   return sum;}

public int secondMethod (int n){
   return (n * (n+1))/2;}
```

Example: Finding Numbers in a Phone Book

- Before the internet, we used to have large paper volumes called **phone books** that contained the telephone number for every subscriber in a certain area, in alphabetical order by the subscriber’s name. (You can still find these books around today, and you may even have used one.)

- If you want to find a particular person’s number, one method to do so would be to look at the first name in the book, then the second, then the third, and so on until you find the target name. This **linear search** is a correct but not efficient algorithm if the names are in order, because you are not making any use of the order. The time is again a linear function of n, the book’s size.

- A better method is **binary search**, where you keep a section of the book in mind that contains the name, and repeatedly try the middle name of this section so that you know whether the target is in the first or second half.

- As we’ll see later, binary search takes time proportional to the **log** of n.

Being Usefully Vague About Functions

- DJW look at the function $N^4 + 100N^2 + 500$. On page 44 they have a table giving the values of each of the three pieces for various N. For $N = 1$ the "500" gives most of the total of 601. For $N = 10$ the first two pieces each give half the total of 20,500. For $N = 100$ the first piece gives nearly all of the total of 101,000,500.

- 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.

- 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, we might have dishes to wash that take varying time, but never more than 30 seconds. This would be O(1) time. 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 DJW list on page 45 are **logarithmic**, **n log n**, **quadratic**, **cubic**, and **exponential**. They have a table of values.

- In general "O(f)" means "grows proportionally with f(n)", or more precisely "is bounded above by something that 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, we can 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).

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

- Our phone book example is one case of a general problem: 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).