IE170: Algorithms in Systems Engineering: Lecture 3

Jeff Linderoth

Department of Industrial and Systems Engineering
Lehigh University

January 19, 2007

Taking Stock

Last Time

- Lots of funky math
  - Playing with summations: Formulae and Bounds
  - Sets
- A brief introduction to our friend Θ

This Time

- Questions on Homework?
- Θ, O and Ω
- Recursion
- Analyzing Recurrences

Comparing Algorithms

- Consider algorithm A with running time given by f and algorithm B with running time given by g.
- We are interested in knowing

$$L = \lim_{n \to \infty} \frac{f(n)}{g(n)}$$

- What are the four possibilities?
  - $L = 0$: g grows faster than f
  - $L = \infty$: f grows faster than g
  - $L = c$: f and g grow at the same rate.
  - The limit doesn't exist.

Θ Notation

- We now define the set

$$\Theta(g) = \{f : \exists\, c_1, c_2, n_0 > 0 \text{ such that } c_1 g(n) \leq f(n) \leq c_2 g(n)\ \forall n \geq n_0\}$$

- If $f \in \Theta(g)$, then we say that f and g grow at the same rate or that they are of the same order.
- Note that

$$f \in \Theta(g) \Leftrightarrow g \in \Theta(f)$$

- We also know that if

$$\lim_{n \to \infty} \frac{f(n)}{g(n)} = c$$

for some constant c, then $f \in \Theta(g)$.

Big-O Notation

$O(g) = \{f \mid \exists$ constants $c, n_0 > 0$ s.t. $f(n) \leq cg(n)\ \forall n \geq n_0\}$

- If $f \in O(g)$, then we say that "f is big-O of" g or that g grows at least as fast as f
- If we say $2n^2 + 3n + 1 = 2n^2 + O(n)$ this means that $2n^2 + 3n + 1 = 2n^2 + f(n)$ for some $f \in O(n)$ (e.g. $f(n) = 3n + 1$).

Big-Ω Notation

$\Omega(g) = \{f \mid \exists$ constants $c, n_0 > 0$ s.t. $0 \leq cg(n) \leq f(n)\ \forall n \geq n_0\}$

- $f \in \Omega(g)$ means that g is an asymptotic lower bound on f
- f "grows faster than" g

Note

- $f \in \Theta(g) \Leftrightarrow f \in O(g)$ and $f \in \Omega(g)$.
- $f \in \Omega(g) \Leftrightarrow g \in O(f)$.

Strict Asymptotic Bounds. "Little oh"

$$f \in o(g) \Leftrightarrow \lim_{n \to \infty} \frac{f(n)}{g(n)} = 0$$

$$f \in \omega(g) \Leftrightarrow g \in o(f) \Leftrightarrow \lim_{n \to \infty} \frac{f(n)}{g(n)} = \infty$$

Note

- $f \in o(g) \Rightarrow f \in O(g) \setminus \Theta(g)$.
- $f \in \omega(g) \Rightarrow f \in O(g) \setminus \Theta(g)$.

Comparing Functions

- The notation we have just defined gives us a way of ordering functions.
- This gives us a method for comparing algorithms based on their running times!

The Upshot!

- $f \in O(g)$ is like "$f \leq g$,"
- $f \in \Omega(g)$ is like "$f \geq g$,"
- $f \in o(g)$ is like "$f < g$,"
- $f \in \omega(g)$ is like "$f > g$," and
- $f \in \Theta(g)$ is like "$f = g$."

Examples

Some Functions in $O(n^2)$

- n^2
- $n^2 + n$
- $n^2 + 1000n$
- $1000n^2 + 1000n$
- n
- $n^{1.9999}$
- $n^2/\lg\lg n$

Some Functions in $\Omega(n^2)$

- n^2
- $n^2 + n$
- $n^2 + 1000n$
- $1000n^2 + 1000n$
- n^3
- $n^{2.0001}$
- $n^2/\lg\lg n$

A Question

Which of these are $o(n^2)?, \omega(n^2)$?

Commonly Occurring Functions

Polynomials

- $f(n) = \sum_{i=0}^{k} a_i n^i$ is a polynomial of degree k
- Polynomials f of degree k are in $\Theta(n^k)$.

Exponentials

- A function in which n appears as an exponent on a constant is an exponential function, i.e., 2^n.
- For all positive constants a and b, $\lim_{n\to\infty} \frac{n^a}{b^n} = 0$.
- This means that exponential functions always grow faster than polynomials

More Functions

Logarithms

- $x = \log_n(a) \Leftrightarrow b^x = a$
- Logarithms of different bases differ only by a constant multiple, so they all grow at the same rate.
- A polylogarithmic function is a function in $O(\lg^k)$.
- Polylogarithmic functions always grow more slowly than polynomials.

Factorials

- $n! = n(n-1)(n-2)\cdots(1)$
- $n! = o(n^n)$, $n! = \omega(2^n)$
- $\lg(n!) = \Theta(n \lg n)$

Logs

- $a^n a^m = a^{n+m}$
- We use the notation
  - $\lg n = \log_2 n$
  - $\ln n = \log_e n$
  - $\lg^k n = (\lg n)^k$
- Changing the base of a logarithm changes its value by a constant factor

Log Rules

- $a = b^{\log_b a}$
- $\lg\left(\prod_{k=1}^{n} a_k\right) = \sum_{k=1}^{n} \lg a_k$
- $\log_b a^n = n \log_b a$
- $\log_b a = (\log_c a)/(\log_c b)$
- $\log_b a = 1/(\log_a b)$
- $a^{\log_b n} = n^{\log_b a}$

Problem Difficulty

- The difficulty of a problem can be judged by the (worst-case) running time of the best-known algorithm.
- Problems for which there is an algorithm with polynomial running time (or better) are called polynomially solvable.
- Generally, these problems are considered to be easy.
  - Formally, they are in the complexity class $\mathcal{P}$
- There are many interesting problems for which it is not known if there is a polynomial-time algorithm.
- These problems are generally considered difficult.
  - This is known as the complexity class $\mathcal{NP}$.

A++++++++++++++++++++++++

- You will get a very good grade in this class if you prove $\mathcal{P} = \mathcal{NP}$
- It is open of the great open questions in mathematics: Are these truly difficult problems, or have we not yet discovered the right algorithm?
- If you answer this question, you can win a million dollars: `http://www.claymath.org/millennium/P_vs_NP/`
- Most important, you can get the jokes from the Simpsons: `www.mathsci.appstate.edu/~sjg/simpsonsmath/`
- In this course, we will stick mostly to the easy problems, for which a polynomial time algorithm is known.

Analyzing Recurrences

Deep Thoughts

To understand recursion, we must first understand recursion

- General methods for analyzing recurrences
  - Substitution
  - Master Theorem
  - Generating Functions
- Note that when we analyze a recurrence, we may not get or need an exact answer, only an asymptotic one
- We may prove the running time is in $O(f)$ or $\Theta(f)$

Good Stuff

- If you are only concerned about the asymptotic behavior of a recurrence, then
  1. You can ignore floors and ceilings: (Asymptotic behavior doesn't care if you round down or up)
  2. We assume that all algorithms run in $\Theta(1)$ (Constant time) for a small enough fixed input size n. This makes the base case of induction easy.

A Few Examples of Recurrences

- This recurrence arises in algorithms that loop through the input to eliminate one item.

$$T(n) = \begin{cases} 1 & n = 1 \\ T(n-1) + n & n > 1 \end{cases}$$

- This recurrence arises in algorithms that halve the input in one step.

$$T(n) = \begin{cases} 1 & n = 1 \\ T(n/2) + 1 & n > 1 \end{cases}$$

Soms More Recurrences

- This recurrence arises in algorithms that halve the input in one step, but have to scan through the data at each step.

$$T(n) = \begin{cases} 1 & n = 1 \\ T(n/2) + n & n > 1 \end{cases}$$

- This recurrence arises in algorithms that quarter the input in one step, but have to scan through the data 4 times at each step.

$$T(n) = \begin{cases} 1 & n = 1 \\ T(n/4) + 4n & n > 1 \end{cases}$$

Solving Recurrences by Substitution

A Simple Two Part Plan

1. Guess an answer
2. Use induction to prove or disprove your guess

- Here let's show that if

$$T(n) = T(\lceil n/2 \rceil) + 1 \Rightarrow T \in O(\lg n)$$

The Master Theorem

- Most recurrences that we will be interested in are of the form

$$T(n) = \begin{cases} 1 & n = 1 \\ aT(n/b) + f(n) & n > 1 \end{cases}$$

- The Master Theorem tells us how to analyze recurrences of this form.
  - If $f \in O(n^{\log_b a - \varepsilon})$, for some constant $\varepsilon > 0$, then $T \in \Theta(n^{\log_b a})$.
  - If $f \in \Theta(n^{\log_b a})$, then $T \in \Theta(n^{\log_b a} \lg n)$.
  - If $f \in \Omega(n^{\log_b a + \varepsilon})$, for some constant $\varepsilon > 0$, and if $af(n/b) \leq cf(n)$ for some constant $c < 1$ and $n > n_0$, then $T \in \Theta(f)$.
- How do we interpret this?

A Few More Examples

- This recurrence arises in algorithms that partition the input in one step, but then make recursive calls on both pieces.

$$T(n) = \begin{cases} 1 & n = 1 \\ 2T(n/2) + 1 & n > 1 \end{cases}$$

- This recurrence arises in algorithms that scan through the data at each step, divide it in half and then make recursive calls on each piece.

$$T(n) = \begin{cases} 1 & n = 1 \\ 2T(n/2) + n & n > 1 \end{cases}$$

- We can analyze these using the Master Theorem.

A Few More Comments on Recursion

- Generally speaking, recursive algorithms should have the following two properties to be guarantee well-defined termination.
  - They should solve an explicit base case.
  - Each recursive call should be made with a smaller input size.
- All recursive algorithms have an associated tree that can be used to diagram the function calls.
- Execution of the program essentially requires traversal of the tree.
- By adding up the number of steps at each node of the tree, we can compute the running time.
- We will revisit trees later in the course.

The Call Stack

- The call stack of a program keeps track of the current sequence of function calls.
- When a new function call is made, data for the current one is saved on the call stack.
- When a function call returns, it returns to the next function on the top of the stack.
- The stack depth is the maximum number of functions on the stack at any one time.
- In a recursive program, the stack depth can be very large.
- This can create memory problems, even for simple recursive programs.
- There is also an overhead associated with each function call.

Next Time

- Homework is due!
- Simple Sorting and Its Analysis
- Three Hours of Fun-Filled Lab

Go Bears!