CSC207.01 2013F, Class 19: Analyzing Algorithms

Overview

- Preliminaries
  - Admin.
  - Questions on HW5.
  - HW6.
- Comparing algorithms.
- Potential problems in computing running time.
- Asymptotic analysis.
- Big-O, formalized.
- Implications of Big-O.
- Doing informal asymptotic analysis.
- Some recurrence relations.
- Experimental analysis.

Admin

- Reading for Wednesday: Linear and Binary Search in Java. (And yes, it's ready.)
- EC Opportunities
  - CS Extras Thursday @ 4:30: Adam, Jordan, and Sean on SysAdmin stuff
  - No Learning from Alumni this week
  - CS Table Friday (Coding the Law)
  - Others?
- Other things
  - Poweshiek CARES March Thursday, Oct. 3. Meet at Drake at 5 p.m.
  - GHS Homecoming Parade Thursday, Oct. 3. If you've never seen a small-town homecoming parade, it's worth it.
- Mr. Stone will be guest lecturing (or at least supervising lab) on Wednesday and Friday.
  - Support each other

HW5

- I'm having trouble with ArrayLists. ArrayList incidents = new ArrayList(); return incidents.toArray();
- Why am I getting this strange message about "incompatible version"
  - You need Java 7
  - If you want, you can recompile yourself; simple-ushahidi-api on github
  - Or grab from our examples folder
  - If you use Java 6, you won't be able to do https urls, ask TY for a URL without https http://burgermap.org

HW6

- Fun and open-ended (Plus the legendary Dutch National Flag)

Comparing algorithms

- There's more than one algorithm to solve any given problem.
- Example: Exponentiation x^n for double x and non-negative integer n
  - for loop
  - recursively double pow(double x, int n) { if (n == 0) return 1; else return x * pow(x, n-1); }
  - recursively, using divide and conquer double pow(double x, int n) { if (n == 0) return 1; else if (n % 2 = 0) { double tmp = pow(x, n/2); return tmp*tmp; } else return x * pow(x, n-1); }
  - Factor n, find x^n for each prime factor, then multiply together
  - John Napier (and other logartihmic folks) Table of e^n and ln_n
- You cannot use the built-in pow method. We're assuming that you're implementing it.
- Which is best?
  - Fastest/Running time efficiency (parameterized by input size)
  - Lines of code
  - Most elegant
  - Memory efficiency (parameterized by input size)
  - Safety from errors
  - Accuracy
- Most of the time, running time is the most important (after correctness)

Potential problems in computing running time

- Strategy one: Count the number of steps
  - For loop exponent: increment i N times, multiply N times, test N times; a few more assignments
    - May be easiest to assume that most operations take the same amount of time.
- Strategy two: Implement them all and run them on some inputs
  - A lot of effort
  - Inputs have a big effect (in the sense that we can see very different running times on the same "size" input with the same algorithm)
  - Running programs is unpredicatable
- For our first pass: SIMPLIFY AND MODEL

Asymptotic analysis

- Look at the shape of the curve that bounds the running time (for the worst case of each input size)
- Goal: A way to compute them and a way to compare them.
- How fast does it grow? linear, quadratic, cubic, expontential, logarithmic, constant time
- Ways to think about these: What usually happens if I double the size of the input?
  - Linear time: Double the input -> Double the time

- Quadratic: Double the input -> Quadruple the time
- Constant: Double the input -> Same time
- Logaraithmic (base 2): Double the input -> Increase by a constant
- Exponential: Square the time

Big-O, formalized

- O(g(n)) is a SET of functions
- f(n) is in O(g(n)) iff
  - Exists n*0 > 0*
  - *Exists d > 0*
  - *|f(n)| <= |d*g(n)| for essentially all n > n*0

Implications of Big-O

- O is no 0.
- if f(n) is in O(g(n)) and g(n) is in O(h(n)), f(n) is in O(h(n))
- if f(n) is in O(g(n)), c*f(n) is also in O(g(n))
- O(c*g(n)) = O(g(n)/c) = O(g(n))
- if f(x) = g(x) + h(x) and g(x) is in O(h(x)), f(x) is in O(h(x)) f(x) = 2000x + (x^2)/3 f(x) <= g(x) + h(x) <= d*h(x) + h(x) <= (d+1)h(x)

Doing informal asymptotic analysis

- Iterative
  - Count steps
  - Count loop iterations
  - Multiply
- Recursive
  - Build recursive definitions of running time Binary search time(n) <= c + time(n/2) time(n) = q*log(n) for some q

Some recurrence relations

Experimental analysis

Commons, 543 Howard Street, 5th Floor, San Francisco, California, 94105, USA.