CS3203 #3

6/2/04

Janak J Parekh

Administrivia

- Textbooks should be in the bookstore
- Office hours established?

Algorithms

- An **algorithm** is a finite set of precise instructions for performing a computation or for solving a problem.
  - Term is a corruption of the name *al-Khowarizmi*, whose book on Hindu numerals is the basis of modern decimal notation.
  - What's a step? Good question! Depends on context. Algebraic operations are a step, for instance.
- Example: given a sequence of integers $a_1, \ldots, a_n$, determine if they're in increasing order
  - Express using **psuedocode**.
  - Roughly Pascal-like.

```
output := TRUE
i := 2
while (i <= n and output = TRUE)
begin
   if a_i < a_{i-1} then output := FALSE
   i := i + 1
end
```

Fundamentals of most algorithms

- Input, output
- Definiteness: the steps of an algorithm must be defined precisely
- Correctness: An algorithm should produce the correct output for each set of input
- Finite
- Effective: It must be possible to perform each step of an algorithm exactly and in a finite amount of time
- Generality: Procedure should be applicable for all problems of the desired form, and not just for a particular set of input values.

Common categories of algorithms

- Code for these are in the book
- Searching algorithms
  - How to find information inside a long list?
    - Linear search: go through one item at a time
  - What if it's ordered?
    - Binary search: start in the middle and divide the search space by half each time.
- Sorting algorithms
  - Bubble sort: perhaps the simplest

    procedure *bubblesort*($a_1, \ldots, a_n$)
    for $i := 1$ **to** $n - 1$
        for $j := 1$ **to** $n - 1$
            if $a_j > a_{j+1}$ **then** interchange a_j and a_{j+1}
    {$a_1, \ldots, a_n$ is in increasing order}
  - Insertion sort
  - There are others, of course

Greedy algorithms

- Common way to solve **optimization problems**, where the goal is to find a solution that either minimizes or maximizes the value of some parameter.
- The idea is to “choose the best choice at each step”, i.e., optimize locally instead of globally, and this works for a surprisingly large number of problems (but not all!)
- Example: greedy change-making algorithm is optimal if you have all coins… but *not* necessarily if you’re missing one type of coin.
- Book uses a proof by contradiction to show that this works
- We’ll see a lot of greedy algorithms in the graph theory portion of the course

Growth of functions

- Fundamental idea: we're not concerned with the precise number of steps an algorithm takes
  - Just buy a PC that's X times as fast
- Rather, we're more concerned with how much work the algorithm does as the size of the input increases.
- **Big-O** notation lets us focus on growth and avoids any constants
- Fundamental algebraic definition: let *f* and *g* be functions from the set of integers or the set of real numbers to the set of real numbers. We say that *f(x)* is *O(g(x))* if there exist constants *C* and *k* such that

$$|f(x)| \le C|g(x)| \text{ whenever } x > k$$

How to use?

- Do a little algebra.
- Example: prove $5x^4-37x^3+13x-4=O(x^4)$
- Choosing C=59 and x=1 fulfills the inequality.
- Note that this is not a “genuine equality”.
- If h(x) has larger (absolute) values than g(x) for sufficiently large x, it also follows that f(x) = O(h(x)).
- Example 2: Show that $7x^2$ is $O(x^3)$
  - But $7x^3$ is NOT $O(x^2)$!
- Other important theorems:
  - If $f(x) = a_n x^n + a_{n-1} x^{n-1} + \dots + a_1 x + a_0$, then f(x) is $O(x^n)$ (similar to problem above; can use triangle inequality to prove)
  - Estimate the sum of the first n integers using Big-Oh.

More Big-O

- Common big-O functions used in estimates
  - 1, log n, n, n log n, n^2, 2^n, n!
- If $f_1(x)$ is $O(g_1(x))$ and $f_2(x)$ is $O(g_2(x))$, $(f_1 + f_2)(x)$ is $O(\max(|g_1(x)|, |g_2(x)|))$.
- If $f_1(x)$ is $O(g(x))$ and $f_2(x)$ is $O(g(x))$, $(f_1 + f_2)(x)$ is $O(g(x))$.
- If $f_1(x)$ is $O(g_1(x))$ and $f_2(x)$ is $O(g_2(x))$, $(f_1f_2)(x)$ is $O(g_1(x)g_2(x))$.
- What's a big-O estimate of $f(x)=(x+1)\log(x^2+1)+3x^2$?

Big-Omega and Big-Theta

- Big-O is *only an upper bound*
- Not always useful
- We say f(x) is Ω(g(x)) if there are positive constants C and k s.t.

$$|f(x)| \geq C|g(x)| \text{ whenever } x > k$$

- We say f(x) is Θ(g(x)) if f(x) is O(g(x)) and f(x) is Ω(g(x)).
- Big-Omega is a lower bound, and Big-Theta provides both a lower and upper bound (latter being "on the order of").
- Example: Show that $7x^2+1$ is $\Theta(x)$.
- If $f(x) = a_n x^n + a_{n-1} x^{n-1} + \ldots + a_1 x + a_0$, then f(x) is $\Theta(x^n)$.

Complexity of algorithms

- Now that we know how to express upper/lower bounds, we'd like to analyze our algorithms and determine their *time* and *space* complexity.
  - Space complexity is not covered in this class.
- Time complexity is described in terms of the number of operations required instead of actual computer time.
- Largely informal discussion at this point – take Analysis of Algorithms if you want a more formalized approach.

Examples

- Determine the time complexity of the linear search algorithm in terms of the number of comparisons.
  - The book uses comparisons as the *basic operation*. Counting is not considered, although we could if we want.
  - Two comparisons per step, plus two more (exit condition, and outside comparison). Therefore, $\Theta(n)$ comparisons.
- Binary search?
  - For simplicity, assume there are $n = 2^k$ elements in the list. At each step, we're reducing k by one (i.e., looking at half the list). Therefore, 2k+2 comparisons, or 2 log n + 2 comparisons, which is $\Theta(\log n)$.
- This kind of complexity is *worst-case analysis*. There's also average-case analysis, which we'll generally avoid as it's much more complicated.

Complexity cont'd.

- Complexity of bubble sort?
  - First time, *n-1* comparisons, then n-2, …, 1 comparison.
  - Sum of this is n(n-1)/2, which is $\Theta(n^2)$.
- Implications?
  - O(1) is constant – fastest possible
  - O(log n) is logarithmic – pretty fast
  - O(n) relative to the size of input
  - O(n log n) doesn't have a name, but is common for "fast" sorts.
  - $O(n^2)$ is polynomial – grows, but not too fast
  - $O(b^n)$ where b > 1 is *exponential*; this is the first category of intractable; grows way too fast to be useful; "NP" class of problems
  - O(n!) is factorial
- Assumes O(n) as a "good" estimate
- Page 150 gives some practical implications in the table (* means more than 10^{100} years)
- Then there are the *unsolvable problems*
  - Halting problem

Integers and division

- Number theory!
- Why?
  - Basis of important algorithms in computer science
- Especially divisibility of numbers
  - Prime number – critical to cryptography
  - Modular arithmetic (division and getting a remainder)
- Divisibility
  - If a and b are integers, $a \neq 0$, a divides b if there is an integer *c* such that b=ac. a is a *factor* of b, and b is a *multiple* of a. $a \mid b$.
  - Note we're primarily interested in integers.
- Some basic proprerties:
  - if $a|b$ and $a|c$, then $a|(b+c)$
    - Proof: use the definition and add them together.
  - if $a|b$, then $a|bc$ for all *c*
  - if $a|b$ and $b|c$, then $a|c$.
  - if $a|b$ and $a|c$, $a|mb+nc$ whenever m and n are integers (just apply the 2nd property to 1)

Primes

- A positive integer *p* greater than 1 is called *prime* if the only positive factors of *p* are 1 and *p*.
  - An integer greater than 1 that is divisible by others is *composite*.
  - Note greater than 1; 1 is neither prime nor composite.
  - First few primes?
- Leads to **fundamental theorem of arithmetic**
  - Every positive integer greater than 1 can be uniquely written as a prime or as the product of two or more primes where the prime factors are written in the order of nondecreasing size.
  - “Prime factorization” of any number
- If n is a composite integer, then n has a prime divisor less than or equal to the square root of n
  - Useful in determining primality of reasonably small numbers, e.g., show than 101 is prime.
- There are infinitely many primes.
  - Use the fundamental theorem of arithmetic, and generate a prime Q such that it’s the product of all known primes, plus 1.

Primes (II)

- Quest to find larger and larger primes. One such set is the set of **Mersenne primes**, which are integers of the form 2^p-1. (Not all are!)
- The ratio of the number of primes not exceeding x and x/lnx approaches 1
  - In other words, the odds that a randomly selected positive integer x is prime are approximately 1/ln x.

Division revisited

- **Division algorithm** – given integer *a* and positive integer *d*, there exist unique integers *q* and *r*, with 0 <= r < d, such that $a = dq + r$.
  - d is the divisor, a is the dividend, q is the quotient and r is the remainder.
  - q = a **div** d, and r = a **mod** d
  - Remainders cannot be negative!

GCD and LCM

- Greatest common divisor is the largest integer d such that d|a and d|b. d = gcd(a,b), where a and b both aren't zero
- a and b are *relatively prime* if their gcd is 1.
- Can use prime factorization to find gcd
  - Take as many factors as possible and multiply them together.
- Least common multiple of positive integers *a* and *b* is the smallest positive integer that is divisible by both; lcm(a,b)
  - Again, can use prime factorization; this time, need each term, but only max of any common ones
- For positive integers a, b, ab = gcd(a,b) * lcm(a,b)

Modular arithmetic

- If a,b are integers and m is a positive integer, we say a is *congruent to b modulo m* if *m* divides a – b. Notation is a ≡ b (mod *m*)
- Another way of saying this is that two numbers have the same remainder.
  - a ≡ b (mod *m*) if a **mod** m and b **mod** m.
- a and b are congruent modulo m if and only if there is an integer k such that a = b+km.
- All integers congruent to an integer *a* mod m lets you create *congruence classes*

More modular arithmetic

- If $a \equiv b \pmod{m}$ and $c \equiv d \pmod{m}$, then $a+c \equiv b+d \pmod{m}$ and $ac \equiv bd \pmod{m}$
- Why do we care?
  - Hashing is a way of rapidly storing and retrieving information
    - $h(k) = k$ **mod** m
    - Congruences suggest a collision, which needs to be dealt with.
  - Generating random numbers
  - Cryptology
    - Caesar's encryption method: $f(p) = (p+3)$ **mod** 26
    - In other words, formalizing a very old mechanism
    - Of course, you can use more interesting functions
    - Inverse generates the decryption function

Representing integers

- We've been assuming base 10 all along. However, computers don't necessarily use that.
  - Computers also use base 2, base 8, and base 16 commonly.
- Let *b* be a positive integer greater than 1. Then if *n* is a positive integer, it can be expressed in the form
  - Base *b* expansion of *n*
  - $n = a_k b^k + a_{k-1} b^{k-1} + \ldots + a_1 b + a_0$ where k is nonnegative, $a_0, \ldots, a_k$ are less than b but also nonnegative, and $a_k \neq 0$.
  - For example, binary expansion (pg 169)

Base conversion

- Divide repeatedly by the base.
- Keep the remainders, but use them *backwards*.
- This is just a variation of the previous base expansion.
- Algorithm written out on page 171.
- Can also use table lookups.
  - Remember that hex uses letters…

Integer algorithms with respect to base

- Suppose the binary expansions of *a* and *b* are3 a = $(a_{n-1}a_{n-2}\ldots a_1a_0)_2$ and b = $(b_{n-1}b_{n-2}\ldots b_1b_0)_2$.
- How to add?
- We add their rightmost bits and carry over as necessary.
  - Algorithm on page 173; O(n).
  - Similar to base 10, just keep track of base 2.
  - Multiplication works similarly.

Computing div and mod

- Just subtract the divisor repeatedly and increase the quotient by 1 as long as you can.
- The remainder and quotient are the answer.
  - If the dividend was negative, just flip the quotient.

Euclidean algorithm

- Very fast way of determining the greatest common divisor
- Repeatedly divide the larger by the smaller, and keep the smaller and remainder.
- Therefore, gcd(150,8) = gcd(8,6) = gcd(6,2) = 2
- Based on the following result: Let $a = bq+r$, where *a, b, q, r*are integers. Then gcd(a,b) = gcd(b, r).
- Algorithm on page 179.

Next time

- Reasoning, induction, recursion