Lecture 11: 5/7/2009

Announcements: Ps6 out: **Due Tuesday**
Midterm back Tues, May 12

Common functions (see 3.4 in book, mostly review I assume): log, exponent, floor, ceiling, mod,

Skip 3.5

3.6: Recursively Defined Functions

A recursive function is defined by referring to itself. We saw this in defining T(n) the minimum number of moves to shift n disks in the Tower of Hanoi problem:

T(n) = 1, n=1
T(n)= 2 T(n-1) +1 for n>1

Domain, co-domain, range? N, N, a subset of N: (2^n-1), n=1, 2, …

Two key properties that make this type of definition work: non-recursive definitions for small *base* values and each time we refer to the function on the right side, the argument is smaller than on the left side.

Book does n!:
f(0)= 1
f(n) = n * f(n-1) n >0

I'll do fibinacci function (Def 3.2 in book):

F(0)=F(1) = 1
F(n) = F(n-1)+ F(n-2) for n >1

F(2)= F(2-1) +F(2-2) = F(1) +F(0) = 1 +1 = 2
F(3)= F(3-1) + F(3-2) = F(2) + F(1)= 2+1 (or expand F(2))

Give tree and better way to compute.

Function **Fibo**(n)

If ((n=0) OR (n=1)) Return(1);

Else, Return(**Fibo**(n-1) + **Fibo**(n-2))

Fibo is a *recursive* function

Comments: should make sure n not negative and declare n to be an integer.
Give tree of recursive calls. Argue number of calls at least 2^(n/2).

Alternative:

A an array of size n.
A[0] ← 1;
A[1] ←1;

For *i* ←- 2 to n DO
    A[i] ← A[i-2] + A[i-1]; (*)

Statement labeled (*) is executed n-2 times and we assume takes at most some constant about of time c (time to look up or assign to an array), so total time is at most 2*c + (n-1) *c = n *c +c

Note that the above is a function of n, call it T(n), known as a *linear* function (if n is twice as large, then time, T(n) is twice as much).

Function Growth rates (3.9) (we skipped 3.7 and 3.8, will return to them later)

We want to be able to compare functions and want to ignore less important terms and constants. Lets us compare how different programs perform (for example, in the example for Fibonacci, the second program with T(n) = n *c will significantly outperform the recursive one, with 2^n calls, even if c is very large eventually, and typically for moderate n).

Notation: Big O and later Θ ("Big Theta") and Ω (big Omega).

We say $x^3 \in O(2^x)$. Kind of like saying x^3 \le 2^x "for large x and if you don't care about constants"

.....
Def (Def 3.4 from Schaum's)
O(g) = {f: \R -> \R such that exists N>0,C>0 s.t.
|f(n)| <= C |g(n)| for all n>=N}
.....

You can actually forget about the | . | and just imagine that f is non-negative valued: replace f by |f| in case it's not.

Example:
$n^x + 100n \in O(n^x)$ YES
$10\ n^2 \in O(n^2)$ YES
$10n^2 + 100n + \log N$ in $O(n^2)$ YES

$n \log n$ in $O(n^2)$ YES
$n^2 \log$ in $O(n \log n)$ NO

As a practical matter it is pretty easy to get the "right" big O class for a function:

1) throw away all constants multiplying terms or added to them (e.g. 10x+5 becomes x) Note that you can't throw away constants that are exponents (e.g. n^2 can't throw away the 2).

2) among terms added together, throw away all but the fastest growing term
(apply to examples above)

How to prove things like this?
Suppose want to show
$10\ n \log(n) + 50n + 1 \in O(n \log n)$

must find a large enough C, N such that

$10\ n \log(n) + 50n + 1 <= C\ n \log n$

for all n >= N.
What C, N would you like? would you like? How about

10 n log(n) + 50n + 1 <= 61 n log n

Check: true if

50n + 1 <= 51 n log n

Now 50n < 50n log n if n\ge 3 So the above is true if

1 <= n log n

But for n >= 3, this is certainly true.
Doing it in the forward direction:

1 <= n log n
10n log n + 50n + 1 <= 10n log n + 50 n log n + n log n <= 61 n log n

when n>=3

n! vs 2^n
n! ∈ O(2^n) NO
2^n ∈ O(n!) Yes

n! = (n/e)^n \sqrt(2\π n) (1 + O(1/n))

ln n! \approx n ln n - n

n	lg n	n	n lg n	n^2	n^3	2^n
10	4	10	40	100	10^3	1024
100	7	100	700	1000	10^6	10^30
1000	10	1000	10^4	10^6	10^9	10^300

.....
Θ(g) = {f: R -> R : exists c,C,N such that
c g(n) <= |f(n)| <= C g(n)
.....

Intuitively: O(f(n)) is all functions of growth rate f(n) or **less**. Θ(f(n)) is all functions of the **same** growth rate as f(n).

Θ(n^2) contains: 7n^2, 3n^2 + 100lg n
doesn't contain n^2 lg n (too big)
n lg^2 n (too small)

Draw a picture of common growth rates

Θ(n!)
Θ(2^n)
Θ(n^3)
Θ(n^2)
Θ(n log n log log n)
Θ(n lg n)
Θ(n)
Θ(sqrt(n)
Θ(log n)
Θ(1)

exercise: where is \sqrt(n)

How to compare
n^0.5 n log n
n^0.5 n^0.5 n^0.5 log n
Bigger

How long does the following code take to run

```
for i=1 to n do
  for j=1 to i do
    s += (i+j)^2 - (i+j)
```

O(n^2)
Θ(n^2)

O(n^3) is true ... but not "tight"