Course overview:

webpage: www.cc.gatech.edu/vigoda/3510
lecture schedule there & will post scan of my notes (usually after class)

Prof. Eric Vigoda
TA: Ioannis Pasareus (Pronounced Yannis)

Office hours: TBD (see class webpage)
Eric: MW 1-2
Ioannis: TuTh afternoon

No laptops during class (unless taking notes)
5 midterms, see webpage for tentative dates + final exam

Grades: HW 5%
Exams 70%
Final 25%
HW: not worth a lot — so no point to cheat
but will help a lot on exams
for harder problems — try study group but
write up on your own
No late homeworks — can email it.

Textbook: Algorithms by Dasgupta, Papadimitriou, & Vazirani
$35 on Amazon (in bookstore too)
not sure price?
free version online but different hw numbers!

On reserve at library
(also 2 other books)
Should know already: $O()$ notation

For functions $f(n) \neq g(n)$,

$$f(n) = O(g(n)) \text{ if there is a constant } c > 0 \text{ where } f(n) \leq c \cdot g(n).$$

Examples:

a) $f(n) = 3n^2 + 10n - 5n^2 + 1.7n^3$

$$f(n) = O(n^3) \text{ and } f(n) = O(n^5)$$

b) $f(n) = 5(\log n)^0 + 7\sqrt{n}$

$$f(n) = O(\sqrt{n})$$

c) $f(n) = n^2 \log n + 1000n^2 + 10^{10}n^{1.5}$

$$f(n) = O(n^2 \log n)$$
Should know: manipulating log's

\[ \log_2 n: \text{base 2 if don't specify base.} \]

\[ 2^{\log_2 n} = n \]

\[ \log_{10} n = O(\log n) \]

\[ \frac{1}{\log_{10} n} = \log_{\log_{10} n} = O(\log n) \]

**Exercise:**

\[ f(n) = n^2, \ g(n) = 2^{4 \log n} \]

a) is \( f(n) = O(g(n)) \)?

b) is \( g(n) = O(f(n)) \)?

(or both?)

Can rewrite \( g(n) \):

\[ g(n) = 2^{4 \log n} = \left(2^{\log n}\right)^4 = n^4 \]

So, \( f(n) = O(g(n)) \)

but \( g(n) \neq O(f(n)) \)
We'll use pseudocode:

Example: computing Fibonacci numbers

\[ 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, \ldots \]

defined by:

\[ F_0 = 0, F_1 = 1, \]
\[ \text{and for } n > 1, \]
\[ F_n = F_{n-1} + F_{n-2} \]

Natural algorithm:

\[
\text{Fib1}(n): \quad \begin{align*}
\text{if } n = 0, & \quad \text{return } 0 \\
\text{if } n = 1, & \quad \text{return } 1 \\
\text{return } (\text{Fib1}(n-1) + \text{Fib1}(n-2))
\end{align*}
\]

What's running time?

Look at running time as a function of input size \( n \).

Other examples: \( n = \# \text{ of numbers to sort} \),
\( n = \# \text{ of vertices in input graph} \),
\( n = \# \text{ of bits in input numbers to multiply} \).
Look at time in $O(1)$
So ignoring constant factors
Machine independent analysis

Model of computation:

- $O(1)$ time to add, subtract, multiply, divide
- Any basic arithmetic operation.
- Unlimited memory
- $O(1)$ time to read/write unit of memory.

Analyzing Fib1:

Let $T(n) = \#$ of steps for computing $n^{th}$ Fibonacci #

- $T(0) = O(1)$
- $T(1) = O(1)$

For $n > 1$, $T(n) = T(n-1) + T(n-2) + O(1)$

But $T(n) \geq F_n$
\[ F_n = \frac{\phi^n}{\sqrt{5}} \text{ where } \phi = \frac{1 + \sqrt{5}}{2} \approx 1.618 \text{ is the } \]

"golden ratio." 

exponential-time algorithm.

Why is Fib1 so slow?

\[
\begin{array}{c}
\text{F}_n \\
\text{F}_{n-1} \quad \text{F}_{n-2} \\
\text{F}_{n-2} \quad \text{F}_{n-3} \quad \text{F}_{n-3} \quad \text{F}_{n-4} \\
\text{F}_{n-3} \quad \text{F}_{n-4} \\
\end{array}
\]

Recomputing many times the answer 

to small subproblems
Better approach:
only compute the answer to each subproblem once. Start with smallest \( \rightarrow \) largest (bottom-up approach)

\[
\text{Fib2}(n):
\]
if \( n = 0 \), return (0)
if \( n = 1 \), return (1)
create an array \( F[0..n] \)
\( F[0] = 0, F[1] = 1 \)
for \( i = 2 \rightarrow n 
\]
\( F[i] = F[i-1] + F[i-2] \)
return \( F[n] \)

\[
\text{Running time:}
\]
\( O(1) \) time per \( i \)
\( \Rightarrow O(n) \) total time.