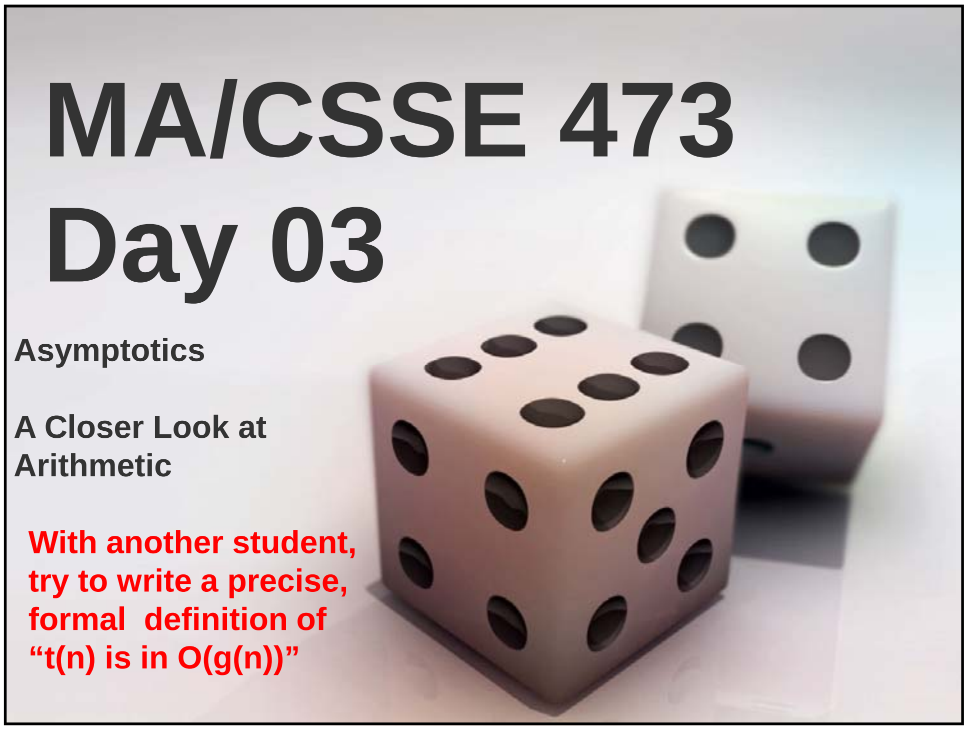
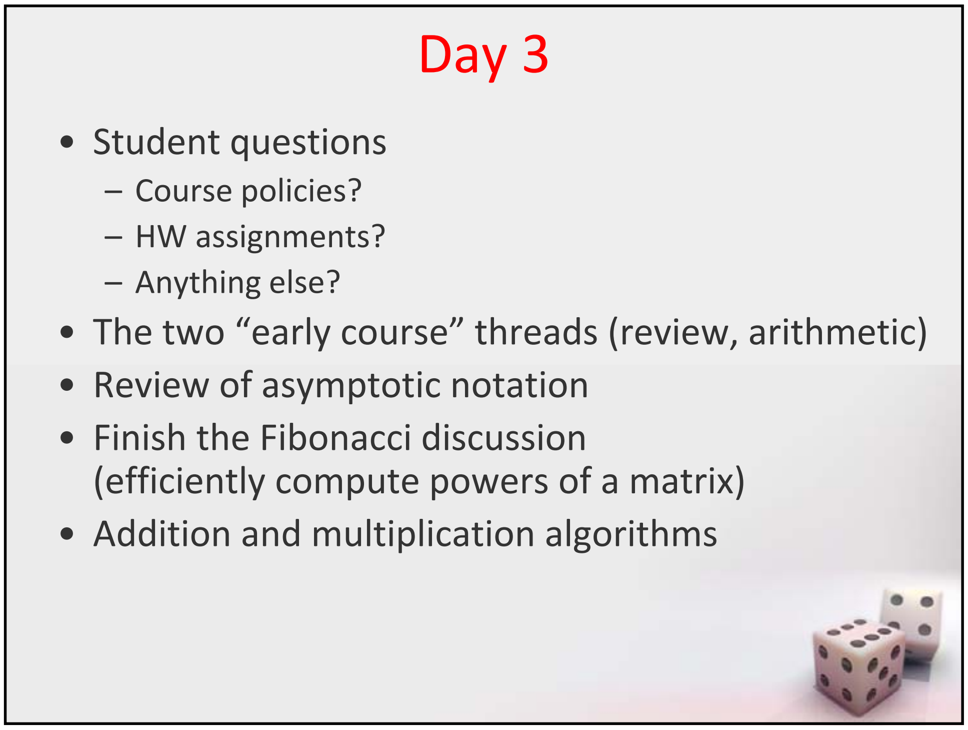


MA/CSSE 473 Day 03

Asymptotics

A Closer Look at Arithmetic

With another student, try to write a precise, formal definition of "t(n) is in O(g(n))"



Day 3

- Student questions
  - Course policies?
  - HW assignments?
  - Anything else?
- The two "early course" threads (review, arithmetic)
- Review of asymptotic notation
- Finish the Fibonacci discussion (efficiently compute powers of a matrix)
- Addition and multiplication algorithms

Two threads in lectures

- Each day at the beginning of the course
- Some review
- Continue with discussion of efficiency of Fibonacci and arithmetic.

Review thread for today:
Asymptotics (O, ϴ, Ω)
Mostly a recap of my 230 lecture on same topic.
I

Rapid-fire Review: Definitions of O, ϴ, Ω

- I will re-use many of my slides from CSSE 230
  - Some of the pictures are from the Weiss book.
- And some pictures are Levitin's.
- A very similar presentation appears in Levitin, section 2.2
- Since this is review, we will move much faster than in 230

Asymptotic Analysis

- We usually only care what happens when N (the size of a problem's input) gets to be large
- Is the runtime linear? quadratic? exponential? etc.

Asymptotic order of growth

Informal definitions

A way of comparing functions that ignores constant factors and small input sizes

- O(g(n)): the class of functions t(n) that grow no faster than some constant times g(n)
- Θ(g(n)): the class of functions t(n) that grow at the same rate as g(n)
- Ω(g(n)): the class of functions t(n) that grow at least as fast as some constant times g(n)

Formal Definition

- We will write a precise formal definition of "$t(n) \in O(g(n))$"
  - This is one thing that students in 473 should soon be able to write from memory…
  - … and also understand it, of course!

Big-oh (a.k.a. Big O)

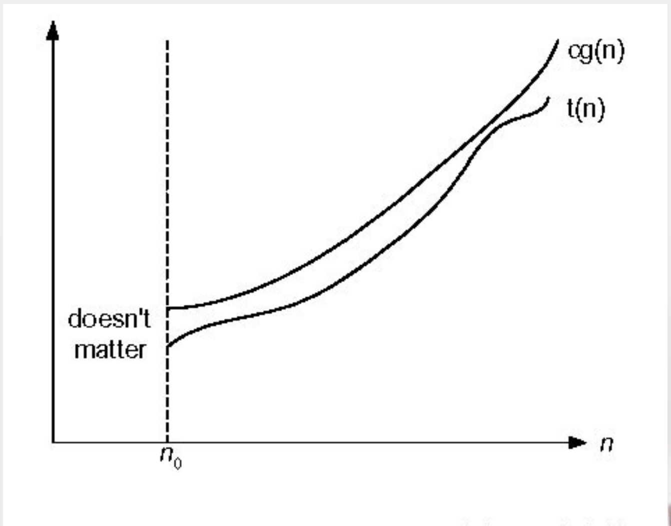


Figure 2.1 Big-oh notation: $t(n) \in O(g(n))$

Prove a Big O Property

- For any function g(n), O(g(n)) is a set of functions
- We say that **t(n) $\in$O(g(n))** iff there exist two non-negative constants **c** and **n_0** such that
  for all $n \geq n_0$, $t(n) \leq c\, g(n)$
- **Rewrite using $\forall$ and $\exists$ notation**
- **Use the formal definition to prove:**
  If $f(n) \in O(g(n))$ and $t(n) \in O(g(n))$,
  then $f(n)+t(n) \in O(g(n))$

Answer (Summer Only)

- Recall: **t(n) $\in$O(g(n))** iff there exist two positive constants **c** and **n_0** such that
  for all $n \geq n_0$, $t(n) \leq c\, g(n)$
- If $f(n) \in O(g(n))$ and $t(n) \in O(g(n))$,
  then $f(n)+t(n) \in O(g(n))$
- **Proof** By definition, there are constants c_1, c_2, n_1, n_2, such that for all $n \geq n_1$, $f(n) \leq c_1\, g(n)$, and for all $n \geq n_2$,
  $t(n) \leq c_2\, g(n)$. Let $n_0 = \max(n_1, n2)$, and let $c = c1 + c2$.
  Then for any $n \geq n_1$, $f(n)+t(n) \leq c_1\, g(n) + c_2\, g(n) = c\, g(n)$.

Big-omega

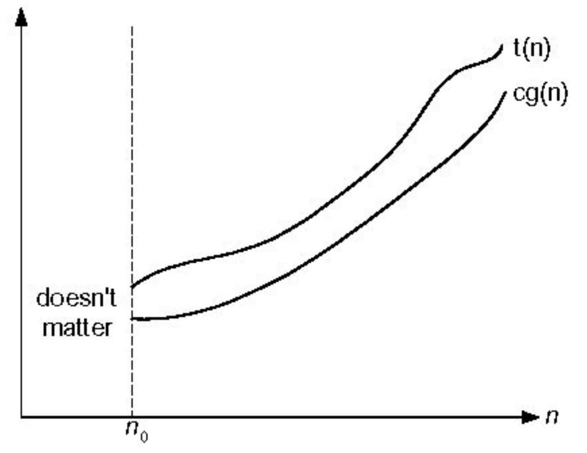


Fig. 2.2 Big-omega notation: $t(n) \in \Omega(g(n))$

Big-theta

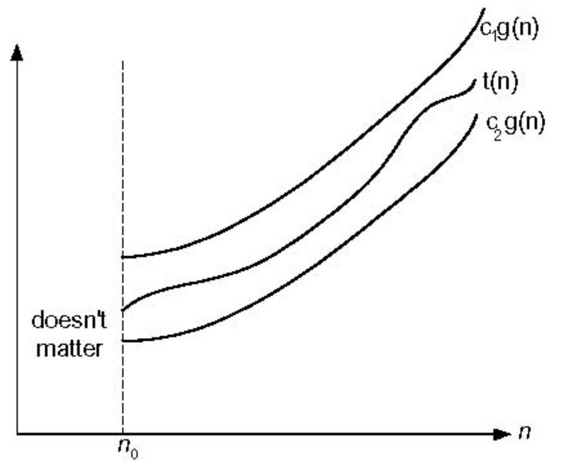


Figure 2.3 Big-theta notation: $t(n) \in \Theta(g(n))$

Big O examples

- **All that we must do to prove that t(n) is O(g(n)) is produce a pair of numbers c and n_0 that work for that case.**
- $t(n) = n$, $g(n) = n^2$.
- $t(n) = n$, $g(n) = 3n$.
- $t(n) = n + 12$, $g(n) = n$.
  We can choose $c = 3$ and $n_0 = 6$, or $c = 4$ and $n_0 = 4$.
- $t(n) = n + \sin(n)$
- $t(n) = n^2 + \text{sqrt}(n)$

In CSSE 230, we do these in great detail in class.

In 473, I say, "work on them if you need review/practice", and I give you a few possible answers on the next slide.

Answers to examples

- For this discussion, assume that all functions have non-negative values, and that we only care about **n**≥0.
  For any function g(n), O(g(n)) is a set of functions We say that a function f(n) is (in) O(g(n)) if there exist two positive constants **c** and $\mathbf{n_0}$ such that for all $n \geq n_0$, $f(n) \leq c\, g(n)$.
- **So all we must do to prove that f(n) is O(g(n)) is produce two such constants.**
- $f(n) = n + 12$, $g(n) = ???$.
  - **$g(n) = n$. Then $c = 3$ and $n_0 = 6$, or $c = 4$ and $n_0 = 4$, etc.**
  - $f(n) = n + \sin(n)$: **$g(n) = n$, $c = 2$, $n_0 = 1$**
  - $f(n) = n^2 + \text{sqrt}(n)$: **$g(n) = n2$, $c=2$, $n_0 = 1$**

Limits and asymptotics

- Consider the limit $$\lim_{n\to\infty} \frac{t(n)}{g(n)}$$
- What does it say about asymptotics if this limit is zero, nonzero, infinite?
- We could say that knowing the limit is a sufficient but not necessary condition for recognizing big-oh relationships.
- It will be useful for many examples in this course.
- **Challenge:** Use the formal definition of limit and the formal definition of big-oh to prove these properties.

Apply this limit property to the following pairs of functions

1. N and N^2
2. $N^2 + 3N + 2$ and N^2
3. $N + \sin(N)$ and N
4. $\log N$ and N
5. $N \log N$ and N^2
6. N^a and a^N $(a > 1)$
7. a^N and b^N $(a < b)$
8. $\log_a N$ and $\log_b N$ $(a < b)$
9. $N!$ and N^N

Big-Oh Style

- **Give tightest bound you can**
  - Saying that **$3N+2 \in O(N^3)$** is true, but not as useful as saying it's $O(N)$ [What about **$\Theta(N^3)$** ?]
- **Simplify:**
  - You *could* say:
  - $3n+2 \in O(5n-3\log(n) + 17)$
  - and it would be technically correct…
  - But **$3n+2 \in O(n)$** is better.
- **true or false? $3n+2 \in O(n^3)$**

Interlude

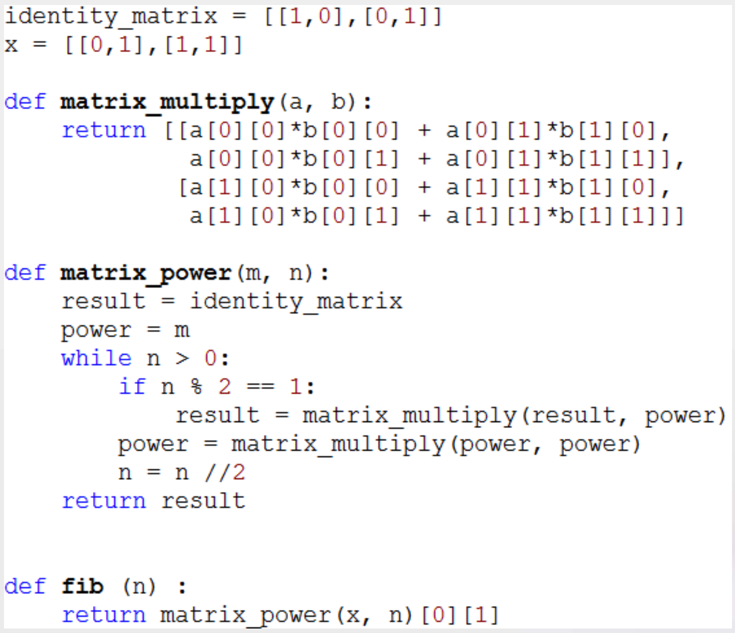
If you have been procrastinating on HW1, this is your last chance. Good luck!

BACK TO OUR ARITHMETIC THREAD

More efficient Fibonacci algorithm?

- Let X be the matrix $\begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix}$
- Then $\begin{pmatrix} F_1 \\ F_2 \end{pmatrix} = X \cdot \begin{pmatrix} F_0 \\ F_1 \end{pmatrix}$
- also $\begin{pmatrix} F_2 \\ F_3 \end{pmatrix} = X \cdot \begin{pmatrix} F_1 \\ F_2 \end{pmatrix} = X^2 \cdot \begin{pmatrix} F_0 \\ F_1 \end{pmatrix}, \ldots, \begin{pmatrix} F_n \\ F_{n+1} \end{pmatrix} = X^n \cdot \begin{pmatrix} F_0 \\ F_1 \end{pmatrix}$
- How many additions and multiplications of numbers are needed to compute the product of two 2x2 matrices?
- If n = 2^k, how many matrix multiplications does it take to compute X^n?
  - What if n is not a power of 2?
  - Implement it with a partner **(details on next slide)**
  - Then we will analyze it
- But there is a catch!

```
identity_matrix = [[1,0],[0,1]] #a constant
x = [[0,1],[1,1]]               #another constant

def matrix_multiply(a, b): #why not do loops?
    return [[a[0][0]*b[0][0] + a[0][1]*b[1][0],
             a[0][0]*b[0][1] + a[0][1]*b[1][1]],
            [a[1][0]*b[0][0] + a[1][1]*b[1][0],
             a[1][0]*b[0][1] + a[1][1]*b[1][1]]]

def matrix_power(m, n):  #efficiently calculate m^n
    result = identity_matrix
    # Fill in the details



    return result


def fib (n) :
    return matrix_power(x, n)[0][1]

# Test code
print ([fib(i) for i in range(11)])
```



```
identity_matrix = [[1,0],[0,1]]
x = [[0,1],[1,1]]

def matrix_multiply(a, b):
    return [[a[0][0]*b[0][0] + a[0][1]*b[1][0],
             a[0][0]*b[0][1] + a[0][1]*b[1][1]],
            [a[1][0]*b[0][0] + a[1][1]*b[1][0],
             a[1][0]*b[0][1] + a[1][1]*b[1][1]]]

def matrix_power(m, n):
    result = identity_matrix
    power = m
    while n > 0:
        if n % 2 == 1:
            result = matrix_multiply(result, power)
        power = matrix_multiply(power, power)
        n = n //2
    return result


def fib (n) :
    return matrix_power(x, n)[0][1]
```

Why so complicated?

- Why not just use the formula that you probably proved by induction in CSSE 230* to calculate F(N)?

$$f(N) = \frac{1}{\sqrt{5}}\left[\left(\frac{1+\sqrt{5}}{2}\right)^n - \left(\frac{1-\sqrt{5}}{2}\right)^n\right]$$

***See Weiss, exercise 7.8**

For review, this proof is part of HW1.

Can we do better than O(k^2)?

- Is there an algorithm for multiplying two k-bit numbers in time that is less than O(k^2)?
- **Basis:** A discovery of Carl Gauss (1777-1855)
  - Multiplying complex numbers:
  - **(a + bi)(c+di) = ac – bd + (bc + ad)i**
  - Needs **four** real-number multiplications and **three** additions
  - But **bc + ad = (a+b)(c+d) – ac –bd**
  - And we have already computed **ac** and **bd** when we computed the real part of the product!
  - Thus we can do the original product with 3 multiplications and 5 additions
  - Additions are so much faster than multiplications that we can essentially ignore them.
  - A little savings, but not a big deal until applied recursively!

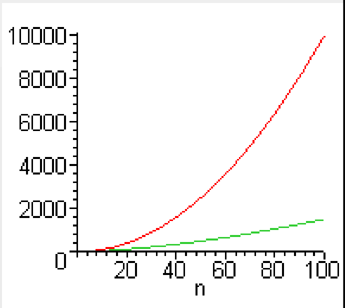
Code for Gauss-based Algorithm

```
def multiply(x, y, n):
    """multiply two integers x and y, where n >= 0
       is a power of 2, and as large as the maximum number of bits in x or y"""

    if n == 1:
        return x * y

    n_over_two = n // 2  # simply shifts the bits one to the right.

    two_to_the_n_over_two = 1 << n_over_two


    xL, xR = x // two_to_the_n_over_two, x % two_to_the_n_over_two
    yL, yR = y // two_to_the_n_over_two, y % two_to_the_n_over_two
    # note that these two operations could be done by bit shifts and masking.

    p1 = multiply (xL,    yL,    n_over_two)
    p2 = multiply (xL+xR, yL+yR, n_over_two)
    p3 = multiply (xR,    yR,    n_over_two)


    return (p1 << n) + ((p2 - p3 - p1) << n_over_two) + p3
```

Is this really a lot faster?

- Standard multiplication: $\Theta(k^2)$
- Divide and conquer with Gauss trick: $\Theta(k^{1.59})$
  - Write and solve the recurrence
- But there is a lot of additional overhead with Gauss, so standard multiplication is faster for small values of n.

```
plot( {n^2, n^1.59}, n=0..100);
```



- In reality we would not let the recursion go down to the single bit level, but only down to the number of bits that our machine can multiply in hardware without overflow.