

MA/CSSE 473

Day 04

Arithmetic
Asymptotics



MA/CSSE 473 Day 03

- HW 1 is due now
 - Solution will be available on ANGEL at 9:15
- HW 2 is available
- **Student Questions**
- **HW1 discussion**
- Big O, Big Omega, Big Theta
- Continue Algorithm Overview/Review
 - A Closer Look at Multiplication
 - Four different multiplication algorithms
 - Divide and Conquer
 - Divide and Conquer plus Gauss
 - Impact on Fibonacci
 - Integer Primality Testing and Factoring



Review: Definitions of O , Θ , Ω

- I will re-use some of my slides from CSSE 230
- And some of Levitin's slides
- A very similar presentation appears in Levitin, section 2.2
- Hopefully, since this is review, it can go quicker than in 230



Asymptotic Analysis

- We only really care what happens when N (the size of a problem) gets large
- Is the function linear? quadratic? etc.
- Two 7" pizzas

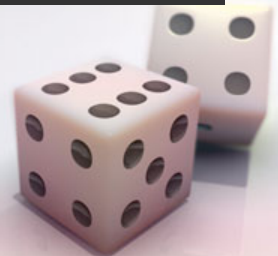


Asymptotic order of growth

Informal definitions

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

- $O(g(n))$: class of functions $f(n)$ that grow no faster than $g(n)$
- $\Theta(g(n))$: class of functions $f(n)$ that grow at same rate as $g(n)$
- $\Omega(g(n))$: class of functions $f(n)$ that grow at least as fast as $g(n)$



Formal Definition

- With another person, try to write a precise formal definition " $F(n) \in O(g(n))$ "
 - This is one thing that students in this course should eventually be able to do from memory...
 - ... and also understand, of course!



Big-oh

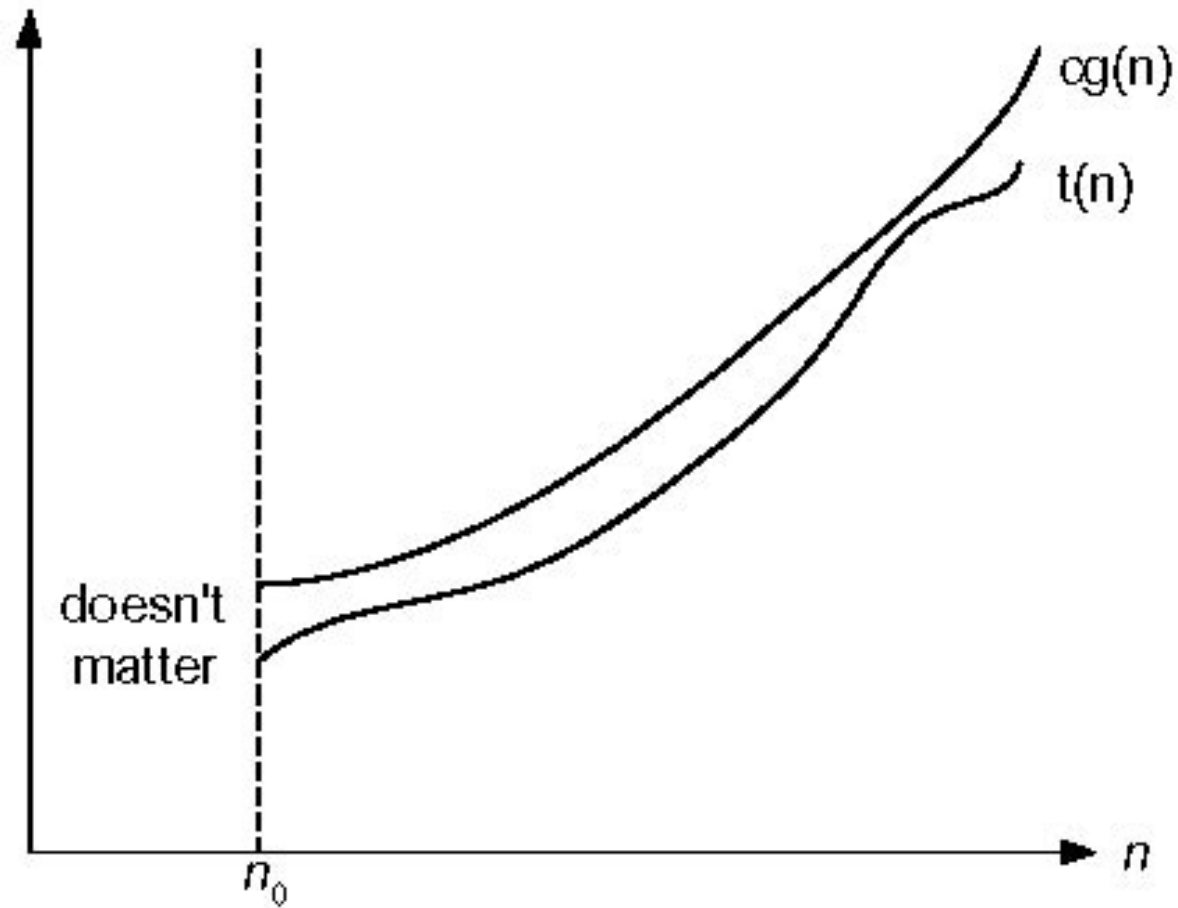


Figure 2.1 Big-oh notation: $t(n) \in O(g(n))$



Prove a Big O Property

- For any function $g(n)$, $O(g(n))$ is a set of functions
- We say that $f(n) \in O(g(n))$ if there exist two positive constants c and n_0 such that for all $n \geq n_0$, $f(n) \leq c g(n)$
- **Rewrite using \forall and \exists notation**
- If $f(n) \in O(h(n))$ and $g(n) \in O(h(n))$, then $f(n)+g(n) \in O(h(n))$
- Let's prove it



Big-omega

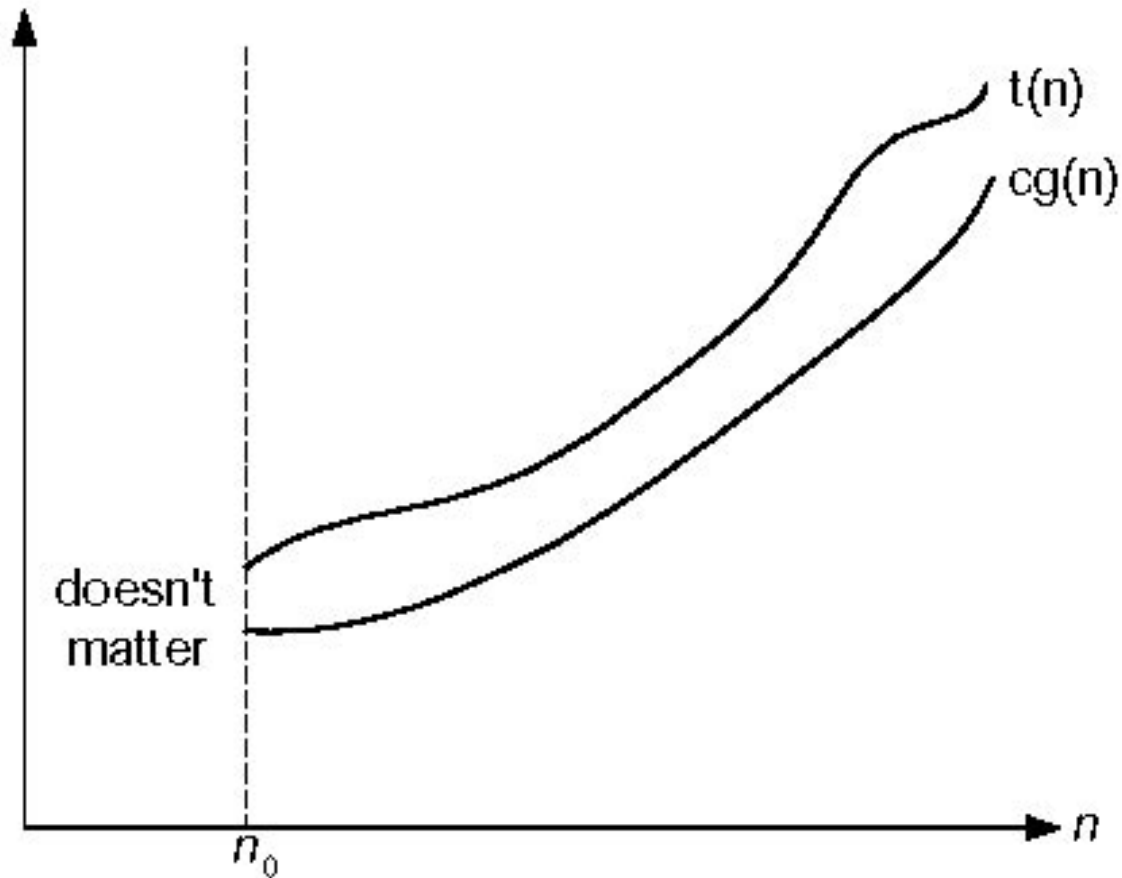


Fig. 2.2 Big-omega notation: $t(n) \in \Omega(g(n))$



Big-theta

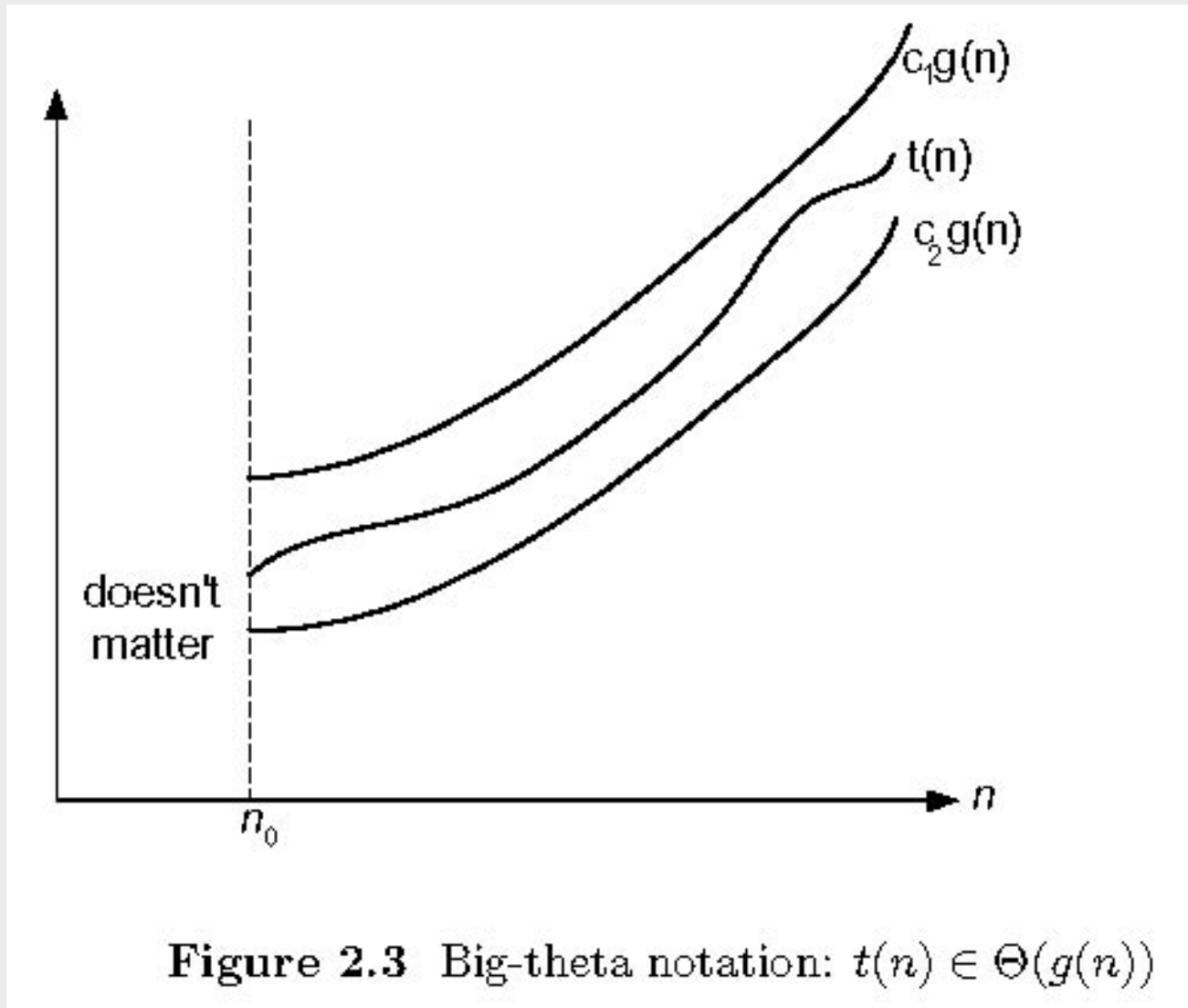


Figure 2.3 Big-theta notation: $t(n) \in \Theta(g(n))$



Big O examples

- All that we must do to prove that **$f(n)$ is $O(g(n))$** is produce a pair of numbers c and x_0 that work for that case.
- $f(n) = n, g(n) = n^2$.
- $f(n) = n, g(n) = 3n$.
- $f(n) = n + 12, g(n) = n$.
We can choose $c = 3$ and $n_0 = 6$, or $c = 4$ and $n_0 = 4$.
- $f(n) = n + \sin(n)$
- $f(n) = n^2 + \text{sqrt}(n)$



Limits and asymptotics

- Consider the limit

$$\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)}$$

- 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 sufficient for most examples in this course.
- **Challenge:** Use the formal definition of limit and the formal definition of big-oh to prove these properties.



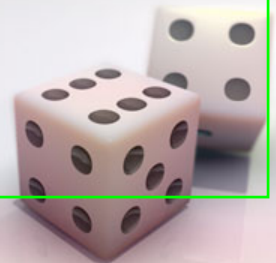
Apply this limit property to the following pairs of functions

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 N^n
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:
 - $3n+2$ is $O(5n-3\log(n) + 17)$
 - and it would be technically correct...
 - But $3n+2 \in O(n)$ is better
- **But... if I ask “true or false: $3n+2 \in O(n^3)$ ”, what’s the answer?**
 - True!



Recap: Gauss discovery

- Discovery of Carl Gauss (1777-1855)
 - Multiplying complex numbers:
 - $(a + bi)(c+di) = ac - bd + (bc + ad)i$
 - Seems to need four real-number multiplications and two additions
 - But $bc + ad = (a+b)(c+d) - ac - bd$
 - And we have already computed ac and bd for the real part!
 - 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!



Review: The Master Theorem

- The Master Theorem for Divide and Conquer recurrence relations:
- Consider the recurrence $T(n) = aT(n/b) + f(n)$, $T(1)=c$, where $f(n) = \Theta(n^k)$ and $k \geq 0$,
- The solution is
 - $\Theta(n^k)$ if $a < b^k$
 - $\Theta(n^k \log n)$ if $a = b^k$
 - $\Theta(n^{\log_b a})$ if $a > b^k$



Recap: New Multiplication Approach

- **Divide and Conquer**

- To multiply two n -bit integers x and y :

- Split each into its left and right halves so that

$$x = 2^{n/2} x_L + x_R, \quad \text{and} \quad y = 2^{n/2} y_L + y_R$$

- The straightforward calculation of xy would be

$$(2^{n/2} x_L + x_R)(2^{n/2} y_L + y_R) = 2^n x_L y_L + 2^{n/2}(x_L y_R + x_R y_L) + x_R y_R$$

- Code on next slide

- We can do the four multiplications of $n/2$ -bit integers using four recursive calls, and the rest of the work (a constant number of bit shifts and additions) in time $O(n)$

- Thus $T(n) =$. Solution?



Code for divide-and-conquer

```
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  
  
    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, yR, n_over_two)  
    p3 = multiply (xR, yL, n_over_two)  
    p4 = multiply (xR, yR, n_over_two)  
  
    return (p1 << n) + ((p2 + p3) << n_over_two) + p4
```



Faster Multiplication Approach

- **Divide and Conquer a la Gauss**

- To multiply two n -bit integers x and y :

- Split each into its left and right halves so that

$$x = 2^{n/2}x_L + x_R, \quad \text{and} \quad y = 2^{n/2}y_L + y_R$$

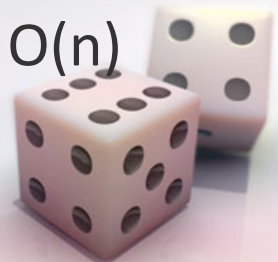
- The fancier calculation of xy is

$$(2^{n/2}x_L + x_R)(2^{n/2}y_L + y_R) = \\ 2^n x_L y_L + 2^{n/2}((x_L + x_R)(y_L + y_R) - x_L y_L - x_R y_R) + x_R y_R$$

- Code on next slide

- We can do the three multiplications of $n/2$ -bit integers using three recursive calls, and the rest of the work (a constant number of bit shifts and additions) in time $O(n)$

- Thus $T(n) =$. Solution?



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  
  
    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 faster?

- Standard multiplication: $\Theta(n^2)$
- Divide and conquer with Gauss trick: $\Theta(n^{1.59})$
- 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
down to the single bit level, but
the number of bits that our
machine can multiply in hardware
overflow.

