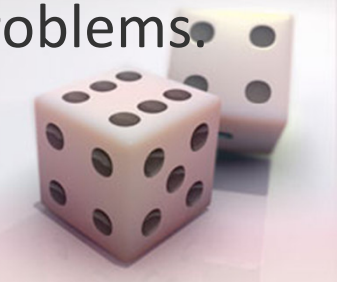
MA/CSSE 473 Day 03

Asymptotics

A Closer Look at Arithmetic

Announcements

- Optional session on basics of mathematical induction: Tomorrow 10th hour.
  - Room: O267
- A quote from the HW1 document:
  - **A look ahead to HW 2:** Partly because it is due early in the term, I made HW 1 rather short. HW 2 is longer, and is due two days after HW 1. So ideally you should do a few problems from HW 2 before HW 1 is due. But because some students have so many adjustments at the beginning of the term, I am not requiring you to have them done until Wednesday.
  - If you have not done any of the HW2 problems yet; be sure to start soon. They tend to be harder than HW1 problems.



Two threads in lectures

- Each day at the beginning of the course
- A little review (today it's a lot)
- Continue with a efficiency of Fibonacci and arithmetic.

Review thread for today:
Asymptotics (O, Θ, Ω)

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

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

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 soon be able to do from memory…
  - … and also understand, of course!

Q 1

Big-oh

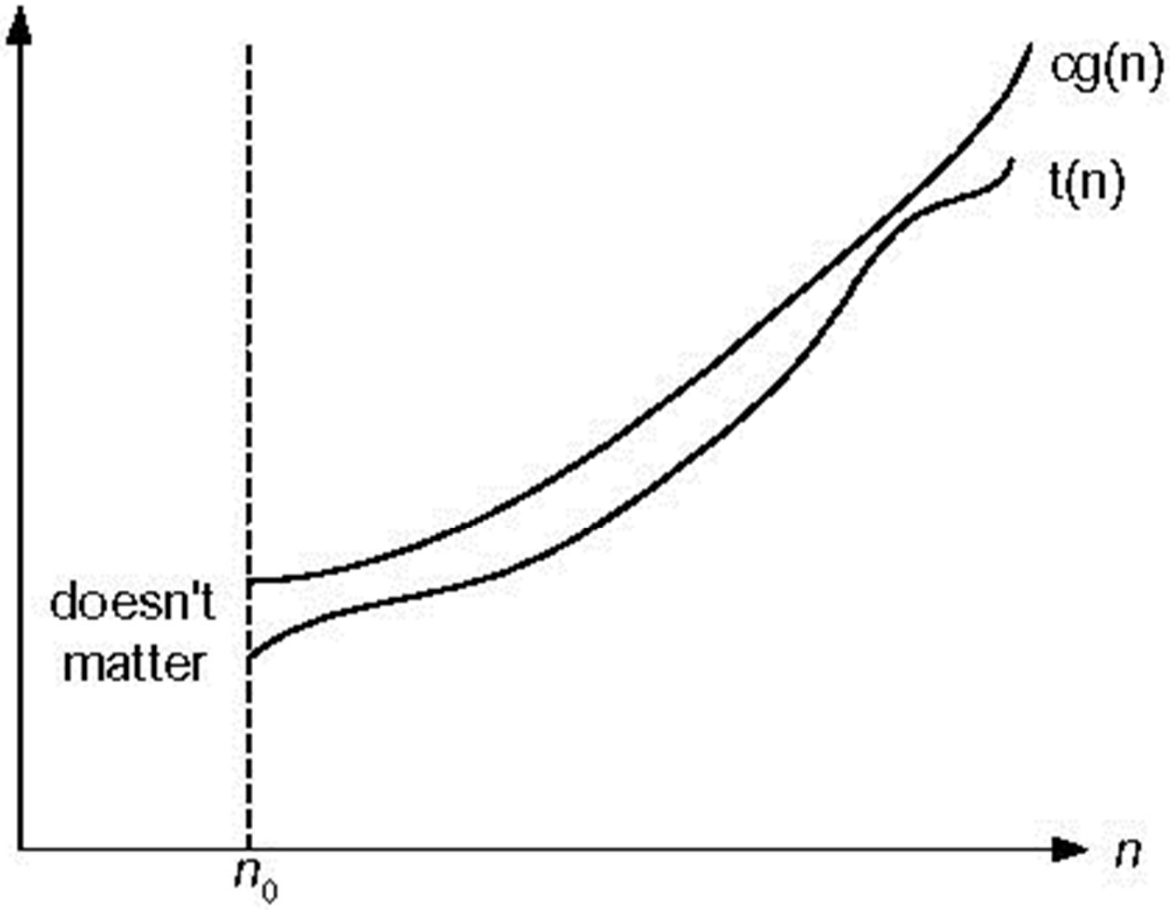


Figure 2.1 Big-oh notation: $t(n) \in O(g(n))$

Prove a Big O Property

- For any function g(n), O(g(n)) is a set of functions
- We say that **$f(n) \in O(g(n))$** iff 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

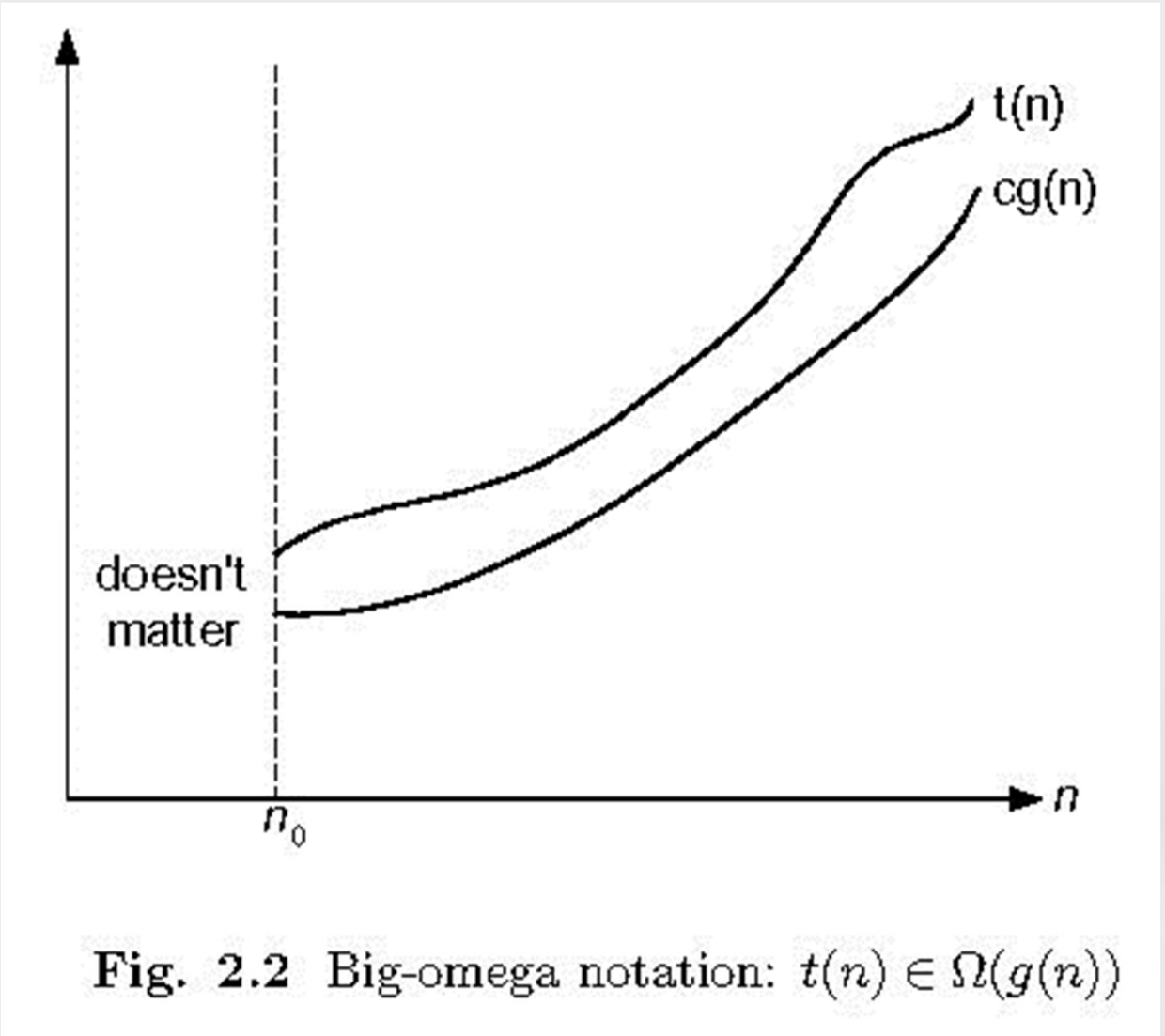


Fig. 2.2 Big-omega notation: $t(n) \in \Omega(g(n))$

Big-theta

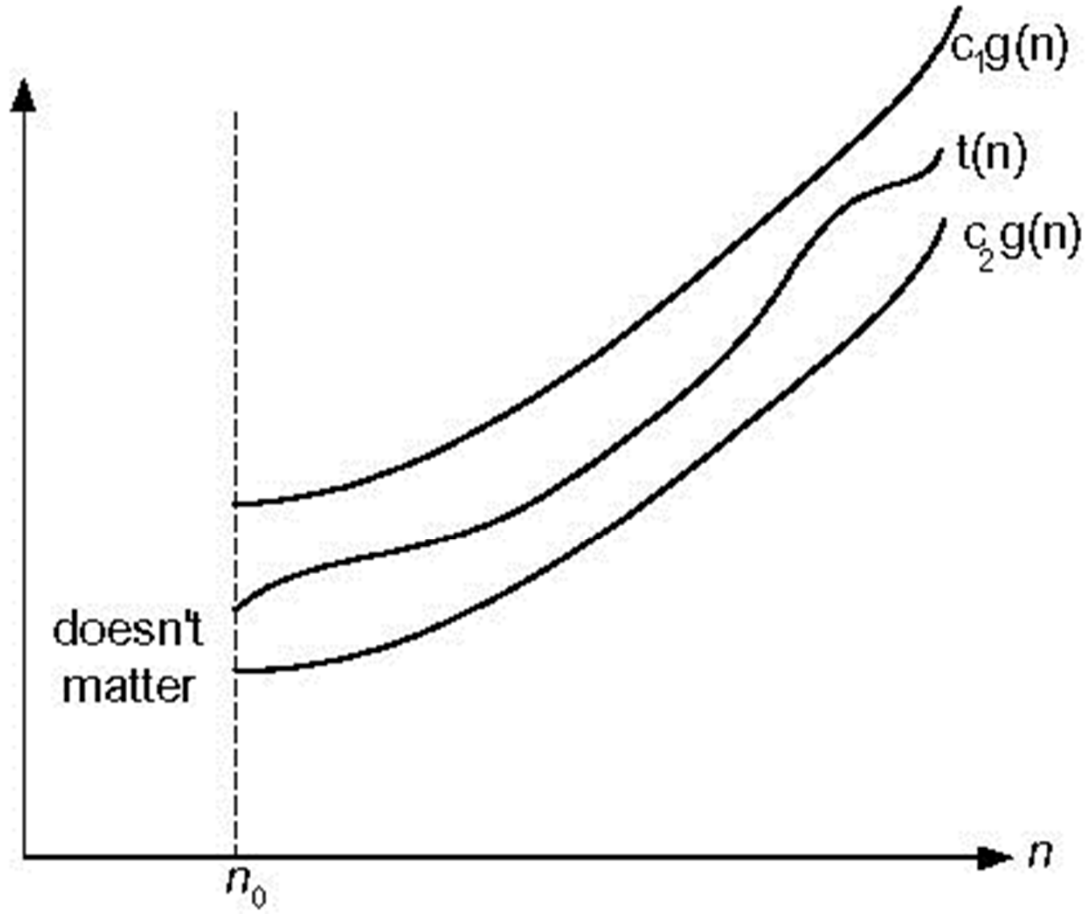


Figure 2.3 Big-theta notation: $t(n) \in \Theta(g(n))$

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 + sqrt(n)

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

In 473, I will say, "work on them if you need to, " and 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 + sqrt(n): **g(n) = n2, c=2, n_0 = 1**

Limits and asymptotics

- Consider the limit $$\lim_{n \to \infty} \frac{f(n)}{g(n)}$$

- 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-3log(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!

BACK TO FIBONACCI AND ARITHMETIC THREAD

Recap: Fibonacci Numbers

- Straightforward recursive algorithm:
  - Correctness is obvious
  - $T(N) \geq F(N)$, which is exponential ($\approx 2^{0.69N}$)

```
def fib1(n):
    if n==0:
        return 0
    if n==1:
        return 1
    return fib1(n-1) + fib1(n-2)

print fib1(6), fib1(7), fib1(8)
```

- Algorithm with storage to avoid recomputing:
  - Again, correctness is obvious
  - And the run-time is linear
    - until we dig deeper later

```
def fib2(n):
    nums = [0]*(n+1)
    nums[0] = 0
    nums[1] = 1
    for i in range(2, n+1):
        nums[i] = nums[i-1] + nums[i-2]
    return nums[n]
```

A more efficient 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 $\begin{pmatrix} F_2 \\ F_3 \end{pmatrix} = X \cdot \begin{pmatrix} F_1 \\ F_2 \end{pmatrix} = X^2 \cdot \begin{pmatrix} F_0 \\ F_1 \end{pmatrix}, and \begin{pmatrix} F_n \\ F_{n+1} \end{pmatrix} = X^n \cdot \begin{pmatrix} F_0 \\ F_1 \end{pmatrix}$
- How many additions and multiplications of numbers are necessary to multiply 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 (next slide)
  - Then we will analyze it
- But there is a catch!

```
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
    # Fill in the details




    return result


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

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]
```

Back to the end of the 2nd previous slide!

Q 4

Why so complicated?

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

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

The catch!

- Are addition and multiplication constant-time operations?
- We take a closer look at the "basic operations"
- **Addition first:**
- At most, how many digits in the sum of three decimal one-digit numbers?
- Is the same result true in binary and every other base?
- Add two n-bit positive integers (53+35):

Carry:

```
1           1   1   1
    1   1   0   1   0   1     (35)
    1   0   0   0   1   1     (53)
-------------------------
1   0   1   1   0   0   0     (88)
```

- So adding two n-bit integers is O( ).

Multiplication

- Example: multiply 13 by 11

				1	1	0	1	
			x	1	0	1	1	
				1	1	0	1	(1101 times 1)
			1	1	0	1		(1101 times 1, shifted once)
		0	0	0	0			(1101 times 1, shifted twice)
	1	1	0	1				(1101 times 1, shifted thrice)
1	0	0	0	1	1	1	1	(binary 143)

- There are n rows of $2n$ bits to add, so we do an O(n) operation n times, thus the whole multiplication is O( ) ?
- Can we do better?

Multiplication by an Ancient Method

- This approach was known to Al Khwarizimi
- According to Dasgupta, *et al*, still used today in some European countries
- Repeat until 1st number is 1, keeping all results:
  - Divide 1st number by 2 (rounding down)
  - double 2nd number
- Example

11	13
5	26
~~2~~	~~52~~
1	104
	143

Then strike out any rows whose first number is even, and add up the remaining numbers in the second column.

- Correct? Analysis

Q 9

Recursive code for this algorithm

```
def multiply(m, n):
    "multiply two integers m and n, where n >= 0"
    if n == 0:
        return 0
    z = multiply (m, n // 2)
    if n % 2 == 0:
        return 2 * z
    return m + 2 * z

print (multiply(12, 17))
```

For reference: 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 $a > b^k$

For details, see Levitin pages 483-485 or Weiss section 7.5.3.

Grimaldi's Theorem 10.1 is a special case of the Master Theorem.

We will use this theorem often. You should review its proof soon (Weiss's proof is a bit easier than Levitin's).

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

    $x = 2^{n/2}x_L + x_R$, and $y = 2^{n/2}y_L + y_R$
  - The straightforward calculation of xy would be

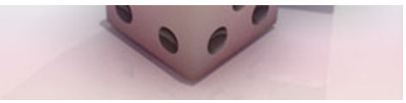
    $(2^{n/2}x_L + x_R)(2^{n/2}y_L + y_R) =$

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

Q 10

Code for divide-and-conquer multiplication

```
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 # a single right bit-shift


    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
```



Can we do better than $O(n^2)$?

- Is there an algorithm for multiplying two n-bit numbers in time that is less than $O(n^2)$?
- **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!

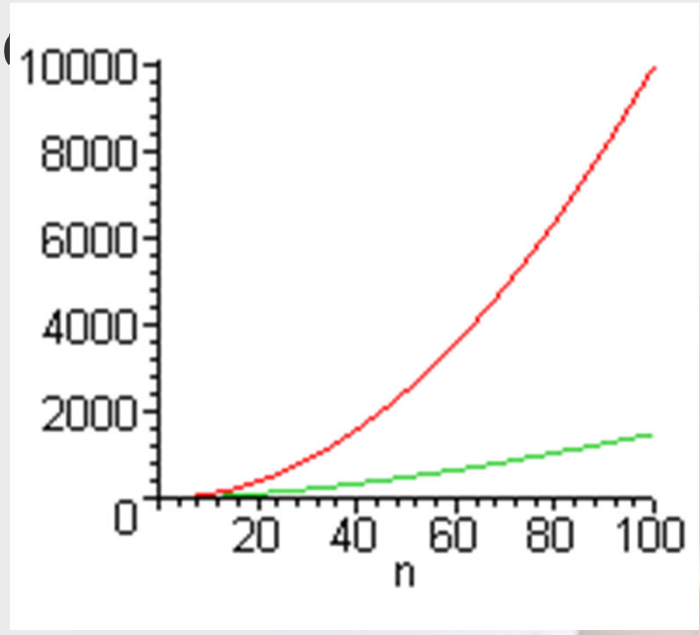
Code for Gauss-based Algorithm

```
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: $\Theta(n^2)$
- Divide and conquer with Gauss trick: $\Theta(n^{1.59})$
  - Write and solve the recurrence
- But there is a lot of additional overhead with Gauss, so standard multiplicati 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.