Data Structures and Algorithms

Analysis of Algorithms Cont.

Parts of this presentation are based on the slides of Prof. David Galles

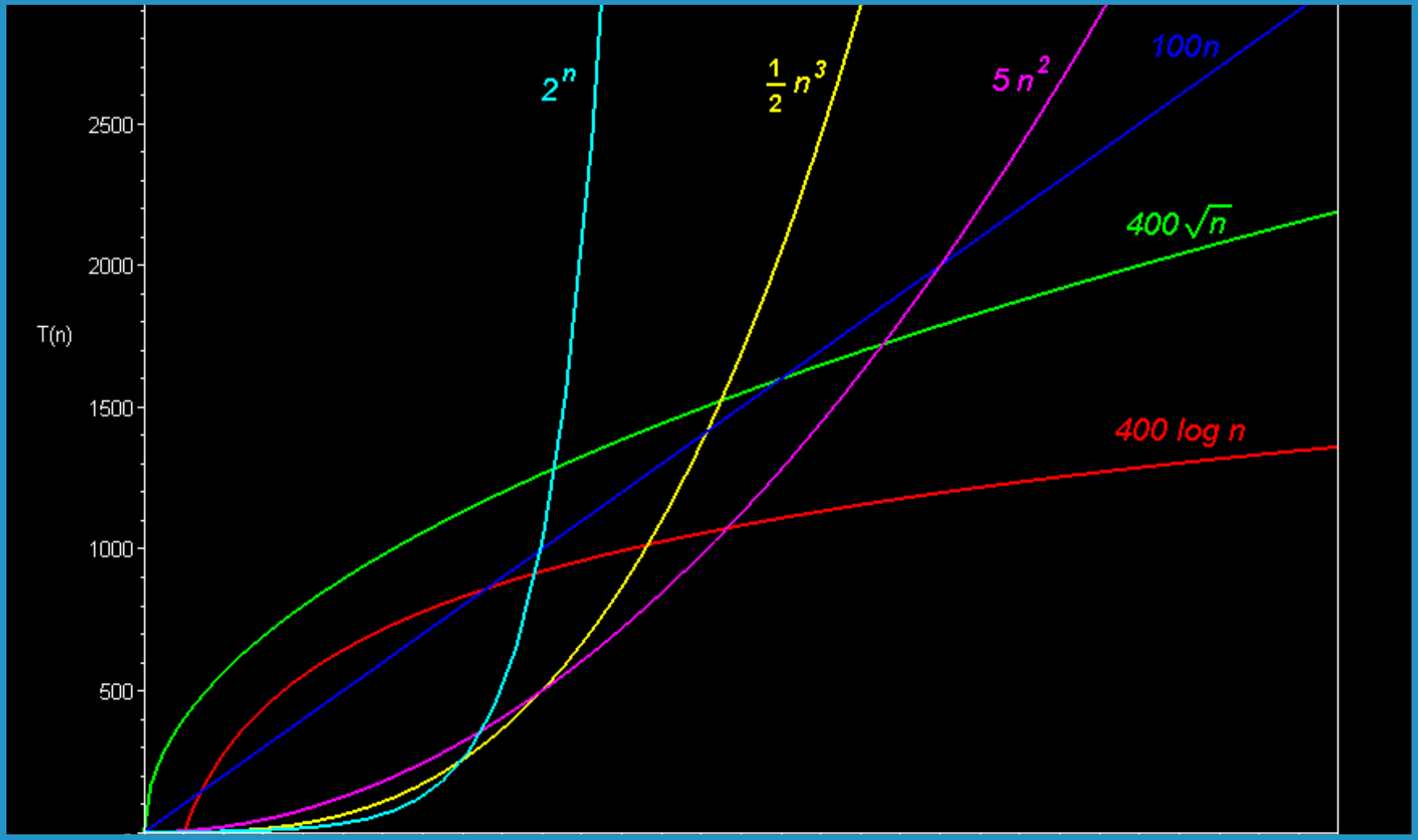
Announcements

- Homework 1 is out
- Teaching Assistant: Anthony Knox
  - Office hours: Monday and Friday 2:30-3:30pm

Recap: Running Time

- The number of simple operations required for an input of size n
- Is a function of n
- Can compare algorithms by comparing the corresponding functions

Function growth rate



http://science.slc.edu/~jmarshall/courses/2002/spring/cs50/BigO/

Mathematical Foundations

- Summations, plots of functions

Arithmetic Series

$1 + 2 + 3 + \ldots + n = n*(n+1) / 2$

Can prove by induction

Geometric Series

- The sum of terms
- Each term after the first is found by multiplying the previous one by a constant

$r^0 + r^1 + r^2 + \ldots + r^{n-1} = (1-r^n) / (1-r)$

- Ex1: compute $2^0 + 2^1 + 2^2 + 2^3 + 2^4$
- If $-1 < r < 1$, the solution is $1 / (1- r)$
- Ex2: compute $1 + 1/2 + 1/4 + 1/8 + 1/16 + 1/32 + \ldots$

Logarithm Rules

$\log(a*b) = \log(a) + \log(b)$

$\log(a/b) = \log(a) - \log(b)$

$\log(a^b) = b * \log(a)$

Changing the base: $\log_a b = \log_c b / \log_c a$

Comparing Growth Rates

- Which of these two grows faster?

- $n^2 * \log n$ and 2^n (Hint: take the logarithm of both)
- 10^5 and $0.01*n$
- $0.1*n^3$ and $1000*\log n$
- n^{10} and 1.01^n

Exercise

- Rank the following functions in the order of growth (from slowest to fastest)

  2^{1000}

  $2^{\log(n)}$

  $3n^{0.01}$

  $n^3 + 2n^2$

  $6n\log(n) + n$

  $\log^2 n$

  $4n^{3/2}$

  $\sqrt{\log n}$

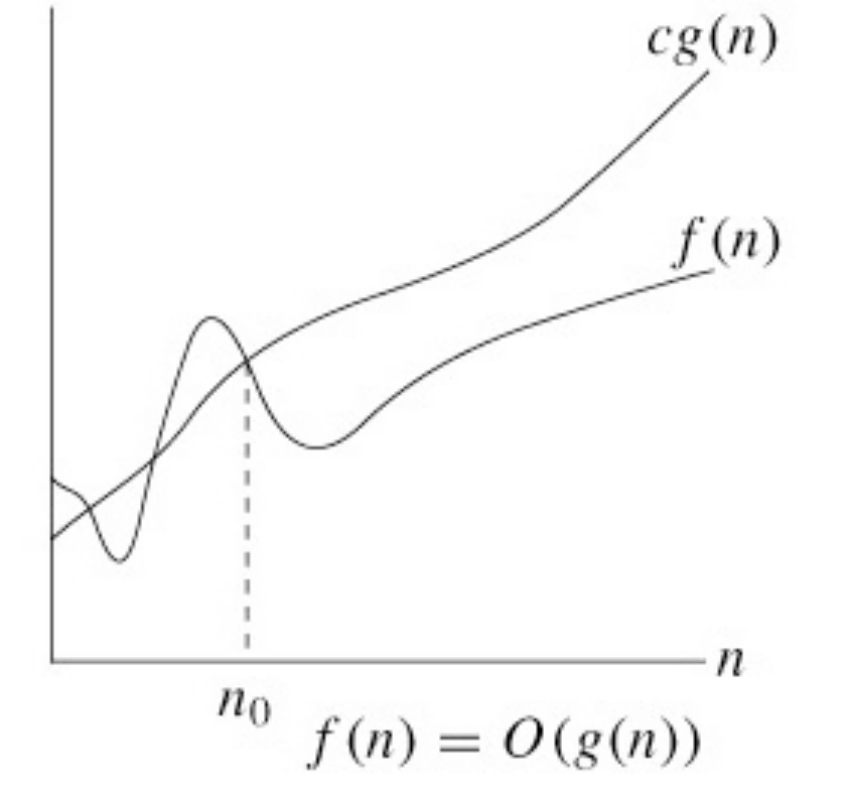
Big O

- The most common method & notation for discussing the running time
- “Asymptotic upper bound”

- f(n) = O(g(n))
- n is the input size

Big O: f(n) = O(g(n))

- Intuitively: f(n) does not grow faster than g(n)
  - C*g(n) is an upper bound



From Cormen, Leiserson and Rivest, "Introduction to Algorithms"

Big O

- $f(n) = O(g(n))$

There exist constants C, N such that

$f(n) \leq C^* g(n)$

for $n>N$

Can assume f(n), g(n) are positive since they are time complexity functions

Example

$f(n) = 3n^2 + 2n = ?$

$3n^2 + 2n < 3n^2 + 2n^2 = 5n^2$

Example

$f(n) = 3n^2 + 2n = ?$

$3n^2 + 2n < 3n^2 + 2n^2 = 5n^2$

So for $g(n) = n^2$, $C = 5$, $N = 1$:

$f(n) <= C*g(n)$ $n > N$

Hence $f(n) = O(n^2)$

Exercise: Prove the following:

- $3n = O(n)$
- $n = O(2^n)$

g(n) is *not* guaranteed to be a *tight* upper bound!

Common Classes of Functions

$O(1)$	Constant
$O(\log(n))$	Logarithmic
$O(n)$	Linear
$O(n \log(n))$	
$O(n^2)$	Quadratic
$O(n^c)$	Polynomial
$O(c^n)$	Exponential

Where does "O" come from?

- O = Order (rate of growth), Ordnung
- Also called Landau's symbol

  After the German mathematician who invented this notation

Big O Guidelines

- If f(n) is the sum of different functions, the faster growing function determines the order of f(n)
- Ex: Let $f(n) = 10 + \log(n) + (5 \log(n))^3 + n^3 + n$
- f(n) = O(?)

Big O Guidelines

- If f(n) is the sum of different functions, the faster growing function determines the order of f(n)
- Ex: $f(n) = 10 + \log(n) + (5 \log(n))^3 + n^3 + n = O(n^3)$
- Constants and low-order terms don't matter – do not include them

Big O Guidelines

- If $f_1(x)$ is $O(g_1(x))$, $f_2(x)$ is $O(g_2(x))$, then $(f_1 + f_2)(x)$ is $O(\max(g_1(x), g_2(x)))$
- If $f_1(x)$ is $O(g_1(x))$ and $f_2(x)$ is $O(g_2(x))$, then $(f_1 f_2)(x)$ is $O(g_1(x)\, g_2(x))$

Big O Guidelines

- Lower order terms matter if we multiply by them!
- $f(n) = 10 * \log(n) * (5 \log(n))^3 * n^3 * n = O(n^4 * (\log n)^4)$

Examples

- 2^{1000}
- $2*n^3+10$
- $8n^2 + 10n * \log(n) + 100n + 10^{10}$
- $f_{\text{linear search}}(n) = O(n)$
- $f_{\text{binarySearch}}(n) = O(\log(n))$

Big O Examples: True or False?

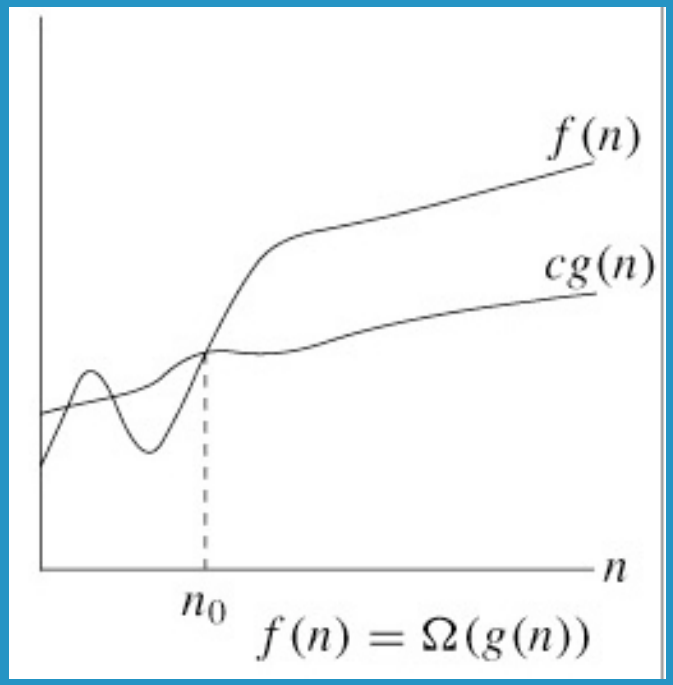
- $\sqrt{n} \in O(n)$
- $\lg n \in O(2^n)$
- $\lg n \in O(n)$
- $n \lg n \in O(n)$
- $n \lg n \in O(n^2)$
- $\sqrt{n} \in O(\lg n)$
- $n^3 + n \lg n + n\sqrt{n} \in O(n \lg n)$
- $n^3 + n \lg n + n\sqrt{n} \in O(n^3)$

Big O Examples

- $\sqrt{n} \in O(n)$
- $\lg n \in O(2^n)$
- $\lg n \in O(n)$
- ~~$n \lg n \in O(n)$~~
- $n \lg n \in O(n^2)$
- ~~$\sqrt{n} \in O(\lg n)$~~
- ~~$n^3 + n \lg n + n\sqrt{n} \in O(n \lg n)$~~
- $n^3 + n \lg n + n\sqrt{n} \in O(n^3)$

Big Omega

- f(n) = Ω(g(n)) if
  - Intuitively: Lower bound
  - Formally: there are positive constants C and N
    f(n) ≥ C*g(n)) when n ≥ N



Big Omega

- Not guaranteed to be a tight lower bound

- Examples:

  $3n^2$ in Omega(n^2)

  $3n^2$ in Omega(n)

  $3n^2$ in Omega(log n)

Big Ω Examples: True or False?

- $\sqrt{n} \in \Omega(n^{0.25})$
- $n*\lg n \in \Omega(\lg n)$
- $\lg n \in \Omega(n)$
- $n^2 \in \Omega(n*\log n)$

Big Ω Examples: True or False?

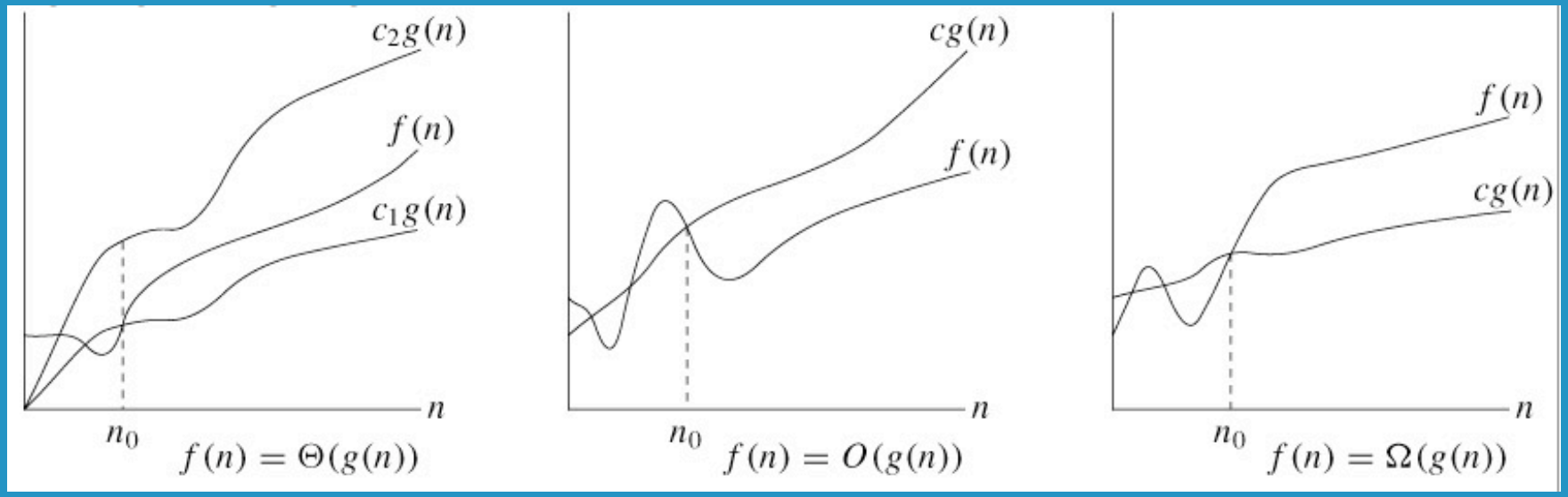
- $\sqrt{n} \in \Omega(n^{0.25})$
- $n*\lg n \in \Omega(\lg n)$
- ~~$\lg n \in \Omega(n)$~~
- $n^2 \in \Omega(n*\log n)$

Big Theta

- $f(n) = \Theta(g(n))$ if
  - Informally: f is bound from above and below by g(n)
  - A tight bound
  - Formally: if $f(n) = O(g(n))$ and $f(n) = \Omega(g(n))$

  There exist constants C_1, C_2, N such that
  $C_1 g(n) <= f(n) <= C_2 g(n)$, for $n >= N$
  That creates an envelope that encapsulates the function

Θ() vs O() vs Ω()



From Cormen, Leiserson and Rivest, "Introduction to Algorithms"

Big Theta: Example

- $3n \lg(n) + n + 5 \lg(n)$ is $\Theta(n \lg(n))$

Justification:

$3n \lg(n) <= 3n \lg(n) + n + 5 \lg(n) <= (3+1+5)n \lg(n)$

for $n >= 2$

Big-Oh Guidelines

- Constants and added low-order terms don't matter – do not include them
- Simple program statements: Theta(1)
- Loops: Theta(inside) * # of iterations
- Conditional (if) statements: O(Test + longest branch)

```
if (i<5)
    sum = 5;
else
    for (int j=0; j<n; j++)
        sum++;
```

Examples

```
for (i=1; i<n; i++)
    sum++;
```

Examples

```
for (i=1; i<n; i++) // n times
    sum++;          // O(1)
```

Running time: O(n), Omega(n), Theta(n)

Examples

```
for (i=1; i<n; i=i+2)
    sum++;
```

Examples

```
for (i=1; i<n; i=i+2)      Executed n/2 times
    sum++;                 O(1)
```

Running time: O(n), Omega(n), Theta(n)

Examples

```
for (i=1; i<n; i=i+1)
 for (j=1; j<n/2; j++)
    sum++;
```

Running time: ?

Examples

```
for (i=1; i<n; i=i+1)
 for (j=1; j<n/2; j++)
    sum++;
```

Executed n times

Executed n/2 times

O(1)

Running time: $O(n^2)$, $\Omega(n^2)$, $\Theta(n^2)$

Examples

```
for (i=1; i<n; i++)
 for (j=1; j< n; j++)
   for (int k=1; k<n; k++)
            func += 3*j + 4*k;
```

Running time: ?

Examples

```
for (i=1; i<n; i++)          Executed n times
 for (j=1; j< n; j++)        Executed n times
   for (int k=1; k<n; k++) Executed n times
          func += 3*j + 4*k;    O(1)
```

Running time: $O(n^3)$, $Omega(n^3)$, $Theta(n^3)$

Example: i is powers of 2

```
for (i=1; i<n; i=i*2)
    sum++;
```

Example: i is powers of 2

```
for (i=1; i<n; i=i*2)      Executed log n times
    sum++;                 O(1)
```

i: 1, 2, 4, 8, ... $= 2^0, 2^1, 2^2, \ldots$

How many iterations before we reach n?

$2^k = n$

$k = \log n$

Running time: O(log n), Omega(log n), Theta(log n)

Examples: non-nested loops

```
int sum = 0;
int i;
for (i=1; i<n; i++)
    sum++;
for (i=1; i<n; i=i*2)
    sum++;
```

Examples

```
int sum = 0;                    O(1)
int i;                          O(1)
for (i=1; i<n; i++)             O(n)
    sum++;                      O(1)
for (i=1; i<n; i=i*2)           O(log n)
    sum++;                      O(1)
```

Running time: O(n), Omega(n), Theta(n)

Example: j depends on i

```
for (i=1; i<=n; i++)
    for (j=1; j<=i; j++)
      sum++;
```

Running time: ?

Example: j depends on i

```
for (i=1; i<=n; i++)        Executed n times
    for (j=1; j<=i; j++)    Executed <= n times
      sum++;                O(1)
```

Running time: $O(n^2)$

What about Theta ? Need to do a summation!

Example: j depends on i

```
for (i=1; i<=n; i++)
    for (j=1; j<=i; j++)
      sum++;
```

The exact # of times sum++ is executed:

$1 + 2 + 3 + \ldots + n = n(n+1)/2 \sim \text{Theta}(n^2)$

Example:

```
for (i=1; i<n; i=i*2)
    for (j=0; j<i; j++)
        sum++;
```

Example: Computing BigO

```
for (i=1; i<n; i=i*2)          log n times
    for (j=0; j<i; j++)        <= n times
        sum++;
```

O(n log n)

What about Omega and Theta?

Example: Computing Theta

```
for (i=1; i<n; i=i*2)
    for (j=0; j<i; j++)
        sum++;
```

To compute Theta, need to do a summation:

$i=2^0=1$: sum++ executed once

$i=2^1=2$: sum++ executed twice

$i=2^2=4$: sum++ executed four times

$i=2^3=8$: sum++ executed eight times

...

$2^0 + 2^1 + 2^2 + 2^3 + \ldots + 2^{\log(n-1)} = ?$

log n terms

Reminder: Geometric Series

- The sum of terms
- Each term after the first is found by multiplying the previous one by a constant

$$r^0 + r^1 + r^2 + \ldots + r^{n-1} = (1-r^n) / (1-r)$$

- Can use this formula to compute

$$2^0 + 2^1 + 2^2 + 2^3 + \ldots + 2^{\log(n-1)}$$

Example: Computing Theta

```
for (i=1; i<n; i=i*2)
    for (j=0; j<i; j++)
        sum++;
```

$$2^0 + 2^1 + 2^2 + 2^3 + \ldots + 2^{\log(n-1)} =$$
$$= (1-2^{\log(n-1)+1})/(1-2) =$$
$$= 2^{\log(n-1)+1} - 1 =$$
$$= 2^{\log(n-1)} * 2^1 - 1 = (n-1)*2 - 1$$
$$= \text{Theta}(n)$$

Example:

```
for (i=1; i<=n; i=i+1)
    for (j=1; j<=i; j=j*2)
        sum++;
```

O(n log n) – why?

Compute Theta

Example:

```
for (i=1; i<=n; i=i+1)
    for (j=1; j<=i; j=j*2)
        sum++;
```

Compute Theta: need to do a summation

log 1 + log 2 + ... + log n > log (n/2) + ..
log (n) > log(n/2)*n/2 = Omega(nlog(n))

Since it's also O(nlogn), it is Theta(nlog(n))