CSE-205
Algorithms

Asymptotic Notation

Analyzing Algorithms

- Predict the amount of resources required:
  - memory: how much space is needed?
  - computational time: how fast the algorithm runs?
- FACT: running time grows with the size of the input
- Input size (number of elements in the input)
  - Size of an array, polynomial degree, # of elements in a matrix, # of bits in the binary representation of the input, vertices and edges in a graph

Def: Running time = the number of primitive operations (steps) executed before termination

- Arithmetic operations (+, -, *), data movement, control, decision making (*if, while*), comparison

Algorithm Analysis: Example

- *Alg.:* MIN (a[1], …, a[n])

```
            m ← a[1];
            for i ← 2 to n
                 if a[i] < m
              then m ← a[i];
```

- **Running time**:
  - the number of primitive operations (steps) executed before termination

  $T(n) = 1$ [first step] + (n) [for loop] + $(n-1)$ [if condition] + $(n-1)$ [the assignment in then] = $3n - 1$

- Order (rate) of growth:
  - The leading term of the formula
  - Expresses the asymptotic behavior of the algorithm

Typical Running Time Functions

- 1 (constant running time):
  - Instructions are executed once or a few times
- logN (logarithmic)
  - A big problem is solved by cutting the original problem in smaller sizes, by a constant fraction at each step
- N (linear)
  - A small amount of processing is done on each input element
- N logN
  - A problem is solved by dividing it into smaller problems, solving them independently and combining the solution

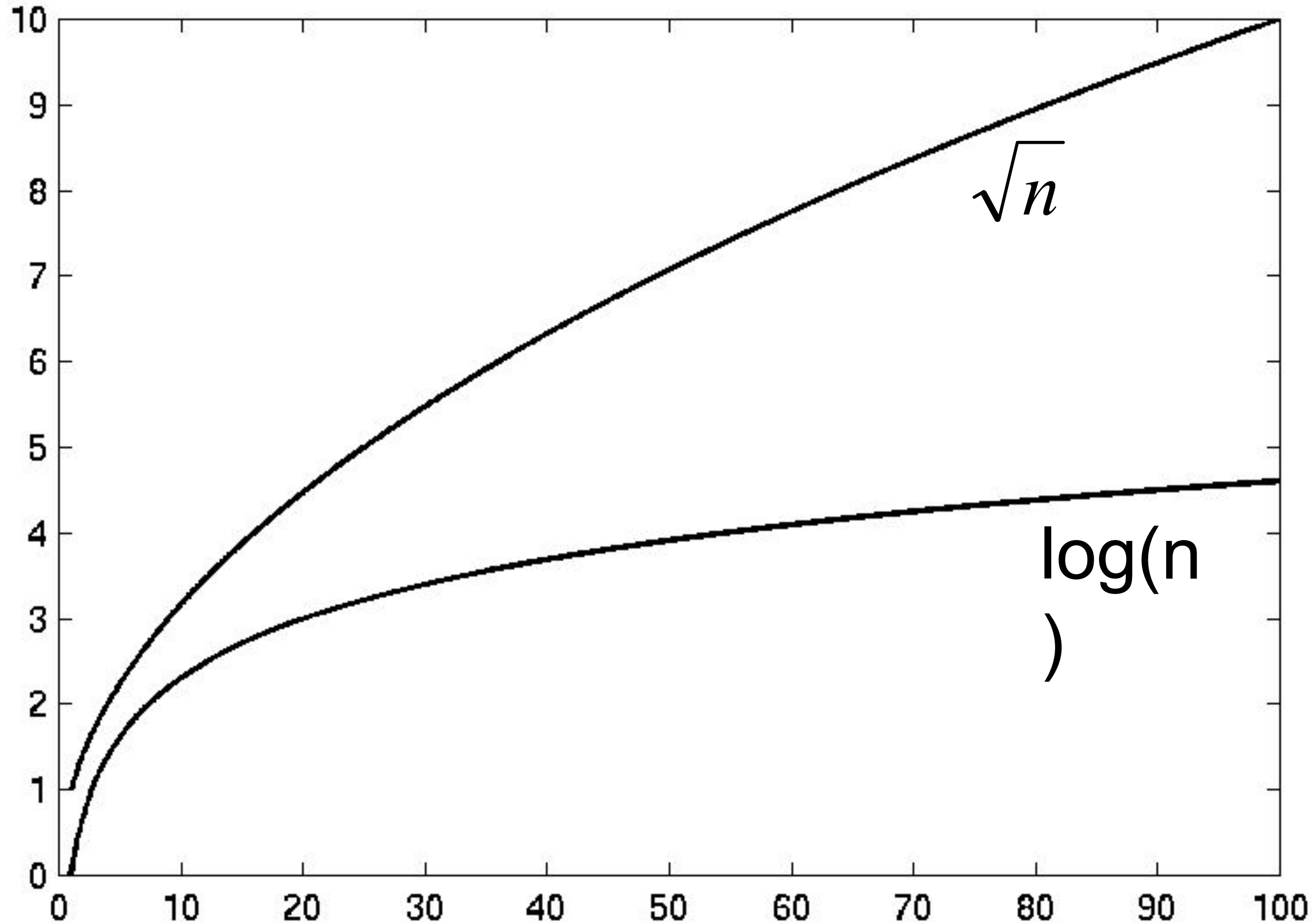
Typical Running Time Functions

- N^2 (quadratic)
  - Typical for algorithms that process all pairs of data items (double nested loops)
- N^3 (cubic)
  - Processing of triples of data (triple nested loops)
- N^K (polynomial)
- 2^N (exponential)
  - Few exponential algorithms are appropriate for practical use

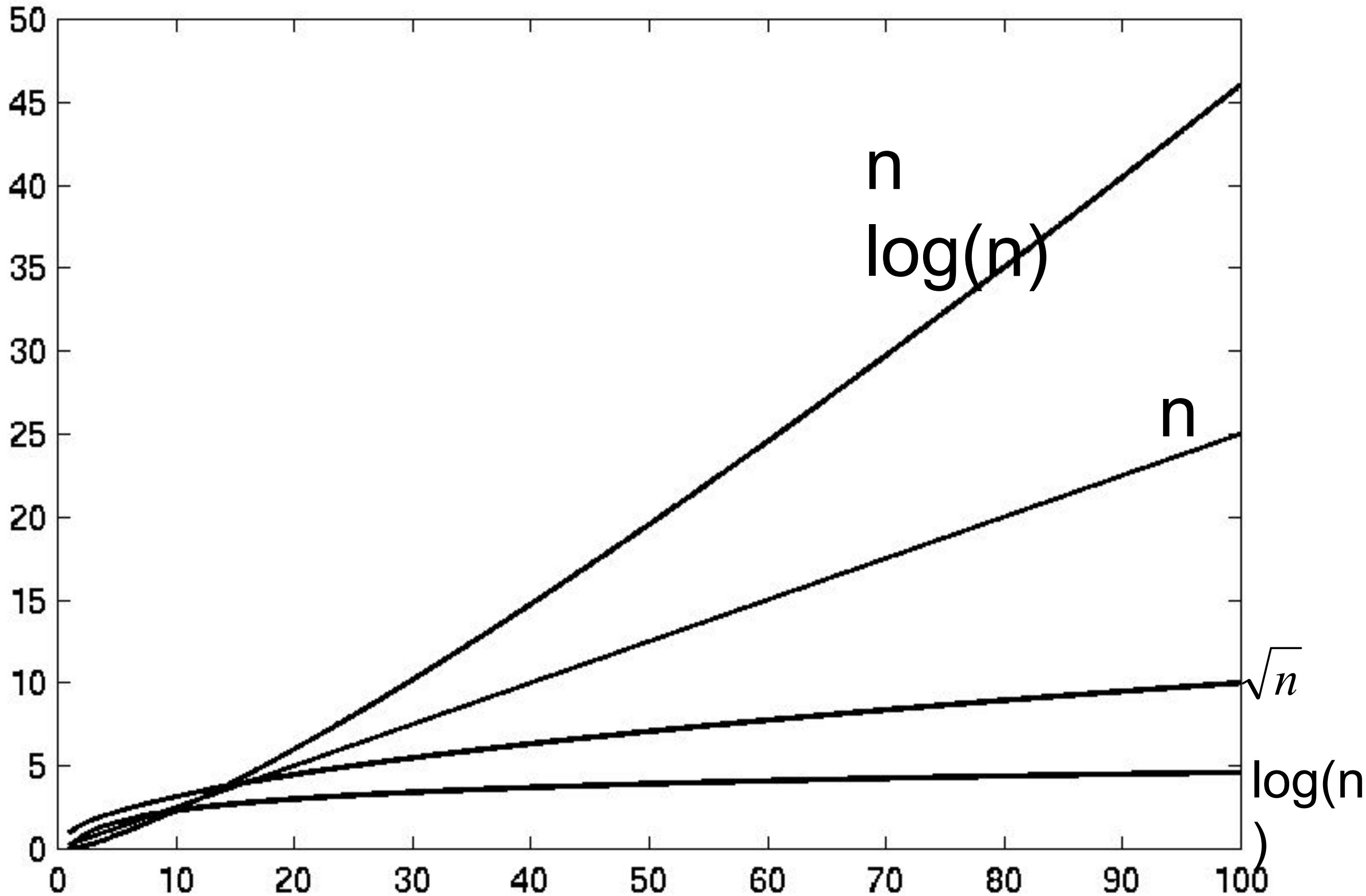
Growth of Functions

n	1	lgn	n	nlgn	n^2	n^3	2^n
1	1	0.00	1	0	1	1	2
10	1	3.32	10	33	100	1,000	1024
100	1	6.64	100	664	10,000	1,000,000	1.2×10^{30}
1000	1	9.97	1000	9970	1,000,000	10^9	1.1×10^{301}

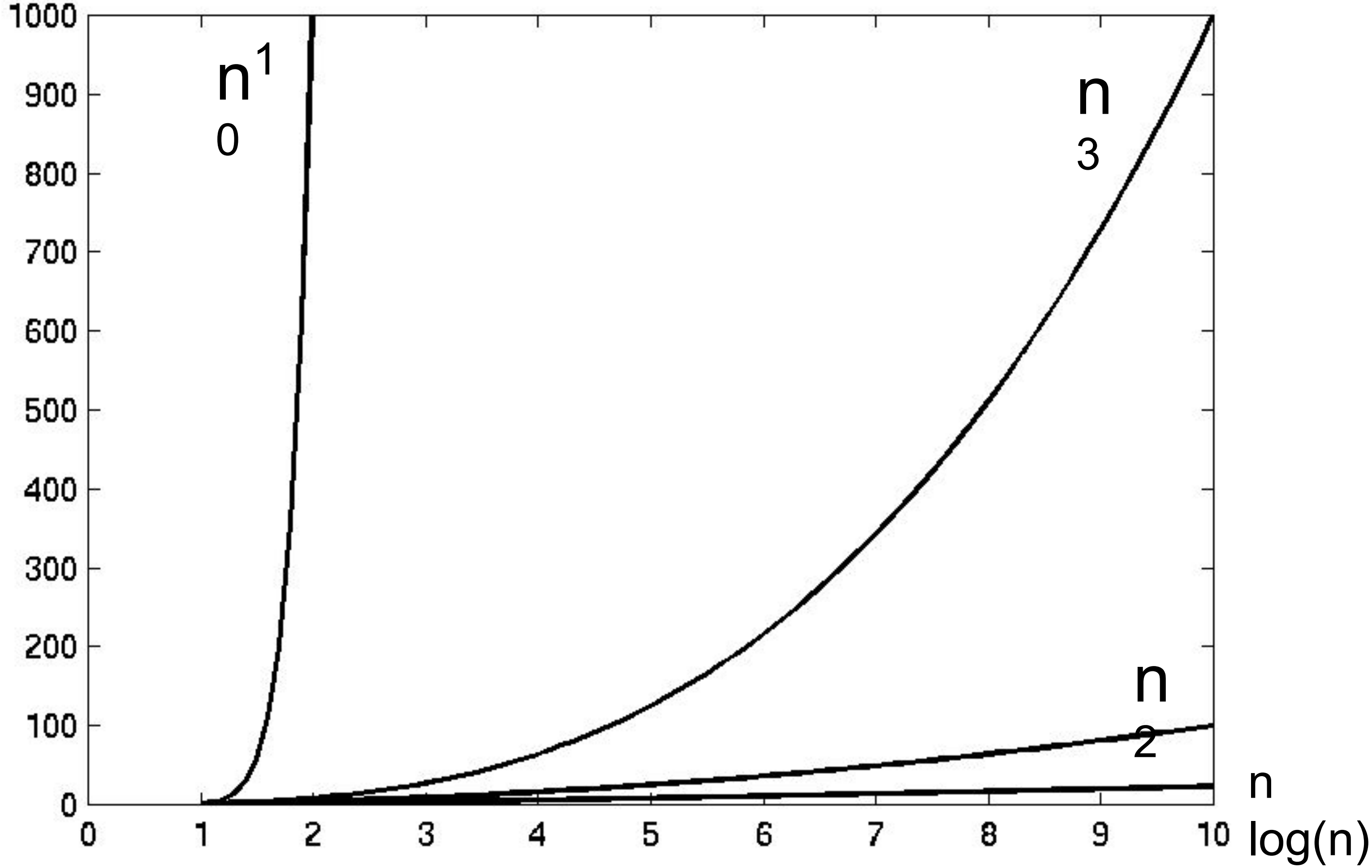
Complexity Graphs



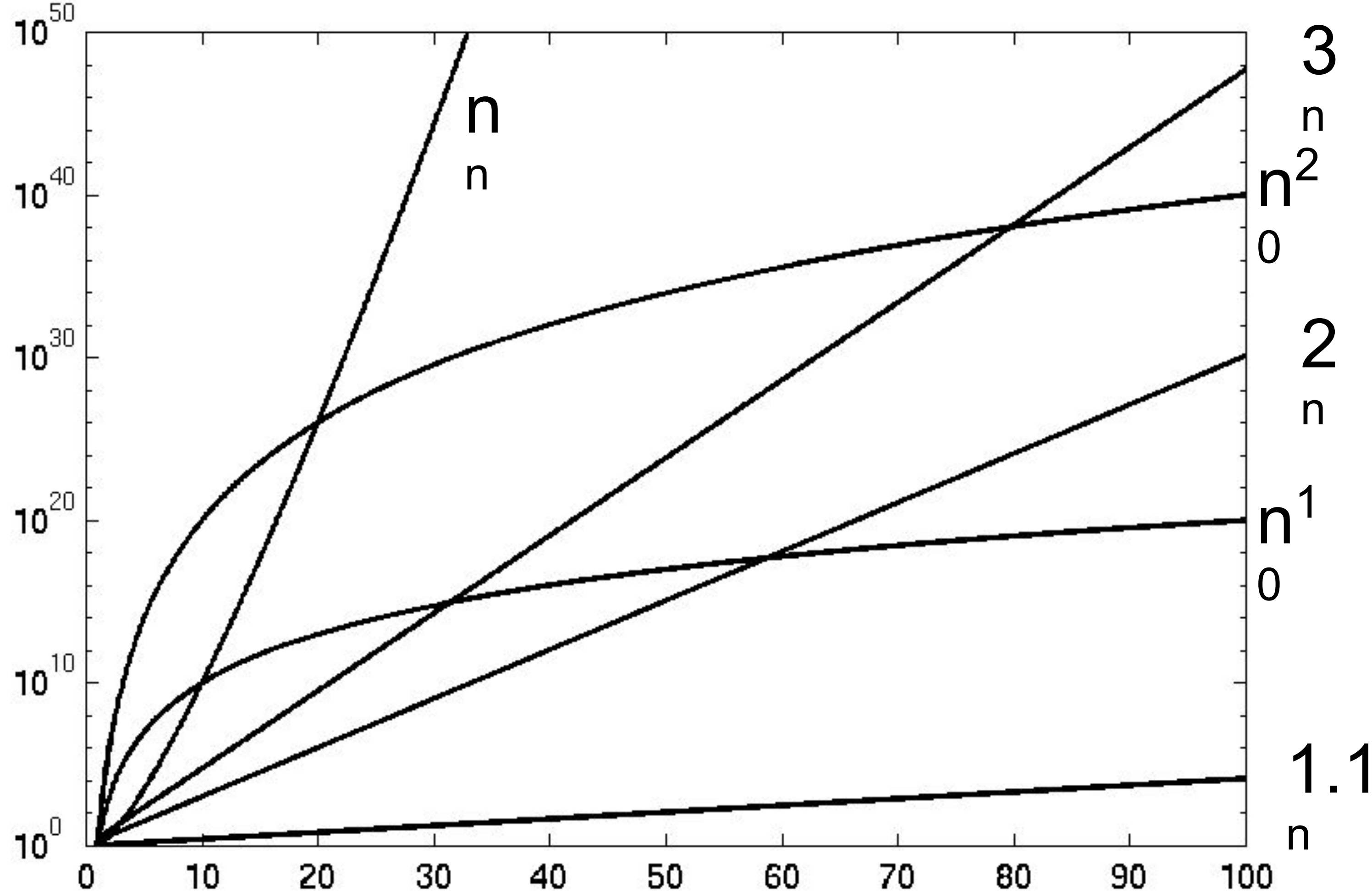
Complexity Graphs



Complexity Graphs



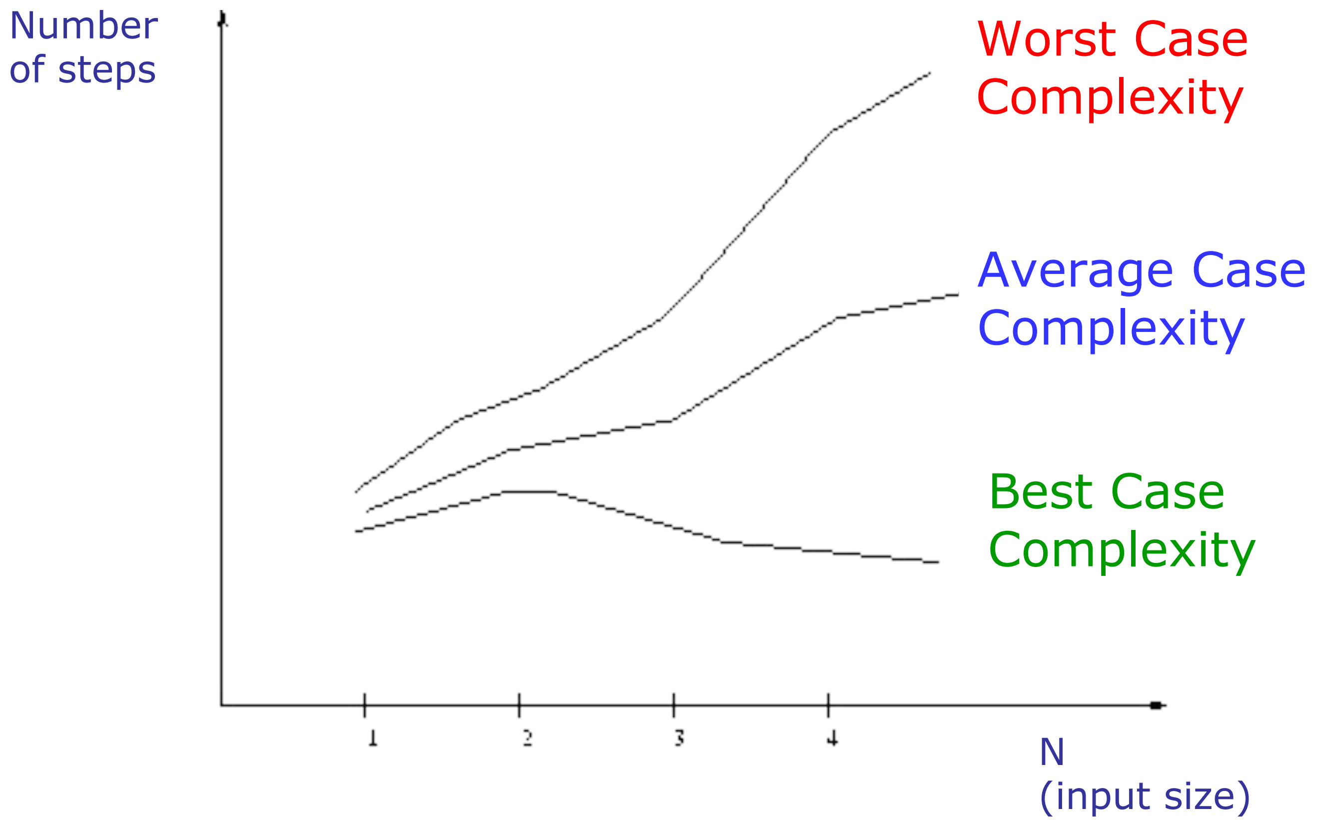
Complexity Graphs (log scale)



Algorithm Complexity

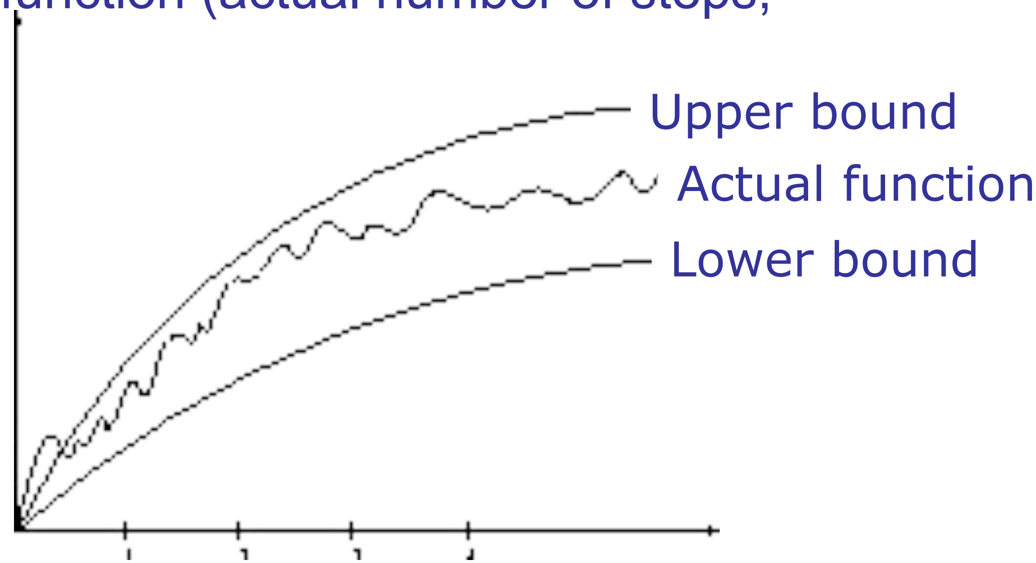
- Worst Case Complexity:
  - the function defined by the *maximum* number of steps taken on any instance of size *n*
- Best Case Complexity:
  - the function defined by the *minimum* number of steps taken on any instance of size *n*
- Average Case Complexity:
  - the function defined by the *average* number of steps taken on any instance of size *n*

Best, Worst, and Average Case Complexity



Doing the Analysis

- It's hard to estimate the running time exactly
  - Best case depends on the input
  - Average case is difficult to compute
  - So we usually focus on worst case analysis
    - Easier to compute
    - Usually close to the actual running time
- Strategy: find a function (an equation) that, for large n, is an upper bound to the actual function (actual number of steps, memory usage, etc.)



Motivation for Asymptotic Analysis

- An *exact computation* of worst-case running time can be difficult
  - Function may have many terms:
    - $4n^2 - 3n \log n + 17.5\, n - 43\, n^{2/3} + 75$
- An *exact computation* of worst-case running time is unnecessary
  - Remember that we are already approximating running time by using RAM model

Classifying functions by their Asymptotic Growth Rates (1/2)

- asymptotic growth rate, asymptotic order, or order of functions
  - Comparing and classifying functions that ignores
    - *constant factors* and
    - *small inputs*.
- The Sets big oh O(g), big theta Θ(g), big omega Ω(g)

Classifying functions by their Asymptotic Growth Rates (2/2)

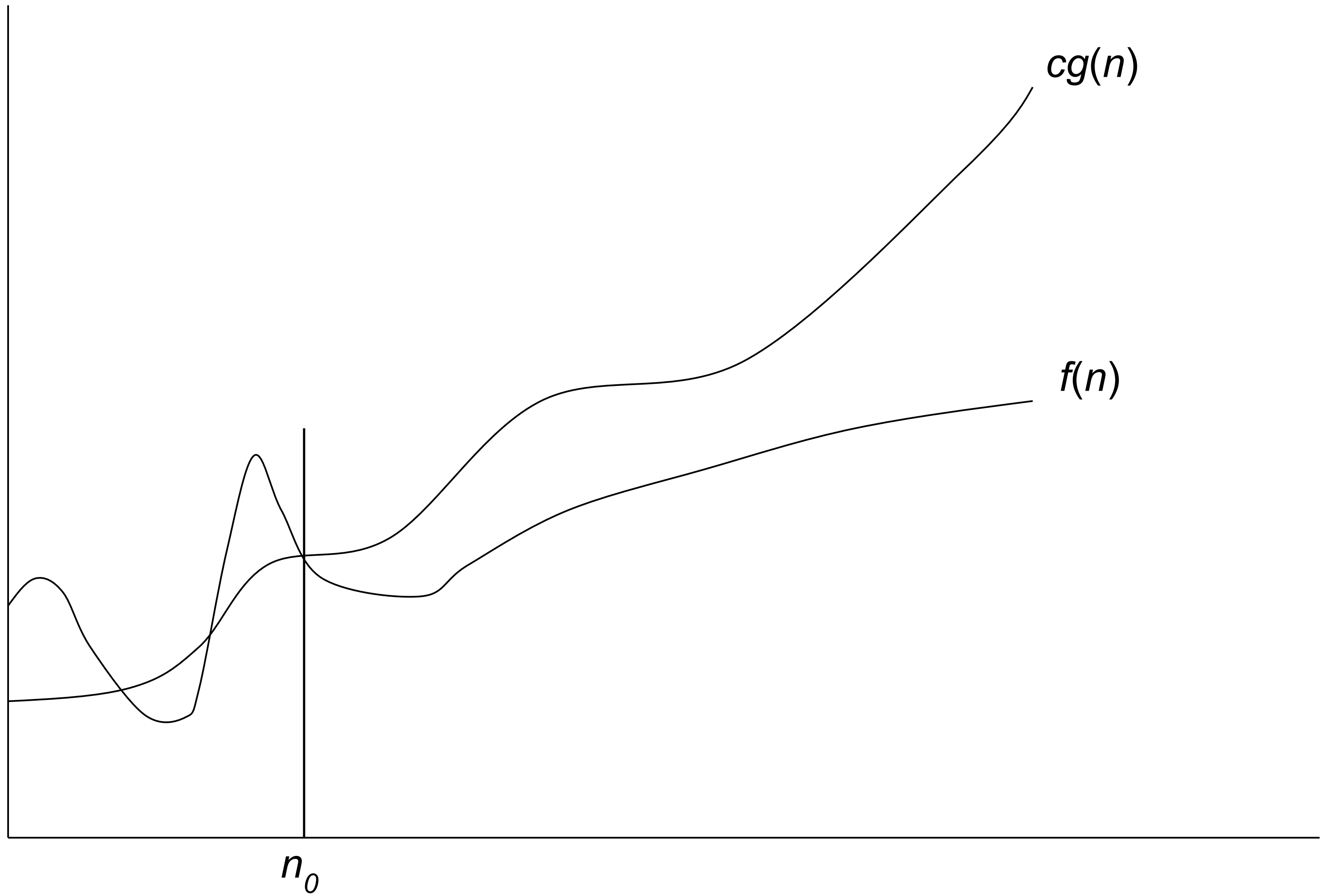
- O(g(n)), Big-Oh of g of n, the Asymptotic Upper Bound;
- Θ(g(n)), Theta of g of n, the Asymptotic Tight Bound; and
- Ω(g(n)), Omega of g of n, the Asymptotic Lower Bound.

Big-O

$$f(n) = O(g(n)) : \text{there exist positive constants } c \text{ and } n_0 \text{ such that}$$
$$0 \le f(n) \le cg(n) \text{ for all } n \ge n_0$$

- What does it mean?
  - If $f(n) = O(n^2)$, then:
    - $f(n)$ can be larger than n^2 sometimes, **but…**
    - We can choose some constant c and some value n_0 such that for **every** value of n larger than n_0 : $f(n) < cn^2$
    - That is, for values larger than n_0, $f(n)$ is never more than a constant multiplier greater than n^2
    - Or, in other words, $f(n)$ does not grow more than a constant factor faster than n^2.

Visualization of $O(g(n))$



Examples

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

  $2n^2 \le cn^3 \Rightarrow 2 \le cn \Rightarrow c = 1$ and $n_0 = 2$

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

  $n^2 \le cn^2 \Rightarrow c \ge 1 \Rightarrow c = 1$ and $n_0 = 1$

- $1000n^2+1000n = O(n^2)$:

$1000n^2+1000n \le cn^2 \le cn^2+ 1000n \Rightarrow c=1001$ and $n_0 = 1$

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

  $n \le cn^2 \Rightarrow cn \ge 1 \Rightarrow c = 1$ and $n_0 = 1$

Big-O

$$2n^2 = O\left(n^2\right)$$

$$1{,}000{,}000n^2 + 150{,}000 = O\left(n^2\right)$$

$$5n^2 + 7n + 20 = O\left(n^2\right)$$

$$2n^3 + 2 \neq O\left(n^2\right)$$

$$n^{2.1} \neq O\left(n^2\right)$$

More Big-O

- Prove that: $20n^2 + 2n + 5 = O\left(n^2\right)$
- Let c = 21 and n_0 = 4
- $21n^2 > 20n^2 + 2n + 5$ for all $n > 4$

  $n^2 > 2n + 5$ for all $n > 4$

  TRUE

Tight bounds

- We generally want the tightest bound we can find.
- While it is true that $n^2 + 7n$ is in $O(n^3)$, it is more interesting to say that it is in $O(n^2)$

Big Omega – Notation

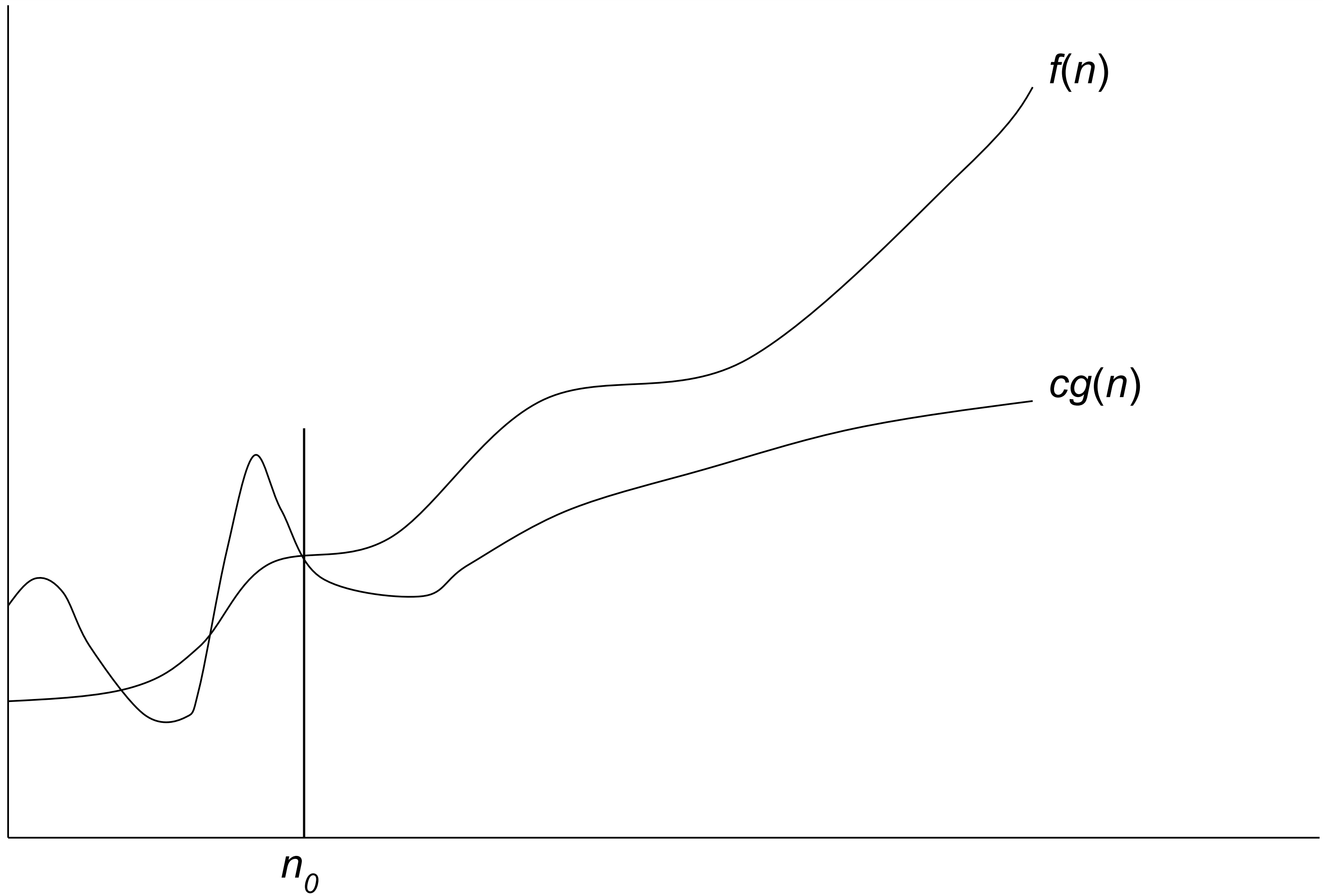
- $\Omega()$ – A **lower** bound

$f(n) = \Omega(g(n))$: there exist positive constants c and n_0 such that

$$0 \le f(n) \ge cg(n) \text{ for all } n \ge n_0$$

  - $n^2 = \Omega(n)$
  - Let $c = 1$, $n_0 = 2$
  - For all $n \geq 2$, $n^2 > 1 \times n$

Visualization of $\Omega(g(n))$



Θ-notation

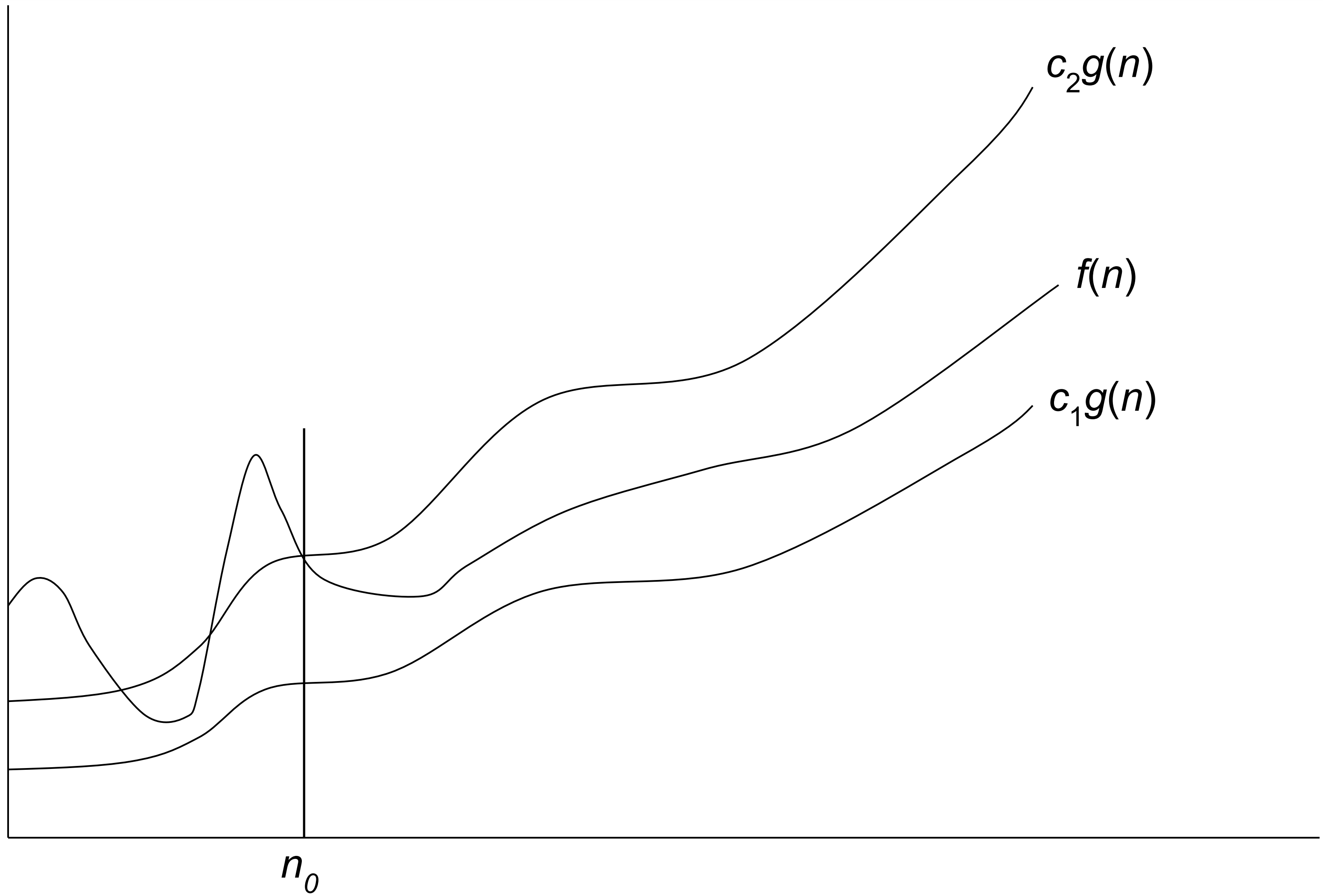
- Big-*O* is not a tight upper bound. In other words $n = O(n^2)$
- Θ provides a tight bound

$$f(n) = \Theta(g(n)) : \text{there exist positive constants } c_1, c_2, \text{and } n_0 \text{ such that}$$
$$0 \le c_1 g(n) \le f(n) \le c_2 g(n) \text{ for all } n \ge n_0$$

- In other words,

$$f(n) = \Theta(g(n)) \Rightarrow f(n) = O(g(n)) \text{ AND } f(n) = \Omega(g(n))$$

Visualization of $\Theta(g(n))$



A Few More Examples

- $n = O(n^2) \neq \Theta(n^2)$
- $200n^2 = O(n^2) = \Theta(n^2)$
- $n^{2.5} \neq O(n^2) \neq \Theta(n^2)$

Example 2

- Prove that: $20n^3 + 7n + 1000 = \Theta\left(n^3\right)$
- Let c = 21 and n_0 = 10
- $21n^3 > 20n^3 + 7n + 1000$ for all $n > 10$

  $n^3 > 7n + 5$ for all $n > 10$

  TRUE, but we also need…
- Let c = 20 and n_0 = 1
- $20n^3 < 20n^3 + 7n + 1000$ for all $n \geq 1$

  TRUE

Example 3

- Show that $2^n + n^2 = O(2^n)$
- Let $c = 2$ and $n_0 = 5$

$$2 \times 2^n > 2^n + n^2$$

$$2^{n+1} > 2^n + n^2$$

$$2^{n+1} - 2^n > n^2$$

$$2^n(2-1) > n^2$$

$$2^n > n^2 \quad \forall n \geq 5 \quad ✔$$

Asymptotic Notations - Examples

- Θ notation
  - $n^2/2 - n/2$ $= \Theta(n^2)$
  - $(6n^3 + 1)\lg n/(n + 1)$ $= \Theta(n^2 \lg n)$
  - n vs. n^2 $n \neq \Theta(n^2)$
- Ω notation
  - n^3 vs. n^2 $n^3 = \Omega(n^2)$
  - n vs. $\log n$ $n = \Omega(\log n)$
  - n vs. n^2 $n \neq \Omega(n^2)$
- O notation
  - $2n^2$ vs. n^3 $2n^2 = O(n^3)$
  - n^2 vs. n^2 $n^2 = O(n^2)$
  - n^3 vs. $n\log n$ $n^3 \neq O(n\lg n)$

Asymptotic Notations - Examples

- For each of the following pairs of functions, either f(n) is O(g(n)), f(n) is Ω(g(n)), or f(n) = Θ(g(n)). Determine which relationship is correct.

– $f(n) = \log n^2$; $g(n) = \log n + 5$	$f(n) = \Theta(g(n))$
– $f(n) = n$; $g(n) = \log n^2$	$f(n) = \Omega(g(n))$
– $f(n) = \log \log n$; $g(n) = \log n$	$f(n) = O(g(n))$
– $f(n) = n$; $g(n) = \log^2 n$	$f(n) = \Omega(g(n))$
– $f(n) = n \log n + n$; $g(n) = \log n$	$f(n) = \Omega(g(n))$
– $f(n) = 10$; $g(n) = \log 10$	$f(n) = \Theta(g(n))$
– $f(n) = 2^n$; $g(n) = 10n^2$	$f(n) = \Omega(g(n))$
– $f(n) = 2^n$; $g(n) = 3^n$	$f(n) = O(g(n))$

Simplifying Assumptions

- 1. If $f(n) = O(g(n))$ and $g(n) = O(h(n))$, then $f(n) = O(h(n))$
- 2. If $f(n) = O(kg(n))$ for any $k > 0$, then $f(n) = O(g(n))$
- 3. If $f_1(n) = O(g_1(n))$ and $f_2(n) = O(g_2(n))$,
- then $f_1(n) + f_2(n) = O(\max(g_1(n), g_2(n)))$
- 4. If $f_1(n) = O(g_1(n))$ and $f_2(n) = O(g_2(n))$,
- then $f_1(n) * f_2(n) = O(g_1(n) * g_2(n))$

Example

- Code:
- `a = b;`

- Complexity:

Example

- Code:
- `sum = 0;`
- `for (i=1; i <=n; i++)`
- `    sum += n;`

- Complexity:

Example

- Code:

```
 sum = 0;
 for (j=1; j<=n; j++)
     for (i=1; i<=j; i++)
         sum++;
 for (k=0; k<n; k++)
     A[k] = k;
```

- Complexity:

Example

- Code:
- `sum1 = 0;`
- `for (i=1; i<=n; i++)`
- `   for (j=1; j<=n; j++)`
- `      sum1++;`
- Complexity:

Example

- Code:
- `sum2 = 0;`
- `for (i=1; i<=n; i++)`
- `   for (j=1; j<=i; j++)`
- `      sum2++;`
- Complexity:

Example

- Code:
- `sum1 = 0;`
- `for (k=1; k<=n; k*=2)`
- `   for (j=1; j<=n; j++)`
- `      sum1++;`
- Complexity:

Example

- Code:
- `sum2 = 0;`
- `for (k=1; k<=n; k*=2)`
- `   for (j=1; j<=k; j++)`
- `      sum2++;`
- Complexity:

Recurrences

Def.: Recurrence = an equation or inequality that describes a function in terms of its value on smaller inputs, and one or more base cases

- E.g.: $T(n) = T(n-1) + n$
- Useful for analyzing recurrent algorithms
- Methods for solving recurrences
  - Substitution method
  - Recursion tree method
  - Master method
  - Iteration method