CSE-221
Design & Analysis of Algorithm

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

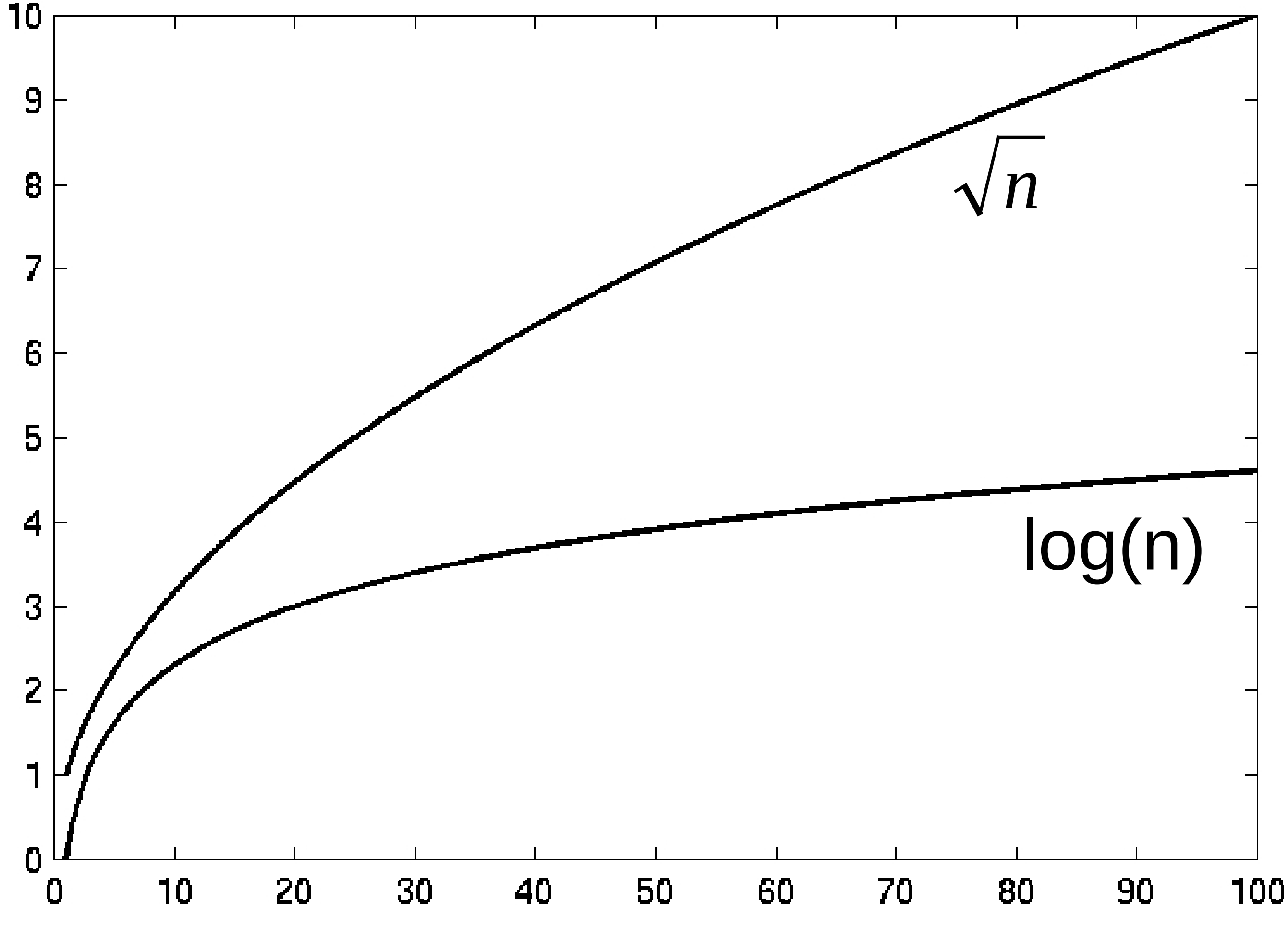
Typical Running Time Functions

- 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

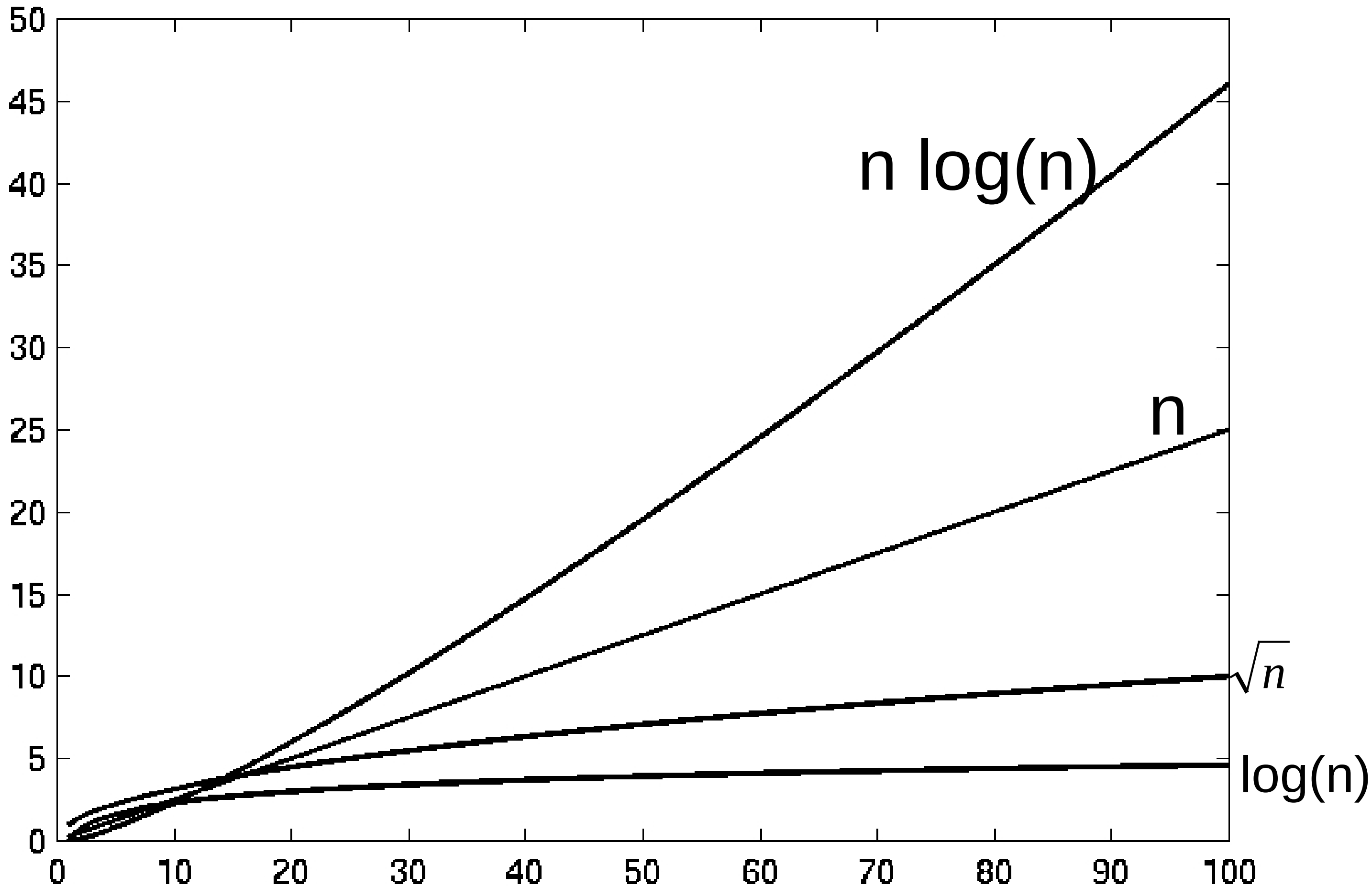
Growth of Functions

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}

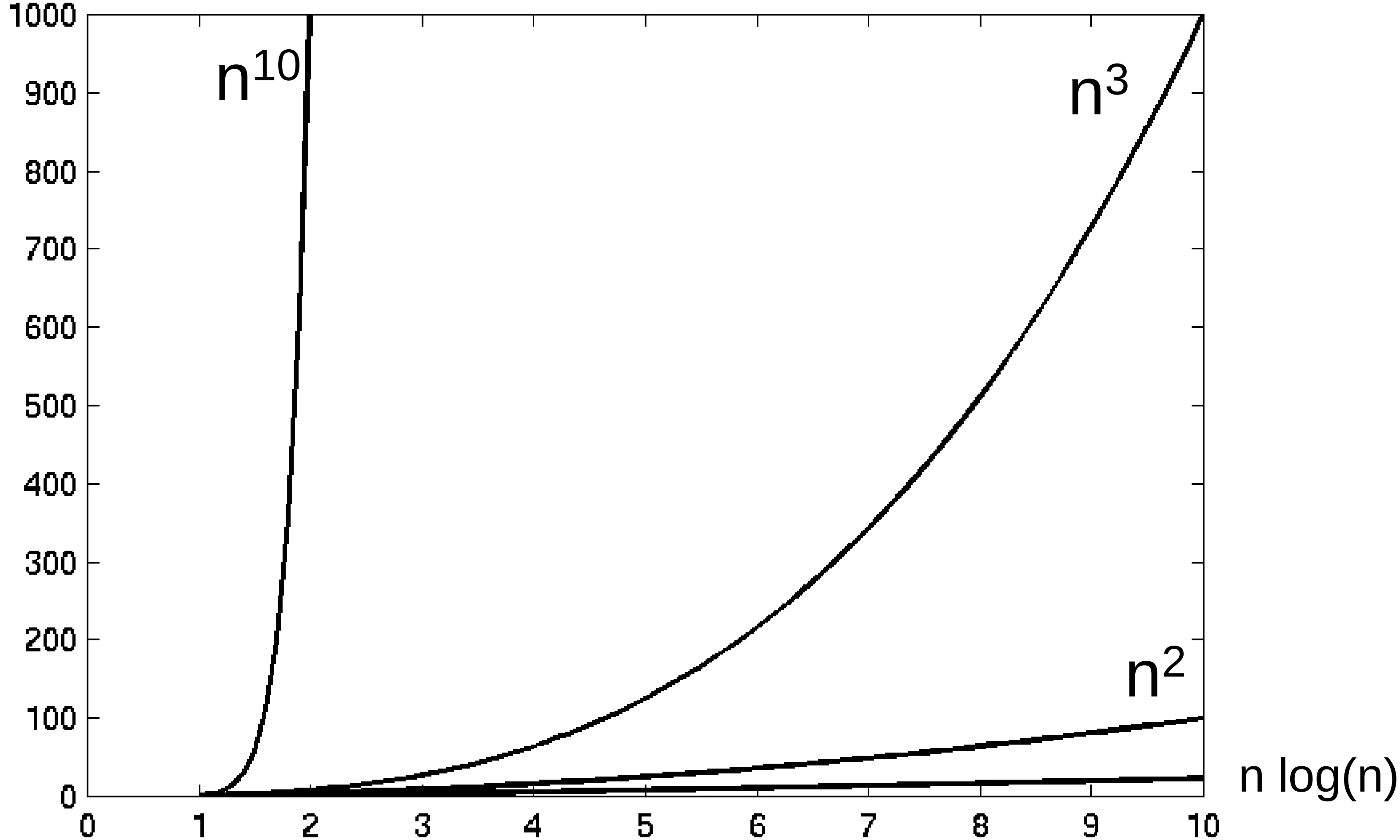
Complexity Graphs



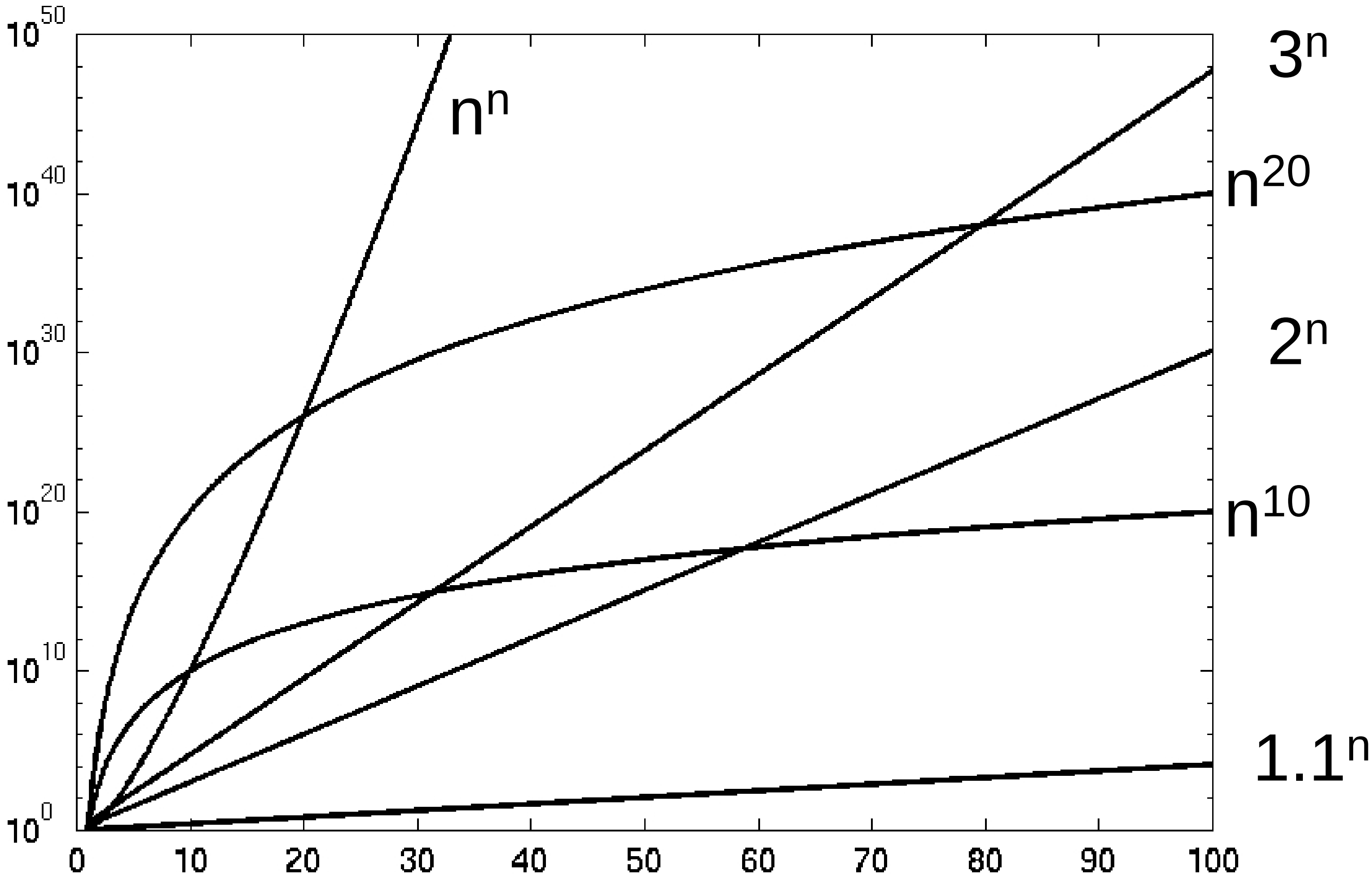
Complexity Graphs



Complexity Graphs



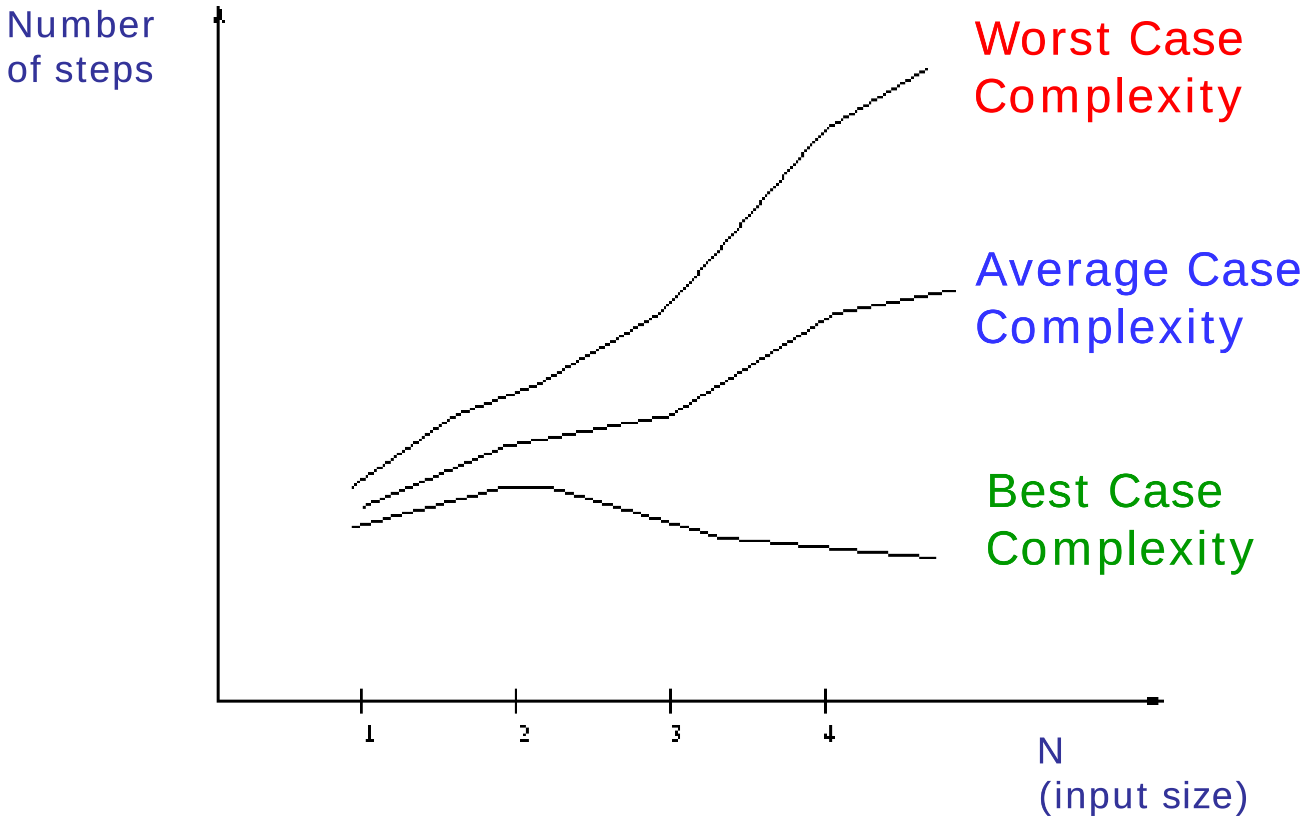
Complexity Graphs (log scale)



Algorithm Complexity

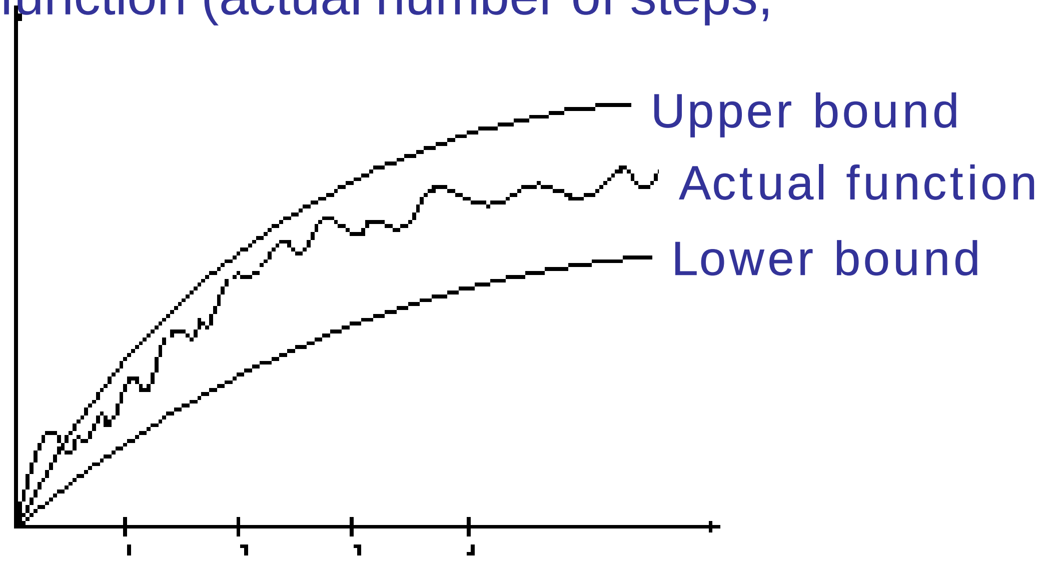
- Worst Case Complexity:
  - the function defined by the *maximum* number of steps taken on any instance of size *n*
- Best Case Complexity:
  - the function defined by the *minimum* number of steps taken on any instance of size *n*
- Average Case Complexity:
  - the function defined by the *average*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



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 $\Theta(g)$, big omega $\Omega(g)$

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

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

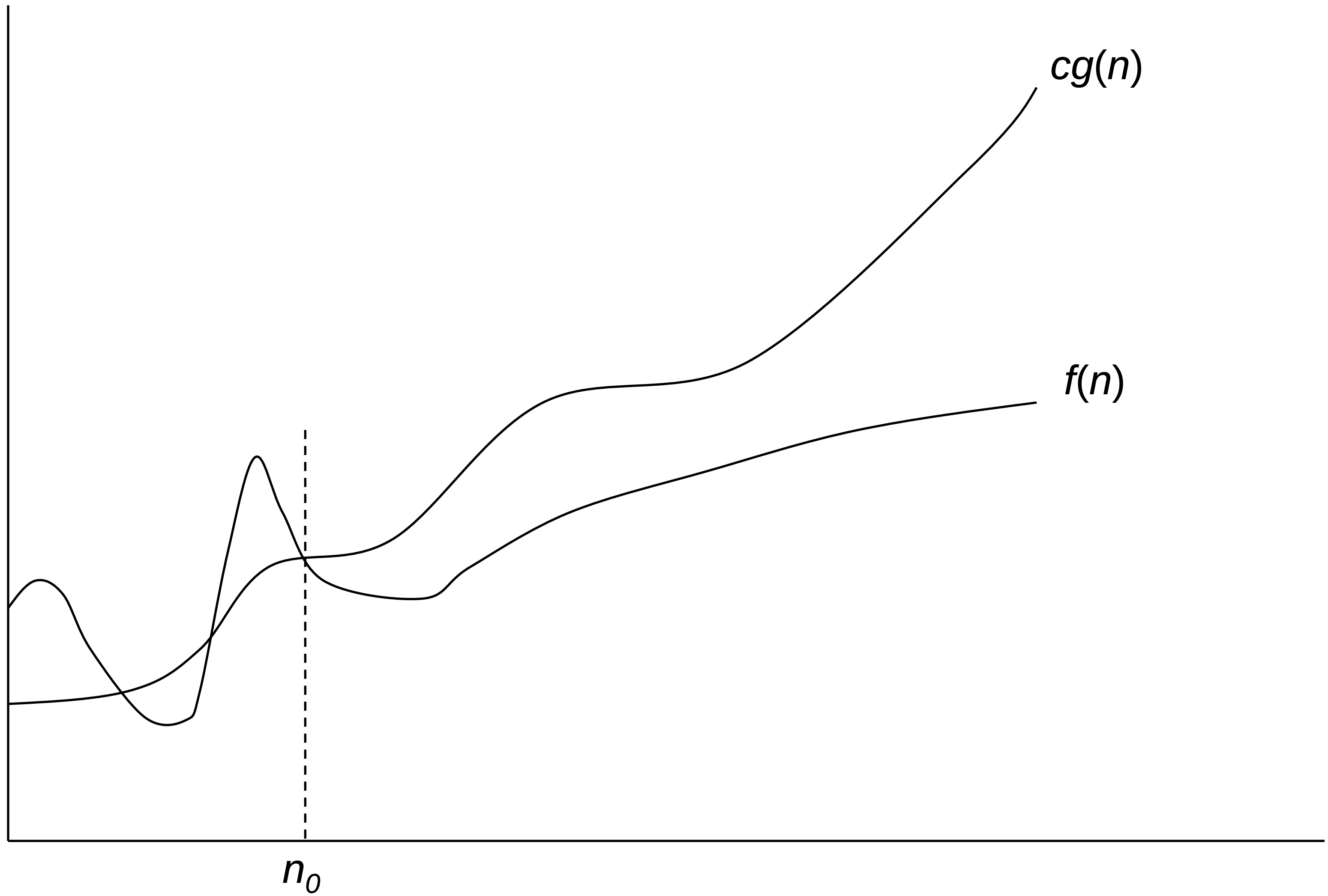
Big-O

$f(n) = O(g(n))$: there exist positive constants c and n_0 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 $O(g(n))$



Big-O

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

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

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

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

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

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

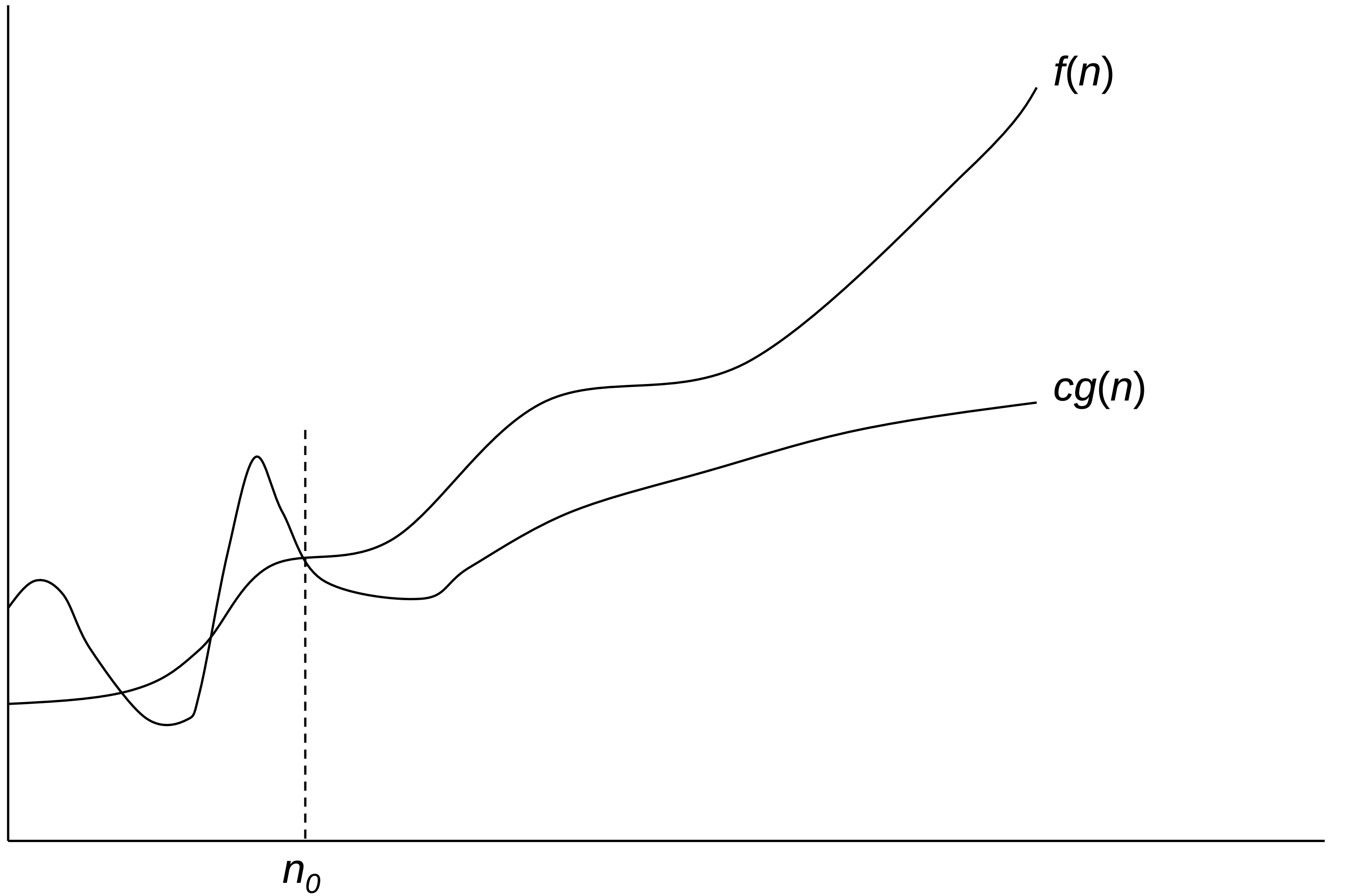
- $\Omega()$ – A **lower** bound

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

$$0 \leq f(n) \geq cg(n) \text{ for all } n \geq n_0$$

  - $n^2 = \Omega(n)$
  - 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

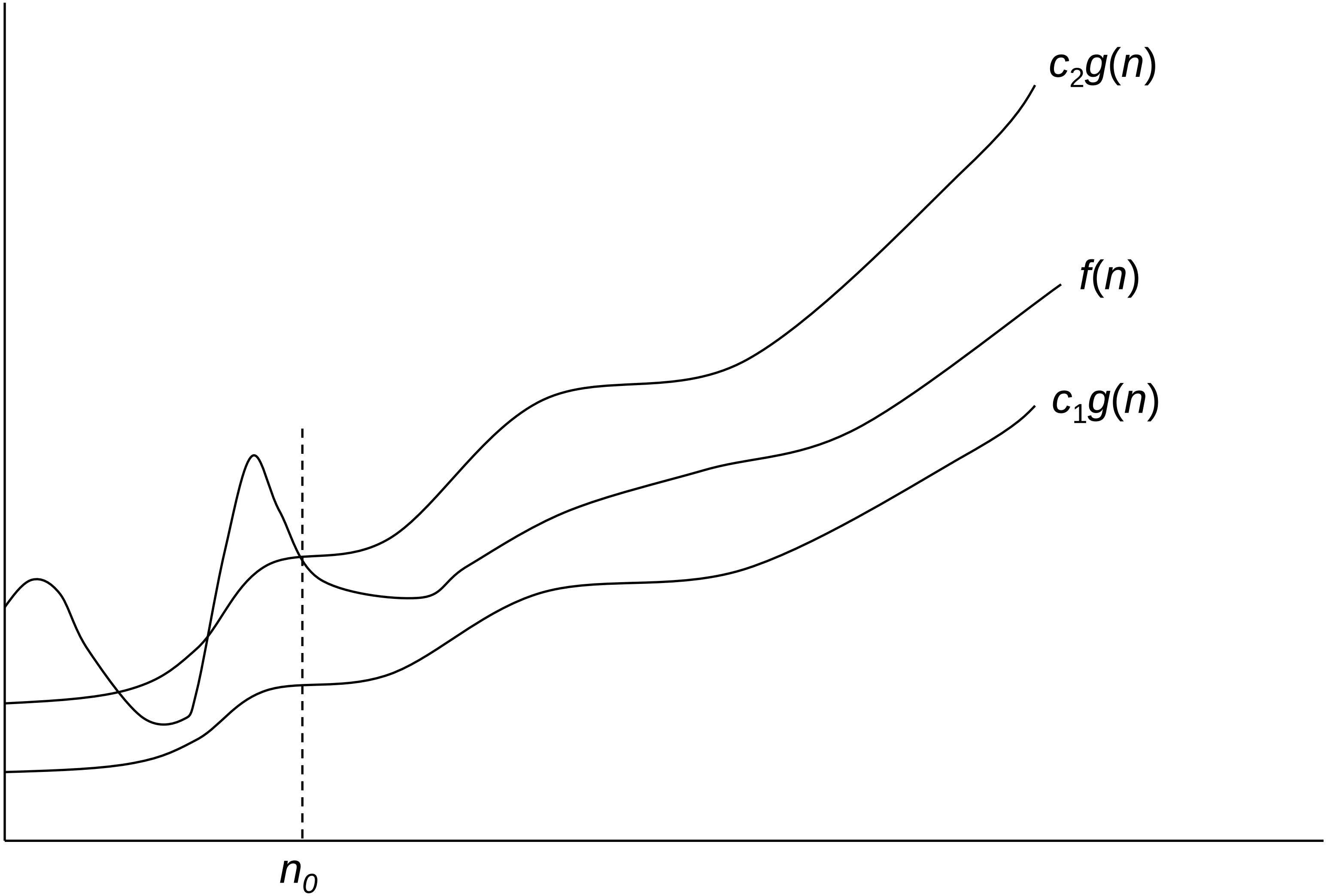
$f(n) = \Theta(g(n))$: there exist positive constants c_1, c_2, and n_0 such that

$$0 \le c_1 g(n) \le f(n) \le c_2 g(n) \text{ for all } n \ge n_0$$

- In other words,

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

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



Example

- Code:
- `a = b;`

- Complexity:

Example

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

```
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: