# Analysis of Algorithms & Orders of Growth

Revised for use by Midwestern State University

# Analysis of Algorithms

An ***algorithm*** is a finite set of precise instructions for performing a computation or for solving a problem.

What is the goal of analysis of algorithms?

To compare algorithms mainly in terms of running time but also in terms of other factors (e.g., memory requirements, programmer's effort etc.)

What do we mean by runtime analysis?

**Determine how running time increases as the size of the problem increases**.

# Example: Searching

Problem of *searching an ordered list*.

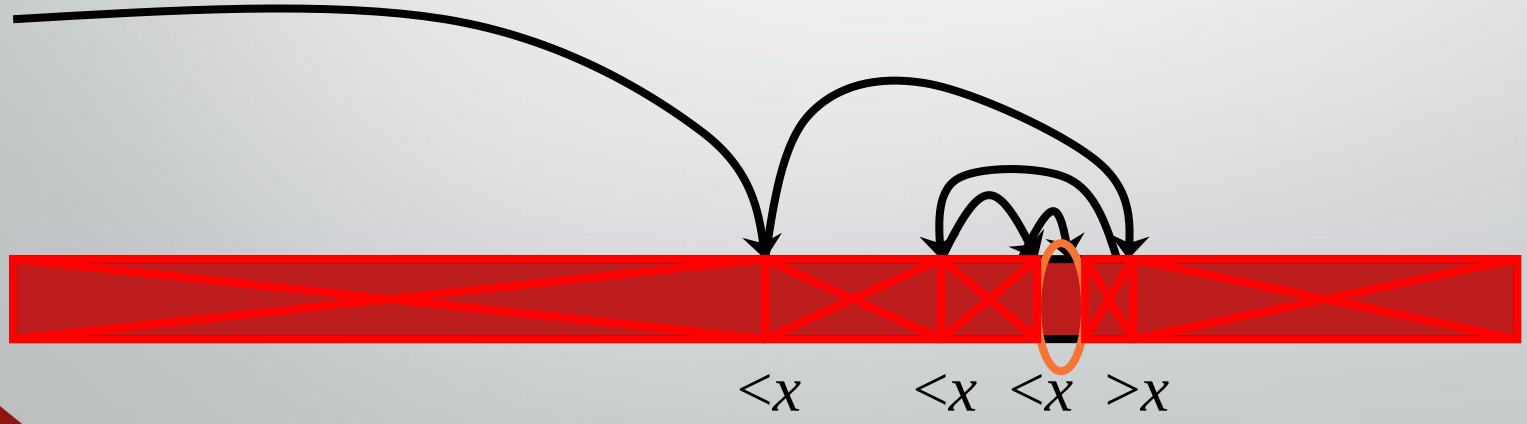
- Given a list *L* of *n* elements that are sorted into a definite order (*e.g.*, numeric, alphabetical),
- And given a particular element *x*,
- Determine whether *x* appears in the list, and if so, return its index (position) in the list.

# Linear Search Algorithm

```
Set found to false
Set position to –1
Set index to 0
    While index < number of elts and found is false
        If list [index] is equal to search value
            found = true
            position = index
        End If
        Add 1 to index
End While
Return position
```

# Search alg. #2: Binary Search

Basic idea: On each step, look at the *middle* element of the remaining list to eliminate half of it, and quickly zero in on the desired element.



# Binary Search Algorithm

1. Divide a sorted array into three sections:
   - middle element
   - elements on one side of the middle element
   - elements on the other side of the middle element
2. If the middle element is the correct value, done. Otherwise, go to step 1, using only the half of the array that may contain the correct value.
3. Continue steps 1 and 2 until either the value is found or there are no more elements to examine.

# Is Binary Search more efficient?

**Number of iterations**:

- For a list of *n* elements, Binary Search can execute at most $\log_2 n$ times!!
- Linear Search, on the other hand, can execute up to *n* times !!

Average Number of Iterations*

| Length | Linear Search | Binary Search |
|---|---|---|
| 10 | 5.5 | 2.9 |
| 100 | 50.5 | 5.8 |
| 1,000 | 500.5 | 9.0 |
| 10,000 | 5000.5 | 12.0 |

*Assuming each position as well as not found are all equally likely to occur.

# Is Binary Search more efficient?

**Number of computations per iteration**:

- Binary search does more computations than Linear Search per iteration.

**Overall**:

- If the number of components is small (say, less than 20), then Linear Search is faster.
- If the number of components is large, then Binary Search is faster.

# How do we analyze algorithms?

We need to define a number of objective measures.

(1) Compare execution times?

***Not good***: times are specific to a particular computer !!

(2) Count the number of statements executed?

***Not good***: number of statements vary with the programming language as well as the style of the individual programmer.

# Example (# of statements)

**Algorithm 1**

```
arr[0] = 0;
arr[1] = 0;
arr[2] = 0;
...
arr[N-1] = 0;
```

**Algorithm 2**

```
for(i=0; i<N; i++)
  arr[i] = 0;
```

# How do we analyze algorithms?

(3) Express efficiency as a function of the input size $n$ (i.e., $f(n)$)

- To compare two algorithms with running times $f(n)$ and $g(n)$, we need a **rough measure** of **how fast a function grows**.
- Such an analysis is independent of machine time, programming style, etc.

# Computing running time

- Associate a "cost" with each statement and find the "total cost" by finding the total number of times each statement is executed.
- Express running time in terms of the size of the problem.

***Algorithm 1***

| | Cost |
|---|---|
| arr[0] = 0; | c1 |
| arr[1] = 0; | c1 |
| arr[2] = 0; | c1 |
| ... | |
| arr[N-1] = 0; | c1 |
| | ---------- |
| c1+c1+...+c1 = c1 x N | |

***Algorithm 2***

| | Cost |
|---|---|
| for(i=0; i<N; i++) | c2 |
| arr[i] = 0; | c1 |
| | ---------- |
| (N+1) x c2 + N x c1 = (c2 + c1) x N + c2 | |

# Computing running time (cont.)

| | *Cost* |
|---|---|
| sum = 0; | c1 |
| for(i=0; i<N; i++) | c2 |
| for(j=0; j<N; j++) | c2 |
| sum += arr[i][j]; | c3 |

---

$$c1 + c2 \times (N+1) + c2 \times N \times (N+1) + c3 \times N \times N$$

# Calculate each polynomial for n =1, n = 2, n = 10 and n = 50

$f(n) = 45n^3$ $f(n) = 45n^3 + 20n^2 + 19$

| | $f(n) = 45n^3$ | $f(n) = 45n^3 + 20n^2 + 19$ |
|---|---|---|
| f(1) | 45 | 84 |
| f(2) | 360 | 459 |
| f(10) | 45,000 | 47,019 |
| f(50) | 5,625,000 | 5,675,019 |
| f(1,000) | 45,000,000,000 | 45,020,000,019 |

# Comparing Functions Using Rate of Growth

- Consider the example of buying *elephants* and *goldfish:*

  **Cost**: cost_of_elephants + cost_of_goldfish

  **Cost** ~ cost_of_elephants (**approximation**)

- The low order terms in a function are relatively insignificant for **large** *n*

$$n^4 + 100n^2 + 10n + 50 \quad \sim \quad n^4$$

*i.e.*, $n^4 + 100n^2 + 10n + 50$ and $n^4$ have the same rate of growth

# Rate of Growth ≡ Asymptotic Analysis

- Using *rate of growth* as a measure to compare different functions implies comparing them **asymptotically**.
- If $g(n)$ is *faster growing* than $f(n)$, then $g(n)$ always eventually becomes larger than $f(n)$ **in the limit** (for large enough values of $n$).

# Example

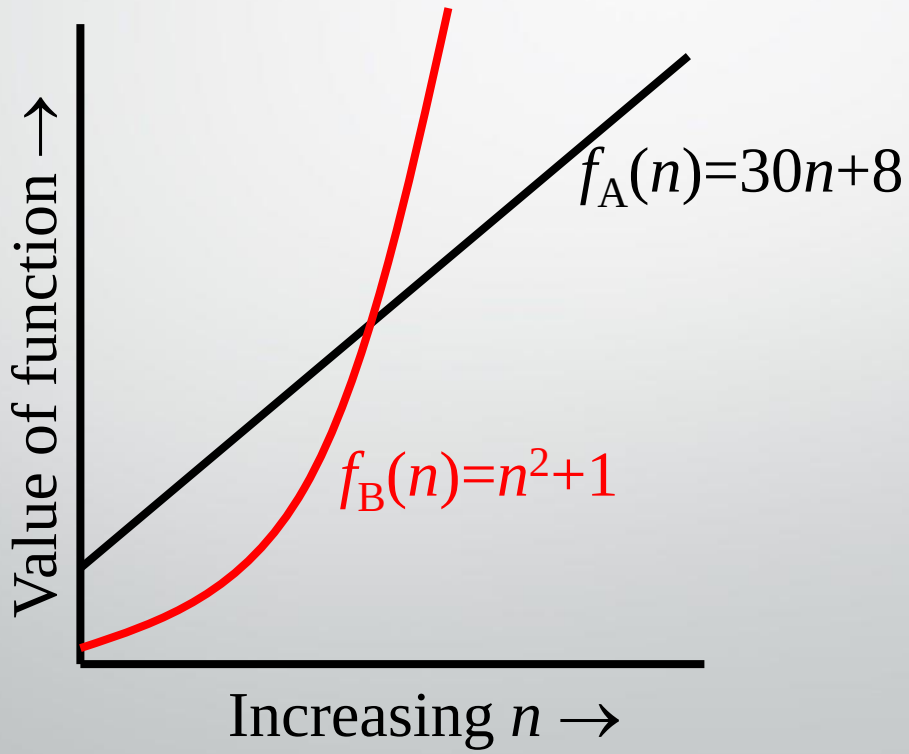
Suppose you are designing a web site to process user data (*e.g.*, financial records).

Suppose program A takes $f_A(n)=30n+8$ microseconds to process any $n$ records, while program B takes $f_B(n)=n^2+1$ microseconds to process the $n$ records.

Which program would you choose, knowing you'll want to support millions of users?

# Visualizing Orders of Growth

On a graph, as you go to the right, a faster growing function eventually becomes larger…



# Big-O Notation

- We say $f_A(n)=30n+8$ is *order n*, or $O(n)$.
  It is, **at most**, roughly *proportional* to $n$.

- $f_B(n)=n^2+1$ is *order* $n^2$, or $O(n^2)$. It is, **at most**, roughly proportional to $n^2$.

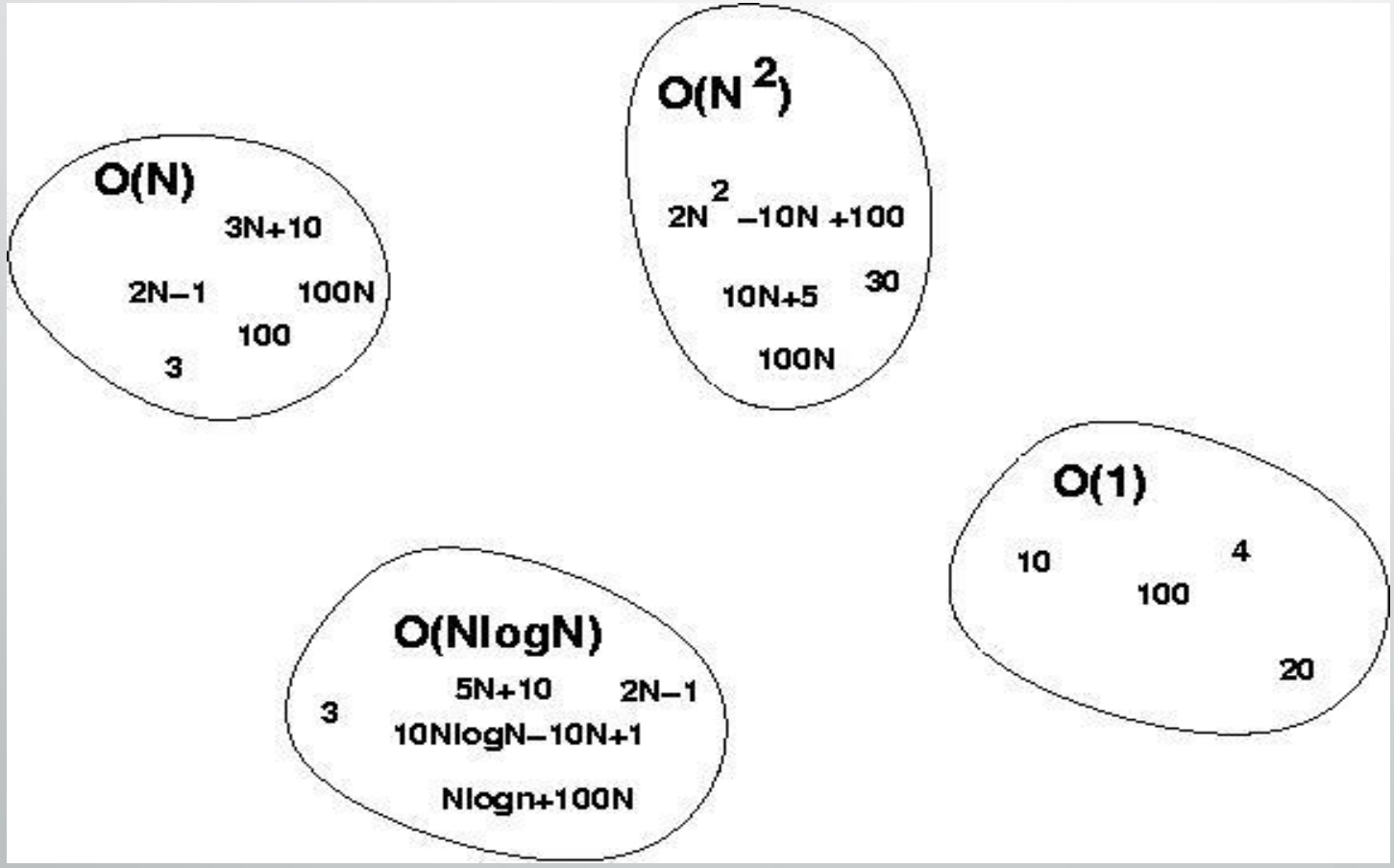
- In general, an $O(n^2)$ algorithm will be slower than $O(n)$ algorithm.

- **Warning**: an $O(n^2)$ function will grow faster than an $O(n)$ function.

# More Examples …

- We say that $n^4 + 100n^2 + 10n + 50$ is of the order of $n^4$ or $O(n^4)$
- We say that $10n^3 + 2n^2$ is $O(n^3)$
- We say that $n^3 - n^2$ is $O(n^3)$
- We say that 10 is $O(1)$,
- We say that 1273 is $O(1)$

# Big-O Visualization



# Computing running time

***Algorithm 1***

| | Cost |
|---|---|
| arr[0] = 0; | c1 |
| arr[1] = 0; | c1 |
| arr[2] = 0; | c1 |
| ... | |
| arr[N-1] = 0; | c1 |

c1+c1+...+c1 = c1 x N

***Algorithm 2***

| | Cost |
|---|---|
| for(i=0; i<N; i++) | c2 |
| arr[i] = 0; | c1 |

(N+1) x c2 + N x c1 =
(c2 + c1) x N + c2

$O(n)$

# Computing running time (cont.)

| | *Cost* |
|---|---|
| sum = 0; | c1 |
| for(i=0; i<N; i++) | c2 |
| for(j=0; j<N; j++) | c2 |
| sum += arr[i][j]; | c3 |
| | ---------- |

$c1 + c2 \times (N+1) + c2 \times N \times (N+1) + c3 \times N \times N$

$O(n^2)$

# Examples

```
i = 0;
while (i<N) {
   X=X+Y;                    // O(1)
   result = mystery(X);      // O(N), just an example...
   i++;                      // O(1)
}
```

- The body of the while loop: O(N)
- Loop is executed: N times

**N x O(N) = O(N²)**

# Examples (cont.'d)

```
if (i<j)
    for ( i=0; i<N; i++ )
      X = X+i;
else
    X=0;
```

O(N)

O(1)

Max ( O(N), O(1) ) = O (N)

# Other Complexity Measures

- o notation: asymptotic “less than”:
  - f(n)=o(g(n)) implies: f(n) “<” cg(n) , $\forall n \geq \boldsymbol{n_o}$
- $\Omega$ notation: asymptotic “greater than or equal to”:
  - f(n)= $\Omega$ (g(n)) implies: f(n) “≥” cg(n) , $\forall n \geq \boldsymbol{n_o}$
- $\omega$ notation: asymptotic "greater than"
  - $\omega$ (g(n)) implies: $f(n) > cg(n)$, $\forall n \geq \boldsymbol{n_o}$
- $\Theta$ notation: asymptotic “equality”:
  - f(n)= $\Theta$ (g(n)) implies: f(n) “=” cg(n) , $\forall n \geq \boldsymbol{n_o}$

# Definition: *O(g), at most order g*

Let *f*, *g* be functions $\mathbf{R} \to \mathbf{R}$.

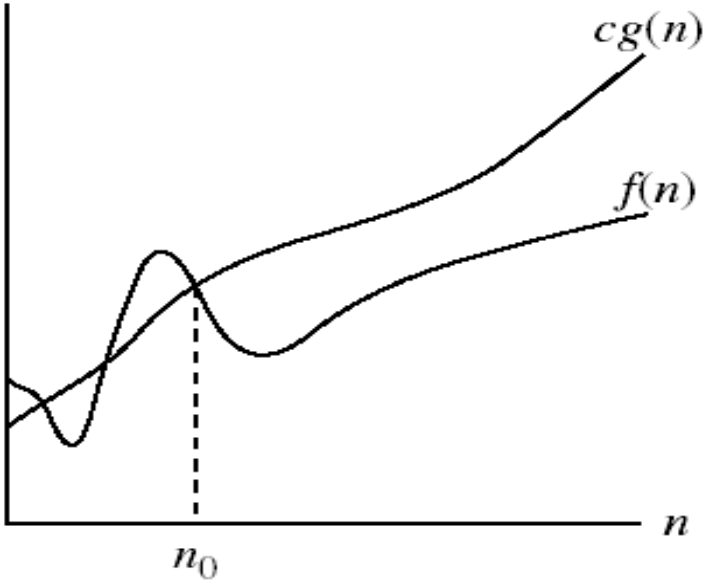
- We say that "*f* is *at <u>most</u> order g*", if:

$$\exists c, n_0: \ \mathbf{0} \leq f(n) \leq cg(n), \ \forall n \geq n_0$$

"Beyond some point $n_0$, function *f* is at most a constant *c* times *g* (*i.e.*, proportional to *g*)."

- "*f* is *at most order g*", or "*f* is O(*g*)", or "*f*=O(*g*)" all just mean that $f \in O(g)$.
- Sometimes the phrase "at most" is omitted.

# Big-O Visualization




$g(n)$ is an ***asymptotic upper bound*** for $f(n)$.

# Common orders of magnitude

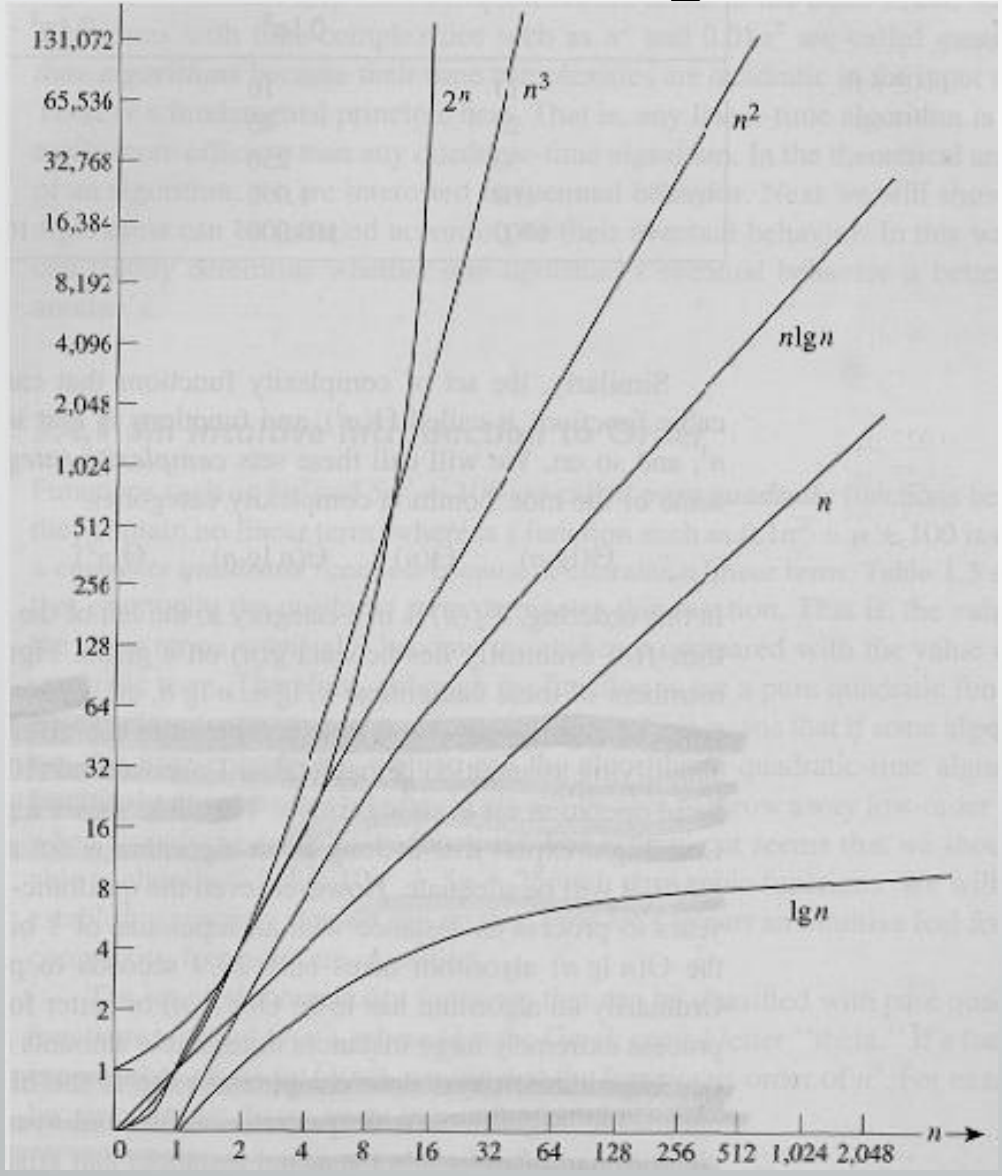


**Table 1.4** Execution times for algorithms with the given time complexities

| $n$ | $f(n) = \lg n$ | $f(n) = n$ | $f(n) = n \lg n$ | $f(n) = n^2$ | $f(n) = n^3$ | $f(n) = 2^n$ |
|---|---|---|---|---|---|---|
| 10 | 0.003 $\mu$s* | 0.01 $\mu$s | 0.033 $\mu$s | 0.1 $\mu$s | 1 $\mu$s | [illegible] $\mu$s |
| 20 | 0.004 $\mu$s | 0.02 $\mu$s | 0.086 $\mu$s | 0.4 $\mu$s | 8 $\mu$s | [illegible] ms† |
| 30 | 0.005 $\mu$s | 0.03 $\mu$s | 0.147 $\mu$s | 0.9 $\mu$s | 27 $\mu$s | [illegible] s |
| 40 | 0.005 $\mu$s | 0.04 $\mu$s | 0.213 $\mu$s | 1.6 $\mu$s | 64 $\mu$s | 18.3 min |
| 50 | 0.005 $\mu$s | 0.05 $\mu$s | 0.282 $\mu$s | 2.5 $\mu$s | [illegible]25 $\mu$s | 13 days |
| $10^2$ | 0.007 $\mu$s | 0.10 $\mu$s | 0.664 $\mu$s | 10 $\mu$s | 1 ms | $4 \times 10^{15}$ years |
| $10^3$ | 0.010 $\mu$s | 1.00 $\mu$s | 9.966 $\mu$s | 1 ms | 1 s | |
| $10^4$ | 0.013 $\mu$s | [illegible]0 $\mu$s | 130 $\mu$s | 100 ms | 16.7 min | |
| $10^5$ | 0.017 $\mu$s | 0.10 ms | 1.67 ms | 10 s | 11.6 days | |
| $10^6$ | 0.020 $\mu$s | 1 ms | 19.93 ms | 16.7 min | 31.7 years | |
| $10^7$ | 0.023 $\mu$s | 0.01 s | 0.23 s | 1.16 days | 31,709 years | |
| $10^8$ | 0.027 $\mu$s | 0.10 s | 2.66 s | 115.7 days | 3.17 $\times$ [illegible] years | |
| $10^9$ | 0.030 $\mu$s | 1 s | 29.90 s | 31.7 years | | |

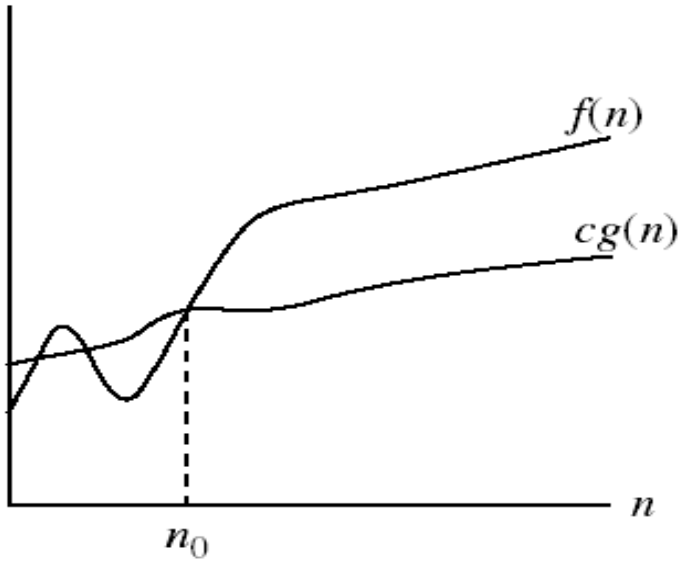
*1 $\mu$s = $10^{-6}$ second.
†1 ms = $10^{-3}$ second.

# Definition: $\Omega(g)$, *at least order g*

Let $f,g$ be any function $\mathbf{R} \to \mathbf{R}$.

- We say that "*f is at least order g*", written $\Omega(g)$, if
  $$\exists c, n_0 : f(n) \geq cg(n), \forall n \geq n_0$$
- "Beyond some point $n_0$, function $f$ is at least a constant $c$ times $g$ (*i.e.*, proportional to $g$)."
  - Often, one deals only with positive functions and can ignore absolute value symbols.
- "$f$ is *at least order g*", or "$f$ is $\Omega(g)$", or "$f = \Omega(g)$" all just mean that $f \in \Omega(g)$.

# Big- $\Omega$ Visualization




$g(n)$ is an ***asymptotic lower bound*** for $f(n)$.

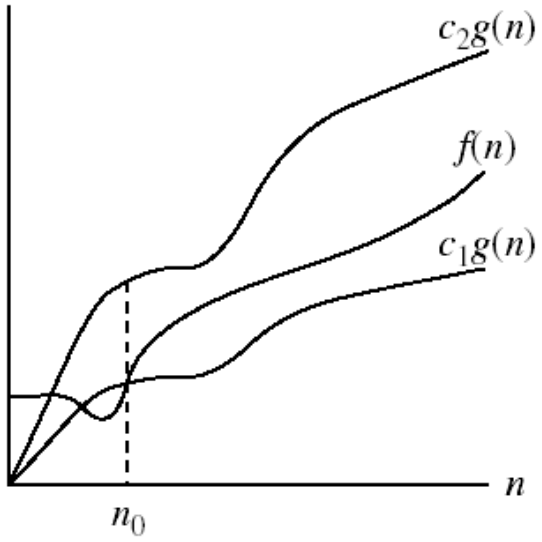
# Definition: $\Theta(g)$, *exactly order g*

- If $f \in O(g)$ and $g \in O(f)$ then we say "*g and f are of the same order*" or "*f is (exactly) order g*" and write $f \in \Theta(g)$.
- Another equivalent definition:

  $\exists c_1 c_2, n_0: c_1 g(n) \leq f(n) \leq c_2 g(n), \forall n \geq n_0$
- "Everywhere beyond some point $n_0$, $f(n)$ lies in between two multiples of $g(n)$."
- $\Theta(g) \equiv O(g) \cap \Omega(g)$

  (i.e., $f \in O(g)$ and $f \in \Omega(g)$ )

# Big- Θ Visualization




$g(n)$ is an ***asymptotically tight bound*** for $f(n)$.

# Review: Orders of Growth

Definitions of order-of-growth sets, $\forall g{:}\mathbf{R}\rightarrow\mathbf{R}$

- $O(g) \equiv \{f \mid \exists\, c, \boldsymbol{n_o}: f(n) \leq cg(n),\ \forall n \geq \boldsymbol{n_o}\}$
- $o(g) \equiv \{f \mid \forall c\ \exists \boldsymbol{n_o}: f(n) < cg(n), \forall n \geq \boldsymbol{n_o}\}$
- $\Omega(g) \equiv \{f \mid \mid \exists c, \boldsymbol{n_o}: f(n) \geq cg(n), \forall n \geq \boldsymbol{n_o}\}$
- $\omega(g) \equiv \{f \mid \forall\ c\ \exists \boldsymbol{n_o}: f(n) > cg(n),\ \forall n \geq \boldsymbol{n_o}\}$
- $\Theta(g) \equiv \{f \mid \exists c_1 c_2,\ \boldsymbol{n_o}: c_1 g(n) \leq f(n) \mid \leq c_2 g(n),\ \forall n \geq \boldsymbol{n_o}\}$

# Rate of Growth