Analysis of Algorithms

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

Input Size

- 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

Types of Analysis

- Worst case
  - Provides an upper bound on running time
  - An absolute guarantee that the algorithm would not run longer, no matter what the inputs are
- Best case
  - Provides a lower bound on running time
  - Input is the one for which the algorithm runs the fastest

$$\boxed{\textit{Lower Bound} \le \textit{Running Time} \le \textit{Upper Bound}}$$

- Average case
  - Provides a prediction about the running time
  - Assumes that the input is random

How do we compare algorithms?

- We need to define a number of <u>objective measures</u>.

(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.

Ideal Solution

- Express running time as a function of the input size *n* (i.e., $f(n)$).
- Compare different functions corresponding to running times.
- Such an analysis is independent of machine time, programming style, etc.

Example

- Associate a "cost" with each statement.
- Find the "total cost" by finding the total number of times each statement is executed.

Algorithm 1

	Cost
arr[0] = 0;	c_1
arr[1] = 0;	c_1
arr[2] = 0;	c_1
...	...
arr[N-1] = 0;	c_1

$$c_1+c_1+...+c_1 = c_1 \times N$$

Algorithm 2

	Cost
for(i=0; i<N; i++)	c_2
arr[i] = 0;	c_1

$$(N+1) \times c_2 + N \times c_1 =$$
$$(c_2 + c_1) \times N + c_2$$

Another Example

- ***Algorithm 3*** *Cost*

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

$c_1 + c_2 \times (N+1) + c_2 \times N \times (N+1) + c_3 \times N^2$

Asymptotic Analysis

- To compare two algorithms with running times *f(n)* and *g(n),* we need a **rough measure** that characterizes **how fast each function grows.**
- *Hint:* use *rate of growth*
- Compare functions in the limit, that is, **asymptotically!**

  (i.e., for large values of *n*)

Asymptotic Notation

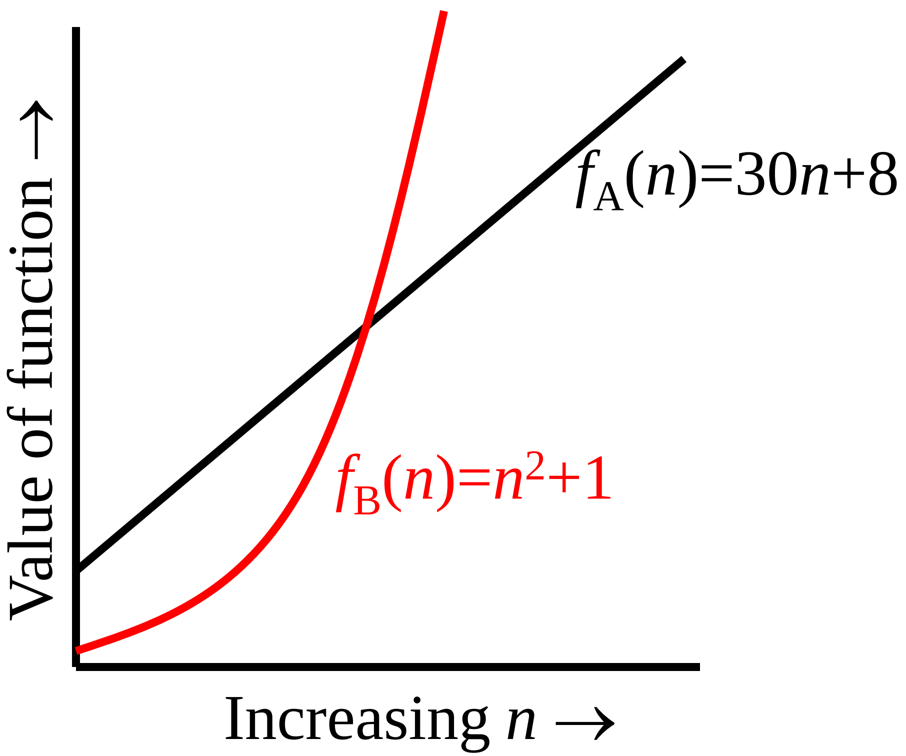
- O notation: asymptotic "less than":
  - $f(n)=O(g(n))$ implies: $f(n)$ "$\leq$" $g(n)$
- Ω notation: asymptotic "greater than":
  - $f(n)=\Omega(g(n))$ implies: $f(n)$ "$\geq$" $g(n)$
- Θ notation: asymptotic "equality":
  - $f(n)=\Theta(g(n))$ implies: $f(n)$ "$=$" $g(n)$

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, any $O(n^2)$ function is faster-growing than any $O(n)$ function.

Visualizing Orders of Growth

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



More Examples …

- $n^4 + 100n^2 + 10n + 50$ is $O(n^4)$
- $10n^3 + 2n^2$ is $O(n^3)$
- $n^3 - n^2$ is $O(n^3)$
- constants
  - 10 is $O(1)$
  - 1273 is $O(1)$

Back to Our Example

Algorithm 1

	Cost
arr[0] = 0;	c_1
arr[1] = 0;	c_1
arr[2] = 0;	c_1
...	
arr[N-1] = 0;	c_1

$c_1+c_1+...+c_1 = c_1 \times N$

Algorithm 2

	Cost
for(i=0; i<N; i++)	c_2
arr[i] = 0;	c_1

$(N+1) \times c_2 + N \times c_1 =$
$(c_2 + c_1) \times N + c_2$

- Both algorithms are of the same order: *O(N)*

Example (cont'd)

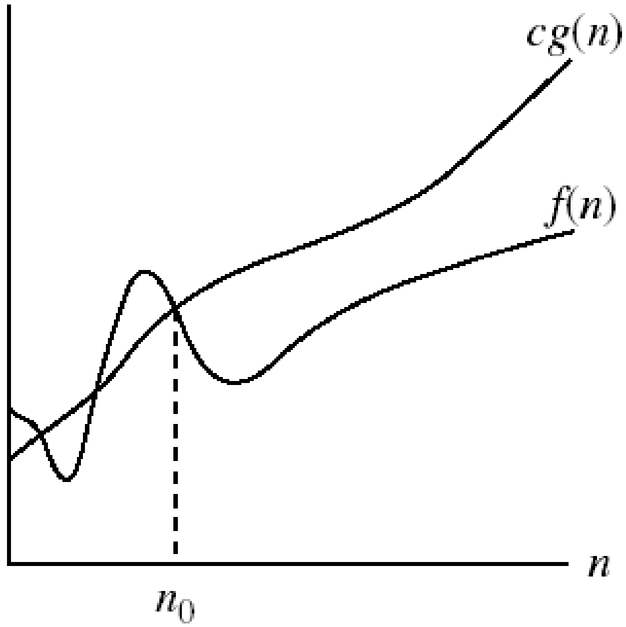
Algorithm 3	***Cost***
sum = 0;	c_1
for(i=0; i<N; i++)	c_2
for(j=0; j<N; j++)	c_2
sum += arr[i][j];	c_3

$c_1 + c_2 \; x \; (N+1) + c_2 \; x \; N \; x \; (N+1) + c_3 \; x \; N^2 = O(N^2)$

Asymptotic notations

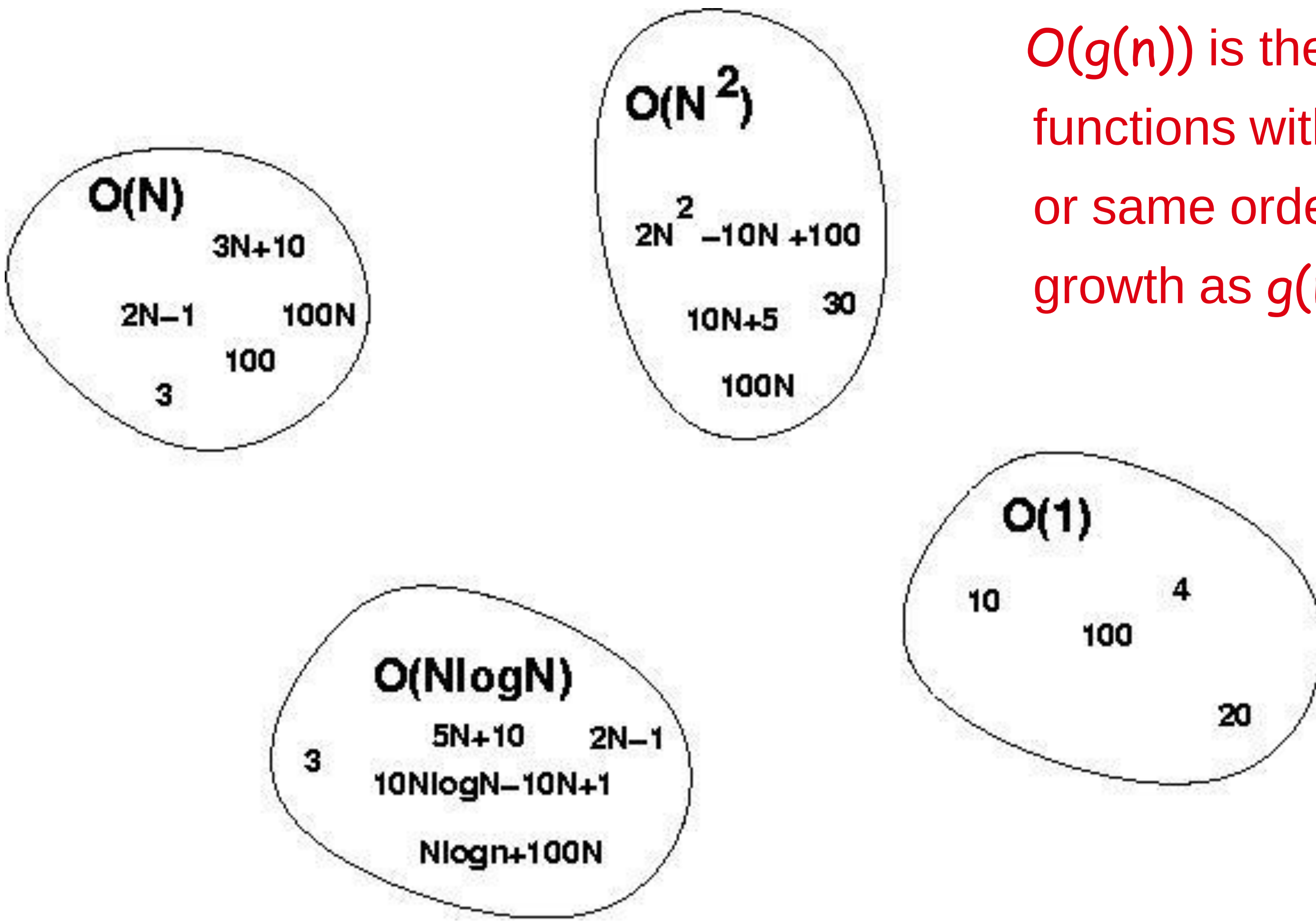
- *O-notation*

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



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

Big-O Visualization



$O(g(n))$ is the set of functions with smaller or same order of growth as $g(n)$

Examples

- $n^2 = O(n^2)$:
- $n^2 = O(n^2)$: $n^2 \le cn^2 \Rightarrow c \ge 1 \Rightarrow c = 1$ and $n_0 = 1$
- $1000n^2+1000n = O(n^2)$:

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

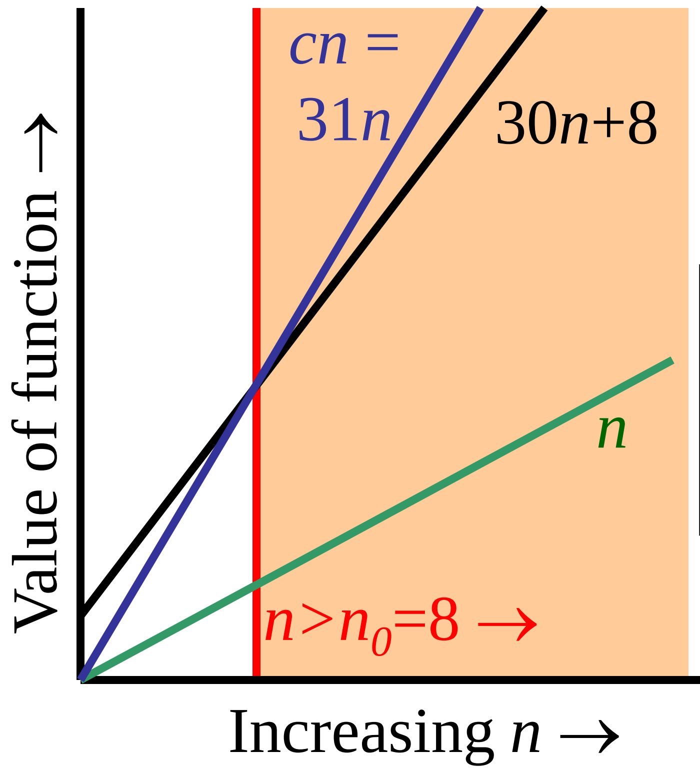
- $n = O(n^2)$: $n \le cn^2 \Rightarrow cn \ge 1 \Rightarrow c = 1$ and $n_0 = 1$

More Examples

- Show that $30n+8$ is $O(n)$.
  - Show $\exists c, n_0$: $30n+8 \leq cn$, $\forall n > n_0$.
    - Let $c=31$, $n_0=8$. Assume $n > n_0 = 8$. Then $cn = 31n = 30n + n > 30n+8$, so $30n+8 < cn$.

Big-O example, graphically

- Note $30n+8$ isn't less than n *anywhere* ($n>0$).
- It isn't even less than $31n$ *everywhere*.
- But it *is* less than $31n$ <u>everywhere to the right of $n=8$</u>.



$\boxed{30n+8 \in O(n)}$

No Uniqueness

- There is no unique set of values for n_0 and c in proving the asymptotic bounds
- Prove that $100n + 5 = O(n^2)$
  - $100n + 5 \le 100n + n = 101n \le 101n^2$

    for all $n \ge 5$

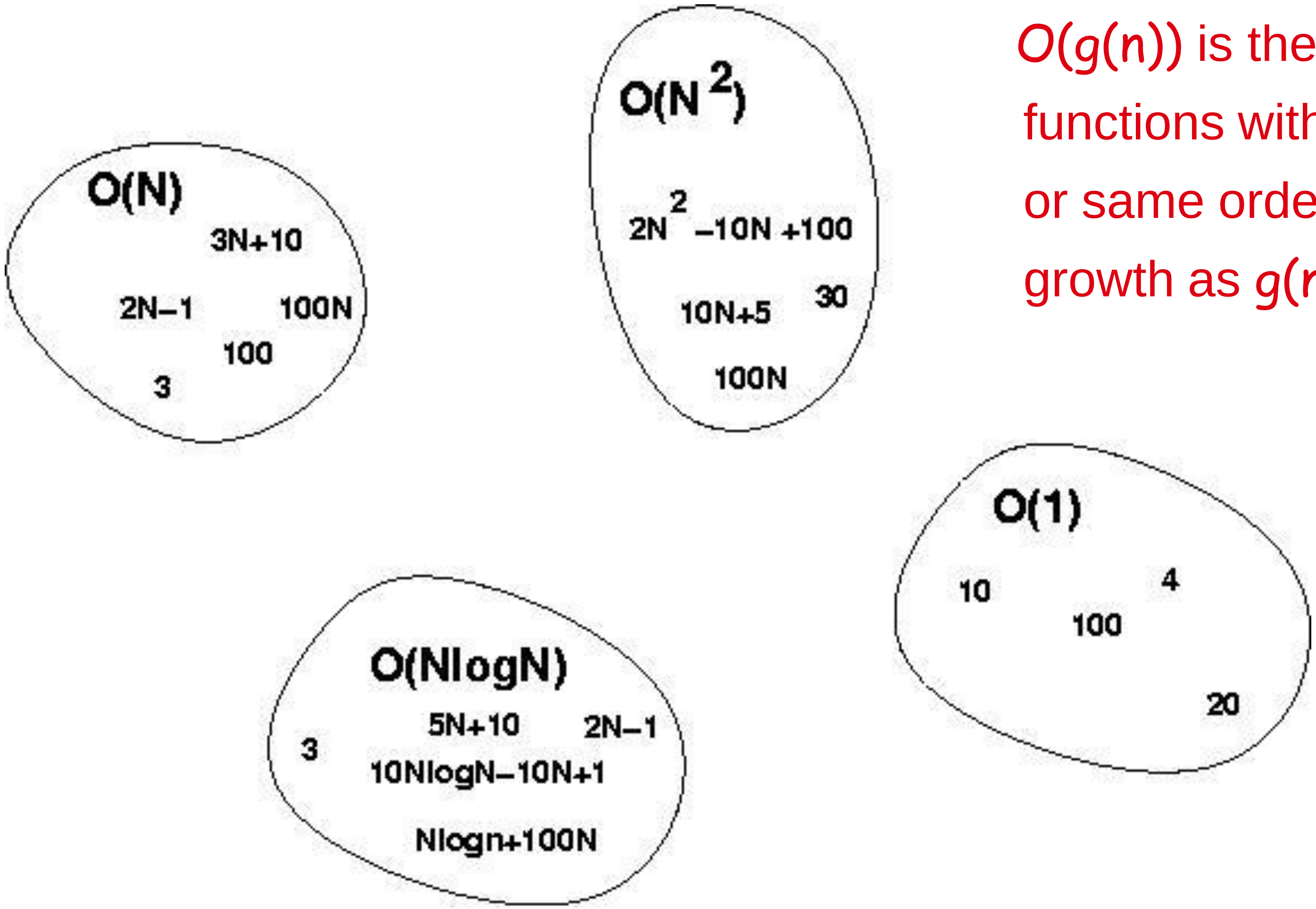
    $n_0 = 5$ and $c = 101$ is a solution
  - $100n + 5 \le 100n + 5n = 105n \le 105n^2$

    for all $n \ge 1$

    $n_0 = 1$ and $c = 105$ is also a solution

Must find **SOME** constants c and n_0 that satisfy the asymptotic notation relation

Big-O Visualization

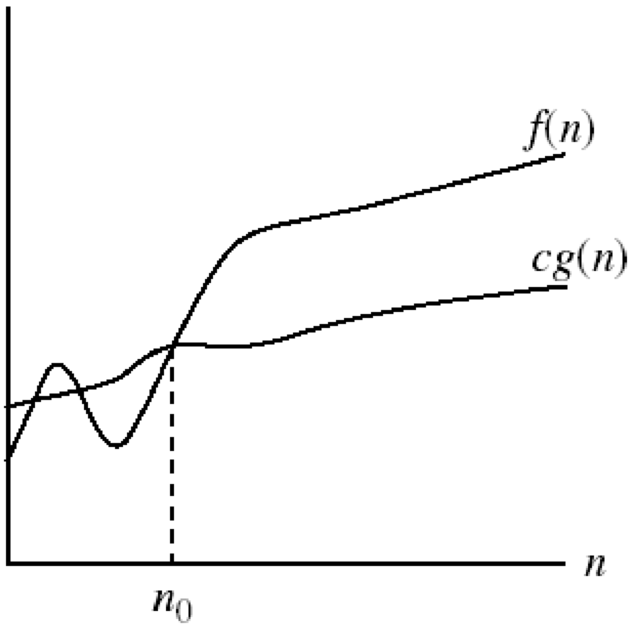


$O(g(n))$ is the set of functions with smaller or same order of growth as $g(n)$

Asymptotic notations (cont.)

- Ω *- notation*

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



Ω(*g*(n)) is the set of functions with larger or same order of growth as *g*(n)

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

- $5n^2 = \Omega(n)$

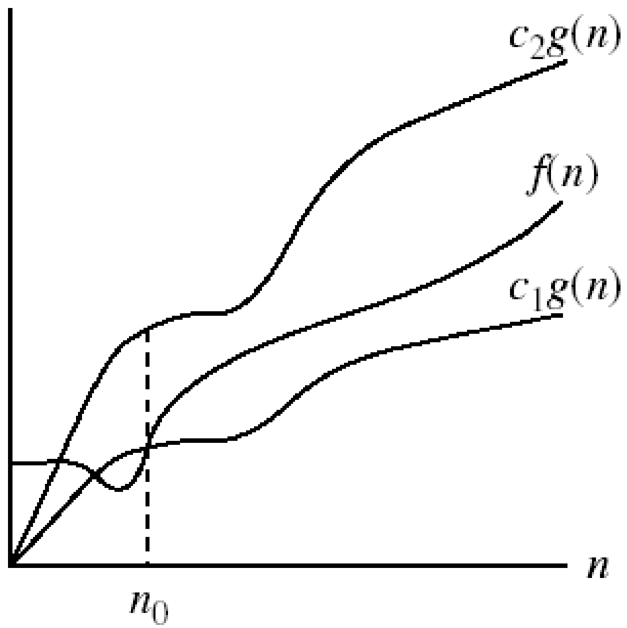
  $\exists\ c, n_0$ such that: $0 \leq cn \leq 5n^2$ $\Rightarrow cn \leq 5n^2$

  $\Rightarrow c = 1$ and $n_0 = 1$

Asymptotic notations (cont.)

- *Θ-notation*

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



$\Theta(g(n))$ is the set of functions with the same order of growth as $g(n)$

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

Examples

- $n^2/2 - n/2 = \Theta(n^2)$
    - $\frac{1}{2} n^2 - \frac{1}{2} n \leq \frac{1}{2} n^2 \ \forall n \geq 0 \quad \Rightarrow \quad c_2 = \frac{1}{2}$
    - $\frac{1}{2} n^2 - \frac{1}{2} n \geq \frac{1}{2} n^2 - \frac{1}{2} n * \frac{1}{2} n \ (\forall n \geq 2) = \frac{1}{4} n^2$

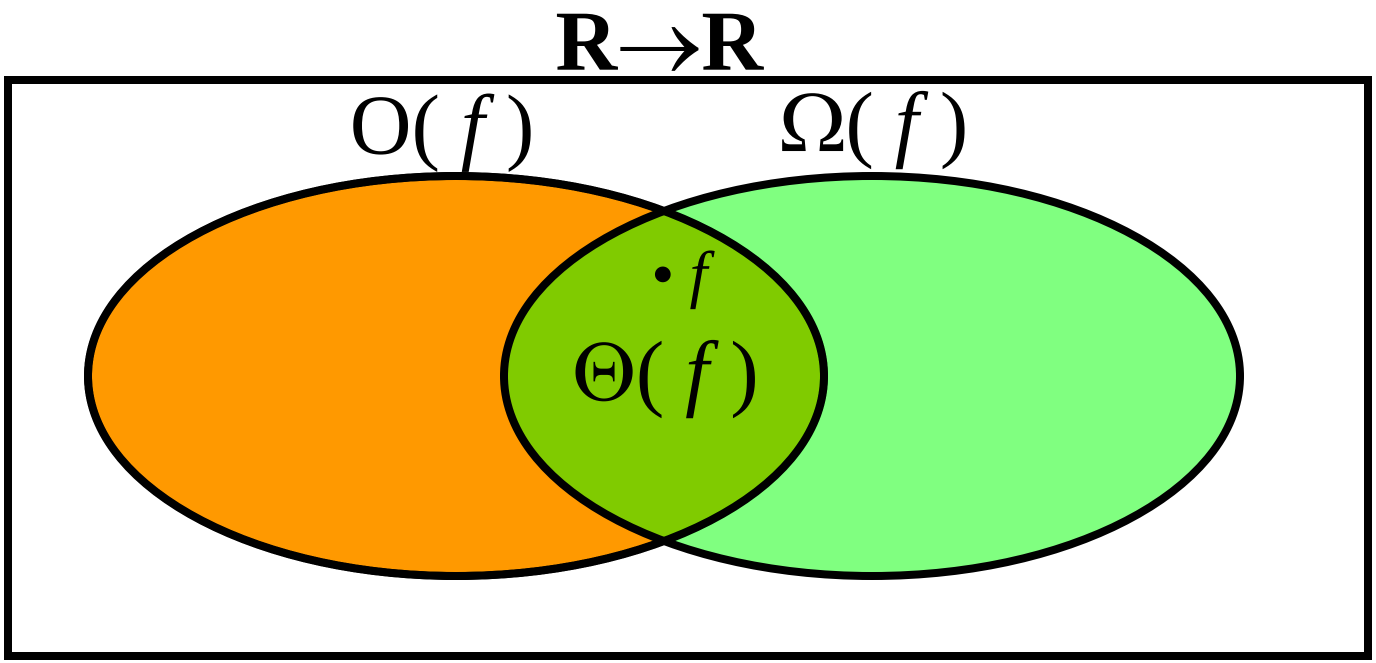
      $\Rightarrow \quad c_1 = \frac{1}{4}$

- $n \neq \Theta(n^2)$: $c_1 n^2 \leq n \leq c_2 n^2$

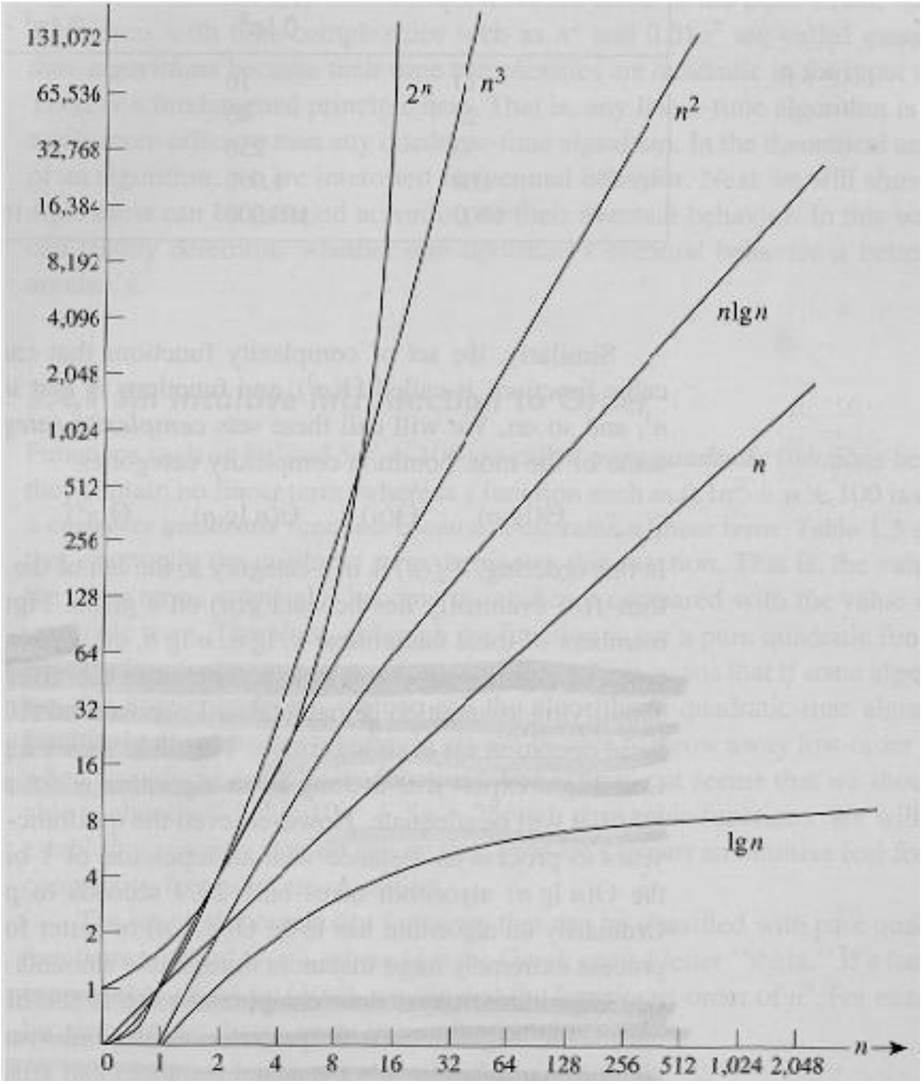
  $\Rightarrow$ only holds for: $n \leq 1/c_1$

Relations Between Different Sets

- Subset relations between order-of-growth sets.



Common orders of magnitude



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 μs*	0.01 μs	0.033 μs	0.1 μs	1 μs	1 μs
20	0.004 μs	0.02 μs	0.086 μs	0.4 μs	8 μs	1 ms†
30	0.005 μs	0.03 μs	0.147 μs	0.9 μs	27 μs	1 s
40	0.005 μs	0.04 μs	0.213 μs	1.6 μs	64 μs	18.3 min
50	0.005 μs	0.05 μs	0.282 μs	2.5 μs	125 μs	13 days
10^2	0.007 μs	0.10 μs	0.664 μs	10 μs	1 ms	4×10^{13} years
10^3	0.010 μs	1.00 μs	9.966 μs	1 ms	1 s	
10^4	0.013 μs	10 μs	130 μs	100 ms	16.7 min	
10^5	0.017 μs	0.10 ms	1.67 ms	10 s	11.6 days	
10^6	0.020 μs	1 ms	19.93 ms	16.7 min	31.7 years	
10^7	0.023 μs	0.01 s	0.23 s	1.16 days	31,709 years	
10^8	0.027 μs	0.10 s	2.66 s	115.7 days	3.17×10^7 years	
10^9	0.030 μs	1 s	29.90 s	31.7 years		

*1 μs = 10^{-6} second.
†1 ms = 10^{-3} second.

Logarithms and properties

- In algorithm analysis we often use the notation "log n" without specifying the base

Binary logarithm $\lg n = \log_2 n$

Natural logarithm $\ln n = \log_e n$

$$\lg^k n = (\lg n)^k$$

$$\lg \lg n = \lg(\lg n)$$

$$\log x^y = y \log x$$

$$\log xy = \log x + \log y$$

$$\log \frac{x}{y} = \log x - \log y$$

$$a^{\log_b x} = x^{\log_b a}$$

$$\log_b x = \frac{\log_a x}{\log_a b}$$

More 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.

  - $f(n) = n$; $g(n) = \log n^2$ $\quad$ $f(n) = \Omega(g(n))$
  - $f(n) = \log \log n$; $g(n) = \log n$ $\quad$ $f(n) = O(g(n))$
  - $f(n) = n$; $g(n) = \log^2 n$ $\quad$ $f(n) = \Omega(g(n))$
  - $f(n) = n \log n + n$; $g(n) = \log n$ $\quad$ $f(n) = \Omega(g(n))$

  - $f(n) = 2^n$; $g(n) = 10n^2$ $\quad$ $f(n) = \Omega(g(n))$
  - $f(n) = 2^n$; $g(n) = 3^n$ $\quad$ $f(n) = O(g(n))$

Properties

- *Theorem:*

$$f(n) = \Theta(g(n)) \Leftrightarrow f = O(g(n)) \text{ and } f = \Omega(g(n))$$

- Transitivity:
  - $f(n) = \Theta(g(n))$ and $g(n) = \Theta(h(n)) \Rightarrow f(n) = \Theta(h(n))$
  - Same for O and Ω
- Reflexivity:
  - $f(n) = \Theta(f(n))$
  - Same for O and Ω
- Symmetry:
  - $f(n) = \Theta(g(n))$ if and only if $g(n) = \Theta(f(n))$
- Transpose symmetry:
  - $f(n) = O(g(n))$ if and only if $g(n) = \Omega(f(n))$