Algorithms

Analysis of Algorithms

Goal:

- To analyze and compare algorithms in terms of *running time* and *memory requirements* (i.e. **time** and **space complexity**)

- Running time and space requirements change as we increase the input size n

  Input size (number of elements in the input)

  - size of an *array* or a *matrix*
  - No of bits in the binary representation of the input
  - vertices and/or edges in a graph, etc.

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

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

Lower Bound ≤ Running Time ≤ Upper Bound

Computing the Running Time

High-level programming languages have statements which require a large number of low-level machine language instructions to execute (a function of the input size n).

For example, a subroutine call can not be counted as one statement; it needs to be analyzed separately

- Associate a "**cost**" with each statement.

  Find the "total cost“ by multiplying the cost with the total number of times each statement is executed.

Example

Algorithm X	*Cost*
sum = 0;	c_1
for(i=0; i<N; i++)	c_2
for(j=0; j<N; j++)	c_3
sum += arrY[i][j];	c_4

Total Cost = $c_1 + c_2 * (N+1) + c_3 * N * (N+1) + c_4 * N^2$

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** with respect to **n**

- In other words, we are interested in how they behave **asymptotically** (i.e. for large *n*) (called rate of growth)

- **Big O notation**: asymptotic "less than" or "at most":

  f(n)=O(g(n)) implies: f(n) "≤" g(n)

- **Ω notation**: asymptotic "greater than" or "at least":

  f(n)= Ω (g(n)) implies: f(n) "≥" g(n)

- **Θ notation**: asymptotic "equality" or "exactly":

  f(n)= Θ (g(n)) implies: f(n) "=" g(n)

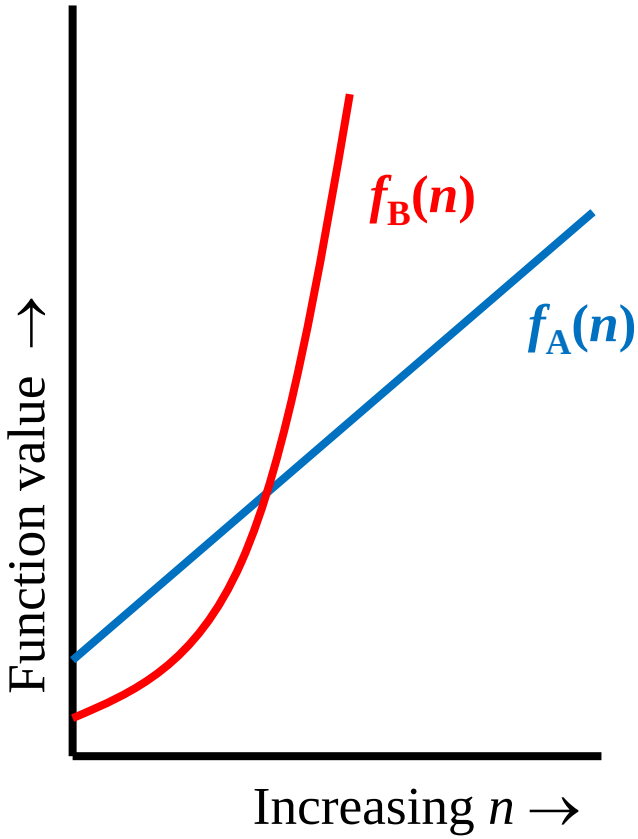
Big-O Notation

- We say

  $f_A(n) = 7n+18$ is ***order n***, or **O (n)**
  It is, at most, roughly *proportional* to n.

  $f_B(n) = 3n^2+5n+4$ is ***order n²***, 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.



More Examples …

- $n^4 + 100n^2 + 10n + 50$ ➔ $O(n^4)$
  $10n^3 + 2n^2$ ➔ $O(n^3)$
  $n^3 - n^2$ ➔ $O(n^3)$

- constants
  10 is $O(1)$
  1273 is $O(1)$

- what is the rate of growth for ***Algorithm X*** studied earlier (in Big O notation)?

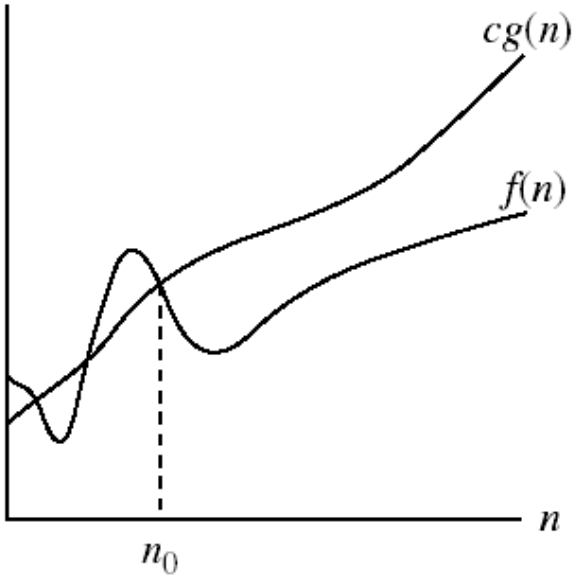
  $Total\ Time = c_1 + c_2*(N+1) + c_2 * N*(N+1) + c_3*N^2$

  If c_1, c_2, c_3 , and c_4 are constants then ***Total Time = O(N^2)***

Definition of Big O

- *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 $f(n)$.

Definition of Small o

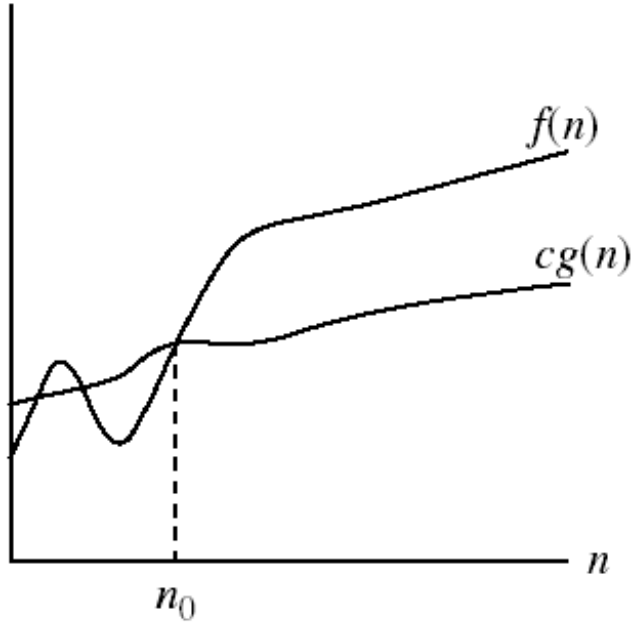
- commonly written as o, is an Asymptotic Notation to denote the upper bound (that is not asymptotically tight) on the growth rate of runtime of an algorithm.

- f(n) is o(g(n)), if for all real constants c (c > 0) and n0 (n0 > 0), f(n) is < c g(n) for every input size n (n > n0).

- The definitions of O-notation and o-notation are similar. The main difference is that in f(n) = O(g(n)), the bound f(n) <= g(n) holds for some constant c > 0, but in f(n) = o(g(n)), the bound f(n) < c g(n) holds for all constants c > 0.

Definition of Ω

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



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

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

Small-omega

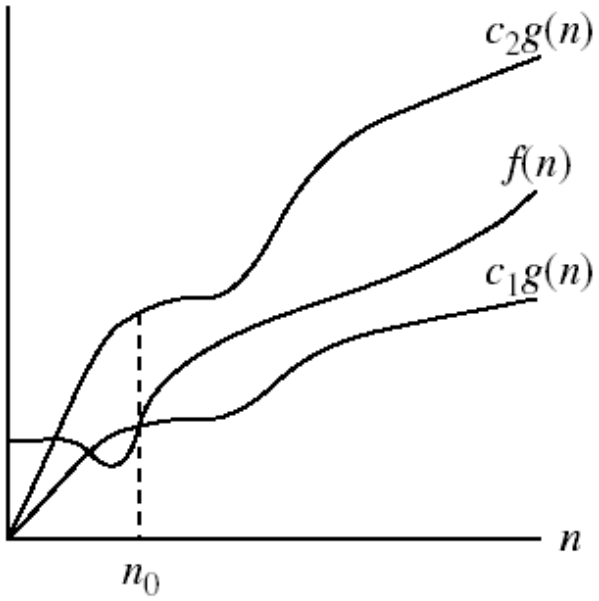
- Small-omega, commonly written as ω, is an Asymptotic Notation to denote the lower bound (that is not asymptotically tight) on the growth rate of runtime of an algorithm.

- f(n) is ω(g(n)), if for all real constants c (c > 0) and n0 (n0 > 0), f(n) is > c g(n) for every input size n (n > n0).

- The definitions of Ω-notation and ω-notation are similar. The main difference is that in f(n) = Ω(g(n)), the bound f(n) >= g(n) holds for some constant c > 0, but in f(n) = ω(g(n)), the bound f(n) > c g(n) holds for all constants c > 0.

Definition of Θ

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

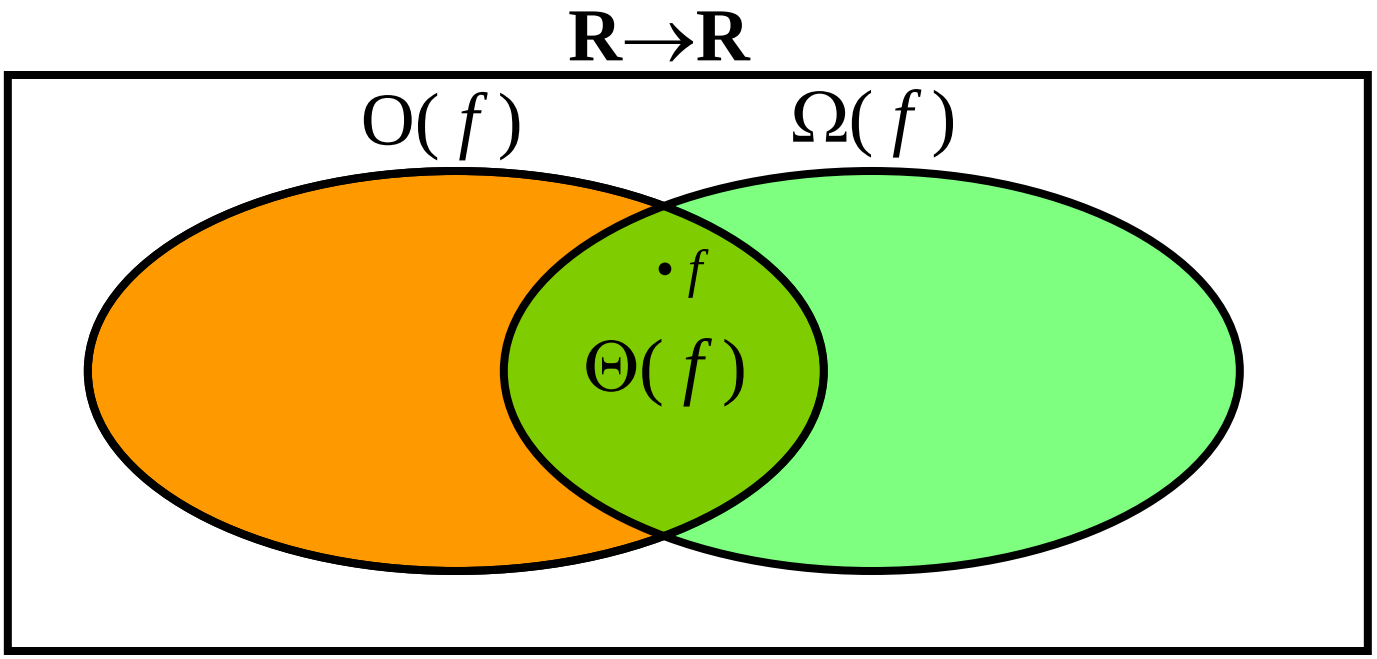


$\Theta(g(n))$ is the set of functions with the same order of growth as $g(n)$

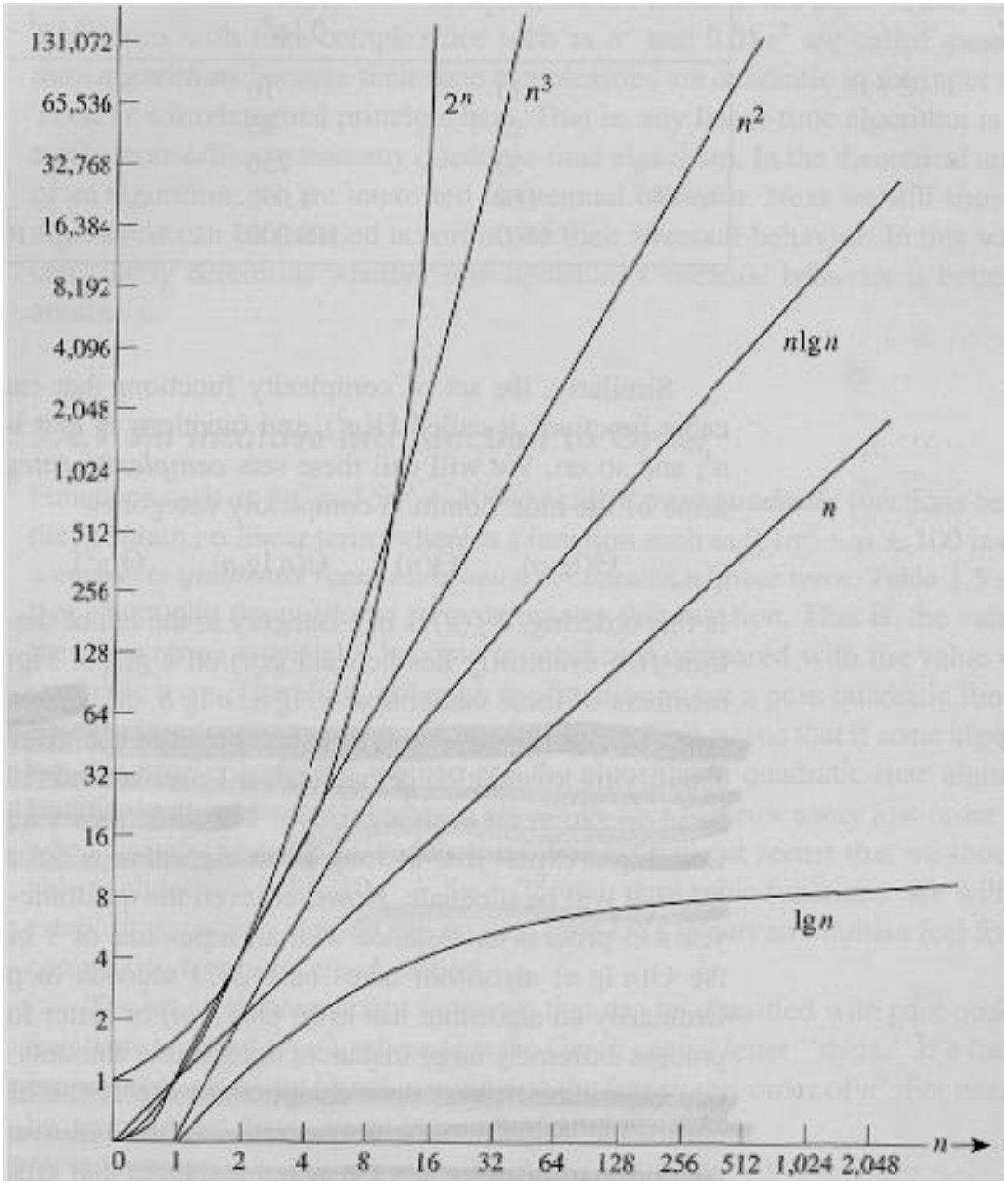
$g(n)$ is an ***asymptotically tight bound*** for $f(n)$.

Relations Between Different Sets

- Subset relations between order-of-growth sets.



Common orders of magnitude



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 $g(n) = \Omega(f(n))$

Complexity Analysis

Algorithm	Data structure	Time complexity: Best	Time complexity: Average	Time complexity: Worst	Space complexity: Worst
Quick sort	Array	O(n log(n))	O(n log(n))	$O(n^2)$	O(n)
Merge sort	Array	O(n log(n))	O(n log(n))	O(n log(n))	O(n)
Heap sort	Array	O(n log(n))	O(n log(n))	O(n log(n))	O(1)
Bubble sort	Array	O(n)	$O(n^2)$	$O(n^2)$	O(1)
Insertion sort	Array	O(n)	$O(n^2)$	$O(n^2)$	O(1)
Selection sort	Array	$O(n^2)$	$O(n^2)$	$O(n^2)$	O(1)