Algorithms

Informally Definition

An algorithm is any **well-defined computational procedure** that takes some values or set of values as **input** and produces some values or set of values as **output**

Formal Definition

As a sequence of **computational steps** that transforms the **input** into **output**

Algorithms

Properties of algorithms:

- **Input** from a specified set,
- **Output** from a specified set (solution),
- **Definiteness** of every step in the computation,
- **Correctness** of output for every possible input,
- **Finiteness** of the number of calculation steps,
- **Effectiveness** of each calculation step and
- **Generality** for a class of problems.

?

Suppose computers were infinitely fast and computer memory are free

Is there any reason to study algorithm ?

Yes

- Demonstrate that solution methods terminates and does so with correct answer.

If computers were infinitely fast, any correct method for solving a problem would do.

You would probably want your implementation to be within the bounds of good software engineering practice

In reality

Computers may be fast, but they are not infinitely fast and

Memory may be cheap but it is not free

Computing time is therefore a bounded resource and so is the space in memory

Complexity

In general, we are not so much interested in the time and space complexity for small inputs.

For example, while the difference in time complexity between linear and binary search is meaningless for a sequence with **$n = 10$**, it is gigantic for **$n = 2^{30}$**.

Complexity

For example, let us assume two algorithms A and B that solve the same class of problems.

The time complexity of A is **$5{,}000n$**, the one for B is **$\lceil 1.1^n \rceil$** for an input with **n** elements.

For **$n = 10$**, A requires **50,000** steps, but B only **3**, so B seems to be superior to A.

For **$n = 1000$**, however, A requires 5,000,000 steps, while B requires $2.5 \cdot 10^{41}$ steps.

Complexity

Comparison: time complexity of algorithms A and B

Input Size	Algorithm A	Algorithm B
n	5,000n	$\lceil 1.1^n \rceil$
10	50,000	3
100	500,000	13,781
1,000	5,000,000	$2.5 \cdot 10^{41}$
1,000,000	$5 \cdot 10^9$	$4.8 \cdot 10^{41392}$

Complexity

This means that algorithm B cannot be used for large inputs, while algorithm A is still feasible.

So what is important is the **growth** of the complexity functions.

The growth of time and space complexity with increasing input size n is a suitable measure for the comparison of algorithms.

Growth Function

The order of growth / rate of growth of the running time of an algorithm gives a simple characterization of the algorithm efficiency and allow us to compare the relative performance of alternative algorithm

Asymptotic Efficiency Algorithm

When the input size is large enough so that the rate of growth / order of growth of the running time is relevant

That is we are concerned with how the running time of an algorithm increases with the size of the input in the limit, as the size of the input increases without bound

Asymptotic Efficiency Algorithm

Usually, an algorithm that is asymptotically more efficient will be the best choice for all but very small inputs.

Asymptotic Notation

The notations we use to describe the asymptotic running time of an algorithm are defined in terms of functions whose domains are the set of natural numbers N = {0, 1, 2,.... }

Asymptotic Notation

Asymptotic notations are convenient for describing the worst-case running time function T(n), which is defined only on integer input size.

Asymptotic Notation

Let n be a non-negative integer representing the size of the input to an algorithm

Asymptotic Notation

Let **f(n)** and **g(n)** be two positive functions, representing the number of **basic calculations** (operations, instructions) that an algorithm takes (or the number of memory words an algorithm needs).

Asymptotic Notation

- Θ - Big Theta
- O – Big O
- Ω - Big Omega
- o – Small o
- ω - Small Omega

Θ - Notation

For a given function g(n), we denote by $\Theta(g(n))$ the set of functions

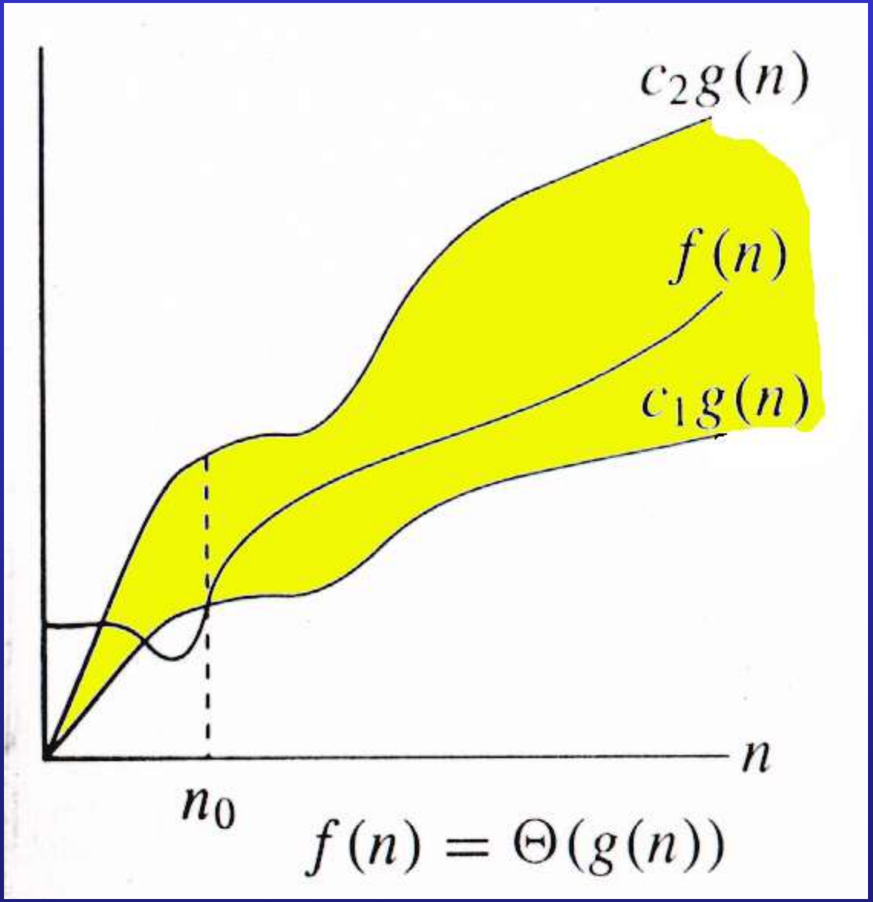
$\Theta(g(n))$ = {f(n) : there exist positive constants c_1 , c_2 , and n_0 such that $0 \le c_1 g(n) \le f(n) \le c_2 g(n)$ for all $n \ge n_0$

$f(n) \in \Theta(g(n))$ $\qquad$ $f(n) = \Theta(g(n))$

Θ - Notation

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

f(*n*) and *g*(*n*) are nonnegative, for large *n*.



Example

$\Theta(g(n)) = \{f(n) : \exists$ **positive constants** c_1, c_2, **and** n_0, **such that** $\forall n \geq n_0$, $0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n)$ $\}$

$$1/2n^2 - 3n = \Theta(n^2)$$

Determine the positive constant n_0, c_1, and c_2 such that

$$c_1 n^2 \leq 1/2n^2 - 3n \leq c_2 n^2 \quad \forall n \geq n_0$$

Example Contd ...

$c_1 \le 1/2 - 3/n \le c_2$ *(Divide by n^2)*

$c_1 = 1/14$, $c_2 = \frac{1}{2}$, $n_0 = 7$

$1/2n^2 - 3n = \Theta(n^2)$

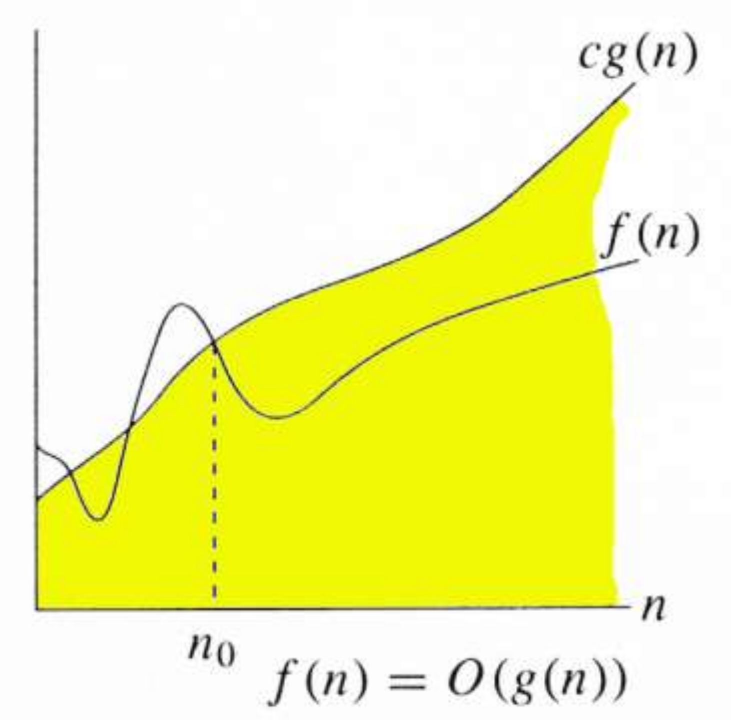
O-Notation

For a given function $g(n)$

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

Intuitively: Set of all functions whose *rate of growth* is the same as or lower than that of c. *g*(*n*)

O-Notation



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

$f(n) = \Theta(g(n)) \Rightarrow f(n) = O(g(n))$.

$\Theta(g(n)) \subset O(g(n))$.

Example

$O(g(n)) = \{f(n)$: there exist positive constants c and n_0 such that $0 \leq f(n) \leq c\, g(n)$ for all $n \geq n_0$ }

Any linear *function* $an + b$ is in $O(n^2)$. **How?**

$C = a + |b|$ and $n_0 = 1$

Big-O Notation (Examples)

$f(n) = 5n+2 = O(n)$ // $g(n) = n$

- $f(n) \le 6n$, for $n \ge 3$ ($C=6$, $n_0=3$)

$f(n)=n/2 - 3 = O(n)$

- $f(n) \le 0.5\,n$ for $n \ge 0$ ($C=0.5$, $n_0=0$)

$n^2-n = O(n^2)$ // $g(n) = n^2$

- $n^2-n \le n^2$ for $n \ge 0$ ($C=1$, $n_0=0$)

$n(n+1)/2 = O(n^2)$

- $n(n+1)/2 \le n^2$ for $n \ge 0$ ($C=1$, $n_0=0$)

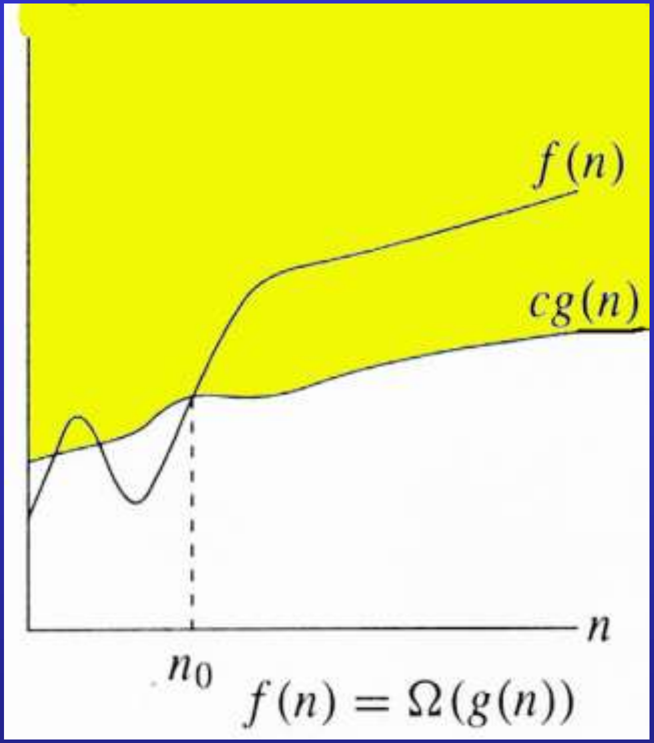
Ω - Notation

For a given function g(n)

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

Intuitively: Set of all functions whose *rate of growth* is the same as or higher than that of *g*(*n*).

Ω - Notation

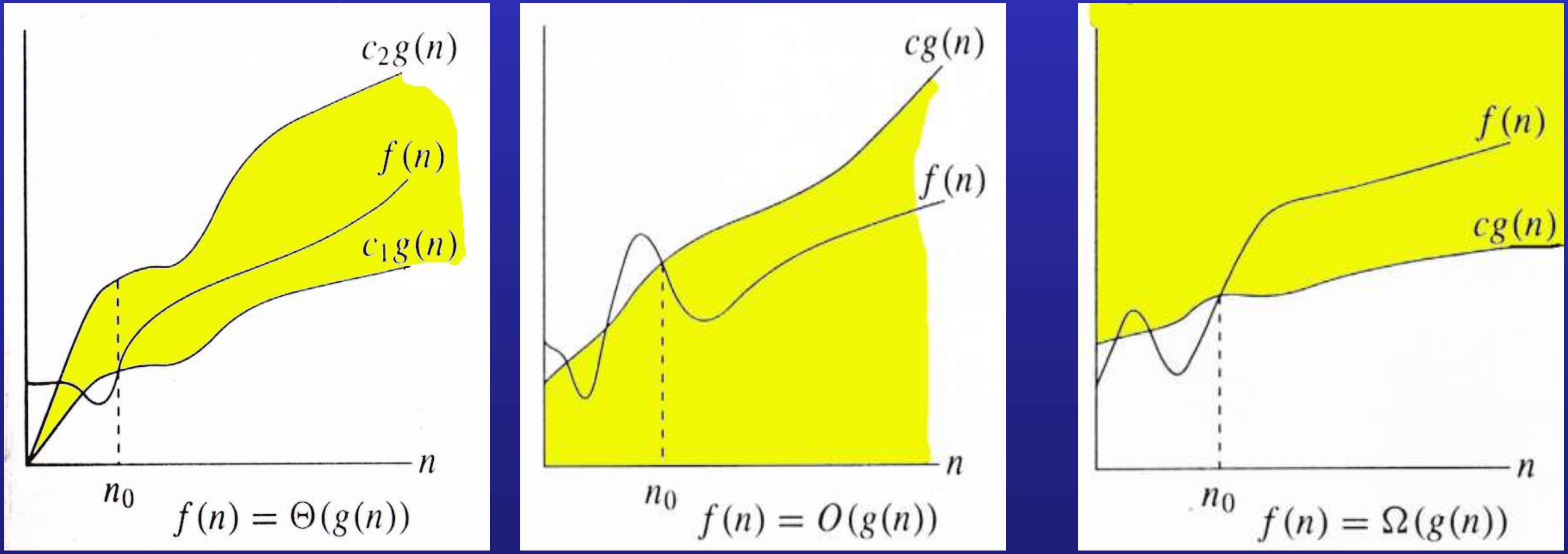


$g(n)$ is an *asymptotic lower bound* for $f(n)$.

$f(n) = \Theta(g(n)) \Rightarrow f(n) = \Omega(g(n))$.

$\Theta(g(n)) \subset \Omega(g(n))$.

Relations Between Θ, *O*, Ω



Relations Between Θ, O, Ω

For any two function f(n) and g(n), we have f(n) = Θ(g(n)) if and only if f(n) = O(g(n)) and f(n) = Ω(g(n))

That is

$$\Theta(g(n)) = O(g(n)) \cap \Omega(g(n))$$

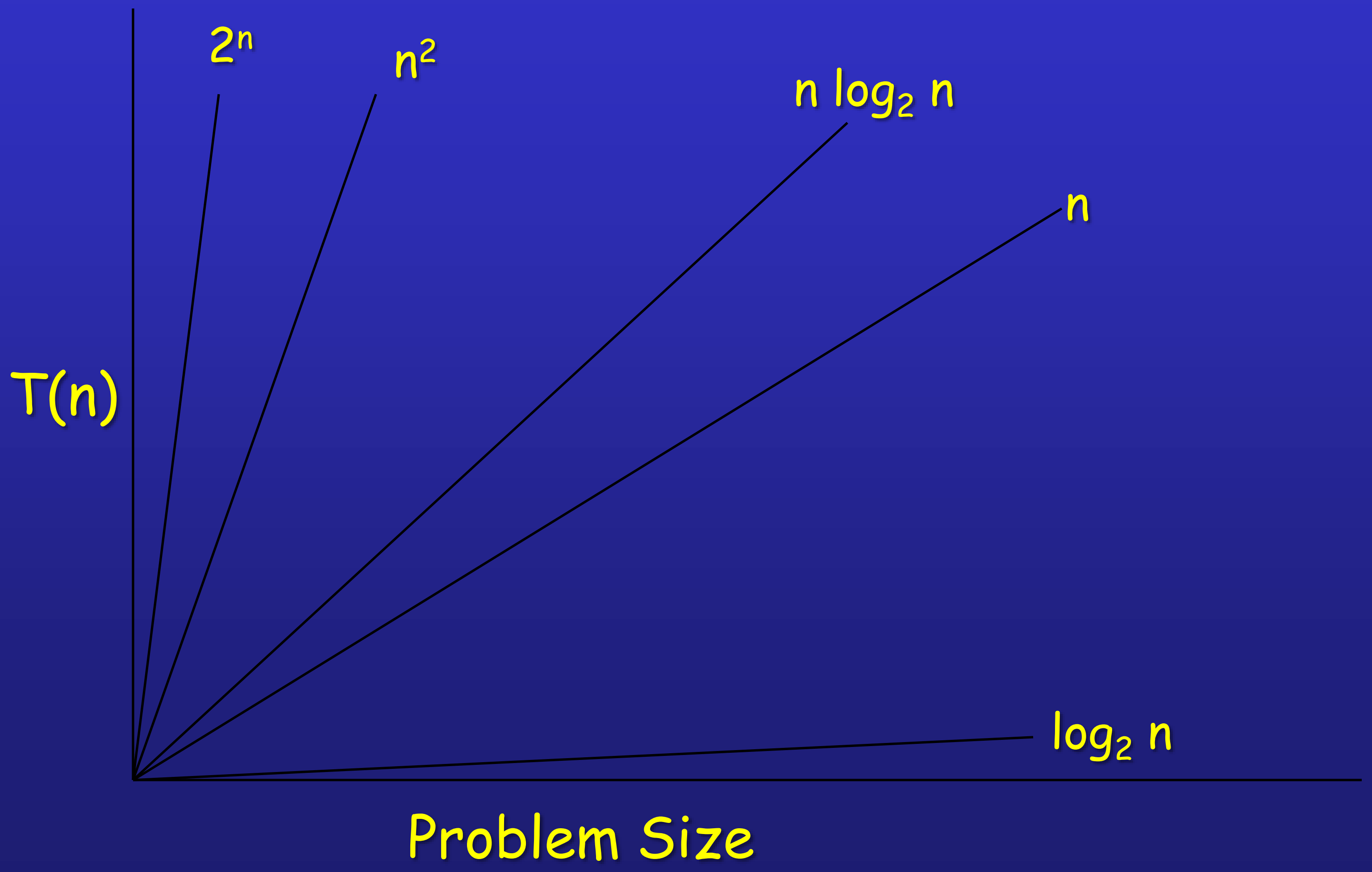
The Growth of Functions

"Popular" functions $g(n)$ are
$n \log n$, 1, 2^n, n^2, $n!$, n, n^3, $\log n$

Listed from slowest to fastest growth:

- 1
- $\log n$
- n
- $n \log n$
- n^2
- n^3
- 2^n
- $n!$

Comparing Growth Rates



Example: Find sum of array elements

Algorithm ***arraySum*** **(*A*, *n*)**	
Input array ***A*** of ***n*** integers	
Output Sum of elements of *A*	***# operations***
sum ← *0*	**1**
for *i* ← 0 **to** *n* – 1 **do**	**n+1**
sum ← *sum* + *A* [*i*]	**n**
return *sum*	1

Input size: n (number of array elements)

Total number of steps: 2n + 3

Example: Find max element of an array

Algorithm *arrayMax*(A, n)	
Input array A of n integers	
Output maximum element of A	***# operations***
currentMax $\leftarrow A[0]$	1
for $i \leftarrow 1$ **to** $n - 1$ **do**	**n**
if $A[i] >$ *currentMax* **then**	**n -1**
currentMax $\leftarrow A[i]$	**n -1**
return *currentMax*	1

- Input size: n (number of array elements)
- Total number of steps: 3n