Algorithms

Dr. Sambit Bakshi
Computer Science & Engineering

Definition

An algorithm is any **well-defined finite number of computational procedure** that receives some values or set of values as **input** and produces some desirable values or set of values as **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.

Reason to Study Algorithm

1. Correctness
2. Finiteness

*** Even if computers were infinitely fast and computer memory is free ***

In reality

- Computers may be fast, but not infinitely fast
- Memory may be large but may be occupied
- Time complexity, Space Complexity

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 $\mathbf{5000}\boldsymbol{n}$, the one for B is $\lceil \mathbf{1.1}^{\boldsymbol{n}} \rceil$ for an input with **n** elements.

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

For $\boldsymbol{n} = \mathbf{1000}$, however, A requires 50,00,000 steps, while B requires $\mathbf{2.5x10^{41}}$ 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	5,00,000	13,781
1,000	50,00,000	2.5×10^{41}
1,000,000	5×10^9	4.8×10^{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.

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 as the size of the input increases without bound.
- Usually, an algorithm that is asymptotically more efficient will be the best choice.

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 $N = \{0, 1, 2, \ldots\}$

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

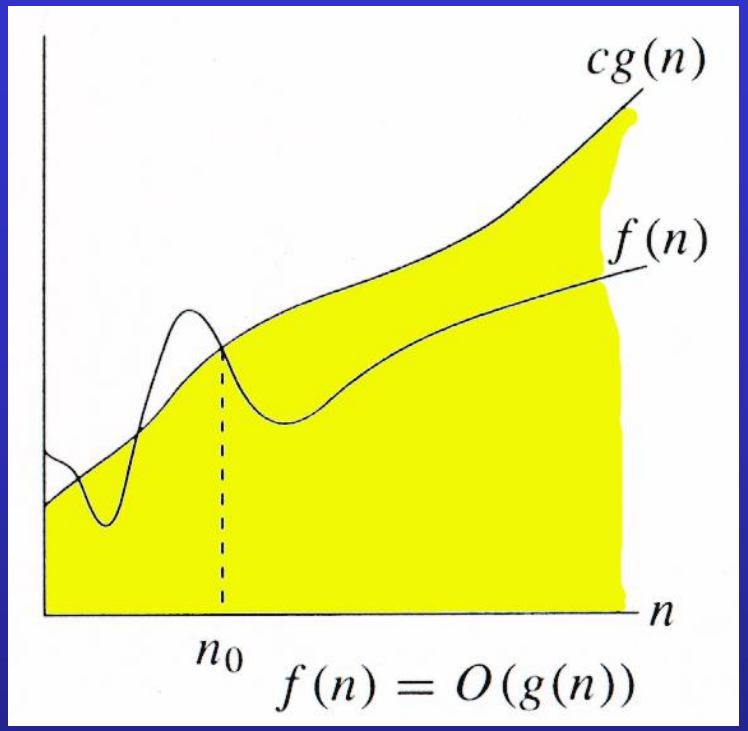
O-Notation

For a given function g(n)

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

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)*.

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

Example

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

$f(n) = 5n+2$

$0 \le 5n+2 \le 6n$

$n=0 \rightarrow 0 \le 2 \le 0$ (no)

$n=1 \rightarrow 0 \le 7 \le 6$ (no)

$n=2 \rightarrow 0 \le 12 \le 12$ (yes)

$n=3 \rightarrow 0 \le 17 \le 18$ (yes)

So $n_0=2$, $c=6$, $g(n)=n$

$\Rightarrow$ **$f(n) = O(n)$**

Big-O Notation (Examples)

- $f(n) = 5n+2 = O(n)$
- $f(n)=n/2 -3$

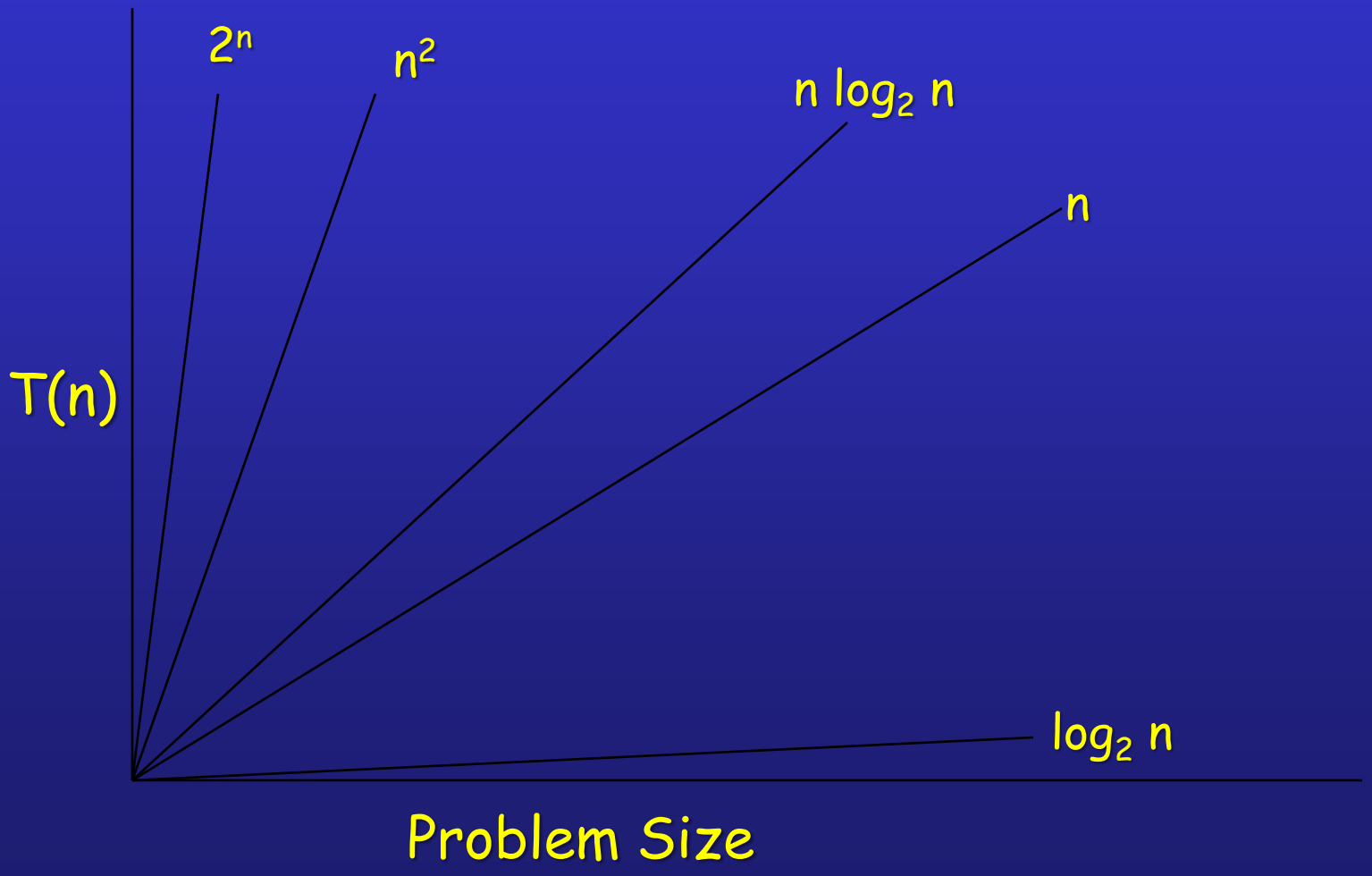
$0 \leq n/2 -3 \leq n/2$; $n_0=6$; $c=1/2$; $f(n)=O(n)$

- $f(n) = n^2-n$

$0 \leq n^2-n \leq n^2$; $n_0=0$; $c=1$; $f(n)=O(n^2)$

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

Comparing Growth Rates



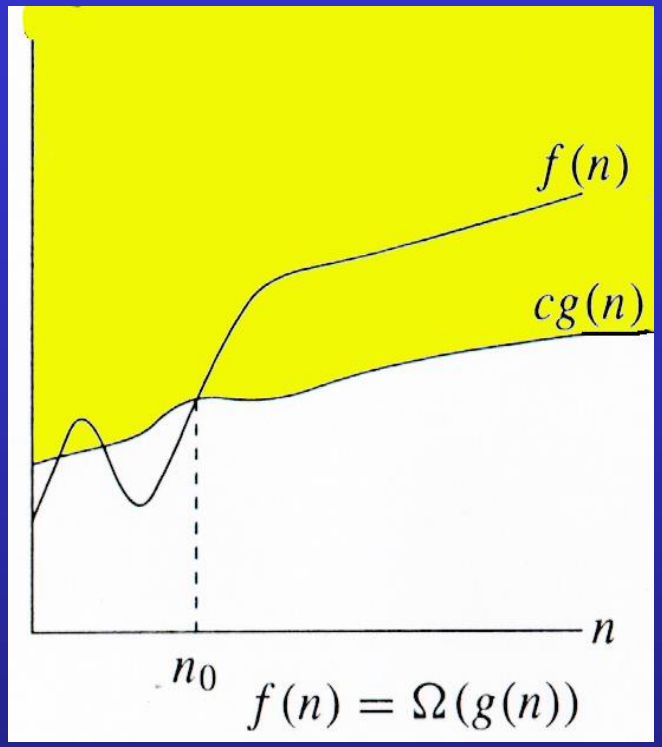
Ω - Notation

For a given function g(n)

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

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

Ω - Notation



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

⇒ f(n) = Ω(g(n)).

Example

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

f(n) = 5n+2

$0 \leq 5n \leq 5n+2$

n=0 → 0≤0≤2 (yes)

n=1 → 0≤5≤7 (yes)

n=2 → 0≤10≤12 (yes)

So n_0=0, c=5, g(n)=n

⇒ **f(n)= Ω(n)**

Θ - Notation

For a given function g(n),

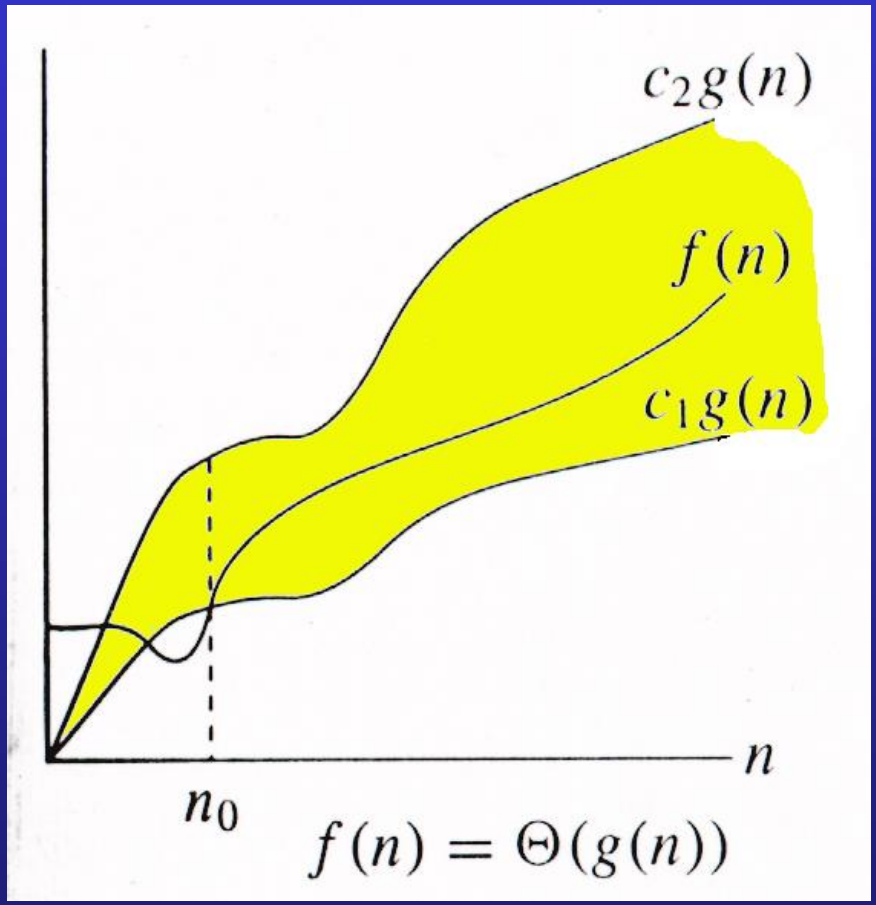
$\Theta(g(n))$ = {f(n) : there exist positive constants c_1 , c_2 , and n_0 such that $0 \leq c_1\, g(n) \leq f(n) \leq c_2\, g(n)$ for all $n \geq n_0$

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

Θ - Notation

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

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



Example

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

- $5n+2 = \Theta(n)$

Determine the positive constant n_0, c_1, and c_2 such that the above conditions satisfies

$$5n \leq 5n+2 \leq 6n$$

Example Contd …

$5n \leq 5n+2$ *is true for all* $n \geq 0$

$5n+2 \leq 6n$ **is true for all** $n \geq 2$

$\Rightarrow$ $5n \leq 5n+2 \leq 6n$ is true for all $n \geq 2$

$\Rightarrow c1=5,\ c2=6,\ n_0=2,\ g(n)=n$

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

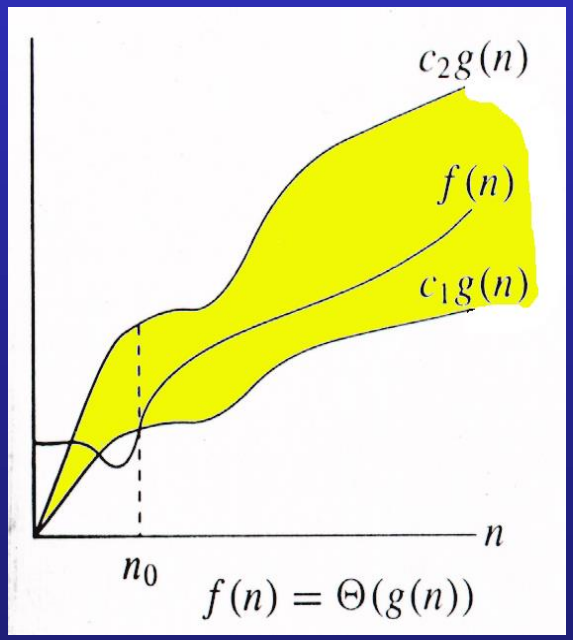
Relations Between Θ, O, Ω

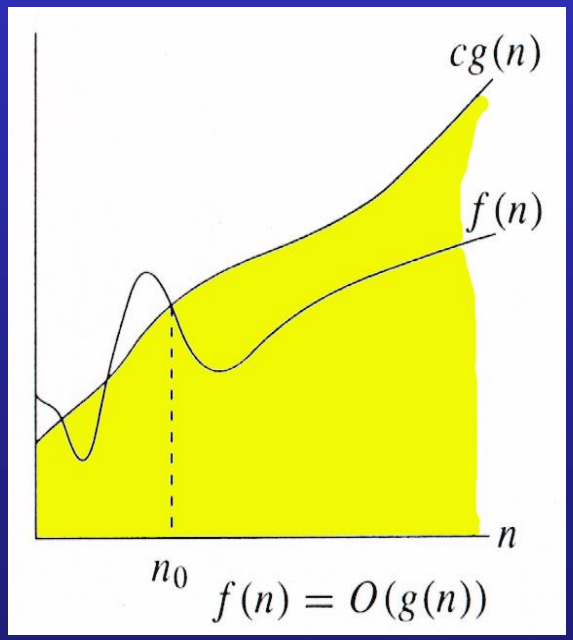
For any two function f(n) and g(n), we have $f(n) = \Theta(g(n))$ if and only if $f(n) = O(g(n))$ and $f(n) = \Omega(g(n))$

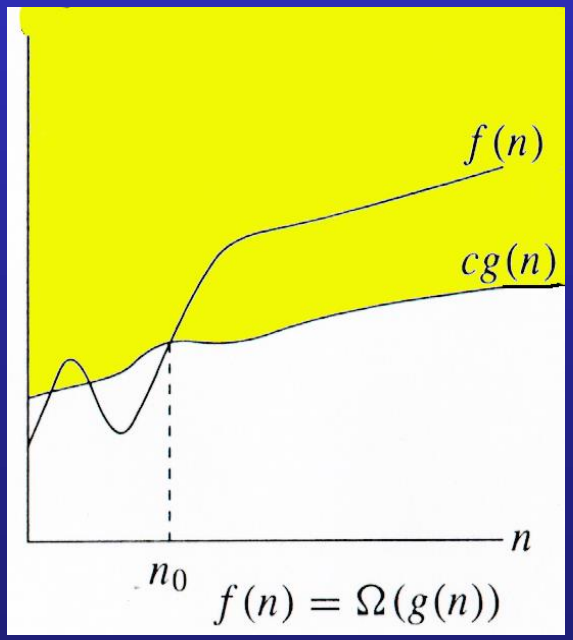
That is

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

Relations Between Θ, O, Ω







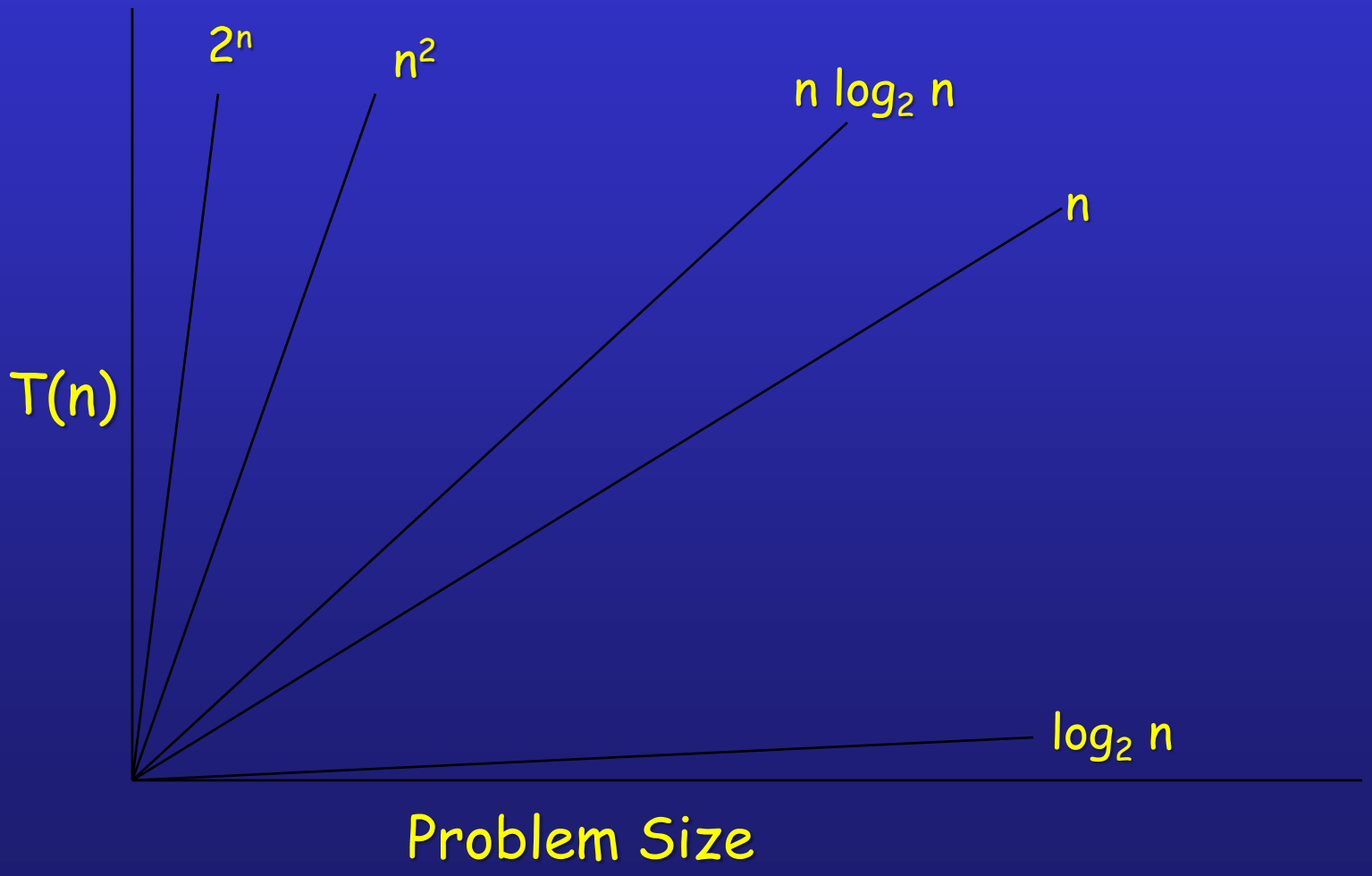
The Growth of Functions

"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 = f(n)

Example: Find max element of an array

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

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