Algorithm Efficiency

Natasha Dejdumrong

CPE 212 Algorithm Design (1/61)

Topics

- Growth function
- Asymptotic notations
- Comparing growth order
- Algorithm complexity
- Running time analysis of recursive algorithms

Measuring Efficiency

Time efficiency

Memory (space) efficiency

- Use "**running time**" f(n) or f(n, m, ...) of algorithm
  - Almost all algorithms run longer on larger input size n.
  - Can also be some parameters n, m specific to the input.
  - Logical to measure algorithm efficiency using running time

RAM Model of Computation

- Each “**basic operation**" (+, -, *, /, comparison) takes 1 step (time unit).
- Loops and subroutine calls composed of many single-step operations.
- Each memory access takes exactly 1 step.

Why RAM Model?

- Multiplication should take more time than addition.
- Access time for data on disk and on cache definitely different.

- But RAM model captures essential behaviour of how algorithm performs.
  - Flat-earth model useful and make sense in certain context.
  - Why not measuring real time?

Running Time Analysis of Non-Recursive Algorithms

- Count # basic operations taken on a given problem instance.
  - Usually most time-consuming operations occur in inner-loops
  - Independent of machine and language.
- Operation count reflects naturally to actual run time.

Ex: MaxElement()

- Determine the running time of the algorithm that finds the maximum element in a finite sequence.

Algorithm MaxElement

Input: A sequence of numbers $a_1, a_2, \cdots, a_n$
Output: The largest element in the input sequence

$maxval \leftarrow a_1$
for $i = 2$ to n **do**
    if $maxval < a_i$ **then**
        $maxval \leftarrow a_i$
Return $maxval$ $\triangleright$ $maxval$ is the largest element

Ex: UniqueElement()

Algorithm UniqueElement

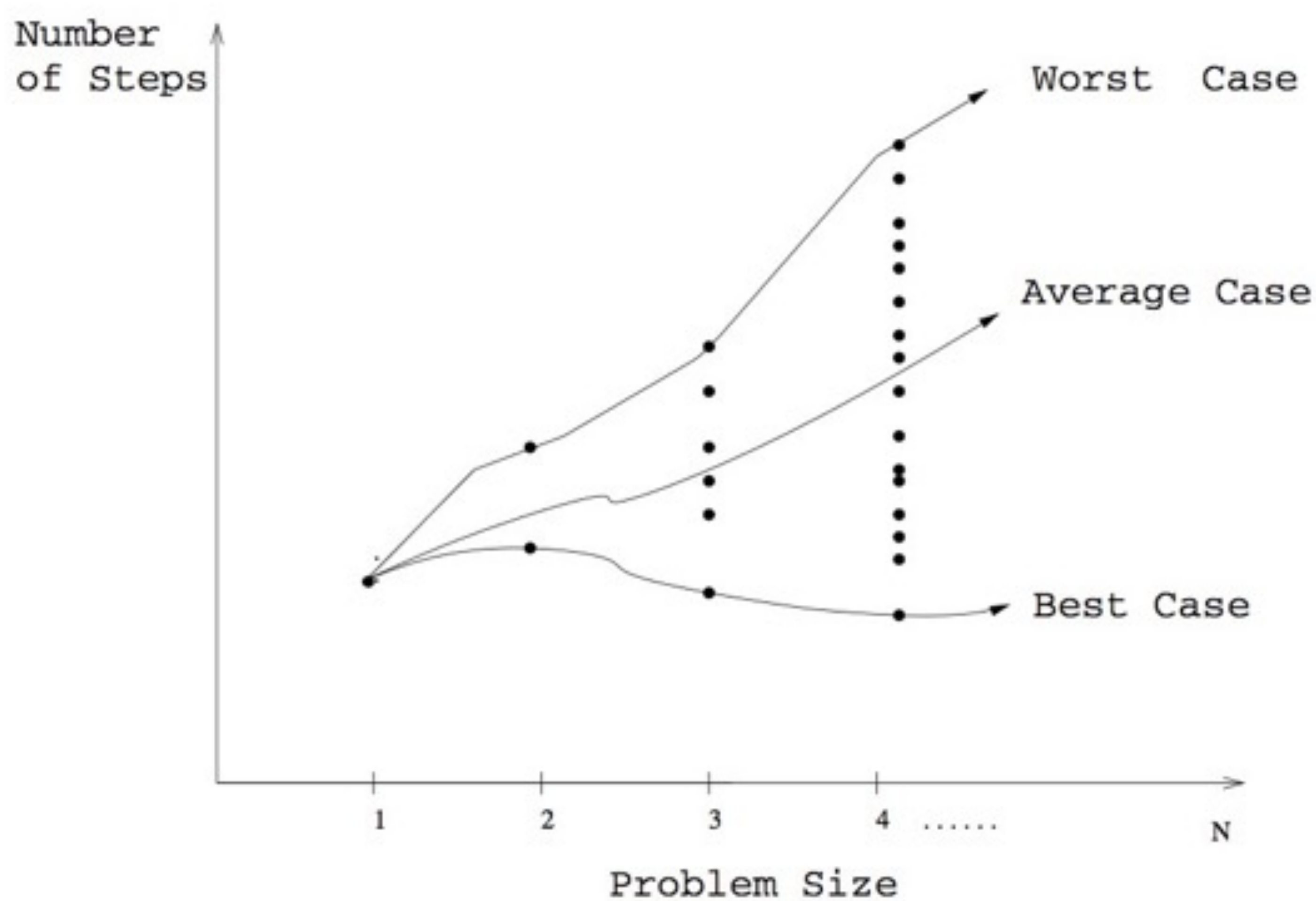
```
Input: A sequence of numbers a_1, a_2, ···, a_n
Output: Return "true" if all elements are distinct and "false" otherwise.

for i = 1 to n − 1 do
    for j = i + 1 to n do
        if a_i == a_j then
            Return false
Return true
```

Best case C(n) = ?

Worst case C(n) = ?

Best-/ Average-/ Worst-case

- Actual running time can depend on both input and algorithm.
- Must know how an algorithm works over all instances



$$C_{worst}(n) = \sum_{i=0}^{n-2} \sum_{j=i+1}^{n-1} 1 = \sum_{i=0}^{n-2} [(n-1) - (i+1) + 1] = \sum_{i=0}^{n-2} (n-1-i)$$

$$= \sum_{i=0}^{n-2} (n-1) - \sum_{i=0}^{n-2} i = (n-1) \sum_{i=0}^{n-2} 1 - \frac{(n-2)(n-1)}{2}$$

$$= (n-1)^2 - \frac{(n-2)(n-1)}{2} = \frac{(n-1)n}{2} \approx \frac{1}{2} n^2$$

We also could have computed the sum $\sum_{i=0}^{n-2} (n-1-i)$ faster as follows:

$$\sum_{i=0}^{n-2} (n-1-i) = (n-1) + (n-2) + \cdots + 1 = \frac{(n-1)n}{2},$$

1. $\sum_{i=l}^{u} 1 = \underbrace{1+1+\cdots+1}_{u-l+1 \text{ times}} = u - l + 1$ (l, u are integer limits, $l \le u$); $\sum_{i=1}^{n} 1 = n$

2. $\sum_{i=1}^{n} i = 1 + 2 + \cdots + n = \frac{n(n+1)}{2} \approx \frac{1}{2}n^2$

3. $\sum_{i=1}^{n} i^2 = 1^2 + 2^2 + \cdots + n^2 = \frac{n(n+1)(2n+1)}{6} \approx \frac{1}{3}n^3$

4. $\sum_{i=1}^{n} i^k = 1^k + 2^k + \cdots + n^k \approx \frac{1}{k+1}n^{k+1}$

5. $\sum_{i=0}^{n} a^i = 1 + a + \cdots + a^n = \frac{a^{n+1}-1}{a-1}$ $(a \ne 1)$; $\sum_{i=0}^{n} 2^i = 2^{n+1} - 1$

6. $\sum_{i=1}^{n} i2^i = 1 \cdot 2 + 2 \cdot 2^2 + \cdots + n2^n = (n-1)2^{n+1} + 2$

7. $\sum_{i=1}^{n} \frac{1}{i} = 1 + \frac{1}{2} + \cdots + \frac{1}{n} \approx \ln n + \gamma$, where $\gamma \approx 0.5772\ldots$ (Euler's constant)

8. $\sum_{i=1}^{n} \lg i \approx n \lg n$

Ex: Linear (Sequential) Search

Algorithm LinearSearch

```
Input: A sequence of numbers a_1, a_2, ..., a_n; An element x
Output: Return the index of element if x is in the sequence, and 0 otherwise.

loc ← 0
for i = 1 to n do
    if x = a_i then
        loc ← i
        break
Return loc
```

General Plan for Analyzing Non-Recursive Algorithms

Step	Ex: UniqueElement()
Decide on parameters indicating an input's size.	
Identify algorithm's basic operations	
Check if basic operations depend on additional properties other than the input size.	
Set up the sum expressing # executions of basic operations.	
Evaluate the sum to closed-form (or establish growth order)	

Efficient vs. Inefficient Algorithms

for $i = 1$ to n **do**

$x_1 \leftarrow x_1 + 1$

$x_2 \leftarrow x_2 + 1$

$x_3 \leftarrow x_3 + 1$

$\vdots$

$x_{100} \leftarrow x_{100} + 1$

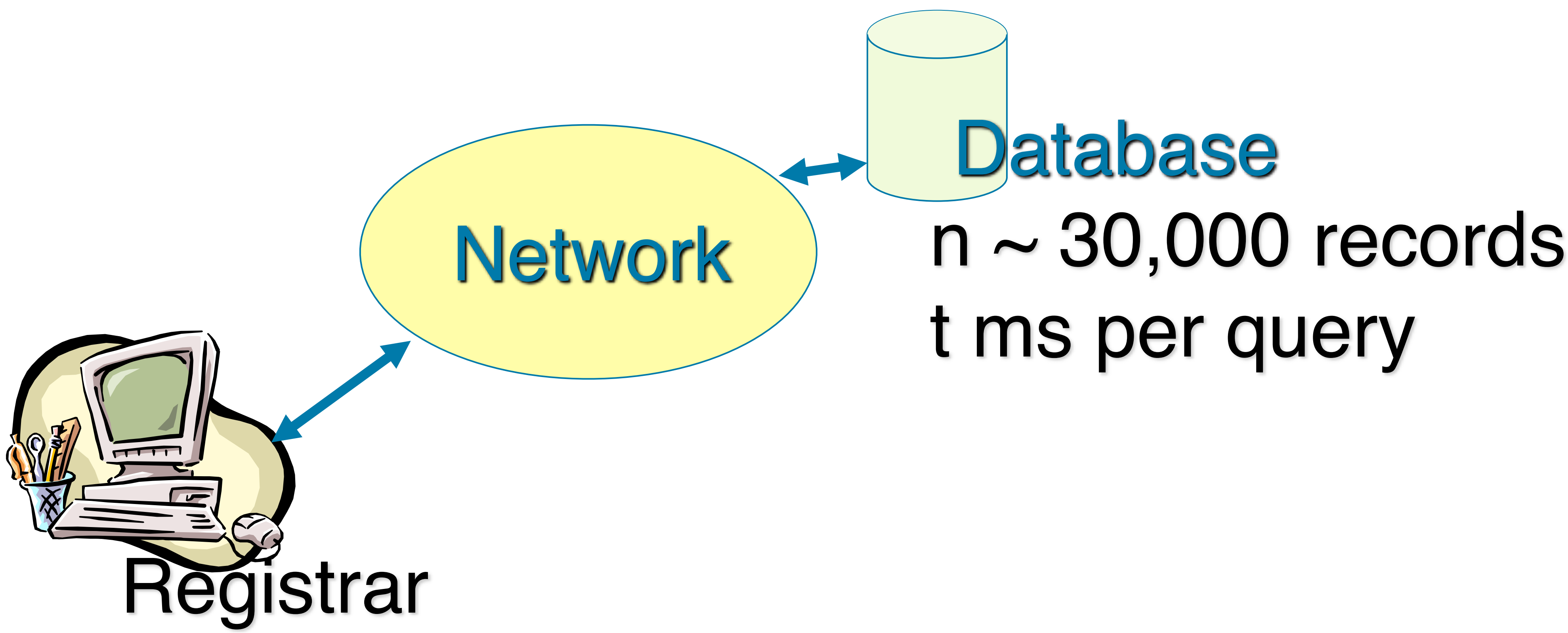
for $i = 1$ to n **do**

for $j = 1$ to n **do**

$x_1 \leftarrow x_1 + 1$

$x_2 \leftarrow x_2 + 1$

- What is the efficiency of each algorithm above?
- Which one run faster?



How long to wait on average?

No. of records (n)	Waiting times (10 msec per search)	
	Linear search (n)	Binary search ($\log_2 n$)
n = 1000	10 s	0.1 s
n = 10000	100 s	0.1329 s
n = 30000	300	0.1487 s

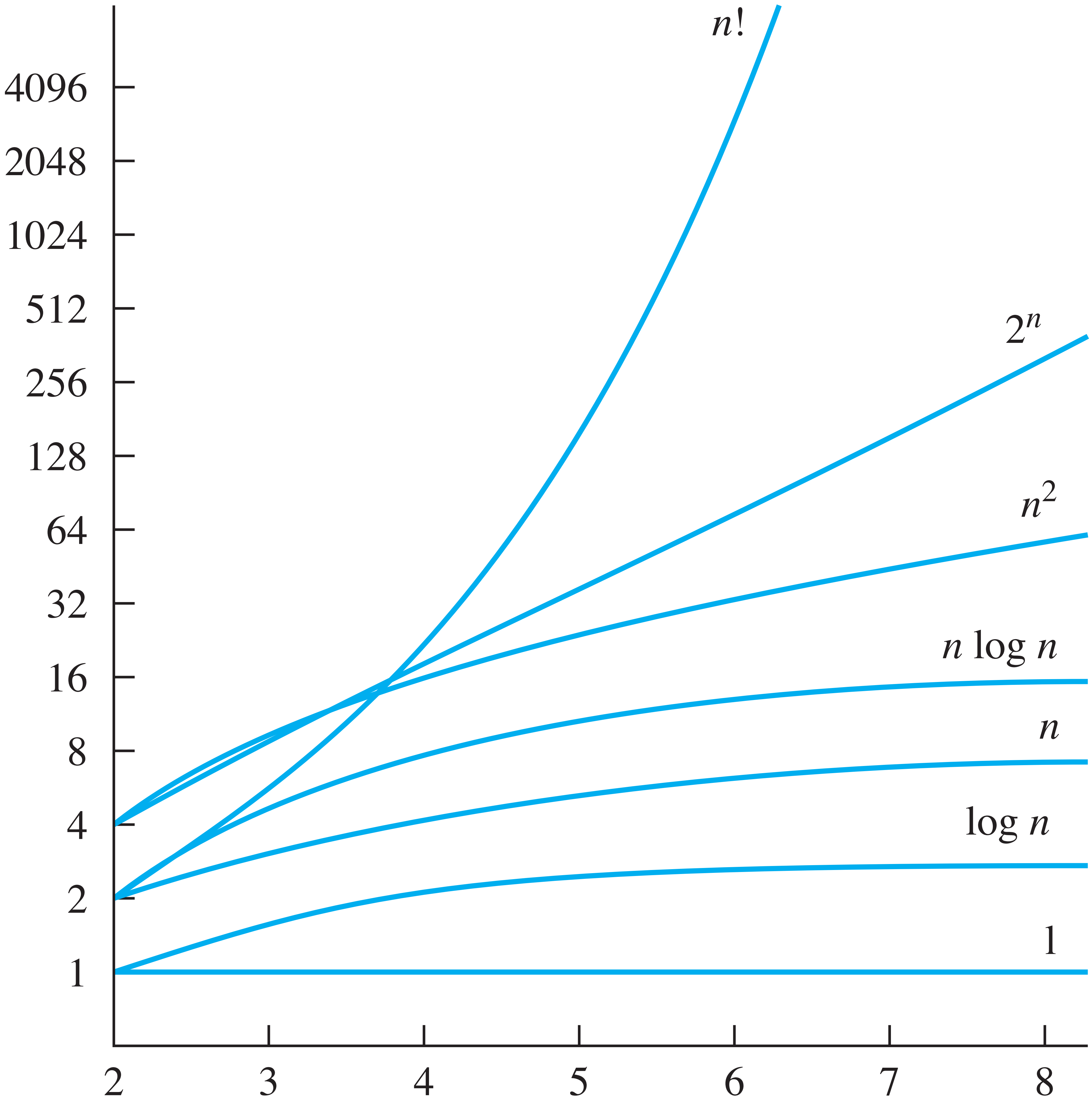
No. of records (n)	Waiting times (1 msec per search)	
	Linear search (n)	Binary search ($\log_2 n$)
n = 1000	1 s	0.01 s
n = 10000	10 s	0.01329 s
n = 30000	30 s	0.01487 s

Lessons Learned

- Running time on small inputs typically does not matter.
- Only when large input n can we distinguish between efficient and inefficient algorithms.

Growth Order and Algorithm

- Consider an algorithm with $f(n) = n^2/4 + n/2 - 3$.
  - What are f(n) at n = 4, 40, 400, and 4,000 ?
  - f(n) behaves like n^2 for large n (within a constant multiple)
  - f(n) has the **growth order** of n^2
- Growth order is an indicator of algorithm efficiency
  - Used to rank and compare algorithms.
  - Determine if it is practical to use a particular algorithm as the input grows.



n!
4096
2048
1024
512
256
128
64
32
16
8
4
2
1
2^n
n^2
n log n
n
log n
1
2
3
4
5
6
7
8

Common Growth Order

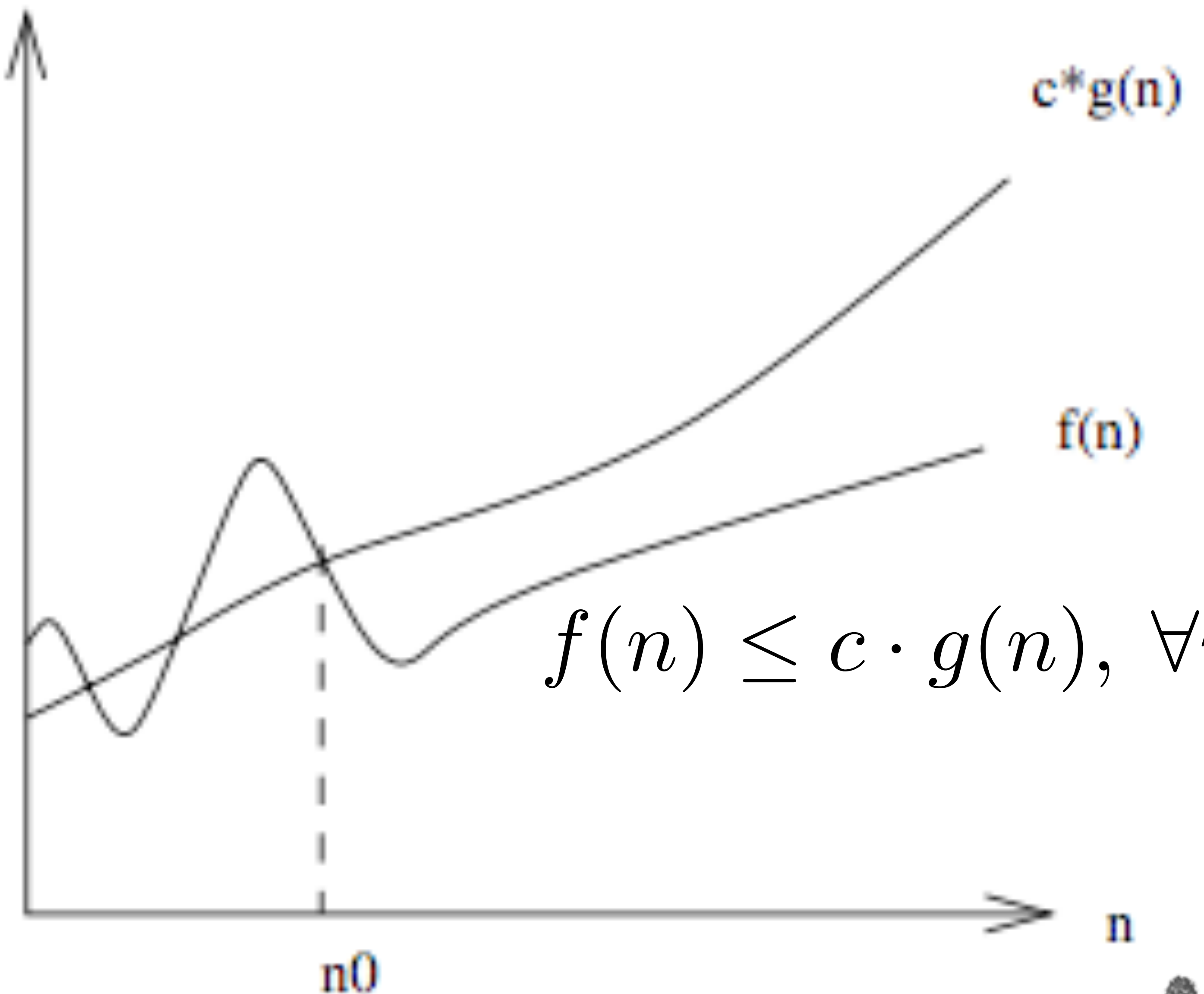
Functions	Name
$f(n) = 1$	Constant function
$f(n) = \log n$	Logarithmic function
$f(n) = n$	Linear function
$f(n) = n \log n$	Linearithmic/Superlinear function
$f(n) = n^b$	Polynomial function
$f(n) = b^n$	Exponential function
$f(n) = n!$	Factorial function

n $f(n)$	$\lg n$	n	$n \lg n$	n^2	2^n	$n!$
10	0.003 μs	0.01 μs	0.033 μs	0.1 μs	1 μs	3.63 ms
20	0.004 μs	0.02 μs	0.086 μs	0.4 μs	1 ms	77.1 years
30	0.005 μs	0.03 μs	0.147 μs	0.9 μs	1 sec	8.4×10^{15} yrs
40	0.005 μs	0.04 μs	0.213 μs	1.6 μs	18.3 min	
50	0.006 μs	0.05 μs	0.282 μs	2.5 μs	13 days	
100	0.007 μs	0.1 μs	0.644 μs	10 μs	4×10^{13} yrs	
1,000	0.010 μs	1.00 μs	9.966 μs	1 ms		
10,000	0.013 μs	10 μs	130 μs	100 ms		
100,000	0.017 μs	0.10 ms	1.67 ms	10 sec		
1,000,000	0.020 μs	1 ms	19.93 ms	16.7 min		
10,000,000	0.023 μs	0.01 sec	0.23 sec	1.16 days		
100,000,000	0.027 μs	0.10 sec	2.66 sec	115.7 days		
1,000,000,000	0.030 μs	1 sec	29.90 sec	31.7 years		

Asymptotic Efficiency Class

- Provide ***asymptotic bounds*** (upper, lower, exact) on the running time of algorithms
  - Big-Oh notation
  - Big-Omega notation
  - Big-Theta notation
- Allow for comparing and ranking growth orders of algorithms solving the same problem.

Big-Oh Notation : Upperbound



$$f(n) \in O(g(n))$$

$$f(n) \leq c \cdot g(n), \forall n \geq n_0$$

- "f(n) is big-Oh of g(n)"
- "f(n) has the growth order bounded above by g(n)"
- "O(g(n)) is a big-Oh estimate of f(n)"

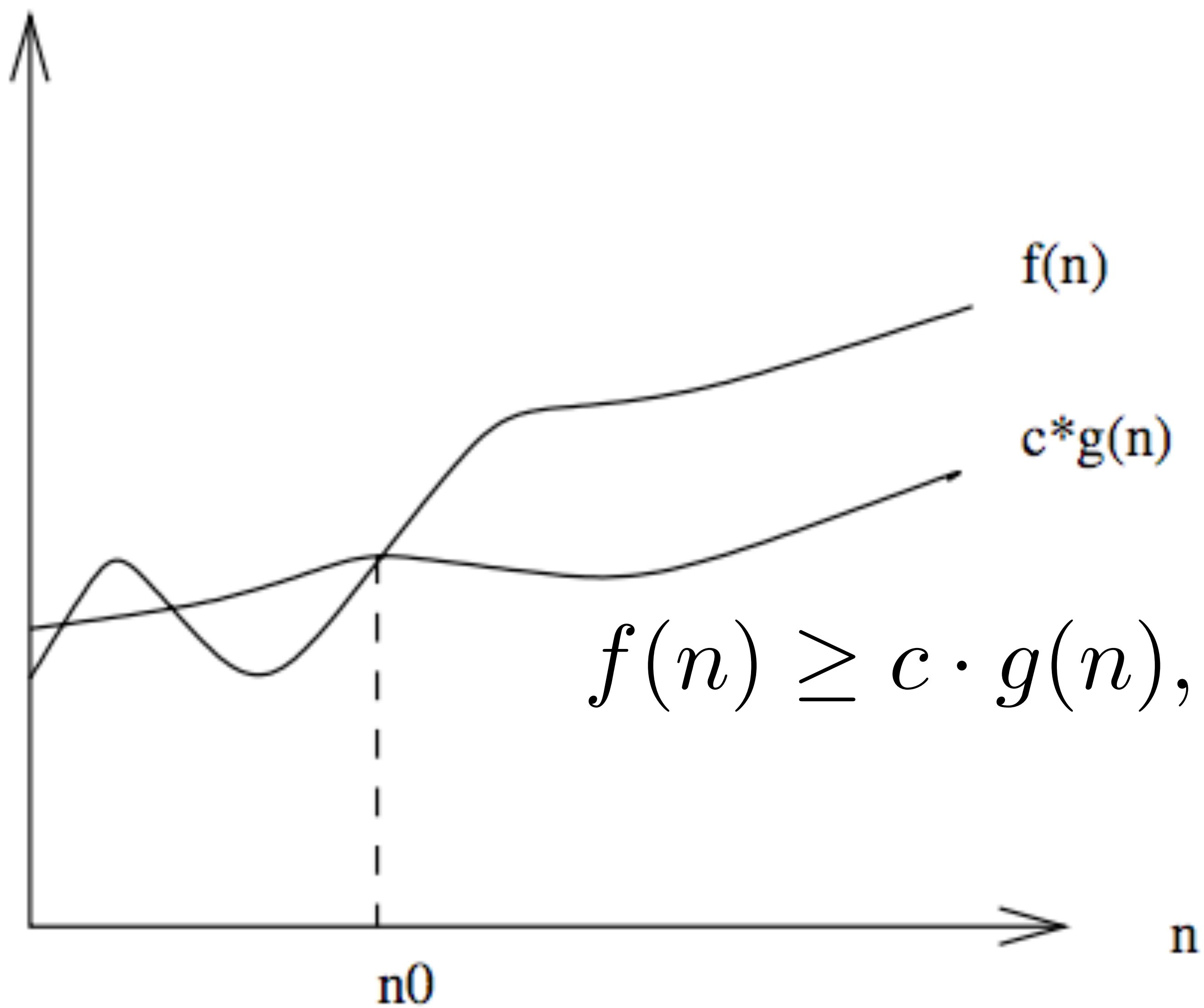
- Let f and g be functions from the set of integers or real numbers to the set of real numbers.
- *We say that **f(n) is O(g(n))** or **f(n) $\in$ O(g(n))** if f(n) is **bounded above** by some constant multiple of g(n) for all large n.*

$$f(n) \leq c \cdot g(n), \; \forall n \geq n_0$$

  - Many g(n) possible for the same f(n)
  - g(n) preferably selected as small as possible.

- Ex: Show that $7x^2$ is $O(x^3)$.
  - When $x > 7$, $7x^2 < x^3$.
  - Take $c = 1$ and $x_0 = 7$ as witnesses to establish that $7x^2$ is $O(x^3)$.
  - Would $c = 7$ and $x_0 = 1$ work?
- Ex: Show that n^2 is not $O(n)$.
  - Suppose there are constants C and n_0 for which $n^2 \leq C_n$, whenever $n > k$.
  - Then (by dividing both sides of $n^2 \leq C_n$) by n, then $n \leq C$ must hold for all $n > n_0$. A contradiction!

Big-Ω Notation: Lowerbound



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

$$f(n) \geq c \cdot g(n),\ \forall n \geq n_0$$

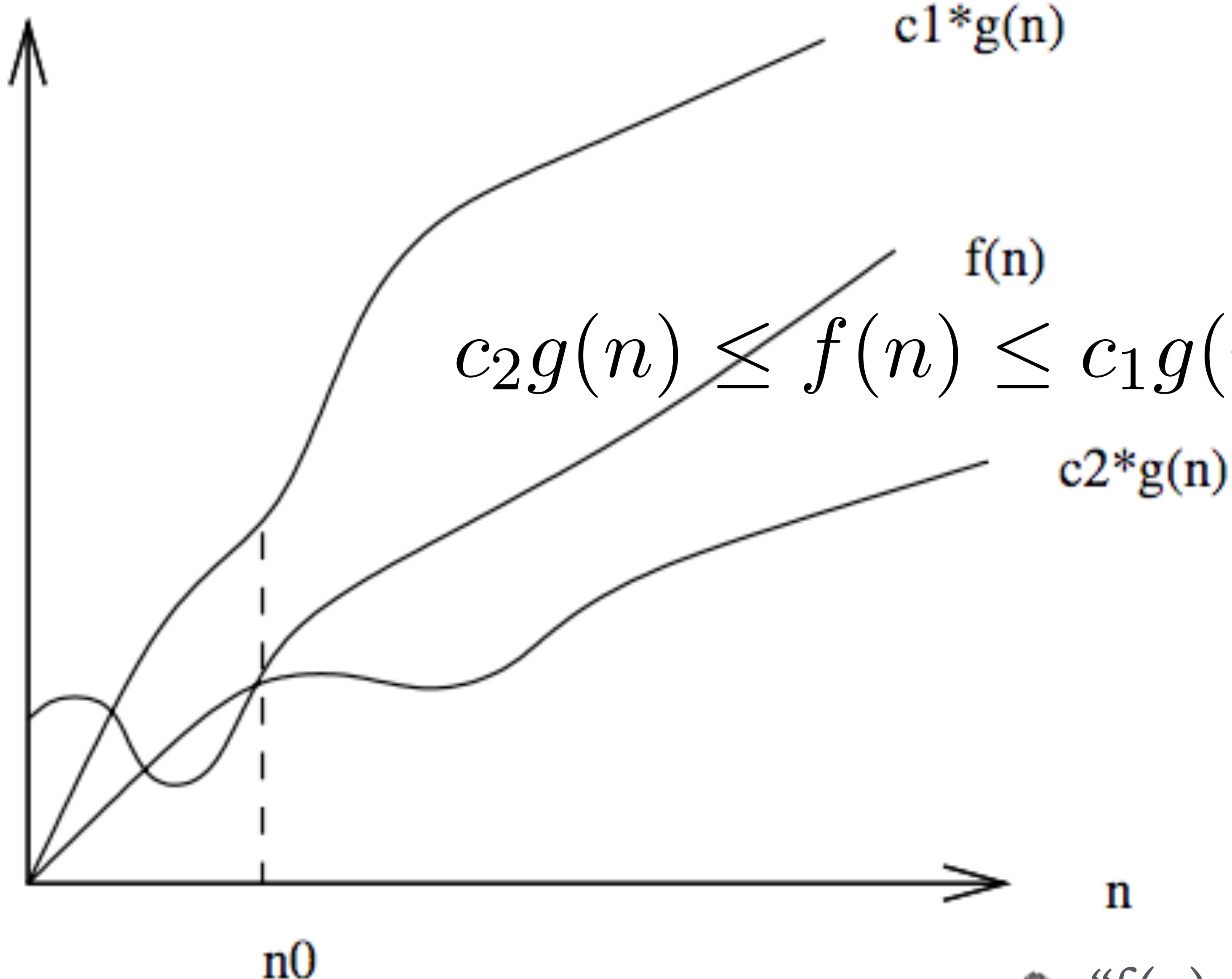
- "f(n) is big-Omega of g(n)"
- "f(n) has the growth order bounded below by g(n)"
- "Ω(g(n)) is a big-Omega estimate of f(n)"

- Let f and g be functions from the set of integers or real numbers to the set of real numbers.
- *Say that **f(n) is Ω(g(n))** or **f(n) ∈ Ω(g(n))** if f(n) is **bounded below** by some constant multiple of g(n) for all large n.*

$$f(n) \geq c \cdot g(n),\ \forall n \geq n_0$$

- Ex: Show that n^3 is $\Omega(n^2)$

Big-Θ Notation: Exact bound



$$\boxed{f(n) \in \Theta(g(n))}$$

$$c_2 g(n) \leq f(n) \leq c_1 g(n),\ \forall n \geq n_0$$

- “f(n) is big-Theta of g(n)”
- “f(n) has the growth order bounded above and below by g(n)”
- “Θ(g(n)) is a big-Oh estimate of f(n)”

- Let f and g be functions from the set of integers or real numbers to the set of real numbers.
- *We say that $f(x)$ is $\Theta(g(n))$ or $f(n) \in \Theta(g(n))$ if $f(n)$ is **bounded both above and below** by some constant multiple of $g(n)$ for all large n.*

$$c_2 g(n) \leq f(n) \leq c_1 g(n), \; \forall n \geq n_0$$

- Most preferred expression for algorithm efficiency.

- Ex: Show that $f(x) = 3x^2 + 8x \log x$ is $\Theta(x^2)$.
  - $3x^2 + 8x \log x \leq 11x^2$ for $x > 1$, since $0 \leq 8x \log x \leq 8x^2$.
  - Hence, $3x^2 + 8x \log x$ is $O(x^2)$.
  - x^2 is clearly $O(3x^2 + 8x \log x)$
  - Hence, $3x^2 + 8x \log x$ is $\Theta(x^2)$.

- Big-Theta is the most preferred expression for algorithm efficiency. Why?

- Note: $f(n)$ is $\Theta(g(n))$ if one of the following is true:
  - $f(n) = O(g(n))$ and $g(n) = O(f(n))$
  - $f(n) = O(g(n))$ and $f(n) = \Omega(g(n))$

Algorithm Complexity

- *If the running time $f(n)$ is $\Theta(g(n))$, the algorithm is said to have the* ***complexity*** */* ***asymptotic efficiency*** *of $\Theta(g(n))$.*

- What is the complexity of
  - MaxElement() algorithm ?
  - Linear search algorithm ?

- Is $\Theta(n)$ algorithm always faster than $\Theta(n^2)$ algorithm?

Basic Asymptotic Efficiency Classes

Class g(n)	Name
1	Constant
$\log n$	Logarithmic
n	Linear
$n \log n$	Linearithmic / Superlinear
$\sqrt{n}$	Sublinear polynomial
n^b	Polynomial
b^n	Exponential
$n!$	Factorial

Comparing Growth Orders

- Given two efficiency classes $g_1(n)$ and $g_2(n)$, how do we know which one grows faster?
- Given running time f(n) and class g(n), how do we check if f(n) is Big-? of g(n).

- Two growth orders can be compared by the following ratio of limits:

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = \begin{cases} 0, & f(n) \text{ has a smaller order of growth than } g(n) \\ c, & f(n) \text{ has the same order of growth than } g(n) \\ \infty, & f(n) \text{ has a larger order of growth than } g(n) \end{cases}$$

- Ex: Compare the orders of growth of
  - $(1/2)n(n-1)$ and n^2
  - $\log_2 n$ and $\sqrt{n}$
  - $n!$ and 2^n

Dominance Relations

- Faster-growing function is said to dominate a slower-growing one
- Suppose f and g belong to different classes, e.,g., $f(n) \notin \Theta(g(n))$
- We say that ***g dominates f if $f(n) \in O(g(n))$, or $g \gg f$***
    - So, g dominates f if it has a larger order of growth.

$$n! \gg c^n \gg n^3 \gg n^2 \gg n^{1+\epsilon} \gg n \log n \gg n \gg \sqrt{n} \gg$$
$$\log^2 n \gg \log n \gg \log n / \log\log n \gg \log\log n \gg \alpha(n) \gg 1$$

Running Time Analysis of Recursive Algorithms

Algorithm Factorial(n)

```
Input: A non-negative integer n
Output: The value of n!

if n = 0 then
    return 1
else
    return Factorial(n − 1)∗n
```

$$F(n) = \begin{cases} F(n-1)\cdot n & \text{for } n > 0 \\ 1 & \text{for } n = 0 \end{cases}$$

- M(n), # basic operations (multiplication), has the recurrence relation:

$$M(n) = \begin{cases} M(n-1)+1 & \text{for } n > 0 \\ 0 & \text{for } n = 0 \end{cases}$$

Method of Backward Substitutions

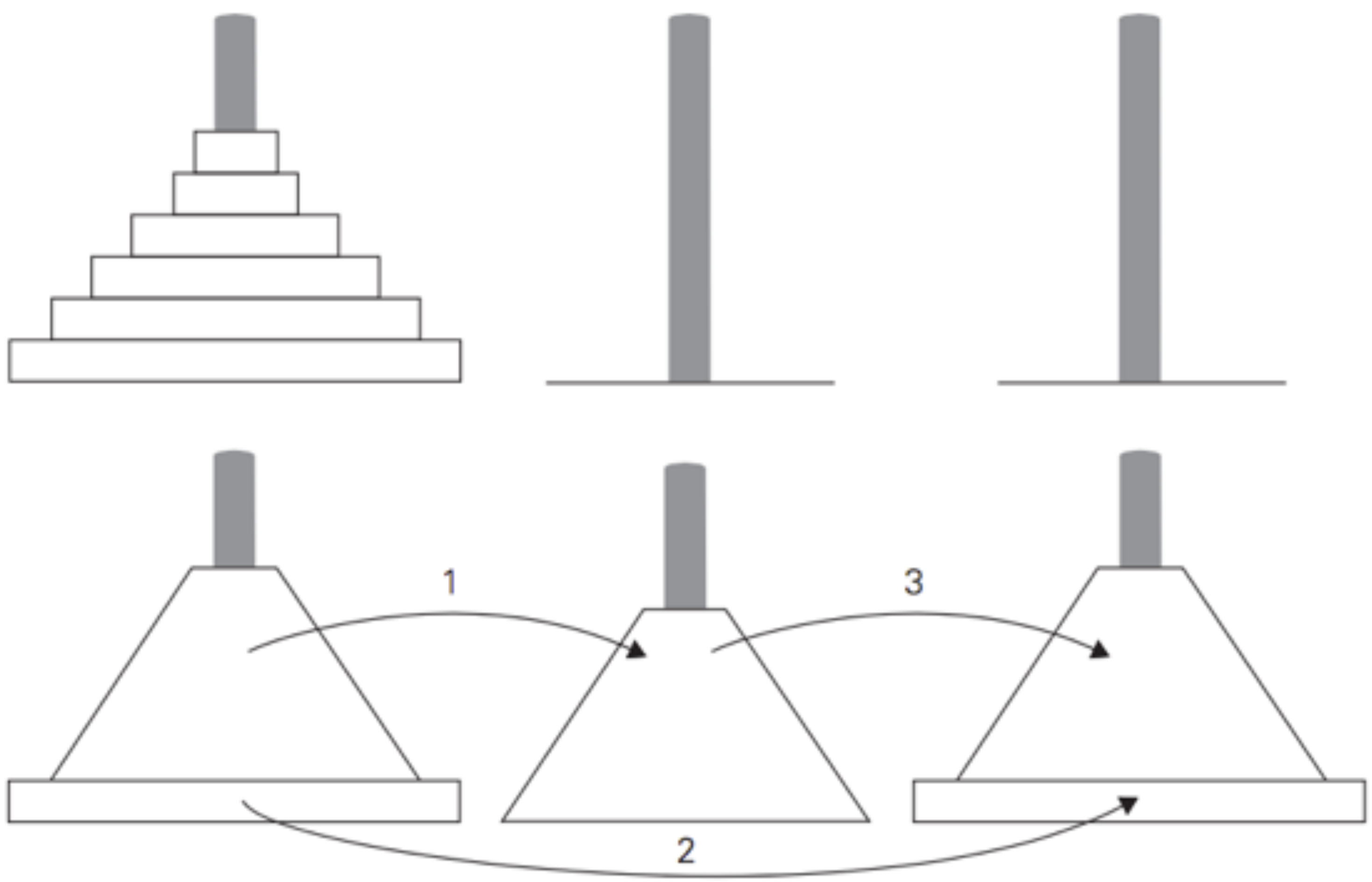
$$
\begin{aligned}
M(n) &= M(n-1)+1 && \text{substitute } M(n-1)=M(n-2)+1\\
&= [M(n-2)+1]+1 = M(n-2)+2 && \text{substitute } M(n-2)=M(n-3)+1\\
&= [M(n-3)+1]+2 = M(n-3)+3.
\end{aligned}
$$

- Therefore,

$$M(n) = M(n-1)+1 = \cdots = M(n-i)+i = \cdots = M(n-n)+n = n.$$

Ex: Tower of Hanoi Puzzle

- Move all disks from left peg to right peg one-by-one
  - Middle peg used as auxilliary
  - Forbidden to to place a larger disk on a smaller one.



- Basic operation = One disk movement
- So, M(n) = M(n-1) + 1 + M(n-1), for n > 1

$$M(n) = \begin{cases} 2M(n-1)+1 & \text{for } n > 1 \\ 1 & \text{for } n = 1 \end{cases}$$

- Applying the method of backward substitutions:

$$\begin{aligned} M(n) &= 2M(n-1)+1 && \text{sub. } M(n-1) = 2M(n-2)+1 \\ &= 2[2M(n-2)+1]+1 = 2^2M(n-2)+2+1 && \text{sub. } M(n-2) = 2M(n-3)+1 \\ &= 2^2[2M(n-3)+1]+2+1 = 2^3M(n-3)+2^2+2+1. \end{aligned}$$

- The pattern of the first three sums suggests that

$$M(n) = 2^i M(n-i) + 2^{i-1} + 2^{i-2} + \cdots + 2 + 1$$
$$= 2^i M(n-i) + 2^i - 1$$

- Since the initial condition is specified for n = 1, which is achieved for i = n-1,

$$M(n) = 2^{n-1} M(n-(n-1)) + 2^{n-1} - 1$$
$$= 2^{n-1} M(1) + 2^{n-1}$$
$$= 2^{n-1} + 2^{n-1} - 1 = 2^n - 1$$

General Plan for Analyzing Recursive Algorithms

- Decide on parameters indicating an input's size.
- Identify algorithm's basic operations
- Check if basic operations depend on additional properties other than the input size. If so, worst-case, average-case, best-case efficiencies have to be derived separately.
- Set up a recurrence relation with appropriate initial condition expressing # basic operations executed.
- Solve the recurrence to closed-form (or establish growth order)

Summary

- Algorithm efficiency measured by (worst-case) running time, which counts the basic operations
- Running time on large inputs distinguishes efficient algorithms from inefficient ones.
- Growth order of running time indicates the asymptotic efficiency of the algorithm
- Big-O and friends allow for classifying algorithms into efficiency classes.