

Fall 2017

Fang Yu

Software Security Lab.
Dept. Management Information
Systems,
National Chengchi University

Data Structures

Lecture 8

Recap

- What should you have learned?
 - Basic java programming skills
 - Object-oriented programming
 - Classes and objects
 - Inheritance, exception handling, generics
 - Java class library
 - Basic data structures and their applications
 - Linear data structure: linked list, array, stack, queue
 - Hierarchical data structures: tree and heap



Midterm on Dec. 7 (9:10-12:00am, 大勇樓105)

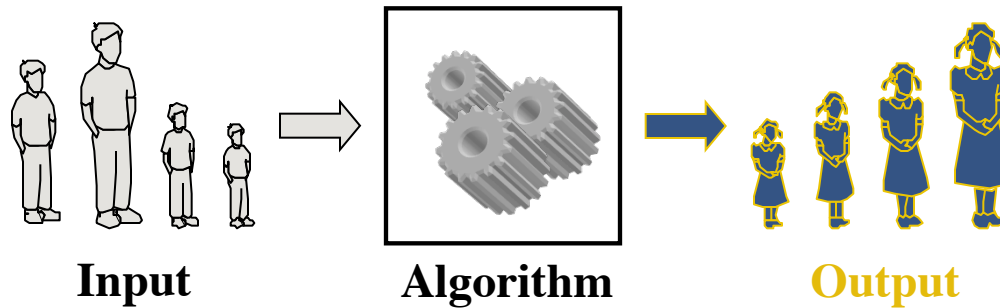


- Lec 1-9, TextBook Ch1-8, 11,12
- How to prepare your midterm:
 - Understand “ALL” the materials mentioned in the slides
 - Discuss with me, your TAs, or classmates
 - Read the text book to help you understand the materials
- You are allowed to bring an A4 size note
 - Prepare your own note; write whatever you think that may help you get better scores in the midterm

Wrap up

- What are you going to learn in the rest of this semester ?
 - Algorithms
 - Analysis of algorithms
 - Brute force, divide and conquer, dynamic programming
 - Sorting
 - Advance data structures
 - Hash table
 - Map and dictionary
 - Graph



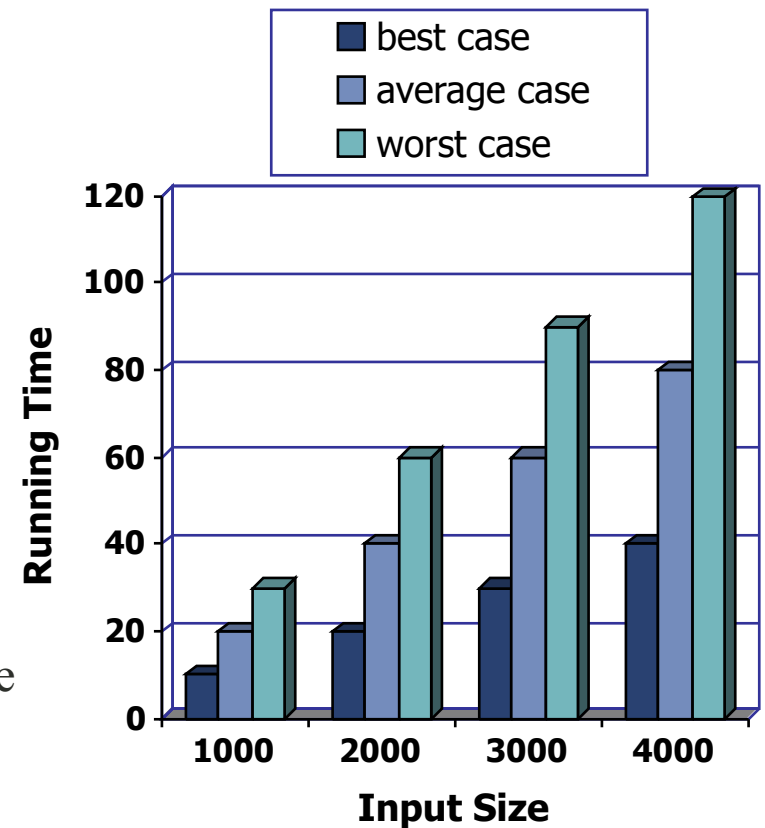


Analysis of Algorithms

How good is your program?

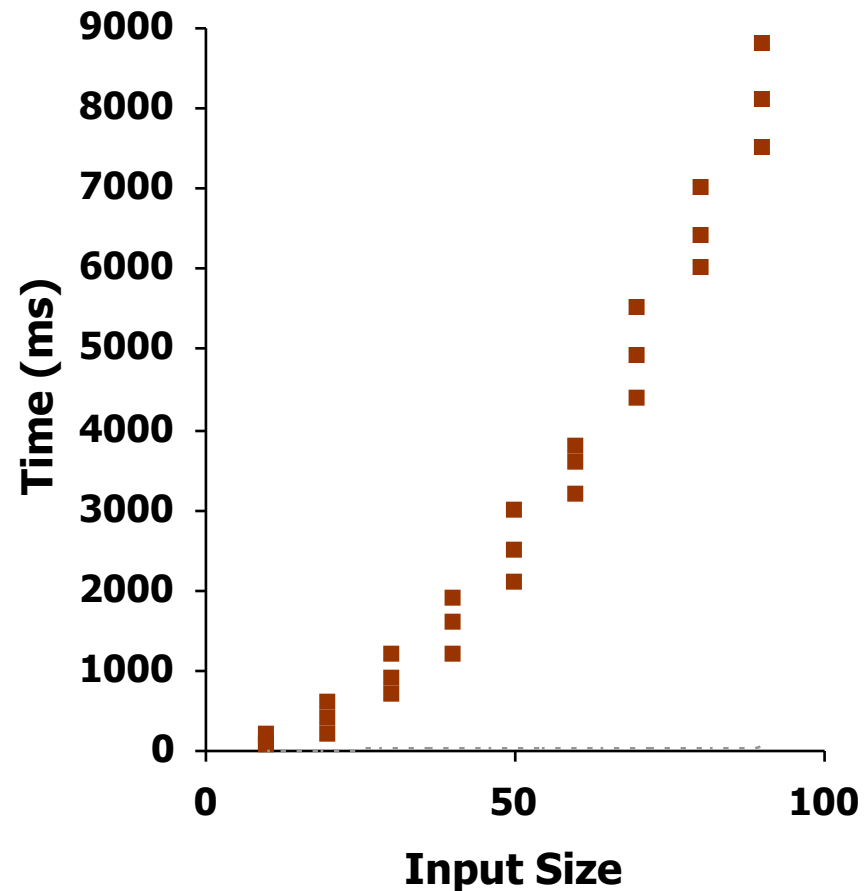
Running Time

- Most algorithms transform input objects into output objects.
- The running time of an algorithm typically grows with the input size.
- Average case time is often difficult to determine.
- We focus on the worst case running time.
 - Easier to analyze
 - Crucial to applications such as games, finance and robotics



Experimental Studies

- Write a program implementing the algorithm
- Run the program with inputs of varying size and composition
- Use a method like `System.currentTimeMillis()` to get an accurate measure of the actual running time
- Plot the results



Limitations of Experiments

- It is necessary to implement the algorithm, which may be difficult
- Results may not be indicative of the running time on other inputs not included in the experiment.
- In order to compare two algorithms, the same hardware and software environments must be used



Theoretical Analysis

- Uses a high-level description of the algorithm instead of an implementation
- Characterizes running time as a function of the input size, n .
- Takes into account all possible inputs
- Allows us to evaluate the speed of an algorithm independent of the hardware/software environment



Pseudocode

- High-level description of an algorithm
- More structured than English prose
- Less detailed than a program
- Preferred notation for describing algorithms
- Hides program design issues

Example: find max element of an array

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

Pseudocode Details

- Control flow
 - **if ... then ... [else ...]**
 - **while ... do ...**
 - **repeat ... until ...**
 - **for ... do ...**
 - Indentation replaces braces
- Method declaration

Algorithm *method* (*arg* [, *arg...*])

Input ...

Output ...



Pseudocode Details

- Method call

var.method (arg [, arg...])

- Return value

return *expression*

- Expressions

← Assignment

(like = in Java)

= Equality testing

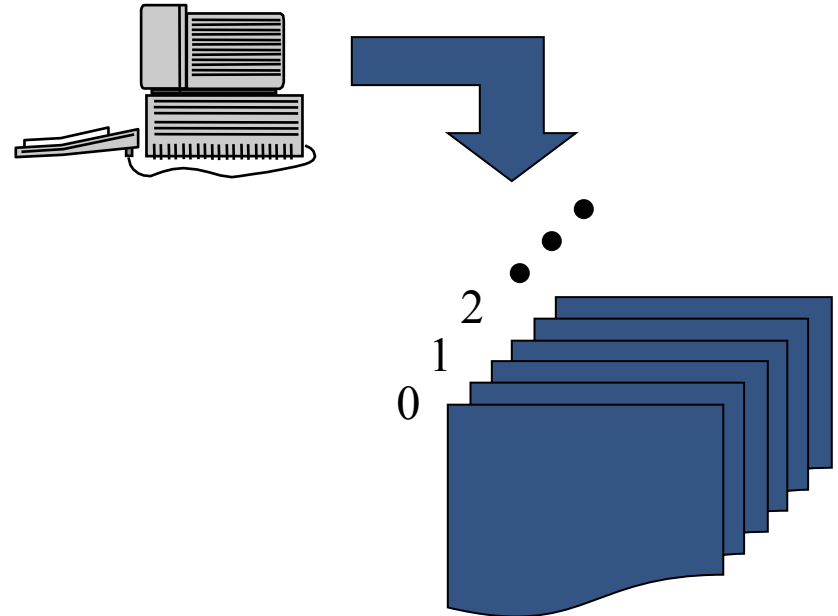
(like == in Java)

*n*² Superscripts and other mathematical formatting allowed

The Random Access Machine (RAM) Model



- A CPU
- An potentially unbounded bank of **memory** cells, each of which can hold an arbitrary number or character
- Memory cells are numbered and accessing any cell in memory takes unit time.

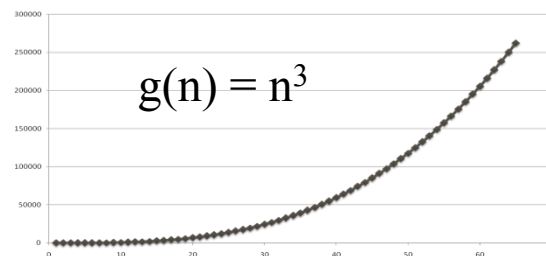
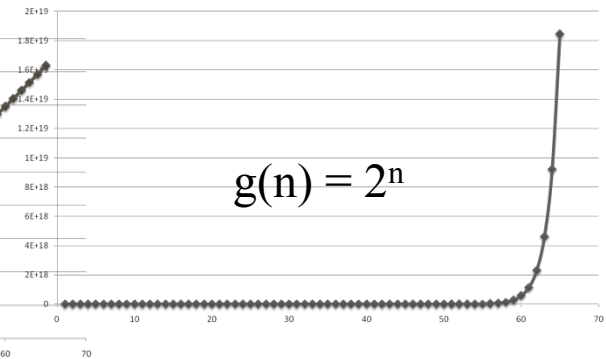
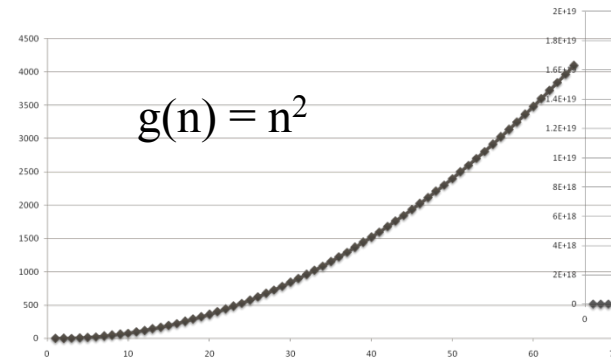
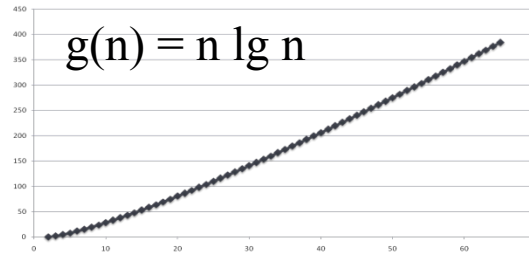
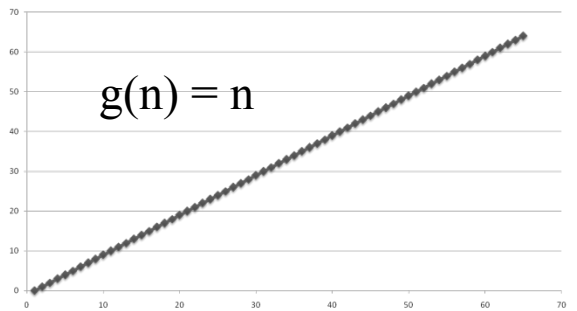
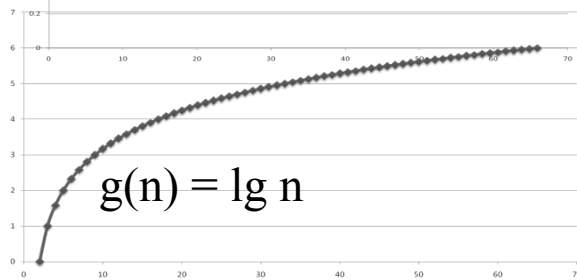
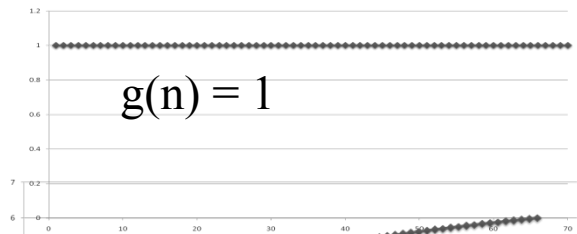


Seven Important Functions



- Seven functions that often appear in algorithm analysis:
 - Constant ≈ 1
 - Logarithmic $\approx \log n$
 - Linear $\approx n$
 - N-Log-N $\approx n \log n$
 - Quadratic $\approx n^2$
 - Cubic $\approx n^3$
 - Exponential $\approx 2^n$

Functions Graphed Using “Normal” Scale



Primitive Operations



- Basic computations performed by an algorithm
- Identifiable in pseudocode
- Largely independent from the programming language
- Exact definition not important (we will see why later)
- Assumed to take a constant amount of time in the RAM model
- Examples:
 - Evaluating an expression
 - Assigning a value to a variable
 - Indexing into an array
 - Calling a method
 - Returning from a method

Counting Primitive Operations

- By inspecting the pseudocode, we can determine the maximum number of primitive operations executed by an algorithm, as a function of the input size

Algorithm <i>arrayMax</i> (<i>A</i> , <i>n</i>)	# operations
<i>currentMax</i> ← <i>A</i> [0]	2
for <i>i</i> ← 1 to <i>n</i> - 1 do	2 <i>n</i>
if <i>A</i> [<i>i</i>] > <i>currentMax</i> then	2(<i>n</i> - 1)
<i>currentMax</i> ← <i>A</i> [<i>i</i>]	2(<i>n</i> - 1)
{ increment counter <i>i</i> }	2(<i>n</i> - 1)
return <i>currentMax</i>	1
	Total 8 <i>n</i> - 2

Estimating Running Time

- Algorithm *arrayMax* executes $8n - 2$ primitive operations in the worst case. Define:
 - a = Time taken by the fastest primitive operation
 - b = Time taken by the slowest primitive operation
- Let $T(n)$ be worst-case time of *arrayMax*. Then
$$a(8n - 2) \leq T(n) \leq b(8n - 2)$$
- Hence, the running time $T(n)$ is bounded by two linear functions

Growth Rate of Running Time



- Changing the hardware/ software environment
 - Affects $T(n)$ by a constant factor, but
 - Does not alter the growth rate of $T(n)$
- The linear growth rate of the running time $T(n)$ is an intrinsic property of algorithm *arrayMax*

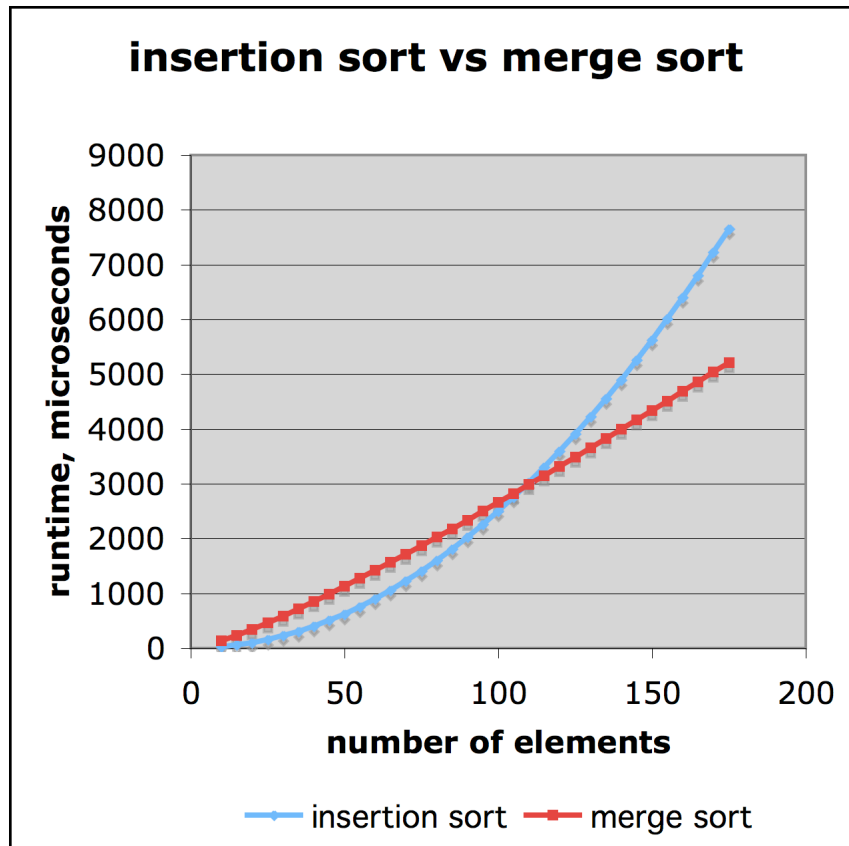
Why Growth Rate Matters



if runtime is...	time for $n + 1$	time for $2n$	time for $4n$
$c \lg n$	$c \lg (n + 1)$	$c (\lg n + 1)$	$c(\lg n + 2)$
cn	$c(n + 1)$	$2cn$	$4cn$
$cn \lg n$	$\sim cn \lg n + cn$	$2cn \lg n + 2cn$	$4cn \lg n + 4cn$
cn^2	$\sim cn^2 + 2cn$	$4cn^2$	$16cn^2$
cn^3	$\sim cn^3 + 3cn^2$	$8cn^3$	$64cn^3$
$c2^n$	$c2^{n+1}$	$c2^{2n}$	$c2^{4n}$

runtime quadruples when problem size doubles

Comparison of Two Algorithms



insertion sort is

$$n^2 / 4$$

merge sort is

$$2 n \lg n$$

sort a million items?

insertion sort takes
roughly **70 hours**

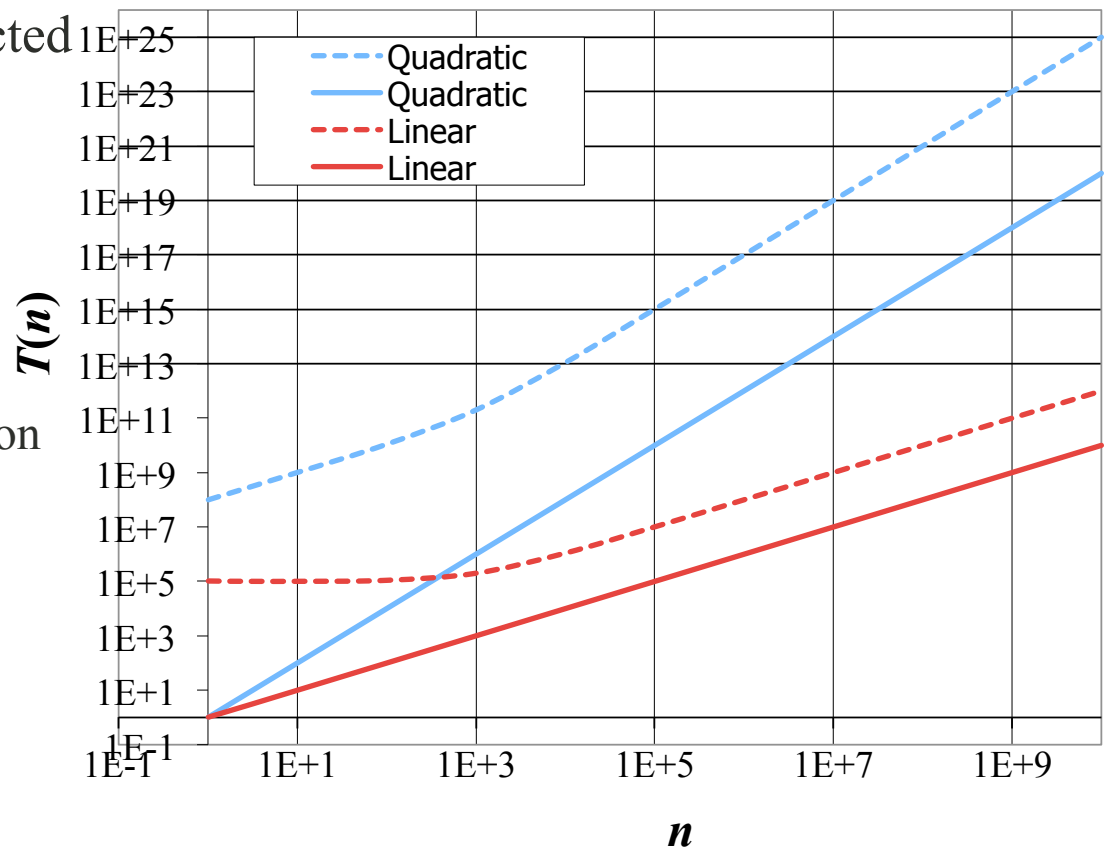
while

merge sort takes
roughly **40 seconds**

This is a slow machine, but if
100 x as fast then it's **40 minutes**
versus less than **0.5 seconds**

Constant Factors

- The growth rate is not affected by
 - constant factors or
 - lower-order terms
- Examples
 - $10^2n + 10^5$ is a linear function
 - $10^5n^2 + 10^8n$ is a quadratic function

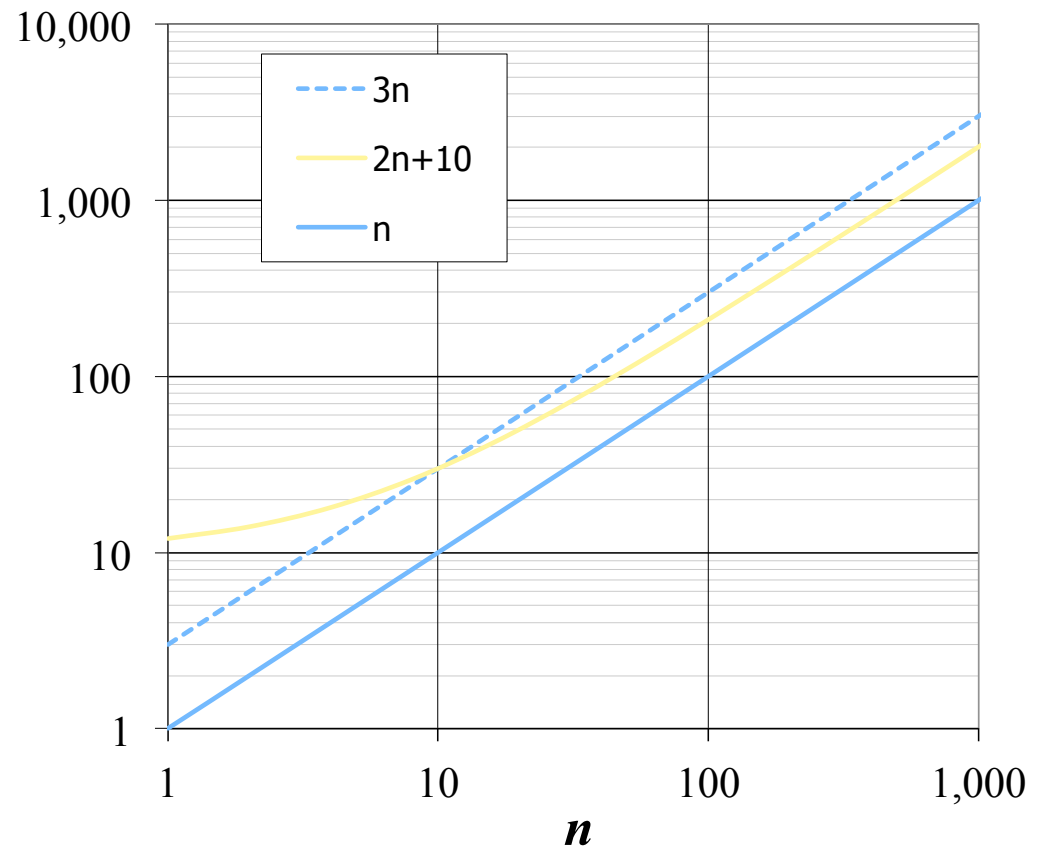


Big-Oh Notation

- Given functions $f(n)$ and $g(n)$, we say that $f(n)$ is $O(g(n))$ if there are positive constants c and n_0 such that

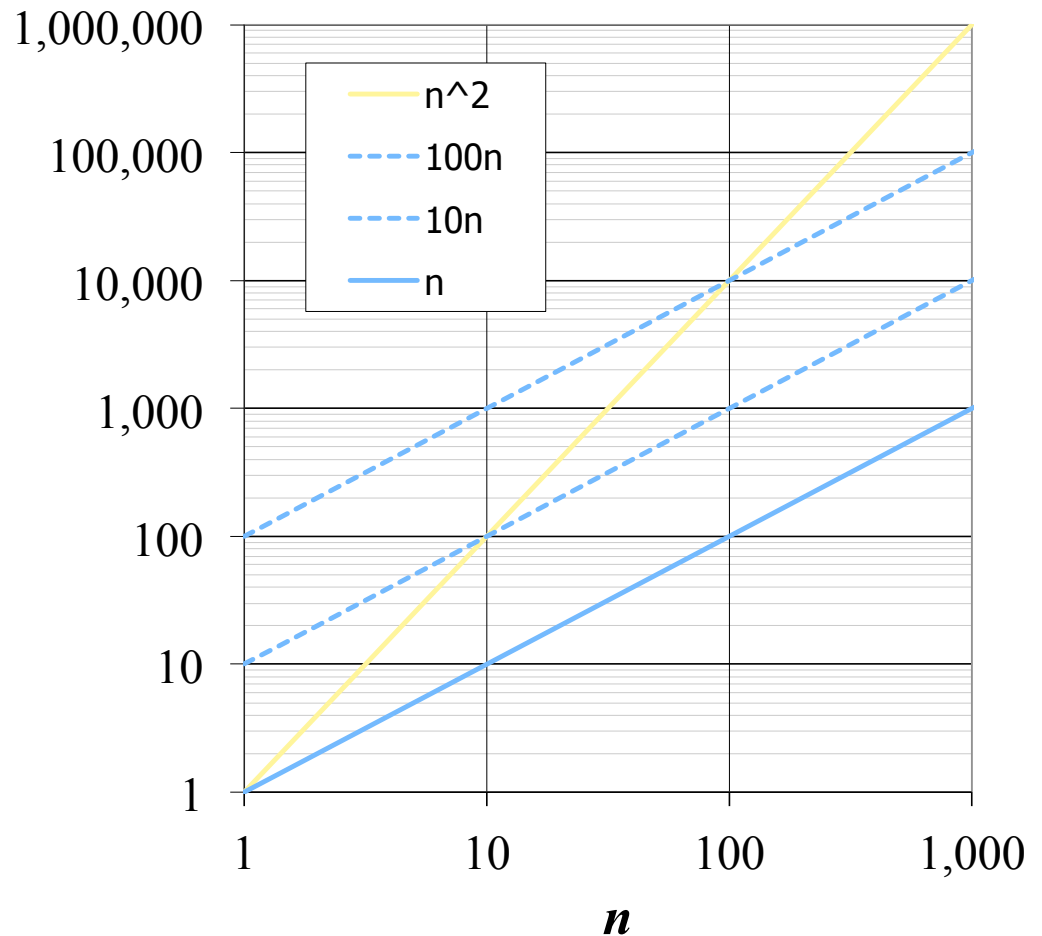
$$f(n) \leq cg(n) \text{ for } n \geq n_0$$

- Example: $2n + 10$ is $O(n)$
 - $2n + 10 \leq cn$
 - $(c - 2)n \geq 10$
 - $n \geq 10/(c - 2)$
 - Pick $c = 3$ and $n_0 = 10$



Big-Oh Example

- Example: the function n^2 is not $O(n)$
 - $n^2 \leq cn$
 - $n \leq c$
 - The above inequality cannot be satisfied since c must be a constant



More Big-Oh Example



- $7n-2$ is $O(n)$
 - need $c > 0$ and $n_0 \geq 1$ such that $7n-2 \leq c \cdot n$ for $n \geq n_0$
 - this is true for $c = 7$ and $n_0 = 1$
- $3n^3 + 20n^2 + 5$ is $O(n^3)$
 - need $c > 0$ and $n_0 \geq 1$ such that $3n^3 + 20n^2 + 5 \leq c \cdot n^3$ for $n \geq n_0$
 - this is true for $c = 4$ and $n_0 = 21$
- $3 \log n + 5$ is $O(\log n)$
 - need $c > 0$ and $n_0 \geq 1$ such that $3 \log n + 5 \leq c \cdot \log n$ for $n \geq n_0$
 - this is true for $c = 8$ and $n_0 = 2$

Big-Oh and Growth Rate

- The big-Oh notation gives an upper bound on the growth rate of a function
- The statement “ $f(n)$ is $O(g(n))$ ” means that the growth rate of $f(n)$ is no more than the growth rate of $g(n)$
- We can use the big-Oh notation to rank functions according to their growth rate

	$f(n)$ is $O(g(n))$	$g(n)$ is $O(f(n))$
$g(n)$ grows more	Yes	No
$f(n)$ grows more	No	Yes
Same growth	Yes	Yes

Big-Oh Rules

- If $f(n)$ is a polynomial of degree d , then $f(n)$ is $O(n^d)$, i.e.,
 1. Drop lower-order terms
 2. Drop constant factors
- Use the smallest possible class of functions
 - Say “ $2n$ is $O(n)$ ” instead of “ $2n$ is $O(n^2)$ ”
- Use the simplest expression of the class
 - Say “ $3n + 5$ is $O(n)$ ” instead of “ $3n + 5$ is $O(3n)$ ”

Asymptotic Algorithm Analysis

- The asymptotic analysis of an algorithm determines the running time in big-Oh notation
- To perform the asymptotic analysis
 - We find the worst-case number of primitive operations executed as a function of the input size
 - We express this function with big-Oh notation



Asymptotic Algorithm Analysis



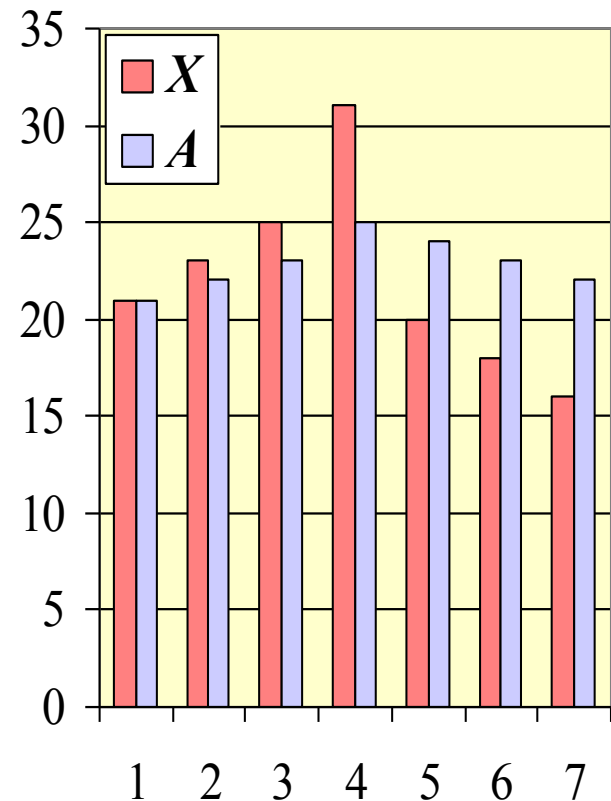
- Example:
 - We determine that algorithm *arrayMax* executes at most $8n - 2$ primitive operations
 - We say that algorithm *arrayMax* “runs in $O(n)$ time”
- Since constant factors and lower-order terms are eventually dropped anyhow, we can disregard them when counting primitive operations

Computing Prefix Averages

- We further illustrate asymptotic analysis with two algorithms for prefix averages
- The i -th prefix average of an array X is average of the first $(i + 1)$ elements of X :

$$A[i] = (X[0] + X[1] + \dots + X[i]) / (i+1)$$

- Computing the array A of prefix averages of another array X has applications to financial analysis



Exercise

- Implement prefixAverage
- **Input:**
 - Get n integers from a txt file
 - The first integer indicates the number of integers (the size of X)
- **Output:**
 - Print out a sequence of integers
 - The ith integer indicates the average of the first i+1 input numbers (starting from the second input)

Input: 4 1 2 3 5

Output: 1 2 2



Prefix Average (Quadratic)

- The following algorithm computes prefix averages in quadratic time by applying the definition

Algorithm *prefixAverages1*(X, n)

Input array X of n integers

Output array A of prefix averages of X

$A \leftarrow$ new array of n integers

for $i \leftarrow 0$ to $n - 1$ **do**

$s \leftarrow X[0]$

for $j \leftarrow 1$ to i **do**

$s \leftarrow s + X[j]$

$A[i] \leftarrow s / (i + 1)$

return A

#operations

n

n

n

$1 + 2 + \dots + (n - 1)$

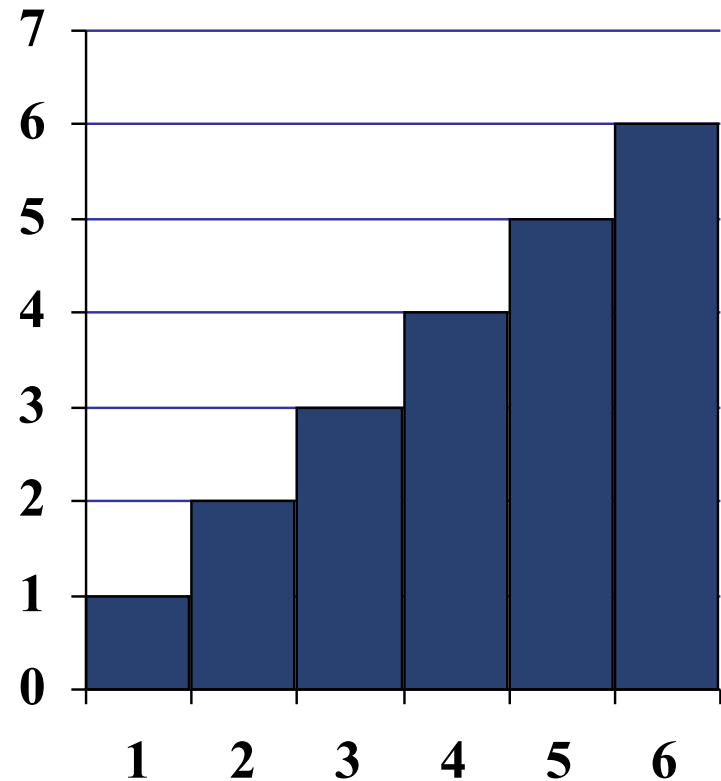
$1 + 2 + \dots + (n - 1)$

n

1

Arithmetic Progression

- The running time of *prefixAverages1* is $O(1 + 2 + \dots + n)$
- The sum of the first n integers is $n(n + 1) / 2$
 - There is a simple visual proof of this fact
- Thus, algorithm *prefixAverages1* runs in $O(n^2)$ time



Prefix Average (Linear)

- The following algorithm computes prefix averages in linear time by keeping a running sum
- Algorithm *prefixAverages2* runs in $O(n)$ time

Algorithm *prefixAverages2*(X, n)

Input array X of n integers

Output array A of prefix averages of X #operations

$A \leftarrow$ new array of n integers n

$s \leftarrow 0$ 1

for $i \leftarrow 0$ **to** $n - 1$ **do** n

$s \leftarrow s + X[i]$ n

$A[i] \leftarrow s / (i + 1)$ n

return A 1

Relatives of Big-Oh



- big-Omega

- $f(n)$ is $\Omega(g(n))$ if there is a constant $c > 0$
- and an integer constant $n_0 \geq 1$ such that
- $f(n) \geq c \cdot g(n)$ for $n \geq n_0$

- big-Theta

- $f(n)$ is $\Theta(g(n))$ if there are constants $c' > 0$ and $c'' > 0$ and an integer constant $n_0 \geq 1$ such that $c' \cdot g(n) \leq f(n) \leq c'' \cdot g(n)$ for $n \geq n_0$

Intuition for Asymptotic Notation



- Big-Oh
 - $f(n)$ is $O(g(n))$ if $f(n)$ is asymptotically **less than or equal** to $g(n)$
- big-Omega
 - $f(n)$ is $\Omega(g(n))$ if $f(n)$ is asymptotically **greater than or equal** to $g(n)$
- big-Theta
 - $f(n)$ is $\Theta(g(n))$ if $f(n)$ is asymptotically **equal** to $g(n)$

Examples of Using Relatives of Big-Oh

- **$5n^2$ is $\Omega(n^2)$**
 - $f(n)$ is $\Omega(g(n))$ if there is a constant $c > 0$ and an integer constant $n_0 \geq 1$ such that $f(n) \geq c \cdot g(n)$ for $n \geq n_0$
 - let $c = 5$ and $n_0 = 1$
- **$5n^2$ is $\Omega(n)$**
 - $f(n)$ is $\Omega(g(n))$ if there is a constant $c > 0$ and an integer constant $n_0 \geq 1$ such that $f(n) \geq c \cdot g(n)$ for $n \geq n_0$
 - let $c = 1$ and $n_0 = 1$
- **$5n^2$ is $\Theta(n^2)$**
 - $f(n)$ is $\Theta(g(n))$ if it is $\Omega(n^2)$ and $O(n^2)$. We have already seen the former, for the latter recall that $f(n)$ is $O(g(n))$ if there is a constant $c > 0$ and an integer constant $n_0 \geq 1$ such that $f(n) \leq c \cdot g(n)$ for $n \geq n_0$
 - Let $c = 5$ and $n_0 = 1$

Coming Up...

- Big O: Read Text Book 4
- Divide and Conquer/Sorting: Read Text Book 11

