

# Analysis Tools

Jyh-Shing Roger Jang (張智星)

CSIE Dept, National Taiwan University

# Several Basic Functions

---

- Polynomial
  - Constant function
  - Linear function
  - Quadratic function
  - Cubic function
- Logarithm function
- N-Log-N function
- Exponential function

# Family of Polynomials

- Constant function

- $f(n)=1$

n: problem size

- Linear function

- $f(n)=n$

- Quadratic function

- $f(n)=n^2$

- Cubic function

- $f(n)=n^3$

- A general polynomials

- $f(n)=a_0+a_1n+a_2n^2+a_3n^3+...+a_dn^d$

# The Logarithm Function

- $f(n) = \log_2(n) = \log(n)$

The default base is 2.

- Definition of logarithm

$$x = \log_b n \text{ if and only if } b^x = n.$$

- Some identities

1.  $\log_b ac = \log_b a + \log_b c$
2.  $\log_b a/c = \log_b a - \log_b c$
3.  $\log_b a^c = c \log_b a$
4.  $\log_b a = (\log_d a) / \log_d b$
5.  $b^{\log_d a} = a^{\log_d b}$

- More...

$$y = \ln x \Rightarrow y' = \frac{1}{x} \Rightarrow \int \frac{1}{x} dx = \ln x$$

The base is  
 $e = 2.718281828\dots$

# The N-Log-N Function

---

- $f(n) = n \cdot \log(n)$

# The Exponential Function

- $f(n)=a^n$
- Some identities (for positive a, b, and c)

$$a^{(b+c)} = a^b a^c$$

$$a^{bc} = (a^b)^c$$

$$a^b / a^c = a^{(b-c)}$$

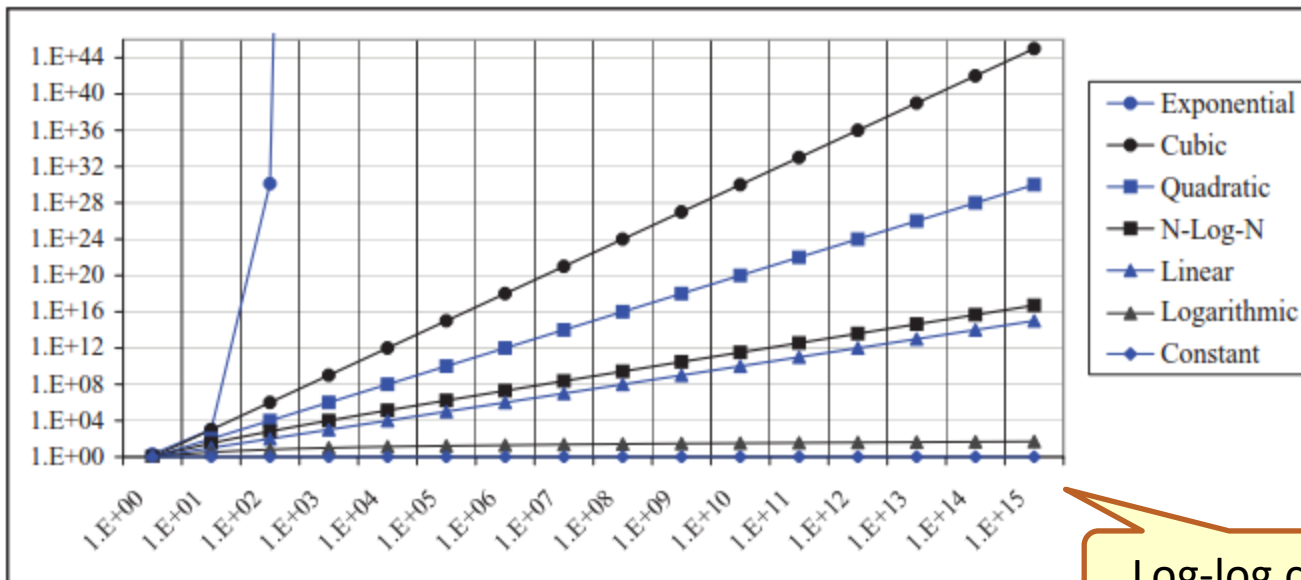
$$b = a^{\log_a b}$$

$$b^c = a^{c \cdot \log_a b}$$

# Growth Rate Comparisons

<i>constant</i>	<i>logarithm</i>	<i>linear</i>	<i>n-log-n</i>	<i>quadratic</i>	<i>cubic</i>	<i>exponential</i>
1	$\log n$	$n$	$n \log n$	$n^2$	$n^3$	$a^n$

**Table 4.1:** Classes of functions. Here we assume that  $a > 1$  is a constant.

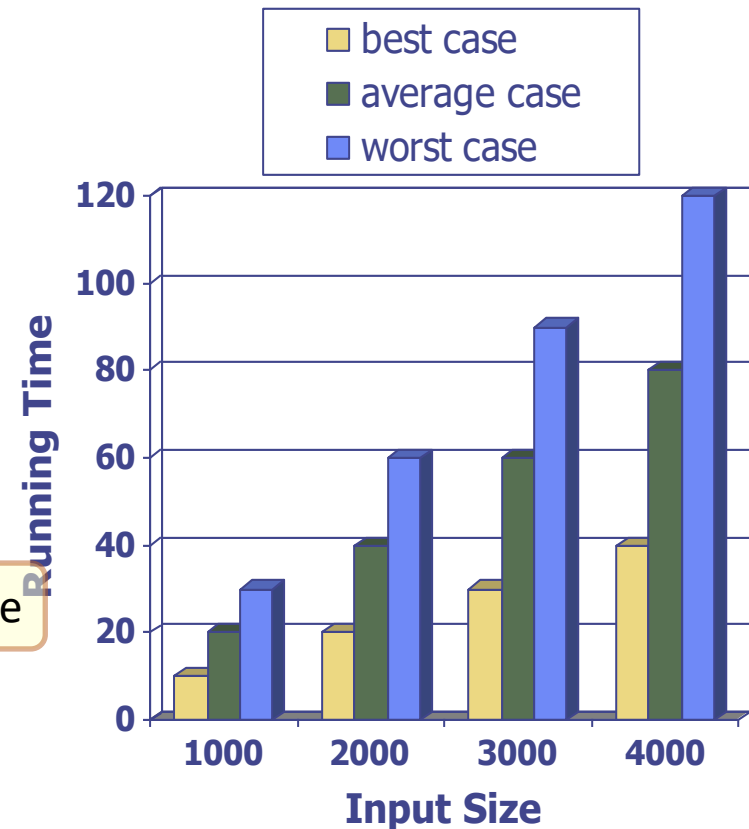


$$y = n^k \Rightarrow \log y = k \log n \Rightarrow Y = kX$$

# 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

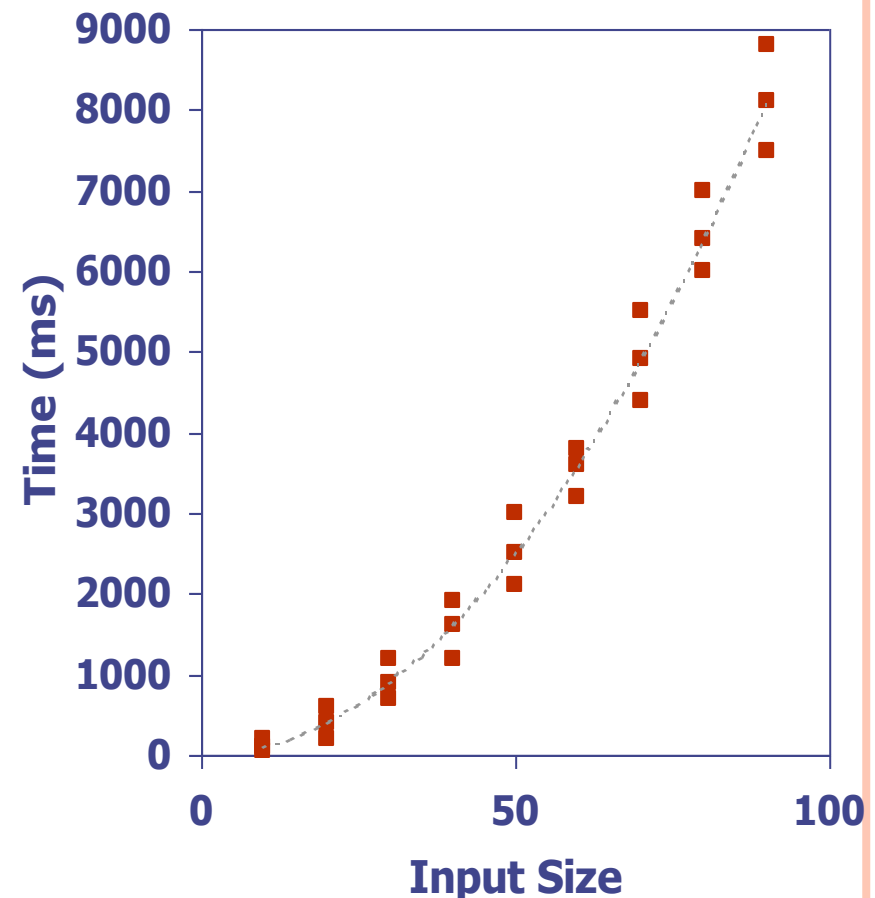
Sorting, for example





# Performance Measurement via Experiments

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



# Limitations of Experimental Studies

---

- 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 Performance 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

# ABOUT PRIMITIVE OPERATIONS

---

## ○ 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

## ○ 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> ( $A, n$ )	# operations
<i>currentMax</i> $\leftarrow A[0]$	2
for $i \leftarrow 1$ to $n - 1$ do	$2n$
if $A[i] > \textit{currentMax}$ then	$2(n - 1)$
<i>currentMax</i> $\leftarrow A[i]$	$2(n - 1)$
{ increment counter $i$ }	$2(n - 1)$
return <i>currentMax</i>	1
Total	$8n - 2$

# Estimating Running Time

- Algorithm *arrayMax* executes  $8n - 2$  primitive operations in the worst case. Define:

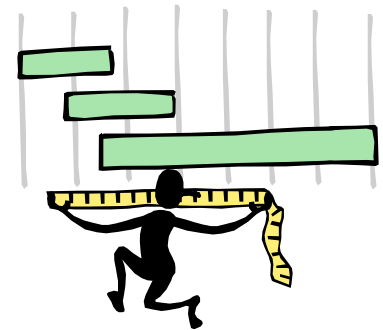
$a$  = Time taken by the fastest primitive operation

$b$  = 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
- Change the hardware environment only affects  $a$  and  $b$ .
- The linear growth rate of the running time  $T(n)$  is an intrinsic property of algorithm *arrayMax*



# 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

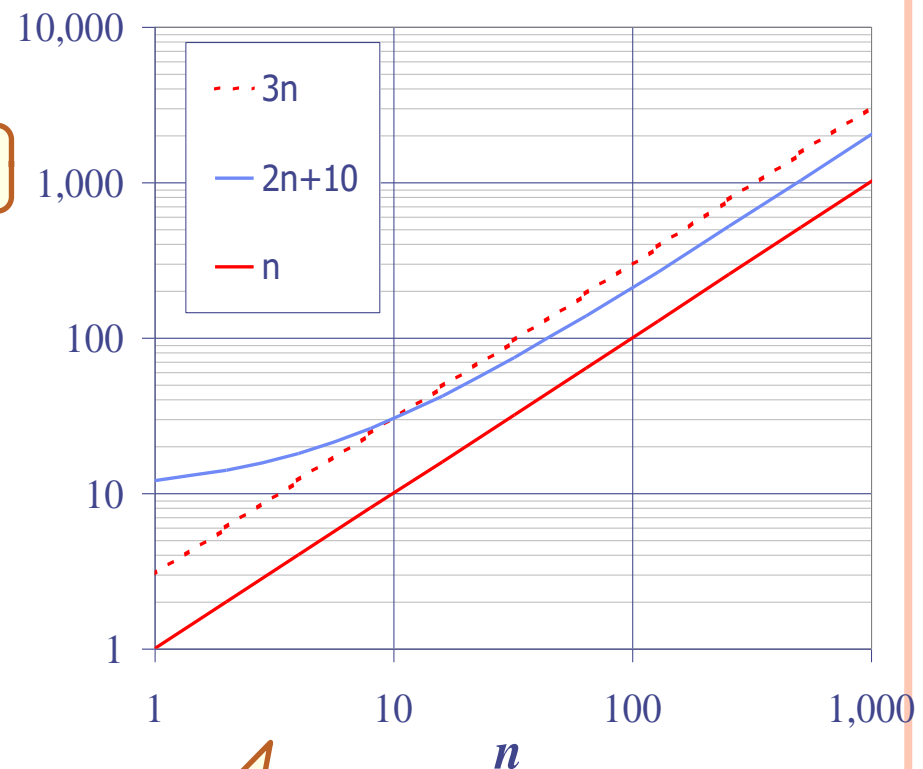
$g(n)$ : basic functions

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

Quiz!

- Example:  $2n + 10$  is  $O(n)$

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

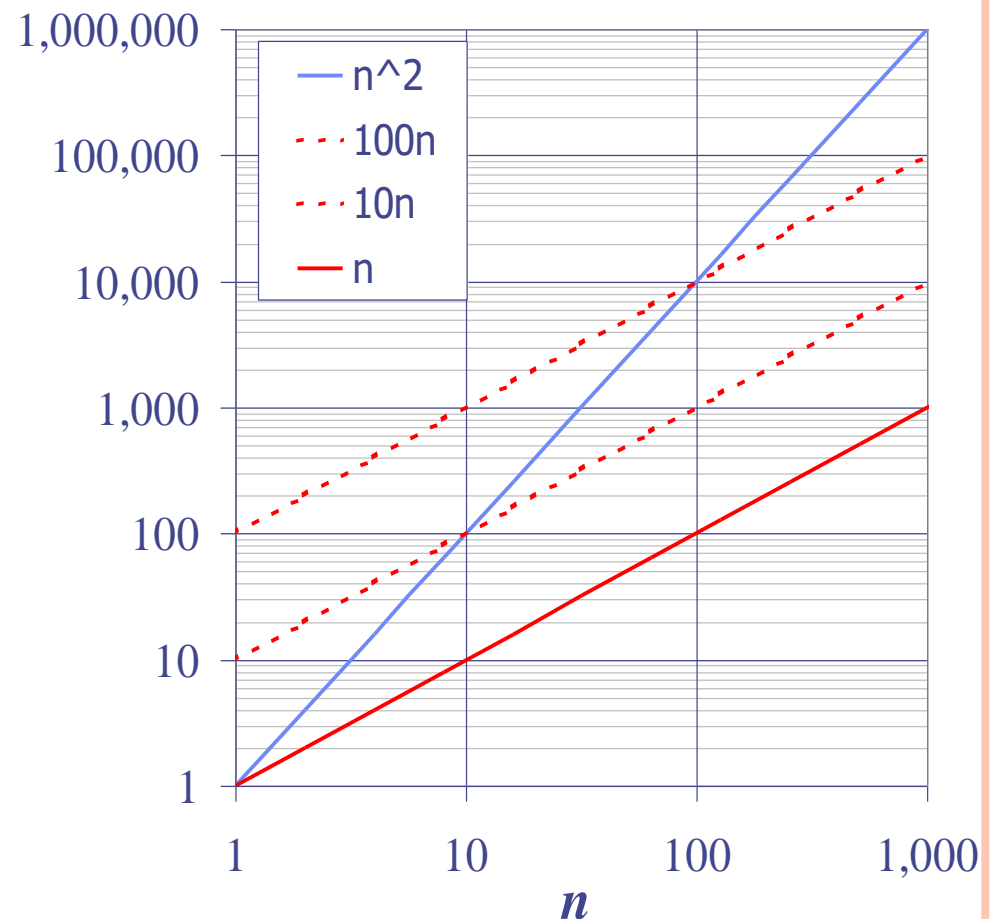


Log scale

# 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 Examples

- $7n-2$  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  $c = 28$  and  $n_0 = 1$
- $3 \log n + 5$  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  $n_0 = 2$

Quiz!

# 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**

That is, when  $n$  is big!

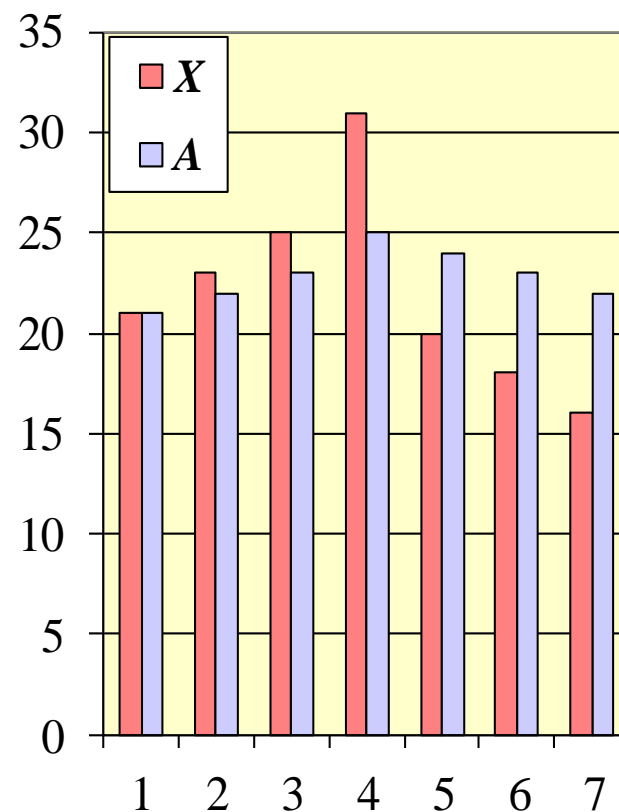
  - We find the worst-case number of primitive operations executed as a function of the input size
  - We express this function with big-Oh notation
- Example:
  - We determine that algorithm *arrayMax* executes at most  $8n - 2$  primitive operations
  - We say that algorithm *arrayMax* “runs in  $O(n)$  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



## Prefix Averages (Quadratic)

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

**Algorithm** *prefixAverages1*( $X, n$ )

**Input** array  $X$  of  $n$  integers

**Output** array  $A$  of prefix averages of  $X$     #operations

$A \leftarrow$  new array of  $n$  integers     $n$

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

$s \leftarrow X[0]$      $n$

**for**  $j \leftarrow 1$  **to**  $i$  **do**     $1 + 2 + \dots + (n - 1)$

$s \leftarrow s + X[j]$      $1 + 2 + \dots + (n - 1)$

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

**return**  $A$     1

## Prefix Averages (Linear)

- The following algorithm computes prefix averages in linear time by keeping a running sum

**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**

$n$

$s \leftarrow s + X[i]$

$n$

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

$n$

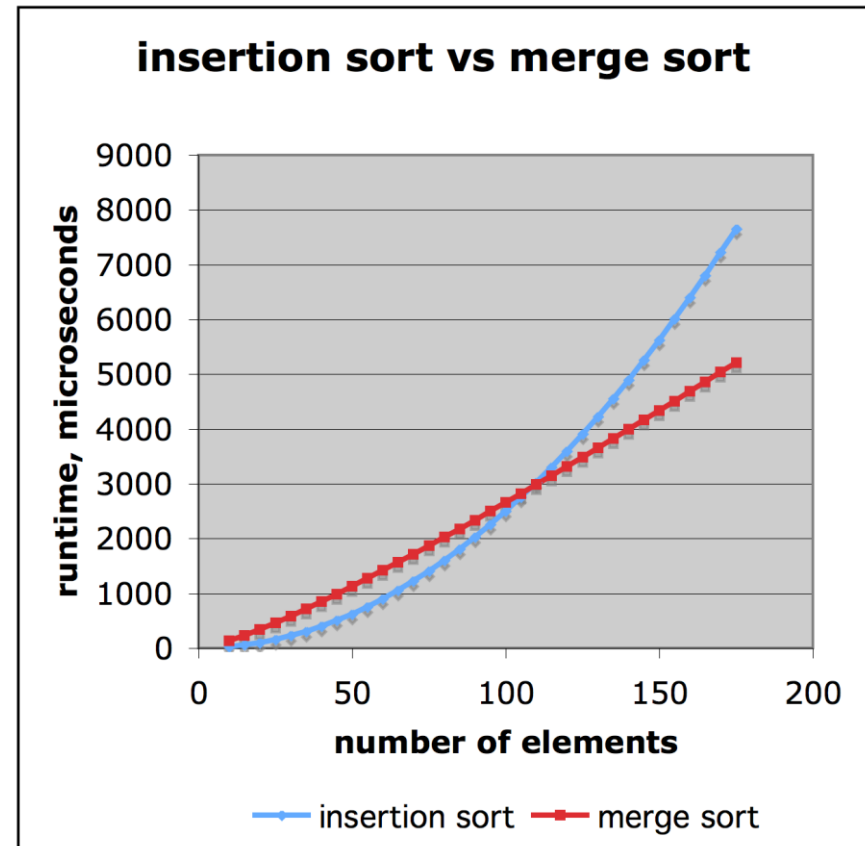
**return**  $A$

1

# Comparison of Two Algorithms

- Two sorting algorithms
  - Merge sort is  $O(n \log n)$
  - Insertion sort is  $O(n^2)$
- To sort 1M items
  - Insertion sort → 70 hours
  - Merge sort → 40 seconds
- For a faster machine
  - Insertion sort → 40 minutes
  - Merge sort → 0.5 seconds

Slide by Matt Stallmann



# 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: upper bound
  - $f(n)$  is  $O(g(n))$  if  $f(n)$  is asymptotically **less than or equal** to  $g(n)$
- Big-Omega: lower bound
  - $f(n)$  is  $\Omega(g(n))$  if  $f(n)$  is asymptotically **greater than or equal** to  $g(n)$
- Big-Theta: average bound
  - $f(n)$  is  $\Theta(g(n))$  if  $f(n)$  is asymptotically **equal** to  $g(n)$

$$1 < \log(n) < n^{1/2} < n < n \log(n) < n^2 < n^3 < n^4 < \dots < 2^n < 3^n < 4^n < \dots < n^n$$

## Example Uses of the 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  $n \geq n_0$   $\rightarrow$  let  $c = 5$  and  $n_0 = 1$

### ○ $5n^2$ is $O(n^2)$

- $f(n)$  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  $n \geq n_0$   $\rightarrow$  let  $c = 5$  and  $n_0 = 1$

### ○ $5n^2$ is $\Theta(n^2)$

- $f(n)$  is  $\Theta(g(n))$  if it is  $\Omega(n^2)$  and  $O(n^2)$ .

# Computing Powers

- To compute the power function  $p(x, n) = x^n$

- Method 1

- $n, n-1, n-2, \dots, 2, 1 \rightarrow O(n)$

$$p(x, n) = \begin{cases} 1 & \text{if } n = 0 \\ x \cdot p(x, n-1) & \text{otherwise} \end{cases}$$

- Method 2

- $n, n/2, n/4, \dots, 2, 1 \rightarrow O(\log n)$

$$p(x, n) = \begin{cases} 1 & \text{if } n = 0 \\ x \cdot p(x, (n-1)/2)^2 & \text{if } n > 0 \text{ is odd} \\ p(x, n/2)^2 & \text{if } n > 0 \text{ is even} \end{cases}$$

# Element Uniqueness Problem (1/2)

- To determine if the elements in a vector are all unique
  - $O(2^n)$  implementation by recursion

```
bool isUnique(const vector<int>& arr, int start, int end) {
    if (start >= end) return true;
    if (!isUnique(arr, start, end-1))
        return false;
    if (!isUnique(arr, start+1, end))
        return false;
    return (arr[start] != arr[end]);
}
```

- $O(n^2)$  implementation by looping

```
bool isUniqueLoop(const vector<int>& arr, int start, int end) {
    if (start >= end) return true;
    for (int i = start; i < end; i++)
        for (int j = i+1; j <= end; j++)
            if (arr[i] == arr[j]) return false;
    return true;
}
```

## Element Uniqueness Problem (2/2)

- To determine if the elements in a vector are all unique
  - $O(n \log n)$  implementation → Sort the vector first and check for neighboring duplicate elements

```
bool isUniqueSort(const vector<int>& arr, int start, int end) {
    if (start >= end) return true;
    vector<int> buf(arr);           // duplicate copy of arr
    sort(buf.begin()+start, buf.begin()+end); // sort the subarray
    for (int i = start; i < end; i++) // check for duplicates
        if (buf[i] == buf[i+1]) return false;
    return true;
}
```

- A faster average-case running time can be achieved by using the hash table data structure (Section 9.2).

# How to Prove Statements

- By **giving counter example**
  - Roger claims that every number of the form  $2^i - 1$  is a prime, where  $i$  is an integer greater than 1. Prove he is wrong.
- By **contrapositive**: Switching the hypothesis and conclusion of a conditional statement and negating both.
  - If  $p$  is true, then  $q$  is true  $\iff$  If  $q$  is not true, then  $p$  is not true
  - If  $a * b$  is even, then  $a$  is even or  $b$  is even.  $\iff$  If  $a$  is odd and  $b$  is odd, then  $a * b$  is odd.
  - Be aware of “DeMorgan’s Law”
- By **math induction**
  - Example: Prove that  $F(n) < 2^n$ , where  $F(n)$  is the Fibonacci function with  $F(n+2) = F(n+1) + F(n)$ , and  $F(1) = 1$ ,  $F(2) = 2$ .

## Exercise on Big Oh

---

- Given functions  $f(n)$  and  $g(n)$ , under what condition do we say that  $f(n)$  is  $O(g(n))$ ?
  - Note that  $g(n)$  is one of the basic functions, such as  $n^3$ .
  - What is the physical meaning of the definition?
- Use the definition of big oh to explain why  $3n^2 + n\log(n) + 2n + 5\log(n)$  is  $O(n^2)$ .

# Exercise on Proof by Induction

---

- Proof by induction
  - Prove that  $F(n) < 2^n$ , where  $F(n)$  is the Fibonacci function satisfying  $F(n+2) = F(n+1) + F(n)$ , with  $F(1) = 1$ ,  $F(2) = 2$ .
- 3 steps in proof by induction
  - Base case
  - Inductive Hypothesis
  - Inductive Step