Data Structures (in C++)

- Introduction -

Jinsun Park

Visual Intelligence and Perception Lab., CSE, PNU

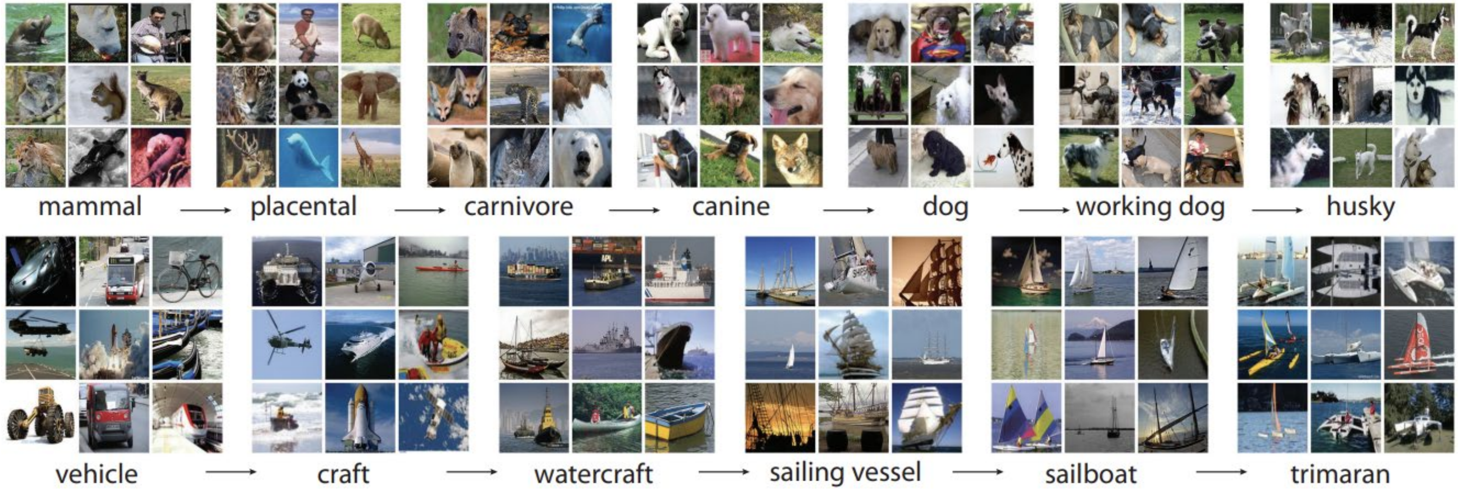
Data & Data Structure

- **Data**
  - Collection of (raw) information
- **Data Structure**
  - A systematic way of organizing and accessing data
  - A set of data arrangement rules
  - The most important goal is the efficient data processing
    (*i.e.*, fast data access, searching, sorting, etc.)

https://cs.stanford.edu/people/karpathy/cnnembed/

Random images

Deng et al., "Imagenet: A large-scale hierarchical image database.", CVPR 2009



Hierarchically organized images

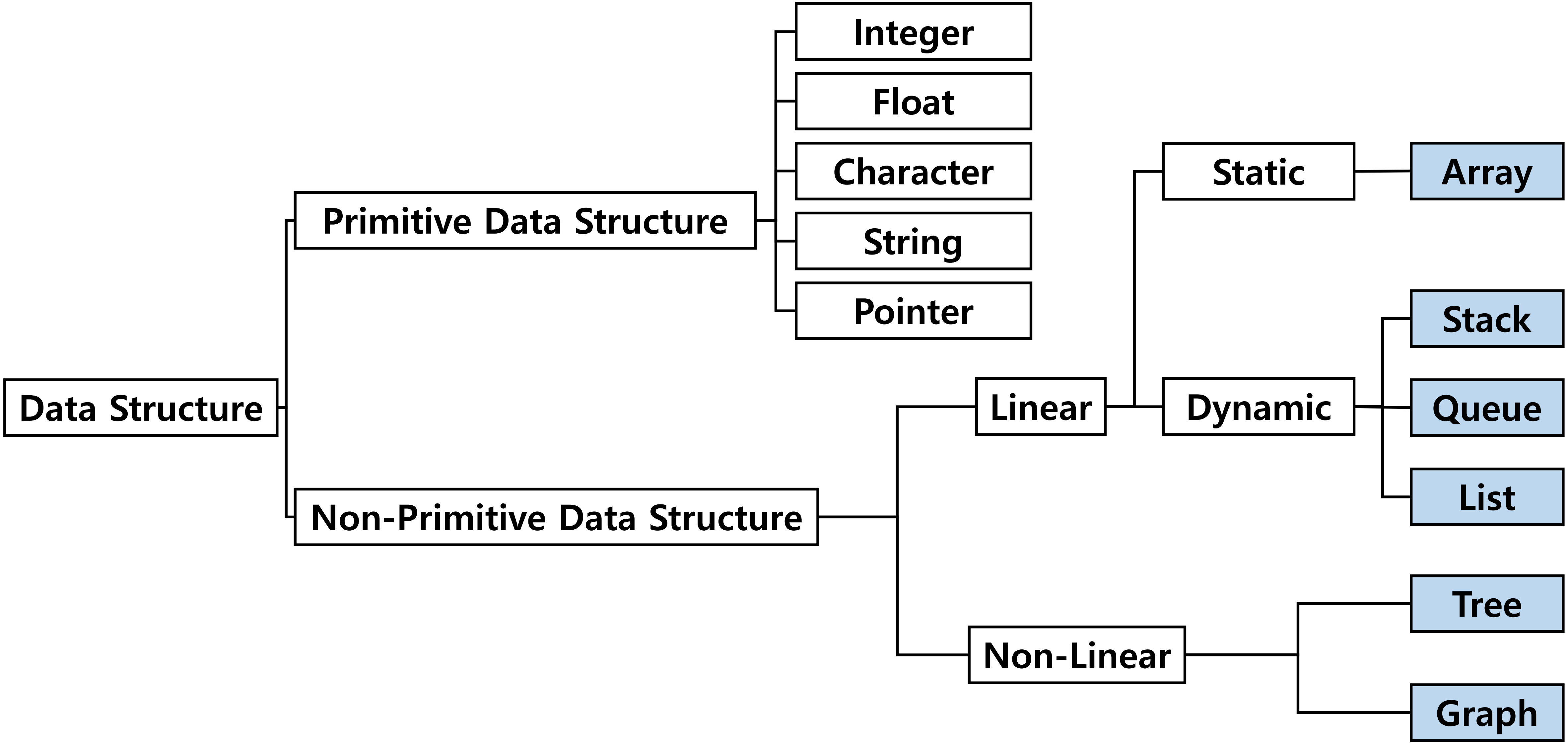
What is a "Good" Data Structure?

- Assume that we have three data structures:

	Data Structure A	Data Structure B	Data Structure C
Insertion	Very Fast	Slow	Fast
Searching	Slow	Very Fast	Fast
Memory Consumption	Low	Low	High

- What is a good (or bad) data structure for:
  - Search engines (e.g., Google)
  - Industrial machines with a limited memory size
  - Web browsing history

Classification of Data Structure



Abstraction & Abstract Data Type (ADT)

- **Abstraction**
  - Simple descriptions of fundamental parts of a (complicated) system
  - Naming and explanation of a functionality
    - No internal details are given

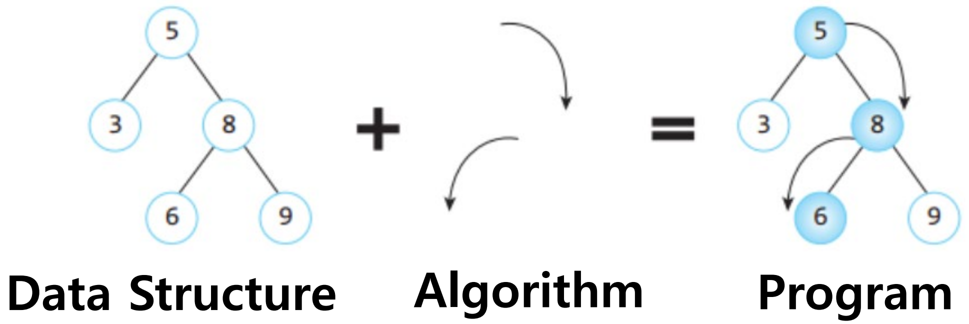
- **Abstract Data Type (ADT)**
  - A mathematical model of a data structure
  - Specifies what each operation does
  - Does not specify how it does it
  - ADT = Data + Operations

```
Data: Integers ∈ [1, INT_MAX]
Operations:
▪ add(x,y): return x+y if (x+y) <= INT_MAX,
  return INT_MAX otherwise
▪ distance(x,y): return |x-y|
▪ equal(x,y): return TRUE if x == y, return
  FALSE otherwise
▪ successor(x): return x+1 if (x+1) <=
  INT_MAX, return INT_MAX otherwise
```

Natural Number ADT

Algorithm & Program

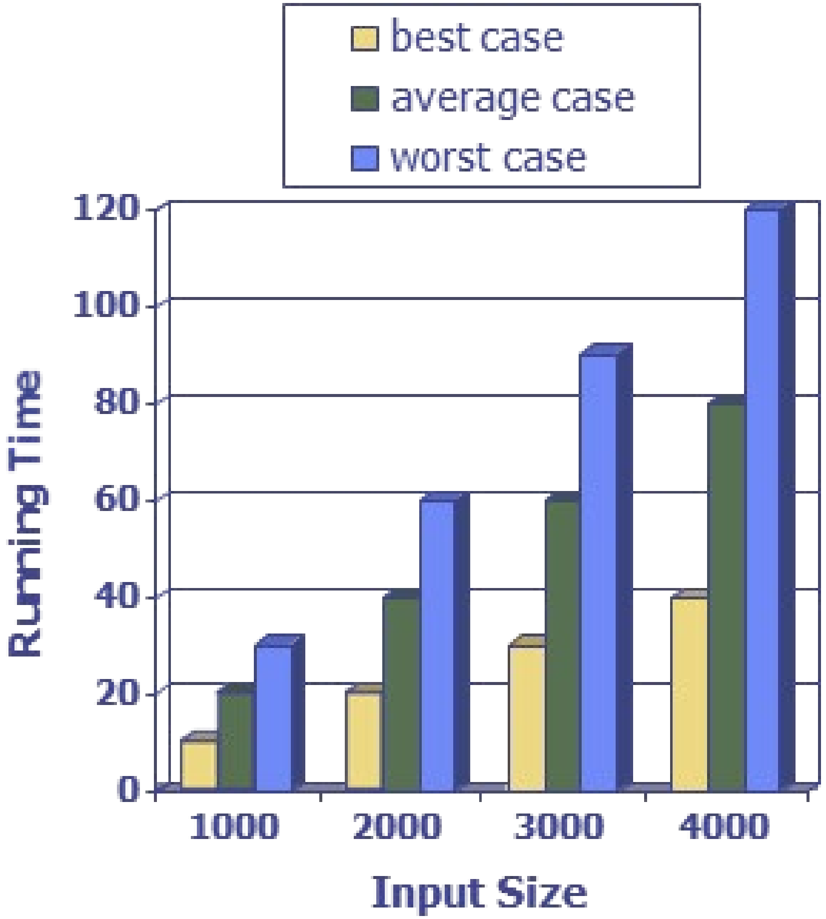
- **Algorithm**
  - A step-by-step procedure for performing some task in a finite amount of time
  - (Typical) Program = Data Structure + Algorithm



- **All algorithms must satisfy the following criteria:**
  - Zero or more input values
  - One or more output values
  - Clear and unambiguous instructions
  - Atomic steps that take constant time
  - No infinite sequence of steps
  - Feasible with specified computational device

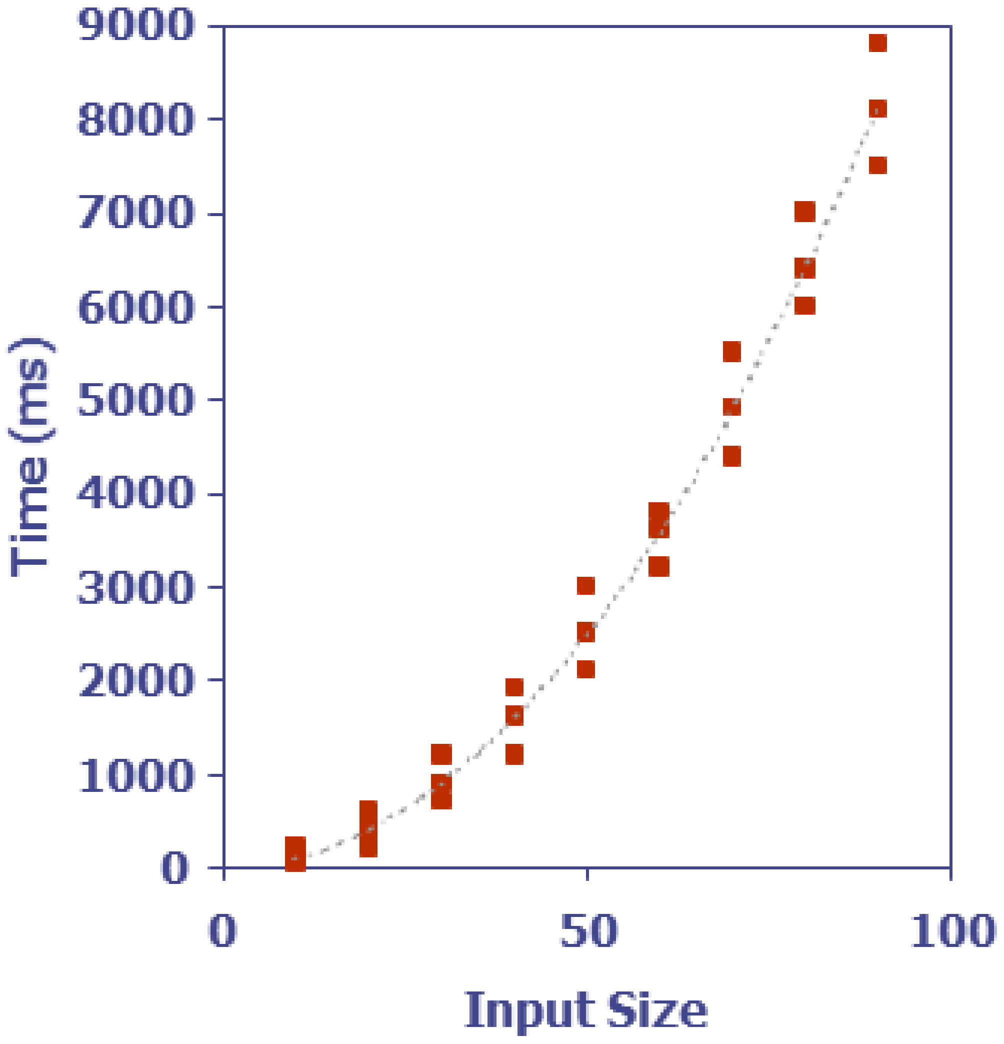
Algorithm Analysis

- **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

- **Experimental Studies**
  - Write a program implementing the algorithm
  - Run the program with inputs of varying size and composition
  - Use a method like *clock()* to get an accurate measure of the actual running time



- **Limitations**
  - 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

- **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

What we are typically interested in

- **Time Complexity**
  - The amount of time taken by an algorithm

- **Space Complexity**
  - The amount of space or memory taken by an algorithm

Pseudocode

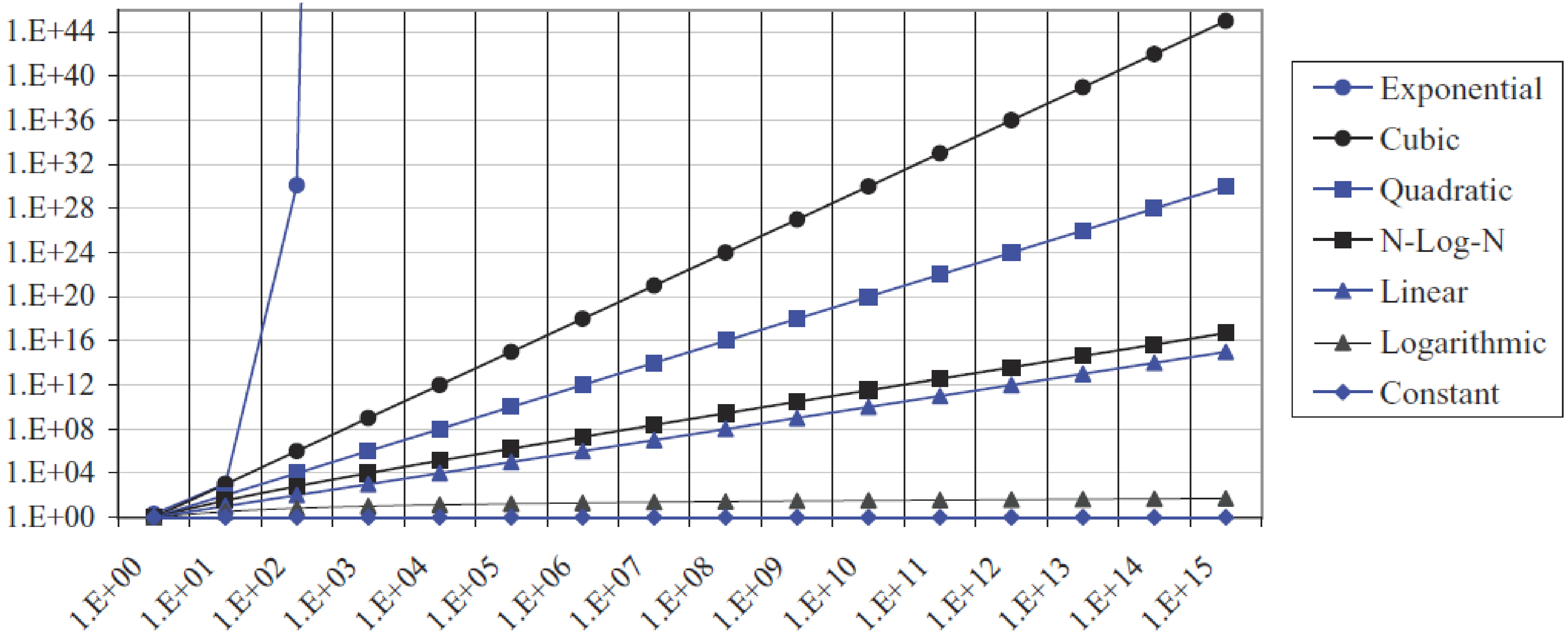
- **Pseudocode**
  - High-level description of an algorithm
  - More structured than English prose
  - Less detailed than a program
  - Preferred notation for describing algorithms
  - Hides program design issues

Algorithm $arrayMax(A, n)$
  Input array A of n integers
  Output maximum element of A

  $currentMax \leftarrow A[0]$
  for $i \leftarrow 1$ **to** $n - 1$ **do**
    if $A[i] > currentMax$ **then**
      $currentMax \leftarrow A[i]$
  return $currentMax$

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 $\approx 2^n$



Growth rates (plotted in a log-log chart)

Primitive Operations

- **Primitive Operations**
  - Basic computations performed by an algorithm
  - Identifiable in pseudocode
  - Largely independent from the programming language
  - Exact definition is not important
  - Assumed to take a constant amount of time

- **Examples**
  - Assigning a value to a variable
  - Calling a function
  - Arithmetic operation
  - Comparison
  - Indexing into an array
  - Following an object reference
  - Returning from a function

Estimating Running Time

- **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

- Indexing
- Assignment
- Comparison
- Arithmetics
- Returns

Algorithm *arrayMax*(A, n)	# operations
$currentMax \leftarrow A[0]$	2
for $i \leftarrow 1$ **to** $n - 1$ **do**	$2n$
if $A[i] > currentMax$ **then**	$2(n - 1)$
$currentMax \leftarrow A[i]$	$2(n - 1)$
{ increment counter i }	$2(n - 1)$
return $currentMax$	1
Total	$8n - 3$

Estimating Running Time

- Algorithm ***arrayMax*** executes $8n - 3$ 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,

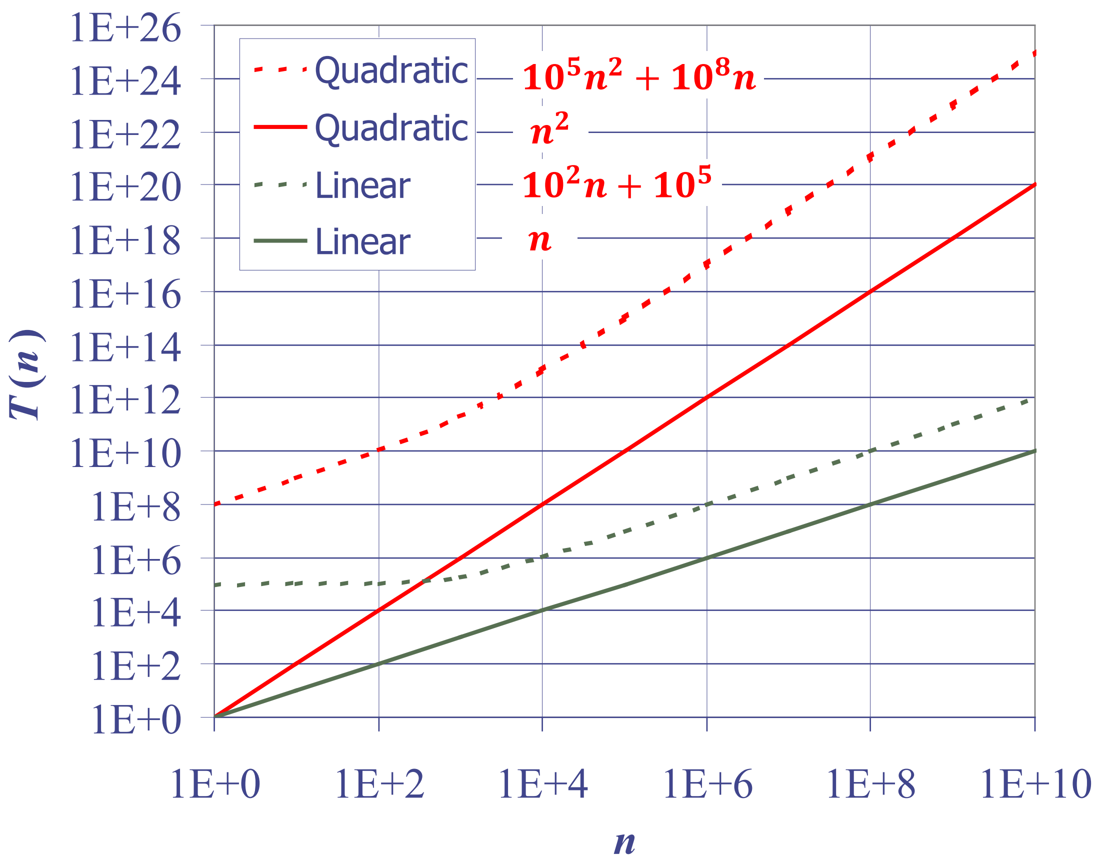
$$a\,(8n - 3) < T(n) < b(8n - 3)$$

Hence, the running time $T(n)$ is bounded by two linear functions

Growth Rate

- **The growth rate is not affected by:**
  - constant factors
  - lower-order terms

- **Examples**
  - $10^2 n + 10^5$ is a linear function
  - $10^5 n^2 + 10^8 n$ is a quadratic function



Growth rates (plotted in a log-log chart)

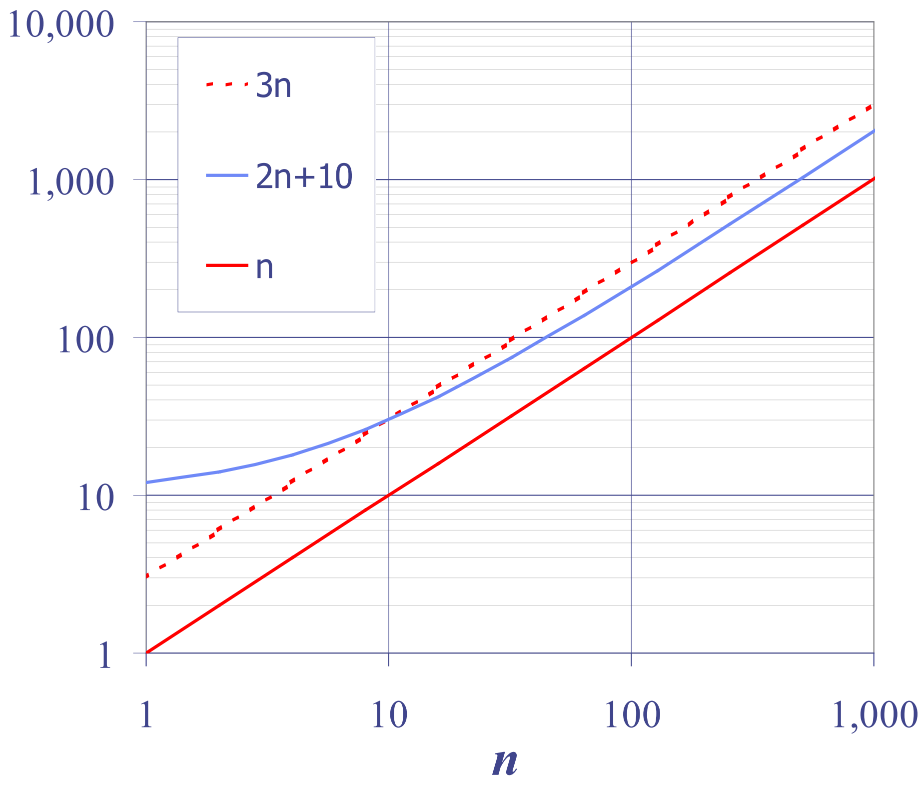
Big-Oh Notation

- 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

$f(n)$ is big-Oh of $g(n)$
or
$f(n)$ is order of $g(n)$

$$f(n) \leq cg(n) \quad for \quad n \geq n_0$$

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



Big-Oh Notation

- **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

- **Big-Oh Rules**
  - If $f(n)$ is a polynomial of degree d, then $f(n)$ is $O(n^d)$
    - Drop lower-order terms
    - 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

- Example:
  - We determine that algorithm ***arrayMax*** executes at most $8n - 3$ primitive operations
  - We say that algorithm ***arrayMax*** "runs in $O(n)$ time"

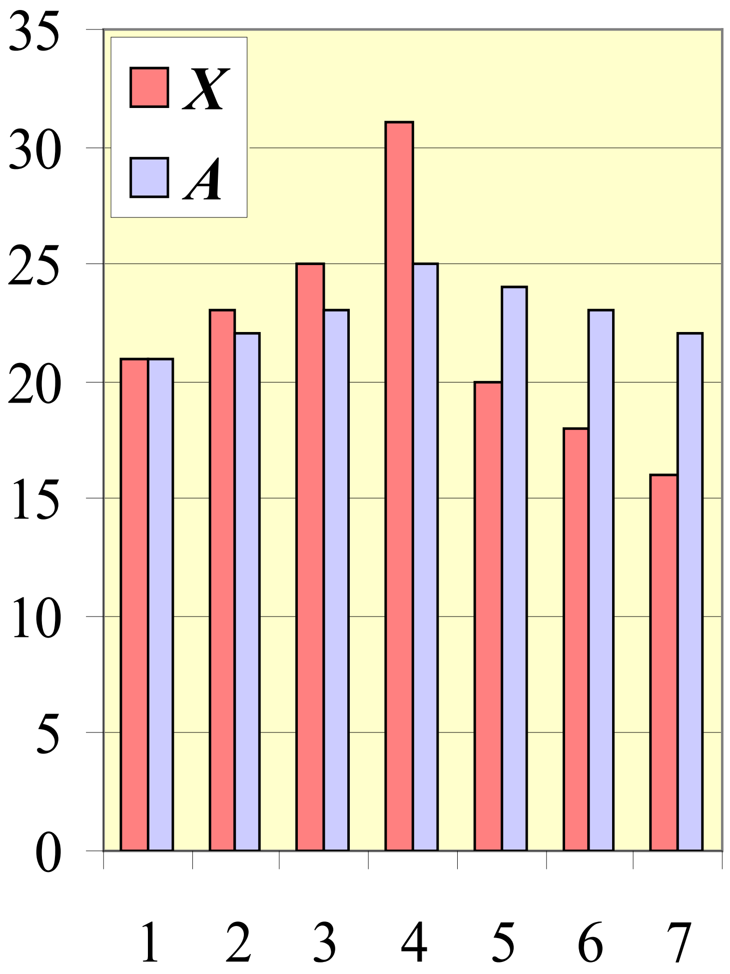
- Since constant factors and lower-order terms are eventually dropped anyhow, we can disregard them when counting primitive operations

Example: 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 :

$$A[i] = \frac{(X[0] + X[1] + \ldots + X[i])}{i+1}$$



Example: Computing Prefix Averages (Quadratic)

- The following algorithm computes prefix averages in quadratic time

- Indexing
- Assignment
- Comparison
- Arithmetics
- Returns

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**	$1 + 2 + \ldots + (n - 1)$
$s \leftarrow s + X[j]$	$1 + 2 + \ldots + (n - 1)$
$A[i] \leftarrow s / (i + 1)$	n
return A	1

$\propto n^2$

- Algorithm *prefixAverages1* runs in $O(n^2)$ time

Example: Computing Prefix Averages (Linear)

- The following algorithm computes prefix averages in linear time

- Indexing
- Assignment
- Comparison
- Arithmetics
- Returns

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

- Algorithm *prefixAverages2* runs in $O(n)$ time

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

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

- **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

$$c'g(n) \leq f(n) \leq c''g(n) \quad \text{for} \quad n \geq n_0$$

- **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 $g(n)$

Summary

- **Data & Data Structure**

- **Abstraction & Abstract Data Type (ADT)**

- **Asymptotic Algorithm Analysis**
  - Pseudocode
  - Primitive Operations
  - Big-Oh, Big-Omega, and Big-Theta