Introduction and Overview

Yo-Sub Han
CS, Yonsei University

Overview of Unit

① Starters

② Motivation for course

③ Overview of course

④ Big-O notation

Starters

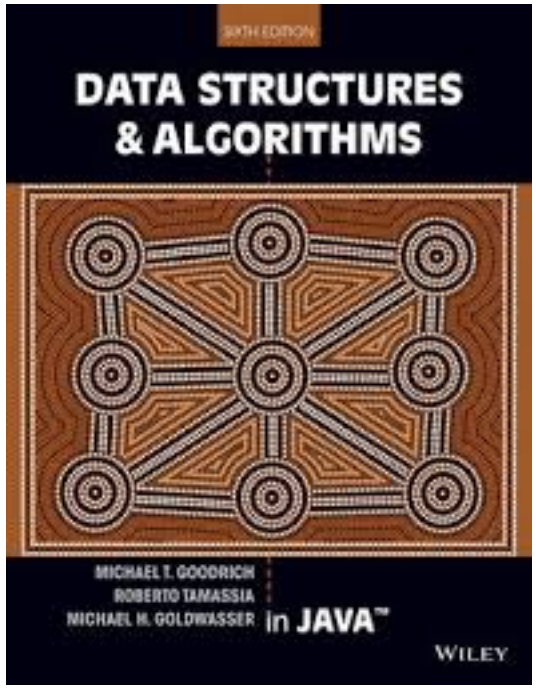
- Instructor: Dr. Yo-Sub Han, EngD 722
  - Email: emmous@yonsei.ac.kr
  - Course URL: `https://www.learnus.org`
  - Office hours: by email appointment
  - Leading the Theory of Computation lab (`http://toc.yonsei.ac.kr`)
  - Working on automata theory, algorithm design and information retrieval
- Teaching assistant:
  - Sicheol Sung (성시철)
  - Email: yonseitoc.ta@gmail.com
  - Office: EngD 709 (Theory of Computation Lab)
  - Office hours: by email appointment
    - Email subject should start with GEK6198:

Class Info.

- This course studies various data strucutres for efficient data storing, searching and sorting.
- The course topic includes lists, stacks, queues, tree/search trees, hashing and sorting
- Time: Mon. 1

Texts

- One required text: *Data structures and algorithms in Java* (6th edition) by Michael T. Goodrich, Roberto Tamassia, Michael H. Goldwasser
- I am using this text because it tries to give an overview of the material that we cover. I will also provide lecture notes (not 100%) based on the text



Philosophy

- Lectures provide a first pass at course material. They provide preliminary understanding and knowledge
- Readings extend and reinforce lecture material
- Homeworks provide training in course material. You learn by doing
- Programmings help you understand what you learned
- Examinations test your personal knowledge of the covered material

Evaluation

- There will be two exams:
  - Midterm: Apr. 24 (offline, FIRM)
  - Final: Jun. 12 (offline, FIRM)
- (Homework) Programming (4) + Written (2): 40% (out of 6 times)
- Midterm + Final exams: 50%
- Class participation including discussion and attendence: 10%
- If you miss more than $\frac{1}{3}$, then your grade is F
  - The number of missing classes directly affects your grade

Plagiarism

Yonsei University precautions against cheating and dishonesty:

> According to the University regulations, the student will get an F grade for the course that (s)he cheated on, and all the other courses for which exams have not yet been written at the time of the offence in the same semester will be granted a W (withdraw) grade.

Note: All written and programming assignments should be done individually. You are allowed to discuss your solutions with colleagues, but you are not allowed to share code or solutions. Sharing code (including "smarter" sharing by changing variable names also) counts as plagiarism and will be treated as outlined above

Acceptable:

- Discussing general solution strategies with a classmate/TA/professor.
- Clarifying what something in the homework description means.
- Discussing aspects of Java (syntax, error messages, etc.) that are not specific to the assignment.
- Sharing test cases with classmates.
- Looking into the reference site.

NOT Acceptable:

- Copying code from the internet.
- Copying code from a classmate.
- Sharing solution code with a classmate.
- Consulting resources that are not explicitly allowed by the Professor. (e.g. copying from online or offline matrials and past issues)
- Telling (or being told by) a classmate exactly how to solve the homework (even if you do not actually give/receive code).

Prerequisites

- Object-oriented programming
  - All programming assignments should be in Java
  - But I am not going to teach you how to program in Java in the class (We have too many more important things to discuss than a few syntax differences between C++ and Java)
- Basic Math
  - Proof technique (inductive proof, contradiction proof)
  - Basic probability
  - And so on
- I-can-do-it mindset
  - I am here to shout "you can do it" but it will be you at the end who DO
  - Work hard: 1-to-3 rule
  - Take initiative
    - Read the textbook—it's really well-written
    - Learn the details by yourself

Inductive Proof

All horses are the same color

- Base case: One horse
  - The case with just one horse is trivial. If there is only one horse in the "group", then clearly all horses in that group have the same color.
- Inductive step
  - Assume that n horses always are the same color. Let us consider a group consisting of $n+1$ horses
  - First, exclude the last horse and look only at the first n horses; all these are the same color since n horses always are the same color. Likewise, exclude the first horse and look only at the last n horses. These too, must also be of the same color. Therefore, the first horse in the group is of the same color as the horses in the middle, who in turn are of the same color as the last horse. Hence the first horse, middle horses, and last horse are all of the same color, and we have proven that:
  - If n horses have the same color, then $n+1$ horses will also have the same color.
  - We already saw in the base case that the rule ("all horses have the same color") was valid for $n = 1$. The inductive step showed that since the rule is valid for $n = 1$, it must also be valid for $n = 2$, which in turn implies that the rule is valid for $n = 3$ and so on
  - Thus in any group of horses, all horses must be the same color

Course Schedule

- 1st: Introduction and Big-O notation
- 2nd – 4th: Linked lists, stacks, queues
- 5th – 8th: trees, heaps, midterm
- 9th – 14th: BST, graphs, sorting, pattern matching
- 15th – 16th: Self-study, final exam

Motivation and Overview

The course will let you

- Familiarize common data structures (e.g.: arrays, lists, graphs, trees and so on) that are often used in software development in practice
- Familiarize common algorithms based on these data structures
- Be able to calculate the theoretical runtime (Big-O) and pick the appropriate data structures for a given programming problem
- Practice Java programming

Overview of Unit

① Starters ✓

② Motivation for course ✓

③ Overview of course

④ Big-O notation

Homework Announcement

- Implement 'sorting' in Java
- Due: Mar. 11, 13:00pm

Introduction

Data structures?

- A data structure is an (efficient) way of organizing data for computer
- The whole process of classifying data according to the data property, organizing, storing and processing it in order to use data efficiently

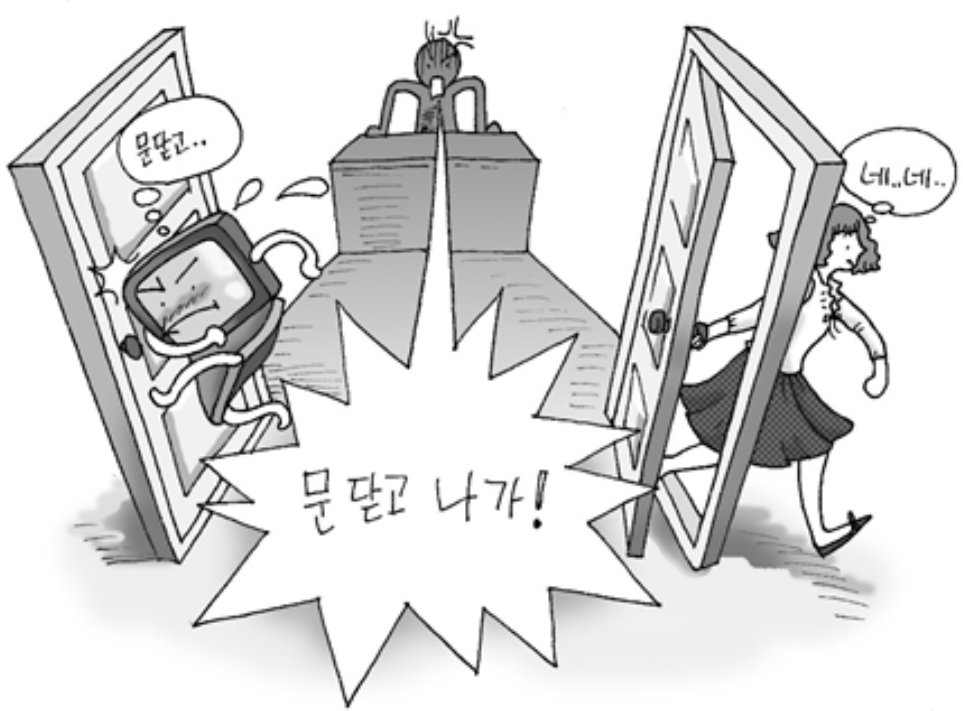
unorganized data structure

organized data structure

Introduction

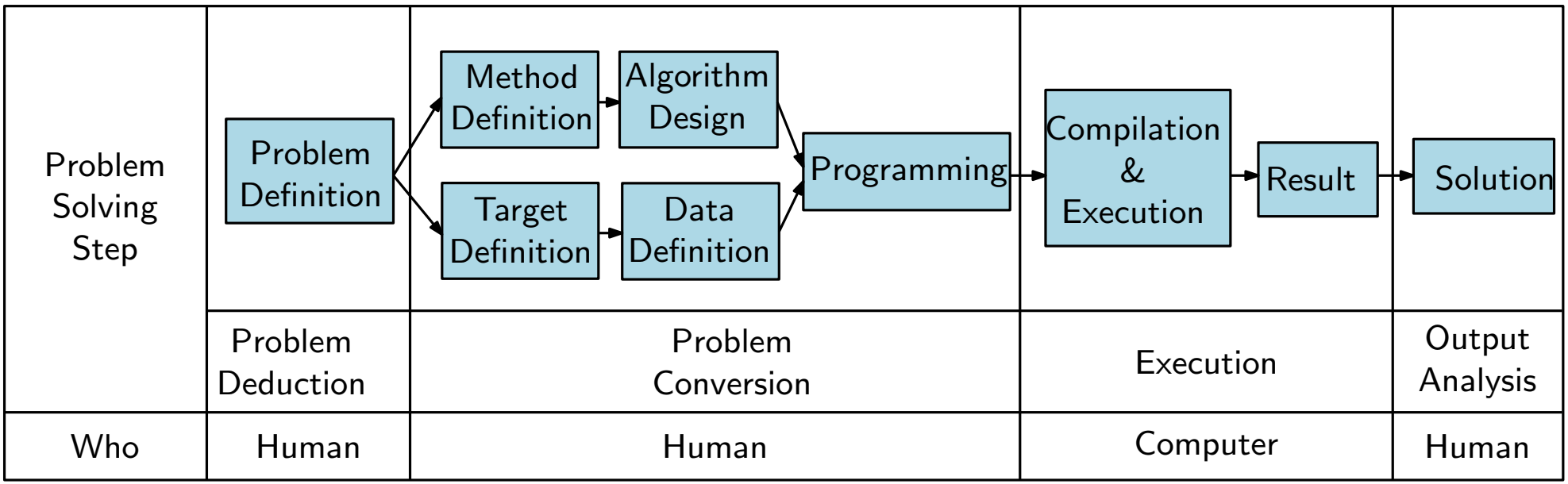
Why do we need to learn data structures in computer science

- The computer cannot understand what you want well
  e.g.: "Close the door and get out"



Introduction

The problem solving steps using computer



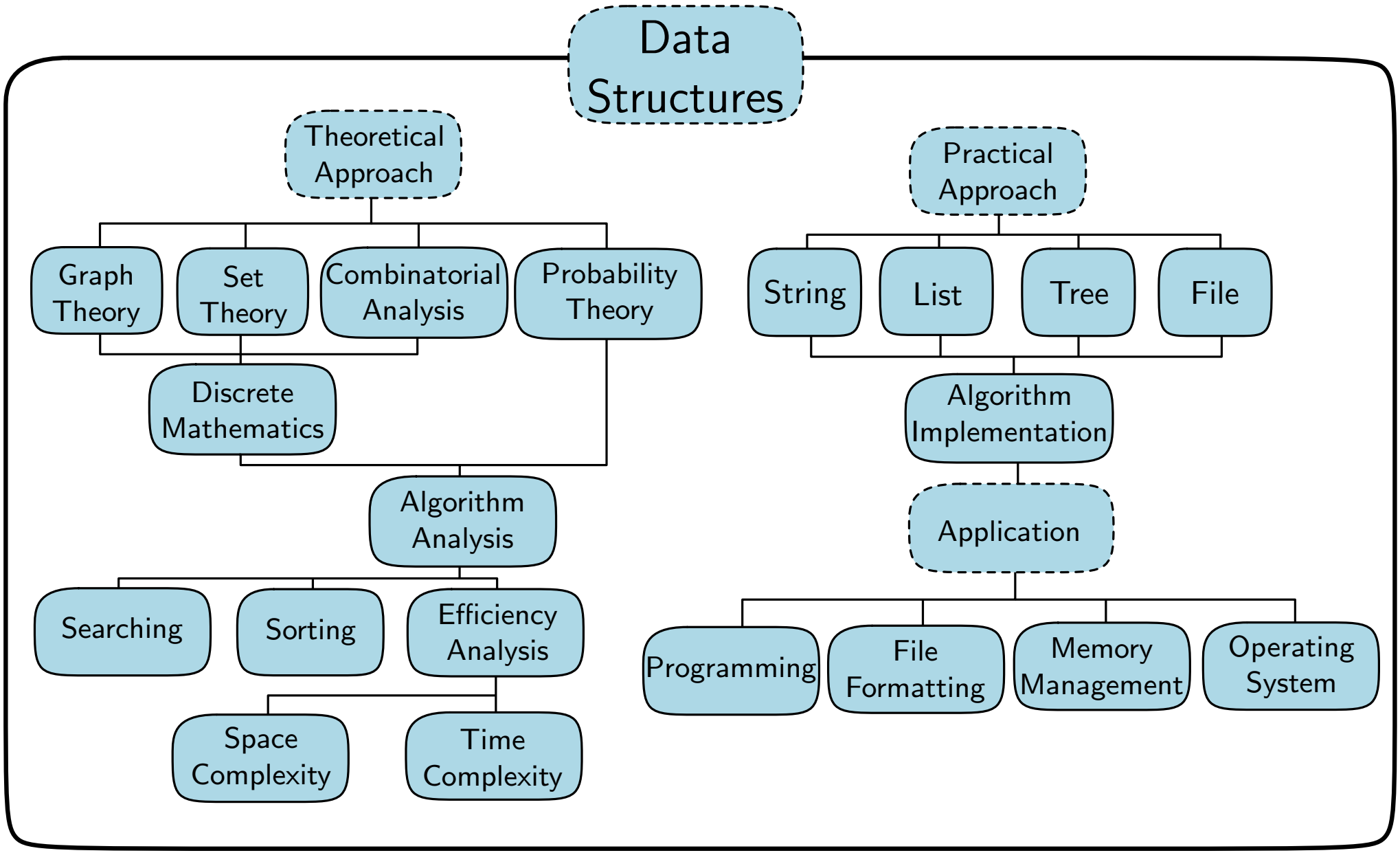
For the computer to process efficiently, it is crucial to write an efficient program, which can be done by well-defining and analyzing the problem.

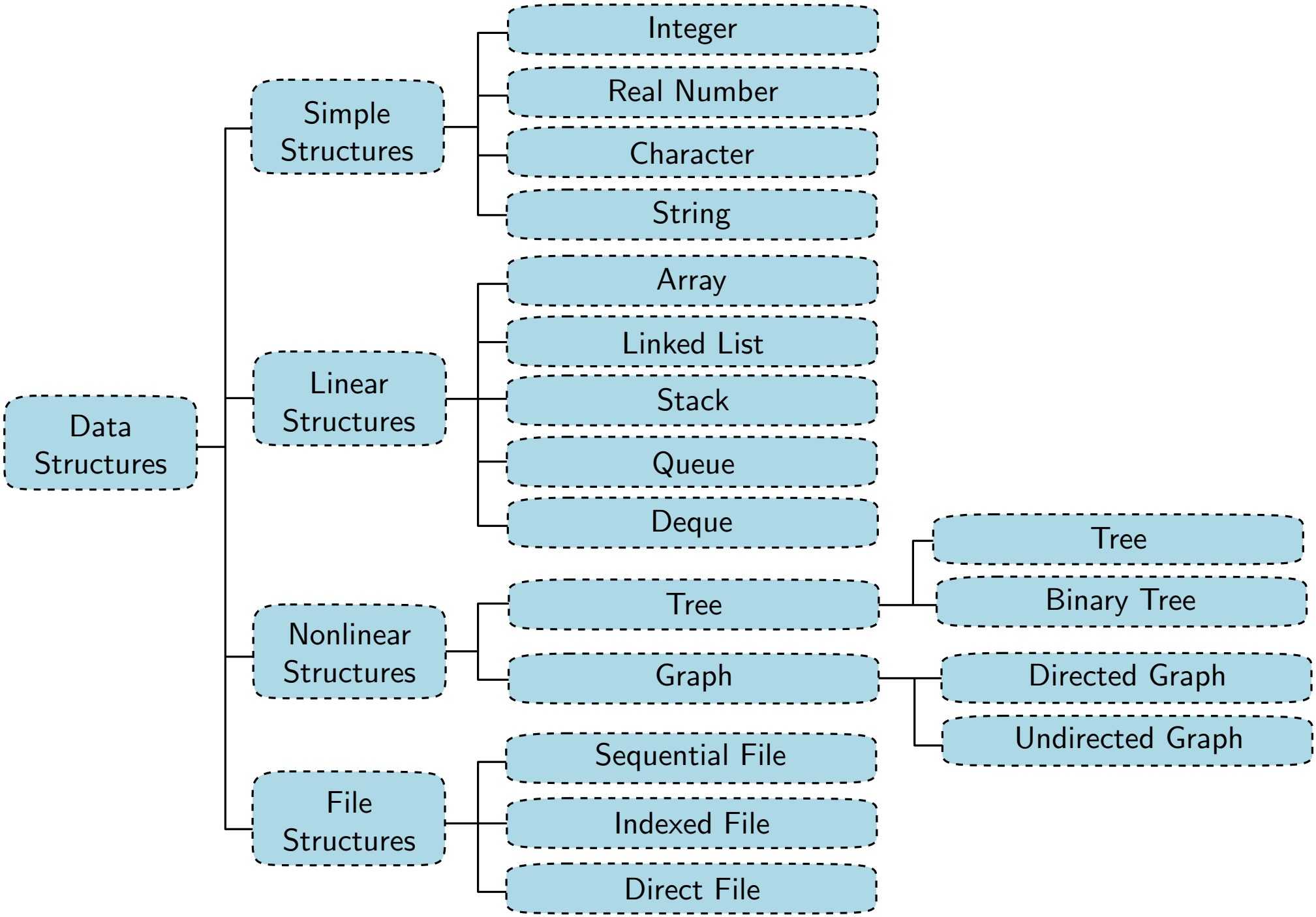
Introduction

Classification by data form

- Simple Structures
  - Primitive data types: e.g., integers, floating point numbers, characters, and strings
- Linear Structures
  - 1-to-1 relationship between front and rear components
  - Lists, linked lists, stacks, queues, deques, and so on
- Nonlinear Structures
  - 1-to-N or M-to-N relationship between front and rear components
  - Trees, graphs, and so on
- File Structures
  - Set of records
  - Sequential files, indexed files, direct files, and so on

Introduction




Data Structures
Simple Structures
Integer
Real Number
Character
String
Linear Structures
Array
Linked List
Stack
Queue
Deque
Nonlinear Structures
Tree
Tree
Binary Tree
Graph
Directed Graph
Undirected Graph
File Structures
Sequential File
Indexed File
Direct File

Data Structures VS Algorithms

Two sides of the same coin

- Data structures: A data structure is an (efficient) way of organizing data for computer
- Algorithms: An algorithm is an (efficient) way of solving a problem

You can design a better algorithm using an appropriate data structure for a given problem

You may use a different data structure for a given algorithm and obtain a better performance

Data Structures VS Algorithms

Two sides of the same coin

- Data structures: A data structure is an (efficient) way of organizing data for computer
- Algorithms: An algorithm is an (efficient) way of solving a problem

PERFORMANCE!!!

You can design a better algorithm using an appropriate data structure for a given problem

You may use a different data structure for a given algorithm and obtain a better performance

(Computer) Programs

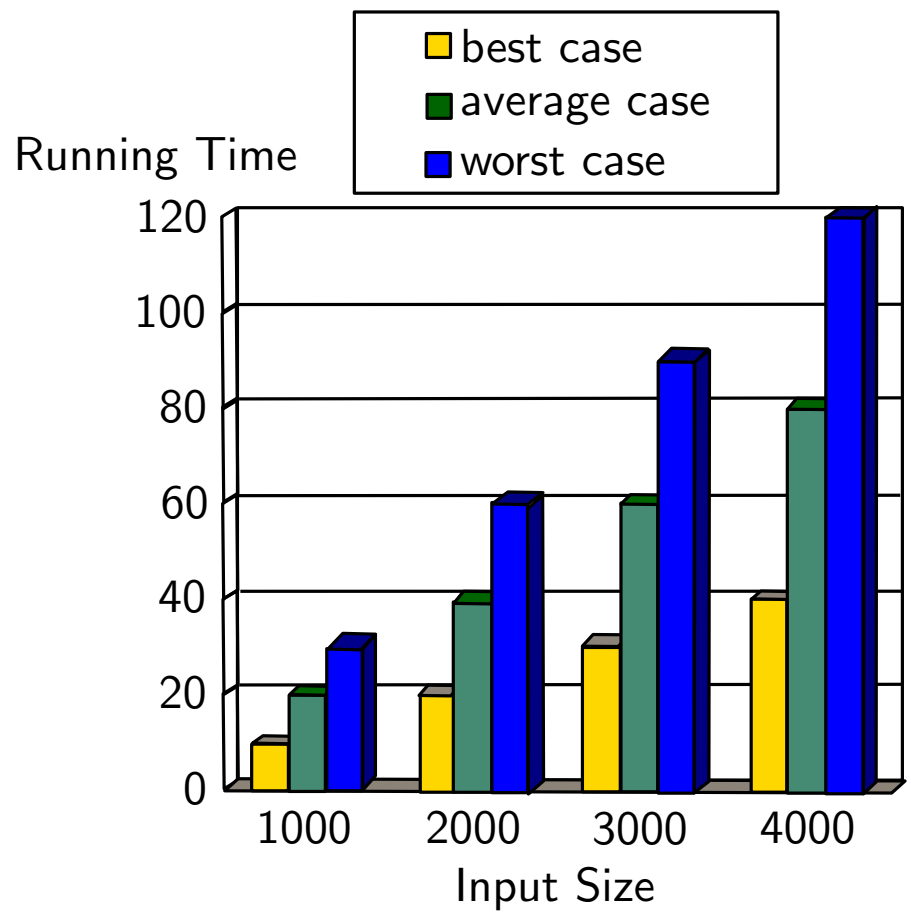
- algorithms
- data structures
- languages

Performance: What do we measure?

- Correctness $\rightarrow$ YES
- Efficiency—What do we mean?
  - Little main memory
  - Little time
  - Little interaction delay
  - Few disk access
  - Few resources
  - In other words, we measure
    - TIME (how FAST) and
    - SPACE (how SMALL)

How to measure program efficiency?

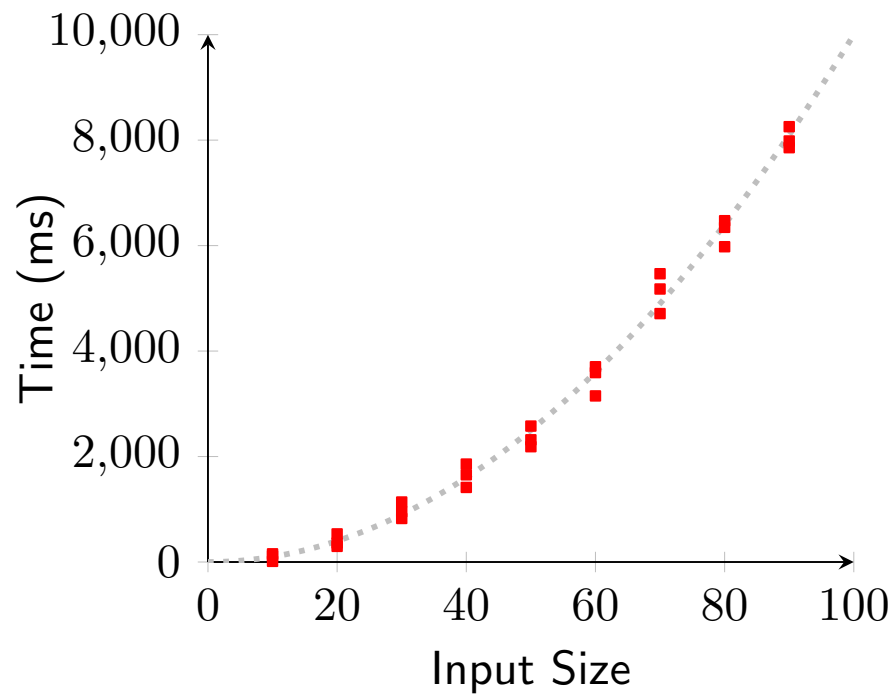
- Most programs receive input data and return output data
- The runtime of a program often grows with the input size
- It's not easy to calculate the average-case runtime
- We measure the worst-case runtime
  - Often easier to analyze
  - Crucial to applications such as finance, medical or robotics



Experimental Measurement

1. Run the program with inputs of varying size and composition, noting the time needed:
2. Plot the results

```
1  long startTime = System.currentTimeMillis();      // record the starting time
2   /*(run the algorithm)*/
3  long endTime = System.currentTimeMillis();        // record the ending time
4  long elapsed = endTime - startTime;               // compute the elapsed time
```



Experimental Measurement: Limits

You should always write a program to see the efficiency of algorithms or data structures

- Time consuming
- Hardware dependency
- Language dependency
- Compiler dependency
- OS dependency
- Many other dependency

Results may not be indicative of the runtime on other inputs that you have not tested

- You should try all possible inputs (almost impossible)

Theoretical Analysis

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

Pseudocodes

- 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

Pseudocode Details

Control flow

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

Method declaration

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

Method call

method(arg[, arg ...])

Return value

return *expression*

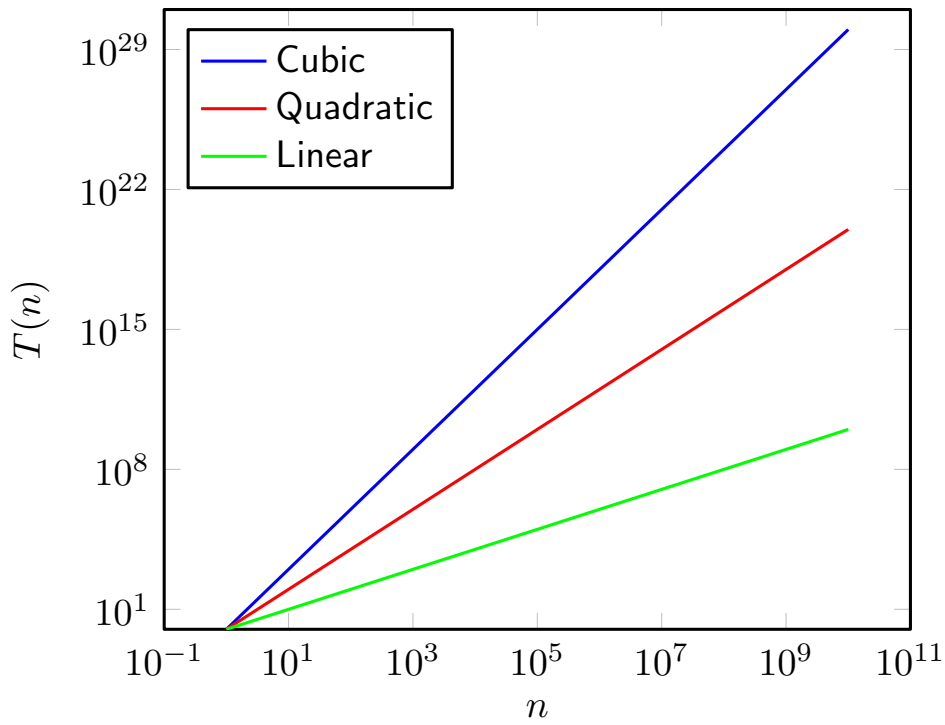
Expressions:

$\leftarrow$ Assignment
$=$ Equality testing

Seven Important Functions

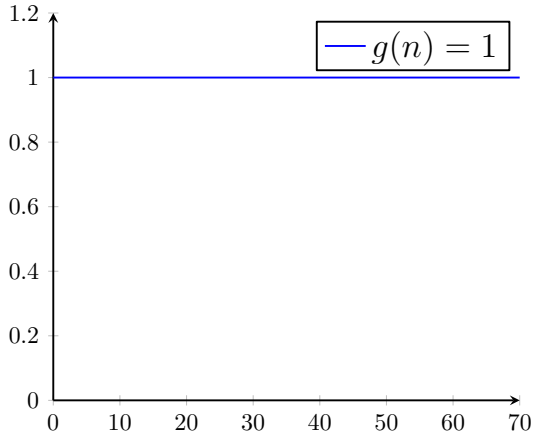
Seven functions that often appear in algorithm analysis:

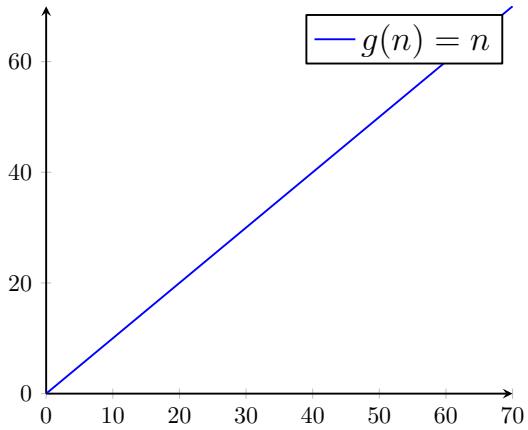
- Constant ≈ 1
- Logarithmic $\approx \log n$
- Linear $\approx n$
- N-Log-N $\approx n \log n$
- Quadratic $\approx n^2$
- Cubic $\approx n^3$
- Exponential $\approx 2^n$

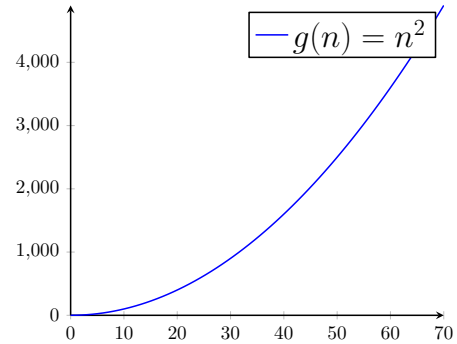


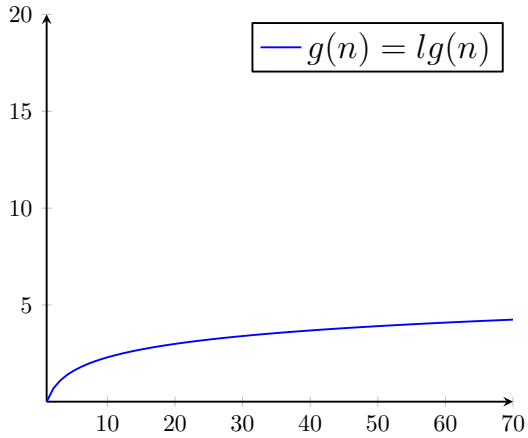
In a log-log chart, the slope of the line corresponds to the growth rate

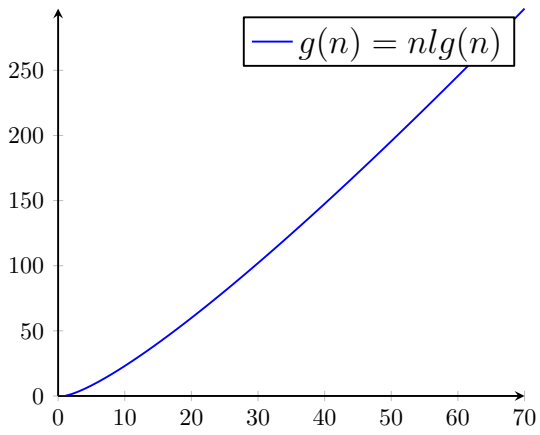
Functions Graphed Using Normal Scale

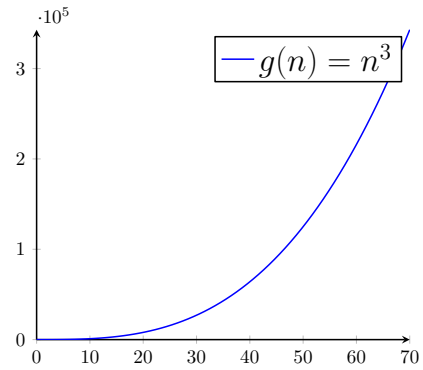


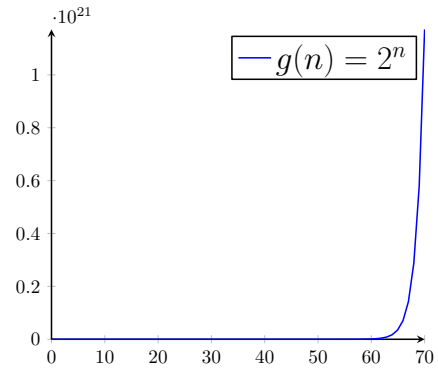












Primitive Operations

- Basic computations performed by an algorithm
- Identifiable in pseudocode
- Largely independent from the programming language
- Exact definition is 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

```
1  /** Returns the maximum value of a nonempty array of numbers. */
2  public static double arrayMax(double[ ] data){
3    int n = data.length;
4    double currentMax = data[0];          // assume the first entry is the biggest (for now)
5    for (int i = 1; i < n; i ++)          // consider all other entries
6      if (data[i] > currentMax)           // if data[i] is the biggest thus far ...
7        currentMax = data[i];             // record it as the current max
8    return currentMax;
9  }
```

Step 3: 2 ops, 4: 2 ops, 5: $2n$ ops, 6: $2n-2$ ops, 7: 0 to $2n-2$ ops, 8: 1 op

Estimating Running Time

Algorithm arrayMax executes $6n + 1$ primitive operations in the worst case, $4n + 3$ in the best case. Define:

- $a =$ Time taken by the fastest primitive operation
- $b =$ Time taken by the slowest primitive operation

Let $T(n)$ be the worst-case runtime of arrayMax. Then

$$a(4n + 3) \leq T(n) \leq b(6n + 1)$$

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

Growth Rate of Running Time

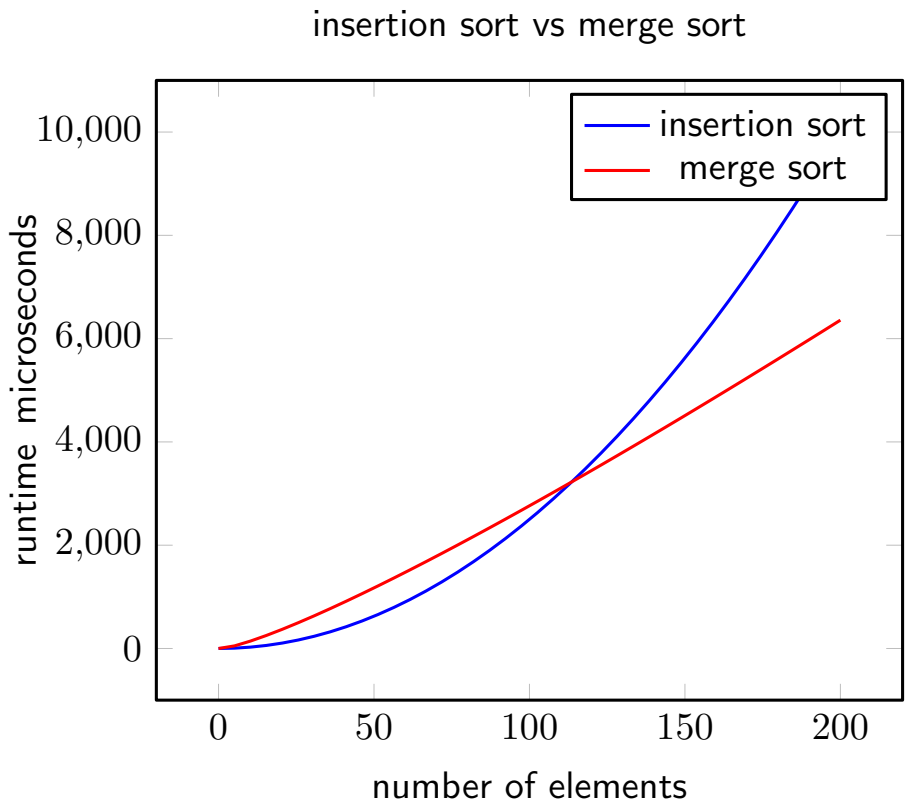
Changing the hardware/software environment

- Affects $T(n)$ by a constant factor, but
- Does not alter the growth rate of $T(n)$

The linear growth rate of the runtime $T(n)$ is an intrinsic property of algorithm arrayMax

Comparison of Two Algorithms

- insertion sort is $\frac{n^2}{4}$
- merge sort is $2n \log n$
- sort a million items?
  - insertion sort takes roughly 70 hours
  - merge sort takes roughly 40 seconds
- This is a slow machine, but if 100x as fast, then it's 40 minutes versus less than 0.5 seconds



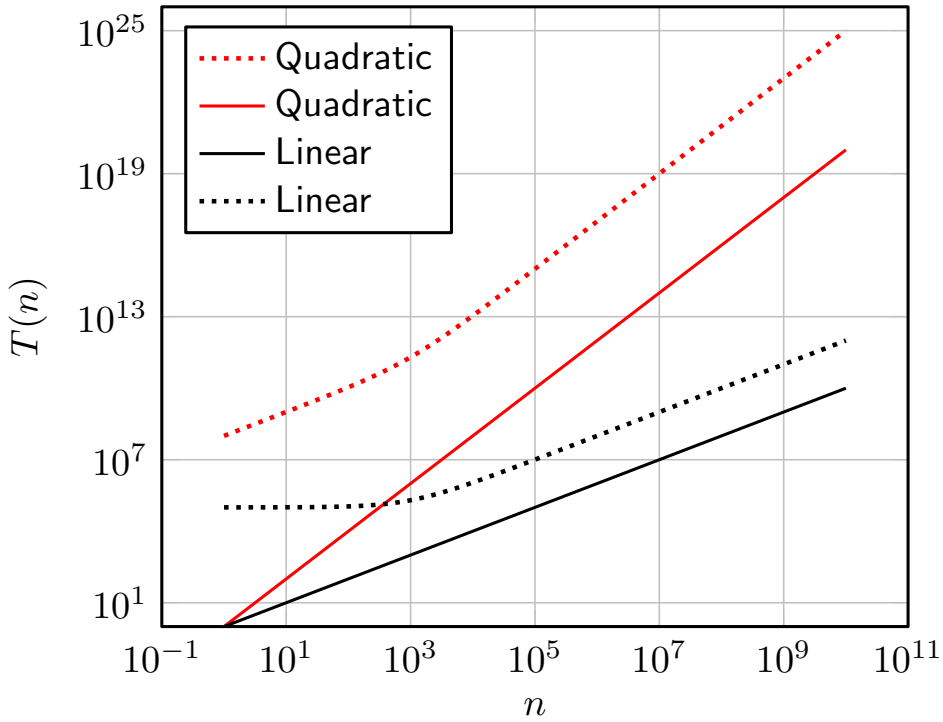
Constant Factors

The growth rate is not affected by

- constant factors or
- lower-order terms

Examples

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



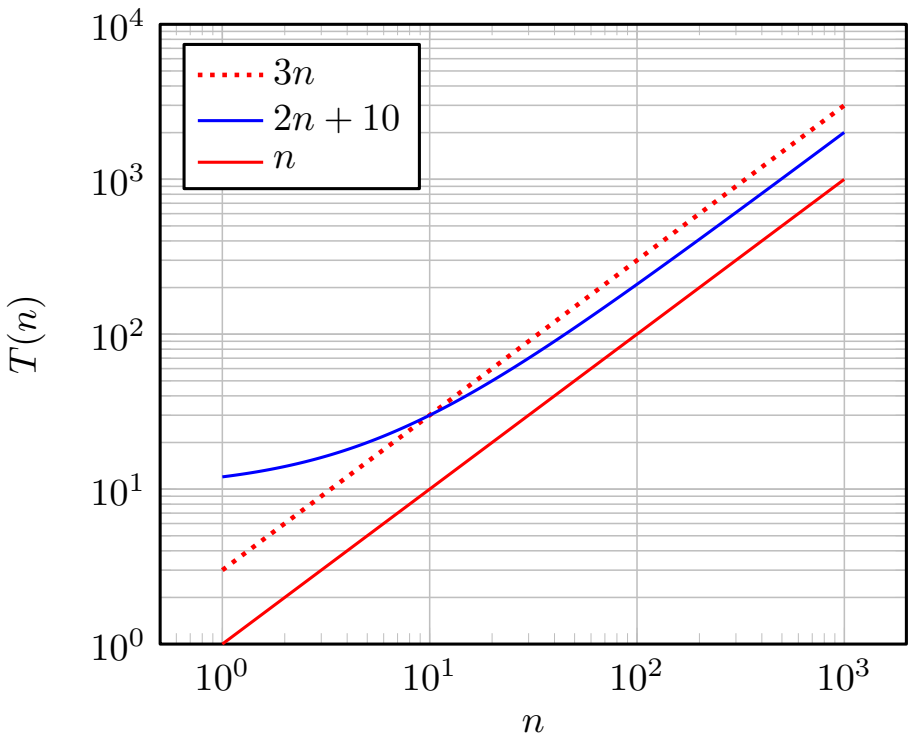
Big-O Notation

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

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

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

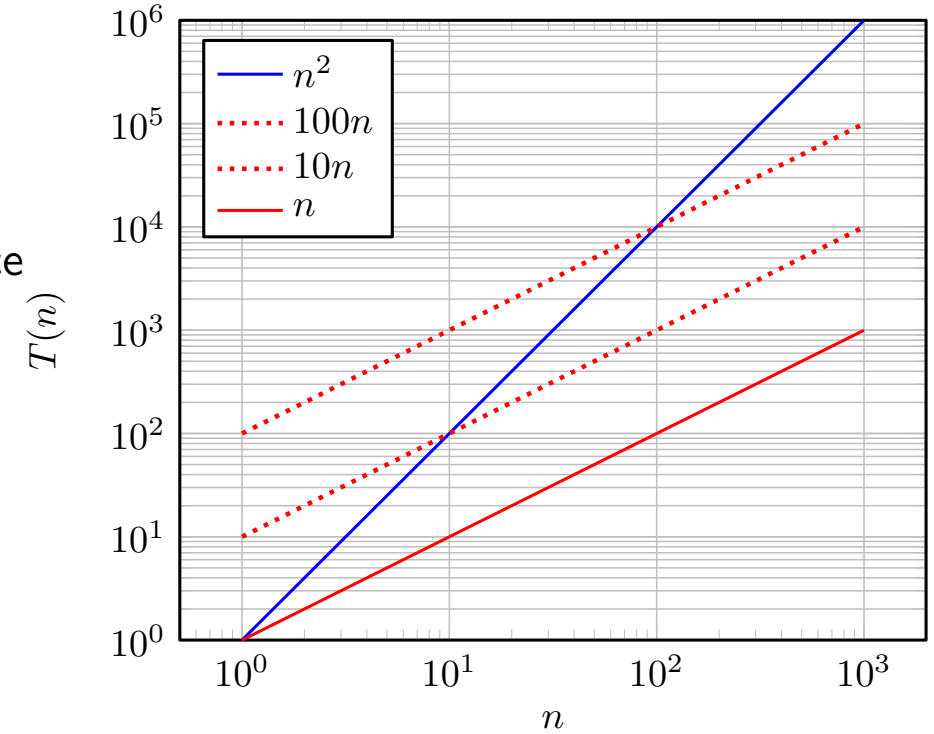
- $2n + 10 \leq cn$
- $(c - 2)n \geq 10$
- $n \geq 10/(c - 2)$
- Pick $c = 3$ and $n_0 = 10$



Big-O Notation

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-O Examples

$7n - 2$

- $7n - 2$ is $O(n)$
- need $c > 0$ and $n_0 \geq 1$ such that $7n - 2 \leq cn$ for $n \geq n_0$
- this is true for $c = 7$ and $n_0 = 1$

$3n^3 + 20n^2 + 5$

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

$3 \log n + 5$

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

Big-O and Growth Rate

- The big-O 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-O 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-O Rules

- If $f(n)$ is a polynomial of a degree d, then $f(n)$ is $O(n^d)$, i.e.,
  - 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

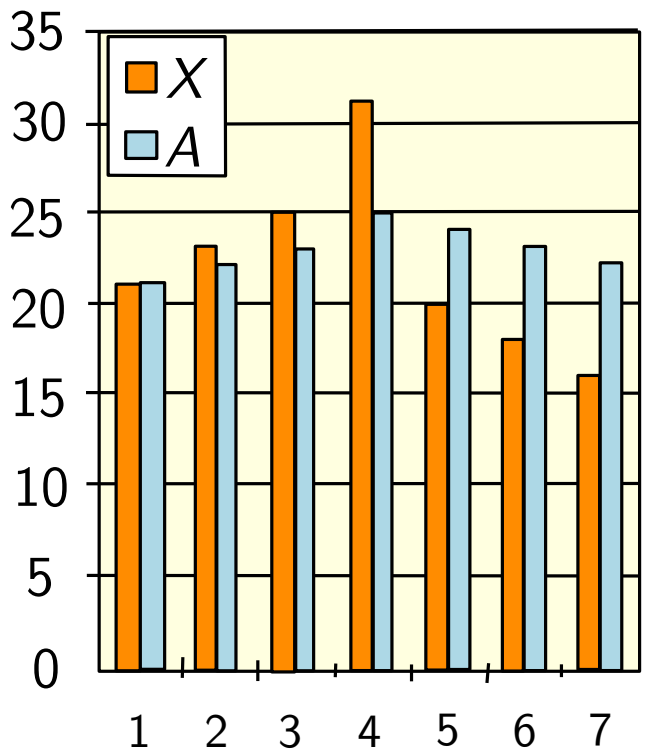
- The asymptotic analysis of an algorithm determines the runtime in big-O notation
- To perform the asymptotic analysis
  - We find the worst-case number of primitive operations executed as a function of the input size
  - We express this function with big-O notation
- Example:
  - We say that algorithm 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

- The i-th prefix average of an array X is an average of the first $(i+1)$ elements of X:

$$A[i] = \frac{X[0] + X[1] + \cdots + X[i]}{i+1}.$$

- Computing the array A of prefix averages of another array X



Prefix Averages (Quadratic)

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

```
1  /** Returns an array a such that, for all i, a[i] equals the average of x[0], ..., x[i]. */
2  public static double[ ] prefixAverage1(double[ ] x){
3    int n = x.length;
4    double[ ] a = new double[n];          // filled with zeros by default
5    for (int i = 0; i < n; i ++) {
6      double total = 0;                   // begin computing x[0] + ... + x[i]
7      for (int j = 0; j <= i; j ++)
8        total += x[j];
9      a[i] = total / (i+1);               // record the average
10   }
11   return a;
12 }
```

Arithmetic Progression

- The running time of prefixAverage1 is $O(1 + 2 + \cdots + n)$
- The sum of the first n integers is $n(n + 1)/2$
- Thus, algorithm prefixAverage1 runs in $O(n^2)$ time

```
1  /** Returns an array a such that, for all i, a[i] equals the average of x[0], ..., x[i]. */
2  public static double[ ] prefixAverage1(double[ ] x){
3     int n = x.length;
4     double[ ] a = new double[n];          // filled with zeros by default
5     for (int i = 0; i < n; i ++)
6        double total = 0;                  // begin computing x[0] + ... + x[i]
7        for (int j = 0; j <= i; j ++)
8           total += x[j];
9        a[i] = total / (i+1);              // record the average
10    }
11    return a;
12 }
```

Prefix Averages 2 (Linear)

The following algorithm uses a running summation to improve the efficiency

```
1  /** Returns an array a such that, for all i, a[i] equals the average of x[0], ..., x[i]. */
2  public static double[ ] prefixAverage2(double[ ] x){
3    int n = x.length;
4    double[ ] a = new double[n];        // filled with zeros by default
5    double total = 0;                   // compute the prefix sum as x[0] + x[1] + ...
6    for (int i = 0; i < n; i ++) {
7        total += x[i];                  // update the prefix sum to include x[i]
8        a[i] = total / (i+1);           // compute the average based on current sum
9     }
10     return a;
11 }
```

Algorithm prefixAverage2 runs in $O(n)$ time!

Relatives of Big-O

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

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

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

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

Intuition for Asymptotic Notation

- big-O: $f(n)$ 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 $g(n)$

Example Uses of the Relatives of Big-O

- $5n^2$ is $O(\quad)$
- $5n^2$ is $\Omega(\quad)$
- $5n^2$ is $\Theta(\quad)$

- big-O: $f(n)$ 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 of Unit

- Starters
- Motivation for course
- Overview of course
- Big-O notation