Data Structures

What This Course Is About

- Data structures is about
  - representation of data
  - manipulation of data
- All programs manipulate data
  - So all programs represent data in some way.
- Then a program uses algorithm(s) to manipulate data
  - Think about OOP:
    - Data members (corresponding to data)
    - Member functions (corresponding to algorithms)
  - So data 是菜，algorithms 則像廚師的煮菜方法或工具, program (programmer) 則是廚師

Illustration

- Data: E.g., Streets of a city in Google Map
- Manipulation: Do something with the data. E.g.,
  - Add a new street
  - Delete a street
  - Make a street one way
  - Find route from your home to a gas station
  - Compute the route cost, etc.

What This Course Is About (cont.)

- Data representation:
  - simple variables: int, float
  - array variables: int [], float []
  - others to be studied
    - linked lists
    - queues
    - stacks
    - trees
    - graphs
    - hash tables
    - search structues…

What This Course Is About (cont.)

- Algorithm:
  - sequence of steps to solve a given problem
  - e.g., compute a best policy to spend your money
    - Your money is data
    - The policy becomes an algorithm if concrete steps are specified to implement the policy.

What This Course Is About (cont.)

- Knowledge of data structures and algorithms are fundamental to Computer Science students

- In this course, we study how to
  - represent data and
  - use algorithms (basics) to manipulate these representations

- Advanced discussions for algorithms are available in higher-level courses

Prerequisites

- C++
- Asymptotic Complexity
  - Big Oh, Theta, and Omega notations
    - To compute time and space complexity

Website

- Teaching material and homework assignments available on E-learning
- TA offers additional assistance

Chapter 1
Basic Concepts

Strategic Viewpoint on Writing Code

- Strategic point of view: Serious programming tasks require good software engineering principles

- So we briefly review relevant software engineering principles before discussing data structures

Software Engineering and Object-Oriented Design

- Refer to the reference textbook below for material covered here
  - Data Abstraction & Problem Solving with C++ (5th Edition) by Frank Carrano
  - Library has it.

- Coding without a solution design increases debugging time

Software Engineering and Object-Oriented Design

- A large software development project requires
  - An overall plan
  - Organization of codes
  - Efficient communication between various modules of the program
  - Team members work together well
  - Final project will give you the chance to practice this.
- Software engineering
  - Provides techniques/guidelines to facilitate the development of computer programs
- Side note: If you pass CPE, it means you are already equipped with skills to solve some problems. To become a coding master, in the future you may rely a lot on the practice of software engineering.

An Examination of Problem Solving

- Problem solving
  - The process of
    - taking the statement of a problem and
    - developing a computer program that solves the problem
- Object-oriented analysis and design (OOA/D)
  - A process and a thinking/working style for problem solving
  - A problem solution is a program consisting of a system of interacting classes of objects
    - Each object has characteristics and behaviors related to the solution
    - Objects commute with each other effectively

Aspects of an Object-Oriented Solution

- A solution module in a C++ program may consist of
  - Stand-alone functions
  - Classes
  - Several functions or classes working closely together
  - Other blocks of code

Aspects of an Object-Oriented Solution

- Create a good set of modules
  - Modules must store, move, alter and delete data
  - Modules may use means (e.g., algorithms) to communicate with one another
- It's important to organize your data collection to have a good structure to
  - facilitate operations on the data (algorithms)

Abstraction

- Abstraction
  - Separates the purpose of a module from its implementation
  - Specifications/operations (what to accomplish) for each module are written before implementation

Abstraction

– Abstract data type (ADT):

  - Is a collection of data and operations on the data
  - You can use an ADT's operations without knowing their implementations or how data is stored, if you know the operations' specifications

Abstraction

- Data structure
  - A construct that you design for a program to store a collection of data

Object-Oriented Analysis and Design

- Analysis
  - Process to develop
    - An understanding of the problem
    - The requirements of a solution
      - *What* a solution must be and do
      - Not *how* to design or implement it
  - Generates an accurate understanding of what end users will expect the solution to be and do
  - This is like the sales department in a company surveys what customers would like without knowing how to do it.
    - How to do it is a company's R&D's task.
  - In short, think about how to formulate the problem, not how to solve it

Object-Oriented Analysis and Design

- Object-oriented analysis (OOA)
  - Expresses an understanding of the problem and the requirements of a solution in terms of objects
  - Objects can represent
    - Real-world objects
    - Software systems
    - Ideas
    - Partial solutions → building blocks to build something bigger
  - OOA describes objects and their interactions

Object-Oriented Analysis and Design

- Object-oriented design (OOD)
  - Expresses an understanding of a solution that fulfills the requirements discovered during OOA
  - Describes a solution in terms of
    - Software objects
    - The collaborations of these objects
      - Objects collaborate when they send messages (call each other's operations)
      - Collaborations should be meaningful and minimal
  - Creates one or more models of a solution
    - Some emphasize interactions among objects
    - Others emphasize relationships among objects

The Software System Life Cycle

- Describes the phases of software development from ‘conception’ to ‘deployment’ to ‘replacement’ to ‘deletion’
  - We will examine the phases from project conception to deployment to end users
  - Replacement: Software needs maintenance to correct errors and add features
  - Deletion: Eventually software is retired
  - Think about Windows OS

Operation Contracts

- A term used in Software Engineering
- A module's operation contract specifies its
  - Purpose
  - Assumptions
  - Input
  - Output
- Used to document code, particularly in header files

Operation Contracts

- Specify data flow among modules
  - What data is available to a module?
  - What does the module assume?
  - What actions take place?
  - What effect does the module have on the data?

Operation Contracts

- Contract shows the responsibilities of one module to another
- Does *not* describe how the module will perform its task
- Precondition: Statement of conditions that must exist before a module executes
- Postcondition: Statement of conditions that exist after a module executes

Operation Contracts – Example

First draft specifications

```
sort(anArray, num)
// Sorts an array.
// Precondition: anArray is an array of n integers; n > 0.
// Postcondition: The integers in anArray are sorted.
```

Operation Contracts – Example

Revised specifications

```
sort(anArray, num)
// Sorts an array into ascending order.
// Precondition: anArray is an array of n integers; 1 <= n  <=
// MAX_ARRAY, where MAX_ARRAY is a global constant that
   specifies the maximum size of anArray.
// Postcondition: anArray[0] <= anArray[1] <= … <= anArray[n -
   1].
```

Verification

- Assertion: A statement about a particular condition at a certain point in an algorithm
  - Preconditions and postconditions are examples of assertions
- Invariant: A condition that is always true at a certain point in an algorithm
  - E.g., Does the value of π become negative?
- Loop invariant: A condition that is true before and after each execution of an algorithm's loop

Verification

- It is possible to prove the correctness of some algorithms
  - Like proving a theorem in geometry
  - Starting with a precondition, you prove that each assertion before a step in an algorithm leads to the assertion after the step until you reach the postcondition

What is a Good Solution?

- A solution is good if:
  - The total cost it incurs over all phases of its life cycle is minimal
- The cost of a solution includes:
  - Computer resources that the program consumes
  - Difficulties encountered by users
  - Consequences of a program that does not behave correctly
- Programs must be well structured and documented
- Efficiency is one aspect of a solution’s cost

Key Issues in Programming

1. Modularity
2. Style
3. Ease of Use
4. Fail-safe programming
5. Debugging
6. Testing

Key Issues in Programming: Modularity

- Modularity has impact on
  - Construction of programs
  - Debugging of programs
  - Readability of programs
  - Modification of programs
  - Elimination of redundant codes

Key Issues in Programming: Style

1. Use of private data members
2. Proper use of methods
3. Proper use of reference arguments
4. Avoidance of global variables in modules
5. Error handling
6. Readability
7. Documentation

Key Issues in Programming: Ease of Use

- In an interactive environment, the program should prompt the user for input in a clear manner
- A program should always echo its input
- The output should be well labeled and easy to read

Key Issues in Programming: Fail-Safe Programming

- Fail-safe programs will perform reasonably no matter how anyone uses it
- Test for invalid input data and program logic errors
- Check invariants
- Enforce preconditions/postconditions
- Check argument values
- These issues are important when you work on term projects

Key Issues in Programming: Debugging

- Programmers must systematically check a program's logic to find where an error occurs

- In large systems, this is a difficult issue.

(E.g., evolution of Windows OS)

- Tools to use while debugging, e.g.,
  - Single-stepping
  - Breakpoints
  - `cout` statements
  - Dump functions

Key Issues in Programming: Testing

- Levels of testing
  - Unit testing: Test methods in classes, then classes per se
  - Integration testing: Test interactions among modules
  - System testing: Test entire program
  - Acceptance testing: Show that system complies with requirements

Key Issues in Programming: Testing

- Types
  - Open-box (white-box) testing
    - Test knowing the implementation
    - Test all lines of code
  - Closed-box (black-box) testing
    - Test knowing only the specifications

Key Issues in Programming: Testing

- Developing test data
  - Include boundary values
  - Need to know expected results

- Techniques
  - assert statements to check invariants

Key Issues in Programming: Testing

- Stubs
  - An incompletely implemented method for acknowledging that it was called
- Drivers
  - A module that tests another module
  - E.g., a `main` function

Summary of Software Engineering

- Software engineering
  – Methods to facilitate development of programs
- Software life cycle
  – Phases through which software progresses, from (1) conception to (2) deployment to (3) replacement to (4) deletion
- Loop invariant
  – Property that is true before and after each iteration of a loop
- Evaluating the quality of a solution
  – Correctness, efficiency, development time, ease of use, cost of modification

Summary of Software Engineering

- Solutions should be easy to modify
  - Modular
  - Each function/method should be as independent as possible and perform one well-defined task

- Make coding clear:
  - Each function/method should have comments: purpose, precondition and postcondition

Summary of Software Engineering

- A program should be as fail-safe as possible
- Effective use of available diagnostic aids to make debugging easier
- Use “dump functions” to help examine and debug the contents of data structures

Algorithms

- Definition
  - A finite set of instructions to accomplish a particular task

- Criteria
  - Input
  - Output
  - Definiteness: clear and unambiguous
  - Finiteness: terminate after a finite number of steps
  - Effectiveness: instructions basic enough to be carried out

Algorithms

- Difference between an algorithm and a program:
  - program does not have to satisfy the 4th condition; i.e.,
    - Program doesn’t have to be finite

- Examples:
  - ‘Sorting’ a sequence of data
  - ‘Search’ for a particular item in a dataset
  - Higher-order Artificial Intelligence algorithms, etc.

Performance Evaluation

- How to evaluate a program’s performance?
  - Using “sorting” as an example
    - Data structure: Array
    - Algorithm: Sorting
    - Performance evaluation: Time complexity

Sorting

- Goal: Rearrange a[0], a[1], ..., a[n-1] into ascending order.
  - When done, a[0] <= a[1] <= ... <= a[n-1]
- E.g., 8, 6, 9, 4, 3 => 3, 4, 6, 8, 9

Sorting Methods

- Insertion Sort
- Selection Sort
- Quick Sort
- Bubble Sort
- Heap Sort
- Count Sort
- Shaker Sort
- Shell Sort
- Merge Sort

To be used for illustration

Insertion Sort – Key Idea

- Key idea:
  - Given a sorted list
  - Want to insert a new element, then
  - Sort
- E.g., given a sorted list 3, 6, 9, 14
  - Want to insert 5
  - Then want to get: 3, 5, 6, 9, 14

Insertion Sort – Key Idea

- How to do it?
  - Add the new element (5) to the end of the list
  - Compare 5 and the last element (14) of the list
    - 5<14 → Shift 14 right to get 3, 6, 9, , 14 (don't insert 5 yet)
  - Compare 5 with 9 and
    - 5<9 → Shift 9 right to get 3, 6, , 9, 14 (don't insert 5 yet)
  - Compare 5 with 6 and
    - 5<6 → Shift 6 right to get 3, , 6, 9, 14 (don't insert 5 yet)
  - Compare 5 with 3 and
    - 5>3 → Bingo! So insert 5 now to get 3, 5, 6, 9, 14
  - So, the key idea is: compare, shift and insert

Insertion Sort – Key Idea

```
// insert t into a[0:i-1]

int j;
for (j = i - 1; j >= 0 && t < a[j]; j--)
    a[j + 1] = a[j];
a[j + 1] = t;
```

starting from the last one of a, so j=i-1

Search backwards

Compare t and a[j]

∵ t < a[j], so shift a[j] right

∵ t is not < a[j], so insert t here

Insertion Sort – Big Picture

- Start with a sequence of size 1
- Repeatedly insert remaining elements (by the key idea just discussed)

Insertion Sort – Big Picture

- Goal: Sort the sequence 7, 3, 5, 6, 1
  - Start with 7 and insert 3 by the key idea => 3, 7
  - Insert 5 by the key idea => 3, 5, 7
  - Insert 6 by the key idea => 3, 5, 6, 7
  - Insert 1 by the key idea => 1, 3, 5, 6, 7

Insertion Sort – Big Picture

```
for (int i = 1; i < a.length; i++)
{// insert a[i] into a[0:i-1]
   int t = a[i];
   int j;
   for (j = i - 1; j >= 0 && t < a[j]; j--)
       a[j + 1] = a[j];
   a[j + 1] = t;
}
```

Add “t” to the end of the list

Compare, shit and insert (by the key idea discussed)

Complexity

- Space complexity: amount of memory
- Time complexity: Amount of computing time
  - Count a particular operation
  - Count number of steps
  - Asymptotic complexity
- Other types of complexity are possible:
  - E.g., programming complexity, debugging complexity, complexity of comprehension.

Space Complexity

- $S(P) = c + S_p(I)$
  - c: fixed space (instructions, simple variables, constants)
  - $S_p(I)$: depends on characteristics of instance I
    - Characteristics: number, size, values of I/O associated with I
- If n (number of components) is the only characteristic, then $S_p(I) = S_p(n)$

Space Complexity

Program 1.16: Function to compute ***a+b+b*c+(a+b-c)/(a+b)+4.0***

```
float  Abc(float  a, float b, float c)
{
     return  a+b+b*c+(a+b-c)/(a+b)+4.0;
}
```

- To store the values of a, b, c and the value returned by Abc, only need 4 fixed memory units.
- Obviously, the space needed for this function doesn’t depend on the instance characteristic.
- So the space complexity: $S_p(I) = 0$.

Space Complexity

Program 1.18: Recursive function for sum

```
float Rsum(float *a, const int  n)
{
     if (n <= 0) return  0;
     else return  (Rsum(a,n-1) + a[n-1]);
}
```

- *a* is the address of *a*[0] => requires one memory unit on the stack
- Requires one memory unit to store a copy of *n*
- Require one memory unit for the returned value
- Require one memory unit for the returned address

So a recursion requires 4 memory units.

- Since the depth of recursion is n+1, the recursion stack space needed is 4(n+1).
- I.e., space complexity $S_p(I) = 4(n+1)$. E.g., n=1000, the recursion stack space is 4004.

Time Complexity

- $T(P) = c + T_p(I)$
  - c: compile time
  - $T_p(I)$: program execution time
    - Depends on characteristics of instance I

- Predict the <u>growth in run time</u> as the instance characteristics change

Time Complexity – Comparison Count

- Using “comparison count” for time complexity analysis
- Pick an instance characteristic, say *n*,
  - Eg, *n* = “a.length” for insertion sort
- Determine “comparison count” as a function of the instance characteristic.

Comparison Count

```
for (j = i - 1; j >= 0 && t < a[j]; j--)
    a[j + 1] = a[j];
```

- How many comparisons are made?
- Number of comparisons depends on
  - a
  - t
  - i

Comparison Count

- Worst-case count = maximum count
- Best-case count = minimum count
- Average count
- Worst case and average case analysis is much more useful in practice

Worst-Case Comparison Count

```
for (j = i - 1; j >= 0 && t < a[j]; j--)
    a[j + 1] = a[j];
```

- a = [1, 2, 3, 4] and t = 0 => t < a[j] needs 4 comparisons
  - General form: a = [1,2,3,...,i] and t = 0 => t < a[j] needs i comparisons

Worst-Case Comparison Count

```
for (int i = 1; i < n; i++)
   for (j = i - 1; j >= 0 && t < a[j]; j--)
      a[j + 1] = a[j];
```

- Total comparisons (the worst case)

$$= 1 + 2 + 3 + \ldots + (n-1) = (n-1)n/2$$

For i=1,
J=0

For i=2,
J=1, 0

For i=3,
j=2, 1, 0

For i=n-1,
j=n-2, ..., 2, 1, 0

Time Complexity – Step Count

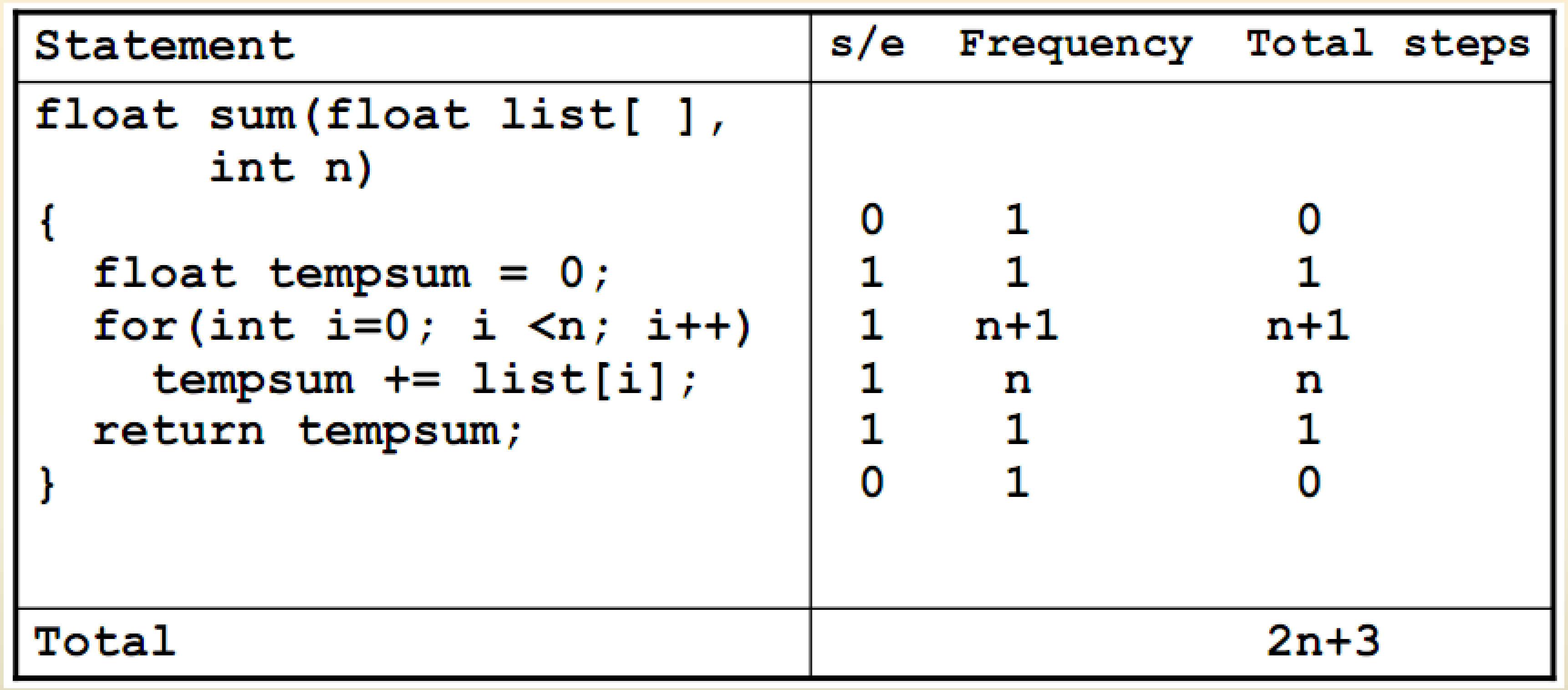
- Another way for time complexity
- A program step is a syntactically or semantically meaningful program segment
  - Its execution time is independent of the instance characteristics.
- “10 adds”, “100 subtracts”, “1000 multiplies” can all be counted as a single step
- “*n* adds” cannot be counted as 1 step, since it depends on *n*

Time Complexity – Step Count

- Just think of pressing "F10" as an analogy in MS VC++

- How to do this:
  - Determine the total number of steps contributed by each statement step per execution (s/e) × frequency, where frequency = number of executions
  - Add up the contribution of all statements

Step Count



Statement	s/e	Frequency	Total steps
`float sum(float list[ ],`			
`          int n)`			
`{`	0	1	0
`  float tempsum = 0;`	1	1	1
`  for(int i=0; i <n; i++)`	1	n+1	n+1
`    tempsum += list[i];`	1	n	n
`  return tempsum;`	1	1	1
`}`	0	1	0
Total			2n+3

- s/e: steps per execution

Problem with Step Count

- Difficult to determine the exact step counts
  - E.g., What a step stands for is inexact, e.g.,
    - x := y v.s. x := y + z + (x/y) + ...
  - So “exact step count” is not useful for comparison
- A better approach is ”asymptotic complexity” for running time calculations
  - See the following

Asymptotic Notation – Big “oh”

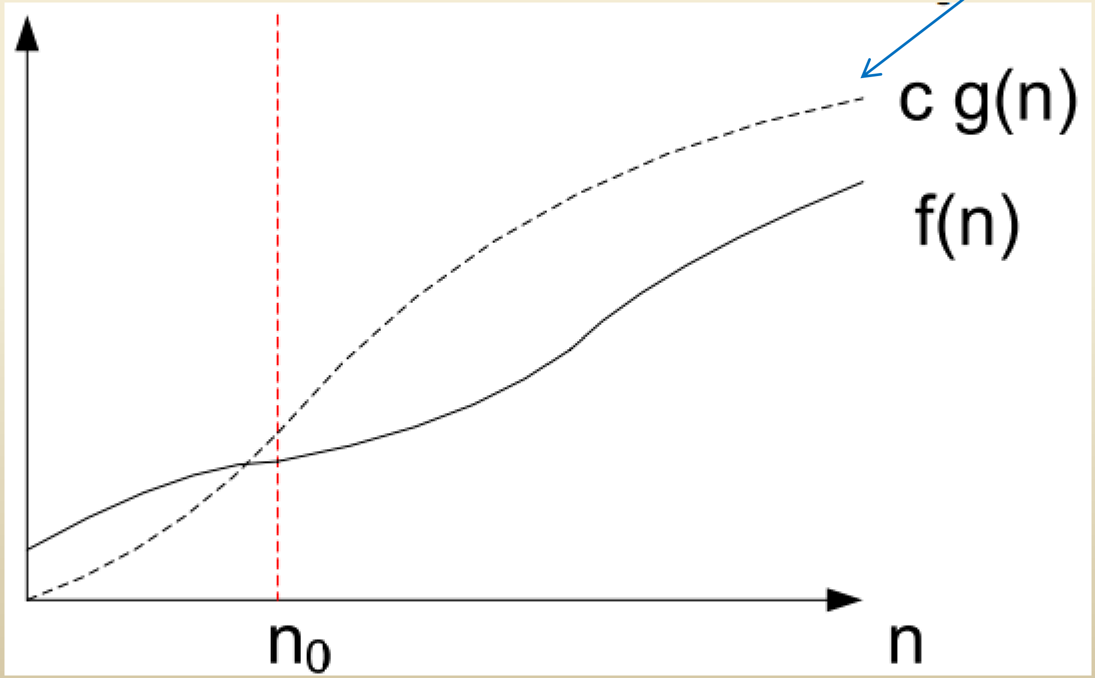
- $f(n)=O(g(n))$ iff

  $\exists$ a real constant $c>0$ and an integer constant $n_0 \geq 1$,

  $\ni f(n) \leq cg(n) \; \forall \; n, n \geq n_0$

- $cg(n)$ is a upper bound of $f(n)$

Upper bound of $f(n)$ when $n \geq n_0$



Asymptotic Notation – Big "oh"

- Examples
  - (1) $3n+2 = O(n)$

    $3n+2 \leq 4n$ for all $n \geq 2$
  - (2) $10n^2+4n+2=O(n^2)$

    $10n^2+4n+2 \leq 11n^2$ for all $n \geq 10$
  - (3) $3n+2 = O(n^2)$

    $3n+2 \leq n^2$ for all $n \geq 4$ ➔Not good enough, O(n) in example (1) is a lower upper bound than $O(n^2)$
    - (一個程式執行要9秒的時間，則最多可於10秒或100秒內跑完何者為較精確的說法?)
- g(n) should be a lowest upper bound
- A most-seen performance measure because we care about how fast an algorithm runs

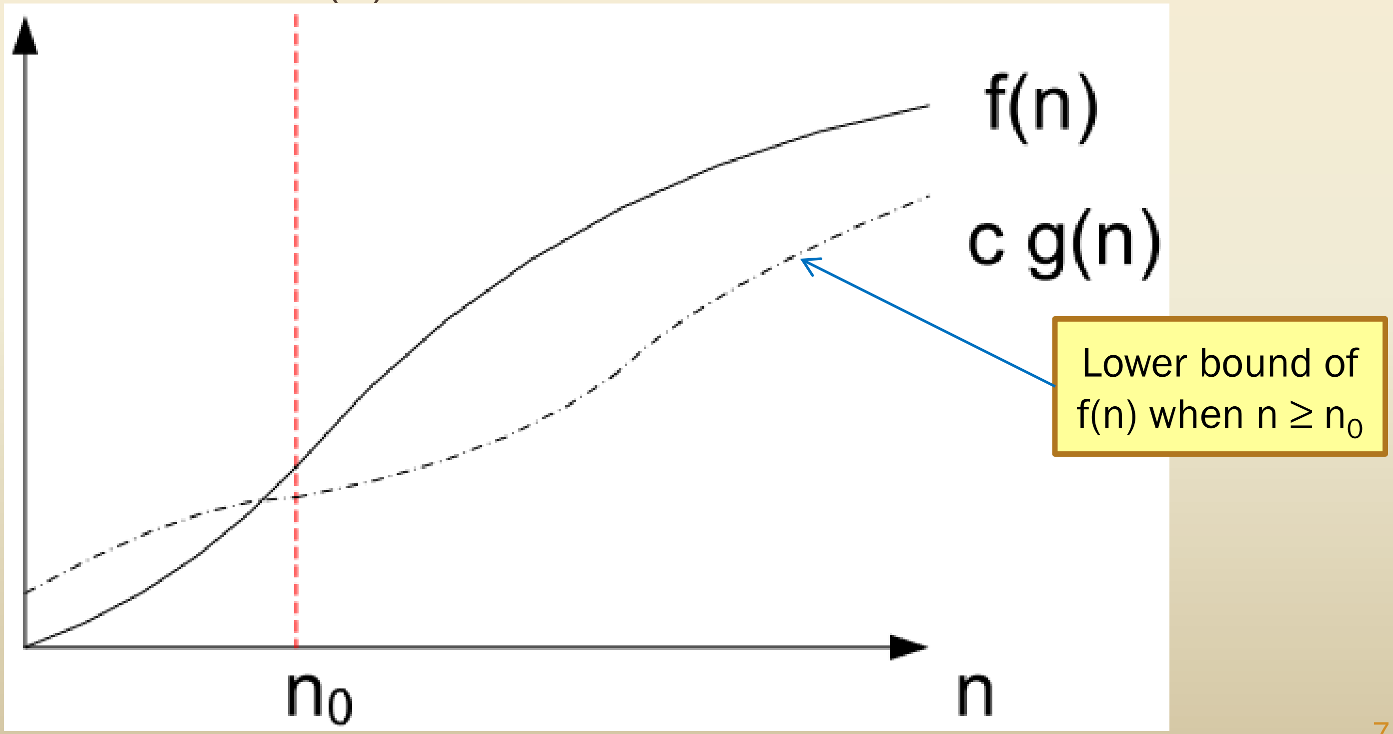
Asymptotic Notation – Omega

- $f(n)=\Omega(g(n))$ iff

  ∃ a real constant $c>0$ and an integer constant $n_0 \geq 1$,

  ∋ $f(n) \geq cg(n)$ ∀ n, $n \geq n_0$

- cg(n) is a lower bound of f(n)



Asymptotic Notation – Omega

- Examples
  - (1) $3n+3=\Omega(n)$

    $3n+3\geq 3n$ for all $n\geq 1$
  - (2) $6*2^n+n^2=\Omega(2^n)$

    $6*2^n+n^2\geq 2^n$ for all $n\geq 1$
  - (3) $3n+3=\Omega(1)$

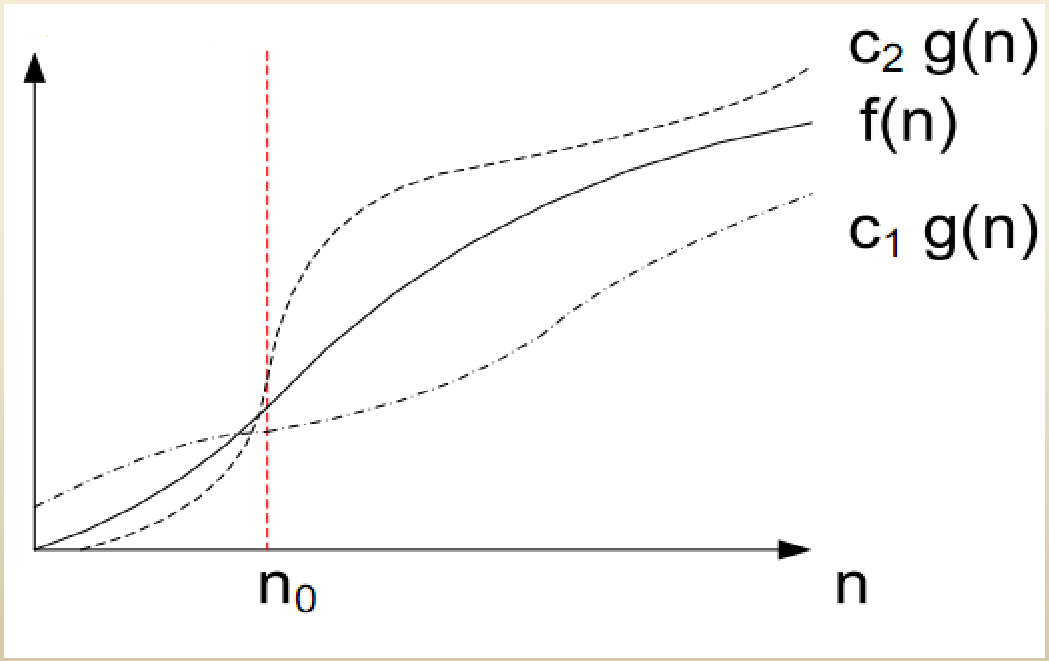
    $3n+3\geq 3$ for all $n\geq 1$ ➔Not good enough, $\Omega(n)$ in example (1) is a higher lower bound than $\Omega(1)$
  - (一個程式執行要101秒的時間，則最少需要10秒或100秒跑完何者為較精確的說法?)
- g(n) should be the highest lower bound

Asymptotic Notation – Theta

- $f(n)=\Theta(g(n))$ iff

  $\exists$ two positive real constants $c_1,c_2>0$, and an integer constant $n_0 \geq 1$, $\ni c_1 g(n) \leq f(n) \leq c_2 g(n)$ $\forall$ $n, n \geq n_0$

- Find both the upper and lower bound of f(n)



Asymptotic Notation – Theta

- Examples
  - $3n+2=\Theta(n)$

    $3n \le 3n+2 \le 4n$, for all $n \ge 2$
  - $10n^2+4n+2=\Theta(n^2)$

    $10n^2 \le 10n^2+4n+2 \le 11n^2$, for all $n \ge 5$

- Multiples of g(n) should be both lower bound & upper bound

Running Time Calculations

- For loop

```
for (i=0; i<n; i++)
{
     x++;
     y++;
     z++;
}
```

- Time complexity = $n \times 3 = O(n)$

Running Time Calculations

- Nested for loops

```
for (i=0; i <n; i++)
     for (j=0; j<n; j++)
          k++;
```

- Time complexity = $n \times n = O(n^2)$

Running Time Calculations (cont'd)

- Consecutive statements

```
for (i=0; i<n; i++)
        A[i]=0;
for (i=0; i<n; i++)
        for (j=0; j<n; j++)
                A[i]+=A[j]+i+j;
```

- Time complexity =

  $\max(1\times n, 1\times n\times n)$

  $= 1\times n\times n$

  $=O(n^2)$

Running Time Calculations (cont'd)

- If/Else

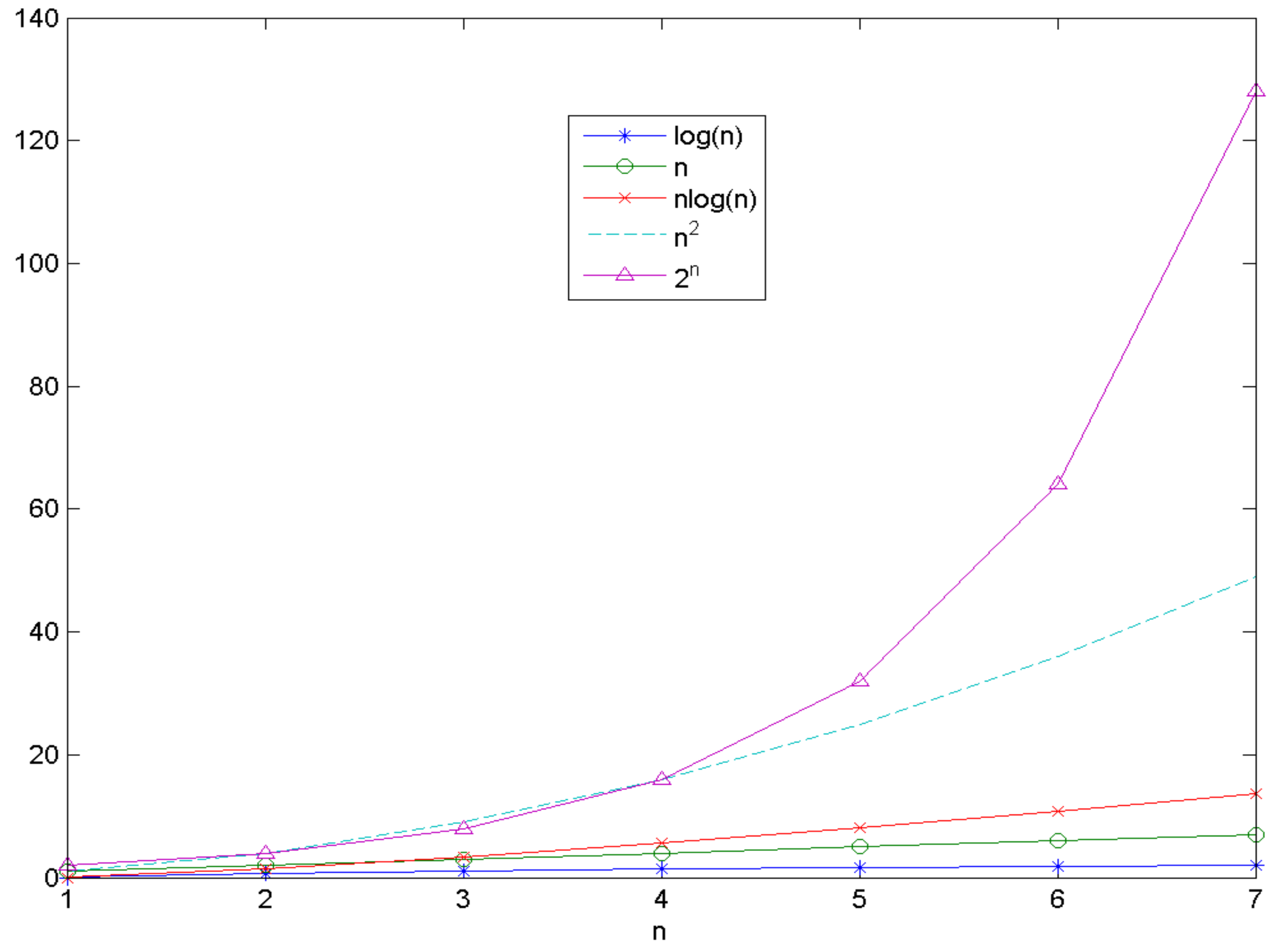
```
if (i>0) {
   i++;
   j++;
} else {
   for (j=0; j<n; j++)
     k++;
}
```

- Time complexity = max(2, 1×n) = n

Typical Growth Rate

c:	constant
$\log N$:	logarithmic
$\log^2 N$:	Log-squared
N:	Linear
$N\log N$	
N^2:	Quadratic
N^3:	Cubic
2^N:	Exponential

Growth Rate Comparison



Asymptotic Complexity of Insertion Sort

- For insertion sort => time complexity is $O(n^2)$
- What does this mean?
- Any way to measure how much time is needed practically for running insertion sort? We'll talk about this later.

Complexity of Insertion Sort

- Time or number of operations does not exceed **c.n**^{2} on any input of size **n** (**n** suitably large).
- Actually, as you shall see later in the course, for insertion sort
  - the worst-case time is $\Theta(\mathbf{n}^2)$ and
  - the best-case is $\Theta(\mathbf{n})$
- So, the worst-case time is expected to quadruple each time **n** is doubled.

Complexity of Insertion Sort

- Is $O(n^2)$ too much time?
- Is the algorithm practical?

Practical Complexities

- If 10^9 instructions/second, then

n	***n***	***nlogn***	***n^2***	***n^3***
1000	10^{-6} sec	10^{-5} sec	10^{-3} sec	1sec
10000	10^{-5} sec	$1.3*10^{-4}$ sec	0.1 sec	17min
10^6	10^{-6} sec	$2*10^{-5}$ sec	17min	32years

- As of 2017, it is reported that near 100 PFLOPS (petaFLOPS, 10^{15}次的浮點運算 (Floating-point operations --FLOPS)) was accomplished.

Faster Computer V.S. Better Algorithm

- Algorithmic improvement more useful than hardware improvement
  - E.g. 2^n to n^3

Performance Measurement

- Measure actual running time on an actual computer

- What do we need?

Performance Measurement Needs

- Programming language
- Working program
- Computer
- Compiler and options to use
- Test data to use for measurement
  - worst-case data
  - best-case data
  - average-case data
- Timing mechanism — clock

Timing In C++

- Timing library in time.h
- This header file contains definitions of functions to get and manipulate date and time information
  - E.g., time, difftime, clock…

Timing In C++

```
long start, stop;
time(start); // set start to current time
// code to be timed comes here
time(stop); // set stop to current time
long runTime = stop – start;
```

Accurate Timing

- Some executions may be so short that timing functions cannot capture their correct amount of computing time.
  - So make sure your timing is large enough to cover the execution time of the program to be tested
  - Refer to Program 1.29 in the textbook for an example → Self-study

Accurate Timing

- Alternative solution:
  - Repeat testing work many times, so you can take average of all the tests to gain statistical significance

Resources

- Data structure visualization
  - http://www.cs.usfca.edu/~galles/visualization/about.html

Homework Assignment

- Grab homework #1 on e-learning system
- Only need to hand out your paperwork
- Pay attention to the due time