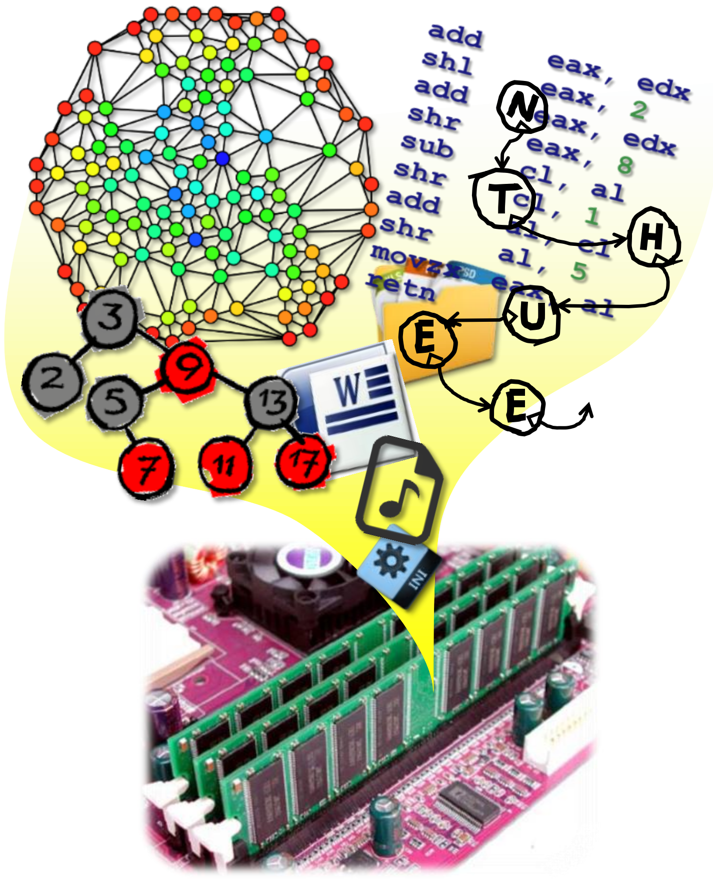
# Data Structures

## CH1 Basic Concepts




Prof. Ren-Shuo Liu

NTHU EE

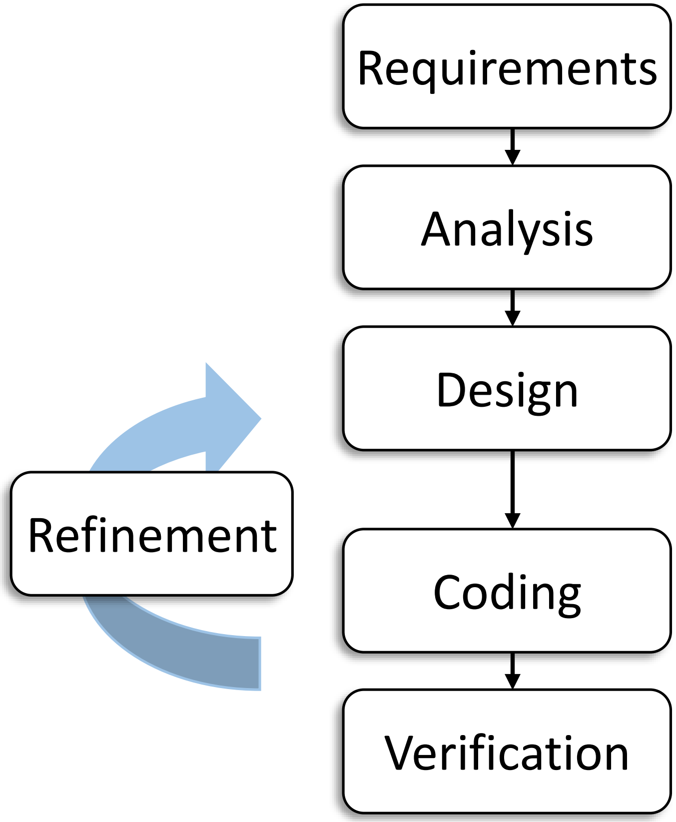
Spring 2018

# Outline

- **1.1 Overview: System Life Cycle**
- 1.2 Object-Oriented Design
- 1.3 Data Abstraction and Encapsulation
- (1.4 Basics of C++)
- 1.5 Algorithm Specification
- (1.6 Standard Template Library)
- 1.7 Performance Analysis and Measurement

# System Life Cycle

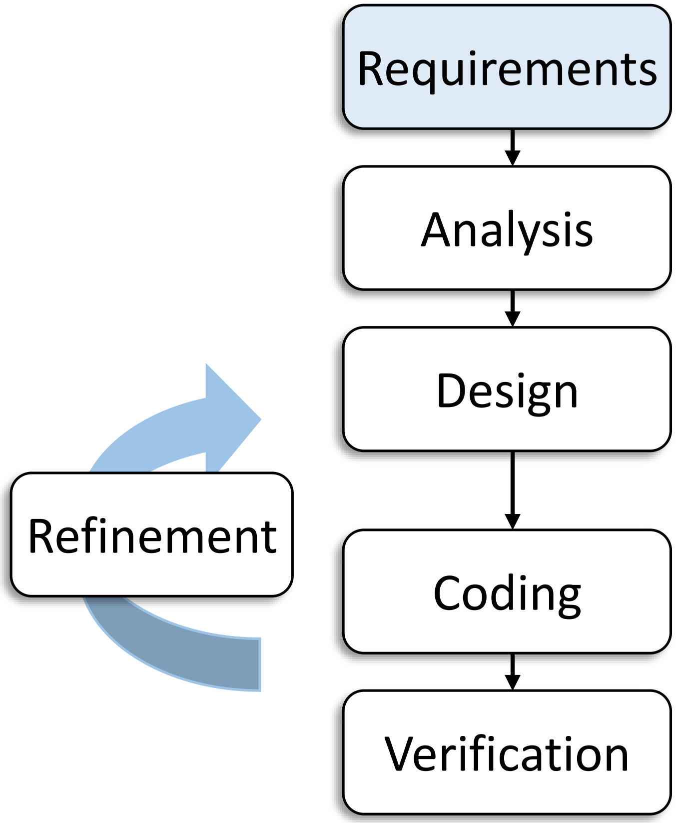
- Five phases
  1. Requirements
  2. Analysis
  3. Design
  4. Refinement and coding
  5. Verification



# Requirements

- Clarify problem specifications
  - Input
    - What are given
  - Output
    - What must be produced

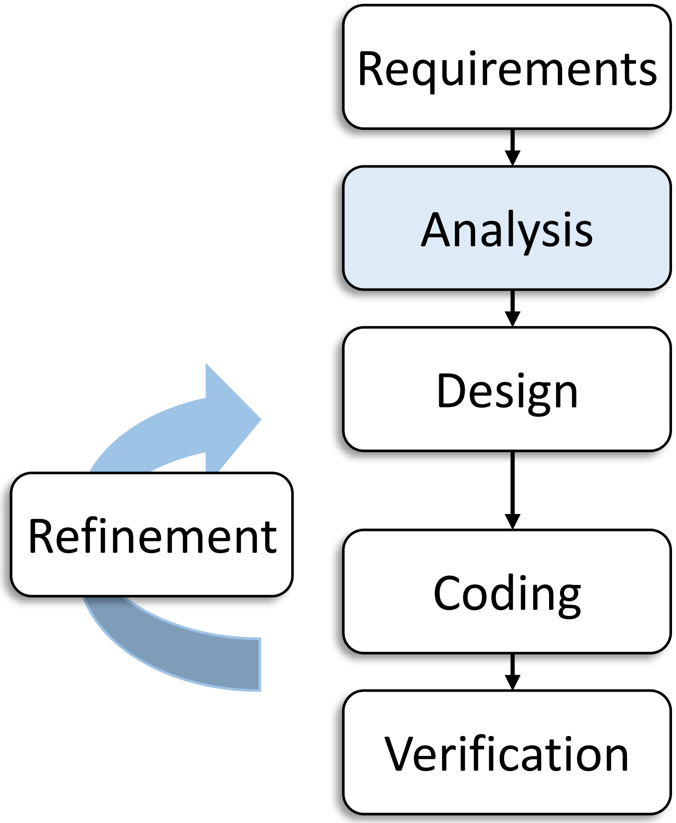
- Initially vague → more precise



# Analysis

- Break down the problem
  - Into manageable pieces
  - Also known as divide and conquer

- Two approaches
  1. Bottom-up (not good)
  2. Top-down (better)



# Bottom-up Analysis

- Issues
  - Too early emphasis on low-level details
  - Lack of prior planning and a big picture

- Risks and difficulties
  - →Resulting system can have many loosely connected and error-ridden segments ☹
  - →Unpractical for tackling large-scale, complex problem

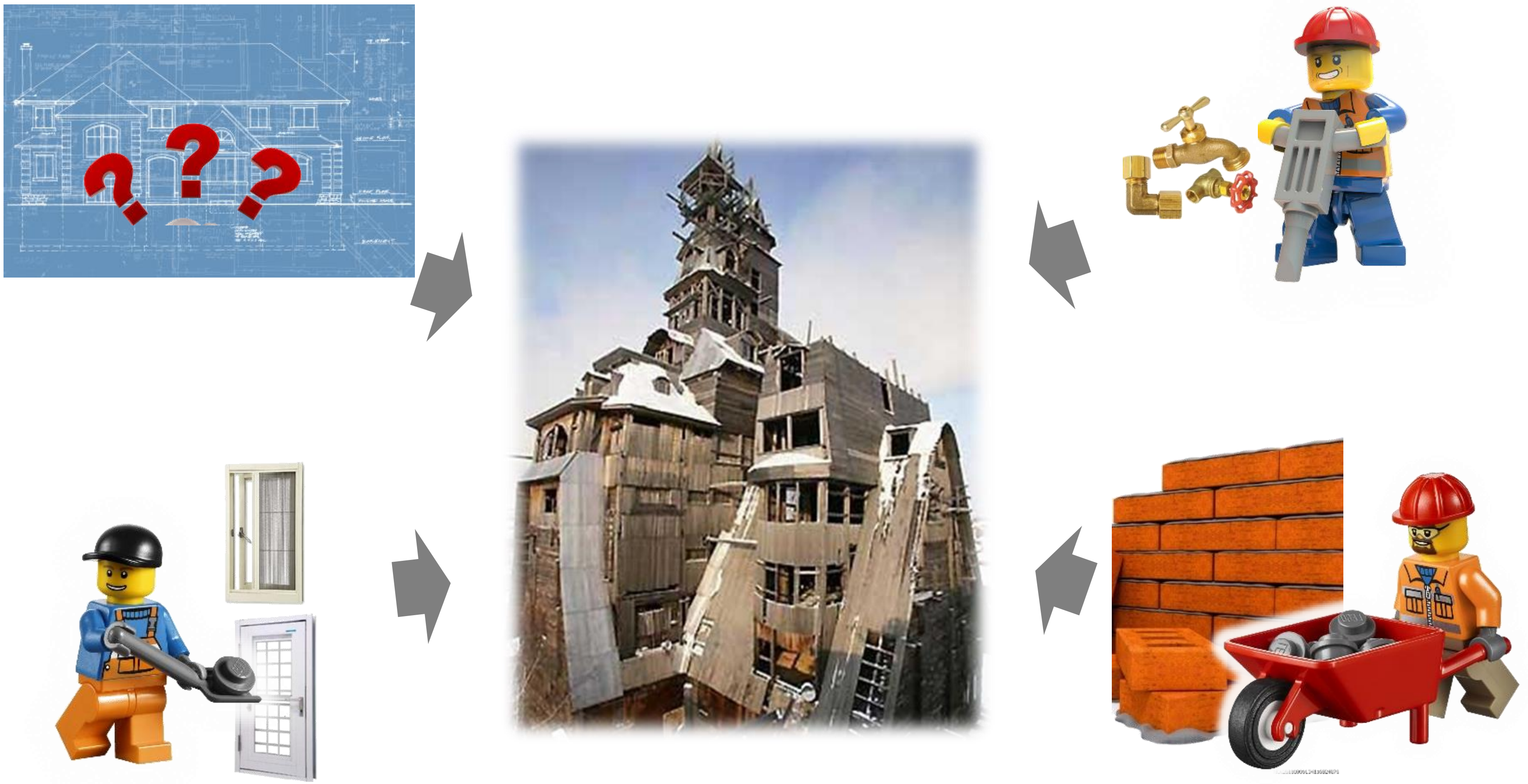
# Top-down Analysis

- Strategies
  - Start from a high-level plan
    - Breaking a problem down into manageable pieces
  - Subsequently refining the plan
    - Gradually taking into account low-level details

- Advantages

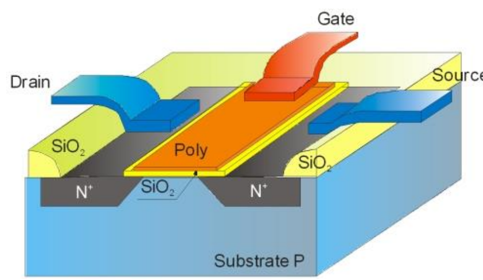
  →Necessary for tackling large-scale, complex problem

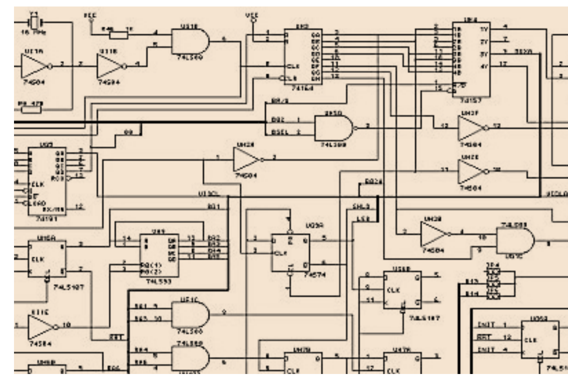
# Risks of Bottom-Up

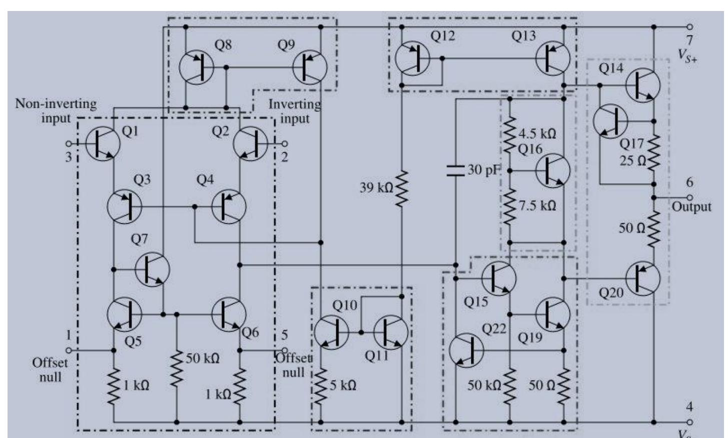


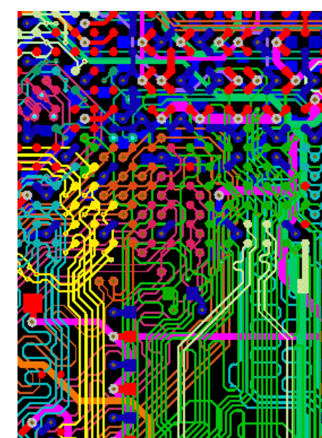
# Difficulties of Bottom-Up

- Please imagine analyzing a smartphone bottom-up
  - Things become complicated



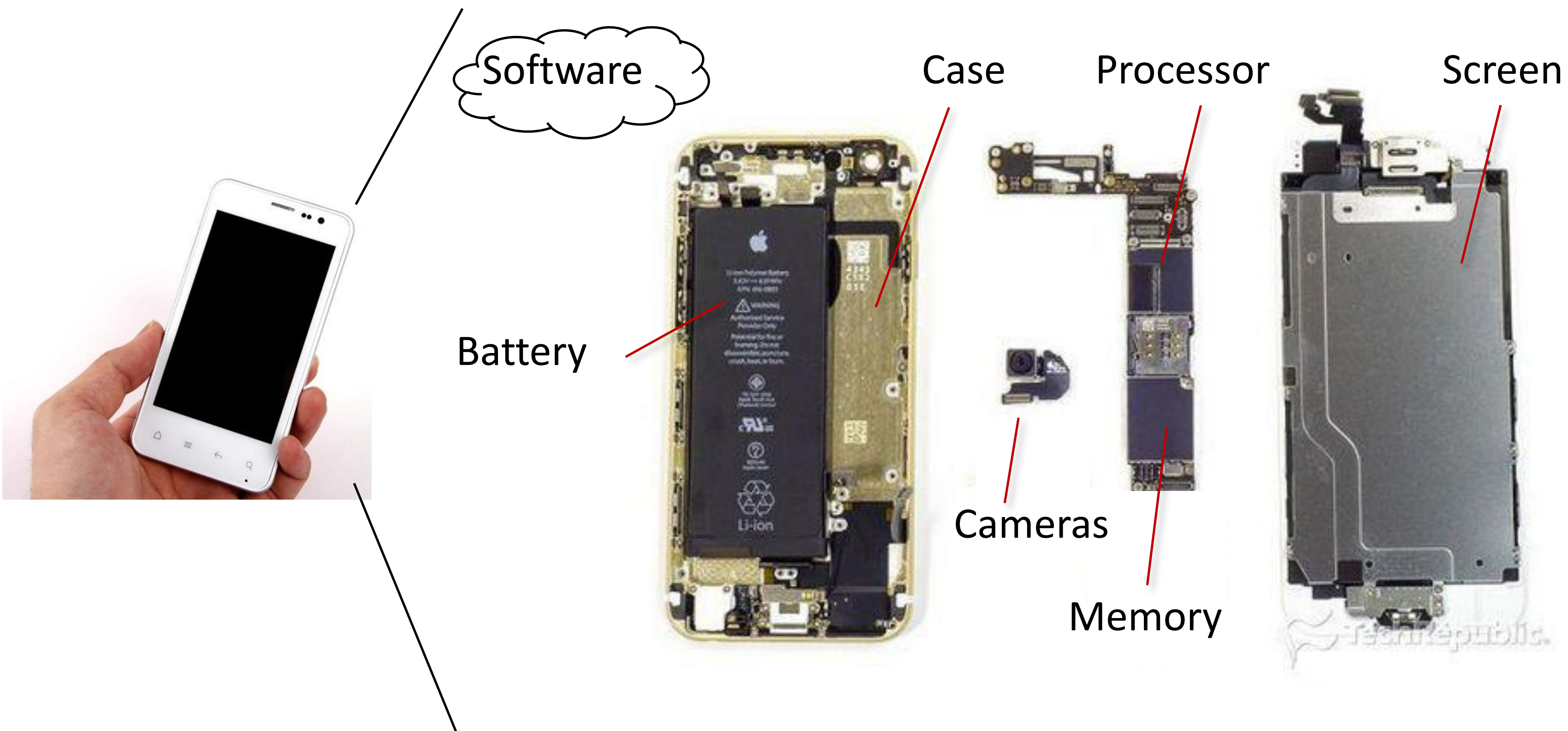





# Benefits of Top-Down

- Now let's alternatively analyze a smartphone top-down

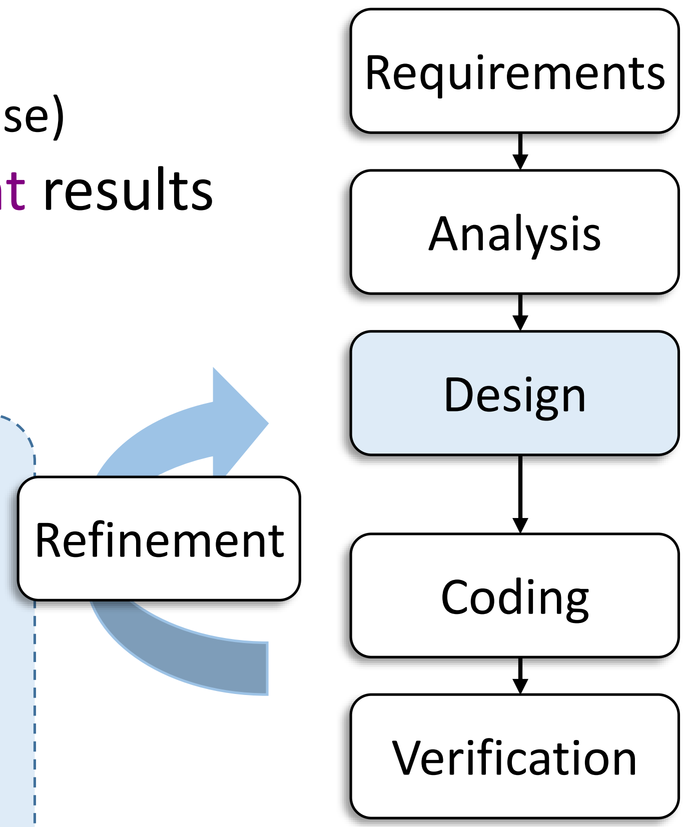


# Design

- Identify
  - Data objects
  - Operations performed on the data types
  - ~~Implementation~~ (Not decided in this phase)
- Produce implementation-independent results
  - Abstract data types
  - Algorithm specifications

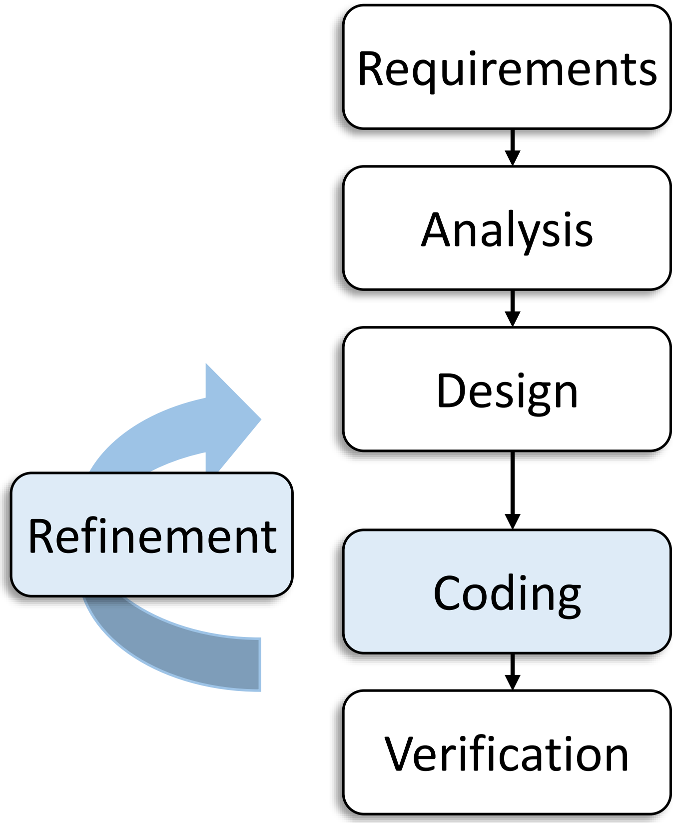
**Scheduling system for NTHU**

- Data objects
  - Students
    - Name, ID, major, and phone #
  - Courses
  - Professors
- Operations
  - Inserting, removing, and searching



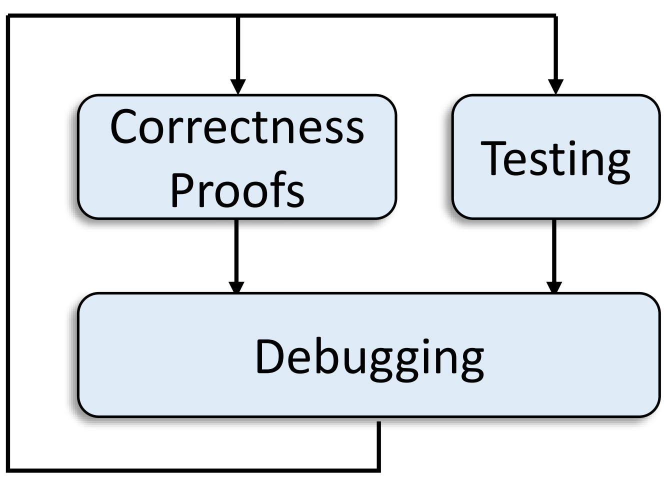
# Coding and Refinement

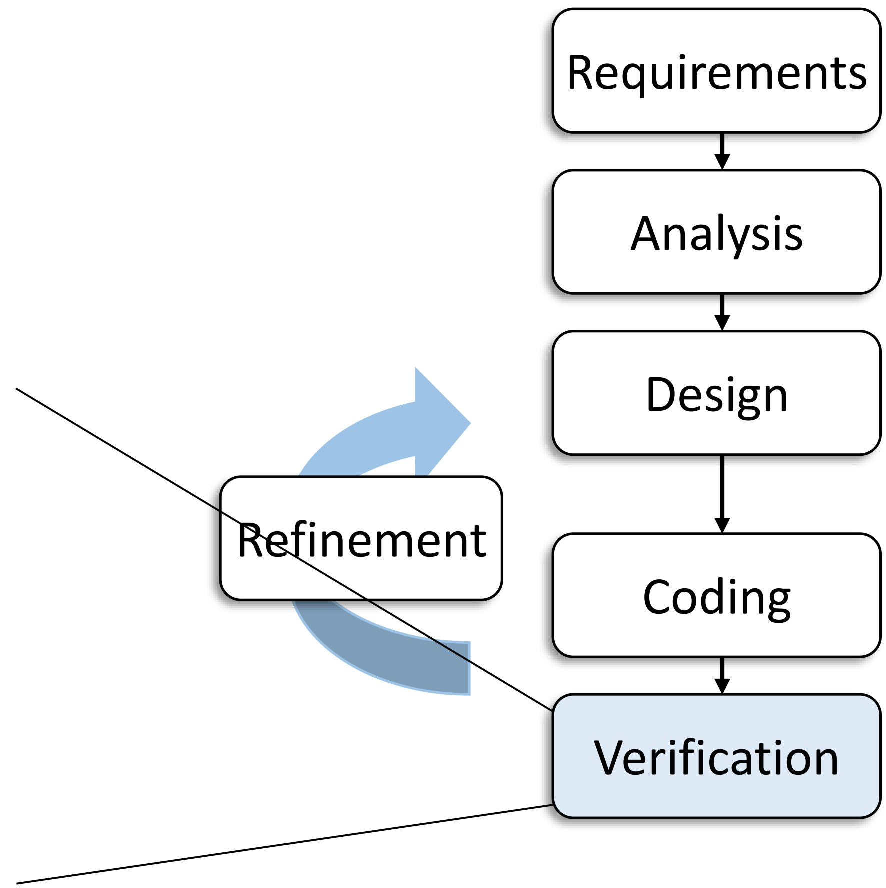
- Decide implementation
  - Representations for objects
  - Algorithms for operations
- Algorithm and object representations affect the efficiency of each other
  - Design the algorithms that are independent of data objects first
- Good design can absorb changes found in this stage easily



# Verification

- Three techniques
    1. Correctness proofs
    2. Testing
    3. Debugging



# Verification (Cont'd)

- Correctness proofs
  - Formal method
  - Typically required for individual algorithm
  - Not easily achievable for the whole program

# Verification (Cont'd)

- Testing
  - Run a program against possible inputs
    - Check correctness
    - Check performance (e.g., execution time)
  - Coverage – a metric for assessing the completeness of testing
    - Testing inputs should be developed to cover as many percentages of codes as possible
      - E.g., all the cases within a switch statement should at least be touched
- Debugging
  - Removal of errors found
  - Well-documented and well-structured program eases debugging

# Outline

- 1.1 Overview: System Life Cycle
- **1.2 Object-Oriented Design**
- 1.3 Data Abstraction and Encapsulation
- (1.4 Basics of C++)
- 1.5 Algorithm Specification
- (1.6 Standard Template Library)
- 1.7 Performance Analysis and Measurement

# Programming Paradigms

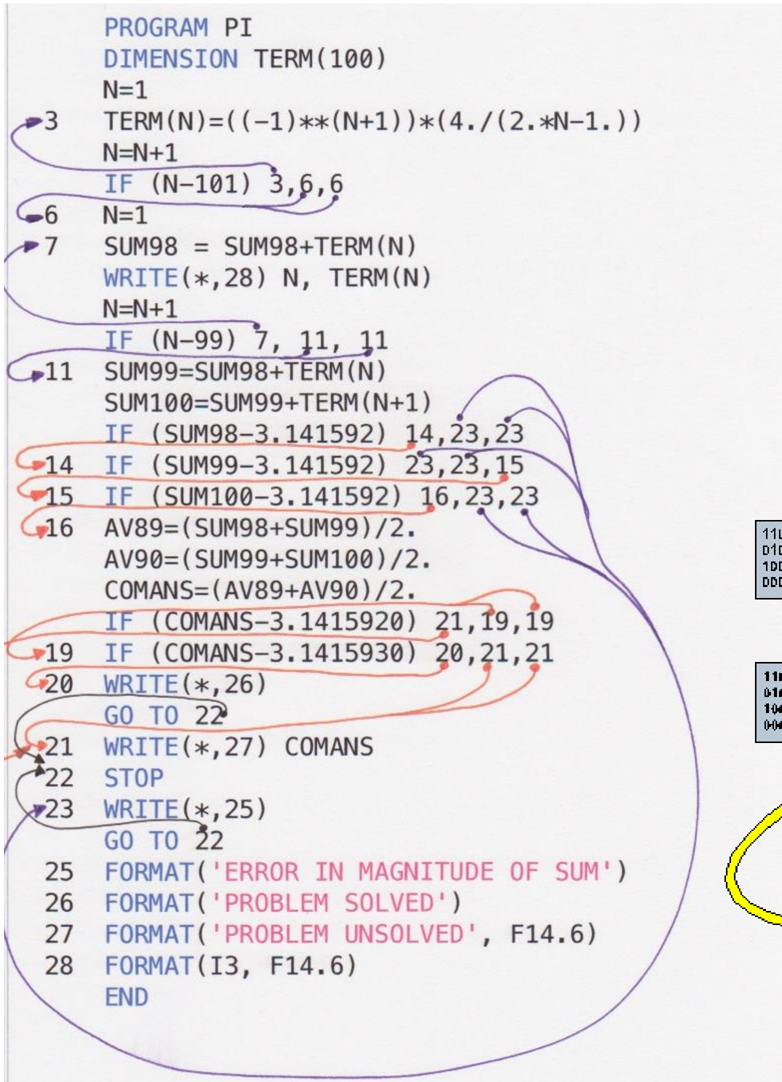
- Non-structured
- Structured
- Object-oriented

More disciplines are imposed on programmers

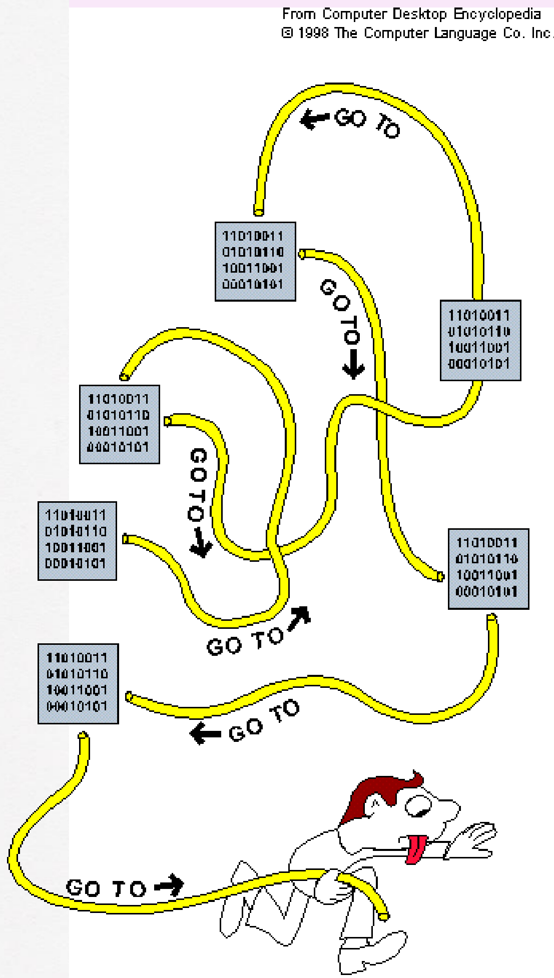
# Non-Structured Programming

- Characteristics
  - Sequentially ordered commands
  - Lines are numbered or labeled
  - Unrestricted jump/branch to any line
- Pros
  - Extremely skillful programmers can find tricky methods to produce high performance or compact code
- Cons
  - Encourage spaghetti codes
  - Poor maintainability
  - Difficult in building large programs (poor scalability)

# Spaghetti Code



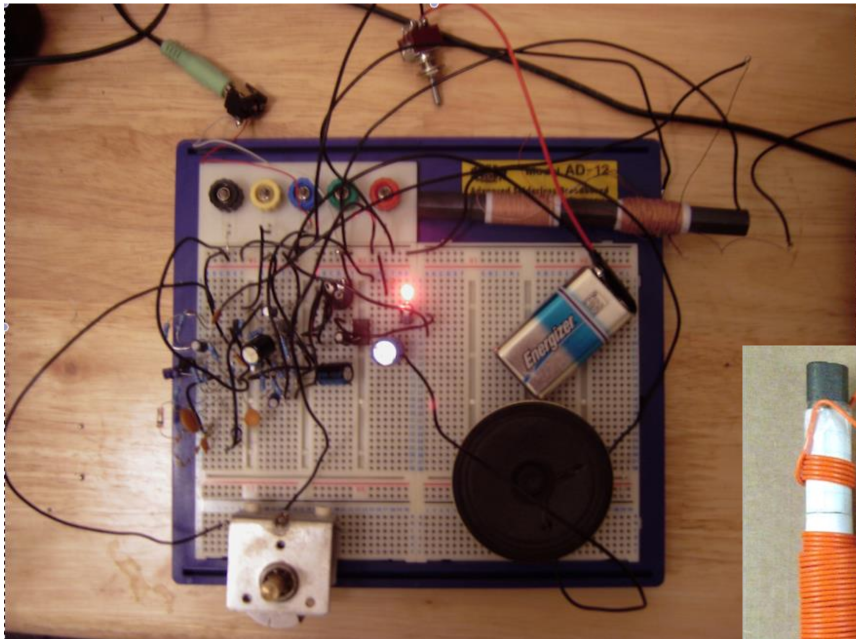
```
      PROGRAM PI
      DIMENSION TERM(100)
      N=1
 3    TERM(N)=((-1)**(N+1))*(4./(2.*N-1.))
      N=N+1
      IF (N-101) 3,6,6
 6    N=1
 7    SUM98 = SUM98+TERM(N)
      WRITE(*,28) N, TERM(N)
      N=N+1
      IF (N-99) 7, 11, 11
 11   SUM99=SUM98+TERM(N)
      SUM100=SUM99+TERM(N+1)
      IF (SUM98-3.141592) 14,23,23
 14   IF (SUM99-3.141592) 23,23,15
 15   IF (SUM100-3.141592) 16,23,23
 16   AV89=(SUM98+SUM99)/2.
      AV90=(SUM99+SUM100)/2.
      COMANS=(AV89+AV90)/2.
      IF (COMANS-3.1415920) 21,19,19
 19   IF (COMANS-3.1415930) 20,21,21
 20   WRITE(*,26)
      GO TO 22
 21   WRITE(*,27) COMANS
 22   STOP
 23   WRITE(*,25)
      GO TO 22
 25   FORMAT('ERROR IN MAGNITUDE OF SUM')
 26   FORMAT('PROBLEM SOLVED')
 27   FORMAT('PROBLEM UNSOLVED', F14.6)
 28   FORMAT(I3, F14.6)
      END
```




**FORTRAN's three-way arithmetic IF**
Jump to one of three locations in the program depending on the whether expression was negative, zero, or positive.

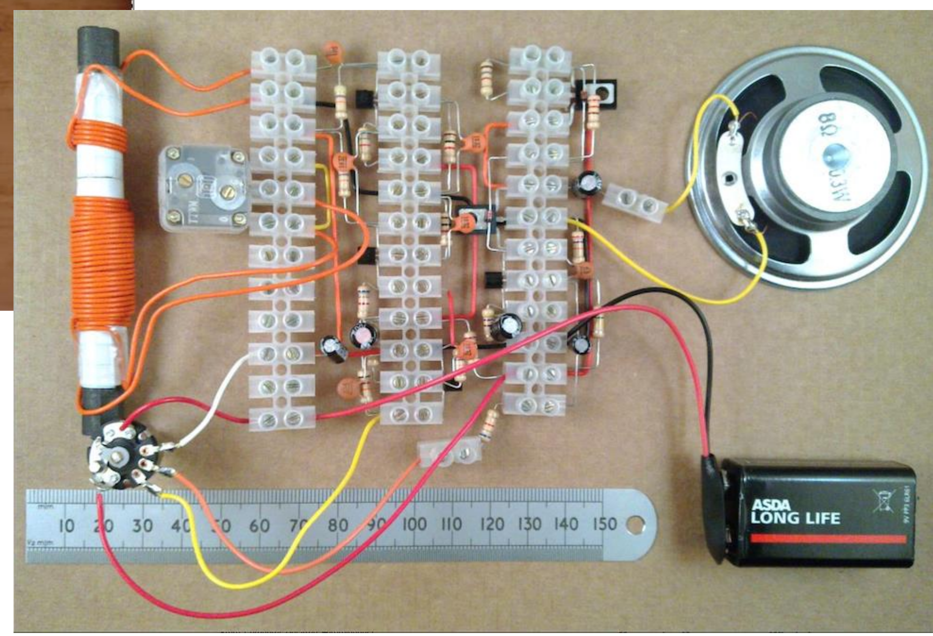
https://craftofcoding.wordpress.com/2013/10/07/what-is-spaghetti-code/
http://www.quora.com/What-does-spaghetti-code-actually-look-like

# Spaghetti Circuit




← Spaghetti circuit

↓ Clean circuit

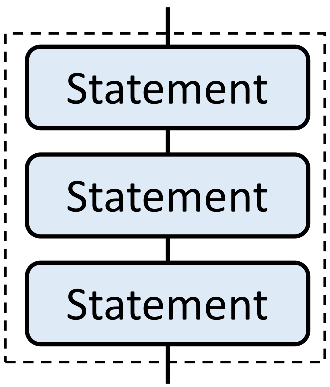



What do you think the possible function of these circuits is?

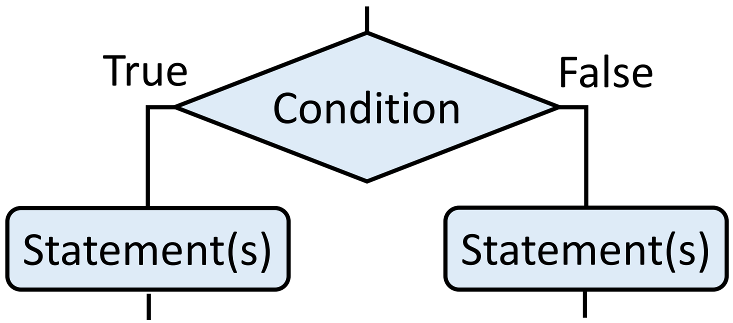
http://www.quora.com/What-does-spaghetti-code-actually-look-like

# Structured Programming

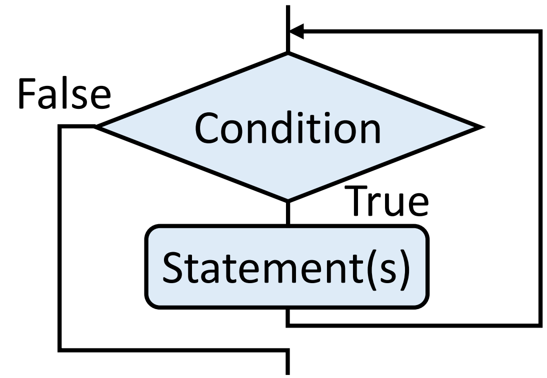
- Basic structures




**Sequence**




**Selection (or choice)**
If(condition) {…} else {…}




**Repetition (or looping)**
While(condition) {…}

- All programs can be equivalently transformed to that use **only** the above three structures

# Structured Programming (Cont'd)

- Pros
  - Easy to understand
  - Easy to maintain
  - Easy to analyze
- Pure structured languages strictly disallow C/C++'s
  - *goto*
  - *break*
  - *continue*

# Structured Programming (Cont’d)

- Compared with non-structured programming
  - Structured programming restricts programmers' freedom
  - Structured programming prevents spaghetti codes
  - Structured programming does not change programmability
    - What problem non-structured programming can solve can also be done using structured programming (and vice versa)

# Structured Programming (Cont'd)

- C and C++ are structured languages but NOT pure ones
  - *goto, break, continue* statements are allowed
- *goto* statement is notorious but not always bad
  - See the example on the right

```
for(x=0; x<1000; x++){
    for(y=0; y<1000; y++){
        for(z=0; z<1000; z++){
            if( g(x, y, z) > 0 ){
                cout << x << ", "
                     << y << ", " << z;
                goto END;
            }
        }
    }
}
END:
```

Code snippet for searching an integer solution of g(x, y, z)>0 in a brute force way. In this example, it is convenient to use goto to leave the nested loops.

# Object-Oriented Programming

- Philosophy of divide-and-conquer is the same as structured programming
- How a project should be decomposed is changed

- Decomposition methods
  1. Algorithmic (functional) decomposition is used for the structured programming method
  2. Object-oriented decomposition is used for the object-oriented programming method

# Algorithmic/Functional Decomposition

- Used by structured programming
- View software as a process
- Decompose software into modules that represent steps of the process
  - In C, the modules are functions

- Compute-centric perspective
- Data structures are a secondary concern

# Object-Oriented (OO) Decomposition

- Used by object-oriented programming
- View software as a set of well-defined objects
  - Objects model entities in the application domain
    - e.g., students, courses, and teachers in a course scheduling system
  - Objects interact with one another
- Algorithmic or functional decomposition is addressed after the system has been decomposed into objects

# OO Decomposition (cont'd)

- Pros
  - Encourage the reuse of software
  - Software becomes more flexible that can evolve as requirements change
  - More intuitive because objects naturally model entities in the application domain

# Definitions

- Object
  - Entity that has a local state and performs computations
    - i.e., a combination of data and operations
- Object-oriented programming
  - Method of implementation in which …
    - Objects are the fundamental building blocks
    - Each object is an instance of some type (or class)
    - Classes are related to each other by inheritance relationships

# Definitions

- A language is said to be an object-oriented language if
  - It supports objects
  - It requires objects to belong to a class
  - It support inheritance

- A language is said to be merely an object-based language if it supports the first two features but does not support inheritance

# Evolution of Programming

- Four generations of higher level languages
  - FORTRAN, etc.
    - Salient feature of evaluating mathematical expression
  - C, Pascal, etc.
    - Emphasis on effectively expressing algorithm
  - Modula, etc.
    - Introduce of the concept of abstract data types (ADT)
  - Smalltalk, Objective C, C++, etc.
    - Emphasis on inheritance between ADTs

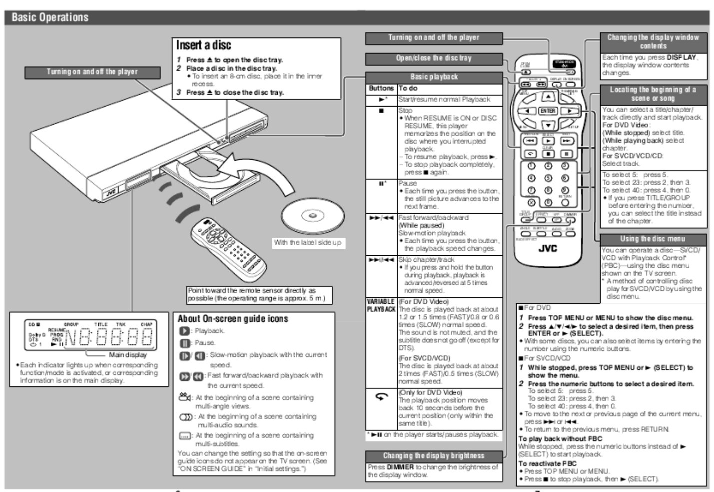
# Outline

- 1.1 Overview: System Life Cycle
- 1.2 Object-Oriented Design
- **1.3 Data Abstraction and Encapsulation**
- (1.4 Basics of C++)
- 1.5 Algorithm Specification
- (1.6 Standard Template Library)
- 1.7 Performance Analysis and Measurement

# Definition

- Data Encapsulation (or Information Hiding) (封裝)
  - Conceal the implementation details of a data object form the outside world
- Data Abstraction (抽象化)
  - Separation between the specification of a data object and its implementation

# DVD Player Analogy


- Encapsulation —the buttons and remote control
  - The only interfaces exposed to users
  - Hide and protect internal (vulnerable, dangerous, and proprietary) design from users
- Abstraction — the user manual
  - Only specify what the function of each button is
  - How the player achieve the function is not mentioned nor restricted

# Definition

- Data Type
  - objects
    +
    operations on the objects

- Abstract Data Type (ADT)
  - Object — Specification | Representation
  - Operation — Specification | Implementation

# Data Types in C++

- Predefined (built-in) types
  - Fundamental types
    - *char*
    - *int*
    - *float*
    - *double*
  - Modifiers
    - *short*
    - *long*
    - *signed*
    - *unsigned*
  - Derived types
    - Pointer (*)
    - Reference (&)
  - Aggregate types
    - Arrays
    - *struct*
    - *class*
- User-defined types
  - *struct*
  - *class*

# ADT Example: *NaturalNumber*

**ADT** *NaturalNumber* is

**objects**:

An ordered subrange of the integers starting at zero and ending at MAXINT on the computer.

**functions**:

for all x, y ∈ *NaturalNumber*; **true**, **false** ∈ *Boolean*
and where +, -, <, ==, = are the usual integer operations

| | | |
|---|---|---|
| *Zero* (): *NaturalNumber* | ::= | 0 |
| *IsZero* (x): *Boolean* | ::= | **if** (x == 0) *IsZero* = **true**<br>else *IsZero* = **false** |
| *Add* (x, y): *NaturalNumber* | ::= | **if** (x+y <= MAXINT) *Add* = x + y<br>else *Add* = MAXINT |
| *Equal* (x, y): *Boolean* | ::= | **if** (x == y) *Equal* = **true**<br>else *Equal* = **false** |
| *Successor* (x): *NaturalNumber* | ::= | **if** (x == MAXINT) *Successor* = x<br>else *Successor* = x +1 |
| *Substract* (x, y): *NaturalNumber* | ::= | **if** (x < y) *Substract* = 0<br>else *Substract* = x – y |

**end** *NaturalNumber*

# ADT Example: *NaturalNumber*

**objects**:

An ordered subrange of the integers starting at zero and ending at MAXINT on the computer.

**functions specification**:

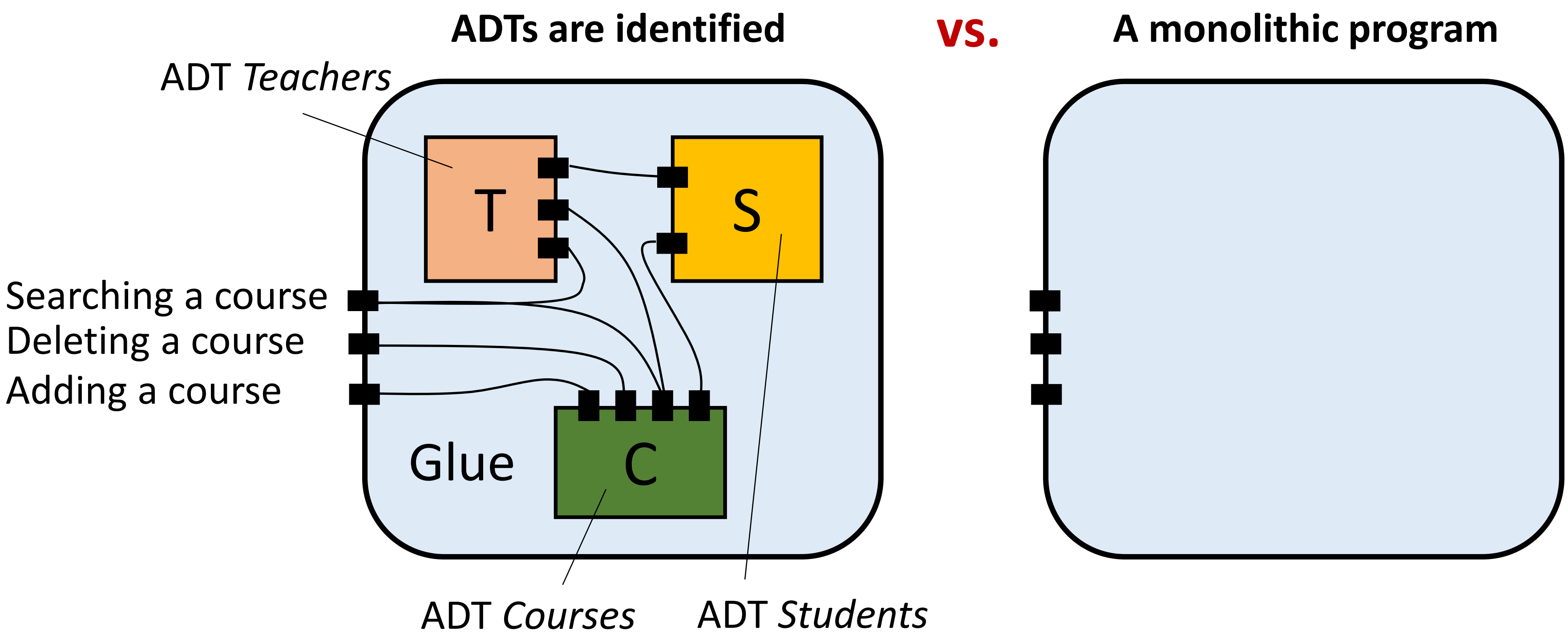
| Format | Return Type | Behavior |
|---|---|---|
| *Zero* () | *NaturalNumber* | 0 |
| *IsZero* (x) | *Boolean* | ***if*** (x == 0)<br>***return true***<br>***else***<br>***return false*** |
| *Add* (x, y) | *NaturalNumber* | ***if*** (x+y <= MAXINT) ***return*** x + y<br>***else return*** MAXINT |
| *Equal* (x, y) | *Boolean* | **if** (x == y) ***return true***<br>***else return false*** |
| *Successor* (x) | *NaturalNumber* | **if** (x == MAXINT) ***return*** x<br>***else return*** (x+1) |
| *Substract* (x, y) | *NaturalNumber* | ***if*** (x < y) ***return*** 0<br>***else return*** (x-y) |

# Advantages of Encapsulation and Abstraction

1. Simplify software development
2. Ease testing and debugging
3. Enable reusability
4. Support modifications to the representation of a data type

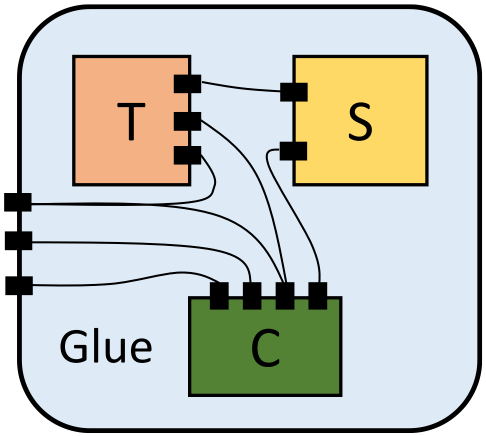
# Comparing Two Scenarios

- Consider developing a course scheduling program for NTHU
  - One can either adopt ADTs or directly dive into coding



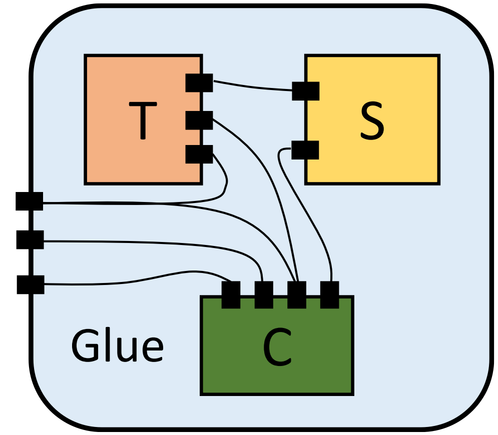
# Simplify Software Development

- With encapsulation and abstraction
  - If we have four programmers
    - They can parallelly work on A, B, C, and Glue
    - No one need to know how another one implement their portion of code
    - More concentration and less interference (especially when the project is large)
  - If we have only one programmer
    - Focus on A, B, C, and Glue one at a time
    - Less things need to be kept in mind



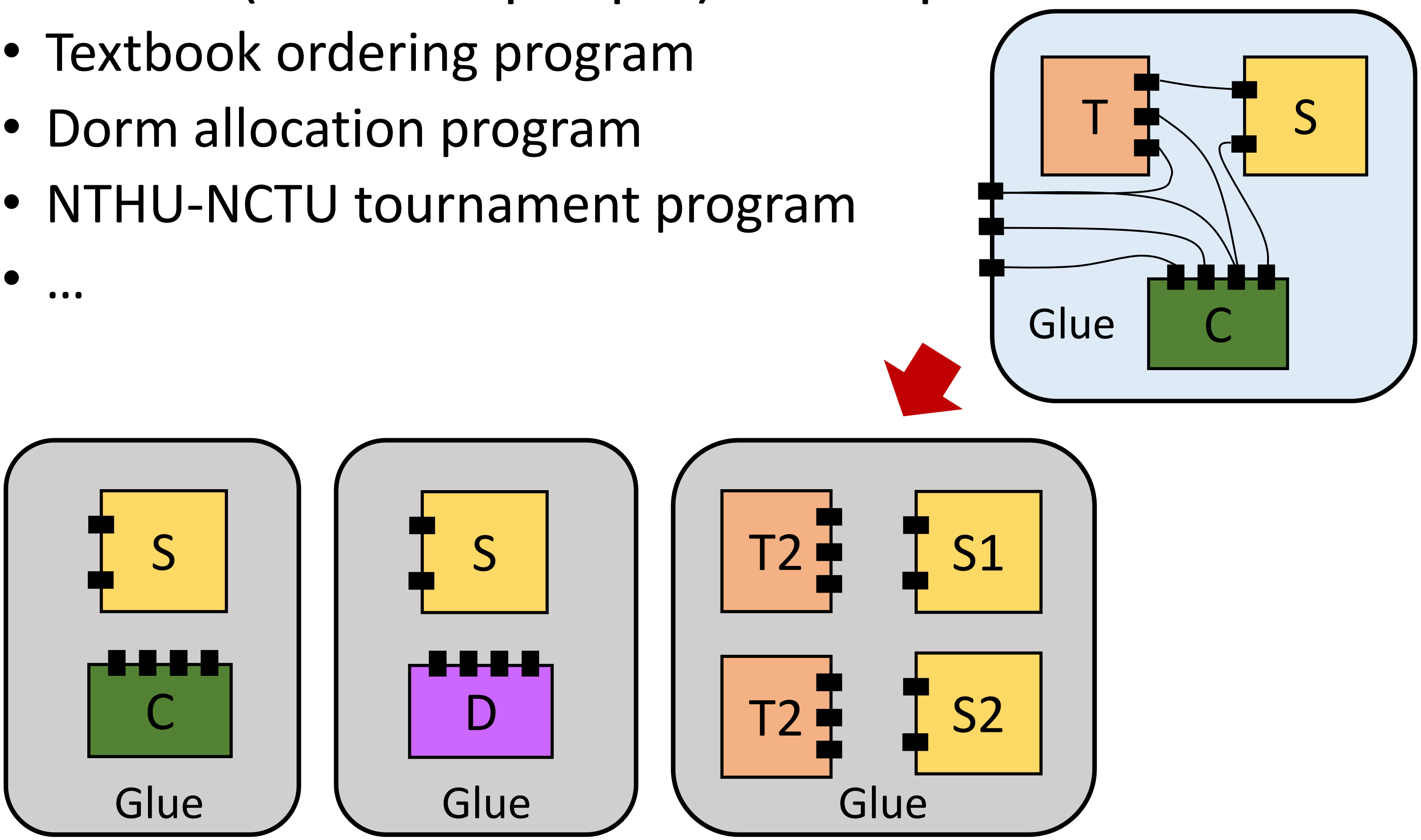
# Testing and Debugging

- With encapsulation and abstraction
  - A, B, C, and Glue can be individually tested and debugged
    - Testing efforts are $T(A) + T(B) + T(C) + T(Glue) \leq T(A+B+C+Glue)$
  - Assume we are confident that some portions, say A, B, and C, are clear, but a bug still exists…
    - → The remainder, say Glue, has the bug
  - Assume we notice the bug is related to a specific operation on a data type, say mistakenly deleting a course…
    - → The bug resides in the corresponding objects and operations



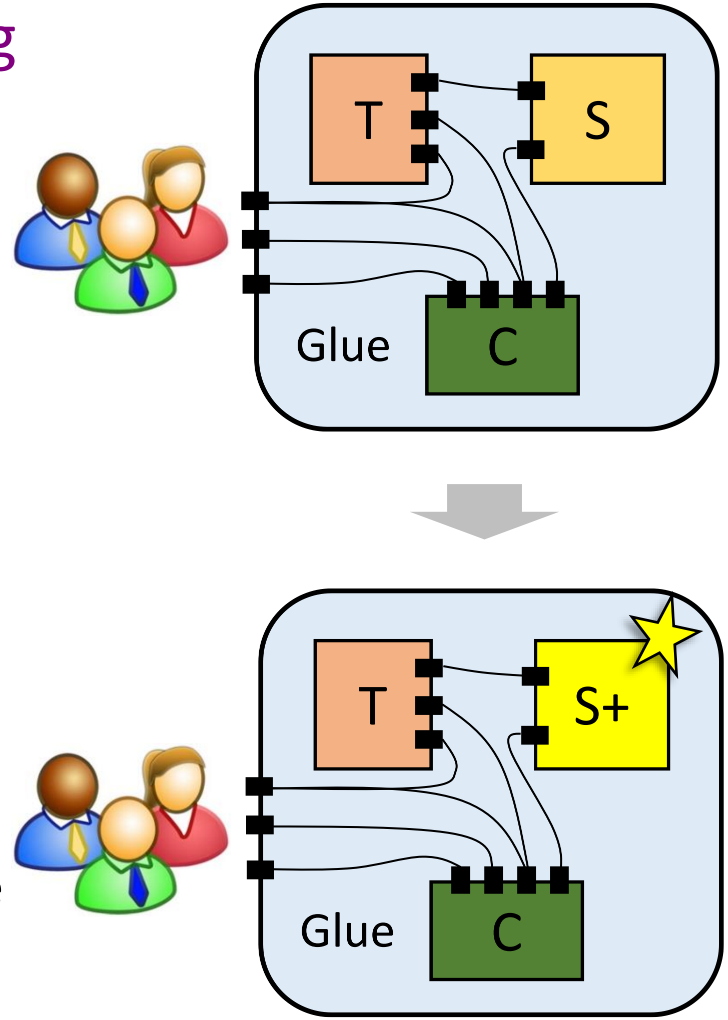
# Reusability

- When we (or other people) develop
  - Textbook ordering program
  - Dorm allocation program
  - NTHU-NCTU tournament program
  - …



# Modifications

- ADTs lead to information hiding
  - Implementation of a data type is invisible to users and the rest of the program
  - Ease changing (e.g., upgrade) a data type without rewriting the entire program or affecting any users
  - Allow us to start from a quick implementation then progressively refine the program
  - Even if we need to modify the interface of a data type
    - We can systematically identify the required modifications to the other parts



# Overhead of Adopting ADT

- Execution time overhead
  - Accessing data through interfacing operations is potentially slower than directly accessing them
- Memory space overhead
  - Every object maintains a table specifying its operations
- Coding is more tedious

- Therefore, C (not C++) is still widely used for programming the following things
  - Operating systems
  - Performance sensitive systems
  - Resource constrained systems

# Outline

- 1.1 Overview: System Life Cycle
- 1.2 Object-Oriented Design
- 1.3 Data Abstraction and Encapsulation
- (1.4 Basics of C++)
- **1.5 Algorithm Specification**
- (1.6 Standard Template Library)
- 1.7 Performance Analysis and Measurement

# Algorithm

- Criteria of an algorithm
- Exampling algorithms
  - Selection sort
  - Binary search
- Recursion
  - Selection sort
  - Binary search
  - Permutation

# Algorithm (Definition)

- A finite set of instructions
  - Input
    - Read zero or more quantities
  - Output
    - Produce one or more quantities
  - Correctness
    - Accomplishes a particular task for all possible inputs
  - Definiteness
    - Each instruction is unambiguous
  - Effectiveness
    - Each instruction is basic enough
  - Finiteness
    - Terminates after a finite number steps for all possible inputs

# Algorithms vs. Programs

- (From computational theorists’ perspective)
- Unlike an algorithm, a program needs not always satisfy “finiteness”
  - Kernel of an operating system is an infinite loop
    - Continuously wait until more tasks are entered
    - Continuously dispatch available tasks

# Algorithms vs. Programs (Cont'd)

- **Which program(s) can always terminate in a finite number of steps**?
  1. Testing whether any given number is a prime
  2. Calculating 10000! (i.e, factorial(10000))
  3. Displaying all prime numbers
  4. Deciphering an RSA-encoded message without knowing the private key

# Algorithms vs. Programs (Cont'd)

- Primality test
  - Even with the brutal force method, it can terminate in a finite number step
- Calculating factorial(10000)
  - Factorial(10000) is an astronomical figure (天文數字) though, it involves a finite number of digits. So the program can terminate in a finite number step
- Displaying all prime numbers
  - Since there are infinitely many primes, this program never terminates

# 10000 Factorial

- 10000 factorial is 35,659 digits long. Here it is:
  2846259680917054518906413212119868890148051401702799230794179994274411
  3400037644437729907867577847758158840621423175288300423399401535187390
  5242116138271617481982419982759241828925978789812425312059465996259867
  0656016157203603239792632873671705574197596209947972034615369811989709
  2611277500484198845410475544642442136573303076703628825803548967461117
  0973695786036701910715127305872810411586405612811653853259684258259955
  8468814643042558983664931705925171720427659740744613340005419405246230
  3436869154059404066227828248371512038322178644627183822923899638992827
  2218797024593876938030946273322925705554596900278752822425443480211275
  5901916942542902891690721909708369053987374745248337289952180236328274
  1217040268086769210451555840567172555372015852132829034279989818449313
  6106403814893044996215999993596708929801903369984844046654192362584249
  4716317896119204123310826865107135451684554093603300960721034694437798
  2349430780626069422302681885227592057029230843126188497606560742586279
  4488271559568315334405344254466484168945804257094616736131876052349822
  8632645292152942347987060334429073715868849917893258069148316885425195
  6006172372636323974420786924642956012306288720122652952964091508301336
  6309827338063539729015065818225742954758943997651138655412081257886837
  0423920876448476156900126488927159070630640966162803878404448519164379
  0807186112370622133415415065991843875961023926713276546986163657706626
  …
  http://gimbo.org.uk/texts/ten_thousand_factorial.txt

# Algorithms vs. Programs (Cont'd)

- Breaking RSA
  - This problem corresponds to factorization (質因數分解)
    - Factorization as well as breaking RSA is feasible in a finite number of steps
  - RSA is based on the belief (not a proof) that factoring large integers (particularly that with exactly two huge prime factors) is difficult
    - E.g., cost thousands of years with a GHz computer
  - Conspiracy theory (陰謀論)
    - Since the proof is unknown nowadays, some people oppositely believe that some countries have efficient ways to do factorization!!

- Interested students may want to take a Cryptography class

# Describing Algorithms

- Many allowable ways
  - Programming languages (e.g., C++)
  - Natural languages
    - Must assure definiteness and effectiveness
  - Pseudocode (e.g., combining C, C++, and English)
    - Less language-dependent
    - More flexibility
  - Graphic representations (i.e., flowcharts)
    - Typically for small and simple algorithms only

# Algorithm Specification

- Examples
  - Selection sort
  - Binary search
  - Permutation generator
- Focuses
  - Inputs and outputs
  - Clear and basic-enough instructions
  - Finiteness and correctness proofs

# Selection Sort

- Input
  - A collection of n integers, $n \geq 1$
- Output
  - A collection of n integers
- Instructions

```
void SelectionSort(int *a, const int n)
{ //Sort the n integers a[0] to a[n-1] into non-decreasing order.
    for(int i=0; i<n; i++) {
        exam a[i] to a[n-1] and suppose the smallest one is at a[j];
        interchange a[i] and a[j];
    }
}
```

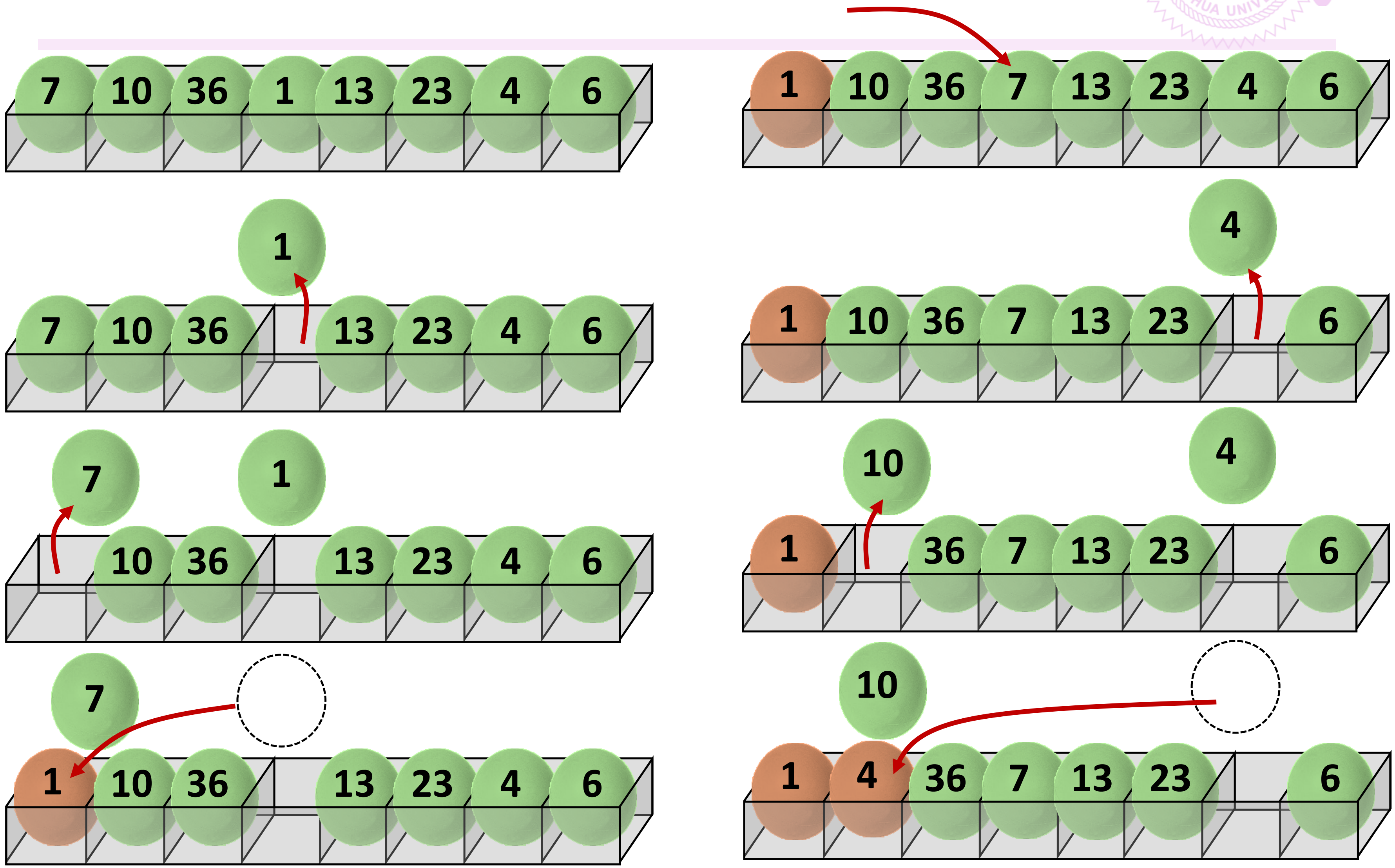
# Selection Sort — C++

```
void SelectionSort(int *a, const int n)
{ // Sort the n integers a[0] to a[n-1] into
  // non-decreasing order.
    for(int i=0; i<n; i++)
    {
        int j=i;
        //find the smallest integer in a[i] to a[n-1]
        for(int k = i+1; k<n; k++)
            if(a[k] < a[j]) j = k;
        swap(a[i], a[j]);
    }
}
```

```
void swap(int & i, int & j)
{
    int temp = i;
    i = j;
    j = temp;
}
```

Passed by reference

# Illustration



# Selection Sort — Proof

- For any i = q, following the execution of the shaded lines, it is the case that a[q]≤a[r], q+1 ≤ r ≤ n-1.
- When i becomes greater than q, a[0] … a[q] is unchanged.
- Hence, after the lines are executed for n-1 times (i.e., 0 ≤ i ≤ n-2), the following n-1 inequalities hold
  - a[0]≤a[r], 1 ≤ r ≤ n-1
  - …
  - a[n-3]≤a[r], n-2 ≤ r ≤ n-1
  - a[n-2]≤a[r], n-1 ≤ r ≤ n-1
- a[0] … a[n-1] is unchanged for the last iteration (i.e., i = n-1)
- Combining these inequalities leads to a[0]≤a[1]≤ … ≤ a[n-1]

```
void SelectionSort(int a[], const int n
{ // Sort the n integers into
  // non-decreasing order.
    for(int i=0; i<n; i++)
    {
        int j=i;
        //find the smallest integer in
        for(int k = i+1; k<n; k++)
            if(a[k] < a[j]) j = k;
        swap(a[i], a[j]);
    }
}
```

# Binary Search

- Input
  - n≥1 distinct integers that are already sorted and stored in the array a[0] … a[n-1]
  - Integer x

- Output
  - If x is present in the array, produce j such that x == a[j]
  - Otherwise, produce -1

# Binary Search — Pseudocode

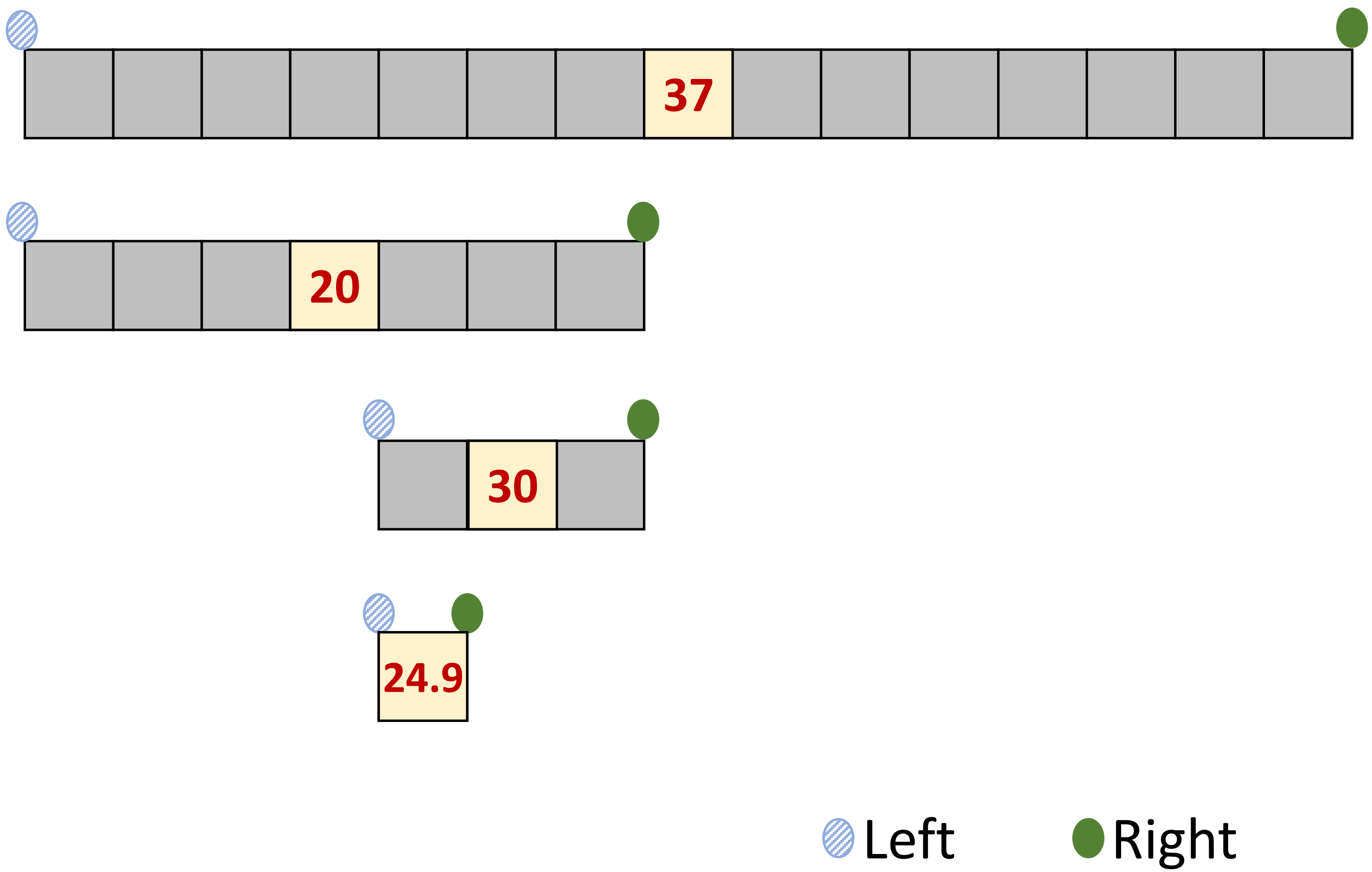
```
void BinarySearch(int *a, const int x, const int n)
{ // Search the sorted array a[0], … , a[n-1] for x
  // left and right are set to the two ends of a[]
  while(there're elements between the two ends)
  {
      Let middle be the middle element;
      if(x < a[middle])         set right to middle-1;
      else if(x > a[middle])    set left to middle+1;
      else                      return middle;
  }
  Not found;
}
```

# Binary Search — C++

```
int BinarySearch(int *a, const int x, const int n)
{ //Search the sorted array a[0]…a[n-1] for x
    int left = 0, right = n-1;
    while(left <= right)
    {//there are more elements
        int middle =(left+right)/2;
        if(x < a[middle])       right=middle-1;
        else if(x > a[middle])  left = middle+1;
        else                    return middle;
    }//end of while
    return -1;
}
```

# Binary Search — Illustration

Search a number, 25, in a sorted array of boxes



# Recursion

- Definition
  - Functions that invoke themselves
    - Directly or Indirectly through other functions

- Recursion is powerful
  - Divide and conquer
  - Method of induction (歸納法)
  - Can simplify the expression of an otherwise complex process

# Recursion (Cont'd)

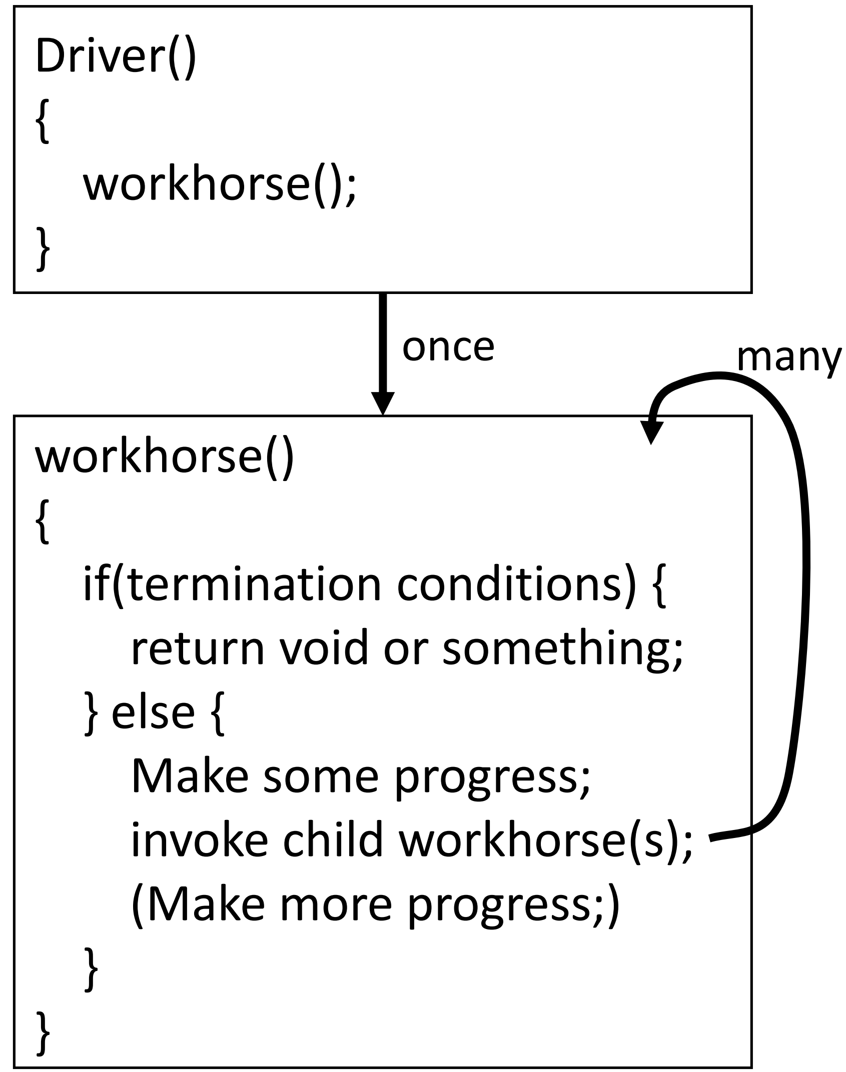
- Recursion is particular useful for
  - Factorial (階乘)
  - Binomial coefficients
  - Binary search
  - Problems that are recursively defined

- Recursion is not limited to the above tasks
  - Recursion can simulate looping
    (Looping can simulate recursion, too)

- Recursion tends to be (i.e., 有這個傾向，但不是絕對) slower than looping
  - Because function calls typically incur more latency than loop branches

# Develop Recursion

- Key components
  - Driver
    - Invoke the first workhorse
  - Workhorse(s)
    - Self-similar piece of the algorithm
  - Termination condition(s)
    - Determine whether no more progress needs be made
    - If a workhorse fails to check termination conditions, the program can never end
  - Make some progress
    - If nothing changes before the workhorse is again invoked, the program can never end



# Recursive Selection Sort

```
void SelectionSort(int a[], const int n)
{
    // 1-entry array does not need sorting
    if(n==1) return;

    int j=0;
    /* find the smallest in the received
         array and place it at the first */
    for(int k = 0; k<n; k++)
        if(a[k] < a[j]) j = k;
    swap(a[0], a[j]);

    SelectionSort(a+1, n-1); //recursion
}
```

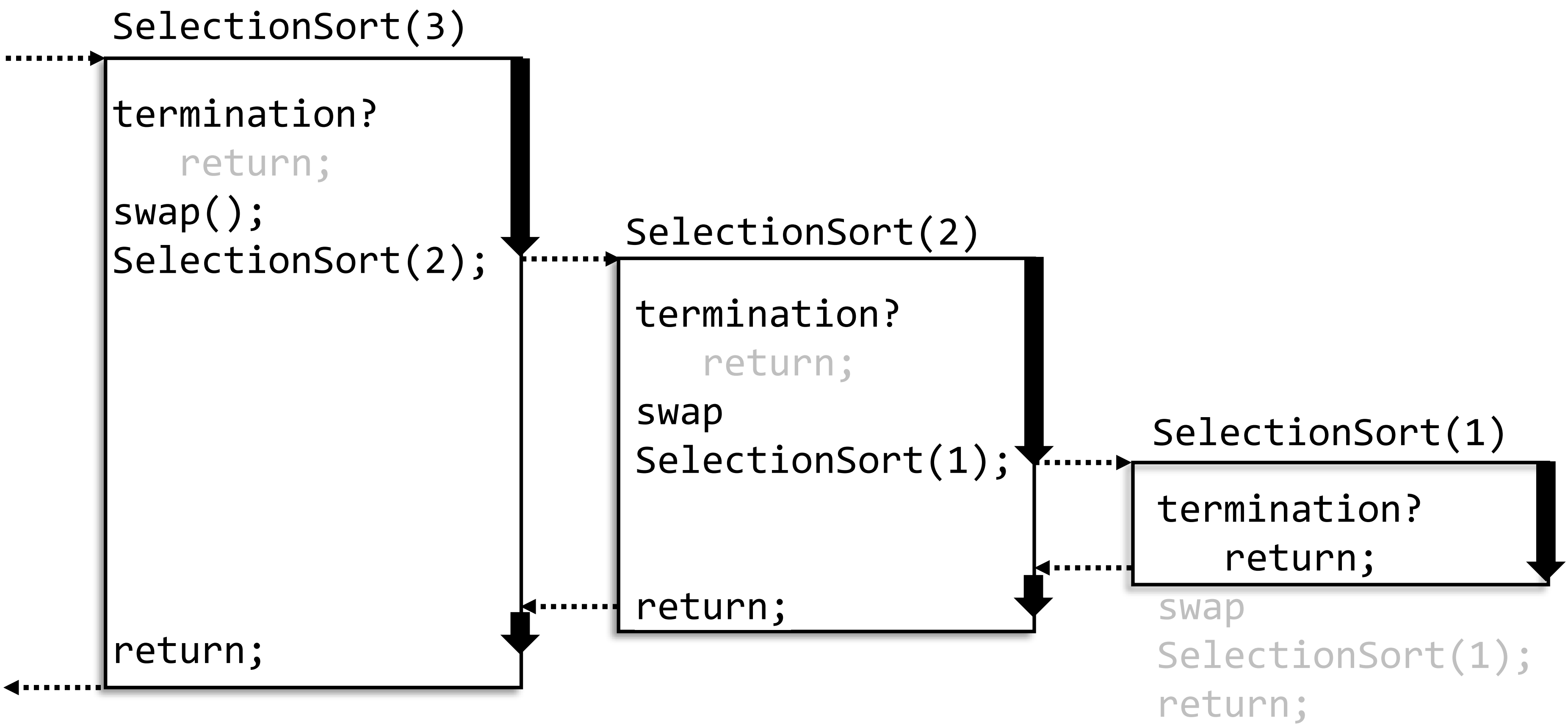
Termination condition

Create a new workhorse to sort the remaining n-1 elements

- This is an exampling recursive algorithm derived from an non-recursive one. In this example, recursion is easier to understand but likely performs slower than its non-recursive counterpart.

# Recursive Selection Sort

- Sort 3 elements



# Recursive Binary Search

```
int BinarySearch(int a[], const int x, const int left, const int right)
{
    // no entries to search
    if(left>right) return -1;

    int middle = (left+right)/2;

    if(x<a[middle])          return BinarySearch(a, x, left, middle-1);
    else if(x>a[middle])     return BinarySearch(a, x, middle+1, right);
    else                     return middle;
}
```

Termination condition

Create a new workhorse to search the half that possibly contain the target

# Permutation Generator

- Input
  - A set of n≥1 elements
- Output
  - Print all n! possible permutations of this set
- Example
  - Permutations of (a, b, c)
    - (a, b, c), (a, c, b),
      (b, a, c), (b, c, a),
      (c, a, b), (c, b, a)

# Permutation Generator — Observation

- Permutations of (a, b, c, d) can be constructed by
  - ‘a’ followed by all permutations of (b, c, d)
  - ‘b’ followed by all permutations of (a, c, d)
  - ‘c’ followed by all permutations of (a, b, d)
  - ‘d’ followed by all permutations of (a, b, c)

- Clue to recursion
  - Solve an n-element problem based on the results of an (n-1)-element problem

# Recursive Permutation Generator

```
void Permutations(int a[], const int k, const int m)
{
  if(k == m) {
        for(int i=0; i<=m; i++) cout << a[i] << " ";
        cout << endl;
        return;
  }

  for(int i=k; i<=m; i++) {
    swap(a[k], a[i]);  //enumerate all possible elements at a[k]
    Permutations(a, k+1, m);  // a workhorse to handle the rest
    swap(a[k], a[i]);  //restore the element
  }
}
```

Termination condition

Note that in this algorithm, a workhorse can generate new workhorses multiple times

It is a bit hard (but still feasible) to transform this algorithm into an non-recursion version

# Outline

- 1.1 Overview: System Life Cycle
- 1.2 Object-Oriented Design
- 1.3 Data Abstraction and Encapsulation
- (1.4 Basics of C++)
- 1.5 Algorithm Specification
- (1.6 Standard Template Library)
- **1.7 Performance Analysis and Measurement**

# Complexity

- Time complexity
  - Amount of execution time a program needs to solve a problem
- Space complexity
  - Amount of memory space a program needs to solve a problem
- We want to find complexity as a function of problem size
  - Problem size ≡ the total amount of input information

# Space Complexity

- Breakdown
  - Problem size-dependent part
    - Variables whose size/number depends on problem size
  - Fixed part
    - Space for storing the program
    - Fixed amount of variables during computation
    - Read-only space for Inputs
    - Write-only space for outputs
- We shall focus on the problem size-dependent part

# Space Complexity (Cont'd)

```
float abc (float a, float b, float c)
{
   float y;
   y = a+b+b*c+(a+b-c)/(a+b);
   return y;
}
```

- **a, b,** and **c** are read-only inputs
- A fixed amount of space is required to do the computation

→ Problem size-dependent part is 0

```
float sum (float *a, const int n)
{
  float s = 0;
  for (int i = 0; i < n; i++)
     s += a[i];
  return s;
}
```

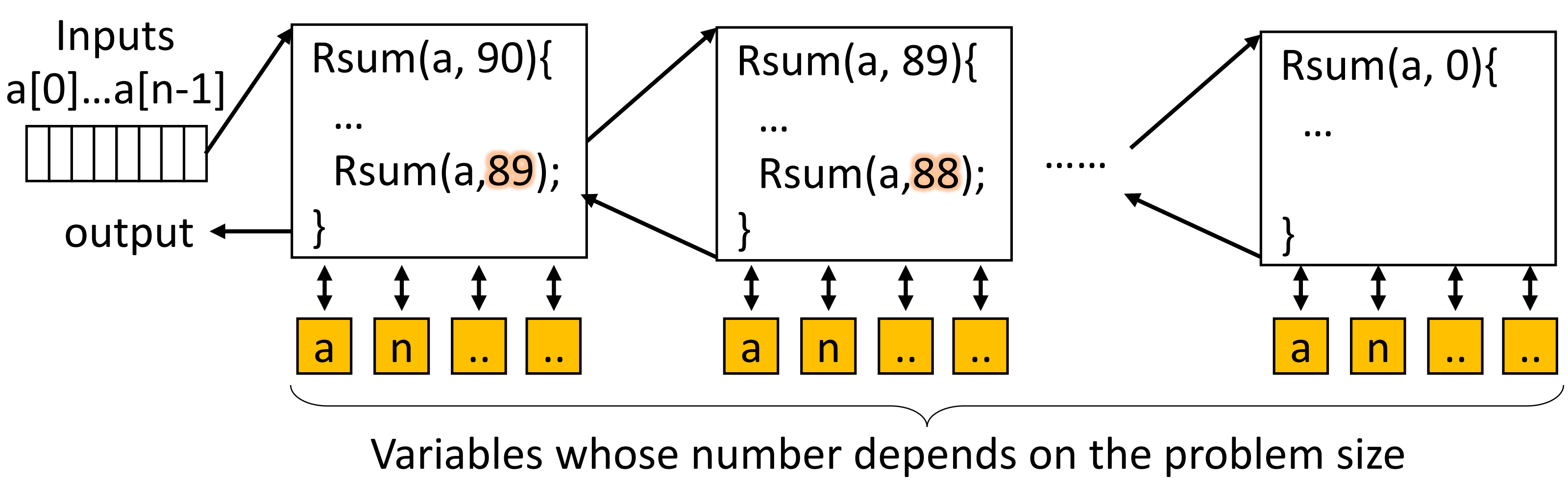
- **a[0]…a[n-1]** are read-only inputs
- Float * **a**, const int **n**, float **s**, int **i**, etc. consume a fixed amount of space

→ Problem size-dependent part is 0

# Space Complexity

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

- a[0]…a[n-1] are read-only inputs
- float * a and int n (and other variables local to Rsum()) consume a fixed amount of space for each execution of Rsum though, Rsum is called n+1 times.
- Space complexity = c(n+1), where c is a constant, say c=4



# Time Complexity

- Breakdown
  - Execution time
  - Compile time (fixed part)
- Execution time is important
  - Problem size, $n$,↑ ⇒ execution time, $t_P(n)$, may ↑
- Compile time is less important
  - Independent of problem size, $n$
  - Only present for the first execution

# Methods to Derive Execution Time

1. Derive the exact formula
   - $t_P(n) = c_a ADD(n) + c_s SUB(n) + c_m MUL(n) + \cdots$
   - However, it is almost impossible to obtain such a formula in real world

2. Step counts

3. Asymptotic notation (漸近表示法) of step counts

4. Real system measurement

# Step Count

- Definition of a step
  - A segment of program whose execution time is independent of problem size

- Example of a step
  - One addition → a step
  - One multiplication → a step
  - 1000 additions → a step
  - 1000 multiplications → a step
  - r = a+b+b*c+(a+b-c)/(a+b)+4.0 → a step

- The following one is NOT a step
  - $n$ additions, where $n$ is the size of the input array

# Zero-Step Program Segments

- Comments
  - // this is binary search
  - /* this is
     * selection sort
     */
- Declarative statements of variables and functions
  - int a;
  - float b, c, d;
  - int max(a, b);
- Brackets, line labels, and the *else* keyword
  - {
  - }
  - } else {
  - END:

# Single-Step Program Segments

- Assignments and expressions
  - int a = 10;
  - b = 0.1;
  - c = a + b * d;
- Looping statements (single-step per loop iteration)
  - for(int i=0; i<n; i+=3)
  - while(j<$n^2$)
  - do … while(n>10)
- Functions that independent of problem size
  - a = max(b, c)
- Conditional statements
  - if(a > 10)
- Unconditional branches
  - goto, break, continue, return

# Those May Depend on Problem Size

- Object/variable construction
  - int *a = new int[size(input)];
- Function execution
  - MatrixAdd(a, b, c); // adding two matrixes
- Parameter passing
  - Passing an object whose size depends on problem size
- Statements that involve the above events
  - int a = sum(a, n);
  - if(search(a, x, n) == true)

# Methods of Obtaining Step Count

- Instrumentation (實際測量)
  - Introduce a new global variable: *count*
  - Initialize *count* to zero
  - Add statements to increment *count* for each step
  - Report *count*

- Table analysis (紙筆分析)
  - List the step count of each program segment
  - List the frequency of each program segment
  - Summarize the total step count

# Step Counting — Example 1

```
float sum (float *a, const int n)
{
  float s = 0;
  for (int i = 0; i < n; i++)
     s += a[i];
  return s;
}
```

# Step Counting Using Instrumentation

```
float sum (float *a, const int n)
{
  float s = 0;
  count++; // count is global
  for (int i = 0; i < n; i++) {
    count++;  // for loop
    s += a[i];
    count++; // assignment
  }
  count++; // last time of for
  count++; // return
  return s;
}
```

Simplified version

```
void sum (float *a, const int n)
{
  for (int i = 0; i < n; i++) {
    count+=2;
  }
  count+=3;
  return;
}
```

# Step Counting Using a Table

| Code | s/e | freq. | subtotal |
|---|---|---|---|
| `float sum (float *a, const int n)` | | | |
| `{` | 0 | | |
| `float s = 0;` | 1 | 1 | 1 |
| `for (int i = 0; i < n; i++)` | 1 | n+1 | n+1 |
| `s += a[i];` | 1 | n | n |
| `return s;` | 1 | 1 | 1 |
| `}` | 0 | | |
| | | **total**: | **2n+3** |

**s/e**: steps per execution

The frequency of executing the control statement is one time more than that of the loop body.

# Step Counting — Example 2

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

- Recursion

# Step Counting — Instrumentation

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

count is a global variable and will be incremented throughout the entire recurrent computation.

# Step Counting — Table

| | | freq. | | subtotal | |
|---|---|---|---|---|---|
| **float** Rsum (**float** *a, **const int** n) | **s/e** | **n=0** | **n>0** | **n=0** | **n>0** |
| { | 0 | | | | |
| **if** (n <= 0) | 1 | 1 | 1 | 1 | 1 |
| **return** 0; | 1 | 1 | 0 | 1 | 0 |
| **else** | 0 | | | | |
| **return** (Rsum(a, n-1) + a[n-1]); | 1+t(n-1) | 0 | 1 | 0 | 1+t(n-1) |
| } | 0 | | | | |
| | | | **total** | **2** | **2+t(n-1)** |

**s/e**: steps per execution

Recurrence relations:

$$t(n) = \begin{cases} 2 + t(n-1), n > 0 \\ 2, otherwise \end{cases}$$

# Solving Recurrence

- Technique
  - Repeatedly substituting

- $t(n) = 2 + \boxed{t(n-1)}$
  $= 2 + \boxed{2 + t(n-2)}$
  $= 2 + 2 + \cdots + 2 + t(0)$
  $= 2n + t(0)$
  $= 2n + 2$

# Step Counting — Example 3

```
void MatAdd (int **a, int **b, int **c, int m, int n)
{
  for (int i = 0; i < m; i++) {
        for (int  j = 0; j < n; j++) {
          c[i][j] = a[i][j] + b[i][j];
      }
  }
  return;
}
```

Program containing nested loops

# Step Counting — Instrumentation

```
void MatAdd (int **a, int **b, int **c, int m, int n)
{
  for (int i = 0; i < m; i++) {
        count++; // for loop i
        for (int  j = 0; j < n; j++) {
          count++; // for loop j
          c[i][j] = a[i][j] + b[i][j];
          count++; // assignment
      }
      count++; // last time of the for loop j
  }
  count++; // last time of the for loop i
  count++; // return statement
  return;
}
```

The textbook omits the return

# Step Counting — Table

| Code | s/e | freq. | subtotal |
|---|---|---|---|
| **void** MatAdd (**int** **a, **int** **b, **int** **c, **int** m, **int** n) | | | |
| { | 0 | | |
| **for** (**int** i = 0; i < m; i++) | 1 | m+1 | m+1 |
| **for** (**int** j = 0; j < n; j++) | 1 | m(n+1) | mn+m |
| c[i][j] = a[i][j] + b[i][j]; | 1 | mn | mn |
| **return**; | 1 | 1 | 1 |
| } | 0 | | |
| | | **total**: | **2mn+2m+2** |

The textbook omits the return

We are allowed to use more than one variables to describe problem size

# Step Counting — Example 4

```
void fibonacci (int n)  //compute the Fibonacci number F[n]
{
    if (n <= 1)   // steps = 1
        cout << n << endl; // F[0] = 0 and F[1] = 1    // steps = 1
    else { // compute F[n]
        int fn;  int fnm2 = 0;  int fnm1 = 1;     // steps = 2
        for (int i = 2; i<=n; i++) {  // steps = n
          fn = fnm1 + fnm2;  ┐
          fnm2 = fnm1;       ├ // steps = 3(n-1)
          fnm1 = fn;         ┘
        } // end of for
        cout << fn << endl;   // steps = 1
    } // end of else
    return;           // steps = 1
} // end of fibonacci
```

If n > 1,

$t(n) = 1 + 2 + n + 3(n-1) + 1 + 1$

$= 4n+1$

Otherwise, $t(n) = 1+1+1 = 3$

# Inexactness of Step Count

- We cannot know which following step count number represents the shortest execution time, where n stands for the problem size
  - Step(Alg1) = n+1
  - Step(Alg2) = n+1000
  - Step(Alg3) = 1000n
  - Step(Alg4) = 1000n+1000

> Since the notion of a step is (deliberately) imprecise.
> 1 step can be 1 multiplication or multiple multiplications

- But we know the execution time of these programs linearly increases with problem size

# Motivation of Asymptotic Notation

- We also know the fifth algorithm exhibits the shortest execution time once the problem size, n, is large enough
  - Step(Alg1) = n+1
  - Step(Alg2) = n+1000
  - Step(Alg3) = 1000n
  - Step(Alg4) = 1000n+1000
  - Step(Alg5) = $\log_2(n)+1000$

Alg1–Alg4: Linearly increase

Alg5: Logarithmically increase

- Asymptotic Notations are introduced to describe/emphasize
  - Trend that an algorithm's step count increases with problem size
  - Classification of problems/algorithms based on the trend

# Asymptotic Notations (O, Ω, Θ)

| | | |
|---|---|---|
| **O** | Big O | Upper bound |
| **Θ** | Theta | Tight bound |
| **Ω** | Omega | Lower bound |

- “f(n) = O(n)” read as
  - “f of n is big O of n”
- We can alternatively say “f(n) ∈ O(n)”
  - “f of n belongs to big O of n”

# Examples of Asymptotic Notations

- Upper-bound (**O**) descriptions of the time complexity (i.e., in step counts)
  - Alg1 : n+1 = **O**(n)
  - Alg2 : n+1000 = **O**(n)
  - Alg3 : 1000n = **O**(n)
  - Alg4 : 1000n+1000 = **O**(n)
  - Alg5 : $\log_2(n)+1$ = **O**(n)
- Meanings
  - Their time complexity **is no more than n**
- n denotes the problem size and we focus on large problem size for asymptotic notations

# Examples of Asymptotic Notations

- Following upper-bound statements are both true
  - Alg5 : $\log_2(n)+1 = \mathbf{O}(n)$
  - Alg5 : $\log_2(n)+1 = \mathbf{O}(\log_2(n))$
- Meanings
  - The time complexity of Alg5 **is no more than log(n)**

# Examples of Asymptotic Notations

- Tight-bound (**Θ**) descriptions
  - Alg1 : n+1 = **Θ**(n)
  - Alg2 : n+1000 = **Θ**(n)
  - Alg3 : 1000n = **Θ**(n)
  - Alg4 : 1000n+1000 = **Θ**(n)
  - Alg5 : $\log_2(n)+1$ = **Θ**($\log_2(n)$)
- Meanings
  - The time complexity of Alg1~4 is **equal to n**
  - The time complexity of Alg5 is **equal to log(n)**

# Examples of Asymptotic Notations

- Lower-bound (**Ω**) descriptions
  - Alg1 : n+1 = **Ω**(n)
  - Alg2 : n+1000 = **Ω**(n)
  - Alg3 : 1000n = **Ω**(n)
  - Alg4 : 1000n+1000 = **Ω**(n)
  - Alg5 : $\log_2(n)+1$ = **Ω**($\log_2(n)$)
- Meanings
  - The time complexity of Alg1~4 **is no less than n**
  - The time complexity of Alg5 **is no less than log(n)**

# Examples of Asymptotic Notations

- These lower-bound (**Ω**) descriptions are true of course
  - Alg1 : n+1 = **Ω**($\log_2(n)$)
  - Alg2 : n+1000 = **Ω**($\log_2(n)$)
  - Alg3 : 1000n = **Ω**($\log_2(n)$)
  - Alg4 : 1000n+1000 = **Ω**($\log_2(n)$)
- Meanings
  - The time complexity of Alg1~4 **is no less than log(n)**

# Asymptotic Notations (O, Ω, Θ)

| | | |
|---|---|---|
| **O** | Big O | Upper bound |
| **Θ** | Theta | Tight bound (i.e., both an upper bound and lower bound ) |
| **Ω** | Omega | Lower bound |

- "f(n) = O(n)" read as
  - "f of n is big O of n"
- We can alternatively say "f(n) ∈ O(n)"
  - "f of n belongs to big O of n"

- "**Big**" O → Upper
- "**Θ**" → A hyphen in the middle → tight bound

# Big O Definitions

- $f(n) = O(g(n))$ *iff*
  - there exist positive constants $c$ and $n_0$ such that $f(n) \le c \cdot g(n)$ for all n, $n \ge n_0$

"iff" means "if and only if" ("⟺")

"≤" suggests that c·g(n) is an upper bound of f(n)

"∀" means "for all"

- Example
  - $n+1 = O(n)$, $n+1 \le 2 \cdot n$ $\forall n \ge 1$
  - $n+1000 = O(n)$, $n+1000 \le 1001 \cdot n$ $\forall n \ge 1$
  - $1000n = O(n)$, $1000n \le 1000 \cdot n$ $\forall n \ge 1$
  - $1000n+1000 = O(n)$, $1000n+1000 \le 2000 \cdot n$ $\forall n \ge 1$
  - $\log(n)+1 = O(\log(n))$, $\log(n)+1 \le 2 \cdot \log(n)$ $\forall n \ge 10$

# Big O Definitions (Cont'd)

- More examples
  - $\underline{2n^2+3n+4} = O(\underline{n^2})$, $\quad \underline{2n^2+3n+4} \le 9 \cdot \underline{n^2} \quad \forall\, n \ge 1$
  - $\underline{2n^2+3n+4} = O(\underline{n^2})$, $\quad \underline{2n^2+3n+4} \le 90 \cdot \underline{n^2} \quad \forall\, n \ge 40$

> We may have an infinite number of c and n0 satisfying the inequality.

  - $\underline{2n^2+3n+4} = O(\underline{n^{2.1}})$,
  - $\underline{2n^2+3n+4} = O(\underline{n^{3}})$,
  - $\underline{2n^2+3n+4} = O(\underline{n^{99}})$,

> Since by definition, Big O does not need to be a tight bound, we may have infinite number of g(n) satisfying the inequality.

  - $\underline{2n^2+3n+4} \ne O(\underline{n^{1.9}})$,

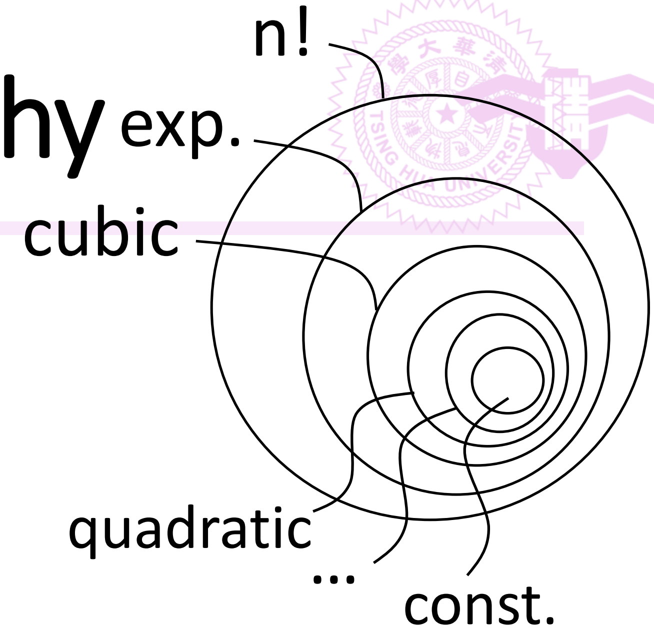
# Big O of a Polynomial Function

- **Theorem 1.2**
  - $f(n) = \sum_{i=0}^{m} a_i n^i = a_m \boldsymbol{n^m} + \ldots + a_1 n + a_0$
    $\Longrightarrow f(n) = \boldsymbol{O}(\boldsymbol{n^m})$
- Proof
  - $f(n) = \sum_{i=0}^{m} a_i n^i \leq \sum_{i=0}^{m} |\boldsymbol{a_i}| n^i$

$$= \boldsymbol{n^m} \sum_{i=0}^{m} |a_i| n^{\boldsymbol{i-m}}$$

$$\leq n^m \sum_{i=0}^{m} |a_i| \quad \text{,for } n \geq 1$$

# Common Big O Hierarchy




| | |
|---|---|
| • O(**n!**) | factorial |
| • O(**2**$^n$) | exponential |
| • O(**n**$^k$) | |
| • … | |
| • O(**n**$^3$) | cubic |
| • O(**n**$^2$) | quadratic |
| • O(**nlog(n)**) | log-linear |
| • O(**n**) | linear |
| • O(**n**$^{0.x}$) | sub-linear |
| • O(**log(n)**) | logarithm |
| • O(**1**) | constant |

O($n^2$) algorithms/problems are also O($n^3$) ones, and so on

Many other classes are not listed here, e.g., O($n^{1.5}$), O(loglog(n)), O(nlog$^2$(n))…

- **O(1)** means that the execution time is independent of problem size
- E.g., time for retrieving the $k^{th}$ entry of an array (of size n) is O(1)

# Omega Definitions

- $f(n) = \Omega(g(n))$ *iff*
  - there exist positive constants $c$ and $n_0$ such that $f(n) \geq c \cdot g(n)$ for all $n$, $n \geq n_0$

↕ Compare with Big O

such that $f(n) \leq c \cdot g(n)$ for all $n$, $n \geq n_0$

- Example
  - $n+1 = \Omega(n)$, $n+1 \geq 1 \cdot n$ $\forall\, n \geq 1$
  - $n+1000 = \Omega(n)$, $n+1000 \geq 1 \cdot n$ $\forall\, n \geq 1$
  - $1000n = \Omega(n)$, $1000n \geq 1000 \cdot n$ $\forall\, n \geq 1$
  - $1000n+1000 = \Omega(n)$, $1000n+1000 \geq 1000 \cdot n$ $\forall\, n \geq 1$
  - $\log(n)+1 = \Omega(\log(n))$, $\log(n)+1 \geq 1 \cdot \log(n)$ $\forall\, n \geq 10$

# Omega Definitions (Cont'd)

- More examples
  - $2n^2+3n+4 = \Omega(n^2)$,
  - $2n^2+3n+4 = \Omega(n^{1.9})$,
  - $2n^2+3n+4 = \Omega(n)$,
  - $2n^2+3n+4 = \Omega(1)$,
  - $2n^2+3n+4 \neq \Omega(n^{2.1})$,

- **Theorem 1.3**
  - $f(n) = a_m \boldsymbol{n^m} + \ldots + a_1 n + a_0,\ a_m > 0$
    $\Longrightarrow f(n) = \boldsymbol{\Omega}(\boldsymbol{n^m})$

# Theta Definitions

- $f(n) = \Theta(g(n))$ *iff*
  - there exist positive constants $c_1$, $c_2$ and $n_0$ such that $c_1 \cdot g(n) \le f(n) \le c_2 \cdot g(n)$ for all n, $n \ge n_0$
  - i.e., $f(n)$ is $O(g(n))$ and $\Omega(g(n))$
- Example
  - $n+1 = \Theta(n)$, $1 \cdot n \le n+1 \le 2 \cdot n$ $\quad \forall\, n \ge 1$
  - $n+1000 = \Theta(n)$, $1 \cdot n \le n+1000 \le 1001 \cdot n$ $\quad \forall\, n \ge 1$
  - $1000n = \Theta(n)$, $1000 \cdot n \le 1000n \le 1000 \cdot n$ $\quad \forall\, n \ge 1$
  - $1000n+1000 = \Theta(n)$, $1000 \cdot n \le 1000n+1000 \le 2000 \cdot n$ $\quad \forall\, n \ge 1$
  - $\log(n)+1 = \Theta(\log(n))$, $1 \cdot \log(n) \le \log(n)+1 \le 2 \cdot \log(n)$ $\quad \forall\, n \ge 10$
- **Theorem 1.4**
  - $f(n) = a_m \boldsymbol{n^m} + \ldots + a_1 n + a_0,\ a_m > 0$
    $\Longrightarrow f(n) = \boldsymbol{\Theta}(\boldsymbol{n^m})$

# Step Counting — Asymptotic Notation

| | s/e | freq. | subtotal |
|---|---|---|---|
| **float** sum (**float** *a, **const int** n) | | | |
| { | 0 | | |
| **float** s **=** 0**;** | 1 | $\Theta(1)$ | $\Theta(1)$ |
| **for** (int i = 0; i < n; i++) | 1 | $\Theta(n)$ | $\Theta(n)$ |
| s += a[i]; | 1 | $\Theta(n)$ | $\Theta(n)$ |
| **return** s; | 1 | $\Theta(1)$ | $\Theta(1)$ |
| } | 0 | | |
| | **overall**: | | $\mathbf{\Theta(n)}$ |

**s/e**: number of steps per execution

# Step Counting — Asymptotic Notation

| (recursion of sum()) | | freq. | | subtotal | |
|---|---|---|---|---|---|
| **float** Rsum (**float** *a, **const int** n) | **s/e** | **n=0** | **n>0** | **n=0** | **n>0** |
| { | 0 | | | | |
| **if** (n <= 0) | 1 | Θ(1) | Θ(1) | Θ(1) | Θ(1) |
| **return** 0**;** | 1 | Θ(1) | 0 | Θ(1) | 0 |
| **else** | 0 | | | | |
| **return** (Rsum(a, n-1) + a[n-1]); | 1+t(n-1) | 0 | Θ(1) | 0 | Θ(1+t(n-1)) |
| } | 0 | | | | |
| | | **overall**: | | **Θ(1)** | **Θ(1+t(n-1))** |

**s/e**: number of steps per execution

# Step Counting — Asymptotic Notation

```
void MatAdd (int **a, int **b, int **c, int m, int n)

{
  for (int i = 0; i < m; i++)                    Θ(m)
     for (int  j = 0; j < n; j++)              } Θ(mn)
        c[i][j] = a[i][j] + b[i][j];
  return;
}
```

**overall**: **$\Theta(mn)$**

# Recursive Permutation Generator

```
void Permutations(int *a, const int k, const int m)
{
  // one element between k and m means one possible permutation
  if(k == m) {
    for(int i=0; i<=m; i++)
      cout << a[i] << " ";
    cout << endl;
    return;
  }

  for(int i=k; i<=m; i++) {
    swap(a[k], a[i]);
    Permutations(a, k+1, m);
    swap(a[k], a[i]);
  }
}
```

k==m
→ $\Theta(t(k, m)) = \Theta(m)$

$\Theta(t(k, m)) =$
$(m-k+1)\times\Theta(t(k+1, m)) + \Theta(1)$

$\Theta(1)$ comes from the if statement

# Recursive Permutation Generator

**Solve the recurrence**

$\Theta(t(k, m)) = (m-k+1)\times\Theta(t(k+1, m)) + \Theta(1)$ Eq. (1)

$\Theta(t(m, m)) = \Theta(m)$ Eq. (2)

Let k=0 and m=(n-1)

$$\begin{aligned}\Theta(t(0, n-1)) &= n \times \Theta(t(1, n-1)) + \Theta(1) \\ &= n \times (n-1)\times\Theta(t(2, n-1)) + \Theta(1) + \Theta(1) \\ &= \ldots \\ &= \underbrace{n \times (n-1) \times (n-2) \ldots \times 2}_{\text{n-1 terms}} \times\Theta(t(n-1, n-1)) + (n-1)\times\Theta(1)\end{aligned}$$

n-1 equations

$$\begin{aligned}&= n! \times \Theta(t(n-1, n-1)) + \Theta(n-1) \\ &= n! \times \Theta(n-1) + \Theta(n-1) \quad \text{... because of Eq. (2)} \\ &= \Theta(n \times n!)\end{aligned}$$

# Binary Search

```
int BinarySearch(int *a, const int x, const int n)
{ //Search the sorted array a[0], … , a[n-1] for x
    int left = 0, right = n-1;
    while(left <= right)
    {//there are more elements
        int middle =(left+right)/2;
        if(x<a[middle])       right=middle-1;
        else if(x>a[middle]) left = middle+1;
        else return  middle;
    }//end of while
    return -1;
}
```

$\Theta(\log(n))$

# Magic Square

| | | | | | |
|---|---|---|---|---|---|
| | | | | | = 65 |
| 15 | 8 | 1 | 24 | 17 | = 65 |
| 16 | 14 | 7 | 5 | 23 | = 65 |
| 22 | 20 | 13 | 6 | 4 | = 65 |
| 3 | 21 | 19 | 12 | 10 | = 65 |
| 9 | 2 | 25 | 18 | 11 | = 65 |
| = 65 | = 65 | = 65 | = 65 | = 65 | = 65 |

# Magic Square Algorithm

```
void magic (int n)
// create a magic square of size n, n is odd
{
  const int MaxSize = 51; // maximum square size
  int square[MaxSize][MaxSize], k, l;

  // check correctness of n
  if ((n > MaxSize) || (n < 1)) {
    cerr << "Error!..n out of range \n";
    eturn;
  }else if (!(n%2)) {
    cerr << "Error!..n is even \n";
    return;
  }
  // n is odd. Coxeter's rule can be used
  for (int i = 0; i < n; i++) // initialize square to 0
     for (int j = 0; j < n; j++)
        square[i][j] = 0;
  square[0][(n-1)/2] = 1; // middle of first row
  // please continue to the next slide…
```

Declarations (MaxSize, square, k, l): $\Theta(1)$

Check correctness of n: $\Theta(1)$

Initialize square to 0: $\Theta(n^2)$

Middle of first row: $\Theta(1)$

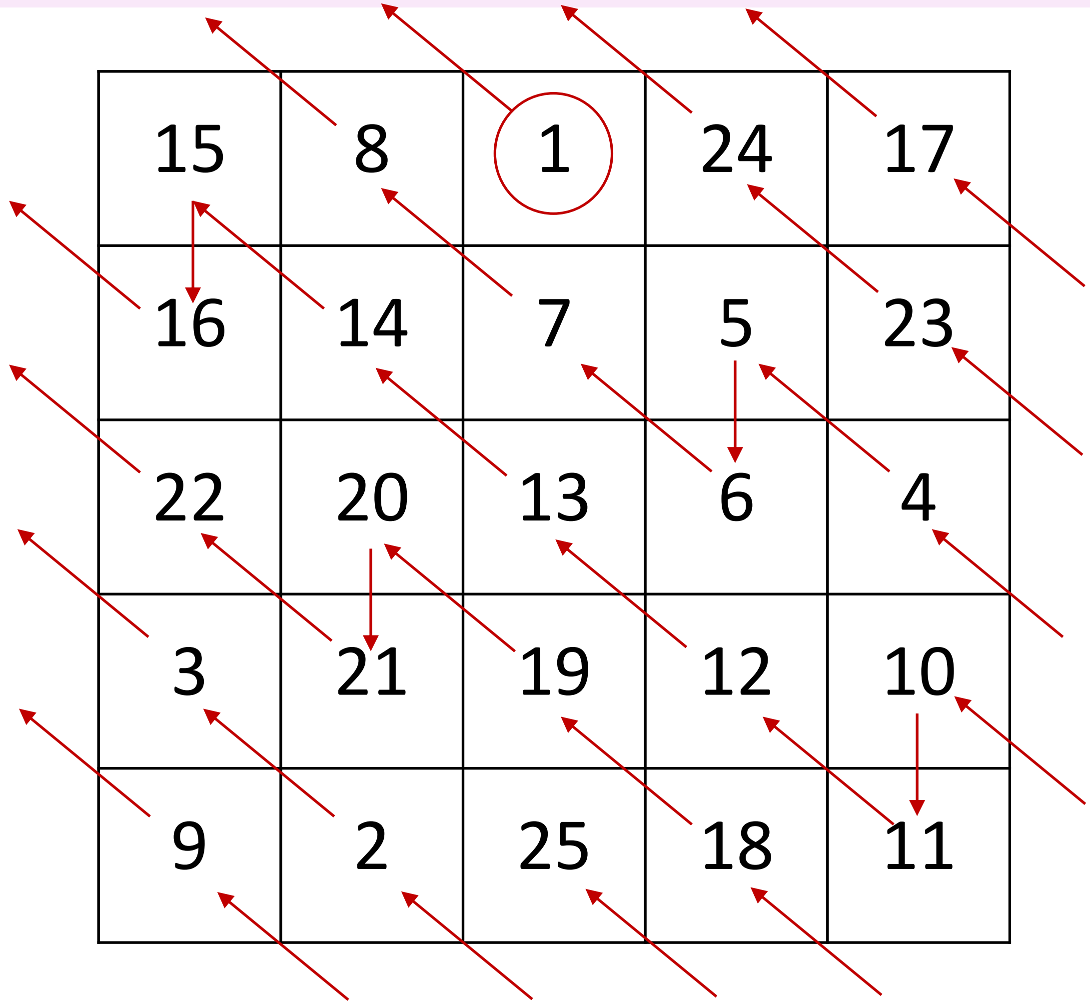
```
  // i and j are current position
  int key = 2; i = 0;                                    } Θ(1)
  int j = (n-1)/2;
  while (key <= n*n) {
  // move up and left
     if (i-1 < 0)  k = n-1; else k = i-1;
     if (j-1 < 0)  l = n-1; else l = j-1;
     if (square[k][l])  i = (i+1)%n;
     else { // square[k][l] is unoccupied                } Θ(n²)
        i = k;
        j = l;
     }
     square[i][j] = key;
     key++;
  } // end of while
  // output the magic square
  cout << "magic square of size " << n << endl;           } Θ(1)
  for ( i = 0; i < n; i++) {
     for ( j = 0; j < n; j++)
        cout << square[i][j] << " ";                      } Θ(n²)
     cout << endl;
  }
}
```

# Magic Square (Cont'd)

- We just show how can we quickly analyze the complexity of an algorithm without knowing all the details

- $\Theta(n^2)$ is the optimal one we can achieve (in terms of asymptotic complexity) to generate an $n^2$ magic square
  - Since there are $n^2$ positions the algorithm must place a number

# Magic Square Underlying Concept

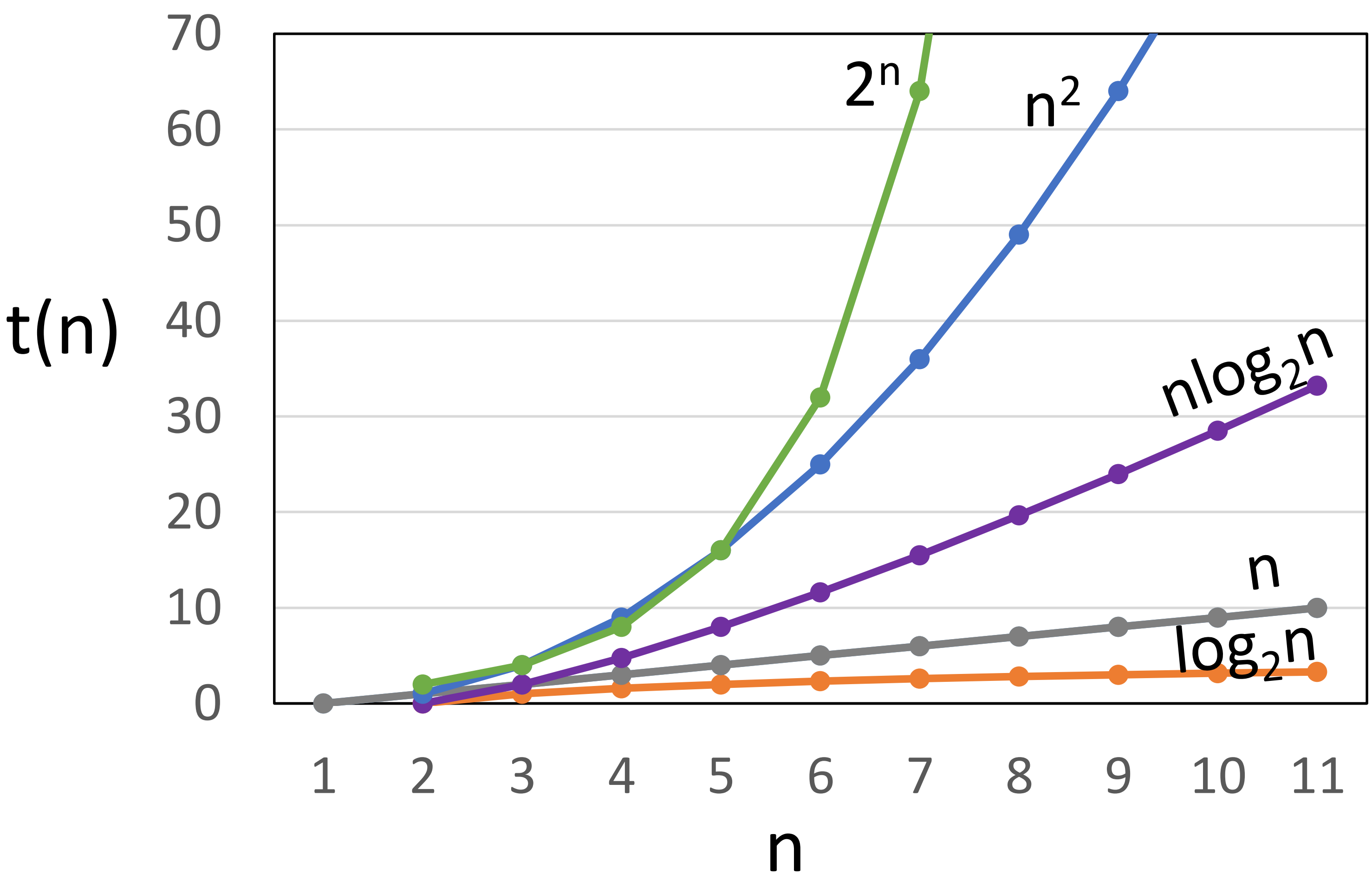


# Practical Complexities

| Prob. size | n | nlog(n) | $n^2$ | $n^3$ | $n^4$ | $2^n$ |
|---|---|---|---|---|---|---|
| $10^3$ | 1 µs | 10 µs | 1 ms | 1 s | 17 min | $3.2 \times 10^{283}$ y |
| $10^4$ | 10 µs | 130 µs | 100 ms | 17 m | 116 d | |
| $10^5$ | 0.1 ms | 1.7 ms | 10 s | 12 d | 3171 y | |
| $10^6$ | 1 ms | 20 ms | 17 m | 32 y | $3 \times 10^7$ y | |

Assume a computer that performs 1 billion steps per second

# Practice Complexity



# Performance Measurement

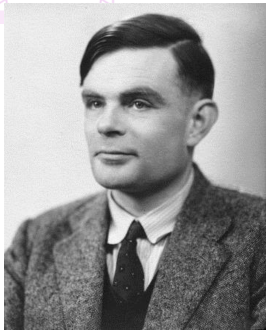
- Techniques
  - Use time-related library functions
    - gettimeofday()
    - clock()
    - time()
  - Repeatedly measure a program to reduce noises
  - Use randomized inputs to obtain best-case, average, and worst-case execution time
  - Predict the execution time of a problem with different input size
    - Regression (curve fitting)
    - Interpolation
    - Extrapolation
- Please read Section 1.7.2 for details

# Performance Measurement

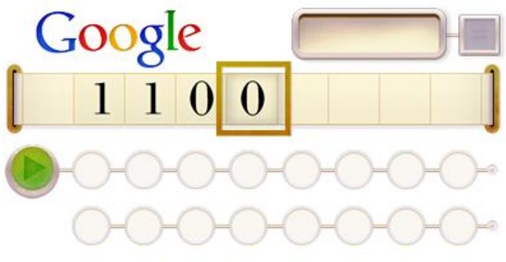
- Limitations of asymptotic analysis
  - For two programs that are both $O(n^2)$ time complexity
    - We cannot tell which is faster
  - For one program that is $O(n)$ and the other is $O(n^2)$
    - Sometimes the problem size is not very large, and the $O(n^2)$ one actually is faster than the $O(n)$ one

- Performance measurement provide actual execution time

# Alan Turing



- One of the greatest computer scientists and computational theorists
  - Complexity analysis is part of computational theory
- Often called the father of modern computing
- Some famous things
  - Turing award (圖靈獎)
    - Nobel Prize of computing
  - Turing machine (圖靈機)
    - Theoretical computer model
    - http://www.google.com/doodles/alan-turings-100th-birthday
  - Turing test (圖靈測試)
    - Test of a computer's ability to exhibit behavior equivalent to human



# Alan Turing (Cont'd)

- The Imitation Game
  - A movie about Alan Turing trying to crack the Enigma code during World War II
  - In Taiwan's theaters recently!!
  - IMDB 8.2

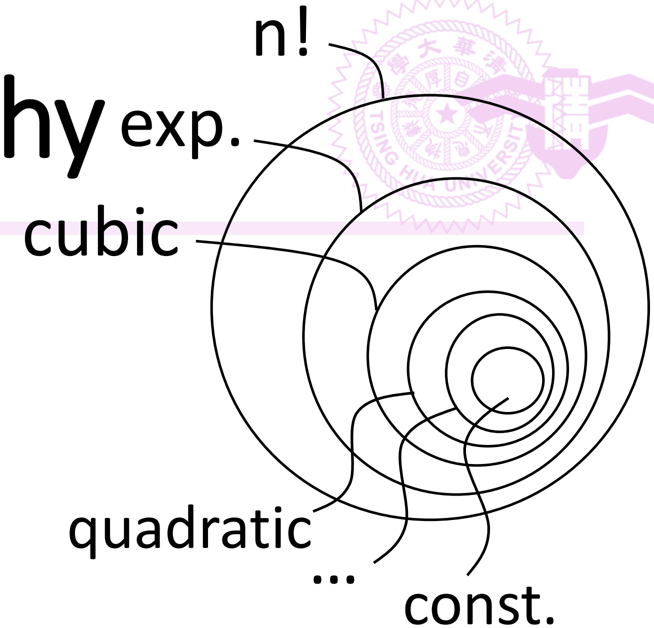

## User Reviews

★★★★★★★★★★ **Compelling and Enthralling from start to finish.**

16 October 2014 | by fruitbat00 (United Kingdom) – See all my reviews

Truly excellent film and definitely Ocsar worthy material for both the film and the actors. The entire cast are amazing.

# Common Big O Hierarchy




| | |
|---|---|
| • $O(\mathbf{n!})$ | factorial |
| • $O(\mathbf{2^n})$ | exponential |
| • $O(\mathbf{n^k})$ | |
| • … | |
| • $O(\mathbf{n^3})$ | cubic |
| • $O(\mathbf{n^2})$ | quadratic |
| • $O(\mathbf{n\log(n)})$ | log-linear |
| • $O(\mathbf{n})$ | linear |
| • $O(\mathbf{n^{0.x}})$ | sub-linear |
| • $O(\mathbf{\log(n)})$ | logarithm |
| • $O(\mathbf{1})$ | constant |

$O(n^2)$ algorithms/problems are also $O(n^3)$ ones, and so on

Many other classes are not listed here, e.g., $O(n^{1.5})$, $O(\log\log(n))$, $O(n\log^2(n))$…

- **O(1)** means that the execution time is independent of problem size
- E.g., time for retrieving the $k^{th}$ entry of an array (of size n) is $O(1)$

# Time Complexity of Learning DS

- $\Theta(1)$
  - Number of weeks in the semester
    = 18 = $\Theta(1)$
  - Number of chapters covered in the semester
    = 8 = $\Theta(1)$
  - Time(read these chapters twice)
    = $2 \times 8 \times \text{Time}_{\text{read\_one\_chapter}}$
    = $\Theta(1)$