國立清華大學 電機工程學系
EE2410 Data Structure

Chapter 1
Basic Concepts

Outline

- **Overview**
  - System Life Cycle
- **Object-Oriented Software Design**
- **Data Abstraction and Encapsulation**
- **Basics of C++**
- **Algorithm Specification**
- **Performance Analysis and Measurement**

Why We Need Data Structure?

- **Elementary programming course**
  - Emphasizes syntax of a language
  - Solves small problems
  - Requires simple construct like array or while
- **This course**
  - Provides techniques to solve large-scale problem
  - Data abstraction, encapsulation, algorithm specification, performance analysis, and measurement are important
  - We discuss not just "data structure" but also "Algorithm"
  - Problem solving techniques are applied to not just software development, but also hardware or system-level design.

System Life Cycle

- **There are five major phases**
  - Requirements
  - Analysis
  - Design
  - Coding
  - Verification

Requirements

- **All Large Programming Projects**
  - **Begin with a set of specifications**
  - **Input**
  - **Output**
  - **Frequently the initial specifications are vague, need rigorous description.**

Question: 討論以下程式的可能輸入方式及方法 (e.g.,鍵盤、檔案、或網路 ?)
(1) 最大組合數計算: finding C(n, m)
(2) 子串搜尋 (e.g., 從網頁搜尋某個字串)
(3) 運算式 (e.g., e = 24 + 8 * 5) 之求值 → 可能需要語意解析程式 (Parser)

Analysis

- **The problem**
  - **Is broken down into manageable pieces**
- **Two are two major approaches**
  - **Top-down partitioning**
  - **Bottom-up integration**

Divide and conquer (D&C)
is an important algorithm design paradigm based on multi-branched recursion.
A divide and conquer algorithm works by recursively
breaking down a problem into two or more sub-problems of the same (or related) type,
until these become simple enough to be solved directly.
The solutions to the sub-problems are then combined
to give a solution to the original problem

Design

- **The programmer designs**
  - Abstract data object
  - Operations on those objects
- **Example**
  - Problem: scheduling system of a university
  - Typical data objects
    - Students, courses, professors
  - Typical operations
    - Inserting, removing, searching
- **So far, the programming**
  - Is language independent

Language Independent

- **For example**
  - The student data object includes
  - (1) Name
  - (2) Social security number
  - (3) permission number
  - (4) major
  - (5) phone number
  - But we haven't decided what is used to implement the list of the students yet

Refinement and Coding

- **First,**
  - **Choose representations for data objects**
  - **Important, since data object may determine the efficiency of the program**
- **Then,**
  - **Write algorithm for each operation**

Verification

- **This phase is to prove correctness**
  - **(1) Testing the problem with a variety of input data**
  - **(2) Debugging until no error exists**
- **Good test data example**
  - **A program with a *switch* statement**
  - **Test data should be chosen so each *case* branch is checked**
- **Debugging practices**
  - **Spaghetti code would be a nightmare when debugging**
  - **Test each unit then whole system**
  - **Documentation is useful**

Outline

- Overview
  - System Life Cycle
- Object-Oriented Software Design
- Data Abstraction and Encapsulation
- Basics of C++
- Algorithm Specification
- Performance Analysis and Measurement

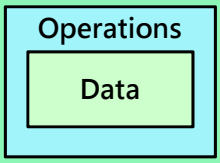
Object-Oriented Programming (OOP)

- **A fundamental change**
  - From the structured programming design method
  - Divide-and-Conquer is still the principle
  - But how a project should be decomposed is different
- **Traditional Programming**
  - Views software as *process*, decomposed into functional modules
- **OOP**
  - Views software as a set of well-defined *objects*
  - These objects interacts with each other to form a software system

(1) 程式 = 運算流程 (Control Flow or Subroutines) +資料結構
(2) 早期的結構化程式以運算流程之設計為主 , 資料結構設計不易重覆使用
(3) 物件導向式語言 :
希望寫程式像堆積木一般 , 而一塊塊的積木是一些容易重覆使用的物件 (Object)
物件 (Object) = 資料結構 (Data) + 一些運算副程式 (Operations)

Definitions of an Object

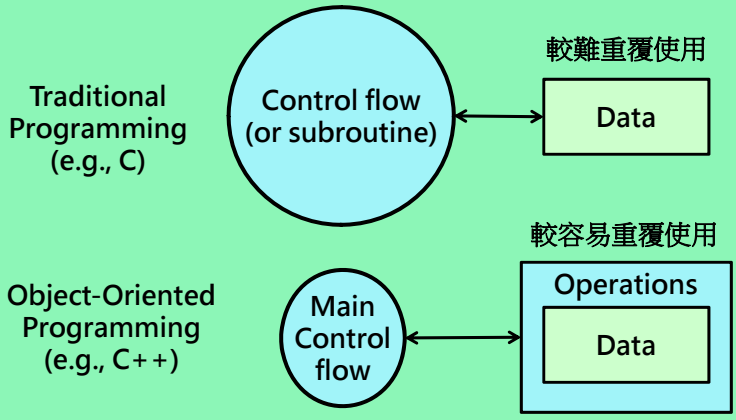
- **An object**
  - Is an entity that performs computations and has a local state, a combination of
  - (1) data
  - (2) procedural elements (or called operations)



- **Object-oriented programming**
  - Is a method of implementation in which
  - (1) objects are the main building blocks
  - (2) each object is an instance of some type (or class)
  - (3) Classes are related to each other by inheritance relationships

資料 (Object) 與資料型態 (Class) 的差別
Example: int *i, j, k*;

OOP – Control Flow vs. Data Structure



Ex: A class of "array" on which you can perform "insert", "retrieve", "delete" and "sort". The elements of the array could be integer, floating point, etc.
➔ "Sorting" is originally part of the control flow, now it is part of the data structure

Object-Oriented Language

- **Three requirements**
  - **(1) It supports objects**
  - **(2) It requires objects to belong to a class**
  - **(3) It supports inheritance**

Comment: True OO programs → use inheritance

Inheritance 的語法範例: *class Stack: public Bag*
→"Bag" 是一個已經定義好的物件類別 , "Stack" 是一個衍生出來的物件類別 ,
→ "Stack" 可以繼承 "Bag" 裡已經有定義的 operations and data

Higher-Level Languages

- **First Generation**
  - **FORTRAN, noted as its ability to evaluate mathematical expressions**
- **Second Generation**
  - **Pascal and C, emphasize on effectively expressing algorithms**
- **Third Generation**
  - **Modula, introduces abstract data types**
- **Fourth Generation**
  - **Object-Oriented Languages, e.g., Smalltalk, Object C, and C++, emphasize inheritance**

Outline

- Overview
  - System Life Cycle
- Object-Oriented Software Design
- Data Abstraction and Encapsulation
- Basics of C++
- Algorithm Specification
- Performance Analysis and Measurement

Data Abstraction & Encapsulation

- Consider a DVD Player
  - (1) The manual tells what the player is supposed to do, instead of how it does it, this is called data abstraction (資料抽象介面 →介面運算的輸出入格式應與內部演算法無關 , 方便日後演算法的更新)
  - (2) The internal representation is hidden from the users, this is called encapsulation (資料包裹性 → 存取不能隨意 , 必須透過介面的 Operations)

Definitions

- **Data Abstraction**
  - Is the separation between the specification of a data object and its implementation
- **Data Encapsulation**
  - Or information hiding is the concealing of the implementation details of a data object from the outside world

Fundamental Data Types of C++

- **Basic Types**
  - char, int, float, double
- **Modification keywords**
  - short, long, signed, unsigned
- **Grouping of Basic Types**
  - array, struct, and class
- **User-Defined Type**
  - *typedef struct _int_pair* {
    int first_num;
    int second_num;
    } int_pair;

使用者自訂複雜資料結構的兩大關鍵字:
struct , *class*
但是 *struct* 所定義的結構不能有資料包裹性
i.e., *class* 才能定義真正的 OOP資料結構

Example Data Type *int*

- **Objects**
  - {0, +1, -1, +2, -2, …, MAXINT, MININT}
- **Operations**
  - + - * /
- **Abstract Data Type (ADT)**
  - A data type organized in a way that the specification is separated from the implementation

Abstract Data Type *NaturalNumber*

ADT *NaturalNumber* is
 objects: An ordered subrange of the integers starting at zero and
 ending at the maximum integer (MAXINT) on the computer
 functions:
 for all *x, y* ∈ *NaturalNumber*; TRUE, FALSE ∈ *Boolean*
 and where +, -, <, ==, and = are the usual integer operations

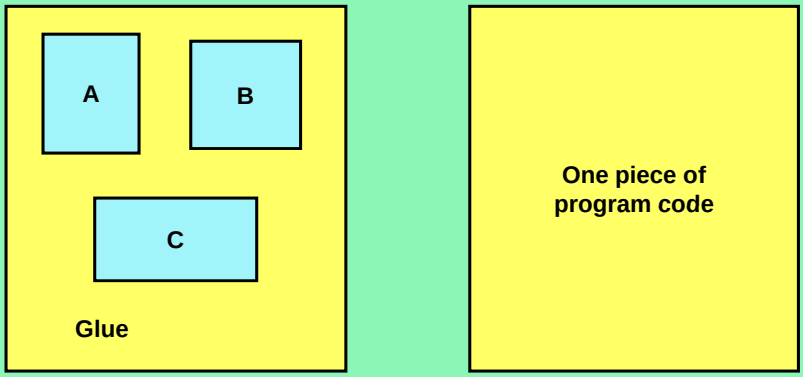
Zero(): *NaturalNumber*	:=	0
IsZero(): *Boolean*	:=	if(*x*==0) *IsZero* = true else *IsZero* = false
Add(*x*, y): *NaturalNumber*	:=	if(*x*+*y*<=MAXINT) *Add* = *x*+*y* else *Add* = MAXINT
Equal(*x*, y): *Boolean*	:=	if(*x*==*y*) *Equal* = true else *Equal* = false
Successor(*x*): *NaturalNumber*	:=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

end *NaturalNumber*

The Advantages of ADT (I)

- **(1) Development Style**

A, B, C are three abstract data types



Good

Not so good !

The Advantages of ADT (II)

- **Testing and Debugging**
  - Programming styles with ADT is easier to debug
  - For example, if every ADT has been tested okay, then only the glue is checked if bugs found during integration
- **Reusability**
  - Data abstraction gives rise to reusability
- **Easier-to-modify**
  - Changes of a data type is localized, I.e., the rest of the program needs not be changed accordingly.

Problem of Not Using Data Encapsulation

- **Consider a program**
  - **That directly accesses internal implementation of the data type**
- **Suppose a change**
  - **Is made to the data type (即直接性資料存取！)**
- **Modification is laborious**
  - **Exhaustive search for instances that access the modified data type and then made appropriate changes – A nightmare !**
  - **(這是直接性資料存取的重大缺點)**

Overhead of ADT

- **Program is slower**
  - **Direct data access versus subroutine invocation**
  - **This is the main reason that C is still in widespread use**
- **Coding is more tedious (但這通常是值得的)**
  - **A lot of simple data-access member functions need to be created**

Outline

- Overview
  - System Life Cycle
- Object-Oriented Software Design
- Data Abstraction and Encapsulation
- Basics of C++
- Algorithm Specification
- Performance Analysis and Measurement

Multiple File Program

- Development Cycle
  - Each source C++ file is compiled individually, producing an object file
    - % g++ –c –I./include file.c –o file.o
  - All object files are linked together, along with other binary library, producing a binary executable file
    - Assume that there is a library file called ./lib/libmylib.a
    - Linking → % g++ file1.o file2.o –L./lib –lmylib –o prog
  - Execute the program
    - % prog <with command line arguments>
    - Example: % prog –A –e 10

% 是作業系統的提示符號

Pre-Processor Directive

- **The Header Files (for 具有許多原始檔案的程式)**
  - Are mostly included at the beginning of each source file
  - Inclusion of a header file for multiple times creates a compilation errors
  - The following pre-processor directives can be used to avoid the above errors

```
#ifndef FILENAME_H
#define  FILENAME_H
// insert contents of the header file here
.
.
.
#endif
```


Utility *Make*

- **Purpose of *Make***
  - To manage the compilation and linking of a large software consisting of multiple files
  - Avoid typing compilation commands repeatedly
- **Procedure of Using *Make***
  - Step1: create a file called "Makefile"
  - Step2: type in "make" each time when any source file or header file is modified. In response to this command, only the files modified will be recompiled, while the others are left intact.

Example of Makefile

```
#---------- define macro names ----------
CC=g++ –g
SOURCE = file1.c file2.c
HEADER = project.h
OBJ = $(SOURCE: .c=.o)

#---------- perform linking when any source file or object file is changed -----
linking: $(SOURCE) $(HEADER) $(OBJ)
<tab>$(CC) $(OBJ) –o ./prog
```


Scope of Variables

- **A Variable is only visible within its scope**
- **Four Types of Scopes**
  - **(1) Global Scope: Variables that are available throughout the entire program**
  - **(2) File Scope: declarations not in a function definition or in a class definition**
  - **(3) Local Scope: Labels used within the function definition**
    (4) Class Scope: Declarations associated with a class definition

Example C++ Program

```
#include <iostream.h>

char course_name[100] = "data structure";

main()
{
  int a = 84;
  printf("Welcome to %s\n", course_name);
  printf("n is %d, n+1 is %d\n", a, add_one(a));
}

Int add_one(int b)
{
   int  c;
   printf("A subroutine for %s\n", course_name);
   c = b + 1;
   return(c);
}
```

A file-scope variable

a is a local variable

b is an input argument

c is a local variable

Global Variables

- **Problem**
  - A global variable defined in file1.C, and to be also used in file2.C
  - → Use *extern* to declare the variable in file2.C

```
#ifdef MAIN  /* macro MAIN is defined in file1.C */
   int  global_variable;
#else
   extern int global_variable /* declare extern in all other files */
#endif
```

Example C++ Program – Global Variable

Source File 1

```
#include <iostream.h>

char course_name[100] = "data structure";

main()
{
  int a = 84;
  printf("Welcome to %s\n", course_name);
  printf("n is %d, n+1 is %d\n", a, add_one(a));
}
```

Source File 2

```
#include <iostream.h>

extern char course_name[100] = "data structure";

Int add_one(int b)
{
   printf("A subroutine for %s\n", course_name);
   return(b+1);
}
```

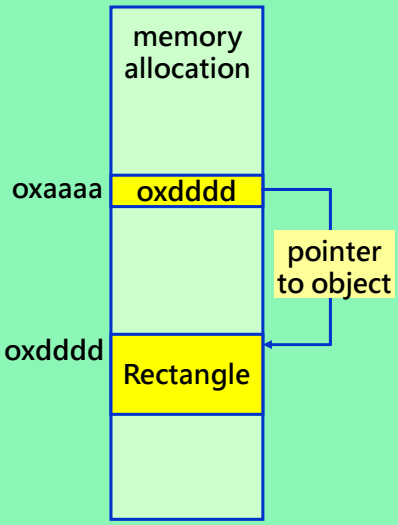

C++ Statement and Operators

- **Dynamic Memory Management**
  - ***"new"*** **and** ***"delete"***
- **Input/Output**
  - **Uses shift left (<<) and shift right (>>) operators**
- **Operator Overloading**
  - **An operator could have multiple functions, depending on the types of operands that it is being applied to**

Data Declaration in C++

- **(1) Constant Value**
- **(2) Variables**
- **(3) Constant Variable**
  - **const int MAX = 500;**
- **(4) Enumeration types**
  - **enum *Boolean* {FALSE, TRUE};**
- **(5) Pointers**
  - **Hold memory addresses of objects**
  - **int *i* = 25;**
  - **int **np;*** — ***np*** **is a pointer to an integer, where * is like "taking content"**
  - ***np* = *&i;*** — ***np*** **points to the location of *i*, where & is like "taking address"**

Object vs. Pointer




```
Rectangle r, s;
Rectangle *t = &s;
```

symbol table

name	type	address
s	Rectangle	oxdddd
t	pointer	oxaaaa

Data Declaration in C++ (con't)

- **(6) Reference types**
  - A unique feature of C++, (which is not available in C)
  - Is a mechanism to provide an alternative name for an object
  - Example

```
int i=5;
int& j=i;
i=7;
printf("i=%d, j=%d", i, j); → both i and j are 7;
```


Outputs in C++

```
#include <iostream.h>

main()
{
  int n=50; float f=20.3;
  cout << "n:" << n << endl;
  cout << "f:" << f << endl;
}

(結果)
n: 50
f: 20.3
```

Inputs in C++

```
#include <iostream.h>

main()
{
  int a, b;
  cin >> a >> b;
}

(結果)
input1:
5 10 <enter>
→ will set a=5; b=10;
```


File IO in C++

```
#include <iostream.h>
#include <fstream.h>

main()
{
  ofstream outFile("my.out", ios::out);
  if(!outFile) {
    cerr << "cannot open my.out" << endl; // standard error device
    return;
  }
  int n=50; float f=20.3;
  outFile << "n: " << n << endl;
  outFile << "f: " << f << endl;
}
```

Functions in C++

- **Two kinds of functions**
  - (1) Regular functions (非附屬於物件內的副程式)
  - (2) Member functions associated with a class
- **A function consists of**
  - Name
  - A list of arguments, also called input signature
  - A return type (output)
  - The body

```
int max(int a, int b)
{
  if(a>b) return a;
  else    return b;
}
```


Parameter Passing in C++

- **(1) Pass by value (傳值呼叫)**
  - Default mechanism
  - When an object is passed by value → it is copied into the function's local storage
  - could be slow when data to be passed is large !
- **(2) Pass by reference (傳地址呼叫)**
  - Done by appending an & to its type specifier
  - E.g., int max(int& a, int& b);
  - When an object is passed by reference → only the address of its location is copied into the function's local store
  - faster but less secure !

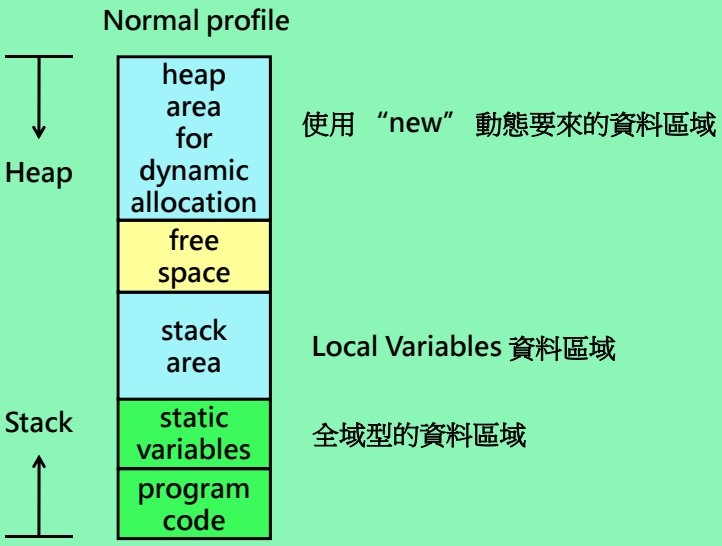
Call By Pointer Example

```
main()
{
  int i, j;
  cout << "Input 2 numbers:" << endl;
  cin >> i >>j;
  if( i > j )
            swap(&i, &j);
            cout << "The smaller number is " << i << endl;
            cout << "The larger is " << j << endl;
};

void swap(int *ptr_x, int *ptr_y)  // call by pointer
{
   int temp;
   temp = *ptr_x;
   *ptr_x = *ptr_y;
   *ptr_y = temp;
}
```

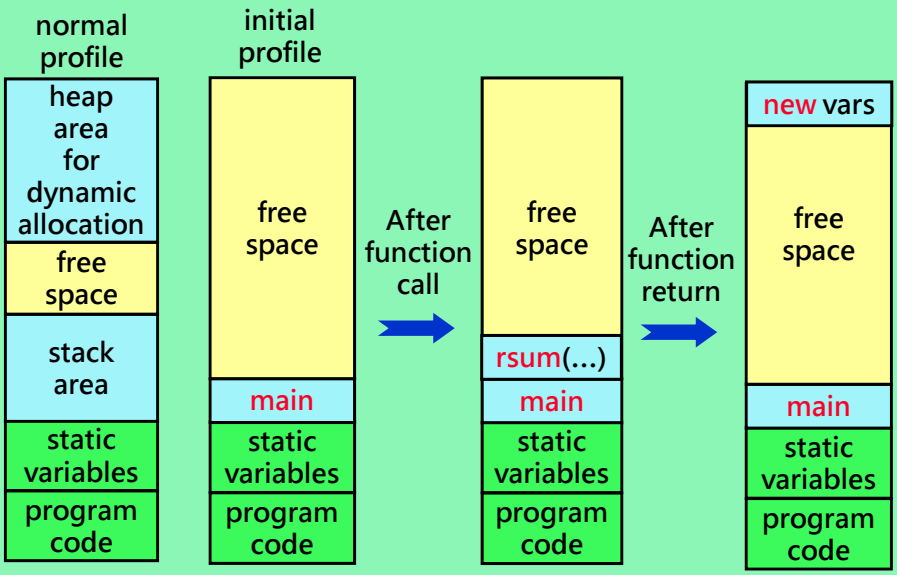

Call By Reference Example

```
main()
{
  int i, j;
  cout << "Input 2 numbers:" << endl;
  cin >> i >>j;
  if( i > j )
            swap(i, j);
            cout << "The smaller number is " << i << endl;
            cout << "The larger is " << j << endl;
};

void swap(int &x, int &y)  // call by reference
{
   int temp;
   temp = x;
   x = y;
   y = temp;
}
```

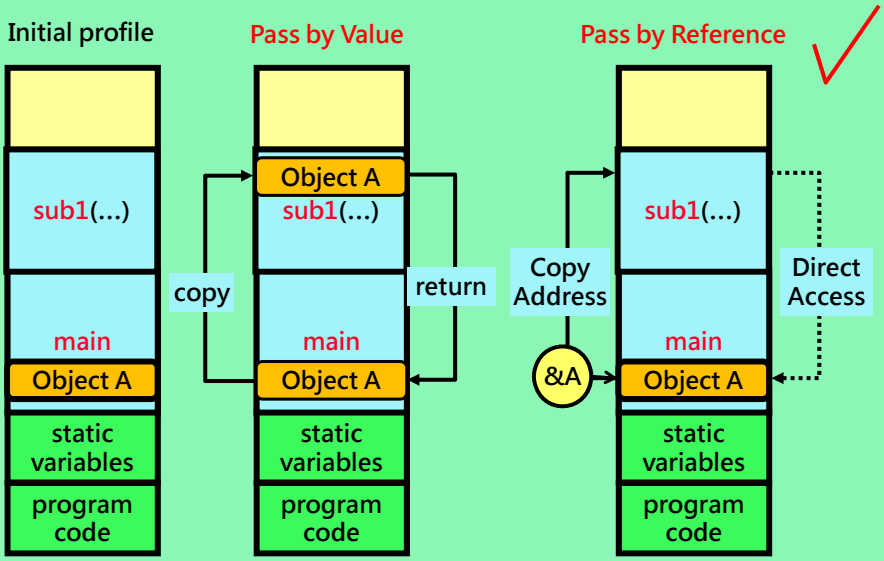
Memory Allocation



Memory Allocation – Subroutine Invocation



Pass-by-Value vs. Pass-by-Reference



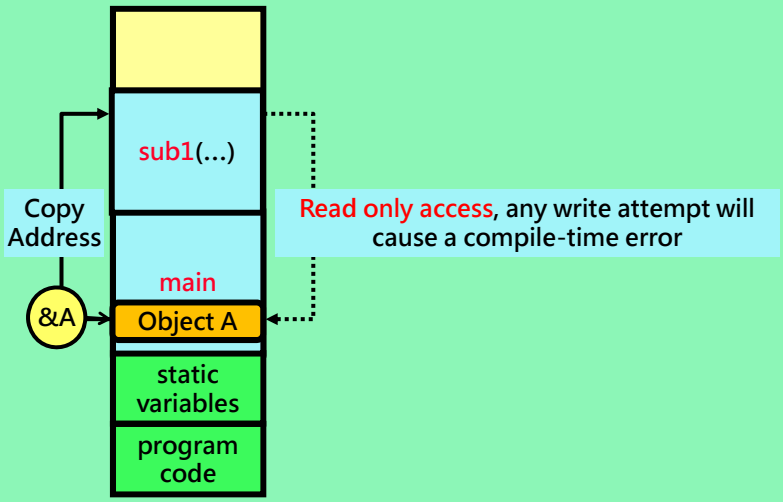
Pass by Const References

- **A Best Method**
  - pass by "**const T& *a***", T is the type of the argument ***a***
  - Faster than pass-by-value if a large chunk of arguments to be passed
  - Better protection of the actual arguments to be passed
  - **Any attempt to modify a *const* argument in the function body will result in a compile-time error**

Improper manipulations of the input arguments
→ could lead to nasty bugs

Illustration: Pass by Const References

Pass by Constant Reference



One Exception

- **Array**
  - **Does not pass by value**
  - **I.e., it is not copied to the function's local store**
  - **Only the pointer of the first element is passed**
  - **Function is not aware of the size of the array**
  - **Often the size of an array is also passed as another argument**

例子: A subroutine that sorts an array of *n* integer elements
Subroutine 結構如下:

```
float sorting(float *a, const int n) {
    // where a is the array name
    …..
}
```

Function Name Overloading

Function over-loading: there can be more than one functions with the same name as long as they have different signatures

```
Int max(int, int);
Int max(int, int, int);
Int max(int*, int);
Int max(float, int);
Int max(int, float);
```


InLine Function

```
Inline int sum(int a, int b)
{
   return (a+b);
}
```

Inline function can eliminate the use of certain preprocessor directives such as #define, which is traditionally used for macro substitution

→ Excessive use of pre-processors make it harder to use debugger or profiler

Dynamic Memory Allocation

- **New**
  - This operator creates an object of the desired type and return a pointer to the data type that follows it.
  - It returns 0 if not being able to create it
- **Delete**
  - Free the data allocated by "new" operator

```
int *ip = new int;
If(ip==0) cerr << "Memory not allocated" << endl
.
.
delete ip;
```


Creating An Array

```
int *jp=new int[10];
if(jp==0) cerr << "Memory not allocated" << endl
.
.
delete [ ] jp;

/* The operator [ ] is used to inform the compiler that
the object being created or deleted is an array
```

Outline

- Overview
  - System Life Cycle
- Object-Oriented Software Design
- Data Abstraction and Encapsulation
- Basics of C++
- Algorithm Specification
- Performance Analysis and Measurement

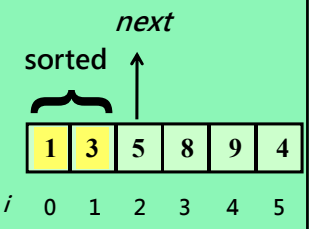
Definition

- Algorithm
  - Is a finite set of instructions that, if followed, accomplishes a particular task.
- Criteria
  - Input
  - Output
  - Definite: each instruction is clear and unambiguous
  - Finiteness: for all cases, the algorithms terminate after a finite number of steps
  - Effectiveness: each instruction must be basic enough

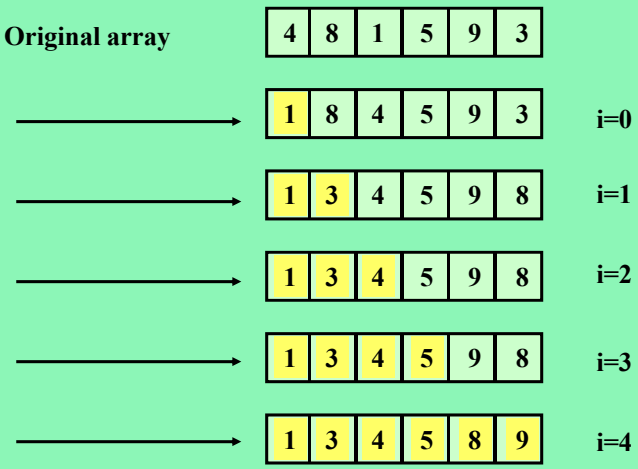
Example: Selection Sort

- **Problem**
  - To sort a collection of n ≧1 integers
- **A Solution**
  - From those integers that are currently unsorted, find the smallest and place it next in the sorted list
- **Selection Sort Algorithm**

```
for(int i=0; i<n; i++) {
  // Fixing the i-th smallest element
  examine a[i] to a[n-1] and suppose the smallest
  integer is at a[j]; // a[j] is the i-th smallest element
  interchange a[i] and a[j];
}
```



Example of Selection Sort



Selection Sort Algorithm

```
1.  void sort (int *a, const int n)
2.  // sort the n integers a[0] to a[n-1] into non-decreasing order
3.  {
4.    for(int i=0; i<n; i++){
5.        // find the smallest integer from a[i] to a[n-1];
6.        int smallest_index = i;
7.        for(int k=i+1; k<n; k++) {
8.            if (a[k] < a[smallest_index]) smallest_index = k;
9.        }
10.       // interchange
11.       int temp=a[i]; a[i]=a[smallest_index];
12.           a[smallest_index ]=temp;
13.   }
14. }
```

The upper limit index of the "for loop" in line 4 can be changed to *n*-1 without damaging the correctness of the algorithm

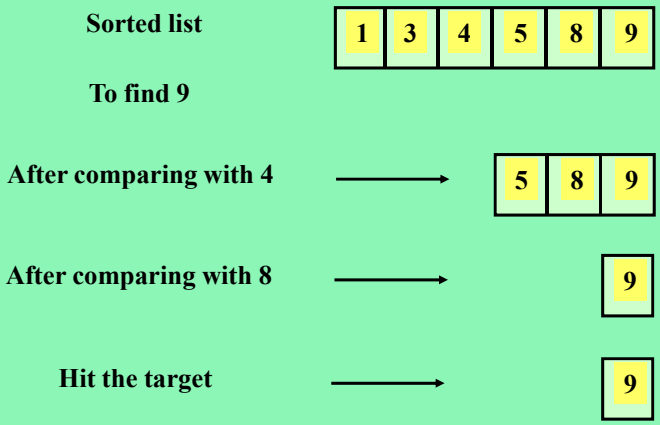
Binary Search

- **Problem**
  - Assume that we have $n \geq 1$ distinct integers that are already sorted in array a[0],…a[*n*-1]
  - Determine if an integer *x* is present, if so, return its index

A sub-routine *compare*

```
1.  char compare(int x, int y)
2.  {
3.    if (x>y) return '>';
4.    else if (x<y) return '<';
5.    else  return '=';
6.  } // end of compare
```

Example of Binary Search



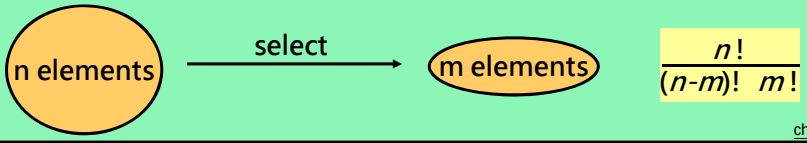
C++ Code for Binary Search

```
1.  binary_search (int *a, const int x, const int n)
2.  // search for the sorted array a[0],…,a[n-1] for x
3.  {
4.    for(int left=0, int right=n-1; left <= right;)
5.    {
6.      middle = (left+right) /2;
7.      switch(compare(x, a[middle])){
8.        case '>': left = middle+1; break;
9.        case '<': right = middle-1; break;
10.       case '=': return middle;
11.     } // end of switch
12.   } // end of for
13.   return –1;
14. } // end of binary search
```

left middle right

Recursive Algorithms

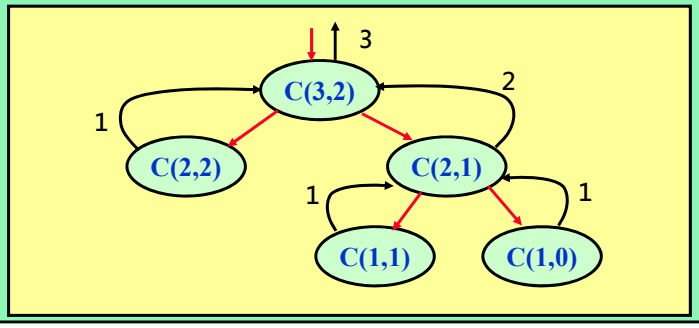
- **Recursion**
  - Is similar to the method of induction which is often used to prove mathematical statements
  - (1) A basis is needed
  - (2) A terminating condition is needed
- **Applications**
  - Recursion is particularly suitable for problem recursively defined
  - E.g., Factorial n!
  - E.g., Binomial coefficient C(n,m) = C(n-1,m) + C(n-1, m-1);



Recursive Binomial Coefficient

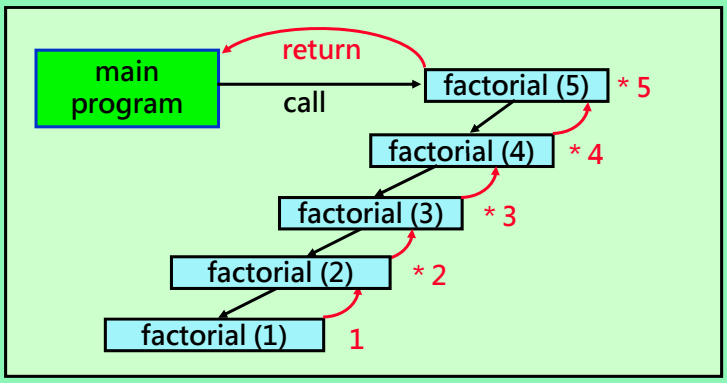
- **C(n, m) = C(n-1, m) + C(n-1, m-1);**

```
int binomial(int n, int m)
{
   if( n < m ) exit(-1);
   if(n==m) return(1);    if(m==0) return(1);
   return( binomial(n-1, m) + binomial(n-1, m-1) ) ;
}
```



Recursive Factorial

```
1.  factorial (int n)
2.  {
3.      if(n==1)   return (1);
4.      else       return( factorial(n-1) * n);
5.  }
```



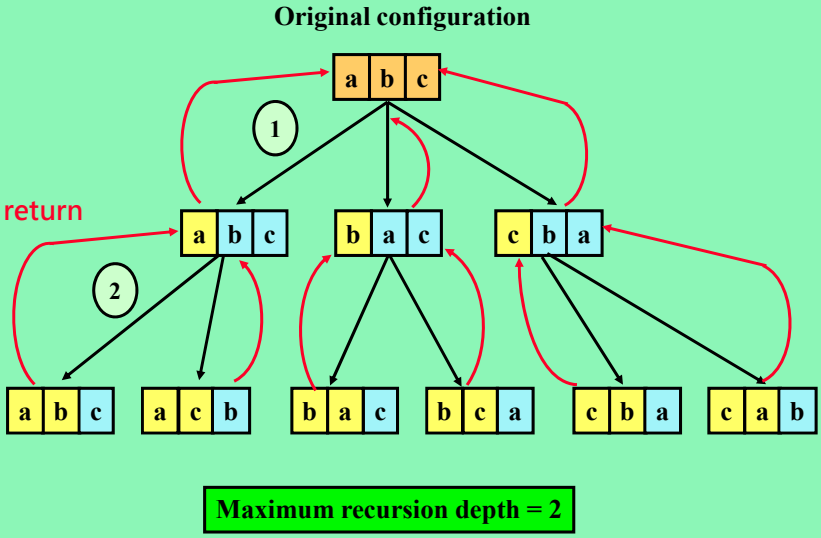
Recursive Binary Search

```
1.  Recursive_BS(int *a, const int x, const int left,
2.                                   const int right)
3.  // search for the sorted array a[left],…,a[right] for x
4.  {
5.    if(left <= right) {
6.       int middle = (left+right) /2;
7.       switch(compare(x, a[middle]){
8.          case '>':
9.              return(Recursive_BS(a, x, middle+1, right));
10.         case '<':
11.             return(Recursive_BS(a, x, left, middle-1));
12.         case '=': return middle;
13.      }
14.   }
15.   return –1;
16. }
```

Permutation

- **Example**
  - A set of symbols {a, b, c}
  - All possible number of permutations is n!
  - {(a, b, c), (a, c, b), (b, a, c), (b, c, a), (c, a, b), (c, b, a)}
- **Recursive Permutation of {a, b, c, d}**
  - {a, permutation of (b, c, d)}
  - {b, permutation of (a, c, d)}
  - {c, permutation of (a, b, d)}
  - {d, permutation of (a, b, c)}

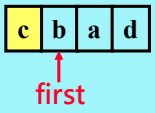
Demo of Recursive Permutation



Maximum recursion depth = 2

Recursive Permutation Generator

```
1.  void perm (char *a, const int first, const int n)
2.  // generate all permutations of a[first],…,a[n-1]
3.  // first - the first element in the undecided region
4.  {
5.    if(first==n-1) {  // terminating condition
6.      for(int i=0; i<n; i++) cout << a[i] << " ";
7.      cout << endl;
8.    }
9.    else {
10.     for (i=first; i<n; i++) {
11.         char temp=a[first]; a[first]=a[i]; a[i]=temp;
12.         perm(a, first+1, n);
13.         temp=a[first]; a[first]=a[i]; a[i]=temp;
14.         // return to original configuration
15.     }
16.   }
17. }
```



Program 1.11

Outline

- Overview
  - System Life Cycle
- Object-Oriented Software Design
- Data Abstraction and Encapsulation
- Basics of C++
- Algorithm Specification
- Performance Analysis and Measurement

Criteria of Judging a Program

1. **Is it functioning?**
2. **Speed (i.e., CPU time)**
3. **Space (i.e., memory requirement)**
4. **Documentation**
5. **Readability**

Complexity

- **Space Complexity**
  - **The amount of memory a program needs to run to complete**
- **Time Complexity**
  - **The amount of computer time a program needs to run to complete**
- **Performance Analysis**
  - **To estimate a program's run time**
- **Performance Measurement**
  - **To actually measure a program's run time**

Space Requirement

- **Fixed Part**
  - Instruction space, space for variables and constants
- **Variable Part**
  - Depends on instance characteristics, and the recursion stack space
  - This part is more important
- **Space requirement of a program P**
  - $S(P) = c + S_P$ (instance characteristics)

    c is a constant and S_P is a function of the problem size

可以簡單的把 instance characteristic 想成 problem size 即可

Example: Space Complexity

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

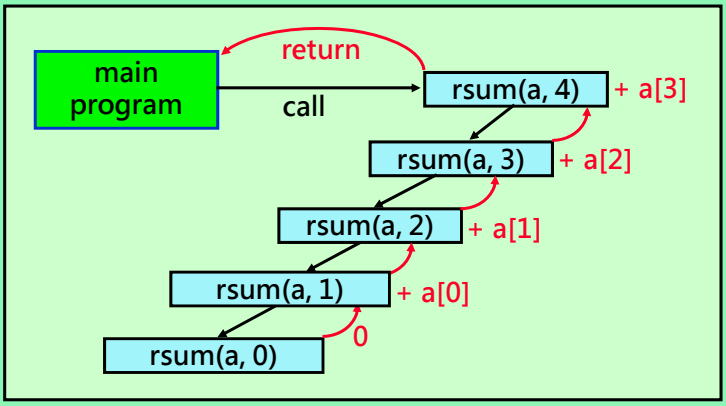
S_P(instance characteristics) = 0;
That is, space is independent of the instance characteristics

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

S_P(instance characteristics) = *n*;

Recursive Summation

```
1.  float rsum(float *a, const int n) {
2.        if(n<=0)  return 0;
3.        else      return(rsum(a, n-1) + a[n-1]);
4.  }
```



Example: Space Complexity

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

(1) Instance characteristics = *n*
(2) Each call to *rsum* requires at least 4 words
space for *a*, *n*, the return value, and the return address
(3) The depth of recursion is *n*+1
(4) The recursion stack space is 4(*n*+1)
(5) For *n* = 1000 → stack space is 4004

Time Complexity

- **Total Time = Compile Time + Run Time**
- **Run Time is of more concern**
  - **t_p(instance characteristics)**
- **A program step**
  - **syntactically or semantically meaningful segment of a program**
- **For example**
  - **return(*a+b+b*c+(a+b-c)/(a+b)*+4.0;) can be regarded as a step**
    → **because it is independent of (instance characteristics)**

Step Counting (僅供參考)

- **(1) Comments: 0**
- **(2) Declarative statements: 0**
  - **int, long, short, char, float, double, const, enum, signed, unsigned, static, extern**
  - **class, struct, union, template**
  - **private, public, protected, friend**
  - **void, virtual**
- **(3) Expression and Assignments: 1**
- **(4) Iteration Statements (for, while, do): <iteration-count>**
- **(5) Switch statements:**
- **(6) If-else statements:**
- **(7) Function invocation: 1**
- **(8) Memory management statements: 1**
- **(9) Jump statements (break, return): 1**

```
for( <init-stmt>; <expr1>; <expr2>)
while <expr> do
do … while <expr>
switch <expr>{
      case cond1: <statement1>
      …
}
```

Example: Step-Counting

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

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


Example: Step Counting For Recursive Program

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

recurrence relation for n>0

$$
\begin{aligned}
t_{rsum}(n) &= 2 + t_{rsum}(n-1) \\
&= 2 + 2 + t_{rsum}(n-2) \\
&= 2 * 2 + t_{rsum}(n-2) \\
&= \dots \\
&= 2n + t_{rsum}(0) \\
&= 2n + 2
\end{aligned}
$$

solved by repeated substitution

Example: Matrix Addition

matrix addition with counting

```
void add (matrix a, matrix b, matrix c, int m, int n)
{
  for (int i = 0; i < m ; i++)
  {
    count++; // for for i
    for (int j = 0; j < n; j++)
    {
      count++; // for for j
      c [i ][j ] = a [i ][j ] + b [i ][j ];
      count++; // for assignment
    }
    count++; // for last time of for j
  }
  count++; // for last time of for i
}
```

Simplified version

```
line  void add (matrix a, matrix b, matrix c, int m, int n)
 1    {
 2      for (int i = 0; i < m; i++)
 3      {
 4        for (int j = 0; j < n ; j++)
 5          count += 2;
 6        count += 2;
 7      }
 8      count++;
 9    }
```


Tabular Method for Iterative SUM

```
1.  float sum(float *a, const int n) {
2.        float s=0;
3.        for(int i=0; i<n; i++){
4.                s += a[i];
5.        } return s;
6.  }
```

line	s/e	frequency	total steps
1	0	1	0
2	1	1	1
3	1	$n+1$	$n+1$
4	1	n	n
5	1	1	1
6	0	1	0
Total number of steps			$2n+3$

s/e: steps per execution

Tabular Method for Recursive SUM

```
1.  float rsum(float *a, const int n) {
2.        if(n<=0)  return 0;
3.        else      return(rsum(a, n-1) + a[n-1]);
4.  }
```

line	s/e	frequency		total steps	
		$n=0$	$n>0$	$n=0$	$n>0$
1	0	1	1	0	0
2(a)	1	1	1	1	1
2(b)	1	1	0	1	0
3	$1+t_{rsum}(n-1)$	0	1	0	$1+t_{rsum}(n-1)$
4	0	1	1	0	0
	Total number of steps			2	$2+t_{rsum}(n-1)$

Tabular Method for Matrix Addition

```
line void add (matrix a, matrix b, matrix c, int m, int n)
 1 {
 2 for (int i = 0; i < m; i++)
 3     for (int j = 0 ; j < n ; j++)
 4         c [i ][j ] = a [i ][j ] + b [i ][j ];
 5 }
```

line	s/e	frequency	total steps
1	0	1	0
2	1	$m+1$	$m+1$
3	1	$m(n+1)$	$mn+m$
4	1	mn	mn
5	0	1	0
	Total number of steps		$2mn+2m+1$

Step Counting of Fibonacci Numbers

```
1  void fibonacci (int n)
2  // compute the Fibonacci number F_n
3  {
4      if (n <= 1)  cout << n << endl; //F_0 = 0 and F_1 = 1
5      else { // compute F_n
6          int fn; int fnm2 = 0; int  fnm1 = 1;
7          for (int i = 2; i <= n; i++)
8          {
9              fn = fnm1 + fnm2 ;
10             fnm2 = fnm1 ;
11             fnm1 = fn ;
12         } // end of for
13         cout << fn << endl;
14     } // end of else
15 } // end of fibonacci
```

$F_0=0$ and $F_1=1$
$F_n = F_{n-1} + F_{n-2}$ for $n \geq 2$

0 1 1 2 3 5 8 13 21 34 55 ...

Program of Fibonacci Sequence Generator

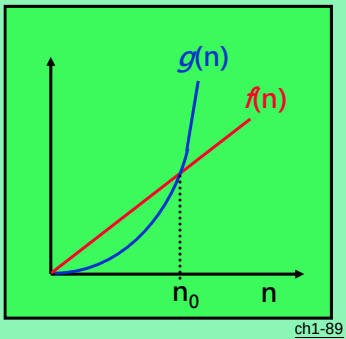
Summary of CPU Time Estimation

- **CPU Time**
  - Is a function of "instance characteristics"
  - Varies as the magnitudes of the inputs increase
- **In BinarySearch**
  - The step count is dependent on the array and '*x*' to be searched
  - Best case, average case, and the worst case are different.

所以我們應該要瞭解的是不同條件下的趨勢，而不只是一個值

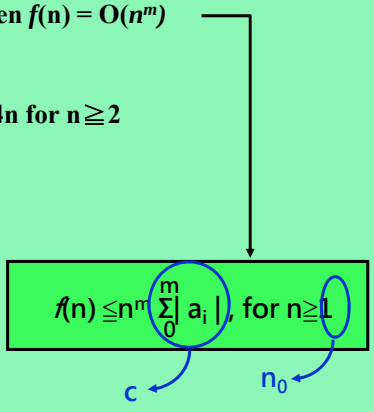
Asymptotic & Big-O Notation

- **Asymptotic Complexity**
  - Concerns about how space or time complexities grow as the size of the problem's inputs grows
- **Big-O Definition**
  - $f(n) = O(g(n))$ iff there exist positive constants c and n_0 such that $f(n) \leqq cg(n)$ for all n, $n \geqq n_0$
  - That is, $g(n)$ is an upper bound of $f(n)$
- **Examples**
  - O(1): constant time computing
  - O(n): linear
  - O(n^2): quadratic
  - O(n^3): cubic
  - O(2^n): exponential
  - O(log n): logarithmic, O(n log n)



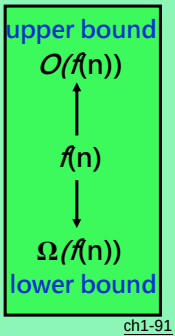
Complexity of Polynomial

- **Theorem 1.2**
  - If $f(n) = a_m n^m + \ldots + a_1 n + a_0$, then $f(n) = O(n^m)$
- **Examples**
  - $3n+2 = O(n)$ → because $3n+2 \leqq 4n$ for $n \geqq 2$
  - $6 \cdot 2^n + n^2 = O(2^n)$
  - $3n+2 \neq O(1)$
  - $10n^2+4n+2 \neq O(n)$



Omega Definition

- **Omega**
  - $f(n) = \Omega(g(n))$ iff there exist positive constants c and n_0 such that $f(n) \geq cg(n)$ for all n, $n \geq n_0$
  - That is, $g(n)$ is a lower bound of $f(n)$
  - There could be multiple lower bounds, but it is often that we choose the tight one
- **Theorem 1.3**
  - If $f(n) = a_m n^m + \ldots + a_1 n + a_0$, $a_m > 0$, then $f(n) = \Omega(n^m)$
- **Examples**
  - $3n+2 = \Omega(n)$ → because $3n+2 \geq 3n$ for $n \geq 2$
  - $6 \cdot 2^n + n^2 = \Omega(n^2)$
  - $10n^2+4n+2 = \Omega(n)$



Theta Definition

- **Theta**
  - $F(n) = \Theta(g(n))$ iff there exist positive constants c_1 and c_2 and n_0 such that $c_1 g(n) \leq f(n) \leq c_2 g(n)$ for all n, $n \geq n_0$
  - That is, $g(n)$ is both a lower bound and upper bound of $f(n)$
- **Theorem 1.4**
  - If $f(n) = a_m n^m + \ldots + a_1 n + a_0$, $a_m > 0$, then $f(n) = \Theta(n^m)$
- **Examples**
  - $3n+2 = \Theta(n)$
  - $6 \cdot 2^n + n^2 = \Theta(2^n)$
  - $3n+2 \neq \Theta(1)$
  - $10n^2+4n+2 \neq \Theta(n)$

Common Recurrence Relation (I)

Reducing problem size by 1 after a constant time

$T(n) = T(n-1) + 1$

$= T(n-2) + 1 + 1$

$= \ldots$

$= T(1) + (n-1)$

$= O(n)$

E.g., iterative summation

Reducing problem size by 1 after a linear time

$T(n) = T(n-1) + n$

$= T(n-2) + n + (n-1)$

$= \ldots$

$= T(1) + O(n^2)$

$= O(n^2)$

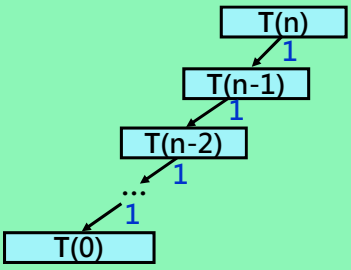
E.g., selection sort algorithm

Common Recurrence Relation (I)

Reducing problem size by 1 after a constant time

$T(n) = T(n-1) + 1$

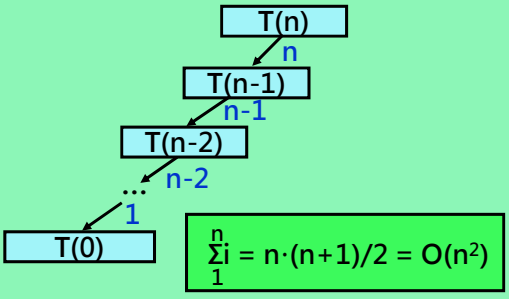
$= O(n)$



Reducing problem size by 1 after a linear time

$T(n) = T(n-1) + n$

$= O(n^2)$



$\sum_{1}^{n} i = n \cdot (n+1)/2 = O(n^2)$

Common Recurrence Relation (II)

Reducing problem size by half after a constant time

$T(n) = T(n/2) + 1$

$= T(n/4) + 1 + 1$

$= \ldots$

$= T(1) + k$

$= O(\log n)$

Assume $n = 2^k$

E.g., Binary search

Reducing problem size by half after a linear time

$T(n) = T(n/2) + n$

$= T(n/4) + n + (n/2)$

$= \ldots$

$= T(1) + (2^k + 2^{k-1} + 2)$

$= O(n)$

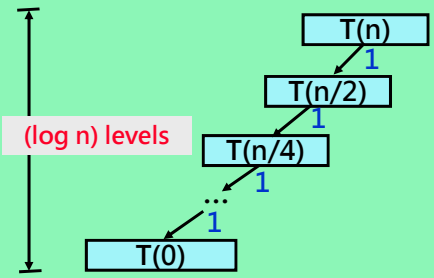
$\sum_{1}^{k} 2^i = (2^{k+1} - 1)$

Common Recurrence Relation (II)

Reducing problem size by half after a constant time

$T(n) = T(n/2) + 1$

$= O(\log n)$

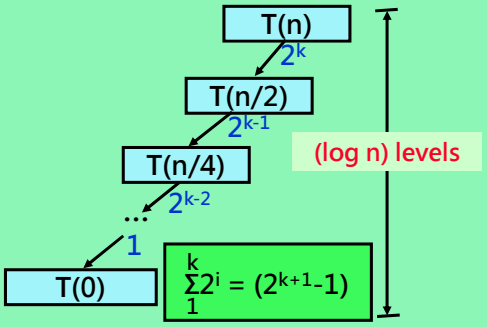


Reducing problem size by half after a linear time

$T(n) = T(n/2) + n$

$= O(n)$

Let $n = 2^k$



Common Recurrence Relation (III)

Split into two equal sub-problems after a constant time

$T(n) = 2T(n/2) + 1$

$= 4T(n/4) + (1 + 2)$

$= \dots$

$= nT(1) + (2+2^{k-1}+2^k)$

$= O(n)$

Split into two equal sub-problems after a linear time

$T(n) = 2T(n/2) + n$

$= 4T(n/4) + n + 2(n/2)$

$= \dots$

$= nT(1) + (n+n+\dots+n)$ (k terms)

$= O(n \cdot \log n)$

Assume $n = 2^k$

Common Recurrence Relation (III)

Split into two equal sub-problem after a constant time

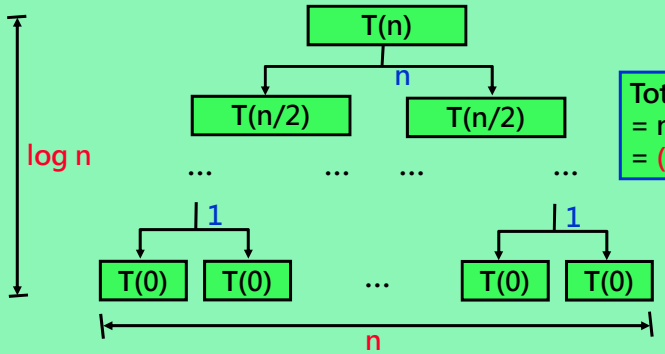
$T(n) = 2T(n/2) + 1$

$= O(n)$

Split into two equal sub-problems after a linear time

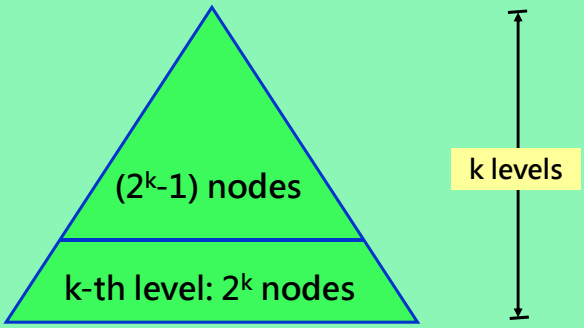
$T(n) = 2T(n/2) + n$

$= O(n \cdot \log n)$



Total number of steps
= n · (#levels)
= (n · log n)

Property of Binary Tree



Total number of nodes in the sub-tree
$= 1 + 2 + 2^2 + \ldots + 2^{k-1}$
$= (2^k-1) / (2-1)$
→ has one node smaller than the last level

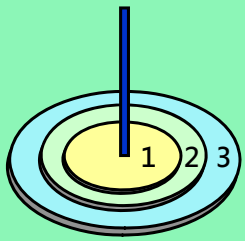
Comparison of Recurrence Relation

- $T(n) = T(n-1) + 1 = O(n)$
- $T(n) = T(n-1) + n = O(n^2)$
- $T(n) = T(n/2) + 1 = O(\log n)$
- $T(n) = T(n/2) + n = O(n)$
- $T(n) = 2T(n/2) + 1 = O(n)$
- $T(n) = 2T(n/2) + n = O(n \cdot \log n)$

這裡每一個公式代表了一種
重要解決問題的思考模式!

Exercise: What is the complexity of a recurrence relation $T(n) = 2T(n-1) + 1$?

Hanoi Towers

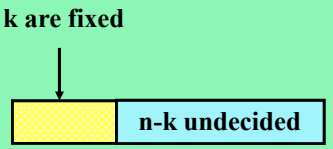


Tower 1 Tower 2 Tower 3

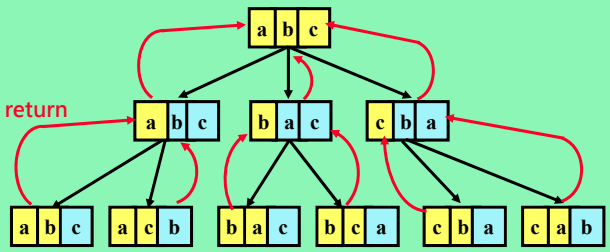
Goal: Move the three disks from Tower 1 to Tower 3
Rules:
(1) One disk can be moved at a time
(2) No disk can be placed on top of a disk with a smaller diameter
Complexity: T(n) = 2T(n-1) + 1 ➔ T(n) = O(2^n)

Asymptotic Complexity Of Permutation Generator

$$T(0, n) = (n) \cdot T(1,n)$$
$$= (n)(n-1) \cdot T(2,n)$$
$$= \ldots$$
$$= (n)(n-1)2 \cdot T(n-1,n)$$
$$= (n)(n-1)2 \cdot O(n)$$
$$= O(n! \cdot n)$$

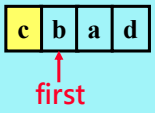


n is the total number elements
k is the number of positions fixed



Recursive Permutation Generator

```
1.  void perm (char *a, const int first, const int n)
2.  // generate all permutations of a[first],…,a[n-1]
3.  // first - the first element in the undecided region
4.  {
5.    if(first==n-1) {  // terminating condition
6.      for(int i=0; i<n; i++) cout << a[i] << " ";
7.      cout << endl;
8.    }
9.    else {
10.     for (i=first; i<n; i++) {
11.       char temp=a[first]; a[first]=a[i]; a[i]=temp;
12.       perm(a, first+1, n);
13.       temp=a[first]; a[first]=a[i]; a[i]=temp;
14.       // return to original configuration
15.     }
16.   }
17. }
```



Program 1.11

Magic Square

- **A magic square**
  - Is an $n \times n$ matrix of the integers 1 to n^2 such that the sum of every row, column, and diagonal is the same

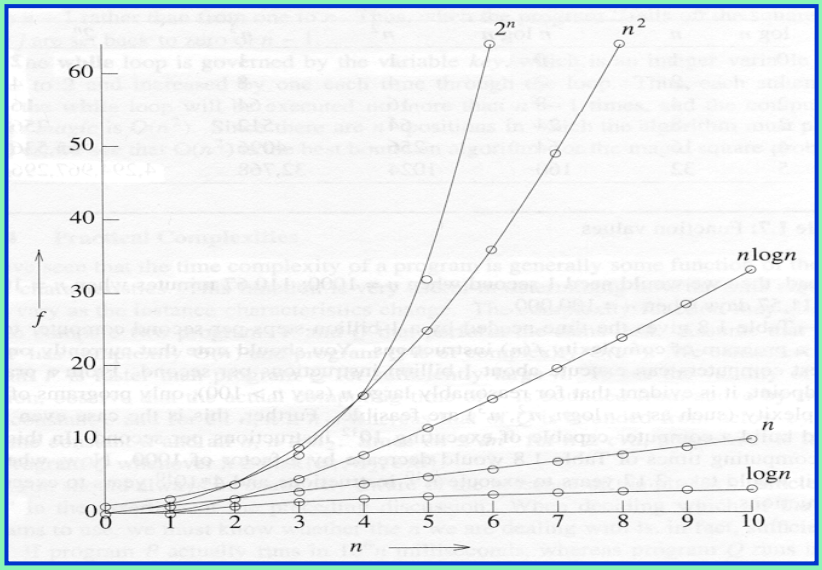
15	8	1	24	17
16	14	7	5	23
22	20	13	6	4
3	21	19	12	10
9	2	25	18	11

Time complexity of magic square = $O(n^2)$

Practical Complexity

log n	n	n·log n	n^2	n^3	2^n
0	1	0	1	1	2
1	2	2	4	8	4
2	4	8	16	64	16
3	8	24	64	512	256
4	16	64	256	4096	65536
5	32	160	1024	32768	4294967296

Comparison of Different Complexities



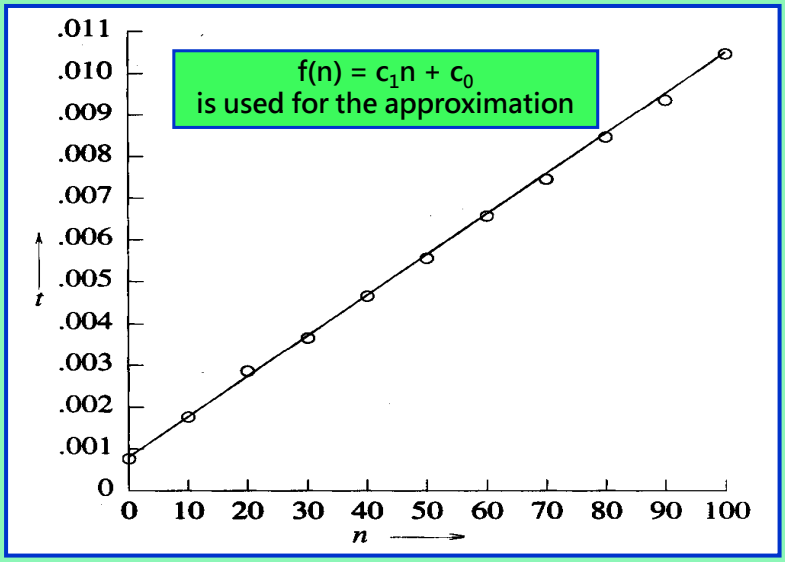
Performance Measurement

- **Performance measurement**
  - is concerned about the actual time and space requirements of a program
  - related to compiler and computer
- **Asymptotic analysis**
  - only tells us the behavior for "sufficiently large" values of *n*
- **Actual time**
  - may not lie exactly on the predicted curve because of the effects of low-order terms that are discarded in the asymptotic analysis

Precision of Measurement Clock

- **Time() command**
  - has a clock precision of only 1/100 second
- **To time a short event**
  - it is necessary to repeat it several times and divide the total time by the number of repetitions
- **Random data**
  - is also commonly used as inputs for average time measurement
- **Measurement**
  - could be for (1) comparison, or for (2) prediction
  - Least-square approximation could be used if the asymptotic complexity is known, e.g., $(a_0 + a_1 n + a_2 n \log n)$ is used to approximate a program with O($n \log n$) complexity

Approximation



Ex1: Measure The CPU Time

```
type_define struct _time_buffer {
    long utime;  long stime;  long cutime;  long cstime;
} time_buffer;

main(){
    time_buffer  T;
    float        start, stop, cpu_time;

    /*------ (1) record the start time --------*/
    times( & T );   start = (float) T.utime; /* a tick is 0.01 second */
    /*------ (2) perform operations to be measured ------*/
    target_function();
    /*------ (3) record the stop time -----*/
    times(&T);   stop = (float) T.utime;
    /*------ (4) measure the elapsed time ------*/
    cpu_time = (stop – start)/100.0;
}
```

Ex2: Measure The CPU Time

```
#include <stdlib.h>
#include <iostream.h>
#include <time.h>       //使用clock()函數 (測試程式目前執行時間)
                        //函數原型  clock_t clock(void)
int main(void)
{
      clock_t start,stop;       int n;

      cout<<endl<<"輸入一整數(10-20之間才不用等太久)"<<endl;
      cin>>n;

      start = clock();      //紀錄開始計算時間
      for(int i=0;i<1000000*n*n;i++) { long s;    s=s+i; }
      stop = clock();       //紀錄計算結束時間
      double usetime;
      usetime=((double)stop-(double)start)/CLK_TCK;  //除以CLK_TCK=1000把單位
      換成秒
      cout << endl << "usetime=“ << usetime << "sec.“ << endl;
      return 0;
}
```


Standard Template Library (STL)

The C++ STL (Standard Template Library) is a generic collection of class templates and algorithms that allow programmers to easily implement standard data structures like queues, lists and stacks.

(Informative web sites about programming in C++ using STL)

C++ reference 網頁: http://www.cppreference.com/wiki/start

STL 網頁: http://www.cppreference.com/wiki/stl/start

The End of
Chapter 1: Basic Concepts !

Next Topic: Arrays