Chapter 1 Basic Concepts

Overview: System Life Cycle

Algorithm Specification

Data Abstraction

Performance Analysis

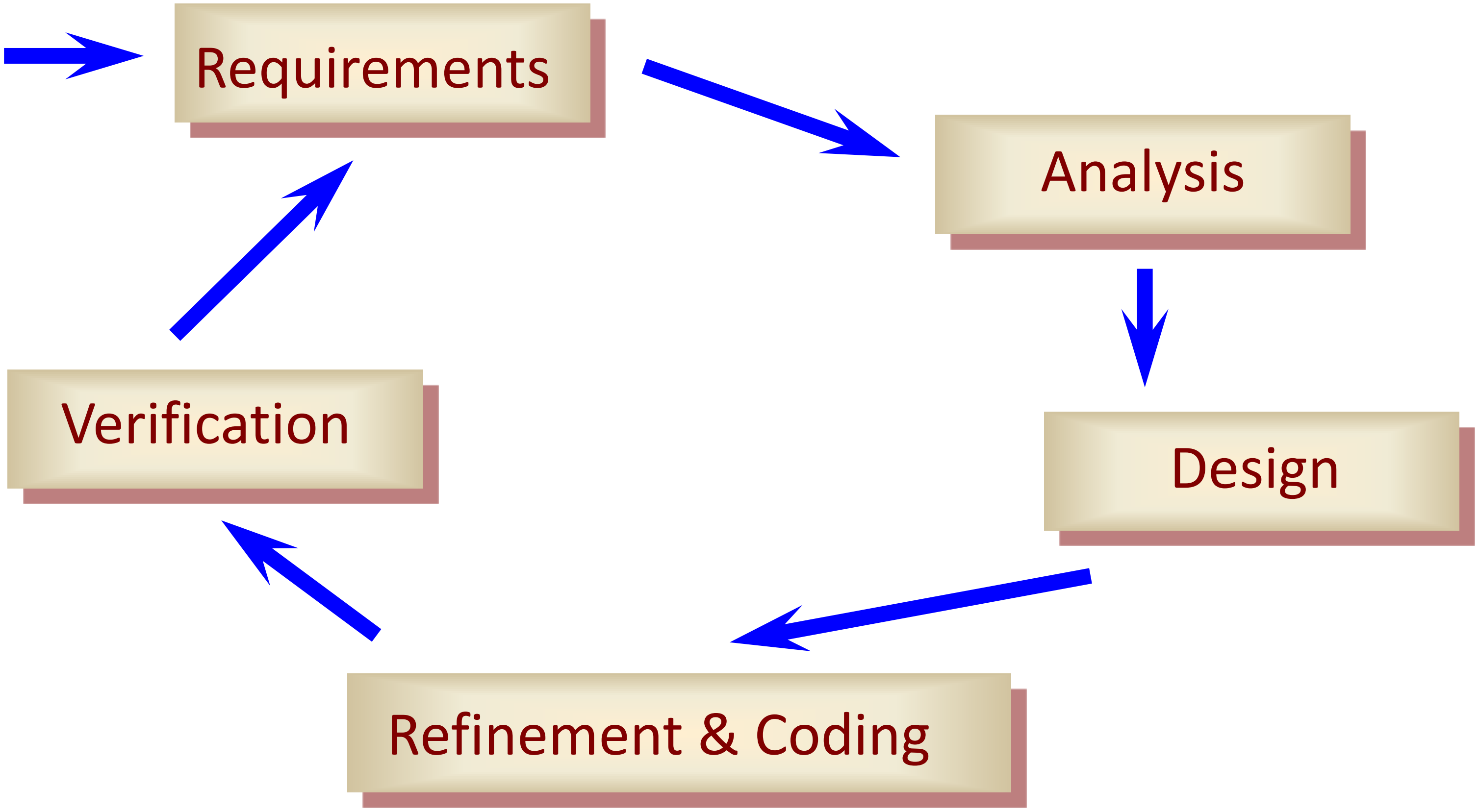
Performance Measurement

Data Structures

- What is the "Data Structure" ?
  - Ways to represent data
- Why data structure ?
  - To design and implement large-scale computer system
  - Have proven correct algorithms
  - The art of programming
- How to master in data structure ?
  - practice, discuss, and think

System Life Cycle

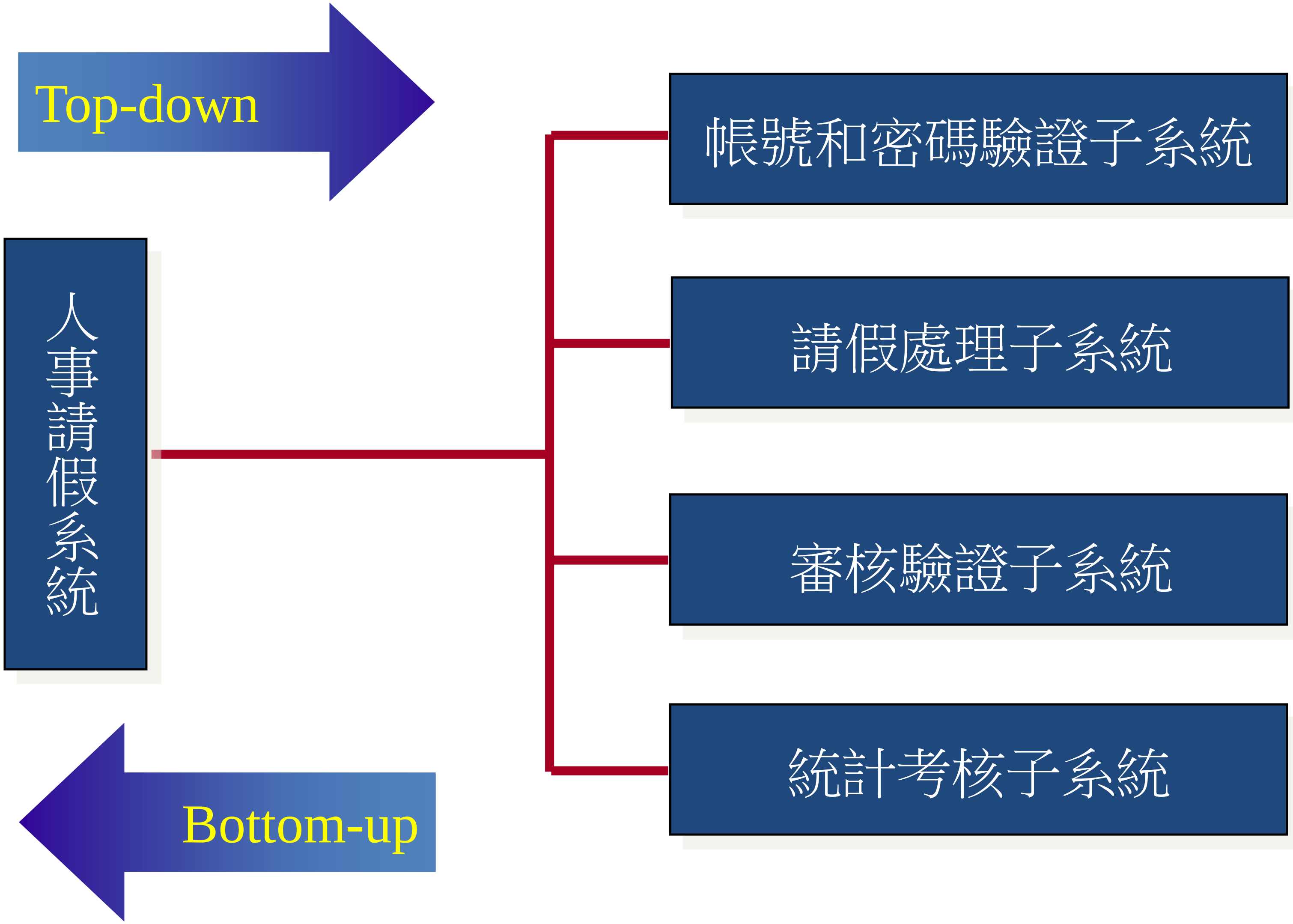
- Summary
  - R A D R C V



System Life Cycle (Cont.)

- Summary
  - R A D R C V
- **Requirements**
  - What inputs, functions, and outputs
- **Analysis**
  - Break the problem down into manageable pieces
  - Top-down approach
  - Bottom-up approach

Example


Top-down
人事請假系統
帳號和密碼驗證子系統
請假處理子系統
審核驗證子系統
統計考核子系統
Bottom-up

System Life Cycle (Cont.)

- **Design**
  - Create **abstract data types** and the **algorithm specifications**

    language independent

- **Refinement and Coding**
  - Determining data structures and algorithms
- **Verification**
  - Developing correctness proofs, testing the program, and removing errors

Verification

- **Correctness proofs**
  - Prove program **mathematically**
    - time-consuming and difficult to develop for large system
- **Testing**
  - Verify that every piece of code runs correctly
    - provide data including all possible scenarios
- **Error removal**
  - Guarantee no new errors generated

Notes:

- Select a proven correct algorithm is important
- Initial tests focus on verifying that a program runs correctly, then reduce the running time

Chapter 1 Basic Concepts

- Overview: System Life Cycle
- Algorithm Specification
- Data Abstraction
- Performance Analysis
- Performance Measurement

Algorithm Specification

- Definition
  - An ***algorithm*** is a finite set of instructions that, if followed, accomplishes a particular task. In addition, all algorithms must satisfy the following criteria:

(1) ***Input***. There are zero or more quantities that are externally supplied.

(2) ***Output***. At least one quantity is produced.

(3) ***Definiteness***. Each instruction is clear and unambiguous.

(4) ***Finiteness***. If we trace out the instructions of an algorithm, then for all cases, the algorithm terminates after a finite number of steps.

(5) ***Effectiveness***. Every instruction must be basic enough to be carried out, in principle, by a person using only pencil and paper. It is not enough that each operation be definite as in (3); it also must be feasible.

Describing Algorithms

- Natural language
  - English, Chinese
    - Instructions must be definite and effectiveness
- **Graphic representation**
  - Flowchart
    - work well only if the algorithm is small and simple
- **Pseudo language**
  - Readable
  - Instructions must be definite and effectiveness

In this text: ***Combining English and C++***

Example

Task：設計一個演算法來測試一個正數 n 是否為質數。

Algorithm：逐一檢查 2, 3, ..., n-1 是否可以整除 n；若都無法整除，則 n 是質數，否則不是質數。

範例：91 是否為質數？

可整除91

~~2~~, ~~3~~, ~~4~~, ~~5~~, ~~6~~, 7, 8, ……, 76

範例：7 是否為質數？

~~2~~, ~~3~~, ~~4~~, ~~5~~, ~~6~~

Example

1. 若 n 小於或等於1，則 n 不是質數；
2. 令 k = 2, 3, ……, n-1，逐一檢驗：
3. 若 k 可以整除 n，則 n 不是質數；
4. 若以上所有的 k 值均無法整除 n，則 n 是質數；

Input: 一個自然數 n。

Output: 回答n 是/否為質數: **Yes No**

Definiteness : 每一行指令都很明確。

Finiteness : 對任一個輸入的自然數 n，此演算法都能在有限的時間內求出 n 是否為質數。

Effectiveness: 每一行指令都簡易至光用紙筆即可做出的程度。

Example (Selection Sort)

From those integers that are currently unsorted, find the smallest and place it next in the sorted list.

i	[0]	[1]	[2]	[3]	[4]
-	30	10	50	40	20
0	10	30	50	40	20
1	10	20	40	50	30
2	10	20	30	40	50
3	10	20	30	40	50

```
for (i = 0; i < n; i++) {
  Examine list[i] to list[n-1] and suppose that the
  smallest integer is  at list[min];

  Interchange list[i] and list[min];
}
```

Program 1.1: Selection sort algorithm

Example (Selection Sort)

- A complete selection sort program which you may run on your computer

```c
#include <stdio.h>
#include <math.h>
#define MAX_SIZE 101
#define SWAP(x,y,t) ((t) = (x), (x)= (y), (y) = (t))
void sort(int [],int); /*selection sort */
void main(void)
{
   int i,n;
   int list[MAX_SIZE];
   printf("Enter the number of numbers to generate: ");
   scanf("%d",&n);
   if( n < 1 || n > MAX_SIZE) {
     fprintf(stderr, "Improper value of n\n");
     exit(1);
   }
   for (i = 0; i < n; i++) {/*randomly generate numbers*/
      list[i] = rand() % 1000;
      printf("%d  ",list[i]);
   }
   sort(list,n);
   printf("\n Sorted array:\n ");
   for (i = 0; i < n; i++) /* print out sorted numbers */
      printf("%d  ",list[i]);
   printf("\n");
}
void sort(int list[],int n)
{
   int i, j, min, temp;
   for (i = 0; i < n-1; i++)  {
      min = i;
      for (j = i+1; j < n; j++)
         if (list[j] < list[min])
            min = j;
      SWAP(list[i],list[min],temp);
   }
}
```

Program 1.3: Selection sort

Example (Selection Sort)

```
for (i = 0; i < n; i++) {
  Examine list[i] to list[n-1] and suppose that the
  smallest integer is  at list[min];

  Interchange list[i] and list[min];
}
```

Program 1.1: Selection sort algorithm

Translating a Problem into an Algorithm (3 Steps)

- Problem
  - Devise a program that sorts a set of $n \geq 1$ integers
- Step I - Concept
  - From those integers that are currently unsorted, find the smallest and place it next in the sorted list
- Step II - Algorithm

```
for ( int i = 0; i < n ; i++)
{
    檢查 list[i]到 list[n-1]並且假設最小的整數是在 list[j] ;
    交換 list[i]和 list[j] ;
}
```

Translating a Problem into an Algorithm(Cont.)

- Step III - Coding

```
void sort ( int *a, const int n)
{ //把 a[0]至 a[n-1]總共 n 個數以遞增的順序排列
    for (int i = 0 ; i < n ; i ++)
    {
        int j = i ;
        //找出 a[i]到 a[n-1]中最小的一個整數
        for (int k = i + 1 ; k < n ; k++)
            if (a[k] < a[j]) j = k;
        swap(a[i], a[j]) ;
    }
}
```

Correctness Proof

- **Theorem**
  - Function *sort(a, n)* correctly sorts a set of $n \geq 1$ integers. The result remains in $a[0], \ldots, a[n-1]$ such that $a[0] \leq a[1] \leq \cdots \leq a[n-1]$.

Proof:

For $\mathrm{i} = q$, following the execution of line 6-11, we have

$$\mathrm{a}[q] \leq \mathrm{a}[r], q < r \leq n-1.$$

For $\mathrm{i} > q$, observing, $a[0], \ldots, a[q]$ are unchanged.

Hence, increasing i, for $i = n-2$, we have

$$a[0] \leq a[1] \leq \cdots \leq a[n-1].$$

Example (Binary Search)

- Binary Search: Searching a sorted list

```
int BinarySearch (int *a, const int x, const int n)
{ //  在排序好的陣列 a[0], …, a[n-1]中找出 x
      初始化 left 和 right ;
      while （還有元素）
      {
            令 middle 為中間的元素 ;
            if (x < a[middle]) 把 right 設定成 middle-1 ;
            else if (x > a[middle]) 把 left 設定成 middle+1 ;
            else return middle ;
      }
      沒找到 ;
}
```

Example (Binary Search)

- Binary Search: Searching a sorted list

[0]	[1]	[2]	[3]	[4]	[5]	[6]
8	14	26	30	43	50	52

left	right	middle	list[middle]	:	searchnum
0	6	3	30	<	43
4	6	5	50	>	43
4	4	4	43	==	43
0	6	3	30	>	18
0	2	1	14	<	18
2	2	2	26	>	18
2	1	-			

Example (Binary Search)

- A complete binary search program which you may run on your computer

```
int BinarySearch (int *a, const int x, const int n)
{ //  在排序好的陣列 a[0], …, a[n-1]中找出 x
    int left = 0, right = n-1 ;
    while (left <= right)
    { //  還有元素
        int middle = (left + right)/2;
        if (x < a[middle]) right=middle-1 ;
        else if (x > a[middle]) left = middle+1 ;
        else return middle ;
    } // while 迴圈結束
    return -1; //  沒找到
}
```

Recursive Algorithms

- **Direct recursion**
  - Functions call themselves
- **Indirect recursion**
  - Functions call other functions that invoke the calling function again
- When is recursion an appropriate mechanism?
  - The problem itself is defined recursively
  - Statements: if-else and while can be written recursively
  - Art of programming
- Why recursive algorithms ?
  - Powerful, express an complex process very clearly

Recursive Implementation of Binary Search

```
int binsearch(int list[], int searchnum, int left,
                                          int right)
{
/* search list[0] <= list[1] <=   · · ·   <= list[n-1] for
searchnum. Return its position if found. Otherwise
return -1 */
   int middle;
   if (left <= right) {
      middle = (left + right)/2;
      switch (COMPARE(list[middle], searchnum)) {
         case -1: return
           binsearch(list, searchnum, middle + 1, right);
         case 0 : return middle;
         case 1 : return
           binsearch(list, searchnum, left, middle - 1);
      }
   }
   return -1;
}
```

Program 1.7: Recursive implementation of binary search

Example (*Permutations*)

- Problem: Given a set of $n \geq 1$ elements, the problem is to print all possible permutations of the set.
- Concept: permutations of (a, b, c, d) can be constructed by writing
  - 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, c, c)

Example (*Permutations*)

```
void perm(char *list, int i, int n)
/* generate all the permutations of list[i] to list[n] */
{
   int j, temp;
   if (i == n) {
      for (j = 0; j <= n; j++)
         printf("%c", list[j]);
      printf("    ");
   }
   else {
   /* list[i] to list[n] has more than one permutation,
   generate these recursively */
      for (j = i; j <= n; j++) {
         SWAP(list[i],list[j],temp);
         perm(list,i+1,n);
         SWAP(list[i],list[j],temp);
      }
   }
}
```

Program 1.8: Recursive permutation generator

```
lv0 perm: i=0, n=2 abc
lv0 SWAP: i=0, j=0 abc
lv1 perm: i=1, n=2 abc
lv1 SWAP: i=1, j=1 abc
lv2 perm: i=2, n=2 abc
print: abc
lv1 SWAP: i=1, j=1 abc
lv1 SWAP: i=1, j=2 abc
lv2 perm: i=2, n=2 acb
print: acb
lv1 SWAP: i=1, j=2 acb
lv0 SWAP: i=0, j=0 abc
lv0 SWAP: i=0, j=1 abc
lv1 perm: i=1, n=2 bac
lv1 SWAP: i=1, j=1 bac
lv2 perm: i=2, n=2 bac
print: bac
lv1 SWAP: i=1, j=1 bac
lv1 SWAP: i=1, j=2 bac
lv2 perm: i=2, n=2 bca
print: bca
lv1 SWAP: i=1, j=2 bca
lv0 SWAP: i=0, j=1 bac
lv0 SWAP: i=0, j=2 abc
lv1 perm: i=1, n=2 cba
lv1 SWAP: i=1, j=1 cba
lv2 perm: i=2, n=2 cba
print: cba
lv1 SWAP: i=1, j=1 cba
lv1 SWAP: i=1, j=2 cba
lv2 perm: i=2, n=2 cab
print: cab
lv1 SWAP: i=1, j=2 cab
lv0 SWAP: i=0, j=2 cba
```

Chapter 1 Basic Concepts

- Overview: System Life Cycle
- Algorithm Specification
- Data Abstraction
- Performance Analysis
- Performance Measurement

Data Abstraction

- Data Types
  A *data type* is a collection of ***objects*** and a set of ***operations*** that act on those objects.
  - A ***data type*** is a collection of ***objects*** and a set of ***operations*** that act on those objects
    - ***Operation:*** Its ***name***, possible ***arguments*** and ***results*** must be specified
  - All programming language provide at least minimal set of predefined data type, plus user defined types

Data Abstraction

- Example of "int"
  - Objects: $0, +1, -1, \ldots, Int\_Max, Int\_Min$
  - Operations: *arithmetic*(**+**, **-**, *, /, and %), *testing* (equality **==** / inequality !=), *assigns* =, *functions*
- The data types of C
  - The basic data types: **char**, **int**, **float** and **double**
  - The group data types: array and **struct**
  - The pointer data type
  - The user-defined types

Abstract Data Type

- Definition
  - An ***abstract data type*** (***ADT***) is a ***data type*** that is organized in such a way that the specification of the objects and the specification of the operations on the objects is separated from the representation of the objects and the implementation of the operation.
  - We know what is does, but not necessarily how it will do it.
- Why abstract data type ?
  - implementation-independent

Operation specification

- function name
- the types of arguments
- the type of the results
- description of what the function does

Classifying the Functions of a Data Type

- **Creator/constructor:**
  - Create a new instance of the designated type
- **Transformers**
  - Also create an instance of the designated type by using one or more other instances
- **Observers/reporters**
  - Provide information about an instance of the type, but they do not change the instance

Notes: An ADT definition will include at least one function from each of these three categories

Example (ADT Natural_Number）

structure *Natural–Number* is

objects: an ordered subrange of the integers starting at zero and ending at the maximum integer (*INT–MAX*) on the computer

functions:

for all $x, y \in$ *Nat–Number*; *TRUE*, *FALSE* $\in$ *Boolean*
and where +, −, <, and == are the usual integer operations

Nat–No Zero( )	::=	0
Boolean Is–Zero(x)	::=	**if** (x) **return** *FALSE*
		else return *TRUE*
Nat–No Add(x, y)	::=	**if** (($x + y$) <= *INT–MAX*) **return** $x + y$
		else return *INT–MAX*
Boolean Equal(x, y)	::=	**if** ($x == y$) **return** *TRUE*
		else return *FALSE*
Nat–No Successor(x)	::=	**if** ($x ==$ *INT–MAX*) **return** x
		else return $x + 1$
Nat–No Subtract(x, y)	::=	**if** ($x < y$) **return** 0
		else return $x - y$

end *Natural–Number*

Structure 1.1: Abstract data type *Natural–Number*

Chapter 1 Basic Concepts

- Overview: System Life Cycle
- Algorithm Specification
- Data Abstraction
- Performance Analysis
- Performance Measurement

Performance Analysis

- Performance evaluation
  - Performance **analysis**
  - Performance **measurement**
- Performance **analysis - prior**
  - an important branch of CS, ***complexity theory***
  - estimate ***time*** and ***space***
  - machine independent
- Performance **measurement -posterior**
  - The actual ***time*** and ***space*** requirements
  - machine dependent

Performance Analysis

- Evaluate a program generally
  - Does the program ***meet*** the original ***specifications*** of the task?
  - Does it ***work correctly***?
  - Does the program contain ***documentation*** that show ***how to use it*** and ***how it works***?
  - Does the program ***effectively use functions*** to create logical units?
  - Is the program's code ***readable***?

Performance Analysis(Cont.)

- Evaluate a program
  - MWGWRERE
  - Meet specifications, Work correctly,
  - Good user-interface, Well-documentation,
  - Readable, Effectively use functions,
  - Running time acceptable, Efficiently use space
- How to achieve them?
  - Good programming style, experience, and practice
  - Discuss and think

Performance Analysis(Cont.)

- Space and time
  - Does the program efficiently use primary and secondary storage?
  - Is the program's running time acceptable for the task?
- Space complexity: storage requirement
- Time complexity: computing time
- Goal: 找出執行時間/使用空間”如何”隨著input size變長
- 什麼是input size? No. of input elements, e.g., array size, width/height of a matrix, …

Space Complexity

- Definition
  - The ***space complexity*** of a program is the amount of memory that it needs to run to completion
- The space needed is the sum of
  - ***Fixed*** space and ***Variable*** space
- Fixed space (***c***)
  - Includes the instructions, variables, and constants
  - Independent of **the number and size of I/O**
- Variable space ($S_P(I)$)

I: input instance (某一個input)

  - Includes dynamic allocation, functions' recursion
- Total space of any program

$$S(P) = c + S_P(I)$$

Example

$S_P(I)$: number, size, values of inputs and outputs associated with *I*, recursive stack space, formal parameters, local variables, return address

```
float rsum(float list[], int n)
{
   if (n) return rsum(list,n-1) + list[n-1];
   return 0;
}
```

Program 1.11: Recursive function for summing a list of numbers

Type	Name	Number of bytes
parameter: float parameter: integer return address: (used internally)	*list*[] *n*	2 2 2 (unless a far address)
TOTAL per recursive call		6

Figure 1.1: Space needed for one recursive call of Program 1.11

$S_{sum}(I) = S_{sum}(n) = 6n$

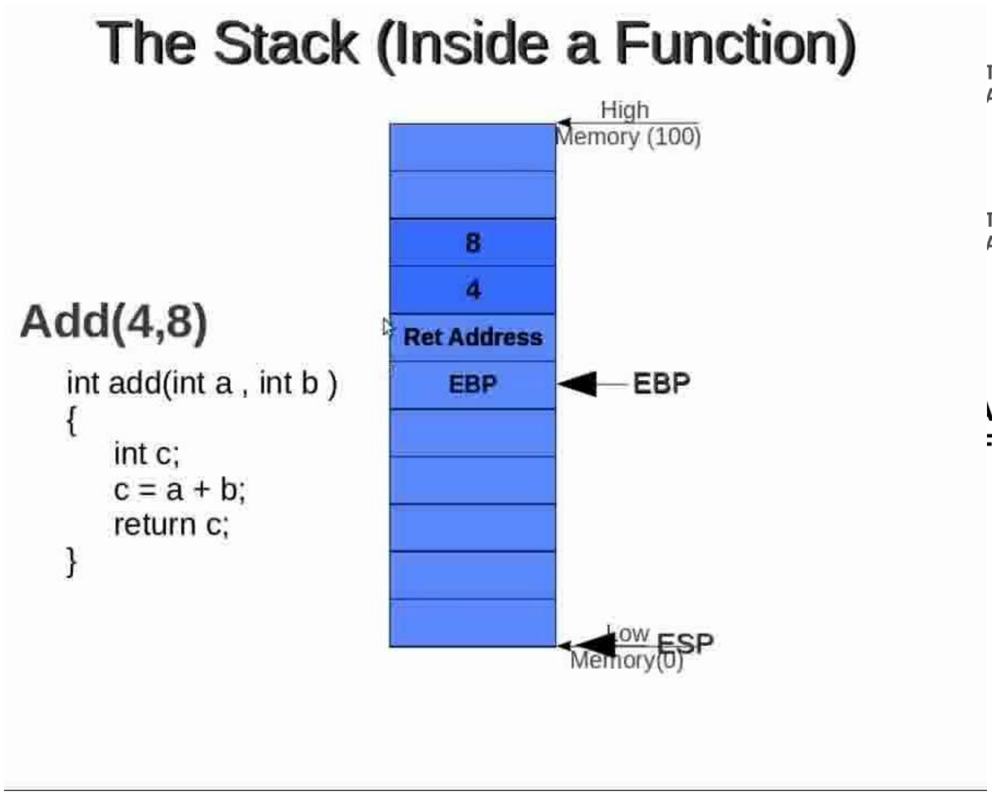
Example

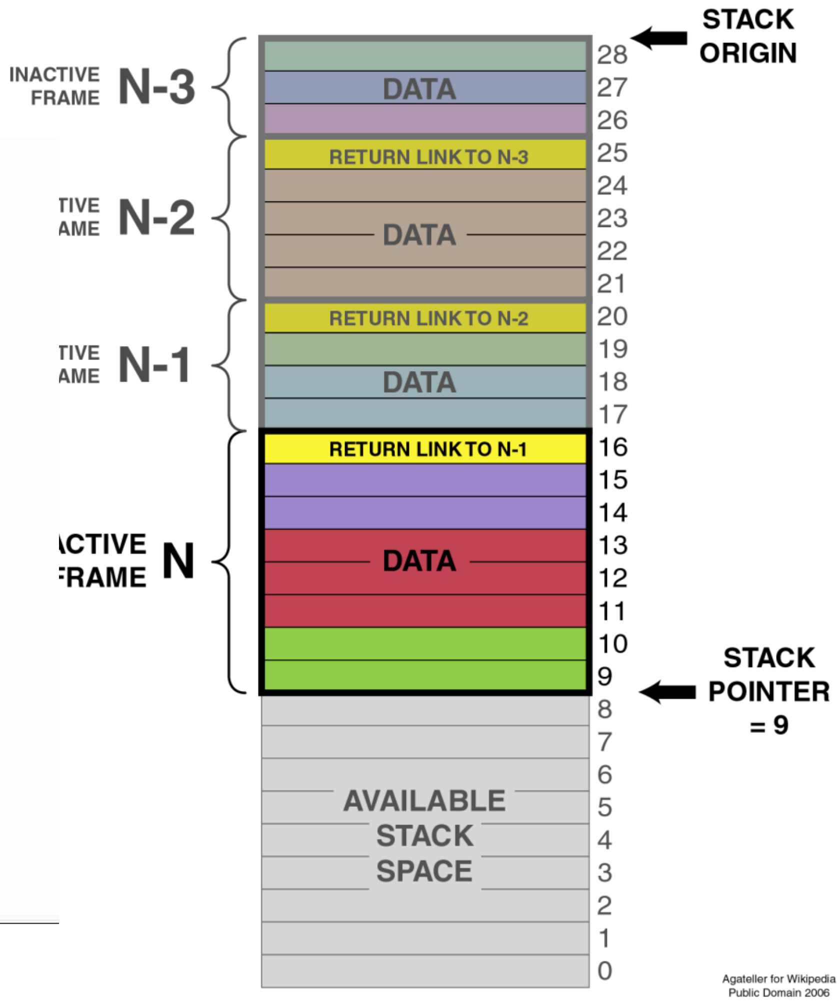
```
## char
## get_character(int i, int j) {
##      return puzzle[i * num_columns + j];
## }


 .globl get_character
 get_character:
        la      $t0, puzzle
        la      $t1, num_columns
        mul     $t2, $a0, $t1           ##t2 = i*num_columns
        add     $t3, $t2, $a1           ##t3 = i*num_columns+j
        lw      $t0, 0($t3)             ##load puzzle[i*num_columns + j]
        move    $v0, $t0
        jr      $ra
```

return

Recursive Stack Space





Example

– Example 1.6

$S_P(I)$: number, size, values of inputs and outputs associated with *I*, recursive stack space, formal parameters, local variables, return address

```
float abc(float a, float b, float c)
{
   return a+b+b*c +(a+b-c)/(a+b)+4.00;
}
```

$S_{abc}(I)=0$

Program 1.9: Simple arithmetic function

- $S_{sum}(I)=S_{sum}(n)=0$.

```
float sum(float list[], int n)
{
   float tempsum = 0;
   int i;
   for (i = 0; i < n; i++)
      tempsum += list[i];
   return tempsum;
}
```

$S_{sum}(I)=S_{sum}(n)=0$

Recall: pass the address of the first element of the array & pass by value

Program 1.10: Iterative function for summing a list of numbers

Time Complexity

- Time Complexity:

$$\boxed{T(P) = c + T_p(I)}$$

  - The time, $T(P)$, taken by a program, P, is the sum of its compile time c and its run (or execution) time, $T_p(I)$
  - Fixed time requirements
    - Compile time (c), independent of instance characteristics
  - Variable time requirements
    - Run (execution) time $T_p(I)$

Time Complexity

- How to evaluate $T(P)$?
- Three choices
  1. Use the system clock
  2. Number of steps performed
     - machine-independent
  3. Asymptotic analysis
     - machine-independent

Use the system clock

- Calculate the execution time of every operation

Add	Subtract	Load	Store
ADD(n)	SUB(n)	LDA(n)	STA(n)
c_a	c_s	c_l	c_{st}

$$T_P(n) = c_a ADD(n) + c_s SUB(n) + c_l LDA(n) + c_{st} STA(n)$$

Is it good to use?

Number of steps performed

- Definition of a **program step**
  - A program step is a syntactically or semantically meaningful program segment whose execution time is independent of the instance characteristics
    - 10 additions can be one step, $\underbrace{\text{100 multiplications}}_{\text{constant}}$ **can** also be one step

p42~p43 有計算C++ 語法之 steps 之概述
原則是”**一個表示式**” 算**一步**

  - 1st method: count by a program
  - 2nd method: build a table to count

“Object vs Class vs Instance” what’s the difference?
https://alfredjava.wordpress.com/2008/07/08/class-vs-object-vs-instance/

Time Complexity in C++

- General statements in a C++ program

	Step count
– Comments	0
– Declarative statements	0
– Expressions and assignment statements	1
– Iteration statements	it all depends on
– Switch statement	it all depends on
– If-else statement	it all depends on
– Memory management statements	1 (or n)
– Function invocation	1 (or depends on f(n)
– Function statements	0
– Jump statements	1 or n
• return f(n) or return 1	

Count by a Program

```
float sum (float *a, const int n)
{
      float s = 0; count++ ; // count 是全域變數
      for (int i = 0; i <n ; i++) {
          count++ ; //  因為 for
              s += a[i] ;
          count++ ; //  因為指派
      }
      count++ ; //  因為最後一次 for 的判斷
      count++ ; //  因為 return
      return s ;
}
```

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

2n+ 3

Example

```
float rsum (float *a , const int n)
{
    count++ ; //  因為 if 的條件判斷
    if (n <= 0) {
        count++ ; //  因為 return
        return 0;
    }
    else {
        count++ ; //  因為 return
        return (rsum (a, n-1) + a [n - 1]) ;
    }
}
```

$$t_{rsum}(0) = 2$$

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

2n+ 2

Example

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

```
void add (int **a, int **b, int **c, int m, int n)
{
    for (int i = 0 ; i < m ; i++)
    {
        count++; //  因為 for i
        for (int j = 0; i < n ; j++)
        {
            count++; / /因為 for j
            c [i][j] = a [i][j] + b [i][j];
            count++ ; //  因為指派
        }
        count++ ; //  因為最後一次的 for j
    }
    count++ ; //  因為最後一次的 for i
}
```

```
void add (int **a, int **b, int **c, int m, int n)
{
    for (int i = 0 ; i < m ; i++)
    {
        for (int j = 0; i < n ; j++)
         count += 2 ;
      count += 2 ;
    }
    count++ ;
}
```

2rows*cols+ 2rows+ 1

Time Complexity (Cont.)

- Note that a step count does not necessarily reflect the complexity of the statement.
- **Step per execution** (s/e): The s/e of a statement is the amount by which count changes as a result of the execution of that statement.

Build a Table to Count

- 2nd method: build a table to count
  - s/e: steps per execution
  - frequency: total numbers of times each statements is executed

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

line	s/e	frequency	Step No.
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 Step No.	$2n + 3$

Remarks of Time Complexity

- Difficulty: the time complexity is not dependent solely on the number of inputs or outputs
- To determine the step count
  - **Best case**, **Worst case**, and **Average**
- Example

```
int BinarySearch (int *a, const int x, const int n)
{ // 在排序好的陣列 a[0], …, a[n-1]中找出 x
    int left = 0, right = n-1 ;
    while (left <= right)
    { // 還有元素
        int middle = (left + right)/2;
        if (x < a[middle]) right=middle-1 ;
        else if (x > a[middle]) left = middle+1 ;
        else return middle ;
    } // while 迴圈結束
    return -1; // 沒找到
}
```

Asymptotic Analysis

- Determining the exact step count is difficult task
- Not very useful for comparative purpose

“3n+3”, “7n+2”, or “2n+15” 執行時間都相差不遠

- Determining the exact step count usually not worth(can not get exact run time)
- Motivation

  Compare the time complexity of two programs that computing the same function and predict the growth in run time as instance characteristics change

To represent “Rate of growth”

Example

- Given program P and Q
- $T_P(n) = c_1n^2 + c_2n$
- $T_Q(n) = c_3n$

- We can see that as n is large, Q will be faster than P, no matter what c_1, c_2, c_3 are

- Example:
  - $c_1 = 1, c_2 = 2, c_3 = 100$, then $c_1n^2 + c_2n^2 > c_3n$ for $n > 98$.
  - $c_1 = 1, c_2 = 2, c_3 = 1000$, then $c_1n^2 + c_2n^2 > c_3n$ for $n > 998$.

Break even point

- 需要知道 c_1, c_2, c_3 的數值嗎?

Asymptotic Analysis

- Running time of an algorithm as a function of input size *n* **for large *n***.
- Expressed using **only** the **highest-order term** in the expression for the exact running time.
  - Instead of exact running time, say $\Theta(n^2)$ or $O(n^2)$.
- Describes behavior of function in the limit.
- Written using ***Asymptotic Notation.***

Asymptotic Notation

- Five asymptotic notations (functions):
  - Big-O (O)
  - Theta (Θ)
  - Omega (Ω)
  - Small-O (o)
  - Small-Omega (ω)

Upper bound(current trend)

Lower bound

Upper and lower bound

- Defined for functions over the natural numbers.
  - **Ex:** $f(n) = \Theta(n^2)$.
  - Describes how $f(n)$ grows in comparison to n^2.

Asymptotic Notation O

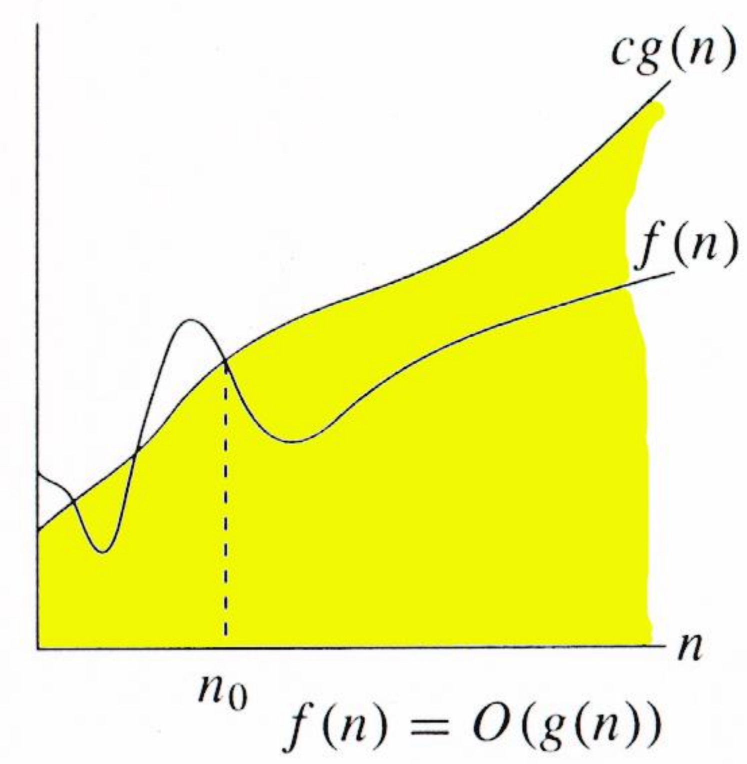
For function $g(n)$, we define $O(g(n))$, big-O of n, as the set:

$$O(g(n)) = \left\{ \begin{array}{c} f(n) : \exists \text{ positive constants } c \text{ and } n_0, \text{ such that } \forall n \geq n_0, \\ \text{we have } 0 \leq f(n) \leq cg(n) \end{array} \right\}$$

$O(g(n))$: Set of all functions whose *rate of growth* is the same as or lower than that of $g(n)$.

$f(n) = O(g(n))$ iff there exist **positive** constants c and n_0 such that $f(n) \leq cg(n)$ for all $n, n \geq n_0$

$g(n)$ is an *asymptotic upper bound* for $f(n)$.



Examples

- Show that $3n^3 = O(n^4)$ for appropriate c and n_0
- How?
  - How to Prove?
  - Find a pair of c and n_0, such that $\forall n \geq n0$, $0 \leq 3n^3 \leq cn^4$, then the proof is done!
- Any linear function $an + b$ is in $O(n^2)$. How?

Asymptotic Notation O (Cont.)

Theorem

If $f(n) = a_m n^m + \cdots + a_1 n + a_0$, then $f(n) = O(n^m)$

Proof:

$$f(n) \leq \sum_{i=0}^{m} |a_i| n^i$$

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

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

Exists $c = \sum_{i=0}^{m} |a_i|$ and $n_0 = 1$, $f(n) \leq cn^m$, for all $n \geq 1$.

So, $f(n) = O(n^m)$.

$f(n) \le cg(n)$ for all $n, n \ge n_0$

Example

- $3n + 2 = O(n)$?
- Yes, since $3n + 2 \le 4n$ for all $n \ge 2$.
- $3n + 3 = O(n)$?
- Yes, since $3n + 3 \le 4n$ for all $n \ge 3$.
- $100n + 6 = O(n)$?
- Yes, since $100n + 6 \le 101n$ for all $n \ge 10$.
- $10n^2 + 4n + 2 = O(n^2)$?
- Yes, since $10\mathrm{n}^2 + 4\mathrm{n} + 2 \le 11n^2$ for all $n \ge 5$.

$f(n) \leq cg(n)$ for all $n, n \geq n_0$

Example

- $1000n^2 + 100n - 6 = \mathrm{O}(n^2)$?
- Yes, since $1000n^2 + 100n - 6 \leq 1001n^2$ for all $n \geq 100$.
- $6 * 2^n + n^2 = \mathrm{O}(2^n)$?
- Yes, since $6 * 2^n + n^2 \leq 7 * 2^{\mathrm{n}}$ for all $n \geq 4$.
- $3n + 3 = \mathrm{O}(n^2)$?
- Yes, since $3n + 3 \leq 3n^2$ for all $n \geq 2$.
- $10n^2 + 4n + 2 = \mathrm{O}(n^4)$?
- Yes, since $10n^2 + 4n + 2 \leq 10n^4$ for all $n \geq 2$.
- $3n + 2 = \mathrm{O}(1)$?
- No. Cannot find c and n_0.

Some Rules

- Rule 1:

  If $T_P(n) = O(f(n))$ and $T_Q(n) = O(g(n))$ Then

  $T_P(N) + T_Q(N) = \max\Big(O\big(f(n)\big), O\big(g(n)\big)\Big)$

  $T_P(N) \times T_Q(N) = O\big(f(n) \times g(n)\big)$

- Rule 2:
  - If $T_P(n)$ is a polynomial of degree k, then $T(n) = \Theta(n^k)$

Running Time Calculation

- For loop

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

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

Running Time Calculation

- Bested loop

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

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

Running Time Calculations

- Consecutive statements

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

- $\max(1 \times n, 1 \times n \times n) = 1 \times n \times n = O(n^2)$

Running Time Calculations

- If/Else

```
if (i >0)
    x++;
else
    for ( i=0; i < n; i++)
        x++;
```

- $\max(1, 1 \times n) = n = O(n)$

Running Time Calculations

- *Recursive*

```
long int F(int n)
{
    if (n <=1)
        return 1;
    else
        return n * F(n -1);
}
```

- $T(n) = T(n-1) + c = T(n-2) + 2c \ldots$
  $= T(1) + (n-1)c$
  $= cn - c + 1$
  $= O(n)$

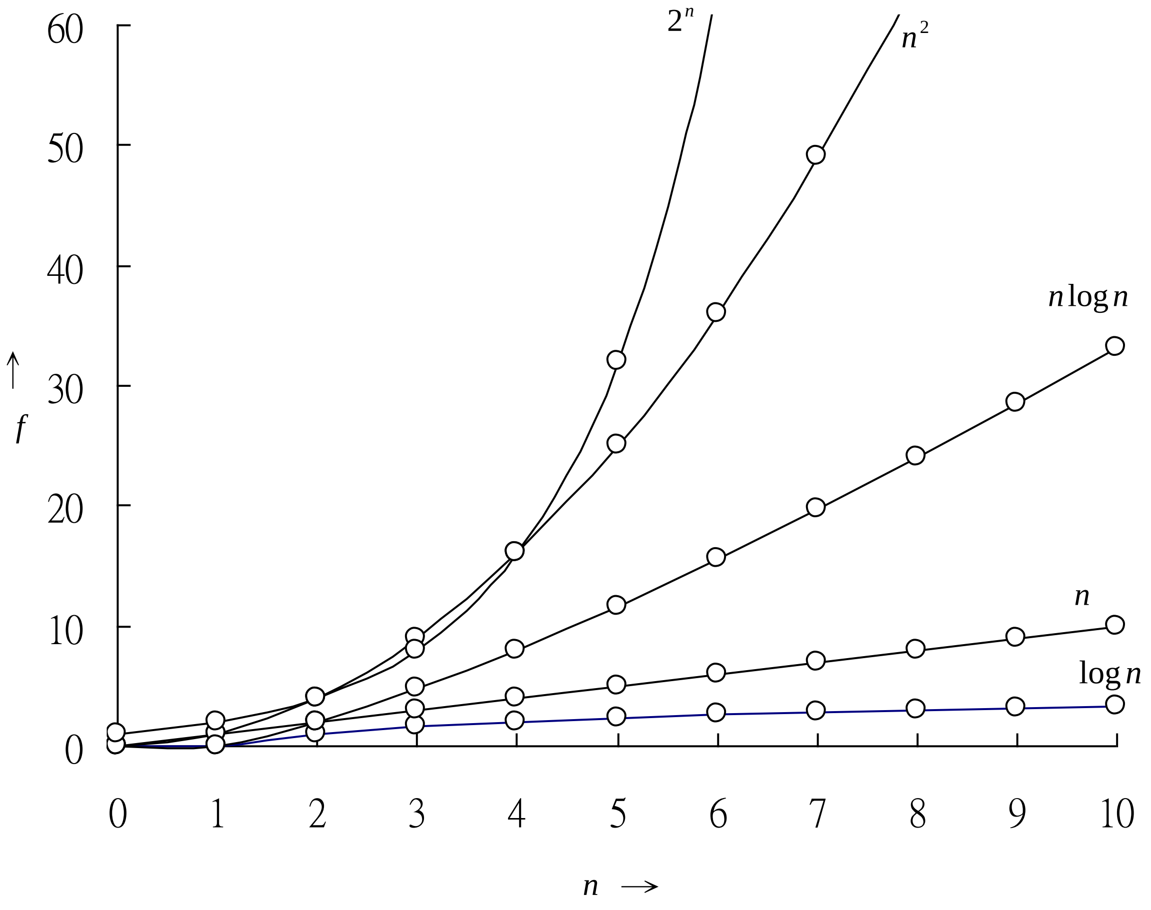
Remarks

- $O(g(n)) = f(n)$ is meaningless
- "=" as "*is*" and not as "equals"
- $g(n)$ is the least upper bound
  - $n = O(n) = O(n^2) = O(n^{2.5}) = O(n^3) = O(2^n)$
- $O(1)$: constant
- O(n): linear
- $O(n^2)$: quadratic
- $O(n^3)$: cubic
- $O(2^n)$: exponential

Faster

Slower

Magnitude



Measuring Efficiency

- Order of magnitude

$$1 < \log n < n < n \log n < n^2 < n^3 < 2^n < 3^n < n/2^{n/2} < n!$$

constant

acceptable

P

$f(n) \setminus n$	10	10^2	10^3
$log_2 n$	3.3	6.6	10
n	10	10^2	10^3
$nlog_2 n$	0.33×10^2	0.7×10^3	10^4
n^2	10^2	10^4	10^6
2^n	1024	1.3×10^3	$> 10^{100}$
$n!$	3^6	$> 10^{100}$	$> 10^{100}$

NP

Need to improve

Execution Times on a 1 BSPS Computer

	$f(n)$						
n	n	$n \log_2 n$	n^2	n^3	n^4	n^{10}	2^n
10	.01 µs	.03 µs	.1 µs	1 µs	10 µs	10s	1µs
20	.02 µs	.09 µs	.4 µs	8 µs	160 µs	2.84h	1ms
30	.03 µs	.15 µs	.9 µs	27 µs	810 µs	6.83d	1s
40	.04 µs	.21 µs	1.6 µs	64 µs	2.56ms	121d	18m
50	.05 µs	.28 µs	2.5 µs	125 µs	6.25ms	3.1y	13d
100	.10 µs	.66 µs	10 µs	1ms	100ms	3171y	$4*10^{13}$y
10^3	1 µs	9.96 µs	1 ms	1*s*	16.67m	$3.17*10^{13}$y	$32*10^{283}$y
10^4	10 µs	130 µs	100 ms	16.67m	115.7d	$3.17*10^{23}$y	
10^5	100 µs	1.66 ms	10s	11.57d	3171y	$3.17*10^{33}$y	
10^6	1ms	19.92ms	16.67m	31.71y	$3.17*10^7$y	$3.17*10^{43}$y	

µs = 百萬分之一秒 = 10^{-6} 秒；ms =千分之一秒 = 10^{-3} 秒
s = 秒；m = 分鐘；h = 小時；d = 日；y = 年；

Time for $f(n)$ instructions on 10^9 instr/sec computer

Function values

Instance characteristic n

Time	Name	1	2	4	8	16	32
1	Constant	1	1	1	1	1	1
$\log n$	Logarithmic	0	1	2	3	4	5
n	Linear	1	2	4	8	16	32
$nlog\ n$	Log Linear	0	2	8	24	64	160
n^2	Quadratic	1	4	16	64	256	1024
n^3	Cubic	1	8	61	512	4096	32768
2^n	Exponential	2	4	16	256	65536	4294967296
$n!$	Factorial	1	2	54	40326	20922789888000	$26313*10^{33}$

Asymptotic Notation Ω

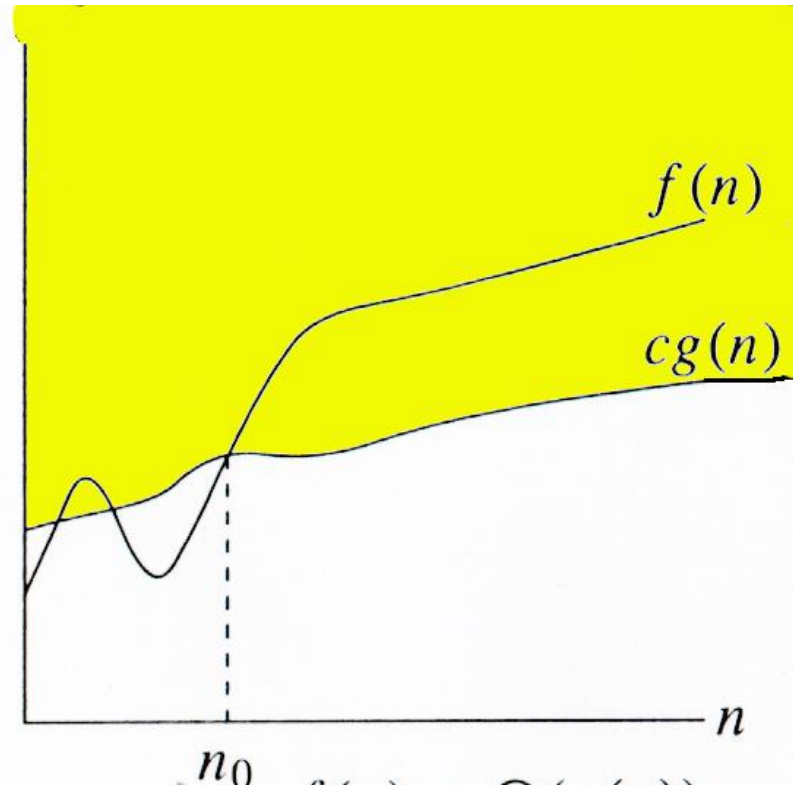
For function $g(n)$, we define $\Omega(g(n))$, big-Omega of n, as the set:

$$\Omega(g(n)) = \left\{ \begin{array}{c} f(n) : \exists \text{ positive constants } c \text{ and } n_0, \text{ such that } \forall n \geq n_0, \\ \text{we have } 0 \leq cg(n) \leq f(n) \end{array} \right\}$$

$\Omega(g(n))$: Set of all functions whose *rate of growth* is the same as or higher than that of $g(n)$.

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

$g(n)$ is an *asymptotic lower bound* for $f(n)$.



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

Asymptotic Notation Ω

- Examples
  - $3n + 2 = \Omega(n)$ as $3n + 2 \geq 3n$ for $n \geq 1$
  - $10n^2 + 4n + 2 = \Omega(n^2)$ as $10n^2 + 4n + 2 \geq n^2$ for $n \geq 1$
  - $6 * 2^n + n^2 = \Omega(2^n)$ as $6 * 2^n + n^2 \geq 2^n$ for $n \geq 1$
- Remarks
  - The largest lower bound
    - $3n + 3 = \cancel{\Omega(1)}$ $\Omega(\mathrm{n})$
    - $10n^2 + 4n + 2 = \cancel{\Omega(n)}$ $\Omega(n^2)$
    - $6 \times 2^n + n^2 = \cancel{\Omega(n^{100})}$ $\Omega(2^n)$
- Theorem
  - If $f(n) = a_m n^m + \ldots + a_1 n + a_0$ and $a_m > 0$, then $f(n) = \Omega(n^m)$

Example

- $\sqrt{n} = \Omega(\lg n)$. Choose c and n_0. How?

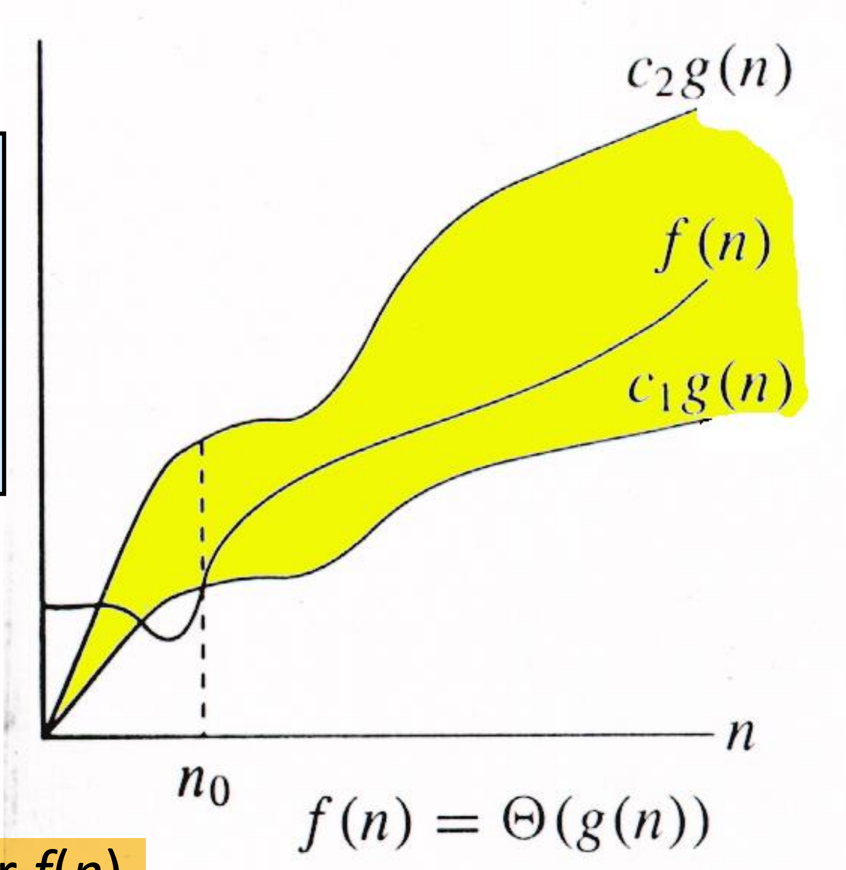
Asymptotic Notation Θ

$\Theta(g(n))$, big-Theta:

$f(n) = \Theta(g(n))$ **if ∃ positive constants c_1, c_2, and n_0, such that**

$$0 \le c_1 g(n) \le f(n) \le c_2 g(n)$$

for $\forall n \geq n_0$



$\Theta(g(n))$: Set of all functions that have the same *rate of growth* as $g(n)$.

$g(n)$ is an *asymptotically tight bound* for $f(n)$.

Asymptotic Notation Θ

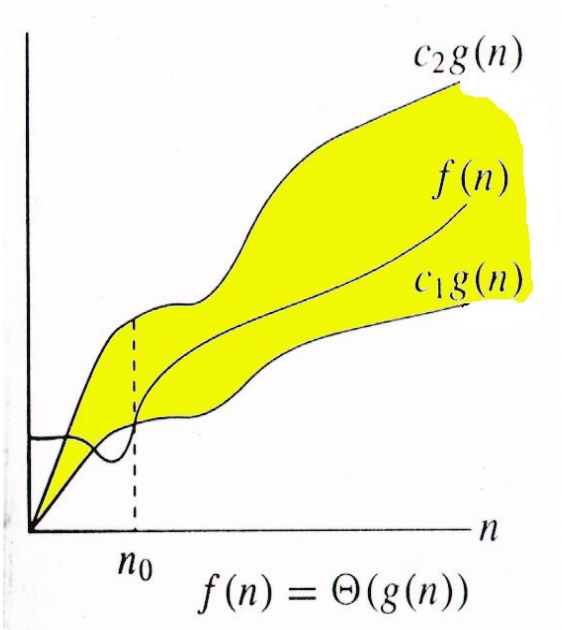
- Examples
  - $3n + 2 = \Theta(n)$ as $3n + 2 \geq 3n$ for $n > 1$ and $3n + 2 \leq 4n$ for all $n \geq 2$
  - $10n^2 + 4n + 2 = \Theta(n^2)$
  - $6 * 2^n + n^2 = \Theta(2^n)$
- Remarks
  - Both an upper and lower bound
  - $3n + 2 \neq \Theta(1)$
  - $10n^2 + 4n + 2 \neq \Theta(n)$
- Theorem
  - *If* $f(n) = a_m n^m + \ldots + a_1 n + a_0$ and $a_m > 0$, *then* $f(n) = \Theta(n^m)$

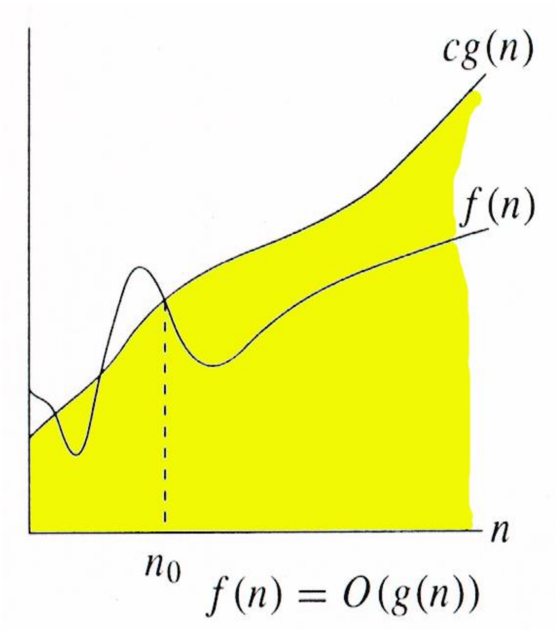
Example

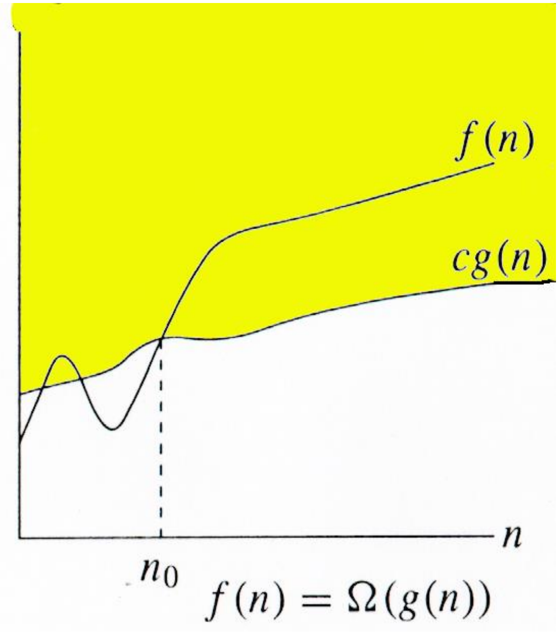
- Is $3n^3 \in \Theta(n^4)$??
- How about $2^{2n} \in \Theta(2^n)$??

Relations Between Θ, Ω, O

$$\Theta\big(g(n)\big) = O(g(n)) \cap \Omega(g(n))$$







Complexity Comparison

- Compare the order of magnitude of a logarithm log n with a power of n, say n^r $(r > 0)$
  - It is difficult to calculate the quotient log n / n^r
  - Need some mathematical tool
- Some usefule mathematics tools
  - Using **limits** in asymptotic analysis
    - Limit comparison test (LCT)
  - Taking **log** for easy of comparison

Complexity Comparison

- Using limits in asymptotic analysis
  - If $\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$ then:

    $f(n)$ has *strictly smaller order of magnitude* than $g(n)$.
  - If $\lim_{n\to\infty} \frac{f(n)}{g(n)}$ is finite and nonzero then:

    $f(n)$ has *the same order of magnitude* as $g(n)$.
  - If $\lim_{n\to\infty} \frac{f(n)}{g(n)} = \infty$ then:

    $f(n)$ has *strictly greater order of magnitude* than $g(n)$.

Complexity Comparison

符號	定義	極限判斷法
Big-O (O)	$f(n) = O(g(n)) \Leftrightarrow \exists c, n_0>0, \ni f(n) \leq cg(n), \forall n \geq n_0.$	$\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$
Small-O (o)	$f(n) = o(g(n)) \Leftrightarrow \forall c>0, \exists n_0>0, \ni f(n) < cg(n), \forall n \geq n_0.$	$\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$
Omega (Ω)	$f(n) = \Omega(g(n)) \Leftrightarrow \exists c, n_0>0, \ni f(n) \geq cg(n), \forall n \geq n_0.$	$\lim_{n\to\infty} \frac{f(n)}{g(n)} = \infty$
Small-Omega (ω)	$f(n) = \omega(g(n)) \Leftrightarrow \forall c>0, \exists n_0>0, \ni f(n) > cg(n), \forall n \geq n_0.$	$\lim_{n\to\infty} \frac{f(n)}{g(n)} = \infty$
Theta (θ)	$f(n) = \theta(g(n)) \Leftrightarrow \exists c_1,c_2, n_0>0, \ni c_1g(n) \leq f(n) \leq c_2g(n), \forall n \geq n_0.$	$\lim_{n\to\infty} \frac{f(n)}{g(n)} = L$

L'Hôpital's rule (羅必達定理)

- Functions f and g which are differentiable on an open interval I except possibly at a point c contained in I, if

$$\lim_{x \to c} f(x) = \lim_{x \to c} g(x) = 0 \text{ or } \pm\infty,$$

$g'(x) \neq 0$ for all x in I with $x \neq c$, and

$\lim_{x \to c} \frac{f'(x)}{g'(x)}$ exists, then

$$\lim_{x \to c} \frac{f(x)}{g(x)} = \lim_{x \to c} \frac{f'(x)}{g'(x)}$$

Example

- Use **L'Hôpital's Rule**

$$f(n) = \ln n \qquad g(n) = n^r, r > 0$$

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = \lim_{n\to\infty} \frac{\ln n}{n^r} = \lim_{n\to\infty} \frac{f'(n)}{g'(n)} = \lim_{n\to\infty} \frac{1/n}{rn^{r-1}} = \lim_{n\to\infty} \frac{1}{rn^r} = 0$$

=> $\ln n$ has strictly smaller order of magnitude than any positive power n^r of n, $r > 0$.

Example

$$f(n) = 3n^2 - 100n - 25 \qquad g(n) = n$$

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = \frac{3n^2 - 100n - 25}{n} = \infty$$

$\Rightarrow 3n^2$ - $100n$ - 25 has strictly greater order than n

$$f(n) = 3n^2 - 100n - 25 \qquad g(n) = n^2$$

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = \frac{3n^2 - 100n - 25}{n^2} = 3$$

$\Rightarrow 3n^2$ - $100n$ - 25 has the same order as n^2

Example

- For $a \geq 0$, $b > 0$, $\lim_{n\to\infty} (\lg^a n / n^b) = 0$,
- so $lg^a n = o(n^b)$, and $n^b = \omega(lg^a n)$
  - Prove using L'Hopital's rule repeatedly
-

Complexity Comparison

- Exponentials
  - Useful Identities

$$a^{-1} = \frac{1}{a}$$

$$(a^m)^n = a^{mn}$$

$$a^m a^n = a^{m+n}$$

  - Exponentials and polynomials

$$\lim_{n \to \infty} \frac{n^b}{a^n} = 0$$

$$\Rightarrow n^b = o(a^n)$$

Logarithms and Exponentials – Bases

- If the base of a logarithm is changed from one constant to another, the value is altered by a constant factor.
  - **Ex:** $\log_{10} n$ * $\mathbf{\log_2 10}$ = $\log_2 n$.
  - Base of logarithm is not an issue in asymptotic notation.
- Exponentials with different bases differ by a exponential factor (not a constant factor).
  - **Ex:** $2^n = \mathbf{(2/3)^n} * 3^n$.

Logarithms

$x = \log_b a$ is the exponent for $a = b^x$.

Natural log: $\ln a = \log_e a$

Binary log: $\lg a = \log_2 a$

$\lg^2 a = (\lg a)^2$

$\lg \lg a = \lg (\lg a)$

$$a = b^{\log_b a}$$

$$\log_c(ab) = \log_c a + \log_c b$$

$$\log_b a^n = n \log_b a$$

$$\log_b a = \frac{\log_c a}{\log_c b}$$

$$\log_b(1/a) = -\log_b a$$

$$\log_b a = \frac{1}{\log_a b}$$

$$a^{\log_b c} = c^{\log_b a}$$

Exercise

Express functions in A in asymptotic notation using functions in B.

A	B	
$5n^2 + 100n$	$3n^2 + 2$	$A \in \Theta(B)$

$A \in \Theta(n^2)$, $n^2 \in \Theta(B) \Rightarrow A \in \Theta(B)$

$\log_3(n^2)$	$\log_2(n^3)$	$A \in \Theta(B)$

$\log_b a = \log_c a / \log_c b$; $A = 2\lg n / \lg 3$, $B = 3\lg n$, $A/B = 2/(3\lg 3)$

$n^{\lg 4}$	$3^{\lg n}$	$A \in \omega(B)$

$a^{\log b} = b^{\log a}$; $B = 3^{\lg n} = n^{\lg 3}$; $A/B = n^{\lg(4/3)} \rightarrow \infty$ as $n \rightarrow \infty$

$\lg^2 n$	$n^{1/2}$	$A \in o(B)$

$\lim_{n\to\infty} \frac{\lg^a n}{n^b} = \lim_{n\to\infty} \frac{a \lg^{a-1} n}{b n^b} = 0$ (here $a = 2$ and $b = 1/2$) $\Rightarrow A \in o(B)$

(Prove using L'Hopital's rule repeatedly)

Example

- $lg(n!) \; = \; \Theta(n \, lg \, n)$
  - Prove using Stirling's approximation (in the text) for lg(n!).

$$lg(n!) = lg(1) + lg(2) + lg(3) + \cdots lg(n)$$

$$\boxed{lg(n!) = nlg(n) - n + O(\lg(n)}$$

Example

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

Line	s/e	Frequency	
1	0	-	$\Theta(0)$
2	1	$\Theta(m)$	$\Theta(m)$
3, 4	1	$\Theta(mn)$	$\Theta(mn)$
5	0	-	$\Theta(0)$
		$t_{Add}(m, n) =$	$\Theta(mn)$

Example

- The more global approach to count steps: focus the variation of instance characteristics

```
int BinarySearch (int *a, const int x, const int n)
{
    int left = 0, right = n-1 ;
    while (left <= right)
    {
        int middle = (left + right)/2;
        if (x < a[middle]) right=middle-1 ;
        else if (x > a[middle]) left = middle+1 ;
        else return middle ;
    }
    return -1;
}
```

worst case $\Theta(\log n)$

Example

```
void Permutations (char *a, const int k, const int m)
{ // generate permutations of a[k], …, a[m]
    if (k = = m)
    {
        for (int i =0; i <=m; i++) cout << a[i] << " " ;
        cout << endl ;
    }
    else // a [k : m]
        for (i = k ; i <= m ; i++)
        {
            swap(a[k], a[i]);
            Permutations(a, k+1, m) ;
            swap(a[k], a[i]) ;
        }
}
```

k= m, (m+1)

k< m,

for loop, m-k times

each call $T_{perm}(k+1, m)$ (T_{perm} (k+1, m))

hence, T_{perm} (k, m)= ((m-k)(T_{perm} (k+1, m)))

Using the substitution, we have

T_{perm} (0, m)= (m(m!)) , m>= 1

=>Tp()=O(max(m+1, m(m!)))=> O(m!)

Useful Summation Function

- **Constant Series:** For integers *a* and *b*, $a \le b$,

$$\sum_{i=a}^{b} 1 = b - a + 1$$

- **Linear Series (Arithmetic Series):** For $n \ge 0$,

$$\sum_{i=1}^{n} i = 1 + 2 + \cdots + n = \frac{n(n+1)}{2}$$

- **Quadratic Series:** For $n \ge 0$,

$$\sum_{i=1}^{n} i^2 = 1^2 + 2^2 + \cdots + n^2 = \frac{n(n+1)(2n+1)}{6}$$

Useful Summation Function

- **Cubic Series:** For $n \geq 0$,

$$\sum_{i=1}^{n} i^3 = 1^3 + 2^3 + \cdots + n^3 = \frac{n^2(n+1)^2}{4}$$

- **Geometric Series:** For real $x \neq 1$,

$$\sum_{k=0}^{n} x^k = 1 + x + x^2 + \cdots + x^n = \frac{x^{n+1} - 1}{x - 1}$$

For $|x| < 1$,

$$\sum_{k=0}^{\infty} x^k = \frac{1}{1 - x}$$

Useful Summation Function

- **Linear-Geometric Series:** For $n \geq 0$, real $c \neq 1$,

$$\sum_{i=1}^{n} ic^{i} = c + 2c^{2} + \cdots + nc^{n} = \frac{-(n+1)c^{n+1} + nc^{n+2} + c}{(c-1)^{2}}$$

- **Harmonic Series:** nth harmonic number, $n \in \mathrm{I}^{+}$,

$$H_n = 1 + \frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{n}$$

$$= \sum_{k=1}^{n} \frac{1}{k} = \ln(n) + O(1)$$

Example

- Magic square
  - An n-by-n matrix of the integers from 1 to n^2 such that the sum of each row and column and the two major diagonals is the same
  - Example, n= 5 (n must be odd)

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

Magic Square (Cont.)

- Coxeter has given the simple rule
  - Put a one in the middle box of the top row.

    Go up and left assigning numbers in increasing order to empty boxes.
  - If your move causes you to jump off the square, figure out where you would be if you landed on a box on the opposite side of the square.

    Continue with this box.
  - If a box is occupied, go down instead of up and continue.

Magic Square (Cont.)

```
void Magic (const int n){
//for n odd create a magic square which is declared as an array
    const int MaxSize = 51; // maximal size of the square
    int square [MaxSize][MaxSize], k, l ;
    // check whether n is odd
    if ((n > MaxSize) || (n < 1))
        throw "Error!..n out of range " ;
    else if (!(n%2)) throw "Error!..n is even \n" ;
    for (int i = 0; i < n; i++)
        fill(square[i], square[i] + n, 0) ; // STL Algorithm
    square[0][(n-1)/2] = 1; //middle of the first row
    // i and j index to the current position
    int key = 2; i = 0; int j = (n-1)/2;
    while (key <= n*n) {
    // move upward 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; // square is occupied, move down
        else { // square[k][l] is empty
            i = k; j = l;
        }
        square[i][j] = key;
        key++;
    } // end of while
```

```
// 輸出魔術方陣
    cout << "magic square of size " << n << endl;
    for ( i = 0; i < n; i++) {
        for ( j = 0; j < n; j++)
        copy(square[i], square[i] + n, ostream_iterator<int>(cout, " "));
        cout << endl;
    }
}
```

Practical Complexities

- Time complexity
  - Generally some function of the instance characteristics
- Remarks on " n "
  - If $T_P = \Theta(n), T_Q = \Theta(n^2)$, then we say *P* is faster than *Q* for "sufficiently large" n.
- For reasonable large n, $n > 100$, only program of small complexity, $n, nlog\ n, n^2, n^3$ are feasible
  - See Table 1.8

Chapter 1 Basic Concepts

- Overview: System Life Cycle
- Algorithm Specification
- Data Abstraction
- Performance Analysis
- Performance Measurement

Performance Measurement

- Obtaining the actual space and time of a program
  - Using Borland C++, 386PC at 25 MHz
  - Time(hsec): returns the current time in hundredths of a sec.
- Goal:

  Obtaining the curve of measurement to obtain the function of execution time.

  - Step 1, analyze g(n), as a start
  - Step 2, write a program to test

Trick1 : to time a short event, to repeat it several times

Trick2 : suitable test data need to be generated based on the algorithm itself

Performance Measurement

- In C’s standard library time.h
  - Clock function: system clock
  - Time function

Summary

- Overview: System Life Cycle
- Algorithm Specification
  - Definition, description
- Data Abstraction- ADT
- Performance Analysis
  - Time and space
    - $O(g(n))$
- Performance Measurement
- Generating Test Data
  - Analyze the algorithm being tested to determine classes of data

Auxiliary

Common Summation Functions

Review on Summations

- Why do we need summation formulas?
  - For computing the running times of iterative constructs (loops). (CLRS – Appendix A)
  - Example: Maximum Subvector
  - Given an array A[1…n] of numeric values (can be positive, zero, and negative) determine the subvector A[i…j] ($1 \leq i \leq j \leq n$) whose sum of elements is maximum over all subvectors.

1	-2	2	2

Review on Summations

- Why do we need summation formulas?
  - For computing the running times of iterative constructs (loops). (CLRS – Appendix A)
  - Example: Maximum Subvector
  - Given an array A[1…n] of numeric values (can be positive, zero, and negative) determine the subvector A[i…j] ($1 \leq i \leq j \leq n$) whose sum of elements is maximum over all subvectors.

1	-2	2	2

Review on Summations

```
MaxSubvector(A, n)
        maxsum ← 0;
        for i ← 1 to n
            do for j = i to n
                    sum ← 0
                    for k ← i to j
                        do sum += A[k]
                    maxsum ← max(sum, maxsum)
        return maxsum
```

- $T(n) = \sum_{i=1}^{n} \sum_{j=i}^{n} \sum_{k=i}^{j} 1$
- NOTE: This is not a simplified solution. What *is* the final answer?

Review on Summations

- **Constant Series:** For integers *a* and *b*, $a \le b$,

$$\sum_{i=a}^{b} 1 = b - a + 1$$

- **Linear Series (Arithmetic Series):** For $n \ge 0$,

$$\sum_{i=1}^{n} i = 1 + 2 + \cdots + n = \frac{n(n+1)}{2}$$

- **Quadratic Series:** For $n \ge 0$,

$$\sum_{i=1}^{n} i^2 = 1^2 + 2^2 + \cdots + n^2 = \frac{n(n+1)(2n+1)}{6}$$

Review on Summations

- **Cubic Series:** For $n \geq 0$,

$$\sum_{i=1}^{n} i^3 = 1^3 + 2^3 + \cdots + n^3 = \frac{n^2(n+1)^2}{4}$$

- **Geometric Series:** For real $x \neq 1$,

$$\sum_{k=0}^{n} x^k = 1 + x + x^2 + \cdots + x^n = \frac{x^{n+1} - 1}{x - 1}$$

For $|x| < 1$,

$$\sum_{k=0}^{\infty} x^k = \frac{1}{1-x}$$

Review on Summations

- **Linear-Geometric Series:** For $n \geq 0$, real $c \neq 1$,

$$\sum_{i=1}^{n} ic^i = c + 2c^2 + \cdots + nc^n = \frac{-(n+1)c^{n+1} + nc^{n+2} + c}{(c-1)^2}$$

- **Harmonic Series:** nth harmonic number, $n \in \mathrm{I}^+$,

$$H_n = 1 + \frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{n}$$

$$= \sum_{k=1}^{n} \frac{1}{k} = \ln(n) + O(1)$$

Review on Summations

- **Telescoping Series:**

$$\sum_{k=1}^{n} a_k - a_{k-1} = a_n - a_0$$

- **Differentiating Series:** For $|x| < 1$,

$$\sum_{k=0}^{\infty} kx^k = \frac{x}{(1-x)^2}$$

Review on Summations

- **Approximation by integrals:**
  - For monotonically increasing $f(n)$

$$\int_{m-1}^{n} f(x)dx \le \sum_{k=m}^{n} f(k) \le \int_{m}^{n+1} f(x)dx$$

  - For monotonically decreasing $f(n)$

$$\int_{m}^{n+1} f(x)dx \le \sum_{k=m}^{n} f(k) \le \int_{m-1}^{n} f(x)dx$$

- **How?**

Review on Summations

- ***n*th harmonic number**

$$\sum_{k=1}^{n} \frac{1}{k} \geq \int_{1}^{n+1} \frac{dx}{x} = \ln(n+1)$$

$$\sum_{k=2}^{n} \frac{1}{k} \leq \int_{1}^{n} \frac{dx}{x} = \ln n$$

$$\Rightarrow \sum_{k=1}^{n} \frac{1}{k} \leq \ln n + 1$$