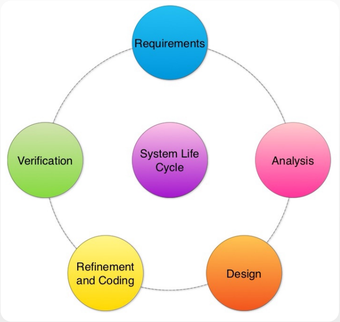
Basic Concepts

Data Structures

Ching-Fang Hsu
Department of Computer Science and Information Engineering
National Cheng Kung University

Overview: System Life Cycle

- Tools and techniques necessary to design and implement large-scale computer systems
  - Data abstraction
  - Algorithm specification
  - Performance analysis and measurement
  - Recursive programming



- The system life cycle -- the development process of programs; five highly interrelated phases

Overview: System Life Cycle (contd.)

- Requirements 開出規格
  - Describing the information that we are given (input) and the results that we must produce (output)
- Analysis
  - Breaking the problem down into manageable pieces
  - Bottom-up & top-down　將問題由大化小

從基本型，額外去修改

- Design
  - The creation of abstract data types
  - The specification of algorithms and a consideration of algorithm design strategies
  - Coding details are ignored!

Overview: System Life Cycle (contd.)

- Refinement and coding Debug & Optimize program
  - Choosing representations for our data objects and writing algorithms for each operation on them
- Verification
  - Correctness proofs
    - The same techniques used in mathematics; time-consuming
  - Testing
    - Good test data should verify that every piece of code runs correctly.
  - Error removal
    - The ease with which we can remove errors depends on the design and coding decisions made earlier.

Algorithm Specification

- Definition: An *algorithm* is a finite set of instructions that, if followed, accomplishes a particular task and must satisfy the following criteria:
  - Input >=0
  - Output >0
  - Definiteness 執行是明確的（只有做和不做）
  - Finiteness Instruction有限次
  - Effectiveness 簡單明瞭

Algorithm Specification (contd.)

演算法有限執行次數，在有限時間內執行完畢

- cf. a program
  - A program does not have to satisfy finiteness condition.
- How to describe an algorithm?
  - In a natural language
    - No violation of definiteness is allowed.
  - By flowcharts 減少文字敘述
    - Working well only if the algorithm is small and simple
      使用時機：演算法較為簡單

Algorithm Specification (contd.)

- Example: Selection Sort (p. 9)
  - Description statements; not an algorithm

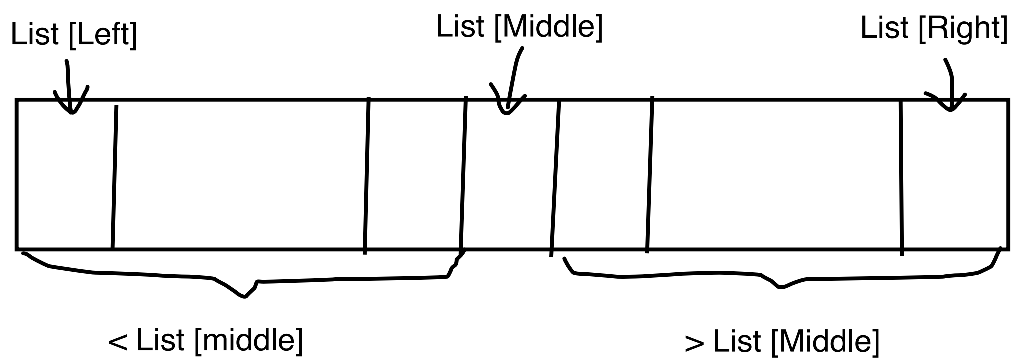
    From those integers that are currently unsorted, find the smallest and place it next in the sorted list.
  - A selection sort algorithm (p. 9, Program 1.2)

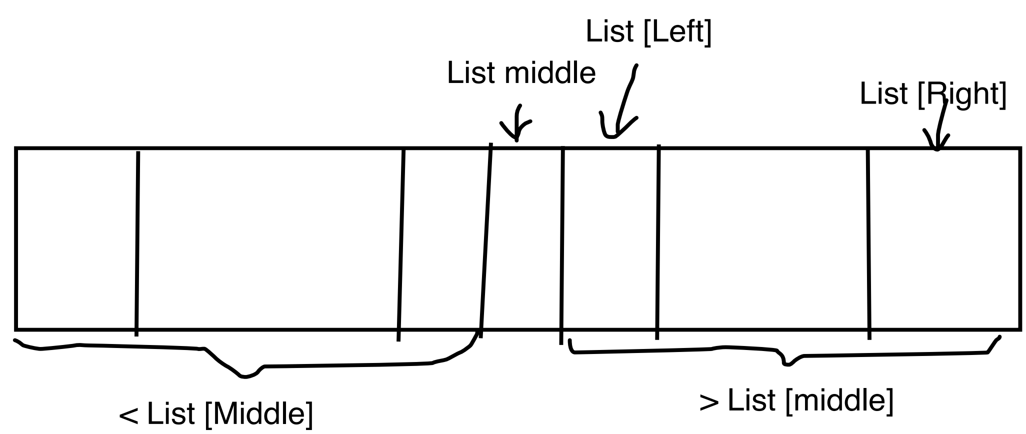
```
for (i=0; i<n; i++) {
  Examine list[i] to list[n-1] and
  suppose that the smallest integer is
  at list [min];
  Interchange list[i] and list[min];
}
```

Algorithm Specification (contd.)

- Example: Binary search (p. 10)
  - Given a sorted array *list* with $n \geq 1$ distinct integers, figure out if an integer *searchnum* is in *list*.
  - Binary search algorithm (p. 2, Program 1.5)

```
while (there are more integers to check) {
  middle = (left + right) / 2;
  if (searchnum < list[middle])
    right = middle - 1;
  else if (searchnum == list[middle])
         return middle;
       else left = middle + 1;
}
```


List [Left]
List [Middle]
List [Right]
< List [middle]
> List [Middle]


List [Left]
List middle
List [Right]
< List [Middle]
> List [middle]

Recursive Algorithms

- Direct recursion
  - Functions call themselves.
- Indirect recursion
  - Functions may call other functions that invoke the calling function again.
- Any function that we can write using assignment, `if-else`, and `while` statements can be written recursively.
  - Easier to understand

Recursive Algorithms (contd.)

- When should we express an algorithm recursively?
  - The problem itself is defined recursively.
  - Example: factorials, Fibonacci numbers, and binomial coefficients
- Example: Binary search
  - Recursive version (p. 15, Program 1.8)

需要Stack的資源

Recursive 執行效率較Iterative 差

```
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 1.8: Recursive implementation of binary search

Data Abstraction

- Definition: A *data type* is a collection of objects and a set of operations that act on those objects.
  - Example: `int` and arithmetic operations
- All programming languages provide at least a minimal set of predefined data types, plus the ability to construct *user-defined types*.
- Knowing the representation of the objects of a data type can be useful and dangerous.

Data Abstraction (contd.)

保護資料（使用者透過operation才能讀取資料）

- Definition: An *abstract data type* (*ADT*) is a data type whose specification of the objects and the operations on the objects is separated from the representation of the objects and the implementation of the operations.

只有描述

只展現function 的方法，不顯現完整code

- Specification vs. Implementation (of the operations of an ADT)

Specification(輪廓)：Input type, output
Implementation(內容)：coding

  - The former consists of the names of every function, the type of its arguments, and the type of its result.

Data Abstraction (contd.)

- Categories of functions of a data type
  - Creator/constructor 使用時機：初使狀態（初使化值）
    destructor
  - Transformers 狀態改變(function)
  - Observers/reporters 查看object的狀態，並非修改
  - Example: p. 20, ADT 1.1

ADT *NaturalNumber* 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 $x, y \in$ *NaturalNumber*; *TRUE*, *FALSE* $\in$ *Boolean*
and where +, −, <, and == are the usual integer operations

NaturalNumber Zero( )	::=	0
Boolean IsZero(x)	::=	**if** (x) **return** *FALSE* **else return** *TRUE*
Boolean Equal(x, y)	::=	**if** ($x == y$) **return** *TRUE* **else return** *FALSE*
NaturalNumber Successor(x)	::=	**if** ($x ==$ *INT_MAX*) **return** x **else return** $x + 1$
NaturalNumber Add(x, y)	::=	**if** (($x + y$) <= *INT_MAX*) **return** $x + y$ **else return** *INT_MAX*
NaturalNumber Subtract(x, y)	::=	**if** ($x < y$) **return** 0 **else return** $x - y$

end *NaturalNumber*

ADT 1.1: Abstract data type *NaturalNumber*

Performance Analysis

- Criteria of performance evaluation can be divided into two distinct fields.

  針對code的效能的估測（不考慮操作環境）
  - *Performance analysis* -- Obtaining estimates of time and space that are machine-independent
  - *Performance measurement* -- Obtaining machine-dependent times

Performance Analysis -- Space Complexity

在完成特定的task,所需的最大記憶耗損量

- Definition: The *space complexity* is the amount of memory that it needs to run to completion.
- Equal to the sum of the following components
  - Fixed space requirements 固定使用的space,需記憶體配置
    - Do not depend on the number and size of the program's inputs and outputs

      Translate to machine code　　Global variable
    - Including the instruction space, space for simple variables, fixed-size structured variables, and constants

Performance Analysis -- Space Complexity (contd.)

每次執行所需的記憶體大小不同，根據input決定大小

- Variable space requirements
  - The space needed by structured variables whose size depends on the particular instance, I, of the problem and the additional space required when a function uses recursion 不同級數，所需大小不同
  - $S_P(I)$: The variable space requirement of a program P working on an instance I
    - Usually a function of some characteristics of the instance I
      - The number, size, and values of the inputs and outputs associated with I

- The total space requirement $S(P)$
  - $\boxed{S(P) = c + S_P(I)}$, where c is a constant representing the fixed space requirements

Performance Analysis -- Time Complexity

- The time, $T(P)$, taken by a program P is the sum of its *compile time* and its *run/execution time*.
  - Compile time
    - Similar to the fixed space component
    - Does not depend on the instance characteristics
  - Execution time T_P 與執行環境優劣有關
    - Machine-independent estimate
    - Counting the number of operations performed in P
    - A problem: How is P divided into distinct steps?

      轉換成多少operation估算（每一步驟所需時間不同）

Performance Analysis -- Time Complexity (contd.)

- Definition: A *program step* is a syntactically meaningful program segment whose execution time is independent of the instance characteristics.

  會影響程式計算結果　無意義：不產生對應程式碼

  initialize設定也算一步

  一個Program Step:
  - 有意義的程式
  - 執行時間是固定的

  - The amount of computing represented by one program step may be different from that represented by another step.
- How to determine the number of steps?
  - Creating a global variable (p.27~29)　加入Global Variable計算程式經過多少步
  - A tabular method (p.30~31)　Total Steps = Single step * Frequency

  表格歸納法（找出函數對應關係，並可找出極大極小值）

```
float sum(float list[], int n)
{
   float tempsum = 0;
   int i;
   for (i = 0; i < n; i++)
   count += 2;
   count +=3;
   return 0;
}
```

Global variable

Program 1.14: Simplified version of Program 1.13

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

Figure 1.2: Step count table for Program 1.11

Statement	s/e	Frequency	Total Steps
void add(int a[][MAX_SIZE] $\cdots$)	0	0	0
{	0	0	0
int i, j;	0	0	0
~~for (i=0; i<rows; i++)~~	1	$rows+1$	$rows+1$
for (j = 0; j < cols; j++)	1	$rows \cdot (cols+1)$	$rows \cdot cols + rows$
c[i][j] = a[i][j] + b[i][j];	1	$rows \cdot cols$	$rows \cdot cols$
}	0	0	0
Total			$2rows \cdot cols + 2rows+1$

Figure 1.4: Step count table for matrix addition

Performance Analysis -- Time Complexity (contd.)

- The *best case step count*
  - The minimum number of steps that can be executed for the given parameters
- The *worst case step count*
  - The maximum number of steps that can be executed for the given parameters
- The *average step count*
  - The average number of steps executed on instances with the given parameters

Performance Analysis -- Asymptotic Notation (O, Ω, Θ)

Big O Omega Theta

- Because of the inexactness of what a step stands for, the exact step count isn't very useful for comparative purposes.
- Definition: $f(n) = O(g(n))$ iff there exist positive constants c and n_0 such that $f(n) \leq cg(n)$ for all n, $n \geq n_0$.

  雙向皆成立

  上限

  可以找出兩個正數c,n0，在 n>=n0情況下,使得不等式成立

  - p.35, Example 1.15
  - $O(1)$ ➩ constant computing time, $O(n)$ ➩ linear, $O(n^2)$ ➩ quadratic, $O(2^n)$ ➩ exponential

Performance Analysis -- Asymptotic Notation (O, Ω, Θ) (contd.)

- $f(n) = O(g(n))$ only states that $g(n)$ is an upper bound on the value of $f(n)$ for all n, $n \geq n_0$ instead of implying how good this bound is.
  - So, $n = O(n^2)$, $n = O(n^{2.5})$, $n = O(n^3)$, $n = O(2^n)$, etc.
    g(n)越小，估計越精確
  - To be informative, $g(n)$ should be as small a function of n as one can come up with for which $f(n) = O(g(n))$.
- Theorem 1.2: If $f(n) = a_m n^m + \ldots + a_1 n + a_0$, then $f(n) = O(n^m)$.
  - <proof> p. 36

Theorem 1.2: If $f(n) = a_m n^m + \ldots + a_1 n + a_0$, then $f(n) = O(n^m)$.

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

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

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

So, $f(n) = O(n^m)$. □

Performance Analysis -- Asymptotic Notation (O, Ω, Θ) (contd.)

- Definition: $f(n) = \Omega(g(n))$ iff there exist positive constants c and n_0 such that $f(n) \geq cg(n)$ for all n, $n \geq n_0$.
  下限
  - To be informative, $g(n)$ should be as large a function of n as possible for which the statement $f(n) = O(g(n))$ is true. g(n)越大，估計越精確
- Theorem 1.3: If $f(n) = a_m n^m + \ldots + a_1 n + a_0$ and $a_m > 0$, then $f(n) = \Omega(n^m)$.

Performance Analysis -- Asymptotic Notation (O, Ω, Θ) (contd.)

- Definition: $f(n) = \Theta(g(n))$ iff there exist positive constants c_1, c_2, and n_0 such that $c_1 g(n) \leq f(n) \leq c_2 g(n)$ for all n, $n \geq n_0$.
- Theorem 1.4: If $f(n) = a_m n^m + \ldots + a_1 n + a_0$ and $a_m > 0$, then $f(n) = \Theta(n^m)$.
- Example: Figure 1.5, p. 38
  - Since the number of lines is a constant, then we can take the maximum of the line complexities as the asymptotic complexity of the function

程式碼行數是固定的

找影響層級最大（乘長幅度最高）

Statement	Asymptotic complexity
`void add(int a[][MAX_SIZE] ··· )`	0
`{`	0
`    int i, j;`	0
`    for (i=0; i<rows; i++)`	$\Theta(rows)$
`        for (j = 0; j < cols; j++)`	$\Theta(rows.cols)$
`            c[i][j] = a[i][j] + b[i][j];`	$\Theta(rows.cols)$
`}`	0
Total	$\Theta(rows.cols)$

Figure 1.5: Time complexity of matrix addition