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
  - 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
- 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
  - 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
  - Output
  - Definiteness
  - Finiteness
  - Effectiveness

Algorithm Specification (contd.)

- 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
    - Working well only if the algorithm is small and simple

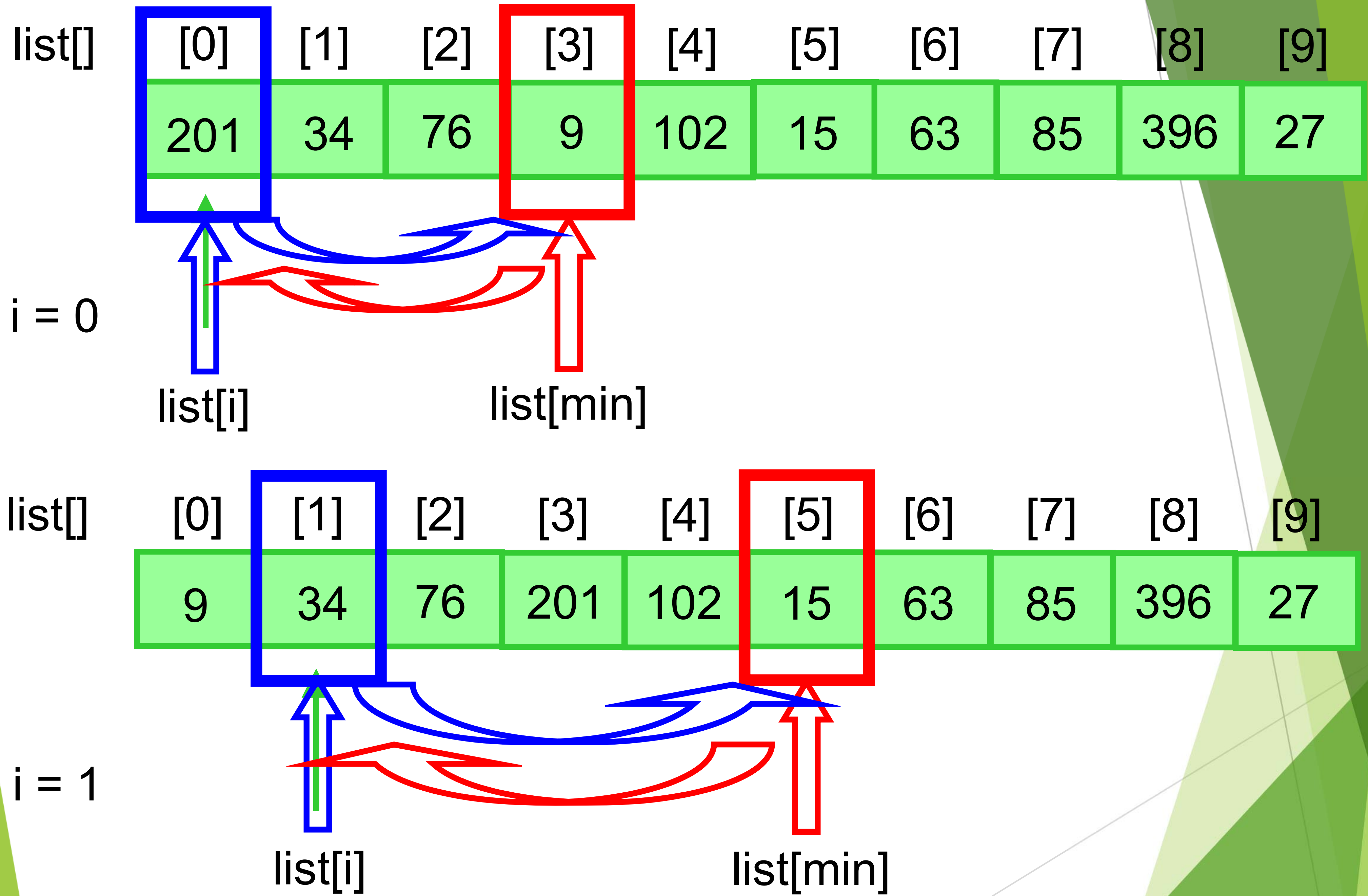
Algorithm Specification (contd.)

- 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.

  - Selection sort algorithm (p. 9, Program 1.2)

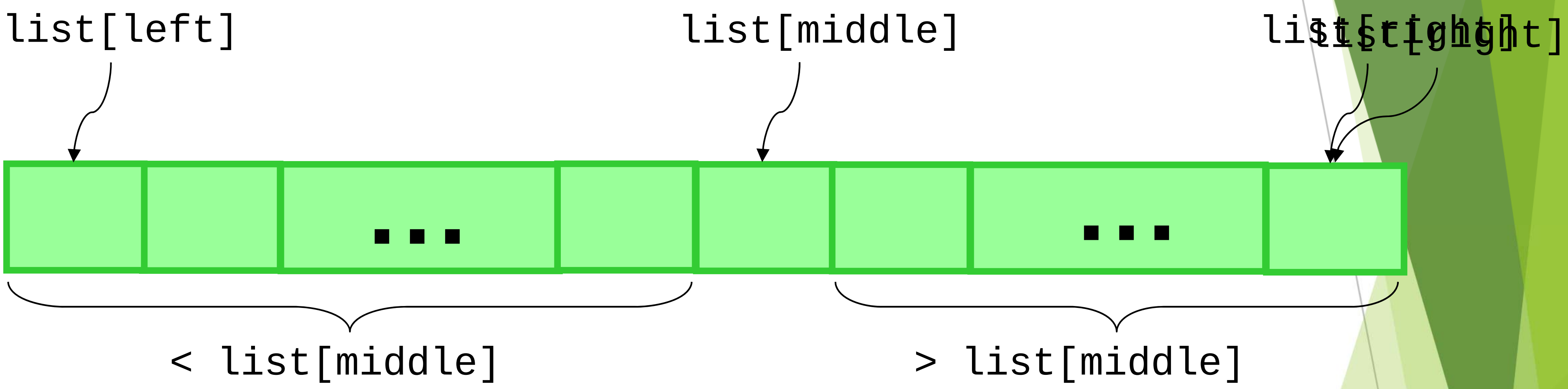
```
for (i=0; i<n; i++) {
  Examine list[i] to list[n-1] and
  suppose that the smallest integer is
  at list [min];
  Interchange list[i] and list[min];
}
```


list[]
[0] [1] [2] [3] [4] [5] [6] [7] [8] [9]
201 34 76 9 102 15 63 85 396 27
i = 0
list[i]
list[min]
list[]
[0] [1] [2] [3] [4] [5] [6] [7] [8] [9]
9 34 76 201 102 15 63 85 396 27
i = 1
list[i]
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. 12, 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]

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)

```
int binarysearch (int list[], int searchnum, int left,
                        int right)
{
 int middle ;
 if (left <= right) {
    middle = (left + right)/2;
    switch(COMPARE(list[middle], searchnum)) {
       case -1 : return
       binarysearch (list, searchnum, middle+1, right);
       case  0 : return middle;
       case  1 : return
       binarysearch (list, searchnum, left, middle-1);
    }
 }
 return -1;
}
```

recursive call

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.)

- 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.
- Specification vs. Implementation (of the operations of an ADT)
  - 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
  - Transformers
  - Observers/reporters
  - 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 ∈ NaturalNumber ; TRUE, FALSE ∈ 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
```

Performance Analysis

- Criteria of performance evaluation can be divided into two distinct fields.
  - *Performance analysis* -- Obtaining estimates of time and space that are machine-independent
  - *Performance measurement* -- Obtaining machine-dependent times

Performance Analysis -- Space Complexity

- 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
    - Do not depend on the number and size of the program’s inputs and outputs
    - Including the instruction space, space for simple variables, fixed-size structured variables, and constants

Performance Analysis -- Space Complexity (contd.)

- 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
  - $S_P(I)$: 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 $S(P)$
  - $S(P) = c + S_P(I)$, 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
    - Machine-independent estimate
    - Counting the number of operations performed in P
    - A problem: How is P divided into distinct steps?

Performance Analysis -- Time Complexity (contd.)

- Definition: A *program step* is a syntactically meaningful program segment whose execution time is independent of the instance characteristics.
  - 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.26~29, Program 1.13~1.18)
  - A tabular method (p.30~31)

Program 1.13

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

Program 1.14 (Simplified version of Program 1.13)

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

Program 1.15

```
float rsum(float list[] , int n)
{

    if (n) {

        return rsum(list, n-1) + list[n-1];
    }

    return list[0] ;
}
```

Program 1.17

```
void add(int a[ ][MAX_SIZE], int b[ ][MAX_SIZE],
                        int c[ ][MAX_SIZE], int row, int cols )
{
   int i, j;
   for (i = 0; i < rows; i++){

        for (j = 0; j < cols; j++) {

           c[i][j] = a[i][j] + b[i][j];

        }

  }

}
```

Program 1.18 (Simplified version of Program 1.17)

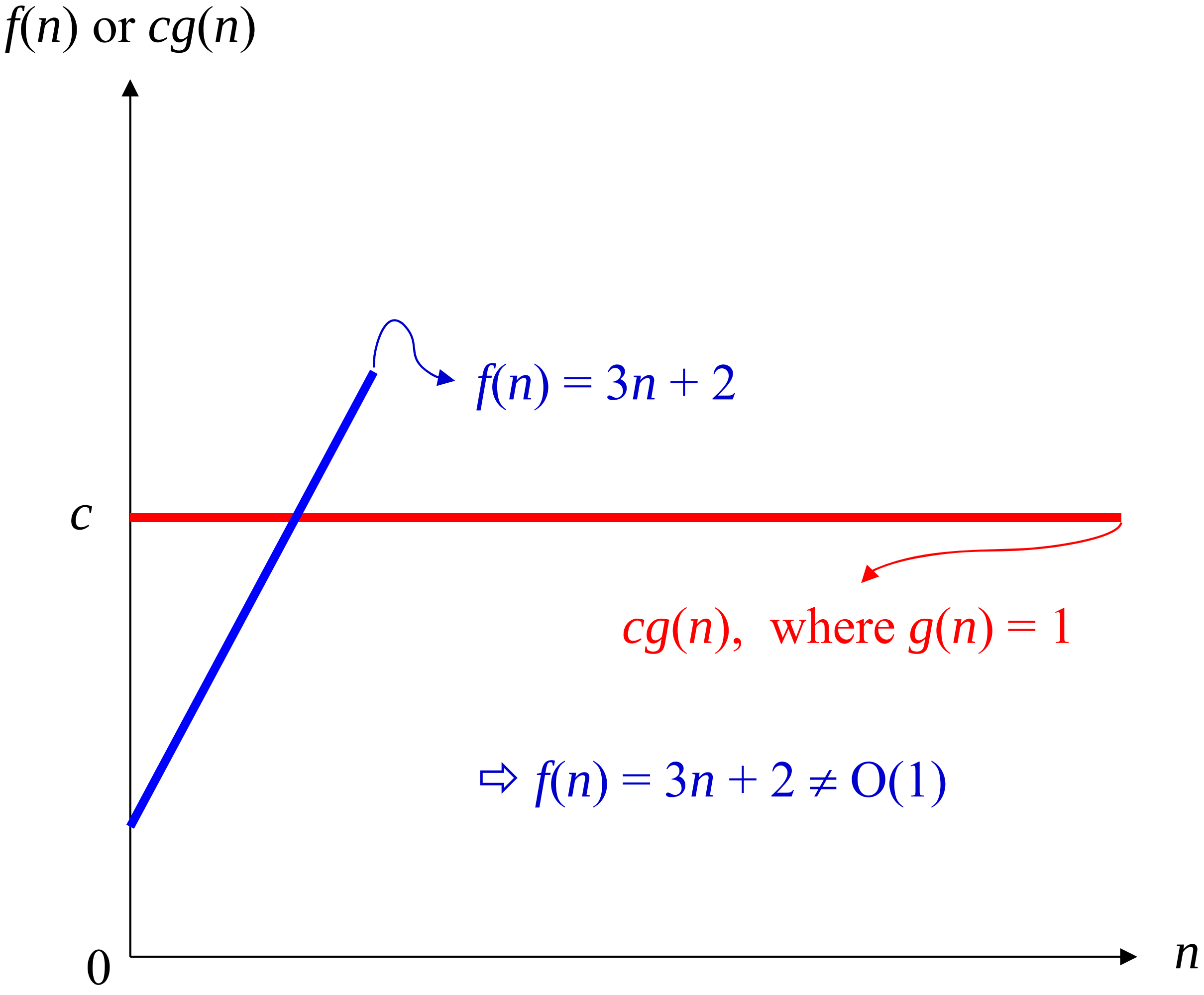
```
void add (int a[][MAX_SIZE], int b[][MAX_SIZE],
    int c[][MAX_SIZE], int rows, int cols)
{
    int i, j;
    for ( i = 0 ; i < rows ; i++) {
        for ( j = 0 ; i < cols ; j++)



    }

}
```

Performance Analysis -- Time Complexity (contd.)

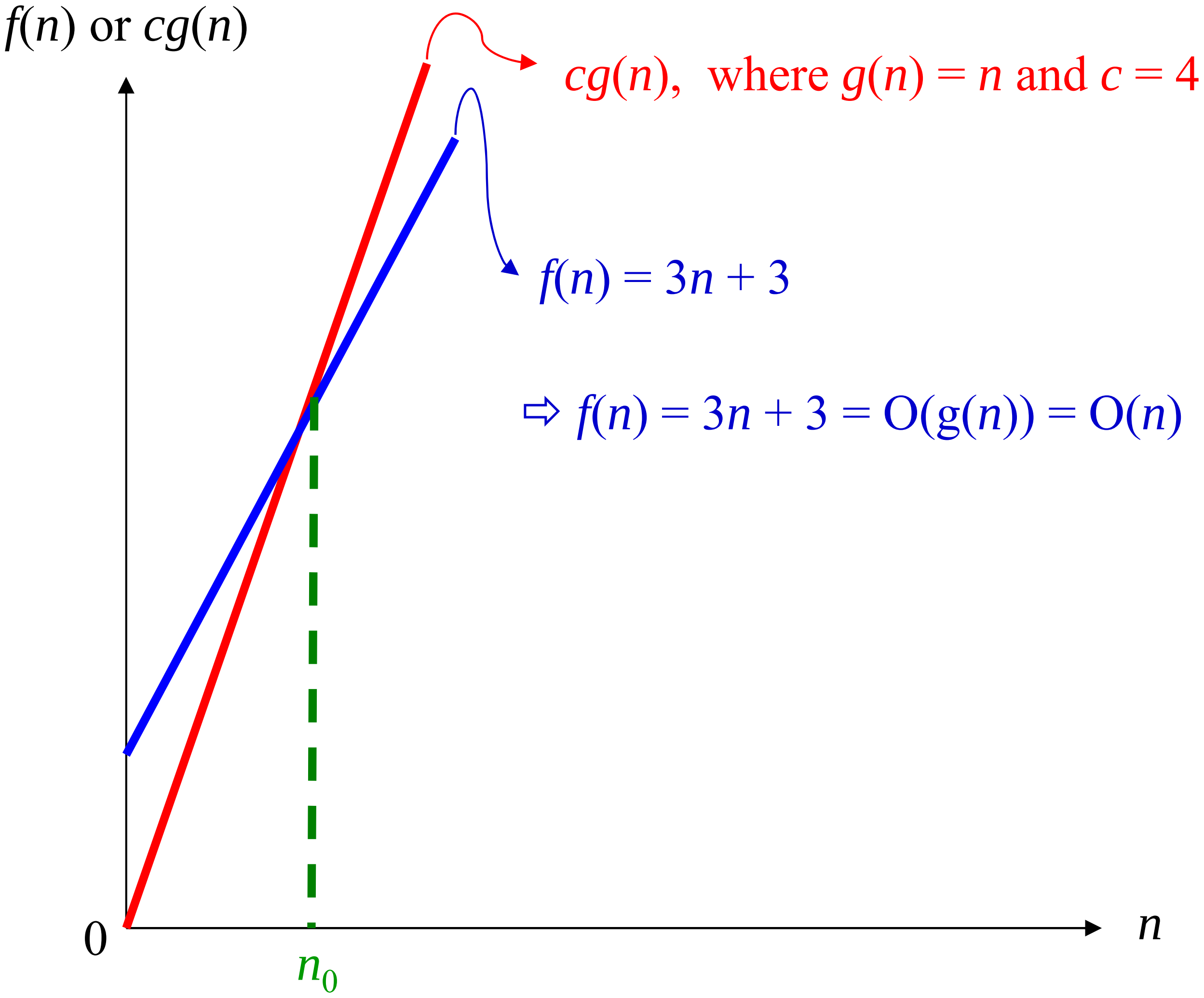
- The *best case step count*
  - The minimum number of steps that can be executed for the given parameters
- The *worst case step count*
  - The maximum number of steps that can be executed for the given parameters
- The *average step count*
  - The average number of steps executed on instances with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

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

- 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 $f(n) \le cg(n)$ for all n, $n \ge n_0$.
  - p.35, Example 1.15
  - $O(1)$ ⇨ constant computing time, $O(n)$ ⇨ linear, $O(n^2)$ ⇨ quadratic, $O(2^n)$ ⇨ exponential



⇨ $f(n) = 3n + 2 \neq O(1)$



⇨ $f(n) = 3n + 3 = O(g(n)) = O(n)$

Performance Analysis -- Asymptotic Notation (O) (contd.)

- $3n+2 = O(n)$, $3n+2 \neq O(1)$
- $3n+3 = O(n)$, $3n+3 = O(n^2)$
- $100n+6 = O($
- $10n^2+4n+$ … $O(n)$
- $1000n^2+1$
- $6*2^n+n^2 = O(2^n)$

$$3n+2 \le c \Rightarrow n \le \frac{c-2}{3}$$

Let $c = 4$, $3n+3 \le 4n$
$\Rightarrow n \ge 3$

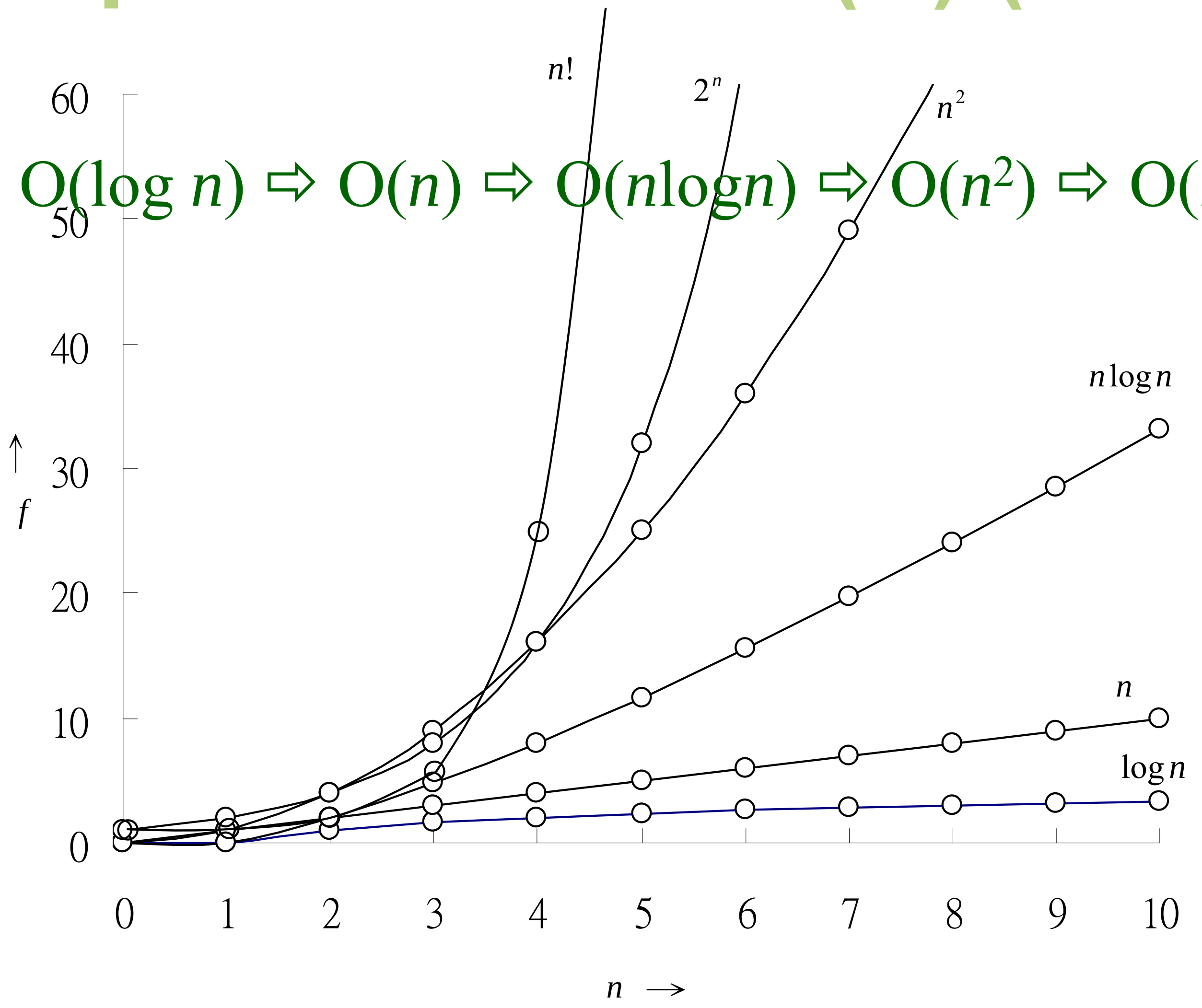
n_0

Performance Analysis -- Asymptotic Notation (O) (contd.)

$\log n$	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	65,536
5	32	160	1024	32,768	4,294,967,296

Performance Analysis -- Asymptotic Notation (O) (contd.)

$O(1) \Rightarrow O(\log n) \Rightarrow O(n) \Rightarrow O(n\log n) \Rightarrow O(n^2) \Rightarrow O(n^3) \Rightarrow O(2^n) \Rightarrow O(n!)$



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.
  - 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

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, $n \geq n_0$.
  - To be informative, $g(n)$ should be as large a function of n as possible for which the statement $f(n) = \Omega(g(n))$ is true.
- **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 $c_1 g(n) \le f(n) \le c_2 g(n)$ for all n, $n \ge n_0$.
- **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: p. 38, Figure 1.5
  - Since the number of lines is a constant, we may simply take the maximum of the line complexities as the asymptotic complexity of the function.

Figure 1.5

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++)`	$\Theta(rows \cdot cols)$
`c[i][j] = a[i][j] + b[i][j];`	$\Theta(rows \cdot cols)$
`}`	0
Total	$\Theta(rows \cdot cols)$