Basic Concepts

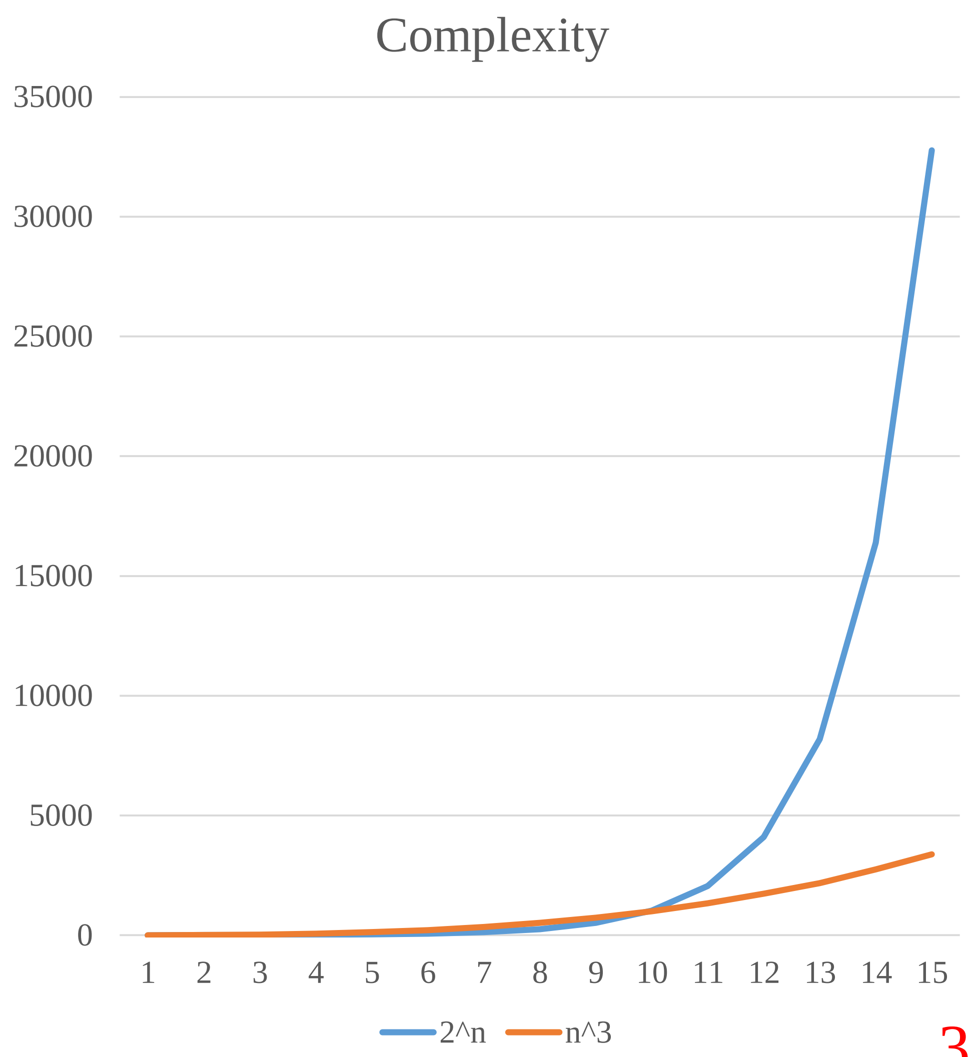
Fan-Hsun Tseng
Department of Computer Science and Information Engineering
National Cheng Kung University

Outline

- What is Data Structure
- System Life Cycle
- Data Abstraction and Encapsulation
- Algorithm Specification
- Performance Analysis and Measurement

What is Data Structure

- Data Structure + Algorithm = Program
- How to complete programs rapidly?
  - Mainframe / Supercomputer?
  - Expensive GPU with more cores?
  - A good algorithm? An appropriate data structure?
- Summary:
  - Data structure: the way to present data
  - Algorithm: the way to process data
  - Programming language: C/C++



Outline

- What is Data Structure
- System Life Cycle
- Data Abstraction and Encapsulation
- Algorithm Specification
- Performance Analysis and Measurement

What is System Life Cycle

- Good programmers regard large-scale computer programs as systems that contain many complex interacting parts.

- As systems, these programs undergo a development process called the system life cycle, includes five phases:
  - Requirements
  - Analysis
  - Design
  - Refinement and Coding
  - Verification

System Life Cycle

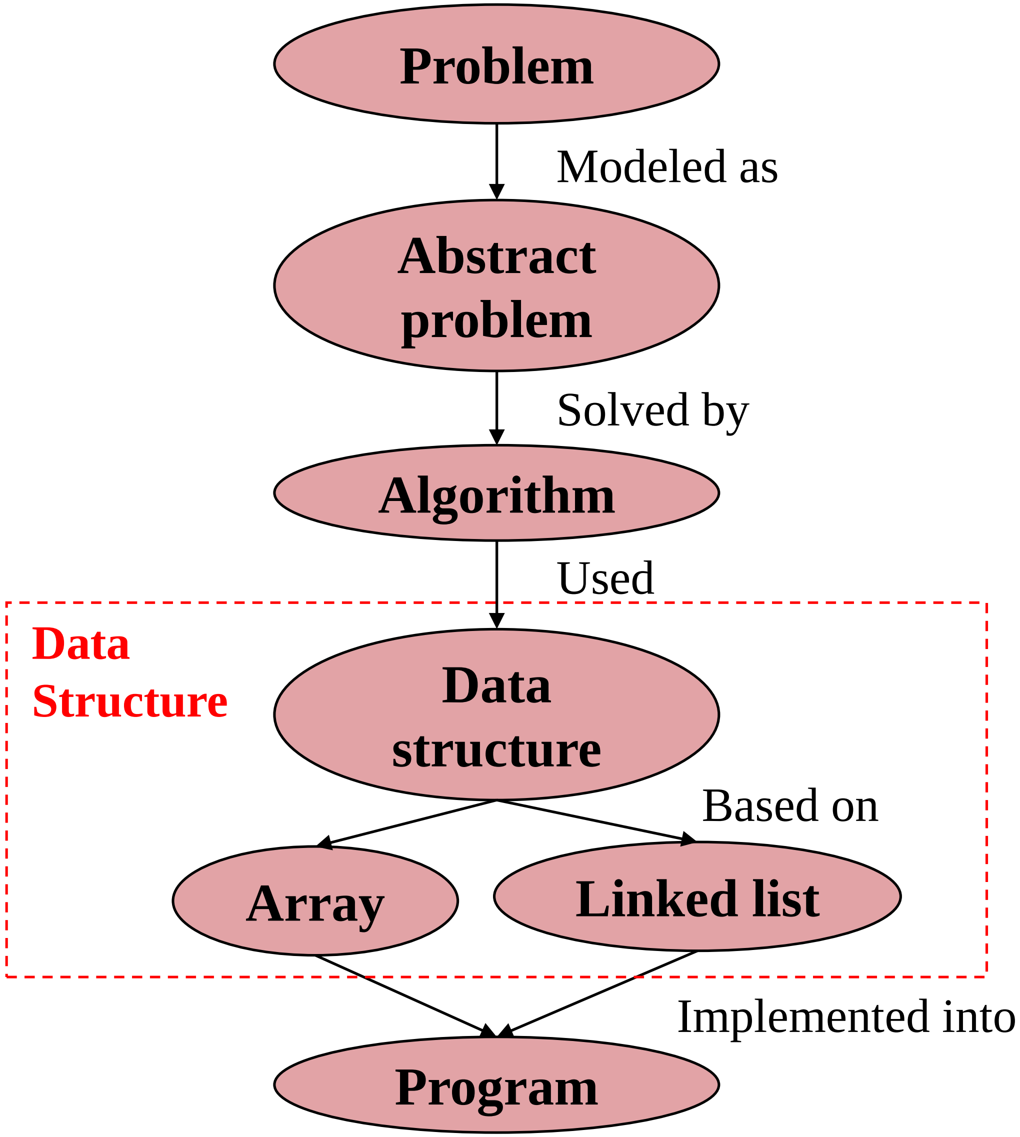
- Requirements
- Analysis:
  - bottom-up vs. top-down
- Design:
  - Data objects: abstract data types
  - Operations: specification & design of algorithms
- Refinement and Coding
  - Choose representations for data objects
  - Write algorithms for each operation on data objects
- Verification
  - Program proving: correctness proofs for the program
  - Testing: correctness & efficiency, testing with a variety of input data
  - Debugging: remove errors to achieve well-documented program

Evaluate Judgements about Programs

- Meet the original specification?
- Work correctly?
- Well-documented?
- Use functions to create logical units?
- Code readable?
- Use storage efficiently?
- Running time acceptable?

Role of Data Structure

- Real problem:
  - Ordering heights
  - Ranking scores
- Abstract problem:
  - Sorting problem
- Algorithm:
  - Bubble sort
  - Quick sort
- Data structure
  - Array
  - Linked list



Outline

- What is Data Structure
- System Life Cycle
- Data Abstraction and Encapsulation
- Algorithm Specification
- Performance Analysis and Measurement

Data Abstraction and Encapsulation

- *Data Encapsulation or Information Hiding* is the concealing of the implementation details of a data object from the outside world

- *Data Abstraction* is the separation between the *specification* of a data object and its *implementation*

- A data type is a collection of *objects* and a set of *operations* that act on those objects

Data Abstraction and Encapsulation (Contd.)

- A data type is a collection of *objects* and a set of *operations* that act on those objects
- 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 operations
- In other words, ADT is implementation-independent
  - Just know what it does, but NOT necessarily how it will do it

Data Abstraction and Encapsulation (Contd.)

- Specification
  - Name of function
  - Type of arguments
  - Types of result
  - Description of what the function does (without implementation details)

- Representation:
  - Implementation details
  - E.g., **char** 1 byte, **short** 2 bytes, **int** 4 bytes, **float** 4 bytes, **double** 8 bytes

Example 1.1 in the Textbook

```
Abstract data type NaturalNumber (p.9)
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 ∈ Nat_Number; TRUE, FALSE ∈ Boolean
    and where +, -, <, and == are the usual integer
    operations.
    Zero ( ):NaturalNumber      ::=  0
    Is_Zero(x):Boolean          ::= if (x) return FALSE
                                    else return TRUE
    Add(x, y):NaturalNumber     ::= if ((x+y) <= INT_MAX)
                                      return x+y
                                    else return INT_MAX
    Equal(x,y):Boolean          ::= if (x== y) return TRUE
                                    else return FALSE
    Successor(x):NaturalNumber  ::= if (x == INT_MAX)
                                      return x
                                    else return x+1
    Subtract(x,y):NaturalNumber ::= if (x<y) return 0
                                    else return x-y
end Natural_Number
```

Outline

- What is Data Structure
- System Life Cycle
- Data Abstraction and Encapsulation
- Algorithm Specification
- Performance Analysis and Measurement

Algorithm Specification

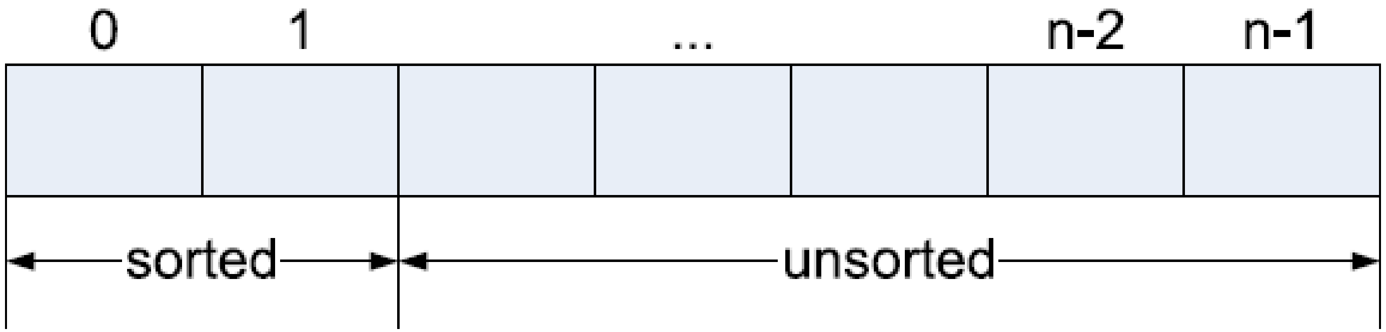
- An algorithm is a finite set of instructions that accomplishes a particular task.

- Criteria
  - Input: zero or more quantities that are externally supplied
  - Output: at least one quantity is produced
  - Definiteness: clear and unambiguous
  - Finiteness: terminate after a finite number of steps
  - Effectiveness: instruction is basic enough to be carried out

- One difference between an algorithm and a program is that the latter does not have to satisfy the fourth condition
  - A program does not have to satisfy the finiteness criteria
  - E.g., OS scheduling

Example 1: Selection Sort

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



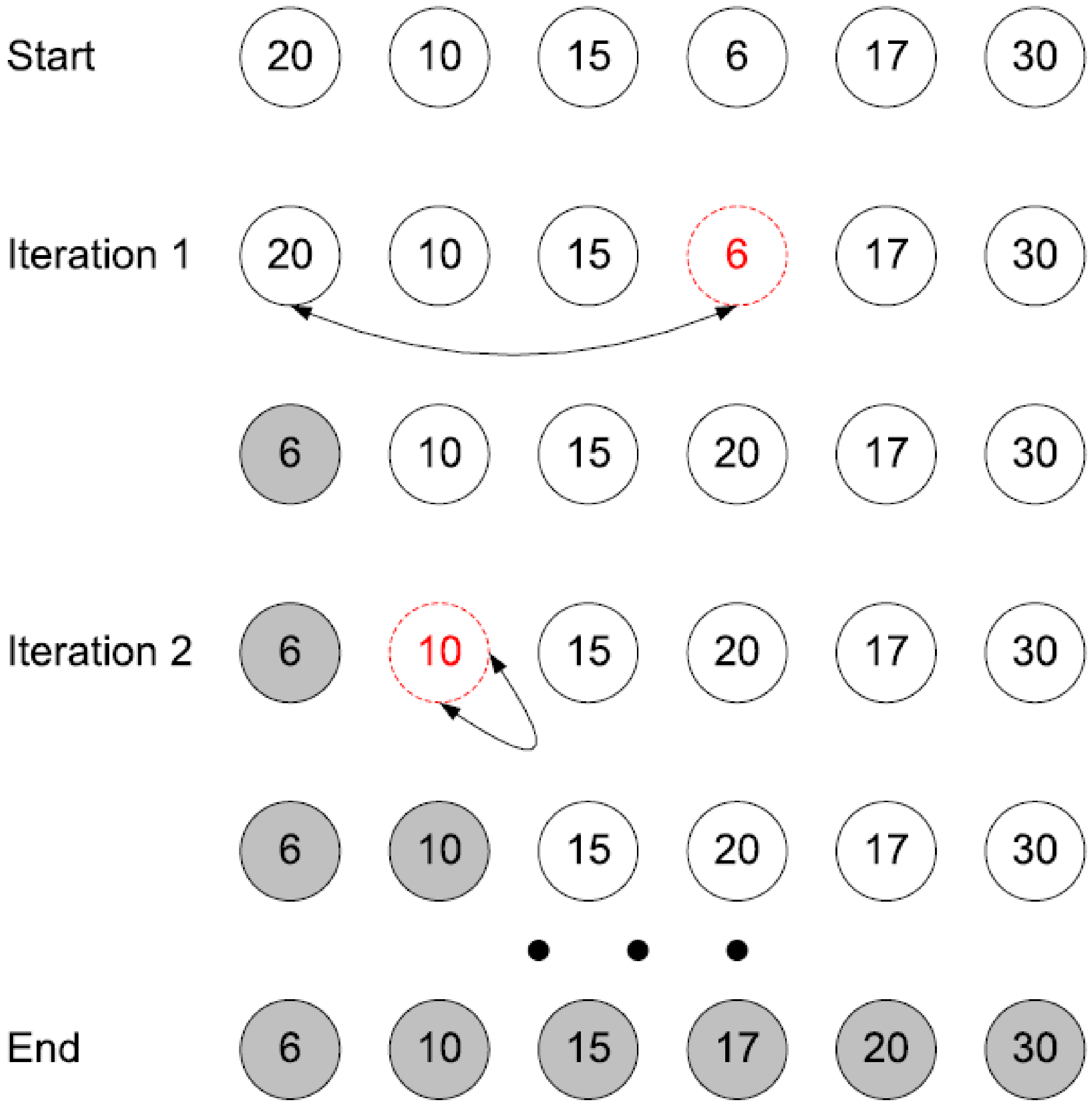

```
for ( i=0; i<n; i++) {
  examine list[i] to list[n-1] and suppose that smallest integer is list[min]
  interchange list[i] & list[min]
}
```

Example 1: Selection Sort (Contd.)

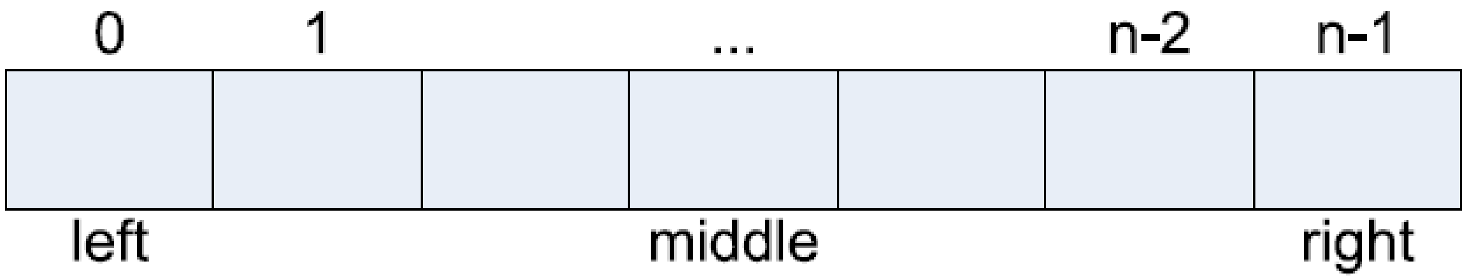
```
void sort(int list[ ], int n)
{
   for (i=0; i<n-1; i++)
  {
      int min = i;
      for (j=i+1; j<n; j++)
         if (list[j]<list[min])
            min=j;
         SWAP(list[i], list[min], temp);
   }
}
```

Example 1: Selection Sort (Contd.)

- Input
  - 20 10 15 6 17 30
- Iteration 1
  - Scan from list[0] to list[5]
  - The smallest one is 6
  - Swap 6 and list[0]
  - 6 10 15 20 17 30



Example 2: Binary Search




```
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;
}
```

Example 2: Binary Search (Contd.)

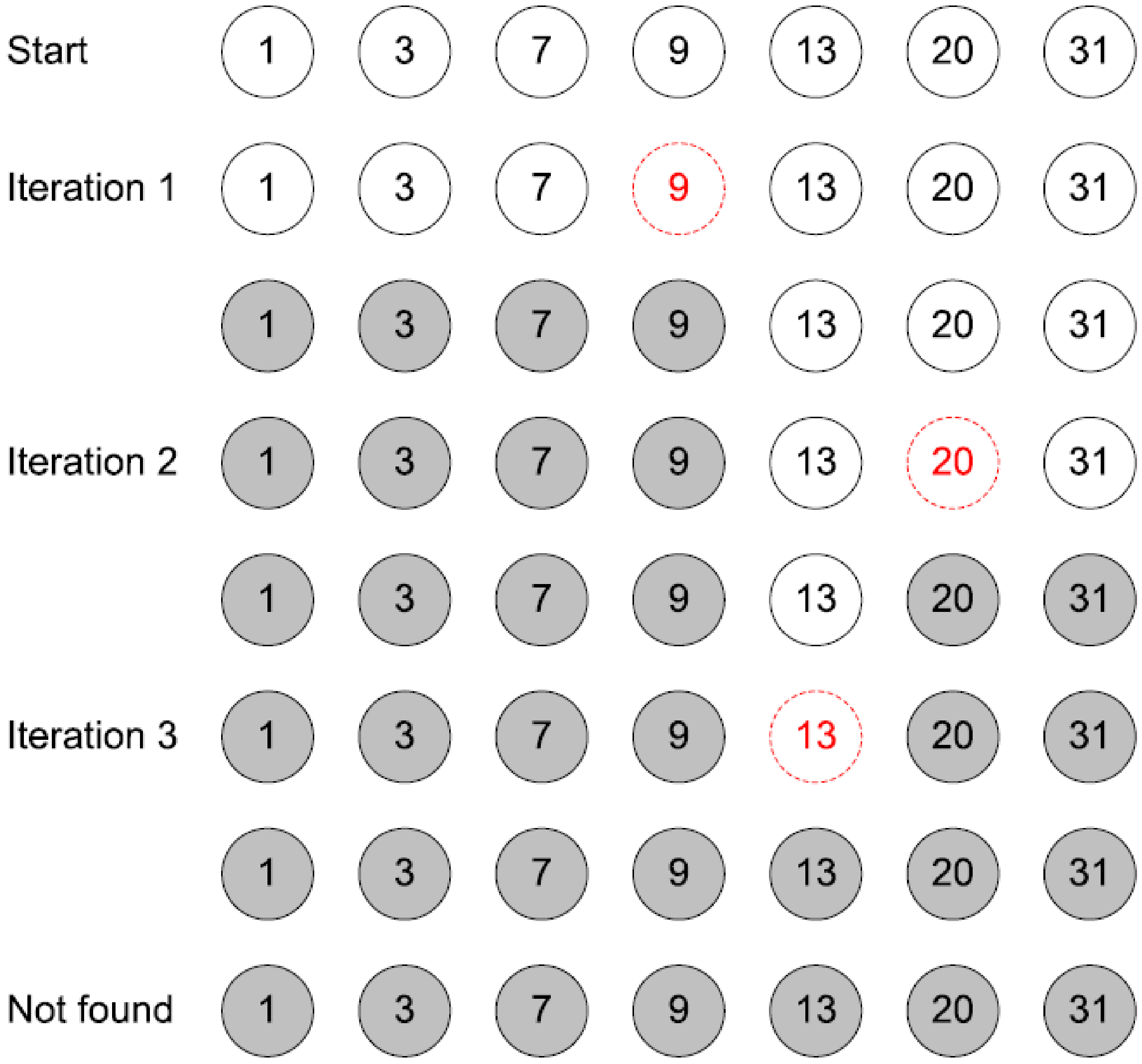
```
int compare (int x, int y) /* return -1 for less than, 0 for equal */
int binsearch(int list[], int searchno, int left, int right)
{
  while (left <= right) {
    middle = (left + right) / 2;
    switch ( COMPARE(list[middle], searchno) ) {
      case  -1:
        left = middle +1;
        break;
      case  0:
        return middle;
      case  1:
        right = middle -1;
      }
  }
}
```

Example 2: Binary Search (Contd.)

- Input
  - 1 3 7 9 13 20 31
- Search for 7

Example 2: Binary Search (Contd.)

- Search for 16



Example 2: Binary Search (Contd.)

- Comparison between sequential search and binary search
  - Binary search is faster than sequential search
  - However, binary search requires the input to be sorted in advance

- Should we always use binary search?
  - Not necessary

Example 3: Selection Problem

- Selection problem: select the k-th largest among N numbers
- Approach 1
  - Read N numbers into an array
  - Sort the array in decreasing order
  - Return the element in position k

- Approach 2
  - Read k elements into an array
  - Sort them in decreasing order
  - For each remaining elements, read one by one
  - Ignored if it is smaller than the k-th element
  - Otherwise, place in correct place and bumping one out of array

Example 3: Selection Problem (Contd.)

- Input
  - 30, 14, 9, 6, 22, 31
- Find the third largest number

- Read three numbers and sort them in descending order
  - 30, 14, 9
- Read next: “6”
  - 30, 14, 9
- Read next: “22”
  - 30, 22, 14
  - 9 has been kicked out
- Read next: “31”
  - 31, 30, 22
  - 14 has been kicked out
- The third largest number is 22

Example 3: Selection Problem (Contd.)

- Which one is better?
  - Implementation difficulty
  - Efficiency
    - Time complexity analysis

- Remember that time complexity is not the only yardstick
  - Space complexity
  - Easy to implement

Recursive Algorithms

- Recursion is usually used to solve a problem in a “divided-and-conquer” manner
- Direct recursion
  - Functions that call themselves before they are done
- Indirect recursion
  - Functions that call other functions that invoke calling function again
- $C(n,m) = n!/[m!(n-m)!]$
  - $C(n,m)= C(n-1,m-1) + C(n-1,m)$
- Boundary condition for recursion

Recursive Summation

- sum(1, n)=sum(1, n-1)+n
- sum(1, 1)=1

```
int sum(int n)
{
   if (n==1)
      return (1);
   else
      return(sum(n-1)+n);
}
```

Recursive Factorial

- $n! = n \times (n-1)!$
- $fact(n) = n \times fact(n-1)$
- $0! = 1$

```
int fact(int n)
{
   if ( n== 0)
      return (1);
   else
   return (n*fact(n-1));
}
```

Recursive Multiplication

- $a \times b = a \times (b-1) + a$
- $a \times 1 = a$

```
int mult(int a, int b)
{
   if ( b==1)
      return (a);
   else
      return(mult(a,b-1)+a);
}
```

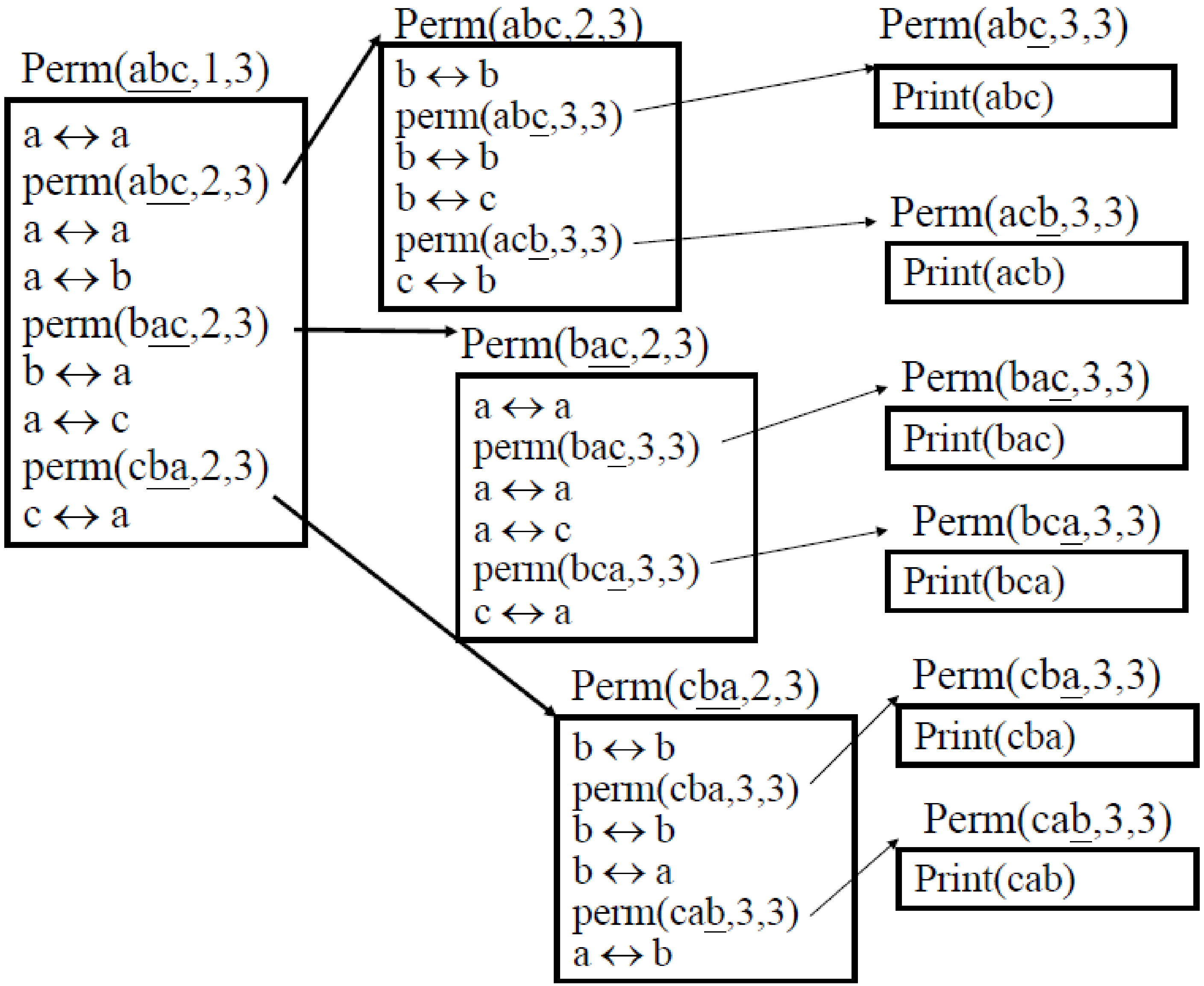
Recursive Binary Search

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

Recursive Permutations

- Permutation of {a, b, c}
  - (a, b, c), (a, c, b)
  - (b, a, c), (b, c, a)
  - (c, a, b), (c, b, a)

- Recursion?
  - a+Perm({b,c})
  - b+Perm({a,c})
  - c+Perm({a,b})



Recursive Permutations (Contd.)

```
void perm(char *list, int i, int n)
{
   if (i==n) {
      for (j=0; j<=n; j++)
         printf("%c", list[j]);
   }
   else {
      for (j=i; j<= n; j++) {
         SWAP(list[i], list[j], temp);
         perm(list, i+1, n);
         SWAP(list[i], list[j], temp);
      }
   }
}
```

Outline

- What is Data Structure
- System Life Cycle
- Data Abstraction and Encapsulation
- Algorithm Specification
- Performance Analysis and Measurement

Performance Evaluation

- Criteria
  - Does the program do what we want it to do?
  - Is the code correct?
  - Is the code readable?

- Performance analysis
  - Machine independent
- Performance measurement
  - Machine dependent

Performance Analysis

- Paper and pencil.

- Don’t need a working computer program or even a computer.

- Some uses of performance analysis
  - determine practicality of algorithm
  - predict run time on large instance
  - compare 2 algorithms that have different asymptotic complexity
  - e.g., $O(n)$ and $O(n^2)$

Performance Analysis (Contd.)

- Complexity theory

- Space complexity
  - Amount of memory

- Time complexity
  - Amount of computing time

Space Complexity

- Space requirement: $S(P) = c + S_p(I)$
  - P: any program
  - c: constant (fixed space, e.g., instruction, simple variables, constants)
  - $S_p(I)$: depends on characteristics of instance I
  - Characteristics: number, size, values of I/O associated with I

- If n is the only characteristic, $S_p(I) = S_p(n)$

Space Complexity (Contd.)

- *Program 1.16: (p.38)*

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

$S_{Abc}(n) = 0$

The space needed by function Abc is independent of the instance characteristics

- *Program 1.17 (p.39)*

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

$S_{Sum}(n) = 0$

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

- n is passed by value, needs 1 word
- a is actually the address of the first element in a[](i.e. a[0]), the space needs 1 word

Space Complexity (Contd.)

- *Program 1.18 (p.39)*

float *Rsum*(float **a*, const int *n*)

{

   if (*n*<=0) return 0;

   else return (*Rsum*(*a*, *n*-1) + *a*[*n*-1]);

}

Assumptions:

Space needed for one recursive call of the program

$$S_{Rsum}(I) = S_{Rsum}(n) = 4(n+1)$$

Instance	Stack Space
n	1 word
a	1 word
Returned value	1 word
Return address	1 word
Depth of recursion	n+1
Total	**4(n+1)**
Ex: n=1000	**4004 recursion stack space**

Time Complexity

- $T(P) = c + T_p(I)$
  - c: compile time
  - $T_p(I)$: program execution time
    - Depends on characteristics of instance I
- Predict the growth in run time as the instance characteristics change

- Compile time (c)
  - Independent of instance characteristics
- Run (execution) time T_p
- A program step is a syntactically or semantically meaningful program segment whose execution time is independent of the instance characteristics

Methods to Compute the Step Count

- Introduce variable count into programs
- Tabular method
- Determine the total number of steps contributed by each statement step per execution × frequency
- Add up the contribution of all statements

Time Complexity (Contd.)

- *Program 1.17 (p.39)*

```
0 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	Step per execution (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**

Time Complexity (Contd.)

- *Program 1.22 (p.46)*

```
0 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; 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			**$2mn+2m+1$**

Time Complexity (Contd.)

- Cases
  - Worst case
  - Best case
  - Average case

Time Complexity (Contd.)

- Worst case and average case analysis is much more useful in practice
- Difficult to determine the exact step counts
- What a step stands for is inexact
  - e.g. x := y v.s. x := y + z + (x/y) + … (both count as one step)
- Exact step count is not useful for comparison
- Step count doesn’t tell how much time step takes
- Just consider the growth in run time as the instance characteristics change

Asymptotic Notations

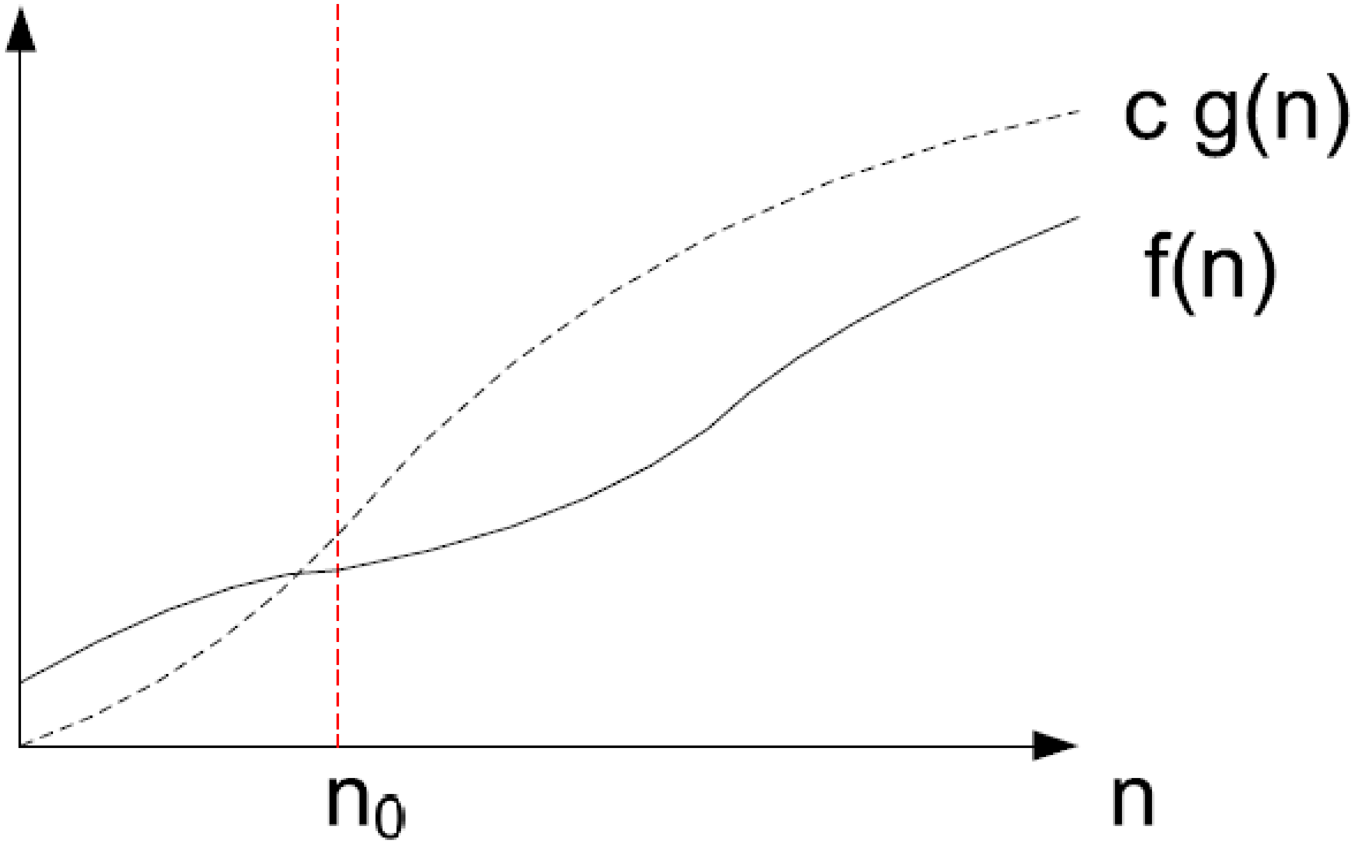
- Big "oh": O
  - $f(n) \le cg(n)$

- Omega: Ω
  - $f(n) \ge cg(n)$

- Theta: Θ
  - $c_1 g(n) \le f(n) \le c_2 g(n)$

Big "oh": O

- $f(n)$=O($g(n)$) if and only if (iff)
  - $\exists$ a real constant $c>0$ and an integer constant $n_0 \geqq 1$, $\ni f(n) \leq cg(n)\ \forall n, n \geq n_0$

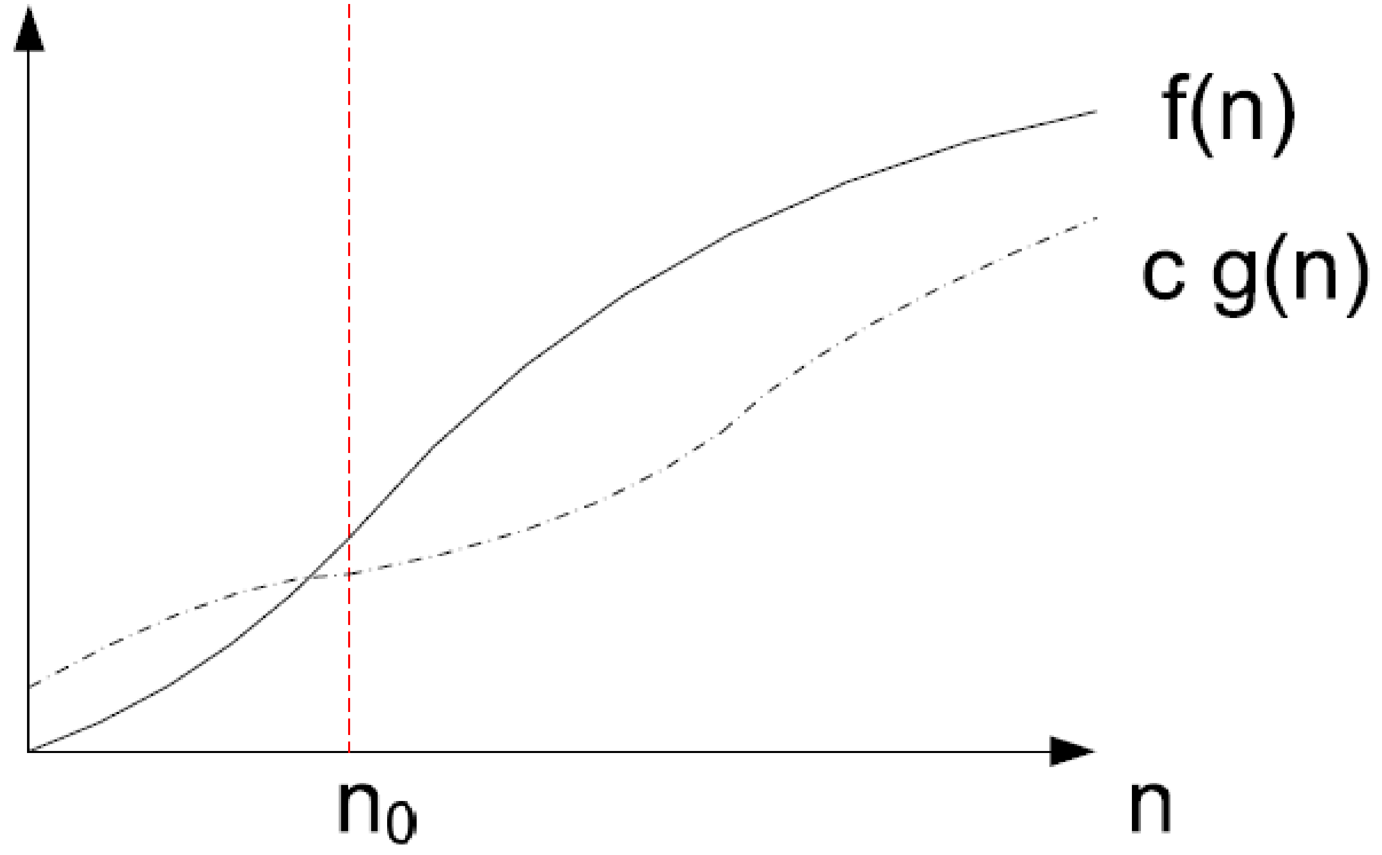


Big "oh": O (Contd.)

- Examples
  - $3n+2 = O(n)$
    $3n+2 \leq 4n$ for all $n \geq 2$
  - $10n^2+4n+2 = O(n^2)$
    $10n^2+4n+2 \leq 11n^2$ for all $n \geq 10$
  - $3n+2 = O(n^2)$
    $3n+2 \leq n^2$ for all $n \geq 4$ → Not tight enough
- $g(n)$ should be a least upper bound

Omega: Ω

- $f(n) = \Omega(g(n))$ iff
  - $\exists$ a real constant $c > 0$ and an integer constant $n_0 \geqq 1$, $\ni f(n) \geq cg(n)\ \forall n,\ n \geq n_0$

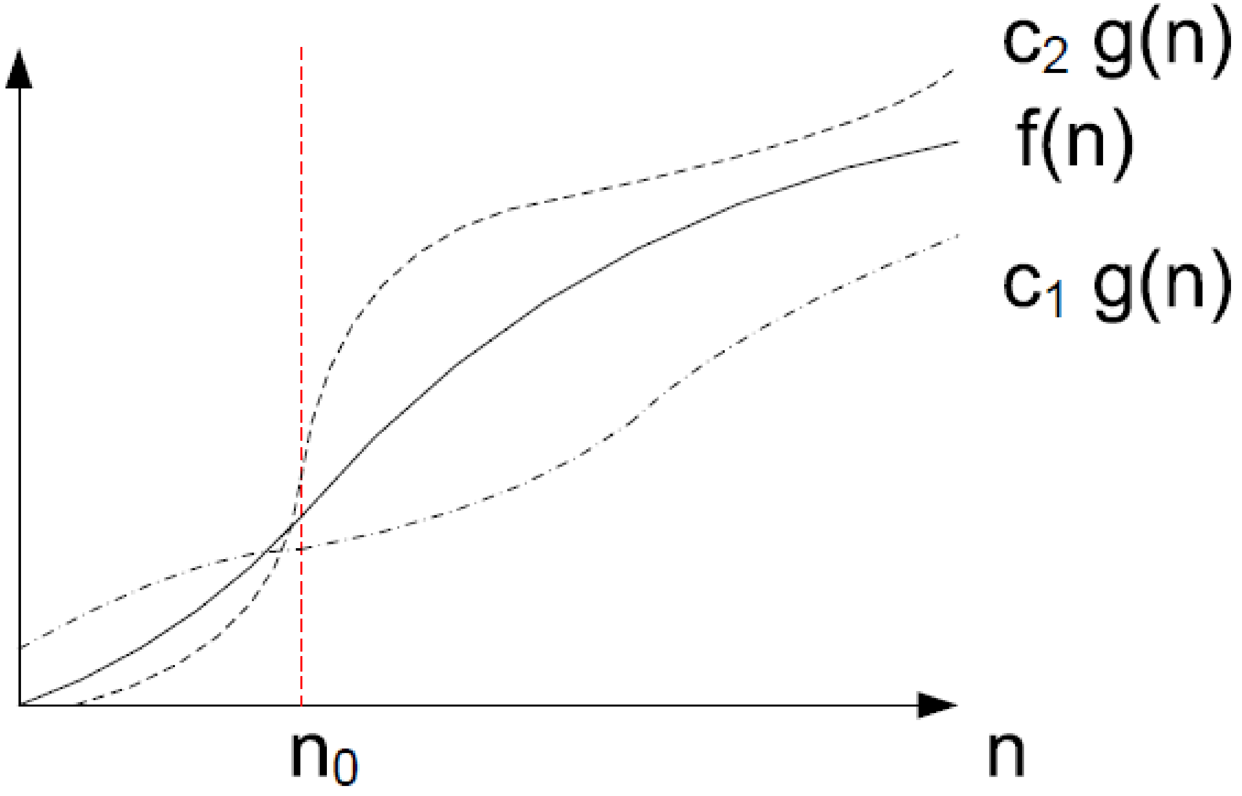


Omega: Ω (Contd.)

- Examples
  - $3n+3 = \Omega(n)$
    $3n+3 \geq 3n$ for all $n \geq 1$
  - $6*2^n+n^2 = \Omega(2^n)$
    $6*2^n+n^2 \geq 2^n$ for all $n \geq 1$
  - $3n+3 = \Omega(1)$
    $3n+3 \geq 3$ for all $n \geq 1$
- g(n) should be a most lower bound

Theta: Θ

- $f(n)=\Theta(g(n))$ iff
  - $\exists$ two positive real constants $c_1,c_2>0$, and an integer constant $n_0 \geqq 1$, $\ni c_1 g(n) \le f(n) \le c_2 g(n)\ \forall n,\ n \ge n_0$



Theta: Θ (Contd.)

- Examples
  - $3n+2 = \Theta(n)$

    $3n \le 3n+2 \le 4n$, for all $n \ge 2$
  - $10n^2+4n+2 = \Theta(n^2)$

    $10n^2 \le 10n^2+4n+2 \le 11n^2$, for all $n \ge 5$
- g(n) should be both lower bound & upper bound

Time Complexity of Some Examples

- For loop

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

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

Time Complexity of Some Examples (Contd.)

- Nested for loops

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

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

Time Complexity of Some Examples (Contd.)

- Consecutive statements

```
for (i=0; i<n; i++)
   A[i]=0;
for (i=0; i<n; i++)
   for (j=0; j<n; j++)
      A[i]+=A[j]+i+j
```

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

Time Complexity of Some Examples (Contd.)

- If/Else

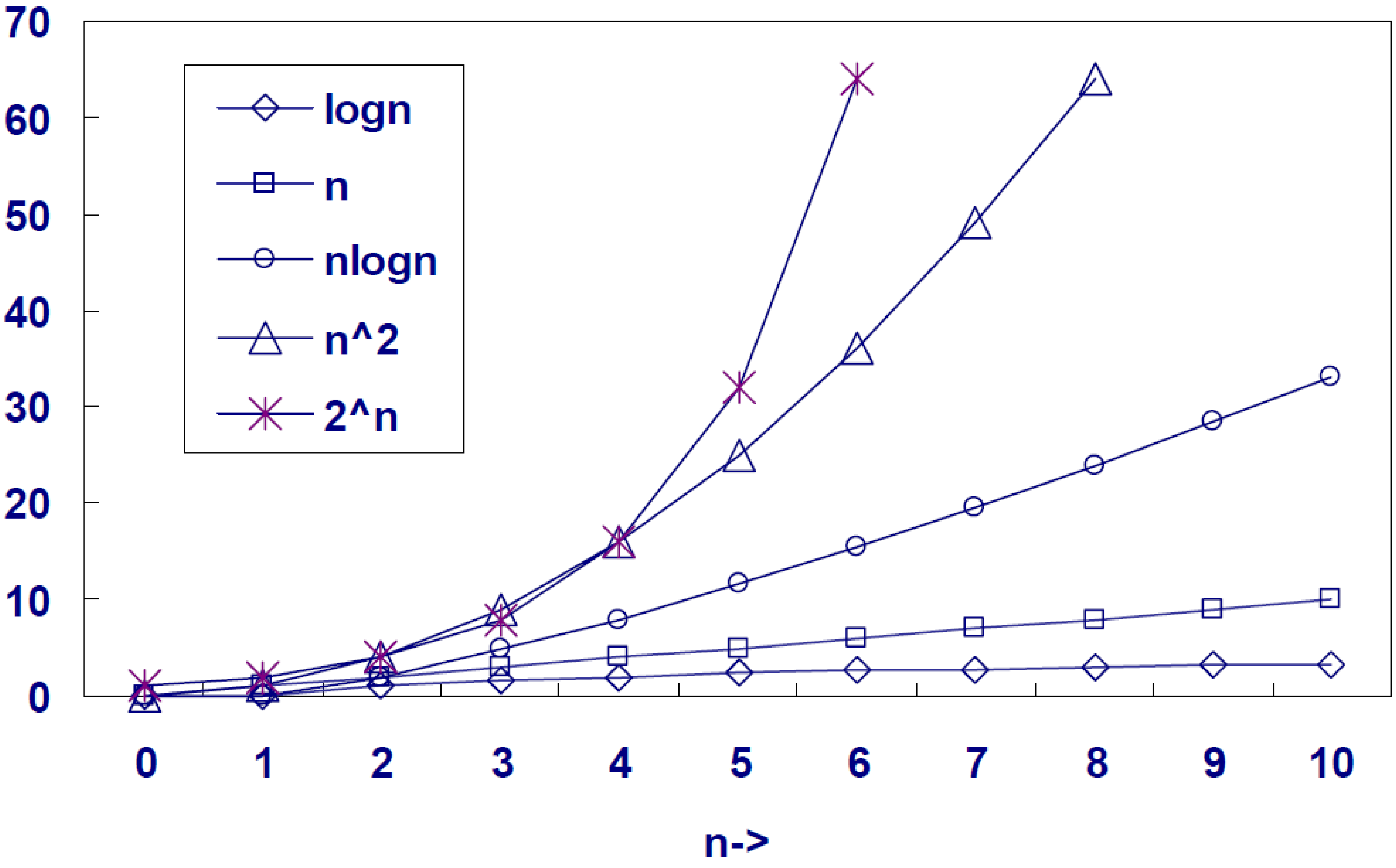
```
if (i>0) {
    i++;
    j++;
}
else {
    for (j=0; j<n; j++)
        k++;
}
```

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

Typical Growth Rate

- c: constant
- $\log \log n$: doubly log
- $\log n$: logarithmic
- $\log^2 n$: Log-squared, as well as $(\log(n))^2$
- n: Linear
- $n\log n$
- n^2: Quadratic
- n^3: Cubic
- 2^n: Exponential

Asymptotic Complexity



Asymptotic Complexity (Contd.)

- Comparison table of execution time between different complexities

$\log n$	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	4,096	65,536
5	32	160	1024	32,768	4,294,967,296

Performance Measurement

- Timing event
- In C's standard library time.h
  - Clock function: system clock
  - Time function