Introduction to Data Structures

Last modified: 9/20/2016

Basic Concepts

- System life cycle
- Algorithm specification
- Data abstraction
- Performance analysis & measurement

Overview: System Life Cycle

- Requirements
- Analysis
  - Bottom-up
  - Top-down
- Design
  - Data objects: abstract data types
  - Operations: specification & design of algorithms

Overview: System Life Cycle (Cont.)

- Coding & Refinement
  - Choose representations for data objects
  - Write algorithms for each operation on data objects
- Verification
  - Correctness proofs: selecting proved algorithms
  - Testing: correctness & efficiency
  - Error removal: well-document

Evaluative judgments about programs

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

Data Abstraction

- Predefined & user defined data type

```
* Struct student {      char last_name;
                        int student_id;
                        char grade; }
```

- Data type: objects & operations

  * integer: +, -, *, /, %, =, ==, atoi()

Data Abstraction (Cont.)

- Representation: char 1 byte, int 4 bytes
- Abstract Data Type (ADT): data type specification(object & operation) is separated from representation.
- ADT is implementation-independent

Abstract data type **Natural_Number** *(p.9)*

ADT ***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**

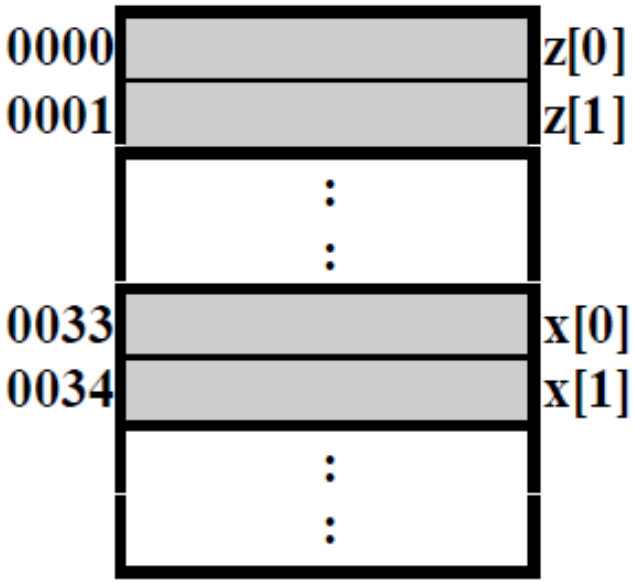
and where +, -, <, ***and*** == ***are the usual integer operations.***

```
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
```

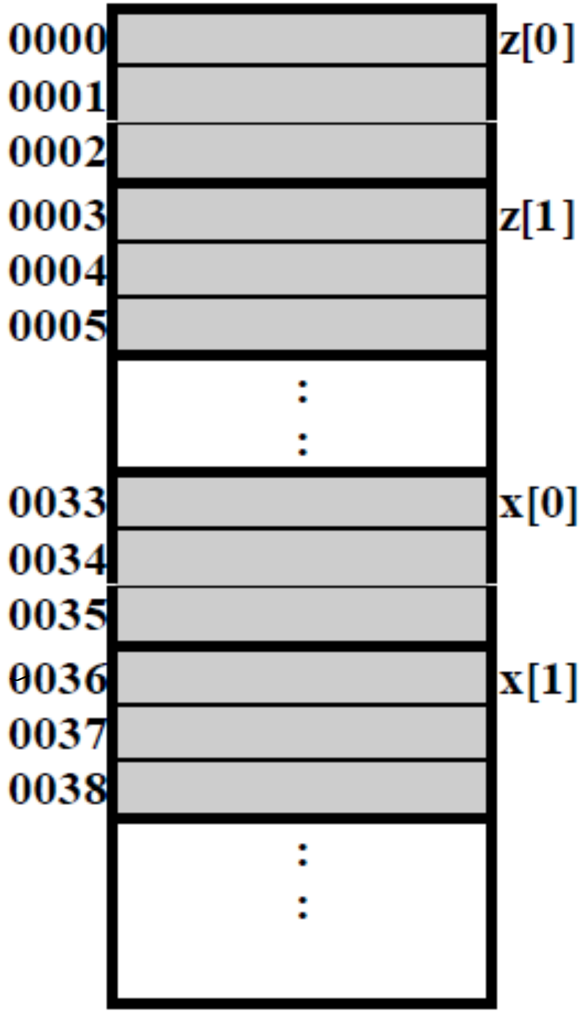
end **Natural_Number**

::= is defined as

```
char x[2], z[2];
for (i=0; i<2; i++)
    z[i]=x[i];
```




```
int x[2], z[2];
for (i=0; i<2; i++)
    z[i]=x[i];
```



Data Abstraction (Cont.)

- Specification
  - name of function
  - type of arguments
  - types of result
  - description of what the function does (without implementation detail)

Algorithm Specification

- Algorithm criteria
  - Input
  - Output
  - Definiteness
  - Finiteness
  - Effectiveness
    - program doesn’t have to be finite (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 (Cont.)

```
void sort(int  list[ ], int n)
{
    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);
        }
}
```

Example of Selection Sort

	A[0]	A[1]	A[2]	A[3]	A[4]	A[5]
Original	34	8	64	51	32	21
after pass 0	8	34	64	51	32	21
after pass 1	8	21	64	51	32	34
after pass 2	8	21	32	51	64	34
after pass 3	8	21	32	34	64	51
after pass 4	8	21	32	34	51	64

Example of Selection Sort (Cont.)

- Detailed (for example, doing pass 3 after pass 2)

	A[0]	A[1]	A[2]	A[3]	A[4]	A[5]
Original	34	8	64	51	32	21
after pass 0	8	34	64	51	32	21
after pass 1	8	21	64	51	32	34
after pass 2	8	21	32	51	64	34
doing pass 2	8	21	32	51	64	34 minimum
					exchange	
after pass 3	8	21	32	34	64	51
after pass 4	8	21	32	34	51	64

# of executions: n * (n-1)

Example of Binary Search

```
Enter a number between 0 and 28: 6
------------------------------------------------------------
 0  2  4  6  8 10 12 14* 16 18 20 22 24 26 28
 0  2  4  6*  8 10 12
6 found in array element 3

Enter a number between 0 and 28: 25
------------------------------------------------------------
 0  2  4  6  8 10 12 14* 16 18 20 22 24 26 28
                        16 18 20 22* 24 26 28
                                     24 26* 28
                                     24*
25 not found
```

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 (Cont.)

```
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 3: Selection Problem

- Selection problem: select the *k*-th largest among *N* numbers
- Solutions
  - Approach 1
    - read N numbers into an array
    - sort the array in decreasing order
    - return the element in position k

Example 3: Selection Problem (cont.)

- Solutions
  - 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 <u>in correct place </u>and bumping one out of array
- Which is better?
- More efficient algorithm?

Recursive Algorithms

- Direct recursion: functions that call themselves
- Indirect recursion: Functions that call other functions that invoke calling function again
- $C(n,m) = n!/[m!(n-m)!]$

  $\Rightarrow C(n,m)=C(n-1,m)+C(n-1,m-1)$
- <u>Boundary condition for recursion</u>

Recursive Factorial

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

  $0! = 1$

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

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

$\underline{4 * fact(3)}$

$4 * \underline{3 * fact(2)}$

$4 * 3 * \underline{2 * fact(1)}$

$4 * 3 * 2 * \underline{1 * fact(0)}$

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 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 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

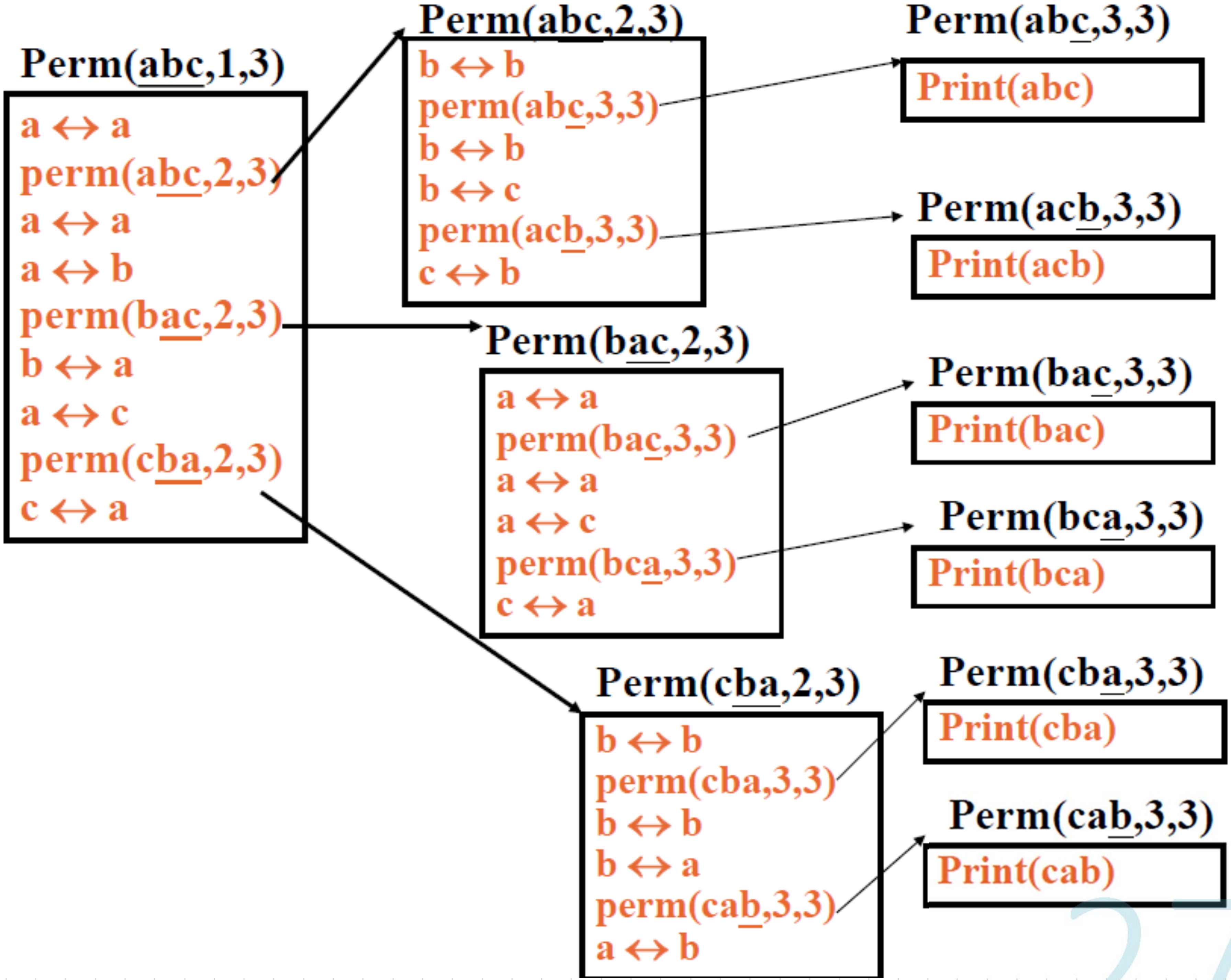
- Permutation of {a, b, c}

  (a, b, c), (a, c, b)

  (b, a, c), (b, c, a)

  (c, a, b), (c, b, a)

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


Perm(abc,1,3)
a ↔ a
perm(abc,2,3)
a ↔ a
a ↔ b
perm(bac,2,3)
b ↔ a
a ↔ c
perm(cba,2,3)
c ↔ a
Perm(abc,2,3)
b ↔ b
perm(abc,3,3)
b ↔ b
b ↔ c
perm(acb,3,3)
c ↔ b
Perm(bac,2,3)
a ↔ a
perm(bac,3,3)
a ↔ a
a ↔ c
perm(bca,3,3)
c ↔ a
Perm(cba,2,3)
b ↔ b
perm(cba,3,3)
b ↔ b
b ↔ a
perm(cab,3,3)
a ↔ b
Perm(abc,3,3)
Print(abc)
Perm(acb,3,3)
Print(acb)
Perm(bac,3,3)
Print(bac)
Perm(bca,3,3)
Print(bca)
Perm(cba,3,3)
Print(cba)
Perm(cab,3,3)
Print(cab)

Recursive Permutations (cont.)

```
void perm(char *list, int i, int n)
{
   if ( i == n) {
      for (j=0; j <=n; j++)
             cout<<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);
      }
    }
}
```

Performance Evaluation

- Performance analysis: machine independent
- Performance measurement: machine dependent

Performance Analysis

- Complexity theory
  - Space complexity: amount of memory
  - Time complexity: amount of computer time

Space Complexity

- $S(P) = c + Sp(I)$

  c: fixed space(instruction, simple variables, constant

  Sp(I): depends on characteristics of instance I

  Characteristics: number, size, values of I/O associated with I

  * if n is the only characteristic, Sp(I) ⇒ Sp(n)

Time Complexity

- $T(P) = c + T_p(I)$

c: compile time (or constant time)

$T_p(I)$: program execution time depends on characteristics of instance I

Characteristic: number, size, values of I/O associated with I

* predict the growth in run time as the instance characteristics change

Time Complexity (cont'd)

- Compile time (C) independent of instance characteristics
- Run (execution) time TP

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

- Definition

A **program step** is a syntactically or semantically meaningful program segment whose execution time is independent of the instance characteristics.

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

$2n + 3$ steps

Tabular Method

Table 1.1: Step count table for Program Sum

steps/execution

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

Recursive summing of a list of numbers

```
float rsum(float list[ ], int n)
{
        count++;        /*for if conditional */
        if (n) {
        count++;  /* for return and rsum invocation */
                return rsum(list, n-1) + list[n-1];
        }
        count++;        /* last execution for if */
        return list[0];
}
```

$\boxed{2n+2}$

Matrix addition

```
void add(matrix a, matrix b, matrix c, int rows, 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];
 }
```

```
void add(matrix a, matrix b, matrix c, int row, int cols )
{
  int i, j;
  for (i = 0; i < rows; i++)
   {
      count++; /* for i for loop */
      for (j = 0; j < cols; j++)
      {
        count++; /* for j for loop */
        c[i][j] = a[i][j] + b[i][j];
        count++; /* for assignment statement */
      }
      count++;    /* last time of j for loop */
   }
  count++;        /* last time of i for loop */
}
```

$$\boxed{2(rows * cols) + 2\ rows + 1}$$

Matrix Addition

Step count table for matrix addition

Statement	s/e	Frequency	Total steps
Void add (int a[	0	0	0
][MAX_SIZE]•••)	0	0	0
{	0	0	0
int i, j;	1	rows+1	rows+1
for (i = 0; i < row; i++)	1	rows•(cols+1)	rows•cols+rows
for (j=0; j< cols; j++)	1	rows•cols	rows•cols
c[i][j] = a[i][j] + b[i][j];	0	0	0
}			
Total	2rows * cols + 2 rows + 1		

```
void add(matrix a, matrix b, matrix c, int row, int
cols ) {
   int i, j;
   for( i = 0; i < rows; i++) {
      for (j = 0; j < cols; j++)
         count += 2;
         count += 2;
   }
   count++;
}
```

$2(rows \times cols) + 2rows + 1$

Suggestion: Interchange the loops when rows >> cols

Time Complexity (cont'd)

- Worst case
- Best case
- Average case

Time Complexity (cont'd)

- 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) + ...

- exact step count is not useful for comparison
  - Step count doesn't tell how much time step takes
- break-even point $(n^2 + 2n)$ v.s. $(10n)$

Asymptotic Notation -Big "oh"

- $f(n)=O(g(n))$ iff
  - $\exists$ positive const. c and n_0, $\ni f(n) \leq cg(n)$ $\forall n, n \geq n_0$
- e.g.
  - $3n+2 = O(n)$ $3n+2 \leq 4n$ 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 $n \geq 4$

* $g(n)$ should be a ***least upper bound***

Asymptotic Notation -Omega

- $f(n)=\Omega(g(n))$ iff
  - $\exists$ positive const. c and n_0, $\ni$ $f(n) \geq cg(n)$ $\forall$ n, $n \geq n_0$
  - e.g.
    - $3n+3 = \Omega(n)$ $\quad 3n+3 \geq 3n$ for all $n \geq 1$
    - $6*2^n + n^2 = \Omega(2^n)$ $\quad 6*2^n + n^2 \geq 2^n$ for all $n \geq 1$
    - $3n+3 = \Omega(1)$ $\quad 3n+3 \geq 3$ for all $n \geq 1$

* g(n) should be a *most lower bound*

Asymptotic Notation -Theta

- $f(n)=\Theta(g(n))$ iff
  - $\exists$ positive constants c1,c2, and n0 $\ni$ $c1g(n) \leq f(n) \leq c2g(n)$ $\forall$ n, $n \geq n0$
  - e.g.
    - **$3n+2 = \Theta(n)$** **$3n \leq 3n+2 \leq 4n$, for all $n \geq 2$**
    - **$10n^2+4n+2= \Theta(n^2)$** **$10n^2 \leq 10n^2+4n+2 \leq 11n^2$, for all $n \geq 5$**

* g(n) should be both *lower bound & upper bound*

Some Rules

- Rule 1:

  If $T1(N)=O(f(N))$ and $T2(N)=O(g(N))$ Then

  (a) $T1(N)+T2(N) = \max(\, O(f(N)),\, O(g(N))\,)$

  (b) $T1(N)*T2(N) = O(\, f(N)*g(N)\,)$

- Rule 2:

  If $T(N)$ is a polynomial of degree k, then $T(N)= \Theta(N^k)$

- Rule 3:

  $(\log N)^k=O(N)$ (Prove it in our discussion board)

Running Time Calculations

- for loops
  - ```
    for (I=0; I <n; I++)
    {   x++;
        y++;
        z++;
    }
    ```
  - n*3
    ```

Running Time Calculations (cont'd)

- nested for loops
  - ```
    for (i=0; i <N; i++)
        for (j=0; j<N; j++)
            k++;
    ```
  - 1*N*N
    ```

Running Time Calculations (cont'd)

- 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*N, 1*N*N)= 1*N*N
```

Running Time Calculations (cont'd)

- If/Else

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

- max(2, 1*N)= 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

Running Time Calculations-Recursive

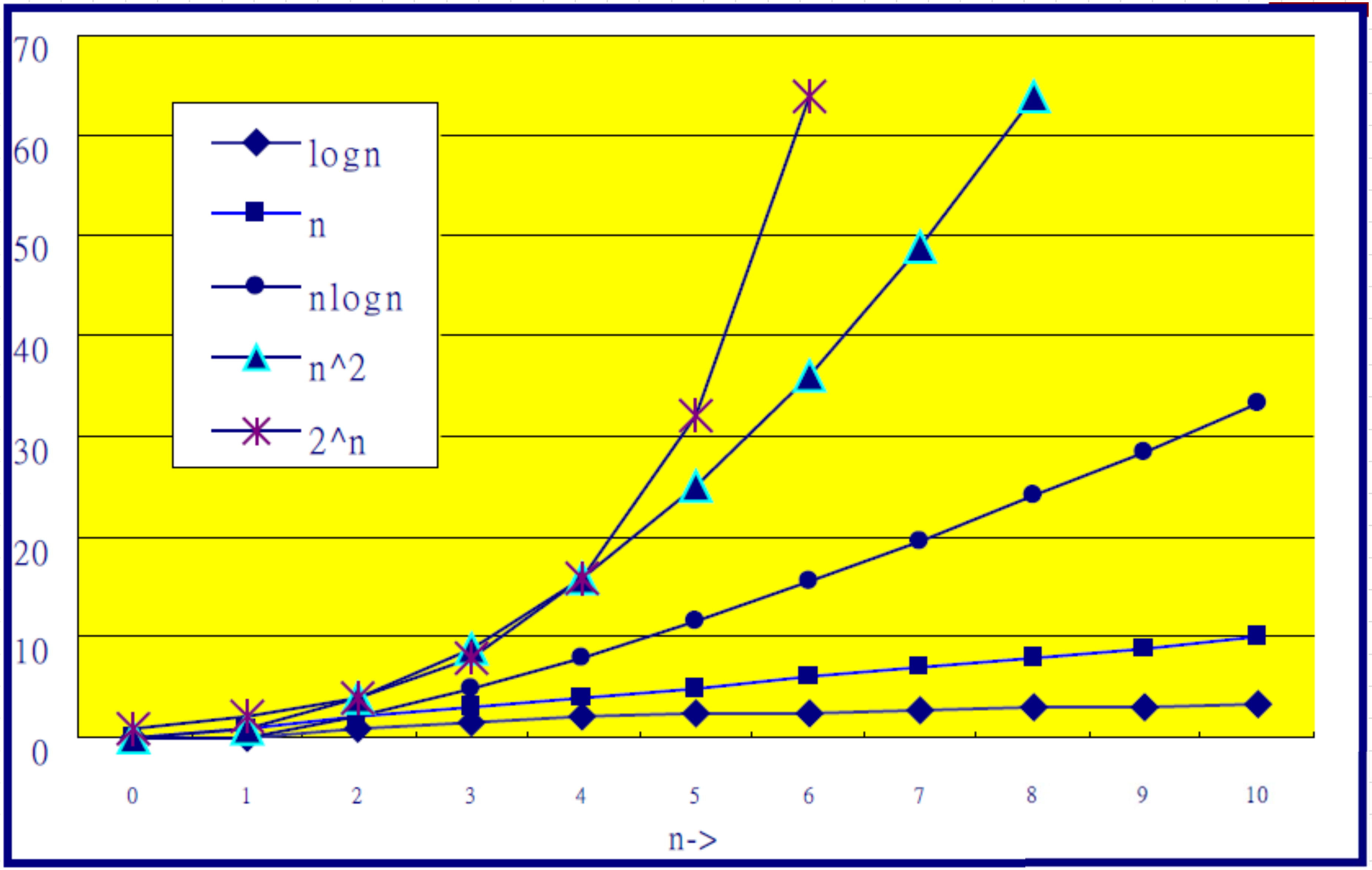
- long int Fb(int N)

```
long int Fb(int N)
{
    if (N<=1)
        return 1;
    else
        return Fb(N-1)+Fb(N-2);
}
```

- $T(N)=T(N-1)+T(N-2)+c$

Typical Growth Rate

- c: constant
- log N: logarithmic
- $\log^2 N$: Log-squared
- N: Linear
- NlogN:
- N^2: Quadratic
- N^3: Cubic
- 2^N: Exponential



70
60
50
40
30
20
10
0
logn
n
nlogn
n^2
2^n
0
1
2
3
4
5
6
7
8
9
10
n->

Performance Measurement

- Timing event
- in C's standard library time.h
  - clock function: system clock
  - time function