DATA STRUCTURE

Lecture 03: Recursion, Algorithm Analysis. (Examples)

Spring 2022

Hsiao-chi Li 李曉祺

RECURSIVE ALGORITHMS

- Recursion is usually used to solve a problem in a *divided-and-conquer* manner
- Direct Recursion
  - Functions that call themselves
- Indirect Recursion
  - Functions that call other functions that invoke calling function again
- $C\binom{n}{m} = \frac{n!}{m!(n-m)!}$
  - $C\binom{n}{m} = C\binom{n-1}{m} + C\binom{n-1}{m-1}$
- Boundary condition for recursion

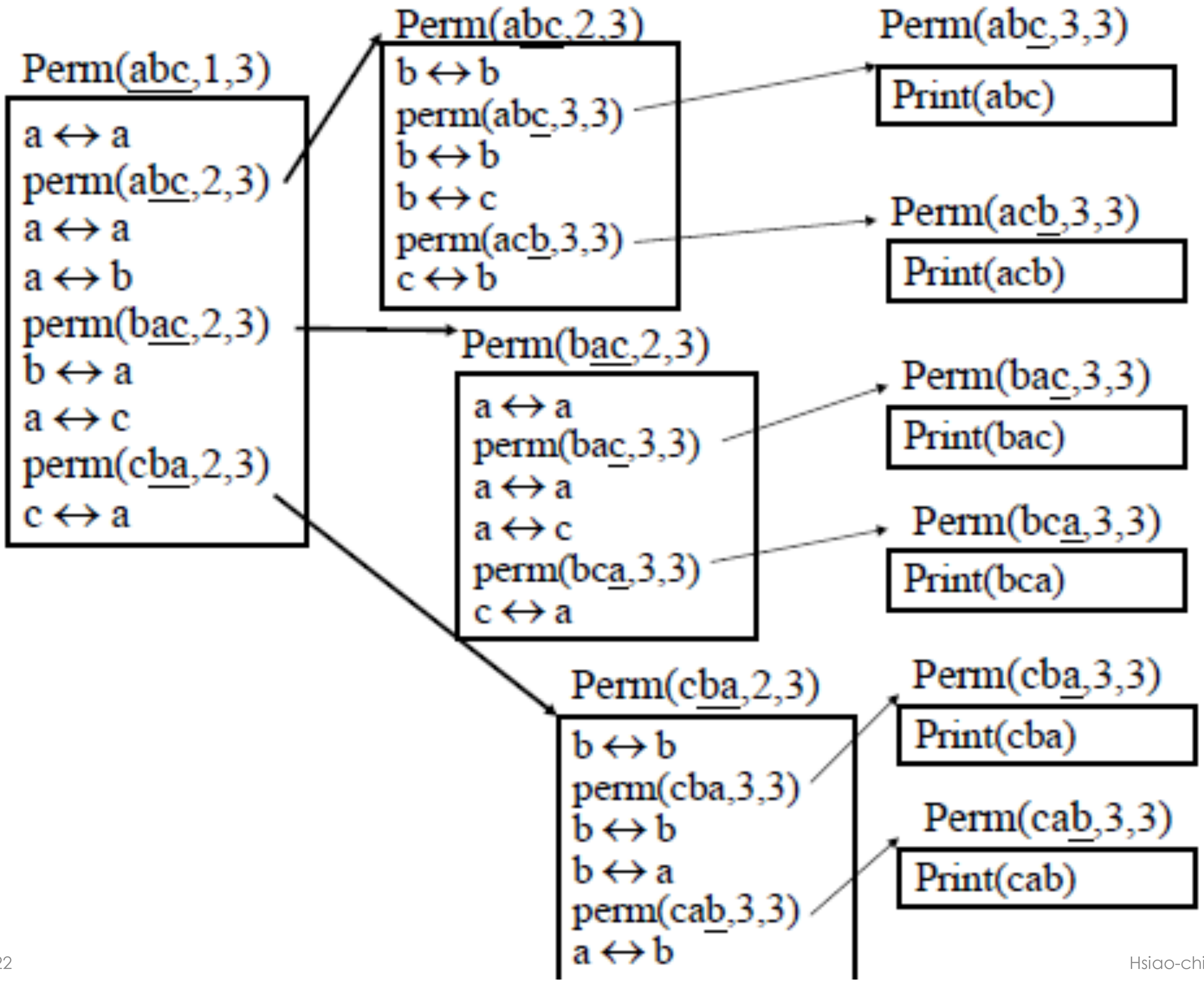
RECURSIVE FACTORIAL

- $n! = n\ (n\text{-}1)!$
- factorial(n) = n×factorial(n-1)
- $0! = 1$

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

RECURSIVE PERMUTATION

- 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})


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 PERMUTATION (CONT'D.)

```
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);
    }
  }
}
```

PERFORMANCE EVALUATION

- Criteria
  - Is it correct?
  - Is it readable?
- Performance analysis
  - Machine Independent
- Performance measurement
  - Machine dependent

PERFORMANCE ANALYSIS

- Complexity theory
- Space Complexity
  - Amount of memory
- Time Complexity
  - Amount of computing time

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 times as the instance characteristics change

TIME COMPLEXITY (CONT'D.)

- Compile time (c)
  - Independent of instance characteristics

- Run (execution) time T_P
  - Real measurement
  - Analysis: counts of program steps

- Definition

  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 (CONT'D.)

```
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 */
  count++;              /* for return */
  return tempsum;
}
```

2n+3 steps

TIME COMPLEXITY (CONT'D.)

```
float rsum(float list[ ], int n)
{
  count++;                  /* for conditional if */
  if (n<=0) {
    count++;                /* for return */
    return 0;
  }
  else {
    count++;                /* for return */
    return rsum(list, n-1) + list[n-1];
  }
  count++;                  /* for return */
  return list[0];
}
```

$$
\begin{aligned}
&T(n) \\
&= 2 + T(n-1) \\
&= 2 + 2 + T(n-2) \\
&= \cdots \\
&= 2n + T(0) \\
&= 2n + 2
\end{aligned}
$$

TABULAR METHOD

Table 1.1: Step count table for Program 1.13 (p.40)

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

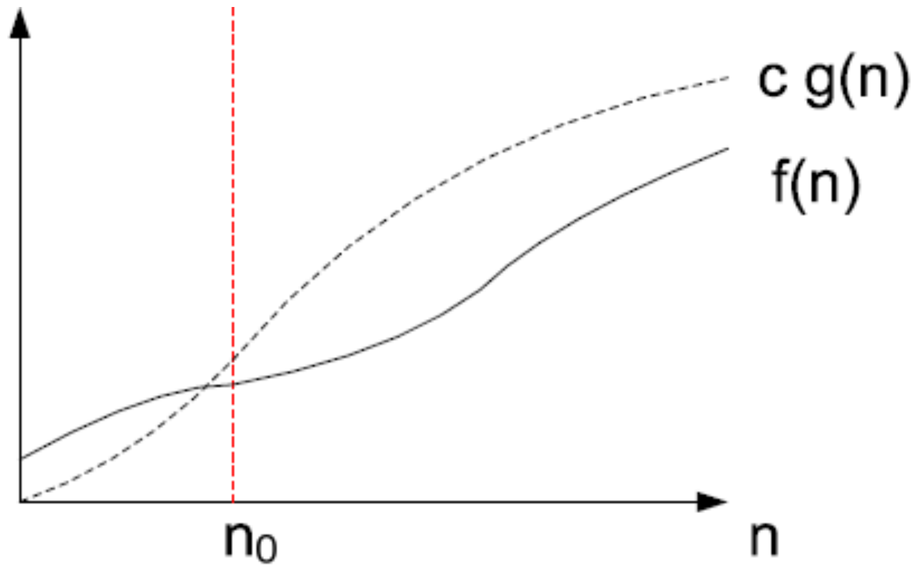
s/e: steps per execution

TIME COMPLEXITY (CONT'D.)

- Difficult to determine the exact step counts
- What a step stands for is inexact
  eg. x := y versus x := y + z + (x/y) + …
- 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 (Time Complexity)
  - Best case
  - Worst case
  - Average case

ASYMPTOTIC NOTATION – BIG "OH"

- $f(n) = O\big(g(n)\big)$ iff
  - $\exists$ a real constant $c > 0$ and an integer constant $n_0 \geq 1$, s.t. $f(n) \leq c \cdot g(n)$, $\forall n \geq n_0$



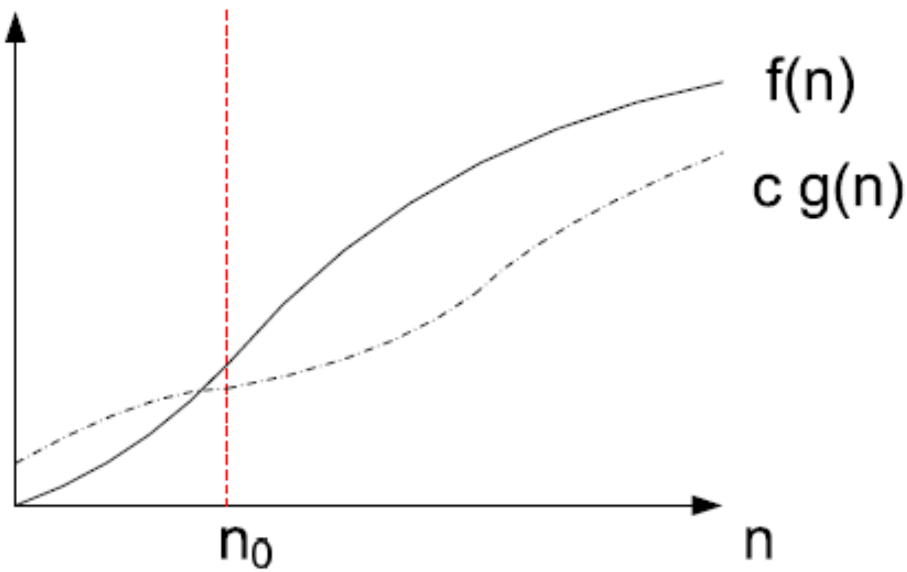
ASYMPTOTIC NOTATION – BIG "OH" (CONT'D.)

- $f(n) = O\big(g(n)\big)$ iff
  - $\exists$ a real constant $c > 0$ and an integer constant $n_0 \geq 1$, s.t. $f(n) \leq c \cdot g(n)$, $\forall n \geq n_0$
  - eg.
    - $3n + 6 = O(n)$
    - $4n^2 + 2n - 6 = O(n^2)$
    - $f(n) = a_m n^m + a_{m-1} n^{m-1} + \ldots + a_1 n + a_0$
      $f(n) = O(n^m)$
- $g(n)$ should be a least upper bound.

ASYMPTOTIC NOTATION - OMEGA

- $f(n) = \Omega\big(g(n)\big)$ iff
  - $\exists$ a real constant $c > 0$ and an integer constant $n_0 \geq 1$, s.t. $f(n) \geq c \cdot g(n)$, $\forall n \geq n_0$

- $g(n)$ should be a most lower bound.



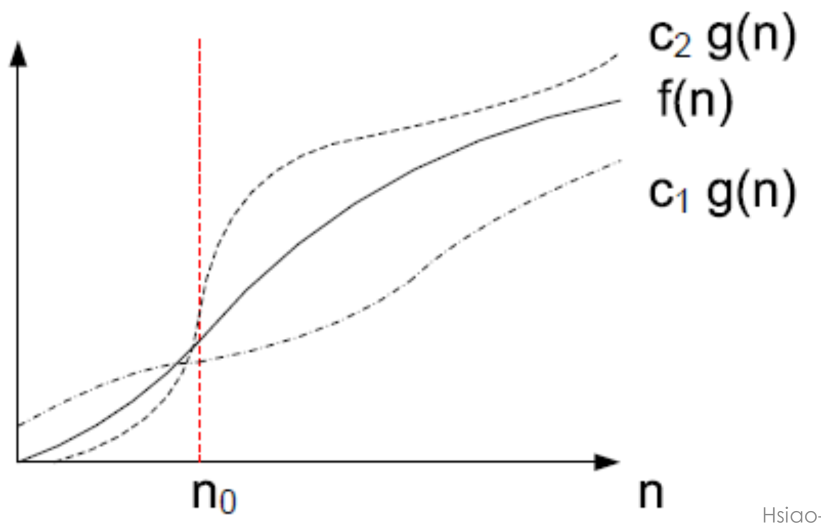
ASYMPTOTIC NOTATION – OMEGA (CONT'D)

- eg.
  - $3n+3 = \Omega(n)$
  - $3n^2+4n-8 = \Omega(n^2)$
  - $6*2^n+n^2 = \Omega(2^n)$

ASYMPTOTIC NOTATION - THETA

- $f(n) = \Theta\big(g(n)\big)$ iff
  - $\exists$ real constants c_1 and $c_2 > 0$ and an integer constant $n_0 \geq 1$, s.t. $c_1 g(n) \leq f(n) \leq c_2 g(n)$, $\forall n \geq n_0$

- $g(n)$ should be both upper bound and lower bound. It is called precise bound.



ASYMPTOTIC NOTATION – THETA (CONT'D)

- eg.
  - f(n) = $3n^2+4n-8$
  - f(n) = log(n!)

- $f(n) = \Theta(g(n)) \Leftrightarrow f(n) = O(g(n))$ and $f(n) = \Omega(g(n))$

RUNNING TIME CALCULATIONS

- For loop

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

RUNNING TIME CALCULATIONS (CONT'D)

- Nested for loops

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

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;
```

n次 --- $O(n)$

n次

n次

$\Big\}\ O(n^2)$

$O(n) + O(n^2)$

$O(n^2)$ #

$\max(n, n^2) = O(n^2)$

RUNNING TIME CALCULATIONS (CONT'D)

- If/Else

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

max(2,n) = n

RUNNING TIME CALCULATIONS (CONT'D)

- Recursive

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

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

$T(k) = T(k-1) + c$

$T(n) = T(n-1) + c$

$= T(n-2) + c + c$

$\vdots$

$= T(1) + n \times c$

$= \Theta(n)$ #

RUNNING TIME CALCULATIONS (CONT'D)

- Example 1: (Tower of Hanoi)
  - $T(n) = 2T(n-1) + 1, T(1) = 1$

  $T(k) = 2T(k-1) + 1$
- Example 2: (Binary Search)
  - $T(n) = T\left(\frac{n}{2}\right) + 1, T(1) = 1$

$f(n) = 1$, $\log_b a = \log_2 1$, $n^{\log_2 1}$
- Example 3: (sum of 0,1,…,n)
  - $T(n) = T(n-1) + n, \mathrm{T}(0) = 0$

- Other example:
  - $T(n) = 2T\left(\frac{n}{2}\right) + n, T(1) = 0$
  - $T(n) = 2T(\sqrt{n}) + 1, T(2) = 1$

$T(n) = 2T(n-1) + 1$

$= 2 \cdot [2T(n-2) + 1] + 1$

$= 2^2 \cdot T(n-2) + 2 + 1$

$= 2^2 \cdot [2T(n-3) + 1] + 2 + 1$

$= 2^3 \cdot T(n-3) + 2^2 + 2 + 1$

$\vdots$

$= 2^n \cdot T(1) + \frac{1[1-2^{n-1}]}{1-2}$

$= \Theta(2^n)$ #

SOME RULES

- Rule 1:
  If $T_1(N) = O\big(f(N)\big)\ and\ T_2(N) = O\big(g(N)\big)$ then
  (a) $T_1(N) + T_2(N) = \max(O\big(f(N)\big), O(g(N)))$
  (b) $T_1(N) \times T_2(N) = O(f(N) \times g(N))$

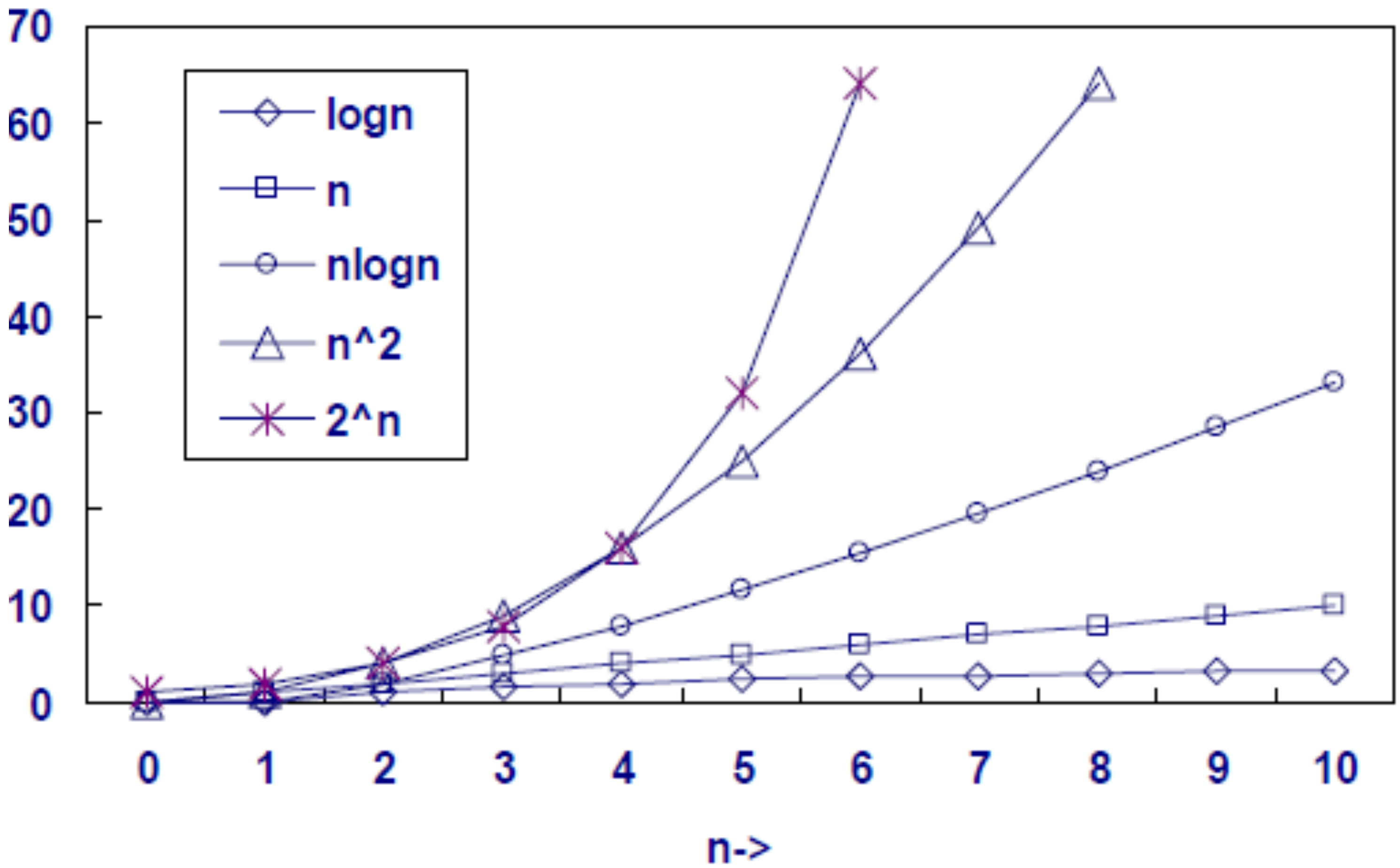
- Rule 2:
  If $T(N)$ is a polynomial of degree k, then
  $T(N) = \Theta(N^k)$

- Rule 3:
  $T(N) = (\log N)^k = \Theta(N)$ (Prove it yourself.)

TYPICAL GROWTH RATE

- c: Constant
- logN: Logarithmic
- $\log^2 N$: Log-squared
- N: Linear
- NlogN:
- N^2: Quadratic
- N^3: Cubic
- 2^N: Exponential

GROWTH RATE



EXAMPLE

BCT

- List the complexity from low to high for the following big-oh representation:

$$\boxed{\sqrt{n}, \log\log n\,, \log^3 n\,, n^2 \log n\,, \log n!\,, n^{1.5}, \left(\frac{3}{2}\right)^n, \log n^5}$$

Time Complexity

set $m=2$

According to the resules, $\begin{cases} T(n) = T(n-1) + (5+2^2 \cdot n) \\ T(1) = 14 \end{cases}$

$$
\begin{aligned}
T(n) &= T(n-1) + (5+2^2 \cdot n) \\
&= T(n-2) + (5+2^2 \cdot (n-1)) + (5+2^2 \cdot n) \\
&= T(n-3) + (5+2^2 \cdot (n-2)) + (5+2^2 \cdot (n-1)) + (5+2^2 \cdot n) \\
&\vdots \\
&= T(1) + [13 + 17 + \cdots + (5+4n)]
\end{aligned}
$$

$\because a_1 = (5+4n)$, $a_{n-1} = 13$, $d = -4$

$\therefore S_{n-1} = \dfrac{[(5+4n)+13](n-1)}{2} = 2n^2+7n-9$

$$
\begin{aligned}
&= 2n^2 + 7n + 5 \\
&= O(n^2)
\end{aligned}
$$

#

PERFORMANCE MEASUREMENT

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

EXAMPLE

- Write a C program that prints put the integer values of x,y,z in ascending order.

加上 count

```c
#include <stdio.h>
int min(int, int);
#define TRUE 1
#define FALSE 0

int main()
{
  int x,y,z;
  printf("x: "); scanf("%d", &x);
  printf("y: "); scanf ("%d", &y);
  printf("z: "); scanf("%d", &z);
  if (min(x,y) && min(x,z)) {/*x is smallest */
     printf("%d ", x);
       if (min(y,z)) printf ("%d %d\n", y,z);
       else printf("%d%d\n", z, y);
     }
```

```c
else if (min(y,x) && min(y,z)){ /*y is the smallest */
     printf("%d ", y);
       if (min(x,z)) printf ("%d %d\n", x,z);
       else printf("%d%d\n", z,x);
     }
  else
     printf("%d %d %d\n", z, y, x);
}


int min(int a, int b) {
  if (a < b) return TRUE;
  return FALSE;
}
```

Your turn.
Introduce step counts into the function.

EXERCISE

```c
void Printmatrix(int matrix[][MAX_SIZE], int rows, int cols)
{
  int i,j;
  for (i = 0; i<rows; i++)
  {
    for (j = 0; j<cols; j++)
      printf("%5d",matrix[i][j]);

    printf("\n");
  }
}
```

1. Introduce step counts into the function.
2. Step count table.
3. Given a square matrix $n \times n$, determine its time complexity.
4. (Additional) Write a recurrence version.

<table>
<tr><th>Statement</th><th>s/e</th><th>f</th><th>Total steps</th></tr>
<tr><td><pre>void Printmatrix(int matrix[][MAX_SIZE], int rows, int cols)
{
  int i,j;
  for (i = 0; i<rows; i++)
  {
    for (j = 0; j<cols; j++)
      printf("%5d",matrix[i][j]);

    printf("\n");
  }
}</pre></td><td></td><td></td><td></td></tr>
<tr><td colspan="3">Total</td><td></td></tr>
<tr><td colspan="3">Complexity (asymptotic notation)</td><td></td></tr>
</table>

PROGRAM EXERCISE: FIBONACCI NUMBERS

- Fibonacci numbers are defined as

$$f_0 = 0, f_1 = 1, and\ f_i = f_{i-1} + f_{i-2}\ for\ i > 1.$$

- Write both recursive and iterative C function to compute f_i.

```c
#include <stdio.h>
int iterFib(int);
int recurFib(int);

int main(){
    int n;
    printf("n:(>=0): ");
    scanf("%d", &n);
    while (n < 0)
    { /*error loop */
      printf("n:(>=0): ");
      scanf("%d", &n);
    }
  printf("%d Fibonacci is %d.\n", n, iterFib(n));
  printf("%d Fibonacci is %d.\n", n, recurFib(n));
}
```

```c
int recurFib(int n)
{ /*recursive version */                  n
  if ((n==0) || (n==1))  return 1;
  return [ (1) ]
}

int iterFib(int n)
{/* find the factorial, return as a double
    to keep it from overflowing */
   int i; f(i) fib(i-1)   fib(i-2)
   int fib, fib1, fib2;                       n
   if ((n == 0) || (n == 1)) return 1;
   [ (2) ]  → 初始化的東西
   for (i = 2; i <=n; i++) {
      [ (3) ]
      [ (4) ]
      [ (5) ]
   }
   return fib;
}
```

↳ 數列中的第i個數 f(n)

- Exception handling: Add some lines that can handle exceptions.

PROGRAM EXERCISE: FIBONACCI NUMBERS

- Test your code using the following cases:

Fib(n)	Output	
N	iterFib	recurFib
0	Fibonacci is 0.	Fibonacci is 0.
1	Fibonacci is 1.	Fibonacci is 1.
2	…	..
9	…	..
13		
-1	…	..
-10	…	..

- Find the running time equation for Fibonacci.