DATA STRUCTURE

Lecture 02: Recursion, Algorithm Analysis.

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 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

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

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

RECURSIVE MULTIPLICATION

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

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

BINARY SEARCH

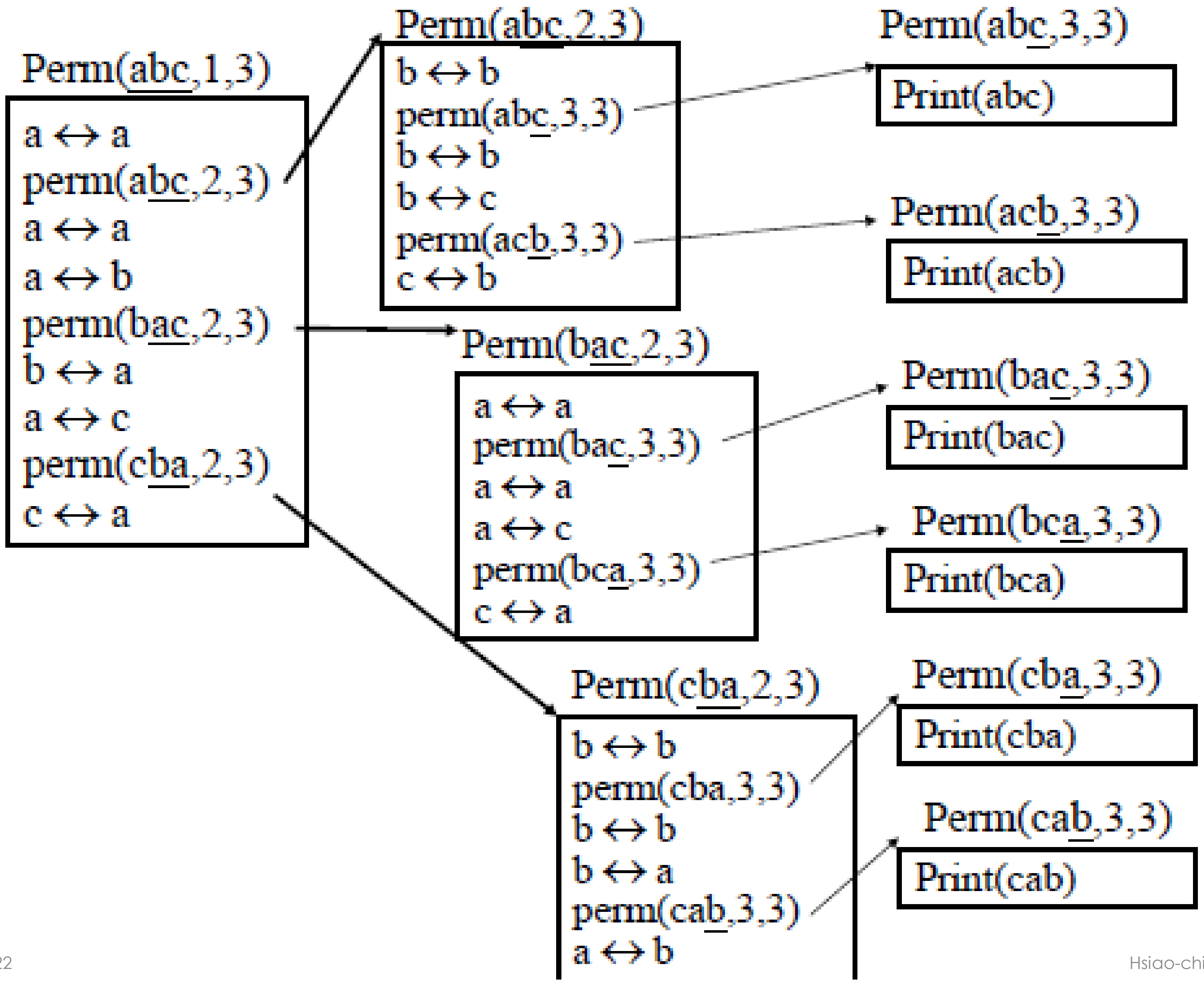
```
int compare(int x, int y)
/* return -1 for less than, 0 for equal */
int binsearch(int list[], int searchno, int left,
    int right)
{
  while (left <= right)  {
    middle = (left + right) / 2;
    switch ( COMPARE(list[middle], searchno) )  {
      case -1:
        left = middle +1;
        break;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

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 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

SPACE COMPLEXITY

- $S(P) = c + S_p(I)$
  - c: fixed space (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, $S_p(I) \Rightarrow S_p(n)$

SPACE COMPLEXITY (CONT'D.)

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

$$\boxed{S_{abc}(I)=0}$$

SPACE COMPLEXITY (CONT'D.)

```
float rsum(float list[ ], int n)
{
  if (n)
    return rsum(list, n-1) + list[n-1];
  return 0;
 }
```

$\boxed{S_{sum}(I)=S_{sum}(n)=6n}$

Assumptions:

Space needed for one recursive call of the program

Type	Name	Number of bytes
Parameter: float	list[ ]	2
Parameter: integer	n	2
Return address: (used internally)		2 (unless a far address)
Total		6

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

$\boxed{2n+3 \text{ steps}}$

TIME COMPLEXITY (CONT'D.)

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

$T(n)$
$=2+T(n-1)$
$=2+2+T(n-2)$
...
$=2n+T(0)$
$=2n+2$

TABULAR METHOD

<u>*Table 1.1: Step count table for Program 1.13 (p.40)*</u>

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

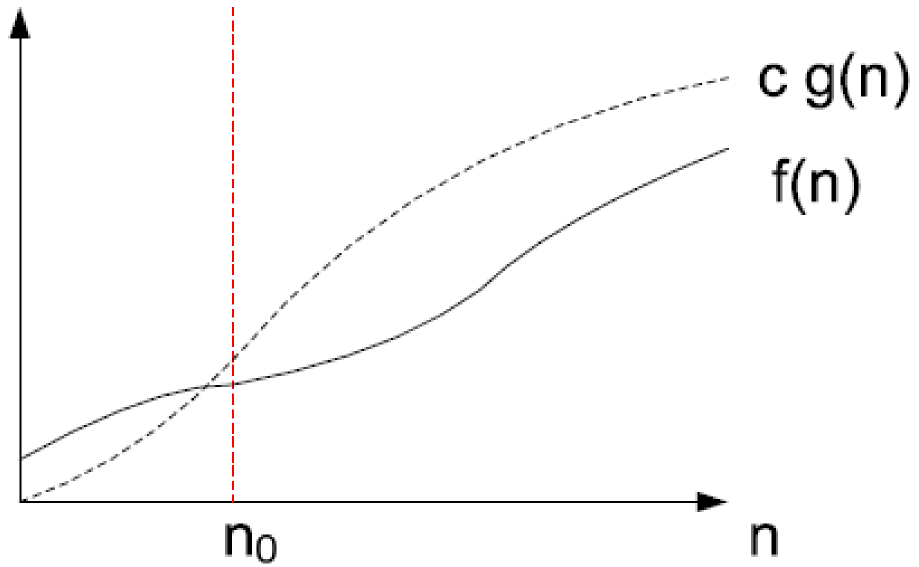
s/e: steps per execution

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. $f(n) \leq c \cdot g(n)$, $\forall n \geq n_0$



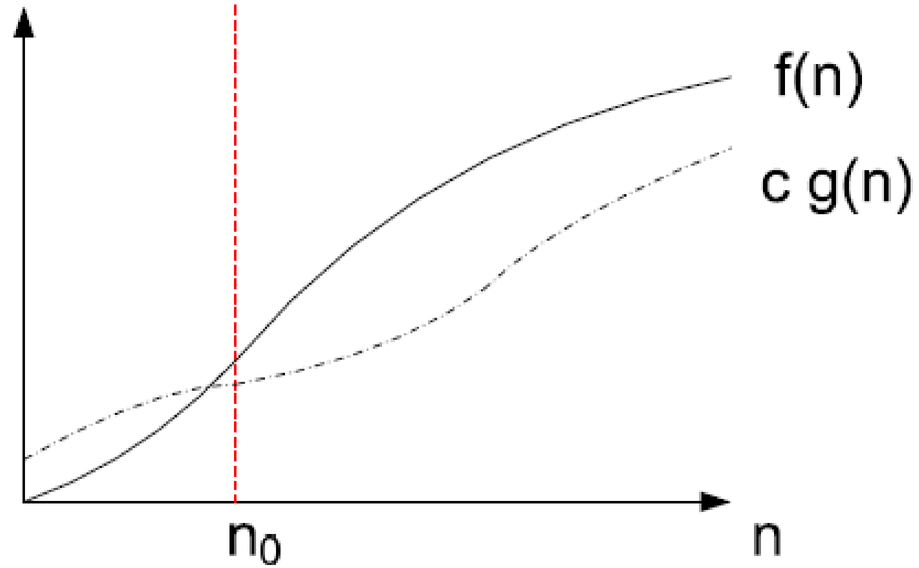
ASYMPTOTIC NOTATION – BIG "OH" (CONT'D.)

- $f(n) = O\big(g(n)\big)$ 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

- $f(n) = \Omega(g(n))$ 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.



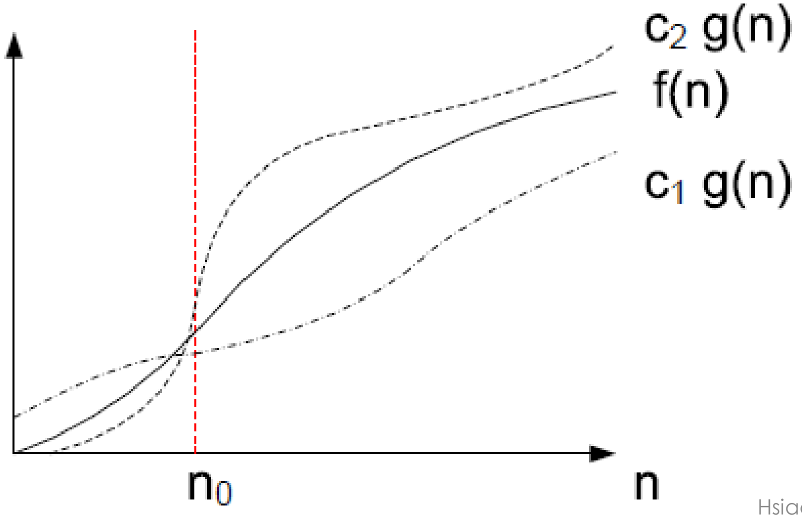
ASYMPTOTIC NOTATION – OMEGA (CONT'D)

- 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;
```

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

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);
}
```

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

RUNNING TIME CALCULATIONS (CONT'D)

- Example 1: (Tower of Hanoi)
  - $T(n) = 2T(n-1) + 1, T(1) = 1$
- Example 2: (Binary Search)
  - $T(n) = T\left(\frac{n}{2}\right) + 1, T(1) = 1$
- Example 3: (sum of 0,1,…,n)
  - $T(n) = T(n-1) + n, \mathrm{T}(0) = 0$
- Other example:
  - $T(n) = 2T\left(\frac{n}{2}\right) + n, T(1) = 0$
  - $T(n) = 2T(\sqrt{n}) + 1, T(2) = 1$

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
  $T(N) = \Theta(N^k)$

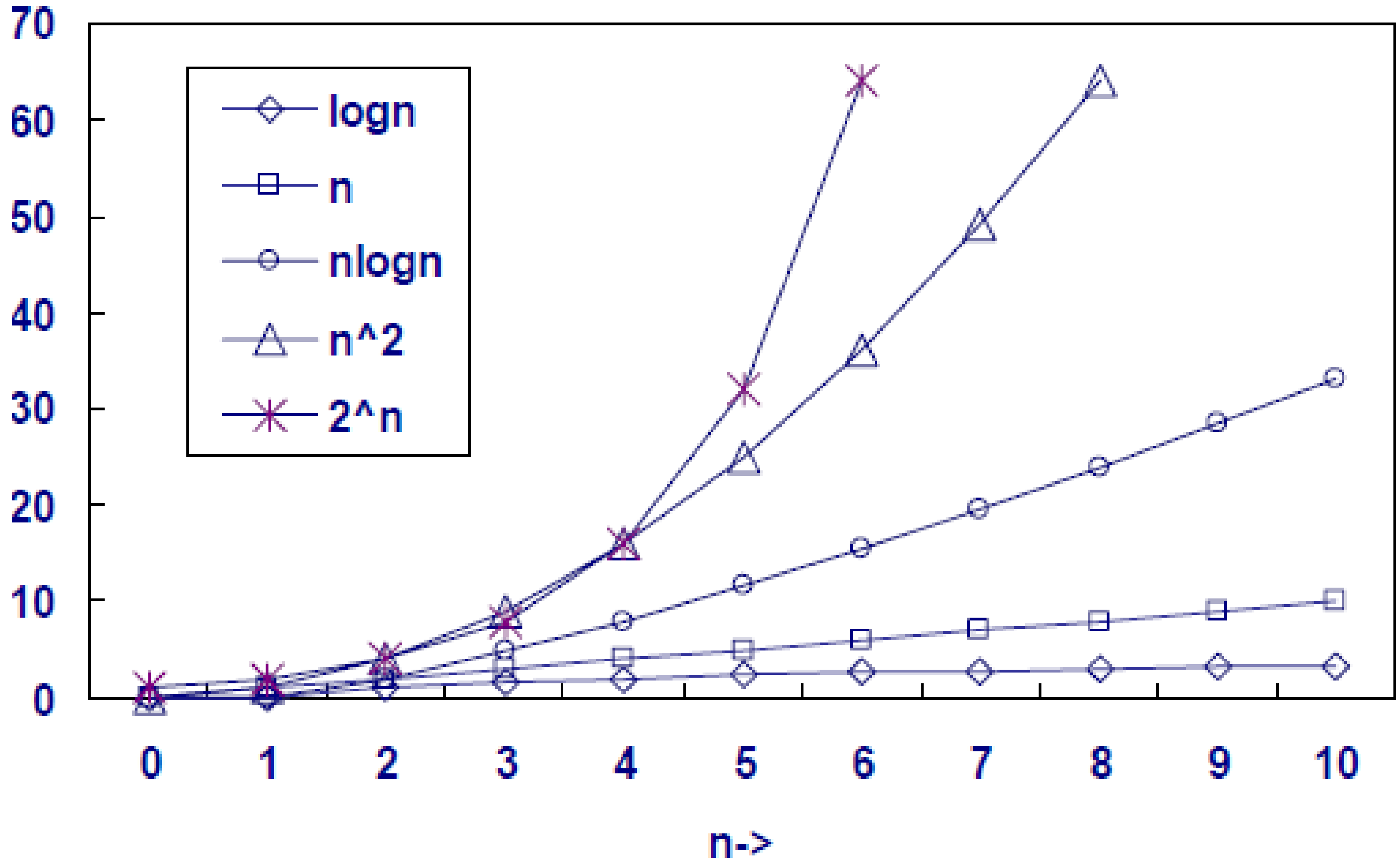
- Rule 3:

  $T(N) = (\log N)^k = \Theta(N)$ (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

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

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

PERFORMANCE MEASUREMENT

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