Lecture 03 Supplementary

Ting ZHANG
2023/09

Highlights of Lecture

- Complexity analysis of algorithms
- Mathematics preliminary for algorithm analysis

Fundamentals of the Analysis of Algorithm Efficiency

- Algorithm analysis framework
- Asymptotic notations渐进符号
- Analysis of non-recursive algorithms
- Analysis of recursive algorithms

Expected Outcomes

The students should be able to

- List the **key steps** in an algorithm's analysis framework
- Define the **three asymptotic notations** and explain their relations
- Compare the **asymptotic growth rate** of two given functions by using the definition of asymptotic notation or Limits
- Describe the key steps in analysis of a **non-recursive** algorithm and a **recursive** algorithm
- Use different ways to **transform a recurrence relation** into its closed form
- Analyze the time complexity of recursive algorithm for computing the nth **Fibonacci** number

Analysis of Algorithms

DEFINITION: Analysis of algorithms means to investigate an algorithm's efficiency with respect to resources: running time and memory space.

Time efficiency: how fast an algorithm runs.
Space efficiency: the space an algorithm requires.

Typically, algorithms run longer as the size of its input increases.
We are interested in how efficiency scales wrt input size

Analysis Framework

- Measuring an input's size
- Measuring running time
- Orders of growth (of the algorithm's efficiency function)
- Worst-case, best-case and average-case efficiency

1. Measuring Input Sizes

- Efficiency is defined as a function of input size.
- Input size depends on the problem.

  Example 1, what is the input size of the problem of sorting n numbers?
  Example 2, what is the input size of adding two n by n matrices?

2. Units for Measuring Running Time

- Should we measure the running time using standard unit of time measurements, such as seconds, minutes?

  Depends on the speed of the computer.

- Count the number of times each of an algorithm's operations is executed.

  Difficult and unnecessary

- Count the number of times an algorithm's basic operation is executed.

  Basic operation: the operation that contributes the most to the total running time. For example, the basic operation is usually the most time-consuming operation in the algorithm's innermost loop.

Input size and basic operation examples

Problem	*Input size measure*	***Basic operation***
Searching for key in a list of n items	Number of list's items, i.e. n	**Key comparison**
Multiplication of two matrices	Matrix dimensions or total number of elements	**Multiplication of two numbers**
Typical graph problem	#vertices and/or edges	**Visiting a vertex or traversing an edge**

Theoretical Analysis of Time Efficiency

Time efficiency is analyzed by determining the number of repetitions of the basic operation as a function of input size.

Assuming $C(n) = (1/2)n(n-1)$,

how much longer will the algorithm run if we double the input size?

input size

$$T(n) \approx c_{op} C(n)$$

running time

execution time
for the basic operation

Number of times
the basic operation
is executed

The efficiency analysis framework ignores the multiplicative constants of C(n) and focuses on the order of growth of the C(n).

3. Order of growth增长级数

Most important: Order of growth within a constant multiple as $n \to \infty$

Example:

How much faster will algorithm run on computer that is twice as fast?

How much longer does it take to solve problem of double input size?

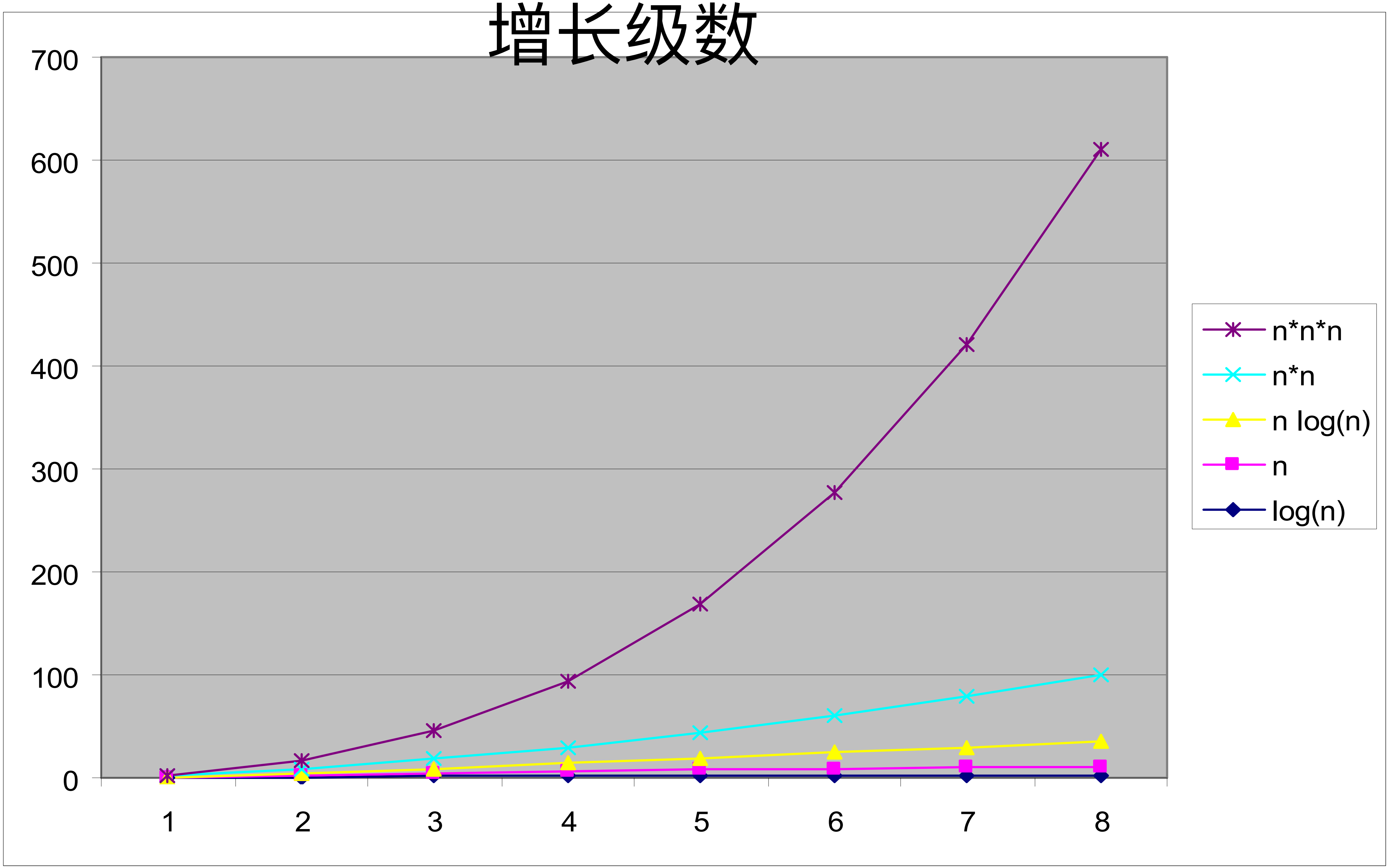
Orders of Growth

Exponential-growth functions

n	$\log_2 n$	n	$n \log_2 n$	n^2	n^3	2^n	$n!$
10	3.3	10^1	$3.3 \cdot 10^1$	10^2	10^3	10^3	$3.6 \cdot 10^6$
10^2	6.6	10^2	$6.6 \cdot 10^2$	10^4	10^6	$1.3 \cdot 10^{30}$	$9.3 \cdot 10^{157}$
10^3	10	10^3	$1.0 \cdot 10^4$	10^6	10^9		
10^4	13	10^4	$1.3 \cdot 10^5$	10^8	10^{12}		
10^5	17	10^5	$1.7 \cdot 10^6$	10^{10}	10^{15}		
10^6	20	10^6	$2.0 \cdot 10^7$	10^{12}	10^{18}		

Orders of growth:

- consider only the leading term of a formula
- ignore the constant coefficient.


增长级数
700
600
500
400
300
200
100
0
1
2
3
4
5
6
7
8
n*n*n
n*n
n log(n)
n
log(n)

Worst-Case, Best-Case, and Average-Case Efficiency

Algorithm efficiency depends on

- the input size n
- also the contents of input.

Example: Sequential Search

Problem: Given a list of n elements and a search key K, find an element equal to K, if any.

Algorithm: Scan the list and compare its successive elements with K until either a matching element is found (*successful search*) or the list is exhausted (*unsuccessful search*)

Sequential Search Algorithm

```
ALGORITHM  SequentialSearch(A[0..n-1], K)

    FOR i from 0 to ( n - 1 )
        IF A[i] = K
            RETURN  i
        END IF
    END FOR
    RETURN  -1
```

- Worst case Efficiency

  Efficiency (# of times the basic operation will be executed) for the worst case input of size n. The algorithm runs the longest among all possible inputs of size n.

- Best case

  Efficiency (# of times the basic operation will be executed) for the best case input of size n. The algorithm runs the fastest among all possible inputs of size n.

- Average case:

  Efficiency (# of times the basic operation will be executed) for a typical/random input of size n. NOT the average of worst and best case. How to find the average case efficiency?

Summary of the Analysis Framework

- Time and space efficiencies : functions of input size

  √ the number of basic operations
  √ the number of extra memory units

- The order of growth of the algorithm’s running time (space) as its input size goes infinity.

- The efficiencies of some algorithms may differ significantly for inputs of the same size.

  √ worst-case
  √ best-case
  √ average case

Asymptotic Growth Rate

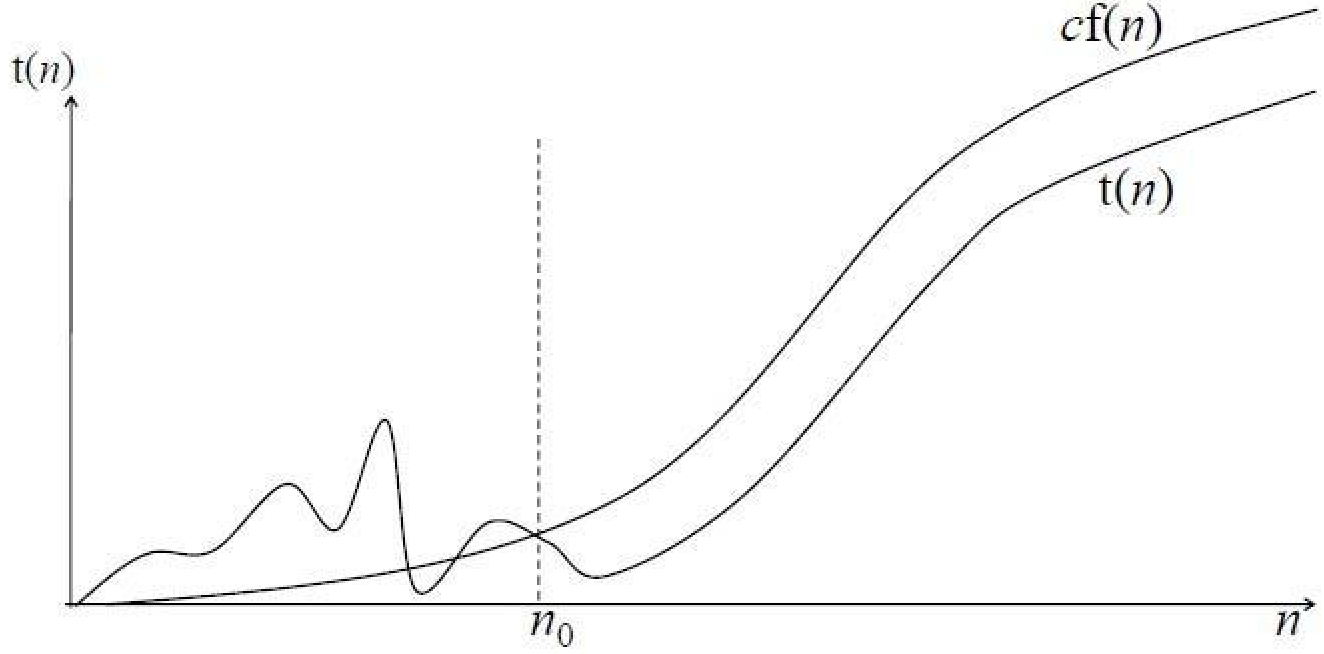
Three notations used to compare orders of growth of an algorithm's basic operation count

$O(f(n))$: class of functions that grow *no faster* than $f(n)$

$\Omega(f(n))$: class of functions that grow *at least as fast* as $f(n)$

$\Theta(f(n))$: class of functions that grow *at same rate* as $f(n)$

$t(n) \in O(f(n))$



There exist n_0 and $c > 0$ such that for all $n > n_0$
$t(n) \leq cf(n)$

O-notation

Formal definition

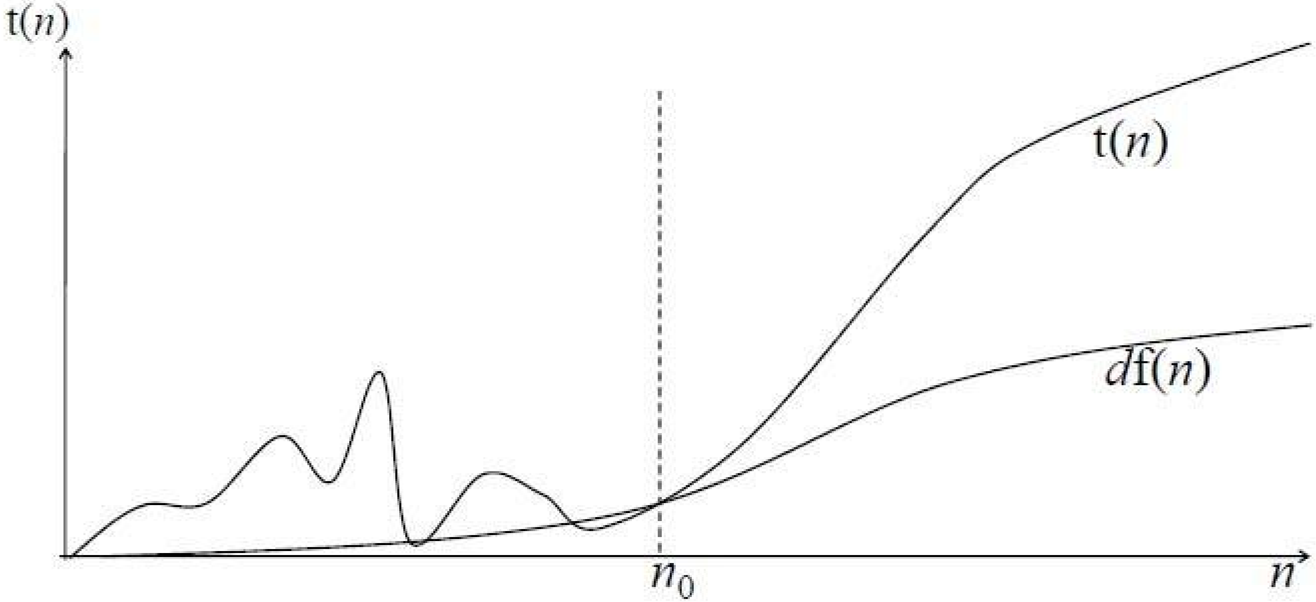
A function $t(n)$ is said to be in $O(f(n))$, denoted $t(n) \in O(f(n))$, if $t(n)$ is bounded above by some constant multiple of $f(n)$ for all large n, i.e., if there exist some positive constant c and some nonnegative integer n_0 such that

$$t(n) \leq cf(n) \text{ for all } n \geq n_0$$

Class Exercises: prove the following using the above definition

$10n^2 \in O(n^2)$
$10n^2 + 2n \in O(n^2)$
$100n + 5 \in O(n^2)$
$5n+20 \in O(n)$

$t(n) \in \Omega(f(n))$



There exist n_0 and $d > 0$ such that for all $n > n_0$
$t(n) \geq d f(n)$

Ω-notation

Formal definition

A function $t(n)$ is said to be in $\Omega(f(n))$, denoted $t(n) \in \Omega(f(n))$, if $t(n)$ is bounded below by some constant multiple of $f(n)$ for all large n, i.e., if there exist some positive constant d and some nonnegative integer n_0 such that

$$t(n) \geq df(n) \text{ for all } n \geq n_0$$

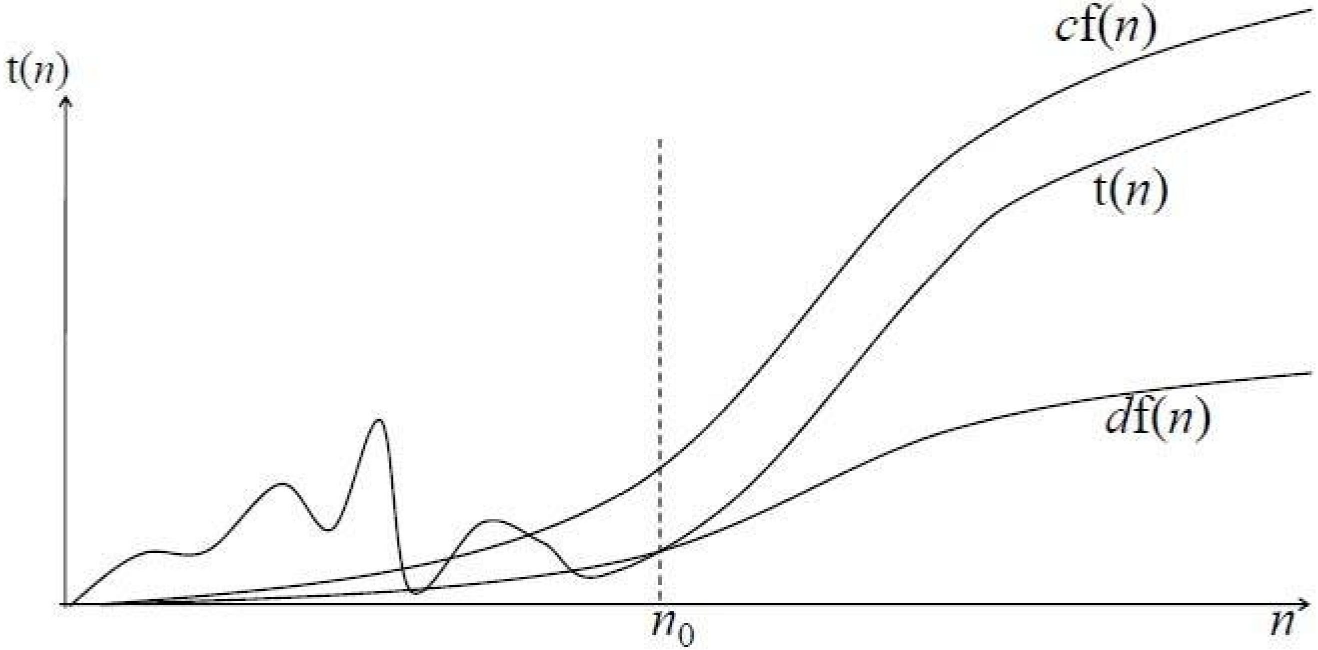
Exercises: prove the following using the above definition

$10n^2 \in \Omega(n^2)$

$10n^2 + 2n \in \Omega(n^2)$

$10n^3 \in \Omega(n^2)$

$t(n) \in \Theta(f(n))$



There exist n_0 and $c,d > 0$ such that for all $n > n_0$
$df(n) \leq t(n) \leq cf(n)$

Θ-notation

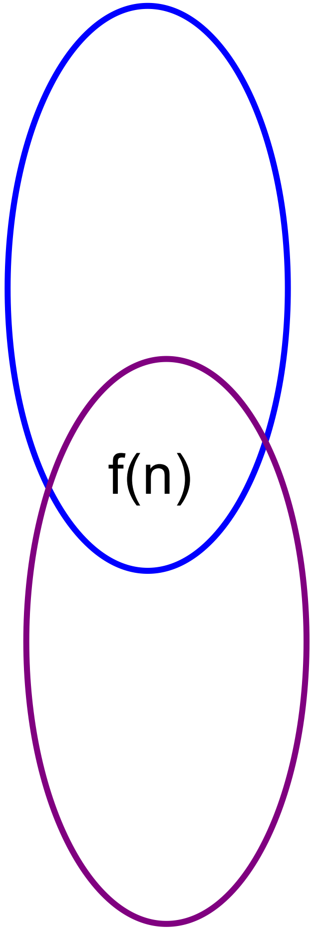
Formal definition

A function $t(n)$ is said to be in $\Theta(f(n))$, denoted $t(n) \in \Theta(f(n))$, if $t(n)$ is bounded both above and below by some positive constant multiples of $f(n)$ for all large n, i.e., if there exist some positive constant c and d and some nonnegative integer n_0 such that

$$d\ f(n) \leq t(n) \leq c\ f(n) \text{ for all } n \geq n_0$$

Exercises: prove the following using the above definition

$10n^2 \in \Theta(n^2)$
$10n^2 + 2n \in \Theta(n^2)$
$(1/2)n(n-1) \in \Theta(n^2)$



>=

$\Omega(f(n))$, functions that grow <u>at least as fast as</u> f(n)

=

$\Theta(f(n))$, functions that grow <u>at the same rate as</u> f(n)

<=

$O(f(n))$, functions that grow <u>no faster than</u> f(n)

Some Properties of Asymptotic Order of Growth

1. $f(n) \in O(f(n))$

2. $f(n) \in O(g(n))$ iff $g(n) \in \Omega(f(\mathrm{n}))$

3. If $f(n) \in O(g(n))$ and $g(n) \in O(h(n))$, then $f(n) \in O(h(n))$

   Note similarity with $a \leq \mathrm{b}$

4. If $f_1(n) \in O(g_1(n))$ and $f_2(n) \in O(g_2(n))$, then
   $$f_1(n) + f_2(n) \in O(\max\{g_1(n), g_2(n)\})$$

The analogous assertions are true for the Ω-notation and Θ-notation.

Some Properties of Asymptotic Order of Growth

4. If $f_1(n) \in O(g_1(n))$ and $f_2(n) \in O(g_2(n))$, then

$$f_1(n) + f_2(n) \in O(\max\{g_1(n), g_2(n)\})$$

Implication: The algorithm's overall efficiency will be Determined by the part with a larger order of growth.

Example: $5n^2 + 3n\log n \in O(n^2)$

Using Limits for Comparing Orders of Growth

$$\lim_{n\to\infty} T(n)/g(n) = \begin{cases} 0 & \text{order of growth of } \boldsymbol{T(n)} < \text{order of growth of } \boldsymbol{g(n)} \\ c>0 & \text{order of growth of } \boldsymbol{T(n)} = \text{order of growth of } \boldsymbol{g(n)} \\ \infty & \text{order of growth of } \boldsymbol{T(n)} > \text{order of growth of } \boldsymbol{g(n)} \end{cases}$$

Examples:

- $10n$ vs. $2n^2$
- $n(n+1)/2$ vs. n^2
- $\log_b n$ vs. $\log_c n$

L'Hôpital's rule

If $lim_{n\to\infty} f(n) = lim_{n\to\infty} g(n) = \infty$ and the derivatives f', g' exist, Then

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = \lim_{n\to\infty} \frac{f'(n)}{g'(n)}$$

Example:

- **$\log_2 n$ vs. n**
- **2^n vs. $n!$**

Stirling's formula: $n! \approx (2\pi n)^{1/2} (n/e)^n$

Orders of growth of some important functions

- All logarithmic functions $\log_a n$ belong to the same class

  $\Theta(\log n)$, no matter what the logarithm's base $a > 1$ is.

- All polynomials of the same degree k belong to the same class:

  $a_k n^k + a_{k-1} n^{k-1} + \ldots + a_0 \in \Theta(n^k)$.

- Exponential functions a^n have different orders of growth for different a's.

  order $\log n$ < order n^α (α>0) < order a^n < order $n!$ < order n^n

Basic Efficiency classes

The time efficiencies of a large number of algorithms fall into only a few classes.

fast — High time efficiency

1	constant
log n	logarithmic
n	linear
$n \log n$	$n \log n$
n^2	quadratic
n^3	cubic
2^n	exponential
$n!$	factorial

slow — low time efficiency

Summary of How to Establish Orders of Growth of an Algorithm's Basic Operation Count

Method 1: Using limits. L'Hôpital's rule

Method 2: Using the properties

Method 3: Using the definitions of O-, Ω-, and Θ-notation.

Time Efficiency of Nonrecursive Algorithms

- Decide on *input size*
- Identify *basic operation*
- Investigate *worst*, *average*, and *best* case efficiency separately if required.
- Set up summation for *C(n)* reflecting the number of times basic operation is executed.
- Simplify summation using standard formulas

Example 1: Maximum element

ALGORITHM $MaxElement(A[0..n-1])$

//Determines the value of the largest element in a given array

//Input: An array $A[0..n-1]$ of real numbers

//Output: The value of the largest element in A

$maxval \leftarrow A[0]$

for $i \leftarrow 1$ **to** $n-1$ **do**

  if $A[i] > maxval$

    $maxval \leftarrow A[i]$

return $maxval$

Example 2: Element uniqueness problem

ALGORITHM $UniqueElements(A[0..n-1])$
//Determines whether all the elements in a given array are distinct
//Input: An array $A[0..n-1]$
//Output: Returns "true" if all the elements in A are distinct
// and "false" otherwise
for $i \leftarrow 0$ **to** $n-2$ **do**
for $j \leftarrow i+1$ **to** $n-1$ **do**
if $A[i] = A[j]$ **return false**
return true

Example 3: Matrix multiplication

ALGORITHM $MatrixMultiplication(A[0..n-1, 0..n-1], B[0..n-1, 0..n-1])$
//Multiplies two n-by-n matrices by the definition-based algorithm
//Input: Two n-by-n matrices A and B
//Output: Matrix $C = AB$
for $i \leftarrow 0$ **to** $n-1$ **do**
    for $j \leftarrow 0$ **to** $n-1$ **do**
        $C[i, j] \leftarrow 0.0$
        for $k \leftarrow 0$ **to** $n-1$ **do**
            $C[i, j] \leftarrow C[i, j] + A[i, k] * B[k, j]$
return C

Example 4: Counting binary digits

ALGORITHM *Binary*(n)

//Input: A positive decimal integer n

//Output: The number of binary digits in n's binary representation

$count \leftarrow 1$

while $n > 1$ **do**

$count \leftarrow count + 1$

$n \leftarrow \lfloor n/2 \rfloor$

return $count$

Mathematical Analysis of Recursive Algorithms

- ✓ Recursive evaluation of n!
- ✓ Recursive solution to the Towers of Hanoi puzzle
- ✓ Recursive solution to the number of binary digits problem

Example 1: Recursive evaluation of *n* !

Iterative Definition

F(n) = 1 if n = 0
F(n) = n * (n-1) * (n-2)... 3 * 2 * 1 if n > 0

Recursive definition

F(n) = 1 if n = 0
F(n) = n * F(n-1) if n > 0

```
Algorithm  F(n)
    if n = 0
        return 1                     //base case
    else
        return F (n -1) * n          //general case
```

Example 1: Recursive evaluation of n !

Two Recurrences

The one for the factorial function value: F(n)

$$F(n) = F(n-1) * n \text{ for every } n > 0$$

$$F(0) = 1$$

The one for number of multiplications to compute $n!$, M(n)

$$C(n) = C(n-1) + 1 \text{ for every } n > 0$$

$$C(0) = 0$$

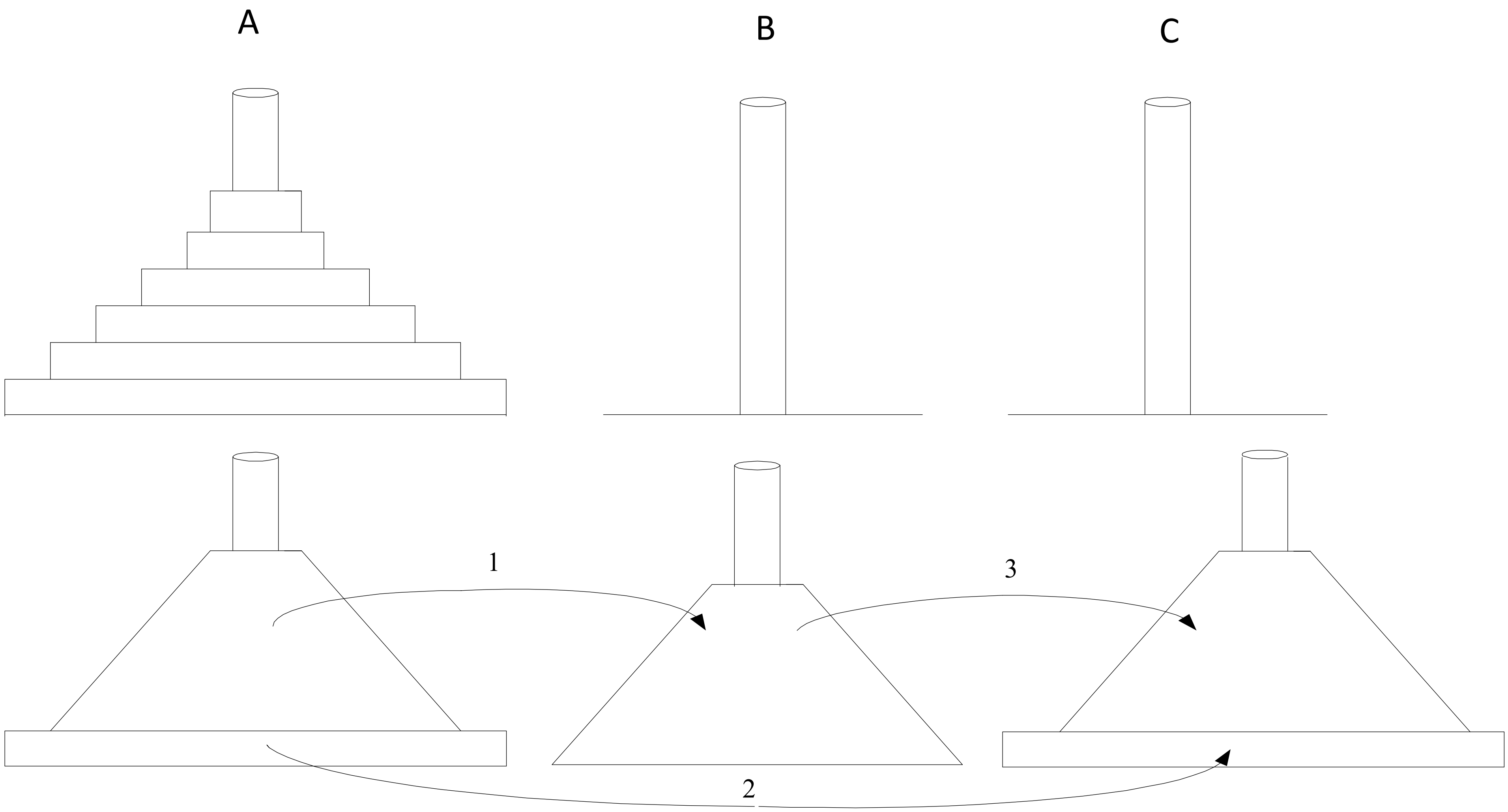
$$C(n) \in \Theta(n)$$

核心是确认“Cn与Cn-1的关系”

Steps in Mathematical Analysis of Recursive Algorithms

- Decide on *input size*
- Identify *basic operation*
- Investigate worst, average, and best case efficiency separately if required.
- Set up a recurrence relation and initial condition(s) for $C(n)$-the number of times the basic operation will be executed for an input of size n (alternatively count recursive calls).
- Solve the recurrence or estimate the order of magnitude of the solution by backward substitution or another method

Example 2: The Tower of Hanoi Puzzle



The Towers of Hanoi Puzzle

Recurrence Relations

$C(n) = 2C(n - 1) + 1$ for every $n > 0$

$C(1) = 1$

Succinctness vs. efficiency

$C(n) \in \Theta(2^n)$

Be careful with recursive algorithms because their succinctness mask their inefficiency.

Example 3: Find the number of binary digits in the binary representation of a positive decimal integer

ALGORITHM $BinRec(n)$

//Input: A positive decimal integer n
//Output: The number of binary digits in n's binary representation
if $n = 1$ **return** 1
else return $BinRec(\lfloor n/2 \rfloor) + 1$

Compute number of additions:
$C(n) = C(\lfloor n/2 \rfloor)+1$ with $C(1) = 0$

$$C(n) = C(\lfloor n/2 \rfloor)+1 \text{ with } C(1) = 0 \rightarrow C(n) \in \Theta(\log n)$$

Fibonacci numbers

The Fibonacci numbers:

0, 1, 1, 2, 3, 5, 8, 13, 21, …

The Fibonacci recurrence:

$$F(n) = F(n\text{-}1) + F(n\text{-}2)$$
$$F(0) = 0$$
$$F(1) = 1$$

General 2nd order linear homogeneous recurrence with constant coefficients:

$$aX(n) + bX(n\text{-}1) + cX(n\text{-}2) = 0$$

Solving $a\mathrm{X}(n) + b\mathrm{X}(n\text{-}1) + c\mathrm{X}(n\text{-}2) = 0$

Set up the characteristic equation (quadratic)

$$ar^2 + br + c = 0$$

Solve to obtain roots r_1 and r_2

General solution to the recurrence

if r_1 and r_2 are two distinct real roots: $\mathrm{X}(n) = \alpha\, r_1^{\,n} + \beta\, r_2^{\,n}$
if $r_1 = r_2 = r$ are two equal real roots: $\mathrm{X}(n) = \alpha\, r^n + \beta\, nr^{\,n}$

Particular solution can be found by using initial conditions

Application to the Fibonacci numbers

$F(n) = F(n\text{-}1) + F(n\text{-}2)$ or $F(n) - F(n\text{-}1) - F(n\text{-}2) = 0$

Characteristic equation:

Roots of the characteristic equation:

General solution to the recurrence:

Particular solution for $F(0) = 0$, $F(1) = 1$:

Golden Ratio

$$F(n) = \frac{1}{\sqrt{5}}(\frac{1+\sqrt{5}}{2})^n - \frac{1}{\sqrt{5}}(\frac{1-\sqrt{5}}{2})^n = \frac{1}{\sqrt{5}}(\phi^n - \hat{\phi}^n)$$

$$F(n) = \frac{1}{\sqrt{5}}\phi^n \text{ rounded to the nearest integer}$$

$$\phi = \frac{1+\sqrt{5}}{2} \approx 1.61803 \text{ is known as golden ratio}$$

Efficiency for Recursive Computation of Fibonacci Number

Number of additions:

$$C(n) = C(n-1) + C(n-2) + 1 \text{ for } n > 1$$

$$C(0) = 0, C(1) = 0$$

$$C(n) = F(n+1) - 1 \in \Theta(\phi^n)$$

Computing Fibonacci numbers

1. Definition-based recursive algorithm $\Theta(\phi^n)$
2. Nonrecursive definition-based algorithm $\Theta(n)$
3. Explicit formula algorithm $\Theta(1)$
4. Logarithmic algorithm based on formula: $\Theta(\log n)$

$$\begin{pmatrix} F(n-1) & F(n) \\ F(n) & F(n+1) \end{pmatrix} = \begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix}^n$$

for $n \geq 1$, assuming an efficient way of computing matrix powers.

Important Recurrence Types

Recurrence Relation
的两种典型形式

Decrease-by-one recurrences

A decrease-by-one algorithm solves a problem by exploiting a relationship between a given instance of size *n* and a smaller size *n* – 1.
Example: n!
The recurrence equation for investigating the time efficiency of such algorithms typically has the form

$C(n) = C(n\text{-}1) + f(n)$

后项展开

Decrease-by-a-constant-factor recurrences

A decrease-by-a-constant algorithm solves a problem by dividing its given instance of size *n* into several smaller instances of size *n/b*, solving each of them recursively, and then, if necessary, combining the solutions to the smaller instances into a solution to the given instance.
Example: binary search.
The recurrence equation for investigating the time efficiency of such algorithms typically has the form

$C(n) = aC(n/b) + f(n)$

后面会给查找表

Decrease-by-one Recurrences

One (constant) operation reduces problem size by one.

$C(n) = C(n-1) + c \qquad C(1) = d$

Solution:

$C(n) = (n-1)c + d$ *linear*

A pass through input reduces problem size by one.

$C(n) = C(n-1) + cn \qquad C(1) = d$

Solution:

$C(n) = [n(n+1)/2 - 1]\, c + d$ *quadratic*

Decrease-by-a-constant-factor recurrences – The Master Theorem

$C(n) = aC(n/b) + f(n)$, where $f(n) \in \Theta(n^k)$, $k>=0$

1. $a < b^k$ $C(n) \in \Theta(n^k)$
2. $a = b^k$ $C(n) \in \Theta(n^k \log n)$
3. $a > b^k$ $C(n) \in \Theta(n^{\log_b a})$

大概是这么考?
关键是找Cn
Recurrence Relation
要按照这个思路找关系,

Examples:

$C(n) = C(n/2) + 1$ $\Theta(\log n)$

$C(n) = 2C(n/2) + n$ $\Theta(n\log n)$

$C(n) = 3C(n/2) + n$ $\Theta(n^{\log_2 3})$

Consider the function $t(n) = 12n^2 + 17/3\ n + 7/5$

Question: what is the "big O" proper expression?

–

In-Class-work:

1. Design an algorithm for matrix multiplication, and then analyze its complexity.

2. Design an algorithm for finding the most frequent element of a vector, then analyze its complexity.

Exercise:

- Is $\log(3n + 4/n) \in O(n)$? Why?

- What is the algorithm analysis framework?