中山大學 SUN YAT-SEN UNIVERSITY

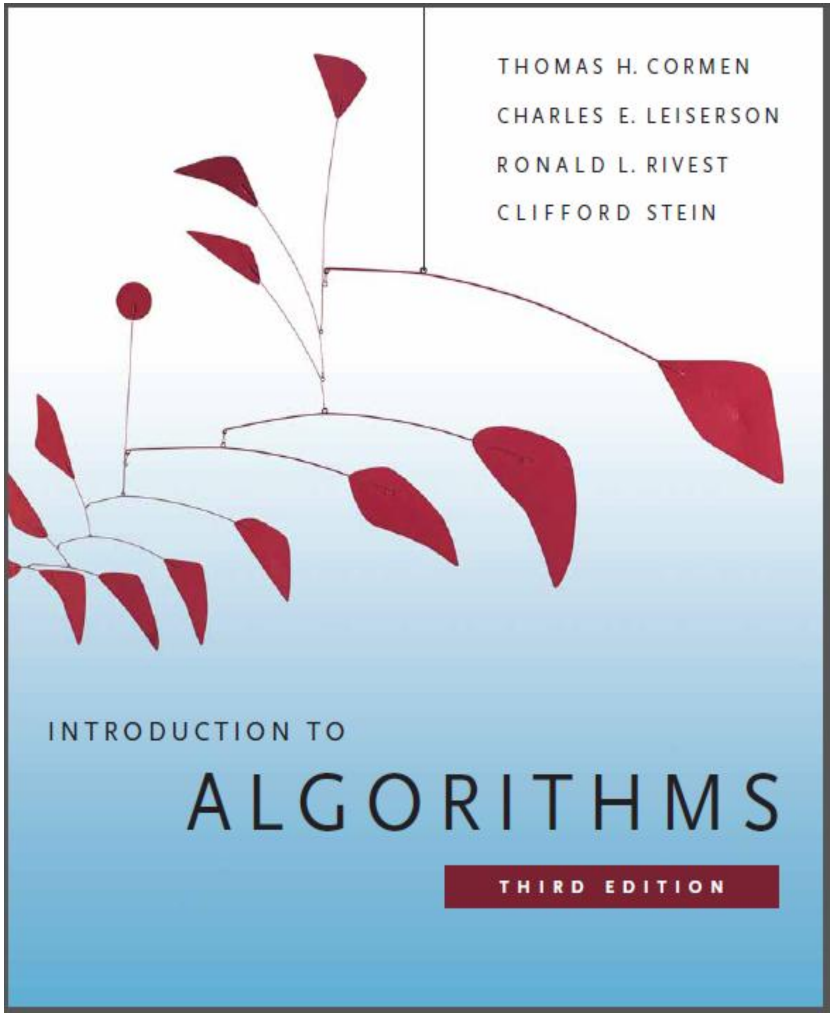
Lecture 1
Introduction

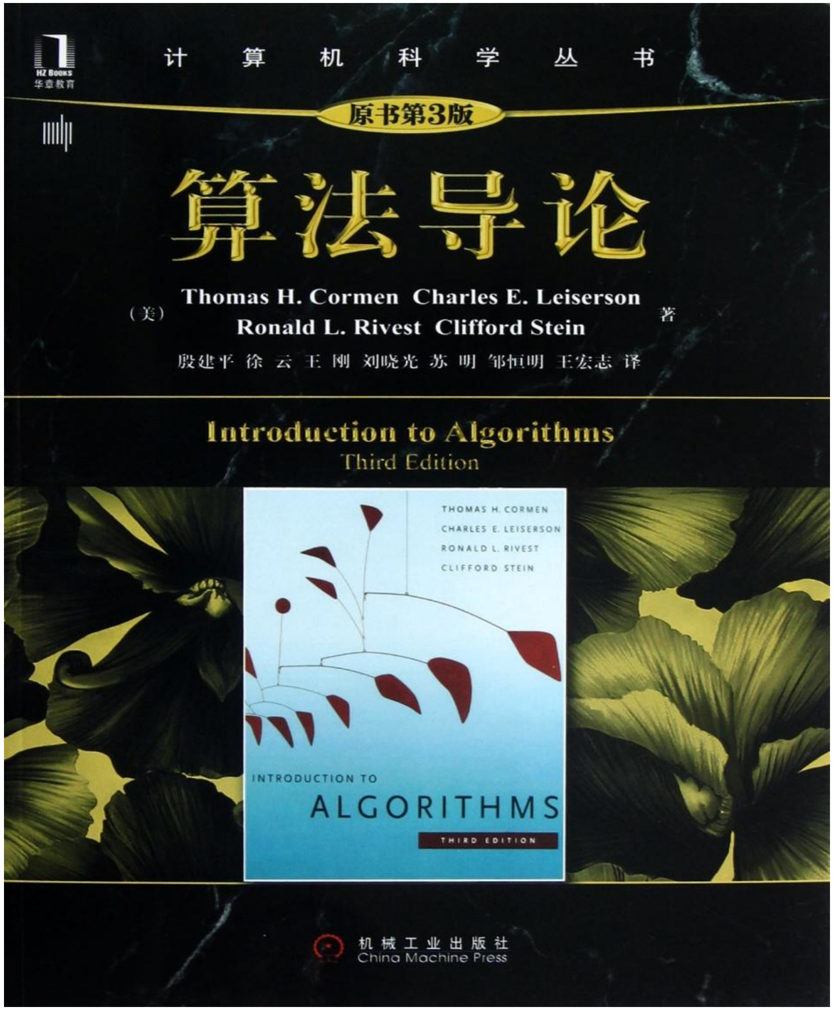
Algorithm Design

zhangzizhen@gmail.com

School of Data and Computer Science, Sun Yat-sen University

Textbook


THOMAS H. CORMEN
CHARLES E. LEISERSON
RONALD L. RIVEST
CLIFFORD STEIN
INTRODUCTION TO
ALGORITHMS
THIRD EDITION


计 算 机 科 学 丛 书
原书第3版
算法导论
（美） Thomas H. Cormen Charles E. Leiserson
Ronald L. Rivest Clifford Stein 著
殷建平 徐 云 王 刚 刘晓光 苏 明 邹恒明 王宏志 译
Introduction to Algorithms
Third Edition
机械工业出版社
China Machine Press

Course Progress

1. Introduction (1 class)	9. NP-completeness (1 cl.)
2. Data structures (1 cl.)	10. Heuristic algorithms (3 cl.)
3. Greedy algorithms (1.5 cl.)	11. Network flows (1 cl.)
4. Divide-and-conquer (1.5 cl.)	12*. Linear programming (1 cl.)
5. Dynamic programming (3 cl.)	13*. Integer programming (1 cl.)
6. Graph algorithms (1 cl.)	14. Project evaluation (1 cl.)
7. Backtracking (1 cl.)	
8. Search algorithms (1.5 cl.)	
Laboratory (5 - 6 cl.)	

http://web.stanford.edu/class/cs161/

Tuesday	Thursday
9/25 Lecture 1: Why are you here? Read: Ch. 1 Slides: [pdf] [pptx] Notes (draft): [pdf]	9/27 Lecture 2: MergeSort, Recurrences, Asymptotics Read: Ch. 2.3, 3 Slides: [pdf] [pptx] Notes (draft): [pdf]
10/2 Lecture 3: Solving Recurrences and the Selection Problem Read: Ch. 4.3-4.5, Ch. 9 Slides: [pdf] [pptx] Notes (draft): [pdf]	10/4 Lecture 4: Randomized Algorithms and QuickSort Read: Ch. 7 and 5.1-5.3 Slides: [pdf] [pptx] Notes (draft): [pdf]
10/9 Lecture 5: Sorting Lower Bounds and O(n)-Time Sorting Read: Ch. 8.1-2 Slides: [pdf] [pptx] Notes (draft): [pdf] Avrim Blum's Notes on sorting lower bounds: [pdf]	10/11 Lecture 6: Binary Search Trees Read: Ch. 12 Slides: [pdf] [pptx] Notes (draft): [pdf]
10/16 MIDTERM 1 (CEMEX Auditorium)	10/18 Lecture 7: Hashing Read: Ch. 11 Slides: [pdf] [pptx] Notes (draft): [pdf]
10/23 Lecture 8: Graphs, BFS, and DFS Read: Ch. 22.1-22.4 Slides: [pdf] [pptx] Notes (draft): [pdf]	10/25 Lecture 9: Finding Strongly Connected Components Read: Ch. 22.5 Slides: [pdf] [pptx] Notes (draft): [pdf]
10/30 Lecture 10: Dynamic Programming and Floyd-Warshall Read: Ch. 25.2, 15.1 Slides: [pdf] [pptx] Notes (draft): [pdf]	11/1 Lecture 11: More Dynamic Programming Read: Ch. 15.4 Slides: [pdf] [pptx] Notes (draft): [pdf]
11/6 MIDTERM 2 (Cubberley Auditorium)	11/8 Lecture 12: Dijkstra's Algorithm and Bellman-Ford Read: Ch. 24.1, 24.3 Slides: [pdf] [pptx] Notes (draft): [pdf]
11/13 Lecture 13: Greedy Algorithms Read: Ch. 16.1, 16.2, 16.3 Slides: [pdf] [pptx]	11/15 Lecture 14: Minimum Spanning Trees Read: Ch. 23 Slides: [pdf] [pptx]
11/20 Thanksgiving Break	11/22 Thanksgiving

Course Evaluation

Attendance	5%
Project	10%
Weekly Programming Assignments	15%
Laboratory Class Reports	15%
Final Online Test	55%

Resources

- ftp: TBD
- The ftp is for you to
  - download slides
  - upload assignments

Current Courses

ID	Name	Teacher	Registrant No.	Status
230	2019软件工程算法设计与分析	张子臻	0	Available

- http://soj.acmm.club/
- Register for the course
  - Weekly assignments will be put here
  - Remember to set your Student ID

Home Problems Contests Courses Ranklist Contest Results Submit Setting Status Management Discuss

Personal Profile Edit

Option	Content	Example/Note
Student ID*		10382001

Overview

1. Introduction
   (a) What are Algorithms?
   (b) Design of Algorithms.
   (c) Analysis of Algorithms.
2. Complexity
   (a) Asymptotic analysis, O and Θ.
   (b) Order of growth.

What are algorithms?

- An algorithm is a well-defined finite set of rules that specifies a sequential series of elementary operations to be applied to some data called the input, producing after a finite amount of time some data called the output.
- An algorithm solves some computational problem.
- Algorithms (along with data structures) are the fundamental “building blocks” from which programs are constructed. Only by fully understanding them is it possible to write very effective programs.

Characteristics of algorithms

- All algorithms must satisfy the following criteria:

  (1) Input（输入）: There are zero or more quantities that are externally supplied.

  (2) Output（输出）: At least one quantity is produced.

  (3) Definiteness（确定性）: Each instruction is clear and unambiguous.

  (4) Finiteness（有穷性）: If we trace out the instructions of an algorithm, then for all cases, the algorithm terminates after finite number of steps.

  (5) Effectiveness（有效性）: Every instruction must be basic enough to be carried out, in principle, by a person using only pencil and paper.

Algorithm design and analysis

- An algorithmic solution to a computational problem will usually involve designing an algorithm, and then analyzing its performance.
- **Design** A good algorithm designer must have a thorough background knowledge of algorithmic techniques, but especially substantial creativity and imagination.
- **Analysis** A good algorithm analyst must be able to carefully estimate or calculate the resources (time, space or other) that the algorithm will use when running. This requires logic, care and often some mathematical ability.
- The aim of this course is to give you sufficient background to understand and appreciate the issues involved in the design and analysis of algorithms.

Design and analysis

- In designing and analyzing an algorithm we should consider the following questions:

  1. What is the problem we have to solve?

  2. Does a solution exist?

  3. Can we find a solution (algorithm), and is there more than one solution?

  4. Is the algorithm correct?

  5. How efficient is the algorithm?

Algorithm design and analysis -- an example

- The Fibonacci sequence is the sequence of integers starting:

  1, 1, 2, 3, 5, 8, 13, 21, 34, 55, . . .

- The formal definition is:

  $F_1 = F_2 = 1$ and $F_n = F_{n-1} + F_{n-2}$.

- Please devise an algorithm to compute F_n.

A naive recursive solution

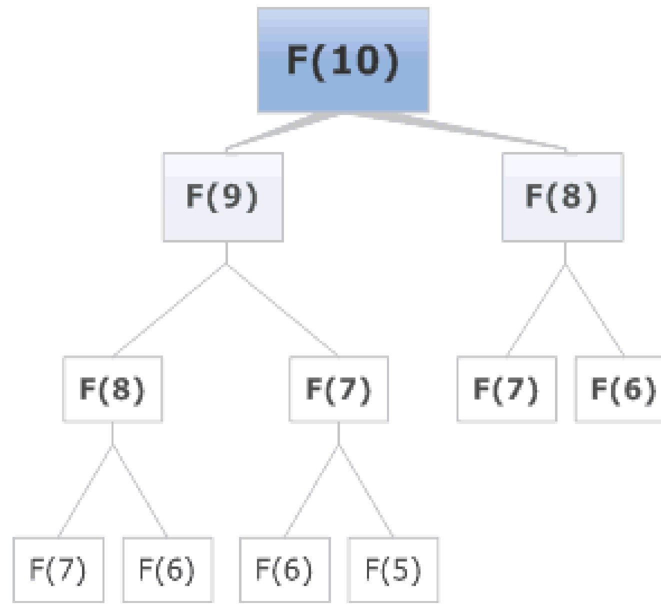
- A naive solution is to simply write a recursive method that directly models the problem.

```
int fib(int n)
{
    return (n < 3 ? 1 : fib(n-1) + fib(n-2));
}
```

- Is this a good algorithm/program in terms of resource usage?
- Timing it on a (2005) iMac gives the following results (the time is in seconds and is for a loop calculating F_n 10000 times).

A naive recursive solution

Value	Time	Value	Time
F_{20}	1.65	F_{24}	9.946
F_{21}	2.51	F_{25}	15.95
F_{22}	3.94	F_{26}	25.68
F_{23}	6.29	F_{27}	41.40



- How long will it take to compute F30, F40 or F50?

- Exercise: Show the number of method calls made to fib() is $2F_n - 1$.

An iterative algorithm

- We can easily re-design the algorithm as an iterative algorithm.

```
int fib(int n) {
    int f_2;            /* F(i+2) */
    int f_1 = 1;        /* F(i+1) */
    int f_0 = 1;        /* F(i) */
    for (int i = 1; i < n; i++) {
        /* F(i+2) = F(i+1) + F(i) */
        f_2 = f_1 + f_0;
        /* F(i) = F(i+1); F(i+1) = F(i+2) */
        f_0 = f_1;
        f_1 = f_2;
    }
    return f_0;
}
```

Value	Time
F_{20}	0.23
F_{21}	0.23
F_{22}	0.23
F_{23}	0.23

Value	Time
F_{10^3}	0.25
F_{10^4}	0.48
F_{10^5}	2.20
F_{10^6}	20.26

A matrix algorithm

- We have the following equation:

$$\begin{bmatrix} f(n) & f(n-1) \\ f(n-1) & f(n-2) \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}^{n-1}$$

- Use Mathematical Induction:

Basis: For n=2,

$$\begin{bmatrix} f(2) & f(1) \\ f(1) & f(0) \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}$$

Inductive step: If n=k, the equation holds, then

$$\begin{bmatrix} f(k) & f(k-1) \\ f(k-1) & f(k-2) \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}^{k-1}$$

A matrix algorithm

Both sides multiply by [1, 1; 1, 0], we get

$$left = \begin{bmatrix} f(k)+f(k-1) & f(k) \\ f(k-1)+f(k-2) & f(k-1) \end{bmatrix} = \begin{bmatrix} f(k+1) & f(k) \\ f(k) & f(k-1) \end{bmatrix}$$

$$right = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}^{k-1} \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}^{k}$$

Thus, the equation holds for *n=k+1*.

- Use the idea of divide-and-conquer

$$A^n = \begin{cases} A^{n/2} * A^{n/2} & n = 2k \\ A^{(n-1)/2} * A^{(n-1)/2} * A & n = 2k-1 \end{cases}$$

A little more…

- General term formula

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

- Prove it use induction.
- Hard to implement.

Evaluating algorithms

- Correctness

  1. Theoretical correctness

  2. Numerical stability

- Efficiency

  1. Time

  2. Space

- An algorithm is efficient if it uses as few resources as possible. In many situations there is a trade-off between time and space, in that an algorithm can be made faster if it uses more space or smaller if it takes longer.
- Although a thorough analysis of an algorithm should consider both time and space, <u>time is considered more important</u>.

Numerical stability

- You can be fairly certain of exact results from a computer program provided all arithmetic is done with the integers $Z = \{. . . ,-3,-2,-1, 0, 1, 2, 3, . . .\}$ and you guard carefully about any overflow.
- However the situation is entirely different when the problem involves real number, because there is necessarily some round-off error when real numbers are stored in a computer.

Accumulation of errors

```c
#include <stdio.h>
int main()
{
    double t = 0.1;
    for (int i = 0; i < 20; i++)
    {
        printf("%.16lf\n", t);
        t += 0.1;
    }
    return 0;
}
```

```
0.1000000000000000
0.2000000000000000
0.3000000000000000
0.4000000000000000
0.5000000000000000
0.6000000000000000
0.7000000000000000
0.7999999999999999
0.8999999999999999
0.9999999999999999
1.0999999999999999
1.2000000000000000
1.3000000000000000
1.4000000000000001
1.5000000000000002
1.6000000000000003
1.7000000000000004
1.8000000000000005
1.9000000000000006
2.0000000000000004
```

Measuring time

- How should we measure the time taken by an algorithm?
- We can do it experimentally by measuring the number of seconds it takes for a program to run — this is often called benchmarking and is often seen in popular magazines. This can be useful, but depends on many factors:
  - The machine on which it is running.
  - The language in which it is written.
  - The skill of the programmer.
  - The instance on which the program is being run, both in terms of size and which particular instance it is.
- So it is not an independent measure of the algorithm, but rather a measure of the implementation, the machine and the instance.

Complexity

- The complexity of an algorithm is a "device-independent" measure of how much time it consumes. Rather than expressing the time consumed in seconds, we attempt to count how many "elementary operations" the algorithm performs when presented with instances of different sizes.
- The result is expressed as a function, giving the number of operations in terms of the size of the instance. This measure is not as precise as a benchmark, but much more useful for answering the kind of questions that commonly arise:
  - I want to solve a problem twice as big. How long will that take me?
  - We can afford to buy a machine twice as fast? What size of problem can we solve in the same time?
- The answers to questions like this depend on the complexity of the algorithm.

Different instances of the same size

- We have assumed that the algorithm takes the same amount of time on every instance of the same size. But this is almost never true, and so we must decide whether to do *best case*, *worst case* or *average case* analysis.
- In *best case* analysis we consider the time taken by the algorithm to be the time it takes on the best input of size *n*.
- In *worst case* analysis we consider the time taken by the algorithm to be the time it takes on the worst input of size *n*.
- In *average case* analysis we consider the time taken by the algorithm to be the average of the times taken on inputs of size *n*.

An example - Insertion sort

```
procedure INSERTION-SORT(A)
1  for j ← 2 to length[A]
2      do key ← A[j]
3          i = j − 1
4          while i > 0 and A[i] > key
5              do A[i + 1] ← A[i]
6                  i = i − 1
7          A[i + 1] ← key
```

- Lines 2-7 will be executed *n* times, lines 4-5 will be executed up to *j* times for *j*=1 to *n*.

Asymptotic notations

- Big-O notation: It defines an asymptotic upper bound for a function *f(n)*.

  Definition: A function *f(n)* is said to be *O(g(n))* if there are constants *c* and n_0 such that

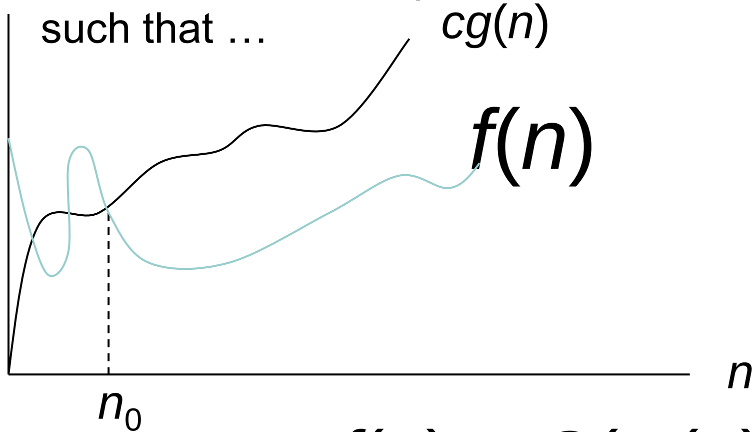
  $f(n) \le cg(n),\ \forall n \ge n_0$.
- Big-Omega notation (Ω): It defines an asymptotic lower bound for a function *f(n)*.
- Big-Theta notation: It defines an asymptotic upper and lower bound for a function *f(n)*.

  Definition: A function *f(n)* is said to be $\Theta(g(n))$ if there are constants c_1, c_2 and n_0 such that

  $0 \le c_1 g(n) \le f(n) \le c_2 g(n),\ \forall n \ge n_0$.
- If we say that $f(n) = \Theta(n^2)$ then we are implying that *f(n)* is approximately proportional to n^2 for large values of *n*.

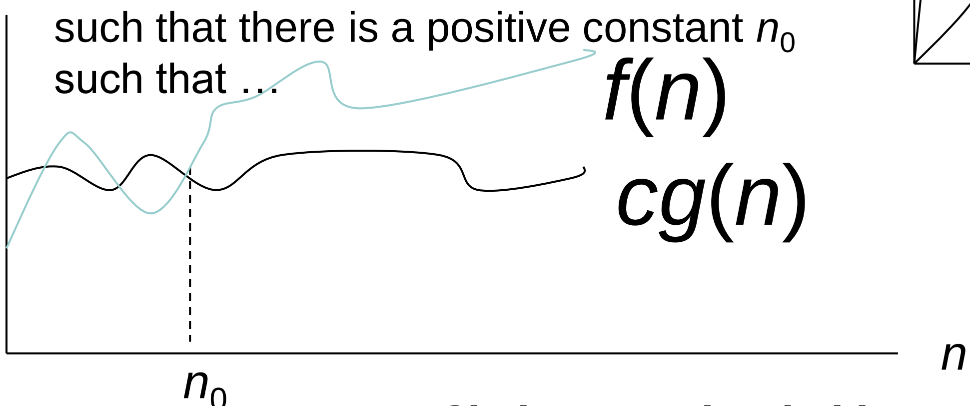
Asymptotic notations

There exist positive constants c such that there is a positive constant n_0 such that …



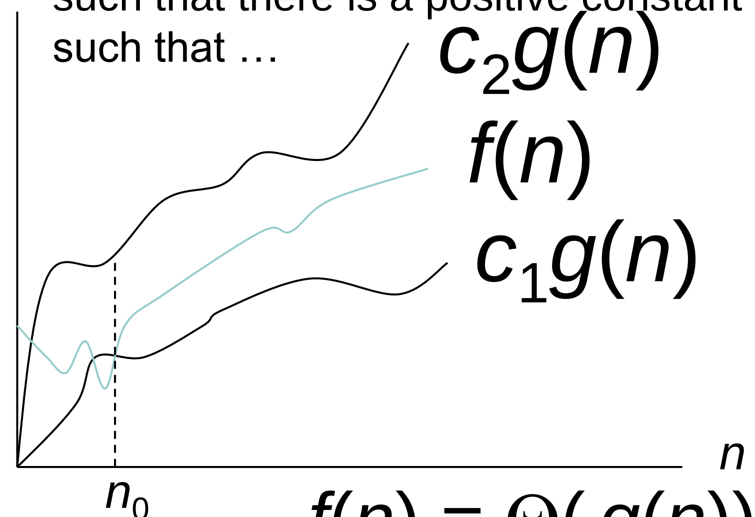
$$f(n) = O(g(n))$$

There exist positive constants c such that there is a positive constant n_0 such that …



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

There exist positive constants c_1 and c_2 such that there is a positive constant n_0 such that …



$$f(n) = \Theta(g(n))$$

Asymptotic notations

Notation: $f(n)$ is	Meaning: Order of f compared to g is		Value of $\lim_{n\to\infty}\left(f(n)/g(n)\right)$
$o\left(g(n)\right)$	$<$	*strictly smaller*	0
$O\left(g(n)\right)$	$\leq$	*smaller or equal*	*finite*
$\Theta\left(g(n)\right)$	$=$	*equal*	*nonzero finite*
$\Omega\left(g(n)\right)$	$\geq$	*larger or equal*	*nonzero*

L'Hôpital's Rule Suppose that:

- $f(x)$ and $g(x)$ are differentiable functions for all sufficiently large x, with derivatives $f'(x)$ and $g'(x)$, respectively.
- $\lim_{x\to\infty} f(x) = \infty$ and $\lim_{x\to\infty} g(x) = \infty$.
- $\lim_{x\to\infty} \frac{f'(x)}{g'(x)}$ exists.

Then $\lim_{x\to\infty} \frac{f(x)}{g(x)}$ exists and $\lim_{x\to\infty} \frac{f(x)}{g(x)} = \lim_{x\to\infty} \frac{f'(x)}{g'(x)}$.

Order of growth

- Asymptotic notations: Θ, O, Ω, o, ω.
- The constant coefficient(s) are ignored.
- Lower order item(s) are ignored, just keep the highest order item.
- The rate of growth, or the order of growth, possesses the highest significance.

- The insertion sort runs in $\Theta(n^2)$.
- $f(n)= \Theta(g(n))$ actually means： $f(n) \in \Theta(g(n))$。

- Typical order of growth: O(1), O(log n), O($\sqrt{n}$), O(n), O($n$log n), O(n^2), O(n^3), O(2^n), O(n!).

- What’s the complexity of each Fib algorithm?

Approximation for factorials

- Stirling's approximation

$$n! = \sqrt{2\pi n}\left(\frac{n}{e}\right)^n\left(1+\Theta\left(\frac{1}{n}\right)\right)$$

$$n! = o(n^n)$$

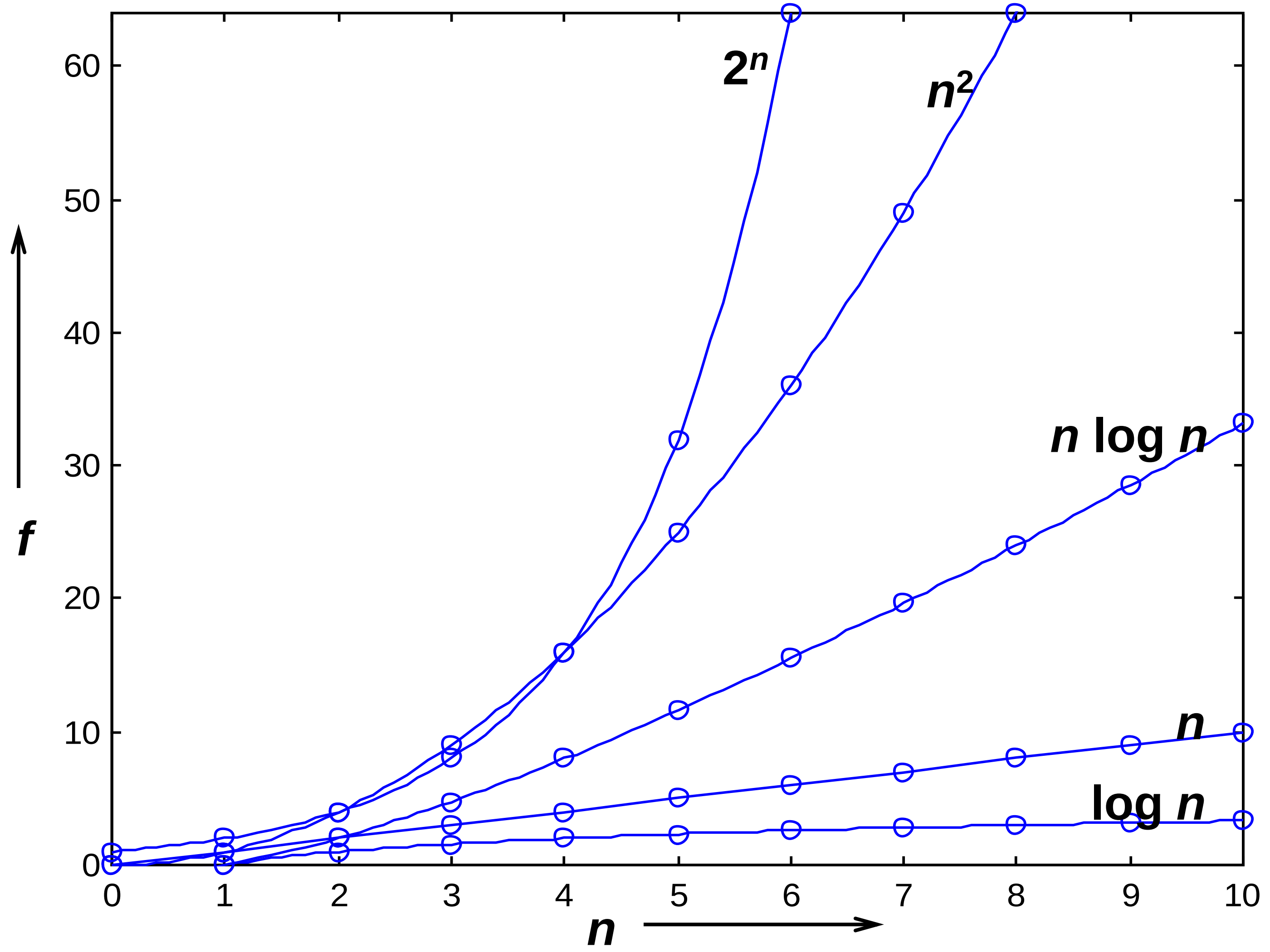
$$n! = \omega(2^n)$$

$$\log(n!) = \Theta(n \log n)$$

Arithmetic operation

- $O(f(n))+O(g(n)) = O(\max\{f(\mathrm{n}),g(n)\})$；
- $O(f(n))+O(g(n)) = O(f(\mathrm{n})+g(n))$；
- $O(f(n))*O(g(n)) = O(f(\mathrm{n})*g(n))$；
- $O(cf(n)) = O(f(\mathrm{n}))$；
- $g(n)= O(f(n)) \Rightarrow O(f(n))+O(g(n)) = O(f(\mathrm{n}))$。

Order of growth



Practice

1. Prove that $F(n) = \frac{1}{\sqrt{5}}\left(\frac{1+\sqrt{5}}{2}\right)^n - \frac{1}{\sqrt{5}}\left(\frac{1-\sqrt{5}}{2}\right)^n$

2. Prove that $f(n)=an^2+bn+c=O(n^2)$.

3. Prove that $O(f(n))+O(g(n)) = O(\max\{f(n),g(n)\})$.

Thank you!