Lecture 1
Introduction

Spring 2023

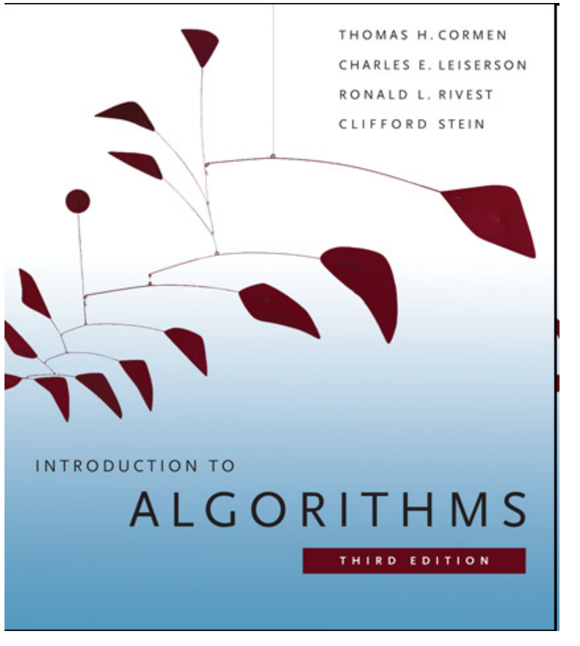
Zhihua Jiang

Textbook & Grade

- Thomas Cormen, Charles Leiserson, Ronald Rivest, and Clifford Stein. Introduction to Algorithms. 3rd ed. MIT Press, 2009. (CLRS book)

  ISBN: 9780262033848.

- Homework+ final examination



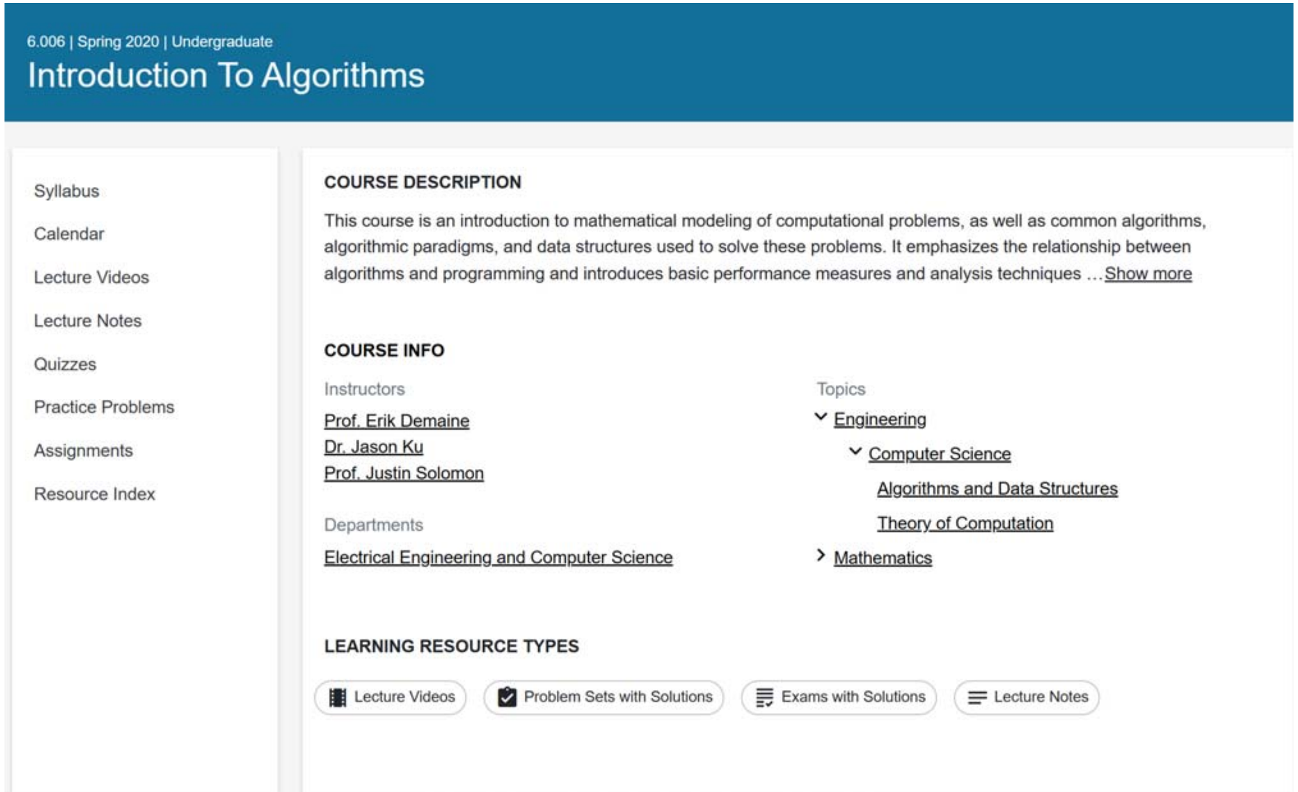
TIPS:

- ✓ The book's electronic version uploaded in the QQ file section;
- ✓ A lot of resources (e.g., videos, lectures, projects, assignments, solutions) are available;
- ✓ To implement algorithms, programming languages like Python/C++/Java are commonly used;
- ✓ The bible book + teacher's lectures, a recommended way to learn this course.

Resources

- MIT 6.006 2020

https://ocw.mit.edu/courses/6-006-introduction-to-algorithms-spring-2020/



MIT Algorithm courses

I. Introduction to Algorithms (6.006)
II. Design and Analysis of Algorithms (6.046J)
III. Advanced Algorithms (6.854J)

LEC #	TOPICS
Unit 1: Introduction	
1	Algorithmic thinking, peak finding
2	Models of computation, Python cost model, document distance
Unit 2: Sorting and Trees	
3	Insertion sort, merge sort
4	Heaps and heap sort
5	Binary search trees, BST sort
6	AVL trees, AVL sort
7	Counting sort, radix sort, lower bounds for sorting and searching
Unit 3: Hashing	
8	Hashing with chaining
9	Table doubling, Karp-Rabin
10	Open addressing, cryptographic hashing
	Quiz 1
Unit 4: Numerics	
11	Integer arithmetic, Karatsuba multiplication
12	Square roots, Newton's method
Unit 5: Graphs	
13	Breadth-first search (BFS)

SESSION	TOPICS
L1	Overview, Interval Scheduling
L2	Divide & Conquer: Convex Hull, Median Finding
R1	Divide & Conquer: Smarter Interval Scheduling, Master Theorem, Strassen's Algorithm
L3	Divide & Conquer: FFT
R2	B-trees
L4	Divide & Conquer: Van Emde Boas Trees
R3	Amortization: Union-find
L5	Amortization: Amortized Analysis
L6	Randomization: Matrix Multiply, Quicksort
R4	Randomization: Randomized Median
L7	Randomization: Skip Lists
L8	Randomization: Universal & Perfect Hashing
R5	Dynamic Programming: More Examples
L9	Augmentation: Range Trees
L10	Dynamic Programming: Advanced DP
L11	Dynamic Programming: All-pairs Shortest Paths
L12	Greedy Algorithms: Minimum Spanning Tree
R6	Greedy Algorithms: More Examples
L13	Incremental Improvement: Max Flow, Min Cut
L14	Incremental Improvement: Matching
R7	Incremental Improvement: Applications of Network Flow & Matching
L15	Linear Programming: LP, Reductions, Simplex
L16	Complexity: P, NP, NP-completeness, Reductions
R8	Complexity: More Reductions

Content in this course

- Introduction
- Divide and conquer
- Computation models
- Sorting and trees
- Dynamic programming
- Greedy algorithm
- NP completeness

Definition of algorithms

A sequence of steps which is used to solve a category of problems.

- Unambiguous: every step is deterministic;
- Mechanical: machine can “understand”;
- Finite: can be implemented in limited steps;
- Input/output: to state the problem size and the result.

Algorithm representation

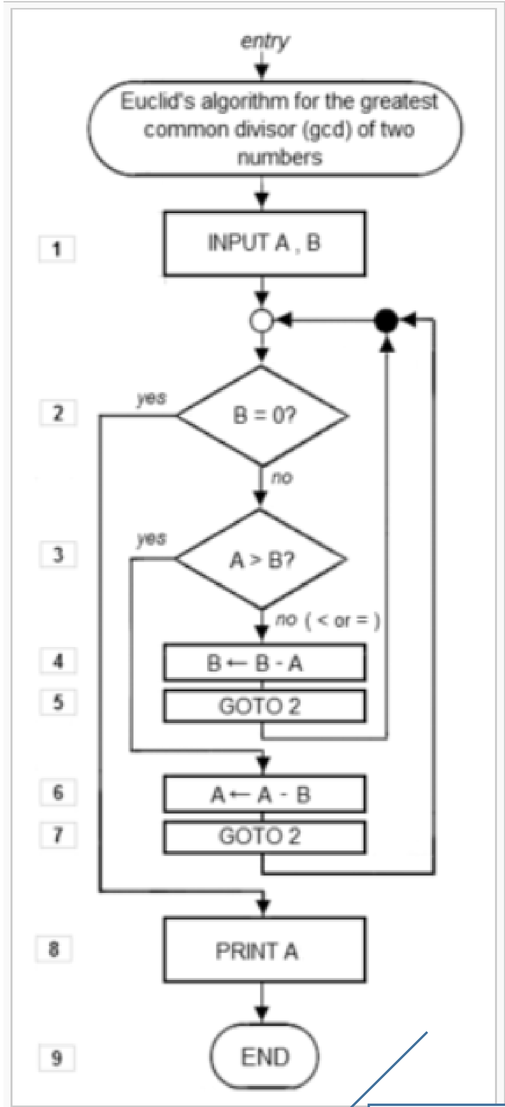
```
5 REM Euclid's algorithm for greatest common divisor
6 PRINT "Type two integers greater than 0"
10 INPUT A,B
20 IF B=0 THEN GOTO 80
30 IF A > B THEN GOTO 60
40 LET B=B-A
50 GOTO 20
60 LET A=A-B
70 GOTO 20
80 PRINT A
90 END
```

Program

<u>function Euclid</u>(a, b)
Input: Two integers a and b with $a \geq b \geq 0$
Output: $\gcd(a, b)$

if $b = 0$: return a
return Euclid$(b, a \bmod b)$

√ Pseudocode



Flow chart

Pseudocode conventions

We use the following conventions in our pseudocode.

- Indentation indicates block structure. For example, the body of the **for** loop that begins on line 1 consists of lines 2–8, and the body of the **while** loop that begins on line 5 contains lines 6–7 but not line 8. Our indentation style applies to **if-else** statements[2] as well. Using indentation instead of conventional indicators of block structure, such as **begin** and **end** statements, greatly reduces clutter while preserving, or even enhancing, clarity.[3]
- The looping constructs **while**, **for**, and **repeat-until** and the **if-else** conditional construct have interpretations similar to those in C, C++, Java, Python, and Pascal.[4] In this book, the loop counter retains its value after exiting the loop, unlike some situations that arise in C++, Java, and Pascal. Thus, immediately after a **for** loop, the loop counter's value is the value that first exceeded the **for** loop bound. We used this property in our correctness argument for insertion sort. The **for** loop header in line 1 is **for** $j = 2$ **to** $A.length$, and so when this loop terminates, $j = A.length + 1$ (or, equivalently, $j = n + 1$, since $n = A.length$). We use the keyword **to** when a **for** loop increments its loop

counter in each iteration, and we use the keyword **downto** when a **for** loop decrements its loop counter. When the loop counter changes by an amount greater than 1, the amount of change follows the optional keyword **by**.

- The symbol "**//**" indicates that the remainder of the line is a comment.
- A multiple assignment of the form $i = j = e$ assigns to both variables i and j the value of expression e; it should be treated as equivalent to the assignment $j = e$ followed by the assignment $i = j$.
- Variables (such as i, j, and *key*) are local to the given procedure. We shall not use global variables without explicit indication.
- We access array elements by specifying the array name followed by the index in square brackets. For example, $A[i]$ indicates the ith element of the array A. The notation "$..$" is used to indicate a range of values within an array. Thus, $A[1..j]$ indicates the subarray of A consisting of the j elements $A[1], A[2], \ldots, A[j]$.
- We typically organize compound data into ***objects***, which are composed of ***attributes***. We access a particular attribute using the syntax found in many object-oriented programming languages: the object name, followed by a dot, followed by the attribute name. For example, we treat an array as an object with the attribute *length* indicating how many elements it contains. To specify the number of elements in an array A, we write $A.length$.

a pointer to the array is passed, rather than the entire array, and changes to individual array elements are visible to the calling procedure.

- A **return** statement immediately transfers control back to the point of call in the calling procedure. Most **return** statements also take a value to pass back to the caller. Our pseudocode differs from many programming languages in that we allow multiple values to be returned in a single **return** statement.
- The boolean operators "and" and "or" are ***short circuiting***. That is, when we evaluate the expression "x and y" we first evaluate x. If x evaluates to FALSE, then the entire expression cannot evaluate to TRUE, and so we do not evaluate y. If, on the other hand, x evaluates to TRUE, we must evaluate y to determine the value of the entire expression. Similarly, in the expression "x or y" we evaluate the expression y only if x evaluates to FALSE. Short-circuiting operators allow us to write boolean expressions such as "$x \neq$ NIL and $x.f = y$" without worrying about what happens when we try to evaluate $x.f$ when x is NIL.
- The keyword **error** indicates that an error occurred because conditions were wrong for the procedure to have been called. The calling procedure is responsible for handling the error, and so we do not specify what action to take.

Algorithm vs. Program

- Some people maybe think that algorithms are just programs. We need to clarity that **programs could be a way to express algorithms**, and they are not exactly same.
- The big difference between algorithms and programs is that **algorithms are for people to communicate** while **programs are for machines to run**.
- Generally, algorithms cannot be directly executed in computers. In additions, it is too precise when we use programs to express algorithms.

Example of T(n)

INSERTION-SORT(A)	*cost*	*times*
for $j = 2$ **to** $A.length$	c_1	n
$\quad key = A[j]$	c_2	$n-1$
$\quad$ // Insert $A[j]$ into the sorted sequence $A[1..j-1]$.	0	$n-1$
$\quad i = j-1$	c_4	$n-1$
$\quad$ **while** $i > 0$ and $A[i] > key$	c_5	$\sum_{j=2}^{n} t_j$
$\qquad A[i+1] = A[i]$	c_6	$\sum_{j=2}^{n}(t_j - 1)$
$\qquad i = i-1$	c_7	$\sum_{j=2}^{n}(t_j - 1)$
$\quad A[i+1] = key$	c_8	$n-1$

The running time of the algorithm is the sum of running times for each statement executed; a statement that takes c_i steps to execute and executes n times will contribute $c_i n$ to the total running time.[6] To compute $T(n)$, the running time of INSERTION-SORT on an input of n values, we sum the products of the *cost* and *times* columns, obtaining

$$T(n) = c_1 n + c_2(n-1) + c_4(n-1) + c_5 \sum_{j=2}^{n} t_j + c_6 \sum_{j=2}^{n}(t_j - 1) + c_7 \sum_{j=2}^{n}(t_j - 1) + c_8(n-1) .$$

t_j: the number of movements to insert the jth element

Ex: 4 2 8 3
j=2 key=2, t2=1
j=3 key=8, t3=0
j=4 key=3, t4=2

Asymptotic notation

- Why not express running time in terms of basic computer steps?
  - Too precise: you need to count the times carefully.
  - Depend on particular machines: the cost could be different.

- Simplification: $5n^3+4n+3 \rightarrow 5n^3 \rightarrow n^3$
  - Leave out **lower-order terms** (insignificant as n grows)
  - Leave out **the coefficient in *the leading term*** (computers will be faster)

Finally, $5n^3+4n+3 = O(n^3)$

O notation

- Definition:

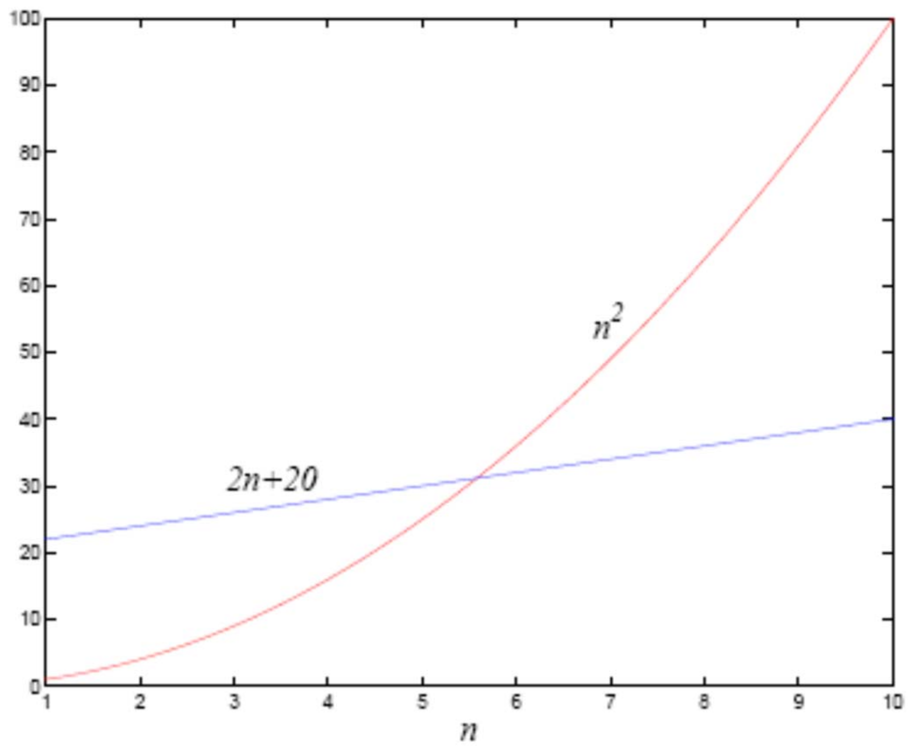
Let $f(n)$ and $g(n)$ be functions from positive integers to positive reals. We say $f = O(g)$ (which means that "f grows no faster than g") if *there is a constant $c > 0$ such that $f(n) \leq c \cdot g(n)$.*

- *n: size of problem*
- $f = O(g)$ is a loose analog of "$f \leqslant g$". It differs from the usual notion $\leqslant$ because of the constant c.

E.g., $2n+20 = O(n^2)$, $n^2 \neq O(2n+20)$, $2n+20 = O(n+1)$

$$\frac{f_2(n)}{f_1(n)} = \frac{2n+20}{n^2} \leq 22 \qquad \frac{f_2(n)}{f_3(n)} = \frac{2n+20}{n+1} \leq 20$$

O notation



- $2n+20 = O(n^2)$: for $n \leqslant 5$, n^2 is smaller; $n > 5$, n^2 is larger.
- *But,* $2n+20$ scales much better than n^2 as n grows. (i.e., $2n+20$ grows no faster than n^2)

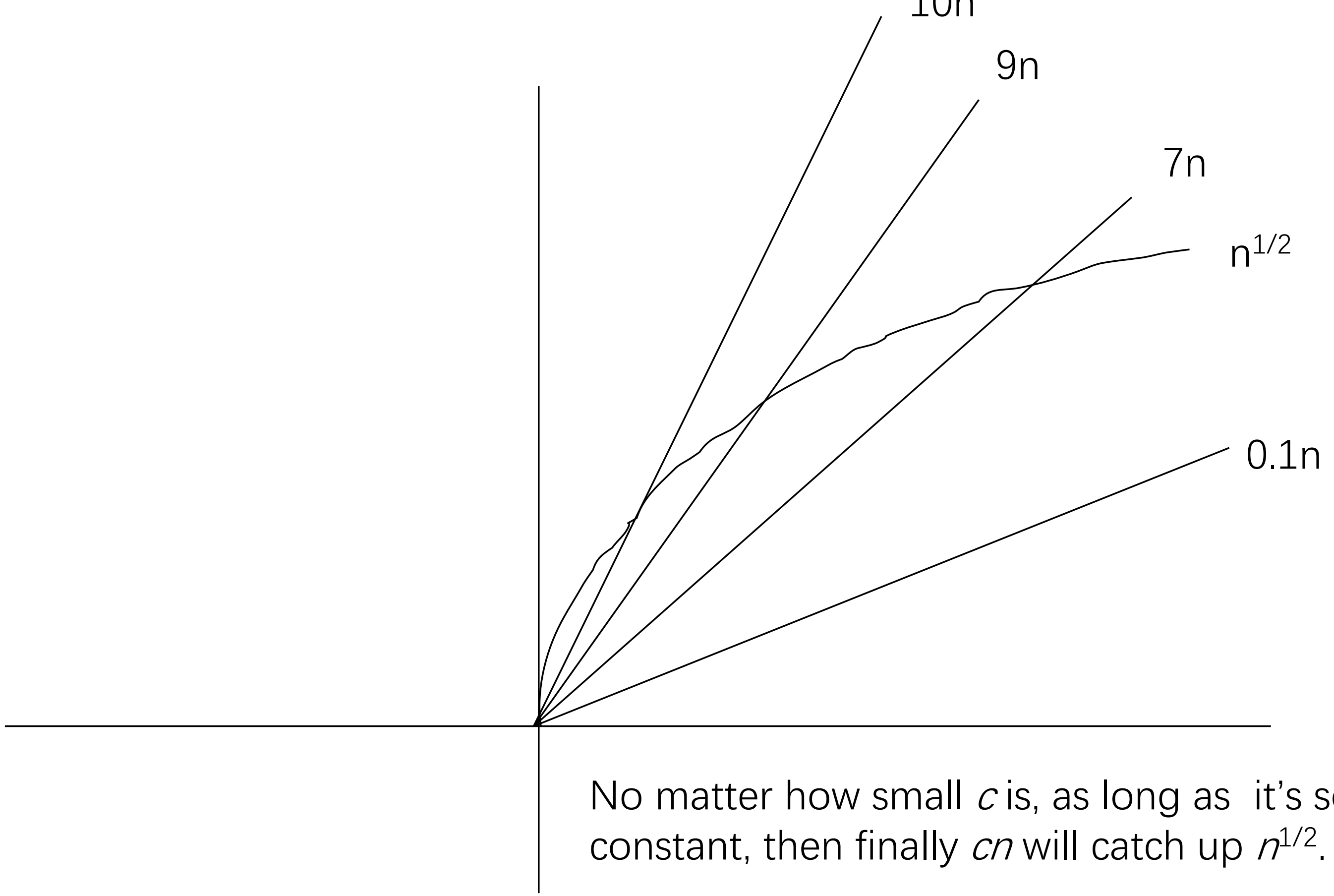
Extended definitions: O, Ω, Θ

- $f(n) = O(n)$: $f(n) \leq c \cdot n$ for some constant c and large n.
  - i.e. $\exists c$, $\exists N > 0$ s.t. $\forall n > N$, we have $f(n) \leq c \cdot n$.

- For example, $f(n) = 10n$.
  - Let $c = 11$, $N = 1$, then $\forall n > N$, we have $10n \leq 11n = 10n + n$
  - So $10n = O(n)$.

- How about, $f(n) = 10n + 5$?
  - Let $c = 11$, $N = 5$, then $\forall n > N$, we have $10n + 5 \leq 11n = 10n + n$
  - So $10n + 5 = O(n)$.

Example: $n^{1/2}$ vs. cn



No matter how small c is, as long as it's some positive constant, then finally cn will catch up $n^{1/2}$.

General definitions

- $f(n) = O(g(n))$: for some constant c, $f(n) \leq c \cdot g(n)$, when n is sufficiently large.
  - i.e. $\exists c$, $\exists N$ s.t. $\forall n > N$, we have $f(n) \leq c \cdot g(n)$.

- $f(n) = o(g(n))$: for any constant c, $f(n) < c \cdot g(n)$, when n is sufficiently large.
  - i.e. $\forall c$, $\exists N$ s.t. $\forall n > N$, we have $f(n) < c \cdot g(n)$.

General definition (cont.)

- $f(n) = \Omega(g(n))$: $f(n) \geq c \cdot g(n)$ for some constant c and large n.
  - i.e. $\exists c$, $\exists N$ s.t. $\forall n > N$, we have $f(n) \geq c \cdot g(n)$.

- $f(n) = \omega(g(n))$: $f(n) > c \cdot g(n)$ for any constant c and large n.
  - i.e.$\forall c$, $\exists N$ s.t. $\forall n > N$, we have $f(n) > c \cdot g(n)$.

- $f(n) = \Theta(g(n))$: $f(n) = O(g(n))$ and $f(n) = \Omega(g(n))$
  - i.e. $c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n)$ for two constants c_1 and c_2 and large n.

Spectrum of functions

|← exponential →| tower

O(1) , ⋯, log* n, ⋯ loglog n, ⋯ log n, log² n, ⋯ ⋯ $n^{1/3}$, $n^{1/2}$, n, n^2, n^3, ⋯ ⋯ 2^n, 2^{n^2}, ⋯ 2^{2^n}, ⋯ $2^{2^{\cdots^2}}$ ⋯

|← constant →| |← logarithmic →| |← polynomial →| double exponential

- $2^{2^{\cdots^2}}$ a tower of height n.
- Faster? 2^{2^n}, $2^{2^{2^n}}$, ⋯
- Exponential: 2^n, 1.001^n, 2^{n^2}
- Polynomial: n, n^2, n^3, n^{100}, $n^{1/2}$, $n^{1/3}$, $n^{0.1}$, $n^{0.01}$,
- Logarithmic: $\log n$, $\log^2 n$, $\log^{1/2} n$,

- Slower? $\log\log n$, $\log\log\log n$, $\cdots \log^* n$.
  - If you take log, how many times to make n down to < 1?
  - E.g., $\log\log\log\log(1024) = 0.79245629369$.
  - So $\log^* n$ is practically a constant.

Commonsense rules

1. Multiplicative constants can be omitted: $14n^2$ becomes n^2.
2. n^a dominates n^b if $a > b$: for instance, n^2 dominates n.
3. Any exponential dominates any polynomial: 3^n dominates n^5 (it even dominates 2^n).
4. Likewise, any polynomial dominates any logarithm: n dominates $(\log n)^3$. This also means, for example, that n^2 dominates $n \log n$.

Properties of asymptotic notations

(1) Transitivity**:**

- $f(n)=\Theta(g(n))$, $g(n)=\Theta(h(n)) \Rightarrow f(n)=\Theta(h(n))$;
- $f(n)=O(g(n))$, $g(n)=O(h(n)) \Rightarrow f(n)=O(h(n))$;
- $f(n)=\Omega(g(n))$, $g(n)=\Omega(h(n)) \Rightarrow f(n)=\Omega(h(n))$;
- $f(n)=o(g(n))$, $g(n)=o(h(n)) \Rightarrow f(n)=o(h(n))$;
- $f(n)=\omega(g(n))$, $g(n)=\omega(h(n)) \Rightarrow f(n)=\omega(h(n))$;

(2) Reflexivity

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

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

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

(3) Symmetry

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

(4) Mutual symmetry

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

$f(n) = o(g(n)) \Leftrightarrow g(n) = \omega(f(n))$

(5) Arithmetic operations:

$O(f(n)) + O(g(n)) = O(\max\{f(n), g(n)\})$

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

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

$O(cf(n)) = O(f(n))$

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

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

Exercise 0.1

In each of the following situations, indicate whether $f = O(g)$, or $f = \Omega(g)$, or both (in which case $f = \Theta(g)$).

	$f(n)$	$g(n)$
(a)	$n - 100$	$n - 200$
(b)	$n^{1/2}$	$n^{2/3}$
(c)	$100n + \log n$	$n + (\log n)^2$
(d)	$n \log n$	$10n \log 10n$
(e)	$\log 2n$	$\log 3n$
(f)	$10 \log n$	$\log(n^2)$
(g)	$n^{1.01}$	$n \log^2 n$
(h)	$n^2 / \log n$	$n(\log n)^2$
(i)	$n^{0.1}$	$(\log n)^{10}$
(j)	$(\log n)^{\log n}$	$n / \log n$
(k)	$\sqrt{n}$	$(\log n)^3$
(l)	$n^{1/2}$	$5^{\log_2 n}$
(m)	$n2^n$	3^n
(n)	2^n	2^{n+1}
(o)	$n!$	2^n
(p)	$(\log n)^{\log n}$	$2^{(\log_2 n)^2}$
(q)	$\sum_{i=1}^{n} i^k$	n^{k+1}

a) $n - 100 = \Theta(n - 200)$

b) $n^{1/2} = O(n^{2/3})$

c) $100n + \log n = \Theta(n + (\log n)^2)$

d) $n \log n = \Theta(10n \log 10n)$

e) $\log 2n = \Theta(\log 3n)$

f) $10 \log n = \Theta(\log(n^2))$

g) $n^{1.01} = \Omega\left(n\left(\log^2 n\right)\right)$

h) $n^2 / \log n = \Omega(n(\log n)^2)$

i) $n^{0.1} = \Omega((\log n)^{10})$

j) $(\log n)^{\log n} = \Omega(n / \log n)$

k) $\sqrt{n} = \Omega((\log n)^3)$

l) $n^{1/2} = O(5^{\log_2 n})$

m) $n2^n = O(3^n)$

n) $2^n = \Theta(2^{n+1})$

o) $n! = \Omega(2^n)$

p) $(\log n)^{\log n} = O(2^{(\log_2 n)^2})$

q) $\sum_{i=1}^{n} i^k = \Theta(n^{k+1})$

$$a)\ \frac{n-100}{n-200}=1+\frac{100}{n-200}\le 2\quad (n\to\infty)$$

$$c)\ (\log n)^2 = O(n)$$

$$g)\ n^{1.01}=n\cdot n^{0.01}=\Omega(n(\log^2 n))$$

$$h)\ \frac{n^2/\log n}{n(\log n)^2}=\frac{n}{(\log n)^3}\to\infty\quad (n\to\infty)$$

$$j)\ \frac{(\log n)^{\log n}}{n/\log n}=\frac{(\log n)^{\log n+1}}{n},(\log n)^{\log n}=n^{\log\log n}$$

$$or\ let\ k=\log n, so\ \frac{k^{k+1}}{2^k}=\left(\frac{k}{2}\right)^k\cdot k\to\infty(k\to\infty)$$

$$k)\ \frac{\sqrt{n}}{(\log n)^3}=\sqrt{\frac{n}{(\log n)^6}}$$

a) $n-100=\Theta(n-200)$
b) $n^{1/2}=O(n^{2/3})$
c) $100n+\log n=\Theta(n+(\log n)^2)$
d) $n\log n=\Theta(10n\log 10n)$
e) $\log 2n=\Theta(\log 3n)$
f) $10\log n=\Theta(\log(n^2))$
g) $n^{1.01}=\Omega(n(\log^2 n))$
h) $n^2/\log n=\Omega(n(\log n)^2)$
i) $n^{0.1}=\Omega((\log n)^{10})$
j) $(\log n)^{\log n}=\Omega(n/\log n)$
k) $\sqrt{n}=\Omega((\log n)^3)$

$$l)\, 5^{\log_2 n} = n^{\log_2 5}$$

$$o)\, n! = n \times (n-1) \times (n-2) \times \ldots \times 1 > 2 \times 2 \times 2 \times \ldots \times 2 \text{ (when } n \geq 4)$$

$$\text{so } n! = \Omega(2^n)$$

$$p)\, 2^{(\log_2 n)^2} = 2^{(\log_2 n) \bullet (\log_2 n)} = n^{\log_2 n}$$

l) $n^{1/2} = O(5^{\log_2 n})$
m) $n2^n = O(3^n)$
n) $2^n = \Theta(2^{n+1})$
o) $n! = \Omega(2^n)$
p) $(\log n)^{\log n} = O(2^{(\log_2 n)^2})$
q) $\sum_{i=1}^{n} i^k = \Theta(n^{k+1})$

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

$$n! = \sqrt{2\pi n}\left(\frac{n}{e}\right)^n\left[1+\theta\left(\frac{1}{n}\right)\right] \text{ (Stirling's formula)}$$

$$\lim_{n\to\infty}\frac{n!}{n^n} = \frac{\sqrt{2\pi n}\left[1+\theta\left(\frac{1}{n}\right)\right]}{e^n} = 0 \Rightarrow n! = o(n^n)$$

$n! = n\times(n-1)\times(n-2)\times\ldots\times 1 < n\times n\times n\times\ldots\times n$

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

prove: $\log(n!)=\theta(n\log n)$

$$\log(n!) = \sum_{i=1}^{n}\log i \le \sum_{i=1}^{n}\log n = n\log n \;\Rightarrow\; \log(n!)=O(n\log n)$$

if n is even (when n is odd, use $\lfloor n/2 \rfloor$ to replace $n/2$):

$$\log(n!) \ge \sum_{i=n/2}^{n}\log i \ge \sum_{i=n/2}^{n}\log(n/2) = n/2\log(n/2) = (n\log n)/2 - n/2$$

when $n \ge 4$, $(n\log n)/2 - n/2 \ge (n\log n)/4$ (since $(n\log n) \ge 2n$)

so $\log(n!) \ge (n\log n)/4 \Rightarrow \log(n!)=\Omega(n\log n)$

therefore, $\log(n!)=\theta(n\log n)$

q)there are n difference formulae:

$$\{i=1\}:2^{k+1}-1^{k+1}=\binom{k+1}{1}\times 1^k+\binom{k+1}{2}\times 1^{k-1}+\ldots+\binom{k+1}{k}\times 1+1$$

$$\{i=2\}:3^{k+1}-2^{k+1}=\binom{k+1}{1}\times 2^k+\binom{k+1}{2}\times 2^{k-1}+\ldots+\binom{k+1}{k}\times 2+1$$

$$\{i=3\}:4^{k+1}-3^{k+1}=\binom{k+1}{1}\times 3^k+\binom{k+1}{2}\times 3^{k-1}+\ldots+\binom{k+1}{k}\times 3+1$$

...

$$\{i=j\}:(j+1)^{k+1}-j^{k+1}=\binom{k+1}{1}j^k+\binom{k+1}{2}j^{k-1}+\ldots+\binom{k+1}{k}j+1$$

$$\{i=j+1\}:(j+2)^{k+1}-(j+1)^{k+1}=\binom{k+1}{1}(j+1)^k+\binom{k+1}{2}(j+1)^{k-1}+\ldots+\binom{k+1}{k}(j+1)+1$$

...

$$\{i=n\}:(n+1)^{k+1}-n^{k+1}=\binom{k+1}{1}n^k+\binom{k+1}{2}n^{k-1}+\ldots+\binom{k+1}{k}n+1$$

then add these n formulae together (only the first term and the last term left):

$$(n+1)^{k+1}-1^{k+1}=\binom{k+1}{1}\sum_{i=1}^{n}i^k+\binom{k+1}{2}\sum_{i=1}^{n}i^{k-1}+\ldots+\binom{k+1}{k}\sum_{i=1}^{n}i+n$$

$$n^{k+1}\le(n+1)^{k+1}\le\left[\binom{k+1}{1}+\binom{k+1}{2}+\ldots+\binom{k+1}{k}\right]\sum_{i=1}^{n}i^k\Rightarrow n^{k+1}=O\left(\sum_{i=1}^{n}i^k\right)$$

$$\sum_{i=1}^{n}i^k\le n\cdot n^k=n^{k+1}\Rightarrow n^{k+1}=\Omega\left(\sum_{i=1}^{n}i^k\right)$$

so $\sum_{i=1}^{n}i^k=\theta(n^{k+1})$

$$C_n^m=\frac{P_n^m}{P_m}=\frac{n!}{m!(n-m)!},C_n^0=1.$$

$\binom{n}{m},C(n,m),\cdots.$

q) $\sum_{i=1}^{n}i^k=\Theta(n^{k+1})$

Exercise 0.2

Show that, if c is a positive real number, then $g(n) = 1 + c + c^2 + \cdots + c^n$ is:

(a) $\Theta(1)$ if $c < 1$.

(b) $\Theta(n)$ if $c = 1$.

(c) $\Theta(c^n)$ if $c > 1$.

The moral: in big-Θ terms, the sum of a geometric series is simply the first term if the series is strictly decreasing, the last term if the series is strictly increasing, or the number of terms if the series is unchanging.

0.2. By the formula for the sum of a partial geometric series, for $c \neq 1$: $g(n) = \frac{1-c^{n+1}}{1-c} = \frac{c^{n+1}-1}{c-1}$.

a) $1 > 1 - c^{n+1} > 1 - c$. So: $\frac{1}{1-c} > g(n) > 1$.

b) For $c = 1$, $g(n) = 1 + 1 + \cdots + 1 = n + 1$.

c) $c^{n+1} > c^{n+1} - 1 > c^n$. So: $\dfrac{c^{n+1}}{c-1} > g(n) > \dfrac{c^n}{c-1}$.