Introduction

1. Algorithms?
2. Order
3. Analysis of Algorithm
4. Some Mathematical Background

What is an **algorithm**?

Simple, unambiguous, mechanical procedure to carry out some task.

Why algorithm instead of program?

1. writing an algorithm is simpler(we don't need to worry about the detailed implementation, or the language syntax)

2. an algorithm is easier to read (if we write eg: an C program to solve a problem, then the other person cannot understand the idea of the problem solving unless he/she understands C).

How to represent an algorithm?

1. Give a description in your own language, e.g. English, Spanish, …

2. pseudo code

3. Graphical

Example – multiplying two positive integers A and B

For example: 45*19

Usually:

```
  45
  19   (x
------------
 405
 45    (+
------------
 855
```

A different algorithm:

Multiplier (A/2)	Multiplicand (B*2)	Result (pick numbers in column 2 when the corresponding number under the multiplier is odd)
45	19	19
22	38	
11	76	76
5	152	152
2	304	
1	608	608 (+
		855

An instance of a problem is a specific assignment of values to the parameters. This algorithm can be used for multiplying any two positive integers, we say that (45, 19) is an **instance** of this problem. Most problems have infinite collection of instances. It's ok to define the **domain** (i.e. the set of instances) to be considered. And the algorithm should work for all instances in that domain. Although the above algorithm will not work if the first operand is negative, this does not invalidate the algorithm since (-45, 19) is not an instance of the problem being considered.

Order:

Usually we use the **frequency count** to compare algorithms. Consider the following 3 programs:

```
(a)
x ← x + y

(b)
for i ← 1 to n do
    x ← x + y
end

(c)
for  i ← 1 to n do
   for j ← 1 to n do
      x ← x + y
   end
end
```

The frequency count of stmt $x \leftarrow x + y$ is 1, n, n^2.
$\therefore$ no matter which machine we use to run these programs, we know that the execution time of (b) is n times the execution time of (a).

Big-O Notation:

Def $f(n) = O(g(n))$ if and only if $\exists$ 2 positive constants c and n_0, such that

$$|f(n)| \leq c \bullet |g(n)| \ \forall \ n \geq n_0.$$

So, g(n) actually is the upper bound of f(n).

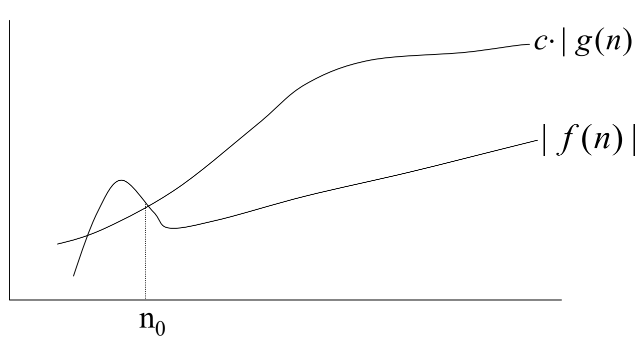


Figure 1. Illustrating "big O"

Examples:

- Is 17 $n^2 - 5 = O(n^2)$?

$$\because \quad 17n^2 - 5 \le 17n^2 \qquad \forall n \ge 1$$
$$(c = 17, n_0 = 1)$$
$$\therefore \quad 17n^2 - 5 = O(n^2)$$

- Is 35 $n^3 + 100 = O(n^3)$?

$$\because \quad 35n^3 + 100 \le 36n^3 \qquad \forall n \ge 5$$
$$(c = 36, n_0 = 5)$$
$$\therefore \quad 35n^3 + 100 = O(n^3)$$

- Is 6 • $2^n + n^2 = O(2^n)$?

$$\because \quad 6 \cdot 2^n + n^2 \le 7 \cdot 2^n \qquad \forall n \ge 5$$
$$(c = 7, n_0 = 5)$$
$$\therefore \quad 6 \cdot 2^n + n^2 = O(2^n)$$

Complexity classes

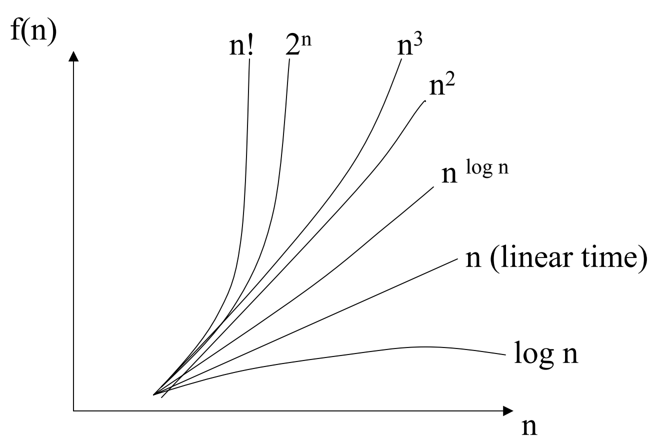


Figure 2. Growth rates of some important complexity classes

Assume that we have a machine that can execute 1,000,000 required operations per sec

	Algorithm 1	**Algorithm 2**	**Algorithm 3**	**Algorithm 4**	**Algorithm 5**
Frequency count	33n	6nlogn	$13n^2$	$3.4n^3$	2^n
n=10 **n=10,000**	< 1 sec < 1 sec	< 1 sec 6 sec	< 1 sec 22 min	< 1 sec 39 days	< 1 sec many many centuries

Table 1. Execution time for algorithms with the given time complexities

Note:

log n
n(linear)
n log n
n^2
n^3

polynomial time (easy or tractable)

2n
n!

exponential time (hard or intractable)

How do we know a given problem is easy?

Give a polynomial time algorithm for the problem

- ♥ Need to be good in
  design of algorithms (CS530)
- ♥ Need to be good in
  analysis of algorithms (CS530)

How do we know a given problem is hard?

Show that it is as hard as those well-known "hard" problems.

- ♥ Need help from the
  theory of NP Completeness (CS530)

Whether your program is the best or not?
(Comparing algorithms)

<u>The problem</u>: Counting the number if 1's in a n-bit string
algorithms:

1. $c \leftarrow 0$;
   for $i \leftarrow 1$ to n do
   if $b_i = 1$ then $c \leftarrow c+1$;

   Complexity: $T(n) = O(n)$

2. Assume $B = b_n b_{n-1} \ldots b_2 b_1$
c ← 0;
while B ≠ 0 do
  c ← c+1;
  B ← B ∧ (B –1); (Note that ∧ is "logical and")

Complexity: $T(n) = O(\#\text{ of 1's})$

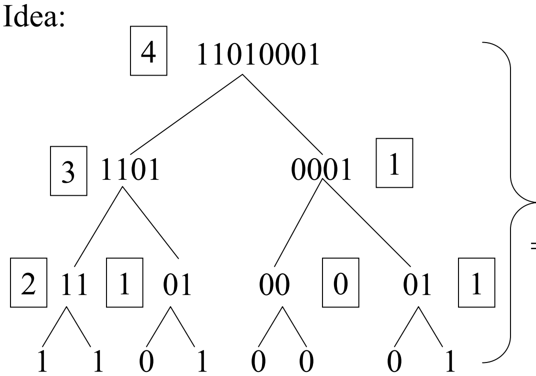
eg: B = 100011

c ← 1		100011	← B
	∧)	100010	← B-1
c ← 2		100010	← B'
	∧)	100001	← B'-1
c ← 3		100000	← B''
	∧)	011111	← B''-1
		000000	← B'''

3. Can we do it in *log n* time?
B = 11010001

	b_8	b_7	b_6	b_5	b_4	b_3	b_2	b_1
B	1	1	0	1	0	0	0	1
Odd		1		1		0		1
Shift even to right		1		0		0		0
$\mathbf{B^1 = B_{odd} + B_{even}}$		10		01		00		01
Odd				01				01
Shift even to right				10				00
$\mathbf{B^2 = B_{odd} + B_{even}}$				0011				0001
Odd								0001
Shift even to right								0011
$\mathbf{B^3 = B_{odd} + B_{even}}$								0100

Idea:



==> $T(n) = O(\log n)$

Note:

- The familiar machine that we used can only execute one operation in each step $\Rightarrow$ sequential machine.
- This stated algorithm requires the machine to have more than one operation done in each step $\Rightarrow$ parallel machine

Ω Notation

Def $f(n) = \Omega(g(n))$ iff $\exists$ two positive constants c and n_0, such that

$$|f(n)| \geq c \cdot |g(n)| \qquad \forall n \geq n_0$$

So, g(n) is the lower bound of f(n).

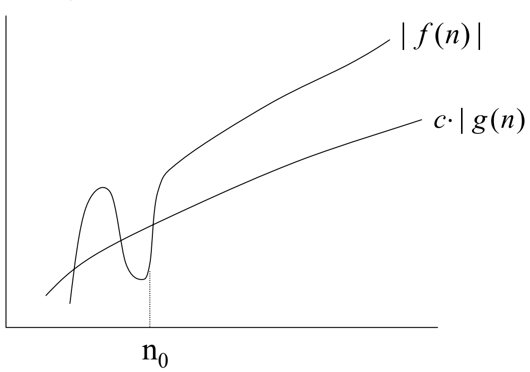


Figure 3. Illustrating Ω

Examples:

•Is $3n+2=\Omega(n)$?

$$\because \quad 3n+2 \geq 3 \cdot n \quad \forall n \geq 1$$
$$(c=3, n_0=1)$$
$$\therefore \quad 3n+2=\Omega(n)$$

•Is $3n+2=\Omega(1)$?

$$\because \quad 3n+2 \geq 1 \cdot 1 \quad \forall n \geq 1$$
$$(c=1, n_0=1)$$
$$\therefore \quad 3n+2=\Omega(1)$$

value of n_0

•Is $6 \cdot 2^n + n^2 = \Omega(n^{100})$?

$$\because \quad 6 \cdot 2^n + n^2 \geq ?\, n^{100} \quad \forall n \geq ?$$

When n is bigger, 2^n will grow faster than n^{100}. (Yes, you can find n_0)

$$\therefore \quad 6 \cdot 2^n + n^2 = \Omega(n^{100})$$

Θ notation

Def $f(n)=\Theta(g(n))$ iff $\exists$ three positive constants, c_1, c_2, n_0 ,

such that $c_1 \cdot |g(n)| \leq |f(n)| \leq c_2 \cdot |g(n)| \quad \forall n \geq n_0$.

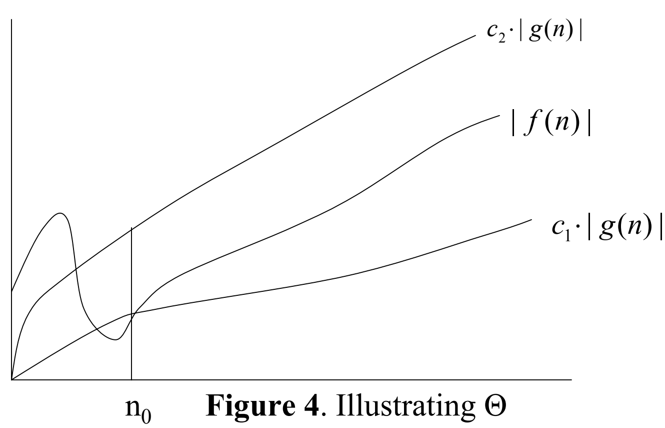


Figure 4. Illustrating Θ

Examples:

- Is $3n + 2 = \Theta(n)$?

$$\because \quad 3n \le 3n+2 \le 4n \qquad \forall n \ge 2$$
$$(c_1 = 3, c_2 = 4, n_0 = 2)$$
$$\therefore \quad 3n+2 = \Theta(n)$$

- Is $3n + 2 = \Theta(n^2)$?

$$\because \quad 3n+2 \ne \Omega(n^2)$$
$$\therefore \quad 3n+2 \ne \Theta(n^2)$$

- Is $6 \cdot 2^n + n^2 = \Theta(2^n)$?

$$\because \quad 6 \cdot 2^n \le 6 \cdot 2^n + n^2 \le 7 \cdot 2^n \qquad \forall n \ge 4$$
$$(c_1 = 6, c_2 = 7, n_0 = 4)$$
$$\therefore \quad 6 \cdot 2^n + n^2 = \Theta(2^n)$$

- Is $4n^3 + 3n^2 = \Theta(n^2)$?

$$\because \quad 4n^3 + 3n^2 \ne O(n^2)$$
$$\therefore \quad 4n^3 + 3n^2 \ne \Theta(n^2)$$

Property of Order

1) Transitive:

If $f(n) = O(g(n))$ and $g(n) = O(h(n))$ then $f(n) = O(h(n))$

If $f(n) = \Omega(g(n))$ and $g(n) = \Omega(h(n))$ then $f(n) = \Omega(h(n))$

If $f(n) = \Theta(g(n))$ and $g(n) = \Theta(h(n))$ then $f(n) = \Theta(h(n))$

2) Reflexive:

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

3) Symmetric:

$$f(n) = \Theta(g(n)) \text{ iff } g(n) = \Theta(f(n))$$

$$f(n) = O(g(n)) \text{ iff } g(n) = \Omega(f(n))$$

Analysis of Algorithms

a) Best Case, Worse Case Analysis

The problem:
Sort n keys in nondecreasing sequence.

Solving strategy:
Insertion sort – insert the i^{th} item into a sorted list of length $(i-1)$ by looking at numbers one at a time, $\forall\ i > 1$.

Example: sort 3,5,4,1

step 1	step 2	step 3	step 4
3	***3***	***3***	***1***
5	***5***	***4***	***3***
4	4	***5***	***4***
1	1	1	***5***

Algorithm:

```
A(0) ← -maxint;   // for efficient to stop the loop
for i ← 2 to n do
    { target ← A(i);
      j ← i –1;
      while (target < A(j))
          { A( j + 1) ← A(j);
            j ← j –1;
      A( j + 1) ← target;
```

Analysis:

Best Case performance is $\Theta(n)$ or $\Omega(n)$, but not $\Omega(n^2)$ since insertion sort runs in $\Theta(n)$ when the input is sorted.

Worst Case performance is $\Theta(n^2)$ or $O(n^2)$

The running time of insertion sort is between $\Omega(n)$ to $O(n^2)$

The running time is bound to $O(n^2)$

b) Average Case Analysis

The problem:

Is the key x in the array **S** of n keys?

Solving strategy:

Sequential search

Algorithm:

Procedure search (n, S, x, location) // n is total # of keys,
S is an array,
x is the key,
location is output parameter

```
begin
        location ← 1;
        while (location ≤ n) and (S(location) ≠ x)
                location ++;
        if location > n  location ← 0;
end;
```


Average Case Analysis:

Assume Probability *prob* $(x = k^{th}) = \frac{1}{n} \quad \forall 1 \le k \le n$

1) When x = k^{th}, # of key comparisons = k

$$\therefore \qquad A(n) = \frac{1}{n} \cdot 1 + \frac{1}{n} \cdot 2 + \cdots + \frac{1}{n} \cdot n = \frac{1}{n} \cdot (\sum_{k=1}^{n} k)$$

$$= \frac{1}{n} \cdot \frac{n(n+1)}{2} = \frac{n+1}{2}$$

about half array is searched.

2) When may not be in the array

Assume *prob* (x in array) = p, $\therefore\ prob(x = k^{th}) = p \cdot \frac{1}{n}$

prob (x not in array) = 1 – p, and it takes
n comparisons to know that is not in the array

$$\therefore \qquad A(n) = \frac{p}{n} \cdot \sum_{k=1}^{n} k + (1-p) \cdot n$$

$$= \frac{p}{n} \cdot \frac{n(n+1)}{2} + (1-p) \cdot n$$

$$= n \cdot (1 - \frac{p}{2}) + \frac{p}{2}$$

If $p = 1,\ A(n) = \frac{n+1}{2}$ (as calculated in case (1))

If $p = \frac{1}{2}$, $A(n) = \frac{3}{4} n + \frac{1}{4}$ (about ¾ of the array is searched)

Design of Algorithms

a) **Divide-and-Conquer**

b) **Greedy**

c) **Dynamic Programming**

d) **Backtracking & Branch-and-Bound**

Some mathematical background

Definition. $Log_b x = y$ if $b^y = x \quad b > 1$

1) Log_b is a strictly increasing function,
$\therefore$ if $x_1 < x_2$ then $Log_b x_1 < Log_b x_2$.

2) Log_b is a one-to-one function,
if $Log_b x_1 = Log_b x_2$ *then* $x_1 = x_2$.

3) $Log_b 1 = 0 \qquad \forall b$

4) $Log_b x^a = a \bullet Log_b x$

5) $Log_b(x_1 \bullet x_2) = Log_b x_1 + Log_b x_2$

6) $x_1^{Log_b x_2} = x_2^{Log_b x_1}$

7) to convert from one base to another, $Log_{b_1} x = \dfrac{Log_{b_2} x}{Log_{b_2} b_1}$

8) sum of consecutive integers $\sum_{i=1}^{n} i = \dfrac{n(n+1)}{2}$

9) geometric sums $\sum_{i=0}^{k} a^i = \dfrac{a^{k+1} - 1}{a - 1}$

10) Suppose f is a continuous, decreasing function, a, b are integers,

then $\int_{a}^{b+1} f(x)dx \le \sum_{i=a}^{b} f(i) \le \int_{a-1}^{b} f(x)dx$

11) Suppose f is a continuous, decreasing function, a, b are integers,

then $\int_{a-1}^{b} f(x)dx \le \sum_{i=a}^{b} f(i) \le \int_{a}^{b+1} f(x)dx$

12) $\int_{a}^{b} \frac{1}{x} dx = Log_e b - Log_e a$

Note:

$Log_2 = Lg$
$Log_e = Ln$