Introduction

Sep 20th, 2018

Algorithm Design and Analysis

YUN-NUNG (VIVIAN) CHEN HTTP://ADA.MIULAB.TW

國立臺灣大學 National Taiwan University

Slides credited from Prof. Hsueh-I Lu & Hsu-Chun Hsiao

Announcement

- Any question should be sent via email
  - Please use [ADA2018] in the subject
  - Please write your 學號 姓名 in the email
- Registration codes were sent out
  - Register (or drop?) the course ASAP
- Slides are available before the lecture starts
- Mini-HW 1 released
  - Due on 9/27 (Thu) 14:20
  - Submit to NTU COOL
- Judge system available
  - Programming part: submit to Online Judge – http://ada18-judge.csie.org

Mini-HW #1

Mini HW #1

Due Time: 2018/9/27 (Thu.) 14:20

Contact TAs: **ada-ta@csie.ntu.edu.tw**

Problem 1

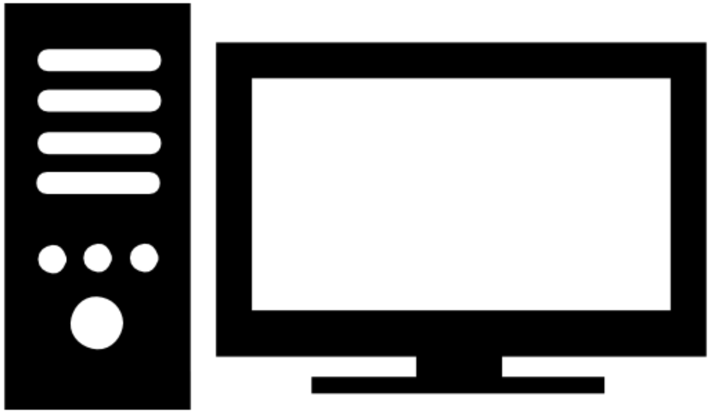
Let $f(n) = g(n) - h(n)$. Given $g(n) = \Theta(F(n))$ and $h(n) = o(F(n))$, prove or disprove $f(n) = \Omega(F(n))$. (Use the definitions of Θ, o and Ω given in textbook.)

Outline

- Terminology
  - Problem (問題)
  - Problem instance (個例)
  - Computation model (計算模型)
  - Algorithm (演算法)
  - The hardness of a problem (難度)
- Algorithm Design & Analysis Process
- Review: Asymptotic Analysis
- Algorithm Complexity
- Problem Complexity

Efficiency Measurement = Speed

- Why we care?
  - Computers may be fast, but they are not infinitely fast
  - Memory may be inexpensive, but it is not free



6 Terminology

Textbook Ch. 1 – The Role of Algorithms in Computing

Problem (問題)

The champion problem

- Input: n distinct integers $A[1], A[2], \ldots, A[n]$.
- Output: the index i with $1 \le i \le n$ such that

$$A[i] = \max_{1 \le j \le n} A[j].$$

Problem Instance (個例)

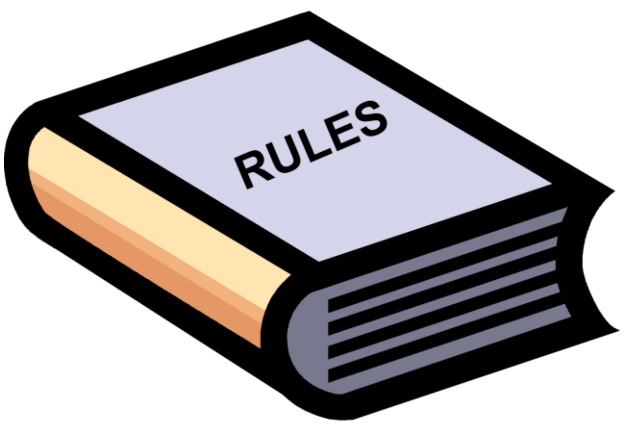
- An instance of the champion problem

$$5 \text{ distinct integers } 7, 4, 2, 9, 8.$$

7	4	2	9	8
A[1]	*A*[2]	*A*[3]	*A*[4]	*A*[5]

Computation Model (計算模型)

- Each problem must have its rule (遊戲規則)
- Computation model (計算模型) = rule (遊戲規則)
- The problems with different rules have different hardness levels



Hardness (難易程度)

- How difficult to solve a problem
  - Example: how hard is the champion problem?
  - Following the comparison-based rule

What does “solve (解)” mean?

What does “difficult (難)” mean?

Problem Solving (解題)

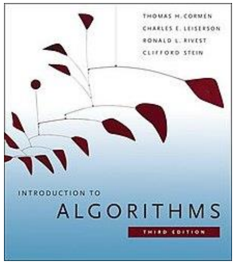
- Definition of "solving" a problem
  - Giving an algorithm (演算法) that produces a correct output for any instance of the problem.

Algorithm (演算法)

- Algorithm: a detailed step-by-step instruction
  - Must follow the game rules
  - Like a step-by-step recipe
  - Programming language doesn’t matter

  → problem-solving recipe (technology)
- If an algorithm produces a correct output for <u>any instance</u> of the problem

→ this algorithm “solves” the problem



“A well-defined computational procedure that transforms some input to some output”

Hardness (難度)

- Hardness of the problem
  - How much effort the best algorithm needs to solve any problem instance
- 防禦力
  - 看看最厲害的賽亞人要花多少攻擊力才能打贏對手

攻擊力 50000

防禦力 100000

14 Algorithm Design & Analysis Process

Algorithm Design & Analysis Process

1) Formulate a **problem**
2) Develop an **algorithm**
3) Prove the **correctness**
4) Analyze **running time/space** requirement

Design Step

Analysis Step

1. Problem Formulation

The champion problem

- Input: n distinct integers $A[1], A[2], \ldots, A[n]$.
- Output: the index i with $1 \leq i \leq n$ such that

$$A[i] = \max_{1 \leq j \leq n} A[j].$$

2. Algorithm Design

- Create a detailed recipe for solving the problem
  - Follow the comparison-based rule
    - 不准偷看信封的內容
    - 請別人幫忙「比大小」

Q1: Is this a comparison-based algorithm?

Q2: Does it solve the champion problem?

- Algorithm: 擂台法

```
int i, j;
j = 1;
for (i = 2;  i <= n;  i++)
    if (A[i] > A[j])
        j = i;
return j;
```

3. Correctness of the Algorithm

- Prove by contradiction (反證法)

The algorithm solves the champion problem.

Proof Let j^* be the correct answer. That is, $A[j^*] = \max\{A[1], \ldots, A[n]\}$.

- If $j^* = 1$, then Step 5 is never reached. Therefore, 1 is correctly returned.
- If $j^* > 1$, then in the iteration of the for-loop with $\boxed{i = j^*}$, j becomes j^*. By definition of j^*, $A[j^*] > A[i]$ holds for each $i = j^* + 1, \ldots, n$. Therefore, in the remaining iterations of the for-loop, the value of j does not change. Hence, at the end of the algorithm, j^* is correctly returned.

```
1.    int i, j;
2.    j = 1;
3.    for (i = 2;  i <= n;  i++)
4.        if (A[i] > A[j])
5.            j = i;
6.    return j;
```

Hardness of The Champion Problem

- How much effort the best algorithm needs to solve any problem instance
  - Follow the comparison-based rule
    - 不准偷看信封的內容
    - 請別人幫忙「比大小」
- Effort: we first use the <u>times of comparison</u> for measurement

```
int i, j;
j = 1;
for (i = 2;  i <= n;  i++)
    if (A[i] > A[j])
        j = i;
return j;
```

(n - 1) comparisons

Hardness of The Champion Problem

- The hardness of the champion problem is (n - 1) comparisons
  a) There is an algorithm that can solve the problem using at most (n – 1) comparisons
     - This can be proved by 擂臺法, which uses (n – 1) comparisons for any problem instance
  b) For any algorithm, there exists a problem instance that requires (n - 1) comparisons
     - Why?

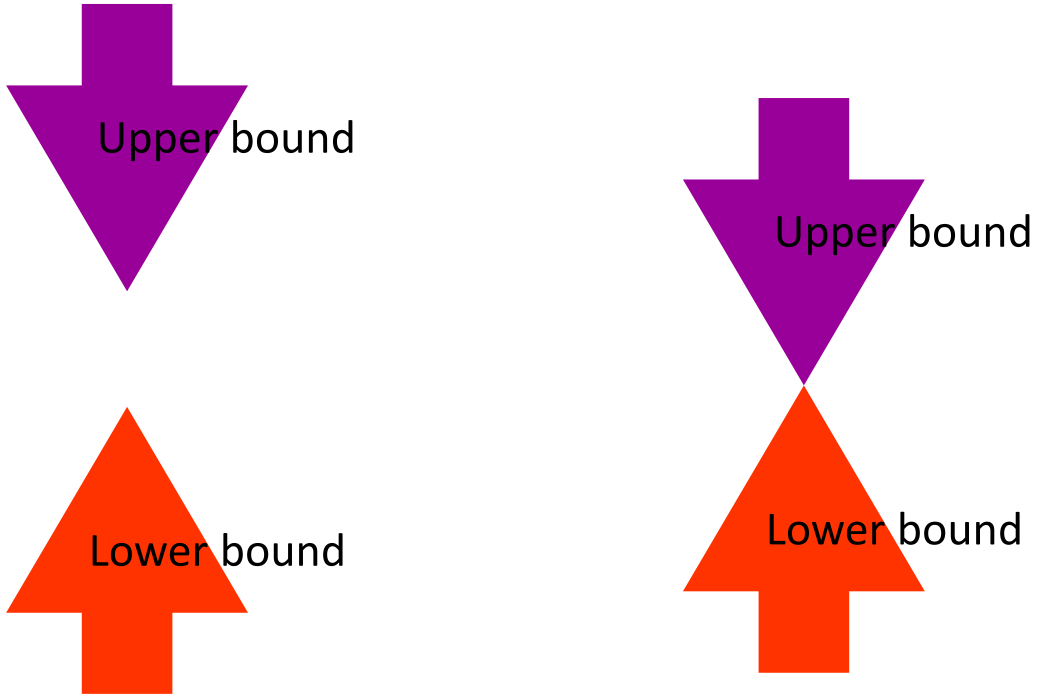
Hardness of The Champion Problem

- Q: Is there an algorithm that only needs (n - 2) comparisons?
- A: Impossible!
- Reason
  - A single comparison only decides a loser
  - If there are only (n - 2) comparisons, the most number of losers is (n - 2)
  - There exists a least 2 integers that did not lose

  → any algorithm cannot tell who the champion is

Finding Hardness

- Use the **upper bound** and the **lower bound**
- When they meet each other, we know the hardness of the problem



Hardness of The Champion Problem

- Upper bound
  - how many comparisons are sufficient to solve the champion problem
  - Each algorithm provides an upper bound
  - The smarter algorithm provides tighter, lower, and better upper bound

- Lower bound
  - how many comparisons in the worst case are necessary to solve the champion problem
  - Some arguments provide different lower bounds
  - Higher lower bound is better

多此一舉擂臺法

```
int i, j;
j = 1;
for (i = 2;  i <= n;  i++)
   if ((A[i] > A[j]) && (A[j] < A[i]))
         j = i;
return j;
```

→ ($2n$ - 2) comparisons

Every integer needs to be in the comparison once
→ (n/2) comparisons

When upper bound = lower bound, the problem is solved.
→ We figure out the hardness of the problem

4. Algorithm Analysis

- The majority of researchers in algorithms studies the time and space required for solving problems in two directions
  - Upper bounds: designing and analyzing algorithms
  - Lower bounds: providing arguments harder
- When the upper and lower bounds match, we have an optimal algorithm and the problem is completely resolved

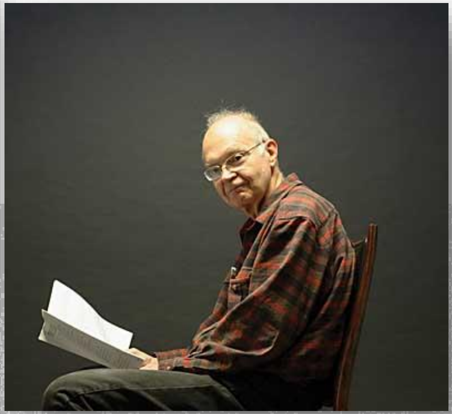
教過

我早就會了!

25 Asymptotic Analysis

O Ω Θ o ω

Edmund Landau
(1877-1938)



Donald E. Knuth
(1938-)

Motivation

- The hardness of the champion problem is exactly $n-1$ comparisons
- Different problems may have different 「難度量尺」
  - cannot be interchangeable
- Focus on the standard growth of the function to ignore the unit and coefficient effects

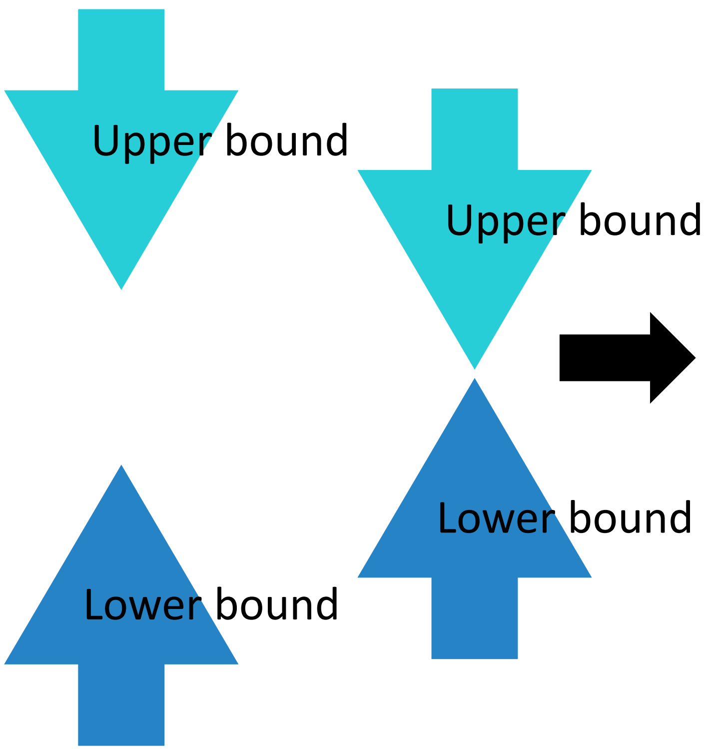
Goal: Finding Hardness

- For a problem *P*, we want to figure out
  - The hardness (complexity) of this problem *P* is $\Theta(f(n))$
    - n is the instance size of this problem *P*
    - $f(n)$ is a function
    - $\Theta(f(n))$ means that "it exactly equals to the growth of the function"
- Then we can argue that under the comparison-based computation model
  - The hardness of the champion problem is $\Theta(n)$
  - The hardness of the sorting problem is $\Theta(n \log n)$

Goal: Finding Hardness

- Use the upper bound and the lower bound
- When they match, we know the hardness of the problem

use $O\left(f(n)\right)$ and $o\left(f(n)\right)$



upper bound is $O\left(h(n)\right)$ & lower bound is $\Omega\left(h(n)\right)$
→ the problem complexity is exactly $\Theta\left(h(n)\right)$

use $\Omega\left(g(n)\right)$ and $\omega\left(g(n)\right)$

Goal: Finding Hardness

- First learn how to analyze / measure the effort an algorithm needs
  - Time complexity
  - Space complexity
- Focus on **worst-case** complexity
  - "average-case" analysis requires the assumption about the probability distribution of problem instances

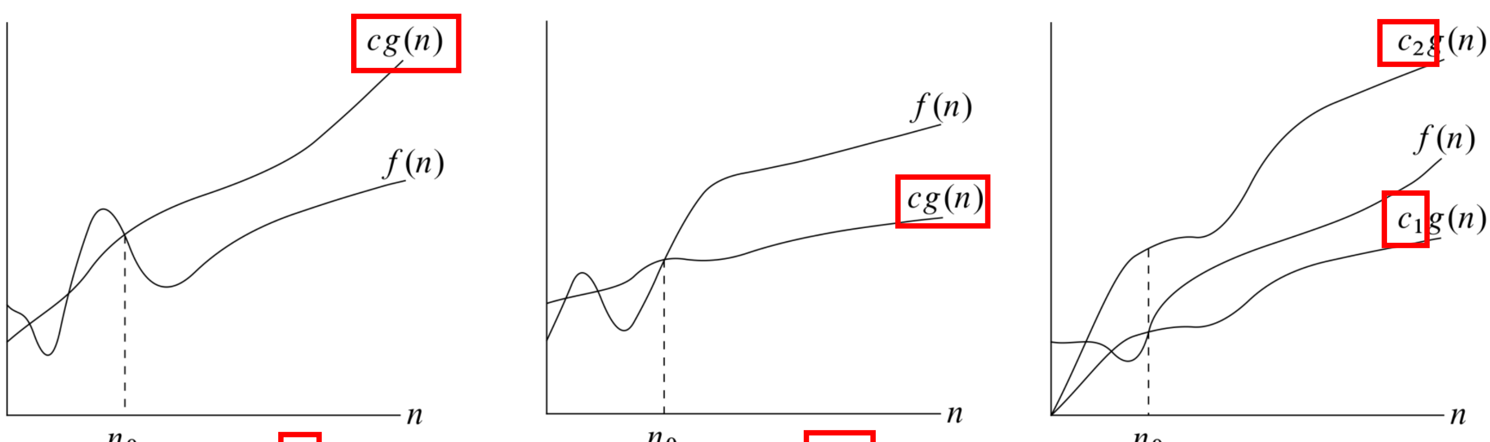
Types	Description
Worst Case	Maximum running time for **any instance** of size *n*
Average Case	**Expected** running time for a **random instance** of size *n*
Amortized	Worse-case running time for **a series of** operations

Review of Asymptotic Notation
(Textbook Ch. 3.1)

- $f(n)$ = time or space of an algorithm for an input of size n
- Asymptotic analysis: focus on the ***growth*** of $f(n)$ as $n \to \infty$

Review of Asymptotic Notation
(Textbook Ch. 3.1)

- $f(n)$ = time or space of an algorithm for an input of size n
- Asymptotic analysis: focus on the ***growth*** of $f(n)$ as $n \to \infty$
- O, or Big-Oh: **upper** bounding function
- Ω, or Big-Omega: **lower** bounding function
- Θ, or Big-Theta: **tightly** bounding function



$f(n) = \boxed{O}(g(n))$ $f(n) = \boxed{\Omega}(g(n))$ $f(n) = \Theta(g(n))$

Formal Definition of Big-Oh
(Textbook Ch. 3.1)

- For any two functions $f(n)$ and $g(n)$,

$$\boxed{f(n) = O(g(n))}$$

if there exist positive constants n_0 and c s.t.

$$0 \le f(n) \le \boxed{c \cdot g(n)}$$

for all $n \ge n_0$.

$g(n)$的某個常數倍$c \cdot g(n)$可以在n夠大時壓得住$f(n)$

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

- Intuitive interpretation
  - $f(n)$ does not grow faster than $g(n)$
- Comments
  1) $f(n) = O\big(g(n)\big)$ roughly means $f(n) \leq g(n)$ in terms of rate of growth
  2) "=" is not "equality", it is like "**ϵ (belong to)**"
     The equality is $\{f(n)\} \subseteq O\big(g(n)\big)$
  3) We do not write $O\big(g(n)\big) = f(n)$
- Note
  - $f(n)$ and $g(n)$ can be negative for some integers n
  - In order to compare using asymptotic notation O, both have to be <u>non-negative</u> for sufficiently large n
  - This requirement holds for other notations, i.e. $\Omega, \Theta, o, \omega$

Review of Asymptotic Notation
(Textbook Ch. 3.1)

- Benefit
  - Ignore the low-order terms, units, and coefficients
  - Simplify the analysis
- Example: $f(n) = 5n^3 + 7n^2 - 8$
  - Upper bound: $f(n) = O(n^3)$, $f(n) = O(n^4)$, $f(n) = O(n^3 \log_2 n)$
  - Lower bound: $f(n) = \Omega(n^3)$, $f(n) = \Omega(n^2)$, $f(n) = \Omega(n \log_2 n)$
  - $\boxed{\text{Tight bound: } f(n) = \Theta(n^3)}$

"=" doesn't mean "equal to"

- Q: $f(n) = O(n^3)$ and $f(n) = O(n^4)$, so $O(n^3) = O(n^4)$?
  - $O(n^3)$ represents **a set of functions** that are upper bounded by cn^3 for some constant *c* when *n* is large enough

O(n^4) = O(n^3) weird

  - In asymptotic analysis, "=" means "**ϵ (belong to)**"

Exercise: $100n^2 = O(n^3 - n^2)$?

- Draft.

$$100n^2 \leq \boxed{100}(n^3 - n^2)$$

$$\leftarrow 200n^2 \leq 100n^3$$

$$\leftarrow \boxed{2} \leq n$$

- Let $n_0 = 2$ and $c = 100$

$$100n^2 \leq 100(n^3 - n^2)$$

holds for $n \geq 2$

$$100n^2 = O(n^3 - n^2)$$

Exercise: $n^2 = O(n)$?

- Disproof.
  - Assume for a contradiction that there exist positive constants c and n_0 s.t.

$$n^2 \leq cn$$

  holds for any integer n with $n \geq n_0$.

  - Assume

$$n = 1 + \lceil \max(n_0, c) \rceil$$

  and because $n > n_0, n > c$, it follows that

$$n^2 > cn$$

  - Due to contradiction, we know that

$$n^2 \neq O(n)$$

Rules
(Textbook Ch. 3.1)

The following statements hold for any real-valued functions $f(n)$ and $g(n)$, where there is a constant n_0 such that $f(n)$ and $g(n)$ are nonnegative for any integer $n \geq n_0$.

- Rule 1: $f(n) = O(f(n))$.
- Rule 2: If c is a positive constant, then $c \cdot O(f(n)) = O(f(n))$.
- Rule 3: If $f(n) = O(g(n))$, then $O(f(n)) = O(g(n))$.
- Rule 4: $O(f(n)) \cdot O(g(n)) = O(f(n) \cdot g(n))$.
- Rule 5: $O(f(n) \cdot g(n)) = f(n) \cdot O(g(n))$.

Other Notations
(Textbook Ch. 3.1)

$$f(n) = O(g(n)) \rightarrow f(n) \le g(n) \text{ in rate of growth}$$
$$f(n) = \Omega(g(n)) \rightarrow f(n) \ge g(n) \text{ in rate of growth}$$
$$f(n) = \Theta(g(n)) \rightarrow f(n) = g(n) \text{ in rate of growth}$$
$$f(n) = o(g(n)) \rightarrow f(n) < g(n) \text{ in rate of growth}$$
$$f(n) = \omega(g(n)) \rightarrow f(n) > g(n) \text{ in rate of growth}$$

Goal: Finding Hardness

- First learn how to analyze / measure the effort an algorithm needs
  - Time complexity
  - Space complexity
- Focus on **worst-case** complexity
  - "average-case" analysis requires the assumption about the probability distribution of problem instances

Using O to give upper bounds on the worst-case time complexity of algorithms

Algorithm Analysis

- 擂台法

```
int i, j;                      O(1) time
j = 1;                         O(1) time
for (i = 2;  i <= n;  i++)     O(n) iterations
    if (A[i] > A[j])           O(1) time
        j = i;                 O(1) time
return j;                      O(1) time
```

Adding everything together
→ an upper bound on the
worst-case time complexity

- The worst-case time complexity is

$$O(1) + O(1) + O(n) \cdot (O(1) + O(1)) + O(1)$$

$$3 \cdot O(1) + O(n) \cdot (2O(1))$$

$$= O(1) + O(n) \cdot O(1) \quad \text{Rule 2}$$

$$= O(1) + O(n) \quad \text{Rule 4}$$

$$= O(n) + O(n) \quad 1 = O(n) \text{ \& Rule 3}$$

$$= 2 \cdot O(n)$$

$$= O(n) \quad \text{Rule 2}$$

Sorting Problem

- Input:

  An array A of n distinct integers.

- Output:

  Reorder A such that $A[1] < A[2] < \cdots < A[n]$.

Algorithm Analysis

- Bubble-Sort Algorithm

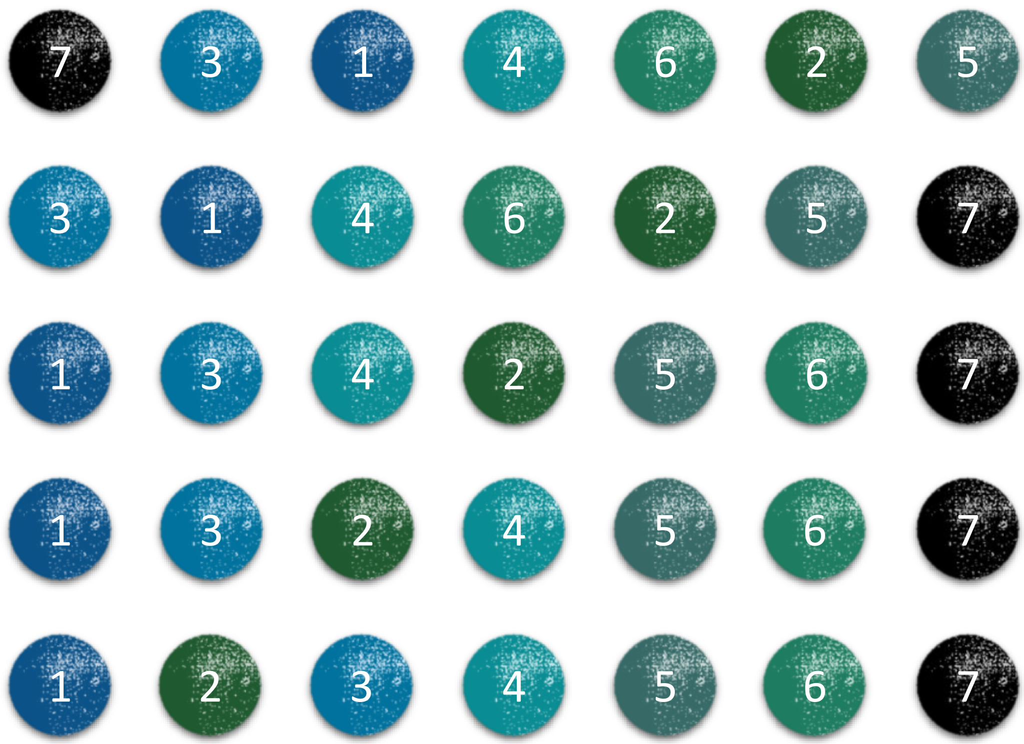
int *i*, *done*; flag	$O(1)$ time
do {	$f(n)$ iterations
done = 1;	$O(1)$ time
for (*i* = 1; *i* < *n*; *i* ++) {	$O(n)$ iterations
if (*A*[*i*] > *A*[*i* + 1]) {	$O(1)$ time
exchange *A*[*i*] and *A*[*i* + 1];	$O(1)$ time
done = 0;	$O(1)$ time
}	
}	
} while (*done* == 0)	

$$
\begin{aligned}
& O(1) + f(n) \cdot (O(1) + O(n) \cdot O(1)) \\
= & O(1) + f(n) \cdot O(n) \\
= & f(n) \cdot O(n) \\
= & O(n^2)
\end{aligned}
$$

$f(n) = O(n)$

prove by induction

Example Illustration



Goal: Finding Hardness

- First learn how to analyze / measure the effort an algorithm needs
  - Time complexity
  - Space complexity
- Focus on **worst-case** complexity
  - "average-case" analysis requires the assumption about the probability distribution of problem instances

Using O to give **upper bounds** on the worst-case time complexity of algorithms

Using Ω to give **lower bounds** on the worst-case time complexity of algorithms

Algorithm Analysis

- 擂台法

int *i*;	$\Omega(1)$ time
int *m* = *A*[1];	$\Omega(1)$ time
for (*i* = 2; *i* <= *n*; *i* ++) {	$\Omega(n)$ iterations
if (*A*[*i*] > *m*)	$\Omega(1)$ time
m = *A*[*i*];	$\Omega(1)$ time
}	
return *m*;	$\Omega(1)$ time

$$
\begin{aligned}
& 3 \cdot \Omega(1) + \Omega(n) \cdot (2 \cdot \Omega(1)) \\
=& \Omega(1) + \Omega(n) \cdot \Omega(1) \\
=& \Omega(1) + \Omega(n) \\
=& \Omega(n)
\end{aligned}
$$

Adding everything together
→ a lower bound on the worst-case time complexity?

Algorithm Analysis

- 百般無聊擂台法

int *i*;	$\Omega(1)$ time
int *m* = *A*[1];	$\Omega(1)$ time
for (*i* = 2; *i* <= *n*; *i* ++) {	$\Omega(n)$ iterations
if (*A*[*i*] > *m*)	$\Omega(1)$ time
m = *A*[*i*];	$\Omega(1)$ time
if (*i* == *n*)	$\Omega(1)$ time
do *i*++ *n* times	$\Omega(n)$ time
}	
return *m*;	$\Omega(1)$ time

$$
\begin{aligned}
& 3 \cdot \Omega(1) + \Omega(n) \cdot (3 \cdot \Omega(1) + \Omega(n)) \\
=& \Omega(1) + \Omega(n) \cdot \Omega(n) \\
=& \Omega(1) + \Omega(n^2) \\
=& \Omega(n^2)
\end{aligned}
$$

Adding together may result in errors.
The safe way is to analyze using **problem instances**.

e.g. try *A*[*i*] = *i* or *A*[*i*]=2(*n* – *i*) to check the time complexity → $\Omega(1)$

Algorithm Analysis

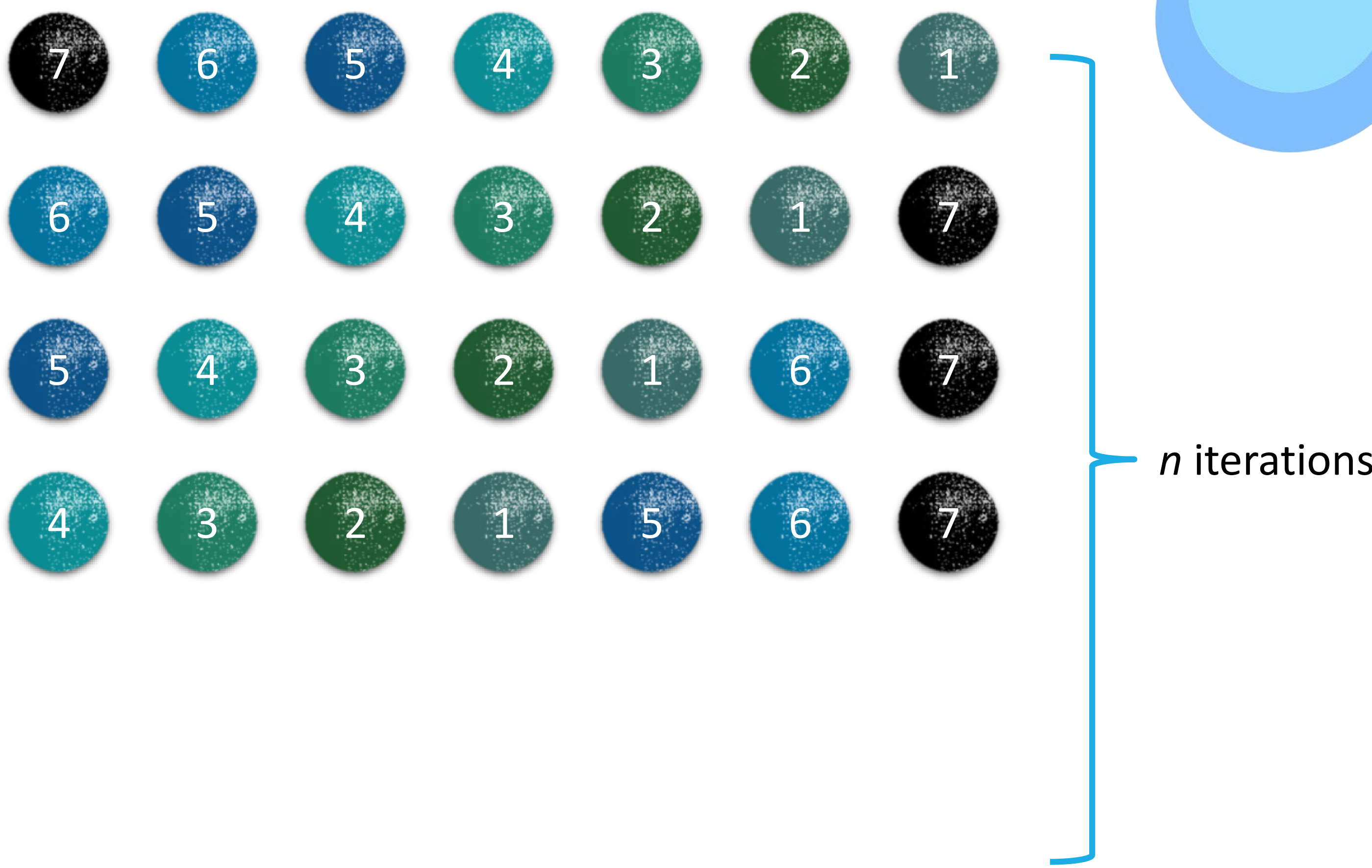
- Bubble-Sort Algorithm

```
int i, done;
do {                                f(n) iterations
    done = 1;
    for (i = 1; i < n; i ++) {      Ω(n) time
        if (A[i] > A[i + 1]) {
            exchange A[i] and A[i + 1];
            done = 0;
        }
    }
} while (done == 0)
```

When A is decreasing, $f(n) = \Omega(n)$. Therefore, the worst-case time complexity of Bubble-Sort is

$$f(n) \cdot \Omega(n) = \Omega(n^2)$$

Example Illustration



49 *Algorithm* Complexity

In the worst case, what is the growth of function an algorithm takes

Time Complexity of an Algorithm

- We say that the (worst-case) time complexity of Algorithm A is $\Theta\big(f(n)\big)$ if

1. Algorithm A runs in time $\boxed{O\big(f(n)\big)}$ &
2. Algorithm A runs in time $\boxed{\Omega\big(f(n)\big)}$ (in the worst case)
   - An input instance $I(n)$ s.t. Algorithm A runs in $\Omega\big(f(n)\big)$ for each n

Tightness of the Complexity

- If we say that the time complexity analysis about $O\big(f(n)\big)$ is tight 上界最低了
- = the algorithm runs in time $\Omega\big(f(n)\big)$ in the worst case
- = (worst-case) time complexity of the algorithm is $\Theta\big(f(n)\big)$
  - Not over-estimate the worst-case time complexity of the algorithm

- If we say that the time complexity analysis of Bubble-Sort algorithm about $O(n^2)$ is tight
- = Time complexity of Bubble-Sort algorithm is $\Omega(n^2)$
- = Time complexity of Bubble-Sort algorithm is $\Theta(n^2)$

Algorithm Analysis

- 百般無聊擂台法

1.	int *i*;	$O(1)$ time
2.	int *m* = *A*[1];	$O(1)$ time
3.	for (*i* = 2; *i* <= *n*; *i* ++) {	$O(n)$ iterations
4.	if (*A*[*i*] > *m*)	$O(1)$ time
5.	*m* = *A*[*i*];	$O(1)$ time
6.	if (*i* == *n*)	$O(1)$ time
7.	do *i*++ *n* times	$O(n)$ time
8.	}	
9.	return *m*;	$O(1)$ time

The worst-case time complexity of 「百般無聊擂臺法」 is $\Theta(n)$.

non-tight analysis

$$
\begin{aligned}
& 3 \cdot O(1) + O(n) \cdot (3 \cdot O(1) + O(n)) \\
= & O(1) + O(n) \cdot O(n) \\
= & O(1) + O(n^2) \\
= & O(n^2)
\end{aligned}
$$

tight analysis

Step 3 takes $O(n)$ iterations for the for-loop, where only last iteration takes $O(n)$ time and the rest take $O(1)$ time.
The steps 3-8 take time

$$O(n) \cdot O(1) + 1 \cdot O(n) = O(n)$$

The same analysis holds for $\Omega(n)$

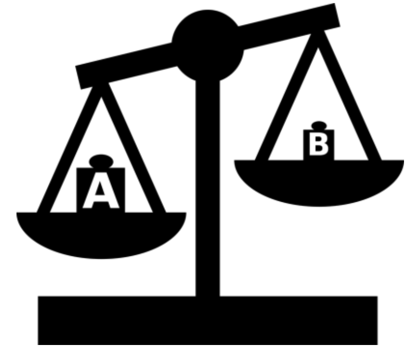
Algorithm Comparison

- Q: can we say that Algorithm 1 is a better algorithm than Algorithm 2 if
  - Algorithm 1 runs in $\boxed{O}(n)$ time
  - Algorithm 2 runs in $\boxed{O}(n^2)$ time

upper bound

- A: No! The algorithm with a lower upper bound on its worst-case time does not necessarily have a lower time complexity. 沒有說哪一個是tight

Comparing A and B



- Algorithm A is no worse than Algorithm B in terms of worst-case time complexity if there exists a positive function $f(n)$ s.t.
  - Algorithm A runs in time $O(f(n))$ &
  - Algorithm B runs in time $\Omega(f(n))$ in the worst case
- Algorithm A is (strictly) better than Algorithm B in terms of worst-case time complexity if there exists a positive function $f(n)$ s.t.
  - Algorithm A runs in time $O(f(n))$ & A<=n
  - Algorithm B runs in time $\omega(f(n))$ in the worst case B>n

  or

  - Algorithm A runs in time $o(f(n))$ &
  - Algorithm B runs in time $\Omega(f(n))$ in the worst case

55 *Problem* Complexity

In the worst case, what is the growth of the function the optimal algorithm of the problem takes

Time Complexity of a Problem

- We say that the (worst-case) time complexity of Problem *P* is $\Theta(f(n))$ if

1. The time complexity of Problem *P* is $O(f(n))$ &
   - There exists an $O(f(n))$-time algorithm that solves Problem *P*
2. The time complexity of Problem *P* is $\Omega(f(n))$
   - Any algorithm that solves Problem *P* requires $\Omega(f(n))$ time

- The time complexity of the champion problem is $\Theta(n)$ because

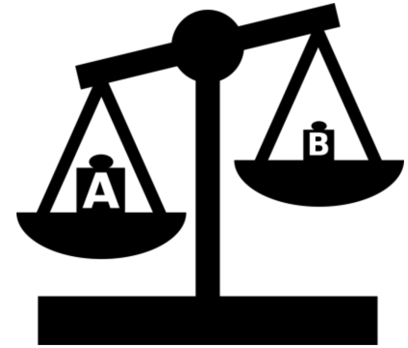
1. The time complexity of the champion problem is $O(n)$ &
   - 「擂臺法」is the $O(n)$-time algorithm
2. The time complexity of the champion problem is $\Omega(n)$
   - Any algorithm requires $\Omega(n)$ time to make each integer in comparison at least once

Optimal Algorithm

最好的

- If Algorithm *A* is an optimal algorithm for Problem *P* in terms of worst-case time complexity
  - Algorithm *A* runs in time $O\big(f(n)\big)$ &
  - The time complexity of Problem *P* is $\Omega\big(f(n)\big)$ in the worst case
- Examples (the champion problem)
  - 擂台法 → optimal algorithm
    - It runs in $O(n)$ time &
    - Any algorithm solving the problem requires time $\Omega(n)$ in the worst case
  - 百般無聊擂台法 → optimal algorithm
    - It runs in $O(n)$ time &
    - Any algorithm solving the problem requires time $\Omega(n)$ in the worst case

Comparing *P* and *Q*



- Problem *P* is no harder than Problem *Q* in terms of (worst-case) time complexity if there exists a function $f(n)$ s.t.
  - The (worst-case) time complexity of Problem *P* is $O\big(f(n)\big)$ & P<=n
  - The (worst-case) time complexity of Problem *Q* is $\Omega\big(f(n)\big)$ Q=>n
- Problem *P* is (strictly) easier than Problem *Q* in terms of (worst-case) time complexity if there exists a function $f(n)$ s.t.
  - The (worst-case) time complexity of Problem *P* is $O\big(f(n)\big)$ &
  - The (worst-case) time complexity of Problem *Q* is $\omega\big(f(n)\big)$

  or
  - The (worst-case) time complexity of Problem *P* is $o\big(f(n)\big)$ &
  - The (worst-case) time complexity of Problem *Q* is $\Omega\big(f(n)\big)$

Concluding Remarks

- Algorithm Design and Analysis Process
  1) Formulate a **problem**
  2) Develop an **algorithm**
  3) Prove the **correctness**
  4) Analyze **running time/space** requirement

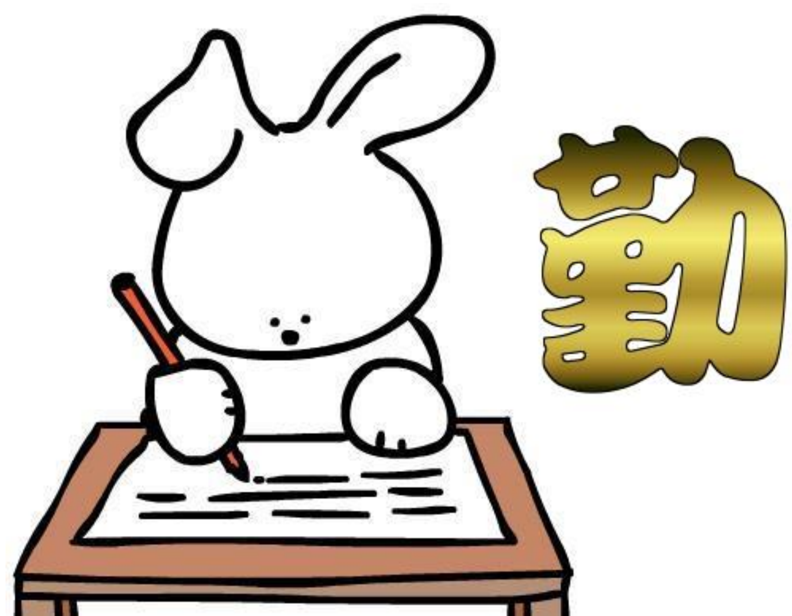
Design Step

Analysis Step

- Usually brute force (暴力法) is not very efficient
- Analysis Skills
  - Prove by contradiction
  - Induction
  - Asymptotic analysis
  - Problem instance
- Algorithm Complexity
  - In the worst case, what is the growth of function an algorithm takes
- Problem Complexity
  - In the worst case, what is the growth of the function the optimal algorithm of the problem takes

Reading Assignment

- Textbook Ch. 3 – Growth of Function



Question?

Important announcement will be sent to @ntu.edu.tw mailbox
& post to the course website

Course Website: http://ada.miulab.org

Email: ada-ta@csie.ntu.edu.tw