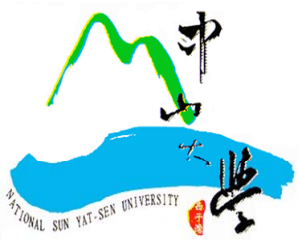
課程名稱：演算法設計與分析

Design and Analysis of Algorithms

1) 本課程計分方式
2) 演算法簡介
3) 本學期授課大綱

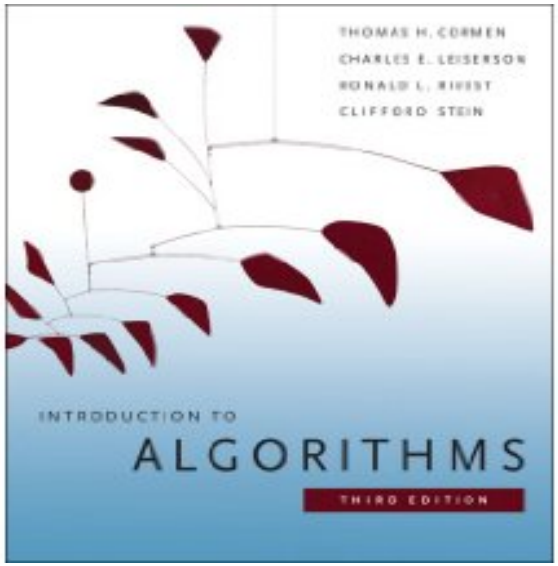


Instructor: 周孜燦 助理教授
國立中山大學電機系
Email: ztchou@ee.nsysu.edu.tw

投影片下載

- 投影片下載網址 **http://wmi.ee.nsysu.edu.tw**
- 投影片密碼 **ee@nsysu**
- 自己注意更新日期（如果有更新，幅度也是非常的小）

Textbook



Introduction to Algorithms

3rd edition, MIT Press, 2009

作者群

Thomas H. Cormen

Charles E. Leiserson

Ronald L. Rivest

Cliff Stein

2002 年電腦諾貝爾獎 Turing Award 得主，
RSA 公司創辦人 親自下海寫書，值得收藏喔！
代理：開發圖書，集體購買洽詢：甘經理 **0916709655**

授課與考試方式

老師約 **80%**（不會**100%**） 按課本的內容或次序來上課，畢竟 **1292** 頁真的太厚了！ 只能挑選重要的部分來教。

投影片就像講義一樣清楚。但如果同學覺得投影片寫的還不夠清楚，可自己對照課本內容來閱讀。除了第五章外，投影片所探討的主題全都可在課本或課本習題裡頭找到

考試題目 **100%** 出自投影片。爲了讓同學考試過關，同時也爲了要有鑑別率，投影片最後自我評量的部分佔 **70%**。如果你只準備自我評量的部分，可以過關，但無法高分。

課程計分方式

- 期中考：**50%**
- 期末考：**50%**（不包含期中考的範圍）

額外加分條件

平常上課不點名。如果老師發覺上課人數突然變少，就會故意點名。點名沒到無所謂；點到名一次，總成績 **+2** 分（相當於點到名一次，期末考就加 **+4** 分）

Prerequisite

基礎知識	我們假設同學修過程式設計的課，看得懂 C 語言 所寫的程式碼。如果至今仍然沒修過，請直接退選。
需自修的知識	完全沒修過機率和離散數學的同學，一定要自行閱讀課本 附錄B 和 附錄C 的部分，不然上課可能會聽不懂。
相輔相成的知識	如果修過資料結構，那是最棒。沒修過也沒關係，本課程的內容和資料結構有部分重疊。
額外好處	希望對 2012 年電機系碩士班丙組的入學考試有幫助
哪些同學不適合修這門課？	由於老師已經在考題上相當放水，所以在給分上沒有彈性。如果你是資工系四年級的同學，打算考非資訊相關研究所，抽不出時間閱讀，請直接退選。截至目前為止，已經有超過三位資工系同學考上研究所仍然被當

上課聽不懂怎麼辦？

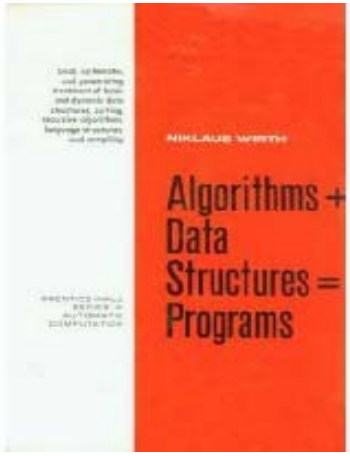
盡量上課的時候「立即」發問，因爲老師下課時間也要休息。

如果你超級害羞，那就 **email** 問我。我一定會回 **email** 給你。

What Is an Algorithm ?

課本的作者說：**An algorithm is a sequence of computational steps that transform the input into the output.**

問：這個跟程式（**program**）有何不同？



從書名 **algorithm + data structure = program**

我們知道演算法著重於解決問題的「策略與方法」，目的在於「有效率地」解決問題。

Input：整數 $n \geq 0$

Output：求出 0！、1！、…、n！

```
#include <stdio.h>
#define ArraySize 5   // 偷懶的寫法，代替 input
int fact(int n) {
if (n==0)
   return 1;
else
   return n*fact(n-1);
}
void main(void) {
int i, factArray[ArraySize];
for (i=0; i <= ArraySize - 1; i++)
   factArray[i] = fact(i);
}
```

採用「遞迴」的方法求出 n！

Name	Value
factArray	0x0012ff68
[0]	1
[1]	1
[2]	2
[3]	6
[4]	24

Watch1 Watch2 Watch

採用「陣列」的資料結構 儲存

Prologue

何謂「有效率地」解決問題？在體驗學習演算法（algorithm）的重要性之前，讓我們先考慮一個問題：

試求出 Fibonacci number $F(n)$

$$F(n) = \begin{cases} 1 & \text{if } n = 0,1 \\ F(n-1) + F(n-2) & \text{if } n \geq 2 \end{cases}$$

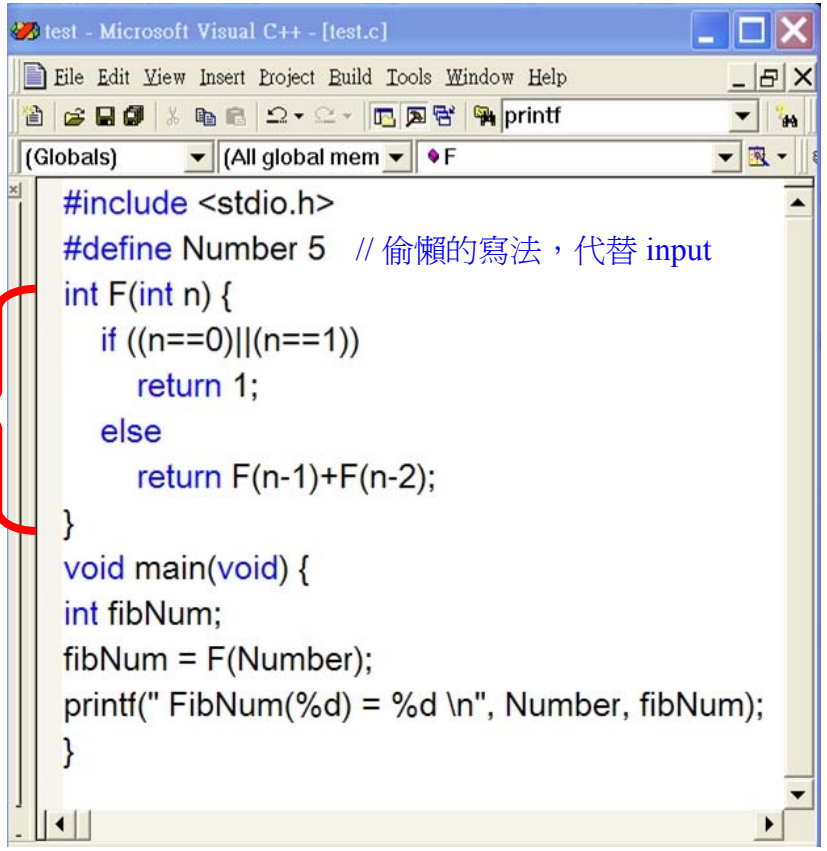
Fibonacci, 1170-1250

好像很喜歡打籃球喔？
要不然怎帶跟周杰倫一樣的帽子（冷笑話）

例如：$F(0) = 1, F(1) = 1, F(2) = 2, F(3) = 3, F(4) = 5, F(5) = 8$

Algorithm1: Recursive Approach

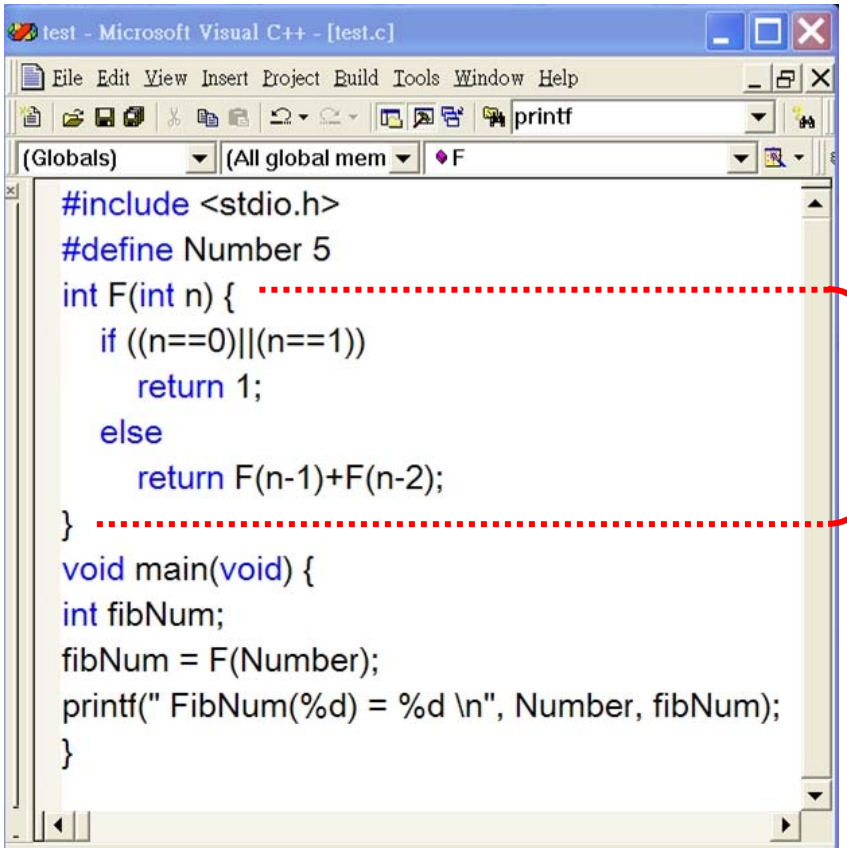
```
int F(int n) {
    if ((n==0)||(n==1))
        return 1;
    else
        return F(n-1)+F(n-2);
}
```

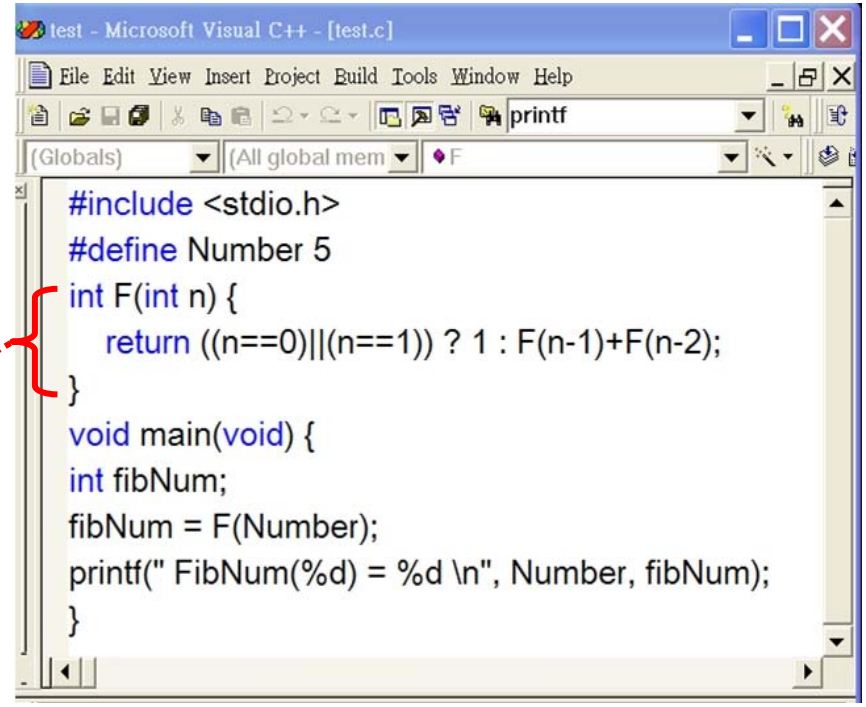


我們假設：給定 **pseudo code**，同學們有辦法自行寫出完整的程式碼。 註：如果做不到，那麼你不適合修這門課

求出 $F(n) = F(n-1) + F(n-2)$ 最直覺的方法便是採用 **recursive** 的方式。但這樣的演算法到底好不好？

Shorter Is Not Always Better





如右圖，剛學會程式設計的人喜歡用「奇特怪異」的指令讓程式碼變得很短，以爲最少指令的程式，便是最好的程式。事實上這是完全錯誤的觀念。因爲這些怪異的指令不但讓程式碼日後難以理解與維護，而且...

How to Judge the Goodness of an Algorithm ?

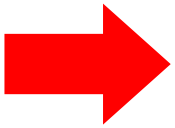
用「執行時間」來判斷演算法的好壞永遠是最可靠的辦法之一。

所以我們要來檢驗程式碼 **F(n)** 的執行時間爲多少？

假設每個「+、－、×、÷、比較」基本指令的執行時間相差不多。我們想問：要求出 **F(n)**，下方的程式碼總共要執行「多少次的指令」？

令 ***S(n)*** 表示求出 ***F(n)*** 所需執行的指令次數

```
int F(int n) {
   if ((n==0)||(n==1))  ..... 1 次
      return 1;
   else
      return F(n-1)+F(n-2); ..... S(n-1) + S(n-2) 次
}
```



$$S(n)=\begin{cases}1 & \text{if } n=0 \text{ or } n=1\\ 1+S(n-1)+S(n-2) & n\geq 2\end{cases}$$

Lower Bound of Running Time $S(n)$

現在，我們要證明 $S(n) > \left(1+\frac{1}{\sqrt{5}}\right)\left(\frac{1+\sqrt{5}}{2}\right)^{n-2} = 1.447 \times 1.618^{n-2}$

這意味著要求出 $F(100)$ ，至少需 $1.447 \times 1.618^{98} \approx 4.37 \times 10^{20}$ 次的計算

BASIS STEP

$S(2) = 3 > \left(1+\frac{1}{\sqrt{5}}\right)\left(\frac{1+\sqrt{5}}{2}\right)^{2-2} = 1.447$ 且 $S(3) = 5 > \left(1+\frac{1}{\sqrt{5}}\right)\left(\frac{1+\sqrt{5}}{2}\right)^{3-2} = 2.34$

INDUCTIVE STEP

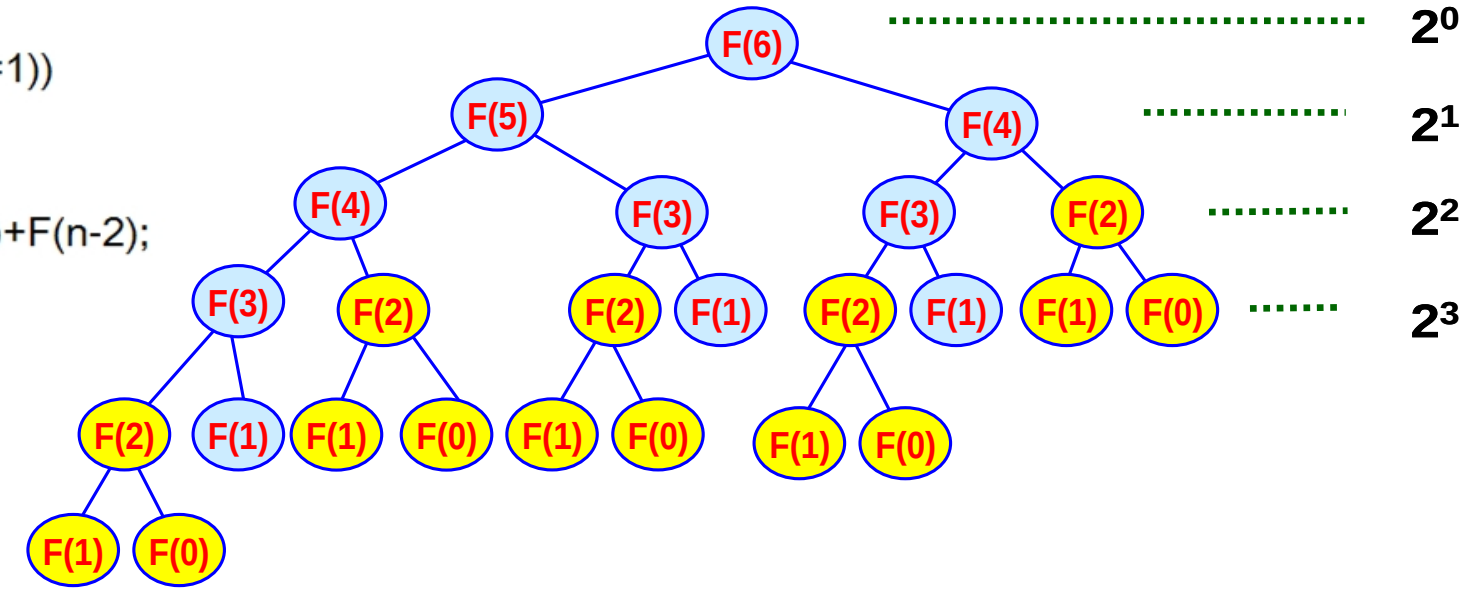
假設對於任意整數 k ， $S(k) > \left(1+\frac{1}{\sqrt{5}}\right)\left(\frac{1+\sqrt{5}}{2}\right)^{k-2}$ 都成立

那麼 $S(k+1) = S(k) + S(k-1) + 1 > \left(1+\frac{1}{\sqrt{5}}\right)\left(\frac{1+\sqrt{5}}{2}\right)^{k-2} + \left(1+\frac{1}{\sqrt{5}}\right)\left(\frac{1+\sqrt{5}}{2}\right)^{k-3} + 1$

$> \left(1+\frac{1}{\sqrt{5}}\right)\left(\frac{1+\sqrt{5}}{2}\right)^{k-2} + \left(1+\frac{1}{\sqrt{5}}\right)\left(\frac{1+\sqrt{5}}{2}\right)^{k-3} > \left(1+\frac{1}{\sqrt{5}}\right)\left(\frac{1+\sqrt{5}}{2}\right)^{k-1}$

Why Running Time Is So Large ?

```
int F(int n) {
    if ((n==0)||(n==1))
        return 1;
    else
        return F(n-1)+F(n-2);
}
```



我們簡單估計 $S(n)$ 的大小。

上圖的樹共有 $n = 6$ 個 level ，所以要計算 $F(n)$，程式至少需執行 $2^0 + 2^1 + 2^2 + \ldots + 2^{n-3} = 2^{n-2} - 1$ 次（扣掉最下面二層不規律的來看）

→ **$S(n) > 2^{n-2} - 1$**。程式執行時間（程式執行指令的次數）會這麼大的主要原因跟 **recursive** 無關，主要在於「有大量的重複計算」。

Algorithm2: Dynamic Programming Approach

註：這兒的 **programming** 乃二次世界大戰期間的術語，爲 **planning** 的意思（採用 **tabular method**）

```
int F(n) {
   F[0] = 1; F[1] = 1;
   for (i=2; i<= n; i++) {
         F[i] = F[i-1] + F[i-2];
   } // 共 n－1 個 loop
   return F[n];
}
```

要計算 $F(n)$，總共只需執行 $2 + (n - 1) = \boldsymbol{n + 1}$ 次。

如果 $n = 100$，總共只需執行 101 次。

想想看，如何不要使用 array。

提示：如右圖，每次都只使用到有顏色的部分，所以只需要三個變數即可取代 array

策略：將之前計算的結果存起來
避免重複計算

n	0	1					
F[n]	1	1					

n	0	1	2				
F[n]	1	1	2				

n	0	1	2	3			
F[n]	1	1	2	3			

n	0	1	2	3	4		
F[n]	1	1	2	3	5		

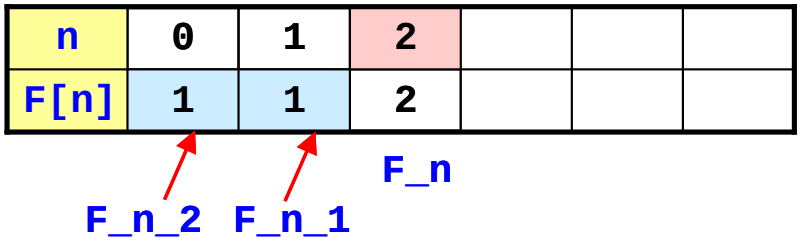
n	0	1	2	3	4	5	
F[n]	1	1	2	3	5	8	

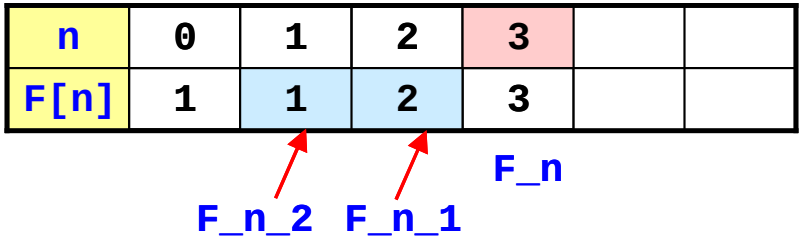
n	0	1	2	3	4	5	6
F[n]	1	1	2	3	5	8	13

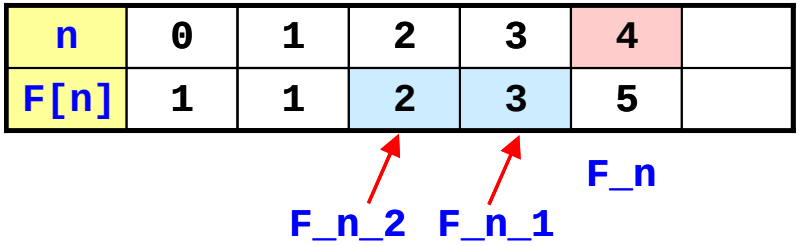
Dynamic Programming Without Using Array

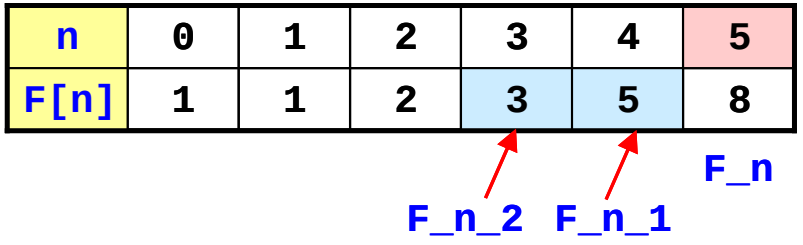
```
int Fib(int n)  {
    int i, F_n;
    int F_n_2=1, F_n_1=1;

    if ((n==0)||(n==1))
      return 1;
    for (i=2; i<=n; i++)
    {
      F_n = F_n_1 + F_n_2;
      F_n_2 = F_n_1;
      F_n_1 = F_n;
    }
    return F_n;
}
```





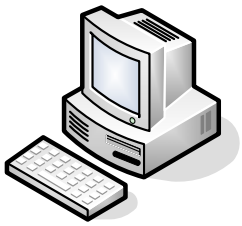




Algorithms as a Technology

F(100)

Algorithm1
(dynamic programming)



個人電腦，每秒只能執行 10^5 個指令。因爲存取 array 的關係，每個指令的執行時間約爲執行一個組合語言指令時間的 10 倍
→ 所以計算 *F*(100) 約需 $101\times10/10^5$ 秒～**10^{-2} 秒**。

F(100)

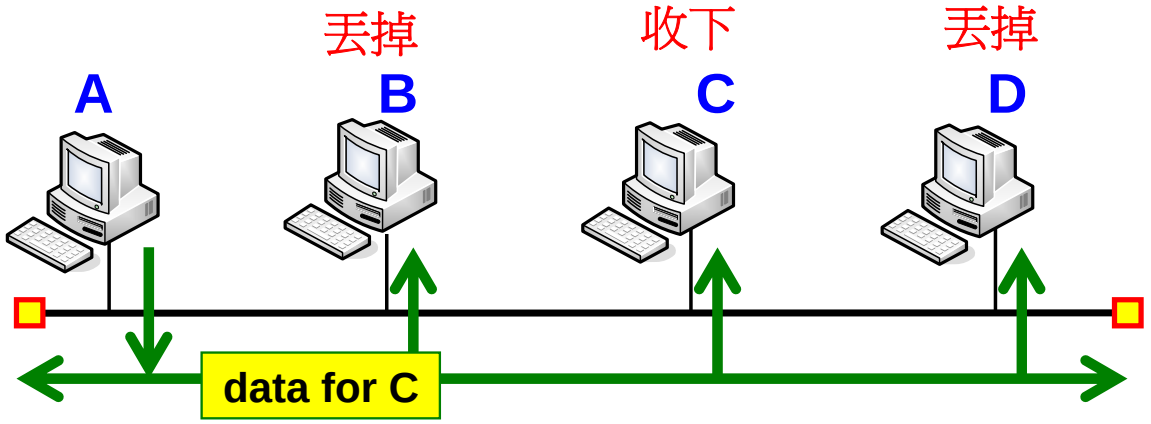
Algorithm2
(recursive)

超級電腦，每秒能執行 10^{12} 個指令，速度是個人電腦的 100 萬倍快。且用組合語言撰寫。那麼計算 *F*(100) 約需 $4.37\times10^{20}/10^{12}$ 秒～**13.86 年**。

很差的演算法，給你超級電腦也沒用
→ 演算法可以視爲是一種高科技，可爲人類帶來重大的貢獻

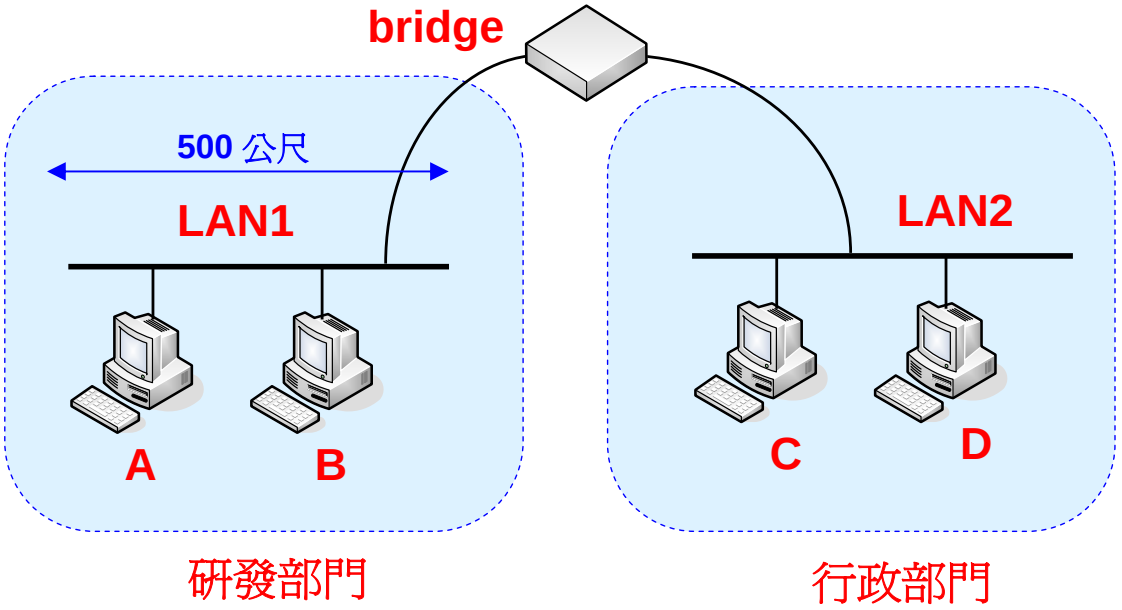
The Practical Value of Algorithms

雖然演算法大多考慮簡化後的問題，看起來很理論，但其實它的實用價值很高。讓我們來看一個和 **Ethernet bridge** 有關的問題。



目前最多人使用的區域網路（LAN，local area network）爲 Ethernet。Ethernet 的特性是：當 LAN 上的任何一台電腦（比如 A 要送資料給 C）傳送資料時，LAN 上的所有成員都會收到。除了 C 之外，其餘的電腦收到資料之後，發覺 destination 不是自己，便會把資料丟掉。

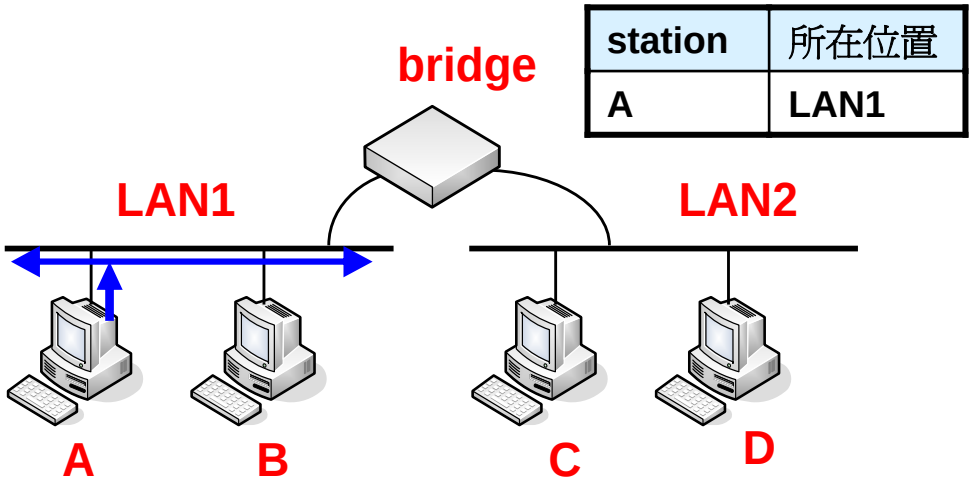
The Necessity of Bridge



自從 Ethernet 發明之後，很多公司的部門都開始架設 Ethernet，讓部門內的同仁彼此能透過電腦互相交換資料。然而每個 Ethernet segment 的最大長度只有 500 公尺，這讓部門之間的電腦資料交換變得很麻煩。

1983 年，DEC 的經理 Mark Kempf 發明了**橋接器（bridge）**，用來連接不同的 LANs（如 LAN1 和 LAN2）並且 LAN 上面的每台電腦都不用重新設定（意即 **plug-and-play**）

Bridge's Learning Algorithm

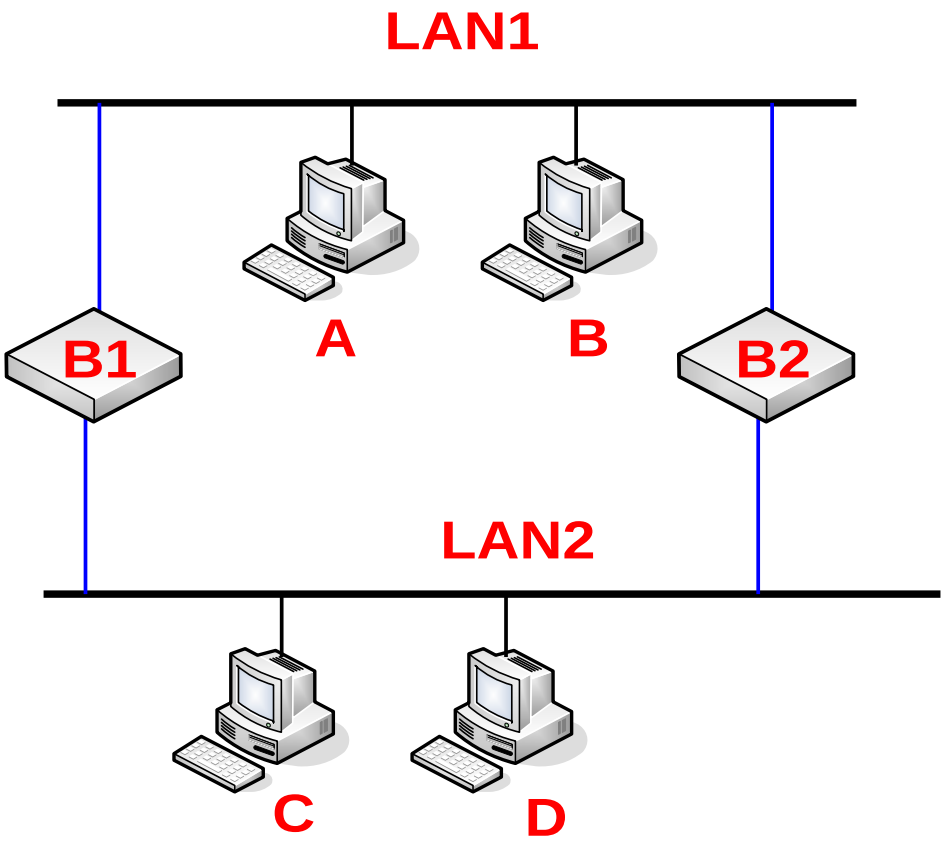


station	所在位置
A	LAN1

Bridge（美國專利號碼：4,597,078）的運作如下：

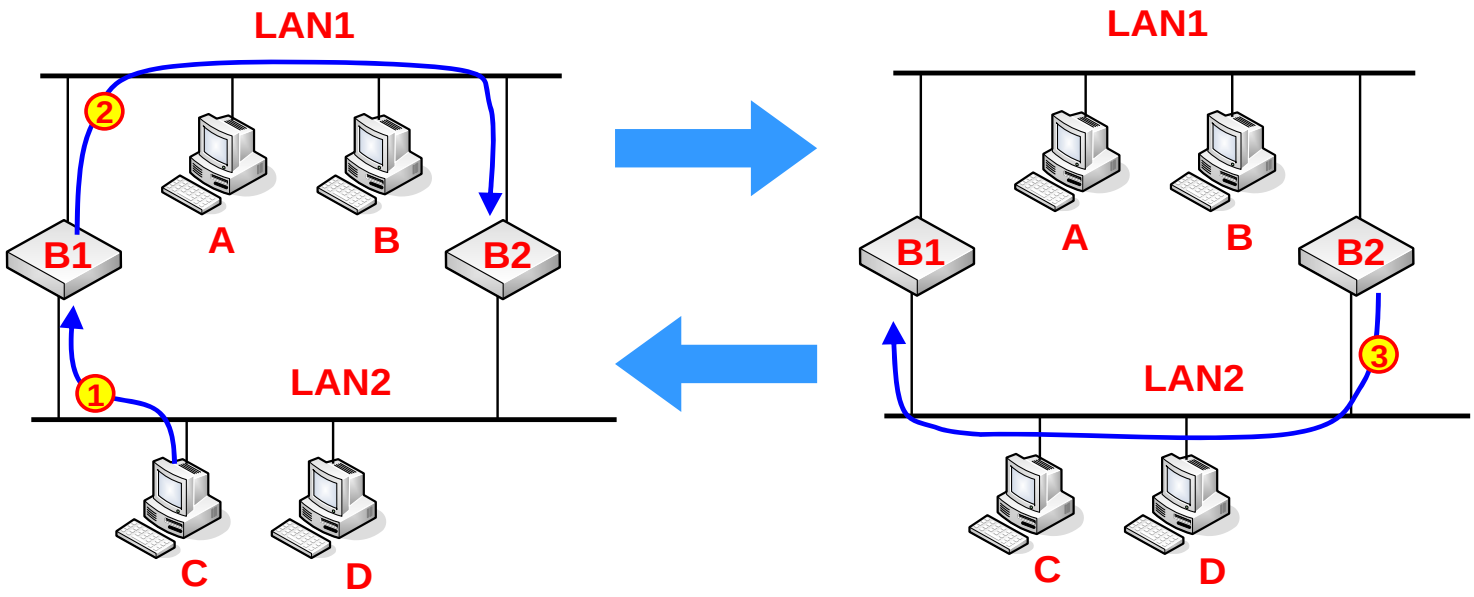
- Bridge 裡頭 keep 一份 learning cache，一開始裡頭都是空的。
- 如果 A 想送資料給 B，由於 bridge 不知道 B 在哪兒，所以會將這份資料轉送到所有和它相連的 LAN（此例爲 LAN 2）。但 bridge 知道 A 的訊號是由 LAN 1 發出，所以會在 learning cache 裡頭記錄 A 的位置是在 LAN 1。以後如果 B 要送資料給 A，bridge 會把收到的資料丟掉，不會轉送到 LAN 2。而若 C 要送資料給 A，bridge 會將資料轉送到 LAN 1。

Adding Redundant Bridges for Reliability



爲了避免一旦 bridge 壞掉，LAN 1 和 LAN 2 之間的電腦就無法溝通，公司可能會買了二台 bridges 連接 LAN 1 和 LAN 2，以增加 reliability，以確保即使其中一台 bridge 壞掉，網路還是處於 connected 狀態。

Broadcast Strom Problem

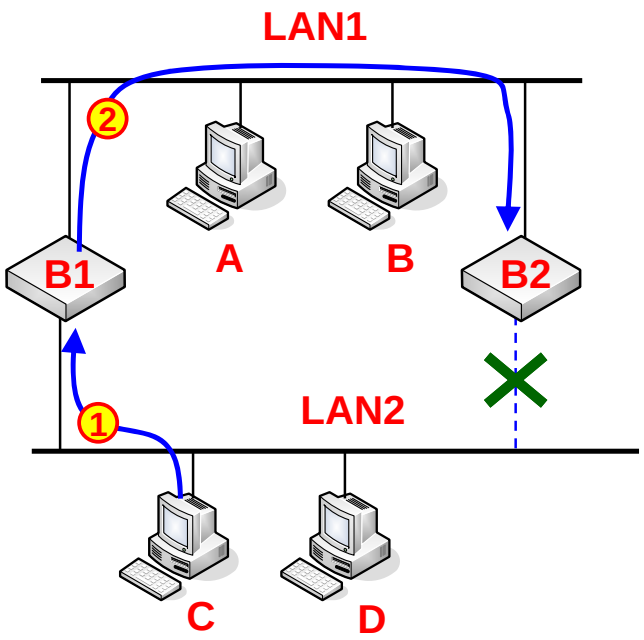


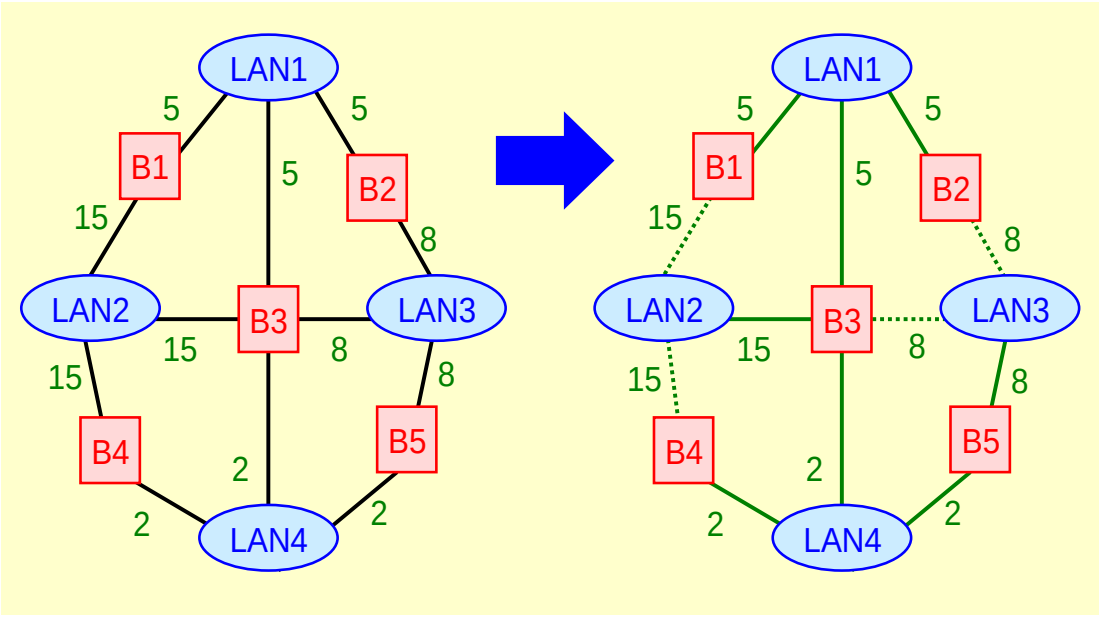
(1) 如左圖，當 C 想傳資料給 B，當 bridge B1 的 learning cache 裡頭沒有 B 的資料時，B1 便會將這份資料轉送到 LAN 1。

(2) 當 B2 透過 LAN 1 收到這份資料時，由於 B2 也不知道 B 在哪兒，所以會將這份資料轉送到 LAN 2（由於 B 從未發言，所有 bridges 都不知道 B 位在哪個 LAN）

(3) 如此一來，圖中 ② 和 ③ 將週而復始的出現，永遠停不下來。這個問題被稱之爲 broadcast storm problem。其根本原因在於網路裡頭存在有 loop

Minimum Cost Spanning Tree





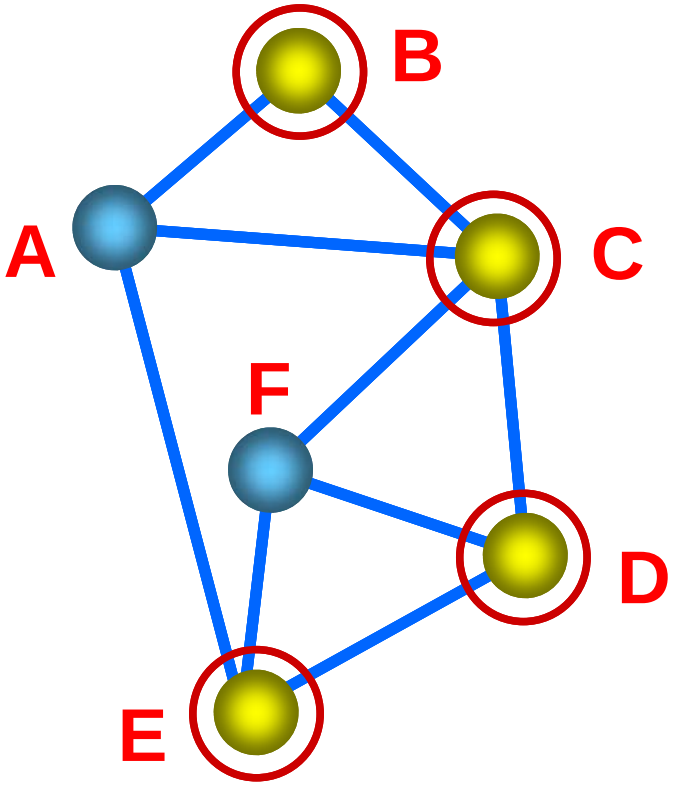
正確的作法是 B1 或 B2 必須 disable 掉其中一條 link 的運作，讓整個網路不存在 loop。如右上圖，我們可以把整個網路用 graph 來表示，其中的 circular node 表示爲 LAN，square node 表示 bridge。每一條 edge 上面的 weight 表示成本（傳輸時間），那麼建構一個成本最低的 bridge network 相當於尋找 minimum cost spanning tree。在附錄，我們將看到此一演算法。

註：最後虛線的 links 通通都需 disable 掉。

Radia Perlman：Making the Bridge Practical

發明 bridge 的二年後，1985 年，Radia Perlman（Mark Kempf 的部下）發明 distributed minimum cost spanning tree 演算法，用來解決 broadcast storm problem，隔年獲得 MIT 博士學位。此一演算法在 1990 年正式成爲 IEEE 802.1d 國際標準。1997 年， Radia Perlman 加入 Sun Micro（發明 Java 的公司），擔任網路部門執行長，至今共獲得 80 幾件專利。

Vertex Cover Problem

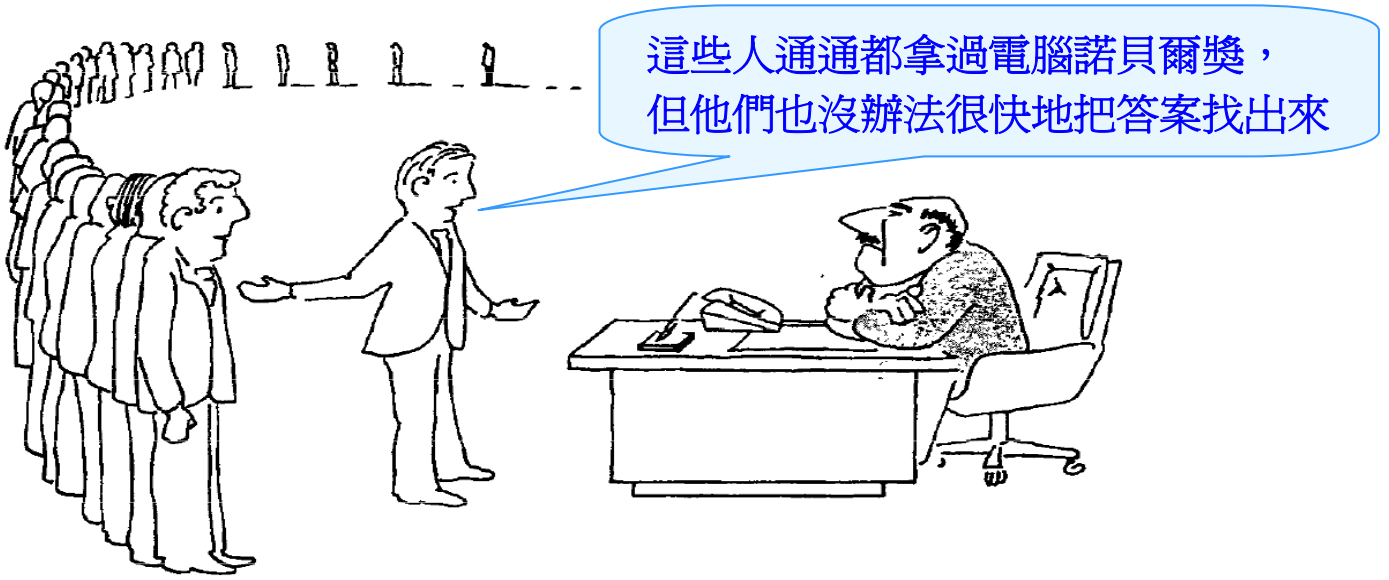


接下來我們來看一個看起來很容易理解的問題

- Input: a graph G
- Output: a smallest vertex subset of G that covers all edges of G.

→ 如何在最少的道路交叉口擺路燈，以照亮所有的道路

Not All Problems Can Find Efficient Algorithms



"I can't find an efficient algorithm, but neither can all these famous people."

學習演算法，除了想學解題方法之外，有時候我們對問題的本身也很感興趣。Vertex Cover Problem 被證明是屬於 **NP-complete**。 這暗示著：這個問題至今沒有人有辦法找出有效率的演算法（在多項式的步驟之內解決此一問題），即使是拿過 Turing Award 的人也沒辦法。（注意：**NP-complete problem** 不是無解，而是還沒有找到多項式時間的解）

如果有一天你的指導教授（老闆）叫你解決這個問題該怎麼辦？

重金懸賞：CMI Millennium Prize

http://www.claymath.org/millennium/

- Clay Mathematics Institute (Cambridge, MA, USA) offered US$1,000,000（一百萬美金，約三千萬台幣）for each of seven open problems on May 24, 2000 at Paris.
  - | Birch and Swinnerton-Dyer Conjecture | Hodge Conjecture | Navier-Stokes Equations | P vs NP | Poincare Conjecture | Riemann Hypothesis | Yang-Mills Theory |

Approximation Algorithms to NP Problems

雖然沒有人可以證明 NP-complete problems 是否一定不存在有效率的演算法，但一般相信是不存在的。

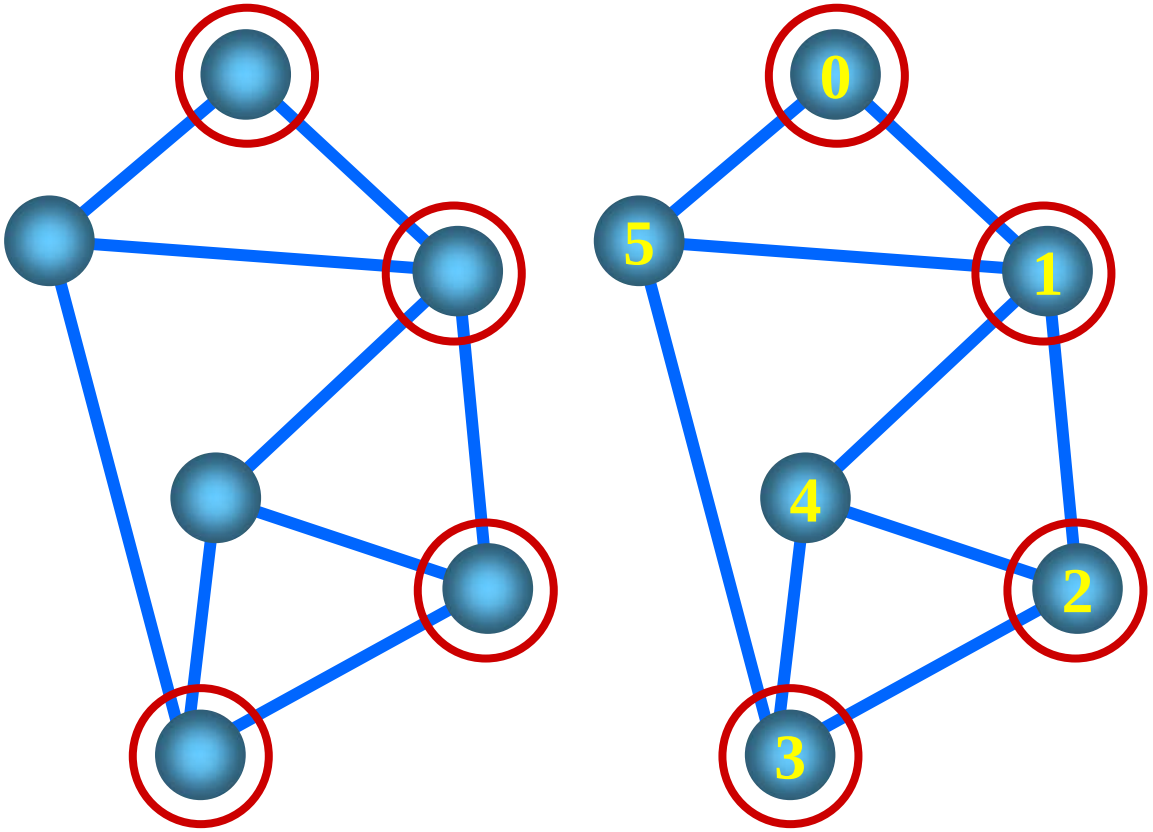
遇到 NPC 問題時，你可以跟你的老闆商量一下：如果你有辦法設計一個有效率的演算法，雖然未必是最佳解，但可以保證品質不會太差，看你老闆是否能接受。這就是 **approximation algorithm** 的精神：**放下對完美的堅持，往往就可以找到新的出路（退一步海闊天空）**

安徽省 桐城縣

康熙大學士兼禮部尚書張英

千里修書只為牆，
讓他三尺又何妨。
長城萬里今猶在，
不見當年秦始皇。

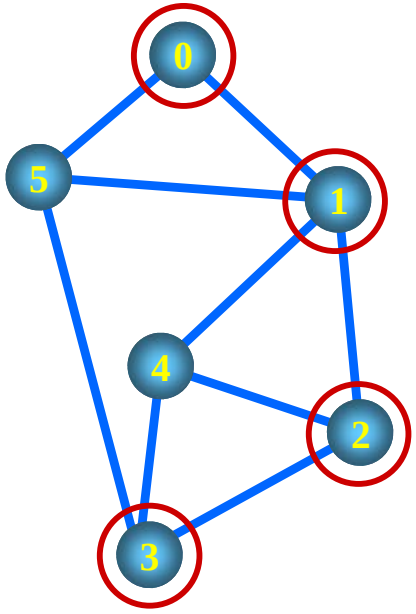
Approximation Algorithm for Vertex Cover Problem



- **Initially, let *S* be an empty set.**
- **Repeat until *G* has no edges:**
  - **Arbitrarily choose an edge (*u*, *v*) of *G*.**
  - **Insert *u* and *v* into *S*.**
  - **Delete all edges of *G* incident to *u* or *v*.**
- **Output *S*.**

Approximation Algorithm Written in C

再次強調：
給定演算法，
同學們要有辦法
自行寫出完整的
程式碼。




```
#include <stdio.h>
#define vertexNumber 6
int graph[vertexNumber][vertexNumber] = {
        {0, 1, 0, 0, 0, 1}, {1, 0, 1, 0, 1, 1}, {0, 1, 0, 1, 1, 0},
        {0, 0, 1, 0, 1, 1}, {0, 1, 1, 1, 0, 0}, {1, 1, 0, 1, 0, 0},
    };
static int vertexSet[vertexNumber];
int isEmptyEdge(void) {
    int i, j;
    for (i=0; i<=vertexNumber-1; i++)
        for (j=0; j<=vertexNumber-1; j++)
            if (graph[i][j]==1)
                return 0;
    return 1;
}
void clearEdge(int i) {
    int j;
    for (j=0; j<=vertexNumber-1; j++) {
            graph[i][j]=0;  graph[j][i]=0;
    }
}
void main(void) {
    int i, j;
    while (isEmptyEdge()!=1) {
        for (i=0; i<=vertexNumber-1; i++)
            for (j=0; j<=vertexNumber-1; j++)
                if  (graph[i][j]==1) {
                    vertexSet[i]=1; vertexSet[j]=1;
                    clearEdge(i); clearEdge(j);
                }
    }
    for (i=0; i<=vertexNumber-1; i++)
        if (vertexSet[i]==1)
            printf("node %d in Vertex Cover Set \n", i);
}
```

Three Questions about Approximation Algorithm

- ✓ Q1: Is the output vertex set indeed a vertex cover of the input graph?
- ✓ Q2: Does the algorithm run in polynomial time?

  往後課程會看到，只需 $O(\max\{V, E\})$ 的時間
- ✓ Q3: Is the quality of the output solution close to optimal?

本課程將會學習到此一演算法所產生的 **vertex cover set** 大小，即使在最糟糕的情況之下，最多只會是 **optimal solution** 的 **2** 倍。我們稱這樣的 **approximation algorithm** 爲 **2-approximation algorithm**。

Not All Approximation Algorithms Are Obvious

下面這個 approximation algorithm 2 應該比剛才那個好吧？

- **Initially, let *S* be an empty set.**
- **Repeat until *G* has no edges:**
  - **Arbitrarily select a vertex *u* of highest degree.**
  - **Insert *u* into *S*.**
  - **Delete all incident edges of *u*.**
- **Output *S*.**

讓我們檢驗這個 graph

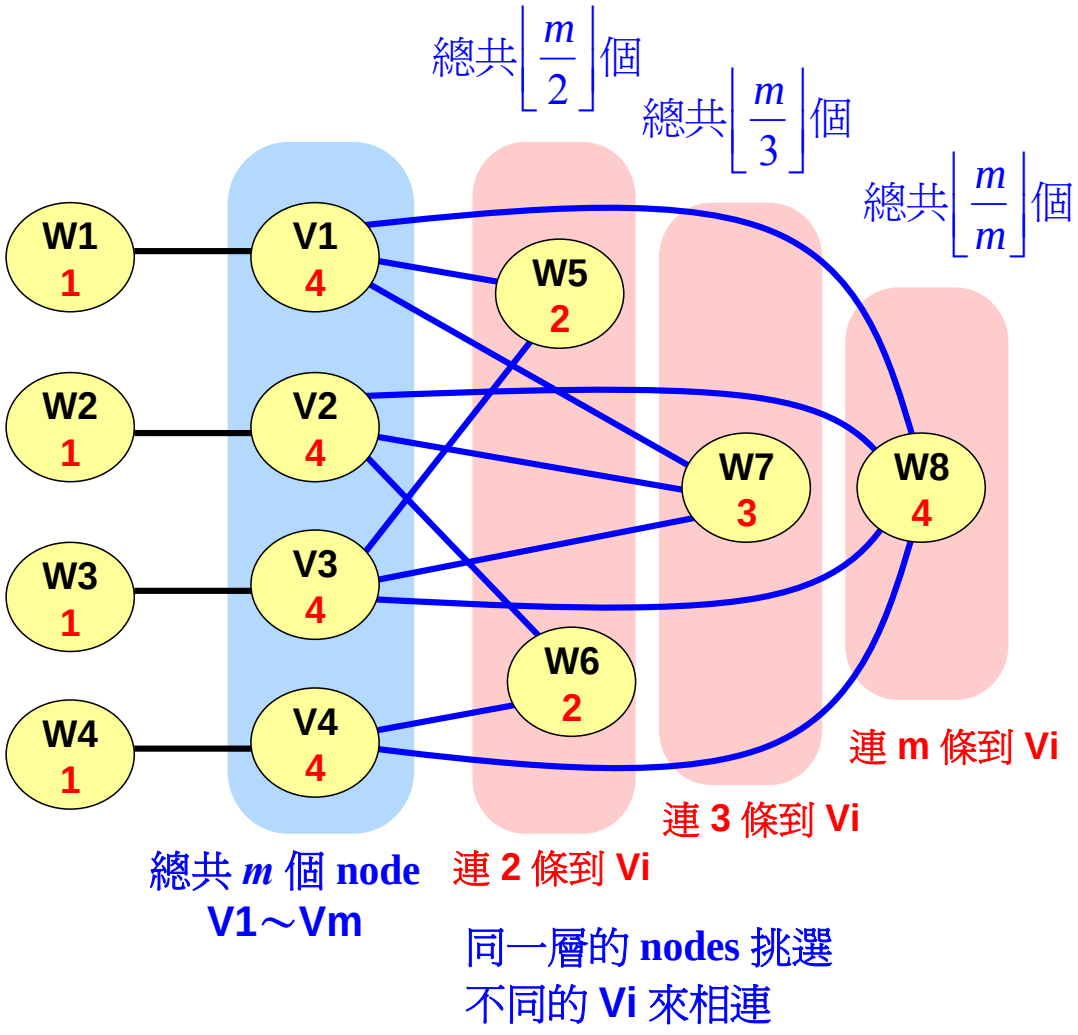
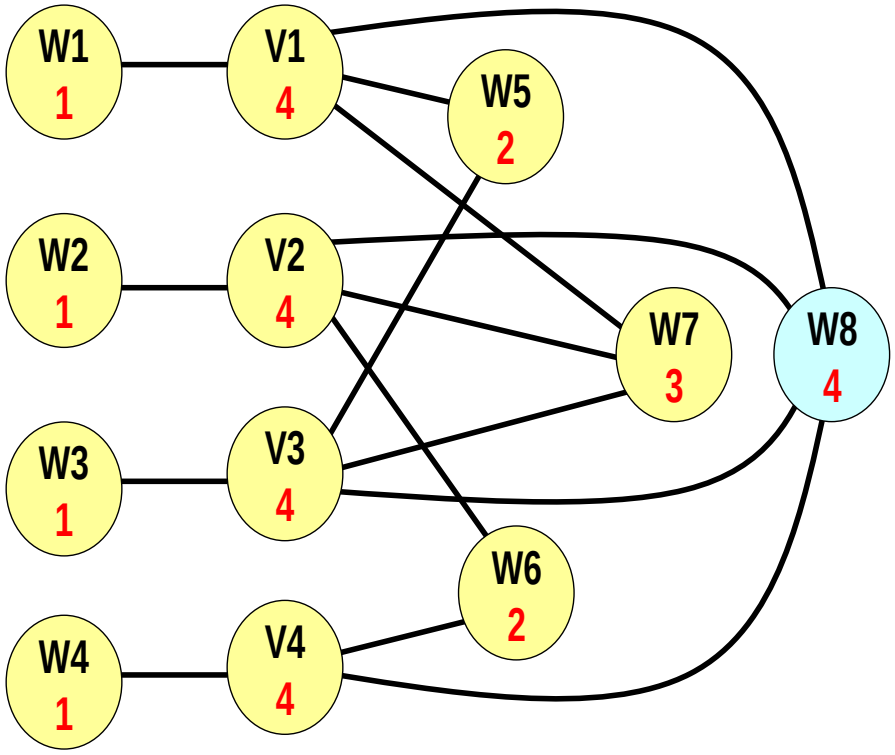


Illustration (1/2)

Step 1



Step 2

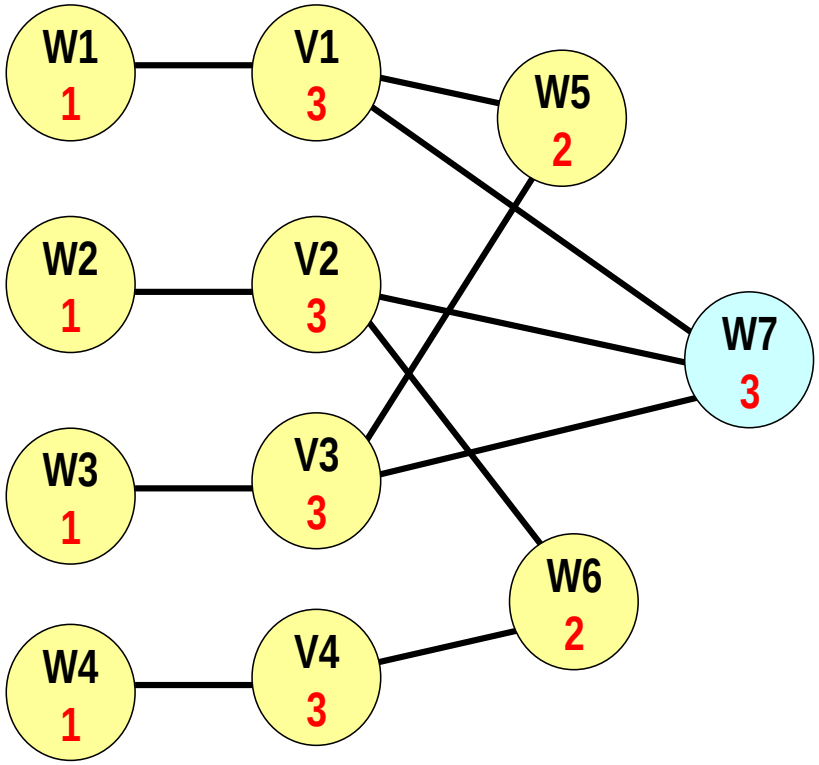
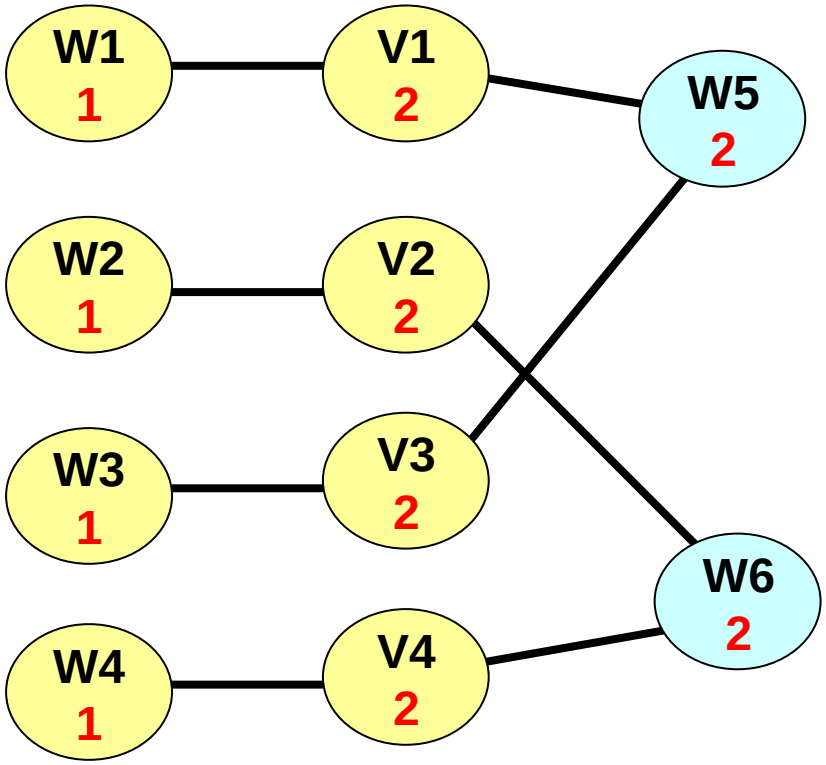
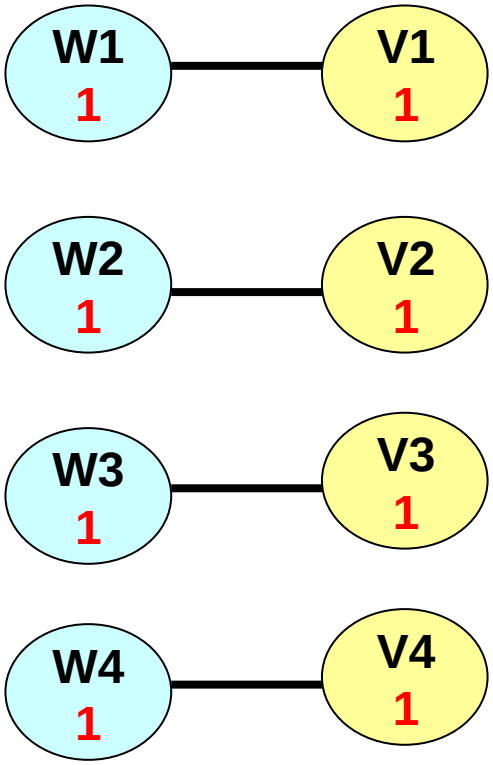


Illustration (2/2)

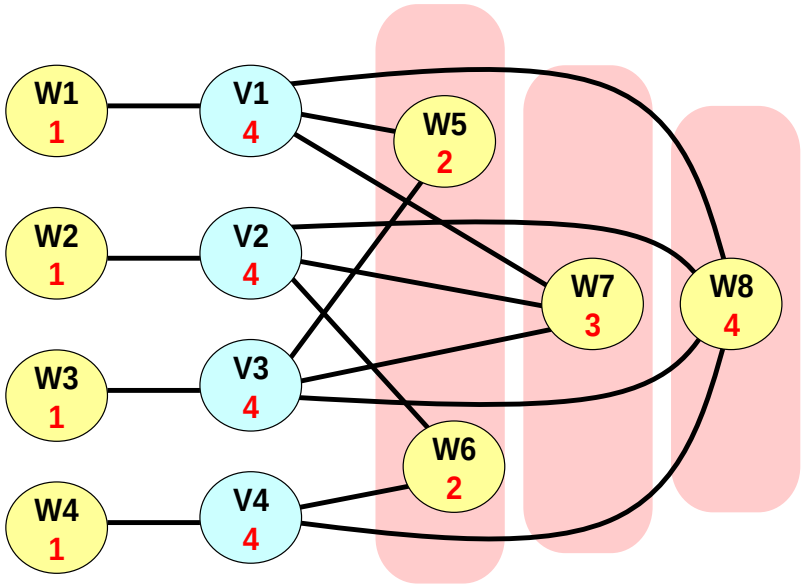
Step 3

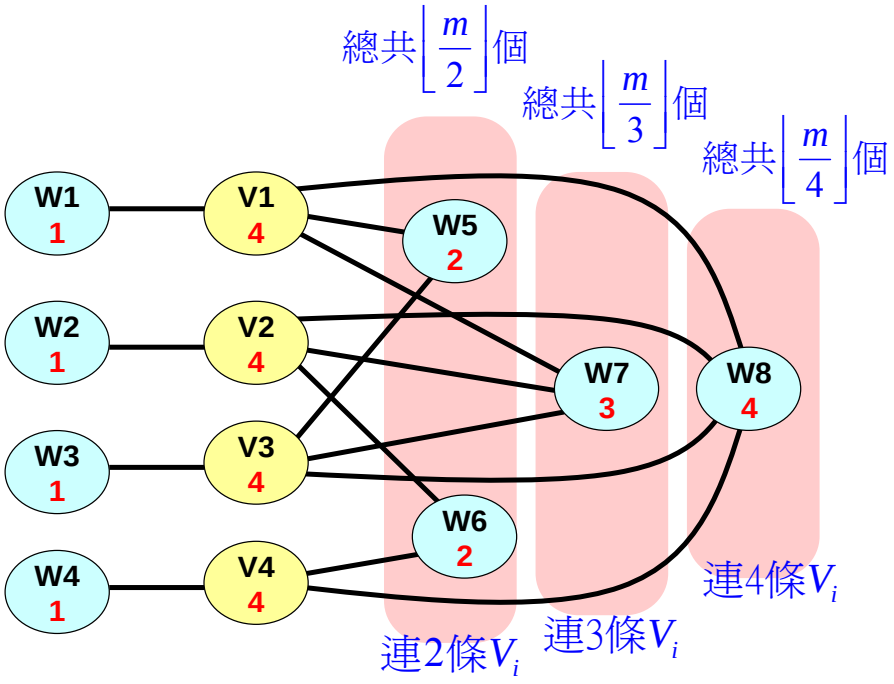


Step 4



Approximation Algorithm 2 Is Even Worse





左圖是 optimal solution（藍色 node），共有 m 個 vertices

右圖是 approximation algorithm 2 最糟糕的結果（藍色 node），可能選到

$$m+\left\lfloor \frac{m}{2} \right\rfloor+\left\lfloor \frac{m}{3} \right\rfloor+\left\lfloor \frac{m}{4} \right\rfloor+\left\lfloor \frac{m}{5} \right\rfloor+\left\lfloor \frac{m}{6} \right\rfloor+\cdots+\left\lfloor \frac{m}{m} \right\rfloor=\sum_{j=1}^{m}\left\lfloor \frac{m}{j} \right\rfloor \text{ 個}$$

當 $m \geq 6$，vertex cover set 的大小超過 optimal 的 2 倍。這說明

有時候設計 approximation algorithm 並不是非常直覺

How Goodness of the 2-Approximation Algorithm

- The 2-approximation algorithm was known for 30 years.
- It remains the best known approximation algorithm for the vertex cover problem!
- Finding a 1.166-approximation is known to be NP-complete.
- Even a 1.9-approximation would be a significant breakthrough.
- 讓我們向公園路燈管理局致敬 ☺

Max 3-SAT Problem

令變數 x_i 爲 Boolean variable（其值爲 0 或 1）

若一個 Boolean formula F 總共有 m 個括弧， n 個 Boolean variables，

例如：$F = \left(x_1 \vee \overline{x_2} \vee x_4\right) + \left(x_2 \vee x_3 \vee \overline{x_4}\right) + \left(\overline{x_1} \vee \overline{x_3} \vee x_5\right)$

括弧裡頭的 Boolean variables 最多只有 3 個，

且用 OR（用符號 $\vee$ 表示）來連結；而括弧之間用 +（或用 AND 連結）

則我們稱此一 Boolean formula F

具有 3-conjunction normal form（簡稱 3-CNF）

MAX 3SAT Problem：

給一個具有 3-CNF 的型式 Boolean formula F ，

裡頭總共有 m 個括弧， n 個 Boolean variables，

要如何決定每一個 Boolean variable x_i 的值才能讓 F 的值最大

例如：當 $x_1 = x_3 = x_5 = 1$ 且 $x_2 = x_4 = 0$ 時，$F = 3$ （最大值）

The Power of Randomized Algorithm

MAX 3-SAT Problem 剛好也是 NP-complete problem，所以幾乎不太可能找到有效率的演算法，所以我們如何退而求次，設計一個 approximation algorithm？ 是不是很難，想不太出來？

```
randomized_MAX-3SAT(void)
{
  for (i = 1; i <= n; i++)
     x_i = random{0,1};
}
```

擲杯問神明最快！ 對於每個 x_i ，擲到「有杯」就設爲 **1**，否則設爲 **0**。
→ 我們將會學到這樣的方法所產生的解，其期望值最少有最佳解的 **7/8**。

瞎貓碰見死老鼠！ 沒想到擲杯居然這麼好用。

Syllabus

由於課本非常的厚（共 **1292** 頁），我們只挑下列主題來介紹

- Part 1：對於不是 NP-complete problems，我們將學習下列技術（並搭配學習演算法的效能分析技巧）：
  - Divide-and-Conquer
  - Randomized Algorithms
    （亦可成爲 **NP-complete problems** 的妥協方案之一）
  - Dynamic Programming
  - Greedy Algorithms
  - Graph Algorithms（大量用在網路相關的問題）
  - Linear Programming and Game Theory（最佳化問題，論文經常遇到）
- Part 2：如何判斷一個問題是否爲 NP-complete
- Part 3：Approximation Algorithm
  - 遇到 **NP-complete problems** 的妥協方案之一

自我評量

1. 試設計一個例子證明利用下面的 approximation algorithm 求出來的 vertices 數目可能超過 optimal 值的 2 倍。

- **Initially, let *S* be an empty set.**
- **Repeat until *G* has no edges:**
  - **Arbitrarily select a vertex *u* of highest degree.**
  - **Insert *u* into *S*.**
  - **Delete all incident edges of *u*.**
- **Output *S*.**