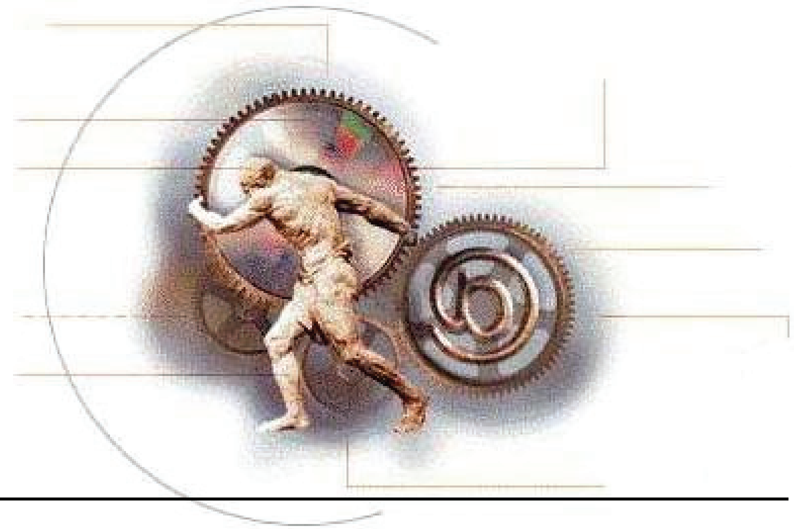


Chapter 8

NP Theory序論

An Introduction to the Theory of NP

Outlines

- 本章重點
  - Polynomial Time
  - Intractability
  - Optimization Problems vs. Decision Problems
  - The Theory of NP
    - P problem
    - NP problem
    - NP-complete problem
    - NP-hard problem
  - 如何証明某個問題為NP-complete問題

Polynomial Time (多項式時間)

- 什麼是　“多項式時間 **(Polynomial Time)**”?

Polynomial Time

Non-Polynomial Time

$f(n) \setminus n$	10	10^2	10^3
$log_2 n$	3.3	6.6	10
n	10	10^2	10^3
$nlog_2 n$	0.33×10^2	0.7×10^3	10^4
n^2	10^2	10^4	10^6
2^n	1024	1.3×10^3	$> 10^{100}$
$n!$	3^6	$> 10^{100}$	$> 10^{100}$

多項式時間的演算法(Polynomial-time Algorithm)

- **Def:**
  - 一個稱為多項式時間的演算法(**Polynomial-time Algorithm**) 必須符合：在合理的輸入大小 (input size)下，該演算法於最差情況(Worst-case)的時間複雜度以多項式函數為限。
  - 因此，若n為input size，存在一個多項式函數 $p(n)$ ，則：

$$W(n) \in O(p(n)).$$

- **Polynomial-time computable**
  - 一函數 f(x) 為polynomial-time computable，若且為若存在一演算法，使得對所有的輸入 x，皆可在Polynomial Time內求得 f。

Intractability (難解問題)

- 如果我們講「這個問題很難」，這句話可能有兩種不同的意義：
  - [意義 1]: 這個問題也許目前已經有一些還不錯的近似解法，只是想進一步找出真正最佳的方法是件困難的事
  - [意義 2]: 這個問題本身就難以找出解決方法。
- 第一種意思指的是對人而言很困難，而第二種意思指的是對計算機而言很難。
- 在探討問題的難度時，比較正確的講法應該是指一個問題是易解的 (**tractable**) 或是難解的 (**intractable**)。

難解 (Intractable)問題

- 在資訊科學領域中，若無法在最差情況(Worst-case)下，以多項式時間的演算法來解決某個問題，該問題就被稱為難解 (**Intractable**)問題。
  - 一個難解的問題，必須沒有任何多項式時間的演算法可以解它。
- 但是，如果有一個問題在最差情況(Worst-case)下，目前還找不到一個Polynomial-Time Algorithm解它，但是也無法保証未來就找不到Polynomial-Time Algorithm來解這個問題，則*無法証明該問題是* ***Intractable***.
  - For example:
    - 早期，利用 **brute-force algorithm (暴力演算法)** 解連鎖矩陣相乘問題 (Chained Matrix Multiplication problem) ，其時間複雜度為**non-polynomial time**.
    - 然而，若以 **dynamic programming algorithm** (Algorithm 3.6) 來解，則其時間複雜度為$\Theta(n^3)$.

- 就難解性而言，問題的主要分類可分為三種:
  - Problems for which polynomial-time algorithms **have been found**
    - 如：最短路徑問題、MST問題、排序問題、搜尋問題...
  - Problems that have been proven to be **intractable**
    - 時間複雜度被証明為**指數複雜度(以上)**的問題。如：河內塔問題
    - 不存在有解決問題之演算法的問題。如：程式停止問題 (Halting Problem)、功能相等問題(Equivalence Problem)...
  - Problems that have not been proven to be intractable, but for which polynomial-time algorithms have never been found

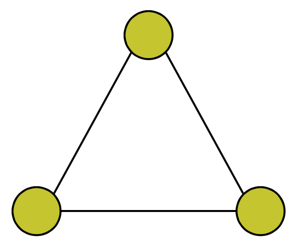
大多數的問題不是落在第一類，就是落在第三類。

- 第三類問題中，頗具知名度的是旅行推銷員問題 **(Traveling Salesman Problem)**

The Traveling Salesman Problem; TSP

- **TSP問題：**
  - 一個銷售員會不斷地花費時間去拜訪**n**個城市。**(A salesman spends his time visiting *n* cities cyclically.)**
  - 在一趟的旅程中，他只會拜訪每一個城市一次，而且當他回到原本的起始城市後就會停止此趟的拜訪旅程。**(In one tour he visits each city exactly once, and finishes up where he started.)**
  - 什麼樣的拜訪旅程會使該銷售員所花費的旅行距離(成本)最少？**(In what order should he visit the cities to minimize the distance traveled?)**

- 若採用暴力法去解**TSP**問題，則會發現要找出所有可能的路徑所花費的時間是呈**指數 (Exponentially)** 成長的**!!**
  - **3 cities → 1 solution.**
  - **10 cities → 181,440 possible tours**
  - **n cities → (n-1)!/2 possible tours**



- 若 n=26，則有 25! /2條不同路徑：
  - 25!=15511210043330985984000000≅1.55×10^{25}這個數字寫來輕鬆，究竟有多大？
  - 假設電腦每秒可計算 10^6 條路徑的成本，一年有 3.15×10^7秒， 故一年可計算 3.15×10^{13}條路徑，求出所有路徑的成本需時

$$\frac{1.55 \times 10^{25}}{3.15 \times 10^{13}} \cong 5 \times 10^{11}(\text{年})$$

  - 即便是對不太大的 n=26，就需時五千億年，顯然這種方法毫無用處。

Optimization Problem vs. Decision Problem

- 在所有的問題當中，除了過去所討論過的各種最佳化問題 (Optimization Problem) 以外，尚有另外一種型態的問題: 決策問題 **(Decision Problem)**
  - **Decision Problem**: 此類問題輸出的答案非常簡單，就是 “**yes**” 或 “**no**” 兩者之一。

- **The partition problem (分割問題):**
  - 給予一組正整數的集合$S=\{a_1, a_2, \dots, a_n\}$，問: 是否可以將其分割成兩個子集合$S_1$與$S_2$，而此兩個子集合的個別總和相等。
  - Ex: Let $S = \{13, 2, 17, 20, 8\}$. The answer to this problem instance is "**yes**" because we can partition S into $S_1 = \{13, 17\}$ and $S_2 = \{2, 20, 8\}$.
- **The Sum of Subset Problem (部份集合的和問題):**
  - 給予一組正整數的集合$S=\{a_1, a_2, \dots, a_n\}$及一個常數$c$，問: 集合$S$中是否存在一組子集合$S'$，此子集合$S'$的數字總合為$c$。
  - Ex: Let $S = \{12, 9, 33, 42, 7, 10, 5\}$ and $c = 24$.
    - The answer of this problem instance is "**yes**" as there exists $S' = \{9, 10, 5\}$ and the sum of the elements in S' is equal to 24.
    - If c is 6, the answer will be "**no**".

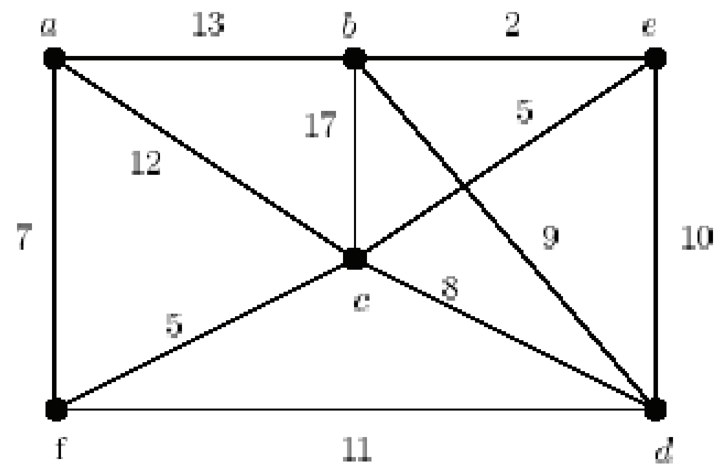
- **The Satisfiability Problem (滿足問題; SAT):**
  - 給一個布林函數E，我們對存在於此函數E中的一些變數分別指派True或False，使這個函數結果為True。
  - Ex: Let $E = (-x_1 \vee x_2 \vee -x_3) \wedge (x_1 \vee -x_2) \wedge (x_2 \vee x_3)$. Then the following assignment will make E true and the answer will be “**yes**”.

    $$x_1 \leftarrow F,\ x_2 \leftarrow F,\ x_3 \leftarrow T$$

  - If E is $-x_1 \wedge x_1$, there will be no assignment which can make E true and the answer will be “**no**”.
  - 此問題為第一個被証明是屬於**NP-Complete**的問題 (by S. A. Cook, 1971).
- **The Minimal Spanning Tree Problem (最小擴張樹問題):**
  - Given a graph G, find a spanning tree T of G with **the minimum length**.

◆ The Traveling Salesperson Problem (旅行推銷員問題):

- 給予一個圖G = (V,E) ，找出一個由該圖的某一點出發所構成的cycle，此cycle會經過該圖中的每個點一次而再回到出發點，同時其總長度為最短
- Ex: Consider following graph. There are two cycles satisfying our condition. They are C_1 = a→b→e→d→c→f→a and C_2 = a→c→b→e→d→f→a. C_1 is **shorter** and is the solution of this problem instance.



- Some problems:
  - The Partition Problem (Decision Problems)
  - The Sum of Subset Problem
  - The Satisfiability Problem
  - The Minimal Spanning Tree Problem
  - The Traveling Salesperson Problem

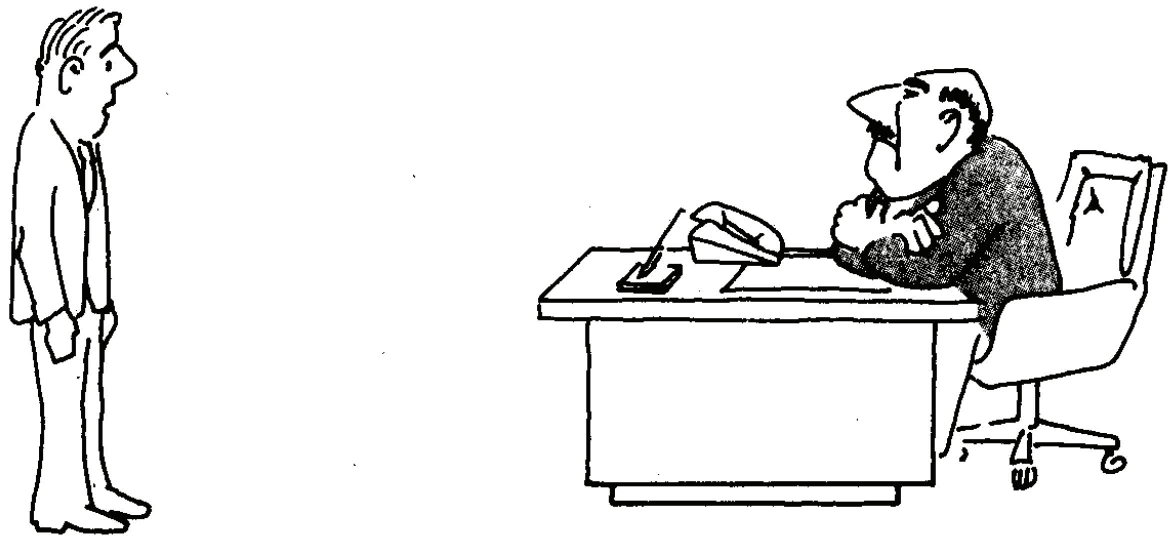
  (Optimization Problems)

- 一般來說，最佳化問題(Optimization Problems)比決策問題(Decision Problems)要來得難處理!!

- 最佳化問題均可找出一個與其對應的決策問題。
- **For example (MST Problem):**
  - Given a graph *G* and a constant *c*.
  - The total length of the spanning tree of the graph *G* is *α*:
    - If $\alpha < c$, then the answer is **"yes"**,
    - **otherwise**, its answer is **"no"**.
- 這個決策版本的最小擴張樹問題，可以稱為最小擴張樹決策問題(**The minimal spanning tree decision problem**)
- 如果要解某個最佳化問題的決策版本(Decision version of the optimization problem)已經很困難了，則該問題的最佳化版本一定更難解決。

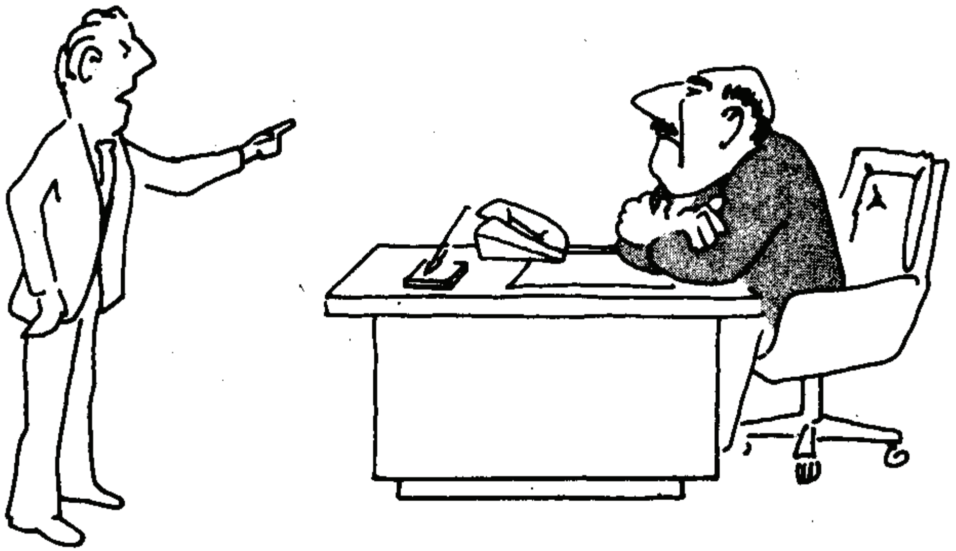
The Theory of NP

- 假設你在一個公司上班。有一天，上司叫你去為某個對公司很重要的問題找出有效率的演算法。
- 結果，你研究了很長的一段時間，沒有任何進展，你去找你的上司...



我想不出好方法，我可能太笨了！

◆結果，你的上司說要開除掉你這個豬頭，並由一個演算法設計專家來取代你。此時，你很不爽地對他說...



我想不出好方法，
因為不可能有這種好方法！

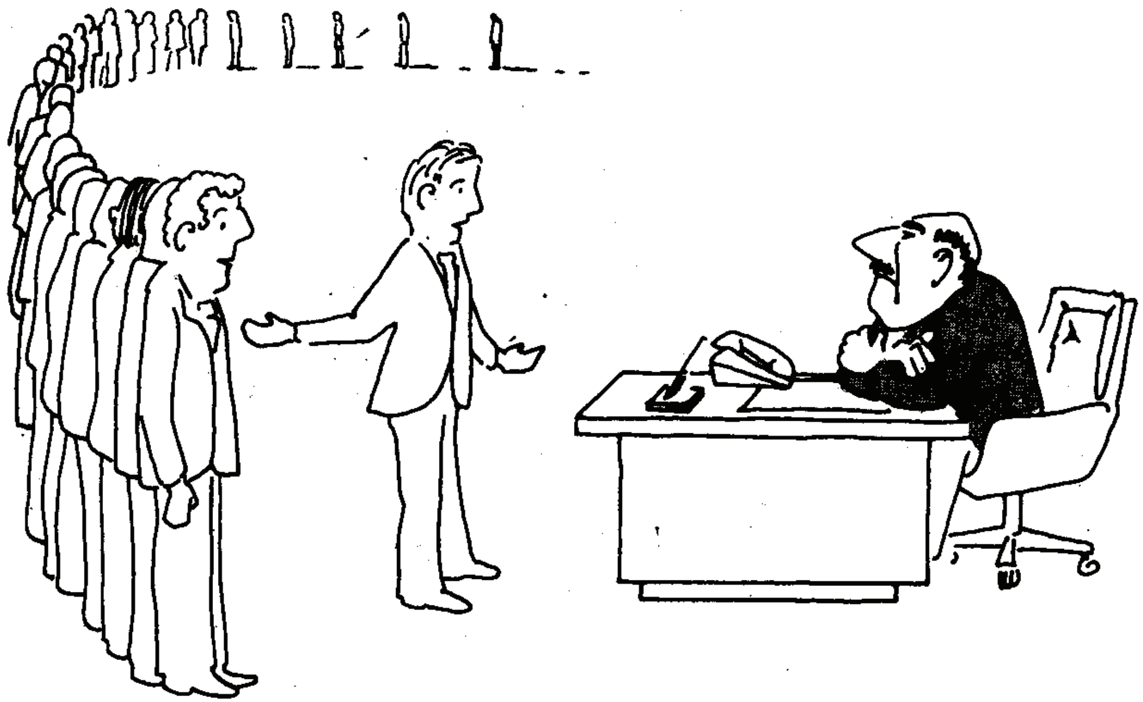
- 你的上司因為你的強辯，很不情願地再給你一段時間去証實你說的話。
- 結果，你又試了很長的一段時間，還是失敗了!!此刻的你既無法找出一個有效率的演算法，又無法証明這樣的演算法是不存在的...

你快要被開除了

世界上很多的電腦科學家正在為旅行推銷員問題(**TSP**)找尋一個較有效率的演算法。但是，到目前為止卻沒有人能發展出一個<u>在最差情況下，時間複雜度比指數複雜度要來得好</u>的**TSP**演算法。不過，<u>也沒有人証明出找到這種演算法是不可能的</u>…

- 你找到了一線生機!! 因為，你只要能証明找出公司問題之有效率演算法的難度和找出旅行推銷員問題的有效率演算法是**一樣難**的 (亦即：兩者是同一類的問題)，代表著上司要求你解決的問題也曾難倒很多電腦科學家
- 你終於証明出來公司的問題和旅行推銷員問題是同一等級的…

◆你被加薪了，因為你讓公司節省了許多經費。



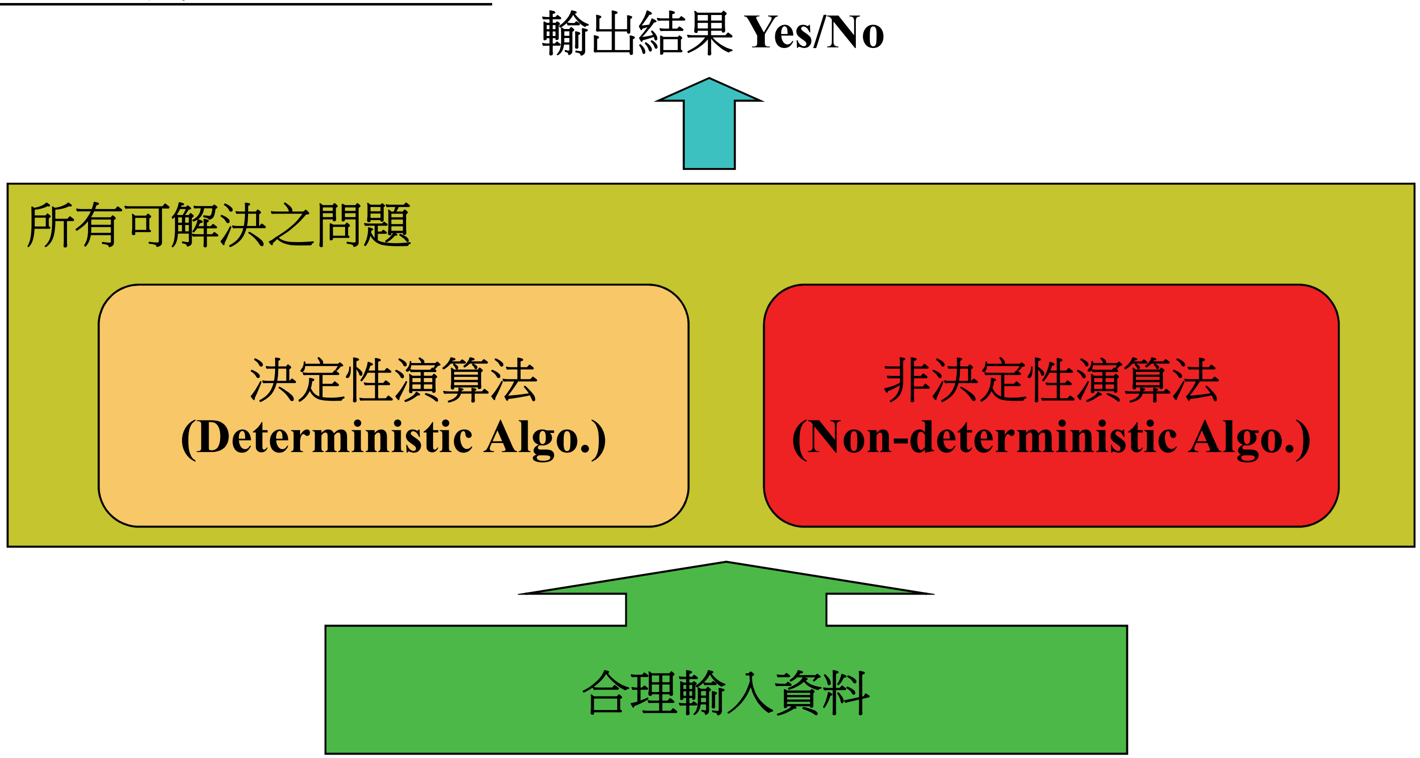
我想不出好方法，
因為這些名人專家也不會！

- 思考：
  - Computer Science對於電腦所處理之問題的難易區分標準為何？
  - 將一個問題歸類(轉化)(reduction or transform)為某一個已知問題的概念與作法為何？
  - 証明一些問題很難為何這麼重要？

以下課程內容所討論到的問題，若無特別說明，皆以“決策版本”的問題為主

Deterministic v.s. Non-deterministic

- 除了不存在有解決問題之演算法的問題以外，在可以解決的問題當中，又可以分成 “簡單問題” 和 “困難問題” 兩類。而問題的難易之分取決於<u>所使用的演算法類型</u>或<u>使用之計算機器的類型</u>：



- **Deterministic Algorithm** (決定性演算法)
  - **Def:** 這類演算法在做任何事時，該演算法的下一步只有一件事可以做。**(Permitting at most one next move at any step in a computation)**
  - 是指演算法中每一個步驟的運算都需要被唯一定義，因此產生的結果也是唯一的。
  - 能夠執行決定性演算法的機器，稱為決定性的機器 **(Deterministic Machine)**。電腦就是一種決定性的機器。
    - 由於在此類計算機器運作的演算法在處理問題時，每一步只有一件事，因此，只要有一個處理器即可，故容易實現。

- **Non-deterministic Algorithm** (非決定性演算法)
  - **Def:** 這類演算法在做任何事時，該演算法的下一步可能會有無限多件事可以選擇。 **(Permitting more than one choice of next move at some step in a computation)**
  - 演算法中每一個步驟的運算無法被唯一定義。
  - 能夠執行非決定性演算法的機器，稱為非決定性的機器 **(Non-deterministic Machine)**。
    - 由於非決定性演算法在執行時，每一步可能有無限多件事要處理，故非決定性計算機器需假設**有無限多個處理器可平行處理**。因此，非決定性計算機器的計算能力比決定性計算機器要強大。
    - 但是，實際上並不存在此種機器。

- **Non-deterministic Algorithm**的執行步驟分成兩個階段：
  - **猜測階段(Guess)**
    - 由於沒有一個既定的程序來從事此階段的猜測工作，因此本階段是 **Non-deterministic**
    - 對於本階段，我們只知道一件事：
      - 如果一個問題有正確解的話，此階段一定可以將這個正確解給猜出來；反之，若該問題沒有正確解的話，則此階段就會隨便給解答。
      - 至於猜測階段是怎麼將這個解答給找出來的，我們無從得知(不論所給的解是否為正確解)。
  - **驗証階段(Verification)**
    - 將上一階段所猜出來的結果加以驗証是否為真 **(True)**

Nondeterministic SAT

- 以一個具有n個變數之布林函數E之滿足問題 (SAT) 為例：
  - 如果此問題是使用決定性演算法，則時間複雜度為$O(2^n)$
  - 如果此問題是使用非決定性演算法，則時間複雜度為$O(n)$

/* Guess */

for i = 1 to n do

　　$x_i \leftarrow$ choice(true, false);

/* Verification */

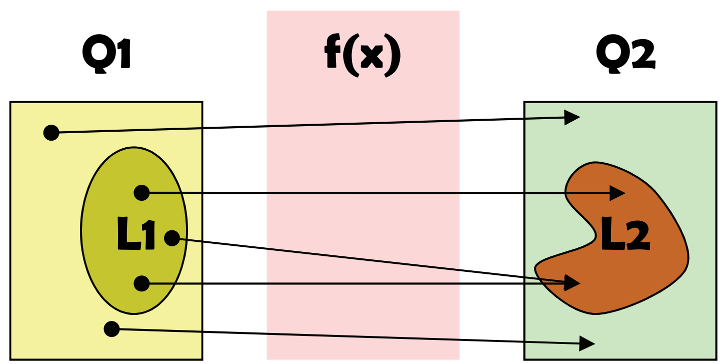
if $E(x_1, x_2, \ldots, x_n)$ is true then

　　success;

else failure;

Polynomial-Time Reducible (多項式時間的轉化)

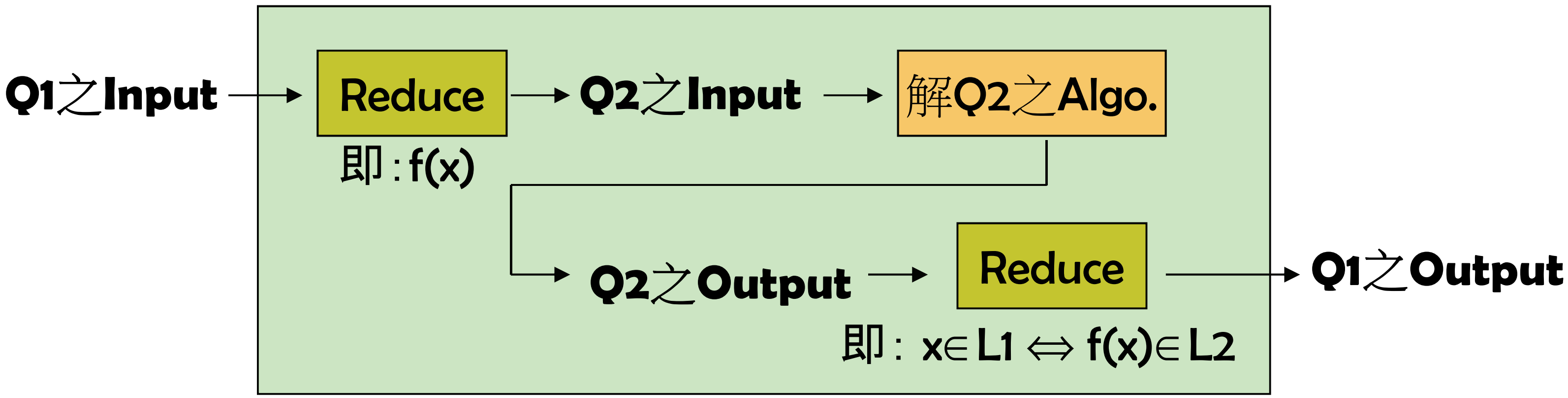
- **Def:** 若有兩個問題Q1和Q2，其解集合分別為**L1**和**L2**：
  - 如果**Q1**可以多項式時間轉化成**Q2** (即：**L1** $\leq_p$**L2** 或 **Q1** $\leq_p$**Q2**)，則
    - 存在一個函數 **f(x)** /*此函數不一定是實質的數學公式，也可能是一個虛的概念或意涵*/
    - 此函數 **f(x)** 為**polynomial-time computable**
    - 該函數 **f(x)** 使得對所有**x**而言，**x∈L1** 若且唯若 **f(x)∈L2** (**x∈L1** ⇔ **f(x)∈L2**)



所有在**L1**內的元素，經**f(x)**轉化後必在**L2**中；而原本不在**L1**內的元素，經**f(x)**轉化後則必不在**L2**中

- 這個**Reduce**的作用類似 "歸類" 。**Q1**和**Q2**可被視為同一類型的問題。

- 為何要做**Reduce?**
  - **Q1 reduce** 成**Q2**，表示**Q1**問題可以由處理**Q2**問題的演算法所解決。



- 前述定義隱含一些概念：
  - **Q1**問題可以由處理**Q2**問題的演算法所解決。若**Q2**問題有一個有效率的演算法，可以在多項式時間內將**Q2**問題給解掉，表示**Q1**問題也一定可以在多項式時間內被解掉。
  - 因此，函數 **f(x)** 必須是要 **polynomial-time computable**。
    - 這是因為解**Q1**問題的時間相當於 "函數**f(x)**的轉換時間 **+** 解**Q2**問題之演算法的解題時間" 所構成。
    - 若轉換時間過長，則解**Q2**問題之演算法就算是再快，也無助於加速對**Q1**問題之求解。
- 可表示成 $L1 \propto L2$ 或 $Q1 \propto Q2$

- Q1 $\propto$ Q2的意義：
  - Q2問題比Q1問題難 (雖然兩者是同一類型的問題)
  - 若Q2有解，則Q1就有解
  - 想要証明Q1和Q2一樣難，則需証出Q1 $\leq_p$Q2 且Q2 $\leq_p$Q1
  - 若Q1 $\propto$ Q2且Q2 $\propto$ Q3，則Q1 $\propto$ Q3 (遞移性)
- <u>範例</u>：假設現在有下列兩個問題：
  - Q1: 一台電梯中有4個人，其中是否有三個人彼此互相認識?
  - Q2:有一個4個頂點之無向圖G=(V,E)，其中是否存在一個三角形?

証明Q1 $\propto$ Q2。

◆ 解：

- 有一個函數f，使得：
  - 人$_i$ → 頂點vi
  - 人$_i$和 人$_j$互相認識 → (vi, vj)∈E
  - 人$_i$和 人$_j$互不認識 → (vi, vj)∉E

  此函數為一個虛的轉換概念
- 說明此函數f 為polynomial time computable
  - 因為轉換後的圖G=(V,E)，是以相鄰矩陣表示，該矩陣的大小為 $|V|^2$。若要將該矩陣填滿值則需時 $O(|V|^2)$。
  - ∴函數f為Polynomial time computable
- 說明該函數f(x)使得對所有x而言，x∈L1 ⇔ f(x)∈L2
  - 有三人互相認識 ⇔ 有一個三角形在圖形G中
  - (証明⇒)：人$_l$和 人$_j$和 人$_k$互相認識 ⇒ vi， vj和vk兩兩有邊相連⇒ vi， vj和vk形成三角形
  - (証明⇐)：(理由同上)，人$_l$和 人$_j$和 人$_k$互相認識

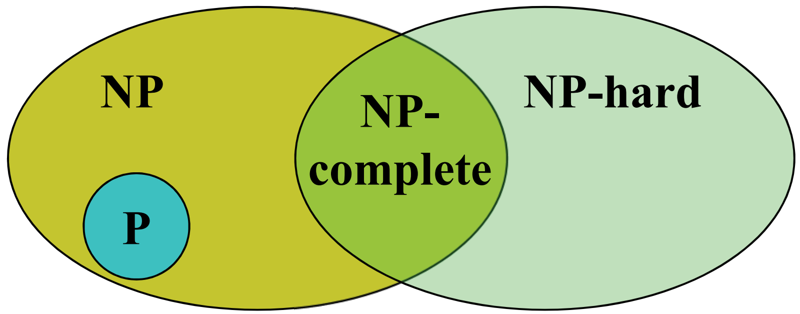
Q1 ∝ Q2得証

P, NP, NP hard, NP complete

- **P:**
  - 是一群Decision Problem的集合，這些問題皆可利用Deterministic Algorithm於Worst Case的情況下，在**Polynomial Time**的複雜度內被解決。
- **NP:**
  - 所謂 "NP" 是指Non-deterministic與Polynomial兩個單字的簡寫
  - 這類的問題只要給個解答，可以在Polynomial Time的複雜度內很快地驗證 **(Verify)** 出這個解答是否正確。
    - 由於決定性演算法為非決定性演算法的一個特例，且容易找到答案也會容易驗証答案。因此，P可視為NP的一個特例。
    - $P \subseteq NP$ (✓)
    - $P = NP$ (?)

- **NP-hard:**
  - 是一群**Decision Problem**的集合。當 **Q** $\in$ **NP-hard** 若且唯若所有屬於**NP**的**Decision Problem Q'** ($\forall Q' \in NP$) 皆可多項式時間轉化成**Q** ($Q' \propto Q$)
  - 此類問題至今仍未找到一個多項式複雜度的決定性演算法，且一般相信沒有多項式複雜度的決定性演算法存在。
- **NP-complete:**
  - 是一群**Decision Problem**的集合。 若某一個問題**Q**屬於**NP-complete**，則滿足以下兩個條件:
    - **Q**屬於**NP**
    - **Q**屬於**NP-hard**

- 一般而言，理論學家相信上述問題之集合圖示如下:



- **NP-hard**與**NP-complete**的關係：
  - 所有**NP-complete**問題都是**NP-hard**問題 **(**如：旅行推銷員問題**)**；但是**NP-hard**問題不見得是**NP-complete**問題 **(**如：程式停止問題，它不是**NP**問題**)**
  - **NP-complete** $\subseteq$ **NP-hard**
  - 如果有一個**NP-hard**問題能夠找到多項式複雜度的決定性演算法，則所有**NP-complete**問題也都存在多項式複雜度的決定性演算法。

Summary

- Nearly all of the decision problems are NP problems.
- In NP problems, there are some problems which have **polynomial algorithms**. They are called **P problems**.
  - **Every P problem must be an NP problem.**
- There are a large set of problems which, up to now, have **no polynomial algorithms**.

- Some important properties of **NP-complete problems**:
  - Up to now, no NP-complete problem has any worst case polynomial algorithm.
  - If any NP-complete problem can be solved in polynomial time, **NP = P**.
  - If **the decision version of an optimization problem is NP-complete**, this **optimization problem** is called **NP-hard**.
- **We can conclude that all NP-complete and NP-hard problems must be difficult problems** because
  - They do not have polynomial algorithms at present.
  - It is quite unlikely that they can have polynomial algorithms in the future.

如何証明某個問題為NP-complete問題

- 証明某問題為NPC的理由：
  - 所有的NP-complete問題都是具有相關性的。因此，若有一個NP-complete問題能夠找到多項式複雜度的決定性演算法來解它，若且唯若所有NP-complete問題也都存在多項式複雜度的決定性演算法。
- 証明方法：欲証明Q∈NPC，則有以下兩個步驟：

(簡單) ① 証明Q∈NP (can guess an answer, and check it in polynomial time)

(複雜) ② 找一個已知的NPC問題 Q' ，証明Q'∝Q

- ○ 存在一個函數f(x)，
- ○ 此函數f(x)為polynomial-time computable，
- ○ 該函數f(x)使得對所有x而言，x∈L1 若且唯若 f(x)∈L2 。

(L1為Q'問題的解集合；L2為Q問題的解集合)

為什麼經由前面兩步驟就可以証明Q∈NPC

- 理由：
  - 步驟 1 主要在說明Q是屬於**NP**問題
  - 步驟 2 主要在說明Q是屬於**NP-hard**問題
    - 因為**NP-complete**問題既是屬於**NP**問題、也是屬於**NP-hard**問題。而 **Q'** 為一個已知的**NP-complete**問題，因此 **Q'** 即是屬於**NP**問題，也屬於**NP-hard**問題。
    - 因為 **Q'** 有**NP-hard**問題的血統，因此我們可以得知所有的**NP**問題 $\propto$ **Q'**。
    - 由於 “所有的**NP**問題 $\propto$ **Q'** ”。若我們能夠証明出**Q'**$\propto$**Q**，根據遞移性，就可以得知所有**NP**問題 $\propto$ **Q** (∵所有**NP**問題 $\propto$ **Q'** $\propto$ **Q**)。因此 **Q** 即屬於**NP-hard**問題。
  - **Q**如果既是屬於**NP**問題，也是屬於**NP-hard**問題，則**Q**即屬於**NP-complete**問題。

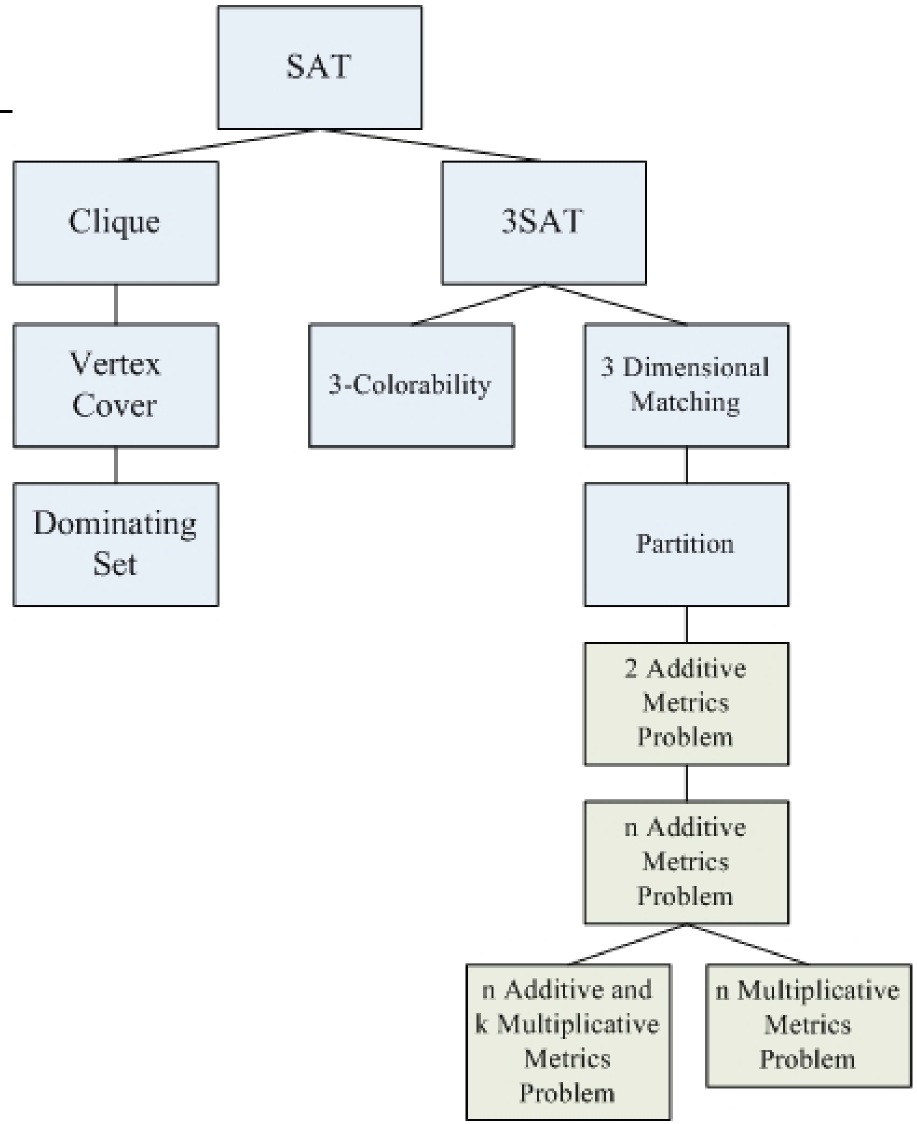
- 庫克定理 (Cook's Theorem)

SAT屬於NP-Complete

(SAT∈NPC)

  - 全世界第一個被很辛苦地証明出來的**NPC**問題
  - 往後的學者所証明出之**NPC**問題，皆是藉由該定理所陸續推導出來

- 後續**NPC**問題大致的推衍流程如右圖 (各家版本不一)。



Excerpt from "Introduction to Algorithms, UDI MANBER"
Figure 11.1

- 証明下列定理：
  - 3-SAT問題為NP-Complete
  - Clique (結黨) 問題為NP-Complete
  - Vertex-Cover (頂點覆蓋) 問題為NP-Complete
  - Dominating Set (支配集) 問題為NP-Complete

証明定理：3-SAT問題為NP-Complete

- 何謂3-SAT問題:
  - 為SAT問題的特定型態。給一個SAT函數，且此函數中每一個括號內皆恰有3個變數，則我們對存在於此函數E中的一些變數分別指派True或False，使這個函數結果為True。
  - Ex: Let $E = (-x_1 \vee x_2 \vee -x_3) \wedge (x_1 \vee -x_2 \vee -x_4) \wedge (-x_5 \vee x_2 \vee x_3)$. Then the following assignment will make E true and the answer will be "yes".

$$x_1 \leftarrow T,\ x_2 \leftarrow T,\ x_3 \leftarrow T,\ x_4 \leftarrow F,\ x_5 \leftarrow F$$

- 【証明方法】欲証明$Q \in NPC$，則：
  - 証明$Q \in NP$ (can guess an answer, and check it in polynomial time)
  - 找一個已知的NPC問題 Q'，証明$Q' \propto Q$
    - 存在一個函數f(x)，
    - 此函數f(x)為polynomial-time computable，
    - 該函數f(x)使得對所有x而言，$x \in L1$ 若且唯若 $f(x) \in L2$ 。

證明Q∈NP

- **Can guess an answer, and check it in polynomial time**
- 給一個布林函數，但限制該函數的每一個括號內恰含3個變數。
- 由以下的非決定性演算法得知此問題在猜一組解答僅需**O(n)**，而驗証解答僅需常數時間**O(1)**。由於可在**Polynomial Time**的複雜度內很快地驗證出解答是否正確，故得知此問題為**NP**問題。

```
/* Guess */
  for i = 1 to n do
      xi ← choice(true, false);
/* Verification */
  if E(x1, x2, … ,xn) is true then
      success;
  else failure.
```

証明Q'∝Q

- SAT問題是一個已知的NPC problem (by 庫克定理)。因此，我們令它為Q'，而待驗証的3-SAT問題為Q。
- 証明Q'∝Q即是說明下列三個事項:
  - 是否存在一個函數f(x)，可以將Q'的問題型態轉換成Q的問題型態
  - 此函數f(x)是否為polynomial-time computable，
  - 該函數f(x)是否對所有x而言，$x \in L1 \Leftrightarrow f(x) \in L2$
    (L1為Q'問題的解集合；L2為Q問題的解集合)

- 【說明事項 1】是否存在一個函數f(x)，可以將Q'的問題型態轉換成Q的問題型態。
  - 給定一個SAT問題的布林函數E，試著將此函數 E 轉換成每個括號內的變數個數均恰有三個之新的布林函數 E'，且此新函數 E' 不能變更原函數 E 的邏輯意涵。若能成功轉換，則表示 Q' 與 Q 之間確實存在一個函數 f(x)。

變數個數	函數 E 括號內情況	轉換後之函數 E' 括號情況
1	(x_1)	$(x_1 \vee y_1 \vee y_2) \wedge (x_1 \vee \bar{y}_1 \vee y_2) \wedge (x_1 \vee y_1 \vee \bar{y}_2) \wedge (x_1 \vee \bar{y}_1 \vee \bar{y}_2)$
2	$(x_1 \vee x_2)$	$(x_1 \vee x_2 \vee y_1) \wedge (x_1 \vee x_2 \vee \bar{y}_1)$
3	$(x_1 \vee x_2 \vee x_3)$	不需做轉換
多於3	$(x_1 \vee x_2 \vee \ldots \vee x_k)$	$(x_1 \vee x_2 \vee y_1) \wedge (\bar{y}_1 \vee x_3 \vee y_2) \wedge (\bar{y}_2 \vee x_4 \vee y_3) \wedge \ldots \wedge$ $(\bar{y}_{k-4} \vee x_{k-2} \vee y_{k-3}) \wedge (\bar{y}_{k-3} \vee x_{k-1} \vee x_k)$

  - 由上表得知，Q'與Q之間確實存在一個函數f(x)

- 【說明事項 2】此函數f(x)是否為polynomial-time computable。
  - 假設一個 SAT 問題的布林函數 E 有 n 個括號，且每個括號中最多有 k 個變數，則函數 E 轉換成 3-SAT 問題之布林函數 E' 所花費的時間複雜度最多只需要 O(nk)。
  - 在SAT問題的布林函數 E 中，某一個括號內的變數：
    - 若只有一個或兩個變數時，則所需的轉換時間為常數時間 (∵轉換過程固定不變)。
    - 若有三個變數時，則不需要轉換時間，轉換時間為 0 。
    - 若有多於三個變數 (即：k > 3) 時，則需要的轉換時間函數 k-2，時間複雜度為 O(k)。
  - 由於可能有n個括號，因此所需要花費在轉換上的時間複雜度最多為 O(nk)
  - 由上述說明，可得知此函數f(x)為polynomial-time computable。

- 【說明事項 3】此函數f(x)使得對所有x而言，x∈L1 若且唯若 f(x)∈L2 。
  (註：L1為Q'問題的解集合；L2為Q問題的解集合)
  - 以本題來說，要証明 "有一組解可使SAT問題的布林函數 E 為True ⇔ 有一組解可使 3-SAT 問題的布林函數 E' 為True"
  (E is satisfiable ⇔ E′ is satisfiable)

① 【先証 E is satisfiable ⇒ E′ is satisfiable】

E is True
- 若要讓一個SAT問題的布林函數 E 為True，則每一個括號均要為True。
- 若要讓一個括號為True，則此括號中至少要有一個變數為True。

⇓

E' is True
- 若有一組可讓原SAT問題之布林函數 E 為True的解，也能使轉換後的3-SAT問題之布林函數 E' 為True，則該組解能讓 E' 中的每一個括號均能夠為True。
- 將原本在SAT問題內為True之 xi 變數所在的括號內的 yi 變數設成False；原本在SAT問題內為False之變數所在的括號內的 yi 變數設成True，則此組解可使 E'為True

② 【再証 E is satisfiable ⇐ E′ is satisfiable】

E' is True

- 若要讓一個**3-SAT**問題的布林函數 **E'** 為**True**，則每一個括號均要為**True**。
- 若要讓一個括號為**True**，則去除掉 **yi** 變數不看，此括號中至少要有一個**xi**變數為**True**。

⇓

E is True

- 若有一組可讓原**3-SAT**問題之布林函數 **E'** 為**True**的解，也必能使**SAT**問題之布林函數 **E** 為**True**。

- 由 ① 與 ② 的証明，可以得知 "有一組解可使**SAT**問題的布林函數 **E** 滿足 ⇔ 有一組解可使 **3-SAT** 問題的布林函數 **E'** 滿足"

- 由前面一系列說明，我們可以得知 **SAT ∝ 3-SAT** (∵Q' ∝ Q)。
- 更進一步地，我們可以得知 "**3-SAT**問題 ∈ **NP-Complete Problem**"

說明範例

- An instance E in SAT :

$(x_1 \vee x_2) \wedge (-x_3) \wedge (x_1 \vee -x_2 \vee x_3 \vee -x_4 \vee x_5 \vee x_6)$

$x_1 \vee x_2$

$-x_3$

$x_1 \vee -x_2 \vee x_3 \vee -x_4 \vee x_5 \vee x_6$

- The instance E′ in 3-SAT :

$x_1 \vee x_2 \vee y_1$

$x_1 \vee x_2 \vee -y_1$

$-x_3 \vee y_2 \vee y_3$

$-x_3 \vee -y_2 \vee y_3$

$-x_3 \vee y_2 \vee -y_3$

$-x_3 \vee -y_2 \vee -y_3$

$x_1 \vee -x_2 \vee y_4$

$-y_4 \vee x_3 \vee y_5$

$-y_5 \vee -x_4 \vee y_6$

$-y_6 \vee x_5 \vee x_6$

SAT E $\xleftrightarrow[\text{若且唯若}]{f(x)}$ 3-SAT E′

近似演算法 (Approximation Algorithm)

- 一個問題Q若經由上述証明方式，得知其屬於NP-complete問題，則代表此問題目前尚無有效率的演算法可以解決(即：無法在Polynomial Time內解決)。
- 然而，某些屬於NP-complete的問題卻常常出現在各種領域!!若我們可退而求其次，去找尋一個近似解而非最佳解的話，則能夠預期以有效率的方式解決此問題。此即Approximation Algorithm的精神。
- 設計一個近似演算法需注意的Issue:
  - 近似演算法的時間複雜度要很低 (至少要為Polynomial Time)
  - 需保証近似演算法所求出的解也是該問題的可行解
  - 在最差的情況下，用近似演算法所求出之近似可行解有多靠近最佳解

Approximation Ratio

- **Approximation Ratio**用來定義所求出之可行解有 “多靠近” 最佳解：
  - 對於某個問題而言，在給定輸入為**x**之情況下，令其最佳解為**Opt(x)**，而利用近似演算法**A**所求出的解為**A(x)**。若此近似演算法**A**為**ε-approximation**，則滿足：

$$\max\left(\frac{|A(x)|}{|Opt(x)|}, \frac{|Opt(x)|}{|A(x)|}\right) \le \varepsilon, \text{ for all } x.$$

    其中，**ε**稱為近似演算法的**Approximation Ratio**。
  - 某問題的近似解，與最佳解之間的差距不會超過 **(**或低於**) ε**倍

- 上述定義同時針對最大化和最小化問題之解做考量。
  - 若是最小化問題，則 $\frac{|A(x)|}{|Opt(x)|}$ 會比 $\frac{|Opt(x)|}{|A(x)|}$ 大，因此 $\frac{|A(x)|}{|Opt(x)|}$ 為該問題之 **Approximation ratio**。
  - 例如：**TSP**最佳化問題為**NP-complete**問題。課本上提出兩個解該問題的近似演算法。

$\frac{|A(x)|}{|Opt(x)|} \le \varepsilon$
- **[Algo. 1]** 此近似演算法的解與最佳解之差距 (定理9.6)：**minapprox < 2 × mindist**
- **[Algo. 2]** 此近似演算法的解與最佳解之差距 (定理9.7)：**minapprox2 < 1.5 × mindist**

  - 若是最大化問題，則前項公式的 $\frac{|Opt(x)|}{|A(x)|}$ 會比 $\frac{|A(x)|}{|Opt(x)|}$ 大，因此 $\frac{|Opt(x)|}{|A(x)|}$ 為該問題之**Approximation ratio**。即：

$$\frac{|Opt(x)|}{|A(x)|} \le \varepsilon$$