

前言

身為一個就讀資訊系又熱愛數學的學生，在大學探索的這幾年中，我深深地被『計算理論』這門學科吸引。簡單來說，計算理論的核心目標是透過數量化的方法，來分析「計算」這一件事情。乍聽之下，「計算」這件事情有點機械化，讓人聯想到電腦或是快速運轉的大型機器，但是如果仔細觀察，「計算」其實發生在日常生活中的各個角落。任何牽扯到需要量化分析、做出判斷決策的事情，幾乎都可以被納入廣義的「計算」概念中。而計算理論想要做的事情，就是要盡可能精準地用數學模型描述這些計算的抽象概念，試圖分析其極限，甚至複製其計算的能力，希望能夠最終對於現實的生活帶來更多的便利性。

§ NTU自主學習者計畫

由於在學校缺乏相關課程的資源，因此我大多是透過網路上國外課程的教材，自學計算理論的基礎知識以及一些比較近代的發展和應用。然而，在僅靠著自我要求，且同時還有不少外務的情況之下，這樣的學習總是斷斷續續、成效不彰。因此在看到了這次台大教學發展中心主辦的自主學習者計畫，讓我起心動念可以將平常純粹自我要求的學習，透過事先計畫以及

定期追蹤的方式，更有系統性地進行，也希望可以達到比較好的學習成果。

很幸運地，最後我以「探索人類及自學的極限」這個主題，入選了自主學習者計畫，要在八個月的期間，完成事先提案規劃好的自學計畫。

計畫分為兩大主軸，一個是專業課程的學習，預計研讀哈佛大學 Salil Vadhan 教授的 Pseudorandomness 教材以及麻省理工學院 Dana Moshkovitz 教授的 Advanced Complexity Theory 課程。兩個課程的部分內容我以前曾經試圖自學過，但是一直沒有完整地從頭讀完一遍。再加上沒有實際的練習題目，於是對於內容的掌握程度不高。因此，在這個自學計畫中，我也嘗試著跟隨閱讀的進度寫習題。最後雖然完成度大約只有一半，但是實際操作一遍，真的會對於內容有更深入的認識。

計畫的另一個方向，則是定期的追蹤我自學的狀況，希望可以更清楚地瞭解自己在自學時會遇到的困難，以及造成學習不順利的原因，並且嘗試在發現問題後及時解決。於是我架構了一個部落格（www.cnychou.tw），在上面記錄了每一週的學習回顧心得，以及一些閱讀完課程的筆記。

§ 《探索計算的極限》

在計畫進行的第一個階段，我撰寫了十幾篇的英文筆記，放在部落格上面，一方面是作為自己未來複習參考使用，也期待可以分享給有興趣學習的人。不過在第二階段期初的一次聚會，看到其他計畫的夥伴選擇比較平易近人的方式，例如影片、圖文並茂的文章等等，來介紹各自的專業領域。看到他們豐富的成果，讓我不禁覺得自己雖然做了不少的事情，也試圖想要和其他人分享，但是做成了英文筆記，對於我自己來說雖可以學到不少，可是對於外人想要閱讀的吸引力不大。畢竟網路上有這麼多國外學校的資源，何必要看我這個大學生打出來破破爛爛的英文呢？

於是在二月初，我改變了原先計畫的學習記錄方式，著手規劃撰寫一系列的中文計算理論學習資源，希望可以用平易近人的方式，吸引潛在會對這方面有興趣的學子，有個容易的管道，接觸和學習計算理論。最後將這個寫作主題訂為《探索計算的極限》，預計分為兩大部分：基礎知識及專題介紹。三個月下來，目前完成了基礎知識部分的 15 篇文章。剩下專題介紹的部分，計畫在畢業後定期的撰寫，希望可以累積到一定的規模。

§ 心得及感謝

這八個月下來，看著原本預計的學習進度慢慢完成，部落格上面的心得記錄累積到二十多篇，專題文章也一篇篇的完成，實在是令我感到很興奮。自主學習者計畫讓我下定了決心有系統性的進行自學，再搭配定期的自我檢討和改進，即使大四這一年充滿了各式各樣繁雜的事務，我仍然能夠擠出時間，享受學習的樂趣。更可貴的是，在過程中我更加認識自己學習的優缺點，並且養成學習記錄、檢討、規劃時間的好習慣。

最後，十分感謝台大教學發展中心的石美倫老師和李建模老師，在計畫中給了我們很多建議和鼓勵，並且提供大量的資源做各種嘗試。也很感謝六位好友：李威承、張庭瑋、周紀愷、謝萱、蔡采純、侯宗誠，幫忙試讀文章，提供我非常多寶貴的經驗，讓我更知道該如何用文字傳遞數理知識。也謝謝老同學王千華，在畢業前忙碌之際，幫忙繪製封面及插圖，在討論聊天的過程中，也讓我開始思考一些結合藝術和理論知識的可能。

周紀寧，台北，2016春

推薦序

目錄

前言	2
推薦序	6
目錄	7
<u>探索計算的極限</u>	<u>9</u>
導讀	10
<u>第 1 章 把自己當成電腦來思考</u>	<u>13</u>
1.1 用位元來思考	14
1.1*基礎的數學背景	23
1.2 如何定義問題	34
1.2* 計算理論的好朋友：圖論	43
1.3 如何解決問題：演算法	54
1.3* 淺談時間複雜度	64
<u>第 2 章 計算模型</u>	<u>77</u>

2.1 計算模型的血淚史	78
2.2 圖靈機	90
2.2* 不可決定性問題：停機問題	102
2.3 其他計算模型	114
第 3 章 複雜度動物園	125
3.1 複雜度類別	126
3.1*確定性與不確定性	142
3.2 相對化	151
3.3 簡約方法與完全性	164
3.3* 簡約方法實戰演練	177
<u>主動學習者計畫 《探索人類與自學的極限》</u>	<u>190</u>
每週心得回顧	190

探索計算的極限

導讀

人類和其他生物最不一樣的地方就是在於我們擁有高等的思考能力，然而這些思考能力的來源是什麼？這些能力是透過什麼樣的方式實踐出來的？我們可以具體的把這些能力抽象出來，並且試圖理解和分析人類智能的極限嗎？計算理論就是在做這件事情。我們好奇到底為什麼會有思考能力上面的差距，也很想知道哪些問題是我們人類很難處理的，更重要的是，我們希望將自身的能力複製，抽象成計算的模型，如此一來就可以徹底地將能力發揮。

從二十世紀開始，科學和數學的發展日益成熟，當時的頂尖學者開始思考是否可以系統性且機械性的將人類的解決問題能力運用在每個地方上面。幾十年下來，許許多多成功和失敗的故事讓我們更加了解自身能力的範圍在哪，而伴隨出現的重要產物：電腦，更是改變了人類的發展史，徹底影響人與人以及人與自然的關係。然而最重要的問題仍然是懸而未解的，我們仍然不知道到底人類為何具有智能，也不知道自己能力的極限在哪裡，會不會有些事情是我們天生就做不到的？

計算理論就是一門純然使用數學來分析智能的學科，從理論還有實務的觀點同時出發，試圖用理性的思考、抽象的數學

模型、量化的分析甚至物理性的限制，想要回答身為人類最想知道的根本問題。

在這一系列的專題文章中，我將會帶領大家一窺計算理論的面貌。由於關於計算理論的資源大多是英文的，而且需要大量的專業背景，使得一般的中文讀者比較無法親近，也造成計算理論在國內並不受到關注。我希望能夠在這系列的文章中，用既平易近人又不失專業的風格，讓對於數學、資訊、理論還有哲學的朋友，能夠認識在理論資訊中一些很有趣的理論和結果。

內容將會分為兩大部分：基礎的背景知識和專題介紹。

在基礎的背景部分，我們將先引入一些基本的符號以及思考方式，接著從歷史發展的角度帶領讀者看看計算理論的領域是如何從無到有，發展到現在的樣子。最後，一些常見的定義和名詞將會被仔細解說，讓即使沒有任何背景的人，都可以在下一個專業部分有足夠的知識。

而在專題介紹的部分，將會深入淺出的討論各個近十幾年來發展迅速的新興領域，例如：近代密碼學、量子計算、隨機計算、互動式證明、平方和等等，主要的目的是希望讓讀者可

以在兩三篇文章中了解那個主題的核心概念，知道當初的發明人是想要解決什麼問題，後來又是如何解決的。

那就話不多說了，讓我們一起探索計算的極限吧！

基礎背景

第 1 章 把自己當成電腦來思考

1.1 用位元來思考

內容：位元(bit)

困難度：★☆☆☆☆

對於一般接觸資訊領域不深的朋友，一被問到電腦的原理是什麼，大概都會搔搔頭然後說：「就是一堆的0和1組成的吧！？」。這個回答完全正確，不過我們的好奇心可不能就這麼打住，為什麼只靠著0和1電腦就有這麼強大的力量可以幫我們處理這麼多事情，解決這麼多問題呢？

這一切就要從0和1代表的意義開始說起。在接下來的篇幅中，我將要說服你，我們可以用0和1做很多事情！

§ 電腦是如何使用0/1？

在物理學中的基本粒子可能是夸克、輕子、玻色子等等，在社會學中最小的單位是人，而在資訊科學中，最小的單位則是「位元(bit)」。位元就是一個0或是1的二進位數字，當他只有一個的時候，並不能做太多的事情，就像其他基本單位在其領域中能做的事情不多一樣。然而當我們把許多位元擺在一起，能做的事情就變得很多了。

基本上來說，電腦能用位元來做兩大類的事情：**當成數字來做計算、當成儲存的單位**。這兩種看位元的方法其實在本質上是一樣的概念，但是因為在操作上的意義不太一樣，於是特別分開討論。

§ 把位元當作數字來計算：二進位

平時我們習慣的數字操作都是由十進位組成的，像是我今天晚餐花了有一百三十五元，寫成阿拉伯數字：135，就是有 1 個 100、3 個 10 和 5 個 1。十進位表示方法利用 10 的冪次(也就是 1,10,100,1000,10000,...)乘上 0-9 之間的數字後加起來，完備的定義了我們日常需要用到的整數系統。

相較於十進位，其實二進位簡單多了，在二進位的世界只有 0 和 1，我們改成利用 2 的冪次(也就是 1,2,4,8,16,...)乘上 0 或 1 後加起來，同樣成功地將所有的整數表達清楚。在下面的表格中，簡單地舉了幾個 10 進位和 2 進位之間的轉換關係。

二進位	10	101	1010	1111	10010
十進位	2	5	10	15	?

你知道上面問號中的數字應該是多少嗎？沒錯，就是 18。

§ 把位元當成儲存的單位：編碼

當我們知道一堆的0和1其實可以拿來當成一般的整數來使用後，好玩的事情就開始了。只要是任何有限的玩意兒，我們都可以把它編碼成0/1組成的字串！於是一切我們平常可以理解的任何東西，都可以轉化成位元的表示，儲存在電腦中，甚至被電腦程式所理解。這樣說應該非常抽象，讓我們來看些實際的例子吧！

假設你是一家蛋塔店的老闆，你們賣的蛋塔有多達二十多種，每一種的名字都非常特別，例如：椰香幾何蛋塔、起司代數蛋塔、咖哩圖論蛋塔等等，雖然這些有趣的名字十分受到客人的歡迎，但是對於管理帳務的店員來說非常的痛苦，每筆訂單中的餐點名稱都太複雜，很容易搞混，使得必須花很多時間處理。

於是一個很直覺的方法就是幫這些蛋塔取簡化的代號！例如用英文字母來表示，或是取原本名稱的字首兩個字等等，許多餐廳也是用這樣的方式來管理菜單的。而現在，我們也可以用數字來進行編碼，把每一個口味對應到一個數字，如此一來，在管理的時候，就等於對著這些編碼後的數字做處理。雖然直接看這些數字不像原本的名字一樣有意義，但是經過還原

之後，就可以知道原本代表的東西是什麼，完全不會犧牲掉任何的資訊。

§ 電腦是如何在位元上面做運算？

當我們把位元看成數字的時候，位元們就成為了數字的載具，像是我們平常熟悉的0-9一樣，讓我們可以直接操作加減乘除，只不過變成用二進位表示罷了。

而一旦我們把位元當作儲存的空間時，就需要制定出一種特別的編碼系統，如此一來每台不同的電腦才能順利的編碼和解碼出原本想要儲存的內容。最常見的位元編碼系統即是 [ASCII](#)(美國資訊交換標準代碼)，這個編碼系統每次使用八個位元為一個單位，於是可以處理二的八次方(=256)中不同的狀態。下表是一些實際的例子，讓大家體會看看用位元儲存東西的感覺。

位元表示	0100 0001	0010 0100	0010 1010
儲存內容	A	\$	*

也許大家會覺得奇怪，為什麼0010 0100就是代表\$呢？這是因為ASCII他是一個大家公認的標準，所以為什麼要把某個符號用特定的二進位數字表示，其實都是當初的人心情好才這

樣決定的！(當然我相信他們應該是有自己的理由，不過應該不太重要)

現在我們知道了可以利用位元做數值的運算，也可以當作儲存的空間，那麼接下來就讓我們來多認識一些關於位元的性質吧！

首先回到蛋塔的問題，假如現在總共有24種口味的蛋塔，那麼我們會需要用幾個位元來儲存每一筆訂單的蛋塔類別呢？要回答這個問題，我們可以從另外一個角度來想，如果現在有 n 個位元，可以儲存幾種不同的東西？如果我們可以找到一個最小的 n^* 使得 n^* 個位元可以儲存的種類超過9，那麼我們就可以拿這 n^* 個位元來儲存蛋塔的類別了！於是，這就變成一個簡單的指對數問題了。

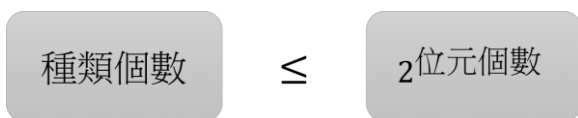

$$\text{種類個數} \leq 2^{\text{位元個數}}$$

圖 1-1：如何決定位元個數。

經過簡單計算後，我們將會知道如果有 n 個位元，最多可以儲存 2^n 種不同的東西。(想一想為什麼？)因此當我們

想要存 m 種東西時，會希望找到最小的 n^* 使得 $m \leq 2^{n^*}$ 。而其實這個 n^* 就會是 $\lfloor \log_2 m \rfloor$ 。(想一想為什麼？)

§ 二進位和其他進位的差別？

在這一個章節，我將要說服大家二進位的位元和其他進位的儲存方式只有差常數倍的差距，因此電腦使用位元來當作最小的單位是夠好的！

讓我們從簡單的比較開始：二進位vs.三進位。當我們想要儲存 m 種東西時，會需要幾個儲存單位？從上一個章節末端的分析，我們知道分別總共需要 $\lfloor \log_2 m \rfloor$ 個位元還有 $\lfloor \log_3 m \rfloor$ 個三進位的基本元素。這兩個數字看起來不太一樣，其實透過對數的基本性質推導過後($\log_2 m = \log_2 3 \log_3 m$)，我們可以發現所需要的位元個數大約是所需要的三進位基本元素個數的 $\log_2 3$ 倍！而如果拿某個 k 進位的方式來和二進位比較，所需要的元素個數只會差 $\log_2 k$ 倍。也就是說，當 k 是個常數時，這樣的差距就只是個常數，對我們來說影響不大。

而真正吸引資訊科學家使用二進位當作電腦的基本單位的原因，可以有兩個方面：

1. 實作方面：當我們使用 k 進位時，就會需要 k 種不同的狀態來區分 k 種不同的數值。在現實世界中，很容易找到擁有

兩種不同狀態的物質，例如用電壓的高與低分別代表1和0。我們可以發現，表示兩種不同的狀態相較於其他是最容易的。

2. 平均空間：在真正的實用層面，我們不太可能剛剛好將所有的空間百分之百的利用，延續蛋塔訂單的例子，如果現在有五種蛋塔，那麼用三個二進位數字表示的缺點就是浪費了三個不可能出現的空間。然而我們可以發現，這樣的浪費和其他進位比較起來算是客氣多了。如果原本我們選的是四進位數字，那麼現在我們就會浪費 $16-5=11$ 個空間了。

§ 結語

在這一篇介紹二進位和0/1編碼的文章中，我們對於電腦底層的運作方式有了一些基本的了解。簡單來說，電腦就是用一堆的0/1將所有的訊息編碼起來，例如我們使用的文字(A,B,C,...)或是數字(1,2,3,...)甚至一些特殊符號(+,*,&,...)，這些東西在電腦裡面都有唯一的表示方法，而且一旦使用了二進位，除了省空間之外(其實只要不是一進位，其他進位方式都很省空間，差別只有在常數的可接受範圍內)，在實作方便也容易很多(可以用電位的高低來代表0或是1)。

在接下來的幾篇文章，我們將要來看看從資訊理論的角度是怎麼樣來利用0/1來定義問題還有解決問題。我們將會發現，

透過這樣看似簡單的方法，除了方便儲存運算之外，竟然也可以幫助我們量化問題的大小、難度等等。

本章小總結

- 資訊科學中最基本的單位是「位元」
- 電腦利用位元當作計算以及儲存東西的載具

1.1*基礎的數學背景

內容：集合(Set)、卡式積(Cartesian product)、幕集(Power set)、
可數無限(Countably infinite)

困難度：★★☆☆☆

在計算理論的領域當中我們會大量的使用有關於**集合**的數學工具，像是在〈用0與1來思考〉一文中，講解二進位數字可以表達的可能種類數量時，就用到了一些些相關的概念。為了幫助比較沒有背景的讀者在未來能夠熟悉經常使用的語法，讓我在這篇附錄中為大家做一個簡單的介紹。

接下來我們會分成三個部分：集合的基本定義與性質、卡式積(Cartesian product)與幕集(Product)、無限。建議讀者可以根據自己的背景知識選擇不熟的段落閱讀。

§ 集合(Set)

集合是最單純的數學物件，基本上一切的東西我們都可以看成是集合，其他的數學概念都是再加了一些條件和規則之後才演變出來的。正式一點來說，我們可以這樣子定義：

定義 1-1 (集合, *Set*)：集合就是一個搜集了相異物品的數學物件。

舉例來說：班級是一個集合，裡面的物品就是每一個學生；正整數也是一個集合，裡面的物品就是1,2,3,...這些...正整數；有趣的是，一個什麼都沒有的東西，本身也是個集合，我們稱之為**空集合(empty set)**，符號用 \emptyset 表示。很重要的一點在於，集合裡面的元素一定都是**相異**的，如果你放了兩個相同的東西進去，從集合的觀點來看，是會被視為同一個東西！於是當我們要問一個集合的大小時，只會算裡面相異元素的個數是多少，而在符號上我們定義集合 S 的大小 (Cardinality) 為 $|S|$ 。

而如果我們從一個集合中拿了一部分出來，這個小部分我們就稱為是原本集合的**子集合(subset)**。在符號上，我們用 $S \subseteq A$ 來表示集合 S 是集合 A 的一個子集合。要特別注意的是，一個集合也會是自己的子集合。而如果一個子集合不等於原本的集合，則我們稱它為**真子集(proper subset)**。此時，子集合的符號可以改寫成： \subset 。

集合與集合之間可以有互動，在這邊我們介紹兩個最基本的互動：聯集與交集。集合 A 和集合 B 的**聯集(union)**就是把

A 和 B 的所有東西收集起來得到的新集合，用符號表示為 $A \cup B$ 。 A 和 B 的**交集(intersection)**則是把 A 和 B 共同有的東西收集起來，用符號表示為 $A \cap B$ 。

通常我們在定義一個集合的時候會這樣寫：

$$S = \{s \in \Omega \mid s \text{ has property } P\}$$

意思就是說集合 S 的元素是來自於 Ω 這個集合中符合性質 P 的元素。舉例來說，當我們用 \mathbb{Z} 表示整數的集合時，包含所有偶數的集合Even就會是：

$$\text{Even} = \{x \in \mathbb{Z} \mid x \text{ is even}\}$$

一個小小的概念補充，集合本身也可以成為另外一個集合的元素！例如：包含所有 \mathbb{Z} 中大小為3的子集合的集合(= $\{S \subset \mathbb{Z} \mid |S| = 3\}$)。

§ 卡式積(Cartesian product)、冪集(Power set)

除了上面兩種最基本的集合操作之外，還有一些進階的操作是根據實際的需求產生的。在這邊我要介紹卡式積(Cartesian product)和冪集(Power set)。

卡式積

想像當我們要**同時**使用兩個集合 A 和 B 時，該如何用集合的語言表示呢？這時卡式積就幫了一個大忙，他讓我們能

夠將集合 A 和集合 B 並列的使用，同時維持了原本集合的性質。

定義 1-2 (卡式積, Cartesian product) : 集合 A 和 B 的卡式積是 $A \times B = \{(a, b) | a \in A, b \in B\}$ 。

從定義中我們可以看出來，卡式積把 A 和 B 的元素透過 $()$ 還有,並列的表示，如此一來達到同時使用的效果！這種表示方法又會被稱為多元組(tuple)。

舉例來說，我們現在有一個男生的集合 Boy 和女生的集合 Girl ，我們的目標是要研究男生和女生之間的配對，然而此時我們到底該在哪個幾何上面做事情呢？有了卡式積之後，我們可以方便地直接在 $\text{Boy} \times \text{Girl}$ 上面研究所有可能的配對！

此外，我們還可以把超過兩個集合做卡式積得到更大的合併集！而符號也是直接沿用，例如： $A \times B \times C \times D = \{(a, b, c, d) | a \in A, b \in B, c \in C, d \in D\}$ 。更好玩的是，我們還可以把一個集合對自己做卡式積！如果我把集合 A 對自己做卡式積 n 次，在符號上我們可以直接寫成 A^n 。會這樣寫的目的是因為整個動作就像是從集合 A 裡面拿東西拿了 n ，和排列組合中取後放回一樣，我們知道這樣可能的組合有 $|A|^n$

種，於是乾脆就用 A^n 這樣的記號來代表收集所有組合的集合。

更進一步，我們會用一個叫做克萊星(Kleene star)的符號來表示某個集合所有有限的卡式積。

定義 1-3 (克萊星, Kleene star) : 集合 A 的克萊星是

$$A^* = \bigcup_{n \in \mathbb{N}} A^n。$$

在直觀意義上，克萊星是一個包含所有**有限組合**的集合，以 $\{0,1\}^*$ 為例，所有0和1組成的**有限字串**，都會被搜集在 $\{0,1\}^*$ 裡面。於是和〈用位元來思考一文〉做結合，我們可以發現， $\{0,1\}^*$ 包含了所有電腦會處理的東西！（想一想，畢竟電腦處理的一定是有限的字串而不會是無限長）

冪集

而當我們現在有興趣的是一個集合**本身**的各種組合可能性時，改怎麼用集合的語言來表示呢？這時候冪集就該出場了！

定義 1-4 (冪集, Power set) : 集合 A 的冪集是 $2^A =$

$$\{S \subseteq A\}$$

從定義中就很明顯可以感受到幕集的意義，舉個簡單的例子：如果集合 A 是 $\{1,2,3\}$ ，那麼他的幕集就是 $2^A = \{\emptyset, \{1\}, \{2\}, \{3\}, \{1,2\}, \{1,3\}, \{2,3\}, \{1,2,3\}\}$ 。會使用這樣的符號是因為幕集的動作就像是針對每一個元素，決定要把它放入一個子集中還是不要放入，我們可以算出這樣的可能性有 $2^{|A|}$ 種，於是乾脆就用 2^A 來表示搜集這些可能性的組合。最後，回到我們喜歡的 $\{0,1\}^*$ 為例，他的幕集 $2^{\{0,1\}^*}$ 代表的意思是什麼呢？沒錯，就是所有可能的有限字串集合的集合！

§ 無限(Infinite)

在大學以前，大部分人認識的無限就是：「那個很大很大的東西」。然而，在現實上，無限並沒有那麼單純，舉個最簡單的例子，你知道無限還有分大小的嗎！？

不過這邊我不打算花太多的篇幅介紹所有關於無限的基本知識，雖然我其實還蠻想的，因為實在非常有趣。在這短短幾個段落中，我只會提供一些會和之後有關的背景知識，有興趣多瞭解的人可以看看這本我強烈推薦關於無限的書[1]。

可數無限(countably infinite)在是所有無限中最小的，我們把其他的無限種類稱為不可數無限(uncountably infinite)，從

數學上定義的大小關係中，不可數無限是嚴格的比可數無限還要大，然而有趣的是，在不可數無限之中，還有分非常多不同的無限類別，一個比一個還要大[2]。舉例來說，整數集合 \mathbb{N} 就是可數無限的，實數集合 \mathbb{R} 則是不可數無限的。

在正式介紹可數無限之前，要先給大家建立在無限中如何比較集合大小的觀念。

我們說一個集合 A 大於等於集合 B ，可以換個角度來想，就是當我每次都從兩個集合各自取出一個元素的時候，最後集合 B 不會比集合 A 還要慢被全部拿光。而這樣的概念可以用函數的方法來描述。如果我可以從集合 B 定義出一個單射(injective)的函數到 A 上面，則每當我從 B 把一個元素丟掉時，都可以相對應的把 A 裡面的一個元素丟掉，於是達到上面的直觀想法。

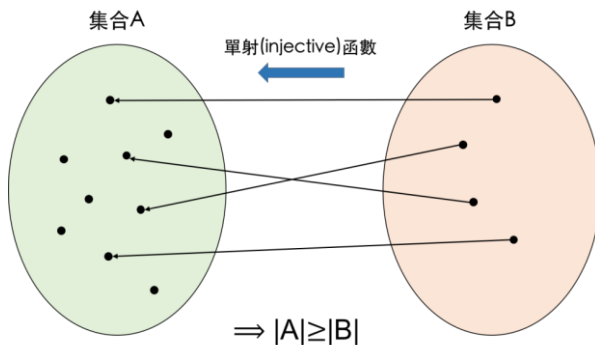


圖 1-2：集合的大小關係：集合A大於等於集合B若且唯若存在一個從B到A的單射函數。

而如果兩個集合的大小是一樣的，那就代表上面這個對應的過程，應該是雙向的，正式點的說法，就是在集合A和集合B之間存在一個**雙射**(bijective)函數。

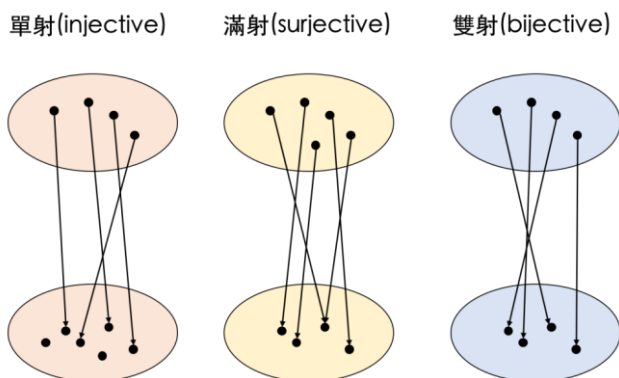


圖 1-3：函數的三種類型：單射、滿射、雙射。

可數無限的定義非常貼近直觀思維，任何可以被「數」的無限集合，都是大小可數無限的。讓我們從最基本的正整數集合 \mathbb{N} 開始， \mathbb{N} 的元素是 $\{1, 2, 3, \dots\}$ ，在研究 \mathbb{N} 的元素時，我們可以一個一個的「數」，而這個數的方法就是很直接地照著正整數大小順序 $1, 2, 3, \dots$ 這樣數下去。

看到這裡有沒有覺得和之前定義集合的大小關係有點相似？沒錯，其實上面數數的過程，就是個從正整數到另一個集合的**雙射**函數！重要的地方在於我們並不是要真的把集合內的所有元素數完，畢竟當兩個集合都是無限大的時候這對我們來說是做不到的。我們可以做的是去確定有一種數元素方法，使得我突然問你這個集合中第 n 個元素是誰的時候，你可以告訴我是哪一個元素。相反的角度也是一樣，當我隨便拿一個元素時，你也要有能力告訴我他的編號是多少。

接下來以整數 \mathbb{Z} 集合為例， \mathbb{Z} 的元素是 $\{\dots, -3, -2, -1, 0, 1, 2, 3, \dots\}$ 。和 \mathbb{N} 不同的是，他搜集了從負的無窮大到正的無窮大中所有的整數，這時候我們有辦法找到一個數 \mathbb{Z} 的辦法嗎？想一想再看看下圖的參考解答吧！

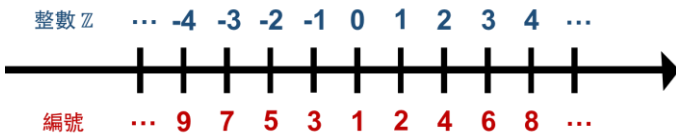


圖 1-4：整數 \mathbb{Z} 是可數的。

所以，只要一個無限集合能夠被數，那麼他的大小就是可數無限！

對於我們來說，有兩個關於可數無限的重要觀念：
 $\{0,1\}^*$ 是可數無限的、可數無限的集合是可以列表的
(enumerable)。前者的證明會需要花一些篇幅，所以在這邊
就省略，請讀者先接受這件事實。而這個事實重要的地方是
在於 $\{0,1\}^*$ 包含了所有電腦可以處理的字串，也就是說電腦能
夠處理的東西即使再多，都還是可數無限的！而為什麼可數
無限重要，就跟我想要強調的第二點有關，因為只要一個集
合是可數無限的，那麼他裡面的元素就可以被列(enumerate)
出來。就像是上面的例子中提到，你給我一個編號 n ，我可以
迅速地告訴你這個集合中第 n 個元素是什麼。這樣子可以被編
號的好性質，對於未來的分析有很大的幫助。

本章小總結

- 集合(Set)、子集(subset)、真子集(Proper subset)、聯集(Union)、交集(Intersection)
- 卡式積(Cartesian product)、克萊星(Kleene star)、冪集(Power set)
- 可數無限(Countably infinite)

參考資料

- [1] 《無限大的祕密：突破科學與想像極限的「無限」簡史》，John D. Barrow，臉譜出版社。
- [2] [Ordinal Number](#)，Wiki。

1.2 如何定義問題

內容：決定性問題(Decision problem)、質數問題(Prime)

困難度：★★☆☆☆

我們在〈用位元看問題〉中介紹了電腦使用位元編碼一切事物的概念，在接下來所有的討論中，我們都將會延續這樣的觀念，把一切平常我們習慣的語言、數字、運算等等都用抽象的位元來表示，當然我們不會直接把編碼後的結果放在你眼前，我相信應該沒有人可以使用正常的速度閱讀這樣的東西。重要的關鍵是在於我們心中要把一切也視為由0和1組成，才能站在電腦的角度思考。

在本篇文章中，我們就要來看看該如何使用位元的觀點來看待所謂的「問題」。我們將會對「問題」這個概念下一個數學的定義，並且更進一步定義「問問題的過程」，接著舉一個簡單的例子，給大家一些實際的感覺。

§ 決定性問題(Decision problem)

在這邊我們將直接對「問題」這個概念下一個特別的數學定義。在介紹之前，我必須先告訴讀者這並不是對於「問題」的唯一定義方式，有興趣的人可以參考延伸閱讀的文章

來瞭解為什麼我們要這樣定義「問題」以及一些其他的定義形式。

定義 1-5(決定性問題, Decision problem)： 令 $P \subseteq \{0,1\}^*$ 是一個決定性問題。則對於任何字串 $x \in \{0,1\}^*$ ，
 $x \in P$ 若且唯若 x 是問題 P 的一個答案。

看完**定義 1-5**相信對於沒有資訊或是數學背景的人來說腦中一定是充滿了問號，請大家耐著性子，讓我在接下來的篇幅，帶著各位從定義中字面的解讀開始，一一剖析這些文字和符號背後代表的概念。

1. 定義 1-5的文字解讀：從〈基礎的背景知識〉裡面，我們學到了 $\{0,1\}^*$ 所代表的意思是所有由0和1組成的有限長度字串。站在電腦的角度來思考，0和1組成的字串其實就是所有可能的編碼結果。換回人類的思維，也就是所有我們的文章、數字、證明等任何你想得到的東西。 $P \subseteq \{0,1\}^*$ 的意思是 P 是任何字串的子集合。或是換個角度， P 是一個搜集了某些字串的集合。再從人類的思維觀點來看，我們可以想成 P 搜集了這個「問題的解答」。

2. 定義 1-5的核心概念：一個決定性問題的數學定義，其實是定義為一個搜集所有解答的集合。重要的地方在於，在定

義的當下，我們並不見得知道問題 P 裡面到底包含了哪些字串（不然問題就被解決了）。

在定義完問題本身之後，我們還要用數學來描述問問題的過程。而這個過程其實就是兩個步驟：給定一個解答的候選人、確認這個解答候選人是不是真的解答。嚴謹的來說就是像下面這樣：

- 給定一個 $x \in \{0,1\}^*$ （解答候選人）。
- $x \in P$ 若且唯若 x 是問題 P 的答案。

§ 質數問題(Prime)

看了令人頭昏腦脹的定義1之後，讓我們來看看一個簡單的例子抓抓感覺吧！在這邊我要舉的例子是：質數問題(Prime)。從人類思維的觀點來解釋這個問題很簡單：「請問有哪些數是質數？」，那麼從電腦的觀點該如何定義質數問題呢？

$$PRIME = \{x \in \{0,1\}^* \mid x \text{ is a prime}\}$$

在這邊我們把質數問題定義成一個集合PRIME，在這個集合之中包含了所有質數的二進位編碼。舉例來說，7是一個質數，他的二進位編碼是111，因此我們就知道 $111 \in$

PRIME。而12不是一個質數，所以我們就知道 $1100 \notin \text{PRIME}$ 。

因此，一個完整的質數問題過程就是將想要問的數字編碼成二進位代碼 x ，然後確認 x 是否在PRIME裡面。

除了決定性問題之外，還有另外兩個常見的問題類型：搜尋問題(Search problem)和最佳化問題(Optimization problem)。在正式介紹他們之前，讓我們再用一個不同的角度來看看決定性問題(Decision problem)。

定義 1-6(決定性問題 - 函數型定義)：令 $P: \{0,1\}^* \rightarrow \{0,1\}$ 是一個決定性問題。則對於任何字串 $x \in \{0,1\}^*$ ， $P(x) = 1$ 若且唯若 x 是問題 P 的一個答案。

其實上面這個函數型的定義是和**定義 1-5**中集合型的定義等價，我們會交替的使用這兩種定義是在於決定性問題的答案只有兩種：正確或錯誤，因此我們既可以使用在集合內/外來表示結果，也可以使用函數的輸出0/1來代表結果。

那麼如果我們想要定義有不只一個結果的問題，該怎麼做呢？

§ 搜尋問題(Search problem)

讓我們從一個生活化的問題出發。假設你今天剛看懂一個新的定理，於是心情很好，想要買一個布朗尼來犒賞自己，剛好你的朋友是個甜點專家，手中有超過一萬筆的甜點介紹，請問你要怎麼從裡面找到和布朗尼有關的資訊呢？

做法非常簡單，就是把所有關於布朗尼的甜點資訊抓出來就對了。(這裡我們就先別管速度問題了)不過資料量實在太多了，於是我們想要請電腦來幫忙處理，然而如此一來我們就必須把問題用數學模型來描述，這樣電腦才看得懂！

這時候一個最直接的方法就是在這個擁有一萬筆甜點介紹的資料庫上建立一個函數(function) 甜點介紹(x)，這個函數的輸入(input)是一個我們想要搜尋的甜點，輸出(output)則是有提到這個甜點的介紹資訊。如此一來我們想要找布朗尼的問題就可以變成下面這件事情：

Q：有哪些甜點介紹會出現在 甜點介紹(“布朗尼”) 裡面？

舉例來說，如果甜點介紹73,93,和1729裡面出現了布朗尼，那麼我們就會知道73,93,1729 \in 甜點介紹(“布朗尼”)。所

以這個布朗尼問題就會變成是找尋所有在**甜點介紹**(“**布朗尼**”)裡面的甜點介紹！

那麼現在就讓我們從上面這個例子得到的概念轉換成數學的定義吧！

定義 1-7(搜尋問題, Search problem)：令 $P: \{0,1\}^* \rightarrow 2^{\{0,1\}^*}$ 是一個搜尋問題。若且唯若 y 是 x 的目標之一。

要理解上面這個定義，關鍵在於搞懂 $2^{\{0,1\}^*}$ 是什麼？ $2^{\{0,1\}^*}$ 是字串 $\{0,1\}^*$ 集的冪集(power set)，也就是搜集了所有字串集合的集合。在這邊的意思就是問題 P 會將輸入對應到一個字串的集合，裡面就是包含了對應的目標。

除了用上面這個複雜的函數定義之外，搜尋問題也可以用其他的方式來描述，例如用雙輸入的函數(2-input function)或是用關係代數(relational algebra)。

§ 最佳化問題(Optimization problem)

那如果當我們的問題會牽扯到和分數、價值有關的時候，又該怎麼樣來用數學描述呢？

這次讓我們從一個經典的電腦科學問題開始：旅行銷售員問題(Traveling Salesman Problem, TSP)。問題的輸入會是一個國家的地圖，裡面有很多個城市，然後我們會知道兩個有道路相連的城市之間所需要花的通車時間。這個問題的目標就是要找到一條旅行的方法，讓銷售員可以在最短的時間難走完所有的城市。

對於這個問題，我們可以把每種可能的繞行方法當作一個解答候選人，而這樣的一個繞行方法會對應到所需要花的總時數，我們的目標就是要找到擁有最短總時數的那個繞行方法。

讓我們試著利用數學來描述這個問題。整個問題可以用一個路徑時數的函數來表示：*路徑時數*(*map*, *route*)。這個函數將會有兩個輸入，分別是地圖和路徑，函數的輸入即是這條路徑在地圖上所要花的總時數。為了避免錯誤的輸入(有可能輸入的路徑沒有走完所有城市)，我們可以把錯誤輸入的函數值都設為無限大。如此一來我們的問題就變成：

TSP: 給定一個地圖*mymap*，請問是哪個路徑極小化函數*路徑時數*(*mymap*, *route*)?

將這樣的想法變成數學定義就會是：

定義 1-8(最佳化問題, Optimization problem) : 令 P 是一個定義在目標函數 $O: \{0,1\} \times \{0,1\} \rightarrow \mathbb{Q}$ 上的最佳化問題。則對於輸入 x ， P 的目標是找到 y 使得 $O(x, y)$ 被極大或是極小化。

在這裡使用有理數 \mathbb{Q} 當作分數的值域是因為一般的電腦基本上是無法處理實數的！

§ 結語

目前為止介紹的三種問題類型幾乎可以把生活中會遇到的所有問題都描述清楚，而在接下來的討論之中，我們就是要來分析當我們用這樣的數學語言來描述問題時，可不可以來量化問題的難度？來比較哪些問題簡單哪些問題難？

本章小總結

- 「問題」的數學定義：決定性問題
- 「問問題過程」的數學定義
- 質數問題(Prime)
- 搜尋問題(Search problem)
- 最佳化問題(Optimization problem)
- 旅行銷售員問題(TSP)

1.2* 計算理論的好朋友：圖論

內容：圖論(Graph Theory)

困難度：★★☆☆☆

圖論(Graph Theory)是應用數學與離散數學中一個重要的領域，除了一些純數學的重要發展之外，圖論在資訊界也被廣泛的應用。從直觀上來說，圖論把現實生活中的物件看作是一個一個單一的物體，然後在這些物體之間會有一些兩兩的關聯，而圖論的目標就是要分析在這樣的架構下衍生出來的性質。

舉例來說，我們把每個人想成是一個一個圖論中的物體，而兩兩之間的關係是代表這兩個人是不是好朋友。於是，使用這樣的定義，我們可以建構出一個代表朋友關係的圖(graph)，想要分析這群人之間的好友關係，將會等價於直接在這個朋友關係圖上面做分析。譬如說，我們想要知道這群人之中最多可以找到幾個人是兩兩都不是朋友的，這樣的問題可以等同於在朋友關係圖上面尋找最大的點集和使得這些點兩兩之間都沒有朋友連線！請參考圖 1-5中舉的例子。

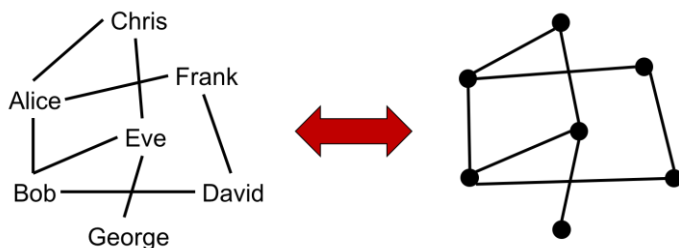


圖 1-5：從朋友關係圖到抽象的圖。

也許從上面的例子還感受不到圖論的功用，但是現在想像你要針對不同群的人們，來找解決特定的問題時，透過圖論抽象的功力，可以幫助你簡化問題，更容易處理。

§ 基礎定義

嚴格來說，在圖論中我們定義一個圖 G 是由兩個元件組成的，分別是點(vertex)和邊(edge)。所有圖 G 的點構成了點集(vertex set)，而一條邊就是從點集中選取兩個點作為端點，所有的邊則構成了邊集(edge set)。於是在描述一個圖 G 的時候，我們會用 $V(G)$ 和 $E(G)$ 來分別表示 G 的點集和邊集。

定義 1-9 (圖, graph)：一個圖 G 是由點集 $V(G)$ 和邊集 $E(G)$ 組成的。其中 $E(G)$ 的任意元素 e ，會是一個 $V(G)$ 的二元組，例如 $e = (x, y)$ ， $x, y \in V(G), x \neq y$ 。

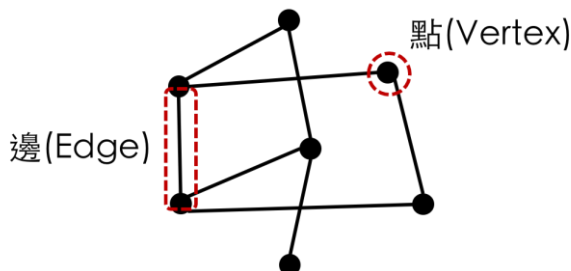


圖 1-6：一個圖(graph)是由點(vertex)和邊(edge)組成的。

這邊要特別強調，關於圖的定義其實有非常多神祕的延伸，例如：邊是有方向性的、兩個點之間可以有很多條邊、某個點可以有個邊連到自己、高維度的邊等等。每一種新的定義都會對應到一些有趣的應用，但是在這邊我們先把定義放在最簡單的情況，也就是邊是沒有方向性的、兩點之間只會有一條邊、邊的兩個端點是相異的。如果之後會使用到比較複雜的定義，會在特別指出。

在接下來的篇章，我們將會看到一些圖論在資訊領域有趣的問題，以下的例子將會延續最一開始朋友關係圖的例子，希望能夠在介紹抽象問題的同時，帶給讀者清楚的應用可能性。

§ 全部都是好朋友：點團(clique)

首先，讓我們設想一種情況，現在你是一個班級的導師，你想要指派班上一群默契最好的同學來幫忙準備期末的同樂會。你希望人手越多越好，同時又希望這些幫忙的同學互相都很熟絡，也就是說，兩兩都是好朋友，這時候你該怎麼做呢？這樣兩兩之間都是朋友，也就是點與點之間都有邊的集合，在圖論中被稱作一個點團(clique)。

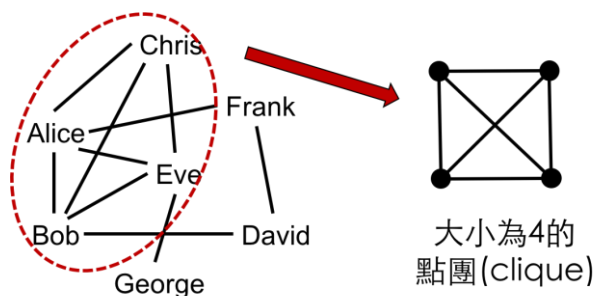


圖 1-7：點團的實例。在左邊的朋友關係圖中有四個人兩兩是朋友，分別為 Alice, Bob, Chris 和 Eve，將他們四人獨立出來看，會形成一個大小為 4 的點團(clique)。

於是，想要在朋友關係圖中尋找一個兩兩都是好朋友的小團體，其實就會等價於在抽象過後的圖裡面尋找一個點

團！而一個尋找大小為 k 的點團問題，在計算理論中被稱為 $k - CLIQUE$ 。

$$k - CLIQUE = \{G | G \text{ 有一個大小為 } k \text{ 的點團}\}$$

最後，讓我們換個角度來想，如果今天是要找一個小團體，裡面兩兩之間都不是朋友，也就是兩兩之間沒有邊相連，那麼是不是和點團有異曲同工之妙呢？沒錯，這樣的集合被稱作為獨立集(independent set)，在圖論中獨立集和點團有許多緊密連結的關係，不過在這邊我們先點到為止，以後有機會再詳細介紹。

§ 點覆蓋(vertex cover)

明天就是一年一度的園遊會了，身為班級導師的你，想要在班上籌組一個準備小組，來想想看該如何在園遊會大展身手。而為了要讓小組能夠有多元的聲音，於是和當初期末同樂會不同的是，現在你希望參加小組的人可以代表不同的聲音，也就是說每個朋友之間都可以至少推派一個人來參加準備小組，如此一來就可以在不需要將全班同學都納入準備小組的情況之下，聽到來自不同小團體間的建議。

把上述的想法轉換成抽象的圖論問題，就是「點覆蓋問題(vertex cover problem)」。圖 G 的一個點覆蓋(vertex cover)是

$V(G)$ 的子集，對於 $E(G)$ 中的邊都至少有一個端點是在點覆蓋裡面。用數學的語言我們會這樣寫：

定義 1-10 (點覆蓋, vertex cover)：給定一個圖 G ， $C \subseteq V(G)$ 是一個點覆蓋當任何 $e = (x, y) \in E(G)$ ，至少 $x \in C$ 或是 $y \in C$ 。

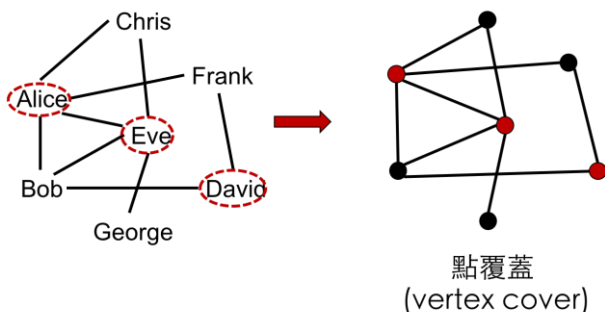


圖 1-8：點覆蓋(vertex cover)。

圖五是在7個點的圖中一個大小為3的點覆蓋。一個有趣的小觀察是，那些不在點覆蓋中的點會形成一個獨立集(independent set)！也就是那些不在點覆蓋中的點兩兩之間不會有邊存在，由此可知，點團(clique)、獨立集(independent set)、點覆蓋(vertex cover)之間有強烈的連結關係！

§ 配對(matching)

現在，身為班級導師的你，想要把同學們分成兩個兩個一組，並且希望每個小組中的成員，互相都是朋友，請問你該怎麼做呢？

這樣的問題在圖論中被稱為「配對問題(matching problem)」，目標就是要在一個圖中，把相鄰的兩點搜集起來，成為一個配對(matching)，同時希望這一個配對能夠越大越好。如果每個點都有成功被配對到的話，那麼這樣的配對就被稱為完美配對(perfect matching)。

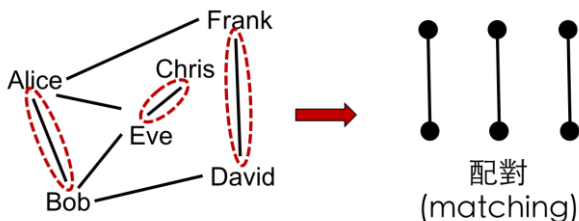


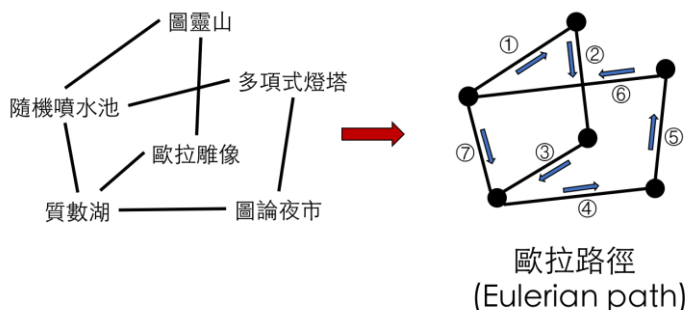
圖 1-9：配對(matching)。

§ 路徑問題

終於來到寒暑假了，忙了一整個學期，你決定要好好利用放假的時間出國玩，好好犒賞自己一番。然而，想要去的

地方太多了，擁有的時間和錢也有限，該如何有效率的規劃呢？

圖六是你要去的第一個城市：歐拉城，的城內地圖，在這邊點代表是一個景點，邊則是連接景點的道路。因為歐拉城的每一個角落都太美麗了，於是你想要把所有的道路都走過一遍，但是礙於時間因素，希望能夠盡量不要重複走過同一個道路，你有辦法判斷這是有可能的嗎



圖表 1-10：歐拉路徑(Eulerian path)。

這一個問題把所有邊都恰走過一次的問題被稱為歐拉路徑(Eulerian path)問題，源自於1736年歐拉(Euler)解出的七橋問題。這個問題擁有一個非常快的演算法：檢查是否除了起

點和終點外的每個點連出去的邊數都是偶數。讀者可以在茶餘飯後的時間想一想這是為什麼？

然而，如果將問題從每邊恰走一次改成每個點恰走一次，問題就變得困難許多了，而這個問題則被稱為漢彌爾頓路徑(Hamiltonian path)問題。

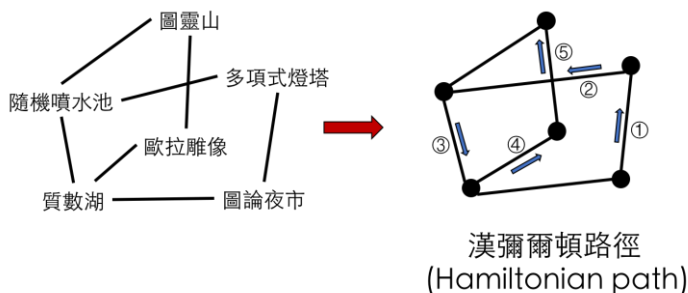


圖 1-11：漢彌爾頓路徑(Hamiltonian path)。

§ 結語

圖論是一個把抽象關係用某種幾何概念表達出來的數學領域，許多生活中常見的問題都可以在圖論中找到對應的數學形式，因此借助圖論抽象後的好操作性質，可以幫助我們更深刻的了解一個現實中的問題。

在理論資訊的領域中，圖論被非常廣泛的應用在各個角落，例如在偽隨機(Pseudorandomness)中重要的擴展圖(expander graph)，或是在近似複雜度(approximate complexity)中的唯一遊戲猜測(Unique Game Conjecture, UGC)等。這些圖論有趣的應用，在未來我們將會一一探索。

在這篇簡短的介紹中，先讓沒有接觸過圖論的讀者熟悉基本的定義與常見的問題，在之後的文章中圖論的概念或是問題將會不時的出現，到時候可以再翻回這篇複習。

本章小總結

- 圖論(graph theory)
- 點團(clique)
- 獨立集(independent set)
- 點覆蓋(vertex cover)
- 配對(matching)
- 歐拉路徑(Eulerian path)
- 漢彌爾頓路徑(Hamiltonian path)

1.3 如何解決問題：演算法

內容：演算法、時間複雜度

困難度：★★☆☆☆

在〈如何定義問題〉一文中，我們看到了該如何用數學的語言，將所謂的「問題」，用電腦可以接受的0/1語言來表示。緊接著在本文中，我們將要看看電腦是「如何」來解決問題的，更進一步，我們還要討論對於電腦來說，什麼是簡單的問題？什麼是難的問題？

§ 什麼是演算法(Algorithm)

隨著資訊科學日益發達，演算法這個字眼也時常出現在報章雜誌或是我們的生活中。而每個人對於演算法的描述也略有不同，有人喜歡從設計的層面介紹，有人喜歡用數學來定義。在這邊我決定要用最抽象的方式來解釋演算法是什麼：

演算法(Algorithm)的目的是要解決問題，透過三個步驟：輸入、計算、輸出，系統性地完成任務。

透過把演算法的概念簡化到這三個步驟，我們可以更方面的思考該如何把解決問題的能力，從人腦的抽象思考中，轉換成可以機械性、系統性完成的步驟。

以之前提到的質數判定問題為例，平時在算數學的時候，我們多少會根據經驗來判斷要如何快速的確認某個數是否是質數。例如說143，數字敏感度高的人一看就知道是 11×13 所以不是質數。最關鍵的地方是，我們再判定一個數是否是質數的時候，通常會隨著輸入而做出不同的判斷過程，換句話說，我們解決問題的步驟會受到經驗的影響而不太固定的。但是如果這時候我們希望請電腦幫我們處理這個問題，就沒辦法這麼隨性了(如果你是很厲害的演算法設計師就另當別論)。要有**系統性**和**機械性**的解決問題，我們必須將處理的步驟越簡化越好。於是，照著最簡單質數的定義，我們可以設計出下面這個簡單的演算法：

演算法（質數問題）

- 1 給定輸入 x
- 2 令 $q = 2$
 - A. 如果 $q \geq \sqrt{x}$ ，則跑到步驟3
 - B. 如果 q 整除 x ，則輸出“合數”
 - C. 若非則將 $q \leftarrow q + 1$ 並且回到步驟A

3 輸出“質數”

從上面這個質數判定演算法的例子，我們可以感受到這種處理問題的機械性與系統性，雖然這可能不是最快的方式，但是我們可以很容易的證明這個演算法可以幫助我們順利判定質數！

§ 演算法的優劣？

現在我們對演算法有了最基本的認識，接下來對於理論資訊學家在意的東西就是，該如何判定一個演算法是好是壞？在開始之前，讓我們先做個腦力激盪，來想想看有哪些可能的演算法評量標準？

- 速度：越快越好
- 空間：佔的空間越少越好
- 演算法可讀性：越容易理解越好
- 延伸性：越容易增加功能越好
- ...

以上這些都是常見的演算法評量標準，我們可以發現有些標準是比較容易量化比較的，有些則是會受到主觀意見的影響。身為一個怕麻煩的資訊科學家，我們傾向使用**方便量**

化且具有指標意義的評量標準來衡估演算法的好壞。而最常見也最容易分析的衡量標準就是：**時間**。

繼續上面質數判定的問題，當一個很有數學天分的人要判斷143是不是質數時，他第一時間就想到用11來分解，於是只花了一次的嘗試，就確定143不是質數。然而，當我們使用上面的演算法，會從2開始嘗試能否整除143直到11，一路下來需要花十次的嘗試，也就是說足足多花了十倍的測試次數！

不過也許你會反駁，那個數學天才有什麼厲害，他只不過是直覺好，如果我拿一個很大的數字要他來分辨，他大概還是需要嘗試很多次吧？這時候數學天才的演算法和我們上面的演算法比較起來也許就不會差這麼多了。

因此，我們可以體悟到演算法的表現好壞，是會隨著輸入而改變的。兩個演算法可能在不同的輸入上各自表現得比較好，也許其中一個會在大部分的輸入中表現得好，所以我們說他比較厲害。然而也許這個演算法又會在某些特別的輸入上面花非常多的時間。我們到底該如何有系統性且形式化的刻劃出演算法在時間上需要花的時間呢？更進一步，我們該如何比較兩個不同的演算法？

§ 時間複雜度(Time complexity)

首先讓我們來看看該如何將演算法花費的時間用簡單的方式描繪清楚。

直覺上我們會根據演算法步驟的次數多寡來衡量快與慢，然而隨著輸入東西的不同，直接將步驟次數作比較是不太恰當的。舉例來說，拿一個三位數字的質數判定時間和十位數字的質數判定時間做比較，即使用再笨的方法都應該是三位數字的方法比較快。

所以到底該如何刻畫不同輸入在同樣演算法下的運算快慢呢？為了解決這個問題，時間複雜度(Time complexity)的概念誕生了！

時間複雜度的中心思想，是想要將演算法所需要花的時間（步驟次數）和輸入的大小扯上關係。舉例來講，上面提供的質數判定演算法，對一個大小是 x 的數字來說，他所需要的執行步驟最多不會超過 \sqrt{x} 次，於是我們會說：這個演算法的時間複雜度是輸入長度開根號。透過將所需要花的時間和輸入長度做連結，我們可以把這樣的關係寫成一個時間複雜度函數，這一個函數將問題的長度，對應到所需要花的時間。

舉例來說，假如一個演算法在輸入大小1-8時所需要花費的時間如下：

輸入大小	1	2	3	4	5	6	7	8
所需時間	3	4	5	8	11	11	15	16

我們可以發現演算法所需時間和輸入大小呈現接近線性(linear)的關係，也就是說所需時間大約是輸入大小乘於1~3左右。這時候在習慣上我們會說，這個演算法的時間複雜度是 $O(n)$ 。對於 O (Big-Oh)這個符號，現在我先給一個比較不正式的說明，在本篇的進階附錄中會有詳細的介紹。直觀上來說，當一個演算法的時間複雜度是 $O(f(n))$ 時，代表著存在一個常數項的係數，使得 $f(n)$ 乘上這個係數後是演算法所需時間的上界(Upper bound)。因此，使用 O 來描述演算法的快慢，可以幫助我們省略常數項帶來的小小擾動，精準刻劃出所需時間隨著輸入大小變化的關係。

以上面的演算法為例，之所以說他的時間複雜度是 $O(n)$ 是因為我們可以找到一個常數，例如3，使得面對所有可能的輸入大小，演算法A需要的時間都不會超過 $3n$ 。

§ 如何比較時間複雜度的快慢

現在我們學會了利用時間複雜度來描述演算法需要花的時間多寡，那當我們有了兩個不同演算法的時間複雜度後，該如何比較優劣呢？

現在讓我們假設有另外兩個演算法，他們執行所需要的時間和輸入大小的關係是：當輸入大小為 n 時，演算法 A 需要 $A(n) = 10n$ ，演算法 B 則需要 $B(n) = n^2$ 。根據時間複雜度的定義，演算法 A 的時間複雜度是 $O(n)$ ，演算法 B 的時間複雜度是 $O(n^2)$ 。到底是誰的表現比較好呢？讓我們來看看這兩個函數在前20個輸入所需時間的大小關係：

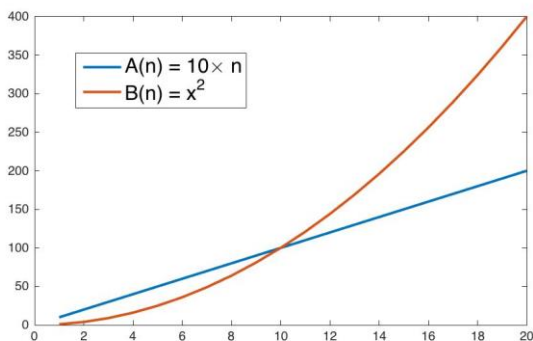


圖 1-12：演算法時間比較。

我們可以發現，在輸入大小為10之前的時候，演算法B比演算法A還要來得快。然而當輸入大小超過10之後，演算法A就比演算法B還要快了！於是對於大部分的輸入來看，演算法A需要的時間比演算法B還要少，所以我們會說演算法A比較好。

從這個例子中，我們可以發現，某些演算法雖然可能在輸入值小的時候表現得比較好，但是當他的時間複雜度函數隨輸入大小的變化比較劇烈時，對於那些比較大的輸入來說，他的表現就會比較差了。

在這邊一個重要的觀念是，我們在意的是當輸入大小跑到很大的時候，演算法的表現優劣。也許你會覺得不太舒服，為什麼不管輸入小時的時間快慢，然而在這邊必須講清楚，這是計算理論做事的一個態度：

我們不在乎瑣碎的常數變化，在意的是推廣後的漸進(Asymptotic)變化。

§ 結語

在這一篇針對解決問題的文章中，我們了解到電腦是如何透過演算法系統性且機械化的處理問題。在後半段更是學到如何用時間複雜度衡量演算法的優劣。透過認識電腦是怎

麼解決問題的，我們才能在未來更加精準的分析電腦的極限到底是在哪裡。一個問題的困難與否，取決於他是否有夠好夠快的演算法：假如某個問題有很快的演算法，對電腦來說就是個簡單的問題，畢竟這代表他很容易被解決。然而如果我們證明了某個問題沒有快速的演算法，就等於證明他是個困難的演算法，因為電腦必須花很多的時間才能處理。

經過了這三篇〈把自己當成電腦來思考〉的文章後，希望能夠讓讀者能夠認識到電腦是怎麼解決問題的？我們該如何量化的分析電腦解決問題的能力？在接下來的章節，我們將開始更具體的剖析電腦的抽象結構，更進一步的來探討電腦的極限在哪裡！

本章小總結

- 演算法(Algorithm)
- 時間複雜度(Time complexity)

1.3* 淺談時間複雜度

內容：時間複雜度

困難度：★★★☆☆

在〈如何解決問題：演算法〉中，我們學會了透過時間複雜度來量化分析一個演算法的表現優劣。那麼現在你算出了一個演算法的時間複雜度是 $O(2^{\log \log \log n})$ ，他的速度到底是快還是慢？你要怎麼和其他的演算法做比較？

此外，常用的時間複雜度符號 $O(\text{Big-Oh})$ 的意思是演算法所需時間的上界(Upper bound)，另外還有下界時間複雜度符號 Ω 和上下界時間複雜度符號 Θ ，他們的定義基本上就是 O 換了個方向，在下面會在嚴謹的介紹。現在讓我們先仔細想想用這樣的方法分析演算法的直觀意義是什麼，其實這樣一個把所需時間用別的函數上下包住的方法是一種**最差情況分析**(Worst-case analysis)。也就是說我們用 O 刻劃出演算法表現最差的情形，然而實際上也許這個演算法只有在遇到某幾個特別的輸入才會表現比較差，這時候我們有沒有更精準評量演算法好壞的方法呢？

綜合上面的一些想法，在本文中，我們將要針對三個方向做討論：各種常見的時間複雜度函數、其他的时间複雜度符號、最差情況與平均情況分析。

§ 常見的時間複雜度函數

今天我要跟大家介紹十個不同的時間複雜度函數，聽起來好像很多很可怕，但是如果能夠掌握的核心精神，相信大家一定可以順利理解每個的不同。

首先我要先把時間複雜度函數分成兩大類：易解的(tractable)和不易解的(intractable)。當一個演算法的時間複雜度是易解的，則對於我們來說這個演算法想要解決的問題就是比較簡單的。換個角度，如果一個問題沒有易解的演算法，只有不易解的演算法，那麼這個問題就會被視為困難的問題。

現在我們理解了易解和不易解演算法帶來的含義之後，就讓我們看看有哪些時間複雜度函數是易解的哪些是不易解的吧！

易解的(tractable)

對於理論資訊學家來說，目前公認的易解時間複雜度就是比**多項式時間**(polynomial time)還要快，或是一樣快的函數。多項式時間的複雜度函數長的樣子會像是 $O(n^k)$ ，其中 k 會是哪一個固定的常數。例如 $O(n^2)$, $O(n^{100})$, $O(n^{3.14})$ 。

也許有些人會對於這樣的定義感到懷疑，為什麼我們可以讓多項式的指數是任意的常數？如果像 $O(n^{100})$ 的指數是100，這樣當 n 很大的時候不也是花很多時間嗎？這個問題是比較屬於哲學層面的問題，因為即使當 n 很大使得 n^{100} 也很大的時候，雖然變大的速度快，但是仍然會比其他令我們懼怕的函數，例如指數函數 $O(2^n)$ 慢一些。尤其是這樣的差距會隨著 n 跑到超級超級大的時候越拉越開，請看圖 1-14 感覺一下 n^{100} 和 2^n 的成長快慢。

我們可以發現，當輸入大小 n 超過1000左右之後， 2^n 開始快速的拉開和 n^{100} 的距離，因為 n 每增加一，他的函數值就直接乘上兩倍。然而對於 n^{100} 來說，每增加一，他的函數值只會增加 $(n+1)^{100} - n^{100} \leq 2n^{100}$ ，也就是不到原本的兩倍。於是我們可以從這裡看出為什麼最後 2^n 會遠遠超過 n^{100} 。

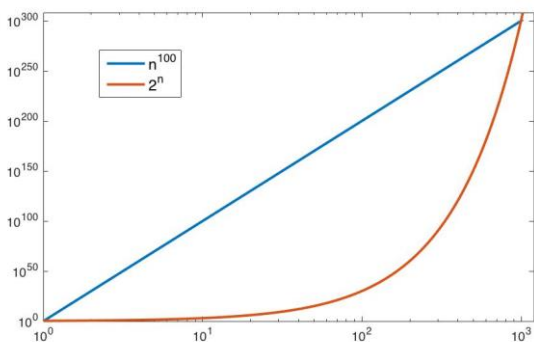


圖 1-13：藍色的線是代表多項式函數的 n 的 100 次方，紅色的線則是代表指數函數的 2 的 n 次方。我們可以看到在 $n < 1000$ 時，藍色遠遠大於紅色，然而紅色急起直追，在超過 1000 沒多久就把藍色遠遠甩在後頭。*請注意圖片的 x 軸和 y 軸都是用對數座標。

如果你被說服了使用多項式時間作為易解時間複雜度的定義後，可能會好奇在之中一定還是有快慢的差異，我們又是如何區分這些差異呢？以下我將列出幾個常見易解時間複雜度函數，這些函數的成長速度都不會比多項式成長得快。（下面的時間快慢是由快到慢，也就是所需時間的成長速度由慢到快）

- 常數時間(Constant time)： $O(1)$ 也就是無論輸入大小是多少，演算法都可以在固定的時間，例如說十個步驟內，解出答案！
- 對數時間(Logarithmic time)： $O(\log n)$
- 多項對數時間(Polylogarithmic time)： $O(\log^k n)$
- 線性時間(Linear time)： $O(n)$
- 線性對數時間(Linearithmic time)： $O(n \log n)$
- 二次時間(Quadratic time)： $O(n^2)$

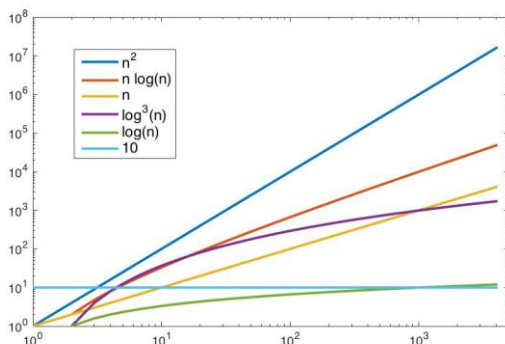


圖 1-14：易解的時間複雜度函數。

不易解的(intractable)

從上面介紹多項式時間的例子中，提到了 $O(2^n)$ 這個時間複雜度函數，他又被我們正式的稱呼為指數時間。他令人討厭的地方在於它所需要的時間會隨輸入大小增加一，就直接翻倍，於是成長的超級快，從圖一中就可以感受到指數時間和多項式的差別。

不過在指數時間和多項式時間之間還有兩個我們會感興趣的複雜度函數：

- 半多項式時間(Quasi-polynomial time)： $O(2^{\text{poly}(\log n)})$
- 次指數時間(Sub-exponential time)： $O(2^{n^\epsilon})$
- 指數時間(Exponential time)： $O(2^n)$

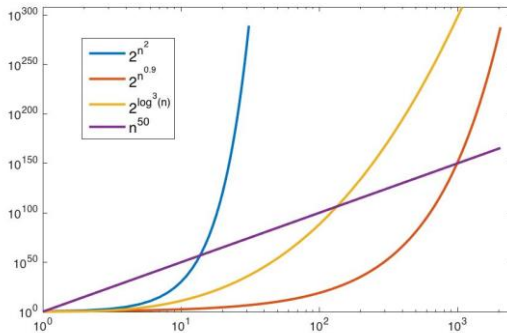


圖 1-15：不易解的時間複雜度函數。

一個小小的補充，在最近(2015年)，有個長期被大家覺得是在多項式時間和指數時間的問題：圖同構問題，被找到有半多項式時間的演算法，堪稱近十年來理論資訊界最大的突破之一[1]。

§ 其他的時間複雜度符號

在這裡我們總共要介紹另外四個時間複雜度符號： $\Omega, \Theta, o, \omega$ 。而在開始之前，先讓我們看看最常見的 O 在數學上的正式定義。

定義 1-11 (O , Big-Oh)：一個函數 $A(n)$ 是 $O(f(n))$ 若且唯若存在一個常數 $c > 0$ 和整數 N 使得當 $n \geq N$ ，我們都有 $A(n) \leq cf(n)$ 。

除此之外 O 還有一個方便的等價定義，在這邊也一併附上：

定義 1-12 (O , Big-Oh, 極限定義)：一個函數 $A(n)$ 是 $O(f(n))$ 若且唯若 $\lim_{n \rightarrow \infty} \frac{A(n)}{f(n)} < \infty$ 。

兩個定義的等價關係可以使用基本的 ϵ, δ 方法說明清楚。

這樣子定義的目的，其實就是想要刻畫函數 $A(n)$ 在 n 很大時的上界表現。我們可以注意到透過極限或是存在一個 N 的手法，可以忽略掉 $A(n)$ 在 n 小的時候的表現，也就是說我們可以全心的專注在 $A(n)$ 的漸進(Asymptotic)表現！

O 描述了函數漸進表現時的上界，我們同樣的也可以定義方便的符號來描述下界。為了方便，在接下來的定義我都採取極限的表示方法。

定義 1-13 (Ω , Big-Omega) : 一個函數 $A(n)$ 是 $\Omega(g(n))$ 若且

唯若 $\lim_{n \rightarrow \infty} \frac{g(n)}{A(n)} < \infty$ 。

如果 $A(n)$ 可以被一個函數 $h(n)$ 給上下包住，也就是說 $A(n) = O(h(n))$ 且 $A(n) = \Omega(h(n))$ ，那麼我們可以用 $A(n) = \theta(n)$ 來表示。

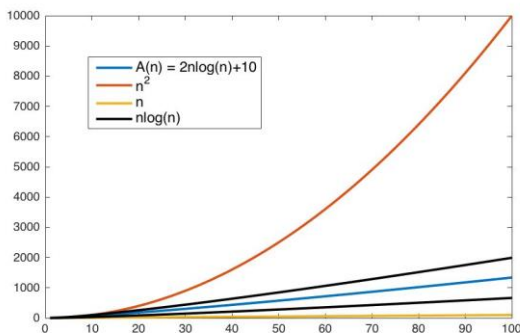


圖 1-16：藍色的線是我們觀察的函數，可以看得出來紅色是一個嚴格的上界，橘色是嚴格的下界，黑色則是剛剛好抓到了藍色的最大項次。

仔細觀察上面的定義，可以發現這裡的上下界都是可以接受同樣數量級的，那如果我們現在想要找一個緊(strict)的上下界，那麼則可以改使用下面這兩個複雜度符號。

定義 1-14 (o 和 ω , Small-oh & Small-omega)：一個函數

$A(n)$ 是

$o(f(n))$ 若且唯若 $\lim_{n \rightarrow \infty} \frac{A(n)}{f(n)} = 0$ 。

$\omega(g(n))$ 若且唯若 $\lim_{n \rightarrow \infty} \frac{g(n)}{A(n)} = 0$ 。

這些不同的符號看起來混亂，剛開始不太熟悉是正常的。這邊想要傳達給讀者一個重要觀念，在做複雜度分析的時候，我們在意的是漸進的表現，也就是當 n 很大的時候，所以在這個時候通常只有**最大項次**(leading term)會影響整個函數的成長快慢。而這個最大項次就是上面這些 $O, \Omega, \Theta, o, \omega$ 想要捕捉的。因為在很多時候，我們並沒辦法很精確地把最大項次寫出來，所以需要靠這些不同的觀點來旁敲側擊實際的狀況。

§ 最差情況(Worst-case)與平均情況(Average-case)

目前為止，我們透過時間複雜度函數描述了演算法解決問題時需要的時間，並且使用時間複雜度的符號，將函數簡化，只專注在最高項次的影響。在定義時間複雜度函數的時候，我們是利用輸入的長短做區分，也就是把長度同樣是 n 的輸入歸為一類，然後比較每一類所需時間的不同，刻劃出時間和 n 之間的關係。

然而當初我很刻意的避開了一個小小的問題：如果對於同樣大小的輸入有很不一樣的運算時間怎麼辦？舉例來說，在做質因數分解時，遇到一個長度是100的偶數和長度是100的質數，前者在第一輪就會被發現適合成數然後被判定為非

質數，然而後者卻要很辛苦的把所有的可能因數測試完。很明顯的，兩個輸入雖然長度相同，但是所需的時間差很多。

那我們該如何描述這樣子的現象呢？有兩種方法：最差情況分析(worst-case analysis)和平均情況分析(average-case analysis)。

最差情況分析的概念就是把同樣是大小為 n 的輸入中，找到演算法跑最慢的，拿他的時間作為代表放入時間複雜度函數中。於是我們可以知道，對於任何大小為 n 的輸入，所花的時間一定不會超過時間複雜度函數，這等同於給了演算法運算時間一個上界。

因為有些人認為最差情況可能不太容易出現，怎麼可以因為一粒老鼠屎就否定了整個演算法。於是出現了偏向貝式(Bayesian)思考的平均情況分析。在這樣的架構下，我們不再考慮所需時間的上界，而是在對於輸入分布的假設前提(prior distribution)下，提供一個所需時間的期望值。通常我們會取均勻分布(uniform distribution)當作prior distribution，然後算出演算法的平均時間複雜函數。

兩種分析方法沒有優劣之分，完全是差在關注的目標不同，所以往後在分析演算法的時候，要搞清楚在意的是平均

的表現，還是最壞情況的表現，如此一來才會推導出有意義的時間複雜度。

現在我們對於時間複雜度有了非常粗淺的了解，在往後的討論當中，這些基本工具將會不斷的出現，幫助我們量化問題的難度、演算法的好壞等等。最重要的是，複雜度使用的語言不太在意微小的變化，而是關注大方向上的變化，因此也許在輸入不大時，複雜度並不能提供很貼切的分析結果。但是當我們永遠不會知道接下來會遇到的輸入有可能會多大，複雜度的分析至少提供了我們一些概念，知道未來可能會遇到的情況為何。

本章小總結

- 時間複雜度函數
- 時間複雜度符號
- 最差情況(Worst-case)與平均情況(Average-case)分析

參考資料

[1] [Graph Isomorphism in Quasipolynomial Time](#) , László Babai .

基礎背景

第 2 章 計算模型

2.1 計算模型的血淚史

內容：計算模型的發展史、自動機

困難度：★★★☆☆

在前面幾篇文章中，我們將「問題」用數學的方法定義為判斷給定的輸入是正確與否。接著，我們學會了如何利用演算法來解決這樣子的問題。但是該如何設計演算法呢？有了演算法之後要怎麼執行呢？可以如何處理更多不同的問題嗎？面對這樣的需求，計算模型的概念相應而生。

計算模型是一種具有系統性、機械性以及全能性的解決問題工具。不像演算法只能針對特定的問題，它的目標是可以實現不同的演算法，解決不同的問題！它可以是具有實體的，例如說算盤、計算機、電腦，於是人們可以直接操作，利用它們來解決真實的問題。此外，它也可以是抽象的，像是之後會介紹的自動機、圖靈機。透過數學的分析，讓我們了解對應的實體計算模型能夠處理問題的極限在哪裡。

「系統性」指的是人們希望透過計算模型，複製我們解決問題的能力，把原本只能處理小問題的經驗，擴大到任意相似的問題。例如說質因數分解，我們從處理比較小的數字

中觀察到特定的步驟，接著推廣到可以處理任何的整數。

「機械性」則是在描述解決問題的過程是可以被清楚追蹤了解的，每一個步驟都是預先被設計好，可以讓機器或是人在不用知道自己在幹什麼的情況之下，照著指令就可以解決問題。

前兩個特性在之前介紹演算法的時候就有簡單的提到，而計算模型和演算法最不同的地方就是在於它的全能性。「全能性」是在說一個解決問題的方法可以處理很多種問題。也就是說不像演算法通常是侷限在解決特定的問題，計算模型就像是一個平台，它可以處理的是一個類別的問題，人們根據它可以做的事情，設計出演算法，然後解決問題。

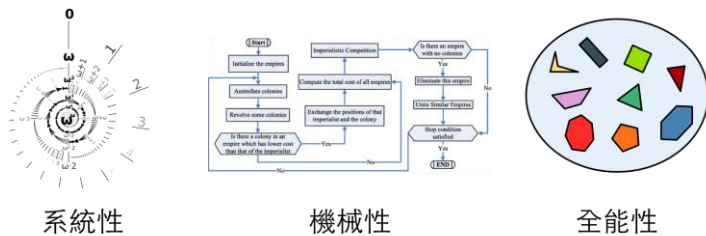


圖 2-1：計算模型的目標。

用比較非正式的觀點來看，一個最天然的計算模型就是我們的大腦！透過學習還有舉一反三，我們可以解決各式各

樣的難題。此外，把學習的經驗寫下來之後，便可以照著規則重複之前的想法，應用在未來將會面對到的類似問題。然而人腦固然有其厲害之處，但是最大的缺點就是在於速度慢、記性不好、會出小差錯等等，於是如果可以擁有類似大腦，但是又快又不容易錯的計算模型，那麼將會大幅提升人類處理問題的能力！

在這個章節的兩篇文章當中，首先我們將從歷史發展的角度來認識計算模型，並且探討該如何分析計算模型的能力？它的極限在哪裡？最後，我們將會看到一些近代最新穎的計算模型，例如量子電腦、對話式證明系統等等。希望透過這樣的介紹，讓大家對於計算模型有個基本的想法和概念，在之後的專題介紹中就可以比較輕鬆的一起探索各個計算模型的極限！

我們的腦袋本身就是個強大的計算模型，能夠系統性的學習然後解決問題，並且可以處理各式各樣不同的問題，我們的祖先就是靠著這個武器征服了地球。然而打敗了其他生物之後，人們開始不滿足，想要自我超越，這時候才發現人的腦袋雖然強，但是還是有不少缺點，例如會忘東忘西、沒辦法同時處理太多事情、對於瑣碎的運算有時候會出錯等等，於是開始有人在想到底有什麼辦法可以幫助人類突破人

腦的限制呢？因此以人腦的優缺點為借鏡，許許多多的計算模型就誕生了。

在計算模型的發展過程中，根據不同的需求有著不同模型的產生，同時也有一些需求被證明出不存在相應的計算模型。在接下來的篇幅中，我們將要踩著前人的成功與失敗，一窺計算模型的血淚史。

§ 實體的計算模型

實體的計算模型最早可以溯源到中國的算盤，當時人們利用這中小小的輔助工具協助商業上金錢計算的需要。而在西方，記錄顯示大約在十七世紀初葉，巴斯卡(Pascal)發明了巴斯卡計算機(Pascal's calculator)，可以處理基本的算術問題。



圖 2-2：算盤和巴斯卡計算機。

因為當時的數學及科學發展遠遠不及現在，大部分會需要的最多就只有算術問題，也因此在此後的兩三百年之間，幾乎沒有太多理論上的突破，只有在實體的操作上隨著工業化的發展才有很大的進步。

§ 希爾伯特(David Hilbert)的失敗

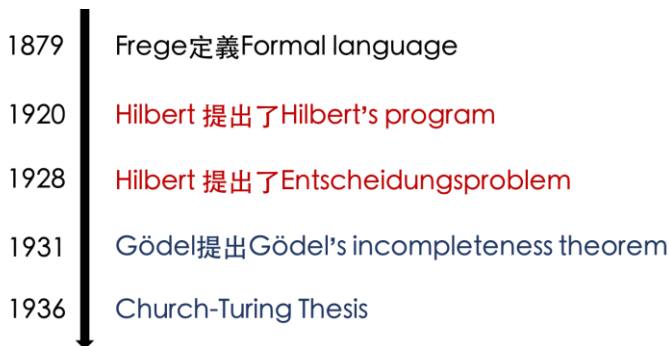
時間來到十九世紀末，當時的數學、科學、邏輯蓬勃發展，弗雷格(Gottlob Frege)在1879年定義了形式語言(Formal language)，開啟了後人使用數學邏輯的方式來定義問題，我們在前幾篇提到定義問題的方式，就是跟隨形式語言的這一套系統建立的。隨著數學的蓬勃發展，人們便開始思考，可不可以有一個計算模型幫忙人類解決數學的證明問題呢？當時數學界的領導者：大衛·希爾伯特(David Hilbert)便提出了「希爾伯特程式(Hilbert's program)」[1]的概念，希望可以找到一個系統性的方式，判斷數學定理的正確與否，建立數學穩固的基石。

然而，在1931年，哥德爾(Gödel)提出了震撼邏輯與數學界的「哥德爾不完備定理(Gödel's incompleteness theorem)」[2]，證明了在夠強的公理系統中存在一個無法被確證也無法被否正的陳述，更進一步，一個公理系統無法證明自己的一致性(consistency)，也就是說這個公理系統無法證明自己會不

會產生矛盾的結論！哥德爾拋出的震撼彈等於宣告了Hilbert's program的死刑，在某種程度上顯示出證明數學完備性的困難。

除了Hilbert's program之外，Hilbert在1928年退而求其次的提出了「決策問題(Entscheidungsproblem)」，期望可以有個演算法來判斷一個一階邏輯(first-order logic)的式子是否為真(原本Hilbert's program是針對數學定理的正確與否，現在只想要判斷比較簡單的邏輯式子是否正確)。然而天不從人願，阿隆佐·丘奇(Alonzo Church)和艾倫·圖靈(Alan Turing)各自發表了獨立的研究，用新的計算模型指出Entscheidungsproblem無法被解決。

雖然這一連串Hilbert的失敗讓近代數學看似出現了自我限制，然而上面提到的兩個推翻Hilbert夢想的研究卻開啟了璀璨的全新領域。接下來，我們便要看看Church和Turing是提出什麼樣的計算模型徹底改變了世界。



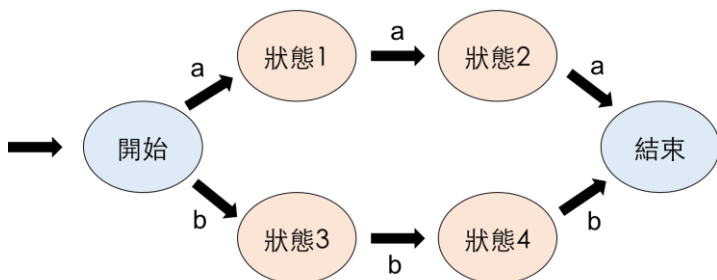
圖表 2-3：計算模型發展簡史。

§ 自動機(Automata)

自從電影《模仿遊戲》(The Imitation Game)在2014年上映後，Turing瞬間成為了家喻戶曉的名人。從電影當中，我們約略可以知道Turing在二戰期間製造了一台破密的機器，協助同盟國擊敗了德軍。故事的結果固然感動人心，然而Turing的貢獻不僅止於建造出破密機器，更重要的是他一手打造出完整的理論架構，發明了圖靈機(Turing machine)這一個抽象的計算模型概念，開啟了往後對於自動機(Automata)的研究。現在我們身邊的每一台電腦，雖然各自實作的硬體結構有所不同，但是最根本的抽象概念全部都是來自於Turing machine。

現在我們將Turing machine以及之後一系列的相關發展稱為「自動機理論」(Automata Theory)，這些計算模型有別於之前需要依靠人類操作的算盤還有計算機等等，他們就像是被注入了「靈魂」，行為能力的自由度大幅提升，可以處理更全面性的問題。

從高觀點的角度來看，自動機是有三個重要的部分組成的：符號、規則、狀態。這三個要素模仿著人類思考和解決問題的方式與過程，試圖讓計算模型擁有更強大的能力。符號就像是我們使用的語言，或是溝通的媒介，既然我們需要透過語言才能理解問題並且解決，同樣的計算模型也會要有屬於自己的語言。而規則決定了計算模型會如何處理問題，該在什麼時候做哪些事情，就好比是人類世界中的SOP(Standard Operational Process)。最特別也最重要的概念則是「狀態」，仔細觀察平常我們的思考方式，雖然我們都會有自己的SOP來決定該做些什麼，但是這個SOP是會隨著不同的情況，轉換成不同的SOP，而這就是自動機裡面「狀態」的核心概念！如果我們把所有要處理的狀況都硬寫成一個SOP，那麼隨著問題變複雜，腦袋應該會先受不了吧！但是現在我們將SOP根據不同的情境，切割不同的小SOP，根據需要的不同，拿出相對應的來使用，這樣會大幅度提升解決問題的能力。



圖表 2-4：自動機。圖中的圓圈即是「狀態」，黑色線條代表「規則」，線上面的英文字母則是「符號」。

例如圖四中的一個自動機，就是要判斷輸入的字串是否為aaa或是bbb。在最開始的狀態，當自動機讀到a時會改變成狀態1，如果是遇到b則會改成狀態3，萬一都不是的話，便會輸出No，代表這台自動機處理的問題中，你給的輸入對應的答案是錯的。一旦輸入給的是aaa或bbb，最終就可以順利地走到結束狀態，而自動機就會輸出Yes。

Turing就是透過了對於人類思考方式的細微觀察，建造了Turing machine這個偉大的計算模型。Turing machine是一個既符合人類思考方式，又具備強大計算能力的計算模型。在1936年被提出之後，至今仍然被視為計算理論中的基石。關於Turing machine詳細的介紹將會在下篇文章中出現。

§ 計算理論界天大的誤會：邱奇·圖靈論文

在坊間一直流傳著一種說法：「根據邱奇·圖靈論文 (Church-Turing Thesis)，Turing machine能夠解決所有其他計算模型能夠處理的問題，甚至跟人腦一樣強！」，然而當初Turing和他的老師Church其實從來沒有說過任何類似的話，他們也從未認為Turing machine可以解決所有人類都可以解決的問題。簡單來說，目前被世人稱為Church-Turing Thesis的那幾篇論文，裡面的內容僅限於探討Turing machine本身的極限，並沒有扯到所有的計算模型或是人腦的計算。

然而Church-Turing Thesis不斷的被誤用至今，甚至連筆者當初學習自動機理論的時候，都被誤導直到撰寫這篇文章查詢資料的時候，才發現許多人都被蒙在鼓裡。看來除了學習之外，更重要的是要去質疑學習的內容，才不會「盡信書，不如無書」。

關於Church-Turing Thesis的相關資訊，在[3]有非常詳盡的資料，也有對於世人的誤用有很完整的分析整理。另外，後來人們也有陸陸續續建構出比Turing machine還要強大的計算模型，例如：超電腦(Hypercomputer)。這些模型通常變得更複雜更難以讓人分析，再加上其實目前連Turing machine都尚未被完全了解，所以現在資訊理論界的重心還是會放在

Turing machine還有其他實用又好操作的計算模型上。關於超電腦以及其他近代更強大的計算模型，可以參考[4]的介紹。

在這篇文章中我們從歷史的觀點來認識計算模型，從最原初簡單的機械性操作儀器，到了Hilbert想要嘗試的全能型機器以及Turing建構出來簡單又強大的Turing machine。這一系列的發展，說穿了就是為了滿足人類探索計算極限的過程。我們希望能夠更了解自身智能的成因，並且抽象出來變成實體的物體，透過機械化、系統化這些概念，搭配量化的分析，期待能夠解決更多的問題。

本章小總結

- 希爾伯特程式(Hilbert's program)
- 決策問題(Entscheidungsproblem)
- 自動機(Automata)
- 圖靈機(Turing machine)
- 邱奇・圖靈論文 Church-Turing Thesis)

參考資料

- [1] : Hilbert's program , Stanford Encyclopedia of Philosophy 。
- [2] : Gödel's incompleteness theorem , Stanford Encyclopedia of Philosophy 。
- [3] : Church-Turing Thesis , Stanford Encyclopedia of Philosophy 。
- [4] : Copeland, B.J. 1997. 'The Broad Conception of Computation'. American Behavioral Scientist, 40, 690-716.

2.2 圖靈機

內容：圖靈機(Turing machine) 、可決定性(decidable)、可辨識性(recognizable)

困難度：★★★☆☆

圖靈機(Turing machine)是偉大電腦科學家、數學家、哲學家圖靈(Alan Turing)在1936年[1]提出的計算模型。Turing machine和另外兩個同時期提出的計算模型： λ 演算(λ -calculus)及 μ 遞歸函數(μ -recursive function)是等價的。Turing machine重要的意義在於它既是一個很符合人類思考方式，又具備強大計算能力的計算模型。（在Turing machine之前的計算模型，沒有一個可以同時兼顧這兩個性質）

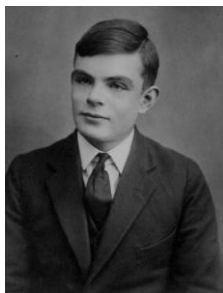


圖 2-5：Alan Turing (1912-1954)。

不過很可惜的是，Turing machine並不是最強大的計算模型。雖然在坊間流傳一種說法：「根據邱奇・圖靈論文（Church-Turing Thesis），Turing machine是最厲害的計算模型，甚至跟人腦一樣厲害」，然而這其實是個天大的誤會。Turing machine並非萬能，但是因為Turing machine簡單好操作，而且又尚未被人們完全了解，於是目前界算理論界主要的工作都還是圍繞在Turing machine之上。

在接下來的篇幅，就讓我們看看Turing machine背後的結構到底是什麼？為什麼人們覺得它和人腦可以有效解決的能力是一樣的？以及如何正式的定義Turing machine的能力？

§ Turing machine的操作型定義

讓我們先從Turing machine的正式定義開始，然後用幾個例子來熟悉背後的運作模式，最後試著理解為何人們會認為Turing machine可以代表人類解決問題的能力。在看定義之前，請讀者先回想一下在〈計算模型的血淚史〉中提到的自動機三要素：符號、規則、狀態，請在心中找出來這三個概念分別對應到哪個Turing machine的元素。

定義 2-1 (圖靈機 Turing machine)：一個 Turing machine 是由以下六個元件組成的： $(Q, \Sigma, q_{start}, q_{accept}, q_{reject}, \delta)$ 。其中 Q 代表所有狀態的集合， Σ 代表所有符號的集合， $\delta: Q \times \Sigma \rightarrow Q \times \Sigma \times \{L, R\}$ 則是轉換函數，其中 L 代表向左， R 代表向右。而 $q_{start}, q_{accept}, q_{reject}$ 分別代表開始的狀態、接受的狀態和拒絕的狀態。

註1：坊間有許多各式各樣對 Turing machine 的定義，基本上這些定義都是等價的，在這邊選擇使用最簡單的定義方便讀者理解。

註2：看到上面的符號請不要害怕，之後會詳細說明。

一台 Turing machine $T = (Q, \Sigma, q_{start}, q_{accept}, q_{reject}, \delta)$ 會有一個無限長的帶子(tape)，上面一格一格的可以放置 Σ 裡面的符號，在一開始什麼都沒有的時候，會放上空白符號 \sqcup 。當接收到輸入 $x = x_1 x_2 \cdots x_n$ 時，他會將 $x_1 x_2 \cdots x_n$ 放在袋子上面，並且把指針(head)指向第一個符號 x_1 。請參考圖一當輸入為 abcd 時的例子。

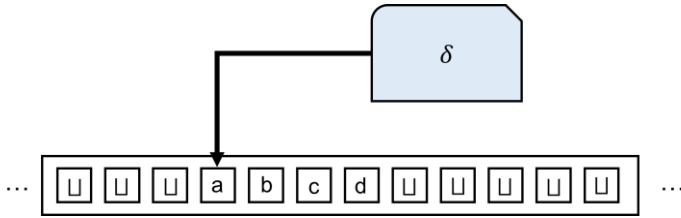


圖 2-6：Turing machine。當輸入為 abcd 時。

整個Turing machine的關鍵在於轉換函數 δ ，如果搞清楚 δ 是在做什麼，那麼就了解Turing machine的運作原理了。轉換函數會根據現在Turing machine的狀態、符號來決定該變成哪一個新的狀態、寫下哪一個新的符號、決定指針要向右還是向左。這也就是為什麼轉換函數的定義會是 $\delta: Q \times \Sigma \rightarrow Q \times \Sigma \times \{L, R\}$ 。

讓我們用下面這個例子來熟悉Turing machine的運作方式。

例子 1：

考慮Turing machine $T = (\{q_0, q_a, q_r\}, \{a, b, c\}, q_0, q_a, q_r, \delta)$ ，其中 q_0 是起始狀態， q_a 是接受狀態， q_r 是拒絕狀態。而使用的符號則是a,b,c。轉換函數則是定義如下：

輸入		輸出		
Q	Σ	Q	Σ	$\{L, R\}$
q_0	a	q_0	\sqcup	R
q_0	b	q_r	\sqcup	R
q_0	c	q_r	\sqcup	R
q_0	\sqcup	q_a	\sqcup	R

經過一些簡單的嘗試，可以發現這個Turing machine解決的問題是： $\{s \in \{a, b, c\}^* \mid s \text{ 沒有 } a \text{ 以外的符號}\}$ 。以下用兩個輸入作為範例： $s_1 = aaa$ ， $s_2 = abc$ 。

▪ 輸入： $s_1 = aaa$

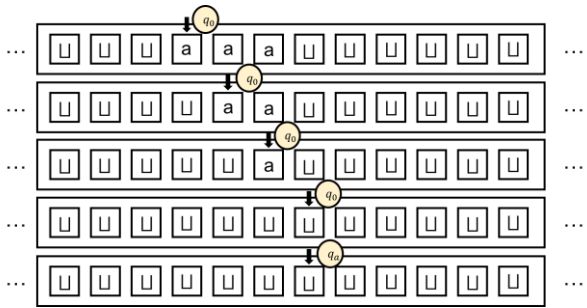


圖 2-7：輸入為 aaa 時的計算途徑。

最終Turing machine進入接受狀態 q_a ，於是輸出Yes。

- 輸入： $s_2 = abc$

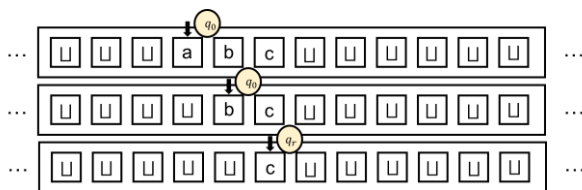


圖 2-8：輸入為 abc 時的計算途徑。

最終Turing machine進入拒絕狀態，於是輸出No。

■

§ 為什麼Turing machine很厲害？

從上面Turing machine的操作行定義也許大家還感受不太到Turing machine為什麼可以這麼厲害。在這個段落，我就要試著說服大家Turing machine可以做到很多事情！

首先，Turing machine開創了記憶體(memory)的概念。透過指針在帶子上更動符號，利用物理性的方式記錄在某個執行階段時產生的結果，不必受到狀態的影響，能夠充分發揮地域性(localization)的優點。就像我們的人腦一樣，會把不同事情的記憶放在大腦不同的位置，如此一來不會互相打架起衝突，只要分配運作得當，就可以發揮很大的作用。

再來就是指針巧妙的設計，模仿了在記憶體之間游移的感覺，充分的利用幾何上的優勢，用物理性的動作取代讓人昏頭轉向的數學式子操作。

§ 可決定性(Decidable)與可辨識性(Recognizable)

從上一個段落的結論，我們知道目前人們認為Turing machine和人腦有著相同的解決問題能力，差異只是在於解決的快慢效率。而在我們正式進入計算理論的範疇討論解決問題的效率之前，先讓我們正式的看看什麼叫做解決問題？有哪些問題是可以被解決的？難道有些問題是沒辦法被解決的？

直覺上，一個問題是可以被解決的相對應的意思就是說存在著一套解決的辦法。轉換成我們喜歡的數學方式來描述，就是說一個問題擁有對應的演算法。或是用自動機理論的術語來說，就是存在一個Turing machine可以解決這個問題。於是，我們將擁有以上特性的問題通稱為可決定(decidable)的問題。

定義 2-2 (可決定性, decidable) : 一個問題 $P \subseteq \{0,1\}^*$ 是可決定的，則存在一個演算法 A ，使得輸入 $x \in P$ 若且唯若 $A(x) = Yes$ 。

註：在這邊我用特別強調若且唯若這個條件，請先原諒我賣個關子，這樣做的原因將會留到介紹可辨識性的概念後一起解釋。

大部分常見的問題都是可決定的，因為通常都具有一個演算法可以解決他。而要說明一個問題是可決定的也很簡單，找一個演算法就對了。不過很遺憾的是，並不是所有問題都是可決定性的，例如有名的停機問題(Halting problem)，就是一個被證明為不可決定的問題，我們將會在附錄中討論並且證明。

當人們發現有不可決定的問題存在時，就開始想有什麼樣的方式可以來處理這一種類型的問題呢？這時候可辨識性(recognizable)的概念就誕生了。可辨識性的概念相對來說比較不直覺，讓我們先看看他的正式定義。

定義 2-3 (可辨識性, recognizable) : 一個問題 $P \subseteq \{0,1\}^*$ 是可辨識的，則存在一個演算法 A ，使得當輸入 $x \in P$ 時， $A(x) = \text{Yes}$ 。

請將**定義 2-2**還有**定義 2-3**交叉比較一下，有沒有發現不同的地方？還是你覺得兩個定義是等價的？其實兩個定義是完全不同的，最重要的差別是在於當 $x \notin P$ 時，可決定性要

求演算法一定不能輸出 Yes ，而可辨識性對於這種情況則是完全沒有任何要求！請參考下圖獲得一些幾何上的想法。

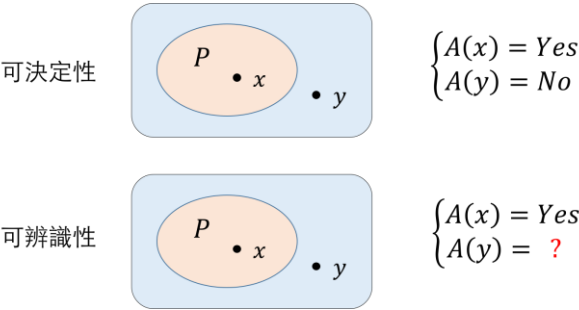


圖 2-9：可決定性 vs. 可辨識性

上面提到不可決定的停機問題，其實會是個可辨識的問題，也就是說雖然不存在一個能夠完全決定停機問題的演算法，但是至少存在一個演算法他可以在面對停機問題的正确輸入時，回答 Yes 。

這裡讓我用一個生活化的例子來解釋可決定性和可辨識性的概念：

例子 2：炸彈問題

炸彈問題 = { $bomb \in \{0,1\}^*$ | x 是一個真的炸彈 }

炸彈問題的目標是要辨別輸入 x 是不是一個真的炸彈，炸彈問題本身就像是圖一中的 P 一樣，是包含所有真的炸彈的集合。 $bomb$ 本身有可能是真的炸彈，就像圖23中的點 x ，也有可能是空包彈，就像圖一中的點 y 。

如果我們說炸彈問題是可決定性的，則代表存在一個演算法 A_1 ，使得面對一個炸彈輸入 $bomb$ 的時候，如果

- $bomb$ 是真的炸彈，則 $A_1(bomb) = \text{Yes}$ 。
- $bomb$ 是空包彈，則 $A_1(bomb) = \text{No}$ 。

然而說不定，炸彈問題沒有想像中的容易，實際上並不存在 A_1 這樣的演算法，也就是說炸彈問題是不可決定的。於是我們退而求其次，希望至少有一個演算法 A_2 能夠在當 $bomb$ 是真的炸彈時正確的回覆，至於如果是空包彈，回答正確與否我們並不在意，畢竟誤判也不會造成人員損傷。於是，如果存在 A_2 這樣的演算法，則我們稱炸彈問題為可辨識的。

■

於是從炸彈問題的例子中，可以讓我們感受到可決定性和可辨識性的差別。可決定性會直接意味著可辨識性，然而當找不到一個對於正確和錯誤的輸入都可以準確解決的演算

法時，我們退而求其次追求只在正確輸入表現正常，這就是可辨識性的概念。

然而還是有存在一些不可辨識的問題，這樣的問題有可能是在反可辨識的(co-recognizable)，也就是說當輸入是一個錯誤的輸入時，演算法可以告訴你他是錯的，但是面對一個正確的輸入時，這個演算法無能為力。以炸彈問題為例，那麼如果只存在可以辨識空包彈的演算法，那麼炸彈問題就是反可辨識的。

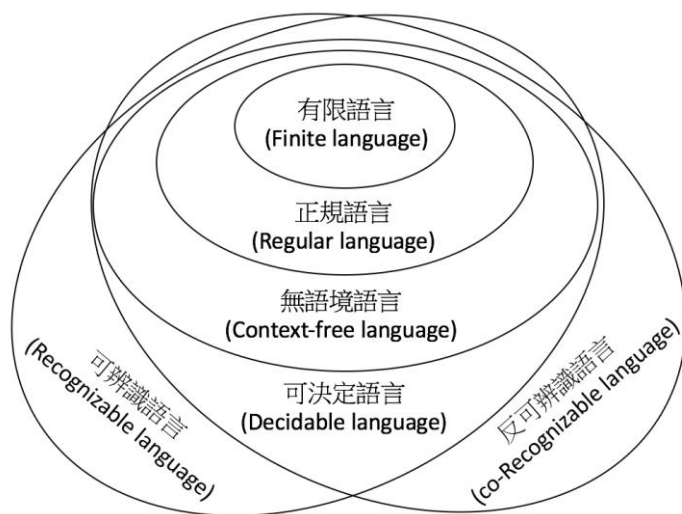


圖 2-10

於是，我們根據問題可以被解決的程度將之分類。如果一個問題可以被很陽春的自動機解出來，相較起來他就會是一個簡單的問題。而如果一個問題連Turing machine都沒辦法決定，甚至無法辨識，那麼就會被認為是一個困難的問題。

本章小總結

- 圖靈機(Turing machine)
- 可決定性(Decidable)
- 可辨識性(Recognizable)

參考資料

[1] : Turing, A.M. (1936). "On Computable Numbers, with an Application to the Entscheidungs problem". Proceedings of the London Mathematical Society. 2 (1937) 42: 230–265.

2.2* 不可決定性問題：停機問題

內容：停機問題(Halting problem)、對角化技術(Diagonalization)

困難度：★★★☆☆

在〈計算模型的血淚史〉中，我們很快速地認識了計算模型的核心精神、基本的模型定義、以及解決能力強弱的定義。在最後的段落，我告訴大家不同的計算模型會擁有不一樣的計算能力，也就是強的計算模型，例如Turing machine，可以解的問題會比弱的計算模型，例如有限狀態機，還要多。

而從Church-Turing Thesis中，人們相信，Turing machine能夠做的事情和人腦是一樣多的。換句話說，任何人類想得到的演算法，我們都可以透過Turing machine實作出來！也因此，人們非常好奇Turing machine到底能夠處理什麼問題？哪些是可以被解決而哪些是不行？

通常要說明一個問題可以被解決是比較方便的，我們只要設計出相對應的演算法就好了。但是，如果要說明一個問題是無法被解決的，我們可以怎麼做呢？最直接的辦法是列舉所有可能的演算法，然後說明沒有一個可以解決這一個問

題，於是這個問題是不可解的。不過我們可以馬上發現這樣做實在太麻煩了，光是想到要把所有可能的演算法列舉出來就已經令人頭昏眼花了。於是，另外一個可行的方法是透過謬誤法。具體來說，就是換個方向假設這個我們認為是不可解的問題是可以被解決的，於是存在一個相對應的演算法。接著我們再利用這個演算法推導出矛盾的結果，最後得出問題是不可解的結論。

在本篇文章中，我們將要使用謬誤法的精神，搭配計算理論中一個經典的工具：對角化技術，證明第一個Turing machine不可決定的問題。

§ 停機問題(Halting problem)

停機問題(Halting problem)是一個最常見的不可決定問題，在我們證明這個結果之前，先讓我們正式的介紹這個問題。

停機問題的輸入是 $([T], x)$ ，其中 $[T]$ 是Turing machine T 的描述(description)， x 則是 T 的一個輸入。這裡我們特別強調給的是Turing machine的描述，是因為通常Turing machine是來自於一個操作型的定義，然而當我們想要把一個Turing machine給其他人使用的時候，我們就必須把這樣的操作性定

義具體描述出來，如此一來才能和其他人溝通。如果實在很難想像，可以把Turing machine的描述想成是相對應的程式。

而我們正式定義停機問題為：

$$HALT = \{([T], x) | T(x) \text{ 會停止} \}$$

於是，當我們說一個演算法 H 可以決定停機問題，意思是說當面對輸入 $([T], x)$ 時， $H([T], x) = Yes$ 若且唯若 T 遇到輸入 x 的時候總有一天會停止。

不過很可惜的是，停機問題是不可決定的，也就是說，上述的演算法是無法存在的。

定理 2-1：停機問題是不可決定的。

接下來就讓我們來證明停機問題是不可決定的吧！

§ 證明停機問題是不可決定

在正式開始之前，先讓我們從下面的流程圖中，瞭解整個證明的流程。

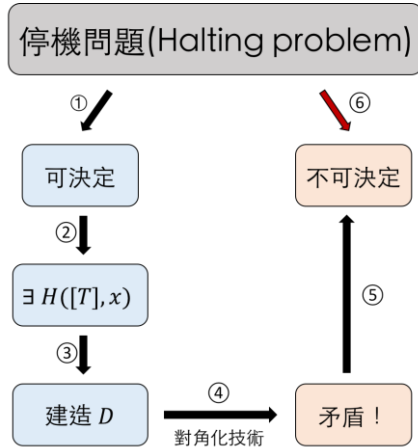


圖 2-11：停機問題的證明：(1) 假設停機問題可決定 (2) 存在演算法 H 可以決定停機問題 (3) 建構出新的演算法 D (4) 利用對角化技術從 D 推導出矛盾 (5) 於是根據謬誤法，停機問題是不可決定的。

首先我們假設停機問題是可決定的，於是存在一個演算法(i.e. Turing machine) H 使得

$$H([T], x) = Yes \Leftrightarrow T(x) \text{ 會停止}$$

接著，我們想要從 H 存在的假設中建造出一個有問題的演算法 D 。概念上 D 的目的是要故意和每個其他的Turing machine在某個輸出上做出不同的表現，如此一來他和其他所

有的Turing machine都會表現的不太一樣，因此當我們要考慮 D 和自己比較的時候，就會產生矛盾。有詳細解釋之前，讓我們先看看 D 的定義：

- D 的輸入為 $[T]$
 - 當 $H([T], [T]) = \text{Yes}$ 時，讓 D 進入無窮迴圈
 - 當 $H([T], [T]) = \text{No}$ 時，輸出 Yes

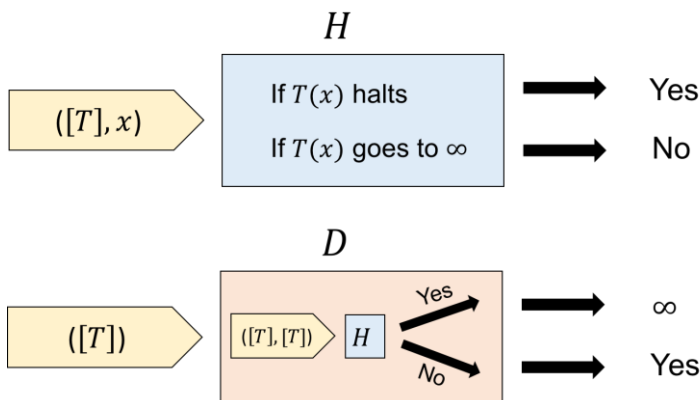


圖 2-12：演算法 H 和演算法 D 的定義。

這邊要注意的地方在於，一個Turing machine的描述也可以被當成是輸入！畢竟描述也是一個字串，雖然有可能很長，仍然是個合法的輸入。

此時，讓我們來看看當 D 的輸入是 $[D]$ 時會發生什麼事情？

如果 $D([D]) = Yes$ ，那麼代表 $H([D], [D]) = No$ ，也就是說 $D([D])$ 不會停止下來。但是我們假設了 $D([D]) = Yes$ ，也就是說假設會停下來，於是這個情況不可能發生，產生矛盾！

如果 $D([D])$ 進入無窮迴圈，那麼代表 $H([D], [D]) = Yes$ ，也就是說 $D([D])$ 會停下來，和原本假設的無窮迴圈產生矛盾！

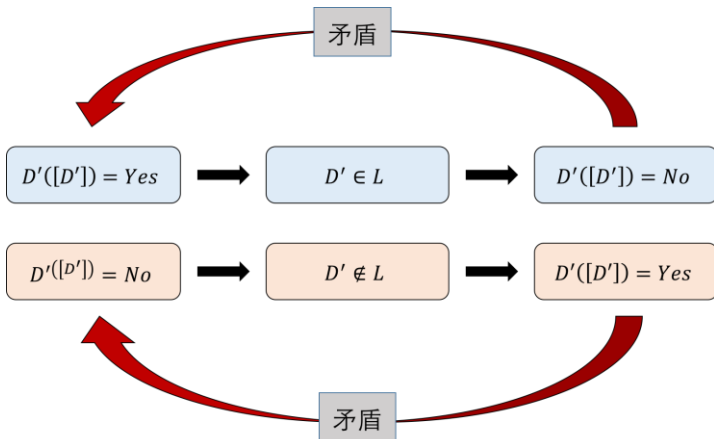


圖 2-13：對角化技術產生的矛盾。

當我們考慮 D 的輸入是他自己的時候，兩種可能發生的結果都會產生矛盾！於是 H 不存在，停機問題是不可決定的。

§ 對角化技術(Diagonalization)

在證明停機問題是不可決定的時候，最後一步採取了一個很奇怪的動作：把演算法 D 當成是輸入餵給自己。如此把自己丟給自己而產生矛盾的方法在數學上被稱為對角化技術(Diagonalization)。對角化技術是在1891年被德國數學家Cantor發明，用來證明實數的大小是不可數的[1]。一般來說有兩種方式來理解對角化技術的核心概念，第一個是透過幾何的直覺，另一個則是透過集合還有函數的概念。

- 對角化技術中的幾何直覺

讓我們用停機問題的證明當作例子，來看看背後幾何的直觀到底是什麼。

首先列舉所有的演算法，從 T_1, T_2, \dots ，接著製作一個表格來記錄在演算法 H 之下，面對輸入 $([T_i], [T_j])$ 的時候，得到的結果會是什麼？在這邊要注意的地方是，因為演算法可以被看成是一個相對應Turing machine的描述，也就是一個字串。

所以可以把演算法和字串可以被列舉的特性連結在一起，如此一來製做成表格就是一件可行的事情。

$\begin{matrix} x \\ [T] \end{matrix}$	$[T_1]$	$[T_2]$	$[T_3]$	$[T_4]$	$[T_5]$	\dots
$[T_1]$	Yes	No	No	Yes	No	\dots
$[T_2]$	Yes	No	Yes	No	Yes	\dots
$[T_3]$	No	Yes	No	Yes	No	\dots
$[T_4]$	No	No	No	Yes	Yes	\dots
$[T_5]$	No	Yes	No	Yes	No	\dots
\vdots	\vdots	\vdots	\vdots	\vdots	\vdots	\ddots

圖 2-14：演算法 H 的輸入輸出表格。

$\begin{matrix} x \\ [T] \end{matrix}$	$[T_1]$	$[T_2]$	$[T_3]$	$[T_4]$	$[T_5]$	\dots
$[T_1]$	∞	No	No	Yes	No	\dots
$[T_2]$	Yes	Yes	Yes	No	Yes	\dots
$[T_3]$	No	Yes	Yes	Yes	No	\dots
$[T_4]$	No	No	No	∞	Yes	\dots
$[T_5]$	No	Yes	No	Yes	Yes	\dots
\vdots	\vdots	\vdots	\vdots	\vdots	\vdots	\ddots

圖 2-15：演算法 D 的輸入輸出表格(灰色部分)。

製作完演算法 H 將表格後，接下來看看演算法 D 的輸入輸出表格是什麼吧？可以發現，演算法 D 的輸入輸出表格其實就會是在圖 2-14將對角線翻轉過來，把原本是 Yes 將改成無限大，把 No 改成 Yes 。如此一來，一旦想要知道 $D([D])$ 是什麼的時候，就會發現， $D[D]$ 既不會停下來，也不會一直執行，因而形成了矛盾。

這樣一個把對角線翻轉的概念，就是對角化技術的關鍵幾何直觀！

- 用集合和函數描述對角化技術

使用列表的方式，可以很直覺的從幾何上得到對角化技術的原始想法，但是如果要把這個概念抽象化，推廣到更一般的情況，那麼我們就必須咬著牙，嘗試把這些想法寫成數學的形式，也就是用集合還有函數的概念來表達。

首先觀察上面在做的事情，我們將所有的Turing machine排排站，然後看看每個Turing machine對於每一個輸入會不會停下來。此時，這個列表的動作其實等價於定義一個函數 $f: \{0,1\}^* \rightarrow 2^{\{0,1\}^*}$ ，使得 $f([T])$ 裡面搜集了所有 T 遇到會停下來的輸入。舉例來說，圖 2-14就會變成下面這個函數：

- $f([T_1]) = \{[T_1], [T_4]\}$

- $f([T_2]) = \{[T_1], [T_3], [T_5]\}$
- $f([T_3]) = \{[T_2], [T_4]\}$
- $f([T_4]) = \{[T_4], [T_5]\}$
- \vdots

由於整件事情的目標是要證明這樣一個函數 f 是不存在的，也就是說如果我們可以計算 f 那麼就可以推導出矛盾。而使用上面的幾何直觀，可以透過將對角線翻轉，製造出既不正確又不錯誤的集合造成矛盾。但是在這邊將對角線翻轉的概念要怎麼實踐呢？首先，對角線在這邊的意義在於判斷一個Turing machine T 是否會在 $f([T])$ 裡面。由於假設了 f 是可以被計算的，所以可以建造一個特殊的Turing machine D ，利用 f 把每一個Turing machine面對自己的結果翻轉，也就是讓 D 和每一個Turing machine T 在面對輸入是 $[T]$ 的時候表現不一樣。於是可以得到

$$f([D]) = \{[T] \mid [T] \notin f([T])\}$$

然而這是後矛盾就出現了，因為我們無法確認 $[D]$ 到底有沒有在 $f([D])$ 裡面！也就是說 $f([D])$ 根本不應該存在。然而在 f 存在的前提之下，一定可以造出 $f([D])$ 這個矛盾的東西，於是我們只好得出結論： f 不存在(無法計算)。

透過將一切操作集合和函數的語言描述，可以讓我們嚐到不同風味的對角化技術。

§ 結語

在這篇文章中，我們看到了如何利用對角化技術證明了停機問題是不可決定的。這件事情帶來了幾個不同觀點的影響：

對於理論學家來說，等同於知道了Turing machine是無法決定任何問題的，竟然連停機問題這樣看起來很單純的事情都無法處理。

利用停機問題，搭配上未來會介紹的簡約方法(Reduction)，可以在更多問題證明為是不可決定的。

透過演算法(Turing machine)是可列舉的特性，在往後許多重要的定理，都可以仿照證明停機問題是不可決定的歸謬法，使用對角化技術製造矛盾。

本章小總結

- 停機問題(Halting problem)
- 對角化技術(Diagonalization)

參考資料

[1] : Georg Cantor (1892). "Ueber eine elementare Frage der Mannigfaltigkeitslehre"

2.3 其他計算模型

內容：簡介其他計算模型、隨機圖靈機、電路、互動式證明

困難度：★★☆☆☆

目前為止的討論，因為Church-Turing Thesis的猜測，重心都是放在Turing machine，畢竟大家相信Turing machine能解決的問題是和人腦能夠解決的差不多。不過在實務上，除了考慮解決問題的能力之外，人們也會在乎處理問題時的效率快慢以及各種資源的分配利用。因此，許許多多的計算模型相繼誕生，有的是為了在應用層面實現的方便程度；有的是為了捕捉到更多人類思考的精神，希望可以增加解決問題的能力；有的則是把物理世界觀察到的現象拿來使用。

每個不同的計算模型都會有他的優點和缺點，對於理論資訊學家來說，探討這些模型的極限可以幫助人們更加瞭解什麼時候該使用哪個計算模型。此外，更重要的則是認識不同計算模型之間的關係。有些關係是絕對包含的，例如：任何Turing machine在多項式時間可以處理的問題，都可以被多項式大小的電路(circuit)解決。有些關係則是等價的，例如：多項式次數的互動式證明(Interactive proof)，和多項式空間的Turing machine的解決問題能力是等價的[1]。最令人心醉神迷

的則是曖昧不明的關係，例如量子(Quantum)Turing machine 和非確定性(Non-deterministic)Turing machine之間的強弱消長 [2]。以上這些例子讀者不必現在就瞭解其背後的原理是什麼，甚至看不懂代表的意義也沒有關係，在這邊要帶給大家的感覺是，不同的計算模型之間會有許多有趣的關係，就像人與人之間一樣。一旦我們越瞭解他們之間的關係，就可也更加善加利用各自的長處，帶來更好的效益。

在接下來部分，將會以比較輕鬆的方式，介紹一些近代新出現又很重要的計算模型。對於一些特殊的名詞不熟悉是正常的，畢竟在這邊的目的是要引起讀者的興趣，更深入的細節將會在第二部分的專題文章詳細討論。那就放輕鬆，一窺計算理論五彩繽紛的世界吧！

§ 隨機圖靈機(Randomized Turing machine)

隨機Turing machine的概念很單純，就是讓Turing machine在運作的時候，可以使用到隨機性(randomness)。然而該如何把隨機性加入Turing machine之中？加了隨機性有什麼好處？該怎麼樣定義相對應的複雜度類別？隨機Turing machine的複雜度類別和一般的差別在哪裡？一旦人們決定把隨機性加入Turing machine之後，上面這些問題馬上就蹦出來

等著被解決了。那就讓我們來看看前人是怎麼處理這些問題的吧！

首先是該如何把隨機性加入Turing machine之中，在這邊必須要注意再加入的同時如何不會影響到其他資源的效率，例如會不會讓執行的時間變慢？會不會讓所需要的空間變多？目前大家常常會使用的隨機性加入原則有下面這幾種，他們在時間和空間的影響上都只有差常數倍的不同，所以基本上我們會視為是一樣的。

- 有一個「擲硬幣的黑盒子(coin-tossing black box)」，讓Turing machine可以隨時隨地直接向這個黑盒子取用隨機位元(random bit)。
- 有一個黑盒子(black box)，讓Turing machine可以隨時隨地直接向這個黑盒子取用 $\{1, 2, \dots, n\}$ 中的一個隨機數字(random number)。
- 在Turing machine上面的每個格子都放上隨機位元。

有不同的證明當中，使用不同的隨機性機入方式會有不同的方便程度，不過基本上，前兩個使用黑盒子的方式就很方便了。

接著當然就要開始說服各位加入隨機性的確是對於計算有顯著的差距，而在這邊將要用兩個經典的例子告訴大家隨機性的能力。

- 多項式身份測試(Polynomial identity testing)：要如何確認兩個多項式是一樣的？[3]
- 完美配對(Perfect matching)：要如何知道一個圖(graph)裡面有一個完美配對？[4]

上面這兩個問題都擁有多項式時間的隨機演算法，然而兩者都沒有次指數時間(Sub-exponential time)的非隨機性演算法。

§ 電路(Circuit)

電路(circuit)是一個由輸入、計算閘和電線組成的計算模型。

輸入是由一串位元字串所組成，透過電線和一些計算閘相連接。計算閘是則是一些簡單的固定函數，例如：及閘(And gate)、或閘(Or gate)、互斥或閘等，擁有兩個位元的輸入和一個位元的輸出。

輸入	及閘 (And gate)	或閘 (Or gate)	互斥或閘 (Xor gate)
(1,1)	1	1	0
(1,0)	0	1	1
(0,1)	0	1	1
(0,0)	0	0	0

在經過電線的牽引之後，最一開始的輸入位元會經歷各式各樣的計算閘然後匯集在一個輸出，成為電路的計算結果。

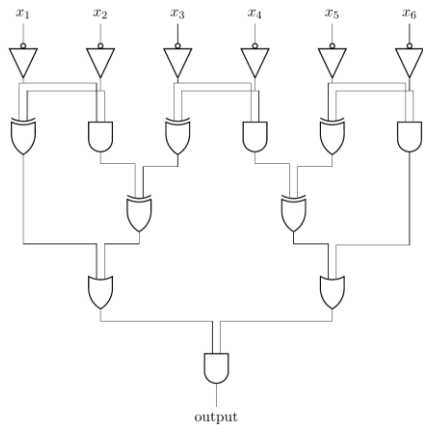


圖 2-16：電路(circuit)。

在圖二中，最上方的 x_1, x_2, \dots, x_6 就是輸入，經過了一連串的計算閘後，最後在下方整合成輸出。

為什麼要使用電路？其實最根本的原因是來自於布林公式(Boolean formula)，也就是由一些取值於0,1的變數，還有一些運算子(operator)組成的式子，例如：

$$f(x_1, x_2, x_3) = x_1 \wedge (x_2 \vee x_3)$$

在電路計算模型中最基礎的一個定理就是在說任何的布林公式，都可以轉換成一個電路，同時大小的改變不會差太多。

電路計算模型一個很重要的特性是在於，它是非均一性(non-uniform)的。也就是說，面對同樣的一個問題，對於不同長度的輸入，會需要不同的電路。以圖二為例，計算的是當計算長度為6的問題輸入，如果輸入的長度變了，就需要換成另外一個電路來使用。因為這種非均一性的緣故，當我們說一個問題可以被電路解決的時候，通常背後的意思是說存在一群電路 C_1, C_2, \dots ，接受輸入大小 $1, 2, \dots$ 。

有了非均一性的概念，就可以來看看該如何分析電路的複雜度了。一般來講，會根據電路的一些外在性質來做分類，例如有下列幾個常見的分類指標：

- 計算閘的個數
- 從輸入到輸出最多會經過幾個計算閘
- 計算閘的類型

如果一個問題擁有一群計算閘個數是輸入大小多項式電路，則人們會把它放入一個被稱為 $P/poly$ 的複雜度類別；如果一個問題擁有一群從輸入到輸出所經過最多的計算閘各式是輸入對數層級的電路，那麼它會被放入 NC 這個複雜度類別。而假如我們讓計算閘可以有無上限的輸入個數，則複雜度類別會晉升成為 AC 。

電路是一個看起來很單純，其實很複雜很難分析的計算模型，雖然它具有非均一性，但是卻和Turing machine有很強的連結，有之後專題介紹的部分，我們將會看到電路是怎麼影響著計算理論的發展。

§ 互動式證明(Interactive proof)

互動式證明是二十世紀末期的璀璨之星，當1992年Shamir證明出 $IP = PSAPCE$ 時，震驚了理論資訊界，因為這是第一個正式的非對角化證明。換句話說，互動式證明跳脫了傳統計算理論習慣的證明方式，採取完全不同的方法，因此解決了原本無法證明的問題。也因為這樣，當時人們一度

以為 P vs. NP 的問題有希望被解決，雖然最後仍然無功而返。

互動式證明的想法類似於近代密碼學中攻防的概念，假想現在有兩個人，一個是擁有至高無上能力的證明者 (prover)，另一個是只能有多項式時間做事的驗證者 (verifier)。現在驗證者拿到了一個問題的實例，並且希望知道這個實例是對還是錯的。例如他拿到了兩個圖(graph)，然後他想要知道這兩個圖是否不同構(non-isomorphism)，也就是說兩個圖之間是否不存在一個保有邊(edge)關係的對射(bijection)。

現在驗證者可以做的事情就是向這個證明者問問題，而證明者就會回復他。如此一來一往，在規定的次數之內，看完了證明者的所有回覆，驗證者必須決定最後的答案是對或錯。以下就以上面的圖不同構(graph non-isomorphism)問題為例。

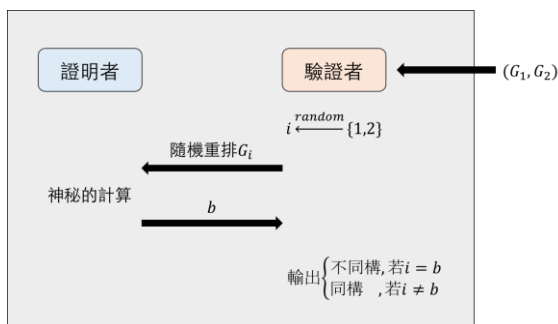


圖 2-17：圖不同構的互動式證明。

簡單驗證一下：

- 當 G_1 和 G_2 不同構時，證明者的回答必定是正確的。
- 當 G_1 和 G_2 同構時，證明者的回答有一半的機會是錯的。

於是我們發現上面這個互動式證明可以解決圖不同構問題！這邊特別強調圖不同構問題是在於，圖不同構問題本身是在 NP 之外的，也就是說本來他被認為是一個很難的問題。但是如今竟然可以用互動式證明在一個回合之內解決，由此可以感受到互動式證明的潛在能力。

§ 結語

在以上短短的介紹之中，讀者可以看到幾個近代著名的計算模型的發展過程，以及他們和其他計算模型的關係。可以發現很不一樣的計算模型，在某些層面卻又緊密的結合。

而且透過不同的方式來試圖回答原本模型的問題，有時候反而提供了更好的工具。

在未來的專題介紹部分，將會對各個計算模型做詳細的介紹，希望可以帶領大家更深入認識他們背後有趣的知識。

本章小總結

- 隨機圖靈機(Randomized Turing machine)
- 電路(Circuit)
- 互動式證明(Interactive proof)

參考資料

[1] : Adi Shamir. $IP = PSPACE$. Journal of the ACM, volume 39, issue 4, p. 869–877. October 1992. ◦

[2] : Vazirani, Umesh. "A survey of quantum complexity theory." Proceedings of Symposia in Applied Mathematics. Vol. 58. 2002. ◦

[3] : Saxena, Nitin. "Progress on Polynomial Identity Testing." Bulletin of the EATCS99 (2009): 49-79.

[4] : Karp, Richard M., Eli Upfal, and Avi Wigderson. "Constructing a perfect matching is in random NC." Proceedings of the seventeenth annual ACM symposium on Theory of computing. ACM, 1985.

基礎背景

第 3 章 複雜度動物園

3.1 複雜度類別

內容：複雜度類別(complexity class)、時間階層定理(time hierarchy theorem)

困難度：★★★☆☆

歡迎來到複雜度動物園！趁今天是個好天氣，讓我們來瞧瞧複雜度動物園裡面有什麼有趣的事物吧！

在這趟複雜度之旅中，我們除了要認識一些常見的複雜度類別之外，更重要的是要了解他們之間的關聯。就像生物的藉們綱目科屬種一樣，這些複雜度類別有些屬於比較高的層次，包含了下面的其他類別，有些則是處於底層。越上層的類別代表越困難，包含的問題越多，而越下面的類別則是相反。但是比較神秘的地方是在於，有些複雜度類別之間的關係並不為人所知，也就是說人們目前還沒有能力找到他們之間的關係，而這樣的開放式關係，也是目前許多研究人員努力的目標。

這次的參觀時間有限，讓我們拿起複雜度動物園的導覽圖，一起探索這個神秘的世界吧！

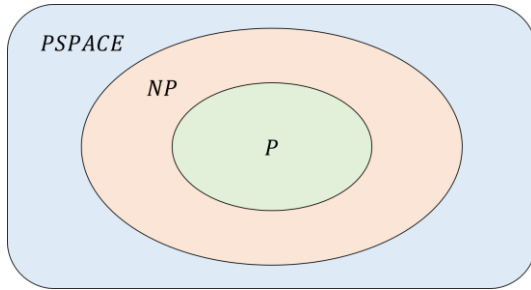


圖 3-1：陽春版的複雜度動物園。

複雜度(complexity)，是在描述一個計算模型解決一個問題時，在各種資源(例如：時間、空間、隨機性、容錯率等等)的分配結果。而複雜度類別(complexity class)在做的事情，則是把同樣分配資源方法下的計算模型，可以解決的所有問題都聚集在一起，搜集成一個類別，把他們視為一樣困難(或是容易)的問題。

秉持著這樣的精神，我們可以定義出第一個最單純的複雜度類別： P 。

定義 3-1(P , 多項式時間類別)：複雜度類別 P 搜集了所有可以在多項式時間之內被 Turing machine 解決的問題。

P 是根據解決問題所需要在Turing machine上花的時間定義出來的複雜度類別，由於要求了必須在多項式時間內完成，有許多常見的問題因為沒有多項式時間複雜度的演算法，所以沒有被列在 P 的行列之中。除了多項式時間之外，我們也可以對於任何時間的長短來定義時間複雜度類別。

定義 3-2($TIME(f(n))$, $f(n)$ 時間複雜度類別)：複雜度類別 $TIME(f(n))$ 搜集了所有可以在 $O(f(n))$ 時間內被 Turing machine 解決的問題。

透過這樣的分類，在某種程度上將問題的困難度用已知的解決時間快慢來區分。

同樣的，可以根據在Turing machine上使用的空間做為複雜度類別的定義標準。

定義 3-3($PSAPCE$, 多項式空間類別)：複雜度類別 $PSAPCE$ 搜集了所有可以在多項式空間之內被 Turing machine 解決的問題。

同樣的，空間複雜度可以推廣到定義在任何複雜度函數上。

定義 3-4 ($SAPCE(f(n))$, $f(n)$ 空間類別)：複雜度類別 $SAPCE(f(n))$ 搜集了所有可以在 $O(f(n))$ 空間之內被 Turing machine 解決的問題。

除了常見的時間與空間之外，根據不同的計算模型，也可以定義出相對應適合的複雜度類別。以下將簡單介紹兩個其他計算模型的複雜度類別，在此並不要求讀者全部了解，只是想要讓大家感受除了時間和空間之外還有很多不同重要的資源會讓我們在意。

第一個例子是電路(Circuit)，人們根據下面幾個資源的多寡定義出不同的複雜度類別：

計算閘的個數：計算閘的個數和輸入大小之間的關係。例如 $SIZE(f(n))$ 即是搜集了所有當輸入大小為 n 時，可以被大小為 $f(n)$ 的電路解決的問題。

從輸入到輸出最多會經過幾個計算閘：又被稱為電路的深度(depth)。例如 NC_1 即是搜集所有當輸入大小為 n 時，可以被深度 $O(\log n)$ 的電路解決的問題。直觀來看，深度越深的電路可以解決更多的問題。

計算閘的類型：基本的計算閘只有提供且(and)、或(or)、多數(majority)的功能，此外，計算閘可以容許的輸入個數也會有所不同。例如：**AC**即是搜集可以被不限制輸入多寡的且閘和或閘電路解決的問題。直觀來看，功能越多或是容許的輸入量越多，電路的能力會更強。

另外一個有趣的例子是隨機圖靈機(Randomized Turing machine)，我們可以根據誤差類型的不同，定義出不同的複雜度類別。

一旦加入了隨機性，除了過程將會變得不固定之外，輸出的解答也就會有可能不會是永遠都是正確的。舉例來說，一個隨機演算法有可能只有在90%的時候會回答正確的答案，在另外10%的時候，會輸出錯誤的答案。因此在定義隨機複雜度類別的時候，除了考慮各種資源的使用狀況，還需要考慮演算法輸出的正確性與否。

再考慮決定性問題的狀況之下，解答會有兩種可能：對(Yes)或錯(No)。而輸出也是有這兩種可能，因此交叉搭配起來就變成有了四種組合如下。

解答\輸出	Yes	No
Yes	O	第一類型錯誤

No	第二類型錯誤	O
----	--------	---

而隨機複雜度類別，就是根據是否擁有這兩種類型的錯誤來定義。

隨機複雜度類別	第一類型錯誤	第二類型錯誤
<i>BPP</i>	O	O
<i>RP</i>	O	X
<i>co - RP</i>	X	O
<i>ZPP</i>	X	X

對於每個計算模型，都可以根據所擁有的資源定義出許多具有深層意義的複雜度類別。而計算理論很主要的一個工作，就是刻畫不同複雜度類別之間的交互關係。以下把複雜度類別之間可能的關係分成三種，分別做簡單的介紹，讓讀者感受到計算理論的終極目標為何。

- 同樣資源間的關係
- 不同資源間的關係
- 不同計算模型間的關係

§ 同樣資源的複雜度類別關係

直觀的來看，當我們讓計算模型擁有更多資源的時候，他能夠解決的問題會變得更多。用複雜度類別的語言來說，

就是比較大的類別會包含小的類別。例如： $P \subseteq EXP$ ，多項式時間類別會被包含在指數時間類別之內。然而這時候我們可能就會想要問， EXP 是嚴格(strictly)的大於 P 嗎？也就是說有沒有問題一定需要指數時間才能解決，多項式時間是不夠的？如果沒有的話，那麼大家都用多項式時間就好了，何必要多浪費時間花到指數這麼多？

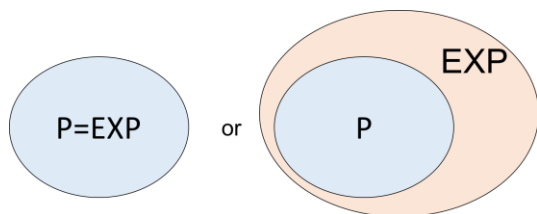


圖 3-2： P 與 EXP 之間的關係。

不過很可惜的是， EXP 的確嚴格的大於 P 。於是更進一步，我們會好奇，需要增加多少時間才有可能讓複雜度類別嚴格的提升？嚴謹的來說，當我們令 $TIME(f(n))$ 表示當輸入大小為 n 需要費時 $f(n)$ 的複雜度類別，什麼樣子的 $g(n)$ 可以使得 $TIME(f(n)) \subsetneq TIME(g(n))$ ？也就是後者嚴格的大於前者？時間階層定理提供了 $g(n)$ 的一個充分條件。

定理 3-1(時間階層定理, Time hierarchy theorem)：當 $f(n) \log f(n) = o(g(n))$ 時， $TIME(f(n)) \subsetneq TIME(g(n))$ 。

時間階層定理告訴我們在Turing machine上，擁有的時間複雜度每從 $f(n)$ 增加到 $f(n) \log f(n)$ ，計算的能力就會嚴格的增加！也就是說會存在一個問題 L ，使得 $L \in \mathbf{TIME}(f(n) \log f(n))$ ，但是 $L \notin \mathbf{TIME}(f(n))$ 。

而時間階層定理的證明則是運用了經典的對角化技術（在〈不可決定性問題：停機問題〉中有詳細介紹），以下簡單提供證明的高觀點描述。

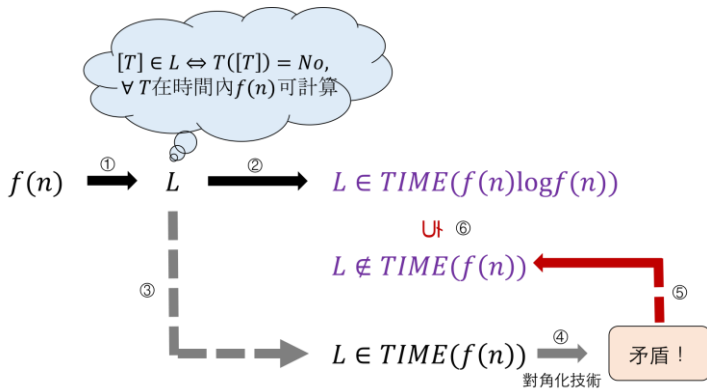


圖 3-3：時間階層定理的證明：(1)根據 f 構造特殊問題 L (2)因為存在僅有對數支出的模擬演算法，所以 L 在 $\mathbf{TIME}(f(n) \log f(n))$ 裡面 (3)假設 L 在 $\mathbf{TIME}(f(n))$ 裡面 (4)使用對角化技術推得矛盾 (5)於是 L 不再 $\mathbf{TIME}(f(n))$ 裡面 (6) $\mathbf{TIME}(f(n) \log f(n))$ 嚴格的大於

$\mathbf{TIME}(f(n))$ 。

和證明停機問題時一樣，我們希望可以造出一個會產生矛盾的問題 L ，而手段一樣是透過把輸出結果刻意製造成和別人不一樣。然而在這邊多了一個要考量的點：該如何知道別人的輸出？最直接的方法就是設計一個特殊的Turing machine：全能圖靈機(universal Turing machine)，universal Turing machine U 的功能就是去執行其他的Turing machine。也就是說，他的輸入就是一個Turing machine T 和那個Turing machine的輸入 x ，而輸出就是執行 $T(x)$ 的結果，也就是 $U([T], x) = T(x)$ 。然而，要製造出這樣的universal Turing machine，必須要付出一些時間上的代價，也就是可能要花比較多的時間。而目前知道最快的universal Turing machine，可以在 $f(n) \log f(n)$ 的時間執行完原本需要 $f(n)$ 時間的Turing machine[1]。我們把universal Turing machine需要多付出對數倍的現象稱為：對數支出(logarithmic overhead)。

有了universal Turing machine之後，我們就可以利用它來幫忙製造可以在 $\mathbf{TIME}(f(n) \log f(n))$ 但是不在 $\mathbf{TIME}(f(n))$ 裡面的問題了。製造的想法是利用對角化技術，這個問題 L 的輸入會是所有可以在 $O(f(n))$ 內執行完的Turing machine， L 的目的是執行 $T([T])$ 之後，把答案顛倒過來。也就是， $[T] \in L \Leftrightarrow T([T]) = No$ 。

首先我們可以知道 $L \in \mathbf{TIME}(f(n) \log f(n))$ ，因為從[1]中我們知道有一個只有對數支出的universal Turing machine，我只要利用它就可以把 $T([T])$ 算出來，也因此就可以幫 L 決定任何輸入的結果。如此一來，只要我們能夠證明 $L \notin \mathbf{TIME}(f(n))$ ，就等同於拆開(separate) $\mathbf{TIME}(f(n))$ 和 $\mathbf{TIME}(f(n) \log f(n))$ 了！

而要證明不存在性，最方便的技巧就是用謬誤法，假設 $L \in \mathbf{TIME}(f(n))$ ，也就是存在一個僅需時 $f(n)$ 的演算法 D' 可以解決問題 L 。因為 D' 只需要 $f(n)$ 的時間，所以根據 L 的定義，可以把 $[D']$ 餵給自己。此時，若 $D'([D']) = \text{Yes}$ ，則代表 $D' \in L$ ，根據 L 的定義，我們有 $D'[D'] = \text{No}$ ，產生矛盾。若結果是 $D'([D']) = \text{No}$ ，則代表 $D' \notin L$ ，根據 L 的定義，我們有 $D'[D'] = \text{Yes}$ ，還是矛盾。因此，根本不會存在演算法 D' ，也就是說 $L \notin \mathbf{TIME}(f(n))$ ！

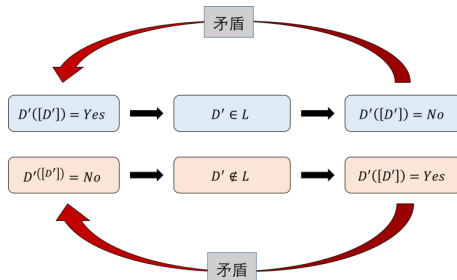


圖 3-4：如果 L 存在一個時間複雜度為 $f(n)$ 的 Turing machine，
那麼將會導致矛盾。

於是，對於任何的 $f(n)$ ，我們都可以造出一個相對應的問題 L 使得 $L \in \mathbf{TIME}(f(n) \log f(n))$ 但是 $L \notin \mathbf{TIME}(f(n))$ 。於是時間階層定理成立。

除了時間階層定理之外，對於各種不同的資源，也有相對應的階層定理，例如：空間階層定理、不確定性時間階層定理等等。透過這些階層定理，幫助我們更了解特定資源對於計算模型能力的影響。

§ 不同資源的複雜度類別關係

再有了階層定理告訴我們單一資源的影響之後，我們開始更有野心的想要知道不同資源之間的交互關係，例如：時間和空間複雜度類別的互相包含關係、時間和空間的權衡結果、電路複雜度類別之間的階層關係等等。透過了解資源之間的關聯，可以幫助我們找到最有效率的解決問題方式。

以下將介紹兩個在 Turing machine 中常見的複雜度類別關係。

- 時間與空間複雜度關係

$$L \subseteq NL \subseteq P \subseteq NP \subseteq PSPACE \subseteq EXP$$

在這邊 $L, NL, PSPACE$ 是指空間複雜度， L 代表只用到了對數空間， NL 則是多加了不確定性(non-determinism)（在附錄中會仔細介紹不確定性的概念）， $PSPACE$ 則是用到了多項式空間。

而 P, NP, EXP 都是時間複雜度， P 是我們熟悉的多項式時間， NP 是加上不確定性的多項式時間， EXP 則是指數時間。

從上面的關係式中可以發現，擁有同樣多的時間和空間，空間可以做的事情會比較多。例如： $P \subseteq PSPACE$ 。直觀上來看，這是因為可以把在時間 $f(n)$ 內做完的事情，都寫下來在同樣大小的空間中完成。由於我們尚未正式介紹空間複雜度的嚴格定義，所以在這邊就不詳細多講，只給大家一個感覺。

▪ 時間與空間的權衡

除了各自比較時間和空間的能力差別，我們也可以同時考慮擁有不同數量的時間和空間時，解決問題的能力是如何。

定義 3-5(時空權衡類別, Time/space tradeoff) : 時空權衡類別 $TISP(f(n), g(n))$ 包含了所有可以被 Turing machine 在 $f(n)$ 時間和 $g(n)$ 空間內解決的問題。

一個很有趣的結果告訴我們，

$$NP \not\subseteq TISP(n^{1.1}, n^{0.2})$$

上面這個關係可以解讀成 NP 是一個夠困難的類別，他至少比 $TISP(n^{1.1}, n^{0.2})$ 還要厲害。

§ 不同計算模型的複雜度類別關係

除了同一個模型內的複雜度關係之外，人們更渴望的是知道在不同計算模型之間的複雜度類別關係。舉例來說，電路和 Turing machine 間的關係、互動式證明和 Turing machine 的關係、Randomized Turing machine 和 Turing machine 之間的關係。透過不同模型之間複雜度關係的刻畫，可以幫助我們更加了解計算模型的能力。

以下以隨機複雜度類別為例。有之後的專題介紹中，我們將會看到更多計算模型之間的複雜度關係。

目前人們認為隨機複雜度和一般複雜度類別的關係是像下圖一樣：

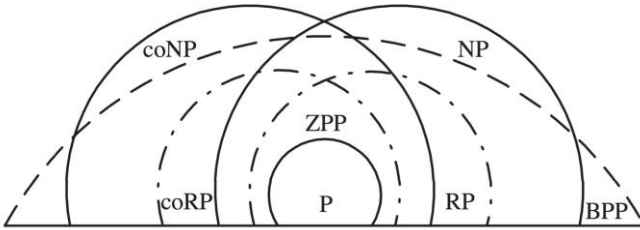


圖 3-5：隨機複雜度類別與一般複雜度類別。

用一句話總結這張圖：雖然隨機性可以增加Turing machine的能力，但是對於非確定性(non-determinism)的Turing machine來說，隨機性的影響仍然是未知。

§ 結語

本文在一開始引入了複雜度類別的概念，這是理論資訊界刻畫計算模型能力的方式。透過比較不同複雜度類別之間的大小包含關係，人們更加認識計算模型能力的來源與極限，因此可以更加有效率和善用計算模型來幫助我們解決問題。

不過使用複雜度類別來描述計算的能力其實有一個潛在的風險，因為他是一個最差情況分析(Worst-case analysis)，也就是說，通常我們會太過於保守，考慮到一些其實在現實生活中不太可能會發生的情況。這是計算理論最被實務者

(practitioner)詬病的地方。不過理論學家也不是省油的燈，有近十幾年來計算理論的發展漸漸趨向平均情況分析(Average-case analysis)或是細緻複雜度(Fine-grained complexity)，考慮的許多更貼近一般實作時會遇到的問題。但是在了解這些最新穎的研究之前，我們還是必須先熟悉傳統計算理論的內容，因此在這一系列的文章，主要會跟隨著傳統計算理論的脈絡，搭配近代貼近實務的延伸，希望可以帶給讀者扎實又不脫離現實的學習。

§ 後記

這篇文章會使用「複雜度動物園」是源自於現在任職於MIT的Scott Aaronson教授，他建立了複雜度動物園(complexity zoo)，並且擔任動物園園長，是目前關於複雜度類別最完整的資料來源之一。

本章小總結

- 計算複雜度類別
- 單一資源的複雜度類別關係
- 時間階層定理
- 同樣計算模型的複雜度類別關係
- 不同計算模型的複雜度類別關係

參考資料

[1] : Arora and Barak, 2009, Computational Complexity: A Modern Approach, Theorem 1.9.

3.1*確定性與不確定性

內容：不確定性(non-determinism)、計算組態(computing configuration)、證書(certificate)

困難度：★★★☆☆

確定性(determinism)與不確定性(non-determinism)是計算理論還有自動機理論中很重要的觀念，確定性描繪了有單純情形中我們能夠達到的計算，而不確定性則描繪了我們能夠做得最好的情形。因此比較確定性與不確定性的差別，某種程度上是在探索一個計算模型能夠達到的極限。

基本上目前為止我們討論都是確定性的計算模型，也就是說這些計算模型在執行的時候，每一步都是可以根據一開始設計好的規則預測知道。以Turing machine為例，所有的計算規則都被記錄在轉換函數裡面，只要照著上面的規則，我們就可以一步步地跟隨Turing machine做的運算直到結束。

然而不確定性的概念顛覆了這樣的思維。抽象來說，不確定性的計算模型就像是「平行世界」，會在計算的過程中產生分身，只要一個分身算出來了，計算就結束了。然而弔

詭的地方在於，在現實上其實只有一個本尊在執行，意象上我們可以把這個本尊看成是表現最好的那個分身。

§ 計算組態(Computing configuration)

第一次看到不確定性的概念，或許會覺得有點詭異，不懂為什麼要這麼樣的定義，這樣做到底有什麼好處？有更深入了解不確定性的概念之前，讓我們先認識計算組態 (computing configuration)

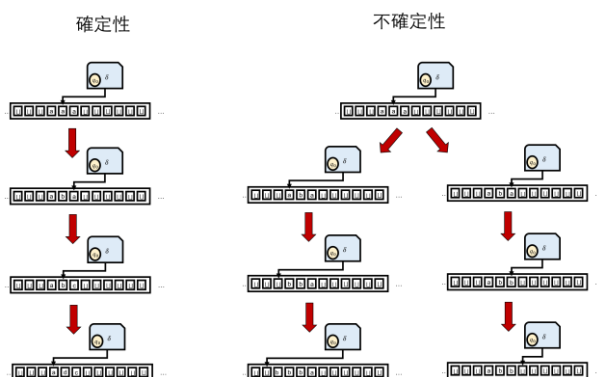


圖 3-6：確定性與不確定性。左圖是確定性的 Turing machine，可以看到他跟隨著轉換函數走著唯一的計算途徑。右圖則是不確定性的 Turing machine，在某些特別的地方，例如圖中最上面的情況，不確定性的 Turing machine 會產生分身幫忙計算。

計算組態這個名稱聽起來很學術，其實概念很簡單。一個計算模型在某個時間點的的計算組態，就是那時候整個計算模型的狀態，以Turing machine為例，就是當時的狀態還有帶子上面的符號。因此我們可以把整個計算的過程，想成是計算組態之間的交替！以圖一為例，左邊的確定性Turing machine從最上面的計算組態一步一步移動到下面的計算組態，描述了他執行時的每一個步驟。這樣的一個圖，通常會被稱作為組態圖(configuration graph)。

於是從計算組態的角度來看不確定性就很容易了。不確定性的想法就是擁有超過一個的轉換函數 δ_1, δ_2 ，一旦執行到某個計算組態使得下一步有不只一種可能時，就會同時執行這些所有的可能，於是就像圖一右邊分叉的概念。而對於每一種計算的可能，我們稱呼為一個計算途徑(computation path)。

於是直觀上來說，不確定性就像是同時使用不只一種規則，在所有可能的計算組態中探索，看看是否能找到最終成功的狀態。

§ 不確定性複雜度類別：NP

有了不確定性的計算模型之後，我們可以開始定義不確定性的複雜度類別了。讓我們從不確定性的Turing machine(Non-determinism Turing machine, NTM)開始。很直接的，我們可以根據一個問題是否可以被NTM解決來定義，但是什麼叫做被不確定性的計算模型解決呢？

從計算組態的觀點來看，可以知道一個不確定性的計算模型在執行的時候，會同時搜索超過一種的可能計算，也就是說可能會有些搜索的路徑成功，有些失敗。

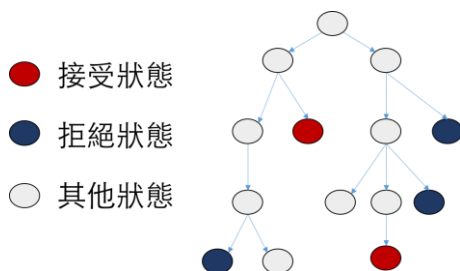


圖 3-7：NTM 的計算組態圖。紅色的點代表接受的計算組台，藍色的代表拒絕，灰色則是其他。

因為不確定性的計算模型會幫我們搜索每一個可能的計算組態，換句話說，只要有可能成功，NTM就會幫我們把這一個成功的計算找出來。於是根據這樣的想法，我們說一個

問題 P 是可以被NTM解決，代表對於任何一個輸入 x ， $x \in P$ 若且唯若，存在至少一個會成功的計算途徑，使得NTM可以走到成功的計算組態。

定義 3-6(不確定性多項式時間類別, NP)：複雜度類別NP搜集了所有可以在多項式時間之內被不確定性 Turing machine 解決的問題。換句話說，若問題 $L \in NP$ ，存在一個多項式時間的不確定性 Turing machine T 使得 $\forall x \in \{0,1\}^*$ ， $x \in L \Leftrightarrow T(x) = Yes$ 。

於是仿造定義多項式時間類別的定義方式，我們可以定義NP為所有可以被NTM在多項式時間內解決的問題。

§ 不確定性的另外一種觀點：證書(Certificate)

從計算組態的觀點，一個問題可以被不確定性的計算模型解決，代表當一個輸入是正確時，若且唯若會存在一個正確的計算途徑。換個角度想，不確定性的計算就像是在問是否存在一個正確的計算途徑！正確計算途徑就像扮演著證書(certificate)的角色，如果對於一個輸入存在著這樣的證書，那就代表這個輸入會是正確的。

於是對於NP這個複雜度類別，可以有一種不同的詮釋觀點。

NP包含了可以在多項式時間內被驗證正確與否的問題。

這裡的關鍵在於：驗證(certify/verify)。從最原始的定義，我們知道**NP**就是包含了可以有多項式長度計算路徑的問題類別。因此對於一個**NP**問題**P**，要決定輸入 x 是否在**P**裡面，等價於詢問是否存在一個多項式長度的正確計算路徑。一旦我們把正確的計算路徑當作問題的證書，那麼**NP**問題其實就是擁有一個簡單(多項式長度)證書的問題。

舉例來說，合成數問題(辨別輸入是否是合成數)就是一個**NP**問題。原因是一旦輸入 x 是合成數，代表可以把 x 寫成 $x = x_1 \times x_2 \times \cdots \times x_k$ ， $k \geq 2$ 。此時， (x_1, x_2, \dots, x_k) 就是 x 有合成數問題中的證書。因為對於一個驗證者來說，只要確認 x_1, x_2, \dots, x_k 都是正整數然後 $x = x_1 \times x_2 \times \cdots \times x_k$ ，就可以知道 x 是一個合成數了。特別注意的是，對於一個驗證者來說，她不必知道該怎麼找到 x 的分解，只要能夠驗證一個分解就可以了！

§ 把不確定性當成猜測的能力！

除了以上用比較分析的角度來看不確定性，從日常生活的直觀來看，我們可以把不確定性看成是一種猜測的能力！以圖二為例，計算是從最上面的計算組態開始，在計算的過

程中陸陸續續會有分叉。然而對於一個實際的計算模型來說，必須要在這些分叉中選一條來執行。不確定性就像是賦予了計算模型一種猜測的能力，可以在這些分叉中選到好的一條路徑，也就是通往成功計算組態的路徑！（如果存在成功計算組態的話）

在現實上不確定性的計算模型是不存在的，我們設計這樣子不確定性的概念某種程度是想要理解一個計算模型的極限在哪裡。有時候計算模型加入了不確定性之後能力沒有變強，像是有限狀態機(finite state automata)，無論是確定性或是不確定性，計算的能力都是一樣的。然而也有些計算模型加入了不確定性之後會變得更厲害，像是由上而下的樹機(top-down tree automata)，加入了不確定性後可以解決更多問題！當然也有些情況是未知的，像是最著名的 P vs. NP 問題，就是在探討對於 Turing machine 來說，加入了不確定性之後，會不會變得更厲害？

§ 結語

確定性與不確定性，目的在於刻畫只能遵守死板板規矩的機器還有具有創造力的人類之不同。在現實生活中，我們只能做出確定性的計算模型。然而有一些模型被證明出不管是否具有不確定性，能力都還是一樣的，但是對於一些常見

的計算模型，例如：**Turing machine**，不確定性造成的影響仍然是尚待被解決的問題。

了解不確定性對於計算模型的影響，某種程度上是在探索電腦計算的極限，試圖從中找出人腦具有不一樣能力的可能原因。然而這樣子的問題是極度困難的，只要能夠證明任何一個結果，都會是很大的突破。計算理論中的核心問題：***P*** vs. ***NP***，就是目前最大的一個未解之謎，在接下來幾篇文章中，我們將要探討為什麼這個問題是重要的。

本章小總結

- 不確定性(non-determinism)
- 計算組態(computing configuration)
- 證書(certificate)

3.2 相對化

內容：相對化(relativization)

困難度：★★★★☆☆

§ 類別關係

在〈複雜度類別〉一文中，我們學會了複雜度的正式定義，也了解到不同的複雜度之間會有著相交或包含的關係。類別之間的關係，直接說明了對應的計算模型強弱之分，於是該如何正確的分辨複雜度類別之間的關係成了一個重要的課題。然而要證明類別關係有時候並不怎麼單純，例如著名的 P vs. NP 問題，就是想要了解這兩個複雜度類別是包含關係還是等價關係，從問題形成到現在半個世紀左右，仍然沒有非常顯著的進展。

在這篇文章中，我們將要學習一個最基礎認識複雜度類別關係的方法：相對化(relativization)。相對化其實並沒辦法證明類別之間的關係，但是他時常能夠提供我們很好的直覺，幫助朝正確的方向努力。

在正式開始之前，先讓我們分析一下複雜度之間可能有著什麼樣的關係？如果要證明一個複雜度關係，需要說明哪

一些事情？首先，我們知道兩個集合之間的交互關係，只有四種可能：相等、包含、重疊但互不包含、沒有交集。在複雜度的世界，只有前三種有機會發生，因為任何有意義的複雜度類別，都可以處理一些最基本的問題，於是不會發生兩個複雜度類別能夠解決的問題完全沒有交集。因此，當考慮兩個複雜度類別**A**和**B**時，可能的關係有 **$A = B$** 、 **$A \subsetneq B$** 、 **$B \subsetneq A$** 或是**A**和**B**相交但互不包含。

要去證明兩個複雜度類別之間的關係，可以先從最簡單的單向關係出發，也就是先確定到底是 **$A \subseteq B$** 還是 **$A \not\subseteq B$** 。一旦把兩個方向的資訊都確定後，就可以很明確地知道**A**和**B**之間的關係了。

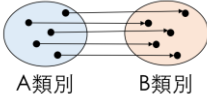
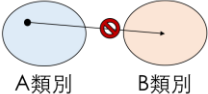
目標	A類別 \subseteq B類別	A類別 $\not\subseteq$ B類別
方法	<div>$\forall P \in \text{A類別}, P \in \text{B類別}$</div>  <div>A類別 B類別</div>	<div>$\exists P \in \text{A類別}, P \notin \text{B類別}$</div>  <div>A類別 B類別</div>

圖 3-8：如何證明複雜度類別之間的關係。

然而說來容易，實際做起來很困難，目前能夠證明單向關係的方法仍然非常陽春，透過集合的語言做論述，偶爾搭配上一些技巧，例如之前學到的對角化方法。因此大方向來

看，要證明 $A \subseteq B$ ，就是要去論述任何 A 可以解決的問題 B 都可以解決。而如果要證明 $A \not\subseteq B$ ，則是要尋找的一個 A 可以解決但是 B 無法的問題。上述的概念可以從圖 3-8 中認識了解。

§ 相對化(Relativization)

在證明複雜度類別的關係時，常常讓人不知道從何下手，到底是 A 類別被 B 類別包含還是相反呢？這時候，人們通常會透過相對化(relativization)的方式來協助推測複雜度類別之間的關係。直觀的概念是這樣，對於這兩個複雜度類別對應的計算模型，同時加上任何一個功能，此時再來比較兩個加上功能後的計算模型之間的相互關係。很直覺的我們會覺得如果再加上新功能後 $A \subseteq B$ ，那麼在原本的情況時，很有可能也是 $A \subseteq B$ ，於是我們就可以朝這個方向努力。這樣把兩個計算模型都加上新功能的概念，就被稱為相對化。

但是什麼叫做加入新功能？相對化後的結果一定都是對的嗎？就讓我們接著看下去吧！

首先我們會使用一種被稱為神諭(oracle)的方式來幫計算模型加入新功能。oracle 其實就是個黑盒子函數，計算模型不用知道怎麼樣子算這個函數的值，只要對 oracle 做詢問(query)就好了，而且為了方便，我們會讓對 oracle 的詢問不會造成計

算模型的額外負擔，也就是說每一次算函數的值都不需要付出額外的代價！

神諭在直觀上的概念，就像是提供一個原本比較弱的計算模型一個很強的功能，可以代替他處理一個很困難的問題。舉例來說，如果我們把質數問題當成是一個神諭，交給一個計算模型。原本這個計算模型在判斷質數的時候，可能需要花到多項式時間甚至是指數時間，然而有了質數神諭之後，他變成可以在常數時間內判定一個數字是不是質數。如此一來大幅減少了這個計算模型的計算時間，某種程度上讓他的能力變強了！

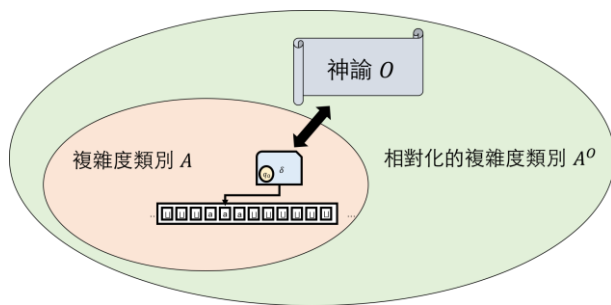


圖 3-9：擁有神諭的複雜度類別。當我們讓原有的計算模型可以在僅付出常數代價的情況下，使用黑盒子計算某個函數，則我們稱之為神諭計算模型，所對應的複雜度類別則是神諭複雜度。

在符號上，對於一個複雜度類別 C ，加入了oracle O 之後產生的新複雜度類別就是 C^O 。使用這樣的符號，相對化的意思其實就是考慮對於任何可能的 O ，兩個複雜度類別 A 和 B ，有沒有 $A^O \subseteq B^O$ 或者是 $B^O \subseteq A^O$ 。

於是當人們想不到該怎麼證明兩個複雜度類別之間的關係時，很常見的一個做法就是來考慮看看相對化之後的關係會是如何，再從結果中猜測真實的關係，並且努力證明出來。然而，這時候很重要的一個關鍵就是，相對化的結果和原本真實的結果會一致嗎？有沒有相對化無法證明的結果呢？很遺憾的是，兩個問題的答案都顯示了相對化方法沒辦法被應用在所有的情况。

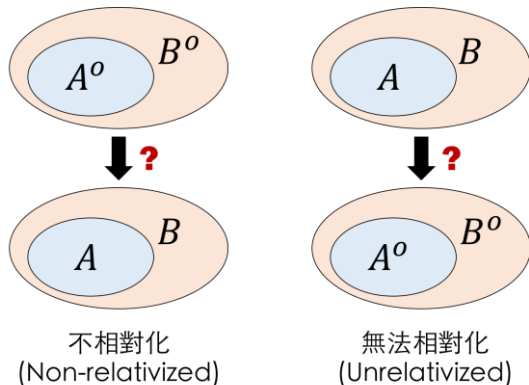


圖 3-10：相對化的失敗。不相對化與無法相對化。

- 相對化的結果和原本真實的結果會一致嗎？

並不會。存在不相對化(non-relativized)的複雜度關係！

在舉例之前，先讓我們了解不相對化的意思。一個複雜度關係，例如 $A \subseteq B$ ，是不相對化的，代表存在一個神諭 O ，使得 $A^O \not\subseteq B^O$ 。也就是說加入了某一個功能後，改變了兩個複雜度類別原本的相對關係。這樣代表了我們無法從相對化的結果直接推論出真實的結果！

第一個不相對化的結果是在1992年互動式證明興盛時期出現的，Shamir在[1]中證明了 $PSPACE = IP$ ，但是[2]卻證明對於隨機的神諭(random oracle) R ，會有 $PSPACE^R \neq IP^R$ 。也就是說隨便拿一個oracle，都會改變 $PSPACE$ 和 IP 的關係！由此可知，相對化得到的結果只能僅供參考，並不能當作實際的證明方法。

- 有沒有相對化無法證明的結果呢？

有。存在無法相對化(unrelativized)的複雜度關係！

無法相對化的意思是說，一個複雜度關係，例如 $A \subseteq B$ ，可以找到兩個oracles O_1 和 O_2 使得 $A^{O_1} \subseteq B^{O_1}$ 但是 $A^{O_2} \not\subseteq B^{O_2}$ 。也就是說，兩個oracles得出來的複雜度關係是不同

的！於是根本沒辦法有一個相對化的結論，因為加入不同的新功能可能得到不同的結果！

第一個例子是Baker, Gill, Solovay在1975年[3]針對 P vs. NP 問題得到的結果。他們找到兩個oracles O_1 和 O_2 ，使得 $P^{O_1} = NP^{O_1}$ 但是 $P^{O_2} \neq NP^{O_2}$ 。這樣的結果粉碎了使用可相對化工具來證明 P vs. NP 的可能性。因為所謂可相對化工具，就是指這些工具得到的結果都不會受到相對化的影響，例如對角化技術即是可相對化工具的最佳代表。Baker, Gill, Solovay的結果等同於宣告使用對角化技術證明 P vs. NP 的死刑！

§ Baker, Gill, Solovay

在上一段中，提到了Baker, Gill, Solovay三人在1975年證明了 P vs. NP 的關係是無法相對化的，因此粉碎眾人對於相對化的依賴，了解到相對化方法只能僅供參考，並不能當作正式的證明，更不見得能夠帶給我們正確的猜想。接下來，就讓我們來看看這三人當初到底是怎麼證明 P vs. NP 無法相對化的，希望可以從中對於相對化有更深刻的了解。首先讓我們正式把定理陳述一次。

定理 3-2(Baker, Gill, Solovay, 1975) : 存在兩個神諭 O_1 和 O_2 , 使得 $P^{O_1} = NP^{O_1}$ 但是 $P^{O_2} \neq NP^{O_2}$ 。

這個定理包含了兩個部分，第一個部分要找到一個神諭，使得 P 和 NP 拿到後會擁有一樣的能力。第二個部分則是要找到一個神諭，使得 NP 拿到之後可以嚴格的比 P 還要厲害。

▪ $P^{O_1} = NP^{O_1}$

這個方向是比較簡單的部分，只要選取 O_1 為一個 NP 完全(NP -complete)的問題 O_1 就好了。意思就是說 O_1 可以在多項式時間內幫忙解決任何 NP 之內的問題，於是有了它之後， P 就和 NP 有一樣的能力了。因為我們尚未進入關於簡約方法(reduction)的介紹，所以目前讀者可以先用這樣的直觀來理解這個方向即可。

▪ $P^{O_2} \neq NP^{O_2}$

要證明這個方向，我們必須要找到 O_2 以及一個問題 L_{O_2} ，使得 $L_{O_2} \notin P^{O_2}$ 但是 $L_{O_2} \in NP^{O_2}$ 。Baker, Gill, Solovay的作法十分巧妙，主要的觀察是在於任何一個多項式時間的Turing machine T ，最多只能向神諭呼叫多項式次，也就是說我們可以找個一個輸入長度 n_T ，使得在 $\{0,1\}^{n_T}$ 當中，大部分可能輸

入都沒有被 M 使用到。於是對於一個問題，我可以利用那些沒有被 M 使用到的輸入，來製造讓 M 會答錯的結果！

在這邊，我們選擇了一個看似很簡單，但其實並不單純的問題：

$$L_{O_2} = \{n \in \mathbb{N} \mid \exists x_n \in \{0,1\}^n, O_2(x_n) = Yes \}$$

L_{O_2} 這個問題想要問的是，有哪些長度的輸入是有正確輸入的？在這裡使用 L_{O_2} 是一個非常聰明的想法，因為這樣可以成功地讓原本是在二進位的問題 O_2 膨脹變成一進位的問題 L_{O_2} ，如此一來可以利用到 P 只能做多項式次運算但是 NP 可以猜測的特性。

做完這樣的設定後，接下來重要的關鍵就是要找到那個神秘的 O_2 ，使得從 O_2 延伸出來的 L_{O_2} 會不在 P 但是在 NP 裡面。而我們的想法很簡單，就是仿造對角化技術的方式，把所有Turing machine列舉出來，並且故意讓每個Turing machine都在至少一個輸入上和 L_{O_2} 的正確答案不一樣！

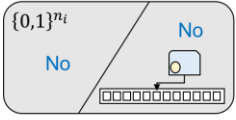
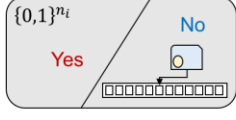
$T_i^{O_2}(n_i)$	O_2	L_{O_2}
Yes		No
No		Yes

圖 3-11：如何建造拆散 P 和 NP 的神諭以及對應的問題。當第 i 台 Turing machine 對於輸入 n_i ，的結果是 Yes 時，我們希望讓 n_i ，在問題的答案中是錯的。當第 i 台 Turing machine 對於輸入 n_i ，的結果是 No 食，我們希望讓 n_i ，在問題中的答案是對的。如此一來，問題就和第 i 台 Turing machine 在 n_i ，上面有不一樣的結果。

從圖三中，我們可以更加知道該如建造 O_2 和 L_{O_2} 。對於每一個 Turing machine $T_i^{O_2}$ 來說，我們希望找到一個 n_i ，讓 $T_i^{O_2}$ 和 L_{O_2} 在 n_i 上有不一樣的結果。選這個 n_i 的方法可以很暴力，只要符合下面兩個條件就好：

確定 n_i 夠大使得 $T_i^{O_2}$ 呼叫神諭的次數不超過 2^{n_i} 的一半，也就是有一半長度為 n_i 的輸入都沒有被 $T_i^{O_2}$ 問到。

在 O_2 中，所有長度為 n_i 的輸入對應的輸出值都還沒有確定，如此一來我們才能針對 $T_i^{O_2}$ 對 n_i 的輸出來決定 O_2 的輸出要用什麼。

當 n_i 能滿足這兩個條件，我們就可以根據圖三的方法來決定長度為 n_i 的輸入在 O_2 中的輸出是什麼。

- 當 $T_i^{O_2}$ 對於 n_i 的回答是Yes時：我們希望 L_{O_2} 中不含有 n_i ，於是把 O_2 中所有長度為 n_i 的輸入都設成No。
- 當 $T_i^{O_2}$ 對於 n_i 的回答是No時：我們希望 L_{O_2} 中含有 n_i ，於是把一個沒有被 $T_i^{O_2}$ 向神諭問過且長度為 n_i 的輸入在 O_2 中設為Yes。

如此一來，我們就可以成功的構造出一個無法被 $T_i^{O_2}$ 解決的問題！

最後，因為不確定性的Turing machine有猜測的能力，所以可以輕鬆地解決 L_{O_2} ，因此我們有：

$$L_{O_2} \notin P^{O_2}$$

$$L_{O_2} \in NP^{O_2}$$

因此， $P^{O_2} \not\subseteq NP^{O_2}$ 。

§ 結語

了解複雜度類別之間的關係一直以來都是理論資訊學家追求的梦想，被視為理論聖杯的 **P** vs. **NP** 問題，就是在問兩個複雜度類別的關係到底是等價還是分開(separation)。然而受限於工具的有限，大部分類別之間的關係都是讓人很難直接處理。於是相對化的概念相應而出，透過幫兩個計算模型同時增強功能，再來比較相對關係，希望從中可以獲取一些感覺。然而相對化方法終究只是一個輔助的工具，從不相對化和無法相對化的例子，可以知道我們沒辦法依賴相對化方法得到確切的複雜度類別關係。

從對於相對化方法的認識，我們可以感受到複雜度類別之間的關係並不是那麼的直接，幫計算模型加了功能之後，有時候會改變關係，有時候則不會。以下引用[4]中描述相對化的一段話，十分精準的刻畫相對化帶來的意義，送給大家當作帶回家的消息(take-home message)。

Oracles do not relativize complexity classes, they only relativize the machine.

神諭無法相對化複雜度類別，他們只會相對化計算模型。

- [4]

本章小總結

- 相對化(relativization)
- 不相對化(non-relativized)
- 無法相對化(unrelativized)

參考資料

- [1] Adi Shamir. $P = PSPACE$. Journal of the ACM (JACM), 39(4):869–877, 1992.
- [2] : Richard Chang, Benny Chor, Oded Goldreich, Juris Hartmanis, Johan Håstad, Desh Ranjan, and Pankaj Rohatgi. The random oracle hypothesis is false. Journal of Computer and System Sciences, 49(1):24–39, 1994.
- [3] : Theodore Baker, John Gill, and Robert Solovay. Relativizations of the $P = ? NP$ question. SIAM Journal on computing, 4(4):431–442, 1975.
- [4] : Juris Hartmanis, Richard Chang, Suresh Chari, Desh Ranjan, and Pankaj Rohatgi. Relativization: A revisionistic retrospective. In Bulletin of the EATCS. Citeseer, 1992.

3.3 簡約方法與完全性

內容：困難度下界(Hardness lower bound)、簡約方法(Reduction)、完全性(Completeness)

困難度：★★★★☆☆

複雜度類別把一個計算模型在某種資源分配狀態下可以解決的問題聚在一起，某種程度上刻畫了問題的難易之分。需要越多資源才能解決的問題被視為比較困難的，反之則是比較簡單。透過實際演算法的設計，我們可以知道一個問題是不是落在某個複雜度類別裡面，例如：我們會在多項式時間內解決質數問題，那麼質數問題就會是在P裡面。然而另外一個方向就比較麻煩了，要如何說一個問題不在某個複雜度類別裡面呢？這樣子探討問題是否不在某個複雜度類別的概念被稱為困難度下界(hardness lower bound)，也就是想要證明一個問題是至少有多困難的，例如：單調函數(monotone function)是不存在多項式大小的電路。

§ 困難度下界(Hardness lower bound)

最直接照著定義的方法，困難度下界的目的就是證明一個問題不可能有某種類型的演算法，然而這光想就很可怕，

直接暴力的處理要說明非常非常多潛在的演算法是無效的。這樣子很有氣魄直接硬碰硬的方式，通常被稱為非條件性方法(unconditional method)，也就是說這種方法不需要做任何的假設或猜想，可以直接的說明一個問題不可能擁有某種類型的演算法。這樣的方法時常在電路(circuit)中出現，人們會把某個大小所有可能的電路都弄出來，然後說無法解決目標的問題，於是這個問題需要更大的電路，因此替問題找到了一個困難度的下界。

但是通常要使用非條件性方法是不太容易的，需要在很方便控制大小的計算模型上才比較有可能成功，目前幾乎沒有證明出什麼非常具有突破性的困難度下界。於是人們往另外一個方向嘗試：條件性方法(conditional method)。

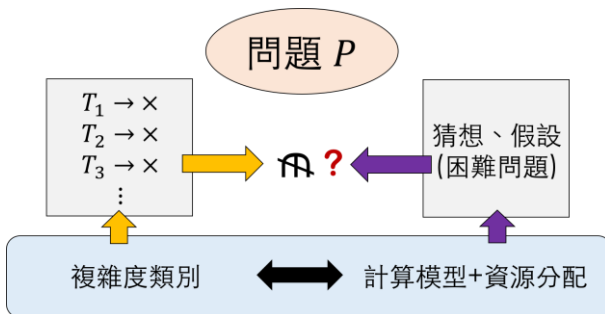


圖 3-12：證明困難度下界的方法。黃色的路徑是非條件性方法，紫色的路徑是條件性方法。

顧名思義，條件性方法的證明過程依賴著一個條件，也就是「假說」或「猜想」。在基於相信假說與猜想會成真的情況之下，我們利用假說與猜想帶來的結果，推論出其他問題或是複雜度類別的困難度下界。用一個生活化的例子來想，現在我們想要證明很困難的事情是：「台大醫學系是個很難考上的科系」。使用非條件性的方法來證明這個事情的困難性，我們可以列舉歷年台大醫學系的分數，因此推論說這件事情的確很困難。然而當這些資料沒有用的時候，可以如何說明台大醫學系很難考呢？假設現在有個大家公認都可能成立的假說是：「台大是間很難考的學校」。於是我們可以由「因為台大醫學系在台大裡面」，加上假說後推論：「台大醫學系是個很難考上的科系」。雖然同樣的我們其實並不知道這個假說的正確性，可是根據大家對他的相信，我們可以把這個相信推廣到對於台大醫學系很難考上的這個猜想，因此根據至少台大醫學系至少不會比台大好考，來相信台大醫學系是難考上的。

這樣子把未經嚴格證明的困難性猜想，延伸到其他困難性保證的方法，就被大家稱為條件性方法。而傳統中條件性方法中的佼佼者就是這次的主角：簡約方法(Reduction)。

§ 簡約方法(Reduction)

簡約方法是計算理論界最常見的非條件性困難度下界方法，它的核心概念是避開直接說一個問題是困難的，而是改去說一個問題不會很簡單。用一句話來形容簡約方法，那就是：

一個問題的解決方法可以幫忙處理其他問題的話，他至少不會太簡單

於是，想要說明一個問題不是簡單的，其實等同於說當你會解這個問題之後，就有能力解很多其他的問題了！以兩個問題之間的困難度為例，如果任何會解問題 A 的人都會解問題 B ，那麼換個角度來想，其實就是說明問題 A 不會比問題 B 簡單！

讓我們用一個例子來理解上面的抽象概念。假設問題 A 是一個高中的數學題目，問題 B 是一個國中的數學題目。就常理而言，會解問題 A 的人，基本上都會解問題 B ，否則他應該是猜對的，不然怎麼可能會高中數學但是不會國中數學？於是，根據會解問題 A 代表會解問題 B ，我們可以有一種感覺：問題 A 至少不會比問題 B 還要簡單。

但是，我們該如何說明會解 A 就會解 B 的這個過程呢？換個角度想，可以把上面的過程，看成是我們可以把問題 B 請會解問題 A 的人來幫忙解決。更進一步，也可以想成我們可以把問題 B 包裝成問題 A 的樣子，讓會解問題 A 的人來解決！接續國中數學題目的例子，我們想要請會解高中數學題目的人來解國中數學題目，最簡單的方式就是把每個國中數學的題目，都改寫成高中數學題目的格式。於是，會解高中數學的人，看到這個改寫過後的題目之後，解可以輕鬆解決，於是就把國中數學題目解開了！這樣子包裝、改寫的過程，就是簡約方法！

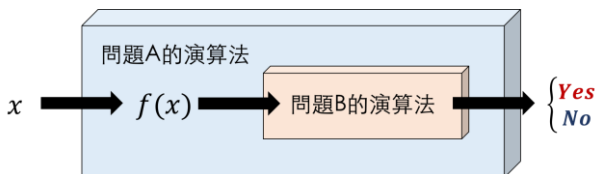


圖 3-13：簡約方法。當問題 A 可以被簡約到問題 B ，代表我們可以把 A 的輸入 x ，轉換成問題 B 的輸入 $f(x)$ ，並且讓問題 B 的演算法來幫忙解決。如此一來透過問題 B 的演算法，構造出一個問題 A 的演算法。

正式來說，問題 A 可以被簡約到(be reducible to)問題 B ，代表存在一個方式，可以把問題 A 的題目轉換成問題 B 的題目，請會解問題 B 的人來解決！

定義 3-7(簡約方法 Reduction)：當存在一個函數 $f: \{0,1\}^* \rightarrow \{0,1\}^*$ 使得 $\forall x \in \{0,1\}^* x \in A \Leftrightarrow f(x) \in B$ ，我們稱問題 A 可以被簡約到(be reducible to)問題 B 。符號上我們寫做 $A \leq_r B$ 。

§ 利用簡約方法來建立困難度下界

當問題 A 可以被簡約到問題 B 時，意味著問題 B 不會比問題 A 還要簡單。於是當我們想要說明一個問題很困難的時候，可以把這個問題簡約到一個被公認是很困難的問題，因此就可以知道這個問題是困難的！也許大家還沒有太大的感覺，那麼現在就讓我們來看一個實際由簡約方法建立困難度下界的例子吧！

例子2：空問題是不可決定的

我們定義「空問題(Empty problem)」如下：

定義 2(空問題, Empty problem) : 空問題的輸入是一個描述Turing machine的字串 $[T]$ ，空問題接受 $[T]$ 若且唯若對於任何Turing machine的輸入 $x \in \{0,1\}^*$ ， $T(x) = No$ 。也就是說

$$EMPTY = \{[T] | T(x) = No, \forall x \in \{0,1\}^*\}$$

經過一些觀察之後，我們可以發現，要知道一個Turing machine是不是對於所有輸入都是回答 No 其實必須要把所有的輸入結果都看一遍。某種程度來講，這是一件蠻困難的事情，於是，很直接的我們可能會猜測要解決空問題並不是那麼的容易。然而要直接證明空問題很難實在有點令人難以下手，那不如讓我們來試試看用簡約方法來證明空問題是很難的吧！

目前我們知道停機問題是一個不可決定的問題，那麼不如試試看把停機問題簡約到空問題，如此以來，就可以說明空問題不會比停機問題簡單，因此空問題一定也是不可決定的！（請把這一段的邏輯搞清楚後再繼續往下看）

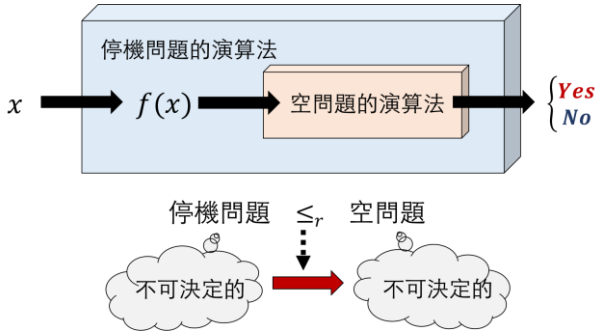


圖 3-14：把停機問題簡約到空問題。如果我們把停機問題簡約到空問題，那麼就代表空問題不會比停機問題簡單，因此也是不可決定的。

有了這樣簡約方法的想法之後，接下來就是構造簡約的過程囉！我們要做的事情，就是要將停機問題的一個輸入 $([T], x)$ 轉換成空問題的輸入 $[T_x]$ ，並且確保當 $([T], x) \in \text{HALT}$ 時若且唯若 $[T_x] \in \text{EMPTY}$ 。此時的轉換過程，其實就會是圖三中的函數 f 。以下將提供一個轉換的方式，並不會是唯一的，讀者可以嘗試看看可否構造出更簡潔的方法。

簡約過程：對於停機問題的輸入 $([T], x)$ ，我們定義一個新的 Turing machine T_x 如下：面對輸入 $y \in \{0,1\}^*$ ， $T_x(y) = \text{Yes}$ 若且唯若 $T(x)$ 停下來。令這個 T_x 為簡約到空問題的數入。

於是我們可以發現，當 $T(x)$ 會停下來時， T_x 會接受任何輸入，因此 $[T_x]$ 並不會在空問題之內。而當 $T(x)$ 不會停下來時， T_x 會拒絕任何輸入，因此 $[T_x]$ 會在空問題之內。由此可知， $([T], x) \in HALT$ 若且唯若 $[T_x] \in EMPTY$ ，也就是我們可以把空問題簡約到停機問題！

從簡約方法的精神來看，我們可以知道空問題不會比停機問題難，所以他至少是不可決定的。這也可以利用謬誤法來驗證：假設存在一個Turing machine可以決定空問題，則根據上述的簡約過程，這個Turing machine也可以幫忙決定停機問題。然而，根據之前的對角線方法，我們知道停機問題是不可決定的，於是這樣的Turing machine必定不存在，也就是說空問題也是不可決定的。

§ 完全性(Completeness)

有了簡約方法的概念之後，接下來我們就可以開始使用它來幫助我們了解問題之間的困難關係了。首先我們在意的可能會是在同一個複雜度類別之中，問題的難易差別。以 NP 這個不確定性多項式時間複雜度為例，雖然裡面的問題都可以在多項式時間內驗證，但是有些已經有快速的演算法，有些仍然只有指數演算法，我們該如何說明這些問題的困難度也有差異之分？

這時候完全性(Completeness)的概念就誕生了！完全性想要描述在同一個複雜度類別之中，最困難的問題。而捕捉最困難問題的方式就是透過簡約方法，一旦所有在複雜度類別 \mathbf{C} 之中的問題，都可以被簡約到問題 A ，那麼我們就會說問題 A 是一個 \mathbf{C} -完全問題，也就是說問題 A 是複雜度類別 \mathbf{C} 中最困難的問題之一！

定義 3-8(完全性 completeness)：對於複雜度類別 \mathbf{C} 來說，當以下兩件事情成立時，問題 A 是一個 \mathbf{C} -完全問題。

- $A \in \mathbf{C}$ 。
- $\forall B \in \mathbf{C}, B \leq_r A$ 。

在這邊讓我們再仔細想想看完全性帶來的涵義是什麼。在完全性的定義當中，有兩個主角，一個是複雜度類別 \mathbf{C} 一個是在 \mathbf{C} 裡面的問題 A 。我們說問題 A 是 \mathbf{C} -完全當所有在 \mathbf{C} 裡面的問題都可以被簡約到 A ，換句話說就是 A 不會比 \mathbf{C} 裡面任何一個問題還要簡單！以複雜度類別 \mathbf{NP} 為例，當我們說一個問題是 \mathbf{NP} -完全(\mathbf{NP} -complete)的時候，就代表這個問題會是 \mathbf{NP} 中最困難的問題！

然而，說明一個問題是 \mathbf{C} -完全的，其實只是說明了他會是在複雜度類別 \mathbf{C} 裡面最困難的問題，我們並不能從這個結果知道他和其他不在這個複雜度類別的問題之間的難易關係。

像是目前大家對於***NP-complete***的問題和其他複雜度類別之間的關係仍然掌握的不是很清楚，要說明***NP-complete***的問題是真的很困難的，人們還是必須要對於***NP***這個複雜度類別有完整的了解。換句話說，想要說明***NP-complete***的問題非常難，其實就是必須證明***P***不等於***NP***。

§ 結語

在計算理論的領域中，要說明一個問題的困難性是不容易的。透過簡約方法，我們可以把問題的難度簡約到另外一個問題上面，因此只要被簡約的問題是困難的，簡約過去的問題也會跟著是困難的。

而當我們要討論複雜度類別的困難度時，則可以透過完全性的觀點。一個***C***-完全性的問題，會是複雜度類別***C***中最困難的問題，於是要討論複雜度類別的困難度，其實只要考慮完全性的問題就好了。因此，要解決人們最在意的***P*** vs. ***NP***問題，真正要面對的，就是那些***NP-complete***的問題。而目前人們知道的***NP-complete***問題有非常多，有些很常見，有些則是非常隱晦。在接下來的補充文章中，我們將會一窺***NP-complete***的奇幻世界。

本章小總結

- 簡約方法(reduction)
- 完全性(completeness)

3.3* 簡約方法實戰演練

內容：布林可滿足性問題(Boolean satisfactory problem)、庫克
• 拉文因定理(Cook-Levin Theorem)

困難度：★★★★☆

在〈簡約方法與完全性〉中，我們學會了如何透過簡約方法(reduction)來說明問題之間的困難度關係。同時，我們也看到如何定義出完全性(completeness)的概念，把一個複雜度類別中最困難的問題找出來。

然而，在之前的討論中，是屬於高層次的定義，我們並未實際做出一個正式的reduction，也沒有證明出一個complete的問題。在這一章節，我們將要實際的操作簡約方法，看看第一個**NP-complete**的問題是如何被找出來的。此外，也將看到如何利用這一個結果來推導出更多**NP-complete**的問題。

一旦知道問題是**NP-complete**後，某種程度上可以相信他是一個困難的問題，畢竟如果可以很快(多項式時間內)的解開一個**NP-complete**的問題，就等於可以把所有**NP**的問題都快速解決。雖然人們還無法排除**NP**也許是個簡單的複雜度類別這件事情，但是種種跡象都顯示**NP**不太可能是簡單的。於

是建立在相信**NP**不是簡單的信仰之下，我們可以更進一步相信**NP-complete**的問題是不簡單的。

那就讓我們看看該如何證明一個問題是**NP-complete**的吧！

§ 熱身：布林可滿足性問題

再推導第一個**NP-complete**問題的reduction之前，我們必須先認識一下問題本身，如此一來才能更清楚地瞭解未來證明的過程。這一個重要的問題就是：布林可滿足性問題 (Boolean satisfactory problem)。

在正式定義這個問題之前，先讓我們迅速地複習一下布林代數 (Boolean algebra)。布林代數要處理的是只會需要兩種答案的任何運算，最常見的就是判斷事情的對錯。

在布林代數中，所有的元素都只會是1或是0，也可以想成是對(true)或錯(false)。透過三個運算子：且(\wedge)、或(\vee)和反(\neg)，結合變數，形成布林式子 (Boolean formula)，如此一來當帶入了變數的值之後，就可以透過運算子的規則算出式子的輸出是什麼。舉例來說，以下是個簡單的Boolean formula。

$$\phi(x, y) = (x \vee y) \wedge (\neg x \vee \neg y)$$

對於 $\phi(x, y)$ 來說，有四種可能的輸入： $(x, y) = (0, 0), (0, 1), (1, 0), (1, 1)$ 。而得到的輸出則分別會是 false, true, true, false。不知道讀者有沒有發現，其實 ϕ 在這邊就是扮演了互斥或閘(Xor gate)的角色！而在這邊，我們把可以讓Boolean formula取值為true的輸入稱為滿足輸入(satisfying assignment)。

更進一步，我們可以發現，任何的布林函數(Boolean function)，也就是長得像是 $f: \{0, 1\}^n \rightarrow \{0, 1\}$ 的函數，都可以寫成Boolean formula的樣子！聽到這邊，也許你會想說，這樣不是太好了，只要會快速的處理Boolean formula，那不就等於可以把任何的問題都快速的解決了嗎？然而很可惜的是，Boolean formula並非都是很好解決的，某些很複雜的Boolean function對應到的Boolean formula光是大小就是指數層級(對於輸入長度 n 來說)，也就是只是單純取這個Boolean formula的值就需要指數層級的時間了！

因此，一個我們退而求其次，先看看那些比較小的Boolean formula，也就是大小為多項式層級的式子，至少這些式子可以很迅速的取值。於是，拉回我們習慣的決定性問

題，可以定義布林可滿足性問題(Boolean satisfiability problem)如下。

定義 3-9 (布林可滿足問題, Boolean satisfiability problem)：布林可滿足問題SAT 搜集了所有擁有滿足輸入且長度為多項式的 Boolean formula。換句話說， $SAT = \{\phi(x_1, \dots, x_n) \mid |\phi| = \text{poly}(n), \exists x_1^*, \dots, x_n^* \text{ s.t. } \phi(x_1^*, \dots, x_n^*) = \text{true}, \forall n \in \mathbb{N}\}$ 。

首先，我們可以發現SAT會是一個**NP**-complete的問題，因為我們可以在多項式時間內驗證一個輸入是否是滿足輸入。然後SAT是否為**P**就是一個未知的問題，目前大家都相信**P**應該不在**P**裡面，更進一步，接下來的定理將會證明SAT是**NP**-complete的，也就是說 $SAT \in P \Leftrightarrow P = NP$ ！

§ 第一個NP-complete問題：Cook-Levin Theorem

從完全性的定義出發，我們知道想要證明SAT是**NP**-complete，必須對每一個**NP**問題都建構出一個簡約過程到SAT。換句話說，就是要利用SAT來幫忙解決所有的**NP**問題！

乍聽之下要做到這件事情非常的困難，不過其實雖然略為複雜，但是這一個簡約過程的建造其實非常直接，很容易理解。

開始之前，先讓我們觀察一下**NP**裡面的問題有什麼共同的特徵。回顧一下之前對於不確定性(nondeterminism)還有 Turing machine 的介紹，根據定義，一個**NP**問題 L 可以在多項式時間內被不確定性的Turing machine解決。而依照不確定性的想法，其實就是在說當輸入 x 在 L 裡面時，會存在一個正確的計算途徑(computing path)，讓解決問題的Turing machine順利輸出yes。

因此，對於**NP**問題 L 來說，原本的問題是輸入 x 是否在 L 之中，現在可以等價於問：是否存在一個 x 的正確計算途徑？

有了這樣的概念之後，事情就撥雲見日了。因為整個計算途徑都會被記錄在Turing machine上面，而且計算途徑中的每一步都必須滿足Turing machine原先的規則。因此，我們可以某種程度上把Turing machine帶子上的每個格子在每個時間點下看成一個變數，接著利用Boolean運算子來檢查這些格子有沒有遵守規則，並且最終走到接受的狀態。也就是說，我們可以構造出一個Boolean formula ϕ_x ，使得 $x \in L$ 若且唯若

ϕ_x 是可以被滿足的。（ ϕ_x 可以被一個正確的計算途徑滿足！）

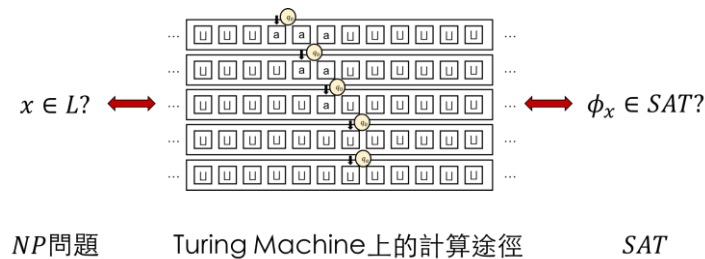


圖 3-15：Cook-Levin 定理的整體概念。將任何的 NP 問題，轉換成詢問是否存在一個正確的計算途徑，再接著把這個問題用 Boolean formula 呈現出來，因此變成一個 SAT 問題。

跟隨上述的簡約概念，Stephan Cook [1]和Levin Leonid [2]在1970年代各自獨立的提出了SAT是**NP-complete**的想法，現在被人們合稱為Cook-Levin定理。

定理 3-3 (Cook-Levin Theorem) : SAT 是**NP-complete**。

Cook-Levin定理的核心精神就如同圖一所示，將任何的 **NP**問題使用簡約方法，想成是在尋找正確的計算途徑。接著再簡約到一個Boolean formula，變成了SAT問題。如果讀者有興趣深入瞭解，可以自行在網路上搜尋來自全世界優秀課程的教材。

§ Karp's 21 NP-Complete問題

Cook在1971年提出SAT是NP-complete的結果後，馬上獲得理論資訊界的重視，人們也開始思索，到底有哪些問題會是NP-complete的，也就是說，有哪些問題會是NP中最困難的問題？

也許乍聽之下，會覺得在NP中最困難的問題應該不多，能夠找到SAT也只是剛好而已罷了。不過令人意外的是，在1972年Richard Karp轟動一時的經典論文：“Reducibility Among Combinatorial Problems” [3] 告訴所有理論學家，其實有非常多的問題是NP-complete的！在這篇論文中，他整理了21個常見的問題，並且使用簡約方法證明他們全部都是NP-complete的。例如在介紹圖論時提到的點團問題(Clique)、Hamiltonian path問題，或是資訊系學生很熟悉的背包問題(Knapsack)等等。Karp利用各種漂亮的簡約方法，告訴嘗試多年的演算法學家，如果你可以替這些常見的NP-complete問題找到快速的演算法，就等同於找到了一個快速的全能(universal)演算法！

在這邊，我們將以點團問題(Clique)為例，感受一下簡約方法的美妙。首先讓我們回顧一下點團問題是什麼？

問題（點團問題, Clique）：給定一個圖 G 和整數 k ，請問 G 裡面是否有一個大小為 k 的點團？

要證明點團問題是**NP-complete**的，我們必須將一個已知為**NP-complete**的問題簡約至點團問題。而在這邊我們要使用一個SAT的變形：**3SAT**，作為我們簡約的基礎。在定義**3SAT**之前，我們需要先認識什麼事conjunctive form。

定義 3-10 (conjunctive form, CNF)：一個 Boolean formula ϕ 如果是 conjunctive form，則 ϕ 的形式為 $(\cdot \vee \cdots \vee \cdot) \wedge (\cdot \vee \cdots \vee \cdot) \wedge \cdots \wedge (\cdot \vee \cdots \vee \cdot)$ 。

舉例來說， $\phi(w, x, y, z) = (\neg x \vee y \vee z) \wedge (\neg z \vee y) \wedge (w \vee \neg x)$ 。其中，一組括號我們稱之為句子(clause)而一個conjunctive form的每個句子長度都為 k ，則我們稱呼它是一個**kCNF**。於是，我們可以定義**3SAT**如下。

問題（3SAT）：給定一個**3CNF** ϕ ，請問 ϕ 是否可以被滿足？

而從上述SAT是**NP-complete**的結論中，經過一些布林代數上的操作，我們可以同樣的證明**3SAT**也是**NP-complete**的。但是這個過程稍微瑣碎無味一些，比較不適合入門的人，因此在這邊請讀者先接受**3SAT**是**NP-complete**的事實，

接下來我們就要運用這個事實來證明點團問題也是 **NP-complete** 的！

§ 點團問題是NP-complete的！

讓我們簡單回顧一下簡約方法的思考邏輯：現在我們想要把3SAT簡約到Clique，於是必須要做的就是建構一個轉換方法，將任何3SAT的問題實例（也就是一個3CNF ϕ ），轉換成Clique的問題實例（也就是一個圖 G 和參數 k ），使得 ϕ 可以被滿足若且唯若轉換後的圖 G 中存在一個大小為 k 的點團！

假設我們拿到的3SAT實例 ϕ 是

$$\phi(x_1, \dots, x_n) = \bigwedge_{1 \leq j \leq m} (l_{j,1} \vee l_{j,2} \vee l_{j,3})$$

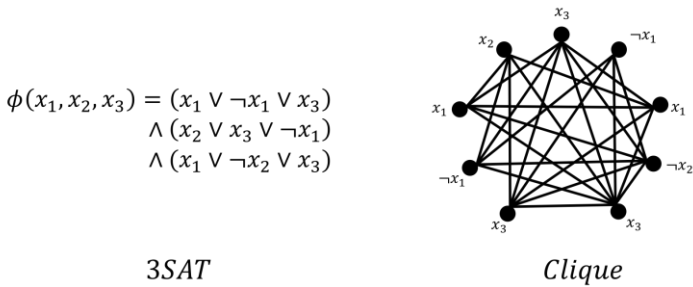
其中 m 是一個不會超過多項式 n 的整數。 $l_{j,i}$ 則是布林代數中所謂的文字(literal)，其實就是變數 x_1, \dots, x_n 之中的一個，或是加了一個反運算(negation)，變成 $\neg x_1, \dots, \neg x_n$ 其中一個。

現在我們已經了解3SAT的輸入是長得什麼樣子了，接下來就是要從 ϕ 來構造圖 G 了！我們的構造方式如下：

- 取 $V(G) = \bigcup_{1 \leq j \leq m} \{l_{j,1}, l_{j,2}, l_{j,3}\}$ 。

- 取 $E(G) = \{(l_{j_1, i_1}, l_{j_2, i_2}) | 1 \leq j_1, j_2 \leq m, 1 \leq i_1, i_2 \leq 3, l_{j_1, i_1} \neq \neg l_{j_2, i_2}\}$
- 令 $k = m$

下圖是一個實際的例子



圖表 3-16：從 3SAT 簡約到點團的實例。

那麼現在讓我們來看看為什麼這個簡約方法是正確的吧！要確認簡約方法的正確性，我們要檢查兩件事情：

- 當 $\phi \in 3SAT$ 時， $G \in Clique$ 。
- 當 $G \in Clique$ 時， $\phi \in 3SAT$ 。

以下就讓我們分別針對這兩個方向來作討論。

首先，當 ϕ 可以被滿足時，把每個 clause 中為真的文字中選一個到集合 S 裡面。因為在 G 中，兩個文字 $l_{j_1, i_1}, l_{j_2, i_2}$ 只會在

當他們互為反變數時才會不相連。而 S 裡面的所有文字都是來自一個合法的解，也就是說，變數 x 和 $\neg x$ 不可能同時在 S 裡面。於是，在 S 中的所有文字會在 G 中兩兩相連，也就是說 S 是一個點團！而根據 S 的取法，每一個clause剛好提供一個文字給 S ，於是我們更進一步知道 S 是一個大小為 k 的點團！

而當 G 中有一個大小為 k 的點團 S 時，首先我們觀察這些文字必定是來自不同的clause，因為在同一個clause的文字之間沒有邊，因此不可能同時存在一個點團內，於是 S 中剛好有來自每個clause各一個的文字。而現在我們把在 S 中的文字都設為true，也就是說如果 $x \in S$ ，那麼就把 x 設為true；如果 $\neg x \in S$ ，則設 x 為false。假如 x 和 $\neg x$ 都不在 S ，那麼就隨便令 x 為true。如此一來，根據 G 中邊的取法，我們可以保證這樣的輸入是合法的，也就是不會存在 x 和 $\neg x$ 同時被設成true的情況。更進一步，由於相對應的文字在每個clause中都被設為true了，這樣的輸入滿足了 ϕ ！於是我們知道 $\phi \in 3SAT$ 。

§ 結語

對我來說，簡約方法是計算理論中最有趣的一個想法之一，我很喜歡透過建立轉換的方式，使用其中一個問題的演算法，來幫助另外一個問題。如此一來，既可以是演算法的互相串連，同時也是建立困難度的重要工具。

在這篇補充的文章中，我們很迅速地看了第一個**NP-complete**問題是如何被找到的，並且實際運用簡約方法，證明了Clique問題是NP-complete的。在未來許多的下界和困難度證明中，類似的概念將會一再的重複，期待大家都可以享受這樣簡約的過程！

參考資料

- [1] Cook, Stephen A. "The complexity of theorem-proving procedures." Proceedings of the third annual ACM symposium on Theory of computing. ACM, 1971.
- [2] Levin, Leonid A. "Universal sequential search problems." Problemy Peredachi Informatsii 9.3 (1973): 115-116.
- [3] Karp, Richard M. Reducibility among combinatorial problems. springer US, 1972.

主動學習者計畫

《探索人類與自學的極限》

每週心得回顧

第一週回顧

這禮拜是NTU Active Learner計畫的第一個禮拜，整體看來都在進度上，同時也在禮拜一的時候架了這一個部落格，接下來一些學習的心得，還有計劃遇到的困難等等都會記錄在這邊。

先從有趣的事情開始好了，我在禮拜一讀完Pseudorandomness第二章的前兩節後，馬上忍不住試用新的部落格，打了一篇名為：「Polynomial Identity Testing」的文章，紀錄一些看完的想法。後來我又陸續放上了幾篇文章，一切看起來風平浪靜，和預期的差不多。

時間到了禮拜三，我的gmail信箱出現了一封wordpress（這個部落格使用的軟體）寄來的奇怪信件，信中提到有位網友在我的第一篇文章留言了！當下我真的是又喜又驚，開心的是竟然有人會回覆我的文章，而且內容蠻正面的，把他的留言翻譯成中文意思就是他很被我的文章感動。同時更是感到驚訝我寫得有點專業性的內容，原本預期沒什麼人會有興趣看，結果在第三天就遇到了知音。於是我很興奮地快速回覆了他的留言，還很自豪地跟室友炫耀這件事情。

隔天一醒來打開電子信箱，哇塞又有三個人在我的文章下面留言了，而且看起來他們都非常喜歡，還有人說他把我的部落格加入他的**bookmark**了！這時候我開始覺得怪怪的了，透過他們留下來的聯絡資料，沿路找到了一些沒什麼在管理的部落格，於是我開始在懷疑會不會這只些無聊的人或程式呢？結果當天晚上再度查看時，留言數目已經暴增到了**50**，而且開始出現一些文不對題的東西，像是「你看起來不像是本地人，不過英文用的還不錯，韓國人應該會很喜歡你的部落格」、「看起來你很需要一台濾水機」、「看完你的文章後我對新陳代謝更加瞭解了！」。我看了實在是欲哭無淚，理性上覺得這些一定都是來自什麼奇怪學校資工系資料探勘課程的無聊**project**，但是感性上又覺得說不定裡面有那麼一個是我的知音，千萬不可以就隨便刪除了，因此我決定再等待一下。於是當我禮拜五早上打開電腦時，我發現留言數已經逼近兩百了，照這樣下去，下個禮拜應該就可以突破千人大關了吧...於是我上網查了一下，發現看來不少人遇到類似的情況，而且**wordpress**也有專門的防垃圾訊息的外掛可以使用，於是我立刻裝上，從此以後一切又回覆到了平靜。

這次被垃圾訊息塞爆部落格的事件，讓我有兩個感受，在這邊簡單分享一下。首先是感覺到人真的很容易受到外在

的影響，在看到有人留言鼓勵我之後，我突然變得很有動力，連續打了兩三篇文章，想說不能辜負他們對我的讚許。現在想想，很好奇如果這一切都沒有發生，那我當時會這麼有拼勁嗎？哈哈這當然不會知道，但是不可否認的是受到了相當大正面的鼓勵。再來就是讓我蠻意外這些垃圾訊息竟然全部都是正面的文字，這也是讓我在感性層面很不能接受的部分，這些看起來這麼充滿能量的訊息，怎麼會是垃圾訊息呢！？而且這和平時我們看到的一些網路霸凌方向相反，說老實話，對於一個新手來說，這些是還蠻不錯的鼓勵（如果只有十個以內應該還不錯啦），因此我在最後也感到很矛盾，到底該怎麼處理呢？雖然最後的決定是趕盡殺絕，不過還是留下了最一開始的那一篇留言，就當作小小的紀念吧！

沒想到第一個禮拜的心得竟然大部份的內容是在講關於這個部落格遇到的故事，對於實際的學習過程沒有著墨太多，不過就當作一個輕鬆的開始吧，接下來就要開始認真接受知識的沐浴囉！

本週總結

§ 預期目標

- Computational complexity: Basic Circuit Complexity:

$P/poly$; Circuit Depth: NC, AC

- Pseudorandomness: 2.1-2.2

§ 執行成果

- Computational complexity: 100%
- Pseudorandomness: 100%

第二週回顧

這禮拜是NTU Active Learner計畫的第二週，果然和預期的一樣仍然充滿了幹勁，不知道一個月後是否還能維持呢？

本週的進度是隨機複雜度類別還有隨機性質 (randomness) 和其他計算模型性質的比較，例如隨機性質比非決定性 (nondeterministic) 還要強，但是比非均一性 (nonuniformity) 弱。而最令人感動的則莫屬機率方法 (probabilistic method) 了！每次看到使用機率方法的證明都令我心跳加快，忍不住叫好。

簡單報告一下學習的狀況，在這週開始的前兩天，就用零零碎碎的時間把pseudorandomness的2.3看完，然後在禮拜四晚上讀了complexity的進度，並且打了一篇心得文，主要是介紹隨機負責度類別背後的想法，以及和一般複雜度類別不太一樣的地方。

另外這週也開始進行作業了，由於之前曾經寫過一些，所以不算從零開始，但是剩下的也就是比較困難的。光是pseudorandomness第二單元四五題處理恆零多項式測驗 (polynomial identity testing) 的延伸，我就搞了半天仍沒有想

法。於是和鍾老師*約了在禮拜二早上討論，但是結束後聽得朦朦懂懂，又花了一段時間，終於在禮拜三晚上開竅了，其實重要的不是在於演算法問題的本身，更重要的是你對於當前的設定(是在有限體上還是整數群上?)，以及你想要的結果(再多項式時間內達到很小的誤差)有沒有很清楚的認知，找到是哪裡可能出問題，如此一來式子和目標一寫下，答案就呼之欲出了。

基本上這禮拜的學習蠻享受的，雖然在寫習題的過程中花了不少冤枉時間，但是我想這訓練到的是看題目還有面對未知事情的能力吧！

註解：*鍾楷閔老師是我在中研院資訊所實習的指導老師，目前仍有固定的研究合作。

本週總結

§ 預期目標

- Randomized Computation: BPP, RP, ZPP
- Pseudorandomness: 2.3

§ 執行成果

- Computational complexity: 100%
- Pseudorandomness: 100%

第三週回顧

這禮拜在不知不覺的匆匆忙忙中結束了，相對起來主動學習計畫的執行顯得比較慌亂一點，有超過一半的內容是邊吃飯邊看過的，習題大多也是在搭車的路上，或是騎腳踏車的時候想。該看的都還是有看完，不過就沒有時間好好整理一下，題目更是沒有完整的時間好好思考，這大概就是沒有排每週固定時間的下場吧！

不過這禮拜可是非常精彩，禮拜一的英文讀書會是第一次請外籍老師來幫我們上課，一群人在清晨六點半聚在BOT的交誼廳練習英文口說，不管在心理上或是身體上都很累，但是每次結束回到學校時，看到從總圖上方緩緩升起的太陽，就覺得新的一個美好禮拜要開始了。

禮拜二在台大電機系聽了偶像：**錯誤！超連結參照無效**。的演講。雖然遲到了一下，而且在中間突然轉到approximation的時候就完全跟不上，仍然覺得學到很多，從他鋪陳敘述研究的方式，可以感受到大師做研究的風範。禮拜三和中研院的學長們一起去交大參加Spectral Graph Theory的workshop，詳細內容可以參考這一篇。雖然現在網路日益發達，許多演講的影片、教學等等都可以直接有線上的版本可以遠距閱讀，但是在現場聽大師的演講，不同時做

其他事情，和身邊的人一起全神貫注時，獲得的不只是當下的內容而已，似乎更可以了解清楚整個的脈絡，所以我想這種實體的研討會、演講、工作坊應該還是不可能完全被網路取代吧！

由於Luca來台灣十天，然後其實只有給兩個talk，於是很厚臉皮的請王老師約了Luca來聊最近在做的研究，而他也很爽快的答應了，在禮拜五的下午有一個小時的時間和他聊聊最近在做的事情。不過我和王老師心理都知道我們是在這個領域的新手，很害怕問蠢問題，不過我想有問總比沒問好，而且機會難得，有很多高觀點的看法還有現在最前端研究的狀態和瓶頸可以藉這個機會好好搞清楚。於是就在禮拜四的時候花了不少的時間在準備和Luca聊天要問的問題。

當天和他先約在學校附近的咖啡店，Luca果然是道地的義大利人，點了一杯espresso，讓在一旁的我很讚嘆。後來回到了王老師的辦公室，開始了討論。從我們的題目開始說起，然後問了一些有關這個領域最近的發展，看起來已知的工具和我們認知中的差不多，唯一很強大的工具是一個叫做「平方和」(Sum of Square)的方法，我從十月初研究到最近，非常的有趣，但是也很深奧，在近年來理論CS這塊扮演蠻前衛的角色，很有希望破掉一個原本覺得很可能會成立的

Unique Game Conjecture，但是一切都還在進行當中。而我和王老師現在在做的研究是希望用它來建構某一類問題的時間複雜度lower bound，然而目前看到類似的研究都用了非常複雜的方法，於是我們很好奇有沒有其他容易的方式，但是從Luca的口氣聽來，這已經是目前最好的方法了，看來之後的研究還有很辛苦的一段路要走...

這一次和Luca的小小聚會，除了很榮幸能夠和大師交流之外，我覺得也從王老師身上學到了不少東西，從他和其他學者的互動中，學到了很多在比較偏學術的場合要注意的溝通表達技巧。而且也實在很感謝他幫我約到了Luca，這是我一個月之前怎麼想也想不到的事情吧！真是幸運能夠跟王老師一起做研究！

下禮拜生活又要回到了正軌，該打起精神繼續努力了！

本週總結

§ 預期目標

- Randomized Computation: Promise Problems; Randomized Reductions
- Pseudorandomness: 2.4

§ 執行成果

- Computational complexity: 100%
- Pseudorandomness: 100%

第四週回顧

這禮拜進入了自主學習者的第一個緩衝週，基本上因為進度都有跟上，所以這週大部份的時間都是花在做習題。然而事情沒有計畫的簡單，Pseudorandomness的習題大部份除了有很多小題之外，大多都需要一些全新的想法才有可能解得出來。像是第二章第九題的Spectral Graph Theory，那六個小題基本上就是那個領域最開始一些創新想法的簡單版，幾乎都需要一點點巧思才有可能解開。不過也因為如此，多少有訓練到自己看問題時的開放思維吧！

但是此時思考時間還有看參考資料之間的權衡就是一件很重要的事情了，網路上其實都可以多多少少查得到習題的解法，如果早一點放棄，趕快解出來，實在可以省非常多的時間，而這些時間就可以拿來讀更多東西或是寫其他的習題。但是用這樣子的方式學習，就有點失去了訓練思考的機會，誰知道哪天當自己做研究的時候，會不會就缺少了這樣子創新思考的能力？

我覺得，兩種的極端都不是一個好的學習方式，如果花太多時間自己想的确很容易陷入死胡同浪費寶貴的時間，直接看解答就和念書沒什麼兩樣，失去訓練研究能力的機會。

看來如何平衡兩者的時間和時機是個非常重要的學習課題呀！也許，這也是當初Salil出題時的想法呢！

本週總結

§ 預期目標

- Computational complexity: Review
- Pseudorandomness: Review

§ 執行成果

- Computational complexity: 100%
- Pseudorandomness: 100%

第五週回顧

隨著年底的到來，時間果然過得特別快，事情也令人多的感覺做不完。這禮拜身邊歡樂的氣氛讓自己稍微鬆懈了一下，同時週末要回家一趟，事情將要無法順利做完是意料中的事。看著行事曆一排排的死線，心中又很想要一些別的東西，做一些別的事情，真是左右為難啊！

還好當初在規劃自主學習計畫時有預料到這學期結尾的忙亂，於是刻意排一些之前已經看過的範圍，所以這週的進度很順利地利用兩個吃飯的時間還有半個晚上的圖書館解決了。而且看第二遍的好處除了幫這次省了一些時間之外，更讓我對於同樣的主題，有了比較深刻的見解和想法。例如這禮拜讀的counting complexity，是我之前最頭痛的地方，怎麼看都很沒有感覺，就是搞不懂為什麼要介紹這些定理，到底是重要在哪裡。這禮拜二重讀一遍時，突然恍然大悟，腦中頓時把counting complexity和其他不同的complexity定義連結在一起了！counting complexity的初衷就是要為另外一種類型的問題定義複雜度，而一些看似突兀的定理，想要做的事情其實很簡單，就是為了要把counting和其他的複雜度類別做連結。像是有名的Toda定理，就是將counting complexity和多項式階層(polynomial hierarchy)做了連結。

這禮拜雖然在忙亂中度過了，不過為了完成自主學習的計畫，還是想辦法硬著頭皮趕上進度。其實這樣還蠻有成就感的，一個禮拜這樣一點一點的累積，看著進度表上的完成率逐漸攀升，作業的答題率也越變越高，心中的成就感、學習的喜悅還有規劃好的進度壓力讓我更有動力和能量繼續完成學習，繼續加油！

本週總結

§ 預期目標

- Computational Complexity: Counting ;Approximate Counting, Uniform Sampling
- Pseudorandomness: 3.1-3.2

§ 執行成果

- Computational complexity: 100%
- Pseudorandomness: 100%

第六週回顧

2015 年的最後一個禮拜在放榜的緊張情緒，還有三連假的悠閒中度過了。公費留學考試在這幾年刻意將放榜的日子選在12/31號，就像大學指考的放榜選在父親節一樣，讓大家在特別的日子中面對人生中重要的結果出爐。

12/31禮拜四一早和老師meeting結束後，就很緊張地打開網路查詢放榜結果，但是網站上面什麼東西都沒有，像是根本沒有這回事一樣。在圖書館待了一個早上，大概每隔十多分鐘就會刷新一下所有可能的網頁，但是依舊是沒有任何動靜。緊張的情緒延續到下午，上高等統計推論的時候，實在很難專注在黑板上一行行的式子。然而，一直到吃完晚餐教育部的網站仍然遲遲沒有公佈放榜結果，只有一則關於新黨去抗議慰安婦事件的新聞稿。終於在我即將放棄等待時，終於在八點多的時候看到放榜的公告了，然後也順利地考上了這次的公費留學，放下心中的一顆大石頭。

不過開心的時間可沒辦法多久，還有很多東西等著學習呢！這禮拜Pseudorandomness進入了第三章，開始介紹一些去亂數(derandomization)的技巧，裡頭夾雜了許多有趣的數學和機率，非常好玩，細節將會在之後陸續整理放上來。而computational complexity輪到了平均複雜度的章節，是我之

前一直很沒有感覺的地方，對於定義的方式抓不太到想法。不過這禮拜再看的時候就有比較能夠理解每個定理的目的和核心概念，還有他們想要做到的事情是什麼，但是這實在是一個很大的主題，現在還都只是輕輕地品嚐一下而已，以後要再深入瞭解還有很長的一段路啊！

很快的再過兩個禮拜大四上學期就要結束了，這是第一個沒有打棒球隊的學期，也是幾乎都在讀書的學期，走到現在說實在的確有點開始疲乏了，東西常常讀不太進去。最近有開始試著做些放鬆的事情，像是打打撞球或是看個影片，調整讀書的心情和節奏。不過最重要的還是找到一個方向吧，是時候再檢討回想看看自己想要的是什麼，有時候一直忙碌地向前衝不見得就是比較好的。

本週總結

§ 預期目標

- Computational Complexity: Average-Case Complexity
- Pseudorandomness: 3.3-3.4

§ 執行成果

- Computational complexity: 100%
- Pseudorandomness: 100%

第七週回顧

期末考前的一個禮拜，是最輕鬆，也是最讓人無法專注好好把一件事情完成的一週。這次的進度來到了我之前就很有興趣的主題：互動式證明(Interactive Proofs)。在二十多年前，IP(Interactive Proofs的一個複雜度類別)被證明出和PSPACE(多項式空間複雜度)等價時，正式宣告了相對化技術(relativization technique)的死刑，並且開創了一個新的可能，緊接著Arora也發表了著名的PCP定理，利用互動式證明的概念，將NP定位在PCP(log n, 1)，讓人們一度以為有機會解決P vs. NP的問題，不過二十年過去了，P vs. NP仍然懸而未解，倒是互動式證明的概念帶起了許多近代密碼學重要的概念。

拉回主動者學習計畫，這禮拜實在是過得有點渾渾噩噩，於是在最後沒有完成應該唸完的進度。不過在作業方面倒是有不錯的進展，最近Pseudorandomness讀到成對式獨立(pairwise-independent)相關的作業寫得還算得心應手，尤其是經過連續幾個題組的洗禮，等於是走過了兩三篇論文的结果，讓我對於這個去隨機化(derandomization)的技術有了更深刻的了解。

隨著這一個section的結束，這一學期也進入尾聲，在被知識沐浴的同時，時間仍然不留情地快速流過。突然感慨不知道十年二十年後，現在花了不少時間學的一些東西，搞半天才弄懂的定理，會不會還依然清晰，又或是早就忘光了？人終究還是個很local的生物，還是別想太多，做現在自己認為是值得做的事情吧！

本週總結

§ 預期目標

- Computational Complexity: Interactive Proofs:
IP; Variants of Interactive Proofs; AM vs. IP, AM vs. PH
- Pseudorandomness: 3.5

§ 執行成果

- Computational complexity: 100%
- Pseudorandomness: 100%

第八週回顧

當初在規劃進度的時候，特地設想到期末考中可能會有許多無法預期的事情發生，很可能沒辦法專心的學習新的範圍，因此故意把複習週排在這個時間點。而的確這禮拜因為一些煩心的事情，還有考試及meeting，進行的不是很順利。前四天下來，除了想了一些題目之外，幾乎沒有把之前的落後的進度補起來。再加上禮拜五回家、禮拜六投票日、禮拜日出發去家庭旅遊，即使在沒有多餘進度安排的情況之下，這禮拜的複習還是失敗了。

事後檢討起來，我認為有幾個可以改進的地方如下：

除了安排進度之外，更要排出「固定」的學習時間。雖然這是自主學習，但是如果是很隨性的找閒暇時間來閱讀資料和想題目，一旦事情多了起來，或是出現一些臨時狀況，就容易不知不覺拖累進度。因此仿效學校課程有固定的上課時間，如果自主學習也有定下每週閱讀、討論的時程，那麼相信可以減少進度落後的可能性。

善加區分「專注時間」和「放鬆時間」。大三時我修了一門coursera上的課程：Learning How to Learn，在課程的最一開始，老師就向大家介紹這兩個不同的學習狀態。當我

們處於專注時間時，可以處理一些瑣碎細膩的問題，以數學為例，這時就很適合把證明的細節走過一遍，了解每一個參數的意義還有式子之間的關聯。然而在放鬆時間的時候，並不意味什麼事情都不能做，此時我們可以任由創意奔騰，想想之前卡住很久的問題，或是對自己挑戰之前看完的東西，說不定就可以發現一些有趣沒有想過的方向。在這兩個不同的時間，適合學習的方式不太一樣，如果弄錯了，時常會事倍功半。反之，能夠如魚得水的掌控自己學習的狀況，那麼就可以完全發揮學習的效果了！

雖然很遺憾的這禮拜沒有完成預期的目標，但是從失敗中學習，期待之後遇到類似情況的時候可以處理得更好！

本週總結

§ 預期目標

- Computational Complexity: Review
- Pseudorandomness: Review

§ 執行成果

- Computational complexity: 100%
- Pseudorandomness: 100%

第九週回顧

繼上週期末考造成進度拖累之後，這禮拜面對的是五天四夜的家庭旅遊，再加上禮拜五下午有一個期末口試和報告，自主學習計畫面臨嚴峻的考驗。

再仔細思考規劃過後，覺得不可能再出去玩的時候讀書，這樣不但效率會比較差，還很有可能影響心情，加上這次打算要每天打遊記，於是橫下心來，決定把所有東西留到週末一次讀完！

於是開開心心的和家人玩了五天四夜之後，順利考完模型理論的口試還有專題的期末報告，禮拜六一早我便來到了圖書館，開始奮戰。由於是新的一個階段的開始，這禮拜的 **Pseudorandomness** 部分進入了全新的章節：**Expander graph**。**Expander graph** 是應用數學中一個重要的工具，因為它具有高連結性，但是點與點之間卻是稀疏的，使得我們使用的時候付出的成本比較低，但是卻可以獲得很好的遍歷性質。我很興奮地迅速把基本的定義還有簡單的參數關係推導看完，並且做了幾題相關的題目，順利的將這禮拜的進度完成。

雖然剛好遇到了家庭旅遊，不過我在一開始便訂好計畫，並且在玩的時候好好的玩，在唸書的時候心無旁騖的享受，於是最終還是順利地達成目標。很高興這次的經驗告訴了我，讀書和生活都是很重要的，彼此之間是可以相互共存，只要能夠先計畫好，然後不要東想西想的，在當下享受正在做的一切，事情都可以順利做好的！

本週總結

§ 預期目標

- Computational Complexity: PCPs and Hardness of Approximation; NP in PCP(poly(n), O(1))
- Pseudorandomness: 4.1

§ 執行成果

- Computational complexity: 100%
- Pseudorandomness: 100%

第十週回顧

上禮拜結束所有的期末報告之後，正式來到了大學生涯最後一個寒假。雖然是個假期，但是對於準備要申請國外研究所的我來說，這是當兵前最後一個空擋好好為接下來的研究還有課業打穩腳步。除了持續進行原本的自主學習計畫之外，這段期間還多了一個重要的學習目標：代數。因為在幾個月前得知當我放棄雙主修數學系改成輔修之後，原本可以抵免系上的線性代數課程變成無法充兼數學系的輔系必選學分，也就是說我必修再多修一門代數相關的課程。為了要拿到數學系的輔系，我必須跳過代數導論上，直接挑戰下學期的代數導論下！之前聽幾個雙主修同學聊過代數導論這門課，大家紛紛認為是一門很硬不容易讀的科目，看來下學期除了自己的研究之外，在修課方面也會面臨不小的挑戰。

也因此我借了代數的課本，打算在寒假把上學期的課程內容努力唸完，換句話說，我要在短短四個禮拜的時間讀完一學期的教材並且熟絡到一定的程度，真是刺激呀！於是這禮拜就在代數還有計算理論以及研究之間過去了，算是順利的把代數的第一個大章節：群（Group）快速看完第一遍，接下來要重新仔細看過定理和證明的細節，然後希望可以把剩下兩個章節：環（Ring）和體（Field）掃過一遍。

至於自主學習計畫的部分，也規規矩矩地達成了目標，不過目前在習題的部分遇到了一些瓶頸，希望在之後幾個禮拜可以有所突破！

本週總結

§ 預期目標

- Computational Complexity: Linearity Testing, More Inapproximability; Algebraic Complexity: AlgP, AlgNP
- Pseudorandomness: 4.2

§ 執行成果

- Computational complexity: 100%
- Pseudorandomness: 100%

第十一週回顧

這禮拜計算理論和pseudorandomness的進度都分別來到最精彩的部份，前者進入「代數複雜度」(Algebraic complexity)的領域，目標是研究進行平時常見的代數運算所花費的計算複雜度，很神奇的是前人把這些看起來與電腦習慣使用的0,1位元運算做了很深刻的連結，提出了對應的複雜度類別，讓人們可以對最直接的數學運算做分析，不必再將問題轉換成電腦可以表達的方式。

而pseudorandomness的進度則來到了擴展圖(expander graph)的建構。在前兩個禮拜的閱讀中，看到了很多擴展圖實際的應用，像是如果我們在它上面進行隨機漫步(random walk)，並用經過的點產生偽亂數(pseudorandomness)，如此一來所需要花的真實亂數成本將會大幅減少！然而建構擴展圖並非一件簡單的事情，要做出一個具有好性質的擴展圖做簡單的方法是用亂數構成。然而從應用端出發，我們就是為了減少亂數的消費才使用擴展圖，現在如果連建構擴展圖都需要很多亂數了，那豈不是本末倒置！？於是這禮拜在研究的方向就是要如何在不依靠亂數的情況下建立一個好的擴展圖，簡單來說，透過一些適當的圖像操作，我們可以讓一個小的擴展圖越變越大，並且控制參數使得擴展的程度

(expansion)仍然維持到我們可以接受的範圍內。詳細的介紹可以參考整理出來的筆記：[Expander Graphs - Construction.](#)

本週總結

§ 預期目標

- Computational Complexity: Algebraic Complexity: Perm vs. Det; Algebraic Circuit Lower Bounds
- Pseudorandomness: 4.3

§ 執行成果

- Computational complexity: 100%
- Pseudorandomness: 100%

第十二週回顧

才覺得寒假沒開始多久，農曆新年就緊接著來了，雖然這幾年越來越沒有像小時候那樣過節的氣氛，但是每年到了這個時候仍然都是最放鬆的時刻。今年春節的作息比較一致，每天起來吃個早餐開始唸一些書，泡完咖啡後再唸一下書，中午和家人聚餐吃飯聊聊天，午覺後再繼續唸書，晚上可能會看個電影或是影集，睡前再看看小說。生活被書本、食物和電影填滿了，三個都是我最喜歡做的事情，這樣的日子過得還真是愜意呀！

這禮拜的自主學習計畫是第一階段的最後一個讀書週，computational complexity的教材已經進入了最後一個章節：量子計算(quantum computing)，而pseudorandomness的部分也將要完結有關擴展圖(expander graph)的討論。今天就來簡單談一談量子計算吧！

用最不技術性的講法來解釋量子計算，要從他和傳統計算(classical computing)的差別談起。現在所有的電腦都是根據圖靈機(Turing machine)的模型來運算，而一切運算的最基礎單位就是位元(bit)，也就是平時人們喜歡說電腦是由0-1構成的。而量子電腦的不同之處就在這邊，運用了量子疊加態的特性，電腦科學家憑空創造了所謂的量子位元(qbit)，使

得在我們「觀測」之前，一個量子位元到底是0還是1是無法被確切知道的！雖然現在量子電腦還沒有被量產，製造量子位元的技術也尚未成熟，相關的理論卻已經五花八門。電腦科學家和數學家們運用想像力和漂亮的數學模型成功建立了完整的量子計算理論，在某些應用上已經被證明比傳統電腦還要厲害。舉例來說，質因數分解(**factoring**)在傳統電腦尚未有多項式時間的演算法，然而在量子電腦上卻可以很快速地被解決。也因為許多問題可以在量子電腦上被有效率地解決，所以產生了一些相對應的新問題來思考還有什麼難題是連量子電腦都難以處理的。

於是，量子計算這個領域形成了一個蠻有趣的現象：數學理論遠遠超前於實務上的結果。看著書上一條條的定理和演算法，想像在未來三十年後這一切可能就成真了，真是令人五味雜陳。

本週總結

§ 預期目標

- Computational Complexity: Quantum Computation:
QBP; Quantum Computation: Fourier Transform and
Factoring
- Pseudorandomness: 4.4

§ 執行成果

- Computational complexity: 100%
- Pseudorandomness: 100%

第十三週回顧

經過兩個多月的努力，自主學習計畫的第一階段終於順利的如期完成了！看著一篇篇的筆記還有回顧心得，除了感嘆時間過得很快之外，也很替自己感到高興，能夠在期末考的壓力、全家出遊還有過年的外界誘惑下把預計的進度看完。這一個階段很多的教材是我在去年暑假就一直想要看，但是斷斷續續沒有完成的部分。透過這次自主學習計畫，一半靠自己的毅力還有一半的心理壓力，雖然中途有幾週延誤了進度，最後還是衝刺補了回來，算是達成了以前做不到的任務了吧！

不過當然還是有許多美中不足的地放，既然第一個階段告一個段落，就在這邊做一些檢討吧！

首先是時間的安排，因為兩個月的時間內包含了學期中、考試期間、寒假和過年，可以規律學習的時段非常不固定。在剛開始的前三四個禮拜，我都是利用禮拜一早上沒有課的兩三個小時把內容看一個段落，然後再利用零碎的晚上讀書時間跟週末把剩下的進度看完並且打筆記和回顧心得。然而到了考試和放假期間，在沒有事先規劃好的情況之下，就有點亂了手腳，導致連續三個禮拜進度落後。根據這個失敗的經驗，警惕我對於細節時間規劃的重要性，因此在新的

學期開始課表確定後，我必須明確的制定好自主學習的時間和科目，希望透過這樣的要求，可以讓學習的效果提升，減少延誤的可能性。

除此之外，作業習題也是一個比較嚴重的問題。因為少了一般學校課程強致死線的規定，所以我變成會將作業的優先順序放在比較後面。再加上少了分數和同儕競爭的壓力，在寫作業時也會比較悠閒，造成進度緩慢。雖然這樣的好處是我可以把每個問題想的比較透徹一些，但是在進度上就變得不甚理想。於是在接下來的階段，我打算也要制定作業死線，強迫自己在有壓力的情況之下學習。同時也需要再更積極的和實驗室厲害的學長請教，不要再閉門造車。

隨著新學期的開始，自主學習計畫也要正式進入下半階段，期待在這兩個多月中，可以扎實有效率的把學習目標達成，並且改進之前自學遇到的問題，找到適合自己的一套自主學習方法！

本週總結

§ 預期目標

- Computational Complexity: Review
- Pseudorandomness: Review

§ 執行成果

- Computational complexity: 100%
- Pseudorandomness: 100%

第十四週回顧

這禮拜是期待已久的開學週，在台大的最後一個學期，修課數量明顯的減少，從以往大約5-7門的主科到這次只有三門，不過這三門都是我很熱愛而且扎實的課(有兩門課每週都有作業!)。此外還很幸運地選上咖啡學和音樂欣賞，看來這學期將會有充實的知識饗宴，又有豐富的感官享受！

而本週的自主學習計畫也很順利的在預計的時間內完成，新的這一個階段開始閱讀MIT一門進階計算理論課的課程講義。這些講義是當初修課學生輪流根據老師上課的內容編輯而成的，MIT不愧是世界第一學府，雖然「只是」學生們做出來的講義，內容水準之高實在令人讚嘆，讀起來非常簡潔有力，重點和篇幅的分配很得當，讓我在學習專業內容之餘，也可以學習到那些厲害的學生是怎麼樣子把剛學會的知識編輯成容易懂的文章。

除此之外，禮拜五的晚上是自主學習計畫的第一次期中聚會。看到其他夥伴們這兩個月來學習和成長，很開心在自己打拼的過程中可以認識一群同樣對於學習有著熱情的朋友。大家也聊了許多經營專頁、部落格或網站的經驗，看來對於專業性還有大眾性之間的拿捏是每個人共同面對的重要問題之一。而我也在這次聚會後思考許多，決定要在接下來

的兩三個月著手撰寫中文版的計算理論專題系列文章。希望透過平易近人的筆觸，讓有興趣的人能夠品嚐計算理論的有趣。目前已經在規劃當中，預計在一到兩週後將會開始進行連載！

本週總結

§ 預期目標

- Computational Complexity: The polynomial hierarchy and time-space lower bound
- Pseudorandomness: 5-1

§ 執行成果

- Computational complexity: 100%
- Pseudorandomness: 100%

第十五週回顧

這禮拜開始著手規劃撰寫計算理論相關的中文科普文章，約到了三位背景不太相似的朋友要在禮拜日幫我審閱試寫的四篇文章。希望可以先將目標的對象還有寫作風格確定下來，瞭解一下適合的文章內容、長度等。此外，我也簡單列出了可能的寫作大綱，目前計畫分成兩大部分，第一部份是基礎背景的介紹，讓原本對於資訊理論不熟的人，可以認識基本的概念和定義，這樣在第二部分的專題介紹中會更能抓住核心的蓋念。

禮拜日的初稿見面會是在新生南路巷子內的小飯廳舉行，總共一個半小時，三位朋友將我的四篇文章看完後提出了很多寶貴的建議。首先是確認讀者的定位。原本我很貪心的想要讓很多來自不同背景的人可以看懂，所以會在一些簡單的地方著墨太多，然後又不小心在其他地方邏輯跳太快，造成讀者的混淆。最後我們認為可以將這個系列文定位成和科學人雜誌上的文章類似，一開始鎖定的客群就是對數理有興趣的人。也因此雖然會顧及大家程度上的不同，但是對於基礎的數學觀念會為必備知識。畢竟非誠勿擾，希望能夠將計算理論介紹給真正有心也有興趣要認識的人！

再來就是對我的寫作風格提出了不少的建議方向，像是在舉例的過程可以適當的和數學定義做的交叉對照，才不會讓讀者看的迷迷糊糊；另外更需要多加一些示意圖，幫助釐清觀念，也方便想像和記憶。

很開心能夠有三位這麼棒的朋友協助我做這一件以前從來沒有嘗試過的事情，在和其他人討論溝通的過程當中往往可以看到許多自己沒有注意到的盲點。下個禮拜我會把這次討論會得到的建議放進目前的文章中，然後再努力完成接下來幾篇的初稿。順利的話應該可以在月底前開始連載！

本週總結

§ 預期目標

- Computational Complexity: Relativization and its limits;
Baker-Gill-Solovay; Boolean circuits, simple circuit
lower bound, uniformity vs. non-uniformity (Karp-Lipton)
- Pseudorandomness: 5-2

§ 執行成果

- Computational complexity: 100%
- Pseudorandomness: 100%

第十六週回顧

隨著新學期的開始，除了原本的自主學習計畫之外，又多出了幾個新的主題想要涉獵，有些是之前有簡短看過一點的，有些則是完全陌生的領域。最後我決定除了原本進行的這兩個課程之外，再另外加上兩個自學的支線，一個是UC Berkeley的Luca Trevisan教授在這學期開的Graph Partitioning, Expanders and Spectral Methods，另一個則是UIUC的Yihong Wu教授開的Information-theoretic methods in high-dimensional statistics。

第一個主題是我之前就略有接觸，在上學期末有參加一個在交大舉行的Spectral Graph Theory Workshop聽過幾個相關的演講。後來也很榮幸跟Luca Trevisan教授討論過研究，當時聽他說這學期會開和Spectral Graph Theory相關的課程，會把詳細的課程講義都放在網路上，難得對於這塊研究領域有專門整合理論和應用的課，我當然絕對不能錯過，於是從一個多月前就開始閱讀她的課堂講義，目前把前八份都讀完了。但是畢竟只是讀過還是有時候會跳過細節，並不見得能夠深入真的了解關鍵的地方，於是參考這次自主學習計畫的經驗，打算也要開始試著記錄一些學習後的筆記，有空也會放到部落格上面。

第二個主題則是參加我的專題指導老師王奕翔教授招集的讀書會，我們要看的這門課是今年第一次開設，主題也是偏向整合最近消息理論和高維度統計這些領域混合在一起的研究，連開課老師都表示不知道最後的課程進度會走到哪裡，是一個非常前端的學習資源，期待可以好好把握，學到很多東西！

於是從下週開始我就會把這兩個新的學習支線偷偷加入自主學習計畫的進度中，雖然負擔會大一些，但是最近已經開始慢慢有感覺到自己時間安排方面的成長，如果可以繼續維持下去，一定沒有問題的！

本週總結

§ 預期目標

- Computational Complexity: ACO and switching lemma
- Pseudorandomness: 5-3

§ 執行成果

- Computational complexity: 100%
- Pseudorandomness: 100%

第十七週回顧

這禮拜是個忙亂的一週，兩個作業的死線把讀書的計畫稍微打亂了。再加上週末回台中看棒球校隊打大專盃複賽，更加壓縮了原本的自由時段，所以這禮拜的自主學習計畫就只有完成了一半。

學期之間真的是既可以學到很多東西，又會把時間擠得很滿，沒有休閒的日子。最近無論是在圖論、代數導論的課程中，都學了很多之前從來沒有聽過的東西。尤其是圖論介紹的Ramsey Theory最讓我驚艷。Ramsey Theory處理的是類似於證明：

- $1, 2, \dots, 2n$ 之中任取 $n+1$ 個數字，必定有兩個數字互質。
- 六個人之中必有三人互相認識或是三人互相都不認識。
- 平面上任意九個三點不共線的點，必存在一個凸五邊形。

隨著問題越來越複雜，我們根本無法找到確切的Ramsey數，只能有非常鬆的估計。在做習題還有看定理證明的過程當中，時常會被前人美麗的巧思給感動，但是看到結果還是距離真實非常遙遠，心裡就會非常惆悵，覺得人類怎麼這麼弱。從離散數學大師Erdős所舉的例子中，我們大約可以感受到人類能力的渺小：如果現在來了一群武力遠遠勝過我們的外星人，他叫我們算出 $R(5,5)$ （某個Ramsey數），如果把所

有人還有電腦叫來一起算，我們還有機會弄出來。但是如果他們是要我們算 $R(6,6)$ （下一個Ramsey數），那我們倒不如直接想該怎麼打敗外星人了！

下個禮拜也是個非常有挑戰性的一週，在禮拜三晚上要在讀書會報告輪到我負責的段落，禮拜四有個期中考，禮拜五又有個看起來很難的作業要交。希望能夠一件一件慢慢完成！

本週總結

§ 預期目標

- Computational Complexity: Razborov's monotone lower bound
- Pseudorandomness: 5-4

§ 執行成果

- Computational complexity: 100%
- Pseudorandomness: 100%

第十八週回顧

這是一個非常飽滿的禮拜，禮拜一大部分的時間都在處理研究還有跟老師討論，禮拜二則是花了一半的時間準備讀書會要報告的講義，禮拜三則是從早到晚滿滿的行程，禮拜四下午進行了這學期第一個期中考，一直到了禮拜五才終於有喘息的空間。

幸好自主學習計畫剛好在本週輪到了複習的緩衝時間，所以在週末可以讓我即時地把上禮拜落後的進度趕快補上。不過在未來的幾個月，相信許多事情只會變得更多更重，該如何在時間的夾縫中求生存變成非常關鍵的議題。我發現這學期我做事情的方式有點切成太大塊，會花一整個晚上在做講義、寫作業，或是花一個下午打文章、弄研究。因此沒辦法完成太多的事情，而且也時常會因為邊際效應遞減，造成效率沒有很好。這讓我想起之前在線上課程看到的「番茄工作法(Pomodoro Technique)」，大致上的概念就是利用外在的工具，例如計時器，來控制自己做事情的時間區塊大小。透過外界的幫忙，讓自己可以在固定的一段時間就切換工作的內容，當轉換一件事情的時候，可以讓效率提高到亢奮的高點。一旦疲累但是不自知的時候，外界的工具會直接干擾

提醒你檢視自己目前的工作效率，此時就可以問問看自己，是否該休息一下或是換個事情做了！

這學期是影響我未來幾年很關鍵的一段時間，要同時兼顧學習、研究、雜事還有健康等等，透過自主學習計畫每個禮拜的反思回顧，我可以即時的注意到自己時間規劃還有做事效率上面的問題，並且馬上調整改進。真沒想到原本純粹是學習導向的計畫，竟然也有意料之外的功效！

本週總結

§ 預期目標

- Computational Complexity: Review
- Pseudorandomness: Review

§ 執行成果

- Computational complexity: 100%
- Pseudorandomness: 100%

第十九週回顧

這禮拜開始嘗試用一個電腦的App（Pomodone）來監測我的時間分配，然後發現的確出現不少的問題。首先是時間分配非常不均勻，我花了超過三分之一的時間在圖論上面，因為這門課每個禮拜都有五到六題的作業，而且最近的題目困難到連我要把解答看懂都要花很久的時間（因為大部分的題目都是十幾年前人家的一篇論文...）。於是這樣弄下來，我平均一個禮拜不算上課大概都還花了十五個小時在上面。

而另外一個問題就是像上禮拜發現的一樣，我會很執著的把一件事情做完才去做別的事情。例如在寫這次計劃的期末呈現專題文章時，我有時候就會一連花個兩三個小時不間斷的寫作。但是通常這樣子寫作的方式，超過一個小時之後，效率就會稍微下降，特別是在撰寫比較專業的部分，有時候我自己也還要整理思考一下，那時效率變得特別的low。如果我要做的事情不多，這樣子分配時間也許還過得去。但是因為我很貪心，想要完成太多事情，一旦這樣子一次做完一件事情，會變得有點奢侈，浪費了時間的邊際效應。

透過Pomodone的幫助，讓我抓出了時間分配的癥結點，再來就是要嘗試解決問題了！下週開始，面對困難的圖論習題，要更果決的早點放棄自己想，趕快查別人是怎麼做

的。在做研究或是寫作的時候，也要透過Pomodone提醒我已經花太多時間，可以適時地做些調整和休息。既然想要做的事情很多，就必須做出更大的努力！

本週總結

§ 預期目標

- Computational Complexity: Natural Proofs
- Pseudorandomness: 6-1

§ 執行成果

- Computational complexity: 100%
- Pseudorandomness: 100%

第二十週回顧

雖然最近被研究還有寫文章佔據了大部分的時間，不過靠著番茄工作法(Pomodoro Technique)的幫助，讓我可以很大塊的忙碌時間中，抽取出小塊小塊的零碎時間，來完成其他的事情。這對於自主學習計畫的進度，或是修課的複習、功課等等，都有看起來雖小，但是重要的關鍵影響。

不過我發現個人的情緒還是一個很難掌握的東西，即使有了外在的工具協助，做事的興頭上的時候，還是會很不受控制，造成原本的計畫無法順利完成。雖然有時候這樣的爆發會帶來一些意想不到的收穫，例如研究上的一些小突破，但是這樣對於進度的規劃長久下來還是會造成不小的傷害，畢竟如果長期沒辦法把所有安排的事情做完，慢慢就會累積成不好的習慣，開始忽視原本預定要做好的事情。

於是我就在想是不是要在規劃進度的時候就不要太有野心，安排少一點的內容，讓即使發生了突發狀況，例如：作業不小心花太多時間，都還是可以有很大的機會完成。但是想想又覺得這樣往後退也不是辦法，畢竟想要做的事情還是在那裡，把進度捨棄，就等同於放棄完成那些目標，這真的是我想要的嗎？

時間分配管理真的是一生的課題，也是永遠沒有最佳解答的大哉問。有時候真的會很想要逃避，不敢面對自己的黑暗面，選擇忽視曾經累積下來的失敗和錯誤。但是到了一個程度後，總是必須正面接觸，想辦法改進解決。最近一個很大的問題就是在讀書的時候會挑當下喜歡的事情做，沒辦法強迫自己把進度上一些比較繁冗的事情快速搞定，既然我已經找出問題了，要解決還是要靠自己，是展現意志力的時候了！

本週總結

§ 預期目標

- Computational Complexity: The frontiers of circuits lower bounds
- Pseudorandomness: 6-2

§ 執行成果

- Computational complexity: 100%
- Pseudorandomness: 100%

第二十一週回顧

最近除了正常的讀書做研究之外，大部分的時間都花在寫文章上面，而在數次的閱稿餐會中，讓來自不同背景的朋友看看我寫的文章，才發現原本我覺得很平易近人的內容，其實大家的感受都差很多。最明顯的例子就是介紹一個新觀念的方法，我在介紹一個新的東西時，會很習慣從比較高觀點，或是抽象的角度做宏觀的解釋。有時候自己掌握的不太好，就會變得有些虛無縹緲，讓人抓不到重點，在幾篇設定為大方向介紹的文章中就很明顯地出現這種情況。於是這時候幫我試讀的朋友就會很痛苦，紛紛向我抱怨沒辦法抓到我想要表達的重點，看過去覺得講的內容頭頭是道，但是看完後卻覺得沒有實際接受到什麼內容。

聽到這樣的回饋心中的感覺是既難過又開心，難過的是自己沒有如原本預期的把知識用親切的方式傳達出去，開心的是有人提早發現，讓我可以即時改進修正。雖然這是預期中會發生的事情，但是沒想到的是在我有刻意要避免讓人難懂，並且多舉例的情況之下，還是讓不少程度還不錯的朋友仍然吸收困難。讓我不禁很佩服寫教科書的那些學者，平時我們偶爾會抱怨教科書寫得讓人不太好理解，但是大多數的時候，最終都還是可以順利搞懂。這次換自己要把所學傳

遞給別人的時候，即使是些很基礎的東西，卻發覺要轉換成容易吸收的文字竟然比自己讀懂還困難。這樣到底是不是意味著其實我之前根本沒有搞得很懂過呢？

這大概也是自主學習計畫中意想不到的收穫吧！從自己嘗試寫文章的過程中，發現真正的學習是要能夠很精準的把所學轉換成讓人能夠接受的語言，而不是閉門造車，自己以為自己有懂就好。此外，走過了一次這樣的過程後，之後我在看書的時候，更會開始注意作者是怎麼樣子呈現一個概念。例如之前借了一本關於計算理論的英文科普書，翻了幾頁覺得很簡單沒有什麼特別，於是塵封了一陣子。最近再度翻起來看，突然覺得很有意思。看到一些我也曾經試著在文章中介紹的觀念出現在書本中，作者用不同的方法詮釋，把我寫得很死板僵硬的東西變得活靈活現。讓我體會到寫東西有時候不是量多量少的問題，而是在於一個溝通的過程，這個過程是否是讓人舒服的，是否有真正傳達到什麼東西，這才是最後會流傳很久的。

本週總結

§ 預期目標

- Computational Complexity: MAEXP lower bound; rigidity;
- Pseudorandomness: 6-3

§ 執行成果

- Computational complexity: 100%
- Pseudorandomness: 100%

第二十二週回顧

最近自主學習的進度來到了隨機抽出器(Randomness extractor)，實在非常的神秘，在這邊簡單記錄一下目前看完的心得。

隨機性在演算法的設計中扮演很重要的一個角色，許多問題因為使用了隨機性，而變成可以在非常快的時間內執行完，例如多項式相等測驗、質數測試等等。然而真正的隨機性並不見得是可以像在數學的世界中這樣予取予求，有時候我們只能接觸到一些接近隨機的來源，而非理想中完美的隨機性，那這時候該怎麼辦呢？

隨機抽取器這時候就派上用場了！面對一些比較不純淨的隨機來源，隨機抽取器可以幫我們把這些有點雜亂的輸入，做類似淨化的動作，把裡面完美的部分努力地抽取出來，提供演算法使用。而神奇的地方是，有時候隨機抽取器不見得可以表現得非常好，會損失不少原本帶有的隨機性。然而，此時一旦加上了一點點的完美隨機性後，透過這些完美的小傢伙，隨機抽取器竟然可以變得生龍活虎，把幾乎所有的隨機性都抽取出來了！

這樣的結果不但在理論上有非常好的分析，並且和許多其他偽隨機的構造有緊密的連結，同時也可以被方便的應用實際的演算法中，實在是個很厲害的研究領域。

本週總結

§ 預期目標

- Computational Complexity: Communication complexity
- Pseudorandomness: 6-3

§ 執行成果

- Computational complexity: 100%
- Pseudorandomness: 100%

