算法 Algorithms

第三章

+ Outline

- 算法 (Algorithm)
  1. 搜索問題
  2. 排序問題
  3. 優化問題
- 算法複雜度 (Algorithm Complexity)

算法 Algorithm (3.1)

- 算法(Algorithm): 接收有效輸入值(Input)並產生所需輸出值(Output)的包含<u>**指定步驟**</u>的一系列流程。

+ 範例

- 描述一個能在有限序列的整數中搜索最大值的算法:
  1. 把第一個數(1st #)視為暫時的最大值(Max)
  2. 把 Max 跟第二個數(2nd #)比較大小, 若 2nd # 比 Max 大, 則 Max := 2nd #
  3. 重覆直至比較所有值, max中的值為所求最大值。

- 偽代碼 (Pseudocode):

procedure $max(a_1, a_2, \ldots., a_n$: integers)

$max := a_1$

for $i := 2$ to n

  if $max < a_i$ **then** $max := a_i$

return max {max is the largest element}

- E.g. 11, 23, 4, 67, 9

偽代碼 (Pseudocode)

- 定義:
  Procedure algorithm name (list and properties of variables)
- 注釋: {comment}
- 賦值: variable **:=** expression
- 條件結構:
  - **if** condition **then** statement or block of statements
  - **if** condition **then** statement 1
    else statement 2
  - **if** condition 1 **then** statement 1
    else if condition 2 **then** statement 2
    else if condition 3 **then** statement 3 …

+ 偽代碼 (Pseudocode)

- 循環結構:
  - **for** variable := initial value **to** final value
    statement or block of statements
  - **for** elements with a certain property
    statement or block of statements
  - **while** condition
    statement or block of statements

- 返回語句: **return** output of algorithm

更多偽代碼請參考附錄**C (Appendix 3)**

利用算法解決問題的例子

- 比如以下三類:
  一. 搜索算法: 搜索特定元素在列表中的位置
  二. 排序算法: 把元素由小到大作排序
  三. 優化算法: 確定所有輸入的可能值中的最優值(最大值或最小值)。

一、搜索算法 Searching Algorithms

- 一般的搜索問題是在一含不同元素 $a_1,a_2,\ldots,a_n$ 的列表中定位元素 x，或者確認它不在列表中。
  - 如: 圖書館在允許某人借閱另一本書之前可能想要先檢查Ta是否在書籍過期的名單中。

一、線性搜索 Linear Search

- 搜索特定元素 x 在列表中的位置的算法:

如: $(x = 7)$　　4, 7, 3, 6, 1, 0, 9

1. 預設位置為1, 比較 x 和 a_1, 若不相等, 位置+1;
2. 繼續比較 x 和 a_2, 若不相等, 位置+1;
3. 重覆直至比較所有值直到找到相等的值, 若最後沒有相等的即傳回 -1。

procedure *linear search* (x:integer, $a_1, a_2, \ldots, a_n$: distinct integers)
$i := 1$
while ($i \leq n$ and $x \neq a_i$)
　　$i := i + 1$
if $i \leq n$ **then** *location* $:= i$ {location is the subscript i of the term a_i equal to x}
else *location* $:= -1$ {location is set to -1 if x is not found}
return *location*

一、二分搜索 Binary Search

- 在一個由小到大排列的列表中以比較中間位置數值來搜索 x 的算法:

如: (x = 7)　　0, 1, 3, 4, 6, 7, 9

1. 將要找的元素 x 與中間元素進行比較。如果中間元素較小，則搜索會跳到列表的上半部分繼續；否則搜索從列表的下半部分繼續(包括中間位置)。
2. 重複此過程，直到我們有一個大小為 1 的列表。如果我們要查找的元素與列表中的元素相等，則傳回對應位置。否則，傳回 -1 以表示未找到該元素。

procedure binary search(x: integer, $a_1, a_2, \ldots, a_n$: increasing integers)
$i := 1$ {i is the left endpoint of interval}
$j := n$ {j is right endpoint of interval}
while $i < j$
　　$m := \lfloor (i + j)/2 \rfloor$
　　if $x > a_m$ **then** $i := m + 1$
　　else $j := m$
if $x = a_i$ **then** *location* $:= i$ {location is the subscript i of the term a_i equal to x}
else *location* := -1 {location is set to -1 if x is not found}
return *location*

+ Linear Search vs. Binary Search

Example: 4, 7, 3, 6, 1, 0, 9

Python Outputs:

In [41]:
```
#Perform Linear Search
x = 7
array1 = [4,7,3,6,1,0,9]
linear_search(x, array1)
```
Out[41]: 1

In [42]:
```
#Perform Binary Search
x = 7
array2 = [0,1,3,4,6,7,9]
binary_search(x, array2)
```
Out[42]: 5

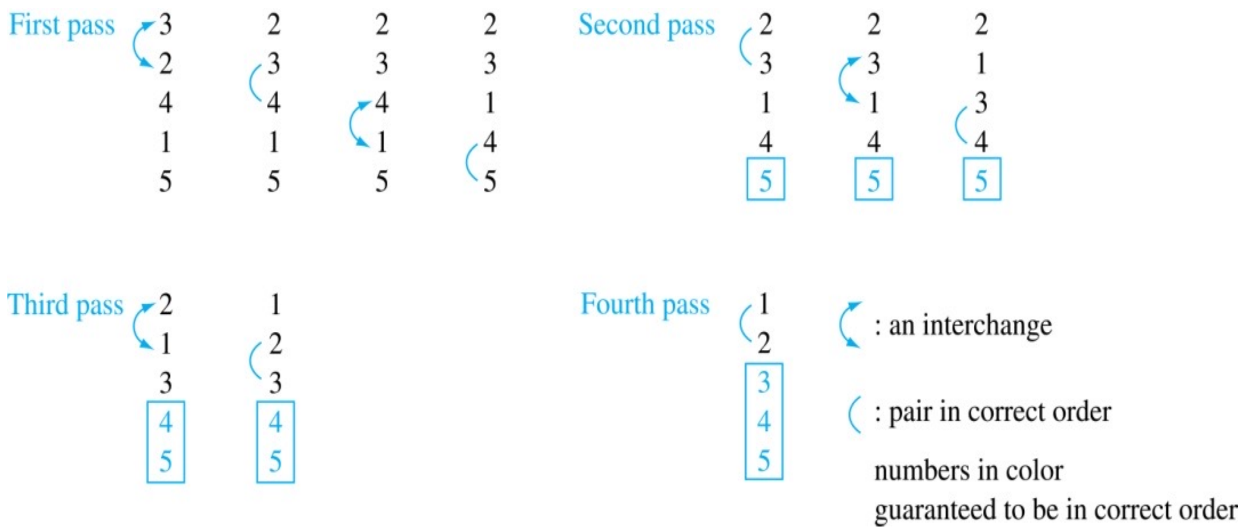
二、排序算法 Sorting Algorithms

- 用以對一列表進行排序
  - 如: 電話、價格、用戶名等排序

二、冒泡排序 Bubble Sort

- 通過比較相鄰元素並交換順序不對的元素作排序。

procedure *bubblesort*($a_1,\ldots,a_n$: real numbers with $n \geq 2$)
  for $i := 1$ to $n-1$
    for $j := 1$ to $n-i$
      if $a_j > a_{j+1}$ **then** interchange a_j and a_{j+1} $\{a_1,\ldots, a_n$ is now in increasing order$\}$



二、插入排序 Insertion Sort

- 通過不斷比較前方的元素大小並插入至正確順序位置排序。

procedure *insertion sort* ($a_1,\ldots,a_n$: real numbers with $n \geq 2$)
  for $j := 2$ to n
    $i := 1$
    while $a_j > a_i$
      $i := i + 1$
    $m := a_j$ {Hold a_j temporarily}
    for $k := 0$ to $j - i - 1$
      $a_{j-k} := a_{j-k-1}$ {move location one by one}
    $a_i := m$
{Now $a_1,\ldots,a_n$ is in increasing order}

+ Bubble Sort vs. Insertion Sort

Example: 42, 19, 32, 11, 8

Python Outputs:

```
In [30]: #Perform Bubble Sort
         array3 = [42,19,32,11,8]
         bubblesort(array3)

         Sorted Array: [8, 11, 19, 32, 42]
```

```
In [2]: #Perform Insertion Sort
        array4 = [42, 19, 32, 11, 8]
        insertionsort(array4)

        Sorted Array: [8, 11, 19, 32, 42]
```

三、貪婪算法 Greedy Algorithms

- 根據定義何謂“最優”，算法會在每一步都選擇“最優”方案。
  - 如: 在一段路線上選擇最短路程。

三、找零錢的貪婪算法 Greedy Change-Making Algorithm

- 利用最少硬幣數找零:

procedure *change*(c_1, c_2, …, c_r: values of coins, where $c_1 > c_2 > \ldots > c_r$; n: a positive integer)

for $i := 1$ to r

$d_i := 0$ [d_i counts the coins of denomination c_i]

while $n \geq c_i$

$d_i := d_i + 1$ [add a coin of denomination c_i]

$n = n - c_i$

[d_i counts the coins c_i]

+ Greedy Algorithms

- 以美元的硬幣為例: $c_1 = 25$, $c_2 = 10$, $c_3 = 5$, and $c_4 = 1$
- $n = 97$

```
In [27]: #Perform Change-Making Algorithm
         coin_values = [25,10,5,1]
         n=97
         change_making(coin_values,n)

Out[27]: [3, 2, 0, 2]
```

函數的增長 Growth of Functions (3.2)

- 在計算機科學和數學領域，很多時候我們會在意函數的增長速度。
- 計算機科學中，我們想知道當輸入的**Input**增加，一個算法運算速度的變化，以此來：
  - 比較兩個解決同樣問題的算法的效率
  - 確定一個算法在輸入值變多時是否實際上可用

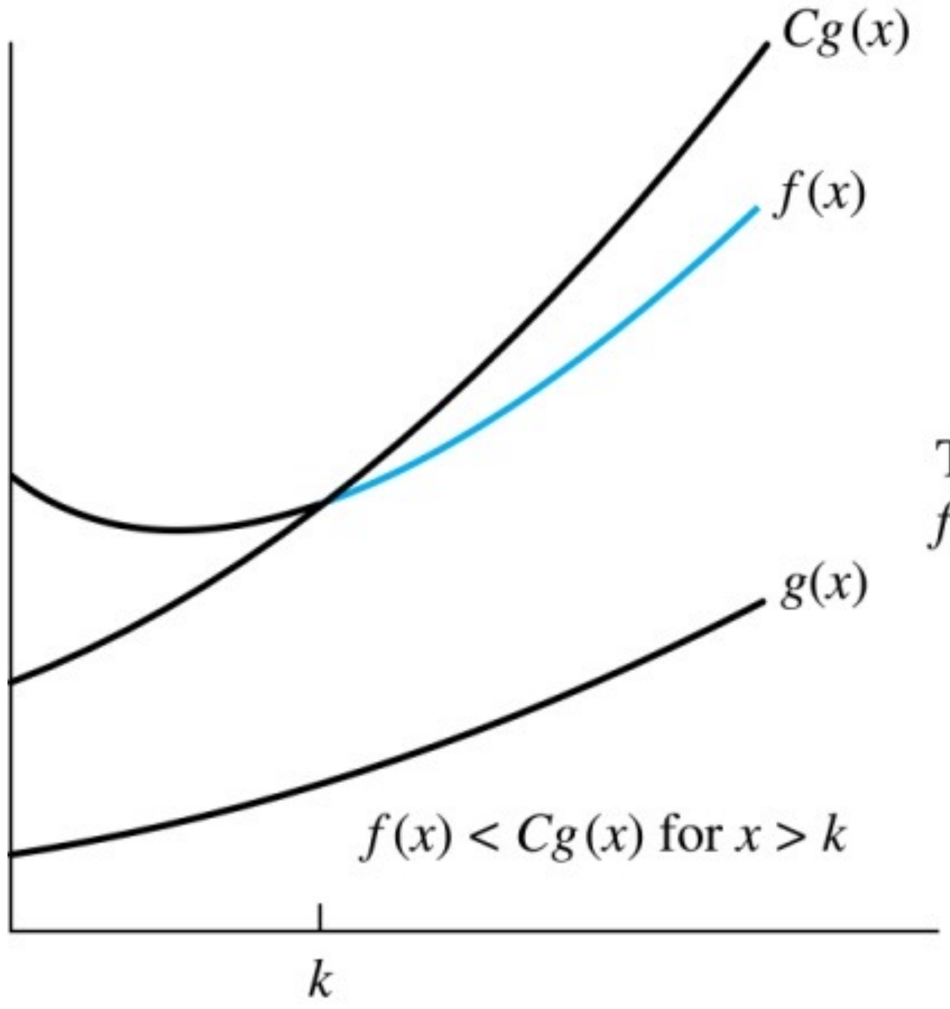
大O記號 Big-O Notation

- 稱 $f(x)$ 是 $O(g(x))$ 若存在常數 C 和 k 令每當 $x > k$ 都有

$$|f(x)| \leq |C(g(x))|$$

  - 讀作 "$f(x)$ is big-O of $g(x)$"

大O記號 Big-*O* Notation



The part of the graph of $f(x)$ that satisfies $f(x) < Cg(x)$ is shown in color.

例1

A. 證明函數$f(x) = 4x^2 - 8x + 1$是 $O(x^2)$ 的；

B. 判斷函數 $\log(n+1)$ 和 $\log(n^2+1)$ 是否是 $O(\log n)$ 的。

step 1 To prove that $f(x) = 4x^2 - 8x + 1$ is $O(x^2)$, we need to show that there exists a constant C such that $|f(x)| \leq Cx^2$ for all x sufficiently large

step 2 Simplify $f(x)$ to get $f(x) = 4x^2 - 8x + 1$

step 3 Notice that $-8x$ and $+1$ are of lower order than x^2, so as x grows, $4x^2$ will dominate the expression

step 4 Choose $C = 5$ (for example), then for all $x > 1$, we have $|4x^2 - 8x + 1| \leq 4x^2 + 8x + 1 \leq 4x^2 + 8x^2 + x^2 = 13x^2 \leq 5x^2$

step 5 Therefore, $f(x)$ is $O(x^2)$

step 1 To determine if $\log(n+1)$ is $O(\log n)$, we need to show that there exists a constant C such that $\log(n+1) \leq C \log n$ for all n sufficiently large

step 2 Using properties of logarithms, we can write $\log(n+1) = \log n + \log(1 + 1/n)$

step 3 Since $\log(1 + 1/n)$ approaches 0 as n approaches infinity, we can find a constant C such that $\log(n+1) \leq \log n + C$ for all n sufficiently large

step 4 Therefore, $\log(n+1)$ is $O(\log n)$

step 1 To determine if $\log(n^2+1)$ is $O(\log n)$, we need to show that there exists a constant C such that $\log(n^2+1) \leq C \log n$ for all n sufficiently large

step 2 Using properties of logarithms, we can write $\log(n^2+1) = \log n^2 + \log(1 + 1/n^2)$

step 3 Since $\log n^2 = 2 \log n$ and $\log(1 + 1/n^2)$ approaches 0 as n approaches infinity, we can find a constant C such that $\log(n^2+1) \leq 2 \log n + C$ for all n sufficiently large

step 4 Therefore, $\log(n^2+1)$ is $O(\log n)$

大O記號 Big-O Notation

我們想要找出函數的**最佳**大O估算，即其可取的階最小的簡單函數。

- 令$A(f,g,n)$: "$f(n)$ *is* $O\big(g(n)\big)$." 則 $\forall n \in N$：
  - $A(f_1,g_1,n) \wedge A(f_2,g_2,n) \to A(f_1f_2,g_1g_2,n)$
  - $A(f_1,g_1,n) \wedge A(f_2,g_2,n) \to A(f_1+f_2,\max(g_1,g_2),n)$

E.g. $f_1(n) = 3n^2 + 2, f_2(n) = 99n^3 + 4n + 1$

$\therefore (f_1f_2)(n)$ is $O(n^5)$;

$(f_1+f_2)(n)$ is $O(n^3)$.

例2

- 求下列函數的最佳大O估算: (請用階最小的簡單函數)
  - A. $(n^2+8)(n+1)$
  - B. $(n\log n+n^2)(n^3+2)$
  - C. $(n!+2^n)(n^3+\log(n^2+1))$

step 1 To find the Big O estimate for $A=\left(n^2+8\right)(n+1)$, we look for the highest degree term after expanding the product

step 2 Expanding A, we get $A=n^3+n^2+8n+8$. The term with the highest degree is n^3

step 3 Therefore, the Big O estimate for A is $O(n^3)$

step 1 To find the Big O estimate for $B=\left(n\log n+n^2\right)\left(n^3+2\right)$, we multiply the highest degree terms from each factor

step 2 The highest degree term in the first factor is n^2 and in the second factor is n^3

step 3 Multiplying these gives $n^2\cdot n^3=n^5$

step 4 Therefore, the Big O estimate for B is $O(n^5)$

step 1 To find the Big O estimate for $C=\left(n!+2^n\right)\left(n^3+\log\left(n^2+1\right)\right)$, we identify the term with the highest growth rate

step 2 The term $n!$ grows faster than 2^n, and n^3 grows faster than $\log\left(n^2+1\right)$

step 3 Therefore, the product of the highest growth rate terms is $n!\cdot n^3$

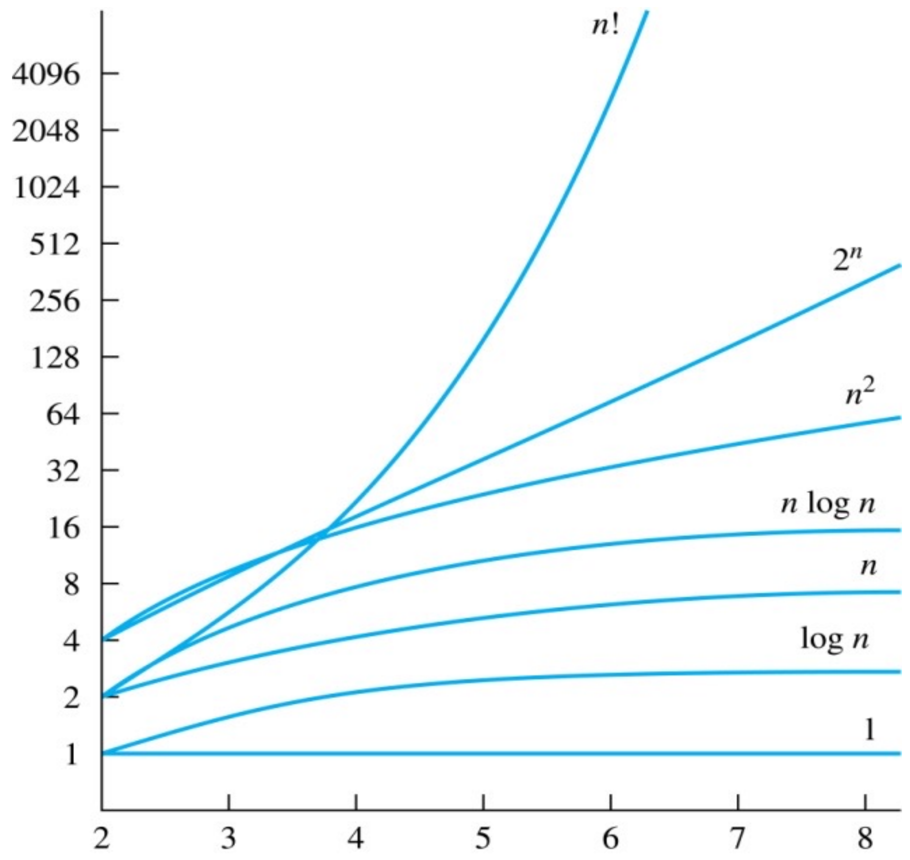
step 4 Since $n!$ grows faster than any polynomial, the Big O estimate for C is dominated by $n!$

step 5 Therefore, the Big O estimate for C is $O(n!)$

常見的函數大O估算及其排序

$$1 < \log n < n < n \log n < n^2 < 2^n < n!$$

- 當 n 越大, 各類函數的增長變化:



+ 算法的複雜度 Complexity of Algorithm (3.3)

- 運行一個算法需時多久?
  - 通過計算運算步驟的量

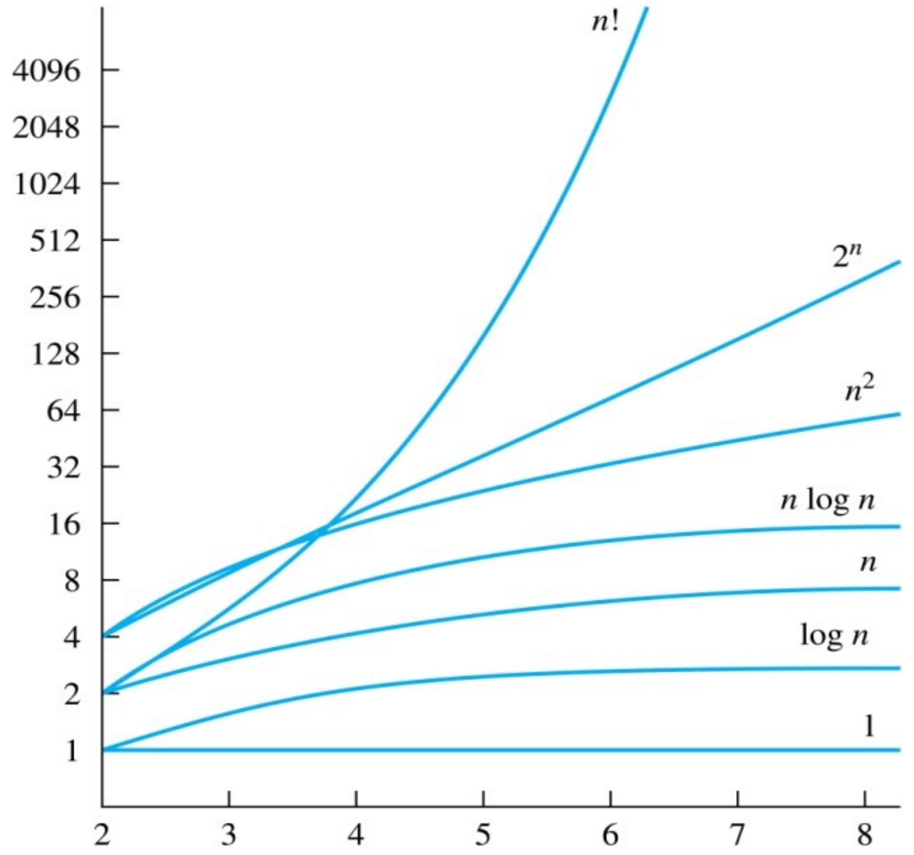
- 算法分析步驟:

1. 準確描述算法步驟
2. 定義大小為 n 的一次運算
3. 計算以 n 作大小的算法之運算量 $f(n)$

時間複雜度 Time Complexity

3.3 表1:

複雜度	術語
$O(1)$	Constant complexity 常量複雜度
$O(\log n)$	Logarithmic complexity 對數複雜度
$O(n)$	Linear complexity 線性複雜度
$O(n \log n)$	Linearithmic complexity 線性對數複雜度
$O(n^b)$	Polynomial complexity 多項式複雜度
$O(b^n), b > 1$	Exponential complexity 指數複雜度
$O(n!)$	Factorial complexity 階乘複雜度



算法的複雜度 Complexity of Algorithm

- 假設每次運算需時10^{-11}秒, 運行一個算法需時多久?

TABLE 2 The Computer Time Used by Algorithms.

Problem Size	*Bit Operations Used*					
n	$\log n$	n	$n \log n$	n^2	2^n	$n!$
10	3×10^{-11} s	10^{-10} s	3×10^{-10} s	10^{-9} s	10^{-8} s	3×10^{-7} s
10^2	7×10^{-11} s	10^{-9} s	7×10^{-9} s	10^{-7} s	4×10^{11} yr	*
10^3	1.0×10^{-10} s	10^{-8} s	1×10^{-7} s	10^{-5} s	*	*
10^4	1.3×10^{-10} s	10^{-7} s	1×10^{-6} s	10^{-3} s	*	*
10^5	1.7×10^{-10} s	10^{-6} s	2×10^{-5} s	0.1 s	*	*
10^6	2×10^{-10} s	10^{-5} s	2×10^{-4} s	0.17 min	*	*

* 代表需時多於 10^{100} 年

例3

- 求以下算法的複雜度(Complexity):

procedure *bubblesort*($a_1,\ldots,a_n$: real numbers with $n \geq 2$)
  for $i := 1$ to $n - 1$
    for $j := 1$ to $n - i$
      if $a_j > a_{j+1}$ **then** interchange a_j and a_{j+1} $\{a_1,\ldots, a_n$ is now in increasing order$\}$

+ 教材對應閱讀章節及練習

- 3.1, 3.2(只看 Big-O), 3.3
- 對應習題: (可視個人情況定量)
  - 3.1: All (試試看能否寫出大致流程, 並根據答案了解步驟)
  - 3.2: 1-8, 19, 21-22, 26-27.
  - 3.3: 1-14, 20-21