Lecture tutorial

本份講義主要著重在演算法的應用。以下會分成不同主題解說。

Time complexity/ Space Complexity

在學習算法時，Time complexity 與 Space complexity 是兩個非常重要的概念，它們分別用來衡量一個演算法在輸入規模變大時，所需的執行時間與記憶體空間的成長幅度。

Time Complexity

以 time complexity 為例，一個 time complexity 為$O(n)$ 的算法，當輸入長度從k增加到2k時，執行時間大約也會增加為原來的兩倍。，相對地，如果一個算法 time complexity 為$O(n^2)$，當輸入長度從k增加到2k時，執行時間大約也會增加為原來的四倍。。因此縱使兩個同樣 time complexity 為 $O(n)$ 的算法，它們實際執行速度仍可能有所差異，只是他們成長幅度相同，因為time complexity只描述成長趨勢，並未考慮一些實作細節的常數因子。
下面舉幾個例子

```
for(int i = 0 ; i < n ; i++) {
    sum += i;
}
```

```
for(int i = 0 ; i < n ; i++) {
    sum += i;
}
for(int i = 0 ; i < n ; i++) {
    sum += i;
}
for(int i = 0 ; i < n ; i++) {
    sum += i;
}
```

上面兩隻程式很明顯的右邊計算量為左邊三倍，因此在n足夠大時，時間上也約為左邊的三倍，但他們的time complexity 皆為O(n)

```
for(int i = 0 ; i < n ; i++) {
    for(int j = 0 ; j < n ; j++) {
        sum += i+j;
    }
}
```

而上方則為一個time complexity則為 O(n^2)的範例程式，可以發現當輸入 n 加倍時，程式所需執行時間會成長為原來的4倍

我們在學習演算法過程中，經常會遇到將資料/範圍分成兩塊或是更多塊處理，並持續做這個操作直到資料/範圍變成1為止，考慮一個簡單的簡單的情況為將資料分成兩塊，假設輸入長度為 n，每次操作後他長度都會剩 ½，問幾次操作後他長度會變成 1，也就是 n 要除幾次2 之後會變成 <=1 的數字，從下列式子中我們可以推導出這類演算法所需的操作次數 k:

$n/2^k = 1$
-> $n = 2^k$(等號兩邊乘以2^k)
-> $logn = k\ log\ 2$ (兩邊同時取log)

而 log2 是常數在計算複雜度的時候可以忽略掉，因此就可以得到，如果有演算法用到相同概念，他的複雜度就會是$O(logn)$。常見的例子包含我們後面會提到的 binary search。

而計算程式複雜度的方式主要就是要將程式拆解，去計算每一段的時間複雜度計算出來並將它加以組合

以上方例子來說

```
for(int i = 0 ; i < n ; i++) {
    for(int j = 0 ; j < n ; j++) {
        sum += i+j;
    }
}
```

`for(int j = 0 ; j < n ; j++)`，這段程式碼的時間複雜度為$O(n)$，因此可以將以上程式碼當成下面的程式碼

```
for(int i = 0 ; i < n ; i++) {
/* The time complexity of function(n) is O(n)*/
    function(n);
}
```

而`for(int i = 0 ; i < n ; i++)`，這段程式碼的時間複雜度也為$O(n)$，而他的每一步step都會執行一段時間複雜度為$O(n)$的程式碼，因此他的時間複雜度即為

$O(n * n) = O(n^2)$

但若將`for(int j = 0 ; j < n ; j++)`在最外面

```
for(int i = 0 ; i < n ; i++) {
    sum += i;
}
for(int j = 0 ; j < n ; j++) {
    sum += j;
}
```

則時間複雜度會變成$O(n + n) = O(n)$，因為常數部分不影響程式所需時間的成長幅度。

Space Complexity

Space Complexity的計算與Time Complexity類似，只是改為計算程式所需記憶體的成長幅度

space complexity主要分為兩部分

1. 輸入的 space complexity
2. 一撇除輸入外的 space complexity

譬如說題目為輸入一個 array 請你輸出他的和

1. 輸入的space complexity為O(n) (array長度為 n)
2. 而撇除輸入外的space complexity，只需要宣告一個變數累加總和，因此額外的space complexity就是O(1)

在提出解決方案中除了討論 time complexity，討論 space complexity 並清楚區分這兩部分也非常重要。

Recursion

recursion 最基礎的概念為我們寫了一個函數，且該函數內會呼叫自己。譬如說下列的程式就是一個計算 1~n 數字總和的 recursion：

```
void sum(int n){
   if (n == 1) {
       return 1;
   }
   return sum(n-1) + n;
}
```

在學習遞迴（recursion）時，主要需要考慮兩個重點：

1. 結束(邊界)條件:為了避免演算法無限執行，recursion必須在處理到資料量很小時停止，並直接進行處理。這通常是當資料只有一個元素時的情況。例如在排序的情境中，單一元素本身就已經是排序好的，因此如果我們在排序演算法中使用recursion，當資料只剩下一個元素時，就不需要再做recursion。
2. 回傳結果的處理：recursion 另一個重點是如何回傳正確的結果，當進入結束條件時，我們可以直接處理並回傳結果，如果還沒達到結束條件，則會在目前的函式中重複呼叫自己一次或多次。處理這部分的技巧在於「相信recursion會正確地回傳結果」。例如在計算 1 到 n 的總和這個例子中，我們要相信 sum(n-1) 回傳結果是對的，並由該回傳結果去算出sum(n)的結果。另一個例子是我們之後章節會題到的 merge sort，若我們用recursion來實作 sort，會先將輸入的array切成兩段，並對這兩個 array 用 recursion 的方式做排序，此時我們可以假設該recursion function會回傳一個 sort 好的 array，接下來我們要解決的問題就是，如何將這兩個 sorted 的 array 合併成一個新的 sorted array.

```
int[] sort(int[] array){
   array1 = first half of array
   array2 = last half of array
   array1 = sort(array1)
   array2 = sort(array2)
   // array1 and array2 is sorted
   // merge the array1 and array2 to a sorted array.
}
```

Binary Search

很多題目可能第一眼會覺得跟 binary search 無關，但只要將題目換另一種形式呈現，就可以用 binary search 解決。
簡單來說能夠使用 binary search 的題目基本上都能簡化（參考 Reduction 章節）成能夠以讓範圍內任一個數字x，配上一個條件f(x) `True/False` 來思考的題目，當f(a)為True時，任何一個i>=a，f(i)都會是True，而當f(a)為False時，任何一個i<=a，f(i)都會是false，而我們的目的就是要知道所有的f(i)，並以此推算出所需要的答案。而大約 80% 的題目都可用切成兩半和符合條件的最小(大)值來簡化。

切成兩半範例

用一個簡單的題目說明：
給定一個被 rotated 的 sorted array，請問這個 array 被 rotated 幾次？（rotate 一次代表最左邊的數字會跑到最右邊）

這個題目其實只要找到最小的數字（原本最左邊）跑到哪個位置就可以知道 array 被 rotated 幾次。也就是可以把題目改成“尋找 array 中的最小值” 或是 “尋找 array 中第一個比目前最左邊小的數字”

轉換題目後我們以目前最左邊的數字為基準，大於等於他的數字為 True，小於他的為 False。在搜尋的過程中如果搜到的數字是 True 那代表左邊的數字也都會是 True（比基準大），因此可以忽視左半段只搜右半段 array。相反的，如果搜尋到的數字是 False，那代表右邊的數字也都會是 False（比基準小），因此可以忽視右半段只搜左半段。

以下使用 test case 講解：

```
Input: [3,4,5,1,2]
Output: 2

以下 l 代表左邊界，r 代表右邊界，T/F 分別代表 True/False

l=0 r=5       l=2 r=5       l=2 r=3
3,4,5,1,2 -> 3,4,5,1,2 -> 3,4,5,1,2
T ? ? ? ?     T T T ? ?     T T T F F

填完所有後 F 第一個出現的地方就是最小值
```

符合條件的符合條件的最小(大)值範例

這種類型題目通常會給一些條件你要在符合條件的情況下讓最後的值最小或最大，但需注意該答案有單調性(monotonic)，舉最小值為例，假設有一條公路長度為 n 公里，我們想知道每小時至少要跑幾公里，才能在 t 小時內時間走完他，那我們可以知道需要 n/t 公里/小時的速度才能滿足我們的需求，而我們發現若速度 > n/t 公里/小時也可以滿足我們的需求，這就是所謂的單調性(monotonic)
下面給一道題目做為範例：

給定若干跟長度不同的竿子，你總共可以切 n 刀，要怎麼在切完之後讓最長的竿子的值最小？
通常看到這個題目會覺得這跟 binary search 完全沒關，畢竟直觀來看完全沒一串數字讓我們來"搜尋"。但其實以上題目可以轉換成這種形式：給你 n 刀的機會，有辦法讓最長的竿子的長度為 x 公分嗎？
假設竿子最長是 10 公分，那答案可能會是 1 到 10 公分。如果 $x = 5$ 可以做到，就代表比 x 大的數值都可以（也就是右半部都是 True），相反的如果 $x = 5$ 不能做到，就代表比 x 小的數值都不能（也就是左半部都是 False）

Pseudo Implementation

以下說明 *r,l* 兩個指標移動的位置，配合 True/False 去判斷就能夠知道自己的指標目前指到哪裡。（以下以左邊是 true 右邊是 false 舉例）

Scenario one

```
while((r-l)>1) {
    mid = (l+r) / 2
    if(match condition){
        l = mid //把 l 移到最後一個 true
    } else {
        r = mid //把 r 移到第一個 false
    }
}
```

Scenario two

```
while(l<=r) {
    mid = (l+r) / 2
    if(match condition){
        l = mid + 1 //把 l 移到最後一個 true 的下一格
    } else {
        r = mid - 1 //把 r 移到第一個 false 的前一格
    }
}
```

Binary Search implementation tips

在實作 Binary Search 時，許多人最大的困惑之一，是 迴圈的結束條件 以及 `left`、`right` 指標的定位。如果這部分想不清楚，就很容易陷入錯誤。以下為各個條件下的思考方式。

1. 把 Binary Search 想像成 True/False 陣列搜尋

假設我們的搜尋條件可以被抽象成一個True/False陣列，整個過程就像在一個形狀為

`TTTT???FFFF`

的序列中尋找所有`?` 的答案（`?` 的位置是未知的 True 或 False）。
Binary Search 的核心，就是不斷縮小範圍，直到知道所有位子的答案是T還是F。

在這個模型下，你可以明確定義：

- **`left`** 指標：代表第一個問號的位置，或是最後一個 True 的位置。
- **`right`** 指標：代表最後一個問號的位置，或是第一個 False 的位置。

2. 思考 `mid` 判斷後該怎麼移動

每次計算 `mid` 並檢查條件時：

- 如果 `mid` 的結果是 **True**，代表分界點在右側，因此你應該移動 `left`，並維持`left` 是最後一個 True或是第一個?的位置的性質。
- 如果 `mid` 的結果是 **False**，代表分界點在左側，因此你應該移動 `right`，並維持 `right` 是第一個 False或是最後一個?的位置的性質。

關鍵就是：無論怎麼更新，都要確保 **`left`** 與 **`right`** 的語意一致，不要中途改變定義。

下面舉幾個例子

若搜尋途中整個True/False 陣列的形狀為這個樣子，而紅色的字為當前 `left` 跟 `right`的位置

`TTTT???FFFF`

當我們搜尋 middle 為T時，整個陣列會變成

`TTTTTT?FFFF`

而 middle則指向藍色的位子，為了維持`left` 與 `right` 的語意一致，所以這時候我們就要將 `left = middle`

另一個例子，若搜尋途中整個True/False 陣列的形狀為這個樣子，而紅色的字為當前 `left` 跟 `right`的位置

`TTTT???FFFF`

當我們搜尋 middle 為T時，整個陣列會變成

`TTTTTT?FFFF`

而 middle則指向藍色的位子，為了維持 `left` 與 `right` 的語意一致，所以這時候我們就要將 `left = middle + 1`

3. 迴圈結束條件的判斷

當搜尋結束時，True/False 陣列會收斂成類似：

`TTTTFFFF`

這時可以去想 `left` 與 `right` 照我們剛剛的語意，他的位子會在哪裡，譬如說若 `left` 在最後一個T(紅色位子)而 `right` 在第一個F(藍色位子)，

`TTTTFFFF`
因此就可以知道結束條件為 `left` 和 `right` 相差為1，因此條件就會是 `while(right - left > 1)` 也就是當他們相差>1時應該繼續搜尋

而另一個例子

這時可以去想 `left` 與 `right` 照我們剛剛的語意，他的位子會在哪裡，譬如說若 `left` 在第一個問號，這時會發現第一個問號其實是最後一個T後面那個位子(紅色位子)，而 `right` 在最後一個問號，這時會發現最後一個問號其實是第一個F前面那個位子(藍色位子)。

`TTTTFFFF`

因此就可以知道結束條件為 `left` > `right` 時，因此條件就會是 `while(left <= right)`也就是當 `left` 還沒大於 `right` 時就要繼續搜尋。

Sliding window

Sliding window 靠兩個 pointers 去維護一段資訊，通常是靠移動 left pointer 跟 right pointer 去擴展或縮減去更新資訊，並找出所有符合條件的元素或元素集合，並藉由這些元素集合去計算出答案，這邊的元素集合大部分情況下都是指 subarray。判斷一個題目是否能使用 Sliding window 實作，首先要思考如果原本的 subarray 不符合條件（這邊條件指符合題目的

答案），那包含這個 subarray 的其他 subarray 也不會符合條件，反之如果原本的subarray符合條件，那所有被這個 subarray 包含的 subarray 也都會符合條件。

分辨是否使用

以下兩個題目：

1. 給定一串數字陣列，找出最長沒重複數字的子陣列有多長？
2. 給定一串由 01 組成的陣列，找出一段最長 0 和 1 數量相等的子陣列有多長？

第一題能夠用 sliding window 實作因為如果某段陣列 `1,3,6,2,5` 符合條件，那他的子陣列也都不會有重複數字（符合條件）。反之如果某段陣列 `1,3,3,5,2` 不符合條件，那包含他的其他陣列也都會有重複數字（不符合條件）。

第二題不能使用 sliding window 實作因為某段陣列 `0,1,0,1` 符合條件，但他的子陣列 `0,1,0` 的 0 跟 1 數量並不相等（不符合條件）。反之某段陣列 `0,1,0` 不符合條件，但包含他的陣列 `0,1,0,1` 的 0 跟 1 數量卻相等（符合條件）。因此並不符合使用 sliding window 的前提。

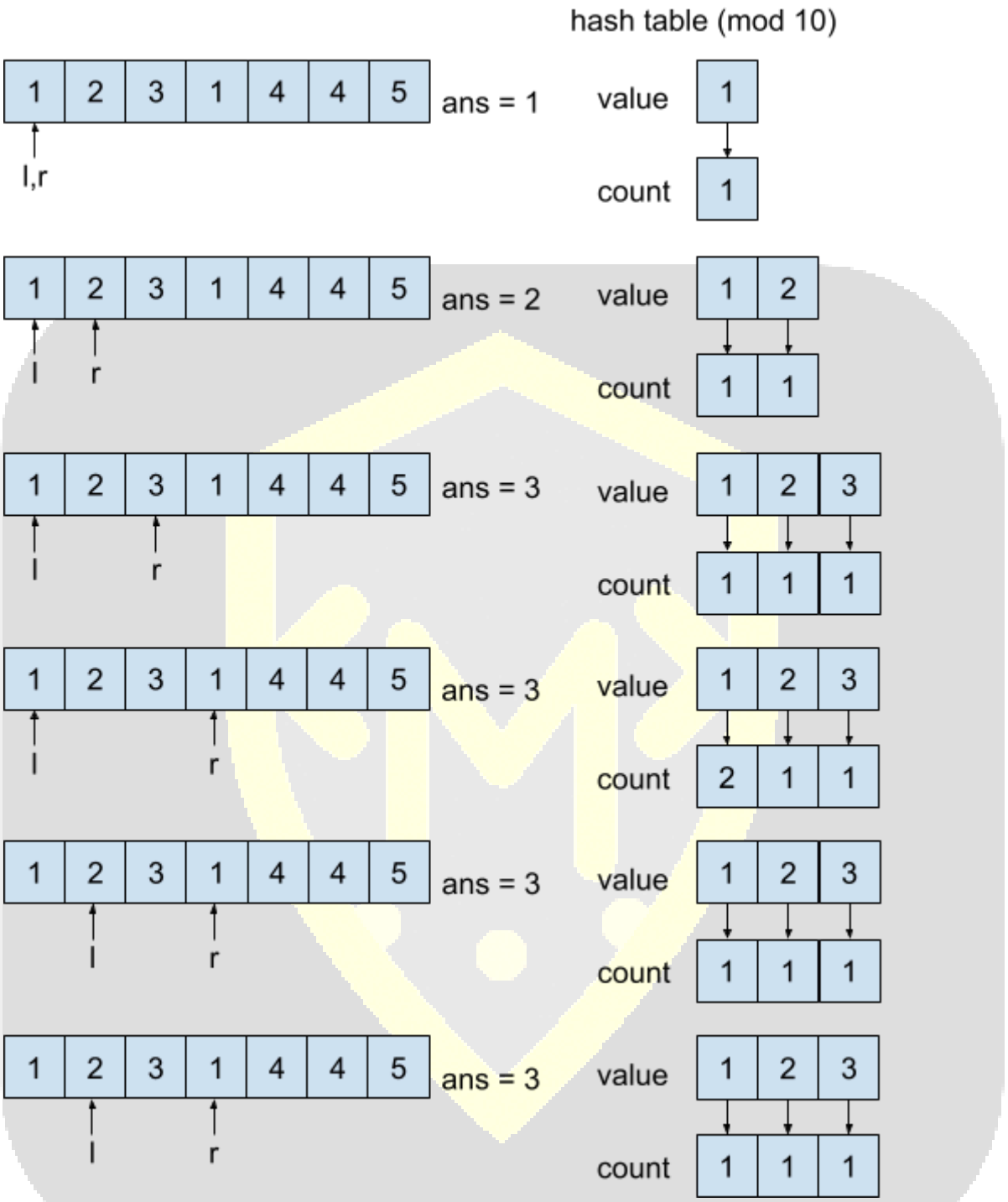
Sliding window 主要分成四部分

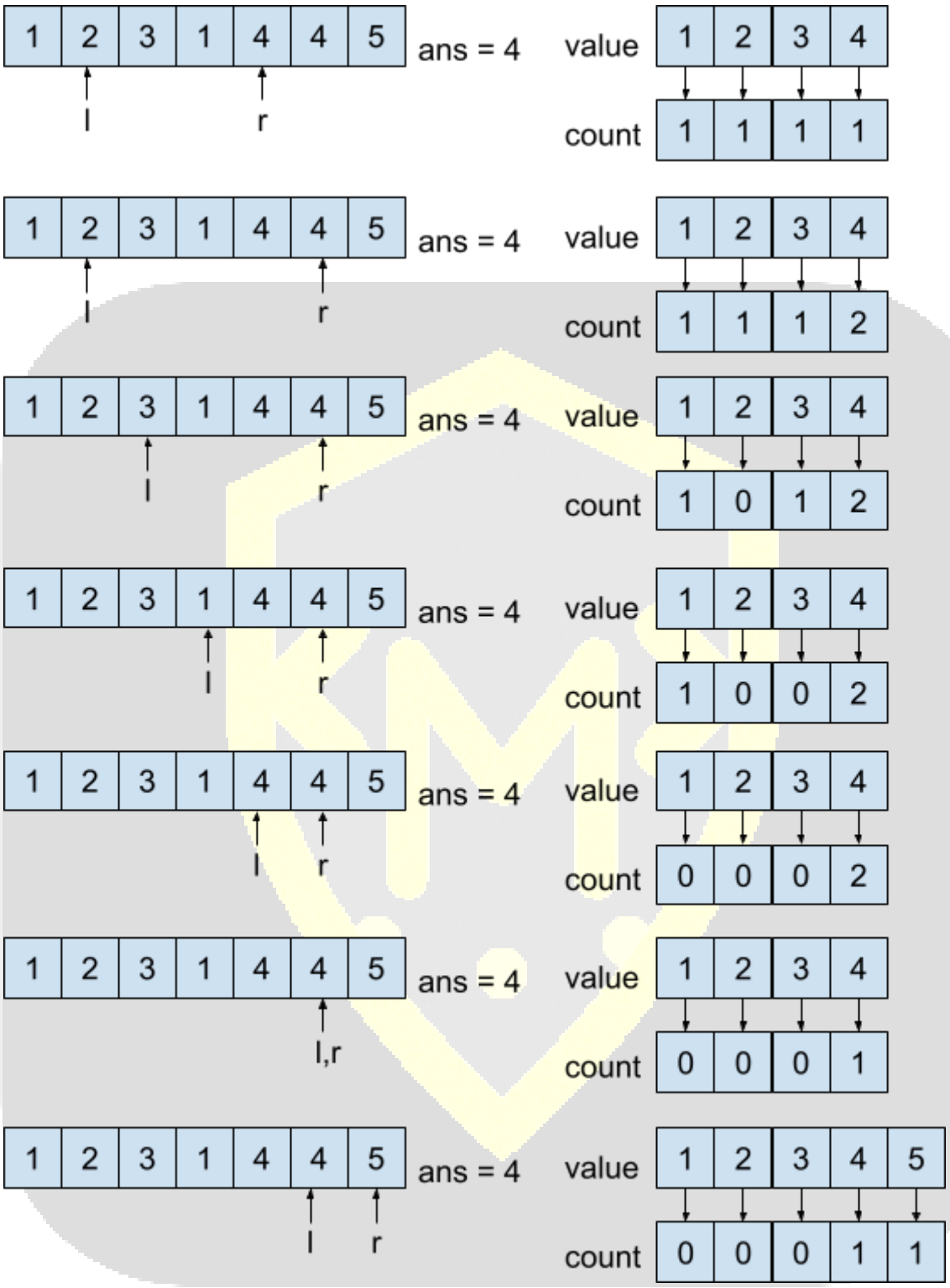
1. 決定要儲存的資訊，且該資訊可以幫助我們判斷當前元素集合是否符合條件
2. 當加入一個元素時如何更新資訊
3. 當刪除一個元素時如何更新資訊
4. 更新答案

範例

以下範例會使用上述的四個部分來執行。

通常在做 sliding window 的時候會使用 hash table 來儲存資訊（hash table 的用法請參考對應章節）。以下用『給定一串數字陣列，找出最長沒重複數字的子陣列有多長？』這個題目舉例。l 和 r 分別代表子陣列中最左邊的 index 和最右邊的 index，我們要找到對於每個 r 他對應的 l 最遠可以到哪裡。以下範例陣列為 `1,2,3,1,4,4,5`，其答案為 4。





Greedy

Greedy演算法的核心策略是：在每一步都做出當下看起來最有利的選擇，例如選擇當下價值最高或成本最低的選項，而不考慮後續的影響或整體的最優解。

這種策略並不總是能保證得到全局最優解，但在某些結構良好的問題中，若經過合理轉換或與其他操作（如枚舉）搭配使用，Greedy 方法可以有效且快速地找到最佳或近似最佳的解答。

以 LeetCode 上的經典問題「Best Time to Buy and Sell Stock」為例：sell stock，假設你有每天的股價，你只能進行一次買入與一次賣出操作，目標是最大化利潤。

如果我們決定在第i天賣出股票，那麼最理想的情況就是在這之前的某一天以最低的價格買入。
因此，我們可以遍歷每一天，維護一個目前遇到的「最低價格」，並在每一步計算當下賣出可得到的最大利潤，持續更新答案。

這就是一種 Greedy 思維的應用，在每一天只關注當下的「賣出價」與「歷史最低買入價」，而不回頭檢查未來的價格，並搭配枚舉假設每一天賣出最後得到最佳答案。

以下為一個範例程式碼：

```
class Solution {
    public int maxProfit(int[] prices) {
        int minPrice = prices[0];
        int ans = 0;
        for(int i = 0 ; i < prices.length ; i++){
            ans = Math.max(prices[i] - minPrice, ans);
            minPrice = Math.min(minPrice, prices[i]);
        }
        return ans;
    }
}
```

Dynamic programming

Dynamic Programming（以下簡稱 DP）是一種利用程式具備大量記憶空間的特性，來有效計算出問題答案的方法。許多問題若直接求解會非常困難或耗時，但透過計算不同子問題的結果並將這些結果記錄下來，便能一步步建構出原問題的解答。

有些題目若採用 Greedy 無法保證找出最佳解，這時候就可能需要透過 DP 來解決。

範例說明：最少硬幣問題

假設一個國家的貨幣單位有 $2、$3、$5，我們想知道湊出剛好 $6 元時，最少需要幾個硬幣。

如果使用 **Greedy** 策略：每次都拿不超過當前需求中最大的幣值，我們會先選擇 $5，此時還差 $1，但 $1 是無法被現有幣值湊出來的。因此，這個策略失敗。

但如果改用 **DP** 方法，我們會先計算出湊出 $1、$2、$3、$4、$5 時，各自所需的最少硬幣數。接著，湊出 $6 的方式就能從這些狀態中組合出來：

- 拿一個 $2，加上湊出 $4 的最少硬幣數
- 拿一個 $3，加上湊出 $3 的最少硬幣數
- 拿一個 $5，加上湊出 $1 的最少硬幣數

這三種方案中，選擇使用硬幣數量最少的那個，即為湊出 $6 的最佳解。

DP的核心概念

DP 的核心由兩個要素構成：**state** 與 **state transition**。

- **State**：
  指的是問題中會變動的條件。我們通常會使用一維或多維陣列來儲存每個狀態的結果。
  以上述的換硬幣問題為例，我們可以定義 `dp[i]` 表示「湊出金額 $i 所需的最少硬幣數量」。

- **State Transition**：
  指的是如何從已知狀態推導出新狀態。
  在換硬幣的問題中，若我們想求出 `dp[i]`，可以透過以下幾種方式：
  - `dp[i] = min(dp[i - 2] + 1, dp[i - 3] + 1, dp[i - 5] + 1)`
  - 每次從一個幣值出發，試著將它加到先前已知的最小解中。

當我們把state跟state transition都定義好後，其實DP的程式也大概寫完了，以下是換硬幣的程式範例：

```
/**
 * denominations: 該國家可用的硬幣面額
 * amount: 目標金額
 * 回傳值: -1 代表無法用這些幣值湊出目標金額
 */
int changeCoins(int[] denominations, int amount) {
    int[] dp = new int[amount + 1];
    dp[0] = 0; // 湊出 $0 不需任何硬幣

    // 初始設定:其餘金額先設為 -1，表示尚未可達
    for (int i = 1; i <= amount; i++) {
        dp[i] = -1;
    }

    for (int i = 1; i <= amount; i++) { //計算每個state答案
        for (int j = 0; j < denominations.length; j++) {// state
```

```
transition
            int coin = denominations[j];
            if (coin <= i && dp[i - coin] != -1) {
                if (dp[i] == -1) {
                    dp[i] = dp[i - coin] + 1;
                } else {
                    dp[i] = min(dp[i], dp[i - coin] + 1);
                }
            }
        }
    }

    return dp[amount];
}
```

Time complexity

在 DP 中，time complexity通常等於：

狀態數量 × 每個狀態轉移的操作數

以換硬幣問題為例：

- 狀態數為 `O(amount)`
- 每個狀態要考慮 `|denominations|` 種轉移方式
  → time complexity 為 `O(amount × |denominations|)`

小技巧：如何定義 DP 狀態？

有時候我們會不確定要如何定義足夠又有效的狀態。最保險的做法是，把所有與問題有關的變動條件都當成 **state** 的維度。

例如：除了金額 `amount` 之外，還可以將「前幾種幣值」當成另一個狀態維度。

此時，我們可以定義 `dp[i][j]` 表示「用前 i 種幣值湊出金額 $j 所需的最少硬幣數」，這樣的狀態定義更完整，未來處理變化題也更具彈性，大家也可以想想這時的state transition變成如何。

Backtracking

Backtracking 是一種常見的暴力搜尋技巧，透過電腦快速運算的能力，有系統地嘗試所有可能的解答。這種方法常應用於益智遊戲的求解，例如 Sudoku, Tangram, 或目前 LinkedIn 上的小遊戲如 Zip, Tango, Queens。

其核心概念為「**Try and Error**」，在搜尋過程中不斷嘗試所有可能的解，若某條路徑不通，便回溯並嘗試其他選項。簡單來說，這就像是猜答案的過程。

許多人在解這類問題時，容易把解益智遊戲的直覺思路直接套用進程式邏輯中，反而會讓程式變得難以實作。與其硬套直覺，我們更應該思考如何將這些問題系統化，轉換成程式易於操作的格式。

提升效率的關鍵

Backtracking 最核心的挑戰在於「如何有效率地嘗試所有可能解」。這部分每個人見解不同，筆者認為一個重要的技巧是「設計一組具唯一性、明確的答案填寫順序」。這樣可以避免重複嘗試與無謂的運算，能達到這個條件，在大部分情況下就可以讓程式足夠有效率在幾秒內內跑出我們所需要的答案。

範例說明：**Sudoku（LeetCode 37）**

以 Sudoku 為例，我們可以從 1st row, 1st column 開始，依序從左至右、從上至下填數字。每個空格嘗試從 1 到 9，並檢查是否與該行、該列及其所屬 3x3 區塊內的數字產生衝突。這樣的填寫順序具備明確性與一致性，有助於提高搜尋效率。以下為 [leetcode 37 Sudoku Solver](#)的程式碼

```
class Solution {
    boolean backtracking(int x, int y, char [][] board){
        //代表所有數字的已經填滿
        if(x==9){
            return true;
        }
        //代表這個row填滿了，要去下一個row
        if(y==9){
            return backtracking(x+1, 0, board);
        }
        // 如果這個位子是題目給定的數字則跳過
        if(board[x][y] != '.'){
            return backtracking(x, y + 1, board);
        }
        for(char number = '1'; number <='9' ; number++) {
            boolean valid = true;
            // 檢查行與列是否有衝突
            for(int i = 0 ; i < 9 ; i++){
                if(board[x][i] == number) {
                    valid = false;
                    break;
                }
                if(board[i][y] == number) {
                    valid = false;
                    break;
                }
            }

            // 檢查 3x3 區塊是否有衝突
```

```
            // (blockX, blockY)為當前格子(x,y)所處區域最左上角的格子
            int blockX = (x / 3) * 3;
            int blockY = (y / 3) * 3;
            for (int i = blockX; i < blockX + 3; i++) {
                for (int j = blockY; j < blockY + 3; j++) {
                    if (board[i][j] == number) {
                        valid = false;
                        break;
                    }
                }
            }
            // 如果在(i,j)填入number是合法的，就繼續backtracking到下一個要填的位子，如果發現找到答案則直接return false,否則就將填的數字還原
            if(valid) {
                board[x][y] = number;
                if(backtracking(x, y+1, board)) {
                    return true;
                }
                // 發現找不到答案，將填的數字還原成"."
                board[x][y] = '.';
            }
        }
        return false;
    }
    //題目會給定一個9*9的數字，其中'.'代表這個位子沒有填數字
    public void solveSudoku(char[][] board) {
        backtracking(0, 0, board);;
    }
}
```

Backtracking 類型的題目，通常無法預先證明其有良好的time complexity，甚至在未實際執行程式之前，也難以確認其正確性與效率。

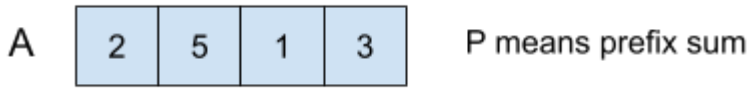
但實際上在現實生活中，這類型的題目很常遇到，譬如說給定一個益智遊戲，請撰寫一支程式自動尋找答案。除了考察 Backtracking 策略的運用能力，更重要的是：

1. 如何將問題轉換為程式輸入格式
2. 如何用合適的資料結構建模問題

以 Sudoku 為例，可以將題目用將 9×9 的array獨入，空格用特定符號(如 .)表示，再進行backtracking。

Prefix sum

prefix sum 是指計算一串數字裡各項元素到起點的總和或距離。例如一個有 n 個元素的陣列，我們會去算 $0\sim0$, $0\sim1$, $0\sim2$, ..., $0\sim n-1$ 的和。
他的優點是只要用 $O(n)$ 的時間做好前置操作，那後面每次對於某類型的區間詢問就只需要 $O(1)$ 即可得到答案。



P[0] = 2
P[1] = P[0] + 5 = 7
P[2] = P[1] + 1 = 8
P[3] = P[2] + 3 = 11

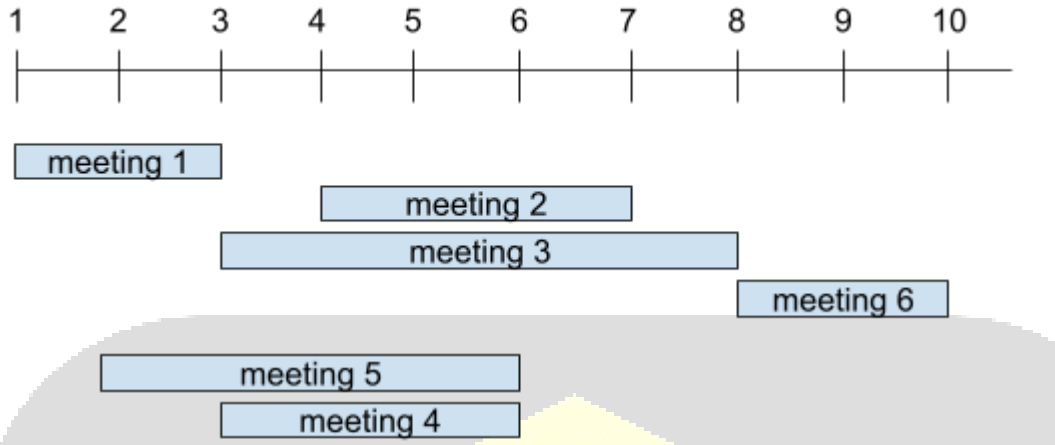
假設題目會問你一個陣列中 left 到 right 之間的數字總和是多少，這時候只要把 $P[right] - P[left]$ 就是答案了

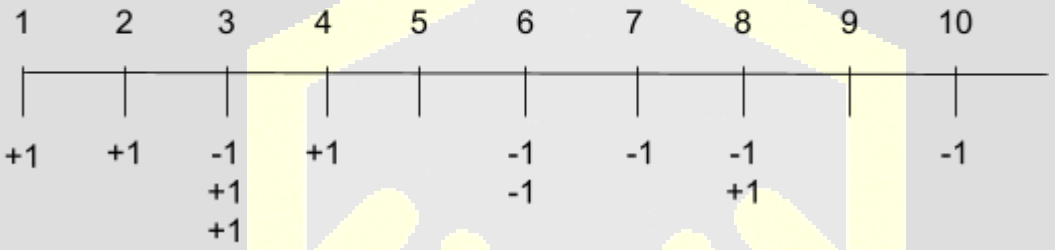
Prefix sum 進階應用

在 Prefix sum 的進階應用中，通常這類題目會給你很多個區間段，並問你區間段是否有重複，而區間段可能是時段或是線段。我們必須先將每個區間段拆成 start 和 end，例如 10 個區段會有 10 個 start 和 10 個 end。我們可以把 start 想成從這邊開始會開始有東西(+1)，而 end 想成從這邊開始會少一個東西(-1)。
我們可以確認好每個點會需要多幾個東西或少幾個東西，如果同一個點遇到多個 start 或 end 就將他們 +1, -1 全部合起來就會是那個點最後有幾個東西。之後如果要問某個點會有幾個重複的區段，使用前綴和的技巧，如果你知道前一個點的有幾個重複區段，那你只要再加上這個點有幾個東西的數字，你就會知道這個點有幾個重複的區段。

範例(Meeting room)

給定一堆會議的時段，試問最少需要幾個會議室才能滿足條件？
最少需要幾個會議室的意思其實就是問最多會有幾個會議時間有重疊，也就是我們要知道每個時間點重疊的會議數量。
以下用圖解釋作法，不過記得一般題目給的數字可能會很大或是用毫秒算，因此不能直接用陣列去存這些東西，我們需要用 map 之類的資料結構紀錄每個點的值，然後最後要從小到大排序，這樣才能知道前一個點是甚麼。因此此類題目時間複雜度是 $O(nlog)$。





P[1] = 1
P[2] = P[1] + 1 = 2
P[3] = P[2] + (-1+1+1) = 3
P[4] = P[3] + 1 = 4
…
P[10] = P[9] + (-1) = 0

In the process we should record currently max meeting counts
Thus, the minmum meeting rooms we need is 4.

Sorting

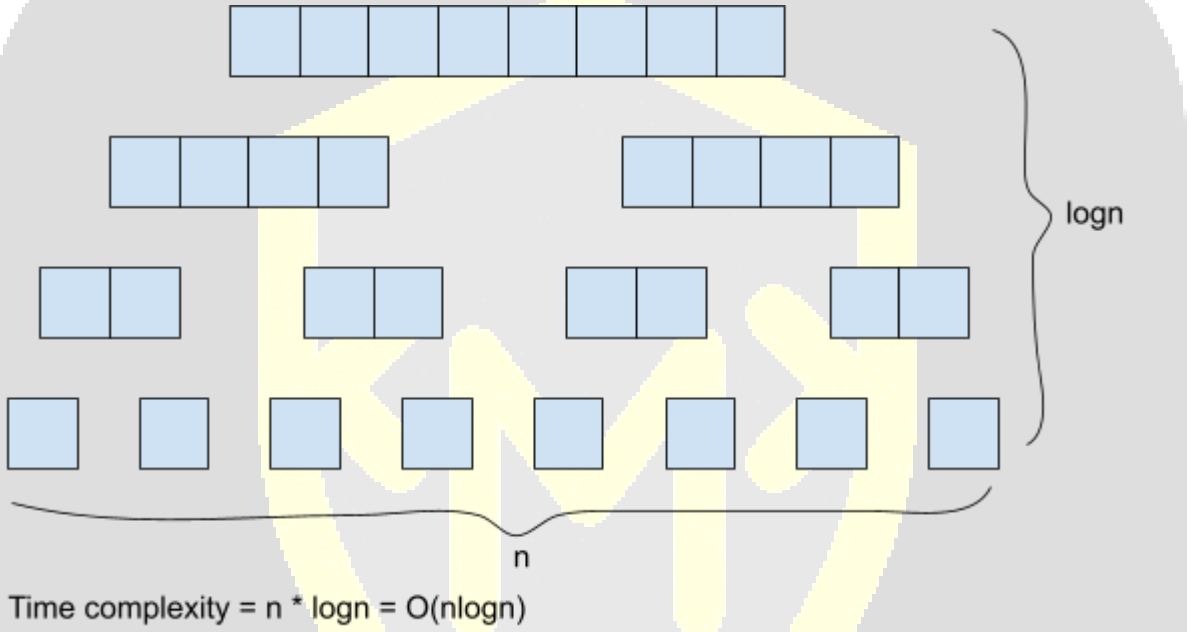
Sorting 是用來將一組元素依某種順序（例如由小到大）排列的操作。要使用排序演算法，最關鍵的是要定義好元素之間的比較規則（也就是比較函式）。

- 排序的實作方式大致可以分為兩大類：
  - 基於比較函式：這類排序使用比較函式來決定元素之間的順序，常見的策略有兩種：
    - 每次找出集合中的最小值 / 最大值
      - 代表性演算法：Heap Sort、Merge Sort
    - 確定某個元素在整體中的正確位置
      - 代表性演算法：Quick Sort
  - 非基於比較函式
    - 這類排序不使用元素間的比較，而是利用額外的資訊（如元素數值範圍）來進行排序：

- 代表性演算法：Counting Sort
- 此類方法的前提是元素的數值範圍有限（例如整數且範圍在 0～k 之間），否則空間成本會過高。

Merge sort (with Divide and Conquer)

Merge sort 會運用到 divide and conquer 的技巧，也就是拆解+合併的運算過程，整個運算的過程靠遞迴進行。以下會分成 divide 和 conquer 兩部分講解。
時間複雜度為 $O(nlogn)$，因為總共會將 array 分成 $logn$ 層，且每層都會有 n 個元素，可以把他想像成用長乘寬算面積，因此時間複雜度為$O(logn)\ *\ O(n)\ =\ O(nlogn)$

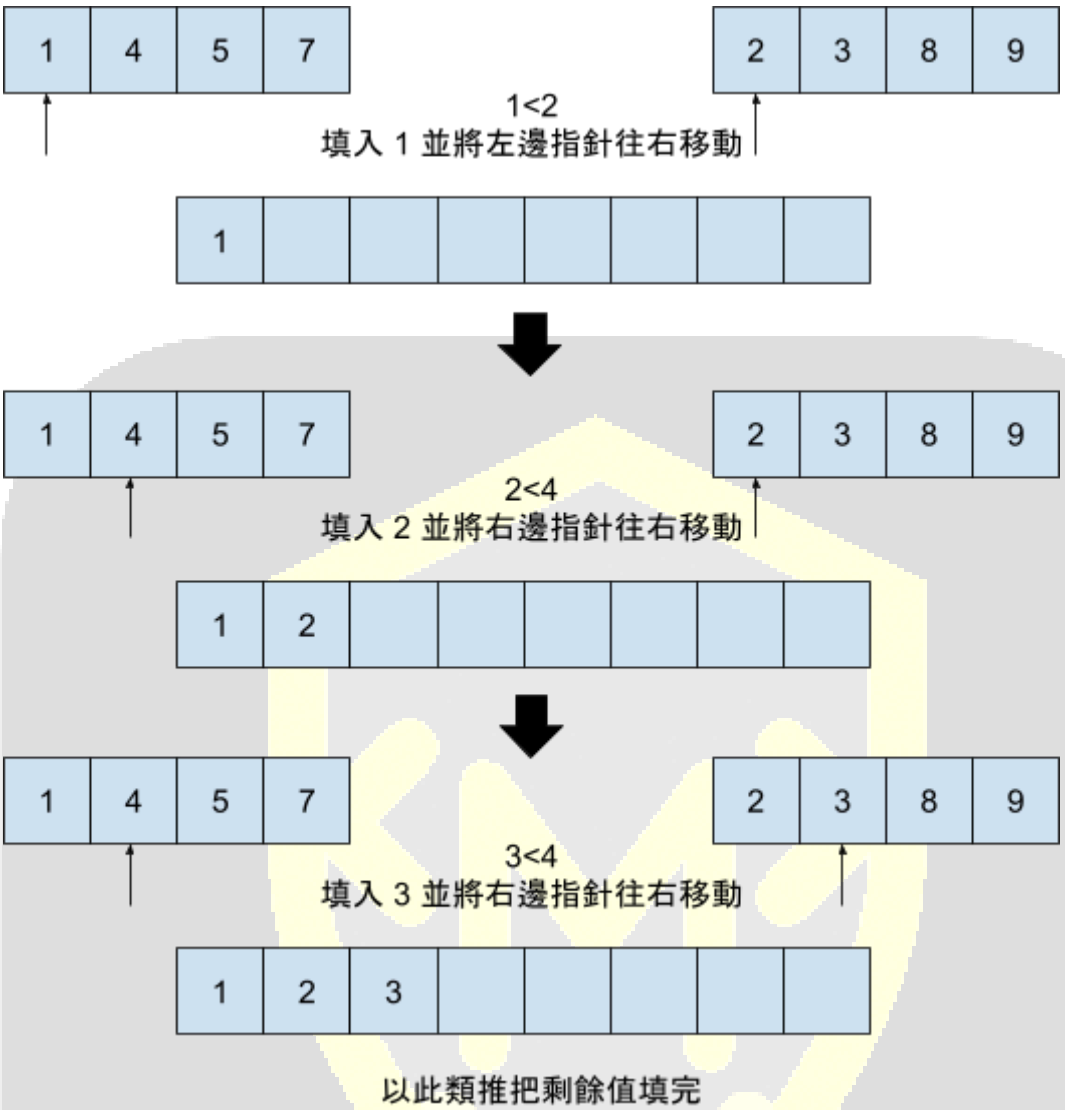


Divide

將原本 array 不斷拆成兩半再往下傳，在實作遞迴時將兩半陣列各自傳遞下去，直到拆到陣列長度為一時可開始做 conquer。

Conquer

由於左右兩半陣列都是 sorted array，因此可用以下方式合併兩個陣列。一開始將兩個指針各指在陣列的最左邊，然後比較兩邊的值大小，比較小的就將他拿出來放到新的陣列，然後把指針向右移動。當左右兩半陣列的指針都指向最後一格，即代表合併完成，可以回傳結果到上一層。
以下圖片為合併的示範，最後合併完可得到 `[1,2,3,4,5,7,8,9]`



Pseudo Implementation

```
void mergeSort(int[] arr) {
    // recursion end condition
    if (arr.length <= 1) {
        return;
    }
    int mid = arr.length / 2;

    int[] left = new int[mid];
    int[] right = new int[arr.length - mid];

    for (int i = 0 ; i < mid ; i++) {
        left[i] = arr[i];
    }
    for (int i = mid ; i < arr.length ; i++) {
        right[i - mid] = arr[i];
    }
```

```
    mergeSort(left);
    mergeSort(right);

    merge(arr, left, right);
}

void merge(int[] arr, int[] left, int[] right) {
    int i = 0, j = 0, k = 0;

    while (i < left.length && j < right.length) {
        if (left[i] <= right[j]) {
            arr[k++] = left[i++];
        } else {
            arr[k++] = right[j++];
        }
    }

    while (i < left.length) {
        arr[k++] = left[i++];
    }
    while (j < right.length) {
        arr[k++] = right[j++];
    }
}
```

Quick Sort

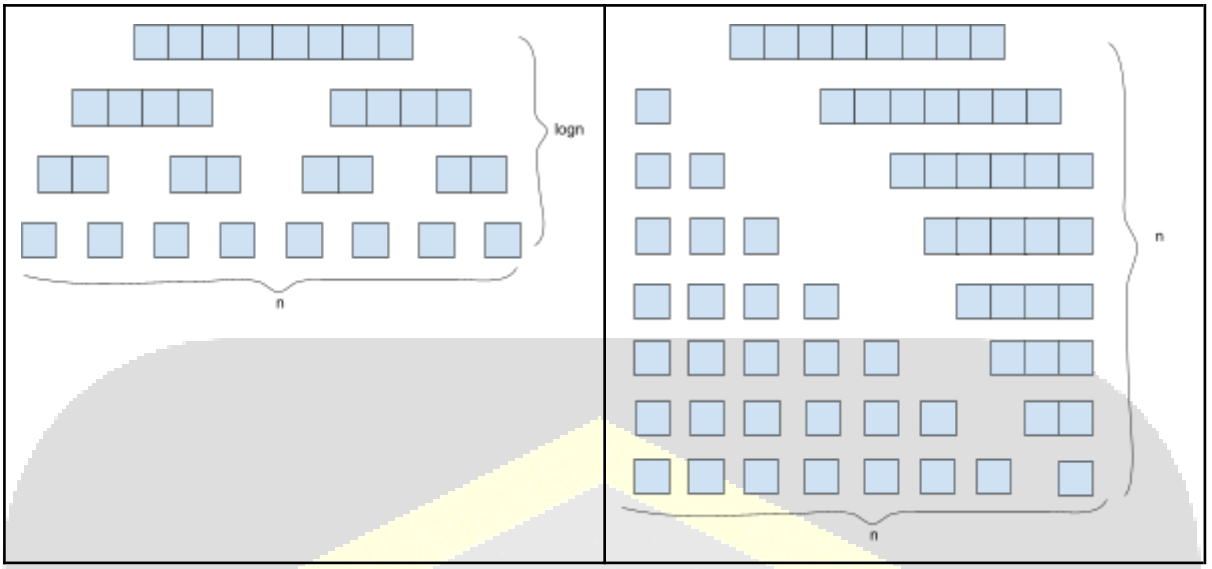
Quick sort 跟 merge sort 很像，都需要將陣列做劃分，且一路劃分到只剩一或零個元素為止，差別在 merge sort 是先全部劃分好才再做排列而 quick 是邊劃分邊排列。實作 quick sort 時，我們需要先找到一個 pivot(基準值)，尋找 pivot 的方法沒有一定的標準，可以隨機找也可以挑陣列裡第一個元素或最後一個元素。接著我們將比 pivot 小的值都移到他的左邊，比 pivot 大的值移到他的右邊。接著再從 pivot 的左半部跟右半部個挑一個新的 pivot 來排序左半部和右半部，照這個方式不斷拆分+排序子陣列就能排序好整個陣列。
以下用圖示為 quick sort 的過程，其規則為：

1. 選陣列中第一個元素當 pivot
2. j 的範圍為 [pivot + 1, a.length - 1]，當 j == a.length 代表所有元素已經跟 pivot 比較過了，可以交換 a[pivot] 和 a[i] 的值
3. i 的初始位置跟 pivot 一樣，若 a[j] < a[pivot]，則交換 a[i+i] 和 a[j] 的值，然後 i = i+1
4. 當交換完 a[pivot] 和 a[i] 的值後，pivot 被放置在正確的排序位置，接下來就分別重複上述動作處理 pivot 左邊和右邊的子陣列

Time complexity

Quick sort 的平均或最佳時間複雜度均為 $O(nlogn)$，不過要注意 quick sort 的最差時間複雜度會來到 $O(n^2)$，因為如果每次選的 pivot 都是最小或最大值那拆分的層數就會來到 n 層，因此時間複雜度會來到 $O(n * n) = O(n^2)$。

Best case $O(nlogn)$	Worse case $O(n^2)$



Pseudo Implementation

```
// select the first element as the pivot
void quickSort(int[] arr, int left, int right) {
    if (left >= right) return;

    int pivotIndex = partition(arr, left, right);
    quickSort(arr, left, pivotIndex - 1);
    quickSort(arr, pivotIndex + 1, right);
}
int partition(int[] arr, int left, int right) {
    int pivot = arr[left];
    int i = left + 1;

    for (int j = left + 1; j <= right; j++) {
        if (arr[j] < pivot) {
            swap(arr, i, j);
            i++;
        }
    }
    swap(arr, left, i - 1);
    return i - 1;
}
```

Counting Sort

Counting Sort，顧名思義是透過「計數」的方式進行排序。它特別適用於數字範圍固定且不大的情況。假設輸入的數字都落在 `0` 到 `limit - 1` 的範圍內，我們可以建立一個大小為 `limit` 的計數陣列 `count`，用來紀錄每個數字出現的次數。

具體做法是，對每個出現在原陣列中的數字 `i`，就讓 `count[i]` 加一。完成計數後，再依照 `count` 陣列的順序，將每個數字按照出現次數寫回原始陣列中，即可完成排序。

Counting Sort 的時間複雜度為 **O(n + limit)**，其中 `n` 為輸入陣列長度，`limit` 為數字的最大範圍。

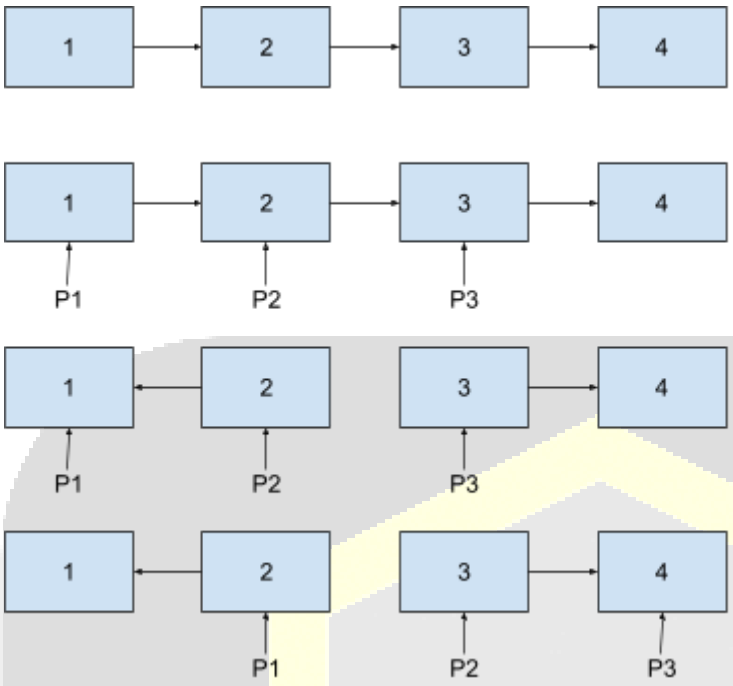
Pseudo Implementation

```
// select the first element as the pivot
void countingSort(int[] nums, int limit) {
    int[] count = new int[limit];
    for (int i = 0 ; i < nums.length ; i++) {
        count[nums[i]]++;
    }
    int index = 0;
    for (int i = 0 ; i < limit ; i++) {
        while(count[i] != 0) {
            count[i]--;
            nums[index] = i;
            index++;
        }
    }
}
```

Linked list

Main point

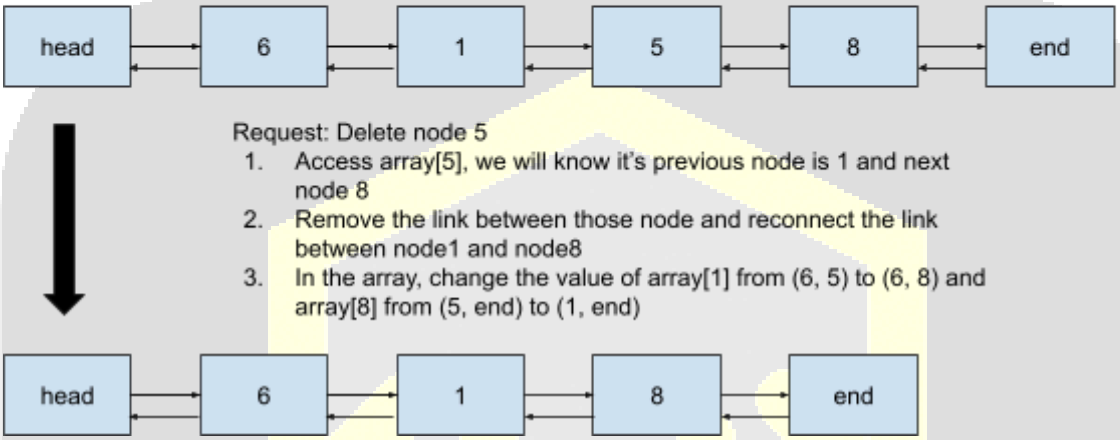
基本上 linked list 的題目所需要的操作就是：移除 link、增加 link和產生一個 pointer 指向 某個 node。而 linked list 最主要的重點是不管怎麼更改 link，都要確保每個 link 在刪掉之前他所指向的 node 都有別人指到，注意不能夠有孤兒 node 發生的狀況。
以下用反轉 linked list 的題目來示範如何確保每個 node 在操作時都還是會被指到



實作應用

很多人會好奇為甚麼解演算法題目的時候會用到 linked list，畢竟 array 就能夠儲存數值，使用 linked list 的優點何在。
在 array 要做 add 或 delete 一個值的操作需要 $O(n)$，反觀使用 linked list 配上 array 只需要 $O(1)$。以下用範例說明刪除一個 node 的操作
1. 藉由 array[5] 可得知 node 5 的前一個 node 是 1 下一個 node 是 8
2. 移除 node1 node5 和 node5 node8 之前的 link 然後重新連結 node1 和 node8
3. 在 array 裡將 array[1] 的值從(6,5) 修改為 (6,8)，array[8] 的值從(5,end) 修改為 (1,end)

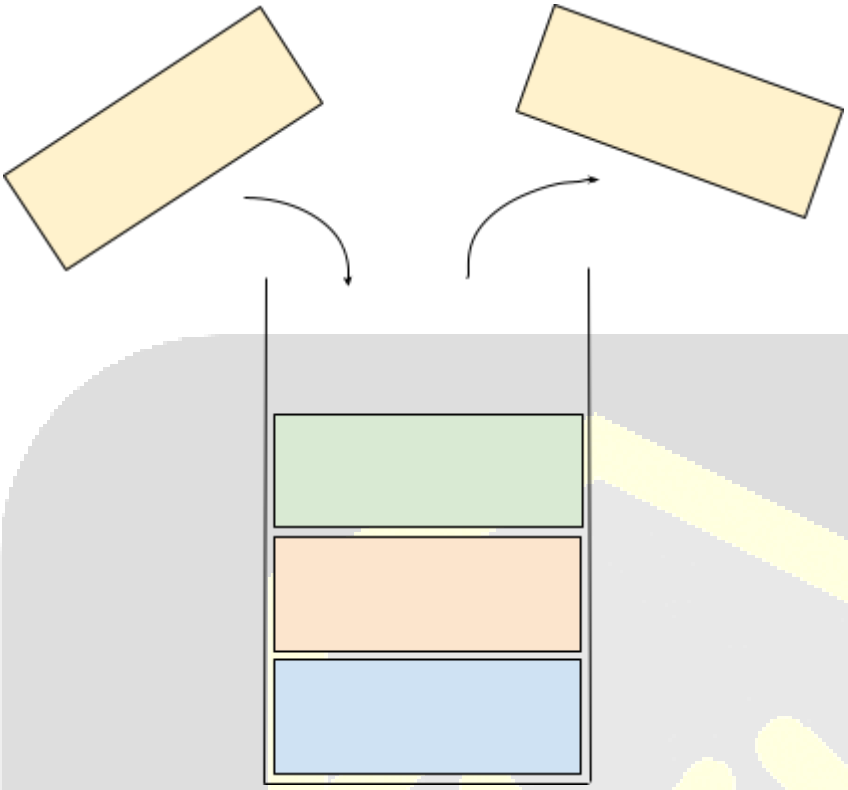
0	1	2	3	4	5	6	7	8	9	Index
?	6	?	?	?	1	head	?	5	?	Value: pointer to previous
?	5	?	?	?	8	1	?	end	?	Value: pointer to next



Data structure

Stack

Stack 的最重要的點是 first in last out (FILO)最先放入的元素會在最後才被取出。這種結構在生活中也常見，例如洗衣籃，最先丟進去的衣服，通常會被後面堆疊的衣服壓在下面，因此要最後才能拿出來。他最經典的題目是大家耳熟能詳的合法括號題:「檢查一串字串的括號是否合法」。我們會從最左邊開始遍歷字串，遇到左括號就把他丟到 stack 裡，而遇到右括號則會將 stack 最上面的左括號 pop 出來，如果兩者匹配的話即可繼續檢查，如果不匹配代表字串不合法，最後再檢查stack裡是不是空的就可以知道是不是合法的括號匹配，如果stack內還有東西代表還有左括號沒有配到右括號，因此不合法，反之所有括號都能匹配，因此合法。



Queue

Queue 的核心特性是 FIFO（First In, First Out），也就是先進先出。最先被放入的元素會最先被取出。這種結構在現實生活中非常常見，例如排隊，正常情況下，先到的人會先被服務。

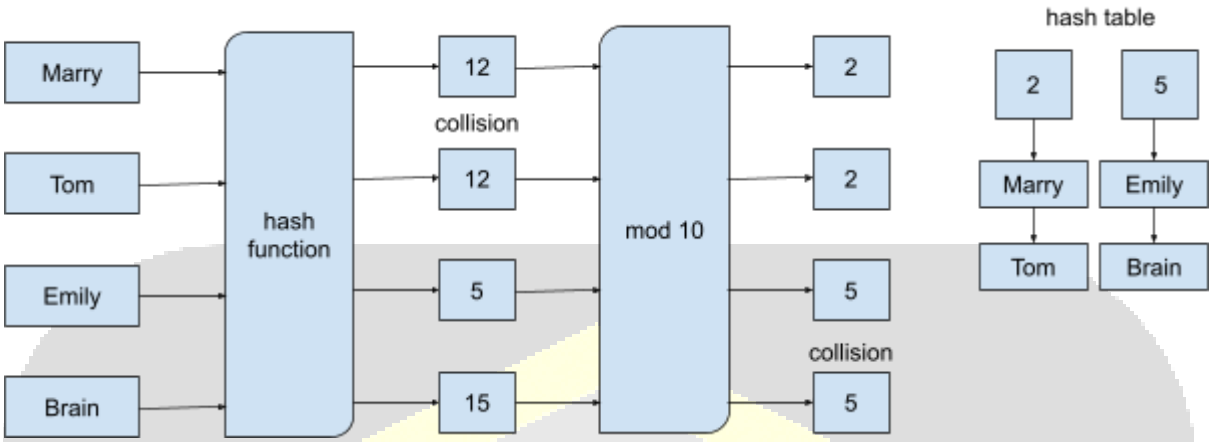
Queue 最經典的應用情境之一，就是在圖論演算法中的 BFS。在 BFS 中，我們會依序探索某個節點的所有鄰居，並使用 queue 來記錄接下來要拜訪的節點，確保搜尋是按層進行的，相關的觀念會在後面章節詳細提相關的觀念會在後面章節詳細提到。

Hash table

Hash table 的實作方式為創造一個 hash function 去 hash 要存進 table 的 objects，經由 hash function 加密後的 object 會變成一個 hash value（會是 integer）。由於 hash value 的值可能很大超過一般陣列可儲存的空間，因此一個常見作法會把 hash value mod 一個數字後再存進 table 裡對應 key 那格，至於要 mod 甚麼數字會由整個 table 大小做決定，因此數字也會隨之變動。注意不同的 objects 在經過 hash function 或 mod 之後可能會變成同個值，因而發生 collision。這時我們就會把這些 objects 放在同個 key 欄位裡（可以用二維陣列或 linked list 實作）。

例如假設 “Tom” 跟 “Marry” 在經過 hash function 之後得到的 hash value 都是 12，將 $12 \ mod \ 10$ 後得到 2，因此最後會將 “Tom” 跟 “Marry” 都放到 key 為 2 那格，而 ”Emily” 跟

”Brain”雖然 hash value 不相同，但mod出來後會是同一個值，因此在 hash table 內還是會儲存到同一個位子造成 collision。



當很多 key 下面存的 value 已經變非常多，不太能在用 $O(1)$ 的方式去拿到 value 時，就要考慮將 mod 的數字調整更大並同時調整整個 table 的存放。

比較 hash table 和 sorted map/sorted set

Hash table 因為是靠 key 直接去取得 value，只要整個 table 維護的夠好就能在 平均$O(1)$ 的時間內增加或刪除值，但如果一直collision最差情況可能會到 $O(n)$。
而 sorted map/sorted set 是靠平衡樹的方式去搜尋 value（可能是 balanced binary search tree或紅黑樹），因此需要花 $O(logn)$ 的時間去增加或刪除值。

使用時機

sorted map/sorted set 有順序性（會以大小或字典序做排列）但 hash table 靠的是 key 所以沒有順序性。因此今天如果題目是要找特定東西，且跟其他人無關就適合用 hash table。

例如「給定一個陣列有不同數字和一個目標數字，請問陣列裡是否有兩個數字加起來和為目標數字？」。我們可以遍歷陣列當遇到某個數字時，若此數字不在 table 裡就將目標數字減去它然後存在 table 裡，若此數字在 table 裡就代表之前有個數字跟他加起來會是目標數字。

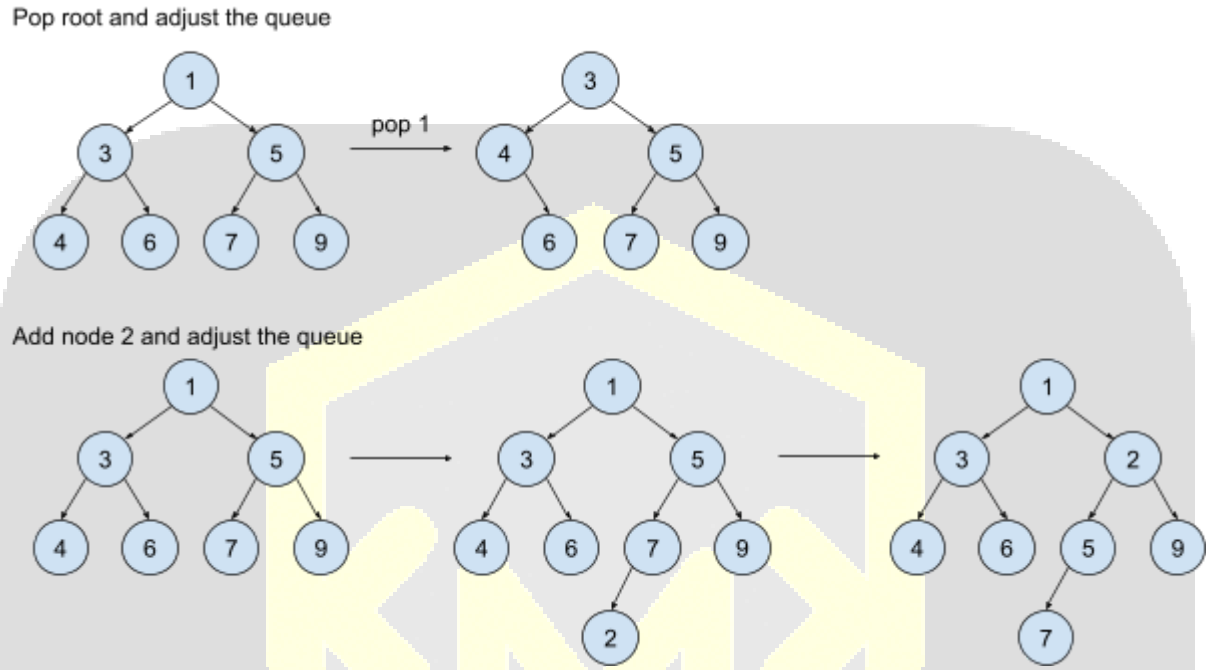
然而如果題目改成「給定一個陣列有不同數字和一個目標數字，請給我兩個數字加起來最接近目標數字？」就不能使用 hash table 而要使用 sorted map/sorted set，因為現在變成我們不是要找一個特定的值而是有可能需要附近的其他值。例如陣列為 `[2,4,6]` 目標數字為 5 今天當 2 找不到對應的數字合成 3，他就需要靠 sorted map/sorted set 的 binary search 去搜尋最近的數字為 4 可以加起來為 5。

Priority queue

原理

Priority queue 主要是讓priority最高的元素優先出來，通常是用 heap 來實作，而 heap 又分為 min heap 及 max heap，通常會是以 binary tree 的方式呈現，在 min heap 中根的值一定是最小，且他的所有子樹也都會保持根是最小值。當我們今天要加一個新的點進到 min heap

裡，我們會先把它放在最下面（本來是 null 的位置），然後再藉由和 parent 比較大小一路調整往上位移。此時間複雜度為 $O(tree的高度)$，因此若將該 binary tree 每一次盡量填滿，tree 的高度就會是 log n，因此複雜度就會是 $O(tree的高度)$。每次從跟 pop 出最小值後需要將新的最小值移到根，一種方式為一路往下維護他的子樹（若左子樹數值比右子樹小就把左邊的點一上去然後往下維護左子樹）。Max heap 的做法跟 min heap 一樣只是根放的是最大值。



另一種實作min heap的方法

我們剛剛介紹到min heap，可以藉由把下面的點推上來實作，但這時會發現，我們很難去確定說接下來要補值要補到哪個節點，及未來重複做刪除或加入節點時會不會造成該二元樹不平衡(實際上即便不平衡，他也可以維持在O(logn)的複雜度，此處的n為總共進入過heap的資料量)，因此又另一種做法為將最後一個節點補到根的位置

最後一個節點補到根

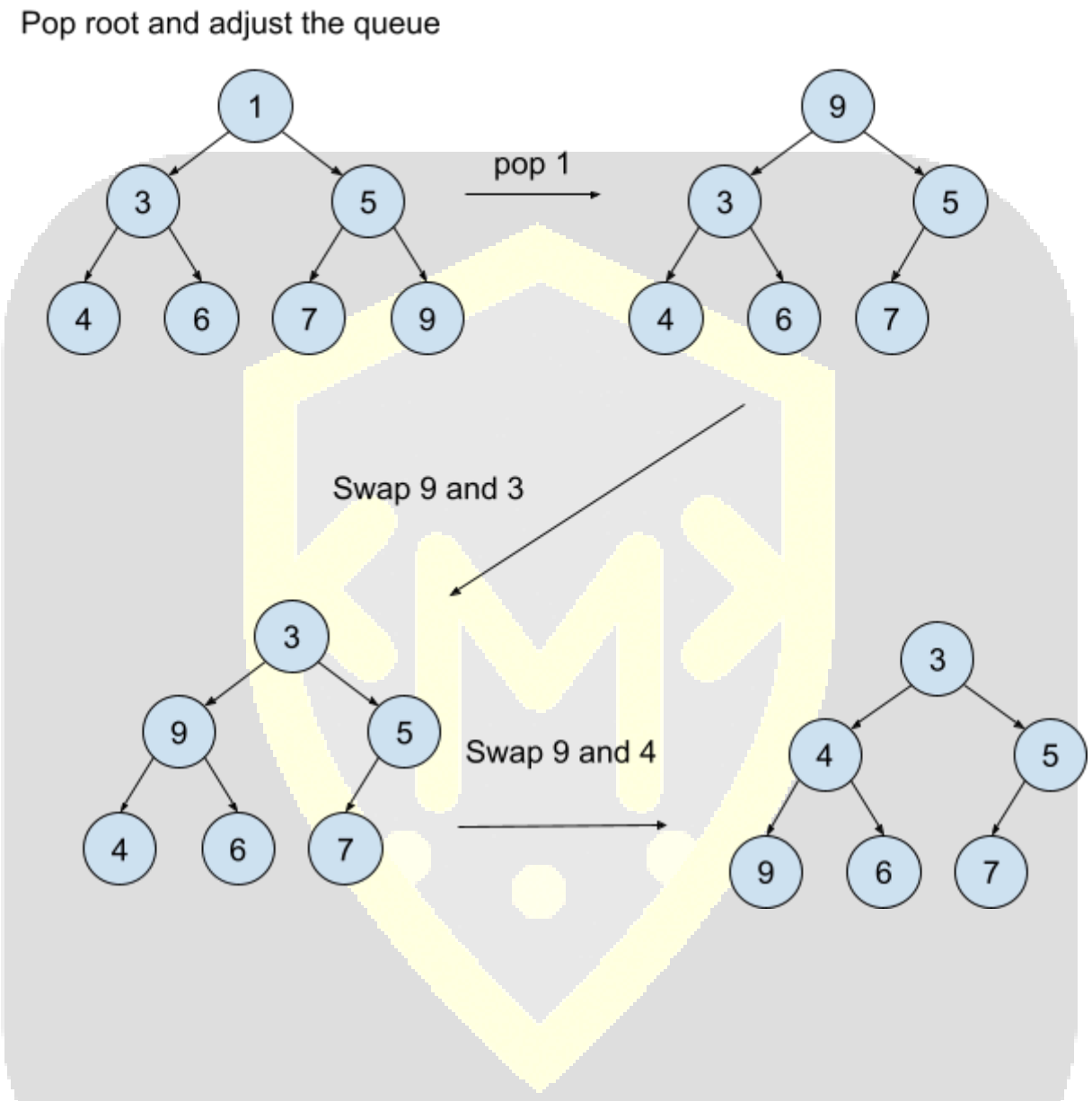
為了簡化這個問題，實務上我們通常使用另一種做法來實作 Min Heap，並在刪除最小值時採取以下做法：

1. 取出根節點（最小值 ） — 這就是我們要返回的結果。
2. 將最後一個節點（陣列尾端）移到根的位置 — 這樣可以確保陣列仍然代表一棵完全二元樹，因為只是「移走尾端」再「補到開頭」。
3. 從根節點往下調整（**Heapify**） — 依序與左右子節點比較，與較小的那個交換，直到滿足 Min Heap 條件。

這個方法的好處：

- 不需要額外尋找「下一個補位的節點」位置。
- 完全二元樹的結構自然維持，因為陣列末端移除與插入都是 O(1)。

- 下濾過程的時間複雜度仍然是 **O(log n)**。
- 以上過程可以使用array實作，可以有較小的常數。



範例

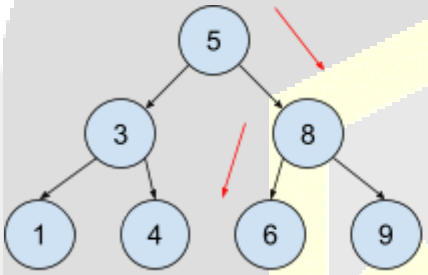
今股票會不定期上漲更新，使用者會不斷設定不同金額的 alert。當股票價格更新後，所有小於股票更新後價格的 alert 就都要被啟動通知使用者。題目的輸入形式會是「設定 alert」跟「更新價格」不斷交錯。例如輸入為 alert 155, alert 161, alert 157, update 150, alert 151，當股票更新成 160 時，alert 155 跟 alert 157 這兩個 alert 應該被啟動。
此題作法為建立一個 min heap，每收到一個新的 alert 就把放入 priority queue 裡，而每收到一個 update 訊息後就一次從 queue 裡面 pop 出最小值，直到他的最小值大於股票 update 後的值。
如果總共有 n 個 alert 跟 m 跟 update，建立 priority queue 的時間複雜度會是 $O(nlogn)$ 而經過 m 個 update 需要 $O(mlogn)$。每次 pop 出最小值需要花 $O(logn)$ 來維護整個 queue。

（如果今天題目的 alert 跟 update 一開始就給好而不是一個一個給，那使用 sorted array 處理的時間複雜度也差不多，priority queue 在這題的好處就是當有新的 alert 加進來時使用 $O(logn)$ 就能維護）。

Balanced binary search tree

原理

Binary search tree 為一個有 root 的樹，通常是以一個二元樹形式，也就是一個節點最多只有左右兩個子節點，每一個節點上有一個數字，且該節點左子樹的所有節點的數字都比該數字小，該節點右子樹所有節點的數字都比該數字大，因此要搜尋數字存不存在只要從root根據搜尋數字與當下節點數字的大小關係來決定要走哪裡即可。



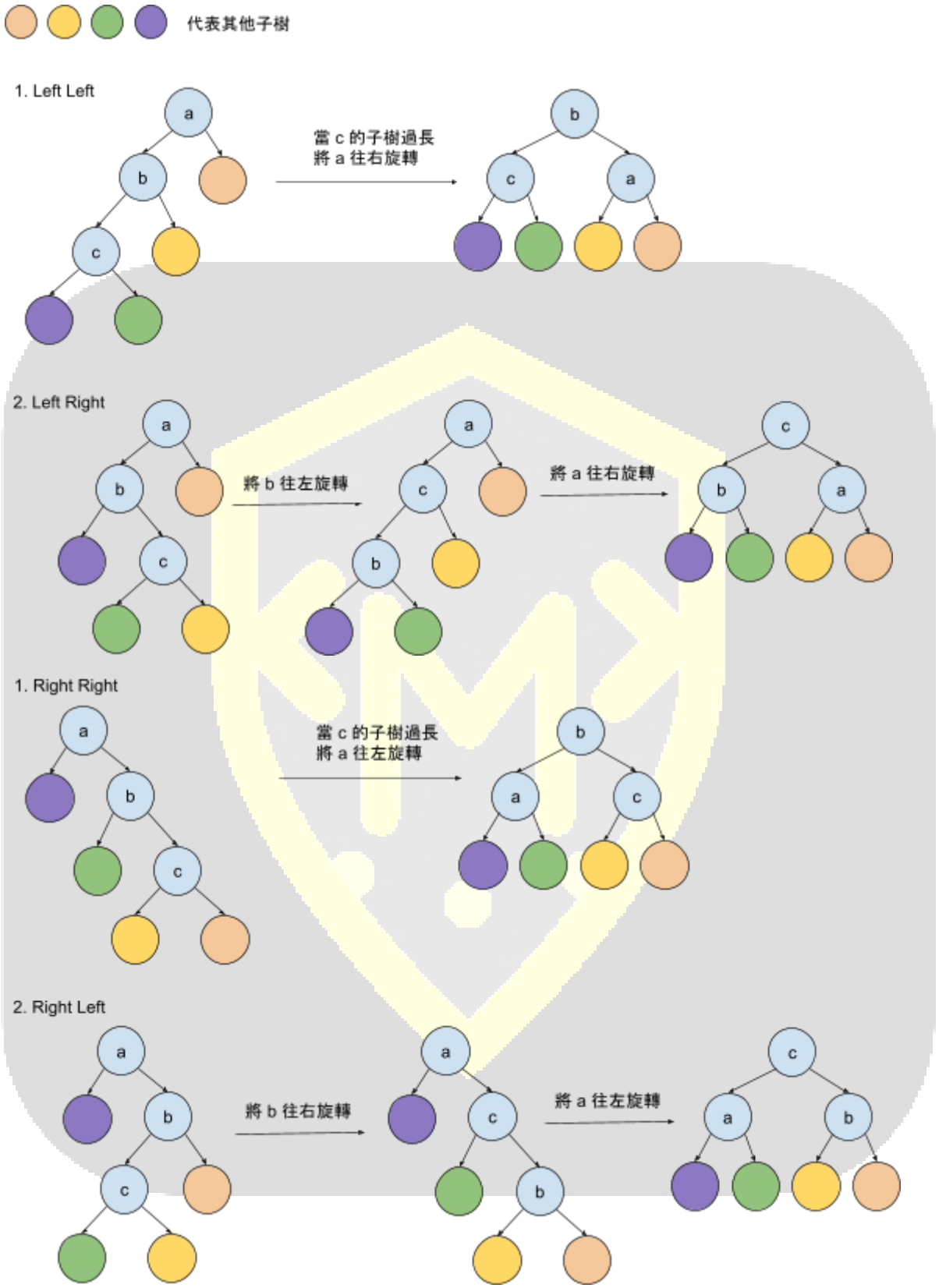
搜尋 6 這個點的步驟：

1. 從 root 往下搜尋，6 > 5 所以要往右子樹走
2. 來到 node 8，6 < 8 所以要往左子樹走
3. 走到 node 6

而 balanced 的目的是想要支援快速的加入/刪除元素，其主要核心概念為保證從root到任何一個 node 最多經過 $log(n)$ 個的 node，大部分的程式語言都有相關的 library 可維護加入/刪除元素，並讓他保持 balanced，

平衡方式

先判斷好 a, b, c 這三點誰的值在中間那最後那個點所在的位置就會是 root。因此在旋轉時要確保最終結果會符合上述安排。
下列為四種平衡 binary tree 時的情況以及平衡方式：



由以上的平衡方式，我們可以將任何一個node的兩個子樹最深的深度相差維持在<=1，沒有子樹的情況我們定義深度為0，而當插入或刪除節點破壞了平衡條件時，會透過以上四種平衡操作使的樹高維持在O(log n)，以上的介紹即為二元平衡搜尋樹AVL tree的相關介紹。

為什麼 AVL Tree 的高度最多是 O(log n)

對於一棵高度為 h 的 AVL 樹，要達到最少節點數，我們希望它的兩個子樹的高度差恰好為 1（因為差 0 會有更多節點），我們可以得到以下式子，(node(h)為高度為h的AVL樹最少需要的節點數量)。

node(h) = node(h-1) + node(h-2) +1 // 一個子樹的高度為h-1，另一個為h-2，+1則為該子樹的根節點(root)

觀察上式，這其實就是 **Fibonacci** 數列（只差一個常數 1），所以：

node(h) ≥ F(h+2) −1

其中 F(k) 是第 k 個 Fibonacci 數。

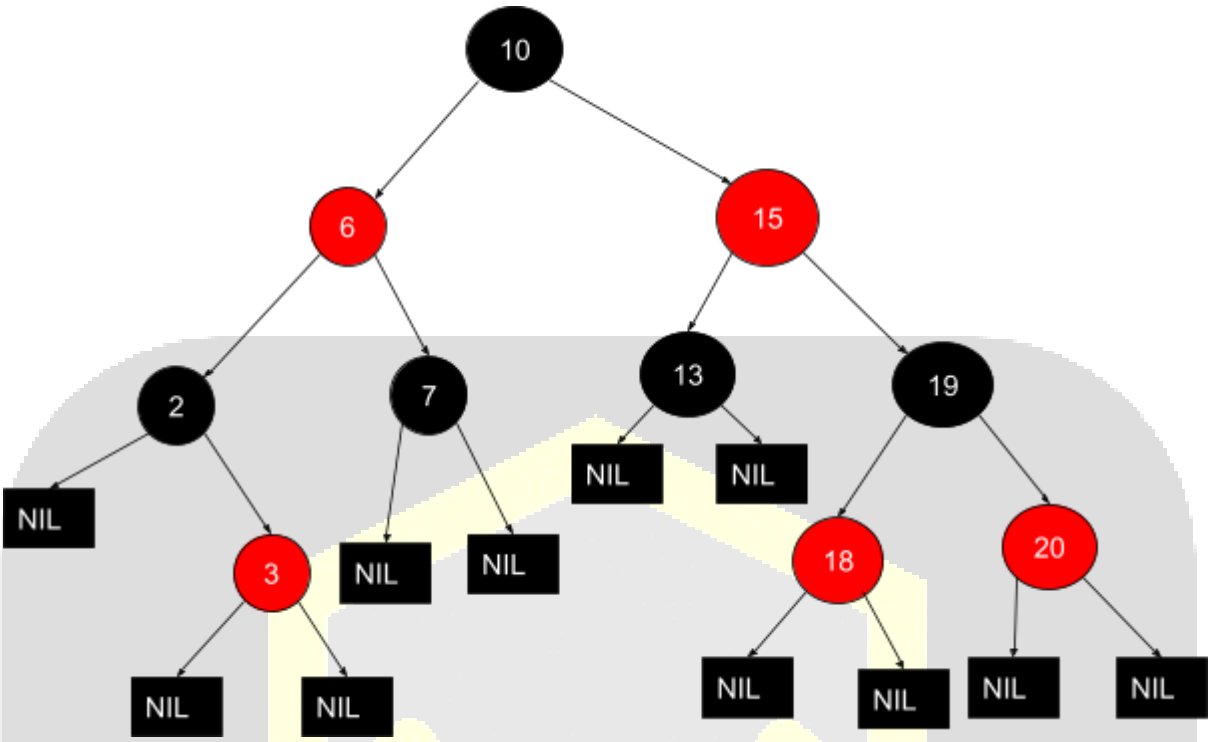
而F(k) 為一個近乎1.5倍成長的數字，因此要建成AVL樹的最少點數，也會近乎於1.5倍成長，因此可以反推出一個n個節點的AVL樹高度的複雜度約為O(log n)。

紅黑樹簡介

紅黑樹（Red-Black Tree）是一種自平衡二元搜尋樹（BST），它透過對節點著色並遵守一組規則，來保證樹的高度接近對數階（O(log n)）。

每個節點有一個顏色標記（紅色或黑色），並必須滿足以下 紅黑規則：

1. 每個節點不是紅色就是黑色。
2. 根節點一定是黑色。
3. 每條從根到葉子的路徑上，黑色節點的數量都一樣。
4. 紅色節點不能有紅色的子節點（不能連續兩個紅色）。
5. 所有葉子節點（**NIL** 節點，該節點不存在數值 ）視為黑色。



而紅黑數的實作較為複雜，因此這邊就不贅述了，有興趣的學員可以自行搜尋相關關鍵字。

Graph

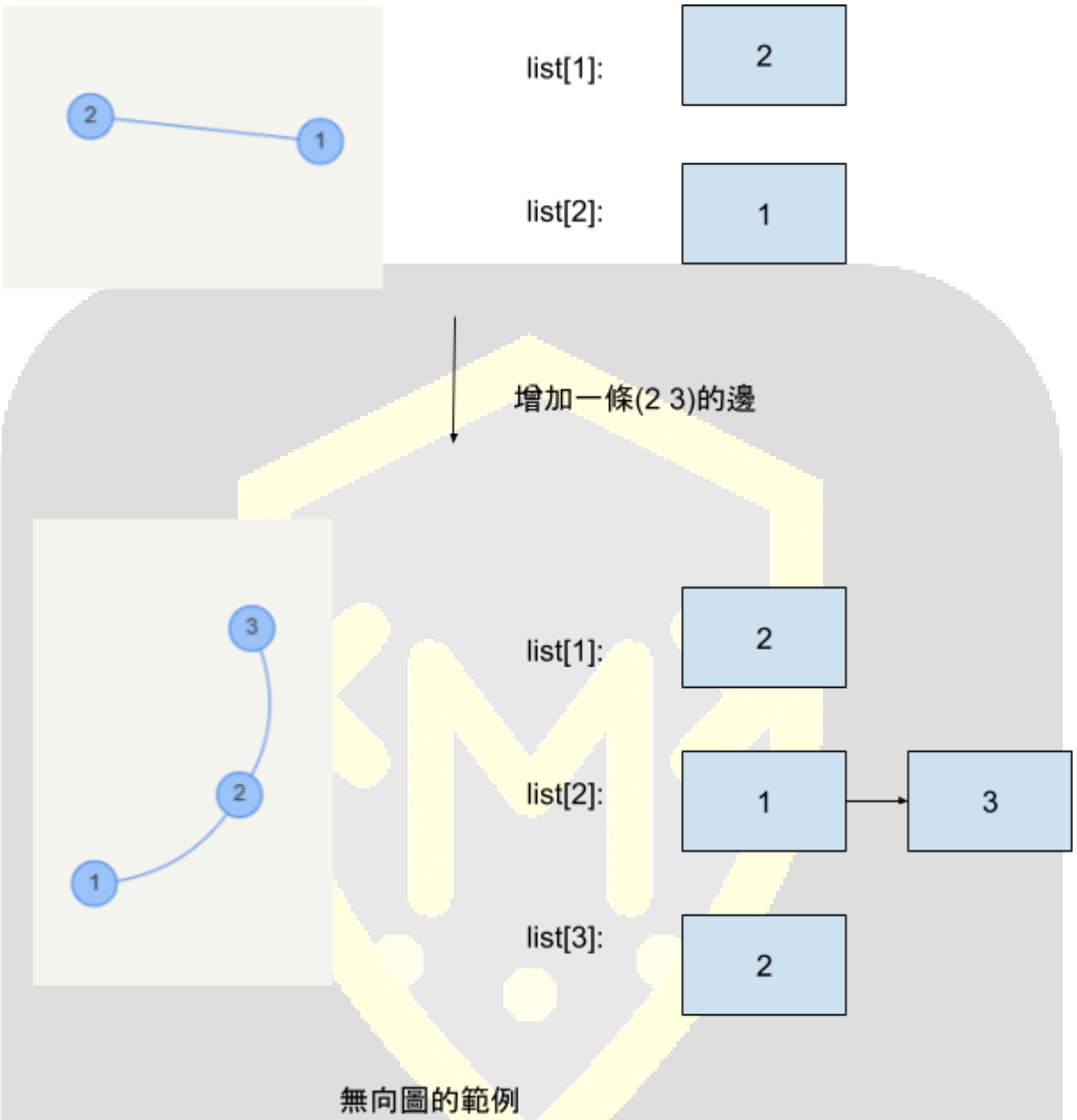
**DFS 跟 BFS 的區別在於一個是 stack 一個是 queue*

Adjacency Matrix/ List

Adjacency matrix 或 Adjacency list 通常適用於 graph 的題目，他們的目的是方便儲存點跟邊的資訊

Adjacency List

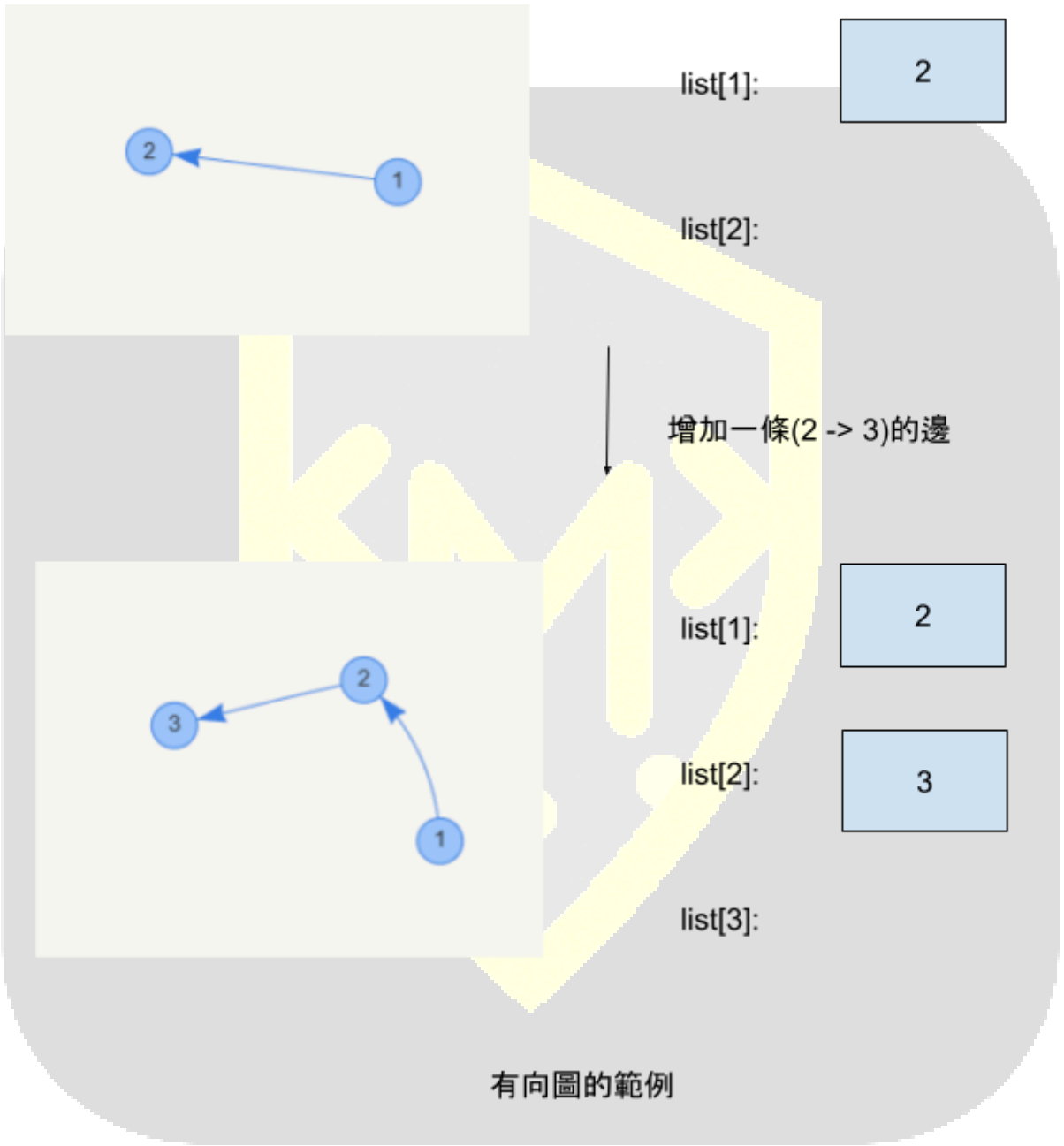
List 的存法怎麼設定因人而異，不過通常會先開好所有點的陣列然後再把它連到的點 push 進去。
今有一邊連結 x 和 y 兩點，若是無向圖（undirectly graph）則在 x 那格中存放 y，在 y 那格也會存放 x，也就是我們會將一條邊拆成兩條有向邊一條為 x->y 一條為 y->x 。若是有向圖（directly graph）且為 x 指向 y，那則在 x 那格中存放 y 即可




```
List<Integer>[] adjacencyList;
void init(int n) {
    // 初始化一個n個點的graph,  需要+1是因為有可能點的數字從1開始
    adjacencyList = new ArrayList[n + 1];
    for (int i = 0; i <= n; i++) {
        adjacencyList[i] = new ArrayList<>();
    }

}
void addEdge(int u, int v) {
```

```
    // 增加一條u到v的邊
    adjacencyList[u].add(v);
    // 增加一條v到u的邊
    adjacencyList[v].add(u);
}
```



有向圖的範例

```
List<Integer>[] adjacencyList;
void init(int n) {
    // 初始化一個n個點的graph,  需要+1是因為有可能點的數字從1開始
    adjacencyList = new ArrayList[n + 1];
    for (int i = 0; i <= n; i++) {
        adjacencyList[i] = new ArrayList<>();
    }
```

```
}
void addEdge(int u, int v) {
    // 增加一條u到v的邊
    adjacencyList[u].add(v);
}
```

Adjacency Matrix

一開始開好一個 $n * n$ 的二維陣列，並將所有值都設為零。今有一邊連結 x 和 y 兩點，則把 x, y 那格改成 1。

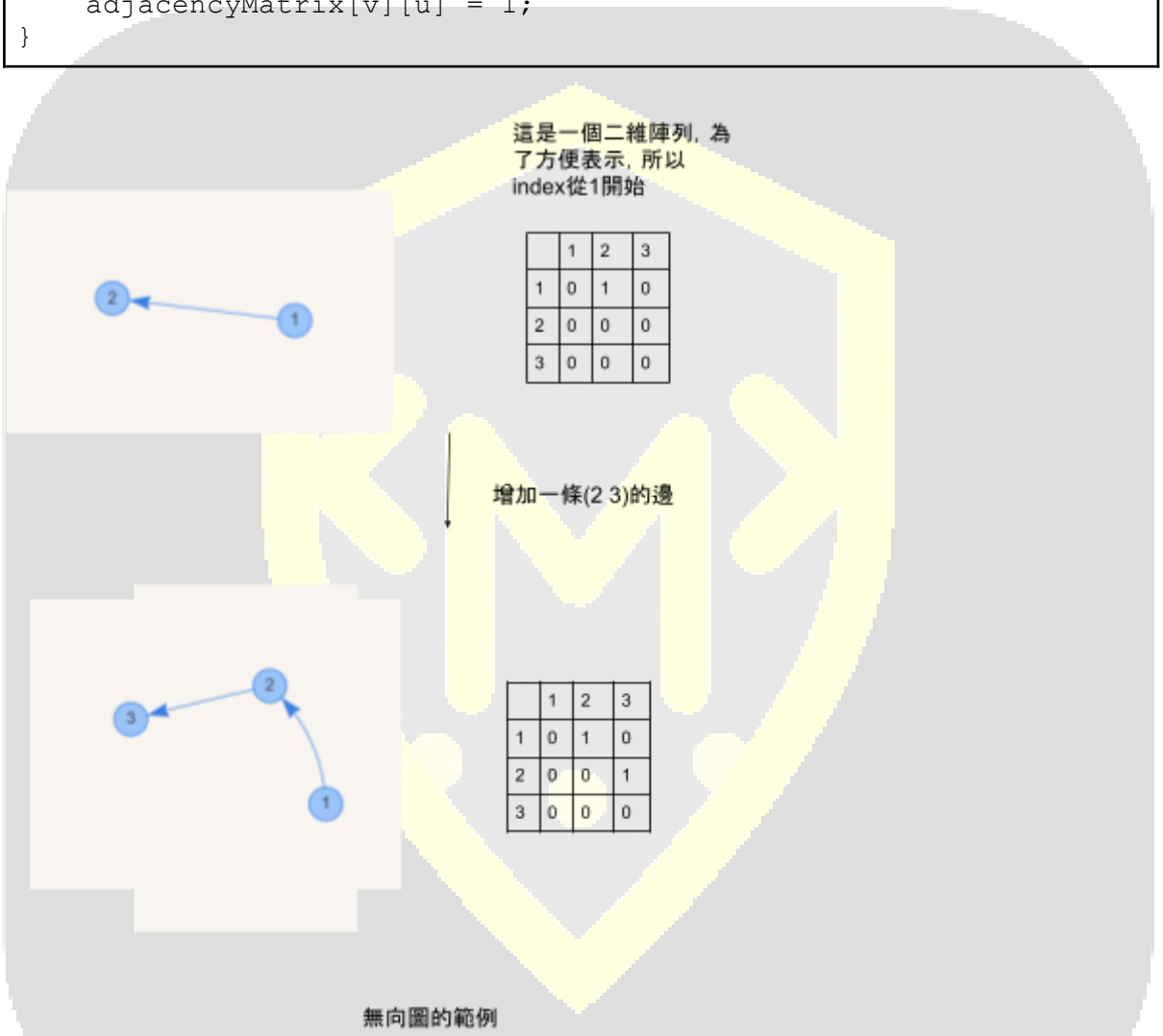
如果圖的邊很稀疏時適合用 adjacency list，因為最佳情況可能只會用到 $O(m)$ 空間，但若是用 matrix 空間一定是 $O(n^2)$。一般狀況用哪個方式存資料都一樣，不過如果題目常常問 x, y 兩點中間有沒有邊的話就適合用 matrix 因為 $O(1)$ 即可知道結果。

	1	2	3
1	0	1	0
2	1	0	0
3	0	0	0

	1	2	3
1	0	1	0
2	1	0	1
3	0	1	0

無向圖的範例

```
int[][] adjacencyMatrix;
void init(int n) {
    // 初始化一個n個點的graph,  需要+1是因為有可能點的數字從1開始
    adjacencyMatrix = new int[n + 1][n + 1];
}
void addEdge(int u, int v) {
    // 增加一條u到v的邊
    adjacencyMatrix[u][v] = 1;
    // 增加一條v到u的邊
    adjacencyMatrix[v][u] = 1;
}
```



	1	2	3
1	0	1	0
2	0	0	0
3	0	0	0

	1	2	3
1	0	1	0
2	0	0	1
3	0	0	0


```
int[][] adjacencyMatrix;
void init(int n) {
    // 初始化一個n個點的graph,  需要+1是因為有可能點的數字從1開始
    adjacencyMatrix = new int[n + 1][n + 1];
}
void addEdge(int u, int v) {
    // 增加一條u到v的邊
    adjacencyMatrix[u][v] = 1;
}
```

轉換題目成 Graph 形式

將題目轉換成圖的要點就是定義好甚麼是點甚麼是邊，且有可能點會指向自己。以 Leetcode 200 題為例：
給定一個二維陣列包含 1 和 0，1 代表島 0 代表海，上下左右相鄰的 1 代表同個島，試問這個地圖裡有多少個島？
這題的點就是陣列裡值為 1 的元素，而邊則是由相鄰兩個 1 產生。想要知道有幾個島可以遍歷所有 1 的點，使用 DFS 把經過的點標起來，這樣走過的點就不會再被算到。

範例

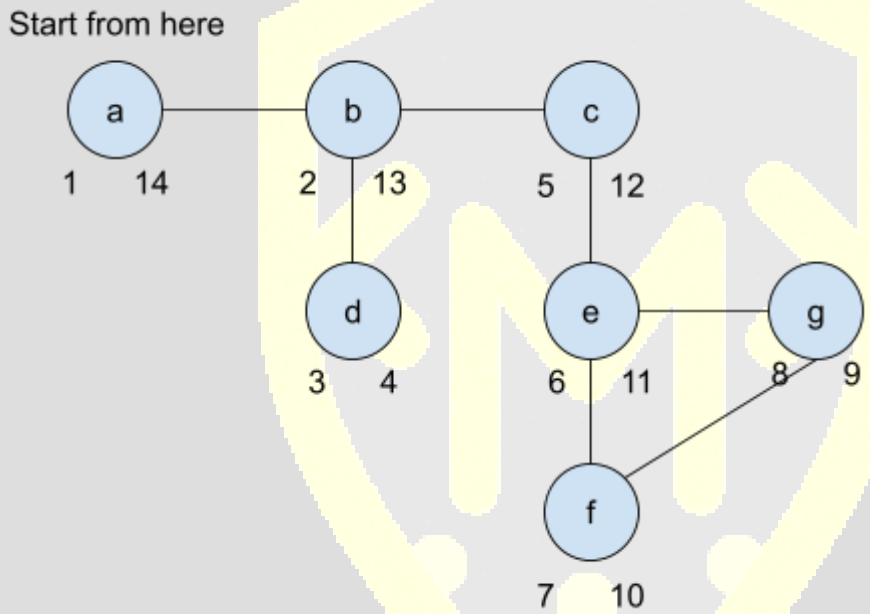
```
public class Solution {
	int n,m;
	static int[] dx = {0,0,1,-1};
	static int[] dy = {1,-1,0,0};
	static int directionCount = 4;
	private void dfs(int x, int y, int[][] grid) {
		grid[x][y] = 0;
		for (int i = 0 ; i < directionCount ; i++) {
			int targetX = x + dx[i], targetY = y + dy[i];
			if (targetX >= 0 && targetX < n && targetY >=0 &&
		targetY < m && grid[targetX][targetY] == 1 ) {
				dfs(targetX, targetY, grid);
			}
		}
	}
	public int numberOfIslands(int[][] grid) {
		n = grid.length;
		if (n == 0) {
			return 0;
		}
		m = grid[0].length;
		int ans = 0;
		for ( int i = 0 ; i < n ; i++) {
			for ( int j = 0 ; j < m ; j++) {
				if (grid[i][j] == 1) {
					dfs(i,j,grid);
					ans++;
				}
			}
		}
		return ans;
	}
```

```
}
```

Depth-first search (DFS)

DFS 的實作方式運用到了 Stack 的技巧，每遇到一個點就將點放入 stack，而最先遇到的點會最後才被拿出。
實作需要使用遞迴遍歷每個點，每走到一個點會用迴圈遍歷每個與此點連接的點，若連接的點沒走過就繼續走下去，走過的點會標記起來這樣才不會重複走到（可以設一個 visited[n] 的陣列紀錄點是否有走過）。

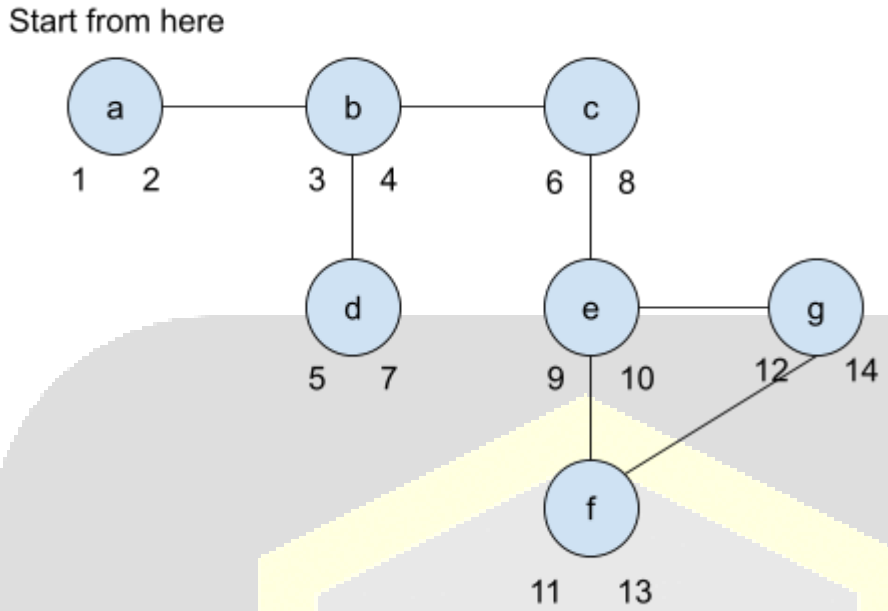
下圖為使用 DFS 遍歷圖時點進入跟移出 stack 的順序，點左下的數字代表進入右下角代表移出。



Breadth-first search (BFS)

BFS 的實作方式運用到 queue 的技巧，使用迴圈遍歷每個點，如果此點還未經過就把它塞進 queue。使用 while 迴圈判斷 queue 是否為空，不是的話就將裡面的點丟出來然後去看跟此點相連的其他點，並將和他相連的點且還未標記過的點（一樣可以設一個 visited[n] 的陣列紀錄點是否有走過）丟進 queue。

下圖為使用 BFS 遍歷圖時點進入跟移出 queue 的順序，點左下的數字代表進入右下角代表移出。



最短路徑

只有圖的邊長都是 1 的情況才可以用 BFS 實作找尋最短路徑，因為當邊長不一樣時我們得確保是邊長最短的點先被 pop 出來。以下兩種情況要以別的方式實作：

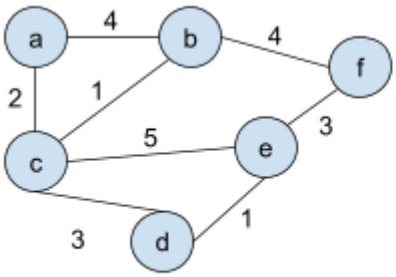
邊長只有 0 和 1

使用 double end queue 存放邊，如果邊長是 1 就將此點從 queue 後面放入，如果邊長是 0 則是將此點從 queue 前面放入，這樣才能讓 queue 維持前面數值比較小。這樣 0 的邊才會先被拿出來。

邊長長度都不同

使用 Dijkstra Algorithm 會需要用到 priority queue。實作方式為一開始全部點與原點的距離都設為無限大，並把原點的值改為 0 放入 queue，並依次更新與原點相連的點並將其放入 queue。之後我們需要將目前在 queue 離原點最近的點 pop 出來去更新其他點的值，這樣確保每個點能夠更新別人的前提是他已經不會在被更動了（也就是此點與原點的距離已經是是最小值）。更新點的時候要記得更改他的 parent（parent 代表是哪個點去更新這個點），這樣在最後才能由最後一個點推得最短路徑。
為甚麼每次都 pop queue 裡面最小的值能夠維持最短路徑，因為如果這個點已經是目前最小的值那就不可能有任何其他點能讓這個點的值變更小，因此可以拿這個點去更新其他點。

以下示範從 a 點到其他點的距離更新方式：



Point	Distance	parent
a	0	a
b	infinity	null
c	infinity	null
d	infinity	null
e	infinity	null
f	infinity	null

Calculate the distance from a to other points

Point	Distance	parent
a	0	a
b	4	a
c	2	a
d	infinity	null
e	infinity	null
f	infinity	null

pop a
push b, c

Point	Distance	parent
a	0	a
b	3	c
c	2	a
d	5	c
e	7	c
f	infinity	null

pop c
push d, e

Point	Distance	parent
a	0	a
b	3	c
c	2	a
d	5	c
e	7	c
f	7	b

pop b
push f

Point	Distance	parent
a	0	a
b	3	c
c	2	a
d	5	c
e	6	d
f	7	b

pop d

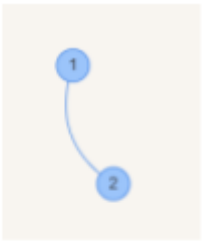
Point	Distance	parent
a	0	a
b	3	c
c	2	a
d	5	c
e	6	d
f	7	b

pop e

Disjoint sets

Disjoint sets(併查集)是管理不同的集合，並快速判斷他們是否為同個集合，以及快速合併兩個不同的集合。Disjoint sets 常見的功能為以下三種：

- Make(x)：建立一個新的只包含元素 x 的集合，parent 值也設成自己
- Find(x)：返回包含元素 x 的集合的代表元素。
- Union(x, y)：將 x 和 y 的兩個集合合併為一個。



parent[x] 的目的為找到 x 所處的聯通塊代表的點

x	1	2
parent[x]	1	1

假設我們為圖增加一個點 3
會將他的連通塊的代表點設成他自己，因為該連通塊只有他一個點而已

x	1	2	3
parent[x]	1	1	3

接著我們加入(2, 3)的邊

3跟其他點屬於同一個連通塊了，因此將他的parent設為與其他人的代表點相同

x	1	2	3
parent[x]	1	1	1

通常每個元素會存放自己的值以及 parent 的值，Find(x) 就是藉由不斷去尋找 parent，當自己的值等於 parent 時就代表找到此集合的頂點。兩個集合合併時，若追求時間優化就會將小集合指向大集合，也就是 merge by rank 的做法（將小集合頂點的 parent 指向大集合的頂點）

Find(x)

```
int Find(int x) {
    if (x == parent[x])
        return x;
    // 路徑壓縮方式，在過程中讓 child 的點直接指向最頂層的點
    return parent[x] = find_set(parent[x]);
}
```

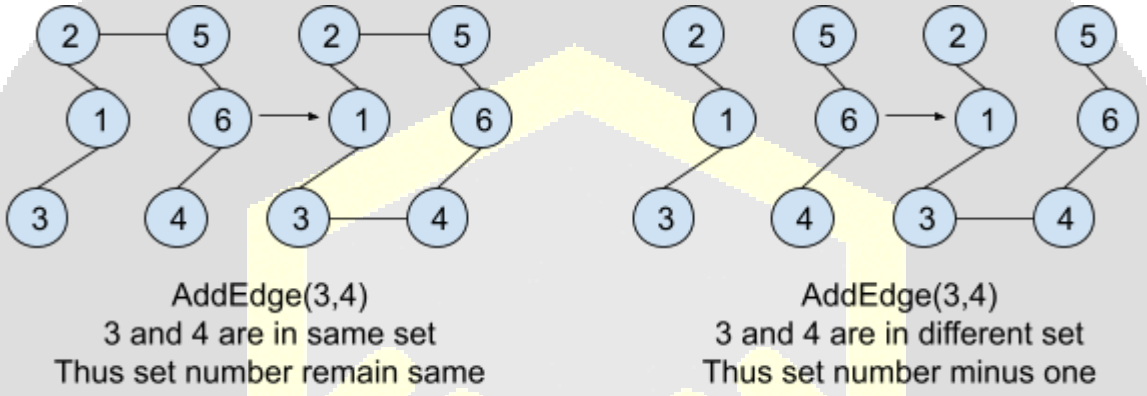
Union(x,y)

```
void Union(int x, int y) {
    x= Find(x);
    y= Find(y);
    // 可以用 size 來記錄哪個集合比較大再去合併
    if (x != y)
        parent[y] = x;
}
```

範例

給定一個有圖含有一些集合，今每次用 AddEdge(x,y) 隨機將兩個點連接，試問每次在 AddEdge(x,y) 操作後此圖有幾個集合？
隨機將兩點連接可能會有以下三種情況：

1. 若兩點本來都沒別的邊，那代表集合數減一
2. 若一點有別的邊另一點沒有，那集合數減一
3. 若兩點都有別的邊，因為有可能他們其實屬於同個集合，因此要用 Find(x) 先去找他們的 parent，如果 parent 一樣代表為同一個集合，那麼集合數量不變。如果 parent 不一樣代表不同集合，就要使用 Union(x,y) 讓兩個點相連使集合的數量減一。



Reduction 轉化題目的關鍵思維

這章是筆者認為最重要的一章，主旨在於介紹如何有系統地將一個題目轉化為對應的演算法解法。這也正是軟體工程師在遇到問題時一個很重要的能力：思考與建模的能力。
許多人在刷題時，可能對各種演算法已有一定理解，譬如會寫 Binary Search、Dynamic Programming 等經典題目，但一旦遇到實際的問題，卻常常卡在「不知道該用什麼作法」進而用複雜的設計讓 happy path 通過，而特殊的情境只能 case by case 處理。但若看到更強的軟體工程師解法後才驚覺：「原來可以這樣解！」這種落差，其實就來自於「無法辨識題目背後的演算法核心元素」。
每種演算法其實都有其關鍵結構或典型特徵。舉例來說，Binary Search 最重要的思考點是：能否將問題轉化為一個 true/false 的查詢？只要能觀察出這一點，幾乎就能順利導入 Binary Search。
換句話說，這章所講的，不只是技巧，而是一種「從問題中抽象出可套用解法的能力」。這也是多數軟體工程師背後最核心的考察——因為在實際工作中，當你在開發一個專案時，經常會遇到多種可能的解法，這時你需要懂得辨認、拆解問題的關鍵，才能知道該選用什麼樣的 API、設計模式、資料結構甚至整體架構來解決它。

因此，建議在閱讀這章時，可以先回想你對各種常見演算法的理解，嘗試自行歸納每種演算法背後的「關鍵元素」。下方列出常見演算法與其典型特徵。

- Recursion
    - 使用recursion時，最關鍵的步驟有兩個：
        - 明確定義遞迴的輸入（input）與輸出（output）
        - 假設recursion本身的結果是正確的（也就是相信它會正確處理較小或較簡單的子問題），並藉由這個結果來推導當前 input 的 output

- Binary search
  - 使用 Binary search 時，最關鍵的步驟為找出搜尋的 True/False array，假設我們有一個是非題，其條件與某個數字 x 有關，而我們知道這個問題的答案會呈現出「單調性」——也就是：
    - 當 $x \le k$ 時，答案為 True
    - 當 $x > k$ 時，答案為 False

    或是相反的情況：
    - 當 $x \le k$ 時，答案為 False
    - 當 $x > k$ 時，答案為 True

    這代表整個答案序列會呈現出類似「前段全為 True，後段全為 False」或「前段全為 False，後段全為 True」的結構。
  - 常見的變化
    - 問 k 次操作後最大（最小）值為何 → 問能用 <=k 次操作達成的最大(小)值。
    - 該問題的是非題為，問我們要達成某一個值時，所需操作數是否 <=k
- Sliding window
  - Sliding Window 的核心策略是枚舉每個可能的 **right** 邊界，並在此基礎上向左尋找符合條件的 left 邊界，形成一個合法的 subarray
  - 這種技巧特別適用於題目條件具備「單調性」的情況，也就是滿足下列兩個特性之一：
    - 當一個 subarray 符合該條件時，若移除頭部或尾部的元素，仍會繼續滿足條件。
    - 當一個 subarray 不符合該題目條件時，無論再加入多少元素於頭尾，都無法變成合法的 subarray。
  - 使用Sliding window時，應從題目觀察及思考出以下四個部分：
    - 明確定義限制條件
      - 決定何時需要縮小 subarray，也就是何時該移動 left 邊界。
    - 確認需要維護的資訊
      - 在整個 sliding 過程中，必須維持哪些資訊（例如總和、頻率、最大值等）來判斷 subarray 是否有效及計算答案。
    - 新增元素時的更新策略
      - 新增一個元素進入 subarray 時，應該如何更新你所維護的資訊
    - 移除元素時的更新策略
      - 從 subarray 中移除一個元素時，應該如何正確地調整維護的資訊。
- Dynamic programming
  - 設計Dynamic programming解法時，最核心的兩個要素的是state 和transition
    - state
      - 狀態通常用一個array (或multiple dimension array)表示，例如 dp[i][j]。其中 i 和 j 代表影響答案的子問題的參數。
      - 如果不確定該如何正確定義狀態，可以觀察題目中有哪些「正整數條件」或「限制條件」，那些常常就是你的 state 維度。
    - transition
      - 假設較小的子問題（其他 state）都已經有解，思考該如何利用這些結果來計算出當前的 dp[i][j]
- Graph
  - 設計Graph解法時，最重要的是理解Graph的基本組成：node 和 edge

        - node：代表問題中的個別元素或狀態
        - edge：代表節點與節點之間的關係、互動或轉換條件
    - 建 graph 的關鍵：如何將題目的元素轉換為 graph 的 node 與 edge
        - 分析題目中哪些部分可以視為 node，哪些關係可以視為 edge。
        - 根據題意決定是使用 Directed Graph 或 Undirected Graph，邊是否含有Weight等。
- Backtracking
    - 在設計 Backtracking 解法時，最重要的是 Try and Error 及 設計一組具唯一性、明確的答案填寫順序
        - Try and Error
            - 首先要釐清題目的限制條件，判斷在什麼情況下當前的選擇會違反這些條件。
            - 接著設計如何嘗試一個可能的選項、將其加入目前的解（答案集合），以及在不符合條件或探索完該路徑後，如何還原狀態以嘗試其他可能性（這就是「回溯」的部分）。
        - 設計一個具唯一性且明確的解答填寫順序
            - 思考一組正確答案的形成順序：從哪裡開始填？每一步有幾種選擇？按照什麼順序嘗試？
            - 明確的順序設計能幫助減少重複計算與不必要的嘗試，使程式更有效率。
- Preflix sum
    - 能夠使用 Prefix Sum 解法的關鍵觀察在於：一個區間的答案可以透過「整體（母集合）」與「部分（子集合）」之間的快速相減來取得，以下為幾個常見範例。
        - 區間內的元素總和 / 數量：
            - 這類問題非常適合使用 Prefix Sum，因為可以透過 prefix[end] - prefix[start] 的方式，快速取得某一段範圍內的總和或數量，時間複雜度為 O(1)。
        - 區間內的最大值 / 最小值：
            - 這類型就不適合使用 Prefix Sum，因為最大值/最小值無法透過減法逆推出來。我們無法僅靠 max(母集合) - max(子集合) 推出正確答案，而是需要重新掃描區間中的元素。
        - 總結來說，Prefix Sum 適用於「可疊加」且滿足減法關係的區間性質，例如：和、數量、位移等。
- Sorting
    - 使用 比較式排序 時，請優先思考「我能否比較出誰大誰小？」
    - 使用 非比較式排序 時，請觀察「數值範圍是否有限？是否可以直接統計？」
- Data structure
    - 以下將介紹如何知道各個資料結構使用時機
        - Link List
            - 快速支援以下操作
                - 刪除某個元素
                - 在某個元素後加入一個元素
                - 找到某個元素的下一個/前一個元素
                - 頭尾元素為何
            - 不支援以下操作
                - random access， 也就是快速的知道第 i 個資料為什麼時。

- Stack
  - 快速支援以下操作
    - 加入元素
    - access 或移除上一個未被移除且最晚加入的資料
- Queue
  - 快速支援以下操作
    - 加入元素
    - access 或移除上一個未被移除且最早加入的資料
- Hash Table
  - 快速支援以下操作
    - 知道某個 key 存在與否，及存取該key的 value
- Priority queue
  - 快速支援以下操作
    - 加入元素
    - access 或移除最大(小)元素
  - 不支援以下操作
    - 同時access最大及最小元素
- Balanced binary search tree
  - 快速支援以下操作
    - 加入元素
    - 找到任一個元素
    - 若找不到某元素，則找出第一個比它大(小)的元素
  - 不支援以下操作
    - 無法比較大小的元素

More practice

以下提供相關練習題給大家練習，學習完對應課程後可到下列找相關主題更進一步練習

- Binary search
  - Leetcode 35 https://leetcode.com/problems/search-insert-position/description/
  - Leetcode 278 https://leetcode.com/problems/first-bad-version/description/
  - Leetcode 33 https://leetcode.com/problems/search-in-rotated-sorted-array/description/
  - (multitopic)Leetcode 2333 https://leetcode.com/problems/minimum-sum-of-squared-difference/description
- Sliding window
  - Leetcode 3 https://leetcode.com/problems/longest-substring-without-repeating-characters/description/
  - Leetcode 1358 https://leetcode.com/problems/number-of-substrings-containing-all-three-characters/description/
- Dynamic programming
  - Leetcode 64 https://leetcode.com/problems/minimum-path-sum/description/

- Backtracking
  - Leetcode 37 https://leetcode.com/problems/sudoku-solver/description/
- Prefix sum
- Sorting
  - Leetcode 23 https://leetcode.com/problems/merge-k-sorted-lists/description/
  - Leetcode 1636 https://leetcode.com/problems/sort-array-by-increasing-frequency/description/
- Linked list
  - Leetcode 21 https://leetcode.com/problems/merge-two-sorted-lists/description/
- Stack
  - Leetcode 20 https://leetcode.com/problems/valid-parentheses/
- Queue
  - Leetcode 210 https://leetcode.com/problems/course-schedule-ii/description/
- Hash table
  - Leetcode 1 https://leetcode.com/problems/two-sum/
- Graph - DFS
  - leetcode 100 https://leetcode.com/problems/same-tree/description/
  - leetcode 785 https://leetcode.com/problems/is-graph-bipartite/description/
  - leetcode 207 https://leetcode.com/problems/course-schedule/description/
    - follow up, a list of solution,need BFS
  - leetcode 226 https://leetcode.com/problems/invert-binary-tree/description/
- Graph - BFS
  - leetcode 2290 https://leetcode.com/problems/minimum-obstacle-removal-to-reach-corner/description/
    - follow up, some wall need two unit cost to remove
  - leetcode 2258 https://leetcode.com/problems/escape-the-spreading-fire/description/
- Graph - Disjoint sets
  - Leetcode 695 https://leetcode.com/problems/number-of-islands/description/
  - Leetcode 1579 https://leetcode.com/problems/remove-max-number-of-edges-to-keep-graph-fully-traversable/description/
- Priority queue
  - Leetcode 215 https://leetcode.com/problems/kth-largest-element-in-an-array/description/
  - Leetcode 1834 https://leetcode.com/problems/single-threaded-cpu/description/
- Trie
  - Leetcode 1268 https://leetcode.com/problems/search-suggestions-system/description/