基礎資料結構與演算法

TYIC 桃高資訊社

資料結構與演算法

資料結構與演算法

(Data Structure & Algorithm，簡稱 DSA)

在程式設計中有著非常重要的地位

使用好的資料結構和演算法

可能會使程式的執行速度變得更快

而使用不妥當的資料結構和演算法

則可能會使程式的執行速度變得緩慢

O (大 O 符號)

$O(x)$ 是用來表示一個函數趨近的上界，其定義為：

若有一個足夠大的正實數 x_0 和兩個 x 的函數 $f(x)$ 和 $g(x)$

且 $(\forall x \geq x_0)\ (\exists M > 0)\ (|f(x)| \leq M|g(x)|)$

則 $\lim_{x \to \infty} f(x) = \lim_{x \to \infty} O(g(x))$

又常將 $\lim_{x \to \infty}$ 省略，故簡寫為 $f(x) = O(g(x))$

舉例：若有一演算法，資料量對時間函數為 $T(n) = 3n^2 + 10n + 2$

可取 $M = 15$，則 $T(n) = O(n^2)$

$f(x) = O(g(x))$ 的可能最小 $g(x)$ 判別法：$g(x)$ 為 $f(x)$ 只保留主導項(增長最快的項)，並將係數變為 1

如 $T(n) = 3n^2 + 10n + 2$，只保留主導項 $3n^2$，並將係數變為 1，即 $3n^2$ 變為 n^2，故 $T(n) = O(n^2)$

Ω (大 Ω 符號)

與 $O(x)$ 相似，$\Omega(x)$ 是用來表示一個函數趨近的下界，其定義為：

若有一個足夠大的正實數 x_0 和兩個 x 的函數 $f(x)$ 和 $g(x)$

且 $(\forall x \geq x_0)\ (\exists M > 0)\ (|f(x)| \geq M|g(x)|)$

則 $\lim_{x \to \infty} f(x) = \lim_{x \to \infty} \Omega(g(x))$

又常將 $\lim_{x \to \infty}$ 省略，故簡寫為 $f(x) = \Omega(g(x))$

舉例：若有一演算法，資料量對時間函數為 $T(n) = 3n^2 + 10n + 2$

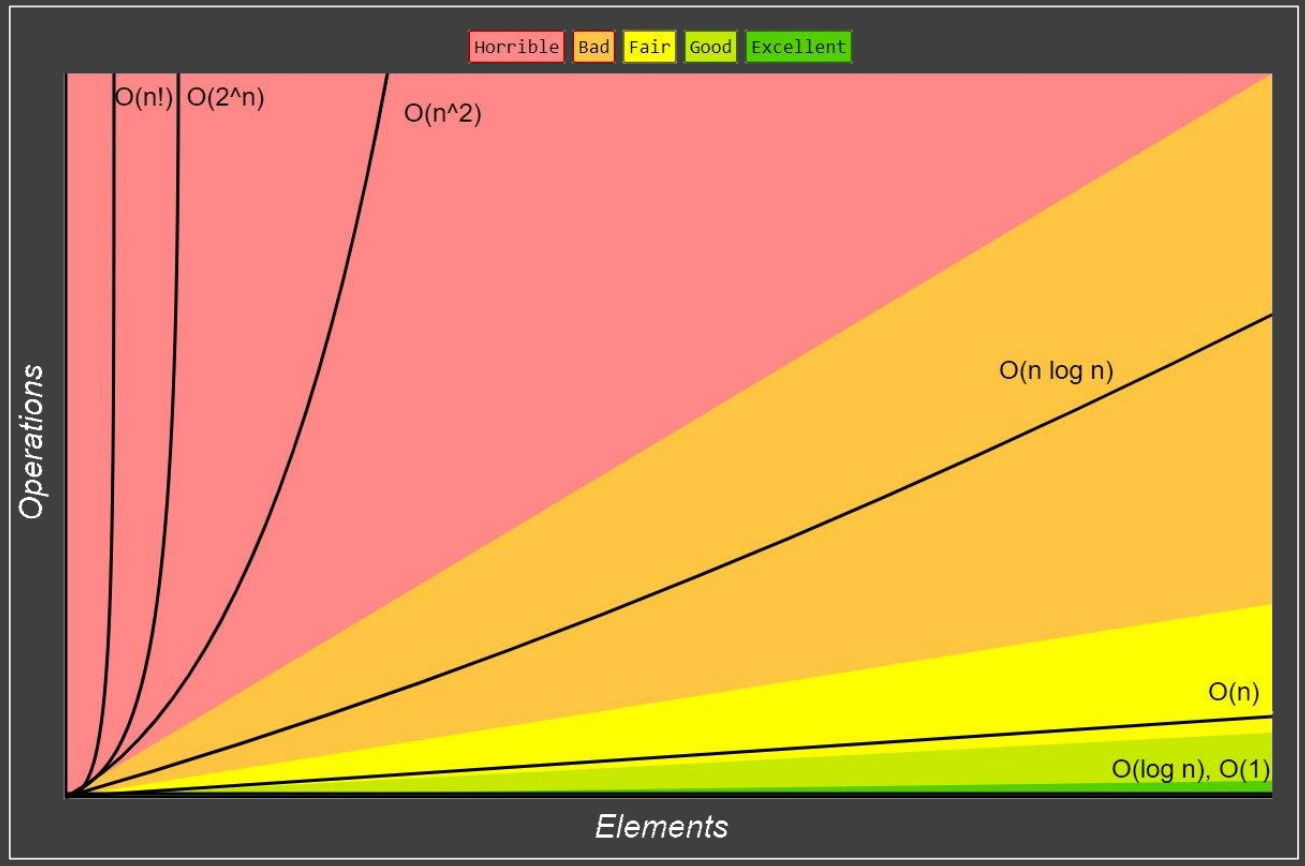
可取 $M = 3$，則 $T(n) = \Omega(n^2)$

Θ (大 Θ 符號)

$\Theta(x)$ 是在函數的趨近上下界相等時
用來表示函數的趨近界線
即若 $f(x) = O(g(x)) = \Omega(g(x))$，則 $f(x) = \Theta(g(x))$
舉例：若有一演算法，資料量對時間函數為 $T(n) = 3n^2 + 10n + 2$
已知 $T(n) = O(n^2) = \Omega(n^2)$，則 $T(n) = \Theta(n^2)$

時間複雜度

時間複雜度(time complexity)
是用於描述某一演算法
資料量與執行時間的關係
常用大 O 符號或大 Θ 符號來表示
常見的時間複雜度有：
常數時間 $O(1)$、對數時間 $O(\log n)$
線性時間 $O(n)$、二次時間 $O(n^2)$
線性對數時間 $O(n \log n)$
需注意，因為 n 不可能趨近無限大
故各時間複雜度大小之排序
不一定為真實執行時間多寡之排序



空間複雜度

空間複雜度(space complexity)
是用於描述某一演算法資料量與所需儲存空間的關係
常用大 O 符號或大 Θ 符號來表示
常見的空間複雜度有：$O(1)$、$O(\log n)$、$O(n)$、$O(n^2)$、$O(n \log n)$
同樣的，因為 n 不可能趨近無限大
故各空間複雜度大小之排序不一定為真實使用空間多寡之排序

通常在研究演算法時，時間複雜度的重要性會遠大於空間複雜度
並且同一演算法的空間複雜度也可能因不同實現方式而異

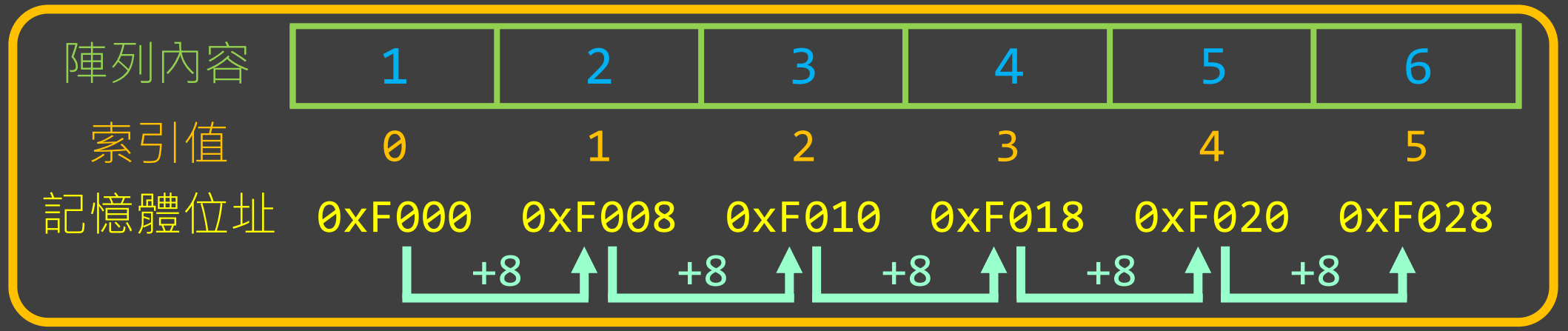
陣列

考慮儲存 5 筆資料，可以宣告 5 個變數來儲存
但考慮儲存 100 筆資料，宣告 100 個變數顯然不太現實
此時便可以使用陣列(array)來儲存多個相同型別的資料
陣列裡每個儲存的資料稱為元素(element)
陣列的長度(大小)在創建後不可變更

陣列存取的時間複雜度為 $O(1)$，原因後方會陳述
搜尋的時間複雜度為 $O(n)$，原因是因為需要循序搜尋
插入和刪除的時間複雜度為 $O(n)$
原因是因為需要將插入或刪除元素後方的元素向後或向前移動

陣列

陣列在記憶體中是連續的，即陣列中每個元素的記憶體位址皆相鄰
所以若要存取陣列中的元素，只需對記憶體位址進行簡單運算即可：
已知陣列元素記憶體大小為 s、索引值 0 元素之記憶體位址為 a
則陣列索引值 n 元素之記憶體位址即為 $a + s \times n$
故陣列存取的時間複雜度為 $O(1)$，這稱為隨機存取(random access)



串列與鏈結串列

串列(list)是一種資料結構，可以像陣列一樣儲存元素
但是儲存容量可以動態增長，也就是說元素的數量可以不固定

鏈結串列(linked list)是一種資料結構
由若干個節點(node)組成，每個節點記錄了一筆資料
鏈結是指使用指標來紀錄上一個和下一個節點的記憶體位址
只紀錄上一個或下一個元素的串列
稱為「單向鏈結串列(singly linked list)」
同時紀錄上一個和下一個元素的串列
稱為「雙向鏈結串列(doubly linked list)」

鏈結串列

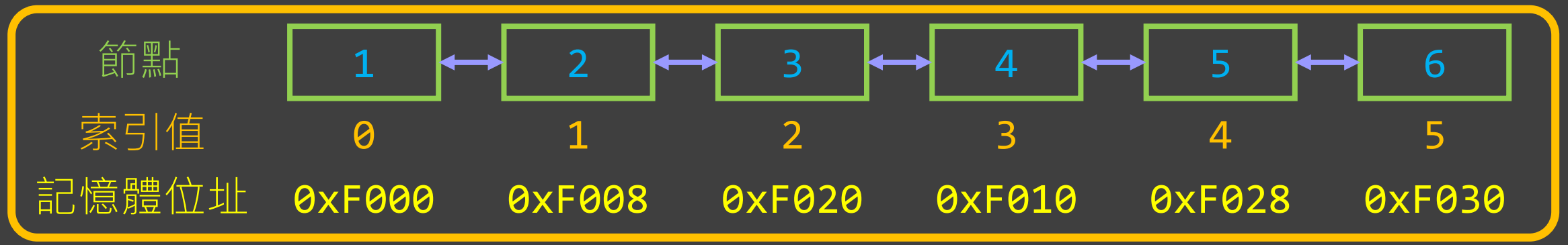
鏈結串列存取的時間複雜度為 $O(n)$

因為必須依次讀取每個節點，這稱為順序存取(sequential access)

搜尋的時間複雜度為 $O(n)$，原因是因為需要循序搜尋

插入和刪除的時間複雜度為 $O(1)$

原因是因為只需要將原有鏈結斷開，並重新連接上新的節點即可



陣列與鏈結串列

下表為陣列與鏈結串列對於四種基本操作的時間複雜度

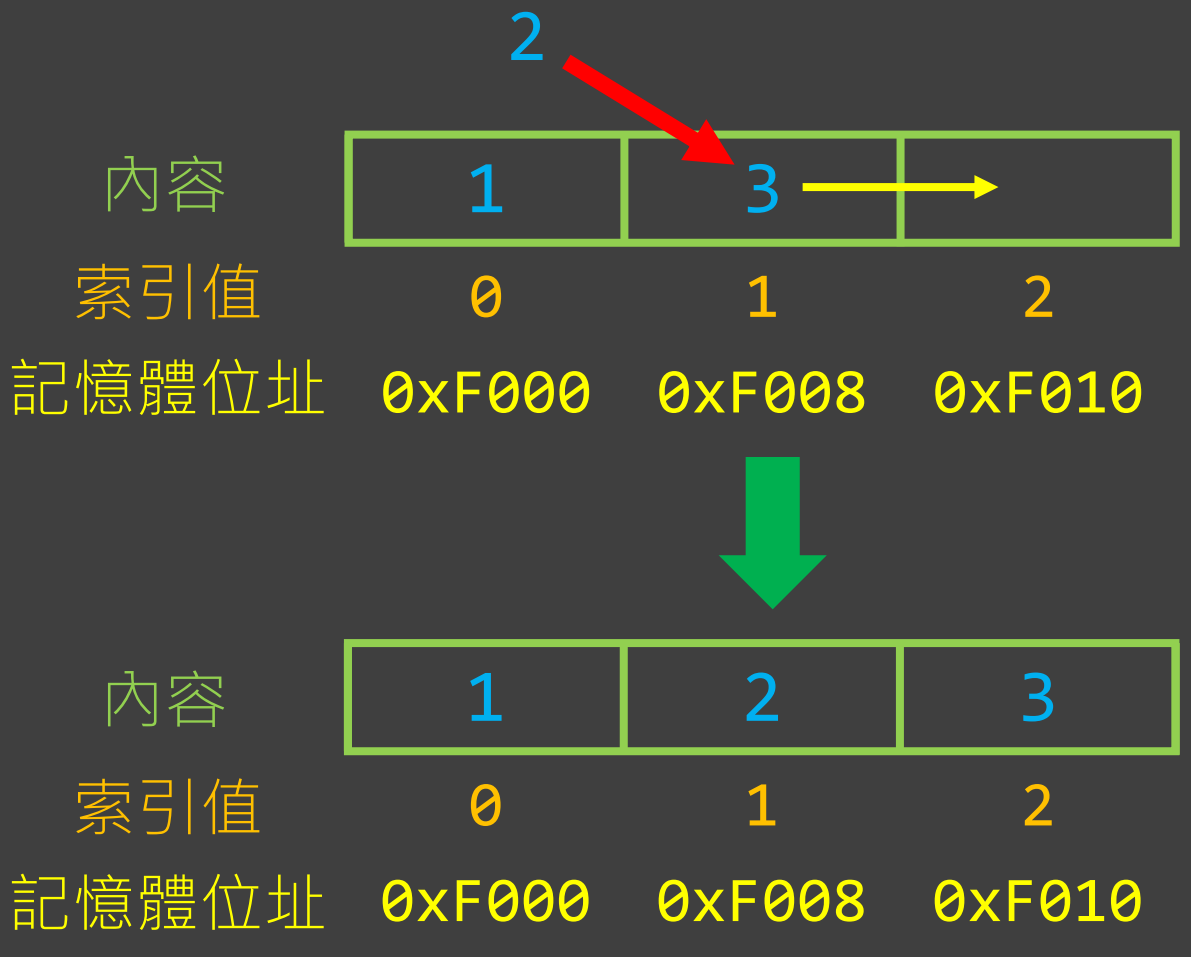
陣列的優勢在存取，而鏈結串列的優勢在插入和刪除

資料結構	陣列	鏈結串列
存取	$O(1)$	$O(n)$
搜尋	$O(n)$	$O(n)$
插入	$O(n)$	$O(1)$
刪除	$O(n)$	$O(1)$

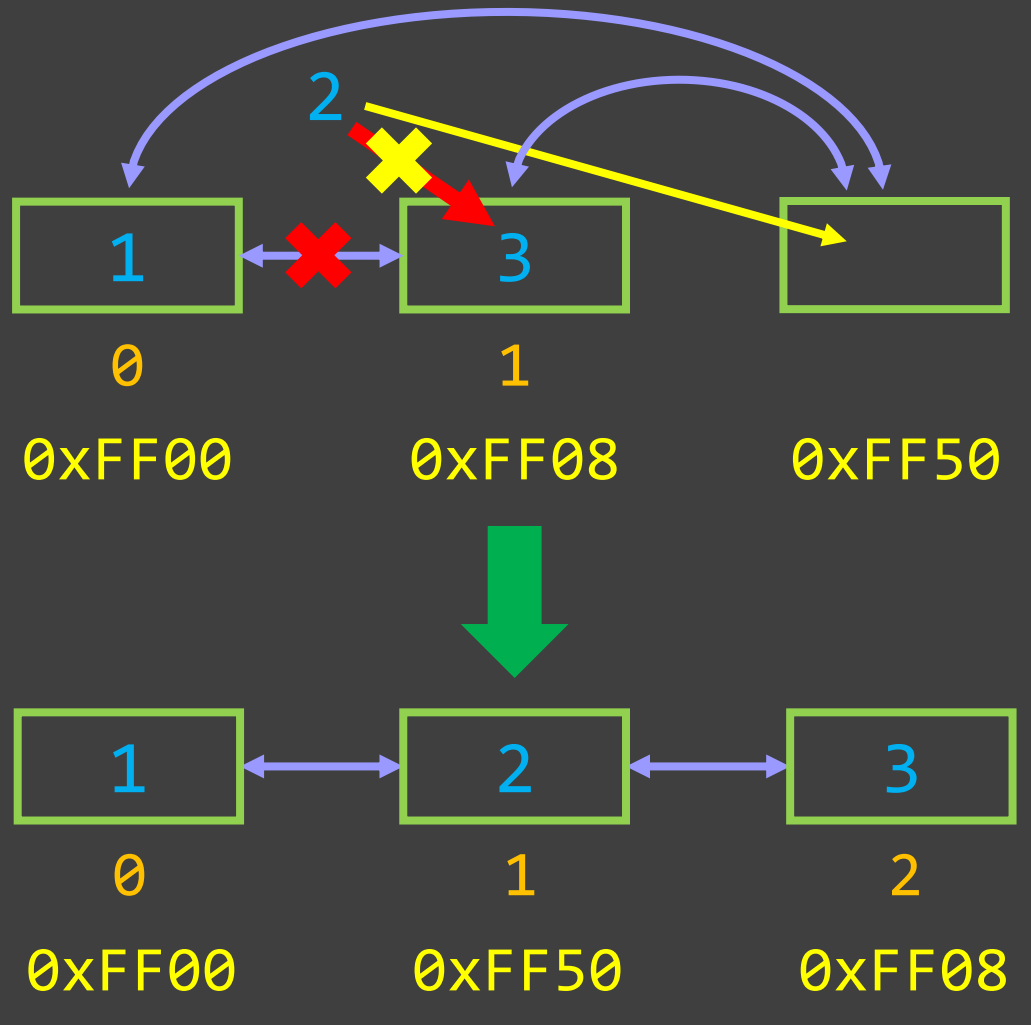
陣列與鏈結串列

將 2 插入 1 和 3 之間

陣列



鏈結串列



佇列、堆疊、雙端佇列

佇列(queue)是一種資料結構
元素從尾部(tail)進入並從頭部(head)出來
為先進先出(first-in-first-out，簡稱 FIFO)

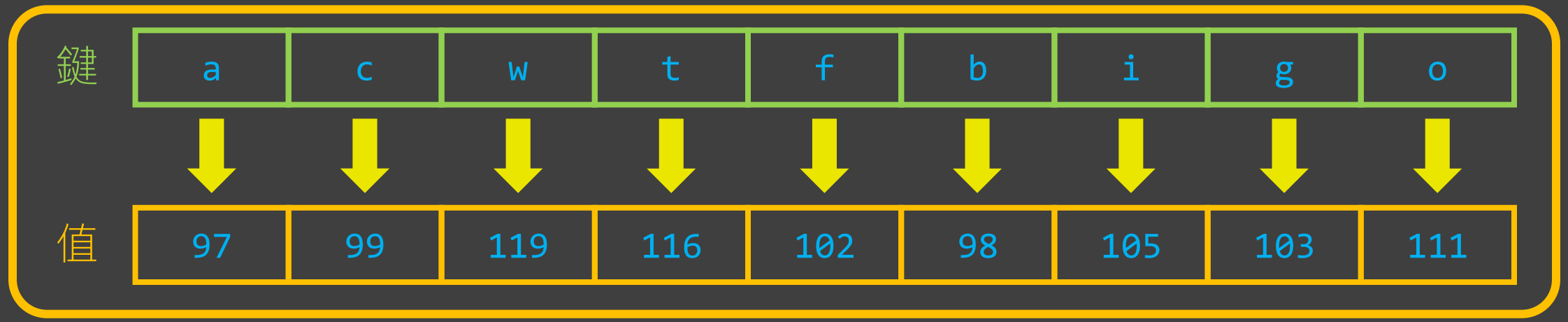
堆疊(stack)與佇列類似，但元素只能從頭部進出
為後進先出(last-in-first-out，簡稱 LIFO)
元素進入稱為推入(push)、出來稱為彈出(pop)

雙端佇列(deque)是同時具有佇列和堆疊性質的資料結構
元素可以從頭部或尾部進出

佇列、堆疊、雙端佇列

鍵值映射

鍵值映射(key-value mapping)是一種資料結構
代表一些相同型別且不重複的鍵(key)
各自映射到一個相同型別的值(value)
每組鍵和值稱為鍵值對(key-value pair、map entry)



迴圈找最大、最小值

對於多個值，想要找尋最大值、最小值

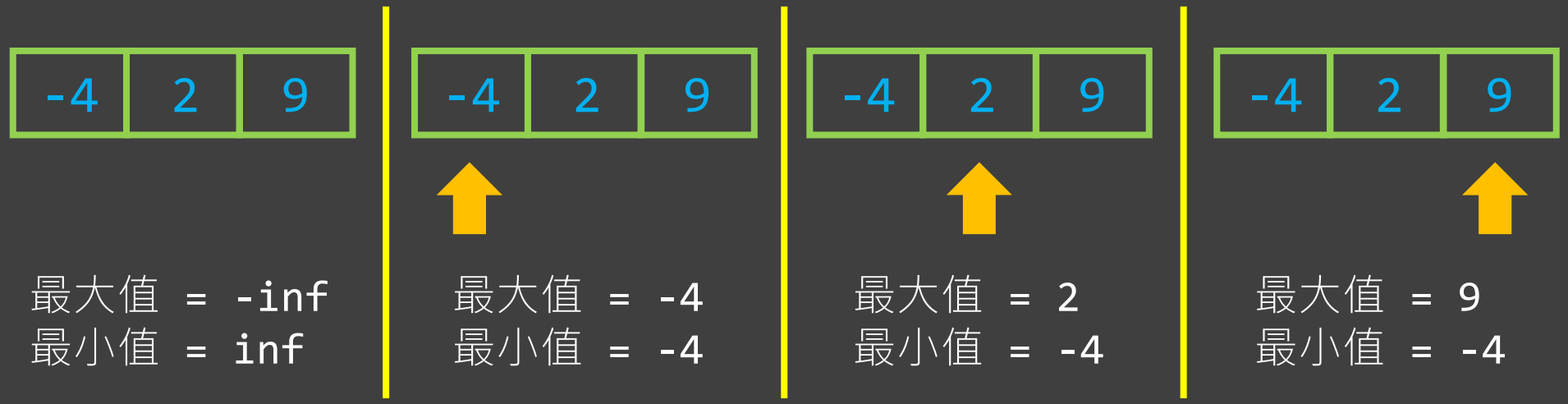
除了對資料排序外，也可利用以下方法，時間複雜度為 $O(n)$：

依序讀取每個值，若較當前的最大值大或最小值小

則將最大值或最小值變為該值

特別注意，最大值須初始化成比所有可能值小的數

最小值須初始化成比所有可能值大的數



迴圈找最大、最小值

```java
import java.util.Scanner;

public class Main1 {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        int n = scanner.nextInt(); // 獲取資料個數
        int max = -2147483648, min = 2147483647;
        for (int i = 0; i < n; i++) {
            int p = scanner.nextInt(); // 讀入資料
            if (p > max) max = p;
            if (p < min) min = p;
        }
        System.out.printf("max = %d, min = %d", max, min);
    }
}
```

java

```
10
-1 5 -9 8 1000 2 -1999 2 0 1
max = 1000, min = -1999
```

console

循序搜尋法

循序搜尋法(線性搜尋法，linear search)是一種常見的搜尋法

其為依序比對每一個資料直到找到正確的資料，時間複雜度為 $O(n)$

```java
import java.util.Scanner;

public class Main2 {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        int n = scanner.nextInt(); // 獲取資料個數
        int[] arr = new int[n];
        for (int i = 0; i < n; i++) arr[i] = scanner.nextInt(); // 讀入資料
        int target = scanner.nextInt(); // 讀入目標資料
        // 循序搜尋法
        for (int i = 0; i < n; i++) {
            if (arr[i] == target) {
                System.out.println(i);
                return;
            }
        }
        System.out.println("Not found.");
    }
}
```

java

```
10
-2 5 9 10 22 33 44 89 101 777
101
8
```

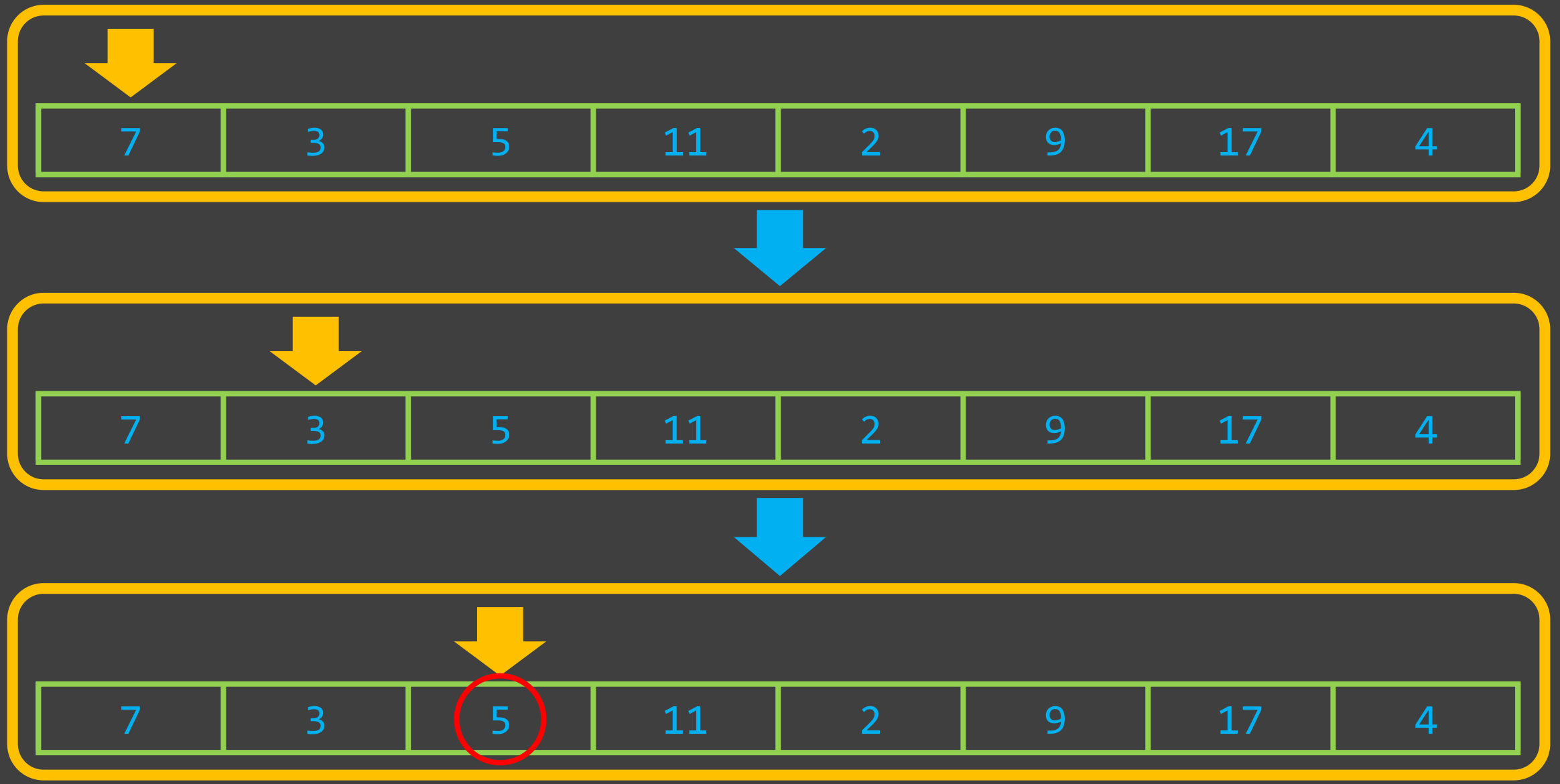
console

```
10
-2 5 9 10 22 33 44 89 101 777
102
Not found.
```

console

循序搜尋法

找 5



循序搜尋法的衍伸應用

循序搜尋法在資料已排序時
也可以用來找
大於目標的最小索引值
即若要將目標插入資料時
要插入到的索引值

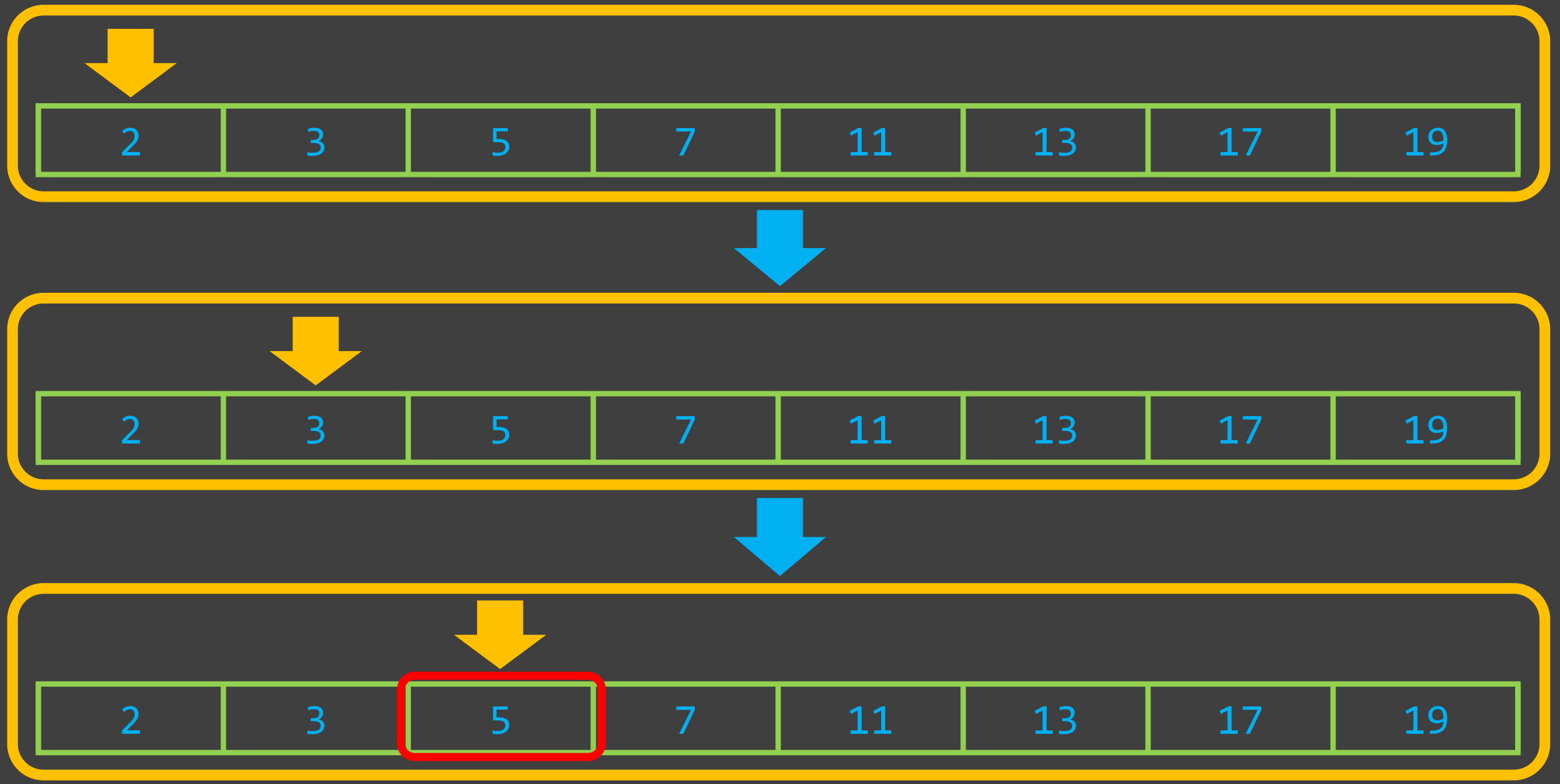
```
10
-2 5 9 10 22 33 44 89 101 777
101
8                          console
```

```
10
-2 5 9 10 22 33 44 89 101 777
102
Not found.                 console
```

```java
import java.util.Scanner;

public class Main3 {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        int n = scanner.nextInt(); // 獲取資料個數
        int[] arr = new int[n];
        for (int i = 0; i < n; i++)
            arr[i] = scanner.nextInt(); // 讀入資料
        int target = scanner.nextInt(); // 讀入目標資料
        // 循序搜尋法
        for (int i = 0; i < n; i++) {
            if (arr[i] > target) {
                System.out.println(i);
                return;
            }
        }
        System.out.println(n);
    }
}
```

循序搜尋法的衍生應用

找 4



二分搜尋法

二分搜尋法(binary search)是一種常見的搜尋法

在使用二分搜尋法前須將資料由小到大排序

其原理為：每次只會搜尋可能區間中間的資料

若在搜尋到較目標大的資料時，下次搜尋只會搜尋較小的資料

若在搜尋到較目標小的資料時，下次搜尋只會搜尋較大的資料

重複直到搜尋到目標資料，或是可能區間無效，即找不到目標資料

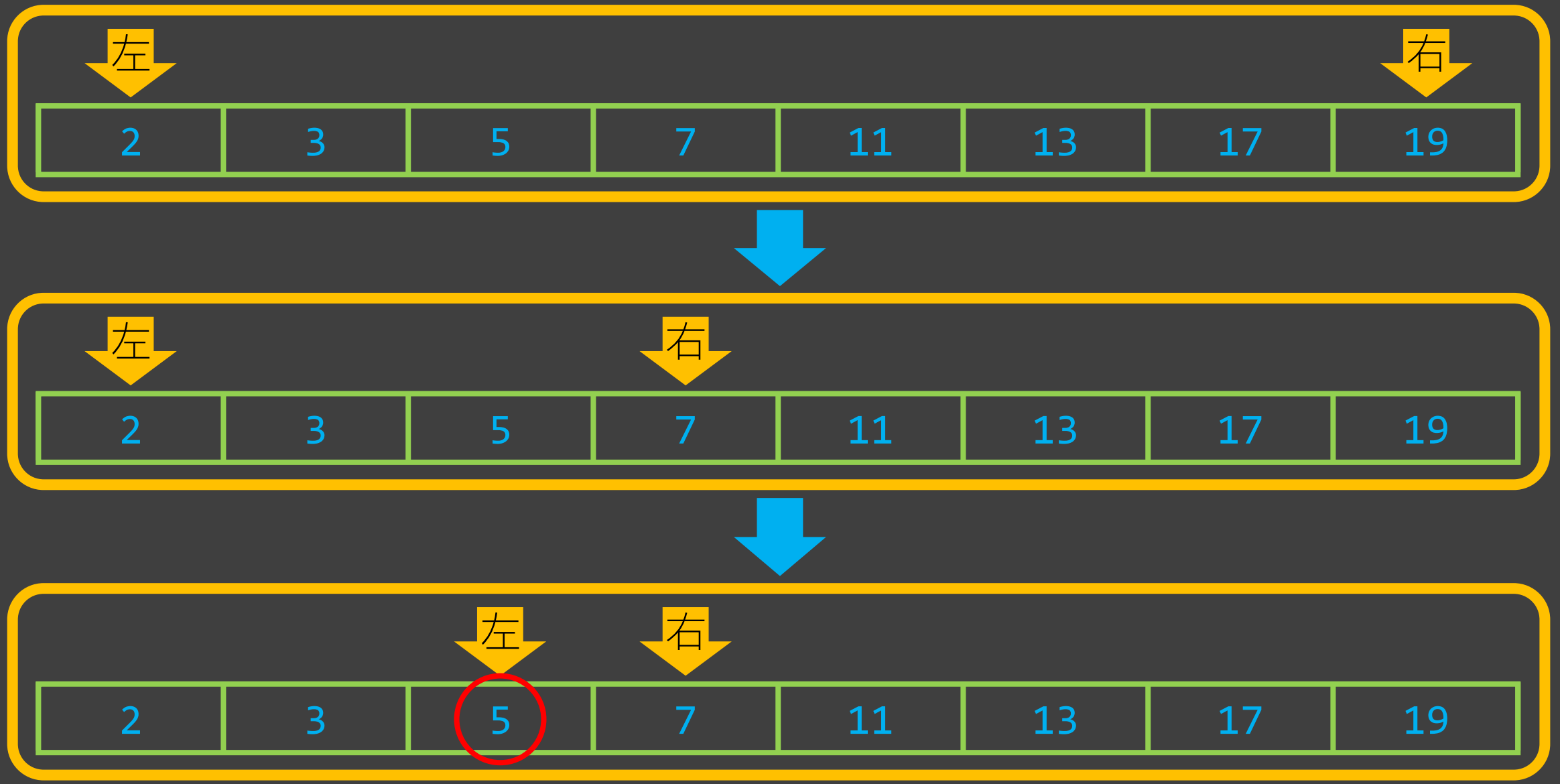
程式上實現，會使用兩個變數來紀錄可能區間的左邊界與右邊界

由於二分搜尋法一次就可以排除一半的可能

故時間複雜度為 $O(\log n)$，效率較循序搜尋法較高

二分搜尋法

找 5



二分搜尋法

```java
import java.util.Scanner;

public class Main4 {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        int n = scanner.nextInt(); // 獲取資料個數
        int[] arr = new int[n];
        for (int i = 0; i < n; i++)
            arr[i] = scanner.nextInt(); // 讀入資料
        int target = scanner.nextInt(); // 讀入目標資料
        // 二分搜尋
        int l = 0; // 左邊界，目標的最小可能索引值
        int r = n - 1; // 右邊界，目標的最大可能索引值
        while (l <= r) {
            int mid = l - (l - r) / 2; // 取中間的資料
            if (arr[mid] == target) {
                System.out.println("Target Index: " + mid);
                return;
            }
            if (arr[mid] > target) r = mid - 1;
            else l = mid + 1;
        }
        System.out.println("Target Not found.");
    }
}
```

java

```
10
-2 5 9 10 22 33 44 89 101 777
101
Target Index: 8
```

console

```
10
-2 5 9 10 22 33 44 89 101 777
102
Target Not found.
```

console

二分搜尋法的衍伸應用

二分搜尋法在找不到目標時
可以找大於目標的最小索引值
也就是若要將目標插入資料時
要插入到的索引值

```
10
-2 5 9 10 22 33 44 89 101 777
102
Target Insert Index: 9
```
console

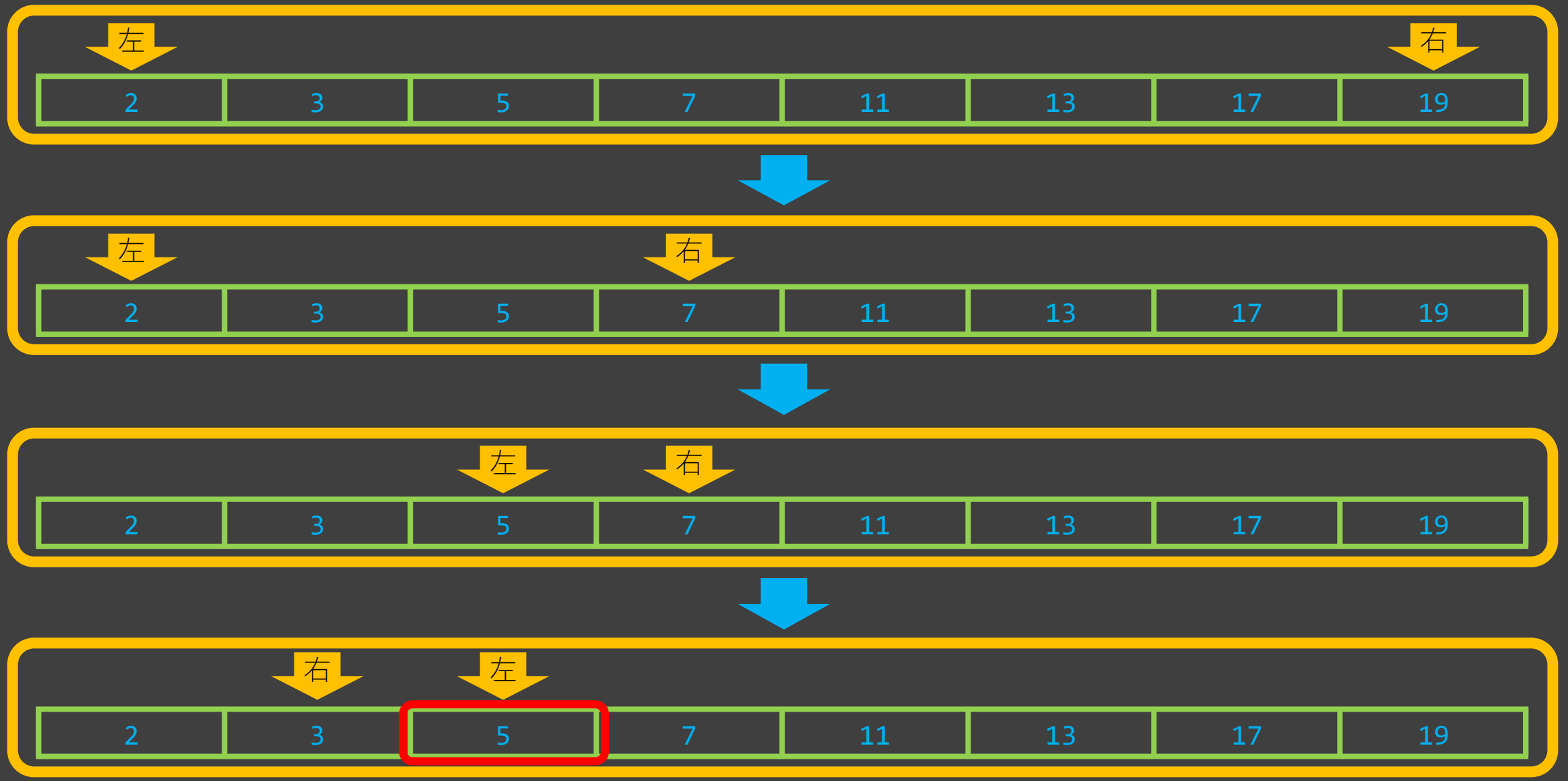
```
5
1 2 3 4 5
0
Target Insert Index: 0
```
console

```
5
6 7 8 9 10
100
Target Insert Index: 5
```
console

```java
import java.util.Scanner;

public class Main5 {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        int n = scanner.nextInt(); // 獲取資料個數
        int[] arr = new int[n];
        for (int i = 0; i < n; i++)
            arr[i] = scanner.nextInt(); // 讀入資料
        int target = scanner.nextInt(); // 讀入目標資料
        // 二分搜尋
        int l = 0; // 左邊界，目標的最小可能索引值
        int r = n - 1; // 右邊界，目標的最大可能索引值
        while (l <= r) {
            int mid = l - (l - r) / 2; // 取中間的資料
            if (arr[mid] == target) {
                System.out.println("Target Index: " + mid);
                return;
            }
            if (arr[mid] > target) r = mid - 1;
            else l = mid + 1;
        }
        System.out.println("Target Insert Index: " + l);
        // 資料應插入的索引值即為左邊界
    }
}
```
java

二分搜尋法的衍生應用

找 4



遞迴

遞迴(Recursion)是指函式自己呼叫自己，直到終止

所以遞迴可以用來處理可以拆分成許多相似小問題的大問題

如費波那契數列(Fibonacci sequence)

Google 搜尋「遞迴」

OEIS A000045 費波那契數列

第 0 項為 0，第 1 項為 1

之後的每一項皆是前兩項之和

以數學遞迴關係式表示就是

$$\begin{cases} F_0 = 0, F_1 = 1 \\ F_n = F_{n-1} + F_{n-2} \ (n \geq 2) \end{cases}$$

以 F_4 為例： $F_4 = F_3 + F_2$

$= F_2 + F_1 + F_1 + F_0$

$= F_1 + F_0 + F_1 + F_1 + F_0 = 3$

```java
import java.util.Scanner;

public class Main2 {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        int n = scanner.nextInt();
        System.out.println(fib(n));
    }

    static int fib(int n) {
        if (n < 2) return n;
        return fib(n - 1) + fib(n - 2);
    }
}
```

java

```
10
55
```

console

最大公因數

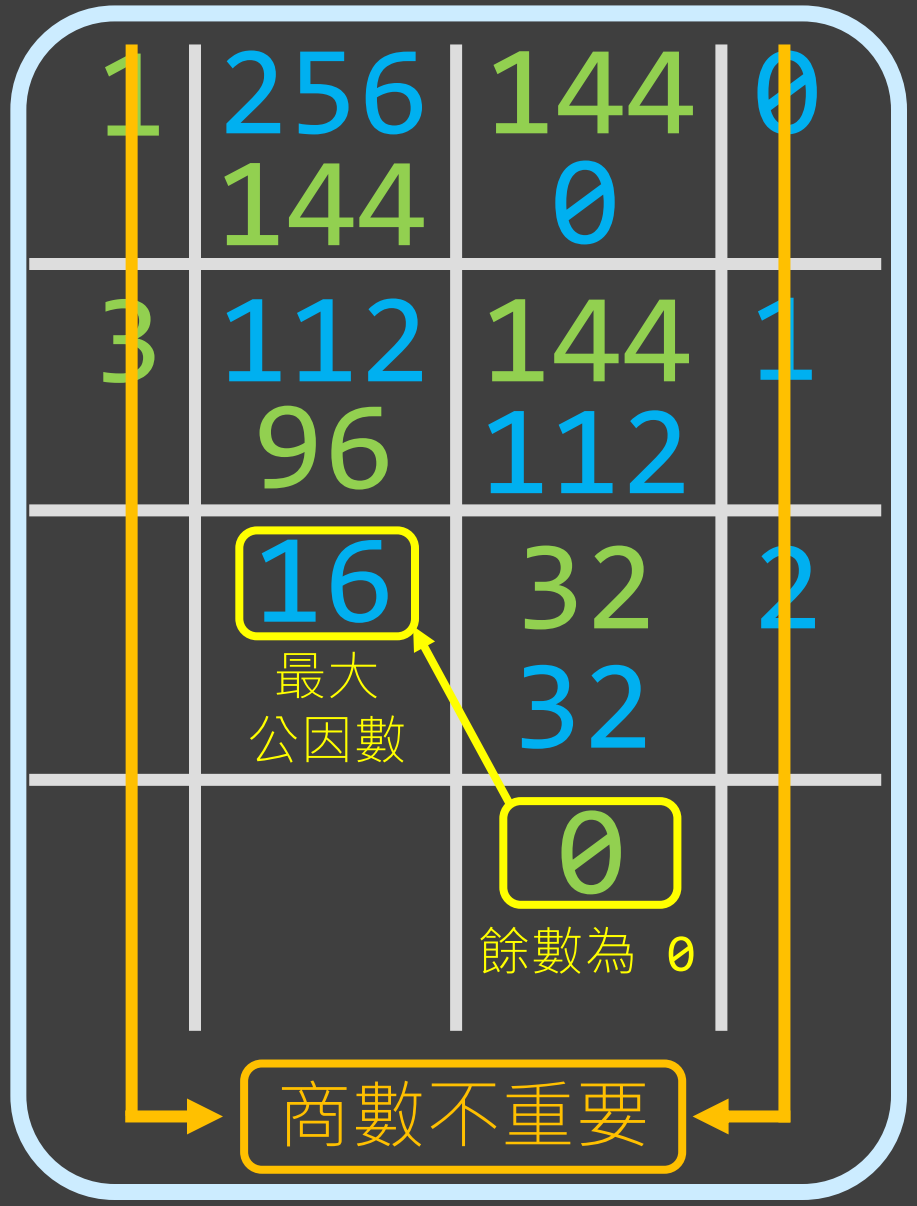
最大公因數(greatest common divisor，簡稱 gcd)

程式實現常使用程式碼簡潔的

輾轉相除法(歐幾里得算法，Euclidean algorithm)

其說明：若 $a = bq + r$，則 $\gcd(a, b) = \gcd(b, r)$

```java
static int gcd(int a, int b) {
    if (b == 0) return a;
    return gcd(b, a % b);
}

static int gcd(int a, int b, int c) {
    return gcd(gcd(a, b), c);
}
```

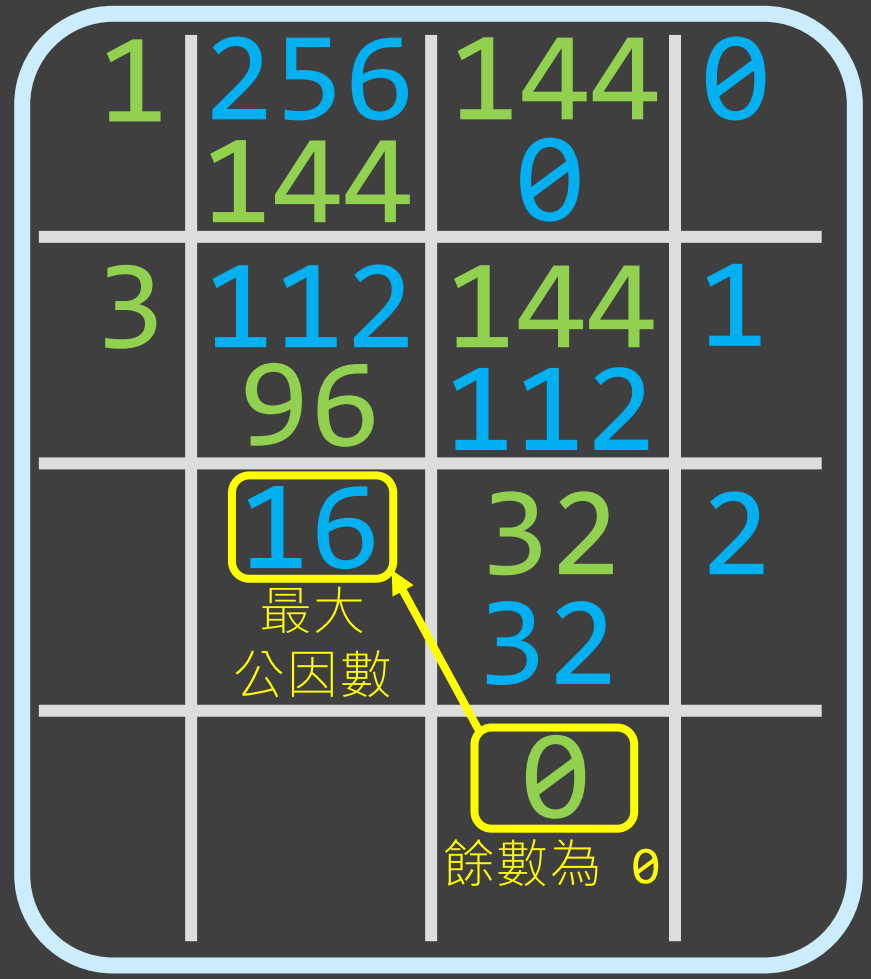


補充：輾轉相除法證明

1. 已知 $a = bq + r$ $(a , b , r \in \mathbb{N})$，設 $\gcd(a , b) = g$ $(g \in \mathbb{N})$
則 $a = mg , b = ng$ $(m , n \in \mathbb{N})$，且 $\gcd(m , n) = 1$
故 $r = a – bq = mg – nqg = g(m – nq)$ 必有因數 g

2. 設 $\gcd(b , r) = pg$ $(p \in \mathbb{N})$
則 $n = pu , (m – nq) = pv = m – puq$ $(u , v \in \mathbb{N})$
得 $m = pv + puq = p(v + uq)$ 必有因數 p
$\gcd(m , n) = p = 1$，故 $\gcd(b , r) = g$

3. 當 $r = 0$ 時 $a = bq$，則 $\gcd(a , b) = b = g$



最小公倍數

最小公倍數(least common multiple，簡稱 lcm)

程式實現常使用數學性質 $\text{lcm}(a, b) = \frac{|ab|}{gcd(a, b)}$

先求出最大公因數，再求出最小公倍數

```java
static int gcd(int a, int b) {
    if (b == 0) return a;
    return gcd(b, a % b);
}
```

```java
static int lcm(int a, int b) {
    return a * b / gcd(a, b);
}

static int lcm(int a, int b, int c) {
    return lcm(lcm(a, b), c);
}
```

獲取一正整數位數

若一正整數 n 滿足 $10^n \leq x = a \times 10^n < 10^{n+1}\ (1 \leq a < 10)$

則 $\log(10^n) = n \leq \log(x) = n + \log(a) < \log(10^{n+1}) = n + 1$

又 $0 \leq \log(a) < 1$，得 $[\log(x)] = n$ (註：$[m]$ 為下取整函數，如 $[2.7] = 2$)

又已知 10^n 為 $n + 1$ 位數，故 x 為 $n + 1 = [\log(x)] + 1$ 位數

```java
import java.util.Scanner;

public class Main1 {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        int n = scanner.nextInt();
        System.out.printf("%d has %d digit(s).", n, (int) Math.log10(n) + 1);
    }
}
```

java

```
900999
900999 has 6 digit(s).
```

console

```
123
123 has 3 digit(s).
```

console

獲取一正整數之每一位數

末位數字即為該正整數除以 10 的餘數

該正整數除以 10 的商即為去除末位數字後的其他位數字

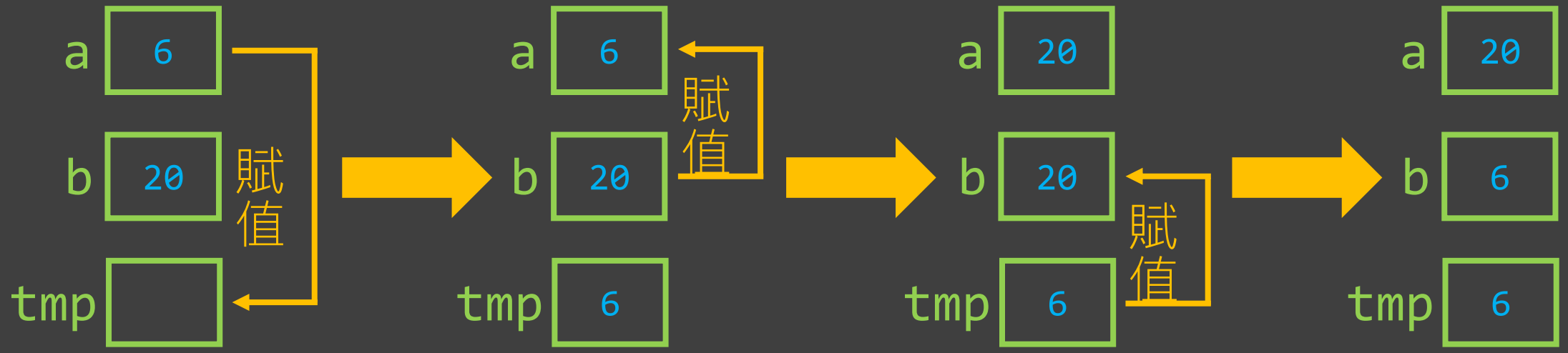
```java
import java.util.Scanner;

public class Main2 {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        int n = scanner.nextInt();
        while (n != 0) {
            System.out.println(n % 10);
            n /= 10;
        }
    }
}
```

```
12345
5
4
3
2
1
```
console

```
114514
4
1
5
4
1
1
```
console

交換

在許多演算法中，會需要將兩個變數 a、b 的資料交換(swap)
若直接將變數 b 賦值給變數 a，會導致變數 a 的資料遺失
故應先將變數 a 賦值給另一個變數 tmp，避免資料遺失
再將變數 b 賦值給變數 a，最後將變數 tmp 賦值給變數 b



交換

```java
import java.util.Scanner;

public class Main3 {
    public static void main(String[] args) {
        // 將資料讀入變數 a, b
        Scanner scanner = new Scanner(System.in);
        int a = scanner.nextInt(), b = scanner.nextInt();
        System.out.printf("BEFORE swap: a = %d, b = %d %n", a, b);
        // 交換資料
        int tmp = a;
        a = b;
        b = tmp;
        System.out.printf("AFTER swap: a = %d, b = %d %n", a, b);
    }
}
```

java

```
6 8
BEFORE swap: a = 6, b = 8
AFTER swap: a = 8, b = 6
```

console

排序

排序是一個非常常見、重要的問題

排序主要分為比較排序和非比較排序

常見的比較排序：

氣泡排序(bubble sort)、選擇排序(selection sort)

插入排序(insertion sort)、合併排序(merge sort)

快速排序(quick sort)、Tim 排序(Timsort)

常見的非比較排序：基數排序(radix sort)

因非比較排序對資料類型限制較多，故比較排序較常使用

因要讀取每筆資料，大部分排序的時間複雜度不可能小於 $O(n)$

排序

排序演算法還有穩定性(stability)的問題

穩定性是指對於相同的資料是否會改變他們之間的前後順序

穩定排序(stable sort)不會改變相同資料的順序

而不穩定排序(non-stable sort)則會

常見的穩定排序：

氣泡排序、選擇排序(插入)、插入排序、合併排序

快速排序(插入)、Tim 排序、基數排序

常見的不穩定排序：選擇排序(交換)、快速排序(交換)

選擇排序演算法時，通常會優先選擇穩定排序

氣泡排序法

氣泡排序法(bubble sort)是一種非常簡單的排序法

其原理為：

由左到右，依序比較兩個相鄰的資料，最後一個元素除外

若第一個資料比第二個資料大，便交換這兩個資料

重複 n - 1 次

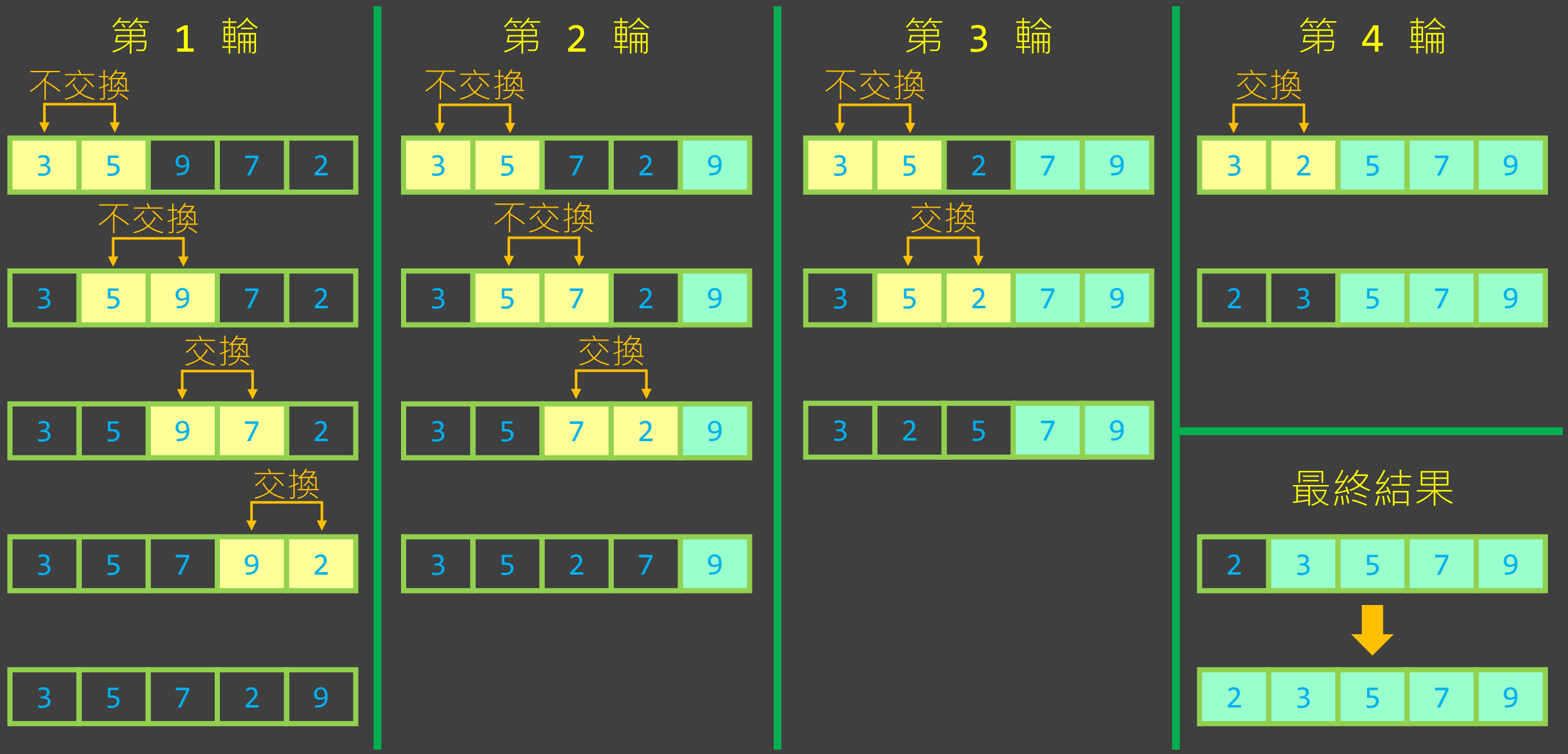
因每輪的最大值都會像氣泡一樣浮上來，被交換到最右邊

故最終就會將資料由小到大排序完成

總共須比較、交換資料 $n-1+n-2+\cdots+1=\frac{n^2-n}{2}$ 次

故氣泡排序法的時間複雜度為 $O(n^2)$

氣泡排序法


第 1 輪
不交換
3 5 9 7 2
不交換
3 5 9 7 2
交換
3 5 9 7 2
交換
3 5 7 9 2
3 5 7 2 9
第 2 輪
不交換
3 5 7 2 9
不交換
3 5 7 2 9
交換
3 5 7 2 9
3 5 2 7 9
第 3 輪
不交換
3 5 2 7 9
交換
3 5 2 7 9
3 2 5 7 9
第 4 輪
交換
3 2 5 7 9
2 3 5 7 9
最終結果
2 3 5 7 9
2 3 5 7 9

氣泡排序法

若整輪皆沒有資料交換
表示排序已完成
可提前結束排序

```
5
6 2 -1 0 7
[-1, 0, 2, 6, 7]
```
console

```java
import java.util.Arrays;
import java.util.Scanner;

public class Main1 {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        int n = scanner.nextInt(); // 獲取資料個數
        int[] arr = new int[n];
        for (int i = 0; i < n; i++)
            arr[i] = scanner.nextInt(); // 讀入資料
        // 氣泡排序
        for (int i = 0; i < n - 1; i++) {
            boolean flag = true; // 用於判斷是否要提早結束排序
            for (int j = 0; j < n - 1 - i; j++) {
                if (arr[j] > arr[j + 1]) {
                    // 交換兩個資料
                    int tmp = arr[j];
                    arr[j] = arr[j + 1];
                    arr[j + 1] = tmp;
                    flag = false; // 有資料交換，表示須繼續排序
                }
            }
            if (flag) break; // 該輪沒有資料交換，表示排序已提前完成
        }
        System.out.println(Arrays.toString(arr));
    }
}
```
java

選擇排序法

選擇排序法(selection sort)也是一種非常簡單的排序法

其原理為：將資料分為左邊的已排序資料及右邊的未排序資料

從右邊的未排序資料找出最小值

插入到左邊的已排序資料最末端(或與未排序資料的最左邊交換)

最終就會將資料由小到大排序完成

總共須比較資料 $n + n - 1 + \cdots + 2 = \frac{n^2 - n}{2}$ 次，時間複雜度為 $O(n^2)$

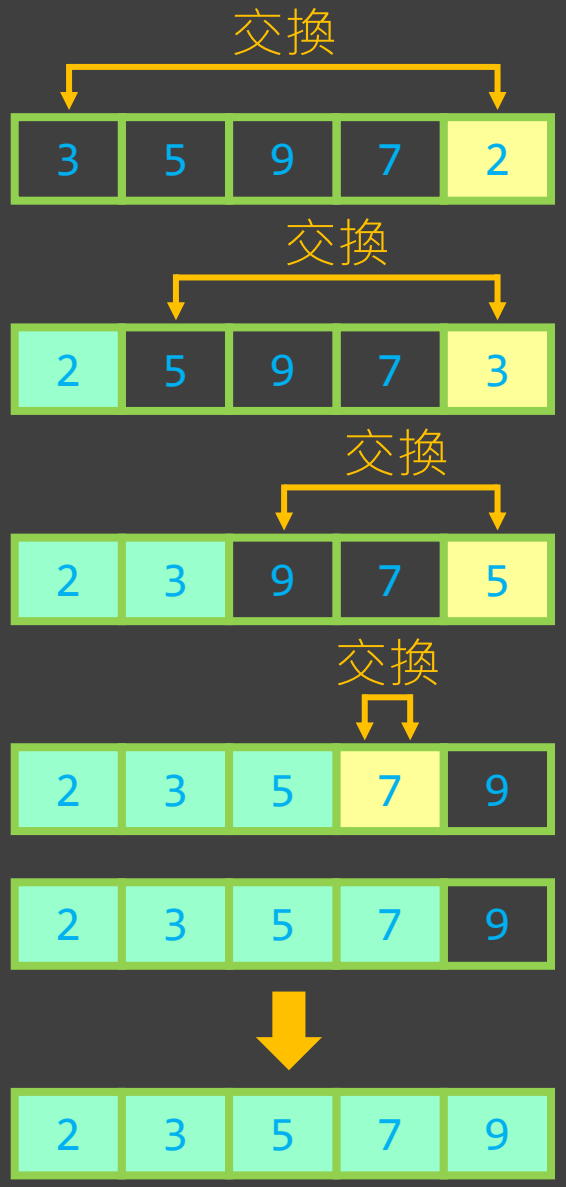
但只須交換(或插入)資料 $n - 1$ 次，時間複雜度為 $O(n)$

故整個選擇排序法的時間複雜度 $O(n^2)$

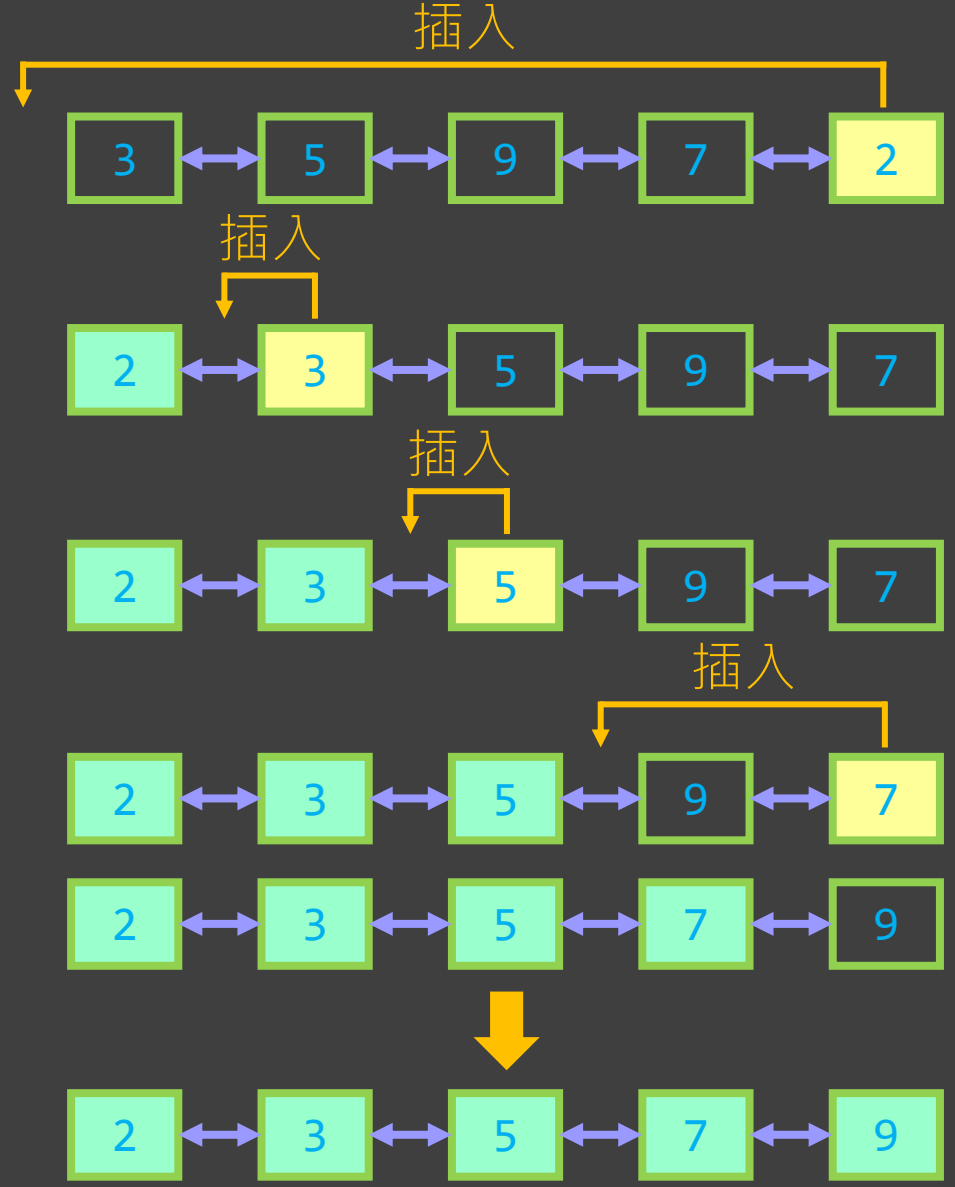
交換常用於陣列，而插入則常用於鏈結串列

選擇排序法

交換（不穩定排序）



插入（穩定排序）



選擇排序法(交換)

```java
import java.util.Arrays;
import java.util.Scanner;

public class Main3 {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        int n = scanner.nextInt(); // 獲取資料個數
        int[] arr = new int[n];
        for (int i = 0; i < n; i++) arr[i] = scanner.nextInt(); // 讀入資料
        // 選擇排序
        for (int i = 0; i < n - 1; i++) {
            int minIndex = i;
            for (int j = i; j < n; j++) if (arr[j] < arr[minIndex]) minIndex = j;
            int tmp = arr[i];
            arr[i] = arr[minIndex];
            arr[minIndex] = tmp;
        }
        System.out.println(Arrays.toString(arr));
    }
}
```

java

```
5
3 5 9 7 2
[2, 3, 5, 7, 9]
```

console

```
8
-1 8 -5 66 7 0 1 2
[-5, -1, 0, 1, 7, 8, 28, 66]
```

console

選擇排序法(插入)

```java
public class Main2 {
    // 定義鏈結串列
    private static class IntNode {
        public final int value;
        public IntNode next;
        public IntNode previous;

        public IntNode(int value) {
            this.value = value;
        }

        public static void linkTwoNodes(IntNode previous, IntNode next) {
            if (Objects.nonNull(previous)) previous.next = next;
            if (Objects.nonNull(next)) next.previous = previous;
        }
    }
```

```
5
3 5 9 7 2
2 3 5 7 9
```
console

```
8
-1 8 -5 66 7 0 1 2
-5 -1 0 1 2 7 8 66
```
console

```java
public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        int n = scanner.nextInt(); // 獲取資料個數
        // 建立鏈結串列
        IntNode firstNode = null, currentNode = null;
        for (int i = 0; i < n; i++) {
            IntNode newNode = new IntNode(scanner.nextInt());
            if (Objects.isNull(firstNode)) {
                currentNode = newNode;
                firstNode = newNode;
            } else {
                IntNode.linkTwoNodes(currentNode, newNode);
                currentNode = newNode;
            }
        }
        // 選擇排序
        IntNode sortLastNode = null;
        for (int i = 0; i < n - 1; i++) {
            IntNode minNode = null;
            currentNode = Objects.isNull(sortLastNode) ? firstNode : sortLastNode.next;
            for (int j = i; j < n; j++) {
                if (Objects.isNull(minNode) || currentNode.value < minNode.value)
                    minNode = currentNode;
                currentNode = currentNode.next;
            }
            IntNode.linkTwoNodes(minNode.previous, minNode.next);
            if (Objects.isNull(sortLastNode)) {
                IntNode.linkTwoNodes(minNode, firstNode);
                firstNode = minNode;
            } else {
                IntNode.linkTwoNodes(minNode, sortLastNode.next);
                IntNode.linkTwoNodes(sortLastNode, minNode);
            }
            sortLastNode = minNode;
        }
        // 輸出
        currentNode = firstNode;
        while (Objects.nonNull(currentNode)) {
            System.out.print(currentNode.value + " ");
            currentNode = currentNode.next;
        }
    }
}
```
java

插入排序法

插入排序法(insertion sort)也是一種非常簡單的排序法
其原理為：將資料分為左邊的已排序資料及右邊的未排序資料
依序將右邊的未排序資料，插入到左邊已排序資料的正確位置
最終就會將資料由小到大排序完成
插入排序法的時間複雜度為 $O(n^2)$
且為所有時間複雜度為 $O(n^2)$ 排序演算法中平均執行時間最短的
所以在資料量較少(通常為少於 64 個)
或是資料已部分排序的情況下很常被使用

插入排序法(陣列)

插入排序法對於陣列

每次將未排序資料插入已排序資料

須由末端依序將已排序資料向後移

直到找到小於欲插入資料的資料

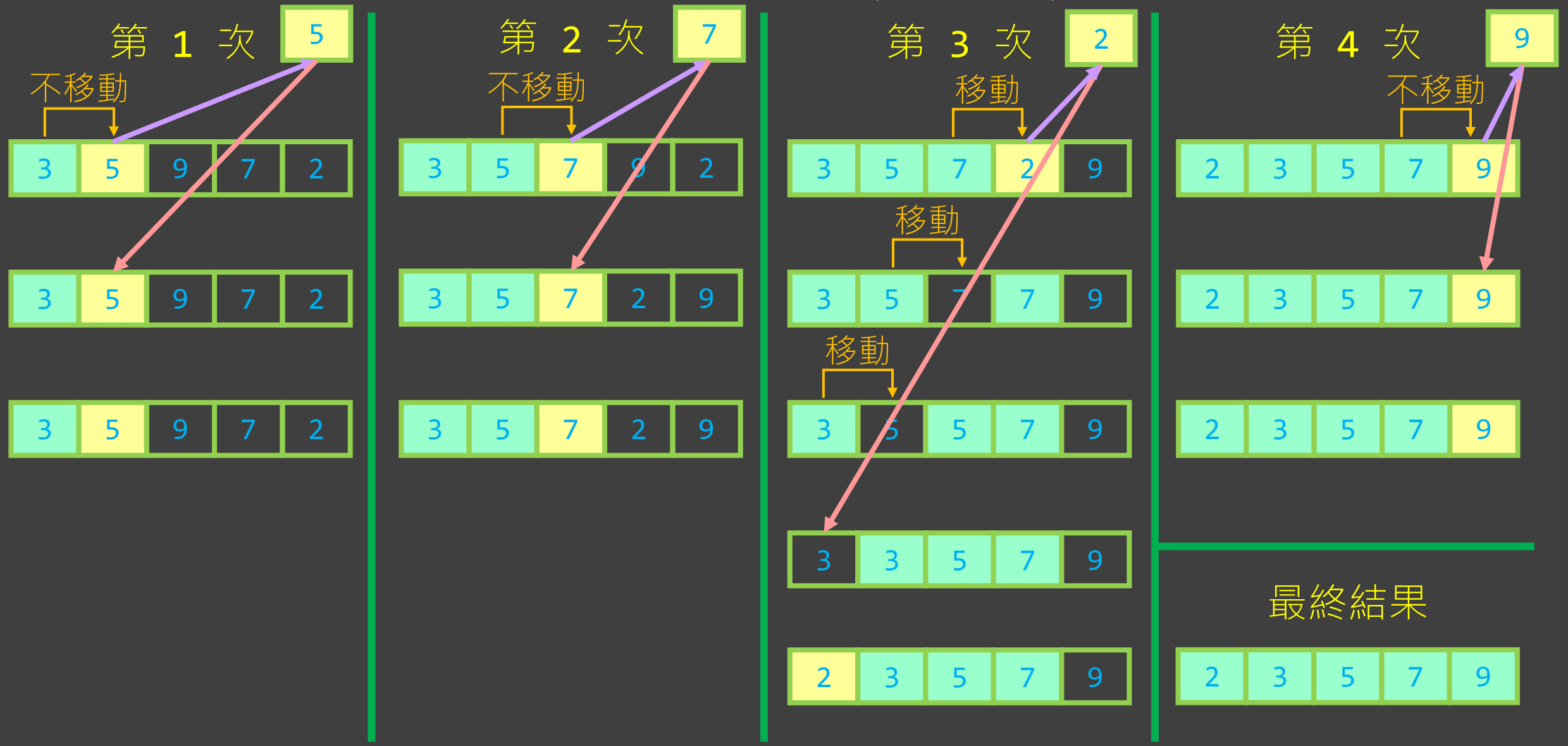
然後再將欲插入資料放入該放的位置

因共須將 $0 \sim 1 + 0 \sim 2 + \cdots + 0 \sim (n-1)$ 個不定數量的資料向後移動

平均總共須向後移動 $\frac{1+2+\cdots+n-1}{2} = \frac{n^2-1}{4}$ 個資料

所以對於陣列，整個插入排序法的時間複雜度為 $O(n^2)$

插入排序法(陣列)


第 1 次
5
不移動
3 5 9 7 2
3 5 9 7 2
3 5 9 7 2
第 2 次
7
不移動
3 5 7 9 2
3 5 7 2 9
3 5 7 2 9
第 3 次
2
移動
3 5 7 2 9
移動
3 5 7 7 9
移動
3 5 5 7 9
3 3 5 7 9
2 3 5 7 9
第 4 次
9
不移動
2 3 5 7 9
2 3 5 7 9
2 3 5 7 9
最終結果
2 3 5 7 9

插入排序法(陣列)

```java
import java.util.Arrays;
import java.util.Scanner;

public class Main4 {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        int n = scanner.nextInt(); // 獲取資料個數
        int[] arr = new int[n];
        for (int i = 0; i < n; i++) arr[i] = scanner.nextInt(); // 讀入資料
        // 插入排序
        for (int i = 1; i < n; i++) {
            int key = arr[i];
            int j = i - 1;
            while (j >= 0 && key < arr[j]) {
                arr[j + 1] = arr[j];
                j--;
            }
            arr[j + 1] = key;
        }
        System.out.println(Arrays.toString(arr));
    }
}
```

java

```
5
3 5 9 7 2
[2, 3, 5, 7, 9]
```

console

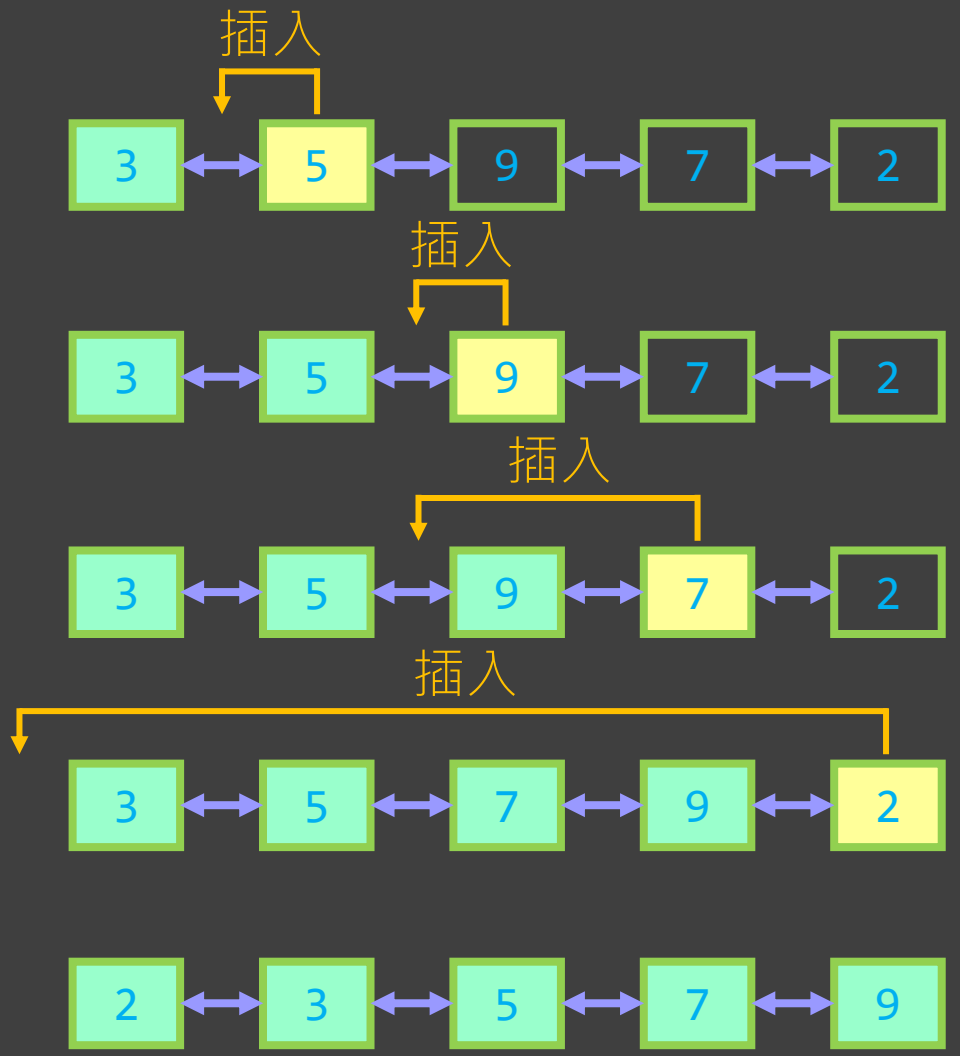
```
8
-1 8 -5 66 7 0 1 2
[-5, -1, 0, 1, 7, 8, 28, 66]
```

console

插入排序法(鏈結串列)

插入排序法對於鏈結串列
每個資料尋找應插入位置的時間複雜度為 $O(n)$
插入該資料的時間複雜度為 $O(1)$
故每個資料排序的時間複雜度為 $O(n)$，又重複 `n - 1` 次
所以對於鏈結串列，整個插入排序法的時間複雜度為 $O(n^2)$

插入排序法(鏈結串列)



插入排序法 (鏈結串列)

```java
public class Main5 {
    // 定義鏈結串列
    private static class IntNode {
        public final int value;
        public IntNode next;
        public IntNode previous;

        public IntNode(int value) {
            this.value = value;
        }

        public static void linkTwoNodes(IntNode previous, IntNode next) {
            if (Objects.nonNull(previous)) previous.next = next;
            if (Objects.nonNull(next)) next.previous = previous;
        }
    }
```

```
5
3 5 9 7 2
2 3 5 7 9
```
console

```
8
-1 8 -5 66 7 0 1 2
-5 -1 0 1 2 7 8 66
```
console

```java
public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        int n = scanner.nextInt(); // 獲取資料個數
        // 建立鏈結串列
        IntNode firstNode = null, currentNode = null;
        for (int i = 0; i < n; i++) {
            IntNode newNode = new IntNode(scanner.nextInt());
            if (Objects.isNull(firstNode)) {
                currentNode = newNode;
                firstNode = newNode;
            } else {
                IntNode.linkTwoNodes(currentNode, newNode);
                currentNode = newNode;
            }
        }
        // 插入排序
        IntNode sortLastNode = firstNode;
        for (int i = 1; i < n; i++) {
            IntNode keyNode = sortLastNode.next;
            currentNode = sortLastNode;
            while (Objects.nonNull(currentNode) && keyNode.value < currentNode.value)
                currentNode = currentNode.previous;
            if (currentNode == sortLastNode) sortLastNode = sortLastNode.next;
            else {
                IntNode.linkTwoNodes(keyNode.previous, keyNode.next);
                if (Objects.isNull(currentNode)) {
                    IntNode.linkTwoNodes(keyNode, firstNode);
                    firstNode = keyNode;
                } else {
                    IntNode.linkTwoNodes(keyNode, currentNode.next);
                    IntNode.linkTwoNodes(currentNode, keyNode);
                }
            }
        }
        // 輸出
        currentNode = firstNode;
        while (Objects.nonNull(currentNode)) {
            System.out.print(currentNode.value + " ");
            currentNode = currentNode.next;
        }
    }
}
```
java