

資料結構

報告者：徐湧棠
改編自劉定恆學長的簡報

目錄

- 第一天：簡介資料結構、陣列和對稱矩陣與Big-O
- 第二天：Linked List (鏈結串列)
- 第三天：Stacks & Queues (堆疊與佇列)
- 第四天：Trees (樹狀結構)
- 第五天：Sorting Algorithms (排序演算法)

資料結構是什麼？(Gemini)

- 資料結構是一種專門用於組織、儲存和管理電腦資料的手段，以便有效率地存取和使用數據，並定義資料元素之間的關係以及可執行的操作（例如搜尋、新增、刪除）。不同的資料結構，例如陣列(arrays)、鏈結串列(linked lists)、堆疊(stacks)、佇列(queues)、樹(trees)和圖(graphs)，適用於不同的任務，並針對程式的速度或記憶體使用進行最佳化。

關鍵概念

- 組織結構：以系統化的格式（例如，線性、層級結構）排列資料。
- 效率：選擇合適的結構可以提高性能（時間/空間），例如，搜尋、插入和刪除等操作。
- 關係：定義資料值之間的相互關係，（例如，順序、父子關係）。
- 操作：指定可套用於資料的函數。

Big O

如何評分程式寫得好不好？Big-O

電腦硬體速度差異很大，所以我們不用「秒數」來衡量程式快慢，而是看「當資料量 (N) 變大時，步驟增加的趨勢」，且通常專注於「最壞情況 (Worst Case)」。

$O(1)$ 常數時間

瞬間完成。無論 N 是一十筆還是一百萬筆，執行時間都不變。例如：讀取陣列的第一個元素。

$O(N)$ 線性時間

穩紮穩打。資料有 100 筆就要跑 100 步。例如：用迴圈從頭到尾掃描一次尋找資料。

$O(N^2)$ 平方時間

效能殺手。通常出現在「雙層巢狀迴圈」。資料量變 10 倍，時間會暴增 100 倍！

5 個常用的 Big-O 生活比喻與程式對照

複雜度	名稱	生活比喻	資料量 $n=1,000$ 時的步驟數	程式範例
$O(1)$	常數時間	開置物櫃拿東西	1 次	陣列取值 <code>arr[5]</code> 、 <code>Stack</code> 的 <code>pop()</code>
$O(\log n)$	對數時間	猜數字每次切一半	約 10 次	二元搜尋法 (Binary Search)
$O(n)$	線性時間	一個人一個人點名	1,000 次	用一個 <code>for</code> 迴圈找陣列裡的最大值
$O(n \log n)$	準線性時間	先分類再合併排序	約 10,000 次	快速排序 (Quick Sort)、合併排序 (Merge Sort)
$O(n^2)$	平方時間	全班每個人互相握手	1,000,000 次	雙重 <code>for</code> 迴圈、氣泡排序 (Bubble Sort)

例子

Given a sorted array with N elements

Example: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

Question:

Is number X in the array?

What is the worst case?

In other words, how many times does the loop run, if I want to find the number 9

// linear search $O(n)$

```
for (i = 0; i < n; i++)  
    if (a[i] == x) return true;
```

// Binary search $O(\log n)$

```
while (l <= r) {  
    mid = (l + r) / 2;  
    if (a[mid] == X) return true;  
    else if (a[mid] < X) l = mid + 1;  
    else r = mid - 1;  
}
```

例子



氣泡排序法(Bubble Sort)



重複地走訪整個數列，每次比較相鄰兩個元素，如果順序錯誤就把他們交換，直到不需要再交換為止

7	2	9	6
7	2	9	6
2	7	9	6
2	7	9	6
2	7	6	9
2	7	6	9
2	7	6	9
2	6	7	9
2	6	7	9
2	6	7	9
2	6	7	9

Pseudocode

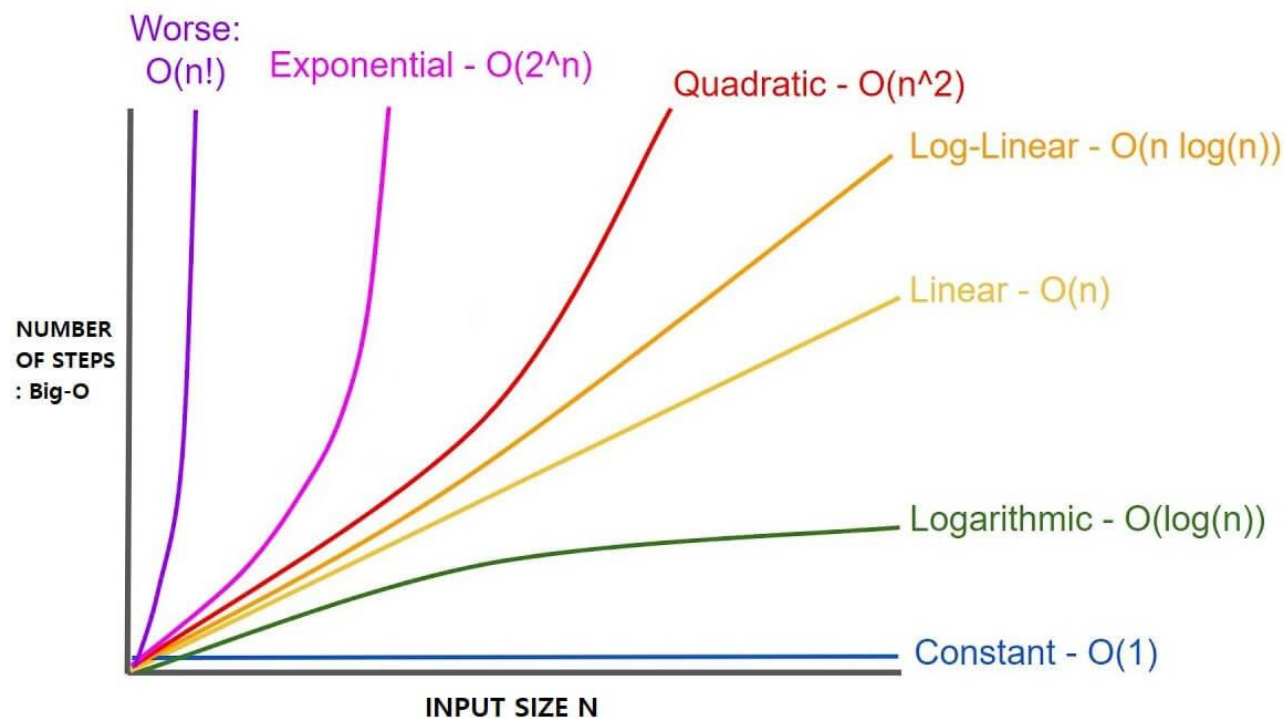
```
for i = 0 to N-1
  for j = 0 to N-i-2
    if a[j] > a[j+1]
      swap(a[j], a[j+1])
```

What is the time complexity of the bubble sort?

Big O

小提醒：在算 Big-O 時抓大放小的原則：
省略常數： $O(2n)$ 依然記為 $O(n)$ （因為量級一樣）。

只留最高次方： $O(n^2 + n)$ 記為 $O(n^2)$
（因為當 n 很大時， n 可以忽略不計）。



$$O(\log n) < O(\sqrt{n}) < O(n) < O(n \log n) < O(n^2) < O(2^n) < O(n!) < O(n^n)$$

What is the Big(O) of this function?

```
int mystery(int n) {  
    int count = 0;  
    for (int i = 1; i <= n; i *= 2) {  
        for (int j = 0; j < i; j++) {  
            count++;  
        }  
    }  
    return count;  
}
```

第 1 圈 (當 $i = 1$) : 內層跑 **1** 次

第 2 圈 (當 $i = 2$) : 內層跑 **2** 次

第 3 圈 (當 $i = 4$) : 內層跑 **4** 次

第 4 圈 (當 $i = 8$) : 內層跑 **8** 次

...

最後一圈 (當 $i = 2^k$) : 內層跑 2^k 次 (注意, 這裡的 $2^k \leq n$)

所以, 整支程式 `count++` 被執行的總次數 $T(n)$ 會是這串數字的和:

$$T(n) = 1 + 2 + 4 + 8 + \dots + 2^k$$
$$\text{總和} = \frac{1 \times (1 - 2^{k+1})}{1 - 2} = 2^{k+1} - 1 = 2 \times 2^k - 1$$

因為迴圈的條件是 $i \leq n$, 所以 $2^k \leq n$

根據 Big-O 的「抓大放小」原則 (忽略常數倍數與減法) :

我們把 $2n - 1$ 的常數 2 和 -1 省略掉, 最終的複雜度就是: $O(n)$

Array

Array & Vector

	Array (e.g. <code>int arr[5];</code>)	Vector (e.g. <code>std::vector<int> vec;</code>)
Size	Fixed; declared size cannot change	Dynamic; can grow or shrink
Memory	Contiguous block, efficient access	vector can reallocate; elements remain contiguous, but addresses may change.
Flexibility	Low; must know size beforehand.	High; handles resizing automatically.
Use Case	When size is known and performance is critical, like fixed-data structures	General-purpose lists, when element count varies

什麼是 Array (陣列)？極速 $O(1)$

Array 就像是火車站裡「連續的置物櫃」。

- **記憶體連續**：作業系統會分配一整塊存放相同型態變數「連續且固定大小」的記憶體。
- **Index (索引) 為何從 0 開始**？因為它代表**距離起點的位移 (Offset)**。
- **極速讀取**：要找第 N 個資料，電腦不用從頭數，可以用數學公式直接「算」出位置，時間複雜度為 $O(1)$ 。

ROW(列) COLUMN(行)

$\text{int A}[2][3] = \{ \{0, 1, 2\}, \{3, 4, 5\} \};$

COLUMN 共有3行

ROW	$A[0][0]$ 0	$A[0][1]$ 1	$A[0][2]$ 2
共有2列	$A[1][0]$ 3	$A[1][1]$ 4	$A[1][2]$ 5

Array 的致命傷：插入與刪除 $O(N)$

既然查詢這麼快，為什麼我們還需要其他資料結構？因為 Array **不允許插隊**！

問題：硬要插隊

假設置物櫃已經裝了 A, B, C，現在有一個 X 要硬擠進 Index 1 的位置。

痛苦大搬風

為了騰出位置，原本在 Index 1 和之後的資料 (B, C)，全部都要 **往後搬移一格**。

代價高昂

如果資料有 10 萬筆，就要搬 10 萬次。這使得 Array 的插入與刪除時間複雜度高達 **$O(N)$** 。



Array(陣列)

Q:完成以下程式



宣告維度是4的字串陣列，依序存入：Ironman Hulk Thor CaptainAmerica



用函式刪除Thor，並用函式插入" Black Widow" 到Hulk的前面



用函式把陣列顛倒，並用函式印出所有元素

```
Ironman Hulk Thor CaptainAmerica
====Deleting element: Thor
Ironman Hulk CaptainAmerica
====Inserting element: Black Widow at
position: 1
Ironman Black Widow Hulk CaptainAmerica
====Reversing array
CaptainAmerica Hulk Black Widow Ironman
```

對稱矩陣

什麼是對稱矩陣？

- 它是一個 $N \times N$ 的二維陣列 (矩陣)。
- 特徵是：以左上到右下的「主對角線」為軸，**右上角的數值完全等於左下角的數值**。
- 數學定義：對於所有的 i, j ，都有 $A[i][j] == A[j][i]$ 。
- 應用：例如「城市間的距離表」，台北到台中的距離，等於台中到台北的距離。
- 只需存取上/下三角，節省一半空間

$$A = \begin{bmatrix} -9 & 2 & -3 & 7 \\ 2 & 2 & -4 & 0 \\ -3 & -4 & 4 & 5 \\ 7 & 0 & 5 & -5 \end{bmatrix}$$

黃色為主對角線，兩側數值完全對稱

為什麼要壓縮對稱矩陣？

既然兩邊的資料一模一樣，存兩份不就浪費記憶體了嗎？

- **目標：**我們只儲存「主對角線」加上「下三角矩陣 (Lower Triangular)」的部分。
- 原本需要 $N \times N$ 的空間，壓縮後我們只需要一個一維陣列 (1D Array) 來裝。
- **節省空間：**需要的空間大幅減少到 $N(N+1)/2$ 。

Q:用一維陣列表示，元素個數為多少？(n=4)
[-9 2 2 -3 -4 4 7 0 5 -5]->10個

$$A = \begin{bmatrix} -9 & 2 & -3 & 7 \\ 2 & 2 & -4 & 0 \\ -3 & -4 & 4 & 5 \\ 7 & 0 & 5 & -5 \end{bmatrix}$$

將 2D 對應到 1D

假設我們按「列 (Row)」把下三角元素依序放入 1D Array：

$$A[i][j] \rightarrow A' [i']$$

$$i' = j + i(i+1)/2$$

$$A[2][0] \rightarrow A' [3]$$

(條件： $i \geq j$ ，也就是只取下半部)



對稱陣列

Q:使用一維陣列完成以下程式，使輸出結果如右



```
Struct symmetricArray{  
    int* data;  
  
    int n; //矩正大小  
  
    建構函式  
  
    解構函式  
  
    set函式  
  
    plus函式  
  
    show函式  
  
}
```

```
int main(){  
    //宣告3個對稱陣列  
    //設定A和B的元素值  
    //陣列C=A+B  
  
}  
  
symmetricArray(int N)  
void set(int i, int j, int value)  
void show()  
void plus(symmetricArray A, symmetricArray B)
```

Tips:

$A[i][j] \rightarrow A' [i']$

$i' = j + i(i+1)/2$

$A[2][0] \rightarrow A' [3]$

```
0  
1 2  
3 4 5  
//A陣列  
0  
2 4  
6 8 10  
//B陣列  
0  
3 6  
9 12 15  
//C陣列
```

HW





HW1

使用二維陣列完成以下程式，使輸出結果如右



```
Struct symmetricArray{  
    int** data;  
  
    int n;  
    建構函式  
    set函式  
    plus函式  
    show函式 }
```

```
int main(){  
    //宣告3個對稱陣列  
    //用for迴圈設定元素值  
    //陣列C=A+B  
}
```

```
0  
1 2  
3 4 5  
//A陣列  
0  
2 4  
6 8 10  
//B陣列  
0  
3 6  
9 12 15  
//C陣列
```