搜尋、排序與 Big O：用對演算法、看懂效率

Deitel & Deitel《Java How to Program: Early Objects》11e, Ch.19

§ 1　一、名詞速查表

中文	English	一句話定義	科目	章節
搜尋	searching	在一組資料中找出某個目標值的位置	Java	CH19
線性搜尋	linear search	從頭逐一比對，找到或掃完為止	Java	CH19
二分搜尋	binary search	在「已排序」陣列每次砍半縮小範圍	Java	CH19
排序	sorting	把資料依大小重新排成遞增（或遞減）	Java	CH19
選擇排序	selection sort	每輪選出剩餘最小值，換到前面就位	Java	CH19
插入排序	insertion sort	每筆往左插入到前段已排序的正確位置	Java	CH19
合併排序	merge sort	分治：拆半各自排好，再合併兩段	Java	CH19
分治	divide and conquer	把大問題拆成小問題、解完再合起來	Java	CH19
Big O 記號	Big O notation	描述執行量隨資料量 n 成長的「級數」	Java	CH19
常數時間	O(1) / constant time	不隨 n 增加，固定步數完成	Java	CH19
線性時間	O(n) / linear time	工作量與 n 成正比	Java	CH19
對數時間	O(log n) / logarithmic	每步砍半，n 加倍只多一步	Java	CH19
平方時間	$O(n^2)$ / quadratic time	工作量與 n 的平方成正比，巢狀迴圈常見	Java	CH19
哨兵值	sentinel value	找不到時回傳的特殊值（如 `-1`）	Java	CH19
鍵值	search key	要找的目標值	Java	CH19

§ 2　二、核心概念

兩種最基本的資料處理：**搜尋**（在一堆資料裡找某個值在哪）與**排序**（把資料排好）。同一件事常有好幾種演算法，差別不在「做不做得到」，而在「資料變多時要花多少工」。描述這個成長速度的工具就是 **Big O 記號**。

> Big O 看的是**最壞情況下、隨 n 成長的級數**，不是精確秒數。`O(n)` 和 `O(2n)`、`O(n+100)` 都算 `O(n)`：常數倍與低次項在 n 很大時可忽略，只留最高次項。

搜尋有兩條路：資料**沒排序**只能**線性搜尋**（逐格掃，`O(n)`）；資料**已排序**可用**二分搜尋**（每次砍半，`O(log n)`，快得多）。代價是得先排序。

排序有多種演算法：**選擇排序**、**插入排序**都是 `O(n²)`（巢狀迴圈、好懂但慢）；**合併排序**用分治到 `O(n log n)`（快、但要額外空間）。挑哪個，看資料量與情境，這正是 Big O 幫你判斷的事。

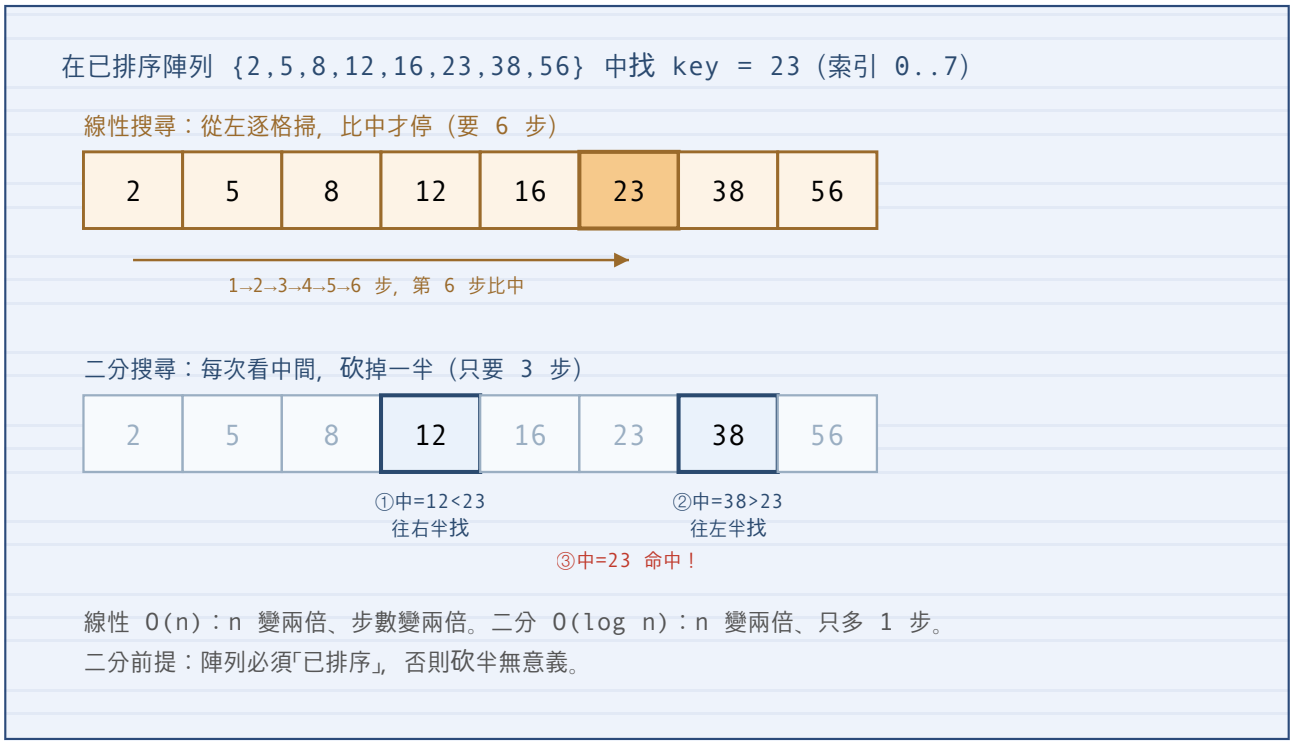
§ 3 三、主要內容

3.1 1. 線性搜尋（Linear Search）

```java
// 從頭逐一比對，找到回傳索引，掃完沒找到回傳 -1（哨兵值）
public static int linearSearch(int[] data, int key) {
    for (int i = 0; i < data.length; i++) {
        if (data[i] == key)    // 比中了
            return i;          // 回傳所在索引
    }
    return -1;                 // 整個掃完都沒有
}
```

- 資料**不必排序**，任何陣列都能用。
- 最壞情況（目標在最後或不存在）要比對 `n` 次 → `O(n)`。
- 平均也約掃一半 → 還是 `O(n)`。

3.2 2. 線性搜尋 vs 二分搜尋對照圖（招牌）



3.3 3. Big O 記號：看成長級數

`O(...)` 描述「資料量 n 變大時，工作量怎麼成長」，只留最高次項、丟掉常數倍與低次項：

Big O	名稱	意思	例子
`O(1)`	常數	不隨 n 變，固定步數	取陣列某一格 `a[i]`、比大小
`O(log n)`	對數	每步砍半，n 加倍只多一步	二分搜尋
`O(n)`	線性	與 n 成正比	線性搜尋、走訪一次
`O(n log n)`	線性對數	比 O(n) 略多，遠優於 $O(n^2)$	合併排序
`O(n²)`	平方	與 n^2 成正比（巢狀迴圈）	選擇排序、插入排序

```
n = 1000 時的差距（直覺感受）：
  O(log n) ≈ 10        O(n) = 1000
  O(n log n) ≈ 1萬   O(n²) = 100萬
```

- **化簡規則**：`O(2n)`→`O(n)`、`O(n²+n)`→`O(n²)`、`O(5)`→`O(1)`。
- **巢狀迴圈**：兩層各跑 n 次 → 約 `n × n` = `O(n²)`。

3.4 4. 二分搜尋（Binary Search）

```java
// 前提：data 必須已排序（遞增）。每次比中間，砍掉不可能的一半。
public static int binarySearch(int[] data, int key) {
    int low = 0, high = data.length - 1;
    while (low <= high) {
        int mid = (low + high) / 2;      // 中間索引
        if (key == data[mid])  return mid;      // 命中
        else if (key < data[mid]) high = mid - 1;  // 在左半
        else                      low  = mid + 1;  // 在右半
    }
    return -1;                           // 範圍縮到空，沒找到
}
```

- **每一輪把搜尋範圍砍半** → 最多約 $\log_2 n$ 輪 → $O(\log n)$。
- 1000 筆線性搜尋最壞 1000 次；二分搜尋最多約 10 次。
- **務必先排序**；迴圈條件是 `low <= high`（相等時還剩一格要檢查）。

3.5 5. 選擇排序（Selection Sort）逐輪圖（招牌）

3.6 6. 選擇排序的程式

```java
// 每一輪：在 [i..length-1] 找最小，換到 i 就位
public static void selectionSort(int[] a) {
    for (int i = 0; i < a.length - 1; i++) {
        int minIdx = i;                  // 假設目前 i 最小
        for (int j = i + 1; j < a.length; j++)
            if (a[j] < a[minIdx]) minIdx = j;   // 找更小的
        int t = a[i]; a[i] = a[minIdx]; a[minIdx] = t;  // 交換就位
    }
}
```

- 外迴圈 `n` 輪、每輪內迴圈約掃 `n` 格 → 巢狀 → `O(n²)`。
- 不管原本排得多亂或多齊，比較次數都一樣（不會因「快排好」而變快）。

3.7 7. 插入排序（Insertion Sort）

```java
// 每筆 a[i] 往左插入到前段已排序的正確位置
public static void insertionSort(int[] a) {
    for (int i = 1; i < a.length; i++) {
        int key = a[i];                  // 待插入的這一筆
        int j = i - 1;
        while (j >= 0 && a[j] > key) {   // 比 key 大的往右挪一格
            a[j + 1] = a[j];
            j--;
        }
        a[j + 1] = key;                  // key 放進空出來的位置
    }
}
```

- 想像整理手上的撲克牌：每抽一張，往左插到對的位置。
- 最壞（完全逆序）仍是 `O(n²)`；但若資料「幾乎排好」會很快（接近 `O(n)`）。

3.8 8. 合併排序（Merge Sort）：分治到 O(n log n)

```java
// 分治：拆成兩半各自排好，再合併兩段已排序陣列
public static void mergeSort(int[] a, int low, int high) {
    if (low < high) {                        // 還有 2 格以上才需排
        int mid = (low + high) / 2;
        mergeSort(a, low, mid);              // 排左半
        mergeSort(a, mid + 1, high);         // 排右半
        merge(a, low, mid, high);            // 合併兩段
    }
}
```

- **分治三步**：拆半 → 各自遞迴排好 → 合併（merge：兩段已排序，從頭比較較小者放回）。
- 拆 `log n` 層、每層合併共掃 `n` 格 → `O(n log n)`，遠快於 `O(n²)`。
- 代價：合併要用到額外的暫存陣列（不是「原地」排序）。

3.9 9. 排序演算法效率對照

演算法	時間	額外空間	特點
選擇排序	`O(n²)`	`O(1)`	最好懂、交換次數少、不因資料而變快
插入排序	`O(n²)`（最好 `O(n)`）	`O(1)`	資料近乎排好時很快
合併排序	`O(n log n)`	`O(n)`	大資料快很多，需額外空間

- 資料小、求簡單 → 選擇／插入排序就夠。
- 資料大、要快 → 合併排序（或實務上的 `Arrays.sort`）。

§ 4 四、語法與 API 速查

```
// 線性搜尋（不必排序）：找到回索引、否則 -1
for (int i = 0; i < a.length; i++) if (a[i] == key) return i;
return -1;

// 二分搜尋（須已排序）：每次砍半
int low = 0, high = a.length - 1;
while (low <= high) {
    int mid = (low + high) / 2;
    if (key == a[mid]) return mid;
    else if (key < a[mid]) high = mid - 1;
    else                   low  = mid + 1;
}

// 選擇排序：每輪選最小換到前面
for (int i = 0; i < a.length - 1; i++) {
    int m = i;
    for (int j = i + 1; j < a.length; j++) if (a[j] < a[m]) m = j;
    int t = a[i]; a[i] = a[m]; a[m] = t;
}

// 插入排序：每筆往左插到對的位置
for (int i = 1; i < a.length; i++) {
    int key = a[i], j = i - 1;
    while (j >= 0 && a[j] > key) { a[j+1] = a[j]; j--; }
    a[j+1] = key;
}

// 交換兩格的標準寫法
int t = a[x]; a[x] = a[y]; a[y] = t;

// 標準函式庫（實務直接用，不必手寫）
java.util.Arrays.sort(a);             // 排序整個陣列
java.util.Arrays.binarySearch(a, k); // 二分搜尋（須先排序）
```

- **搜尋選型**：沒排序 → 線性 `O(n)`；已排序 → 二分 `O(log n)`。
- **Big O 化簡**：丟常數倍與低次項，只留最高次項。

- **巢狀迴圈**＝平方時間的訊號（`O(n²)`）。

§ 5 五、常見錯誤

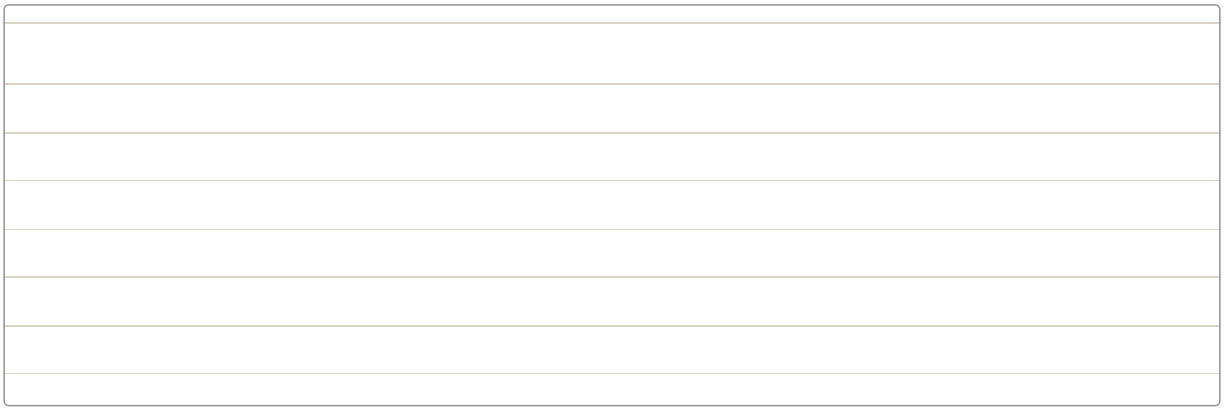
- **沒排序就二分搜尋**：二分前提是「已排序」，亂序陣列砍半的結果無意義、會找錯或找不到。
- **二分迴圈條件寫 `low < high`**：相等時還剩一格沒檢查，會漏掉答案；要用 `low <= high`。
- **二分更新邊界沒 ±1**：`high = mid`／`low = mid`（少了 `-1`／`+1`）在剩一格時造成無窮迴圈；要 `high = mid - 1`、`low = mid + 1`。
- **線性搜尋沒回哨兵值**：找不到要回 `-1`（或約定值），呼叫端才能判斷「沒找到」。
- **找最大／最小初值設 0**：選擇排序找最小應從 `a[i]`（或記索引 `minIdx=i`）起，不要設 0；全負數會錯。
- **交換少了暫存變數**：`a[x]=a[y]; a[y]=a[x];` 會把兩格都變成同值；要先存 `int t=a[x]`。
- **把 Big O 當精確秒數**：Big O 是成長級數、看 n 很大時的趨勢，不是「跑幾秒」；`O(n)` 不一定比 `O(n²)` 在小 n 時快。
- **常數倍誤以為改變級數**：跑兩個各 n 次的「非巢狀」迴圈是 `O(n)`（=`O(2n)`），不是 `O(n²)`；只有「巢狀」才相乘。
- **插入排序內層忘記 `j>=0` 檢查**：往左挪到開頭時 `a[j]` 會越界；條件要 `j >= 0 && a[j] > key`。

§ 6 六、練習題

例題 1：線性搜尋回傳索引

寫一個 `static int linearSearch(int[] data, int key)`，從頭逐一比對，找到回傳索引、沒找到回傳 `-1`。在 `main` 中以 `int[] a = {12, 5, 8, 20, 3}` 找 `20`（應印 `3`）與 `99`（應印 `-1`）。

> 1. `for (int i = 0; i < data.length; i++)`：`if (data[i]==key) return i` 2. 迴圈結束（都沒比中）→ `return -1` 3. `main` 呼叫兩次、各印結果



```
public class LinearSearch {
    public static int linearSearch(int[] data, int key) {
        for (int i = 0; i < data.length; i++)
            if (data[i] == key) return i;
        return -1;
    }
    public static void main(String[] args) {
        int[] a = {12, 5, 8, 20, 3};
        System.out.println(linearSearch(a, 20));   // 3
        System.out.println(linearSearch(a, 99));   // -1
    }
}
```

> 易錯：找不到沒回 `-1`；比較寫成 `=`（賦值）而非 `==`。

例題 2：二分搜尋（已排序陣列）

給已排序 `int[] a = {2, 5, 8, 12, 16, 23, 38, 56}`，寫 `static int binarySearch(int[] a, int key)` 用每次砍半找值；在 `main` 找 `23`（印 `5`）與 `7`（印 `-1`）。

1. `low=0`、`high=a.length-1`；`while (low <= high)` 2. `mid=(low+high)/2`；命中回 `mid`、`key`

```
public class BinarySearch {
    public static int binarySearch(int[] a, int key) {
        int low = 0, high = a.length - 1;
        while (low <= high) {
            int mid = (low + high) / 2;
            if (key == a[mid]) return mid;
            else if (key < a[mid]) high = mid - 1;
            else                   low  = mid + 1;
        }
        return -1;
    }
    public static void main(String[] args) {
        int[] a = {2, 5, 8, 12, 16, 23, 38, 56};
        System.out.println(binarySearch(a, 23));   // 5
        System.out.println(binarySearch(a, 7));    // -1
    }
}
```

易錯：條件用 `low < high` 漏一格；邊界沒 ±1 造成無窮迴圈；陣列沒排序就用二分。

例題 3：選擇排序並印出排序前後

給 `int[] a = {29, 10, 14, 37, 13}`，用**選擇排序**排成遞增，印出排序前與排序後（各一行，空白分隔）。

> 1. 先印原陣列 2. 外迴圈 `i` 從 0 到 `length-2`：內迴圈找 `[i..]` 最小的索引 `minIdx` 3. 交換 `a[i]` 與 `a[minIdx]`；排完再印一次

```
import java.util.Arrays;
public class SelectionSort {
    public static void selectionSort(int[] a) {
        for (int i = 0; i < a.length - 1; i++) {
            int minIdx = i;
            for (int j = i + 1; j < a.length; j++)
                if (a[j] < a[minIdx]) minIdx = j;
            int t = a[i]; a[i] = a[minIdx]; a[minIdx] = t;
        }
    }
    public static void main(String[] args) {
        int[] a = {29, 10, 14, 37, 13};
        System.out.println("Before: " + Arrays.toString(a));
        selectionSort(a);
        System.out.println("After:  " + Arrays.toString(a));  // [10, 13,
14, 29, 37]
    }
}
```

易錯：交換少了暫存變數導致兩格同值；`minIdx` 初值設 0 而非 `i`。

§ 7 七、自我檢核

- [ ] 會寫線性搜尋（找到回索引、沒找到回 `-1`），知道它不必排序、最壞 `O(n)`。
- [ ] 會寫二分搜尋（`low<=high`、`mid`、邊界 `±1`），知道它須先排序、`O(log n)`。
- [ ] 講得出 Big O 的意義（成長級數、丟常數與低次項），分得清 `O(1)`／`O(log n)`／`O(n)`／`O(n log n)`／`O(n²)`。
- [ ] 知道巢狀迴圈通常是 `O(n²)`，兩個非巢狀迴圈仍是 `O(n)`。
- [ ] 會寫選擇排序（每輪選最小換到前面）、知道它 `O(n²)`。
- [ ] 會寫插入排序（每筆往左插到對的位置）、知道它最壞 `O(n²)`、近乎排好時接近 `O(n)`。
- [ ] 講得出合併排序用分治達 `O(n log n)`、代價是額外空間。
- [ ] 會用 Big O 比較演算法效率，知道資料量大時 `O(n log n)` 遠優於 `O(n²)`、二分遠優於線性。