進階資料結構與演算法

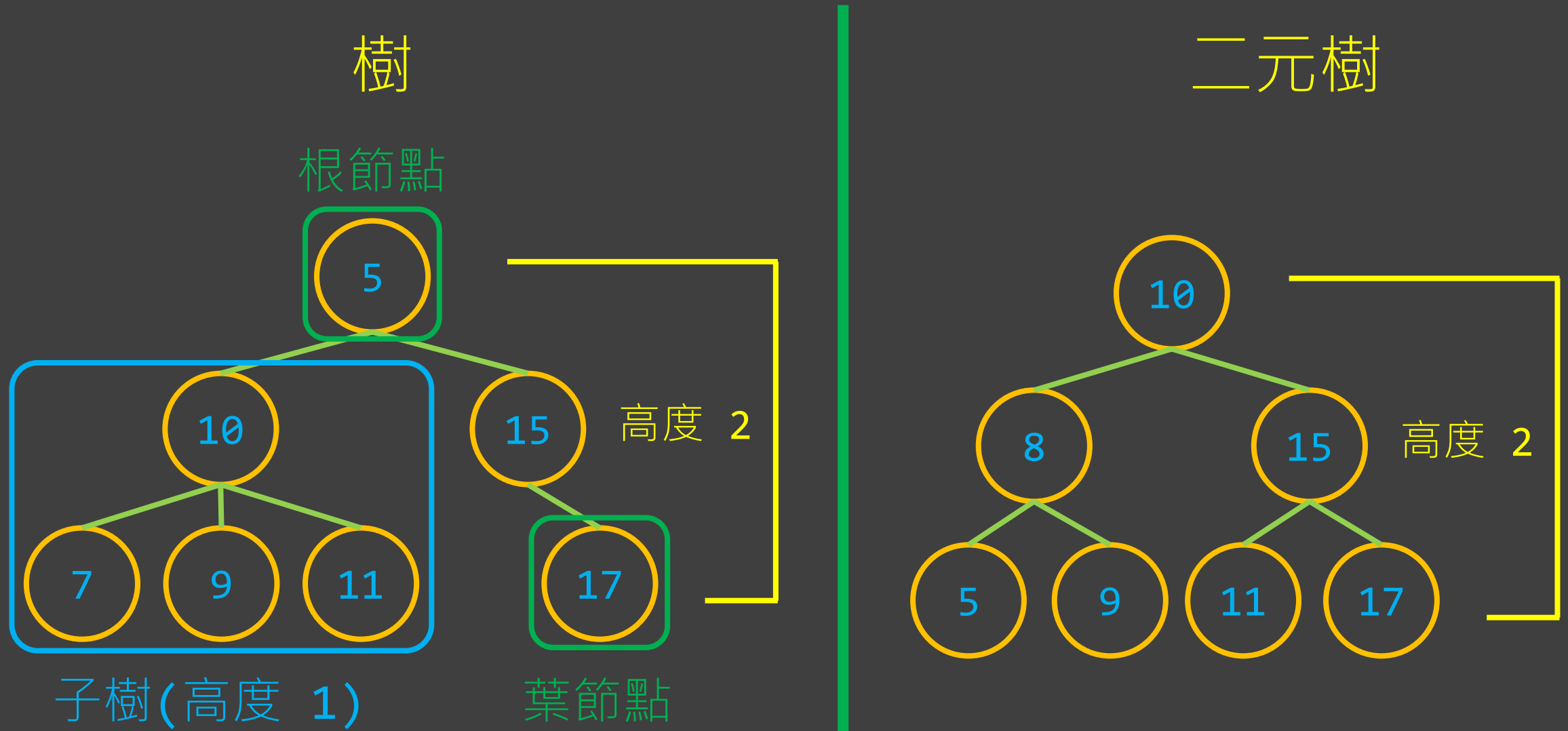
TYIC 桃高資訊社

樹與二元樹

樹(tree)是一種資料結構，樹中的每個資料稱為節點(node)
樹中的資料(節點)不可重複，且節點會連結其他的節點
連結的節點之間會形成父子關係，但連結不可成環
每個節點只有一個父節點，但可以有很多個子節點
相同父節點的節點為兄弟節點，父節點的兄弟節點為叔伯節點
父節點的父節點為祖父節點，父節點為兄弟節點的節點為堂兄弟節點
樹中的首個資料為根節點(root)，無子節點的資料為葉節點(leave)
從任一葉節點到根節點的最大路徑數為該樹的高度(height)

二元樹(binary tree)是一種樹，但每個節點最多只有兩個子節點

樹與二元樹



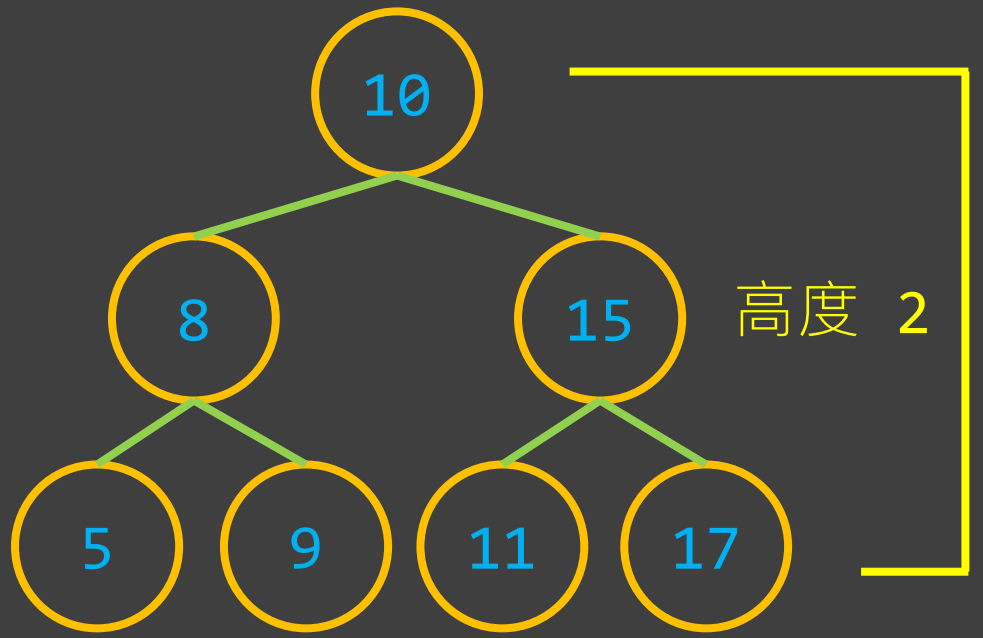
二元搜尋樹

二元搜尋樹(binary search tree)是一種特殊的二元樹
在二元搜尋樹中，比根節點小的資料會放到左子樹中
比根節點大的資料會放到右子樹中
這樣使用二分搜尋法就會非常快速
所以其存取(access)效率較鏈結串列高、較陣列低
搜尋(search)效率較陣列和鏈結串列高
插入(insert)、刪除(delete)效率較陣列高、較鏈結串列低

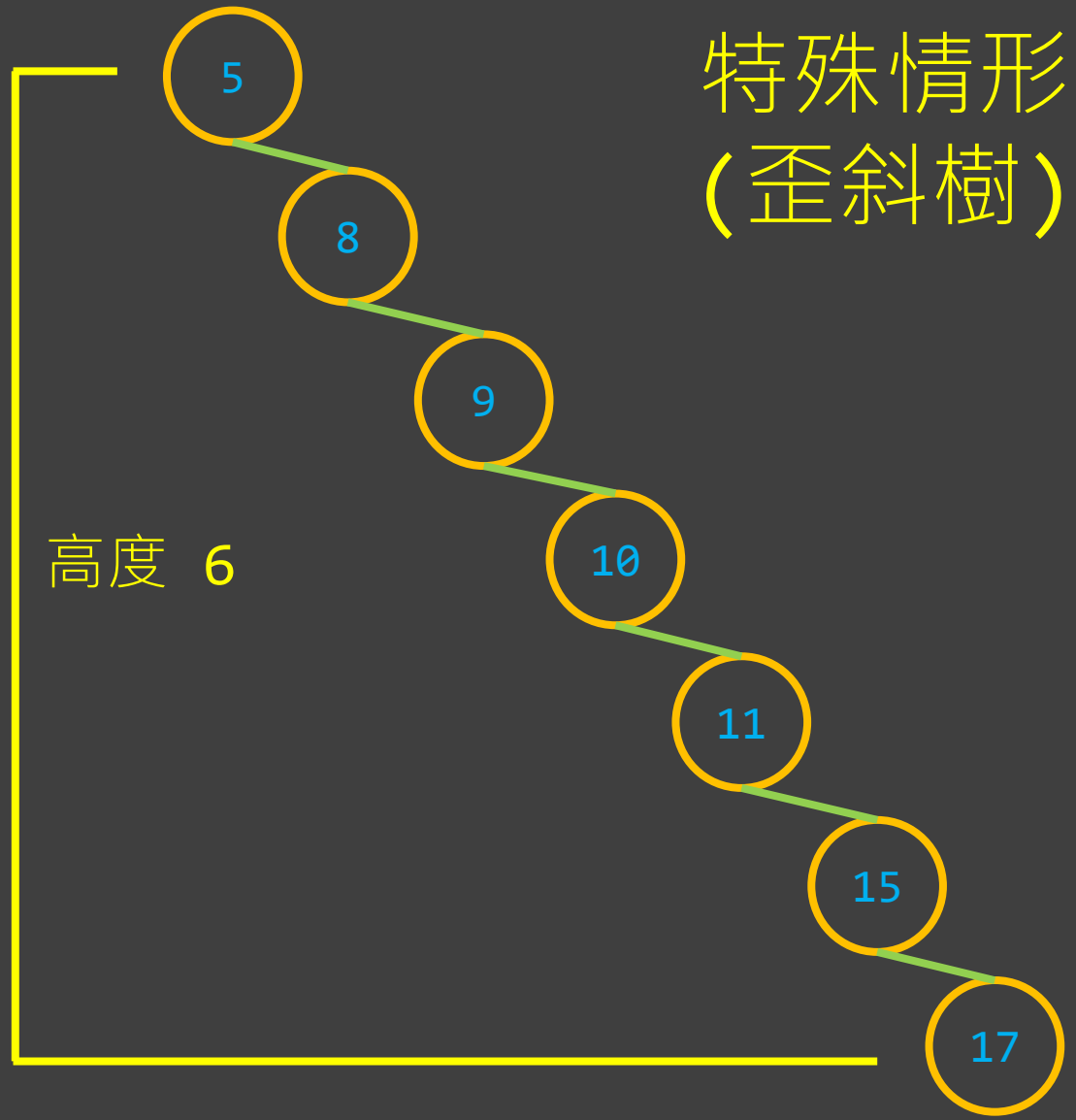
若資料已排序，則放入二元搜尋樹後會變為歪斜樹(skewed tree)

二元搜尋樹

正常情形



特殊情形
(歪斜樹)



紅黑樹

紅黑樹(red-black tree)是一種二元搜尋樹

紅黑樹會自平衡(self-balancing)，避免出現剛剛的特殊情形

紅黑樹的葉節點皆為空資料(null 或 nil)，並定義了幾條規則：

1. 節點是紅色或黑色 2. 根節點是黑色

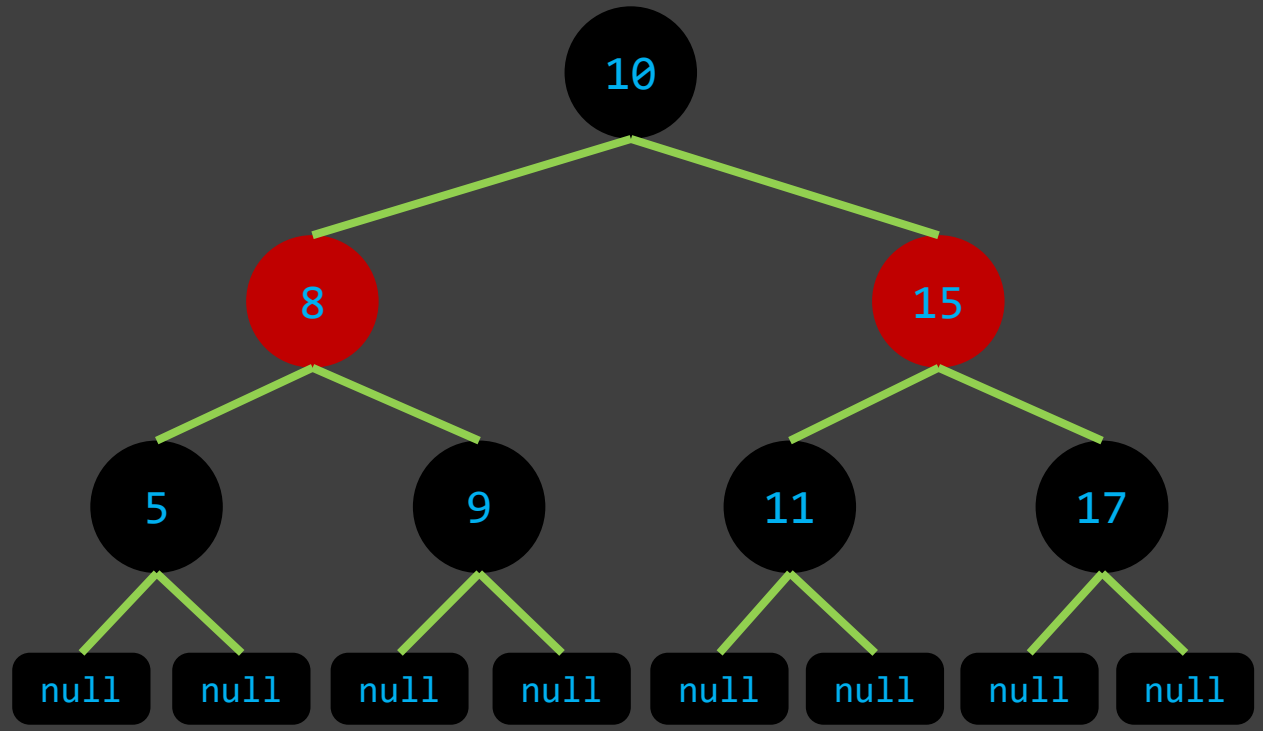
3. 所有的葉節點都是黑色 4. 相接節點不能皆為紅色

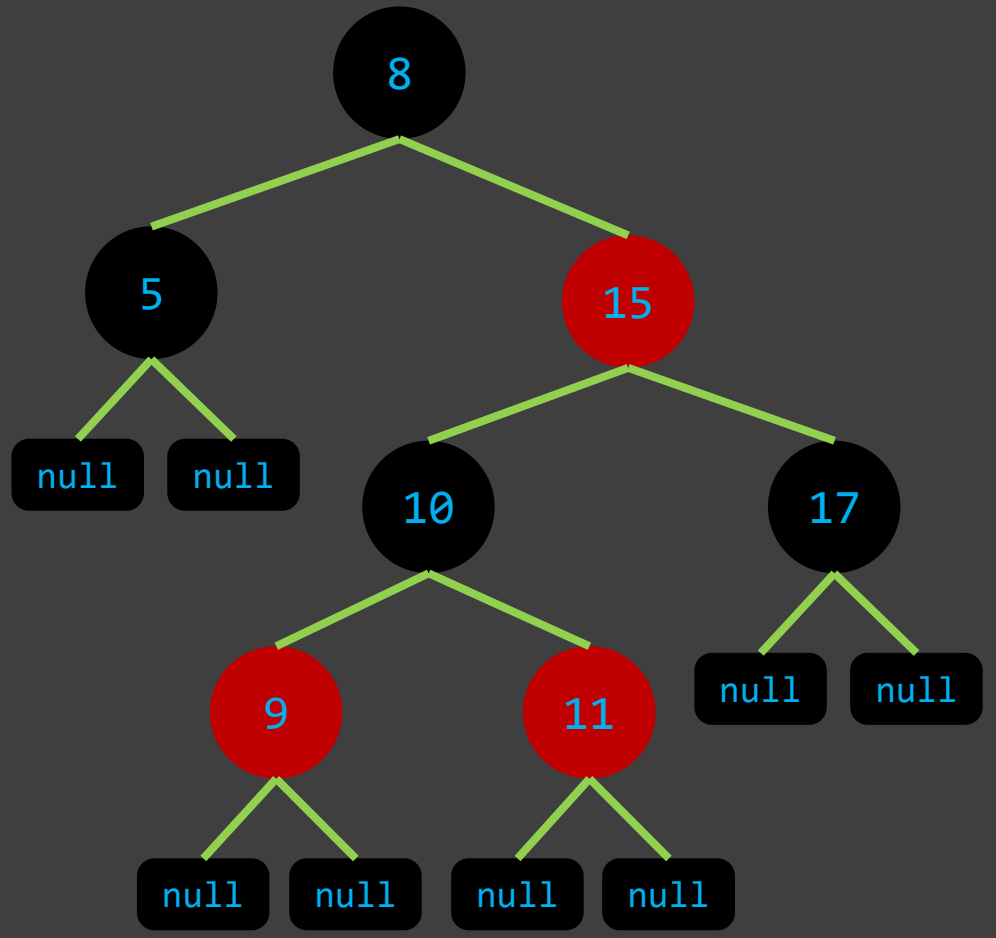
5. 從根節點到任一葉節點的黑色節點數量皆相同

這樣的規則使得最長路徑長度不超過最短路徑長度的兩倍

紅黑樹線上模擬器

紅黑樹


10
8
15
5
9
11
17
null
null
null
null
null
null
null
null


8
5
15
null
null
10
17
9
11
null
null
null
null
null
null

分治法

分治法(divide and conquer)

是指將解決問題分為三部分：

分：將大問題拆解成多個相似且獨立的小問題

治：分別解決小問題

若小問題也不好解決

可再次分治直到小問題可以解決

合：將小問題的答案合併成大問題的答案

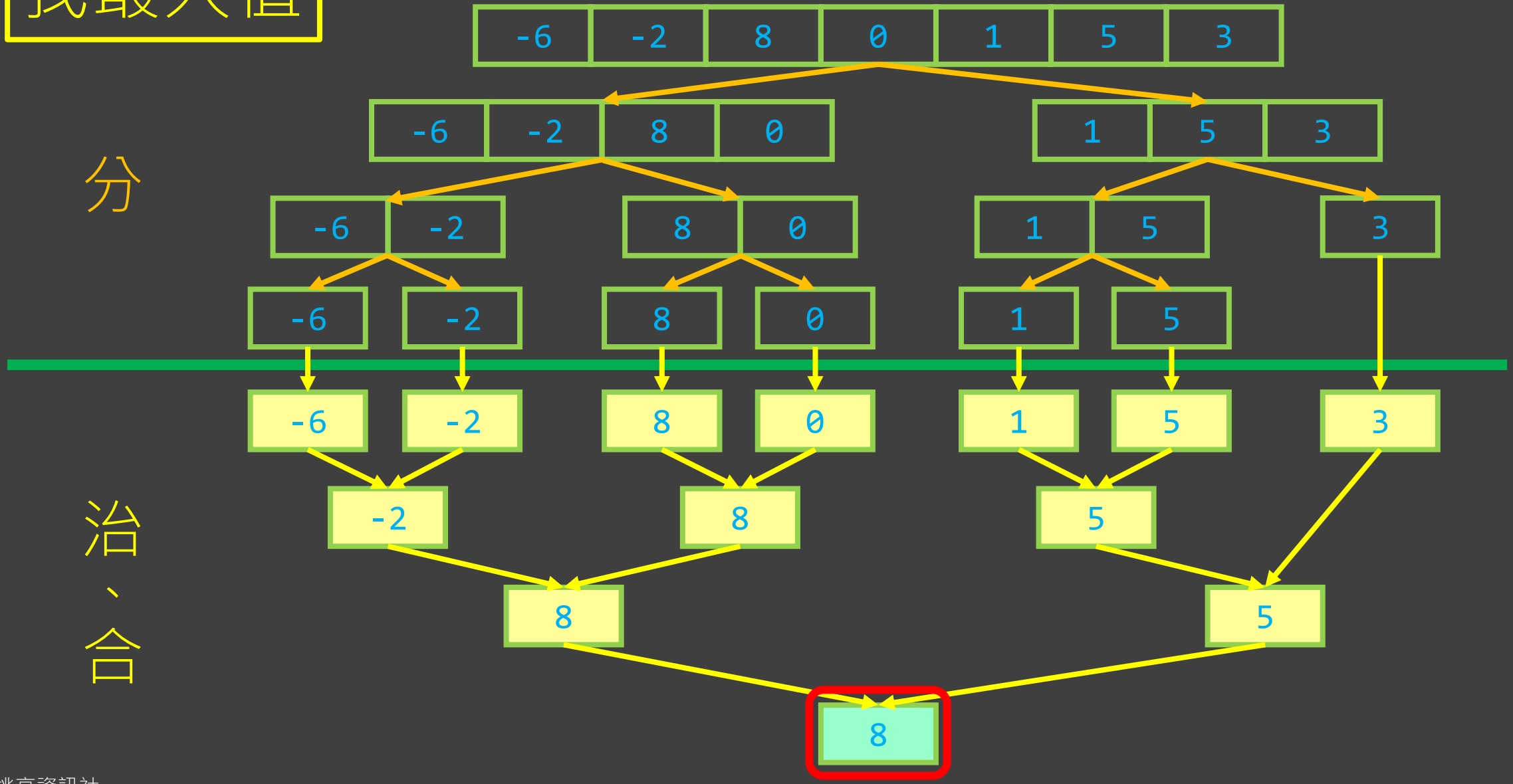
通常分治法會以遞迴實現

分治法找最大、最小值

找最大、最小值除了可以用時間複雜度為 $O(n)$ 迴圈法外
也可以使用分治法來解決，時間複雜度為 $O(\log n)$ ：
分：將資料拆分為兩組
治：找出拆分資料的最大、最小值
若只有一個資料，則該資料即為最大、最小值
否則繼續分治，直到可以找出最大、最小值
合：比較兩組資料的最大、最小值
得出原資料的最大、最小值

分治法找最大、最小值

找最大值



分治法找最大、最小值

```java
import java.util.Scanner;

public class Main1 {
    // 回傳陣列為 [最大值, 最小值]
    public static int[] search(int[] arr, int start, int end) {
        // 治：若只有一個資料，則該資料即為最大、最小值
        if (start == end) return new int[]{arr[start], arr[start]};
        // 分：將資料分成兩組；治：找出拆分資料的最大、最小值
        int mid = start - (start - end) / 2;
        int[] left = search(arr, start, mid);
        int[] right = search(arr, mid + 1, end);
        // 合：比較兩組資料的最大、最小值，得出原資料的最大、最小值
        return new int[]{Math.max(left[0], right[0]), Math.min(left[1], right[1])};
    }

    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[] result = search(arr, 0, arr.length - 1);
        System.out.printf("max = %d, min = %d", result[0], result[1]);
    }
}
```

java

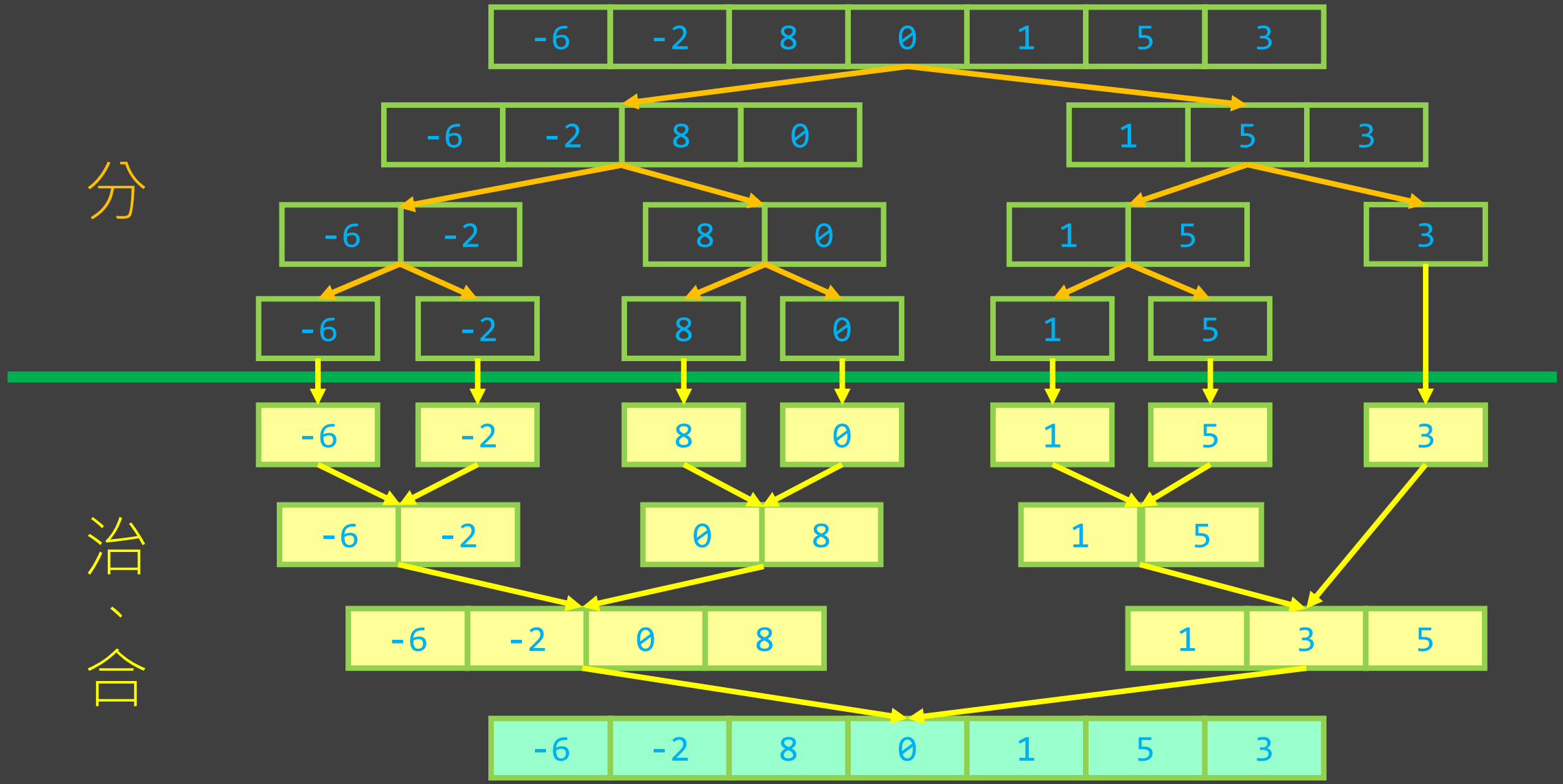
```
7
-2 -6 8 0 1 5 3
max = 8, min = -6
```

console

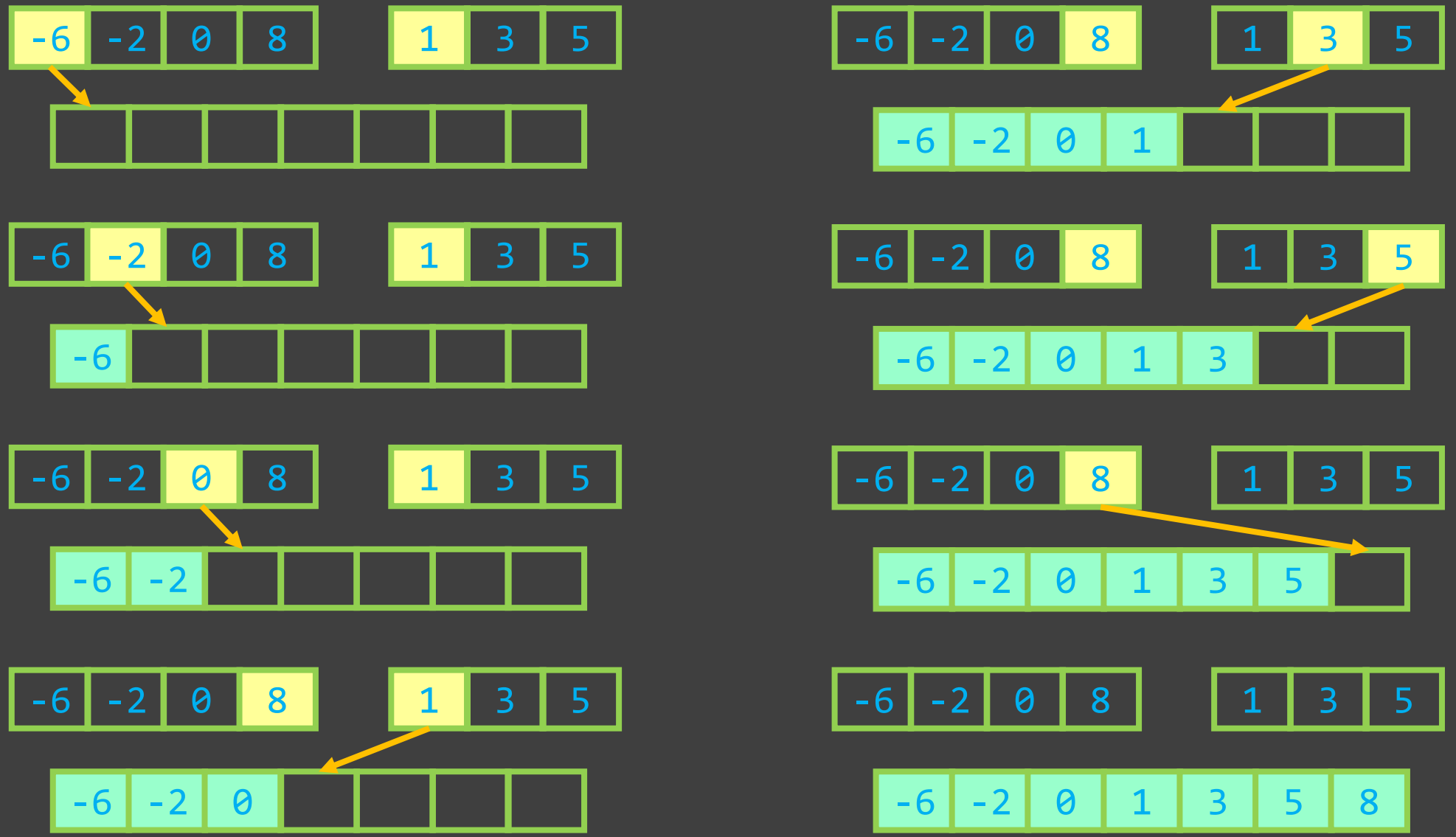
合併排序法

合併排序法(merge sort)是一種較為高效的排序演算法
時間複雜度為 $\Theta(n \log n)$ ，其使用了分治法：
分：將資料拆分為兩組
治：分別排序兩組資料
若只有一個資料則為排序完成，否則繼續分治直到可以排序
合：合併兩組有序資料
其中合併兩組有序資料的做法為：
比較兩組資料的最小值，將較小的放入新陣列
重複直到兩組資料皆被放入新陣列
需注意，此排序法空間複雜度為 $O(n)$，較其他常見排序法高

合併排序法


分
治、合
-6 -2 8 0 1 5 3
-6 -2 8 0
1 5 3
-6 -2
8 0
1 5
3
-6
-2
8
0
1
5
3
-6 -2
0 8
1 5
-6 -2 0 8
1 3 5
-6 -2 8 0 1 5 3

合併兩組有序資料


-6 -2 0 8
1 3 5
-6 -2 0 8
1 3 5
-6 -2 0 1
-6 -2 0 8
1 3 5
-6
-6 -2 0 8
1 3 5
-6 -2 0 1 3
-6 -2 0 8
1 3 5
-6 -2
-6 -2 0 8
1 3 5
-6 -2 0 1 3 5
-6 -2 0 8
1 3 5
-6 -2 0
-6 -2 0 8
1 3 5
-6 -2 0 1 3 5 8

合併排序法

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

public class Main1 {
    // 合併排序法
    private static void mergeSort(int[] arr, int start, int end) {
        if (start == end) return; // 治：若只有一個資料則為排序完成
        // 分：將資料拆分為兩組；治：分別排序兩組資料
        int mid = start - (start - end) / 2;
        mergeSort(arr, start, mid);
        mergeSort(arr, mid + 1, end);
        // 合：合併兩組有序資料
        int[] tmp = new int[end - start + 1];
        int lIndex = start, rIndex = mid + 1, i = 0;
        while (lIndex <= mid && rIndex <= end) tmp[i++] = arr[lIndex] < arr[rIndex] ? arr[lIndex++] : arr[rIndex++];
        while (lIndex <= mid) tmp[i++] = arr[lIndex++];
        while (rIndex <= end) tmp[i++] = arr[rIndex++];
        for (i = 0; i < tmp.length; i++) arr[start + i] = tmp[i];
    }

    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
        mergeSort(arr, 0, arr.length - 1);
        System.out.println(Arrays.toString(arr));
    }
}
```

java

```
7
-6 -2 8 0 1 5 3
[-6, -2, 0, 1, 3, 5, 8]
```

console

快速排序法

快速排序法(quick sort)是一種較為高效的排序演算法
許多程式語言的內建函式庫也是使用快速排序法或其變種
平均時間複雜度為 $\Theta(n \log n)$、最壞時間複雜度為 $\Theta(n^2)$
快速排序法為不穩定排序

快速排序法

快速排序法使用了分治法：

分：選定一個資料作為基準(pivot)

將比基準小的資料分為一組，基準外的其餘資料分為另一組

治：分別排序兩組資料

若只有一個資料則為排序完成，否則繼續分治直到可以排序

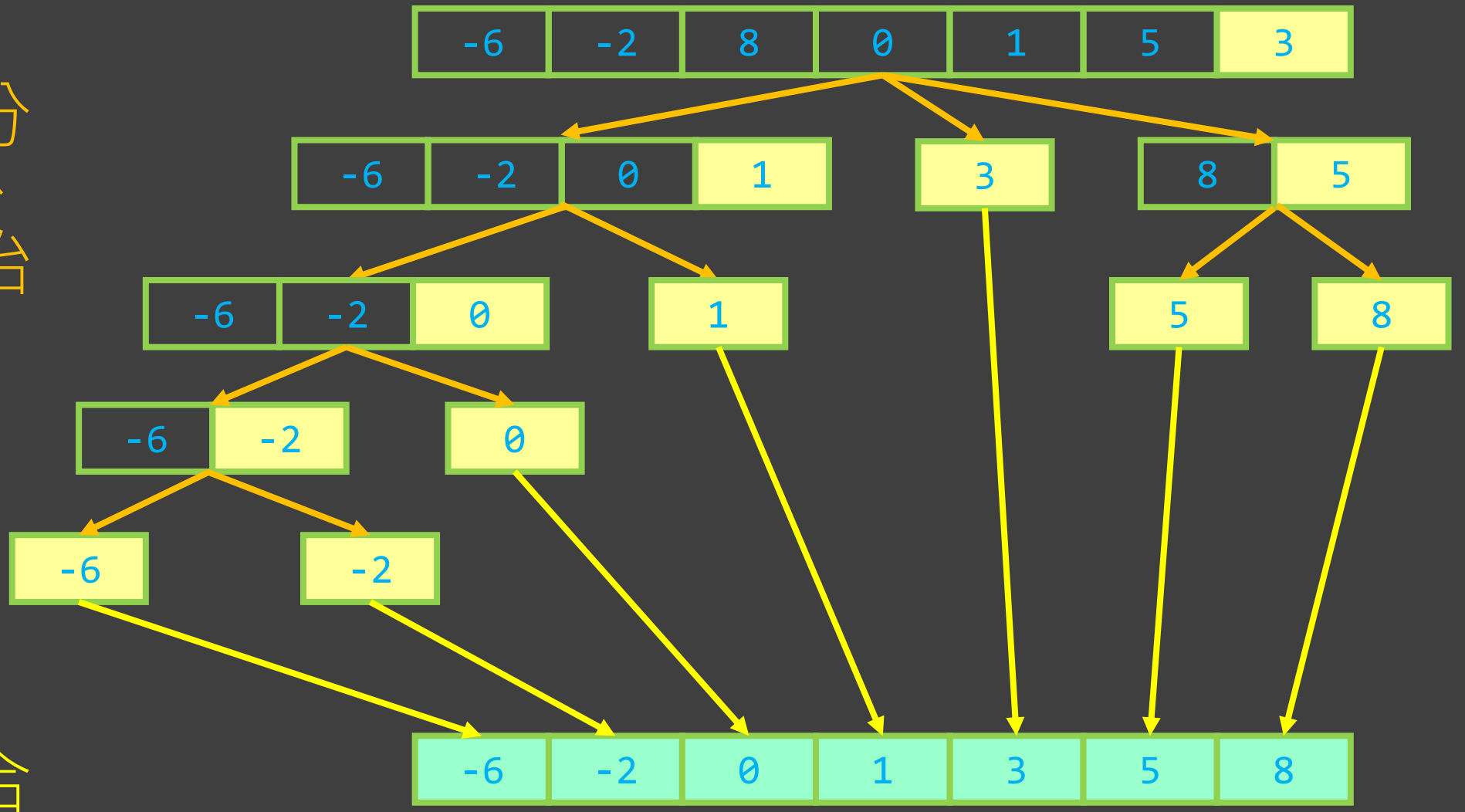
合：將比基準小的資料、基準、其餘資料依序連接在一起

在就地將資料分組時會進行以下步驟：

1. 若基準不是陣列的最後一個資料，將兩者交換
2. 將所有小於基準的資料與前方的資料交換

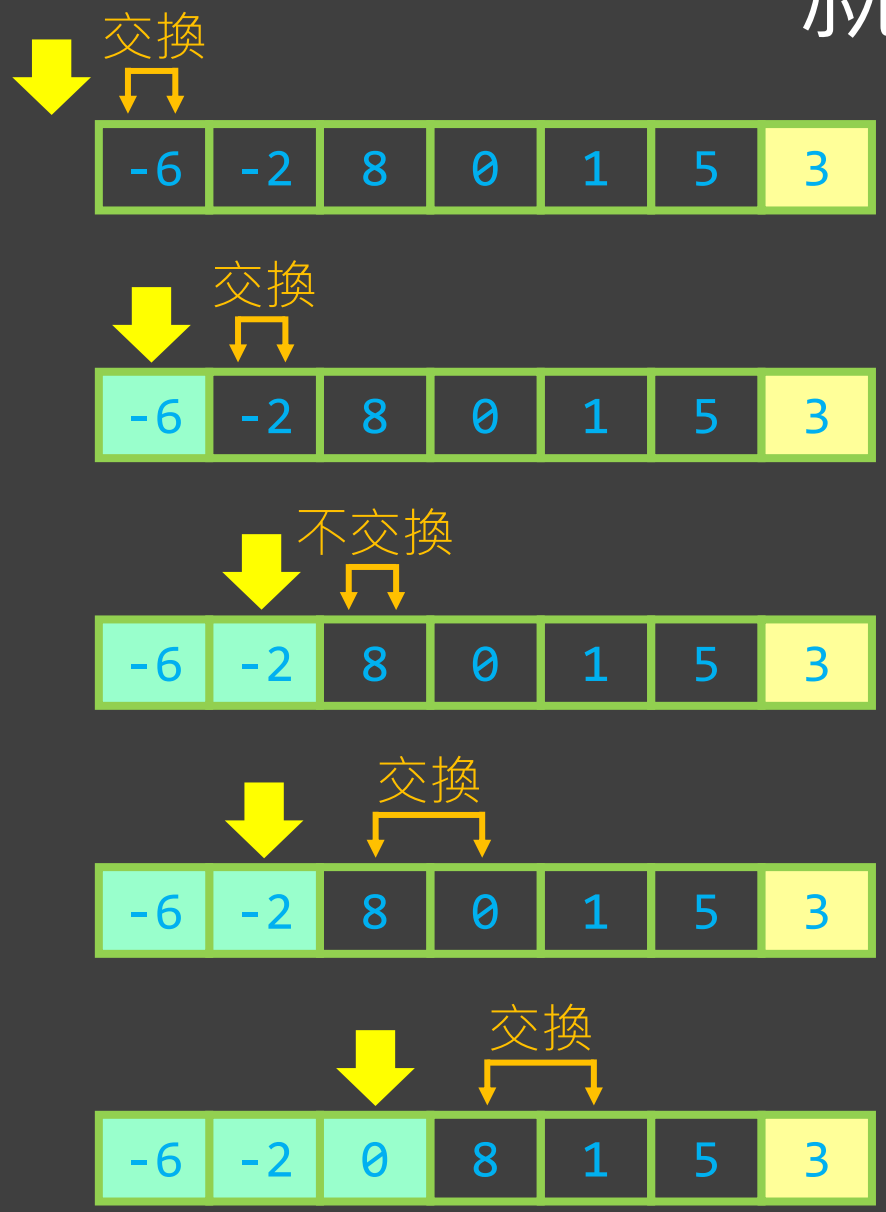
快速排序法

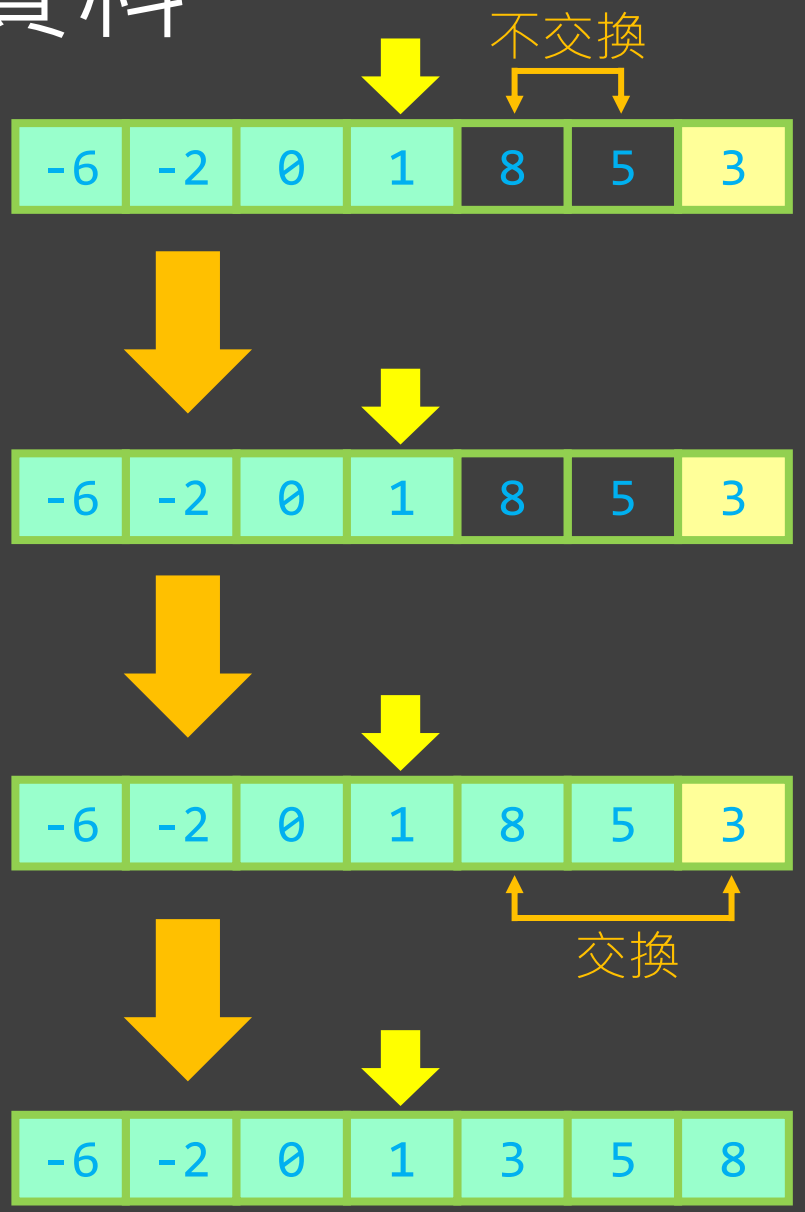
分、治



合

就地分組資料


交換
-6 -2 8 0 1 5 3
交換
-6 -2 8 0 1 5 3
不交換
-6 -2 8 0 1 5 3
交換
-6 -2 8 0 1 5 3
交換
-6 -2 0 8 1 5 3


不交換
-6 -2 0 1 8 5 3
-6 -2 0 1 8 5 3
-6 -2 0 1 8 5 3
交換
-6 -2 0 1 3 5 8

快速排序法

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

public class Main2 {
    // 快速排序法
    private static void quickSort(int[] arr, int start, int end) {
        if (start >= end) return; // 治：若只有一個資料或沒有資料則為排序完成
        // 分：選定一個資料作為基準，將比基準小的資料分為一組，基準外的其餘資料分為另一組
        int pivot = arr[end];
        // 將所有小於基準的資料與前方的資料交換
        int lesserIndex = start - 1;
        for (int i = start; i <= end - 1; i++) {
            if (arr[i] < pivot) {
                lesserIndex++;
                int tmp = arr[lesserIndex];
                arr[lesserIndex] = arr[i];
                arr[i] = tmp;
            }
        }
        // 治：分別排序兩組資料；合：將比基準小的資料、基準、其餘資料依序連接在一起
        int tmp = arr[lesserIndex + 1];
        arr[lesserIndex + 1] = arr[end];
        arr[end] = tmp;
        quickSort(arr, start, lesserIndex);
        quickSort(arr, lesserIndex + 2, end);
    }

    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
        quickSort(arr, 0, arr.length - 1);
        System.out.println(Arrays.toString(arr));
    }
}
```

java

```
7
-6 -2 8 0 1 5 3
[-6, -2, 0, 1, 3, 5, 8]
```

console