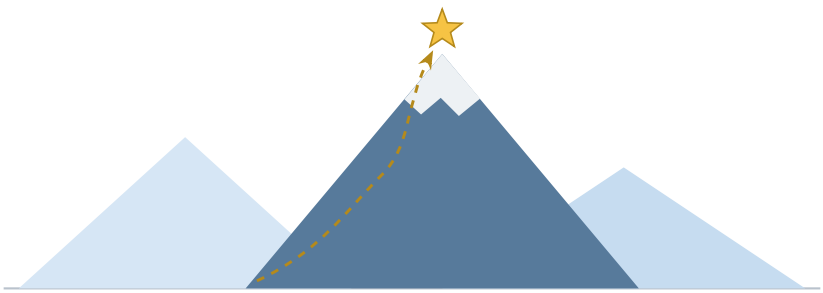
COMP1110/1140/6710 —Structured Programming

Java Lab 3：登山演算法

完全學習教材

2D 陣列 • 列舉（enum） • 貪婪爬坡 • 矩形二分搜尋



教材內容　觀念講解、圖解、逐步實作、完整測試、延伸挑戰

隨附程式　`Direction.java`、`Mountain.java`（含完整中文註解）

適用對象　已學過變數、迴圈、函式與一維陣列的 Java 初學者

依據課程講義「Lab 3: Mountain Climbing」編寫

目錄

1 這份教材要教你什麼 3

2 背景知識 A：二維陣列 3

2.1 從一維到二維：陣列的陣列 . . . 3
2.2 直接初始化：巢狀大括號 . . . 3
2.3 存取資料：`map[row][column]` . . . 4
2.4 用 y 與 x 思考：方向與鄰居 . . . 4
2.5 巡訪整張地圖：巢狀迴圈 . . . 5

3 背景知識 B：列舉（enum） 5

3.1 為什麼需要 enum？ . . . 6
3.2 本 Lab 的 `Direction` 列舉 . . . 6
3.3 enum 與 switch 是天生一對 . . . 6

4 背景知識 C：契約式思考——前置條件、後置條件與 assert 7

4.1 函式是一份契約 . . . 7
4.2 用 assert 把契約寫進程式裡 . . . 8

5 問題設定：高度圖與黃金山頂 8

5.1 規則整理 . . . 8
5.2 邊界處理的核心技巧：`heightAt` . . . 9

6 Activity 1：`lookAround`——觀察四周 9

6.1 規格 . . . 9
6.2 逐步推導 . . . 9
6.3 兩個值得注意的設計決定 . . . 10
6.4 測試策略 . . . 11

7 Activity 2：`climb`——一步一步爬到山頂 11

7.1 貪婪爬坡法 . . . 11
7.2 完整實作 . . . 12

8 Activity 3：`fastTravel`——矩形二分搜尋 13

8.1 動機：climb 太慢了 . . . 13
8.2 把二分搜尋升級到 2D . . . 13

8.3 完整實作 14
8.4 效能比較 16

9 測試工程：錐形地圖與窮舉驗證 16

10 編譯、執行與繳交 17
10.1 編譯與執行 17
10.2 用 git 繳交 17

11 延伸挑戰 18

A 完整程式碼：`Direction.java` 18

B 完整程式碼：`Mountain.java` 19

1　這份教材要教你什麼

想像你是一位在「數位山脈」中工作的測量員。你手上有一張**高度圖**（heightmap）：一個 2D 的整數格子，每個數值代表該座標的海拔高度。地圖裡藏著一個**黃金山頂**（golden peak）——全地圖的最高點。你的任務是寫出一系列演算法，讓程式自己「爬」上這座山：從一個格子移動到另一個格子、分析周圍的坡度來判斷哪個方向是上坡，最後還要最佳化路線，用最少的步數登頂。

本教材對應課程 Lab 3 的三個活動，完成後你將學會：

- **撰寫非平凡的陣列程式**：宣告、初始化、巡訪 2D 陣列，並安全地處理邊界。
- **使用列舉**（enumeration，`enum`）：用型別安全的方式表達「方向」這種有限的選項。
- **培養搜尋演算法的直覺**：從一步一步的貪婪爬坡，進化到對數時間的二分搜尋。
- **用前置條件、後置條件與 `assert` 寫出**「自我檢查」的程式，並做到完整的分支覆蓋測試。

如何使用本教材

第 2–4 節是背景知識；已經熟悉 2D 陣列與 enum 的讀者可以直接跳到第 5 節的問題設定。三個活動（第 6–8 節）請依序閱讀，因為 `climb` 和 `fastTravel` 都建立在 `lookAround` 之上。隨附的 `Mountain.java` 可以直接編譯執行（見第 10 節），完整原始碼收錄於附錄。

2　背景知識 A：二維陣列

2.1　從一維到二維：陣列的陣列

一維陣列 `int[]` 是「一排整數」。那要怎麼表示一張**表格**（例如高度圖）呢？Java 的答案很直接：**把「每一列」做成一個一維陣列，再用另一個陣列把所有列裝起來**。所以 `int[][]` 讀作「`int` 陣列 的 陣列」。

```
// 建立一張 24 列（rows）高、12 行（columns）寬的格子
int[][] heightmap = new int[24][12];
```

程式 1　宣告並建立一張 24 列 × 12 行的高度圖

有一點容易混淆：`new int[24][12]` 其實是「24 個 `int[12]`」——24 列，每列 12 個整數。Java 把維度寫成這個順序，是為了跟**索引的寫法**一致（見下文）。

2.2　直接初始化：巢狀大括號

從 Mountain Lab 產生器複製地圖資料時，數值會包在**巢狀大括號**裡；每一組內層大括號就是一個「水平的列」：

```
int[][] smallMap = {
    {10, 20, 10},   // Row 0（第 0 列）
```

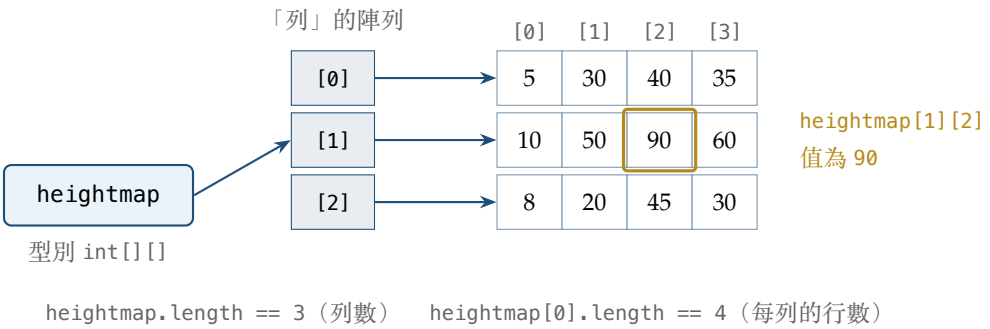


圖 1　2D 陣列的真面目：一個裝著「列」的陣列，每一列又是一個一維陣列。

```
    {20, 50, 20},   // Row 1（第 1 列）
    {10, 20, 10}    // Row 2（第 2 列）
};
```

程式 2　講義中的 3×3 範例地圖

2.3　存取資料：`map[row][column]`

要讀取某一格的高度，用兩組方括號：`map[row][column]`。

- 第一個數字是垂直位置：你在哪一列（row）。
- 第二個數字是水平位置：你在哪一行（column）。

⚠ Java 陣列從 0 開始編號

Java 的陣列是 0-indexed：地圖的左上角永遠是 `map[0][0]`，右下角是 `map[map.length - 1][map[0].length - 1]`。合法索引範圍是 $0 \le$ `row` $<$ `map.length`，超出範圍就會拋出 `ArrayIndexOutOfBoundsException`。

2.4　用 y 與 x 思考：方向與鄰居

在高度圖上導航時，我們習慣把兩個索引取名為 `y` 與 `x`：

- **列 (y)**：改變第一個索引，就是往北或南移動。
- **行 (x)**：改變第二個索引，就是往東或西移動。

站在 `[y][x]` 這一格想「看看四周」時，只要調整索引：

方向	意義	索引寫法	變化
北 North（上）	上一列	`map[y - 1][x]`	y 減 1
南 South（下）	下一列	`map[y + 1][x]`	y 加 1
東 East（右）	右一行	`map[y][x + 1]`	x 加 1
西 West（左）	左一行	`map[y][x - 1]`	x 減 1

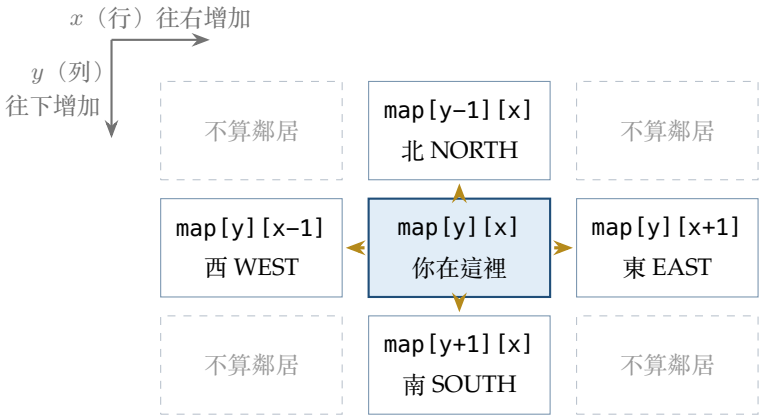


圖 2　站在 `[y][x]` 環顧四周。特別注意：y 往「下」增加，所以「北」是 $y-1$，跟數學課的 y 軸方向相反！

⚠ 最經典的錯誤：把 y 和 x 寫反

`map[y][x]` 的第一個索引是**列**（垂直、y），第二個才是**行**（水平、x）。跟平面座標 (x, y) 的習慣順序剛好**相反**。寫錯順序時，方形地圖可能「碰巧沒事」，換成長方形地圖就會當掉或算錯——這正是本 Lab 用 24×12 這種非正方形地圖的原因之一。

2.5　巡訪整張地圖：巢狀迴圈

處理 2D 陣列的標準寫法是「迴圈中的迴圈」——外層走過每一列，內層走過該列的每一行：

```
for (int y = 0; y < map.length; y++) {          // 每一列
    for (int x = 0; x < map[0].length; x++) {   // 該列的每一行
        System.out.print(map[y][x] + " ");
    }
    System.out.println();   // 換列時順便換行
}
```

程式 3　巡訪 2D 陣列的標準模式

💡 鋸齒狀陣列（ragged array）

因為 2D 陣列其實是「陣列的陣列」，Java 允許每一列的長度**不同**（例如三角形的資料）。本 Lab 的高度圖保證是矩形，但寫程式時仍應把「地圖是非空矩形」寫成**前置條件**明確檢查——我們稍後會用 `isRectangular` 這個輔助函式搭配 `assert` 來做這件事。

3　背景知識 B：列舉（enum）

3.1 為什麼需要 enum？

「方向」只有五種合法值：北、東、南、西、山頂。如果用整數表示（0 代表北、1 代表東……），會有兩個大問題：

1. **沒有型別保護**：`int dir = 7;` 完全合法，但 7 是什麼方向？程式進入無意義的狀態，編譯器卻不會阻止你。
2. **可讀性差**：看到 `if (dir == 2)`，你得去翻文件才知道 2 是南還是西。

`enum`（enumeration，列舉）一次解決兩個問題：它宣告一個新型別，而且這個型別只有你列出的那幾個值。

3.2 本 Lab 的 `Direction` 列舉

課程儲存庫已提供 `Direction.java`（**不可修改**），內容等價於：

```
public enum Direction {
    NORTH,   // 北：往上一列（y - 1）
    EAST,    // 東：往右一行（x + 1）
    SOUTH,   // 南：往下一列（y + 1）
    WEST,    // 西：往左一行（x - 1）
    PEAK     // 山頂：目前格子比四個鄰居都高
}
```

程式 4　`Direction` 列舉的核心宣告

每個列舉值（如 `Direction.NORTH`）都是 `Direction` 型別的**唯一物件**，所以可以放心用 `==` 比較：

```
Direction dir = lookAround(y, x, heightmap);
if (dir == Direction.PEAK) {
    System.out.println("登頂成功！");
}
```

只要你的 Java 檔跟 `Direction.java` 放在**同一個資料夾**，就能直接使用這個型別，不需要（也不應該）自己重新定義。

3.3 enum 與 switch 是天生一對

根據方向決定「怎麼移動」時，`switch` 比一長串 `if-else` 清楚得多。Java 14 之後更推薦**箭頭 switch**：

```
// 傳統寫法：每個 case 要自己 break，
// 忘了就會「掉落」到下一個 case（fall-through）——經典 bug！
```

```java
switch (dir) {
    case NORTH:
        y--;
        break;      // 沒寫 break 的話會繼續執行 case EAST！
    case EAST:
        x++;
        break;
    // ...
}

// 箭頭寫法（Java 14+）：每個分支獨立，不會掉落，推薦！
switch (dir) {
    case NORTH -> y--;   // 北：列索引減 1（往上）
    case EAST  -> x++;   // 東：行索引加 1（往右）
    case SOUTH -> y++;   // 南：列索引加 1（往下）
    case WEST  -> x--;   // 西：行索引減 1（往左）
    case PEAK  -> { }    // 什麼都不做
}
```

程式 5　傳統 switch（左）容易忘記 break；箭頭 switch（右）沒有這個陷阱

switch 的分支完整性

對 enum 使用箭頭 switch 時，如果漏列了某個列舉值又沒有 `default`，編譯器會直接報錯。這是 enum 的另一個好處：**編譯器幫你檢查有沒有漏掉情況**。

4　背景知識 C：契約式思考——前置條件、後置條件與 assert

4.1　函式是一份契約

一個設計良好的函式，就像一份**契約**（contract）：

- **前置條件**（precondition）：呼叫者必須保證的事。「你給我的參數必須長這樣，否則後果自負。」
- **後置條件**（postcondition）：函式必須保證的事。「只要你遵守前置條件，我保證回傳值一定滿足這些性質。」

例如 `lookAround(y, x, heightmap)` 的契約可以這樣寫：

`lookAround` 的契約

前置條件： (1) `heightmap` 是非空的矩形 2D 陣列；(2) $0 \le y <$ 列數，且 $0 \le x <$ 行數。
後置條件： 回傳 `PEAK` 時，目前格子嚴格高於四個鄰居；回傳其他方向時，該方向的鄰居高

度是四鄰之最大，且目前格子的高度 ≤ 該鄰居的高度。

4.2　用 assert 把契約寫進程式裡

Java 的 `assert` 語句讓契約「活」起來——條件不成立就立刻停下來：

```
assert 條件;                  // 條件為 false 時拋出 AssertionError
assert 條件 : "錯誤訊息";     // 加上說明訊息，除錯更快
```

程式 6　assert 的兩種寫法

⚠ assert 預設是「關閉」的！

執行 `java Mountain` 時，所有 `assert` 都會被**直接跳過**。必須加上 `-ea`（enable assertions）旗標：`java -ea Mountain`。忘記 `-ea` 是「測試明明全過、其實根本沒跑」的頭號原因。

偵測 assert 有沒有開啟的小技巧

下面這招利用「`assert` 沒開就不執行」的特性，在 `main` 一開始自我檢查（隨附程式就是這樣寫的）：

```
boolean assertionsEnabled = false;
assert assertionsEnabled = true;   // 有開啟才會執行這個賦值
if (!assertionsEnabled) {
    System.out.println("【警告】assert 未啟用！請改用：java -ea Mountain");
}
```

5　問題設定：高度圖與黃金山頂

5.1　規則整理

把講義的設定翻成精確的規格：

- 高度圖是 $m \times n$ 的 2D 整數陣列；`heightmap[y][x]` 是座標 (y, x) 的海拔。
- 地圖上有唯一的最高點（黃金山頂）。
- Mountain Lab 產生的地圖保證：**局部最佳就是全域最佳**——在任何一格往「最高的鄰居」走，都會嚴格更接近山頂。（這個保證是 `climb` 與 `fastTravel` 正確性的基石！）
- **地圖邊界之外的高度視為 0 或某個非常低的數字。**

5.2　邊界處理的核心技巧：heightAt

「山頂在角落」時，某些鄰居根本不存在；直接讀 map[-1][0] 會當掉。與其在主邏輯裡寫四段重複的邊界判斷，不如把「安全讀取」包成一個小函式——這就是**抽象化**的威力：

```java
/** 「界外」的高度值：選 int 的最小值，約 -21 億。 */
static final int OUT_OF_BOUNDS = Integer.MIN_VALUE;

/** 安全讀取 (y, x) 的高度；越界時回傳 OUT_OF_BOUNDS。 */
static int heightAt(int y, int x, int[][] heightmap) {
    boolean insideRows = 0 <= y && y < heightmap.length;
    boolean insideCols = 0 <= x && x < heightmap[0].length;
    if (!insideRows || !insideCols) {
        return OUT_OF_BOUNDS;   // 界外：回傳「超級低」的高度
    }
    return heightmap[y][x];     // 界內：正常讀取
}
```

程式 7　heightAt：越界一律回報「超級低」

為什麼選 Integer.MIN_VALUE 而不是 0？

講義允許「0 或某個非常低的數字」。但如果地圖上真的出現高度 0（甚至負數，例如海溝），用 0 當界外值就可能把「界外」誤認成最高的鄰居。Integer.MIN_VALUE 是 int 能表示的最小值，**絕不可能**比任何真實高度大，因此永遠不會勝出。這種「哨兵值」（sentinel value）的選擇是防禦式程式設計的常見考量。

6　Activity 1：lookAround——觀察四周

6.1　規格

```java
Direction lookAround(int y, int x, int[][] heightmap)
```

比較 (y, x) 與北、南、東、西四個鄰居的高度，回傳 **最高鄰居所在的方向**（NORTH / EAST / SOUTH / WEST）；若目前格子比所有鄰居都高，回傳 PEAK。

6.2　逐步推導

整個函式其實只有四個步驟：

1. **檢查前置條件**：地圖是非空矩形、(y, x) 在圖內。
2. **讀取五個高度**：自己一個、鄰居四個——用 heightAt，邊界自動安全。

3. **找出四鄰的最大值**：Math.max 一次比兩個，巢狀呼叫三次就能比出四個值的最大。
4. **判斷**：自己比最大鄰居還高 ⇒ PEAK；否則回傳最大鄰居的方向。

```java
static Direction lookAround(int y, int x, int[][] heightmap) {
    // ---- 前置條件檢查（執行時要加 -ea 才會生效）----
    assert isRectangular(heightmap)
            : "前置條件違反：heightmap 必須是非空的矩形陣列";
    assert 0 <= y && y < heightmap.length
            : "前置條件違反：y=" + y + " 超出列範圍";
    assert 0 <= x && x < heightmap[0].length
            : "前置條件違反：x=" + x + " 超出行範圍";

    // ---- 讀取自己與四個鄰居的高度 ----
    int current = heightmap[y][x];                  // 腳下的高度
    int north   = heightAt(y - 1, x, heightmap);    // 北：上一列
    int east    = heightAt(y, x + 1, heightmap);    // 東：右一行
    int south   = heightAt(y + 1, x, heightmap);    // 南：下一列
    int west    = heightAt(y, x - 1, heightmap);    // 西：左一行

    // ---- 找出四個鄰居中的最大高度 ----
    int best = Math.max(Math.max(north, east), Math.max(south, west));

    // ---- 判斷結果 ----
    if (current > best) {
        return Direction.PEAK;      // 我比所有鄰居都高：山頂！
    }
    // 依「北 → 東 → 南 → 西」的優先順序，回傳第一個最高的鄰居
    if (north == best) return Direction.NORTH;
    if (east == best)  return Direction.EAST;
    if (south == best) return Direction.SOUTH;
    return Direction.WEST;          // 剩下的只可能是西
}
```

程式 8　lookAround 完整實作（節錄自 Mountain.java）

6.3　兩個值得注意的設計決定

(一)「嚴格更高」才算山頂。判斷式是 current > best 而不是 >=。若自己跟最高鄰居一樣高，我們選擇回傳那個鄰居的方向，而不是宣稱自己是山頂——因為山頂的定義是「比鄰居都高」。

(二) 打平（tie）時行為要明確。若兩個鄰居同為最高（例如講義 3×3 地圖的角落，東、南同為 20），程式依「北 → 東 → 南 → 西」的順序取第一個。Mountain Lab 的有效地圖沿途不會出現這種情況，但讓程式行為明確且可預測永遠是好習慣——這也讓我們的測試可以放心斷言結

果。

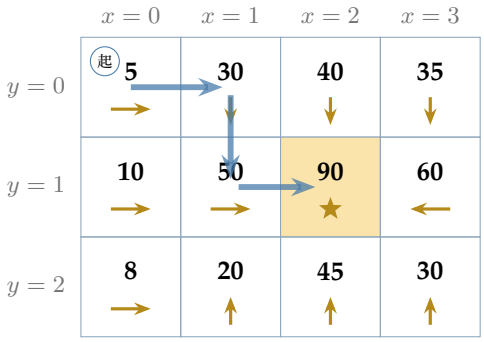


圖 3　3 × 4 測試地圖：每一格的小箭頭是 `lookAround` 的回傳方向（星號 = `PEAK`），對應 Mountain Lab 的「flow arrows」功能。粗藍色箭頭是第 7 節 `climb` 從 `[0][0]` 出發的實際路線（3 步登頂）。

6.4　測試策略

Activity 1 的測試清單

講義要求在 `main` 用 `assert` 徹底測試，目標是**完整分支覆蓋**——五種 `Direction` 值每一種都至少被回傳一次，並特別關照**陣列邊緣**的座標。隨附 `Mountain.java` 的做法：

1. **講義範例圖**（3 × 3）：山頂、四個邊、以及「打平」的角落。
2. **手工設計圖**（3 × 4，圖 3）：鄰居高度全部相異，12 格逐一斷言，一張圖就覆蓋全部五種回傳值與四條邊。
3. **退化地圖**：1 × 1（四鄰全是界外）、單列 1 × 5、單行 5 × 1。
4. **窮舉測試**：程式產生「錐形地圖」（見第 9 節），讓山頂輪流放在每個位置，再對每一格比對理論答案——共 87,667 筆。

7　Activity 2：`climb`——一步一步爬到山頂

7.1　貪婪爬坡法

```
1 int[] climb(int[][] heightmap)
```

策略非常直觀，叫做**貪婪爬坡法**（greedy hill climbing）：

1. 站在 `[0][0]`。
2. `lookAround` 看看四周。
3. 是 `PEAK`？結束！否則往回報的方向走一步，回到步驟 2。

「貪婪」的意思是：每一步只看眼前哪邊最高就往哪邊走，從不回頭、也不做長遠規劃。一

般的地形上貪婪法可能被「假山頂」（局部最高點）騙住；但 Mountain Lab 的地圖保證局部最佳就是全域最佳，所以貪婪法在這裡必定成功。

為什麼一定會停？——迴圈的終止性

在有效地圖上，每走一步腳下的高度都嚴格上升（走向的是「比自己高」的最高鄰居）。高度不可能無限上升（格子數量有限），所以走的步數不可能超過格子總數，迴圈保證終止。隨附程式還把這個推理寫成防禦性 **assert**：一旦步數超過 $m \times n$，代表地圖違反前置條件（例如出現同高的「高原」讓你原地打轉），與其無窮迴圈，不如立刻失敗、給出清楚的錯誤訊息。

7.2 完整實作

```
static int[] climb(int[][] heightmap) {
    assert isRectangular(heightmap)
            : "前置條件違反：heightmap 必須是非空的矩形陣列";

    int y = 0;          // 目前位置：從左上角 [0][0] 出發
    int x = 0;
    int steps = 0;      // 已經走了幾步

    Direction dir = lookAround(y, x, heightmap);
    while (dir != Direction.PEAK) {       // 還不是山頂就繼續走
        switch (dir) {                    // 依方向更新座標
            case NORTH -> y--;            // 北：往上一列
            case EAST  -> x++;            // 東：往右一行
            case SOUTH -> y++;            // 南：往下一列
            case WEST  -> x--;            // 西：往左一行
            case PEAK  -> { }             // 不可能（while 已排除）
        }
        steps++;

        // 防禦性檢查：步數絕不可能超過格子總數
        assert steps <= heightmap.length * heightmap[0].length
                : "步數超過格子總數：這張地圖大概不是有效的單峰地圖";

        dir = lookAround(y, x, heightmap);   // 到新位置再看一次
    }

    // 依講義要求：印出總步數
    System.out.println("'climb' found the peak in " + steps + " steps!");

    // 後置條件：回傳的座標必須真的是山頂
```

```
    assert lookAround(y, x, heightmap) == Direction.PEAK
            : "後置條件違反：climb 停下來的位置不是山頂";

    return new int[]{y, x};   // 打包成長度 2 的新陣列 {y, x}
}
```

程式 9　climb 完整實作（節錄自 Mountain.java）

幾個細節值得留意：

- 回傳值是「新建的長度 2 陣列」new int[]{y, x}——先 y 後 x，跟索引順序一致。
- 步數訊息的格式依講義：'climb' found the peak in 39 steps!。
- 山頂如果就在起點（[0][0] 本身最高），迴圈一次都不會執行，印出 0 步——這是重要的邊界情況。

Activity 2 的測試清單

1. 山頂在起點：期望 0 步、回傳 {0, 0}。
2. 山頂在對角（離起點最遠）與正中央。
3. 步數驗證：在有效地圖上每步都嚴格接近山頂，所以步數應該恰好等於起點到山頂的曼哈頓距離 $|y_{peak}| + |x_{peak}|$——這是比「答案對不對」更強的檢查。
4. 比較 int[] 要用 Arrays.equals(a, b)；寫 a == b 比的是「是不是同一個物件」，幾乎一定是 false！

8　Activity 3：fastTravel——矩形二分搜尋

8.1　動機：climb 太慢了

climb 可靠但緩慢：$m \times n$ 的地圖上，最壞要走 $(m-1)+(n-1)$ 步（起點在一角、山頂在對角）。1024×1024 的地圖就是最多 2046 步。能不能不要一格一格走，而是「跳」著找？

回想一維二分搜尋：在排好序的陣列裡找目標時，每次探測中間元素，就能丟掉一半的搜尋範圍。關鍵在於：中間元素的資訊告訴我們「目標一定不在哪半邊」。

8.2　把二分搜尋升級到 2D

在高度圖上，lookAround 就扮演這個「指路」的角色。我們隨時維護一個搜尋矩形 [top..bottom]×[left..right]，並維持一個迴圈不變量（loop invariant）：山頂一定在矩形裡。每一輪：

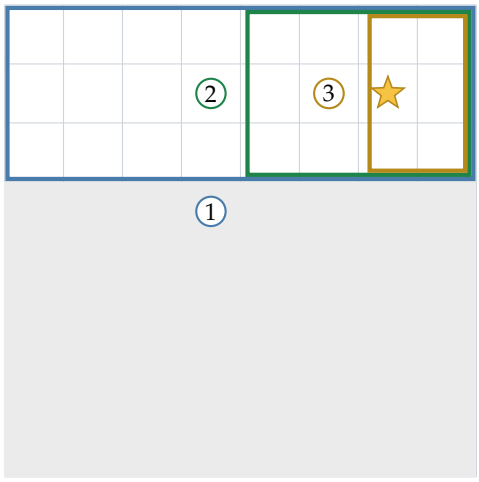
1. 取矩形中心點 (midY, midX)，呼叫 lookAround。
2. 回傳 PEAK ⇒ 找到了，結束。

3. 否則依方向砍掉一半：

lookAround 回傳	推論	矩形更新
NORTH	山頂的列 $<$ midY	bottom = midY − 1
SOUTH	山頂的列 $>$ midY	top = midY + 1
EAST	山頂的行 $>$ midX	left = midX + 1
WEST	山頂的行 $<$ midX	right = midX − 1

為什麼「方向」能拿來砍範圍？——正確性論證

有效地圖保證：lookAround 給的方向讓你 嚴格更接近山頂。假設中心點得到 NORTH，表示「往北走一步會更接近山頂」。如果山頂的列號 $\geq$ midY（在同一列或更南邊），往北走只會拉遠或不變與山頂的垂直距離——矛盾！所以山頂的列號必定 $<$ midY，中心點那一列以南（含該列）可以整片安全地砍掉。其他三個方向同理。注意：一次探測只縮小一個維度（南北向或東西向其中之一），這正是步數上限是 $\lceil \log_2 m \rceil + \lceil \log_2 n \rceil + 1$ 而不是 max 的原因。



探測 ① 在 $(3,3)$ 得到 NORTH
⇒ 只留上面 3 列（藍框）

探測 ② 在 $(1,3)$ 得到 EAST
⇒ 只留右半（綠框）

探測 ③ 在 $(1,5)$ 得到 EAST
⇒ 再砍一半（金框）

探測 4 在 $(1,6)$ 得到 PEAK，完成！

圖 4　fastTravel 在 8×8 地圖（山頂在 $(1,6)$）上的搜尋過程：4 次探測登頂。灰色區域是第一次探測後就再也不用考慮的部分。同一張圖 climb 要走 $1+6=7$ 步。

8.3 完整實作

```
static int[] fastTravel(int[][] heightmap) {
    assert isRectangular(heightmap)
            : "前置條件違反：heightmap 必須是非空的矩形陣列";

    // 搜尋範圍：一開始是整張地圖（都是「含」邊界的索引）
    int top    = 0;
    int bottom = heightmap.length - 1;
```

```
    int left   = 0;
    int right  = heightmap[0].length - 1;

    int steps = 0;   // 探測（probe）了幾次

    while (true) {
        // 取搜尋矩形的中心點（整數除法自動向下取整）
        int midY = (top + bottom) / 2;
        int midX = (left + right) / 2;

        steps++;
        Direction dir = lookAround(midY, midX, heightmap);

        if (dir == Direction.PEAK) {
            System.out.println(
                    "'fastTravel' found the peak in " + steps + " steps!");
            return new int[]{midY, midX};
        }

        // 根據方向，砍掉「山頂不可能在」的那半邊
        switch (dir) {
            case NORTH -> bottom = midY - 1;  // 山頂在更北：往上縮
            case SOUTH -> top    = midY + 1;  // 山頂在更南：往下縮
            case EAST  -> left   = midX + 1;  // 山頂在更東：往右縮
            case WEST  -> right  = midX - 1;  // 山頂在更西：往左縮
            case PEAK  -> { }                 // 不可能（上面已 return）
        }

        // 迴圈不變量：山頂永遠在 [top..bottom] x [left..right] 裡
        assert top <= bottom && left <= right
                : "搜尋矩形變成空的：這張地圖大概不是有效的單峰地圖";
    }
}
```

程式 10　fastTravel 完整實作（節錄自 Mountain.java）

迴圈不變量是二分搜尋的靈魂

寫任何二分搜尋，先想清楚不變量：「答案永遠在目前範圍裡」。每次更新範圍前問自己：這樣砍，會不會把答案砍掉？砍完範圍有沒有嚴格變小（避免無窮迴圈）？上面的程式把不變量直接寫成 assert——萬一地圖無效（不滿足前置條件），程式會立刻告訴你，而不是默默跑出錯誤答案。

8.4 效能比較

	climb（逐步爬坡）	**fastTravel（矩形二分）**
策略	每步走向最高的鄰居	每次探測搜尋矩形的中心
一次動作獲得的資訊	前進一格	排除約一半的搜尋範圍
步數量級（$m \times n$ 地圖）	$O(m+n)$，最壞 $(m-1)+(n-1)$ 步	$O(\log m + \log n)$，至多 $\lceil \log_2 m \rceil + \lceil \log_2 n \rceil + 1$ 次
24×12、山頂在對角	34 步	8 次探測
128×128、山頂在 $(100, 111)$	211 步	9 次探測
1024×1024（估算）	最多 2046 步	約 21 次探測

表中 24×12 與 128×128 兩列是隨附程式的實測結果（見下一節的執行輸出）。地圖越大，差距越驚人——這就是**對數時間**演算法的威力。

9 測試工程：錐形地圖與窮舉驗證

手寫測試地圖很快就會遇到瓶頸：想測「山頂在任何位置都正確」，總不能手刻幾百張地圖。隨附程式用了一個小小的**測試工程技巧**——寫一個**地圖產生器**，再配一個**理論答案公式**：

```java
/** 以 (peakY, peakX) 為山頂的「錐形」地圖：高度隨距離遞減。 */
static int[][] coneMap(int rows, int cols, int peakY, int peakX) {
    int[][] map = new int[rows][cols];
    int base = 3 * rows + 2 * cols;   // 讓所有高度都是正數
    for (int y = 0; y < rows; y++) {
        for (int x = 0; x < cols; x++) {
            map[y][x] = base - 3 * Math.abs(y - peakY)
                             - 2 * Math.abs(x - peakX);
        }
    }
    return map;
}
```

程式 11　錐形測試地圖：答案可以用公式算出來

兩個方向的權重故意取**不同值**（3 與 2），這樣四個鄰居永遠不會同高、每一格的正確答案唯一，而且有封閉公式：列還沒對齊山頂時答案必是 `NORTH/SOUTH`（權重 3 壓過 2），列對齊後看東西，兩者都對齊就是 `PEAK`。於是可以**窮舉**——每種地圖尺寸 × 每個山頂位置 × 每個座標——逐一比對 `lookAround` 的輸出，一口氣驗證 87,667 筆。同一批地圖也拿來系統化測試 `climb`（斷言步數恰等於曼哈頓距離）與 `fastTravel`（斷言探測數不超過理論上限）。

實際執行 `java -ea Mountain` 的輸出節錄：

```
=== Lab 3: Mountain Climbing ── 完整測試 ===
```

```
[1] lookAround：3x3 講義範例地圖 ... 通過
[2] lookAround：3x4 地圖 12 格全驗證（分支覆蓋）... 通過
[3] lookAround：1x1／單列／單行 退化地圖 ... 通過
[4] lookAround：錐形地圖窮舉測試，共比對 87667 個座標 ... 通過
[5] climb：手工地圖測試
'climb' found the peak in 0 steps!
'climb' found the peak in 4 steps!
    ...
    地圖 24x12，山頂在 (23,11)：
      'climb' found the peak in 34 steps!
      'fastTravel' found the peak in 8 steps!
    ...
[7] 步數比較：128x128 地圖，山頂在 (100, 111)
    'climb' found the peak in 211 steps!
    'fastTravel' found the peak in 9 steps!
    同一座山，climb 走了 211 步；fastTravel 只探測 9 次！

=== 全部測試通過！ ===
```

10 編譯、執行與繳交

10.1 編譯與執行

把 `Direction.java` 與 `Mountain.java` 放在同一個資料夾，然後：

```
$ javac Direction.java Mountain.java   # 編譯（產生 .class 檔）
$ java -ea Mountain                    # 執行，-ea 開啟 assert
```

⚠ 再提醒一次

沒有 `-ea`，所有 `assert` 都不會執行——測試「看起來全過」，其實根本沒跑。隨附程式在 `main` 開頭會自動偵測並警告這種情況。

10.2 用 git 繳交

依照講義，完成後記得 add、commit、push：

```
$ git add Mountain.java                    # 或你的檔案名稱
$ git commit -m "Completed mountain lab"
$ git push
```

離開實驗課之前，確認你已經跟助教討論過你的程式，而且所有工作都已經 commit 並

push 到 GitLab！

11 延伸挑戰

以下挑戰不在 Lab 要求範圍內，但能大幅加深理解：

1. **流向圖產生器**：寫 `printFlowMap(int[][] heightmap)`，每格印出一個字元（`^`、`>`、`v`、`<`、`P`）代表 `lookAround` 的結果，重現圖 3 的效果。
2. **記錄完整路徑**：讓 `climb` 除了山頂座標外，也記下沿途經過的每個座標（提示：使用 `ArrayList<int[]>`，在 `java.util` 套件裡）。
3. **高原問題**：手工造一張含相鄰同高格子的地圖，觀察 `climb` 的防禦性 `assert` 如何避免無窮迴圈。想想：如果把 `lookAround` 的判斷改成「不嚴格更高就算 `PEAK`」，會有什麼新問題？
4. **多峰地圖**：造一張有兩個局部高點的地圖。`climb` 會發生什麼事？`fastTravel` 呢？哪一個的錯誤比較「安靜」（不容易被發現）？
5. **證明步數上限**：解釋為什麼 `fastTravel` 的探測次數不超過 $\lceil \log_2 m \rceil + \lceil \log_2 n \rceil + 1$。（提示：每次非 `PEAK` 探測，某一個維度的範圍長度至少減半。）
6. **八方向擴充**：若允許斜走（東北、西北……），`Direction` 的五個值就不夠用了。在不修改 `Direction.java` 的前提下，設計你自己的 `Direction8` 列舉並改寫三個函式。

A 完整程式碼：`Direction.java`

```
/**
 * Direction.java ——— 方向列舉（enum）
 * Lab 3: Mountain Climbing（登山實驗）
 *
 * 這個檔案定義了在高度圖（heightmap）上移動時使用的五個「方向值」。
 *
 * 【重要】在課程的原始儲存庫中，這個檔案已經提供，
 * 而且講義明確要求「不要修改這個檔案」。
 * 這裡為了教材的完整性重新列出，並補上教學用註解。
 *
 * ——————————————————————————————————————
 * 為什麼要用 enum，而不是用 int 常數（0 代表北、1 代表東……）？
 * ——————————————————————————————————————
 * 1. 型別安全（type safety）：
 *    編譯器保證 Direction 型別的變數只可能是這五個值之一，
 *    絕對不會出現 dir = 7 這種毫無意義的狀態。
 *
 * 2. 可讀性：
 *    Direction.NORTH 一看就懂；數字 0 則要查文件才知道意思。
```

```
 *
 * 3. 搭配 switch 使用時，若漏寫某個分支，
 *    現代的 switch 運算式（Java 14+）會直接編譯錯誤，幫你抓漏。
 *
 * enum 的每個值（NORTH、EAST……）其實都是 Direction 型別的
 * 「唯一物件」，可以直接用 == 比較，也可以放進 switch。
 */
public enum Direction {

    /** 北（North）：往「上」一列移動，也就是列索引減 1（y - 1）。 */
    NORTH,

    /** 東（East）：往「右」一行移動，也就是行索引加 1（x + 1）。 */
    EAST,

    /** 南（South）：往「下」一列移動，也就是列索引加 1（y + 1）。 */
    SOUTH,

    /** 西（West）：往「左」一行移動，也就是行索引減 1（x - 1）。 */
    WEST,

    /**
     * 山頂（Peak）：目前所在的格子比東西南北四個鄰居都「嚴格更高」，
     * 不需要再移動了——你已經站在山頂上！
     */
    PEAK
}
```

程式 12 Direction.java 完整內容

B 完整程式碼：Mountain.java

```
import java.util.Arrays;
import java.util.HashSet;

/**
 * Mountain.java —— Lab 3: Mountain Climbing（登山實驗）完整參考實作
 * 課程：COMP1110/1140/6710 — Structured Programming
 *
 * ─────────────────────────────────────────────
 * 情境說明
 * ─────────────────────────────────────────────
 * 你是一位在「數位山脈」中工作的測量員。
 * 地形用一張高度圖（heightmap）表示：一個 2D 整數陣列，
 * heightmap[y][x] 就是座標 (y, x) 的海拔高度。
 * 地圖中存在唯一的「黃金山頂」：全地圖的最高點。
```

```
 *
 * 本檔案依照講義完成三個活動：
 *   Activity 1: lookAround ——— 觀察四周，回傳最高鄰居的方向
 *   Activity 2: climb      ——— 從 [0][0] 一步一步爬到山頂
 *   Activity 3: fastTravel ——— 用「矩形二分搜尋」快速找到山頂
 *
 * ————————————————————————————————————————
 * 編譯與執行（-ea 會啟用 assert，測試才會真正生效！）
 * ————————————————————————————————————————
 *   javac Direction.java Mountain.java
 *   java -ea Mountain
 */
public class Mountain {

    // ================================================================
    // 常數與共用輔助函式
    // ================================================================

    /**
     * 「界外」的高度值。
     *
     * 講義說：「地圖邊界之外的高度，可以當成 0，
     * 或某個非常低的數字。」
     * 我們選擇 Integer.MIN_VALUE（int 的最小值，約 -21 億），
     * 這樣即使地圖上真的出現 0 或負數高度，
     * 界外也永遠不可能被誤認成「最高的鄰居」。
     */
    static final int OUT_OF_BOUNDS = Integer.MIN_VALUE;

    /**
     * 安全地讀取座標 (y, x) 的高度；
     * 若座標超出地圖範圍，回傳 OUT_OF_BOUNDS。
     *
     * 這個小函式是本作業「邊界處理」的核心技巧：
     * 與其在 lookAround 裡寫四段重複的 if 邊界判斷，
     * 不如把「安全讀取」包成一個函式，讓主要邏輯保持乾淨。
     * 這種把麻煩細節包起來的做法，叫做「抽象化」（abstraction）。
     *
     * 前置條件：heightmap 是非空的矩形陣列（見 isRectangular）。
     * 後置條件：
     *   - (y, x) 在地圖內 → 回傳 heightmap[y][x]
     *   - (y, x) 在地圖外 → 回傳 OUT_OF_BOUNDS
     */
    static int heightAt(int y, int x, int[][] heightmap) {
        // y 必須是合法的「列」索引：0 <= y < 列數
        boolean insideRows = 0 <= y && y < heightmap.length;
        // x 必須是合法的「行」索引：0 <= x < 行數
        boolean insideCols = 0 <= x && x < heightmap[0].length;
```

```
63
64          if (!insideRows || !insideCols) {
65              return OUT_OF_BOUNDS;   // 界外：回傳「超級低」的高度
66          }
67          return heightmap[y][x];     // 界內：正常讀取
68      }
69
70      /**
71       * 檢查 heightmap 是否為「非空的矩形」2D 陣列。
72       *
73       * Java 的 2D 陣列其實是「陣列的陣列」，
74       * 每一列的長度可以不同（稱為 ragged array，鋸齒狀陣列），
75       * 所以「是不是矩形」需要自己檢查。
76       * 這個函式只拿來寫前置條件的 assert 用。
77       */
78      static boolean isRectangular(int[][] map) {
79          if (map == null || map.length == 0) return false;      // 沒有任何列
80          if (map[0] == null || map[0].length == 0) return false; // 第一列是空的
81          for (int[] row : map) {
82              // 每一列都必須存在，而且長度要跟第一列一樣
83              if (row == null || row.length != map[0].length) return false;
84          }
85          return true;
86      }
87
88      // ================================================================
89      // Activity 1: lookAround ——— 觀察四周
90      // ================================================================
91
92      /**
93       * 觀察座標 (y, x) 的四個鄰居，決定該往哪個方向走。
94       *
95       * 規則：
96       *   - 比較北、東、南、西四個鄰居的高度，
97       *     回傳「最高鄰居」所在的方向。
98       *   - 如果目前的格子比四個鄰居都「嚴格更高」，
99       *     回傳 Direction.PEAK（你就是山頂）。
100      *   - 超出地圖邊界的鄰居視為高度 OUT_OF_BOUNDS（超級低），
101      *     所以永遠不會勝出。
102      *
103      * 打平規則（tie-breaking）：
104      *   若有多個鄰居同為最高，依 北 → 東 → 南 → 西 的順序取第一個。
105      *   （Mountain Lab 產生的有效地圖，沿途不會出現這種情形，
106      *   但寫程式時仍要讓行為「明確、可預測」。）
107      *
108      * 前置條件（precondition）：
109      *   1. heightmap 是非空的矩形 2D 陣列。
110      *   2. 0 <= y < 列數，0 <= x < 行數（座標必須在地圖內）。
```

```
111      *
112      * 後置條件（postcondition）：
113      *   - 回傳 PEAK 時：heightmap[y][x] 嚴格大於四個鄰居的高度。
114      *   - 回傳其他方向時：該方向鄰居的高度是四個鄰居中的最大值，
115      *     且目前格子的高度 <= 該鄰居的高度
116      *     （也就是：目前格子並沒有嚴格高於所有鄰居）。
117      */
118     static Direction lookAround(int y, int x, int[][] heightmap) {
119         // ---- 前置條件檢查（執行時要加 -ea 才會生效）----
120         assert isRectangular(heightmap)
121                 : "前置條件違反：heightmap 必須是非空的矩形陣列";
122         assert 0 <= y && y < heightmap.length
123                 : "前置條件違反：y=" + y + " 超出列範圍";
124         assert 0 <= x && x < heightmap[0].length
125                 : "前置條件違反：x=" + x + " 超出行範圍";
126 
127         // ---- 讀取自己與四個鄰居的高度 ----
128         int current = heightmap[y][x];                // 腳下的高度
129         int north   = heightAt(y - 1, x, heightmap);  // 北：上一列
130         int east    = heightAt(y, x + 1, heightmap);  // 東：右一行
131         int south   = heightAt(y + 1, x, heightmap);  // 南：下一列
132         int west    = heightAt(y, x - 1, heightmap);  // 西：左一行
133 
134         // ---- 找出四個鄰居中的最大高度 ----
135         // Math.max 一次只能比兩個數，所以用巢狀呼叫比出四個的最大值。
136         int best = Math.max(Math.max(north, east), Math.max(south, west));
137 
138         // ---- 判斷結果 ----
139         if (current > best) {
140             return Direction.PEAK;      // 我比所有鄰居都高：山頂！
141         }
142         // 依「北 → 東 → 南 → 西」的優先順序，回傳第一個最高的鄰居。
143         if (north == best) return Direction.NORTH;
144         if (east == best)  return Direction.EAST;
145         if (south == best) return Direction.SOUTH;
146         return Direction.WEST;          // 剩下的只可能是西
147     }
148 
149     // ================================================================
150     // Activity 2: climb ——— 一步一步爬到山頂
151     // ================================================================
152 
153     /** 僅供測試／示範用：記錄最近一次 climb 花了幾步。 */
154     static int lastClimbSteps = -1;
155 
156     /**
157      * 從 [0][0] 出發，每一步都走向「最高的鄰居」，
158      * 直到 lookAround 回報 PEAK 為止。這種策略叫做
```

```
     * 「貪婪爬坡法」（greedy hill climbing）。
     *
     * 為什麼這樣一定會到達山頂？
     * 因為 Mountain Lab 產生的地圖保證：
     * 「局部最佳就是全域最佳」——lookAround 給的方向
     * 永遠讓你嚴格更接近山頂，絕不會把你帶遠。
     * 每走一步，腳下的高度嚴格上升，所以不可能繞圈子，
     * 也不可能走回頭路，最後一定停在山頂。
     *
     * 前置條件：
     *   heightmap 是非空的矩形 2D 陣列，
     *   且為「有效地圖」：唯一山頂、沿途方向必定指向山頂。
     * 後置條件：
     *   回傳長度為 2 的新陣列 {y, x}：山頂的座標，
     *   且 lookAround(y, x, heightmap) == PEAK。
     *
     * @return 山頂座標 {y, x}
     */
    static int[] climb(int[][] heightmap) {
        // ---- 前置條件 ----
        assert isRectangular(heightmap)
                : "前置條件違反：heightmap 必須是非空的矩形陣列";

        int y = 0;          // 目前位置：從左上角出發
        int x = 0;
        int steps = 0;      // 已經走了幾步

        // 先看看四周；只要還不是山頂，就繼續走
        Direction dir = lookAround(y, x, heightmap);
        while (dir != Direction.PEAK) {
            // 依照方向更新座標。
            // 這裡用 Java 14+ 的「箭頭 switch」：
            // 每個分支互相獨立，不會像傳統 switch 忘了 break 而「掉落」。
            switch (dir) {
                case NORTH -> y--;   // 北：列索引減 1（往上）
                case EAST  -> x++;   // 東：行索引加 1（往右）
                case SOUTH -> y++;   // 南：列索引加 1（往下）
                case WEST  -> x--;   // 西：行索引減 1（往左）
                case PEAK  -> { }    // 不可能到這裡（while 已排除）
            }
            steps++;

            // 防禦性檢查：有效地圖上，路徑高度嚴格上升，
            // 走的步數絕不可能超過格子總數。
            // 若超過，代表地圖無效（例如有高原），與其無窮迴圈，
            // 不如讓 assert 立刻失敗、告訴我們哪裡出問題。
            assert steps <= heightmap.length * heightmap[0].length
                    : "步數超過格子總數：這張地圖大概不是有效的單峰地圖";
```

```
            dir = lookAround(y, x, heightmap);  // 到新位置再看一次四周
        }

        // 依講義要求：印出總步數
        System.out.println("'climb' found the peak in " + steps + " steps!");
        lastClimbSteps = steps;   // 記下來，讓 main 的測試可以驗證

        // ---- 後置條件：回傳的座標必須真的是山頂 ----
        assert lookAround(y, x, heightmap) == Direction.PEAK
                : "後置條件違反：climb 停下來的位置不是山頂";

        return new int[]{y, x};   // 打包成長度 2 的新陣列 {y, x}
    }

    // ================================================================
    // Activity 3: fastTravel ——— 矩形二分搜尋
    // ================================================================

    /** 僅供測試／示範用：記錄最近一次 fastTravel 探測了幾次。 */
    static int lastFastTravelSteps = -1;

    /**
     * 用「2D 版的二分搜尋」快速找到山頂。
     *
     * 核心想法：
     *   我們隨時維護一個「山頂必定在裡面」的矩形搜尋範圍
     *   [top..bottom] × [left..right]（含邊界）。
     *   每一輪：
     *     1. 取矩形的中心點 (midY, midX)，呼叫 lookAround。
     *     2. 若回傳 PEAK：直接找到山頂，結束。
     *     3. 否則，方向告訴我們山頂在中心點的哪一側：
     *        - NORTH → 山頂的列 <  midY，砍掉下半部（bottom = midY - 1）
     *        - SOUTH → 山頂的列 >  midY，砍掉上半部（top    = midY + 1）
     *        - EAST  → 山頂的行 >  midX，砍掉左半部（left   = midX + 1）
     *        - WEST  → 山頂的行 <  midX，砍掉右半部（right  = midX - 1）
     *   每一輪都把矩形的某一邊縮小約一半，
     *   所以步數是 O(log m + log n)，遠小於 climb 的 O(m + n)。
     *
     * 為什麼「方向」可以拿來砍範圍？
     *   有效地圖保證 lookAround 的方向讓你「嚴格更接近山頂」。
     *   如果往北走會更接近，山頂的列號必定小於現在的列號——
     *   所以山頂絕不可能在中心點那一列或更南邊，可以安心砍掉。
     *
     * 前置條件：同 climb。
     * 後置條件：同 climb ——— 回傳山頂座標 {y, x}。
     *
     * @return 山頂座標 {y, x}
```

```
255      */
256     static int[] fastTravel(int[][] heightmap) {
257         // ---- 前置條件 ----
258         assert isRectangular(heightmap)
259                 : "前置條件違反：heightmap 必須是非空的矩形陣列";
260 
261         // 搜尋範圍：一開始是整張地圖（注意都是「含」邊界的索引）
262         int top    = 0;                          // 最上面可能的列
263         int bottom = heightmap.length - 1;       // 最下面可能的列
264         int left   = 0;                          // 最左邊可能的行
265         int right  = heightmap[0].length - 1;    // 最右邊可能的行
266 
267         int steps = 0;   // 探測（probe）了幾次
268 
269         while (true) {
270             // 取搜尋矩形的中心點（整數除法自動向下取整）
271             int midY = (top + bottom) / 2;
272             int midX = (left + right) / 2;
273 
274             steps++;
275             Direction dir = lookAround(midY, midX, heightmap);
276 
277             if (dir == Direction.PEAK) {
278                 // 依講義要求：印出總步數
279                 System.out.println(
280                         "'fastTravel' found the peak in " + steps + " steps!");
281                 lastFastTravelSteps = steps;
282                 return new int[]{midY, midX};
283             }
284 
285             // 根據方向，砍掉「山頂不可能在」的那半邊
286             switch (dir) {
287                 case NORTH -> bottom = midY - 1;  // 山頂在更北：往上縮
288                 case SOUTH -> top    = midY + 1;  // 山頂在更南：往下縮
289                 case EAST  -> left   = midX + 1;  // 山頂在更東：往右縮
290                 case WEST  -> right  = midX - 1;  // 山頂在更西：往左縮
291                 case PEAK  -> { }                 // 不可能（上面已 return）
292             }
293 
294             // 迴圈不變量（loop invariant）：
295             // 山頂永遠在 [top..bottom] × [left..right] 這個矩形裡。
296             // 若矩形變成空的，代表地圖違反了前置條件。
297             assert top <= bottom && left <= right
298                     : "搜尋矩形變成空的：這張地圖大概不是有效的單峰地圖";
299         }
300     }
301 
302     // ================================================================
```

```
303     // 測試輔助工具
304     // ================================================================
305
306     /**
307      * 產生一張「錐形」測試地圖：
308      * 以 (peakY, peakX) 為山頂，高度隨距離遞減——
309      *   高度 = base − 3 * |y − peakY| − 2 * |x − peakX|
310      *
311      * 兩個方向刻意用「不同」的權重（3 與 2），
312      * 這樣任何一格的四個鄰居永遠不會同高（不會打平），
313      * 每一格的正確答案就唯一，方便寫測試。
314      *
315      * 這種地圖是有效的單峰地圖：
316      * 任何位置往最高鄰居走，都嚴格更接近山頂。
317      */
318     static int[][] coneMap(int rows, int cols, int peakY, int peakX) {
319         int[][] map = new int[rows][cols];
320         int base = 3 * rows + 2 * cols;   // 讓所有高度都是正數
321         for (int y = 0; y < rows; y++) {
322             for (int x = 0; x < cols; x++) {
323                 map[y][x] = base − 3 * Math.abs(y − peakY)
324                                  − 2 * Math.abs(x − peakX);
325             }
326         }
327         return map;
328     }
329
330     /**
331      * 錐形地圖上，座標 (y, x) 的「理論正確答案」。
332      *
333      * 因為列方向權重（3）大於行方向權重（2）：
334      *   − 只要列還沒對齊山頂（y != peakY），
335      *     南北方向的高度差 ±3 一定壓過東西的 ±2，
336      *     所以答案必是 NORTH 或 SOUTH。
337      *   − 列對齊之後才輪到東西方向。
338      *   − 兩者都對齊，就是 PEAK。
339      * 有了這個「封閉公式」，我們就能窮舉地圖上每一格，
340      * 把 lookAround 的輸出跟理論答案逐一比對。
341      */
342     static Direction expectedOnCone(int y, int x, int peakY, int peakX) {
343         if (y == peakY && x == peakX) return Direction.PEAK;
344         if (y != peakY) {
345             return (peakY < y) ? Direction.NORTH : Direction.SOUTH;
346         }
347         return (peakX > x) ? Direction.EAST : Direction.WEST;
348     }
349
350     /** 計算 ceil(log2(n))，用來檢驗 fastTravel 的步數上限。 */
```

```
351     static int ceilLog2(int n) {
352         if (n <= 1) return 0;
353         // numberOfLeadingZeros：int 的 32 位元中，開頭有幾個 0。
354         // 32 - nlz(n - 1) 剛好就是 ceil(log2(n))。
355         return 32 - Integer.numberOfLeadingZeros(n - 1);
356     }
357 
358     // ================================================================
359     // 測試（全部用 assert；請以 java -ea Mountain 執行）
360     // ================================================================
361 
362     /** 測試 1：講義裡的 3×3 範例地圖。 */
363     static void testLookAroundOnLabExample() {
364         int[][] smallMap = {
365                 {10, 20, 10},   // Row 0
366                 {20, 50, 20},   // Row 1
367                 {10, 20, 10}    // Row 2
368         };
369 
370         // 中央 50 比四周（20,20,20,20）都高 → PEAK
371         assert lookAround(1, 1, smallMap) == Direction.PEAK;
372 
373         // 四個「邊」的中點：最高鄰居都是中央的 50
374         assert lookAround(0, 1, smallMap) == Direction.SOUTH; // 下方 50 最高
375         assert lookAround(1, 0, smallMap) == Direction.EAST;  // 右方 50 最高
376         assert lookAround(2, 1, smallMap) == Direction.NORTH; // 上方 50 最高
377         assert lookAround(1, 2, smallMap) == Direction.WEST;  // 左方 50 最高
378 
379         // 角落 (0,0)：東（20）與南（20）同高「打平」。
380         // 依我們文件化的打平規則（北→東→南→西），應回傳 EAST。
381         assert lookAround(0, 0, smallMap) == Direction.EAST;
382 
383         System.out.println("[1] lookAround：3x3 講義範例地圖 ... 通過");
384     }
385 
386     /** 測試 2：手工設計的 3×4 地圖，全部 12 格逐一驗證（完整分支覆蓋）。 */
387     static void testLookAroundFullBranchCoverage() {
388         // 山頂在 (1, 2) 的 90。所有相鄰高度都不同，答案唯一。
389         int[][] map = {
390                 { 5, 30, 40, 35},   // Row 0
391                 {10, 50, 90, 60},   // Row 1
392                 { 8, 20, 45, 30}    // Row 2
393         };
394 
395         // ---- Row 0（上邊界：北邊是界外）----
396         assert lookAround(0, 0, map) == Direction.EAST;  // 角落：東30 > 南10
397         assert lookAround(0, 1, map) == Direction.SOUTH; // 南50 最高
398         assert lookAround(0, 2, map) == Direction.SOUTH; // 南90 最高
```

```
399         assert lookAround(0, 3, map) == Direction.SOUTH; // 角落：南60 > 西40
400
401         // ---- Row 1 ----
402         assert lookAround(1, 0, map) == Direction.EAST;  // 左邊界：東50 最高
403         assert lookAround(1, 1, map) == Direction.EAST;  // 東90 最高
404         assert lookAround(1, 2, map) == Direction.PEAK;  // 山頂本人！
405         assert lookAround(1, 3, map) == Direction.WEST;  // 右邊界：西90 最高
406
407         // ---- Row 2（下邊界：南邊是界外）----
408         assert lookAround(2, 0, map) == Direction.EAST;  // 角落：東20 > 北10
409         assert lookAround(2, 1, map) == Direction.NORTH; // 北50 最高
410         assert lookAround(2, 2, map) == Direction.NORTH; // 北90 最高
411         assert lookAround(2, 3, map) == Direction.NORTH; // 角落：北60 > 西45
412
413         // 五種 Direction 值（含 PEAK）都至少被回傳一次 → 完整分支覆蓋
414         System.out.println("[2] lookAround：3x4 地圖 12 格全驗證（分支覆蓋）... 通過");
415     }
416
417     /** 測試 3：退化地圖——1×1、單列、單行。邊界處理最容易出錯的地方。 */
418     static void testLookAroundDegenerateMaps() {
419         // 1×1 地圖：四個鄰居全是界外 → 自己就是山頂
420         int[][] tiny = {{42}};
421         assert lookAround(0, 0, tiny) == Direction.PEAK;
422
423         // 單列地圖（1×5）：只能往東西走
424         int[][] oneRow = {{10, 20, 30, 40, 25}};
425         assert lookAround(0, 0, oneRow) == Direction.EAST;  // 20 在右邊
426         assert lookAround(0, 3, oneRow) == Direction.PEAK;  // 40 是山頂
427         assert lookAround(0, 4, oneRow) == Direction.WEST;  // 40 在左邊
428
429         // 單行地圖（5×1）：只能往南北走
430         int[][] oneCol = {{10}, {20}, {30}, {40}, {25}};
431         assert lookAround(0, 0, oneCol) == Direction.SOUTH; // 20 在下面
432         assert lookAround(3, 0, oneCol) == Direction.PEAK;  // 40 是山頂
433         assert lookAround(4, 0, oneCol) == Direction.NORTH; // 40 在上面
434
435         System.out.println("[3] lookAround：1x1／單列／單行 退化地圖 ... 通過");
436     }
437
438     /** 測試 4：在多張錐形地圖上「窮舉」每一格，逐一比對理論答案。 */
439     static void testLookAroundExhaustivelyOnConeMaps() {
440         int[][] shapes = {{1, 1}, {1, 8}, {8, 1}, {5, 5}, {9, 7}, {12, 24}};
441         int checked = 0;
442
443         for (int[] shape : shapes) {
444             int rows = shape[0];
445             int cols = shape[1];
446             // 讓山頂輪流出現在地圖的「每一個」位置
```

```
            for (int peakY = 0; peakY < rows; peakY++) {
                for (int peakX = 0; peakX < cols; peakX++) {
                    int[][] map = coneMap(rows, cols, peakY, peakX);
                    // 再對地圖上的「每一格」比對答案
                    for (int y = 0; y < rows; y++) {
                        for (int x = 0; x < cols; x++) {
                            Direction got  = lookAround(y, x, map);
                            Direction want = expectedOnCone(y, x, peakY, peakX);
                            assert got == want
                                    : "在 " + rows + "x" + cols + " 地圖（山頂 ("
                                    + peakY + "," + peakX + ")）的座標 ("
                                    + y + "," + x + ")：期望 " + want
                                    + "，卻得到 " + got;
                            checked++;
                        }
                    }
                }
            }
        }
        System.out.println("[4] lookAround：錐形地圖窮舉測試，共比對 "
                + checked + " 個座標 ... 通過");
    }

    /** 測試 5：climb 的手工小地圖測試（山頂在不同角落）。 */
    static void testClimbOnHandCraftedMaps() {
        System.out.println("[5] climb：手工地圖測試");

        // (a) 山頂就在出發點 [0][0]：應該 0 步完成
        int[][] cornerPeak = {
                {99, 80, 60},
                {85, 70, 50},
                {62, 55, 40}
        };
        int[] p1 = climb(cornerPeak);
        assert Arrays.equals(p1, new int[]{0, 0})
                : "山頂應在 (0,0)，卻回傳 " + Arrays.toString(p1);
        assert lastClimbSteps == 0 : "從山頂出發應該走 0 步";

        // (b) 山頂在對角（右下角）：距離出發點最遠
        int[][] farPeak = {
                { 5, 12, 20},
                {14, 30, 44},
                {22, 48, 77}
        };
        int[] p2 = climb(farPeak);
        assert Arrays.equals(p2, new int[]{2, 2})
                : "山頂應在 (2,2)，卻回傳 " + Arrays.toString(p2);
        // 有效地圖上每一步都嚴格接近山頂，
```

```
        // 所以步數 = 曼哈頓距離 |2-0| + |2-0| = 4
        assert lastClimbSteps == 4 : "步數應為 4，卻是 " + lastClimbSteps;

        // (c) 講義的 3×3 範例：山頂在正中央
        int[][] smallMap = {
                {10, 20, 10},
                {20, 50, 20},
                {10, 20, 10}
        };
        int[] p3 = climb(smallMap);
        assert Arrays.equals(p3, new int[]{1, 1})
                : "山頂應在 (1,1)，卻回傳 " + Arrays.toString(p3);
        assert lastClimbSteps == 2 : "步數應為 2，卻是 " + lastClimbSteps;

        System.out.println("    手工地圖全部通過");
    }

    /** 測試 6：climb 與 fastTravel 在各種尺寸、各種山頂位置的錐形地圖上。 */
    static void testClimbAndFastTravelOnConeMaps() {
        System.out.println("[6] climb + fastTravel：錐形地圖系統化測試");

        int[][] shapes = {{1, 1}, {1, 9}, {9, 1}, {5, 8}, {24, 12}};

        for (int[] shape : shapes) {
            int rows = shape[0];
            int cols = shape[1];

            // 測試的山頂位置：四個角落 + 正中央（用 HashSet 去掉重複）
            int[][] peaks = {
                    {0, 0}, {0, cols - 1},
                    {rows - 1, 0}, {rows - 1, cols - 1},
                    {rows / 2, cols / 2}
            };
            HashSet<String> seen = new HashSet<>();

            for (int[] pk : peaks) {
                int peakY = pk[0];
                int peakX = pk[1];
                if (!seen.add(peakY + "," + peakX)) continue; // 跳過重複位置

                int[][] map = coneMap(rows, cols, peakY, peakX);
                System.out.println("    地圖 " + rows + "x" + cols
                        + "，山頂在 (" + peakY + "," + peakX + ")：");

                // ---- climb ----
                System.out.print("      ");
                int[] got = climb(map);
                assert Arrays.equals(got, new int[]{peakY, peakX})
```

```
                        : "climb 找錯山頂：" + Arrays.toString(got);
                // 步數應等於從 (0,0) 到山頂的曼哈頓距離
                assert lastClimbSteps == peakY + peakX
                        : "climb 步數應為 " + (peakY + peakX)
                        + "，卻是 " + lastClimbSteps;

                // ---- fastTravel ----
                System.out.print("      ");
                int[] got2 = fastTravel(map);
                assert Arrays.equals(got2, new int[]{peakY, peakX})
                        : "fastTravel 找錯山頂：" + Arrays.toString(got2);
                // 步數上限：每次探測至少砍掉某一邊的一半
                int bound = ceilLog2(rows) + ceilLog2(cols) + 2;
                assert lastFastTravelSteps <= bound
                        : "fastTravel 步數 " + lastFastTravelSteps
                        + " 超過理論上限 " + bound;

                // ---- 兩種方法必須找到同一個山頂 ----
                assert Arrays.equals(got, got2)
                        : "climb 與 fastTravel 的答案不一致！";
            }
        }
        System.out.println("    系統化測試全部通過");
    }

    /** 示範：在大地圖上比較 climb 與 fastTravel 的步數差距。 */
    static void demoSpeedComparison() {
        System.out.println("[7] 步數比較：128x128 地圖，山頂在 (100, 111)");
        int[][] big = coneMap(128, 128, 100, 111);

        System.out.print("    ");
        int[] a = climb(big);        // 預期 100 + 111 = 211 步
        System.out.print("    ");
        int[] b = fastTravel(big);   // 預期最多 7 + 7 + 2 = 16 次探測

        assert Arrays.equals(a, b);
        assert lastClimbSteps == 211;
        assert lastFastTravelSteps <= 16;
        System.out.println("    同一座山，climb 走了 " + lastClimbSteps
                + " 步；fastTravel 只探測 " + lastFastTravelSteps + " 次！");
    }

    // ================================================================
    // main：依序執行所有測試
    // ================================================================

    public static void main(String[] args) {
        // 小技巧：故意利用 assert 的「副作用」偵測 -ea 是否開啟。
```

```
        // 若 assert 沒啟用，這行不會執行，assertionsEnabled 維持 false。
        boolean assertionsEnabled = false;
        assert assertionsEnabled = true;   // 啟用時才會把它設成 true
        if (!assertionsEnabled) {
            System.out.println("【警告】assert 未啟用！以下測試不會真正檢查。");
            System.out.println("        請改用：java -ea Mountain");
        }

        System.out.println("=== Lab 3: Mountain Climbing — 完整測試 ===");
        testLookAroundOnLabExample();          // Activity 1
        testLookAroundFullBranchCoverage();    // Activity 1
        testLookAroundDegenerateMaps();        // Activity 1
        testLookAroundExhaustivelyOnConeMaps();// Activity 1
        testClimbOnHandCraftedMaps();          // Activity 2
        testClimbAndFastTravelOnConeMaps();    // Activity 2 + 3
        demoSpeedComparison();                 // Activity 2 vs 3

        System.out.println();
        System.out.println("=== 全部測試通過！ ===");
    }
}
```

程式 13 Mountain.java 完整內容