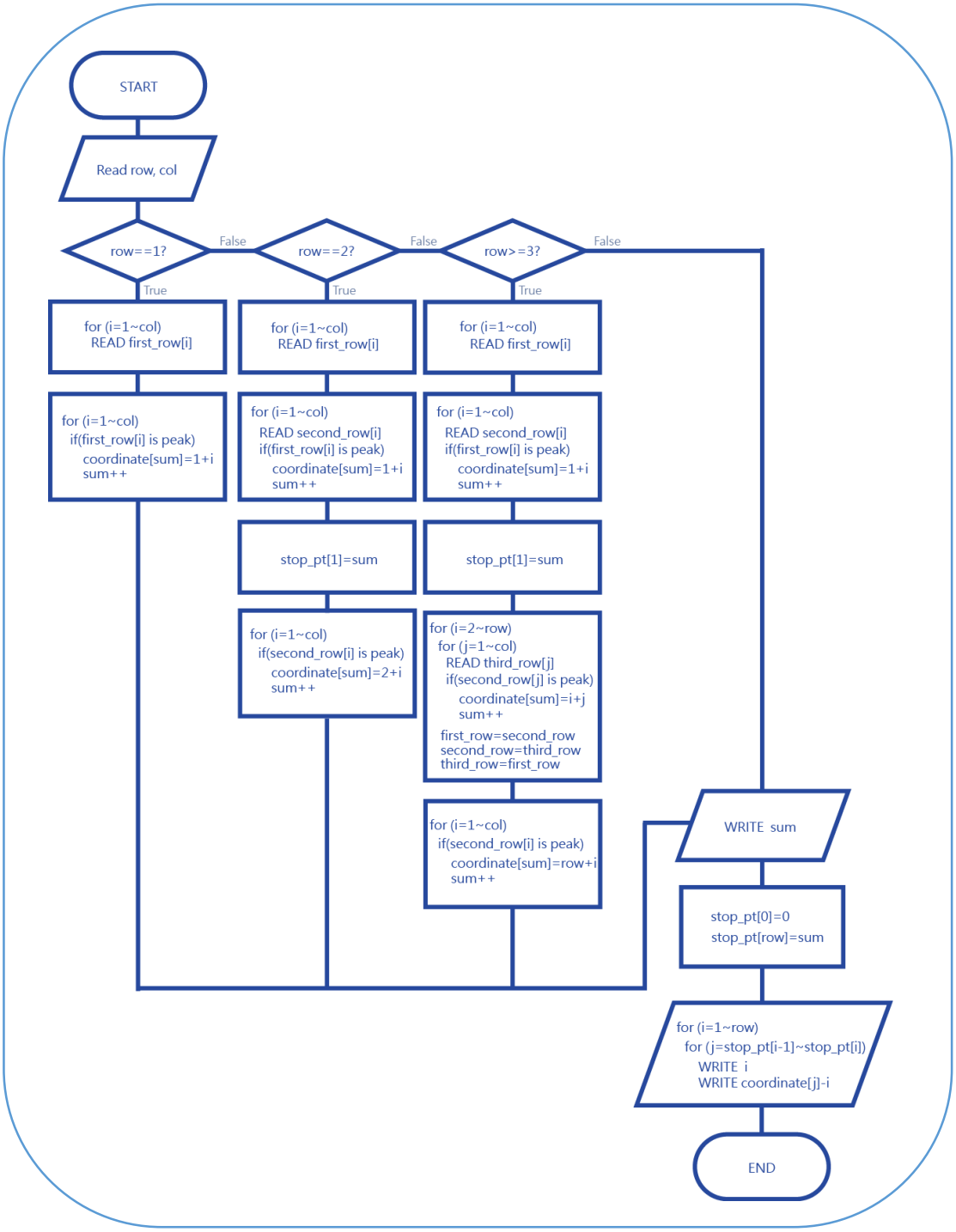
PROJECT1 – PEAK FINDER

106000103 趙貞豪

Project Description

-Program Flow Chart:



-Detailed Description

首先說明個變數所代表的意義：

變數名稱	意義
first_row	存放相對第一個 row 的值
second_row	存放相對第二個 row 的值
third_row	存放相對第三個 row 的值
sum	峰點總數
coordinate	峰點的 row 跟 col 的相加值
stop_pt	每一次 row 運算完後峰點的總數

程式的一開始，讀取 row, col 的值，先做分類的動作，有兩個 special case 分別是 row=1 跟 row=2 的時候，row=1 的時候，我們就只需檢查完一行後，把結果加入 coordinate 當中，最後輸出即可；而 row=2 時也相對簡單，基本的概念是先讀取第一行存到 first_row 這個陣列當中，再來，讀取第二行存到 second_row 的同時檢查第一行是否有峰點的出現，若有出現，則加入 coordinate 陣列當中，最後再掃一次 second_row 檢查剩下的峰點，最後輸出；當 row>=3 時，將進入一個比較 general 的處理方式，我一樣先處理第一行的讀取，以及第二行的讀取和第一行的檢查，這樣一來，我能得到了一些初始資料：first_row, second_row 與 coordinate 當中存入關於第一行峰點的資料。之後進入一個兩層 for loop，我要利用剛剛得到的初始資料，從 i=2~row-1, j=1~col，每一次讀進一個新的值到 third_row 當中，並檢查 second_row 當中的值是否為峰點，如果是，則將 i+j 的值存入 coordinate 當中，sum++，並在每一次 row 之間都記錄一次 stop_pt，把 sum 的狀況記錄下來，並在每一次 row 檢查完後 swap 三個 array（我用三個指標互換的方式執行，也就是 first_row = second_row, second_row = third_row, third_row = first_row）。最後，再掃過最後一列檢查峰點，並輸出結果。

first_row, second_row, third_row 的運用，是為了省 memory space，因為比起開一個二維陣列去存取 matrix 的資料，我每比較一個峰點只需要左右上下的資料就好，多餘的資訊在運算的過程中就可以忘記了，因此在一個 1000*1000 的二維陣列當中，我只需要 1000*3 就夠了。

coordinate 跟 stop_pt 的運用也是為了省空間，我原本需要用兩個空間去記峰點的 row 跟 col，但是因為測資範圍在 1000 內，int 不會有 overflow 的問題（1000+1000=2000），所以我把兩者相加做紀錄，並配合 stop_pt 記錄在各 row 之間 sum 總數的變化。因此原本需要 row 跟 col 各 1000000 的儲存空間，我只需 coordinate 存一次，以及在各個 row 紀錄峰點的 stop_pt 配合，用 1000000+1000

的空間儲存。

Test case Design

-Detailed Description of the Test case

其實我做 test case 的時候考量兩點，第一點是速度，速度很容易被輸出檔案拖慢，所以輸出多會是衡量的一個方式，我的 test case 的 output 會有 333666 個峰點，算是數量來說不少。

第二個是正確性，其實這點很主觀，我原本是要出整片平原的測資，也就是輸出為 1000000 個峰點，但是後來因為有一次我用這個測資測自己的演算法，發現出錯，我才改用這個測資。這個測資的特點是隔兩列峰點，所以峰點會出現在 1, 4, 7, … n（all n%3==1）行，我原本的寫法是認為每一行都會有輸出峰點，當然這跟部分的情況會吻合（我為了卯起來測時間，所以測資都餵很大，所以峰點都離很近），但其實峰點是有可能很疏離的。

討論

關於時間的節省，其實說我想不到什麼好方法，我只能分享一些小發現。我 1 最初先是造了一個普通版本的 peak_finder，它的方式是先將陣列存起來，之後針對每一個點判斷是否為峰點，是的話就將 sum++，把 i, j 座標分別存在兩個陣列當中，最後輸出。後來，2 經過改寫後（也就是我現在的寫法），意外地將輸出的 std::endl 改成\n，卻發現輸出在速度上有減少，實在非常神奇。

這兩者之間的差距甚大，當矩陣數量非常大時，兩者速度可以差到 3 倍左右，std::endl 非常耗時間，若是輸出 std::endl 等於說要把緩衝區清掉，等於多執行了一些的東西。所以後來我改成用\n 來輸出換行後就快了許多。以下圖說明兩者差異：

1

```
///output.
out_file<<sum<<std::endl;
for(int i=0;i<sum;i++){
    out_file<<x[i]<<" "<<y[i]<<std::endl;
}
```

2

```
///output.
out_file<<sum<<"\n";
stop_pt[0]=0;
stop_pt[row]=sum;
for(int i=1;i<=row;i++){
    for(int j=stop_pt[i-1];j<stop_pt[i];j++){
        out_file<<i<<" "<<coordinate[j]-i<<"\n";
    }
}
```

還有另一個關於時間的機制，就是因為我的迴圈設計是依據 row 的大小而定，但是其實並不是每一個 row 都有峰點，所以嘗試將輸出改成每輸出一筆資料就 sum--，而 sum=0 就會停止迴圈，看會不會變快。以下圖說明：

3

```
///output.
out_file<<sum<<"\n";
stop_pt[0]=0;
stop_pt[row]=sum;
for(int i=1;i<=row&&sum>0;i++){
    for(int j=stop_pt[i-1];j<stop_pt[i];j++){
        out_file<<i<<" "<<coordinate[j]-i<<"\n";
        sum--;
    }
}
```

不過 2 跟 3 的差別也只有在極端的 case 才會出現，像是峰點只出現在第一個 row 等等，如果對一般多峰點的情形，其實有點多此一舉。因此我最後沒有採用。