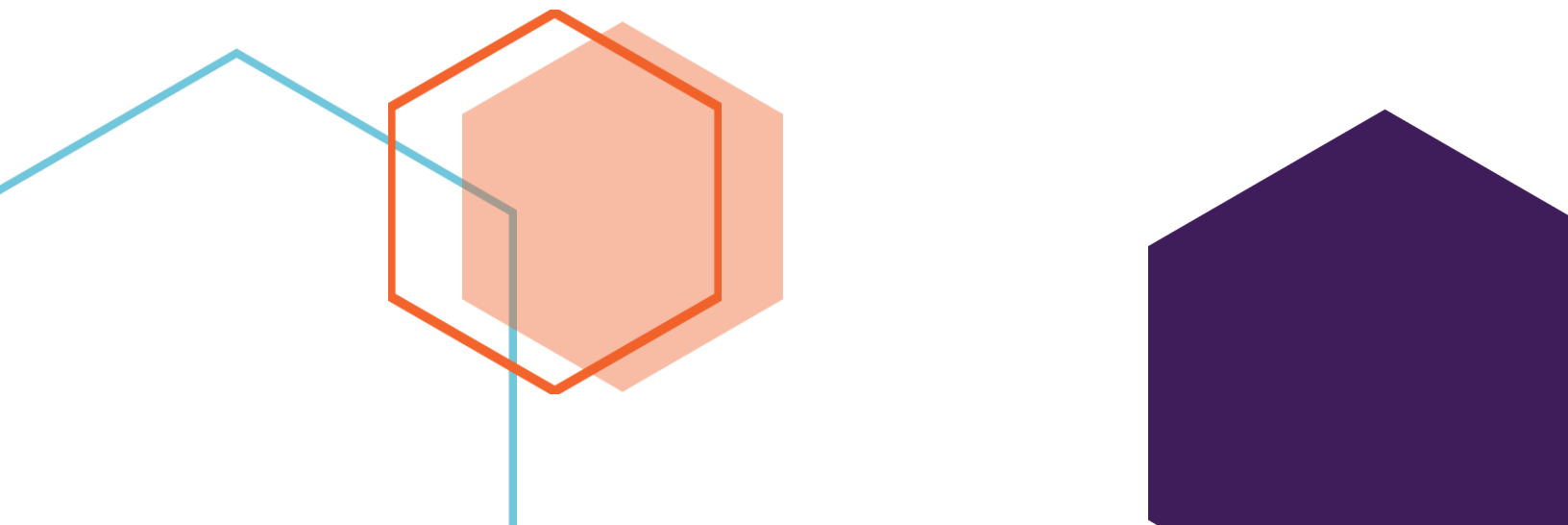




Algorithm to find all peak in 2D array

Sajjad Ranjbar Yazdi
ID: 9812223294
30 Nov

In matrices, a peak element is an element that is larger than its four adjacent elements (up, right, left, and bottom). The goal is to design an algorithm that, upon receiving a matrix, prints all of its peak elements.



Algorithm to find all peak in 2D array

Matrix structure

Suppose the matrix A is given to us as m.n matrix. As we said a peak element is an element that is larger than its four adjacent elements (up, right, left, and bottom). It is a little harder to find the peak elements at the edge of the matrix. Because not all adjacent elements can be analyzed for them. For example, the element in the house [0, 0] has no upper and left neighbors. As a result, the analyzing of peak elements for these elements requires the use of many conditional statements and reduces the performance of the algorithm. To prevent this subject, we fill the matrix round with infinite minus numbers...

Phase 1

The algorithm is very simple in the first phase. In this step, the matrix is mapped linearly and the elements that meet the above conditions are stored. The element can be printed immediately after identification. But to make the codes more perceptible, the elements have not been printed immediately.

Phase 2

The basic idea of problem solving in this section is to use **binary search**. In the first step, we present a function to find the peak elements in a one-dimensional array, and then we generalize it to two dimensions.

Conclusion

Investigation of time complexity and totality of phases.

GOAL

...

In matrices, a peak element is an element that is larger than its four adjacent elements (up, right, left, and bottom). The goal is to design an algorithm that, upon receiving a matrix, prints all of its peak elements.

To do this, in the first step, we will prepare the matrix for the algorithm. Then, in two different phases, we design two different algorithms to do this.

Matrix structure

Suppose the matrix A is given to us as m.n matrix. As we said a peak element is an element that is larger than its four adjacent elements (up, right, left, and bottom). It is a little harder to find the peak elements at the edge of the matrix. Because not all adjacent elements can be analyzed for them. For example, the element in the house $[0, 0]$ has no upper and left neighbors. As a result, the analyzing of peak elements for these elements requires the use of many conditional statements and reduces the performance of the algorithm. To prevent this subject, we fill the matrix round with infinite minus numbers.

In order to avoid wasting time and also not to use additional functions, we do this in both phases when forming a matrix in the constructor functions. In simple terms, each matrix has a shell of minimum integer (minus infinite) and the main matrix is located in its central core. To better understand this issue, consider the following example. In Figure 1.1 you can see the input matrix. To prepare this matrix. We draw an minus infinite integers shells around it.

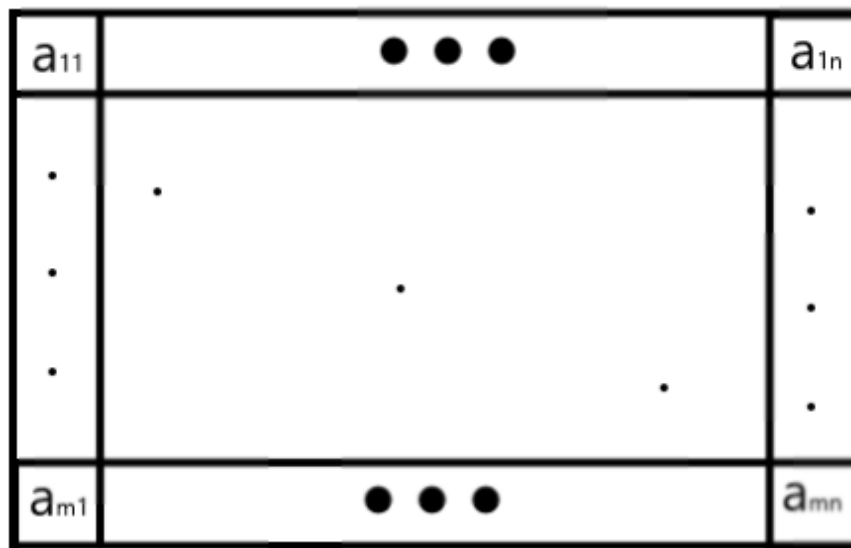


Figure 1.1

Algorithm to find all peak in 2D array

...

In Figure 1.2, all the values in the **blue** part are minus infinite.

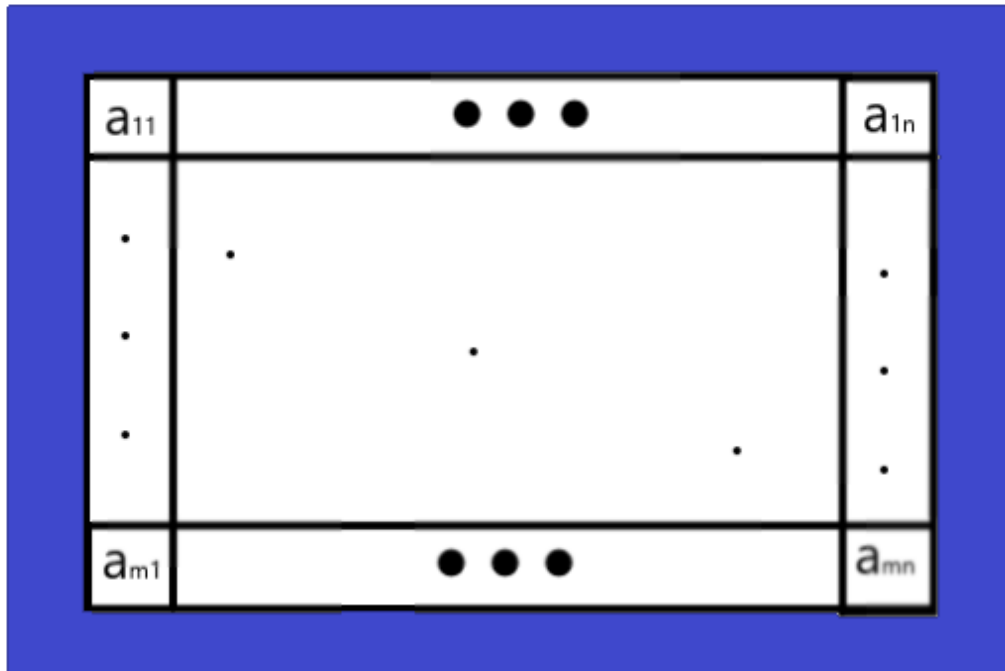


Figure 1.2



Phase1

The algorithm is very simple in the first phase. In this step, the matrix is mapped linearly and the elements that meet the above conditions are stored. The element can be printed immediately after identification. But to make the codes more perceptible, the elements have not been printed immediately.

In the first part of the class, we define the required matrix variables. The main variables of the algorithm include data, number of rows and columns. The confirm variable is also defined for the eloquence of the algorithm. All values of this variable are initially equal to false, and if an element is known as a peak element, the corresponding variable is equal to true. Eventually all the confirmed information will be printed in the Print function.

```
private:
    long long int** data;
    int column, row;

    //confirmation for peak elements
    bool** confirm;
```

In the second step, we are faced with two functions, Make and Set. First, the assumptions for the matrix are prepared in the Make function and then the program is referred to the set function. The set function is designed to receive the requested values from the user and place them in the matrix. Note that the number of rows and columns has already been received from the user by the constructor.

```
void matrix::make()
{
    //make confirmation matrix
    confirm = new bool* [row];
    for (int i = 0; i < row; ++i)
        confirm[i] = new bool[column];

    //make matrix of informations
    data = new long long int* [row];
    for (int i = 0; i < row; ++i)
        data[i] = new long long int[column];

    //The margin elements are equal to INT_MIN
    for (int counter1 = 0; counter1 < row; counter1++)
        for (int counter2 = 0; counter2 < column; counter2++)
            if (counter1==0 || counter2==0 || counter1 == row-1 || counter2==column - 1)
                data[counter1][counter2] = INT_MIN;
}
```

Algorithm to find all peak in 2D array

• • •

```
void matrix::set()
{
    //make elements
    make();
    for (int counter1 = 1; counter1 <= row - 2; counter1++)
        for (int counter2 = 1; counter2 <= column - 2; counter2++)
        {
            //get elements of matrix from user
            cin >> data[counter1][counter2];
            //all the elements of confirmation are equal to false
            confirm[counter1][counter2] = false;
        }
}
```

Pop functions are designed solely for the sake of more code rhetoric and are not the main functions. These functions are only for checking elements with adjacent information. The check function is the main part of the program and with the help of pop functions, the peak elements are identified. The each confirm corresponding to peak element will be equal to true by this function.

```
bool popup(int counter1, int counter2) { if (data[counter1][counter2] > data[counter1 - 1][counter2]) { return true; } }
bool popright(int counter1, int counter2) { if (data[counter1][counter2] > data[counter1][counter2 + 1]) { return true; } }
bool popdown(int counter1, int counter2) { if (data[counter1][counter2] > data[counter1 + 1][counter2]) { return true; } }
bool popleft(int counter1, int counter2) { if (data[counter1][counter2] > data[counter1][counter2 - 1]) { return true; } }
```

The print function is also very simple. This function prints all the confirmed elements as peak elements.

```
void matrix::print()
{
    //All peak elements are printed
    for (int counter1 = 1; counter1 < row - 1; counter1++)
        for (int counter2 = 1; counter2 < column - 1; counter2++)
            if (confirm[counter1][counter2])
                cout << data[counter1][counter2] << endl;
}
```

Now that we know how this algorithm works, let's look at a few examples.

```
int main()
{
    int column, row;
    cin >> column >> row;
    matrix m1(column, row);
    m1.set();

    m1.check();
    m1.print();

    system("pause");
    return 0;
}
```

Algorithm to find all peak in 2D array

...

Input

```
5 5
1 4 7 3 8
2 5 3 7 5
2 8 4 6 3
2 3 6 4 8
7 4 9 2 5
```

Output

```
7 8 7 8 8 7 9
```

Input

```
3 4
2836 -876 -109
7836 4354 -897
2673 9873 9083
2983 9973 9982
```

Output

```
-109
7836
9982
```



Phase 2

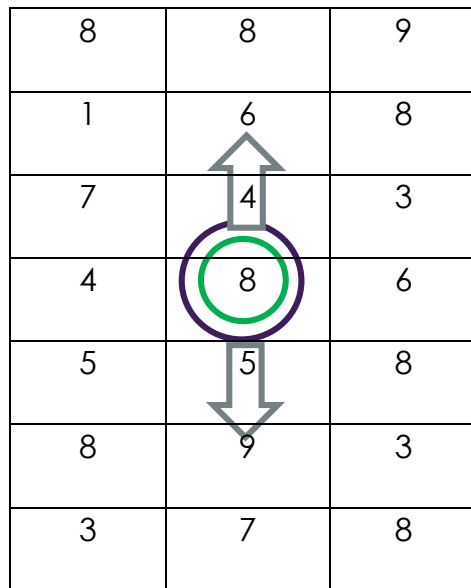

The basic idea of problem solving in this section is to use **binary search**. In the first step, we present a function to find the peak elements in a one-dimensional array, and then we generalize it to two dimensions. Since it is much easier to work with strings in Python than CPP, I have preferred to use this language in this section. Constructive functions are the method of using data as in phase one. As a result, I refrain from re-explaining it.

```
def Div(lst, low, high, c):
    mid = (low + high)//2
    if low > high:
        return ""
    else:
        if mid+1 < len(lst) and lst[mid][c] > lst[mid+1][c] and lst[mid][c] > lst[mid-1][c]
and lst[mid][c] > lst[mid][c+1] and lst[mid][c] > lst[mid][c-1]:
            return str(lst[mid][c]) + " " + Div(lst, low, mid-1, c) + Div(lst, mid+1, high,
c)
        else:
            return Div(lst, low, mid-1, c) + Div(lst, mid+1, high, c)
```

Consider the first column of a matrix. The middle element of this column is called the mid. This element is compared to all its adjacent elements and if it is a peak, it is returned by the function. Following the return, go to the up and down half of the mid and continue the same operation as before.

```
return str(lst[mid][c]) + " " + Div(lst, low, mid-1, c) + Div(lst, mid+1, high, c)
```

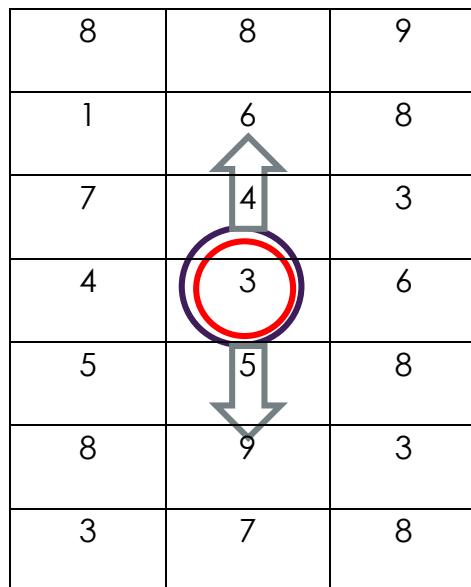


Algorithm to find all peak in 2D array



8	8	9
1	6	8
7	4	3
4	8	6
5	5	8
8	9	3
3	7	8

What if the mid value is not equal to the peak element? In this case, the program ignores this value and resumes the same process of going up and down.

```
return Div(lst, low, mid-1, c) + Div(lst, mid+1, high, c)
```

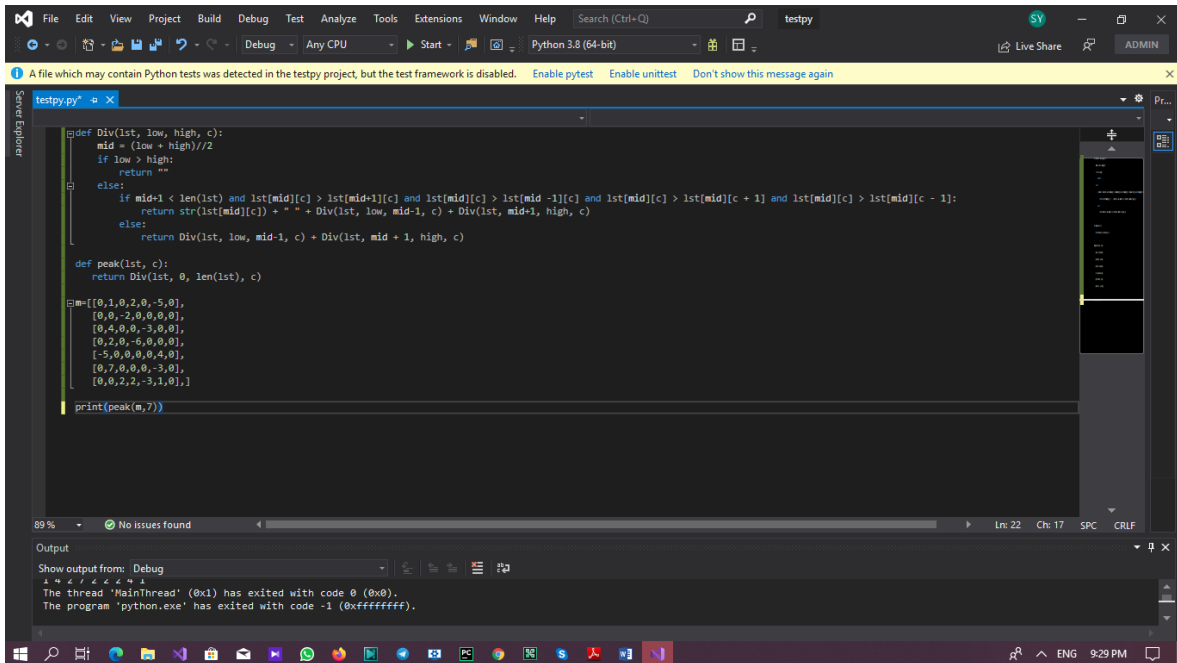


8	8	9
1	6	8
7	4	3
4	3	6
5	5	8
8	9	3
3	7	8

Algorithm to find all peak in 2D array



Note the example in Figures 3 and 4.



```
def Div(lst, low, high, c):
    mid = (low + high)//2
    if low > high:
        return ""
    else:
        if mid-1 < len(lst) and lst[mid][c] > lst[mid-1][c] and lst[mid][c] > lst[mid+1][c] and lst[mid][c] > lst[mid][c+1] and lst[mid][c] > lst[mid][c-1]:
            return str(lst[mid][c]) + " " + Div(lst, low, mid-1, c) + Div(lst, mid+1, high, c)
        else:
            return Div(lst, low, mid-1, c) + Div(lst, mid+1, high, c)

def peak(lst, c):
    return Div(lst, 0, len(lst), c)

m=[[0,1,0,2,0,-5,0],
   [0,0,-2,0,0,0,0],
   [0,4,0,0,-3,0,0],
   [0,2,0,-6,0,0,0],
   [-5,0,0,0,0,4,0],
   [0,7,0,0,0,-3,0],
   [0,0,2,2,-3,1,0]]

print(peak(m,7))
```

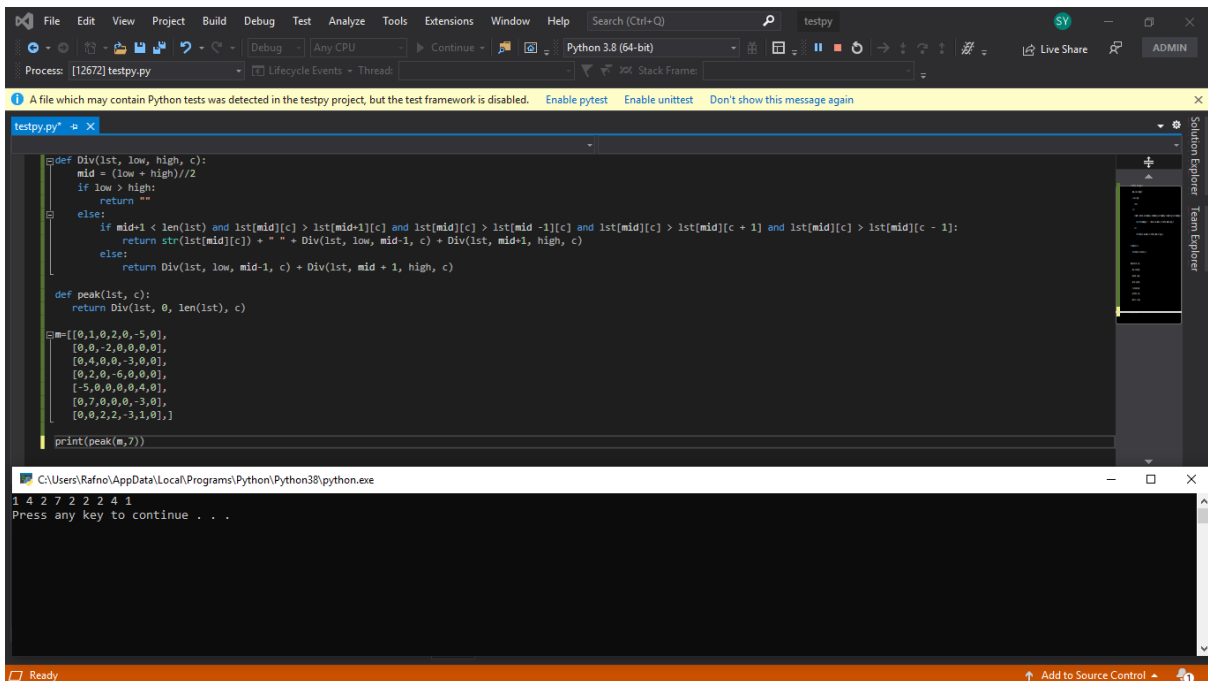
Output

Show output from: Debug

The thread 'MainThread' (0x1) has exited with code 0 (0x0).

The program 'python.exe' has exited with code -1 (0xffffffff).

Figure 3



```
def Div(lst, low, high, c):
    mid = (low + high)//2
    if low > high:
        return ""
    else:
        if mid-1 < len(lst) and lst[mid][c] > lst[mid-1][c] and lst[mid][c] > lst[mid+1][c] and lst[mid][c] > lst[mid][c+1] and lst[mid][c] > lst[mid][c-1]:
            return str(lst[mid][c]) + " " + Div(lst, low, mid-1, c) + Div(lst, mid+1, high, c)
        else:
            return Div(lst, low, mid-1, c) + Div(lst, mid+1, high, c)

def peak(lst, c):
    return Div(lst, 0, len(lst), c)

m=[[0,1,0,2,0,-5,0],
   [0,0,-2,0,0,0,0],
   [0,4,0,0,-3,0,0],
   [0,2,0,-6,0,0,0],
   [-5,0,0,0,0,4,0],
   [0,7,0,0,0,-3,0],
   [0,0,2,2,-3,1,0]]

print(peak(m,7))
```

C:\Users\Rafael\AppData\Local\Programs\Python\Python38\python.exe

1 4 2 7 2 2 2 4 1

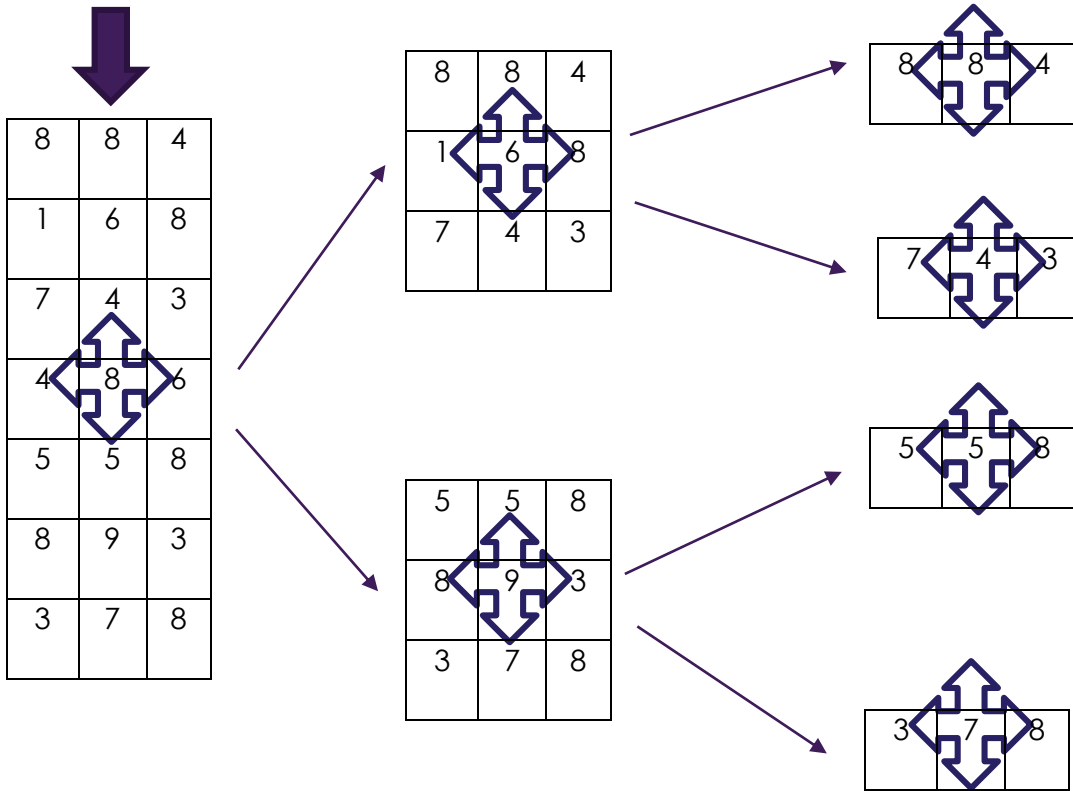
Press any key to continue . . .

Figure 4

Algorithm to find all peak in 2D array

...

The general process of the algorithm is examined in the following example.





Conclusion

Investigation of time complexity and totality of phases. In the first phase, two four loops are used and it is obvious that the time complexity of the order is $O(m \cdot n)$. The main point is about the second phase. This algorithm is made up of two parts. As a result, we have to examine it in two steps.

The first part of the algorithm in the second phase identifies all the peak elements in a column with the binary search method. Since number of elements in each column equal to the number of rows, the complexity time of this function is equal to $O(\text{row})$. In the second part, the algorithm does the same thing for each column, so the number of all columns must be repeated, which means $O(\text{column})$.

According to the contract, the number of rows and columns is determined by the user. The second part is in the heart of the first part and does not follow it. As a result, the time complexity of the whole program is equal to $O(m \cdot n)$.

Algorithm to find all peak in 2D array



References:

1. [Reference](#)
2. [Reference](#)
3. [Reference](#)
4. [Reference](#)
5. [Reference](#)
6. [Reference](#)
7. [Reference](#)
8. [Reference](#)
9. [Reference](#)
10. [Reference](#)