Flooded Island

Here we show how the Codility Challenge codenamed Nitrogenium-2013 can be solved. You can still give it a try, but no certificate will be granted. The problem asks you to count the number of islands on consecutive days with different water levels.

Slow solution $O(N \cdot M)$

The simplest solution is to count separately, for every water level $B_0, B_1, \ldots, B_{N-1}$, the number of islands in $O(N)$ time by simple counting the shores of the islands. In other words, we count the number of indices j such that A_j is above the water and A_{j+1} is under water.

1: Slow solution in Python — $O(N \cdot M)$.

```
def slowSolutionPython(A, B):
    N = len(A)
    M = len(B)
    result = [0] * M
    for i in xrange(M):
        water = B[i]
        island = 0
        for j in xrange(N):
            if A[j] > water and (j == N - 1 or A[j + 1] <= water):
                island += 1
        result[i] = island
    return result
```

Below is an analogical solution in Java-8 which makes use of streams.

2: Slow solution in Java-8 — $O(N \cdot M)$.

```
class slowSolutionJava {
    int[] solution(int[] A, int[] B) {
        int N = A.length;
        return Arrays
            .stream(B)
            .map(water -> IntStream
                .range(0, N)
                .filter(j -> (A[j] > water) && (j == N - 1 || A[j + 1]
                    <= water))
                .map(i -> 1)
                .sum())
            .toArray();
```

```
12        }
13    }
```

The time complexity of the above algorithm is $O(N \cdot M)$, which is far from optimal.

Golden solution $O(N + M + \max(A) + \max(B))$

Imagine that the water level only decreases on consecutive days, and that on the first day all the plots are under water, so there aren't any islands. Next, we will investigate how the number of islands changes depending on the water level. More precisely, we want to know the value by which the number of islands increases or decreases if the water level decreases to some level.

Each island has two sides (shores): left and right. Instead of counting the islands, we can count their right sides. A position j is the right side of an island only if:

- $A_j > A_{j+1}$ (or $j = N - 1$),
- the water level is below A_j, but not below A_{j+1}.

So for every position j, such that $A_j > A_{j+1}$ (or $j = N - 1$), when the water level falls below A_j the number of islands increases by one, but when it falls below A_{j+1} it decreases by one.

For any water level, the number of islands is just the sum of changes in the number of islands as the water level was decreasing.

3: Golden solution in Python — $O(N + M + \max(A) + \max(B))$.

```python
def solution(A, B):
    N = len(A)
    M = len(B)
    size = max(max(A), max(B))
    island = [0] * (size + 2)
    for i in xrange(1, N):
        if A[i - 1] > A[i]:
            island[A[i - 1]] += 1
            island[A[i]] -= 1
    island[A[N - 1]] += 1
    for i in xrange(size, -1, -1):
        island[i] += island[i + 1]
    result = [0] * M
    for i in xrange(M):
        result[i] = island[B[i] + 1]
    return result
```

4: Golden solution in Java-8 — $O(N + M + \max(A) + \max(B))$.

```java
class Solution {
    int maxElem(int[] A) {
        return Arrays.stream(A).max().getAsInt();
    }
    int[] solution(int[] A, int[] B) {
        int N = A.length;
        int size = Math.max(maxElem(A), maxElem(B));
        int island[] = new int[size + 2];
        IntStream
            .range(0, N - 1)
            .filter(j -> A[j] > A[j + 1])
```

```
12              .forEach(j -> {
13                  island[A[j]] += 1;
14                  island[A[j + 1]] -= 1;
15              });
16          island[A[N - 1]] += 1;
17          IntStream
18              .range(-size, 0)
19              .forEach(i -> island[-i] += island[-i + 1]);
20          return Arrays
21              .stream(B)
22              .map(water -> island[water + 1])
23              .toArray();
24      }
25  }
```

The time complexity of the above algorithm is $O(N + M + \max(A) + \max(B))$.