Beacon Platform

2021 May 15

Q1. Dice with same number

A six-sided die is a small cube with a different number of pips on each face (side), ranging from 1 to 6. On any two opposite sides of the cube, the number of pips adds up to 7; that is, there are three pairs of opposite sides: 1 and 6, 2 and 5, and 3 and 4.

There are N dice lying on a table, each showing the pips on its top face. In one move, you can take one die and rotate it to an adjacent face. For example, you can rotate a die that shows 1 so that it shows 2, 3, 4 or 5. However, it cannot show 6 in a single move, because the faces with one pip and six pips visible are on opposite sides rather than adjacent.

You want to show the same number of pips on the top faces of all N dice. Given that each of the dice can be moved multiple times, count the minimum number of moves needed to get equal faces.

Write a function:

```
def solution(A)
```

that, given an array A consisting of N integers describing the number of pips (from 1 to 6) shown on each die's top face, returns the minimum number of moves necessary for each die to show the same number of pips.

For example, given:

- A = [1, 2, 3], the function should return 2, as you can pick the first two dice and rotate each of them in one move so that they all show three pips on the top face. Notice that you can also pick any other pair of dice in this case.
- A = [1, 1, 6], the function should also return 2. The only optimal answer is to rotate the last die so that it shows one pip. It is necessary to use two rotations to achieve this.
- A = [1, 6, 2, 3], the function should return 3. For instance, you can make all dice show 2: just rotate each die which is not showing 2 (and notice that for each die you can do this in one move).

Assume that:

- N is an integer within the range [1..100];
- each element of array A is an integer within the range [1..6].

In your solution, focus on **correctness**. The performance of your solution will not be the focus of the assessment.

Solution

Build an array of size 6.

```
array[0] = number of flip to get all dice showing 1
array[1] = number of flip to get all dice showing 2
array[2] = number of flip to get all dice showing 3
...
return min(array)
```

Q2. All bulbs on (variant of frog-leap question in Volant)

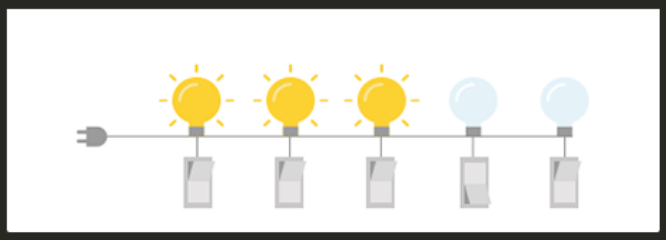
There are N bulbs, numbered from 1 to N, arranged in a row. The first bulb is plugged into the power socket and each successive bulb is connected to the previous one (the second bulb to the first, the third bulb to the second, etc.).

Initially, all the bulbs are turned off. At moment K (for K from 0 to N−1), we turn on the A[K]-th bulb. A bulb shines if it is on and all the previous bulbs are turned on too.

Write a function `solution` that, given an array A of N different integers from 1 to N, returns the number of moments for which every turned on bulb shines.

Examples:

1. Given A=[2, 1, 3, 5, 4], the function should return 3.



- At the 0th moment only the 2nd bulb is turned on, but it does not shine because the previous one is not on.
- At the 1st moment two bulbs are turned on (1st and 2nd) and both of them shine.
- At the 2nd moment three bulbs are turned on (1st, 2nd and 3rd) and all of them shine.
- At the 3rd moment four bulbs are turned on (1st, 2nd, 3rd and 5th), but the 5th bulb does not shine because the previous one is not turned on.
- At the 4th moment five bulbs are turned on (1st, 2nd, 3rd, 4th and 5th) and all five of them shine.

There are three moments (1st, 2nd and 4th) when every turned on bulb shines.

2. Given A=[2, 3, 4, 1, 5], the function should return 2 (at the 3rd and 4th moment every turned on bulb shines).

3. Given A=[1, 3, 4, 2, 5], the function should return 3 (at the 0th, 3rd and 4th moment every turned on bulb shines).

Write an **efficient** algorithm for the following assumptions:

- N is an integer within the range [1..100,000];
- the elements of A are all distinct;
- each element of array A is an integer within the range [1..N].

Here is my solution :

```python
def solution(A):
    # write your code in Python 3.6
    ans = 0
    pos = -1
    is_on = [False] * len(A)
    for n in range(len(A)) :

        is_on[A[n]-1] = True
        if pos+1 == A[n]-1 :
            pos = pos + 1
            while pos < len(A)-1  :
                if is_on[pos+1] : pos = pos + 1
                else : break

        if pos == n : ans = ans + 1
    return ans
```

Q3. Max depth of pit

A non-empty array A consisting of N integers is given. A *pit* in this array is any triplet of integers (P, Q, R) such that:

- $0 \leq P < Q < R < N$;
- sequence [A[P], A[P+1], ..., A[Q]] is strictly decreasing, i.e. A[P] > A[P+1] > ... > A[Q];
- sequence A[Q], A[Q+1], ..., A[R] is strictly increasing, i.e. A[Q] < A[Q+1] < ... < A[R].

The *depth* of a pit (P, Q, R) is the number **min**{A[P] − A[Q], A[R] − A[Q]}.

For example, consider array A consisting of 10 elements such that:

```
A[0] =  0
A[1] =  1
A[2] =  3
A[3] = -2
A[4] =  0
A[5] =  1
A[6] =  0
A[7] = -3
A[8] =  2
A[9] =  3
```

Triplet (2, 3, 4) is one of pits in this array, because sequence [A[2], A[3]] is strictly decreasing (3 > −2) and sequence [A[3], A[4]] is strictly increasing (−2 < 0). Its depth is **min**{A[2] − A[3], A[4] − A[3]} = 2. Triplet (2, 3, 5) is another pit with depth 3. Triplet (5, 7, 8) is yet another pit with depth 4. There is no pit in this array deeper (i.e. having depth greater) than 4.

Write a function:

```
def solution(A)
```

that, given a non-empty array A consisting of N integers, returns the depth of the deepest pit in array A. The function should return −1 if there are no pits in array A.

For example, consider array A consisting of 10 elements such that:

```
A[0] =  0
A[1] =  1
A[2] =  3
A[3] = -2
A[4] =  0
A[5] =  1
A[6] =  0
A[7] = -3
A[8] =  2
A[9] =  3
```

the function should return 4, as explained above.

Write an **efficient** algorithm for the following assumptions:

- N is an integer within the range [1..1,000,000];
- each element of array A is an integer within the range [−100,000,000..100,000,000].

Define 3 states : reset / downward / upward.
Define 3 trends : `A[n]<A[n-1]` / `A[n]>A[n-1]` / `A[n]==A[n-1]`

There are 3 x 3 combinations :

- 3 states
- 3 trends