Codility_

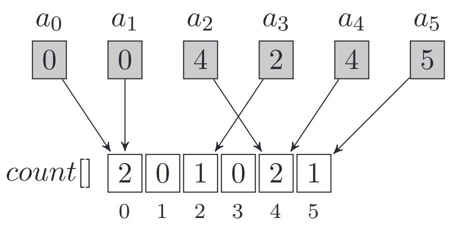
Chapter 4

Counting elements

A numerical sequence can be stored in an array in various ways. In the standard approach, the consecutive numbers $a_0, a_1, \ldots, a_{n-1}$ are usually put into the corresponding consecutive indices of the array:

$$A[0] = a_0 \quad A[1] = a_1 \quad \ldots \quad A[n-1] = a_{n-1}$$

We can also store the data in a slightly different way, by making an array of counters. Each number may be counted in the array by using an index that corresponds to the value of the given number.



Notice that we do not place elements directly into a cell; rather, we simply count their occurrences. It is important that the array in which we count elements is sufficiently large. If we know that all the elements are in the set $\{0, 1, \ldots, m\}$, then the array used for counting should be of size $m + 1$.

4.1: Counting elements — $O(n + m)$.

```
def counting(A, m):
    n = len(A)
    count = [0] * (m + 1)
    for k in xrange(n):
        count[A[k]] += 1
    return count
```

The limitation here may be available memory. Usually, we are not able to create arrays of 10^9 integers, because this would require more than one gigabyte of available memory.

Counting the number of negative integers can be done in two ways. The first method is to add some big number to each value: so that, all values would be greater than or equal to zero. That is, we shift the representation of zero by some arbitrary amount to accommodate all the negative numbers we need. In the second method, we simply create a second array for counting negative numbers.

4.1. Exercise

Problem: You are given an integer m ($1 \leqslant m \leqslant 1\,000\,000$) and two non-empty, zero-indexed arrays A and B of n integers, $a_0, a_1, \ldots, a_{n-1}$ and $b_0, b_1, \ldots, b_{n-1}$ respectively ($0 \leqslant a_i, b_i \leqslant m$).

The goal is to check whether there is a swap operation which can be performed on these arrays in such a way that the sum of elements in array A equals the sum of elements in array B after the swap. By swap operation we mean picking one element from array A and one element from array B and exchanging them.

Solution $O(n^2)$: The simplest method is to swap every pair of elements and calculate the totals. Using that approach gives us $O(n^3)$ time complexity. A better approach is to calculate the sums of elements at the beginning, and check only how the totals change during the swap operation.

4.2: Swap the elements — $O(n^2)$.

```
def slow_solution(A, B, m):
    n = len(A)
    sum_a = sum(A)
    sum_b = sum(B)
    for i in xrange(n):
        for j in xrange(n):
            change = B[j] - A[i]
            sum_a += change
            sum_b -= change
            if sum_a == sum_b:
                return True
            sum_a -= change
            sum_b += change
    return False
```

Solution $O(n + m)$: The best approach is to count the elements of array A and calculate the difference d between the sums of the elements of array A and B.

For every element of array B, we assume that we will swap it with some element from array A. The difference d tells us the value from array A that we are interested in swapping, because only one value will cause the two totals to be equal. The occurrence of this value can be found in constant time from the array used for counting.

4.3: Swap the elements — $O(n + m)$.

```
def fast_solution(A, B, m):
    n = len(A)
    sum_a = sum(A)
    sum_b = sum(B)
    d = sum_b - sum_a
    if d % 2 == 1:
        return False
    d //= 2
    count = counting(A, m)
    for i in xrange(n):
        if 0 <= B[i] - d and B[i] - d <= m and count[B[i] - d] > 0:
            return True
    return False
```

Every lesson will provide you with programming tasks at http://codility.com/programmers.