codility

WE TEST CODERS

Chapter 5

Prefix sums

There is a simple yet powerful technique that allows for the fast calculation of sums of elements in given slice (contiguous segments of array). Its main idea uses prefix sums which are defined as the consecutive totals of the first $0, 1, 2, \ldots, n$ elements of an array.

	a_0	a_1	a_2	...	a_{n-1}
$p_0 = 0$	$p_1 = a_0$	$p_2 = a_0 + a_1$	$p_3 = a_0 + a_1 + a_2$	...	$p_n = a_0 + a_1 + \ldots + a_{n-1}$

We can easily calculate the prefix sums in $O(n)$ time complexity. Notice that the total p_k equals $p_{k-1} + a_{k-1}$, so each consecutive value can be calculated in a constant time.

5.1: Counting prefix sums — $O(n)$.

```
def prefix_sums(A):
    n = len(A)
    P = [0] * (n + 1)
    for k in xrange(1, n + 1):
        P[k] = P[k - 1] + A[k - 1]
    return P
```

Similarly, we can calculate suffix sums, which are the totals of the k last values. Using prefix (or suffix) sums allows us to calculate the total of any slice of the array very quickly. For example, assume that you are asked about the totals of m slices $[x..y]$ such that $0 \leqslant x \leqslant y < n$, where the total is the sum $a_x + a_{x+1} + \ldots + a_{y-1} + a_y$.

The simplest approach is to iterate through the whole array for each result separately; however, that requires $O(n \cdot m)$ time. The better approach is to use prefix sums. If we calculate the prefix sums then we can answer each question directly in constant time. Let's subtract p_x from the value p_{y+1}.

p_{y+1}	a_0	a_1	...	a_{x-1}	a_x	a_{x+1}	...	a_{y-1}	a_y
p_x	a_0	a_1	...	a_{x-1}					
$p_{y+1} - p_x$					a_x	a_{x+1}	...	a_{y-1}	a_y

5.2: Total of one slice — $O(1)$.

```
def count_total(P, x, y):
    return P[y + 1] - P[x]
```

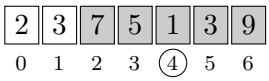
We have calculated the total of $a_x + a_{x+1} + \ldots + a_{y-1} + a_y$ in $O(1)$ time. Using this approach, the total time complexity is $O(n + m)$.

5.1. Exercise

Problem: You are given a non-empty, zero-indexed array A of n ($1 \leqslant n \leqslant 100\,000$) integers $a_0, a_1, \ldots, a_{n-1}$ ($0 \leqslant a_i \leqslant 1\,000$). This array represents number of mushrooms growing on the consecutive spots along a road. You are also given integers k and m ($0 \leqslant k, m < n$).

A mushroom picker is at spot number k on the road and should perform m moves. In one move she moves to an adjacent spot. She collects all the mushrooms growing on spots she visits. The goal is to calculate the maximum number of mushrooms that the mushroom picker can collect in m moves.

For example, consider array A such that:



The mushroom picker starts at spot $k = 4$ and should perform $m = 6$ moves. She might move to spots $3, 2, 3, 4, 5, 6$ and thereby collect $1 + 5 + 7 + 3 + 9 = 25$ mushrooms. This is the maximal number of mushrooms she can collect.

Solution $O(m^2)$: Note that the best strategy is to move in one direction optionally followed by some moves in the opposite direction. In other words, the mushroom picker should not change direction more than once. With this observation we can find the simplest solution. Make the first $p = 0, 1, 2, \ldots, m$ moves in one direction, then the next $m - p$ moves in the opposite direction. This is just a simple simulation of the moves of the mushroom picker which requires $O(m^2)$ time.

Solution $O(n+m)$: A better approach is to use prefix sums. If we make p moves in one direction, we can calculate the maximal opposite location of the mushroom picker. The mushroom picker collects all mushrooms between these extremes. We can calculate the total number of collected mushrooms in constant time by using prefix sums.

5.3: Mushroom picker — $O(n+m)$

```
def mushrooms(A, k, m):
    n = len(A)
    result = 0
    pref = prefix_sums(A)
    for p in xrange(min(m, k) + 1):
        left_pos = k - p
        right_pos = min(n - 1, max(k, k + m - 2 * p))
        result = max(result, count_total(pref, left_pos, right_pos))
    for p in xrange(min(m + 1, n - k)):
        right_pos = k + p
        left_pos = max(0, min(k, k - (m - 2 * p)))
        result = max(result, count_total(pref, left_pos, right_pos))
    return result
```

The total time complexity of such a solution is $O(n + m)$.

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