codility
WE TEST CODERS

Chapter 1

Iterations

In programming, iterating means repeating some part of your program. This lesson presents basic programming constructions that allow iterations to be performed: "for" and "while" loops.

1.1. For loops

If you want to repeat some operations a given number of times, or repeat them for each element in some collection, a "for" loop is the right tool to use. Its syntax is as follows:

1.1: For loop syntax

```
1  for some_variable in range_of_values:
2      loop_body
```

The for loop repeats `loop_body` for each value in turn from the `range_of_values`, with the current value assigned to `some_variable`. In its simplest form, the range of values can be a range of integers, denoted by: `range(lowest, highest + 1)`. For example, the following loop prints every integer from 0 to 99:

```
1  for i in range(0, 100):
2      print i
```

Looping over a range of integers starting from 0 is a very common operation. (This is mainly because arrays and Python lists are indexed by integers starting from 0; see Chapter 2 Arrays for more details.) When specifying the range of integers, if the starting value equals zero then you can simply skip it. For example, the following loop produces exactly the same result as the previous one:

```
1  for i in range(100):
2      print i
```

Example: We are given some positive integer n. Let's compute the factorial of n and assign it to the variable `factorial`. The factorial of n is $n! = 1 \cdot 2 \cdot \ldots \cdot n$. We can obtain it by starting with 1 and multiplying it by all the integers from 1 to n.

```
1  factorial = 1
2  for i in range (1, n + 1):
3      factorial *= i
```

Example: Let's print a triangle made of asterisks ('$\star$') separated by spaces. The triangle should consist of n rows, where n is a given positive integer, and consecutive rows should contain 1, 2, ..., n asterisks. For example, for $n = 4$ the triangle should appear as follows:

```
*
* *
* * *
* * * *
```

We need to use two loops, one inside the other: the outer loop should print one row in each step and the inner loop should print one asterisk in each step[2].

```
1 for i in range(1, n + 1):
2     for j in range(i):
3         print '*',
4     print
```

The `range` function can also accept one more argument specifying the step with which the iterated values progress. More formally, `range(start, stop, step)` is a sequence of values beginning with `start`, whose every consecutive value is increased by `step`, and that contains only values smaller than `stop` (for positive `step`; or greater than `stop` for negative `step`). For example, `range(10, 0, -1)` represents sequence 10, 9, 8, ..., 1. Note that we cannot omit `start` when we specify `step`.

Example: Let's print a triangle made of asterisks ('$\star$') separated by spaces and consisting of n rows again, but this time upside down, and make it symmetrical. Consecutive rows should contain $2n - 1$, $2n - 3$, ..., 3, 1 asterisks and should be indented by 0, 2, 4, ..., $2(n - 1)$ spaces. For example, for $n = 4$ the triangle should appear as follows:

```
* * * * * * *
  * * * * *
    * * *
      *
```

The triangle should have n rows, where n is some given positive integer.

This time we will use three loops: one outer and two inner loops. The outer loop in each step prints one row of the triangle. The first inner loop is responsible for printing the indentations, and the second for printing the asterisks.

```
1 for i in range(n, 0, -1):
2     for j in range(n - i):
3         print ' ',
4     for j in range(2 * i - 1):
5         print '*',
6     print
```

1.2. While loops

The number of steps in a for loop, and the values over which we loop, are fixed before the loop starts. What if the number of steps is not known in advance, or the values over which

[2]There is a clever idiom in Python denoting a string repeated a number of times. For example, `'*' * n` denotes a string comprising n asterisks. But to make our examples more instructive, we will not use this idiom here. Instead we will use the `print` statement ended with a comma. It doesn't print the newline character, but follows the output with a single space.

we loop are generated one by one, and are thus not known in advance either? In such a case, we have to use a different kind of loop, called a "while" loop. The syntax of the while loop is as follows:

1.2: While loop syntax

```
while some_condition:
    loop_body
```

Before each step of the loop, `some_condition` is computed. As long as its value is true[3], the body of the loop is executed. Once it becomes false, we exit the loop without executing `loop_body`.

Example: Given a positive integer n, how can we count the number of digits in its decimal representation? One way to do it is convert the integer into a string and count the characters. Here, though, we will use only arithmetical operations instead. We can simply keep dividing the number by ten and count how many steps are needed to obtain 0.

```
result = 0
while n > 0:
    n = n // 10
    result += 1
```

Example: The Fibonacci numbers[4] form a sequence of integers defined recursively in the following way. The first two numbers in the Fibonacci sequence are 0 and 1, and each subsequent number is the sum of the previous two. The first few elements in this sequence are: 0, 1, 1, 2, 3, 5, 8, 13. Let's write a program that prints all the Fibonacci numbers, not exceeding a given integer n.

We can keep generating and printing consecutive Fibonacci numbers until we exceed n. In each step it's enough to store only two consecutive Fibonacci numbers.

```
a = 0
b = 1
while a <= n:
    print a
    c = a + b
    a = b
    b = c
```

1.3. Looping over collections of values[5]

We have seen how to loop over integers. Is it possible to loop over values of other types? Yes: using a "for" loop, we can loop over values stored in virtually any kind of container. The `range` function constructs a list containing all the values over which we should loop. However, we can pass a list constructed in any other way.

[3]Note that the condition can yield any value, not only `True` or `False`. A number of values other than `False` are interpreted as false, e.g. `None`, `0`, `[]` (empty list) and `''` (empty string).

[4]You can read more about the Fibonacci numbers in Chapter 13.

[5]If you are not familiar with various built-in data structures such as lists, sets and dictionaries, you can safely skip this section. Looping over lists of values will be explained in Chapter 2.

Example: The following program:

```
1 days = ['Monday', 'Tuesday', 'Wednesday', 'Thursday',
2         'Friday', 'Saturday', 'Sunday']
3 for day in days:
4     print day
```

prints all the days of the week, one per line.

When we use the `range` function, we first build a list of all the integers over which we will loop; then we start looping. This is memory-consuming when looping over a long sequence. In such cases, it's advisable to use – instead of `range` – an equivalent function called `xrange`. This returns exactly the same sequence of integers, but instead of storing them in a list, it returns an object that generates them on the fly.

If you loop over a set of values, the body of the loop is executed exactly once for every value in the set; however, the order in which the values are processed is arbitrary.

Example: If we modify the above program slightly, as follows:

```
1 days = set(['Monday', 'Tuesday', 'Wednesday', 'Thursday',
2             'Friday', 'Saturday', 'Sunday'])
3 for day in days:
4     print day
```

we might get the days output in some strange order, e.g.:

```
Monday
Tuesday
Friday
Wednesday
Thursday
Sunday
Saturday
```

Looping over a dictionary means looping over its set of keys. Again, the order in which the keys are processed is arbitrary.

Example: The following program:

```
1 days = {'mon': 'Monday', 'tue': 'Tuesday', 'wed': 'Wednesday',
2         'thu': 'Thursday', 'fri': 'Friday', 'sat': 'Saturday',
3         'sun': 'Sunday'}
4 for day in days:
5     print day, 'stands for', days[day]
```

might output e.g.:

```
wed stands for Wednesday
sun stands for Sunday
fri stands for Friday
tue stands for Tuesday
mon stands for Monday
thu stands for Thursday
sat stands for Saturday
```

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

codility

WE TEST CODERS

Chapter 2

Arrays

Array is a data-structure that can be used to store many items in one place. Imagine that we have a list of items; for example, a shopping list. We don't keep all the products on separate pages; we simply list them all together on a single page. Such a page is conceptually similar to an array. Similarly, if we plan to record air temperatures over the next 365 days, we would not create lots of individual variables, but would instead store all the data in just one array.

2.1. Creating an array

We want to create a shopping list containing three products. Such a list might be created as follows:

```
shopping = ['bread', 'butter', 'cheese']
```

(that is, `shopping` is the name of the array and every product within it is separated by a comma). Each item in the array is called an element. Arrays can store any number of elements (assuming that there is enough memory). Note that a list can be also empty:

```
shopping = []
```

If planning to record air temperatures over the next 365 days, we can create in advance a place to store the data. The array can be created in the following way:

```
temperatures = [0] * 365
```

(that is, we are creating an array containing 365 zeros).

2.2. Accessing array values

Arrays provide easy access to all elements. Within the array, every element is assigned a number called an index. Index numbers are consecutive integers starting from 0. For example, in the array `shopping = ['bread', 'butter', 'cheese']`, 'bread' is at index 0, 'butter' is at index 1 and 'cheese' is at index 2. If we want to check what value is located at some index (for example, at index 1), we can access it by specifing the index in square brackets, e.g. `shopping[1]`.

2.3. Modifying array values

We can change array elements as if they were separate variables, that is each array element can be assigned a new value independently. For example, let's say we want to record that on the 42nd day of measurement, the air temperature was 25 degrees. This can be done with a single assignment:

```
temperatures[42] = 25
```

If there was one more product to add to our shopping list, it could be appended as follows:

```
shopping += ['eggs']
```

The index for that element will be the next integer after the last (in this case, 3).

2.4. Iterating over an array

Often we need to iterate over all the elements of an array; perhaps to count the number of specified items, for example. Knowing that the array contains of N elements, we can iterate over consecutive integers from index 0 to index $N-1$ and check every such index. The length of an array can be found using the `len()` function. For example, counting the number of items in shopping list can be done quickly as follows:

```
N = len(shopping)
```

Let's write a function that counts the number of days with negative air temperature.

2.1: Negative air temperature.

```
def negative(temperatures):
    N = len(temperatures)
    days = 0
    for i in xrange(N):
        if temperatures[i] < 0:
            days += 1
    return days
```

Instead of iterating over indexes, we can iterate over the elements of the array. To do this, we can simply write:

```
for item in array:
    ...
```

For example, the above solution can be simplified as follows:

2.2: Negative air temperature — simplified.

```
def negative(temperatures):
    days = 0
    for t in temperatures:
        if t < 0:
            days += 1
    return days
```

In the above solution, for every temperature, we increase the number of days with a negative temperature if the number is lower than zero.

2.5. Basic array operations

There are a few basic operations on arrays that are very useful. Apart from the length operation:

```
len([1, 2, 3]) == 3
```

and the repetition:

```
['Hello'] * 3 == ['Hello', 'Hello', 'Hello']
```

which we have already seen, there is also concatenation:

```
[1, 2, 3] + [4, 5, 6] == [1, 2, 3, 4, 5, 6]
```

which merges two lists, and the membership operation:

```
'butter' in ['bread', 'butter', 'cheese'] == True
```

which checks for the presence of a particular item in the array.

2.6. Exercise

Problem: Given array A consisting of N integers, return the reversed array.

Solution: We can iterate over the first half of the array and exchange the elements with those in the second part of the array.

2.3: Reversing an array.

```
1 def reverse(A):
2     N = len(A)
3     for i in xrange(N // 2):
4         k = N - i - 1
5         A[i], A[k] = A[k], A[i]
6     return A
```

Python is a very rich language and provides many built-in functions and methods. It turns out, that there is already a built-in method `reverse`, that solves this exercise. Using such a method, array A can be reversed simply by:

```
1 A.reverse()
```

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

codility

WE TEST CODERS

Chapter 3

Time complexity

Use of time complexity makes it easy to estimate the running time of a program. Performing an accurate calculation of a program's operation time is a very labour-intensive process (it depends on the compiler and the type of computer or speed of the processor). Therefore, we will not make an accurate measurement; just a measurement of a certain order of magnitude.

Complexity can be viewed as the maximum number of primitive operations that a program may execute. Regular operations are single additions, multiplications, assignments etc. We may leave some operations uncounted and concentrate on those that are performed the largest number of times. Such operations are referred to as *dominant*.

The number of dominant operations depends on the specific input data. We usually want to know how the performance time depends on a particular aspect of the data. This is most frequently the data size, but it can also be the size of a square matrix or the value of some input variable.

3.1: Which is the dominant operation?

```
1  def dominant(n):
2      result = 0
3      for i in xrange(n):
4          result += 1
5      return result
```

The operation in line 4 is dominant and will be executed n times. The complexity is described in Big-O notation: in this case $O(n)$ — *linear* complexity.

The complexity specifies the order of magnitude within which the program will perform its operations. More precisely, in the case of $O(n)$, the program may perform $c \cdot n$ operations, where c is a constant; however, it may not perform n^2 operations, since this involves a different order of magnitude of data. In other words, when calculating the complexity we omit constants: i.e. regardless of whether the loop is executed $20 \cdot n$ times or $\frac{n}{5}$ times, we still have a complexity of $O(n)$, even though the running time of the program may vary. When analyzing the complexity we must look for specific, worst-case examples of data that the program will take a long time to process.

3.1. Comparison of different time complexities

Let's compare some basic time complexities.

3.2: Constant time — $O(1)$.

```
def constant(n):
    result = n * n
    return result
```

There is always a fixed number of operations.

3.3: Logarithmic time — $O(\log n)$.

```
def logarithmic(n):
    result = 0
    while n > 1:
        n //= 2
        result += 1
    return result
```

The value of n is halved on each iteration of the loop. If $n = 2^x$ then $\log n = x$. How long would the program below take to execute, depending on the input data?

3.4: Linear time — $O(n)$.

```
def linear(n, A):
    for i in xrange(n):
        if A[i] == 0:
            return 0
    return 1
```

Let's note that if the first value of array A is 0 then the program will end immediately. But remember, when analyzing time complexity we should check for worst cases. The program will take the longest time to execute if array A does not contain any 0.

3.5: Quadratic time — $O(n^2)$.

```
def quadratic(n):
    result = 0
    for i in xrange(n):
        for j in xrange(i, n):
            result += 1
    return result
```

The result of the function equals $\frac{1}{2} \cdot (n \cdot (n+1)) = \frac{1}{2} \cdot n^2 + \frac{1}{2} \cdot n$ (the explanation is in the exercises). When calculating the complexity we are interested in a term that grows fastest, so we not only omit constants, but also other terms ($\frac{1}{2} \cdot n$ in this case). Thus we get quadratic time complexity. Sometimes the complexity depends on more variables (see example below).

3.6: Linear time — $O(n+m)$.

```
def linear2(n, m):
    result = 0
    for i in xrange(n):
        result += i
    for j in xrange(m):
        result += j
    return result
```

Exponential and factorial time

It is worth knowing that there are other types of time complexity such as factorial time $O(n!)$ and exponential time $O(2^n)$. Algorithms with such complexities can solve problems only for very small values of n, because they would take too long to execute for large values of n.

3.2. Time limit

Nowadays, an average computer can perform 10^8 operations in less than a second. Sometimes we have the information we need about the expected time complexity (for example, Codility specifies the expected time complexity), but sometimes we do not.

The time limit set for online tests is usually from 1 to 10 seconds. We can therefore estimate the expected complexity. During contests, we are often given a limit on the size of data, and therefore we can guess the time complexity within which the task should be solved. This is usually a great convenience because we can look for a solution that works in a specific complexity instead of worrying about a faster solution. For example, if:

- $n \leqslant 1\,000\,000$, the expected time complexity is $O(n)$ or $O(n \log n)$,
- $n \leqslant 10\,000$, the expected time complexity is $O(n^2)$,
- $n \leqslant 500$, the expected time complexity is $O(n^3)$.

Of course, these limits are not precise. They are just approximations, and will vary depending on the specific task.

3.3. Space complexity

Memory limits provide information about the expected space complexity. You can estimate the number of variables that you can declare in your programs.

In short, if you have constant numbers of variables, you also have constant space complexity: in Big-O notation this is $O(1)$. If you need to declare an array with n elements, you have linear space complexity — $O(n)$.

More specifically, space complexity is the amount of memory needed to perform the computation. It includes all the variables, both global and local, dynamic pointer data-structures and, in the case of recursion, the contents of the stack. Depending on the convention, input data may also be included. Space complexity is more tricky to calculate than time complexity because not all of these variables and data-structures may be needed at the same time. Global variables exist and occupy memory all the time; local variables (and additional information kept on the stack) will exist only during invocation of the function.

3.4. Exercise

Problem: You are given an integer n. Count the total of $1 + 2 + \ldots + n$.

Solution: The task can be solved in several ways. Some person, who knows nothing about time complexity, may implement an algorithm in which the result is incremented by 1:

3.7: Slow solution — time complexity $O(n^2)$.

```
def slow_solution(n):
    result = 0
    for i in xrange(n):
        for j in xrange(i + 1):
            result += 1
    return result
```

Another person may increment the result respectively by $1, 2, \ldots, n$. This algorithm is much faster:

3.8: Fast solution — time complexity $O(n)$.

```
def fast_solution(n):
    result = 0
    for i in xrange(n):
        result += (i + 1)
    return result
```

But the third person's solution is even quicker. Let us write the sequence $1, 2, \ldots, n$ and repeat the same sequence underneath it, but in reverse order. Then just add the numbers from the same columns:

1	2	3	$\ldots$	$n-1$	n
n	$n-1$	$n-2$	$\ldots$	2	1
$n+1$	$n+1$	$n+1$	$\ldots$	$n+1$	$n+1$

The result in each column is $n + 1$, so we can easily count the final result:

3.9: Model solution — time complexity $O(1)$.

```
def model_solution(n):
    result = n * (n + 1) // 2
    return result
```

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

codility
WE TEST CODERS

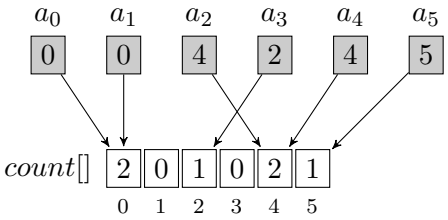
Chapter 4

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 — $O(n + m)$.

```
1  def counting(A, m):
2      n = len(A)
3      count = [0] * (m + 1)
4      for k in xrange(n):
5          count[A[k]] += 1
6      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`.

Codility

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]
```

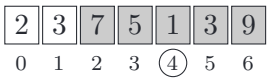
We have calculated the total of $a_x + a_{x+1} + \ldots + a_{y-1} + a_y$ in $O(1)$ time. Using this approach, the total time complexity is $O(n + m)$.

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`.