Overview

This diagnostic test asks you to write programs to solve unfamiliar problems. They are not as hard as USACO Silver problems, but solving them will show that you are ready to start wrestling with Silver-level material.

For this test – and as a USACO Silver student – we expect you to be able to write and run programs in C++, Java, or Python on your computer, or in a coding environment online.

The test has two problems. For each problem, we present a description followed by a sample case. Then, we list four test cases; your goal is to write a program that can compute the correct output for each of them. The first test case for each of our problems is always solvable by hand (and you are welcome to do this as a secondary check), but the other test cases will require you to code!

It's up to you how to structure your code for each problem, and you can run it as many times as you want – you can decide if/when you think it is correct. As in a programming contest, you will not be scored on coding style – just on the final outputs, which you should write down. Your code must be able to produce answers to the problems within the usual USACO time limits: 2 seconds for C++, 4 seconds for Java and Python.

Just like in the USACO rules, you are free to consult reference material for the programming language you are using. Unlike in the USACO, we are not setting a time limit for this diagnostic test, but we anticipate that the test will take anywhere from 30 to 90 minutes.

The test begins on the next page, and solutions appear at the end. Good luck!

1. Bessie the cow has a list of N nonnegative integers $(1 \leq N \leq 10^5)$, each of which is between 0 and $K-1$, inclusive ($1 \leq K \leq 10^5$). She wants to find the length of a longest possible sublist without any repeats. For example, in the sublist $[1, 4, 9, 6, 1]$, the longest sublists without any repeats are $[1, 4, 9, 6]$ and $[4, 9, 6, 1]$, both of which have length 4.

   Your code will have to generate each list using this rule: the i^{th} number in Bessie's list (counting starting from 1) is $(i^3 - i^2)$ modulo K. (That is, take the remainder of $i^3 - i^2$ when it is divided by K; you can use the % operator in your language.) Think of this as just a way to avoid making you copy-paste large inputs or read from a file – the problem doesn't have anything to do with $i^3 - i^2$ per se.

   Your task: Write a program that takes in one line with N and one line with K, and outputs the answer, as described above.

   Sample input:
   6
   7

   Correct output:

   4

   For $N = 6$ and $K = 7$, the values of $i^3 - i^2$ for $i = 1, 2, 3, 4, 5, 6$ are

   $$1 - 1 = 0, 8 - 4 = 4, 27 - 9 = 18, 64 - 16 = 48, 125 - 25 = 100, 216 - 36 = 180$$

   Taking these modulo 7 gives $0, 4, 4, 6, 2, 5$. The longest sublist with no repeated elements is $4, 6, 2, 5$, and this has length 4.

   The test cases are on the next page.

Test case 1:

```
11
10
```

Your program's output:

Test case 2:

```
100
777
```

Your program's output:

Test case 3:

```
1250
99997
```

Your program's output:

Test case 4:

```
100000
98765
```

Your program's output:

2. Bessie is about to be dropped into an $N \times N$ grid of cells $(1 \leq N \leq 1000)$. Each cell contains an arrow pointing in one of the four cardinal directions. Specifically, the cell in row i and column j (with both the row and column counted starting from 0) contains an arrow according to the value of $(ij + i + 2j)$ modulo K (with $4 \leq K \leq 100$, K divisible by 4) – the arrow points down, left, right, or up according to whether the value is in the range $[0, \frac{K}{4}), [\frac{K}{4}, \frac{K}{2}), [\frac{K}{2}, \frac{3K}{4})$, or $[\frac{3K}{4}, K)$. (This is a lot to parse, but the sample case below should help to clarify!)

   When Bessie is in a grid cell, she makes a unit move in the direction of the arrow on that cell. This might cause her to fall off the grid. (Don't worry – she lands safely in a comfortable pile of hay!) On the other hand, the cell's arrow might move her to another cell, which has its own arrow that she must follow, and so on. She might eventually fall off, or she might stay on the grid an infinitely long time.

   Of the N^2 possible starting positions for Bessie, count how many of them result in her staying on the grid.

   Your task: Write a program that takes in one line with N and one line with K, and outputs the answer, as described above.

   Sample input:

   5
   8

   Correct output:

   6

   The values for the 25 cells are

0	2	4	6	8
1	4	7	10	13
2	6	10	14	18
3	8	13	18	23
4	10	16	22	28

   . Modulo 8, these are:

0	2	4	6	0
1	4	7	2	5
2	6	2	6	2
3	0	5	2	7
4	2	0	6	4

   Turning these into directions, we get:

↓	←	→	↑	↓
↓	→	↑	←	→
←	↑	←	↑	←
←	↓	→	←	↑
→	←	↓	↑	→

   For a total of 6 of these cells, Bessie will stay on the grid if she starts there. Starting in any other cell will eventually cause her to fall off.

   The test cases are on the next page.

Test case 1:

```
3
4
```

Your program's output:

Test case 2:

```
50
76
```

Your program's output:

Test case 3:

```
999
4
```

Your program's output:

Test case 4:

```
1000
8
```

Your program's output:

Don't look at the next page until you've attempted all the problems!

Answers, Scoring Guide, and Explanations

Answers that took longer than the USACO time limit (2 seconds for C++, 4 seconds for Java and Python) to output should **not** be counted even if they are correct.

Answers

1.
   - Test case 1: 3
   - Test case 2: 59
   - Test case 3: 407
   - Test case 4: 845 (C++ users might have had an overflow bug here)
2.
   - Test case 1: 3
   - Test case 2: 2106
   - Test case 3: 808420
   - Test case 4: 732750

Scoring Guide

- If you got 7-8 of these test cases right, you have a solid foundation for our USACO Silver course.
- If you got 5-6 of these test cases right (or could do so after fixing some small bugs), you probably have the tools you need to succeed in our USACO Silver course... but more practice always helps. Our USACO Silver course may still be a good fit for you if you are self-motivated and you have enough time to devote to working hard in the class.
- If you got 4 or fewer of the test cases right, our USACO Silver course may be too challenging for you at the moment. We recommend considering our USACO Bronze course, which can be quite useful even if you are already able to solve some real Bronze problems. We also suggest checking out our explanations of these problems.

Explanations and example solutions appear starting on the next page.

In **Problem 1**, a brute-force solution is to just check all possible intervals: for every possible starting point, for every possible ending point, look through the elements in that interval and see (e.g., using a set) whether there are any duplicates. Then we take the longest length of any interval with no duplicates.

Unfortunately, this runs in cubic $O(N^3)$ time, and is too slow to pass the last two test cases. So we need to be more clever about which intervals we check.

This problem is well-suited to a "two pointers" strategy. The word "pointer" is used here more in the sense of "index", not in the sense of a C++ pointer to an address in memory; think of them as just being arrows that point to elements of our list. We start with both pointers pointing at index 0, and then alternate between the following:

- Move the right pointer ahead until one of two things happen:
  - Adding the next element would cause us to have a duplicate in our interval; call this element the "blocker". At this point, we stop moving the right pointer, and update our best interval length seen so far (since the current interval is valid).
  - We reach the end of the list. At this point we can update our best interval length seen so far, and then stop overall and return that answer. (All we can do at this point would be move the left pointer ahead, and that would just give us a smaller answer.)
- Move the left pointer ahead until it is past the element that would have caused the "blocker" to be a duplicate. This frees up the right arrow to begin advancing again.

To efficiently keep track of which elements are in our interval, we maintain a set. This allows us to quickly check whether the next element in the list would be a duplicate. We just need to make sure we add to our set when we advance the right pointer, and remove from our set when we advance the left pointer.

The only remaining piece of the puzzle is to convince ourselves that this algorithm can't fail to find an optimal answer. Here's an informal explanation. Consider any optimally large interval in the list from some index i to some index j. Since this interval is optimally large, either j is the end of the list, or the element at index $j+1$ must be a duplicate (or else we could have gotten an even larger interval by extending our supposedly "optimal" interval to include it). In either case, our right pointer will definitely stop at j at some point during the algorithm. Since the left pointer only moves forward when it absolutely has to (to allow the right pointer to move past a blocker), our left pointer can never be farther ahead than it "should" be relative to the optimal answer. So at the time our algorithm stops its right pointer at index j, its left pointer is at i – we can't miss this answer. (Even if more than one optimal interval exists in the list, the preceding argument applies to all of them.)

A Python implementation appears on the next page.


```python
n = int(input())
k = int(input())
ls = [(i**3 - i**2) % k for i in range(1, n+1)]

# Our strategy will be to start with an interval consisting
#   of just the first element...
left_ptr = 0
right_ptr = 0
# ...and then repeatedly expand the right end until we find
#   a duplicate, and contract the left end until we are past
#   the element that would cause the duplicate.
best = 1
s = set()
s.add(ls[0])
while True:
    # Expand right end as long as it would not cause us to
    #   have a duplicate
    while right_ptr < n-1 and ls[right_ptr+1] not in s:
        right_ptr += 1
        s.add(ls[right_ptr])
    best = max(best, right_ptr - left_ptr + 1)
    # If the right end is all the way at the end of the list,
    #   there's no need to continue, because we can only
    #   contract the left end from here, which would just give
    #   us a smaller answer that couldn't be a new best.
    if right_ptr == n-1:
        break
    # Otherwise the element just beyond our right end is the
    #   "blocker" that would have been a duplicate.
    blocker = ls[right_ptr+1]
    # Contract the left end until we are past the element the
    #   blocker would have duplicated.
    matched_blocker = False
    while not matched_blocker:
        curr = ls[left_ptr]
        matched_blocker = (curr == blocker)
        s.remove(curr)
        left_ptr += 1

print(best)
```


In **Problem 2**, a brute force solution is to try every starting point, simulate Bessie's movement starting from there, and see whether she stays on or falls off. A complication is that if Bessie stays on, she will loop around forever, so if we just simulate that, our code will run forever! One way to handle this is to reason that Bessie can't possibly take more than N^2 steps without revisiting a grid cell, so if she is still on the board after at least that many steps, we know she is looping forever and we can stop simulating. Another is to have Bessie remember where she has been (e.g., using a set), and stop if she sees that her current cell is one she has already been in.

For an algorithm fast enough to pass all the cases, we'll need a global version of the "have Bessie remember what she has learned" idea. We start simulating from some starting point, once we either detect a loop or see that Bessie falls off the edge, we can label every cell we took along the way to indicate that we know those are all "safe" or all "unsafe". If at any point we encounter a labeled cell, we have our answer for that simulation and we don't need to proceed further (but we should still label the trail of cells taken to get to that labeled cell).

With this strategy in place, we can go through the grid checking every possible starting point. Although some individual checks might take time proportional to the number of cells in the grid, we never traverse (i.e. go into and then out of) any cell more than once, which puts a limit on how much total work our solution can possibly do.

A Python implementation appears on the next page.


```python
n = int(input())
k = int(input())

# For clarity, we're going to store the grid as an array of
#   symbols v < > ^ meaning down, left, right, up.
grid = [[None for __ in range(n)] for _ in range(n)]
for i in range(n):
    for j in range(n):
        v = (i*j + i + 2*j) % k
        if 0 <= v < k/4:
            arrow = 'v'
        elif k/4 <= v < k/2:
            arrow = '<'
        elif k/2 <= v < (3*k)/4:
            arrow = '>'
        else:
            arrow = '^'
        grid[i][j] = arrow

# For each possible starting cell, this holds:
# * True if we know that Bessie stays on if she starts here.
# * False if we know that Bessie falls off if she starts here.
# * None if we don't know the answer yet.
mem = [[None for __ in range(n)] for _ in range(n)]

def stays_on(i, j):
    r = i
    c = j

    # The set of grid cells seen during this exploration.
    sofar_set = set()

    # Continue following the current arrow until one of the
    #   following happens:
    # 1. We reach a cell for which we already know the answer,
    #   in which case we return that answer.
    # 2. We reach a cell we have seen on this exploration,
    #   which implies that Bessie will loop forever and thus
    #   stays on.
    # 3. Bessie falls off the grid.
    stays_on = None
    while True:
        sofar_set.add((r, c))
        if grid[r][c] == 'v':
            r += 1
        elif grid[r][c] == '<':
            c -= 1
        elif grid[r][c] == '>':
            c += 1
        else: # ^
            r -= 1
        if not ((0 <= r < n) and (0 <= c < n)): # Bessie fell off
            stays_on = False
            break
        elif mem[r][c] is not None: # we've been here, we know how this ends
            stays_on = mem[r][c]
            break
        elif (r, c) in sofar_set:
            stays_on = True
            break # Bessie will loop forever
    # Since we know that all cells in our chain point to the same result,
    #   we can remember that result.
    for r, c in sofar_set:
        mem[r][c] = stays_on
    return stays_on

# Now just do this for every cell.
total = 0
for i in range(n):
    for j in range(n):
        if mem[i][j] is not None:
            if mem[i][j]:
                total += 1
        else:
            total += stays_on(i, j)

print(total)
```