Overview

This diagnostic test asks you to write programs to solve unfamiliar problems. They are not as hard as USACO Bronze problems, but solving them will show that you are ready to wrestle with Bronze problems!

For this test – and as a USACO Bronze student – you will need to be able to write and run basic programs in C++, Java, or Python on your computer, or in a coding environment online. If you're not already familiar with how to do this, then you should take the test later, after getting more practice with a language of your choice.

The test has two problems. For each problem, we present a description followed by one or more sample cases. Then, we list some test cases. Your goal is to write a program that can compute the correct output for each test case. 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. Although we are not scoring you on how fast your code runs, your code should be able to produce answers to the problems within a few seconds at most.

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 60 minutes.

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

1. Bessie the cow is playing a game to pass the time. She starts with a list of N elements (N is at least 3), in which the elements are $[1, 4, 9, \ldots, N^2]$. For example, for $N = 5$, the list would be $[1, 4, 9, 16, 25]$.

   Then she repeats the following K times:

   - Find the largest value in the list. (If there is a tie, choose the rightmost of the tied values.)
   - Divide that value by 2, rounding down to the nearest integer.

   Your task: Write a program that takes N and K, and outputs the final three elements in the list once this process is complete.

   - Sample input: $N = 3, K = 6$
   - Correct output: $1, 1, 0$

   The list evolves as follows: $[1, 4, 9] \to [1, 4, 4] \to [1, 4, 2] \to [1, 2, 2] \to [1, 2, 1] \to [1, 1, 1] \to [1, 1, 0]$. (You do not need to provide this kind of explanation for your own outputs; we provide them for the sample cases here just for clarity.)

   - Sample input: $N = 4, K = 4$
   - Correct output: $4, 4, 2$

   The list evolves as follows: $[1, 4, 9, 16] \to [1, 4, 9, 8] \to [1, 4, 4, 8] \to [1, 4, 4, 4] \to [1, 4, 4, 2]$.

   - Test case 1: $N = 5, K = 7$
   - Your program's output: _____ , _____ , _____

   - Test case 2: $N = 100, K = 584$
   - Your program's output: _____ , _____ , _____

   - Test case 3: $N = 1000, K = 2195$
   - Your program's output: _____ , _____ , _____

2. Let's define a *squbе* as a number that can be written as $a^2 + b^3$, where a and b are different positive integers. For example:
   - 52 is a squbе, since it can be written as $5^2 + 3^3$.
   - 17 is a squbе. It can actually be written in two ways: as $3^2 + 2^3$, or as $4^2 + 1^3$.
   - 4 and 8 are not squbеs, even though 4 is a square and 8 is a cube. Even though 0 is both a square and a cube, notice that the problem requires both a and b to be positive integers.
   - 2 is not a squbе. It could be written as $1^2 + 1^3$, but then we'd have $a = 1$ and $b = 1$, and those are not different numbers.

   The first eight squbеs are $5, 9, 10, 17, 24, 26, 28, 31$.

   Your task: Write a program that takes a number K and finds the K^{th} squbе.

   Note: We recommend that you write a solution that uses only integers and never has to take a square root or cube root. (Doing that would use floating-point numbers, which do not always exactly equal the numbers they are trying to represent, and this can potentially lead to errors.)

   - Sample case 1: $K = 5$
   - Correct output: 24

   - Test case 1: $K = 9$
   - Your program's output: _____

   - Test case 2: $K = 99$
   - Your program's output: _____

   - Test case 3: $K = 999$
   - Your program's output: _____

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

Answers, Scoring Guide, and Explanations

Answers

1.
   - Test case 1: $4, 2, 3$
   - Test case 2: $18, 19, 19$
   - Test case 3: $62250, 31187, 31250$
2.
   - Test case 1: 33
   - Test case 2: 424
   - Test case 3: 5988

Scoring Guide

- If you got all of these test cases right, you have an excellent foundation for our USACO Bronze course.
- If you got at least 4 of these 6 test cases right (or could do so after fixing some small bugs), you probably have the tools you need to succeed in USACO Bronze... but more practice always helps. Our USACO Bronze 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 3 or fewer of the test cases right, USACO Bronze is probably too challenging for you at the moment. We recommend reading the solutions carefully, and getting some further practice with your favorite programming language.

Explanations (along with sample Python code) appear on the remaining pages.

Explanations

1. Here we just need to follow the rules and see what happens. We create the initial list and then repeat the following K times: go through it, find the maximum value (breaking ties by taking the rightmost of the tied values), and divide it by 2. The integer division operator (`/` in C++ and Java, `//` in Python) conveniently already has the "round down to the nearest integer" behavior we want.

```
def solve(n, k):
  arr = []
  for i in range(1, n+1):
    arr.append(i**2)
  for _ in range(k):
    # Find the maximum value in the array.
    # Set the starting "maximum" to be something we
    # know is smaller than any actual array value,
    # so we can be sure it gets overwritten.
    mx = -1
    mx_index = None # This will also be overwritten.
    for i in range(len(arr)):
      # We use >= instead of = here so that
      # the rightmost value wins in a tie.
      if arr[i] >= mx:
        mx = arr[i]
        mx_index = i
    # Use // to get "floor division"
    # (i.e., integer division that rounds down)
    arr[mx_index] //= 2
  # Return the last three elements of the array.
  return(arr[-3:])
```

This solution has to go through the entire list every time to find the maximum value. That turns out to be fine for the test cases in this problem, but it would be way too slow for, say, $N = 100000, K = 1000000$. If you found yourself wondering whether there might be a way to avoid looping through the list every time, then we hope that you eventually take CodeWOOT, which introduces a data structure called a *priority queue* that is perfect for this situation!

2. This problem is pretty different from the previous one, which presented some rules and asked you to "simulate" them. Here it's not very clear how to get started!

   Suppose we had a function `is_sqube(x)` that took a number x and determined whether or not it was a sqube. Then, to find the K^{th} sqube, we could check the numbers $1, 2, 3, \ldots$ to see whether they were squbes, and keep track of the number of squbes found. Once we found our K^{th} sqube, we could return that value.

   One way to implement such a function is to check whether the input number breaks down into a square and a cube. We can take the number x, subtract the first cube (1) from it, and then determine whether the leftover $x - 1$ is a square. Then we check whether $x - 2^3$ is a square, and $x - 3^3$, and so on. If we ever find a square, we can return "true". On the other hand, if we try subtracting off all cubes that are smaller than x without ever leaving a square behind, we can return "false."

   This still leaves us with one issue, though: how can we check whether a number is a square? One way is to take the square root of it directly, and see whether that is an integer, e.g., by seeing whether it equals the number it rounds to. That works fine for this problem, but there is a subtle danger lurking under the hood: some non-integer numbers can't be stored exactly in memory, and this can cause errors! We'll get more into this in our course.

   The approach that we detailed above works for this problem, but there is a way to avoid taking square (or cube) roots at all. We know how big K can possibly get in our test cases, so we can take advantage of that! Instead of going through the positive integers and seeing which ones are squbes, we can just compute enough squbes that we know the first K will be included.

   Since 1 is a cube, we know that $2^2 + 1^3, 3^2 + 1^3, \ldots$ are all squbes. So the 999^{th} sqube, for instance, can't be larger than $1000^2 + 1^3$. Therefore, any sqube $a^2 + b^3$ that is within the first 999 squbes must have $a \leq 1000$.

   Can we place a similar bound on b? Since cubes grow faster than squares, we can already be sure that $b \leq 1000$ as well. (We *could* come up with an even tighter bound, but 1000 will be fine for our purposes.)

   Therefore, if we know that $a \leq 1000$ and $b \leq 1000$, we can just compute all values $a^2 + b^3$ with $1 \leq a \leq 1000, 1 \leq b \leq 1000$, and add them to a big list of squbes. Then we can sort the list and take the K^{th} value.

   However, we do have to be somewhat careful as we do this:

   - We have to remember not to use pairs with $a = b$.
   - There might be multiple (a, b) pairs that produce the same sqube: we saw the example of $4^2 + 1^1 = 3^2 + 2^3 = 17$. So if we just put every sqube into a big list, we might include duplicates, which will throw off our answer!

   To get around that second difficulty, we could make a big list and then prune duplicates, but it is easier to put all our values into a `set` data structure. A set exists specifically to hold a collection of unique items, which is exactly what we want here! We can put all our values in a set, let the set do the work of handling duplicates, and then convert it back to a list and sort it.

   An implementation of this idea appears on the next page.


```
def find_kth_sqube(k):
  squares = []
  cubes = []
  for x in range(1, k + 1):
    squares.append(x**2)
    cubes.append(x**3)
  # We would normally use list comprehensions for the above:
  # squares = [x**2 for x in range(1, k + 1)]
  # cubes = [x**3 for x in range(1, k + 1)]
  # But we've tried to keep it more readable here in case you haven't seen
  # list comprehensions.
  squbes_set = set()
  for a in range(len(squares)):
    for b in range(len(cubes)):
      if a == b:
        continue
      squbes_set.add(squares[a] + cubes[b])
  squbes_list = sorted(list(squbes_set))
  # We take the (k-1)th index here because lists are 0-indexed.
  return squbes_list[k-1]
```