Code Jam 2018 - World Finals

Go, Gophers!

Problem

Earlier this year, the Code Jam team planted an orchard with the help of an industrious gopher. It must have told other gophers, because we now have somewhere between 2 and 25 gophers living in the orchard. But it is hard to be sure exactly how many there are, because these gophers only emerge from their underground tunnels to eat at night, and we are too tired after a hard day of tree-pruning to stay up and watch for them. However, we do know how to make one "gopher snack" per day, which we can leave out each night to see whether it gets eaten. We think we can use this information to determine the number of gophers.

Here is what we know about the way that gophers eat. The N gophers meet during one day in a council to determine an order in which they will emerge over the following N nights, one at a time. Then, during each of the i-th of the next N nights, the i-th gopher in the order emerges and looks for a gopher snack. Each gopher has its own particular taste level (which never changes), and it will eat a snack if and only if the snack's quality level is at least as high as that gopher's taste level. During the day after the N-th gopher in the order has emerged, the gophers choose a new order and the process continues. Notice that even if a gopher chooses not to eat the snack that it finds, it still does not emerge again until it comes up in the next order chosen by the council.

We must make exactly one new gopher snack each day; even if a snack is not eaten, it spoils and cannot be reused the next night. Each morning, we learn whether or not the previous night's snack was taken.

Today, we know that the gophers are meeting in their council to determine their next order, so tonight will mark the start of that order. We are willing to devote some serious time to this investigation — as many as 10^5 nights. Using **S** or fewer snacks, can you help us figure out how many gophers there are?

Input and output

This problem is interactive, which means that the concepts of input and output are different than in standard Code Jam problems. You will interact with a separate process that both provides you with information and evaluates your responses. All information comes into your program via standard input; anything that you need to communicate should be sent via standard output. Remember that many programming languages buffer the output by default, so make sure your output actually goes out (for instance, by flushing the buffer) before blocking to wait for a response. See the FAQ for an explanation of what it means to flush the buffer. Anything your program sends through standard error is ignored, but it might consume some memory and be counted against your memory limit, so do not overflow it. To help you debug, a local testing tool script (in Python) is provided at the very end of the problem statement. In addition, sample solutions to a previous Code Jam interactive problem (in all of our supported languages) are provided in the analysis for Number Guessing.

Initially, your program should read a single line containing a single integer **T** indicating the number of test cases. Then, you need to process **T** test cases. For each test case, your program will first read one line containing one integer **S**: the maximum number of snacks you can use. Then, your program will process up to **S** + 1 exchanges with our judge, in which the last exchange must be a guess at the answer.

For the i-th exchange, your program needs to use standard output to send a single line containing an integer Q_i.

- If Q_i is in the inclusive range [1, 10^6], it represents that you will leave out a gopher snack with quality level Q_i. In response, the judge will print a single line with a single integer: 1 if the gopher ate the snack, or 0 if it did not. This line will be printed to your input stream, as described above, and your program must read it through standard input. Then, you can start another exchange.
- If Q_i is in the inclusive range [-25, -2], it represents that your answer to the test case is that there are $-Q_i$ gophers. If your answer is correct, the judge will proceed to the next test case, if there is one.

The judge will print a single line with the integer -1, and then stop sending output to your input stream, if any of the following happen:

1. Your program sends a malformed or out-of-bounds value (e.g., 1000001, -1, or GO_IS_THE_BEST_LANGUAGE), or too many values (e.g., 1 2).
2. Your program sends a value not in the inclusive range [-25, -2] after having already sent **S** values for the current test case.
3. Your program sends a value in the inclusive range [-25, -2] that is not a correct answer. Note that this means that you only get one chance to answer a test case correctly.

If your program continues to wait for the judge after receiving -1, your program will time out, resulting in a Time Limit Exceeded error. Notice that it is your responsibility to have your program exit in time to receive the appropriate verdict (Wrong Answer, Runtime Error, etc.) instead of a Time Limit Exceeded error. As usual, if the total time or memory is exceeded, or your program gets a runtime error, you will receive the appropriate verdict.

You should not send additional information to the judge after solving all test cases. In other words, if your program keeps printing to standard output after sending the answer to the last test case, you will get a Wrong Answer judgment.

Important warning

Context switching between your program and the judge is expensive, and more so in our judging system. As opposed to other interactive problems, we found it necessary in all our reference solutions for this problem to bundle the exchanges to the server. That is, instead of "print taste level, read response, print taste level, read response" we can do "print taste level, print taste level, read response, read response" which requires less context switching.

Benchmarks: To give you some idea of how a given bundling of queries will perform in our system, we are providing some benchmarks. We wrote a program that performs **S** = 10^5 exchanges bundled into groups of specific sizes B — that is, it prints B taste levels, then reads B responses, then prints B more, then reads B more, and so on, **S** / B times. We implemented this in both Python and C++, always printing the B taste levels to a string variable and printing that string later, ensuring the buffer is not flushed within a bundle. Here are the results for each bundle size B, in seconds (rounded up to the next half-second, and taking the worst case over multiple runs):

B	1	10	50	100	200	500	10^5
Python	167	21	6.5	5.5	5	5	>250
C++	130	18	5.5	5.5	4.5	2.5	>250

Notice that with somewhat small bundle sizes, the context switching time gets below 5s per test, which is under a minute per test set.

Limits

$1 \le \mathbf{T} \le 10$.
Time limit: 90 seconds per test set. **(See important warning in the input/output section).**
Memory limit: 1GB.
The number of gophers is between 2 and 25, inclusive. The taste level of each gopher is between 1 and 10^6, inclusive. **S** = 10^5.

Test set 1 (Visible)

No two gophers have the same taste level.
The order in which the gophers emerge each night is chosen uniformly at random from all possible orders, and independently of all other orders.

Test set 2 (Hidden)

The GCD of the set {x : there exist exactly $x \ge 1$ gophers in the input that share a taste level} = 1.
The order in which the gophers emerge is chosen independently of the provided snacks.

For each test case, the multiset of taste levels and the seed for the random number generation are generated by the problem setters in advance of the contest, and will be the same for any contestant, for any submission. That means two submissions that offer the same number s_i of snacks for test case i will see the gophers emerge in the same order.

For example, the following scenario would be possible in either of the test sets:

- two gophers, one with taste level 1, and one with taste level 2

The following scenario would be possible in test set 2, but not in test set 1:

- three gophers, two with taste level 1, and one with taste level 2

The following scenarios would not be possible in either of the test sets:

- six gophers, four with taste level 1, and two with taste level 2
- two gophers, both with taste level 7

Sample Interactions

The following interaction is for Test set 1.

```
// In this example, the problem setters have already determined that the first
// test case has two gophers with taste levels 1 and 2 (we will call them A
// and B, respectively), and that the second test case has four gophers with
// taste levels 1, 999, 123, and 4567 (we will call them C, D, E, and F,
// respectively).
// The judge randomly generates the first order: A, B.
t = readline_int()           // Code reads 2 into t.
s = readline_int()           // Code reads 100000 into s.
printline 1 to stdout        // Code sends a snack with quality level 1.
flush stdout
resp = readline_str()        // Code reads 1 into resp (gopher A ate the
                             //   snack).
printline 1 to stdout
flush stdout
resp = readline_srt()        // Code reads 0 into resp (gopher B did not eat
                             //   the snack).
                             // Judge randomly generates B, A as the next
                             //   order.
printline 2 to stdout
flush stdout
resp = readline_str()        // Code reads 1 into resp (gopher B ate the
                             //   snack).
printline 1 to stdout
flush stdout
resp = readline_str()        // Code reads 1 into resp (gopher A ate the
                             //   snack).
                             // Judge randomly generates B, A as the next
                             //   order.
printline 2 to stdout
flush stdout
resp = readline_str()        // Code reads 1 into resp (gopher B ate the
                             //   snack).
printline 2 to stdout
flush stdout
resp = readline_str()        // Code reads 1 into resp (gopher A ate the
                             //   snack).
printline -2 to stdout       // Code correctly determines that the only
flush stdout                 //   scenario consistent with the information
                             //   given so far is two gophers with taste
                             //   levels 1 and 2.
                             // Judge rules that the answer is correct, and
```

```
                                //   prepares the next test case...
                                // Judge randomly generates C, E, F, D as the
                                //   first order.
s = readline_int()              // Code reads 100000 into s. (This also shows
                                //   that the answer to the first test case was
                                //   correct.)
printline 0 to stdout           // Code sends an invalid value.
flush stdout
resp = readline_str()           // Code reads -1 into resp.
exit                            // Code exits to avoid an ambiguous TLE error.
```

The following interaction is for Test set 2. Notice that the interactions in the first test case are the same as in the previous example, but the outcome is different.

```
// In this example, the problem setters have already determined that the first
// test case has three gophers with taste levels 1, 2, and 1; we will call
// them A, B, and C, respectively, and they will be ordered ABCCBAABCCBA...
t = readline_int()              // Code reads 1 into t.
s = readline_int()              // Code reads 100000 into s.
printline 1 to stdout
flush stdout
resp = readline_str()           // Code reads 1 into resp (gopher A ate
                                //   the snack).
printline 1 to stdout
flush stdout
resp = readline_srt()           // Code reads 0 into resp (gopher B did not eat
                                //   the snack).
printline 1 to stdout
flush stdout
resp = readline_str()           // Code reads 1 into resp (gopher C ate the
                                //   snack).
printline 2 to stdout
flush stdout
resp = readline_str()           // Code reads 1 into resp (gopher C ate the
                                //   snack).
printline 2 to stdout
flush stdout
resp = readline_str()           // Code reads 1 into resp (gopher B ate the
                                //   snack).
printline -2 to stdout          // Code erroneously decides that there
                                //   are two gophers A and B with taste levels
                                //   1 and 2; this is consistent with the
                                //   information given so far for the order
                                //   A,B,A,B,A, but the true number of gophers
flush stdout                    //   is different, so judge rules it is wrong.
s = readline_str()              // Code tries to read s but gets -1, meaning
                                //   that the answer to the last test case was
                                //   wrong.
exit                            // Code exits to avoid an ambiguous TLE error.
```

Testing Tool

You can use this testing tool to test locally or on our platform. To test locally, you will need to run the tool in parallel with your code; you can use our interactive runner for that. For more information, read the instructions in comments in that file, and also check out the Interactive Problems section of the FAQ.

Instructions for the testing tool are included in comments within the tool. We encourage you to add your own test cases. Please be advised that although the testing tool is intended to simulate the judging system, it is **NOT** the real judging system and might behave differently. If your code passes the testing tool but fails the real judge, please check the Coding section of the FAQ to make sure that you are using the same compiler as us.

Download testing tool