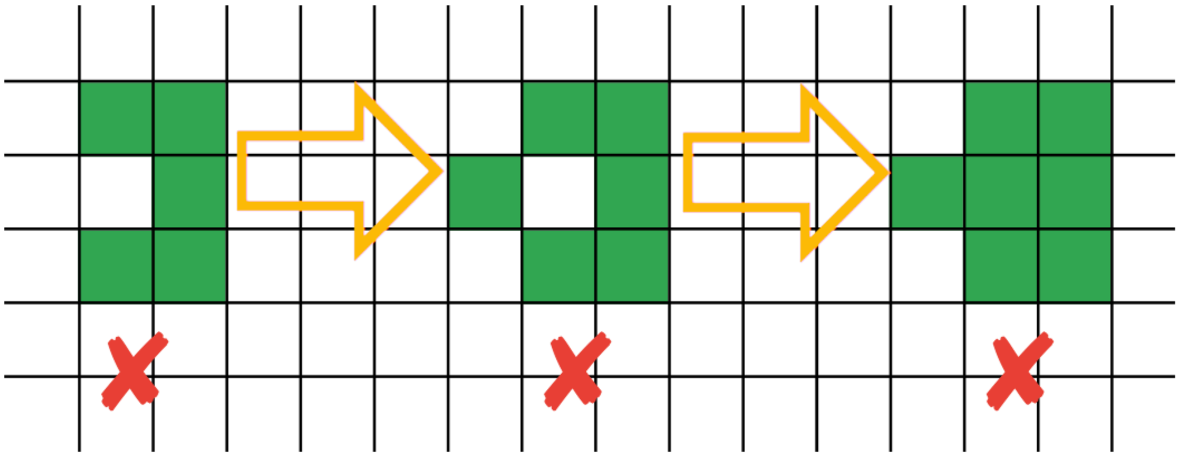
Go, Gopher!

Problem

Code Jam Team은 2차원배열의 모양인 1000x1000 과수원을 방금 구매하였다. 이제 그 과수원을 가꾸는 작업을 진행해야 되는데 - AVL, binary, red-black, splay 등등을 심을 예정이다. - 그래서 우리는 땅에 구멍을 내는 작업을 진행할 것이다.

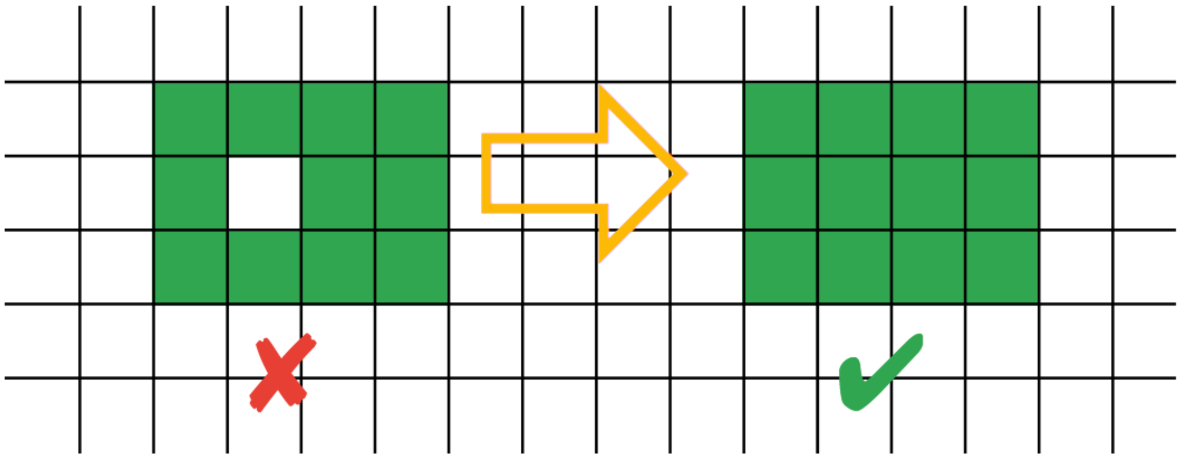
- 매번 일정량의 나무를 활용하기 위해서는 최소한 A개의 준비된 칸이 필요하다.
- 제대로 나무를 가꾸기 위해서는, 준비된 칸들은 연속된 하나의 직사각형 모양을 이루어야 하는데, 그 직사각형을 이루는 칸들은 모두 준비가 완료된 상태여야 한다.

참고로, 직사각형 밖에 위치한 다른 칸들은 준비된 상태이면 안된다. 그리고 우리는 이 직사각형들이 정돈된 것을 원한다.



예를 들어, A=11일때, 11개의 칸들이 준비가 완료된 상태임에도 불구하고 직사각형은 3x4의 사이즈를 가지고 있고, 그 안의 한 셀이 준비가 되어 있지 않은 상태이다. 결과적으로, 3x4 칸이 모두 준비상태가 아니기 때문에 준비가 되지 않은 영역이라고 판단한다. 하지만 만약에 마지막 남은 그 칸을 준비상태로 채우게 된다면, 그 영역들은 준비상태가 된다.

아래의 다른 예제로, A=6일 경우를 보자. 3x2 공간이 채워져 있지만, 그 외의 공간에 준비된 칸들이 존재한다. 이 경우에는 준비가 되지 않은 상태라고 가정한다. -> 아마 직사각형 모양에만 정보들이 채워져 있어야 한다는 뜻인듯...



사람이 직접 홈을 파는것은 어려운 작업이니, 우리는 Google Go 팀으로부터 Go gopher라는 것을 사용할 것인데, 이 놈은 칸들을 준비상태로 만들어주는데에 사용될 것이다. 우리는 gopher를 통해 활용되지 않은 특정 영역에 대한 작업을 수행할 수 있다. 하지만, gopher에 대한 알고리즘을 완성한 것은 아니므로, 이 놈은 pseudo-random 방식을 통해 똑같은 방식으로 3x3만큼의 공간을 선택할 것이다.

우리는 gopher를 1000번 이하로 사용하여 테크니컬하게 특정 영역들을 준비된 상태로 만들어야 한다. gopher를 사용할때, 당신은 gopher가 어떤 칸들을 준비상태로 만들었는지에 대한 정보를 계정에 등록을 하는 방식을 사용해도 된다. 다만 직사각형의 모양 또는 영역이 어느 위치에 해당하는지에 대한 정보에 대한 것은 언급할 필요가 없다.

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 here.

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 read a single line containing a single integer **A** indicating the minimum required prepared rectangular area. Then, your program will process up to 1000 exchanges with our judge.

For each exchange, your program needs to use standard output to send a single line containing two integers I and J: the row and column number you want to deploy the gopher to. The two integers must be between 2 and 999, and written in base-10 without leading zeroes. If your output format is wrong (e.g., out of bounds values), your program will fail, and the judge will send you a single line with `-1 -1` which signals that your test has failed, and it will not send anything to your input stream after that. Otherwise, in response to your deployment, the judge will print a single line containing two integers I' and J' to your input stream, which your program must read through standard input.

If the last deployment caused the set of prepared cells to be a rectangle of area at least **A**, you will get I' = J' = 0, signaling the end of the test case. Otherwise, I' and J' are the row and column numbers of the cell that was actually prepared by the gopher, with abs(I'-I) ≤ 1 and abs(J'-J) ≤ 1. Then, you can start another exchange.

If your program gets something wrong (e.g. wrong output format, or out-of-bounds values), as mentioned above, the judge will send I' = J' = -1, and stop sending output to your input stream afterwards. If your program continues to wait for the judge after reading in I' = J' = -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.

If the test case is solved within 1000 deployments, you will receive the I' = J' = 0 message from the judge, as mentioned above, and then continue to solve the next test case. After 1000 exchanges, if the test case is not solved, the judge will send the I' = J' = -1 message, and stop sending output to your input stream after.

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 receiving I' = J' = 0 message from the judge for the last test case, you will receive a Wrong Answer judgment.

Please be advised that for a given test case, the cells that the gopher will pick from each 3x3 block are (pseudo-)random and independent of each other, but they are determined using the same seed each time for the same test case, so a solution that gives an incorrect result for a test case will do so consistently across all attempts for the same test case. We have also set different seeds for different test cases.

Limits

1 ≤ **T** ≤ 20. Memory limit: 1 GB.

Test set 1 (Visible)

A = 20. Time limit (for the entire test set): 20 seconds.

Test set 2 (Hidden)

A = 200. Time limit (for the entire test set): 60 seconds.

Sample interaction

```
  t = readline_int()         // reads 2 into t
  a = readline_int()         // reads 3 into a
  printline 10 10 to stdout  // sends out cell 10 10 to prepare
  flush stdout
  x, y = readline_two_int()  // reads 10 11, since cell 10 11 is prepared
  println 10 10 to stdout  // sends out cell 10 10 again to prepare
  flush stdout
  x, y = readline_two_int()  // reads 10 10, since cell 10 10 is prepared
  println 10 12 to stdout  // sends out cell 10 12 to prepare
  flush stdout
  x, y = readline_two_int()  // reads 10 11, since cell 10 11 is prepared again
  println 10 10 to stdout  // sends out cell 10 10 to prepare
  flush stdout
  x, y = readline_two_int()  // reads 11 10, since cell 11 10 is prepared
  println 11 10 to stdout  // sends out cell 11 10 to prepare
  flush stdout
  x, y = readline_two_int()  // reads 0 0; since cell 11 11 is prepared, a
rectangle of size 4
```

The pseudocode above is the first half of a sample interaction for one test set. Suppose there are only two test cases in this test set. The pseudocode first reads the number of test cases into an integer `t`. Then the first test case begins. For the first test case, suppose **A** is 3 (although, in the real test sets, **A** is always either 20 or 200). The pseudocode first reads the value of **A** into an integer `a`, and outputs `10 10` the location of the cell to prepare. By (pseudo-)random choice, the cell at location 10 11 is prepared, so the code reads `10 11` in response. Next, the code outputs cell `10 10` again for preparation, and the gopher prepares `10 10`this time. The code subsequently sends `10 12` with the goal of finishing preparing a rectangle of size 3, but only gets cell `10 11`prepared again. `10 10` is then sent out, and this time `11 10` is prepared. Notice that although the prepared area is of size 3, a rectangle has not been formed, so the preparation goes on. In the end, the pseudocode decides to try out cell `11 10`, and `0 0` is sent back, which implies that cell 11 11 has been prepared, completing a rectangle (or square, rather) or size 4. As a result, the first test case is successfully solved.

```
  a = readline_int()         // reads 3 into a
  println 10 10 to stdout  // sends out cell 10 10 to prepare
  x, y = readline_two_int()  // does not flush stdout; hangs on the judge
```

Now the pseudocode is ready for the second test case. It again first reads an integer `a = 3` and decides to send cell `10 10` to prepare. However, this time, the code forgets to flush the stdout buffer! As a result, 10 10 is buffered and not sent to the judge. Both the judge and the code wait on each other, leading to a deadlock and eventually a Time Limit Exceeded error.

```
a = readline_int()          // reads 3 into a
println 1 1 to stdout       // sends out cell 1 1 to prepare
x, y = readline_two_int()   // reads -1 -1, since 1 is outside the range [2, 999]
println 10 10 to stdout     // sends a cell location anyway
x, y = readline_two_int()   // hangs since the judge stops sending info to stdin
```

The code above is another example. Suppose for the second test case, the code remembers to flush the output buffer, but sends out cell `1 1` to prepare. Remember that the row and column of the chosen cell must both be in the range [2, 999], so 1 1 is illegal! As a result, the judge sends back `-1 -1`. However, after reading `-1 -1` into x and y, the code proceeds to send another cell location to the judge, and wait. Since there is nothing in the input stream (the judge has stopped sending info), the code hangs and will eventually receive a Time Limit Exceeded error.

Note that if the code in the example above exits immediately after reading `-1 -1`, it will receive a Wrong Answer instead:

```
a = readline_int()          // reads 3 into a
println 1 1 to stdout       // sends out cell 1 1 to prepare
x, y = readline_two_int()   // reads -1 -1, since 1 is outside the range [2, 999]
exit                        // receives a Wrong Answer judgment
```

Local Testing Tool

To better facilitate local testing, we provide you the following script. Instructions are included inside. You are encouraged to add more test cases for better testing. 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 here to make sure that you are using the same compiler as us.