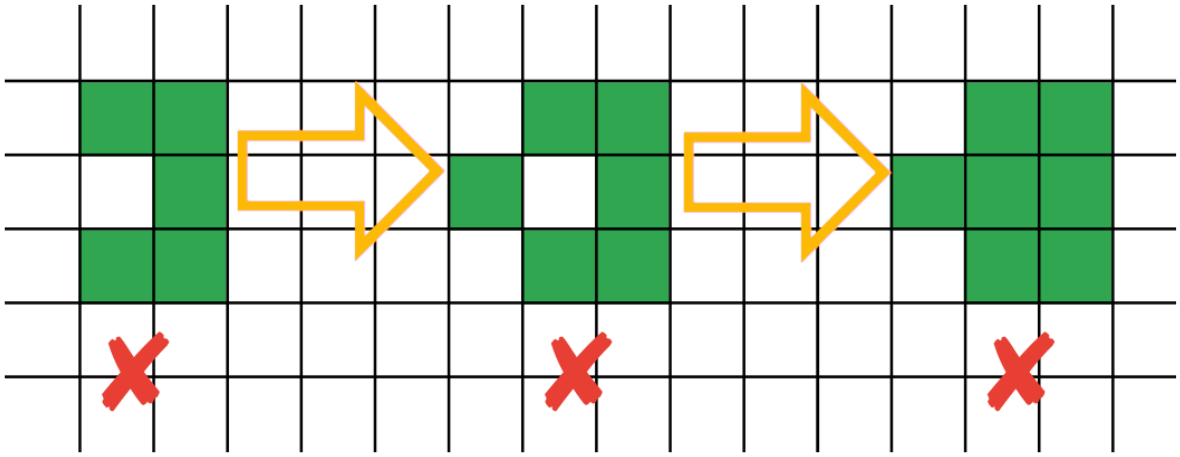
Go, Gopher!

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.

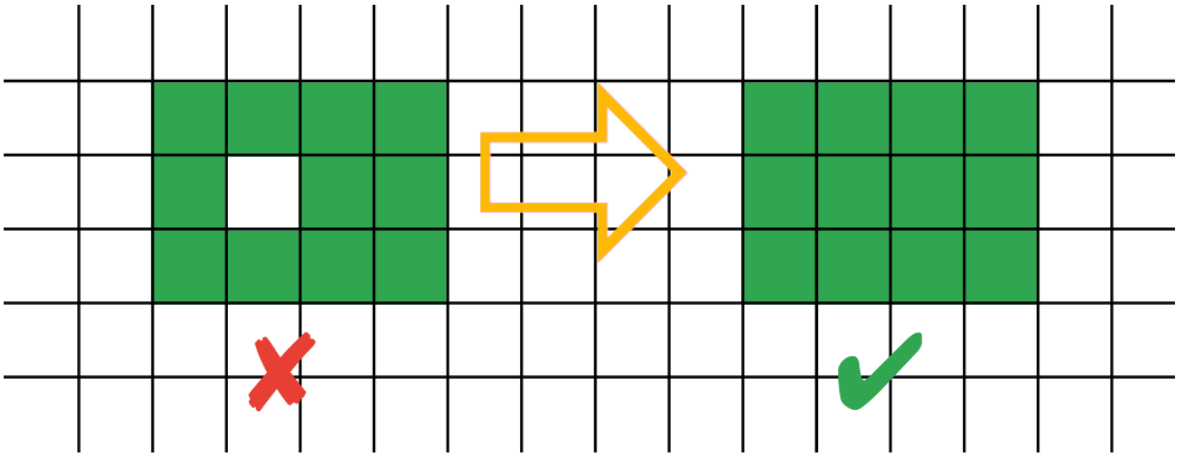
- 매번 일정량의 나무를 활용하기 위해서는 최소한 A개의 준비된 칸이 필요하다.
- 제대로 나무를 가꾸기 위해서는, 준비된 칸들은 연속된 하나의 직사각형 모양을 이루어야 하는데, 그 직사각형을 이루는 칸들은 모두 준비가 완료된 상태여야 한다.

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

언제나 문제는 주고받는 과정에서 발생하는데, input과 output에 대한 정보가 standard Code jam problems와는 다르다는 것이다(정확히 무엇을 의미하는지 모르겠음). 당신은 여러개의 프로세스와 상호작용을 할 것인데, 정보와 당신의 응답에 대한 검증을 제공할 것이다. 모든 정보들은 standard input을 거쳐 프로그램으로 전달될 것이다; 즉, 어떤 통신을 원하든 그 정보는 standard output을 통해 전달되어야 한다. 많은 프로그래밍 언어들이 output을 buffer에 할당함으로써 응답에 대한 wait 전환을 방지한다는 사실을 기억하자. (flushing 해라는 의미인듯). 어떤 에러가 전송될 경우에는 Ignored 될 것인데, 그 과정에서 메모리가 사용되어 메모리 할당제한에 걸릴수가 있으니 오버플로우를 주위해야 한다. 디버깅을 위해서는, local testing tool script(python)이 제공된다. 추가적으로, 예전 Code Jam Interactive problem에 대한 해결법들은 여기(링크: here)에 제시되어 있다.

첫번째로, 당신의 프로그램은 single Integer T를 포함한 1줄을 읽어오는데 이 T는 test cases의 개수를 의미한다. 즉, 당신은 T의 테스트 케이스들을 process 해야 한다.

각각의 테스트 케이스들에 대하여 당신의 프로그램은 single Integer A를 포함한 1줄을 읽어올 것인데 이 A는 직사각형 영역의 최소 크기를 의미한다. 그다음으로, 당신의 프로그램은 1000가지의 상호작용을 진행할 것이다.

각각의 상호작용들에 대하여 당신의 프로그램은 2개의 integer I 과 j를 포함한 standard output을 사용하여야 하는데: gopher에게 줄 row와 column정보를 의미한다. 2개의 integersms 2~999사이의 값을 가져야 하며, base-10의 형태로 사용되어야 한다. 만약 당신의 output format이 잘못됬다면, 프로그램이 실패하고, judge가 -1 -1을 당신에게 보내줄 것인데, 이는 당신의 test가 실패했다는 것을 의미한다. 성공을 하였다면, judge가 I과 j를

포함한 한 줄을 당신의 프로그램에게 전달해 줄 것이다.

만약 당신의 마지막 결과가 최소한의 A의 결과와 일치하면, 당신은 I=j=0을 받을 것이다. 만약 다른 경우에는, I과 j는 gopher에 의해 할당되는 공간에 대한 정보를 담고 있을 것이다.

만약 당신의 프로그램이 먼가가 잘못되고 있다면, I=j=-1이 전송될 것이고, 통신이 중단될 것이다. 만약 당신의 프로그램이 I=j=-1후에도 통신을 기다린다면, time-out으로 인해 종료될 것이고 error가 발생할 것이다. 중요한건 제한된 시간안에 결과를 얻어야 한다는 것이다.

만약 test 케이스가 1000번안에 해결된다면, I=j=0를 받을것인데, 그 다음 test case를 수행하면 된다. 1000번의 상호작용 후에, test case가 해결되지 않았을 경우에는 I=j=-1메시지를 받을 것이다.

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.

당신은 모든 test case가 해결된 이후에 그 어떤 추가적인 내용도 전송하면 안된다. 또는 I=j=0을 받은 후에도 계속적으로 어떤 작업을 수행하면 Wrong Answer로 판단될 것이다.

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.

밑에있는 given case들을 참ㄱ소하여라. gopher에 의해 선택되는 3x3 블럭들은 (pseudo-)random에 의한 것이고 서로 독립적이다. 하지만 그것들은 same test case에 대하여 똑같은 seed를 가지고 있기 때문에, incorrect result가 발생하면 매번 똑같은 결과를 가질 것이다. 참고로 test case마다 다른 seed를 준비했다.

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
  printline 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
  printline 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
  printline 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
  printline 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
  printline 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
printline 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]
printline 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
printline 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.