Submissions

Testrun

0pt	Not attempted **0/64 users** correct (0%)

almost_sorted

1pt	Not attempted **194/203 users** correct (96%)
7pt	Not attempted **104/187 users** correct (56%)

mutexes

2pt	Not attempted **84/147 users** correct (57%)
20pt	Not attempted **48/69 users** correct (70%)

johnny

2pt	Not attempted **91/105 users** correct (87%)
30pt	Not attempted **17/70 users** correct (24%)

highest_mountain

1pt	Not attempted **43/61 users** correct (70%)
37pt	Not attempted **0/9 users** correct (0%)

Top Scores

mk.al13n	63
ecnerwala	63
shik	63
Marcin.Smulewicz	56
WJMZBMR	56
bmerry	43
ZbanIlya	43
wan92hy	42
simonlindholm	42
dreamoon	42

Problem C. mutexes

This contest is open for practice. You can try every problem as many times as you like, though we won't keep track of which problems you solve. Read the Quick-Start Guide to get started.

small 2 points *2 minute timeout*	The contest is finished.
large 20 points *10 minute timeout*	The contest is finished.

Don't know what distributed problems are about? See our guide.

Problem

In writing multi-threaded programs one of the big problems is to prevent concurrent access to data. One of the more common mechanisms for doing this is using *mutual exclusion locks* (also called *mutexes*). A *mutex* is something that can be acquired by only one thread at a time. If one thread has already acquired the mutex, and a second thread tries to acquire it, the second thread will wait until the first thread releases the mutex, and only then will it proceed (with acquiring the mutex and doing whatever it planned on doing next).

A danger when using mutexes is *deadlock* — a situation where some threads block each other and will never proceed. A deadlock occurs when each one thread has already acquired mutex **A**, and now tries to acquire mutex **B**, while another thread has already acquired mutex **B** and tries to acquire mutex **A** (more complex scenarios with more threads are also possible, but we will only be concerned with the two-thread situation).

You are now analyzing a two-threaded program, and trying to determine whether it will deadlock. You know the exact set of mutex operations (acquire and release) each of the two threads will perform, in order. However, you do not know how quickly each thread will perform each of its operations — it is possible, for instance, for one thread to perform almost all of its operations, then for the other thread to catch up, and then for the first thread to proceed.

You are interested in determining whether it is possible that the two threads will deadlock. Initially all the mutexes are released. We assume that when a thread has finished all of its operations, it releases all the mutexes it still has acquired.

Input

The input library will be called "mutexes"; see the sample inputs below for examples in your language. It will define two methods: NumberOfOperations(i), which will return the number of operations thread *i* performs (*i* has to be 0 or 1), and GetOperation(i, index), which will report what the *index*th operation performed by thread *i* is (where *i* is 0 or 1, and $0 \le index <$ NumberOfOperations(i)). This will be a positive number *X* if the *index*th operation is to acquire mutex *X*, and a negative number *-X* if the *index*th operation is to release mutex *X*.
The sequence of operations for a single thread will always be valid, that is, a given thread will never try to acquire a lock it has already acquired (and not yet released), or release a lock it has already released (and not yet acquired) or has never acquired in the first place. A thread's first operation on a lock (if any) will always be an acquire operation.
One call to GetOperation will take approximately 0.005 microseconds, with the exception of the first call, which will cache the input values and might take up to 100 milliseconds.

Output

Output the smallest total number of operations the two threads can perform before deadlocking (including the last two acquire operations), if a deadlock is possible, or the word OK if a deadlock can't happen.

Limits

Each node will have access to 256MB of RAM, and a time limit of 4 seconds.
$-10^5 \le$ GetOperation(i, index) $\le 10^5$ for all valid *i* and *index*. GetOperation will never return 0.

Small input

Your solution will run on 10 nodes.
$1 \le$ NumberOfOperations(i) ≤ 1000 for both possible values of *i*.

Large input

Your solution will run on 100 nodes.
$1 \leq \text{NumberOfOperations}(i) \leq 4 \times 10^4$ for both possible values of *i*.

Sample

```
Input                          Output

See sample input files below.  For sample input 1:
                               OK
                               For sample input 2:
                               7
                               For sample input 3:
                               6
```

The fastest way to deadlock in the third example is for the first thread to perform the first three operations (ending up with mutexes 1, 2 and 3), then for the second thread to perform the first operation (acquiring mutex 4). At this point both threads try to perform one operation more (the first thread trying to acquire mutex 4, the second thread trying to acquire mutex 3) and deadlock.

Sample input libraries:
Sample input for test 1: mutexes.h [CPP] mutexes.java [Java]
Sample input for test 2: mutexes.h [CPP] mutexes.java [Java]
Sample input for test 3: mutexes.h [CPP] mutexes.java [Java]