Distributed Online Round

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 A. Testrun

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
0 points
2 minute timeout

The contest is finished.

Problem

This is a way to test your solutions, not a real problem!

When you submit a solution to this problem, it will run one testcase on a 100 nodes. This will allow you to estimate how fast your solution will run on our system.

Remember to change your solution appropriately before submitting it for real, so you don't fail because of a compilation error! The best way to check is to run your solution on the small input before submitting to the large input.

Input

There is no input for this problem. This means you should not include / import an input library.

Output

Doesn't really matter what you output. If your solution runs successfully to completion, it will be judged as "Wrong Answer".

Limits

Each node will have access to 1 GB of RAM, and a time limit of 26 seconds. The maximum number of messages a single node can send is 5000, and the maximum sum of the sizes of those messages is 8MB.
This problem only has one small test case. It will run on 100 nodes.

Problem B. almost_sorted

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 1 points *2 minute timeout*	The contest is finished.
large 7 points *10 minute timeout*	The contest is finished.

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

Problem

As a *very important director* of a *very important company*, you have a lot of files to keep track of. You do this by carefully keeping them sorted. However, you took a well-deserved vacation last week, and when you came back, you discovered to your horror that someone has put the files out of place!

They are not very much out of place, in fact they're still *almost* sorted. More precisely, the file that should be in position i if the files were sorted is now at most K positions away — that is, somewhere between position $i - K$ and $i + K$, inclusive.
However, you can't work like this. So, you ask your assistants to put the files into their correct places. Each file has an identifier, and files with a larger identifier should be placed after those with the smaller identifier. They will not change the relative order of files with the same identifier.
To verify the files are sorted correctly, you will ask your assistants to calculate a simple checksum — for each file multiply that file's identifier by its position (beginning from 0), and sum this for all files, modulo 2^{20}.

Unfortunately, to make use of the checksum, you have to know what it's value should be. So, write a program that will output the expected checksum after sorting the files.

Input

The input library will be called "almost_sorted"; see the sample inputs below for examples in your language. It will define three methods: NumberOfFiles(), which will return the number of files, MaxDistance() — the maximum difference between the current and desired position of any file, and Identifier(i), which will return the value of the identifier of the file that's currently standing on position i, for $0 \le i <$ NumberOfFiles().
A single call to Identifier() will take approximately 0.04 microseconds.

Output

Output one number — the value of the checksum for the sorted sequence of files, modulo 2^{20}.

Limits

Each node will have access to 128MB of RAM, and a time limit of 3 seconds.
$0 \le$ Identifier(i) $\le 10^{18}$, for $0 \le i <$ NumberOfFiles().

Small input

Your solution will run on 10 nodes.
$0 \le$ MaxDistance() $<$ NumberOfFiles() ≤ 1000.

Large input

Your solution will run on 100 nodes.
$1 \le$ NumberOfFiles() $\le 10^8$.
$0 \le$ MaxDistance() $<$ NumberOfFiles().
$0 \le$ MaxDistance() $\le 10^6$.

Sample

Input	Output
See sample input files below.	For sample input 1: 0 For sample input 2: 7000 For sample input 3: 154112

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

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 0K 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.
$1 \le \text{NumberOfOperations}(i) \le 4 \times 10^4$ 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]

Problem D. johnny

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 30 points *10 minute timeout*	The contest is finished.

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

Problem

You and Johnny play a very simple card game. Both players have a deck of cards. Both draw a card at random from their deck, and the player with the better card wins. "Better" is a complex concept, but for any two cards, you (and Johnny) know which one is better, with no ties allowed. Note that this is not necessarily transitive: if card A is better than B, and B is better than C, it is possible that C is better than A.

Johnny is very, very unhappy when he loses. So, you would like to make sure that he will win. You have N cards, and you want to split them into two non-empty decks (with no cards left over), one for you and one for Johnny, so that whatever card you draw from your deck and whatever card Johnny draws from his, Johnny will win.

If it is possible to do this in multiple ways, you want Johnny's deck to be as large as possible (as long as your deck is not empty).

Input

The input library will be called "johnny"; see the sample inputs below for examples in your language. It will define two methods: NumberOfCards(), which will return the number of cards, and IsBetter(i, j), which will return true if card i is better than card j, for $0 \le i, j <$ NumberOfCards(). If called with $i=j$, it will return false. One call to IsBetter will take approximately 0.03 microseconds.

Output

If it is possible to split the cards so that Johnny will always win, output one number: the largest possible size of Johnny's deck. If it is impossible to split the cards in such a way, output the word `IMPOSSIBLE`.

Limits

Each node will have access to 256MB of RAM, and a time limit of 3 seconds.

Small input

Your solution will run on 10 nodes.
$2 \le$ NumberOfCards() ≤ 1000.

Large input

Your solution will run on 100 nodes.
$2 \le$ NumberOfCards() $\le 2 \times 10^4$.

Sample

Input	Output
See sample input files below.	For sample input 1: 1 For sample input 2: IMPOSSIBLE For sample input 3: 4

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

Problem E. highest_mountain

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.

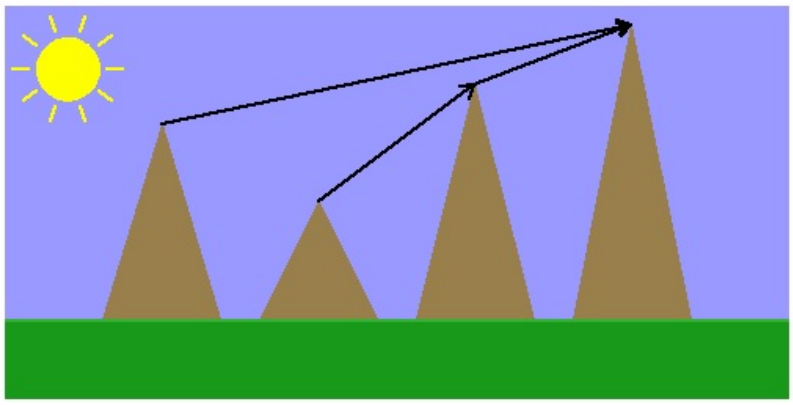
small 1 points *2 minute timeout*	The contest is finished.
large 37 points *10 minute timeout*	The contest is finished.

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

Problem

You were born and live in a small town in a remote mountain range extending east to west. In this mountain range there is a peak every kilometer, and there are no intermediate peaks. Recently, you checked on the Internet what the highest peak in the range is, and were suprised — from your town a different peak seems to be the highest. And when you went for a walk to the nearest mountain top, yet another peak appeared to be highest. You're not sure the Internet data is correct (they write all sorts of stuff on the Internet!), so you decided to compile your own list of potentially highest peaks in the range.

Your list will contain every peak **B** with the following property: if **B** is visible from some other peak **A**, no peak beyond **B** is visible from **A**. Formally, this means that if **B** lies, say, to the east of **A**, then all peaks between **A** and **B** are below the line connecting **A** and **B**, and all the peaks to the east of **B** are below or on that line.



In this example, the fourth peak is the last one you see to the east from the first and third peaks. From the second peak, to the east you only see the third peak. Your list will include only peaks 1 and 4: peak 2 is visible from peak 3, but is not the farthest (1 is), and peak 3 is visible from peak 1, but is not the farthest (4 is).

The rationale for this criterion is that you figure that if from some peak **A** you can see peak **C**, and you can also see some other peak **D** that lies in the same direction and is more distant, then **C** is'obviously not the highest peak in the range (because either **A**, or **D**, is higher). You don't trust your intuition any more, so even if the highest visible peak in any direction appears to be much lower than the one you're standing on (for instance, you are standing on a peak of height 1000, and the next and last peak to the east is of height 1), you will consider the peak of height 1 to be a candidate for your list.

Input

The input library will be called "highest_mountain"; see the sample inputs below for examples in your language. It will define two methods:
NumberOfPeaks(), which will return the number of peaks in the range, and
GetHeight(i), which will return the height of the *i*th peak from the west, for $0 \le i < \text{NumberOfPeaks}()$.
One call to GetHeight will take approximately 0.1 microseconds.

Output

Output one number: the total number of the peaks that will be included in your list.

Limits

Each node will have access to 128MB of RAM, and a time limit of 6 seconds.
$0 \le \text{GetHeight}(i) \le 10^9$ for all i with $0 \le i < \text{NumberOfPeaks}()$.

Small input

Your solution will run on 10 nodes.
1 ≤ NumberOfPeaks() ≤ 1000.

Large input

Your solution will run on 100 nodes.
1 ≤ NumberOfPeaks() ≤ 4×10^8.

Sample

Input	Output
See sample input files below.	For sample input 1: 2 For sample input 2: 3 For sample input 3: 4

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