Distributed Round 1 2016

Submissions

Testrun
0pt | Not attempted **0/422 users** correct (0%)

oops
2pt | Not attempted **893/925 users** correct (97%)
12pt | Not attempted **756/882 users** correct (86%)

rps
1pt | Not attempted **789/857 users** correct (92%)
15pt | Not attempted **585/783 users** correct (75%)

crates
8pt | Not attempted **557/673 users** correct (83%)
25pt | Not attempted **258/433 users** correct (60%)

winning_move
3pt | Not attempted **635/700 users** correct (91%)
34pt | Not attempted **49/309 users** correct (16%)

Top Scores

simonlindholm	100
tomconerly	100
eatmore	100
cgy4ever	100
bmerry	100
Simon.M	100
Klockan	100
tczajka	100
tkociumaka	100
Zlobober	100

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.

Distributed Round 1 2016

A. Testrun
B. oops
C. rps
D. crates
E. winning_move

Contest Analysis

Questions asked 8

Submissions

Testrun
0pt | Not attempted 0/422 users correct (0%)

oops
2pt | Not attempted 893/925 users correct (97%)
12pt | Not attempted 756/882 users correct (86%)

rps
1pt | Not attempted 789/857 users correct (92%)
15pt | Not attempted 585/783 users correct (75%)

crates
8pt | Not attempted 557/673 users correct (83%)
25pt | Not attempted 258/433 users correct (60%)

winning_move
3pt | Not attempted 635/700 users correct (91%)
34pt | Not attempted 49/309 users correct (16%)

Top Scores

simonlindholm	100
tomconerly	100
eatmore	100
cgy4ever	100
bmerry	100
Simon.M	100
Klockan	100
tczajka	100
tkociumaka	100
Zlobober	100

Problem B. oops

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

Problem

Oops

The team preparing the Distributed Code Jam made a mess and needs your help. The statement and solutions for this problem were lost minutes before the contest, and all that we could recover were these two correct but far too slow (and misguided) solutions, one per language. Fortunately, we still have the test data. Can you reconstruct the statement and solve the problem properly based on the recovered slow solutions?

Notice that in this problem 20 nodes are used to run both the Small and the Large datasets, which is not the usual number for Distributed Code Jam problems. 20 nodes were also used to run the solutions and produce the answers for the examples.

The C++ solution:

```
#include <message.h>
#include <stdio.h>
#include "oops.h"

#define MASTER_NODE 7
#define DONE -1

int main() {
  long long N = GetN();
  long long nodes = NumberOfNodes();
  long long my_id = MyNodeId();
  long long best_so_far = 0LL;
  for (long long i = 0; i < N; ++i) {
    for (long long j = 0; j < N; ++j) {
      if (j % nodes == my_id) {
        long long candidate = GetNumber(i) - GetNumber(j);
        if (candidate > best_so_far) {
          best_so_far = candidate;
          PutLL(MASTER_NODE, candidate);
          Send(MASTER_NODE);
        }
      }
    }
  }
  PutLL(MASTER_NODE, DONE);
  Send(MASTER_NODE);

  if (my_id == MASTER_NODE) {
    long long global_best_so_far = 0;
    for (int node = 0; node < nodes; ++node) {
      long long received_candidate = 0;
      while (true) {
        Receive(node);
        received_candidate = GetLL(node);
        if (received_candidate == DONE) {
          break;
        }
        if (received_candidate > global_best_so_far) {
          global_best_so_far = received_candidate;
        }
      }
    }
    printf("%lld\n", global_best_so_far);
  }
  return 0;
}
```

The Java solution:

```
public class Main {
  static int MASTER_NODE = 7;
  static int DONE = -1;
```

```
  public static void main(String[] args) {
    long N = oops.GetN();
    long nodes = message.NumberOfNodes();
    long my_id = message.MyNodeId();
    long best_so_far = 0L;
    for (long i = 0; i < N; ++i) {
      for (long j = 0; j < N; ++j) {
        if (j % nodes == my_id) {
          long candidate = oops.GetNumber(i) - oops.GetNumber(j
          if (candidate > best_so_far) {
            best_so_far = candidate;
            message.PutLL(MASTER_NODE, candidate);
            message.Send(MASTER_NODE);
          }
        }
      }
    }
    message.PutLL(MASTER_NODE, DONE);
    message.Send(MASTER_NODE);

    if (my_id == MASTER_NODE) {
      long global_best_so_far = 0;
      for (int node = 0; node < nodes; ++node) {
        long received_candidate = 0;
        while (true) {
          message.Receive(node);
          received_candidate = message.GetLL(node);
          if (received_candidate == DONE) {
            break;
          }
          if (received_candidate > global_best_so_far) {
            global_best_so_far = received_candidate;
          }
        }
      }
      System.out.println(global_best_so_far);
    }
  }
}
```

Input

The input library is called "oops"; see the sample inputs below for examples in your language. It defines two methods:

- **GetN()**:
  - Takes no argument.
  - Returns a 64-bit number.
  - Expect each call to take 0.05 microseconds.
- **GetNumber(i)**:
  - Takes a 64-bit number in the range 0 ≤ i < GetN().
  - Returns a 64-bit number.
  - Expect each call to take 0.05 microseconds.

Output

Output what either of the solutions above would output, if they ran on 20 nodes without any limits on memory, time, number of messages or total size of messages.

Limits

Time limit: 6 seconds.
Memory limit per node: 128 MB.
Maximum number of messages a single node can send: 1000.
Maximum total size of messages a single node can send: 8 MB.
Number of nodes: 20.
$-10^{18} \le$ GetNumber(i) $\le 10^{18}$, for all i.

Small dataset

1 ≤ GetN() ≤ 30,000.

Large dataset

1 ≤ GetN() ≤ 10^9.

Sample

Input	Output
See input files below.	For sample input 1: 93 For sample input 2: 0 For sample input 3: 14

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

Problem C. rps

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

Problem

Remarkably Parallel Scenario

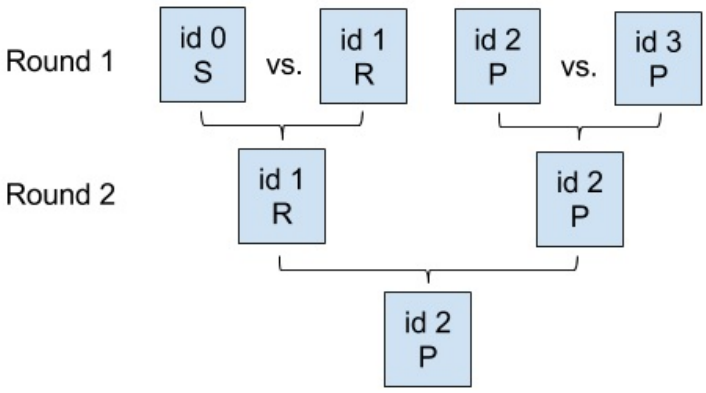
Your Rock-Paper-Scissors tournament yesterday went so well that you've been asked to organize another one. It will be a single-elimination tournament with **N** rounds. $2^{\mathbf{N}}$ players will participate, and they will have unique ID numbers in the range 0 through $2^{\mathbf{N}}-1$, inclusive.

Initially, the players will be lined up from left to right, in increasing order by ID number. In each round, the first and second players in the lineup (starting from the left) will play a match against each other, and the third and fourth players in the lineup (if they exist) will play a match against each other, and so on; all of these matches will occur simultaneously. The winners of these matches will remain in the lineup, in the same relative order, and the losers will leave the lineup and go home. Then a new round will begin. This will continue until only one player remains in the lineup; that player will be declared the winner.

In each Rock-Paper-Scissors match, each of the two players secretly chooses one of *Rock*, *Paper*, or *Scissors*, and then they compare their choices. Rock beats Scissors, Scissors beats Paper, and Paper beats Rock. If one player's choice beats the other player's choice, then that player wins and the match is over. You are tired of worrying about ties, so you have decided that if the players make the same choice, the player on the left wins the match.

You know that the players this year are not very strategic, and each one has a preferred move and will only ever play that move. Fortunately, you know every player's preferred move, so you can figure out: what is the ID number of the player who will win the tournament?

Here's an example tournament with **N** = 2:



In Round 1, player 1 beats player 0 (since Rock always beats Scissors), and player 2 beats player 3 (since the player with the lower number wins a tie). In Round 2, player 2 beats player 1 (since Paper always beats Rock). So, player 2 is the winner.

Input

The input library will be called "rps"; see the sample inputs below for examples in your language. It defines two methods: GetN(), which returns the number **N** of rounds in the tournament, and GetFavoriteMove(id), which returns the favorite move of the player with ID number id, for $0 \le id < 2^{GetN()}$. This move will always be either R, P, or S, representing Rock, Paper, or Scissors, respectively.

- **GetN()**:
  - Takes no argument.
  - Returns a 64-bit number: the **N** value from the statement.

    - Expect each call to take 0.09 microseconds.
- **GetFavoriteMove(id)**:
    - Takes a 64-bit number in the range $0 \le i < 2^{GetN()}$.
    - Returns a character.
    - Expect each call to take 0.09 microseconds.

Output

Output one value: the ID number of the winning player.

Limits

Time limit: 3 seconds.
Memory limit per node: 128 MB.
Maximum number of messages a single node can send: 1000.
Maximum total size of messages a single node can send: 8 MB.
GetFavoriteMove(id) is always one of R, P, or S for all valid values of id.

Small dataset

Number of nodes: 10.
$1 \le GetN() \le 10$.

Large dataset

Number of nodes: 100.
$1 \le GetN() \le 28$.

Sample

Input	Output
See input files below.	For sample input 1: 5 For sample input 2: 0 For sample input 3: 2

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

Problem D. crates

small 8 points *2 minute timeout*	The contest is finished.
large 25 points *10 minute timeout*	The contest is finished.

Problem

Rearranging Crates

You are the manager of the warehouse of the largest wharf in the area. The warehouse is tall and long, but narrow, so you have created a single row of stacks of crates. Unfortunately, the ship unloaders are usually sloppy and in a rush, so the number of crates can vary wildly between stacks, making them look like an uneven skyline of buildings.

The District Crate Judge (DCJ) is visiting the facilities tomorrow, and you want to get a perfect score in her assessment. You decided to use an old crane to rearrange the crates to make them into a neatly organized crate wall that will impress the DCJ, but this task may take a long time! Your crane can only handle one crate at a time. Moreover, the crane's wheels are not strong enough to move if the crane is handling a crate, so the crane can only drop off a crate on a stack adjacent to the one the crate came from.

The crane starts off at the leftmost stack, without any crates. The crane can carry out only the following sequence of actions, which we will call a *move*:

1. Position the crane at any stack with at least one crate. This may be the stack the crane is already at.
2. Pick up the top crate from the stack the crane is at. (Even if this causes the stack to have zero crates, it still counts as a stack.)
3. Put that crate on top of a stack directly adjacent to the stack the crane is at.

Notice that if you want to move a crate two stacks down to the right, for example, this will take two moves.

You must use some number of moves to transfer crates around so that all stacks are as even as possible, with all the excess distributed among the leftmost possible stacks. If there are N stacks and a total of C crates, you want the $C\%N$ leftmost stacks to have $\lceil C/N \rceil$ crates each and the rest of the stacks to have $\lfloor C/N \rfloor$ crates each. For instance, if there are 3 stacks and a total of 8 crates, you want the stack heights to be 3, 3, 2, from left to right.

What is the minimum number of moves you need to use to balance the stacks as specified above? Since the result can be really big, output it modulo 10^9+7 (1000000007).

Input

The input library is called "crates"; see the sample inputs below for examples in your language. It defines two methods:

- **GetNumStacks()**:
  - Takes no argument.
  - Returns a 64-bit number: the number of stacks.
  - Expect each call to take 0.12 microseconds.
- **GetStackHeight(i)**:
  - Takes a 64-bit number in the range $1 \le i \le$ GetNumStacks().
  - Returns a 64-bit number: the starting number of crates in stack i, counting from left to right and starting from 1.
  - Expect each call to take 0.12 microseconds.

Output

Output a single line with a single integer, the remainder of dividing the minimum number of moves needed to balance the stacks by 10^9+7 (1000000007).

Limits

Time limit: 6 seconds.
Memory limit per node: 128 MB.
Maximum number of messages a single node can send: 1000.
Maximum total size of messages a single node can send: 8 MB.

$1 \le$ GetStackHeight(i) $\le 10^9$, for all i.

Small dataset

Number of nodes: 10.
$1 \le$ GetNumStacks() $\le 10^6$.

Large dataset

Number of nodes: 100.
$1 \le$ GetNumStacks() $\le 10^9$.

Sample

Input	Output
See input files below.	For sample input 1: 3 For sample input 2: 5 For sample input 3: 0

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

Problem E. winning_move

small 3 points *2 minute timeout*	The contest is finished.
large 34 points *10 minute timeout*	The contest is finished.

Problem

The Only Winning Move

Perhaps you have played this game before: Every player submits a positive integer. The winner, if any, is the player who submitted the smallest positive integer that was submitted by no other player.

Your friends invited you to play this game, but you decided that game theory is complex and the only winning move is not to play. Instead, you volunteered to judge the game. Given the players' choices, can you determine what the winning number was?

Input

The input library will be called "winning_move"; see the sample inputs below for examples in your language. It defines two methods:

- **GetNumPlayers()**:
  - Takes no argument.
  - Returns a 64-bit number: the number of players in the game.
  - Expect each call to take 0.12 microseconds.
- **GetSubmission(playernum)**:
  - Takes a 64-bit number in the range 0 ≤ playernum < GetNumPlayers().
  - Returns a 64-bit number: the number chosen by the player with ID number playernum.
  - Expect each call to take 0.12 microseconds.

Output

Output one value: the winning number, or 0 if there was no winner.

Limits

Time limit: 4 seconds.
Memory limit per node: 128 MB.
Maximum number of messages a single node can send: 1000.
Maximum total size of messages a single node can send: 1 GB.
$1 \le \text{GetSubmission(playernum)} \le 10^{18}$, for all playernum.

Small dataset

Number of nodes: 10.
1 ≤ GetNumPlayers() ≤ 1000.

Large dataset

Number of nodes: 100.
1 ≤ GetNumPlayers() ≤ 35000000.

Sample

Input	Output
See input files below.	For sample input 1: 4 For sample input 2: 3 For sample input 3: 0

Sample input libraries:
Sample input for test 1: winning_move.h [CPP] winning_move.java [Java]

Sample input for test 2: winning_move.h [CPP] winning_move.java [Java]
Sample input for test 3: winning_move.h [CPP] winning_move.java [Java]

Distributed Round 2 2016

Submissions

Testrun
0pt | Not attempted **0/74 users** correct (0%)

again
1pt | Not attempted **401/409 users** correct (98%)
14pt | Not attempted **368/399 users** correct (92%)

lisp_plus_plus
3pt | Not attempted **390/399 users** correct (98%)
17pt | Not attempted **355/385 users** correct (92%)

asteroids
5pt | Not attempted **283/305 users** correct (93%)
25pt | Not attempted **91/170 users** correct (54%)

gas_stations
8pt | Not attempted **191/233 users** correct (82%)
27pt | Not attempted **28/95 users** correct (29%)

Top Scores

eatmore	100
Marcin.Smulewicz	100
tozangezan	100
Errichto.rekt	100
mnbvmar	100
qwerty787788	100
sevenkplus	100
tczajka	100
fhlasek	100
wata	100

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.

Submissions

Testrun
0pt | Not attempted 0/74 users correct (0%)

again
1pt | Not attempted 401/409 users correct (98%)
14pt | Not attempted 368/399 users correct (92%)

lisp_plus_plus
3pt | Not attempted 390/399 users correct (98%)
17pt | Not attempted 355/385 users correct (92%)

asteroids
5pt | Not attempted 283/305 users correct (93%)
25pt | Not attempted 91/170 users correct (54%)

gas_stations
8pt | Not attempted 191/233 users correct (82%)
27pt | Not attempted 28/95 users correct (29%)

Top Scores

eatmore	100
Marcin.Smulewicz	100
tozangezan	100
Errichto.rekt	100
mnbvmar	100
qwerty787788	100
sevenkplus	100
tczajka	100
fhlasek	100
wata	100

Problem B. again

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

Problem

... we did it again

You know the issue: just as in the Oops problem from Distributed Round 1, we have lost our problem statement and the correct solutions, and we only have these two correct but slow solutions, one per supported language. Once again, we still have our test data. Can you still solve the problem?

Notice that in this problem 20 nodes are used to run both the Small and the Large datasets, which is not the usual number for Distributed Code Jam problems. 20 nodes were also used to run the solutions and produce the answers for the examples.

The C++ solution:

```
#include <message.h>
#include <stdio.h>
#include "again.h"

#define MASTER_NODE 0
#define SENDING_DONE -1
#define LARGE_PRIME 1000000007

int main() {
  if (MyNodeId() == MASTER_NODE) {
    long long result = 0;
    for (int node = 1; node < NumberOfNodes(); ++node) {
      while (true) {
        Receive(node);
        long long value = GetLL(node);
        if (value == SENDING_DONE) {
          break;
        } else {
          result = (result + value) % LARGE_PRIME;
        }
      }
    }
    printf("%lld\n", result);
    return 0;
  } else {
    for (long long i = 0; i < GetN(); ++i) {
      for (long long j = 0; j < GetN(); ++j) {
        long long value = GetA(i) * GetB(j);
        if ((i + j) % NumberOfNodes() == MyNodeId()) {
          PutLL(MASTER_NODE, value);
          Send(MASTER_NODE);
        }
      }
    }
    PutLL(MASTER_NODE, SENDING_DONE);
    Send(MASTER_NODE);
  }
  return 0;
}
```

The Java solution:

```
public class Main {
  static int MASTER_NODE = 0;
  static long SENDING_DONE = -1;
  static long LARGE_PRIME = 1000000007;

  public static void main(String[] args) {
    if (message.MyNodeId() == MASTER_NODE) {
      long result = 0;
      for (int node = 1; node < message.NumberOfNodes(); ++node
        while (true) {
          message.Receive(node);
          long value = message.GetLL(node);
          if (value == SENDING_DONE) {
```

```
            break;
          } else {
            result = (result + value) % LARGE_PRIME;
          }
        }
      }
      System.out.println(result);
    } else {
      for (long i = 0; i < again.GetN(); ++i) {
        for (long j = 0; j < again.GetN(); ++j) {
          long value = again.GetA(i) * again.GetB(j);
          if ((i + j) % message.NumberOfNodes() == message.MyNo
            message.PutLL(MASTER_NODE, value);
            message.Send(MASTER_NODE);
          }
        }
      }
      message.PutLL(MASTER_NODE, SENDING_DONE);
      message.Send(MASTER_NODE);
    }
  }
}
```

Input

The input library is called "again"; see the sample inputs below for examples in your language. It defines three methods:

- **GetN()**:
  - Takes no argument.
  - Returns a 64-bit number.
  - Expect each call to take 0.05 microseconds.
- **GetA(i)**:
  - Takes a 64-bit number in the range 0 ≤ i < GetN().
  - Returns a 64-bit number.
  - Expect each call to take 0.05 microseconds.
- **GetB(i)**:
  - Takes a 64-bit number in the range 0 ≤ i < GetN().
  - Returns a 64-bit number.
  - Expect each call to take 0.05 microseconds.

Output

Output what either of the solutions above would output, if they ran on 20 nodes without any limits on memory, time, number of messages or total size of messages.

Limits

Time limit: 2 seconds.
Memory limit per node: 128 MB.
Maximum number of messages a single node can send: 1000.
Maximum total size of messages a single node can send: 8 MB.
Number of nodes: 20.
$0 \le GetA(i) \le 10^9$, for all i.
$0 \le GetB(i) \le 10^9$, for all i.

Small input

$1 \le GetN() \le 30{,}000$.

Large input

$1 \le GetN() \le 10^8$.

Sample

Input	Output
See input files below.	For sample input 1: 999979007 For sample input 2: 2390 For sample input 3: 0

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

Problem C. lisp_plus_plus

small 3 points *2 minute timeout*	The contest is finished.
large 17 points *10 minute timeout*	The contest is finished.

Problem

Lisp++

Alyssa is a huge fan of the Lisp programming language, but she wants it to have even more parentheses, so she is designing a new language, Lisp++. A valid Lisp++ program consists of one of the following. (In this specification, *P* stands for some valid program -- not necessarily the same program each time.)

- `()` Literally, just an opening parenthesis and a closing parenthesis.
- `(`*P*`)` A program within a pair of enclosing parentheses.
- *PP* Two programs (not necessarily the same), back to back.

Alyssa is working on a compiler for Lisp++. The compiler must be able to evaluate a string consisting of `(` characters and/or `)` characters to determine whether it is a valid Lisp++ program, and provide the user with some helpful information if it is not. If the program is valid, the compiler should print -1. Otherwise, it should print the length of the longest prefix that could be extended into a valid program by adding zero or more additional characters to the end. If that prefix is the empty prefix, the compiler should print 0. In particular, if the input string is not a valid program, but can be extended to a valid program, the compiler should print the length of the input string.

For example:

- `()(()())` is a valid program, so the compiler should print -1.
- `((()))` is not a valid program. The prefix `(())` is the longest prefix that is or could be extended into a valid program (in this case, it already is one), so the compiler should print 4. The only longer prefix is `((()))` (i.e., the entire string), but there is no way to add (any number of) characters to the end of that string to make it into a valid program.
- `)` is not a valid program. The prefix `)` cannot be extended into a valid program. The empty prefix is not a valid program, but it can easily be extended into one (by adding `()`, for example). So the compiler should print 0.

Given a string, what should Alyssa's compiler print?

Input

The input library is called "lisp_plus_plus"; see the sample inputs below for examples in your language. It defines two methods:

- **GetLength()**:
  - Takes no argument.
  - Returns a 64-bit integer: the length of the string.
  - Expect each call to take 0.07 microseconds.
- **GetCharacter(i)**:
  - Takes a 64-bit integer in the range $0 \le i <$ GetLength().
  - Returns a character: either `(` or `)`, the character at position i.
  - Expect each call to take 0.07 microseconds.

Output

Output a single line with a single integer: what the compiler should print, as described above.

Limits

Time limit: 3 seconds.
Memory limit per node: 128 MB.
Maximum number of messages a single node can send: 1000.
Maximum total size of messages a single node can send: 8 MB.
GetCharacter(i) is always `(` or `)`.

Small input

Number of nodes: 10.
$1 \le$ GetLength() $\le 10^6$.

Large input

Number of nodes: 100.
$1 \le \text{GetLength}() \le 10^9$.

Sample

Input	Output
See input files below.	For sample input 1: -1 For sample input 2: 4 For sample input 3: 0

The sample input files are the same examples from the problem statement, in the same order.

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

Problem D. asteroids

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 5 points *2 minute timeout*	The contest is finished.
large 25 points *10 minute timeout*	The contest is finished.

Problem

Asteroids

You are playing a game of DCJAsteroids in your old arcade machine. The game is played on a screen that displays a matrix of square cells. The matrix has a fixed width and infinite height. Each cell may contain your spaceship, an asteroid, or some amount of helium. The goal of the game is to collect helium without colliding with any asteroids. Your spaceship always stays in the bottom row of the screen, and it can only move horizontally.

The game is turn-based. On a turn, you can remain where you are, move to the cell on your left (unless you are in the left-most column), or move to the cell on your right (unless you are in the right-most column). After you move (or not) your spaceship, every other item in the screen moves to the cell directly below it. The cells in the bottom row fall off the bottom of the screen and disappear.

For instance, suppose this is a current arrangement, with X representing your spaceship, # representing a cell occupied by an asteroid, and . representing a cell not occupied by an asteroid (to keep it simple, we do not depict the amount of helium in this example).

```
#.##.
#..#.
..X..
```

If you decide to stay at your current position, then the resulting state after the turn is resolved will be:

```
#.##.
#.X#.
```

If you decide to move left, then the result after the turn is resolved will be:

```
#.##.
#X.#.
```

If you ever move the spaceship into a cell occupied by an asteroid or an asteroid moves into the cell occupied by your spaceship, the spaceship is destroyed. In the example above, if you decided to move right, after the turn is resolved, your spaceship and an asteroid would occupy the same cell, so the spaceship would be destroyed. If you decided to move left two turns in a row, the second move would make your spaceship move into an asteroid, so it would be destroyed as well.

Each cell not occupied by an asteroid may have a different amount of helium. We assign a digit 0 through 9 to each non-asteroid cell (instead of .); the number in a cell represents the number of points you earn from the helium in that cell. Each time your spaceship moves into a cell containing helium or a cell containing helium moves into your spaceship's position, you collect the helium there and are awarded the points. You can only collect the helium in each cell once; after that, your spaceship can still be in that cell, but you collect no additional helium and receive no additional points for it.

You will be given a matrix of characters representing the asteroids and helium that are approaching. You may assume that all rows below and above the input matrix are full of worthless helium (i.e., they are rows of 0s). You may choose the position you are in when the bottom row in the input matrix falls into your current row.

If there is no way to navigate through the entire input matrix without the spaceship being destroyed, output -1. Otherwise, output the maximum number of points you can accumulate while navigating through the entire input matrix. You are considered done navigating only when every row in the input matrix has already disappeared from the screen.

Let us add point values to the empty cells in the example above (assume your current position has 0 value). If you start by going left and then stay for two turns, you would go through these states and emerge without being destroyed:

```
#1##9    00000    00000     00000
#23#9 -> #1##9 -> 00000  -> 00000
66X15    #X3#9    #X##9     0X000
```

In this case you collect 6+2 points in the first turn, 1 in the second turn and 0 in the third, yielding a total of 9 points. If instead you stay for the first turn, then move left, and then stay again, this would happen:

```
#1##9    00000    00000     00000
#23#9 -> #1##9 -> 00000  -> 00000
66X15    #2X#9    #X##9     0X000
```

In this case you collect 3 points in the first turn, 2+1 points in the second and 0 in the third, for a total of 6 points. Any other sequence of moves gets the spaceship destroyed.

Input

The input library is called "asteroids"; see the sample inputs below for examples in your language. It defines three methods:

- **GetHeight()**:
  - Takes no argument.
  - Returns a 64-bit integer: the height of the input matrix.
  - Expect each call to take 0.05 microseconds.
- **GetWidth()**:
  - Takes no argument.
  - Returns a 64-bit integer: the width of both the screen and the input matrix.
  - Expect each call to take 0.05 microseconds.
- **GetPosition(i, j)**:
  - Takes two 64-bit integers in the ranges $0 \le i <$ GetHeight(), $0 \le j <$ GetWidth().
  - Returns a character: the contents of the cell at row i and column j of the input matrix. The character is # if the cell contains an asteroid and a digit if not, representing the point value of the helium you can collect from that cell, as explained above. Rows are numbered from bottom to top (so your spaceship will enter the rows in increasing order). Columns are numbered from left to right, as explained above.
  - Expect each call to take 0.05 microseconds.

Output

Output a single line with a single integer: the maximum number of points you can accumulate in your journey through the asteroids without being destroyed, or -1 if it is impossible to avoid destruction.

Limits

Time limit: 5 seconds.
Memory limit per node: 128 MB.
Maximum number of messages a single node can send: 1000.
Maximum total size of messages a single node can send: 8 MB.
GetPosition(i, j) is a digit or #, for all i and j.

Small dataset

Number of nodes: 10.
$1 \le$ GetHeight() ≤ 1000.
$1 \le$ GetWidth() ≤ 1000.

Large dataset

Number of nodes: 100.
$1 \le$ GetHeight() $\le 30{,}000$.
$1 \le$ GetWidth() $\le 30{,}000$.

Sample

Input	Output
See input files below.	For sample input 1: 10 For sample input 2: 22 For sample input 3: -1

For ease of reading, these are the input matrices in the samples:

```
8##123
#999#1
21##11
52#11#

1#78
0011
#2#9
0136
0#8#
21#9

0##
000
##0
```

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

Problem E. gas_stations

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 8 points *2 minute timeout*	The contest is finished.
large 27 points *10 minute timeout*	The contest is finished.

Problem

Gas Stations

You are about to embark on the road trip of a lifetime, covering millions or even billions of kilometers! Of course, gas for such a long trip can get really expensive, and you are on a budget, so you should plan ahead. Fortunately, the road you are taking has a lot of gas stations: there is one at your starting point, one every kilometer thereafter. There is no gas station at your ending point (not that it would do you any good).

Different gas stations may charge different prices per liter of gas. Your car is really old, so it can run for exactly 1 kilometer with 1 liter of gas. You can never have more liters of gas than the tank can hold, but you can add gas to the tank at any station; you do not need to wait for the tank to be empty. Each time you advance a kilometer, your tank's contents are reduced by 1 liter. It is OK if the tank becomes empty exactly at the end of your trip or exactly as you reach a gas station (in which case you can add gas there).

Given the length of your route, the size of your tank and the price per liter of each station, what is the minimum amount of money you need to complete your trip? Your starting point is exactly at the station at the 0 kilometer mark, and your tank starts empty.

Input

The input library is called "gas_stations"; see the sample inputs below for examples in your language. It defines three methods:

- **GetNumKms()**:
  - Takes no argument.
  - Returns a 64-bit integer: the total length in kilometers of your trip.
  - Expect each call to take 0.15 microseconds.
- **GetTankSize()**:
  - Takes no argument.
  - Returns a 64-bit integer: the maximum amount of liters of gas you can have in your tank.
  - Expect each call to take 0.15 microseconds.
- **GetGasPrice(i)**:
  - Takes a 64-bit integer in the range $0 \le i <$ GetNumKms().
  - Returns a 64-bit integer: the price of each liter of gas in the station exactly i kilometers from your starting point.
  - Expect each call to take 0.15 microseconds.

Output

Output a single line with a single integer: the minimum amount of money you need to pay for all the gas necessary for your trip.

Limits

Time limit: 5 seconds.
Memory limit per node: 128 MB.
Maximum number of messages a single node can send: 1000.
Maximum total size of messages a single node can send: 8 MB.
$1 \le$ GetTankSize() $\le$ GetNumKms().
$1 \le$ GetGasPrice(i) $\le 10^9$, for all i.

Small dataset

Number of nodes: 10.
$1 \le$ GetNumKms() $\le 10^6$.

Large dataset

Number of nodes: 100.
$1 \le$ GetNumKms() $\le 5 \times 10^8$.

Sample

Input	Output
See input files below.	For sample input 1: 7 For sample input 2: 7 For sample input 3: 11

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

Distributed World Finals 2016

Submissions

Testrun

0pt	Not attempted **0/4 users** correct (0%)

encoded_sum

6pt	Not attempted **13/13 users** correct (100%)
11pt	Not attempted **12/12 users** correct (100%)

air_show

5pt	Not attempted **14/14 users** correct (100%)
20pt	Not attempted **1/4 users** correct (25%)

toothpick_sculpture

10pt	Not attempted **9/10 users** correct (90%)
15pt	Not attempted **0/3 users** correct (0%)

gold

15pt	Not attempted **6/10 users** correct (60%)
18pt	Not attempted **4/6 users** correct (67%)

Top Scores

bmerry	65
sevenkplus	65
fhlasek	65
mnbvmar	65
eatmore	52
Merkurev	47
ikatanic	37
tozangezan	32
tmt514	32
wafrelka	22

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. encoded_sum

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 6 points *2 minute timeout*	The contest is finished.
large 11 points *10 minute timeout*	The contest is finished.

Problem

Encoded Sum

You are an archaeologist studying a lost civilization. This civilization used a decimal (base 10) number system like our own: their numbers were made up of digits from 0 through 9, with the most significant digit on the left. However, this civilization used the ten letters A through J, in some order, to represent the ten digits 0 through 9, in a one-to-one mapping.

You have just discovered two scrolls from that civilization. Each contains a number representing the population of one of the two regions of the civilization. The numbers are of the same length. You do not know the civilization's letter-to-digit mapping, but you know it is consistent across the two documents. You want to know the maximal possible sum of those two numbers. Please note that the resulting numbers can have leading zeros.

Given these two scrolls, return the maximal possible sum. Since the output can be a really big number, we only ask you to output the remainder of dividing the result by the prime 10^9+7 (1000000007).

Input

The input library is called "encoded_sum"; see the sample inputs below for examples in your language. It defines three methods:

- **GetLength()**:
  - Takes no argument.
  - Returns a 64-bit integer: the length of the number on either scroll.
  - Expect each call to take 0.1 microseconds.
- **GetScrollOne(i)**:
  - Takes a 64-bit number in the range $0 \le i <$ GetLength()
  - Returns an uppercase character (in the inclusive range A through J) representing the i-th character in the first scroll, counting starting from 0, starting from the left.
  - Expect each call to take 0.1 microseconds.
- **GetScrollTwo(i)**:
  - Takes a 64-bit number in the range $0 \le i <$ GetLength()
  - Returns an uppercase character (in the inclusive range A through J) representing the i-th character in the second scroll, counting starting from 0, starting from the left.
  - Expect each call to take 0.1 microseconds.

Output

Output a single line with a single integer: the maximal sum modulo the prime 10^9+7 (1000000007), as described above.

Limits

Number of nodes: 100. **(Notice that the number of nodes is the same for both the Small and Large datasets.)**
Time limit: 3 seconds.
Memory limit per node: 128 MB.
Maximum number of messages a single node can send: 1000.
Maximum total size of messages a single node can send: 8 MB.
$1 \le$ GetLength() $\le 10^9$.

Small dataset

GetScrollOne(i) is either uppercase A or uppercase B, for all i.
GetScrollTwo(i) is either uppercase A or uppercase B, for all i.

Large dataset

GetScrollOne(i) is an uppercase letter between A and J, inclusive, for all i.
GetScrollTwo(i) is an uppercase letter between A and J, inclusive, for all i.

Sample

Input	Output
See input files below.	For sample input 1: 17 For sample input 2: 1888 For sample input 3: 395283726

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

Problem C. air_show

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

Problem

Air Show

We are planning an awesome air show. The most impressive act features two airplanes doing acrobatic moves together in the air. The flight plan for each plane is already finalized. These plans may cause the planes to get very close to each other, which is dangerous; you have been hired to assess the extent of this risk.

A flight plan for one plane is a sequence of **N** timed segments. Within each segment a plane flies straight at a constant speed. However, a plane may change direction and speed dramatically when changing segments. Formally, the flight plan consists of an ordered list of **N** + 1 3-dimensional points P_0, P_1, ..., $P_{\mathbf{N}}$ and an ordered list of **N** transition times T_0, T_1, ..., $T_{\mathbf{N}-1}$. As its i-th move, for each i in {0, 1, ..., **N** - 1}, the plane following this plan flies from P_i to P_{i+1} at a constant speed in exactly T_i seconds. Since the planes are working together as a single act, the sum of the times for each segment must be equal for both planes.

A transition instant is a time in between two consecutive moves (even when two consecutive moves happen to require no change in speed or direction). That is, the **N** - 1 transition instants for a plane with the flight plan above happen exactly at times T_0, $T_0 + T_1$, $T_0 + T_1 + T_2$, ..., $T_0 + T_1 + ... + T_{N-2}$. The starting and finishing times are not considered transition instants.

Transition instants are the most dangerous times for pilots. Given a minimum safe distance **D**, for each plane *p*, we ask you to count the number of transition instants when *p* is at a distance strictly less than **D** from the other plane. You may consider each plane to be a single point. If both planes occupy the same point at the same instant, no collision occurs; the planes just pass through each other and continue.

The arithmetic for this problem for our official solution requires integers with more than 64 bits. `__int128` is available in our C++ installation and `BigInteger` is available for Java.

Input

The input library is called "air_show"; see the sample inputs below for examples in your language. It defines four methods:

- **GetSafeDistance()**:
  - Takes no argument.
  - Returns a 64-bit integer: the minimum safe distance **D**.
  - Expect each call to take 0.5 microseconds.
- **GetNumSegments()**:
  - Takes no argument.
  - Returns a 64-bit integer: the number of segments **N** of each flight plan.
  - Expect each call to take 0.5 microseconds.
- **GetTime(a, i)**:
  - Takes two 64-bit integers in the ranges $0 \le a < 2$, $0 \le i <$ GetNumSegments().
  - Returns a 64-bit integer: the time used for move i of plane a (shown as T_i above).
  - Expect each call to take 0.5 microseconds.
- **GetPosition(a, i)**:
  - Takes two 64-bit integers in the ranges $0 \le a < 2$, $0 \le i \le$ GetNumSegments().
  - Returns a 64-bit integer: an encoding of point i of the flight plan of plane a (shown as P_i above). Point (x, y, z), with each coordinate ranging between 0 and 2^{20}-1, is encoded as the integer $x \times 2^{40} + y \times 2^{20} + z$.
  - Expect each call to take 2.5 microseconds.

Output

Output a single line with two integers r_0 and r_1 separated by a single space, where r_i is the number of dangerous moments for plane i in which it is strictly closer than GetSafeDistance() to the other plane.

Limits

Number of nodes: 100. **(Notice that the number of nodes is the same for both the Small and Large datasets.)**
Time limit: 14 seconds.
Memory limit per node: 512 MB.
Maximum number of messages a single node can send: 1000.
Maximum total size of messages a single node can send: 128 KB. **(Notice that this is less than usual.)**
$1 \le$ GetSafeDistance() $< 2^{20}$.
$2 \le$ GetNumSegments() $\le 10^8$.

Small dataset

$0 \le$ GetPosition(a, i) $< 2^{40}$, for all a and i. (The x-coordinate of all points is 0, while coordinates y and z range between 0 and $2^{20} - 1$.)
GetTime(a, i) = 1, for all a and i.

Large dataset

$0 \le$ GetPosition(a, i) $< 2^{60}$, for all a and i. (All coordinates range between 0 and $2^{20} - 1$.)
$1 \le$ GetTime(a, i) $< 10^9$, for all a and i.
GetTime(0, 0) + GetTime(0, 1) + ... + GetTime(0, GetNumSegments() - 1) = GetTime(1, 0) + GetTime(1, 1) + ... + GetTime(1, GetNumSegments() - 1). (The sum of all the values of GetTime(a, _) for each a is the same.)
GetTime(0, 0) + GetTime(0, 1) + ... + GetTime(0, GetNumSegments() - 1) $\le 10^{12}$. (The total time of a flight plan does not exceed 10^{12}.)

Sample

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

Note that the last 2 sample cases would not appear in the Small dataset.
Sample input libraries:
Sample input for test 1: air_show.h [CPP] air_show.java [Java]
Sample input for test 2: air_show.h [CPP] air_show.java [Java]
Sample input for test 3: air_show.h [CPP] air_show.java [Java]

Problem D. toothpick_sculpture

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.

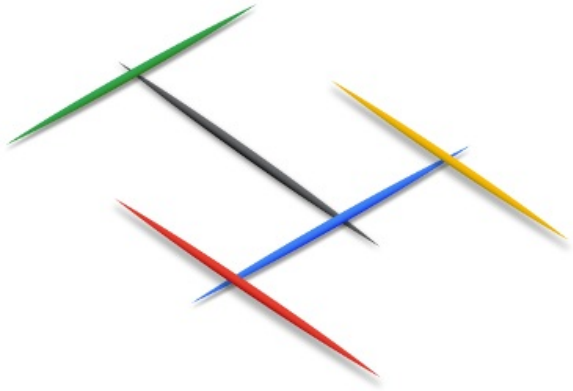
small 10 points *2 minute timeout*	The contest is finished.
large 15 points *10 minute timeout*	The contest is finished.

Problem

Toothpick Sculpture

Your friend Cody-Jamal is an artist. He is working on a new sculpture concept made of toothpicks. Each toothpick has two ends. Each sculpture features a single *foundational toothpick* resting horizontally on the ground. Every other toothpick rests horizontally with its midpoint on one of the ends of exactly one other toothpick. This means each toothpick can have zero, one or two other toothpicks resting on it, forming a binary tree of toothpicks, with the foundational toothpick as the root. Toothpicks do not touch otherwise. Let t_1 be any toothpick, and let t_1 rest on t_2, t_2 rest on t_3, and so on; there is an eventual t_k that is the foundational toothpick.

For instance, in the following picture you can see a sculpture with 5 toothpicks. The black toothpick is the foundational toothpick, resting on the ground. The green and blue toothpicks are resting on the black toothpick, while the red and yellow toothpicks are resting on the blue one.



Cody-Jamal made exactly 1000 sculptures, each using exactly **N** toothpicks, that were exhibited for 1000 days in the largest 1000 cities in the world. As the grand finale, he is planning on placing the foundational toothpicks of 999 of his sculptures on open ends of toothpicks of other sculptures, effectively creating a larger version of a sculpture with the same concept, with 1000 × **N** total toothpicks. He plans on displaying it by placing the foundational toothpick of the structure on top of the Burj Khalifa in Dubai, the tallest building in the world.

Cody-Jamal is enthusiastically telling you his new plan, when your engineering mind notices that the high winds at that altitude will make the structure unstable, so you suggest gluing touching toothpicks together. Cody-Jamal's artistic vision, however, disagrees. The fragility of the construction is part of the conceptual appeal of the piece, he says. You two decide to compromise and use almost invisible carbon nanotube columns to support some toothpicks.

A toothpick (foundational or not) is individually stable if and only if it is supported by a carbon nanotube column. Your math quickly shows that the whole structure can be considered stable if and only if, for each toothpick t that is not the foundational toothpick, either t is individually stable, or t rests on an individually stable toothpick.

Carbon nanotube columns are expensive, though. The cost of the column required to stabilize each toothpick may vary depending on several factors. You decide to write a computer program to assist in choosing a set of toothpicks good enough to make the structure stable while minimizing the sum of the cost of the carbon nanotubes required to stabilize each toothpick.

Input

Toothpicks are numbered with integers between 0 and 1000**N** - 1, inclusive. Integers between 0 and 999 identify the foundational toothpicks of the original sculptures, and 0 is also the foundational toothpick of the large combined sculpture. The numbering of all of the other toothpicks is arbitrary. By way of construction, if you consider a graph made of all toothpicks with two toothpicks

being adjacent if they touch each other, the **N** toothpicks corresponding to any original sculpture form a connected subgraph.

The input library is called "toothpick_sculpture"; see the sample inputs below for examples in your language. It defines three methods:

- **GetNumToothpicks()**:
  - Takes no argument.
  - Returns a 64-bit integer: the number **N** of toothpicks in each original sculpture.
  - Expect each call to take 0.06 microseconds.
- **GetCost(i)**:
  - Takes a 64-bit integer in the range $0 \le i < 1000 \times$ GetNumToothpicks().
  - Returns a 64-bit integer: the cost of stabilizing toothpick i.
  - Expect each call to take 0.06 microseconds.
- **GetToothpickAtEnd(i, e)**:
  - Takes two 64-bit integers in the ranges $0 \le i < 1000 \times$ GetNumToothpicks(), $0 \le e < 2$.
  - Returns a 64-bit integer: the id of the toothpick resting on end e of toothpick i in the large sculpture, or -1 if there is no toothpick resting there.
  - Expect each call to take 0.06 microseconds.

Output

Output a single line with a single integer: the minimum sum of the cost to build the necessary carbon nanotubes and make the structure stable.

Recursive solutions, beware! The process stack size is limited to 8 MB and the JVM thread stack size is limited to 1MB. Attempting to change this programatically will result in a Rule Violation.

Limits

Number of nodes: 100. **(Notice that the number of nodes is the same for both the Small and Large datasets.)**
Time limit: 6 seconds.
Memory limit per node: 512 MB.
Maximum number of messages a single node can send: 5000.
Maximum total size of messages a single node can send: 8 MB.
$1 \le$ GetNumToothpicks() $\le 10^6$.
$1 \le$ GetCost(i) $< 10^9$, for all i.
$-1 \le$ GetToothpickAtEnd(i, e) $<$ GetNumToothpicks() $\times 1000$, for all i, e.
GetToothpickAtEnd(i, e) $\neq 0$, for all i, e.
Let *G* be the graph with toothpicks as nodes and two nodes connected if and only if the corresponding toothpicks touch in the sculpture:
G is a connected tree.
Removing the edges between toothpicks 1 through 999 and the toothpicks they are resting on results in exactly 1000 connected components of GetNumToothpicks() nodes each.

Small dataset

GetToothpickAtEnd(i, 1) = -1, for all i.

Large dataset

No additional limits.

Sample

Input	Output
See input files below.	For sample input 1: 250000 For sample input 2: 37500 For sample input 3: 5529586

Note that the last sample case would not appear in the Small dataset.

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

 Google Home - Terms and Conditions - Privacy Policies and Principles

Powered by

Google Cloud Platform

Problem E. gold

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 15 points *2 minute timeout*	The contest is finished.
large 18 points *10 minute timeout*	The contest is finished.

Problem

Gold

A long, long time ago, on an east-west road in southeastern Asia, an ancient emperor was fleeing from the ruins of his fallen city, carrying a sack full of his gold. At times, he glanced back over his shoulder and saw pursuers chasing him, so he threw out a nugget of gold from his sack to the roadside, hoping to lighten his load and provide a distraction. The story continues, but what happened later is less important - the important part is that there is gold lying by the roadside to be picked up!

So, you took your trusty metal detector, and went to search for gold on the road. You have a really fast car, but the detector itself is somewhat slow to set up and operate, and also a bit inaccurate - it can only tell you in which direction the nearest nugget of gold is, but not how far. Also, your car cannot handle off-road driving, so you cannot triangulate; you will just need to search a bit longer.

We will represent the road as a straight line of length **L**. There will be **N** nuggets of gold on the road, at integer positions, no more than one nugget at each position. You will be able to set up the detector at integer positions on the road. After setting up, the detector will provide one of the four possible answers:

- The nearest nugget is to the east (towards decreasing position numbers),
- The nearest nugget is to the west (towards increasing position numbers),
- The nearest nugget to the west and to the east are equally distant, or
- There is a nugget at this position.

Input

The input library will be called "gold", see the sample inputs below for examples in your language. It will define three methods:

- **NumberOfNuggets()**:
  - Takes no argument.
  - Returns a 64-bit integer: the number of nuggets on the road.
  - Expect each call to take 0.2 microseconds.
- **RoadLength()**:
  - Takes no argument.
  - Returns a 64-bit integer: the number of positions on the road.
  - Expect each call to take 0.2 microseconds.
- **Search(i)**:
  - Takes one 64-bit integer argument in the range $0 \le i <$ RoadLength().
  - Returns a character describing the output of the metal detector: < if the nearest nugget is to the east, > if it is to the west, = if the nearest nuggest to the east and west are equally distant, or X if there is a nugget as position *i*.
  - Expect each call to take 0.2 microseconds.

Output

As printing all the nugget positions would require a lot of printing, you should output one number - the bitwise XOR of the positions of all the nugget positions - as a proof you found them all.

Limits

Number of nodes: 100. **(Notice that the number of nodes is the same for both the Small and Large datasets.)**
Time limit: 15 seconds. **(There is a 10 second overhead of initializing the test data that is not counted against this limit, so each reported time is 10 seconds more than the time your solution executed, up to a maximum of 25.)**
Memory limit per node: 512 MB.
Maximum number of messages a single node can send: 5000.
Maximum total size of messages a single node can send: 8 MB.

$1 \leq$ NumberOfNuggets() $\leq 10^7$.
$1 \leq$ RoadLength() $\leq 10^{11}$.

Small input

The positions of the nuggets will be generated using a pseudorandom number generator such that the probability of any subset of positions being chosen is the same.

Large input

No additional limits.

Sample

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

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