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.

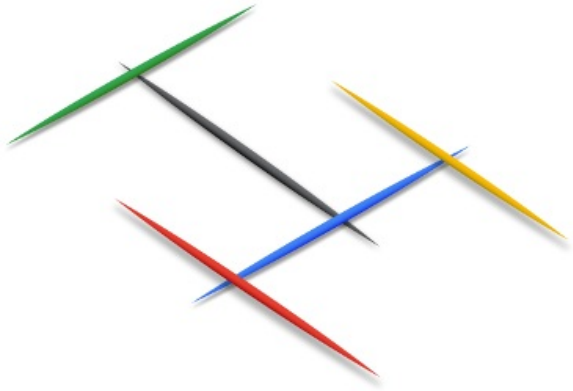
small 10 points *2 minute timeout*	The contest is finished.
large 15 points *10 minute timeout*	The contest is finished.

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]
