acm International Collegiate Programming Contest

IBM event sponsor

2015 ACM ICPC
Southeast USA Regional
Programming Contest
Division 1

Hosted by:

College of Charleston
Florida Institute of Technology
Georgia Institute of Technology
University of West Florida

Airports

An airline company offers flights out of ***n*** airports. The flight time between any given pair of airports is known, but may differ on direction due to things like wind or geography. Upon landing at a given airport, a plane must be inspected before it can be flown again. This inspection time is dependent on the airport at which the inspection is taking place.

Given a set of ***m*** flights that the airline company must realize, determine the minimum number of planes that the company needs to purchase. The airline may add unscheduled flights to move the airplanes around if that would reduce the total number of planes needed.

Input

Each input will consist of a single test case. Note that your program may be run multiple times on different inputs. The first line of input contains two integers ***n*** and ***m*** ($2 \le n,m \le 500$). The next line contains ***n*** space-separated nonnegative integers less than or equal to 10^6, where the i^{th} integer denotes the amount of time (in minutes) that an inspection takes at airport *i*.

Each of the next ***n*** lines contains ***n*** space-separated nonnegative integers less than or equal to 10^6. On the i^{th} line, The j^{th} integer indicates the amount of time it takes to fly from airport *i* to airport *j*. It takes no time to fly from an airport to itself. Note that the flight time from airport *i* to *j* is not necessarily the same as the flight time from airport *j* to *i*.

The next ***m*** lines contain three space-separated integers, ***s***, ***f***, and ***t***, ($1 \le s,f \le n$, $s \ne f$, $1 \le t \le 10^6$) indicating that a flight must start at airport ***s***, end at airport ***f***, and fly out from airport ***s*** at exactly time ***t*** heading directly to airport ***f***.

Output

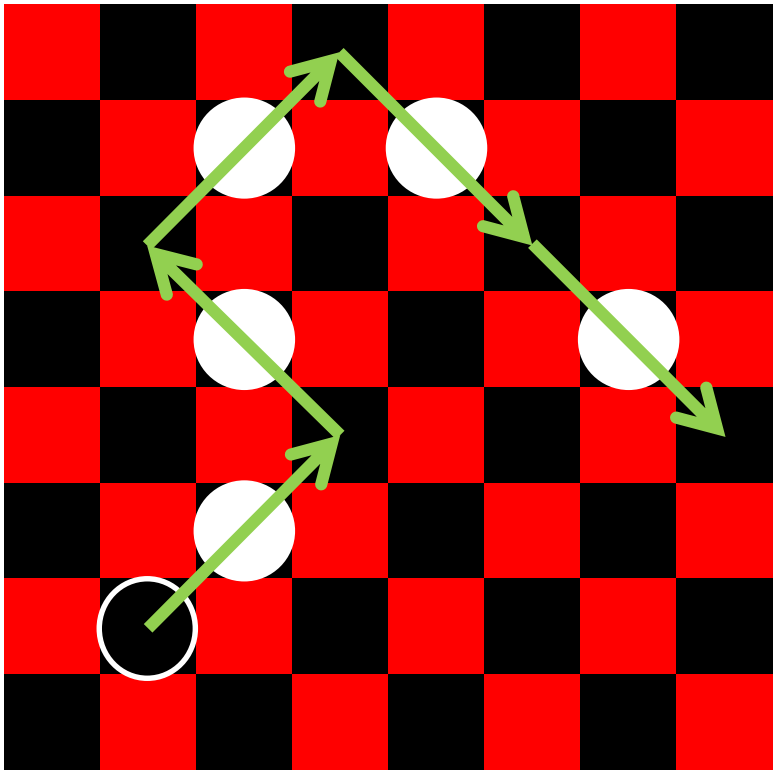
Output a single positive integer indicating the minimum number of planes the airline company must purchase in order to realize the ***m*** requested flights.

Sample Input	Sample Output
2 2 1 1 0 1 1 0 1 2 1 2 1 1	2
2 2 1 1 0 1 1 0 1 2 1 2 1 3	1
5 5 72 54 71 94 23 0 443 912 226 714 18 0 776 347 810 707 60 0 48 923 933 373 881 0 329 39 511 151 364 0 4 2 174 2 1 583 4 3 151 1 4 841 4 3 993	3

Checkers

Checkers is played on a square ***n*x*n*** grid (typically ***n*** equals 8, 10, or 12, but for this problem, ***n*** will range from 2 up to 26). The board has squares colored red and black, and all pieces move only on the black squares. Red and Black squares alternate, so that no two squares that share a side are ever of the same color. The two players are called *Black* and *White*, and their pieces are so colored. There are two kinds of pieces, *Checkers* and *Kings*, but for this problem, we will only be concerned with Kings. Kings may jump a piece of the other color in one diagonal hop, capturing the piece (removing it from the board). If such a capture is possible, the jumping piece may continue jumping and capturing pieces of the other color until no more jumps are possible. A King may jump in any of the four diagonal directions.

In order to perform a jump, the piece jumped must be immediately adjacent (diagonally) to the piece jumping, and the square on the other side of the jumped piece must be vacant.



In this problem, it is Black's turn to move. Given a position of checkers, you must determine if it is possible for a Black King to jump all of White's Kings in a single move, and if so, how many Black Kings are able to do so.

Input

Each input will consist of a single test case. Note that your program may be run multiple times on different inputs. The first line of input contains an integer ***n*** (2≤***n***≤26), the size of the board. The following ***n*** lines describe the board. Each line will contain exactly ***n*** characters, and each character will be one of '**.**', '**_**', '**B**', or '**W**', indicating the contents of that square, as follows:

. Indicates a Red square. No Kings may be placed on a Red square.

_ indicates a Black square that is unoccupied.

B indicates a Black square with a Black King.

W indicates a Black square with a White King.

You may assume that the given board is well-formed; that is, Black and Red squares will alternate through every row and every column, and no Kings will be on any Red square.

Output

Output a single integer indicating the number of Black Kings that can capture all of the White Kings in a single move.

Sample Input	**Sample Output**
8 ·_·_·_·_ _·_·_·_· .W._.B._ _.W.W._. .W.B._·_ _·_·_·_· .W._.W._ _·_·_·_·	0
10 ·_·_·_·_·_ _.W.W._·_· ·_·_·_·_·_ _.W.W._·_· ·_·_·_·_·_ _.W.W.W.W. ·_·_·_·_·_ _.W.W.W.W. .B.B.B._·_ _·_·_·_·_·	1 (Note: It's the middle B that can jump all of the Ws).

Coverage

A cellular provider has installed ***n*** towers to support their network. Each tower provides coverage in a 1km radius, and no two towers are closer than 1km to each other. The coverage region of this network is therefore the set of all points that are no more than 1km away from at least one tower. The provider wants as much of this region as possible to be connected, in the sense that a user at any point within a connected subregion can travel to any other point within the connected subregion without having to exit the subregion. Their current installation of towers may or may not already form a single connected region, but they have the resources to build one more tower wherever they want, including within 1km of an existing tower. Given that the provider is able to build one more tower, what is the maximum number of towers (including the new one) that can be included within a single connected subregion of coverage?

Input

Each input will consist of a single test case. Note that your program may be run multiple times on different inputs. The first line of input consists of a single integer ***n*** ($1 \le n \le 5{,}000$) denoting the number of existing towers. Next follow ***n*** lines, each with 2 space-separated floating-point numbers ***x*** and ***y*** ($0 \le x, y \le 100{,}000$), denoting the location of a tower in km. It will be guaranteed that the optimal number of towers will not change even if the coverage radius of all the towers is increased or decreased by 10^{-6} km.

Output

Ouput a single integer, denoting the maximum number of towers that can be within a single connected subregion of the network after installing one additional tower.

Sample Input	Sample Output
5 1.0 1.0 3.1 1.0 1.0 3.1 3.1 3.1 4.2 3.1	6
5 1.0 1.0 3.1 1.0 1.0 3.1 3.1 3.1 10.0 10.0	5

Gears

A set of gears is installed on the plane. You are given the center coordinate and radius of each gear, which are all integer-valued. For a given source and target gear, indicate what happens to the target gear if you attempt to turn the source gear. Possibilities are:

- The source gear cannot move, because it would drive some gear in the arrangement to turn in both directions.
- The source gear can move, but it is not connected to the target gear.
- The source gear turns the target gear, at a certain ratio

If the source gear cannot move, give this result, even if the source and target gears are not connected.

Input

Each input will consist of a single test case. Note that your program may be run multiple times on different inputs. The first line of input contains a single integer n ($1 \le n \le 1{,}000$), the total number of gears. Following this will be n lines, one per gear, containing the x, y ($-10{,}000 \le x,y \le 10{,}000$) and r ($1 \le r \le 10{,}000$) values for the gear, where (x,y) is the position of the axle of the gear, and r is its radius. Assume that the teeth of the gears are properly designed, and accounted for in the radius, so that any gear will mesh with any other gear if (and only if) they are tangent to each other. The gears will never overlap.

Output

Output a single line, with the following content, based on the result:

- `-1` if the source gear cannot move.
- `0` if the source gear can move but is not connected to the target.
- a b if the source gear moves the target gear, where a and b are two space-separated integers, and $a:b$ is the ratio of source gear revolutions to target gear revolutions reduced to its lowest form (i.e. they have no common factor other than 1).
    - a is always positive.
    - If the target turns in the same direction as the source, b is positive.
    - If the target turns in the opposite direction as the source, b is negative.

Sample Input	Sample Output
2 0 0 100 0 300 200	2 -1
2 0 0 100 0 300 100	0
16 10 10 5 20 10 5 30 10 5 40 10 5 10 20 5 20 20 5 30 20 5 40 20 5 10 30 5 20 30 5 30 30 5 40 30 5 10 40 5 20 40 5 30 40 5 40 40 5	1 1
3 0 0 1 0 3 2 4 0 3	-1

Grid

You are on an ***n***x***m*** grid where each square on the grid has a digit on it. From a given square that has digit ***k*** on it, a *Move* consists of jumping exactly ***k*** squares in one of the four cardinal directions. A move cannot go beyond the edges of the grid; it does not wrap. What is the minimum number of moves required to get from the top-left corner to the bottom-right corner?

Input

Each input will consist of a single test case. Note that your program may be run multiple times on different inputs. The first line of input contains two space-separated integers ***n*** and ***m*** ($1 \le n,m \le 500$), indicating the size of the grid. It is guaranteed that at least one of ***n*** and ***m*** is greater than 1.

The next ***n*** lines will each consist of ***m*** digits, with no spaces, indicating the ***n***x***m*** grid. Each digit is between 0 and 9, inclusive.

The top-left corner of the grid will be the square corresponding to the first character in the first line of the test case. The bottom-right corner of the grid will be the square corresponding to the last character in the last line of the test case.

Output

Output a single integer on a line by itself representing the minimum number of moves required to get from the top-left corner of the grid to the bottom-right. If it isn't possible, output **-1**.

Sample Input	Sample Output
2 2 11 11	2
2 2 22 22	-1
5 4 2120 1203 3113 1120 1110	6

Hilbert Sort

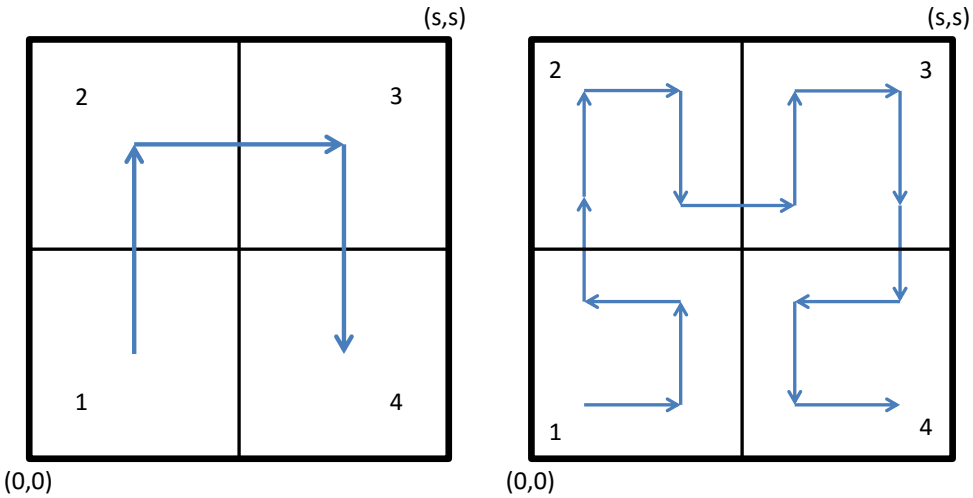
Sorting numerical data not only makes it easy to search for a particular item, but also makes better use of a CPU's cache: any segment of data that's contiguous in memory will describe a set of items that are similar in some sense. Things get more complicated if our data represents points on a 2D grid. If points (***x***,***y***) are sorted by ***x***, breaking ties by ***y***, then adjacent points will have similar ***x*** coordinates but not necessarily similar ***y***, potentially making them far apart. To better preserve distances, we can sort the data along a space-filling curve.

The Hilbert curve starts at the origin (**0**,**0**), finishes at (***s***,**0**), in the process traversing every point in axis-aligned square with corners at (**0**,**0**) and (***s***,***s***). It has the following recursive construction: split the square into four quadrants meeting at (***s***/**2**, ***s***/**2**). Number them 1 to 4, starting at the lower left and moving clockwise. Recursively fill each of them with a suitably rotated and scaled copy of the full Hilbert curve.

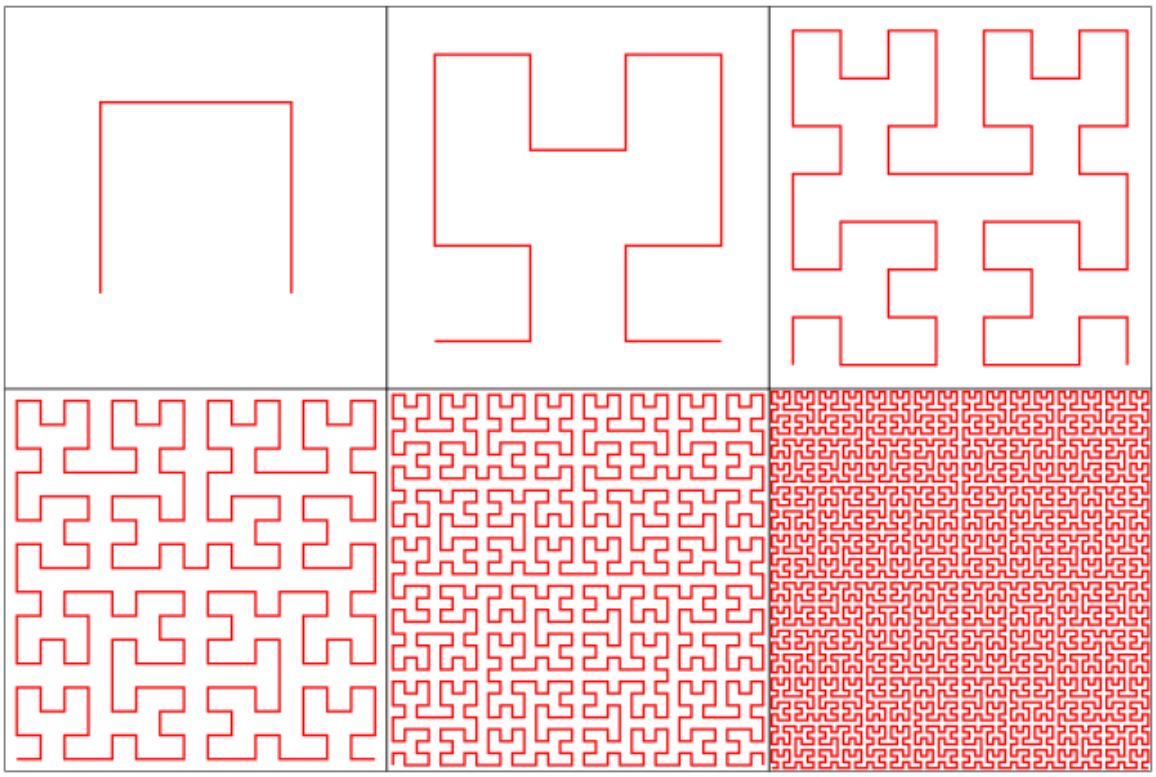
Start with a single point at (***s***/**2**,***s***/**2**). Then, repeat these steps:

- Scale and copy the current construction into each of the 4 quadrants.
- Rotate quadrant 1 by -90 degrees and flip it vertically, so that the start of the curve is closest to the lower left corner (**0**,**0**).
- Rotate quadrant 4 by 90 degrees and flip it vertically, so that the end of the curve is closest to the lower right corner (***s***,**0**).
- Now, connect the end of the curve in quadrant 1 to the start of the curve in quadrant 2, connect the end of quadrant 2 to the start of quadrant 3, and the end of quadrant 3 to the start of quadrant 4.

Here are the first two iterations:



The Hilbert Curve is built by repeating this construction infinitely many times. The following diagram shows the first six steps of building the Hilbert Curve:



Given some places of interest inside of a square region, sort them according to when the Hilbert curve visits them, starting from (0,0). Without going into gory detail about Fractal theory, note that making ***s*** odd guarantees that all integer points are visited just once, so their visitation order in relation to each other is unambiguous.

Input

Each input will consist of a single test case. Note that your program may be run multiple times on different inputs. The first line of input contains two space-separated integers ***n*** and ***s*** ($1 \le n \le 100{,}000$, $1 \le s < 10^9$, ***s*** is odd). The next ***n*** lines describe locations of interest by space-separated integers ***x*** and ***y*** ($0 \le x, y \le s$). No two locations will share the same position.

Output

Output the ***n*** ordered pairs, one per line, with ***x*** and ***y*** separated by a space, Hilbert-sorted according to their positions.

Sample Input	Sample Output
14 25 5 5 5 10 5 20 10 5 10 10 10 15 10 20 15 5 15 10 15 15 15 20 20 5 20 10 20 20	5 5 10 5 10 10 5 10 5 20 10 20 10 15 15 15 15 20 20 20 20 10 15 10 15 5 20 5

The Magical 3

Three is a magic number.

Yes it is; it's a magic number.

Somewhere in the ancient, mystic trinity,

You get three as a magic number.

- Schoolhouse Rock

According to Pythagoras and the Pythagorean School, the number 3 - which they called *triad* - is the noblest of all digits, as it is the only positive integer to equal the sum of all of the positive integers below it (1+2=3), and it is the only positive integer whose sum with those below equals the product of them and itself (1+2+3=1x2x3).

Your task is to find the magic – the magic **3**, that is – when it can be the last digit in a representation of a positive integer in some base. Consider, for example, the number 11. It can be represented as *ONE-THREE* (13) in base 8 and as *TWO-THREE* (23) in base 4. You are to write a program that will find the smallest base for a given positive integer where the input number's representation in that base ends in 3. This is possible for all integers greater than 6.

Input

Each input will consist of a single test case. Note that your program may be run multiple times on different inputs. Each test case will consist of a single line with a single integer ***n*** ($7 \le n < 2^{31}$).

Output

For each test case, output a single integer representing the smallest base in which the input ***n*** ends with a 3.

Sample Input	Sample Output
11	4
42	13
9876	3291

Racing Gems

You are playing a racing game. Your character starts at the X-axis line (***y***=0) and proceeds up the racetrack, which has a boundary at the line ***x***=0 and ***x***=***w***. The finish is at ***y***=***h***, and the game ends when you reach that line. You proceed at a fixed vertical velocity ***v***, but you can control your horizontal velocity to be any value between -***v***/**r** and ***v***/***r***, where ***r*** is a fixed ratio. You may change your horizontal velocity at any time, but your vertical velocity must remain fixed.

There are gems at specific points on the race track. Your job is to collect as many gems as possible (they all have the same value).

How many gems can you collect? You may start at any horizontal position you want (but your vertical position must be 0 at the start).

Input

Each input will consist of a single test case. Note that your program may be run multiple times on different inputs. The first line will contain four integers: ***n*** ($1 \le n \le 10^5$) is the number of gems, ***r*** ($1 \le r \le 10$) is the ratio of vertical velocity to maximum horizontal speed, ***w*** ($1 \le w \le 10^9$) is the width of the track, and ***h*** ($1 \le h \le 10^9$) is the height of the finish line. Following this will be ***n*** lines, each containing an integer ***x*** and ***y*** coordinate ($0 \le x \le w, 1 \le y \le h$), containing the coordinate of a gem. All gems will lie on the race track. None will be on the start line.

Output

Output a single integer on a line by itself representing the maximum number of gems that you can collect.

Sample Input	Sample Output
5 1 10 10 8 8 5 1 4 6 4 7 7 9	3
5 1 100 100 27 75 79 77 40 93 62 41 52 45	3
10 3 30 30 14 9 2 20 3 23 15 19 13 5 17 24 6 16 21 5 14 10 3 6	4

Simplicity

For a string of letters, define the *Simplicity* of the string to be the number of distinct letters in the string. For example, the string **string** has simplicity 6, and the string **letter** has simplicity 4.

You like strings which have simplicity either 1 or 2. Your friend has given you a string and you want to turn it into a string that you like. You have a magic eraser which will delete one letter from any string. Compute the minimum number of letters you must erase in order to turn the string into a string with simplicity at most 2.

Input

Each input will consist of a single test case. Note that your program may be run multiple times on different inputs. The input will consist of a line with a single string consisting of at least 1 and at most 100 lowercase letters.

Output

Output a single integer, indicating the minimum number letters you need to erase in order to give the string a simplicity of 1 or 2.

Sample Input	Sample Output
string	4
letter	2
aaaaaa	0
uncopyrightable	13
ambidextrously	12
assesses	1
assassins	2

Weightlifting

In competitive weightlifting, you must perform a sequence of lifts. You have a constant strength ***s***, and a decreasing energy reserve ***e***. For each lift, you may choose any positive (not necessarily integer) weight *w* to attempt. If ***s*≥*w***, the lift succeeds and your energy goes down by $e_{success}$; if ***s*<*w***, the lift fails and your energy goes down by $e_{failure}$. You may continue attempting lifts as long as ***e*>0**. If at any point ***e*≤0**, you can make no further attempts. Your score is the maximum weight you successfully lift or 0 if every attempt failed.

Ideally, you should lift exactly at your strength limit. However, you do not know your strength **s**. You only know that you can definitely lift the empty bar (25kg), and that the maximum conceivable lift is (225kg). How close to an optimal score can you guarantee? That is, what's the smallest *d* for which you can ensure a score of at least ***s*-*d***?

For example, suppose ***e*=4**, $e_{success}$=1 and $e_{failure}$=2. You try to lift 200kg and fail. Now, ***e*=2**. You try 100kg and succeed. Now, ***e*=1**. You try 150kg and succeed. Now, ***e*=0** and you must stop. You know that you can lift 150kg, but you cannot lift 200kg. Your strength ***s*** must be somewhere between 150kg and 200kg. You scored 150, your optimal score might be as high as (just under) 200. You still don't know ***s***, but you know you're within 50. In this case, ***d*=50**.

That's a specific example, and the strategy used is certainly not optimal. You can do better. What's the smallest value of ***d*** you can get so that you can guarantee a score of at least ***s*-*d*** for any and all possible values of ***s***?

Input

Each input will consist of a single test case. Note that your program may be run multiple times on different inputs. The input consists of a single line with 3 space-separated integers ***e***, $e_{success}$, $e_{failure}$ ($1 \le e, e_{success}, e_{failure} \le 10^7$), where ***e*** is your beginning energy reserve, $e_{success}$ is the amount of energy expended in a successful lift, and $e_{failure}$ is the amount of energy expended in a failed lift.

Output

Output a single line with a real number ***d***, rounded to exactly 6 decimal places, which is the minimum weight in kg such that you can ensure a score of at least ***s*-*d***.

Sample Input	Sample Output
1 3 3	112.500000
12 3 3	13.333333
3000 2 3	0.000000