1 Saruman's army (`army{.i, .o}`)

1.1 Description

Saruman the White must lead his army along a straight path from Isengard to Helm's Deep. To keep track of his forces, Saruman distributes seeing stones, known as *palantirs*, among the troops. Each palantir has a maximum effective range of R units, and must be carried by some troop in the army (i.e., palantirs are not allowed to "free float" in mid-air). Help Saruman take control of Middle Earth by determining the minimum number of palantirs needed for Saruman to ensure that each of his minions is within R units of some palantir.

1.2 Input

The input test file will contain multiple cases. Each test case begins with a single line containing an integer R, the maximum effective range of all palantirs (where $0 \leq R \leq 1000$), and an integer n, the number of troops in Saruman's army (where $1 \leq n \leq 1000$). The next line contains n integers, indicating the positions $x_1, \ldots, x_n$ of each troop (where $0 \leq x_i \leq 1000$). The end-of-file is marked by a test case with $R = n = -1$.

```
0 3
10 20 20
10 7
70 30 1 7 15 20 50
-1 -1
```

1.3 Output

For each test case, print a single integer indicating the minimum number of palantirs needed.

```
2
4
```

In the first test case, Saruman may place a palantir at positions 10 and 20. Here, note that a single palantir with range 0 can cover both of the troops at position 20.

In the second test case, Saruman can place palantirs at position 7 (covering troops at 1, 7, and 15), position 20 (covering positions 20 and 30), position 50, and position 70. Here, note that palantirs must be distributed among troops and are not allowed to "free float." Thus, Saruman cannot place a palantir at position 60 to cover the troops at positions 50 and 70.

2 Fibonacci (`fib{.i , .o}`)

2.1 Description

In the Fibonacci integer sequence, $F_0 = 0$, $F_1 = 1$, and $F_n = F_{n-1} + F_{n-2}$ for $n \geq 2$. For example, the first ten terms of the Fibonacci sequence are:

$$0, 1, 1, 2, 3, 5, 8, 13, 21, 34, \ldots$$

An alternative formula[1] for the Fibonacci sequence is

$$\begin{bmatrix} F_{n+1} & F_n \\ F_n & F_{n-1} \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}^n = \underbrace{\begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix} \cdots \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}}_{n \text{ times}}.$$

Given an integer n, your goal is to compute the last 4 digits of F_n.

2.2 Input

The input test file will contain multiple test cases. Each test case consists of a single line containing n (where $0 \leq n \leq$ 1,000,000,000). The end-of-file is denoted by a single line containing the number -1.

```
0
9
999999999
1000000000
-1
```

2.3 Output

For each test case, print the last four digits of F_n. If the last four digits of F_n are all zeros, print '0'; otherwise, omit any leading zeros (i.e., print F_n mod 10000).

```
0
34
626
6875
```

[1] As a reminder, matrix multiplication is associative, and the product of two 2×2 matrices is given by

$$\begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix} \begin{bmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{bmatrix} = \begin{bmatrix} a_{11}b_{11} + a_{12}b_{21} & a_{11}b_{12} + a_{12}b_{22} \\ a_{21}b_{11} + a_{22}b_{21} & a_{21}b_{12} + a_{22}b_{22} \end{bmatrix}.$$

Also, note that raising any 2×2 matrix to the 0th power gives the identity matrix:

$$\begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix}^0 = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}.$$

3 Football (`football{.i, .o}`)

3.1 Description

Consider a single-elimination football tournament involving 2^n teams, denoted $1, 2, \ldots, 2^n$. In each round of the tournament, all teams still in the tournament are placed in a list in order of increasing index. Then, the first team in the list plays the second team, the third team plays the fourth team, etc. The winners of these matches advance to the next round, and the losers are eliminated. After n rounds, only one team remains undefeated; this team is declared the winner.

Given a matrix $P = [p_{ij}]$ such that p_{ij} is the probability that team i will beat team j in a match determine which team is most likely to win the tournament.

3.2 Input

The input test file will contain multiple test cases. Each test case will begin with a single line containing n ($1 \leq n \leq 7$). The next 2^n lines each contain 2^n values; here, the jth value on the ith line represents p_{ij}. The matrix P will satisfy the constraints that $p_{ij} = 1.0 - p_{ji}$ for all $i \neq j$, and $p_{ii} = 0.0$ for all i. The end-of-file is denoted by a single line containing the number -1.

Note that each of the matrix entries in this problem is given as a floating-point value. To avoid precision problems, make sure that you use either the `double` data type instead of `float`.[2]

```
2
0.0 0.1 0.2 0.3
0.9 0.0 0.4 0.5
0.8 0.6 0.0 0.6
0.7 0.5 0.4 0.0
-1
```

3.3 Output

The output file should contain a single line for each test case indicating the number of the team most likely to win. To prevent floating-point precision issues, it is guaranteed that the difference in win probability for the top two teams will be at least 0.01.

```
2
```

In the test case above, teams 1 and 2 and teams 3 and 4 play against each other in the first round; the winners of each match then play to determine the winner of the tournament. The probability that team 2 wins the tournament in this case is:

$$\begin{aligned} P(\text{2 wins}) &= P(\text{2 beats 1})P(\text{3 beats 4})P(\text{2 beats 3}) + P(\text{2 beats 1})P(\text{4 beats 3})P(\text{2 beats 4}) \\ &= p_{21}p_{34}p_{23} + p_{21}p_{43}p_{24} = 0.9 \cdot 0.6 \cdot 0.4 + 0.9 \cdot 0.4 \cdot 0.5 \\ &= 0.396. \end{aligned}$$

The next most likely team to win is team 3, with a 0.372 probability of winning the tournament.

[2]The input files for this problem quickly get very large; in some cases, simply reading the matrix can be a speed bottleneck for your program. If you are using C++, using `scanf("%lf", &x);` instead of `cin >> x;` to read floating-point values from stdin may greatly speed up your program.

4 Robot (`robot{.i, .o}`)

4.1 Decription

Let $(x_1, y_1), \ldots, (x_n, y_n)$ be a collection of n points in a two-dimensional plane. Your goal is to navigate a robot from point (x_1, y_1) to point (x_n, y_n). From any point (x_i, y_i), the robot may travel to any other point (x_j, y_j) at most R units away at a speed of 1 unit per second. Before it does this, however, the robot must turn until it faces (x_j, y_j); this turning occurs at a rate of 1 degree per second.

Compute the shortest time needed for the robot to travel from point (x_1, y_1) to (x_n, y_n). Assume that the robot initially faces (x_n, y_n). To prevent floating-point precision issues, you should use the `double` data type instead of `float`. It is guaranteed that the unrounded shortest time will be no more than 0.4 away from the closest integer. Also, if you decide to use inverse trigonometric functions in your solution (hint, hint!), try `atan2()` rather than `acos()` or `asin()`.

4.2 Input

The input test file will contain multiple test cases. Each test case will begin with a single line containing an integer R, the maximum distance between points that the robot is allowed to travel (where $10 \leq R \leq 1000$), and an integer n, the number of points (where $2 \leq n \leq 20$). The next n lines each contain 2 integer values; here, the ith line contains x_i and y_i (where $-1000 \leq x_i, y_i \leq 1000$). Each of the points is guaranteed to be unique. The end-of-file is denoted by a test case with $R = n = -1$.

```
10 2
0 0
7 0
10 3
0 0
7 0
14 5
10 3
0 0
7 0
14 10
-1 -1
```

4.3 Output

The output test file should contain a single line per test case indicating the shortest possible time in second (rounded to the nearest integer) required for the robot to travel from (x_1, y_1) to (x_n, y_n). If no trip is possible, print "impossible" instead.

```
7
71
impossible
```

5 Spam (spam{.i, .o})

5.1 Description

To thwart content-based spam filters, spammers often modify the text of a spam email to prevent its recognition by automatic filtering programs. For any plain text string s (containing only upper-case letters), let $\Phi(s)$ denote the string obtained by substituting each letter with its "spam alphabet" equivalent:

A	4	(four)	N	||	(pipe backslash pipe)
B	|3	(pipe three)	O	0	(zero)
C	(	(left-parenthesis)	P	|0	(pipe zero)
D	|)	(pipe right-parenthesis)	Q	(,)	(left-parenthesis comma right-parenthesis)
E	3	(three)	R	|?	(pipe question-mark)
F	|=	(pipe equals)	S	5	(five)
G	6	(six)	T	7	(seven)
H	#	(pound)	U	|_|	(pipe underscore pipe)
I	|	(pipe)	V	\/	(backslash forward-slash)
J	_|	(underscore pipe)	W	\/\/	(backslash forward-slash backslash forward-slash)
K	|<	(pipe less-than)	X	><	(greater-than less-than)
L	|_	(pipe underscore)	Y	-/	(minus forward-slash)
M	|\/|	(pipe backslash forward-slash pipe)	Z	2	(two)

In this scheme, any plain text message s corresponds to exactly one spam-encoded message $\Phi(s)$. The reverse, however, is not necessarily true: a spam-encoded message may correspond to more than one plain text message.

Given a plain text message s, your goal is to determine the number of unique plain text messages whose spam encoding is $\Phi(s)$.

5.2 Input

The input test file will contain multiple test cases. Each test case consists of a single line containing a plain text string s containing from 1 to 100 upper-case letters. The end-of-file is denoted by a single line containing the word "end".

```
BU
UJ
THEQUICKBROWNFOXJUMPEDOVERTHELAZYDOGS
end
```

5.3 Output

For each test case, print the number of unique plain text messages (including the original message) whose spam encoding is $\Phi(s)$. The number of unique plain text messages is guaranteed to be no greater than 1,000,000,000.

```
6
5
144
```

In the first test case, the spam encoding of 'BU' is '|3|_|'. The 6 plain text messages with this spam encoding are 'BU', 'IEU', 'BIJ', 'IEIJ', 'BLI', and 'IELI'.

In the second test case, the spam encoding of 'UJ' is '|_|_|'. The 5 plain text messages with this spam encoding are 'UJ', 'LU', 'IJJ', 'LLI', and 'LIJ'.

6 Tic-Tac-Toe (`ttt.{.i, .o}`)

6.1 Description

In the game of tic-tac-toe, two players take turns marking squares of an init7ially empty 3×3 grid with either X's or O's. The first player always marks squares using X's, whereas the second player always marks squares using O's. If at any point during the game either player manages to mark three consecutive squares in a row, column, or diagonal with his/her symbol, the game terminates.

Given a board configuration, your goal is to determine whether the board configuration represents the possible final state of a valid tic-tac-toe game.

6.2 Input

The input test file will contain multiple cases. Each test case consists of a single line containing 9 characters, which represent the 9 squares of a tic-tac-toe grid, given one row at a time. Each character on the line will either be 'X', 'O' (the letter O), or '.' (indicating an unfilled square). The end-of-file is marked by a single line containing the word "end".

```
XXXOO.XXX
XOXOXOXOX
OXOXOXOXO
XXOOOXXOX
XO.OX...X
.XXX.XOOO
X.OO..X..
OOXXXOOXO
end
```

6.3 Output

For each input test case, write a single line containing either the word "valid" or "invalid" indicating whether the given board configuration is the final state of some possible tic-tac-toe game.

```
invalid
valid
invalid
valid
valid
invalid
invalid
invalid
```