IPSC 2011

problems and sample solutions

Pancakes!

A *filled pancake* consists of a *pancake* and its *filling*. In this problem, you will be given a list of ingredients you have and your task is to prepare as many filled pancakes as possible.

Given 8 cups of milk, 8 egg yolks, 4 tablespoons of sugar, 1 teaspoon of salt, and 9 cups of flour, we can prepare 16 pancakes. For the purpose of this problem, we assume that these quantities scale arbitrarily: for any $x \geq 0$, if you have x times as much of every ingredient, you can prepare $\lfloor 16x \rfloor$ pancakes.

Given the necessary ingredients, you can prepare the following fillings:

- banana pancake: 1 banana
- strawberry pancake: 30 g of strawberry jam
- chocolate pancake: 25 g of chocolate spread
- walnut pancake: 10 walnuts
- $\star$ banana pancake with chocolate: 3/4 of a banana, 10 g of chocolate spread
- $\star$ walnut pancake with chocolate: 7 walnuts, 10 g of chocolate spread
- $\star$ grand mix pancake: 1/3 of a banana, 3 walnuts, 5 g of strawberry jam, and 5 g of chocolate spread

Banana pieces can be combined. For example, you can use three pieces of 1/3 banana each to create a banana pancake. **For the easy subproblem, ignore the pancakes marked by a star**.

Input specification

The first line of the input file contains an integer t specifying the number of test cases. Each test case is preceded by a blank line. Each test case consists of two lines.

The first line of a test case contains five integers c_m, y, s_{su}, s_{sa}, and f, meaning that you have c_m cups of milk, y egg yolks, s_{su} tablespoons of sugar, s_{sa} teaspoons of salt, and f cups of flour.

The second line contains four integers b, g_s, g_c, and w, meaning that you have b bananas, g_s grams of strawberry jam, g_c grams of chocolate spread, and w walnuts.

All quantities are between 0 and 10^6, inclusive.

Output specification

For each test case, output a single line containing a single integer – the maximum number of filled pancakes you can prepare.

Example for the easy subtask

input

```
2

16 16 8 2 17
10 47 100 19

16 16 8 2 17
10 470 100 19
```

output

```
16
30
```

In the first test case, we are limited by the number of fillings: we can make 10 banana pancakes, 1 strawberry pancake, 4 chocolate pancakes, and 1 walnut pancake. In the second test case, we only have enough flour for 30 pancakes.

If we solved the same input in the hard subtask, the first answer would be 17: First, make 3 grand mix pancakes. You are left with 9 bananas, 32 g of strawberry jam, 85 g of chocolate spread, and 10 walnuts. That is enough for another 9+1+3+1=14 easy pancakes.

Task authors

Problemsetter:	Michal 'mišof' Forišek
Task preparation:	Michal 'mišof' Forišek

Solution

Clearly, the problem can be divided into two separate parts: bake as many pancakes as you can, and prepare as many fillings as you can. First, we will consider the pancakes.

Given 8 cups of milk, 8 egg yolks, 4 tablespoons of sugar, 1 teaspoon of salt, and 9 cups of flour, we can prepare 16 pancakes. In other words, if we get c_m cups of milk, we can prepare at most $2c_m$ pancakes. The same goes for the other ingredients. Thus, we get the following formula:

$$\#\text{pancakes} = \min\left(2c_m,\ 2y,\ 4s_{su},\ 16s_{sa},\ \lfloor 16f/9 \rfloor \right)$$

In the easy subproblem, computing the maximum number of fillings is easy: we just have to consider each ingredient separately. For example, if we have g_s grams of strawberry jam, we can prepare $\lfloor g_s/30 \rfloor$ strawberry pancakes. This gives us the following formula:

$$\#\text{fillings\_easy} = b + \lfloor g_s/30 \rfloor + \lfloor g_c/25 \rfloor + \lfloor w/10 \rfloor$$

In the hard subproblem, we can also prepare mixed fillings. Luckily, it can be shown that we never need too many of these:

- For a banana pancake with chocolate, we need 3/4 of a banana and 10 g of chocolate spread.

  Consider 4 banana pancakes with chocolate. Together, these contain 3 bananas and 40 g of chocolate spread. And that is enough to make 4 easy fillings (and still have 15 g of chocolate spread left).

  We can now make an important conclusion: There is always an optimal solution in which we prepare less than 4 banana pancakes with chocolate.

- For a walnut pancake with chocolate, we need 7 walnuts and 10 g of chocolate spread.

  Again, 5 walnut pancakes with chocolate contain enough ingredients to make 3 walnut pancakes and 2 chocolate pancakes instead. Thus, there is always an optimal solution with less than 5 of these pancakes.

- Finally, 30 grand mix pancakes contain enough ingredients for 10 banana pancakes, 5 strawberry pancakes, 6 chocolate pancakes, and 9 walnut pancakes. Thus, there is always an optimal solution with less than 30 of these.

Clearly, the three conclusions can be combined. So it is enough to look for an optimal solution with less then 4, 5, and 30 of the mixed filling types. This can be done by brute force: try all valid cases, and for each of them use the formula for #fillings_easy to compute the number of additional simple fillings.

(Although it wasn't necessary, we can do even better. For example, can you prove that there is always an optimal solution that does not use banana pancakes with chocolate at all?)

It is also worth noting that somebody already did all the programming for us. Our pancake problem is just a special case of a linear optimization problem. The input can easily be turned into an integer linear program, and we can use a general ILP solver to find its optimal solution. Here is the integer linear program (in `lp_solve` syntax) that corresponds to the first example input:

```
filledpancakes; // this is the goal we are maximizing

// these are the actual constraints on the ingredients from the input
cm = 16; y = 16; ssu = 8; ssa = 2; f = 17;
b = 10; gs = 47; gc = 100; w = 19;

// everything below this point is constant -- all values are from the problem statement

filledpancakes <= pancakes;
filledpancakes <= fillings;

8*pancakes <= 16*cm;
8*pancakes <= 16*y;
4*pancakes <= 16*ssu;
1*pancakes <= 16*ssa;
9*pancakes <= 16*f;

fillings = f1 + f2 + f3 + f4 + f5 + f6 + f7;

// for each filling resource, the total amount we use <= the amount that is available
12*f1 + 9*f5 + 4*f7 <= 12*b;
30*f2 + 5*f7 <= gs;
25*f3 + 10*f5 + 10*f6 + 5*f7 <= gc;
10*f4 + 7*f6 + 3*f7 <= w;

// to solve the easy subproblem instead, just add the constraints f5=0; f6=0; f7=0;

int filledpancakes pancakes fillings f1 f2 f3 f4 f5 f6 f7;
```

Quality of Service

For technical reasons, this problem statement is only available online. Sorry for the inconvenience. The example input and output is available, though:

Example

input

```
3

1

2

47
```

output

```
2
47
42
```

Quality of Service – the actual problem statement

The online version of the problem statement greeted the solvers with the following message:

```
HTTP Error 503
Service Overloaded

Sending incomplete response. Please try again in a minute or two.

      v                                                A   a     v
         e             f  e                      e
                                        r                    n            e
                       d
                                 e            i              t   l

[many lines of scattered letters omitted from this booklet]
```

The scattered letters changed every 60 seconds. Here is the complete text you would get by observing the statement for a sufficient amount of time.

```
The above error message is, of course, completely fake. Actual server overload
issues have entirely different symptoms, as anyone who has been using the
internet long enough knows. Overloaded servers often drop connections, time out
or disconnect in the middle of a response. But this response is, in fact,
complete. It's just that the server was intentionally set up to blank out most
of the characters.

Thus, your first priority in this task is to figure out the actual problem
statement. The server periodically changes which characters get blanked out, so
the most straightforward way to decode the whole message is to do exactly as the
first paragraph says: just try again later -- and again, and again, and again,
until you collect enough characters to reconstruct the rest. This can be done
```

manually by refreshing the page in a browser (the most common key shortcut for that is F5), but it's simpler (and, according to our testers, more sanity preserving) to write a script that automates this task. Under UNIX, one could use wget or curl, but those aren't the only options by far.

Anyway, if you can read this, you must have successfully done just that! That's great! Allow me, the author of this task, to congratulate you on behalf of the whole IPSC organizing team. We're really proud of you and hope you win. Good luck in the actual contest tomorrow! (Or today, depending on when you read this.)

Now, to the task at hand. For the easy data set, the situation looks like this: As is often the case, the first line of the easy input file contains an integer T. T test cases follow, each preceded by a blank line. (That's a pretty common convention in IPSC.) However, in this particular task, the blank line doesn't have any real purpose -- every test case consists of a single integer. Let's call this integer A. (A is for "aardvark". I always liked that word.) For each test case, answer a single integer B. (B is for "buffalo". Buffalo buffalo Buffalo buffalo buffalo buffalo Buffalo buffalo. Get it?) It can be pretty much anything, but it can't be the same number as A. Aside from that single condition, we're pretty lenient. There is no single right answer, so don't hesitate to express your individuality! (There are wrong answers, though. Try to avoid them, okay?)

Oh, almost forgot! It is also an IPSC convention that problem statements should include a story. Everyone is surely eagerly awaiting the story, and it wouldn't do to let our fans down, right? Let me think of something... Alright, here goes: Alice and Bob are playing a game. Alice is thinking of a number A and Bob is trying to guess it. What Alice doesn't know is that Bob is psychic and already knows her number. He's bored of winning all the time, so he wants to lose on purpose by guessing a number different from A. However, he doesn't want to appear too incompetent, so it shouldn't differ from A by more than one hundred. Your answers must follow this rule, or Alice could get bored too, and that's never a good thing.

In the hard data set, there is an additional rule: the parity of the test case number and your answer must differ. In other words, in the first, third, fifth etc. test cases, your answer must be even, and in every second test case, your answer must be odd.

The output file should contain T numbers, one for each test case, separated by at least one whitespace. (Extra whitespace doesn't matter.) All numbers in the input file are integers between one and one billion, inclusive. The numbers in the output file must satisfy that constraint as well for your solution to be valid.

That's it. Any solution that satisfies the given conditions will be accepted. Easy as pie, right?

Task authors

Problemsetter:	Tomáš 'Tomi' Belan
Task preparation:	Tomáš 'Tomi' Belan

Solution

The actual problem is rather easy – the hard part was figuring out what you're supposed to do. The irony is that once you've reassembled the full problem statement, it helpfully tells you how to reassemble the full problem statement, and even recommends a few tools to help with that.

The problem statement page is almost blank, containing just a few scattered letters and an error message saying "please try again in a minute or two". If you do, you'll get a different copy, with letters in other places.

It turns out that while the positions of the visible characters are random, the content isn't. Every minute, we make a copy of the original problem statement and randomly replace most of the characters with whitespace. (This is hinted at by the fact that when a place is non-blank in two different copies, it is always the same letter.) Therefore, by downloading enough copies of the page and merging them together, you can recover enough of the problem statement to find out what to do.

Another hurdle is that the paragraph describing the hard data set has much worse character visibility. You can either wait until you have enough information, or you can try to infer it. The paragraph is pretty redundant, with a single rule repeated multiple times.

To summarize the task itself: in test case t, you must read a number A_t and output any number B_t such that the following conditions hold:

- $1 \leq B_t \leq 10^9$,
- $A_t \neq B_t$,
- $|A_t - B_t| \leq 100$,
- $t + B_t$ is odd (hard data set only).

(One final note: During the practice session we received five messages from different teams saying that the problem statement is broken.)

Rotate to divide

We have found an interesting paragraph in an article about physicist Freeman Dyson[1].

> A group of scientists will be sitting around the cafeteria, and one will idly wonder if there is an integer where, if you take its last digit and move it to the front, turning, say, 112 to 211, it's possible to exactly double the value. Dyson will immediately say, "Oh, that's not difficult," allow two short beats to pass and then add, "but of course the smallest such number is 18 digits long." When this happened one day at lunch, William Press remembers, "the table fell silent; nobody had the slightest idea how Freeman could have known such a fact or, even more terrifying, could have derived it in his head in about two seconds."

Problem specification

You will solve a slightly different problem. Instead of doubling the number, your goal is to make it k times **smaller**. We will also consider numbers in different bases. Your task is, given a base b and number k, to find the smallest number in base b which becomes k times smaller after a single rotation. Since the number can be very large, you only need to output its length and its first digit.

The rotation is done by taking the last digit and moving it to the front of the number. Rotation can only be applied to a number that has at least two digits. The last digit can also be a zero. For example, 110 rotates to 011, which is the same as 11.

Input specification

The first line of the input file contains an integer t specifying the number of test cases. Each test case is preceded by a blank line.

Each test case is a single line with two integers: the base $b \geq 2$ and the ratio $k \geq 1$.

In the easy data set we have $b, k \leq 8$, in the hard data set $b, k \leq 2,000,000$.

Output specification

Print one line for each test case. If there is no number in base b such that by rotating it we get a k times smaller number, output "`NO`". Otherwise, find the smallest such number, output its length and its first digit. The first digit should be output in base 10, as shown in the last example below.

Kindly note that the length will always be at least 2, as single-digit numbers cannot be rotated.

Example

input

```
3

5 2

3 2

16 7
```

output

```
2 3
NO
3 12
```

In the first test case, $(31)_5 = (16)_{10}$ *becomes* $(13)_5 = (8)_{10}$*. This is the smallest solution, so we output length 2 and first digit 3. In the last test case,* $(c71)_{16}$ *is rotated to* $(1c7)_{16}$*, so the length is 3 and the first digit of the original number has value 12.*

[1] `http://www.nytimes.com/2009/03/29/magazine/29Dyson-t.html?pagewanted=8`

Task authors

Problemsetter: Lukáš 'lukasP' Poláček
Task preparation: Lukáš 'lukasP' Poláček

Solution

To solve the small input you need to try numbers $b, b+1, b+2, \ldots$ and stop when you find a solution. The problem is the case when there is no solution. It's enough to iterate to b^b or even b^{b-k}, which you could either guess or prove. We will prove this fact in the solution for the hard input.

Note that for $k = b$ the solution is always $(10)_b$. For $k > b$ the solution obviously doesn't exist. For $k = 1$ the solution is always $(11)_b$. So from now on we will assume that $1 < k < b$.

Let us write the solution as $xb+y$, where x has l digits in base b and y is one digit, thus $b^{l-1} \leq x < b^l$ and $0 \leq y < b$. Then the rotation produces number $yb^l + x$. We know this is k times smaller, so we have

$$xb + y = k(yb^l + x).$$

Rewriting leads to

$$x = \frac{y(kb^l - 1)}{b - k}.$$

Since x is an integer, $b - k$ must divide $y(kb^l - 1)$. Furthermore, we get the smallest x by choosing the smallest y. It follows that the smallest y is

$$\frac{b-k}{\gcd(kb^l - 1, b - k)}$$

for fixed length l, where gcd stands for *greatest common divisor* function.

Our algorithm will try all lengths l starting from 1. The number $kb^l - 1$ can be big, but since we are only interested in its common divisor with $b-k$, we do not need to compute it. Denote $a_l = (kb^l - 1) \bmod (b-k)$. Then clearly $\gcd(kb^l - 1, b - k) = \gcd(a_l, b - k)$.

The sequence $(a_l)_{l=1}^{\infty}$ is obviously periodic and its period is at most $b - k - 1$. Hence there is no need to try $l \geq b - k$. This also proves the fact that the smallest solution, if it exists, does not exceed b^{b-k}.

After we calculate y, we need to check that x is within bounds $b^{l-1} \leq x < b^l$ and also find its first digit. As $k \geq 1$, we have $\frac{kb^l - 1}{b-k} \geq b^{l-1}$ which implies $x \geq b^{l-1}$. We will check the upper bound on x while calculating the first digit. If we get that the first digit of x is less than b, then x is less than b^l.

The first digit of x is

$$\left\lfloor \frac{x}{b^{l-1}} \right\rfloor = \left\lfloor \frac{y(kb^l - 1)}{b^{l-1}(b-k)} \right\rfloor.$$

From the previous discussion we have $0 \leq y \leq b - k$. If $y = b - k$, the first digit of x is $\lfloor x/b^{l-1} \rfloor = kb - 1 > b$, so this is never a solution. The case $\frac{y}{b-k} < 1$ remains. In this case, if we increase x by $\frac{y}{b-k}$, it will not affect its first digit. We get:

$$\left\lfloor \frac{x}{b^{l-1}} \right\rfloor = \left\lfloor \frac{x + \frac{y}{b-k}}{b^{l-1}} \right\rfloor = \left\lfloor \frac{ykb^l}{b^{l-1}(b-k)} \right\rfloor = \left\lfloor \frac{ykb}{b-k} \right\rfloor.$$

We now need to check whether this value is less than b. If yes, then we found a solution. If no, then no solution exists for this length: a larger y would just produce a larger first digit.

In total, the time complexity is $O((b-k)\log(b-k))$, since we need to run gcd $b - k$ times and that is the most costly operation.

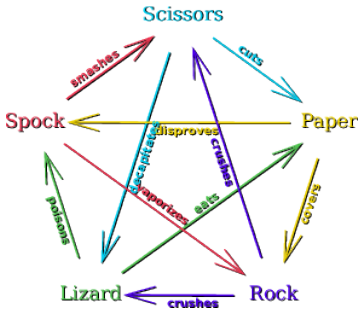
Against a rock play Spock

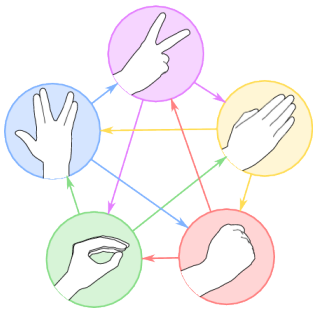
For many people, a Rock-Paper-Scissors game is too simple and thus boring. As IPSC should not be boring, we will be playing an extended version called Rock-Paper-Scissors-Lizard-Spock.

The game contains five gestures: *rock*, *paper*, *scissors*, *lizard*, and *Spock*. Lizard is usually formed by a hand closed to a sock-puppet-like mouth and Spock is formed by the Star Trek Vulcan salute. In each round of the game, both players simultaneously show one of the five gestures.

Each round is evaluated as follows: If the two gestures are equal, the round ends in a tie (nobody wins). Otherwise look at the table below and find the row that contains both gestures shown. The player showing the gesture in the left column wins. To the right of the table you can see figures from Wikipedia that contain the same information.

scissors	cut	paper
paper	covers	rock
rock	crushes	lizard
lizard	poisons	Spock
Spock	smashes	scissors
scissors	decapitate	lizard
lizard	eats	paper
paper	disproves	Spock
Spock	vaporizes	rock
rock	breaks	scissors





Problem specification

The input contains the sequence of gestures we are going to play in the following rounds. For each gesture, output a line containing the name of the gesture you are going to play against it.

To solve the **easy subproblem**, your task is to win every round of the game.

To solve the **hard subproblem**, your task is the same. But you have to do it without using the same gesture twice in a row.

For instance, suppose that in round 1 we played `Spock` and you played `paper`. If we play `rock` in round 2, you can only win by playing `Spock`, you may not play `paper` again. In round 3, you then cannot play `Spock`, but you may play `paper` again, if you wish.

Input specification

The first line contains a positive integer r: the number of rounds we will play.

Each of the next r lines contains one of the five strings `rock`, `paper`, `scissors`, `lizard`, and `Spock`.

Output specification

For each round, output a single line containing your gesture – again, one of the strings `rock`, `paper`, `scissors`, `lizard`, and `Spock`. (Note the uppercase '`S`' in Spock.)

Example

input

```
3
Spock
rock
rock
```

output

```
paper
Spock
paper
```

Task authors

Problemsetter:	Monika Steinová
Task preparation:	Monika Steinová

Solution

The task was meant to be the easiest one in this year's problem set.

Easy subproblem

There is not much to do for this subproblem. We just have to read the input and then output one of the two gestures that will win the round. For each input gesture, we can hard-wire a fixed output gesture. To do that, we may simply use the first five rows of the table in the problem statement.

Hard subproblem

First of all, note that a solution always exists. For each gesture we could play there are two others that beat it, and at most one of these is forbidden at any moment.

To improve the above solution, we have to store both winning answers for each gesture. To process a new gesture from the input, we compare the first winning answer to the gesture we used in the previous round. If they are equal, we use the second winning answer, otherwise we may use the first one.

BFS killer

Mark just solved a traditional programming task: finding the shortest path in a maze. The map of his maze is a rectangular grid with r rows and c columns ($1 \leq r, c \leq 600$). Some of the cells contain walls (denoted by `#`, ASCII code 35), others are empty (denoted by `.`, ASCII code 46). There are two special empty cells: one starting point (denoted by `S`) and one treasure (denoted by `T`).

In each step the player may only move in one of the four cardinal directions. In other words, each two subsequent cells on his path have to share a common side. The treasure is always reachable from the starting point. The player is not allowed to leave the maze or enter a cell with a wall.

Mark's task was to find the length of shortest path from the starting point to the treasure. He solved this task using breadth-first search (BFS). His C++ solution is given in the input file. Below you can find the solution in pseudocode. However, Mark made a common bug in his implementation of the FIFO (first in, first out) queue – he is using a circular array of size k. Show him that his queue is too short!

Problem specification

Find a map of a maze such that if Mark's program is executed on your maze, the queue size will exceed k at some moment. In the easy subproblem, $k = 5\,000$. In the hard subproblem, $k = 15\,000$. (The queue size is the number of cells it contains.) Your maze must satisfy all the constraints given above.

```
define process_cell(r,c):
    if (r,c) is not outside the maze:
        if cell at (r,c) is not a wall:
            if (r,c) is unvisited:
                if treasure is at (r,c), terminate the entire search
                put (r,c) into Q and mark (r,c) as visited

define BFS(startr,startc):
    initialize an empty queue Q
    put (startr,startc) into Q and mark (startr,startc) as visited
    while Q is not empty:
        get (r,c) from the queue
        for each (nr,nc) in (r,c+1), (r-1,c), (r,c-1), (r+1,c):
            process_cell(nr,nc)
```

Input specification

The input file for this problem contains a reference C++ implementation of Mark's solution. The variable `size` in this implementation must exceed k.

Output specification

On the first line output two integers separated by a single space: number of rows r and number of columns c. Then output r rows with c characters in each – the map of your maze.

Example

A syntactically valid submission with maximum queue size 4.

```
3 4
....
.S#.
..#T
```

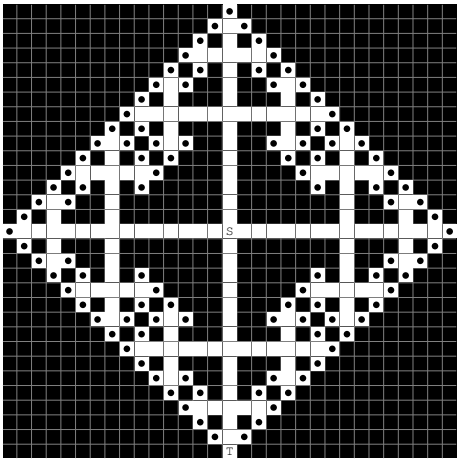
Task authors

Problemsetter:	Michal 'mišof' Forišek
Task preparation:	Peter 'Bob' Fulla

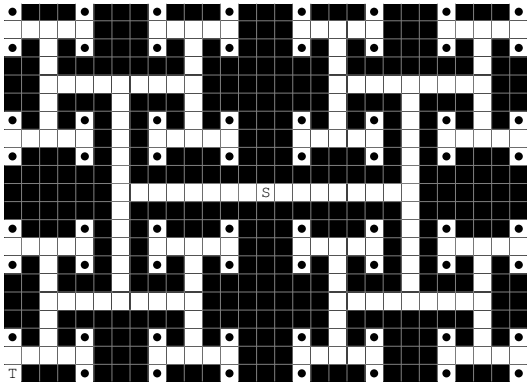
Solution

In order to force Mark's solution to store many items in the queue at the same time, we need to construct a maze in which many cells have the same distance from the starting point. A simple way how to obtain it is to think out an appropriate self-similar pattern (a fractal). These patterns tend to be very easy to implement using recursion. After we have generated the maze, we still need to place the treasure. The simplest and best choice for it is the cell visited by BFS as the last one.

A smaller version of one fractal that solves the easy subproblem is below; the cells with black dots all have the same distance from the starting point. (Full maze is 511×511, maximal queue size is 8 747.)



And here is a smaller version of a fractal which is sufficient to solve the hard subproblem. (Full maze is 381×509, maximal queue size is 16 383.)



(Do you still remember the solutions of IPSC 2010? The same tree fractal was used there in a completely different setting.)

Candy for each guess

Guess the number is a very simple game. Given is a positive integer n. The first player picks a number x from the set $\{0, 1, \ldots, n-1\}$, writes it on a card and places the card into an envelope. The second player then makes guesses. Each guess has the form of an integer g. The first player answers "too low" if $g < x$, "too high" if $g > x$, or "correct" if $g = x$. For each guess the first player gets a candy from the second player. The game ends when the second player gets the answer "correct".

Hannah discovered this game when she was eight and she started playing it with her younger brother Gunnar. Hannah always picked the number and Gunnar was always guessing.

When they were young, Hannah had it easy. Gunnar only knew a few deterministic guessing strategies, such as "binary search". In each game he would use one of the strategies he knew, picked uniformly at random. Hannah knew all Gunnar's strategies, and she always picked a number that maximized the expected number of candies she would get.

As Gunnar grew up, the games became much more involved. Eventually, both he and Hannah became perfect in playing the game.

Problem specification

In the easy input each test case contains n and a list of Gunnar's strategies. Find the expected number of candies Hannah will get per game if she plays optimally. Also, find one optimal strategy for Hannah.

In the hard input each test case contains n. Find the expected number of candies Hannah will get per game if both players play optimally (i.e., try to maximize/minimize the expected number of candies paid per game). Also, find one optimal strategy for Hannah and one optimal strategy for Gunnar.

Strategy format specification

We will only consider strategies where Gunnar never makes a guess that reveals no new information. (All strategies in the input for the easy data set will be such, and all strategies you will provide in the output for the hard data set must be such.)

Each such strategy can be encoded into a sequence of integers: First, you write down the first query he will make. Then you recursively write the strategy used if the guess was too large, and finally the strategy if the guess was too small. (I.e., the encoding is the pre-order traversal of the decision tree.)

For example, for $n = 6$ the strategy $(2, 0, 1, 4, 3, 5)$ means that in the first round Gunnar will guess 2. If the correct number is smaller, he will then guess 0 (and afterwards 1 if necessary), otherwise he will guess 4 (and then possibly 3 or 5).

Input specification – easy subproblem

The first line of the input file contains an integer t specifying the number of test cases. Each test case is preceded by a blank line.

Each test case consists of several lines. The first line contains the integer n (up to 16) and an integer s (up to 100) giving the number of strategies Gunnar knows. Each of the next s lines describes one strategy in the format specified above.

Output specification – easy subproblem

For each test case, output two lines. The first line must contain a fraction p/q giving the exact expected number of candies per game Hannah will gain by playing optimally. This fraction must be reduced (i.e., p and q must be coprime).

The second line must contain one optimal strategy for Hannah – a sequence of integers $h_0, \ldots, h_{n-1}$ giving the relative probabilities of Hannah's choices. That is, Hannah should choose the number i with probability $h_i/(\sum h_i)$. The numbers h_i must not exceed 10^9.

Example – easy subproblem

input

```
1

7 3
0 1 2 3 4 5 6
3 1 0 2 5 4 6
2 1 0 3 6 5 4
```

output

```
13/3
0 0 0 0 1 0 2
```

Gunnar picks one of three strategies: the first is a linear search, the second is a binary search, and the third is ad-hoc. One optimal strategy for Hannah is to pick the number 4 in 1/3 of all games and the number 6 in the remaining 2/3 of all games. This strategy guarantees her to get 13/3 of a candy per game, and there is no better strategy.

Input specification – hard subproblem

The first line of the input file contains an integer t specifying the number of test cases. Each test case is preceded by a blank line. Each test case consists of a single line with the integer n.

Output specification – hard subproblem

For each test case, output several lines. The first line must contain a fraction p/q giving the exact expected number of candies per game Hannah will gain if both players play optimally. This fraction must be reduced (i.e., p and q must be coprime).

The second line must contain one optimal strategy for Hannah – a sequence of integers $h_0, \ldots, h_{n-1}$ giving the relative probabilities of Hannah's choices. That is, Hannah should choose the number i with probability $h_i/(\sum h_i)$. The numbers h_i must not exceed 10^9.

The third line must contain a positive integer s not exceeding 1000 – the number of deterministic strategies Gunnar will be using with a non-zero probability. Each of the remaining s lines should contain $n+2$ space-separated tokens: the encoding of a strategy, a colon, and a positive integer g_i. The numbers g_i are relative probabilities of Gunnar's choices; they must not exceed 10^9.

Example – hard subproblem

input

```
1

2
```

output

```
3/2
1 1
2
0 1 : 1
1 0 : 1
```

The optimal strategies are "pick 0 or 1 with equal probability" and "start by guessing 0 or 1 with equal probability". In 50% of games Gunnar will succeed with his first guess, in the other 50% he will need a second guess.

Clearly, Hannah's strategy guarantees she can expect to gain at least 3/2 candies per game, and Gunnar's strategy guarantees she can expect at most 3/2. Hence both strategies are optimal.

Task authors

Problemsetter:	Michal 'mišof' Forišek
Task preparation:	Michal 'mišof' Forišek, Monika Steinová

Solution

The main punchline of this problem: If you are playing Guess the number against a very smart opponent, the optimal strategy is not the one you would expect. It is not "pick a number at random" if you are Hannah, and it is not "use binary search" if you are Gunnar.

When reading the solution, it may help to note how the two names were chosen: Hannah is the one who Hides the number, and Gunnar is the one who wants to Guess it.

Easy subproblem

First, imagine the case where Gunnar only uses one strategy.

For each of Hannah's choices we can easily determine the number of candies it will yield: It is the number of questions Gunnar will ask. If we draw Gunnar's strategy as a decision tree, then the number of candies Hannah gets if she picks x corresponds to the depth of the node containing x. (And this is certain, there is no randomness involved.)

What changes if Gunnar has more than one strategy?

Hannah picks some x, and at the same time Gunnar picks one of his strategies. Each of the strategies has some fixed payoff for Hannah, so the expected number of candies she will get for picking x can simply be computed as the average of payoffs x yields with each of Gunnar's strategies.

This gives us a really simple algorithm that solves the task:

- for each Gunnar's strategy, build the tree and compute the depth of each node
- for each x, compute the average depth of node x
- find any x for which this average is maximized.

Always playing that x is one of the optimal strategies, and its corresponding average is the expected number of candies Hannah will get per game. (The example was deliberately misleading, having Hannah choose between two strategies. This is never necessary.)

Hard subproblem

First of all, we need to convince ourselves that the problem is not trivial. To see that, let me tell you how it went with Hannah and Gunnar as they were getting older.

For a while, they enjoyed playing the game for $n = 3$. Hannah always picked her number uniformly at random, and Gunnar always used binary search, guessing 1 first and 0/2 next if necessary. Hannah was getting 5/3 candies per game, on average.

Then, one day, Hannah realized that she can do better. Why would she ever pick 1? From that moment on, she started only choosing between 0 and 2, and she was getting two candies per game.

Soon, Gunnar noticed she is not playing any 1s, so he stopped asking about those. Instead, he started using each of the strategies 0 2 1 and 2 0 1 with equal probability. This got Hannah down to 3/2 candies per game.

For a while they experimented with different strategies, until finally they discovered the optimal ones: Hannah would pick each of the numbers 0 and 2 with probability 2/5, and the number 1 with probability

1/5. Gunnar would use the binary search strategy with probability 3/5 and each of the two strategies 0 2 1 and 2 0 1 with probability 1/5.

What is the payoff for these strategies and why are they optimal?

First of all, look at Hannah's choices. This is exactly what we did in the easy subproblem – we know Gunnar's strategies and we are looking for the best strategy for Hannah.

Should Hannah pick 0, in 1/5 of all games she will get one candy, in 4/5 of all games she will get two, for an expected 9/5 of candy per game. The same is true if she picks 2. And if she picks 1, she will get one candy in 3/5 of all games and three candies in the remaining 2/5 of games, again yielding the expected payoff of 9/5 candies per game.

In other words, we now showed that Gunnar's strategy *guarantees* that Hannah will not get more than 9/5 candies per game on average, regardless of what she does.

We may now repeat the same analysis, but this time from Gunnar's point of view, knowing Hannah's probability distribution.

Gunnar has five possible strategies. For the binary search strategy 1 0 2 we get the following: In 1/5 of all games, Hannah picks 1 and gets one candy. In 4/5 of all games, she picks 0 or 2 and gets two candies. Hence if Gunnar picks this strategy, Hannah will get 9/5 candies per game on average.

Repeating the same analysis for the remaining strategies we get that if Gunnar plays 0 2 1 or 0 1 2, Hannah will also get an expected 9/5 candies per game. The remaining two strategies, 0 1 2 and 2 1 0, are worse, each of them giving Hannah an even two candies per game.

Again, this result can be seen in the following way: if Hannah sticks to her probability distribution, she can surely expect to get at least 9/5 candies per game, regardless of what strategies Gunnar uses.

And this shows that 9/5 is the answer for our problem if $n = 3$, and the presented two strategies are optimal – Hannah can make sure that the answer is at least 9/5 and Gunnar can make sure that the answer is at most 9/5.

In general, what we have is called a *zero-sum game*. (The meaning of this name is that one player gains what the other loses.) We can represent this game as a matrix. The rows will correspond to Hannah's n choices, the columns will correspond to Gunnar's guessing strategies. Each entry in the matrix is the number of candies Hannah gets if she and Gunnar pick the corresponding choices.

Each strategy of each player can be formalized as a probability distribution with which they pick the corresponding row or column. Hannah's (the row player's) goal is to maximize the expected value of the chosen element, Gunnar's goal is to minimize it.

It is well known that each zero-sum game has a unique *value* – a number v such that the row player has a strategy that makes sure the expected outcome is at least v and the column player has a strategy that makes sure it is at most v. (In the above example, $v = 9/5$.)

There are multiple ways how to compute the value of a zero-sum game and find the corresponding strategies. One of the most straigthforward ones is to reduce this problem to a linear program in the following way.

Let h_0 to h_{n-1} be Hannah's probability distribution. What can we tell about v, knowing these? We can evaluate all Gunnar's strategies and for each of them compute its expected cost. When this is done symbolically, with h_0 to h_{n-1} being variables, each of Gunnar's strategies gives us a linear inequality that is an upper bound on v. As Hannah tries to maximize v, she is looking for a probability distribution such that all of these constraints are satisfied and v is as large as possible.

For example, for $n = 3$ we get the following linear program:

maximize v

$0 \leq v, h_0, h_1, h_2$

$h_0 + h_1 + h_2 = 1$

$v \leq h_0 + 2h_1 + 3h_2$ (strategy 0 1 2)

$v \le h_0 + 3h_1 + 2h_2$ (strategy 0 2 1)
$v \le 2h_0 + h_1 + 2h_2$ (strategy 1 0 2)
$v \le 2h_0 + 3h_1 + h_2$ (strategy 2 0 1)
$v \le 3h_0 + 2h_1 + h_2$ (strategy 2 1 0)

Still, one problem remains: for larger n the number of Gunnar's strategies starts to be very large. In the rest of the solution we show one possible way how to make the linear program significantly smaller.

Lemma 1. In any optimal strategy for Hannah, all choices have positive probabilities.

Proof. Assume the contrary, suppose that Hannah never picks x. If there are multiple such xs, pick one such that at least one of $x-1$ and $x+1$ has a positive probability.

Clearly, there is an optimal strategy for Gunnar such that he never asks x. Formally, x is always a leaf in his decision tree and he never gets there. This x is then always lower than both $x-1$ and $x+1$. But then Hannah can strictly improve her expected payoff by moving a small non-zero probability from that number to x. This is a contradiction, as this is impossible if both her and Gunnar's strategy were already optimal. Hence the assumption was false.

Lemma 2. There is an optimal strategy for Hannah that is symmetric, i.e., for each i the probability of choosing i and $n-1-i$ is the same.

Proof. If two Hannah's strategies H_1 and H_2 are optimal, then their average – the strategy "use H_1 or H_2 with equal probability" – is optimal as well. If a strategy H is optimal, then its reverse H^R is optimal as well, because of symmetry. (H^R is a strategy where you pick $n-1-i$ if you would pick i using H.)

Pick any optimal strategy H. The average of H and H^R is an optimal strategy, and it is symmetric.

Now take a look at Gunnar's strategies. Once we only consider symmetric strategies for Hannah, some of Gunnar's strategies obviously become useless.

For example, let $n = 4$. Consider the two strategies 0 2 1 3 and 2 0 1 3. What can we tell about these two? See the following table:

	Gunnar plays 0 2 1 3	Gunnar plays 2 0 1 3
Hannah picks (0 or 3)	2 candies	2 candies
Hannah picks (1 or 2)	5/2 candies	2 candies

It can be seen that regardless of Hannah's choice the second strategy is always at least as good as the first one. (In game theory, we say that the second strategy *dominates* the first one.)

Obviously, whenever a strategy G_1 is dominated by a strategy G_2, Gunnar may assign probability zero to strategy G_1 and forget about this strategy.

Originally, there were no two of Gunnar's strategies such that one dominates the other. But once we only consider symmetric strategies for Hannah and merge her choices i and $n-1-i$ into one new choice with average payoff, we will get a lot of dominated strategies.

For $n = 16$ there are 35 357 670 strategies for Gunnar. However, after we only consider symmetric strategies for Hannah and prune dominated strategies for Gunnar, we are left with less than 100 000 of his strategies, and the resulting linear program can be solved quickly.

Also, while not necessary, it is reasonably easy to prune the search that generates Gunnar's strategies. For instance, once we consider symmetric strategies for Hannah, we only need to generate those strategies for Gunnar where the first guess is less than $\lceil n/2 \rceil$. If n is odd and the first guess is in the middle, we may consider only symmetric strategies for Gunnar.

There are also some more involved ways of pruning the strategies. For example, if the first guess is $i < \lceil n/2 \rceil$, in the branch where the answer is more than i the second guess must be at least $n-1-i$. Otherwise we could do these two guesses in opposite order and save some candies.

Divide the rectangle

Given is a rectangle consisting of $r \times c$ unit squares. One of these squares, in row r_r and column c_r, is colored red. A different square, in row r_b and column c_b, is colored blue.

Easy subproblem: Color each of the remaining squares red or blue in such a way that the red part and the blue part have the same shape. (That is, the set of blue squares can be obtained from the set of red squares by using shifts, rotations and reflections.)

Hard subproblem: Color each of the remaining squares red or blue in such a way that the red part has and the blue part have the same shape. Additionally, the red part (and therefore also the blue part) must be connected.

A set of squares S is connected if it is possible to travel between any two of them without leaving S, using horizontal and vertical steps only.

Input specification

The first line of the input file contains an integer t specifying the number of test cases. Each test case is preceded by a blank line.

Each test case consists of three lines. The first line contains the dimensions r and c. The second line contains the coordinates r_r and c_r. The third line contains the coordinates r_b and c_b. You may assume that $1 \leq r_r, r_b \leq r \leq 100$, $1 \leq c_r, c_b \leq c \leq 100$, and $(r_r, c_r) \neq (r_b, c_b)$.

Output specification

For each test case, output r lines containing c characters each, each of the characters being `R` (red) or `B` (blue). If there are multiple solutions, pick any of them. If there is no solution, output one line with the string "`IMPOSSIBLE`". You may output empty lines between test cases.

Example

input

```
3

5 5
1 1
3 4

4 6
1 1
4 6

4 6
1 1
1 2
```

output

```
IMPOSSIBLE

RRRRRR
RRRRRB
RBBBBB
BBBBBB

RBBBBB
RBRRRB
RBBBRB
RRRRRB
```

This output would be correct in both subtasks.

Task authors

Problemsetter: Michal 'mišof' Forišek
Task preparation: Michal 'mišof' Forišek, Vladimír 'usamec' Boža

Solution

Easy subproblem

Clearly, the number of red squares must be equal to the number of blue squares. Hence for a solution to exist the area rc of the rectangle must be even. On the other hand, whenever the area is even, a solution exists. To construct one, we can use many different strategies. Here is one with almost no special cases: Let S be the point at the center of the rectangle. We will pair each square with the one symmetric to it according to S. That is, each square (x, y) will be paired with $(r + 1 - x, c + 1 - y)$. In each pair, we will color one square red and the other one blue. This will ensure that the blue part can be obtained by rotating the red part 180 degrees around S.

The only part missing is that the two squares from the input may now have wrong colors. To fix that, we first recolor the pair containing (r_r, c_r) properly, and then recolor the pair containing (r_b, c_b) properly, and we are done. (Notice that this works even if the given red and blue square form a pair.)

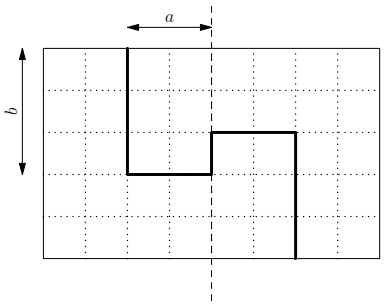
Hard subproblem

The situation is pretty similar in the hard subproblem: Almost whenever the area is even, a solution exists, and we can find one where rotation around S maps red squares to blue ones.

The only special case here are inputs where one of the dimensions is 1 and the other is even. For example, the input that corresponds to "`.R.B....`" does not have a solution: the only two possible solutions are "`RRRRBBBB`" and its reversal, and neither of them works. Luckily, this special case is easy to spot and easy to handle.

For the general case, probably the simplest approach is to implement some simple ways how the rectangle can be divided into two equal parts and then to try this approach on our input file. In some cases it will succeed, in some it will fail. Take one case where it fails, find a new pattern that works for that test case, and repeat the process until you solve all the cases with even area.

The following is the pattern we used in our implementation. First, WLOG assume that c is even (otherwise we swap rows and columns, solve with even c and then transpose the result). Consider the set of patterns shown in the figure on the right.



The pattern is determined by the values a and b ($0 \le a \le c/2$, $0 \le b < r$). It is very easy to construct: first, fill the entire red half with `R`s and the right half with `B`s. Then, change the appropriate $a \times b$ rectangle of `R`s into `B`s and vice versa.

For each solvable input there is at least one pair a, b such that this pattern works. (Of course, we may need to swap `R`s and `B`s and/or flip the pattern according to a horizontal axis.)

Why? If the two given cells are on opposite sides of the dashed axis, $a = 0$ works. If they are on the same side and in the same column, take $a = c/2$ and pick any b between them. And if they are not in the same column, pick a between them and take a sufficiently large b. (Of course, as the constraints were pretty small, the most convenient implementation just tries all combinations of a and b until it hits one that works.)

Elementary math

Nowadays, people use calculators and computers for almost everything. However, we are now working on the Intercontinental Plan for Sabotage of Calculators and soon all calculators will be destroyed. You better dust off you old skills and get a pencil and some paper.

Problem specification

Your task will be to compute the square root of a given number by hand (or if you insist, with help of your computer). In case you have never learned how to do this essential mathematical operation, we provide detailed instructions:

At the beginning, separate the number into pairs of digits, starting at decimal point. In case there is odd number of digits after the decimal point, add single zero digit at the end of the number. In case there is odd number of digits before the decimal point, the first "pair" will consist of only one digit. For example, if the number is 12345.678, we will split the number into pairs "1", "23", "45", "67" and "80". The square-root algorithm will compute the result in several iterations, each iteration taking into account one new pair of digits and appending one digit to the result.

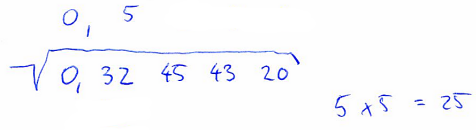
In the first iteration you should estimate the square root of the first pair of digits. That is, you should find the largest integer x such that $x^2 \leq first\_pair$. The value of x will be the first digit of the result. Now you should compute value of x^2 and subtract it from the value of the first pair.

Each of the remaining iterations can be computed as follows: Drop the next pair of digits next to the result of the previous subtraction. Denote the resulting number as t. You need to estimate next digit of the result. For this write "$y\_ \times \_ =$" to some temporary place, where y is twice the value of the result computed so far. Now replace both "_"s with the same digit i such that the result will be less or equal than t and i will be as large as possible. The value of i will be the next digit of our result. After i is found, the result of "$yi \times i$" should be subtracted from the value of t. This step ends the current iteration.

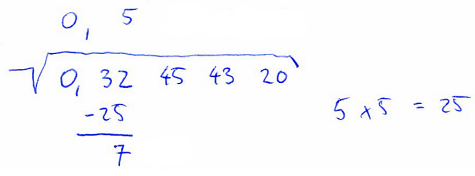
You may see the whole computation in little steps on the following pictures:

1. create pairs

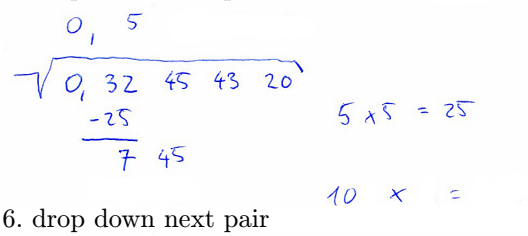
2. estimate first digit



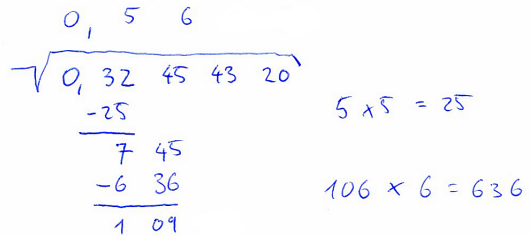
3. subtract



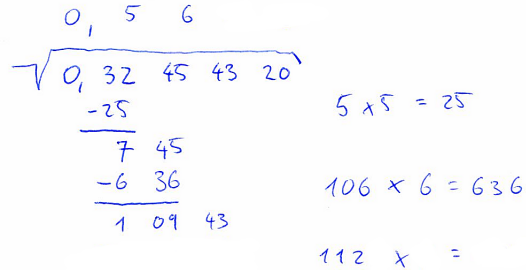
4. drop down next pair



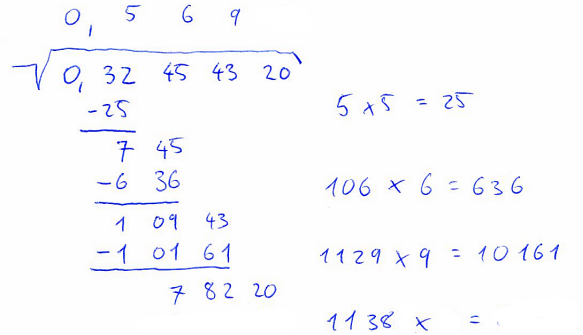
5. estimate next digit



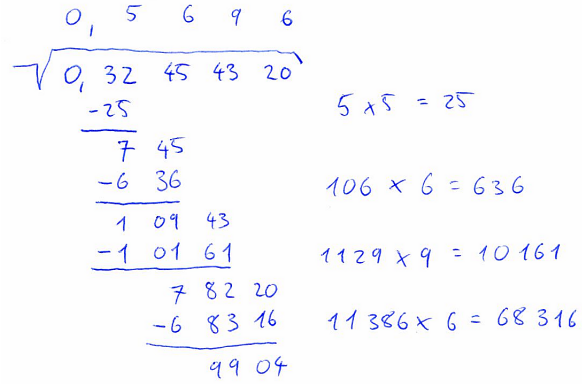
6. drop down next pair



7. estimate next digit



8. final result



Input specification

The first line of the input file contains an integer t specifying the number of test cases. Each test case is preceded by a blank line.

Each test case consists of a single line with a single real number containing up to 16 digits. You may assume that the number does not contain unnecessary leading zeroes. (The "0" in "0.12" is considered neccessary and it will always be present if such a number occurs.)

Output specification

Output will consist of outputs of individual test cases. Put a blank line between consecutive test cases.

The output of one test case is a character matrix consisting of r rows and c columns. The numbers r and c should be as small as possible. Blank spaces in the matrix should contain the character "." (dot).

Basically, the output matrix should look like the result of the handwritten computation shown in the problem statement. It should consist of the following parts: the input value, a square root sign, the result of calculation, and the intermediate calculations needed to obtain the result.

The square root sign should consist of character "_" (underscore), then characters "\/" (backslash and forward slash) on the line below, then as many dashes as necessary "-" and the root sign should be finalized by a backtick "`" (ASCII 96) on the line below. The space under the dashes should be filled with the input value. The input value should be written as two-digit pairs separated with one blank character. (Remember that the first pair may consist of only one digit.) There should also be exactly one blank space on the left and one on the right of the formatted input value.

If the input value contains a decimal sign (a dot in the input file), it must also be present in the output. First, format the digit pairs and the square root sign as described above. Then, replace the appropriate blank space by a comma "," character.

The result should be on the line above the square root sign and each digit of the result should be right-aligned with the corresponding input pair. In case the input value contains a decimal sign, the result must contain a comma in the same column as the input value does.

The intermediate calculations may be divided into two groups: subtractions and multiplications. Each multiplication should be in format "`12.x.3.=.456`" and should be left-aligned on the first position after the end of the square root sign. Each subtraction consists of the result of previous subtraction, the new dropped pair, the subtracted value, an underline, and the result of subtraction. The dropped pair should be aligned with its occurrence in the input. The subtracted value should consist of the character "-" (minus/dash) immediately followed by the value. The value should be split into pairs of digits and it should be right-aligned with the previous row. The underline should consist of several dashes, right aligned with the current computation. The underline must be as short as possible, but long enough so

that both above operands (including the minus sign, but excluding any leading zeroes the operands may have) will be completely above the underline. Finally, the result of subtraction should be right-aligned with the whole computation and should be split into pairs of digits separated by blanks.

Because computing square root of leading zeroes is useless (square root of zero is zero), the computation shown in your output should start with the first pair of digits that is not all zeroes. Note hovewer, that the input and the result must contain all the necessary zeroes. In particular this means that for the input "0" you should only output the square root sign with a zero below and a zero above it.

Example

input

```
3

17

3.14159

0.005
```

output

```
.....4.....
_..----.....
.\/.17.'....
...-16..4.x.4.=.16
...---......
.....1......

....1,.7..7..2.............
_..-----------.............
.\/.3,14.15.90.'...........
...-1..........1.x.1.=.1.....
...--.........................
....2.14.....................
...-1.89........27.x.7.=.189...
...-----.....................
......25.15..................
.....-24.29.....347.x.7.=.2429.
.....------..................
.........86.90...............
.........-70.84..3542.x.2.=.7084
........------...............
.........16.06...............

....0,.0..7.....
_..---------.....
.\/.0,00.50.'....
.........-49..7.x.7.=.49
........---......
..........1.....
```

Task authors

Problemsetter:	Peter 'PPershing' Perešíni
Task preparation:	Peter 'PPershing' Perešíni

Solution

This task was meant to be the "debug nightmare". Even though we are writing this before the contest, we are sure this task will meet our expectations.

Easy subproblem

The easy subproblem consists only of 10 values, most of them small. Thus, the best way to solve this subproblem was to do it by hand. You will get the hang of the square root algorithm, and you will also get several valuable test cases for your implementation. You will really need them when solving the hard subproblem.

Hard subproblem

We will not describe the algorithm solving the task (because the algorithm is pretty clear, the only problem is to get the formatting right). Instead, we will provide some clues where the errors in the implementations may be.

First of all, re-read the sentence "If the input value contains a decimal sign, it must also be present in the output." This sentence implies that the outputs for "1" and "1." should differ – precisely in that the second one must contain two commas.

```
    1
_  ---
 \/ 1 '
   -1  1 x 1 = 1
   --
    0

    1,
_  ---
 \/ 1,'
   -1  1 x 1 = 1
   --
    0
```

Another cause of errors lies in the statement "The subtracted value should consist of the character "-" (minus/dash) immediately followed by the value." Depending on the implementation of your function that writes a right-aligned number, you may accidentally omit the minus sign when outputting the value "-0".

Probably the nastiest thing is the statement "The underline must be as short as possible, but long enough so that that both above operands (including the minus sign, but excluding any *l*eading zeroes the operands may have) ..." What? Leading zeroes? How can the operands have leading zeroes? Well, you may get a zero from a subtraction and then add the next pair of digits. That is the zero that should be ignored when computing the length of the underline.

Another peculiar case is when the underline in the next iteration needs to extend even more to the left than in the previous one. Both the case with the leading zero and the case with a longer underline are shown in the following example:

```
    2  0, 4  0  3
_  ---------------
 \/ 4 16,29 68 28 ‘
   -4               2 x 2 = 4
   --
    0 16
      -0            40 x 0 = 0
      --
      16 29
     -16 16         404 x 4 = 1616
     ------
          13 68
             -0     4080 x 0 = 0
          -----
          13 68 28
         -12 24 09  40803 x 3 = 122409
         ---------
           1 44 19
```

The beginning of the underline may of course skip more than 4 characters to the right. Or for that matter, the biggest temporary calculation does not need to be the last one.

```
     6  8  9  0
_  -------------
 \/ 47 47 47 47 ‘
   -36           6 x 6 = 36
   ---
    11 47
   -10 24        128 x 8 = 1024
   ------
     1 23 47
    -1 23 21     1369 x 9 = 12321
    --------
          26 47
             -0  13780 x 0 = 0
          -----
          26 47
```

Flipping coins

A group of n friends (numbered 1 to n) plays the following game: Each of them takes a fair coin, flips it (getting a head or a tail with equal probability), and without looking at it glues it to her forehead. Then everyone is allowed to look at all other players, but all communication is forbidden during this phase. Finally, each of the players takes a piece of paper and writes down one of "head", "tail", and "pass".

A player is *correct* if she writes down the coin side that is visible on her forehead. A player is *wrong* if she writes down the opposite side. A player that writes "pass" is neither right nor wrong. The players win the game if *at least one of them* is right and at the same time *none of them* is wrong.

Problem specification

One possible strategy is that one of the players guesses "head" and all others pass. With this strategy the players win in exactly 50% of all possible cases. Is there a better strategy for $n = 3$? If yes, find one.

Then, as the harder part of this problem, find optimal strategies for all n from 1 to 8, inclusive.

Strategy format specification

It can be proved that even if we allowed randomized strategies for all players, there would be an optimal deterministic strategy. Hence we will only consider deterministic strategies.

From player i's point of view, there are 2^{n-1} different outcomes. Each outcome can be described by a string of $n-1$ `H`s (heads) and `T`s (tails) – for each player from 1 to n except for i, write her coin side. To write down the strategy for player i, order all outcomes alphabetically, and for each of them write down `H`, `T`, or `P` (pass) – the action player i should take when she sees the corresponding outcome.

The strategy for the entire game consists of n lines, with line i containing the strategy for player i.

Input specification

There is no input.

Output specification

For the easy subproblem, find and submit any strategy for $n = 3$ for which the probability of winning is more than 50%. If there is no such strategy, submit any strategy for which the probability of winning is exactly 50%.

For the hard subproblem, find and submit optimal strategies for all n from 1 to 8, inclusive, in order. If there are multiple optimal strategies for a given n, pick any of them. (That is, the output for this data set should start with one line of length 1, two lines of length 2 each, three lines of length 4 each, etc.)

Example

Example output for the easy subproblem

```
HHHH
PTPP
PPPP
```

This is a syntactically correct strategy for $n = 3$. In this strategy, player 1 always guesses "head", player 3 always passes, and player 2 passes except for one case – when she sees a head on player 1 and a tail on player 3, she guesses "tail".

Task authors

Problemsetter:	Michal 'mišof' Forišek
Task preparation:	Michal 'mišof' Forišek

Solution

This problem is based on a hat puzzle that is attributed to Todd Ebert.

Easy subproblem

Before we present the general solution, we will show a solution for $n = 3$. The strategy shown in the problem statement was not optimal. Here is one better strategy:

Each player will do the following: If you see a head and a tail, pass. Otherwise, say the opposite side to the ones you see. (If you see two heads, say tail, and vice versa.)

With this strategy their winning probability is 3/4. Why is it so? With probability 1/4, all players will throw the same side (all heads or all tails). Then all of them take a guess, all of them are wrong, and they lose. In all remaining cases, the players win. For example, consider the case where players 1 and 2 throw heads and player 3 throws a tail. In this case, players 1 and 2 both pass, and player 3 guesses correctly.

Note the main reason why this solution is better than the naive one: *All of the players* make wrong guesses in each configuration where they lose and only *one of them* makes a correct guess when they win.

(Note that if we take a sum over all configurations, the total numbers of correct and wrong guesses must always be equal. Hence this is the best we can get, and this strategy is optimal for $n = 3$.)

Hard subproblem

How to solve the game optimally for a general n?

We can imagine all possible results of the n coin throws as vertices of a n-dimensional hypercube. Now each edge of the hypercube corresponds to a player in one possible situation. The number of the player is the direction of the edge, and the coins she sees are the values of the other $n - 1$ coordinates.

For example, if $n = 3$, 0 is a head and 1 is a tail, the vertex 001 is the situation "players 1 and 2 threw heads, player 3 threw a tail", and the edge connecting 001 and 011 is the situation "player 2 sees that player 1 has a head and player 3 has a tail".

If in some situation some player decides *not* to pass, she will be correct in one of the two cases and wrong in the other one. By picking her answer, she can decide which case will be which. We can imagine this as placing one blue and one red token on the vertices connected by the corresponding edge. (Blue represents a correct answer, red an incorrect one.)

What we are trying to do is to pick some edges and distribute the red and blue tokens in such a way that the number of good vertices (with some blue but no red tokens) is maximized. In other words, we have to minimize the number of the remaining, bad vertices. To do so, we need one simple observation: every good vertex must be adjacent to some vertex that contains a red token. This means that the bad vertices always form a *dominating set* on the hypercube.

And there is an equivalence. Any dominating set defines a strategy for all the players: For each edge, if one of its vertices is in the dominating set and one is not, the corresponding player in the corresponding situation shall take a guess corresponding to the vertex that is not in the dominating set. In all other situations the players should pass. Clearly, this strategy guarantees that for all vertices *not* in the dominating set the players will win the game.

Hence to find the optimal strategy, we need to find one *smallest dominating set* on a n-dimensional hypercube for n between 1 and 8, inclusive.

In coding theory, the same thing is known as a *smallest binary covering code*.

Computing these is a hard problem. In 2005 it was proved that the smallest dominating set on a 9-dimensional hypercube has size 62. The exact answer for $n = 10$ is not known yet, it is somewhere in the range 107 to 120.

It is worth noting that some special cases are easy. For example, if n has the form $2^k - 1$, one smallest dominating set corresponds to a perfect Hamming error-correcting code that can correct a single error. Hence for each of these n there is a perfect strategy like the one we saw for $n = 3$: one where always either one player is correct, or all of them are wrong.

The known values of smallest dominating set sizes can be found, for example, here: `http://www.sztaki.hu/~keri/codes/2_tables.pdf`. For $n = 7$ and $n = 8$ we can easily construct these from the Hamming code for $n = 7$, and for smaller n we can use brute force.

Grid surveillance

Big Brother needs a new machine that will keep track of all the people moving into town.

Problem specification

The town is a grid $G[0..4095][0..4095]$. Initially, all entries in G are zeroes. The machine will need to support two types of instructions: additions and queries. Each addition modifies a single cell of the grid. Each query asks you about the sum of a rectangle. However, queries can also ask about older states of the grid, not only about the present one.

In order to have a smaller input file (and also in order to force you to process the instructions in the given order) we used the input format given below. There is a helper variable c, initially set to zero. Let $\varphi(c, x) = (x \text{ xor } c) \bmod 4096$. The instructions are processed as follows:

- Each addition is described by three integers x, y, a. To process it, first compute the new coordinates $x' = \varphi(c, x)$ and $y' = \varphi(c, y)$. Then, add a to the cell $G[x'][y']$. Finally, set c to the current value of $G[x'][y']$.
- Each query is described by five integers x_1, x_2, y_1, y_2, t. To process it, first compute the new coordinates $x'_i = \varphi(c, x_i)$ and $y'_i = \varphi(c, y_i)$. If necessary, swap them so that $x'_1 \le x'_2$ and $y'_1 \le y'_2$.

  Next, take the grid G_t that is defined as follows: If $t = 0$, then $G_t = G$. If $t > 0$, G_t is the state of G after the very first t additions. (If t exceeds the total number of additions so far, $G_t = G$.) Finally, if $t < 0$, then G_t is the state of G before the very last $-t$ additions. (If $-t$ exceeds the total number of additions so far, G_t is the initial state.)

  The answer to the query is the sum of all $G_t[i][j]$ where $x'_1 \le i \le x'_2$ and $y'_1 \le j \le y'_2$. This sum is also stored in c.

Input specification

The first line of the input contains an integer r ($1 \le r \le 100$). The second line contains an integer q ($1 \le q \le 20,000$). Each of the next q lines contains one instruction. The sequence of instructions you should process is obtained by concatenating r copies of these q instructions.

Each instruction starts with its type z. If $z = 1$, we are doing an addition, so three integers x, y, a follow ($0 \le x, y < 4096$, $0 \le a \le 100$). If $z = 2$, we are doing a query, so five integers x_1, x_2, y_1, y_2, t follow ($0 \le x_1, x_2, y_1, y_2 < 4096$, $|t| \le 500,000$).

Output specification

For each query, output a single line with the appropriate answer.
In the hard data set, only submit the answers to the last **20,000 queries**.

Example

input

```
2 5
1 2 2 3
1 2 4 1
2 1 5 1 5 -1
2 1 5 1 5 0
2 1 5 1 5 2
```

output

```
3
3
3
0
6
0
```

Task authors

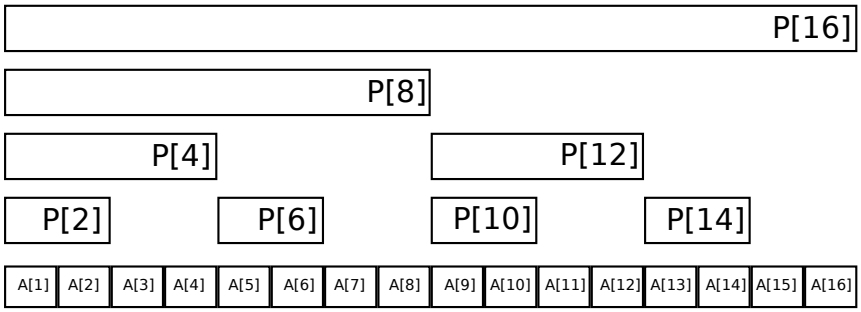
Problemsetter:	Vladimír 'usamec' Boža
Task preparation:	Vladimír 'usamec' Boža

Solution

This is a typical task where all we need is the right data structure.

First of all, we shall describe a data structure that could be used if all queries had $t = 0$, i.e., if we only ask about the current state of the grid. This structure is called a Fenwick tree and is quite well known in programming contests.

First, we shall describe a $1D$ variant of this structure. We have an array A of numbers (indexed from 1 to n) and we want to make the following queries: 1.) add a number to some element and 2.) get the sum of range $\langle 1, x \rangle$ for any x – we denote this by $S(x)$. We make another array P defined as follows. In $P[i]$, where $i = 2^a b$ (a, b are integers and b is odd), we store $P[i] = \sum_{x=i-2^a+1}^{i} A[x]$.



Obr. 1: Fenwick tree for 16 values

Now, we can compute $S(x)$ with the following recursive function: (`last_one(i)` denotes the last bit in the binary notation of i set to 1: for $i = 2^a b$, where b is odd, it returns 2^a.)

```
int get(int x) {
    if (x == 0) return 0;
    else        return P[x] + get(x-last_one(x));
}
```

Or we can write this nonrecursively:

```
int get(int x) {
    int ret = 0;
    for (int i = x; i > 0; i -= last_one(i)) ret += P[i];
    return ret;
}
```

To increment $A[x]$ by c, and update P accordingly, we can use the following procedure:

```
void add(int x,int c) {
    for (int i = x; i <= N; i += last_one(i)) P[i] += c;
}
```

Note that this procedure just updates every sum interval which overlaps the position x.

Also note that each call to this procedure costs us $O(\lg n)$ time.

We now need to make a $2D$ variant of this structure. We'll denote one corner as $(1, 1)$ and the opposite corner as (n, n). We then make another grid P for storing similar sums. In $P[i][j]$, where $i = 2^a b$ (a, b are integers and b is odd), and $j = 2^c d$ (c, d are integers and d is odd), we store $P[i][j] = \sum_{x=i-2^a+1}^{i} \sum_{y=j-2^c+1}^{j} G[x][y]$.

Getting the value of $S(i, j)$ (the sum of rectangle $(1, 1)$ to (i, j), inclusive) is done similarly to the $1D$ variant:

```
int get(int x, int y) {
    int ret = 0;
    for (int i = x; i > 0; i -= last_one(i))
        for (int j = y; j > 0; j -= last_one(j))
            ret += P[i][j];
    return ret;
}
```

And for adding c to $G[x][y]$ and updating P accordingly, we can use the following procedure:

```
void add(int x, int y, int c) {
    for (int i = x; i <= N; i += last_one(i))
        for (int j = y; j <= N; j += last_one(j))
            P[i][j] += c;
}
```

Using the above, the sum of a rectangle with corners (l, d), (r, u) can be obtained as $S(r, u) + S(l - 1, d - 1) - S(r, d - 1) - S(l - 1, u)$.

All of these queries run in $O(\lg^2 n)$ time.

Now we need to make our data structure persistent, so that we can get any value from the past as well. We should note that each addition only changes a few entries of P. For each entry in P, we will store the complete history of its changes. That is, the elements of P won't be numbers, but arrays of pairs. Each pair will represent an update as the timestamp and the new value. To query this array for a particular timestamp, we can use simple binary search.

This aproach leads to $O(\lg^2 n)$ time for addition and $O(\lg^3 n)$ time for each query.

HQ0-9+−INCOMPUTABLE?!

HQ9+ is an esoteric programming language specialized for certain tasks. For example, printing "Hello, world!" or writing a quine (a program that prints itself) couldn't be any simpler. Unfortunately, HQ9+ doesn't do very well in most other situations. This is why we have created our own variant of the language, HQ0-9+−INCOMPUTABLE?!.

A HQ0-9+−INCOMPUTABLE?! program is a sequence of commands, written on one line without any whitespace (except for the trailing newline). The program can store data in two memory areas: the *buffer*, a string of characters, and the *accumulator*, an integer variable. Initially, the buffer is empty and the accumulator is set to 0. The value of the buffer after executing all the commands becomes the program's output.

HQ0-9+−INCOMPUTABLE?! supports the following commands:

command	description
`h`, `H`	appends `helloworld` to the buffer
`q`, `Q`	appends the program source code to the buffer (not including the trailing newline)
`0-9`	replaces the buffer with n copies of its old value – for example, '2' doubles the buffer
`+`	increments the accumulator
`-`	decrements the accumulator
`i`, `I`	increments the ASCII value of every character in the buffer
`n`, `N`	applies ROT13 to the letters and numbers in the buffer (for letters ROT13 preserves case; for digits we define ROT13(d) = (d + 13) mod 10)
`c`, `C`	swaps the case of every letter in the buffer; doesn't change other characters
`o`, `O`	removes all characters from the buffer such that their index, counted from the end, is a prime or a power of two (or both); the last character has index 1 (which is a power of 2)
`m`, `M`	sets the accumulator to the current buffer length
`p`, `P`	removes all characters from the buffer such that their index is a prime or a power of two (or both); the first character has index 1 (which is a power of 2)
`u`, `U`	converts the buffer to uppercase
`t`, `T`	sorts the characters in the buffer by their ASCII values
`a`, `A`	replaces every character in the buffer with its ASCII value in decimal (1–3 digits)
`b`, `B`	replaces every character in the buffer with its ASCII value in binary (exactly eight '0'/'1' characters)
`l`, `L`	converts the buffer to lowercase

`e`, `E`	translates every character in the buffer to l33t using the following table:
	`ABCDEFGHIJKLMNOPQRSTUVWXYZ abcdefghijklmnopqrstuvwxyz 0123456789`
	`48(03=6#|JXLM~09Q257UVW%Y2 a6<d3f9hijk1m^0p9r57uvw*y2 O!ZEA$G/B9`

`?`	removes 47 characters from the end of the buffer (or everything if it is too short)
`!`	removes 47 characters from the beginning of the buffer (or everything if it is too short)

Problem specification

As you can see, HQ0-9+−INCOMPUTABLE?! is much more powerful than HQ9+. Demonstrate this by writing and submitting a HQ0-9+−INCOMPUTABLE?! program that outputs your TIS.

- For the easy subproblem, any valid program will do.
- For the hard subproblem, you must use the command "`+`" at least once.

Limits

Your program must be at most 10 000 commands long. The accumulator is unbounded (it can store an arbitrarily large integer). After each command, the buffer must be at most 10 000 characters long. To prevent code injection vulnerabilities, during the execution of your program *the buffer must never contain non-alphanumeric characters*, i.e. characters other than `A-Z`, `a-z`, and `0-9`. Should this happen, the program fails with a runtime error, and your submission will be rejected.

Examples

program	output
`h5!`	`rld`
`QCq`	`qcQQCq`
`q23`	`q23q23q23q23q23q23`
`h?h`	`helloworld`
`H2O`	`hlwolheo`
`h4op`	`ollwldwlhe`
`hint`	`ccfkrsvzzz`
`q18N`	`d41Ad41Ad41Ad41Ad41Ad41Ad41Ad41A`
`3QAh`	`518165104helloworld`
`Qb`	`0101000101100010`
`opaque`	`094QU3`
`h1Qt`	`1Qdehhllloortw`
`H9999`	(error: buffer size exceeded 10 000)
`quine`	(error: buffer contains "`|`")
`LMAO`	(empty output)

Task authors

Problemsetter:	Tomáš 'Tomi' Belan, Peter 'Bob' Fulla
Task preparation:	Tomáš 'Tomi' Belan, Peter 'Bob' Fulla

Solution

First of all, note that just like in the original HQ9+ language, the value of the accumulator can't be read by any command, we can only modify it. This means the accumulator is useless for our purposes and we can ignore it completely.

How to get a TIS, for example 12345678901234567890, into the buffer? One way is to use the `Q` (quine) command. The program could be based on this: `1234567890123456789 0Q`. This program first replaces the buffer with multiple copies of itself, but since the buffer is empty, nothing actually changes. Then the buffer becomes our TIS followed by the letter `Q`. Getting rid of the last character isn't trivial – we could use the command `O`, but it erases too much. Hence we first interleave the TIS with a harmless command, such as `m`, placing it on the positions that get erased. The resulting program will look like this: `1234mm5m67890m123m4mm56m7m89mm0mmmQO`.

To solve the hard subproblem, our program must contain the `+` command. This makes it impossible to use `Q` – after its execution, the buffer would contain `+`, which is a non-alphanumeric character. Therefore, we use a different approach: we build up a string of letters such that the `E` (leet) command will transform it to our TIS. Note that there may be multiple such strings – for example, `P`, `g` and `q` all get changed to `9`. However, as we'll show below, not all of these strings can be easily created.

We'll build up the string we need letter by letter. The `H` command gives us 10 letters, but we only want one. So we call `H` 33 times and then `?` 7 times. This command sequence first adds 330 characters to the buffer and then removes $7 \times 47 = 329$ of them, leaving a single letter `h`.

We need to append letters other than `h`, though. This can be done with a little trick. We can use a macro called ROT1, which rotates the characters by one instead of thirteen (so `a` becomes `b`, `b` becomes `c`, ..., `z` becomes `a`). If the buffer doesn't contain `z`, ROT1 can be easily implemented with the `I` command. If the buffer does contain `z`, we can't use `I`, because `z` would change into {, which is a non-alphanumeric character. But if the buffer doesn't contain `m`, the sequence `NIN` works. (The first `N` ensures that the buffer doesn't contain `z`, the `I` increments everything and the second `N` undoes the first `N`.) By repeatedly using ROT1, we can rotate the buffer by any amount, as long as it doesn't contain any ROT13 pair (a pair of letters whose ASCII codes are 13 apart).

Now that we have ROT1, we can easily solve the task. Before appending `h` we rotate the finished prefix as needed so that in the current rotated alphabet, the character we want to add is exactly `h`. If we need a capital letter, we can simply use the `C` command before and after the sequence that appends `h`. This temporarily swaps the case of the finished prefix and then changes it back, creating a new `H`.

The only thing left is to deal with the limitation of our ROT1 implementation. We can satisfy the condition by finding a set of letters that is sufficient to generate any digit when run through `E` and yet doesn't contain any ROT13 pair. This set can be found by hand.

Inverting bits

The wannabe scientist Kleofáš has recently developed a new processor. The processor has got 26 registers, labeled `A` to `Z`. Each register is an 8-bit unsigned integer variable.

This new processor is incredibly simple. It only supports the instructions specified in the table below. (In all instructions, `R` is a name of a register, `C` is a constant, and `X` is either the name of a register or a constant. All constants are 8-bit unsigned integers, i.e., numbers from 0 to 255.)

syntax	semantics
`and R X`	Compute the bitwise and of the values `R` and `X`, and store it in `R`.
`or R X`	Compute the bitwise or of the values `R` and `X`, and store it in `R`.
`not R`	Compute the bitwise not of the value `R`, and store it in `R`.
`shl R C`	Take the value in `R`, shift it to the left by `C` bits, and store the result in `R`.
`shr R C`	Take the value in `R`, shift it to the right by `C` bits, and store the result in `R`.
`mov R X`	Store the value `X` into `R`.
`get R`	Read an 8-bit unsigned integer and store it in `R`.
`put R`	Output the content of the register `R` as an 8-bit unsigned integer.

Notes:

- After any instruction other than `not`, if the second argument was a register name, its content remains unchanged.
- Whenever `shl` or `shr` is called, the shifted value of the first argument is truncated on one end and padded with zeroes on the other end. For example, if `X` equals to binary 10110110, then `X shr 2` equals to 00101101.

Example task

Assume that the input contains an arbitrary two 8-bit unsigned integers a and b. Write a program that will read these numbers and produce an integer z such that $z = 0$ if and only if $a = b$. (That is, if a and b differ, z must be positive.)

Solution of the example task

One program solving the example task looks as follows:

```
get A
get B

mov C A
not C
and C B

mov D B
not D
and D A

mov Z C
or Z D

put Z
```

Explanation: There are many possible solutions. In our solution, we read the input to registers `A` and `B`, compute their bitwise xor in register `Z` and output it.

However, we do not have an instruction for xor. We have to assemble it from ands, ors, and nots. First, we create (not-`A` and `B`) in `C` and (not-`B` and `A`) in `D`. Finally, we store (`C` or `D`) in `Z` and we are done.

Why does it work? If the two numbers are identical, then `C` and `D` will both be zero, and so will `Z`. If the two numbers are not identical, consider a digit d where the two numbers differ in their binary representations. If this digit is 0 in `A`, it has to be 1 in `B`. But then it is 1 in `C` and therefore it is 1 in `Z`. And if the digit is 1 in `A`, it has to be 0 in `B`, 1 in `D`, and 1 in `Z`.

Input specification

This problem has no input.

Problem specification

For each data set, you have to write a program for Kleofáš's processor that solves the following task: The input for your program is a sequence of integers, each of them a 0 or a 1. The output of your program must be a sequence of their negations, in order. (That is, change each 0 into a 1 and vice versa.)

Easy subproblem: The sequence contains **exactly 7** integers. In your program, you may only use the instruction `not` at most **once**. Your program must have at most 100 instructions.

(Note that you have to use each of the instructions `get` and `put` exactly 7 times.)

Hard subproblem: The sequence contains **exactly 19** integers. In your program, you may only use the instruction `not` at most **twice**. Your program can have at most 300 instructions.

Output specification

Send us your program as a plain ASCII text. Each line of the text may either contain one complete instruction, or just whitespace. If you submit anything that is not a proper program, it will be judged "Wrong answer" and you will get an appropriate error message.

Task authors

Problemsetter: Michal 'mišof' Forišek
Task preparation: Monika Steinová

Solution

Easy subproblem

There is not much to do in this subproblem. We start by reading the input and storing all input values into bits of one register. Once this is done, we use a single `not` to negate the content of the register, thereby negating all seven input bits. Finally, we extract the seven bits we need from the negated register and output them in the correct order.

Hard subproblem

This subproblem was far from being trivial. At the first glance it seems that using `not` twice we can only negate 16 input bits. Surprisingly, that is not true.

First, we will show how to solve a simpler task. For now, assume that each register can only store a single bit. Can we negate three bits x, y and z using only two `not`s?

Indeed we can, here is one way how to do it. In our description we will use the symbol $\wedge$ for `and`, $\vee$ for `or`, and $\neg$ for `not`. We will use longer, meaningful register names to make the solution more readable. We will start with the three input values in x, y, and z, and we will produce the negations of these three values in three other registers.

1. Compute $lst2 = (x \wedge y) \vee (x \wedge z) \vee (y \wedge z)$. This formula corresponds to the statement "at least 2 inputs are true".
2. Using a `not`, compute $mst1 = \neg lst2$ "at most 1 input is true".
3. Compute $lst1 = x \vee y \vee z$ "at least 1 input is true".
4. Compute $ex1 = lst1 \wedge mst1$ "exactly 1 input is true".
5. Compute $ex3 = x \wedge y \wedge z$ "exactly 3 inputs are true". (Here, "at least 3" is "exactly 3".)
6. Compute $ex13 = ex1 \vee ex3$ "either 1 or 3 inputs are true".
7. Using the second `not`, compute $ex02 = \neg ex13$ "either 0 or 2 inputs are true".
8. Compute $ex0 = mst1 \wedge ex02$ "exactly 0 inputs are true".
9. Compute $ex2 = lst2 \wedge ex02$ "exactly 2 inputs are true".

Using just two `not`s, we just got the four variables $ex0$ to $ex3$. By looking at these, we know the exact number of true inputs, and we can now use this information to construct the correct outputs as follows:

8. $negated{-}x \;=\; ex0 \;\vee\; (ex1 \wedge (y \vee z)) \;\vee\; (ex2 \wedge (y \wedge z))$
9. $negated{-}y \;=\; ex0 \;\vee\; (ex1 \wedge (x \vee z)) \;\vee\; (ex2 \wedge (x \wedge z))$
10. $negated{-}z \;=\; ex0 \;\vee\; (ex1 \wedge (x \vee y)) \;\vee\; (ex2 \wedge (x \wedge y))$

To see why this holds, consider $negated{-}x$ (the other cases are symmetric). If there are no true variables, x is false. If there is 1 true variable and it is one of y and z, x is false, Finally, if there are 2 true variables and both y and z are true, x is false. And clearly this covers all cases when x is false, and precisely in these cases $negated{-}x$ will be true.

Now we are ready to solve the original task. Essentially, with 8-bit registers we are able to do the above process for 8 independent triples of bits at the same time. In other words, this solution would work for up to 24 input bits.

In the solution, we first read the input and pack it into three registers. (In our implementation the first 7 bits are read into `X`, following 7 bits into `Y` and the last 5 bits into `Z`, but you could split them any way you wished.)

Once this is done, we will do exactly the same calculation we described above. We will start by producing `and`s and `or`s of all pairs of registers. (In our implementation, `and`s are stored in registers `K` to `M` and `or`s in `N` to `P`.) Then we compute the first seven steps of the above solution. (These get stored into `A` to `G`.) Finally, we compute the three negated outputs (registers `U` to `W`) and print them out.

Jedi academy

One of the excercises young Jedi must go through in Jedi acedemy training is flying and shooting on a special training machine. Now Luke Skywalker has to examine the results of this training. He wants to know whether the pilots hit something or not.

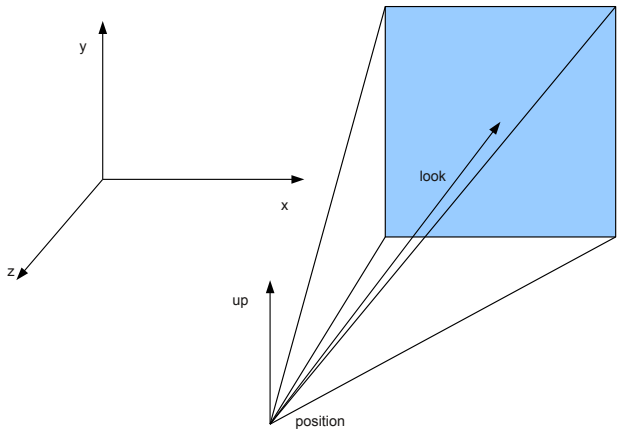
Problem specification

You are given a description of several models of combat ships. Each model is a set of triangles in 3D space. All models are sane (they have a reasonable structure, they are not just random triangles).

The 3D scene consists of several ships called objects. Each object is one of the models, rotated around some axis and moved into some position. The objects are disjoint.

We took snapshots of the scene from several different positions and angles. Each snapshot is described by the camera position (a point), the camera look direction (a vector) and the camera up vector (the direction that will be "up" on the snapshot). Each camera uses a standard perspective projection with a field of view 90 degrees. (Field of view is the angle between the left and right boundary of view, and also the angle between up and down boundary of view. The camera look direction passes through the center of the snapshot.)

The entire scene seen by the camera is projected onto a square screen with fixed dimensions 500×500 pixels. Its lower left corner has coordinates $(0, 0)$, its upper right corner has coordinates $(499, 499)$.



Obr. 2: A right-hand coordinate system (left) and camera description (right).

At the moment of the snapshot, a laser shot was taken from the position of the camera, passing through the point (x, y) on the screen. Your should find whether an object is hit by the shot. If yes, find the closest such object (that is the one actually hit). You may assume that the shot is never on the object boundary (all neighbouring pixels of the snapshot will always correspond to the same object).

Input specification

First line of the input contains an integer m ($1 \leq m \leq 10$), the number of ship models. A description of m models follows. First line of the i-th description contains an integer p_i ($4 \leq p_i \leq 50,000$), the number of points on the model boundary. Next p_i lines contain coordinates of points – three floating point numbers x_{ij}, y_{ij}, z_{ij}. The next line of a description contains the number t_i ($4 \leq t_i \leq 50,000$), the

number of triangles forming the boundary of the model. Each of the last t_i lines describes one triangle using three zero-based indices of points (each number is an integer between 0 and $p_i - 1$, inclusive).

After all model descriptions, the next line contains an integer o ($1 \leq o \leq 5,000$), the number of objects. Next, o object descriptions follow. The object description starts by a number m_i ($0 \leq m_i < m$), model number which is used for the object. The next line contains a description of a rotation of the model. It starts by three floating point numbers x_{ir}, y_{ir}, z_{ir} – the vector specifying the rotation axis. (Always at least one of the numbers is non-zero.). These are followed by a floating point number a_{ir} – the angle of the rotation in degrees. The rotation follows the right-hand rule, so if the vector (x_{ir}, y_{ir}, z_{ir}) points toward you, the rotation will be counterclockwise from your point of view. The rotation axis always passes through $(0, 0, 0)$. The last line of an object description is the object position, three floating point numbers x_{ip}, y_{ip}, z_{ip}. These numbers are added to the coordinates of all points in the model after the rotation (i.e., first we rotate the model, then we translate it).

The next line contains the number v ($1 \leq v \leq 10,000$) of snapshots. Each snapshot is described on four lines. The first line contains the camera position, three floating point numbers. The second line contains the camera look direction, three floating point numbers. The third line contains the camera up vector, three floating point numbers. This vector is always perpendicular to the look direction. The fourth line contains the pixel hit by the shot, two integers c_x, c_y ($0 \leq c_x, c_y < 500$).

Output specification

For each snapshot you should output one number. Either the number of an object hit by the laser shot (object numbers are zero-based), or the number -1, if there is no such object.

Example

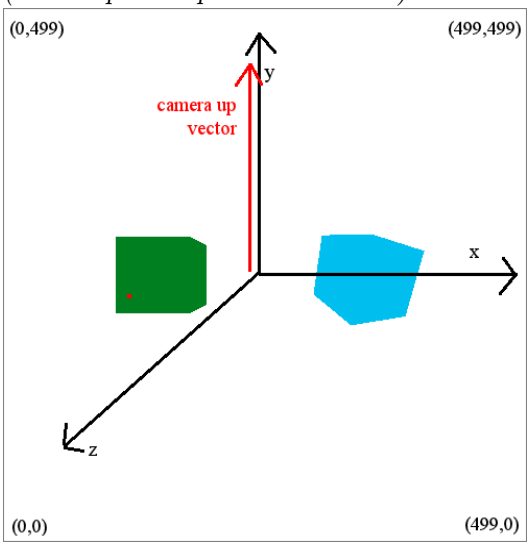
input

```
1
8
-1.000000 -1.000000 -1.000000
1.000000 -1.000000 -1.000000
1.000000 1.000000 -1.000000
-1.000000 1.000000 -1.000000
-1.000000 -1.000000 1.000000
1.000000 -1.000000 1.000000
1.000000 1.000000 1.000000
-1.000000 1.000000 1.000000
12
0 1 2
0 2 3
4 5 6
4 6 7
0 1 4
1 4 5
3 2 7
2 7 6
1 5 2
5 2 6
0 4 3
4 3 7
2
0
0.000000 1.000000 0.000000 0.000000
-3.000000 0.000000 0.000000
0
2.000000 2.000000 0.000000 30.000000
3.000000 0.000000 0.000000
3
0.000000 0.000000 8.000000
0.000000 0.000000 -1.000000
0.000000 1.000000 0.000000
120 229
0.000000 0.000000 6.000000
0.000000 0.000000 -1.000000
1.000000 1.000000 0.000000
380 309
0.000000 0.000000 6.000000
0.000000 0.000000 -1.000000
1.000000 1.000000 0.000000
380 269
```

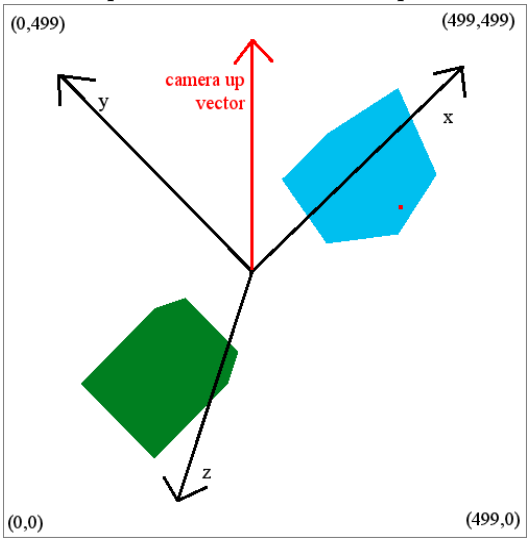
output

```
0
1
-1
```

Here is the picture of the first snapshot (the red point represents the click):



And the picture of the second snapshot:



Task authors

Problemsetter:	Vladimír 'usamec' Boža
Task preparation:	Vladimír 'usamec' Boža

Solution

Given enough time (and motivation) it is possible to implement everything from scratch. From the camera position and the clicked pixel you can compute the direction of a ray that corresponds to the laser shot. To do this, first normalize the camera look direction d and the up vector u. Their cross product is then the right vector r for the camera. Using u and r we can easily turn pixel coordinates into 3D coordinates of a point seen by the camera, and this gives us the ray we needed.

Then all you have to do is to calculate whether there is an intersection between each triangle of each object and this ray and to find the closest intersection.

A nice a short implementation of ray-triangle intersection can be found here: `http://softsurfer.com/Archive/algorithm_0105/algorithm_0105.htm#intersect_RayTriangle()`)

That was **NOT** the intended way how to solve this task.

During the contest, many of you were probably asking questions like: "Are they crazy? Do they really want me to re-implement half of OpenGL from scratch?" Well, we wanted the exact opposite.

The simplest way to solve this task: do not reinvent the wheel, use a library which can do this computation for you. OpenGL is probably the most well-known example of such a library. All you have to do is to use the technique known as picking (one tutorial is for example here `http://nehe.gamedev.net/data/lessons/lesson.asp?lesson=32`), which allows you to find on which object you clicked.

For the hard data set you may also need to optimize the computations a bit. For example, you can make a bounding box for each object and then test for intersection with the bounding box before you test for intersection with each triangle.

Keep clicking, keep flipping

The "Flip it" puzzle is played on a board with black and white tiles. In each step you pick a tile, flip it, and also flip all the adjacent tiles (from white to black and vice versa). The goal of this game is to turn all the tiles to their white side. We found this puzzle too easy, so we picked a harder version for you:

- Instead of square tiles, we will consider a graph with n nodes, each black or white.
- *We only allow clicking on black nodes.*
- When we click on a node x, its colour and the colour of all the adjacent nodes flips.
- Additionally, all edges in the subgraph containing x and all its neighbours are flipped: If two of these vertices were adjacent before, now they are not, and vice versa.
- The goal of the game: At the end, all nodes must be *white* and no two of them may be *adjacent*.

Input specification

The first line of the input file contains an integer t specifying the number of test cases. Each test case is preceded by a blank line.

The first line of each test case contains two integers: n and m. Nodes are numbered from 0 to $n-1$. The second line contains a string of length n. The i-th character of this string is either `B` or `W` and gives the colour of the i-th node at the beginning of the game. Each of the following m lines contains a pair of nodes x_i, y_i that are connected by an edge at the beginning of the game.

In the easy data set, $n \leq 50$, in the hard data set, $n \leq 2000$.

Output specification

For each test case, output the number of clicks and one particular sequence of clicks that wins the game. If there is no such sequence, output a single line containing the number -1.

Example

input

```
2

3 2
BBW
0 1
0 2

7 12
WBBWWWW
0 2
0 3
1 2
1 4
1 6
2 3
2 4
2 6
3 4
3 6
4 5
5 6
```

output

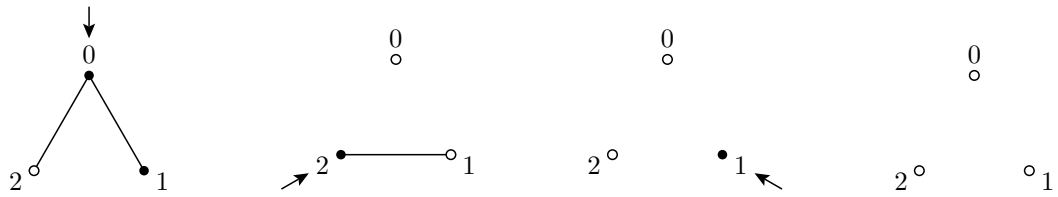
```
3
0 2 1

5
1 6 3 0 2
```

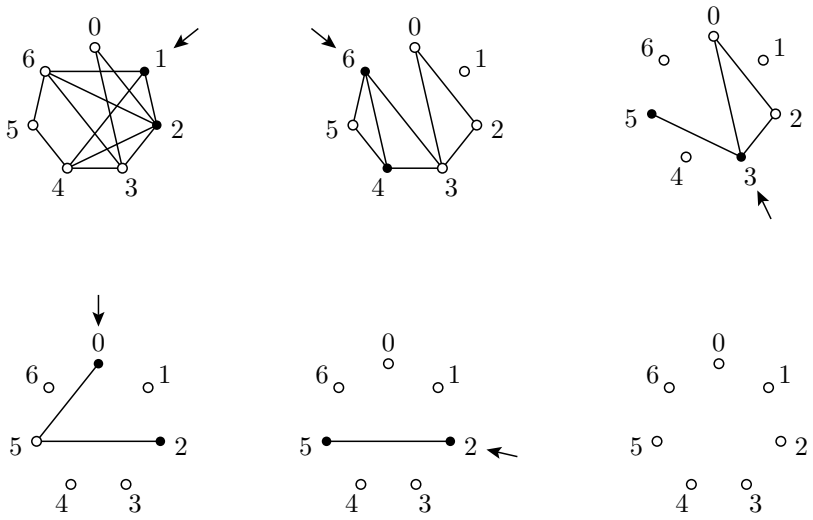
Note that in the first test case, if we started by clicking at 1, we would end up stuck with two adjacent white vertices.

The figures below illustrate the solutions for the two test cases.

Test case #1:



Test case #2:



Task authors

Problemsetter:	Jakub 'kuko' Kováč
Task preparation:	Jakub 'kuko' Kováč

Solution

Let us call a connected component (with more than one vertex) containing only white vertices a *bad* component. Obviously, if we are given a graph with a bad component, we cannot possibly win the game (we can't click on the vertices and they will never become adjacent to any other vertices). Thus, generally, when playing the game, we will try to avoid creating bad components.

We will prove that if all the components are good (contain a black vertex), there is always a black vertex such that clicking on it is *safe* – it doesn't create any bad components. Thus, by searching and clicking on safe vertices, we can win the game.

One possible solution is to try all the black vertices: click on them, check whether a bad component was created (if so, undo it and try another). This algorithm runs in $O(n^4)$ time, since there are at most n clicks, in each step we try at most n black vertices and the checking can be done in $O(n^2)$ by depth or breadth first search.

For a more efficient algorithm, we will prove that the following greedy strategy always works: In each step, we will search for a black vertex such that the number of remaining black vertices after the click is as big as possible. In other words, let u be a black vertex; its *score* is the number of neighbouring white vertices (these will turn black) minus the number of neighbouring black vertices (these will turn white). We will prove that we can win the game by always clicking on a vertex with the highest score.

Statement: *If u is a black vertex with the highest score in G, then it is safe to click on it.*

Proof: The proof is by contradiction: let u be a vertex with the highest score, let G' be the graph which results from G after we click on u and let C be a bad component in G'. Since G did not contain any bad components, there must have been a black vertex v in C, which was adjacent to u in G.

Let b, w and b', w' be the number of black and white neighbours of u and v respectively. Now, all the white neighbours of u must have been neighbours of v (otherwise, they would become black and adjacent to v). Therefore, $w' \geq w$.

Furthermore, all the black neighbours of v must have been neighbours of u (otherwise, they would remain black and adjacent to v), so $b' \leq b$.

From the two inequalities, we have $w' - b' \geq w - b$ and since u had the highest score, this is only possible if $w = w'$ and $b = b'$, which means that u and v share exactly the same neighbourhood. However, in this case, v becomes an isolated vertex in G', which is *not* a bad component – a contradiction.

Acknowledgement: The proof is from Bergeron, *A very elementary presentation of the Hannenhalli-Pevzner theory.*

This solution runs in $O(n^3)$ time and can be further improved by using bit parallelism.

Lame crypto

Bob wants to draw tons of .png images (`http://www.w3.org/TR/PNG/`) and send them to Alice. He does not want anyone else to see those pictures. Therefore, he designed the Super Secret Protocol (SSP) and he uses it to communicate with Alice.

Eve wants to destroy their relationship. She bribed their internet providers to slow down their internet so that she could get access to their communication. Now, every message between Bob and Alice travels so slow that Eve can hold it for up to several hours. She has also drawn an evil image, which she wants to deliver to Alice in Bob's name. Then Alice will break up with Bob for sure[2].

Unfortunately (for Eve), Eve knows exactly nothing about cryptography. Therefore, she hired IPSC organizers to take care of it. And as they were just preparing this contest, they decided to kill two birds with one stone – and let you do the dirty work for them. The evil images (one for each subproblem, see below) are provided in the respective input files.

SSP protocol specification

Alice has a secret key K_A, Bob has a secret key K_B. When Bob wants to send a message M of length n to Alice, the following happens:

1. Bob chooses some (contiguous) subsequence K'_B of size n of key K_B. He sends $M \oplus K'_B$ to Alice.
2. Alice receives message $C = M \oplus K'_B$. She chooses some (contiguous) subsequence K'_A of size n of her key K_A and sends $C \oplus K'_A$ to Bob.
3. Bob receives message D, sends her back $D \oplus K'_B$.
4. Alice receives $E = D \oplus K'_B$. Since $\oplus$ is commutative, $A \oplus A = 0$ and $A \oplus 0 = A$, we know that $E = M \oplus K'_A$. Hence Alice can now compute M as $E \oplus K'_A$.

Notes: $\oplus$ is bitwise xor. Bob needs exactly six minutes to draw an image. Therefore he will always start sending the next image six minutes after Alice receives the previous one.

Problem specification

Your task is to monitor the messages sent between Alice and Bob and to change some of the messages in such a way that Alice will get the evil image.

In the easy subproblem, you can change any message you want. However, in the hard subproblem, you can only change the messages that go from Alice to Bob.

Note that the hard subproblem uses different secret keys and other messages than the easy subproblem. The protocol remains the same.

Input/output specification

We will provide you with two web pages (one for the easy subproblem, one for the hard one) where you can download all messages that were already sent (possibly with your modifications). **You will find a link to these web pages in the Messages section of the website.**

Messages are in binary format, each one has exactly 10000 bytes. The most recently sent message in on the top of the list. If you want to change this message, submit (using the standard submission interface) a binary file containing the new message. If you just wish to deliver it in its original state, submit a text file containing one line with a single word: "`forward`".

Valid submissions during the protocol are **not** counted as incorrect – you will **not** receive penalty minutes for these submissions.

[2] Well, with 95% confidence.

In the hard subproblem you can only change one message from the protocol, the other ones will be forwarded to their recipient automatically.

In both subproblems, the time between any two submissions in different instances of the protocol has to be at least 6 minutes. In other words, you may not try to submit anything while Bob is drawing the next picture. Submissions that violate this rule will be judged as wrong answers; upon solving the task you **will** receive penalty minutes for these submissions.

The replaced message always has to have the same size as the original message (otherwise you will get an wrong answer).

At the end of the protocol, if Alice received anything other than the evil image, you will also get a wrong answer. If Alice receives the evil image, you solved the subproblem.

The evil image and all messages sent by Alice or Bob have exactly 10000 bytes. All images are valid PNG images. Alice and Bob have keys of size 50000 bytes.

Task authors

Problemsetter:	Michal 'Mic' Nánási
Task preparation:	Michal 'Mic' Nánási

Solution

We will now show that the SSP protocol is neither super, nor secret.

Easy subproblem

Let A, B, C be the messages sent by participants in our protocol, let M be the original message and E be the evil message. Let K'_A and K'_B be the subkeys used for encryption in this run of the SSP protocol. From the description of the protocol, we have:

$$\begin{aligned} A &= M \oplus K'_B \\ B &= M \oplus K'_B \oplus K'_A \\ C &= M \oplus K'_B \oplus K'_A \oplus K'_B = M \oplus K'_A \end{aligned}$$

Since operation $\oplus$ is commutative, $X \oplus X = 0$ and $X \oplus 0 = X$, then $A \oplus B = K'_A$. Therefore from the first two messages from the protocol we know Alice's private key. Using this key, we can replace the third message by $E \oplus K'_A$.

Another approach is to replace the first message with the message that contains only zeros. Then the second message will contain Alice's private key.

Hard subproblem

This subproblem is little more tricky. From the first two messages we can deduce the key Alice just used, but that is useless, since we now cannot change the third message – we could only change the second one, and at that moment we did not have enough information to do so. If we knew the Bob's private key K'_B, we could replace the second message with $E \oplus K'_B \oplus A \oplus B$, where $A \oplus B$ is in fact K'_A (see solution for easy subproblem).

How to get the subkey K'_B? When we took the first two messages (A and B), we could compute K'_A. Similarly, K'_B can be computed as $B \oplus C$. But by the time we can compute subkey K'_B, we cannot change any message in the current instance of the protocol.

We can use alternative approach to reconstruct Bob's subkey by submitting message containing zeros back to Bob. The third message will contain Bob's subkey. However, if we use this method Alice will recover some random noise and we will get an wong answer for every such submission.

We will now exploit the fact that Bob's entire secret key K_B is not much larger than the individual keys K'_B used in protocol instances. If we listen to multiple communications, some subkeys will necessarily overlap, and after some time we should be able to reconstruct almost all of K_B.

For each protocol instance, after receiving the first message we can try decrypting it using all subkeys that we can already infer from our partial reconstruction of K_B. If one of these subkeys gives us a valid PNG image, we just found K'_B. In this protocol instance we can change the second message, and at its end we win. If no subkey we tried works, we will simply forward the second message and use the second and third message to compute a new subkey.

Last piece of the puzzle is the recognition of PNG images. Every PNG image starts with an 8 byte sequence 137, 80, 78, 71, 13, 10, 26, 10. It is pretty unlikely that K_B would contain two equal 64-bit long segments – if these 8 bytes are decrypted correctly, we have the right subkey with almost 100% certainty. (Of course, we could also use a library such as ImageMagick to do an exact check.)

My little puppy

You have a puppy. The puppy wants your attention. As every puppy, it wants to sleep, eat, play, and sometimes something more. Be careful, be patient and read its requests carefully. Your reward will be priceless. Well, not really priceless. Your reward will be paid in negative penalty time.

Each time you answer the puppy's request, the puppy will become idle for an unknown period of time. Then the puppy will come with a new request. You have to process that request reasonably quickly. The deadline to process the new request is somewhere between 15 and 25 minutes since it was published. If you miss the deadline or choose an incorrectly option, you lose and your puppy runs away forever. If you choose correctly, you will be rewarded.

You will find a link to the web page with the puppy in the Messages section of the website.

Input specification

For each request, you will get an ASCII art of the situation, and a list of available options you have. Each option is labeled by a lowercase letter.

Output specification

During the appropriate time interval, submit a text file containing a single line with a single letter – the option you chose.

(The data set for the submission can be either M1 or M2, both will work. You are only playing one game, not two of them.)

You may submit your choice for the first request at any moment during the contest. Your time starts running once you make this first submit. (It is recommended to start as early as possible, so that you have the chance to see many requests.)

Example

input

```
  |\_/|
 / @ @ \
( > º < )
 `>>x<<´
 /  O  \

What do you see?

a) a kitten
b) an elephant
c) a dog
d) where?
```

output

```
a
```

Task authors

Problemsetter:	Martin 'rejdi' Rejda
Task preparation:	Martin 'rejdi' Rejda

Solution

This task was just a toy to play with, relax and get some bonus time. There were 16 levels (screens) and 15 answers. Each level had one or more correct answers, and each correct answer gave you -20 minutes to your total time.