IPSC 2018

problems and sample solutions

Partitioning a square **3**
Solution 4

Qizz Quzz **5**
Solution 7

Raw data **9**
Solution 10

Armed bandit **11**
Solution 13

Brain fold **14**
Solution 16

Counting rectangles **18**
Solution 20

Delightful **22**
Solution 24

Encrypted romance **28**
Solution 30

Farmer's code **32**
Solution 33

Git gud **34**
Solution 35

Hats of various colors **38**
Solution 40

Incorrect expression **41**
Solution 42

Jumping over walls **43**
Solution 45

Kids draw the darndest things **47**
Solution 49

Lethargic foe **51**
Solution 53

McDroid's **56**
Solution 58

 licensed under CC BY-SA 4.0 int'l

Problem P: Partitioning a square

In this problem you are given a number n and your task is to produce a square of letters.
The square must have the following properties:

- Each letter is one of the first n lowercase English letters (`a-z`).
- Each of those n letters occurs the same number of times.
- All occurrences of each letter form one 4-connected region (explained below).
- The square is **as small as possible**.

We say that the occurrences of some letter X form a 4-connected region if it is possible to travel from any X to any other X by only moving one step up/down/left/right at a time without stepping onto a letter other than X.

Below is an example of a 6×6 square divided into $n = 4$ equally large 4-connected regions. (Note that this is not the *smallest possible* square that can be divided into 4 regions.)

```
aaaaab
aaaabb
bbbbbd
bddddd
cdddcc
cccccc
```

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 positive integer n.

In the **easy subproblem P1** we have $t = 1$ and the only test case has $n = 4$.

In the **hard subproblem P2** we have $t \leq 50$ and each n is between 1 and 26, inclusive.

Output specification

For each test case, output several lines. The first of those lines should contain an integer s: the side length of your square. The rest of the output for that test case should consist of s lines, each containing s lowercase letters.

What to submit

Do not submit any programs. Your task is to produce and submit the correct **output files** `p1.out` and `p2.out` for the provided input files `p1.in` and `p2.in`.

Example

input

```
1

2
```

output

```
2
ab
ab
```

 licensed under CC BY-SA 4.0 int'l

Task authors

Problemsetter:	Michal "mišof" Forišek
Task preparation:	Michal "mišof" Forišek
Quality assurance:	Michal "majk" Švagerka

Solution

The easy subproblem was easily solvable by hand. The smallest square that can be divided into four regions is a 2×2 square. One possible correct output looks as follows:

```
ab
cd
```

For the hard subproblem, we can start by making the following observation: We want to take a square of some size s and divide it into n equally-large regions. Obviously, that is only possible if n divides s^2. Thus, if we find the smallest such s and successfully construct an $s \times s$ square, we can be certain that it's optimal.

Suppose we have said $s \times s$ square. How can we divide it into n connected regions? There are many techniques that work. For example, you could choose any Hamiltonian path (i.e., a path that visits each cell of the square once) and split it into n segments.

Probably the easiest solution in terms of implementation is the following one:

The size of each region will be $r = s^2/n$. It is obvious that r is also a divisor of s^2. But we can show a stronger statement: in fact, if s is as small as possible, r must be a divisor of s. This is because for any prime p we have two possibilities:

- If the largest power of p that divides n is even, say $2k$, then s divisible by p^k and r is not divisible by p.
- If the largest power of p that divides n is odd, say $2k+1$, then s divisible by p^{k+1} and r divisible by p.

Thus, for the construction we simply split each row of the square into s/r regions.

Example output for $n = 18$:

```
6
aabbcc
ddeeff
gghhii
jjkkll
mmnnoo
ppqqrr
```

 licensed under CC BY-SA 4.0 int'l

Problem Q: Qizz Quzz

After a successful presentation of your coding skills on the Fizz Buzz problem you have advanced to the second round of interviews for your dream job of code ninja at FizzBuzz Corp. In this interview you will tackle the Generalized Fizz Buzz problem.

Generalized Fizz Buzz is similar to the original Fizz Buzz. The program must print positive integers starting from 1, but some special numbers (and their multiples) should be replaced by strings. If something is divisible by multiple special numbers, it must be replaced by a concatenation of all their strings.

In this problem, your task **isn't** to implement Generalized Fizz Buzz. Instead, you will be given a sequence and you need to decide whether it can be the output of a Generalized Fizz Buzz program.

Generalized Fizz Buzz

A particular instance of Generalized Fizz Buzz is defined by the following parameters:

- A positive integer n.
- A nonnegative integer k.
- Distinct positive integers $d_1, \dots, d_k$ such that $d_1 < d_2 < \cdots < d_k$.
- Strings $s_1, \dots, s_k$ of lowercase English letters.

Their meaning is as follows:

- The number n is the length of the sequence the program should output.
- The number k is the number of replacement rules.
- Each of the replacement rules has the form "Instead of multiples of d_i print the string s_i."
- If multiple replacement rules apply, the label that should be printed is obtained by concatenating all the corresponding s_i, ordered **from the smallest to the largest divisor**.

For example, the original Fizz Buzz corresponds to $k = 2$, $d = (3, 5)$, and $s = (\texttt{fizz}, \texttt{buzz})$. The correct output of that program for $n = 17$ would be:

```
1 2 fizz 4 buzz fizz 7 8 fizz buzz 11 fizz 13 14 fizzbuzz 16 17
```

Note that instead of 15 we printed the concatenation of s_1 (`fizz`) and s_2 (`buzz`) because 15 is divisible both by d_1 (3) and by d_2 (5). Also note that you cannot print `buzzfizz` for 15 because you cannot change the order of the s_i when concatenating them.

Subproblem Q1

Samko has written his own implementation of a Generalized Fizz Buzz program. However, Samko's implementation has some additional constraints: It can only handle inputs where $k \leq 2$ and each of the strings s_i has length exactly 4.

Samko has shown you a sequence of labels $\ell_1, \dots, \ell_n$. Find the largest q such that the prefix $\ell_1, \dots, \ell_q$ can be the output of Samko's program.

Subproblem Q2

Monika has also written her own implementation of a Generalized Fizz Buzz program. Monika's program can only handle inputs where each of the strings s_i has between 1 and 25 characters, inclusive. (Her program can handle arbitrarily large values of k and d_i.)

Monika has shown you a sequence of labels $\ell_1, \dots, \ell_n$. Find the largest q such that the prefix $\ell_1, \dots, \ell_q$ can be the output of Monika's program.

 licensed under CC BY-SA 4.0 int'l

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 contains a positive integer $n \leq 1000$: the number of labels in the sequence you were given. The second line contains the sequence: n non-empty tokens $\ell_1, \ldots, \ell_n$ separated by single spaces. Each token will only contain digits and lowercase English letters.

Output specification

For each test case output one line with the corresponding integer q.

Make sure that in **subproblem Q1** the q you output corresponds to Samko's program, while in **subproblem Q2** your output should correspond to Monika's program.

Example

input

```
5

7
mlem mlem mlem mlem mlem mlem fail

7
1 qizz quzz qizz 5 quzzqizz 007

8
1 2 3 4 hunter2 is my password

2
haha hahahaha

9
1 bat 3 batman robin batbat 7 batmanman 9
```

output for Q1

```
6
5
4
2
1
```

output for Q2

```
6
5
4
2
9
```

- In the first example, the first six tokens are the output of a Generalized Fizz Buzz program for $k = 1$, $d_1 = 1$, and $s_1 =$ `mlem`. Both Samko's and Monika's program can handle this input, so for each of them the answer is 6.
- In the second example, it's clear that 2 must be replaced with `qizz` and 3 with `quzz`. The first five tokens match that sequence, but the next one should be `qizzquzz` instead of `quzzqizz`, and the last one should be `7` instead of `007`.
- In the third example, `hunter2` is wrong because it contains both letters and numbers.
- In the fourth example, $d = (1, 2)$ and $s = (\texttt{haha}, \texttt{haha})$. Note that the strings s_i do not have to be distinct.
- In the fifth example, `bat` is already wrong in Q1 because Samko's program cannot handle a three-character string. In Q2 the whole sequence is correct because it's possible that 2 was replaced by `bat`, 4 was replaced by `man`, 5 by `robin`, 6 also by `bat`, and 8 also by `man`.

 licensed under CC BY-SA 4.0 int'l

Task authors

Problemsetter:	Tomi Belan
Task preparation:	Monika Steinová
Quality assurance:	Samuel “Hodobox” Gurský, Michal “mišof” Forišek

Solution

Let us assume that all ℓ_i are either integers or consist of lowercase English letter only. A label consisting of a mixture of digits and letters cannot be a valid label, and a label that begins with a `0` cannot be a valid label either. We can cut the sequence before the first such offending label before processing it in the ways described below.

Easy subproblem

For the easy subtask it was sufficient to try all reasonable possibilities (remember that $k \leq 2$) and to pick the best one among them.

In other words, we can try the following cases:

- For $k = 0$ we just generate the sequence of positive integers.
- For $k = 1$ we try all possible $1 \leq d_1 \leq n$, for each of them we check that ℓ_{d_i} is a 4-character string, and if it is, we generate the corresponding “Fizz sequence”.
- For $k = 2$, once we have a d_1 that was valid for $k = 1$, we also try all $d_2 > d_1$ and whenever we find one that produces a second valid label, we generate the corresponding Fizz Buzz sequence.

Each time we gerate a sequence, we simply find its longest prefix that matches the input sequence.

Easy subproblem again

The previous solution works but depending on the implementation and programming language used it can already be annoyingly slow. Luckily, there is a simple improvement that will speed up the search significantly.

Suppose we are in the case $k = 1$ and we found the smallest j such that the given label ℓ_j is a string. Then obviously $d_1 = j$ is the only possibility we need to try. Larger d_1 won’t give us a better solution because its sequence will have the number j instead of the correct label ℓ_j in j-th position.

Then, in the case $k = 2$, this d_1 is again the only possible d_1 for the same reason. So instead of $\Theta(n^2)$ pairs (d_1, d_2) we now only have to try $\Theta(n)$ possible d_2 with this single d_1.

Hard subproblem

Instead of applying brute force we will now try to deduce the d_i and s_i greedily. This solution will be a fairly straightforward generalization of our better solution for the easy subproblem.

We will go through the given labels from 1 to n. At the beginning, there are no pairs (d_j, s_j).

Each time we process a new label ℓ_i, there are two possibilities:

1. The label ℓ_i is an integer. In this case we need to check two things. First, i must not be a multiple of any of the d_j we already have. Second, ℓ_i must actually be i, not some other integer. (Did you miss this on your first try?) If either check fails, we have no way of saving it, this is the end of the valid prefix.
2. The label ℓ_i is a string. In this case we look at the pairs (d_j, s_j) we already determined, and we use them to construct the (possibly empty) label ℓ'_i we should see according to what we know. Now there are three cases:

 licensed under CC BY-SA 4.0 int’l

- If $\ell_i = \ell'_i$, everything is in order and we move on.
- If ℓ'_i **is not** a proper prefix of ℓ_i, there is no way to fix this problem and thus this is the end of a valid prefix.
- If ℓ'_i **is** a proper prefix of ℓ_i, there is exactly one way to fix this problem: by introducing a new rule with step $= i$ and label $=$ the missing part of ℓ_i.

This way we incrementally construct the only possible set of rules that corresponds to the prefix we already processed. Hence, when we get stuck, we can be certain that the solution we just found is optimal.

An alternate implementation of the same idea is that whenever you determine a new rule (d_i, s_i), you immediately look at all multiples of d_i. If the label of a multiple of d_i starts with s_i, you remove the prefix s_i and keep the rest of the label. And if it doesn't, you cut off the input sequence at that point.

 licensed under CC BY-SA 4.0 int'l

Problem R: Raw data

We received an anonymous tip that a famous hacker group, the Inquisitors of Poorly Secured Code, will try to remotely disable your web browser during tomorrow's contest. We want to make sure you'll still be able to reach us even if your web browser stops working.

Problem specification

To solve the **easy subproblem R1**, send three HTTP POST requests to the address `https://ipsc.ksp.sk/2018/practice/problems/r1`. You must be logged in (the request must have the correct cookies). The POST request body can contain anything.

To solve the **hard subproblem R2**, send three HTTP POST requests to the address `https://ipsc.ksp.sk/2018/practice/problems/r2`. You must be logged in *and* you can't use a web browser. The POST request body can contain anything.

Input specification

There is no input.

Output specification

There is no output. You don't have to submit any files, you will automatically get an **OK** verdict when you send the correct requests.

 licensed under CC BY-SA 4.0 int'l

Task authors

Problemsetter:	Tomi Belan
Task preparation:	Tomi Belan
Quality assurance:	Michal “mišof” Forišek

Solution

The easiest way to be logged in is to use your browser to actually log in. This will create a session and store the session info in your browser's cookies. You can then access these cookies and copy them out. (Depending on your browser, you might need to install an extension, but usually you can examine cookies in your privacy settings.

In our case, we got the `cookies.txt` file shown below. (The values in the last column have been truncated.)

```
# Netscape HTTP Cookie File
.ksp.sk TRUE    /    FALSE    1597233002  _ga GA1.2.1717...
ipsc.ksp.sk FALSE   /    FALSE   0    ipscsessid  8cbcdc...
ipsc.ksp.sk FALSE   /    FALSE   0    ipsc2018ann 3
```

You can then use some tool that will send a specific HTTP request to our webserver and tell it to include these cookies.

The two most commonly used tools for this job are the command-line tools `wget` and `curl`.

They can be used as follows:

```
wget --load-cookies=cookies.txt \
    https://ipsc.ksp.sk/2018/practice/problems/r2 --post-data=hello

curl --cookie cookies.txt --data hello https://ipsc.ksp.sk/2018/practice/problems/r2
```

The HTTP requests made by your browser contain many interesting fields that are not necessary for the bare communication. In particular, your browser identifies itself and its operating system to the webserver by sending a “User-Agent string”. These additional fields were used by our grader to make a heuristic guess whether you are using a browser or not. Thus, you *could* make the request to solve R2 also from your browser as long as you used its developer tools to alter/remove the User-Agent string it sends.

 licensed under CC BY-SA 4.0 int'l

Problem A: Armed bandit

You are in a casino, and you are anxious to play on the new *one-armed bandit* – a slot machine they just installed.

The slot machine has n wheels in a row. The i-th wheel from the left contains the integers from 1 to k_i, inclusive. When you pull the handle, the wheels will spin for a while. When they stop, each wheel will show you one of its integers. The integers are randomly chosen. All random choices are uniform and mutually independent.

Problem specification

You are given the number of wheels n and the size of every wheel k_i.

Easy subproblem A1: Output any sequence of numbers you could see on the given slot machine.

Hard subproblem A2: Suppose we ignore the spaces between the wheels and read the sequence of numbers as one long string of digits. Find the sequence of numbers that produces the *lexicographically smallest* such string.

Lexicographic order

When comparing two different strings S and T, find the smallest index i at which they differ. The one with a smaller character at that index is lexicographically smaller than the other. If there is no such character (i.e., one of the strings is a prefix of the other), the shorter string is the lexicographically smaller one.

For example:

- `22` is lexicographically smaller than `220` because `22` is a prefix of `220`
- `123` is lexicographically smaller than `14` because `2` is less than `4`.

Input specification

The first line of the input file contains an integer $t \leq 100$ specifying the number of test cases. Each test case is preceded by a blank line.

Each test case begins with a line consisting of a single integers n. The following line contains n space-separated integers $k_1, \ldots, k_n$. In both subproblems we have $1 \leq n \leq 1\,000$ and $\forall i : 1 \leq k_i \leq 100$.

Output specification

For each test case, output a single line with n integers separated by single spaces: the numbers as they appear on the individual wheels of the machine.

What to submit

Do not submit any programs. Your task is to produce and submit the correct **output files** `a1.out` and `a2.out` for the provided input files `a1.in` and `a2.in`.

 licensed under CC BY-SA 4.0 int'l

Example for subproblem A1

input

```
2

3
6 6 6

2
10 2
```

output for A1

```
2 4 6
7 2
```

There are many other correct outputs. Any correct output will be accepted.

Example for subproblem A2

input

```
2

3
6 6 6

2
10 2
```

output for A2

```
1 1 1
10 1
```

The lexicographically smallest string consisting of 3 integers from 1 to 6 (inclusive) is `111`.

The lexicographically smallest string consisting of an integer from 1 to 10 (inclusive) and an integer from 1 to 2 (inclusive) is `101`. Note that this string is smaller than `11` because the second character in `101` is smaller than the second character in `11`.

 licensed under CC BY-SA 4.0 int'l

Task authors

Problemsetter:	Michal “mišof” Forišek
Task preparation:	Monika Steinová
Quality assurance:	Samuel “Hodobox” Gurský

Solution

Easy subproblem

Sure, you *could* go with the spirit of the task and generate the output using a random generator, but why bother? It was much easier to output n ones, or simply to copy the sequence $k_1, \dots, k_n$ from the input to the output.

But the actual “pro move” is to solve the hard subproblem and to submit its solution also as the solution to the easy subproblem.

Hard subproblem

When comparing two strings, we are looking at their characters from the left to the right, until we find the first difference. Hence, if we want to make sure that our string will be the lexicographically smallest of all possible strings, we have to construct it greedily from the left to the right. At each position we have to make sure we use the smallest character possible.

What is the best character we can use? There are three possible answers, in order of preference:

- The very best character is no character at all, i.e., the end of the string. This can only be chosen if we are already on the n-th wheel.
- The next best option is a `0`. Whenever the next character can be a `0`, we must make it a `0`.
- If we cannot use a `0`, we have to use a `1`.
- We will never need to use any bigger digits, because one of the above will always apply.

Whenever we move to a new wheel, we have to use a `1` as the next character because the numbers on the wheel don’t start with a `0`. Afterwards, we may have an option to add some `0`s: one if $k_i \geq 10$ and another if $k_i = 100$.

This gives us a very simple algorithm:

- For each of the wheels 1 through $n-1$: use the largest of the numbers $\{1, 10, 100\}$ that is on the wheel.
- For wheel n: use the number 1 (because stopping there is better than adding zeros).

 licensed under CC BY-SA 4.0 int’l

Problem B: Brain fold

Shandyna loves nerd sniping. (Relevant xkcd. Don't worry, she only does it in safe enviroments.)

Last time she got five of them at once. When I saw them, they were still sitting around, trying to imagine a folded sheet of paper.

Problem specification

You have a rectangular piece of paper. You fold it in half several times. For each fold, you pick up one side and place it over the opposite side. There are four sides to pick, so there are four ways to perform each fold. (Two of the folds are horizontal and two are vertical.)

Once you finish the last fold, you pick a straight line and cut along it, through all layers of the folded paper. How many pieces of paper will you have at the end?

Input specification

The first line of the input file contains an integer $t = 100$ 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 a number n ($1 \leq n \leq 10^5$) denoting the number of folds.

The second line consists of n characters specifying the folds in the order they are applied: `T`, `B`, `L` and `R` represent folds for which you pick up the top, bottom, left, and right side, respectively. (Hence, `T` represents the fold that places the top side over the bottom side.)

The last line describes the cut. To make your life easier, the cut will never pass through a corner of the square. As it can be shown that the number of pieces does not depend on the exact points where the cut intersects the sides of the square, we can specify the cut simply by giving the labels of the two sides it intersects. For example, `TR` is a cut that intersects the top and the right side of the folded paper.

In the **easy subproblem B1** each cut can be made horizontally or vertically. That is, the two letters that describe each cut are either `T`+`B` or `L`+`R`.

Output specification

For each test case, output a single line containing x modulo $10^9 + 7$, where x is the number of paper pieces we're left with after cutting the paper.

Example

input

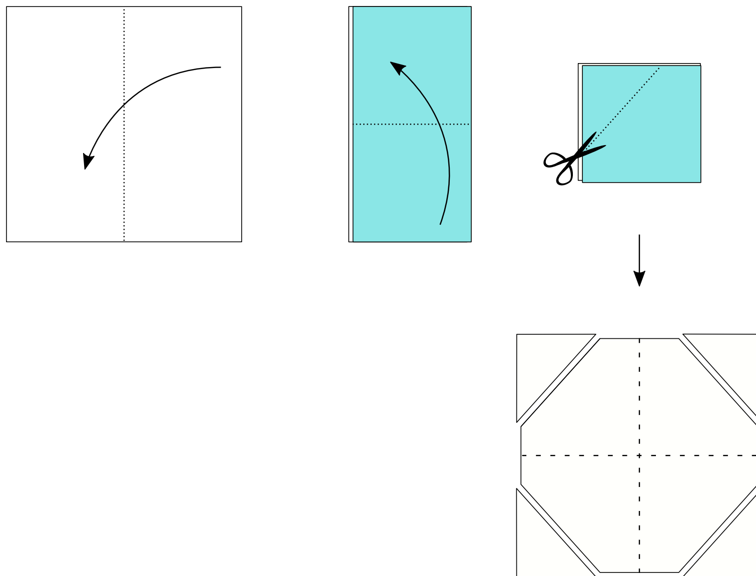
```
2

2
LB
TB

2
RB
LT
```

output

```
3
5
```

The second test case is shown in the figure below. This test case will not appear in subproblem B1.

 licensed under CC BY-SA 4.0 int'l



 licensed under CC BY-SA 4.0 int'l

Task authors

Problemsetter:	Michaela "Šandyna" Šandalová
Task preparation:	Jaroslav Petrucha, Michaela "Šandyna" Šandalová
Quality assurance:	Samuel "Hodobox" Gurský, Michal "mišof" Forišek

Solution

Easy subproblem

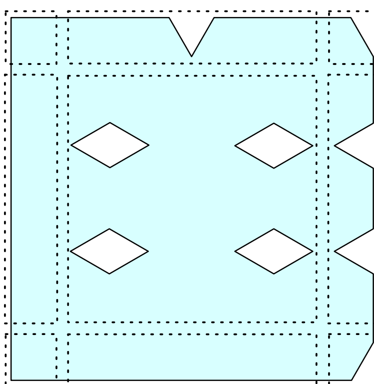
The easiest way to approach this subtask without much thought would be to try several different sequences of folds and cuts. By experimenting, you could find that when the cut is horizontal, only folds along a horizontal line (i.e. Top and Bottom) do actually matter (similarly with vertical cuts). You could probably even discern the correct formula: if there are n horizontal folds and a vertical cut, the number of paper pieces is $2^n + 1$.

The reason for this is quite simple: imagine unfolding the paper that underwent n vertical and m horizontal folds. It consists $2^n \times 2^m$ little pieces of paper connected by folds. If the final cut was horizontal, each row contains one cut from left to right (if it was vertical, each column contains one cut). Thus there are 2^n cuts resulting in $2^n + 1$ pieces of paper.

Hard subproblem

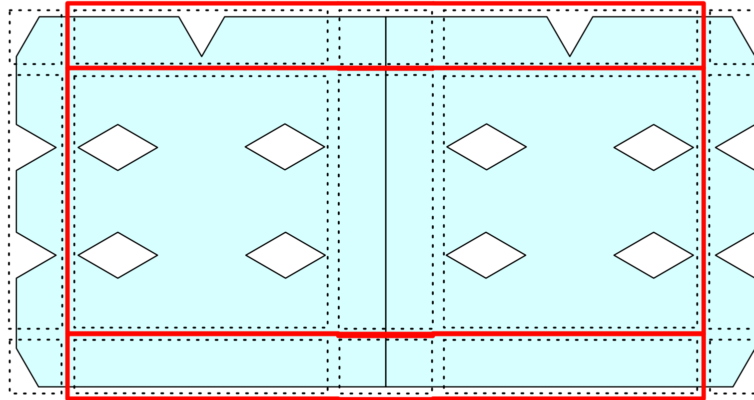
In the easy subproblem, we could find the answer by looking at the unfolded paper. While it may be possible to construct the solution for corners in this way, our solution will unfold the paper gradually, one fold at a time. And we will be counting holes in the paper. Each hole will correspond with one piece of paper apart from the "main" one.

We will split the holes into 9 categories as shown in the picture below: 0-1 hole in each of the corners, some number of holes on one of the sides (but not in any corner) and holes completely contained inside of our paper. Note how the interesting areas are arranged in a 3×3 array.



In the beginning, we have one hole in one of the corners. Afterwards, we process the unfolds in reverse order. During each unfold, we update counts in areas. We will describe the update for Left unfold, all the others are analogous. Please refer to the picture below, if anything becomes unclear.

 licensed under CC BY-SA 4.0 int'l



All the counts in interesting areas in the right column stay the same. All the counts in the left column will become equal to the counts in the right column. The counts in the middle column are the most interesting: we double the original count and add the count from left column.

After we're done processing the unfolds, we just sum up all the holes we found: each hole corresponds with a fallen out piece of paper. Don't forget to add one for the original paper!

 licensed under CC BY-SA 4.0 int'l

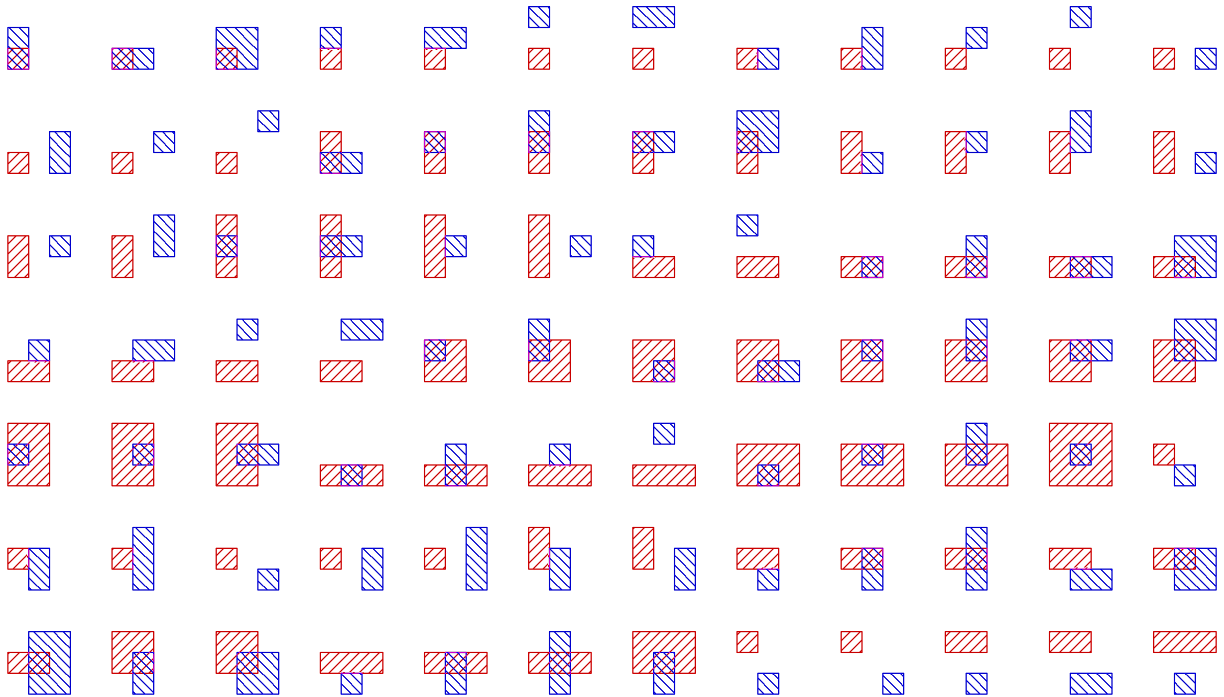
Problem C: Counting rectangles

This entire problem takes place in a standard two-dimensional plane.

Your task is very simple: count all *fundamentally different* sets of n axes-parallel rectangles.

Informally, two sets are fundamentally different if one cannot be transformed into the other by horizontally and vertically stretching some parts of the plane. A precise formal definition is given below.

For $n = 2$ there are 84 such sets. All of them are shown in the following figure. (The two rectangles are indistinguishable but in our figure we show one in red and the other in blue to make the figures where they overlap easy to read.)



Formal definitions

Given an ordered pair of points (P, Q), their relative position is $rp(P, Q) = (sgn(P.x - Q.x), sgn(P.y - Q.y))$. Hence, there are nine possible relative positions, including "Q is somewhere to the left and down of P", "Q is exactly below P", and "P and Q are the same point".

An axes-parallel rectangle is a rectangle such that each of its sides has a positive length and lies parallel to one of the coordinate axes. For an axes-parallel rectangle R we will use $R[0]$ through $R[3]$ to denote its bottom left, bottom right, top left, and top right corner.

Two sets of axes-parallel rectangles $\mathcal{S}$ and $\mathcal{T}$ are called similar if there is a one-to-one map φ from $\mathcal{S}$ to $\mathcal{T}$ such that

$$\forall S_1, S_2 \in \mathcal{S} : \ \forall i, j \in \{0, 1, 2, 3\} : \ rp(S_1[i], S_2[j]) = rp(\varphi(S_1)[i], \varphi(S_2)[j])$$

In human words, if $\mathcal{S}$ and $\mathcal{T}$ each have n rectangles, we call them similar if we can number the rectangles in each set 1 through n in such a way that the relative positions of all their pairs of corners are the same in both cases.

 licensed under CC BY-SA 4.0 int'l

Problem specification

Given n, find the maximum number of sets of n rectangles such that no two sets are similar.

(Note that the order of rectangles in a set does not matter. Also note that sets cannot contain duplicates, hence the n rectangles in each set must be distinct.)

Input specification

There is no input.

Output specification

Let c_n be the number of fundamentally different sets of n rectangles, and let $m_n = c_n \bmod (10^9 + 7)$.
For the **easy subproblem C1** submit an output file with 4 lines, containing m_1 through m_4.
For the **hard subproblem C2** submit an output file with 5000 lines, containing m_1 through m_{5000}.

Example

output for C1

```
1
84
some_bigger_number
an_even_bigger_number
```

(Un)helpful note

Don't bother looking for this sequence in the Online Encyclopedia of Integer Sequences. It's not there :)

 licensed under CC BY-SA 4.0 int'l

Task authors

Problemsetter:	Michal “mišof” Forišek
Task preparation:	Michal “mišof” Forišek, Michal “majk” Švagerka
Quality assurance:	Michal “mišof” Forišek, Michal “majk” Švagerka

Solution

Coordinate compression

Coordinate compression is a useful tool in many problems in computational geometry. This is a simple trick that can be used whenever we have a task in which only relative positions of objects matter but we don’t care about their exact coordinates.

Coordinate compression is performed as follows:

- Collect the x-coordinates of all objects. (E.g., all corners of all rectangles.)
- Find the number d of distinct values in your collection.
- Change the x-coordinates of all objects to the values $0, 1, \ldots, d-1$ in a way that preserves the relative order. (The smallest original x-coordinate becomes 0, the second smallest is changed to 1, and so on.)
- Then, do the same with y-coordinates of all objects.

For example, if the old x-coordinates of your objects were 10, 3, 10, and 47, their new x-coordinates are 1, 0, 1, and 2.

Easy subproblem

Why did we mention coordinate compression? For two reasons.

First, similarity of two sets of rectangles can obviously be defined using coordinate compression: two sets are similar if and only if they are exactly the same after coordinate compression.

Second, we can use this approach to enumerate all distinct arrangements.

Each of our n rectangles corresponds to two x-coordinates and two y-coordinates. Hence, in each direction there are at most $2n$ distinct coordinates. Which means that after coordinate compression all coordinates have to be between 0 and $2n-1$, inclusive.

And for $n = 4$ this gives us a solution that’s fast enough: Generate all possible n-tuples of rectangles on the $2n \times 2n$ grid, and count those that do not change under coordinate compression.

In order to both speed up the generation and to make sure we only generate each set once, imagine that each rectangle is a 4-tuple of coordinates, and generate the rectangles so that these 4-tuples are in lexicographic order.

For $n = 4$ there are $28^2 = 784$ ways to draw a rectangle onto an 8×8 grid, and therefore $(784 \cdot 783 \cdot 782 \cdot 781)/4! \approx 15.6 \times 10^9$ sets of four such rectangles. This is already a reasonably small number, and with a few simple optimisations it is possible to generate all distinct sets for $n = 4$ in under a minute.

Hard subproblem

With n up to 5000, we almost certainly need a polynomial-time solution.

Let’s start by considering a slightly different problem. As in the original, we are drawing n rectangles, but there are two changes. First, each rectangle has a different color (i.e., their order matters). Second, rectangles are now allowed to be identical.

The solution to our original problem will consist of two steps. First we solve the (much easier) new problem we just stated, then we show how to use its result to compute the answer we seek.

 licensed under CC BY-SA 4.0 int’l

New problem, one-dimensional version

Let's first solve the new problem in one dimension: what is the number of fundamentally different sets of n colored intervals on a line?

This can be done using dynamic programming. Let $dp[n][k]$ be the number of arrangements of n intervals such that their endpoints have exactly k different coordinates. We can now start from $dp[1][2] = 1$ and fill the table row by row. For each endpoint of a new segment we either select an endpoint used by any of the already placed segments, or we create a new one.

This intermediate result has an OEIS entry.

New problem, two-dimensional version

To extend this result to two dimensions, we just observe that the two dimensions are completely independent: any interval on the x axis can be paired with any interval on the y axis to get a rectangle. Thus, the number of arrangements of n colored rectangles is simply the square of the number of arrangements of n colored intervals.

Original problem

Now we need to deal with the fact that some of the rectangles may coincide. Let's assume that we already computed the number of arrangments of i distinct labeled rectangles for all $i < n$. Then the number of arrangements of n distinct labeled rectangles is:

$$\text{distinct-labeled}(n) = \text{labeled}(n) - \sum_{i=1}^{n-1} \text{distinct-labeled}(i) \cdot S_2[n,i]$$

where $S_2[n,i]$ is the Stirling number of the second kind, or also the number of partitions of set of size n to i non-empty subsets. Removing the labels is now trivial

$$\text{distinct-unlabeled}(n) = \frac{\text{distinct-labeled}(n)}{n!}$$

In total, we need $O(n^2)$ to calculate the answer for all i between 1 and n.

Note

One can use the inversion formula to express the above quantity using Stirling numbers of the *first* kind:

$$\text{distinct-labeled}(n) = \sum_{i=1}^{n} (-1)^{n-i} \,\text{labeled}(i) \cdot S_1[n,i]$$

 licensed under CC BY-SA 4.0 int'l

Problem D: Delightful

You have found a weird box in your grandma's basement. A careful inspection revealed that it is an old-school computer. You tried to run some simple programs but they all behaved very weirdly. It took you some time before you were able to pinpoint the reason: the computer *uses ternary logic and arithmetic!*

A *trit* is the ternary equivalent of a bit. Our trits can store the values 0, 1, 2. When doing logical operations with trits we imagine that 0 represents false, 1 represents unknown, and 2 represents true.

The computer has 26 registers, labeled `A` through `Z`. Each register is a **40-trit** unsigned integer variable.

Operators

The computer supports the standard arithmetic operators: `+`, `-`, `*`, `/` (integer division), and `%` (modulo). All computations are done modulo 3^{40}. Division and modulo by zero cause a runtime error.

The computer also supports the following tritwise operators:

- `&` (and) returns false if either operand is false, true if both are true, and unknown otherwise
- `|` (or) returns true if either operand is true, false if both are false, and unknown otherwise
- `^` (xor) is addition (without carry) modulo 3
- `~` (unary not) is subtraction from 2
- `<<` (shift left) and `>>` (shift right) behave as expected, new trits are set to 0

Examples (with leading zeros omitted):

- 210_3 `&` $111_3 = 110_3$
- 210_3 `|` $111_3 = 211_3$
- 210_3 `^` $111_3 = 021_3$
- `~` $210_3 = 222\ldots222012_3$ (i.e., 37 twos followed by 012)
- 210_3 `<<` $2 = 21000_3$
- 210_3 `>>` $2 = 2_3$.

Programs

You suspect that there are some instructions for loops and jumps and such, but so far you weren't able to discover any. Thus, in this problem you will have to do without such fancy tools. A program is therefore a finite sequence of commands. Each command will be executed exactly once, in the given order. Each command has one of three forms:

- `[register] = [operand]`
- `[register] = [operand] [operator-with-two-operands] [operand]`
- `[register] = [operator-with-one-operand] [operand]`

Here, operators are the ones listed above and each `[operand]` is either a register or an integer constant that fits into a register.

Constants can be given either in base-10 or in base-3. Base-10 constants have no prefix, base-3 constants have a prefix "`0t`" (zero, lowercase t). For example, `47` and `0t001202` represent the same number.

At least one space between each two tokens is mandatory.

 licensed under CC BY-SA 4.0 int'l

Example

At the beginning of our program's execution, the register X contains the input number and all other registers contain zeros. We want to find the last non-zero digit of X (if any) and set it to zero.

One possible program that does this looks as follows:

```
A = X - 1
B = ~ A
C = B ^ X
X = X & C
```

The explanation is left as an exercise for the reader.

Subproblem D1

Write a program that computes *ndp*: the length of the longest non-decreasing prefix of a number. For example, for any number x that has the form $000122012\ldots_3$ we have $ndp(x) = 6$ because the first six trits of x are in sorted order but the first seven aren't.

Formally, let the ternary representation of x be $x_{39}x_{38}\ldots x_0$. We define $ndp(x)$ as the largest i such that $x_{39} \leq x_{38} \leq \cdots \leq x_{39-i+1}$.

At the beginning of your program's execution, the register X contains the input number and all other registers contain zeros. At the end of your program's execution the register Y must contain the value $ndp(\texttt{X})$, all other registers may contain any value.

Your program must contain **at most 100 commands**.

Subproblem D2

At the beginning of your program's execution, the register X contains the input number and all other registers contain zeros. It is guaranteed that X is between 0 and 10^9, inclusive.

Write a program that will set Y to 1 if X is prime, and leave it at 0 if it isn't. (Recall that the numbers 0 and 1 are not prime.)

Your program must contain **at most 5,000 commands**.

Input specification

There is no input.

Output specification

Your output file must contain a program that meets the above specifications and solves the given subproblem.

Note that empty lines and extra whitespace on each line are ignored, feel free to use those to format your program for better human readability. Only non-empty lines are counted towards the limit on the number of instructions. Remember that at there must be at least one space between any two tokens in your program, and at least one newline between any two commands.

Testing

A very basic interpreter is provided on a best effort basis on an external site. The interpreter will start in a state with zeros in all registers, execute up to 5000 commands of the program you provide, and report either an error that occurred, or the number of commands executed and final values in all registers.

 licensed under CC BY-SA 4.0 int'l

Task authors

Problemsetter:	Michal “mišof” Forišek
Task preparation:	Michal “mišof” Forišek
Quality assurance:	Michal “majk” Švagerka

Solution

The title of this problem is a homage to a truly delightful book on bitwise magic: Hacker’s Delight.

Subproblem D1: sorted prefix

We will show our solution in steps and illustrate it on one example.

Imagine `X` and `X >> 1` written one below another:

```
X          : 01121 20000 00000 00000 00000 00200 00000 00000
A = X >> 1: 00112 12000 00000 00000 00000 00020 00000 00000
```

Each column now contains two consecutive trits of `X`. We are looking for the first column in which `X >> 1` is bigger (if such a column exists). How can we find it? Take the logical or, and look at the first column that does not match `X`.

(In general, note that even though the statement calls `&` “and” and `|` “or”, it is better to think of them as the “min” and “max” operators, as that is another description of what they do.)

Mismatches can be found using xor. In particular, `X ^ ~X` is 2 everywhere. Hence, `X ^ ~A` will be 2 where they match and 0 or 1 where they differ.

```
A = X | A : 01122 22000 00000 00000 00000 00220 00000 00000
A = ~A    : 21100 00222 22222 22222 22222 22002 22222 22222
A = X ^ A : 22221 20222 22222 22222 22222 22202 22222 22222
```

Now we would like to isolate the trits where `X` and `A` mismatched. The first step is obvious: negate `A`. Then, the mismatched positions are 1s and 2s while the matched ones are 0s.

In order to make our life easier, let’s now turn any potential 2s into 1s using another clever trick: note that `A ^ A` has the same non-zero trits, but with 1s and 2s swapped. Thus `A & (A ^ A)` will only have 0s and 1s. In particular, the leftmost 1 is the position where the first comparison failed, and the number of 0s to the left of that 1 is the answer we seek.

```
A = ~ A     : 00001 02000 00000 00000 00000 00020 00000 00000
B = A ^ A   : 00002 01000 00000 00000 00000 00010 00000 00000
A = A & B   : 00001 01000 00000 00000 00000 00010 00000 00000
              ^^^^
              answer = 4
```

Let’s now propagate the 1s towards the right by shifting `A` and or-ing it with itself:

```
B = A >> 1 : 00000 10100 00000 00000 00000 00001 00000 00000
A = A | B  : 00001 11100 00000 00000 00000 00011 00000 00000
```

Each original 1 are now two consecutive 1s. Let’s do the same again with step 2:

 licensed under CC BY-SA 4.0 int’l

```
B = A >> 2 : 00000 01111 00000 00000 00000 00000 11000 00000
A = A | B  : 00001 11111 00000 00000 00000 00011 11000 00000
```

Now we do the same with steps of size 4, 8, 16, and 32, yielding:

```
A          : 00001 11111 11111 11111 11111 11111 11111 11111
```

Now let's "binary-negate" A, for example by subtracting it from the constant $111\dots111_3$.

```
A          : 11110 00000 00000 00000 00000 00000 00000 00000
```

All that remains is to count the 1s in this `A`. As with the propagation of 1s, we will do the population count in a logarithmic number of iterations. In the first iteration we will make groups of two trits, then merge those into groups of four, eight, sixteen, thirty-two, and finally forty.

We will only show the first iteration here: Let `B = A &` $020202\dots_3$, and let `C = (A >> 1) &` $020202\dots_3$. That is, `B` are trits of `A` at even positions and `C` are trits of `A` at odd positions, shifted one to the right. Once we have these two, set `A = B + C`.

```
B          : 01010 00000 00000 00000 00000 00000 00000 00000
C          : 01010 00000 00000 00000 00000 00000 00000 00000
A          : 02020 00000 00000 00000 00000 00000 00000 00000
```

After this iteration, if we consider each two-trit block to be a separate number, those numbers are precisely the numbers of 1s that started in that block.

After the next iteration we would have

```
A          : 00110 00000 00000 00000 00000 00000 00000 00000
```

Note that the `0011` (created by adding `0002` and `0002`) is the ternary representation of the number 4.

Subproblem D2: primality testing

Wait, what? Primality testing?

With only 4,000 instructions allowed, 3401 primes up to $\sqrt{10^9}$, and the need to use at least two instructions per prime ("compute the remainder" and "do something with the remainder"), trial division is probably out of the question. And trying out just a subset of those primes is not worth the effort – we'll miss so many that random tests will almost certainly find a test case we'll solve incorrectly.

So it seems that we need a smarter algorithm. But how? We don't even have an "if" statement, how on Earth should we implement anything non-trivial? Well, it's painful but definitely possible. For example, you can essentially emulate an "if" by evaluating the condition, doing the computation anyway, and multiplying its effects by the value of the condition – so that if the condition was false, you multiply each change by zero and in effect nothing happens.

Signum

Let's start by designing a helper function that returns 0 for 0 and 1 for anything else. This is quite easy to do in constant time: $1 - ((x-1)/(3^{40}-1))$. The constant is the largest possible value. If x is zero, $x-1$ overflows to that value, otherwise it is some smaller value. Thus, the result of division is 1 iff x was zero.

(Alternately, it's easy to construct signum that runs in logarithmic time by propagating set trits similarly to what we did in the easy subproblem.)

 licensed under CC BY-SA 4.0 int'l

Test for equality

In order to test whether two numbers are equal, we just subtract one from the other and use signum to test whether the result is zero.

A suitable primality test

The test we picked for our implementation is a deterministic version of the Miller-Rabin test. In particular, a result from 2013 when Izykowski and Panasiuk showed that $n < 1,050,535,501$ is a prime if it is a strong pseudoprime in bases `336781006125` and `9639812373923155`. Thus, all we need to do are two iterations of the Miller-Rabin test.

Here is a short Python implementation of the test we are going to implement:

```
def is_SPRP(n,a):
    d, s, c, a = n-1, 0, 1, a%n
    if a == 0:
        return True
    while d % 2 == 0:
        d, s = d//2, s+1
    while d:
        if d % 2: c = (c * a) % n
        a, d = (a * a) % n, d // 2
    if c == 1: return True
    for r in range(s):
        if c == n-1: return True
        c = (c * c) % n
    return False

def is_n_leq_billion_prime(n):
    return n > 1 and is_SPRP(n,336781006125) and is_SPRP(n,9639812373923155)
```

Of course, we don't have conditionals and fancy cycles with a variable number of repetitions. Luckily, each of the cycles in the function makes at most $\log_2 n$ steps, so we can just do a fixed number of 30 iterations. (And those will be 30 physical copies of the same sequence of instructions.) Hence, our implementation of the test will look more like this:

```
    if n == 0:
        set n to 1 so that we don't get bitten by a%n
    d, s, c, a = n-1, 0, 1, a%n
    if a == 0:
        return True
    repeat 30 times:
        if d is even: d, s = d//2, s+1
    repeat 30 times:
        if d is odd: c = (c * a) % n
        a, d = (a * a) % n, d // 2
    if c == 1: return True
    repeat 30 times:
        if s is positive:
            if c == n-1: return True
```

 licensed under CC BY-SA 4.0 int'l

```
            decrement s
        c = (c * c) % n
    return False
```

Below are the implementation details for the body of each of the three cycles. First, finding the largest power of two that divides $n-1$. This was actually quite simple. Take $d \bmod 2$ and use that value both to increment or not increment s and to divide or not divide d by 2.

```
B = D % 2
S = S + 1
S = S - B
B = 2 - B
D = D / B
```

Next, computing $c = (a^d) \bmod n$. Again, let $b = d \bmod 2$. If $b = 1$, we want to multiply c by a, and if $b = 0$, we want to multiply it by 1. In order to do this, we will simply multiply c by $(1 + b * (a-1))$. The rest of this cycle is trivial, so here we'll just print the implementation of the part we just explained.

```
B = D % 2
M = A - 1
M = M * B
M = M + 1
C = C * M
C = C % N
```

Finally, we need to check whether some suitable $c^(2^r)$ gives remainder -1 modulo n. This part doesn't use any new tricks: we just use signum both to check whether s is already zero (in which case we make sure to do no changes) and to check whether the current c equals $n-1$.

Putting it all together, we get an implementation with a bit fewer than 1500 commands. Not bad, for a primality test that correctly works all the way up to 10^9.

If you solved this task, did you use a different approach? We would love to hear from you! The submits are a bit hard to read, if you know what I mean ;)

 licensed under CC BY-SA 4.0 int'l

Problem E: Encrypted romance

Alice and Bob are two young lovers who frequently send each other hearts and kisses over the Internet. However, they (rightfully) suspect that Eve might be interested in hearing all their private messages. Alice and Bob are both untouched by modern cryptography, so they have designed their own encryption scheme. As you probably already expect, it's not that good.

In order to be "extra secure", Alice and Bob have chosen n different symmetric ciphers. Whenever one of them wants to send a message to the other, he or she splits it into n pieces (called blocks) and encrypts each block using a different cipher.

Of course, they need some secret keys in order to do all that encryption and decryption. On the last day of summer Alice and Bob picked one shared secret key k. The key k was then used to compute the keys k_i used for the individual ciphers.

Eve has intercepted a message Alice sent to Bob. Eve already knows all ciphers they use, and also the way in which they compute the keys k_i from the original key k. The only information she's missing is the key k itself.

Estimate Eve's chances of successfully decrypting at least one block of the message!

Problem specification

The secret key is an integer k between 0 and $p-1$, inclusive, where p is a prime number.

The blocks of each message are numbered 1 through n. The key Alice and Bob use to encrypt and decrypt block i is denoted k_i. The key k_i was computed from k using the following formula:

$$k_i = ((a_i \cdot k + b_i) \bmod p) \bmod m_i$$

Luckily for our two lovers, Eve is also woefully bad at cryptography. When everything she tried failed, she decided that she will simply try finding the right k using brute force.

You are given all parameters of the encryption scheme, **including** the secret key k that Eve does not know. Estimate the number of keys from the key space that allow Eve to decrypt at least one block of the message.

Formally, you are asked for the number of integers $k' \in [0, p)$ that have the following property: For at least one $i \in \{1, 2, \ldots, n\}$ we have $k_i = ((a_i \cdot k' + b_i) \bmod p) \bmod m_i$. (In words, if Eve uses the key k' instead of the correct key k, she will correctly decrypt at least one block of the message.)

Since the number of such keys may be very large, we are only interested in a rough approximation of the correct number. (Refer to the output specification.)

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 begins with a line consisting of three integers p, k and n, where

- p is the prime number specifying the size of the keyspace.
- k is the secret key selected by Alice and Bob.
- n is the number of blocks in a message.

 licensed under CC BY-SA 4.0 int'l

Then n lines follow, each describing the key generation scheme for one block of the plaintext. Each line contains three space-separated integers a_i, b_i, m_i that are used in the formula to compute the key k_i.

In both subproblems we have $1 \le n \le 100$ and $0 \le k < p$.
In both subproblems we also have $\forall i \in \{1, 2, \ldots, n\}$: $1 \le a_i < p$, $0 \le b_i < p$, and $1 \le m_i \le p$.

In the **easy subproblem E1** we have $t = 400$, $p \le 10^9$ and $\forall i \in \{1, 2, \ldots, n\}$: $m_i \le 1000$.
In the **hard subproblem E2** we have $t = 100$ and $p \le 10^{100}$.

Output specification

For each test case, output a single line with a floating-point number m: the amount of keys k' from the key space that share at least one block encryption key with the actual key k.

Make sure that your output contains at least four **most significant decimal digits** of m (but preferably print more). The value of m may be printed in scientific notation (“`1.2345678E-90`”).

In the **easy subproblem E1** your answer will be accepted if the **relative** error is at most 10^{-1}.
In the **hard subproblem E2** your answer will be accepted if the **relative** error is at most 10^{-2}.

Example

input

```
4

13 10 2
2 0 5
3 1 6

19 18 2
3 4 8
2 2 6

1632899 12351 5
12453 232 789
73829 112 650
11234 893 998
788920 128492 706
123892 99109 551

5228506301 5128935514 5
516969294 4322317375 63664
65456316 100106652 423344
121168466 121978447 442230
616682668 556300002 185418
1631488462 1712350840 628094
```

output

```
4
5
11461
1.42e5
```

In the first sample, we have:

- $k_1 = ((2 \cdot k + 0) \bmod 13) \bmod 5 = 2$
- $k_2 = ((3 \cdot k + 1) \bmod 13) \bmod 6 = 5$.

The set of useful keys is $\{1, 6, 10, 12\}$, as for $k' \in \{1, 6\}$ we correctly decrypt the first block and for $k' = 12$ we correctly decrypt the second block.

In the second sample, the key $k = 18$ leads to $k_1 = 1$ and $k_2 = 0$. There are five good keys: 2, 5, 8, 17, 18. Even though with $k' = 8$ Eve deciphers both blocks correctly, the value 8 is only counted once.

In the third case, any answer in the range $[10315, 12607]$ would be accepted for the easy subproblem. For the hard subproblem, only values within the range $[11347, 11575]$ would be considered correct.

The last sample cannot appear in the easy subproblem, as both $p > 10^9$ and $m_i > 1000$. The correct answer is 142817.

 licensed under CC BY-SA 4.0 int’l

Task authors

Problemsetter:	Michal "majk" Švagerka
Task preparation:	Michal "majk" Švagerka
Quality assurance:	Michal "mišof" Forišek

Solution

We start by reformulating the story. We have n hash tables, the i-th of which has m_i buckets, and we use them to store integers in the range $[0, p)$. The task is to find out how many integers $k' \in [0, p)$ are in collision with k in at least one of the hash tables. Note that the standard term collision would assume that $k \neq k'$, which is not the case in our problem, but this change would only subtract 1 from the answer.

The hash tables described in the statement are selected from a family of universal hash functions. For these it holds that the number of collisions with k in the i-th hash table is $\lfloor \frac{p}{m_i} \rfloor$ or $\lceil \frac{p}{m_i} \rceil$. In other words, the fraction of integers in collision with k is roughly $\frac{1}{m_i}$. It would be tempting to say that the answer is

$$p \cdot \prod_{i=1}^{n} \left(1 - \frac{1}{m_i}\right) .$$

This would be only true if we selected the parameters of the hash functions independently at random, and we don't know whether this is the case. A quick inspection of the input files tells us it's actually the opposite! We need a different approach.

The error tolerance is quite high, which might suggest a probabilistic approach. We can sample integer k' from the range $[0, p)$ and check whether it is in collision with k. If we perform enough experiments, the ratio of the observed collisions to all samples converges towards the ratio of actual collisions within the population of integers. Roughly 10^5 samples for each test case are sufficient to pass the easy subtask.

In the hard subtask, things are a bit more complicated. Firstly, we obviously need to use large integers. If we overcome this hurdle and run the code for quite some time on the hard subtask, we realise that something is not quite right – the output for the majority of test cases seems to be 0. This is definitely not a correct answer, as there is always at least one collision (namely k itself). Is our approach flawed? Or should we simply use more samples?

It turns out that the answer to both questions is yes. The approach is valid (after all it worked on the easy subtask) – the expected number of collisions that we see is proportional to the actual number of collisions, i.e., the quantity we compute is an unbiased estimator of the answer. The problem is, however, with its concentration! Let $p \approx 10^{100}$ and the actual number of collisions be $m \approx 10^{50}$. Let's be generous and sample 10^{10} integers. How many collisions are we expected to see? About 10^{-40}. That's why we don't see any!

The Chernoff bound tells us that the probability that the sum of N independent Bernoulli random variables with probability p is not within $1 \pm \delta$ approximation of its expectation is at most:

$$\Pr\left[|X - \mu| \geq \delta\mu\right] \leq 2e^{-\frac{\delta^2\mu}{3}}$$

To get a reasonable success probability for our amount of test cases, we need at least $\delta^2\mu \geq 20$. Note that $\mu = N \cdot p$, where N is the number of samples. In the case described above, we have $\delta^2 \cdot N \cdot p = 9 \cdot 10^{-44} \cdot N$. Thus, we would need more than 10^{40} experiments, which is clearly way too much. Since δ is fixed, we have to increase p somehow. We can use the following strategy.

 licensed under CC BY-SA 4.0 int'l

Imagine a huge table, where each row corresponds to a hash table and each column is an integer between 0 and $p-1$. In the cell (i, j) we write Y if j is in collision with k in the i-th hash table, and N otherwise. The answer we are looking for is the number of columns with at least one Y. As we know, these might be very rare.

Note that some columns may contain more than one Y. In particular, the column corresponding to the integer k contains n Ys. Now comes the simple yet subtle observation – the number of columns with at least one Y equals the number of top-most Ys (i.e. that have no Ys above them in their respective column).

Given a cell (i, j), we can easily determine whether it is a top-most Y – we just check whether there is a hash collision of j with k in a hash table with lower index than i.

This gives us a new avenue of attack for this problem. We know the total number of Y cells, because we can exactly compute the number of collisions for each hash table. If we find a way to uniformly sample them, we can look at each sampled cell and verify whether it's a top-most Y cell. This will then give us an unbiased estimator of the total number of top-most Y cells, which is precisely the answer we seek.

This leaves us with just one question: how can we uniformly sample all Y cells? This is easy: All collisions in the table i are of form $a_i^{-1} \cdot (x + t \cdot m_i - b_i)$, where x is the bucket to which k belongs in this hash table, a_i^{-1} is the multiplicative inverse of a_i modulo p (recall that p is prime) and t is an integer between 0 and (the number of collisions in this particular hash table, minus 1).

Finally, the probability that a randomly generated Y cell is a top-most one is at least $\frac{1}{n}$, because the table has only n rows. Coming back to our Chernoff bound, we have $\delta^2 \cdot N \cdot \frac{1}{n} \geq 9 \cdot 10^{-6} \cdot N$, so roughly 10^6 samples should give us a sufficient precision.

 licensed under CC BY-SA 4.0 int'l

Problem F: Farmer’s code

Bob is a farmer. Among other endeavours, he grows apple trees. Not all of his trees are perfect, though. Some produce sour fruit, some are too small, and some are just crooked. His dream is a perfectly symmetrical tree with extremely sweet apples. However, he isn’t very fluent in genetics, so he needs your help!

Problem specification

In this problem, you will interact with an online farm dashboard page. You can find the link in the online version of this problem statement.

You have a population of n trees, sorted from oldest to youngest. In the beginning all trees have distinct genotypes (their genetic code) and phenotypes (their appearance) but this may change based on your actions. You are also given an image of your goal – the tree that Bob wants to grow.

You may select any two living trees to cross them. When you do this, the following happens:

- The genetic code of these two trees is crossed and a new tree grows. The new tree is considered the youngest.
- The oldest tree of the whole population dies. As there is one addition and one removal, the size of the population stays the same.
- If you create the tree that is your goal, you will win and the subproblem will be marked as **OK**.

You will see the same set of trees if you connect from two different browser windows.

You can use the *Reset* button if you think that your genetic diversity became insufficient to produce the requested phenotype, or you simply want to start over. The farm will go back to the initial population and you’ll receive a **Wrong answer**. You can reset as often as you want – you are not limited to 10 submissions per subproblem.

However, you can only cross trees a limited number of times. The limit is separate for each subproblem and is displayed on the farm page. The number **will not reset** when you click *Reset*. If you run out, you won’t be able to solve that subproblem.

Input specification

In the **easy subproblem F1** we have $n = 4$.

In the **hard subproblem F2** we have $n = 18$.

Output specification

You don’t have to submit anything. You will automatically get **OK** or **Wrong answer** submissions by interacting with the farm page.

Word of advice

Knowledge of real-life tree genetics probably *won’t* help you solve this problem.

 licensed under CC BY-SA 4.0 int’l

Task authors

Problemsetter:	Michal “majk” Švagerka
Task preparation:	Michal “majk” Švagerka
Quality assurance:	Michal “mišof” Forišek

Solution

In the easy subtask the population is very small, so you should have been able to find it using a sufficiently exhaustive search (and possibly even with some random clicking).

One possible solution: Crossing the first and the second tree gives us a tree with the correct shape, but with the fruits in incorrect branches. The same goes for third and fourth three. Combining those two symmetrical trees yields the requested tree.

In the hard subtask, things are much more involved. Given the query limit, it seems unlikely that we will be able to win by accident, so we need to deduce the representation of the genetic code, the way how the crossover works and how is the genotype mapped to phenotype. In the following text, $\oplus$ will denote the crossover operation.

We soon realise that the crossover is deterministic, i.e. $A \oplus B$ always yields the same tree. Furthemore, the operation is commutative, i.e., the order of the trees doesn't matter. Additionally, the operation is associative – we can easily verify that $A \oplus (B \oplus C) = (A \oplus B) \oplus C$.

What happens if we do $A \oplus A$? (We have to select two distinct trees to cross, but we can always create two copies of the same tree and then combine those if we want to experiment.) For each A we will get the same result: let's call it tree X. It is then easy to verify that X has another interesting property: we have $A \oplus X = A$ for all A.

Observe that we have just verified all axioms of a commutative group and hence we have established that the crossover is a group operation. The order of the group is 2. What groups of order do we know? Perhaps we are dealing with $\mathbb{Z}_{2^k}$ with $\oplus$ being xor?

How many different trees are there in the first place? If we count correctly, each tree has six different fruits, one trunk and one node without any fruit. The tree is rooted, but always at the trunk. By Cayley's formula we know that there are exactly n^{n-2} distinct labelled trees on n vertices. In this problem, this would suggest that there are 2^{18} of them. Could it be just a coincidence that we have exactly 18 trees to play with? They would make a great basis of our group!

How does one prove Cayley's formula? One famous approach is using so called Prufer codes, where each code is a $(n-2)$-tuple of vertex labels. The proof gives a bijection to labelled trees. If we label our vertices from 0 to 7, those Prufer codes become closed on element-wise xor operation!

How to find the labels? Easily: simply try all different permutations of labels and check whether they are consistent with the crossover operations we observe. Given this information, how do we construct the tree we want? This is also simple – we can either use Gaussian elimination or simply try all possible subsets of the input to see which one yields the target tree. There is only one such subset, as the starting trees are indeed linearly independent. The 1-based indices of the trees for the solution are $(1, 2, 5, 6, 8, 9, 13, 14, 16)$.

 licensed under CC BY-SA 4.0 int'l

Problem G: Git gud

We did it! We finally created a fully-functional time machine. It can only send information back and forth, but that's more than enough if you know how to take advantage of it.

Obviously, the first thing we did was to establish communications with ourselves 50 years in the future, so that future-us can send present-us a copy of all the IPSC problems they made and we won't have to do any more work. They immediately agreed and sent us 50 years' worth of problems. And what's even better, the data was not in some crazy future file format that hasn't been invented yet. It was just a Git repository.

But when we tried to open it and find out what's inside, disaster struck. Our computer crashed and the time machine got disconnected. Please help us!

Problem specification

You are given a valid Git repository. Print a checksum of the list of files it contains.

Input specification

The input is a Git repository. Every file and directory in it has a name consisting only of lowercase English letters, numbers, dashes ("`-`") and periods ("`.`"). The repository can be downloaded with this command:

```
git clone https://ipsc.ksp.sk/2018/real/problems/g.git
```

The repository has three branches: `example`, `easy` and `hard`. When you clone it, Git will rename them to `origin/example`, `origin/easy` and `origin/hard` in your copy.

Instead of getting the repository from `ipsc.ksp.sk`, you can use `g_local.py` from the downloadable archive. This script will start a local server containing the same repository. Use it as follows:

```
python3 g_local.py                        # (in one window)
git clone http://localhost:8000/g.git     # (in another window while g_local.py is running)
```

Output specification

Submit a file with a single number: the checksum of the file list, as defined below.

The file list contains one line for each file in the repository. Each file is printed with its full path, using "`/`" (ASCII code 47) as the directory separator. The lines of the file list are sorted lexicographically and each line is followed by a UNIX newline (ASCII code 10). There is no extra whitespace.

The checksum is produced as follows. Interpret the file list as a sequence of ASCII codes. Multiply each of these ASCII codes by its 1-based position in the sequence. The checksum is their sum modulo $10^9 + 7$. For example, the checksum of "`Cat`⟨*newline*⟩" is $(1 \cdot 67 + 2 \cdot 97 + 3 \cdot 116 + 4 \cdot 10) \bmod (10^9 + 7) = 649$.

Example

sorted file list for the "example" branch

```
12/r
12/r.h
3
ll
lll/v.txt
lll/v/-
lll/v0
llll
```

output

```
87437
```

 licensed under CC BY-SA 4.0 int'l

Task authors

Problemsetter:	Tomi Belan
Task preparation:	Tomi Belan
Quality assurance:	TODO

Solution

Let's look at how Git stores information in its "object database". You can read the full details in various places such as the Git Community Book or the Git Book, but here is a quick summary.

A Git repository is a key-value data store containing immutable objects. Every object has a hexadecimal ID and some content. Some objects can contain the IDs of other objects, forming edges in a directed acyclic graph. There are four types of objects:

- "blob" represents a regular file. It doesn't have any outgoing edges and just contains the file content.
- "tree" represents a directory. It contains a list of filenames and IDs of other blobs or trees.
- "commit" represents the state at a certain point in time. It points to a tree and zero or more parent commits. In our case, there are no parent commits.
- "tag" doesn't appear in this problem.

Objects are identified by a hash of their content, forming a ~~blockchain~~ ahem, a Merkle tree. This is important because it means that objects aren't identified by *where they are*, only by *what's inside*. As long as the content is the same, one object can be reused in multiple places. E.g. if two different directories contain the same file, the file content will be stored only once, and the two tree objects will point to the same blob object with different filenames.

This automatic deduplication is usually a good thing. It's the whole reason why Git doesn't make a full copy of the whole project when you change a single file. But it also means that a repository can be much bigger than it seems to be.

Easy subproblem

The `easy` branch contains about 100,000 files whose total size is over 450 GiB, not including filesystem overhead. If you try to switch to it with `git checkout`, you will probably run out of disk space or patience.

But as long as you're careful and use commands that don't read the file content, you should be fine. There are many ways to get the file list. In particular, `git ls-tree` can print it in exactly the right format.

- `git ls-tree -r --name-only origin/easy`
- `git rev-list --objects origin/easy`
- `git show --name-status origin/easy`

Hard subproblem

If you try to switch to the `hard` branch, `git checkout` won't even show you "0%" because it can't compute how many files there are. The `easy` branch had lots of duplicate files, but the graph was still mostly treelike. The `hard` branch is a true directed acyclic graph where almost every node has an indegree and outdegree larger than 1.

When the total number of files is exponential, we clearly need another approach. If it's not possible to enumerate every single file, we must stop relying on Git's own commands and look at the whole graph ourselves. From a bird's-eye view, we must:

1. Get a list of all tree objects.
2. Read the content of every tree object to get the adjacency list.
3. Analyze the graph and compute the answer.

We could do everything in a single program, but it can help to split it into a chain of programs and use the strengths of different programming languages.

Getting a list of all tree objects

First we list all objects:

- `git cat-file --batch-check --batch-all-objects` (in Git 2.6.0 or newer)
- `git verify-pack -v .git/objects/pack/*.idx`

Then we filter out everything that isn't a `tree`.

Reading every tree object

Git doesn't have a built-in way to read a list of tree objects. Here are three possible strategies.

- Loop over the IDs and run "`git ls-tree $id`" or "`git cat-file -p $id`" for each one.
- Print all the tree objects in binary format with `git cat-file --batch`, and parse its output. The binary format can be recognized from a hexdump, but one tricky part is that object IDs of tree entries are stored as 20-byte binary strings instead of 40-byte hexadecimal strings.
- Use a Git library such as `libgit2`.

The first strategy is slower than the other two because of the overhead of spawning so many processes. But developer time (installing a library or understanding the binary format of tree objects) is usually more expensive. You can work on other problems while it's running.

We tested the first strategy on different machines and operating systems. For some unexplained reason, the timing is very inconsistent. One machine can finish in 5-10 minutes and another needs half an hour. In that case, the other two strategies might be more worth it.

Analyzing the graph

Now we finally have a normal algorithmic problem. We have a list of nodes, and for every node, an ordered list of its outgoing edges and their names.

How do we compute the output when the total length of the file list is exponential? We must take advantage of the final checksum and compute it directly in one go. But at least we don't have to deal with sorting – Git already stores them in the correct sorted order, even taking care of the implied "`/`" after directory names.

In the original problem, a tree's contribution to the final output depends on its path from the root, and there can be multiple paths leading to the same tree. We can represent every tree as a function from a "path prefix" string to a "file list" string. In the example from the problem statement, they would be defined like this:

$$\begin{aligned}
f_1(p) &= p + \texttt{"r\textbackslash n"} + p + \texttt{"r.h\textbackslash n"} \\
f_2(p) &= p + \texttt{"-\textbackslash n"} \\
f_3(p) &= p + \texttt{"v.txt\textbackslash n"} + f_2(p + \texttt{"v/"}) + p + \texttt{"v0\textbackslash n"} \\
f_4(p) &= f_1(p + \texttt{"12/"}) + p + \texttt{"3\textbackslash n"} + p + \texttt{"ll\textbackslash n"} + f_3(p + \texttt{"lll/"}) + p + \texttt{"llll\textbackslash n"} \\
\text{result} &= f_4(\texttt{""}) = f_1(\texttt{"12/"}) + \texttt{"3\textbackslash n"} + \texttt{"ll\textbackslash n"} + f_3(\texttt{"lll/"}) + \texttt{"llll\textbackslash n"} = \ldots
\end{aligned}$$

 licensed under CC BY-SA 4.0 int'l

This works, but we need something faster. Something that would allow us to only process each tree node once. Instead of functions on strings, we want polynomials.

Let's describe every string constant x with three numbers l_x, s_x, c_x: its length, its sum, and its checksum (i.e. $\Sigma_{i=1}^{n} i \cdot x[i]$). Instead of describing each tree with a function $f_t(p)$ from string to string, we will have three functions $l_t(l_p, s_p, c_p), s_t(l_p, s_p, c_p), c_t(l_p, s_p, c_p)$ which can tell us the length, sum and checksum of the tree's file list given the length, sum and checksum of the tree's path prefix.

We can prove by induction that the functions will always be "simple" and possible to describe with a constant number of constants. For every tree t:

- $l_t(l_p, s_p, c_p) = a_l l_p + a_0$ where a_l, a_0 are some constants.
- $s_t(l_p, s_p, c_p) = a_s s_p + a_0$ where a_s, a_0 are some constants.
- $c_t(l_p, s_p, c_p) = a_l l_p + a_s s_p + a_{ls} l_p s_p + a_c c_p + a_0$ where $a_l, a_s, a_{ls}, a_c, a_0$ are some constants.

The length and sum are easy, but the checksum is more complex.

Concatenation of two constants. Given two strings x and y, the checksum of their concatenation $x + y$ is $c_x + c_y + l_x s_y$.

Prefix extension. Given a function $f(p)$ describing a tree given its prefix p, we want the function $g(q) = f(q + \texttt{"foo/"})$ describing a named tree entry pointing to that tree given a parent prefix q.

$$
\begin{aligned}
p &= q + e \\
l_p &= l_q + l_e \\
s_p &= s_q + s_e \\
c_p &= c_q + c_e + l_q s_e \\
c_g(l_q, s_q, c_q) &= c_f(l_p, s_p, c_p) \\
c_g(l_q, s_q, c_q) &= a_l l_p + a_s s_p + a_{ls} l_p s_p + a_c c_p + a_0 \\
c_g(l_q, s_q, c_q) &= a_l(l_q + l_e) + a_s(s_q + s_e) + a_{ls}(l_q + l_e)(s_q + s_e) + a_c(c_q + c_e + l_q s_e) + a_0 \\
c_g(l_q, s_q, c_q) &= a_l l_q + a_l l_e + a_s s_q + a_s s_e + a_{ls} l_q s_q + a_{ls} l_q s_e + a_{ls} l_e s_q + a_{ls} l_e s_e + a_c c_q + a_c c_e + a_c l_q s_e + a_0 \\
c_g(l_q, s_q, c_q) &= (a_l + a_{ls} s_e + a_c s_e) l_q + (a_s + a_{ls} l_e) s_q + (a_{ls}) l_q s_q + (a_c) c_q + (a_l l_e + a_s s_e + a_{ls} l_e s_e + a_c c_e + a_0)
\end{aligned}
$$

Concatenation of n functions. Given a tree object t with entries $1, 2, \ldots, n$ represented by functions $l_1, s_1, c_1, \ldots, l_n, s_n, c_n$ (which were created with prefix extension), the checksum $c_t(l_p, s_p, c_p)$ is:

$$
\begin{aligned}
c_t(l_p, s_p, c_p) = {} & c_1(l_p, s_p, c_p) \\
& + c_2(l_p, s_p, c_p) + l_1(l_p, s_p, c_p) \cdot s_2(l_p, s_p, c_p) \\
& + c_3(l_p, s_p, c_p) + (l_1(l_p, s_p, c_p) + l_2(l_p, s_p, c_p)) \cdot s_3(l_p, s_p, c_p) + \ldots
\end{aligned}
$$

Now we have everything we need to recursively compute the coefficients in the functions l_t, s_t, c_t for every tree object, ending with the root tree r, and get the final checksum with $c_r(0, 0, 0)$.

 licensed under CC BY-SA 4.0 int'l

Problem H: Hats of various colors

You and your fellow revolutionaries have been captured by the evil archduke, and now you're all waiting for execution in the archduke's dungeons. Fortunately, a thousand year old tradition states that all prisoners must be given a last chance to go free if they can solve a logic puzzle.

The warden explains what will happen:

- Every prisoner will receive a red or blue hat. The prisoners will be seated so that they can see everyone else's hat, but not their own hat. The prisoners must stay completely silent.
- When the warden shouts "Now!", all prisoners must immediately and simultaneously choose whether to raise their hand or not. Everyone will see everyone else, but they must still stay silent.
- The warden will (privately) ask every prisoner what is the color of their own hat. If all prisoners answer correctly, the archduke has to let them go. If anyone makes a mistake, they will all be executed.

In both subproblems, there are $N = 13$ prisoners. In the **easy subproblem H1**, the hats are of $C = 2$ different colors (red and blue). In the **hard subproblem H2**, there are $C = 4000$ available hat colors.

Problem specification

Write a program that describes the prisoners' strategy.

This problem is special. Usually, you can write a program in any programming language, and you only submit the computed output data, not your source code. This task is the exact opposite. You only submit source code – specifically, a program written in the Lua 5.3 language. (See below for an introduction.)

We will run N copies of your program and allow them to communicate according to the rules. If all N programs answer correctly, you win.

Your program can use these variables:

- `N`: the number of prisoners.
- `C`: the number of different colors.
- `myself`: your own ID – a number between 1 and N, inclusive.
- `colors`: an array of visible hat colors. For every i between 1 and N, `colors[i]` will be a number between 1 and C, inclusive, except for `colors[myself]` which will be `nil`.
- `raise`: a function which takes a single boolean argument (whether you raise your hand or not), and returns an array of N booleans (for each prisoner, whether they raised their hand). You must call `raise` exactly once.
- `answer`: initially unset. You must set it to a number between 1 and C, inclusive – the number you want to announce as the color of your hat.

Limits

These functions won't exist: `debug.debug`, `io.*`, `loadfile`, `math.random`, `math.randomseed`, `os.*`, `package.*`, `print`, `require`.

We will run 300 test cases. (In total, your program will run $300 \times N$ times.) The memory limit is 256 MB (for the sum of your N programs). The time limit is 10 seconds (for the sum of your $300 \times N$ runs).

Testing your program

If you'd like, you can test your solution with our testing program: `lua h-test.lua solution.lua` .

 licensed under CC BY-SA 4.0 int'l

Introduction to Lua

If you don't know the Lua language, here's a quick introduction:

- Variables: `foo_bar = a + b`
- Literals: `nil`, `true`, `false`, `123`, `0xFF`, `"a string"`
- Arithmetic: `1+2 == 3`, `1-2 == -1`, `2*3 == 6`, `3/2 == 1.5`, `3//2 == 1`, `7%5 == 2`, `2^3 == 8`
- Comparison: `a == b`, `a ~= b`, `a < b`, `a <= b`, `a > b`, `a >= b`
- Logic: `a and b`, `a or b`, `not a` (note that `nil` and `false` count as false, and everything else counts as true, including `0` and `""`)
- Math: `math.abs(x)`, `math.floor(x)`, `math.max(a, b, c, d)`, `math.pi`, etc.
- Binary: `a & b` (AND), `a | b` (OR), `a ~ b` (XOR), `a >> b` (right shift), `a << b` (left shift), `~a` (NOT)
- "If" statement: `if ??? then ... elseif ??? then ... else ... end`
- "While" loop: `while ??? do ... end`
- Numeric "for" loop: `for i = 1, 128 do ... end`
- Functions: `function some_name(a, b, c) local d = 7; return a + d; end`
- Local variables (in functions and control structures): `local l = 123`
- Arrays (tables): `arr[idx]` (indexed from 1), `{ 1, 2, 3 }` (new array), `#arr` (array length)
- Comments: `--[[ multiple lines ]]`, `-- until end of line`
- Newlines *and* semicolons are optional: `a = 1 b = 2 + 3 c = 4 * 5`

For more details, refer to the Lua manual and the language grammar, or other on-line resources – though you probably won't need most parts.

Example

```
my_bool = colors[1] == 1 or myself > 6

hands = raise(my_bool)

if hands[1] or not hands[2] then
  answer = 1
else
  answer = C
end
```

 licensed under CC BY-SA 4.0 int'l

Task authors

Problemsetter:	Tomi Belan
Task preparation:	Tomi Belan
Quality assurance:	Michal “majk” Švagerka, Adrián Goga

Solution

In the easy subproblem, one simple strategy is to seat everyone in a loop, and have everyone raise their hand if their right neighbor's color is red, and then say red if their left neighbor raised their hand and blue otherwise. While implementing it, remember that colors are 1 and 2, hands are `true` and `false`, and you can't mix them (Lua doesn't allow doing arithmetic on booleans, and `not 1` and `not 2` are both `false`).

In the hard subproblem, we need something almost optimal. Every prisoner receives $N-1$ bits of information – the raised hands of the other prisoners. (Seeing their hats doesn't tell you anything about your own hat.) So the maximum number of colors you can theoretically distinguish with is 2^{N-1}. To make that happen, everyone must make full use of their raised hand – it must give information to every other prisoner.

Let's convert every color into an $N-1$-bit binary number between 0 and $2^{N-1}-1$, inclusive. When we split those numbers into bits, we can think of the full state as a two-dimensional binary array of N rows (the prisoners) and $N-1$ columns (the individual bits of each prisoner's hat color). Every prisoner sees the whole array except for their own row.

The job of the first $N-1$ prisoners will be to tell everyone about the i-th column. They can count the XOR of all the visible bits in their column and send that to everyone else. When you see whether prisoner 1 raised their hand, and combine it with prisoner 1's own first bit, it will tell you the xor of the whole first column. XOR that with the rows your can see, and it will tell you whether your own first bit is 0 or 1.

The only remaining problem is how everyone can determine their own value in their own column. That's the only missing bit. The final N-th prisoner can send everyone the XOR of all the bits they can see, which allows everyone to compute the XOR of all bits in the whole table, and since they already know all the bits except one, that will tell them its value.

 licensed under CC BY-SA 4.0 int'l

Problem I: Incorrect expression

Number Man is one of the world's most powerful superheroes, thanks to his superhuman mathematical powers. With a single thought he can forecast market movements, factor large numbers, extrapolate object trajectories, compute discrete logarithms, estimate probabilities of catastrophic events, and even calculate his tax rate. His favorite hobbies are knitting and high-frequency trading.

But despite his amazing powers, Number Man is very frustrated right now. His ultra-accurate weather prediction formula started giving ultra-inaccurate results, and he spent the last few hours searching for the cause. Did he really make a mistake? He never makes mistakes! And why does everyone else keep giggling? His teammates probably pranked him and made a tiny change on the blackboard while he was distracted. But what symbol did they change? There are so many options.

Problem specification

First, let's recursively define what is a *valid expression*:

- A sequence of one or more digits is a valid expression. Leading zeros are allowed.
- If `A` is a valid expression, then `(A)` is a valid expression.
- If `A` and `B` are valid expressions, then `A+B`, `A-B` and `A*B` are valid expressions.
- Nothing else is a valid expression. No division, no unary minus, etc.

Number Man has a valid expression E. Every valid expression that has the exact same length as E and differs from E in exactly one character is a *potentially correct expression*. Note that E itself is **not** a potentially correct expression.

To calculate the *weather value* of a valid expression, calculate the expression's value normally (standard precedence rules apply) and then find the nonnegative remainder that value gives modulo $10^9 + 7$. E.g. the weather value of "`0-2*1`" is $10^9 + 5$.

Find the weather value of every potentially correct expression. How many different numbers did you get?

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 and consists of a single line containing the *incorrect but valid expression* E.

In the **easy subproblem I1**, the length of each expression is at most 10 000.

In the **hard subproblem I2**, the length of each expression is at most 1 000 000.

Output specification

For each test case, print a single number: the number of distinct weather values among all potentially correct expressions.

Example

input

```
2

0+0

5*6*7
```

output

```
19
49
```

 licensed under CC BY-SA 4.0 int'l

Task authors

Problemsetter:	Tomi Belan
Task preparation:	Tomi Belan
Quality assurance:	Michal “majk” Švagerka

Solution

There is no clever idea here, just blood, sweat and tears. First you must parse the expression and create a syntax tree. For every node, you must compute its current value, and find a $f(x) = ax + b$ function that can tell you how the whole expression's final value changes if you change the value of this node. Then you compute all the prefix sums, suffix sums, prefix products and suffix products. This will allow you to try every possible character replacement in constant time.

We will add more details to the booklet soon.

 licensed under CC BY-SA 4.0 int'l

Problem J: Jumping over walls

You have always been an avid fan of mazes, and a bit of a cheater. No wonder you came up with a peculiar device that can be used to cheat in a maze. You will simply select a direction and the device will help you jump over all walls in your way, all the way to the nearest empty space in the chosen direction.

You have built two prototypes of the device. Prototype J1 only allows you to choose the 4 cardinal directions for jumping. Prototype J2 also lets you jump diagonally, so you have 8 available directions for each jump.

You would like to start manufacturing more of these devices – many maze fans would love to have one! But before you sell them, you need to test each of them to make sure you don't damage your brand by selling defective pieces. You decided that you'll build a test maze for those tests.

Problem specification

You will be constructing two separate test mazes: one for the J1 device and one for the J2 device. The purpose of the test maze is to make sure that the device jumps correctly in all supported directions, so it must be impossible to solve the maze without using every direction at least once. Constructing mazes costs money, so you must make them as small as possible.

While in the maze, you will only move using the device, never by walking. But note that the device is also able to jump over zero walls – if the very next cell in that direction is empty, you will just jump to that cell.

Each test maze must have the following properties:

- The maze is on a rectangular grid. The area of the rectangle must be as small as possible.
- Some cells are walls (denoted `#`), others are empty (denoted `.`).
- There is exactly one start cell (denoted `A`) and exactly one target cell (denoted `B`). Both are considered empty. They may be located anywhere in the maze.
- We assume that the maze is surrounded by an infinite grid of walls and jumping outside is impossible.
- It must be possible to travel from `A` to `B`.
- It must be impossible to travel from `A` to `B` without using each of the available directions at least once.

The **easy subproblem J1** is to construct a testing maze for the J1 prototype (4 directions), the **hard subproblem J2** is to do the same for the J2 prototype (8 directions).

Input specification

There is no input.

Output specification

The first line of your output should contain two integers: the number of rows r and the number of columns c in your maze. The value $r \times c$ must be as small as possible.

The rest of your output should contain the map of the maze: r lines, each with c characters as specified above.

 licensed under CC BY-SA 4.0 int'l

Example

output

```
5 6
......
.#.#.B
###...
#A#...
#.#...
```

This is a syntactically correct output but it doesn't describe a good test maze. This is because you could solve this maze without ever making a jump down. (For example, you can make three jumps right followed by two jumps up.)

Note that if you had the J2 prototype and started this maze at `A`, you would have exactly four possibilities for your first jump:

- Select the direction "down" and jump by 1.
- Select the direction "up" and jump by 3.
- Select the direction "right" and jump by 2.
- Select the direction "diagonally up and right" and jump by 3.

Note that you cannot choose a jump direction that doesn't lead to any empty cells.

 licensed under CC BY-SA 4.0 int'l

Task authors

Problemsetter: Michal "majk" Švagerka, Michal "mišof" Forišek
Task preparation: Basha Kováčová
Quality assurance: Michal "mišof" Forišek

Solution

Let's look at the problem from the opposite side. Suppose we already have a valid test maze. What then? Well, the maze must have a solution. Let's take one fixed solution that is as short as possible.

Now imagine that we turn all cells that are *not* visited by this particular solution into walls. What do we get? It's easy to realize that we will still have a valid test maze. This is because the original solution still solves the maze – the cells we jumped over in each step were already all walls, and the solution we kept does contain all directions.

This observation should give you a clue how to look for the smallest valid maze: instead of examining all mazes, you can just look for a single path in the above sense.

There were multiple ways that could get you from here to the solution. For example, you could make the educated guess that the optimal path will contain each direction *exactly* once. This makes it really easy to take a fixed size maze, try all such paths, and for each of them run a simple BFS to verify that there are no unwanted shorter solutions. This approach does indeed produce the optimal solutions to both subproblems.

Obviously, when working on this problem we had to be a bit more careful and actually verify that for smaller mazes there are no other solutions either. Still, with the above observation the code is almost the same. In the attached brute-force solution we do two things differently:

- we allow arbitrarily many steps in each direction
- already while generating the path we make sure that there are no shortcuts

Remember that if there is *any* valid maze for the given dimensions, there has to be a valid maze where the shortest solution visits all empty cells. And we can make this condition even more strict: out of all such mazes we are looking for the one where the number of empty cells (equivalently, the number of jumps in the solution) is as small as possible.

Thus, we only need to generate solutions that cannot be shortened. This means that as soon as the solution leaves a row/column/diagonal in J2, it cannot re-enter that line later – doing so would create a shortcut. Additionally, making multiple consecutive jumps in the same direction makes no sense either, as you can shorten the solution by merging them into one jump.

Hence, in our program, as soon as we enter a cell C_1, we try all possibilities for the next cell C_2, but before we recursively continue from C_2 we mark all other cells directly reachable from C_1 as banned. This makes the program fast enough to eliminate all smaller board sizes in a few seconds.

Example solutions

Here is the "lexicographically smallest" optimal solution for subproblem J1:

```
3 3
1##
#54
2#3
```

And this is the solution for subproblem J2:

 licensed under CC BY-SA 4.0 int'l

```
7 7
1######
2######
###5##6
#9#####
#####4#
####3##
#87####
```

For added clarity we used a slightly different format: the digits represent the order in which the empty cells have to be visited. (Thus, `1` is `A` and the biggest number is `B`.)

Proof of optimality

The above observations about the shortest solution can also be easily turned into a mathematical proof that the solutions shown above are optimal. E.g., in the J2 case note that any path has to start somewhere and it has to make at least six non-horizontal jumps (each leading into a new row) and at least six non-vertical jumps (each leading into a new column). Thus, you cannot fit such a path into a grid with fewer than 7 rows or 7 columns.

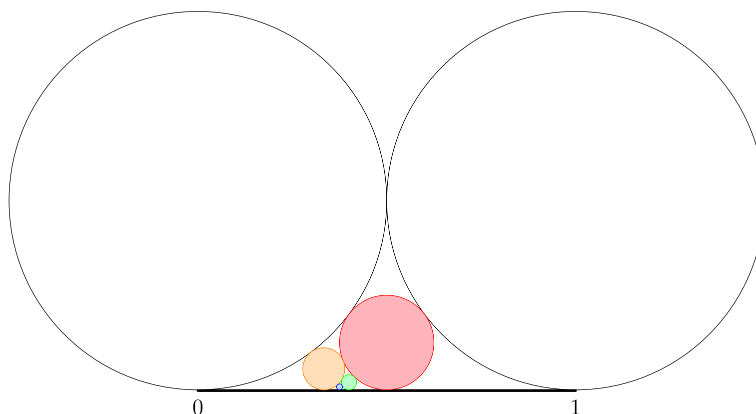
 licensed under CC BY-SA 4.0 int'l

Problem K: Kids draw the darndest things

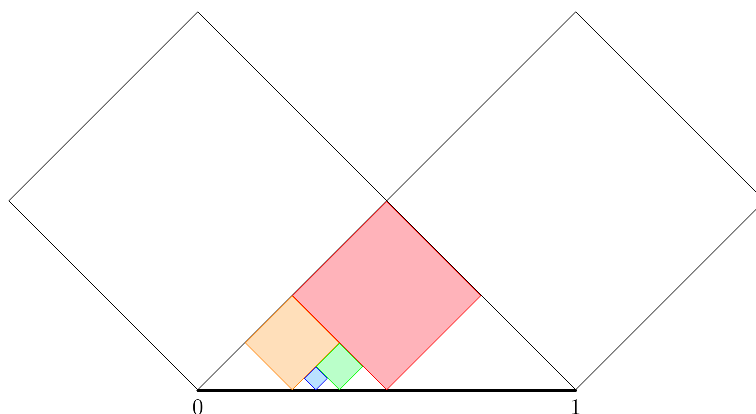
Kelly is drawing pictures with circles. In order to draw a picture, she does the following:

1. She draws a horizontal line segment of length 1 and she assigns the coordinates 0 and 1 to its left and right end, respectively.
2. She draws two equally big circles that touch the line of the segment from above (one at each end of the segment) and touch each other above the middle of the segment. These two circles become her active circles.
3. She draws a third circle that touches the line segment and both active circles.
4. She chooses a sequence of $n \geq 0$ directions, each being either "left" or "right".
5. Exactly n times she repeats the following:
   1. She looks at the next direction. Only one of the two active circles remains active: the one that touches the segment farther in that direction.
   2. The most recently drawn circle becomes active.
   3. She draws a new circle that touches the segment and both active circles.

For example, this is the picture she would draw for the sequence "left, right, left". Circles were drawn in the order red, orange, green, blue. (The blue one is quite tiny, don't miss it!)



As boys often do, Kevin forgot his compass at home. But he also likes pretty pictures, so he decided that he will do the same with squares (rotated 45 degrees, so that their diagonals are parallel to the coordinate axes). For example, for the sequence "left, right, left" Kevin would draw this:



 licensed under CC BY-SA 4.0 int'l

Problem specification

The "circles-to-squares" function c is defined as the only continuous function on $[0, 1]$ with the following property: For any sequence of directions, if Kelly drew her picture and the last circle touches the segment at x_c, and Kevin drew his picture and the last square touches the segment at x_s, then let $c(x_c) = x_s$.

You will be given a collection of inputs for c. Compute the corresponding outputs.

Input specification

The first line of the input file contains an integer $t \leq 100$ specifying the number of test cases. Each test case is preceded by a blank line and consists of a single line.

Each test specifies the number x_c for which you must compute $c(x_c)$. Decimal numbers are not exact, so the inputs are specified as roots of polynomials. Each test case has the form "p k", where p is a polynomial with integer coefficients (see below) and k is a positive integer. Find the k-th smallest distinct real root of p and use that value as x_c. It is guaranteed that it exists and lies in $[0, 1]$.

In the **easy subproblem K1** each p is linear. (Therefore, every x_c is a rational number.)

In the **hard subproblem K2** each p is either linear or quadratic.

Additionally, the input will be such that each line of correct output will have fewer than 40,000 characters.

Output specification

For each test case, output a single line. If the corresponding $x_s = c(x_c)$ is an algebraic number (i.e., it can be expressed as a root of a polynomial with integer coefficients), output it using the same formatting as in the input. You must output the polynomial with the *smallest possible positive degree*. If the corresponding x_s is transcendental, output the line "`transcendental`" instead.

Polynomial formatting for input and output

- The description of a polynomial does not contain any spaces.
- The powers of x in a polynomial are printed in descending order.
- Every term is written as `a*x^n` where `a` and `n` are integers, except that `a*x^1` is written as just `a*x`, and `a*x^0` as just `a`.
- All powers of x are explicitly mentioned, even if their coefficient is zero. (E.g., `2*x^2+3` is invalid.)
- The leading term has a positive coefficient. (E.g., `0*x^2+1*x+1` and `-1*x+2` are invalid.)
- All coefficients are always explicitly mentioned, even if they are 0 or ± 1. (E.g., `x^3-x^2+x+1` is invalid.)
- There is no $d > 1$ that divides all coefficients of the polynomial. (E.g., `2*x+2` is invalid.)

Examples of correctly formatted polynomials include "`1*x^3+0*x^2+0*x-0`" and "`7*x^2-47*x+123456`". For more examples see the input files directly.

Example

input

```
2

2*x-1 1

3*x-1 1
```

output

```
2*x-1 1
4*x-1 1
```

 licensed under CC BY-SA 4.0 int'l

Task authors

Problemsetter:	Michal “mišof” Forišek
Task preparation:	Michal “mišof” Forišek
Quality assurance:	Michal “majk” Švagerka

Solution

For Kevin’s figure we can easily compute the points where the squares touch the line segment: as he follows the sequence of directions, the next point is always halfway between the previous two. Hence, the points are always rational and their denominators are powers of two.

Kelly’s figure is a bit more complicated, but not that much more. After doing a bit of math we can discover that her points are also always rational, but the formula is a bit more tricky. If two circles touch the segment at a/b and c/d (where both fractions are in their reduced form), the one between them will touch it at $(a + c)/(b + d)$.

Hence, if we visualize all possible sequences of directions as a binary tree and assign the coordinate of Kelly’s corresponding point to each vertex of the tree, we will get a known structure: the Stern-Brocot tree, starting from $1/2$.

Thus, Kelly can produce any rational number in $(0, 1)$. And this observation alone was enough to solve the easy subproblem, as follows:

- You read the linear polynomial and compute the fraction f that is its root.
- You find the path that leads to f in the Stern-Brocot tree.
- You go down the same path in the other (dyadic) tree to compute Kevin’s output.

As the set of all rationals on $(0, 1)$ is dense and our function is clearly increasing on this set, it now also follows that the function is correctly defined on the entire interval $[0, 1]$.

The function we have defined in this obscure way does in fact also have a name or two. Poetically, it is known as the “slippery devil’s staircase”, or, more formally, as Minkowski’s question-mark function.

We have already observed that for rational inputs this function returns rational outputs, but not general ones – only such that the denominator is a power of two. What happens if the input is a quadratic irrational? Surprisingly, it turns out that the output will still be a rational number!

Why is that? Because quadratic irrationals have one important property: their continued fractions are periodic. More precisely, they exactly correspond: the value of each periodic continued fraction is the root of a quadratic equation with integer coefficients, and each irrational root of such an equation has an infinite periodic continued fraction. One pair of proofs.

With a bit of simplification we can say that the values in the continued fraction representation of a number describe the path down the Stern-Brocot tree that leads to said number (if it’s rational) or to its better and better approximations (if it isn’t). A periodic continued fraction produces a periodic path. Hence, we then also get a periodic path in Kevin’s dyadic tree, and that’s precisely where the formulas listed in the Definitions section come from. In Kevin’s tree a periodic path corresponds to a simple geometric series and its infinite sum is obviously rational.

Thus, in order to solve the hard problem we do the following:

- Read the at-most-quadratic polynomial from the input.
- Compute its continued fraction representation until you find its pre-period and period.
- Convert those to the path down Kevin’s tree. If it’s finite, compute where it leads, if it’s periodic infinite, compute the sum of the corresponding infinite geometric series.

 licensed under CC BY-SA 4.0 int’l

Step two in more detail: Suppose we want the continued fraction representation of $\psi = \frac{a+b\sqrt{c}}{d}$. Let $e = \lfloor \psi \rfloor$. Then clearly the representation we seek has the form $e + 1/f$ for some continued fraction f. We can compute e directly (in my implementation I use binary search on integers to make sure all my computations are exact) and then we can rearrange the terms to get that f is equal to some possibly-different quadratic irrational. We just keep track of all quadratic irrationals encountered along the way, and as soon as one repeats (which we know has to happen), we just found the preperiod and period of the continued fraction.

 licensed under CC BY-SA 4.0 int'l

Problem L: Lethargic foe

Alice entered a local chess tournament where she will play Bob a total of seven times, wielding white pieces every time. Bob is not a very good chess player and he is notorious for using a lazy strategy. He always mirrors his opponent's moves. For example, when Alice starts by moving her pawn `e2` to `e4` (denoted `1. e4` in Standard Algebraic Notation), Bob will respond by moving his pawn from `e7` to `e5` (`1. ... e5`). When she follows with a queen move `2. Qh5`, he will counter with `2. ... Qh4`. Of course, such a mirroring move is not always possible. For example, if after the said two moves Alice captures the queen by playing `3. Qxh4`, Bob can no longer reply the same way, as he has no queen left. Similarly, if Alice played `3. Qxf7+`, Bob would be in check and the move `3. ... Qxf2+` would be illegal. When such a situation occurs, Bob will just stare at the chessboard until his time runs out.

Problem specification

Alice considers games with Bob a waste of time, so she wants to win them as fast as possible. This means that the number of moves has to be as small as possible, and Bob has to be able to mirror her every move until the move that delivers the checkmate.

However, Alice wants to challenge herself at least a little bit. She wants to play seven different games, and in each of them use a different piece in the move that delivers the checkmate. That is, there has to be a game where the last move is performed by a pawn, a knight, a bishop, a rook, a queen, a king, and any promoted piece, respectively. Your task is to suggest such games to Alice.

Note that the piece considered to deliver the mate is the one that moves in Alice's last turn, even if it's just moving out of the way (a discovered attack). Delivering the mate with a promoted piece counts as a promoted piece victory regardless of whether you promoted it to a queen, a rook, a bishop or a knight. When the promotion itself is a checkmate, it is considered to be a promoted piece move and not a pawn move. Castling is considered to be a king move, not a rook move.

Output specification

Submit a valid file in Portable Game Notation (PGN) containing at most 7 games. We will ignore all comments, tags, and variations (but they still have to be syntactically correct if present) and only consider the main line. So the simplest way is to submit a file containing **only the `movetext` for each game**, separated by blank lines.

Each game has to be valid and follow the above rules. It has to be legal, end with the black king in a checkmate, and every black move has to mirror the immediately previous white move.

For each piece type you can score a point. In order to determine whether you do, we count the number of moves in your shortest game (defaulting to 100 if no game of that type has been submitted) and then we do the same for our best solution. To score a point, your solution has to use at most as many moves as ours.

In the **easy subproblem L1** you need to score at least 3 points.

In the **hard subproblem L2** you need the full 7 points.

 licensed under CC BY-SA 4.0 int'l

Example

Below is an example of a file you might submit. The file contains four consecutive games in PGN, separated by a blank line.

```
1. f3 e5
2. g4?? Qh4#

1. e4 e5 2. Bc4 Bc5 3. Qh5 Nf6 4. Qxf7#

1.c3 c6 2.Qc2 Qc7 3.Qxh7 Qxh2 4.g3 g6 5.Bg2 Bg7 6.Qxg7 Qxg2 7.Qxh8 Qxh1 8.Qxg8#

[FEN "8/3RP2k/6pp/3N4/3n4/8/3rp2K/8 w - - 0 1"]
[SetUp "1"]
1.e8=Q#
```

This submit would get a Wrong answer for the following reasons:

- The first example game is incorrect because it ends with white king in checkmate.
- The second example game is incorrect because the black's third move is not a mirror image of the third white's move.
- The third game is correct. However, you will not get a point because there is a better solution for the queen giving the checkmate.
- The fourth game is incorrect as you didn't use the standard starting position. Note that the mating move would be only considered a checkmate by a promoted piece for the purposes of this task, and not pawn, queen or rook.

 licensed under CC BY-SA 4.0 int'l

Task authors

Problemsetter:	Samuel “Hodobox” Gurský
Task preparation:	Michal “majk” Švagerka
Quality assurance:	Caissa

Solution

This looks like a task that could probably be solved by hand, at least partially, by case distinction. As we will see, getting those 3 points should be rather achievable – solutions for queen, knight and either of pawn and promoted piece are discoverable. Later, we'll discuss how to get the help of a machine.

Queen

Intuitively, a mate with queen is going to be relatively short. A quick online search confirms this, as this is a well known sequence:

```
1. d4 d5 2. Qd3 Qd6 3. Qh3 Qh6 4. Qxc8#
```

Knight

The knight has two advantages in this task – it is fairly slow and also creates non-linear threats. Thanks to the first property, we are able to prove a rather high lower bound on the number of moves, which we can use to construct a solution. The second property tells us that the knight will either need significant help from other white's pieces, or we are after a smothered mate.

Let's assume that the king never moves. In that case, the `b1` knight needs three moves to check the black king. This check can be delivered on any of `c7`, `d6`, `f6` and `g7`. All these are protected, so we must deal with the defenders somehow. The squares `c7` and `g7` seem unlikely – as the defender is adjacent to the king, it would have to leave its starting place, creating an escape route. The square `d6`, on the other hand, has three defenders – pawns `c7` and `e7` and bishop at `f8`. We've already established that the bishop probably shouldn't move, so `e7` needs to be occupied. We can either move the pawn to `e6` and fill `e7` with black knight, or pin the `e7` pawn. Pinning can only realistically be dne with the queen, since rooks are too slow and have other issues (more on that later), so the first option is probably the best. This gives us the following mate:

```
1. Nc3 Nc6 2. Ne4 Ne5 3. e3 e6 4. Ne2 Ne7 5. c3 c6 6. Nd6#
```

The case of `f6` is similar – there are in fact four defenders, but we can deal with them the same way. Regardless of what we do, we need six moves.

Pawn

Next on the list is a pawn. Note that in the beginning, the pawn is six ranks away from the enemy king. The pawn can reduce this by two on its first move, and then by one on the following moves. To reach the distance of one, when the pawn can actually deliver a check (or checkmate) to the enemy king, we need at least four moves shared between the pawn and the king. Four is certainly not going to be enough. What are the challenges?

Assume we start with the f-pawn, aiming for the weak `f7` spot. However, black responds to `1. f4` with `1. ... f5`, and the pawns block each other. We can only avoid this by the pawn capture on its fifth rank, e.g., `1. f4 f5 2. g4 g5 3. gxf5 gxf4`.

 licensed under CC BY-SA 4.0 int'l

Our pawn is now on track to f7, and the king still has no escape route. We need two moves to reach f7 to deliver the final blow, but first we need to protect the pawn from being captured by the king. We can't do that with single move, so we use our knight: `4. Nf3 Nf6 5. Ng5 Ng4`. The rest is simple `6. f6 f3 7. f7#`

Promoted piece

A pawn needs five moves to reach the end of the board. As we established in the previous case, there is at least one extra move needed to avoid the opposing pawn, yielding a lower bound of 6. To achieve six, we need the pawn to promote on d8 or f8 (preferrably to a queen), the f7 (resp. d7) has to be blocked, and the promotion square needs to be protected by some other white piece to avoid capture by the enemy king. We note that d8 attacked by the queen on d1 without any move, provided that the d pawns disappear. This can actually be achieved simply by:

```
1. c4 c5 2. d4 d5 3. dxc5 dxc4 4. c6 c3 5. c7 c2 6. cxd8Q#
```

Rook

Rooks are very peculiar. They move only on ranks and files. How can a rook deliver a mate? We quickly realise that it is not possible on a file – when our rook moves to, say, e3 to attack the enemy king at e8, the black rook can also move to e6 (except when it is a discovered attack, but those are even harder to set up). To mate on a rank, note that the king cannot realistically advance to its fifth rank, so we need to deliver the mate on the opponent's half of the board. How does a rook cross that boundary? Well, the opponent needs to be able to do the same thing on his next move, **on the same file**, which is clearly impossible. Thus our move to the other half of the board has to actually be the checkmate.

The fastest way is probably to somehow open the h file and then do Rxh8#. We need to get rid of Ng8 and Bf8 somehow. One option to do both is to advance both g and h pawns, exchange them on the fifth rank to open the h file, and then offer the knight and bishop for capture. This gives:

```
1. h4 h5 2. g4 g5 3. hxg5 hxg4 4. Nf3 Nf6 5. exf6 exf3 6. Bg2 Bg7 7. fxg7 fxg2 8. Rxh8#
```

We've made quite a few assumptions in the analysis above and it turns out we were wrong. There is one other way of opening the h file, and we don't even need to move the g pawn to get the bishop out of the way – it can be captured on its starting position by a knight. This gives a six-move solution:

```
1. Nf3 Nf6 2. Ng5 Ng4 3. Nxh7 Nxh2 4. Nxf8 Nxf1 5. Ne6 Ne3 6. Rxh8#
```
which is in fact optimal.

Bishop

The case of a bishop is rather difficult – it is rather mobile, it doesn't suffer from the issues the rook did. We are unable to construct any non-trivial lower bounds, yet finding a solution seems hard. We need to be creative here – one solution of length 8 found by hand is:

```
1. e4 e5 2. d4 d5 3. Kd2 Kd7 4. Kc3 Kc6 5. Na3 Na6 6. Be3 Be6 7. dxe5 dxe4 8. Bb5#
```

There is in fact a faster solution only found by a machine:

```
1. e4 e5 2. f4 f5 3. exf5 exf4 4. f6 f3 5. fxg7 fxg2 6. Be2 Be7 7. Bh5#
```

 licensed under CC BY-SA 4.0 int'l

King

Last but not least, let's mate with a king. How can a king mate in the first place? There are two options – either by castling, or using a discovered attack.

The following argument shows that castling is clearly infeasible. Before castling, the white king is located on `e1` and the black on `e8`. After castling, the white rook is either on `d1` or `f1`, but such rook can never deliver check to the black king.

We can thus do discovered check only. For this to work, the corresponding file has to be open, and there needs to be a queen or a rook behind the king. There is, however, one more subtle issue that we need to resolve. If we move our king to, say, `e2`, to deliver the check, the square `e7` is also reachable for the black king and he can evade the check. We can try to cover the said square by some white piece, but that would only mean that the corresponding square `e2` would be also under attack, and our king cannot move there!

How do we get out of this? It's simple when you get the idea: the corresponding square has to be atacked by our king. For instance, with kings at `d3` and `d6`, we can move to `e4` and now black's move to `e5` is clearly impossible. Obviously, this can only happen at the third rank.

We need to solve the following tasks: - bring the king to the third rank - remove the pawns from the king's file - put the queen behind the king - attack the adjacent files

To achieve all of these in a short game, we use the knight's versatility:

`1. d4 d5 2. Kd2 Kd7 3. Kd3 Kd6 4. Nc3 Nc6 5. Nxd5 Nxd4 6. Nb4 Nb5 7. Na6 Na3 8. c3 c6 9. Nf3 Nf6 10. Ng5 Ng4 11. Ke4#`

During the contest, people managed to find solutions that were two moves shorter, with one of them being:
`1. f3 f6 2. Kf2 Kf7 3. Kg3 Kg6 4. Kh3 Kh6 5. e3 e6 6. Bd3 Bd6 7. Bg6 Bg3 8. hxg3 hxg6 9. Kg4#`

Machine solution

Finding all of the above by hand is a hard task, but we can use the help of a computer! Chess has a managable branching factor, and we get a lot of help from the fact that black mirrors our moves, so we can get twice as far. Reaching 6 or 7 moves seems within the realm of possibilities. So we just grab a chess library of choice and we are off to search the state space. (For example, `python-chess` is simple to grasp and we actually used it to check your solutions. Stockfish has bitboards and is written in C++ so we get more raw power.)

There are a couple of optimisations and heuristics that we can employ:

- Ignore positions that are identical due to transposition of moves.
- Hope that there is a solution in which a pawn never moves by a single square from its original position.
- Avoid moving our queen, since the queen solution is easy to find by hand and it limits the number of available moves significantly.
- Reduce the candidates on initial moves (e.g., the solution for a knight will probably start with a knight move).

These heuristics don't guarantee that we find the optimal solution, but they can help in finding one that is good. Once we have a good solution, we can try submitting it and we can also use it as an upper bound for the depth of the search for a better solution. This approach works for all cases except for king, where it is probably not possible to prove the optimality within the contest timeframe. We certainly failed to do so during the contest preparation!

 licensed under CC BY-SA 4.0 int'l

Problem M: McDroid's

The trend is clear: the number of robots in the world is growing and growing. You decided to become an entrepreneur and open the first robot restaurant in the world. Not a restaurant staffed by robots – a restaurant *for* robots. After a long preparation, everything is finally ready for the grand opening.

You have a long menu with many different kinds of delicious robot food. Most humans would probably just be confused because robot recipes don't have names, only numbers. And of course, robots only use binary. You're sure the restaurant will become very popular and it will attract a big crowd. As the owner, your biggest challenge is managing your time and making sure you don't leave anybody (any*bot*y?) waiting. Your second biggest challenge is that it's not always easy to understand what food the robots want.

Problem specification

In this problem, you will interact with an online dashboard page. You can find the link in the online version of this problem statement.

The page has a button to open the restaurant. When you press the button, your restaurant will open and it will start getting robot customers. Your team has just one restaurant. You will see the same restaurant if you access the dashboard from multiple browsers. Note that time continues to run even if you close the page.

Every customer will tell you what they want, and you must choose what food to give them. Some robots might just tell you "`HELLO I WOULD LIKE TO ORDER FOOD ITEM 1101.`", in which case you'd obviously give them `1101`. For other customers figuring out what they want may be more complicated. You'll see.

Robots will arrive in real time. Each robot is only willing to wait for some predetermined period of time (at least a minute, sometimes more). You may have multiple robots waiting in your restaurant at the same time. For each of them you will see the time left to serve them.

When you first open your restaurant, you'll have 10 reputation points.

There are two things that can go wrong at your restaurant:

- If you **serve incorrect food** to a customer, you lose one reputation point and **that customer** leaves.
- If you **fail to serve** a customer in time, you lose one reputation point and **everybody who was waiting** gives up and leaves.

Initially, the rules for solving **subproblems M1 and M2** are as follows:

- If the total time your restaurant was open reaches t_1 = **60 minutes**, you will solve subproblem M1.
- If the total time your restaurant was open reaches t_2 = **180 minutes**, you will solve subproblem M2.

Whenever your reputation reaches zero, you go bankrupt. You can then reopen the restaurant and continue trying to solve this problem. Whenever you reopen your restaurant, the following things will happen:

- You are given another 5 reputation points.
- If you didn't solve **subproblem M1** yet, the threshold t_1 is increased by 20 minutes.
- If you didn't solve **subproblem M2** yet, the threshold t_2 is increased by 20 minutes.
- As an additional penalty, your restaurant now also attracts human customers and you need to serve those as well. Humans are usually much more annoying than robots.

Notes

- The problem uses normal scoring rules. The sooner you begin, the smaller will be your time penalty when you solve a subproblem.

 licensed under CC BY-SA 4.0 int'l

- Please be merciful on our servers. The page auto-updates itself, you don't have to press F5 every millisecond. If you do, the server might temporarily block you and you might fail to serve some customers.
- As always, please contact us if you encounter any technical issues.
- One final word of caution: You should expect that over time the restaurant will become more popular and you *might (wink, wink)* start getting more picky customers.

Input and output specification

There is no input and no output. You will automatically get an **OK** verdict for a subproblem the moment the total time your restaurant was open reaches the corresponding threshold.

 licensed under CC BY-SA 4.0 int'l

Task authors

Problemsetter:	Tomi Belan, Peter “ppershing” Perešíni
Task preparation:	Tomi Belan, Peter “ppershing” Perešíni
Quality assurance:	the whole team helped by testing

Solution

One possible way to solve this task was to do it by hand. This was perfectly viable for the easy subproblem, but you probably had to give up as soon as a robot asked you for ten factorial in binary :)

As in real life, a much better way to run a restaurant is to automate as much of the process as you can. The practice problem R may have served as inspiration: you can use your own program to repeatedly access the restaurant dashboard. The program then needs two more ingredients:

- A collection of functions that handle known customer types.
- A way to alert you whenever a new customer type appears, so that you can implement a new handler.

However, there was an even easier way to solve this task. Instead of implementing a whole new framework to talk to our server you can reuse an existing one. Which one? Well, the one in your browser, of course!

The two main ways to do this were:

- Download the whole dashboard (including the javascript) to your local machine. Modify the local copy of the dashboard javascript to talk to our server instead of your localhost. Then you can implement all the customer handlers directly in that javascript.
- Write the handlers in your local editor and then paste them into your browser's javascript console to add them to the code of the actual dashboard.

The first option is probably safer in that you can restart it as you please without accidentally losing some of the logic you added to the code.

The “elevator music” used on the dashboard to notify you that a customer is waiting is a recording of a piece called “The Flea Waltz” by `Flying_Deer_Fx`, made available under a Creative Commons 0 license.

 licensed under CC BY-SA 4.0 int’l