Problem A: Abundance of sweets

Georg has a new profession: he now works in a candy factory. The job is way more boring than it sounds: he is responsible for quality assurance. And that means taking a box of candies, counting them and checking that none are missing. Help poor Georg! Write a program that will do the job for him.

You will be given a matrix with r rows and c columns. This matrix represents the top view of a box of candies. The matrix will only contain the following characters:

- "`.`": a free spot
- "`o`": the edible part of the candy
- "`<>v^`": candy wrapper

There are exactly two ways how a whole piece of candy looks like:

```
1. >o<    2. v
             o
             ^
```

Whenever you see three characters arranged in this way, you see a whole piece of candy.

Problem specification

For the given matrix count the number of whole pieces of candy it contains.

In the **easy data set** the box will only contain whole pieces of candy and nothing else.

In the **hard data set** the box can also contain fragment of candies: characters `o<>v^` that don't belong to any whole candy. To make your life easier, you may assume that the following configuration will never appear in the hard data set:

```
 v
>o<
 ^
```

Input specification

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

The first line of each test case contains two integers r and c ($1 \leq r, c \leq 400$): the dimensions of the matrix. Each of the next r lines contains one row of the matrix: c characters from the set "`.o<>v^`". (Their ASCII values are 46, 111, 60, 62, 118, and 94.)

Output specification

For each test case, output a single line with one integer – the number of whole candies in the box.

Example

input	output
1 5 4 .>o< v.^. ooo. ^.^. >o<<	3

There are three whole candies: in the first row, in the last row, and in the first column.

Problem B:

(This page is intentionally left blank.)

Problem C: Card game

You are enjoying a lovely weekend afternoon in a new amusement park. The park also offers a variety of activities that are about winning or losing a little money. One such activity has just caught your attention: a simple card game. As a person interested in puzzles and riddles, you almost instantly started to wonder: Should I pay for such a game? Am I expected to win or lose money while playing it?

This game is played with a standard deck of cards – there are four suits and in each suit cards worth 2 through 10, a Jack, a Queen, a King and an Ace. For the purpose of this problem, the Aces have value 1, Jacks 11, Queens 12, and Kings 13. In other words, each of the values 1 through 13 is present four times in the deck.

The simplest version of this game looks as follows: An employee of the amusement park shuffles the deck uniformly at random and gives you a card. Your winnings are equal to the value of the card.

Here is a more complicated version: As before, the dealer gives you a random card. You now have two options: either you take your winnings, or you return the card you have and ask for another one. The second one is again drawn uniformly at random, and you have to keep it.

The general version of this game has x turns, and you know x in advance. In each turn you can keep the card you have and end the game. In each turn except for the last one you can return the card and ask for another one.

In the **easy data set** the card you return is always returned back to the deck. Thus the new card you then get is always picked uniformly at random from a deck of 52 cards. In the **hard data set** the card you return is thrown away, so the more turns you take, the smaller the deck becomes.

Problem specification

You are given the maximal number of turns x. Find the maximum expected amount of money you can win in a single game.

(In other words, find the smallest y with the property: if playing a single game costed more than y, the amusement park would still profit from the game, even if all players played it optimally.)

Input specification

The first line of the input file contains an integer t specifying the number of test cases. Each test case is preceded by a blank line. Each test case is given as a single line with a single integer x (between 1 and 52, inclusive).

Output specification

For each test case, output a single line with one real number – the expected amount you will win if you follow an optimal strategy. Output at least 10 decimal places. Any answer with an absolute or relative error at most 10^{-9} will be accepted.

Example

input	output
1 1	7.000000000000

This is the correct answer both for the easy and the hard version, as for $x = 1$ they are the same: you have to keep the first card you get. From the 52 cards, each card is picked with probability $1/52$. There are 4 cards having each of the values 1 through 13. Hence the expected value of a card is $\sum_{i=1}^{13} i \cdot (4/52) = 7$.

Problem D: Data mania

This year, we, the IPSC organizers, finally decided to rewrite our old webpage and grading system. The old code was full of hacks and it looked more like spagetti code than anything else. During this conversion, we found out that we have quite a lot of data about previous years (teams, submissions, etc.). One of the things that is still missing (as you can see on our page) is the hall of fame. However, we would like to make it less boring than just "winners of the year XY". Thus, we need to analyze our data and calculate the most crazy statistics like "The names of teams with the longest accepted submissions". But as you can guess, we are now quite busy with the preparation of the contest – the new grader is being finished as we speak, and we still need to finish some tasks, including this one. Therefore, you should help us!

Problem specification

You will be given a set of five data tables about previous three IPSC contests (2009-2011). We already did a good job in cleaning these data so you can assume that it is consistent (we will explain later what this means). Note that these data file *are common for both subproblems* (easy and hard).

Your task is to compute results of several data analytics queries that are given in plain English.

Data files specification

The supplied data will consist of the following tables: problems, teams, team members, submissions and submission results. The files are named "`d-problems.txt`", "`d-teams.txt`", "`d-members.txt`", "`d-submissions.txt`" and "`d-messages.txt`".

In general, each table consists of several space-separated columns. Each column holds a string consisting of one or more permitted characters. The permitted characters are letters (`a-z`, `A-Z`), digits (`0-9`), underscores, dashes and colons (`_-:`). All other characters (including spaces) were converted to underscores.

The file `d-problems.txt` contains three space-separated columns. The first column contains the year of the contest, the second column contains the task (one letter, A-Z), and the third column contains the (co-)author of the problem. If the task had multiple co-authors, they will be listed on separate lines.

The file `d-teams.txt` contains five space-separated columns. The first column contains a TIS (a unique identifier of a team; also, teams in different years of the contest never share the same TIS). The second column contains the name of a team, the third column contains its country, the fourth column contains the name of its institution, and the fifth column contains the division ("open" or "secondary").

The file `d-members.txt` contains three space-separated columns. The first column contains the TIS of a team, the second column contains the name of one team member, and the third column contains the age of that member (or zero if age was not specified). If the team had more than one team member, each team member will be listed on a separate line.

The file `d-submissions.txt` contains six space-separated columns. Each row corresponds to one submission received by our server. The first column contains the date when the submission was received in the format "YY-MM-DD". The second column contains the corresponding time in the format "hh:mm:ss". The third column is the TIS of the team who sent the submission. The fourth column is the subproblem identifier. It consists of a letter and a number (1 for easy, 2 for hard), such as "A2" or "G1". The fifth column contains the size of the submitted file (in bytes). Finally, the last column contains the hash of the submitted file.

The file `d-messages.txt` contains four space-separated columns. Each row describes a message shown to one of the teams. The first column contains the TIS of the team. The second column contains a unix timestamp of the moment when the message was created. The third column contains subproblem identifier (in the same format as above). Finally, the fourth column contains the status message which is either

"OK" or "WA". (Note that during actual contests there were several other possible messages including "contest is not running" and "too many submissions". Those were removed when we cleaned up the data.)

You may assume the following:

- For the purpose of conversion between unix timestamps and dates our server runs in the timezone +02:00.
- Each contest starts exactly at noon and lasts until 16:59:59. (Note: we shifted the 2009 contest by -2 hours to guarantee this.)
- The triples (TIS, timestamp, subproblem) are unique, i.e., there are no two concurrent submissions of the same problem from the same team.
- Submissions and messages can be paired using the triple (timestamp, TIS, subproblem). That is:
  - For each submission, there is exactly one corresponding message with its evaluation.
  - For each message there is exactly one corresponding submission.
- Submissions and messages are consistent with problems.
  - For each submission/message, the subproblem is a valid subproblem for that year, i.e. there is a corresponding problem for that year.
  - For each year and the problem, there are exactly two subproblems (easy and hard) and both contain at least one submission.
- For each submission there is a team with the corresponding TIS.
- For each year and subproblem, each team has at most one submission that received the message "OK". That submission, if it exists, is the last one in chronological order.
- For each team there is at least one member.
- For each team there is at most three members.
- For each team member there is a team with the corresponding TIS.

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 one or several consecutive lines describing the analytics query in plain English. Unless otherwise stated, you should assume that

- "subproblem" means year + subproblem,
- a team is identified by its TIS, not by its name,
- an accepted submission is a submission that received the message "OK".

Output specification

For each testcase, output several lines. On the first line, output an integer c – the number of results for that testcase. The next c lines should contain the results as tuples, with elements of a tuple separated by single spaces. (Note that the format of the output is very similar to the format of the data files.)

If the results should be sorted according to some sorting criteria, proceed as follows: Integers and doubles should be sorted according to their numerical value. Strings should be sorted lexicographically according to their ASCII value. Years should be sorted chronologically (in increasing value).

Moreover, if the query involves string comparison, the comparison should be done using ASCII values. You may assume that no query will require comparison of floating-point values for equality.

Example

input

```
2

For each year (in chronological order)
find the first accepted submission.
Print the year, the name of the team
that made it and the time from the start
of the contest. Time should be formatted
as "mm:ss".

For each year (in chronological
order) print the year, the number of
OK submissions and the number of WA
submissions in that year.
```

output

```
3
2009 Bananas_in_Pyjamas 02:54
2010 Gennady_Korotkevich 02:28
2011 Never_Retired 06:16
3
2009 3352 2942
2010 4441 5579
2011 2437 3340
```

Problem E: Evil matching

Consider two sequences of **positive** integers: $T = t_1, t_2, \ldots, t_n$ (text) and $P = p_1, p_2, \ldots, p_m$ (pattern). The classical definition says that P matches T at position k if $p_1 = t_k$, $p_2 = t_{k+1}$, ..., and $p_m = t_{k+m-1}$. One can easily find all positions where P matches T – for example, by using the Knuth-Morris-Pratt algorithm.

Easy is boring. Let's make it a bit harder. We say that P *evil-matches* T at position k if there is a sequence of indices $k = a_0 < a_1 < \cdots < a_m$ such that:

$$\begin{aligned} t_{a_0} + t_{a_0+1} + \cdots + t_{a_1-1} &= p_1 \\ t_{a_1} + t_{a_1+1} + \cdots + t_{a_2-1} &= p_2 \\ &\cdots \\ t_{a_{m-1}} + t_{a_{m-1}+1} + \cdots + t_{a_m-1} &= p_m \end{aligned}$$

In other words we are allowed to group and sum up consecutive elements of the text together before matching them against the pattern. For example, if we have $T = 1, 2, 1, 1, 3, 2$ and $P = 3, 2$, then there are two evil-matches: one at $k = 1$ (with $a_1 = 3, a_2 = 5$), the other at $k = 5$ (with $a_1 = 6, a_2 = 7$).

The *evil compatibility* of a text T and a pattern P is the number of different k such that P evil-matches T at position k.

Problem specification

Given a text T and two patterns P_1, P_2 calculate:

1. The evil compatibility of T and P_1.
2. The evil compatibility of T and P_2.
3. The smallest **positive** integer n for which the evil compatibility of T and $P_1 \cdot n \cdot P_2$ is maximized. (The symbol $\cdot$ represents concatenation.)
4. The evil compatibility of T and $P_1 \cdot n \cdot P_2$ for that n.

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 six lines:

- The first line contains the number n – the length of text.
- The second line contains n positive integers – the elements of text.
- The third line contains the number m_1 – the length of the first pattern.
- The fourth line contains m_1 positive integers – elements of the first pattern.
- The fifth line contains the number m_2 – the length of the second pattern.
- The sixth line contains m_2 positive integers – elements of the second pattern.

You may assume the following constraints:
In each test case of the easy data set: $n \leq 5000$, $m_1, m_2 \leq 600$, and the sum of all elements of $T + P1 + P2$ does not exceed 11 000.
In each test case of the hard data set: $n \leq 11 \cdot 10^6$, $m_1, m_2 \leq 2 \cdot 10^6$, and the sum of all elements of $T + P1 + P2$ does not exceed $22 \cdot 10^6$.

Output specification

For each test case output a single line.
This line should contain four space-separated numbers as described in problem specification.

Example

input

```
1

13
1 1 1 1 1 47 1 1 1 1 1 1 1
3
1 1 2
3
1 1 1
```

output

```
6 8 48 2
```

The first pattern evil-matches the text at positions 1, 2, 7, 8, 9, and 10.
The second pattern evil-matches the text at positions 1, 2, 3, 7, 8, 9, 10, and 11.
The pattern "1,1,2,48,1,1,1" evil-matches the text twice – at position 1 and at position 2.

Note that the value $n = 48$ is indeed optimal:
If $1 \leq n < 47$, the pattern "1,1,2,n,1,1,1" does not evil-match the text at all.
The pattern "1,1,2,47,1,1,1" evil-matches the text only at position 2.
There is clearly no $n > 48$ such that the pattern "1,1,2,n,1,1,1" evil-matches the text at three positions.

Problem F: Fair coin toss

Lolek and Bolek are studying to become magicians. For the last hour they have been arguing whose turn it is to take out the trash. The fair solution would be to toss a coin... but one should never trust a magician when tossing a coin, right?

After looking through all pockets, they found n coins, each with one side labeled 1 and the other labeled 0. For each coin i they know the probability p_i that it will show the side labeled 1 when tossed.

Of course, just knowing their coins did not really solve their problem. They still needed a fair way to decide their argument. After a while, Lolek came up with the following suggestion: they will toss all the coins they have, and xor the outcomes they get. In other words, they will just look at the parity of the number of 1s they see. If they get an even number of 1s, it's Bolek's turn to take out the trash, otherwise it's Lolek's turn.

Now they are arguing again – about the fairness of this method.

Problem specification

A set of coins is called *fair* if Lolek's method is fair – that is, the probability that Bolek will take out the trash has to be exactly 50%.

You are given a description of all coins Lolek and Bolek have.

As a solution of the **easy data set F1** find out whether these coins have a fair *subset*.

As a solution of the **hard data set F2** find out *how many* subsets of the given set of coins are fair.

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 the number n of coins (at most 10 in the easy data set, at most 60 in the hard data set). The second line contains n decimal numbers: the probabilities $p_1, \ldots, p_n$. Each probability is given to exactly 6 decimal places.

Output specification

For each test case output a single line.

For the easy data set this line should contain "0" if the given set has no fair subset, and any positive integer not exceeding 2^{60} if it contains at least one such subset.

For the hard data set this line should contain the exact number of fair subsets.

(Note that a program that correctly solves the hard data set should also solve the easy data set.)

Example

input

```
2

3
0.500000 0.500000 0.500000

4
0.000001 0.000002 0.000003 0.000004
```

output

```
7
0
```

In the first test case each subset is obviously fair. In the second test case no subset is fair. When tossing a subset of these coins, you are almost certain to see all zeroes – Bolek would be really unhappy!

Problem G: Gems in the maze

Scrooge McDuck has a new plan how to increase his wealth. He found ancient ruins with an extraordinary maze. This maze consists of n chambers. The chambers are numbered 0 through $n-1$. Each chamber contains exactly one gem. Chambers are connected by one-way tunnels. Each chamber has exactly two outgoing tunnels: one leads to the chamber with number $(a \cdot v^2 + b \cdot v + c) \mod n$, the other will bring you out of the maze.

You can enter the maze at any location, move along the tunnels and collect the gems. But once you leave the maze, you'll trigger a self-destruct mechanism – the ceiling of the maze will collapse and all the gems that you did not collect will be lost forever.

Scrooge wants to know the maximum number of gems he can take from the maze.

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 consist of a single line containing four integers a, b, c, and n – the numbers that describe one particular maze.

In the easy data set $n \leq 300$. In the hard data set $n \leq 2^{24}$.

Output specification

For each test case, output a single line containing a single integer – the maximum number of gems that can be taken from the maze.

Example

input

```
3

1 2 0 64

0 2 1 47

0 3 5 128
```

output

```
5
23
64
```

The starting chamber matters. For instance, assume that in the first example test case Scrooge starts in the chamber 0. His only two options are a tunnel that leads back to chamber 0 and a tunnel that leads outside – not much of a choice. A much better strategy is to start in the chamber 2 and follow the path $2 \to 8 \to 16 \to 32 \to 0 \to$ *outside.*

Problem H: Harvesting potatoes

There is a rectangular field divided into $r \times c$ square cells. Potatoes grow on all cells of the field. You have a potato harvester and want to harvest all of them. A potato harvester is basically a large car equipped with mechanical spades and other stuff necessary to get the potatoes from the soil to the back of the car. The driver has a switch that turns the mechanical spades on and off.

The harvester operates in passes. In each pass it traverses either a single row or a single column of the field, starting at one border and ending at the opposite one. The capacity of the harvester is limited: in each pass it can only harvest at most d cells. These can be any of the cells it passes through. In this task your goal will be to produce a harvesting schedule that uses *the smallest number of passes.*

In practice there is one more factor: harvester drivers are bad at following complicated instructions. Harvesting contiguous segments is much simpler than repeatedly turning the spades off and on again at the right places. The *difficulty of a given pass* can be measured as the number of times the driver has to flip the switch during the pass. In other words, the difficulty is twice the number of contiguous segments of cells harvested in that pass. (Note: The driver must harvest each cell *exactly* once. It is *not allowed* to leave the spades on while the harvester passes over already harvested cells, as this damages the soil.)

The difficulty of a schedule is the difficulty of the most complicated pass it contains, i.e., the maximum over all difficulties of individual passes.

Problem specification

You will be given the dimensions r and c and the capacity d. For the **easy data set H1** your task is to produce any harvesting schedule with the smallest number of passes. For the **hard data set H2** your task is to produce any harvesting schedule that solves the easy data set and additionally has the smallest possible difficulty (out of all such schedules).

Input specification

The first line of the input file contains an integer t specifying the number of test cases. Each test case is preceded by a blank line. Each test case is given as a single line with three integers: r, c, and d.

Each test case in the easy data set has $r, c \leq 12$. Each test case in the hard data set has $r, c \leq 400$.

Output specification

For each test case, output r rows, each containing c space-separated *positive* integers. These represent the harvesting schedule. More precisely, each integer gives the number of the pass in which the particular cell is harvested. (Additional whitespace is OK as long as the number of tokens in your output is correct.)

Example

input

```
2

2 9 5

3 5 2
```

output

```
1 1 1 1 1 2 2 2 2
3 3 3 3 3 4 4 4 4

1 2 3 4 5
1 2 3 4 5
6 6 7 8 7
```

The first test case: The cells are harvested in rows. The second test case: We first do one pass in each column and then three passes in the last row. The output shows a schedule that is shortest but does not have the smallest possible difficulty – to fix that, swap the 8 with one of the 7s.

Problem I: Invert the you-know-what

Good news everyone! We just "recovered" a password file from the webserver of our arch-enemies!

Input specification

The same input file is used for both data sets. The strings on the left seem to be usernames. The ones on the right... we have no idea. Do you?

Output specification

For each data set you have to submit a file with the following content: a username, a newline, the password for that username, and another newline. (To accommodate multiple operating systems, newlines can be either LF, or CR+LF.)

We will try logging in to the arch-enemies' webserver using that username and password. If the login process succeeds, we will accept your submission.

For the easy data set you may use any valid username. For the hard data set you must use the username `robot7`.

Problem J: Jukebox

The IPSC organisers recently installed a new jukebox in their room. As expected, the jukebox was an instant hit. In fact, there have already been several violent exchanges between people trying to select different songs at the same time, but that is a story for another time. This story is about the jukebox and Milan. As a pianist, Milan likes music and especially likes to play the music he hears. Thus it become unevitable that Milan wants to play music from the jukebox. But here is the problem – the jukebox just contains a lot of recorded music and no music sheets.

In order to recover music sheets from the jukebox, Milan already copied all the music to his laptop. Now he just needs to convert these recordings into music sheets. And that, as you surely guessed, will be your task. Fortunately for you, Milan already made some progress, so you will only need to recover the pitches of separate tones.

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 describes one raw recording. It starts with a line containing a single integer n – the number of recorded samples from a mono microphone. The next n lines will contain a single integer v_i – numerical value of i-th sample. The sample rate is 22050 Hz.

Moreover, for your convenience we provide MP3 files named `j.{example,easy,hard}.testcase#.mp3` containing the same recordings (except for minor differences due to a lossy MP3 compression).

Output specification

For each test case, output all tones of the song. On the first line, output the count c of tones. On the next lines, output c whitespace-separated tones, where each tone is one of the following strings "`A`", "`A#`", "`B`", "`C`", "`C#`", "`D`", "`D#`", "`E`", "`F`", "`F#`", "`G`", "`G#`".

You may assume that each tone in the input matches one of the twelve musical notes. Note that you do not need to determine the exact octave of the tone.

Example

input

```
2

261820
-533
-863
... (216818 lines)

1535092
-51
... (1535091 lines)
```

output

```
8
C D E F G A B C

84
D A A G C A A G F D ...(74 more tones)
```

The first input is the C major scale. The second input is a traditional Slovak folk song "Tota Helpa" (for some strange reason also known in Japan as Hasayaki).

Problem K: Keys and locks

To: Agent 047.
Mission: Acquire suitcase from room #723.
Briefing:
We managed to get you into room #719, which is on the same floor of the hotel. The hotel still uses the old pin tumbler locks. Housekeeping should have a master key that works for the entire floor. Obtain it in order to get to room #723 undetected. This mission statement will self-destruct in five seconds. In case it does not, please tear it into small pieces and eat it.

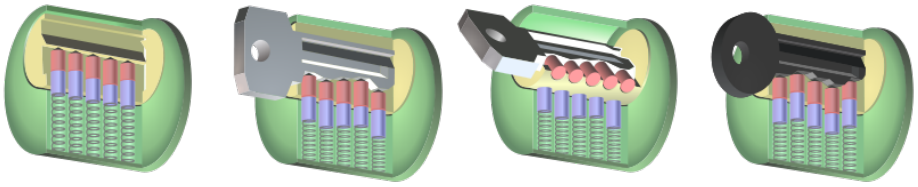
Pin tumbler locks

The photograph on the right shows a door with a pin tumbler lock. The static part of the lock is highlighted in green. The inside part, called the plug, is shown in yellow. This part of the lock rotates when a correct key is inserted and turned.

Below are four drawings of the inside of the lock. The first one shows a lock without any key. Inside the lock there are multiple metal pins (blue+red). If there is no key present, the springs push these pins almost, but not entirely, into the plug.

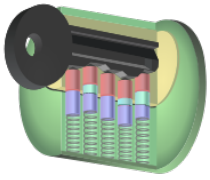
The second and third drawing show what happens when a correct key is inserted into the lock: Each of the pins is actually cut into two separate parts (a blue part and a red part). When a key is inserted into the lock, it pushes the pins out of the plug. The correct key has the property that the boundary between the red and the blue part of each pin exactly matches the boundary of the plug. When this happens, the key can be used to turn the plug and thereby to lock/unlock the door.

The last drawing shows what happens when an incorrect key is inserted into the lock. The pins will be aligned differently and they will block the plug from turning.



Master keys

When you run a hotel, you have the following situation: On one hand, you have a lot of guests, and each guest should only be able to open their own door. On the other hand, you don't want housekeeping to carry a ring with hundreds of keys. It is way more convenient for them to have a *master key* – a single key that can open all the locks.



With pin tumbler locks, creating a master key is really simple. You can even take a set of existing locks+keys and one new key and modify the locks to achieve the desired state – each lock can still be unlocked by its original key, and the new key can unlock all of the locks. How to do this? Each pin in each lock will now be divided in *two places*: one that corresponds to the original key, and one that corresponds to the new master key. (Technically, some of the pins will still be divided in one place. This happens whenever the original key and the master key have the same height at the corresponding location.)

The drawing on the right shows the lock from the previous drawings modified to also admit the second key. Three of the five pins are now divided into three parts (blue, cyan, and red). Now the plug can also be rotated with this new key inside. When that happens, the red and some of the cyan parts of the pins will be inside the plug.

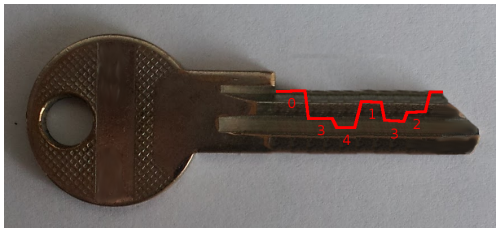
Key profiles and bitting

On the right you can see a *key profile* – an "empty" key. If you want to create a new key for a lock, you need to take the corresponding key profile (one that fits your lock) and then you need to create the right pattern (called *bitting*) on the part of the key that goes into the lock. The locksmiths have machines that do this easily, but you can also do it using an iron file, if you have to. (In this task, you have to.)

Each particular bitting can be described by a string of digits. If you have a lock with p pins, there will be exactly p locations on the key that touch the pins when the key is inserted properly. Each pin can be cut in one of $k \leq 10$ places. (The possible cut locations are sufficiently apart to prevent the same key from working with differently cut pins.) Thus there are exactly k^p different lock-key pairs. Each particular key can be described as a string of p digits from the set $\{0, \ldots, k-1\}$.

Different manufacturers use different encodings of the bitting. In this problem, 0 represents the shortest pin and correspondingly the shallowest cut on the key. (I.e., the larger the number, the more you have to file away when creating the key from an empty profile.) The first number is the pin closest to your hand. (This actually does not matter in this problem.)

In this problem, an empty key profile works as a key with bitting containing all zeroes.



(Note: In practice there is also a standard saying that adjacent pins should have similar cut locations. This is to ensure that the correct key can be inserted and removed smoothly. In this problem we will ignore this and consider all possible keys valid – after all, you will not be using any key too many times.)

The red line in the figure on the right shows how to file the key profile in order to obtain a key for $p = 6$, $k = 5$, and bitting 034132.

Problem specification

You may assume that the bittings of the three actual keys (the master key and the guest keys for rooms #719 and #723) are distinct and that they were generated uniformly at random before you started solving the task. (I.e., our grader will not try to do any nasty tricks.)

Your task is to unlock the room #723, using any key that works. There are two completely independent scenarios with slightly different rules. You will be submitting your actions for the easier scenario as output files for the subproblem K1, and for the harder scenario as K2. Here is the list of allowed actions:

- "`checkin`" – check into room #719 at the hotel. Our answer will have the form: "`room key: XXXXX`", where the string of Xs will be replaced by the bitting of the guest key to room #719. (You are not allowed to modify this key, you will need it to check out without raising any suspicions.)
- "`file A XXXXX`" – use an iron file on key profile number A to produce the specified bitting. Our answer will have the form "`key A new bitting YYYYY`". Note that `YYYYY` may differ from `XXXXX`. We will only apply the changes that are still possible. E.g., if you take a key with bitting 444222 and try to file it to the bitting 333333, you will get the key 444333.
- "`try A R`" – try using key A to unlock room R. The room number must be one of 719 and 723. Our answer will have the form "`success`" or "`failure`". Once you get the answer "`success`" for $R = 723$, you have successfully solved the particular subproblem.
- "`restart`" – only use this if you did something stupid and want to start again from scratch. We will generate new keys for you and reset all your key profiles. Our response will be "`restarted`".

Rules common for both subproblems

- You are allowed to make at most **30 submissions** (not just 10 as for other problems).
- Each submission in which you do not solve a subproblem does count as an incorrect submission. (Using as few submissions as possible is probably not a bad idea.)
- Each submission must be a text file containing between 1 and 20 lines, inclusive.
- Each line must contain exactly one of the commands specified above.
- Commands are processed one by one in the order in which you specified them.
- Lines with incorrect syntax are ignored and an error message is printed for each of them.

Specific rules for the easy subproblem

- The number of different pin heights is $k = 3$.
- The number of pins in the lock is $p = 9$. That means there are $3^9 = 19\,683$ possible keys. (Two of those are the master key and the guest key to #723.)
- At the beginning, you have 160 empty key profiles, numbered 1 through 160. (In other words, you have 160 keys, each with the bitting 000000.)
- You have the additional (possibly useless) information that the bittings for the master key and for the guest key to room #723 differ in all positions. (This does not have to be the case for the key to room #719.)

Specific rules for the hard subproblem

- The number of different pin heights is $k = 9$.
- The number of pins in the lock is $p = 10$. (There are $9^{10} = 3\,486\,784\,401$ different keys.)
- At the beginning, you have 23 empty key profiles, numbered 1 through 23.
- Trying to unlock the room #723 is dangerous and requires a lot of time. The command "`try A 723`" may only be sent *as the only command in the file*. That is, the file must contain just a single line with this command. In all other cases this command will be ignored with an error message.

Communication example (for the easy subproblem)

First submission

```
checkin
file 13 012012012
try 13 723
file 13 111111111
try 9999 -47
```

Our response

```
room key: 202102022
key 13 new bitting 012012012
failure
key 13 new bitting 112112112
try failed -- incorrect syntax
```

Second submission

```
try 13 719
restart
checkin
file 13 012012012
try 13 723
```

Our response

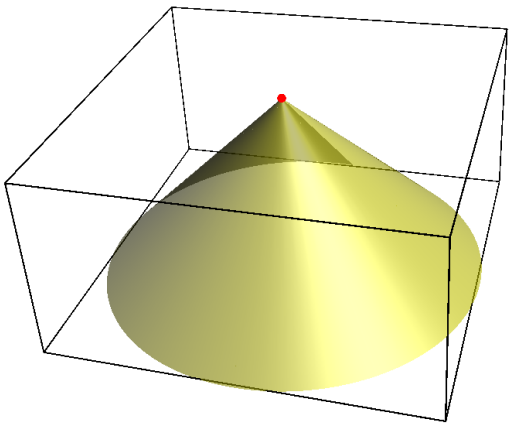
```
failure
restarted
room key: 220012110
key 13 new bitting 012012012
success
```

Notes: The problem statement describes an actual way how master keys for pin tumbler locks are made.
The first four drawings in the problem statement are derived works from pictures in the Wikipedia article "Pin tumbler lock".
Their authors are Wikipedia users Wapcaplet, GWirken and Pbroks13. The drawings are available under the CC BY-SA 3.0 licence.

Problem L: Light in a room

It was a tough year for Per. After finishing his post-doc and going to a lot of interviews, he finally got a job at a university on the other side of the country and had just 2 weeks to find a new apartment. In the end he managed to find one – a really old and dusty apartment. He spent the whole weekend cleaning, throwing away old stuff and repainting the walls. On Sunday evening after 12 hours of work, he couldn't take it anymore. He just rested on the floor, covered in paint and dust.

The room he just painted was empty, only a lamp was mounted to the ceiling. As he switched it on, he noticed that the light rays coming from the lamp formed a cone and covered some parts of the floor and walls. All the work exhausted his body, so he couldn't move, but his mind was still working and wanted to solve some hard problems. He started wondering about the area covered by the light from the lamp.



Obr. 1: An example of light emitted by a lamp.

Problem specification

The room has a horizontal floor, a horizontal ceiling, and vertical walls. The floor is a **convex polygon**. (In the easy data set the polygon is an axes-parallel **rectangle**.) There is a lamp mounted somewhere on the ceiling of the room. The lamp emits a cone of light downwards. The axis of the cone is vertical.

You are given the height of the room, the description of the floor, the location of the lamp and the angle at the apex of the cone of light emitted by the lamp.

Your goal is to calculate the total area of all lit surfaces in the room. (One of these surfaces will always be on the floor. There may be additional lit surfaces on some of the walls.)

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 series of lines. The first line contains space separated floating point numbers l_x, l_y, h, and α. The first two numbers are the horizontal coordinates of the lamp. The third number is the height of the room (and at the same time the vertical coordinate of the lamp). The last number is the angle at which the lamp emits light. (For any ray of light from the lamp, the angle between the ray and the axis of the cone is at most $\alpha/2$.)

The second line of a test case contains the number n of vertices of the floor. Each of the next n lines contains 2 floating point numbers x_i, y_i – the coordinates of the i-th vertex of the floor. The coordinates are given in counterclockwise order. The z-coordinate of the floor is 0.

In the easy data set in each test case we have $n = 4$ and the polygon is an axes-parallel rectangle.

In the hard data set in each test case we have $n \leq 100$ and the polygon is convex.

Output specification

For each test case output one line with one floating point number – the total area of all surfaces that are directly reached by the light of the lamp. Output at least six decimal places. Solutions with a relative or absolute error at most 10^{-6} will be accepted.

Example

input	output
1 5 5 5 91 4 0 0 10 0 10 10 0 10	81.320740643

Problem M: Matrix nightmare

As Obi-Wan would put it: "This isn't the problem statement you are looking for. Move along."

The *double factorial* numbers are the numbers d_i defined by the following recursive formula: $d_0 = 1$, and $\forall i > 0 : d_i = d_{i-1} \cdot i!$. For example, $d_3 = 1! \cdot 2! \cdot 3! = 12$.

Sequences of length n with all elements belonging into the set $\{0, \ldots n-1\}$ are called *limited sequences of order* n. For example, $(0, 2, 0, 1)$ is a limited sequence of order 4. The set of all limited sequences of order n will be denoted $\mathcal{S}_n$.

The *spread factor* of a sequence $A = (a_0, \ldots, a_{n-1})$ is the value $\sigma(A) = \prod_{i=0}^{n-1} \prod_{j=i+1}^{n-1} (a_i - a_j)$.

There is a direct isomorphism between pairs of sequences and sequences of pairs. Formally: Let $A, B \in \mathcal{S}_n$. We can denote their elements as follows: $A = (a_0, \ldots, a_{n-1})$, $B = (b_0, \ldots, b_{n-1})$. The corresponding sequence of pairs $\big((a_0, b_0), \ldots, (a_{n-1}, b_{n-1})\big)$ will be denoted $P_{A,B}$.

Pairs of integers can be ordered lexicographically in the usual fashion: $(a, b) \leq (c, d)$ if either $a < c$, or $(a = c \wedge b \leq d)$. A sequence $P = (p_0, \ldots, p_{n-1})$ of pairs of integers is ordered lexicographically if for all i, $p_i \leq p_{i+1}$. Let $\rho(P) =$ [if P is ordered lexicographically then 1 else 0].

Let M be a $n \times n$ matrix, with rows and columns indexed from 0 to $n-1$. Elements of M will be denoted $m_{r,c}$. A matrix is called a $\mathbb{Z}$-*var matrix* if each element in the matrix is either an integer or a variable. The n-*step traversal weight* of M is the following value:

$$\varphi(M) = \frac{1}{d_{n-1}^2} \cdot \sum_{\substack{A=(a_0,\ldots,a_{n-1}) \\ A \in \mathcal{S}_n}} \sum_{\substack{B=(b_0,\ldots,b_{n-1}) \\ B \in \mathcal{S}_n}} \left(\rho(P_{A,B}) \cdot |\sigma(A)| \cdot \sigma(B) \cdot \prod_{i=0}^{n-1} m_{a_i,b_i} \right)$$

Problem specification

Given is a multivariate polynomial p with integer coefficients. Produce any reasonably small $\mathbb{Z}$-var matrix M such that $\varphi(M) = p$.

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 describing the polynomial. The variables are the letters `a` through `z`, the syntax will be clear from the input file. Each polynomial in the input file contains less than 50 operations (i.e., additions, subtractions and multiplications).

Output specification

For each test case output one matrix in the following format: First its size n, then all its elements in row major order. The elements may be separated by any positive amounts of whitespace. The size of the matrix must not exceed 70. All integers must be between -10^9 and 10^9, inclusive.

If for a given polynomial no such matrix exists, output a single zero instead.

Example

input	output
1 xy	2 1 y x 0