A. The Morning Train

This year witnessed the flawless introduction of the OV-chipcard. Inspired by this success, the Dutch railway company NS is planning to install access gates at the entrance of all their stations. These gates will prevent anyone without a chipcard from entering the station, which will obviously increase safety. However, some people might not have a chipcard yet and may need to buy a ticket first.

Although buying a ticket can take up to thirty minutes, most people that need a ticket can do so in no more than a few minutes. However, there are only a limited number of ticket machines, so lines inevitably form. As people traveling by train tend to be well-mannered no one will skip the line; when it is their turn each person will use the first available ticket machine. A person with a ticket or an OV-chipcard can go through an access gate. Only one person can pass an access gate at a time; doing so takes one second. There is also a limited number of gates, so during rush hour it's still a struggle to get through.

To evaluate their new system, NS wants to simulate a typical work day where everyone needs to catch the morning train. Anyone that has passed the access gates at eight o'clock sharp is presumed to catch the train. The simulation starts at four in the morning, with an empty train station.

Input

On the first line of the input is a positive integer, the number of test cases. Then for each test case:

- A line with two positive integers $n_g, n_t < 10$, the number of access gates and ticket machines, respectively.
- A line with a single non-negative integer $n < 10^6$, the number of travelers.
- n lines, each containing two non-negative integers $t_i < 10^9$ and $d_i \leq 1800$, the arrival time (in seconds after four o'clock) and the time spent at a ticket machine (in seconds, zero means no ticket needed). The lines are sorted on the arrival time (in non-decreasing order). Two people with the same arrival times will get in line in the order in which they appear in the input.

Output

For each test case:

- One line containing a single integer, the number of people who will miss the eight o'clock train.

Example

Input	Output

2
1 1
4
14398 0
14398 0
14399 0
14399 0
1 1
2
14390 9
14399 0	

B. Forklift

This year witnessed the flawless introduction of the OV-chipcard. Thanks to this amazing system many aspects of public transit can now be automated. One of the more innovative changes, intended to reduce the number of fare dodgers, but also to accomodate the elderly and speed up the boarding process, is requiring passengers to travel in boxes. These are more easily accessible than a train and can accomodate people in wheelchairs, or multiple children, in one box!

Gerard drives a forklift truck in the train's compartments to get everyone in the right place, a responsible, well-paid job. All day long he is transporting boxes with passengers to and from the train, leaving them wherever he can. Before departure he has to tidy up and make sure all stacks are equally high, otherwise the train may derail. Since the train should leave on time, Gerard wants to use the shortest route along the stacks that will enable him to equalize them all.

The train has only one long line of equally-spaced stacks (there is half a meter between stacks), Gerard is initially in front of the first stack. Each stack has zero or more equally dimensioned boxes (each box is 1.5 × 1.5 × 1.5 meters, allowing for people and objects of upto 2.60m in length). When he is in front of a stack he can load or unload as many boxes as he likes. How long is the shortest route is that Gerard can take to make sure that all stacks are equally high? Gerard does not need to end where he started.

Input

On the first line of the input is a positive integer, the number of test cases. Then for each test case:

- A line with a single positive integer $n < 10^6$, the number of stacks.
- A line with n non-negative integers $b_i < 10^9$, the number of boxes in the ith stack. The total number of boxes is divisible by n, and is at most 10^9.

Output

For each test case:

- One line containing a single integer, the length in meters of the shortest route that Gerard can take to make sure that all stacks are equally high.

Example

Input	**Output**

`3`
`3`
`3 0 0`
`3`
`0 0 3`
`5`
`0 4 0 11 0`
	`4`
	`8`
	`16`

C. Tilt Maze

This year witnessed the flawless introduction of the OV-chipcard. To engender good will amongst travelers, the Groningen bus company Q-Buzz decided to hand out fun gifts to travelers that use the OV-chipcard. Their marketing devision decided to use small games of manual dexterity: mazes in which the player attempts to roll a small metal ball from the entry to the exit of the maze. The maze is supposed to symbolize the city of Groningen, and the ball a traveler who purposefully moves about within it. When the player has found a path from entry to exit, the maze can be solved in a straightforward manner: simply roll the ball along the path, and delicately balance the maze whenever the ball approaches an intersection to ensure it rolls along the intended path.

Unfortunately, players soon discovered that playing this game on the bus proved very challenging. As the bus constantly shifts and shudders, it's not possible to balance the maze finely enough to control the ball's direction at intersections!

Herbert, a student at the Rijksuniversiteit Groningen, thinks he can solve the puzzle despite this handicap, using the fact that each maze lies on a rectangular grid. He has come up with the following technique: rather than attempt to control the direction of the ball's movement at each intersection, Herbert tilts the maze in one direction and waits until the ball comes to a full stop before reorienting the tilt. To avoid nondeterministic behavior at T-junctions, Herbert tilts the maze in a direction that is not parallel to the gridlines so the ball always clearly rolls in one direction.

Herbert settles on 8 directions of tilt:

- Upwards with a slight slant to the left
- Upwards with a slight slant to the right
- Right with a slight slant upwards
- Right with a slight slant downwards
- Downwards with a slight slant right
- Downwards with a slight slant left
- Left with a slight slant downwards
- Left with a slight slant upwards

The maze consists of a grid of blocking squares and passable squares. When in a square, the ball rolls to the next square in the primary direction if passable, or in the direction of the slight slant when that's passable and the primary direction is blocked. If both the primary direction and the direction of the slight slant are blocked, the ball comes to a rest and falls into the target (if this is the target square) or Herbert can choose a new direction of tilt. The maze is enclosed in plastic so the ball cannot roll past the edge of the maze (all squares outside of the maze can be considered blocked). The ball cannot enter the target while rolling.

Can Herbert solve the maze using this strategy, and if so, how many tilts will he need?

Input

On the first line of the input is a positive integer, the number of test cases. Then for each test case:

- A line with two positive integers $h, w < 10^3$, the height and width of the maze.
- h lines each consisting of w characters, a description of the maze. The following characters are used:

  “`A`” This square is the entry square of the maze.
  “`B`” This square is the exit square of the maze.
  “`X`” This square is blocked.
  “`.`” This square is passable.

Output

For each test case:

- One line containing a single integer, the minimum number of tilts Herbert needs to complete the maze; or “`no solution`” if Herbert cannot solve the maze using his strategy.

Example

Input	**Output**
2 4 5 .X..X ...A. .X..X X.BXX 3 4 XXXX .AB. X..X	2 no solution

D. Downhill

This year witnessed the flawless introduction of the OV-chipcard. Most commonly the OV-chipcard is used as a toy (4 years and older) or as a bookmark, in some cases it is even used to open locked doors. This is why Paul, a traveling salesman from New York, just knew that he had to get his hands on as many of these versatile tools as possible.

The streets in Manhattan, where Paul works, form a regular grid on inclined plane. There are numbered streets running east-west and numbered avenues running north-south. Paul has taken great care to plan an efficient route through the city, visiting all the houses of potential clients.

This Monday morning disaster struck: his car broke down. It still runs, but just barely. He can go no faster than twenty miles per hour, and only downhill.

Fortunately, Manhattan was built on a hill and Paul lives at the highest point, at the intersection of 1st street and 1st avenue. So he is still able to drive south to 2nd street, 3rd street, etc.; or east to 2nd avenue, 3rd avenue, etc.. His regular route is no longer an option, since then he would also have to drive uphill. Paul would still like to visit as many houses as possible, so he can earn enough money to fix his car.

Input

On the first line of the input is a positive integer, the number of test cases. Then for each test case:

- A line with a single natural number [licht]$n < 10^4$[¬licht]$n < 10^6$[all], the number potential clients Paul wants to visit.
- n lines, each containing two positive integers $s_i, a_i < 10^9$, the street and avenue number of the intersections where a house is located. All houses are located at different intersections.

Output

For each test case:

- One line containing a single integer, the maximal number of houses Paul can visit.

Example

Input	Output

2
9
1 2
1 3
1 4
2 2
2 3
2 4
3 2
3 3
3 4
6
3 4
4 2
2 1
6 5
1 6
5 3	

E. Dining Philosophers

This year witnessed the flawless introduction of the OV-chipcard. Word of this has even traveled beyond this life and into the next.

The first circle of hell is reserved for the unbaptised and the virtuous pagans. Among them are most of history's famous philosophers.

So it comes to pass that Immanuel Kant, Socrates, John Locke, René Descartes and Søren Kierkegaard decide to hold a dinner party, to ponder these new troublesome developments in the Netherlands. They have seated themselves at a round table. In front of each philosopher is a dinner plate, between each pair of adjacent plates is a fork. In the center of the table is a large tray of chicken drumsticks, freshly roasted above the fires of hell.

These philosophers are among the most civilised men in history, but even the most decent folks soon degrade to a band of complete idiots at the sight of chicken drumsticks. To prevent this horrible outcome the philosophers have decided on a strict protocol for the dinner. Before making any move on the delicious food a philosopher must hold both of the forks next to his plate. Only then may he pick up and eat exactly one drumstick from the tray. After eating this drumstick, which takes approximately 1 minute, the philosopher is to put down both forks and has the opportunity to share some deep insight with the other participants, which takes only 10 seconds. After that he may try to lift his forks again to resume eating.

Upon agreeing on the protocols each of the philosophers immediately picks up the fork at his right hand side, hoping to get the opportunity to impress the others with his wisdom (not to mention being able to savour the delicious meat he has been staring at for some time now). The philosophers quickly realise the gravity of their dilemma: their protocol is clearly flawed, yet none of them feels compelled to give up their chance to eat.

Kant decides that it is his categorical imperative to do something about this situation. He puts down the fork in his right hand, to give someone else a chance to eat. Socrates, who sits next to Kant, immediately takes advantage of this oportunity and picks up the fork. Since he now holds two forks he takes a single drumstick and eats it.

Input

On the first line of the input is a positive integer, the number of test cases. Then for each test case:

- A line with a single positive integer i, $i < 10^6$, the number of drumsticks on the table.

Output

For each test case:

- One line containing a single integer, the number of drumsticks Immanuel Kant eats.

Example

Input	Output
2 13 65	

F. Settle the Bill

This year witnessed the flawless introduction of the OV-chipcard. Unfortunately the new OV-chipcard does not allow for multiple people to travel on one card. This is bad news for the Computer Science department as they frequently travel together, sharing the costs. They have not settled the last few travel bills yet, expecting that in the long run everything would cancel out. However, because of the economic crisis they agree to settle the bills now, once and for all.

As a certain kind of laziness comes natural to every good computer scientist: they naturally want to minimize the number of transactions. After a few all-nighters Professor Bakker has figured out that $n-1$ is an upper bound for the number of transactions needed to settle debts between n persons. His general theory of settlements is based on the assumption that the number of transactions over x days is described by $t(x) = \big(x^{(n-1)} - 1\big)/\big(x - 1\big)$. In the optimal case where all transactions are settled in a single day this gives $\lim_{x\to 1}\big(x^{(n-1)} - 1\big)/\big(x - 1\big) = n - 1$.

Some people think they can do better though. Can they?

Input

On the first line of the input is a positive integer, the number of test cases. Then for each test case:

- A line containing two positive integers $n < 20$ and $m < 400$, the number of people and the number of debts respectively.
- m lines containing three integers $0 \le f_i, t_i < n$ and $a_i < 10^7$, the zero-based index of the person who owes money, the zero-based index of the person to whom he owes money, and the amount of money owed in Euros.

Output

For each test case:

- One line containing "`tight`" if Professor Bakker's bound is tight (exactly $n-1$ transactions are needed) in this case, or "`loose`" if the bound is not tight (in this case).

Example

Input	**Output**

2
3 2
0 1 20
1 2 10
4 2
0 1 20
2 3 20	

G. Detector Dodging

This year witnessed the flawless introduction of the OV-chipcard. As a result Monique has vowed to dodge fares as often as she can. However, in order for her not to be caught she does have to check in with her OV-chipcard whenever she takes the train. The trick is to avoid checking out at the destination, checking out much closer to home instead.

To prevent suspicion Monique always walks off the platform in a straight line. However, this Saturday morning Monique found herself on the train platform in Groningen, on her way to the IWI Programming Contest. And unfortunately for her they had set up an enormous amount of more or less randomly positioned detectors. She tried making a clean escape, but she could not avoid brushing against a detection rod.

The next time Monique will be prepared! Can you help her by finding the size (in degrees) of the largest arc over which there are no rods?

Input

On the first line of the input is a positive integer, the number of test cases. Then for each test case:

- A line with a single positive integer $n < 10^6$, the number of detection rods.
- n lines, each containing two integers, the x and y coordinates of each rod relative to Monique's current location. $|x|, |y| < 10^6$.

Output

For each test case:

- One line containing the size of the largest arc (in degrees) over which there are no rods. The answer should be between 0 and 360, and have an error of at most 10^{-6} degrees.

Example

Input	**Output**

4
3
2 2
2 -2
-2 -2
8
4 -1
4 2
1 4
-2 4
-4 1
-4 -2
-1 -4
2 -4
8
4 2
4 -1
1 4
-2 4
-4 2
-4 -2
-1 -4
2 -4
8
-9 -1
-4 3
-4 -3
-1 -6
2 4
3 -4
4 -2
7 -1	

H. Java Decompiler

This year witnessed the flawless introduction of the OV-chipcard. The developers claimed that the OV-chipcard is secure, because it is programmed in Java bytecode, which is impossible to decipher. However, security experts have been able to reverse engineer this code and have thereby compromised the chipcard's security.

A Java bytecode program consists of a sequence of instructions. The virtual machine that executes these instructions maintains a stack of integers, and instructions can push and pop values from the stack. The possible instructions are:

[licht]

instruction	**stack before**	**stack after**	**side effect**
`iconst <INT>`	$\ldots$	$\ldots, INT$	
`iload <VAR>`	$\ldots$	$\ldots, VAR$	
`istore <VAR>`	$\ldots, a$	$\ldots$	$VAR = a;$
`iadd`	$\ldots, a, b$	$\ldots, a + b$	
`isub`	$\ldots, a, b$	$\ldots, a - b$	
`imul`	$\ldots, a, b$	$\ldots, a * b$	
`idiv`	$\ldots, a, b$	$\ldots, a/b$	

[¬licht]

instruction	**stack before**	**stack after**	**side effect**
`iconst <INT>`	$\ldots$	$\ldots, INT$	
`iload <VAR>`	$\ldots$	$\ldots, VAR$	
`istore <VAR>`	$\ldots, a$	$\ldots$	$VAR = a;$
`iadd`	$\ldots, a, b$	$\ldots, a + b$	
`isub`	$\ldots, a, b$	$\ldots, a - b$	
`imul`	$\ldots, a, b$	$\ldots, a * b$	
`idiv`	$\ldots, a, b$	$\ldots, a/b$	
`ineg`	$\ldots, a$	$\ldots, -a$	

Here `<VAR>` is a single Java identifier, `<INT>` is an integer.

Your task is to convert a list of bytecode instructions into a readable Java program. [licht]The program should include parentheses around all applications of binary operators.[¬licht]The program should only use parentheses when they are strictly required to produce the given bytecode.[all] Put a single space around all binary operators.

To be precise, the program should consist of one or more ⟨STMT⟩s from the following grammar:
[licht]

$$\begin{aligned}\langle\text{EXP}\rangle &::= \langle\text{INT}\rangle \;\mid\; \langle\text{VAR}\rangle \;\mid\; \text{“(”}\;\langle\text{EXP}\rangle\;\langle\text{BINOP}\rangle\;\langle\text{EXP}\rangle\;\text{“)”}\\ \langle\text{BINOP}\rangle &::= \text{“ + ”} \;\mid\; \text{“ - ”} \;\mid\; \text{“ * ”} \;\mid\; \text{“ / ”}\\ \langle\text{STMT}\rangle &::= \langle\text{VAR}\rangle\;\text{“ = ”}\;\langle\text{PLUSEXP}\rangle\;\text{“;”}\end{aligned}$$

[¬licht]

$$\begin{aligned}
\langle\textsc{plusexp}\rangle &::= \langle\textsc{timesexp}\rangle \;\mid\; \langle\textsc{plusexp}\rangle\,\langle\textsc{plusop}\rangle\,\langle\textsc{timesexp}\rangle \\
\langle\textsc{timesexp}\rangle &::= \langle\textsc{unaryexp}\rangle \;\mid\; \langle\textsc{timesexp}\rangle\,\langle\textsc{timesop}\rangle\,\langle\textsc{unaryexp}\rangle \\
\langle\textsc{unaryexp}\rangle &::= \langle\textsc{primexp}\rangle \;\mid\; \text{“}\texttt{-}\text{”}\,\langle\textsc{primexp}\rangle \\
\langle\textsc{primexp}\rangle &::= \langle\textsc{int}\rangle \;\mid\; \langle\textsc{var}\rangle \;\mid\; \text{“}\texttt{(}\text{”}\,\langle\textsc{plusexp}\rangle\,\text{“}\texttt{)}\text{”} \\
\langle\textsc{plusop}\rangle &::= \text{“ }\texttt{+}\text{ ”} \;\mid\; \text{“ }\texttt{-}\text{ ”} \\
\langle\textsc{timesop}\rangle &::= \text{“ }\texttt{*}\text{ ”} \;\mid\; \text{“ }\texttt{/}\text{ ”} \\
\langle\textsc{stmt}\rangle &::= \langle\textsc{var}\rangle\,\text{“ }\texttt{=}\text{ ”}\,\langle\textsc{plusexp}\rangle\,\text{“}\texttt{;}\text{”}
\end{aligned}$$

Input

On the first line of the input is a positive integer, the number of test cases. Then for each test case:

- A line with a single positive integer $n < 10^6$, the number of assembly instructions that follow.
- n lines, each containing an assembly instruction. Together the instructions form a number of Java statements.

Output

For each test case:

- The equivalent program in Java[¬licht], without any superfluous parentheses[all].

Example

Input	Output

[licht]
3
6
iload x
iconst 2
iadd
iconst 4
imul
istore y
6
iload x
iload y
iload z
iadd
iadd
istore x
6
iconst 42
istore the_answer
iconst 1
iconst 0
idiv
istore the_question
[¬licht]
3
6
iload x
iconst 2
iadd
iconst 4
imul
istore y
6
iload x
iload y
iload z
iadd
iadd
istore x
4
iconst 42
ineg
ineg
istore the_answer
[¬licht]
y = (x + 2) * 4;
x = x + (y + z);
the_answer = -(-42);
	[licht]

I. Compression

This year witnessed the flawless introduction of the OV-chipcard. To induce acceptance of the new system by the public, transport companies introduced an innovative SMS help-service. Unfortunately, the helpdesk's 14.4k baud modem was quickly overwhelmed. To improve the availability of the helpdesk system, the public transport companies want to compress their data traffic. This data mostly consists of text, so they need a compressor that is optimized for text.

Your goal is to write the best compressor (and decompressor) that you can possibly think of. However, since the helpdesk manager is quite desperate by now, even compressing the input by a single character will (partially) satisfy him.

The input consists of one line of English text, consisting of the following characters: the lower case letters a-z, the upper case letters A-Z, digits 0-9, periods '.' and spaces ' '. To ensure compatibility with existing systems the output can only contain those same kinds of characters.

Scoring

A correct implementation that reduces the size of the test input is worth half a point. An implementation whose compression ratio is ranked as better than or equal to at least half of the other contestants' is worth one point. The implementation(s) with the best compression ratio are worth one and a half points.

Furthermore, your (total) penalty time will be multiplied by the compression factor (defined as compressed size/original size). Note that you may submit more than one correct implementation and only the first one counts towards your penalty time, while the last one counts for everything else.

Apart from the usual rules an implementation is only considered correct if it

- Actually compresses the input *on average* (not all lines need to be compressed).
- Can reproduce each original input line *exactly* after compression and subsequent decompression.
- Does *not* use any form of I/O other than reading from stdin and writing to stdout.
- Does *not* use a library for compression. (A base64 encoding library would be permissible.)

Running the compressor

The `test-compressor` program can be used to test a compressor. It reads a text from the standard input. Then, for each line, it runs the compressor once to compress that line, and once to decompress the compressed line. Finally, it compares the result to the original and outputs the compression ratio. The `test-compressor` program also verifies that only valid characters appear in the input and output.

```
Usage: test-compressor COMPRESSOR < INPUT
```

Input

For each run of the program the input consists of exactly two lines.

- On the first line of the input is a single word indicating the mode of the program, either “`compress`” or “`decompress`”.
- On the second line is the string to be compressed/decompressed ($0 < \text{length} < 10^6$). This string satisfies the regular expression `[a-zA-Z0-9.  ]+`.

Output

- One line containing the (de)compressed string.