IPSC 2009

problems and sample solutions

Pathfinding

Authors

Problemsetter: Michal Forišek
Task preparation: Michal Forišek

Problem statement

The first problem of this practice session is pretty easy and standard – you have to find your way through a maze.

Problem specification

You are given a description of a maze. Produce a short sequence of moves that exits it.

Input specification

The first line of the input contains two positive integers R and C – the number of rows and columns in the maze. Rows are numbered 1 to R, and columns 1 to C.

R lines follow. Each of them contains C integers. The c-th integer in the r-th row is 0 if the cell at (r, c) is free, and it is 1 if the cell at (r, c) is a wall.

In the final line are the coordinates of the cell where you start.

Output specification

The output shall contain any sequence of at most 200 moves. The input will be such that at least one such sequence will exist.

All moves must be valid (i.e., may not crash into a wall), and the last move must be the only move that exits the maze.

Describe the moves using letters `U` for up, `D` for down, `L` for left and `R` for right. (Up is the direction in which row numbers decrease, left is the direction in which column numbers decrease.)

You may separate the moves in the output by whitespace.

Flash version

For your convenience, we provide Flash applications in which you can traverse the given mazes using your keyboard. If you successfully exit the maze, the application will print your sequence of moves, you can simply copy it and submit it.

Example

input

```
7 9
1 1 1 1 1 1 1 1 1
1 0 0 0 0 0 0 0 1
1 0 1 1 1 1 1 0 1
1 0 1 0 0 0 1 0 0
1 0 1 1 1 0 1 0 1
1 0 0 0 0 0 1 0 1
1 1 1 1 1 1 1 1 1
6 2
```

output

```
R L U U U U R R R
R R R D D D U R R
```

Solution

The main purpose of this problem was to give the teams a chance to check that they can run a simple Flash application. The easiest way how to solve this task was to play with the application and cross the maze on your own. For the hard data set, you had to use an (almost) optimal path, with no backtracking, but this was still reasonably easy.

Alternately, a simple breadth-first search can be used to find the shortest path through the maze.

Quick search

Authors

Problemsetter: Michal Forišek
Task preparation: Michal Forišek

Problem statement

The second problem in this practice session is based on an actual pen and paper puzzle. The puzzle, as you'll soon discover, is quite tricky – and harder than it might seem.

Problem specification

In this task you will be given a rectangle divided into several polygons. All of these polygons are distinct, except for two that are identical. Your task is to find the two identical ones.

Note: We call two polygons identical iff one of them can be obtained from the other by a combination of translations and rotations. Note that it is **not** allowed to use axis symmetry – i.e., if a polygon is just a mirror image of another one, they are considered distinct for the purpose of this problem.

Input specification

The input is an image of the rectangle, including labels for rows and columns.

Output specification

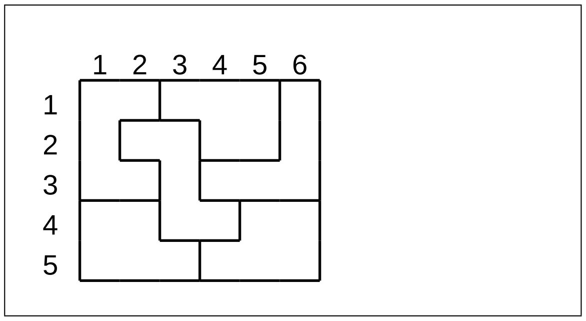
Each polygon in the input is uniquely identified by the coordinates (r, c) of its upper left corner. More precisely, to describe a polygon, we first find the smallest row r in which the polygon contains at least one cell, and then pick the smallest possible column c it contains in this row.

Once you locate the two polygons, output the coordinates (r_1, c_1) and (r_2, c_2) of their upper left corners.

To make the output unique, we require that $r_1 \leq r_2$, and if $r_1 = r_2$, then $c_1 < c_2$. I.e., first output the polygon that starts earlier.

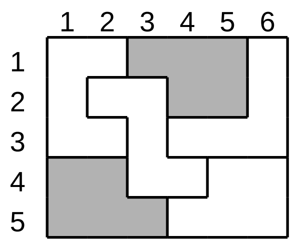
Example

input



output

```
1 3
4 1
```



Note that the polygon at (4,5) is not identical to the highlighted pair.

Solution

The easy set can easily be solved by hand. For the hard set it is probably faster to convert the input into a text-based format such as PNM (ImageMagick may come in handy), flood fill the polygons, and then compare each pair.

Reverse quick search

Authors

Problemsetter: Michal Forišek
Task preparation: Michal Forišek

Problem statement

After creating problem Q, we thought it would be entertaining to put you into problemsetters' shoes for a moment. In this task you will find out a way to create test data for problem Q.

Problem specification

You are given integers R, C, and A such that A divides RC.

Your task is to take a $R \times C$ rectangle and divide it into polygons. Each of the polygons must consist of exactly A unit squares. All polygons must be distinct, except for two polygons that must be identical.

Note: We call two polygons identical iff one of them can be obtained from the other by a combination of translations and rotations. Note that it is **not** allowed to use axis symmetry – i.e., if a polygon is just a mirror image of another one, they are considered distinct for the purpose of this problem.

Input specification

The input contains a single line with the three integers R, C, and A.

Output specification

After you divide the rectangle into $N = RC/A$ polygons, number them from 1 to N (in any order you wish).

Output R rows with C integers in each of them. The c-th integer in the r-th row must be the number of the polygon that contains the cell (r, c).

Any valid output will be accepted.

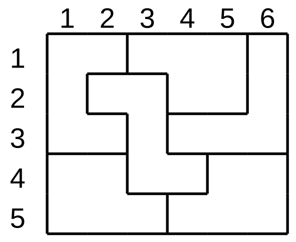
Example

input

```
5 6 5
```

output

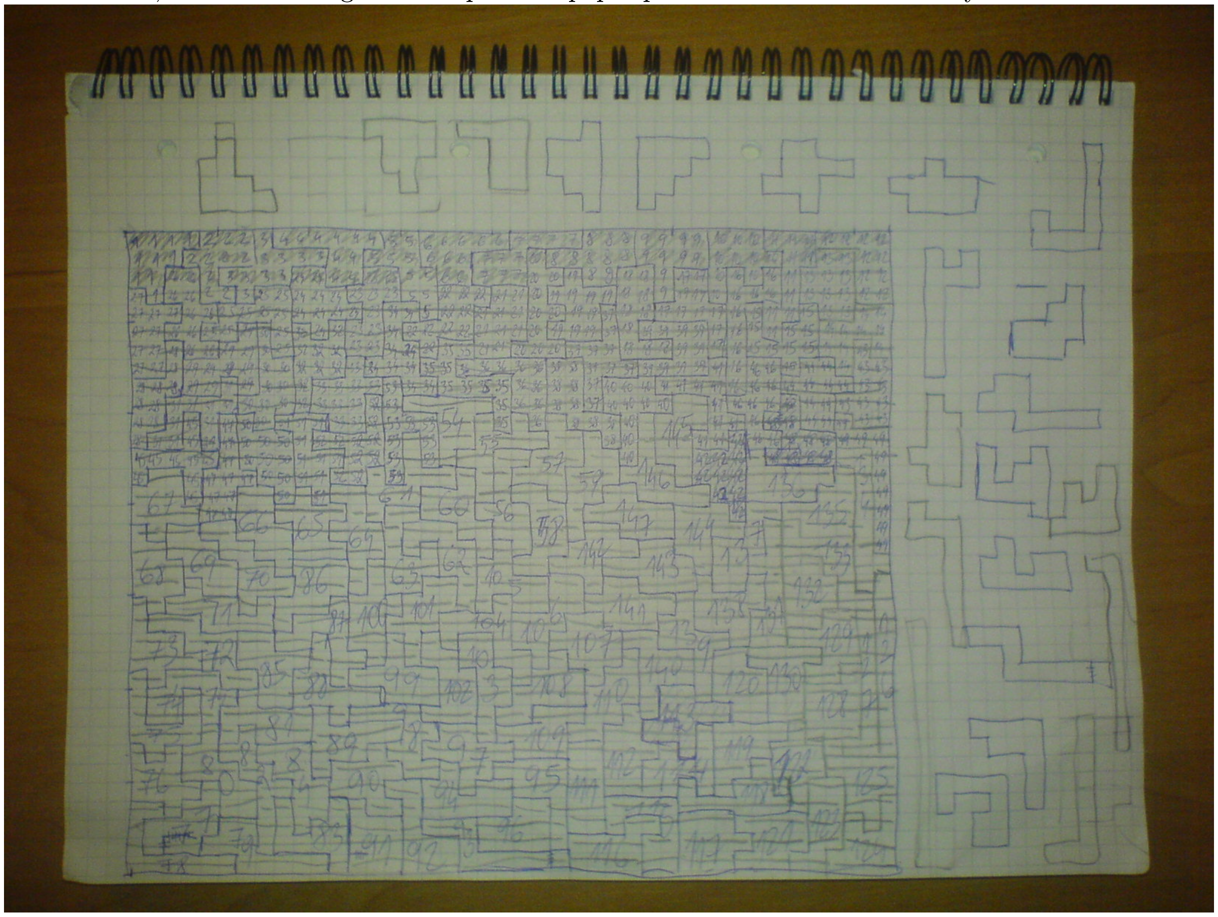
```
1 1 2 2 2 3
1 4 4 2 2 3
1 1 4 3 3 3
6 6 4 4 5 5
6 6 6 5 5 5
```



Solution

Once again, the easy set can easily be solved by hand. For the hard set a greedy approach works. We generated the inputs for Q using a randomized backtracking algorithm – we always find the first cell that does not belong to any polygon, generate a random polygon that contains it, check that the remaining area is still connected, and if yes, we continue recursively.

Of course, those who are good with pen and paper puzzles can solve even R2 by hand:



Arithmetics for dummies

Authors

Problemsetter:	Jozef Šiška
Task preparation:	Jozef Šiška, Michal Nánási

Problem statement

While browsing aimlessly, Peter stumbled upon an old riddle he used to solve on his calculator when he was still young. It was the kind of a riddle where you punch in a bunch of numbers and operators into a simple pocket calculator and then turn it upside down to get the answer:

These come in many different sizes but they are always exactly one foot long. Answer: $103 \times 103 \times 5$.

What are made of ice to keep people warm? Answer: $50 \times 40 \times 250 + 791$.

After a few minutes he found a large amount of such riddles and full of excitement he went to solve them. He turned his computer screen upside down…

… only to find out that he does not have a reasonable calculator program installed on his computer.

Problem specification

You are given multiple sequences of button presses of a simple pocket calculator that consist of digits and arithmetic operators. For each such sequence find the number it would produce on a pocket calculator's display.

Note that the pocket calculator evaluates the operators in the order in which they are given. (I.e., there is no operator precedence.) Assume that the display of the calculator is large enough to show the result, and that its memory is sufficient to store all intermediate results.

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 represents one sequence of button presses for a pocket calculator. The sequence consists of non-negative integers and arithmetic operators and ends with an equal sign. It may also contain spaces to improve readability.

The operator `/` represents integer division, rounded down. You may assume that no test case contains division by zero and that in all test cases all intermediate results are non-negative.

The easy data set only contains the operator `+`.

Output specification

For each sequence from the input file output the number that would be displayed on the calculator.

Example

input

```
4

1 + 1 * 2 =

29 / 5 =

103 * 103 * 5 =

50 * 40 * 250 + 791 =
```

output

```
4
5
53045
500791
```

The first test case shows that there is no operator precedence.
The second one shows that integer division always rounds down.
The last two outputs are the answers to the two riddles in the problem statement:
"shoes" (53045 upside down), and
"igloos" (500791 upside down).

Solution

There are two ways to solve this problem.

The more annoying way is to write your own program that parses the input and does the math.

The advantage of this approach is that the missing operator preference makes it possible to read the numbers and operations one at a time and execute them. This gets even easier in the case of interpreted languages that have some way of evaluating expressions in strings.

Still, probably a better way is to pass the expressions into some existing tool / calculator (such as `bc` in linux).

For the hard test case, we may need to force the tool to evaluate the expressions in correct order. One possibility is to add parentheses to the expression in such a way that the operator preference would not matter. This means adding a closing parenthesis before each operator and an equal number of opening parenthesis to the beginning of the line. This can be achieved directly with a `sed` script (though it may be a bit complicated) or by the combination of `sed` and `grep`.

Another solution is to swap the numbers and operators and use a calculator that uses the reverse polish notation such as `dc`. This can be easily accomplished by a simple regexp: `s,([-+*/]) *([0-9]+),\2 \1,g`. We also need to change the equal sign into a `p` command that prints the result after each line.

Bouncing balls

Authors

Problemsetter: Michal Forišek
Task preparation: Milan Plžík, Michal Forišek, Jakub Kováč

Problem statement

"Behold, my queen", said the jester, "the great Bouncing Ball Bowl!" The queen boredly waved her hand and sarcastically replied: "Let the fun begin!". And the fun begun! The jester spoke a magic word and all the colorful balls in his bowl started to roll and bounce, creating interesting pictures.

The queen watched vividly for a few minutes, but then she started to be bored again. "Just wait a moment, Your Majesty, in a minute they'll. . . " started the jester, but the queen interrupted: "I'm a queen! I don't want to wait! Can't you just fast forward it or something?"

Problem specification

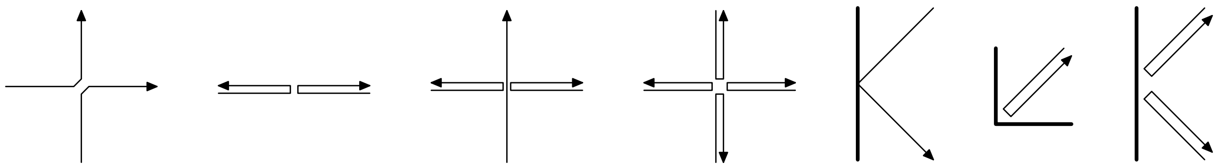
The jester's box is an $X \times Y$ rectangle. The rectangle contains N small balls. At any moment, each ball is travelling at the same speed in one of the four diagonal directions.

The movement of the balls is continuous and for the purpose of this problem we may consider them to be points. When two or more balls meet, they bounce in a way described below.

Your task is to determine the state of the box at given moments in time.

Bouncing specification

Bouncing does not change the speed of the balls. Following images show how the balls bounce off each other, and also off walls. Each image can be rotated arbitrarily. For example, the first image shows that whenever two balls meet at a right angle, they bounce and depart at a right angle again. One particularly tricky case is shown in the third image.



Input specification

The input starts with a line containing the dimensions X and Y of the box. We will use a coordinate system with axes parallel to the sides of the box, $(0, 0)$ at one of the corners and (X, Y) at the opposite corner.

The second line contains the number of balls N.

Each of the next N lines contains four integers x, y, v_x, v_y, where (x, y) are the coordinates of one ball at time 0 and (v_x, v_y) is its current velocity vector. (Each ball will be strictly inside the box and for each ball both v_x and v_y will be equal to ± 1. No two balls will start at the same place.)

The following line contains the number of queen's requests M.

On the last line there are M numbers $t_1, \ldots, t_M$ – the points in time the queen wants to see.

Output specification

As a solution to this problem, we expect a file with M blocks, with the i-th block describing the situation at time t_i.

Each block must contain N lines, and each line must contain the x and y coordinates of one ball. The balls in each block must be sorted – primarily by to their first, secondarily by their second coordinate at that point in time.

You may output an empty line between the blocks.

Example

input

```
6 4
4
1 2 1 1
5 2 1 1
2 1 1 -1
3 1 -1 -1
1
4
```

output

```
1 3
3 2
5 2
6 3
```

Note that the balls that start at (2,1) and (3,1) bounce off each other at a non-integer point in time.

Solution

Simulation of bouncing balls might be slightly problematic, in both implementation and computational aspects. Fortunately, there are several interesting restrictions given in the input specification, especially the fact that all balls have the same speed and they move only diagonally.

In fact, thanks to this setting we can simply ignore collisions of the balls with each other. Visually, we would get exactly the same view if we allowed the balls to pass through each other.

Hence we only have to handle the collisions between each ball and the walls. To do this, it is enough to note that the x coordinate of each ball is periodic with a period $2X$, and the y coordinate is periodic with a period $2Y$.

Thus we can simply process the balls one by one, for each of them we compute its final position in $O(1)$, and then we just sort and output these.

Cryptic punchcards

Authors

Problemsetter: Michal Forišek
Task preparation: Michal Forišek, Vladimír Koutný

Problem statement

Punch cards are an important part of programming history. Quoting Wikipedia, *a punch card is a piece of stiff paper that contains digital information represented by the presence or absence of holes in predefined positions. Now almost an obsolete recording medium, punched cards were widely used throughout the 19th century for controlling textile looms and in the late 19th and early 20th century for operating fairground organs and related instruments. It was used through the 20th century in unit record machines for input, processing, and data storage. Early digital computers used punched cards as the primary medium for input of both computer programs and data, with offline data entry on key punch machines.*

One of the most famous types of punch cards were IBM's 80-column punch cards. Their width is the reason why even nowadays many terminals default to 80 columns of characters.

Various encodings were used to punch the data onto cards. One of the most famous ones is EBCDIC. Quoting the Jargon File, *It exists in at least six mutually incompatible versions, all featuring such delights as non-contiguous letter sequences and the absence of several ASCII punctuation characters fairly important for modern computer languages. [...] Hackers blanch at the very name of EBCDIC and consider it a manifestation of purest evil.*

You probably already know where this leads to.

Problem specification

In this problem you will be given image files showing some punch cards. Imagine that you are several decades in the past. You actually have the same data punched on real paper punch cards. You take them and feed them into the machine...

Your task is to find out what the machine (most probably) read, and use that to determine the output you send us.

Input specification

The input is a set of image files numbered in order in which they should be fed into the machine.

Output specification

Send us a text file containing a single line with a single positive integer.

Solution

As the problem statement suggested, the input contains 80-column punch cards and the encoding used is EBCDIC.

You can read about this encoding at `http://www.cs.uiowa.edu/~jones/cards/codes.html`, and at `http://www.kloth.net/services/cardpunch.php` you can find a service that allows you to create your very own punched card images.

However, getting to the correct answer required not only decoding the information, but also a deeper knowledge of times long past.

Easy data set

In the easy input, three of the cards make sense. They contain the following text:

```
THE ANSWER IS EQUAL TO TWENTY THREE TIMES ONE HUNDRED AND SEVENTEEN THOUSAND
TWO HUNDRED AND ELEVEN PLUS THIRTEEN PLUS FORTY SEVEN MILLION THREE THOUSAND
EIGHT HUNDRED AND FIFTY NINE
```

However, the third card does not contain any useful information – and this should immediately be suspicious.

The third card is a lace card (see `http://en.wikipedia.org/wiki/Lace_card`). These cards usually caused the reader to jam. This is where the formulation "*(most probably)*" from the problem statement comes to play. If we took these four cards and feed them into a card reader, the most probable outcome would be that it reads the first two cards and then gets jammed on the third one.

Hence the text it read is just

```
THE ANSWER IS EQUAL TO TWENTY THREE TIMES ONE HUNDRED AND SEVENTEEN THOUSAND
TWO HUNDRED AND ELEVEN PLUS THIRTEEN PLUS FORTY SEVEN MILLION THREE THOUSAND
```

giving us the answer 49 698 866.

Hard data set

When we decode the cards in the hard input, we get a program in Fortran:

```
      INTEGER COUNT
      COUNT = 47
                                                        COUNT = COUNT + 47
COUNT = COUNT + 47
      COUNT = COUNT + 47
     47 * 47
********** AND NOW WE JUST OUTPUT THE ANSWER **********
      WRITE(*,*) 'THE ANSWER IS ', COUNT
      END
```

Again, this program hides several nasty surprises. Getting a Fortran compiler (such as gfortran) and running it is much better than trying to understand it.

The tricks include:

- Columns 73 to 80 are ignored. The reason: when programs were entered on punch cards, these columns often contained serial numbers of cards. Hence row 3 only reads `COUNT = COUNT + 4`.
- For backwards compatibility, there are multiple ways how to make comments. Line 7 shows the modern one (a line starting with a `*`), line 4 is the original one (a line starting with a `C` in the first column). Hence line 4 is ignored.

- If there is a character in column 6, it means that this row is a continuation of previous one. Additionally, Fortran ignores whitespace. Hence lines 5 and 6 together are equivalent to the single statement `COUNT = COUNT + 477 * 47`.

Thus this program outputs 22 470.

Don't worry about wrong answers

Authors

Problemsetter:	Jakub Kováč, Lukáš Poláček
Task preparation:	Jakub Kováč, Lukáš Poláček

Problem statement

Every year many teams competing in the IPSC have huge penalty times due to lots of incorrect submissions – and also due to the fact that they failed to send us a nice postcard.

We know that many of the tasks are hard, with tricky cases, and try as you might, you just keep getting those `WRONG ANSWER` messages.

To make up for those bad feelings, in this task we decided to be extra nice to you. We will accept almost anything you submit for this problem!

And it gets even better. In this problem, if we somehow manage to disappoint you by a `WRONG ANSWER` message, we will try to make it up to you.

Scoring specification

For each of the data set D1 and D2 you **do not** solve, you score 0 points and gain 0 penalty minutes, as in all other tasks. Solving D1 is worth 1 point, solving D2 is worth 2 points, as in all other tasks.

The formula to calculate the penalty time is different. For this problem, the penalty time is calculated using the formula $T + (-40) \cdot R \cdot D$. Here T is the time in minutes when you submitted a correct solution, $D \in \{1, 2\}$ is the difficulty of the data set, and R is the count of previously rejected submissions for that data set.

In words, for every wrong answer you make we **decrease** your penalty time by 40 minutes in D1 and by 80 minutes in D2.

Note that the submission limit still applies. (You are allowed to make at most 10 submissions for each particular data set.)

Problem specification

1. You submit a text file.
2. We look at its first line. If it contains anything other than a single integer, we will accept your submission.
3. If it does contain a single integer and you already submitted this integer before, we will accept your submission.
4. If we still did not accept your submission, we run the program we provide in the input file to decide whether to accept or reject your submission.

(A small technical detail: The program in `d1.in` may seem really slow, but don't worry, our machine is really fast. Or maybe we are cheating and use a faster version of the same program, who knows.)

Input specification

The input file contains the source code of the program we will use to judge your submissions. The program is written in Python, which should make it readable to almost anyone.

Output specification

You may submit any text file the contest system allows you to submit.

Example

input

```
#!/usr/bin/python
import sys

def correct():
  print("OK")
  sys.exit(0)

def wrong_answer():
  print("Wrong answer")
  sys.exit(1)

# we already know that the first line contains a number, read it
N = int(sys.stdin.readline())

if N==47:
  wrong_answer()
else:
  correct()
```

output

```
42
is the answer to life, universe and everything.

This is just one of very many inputs that would get accepted.
This task is that easy!
```

Solution

Strategy

There were two valid strategies. The simplest one was just to submit something clearly wrong as soon as you read the problem.

The second strategy, obviously, involves solving the problem to find the inputs that cause `WRONG ANSWER`s, and submit these for a time bonus.

For the second strategy, you have to make an educated guess whether to do it or not. More precisely, you should estimate how long it will take you to solve the task. Note that if you spend T minutes solving this task, then each of the tasks you solve afterwards will be submitted T minutes later, thereby accumulating your time penalty. Thus you should only attempt this strategy if you are fast.

Easy data set

Function foo is better known as Euclid's algorithm calculating the greatest common divisor and function bar is a very clumsy way of testing primality of a given number: if p is a prime number then it doesn't have a common divisor greater than one with any number $q < p$ – $\gcd(p, q) = 1$ for every $1 \leq q < p$. On the other hand, if $n = d \cdot k$, than $\gcd(n, d) = d$ and $\gcd(n, k) = k$, so the converse holds too.

Thus we get a WRONG answer only if we find a prime number n in the given interval 100000000–198765432 such that numbers $n + 18$, $n + 36$, etc. are also primes. These numbers couldn't be found using the given program (in a reasonable time). It is way to slow even for testing a single number. A good way to solve this problem was to use the sieve of Eratosthenes and then sweep through the array and find the primes satisfying the given funny condition.

There are 14 such numbers: 186613711, 126605623, 143819771, 118810171, 123372071, 128293463, 196192091, 194990891, 172496083, 175066651, 132058483, 106961383, 117340703 and 142425763.

Hard data set

The hard problem was to find all the square roots of $1 \pmod N$. We call x a square root of a, if a is a square of x i.e., $x^2 = a$ (equivalently, a square root of a is a root of the quadratic equation $x^2 - a = 0$).

The square roots may be different depending on what numbers we work with, for example:

- if we work with positive real numbers, there is only one square root of 3 (but there is no square root of 3 e.g. in rational numbers)
- if we work with all real numbers, there are two square roots of 3 (both $\sqrt{3}$ and $-\sqrt{3}$ satisfy the definition above), but there is no square root of -1 (however, there are two square roots of -1 in complex numbers: i and $-i$)
- if we work with integers, there are two square roots of 4 (but no square root of 5 or -4)
- if we work with integers modulo 7, then $0^2 = 0$, $1^2 = 1$, $2^2 = 4$, $3^2 = 9 \equiv 2$, $4^2 = 16 \equiv 2$, $5^2 = 25 \equiv 4$ and $6^2 = 36 \equiv 1$; so 1, 4 and 2 have exactly two square roots (mod 7) and 3, 5 and 6 don't have a square root (mod 7);

In particular, note that there are two square roots of 1 – this is always true, if we work with integers modulo p, where p is any prime number (except 2). This is because $x^2 - 1 = (x - 1)(x + 1)$ and this is 0 (mod p) if and only if p divides $(x - 1)(x + 1)$, which is if and only if p divides $x - 1$ or p divides $x + 1$. If $0 < x < p$, then either $x = 1$ or $x = p - 1$. Note that $p - 1$ is the same as -1 (mod p) and it is easy to see that $(-1)^2 = (+1)^2 = 1$ (mod p). To sum up:

Fact 1: There are exactly two square roots of 1 (mod p) where p is a prime number, $p > 2$, namely 1 and $p - 1$.

- if we work with integers modulo n, where n is *not* prime, then 1 and $n - 1 \equiv -1 \pmod n$ are surely square roots of 1, however, there are more; for example take integers modulo 15, then $1^2 = 1$, $14^2 \equiv (-1)^2 = 1$, but also $4^2 = 16 \equiv 1$ and $11^2 = 121 \equiv 1 \pmod{15}$ – note that actually $11 = -4 \pmod{15}$, so we should have already known that $11^2 \equiv (-4)^2 = 4^2 \equiv 1$.
- finally, lets take pairs of integers (a, b) modulo 7; lets multiply these pairs this way: $(a, b) \cdot (c, d) = (ac \bmod 7, bd \bmod 7)$, so in particular, $(a, b)^2 = (a^2 \bmod 7, b^2 \bmod 7)$; what are the square roots of $(1, 1)$? well, we already know that there are two square roots of 1 modulo 7, namely ± 1 so there are exactly 4 square roots of $(1, 1)$, namely: $(+1, +1)$, $(+1, -1)$, $(-1, +1)$ and $(-1, -1)$

The last example may be obviously generalized to a simple fact:

> Fact 2: If we work with n-tuples $(a_1, a_2, \ldots, a_n)$, with multiplication defined as multiplying the corresponding elements independently) and there are exactly r_i square roots of 1 for the i-th element, then there are exactly $r_1 \cdot r_2 \cdots r_n$ square roots of $(1, 1, \ldots, 1)$.

Now how can we find the square roots mod 15? A "low level" approach: $15 = 3 \cdot 5$, so if 15 divides $x^2 - 1 = (x-1)(x+1)$, then either 15 divides $(x-1)$ (so $x = 1$), or 15 divides $x+1$ (so $x = 14$), or 3 divides $x-1$ and 5 divides $x+1$ or 5 divides $x-1$ and 3 divides $x+1$. Take the last case: $x - 1 = 5k$ and $x + 1 = 3\ell$ (for some k, ℓ), so $x = 5k + 1 = 3\ell - 1$ and $3\ell - 5k = 2$. Numbers k and ℓ can be found using extended Euclidean algorithm (see `http://en.wikipedia.org/wiki/Extended_Euclidean_algorithm`, `http://en.wikipedia.org/wiki/Bézout's_identity`).

A more "high level" view: There is a one-to-one correspondence between numbers mod 15 and pairs (a, b), where a is mod 3 and b is mod 5; specifically, number n corresponds to pair $(n \bmod 3, n \bmod 5)$. Furthermore, multiplying two numbers mod 15 corresponds to multiplying the corresponding pairs; in particular, square roots of 1 mod 15 correspond to square roots of $(1, 1)$ mod 3,5. We already know the square roots of $(1, 1)$ are $(+1, +1)$, $(+1, -1)$, $(-1, +1)$ and $(-1, -1)$. Thus all square roots of 1 mod 15 are numbers x such that

$$x \equiv \pm 1 \pmod 3 \qquad x \equiv \pm 1 \pmod 5.$$

The solutions can be found by Chinese remainder theorem.
See `http://en.wikipedia.org/wiki/Chinese_remainder_theorem`.

> Chinese remainder theorem: Given pairwise coprime integers $n_1, \ldots, n_k$ and any integers $a_1, \ldots, a_k$, the system of simultaneous congruences
>
> $$x \equiv a_1 \pmod{n_1} \qquad \cdots \qquad x \equiv a_k \pmod{n_k}$$
>
> has one unique solution mod $N = n_1 \cdot n_2 \cdots n_k$.

To solve the hard problem you should factorize number N first.

$$\begin{aligned} N &= 655076673443707551099143071618051666714277042558983269995765119 \\ &= 1157710228741282062487 \cdot 943753934567180439929 \cdot 599561157376811721953 \end{aligned}$$

This is quite a big number with big prime factors, so the naïve factorization algorithms would take centuries to fulfill this task. However using e.g. `http://www.alpertron.com.ar/ECM.HTM`, the number can be factorized in few seconds.

Since $N = pqr$ is a product of 3 prime numbers, there are exactly 8 square roots mod N corresponding to tripples $(\pm 1 \bmod p, \pm 1 \bmod q, \pm 1 \bmod r)$. To find them we used the following program:

```
p = 1157710228741282062487; q = 943753934567180439929; r = 599561157376811721953
m = [p, q, r]

def extgcd(a, b):
  if b == 0: return (a, 1, 0)
  d, u, v = extgcd (b, a%b)
  return (d, v, u-(a/b)*v)


def crt(y, m):
  x = y[0]; M = M2 = m[0]
  for i in xrange(1,len(m)):
    d,u,v = extgcd (M, m[i])
    M2 *= m[i]
    x += (M*u % M2)*(y[i]-x) % M2
    M = M2
  return x

print crt ([p-1,q-1,r-1], m)
print crt ([  1,q-1,r-1], m)
print crt ([p-1,  1,r-1], m)
print crt ([p-1,q-1,  1], m)
print crt ([p-1,  1,  1], m)
print crt ([  1,q-1,  1], m)
print crt ([  1,  1,r-1], m)
```

Excluding the trivial square root 1 (which gave correct answer), these 7 numbers gave the `WRONG ANSWER`:
655076673443707551099143071618051666714277042558983269995765118,
423708042681157396922466745853617748770106296874520998638487956,
409195345538471765371894681196161478839399343815134048305305493,
477249958667785939903924716186324105819048444428311493047736788,
231368630762550154176676325764433917944170745684462271357277163,
245881327905235785727248390421890187874877698743849221690459626,
177826714775921611195218355431727560895228598130671776948028331.
(The first number, $N-1 \equiv -1 \bmod N$, could be found without much effort.)

Easy representation

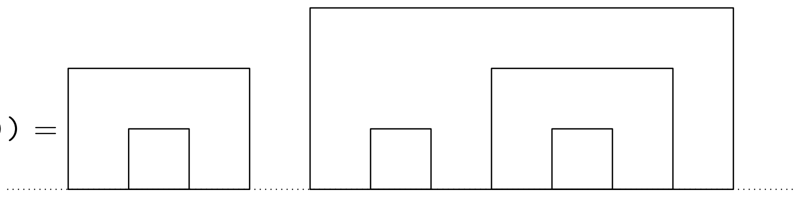
Authors

Problemsetter:	Lukáš Poláček
Task preparation:	Lukáš Poláček, Michal Nánási

Problem statement

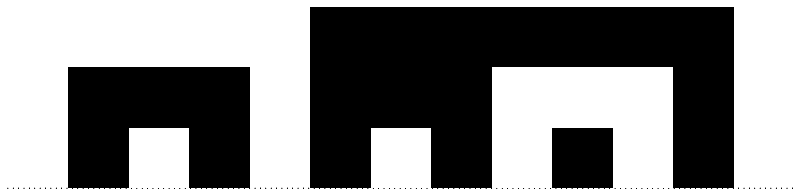
Peter is preparing slides for his lecture on parsing arithmetic expressions. In the first part of the lecture he wants to focus just on parsing brackets. He invented an interesting geometric representation of a correct bracket sequence for his students, because one image is better than a thousand words:

`(())(()(()))` =



Formally, the definition of the geometric representation looks as follows. The simplest correct bracket sequence `()` is represented by a 1×1 square. If `A` is a correct bracket sequence and $g($`A`$)$ its represenation, then the representation for `(A)` is $g($`A`$)$ surrounded by a rectangle two units wider than $g($`A`$)$ and one unit taller than the highest point of $g($`A`$)$. If `A` and `B` are two correct bracket sequences and $g($`A`$)$ and $g($`B`$)$ are their representations, then we get $g($`AB`$)$ by placing $g($`B`$)$ one unit to the right of $g($`A`$)$.

After he finished his slides, Peter started to play with the images he prepared. He painted the bounded areas of the images alternately black and white, in such a way that the outer-most areas are all painted black. For the example above this coloring looks as follows:



Problem specification

You are given a correct bracket sequence. Calculate the area that is colored black.

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 line with a correct bracket sequence. Every line will only contain characters '`(`' and '`)`'.

Output specification

For each test case output one line with one integer – the area of the black part of the corresponding geometric representation.

Example

input

```
2

((()))

(())(()(()))
```

output

```
10
20
```

The second test case is the one shown in the pictures above.

Solution

We will use a stack to calculate the result in linear time. The stack is initially empty. Process the bracket sequence from left to right. Each time we encounter an opening bracket `'('`, we push a value on top of the stack. When we encounter a closing bracket `'('`, we pop a value from the stack. We will discuss later what kind of values will be stored in the stack.

Let us assume that we have already processed some part of the sequence. How does the stack look like? Each value in the stack corresponds to an opening bracket that is not yet closed, i.e. the corresponding closing bracket was not yet processed. Since we process the sequence from left to right, the values in the stack are ordered according to the position of their corresponding opening bracket and the lowermost value corresponds to the leftmost opening bracket.

We will call a matching pair of brackets P *parent* of a matching pair Q, if P was right below Q in the stack. Smilarly, we will call Q *son* of P, if P is parent of Q. Denote by $r(P)$ the rectangle that corresponds to the matching pair P in the geometric representation and denote by $|r(P)|$ its area.

Suppose that the area between $r(P)$ and all rectangles inside $r(P)$ was coloured black. What is the size of this area? All rectangles inside $r(P)$ correspond to the sons of P, because they had to be just above P in the stack. Therefore to calculate the size of the area that was coloured black we need to substract from $|r(P)|$ the areas of all rectangles corresponding to the sons of P.

How to calculate the value $|r(P)|$? We need to know the height and the width of the rectangle $r(P)$. The width is just the distance of the opening and closing bracket of the matching pair P. The height is the maximum of the heights of the rectangles corresponding to the sons of P increased by one. To calculate this, the first two values stored in the stack for pair P will be l and h, were l is the position of the opening bracket and h is the maximum of the heights of the rectangles corresponding to already processed sons of P. We will update the value h each time we pop a son of P. When we encounter the closing bracket of pair P, we can directly use the value h as the height and $k - l$ as the width of $r(P)$, where k is the position of the closing bracket. The third value stored in the stack for P will be the sum of the areas of rectangles corresponding to already processed sons of P. When we push P into the stack, we set this value to zero and we update it each time we pop a son of P from the stack.

Each bracket is processed in constant time, therefore the time complexity is linear. Also the memory complexity is linear, since we use just one stack.

Fish Fillets

Authors

Problemsetter: Peter Perešíni, Michal Forišek
Task preparation: Peter Perešíni, Michal Forišek

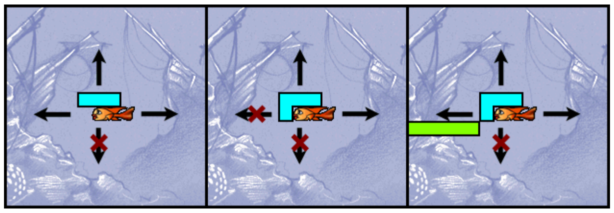
Problem statement

Fish Fillets is little known but extremely cool and challenging logic game made by the Czech company Altar, and later released as open source. In this task you will have to solve two new levels – by any means you can use.

You are controlling two fish trapped in a room. The big blue fish is Agent Max Flounder, and the small orange one is Agent Tina Guppy. The entire game is played on a discrete grid. In each step, you may move one of the fish by one square in one of the four basic directions. The fish can push and carry other objects according to specific rules described later. The player's goal is to get both fish out of the room. Sounds simple – but make one bad move and falling objects will kill your fish.

To get yourselves acquainted with the game, you can watch a video with instructions at `http://www.youtube.com/watch?v=qs1JedhK0fI` or download this video from our server. Alternately, the same rules as in the video are summarized below.

Pushing objects



The fish are only in danger when they move some object. The simplest situation is lifting objects – there is no danger in it. If the fish on the left picture moves up, it lifts the object. If it moves left or right, the object stays in place. If the fish goes on moving left or right long enough, it will eventually come out from under the object, and the object will fall harmlessly down. The only dangerous direction is down. If the fish moves down, the object it was supporting will fall on it and kill it.

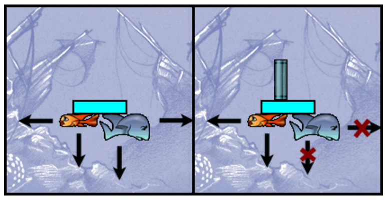
If the fish want to push an object left or right, the object must be supported by some structure or other object. The fish cannot support the object it is pushing. The fish on the middle picture cannot move left because it would get killed.

The fish can push an unsupported object only if it becomes supported in the new position. An example of such a situation is on the right picture.

Steel objects

In some levels you will encounter steel objects. They look like the cylinder on the picture above. Steel objects can only be lifted and pushed by the big fish. The small fish cannot move them. Moreover, if she gets to a situation where she has to support a group of objects that includes some steel objects, she will perish.

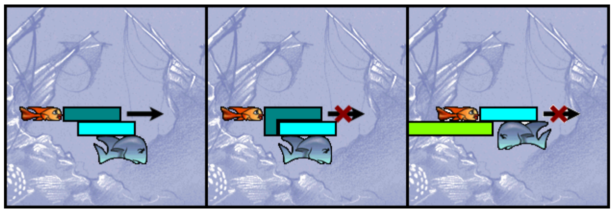
Transferring objects



The fish can transfer objects between themselves. If one fish supports the object and the other one gets into position where the object rests also upon her, the first fish can move away. Any fish can move away in the left picture and the object will stay upon the other.

Be careful with transfer of steel objects. In the situation on the right picture only the smaller fish can move away.

Pushing objects along other objects



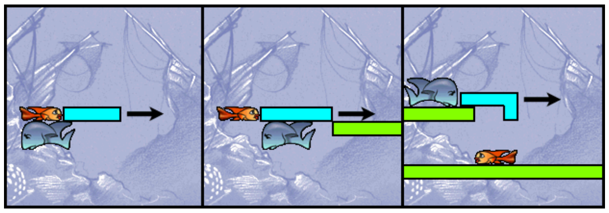
It is allowed to push objects that are not directly supported by the fish. One such case is presented on the left picture.

However, a fish is not allowed to push an object that is directly supported by her partner; in such case the partner would perish. More precisely, this is disallowed in the case where the object would rest directly on the partner after the push. For example, the small fish on the middle picture cannot push right because it would kill the big fish.

For the same reason it is impossible to place an object upon a fish. This is shown on the right picture.

Note that on the middle picture the small fish could swim around and push the L shaped object left.

Falling objects



Falling objects are always deadly. It doesn't matter how long the object falls, if it hits a fish or something that rests upon a fish, the fish is dead.

The object supported directly by a fish can only be pushed if it falls down or rests upon some structure immediately afterwards. This is shown on the left and middle pictures.

Be on the lookout for the situation where the object seemingly hits the fish but at the same time it hits some structure. In such a situation the fish is safe. An example of such a situation is on the right picture.

Problem specification

Your task is to solve two levels of this game. A level is solved if both fish leave the room alive. A fish will leave the room the moment it touches the boundary of the level After a fish leaves, the player can not return it back into the room.

We provide you both with a textual description of the levels and with a Flash applications which you can use to familiarize yourselves with the rules and even to solve the entire levels, if you prefer to do it by hand. If you solve a level, the application will congratulate you and show you the sequence of moves you made. (You can then copy this sequence and submit it.)

Instructions for the Flash application

- R: restarts the level
- S or F2: saves the current position
- L or F3: loads the most recently saved position
- space: switches which fish is active
- arrow keys: moves the active fish one step in the given direction
- some objects are animated – this is eye candy only, ignore it when solving

Input specification

All coordinates in the input are 0-based, with (0,0) in the upper left corner. For each object the initial coordinates of its upper left corner are given. The rest should be obvious.

Output specification

Send us a text file with a sequence of moves that wins the given level. Any such sequence of 5 000 or less moves will be accepted. Each move is a character from the set `UDLRudlr`. Lowercase letters are movements of the small fish, uppercase ones are movements of the big fish. The letters represent the directions: Up, Down, Left and Right. Whitespace does not matter.

Example

input

```
room:
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 0 0 0 0 0 0 0 1 0 0 0 0 0 1
1 0 0 0 0 0 0 0 0 0 0 0 0 0 1
1 0 0 0 0 0 0 0 0 0 0 0 0 0 0
1 0 0 0 0 0 0 0 0 0 0 0 0 0 0
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1

small fish:
row 4 col 1
1 1 1

large fish:
row 1 col 10
1 1 1 1
1 1 1 1

item 1:
row 2 col 9 steel
1 0
1 1
1 1
```

output

```
L R D L L L L
R R R R D R
u u u r r r r d r r r r r r d r
```

The first row of the output is the big fish pushing the steel object to the left. In the second row it exits the room. Finally, the small fish can now swim around the steel object and exit the room as well.

The entire initial position in ASCII art looks as follows:

```
111111111111111
1.......1.BBBB1
1........oBBBB1
1........oo....
1SSS.....oo....
111111111111111
```

Here, `B` *is the big fish,* `S` *the small one, and* `o` *is the steel object.*

Solution

Implementing even the simplest possible state space search for this game is quite a lot of work, and it is not even worth the effort – the number of possible states is so huge that there is no chance it will find a solution.

Instead, the recommended way to solve this task was to use the applet to play, and to use your head to make the moves.

We recorded screencasts of us solving both levels, you can find them here:

- Easy: `http://www.youtube.com/watch?v=8MXogN0GHJw`
- Hard: `http://www.youtube.com/watch?v=DO39o8FBpgA`

The only solution of the hard level submitted during the contest: `http://www.youtube.com/watch?v=1Dgte4lUCEE`

Going to the movies

Authors

Problemsetter: Michal Forišek
Task preparation: Michal Forišek, Lukáš Poláček

Problem statement

N friends decided to go to the local cinema together. They all bought tickets to the same row. As there was still some time left, each of them took her ticket and went shopping until the movie starts.

They all arrived back late, the movie already started. The usher standing at the door agreed to let them in one by one. Each of the girls was supposed to find her place and sit down.

However, the machine that printed their tickets was broken. Instead of consecutive numbers, each girl received a random seat number between 1 and K, where K is the number of seats in their row. The seat numbers they received were not necessarily distinct.

When a girl tries to sit down, she enters the row at the end where seat number 1 is, and walks until she reaches the number on her ticket. If her desired seat is free, she just sits down. If it is already taken, she continues to walk in the same direction until she finds the first free seat, and sits there.

Of course, it is possible that some unfortunate girl will reach the end of the row without finding a place to sit. In that case, the usher comes and throws her out.

Problem specification

You are given the numbers N and K.

Assume that each girl's ticket had a number between 1 and K, inclusive. Each number was drawn uniformly at random, and draws were independent.

Also assume that the entire row was empty when the first girl started to look for her seat.

Compute the probability that at least one girl suffered the sad fate of being thrown out by the usher.

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 containing two integers N and K.

Output specification

For each test case output a single line with the probability as a simplest fraction.

(Do not output any spaces before or after the / sign.)

Example

input

```
3

1 10

2 3

3 3
```

output

```
0/1
1/9
11/27
```

In the third case there are $3^3 = 27$ possibilities. Out of these, in 11 some girl is thrown out. These 11 sequences are: 133, 222, 223, 232, 233, 313, 322, 323, 331, 332, and 333.

For example, if the sequence of numbers were 322, the first girl sits at seat #3, the second one at #2, and then the third one tries to sit at #2, but finds both seat #2 and seat #3 occupied, and she's thrown out.

Solution

First of all, if the number of girls N is greater than the number of places K, the probability that some girl has no place to sit is clearly 1. In the rest of this solution we assume that $N \leq K$.

We can turn the task into a combinatorial one. Instead of computing the probability, we can compute the number of bad sequences.

We can also use complementary counting. Instead of counting the bad sequences, we will find the number of good sequences – i.e., sequences such that all girls find a place to sit.

We will now present a simple argument that can be used to count the good sequences.

Imagine that the girls decided to practice this activity at home. However, instead of using K chairs in a row, they decided to use $K + 1$ chairs and placed them in a circle so that the numbers increased clockwise.

The rest of the game remains essentially the same as before – one after another, each girl enters the circle at the chair with the number she has on her ticket, walks clockwise until she finds the first free chair, and sits down.

As an example, consider the case $K = N = 4$. We have $K+1 = 5$ chairs numbered 1 to 5. Each of the $N = 4$ girls generates a random number between 1 and 5, and then enters the circle and finds her place.

Suppose that the generated numbers are 5, 3, 3, 4, in this order. The first girl sits at the chair #5, the second one at #3. Then the third girl arrives. Her desired chair #3 is already taken, hence she sits at #4. Finally, the last girl arrives. Chairs #4 and #5 are taken, hence she sits at chair #1. In the end, chair #2 remained empty.

Remember that we assume $N \leq K$. Hence whenever this modified game is played, each girl will find a place to sit, and some of the chairs (exactly $K + 1 - N$ of them) will remain empty.

Each of the N girls has exactly $K+1$ chair numbers to choose from, hence there are exactly $(K+1)^N$ ways in which the game can be played.

Imagine that the girls played each of the $(K + 1)^N$ possible games once, and after each game they placed a coin on each of the $K + 1 - N$ chairs that remained empty.

Now comes the most important observation: The game on the circle is perfectly symmetric. Hence after the girls played all $(K+1)^N$ games, we must have the same number of coins on each of the chairs.

And as the girls placed $C = (K+1)^N \cdot (K+1-N)$ coins, the number of coins on each of the $K+1$ chairs must be $C/(K+1) = (K+1)^{N-1} \cdot (K+1-N)$.

We just proved the following fact: In the game on the circle, there are exactly $(K+1)^{N-1} \cdot (K+1-N)$ sequences of chair numbers such that the chair number $K+1$ will remain free at the end. Let $\mathcal{S}$ be the set containing these sequences.

Observe that no sequence in $\mathcal{S}$ may contain the number $K+1$ – because if it did, the first girl with the number $K+1$ would sit there, hence chair $K+1$ would not be free at the end. Thus each sequence in $\mathcal{S}$ only contains numbers 1 to K.

And we are almost done. It's now easy to see that the set $\mathcal{S}$ are precisely all the good sequences in the original game.

(If a sequence is good in the original game, this means that no girl "overflows" past the seat number K – hence if we play it on the circle with $K+1$ chairs, chair $K+1$ will remain empty. And vice versa.)

Hence in our original game, out of all $A = K^N$ sequences we have $G = (K+1)^{N-1} \cdot (K+1-N)$ good ones. The answer is then $1 - G/A$.

(Note that the numbers K and $K+1$ are always relatively prime, hence so are K^N and $(K+1)^{N-1}$. Thus G/A is always almost in reduced form, and the correct answer is easy to compute even if your programming language does not support big integers.)

Hunt!

Authors

Problemsetter:	Michal Nánási, Vladimír Koutný
Task preparation:	Vladimír Koutný, Milan Plžík, Monika Steinová

Problem statement

This year new sheep-hunting rules were set in Wolfenstein:

- wolves are allowed to form packs of up to 10 animals
- hunting is only allowed on a torus divided into 30×30 cells
- there are 60 sheep randomly placed on the torus
- in each round:
  - wolves move before sheep do
  - a move means that an object moves from its cell into one of the four neighboring ones
  - all wolves together can do up to 10 moves
  - no single wolf can do more than 5 moves
  - each sheep can do up to 1 move
  - sheep moves are deterministic – they basically try to move away from the closest wolf
- if at any time a wolf moves to a field occupied by a sheep, the sheep is eaten
- if at any time a wolf moves to a field occupied by another wolf, one of the wolves is eaten

A torus is a surface shaped like a donut. You can imagine the hunting area as a normal 30×30 chessboard, except for the fact that in each column the top and bottom cells are neighbors as well, and the same holds for the leftmost and rightmost cell in each row.

Problem specification

You are controlling the wolves. To gain one point for solving the easy data set, your task is to eat 10 sheep. To gain two more points for solving the hard data set, you have to eat all 60 sheep.

Note that you only play a single game for both data sets, and this game **can not be restarted**.

You are allowed to send us the moves of your wolves multiple times. However, there must be **at least five minutes** between any two consecutive submissions you make. Each time you send us your moves, we evaluate them and then move the sheep in your game.

Scoring

As this task is a bit special, so is the scoring:

- You will get a time penalty (a '*' in the results) for each submission that does not obey the "five minutes between submissions" rule.
- Rejected submissions count towards the data set you are working on. I.e., until you eat 10 sheep all penalties are assigned to the easy data set, afterwards they are assigned to the hard data set.
- The limit is 10 incorrect submissions per data set, as usual.

Input specification

There is no input for this task. You will receive your game description after your first submit.

Output specification

Every round we expect a list of 0 to 10 moves of your wolves (all tokens are separated by any whitespace). Each move is described by a pair `id dir` where `id` is the number of wolf (1 to 10, in the order they appear in the latest state description), `dir` is one of '`U`', '`D`', '`L`', and '`R`' (for moving the wolf up, down, left or right along the grid). Invalid moves in your submission will be ignored.

After any non-rejected submit you will receive the description of the current state.

Which data set to use?

Don't worry about this. Regardless of how many sheep you already ate, you may always submit your moves as your "solution" to either of the data sets H1 and H2. The tester will handle it for you.

State description

The state description you receive after each submit is formatted as follows: It consists of 2 blocks – wolves block and sheep block. Each block starts with a number `N` giving the count of wolves/sheep. This number is followed by N pairs x y ($0 \leq x, y < 30$) describing positions of each wolf/sheep.

You may assume that in each state all living sheep have distinct positions.

The directions up, down, left and right are such that from a cell $(4, 7)$ a move up takes you to $(4, 6)$, and a move left takes you to $(3, 7)$.

Example

game description

```
3
4 0   5 5   29 7
5
4 8   4 28   15 3   18 12   6 27
```

The above game description contains 3 wolves and 5 sheep.

moves

```
1 U
1 U
1 L
3 R
3 D
```

This is a sequence of moves in which first wolf 1 makes three moves, and then wolf 3 makes two moves.

If we apply this sequence of moves to the game state described above, we'll get the following new state:

game description

```
3
3 28   5 5   0 8
4
4 8   15 3   18 12   6 27
```

Note that after two steps wolf 1 ate the sheep at $(4, 28)$.

Now the sheep would move. For example, the sheep at $(6, 27)$ is closest to the wolf at $(3, 28)$, hence it would move away from him.

Solution

The hunt as such was not that difficult. A human player can easily win the game in less than 30 moves. We will now describe one possible strategy. The basic idea of our approach is to move only a few of the wolves in every round.

First we start by moving a majority of wolves to partially "surround" the sheep. If these moves are performed well, the sheep will start to run away in such a way that they create small groups. After such small groups are created (which is a matter of 3-4 rounds) we choose a few (usually 2-3) wolves that are close to some groups. In the following rounds each chosen wolf is used to catch all sheep from one particular group. Using this approach, we make these few wolves much faster than the sheep, so they are able to catch all the sheep before they run away too far. The remaining wolves are used only as scarecrows (well, scaresheeps, to be precise :-)) When a group is caught, we pick new wolf and attack the nearest group. Once all large groups are eaten, we can just greedily pick off the remaining sheep.

Using this strategy, one can usually catch the first 10 sheep in the first 7 rounds, and finish the game after 25 to 30 rounds.

Another nice strategy was to start by making empty moves during the first hour. This requires no thinking and it has the nice effect that the sheep will form nicely edible clusters.

However, the hardest part of this task was not the strategy. The hardest part was time management. Finding the time to make each move distracts the player from solving the other tasks. It may be a better strategy to implement a simple player (e.g., "always move two wolves greedily") and let it play automatically. If you run the program early enough, you can easily get over 50 moves, and this should be enough even for a simple strategy to succeed.

Instructions

Authors

Problemsetter:	Michal Forišek
Task preparation:	Monika Steinová, Tomáš Záthurecký

Problem statement

The wannabe scientist Kleofáš has recently developed a new processor. The processor has got 26 registers, labeled `A` to `Z`. Each register is a 64-bit unsigned integer variable.

This new processor is incredibly simple. It only supports the instructions specified in the table below. (In all instructions, `R` is a name of a register, and `X` is either the name of a register, or a 64-bit unsigned integer constant.)

syntax	semantics
mov R X	Store the value X into R.
and R X	Compute the bitwise and of the values R and X, and store it in R.
or R X	Compute the bitwise or of the values R and X, and store it in R.
xor R X	Compute the bitwise xor of the values R and X, and store it in R.
not R	Compute the bitwise not of the value R, and store it in R.
shl R X	Take the value in R, shift it to the left by X bits, and store the result in R.
shr R X	Take the value in R, shift it to the right by X bits, and store the result in R.

Notes:

- After any instruction other than `not`, if the second argument was a register name, its content remains unchanged.
- If `shl` or `shr` is called with a non-zero second argument, the shifted value of the first argument is padded with zeroes. Note that this means that if the second argument is 64 or more, the result will always be zero, regardless of the first argument.

Example task:

Assume that the register `A` contains an arbitrary input number between 0 and 15, and that all the other registers contain zeroes. Write a program that will set `Z` to 1 if the number of set bits in `A` is odd, and to 0 otherwise.

Solution for the example task:

The answer can easily be computed as the bitwise `xor` of the four lowest bits in `A`.

We can isolate a single bit X by first shifting the number $63 - X$ bits to the left (the X-th bit will become the most significant one, and all more significant bits of `A` become lost), and then 63 bits to the right (the originally X-th bit is now the least significant one).

In this way we can take the four bits of `A` that may be non-zero, and store them into `B` to `E`, respectively. We then compute the bitwise `xor` of these four bits and store it in `Z`.

The entire program solving the example task:

```
mov B A
shl B 63
shr B 63

mov C A
shl C 62
shr C 63

mov D A
shl D 61
shr D 63

mov E A
shl E 60
shr E 63

mov Z B
xor Z C
xor Z D
xor Z E
```

Easy task:

Assume that the register A contains an arbitrary input number, and that all the other registers contain zeroes. Write a program that will increase the value in A by 8 (computing modulo 2^{64}, if necessary). After your program terminates, the other registers are allowed to contain anything. Your program must have less than 64 instructions.

Hard task:

Assume that the register A contains an arbitrary input number, and that all the other registers contain zeroes. Write a program that will change the value in register A into the next larger value with the same number of set bits. After your program terminates, the other registers are allowed to contain anything. Your program must have less than 300 instructions.

In other words, if we regard A as a sequence of 0s and 1s, your task is to produce the next permutation of the given sequence.

For example, if the input is 23, which is 10111 in binary, the output should be 27, which is 11011 in binary. If the input is 24, which is 011000 in binary, the output should be 33, which is 100001.

You may assume that the output for the given number in A is always less than 2^{64}. I.e., the input will be such that the next larger value with the same number of set bits still fits into the 64-bit register.

Input specification

This problem has no input.

Output specification

Send us your program as a text file. Each line of the file may either contain one complete instruction, or just whitespace.

Solution

Easy data set

Consider first the simpler problem: To increment a number in register A by one. In general such a number has binary representation $x01^n$, where x is an arbitrary string of ones and zeroes and notation d^k means k consecutive digits d (so 1^n is n ones). Clearly by incrementing this number by one, we get $x10^n$.

In order to perform this operation on our machine we first separate the rightmost (least significant) block of n ones. We do this in the following way:

First we negate the number to get $y10^n$ (where y is a bitwise negation of x).

```
mov B A
not B
```

Then we replace y by ones. We can do this by propagating that rightmost one to the left by `shl`-ing and `or`-ing similar to this:

```
mov C B
shl C 1
or B C
```

After this block we have something like $y'110^n$ in register B (where y' is some string of ones and zeroes). Clearly by repeating this block (at least) 64 times we can get $1^{64-n}0^n$ in register B. But that will make out program too long. Fortunately we now have (at least) two rightmost ones so we can go faster:

```
mov C B
shl C 2
or B C
```

and we get $y''11110^n$. Now we have four ones so we can go even faster. Another four blocks will complete the work:

```
mov C B
shl C 4
or B C

mov C B
shl C 8
or B C

mov C B
shl C 16
or B C

mov C B
shl C 32
or B C
```

Now we have $1^{64-n}0^n$ in B. That is what we wanted the rightmost ones (in original number) are separated. Some simple operations will finish the incrementation:

Clear out the rightmost ones in A to get $x00^n$,

```
and A B
```

construct the one on $n+1$-th place ($0^{63-n}10^n$),

```
mov C B
shl C 1
not C
and B C
```

and put that one into A.

```
or A B
```

And we are done. Almost. Our task was to increment value in A by 8, not 1. However since binary representation of 8 is 1000 the solution is similar. We just shift the number to the right by 3 positions, add 1 and shift it back (while preserving last 3 bits). Sure you can imagine how to do that.

Also note that the above construction of incrementing by one need not work if the result would overflow. However in order to use it to solve our origonal task this is not important since we shift the inpit number to the right (so result of incrementation will not overflow) and then we shift it left (so excess bits will get lost).

Hard data set

First let's find out how does next permutation of bit sequence look like. Clearly a next permutation of a sequence $x011^m0^n$ is $x100^n1^m$. That is, to get a next permutation we find the rightmost one preceded by a zero, move it one place to the left and push the block of ones right of it far to the right. If there is no one preceded by a zero, then the input is either 0 or its next permutation does not fit in 64-bit register.

In order to find the next permutation we separate the blocks of n zeroes and m ones by using the similar method as in solution of easy task:

```
mov B A

mov C B
shl C 1
or B C

mov C B
shl C 2
or B C

mov C B
shl C 4
or B C

mov C B
shl C 8
```

```
or B C

mov C B
shl C 16
or B C

mov C B
shl C 32
or B C
```

Now we have $1^k111^m0^n$ in B (where $k = 62 - n - m$) and to separate m ones we do

```
mov C A
not C
and C B
```

to get $y100^m0^n$ in C (where y is a bitwise negation of x) and

```
mov D C
shl D 1
or C D

mov D C
shl D 2
or C D

mov D C
shl D 4
or C D

mov D C
shl D 8
or C D

mov D C
shl D 16
or C D

mov D C
shl D 32
or C D
```

to get $1^k100^m0^n$ in C. Now we can clear out the rightmost m ones in A

```
and A C
```

and find the one that was originaly the rightmost one preceded by zero and move it one place to the left (it is the rightmost one of C).

```
mov D C
shl D 1
not D
and D C
or A D
```

Now we have $x100^{m+n}$ in A and 0^k100^{n+m} in D. Using the current values on B,C and D we can reconstruct the block of m ones:

```
shr D 1
or D C
not D
and D B
```

Now we have $0^k001^m0^n$ in D. All we need is to push them n places to the right. We can do this with the help of value in B. First we negate it to get $0^{64-n}1^n$

```
not B
```

and then we will simultaneously shift B and D while there ar som ones in B. This can be done in the following way:

First we check if there is a one in B:

```
mov E B
and E 1
```

Now E is 0 if B has no ones and E is 1 if there are some ones in B. This works because B has form $0^{64-n}1^n$. So we can shift B and D by E.

```
shr B E
shr D E
```

So by these two blocks we can shift D by one place to the right and clear one one from B if there is one. By repeating them (at least) 64 times we can shift D to the right by n places because B originally contained n ones. However this would (probably) make the program too long. Of course we can go faster, for example:

```
mov E B
shr E 7
and E 1
shl E 3
shr B E
shr D E
```

This will shift D by 8 places to the right and clear 8 ones from B if there are at least 8 of them. By using 7 blocks similar to above one we can shift D to the right by n places in a more efficient way:

```
mov E B
shr E 63
and E 1
shl E 6
```

```
shr B E
shr D E

mov E B
shr E 31
and E 1
shl E 5
shr B E
shr D E

mov E B
shr E 15
and E 1
shl E 4
shr B E
shr D E

mov E B
shr E 7
and E 1
shl E 3
shr B E
shr D E

mov E B
shr E 3
and E 1
shl E 2
shr B E
shr D E

mov E B
shr E 1
and E 1
shl E 1
shr B E
shr D E

mov E B
shr E 0
and E 1
shl E 0
shr B E
shr D E
```

And all that remains is ti put those shifted m ones into A

```
or A D
```

and we are done.

Jumbo Airlines

Authors

Problemsetter: Michal Forišek, Monika Steinová
Task preparation: Peter Košinár, Monika Steinová

Problem statement

The new catchphrase of Jumbo Airlines is "No annoying neighbors, each flight a unique experience!"

And as in most cases, the advertisement was produced by the marketing department, without ever consulting the engineers. They only learned about it after the boss asked them to "handle it ASAP".

There are M seats in each row, and there are N rows of seats in the airplane. Hence the seats form an $M \times N$ grid. (For the purpose of this problem we will ignore the presence of aisles.) The airline sells exactly K tickets for each flight.

To make sure that the "no annoying neighbors" part of the motto is satisfied, the seating must obey the following rule: Whenever a seat is occupied, the seats immediately in front of it and behind it, as well as the seats immediately to the left and to the right must remain free.

An *allowed arrangement* is a set of K occupied seats that obeys the rule above.

The "unique experience" part of the motto is then satisfied by using a different arrangement of occupied seats for each flight. (Two seating arrangements are different if there is at least one seat which is occupied in one arrangement and free in the other.)

Problem specification

You are given the numbers M, N and K. Find the number of different allowed seating arrangements.

As this number can be very large, we're only interested in its value modulo 420 047.

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 containing three integers M, N and K.

Output specification

For each test case output a single line with the number of allowed arrangements modulo 420 047.

Example

input

```
3

2 3 2

2 4 4

2 5 1
```

output

```
8
2
10
```

Solution

The most straightforward approach was to perform an exhaustive search on all the possible seating arrangements. Although the time complexity of $O((MN)^K)$ is quite high, the memory requirements are quite favourable ($O(MN)$) and this approach is sufficient for the easy input set.

The hard case was a bit more tricky. All the test cases can be divided into three groups, each of them being solvable in a different way.

The first group consists of cases with M quite small (between 10 and 15) and N of modest size (up to 50). For this group, we can use the dynamic programming approach. It works by considering all the possible patterns of P occupied/free seats in each row and keeping track of the number $A_{K,N,P}$ of possible seating arrangements consisting of N rows, having K occupied seats and ending in pattern P (where P is a pattern of M bits). Then, calculating the value of $A_{K,N+1,P}$ is a matter of summing the corresponding $A_{K',N,P'}$ values for patterns P' which could have been in the preceding row, which are compatible with pattern P (i.e. patterns which wouldn't violate the no-adjacent-occupied-seats condition). Obviously, the number of seats we can use in the preceding N rows, K', is equal to K minus the number of seats used in the latest, $(N+1)$-th row. Implemented directly, this would yield time complexity of $O(2^{2M}NK)$, as the number of possible patterns is 2^M and we're matching each of them against each.

This can be improved by realizing that if a pattern P' is compatible with pattern P, then all sub-patterns of P' are too (i.e. patterns created from P' by leaving out one or more occupied seats). In fact, patterns compatible with P are precisely all the sub-patterns of the complement of P (which might not be an admissible pattern in itself, though). Thus, if we could calculate for each pattern the total number of seating arrangements it and its sub-patterns produce, and do so in an efficient way, we'd be done!

It turns out that this is possible – by considering the seats in the pattern one by one and adding the numbers of arrangements provided by the pattern without the seat occupied to the number of arrangements with that seat occupied. The total time complexity of this approach is then $O(2^M MNK)$, while it uses $O(2^M K)$ memory.

The second group consists of cases with very small M and very large N. Unfortunately, the N factor in the approach mentioned above prevents us from using it directly. However, if one looks at the dynamic programming and recalls that all the calculations performed were linear, it's immediately obvious that the whole process can be seen as multiplication of a vector consisting of $2^M.(K+1)$ coordinates (the counts in the linear programming) by a huge square matrix of the same dimensions. Thus, in order to perform N steps, we only need to raise this matrix to the N-th power and that can be done in time proportional to $\log N$ rather than N. The price paid is both in the memory requirements which raise to $O(2^{2M} \cdot K^2$ and the cubic term in the resulting time complexity $(2^M \cdot K)^3. \log N$, originating from the matrix multiplication.

In fact, the base of the exponential could be lowered by considering only the admissible patterns, rather than all of them – this would end up with the base being equal to the golden ratio $\phi = \frac{1}{2}(\sqrt{5}+1)$, as the number of admissible patterns of M seats is equal to the $(M+2)$-th Fibonacci number and those grow exponentially, with base ϕ.

Finally, the last group contains cases in which both M and N are quite large, which might look scary... However, the value of K is always equal to either 1 or 2 – thus, allowing us to determine the answer explicitly (in fact, it's possible to give the answer in explicit way for each fixed value of K). For $K = 1$, the answer is obviously MN (modulo the magic prime 420,047, of course). For $K = 2$, it can be easily seen to be equal to $\frac{1}{2}((MN)^2 - 5(MN) + 2(M+N))$ by considering the placement of the first seat (corners, sides or inside the rectangle) and the number of remaining possibilities for the second seat.

Karl's shopping

Authors

Problemsetter: Tomáš Záthurecký
Task preparation: Tomáš Záthurecký, Peter Košinár

Problem statement

Karl is going to spend his holiday in Nothingland. Since there is nothing there, he has to buy all supplies now. At the moment, he is waiting at the checkout counter with a shopping cart full of stuff.

Of course, he has a sufficient amount of money in his wallet. However, he prefers to use alternate means of payment if possible: luncheon vouchers, gift certificates, different types of coupons, etc. What makes the matter complicated is that the use of these items is often limited: e.g., luncheon vouchers can only be used to buy food, and gift certificates are often limited to a certain type of gifts.

Problem specification

You are given the number N of items in Karl's shopping cart and their prices. You are also given the number M of vouchers in his wallet, together with the information on their allowed use.

When paying for his shopping, Karl may use vouchers for a larger sum than the cost of the things he is buying. It is also possible to split an item's cost between multiple vouchers and use a voucher to pay for more than one item.

Compute the minimum amount of additional cash money Karl needs to pay for his shopping.

Input specification

The first line of input file contains an integer T specifying the number of test cases. T blocks follows, each block describes one test case. Each block is preceded by a blank line.

Each block starts with line containing two positive integers N (the number of items) and M (the number of vouchers). The second line contains N numbers, the i-th of them being the price of the i-th item in Karl's shopping cart. The third line contains M numbers, the i-th of them being the cash value of the i-th voucher Karl has in his wallet. M lines follow. Each line consists of a number K_i (the count of items such that you can pay for them using the i-th voucher) followed by K_i numbers (the numbers of those items; items are numbered from 1 to N).

Output specification

For each test case output a single number specifying how much cash money Karl needs to pay for his shopping.

Example

input

```
1

3 2
15 20 10
20 30
3 1 2 3
1 3
```

output

```
15
```

Solution

This problem can be transformed into a problem of finding maximum flow in the following way: We construct a graph with a following vertices:

- one source vertex,
- M vertices, each vertex corresponds to a voucher in Karl's wallet,
- n vertices, each vertex corresponds to an item in Karl's shopping cart,
- one sink vertex

and the following edges:

- Edges from source to vertices corresponding to vouchers. Capacity of an edge going from source to a vertex corresponding to i-th voucher is the value of that voucher.
- Edges from voucher vertices to item vertices. An edge going from a vertex corresponding to i-th voucher to vertex corresponding to j-th item exists and has inifinite capacity if and only if i-th voucher can be used to pay for j-th item.
- Edges from item vertices to sink. Capacity of an edge going from a vertex corresponding to i-th item is the price of that item.

Clearly each flow in from source vertex to sink vertex in this graph corresponds to a way of (partial) payment of the items in Karl's shopping cart by vouchers in his wallet. Additionaly to such a payment he needs to use (sum of prices of all items) minus (value of that flow) cash money to pay all his shopping. That means that in order to minimize the amount of cash money we have to maximize the flow through this graph. This can be done e.g. by Ford-Fulkerson algorithm.

Let there be rainbows!

Authors

Problemsetter: Michal Forišek
Task preparation: Michal Forišek, Peter Perešíni

Problem statement

Once upon a time there was the kingdom of Absurdistan. There were N cities in the kingdom.

As it is often the case, some pairs of cities were connected by roads. The Great Vizier of Absurdistan was a terrible scrooge, hence the road network was a tree. (I.e., for any pair of cities there was exactly one way how to get from one to the other, while travelling by roads.)

The roads were made of concrete blocks and they all had a dull grey color.

One day, the Great Vizier came to the conclusion that travelling along dull grey roads makes people unhappy. As he does not want to risk a revolution, he decided to paint the roads using happy colors – the 7 colors of the rainbow.

Each day, he picked two cities s_i and t_i, and a color c_i. Painters with an ample supply of the color c_i were sent to the city s_i, and then they travelled to t_i. On their way, whenever they encountered a road that did not have the color c_i, they painted it to this color.

Problem specification

You are given the number of cities N, the description of the road network, and the sequence of Great Vizier's orders.

For each order, compute the number of roads the painters had to repaint.

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 starts with line containing a positive integer N ($N \leq 1\,000\,000$). For convenience, the cities in Absurdistan are numbered 0 to $N-1$.

$N-1$ lines follow. Each of these lines contains two numbers a_i b_i ($0 \leq a_i, b_i < N$) – the endpoints of one road.

The next line contains a single integer Q ($Q \leq 200\,000$) specifying the number of orders.

Q lines follow, each describing one order. Each description consists of two integers s_i t_i ($0 \leq s_i, t_i < N$) and a string c_i (c_i is one of `red`, `orange`, `yellow`, `green`, `blue`, `indigo`, and `violet`).

In the easy input, you may also assume that for each test case except for one we have $NQ \leq 10^8$. In the remaining test case the road network is a simple line.

Output specification

For each test case output seven lines. Each of these lines should contain a color name (in the order given above) and a number. The number shall be the total number of times some road was painted using the corresponding color.

You may output empty lines between the test cases. (The amount of whitespace in your output does not matter anyway; we do this for readability in the example below.)

Example

input

```
2

8
0 4
7 5
5 6
3 5
1 4
4 2
4 3
4
0 7 red
2 6 orange
4 3 green
1 7 green

12
0 1
1 2
2 3
2 4
2 5
0 6
6 7
7 8
7 9
7 10
11 6
3
3 8 blue
6 1 violet
5 11 blue
```

output

```
red 4
orange 4
yellow 0
green 4
blue 0
indigo 0
violet 0

red 0
orange 0
yellow 0
green 0
blue 10
indigo 0
violet 2
```

We will explain the second test case in detail:

When executing the query "3 8 blue", we paint six roads blue: 3-2, 2-1, 1-0, 0-6, 6-7, and 7-8.

When executing the query "6 1 violet", we paint two roads violet: 6-0 and 0-1. (Both of these roads were previously blue.)

Finally, when executing the query "5 11 blue", we paint four roads blue: 5-2, 1-0, 0-6, and 6-11. Note that the road 2-1 was still blue, hence we did not paint it now.

Thus we painted a road blue 6+4=10 times, and we painted a road violet 2 times.

Solution

We will first show a solution to a special case that corresponded to the easy input: the case where the tree is in fact a simple path.

We can number the edges 1 to $N-1$ in order in which they lie on the path. Each of the edges can have one of eight possible colors.

Let's start by identifying the types of queries we need to process. We can break the query described in the problem statement into two parts:

- given x, y and c, count the edges between x and y inclusive that have color c

- given x, y and c, color all edges between x and y inclusive by color c

We will use an interval tree in order to answer each of the queries in $O(\log N)$.
In each vertex of the interval tree, we will store the following information:

- a boolean stating whether the entire interval represented by this vertex has the same color
- if yes, an integer id of that color
- for each color, the number of edges in our interval that have this color (based on the information stored in our subtree only).

To process a query of the first type, we take the given interval $[x, y]$, recursively descend down the tree and sum up the stored information. To process a query of the second type, we again recursively descend down the tree and update the information. One important issue that needs to be handled correctly are the vertices that were fully colored before we started processing the query. The easiest way how to handle these: whenever we encounter such a vertex during our descent, we mark it as not full, and instead mark both its children as fully painted using the parent's color.

Before we present the general solution, one simple trick. Imagine that you run depth-first search on a rooted tree. For each vertex v, one can store two times: the time $t_d(v)$ when it was discovered, and the time $t_f(v)$ when we finished processing it.

From the nature of depth-first search it is obvious that for any two vertices v, w the intervals $[t_d(v), t_f(v)]$ and $[t_d(w), t_f(w)]$ are either disjoint or nested one inside another. And more precisely, the second case happens if and only if the vertex with the longer interval is an ancestor of the vertex with the shorter one.

This observation can be used in the following way: In $O(N)$ we can precompute all values $t_d(\cdot)$ and $t_f(\cdot)$, and then we can use them to answer queries of the type "is x an ancestor of y?" in constant time.

We will now show a solution to the general problem. Our solution preprocesses a tree with N vertices in $O(N)$ time, and then answers each query in $O(\log^2 N)$ time. Hence the total time complexity of our solution will be $O(N + Q \log^2 N)$.

The first step in our solution is to compute the *heavy-light decomposition* of the given tree. We root the given tree at an arbitrary vertex, and for each vertex v we compute the size of its subtree $s(v)$ (i.e., the number of vertices it contains). Now, let v be an arbitary vertex. If v has a child c such that $s(c) \geq s(v)/2$, then the edge $v - c$ is called *heavy*, otherwise the edge is called *light*.

Note that each vertex can only have at most one child connected by a heavy edge. Hence the heavy edges in a tree form a set of vertex-disjoint paths.

Now we can take the entire tree and decompose it into at most $N - 1$ edge-disjoint paths as follows: Let S be the set of vertices that don't have any child connected by a heavy edge. For each of these vertices we construct one path as follows: we start traveling towards the root, stop after we use the first light edge, and take all edges we used to form a single path. We'll call these paths *important paths* and number them 1 to P.

What does this decomposition accomplish?

Imagine that you are traveling down the tree, starting at the root. How many times can you use a light edge? Clearly, $\lg N$ is an upper bound, because each time you use a light edge the number of vertices in your subtree is at least halved.

We just showed that for any vertex v the path from v to the root only contains at most $\lg N$ light edges. Now imagine that as we walk along the path from v to the root, we are keeping track of the important path we are on. This only changes when we use a light edge. And as there are at most $\lg N$ light edges on our path, we only change the important path at most $\lg N$ times.

Almost the same is true for a path between two arbitrary vertices v and w: Let x be their least common ancestor in the rooted tree. The path from v to w can be split into two paths, v to x and x to w. Each of these two paths only uses edges from at most $\lg N$ important paths, hence the entire path from v to w crosses at most $2 \lg N$ important paths.

For each important path, we will keep an interval tree that will be used in the same way as in the solution to the easy problem.

Hence to answer a query, we first identify the segments of important paths it uses. This can be done in $O(\log N)$. Then we process each segment separately. Processing each segment involves two queries on the interval tree for its important path, hence the total time complexity is $O(\log^2 N)$ per query.

Muzidabutur

Authors

Problemsetter: Peter Perešíni, Michal Nánási, Jozef Šiška
Task preparation: Michal Nánási, Jozef Šiška

Problem statement

Muzidabutur is a very powerful magic phrase. For example, if you write the phrase on a monitor using a non-erasable marker, your local computer administrator will throw you into a waste bin (and we do not mean the folder). The problem is that nobody really remembers this phrase anymore.

The famous archeologist Alabama Steve recently discovered two ancient tomes. In one of them he found a short description of the Muzidabutur. He suspects that one of the sentences in the other tome is Muzidabutur. But the tome is way too thick and he does not have the time to process it by hand.

Problem specification

You are given a description of Muzidabutur, and N queries. For each query decide whether it can be the Muzidabutur.

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 starts with line containing a textual description of the Muzidabutur. You may assume that spaces and characters '`(`' and '`)`' will not occur inside strings and character sets used to define the Muzidabutur.

The next line contains N, which is the number of sentences in the second tome. Each of the next N lines contains one sentence.

Output specification

For each sentence output the string "`YES`" if it exactly matches the description of a Muzidabutur, and output "`NO`" otherwise.

Example

input

```
1

THE LETTER A FOLLOWED BY (ONE OF THE LETTERS abc OR THE STRING gg) AT LEAST 3 TIMES
4
Abggaggc
abbb
Abc
Hello, Abggaggc, how are you?
```

output

```
YES
NO
NO
NO
```

Note that the Muzidabutur is case sensitive, therefore the second sentence is not a Muzidabutur.
Also, the description must match the entire sentence, therefore the last sentence is not a Muzidabutur.

Solution

This solution requires understanding of regular expressions. If you are not familiar with them, we recommended to visit page `http://www.regular-expressions.info`.

The description of Muzidabutur is a regular expression (regexp). Probably your first idea was to write program that matches regexp written in Muzidabutur syntax. This method works, but it is not the easiest one, because there are so many tools for working with regular expressions. A better idea is to rewrite the description of Muzidabutur into some normal regexp syntax and then to use existing tools like `grep`, `sed`, or programming languages with built-in regular expressions.

The transformation from the description to a regexp can *also* be done by using regexp tools (we used `sed`). In this solution we use extended regexps (ERE) as specified by the POSIX standard. It is easy to derive following rules.

1. Pattern '`THE STRING XXXXX`' changes to '`(XXXXX)`'
2. Pattern '`AT LEAST XX TIMES`' changes to '`\{12,\}`'. Similar rules for other types of iteration.
3. Pattern '`OR`' changes to '`|`'
4. Pattern '`FOLLOW BY`' changes to an empty string.
5. Pattern '`ONE OF THE CHARACTERS XXXXX`' changes to '`[XXXXX]`'.
6. Pattern '`ANY CHARACTER EXCEPT XXXXX`' changes to '`[\^XXXXX]`'.
7. Erase all spaces and all backslashes before brackets, add '`\^\\(`' at the beginning of the regexp and '`\\)\$`' at the end of the regexp (we want to enclose whole expression in brackets – '`^a|b\$`' is not correct).

The rules described above are enough to solve the easy input and most test cases from the hard input. In the hard input there were 10 cases with special characters in strings and character sets, and additionally they also contain special characters and sequences from other types of regular expressions. For example, there were character sets '`[:digit]`', '`a][bc`', '`\^47`', and strings containing '`*`', '`\\+`', '`\\t`', '`\\s`' and others. These special cases could be resolved manually, or by improving our transformation rules (i.e., by fixing rules 1, 5 and 6). New rules are listed below.

1 Pattern '`THE STRING XXXXX`' changes to '`([[.X.]][[.X.]][[.X.]][[.X.]][[.X.]])`' – or just add '`\\`' before the special characters.

5 Pattern '`ONE OF THE CHARACTERS XXXXX`' changes to '`[[.X.][.X.][.X.][.X.][.X.]]`'. Another solution is to rearrange characters into proper order ('`\^`' is not on the beginning, '`-`' is on the end, '`]`' is on the beginning).

6 Pattern '`ANY CHARACTER EXCEPT XXXXX`' changes to '`[\^[.X.][.X.][.X.][.X.][.X.]]`'.