Colombian Collegiate Programming League
CCPL 2017

Round 1 – February 18

Problems

This set contains 10 problems; pages 1 to 15.

(Borrowed from several sources online.)

Official site `http://programmingleague.org`

Follow us on Twitter `@CCPL2003`

A - Bounding Box

Source file name: `bounding.c`, `bounding.cpp`, `bounding.java`, *or* `bounding.py`

The Archeologists of the Current Millenium (ACM) now and then discover ancient artifacts located at vertices of regular polygons. The moving sand dunes of the desert render the excavations difficult and thus once three vertices of a polygon are discovered there is a need to cover the entire polygon with protective fabric.

Input

The input contains multiple cases. Each case describes one polygon. It starts with an integer $n \leq 50$, the number of vertices in the polygon, followed by three pairs of real numbers giving the x and y coordinates of three vertices of the polygon. The numbers are separated by whitespace. The input ends with a n equal 0, this case should not be processed.

The input must be read from standard input.

Output

For each line of input, output one line in the format shown below, giving the smallest area of a rectangle which can cover all the vertices of the polygon and whose sides are parallel to the x and y axes. Output the area rounding up and with exactly three decimal places.

The output must be written to standard output.

Sample Input

```
4
10.00000 0.00000
0.00000 -10.00000
-10.00000 0.00000
6
22.23086 0.42320
-4.87328 11.92822
1.76914 27.57680
23
156.71567 -13.63236
139.03195 -22.04236
137.96925 -11.70517
0
```

Sample Output

```
Polygon 1: 400.000
Polygon 2: 1056.172
Polygon 3: 397.673
```

B - Balls and Needles

Source file name: `balls.c`, `balls.cpp`, `balls.java`, *or* `balls.py`

Joana Vasconcelos is a Portuguese artist who uses everyday objects in her creations, like electric irons or plastic cutlery. She is an inspiration to Ana, who wants to make ceiling hanging sculptures with straight knitting needles and balls of wool. For safety reasons, there will be a ball at each end of each needle. Knitting needles vary in colour, length and thickness (to allow intersections of needles).

Sculptures are to be exhibited in room corners, which provide a 3D Cartesian coordinate system, with many lamps on the ceiling. Sculpture designs are made with the coordinates of the centres of the balls of wool in which knitting needles are stuck. That is, each needle N is represented by a set of two different triples: $N = \{(x, y, z), (x', y', z')\}$.

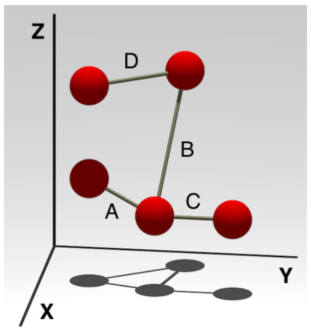
Ana dislikes *closed chains*. A *true closed chain* is a sequence of k distinct needles, $N_1, N_2, \ldots, N_k$ (for some $k \geq 3$), such that:

$$\begin{aligned} N_1 &= \{(x_1, y_1, z_1), (x_2, y_2, z_2)\}, \\ N_2 &= \{(x_2, y_2, z_2), (x_3, y_3, z_3)\}, \ldots, \\ N_k &= \{(x_k, y_k, z_k), (x_{k+1}, y_{k+1}, z_{k+1})\}, \text{ and} \\ (x_{k+1}, &y_{k+1}, z_{k+1}) = (x_1, y_1, z_1). \end{aligned}$$

But her dislike of closed chains is so extreme that the shadow of the sculpture on the floor has to be free of "floor closed chains". Given any needle $N = \{(x, y, z), (x', y', z')\}$, let $N^{\downarrow} = \{(x, y), (x', y')\}$ denote the shadow of needle N on the floor. For Ana (who is an artist), a floor closed chain is also a sequence of k distinct needles, $N_1, N_2, \ldots, N_k$ (for some $k \geq 3$), such that:

$$N_i^{\downarrow} \neq N_j^{\downarrow}, \text{ for every } 1 \leq i < j \leq k \text{ (the } k \text{ needle shadows are all distinct)}$$
$$N_1^{\downarrow} = \{(x_1, y_1), (x_2, y_2)\}, N_2^{\downarrow} = \{(x_2, y_2), (x_3, y_3)\}, \ldots$$
$$N_k^{\downarrow} = \{(x_k, y_k), (x_{k+1}, y_{k+1})\}, \text{ and } (x_{k+1}, y_{k+1}) = (x_1, y_1).$$

Consider the sculpture depicted in the following figure:



This sculpture has the following four knitting needles:

$$A = \{(12, 12, 8), (10, 5, 11)\} \qquad B = \{(12, 12, 8), (4, 14, 21)\}$$
$$C = \{(12, 12, 8), (12, 20, 8)\} \qquad D = \{(4, 14, 21), (10, 5, 21)\}.$$

Write a program that, given the knitting needles of a sculpture, determines whether there is a true or a floor closed chain in the sculpture.

Input

The input file contains several test cases. The first line of each test case has one integer, K, which is the number of knitting needles in the sculpture. Each of the following K lines contains six integers, $x_1, y_1, z_1, x_2, y_2, z_2$, which indicate that $\{(x_1, y_1, z_1), (x_2, y_2, z_2)\}$ is the set of triples of a needle. Any two distinct needles are represented by different sets of triples. You can assume $1 \leq K \leq 50\,000$ and $1 \leq x_i, y_i, z_i < 1\,000$.

The input must be read from standard input.

Output

For each test case, the output has two lines, each one with a string. The string in the first line is:

```
True closed chains
```

if there is some true closed chain in the sculpture;

```
No true closed chains
```

otherwise. The string in the second line is:

```
Floor closed chains
```

if there is some floor closed chain in the sculpture;

```
No floor closed chains
```

otherwise.

The output must be written to standard output.

Sample Input

```
4
12 12 8 10 5 11
12 12 8 4 14 21
12 12 8 12 20 8
4 14 21 10 5 21
3
50 50 50 100 100 100
100 100 100 50 50 90
50 50 90 50 50 50
4
1 1 1 2 2 2
2 2 2 1 5 5
9 4 4 9 4 2
9 4 4 9 9 4
3
1 1 5 1 3 7
1 3 7 4 4 5
4 4 5 1 1 5
```

Sample Output

```
No true closed chains
Floor closed chains
True closed chains
No floor closed chains
No true closed chains
No floor closed chains
True closed chains
Floor closed chains
```

C - Color Length

Source file name: `color.c`, `color.cpp`, `color.java`, *or* `color.py`

Cars painted in different colors are moving in a row on the road as shown in Figure 1. The color of each car is represented by a single character and the distance of two adjacent cars is assumed to be 1. Figure 1 shows an example of such cars on the road. For convenience, the numbers in Figure 1 represent the locations of each car.

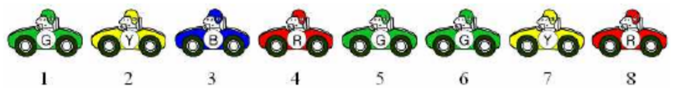


Figure 1. Cars in different colors on the road

For any color *c*, *location(c)* represents set of the locations of the cars painted in color *c* and color length $L(c)$ is defined as follows:

$$L(c) \quad = \quad \max \mathit{location(c)} \quad - \quad \min \mathit{location(c)}.$$

For example, according to Figure 1, $location(G) = \{1, 5, 6\}$, $location(Y) = \{2, 7\}$, $location(B) = \{3\}$, and $location(R) = \{4, 8\}$. Hence the color length of each color and the sum of the color lengths are as follows:

Color	G	Y	B	R	**Sum**
$L(c)$	5	5	0	4	14

In Gyeongju City, almost all the roads including the main street of the city were constructed more than 500 years ago. The roads are so old that there are a lot of puddles after rain. Visitors have complained about the bad condition of the roads for many years. Due to the limited budget, the mayor of the city decided to repair firstly the main street of the city, which is a four-lane road, two lanes for each direction.

However, since the main street is a backbone of the city, it should not be blocked completely while it is under repair, or it is expected that serious traffic jams will occur on almost all the other roads in the city. To allow cars to use the main street during the repair period, the city decided to block only two lanes, one lane for each direction. Hence, the cars in the two lanes for each direction should merge into a single lane before the blocked zone.

For instance, as shown in Figure 2, cars in the two lanes merge into a single lane as shown in Figure 3. To differentiate the cars in the same color, a unique identifier is assigned to each car.

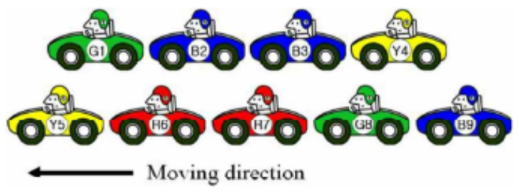


Figure 2. Cars moving in two lanes before merging

Figure 3 shows two different merging scenarios after merging the cars from the two lanes. As shown in Figure 3, cars in the two lanes do not necessarily merge one by one from each lane. The distance between two adjacent cars after merging is also assumed 1.

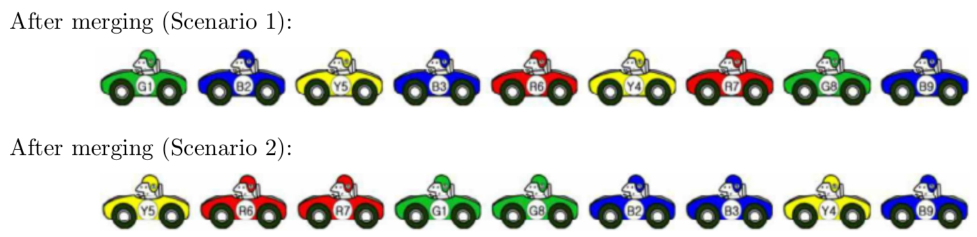


Figure 3. Two different merging scenarios

For each merging scenario shown in Figure 3, the color length for each color and the sum of the color lengths are as follows:

Color	G	Y	B	R	**Sum**
$L(c)$: Scenario 1	7	3	7	2	19
$L(c)$: Scenario 2	1	7	3	1	12

As you can imagine, there are many different ways of merging other than the two examples shown in Figure 3.

Given two character strings which represent the color information of the cars in the two lanes before merging, write a program to find the sum of color lengths obtained from the character string, which is the color information of cars after merging, such that the sum of color lengths is minimized.

Input

The input consists of T test cases. The number of test cases T is given in the first line of the input. Each test case consists of two lines. In the first line, a character string of length n ($1 \leq n \leq 5\,000$) that is the color information of the cars in one lane before merging is given. In the second line, a character string of length m ($1 \leq m \leq 5\,000$) that is the color information of the cars in the other lane is given. Every color is represented as an uppercase letter in English, hence the number of colors is less than or equal to 26.

The input must be read from standard input.

Output

Print exactly one line for each test case. The line should contain the sum of color lengths after merging the cars in the two lanes optimally as described above.

The output must be written to standard output.

Sample Input	**Sample Output**
2 AAABBCY ABBBCDEEY GBBY YRRGB	10 12

D - The Bus Driver Problem

Source file name: `driver.c`, `driver.cpp`, `driver.java`, *or* `driver.py`

In a city there are n bus drivers. Also there are n morning bus routes and n afternoon bus routes with various lengths. Each driver is assigned one morning route and one evening route. For any driver, if his total route length for a day exceeds d, he has to be paid overtime for every hour after the first d hours at a flat r taka per hour. Your task is to assign one morning route and one evening route to each bus driver so that the total overtime amount that the authority has to pay is minimized.

Input

The first line of each test case has three integers n, d, and r, as described above. In the second line, there are n space separated integers which are the lengths of the morning routes given in meters. Similarly the third line has n space separated integers denoting the evening route lengths. The lengths are positive integers less than or equal to 10000. The end of input is denoted by a case with three 0's. You can assume $1 \leq n \leq 100$, $1 \leq d \leq 10\,000$, and $1 \leq r \leq 5$.

The input must be read from standard input.

Output

For each test case, print the minimum possible overtime amount that the authority must pay.

The output must be written to standard output.

Sample Input	**Sample Output**

`2 20 5`
`10 15`
`10 15`
`2 20 5`
`10 10`
`10 10`
`0 0 0`
	`50`
	`0`

E - Soda Surpler

Source file name: `soda.c`, `soda.cpp`, `soda.java`, *or* `soda.py`

Tim is an absolutely obsessive soda drinker, he simply cannot get enough. Most annoyingly though, he almost never has any money, so his only obvious legal way to obtain more soda is to take the money he gets when he recycles empty soda bottles to buy new ones. In addition to the empty bottles resulting from his own consumption he sometimes find empty bottles in the street. One day he was extra thirsty, so he actually drank sodas until he couldn't afford a new one.

Input

The first line of the input file contains an integer N ($N \geq 0$) which denotes the total number of test cases. Each test case consists o a single line with three non-negative integers e, f, c where $e < 1000$ equals the number of empty soda bottles in Tim's possession at the start of the day, $f < 1000$ the number of empty soda bottles found during the day, and $1 < c < 2000$ the number of empty bottles required to buy a new soda.

The input must be read from standard input.

Output

For each test case print a single line with the number of sodas Tim drank on his extra thirsty day.

The output must be written to standard output.

Sample Input	Sample Output

2
9 0 3
5 5 2
	4
	9

F - Forwarding Emails

Source file name: `forward.c`, `forward.cpp`, `forward.java`, *or* `forward.py`

"... so forward this to ten other people, to prove that you believe the emperor has new clothes." Aren't those sorts of emails annoying? Martians get those sorts of emails too, but they have an innovative way of dealing with them. Instead of just forwarding them willy-nilly, or not at all, they each pick one other person they know to email those things to every time – exactly one, no less, no more (and never themselves). Now, the Martian clan chieftain wants to get an email to start going around, but he stubbornly only wants to send one email. Being the chieftain, he managed to find out who forwards emails to whom, and he wants to know: which Martian should he send it to maximize the number of Martians that see it?

Input

The first line of input will contain T ($T \geq 0$) denoting the number of cases. Each case starts with a line containing an integer N ($2 \leq N \leq 50\,000$) denoting the number of Martians in the community. Each of the next N lines contains two integers: u, v ($1 \leq u, v \leq N$ and $u \neq v$) meaning that Martian u forwards email to Martian v.

The input must be read from standard input.

Output

For each case, print the case number and an integer m, where m is the Martian that the chieftain should send the initial email to. If there is more than one correct answer, output the smallest number.

The output must be written to standard output.

Sample Input	Sample Output

3
3
1 2
2 3
3 1
4
1 2
2 1
4 3
3 2
5
1 2
2 1
5 3
3 4
4 5
	Case 1: 1
	Case 2: 4
	Case 3: 3

G - Garden of Eden

Source file name: `garden.c`, `garden.cpp`, `garden.java`, *or* `garden.py`

Cellular automata are mathematical idealizations of physical systems in which both space and time are discrete, and the physical quantities take on a finite set of discrete values. A cellular automaton consists of a lattice (or array), usually infinite, of discrete-valued variables. The state of such automaton is completely specified by the values of the variables at each place in the lattice. Cellular automata evolve in discrete time steps, with the value at each place (cell) being affected by the values of variables at sites in its neighborhood on the previous time step. For each automaton there is a set of rules that define its evolution.

For most cellular automata there are configurations (states) that are unreachable: no state will produce them by the application of the evolution rules. These states are called Gardens of Eden for they can only appear as initial states. As an example consider a trivial set of rules that evolve every cell into 0; for this automaton any state with non-zero cells is a Garden of Eden.

In general, finding the ancestor of a given state (or the non-existence of such ancestor) is a very hard, compute intensive, problem. For the sake of simplicity we will restrict the problem to 1-dimensional binary finite cellular automata. This is, the number of cells is a finite number, the cells are arranged in a linear fashion and their state will be either ‘0’ or ‘1’. To further simplify the problem each cell state will depend only on its previous state and that of its immediate neighbors (the one to the left and the one to the right).

The actual arrangement of the cells will be along a circumference, so that the last cell is next to the first.

Given a circular binary cellular automaton you must find out whether a given state is a Garden of Eden or a reachable state. The cellular automaton will be described in terms of its evolution rules. For example, the table below shows the evolution rules for the automaton: *Cell = XOR(Left, Right)*.

Left $[i-1]$	Cell $[i]$	Right $[i+1]$	New State		
0	0	0	0	$0 * 2^0$	
0	0	1	1	$1 * 2^1$	
0	1	0	0	$0 * 2^2$	
0	1	1	1	$1 * 2^3$	
1	0	0	1	$1 * 2^4$	
1	0	1	0	$0 * 2^5$	
1	1	0	1	$1 * 2^6$	
1	1	1	0	$0 * 2^7$	
				90	= Automaton Identifier

Notice that, with the restrictions imposed to this problem, there are only 256 different automata. An identifier for each automaton can be generated by taking the New State vector and interpreting it as a binary number (as shown in the table). For instance, the automaton in the table has identifier 90. The Identity automaton (every state evolves to itself) has identifier 204.

Input

The input will consist of several test cases. Each input case will describe, in a single line, a cellular automaton and a state. The first item in the line will be the identifier of the cellular automaton you must work with. The second item in the line will be a positive integer *N* ($4 \leq N \leq 32$) indicating the number of cells for this test case. Finally, the third item in the line will be a state represented by a string of exactly N zeros and ones. Your program must keep reading lines until the end of the input.

The input must be read from standard input.

Output

If an input case describes a Garden of Eden you must output the string `GARDEN OF EDEN`. If the input does not describe a Garden of Eden (it is a reachable state) you must output the string `REACHABLE`. The output for each test case must be in a different line.

The output must be written to standard output.

Sample Input	**Sample Output**
`0 4 1111`	`GARDEN OF EDEN`
`204 5 10101`	`REACHABLE`
`255 6 000000`	`GARDEN OF EDEN`
`154 16 1000000000000000`	`GARDEN OF EDEN`

H - Through the Desert

Source file name: `desert.c`, `desert.cpp`, `desert.java`, *or* `desert.py`

Imagine you are an explorer trying to cross a desert. Many dangers and obstacles are waiting on your path. Your life depends on your trusty old jeep having a large enough fuel tank. But how large exactly does it have to be? Compute the smallest volume needed to reach your goal on the other side.

The following events describe your journey:

Fuel consumption *n* means that your truck needs *n* litres of gasoline per 100km. The number *n* is an integer in the range [1..30]. Fuel consumption may change during your journey, for example when you are driving up or down a mountain.

Leak means that your truck's fuel tank was punctured by a sharp object. The tank will start leaking gasoline at a rate of 1 litre of fuel per kilometre. Multiple leaks add up and cause the truck to lose fuel at a faster rate.

However, not all events are troublesome in this desert. The following events increase your chances of survival:

Gas station lets you fill up your tank.

Mechanic fixes all the leaks in your tank.

And finally:

Goal means that you have safely reached the end of your journey!

Input

The input consists of a series of test cases. Each test case consists of at most 50 events. Each event is described by an integer, the distance (in kilometres measured from the start) where the event happens, followed by the type of event as above. In each test case, the first event is of the form '`0 Fuel consumption n`', and the last event is of the form '`d Goal`' (*d* is the distance to the goal). Events are given in sorted order by non-decreasing distance from the start, and the 'Goal' event is always the last one. There may be multiple events at the same distance-process them in the order given. Input is terminated by a line containing '`0 Fuel consumption 0`' which should not be processed.

The input must be read from standard input.

Output

For each test case, print a line containing the smallest possible tank volume (in litres, with three digits precision after the decimal point) that will guarantee a successful journey.

The output must be written to standard output.

Sample Input	Sample Output

```
0 Fuel consumption 10
100 Goal
0 Fuel consumption 5
100 Fuel consumption 30
200 Goal
0 Fuel consumption 20
10 Leak
25 Leak
25 Fuel consumption 30
50 Gas station
70 Mechanic
100 Leak
120 Goal
0 Fuel consumption 0
```
	```
	10.000
	35.000
	81.000
	```

I - Passwords

Source file name: `pass.c`, `pass.cpp`, `pass.java`, *or* `pass.py`

It's that time of the year again when you go back to work and need to choose new passwords for all your services. The rules enforced by the system administrators are very strict, and the password you choose must obey the following restrictions:

- It must contain only letters and digits.
- It must have between A and B characters (inclusive).
- It must have at least one lowercase letter, one uppercase letter and one digit.
- It cannot contain any word of a collection of forbidden words (the blacklist).

A word of the blacklist is considered to be contained in the password if it appears as a substring, that is, if it occurs as a consecutive sequence of characters (regardless of it being upper or lower case). For instance, `swerc` is a substring of `SwErC`, `2016swerc2016`, or `SWERC16`, but it is not a substring of `ICPC` or `sw16erc`.

Additionally, for the purposes of avoiding the blacklist, you cannot use *l33t*. Specifically, some digits can be interpreted as letters, namely the 0 ('o'), 1 ('i'), 3 ('e'), 5 ('s') and 7 ('t'). This implies that for example `5w3rC` would be an occurrence of `swerc`, and that `abcL337def` contains the word `leet`.

You cannot stop thinking about all these rules and you wonder how many different valid passwords there are... Can you calculate how many passwords would obey these rules?

Given a blacklist with N words and two integers A and B, your task is to compute the number of different valid passwords that exist following the given constraints: made up of only letters and digits; length between A and B (inclusive); at least one lowercase letter, one uppercase letter, and one digit; no blacklisted substring. Since this number can be very big, compute it modulo 1 000 003.

Input

The input file contains several test cases. The first line of each test case contains two integers, A and B, specifying respectively the minimum and maximum length of the password. The second line contains an integer N, the number of words of the blacklist. The following N lines each contains a string W_i indicating a word in the blacklist. These words are formed only by lowercase letters. You can assume $3 \leq A \leq B \leq 20$, $0 \leq N \leq 50$, $1 \leq |W_i| \leq 20$.

The input must be read from standard input.

Output

For each test case, the output should contain a single line with an integer indicating the number of valid passwords modulo 1 000 003. A valid password is one that respects all of the given constraints.

The output must be written to standard output.

Sample Input	Sample Output
3 5 9 swerc icpc fbi cia bio z hi no yes	607886

J - Cacho

Source file name: `cacho.c`, `cacho.cpp`, `cacho.java`, *or* `cacho.py`

In Bolivia there is a very popular game called “Cacho”. The game consists rolling five dices (a_1, a_2, a_3, a_4, a_5) and then annotate the result according to certain rules. This time we will focus on one case in particular: “escala”. A “escala” is the scene in which the dices form a sequence of consecutive numbers. More formally a “escala” meets the property:

$$a_i + 1 = a_{i+1}; \qquad 1 \le i \le 4$$

There are two types of “escala”: a ordinary “escala” (it satisfy the property described above), and a “Escala Real” (when the scenery is $1, 3, 4, 5, 6$; in the game this case is also a valid “scala”). Cael is a boy who is learning to play and wants you to help develop a program to check when five dices are forming a “escala”. Note that the “Escala Real” is not a valid “escala” for Cael.

Input

The input begins with a number T ($T \ge 0$), the number of test cases. Below are T lines, each with five numbers a_i ($1 \le a_i \le 6$) in non-decreasing order.

The input must be read from standard input.

Output

In each case, if the five dices form a scale print in one line ‘`Y`’. Otherwise print in one line ‘`N`’ (quotes for clarity).

The output must be written to standard output.

Sample Input	Sample Output

5
1 2 3 4 5
2 3 4 5 6
1 4 4 4 5
1 3 4 5 6
1 2 2 3 6
	Y
	Y
	N
	N
	N