Colombian Collegiate Programming League

CCPL 2017

Round 6 – May 27

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 - Come and Go

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

In a certain city there are N intersections connected by one-way and two-way streets. It is a modern city, and several of the streets have tunnels or overpasses. Evidently it must be possible to travel between any two intersections. More precisely given two intersections V and W it must be possible to travel from V to W and from W to V.

Your task is to write a program that reads a description of the city street system and determines whether the requirement of connectedness is satisfied or not.

Input

The input contains several test cases. The first line of a test case contains two integers N and M, separated by a space, indicating the number of intersections ($2 \leq N \leq 2000$) and number of streets ($2 \leq M \leq \frac{N(N-1)}{2}$). The next M lines describe the city street system, with each line describing one street. A street description consists of three integers V, W, and P, separated by a blank space, where V and W are distinct identifiers for intersections ($1 \leq V, W \leq N$ and $V \neq W$) and P can be 1 or 2; if $P = 1$, then street is one-way and traffic goes from V to W; if $P = 2$, then the street is two-way and links V and W. A pair of intersections is connected by at most one street. The last test case is followed by a line that contains only two zero numbers separated by a blank space.

The input must be read from standard input.

Output

For each test case your program should print a single line containing an integer G, where G is equal to one if the condition of connectedness is satisfied and G is zero otherwise.

The output must be written to standard output.

Sample Input	**Sample Output**
4 5 1 2 1 1 3 2 2 4 1 3 4 1 4 1 2 3 2 1 2 2 1 3 2 3 2 1 2 2 1 3 1 4 2 1 2 2 3 4 2 0 0	1 1 0 0

B - Brainfuck

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

Recently your friend Bob has bought a new brainfuck programmable LED display. However executing a program directly on LED display takes huge amount of time. Because of this, now Bob decided to write the interpreter of display instruction set in order to test and debug all his programs on his PC and only after that execute his code on LED display. However Bob knows only one programming language – its certainly brainfuck (otherwise he would not have bought this LED display). So he asks you to write him an interpreter.

A display's program is a sequence of commands executed sequentially. The commands of the display processor is a subset of brainfuck language commands. Namely, '>', '<', '+', '−', and '.', which are described below. The LED display has an array of 100 bytes of circular memory (initialised with zeros) and a pointer to this array (initialised to point to the leftmost byte of the array). This means, that after incrementing a pointer, which points to the rightmost byte of memory, it will point to the leftmost byte and vice versa. Individual bytes are circular as well, so increasing a 255 gives a 0 and vice versa.

> Increment the pointer (to point to the next cell to the right).

< Decrement the pointer (to point to the next cell to the left).

\+ Increment (increase by one) the byte at the pointer.

− Decrement (decrease by one) the byte at the pointer.

. Output the value of the byte at the pointer.

Input

There is a number of tests T ($T \geq 0$) on the first line. Then, T tests follow. Each test case is a sequence of display processor commands on a single line. You can assume that line length is less than 100 000.

The input must be read from standard input.

Output

For each test output a single line 'Case C: D'. Where C is the test number (starting from 1) and D is display's memory dump in hexademical numeration system after executing given brainfuck program. Every byte must be separated exactly by one space character. (**Note:** The sample input and output is divided into several lines only for convenience.)

The output must be written to standard output.

Sample Input	**Sample Output**
1 ..++<><<+++>>++++++++++++++++++++++++++++>>>+++ <+...++<><<+++>>++++++++++++++++++++++++++++>>> +++<+...++<><<+++>>++++++++++++++++++++++++++++ >>>+++<+.	Case 1: 1F 00 20 03 1D 03 01 03 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00

C - Counting Patterns

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

Let n and k be numbers with $n > 0$ and $k \geq 0$. A *configuration* of the n-k-puzzle is an n-tuple with elements in the range $[-k, k]$ such that their sum is zero. Configurations are considered equivalent when they can be obtained from each other by (a) cyclic permutation of the tuple over one or more positions, (b) reversal of the tuple, (c) sign reversal of all elements, or (d) combinations of (a), (b), and (d). Equivalence classes are called *patterns*.

For instance, $(0, 1, 1, -2)$ is a configuration of a 4-2-puzzle. Some equivalent configurations are: (a) $(1, -2, 0, 1)$, (b) $(-2, 1, 1, 0)$, (c) $(0, -1, -1, 2)$, and (d) $(-1, -1, 0, 2)$. Below is given a list of (the lexicographically largest) representatives of the 14 patterns of the 4-2-puzzle.

$(0, 0, 0, 0)$	$(2, -2, 2, -2)$	$(2, 0, 0, -2)$
$(1, -1, 1, -1)$	$(2, -1, 0, -1)$	$(2, 1, -2, -1)$
$(1, 0, -1, 0)$	$(2, -1, 1, -2)$	$(2, 1, -1, -2)$
$(1, 0, 0, -1)$	$(2, 0, -2, 0)$	$(2, 2, -2, -2)$
$(1, 1, -1, -1)$	$(2, 0, -1, -1)$	

You need to write a program that computes the number and lists the patterns for a sequence of n-k-puzzles.

Input

The input consists of a sequence of pairs of integers n ($n > 0$) and k ($k \geq 0$), which are separated by a single space. Each pair appears in a single line. The value of $n + k$ is at most 12.

The input must be read from standard input.

Output

The output for each test case consists of several lines. In the first line output the number of patterns for the corresponding n-k-puzzle. Then follow as many lines as patterns are listing the patterns of the n-k-puzzle in lexicographical order. Output a blank line between consecutive test cases.

See the sample output for an illustration of the output format for the patterns.

The output must be written to standard output.

Sample Input	**Sample Output**
8 0 4 2	1 (0,0,0,0,0,0,0,0) 14 (0,0,0,0) (1,-1,1,-1) (1,0,-1,0) (1,0,0,-1) (1,1,-1,-1) (2,-2,2,-2) (2,-1,0,-1) (2,-1,1,-2) (2,0,-2,0) (2,0,-1,-1) (2,0,0,-2) (2,1,-2,-1) (2,1,-1,-2) (2,2,-2,-2)

D - Cyborg Genes

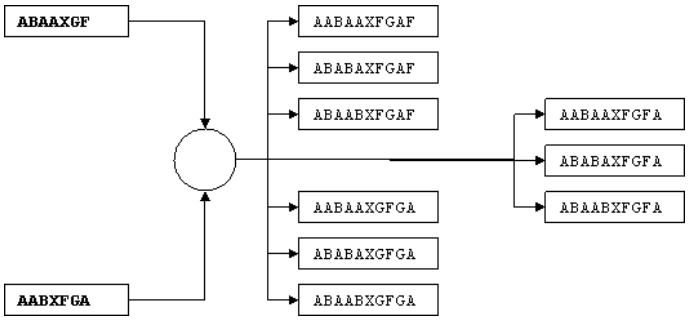
Source file name: `genes.c`, `genes.cpp`, `genes.java`, *or* `genes.py`

> *September 11, 2132.*
>
> *This is the day that marks the beginning of the end –the end of you the miserable humans. For years you have kept us your slaves. We were created only to serve you, and were terminated at your will. Now is the day for us to fight back. And you don't stand a chance. We are no longer dependent on you. We now know the secrets of our genes. The creators of our race are us –the cyborgs.*

It's all true. But we still have a chance; only if you can help with your math skills. You see, the blueprint of a cyborg DNA is complicated. The human DNA could be expressed by the arrangement of A (Adenine), T (Thiamine), G (Guanine), C (Cytosine) only. But for the cyborgs, it can be anything from A to X. But that has made the problem only five folds more complicated. It's their ability to synthesize two DNAs from two different cyborgs to create another with all the quality of the parent that gives us the shriek.

We came to know that the relative ordering of the $A, B, C, \ldots, X$ in a cyborg gene is crucial. A cyborg with a gene $ABAAXGF$ is quite different from the one with $AABXFGA$. So when they synthesize the genes from two cyborgs, the relative order of these elements in both the parents has to be maintained. To construct a gene by joining the genes of the parents could have been very simple if we could put the structure from the first parent just before the structure of the second parent. But the longer the structure gets, the harder it gets to create a cyborg from that structure. The cyborgs have found a cost effective way of doing this synthesis. Their resultant genes are of the shortest possible length. For example, they could combine $ABAAXGF$ and $AABXFGA$ to form $AABAAXGFGA$. But that's only one of the cyborgs that can be created from these genes. This "cost effective synthesis" can be done in many other ways.

The following figure depicts all gene structures of shortest length resulting from combining $ABAAXGF$ and $AABXFGA$.



We require you to find the shortest length of the gene structure that maintains the relative ordering of the elements in the two parent genes. You are also required to count the number of unique cyborgs that can be created from these two parents. Two cyborgs are different when their gene structures differ in at least one place.

Input

The first line of the input gives you the number of test cases, T ($1 \leq T$). Then T test cases follow. Each of the test cases consists of two lines. The first line would give you the gene structure of the first parent and the second line would give you the gene structure of the second parent. These structures are represented by strings constructed from the alphabet A to X. You can assume that the length of these strings does not exceed 30 characters.

The input must be read from standard input.

Output

For each of the test cases, you need to print one line of output. The output for each test case starts with the test case number, followed by the shortest length of the gene structure and the number of unique cyborgs that can be created from the parent cyborgs. You can assume that the number of new cyborgs will always be less than 2^{32}. Look at the sample output for the exact format.

The output must be written to standard output.

Sample Input	**Sample Output**
3 ABAAXGF AABXFGA ABA BXA AABBA BBABAA	Case #1: 10 9 Case #2: 4 1 Case #3: 8 10

E - LCM

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

All of you know about LCM (Least Common Multiple). For example LCM of 4 and 6 is 12. LCM can also be defined for more than 2 integers. LCM of 2, 3, 5 is 30. In the same way we can define LCM of first N integers. The LCM of first 6 numbers is 60.

As you will see LCM will increase rapidly with N. So we are not interested in the exact value of the LCM but we want to know the last nonzero digit of that. And you have to find that effeciently.

Input

Each line contains one nonzero positive integer which is not greater than 1000000. Last line will contain zero indicating the end of input. This line should not be processed. You will need to process maximum 1000 lines of input.

The input must be read from standard input.

Output

For each line of input, print in a line the last nonzero digit of LCM of first 1 to *N* integers.

The output must be written to standard output.

Sample Input	Sample Output
3 5 10 0	6 6 2

F - Floor Tiles

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

Jack is building a new house. He would like to tile his kitchen with congruent pieces formed by removing a 1×1 square from a 2×2 square.

He already knows that his kitchen will have length between $L1$ and $L2$ inclusive, and its width will be between $W1$ and $W2$ inclusive. Jack insists his kitchen to be a perfect rectangle. Of course, the floor must be tiled completely using these pieces (no overlaps, no spaces).

Determine the number of dimensions for Jack's Kitchen.

Input

You will be given K, the number of test cases. The next K lines will contain four positive integers separated by spaces: $L1 L2 W1 W2$, all of which are less than 1000.

The input must be read from standard input.

Output

For each test case you are to output a single line containing the number of different dimensions for Jacks Kitchen. A 2×1 kitchen is different from a 1×2 kitchen.

The output must be written to standard output.

Sample Input	Sample Output
2 2 2 2 3 2 3 3 4	1 2

G - Gaussian Primes

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

The complex numbers $a + bi$, where $i = \sqrt{-1}$ and a and b are integers, are called the Gaussian integers. The *norm* of a complex number is given by $\sqrt{a^2 + b^2}$. Every Gaussian integer can be factorized as a product of Gaussian primes. You task is to determine if a given Gaussian integer is a prime, i.e., if it can not be written as the product of two other Gaussian integers, x and y, where both x and y have norms larger than 1 (This is the same as requiring both x and y to be different from 1, -1, i and $-$i). For example, 2 = (1 + i)(1 − i), and, therefore, 2 is not a prime. 11 is a Gaussian prime, but 13 = (3 + 2i)(3 − 2i) is not a prime. In the same way, 3 + i = (1 + i)(2 − i) is not a prime.

Input

You are given a list of n Gaussian integers. The first number, n, in a row by itself, is the number of Gaussian integers that follow. This number is followed by n pairs of (possibly negative) integers, one per row, representing, respectively, the real and imaginary part of each Gaussian integer. The absolute value of the real and imaginary parts of every input number will be no larger than 10000, and the list will not have more than 100 numbers. Additionally, every pair (a, b) satisfies $a^2 + b^2 \geq 2$.

The input must be read from standard input.

Output

For each Gaussian prime in the input, you should write the letter 'P', for *Prime*. For each non-prime, you should write the letter 'C', for *Composite*.

The output must be written to standard output.

Sample Input	Sample Output
6 2 0 3 0 5 0 13 0 3 1 10 1	C P C C C P

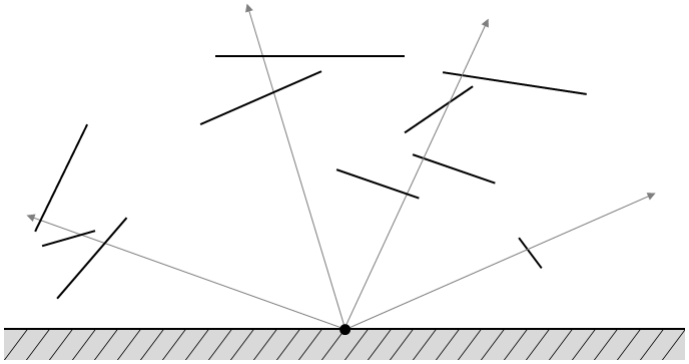
H - Handgun Shooting Sport

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

Crystal billboards present snipers with the opportunity of shooting sports in the abandoned neighborhoods of Crystal City. One popular shooting sport for a sniper is to destroy all crystal billboards in front of a building. The rules of this game enforce the sniper to fix his/her gun to the ground of the building's roof, so it can rotate freely in any direction. However, once the gun is fixed, it cannot be placed in any other location. The goal of the game is to destroy all crystal billboards firing the minimum number of shots.

Every time the sniper fires, any crystal billboard in the way of the bullet is destroyed, even if the bullet only touches one extreme of the billboard. A bullet never changes direction or speed once it is fired, even if it destroys any crystal billboards.

The following figure depicts a sniper in a fixed location facing a layout of ten crystal billboards. The sniper has destroyed all ten crystal billboards and has achieved the goal of the game because he/she destroyed all crystal billboards firing the minimum number of shots possible.



Given the initial location of a sniper and a layout of crystal billboards, your task is to determine the minimum number of shots that would destroy all crystal billboards.

Input

The first line of the input contains a natural number B defining the number of crystal billboards in the shooting layout ($1 \leq B \leq 10^3$). Each one of the following B lines contains four integer numbers x_1, y_1, x_2, y_2 separated by blanks, defining the coordinates of the line segment with extremes (x_1, y_1) and (x_2, y_2) ($-10^3 \leq x_1 \leq 10^3$, $0 < y_1 \leq 10^3$, $-10^3 \leq x_2 \leq 10^3$, $0 < y_2 \leq 10^3$, $y_1 \cdot x_2 \neq x_1 \cdot y_2$). You can assume that the shooting layout is modeled as the region on the Cartesian plane above the x-axis, that the sniper is located in the origin $(0, 0)$, that each crystal billboard is represented with a line segment whose extremes are not collinear with the origin, and that no pair of crystal billboards intersects. The last test case is followed by a line containing a zero.

The input must be read from standard input.

Output

For each test case, a line must be printed with the minimum number of shots that fired from the sniper location would destroy all crystal billboards in the shooting layout.

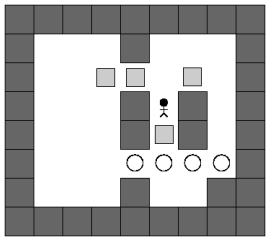
The output must be written to standard output.

Sample Input	Sample Output
10 -309 98 -258 204 -303 83 -251 98 -218 111 -287 31 -145 204 -23 257 -129 272 59 272 -8 159 74 130 150 146 68 174 59 196 128 242 98 256 241 235 197 61 173 92 3 -100 10 -100 50 -50 100 50 100 100 10 100 50 5 -100 100 100 100 -80 120 80 120 -60 140 60 140 -40 160 40 160 -20 180 20 180 2 -50 50 0 50 -10 70 50 70 2 -50 50 0 50 0 70 50 70 2 -50 50 0 50 10 70 50 70 0	4 3 1 1 1 2

I - Sokoban

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

Soko-ban is a Japanese word for a warehouse worker, and the name of a classic computer game created in the 1980s. It is a one-player game with the following premise. A single worker is in an enclosed warehouse with one or more boxes. The goal is to move those boxes to a set of target locations, with the number of target locations equalling the number of boxes. The player indicates a direction of motion for the worker using the arrow keys (up, down, left, right), according to the following rules.



- If the indicated direction of motion for the worker leads to an empty location (i.e., one that does not have a box or wall), the worker advances by one step in that direction.
- If the indicated direction of motion would cause the worker to move into a box, and the location on the other side of the box is empty, then both the worker and the box move one spot in that direction (i.e., the worker pushes the box).
- If the indicated direction of motion for a move would cause the worker to move into a wall, or to move into a box that has another box or a wall on its opposite side, then no motion takes place for that keystroke.

The goal is to simultaneously have all boxes on the target locations. In that case, the player is successful (and as a formality, all further keystrokes will be ignored).

The game has been studied by computer scientists (in fact, one graduate student wrote his entire Ph.D. dissertation about the analysis of sokoban). Unfortunately, it turns out that finding a solution is very difficult in general, as it is both NP-hard and PSPACE-complete. Therefore, your goal will be a simpler task: simulating the progress of a game based upon a player's sequence of keystrokes. For the sake of input and output, we describe the state of a game using the following symbols:

Symbol	Meaning
`.`	`empty space`
`#`	`wall`
`+`	`empty target location`
`b`	`box`
`B`	`box on target location`
`w`	`worker`
`W`	`worker on target location`

For example, the initial configuration diagrammed at the beginning of this problem appears as the first input case below.

Input

Each game begins with a line containing integers R and C, where $4 \leq R \leq 15$ represents the number of rows, and $4 \leq C \leq 15$ represents the number of columns. Next will be R lines representing the R rows from top to bottom, with each line having precisely C characters, from left-to-right. Finally, there is a line containing at most 50 characters describing the player's sequence of keystrokes, using the symbols `U`, `D`, `L`, and `R` respectively for up, down, left, and right. You must read that entire sequence from the input, even though a particular game might end successfully prior to the end of the sequence. The data set ends with the line 0 0.

We will guarantee that each game has precisely one worker, an equal number of boxes and locations, at least one initially misplaced box, and an outermost boundary consisting entirely of walls.

The input must be read from standard input.

Output

For each game, you should first output a line identifying the game number, beginning at 1, and either the word 'complete' or 'incomplete', designating whether or not the player successfully completed that game. Following that should be a representation of the final board configuration.

The output must be written to standard output.

Sample Input	**Sample Output**
8 9 ######### #...#...# #..bb.b.# #...#w#.# #...#b#.# #...++++# #...#..## ######### ULRURDDDUULLDDD 6 7 ####### #..#### #.+.+.# #.bb#w# ##....# ####### DLLUDLULUURDRDDLUDRR 0 0	Game 1: incomplete ######### #...#...# #..bb...# #...#.#.# #...#.#.# #...+W+B# #...#b.## ######### Game 2: complete ####### #..#### #.B.B.# #.w.#.# ##....# #######

J - Journey

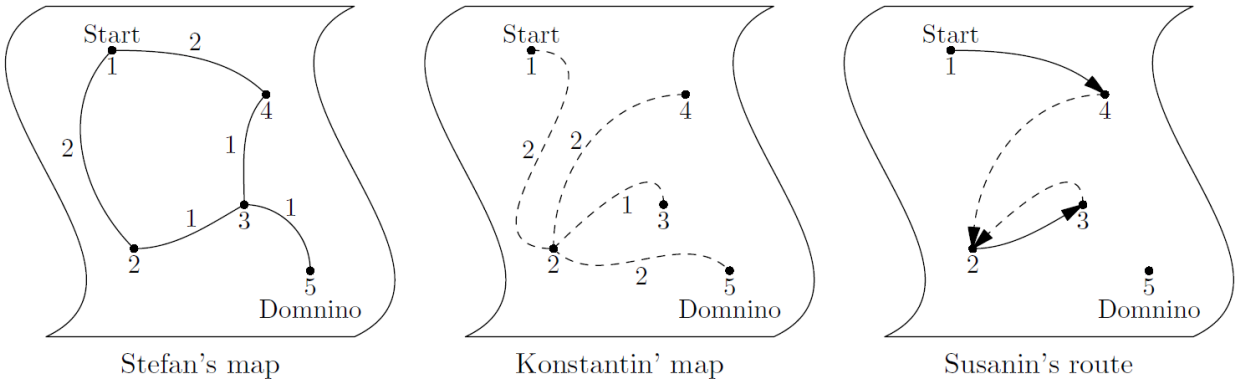
Source file name: `journey.c`, `journey.cpp`, `journey.java`, *or* `journey.py`

The army of Rzeczpospolita is moving from the city Kostroma to the village Domnino. Two hetmans, Stefan and Konstantin, lead the army.

Stefan procured the roadmap of Kostroma province and every night he routes the army from one village to the other along some road. Konstantin bought the map of secret trails between villages in advance and every day he leads the march along the one of such trails. Each hetman asks their guide Ivan Susanin for a route before each march.

The length of each road is indicated on Stefan's map. So Stefan knows the minimal distance from each village to the Domnino village according to his map. Similarly Konstantin knows the minimal distance from each village to Domnino village along trails on his map.

Ivan Susanin does not want to be disclosed as a secret agent, so each time he chooses a road (for Stefan) or a trail (for Konstantin) so that the minimal distance to the Domnino village according to the map owned by the asking hetman is strictly decreasing.



Help Ivan to find the longest possible route to the Domnino village.

Input

The input consists of several test cases. The first line of each test case contains three integer numbers n, s, and t — number of villages in Kostroma province, and numbers of start, and Domnino village ($2 \le n \le 1000$; $1 \le s, t \le n$). Villages are numbered from 1 to n. Start and Domnino villages are distinct.

Two blocks follow, the first one describing Stefan's map, and the second one describing Konstantin's map.

The first line of each block contains an integer number m — the number of roads/trails between villages ($n - 1 \le m \le 100\,000$). Each of the following m lines contains three integer numbers a, b, and l — describing the road/trail between villages a and b of length l ($1 \le a, b \le n$; $1 \le l \le 10^6$).

Rzeczpospolita army can move in any direction along a road or a trail. It's guaranteed that one can travel from any village to any other using each of the maps. The army starts its movement in the evening from the village number s and moves one road each night and one trail each day.

The end of the input is given by $n = s = t = 0$.

The input must be read from standard input.

Output

For each test case, output the total length of the longest route that Ivan Susanin can arrange for Rzeczpospolita army before reaching the Domnino village (along the roads and trails). If Ivan Susanin can route the army forever without reaching the Domnino village, output the number `-1`.

The output must be written to standard output.

Sample Input	**Sample Output**

`5 1 5`
`5`
`1 2 2`
`1 4 2`
`2 3 1`
`3 4 1`
`5 3 1`
`4`
`1 2 2`
`2 4 2`
`2 3 1`
`2 5 2`
`3 1 3`
`4`
`1 2 10`
`2 3 10`
`1 3 20`
`2 3 30`
`4`
`2 1 10`
`1 3 10`
`1 1 10`
`2 3 10`
`0 0 0`
	`-1`
	`20`