CHALLENGE24
INTERNATIONAL 24-HOUR PROGRAMMING CONTEST

Electronic Contest

14th International 24-hour Programming Contest

http://ch24.org

MAIN SPONSOR

SAP

DIAMOND GRADE SPONSORS

FORNAX

Google

Microsoft

ORGANIZER

mave

Magyar Villamosmérnök- és Informatikus-hallgatók Egyesülete

eestec

LC Budapest

PROFESSIONAL PARTNERS

hte

HÍRKÖZLÉSI ÉS INFORMATIKAI TUDOMÁNYOS EGYESÜLET

NJSZT

Electronic Contest

Welcome to the qualifying round of the 14th International 24-hour Programming Contest!

This document is the problem set for the Electronic Contest to be held on February 8th, 2014.

Rules

The Electronic Contest contains multiple problems. You have all the time in the world to solve them, but we take submissions from 10:00 to 16:00 CET. The inputs of the problems can be found in a zip file that you have probably already downloaded from the website.

You can use any platform or programming language to solve the problems. We are interested only in the output files, you don't need to upload the source code of the programs that solved them. Once you are done, you can upload your output files via the submission site: http://sub.ch24.org/sub/. Your solutions will be evaluated on-line.

Problems are scored in three major ways:

- **Time** scoring: these problems have exact solutions. When submissions to these are evaluated, a final score is given immediately. From one team, only one correct submission will be accepted for each input (since the input is either solved or not). Given score decreases with time until the end of the contest, so faster solutions get more points.
- **Competitive** scoring: problems that do not have a known "best" solution. Outputs for these problems compete against each other, and scores are scaled according to the best uploaded output. A team may submit multiple correct submissions to one input (only the latest submission will be taken into consideration).
- **Proportional** scoring: solutions to these problems will be compared against a chosen standard. The final score is calculated from the ratio of the evaluated score of the output to a selected constant. A team may submit multiple correct submissions to one input (only the latest submission will be taken into consideration).

Note that points are awarded per output file and not per problem. If your solution only works for some of the input files, you will still be awarded points for the correct output files. A single output file however is either correct or wrong - partially correct output files are not worth any points.

Additional information for time scoring:

Be quick about uploading the output files, because the scores awarded for every output file decrease with time. Uploading it just before the end of the contest is worth **70%** of the maximum points achievable for the test case. During the contest its value decreases linearly with time. However you should also be careful with uploading solutions. Uploading an incorrect solution is worth negative points. This penalty is additive, if you upload more incorrect solutions, you will receive it multiple times. For some problems, we distinguish format errors (unparsable outputs) from incorrect outputs, and the former will not be penalised.

Please note that for time-scored problems there is no point in uploading another solution for an already solved testcase because you cannot achieve more points with it. Therefore the system will not register additional uploads for solved testcases for those tasks.

For some time-scored problems, after submitting an incorrect solution, there may be a certain short delay (a couple of minutes) until you can re-submit an updated solution. The delay is applied per team per task per input, and is reported on the submission web interface.

Additional information for competitive scoring:

In this case there will be no score penalty for uploading a solution later, so you are able to achieve the maximum amount of points by submitting in the very last minute - if you beat the other teams' solutions, that is. However, to avoid overloading our server, after submitting a correct solution, you may not re-submit an updated solution for a certain short delay (a couple of minutes). The delay is applied per team per task per input, and is reported on the submission web interface.

Scores for competitively scored problems are recalculated occasionally (every few minutes). Your points may decrease in time (when another team submits a better solution than yours).

Please be aware that only your last submission is considered - not your best one.

Good luck and have fun!

About the Submission site

The location of the submission site is:

http://sub.ch24.org/sub/

You will be able to log in to the submission site with your registered team name and password. After login you can access three main views:

Team Status

You can see your team's status here, with all your submissions and the points received for them.

Submit

This is where you can post your solution files. You can upload multiple output files for multiple problems with a single submit. The naming of the output files must strictly match the following format: X99.out - where X is the problem's character code followed by a number (1 or 2 digits) identifying the test case.

Scores

Here you can see the current standings of the contest. This will not be available in the last hour.

Contact

You can contact us at the email address info14@ch24.org with questions.

During the contest we will be available on IRC on the `irc.ch24.org` server (using the default port, `6667`), on the following channels:

- `#challenge24` for general discussion about the contest,
- `#info` for a full summary of announcements (read-only),
- dedicated channels `#a`, `#b`, `#c`, `#d`, `#e`, `#f`, `#g`, `#h`, `#i`, `#j` for problem specific questions.

Note: **all relevant questions/answers will be copied to #info**, which will be also available on the submission site.

Prologue: Managing World Security

Not all citizens are lawful these days - there are criminals, terrorists, axes of evil, and many Very Bad People in general. Fortunately large, international organizations such as WSA (World Security Agency) take care of your security. To do so, WSA needs to employ sophisticated means of espionage, including a few methods that are related to information technology.

This Electronic Contest is part of a test program that aims to use community resources in aid of the WSA Anti-Evil Spying Operations.

Disclaimer: the WSA does not interfere with the best interest of lawful citizens. All operations are targeted on Evil People. If you're Good, you don't have anything to be afraid of!

Task Summary

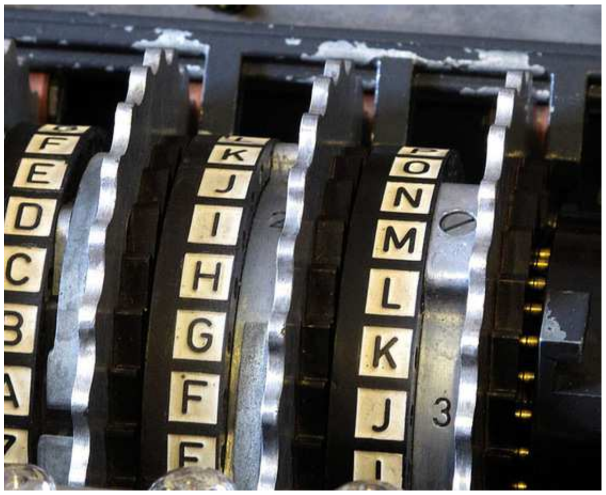
Task	Score	Scoring type	Wrong answer penalty	Delay	Input format
A. Safe	1000	time	-5 points	0	text
B. Wiretapping	1000	time	-5 points	0	text
C. Visual Programming VM	500	time	-2 points	60s	text
D. Visual Programming	2000	time	-5 points	60s	text
E. Travelling	1000	time	-5 points	0	text
F. Forensics	1000	time	-5 points	0	wav
G. Spy Union	1000	time	-5 points	0	text
H. Tile Design	1000	proportional	-5 points	0	text
I. Crowd Control	1000	time	-5 points	0	text
J. Image compression	1000	competitive	0 points	60s	png

- Wrong answer penalty: Penalty after each wrong output submitted.
- Delay: Time duration until no solution can be submitted for the same input.
- Score: most problems have 10 inputs and each one is worth 100 points at most.
- Problem C has only 5 inputs.
- Problem D has 10 inputs, but each one is worth 200 points.

For each task there is a dedicated irc channel for questions: `#a`, `#b`, `#c`, `#d`, `#e`, `#f`, `#g`, `#h`, `#i`, `#j`.

A. Safe

The WSA prints every important document and stores the printouts in large, secure safes. As per policy, safes always must be closed after use, forcing employees to open each safe hundreds of times per day. A key point of optimizing processes at the WSA is to speed up entering passwords for safes.


source: http://commons.wikimedia.org/wiki/File:Enigma_rotors_with_alphabet_rings.jpg

The keyboard keys on the safe are not ordinary keys, every time they are pressed they don't just enter the letter displayed on them as input, but the letter displayed on them changes as well. For each key, there is a cycle of letters that can appear

The cycle of each key (actual letters the key would input push-by-push, in order) is known as well as the password that needs to be entered. The code is accepted only if a password of length N is entered using the last N key presses (that is, there's no way to reset cycles in between). All keys start on the first letter of their cycle (and the cycle can loop around).

The task is to find the least amount of key presses needed to enter the code.

For each keyboard there are several different passwords that the task should be solved for.

Input

The first line contains one number, *N*, the number of keys. The next *N* lines contain the cycle for each key, consisting only of capital English letters. The following line contains *M*, the number of passwords that need to be typed on this keyboard. The next *M* lines contain the *M* passwords.

Output

A single number on a separate line for each input code, in order. If it is possible to enter the code, output the number of key presses necessary, otherwise output "-1".

Example input

```
3
ABC
TEST
CAT
4
TEST
CC
STT
TT
```

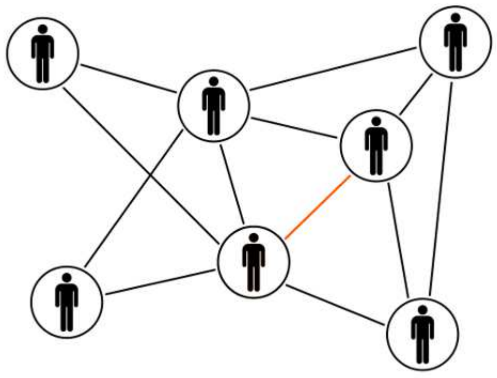
Example output

```
4
4
5
4
```

B. Wiretapping

The WSA used to tap a lot of phone lines - it's a great way of catching evil plots that the terrorist cells carelessly discuss between themselves using the public phone system. Lately, the WSA had to reduce costs by cutting back on traditional wiretapping, and they are trying to determine which few phone lines they will concentrate their remaining resources on.

There is a large list of terrorist cell relationships, otherwise known as phone lines (collected by other means).


http://openclipart.org/detail/99967/black-orange-men-cloud-by-pydubreucq

When the Head of Bad People decides to do something Evil, he hatches the Master Plan; the plan is then broadcast to all cells, using the public phone network.

The WSA assumes that terrorists behave rationally, thus:

- they utilize the smallest possible amount of connections (minimizing number of phone calls)
- they always distribute the plan to every cell.

A talented young agent has chosen a phone line the tap is installed on. Your task is to determine the probability of catching the important broadcast, whichever connection setup the terrorist chose this time.

Input

The first line contains two integers *N* and *E*, the number of cells and potential phone connections, respectively. The next *E* lines are the edges given as a pair of integers *A* and *B*, each between 0 and *N*-1, representing a potential phone call between cells *A* and *B*.

The wiretap is placed on the first connection on the list.

Output

A single number to 1e-6 precision describing the probability of catching the message on the tapped line.

Example input

```
6 12
0 1
1 2
2 3
3 4
4 5
3 0
0 2
1 3
2 4
3 5
4 0
5 1
```

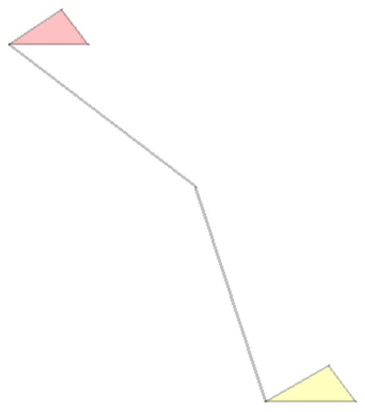
Example output

```
0.4194444444
```

C. Visual Programming - VM

The WSA employs many programmers, who perform a bit better or a bit worse each day. This results in higher or lower quality code accordingly. The management understands the importance of high quality code (especially in such a security-sensitive domain!), therefore they decided to optimize the processes of the WSA software department.

The first step of the effort was to send a few managers to understand the nature of the problems. The programmers had shown thousands of lines of source code, written in various languages the managers had never seen before. They had a hard time trying to understand all this, and after weeks of evaluation, they figured out why: most programming languages are too complicated! They concluded that they must design a new programming language that is much simpler, and mandate its use within the organization.


Almost accurate visualisation of the example program (0.pp) for this task

They came up with something more visual (thus user friendly) and vastly simpler (it has only two different constructions and only one kind of instruction). Programmers were initially sceptical of the practicability of these ideas, so WSA management has hired you to prove the merits of the language by implementing certain example algorithms.

This problem is so large it's split in two separate problems: first you should write and test your toolchain for dealing with polyprog programs, then (using your brand new tools) you can attempt to create the example programs.

programming polyprog

The polyprog machine has 16 registers, of which the first 15 (from 0 to 14) are general purpose, while register 15 is the serial input/output (SIO) device. Registers store 16 bit unsigned integers.

Polyprog programs are represented by a list of polygons and lines. Color is specified in 32 bit RGBA (each component is an integer between 0 and 255). Coordinates are unsigned 16 bit integer x;y pairs. A polygon has a fill color, a stroke color, and at least 3 vertices specified by their coordinates. A line is a color and an ordered list of points (coordinates); the *end points* are the first and last points on the list.

To keep source code clean and readable:

- polygons shall not overlap or touch other polygons
- lines may not overlap or touch polygons except that both endpoints of the lines must be on polygon vertices
- lines may intersect other lines
- a polygon vertex may be the same position with zero or one *line end point*
- line mid points may overlap each other
- the number of points in a line is limited to *LLEN*, including the two endpoints
- dimensions of the program (maximum x and y coordinates) are limited to *DIM* units; (0 <= x < DIM)

and (0 <= y < DIM).

All polygons have at least one line connected to them and a valid program contains at least one polygon.

Limits: *DIM*=65536 and *LLEN*=16.

the polyprog processor

The processor has an instruction pointer which points to one of the polygons. A polygon instruction is executed in multiple stages:

Stage 1: register operation

- register *A* is addressed by the high order 4 bits of fill:red
- register *B* is addressed by the low order 4 bits of fill:red
- register *C* is addressed by the high order 4 bits of fill:green
- register *D* is addressed by the low order 4 bits of fill:green
- register *E* is addressed by the high order 4 bits of fill:blue
- register *F* is addressed by the low order 4 bits of fill:blue
- constant *X* is fill:alpha
- constant *Y* is stroke:red
- constant *Z* is stroke:green*256 + stroke:blue
- constant *T* is stroke:alpha

perform

```
A = (Y + B + C * D + Register[F]) * (255-X) / ( (255-T) + E ) + Z
```

and update *A* accordingly. Register values and constants are converted to 32 bit unsigned integers for the calculation; at the end, the result is truncated back to a 16 bit unsigned integer. (Thus, operations are evaluated modulo 2^{32}, division is integer division with non-negative operands, a zero second operand is fatal error and the final result modulo 2^{16} is stored in *A*.)

Register[F] means the lower 4 bits of the value in *F* selects which register’s value is substituted (the higher 4 bits are ignored).

Stage 2: route selection

Take every vertex of the polygon that has a connected line, sort them by *y*DIM+x* in ascending order, and walk this list checking for the following condition:

- register *Q* is addressed by the low order 4 bits of the x coordinate
- constant *L* is line:red*256 + line:green
- constant *H* is line:blue*256 + line:alpha
- if *(Q >= L)* and *(Q < H)*, the condition is statisfied

Stage 3: jump

Pick the vertex for which the condition was first satisfied in Stage 2; or if there's no such vertex, pick the last vertex in the above order. The instruction pointer is modified to point to the polygon on the other end of the line connected to the picked vertex.

reset

After a reset all general purpose registers are initialized to contain their own addresses (so that the value in register N is N). The instruction pointer is pointing to the polygon that has the vertex with *min(y*DIM+x)*. The program's input stream is set up so that the first read will get the first byte in the input.

serial input and output (SIO)

When register 15 (SIO) is read, it returns the next byte in the input stream until EOF; once EOF is reached, all subsequent reads will result in EOF. When SIO is written, it appends the byte to the output stream. If an instruction (polygon) attempts to read SIO multiple times, only the first attempt will result in reading a byte from the input stream, subsequent reads in the same instruction will reuse the same (cached) value without removing more bytes from the input. It is possible to read and write SIO in the same instruction, in which case the write happens later. The program stops running immediately when EOF is written to the serial output.

SIO data is an unsigned 8 bit integer placed in the least significant 8 bits of the register, leaving the upper 8 bits 0:

```
MSB                                         LSB
15                                            0
0  0  0  0  0  0  0  0  d  d  d  d  d  d  d  d
```

Input EOF is a word with value 256:

```
MSB                                         LSB
15                                            0
0  0  0  0  0  0  0  1  0  0  0  0  0  0  0  0
```

Output EOF is a word with value >255: at least one of the z bits is 1. (In this case, x bits are ignored.)

```
MSB                                         LSB
15                                            0
z  z  z  z  z  z  z  z  x  x  x  x  x  x  x  x
```

program submission syntax

The first line of the program contains two integers *P* and *L* separated by a space; *P* is the number of polygons, *L* is the number of lines. The next *P* lines describe polygons, then the next *L* lines describe lines.

A polygon description starts with 4 integers *SR*, *SG*, *SB*, *SA* for the stroke color components, then 4 integers *FR*, *FG*, *FB*, *SB* for the fill color components and an integer *C*, number of vertices. The rest of the line is a space separated list of *X* space *Y* coordinates of the vertices in CW or CCW order.

A line entry starts with 4 integers *R*, *B*, *G and A* for the color of the line, followed by integer *P* (the number of points) and list of *X* space *Y* coordinates giving the points of the line. *P* is limited to: 2 <= *P* <= *LLEN*.

Tasks

This task is provided as an aid for developing your own toolchain, especially an interpreter, to develop, run and debug polyprogs. **Please make sure you solve all inputs of this task before trying to solve the next problem.**

Any new implementation of polyprog compilers and interpreters must comply with the standard described above. Verification is possible using the reference programs the Managers have provided. An implementor should run all five reference programs and compare the output to the reference outputs. Unfortunately the reference output of these programs is the Intellectual Property of the WSA, but the managers are willing to compare your output to the references and tell if they match.

Input

You need to run five polyprog programs. A reference serial input, serial.in.bin is also provided; before running a reference program, the SIO's read pointer should be reset to the beginning of this file. **Note: serial.in.bin is a binary file.**

Output

The output as written by the program on the SIO. **Output files are binary, but never contain the \0 (nil) character.** If your interpreter outputs \0 executing any of the reference programs, it is broken.

Example input

```
2 1
0 0 0 254 47 0 0 254 3 100 100 110 110 80 110
0 0 0 254 255 18 0 254 3 200 200 210 210 176 210
0 0 0 1 3 80 110 150 150 176 210
```

(*"The Input stream contains pairs of 7-bit unsigned integers in binary format; sum each pair and print the result to the output as 8-bit unsigned int in binary format. The program shall stop when EOF is read on the input stream. If the input stream contains an odd number of bytes, ignore the last byte."*)

Example output

Running the reference solution on serial.in.bin should result in 3 bytes on the output stream - here represented in a hexadecimal format:

```
D3 D1 D5
```

D. Visual Programming

For each problem input you should write programs in the polygon programming language that read the input through the SIO register, evaluate it, and write suitable output through SIO. Each *problem input* is a different programming task. You should upload a polygon program for each. The *problem input* tasks are semi-independent: in case you can write a generic solution that solves more than one task, you may submit the same program to multiple inputs.

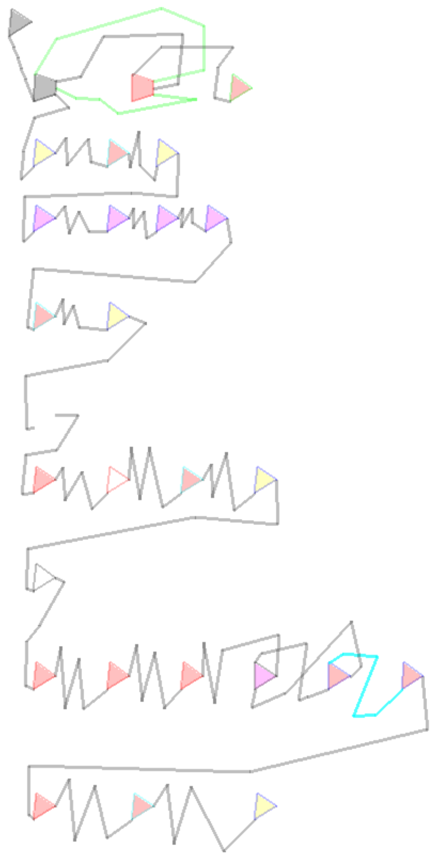
The evaluation server will run your program multiple times with different example *input streams* and determine the raw score as a sum of the following:

- the program outputs the right solution for every example *input stream*: 200 points;
- or the program outputs a wrong solution or crashes or violates the limits: stop testing, report failure, penalty.

Note: in the input and output format description, an EOF at the end of the stream is always assumed; this EOF is not a real character and thus does not count when calculating the length of the stream.

Submissions are evaluated on the reference implementation of the polyprog environment; to avoid contributing to global warming by excess simulation of infinite loops programs are stopped after executing 200000 polygons, without writing an EOF. Such greedy porgrams are obviously not accepted as they fail to terminate their output stream with an EOF.

It is strongly recommended to solve all inputs of the previous task before solving this one!


Reference program four (4.pp) from the previous task

Task summary		
problem id	**serial input syntax**	**short description**
1	list of uint8 and +	calculate sum, no overflow
2	list of uint8 and + and -	calculate sum, no overflow
3	list of uint8 and +, -, * and /	evaluate the expression, no precedence
4	decimal number in ASCII text	decimal to binary converter
5	uint16 binary number	binary to decimal converter
6	plain text input: numbers and operators (+, -, *, /)	evaluate the expression (as in 3), no precedence
7	same as problem 6, plus the ^ operator for generic power	
8	same as problem 7, plus the _ operator for generic root	
9	list of uint64 in binary and + operator	same as problem 1
10	list of uint64 in binary and + and - operators	same as problem 1

Detailed specification		
description	**input**	**output**
Problem 1: Input stream is a list of number-operator pairs. Numbers are 1 byte long unsigned integers, the operator is always a plus sign (’+’). The program shall evaluate the expression on the input and calculate the result and print it on the output as an unsigned 16 bit integer. When printing the output, the most significant byte shall be printed first. The last operator is omitted, the input ends with a number. A valid input stream contains an odd number of bytes, every second byte is a ’+’ operator. If the operator byte is ’-’, ’*’ or ’/’ then the behaviour is undefined (won’t be tested), otherwise if the input is not valid, the program shall print the text "ERROR" and a newline (in 6 bytes) and exit. *The ERROR message* in hex is: `45 52 52 4F 52 0A <EOF>`. *Problem inputs* are designed to yield a sum less than 65536, therefore this example program doesn’t need to deal with overflows.	(hex) `02 2B 07`	(hex) `00 09`

description	input	output

Problem 2: Same as problem 1, except the operator may be a plus or a minus sign ('-'); minus means subtraction. Assuming operations are performed in order of apperance, the *problem inputs* will guarantee that the result of the expression up to any operator will be between -32768 and +32767. Output must be written in two's complement format.	(hex)
	
	`02 2B`
	`07 2D`
	`03`
		(hex)
		
		`00 06`

Problem 3: Same as problem 2, except the operator may be a plus, minus, multiply ('*') and divide ('/'). There is no operator precedence in this task, **operations are executed in order they appear**, so 2+3*4 is not evaluated as 2+(3*4) but as (2+3)*4. Division by zero shall result in *the ERROR message*. Divisions are integer divisions. The dividend is never negative. The *problem inputs* do not contain calculations with any intermediate result out of the -32768 and +32767 range (the program doesn't have to care about overflow conditions).	(hex)
	
	`02 2B`
	`07 2D`
	`03 2A`
	`04 2F`
	`03`
		(hex)
		
		`00 08`

Problem 4: Input stream is a number written as ASCII text in decimal format. The program shall read the input, convert it to an unsigned 8 bit integer and print it in binary format to the serial output.
 	
If the input contains anything else than digits, the program shall quit without printing any character.
 	
Input is always between 0 and 255, range check is not required. Leading zeros shall be accepted.
	(hex)
	
	`31 32`
	`33`
		(hex)
		
		`7B`

Problem 5: Input stream is a list of 16 bit unsigned integers, most significant byte first, thus consisting of an even number of bytes altogether. After the second byte of an item is received it should immediately print an ASCII decimal representation of that number (with leading zeros filled up to a width of 5 characters) and a newline. If the input terminates after odd number of bytes, the program shall print *the ERROR message*.	(hex)
	
	`10 70`
		(hex)
		
		`30 32`
		`36 33`
		`30`

Problem 6: Same as problem 3, but input is ASCII text: each line is a decimal number or an operator. Lines are terminated with a single newline character (ASCII 10, "\n"). The first and the last line of the stream must be numbers. Numbers are unsigned integers between 0 and 32767 and may contain leading zeros. The output format is the same as in problem 5. The program is not required to do anything special for overflows.
 	
If a number-line of the input contains anything other character than digits, the operator-line is not a '+', '-', '/', '*', '^' or '_' or an input line is empty, the program shall print *the ERROR message*. If the operator is '^' or '_' the behaviour is undefined (won't be tested).
	(text)
	
	`12`
	`+`
	`55`
	`/`
	`2`
	`*`
	`3`
		(text)
		
		`00099`

description	input	output

Problem 7: Same as problem 6, with an extra operator "^", which means power: ^*b* means *the result of the previous part* to the power of *b*. The *problem inputs* will not use the power feature to generate numbers larger than 32767. Corner cases:
• 0^0: the result is 1
• 0^1: the result is 0
	(text)
	`02`
	`*`
	`3`
	`^`
	`4`
		(text)
		`01296`

Problem 8: Same as problem 7, with a generic root operator '_': _*b* means *the result of the previous part* to the root of *b*. For example 8_3 is 2. Since this is only a demonstration program, the managers are tolerant with the precision of the root operator: the result is accepted if it's +-2 of the root calculated with infinite precision truncated to integer; the inifinite precision is meant for the root operation only, not for any previous calculations. For example 122_3 is approximately 4.95968 thus the range of acceptance is (4-2)..(4+2), so anything between 2 and 6 (inclusive) is accepted (exception: "_1" must be precise). Corner cases:
• *a_1*: must be exactly *a*, the +-2 tolerance is not applied here
• *a_0*: *the ERROR message*
	(text)
	`2`
	`*`
	`3`
	`+`
	`100`
	`*`
	`19`
	`_`
	`3`
		(text)
		`00012`
		(but 00010, 00011, 00013 and 00014 are also accepted)

Problem 9: Same as problem 1, but for 64 bit unsigned integers. Both input and output numbers are 8 byte long, with the **least significant byte written first** (unlike in the previous problems!).
 	
The addition shall be done modulo 2^{64}.
 	
If the operator byte is '-' the behaviour is undefined (won't be tested), otherwise if the operator is not '+' or the end of the input stream is reached unexpectedly, print *the ERROR message*.
	(hex)
	`86 47 F0 75`
	`12 60 3F 13`
	`2B`
	`7D 0D E5 60`
	`3F 83 C0 2F`
		(hex)
		`03 55 D5 D6`
		`51 E3 FF 42`

Problem 10: Same as problem 9, with an extra operator '-' for subtraction modulo 2^{64}. Input will not contain numbers larger than 2^{63}-1.	(hex)
	`03 00 00 00`
	`00 00 00 00`
	`2B`
	`02 00 00 00`
	`00 00 00 00`
	`2D`
	`09 00 00 00`
	`00 00 00 00`
		(hex)
		`FC FF FF FF`
		`FF FF FF FF`

E. Travelling

Some WSA agents travel frequently on official business, and they figured out how to cheat the WSA's working time accounting system. The trick is that if they have to visit a lot of distant cities, they travel more to the east than to the west, and they use local time zones for logging their time. As long as they don't travel around the globe twice, and they choose a reasonably short path, no one would notice.

source:http://openclipart.org/detail/170988/travel-globe-by-gnokii-170988

At least, so they thought. WSA management eventually discovered the cheating (...no they didn't; rather, someone snitched), and they want to estimate the losses. They handed you 10 lists of cities that agents usually visit (with coordinates).

Find the shortest tour visiting all the listed locations without ever traveling West (longitude shall never decrease, except when crossing the antimeridian) and without traveling around the planet more than twice.

Input

The first line contains N, the number of locations and the following N lines are the (latitude, longitude) coordinates of the locations in degrees. (-90 ≤ latitude ≤ 90 increasing to the North and -180 ≤ longitude ≤ 180 increasing to the East and there are no locations exactly at the same longitude or at the poles)

Output

The first line should contain the length of the optimal tour in km (to one meter precision; Earth is approximated as a perfect sphere with R = 6371 km radius).

The second line should contain N+1 numbers separated by whitespace: the indices of the locations in the order they are visited in the optimal tour (indexing starts from 0, based on the order in the input, the last location must be the same as the first one and the longitude of consecutive locations must not decrease more than twice).

Example input

```
6
40 0.5
-60 50
-10 -80
80 1e2
30 -160
-70 140
```

Example output

```
53904.473044
4 1 5 2 0 3 4
```

F. Forensics

During the last WSA conference, the second most important event after the annual "state of the War On Bad People" presentation was that the director of WSA had been assassinated. Because of the crowd and the panic after the incident, the perpetrator could escape. The WSA hasn't been able to determine the identity of the attacker yet, even with the footage from the numerous installed security cameras.

source: http://commons.wikimedia.org/wiki/File:Nano_Hummingbird.jpg

A small autonomous flying device was used in the attack (a "hunter-seeker"). The device was homemade and built entirely from plastic (probably to evade metal detectors). Possibly because of design contraints imposed by this, the device could only fly in a straight line with constant speed, and emitted a particular, audible humming.

Since there was a microphone installed between the director and the crowd, there is a high quality audio recording of the whole event, including the humming (which was identified as a periodic, fixed frequency signal).

The WSA has a theory that careful evaluation of this record could reveal the exact position of the assassin. The recording is classified Top Secret for the next 80 years; however, the WSA is willing to share an edited version with capable forensics who can first prove their skills on 10 simulated inputs. The inputs are carefully filtered similarly to how the real recording will be (e.g. to remove potentially disturbing sounds that are not directly useful to the task at hand, and to equalize the volume of the humming effect for easier processing).

The room is large and flat, the hunter-seeker flies in a straight horizontal line at an unknown constant speed which is less than the speed of sound. The speed of sound is c=343.2 m/s. The radius of Earth, the gravity, air properties (humidity, friction, etc) can be discarded in such a small room.

Input

A 8 bit wav sampled at 44100 Hz rate. The recording starts when the hunter-seeker appeared.

Output

The distance of the attacker from the microphone in meters to 1 meter precision when he released the hunter-seeker.

Example input

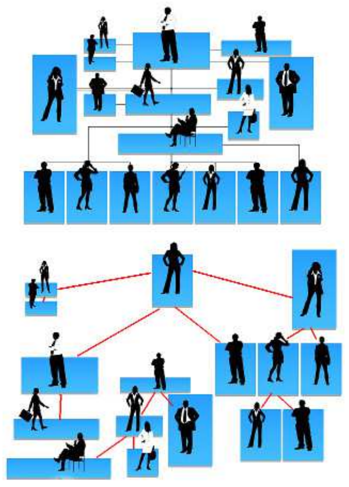
Please refer to 0.wav.

Example output

```
404.475
```

G. Spy Union

The spy network of the WSA has grown too big, downsizing is unavoidable. Unfortunately the Trade Union of Spies is large and strong, and they have spies everywhere. After many months of fruitless fighting, the management of the WSA met the representatives of the union, and together they came up with two directed trees.


source:
http://pixabay.com/en/silhouettes-hierarchy-human-man-81830/

The first tree represents the hierarchy of the WSA. Each employee is a node, and each node is the head of a department (the node together with all its descendants). Each node is tagged with an integer that specifies how many employees have to be in that department to keep the organization operational. The second tree is the hierarchy of the union - a different hierarchy with different integers, but the meaning of the tree is the same. All employees are present in both trees.

Your task is to determine which employees to dismiss to get the smallest possible organization while still keeping all WSA and union departments operational.

Note that department heads may be fired - in which case a subordinate will be the new head; but some departments may have a tag of 0, meaning that the department is superfluous and may be eliminated altogether.

Input

The first line is an integer *N*, the number of employees. Their Employee ID (EID) is an integer between 0 and *N*-1. The next *N* lines refer to employees, the *n*th line to the employee with EID *n*, in *Bw Bu Rw Ru* format. *Bx* is the EID of this employee's boss in the *w* (WSA) or *u* (union) hierarchies (0 <= Bx < *N*, the boss EID of the top manager is their own number). *Rx* is the number of required people in this employee's respective subtree (0 <= Rx <= N).

Output

The first line is *K*, the maximum number of employees who can be fired. The second line contains *K* integers, the EIDs of the dismissed employees.

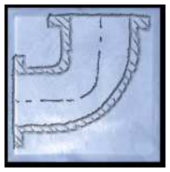
Example input

```
5
1 0 1 2
2 0 1 2
2 1 2 0
2 1 0 1
1 3 0 0
```

Example output

```
2
4 2
```

H. Tile design



source: http://www.photos-public-domain.com/2011/12/30/blue-bathroom-tile-with-swirl-pattern-texture/

In the brand new HQ of the WSA, construction subcontractors are working on the last tweaks, like tiling the personal hygiene facilities (bathrooms). The overall design concept dictates that the wall patterns must match the purpose of each room, so the selected bathroom tile design features pipe sections.

Each tile has exactly one pipe printed on it:

- a straight pipe section;
- or a pipe corner.

Tiles can be rotated but not mirrored.

When designing tiling layouts, there are some psychological constraints as well. Imagine what guests would think seeing a decorative pipe system that just ends abruptly - it may make them imagine that the WSA organization is not a single, seamless, organically interconnected system. To avoid such misunderstandings, the pipeline must not have ends; which is possible by designing a large single loop that covers the wall.

Furthermore, users well-versed in algorithmic security may object to seeing long repeated patterns in such loops, because those would suggest that entropy in the WSA is low. How could they trust the WSA's cryptography and the randomness of its keys if the WSA can't even build a random enough tile layout?

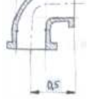
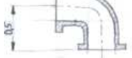
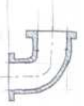
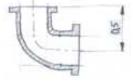
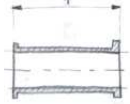
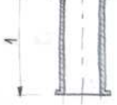
Your task is to help the designer to tile the wall so that all available space is covered by tiles, the pipes on the tiles form a single loop, and the longest repeating pipe sequence of the design is as short as possible.

Input

Three integers, *W*, *H* and *L*, where *W* and *H* are the width and height (measured in number of tiles) of the wall and *L* is the length of the longest repeating pipe line we want to achieve (see below).

Output

The first line is two integers *W*, *H* (matching width and height in the input). The follwing *H* lines form a large "ASCII art" representation of the tile figures, one character per tile, each line containing *W* characters. The meaning of the characters:

character	F	7	J	L	-	I
figure						

So each tile figure is a pipe section with some orientation. The pipe sections have a flow direction as well that is not represented in the plan, but will be visible on the final design, where tiles will be placed so that the flow can go around the wall in positive direction.

A pipe line is a sequence of neighboring tiles with connected pipe sections following the direction of the flow. If the pipe line connects back to itself then it's a loop. All the tiles should form a single loop, but the longest repeating pipe line (*LRPL*) must be as short as possible (ideally *L* or less). A rotated version of the pipe line is considered to be a repetition, but a mirrored one is not. Repetitions can overlap.

Finding a solution with longer than *L* repetitions will earn less than 100 points by the following formula:

Scoring

if `LRPL ≤ L` then `SCORE = 100`
else if `LRPL ≥ L+25` then `SCORE = 0`
else `SCORE = round(0.16*(L+25-LRPL)`^{2}`)`

Example input

```
8 8 5
```

Example output

`8 8`
`F--7``F--7`
`I``F7IL7FJ`
`I``IIL-JL7`
`IIL-7F-J`
`IIF7IIF7`
`LJILJII`**`I`**
`F-JF7LJ`**`I`**
`L--J`**`L--J`**

The longest repeating pipe line in the example is 6 (hilighted), so its score is 92.

Note that the hilighted part below is not a rotated and overlapping 8 length repetition, because pipes are directed.

`8 8`
`F--7F--7`
`I`**`F7`**`IL7FJ`
`I`**`II`**`L-JL7`
`I`**`I`**`L-7F-J`
`II``F7IIF7`
`LJ``ILJIII`
`F-JF7LJI`
`L--JL--J`

I. Crowd control

Because of crowd management difficulties experienced during recent unfortunate events regarding the WSA leadership, the WSA decided to investigate the flow of personnel in public buildings. A stampede of people constrained in passages partially blocked by columns, plants, janitors, and other debris caused unwelcome media attention and made it difficult to control "demonstrators" and other undesirables.

source: http://vector.me/browse/182519/emergency_exit_sign_clip_art

Based on advice from their private hands-on security experts and other consultants, the WSA decided that in the future it wishes to arrange conferences in buildings that can pass people through at a given rate during escape scenarios.

Your task is to find the maximum possible flow rate of people for each given layout. (Keep in mind that the carrying capacity of a corridor is proportional to its cross section.)

Input

The layout is represented as a polygon, two sides of which are the entrance and the exit. All the obstacles are also given as polygons. Other than the neighbouring sides of polygons, the lines never touch.

First line contains 3 numbers, first *o*, the number of obstacles, then *b* end *e*, the beginning and end sides of the outer polygon. The side of the outer polygon between the *b*th and (*b*+1)th vertices is the beginning, and the side of the outer polygon between the *e*th and (*e*+1)th vertices is the end. $0 < b, e < n$ of outer polygon.

Next is the outer polygon, the first line of which is *n*, the number of sides, then *n* lines with the x and y coordinates of the *n* vertices, starting with the 1st vertex. The coordinates are all floating point numbers.

Next come the *o* obstacles in the same format.

Output

The width of a simple corridor with the equivalent flow rate to this layout (as a floating point number).

We accept an output if it is within 0.0001% of the official solution.

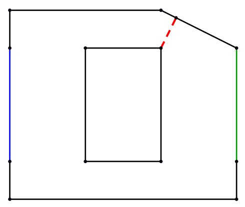
Example input

```
1 2 6
8
0 0
0 1
0 4
0 5
4 5
6 4
6 1
6 0
4
2 1
2 4
4 4
4 1
```

Example output

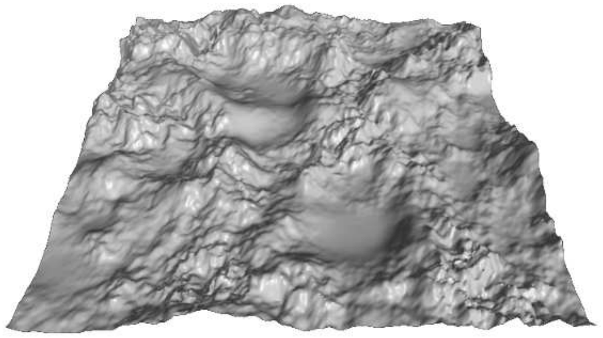
```
1.89442719
```

In the example, a big obstacle in the middle splits the layout to two corridors. The bottom corridor has a width of 1, but the top corridor is only 0.89442719 wide at its narrowest point, marked by the red dashed line, so the total possible flow is equivalent to a corridor that is 1.89442719 wide.



J. Image compression

The latest spy gadget is an earring that has a 2 terapixel camera, a 16 core compound graphics processing and CPU unit, 4 TB of overclocked 512-bit GDDRX RAM, and an user-expandable mass storage interface. It can take and transmit photos of the finest quality, whenever your ear itches; and the massive heatsink lends a very fashionable look to the user.


http://commons.wikimedia.org/wiki/File:Heightmap_rendered.png

Because of certain legal issues (regarding dubious medical concerns about high-energy electromagnetic radiation near the user's brain), the bandwidth that the earring may utilize during wireless image transmission is extremely small. Until these obsolete regulations are removed, the WSA has to compress the images very severely. They came up with a creative lossy compression format that will keep major features reconstructible, and loses only some unimportant details.

The decompressor is already available; your task is to write the compressor. The WSA is looking for an implementation that provides the best image output while staying inside the strict size requirements.

Given a grayscale input png (saved directly from the Intelligent/Imaging Serial CCD), you need to find a suitable compression. The compressed image is described only by a couple of reference points. Each reference point is a pair of coordinates and an intensity. The original image is reproduced by determining the intensity of the missing pixels between reference points using the value of all reference points. The exact method for any given pixel on the output is:

- if there is a reference point on the pixel, output intensity is the same as the intensity of the reference point
- otherwise the intensity of the pixel is the weighted arithmetic mean (average) of all the reference points' intensity. For each reference point, the weight is the reciprocal of the squared distance from the pixel ($1 / (dx^2 + dy^2)$).

Multiple reference points can not be placed on the same coordinates, all reference points must be within the boundaries of the image.

The maximum number of reference points used to store an image is the sum of *SX* and *SY,* the horizontal and vertical dimesions of the image in pixels.

Input

A grayscale png file.

Output

The first line is an integer *N*, the number of reference points used. The following *N* lines are reference points in *X Y I* format where *X* and *Y* are the coordinates of the reference point and *I* is the intensity between 0 (black) and 255 (white).

The top-left pixel of the image is X=0;Y=0.

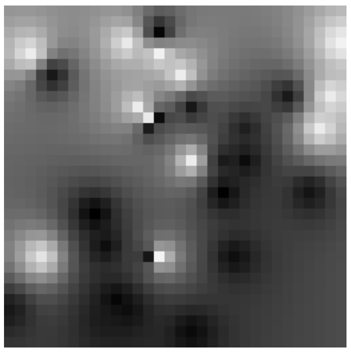
Scoring

Scoring is based on the root mean squared error (RMSE):

Valid submissions are rendered to a grayscale image. The render and the input image are compared. The intensity difference of each pixel is squared, then the average of the resulting values is calculated. The square root of the average is the RMSE. This is then compared to the best submission so far. (The eval score in the submission system is trunc(1000000*RMSE)):

```
  SCORE = 100*(1 - sqrt(1 - BEST/RMSE))
```

Example input



Example output (part 1)

```
32
14 10 12
4 6 24
0 28 24
11 3 233
17 14 236
13 10 234
30 8 235
31 3 242
28 17 20
14 23 243
10 27 6
2 4 236
20 14 27
20 17 1
13 11 28
```

Example output (part 2)

```
9 22 13
17 9 23
22 14 11
16 6 237
9 29 26
14 4 241
22 11 31
17 30 10
12 9 236
26 8 26
8 19 0
21 23 14
3 23 227
14 2 6
29 11 235
13 23 22
11 28 13
```