CHALLENGE24

INTERNATIONAL 24-HOUR PROGRAMMING CONTEST

APRIL 26-28, 2013

http://ch24.org

CHALLENGE24
INTERNATIONAL 24-HOUR PROGRAMMING CONTEST

MAIN SPONSOR

SAP®

DIAMOND GRADE SPONSORS

FORNAX

Google™

skype™

YAHOO!

GOLDEN GRADE SPONSOR

facebook®

ORGANIZER

mave

Electrical Engineering Students' Hungarian Association
www.eestec.hu

eestec
LC Budapest

Contest

Welcome to the 13th International 24-hour Programming Contest!

Rules

The contest starts at 2012-04-27 09:00 CEST and ends at 2012-04-28 09:00 CEST.

No solution can be submitted after the 24 hour time is up.

Web server

General contest related information will be available on our web server at http://server.ch24.org/.

Submission site

The same submission system will be used as during the Electronic Contest. It will be available at http://server.ch24.org/sub/.

Task summary

There are various kinds of problems, with various scoring rules and submission methods. Here we provide a short summary:

Task	Web submission	Interactive	Score decreases with time	Penalty for wrong answer	Time delay after fail/pass	Scaling	Max score
A (Tower Defense)	Yes	No	Yes	-5	0/0	No	1000
B (Dissection)	Yes	No	No	-5	0/120s	Yes	1000
C (Laser - Lines)	Yes	No	Yes	-5	0/0	No	1000
D (Laser - Cover)	Yes	No	Yes	-5	0/0	No	1000
E (Laser - Boxes)	Yes	No	No	-5	0/120s	Yes	1000
F (Laser Hole in the Wall)	Yes	Yes	Yes	0	0/0	No	2500
G (Sabotage)	Yes	No	Yes	-5	0/0	No	1000
H (Hatch)	Yes	No	Yes	-5	0/0	No	1000
I (Eye Practice)	Yes	No	No	0	0/0	Yes	0
J (Eye For-Score)	Yes	No	No	0	0/0	Yes	1000
K (Paper Boy - Delivery)	No	Yes	No	0	0/0	Yes	2500

Task	**Web submission**	**Interactive**	**Score decreases with time**	**Penalty for wrong answer**	**Time delay after fail/pass**	**Scaling**	**Max score**
L (Paper Boy - Agility)	No	Yes	No	0	0/0	Yes	2500
M (Dead-line)	Yes	No	No	-5	0/120s	Yes	1000
N (Zombings)	No	Yes	No	0	0/0	Yes	3500

- Web submission: A static output file must be uploaded through the submission site.
- Interactive: During the solution or the submission, either network communication or other kind of interaction is necessary.
- Score decreases with time: Submitting at the end of the contest is worth 70% of what would be awarded at the beginning.
- Penalty for wrong answer: Wrong answer gets -5 points (different value may be specified explicitly in the task description).
- Time delay after fail/pass: Duration in minutes while no new submission is accepted after a wrong/correct answer.
- Scaling: The score for this problem may change over time depending on submissions by other teams. (Note that your last submission is considered and not your best one.)

Note that the scorings of the Paper Boy and Zombings tasks have an hourly schedule, for details see the task descriptions.

Ports

Port	Task	Service description	Connection direction server <-> team
`80`	-	web	←
`6667`	-	irc	←
`u4242`	-	time server, UDP broadcast destination port	→
`u123`	-	ntp server	←
`u53`	-	dns server	←
`u67,u68`	-	dhcp server	←
`u10001`	KL (Paper Boy)	UDP bicycle telemetry	→
`u10002`	KL (Paper Boy)	UDP video stream (mjpeg)	→
`u10003`	KL (Paper Boy)	UDP control service	←
`u20001`	N (Zombings)	UDP broadcast destination port	→
`20002`	N (Zombings)	TCP control service	←

Ports starting with `u` are UDP ports.

The UDP destination ports should be opened by the teams (if they are interested in the information).

All other services are hosted on `server.ch24.org`.

Contact

General contest related information and data will be published on the web at http://server.ch24.org/.

Important announcements will be made on the `#info` irc channel and will be published on the web as well.

For general discussions and questions join the `#challenge24` irc channel.

There will be separate channels for task related problems as well: `#A`, `#B`, `#CDEF`, `#G`, `#H`, `#IJ`, `#KL`, `#M`, `#N`.

Prologue: Zombie Apocalypse

The majority of human population turned into zombies due to a virus spreading over the internet implanted by extraterrestrial intelligence.

You’re one of the last alive holdouts in a depopulated city. You spend your remaining time with various fanciful pursuits aiming to delay the time of the final munching (of your brain).

You have an inkling that the first infection vectors might have been the auditors you had to deal with a couple of months before - such a short time elapsed since then, and so much has changed...

A. Tower Defense

http://www.instructables.com/id/Free-Boxes-Free-Waste-Removal/

Zombies are after you. You need to improvise some sort of barricade from whatever you find in the streets. The most useful items are large cardboard boxes you load with bricks, as you can build tall stacks of them.

Given *N* boxes with positive weight, strength and height build the tallest tower (each item in the tower must have enough strength to bear the sum of the weights of the items above).

Input

The first line of the input contains N, the following N lines each contains three integers H,W,S, the height, weight and strength of an item.

Output

The output should give the tallest tower: the first line is the height and the second line the list of items of the tower from the bottom to top (items are indexed from 0 based on their position in the input).

Example input

```
10
4 3 8
5 2 9
7 1 4
3 2 5
4 1 3
9 4 6
1 1 5
1 3 7
2 2 4
3 2 6
```

Example output

```
29
1 5 3 6 2 4
```

B. Dissection

Ever wondered why there are so many beheaded zombies walking while beheaded humans are usually laying still? The answer is simple: zombies are more robust due to distributed redundant organs - cutting off only a small part will leave multiple instances of anything vital still connected in the rest of the body. In an ideal world, you would shred zombies to tiny pieces or even grind them to dust. But as in many other fields of life, one can't achieve perfection (there are too many zombies and too few resources for slicing them). However, with some extra cleverness, you can optimize your zombie-slicing-throughput: if you know how zombie organs are distributed, you can minimize the number of cuts needed. As long as each organ will end up separated from all other organs, you are fine, the zombie won't ever come to its horrible pretend life again.

Having a plane and some points in it, give a set of lines that separate the plane into subplanes so that each subplane can contain at most 1 point. The set should be as small as possible.

Point coordinates are all integers. Lines are given by two different points (with integer coordinates). Lines have a "direction" from their first point to their second point; this is used to determine that when a line intersects an input point, the input point is considered to be on the right side of the line.

Lines can only be horizontal (Y1=Y2), vertical (X1=X2), or diagonal (dX/dY = 1 or -1).

Input

The first line is the number of points given. Then an X Y pair is given per line for each point. X and Y are integers.

Output

Lines of X1 Y1 X2 Y2 for each line. X1, Y1, X2, Y2 are integers.

Score

This is a scaled problem, the evaluated score S is the number of lines in the output. The scaled real score is

```
SCORE := round(100 * (1 - sqrt(1 - Smin/S)))
```

Where *Smin* is the best submission so far.

Example input

```
4
3 1
4 5
6 6
8 4
```

Example output

```
3 2 8 7
2 8 8 2
```

CDEF. Laser CNC

http://www.personal.psu.edu/users/s/t/ sts5035/Cnc-World.jpg

One of the most efficient survival strategies for an apocalypse is to keep a CNC machine around. It lets you build your own tools and weapons (as long as the power grid is up, the shops are selling raw materials, etc.)

The next few tasks are about controlling your Laser CNC to cut out shapes from sheets. Since there is only one instance of the real hardware for 30 teams, a special arrangement is employed to let everyone have a go:

- the first three of the four tasks are simulated
- real CNC slots are limited, requests are queued
- the actual material to cut is paper; the device is capable of cutting plastic, but cutting paper goes much faster allowing more cut requests for 24 hours
- **for the 4th task, we accept submissions only until the end of the 22nd hour of the contest (07:00)**

The hardware

The laser cut CNC can move a laser module over the workpiece on two axes (X and Y) in discrete *steps*. The output power of the laser can be controlled between 0% and 100%. If the laser is on, it starts burning the workpiece, which starts to smoke out at the given location. After transferring enough heat at a specific point, a thin workpiece burns through. Moving at a low speed while the laser is on results in a cut along that line.

The CNC runs a tiny controller that interprets a sequence of instructions and program timers to guarantee synchronous operation of motors and the laser. There is a simple language called *laser script* for programming the controller, with two main features: setting the value of registers (syntax: *register=value*) and starting an interpolation (moving the laser to the end coordinates; syntax: *start*). The relevant registers are the following:

name	function
x	current X coordinate (not writable)
y	current Y coordinate (not writable)
tx	target X coordinate
ty	target Y coordinate
dx	speed: divider for the X axis
dy	speed: divider for the Y axis
pwm	laser power
bd	brake delay
dd	drill delay

The normal procedure is setting the value of registers and issuing a *start* command that will start moving until both target coordinates are reached. If on one axis the target coordinate (e.g. *tx*) is reached before on the other (e.g. *ty*), moving stops in that direction. The cut stops when both *tx* and *ty* are reached. *Start* is a blocking operation, the next instruction is not interpreted until *tx;ty* is reached.

Movements along the axes and delays (*bd*, *dd*) need to be synchronized. This is achieved by dividing motor steps and delays from a central timer generating ticks at about 35 kHz:

- STAGE 1: **drill delay:** if pwm is not zero and *dd* is not zero, set laser power to *pwm*, and load a counter with *dd*; decrement the counter each tick until it reaches zero; do not move the motors while doing this;
- STAGE 2: **move:** there is a counter for the X axis, which is loaded with *dx* on *start*, then in each tick the counter is decremented; when the counter reaches 0, a step in the right orientation along X towards tx is generated on the motor, *x* is updated and the counter is loaded with *dx* again; if *x* == *tx*, stop stepping in this direction else go on with STEP2 procedure;
- STAGE 2: the Y axis has a similar counter loaded with *dy* with the same rules, targeting *ty*; the two counters and motor steppings are rendered in parallel, until one of them reaches its target
- STAGE 2: if both targets (*tx* and *ty*) are reached, set motors in brake mode and move on to STEP3
- STAGE 3: **braking delay:** if *bd* is not zero, switch off the laser, load a counter with *bd*, decrement the counter in each tick; when zero is reached, set laser power to *pwm* again, insert 1 tick delay (for timing reasons) and proceed with the next command

The above procedure is called *interpolation*. We distinguish between two kinds of interpolation:

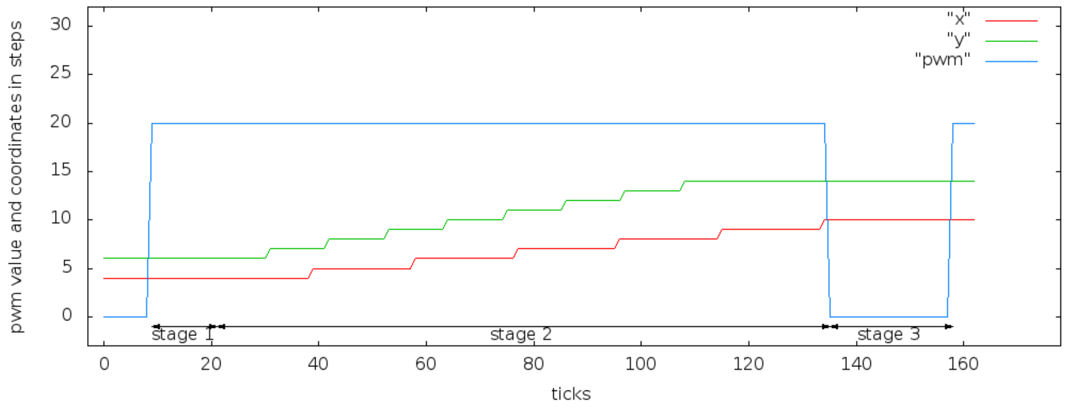
- the *traveling* interpolation: the laser is turned off (*pwm=0*), no cut is performed;
- the *cutting* interpolation: the laser is turned on (*pwm* is larger than zero, the workpiece is being cut during the interpolation.

How deep the cut is depends on many factors, most importantly on the product of the speed of the movement and the laser power. Thus travel speed is much higher than cut speed, and cut speed is chosen to be the highest possible speed at the maximum laser power so that it still cuts the workpiece.

An example laser script:

```
dd=12
bd=23
dx=19
dy=11
pwm=20
tx=10
ty=14
start
```

Assuming the current *x;y* coordinates are 4;6 and the laser is turned off (*pwm*=0), the above script will produce the following step pattern:



Note: the actual values in this example are invalid on the real device (speeds too high) and are chosen for simpler visualization.

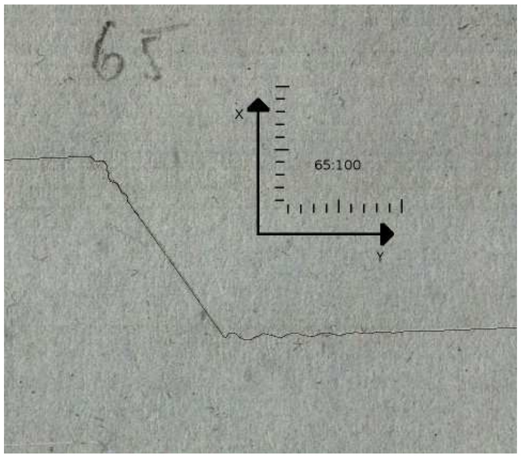
The slope is not calculated properly in this example: *y* reaches its target (14) much sooner than *x* reaches 10. This will result in cutting two line segments, a slope and a short segment aligned with the X axis. However, after *start* finishes, the new *x;y* coordinates are *tx;ty* (10;14). For the braking delay (*bd*), the laser is turned off and the original power is restored after the delay, which will leave the laser turned on at power 20 after the cut.

For traveling between cuts, the laser shall be turned off to avoid cutting across the workpiece. This can be done by inserting a *pwm=0* before the next start command. Since laser power is restored after the braking delay, there will be a very thin spike (a blink of the laser) between a cutting interpolation and a traveling interpolation. This spike is so narrow that it will not affect cut quality.

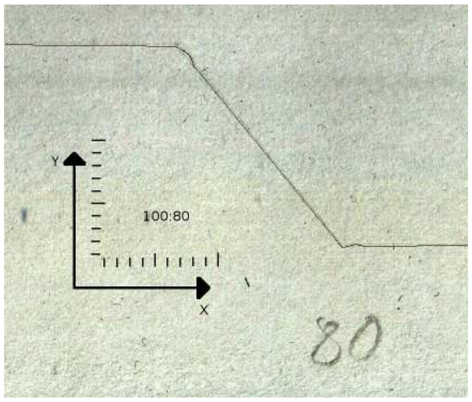
The **drill delay** (*dd*) is provided for two reasons: some material can be cut better when starting from an edge. By drilling a hole before starting the cut, the cut is always started from an edge. When cutting a continuous series of line segments, only the first line requires a non-zero *dd*; drilling between such lines will not just slow down the process but may result in small glitches where lines are connected.

The second delay, **braking delay** (*bd*) is provided as a compensation for the mechanical inertia of the system. An ideal CNC would feature motors strong enough to speed up and slow down the laser module within a fraction of a single step. The real device is not that strong. In the worst case if the laser module is moving very fast in one direction and the motors start moving at a high speed in the other direction, the mechanics may miss steps during the direction change. The firmware will control the motor to rotate N steps in the opposite direction of the current movement but it will stay in place only slowing down or stopping the movement. The firmware will not know about this error and will assume the new coordinates. This sort of inaccuracy sums up after multiple direction changes. To avoid this problem, the braking delay is implemented to allow the mechanics to slow down while motors are trying to keep the laser module in place. Doing this between each two interpolations avoids the rapid change of direction.

Note: not only braking but accelerating the module also takes considerable effort for the motors. This may produce tiny oscillations, especially along the X axis, during both acceleration and braking. There is no delay inserted for acceleration as using a sufficient *bd* will avoid the inaccuracy. Below is a photo of a typical oscillation produced by moving at constant speed along one axis and rapidly speeding up/stopping along the other axis without brake delay.



large oscillations



small oscillations

(scale is 10x10 mm; speed values for this test were **beyond** the recommended/allowed values and bd was 0; operating the device within the absolute minimum/maximum range will reduce oscillations to a nearly unnoticable level)

Choosing parameters which don't violate the *absolute minimum/maximum* values is required. Choosing values from the **"safe value"** column will work for any combination of parameters (e.g. cutting direction, previous state of the device). It is possible to optimize a set of interpolations for speed by using the **"fastest value"** column, but care should be taken to avoid inaccuracy. For example when cutting a series of line segments with only slightly differing angles, the braking delay can be decreased, but it should be set back to the safe value for stopping after fast travel.

property (register)	**unit**	**absolute minimum**	**absolute maximum**	**safe value**	**fastest value without error**	**comment**
travel in direction X (dx)	div	80	5000	110	108	it is not possible to speed up to 80 directly; use a value >=108; using lower values may result in missed steps
travel in direction Y (dy)	div	80	5000	110	108	
cut in direction X (dx)	div	80	5000	450	420	refer to the cut test page; cutting a 10% slower than the maximum speed will not have visible affect on cut quality
cut in direction Y (dy)	div	80	5000	450	320	
brake delay (bd)	div	1000	32000	2000	1500	using a brake delay less than the safe value may result in missed steps when changing direction during travel
drill delay (dd)	div	0	32000	4000	0	drilling is not required for starting a cut on our current workpiece (*dd* should be 0); in case a single hole should be drilled at a specific coordinate, use the safe value without changing *tx;ty*
cut power (pwm)	1	0	1023	any	1023	0 means laser is off; 1023 means laser is at 100% power
X coordinate (tx)	step	0	3200	3190	n/a	restricted by the physical geometry of the device
Y coordinate (ty)	step	0	3200	3190	n/a	

Units:

- div: divider used on the ~35 kHz timer
- step: number of steps along an axis

Laser script submissions are accepted only if *absolute minimum/maximum* ratings are not violated.

Laser script syntax

Each line of a laser script is a comment or instruction or empty line. Comments start with the hashmark (#). Empty lines may contain whitespace. Instructions are:

- *register = 16-bit integer written in decimal format*
- *start*
- *end*

Different registers require signed and unsigned values. The *start* instruction will start an interpolation using the current register settings. No other instruction is executed until the interpolation is finished. The *end* instruction is the last instruction terminates the script. Explicit termination is required.

Some of the evaluators will require specially formatted comments to communicate meta-data (marking coordinates, tagging shapes).

Simulator, practice server

The first 3 problems are evaluated exclusively in the simulator. The simulator uses the firmware code to accurately reproduce the instruction interpretation and timing of the real device and somewhat-accurately simulates the behaviour of the workpiece. The output of the simulator is a png where black pixels represent untouched surface of the workpiece, white pixels represent cuts and gray pixels represent partial cuts or surface scratches. A text file with other properties of the simulation (total time elapsed, bounding box, etc) is also returned.

The real workpiece (standard 80 gram black paper) may behave slightly differently, but it is guaranteed that safe values described in the table above will work with it.

A practice server is provided as a virtual task on the web based submission system. This task is not scored. Teams may submit laser scripts for simulation any time and the resulting png will be available to the team through a web page.

Using the actual hardware

The last task in this group is evaluated using real cutouts, but can also be used for testing the device. There is a queue of submitted cut requests. One team may only have only one submission in the queue. A request is either pending or being executed. While pending, the team may cancel the submission and submit a new script - to the **end** of the queue. Submissions being executed can not be replaced and the team has to wait until they are finished.

The CNC is operated in batch of 1..4 submissions, in 30 minute cycles. In the best case a cycle will process 12 requests, but more often only 8..10. If all 30 teams have a submission in the queue, the last item will be scheduled in 2.5 to 3.75 cycles (75..112 minutes).

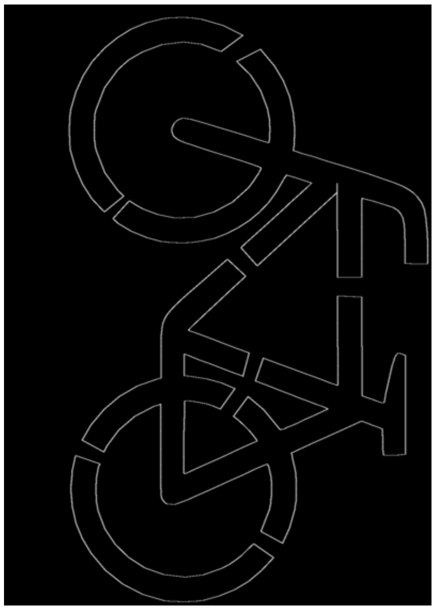
After the submission is cut by the laser, the team needs to visit the laser eval site and decide whether the submission was only for testing and should be discarded or should be tested and scored. The submission is removed from the queue only after it’s been scored or discarded.

Because of the time constraints and properties of the task, long queues are expected by the 22nd hour. A team who starts submitting solutions for this task early will probably have much shorter turnaround times.

There is a test cut page on display at the laser CNC; it demonstrates sample cuts with different settings. We recommend careful evaluation of the sample cuts and solving at least the first two problems on the simulator before touching the real hardware.

Please note: the last submission for the actual hardware is accepted until the end of the 22nd hour of the contest **- no submission is accepted in the last two hours** to let accumulated submissions go through the system until the end of the contest.

C. Laser CNC - Lines



source:
http://www.wpclipart.com/recreation/cycling/bicycles/plain_bicycle_icon_large.png.html

After receiving the CNC, you realize the software supplied by the vendor will not meet the high requirements mandated by the complexity of the anti-zombie weapon technology. You decide to reimplement the system piece by piece, always testing your new tools against the supplied demo designs.

Your first task is to reimplement the lowest level driver that renders vector graphics into (slightly) optimized laser CNC scripts.

Input

The first line of input contains integers N and T, followed by N lines each describing a cut with integer coordinates X1 Y1 X2 Y2. The submitted solution is accepted only if it finishes cutting all lines within T seconds.

Output

A laser script that cuts all N lines precisely:

- no *absolute miniumum* or *absolute maximum* values are violated
- each cut starts and ends at the specified coordinates
- speed and output power requirements are met for cutting the paper through

The evaluator will render the pixmap of the cut (*C*), keep only pixel values of total cut (white, value 255) and will compare it to the pixmap of the reference solution (*R*). Our reference output (*R*) is also provided in png format among the input files.

First *R* is compared to a bloated version of the *C* - all white pixels of *R* must be a white pixel of the bloated *C*. Then the same test is repeated in reverse order (*R* is bloated and matched against the original render of *C*.

Bloating is a special case of blur: the input image is copied to a new image so that each pixel of the new image is white if and only if the 5*5 pixel rectangle centered at the original pixel contains at least one white pixel.

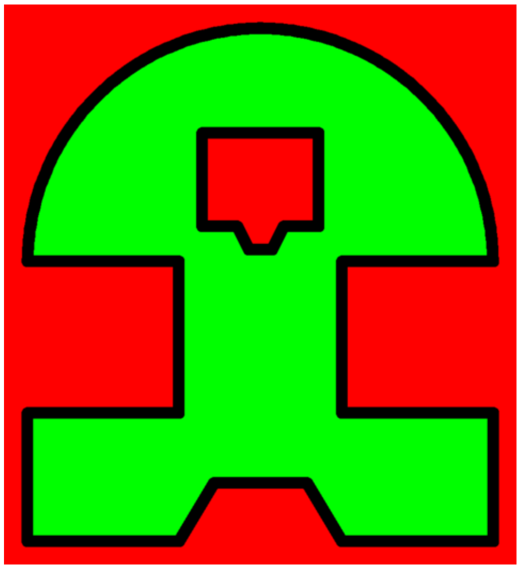
Example input

```
4 100
100 100 100 400
100 400 300 400
300 400 300 100
300 100 100 100
```

Example output

```
bd=2000
dd=0
dx=120
dy=120
tx=100
ty=100
pwm=0
start
dx=450
dy=450
tx=300
pwm=1023
start
ty=400
start
tx=100
start
ty=100
start
pwm=0
start
end
```

D. Laser CNC - Cover



The second layer of your software is a CAD system that can design a cutout of the specified shape. The shape is not specified precisely, but as a set of areas which should or should not be covered by the cutout. This task is common for building 3D objects by slicing them in layers that are cut out one by one and then stacked. Since this process is very time consuming, you will need to optimize your cut script for speed.

You need to provide a script that cuts out polygons in a way that the pixmap rendered cutouts will cover all *green* pixels but never cover any *red* pixel of the input. The script is fed into the simulator and the resulting png is flood-filled with *blue* starting from the middle from a large green area.

The submission is not accepted if the filled cutout does not have a *blue* pixel over any *green* pixel, or has *blue* pixel over any *red* pixel, or there are some components that are only connected with narrow stripes (this is checked by scaling down each side of the pixmap to 25% and the cutout should be still fully connected).

Input

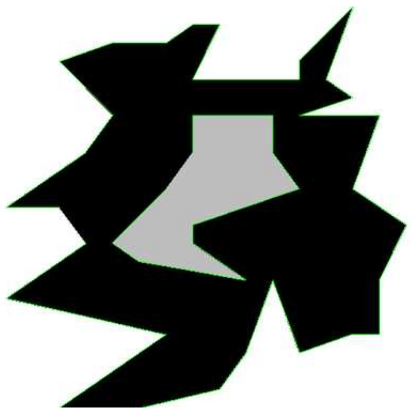
Input is a png containing black (#000000), green (#00ff00) and red (#ff0000) pixels. Black pixels are neutral (does not matter what the rendered png has over them).

Output

Output is a laser script with the lowest possible runtime that meets all requirements.

Example input	Example output
Provided as input 0.png.	See in 0.out, next to 0.png.

E. Laser CNC: Boxes



Access to raw materials is limited during the zombie apocalypse, minimizing scrap is an essential part of the design work flow. It is so important that you decide to solve it in software and apply the solution on any design you create.

Your task is to cut out a set of predefined shapes using the smallest bounding box possible. Shapes must not be rotated or scaled.

Input

The first line contains the number of shapes (integer). The rest of the file describes the parts you need to cut out, in the following format:

- first line is *C* the number of contours
- the next *C* lines are contours: first integer is *V*, the number of vertices the contour has, the next 2 * *V* integers are the x;y coordinates of the vertices

Output

The server expects a valid laser cut script, terminated by an *end* instruction. The sumbission is accepted only if: the pixmap rendered output contains all shapes and the interior of the shapes don't intersect with each other (We will match the shapes to our reference cutouts with a small tolerance).

Scoring

This is a scaled problem, the evaluated score S is the area of the bounding box.

The scaled real score is

```
SCORE := round(100 * (1 - sqrt(1 - Smin/S)))
```

Where Smin is the best submission so far.

Example input

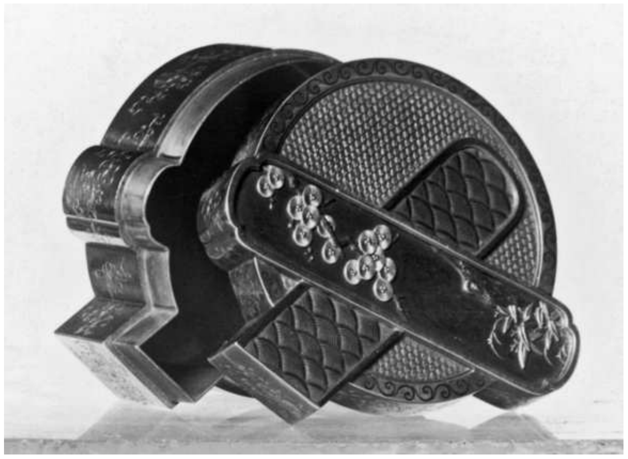
```
2
1
3 100 100 250 280 84 230
1
3 100 100 150 180 184 70
```

Example output

```
# offset 0 -69 -94
# offset 1 55 -44
dx=112
dy=579
tx=31
ty=6
pwm=0
bd=2000
dd=2000
start
dx=3684
dy=453
tx=15
ty=136
pwm=1023
start
dx=470
dy=1560
tx=181
ty=186
start
dx=703
dy=586
tx=31
ty=6
start
dx=119
dy=294
tx=155
ty=56
pwm=0
start
dx=478
dy=1338
tx=239
ty=26
pwm=1023
start
dx=1524
dy=471
tx=205
ty=136
start
dx=849
dy=531
tx=155
ty=56
start
pwm=0
start
end
```

F. Lase CNC - Hole in the Wall

There is a special way for escape from the zombie city. A last-minute low cost parcel service offers the remaining free space on a cargo plane. You want to build a box (or coffin?) that can contain you and your most cherished objects, that can fit the specially shaped empty space that remains on the plane (since the company is not willing to move the already placed cargo for you).



http://upload.wikimedia.org/wikipedia/commons/c/cf/
Japanese_-_Box_in_the_Shape_of_Two_Inksticks_Crossed_on_an_Inkstone_-_Walters_571220.jpg

To prove you can cut out such a box, you need to design and cut out the scaled version of the lid. There is a *go* and a *no-go* mask your lid will be tested against. The *go* mask represents the free space the box will need to fit in, so your lid must be smaller than that. The easiest way to pass the *go* mask test is to design a very small box - however, you wouldn't fit in a very small box. The *no-go* mask is used to test whether your box is big enough. Your lid must collide (not fit in) the *no-go* mask.

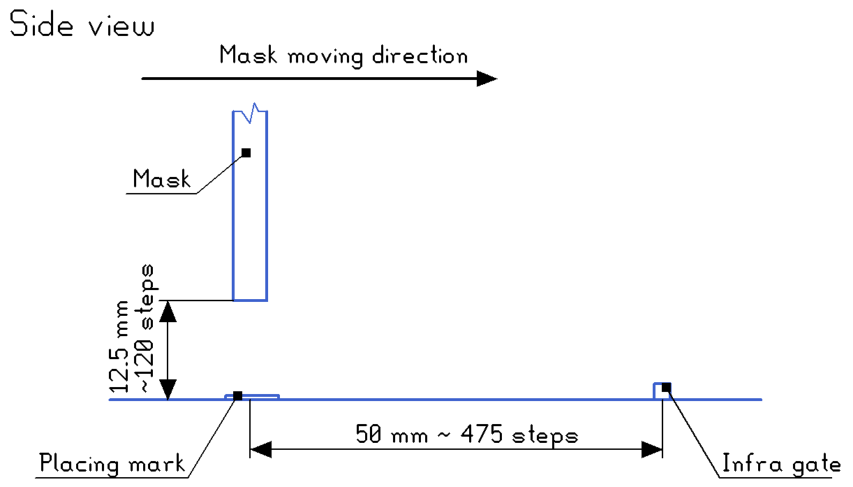
The test is performed on a semi-automated test bench. You need to design a base that should keep your lid standing vertical. The cut must fit in the following bounding box (x;y .. x;y): 0;0 .. 850;1235. To produce the construction, you are allowed to use:

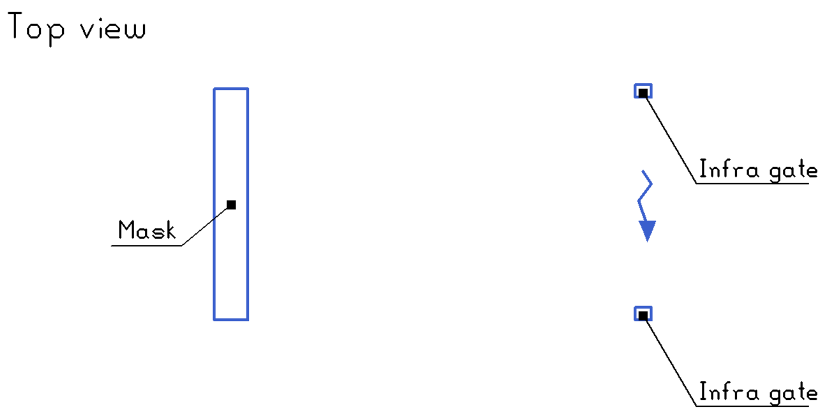
- the laser cutter to cut whatever shape you find appropriate, within 2 minutes (net laser script run time on the simulator)
- your hands to flip the cutout
- your hands to assamble your model
- parts of a multi-part cutout as long as all parts are produced by the same submission

However, you are not allowed to:

- cut with anything else than the laser
- tear the cutout by hand
- do multiple 2-minute runs on the same piece
- aggregate parts of different runs
- use glue or other substances to modify the cutout
- bend, roll, or other modification (only linear flips are allowed)

The test bench will first pull the *go* mask, then the *no-go* mask over the cutout. The cutout is required to stay erect while the *go* mask is passing over it, while it is required to collide with the *no-go* mask and **trip over**. This is validated by an infra opto gate installed next to the mark, as shown on figure below. Once the cutout falls, it has to interrupt the path of the light in the opto gate.





Note: the cutout needs to be sort of a 3d object after assembly: the material we work with is thin paper. Still the cutout needs to be strong enough to stay erect after installation and strong enough to fall down due to the collision with the *no-go* mask, instead of just flipping away. It is also requred that the cutout is thick enough to interrupt the path of the light - after falling, it must be at least 80 steps high at the gate.

Input

There is a single test case for which the *go* and *no-go* masks are provided in png. Please work with at least 2% safety margin for both *go* and *no-go*: the laser cutter and the test bench are not nanometer-precise and you will need to place your assembled object manually on the test bench.

Output

You can submit your laser script to the end of a queue. Each team may have only one script in the queue; once it is evaluated, the team is free to submit the next attempt. As long as the script is not marked as being under evaluation the team may remove the script from the queue (cancel the submission).

We will remove scripts from the queue and feed them in the laser cutter in a panelized manner (multiple teams will share the same cutting slot to reduce overhead). Once the cutouts are ready, the teams will be notified/summoned for evaluation. Teams have 2 minutes to arrive at the test bench after the notification, else the cutout (and the submission) is discarded.

For evaluation, the team needs to assemble the cutout in a controlled environment and place it on the mark on the test bench. The submission is accepted, if:

- the cutout was produced within the specified cut-time
- the assembly of the cutout as specified (no scissors, no glue, etc.) and did not take more than a minute
- the cutout **did not** interrupt the path of the infra light after tested against the *go* mask
- the cutout **did** interrupt the path of the infra light after tested against the *no-go* mask
- there was no external disturbance (e.g. vibration, air blow, etc) during the test
- the evaluation of the cutout succeeded by at most the second attempt

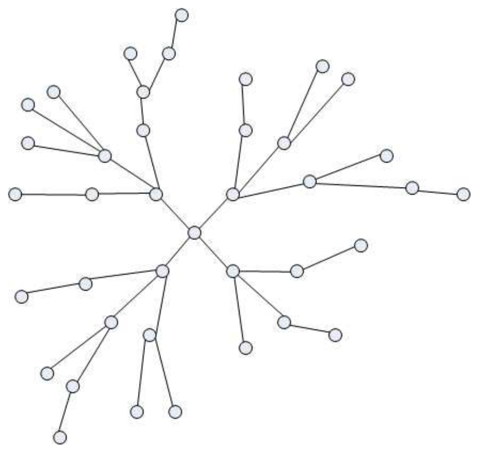
Mask geometry

The mask is given as input mask.las. The scale on the right is placed under the mask, more or less aligned with the mask. The *no-go mask* is next to the scale and the other mask on the left (with the smaller cutout) is the *go mask*. The two masks are centered using the four positioning holes cut around the edges.

On the test bench, ths is the front view of the masks with the cutout in front, the masks moving towards the viewer ("out from the screen"). If orientations are still not clear, please visit the test bench and take a look between two evaluations!

G. Sabotage

Zombies have a special point-to-point inter-zombie communication facility with multiple channels per zombie, with limited range (using smelly pheromones). When zombies are deployed in bigger quantity in large cities, they need to build a network and relay messages so that any zombie can send messages to any other zombie.


http://upload.wikimedia.org/wikipedia/commons/2/24/Network_Tree_diagram.png

Instead of redundancy, they go for low latency so from a possible set of connections between zombie pairs they select some so that the sum of the latency of the selected ones is minimal and all zombies can communicate in the network by routing messages along the selected connections.

Your task is to ruin the zombie network by blocking a few point-to-point connections in a way that the resulting zombie network will have worse latency than the original one. Zombies reconfigure their network as soon as some of their selected connections are blocked and they keep the sum of the latencies to a minimum at all times.

Blocking a connection is expensive, you need to minimize the total cost of your blocking operation.

Input

First line contains N and M the number of zombies and the number of possible network connections, the following M lines each describe a connection with four integers: A, B, L, C, that is a connection between the A,B zombies with latency L and blocking cost C. (Zombies are indexed from 0, the latency of a connection is a fixed value, the same pair of zombies may have several different potential connections between them with different parameters.)

Output

The first line is the minimum cost of the blocking operation.

The next line should contain a list of edge indices, the connections which should be blocked. (After the listed connections are blocked the minimum sum latency achievable in the remaning network should be greater than originally. Edges are indexed from 0 in the order they appear in the input.)

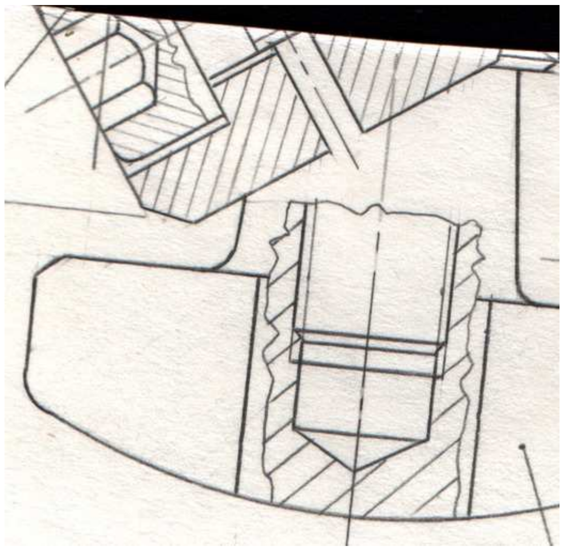
Example input

```
4 7
0 1 1 3
0 2 1 9
0 3 2 1
1 2 2 2
1 3 2 1
2 3 2 2
2 3 3 3
```

Example output

```
3
0
```

H. Hatch



You need some parts manufactured for your zombie protection suit. These parts are made of steel, and steel can not be cut in your laser CNC. Unfortunately most of the craftsmen have already left the city. The ones you can still contact are old fashioned guys, following all the good tradition: they do not work from CAD files but from printouts. No problem, you can print all your drawings - as long as you have ink in your printer.

The zombie apocalypse reveals a lot of suboptimal system features in all the software and hardware around you, and your CAD program is not an exception. You need to hatch (fill using parallel lines) large polygons on those drawings, but your CAD system does not optimize the hatch pattern for saving ink. You obviously can not afford wasting ink like that!

You are given a number of polygons, that make up your drawing. Only adjacent sides of a polygon touch, and the circumference of different polygons never intersects. These polygons separate the plane into shapes and holes: the outermost polygons are shapes, the ones contained immediately in these are holes, the polygons inside those are again shapes, and so on.

Your CAD program will hatch the shapes (but not the holes) with horizontal lines of distance D. (Horizontal means parallel to the x axis.) If a side of a polygon is horizontal, the program will not draw a hatch line over it. If a hatch line touches a vertex of the polygon, it still counts as one line.

Your task is to optimize the offset of these lines to minimize total number of hatch lines.

Input

First line contains two integers: N, the number of polygons, and D, the distance between two adjacent hatch lines. Then the descriptions of the polygons follow. First line of each polygon is a single integer, K_i, the number of vertices in the i-th polygon. The next K_i line contains two integers, the x and y coordinates of the vertices. Vertices can be ordered clockwise or counter-clockwise.

Output

N floats, the best offset for each polygon. If more than one offset gives the same best result, any of them is accepted. If a polygon is a hole, any number can be written. Each number (X) must be 0 <= X < D

Example input

```
2 10
3
0 0
50 10
0 46
3
20 20
5 8
10 6
```

Example output

```
6.0
0.0
```

(Explanation: for the first value, "`0`", "`6`", "`0.`", "`6.`", "`0.0000`" or any other format of 0 or 6 would be accepted, they all need 5 hatch lines. For the second value, anything between 0 and 10 would be accepted)

IJ. Eye

Most zombies are following a simple minded program and aim to follow and catch and bite you. Some zombies, however, are guards or scouts controlled by the evil coordinators of the zombie apocalypse. These zombies have CCDs built into their eyes. You realize you need to trick these zombies by a record-replay attack: you record still pictures of empty streets and rooms and play these pictures back directly into the eyes of the zombies (using projectors and special optics). Your trick is easily revealed if the projected images are not perfectly aligned. Software correction is needed, since it's impossible to mechanically align zombies with projectors.

http://www.photos-public-domain.com/2013/01/21/eyeball-toy/

Your task is to prove you can perform the above attack. We've built a test bench: a camera looking at a TFT screen. You need to provide a png that we will display on the TFT so that a rectangle of the resulting image produced by the camera looks very similar to the input image.

There are two ways submitting for this task: practice submission and a for-score submission.

I. Practice

Practice submissions are displayed, digitized and the result is sent back (no score is awarded). The resulting camera jpegs will be available at the "files for your team" site (http://server.ch24.org/files/).

J. For-score

A for-score submission displays, digitizes and evaluates the digitized image, but but does not return any image, only determines the score. This is a scaled problem - see the scoring section below.

Input

A png file (the target image)

Output

A 1280x1024 pixel png file to be displayed on the screen. In case of **for-score** submissions, the first (top-left) pixel specifies the rectangle on the camera jpeg that will be compared against the input file:

- red, green and blue are read as 8-bit unsigned values
- x = red*256+green
- y = blue
- x;y marks the first pixel on the camera jpeg that is compared
- the width and height of the compared rectangle is the same as the width and height of the input image

Invalid coordinates (when bottom-right of the rectangle is beyond the jpeg) will result in a failed submission.

Scoring

The evaluated score is the RMSE (root-mean-square error) of the selected rectangle of the camera image compared to the input image:

```
RMSE := round(sqrt(SUM((P-Q)*(P-Q))))
```

where the SUM is taken over each P, Q value of each corresponding pixel and color channel of the images.

The scaled real score is

```
SCORE := round(100 * (1 - sqrt(1 - RMSEmin/RMSE)))
```

where RMSEmin is the best submission so far.

Note that the same submission may give different RMSE values due to camera noise. To reduce this effect several camera images will be averaged before a submission is evaluated.

Examples

Offline examples are not available; please submit practice solutions and look at the resulting camera jpegs.

KL. Paper Boy

Your team has an honorable daytime job: you deliver the local newspaper, which contains absolutely essential information about the ongoing zombie apocalypse. Despite of the sudden drop in interest, you and your team decide to implement the best effort service described in your contract and deliver all the papers even if there are no readers anymore. Less traffic on the streets will speed up the process anyway.

https://upload.wikimedia.org/wikipedia/commons/6/68/Cpc464.computer.750pix.jpg

In this task, you have to implement real time dynamic control of 3 simulated bicycles. You have to deliver newspapers, perform stunts, and run obstacle courses in a world full of hazards.

Score for this task is the sum of two halves (both for maximum 2500 points each): "delivery" and "agility".

Delivery

In "delivery", each of your bicycles can gather game points by delivering paper and completing obstacle courses. For each hour of the contest, the game points gathered by the teams are scaled against each other in a linear manner to get the contest score (best team gets maximum hourly score, worse teams get the maximum hourly score multiplied by the ratio of their points to the points received by the best team). Game points are reset after each hour.

In each hour, every mailbox, checkpoint and course are worth points only once per bicycle. To get the theoretical maximum score for a game hour, each bicycle would have to visit every mailbox and checkpoint, and complete every course once, inside the hour.

Each mailbox and checkpoint has a "level": 1 to 4 (easiest to hardest). Courses are chains of checkpoints, and their score is determined by the level of their last checkpoint.

To receive points for a mailbox, a bicycle has to throw a bundled up newspaper and hit the mailbox. Each bicycle can only carry 30 newspapers at once. Newspapers are replenished by touching a reset point, or forcing a manual reset.

To receive points for a checkpoint, a bicycle just has to touch the checkpoint.

To receive points for a course, a bicycle has to touch all checkpoints in the checkpoint chain in order, reaching each subsequent checkpoint within 30 seconds after the previous one, without touching any other checkpoints or any resetpoints.

Awarded game points:

- Mailbox level 1 **- 1 point**
- Mailbox level 2 **- 5 points**
- Mailbox level 3 **- 10 points**
- Mailbox level 4 **- 20 points**
- Checkpoint level 1 **- 2 points**
- Checkpoint level 2 **- 8 points**
- Checkpoint level 3 **- 16 points**
- Checkpoint level 4 **- 32 points**
- Course level 1 **- 20 points**
- Course level 2 **- 80 points**
- Course level 3 **- 160 points**
- Course level 4 **- 320 points**
- Manual reset on a bicycle that's alive **- -1 point**

The hourly maximum contest score is 2500 divided by 24, multiplied by 0.5 for the first hour, 1.5 for the last hour, and linearly scaled in between.

Agility

The game tracks the completion time for obstacle courses (although the time doesn't matter for the "delivery" score). In each 6 hour period, leaderboards are built for each course completed by at least one team, using the completion time: the quickest team gets first place. For each course that the team completed at least once (with any bicycle), a value is calculated:

$$\text{level} * 0.8^{\text{place-1}}$$

Where *place* is the place in the leaderboard for the course (starting with 1), and *level* is the difficulty of the last checkpoint in the source (1 to 4). These values are then summed for the team, and the results are scaled against each other in a linear manner, with maximum score for each 6 hour period being 2500 / 4 = 625.

Bonus Task

Until the end of the 22nd hour (7:00), you can submit your own bicycle map (a file in the described map format), to the "Paper Boy - Your Own Map" task in the submission system. We don't plan to award any score for this. However, if we find a team-submitted map that we like ~~that happens to work in our engine~~, we'll swap the game map to the team's map for the last hour of the contest (from 08:00).

World

The Paper Boy world is a 3D world with an euclidean coordinate system.

Orientation:

- increasing X goes "east"
- increasing Y goes "north"
- increasing Z goes "up"
- 1 unit = 1 meter

The world consists of tiles. Each tile is an axis-aligned box 5 meters in the X direction, 5 meters in the Y direction, and infinitely tall in the Z direction. In other words, if TX and TY are non-negative integers, then each tile contains the following section of the world:

```
X = TX*5.0 ... (TX+1) * 5.0
Y = TY*5.0 ... (TY+1) * 5.0
Z = -infinity ... infinity
```

Each tile contains a type of floor, and may contain some objects. Objects are horizontally centered in the tile, and vertically begin on one of 8 special heights (called layers) - meaning objects stand on layers. One tile can have only one object standing on one layer, so one tile can have at most 8 objects.

The layers and their heights are:

height	layer
-2.5	"moat"
0.0	"road"
0.08	"sidewalk"
2.58	"elevation 1"
5.0	"roof"
5.08	"elevation 2"
7.58	"elevation 3"
10.08	"elevation 4"

There are four types of floors. A floor fills its tile horizontally, at a certain height. Floor types and their heights are:

height	floor type	description
0.0	"road"	normal road surface
0.08	"grass"	can be slippery
0.08	"sidewalk"	physically the same as the road, but slightly higher
-2.5	"lava"	a trench, with lava at the bottom. Don't fall in it

Map File Format

The map file is a text file with LF line endings. The first line of the map file contains a single integer: the map size, in tiles. The map is square, and contains `map_size*map_size` tiles.

The following `map_size` lines contain the rows of the map. The first line has tiles at TY=0, the next line has TY=1, etc.

Each line has `map_size` words. The first row specifies the tile at TX=0, the next one at TX=1, etc.

Each word is 10 characters long.

position	description
1	space (for separation)
2	floor type
3..10	objects in the 8 layers, in order of layer height.

The floor type is one of:

char	description
'r'	for road
'g'	for grass
's'	for sidewalk
'l'	for lava

The object type is marked by one of the following characters: ".1234hdulrftc_|ko".

char	meaning	description
’.’	empty	There’s no object on the given layer.
’1’, ’2’, ’3’, ’4’	mailbox	Mailboxes have four different scores, ’1’ lowest, ’4’ highest. Physically, a mailbox is a standing cylinder, with 0.5m diameter and 1.8m height.
’h’	house	A house is a cube that horizontally fills the tile, and is vertically 5.0m high (so the roof of the house is at layer height + 5 meters).
’c’	reset point	When bicycles are reset, they’re spawned standing on the nearest reset point. Reset points don’t constitute a physical obstacle, but bicycles can touch reset points. A touch is detected when a bicycle wheel intersects a cylinder standing on the reset point with 4.0m diameter and 1.0m height.
’k’	check point	Checkpoints don’t constitute a physical obstacle, but bicycles can touch checkpoints. The sensitive area is the same as with reset points (a cylinder with 4.0m diameter and 1.0m height). Checkpoints have more data at the end of the map file (see below).
’f’	flat surface	Fills the tile horizontally at the layer height, has the same physical properties as a road.
’t’	elevated surface	Fills the tile horizontally at 2.5m above layer height, has the same physical properties as a road.
’u’, ’d’, ’l’, ’r’	ramps	Ramps ascending to one of four directions. Ramps have the same physical properties as roads. They’re rectangles that are at layer height on one edge of the tile, and at 2.5m above layer height on the opposing edge of the tile.
’u’	ascending to the north	It’s a rectangle that is at layer height at the south edge of TY, and at layer height + 2.5m at the north edge of TY+1.
’d’	ascending to the south	It’s a rectangle that is at layer height at the north edge of TY+1, and at layer height + 2.5m at the south edge of TY.
’l’	ascending to the west	It’s a rectangle that is at layer height at the east edge of TX+1, and at layer height + 2.5m at the west edge of TX.
’r’	ascending to the east	It’s a rectangle that is at layer height at the west edge of TX, and at layer height + 2.5m at the east edge of TX+1.
’_’, ’|’	planks	Planks have the same physical properties as roads. They’re rectangles that lay flat on the layer height. They connect an edge of a tile with the opposing edge, but they’re only 1.7m wide, leaving a (5.0-1.7)/2 = 1.65m wide gap on each side.
’_’	west-east plank	A plank connecting the west and east edges of a tile (centered at TY=0.5).
’|’	north-south plank	A plank connecting the north and south edges of a tile (centered at TX=0.5).
’o’	column	A physical obstacle. Columns are cylinders with 2.5m diameter and 2.5m height.

Mailboxes, checkpoints and resetpoints all have indexes. They’re all indexed, separately, beginning with 0, in the order of map file appearance (from TX=0 and up, then TY=0 and up).

After the last encoded map row, the map file ends with metadata on checkpoints - one line per checkpoint, in index order (so order of appearance in map).

Checkpoint lines consist of words, separated by spaces:

```
TX TY layer index next score name
```

- TX, TY are the tile position (redundant data).
- ’layer’ is the layer index, 0..7 (redundant data).
- ’index’ is the checkpoint index (redundant data).
- ’next’ is the index of the checkpoint next in the course, or `-1` if this checkpoint is at the end of a course (or is not part of a course at all).
- ’score’ is 1, 2, 3, or 4.
- ’name’ is the name of the checkpoint. May contain spaces, and may be "None" to indicate that the checkpoint has no name.

Geometry file

To aid with visualization and other algorithms, we provide a file (in the public file sharing area at http://server.ch24.org/public/) that contains a dump of the map geometry - someone may find this useful.

The file is a list of records, each record a line. Each line begins with a prefix character. Possible record types are these:

- `q` *x1 y1 z1 x2 y2 z2 x3 y3 z3 x4 y4 z4* - wall quad between the four vertices
- `s` *x1 y1 z1 x2 y2 z2 x3 y3 z3 x4 y4 z4* - slippery (grass) quad between the four vertices
- `d` *x1 y1 z1 x2 y2 z2 x3 y3 z3 x4 y4 z4* - death (lava) quad between the four vertices
- `k` *x y z score prevCP nextCP name* - checkpoint
- `r` *x y z* - resetpoint
- `m` *x y z score* - mailbox
- `o` *x y z radius height* - column (a cylinder of wall standing on the given point, radius always 1.25, height 2.5)

Protocol

Coordinate system

Units are meters. All angles are radians.

Control (player $\rightarrow$ server)

UDP packets, text based, rows separated by LF.

First row is a single 32 bit unsigned integer sequence number. Packet is ignored if the previously received packet had a >= sequence number. If no packets are received for 10 seconds, the internal counter is reset to 0 (the first packet sent by the player should have a sequence number of 1).

Following rows are commands, one command per row. Commands are composed of words, separated by spaces.

Bicycle control command summary:

command format	description
`<bicycle_id> <thrust> <steer> <weight>`	Set control input
`<bicycle_id> reset`	Move the bicycle to a nearby resetpoint in a standing position. (Uncontrolled, unbalanced bicycles will tip over.)
`<bicycle_id> paper <vx> <vy> <vz>`	Throw newspaper with a certain speed. Only one newspaper may be in the air at a time for a single bicycle (if there's an active newspaper, the throwing command is ignored).
`<bicycle_id> cam`	Focus the camera on the selected bicycle.

parameter	type	value range	description
bicycle_id	integer	1 .. 3	
thrust	float	-1 .. 1	controls the brakes (-1..0) and pedalling force (0..1). Any amount less than -1 for thrust will apply a small, constant reverse thrust.
steer	float	-1 .. 1	controls the front wheel. (left .. right)
weight	float	-1 .. 1	controls the center of mass. (left .. right)
vx,vy,vz	float	*	velocity in m/s. If the given velocity vector is greater than 10m/s, it will be scaled down to 10m/s (keeping its direction).

Control values are saved and used until the following command.

'steer' and 'weight' are inputs to a controller, they aren't immediately actualized.

Update (server → player)

UDP packets, text based, rows separated by LF. Words are separated by single spaces.

- First row is:

  ```
  <sequence_num> <recv_sequence_num> <errors>
  ```

`’sequence_num’` is a 32 bit unsigned integer sequence number incremented with each packet.

`’recv_sequence_num’` is the sequence number of the last received packet from the player.

`’errors’` is the number of protocol errors (increases when control packet couldn’t be processed, or was discarded).

- Subsequent rows are:

  ```
  B bicycle_id bx by bz fx fy fz roll steer weight dead
  ```

  `bicycle_id`: integer, 1 .. 3

  `bx, by, bz`: position of back wheel hub
  `fx, fy, fz`: position of front wheel hub

  `’roll’` is the angle of the bicycle seat above the back wheel from vertical (-PI/2: laying on the left side; PI/2 laying on the right side).

  `’steer’` is the angle of the front wheel to the rest of the bicycle (-PI/2: facing left; PI/2: facing right).

  `’weight’` is the current position of the center of mass (-1: left; 1: right).

  `’dead’` is 0 when the bicycle is alive, 1 when the bicycle is dead.

- For each active newspaper:

  ```
  N bicycle_id x y z
  ```

- Gamestate

  ```
  G bicycle_id newspapers checkpoint
  ```

  This is a game state update, sent every once in a while, and when the gamestate changes.

  `’newspapers’` is the remaining number of newspapers.
  `’checkpoint’` is the index of the last visited checkpoint, if a checkpoint path is being tracked, or -1 when not.

- Killed event

  ```
  K bicycle_id reason
  ```

  This message is sent once when a bicycle is killed. Reason is one of:
  - `HEAD` - person mass touched something.
  - `BUTT` - person mass touched by other person mass (headbutt)
  - `SQUISH` - person mass touched by a bicycle wheel
  - `WHACK` - person mass touched by newspaper thrown by other bicycle

    - `LAVA` - bicycle touched lava

- Newspaper got into mailbox event

  ```
  M bicycle_id index
  ```

  This message is sent once when a newspaper gets into a mailbox.
  `’index’` is the mailbox index.
  Gamestate will also be sent in the same packet.
  The event is sent even when the mailbox has already been filled in this hour (so no points are awarded).

- Resetpoint event

  ```
  R bicycle_id index
  ```

  This message is sent once when a resetpoint is touched.
  Gamestate will also be sent in the same packet.
  `’index’` is the resetpoint index.

- Checkpoint event

  ```
  C bicycle_id index name
  ```

  This message is sent once when a checkpoint is touched.
  Gamestate will also be sent in the same packet.
  `’index’` is the checkpoint index.
  `’name’` is the checkpoint name (may be empty, may contain spaces).

- Course completed event

  ```
  O bicycle_id index time
  ```

  This message is sent once when a course is completed.
  Gamestate will also be sent in the same packet.
  `’index’` is the checkpoint index (of the last checkpoint).
  `’time’` is the course completion time (seconds, floating point).

- Checkpoint timeout event

  ```
  T bicycle_id
  ```

  This message is sent once when 30 seconds elapse since the last checkpoint was touched (so the checkpoint chain is interrupted).
  Gamestate will also be sent in the same packet.

M. Dead-line

You managed to get the parcel service to pick up your box you have designed in the laser CNC task. The only problem remaining is how you get to the checkpoint where they pick you up, because zombies are following you again. If you walk fast enough, they can not catch up and will follow you in a long queue. If they catch you, they will bite you and you will become one of them. If you arrive at the checkpoint too late, you miss the pickup and die in the city; if you arrive too soon, you need to stop and wait and the zombies will bite you.

The only way out of this is finding a route that is long enough that you don't arrive too soon. The horde of zombies is following you like an endless stream, so you can't visit the same point on the map twice.

http://wiki.openstreetmap.org/wiki/File:Public-images-osm_logo.png

Given a map downloaded from OpenStreetMap (http://osm.org), two points, and a constant *M*, find a route with between the two points close to *M* in length but not exceeding it, without visiting a node twice.

Input

The map

The map itself is given in **OpenStreetMap XML** format. The most important entities are <*node*> and <*way*>. (<*relation*> entities should be completely ignored.)

A <*node*> has the following main attributes:

- *id*: a unique 64-bit signed integer identifier
- *lat*: latitude coordinate, in degrees
- *lon*: longitude coordinate, in degrees

A <*way*> has the following kinds of children: <*nd*> and <*tag*>. A <*nd*> element describes a reference to a node through the *ref* attribute, which contains the unique identifier of that node.

Some of the <*nd*> elements may refer to nonexistent nodes via invalid id's -- these <*nd*> elements should be ignored completely. This may result in ways with one or zero nodes, these ways should be ignored as well.

A <*tag*> element describes a key-value pair through the *k* (key) and *v* (value) attributes. Streets correspond to those ways which have a tag with key *highway*. If a street has a tag with key *oneway* and value *yes*, then it can only be used in the direction defined by the listing order of its node references. If a street has a tag with key *oneway* and value *-1*, then it can only be used in the opposite direction. All other streets are bidirectional. No other key/value pairs should be taken into account.

The quests

Each quest is given in a text file which consists of three whitespace-separated numbers *A B M*.

A is the unique identifier of the starting node and *B* is the unique identifier of the finishing node. *M* is the maximum allowed length of the route, in meters. You should use the following formula to calculate the distance of two geographical locations (in meters), substituting the latitudes and longitudes converted to radians:

```
acos(sin(this.Latitude) * sin(other.Latitude) +
     cos(this.Latitude) * cos(other.Latitude) * cos(other.Longitude - this.Longitude)) * R
```

where *R* equals of the radius of the planet, defined as exactly *6371000*.

Output

For each quest, you have to produce an output file containing whitespace-separated integers. The file must contain the number of visited nodes *N*, followed by exactly *N* integers: the unique identifiers of the nodes, in visiting order. A valid submission describes a route from *A* to *B* whose total length is at most *M*.

Scoring

This is a scaled problem. If the total length of the route is *L* then the evaluated score is

```
S := M - L
```

The scaled real score is

```
SCORE := round(100 * (1 - sqrt(1 - Smin/S)))
```

Where *Smin* is the best submission so far.

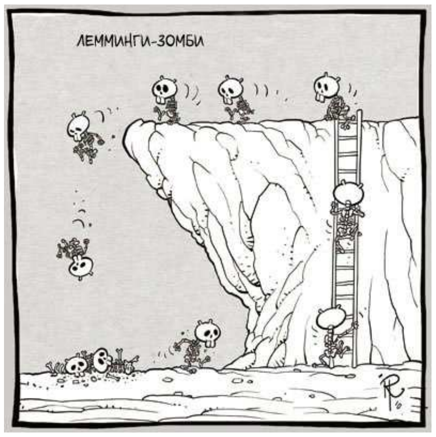
Example input	Example output
`244926905 541151835 1600`	`4 244926905 497096570 527360282 541151835`

N. Zombings

The zombie brain is not famous for its complexity - this makes zombies easy to simulate. Simulating zombies helps you learn how they are best controlled.

We've prepared a simulation for you. You need to prove your ability by controlling your zombies so that more zombies will reach your goal.

The current state of the map and zombies are periodically sent out in UDP broadcast packets. Controlling your zombies is possible using a TCP connection.



http://todamax.kicks-ass.net/2010/zombie-lemminge/

All of the competing teams' zombies move around on the same map. Each team has a separate starting position, where their zombies spawn, but teleports also lead to these starting positions. Each team also has a goal, where they must get as many zombies as they can, to demonstrate your skill in controlling them. Although each player can only control their own zombies, they gain points for any zombies they get to their goal.

Scoring

A simulation game is run every hour with the following schedule: the server is listening to connections and broadcasting the map from 9:40, 10:40, 11:40,.. etc and the first spawn at 9:45, 10:45, 11:45,.. etc.

The hourly maximum contest score is 3500 divided by 24, multiplied by 0.5 for the first hour, 1.5 for the last hour, and linearly scaled in between. The team with the highest amount of points gets this amount, and the other teams get their score linearly based on their points.

Map

The zombies move around on a 2D map, represented by a `width` by `height` matrix. Each position in the map can contain one of the following:

- Empty space.
- Wall.
- Indestructible wall
- Teleport (indestructible)
- Goal for one of the players (indestructible)

Also, there can be several zombies on a position, but they don't form part of the map. Building on a position means replacing space with wall, destroying a position means replacing wall with space. These actions can be applied to indestructible positions (it is not an error to do so), but they have no effect.

Zombie status

At any moment, a zombie's status consists of the following variables:

- Position: x and y coordinates, where the top left corner of the map is (0, 0).
- Direction: facing left or right.
- State: what the zombie is doing at the moment. Can be walking, falling or one of the skills described below. They are referred to as the following enum values in the network protocol:

state	value
walking	0
falling	1
stopping	2
digging	3
shoveling	4
spiral stairs	5
bridging	6

- State time: a wrapped around counter for how long the zombie has been in its current state. It's significance is described for each state separately.

Each turn of game time happens in 4 phases:

1. Build phase: this is when all zombies modify the map, create or destroy. Any pixel of the map that gets built in and destroyed in the same turn will be destroyed.
2. Movement phase: zombies move, using the already modified map from the first phase. This is also when zombies can die, teleport or reach a goal.
3. State time is increased by one, except if in walking state, then as specified below. If it reaches the wrap around for the state, it resets to 0.
4. Command phase: this is when the commands sent to the zombies take effect, all zombies that got a command change state, and their state time resets to 0. They will act according to their new state from the next turn.

Next is the specification of the zombies' behavious in each of these phases, depending on their current state.

Legend for ASCII drawings

```
S = Current position of zombie, facing right
+ = Next position for zombie
```

```
. = Empty space
# = Wall
! = Target for buiding or destruction
0..9 = Target after N turns
```

Build phase

Walking	Nothing	

Digging	If state time is 0, the zombie tries to dig down. In this state, state time wraps around at 8, so this means every 8th turn, including the first turn in this state. If all of the target spots are space or wall, they get destroyed, if any of them are indestructible, none of them change. The target positions are the ones below the zombie, and the next four x coordinates in each direction.	`........S........`
		`####!!!!!!!!!####`
		`#################`

Shoveling	If state time is 0, the zombie tries to dig horizontally. In this state, state time wraps around at 8, so this means every 8th turn, including the first turn in this state. If all of the target spots are space or wall, they get destroyed, if any of them are indestructible, none of them change. The target positions are the ones in front of the zombie (the next column in the direction it is facing), from his level to 10 above.	`.........########`
		`.........!#######`
		`.........!#######`
		`.........!#######`
		`.........!#######`
		`.........!#######`
		`.........!#######`
		`.........!#######`
		`.........!#######`
		`.........!#######`
		`.........!#######`
		`........S!#######`
		`#################`
		`#################`

Stopping	Stopping zombies only have a lifetime of 8 turns. During this time, they crystallize into a wall. When their state time is 0, 2, 4, 6 and 8, they build subsequent layers of a wall out, starting with the column they are in at time 0. In the current column they build into the rows from 3 below them to 10 above them. In the figure to the right, you can see which positions get built on in which round, compared to the position of the zombie (the S in the middle).	`....864202468....`
		`....864202468....`
		`....864202468....`
		`....864202468....`
		`....864202468....`
		`....864202468....`
		`....864202468....`
		`....864202468....`
		`....864202468....`
		`....864202468....`
		`....8642`**`S`**`2468....`
		`####864202468####`
		`####864202468####`
		`####864202468####`
		`#################`
		`#################`

Spiral stairs	When building spiral stairs, state time wraps around at 16, but the behaviour is different when state time < 13, and after. When state time < 13, and state time is even (so when it's 0, 2, 4, 6, 8, 10 or 12), the zombie builds into the spaces in front of it by 1 and to in its direction. (These are marked 1-7 in the figure, as it moves up as well.) Then, in round 14 it builds into the space above himself, into the space in front of that one, and builds up a wall of 15 spaces starting from the space 2 from the one above him in his direction.	`.................`
		`...........8.....`
		`...........8.....`
		`...........8.....`
		`...........8.....`
		`...........8.....`
		`...........8.....`
		`...........8.....`
		`...........8.....`
		`...........8.....`
		`...........8.....`
		`...........8.....`
		`...........8.....`
		`...........8.....`
		`...........8.....`
		`.........888.....`
		`..........77.....`
		`.........66......`
		`........55.......`
		`.......44........`
		`......33.........`
		`.....22..........`
		`...S11...........`
		`#################`
		`#################`

Bridging	If state time is 0, the zombie builds one more piece of his bridge. In this state, state time wraps around at 8, so this means every 8th turn, including the first round in this state. It builds into the spaces in the column in front of it, from the one below, to a depth of 4.	`........S........`
		`#########!.......`
		`#########!.......`
		`#########!.......`
		`#########!.......`
		`#########........`

Falling	Nothing	

Movement phase

In this phase it is possible for the zombie to change state automatically, as described in detail below. If that happens, state time is reset to 0. If the state changes to falling, it the zombie will already fall this turn, so state time will be 1 by the end of the turn. If it changes to walking, it the zombie will only start moving next turn, state time will be 0 at the end of this one.

Walking	First, the zombie looks at the seven spots in front of him, from his level up to 6 above, and steps into the first empty one. If none of them is empty, it changes direction without moving. If it successfully stepped into the first spot (the one directly in front of him), it may descend at most 3 coordinates below, as long as the space under him is empty, without changing to falling.
	
	If it successfully stepped, the state time is set to zero. If it didn't (it changed direction), it keeps counting. If the state time is 4, they die (so they can't survive long when being stuck in a wall).
		`.................`
		`.........7.......`
		`.........6.......`
		`.........5.......`
		`.........4.......`
		`.........3.......`
		`.........2.......`
		`........S1.......`
		`#################`
		`#################`

Digging	1. If state time is 0, and the zombie successfully dug, it moves down one. (That space is always empty, because destruction overrides building if it happens in the same round.)
	2. If state time is 0, but the zombie couldn't dig successfully, because of an indestructible block, it changes to walking, and will start moving in the next turn.
		`........S........`
		`####!!!!+!!!!####`
		`#################`

Shoveling	1. If state time is 0, and the zombie successfully shoveled, it moves in the direction it's working. (That space is always empty, because destruction overrides building if it happens in the same round.)
	2. If state time is 0, but the zombie couldn't shovel successfully, because of an indestructible block, it changes to walking, changes direction, and will start moving in the next turn.
	3. If all the target spots (the next column, from his level to 10 above) are empty it concludes that it's work is done, and changes to walking, and will start moving in the next turn.
		`.........########`
		`.........!#######`
		`.........!#######`
		`.........!#######`
		`.........!#######`
		`.........!#######`
		`.........!#######`
		`.........!#######`
		`.........!#######`
		`.........!#######`
		`.........!#######`
		`........S+#######`
		`#################`
		`#################`

Stopping	Stopping zombies never move, don't fall, and can't be changed to walking by a command either. If their state time is 8, they die.	`....864202468....`
		`....864202468....`
		`....864202468....`
		`....864202468....`
		`....864202468....`
		`....864202468....`
		`....864202468....`
		`....864202468....`
		`....864202468....`
		`....864202468....`
		`....8642S2468....`
		`####864202468####`
		`####864202468####`
		`####864202468####`
		`#################`
		`#################`

Spiral stairs	If state time is 1, 3, 5, 7, 9, 11 or 15 (note: not on 13!), the zombie tries to move. If state time < 13, it’s target is one up and one forward from him, if state time is 15, it is 2 up and one forward from him. If that space is not empty, it changes to walking, changes direction, and will start walking in the next turn. If it is empty, it moves there. If state time is 15 and the movement was successful, it changes direction as well.	....8..... ....8..... ....8..... ....8..... ....8..... ....8..... ....8..... ....8..... ....8..... ....8..... ....8..... ....8..... ....8..... ....+8..... ....888..... ....+77..... ....+66...... ....+55....... ......+44........ .....+33......... ....+22.......... ...S11........... ################# #################
Bridging	If state time is 4, it tries to move one ahead. If that space is not empty, it changes to walking, changes direction, and will start walking in the next turn.	........S+....... #########!....... #########!....... #########!....... #########!....... #########........

After normal movement, if the space under the zombie is empty, it changes to falling mode, and already falls in this turn. (Even if it changed state in this turn to something else.)

Falling happens like this:

1. If the space under the zombie is not empty, it changes back to walking.
2. If state time is 11, it realizes the actual depth and dies of horror.
3. It moves down at most 3 spaces as long as those positions are empty.

After this, the following checks are done:

- If the space below the zombie is a teleport, it moves the zombie to a different location. (Note that this doesn’t change state, if the zombie was falling, its state time remains!)
- If the space below the zombie is a goal space, the zombie disappears ("dies"), and the player that the space belongs to gets a point.

Commands

The commands to zombies consist of the number of the zombie it should be applied to, and the state they should change into. But there are restrictions on what changes are accepted. The restrictions are the following:

- You can never give falling as a command.
- You can never give a stopping zombie a command.
- Falling zombies can only be given stopping as a command.
- Zombies that are at most 50 Manhattan distance from a starting point can't be given commands.

It is however allowed to give walking as a command (except to a falling or stopping zombie), to stop whatever the zombie is doing at the moment.

Whenever a command is successfully given, state time resets to 0.

Summary

The following table summarizes the possible transitions between the different states. "auto" means the transition can happen automatically, "cmd" means it can happen by commanding.

FROM\TO	walking	falling	stopping	digging	shoveling	spiral stairs	bridging
walking	-	auto	cmd	cmd	cmd	cmd	cmd
falling	auto	-	cmd	-	-	-	-
stopping	-	-	-	-	-	-	-
digging	auto (indestructible) / cmd	auto	cmd	cmd	cmd	cmd	cmd
shoveling	auto (indestructible or empty) / cmd	auto	cmd	cmd	cmd	cmd	cmd
spiral stairs	auto / cmd	auto	cmd	cmd	cmd	cmd	cmd
bridging	auto / cmd	auto	cmd	cmd	cmd	cmd	cmd

UDP broadcast protocol

The server broadcasts binary udp messages continously. Each datagram message consist of one or more message chunks. Message chunks are identified by their first byte according to the table below. The first message chunk is always a tick (starts with a T). All numbers are unsigned integers in network byteorder (MSB first). `{n:name}` means an n-byte long unsigned integer referenced with the given name in the description. Other characters in the message chunk specification are bytes verbatim in the input with ASCII coding.

message chunk	description
`T{4:tick}`	following chunks correspond to the state after the `{tick}`th tick
`M{2:x}{2:y}{2:length}{data}`	This chunk contains map data ● {data} has {length} bytes ● after {tick}th tick, the {x}..{x}+{length}-1 of {y}th row of the map is {data} ● {data} consists of bytes: ○ '.': empty ○ '#': wall ○ '%': indestructible wall ○ 'a'-'z': goal of 0-25th teams ○ '0'-'4': goal of 26-30th teams ○ 'A'-'Z': teleport to 0-25th teams' starting point ○ '5'-'9': teleport to 26-30th teams' starting point

`Z{2:zombienum}{2:x}{2:y}{1:statewithtime}`
`S{2:zombienum}{2:x}{2:y}{1:statewithtime}`
	These chunks carry information about the zombies
	● zombie {zombienum} is at {x},{y} coordinates after {tick}th tick
	● facing left if Z, right if S
	● {statewithtime} has two 4-bit parts:
	○ high 4 bits is state time
	○ low 4 bits is state, can be 0-6 as described in the beginning

`C{2:zombienum}{1:state}`	These chunks broadcast all the commands that were given to the zombies, at the time of their execution, if they were successful. {zombienum}th zombie was commanded to do {state} after {tick}th tick

TCP response (server → client)

Control commands are sent using a text based TCP protocol. When a client connects to the server, it sends back various information in text format, each message on separate line:

message	description
`allplayernum {playernum}`	number of players
`yourplayernum {playernum}`	you are player {playernum}
`mapwidth {width}`	width of map
`mapheight {height}`	height of map
`tickfreq {tickfreq}`	number of ticks in a second
`ticknum {ticks}`	number of ticks in the full game
`zombienum {zombienum}`	number of zombies for each player
`firstspawn {firstspawn}`	first zombies spawn at tick {firstspawn}
`spawntime {spawntime}`	time between spawning of zombies
`spawnpoint {playernum} {startx} {starty}`	{playernum}th player's zombies spawn at {startx},{starty}
`skillnum {state} {num}`	each player can use {state} at most {num} times

Unsuccessful commands also get error messages through the tcp connection, successful commands get notification through the udp broadcast.

TCP control (client → server)

The client can send the following command to the server. Each command must be on a separate line.

message	description
`{zombienum} {state} {tick}`	Command your {zombienum}th zombie to change to state {state} (from the table above) after the {tick}th tick. {tick} must be in the future for the server, and the command must conform to the restrictions described in the "Commands" section, otherwise it will fail.