Challenge 24

11th BME International 24-hour Programming Contest

http://ch24.org

Morgan Stanley FORNAX ExxonMobil Google T iGO

Contest

Welcome to the 11th BME International 24-hour Programming Contest!

Rules

The contest starts at 2011-04-30 09:00 CEST and ends at 2011-05-01 09:00 CEST.

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

Submission site

The same submission system will be used as during the Electronic Contest. It will be available through the https://10.0.0.1/sub url.

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	Scaling	Scheduled
A	Yes	No	Yes	Yes	No	No
B	Yes	No	Yes	Yes	No	No
C	Yes	No	Yes	Yes	No	No
D	Yes	No	Yes	Yes	No	No
E	No	Yes	No	Yes	Yes	No
F	No	Yes	No	No	No	Yes
G	No	Yes	Yes	No	No	No
H	No	Yes	No	No	No	Yes
I	Yes	Yes	No	Yes	No	Yes
J	Yes	Yes	No	Yes	No	Yes
K	No	Yes	No	No	No	No
L	No	Yes	No	No	No	Yes

- 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 worth 70% of the score at the begining.

- Penalty for wrong answer: Wrong answer gets -5 points (different value may be specified explicitely in the task description).
- 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.)
- Scheduled: Must be submitted or solved at a scheduled time (cannot be postponed later).

Important:

There are a couple of problems which need immediate attention, namely: GH, IJ

Contact

Important announcements will be made on IRC so please join the `#announcements` channel on the irc server at 10.0.0.1 (using the default port, 6667). The log files of this channel will be published on the https://10.0.0.1/announcements url.

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

There will be separate channels for task related problems as well: `#A`, `#B`, `#C`, `#DE`, `#F`, `#GH`, `#IJ`, `#K`, `#L`.

A. Garbage (1000 points)

We have N cities (numbered 1-N) and bidirectional roads between them. Each road has a length. Some cities have recycle plants which are able to recycle some types of garbage. There are 26 types of garbage (a-z).

The task is to recycle a garbage-string while we have to minimize the total moving cost. The moving cost of a garbage-string is equal to the length of the road (independently from the length of the garbage). We can cut a string into several smaller strings anywhere, but multiple strings cannot be attached to form one longer. We can recycle a string at a city if it contains only the kind of gargabe that can be recycled there.

Input

Each input contains multiple test cases, each test case looks like this:

First line: N M L S (number of cities, number of roads, length of initial garbage string, starting city of the garbage)

Next N lines: k_i t_i (t_i is a k_i characters long string specifying the types of garbages that the ith city can recycle)

Next M lines: a_j b_j w_j (road between cities a and b with length w)

Last line: g (the garbage string (L characters long))

After the last test case, there will be "0 0 0 0" in the last line.

Output

For each test case output the minimal total cost on a separate line, or -1 if the test case is not solvable.

Sample input

```
3 2 6 1
1 a
1 b
2 cb
1 2 1
2 3 2
abbcab
0 0 0 0
```

Sample output

```
4
```

B. Layout (1000 points)

Rabbit has organized an elaborate peer-to-peer network in the forest with his friends-and-relations. Although by some magic the network works perfectly, it turned out to be incredibly intricate, and Rabbit decided to draw a huge Map to help him oversee maintenance operations. The Map should contain Nodes, and Edges between them as straight lines, in a way that no Edge intersects another one.

However hard he tried, Rabbit couldn't draw a proper Map - somehow some Edges always intersected. One of his relations then came up with a clever plan - what if it was allowed for an Edge to go out the side of the map and come in from the other side? (In the language of the forest, this was called a "Toroidal Map".) It turns out this makes it possible to draw the Map - but it's a new and unexpected task for Rabbit, and he asks You for help.

Input and Output

- the first line of the input is two integers - number of Nodes and number of Edges
- then for each Edge, a line follows with two integers - N1 and N2. An edge exists between Nodes N1 and N2 (Nodes are numbered from 0).

The Map will be drawn as 1000x1000 units big. Your output should look like this:

- first, one line per Node, in order, with two integers each - X and Y, specifying the position of the Node on the map. Neither X or Y can be on an edge - so the minimum is 1, the maximum is 999 for both coordinates. Two Nodes cannot occupy the same position.
- then, for each Edge (in order of the input file) a line with two integers: SX and SY.

The Edges will be drawn as straight lines:

```
N1X, N1Y -> N2X + SX*1000, N2Y + SY*1000
```

In short, SX and SY specify the "screen offset". Both SX and SY must be between -2 and +2 (inclusive). If SX and SY are 0, the Edge doesn't cross the side of the map. For example, if SX is -1 and SY is 0, then the Edge will start at N1, cross the left side of the map, come in from the right side and end at N2.

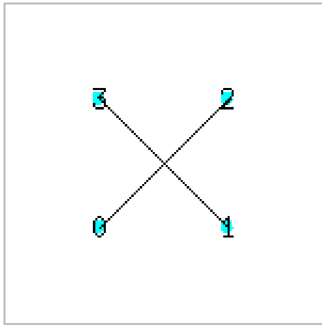
Neither Edges, nor any of their segments may intersect.

Example

For example, take this simple network (note that it's of course possible to draw this network on an ordinary Map):

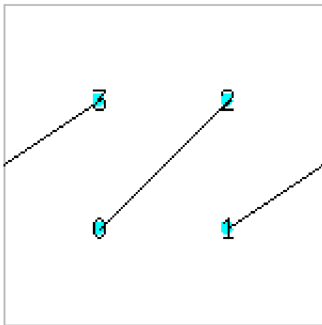
```
4 2
0 2
1 3
```

If you lay out the Nodes in a counter-clockwise order, drawing both Edges locally will result in a bad solution:



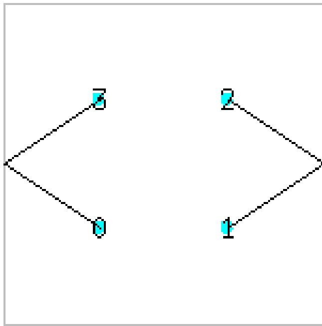

```
300 300
700 300
700 700
300 700
0 0
0 0
```

This can be fixed by having one of the Edges cross the side of the Map:




```
300 300
700 300
700 700
300 700
0 0
1 0
```

Here is another bad solution, where the Edge segments don't intersect, but the Edges themselves do:




```
300 300
700 300
700 700
300 700
-1 0
1 0
```

C. Image compress (1000 points)

A new image compression algorithm is based on repeated rectangular areas on the image.

You need to write a subroutine for the encoder that finds the largest such area pair on a given image.

Input

The input is an NxN size 1 bit per pixel png image (black and white pixels only).

Output

The output should contain six integers separated by space, representing a largest repeated rectangle pair.

format: W H X1 Y1 X2 Y2

where W,H are the width and height of the rectangles and X1,Y1 and X2,Y2 are the two top left corners.

All the pixels in the (X1,Y1,W,H) rectangle must match to the ones in the (X2,Y2,W,H) rectangle and (X1,Y1) != (X2,Y2).

W*H must be the maximum possible. The rectanges must be on the image.

The coordinates of the top left corner of the image is (0,0).

Example

Assume that the input is a 8x8 pixel entirely black image. Then a possible solution is:

```
8 7 0 0 0 1
```

D-E. Stars

D. Stars - Navigation (1000 points)

You are working on the navigation system of a satellite.

The satellite is on a known orbit around Earth so its position is known. However sometimes its orientation is not known so it must be determined in order to get the satellite in an operational state.

The satellite has an onboard camera that can take a grayscale picture.

Your task is to develop an algorithm that can determine the orientation from a single image taken by the onboard camera. An accurate star catalog is available and you can assume that objects of our solar system won't be visible on the image, only the stars listed in the catalog.

Catalog

Each star is given by four numbers in the catalog: ID, RA, DEC, MAG. ID is an identifier, RA and DEC are spherical coordinates and MAG is the magnitude of the star.

The RA, DEC coordinate system can be thought of as a projection of the Earth bound longitude, latitude system to the sky.

The magnitude of the star can be modelled by `-const1*log(brightness/const2)` (a brighter star has smaller MAG).

For more details see the `stars.txt` file.

Input

The input is a grayscale, 1000x1000 pixel png image.

The field of view of the camera is known to be 20 degrees, so 1000 pixel distance on the image means approximately 20 degrees difference in direction.

Output

The output should be two 3d unit length vectors: the first line should hold the direction where the camera looks, the second line is an orthogonal vector that is the upward direction on the image.

The 3d coordinate system oriented so that it is easy to convert between the spherical coordinate system and the 3d one:

- RA=0,DEC=0 is the X direction, so (1,0,0) in the 3d system
- RA=pi/2,DEC=0 is the Y direction, so (0,1,0) in the 3d system
- RA=0,DEC=pi/2 is the Z direction, so (0,0,1) in the 3d system

The direction of the output vectors must be more precise than 0.01 radian.

Samples

Three sample images are provided with reference outputs, so you can calibrate your algorithm: the first one looks at Sirius, the second one looks at (parts of) the Ursa Major constellation, the third one looks to the north.

E. Stars - Galaxy designer (1000 points)

It turns out it is fairly complicated to determine the directions for navigation in our solar system.

Try to design a better star distribution in a way that navigation is easy (assuming you only want to navigate in a central solar system).

Note that the Galaxy Building Project has limited budget so you should use as few stars as possible.

You may use 1000 stars at most, with magnitude in the range [0,10]. Prepare a star catalog in the same format as in the previous task (ID should be an integer in [1,1000]).

Once you submitted a star catalog, shortly you will get 10 images taken in your newly designed galaxy.

To complete this task successfully you have to send back the orientation for each image similarly to the previous task, but with a short timelimit.

Protocol

You have to submit your solutions to the Galaxy designer task to a special submission server that uses a binary protocol. During a successful submission, using a single TCP connection, you will:

1. Send a file: your star catalog
2. Receive 10 files: the test images (in **png** format)
3. Send a file: the 10 orientations (20 vectors on separate lines)
4. Receive a score.

To send a file, you send its length as a 4-byte integer, then the file itself. To receive one, you first read its length as a 4-byte integer, then the file itself. The score is also sent as a 4-byte integer. **All integers** (so the sent and received lengths, and the received score) **are in network byte order** (see `htonl()`, `ntohl()` or similar).

If there is any error during this procedure (such as a protocol error, or an internal error), the server will just close the connection. If the cause is unclear, please ask an organizer (and tell them the exact time of the submission).

After you send your star catalog, the server will render 10 test images (this may take a bit of time, up to a minute), and send them back to you. After the server starts sending the images, you have 30 seconds to calculate the orientations and send them back as a single file (concatenated in order).

Scoring

This is a scaled problem. An accepted submission is worth

$$SCORE = 200 + 800^{(5 - N/M)/4}$$

where N is the number of entries in the star catalog and M is the best submission so far. This means a good submission is worth 200 points, but there is a large bonus if your catalog is short compared to other teams.

For wrong answers the penalty is -5 points.

F. Fungus (3000 points)

The Game

So it happened that one of the organizers accidentally kicked a pile of discarded petri dishes. The virulent strains of fungi that before occupied the fetid creases of his foot have been released into the nutritious gelatin. They are now engaged in a biological fight to the death.

Your team now has the opportunity to control one of these strains and prove its genetic superiority.

Rules

The map is built from rectangular cells with X;Y coordinates (0 <= X < W, 0 <= Y < H). On each cell there may be a stack of food or a stack of fungi of a team (different teams have different type of fungi). A team may command fungi to move to an adjecent cell.

- when N fungi move from a cell to another cell occupied by the same team, N is substracted from the first cell and added to the other cell’s height.
- when N fungi move from a cell to a cell occupied by M food, the food is converted to fungi of the same type, resulting totally in N+M fungi.
- when N fungi of team A move to a cell of M fungi of team B they fight. if N > M, result is N-M fungi of team A; if N < M, result is M-N fungi of team B; if N == M, result is empty cell.

For each tick, moves of a single team are processed. 30 ticks (one for each team) forms a round. Teams may place commands any time, but only the last 500 commands will be considered (in case of multiple commands for the same cell, only the last one will run). Invalid commands are ignored, all valid commands run in "parallel" (one set of fungi will move only once).

After each such move, the new standing is broadcasted to all teams (only diff).

Before the beginning of the game, the original map is published.

Scores are calculated at end of each game for every team in the following way:

- connected components of the team’s fungi are calculated
- for each connected component, its score is calculated:
  - the convex hull of the midpoints of the cells in the component is calculated
  - each cell within the hull (whose midpoint is inside or on the boundary of the hull) is worth:
    - cell occupied by the team: 2
    - cell not occupied by any team: 1
    - cell occupied by another team: 0
- team’s score is the sum of their components’ score
- teams are ranked based on these scores
- if a team is ranked *n*-th in the *k-th* game (see schedule) then it gets round((30-*n*)**k**3000/8700) points

Map format

At the beginning of each game, a PNG image is published on web at https://10.0.0.1/fungus. The size of the PNG image determines W and H. The top-left corner of the image has coordinates 0;0. The top-right corner of the image has coordinates W-1;0. Cells outside the map have RGB color 0,0,0 (black). Initially there is no cell with more than 15 fungi or more than 15 food on it. Cells inside the map have two types:

- food with height h has gray color with (R=G=B=floor((16-h)/16*255)
- fungi with height h has color (R=floor(A*(16-h)/16*255), G=floor(B*(16-h)/16*255), B=floor(C*(16-h)/16*255)) where A,B,C is the reference color of the team

Team number	Team name	Reference color (A,B,C)
1	rusty	1,0,0
2	Grotzsch_Men	1,0.33,0
3	Sparrows24	1,0.67,0
4	The_Teddyborg	1,1,0
5	SRM	0.67,1,0
6	Saratov.SU2.Retired	0.33,1,0
7	BasicInstincts	0,1,0
8	UPC-Unflapipes	0,1,0.33
9	DrinkLess	0.33,1,0.33
10	croSharks	0.67,1,0.33
11	OrelSTU	1,1,0.33
12	pda	1,0.67,0.33
13	UPC-Siesqueva	1,0.33,0.33
14	Monkey_Island	1,0.33,0.67
15	ETs	1,0.33,1
16	Room_101	0.67,0.33,1
17	Eventually_almost_surely_correct	0.33,0.33,1
18	O.o	0.33,0.67,1
19	WarsawEaters	0.33,1,1
20	uw3000	0.33,1,0.67

21	funny-noise	0,1,0.67
22	Scorpions	0,1,1
23	Nemterminisztikus_fogoritmus	0,0.67,1
24	DTA	0,0.33,1
25	UPC-Reisub	0,0,1
26	Jackhammer	0.33,0,1
27	Raf	0.67,0,1
28	balloonsRus	1,0,1
29	groundwater	1,0,0.67
30	WeKings_SB_Forever	1,0,0.33

Communication protocol

Each command consists of coordinates of a cell, 4 numbers of fungi to move in each direction. Each command is in a new line, the six numbers are separated by spaces. For example:

```
6 3 14 0 1 3\n
```

this means: from cell 6;3, move 14 fungi to north, 0 to east, 1 fungus to south and 3 to west. If there are not enough fungi on the cell to do all 4 operations, none of the operations will be done and the command will be considered invalid. If any of the 4 operation is invalid then the whole command is invalid too.

The server may send the following:

- **START** *sec*

The game starts in *sec* seconds.

- **ROUND** *now*

The *now*-th round is starting.

- **TICK** *teamnum*

Team with number *teamnum* is coming.

- *x y owner height*

The cell with x;y coordinates has a stack of *height* fungi of team *owner*. If *owner* == 0 then it has a stack of *height* food.

Schedule

Each game has 500 rounds, each tick lasts 0.2 sec, so a game lasts totally 500*30*0.2 sec = 50 min. The server is started and the initial map is published at every hour. Five minutes later the game begins. It ends at 55 minutes after the hour. Then the score is calculated and there is a 5 minute break before the next server is started.

G-H. Dogfight

Problem G. and H. are coupled. To be succesful in H., G. needs to be solved early in the contest. For more details, please read the *scoring* section below for task H.

G. Dogfight - simulator (500 points)

Introduction

In this network game, 2 cars are playing dogfight. Rules are the following:

- two cars in a fight, each controlled by a team or by AI
- there is a minimal speed - if a car goes slower, it loses the battle
- cars leave trace where they pass by
- cars may not cross any of those traces
- cars may not cross the boundaries (so called 'environment')
- the server updates car A, then car B, then car A again, etc; a snapshot of the traces is sent after each update
- players may change their command for their next turn any time
- command is an acceleration with an upper limit
- all numerics are integers in this game

The player who survives longer wins. If both players survive until a specific timeout, the game ends in a tie.

Simulation

Each player has a current position *x*, *y* and a current velocity *vx*, *vy* where *vx*^2 + *vy*^2 >= *VMIN*^2. Car position is controlled by player supplied acceleration *ax*, *ay*, where *ax*^2 + *ay*^2 <= *AMAX*^2.

Updating a car position is done by the following calculation:

```
check(ax, ay)
vx' = vx + ax
vy' = vy + ay
check(vx', vy')
x' = x + vx'
y' = y + vy'
check(x', y')
```

The next position is connected to the previous one with a trace vector:

```
line(x, y, x', y')
```

The checks verify the acceleration, velocity and trace crossing constraints. If any of the checks fail or the new vector intersects with a previous trace vector then the player loses. Otherwise the new vector is added to the global list of traces and should not be crossed.

Initial condition (x0, y0, vx0, vy0) is sent to the players as part of the protocol.

Protocol

Messages from the server

Server messages are all single line text messages, except for the snapshot message, which contains multiple lines. \n (decimal 10 ASCII) is used to terminate lines. In the following table **bold** means string literal, *italic* means parameter.

syntax	sent to passive players?	description
game *len*	yes	Game status: *len* battles are left before your next battle in this game. If *len* is 0, you need to play in this battle, if negative, you have already played in this game (or in other words in this hour), so you shouldn't expect to play more battles until the next game (next hour).
turn *whose*	no	Sent when a new turn started, before car states are updated. *whose* is **you** or **enemy**.
end *winner*	no	Sent when a battle ends. *winner* has the same syntax as *whose* in **turn**, or **tie** on tie.
envseg *x1 y1 x2 y2*	no	A line segment of the environment
carpos *whose x y*	no	Current position of a car. *whose* is the same as in message **turn**. Position of enemy car is sent only before the first turn.
carvel *whose vx vy*	no	Initial velocity of a car. *whose* is the same as in message **turn**. Velocity of enemy car is sent only before the first turn.
snapshot start *numlines*	no	Sending a base64 encoded JPEG of the current standing. *numlines* lines are following, then a **snapshot end** message. Note: JPEG properties and image characteristics are expected to be different between the simulator and the actual device. Line width and color may change on the physical device during the contest.
snapshot end	no	End of snapshot.

Upon connection, commands are sent in the following order:

1. **game**
2. **envseg**
3. **carvel**
4. **carpos**

5. (if car’s just crashed: **end**)
6. **snapshot start** (for active players only)
7. (base64 JPEG data; for active players only)
8. **snapshot end** (for active players only)
9. (delay)
10. **turn** (for active players only)
11. (loop: go back to step "carpos")

During normal operation the same sequence is used, except for the **game** message that is sent again only when a new battle is started (or if the client reconnects). There is also a short delay between **snapshot end** and **turn**. When a player crosses (or touches) and existing line, the server sends **end** instead of **turn** and a new battle starts shortly. Vehicle position for the player is recalculated right after sending the turn command using the last ax;ay data sent by the player. If no data sent since last update, 0;0 is assumed.

Commands from the client

Command is a single line terminated by a newline (\n, decimal 10 ASCII). The server accepts commands any time, but will execute only the last one in the player’s turn. If there is no command specified (because the player is not connected, not sending commands in time or at all), command ax=0, ay=0 is assumed.

syntax	description
acc *ax ay*	Set acceleration control for the next turn.

Task

The simulator is provided for the teams to have instant access to a sandbox where they can train their AIs any time, without waiting for the hardware. Furthermore teams are initially ordered and paired for task H (dogfight - plotter) according to their performance in the simulator during the first few hours of the contest.

Players can connect the simulator any time, and the server will allocate a new AI instance to the player and a battle starts immediately. Each team can have only one connection at a time, upon a new connection, any existing connection of the team is closed. The server logs the outcome of simulated battles; when task H starts, this information will be used.

Scoring

The first time a team defeats the AI in the simulator, the team gets 500 points.

H. Dogfight - plotter (2500 points)

Task

This round takes place on an actual old-style pen plotter, traces are digitized by an IP camera. The server is started two hours after the contest starts.

Players should be connected to the server all the time. There is one game per hour. Each game consists at most 15 battles. The server will pair up two players (of all 30 possible players) for a battle; those two players are marked active until the end of the battle. All other players are passive. Some messages are sent only to active players, other messages are sent to all players. Commands can be sent only by the two active players.

Scoring

Due to the limited availability of the hardware, the scoring scheme of these tasks is quite complex to allow the best team to score the most. All in all, there are only two things that really matter:

- in the soft period, beat the AI earlier than other teams
- in the hard period, beat all other teams, winning each battle you play

The 24 hours period is split into a soft and a hard fight period. In the soft period, the hardware is not running and teams are playing against an AI in the simulator, all teams in parallel. The simulator is available during the whole contest, but beating the AI in the sim matters more in the soft period. Whenever a team wins over the AI, we save a time stamp. At the beginning of the hard period we determine the initial order-of-strength of teams by those time stamps: teams won against the AI earlier are stronger; strongest team is the first on the list. Those teams who could not defeat the AI during the soft period are randomly placed at the end of the list.

The soft period takes 2 hours, the hard period takes 22 hours.

Once we have an order-of-strength list, hard round starts. In the hard round, before each game we pair up teams according to their place on the order-of-strength list and teams fight eachothers. There are 15 pairs in a game, thus each player plays 1 battle in a game. In that 15 battles, all 30 teams participate exactly once. A game takes slightly less than one hour. If team A beats team B, and team A was lower on the order-of-strength list, we swap the two teams before the next game. This does not affect pairing for the current game but for the next game. On a tie, teams are swapped and neither team gets any score.

At the end, it is not the order-of-strength list that directly matters. Instead, any time a team wins in the hard period, the team earns score proportional to the strength (at that time) of the team that lost the match. Beating the strongest team, 1st placed on the order-of-strength list, is worth 120 points while defeating the weakest team (last on the list) is worth only 4 points. The scale is linear. (So the best team can approximately earn 2500 points during the 22 battles).

Pairing for each game happens as follows, R being the serial number of the current game, first game is 0:

- R % 4 is 0: 1 vs 2, 3 vs 4, 5 vs 6, ..., 29 vs 30
- R % 4 is 1: 1 vs 3, 2 vs 5, 4 vs 7, 6 vs 9, ..., 24 vs 27, 26 vs 29, 28 vs 30
- R % 4 is 2: 1 vs 2, 3 vs 6, 4 vs 5, 7 vs 10, ..., 27 vs 30, 28 vs 29
- R % 4 is 3: 1 vs 4, 2 vs 3, 5 vs 8, 6 vs 7, ..., 25 vs 28, 26 vs 27, 29 vs 30

In other words: teams winning more will slowly climb up the list; sometimes a team need to win to keep its place. Being higher on the list lets the team play games for higher scores against more skilled team AIs.

Example

In the first two hours we will publish example images with appropriate lighting conditions. Please check the announcement channel.

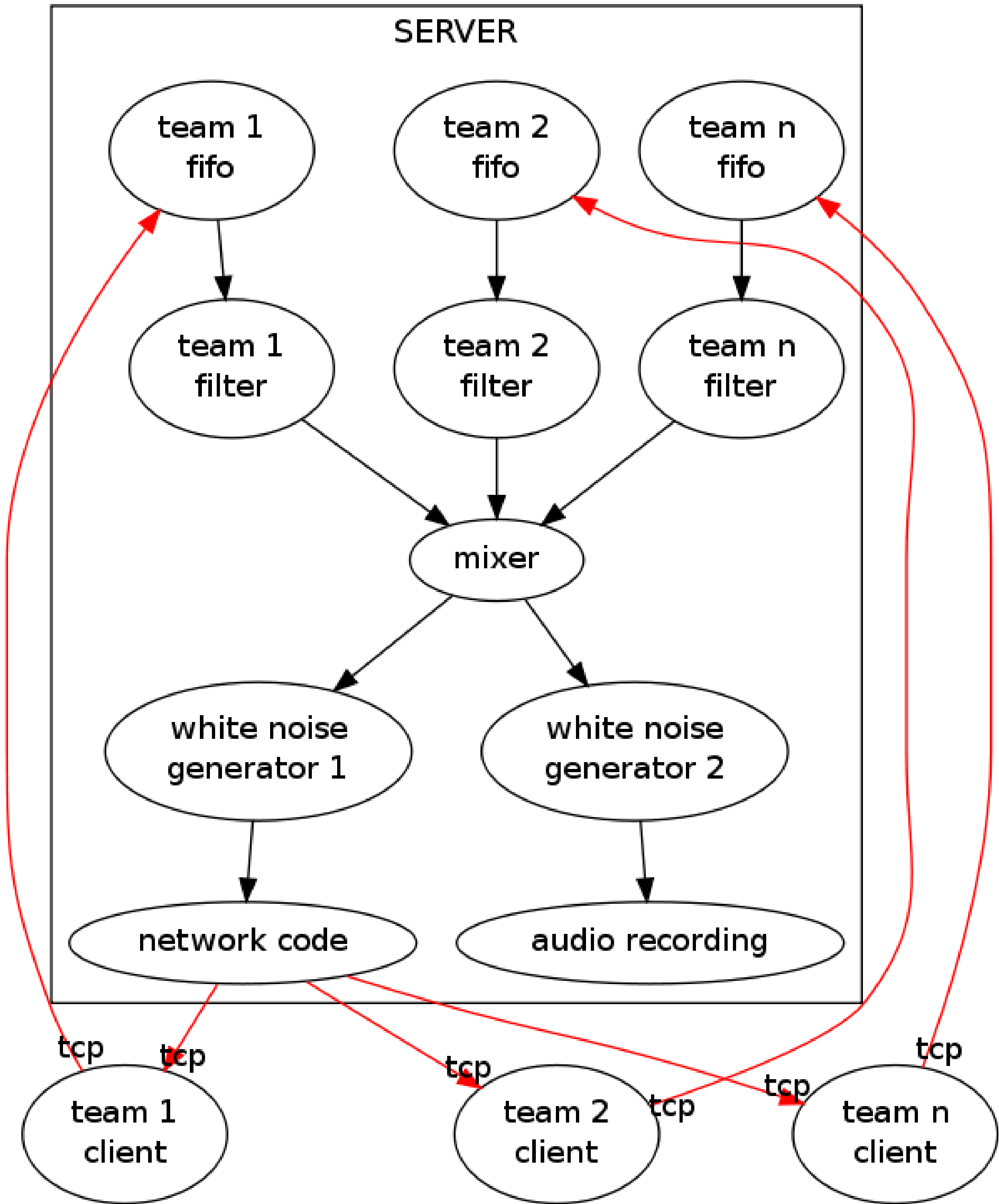
I-J. Radio (500 + 1500 points)

Channel

There is a radio server that accepts 30 connections (one per team). The server reads a stream of nibbles from each team, exactly 12000 nibbles per second (MSB nibble is read first from bytes). MSB of each nibble must be 1 (the "transmit bit"), LSB 3 bits encode an unsigned amplitude between 0 and 7. When a team is not submitting or when transmit bit is not 1, all zeros are assumed from the team.

Furthermore there is an input **fifo** and a **filter** for each team. The fifo is used to eliminate network latency problems: teams should keep the fifo filled for at least 2 seconds ahead in time to ensure smooth transmission. The filter implements a moving average of the last 6 samples of the player.

Signals from all 30 teams are mixed and some white noise is added: active nibbles from the buffer are all summed (<=7*30) and a random number between 0 and 15 (white noise) is added and "broadcasted" (sent back on all active connections) as "live broadcast". Broadcast format is 12000 unsigned bytes per second, each byte is an amplitude value. Because of all the expected network latency on team transmission and server "broadcasting", send and receive will not be in sync, but broadcast will have a varying delay. Data flow paths and connections are illustrated below.


SERVER
team 1
fifo
team 2
fifo
team n
fifo
team 1
filter
team 2
filter
team n
filter
mixer
white noise
generator 1
white noise
generator 2
network code
audio recording
tcp
tcp
tcp
tcp
tcp
tcp
team 1
client
team 2
client
team n
client

The server also hosts a **reference file** provided over http. Once every 30 seconds the file is replaced with a new one. Past reference files are *not* accessible.

Task

Teams should observe the reference file and encode enough information in the live stream to be able to restore the file content later. The server starts checking and scoring this ability after **4 hours** of the contest by making 30 second snippets of the live broadcast available from time to time, each starting exactly when a new reference file was published in the past. The team shall send back a reference file; if the submitted file matches the the reference file that was published in the same 30 second period when the record was taken, then the submission is accepted and the team gets a score, else the submission fails, the team gets a penalty and the team may try again.

Note: due to the method used for recording the stream, playback may have different noise compared to the live broadcast.

Networking and buffering

Each team may have a TCP connection to the server. If a new connection is established from the same team, the server will close the old connection. Team fifo persists accross connections: if the team feeds 5 seconds worth of material in the fifo from a client then reconnects with another, the new client will start appending to that 5 seconds.

Submission

After 4 hours, every 15 minutes the server picks a random snippet from the whole history of the broadcasting and publishes it on a web server as the next input file. Teams should submit the reference file that was available for the longest time during the time the recording was taken.

Scoring

This task is divided into two parts, task I and task J.

Task I consists of the initial 4 hours and a 10 hours submission period, so 40 submissions total. Each accepted submission is worth 12 points, the penalty for wrong answers is -2 points.

After the 14 hours the radio broadcast will be reset. Task J consists of a 10 hour submission period, the published snipets will be only taken from the broadcasting after the 14 hours. There will be no published snipet in the 14th hour and 24th hour so in this task there will be 38 possible submissions. Each accepted submission is worth 40 points and the penalty is -5 points.

The reference file for the same snipet can only be submitted once.

K. Punchcard (2800 points)

Some people strongly believe that the only durable data storage is paper. A popular way to store data on paper is using punch cards or punch tapes. If we already have a processor controlled device that can read/write the cards, we can use spare resources to implement stand-alone applications. To demonstrate the idea, a minimalistic implementation is provided for testing during the 24 hours of the contest.

The firmware provides access to a custom 4-bit processor and is programmed to read programs from punch cards then process data cards running the program. Teams shall submit programs on paper. To avoid producing excess amount of little paper cutouts, we use pens to mark holes on the paper: a black solid mark means 1, no mark means 0. Marks should be at least 4 mm in diameter.

4-bit CPU specification

This is a 4-bit computer, handling nibbles. Whenever multiple nibbles are stored and handled as a single number, the first nibble contains the least significant bits; bits from left to right in a nibble is MSB -> LSB. Example: "0001 1010" as a 8 bit unsigned value is 161 in decimal. The only exception is when explicitly specified otherwise.

4-bit Registers

code	name	description
0	A	acc
1	STATUS	status SDZC: SI page, DI page, zero, carry
2	SI	source index
3	DI	destination index

Instruction set summary

0000	LDAS	move [SI] to A
0001	STAD	move A to [DI]
0010	ADDAD	add A to [DI], store result in [DI]; updates C and Z
0011	ADDAA	add A to [DI], store result in A; updates C and Z
0100	ADDADC	add A to [DI] and C, store result in [DI]; updates C and Z
0101	ADDDAC	add A to [DI] and C, store result in A; updates C and Z
0110	ADDSDC	add [SI] and [DI] and C, store result in [DI]; updates C and Z
0111	NOP	no operation

1000 iiii	LDA i	load A with immed
1001 iiiiiii	JMP i	jump to address i
1010	INVALID	(invalid instruction)
1011 iiiiiii	CALL i	absolute call to address i
1100 rrbb	BTC r,b	set bit b in register r to 0
1101 rrbb	BTS r,b	set bit b in register r to 1
1110 ddss	MOVE d,s	move value from register s to register d
1111 0000	LDAD	move [DI] to A
1111 0001	STAS	move A to [SI]
1111 0010	RET	return from call
1111 0011	NAND	nand A to [SI], store result in A
1111 0100	SKZ	skip next instruction if Z
1111 0101	SKC	skip next instruction if C
1111 0110	LOOPI	decrement CNTR, increment SI, increment DI; skip next instruction if CNTR is zero;
1111 0111	LOOP	decrement CNTR, skip next instruction if zero
1111 1011	SBAD	sub A from [DI], store result in [DI]; updates C and Z
1111 1100	SBDA	sub A from [DI], store result in A; updates C and Z
1111 1101	DECA	decrement A
1111 1110	INCA	increment A
1111 1111	FIN	task finished

Syntax used in the table: [SI] or [DI] means internal random access memory (RAM) or one-time-programmable memory (OTP, paper) indexed by SI or indexed by DI (indirect memory access). Whether RAM or OTP is used depends on S and D bits of the STATUS register (see below).

Data memory

RAM map (page 0):

0	RAM0:	ram
1	RAM1:	ram
2	RAM2:	ram
3	RAM3:	ram
4	RAM4:	ram
5	RAM5:	ram
6	CNTR:	counter
7	IPHI:	instruction pointer high
8	IPLO:	instruction pointer low
9	IPHD:	in: phototransistors (data)
10	IBIN:	in: bin (3210; 0: index; 1: paper end A; 2: paper end B; 3:)
11	OVOL:	out: SPK volume
12	OFRQ:	out: SPK freq
13	ODIR:	out: in linear mode motor directions (3210; 0:- 1=paper_seek 1: pen 0=up 1=down; 2: 0=pen left 1=pen right 3: 0=paper+ 1=paper-); in seek mode target row
14	OMOV:	out: motor enable (3210; 0: mode 0=linear move 1: allow pen; 2: allow left/right; 3: allow paper)
15	DLY:	if >0, delay DLY cycles and set DLY to 0

Any non-ram is device mapped in. in devices should not be written. See how to handle IPHI and IPLO writes below.

memory map (page 1): 0..15: OTP slots on paper

Code memory size: at most 100 nibbles

code memory: 128 nibbles

Reset condition: IP (among with IPLO, IPHI), A, STATUS, SI and DI are all reset to 0. Content of other registers in the internal ram is unspecified. Content of the OTP depends on the input. Code memory beyond the submission length is filled with invalid instructions (to avoid leakage of other team's code).

IPHI and IPLO

IP is updated after the current instruction is already fetched from the code memory. The current instruction thus can access the address of the next instruction in IPHI and IPLO. In case the instruction modifies IPHI or IPLO the very next instruction will be fetched from the new address.

Submission

cards

Cards are preprinted forms on 297*63 mm stripes of paper. There are two type of cards: code and data. Bits on the code cards need to be filled in by the teams using the provided thick tip black pens. Teams may request code cards any time during the contest. Header fields of the code card needs to be filled in using a thin tip pen:

- Team: team name or ID
- Cookie: per team unique identifier of the submission; must end in "/X" where X is the taskid (the ID of the task the submission is for)
- Page: x of y, where x is page number of the given card, y is the total number of pages of the current submission

The leftmost column of preprinted black circles is the index column. Four bits of data is stored in each horizontal row that lines up with such an index dot. Data bits are either left white (for value 0) or marked (for value 1). A marking should be a solid black circle of 4..5 mm in diameter.

The first two rows of the first page (row 0 and 1) of the submission encodes the size of the submission (first row is the least significant nibble; leftmost bit in a row is the most significant bit of the nibble; the size and CRC nibbles are not added to this size). Row 2 contains a CRC calculated for the rest of the lines up to size. Because of the limit on code size, a submission is never longer than 5 pages.

Data cards look exactly like code cards but are marked as data and will be fed in the device after all code cards for a given program. A program consists any amount of code cards between 1 and 5 (inclusive) and a single data card.

CRC

The pseudo-code of the 4 bit CRC algorithm is

```
function crc4(nibbles[], len)
        crcpoly := 0x3
        crc := 0

        for i := 0 to len - 1 do
                crc := crc xor nibbles[i]
                for j := 0 to 3 do
                        crc := crc * 2
                        if (crc and 0x10) != 0 then
                                crc := crc xor crcpoly
                        end
                end
```

```
        end
        crc := crc and 0xf
        return crc
end
```

submission procedure

Teams submit their solution by handing over a full set of code cards to the operator who places the set in the input queue and enters team/cookie/taskid information into the database where the submission is marked as *pending*. Eventually the submission will get feeded in the device by the operator when the device becomes available. After all code cards for a submission are read by the device, it will calculate the CRC and stops with error if it does not match the CRC on row 2 on the first page. When CRC matches, the operator takes a data card with preprinted input data, fills in the header with team name and submisson cookie and feeds it in the device. The device will then start executing the code, reading and writing the data card in random access mode. When a FIN instruction is executed, the device stops and beeps and the operator removes the data card. After evaluation, the operator enters the result of the submission (score) into the database, where the team/cookie/taskid state will be changed from *pending* to *evaluated*. Data cards for test-run submissions are collected by the operator in an unlimited result queue and teams may pick up their data cards (among with their old code cards) any time during the contest. Data cards of for-score submissions are discarded.

Since the capacity of the hardware is limited, **submissions are accepted only in the first 23 hours of the contest.** There is an input queue for the hardware; each team may have a single submission in the queue. A team may cancel its last submission and add a new one, but that will be added at the end of the queue. When the device becomes available, the operator will always take the first submission in the queue. Runtime is accounted from the time the data card was succesfully inserted. If a program does not finish within a predefined timeout, evaluation of the submission is interrupted and the result is "failure due timeout".

Furthermore, for-score submissions always have priority over test-run submissions. In emergency cases (queue grows too long near the end of the contest or hardware failure), organizers may decide to evaluate submissions on a simulated device.

test-run submission

In test-run submissions teams submit both code cards and data card. After the evaluation the team gets back all cards. The team does not get any score by this method. This method is provided for allowing teams to debug their code on the real device in an off-line manner.

for-score submission

When a team is ready with a task, a for-score submission should be entered in the queue. In this case the team submits all code cards for the task and organizers provide a data card. At the end of the evaluation, the team will learn how the code finished (i.e. by running the FIN instruction, timeout, or other error). In case of a succesful execution (program ends with FIN within time), the organizers evaluate the output on the data card and also report whether the result is correct or not, and in case of a correct result, the team gets scores for the input (and no more submission is accepted for the same input). Teams **can not** read the data cards of for-score submissions.

trace submission

A simple web interface is provided for submitting cards containing at most 10 nibbles. In return the server dumps a trace. This service should be used to find out how the CPU works. The specification is as complete as it can be at this point, any question about how the CPU handles a specific instruction shall be checked on the trace submission server.

Tasks

Each task is worth 400 points.

#	task, input, output
1	add two 16 bit numbers input on paper: 4..7: op1 (MSB first) 8..11: op2 (MSB first) output on paper: 0..3: sum (MSB first)
2	sum N 4 bit numbers in a 8 bit accumulator with overflow detection input on paper: 3: number of operands (N) 4..N+4: operands output on paper: 0: sum, least significant nibble 1: sum, most significant nibble 2: non-zero on overflow
3	find largest 4 bit number in input input on paper: 1: number of operands (N) 2..N+2: operands output on paper: 0: largest number
4	count zero bits in a nibble input on paper: 0: input nibble output on paper: 1: number of zeros in input

5	logical operation: rotate right (4-bit). Shift bits one position to the right; LSB falling out on the right should come back in from the left (MSB). input on paper: 0: input nibble output on paper: 1: rotated nibble
6	logical operations: shift right (4-bit). Shift bits one position to the right, discarding excess LSB bit. MSB filler bit is 0. input on paper: 0: input nibble output on paper: 1: shifted nibble
7	Print human-readable roman numbers, readable when card is rotated 90 degrees CW or CCW. For 0 use normal arabic 0. input on paper: 0: input number between 0 and 3 (inclusive) output on paper: from 1: roman digits There shall be one empty line between each roman digit. 0 is 'f9f', I is 'f'.

L. Multitetris (2000 points)

Here is a simple networked two-player variant of tetris - with a couple of twists. Instead of two separate boards, the players play on the same board, with their pieces falling in opposite directions; and they have some control over the pieces their opponent gets (by having the ability to select a piece that *won’t* be spawned in the next turn).

The GAME

In every round there are two players: a left and a right player. The board is like a normal tetris board, except the pieces of left player are "falling" from left to right, the pieces of the right player "falling" from right to left. The players move after each other.
An example of "falling"

```
Left...................................Right
--------------------------------------------
|..........................................|
|...x............................oo........|
|...xx.---->...............<----.oo........|
|...x......................................|
|..........................................|
|..........................................|
|..........................................|
--------------------------------------------
```

Your pieces do not fall until the end of the board, it is stopped some steps before the another player’s wall.
Example

```
Left...................................Right
....|.<--right player’s piece
....|....will fall until this line
--------------------------------------------
|..........................................|
|.....x..........................oo........|
|.....xx.--->..............<----..o........|
|.....x............................o........|
|..........................................|
|...OO.....................................|
|...OO.....................................|
--------------------------------------------
..........left player’s piece----->.|
..........will fall until this line.|
```

If two pieces "meet" somewhere on the board, the current player’s piece is stopped. The next player has the chance to move away, however she doesn’t do so her piece will be also stopped. If a piece stopped, a new piece generated and started from the player’s starting wall. The stopped piece become a stationary wall. *Example of stopped pieces on the board*

```
Left...................................Right
--------------------------------------------
|..........................................|
|...x............................oo........|
|...xx.---->...............<----.oo........|
|...x..............O.......................|
|...............XXOO.......................|
|................XXO.......................|
|..........................................|
--------------------------------------------
```

Points

If there is a full column of stationary wall, it is destroyed, like in the normal tetris game. A player gets **1 point** for each destroyed character belonging to her, and **5 points** for each destroyed character belongs to the other player. *Example: left player destroys a column of stationary wall.*

```
Left...................................Right
--------------------------------------------
|................XX........................|
|................XOO.............oo........|
|..............x.XOO.............oo........|
|.........--->xxx..O.......................|
|.............X.XXOO.......................|
|.............X..XXO.......................|
|.............XXOOOO.......................|
--------------------------------------------

Next step:
Left...................................Right
--------------------------------------------
|.................X........................|
|.................OO.............oo........|
|................XOO.............oo........|
|...............XX.O.......................|
|..............X.XOO.......................|
|..............X..XO.......................|
|..............XXOOO.......................|
--------------------------------------------
```

In the example above the left player got 6*1 + 1*5 = 13 points. After a column is destroyed, the stationary wall "over" the destroyed column fall towards the other player's wall. The stationary wall "under" the destroyed column DOES NOT move anywhere.

Game over

The game is over, if a player's element is stopped before it can fully leave the player's wall. *Example: game over, right player lost the game*

```
Left...................................Right
--------------------------------------------
|...................................XX.O...|
|...x................................XXO...|
|...xx.................................OO.o|
|...x..............O................XXXOOoo| <--- piece cannot leave the wall
|...............XXOO................X......|      next step it will stop
|................XXO.................XX.O..|
|....................................XXOOO.|
--------------------------------------------
```

Moves

In one step a player can:

- step up/down or not step
- rotate the piece clockwise/ counter clockwise or not rotate

at the same time. So, for example one can rotate clockwise AND step up in the same step.

Piece generation

A piece is uniform randomly selected. However, a player (e.g. left player) can explicitely deny an element for the other player (right player) in the next turn. So, the generator will uniform randomly select from all the pieces, except the element denied by the left player.

Scoring

There will be 12 scoring periods (starting at odd hours: 9-11, 11-13 and so on) - these will show up as 12 "inputs" in the submission system. The evaluated scores will be the sums of points gained in the given scoring period; the real scores will be scaled based on these (with the best team getting 166 points in one scoring period). The total score one can get is 12 * 166 = 1992 points.

The Protocol

One team may maintain one TCP connection to the server at once (connecting to the server will kick the previous connection, if any). Disconnecting from the server doesn't affect any game state (ongoing games will remain ongoing). A team may reconnect to continue their game where they left off.

Each connected team is either playing a game against an opponent, or waiting for the next game (it's not possible or necessary to play multiple games at once).

A new round starts every few minutes. At the beginning of a round, all teams that are currently connected but are not playing, are paired up with other teams, and a new game is started for each pair.

All communication happens over TCP, in plain text, is line based (all messages in either direction are a single line), all line endings are unix.

Messages from the server

Messages from the server may consist of multiple tokens, separated by single spaces.

- **NEXTROUND** *seconds*

The team is waiting for the next game. The next round (when there's a possibility of getting in a game with an opponent) begins in *seconds* seconds.

- **AVAILABLE** *teams*

A round begins. The message contains the number of teams currently connected and not playing.

- **LEFT|RIGHT** *nextturn left_score right_score board_rows left_piece right_piece*

The `LEFT` or `RIGHT` message is sent to the left and right players at the beginning of each turn in the game (the message type indicates which side is the currently connected team playing).

- The *nextturn* field is either `left` or `right`; indicates which side plays the next move.
- The *left_score* and *right_score* fields are two integers that show the current score of each side.
- The *board_rows* field is an integer that gives the number of rows in the playing board (this value will not change).
- The *left_piece* and *right_piece* fields are two integers that show what piece is each side currently controlling.

Each `LEFT` or `RIGHT` message will be followed by *board_rows* * `BOARD` messages with the contents of the playing board (from up to down).

- **BOARD** *row...*

This message contains the subsequent row of the playing board. The *row* is a single string that is always the same length.

- spaces indicate empty places
- `x` belongs to the current piece of the left player
- `X` is a stationary wall belonging to the left player
- `o` belongs ot the current piece of the right player
- `O` is a stationary wall belonging to the right player

Messages sent to the server

The only kind of message a team can send is their current move. Moves are only accepted when it's the team's turn in the game; only one move is accepted per turn.

Moves are encoded in three character messages:

- [**ckn**][**uds**][**0123456**]

- The first character is either **c** for *rotate clockwise*; **k** for *rotate counter clockwise*; or **n** for *don't rotate*.
- The second character is either **u** for *move up*; **d** for *move down*; or **s** for *stay (don't move)*.
- The third character is the *piece inhibit control*. If a new piece spawns for the opponent in the next turn, they surely will not get the specified piece (if no new piece spawns, the third character is ignored).

From the last sent `BOARD` message, teams have **one second** to make their move. Not making a move (by not sending anything, or by not even being connected) implies a **ns0** move (after a one second wait).

Pieces

This game has the usual tetris pieces, indexed from 0.

- Piece **0**

  ```
  ....
  ##..
  ##..
  ....
  ```

- Piece **1**

  ```
  ....
  #...
  ###.
  ....
  ```

- Piece **2**

  ```
  ....
  ###.
  #...
  ....
  ```

- Piece **3**

  ```
  ....
  ....
  ####
  ....
  ```

- Piece **4**

  ```
  ....
  ##..
  .##.
  ....
  ```

- Piece **5**

```
....
.##.
##..
....
```

- Piece **6**

```
....
#...
##..
#...
```