CHALLENGE24
INTERNATIONAL 24-HOUR PROGRAMMING CONTEST

14th International 24-hour Programming Contest

http://ch24.org

MAIN SPONSOR

SAP

DIAMOND GRADE SPONSORS

FORNAX

Google™

Microsoft

LogMeIn®

ORGANIZER

mave

Magyar Villamosmérnök- és Informatikus-hallgatók Egyesülete

eestec

LC Budapest

PROFESSIONAL PARTNERS

hte

HÍRKÖZLÉSI ÉS INFORMATIKAI TUDOMÁNYOS EGYESÜLET

NJSZT

Contest

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

Rules

The contest starts at 2014-05-03 09:00 CEST and ends at 2014-05-04 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**	**Scheduled**	**Score decreases with time**	**Penalty for wrong answer**	**Time delay after fail/pass**	**Scaling**	**Queue**	**Max score**
A (Halting problem)	Yes	No	No	Yes	-5	0/0	No	No	1000
B (Bug fixing)	Yes	No	No	No	0	60/60	No	No	4000
C (Complete program)	Yes	No	No	Yes	-5	0/0	No	No	1000
D (Firing game)	Yes	No	No	Yes	-5	0/0	No	No	1000
E (Disease)	Yes	No	No	Yes	-5	0/0	No	No	1000
F (Swap)	Yes	No	No	Yes	-5	60/60	Yes	No	1000
G (Slothlers - Manage)	No	Yes	No	Yes	0	0/0	No	No	500
H (Slothlers - Produce)	No	Yes	No	Yes	0	0/0	No	No	500
I (Slothlers - Tournament)	No	Yes	Yes	No	-5	0/0	No	No	3600
J (Sonar)	No	Yes	Yes	Yes	-5	0/0	No	No	about 5000
K (OSM - Search)	No	Yes	No	Yes	0	0/0	Yes	No	1000

Task	**Web submission**	**Interactive**	**Scheduled**	**Score decreases with time**	**Penalty for wrong answer**	**Time delay after fail/pass**	**Scaling**	**Queue**	**Max score**
L (OSM - Path)	Yes	No	No	Yes	-5	0/0	No	No	1000
M (OSM - Race)	No	Yes	Yes	Yes	-5	0/0	No	No	3000
N (Dog tag)	Yes	No	No	Yes	-5	0/0	No	No	1000
O (Ball)	No	Yes	Yes	Yes	0	0/0	No	Yes	4000

- 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.
- Scheduled: continously running task during the contest with scheduled interactions.
- 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.)
- Queue: Only one team can work on this task at a time (hardware task) so there will be a first-come, first-served queue.

Ports

Port	Task	Service description
`80`	-	web
`6667`	-	irc
`u123`	-	ntp server
`u53`	-	dns server
`u67,u68`	-	dhcp server
`16700`	G	Slothlers - Manage
`16800`	H	Slothlers - Produce
`16900`	I	Slothlers - Tournament
`u17000`, `u17100`	J	Sonar
`16400`	K	OSM - Search
`16500`	L	OSM - Path
`16600`	M	OSM - Race
`16100`	O	Ball control
`16200`	O	Ball video stream

Ports starting with `u` are UDP ports. All 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.

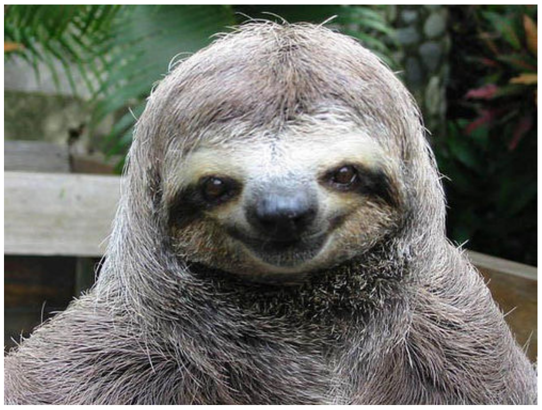
For general discussions and questions join the `#challenge24` irc channel.

There will be separate channels for task related problems as well: `#A`, `#B`, `#C`, `#D`, `#E`, `#F`, `#G`, `#H`, `#I`, `#J`, `#K`, `#L`, `#M`, `#N`, `#O`.

Prologue: sloths

Sloths are cute little animals known for being slow and lazy. This common opinion is based on an old misunderstanding. Sloths are actually highly intelligent, more intelligent than humans and dolphins put together. They are also very focused: they use their extraordinary brain capacity for solving problems that are both more interesting and more relevant than moving around fast or chatting.

Not wasting much thought on social interaction or training their motor system may have stopped us humans from realizing how interesting these animals are, but times are changing. This year's problem set will reveal some of the problems sloths solve routinely and let you compare the efficiency of your team to the average sloth!



J. Random Sloth
source: http://creepypasta.wikia.com/wiki/File:Happy-smiling-sloth.jpg

A. Halting problem (1000 points)

The sloth brain can store virtually any amount of information, so in theory any sloth can know Everything. However, acquiring new information about the sorrunding world is more of a distributed task: the sloth that needs the information may not be the one who acquired it, and asks a friend who in turn may ask another friend. Anyone passing on a question also does some processing and transformation on it. When the question finally reaches the sloth who knows the information, the answer is passed back to the original sloth - or not (if they fall in an infinite loop).

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

Such queries can be modelled with function calls, which makes it easier to understand whether a specific question can be answered (you just need to solve the halting problem, which should be trivial).

The information processing of each sloth can be described by a function *f* that can be queried with one argument *A* and returns a value. This function can be defined by four parameters: *g*, *h*, *X* and *Y* with the following pseudo code:

```
function f(A) {
        if (A == 0) then {
                return g(A + X)
        } else {
                return h(A + Y)
        }
}
```

Where *g* and *h* are functions with similar definitions (queries to other sloths), *X* and *Y* are unsigned integers between 0 and 2^{64}-1.

As a special case *g* or *h* functions may be a simple identity function (returning their argument) instead of a full query to another sloth.

The **+** operation is modulo 2^{64} addition.

Your task is to evaluate various queries from various sloths given the functional model of the sloth computation system.

Input

First line contains two numbers: *N* the number of functions and *Q* the number of queries that needs to be evaluated.

The next *N*-1 lines are the function definitions of the sloths indexed from 1 to *N*-1 and 0 is the index of the identity function. A function is given by four numbers: *g* and *h* function indices and the *X*, *Y* parameters.

The next Q lines are the queries given by two numbers: f the function index and A the argument that the function is applied to.

Output

For each query output a line with a single number: either the returned value or -1 if the query will never finish.

Example input

```
10 10
0 0 4611686018427387904 9223372036854775808
6 2 6917529027641081856 4611686018427387904
8 4 13835058055282163712 11529215046068469760
0 0 13835058055282163712 6917529027641081856
1 0 11529215046068469760 6917529027641081856
2 0 11529215046068469760 16140901064495857664
6 0 11529215046068469760 0
1 2 0 16140901064495857664
9 7 13835058055282163712 4611686018427387904
7 4611686018427387904
6 16140901064495857664
0 4611686018427387904
0 4611686018427387904
7 11529215046068469760
6 6917529027641081856
0 16140901064495857664
0 11529215046068469760
2 9223372036854775808
8 4611686018427387904
```

Example output

```
-1
-1
4611686018427387904
4611686018427387904
-1
-1
16140901064495857664
11529215046068469760
-1
9223372036854775808
```

B. Bug fixing (4000 points)

Some researchers managed to find indirect proof that some sloths are developing software in their idle time. They even managed to capture some of the software in object code format. For this kind of hobby, as it turned out, sloths are simulating a processor that runs in a functional manner. Thanks to many hours of reverse engineering, there is already a detailed description of the imaginary processor.

However, the programs captured seem to be buggy. Researchers believe they may gain more trust from some of the sloths if they can prove themselves to be worthy by playing the sloth-programming-game and fix one of the programs.

How many leaves do you want for dinner?
A F A F A F A F A F A F A A A A 4 1 0 2 3 F O 48 0 F O 49 0 F O 10 0 F 0 F F F A A 3 A A 3 2 1 0 F F F A 2 A 1 0

sources: http://vintageprintable.com/wordpress/vintage-printable-animal/animal-wild-animal-miscellaneous/animal-sloth-photo-two-toed-sloth-hanging-on-a-stick/
http://www.clker.com/clipart-cartoon-speech-bubble.html

Program format

Each program is a tree of operations, with a single operation as its root. The tree is represented as a space-separated list of words. To parse a program, parse its root operation. To parse an operation, consume a word from the input (which will specify the type of operation), then recursively parse the operation's children as needed (depending on the type).

There are 5 kinds of operations:

- `F` *op* = **Function**
  - Parse a nested operation.
  - Stands for a function with one argument (a "lambda abstraction"). The function body is the given operation.
- `A` *func_op arg_op* = **Apply**
  - Parse two nested operations.
  - Stands for a function application. The function in *func_op* is applied to the argument in *arg_op*.
- *index* = **Reference**
  - There's no type character. Instead, a word that is a non-negative integer is a Reference with the given number as its index.
  - Stands for a variable reference. `0` references the argument of the closest parent function; `1` references the argument of the parent function of the closest parent function, and so on.
- `O` *ascii op* = **Output**
  - Parse a non-negative integer (0-127) and a nested operation.
  - Writes the ASCII character specified by the integer to the output (with no buffering), and evaluates to the given operation.

    - Programs will output endlines with a single ASCII 10 (LF).
- `I` *op* = **Input**
    - Parse a nested operation.
    - Reads a single ASCII character from the keyboard, and evaluates to *op b7 b6 b5 b4 b3 b2 b1* - the nested operation (which must be a function) applied to *b7* (the MSB of the input character), the result applied to *b6*, then so on until *b1* (the LSB of the input character).
    - 1 bits must be given by the expression `F F 1`
    - 0 bits must be given by the expression `F F 0`
    - Programs will expect endlines (the Enter key) to be represented by a single ASCII 10.

Examples:

- `F F 1`
    - Function(Function(Ref 1))
    - `function(x) { function(y) { return x } }`
- `F A F A 1 A 0 0 F A 1 A 0 0`
    - Function(Apply(Function(Apply(Ref 1, Apply(Ref 0, Ref 0))), Function(Apply(Ref 1, Apply(Ref 0, Ref 0)))))
    -
      ```
      function(g)
      {
        return
          ( function(x) { return g(x(x)) } )
          ( function(x) { return g(x(x)) } )
      }
      ```
    - Also known as the "Y combinator".
- `F O 65 0`
    - Function(Output(65, Ref 0))
    - `function(x) { return write_and_return('A', x) }`

Evaluation

To execute a program, we use the two functions `step` and `run`, and a number of data types: *thunk*, *closure*, *val* and *environment*.

A *val* is either an *operation* or a *closure*.

thunk and *closure* are distinct types, but both are pairs of a *val* and an *environment*.

An *environment* is a singly linked list with mutable elements. Elements held by the list are either a *val*, or a *thunk*. It should be possible to create a new list head, referencing an existing environment in the "next" pointer of the new node, without disrupting or copying the existing environment.

`run`(*val*, *environment*) returns a *closure*. To execute a program, evaluate its root operation using `run`(*root_op*, *empty_env*) (and discard the result).

`step`(*val*, *environment*) returns a *val* or a *thunk*.

Every data type holds references, and are passed by reference (no need to copy anything).

```
/* the definition of run */

run(val, env)
{
  loop until val is not a closure {

    result := step(val, env)

    if result is a thunk {
      val := result's val
      env := result's environment
    }

    else { /* if result is a val */
      val := result
    }

  }

  return val
}

/* the definition of step */

step(val, env)
{
  if val is a Function operation {
    return a closure made from the function body op and env
  }

  else

  if val is a Reference operation {
    index := the index from the Reference in val
    result := env[ index ]
    /* the list in env is indexed, with first element indexed as 0 /*

    if result is a thunk {
      result := step(result's val, result's environment)
      env[ index ] := result
    }

    return result
  }

  else

  if val is an Apply operation {
    func := run( func_op from Apply, env )
    arg := a thunk made from arg_op from Apply and env

    func_env := the environment in the func closure
    func_val := the val in the func closure

    /* a new environment list node, with arg as the element, and func_env as the rest of the list */
    new_env := prepend arg to func_env

    return a thunk made from func_val and new_env
  }
  else

  if val is an Output operation {
    Write the given character to the output (with no buffering).

    return a thunk made from the output’s operation and env
  }

  else

  if val is an Input operation {
    c := ASCII code of a single character read from the keyboard

    if c is EOF, or > 127, then stop

    in_op := the input’s operation

    true := Function(Function(Ref(1)))  /* F F 1 */
    false := Function(Function(Ref(0))) /* F F 0 */

    /* create 7 Apply operations, nested into each other */
    result := Apply(Apply(Apply(Apply(Apply(Apply(Apply(in_op,
      true if (c & 64) != 0, else false),
      true if (c & 32) != 0, else false),
      true if (c & 16) != 0, else false),
      true if (c & 8) != 0, else false),
      true if (c & 4) != 0, else false),
      true if (c & 2) != 0, else false),
      true if (c & 1) != 0, else false);

    return a thunk made from result and env
  }

  else

  {
     if none of the above (because val is a closure),
      return a thunk made from val and env
  }
}
```

Input

In the inputs directory, there are a number of small working example programs, and a main program.

The main program is a big program with an interactive menu system. The menu system can be used to launch a self test function. The self test mechanism itself is known to work correctly (i.e. it evaluates correctly whether each test passes or fails), but many tests fail. No other function in the program works, but it’s assumed that if the tested subroutines in the program are fixed and the tests pass, then the rest of the program will work too.

While the internals of the main program aren’t well understood, some researchers think the names of the tests may be used as a starting point for mapping the subroutines.

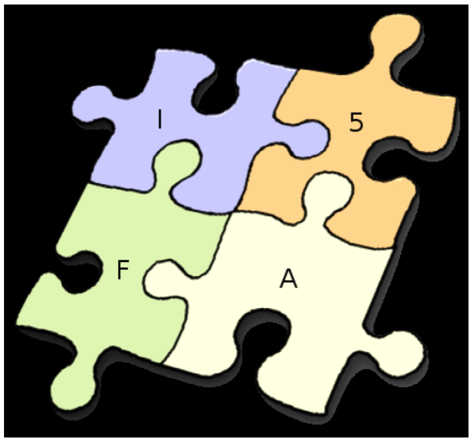
Output

Submit a (partially) fixed main program.

8 tests are run on the program (some input is provided to the submitted program and a certain output is expected; the program itself is not examined). Each passing test is worth 500 points (and it’s not decreasing with time).

The program may be resubmitted an arbitrary number of times (there is no penalty). Only the last submission is scored.

C. Complete program (1000 points)



source: http://commons.wikimedia.org/wiki/File:Jigsaw.png

Dying sloths often whisper their last will in a functional programming language. Breathing may be hard in the last moments and a few syllables are usually missing from these programs. Realizing the importance of the testament of a sloth, you decide to fix those broken programs by guessing the missing tokens.

Given an incomplete program of the previous task (Bug fixing) insert the minimum amount of symbols (F, A, I and O letters or numbers) to make it grammatically correct.

Only the number of inserted tokens matter, the separating whitespace characters don't count. A correct program can be parsed using the following context-free grammar production rules:

```
Op → 'F' Op
Op → 'A' Op Op
Op → 'I' Op
Op → 'O' number Op
Op → number
```

Where the `Op` non-terminal is the start symbol of the grammar, `number` is a terminal token consisting of decimal digits and the quoted characters are single character terminal tokens.

Furthermore the following constraints must hold for a correct program:

- The number following an `'O'` token must be between 0 and 127 inclusive.
- A reference number (the result of the last production rule above) must be smaller than the number of `'F'` nodes above it in the parse tree (so the number of times the first rule is used to derive the number in the last rule must be greater than the derived number).

Input

An incomplete program consisting of F, A, I and O letters and numbers separated by space.

Output

A correct program created from the input by inserting as few tokens as possible. (More than one solution may be possible).

Example input	Example output
`I A O 42`	`I F A O 42 0 0`

D. Firing Game (1000 points)

Sloths didn't like the ***Cutting back middle management*** *task of last year's Electronic Contest, but they did like the idea of optimizing corporate downsizings. They've come up with the following task which they believe to be a better variant of the same story.*

source: http://images.businessweek.com/ss/08/12/1215_layoffs/image/intro.jpg

This is not the first crisis in the history of IGG. Whenever cost reduction was needed in the past, firing employees worked to some degree. This affected workers at the bottom of the food chain more often; managers could generally evade the threat. Over the decades this caused an unusually large amount of middle managers to accumulate. The CEO finally realized there's no other way to balance the structure of the company but to make their positions redundant.

A middle manager is somebody who has subordinates (who might also be managers), but the CEO is not a middle manager. Everyone can have at most M subordinates. If someone is fired, their subordinates are reassigned to their boss. It's not allowed to fire somebody, if that would mean their boss will have more than M subordinates.

The CEO knows that competition usually improves results, so he hired two consultants that compete in firing the middle managers. The consultants fire managers alternatively. The consultant who has no one to fire loses, and won't get paid for his work. Determine which consultant will get paid, the one who fires the first manager, or the one who fires the second. Assume both consultants use the optimal strategy.

Input

First line contains *T* and *M*, where *T* is the number of test cases to be solved. Each test case starts with a line containing a single integer *N*. In the following *N* lines, the *I*th line contains:

- k_1, the number of non-manager subordinates of the *I*th manager
- k_2, the number of manager subordinates of the *I*th manager
- k_2 numbers, that are the numbers of the manager subordinates of the *I*th manager

It is guaranteed that the 0th manager is the CEO, the middle managers are numbered 1 to *N*-1.

Output

One line per test case: 1 or 2, depending on which consultant wins in that case.

Example input

```
2 3
6
1 2 1 3
1 1 4
1 0
1 2 5 2
1 0
2 0
8
0 2 6 7
3 0
0 1 1
2 0
2 1 3
1 0
0 3 4 5 2
2 0
```

Example output

```
2
1
```

In the first case, only managers number 2 and 4 can be fired in any case, so after two steps, the first consultant can’t fire any more people.

E. Disease (1000 points)

Sloths don't have to deal much with STDs. It's not because they are not engaged in romantic activities, but because they calculate and arrange these activities in a way that makes STDs unable to spread. For doing so, they have a standard way modeling these things and they can do all the calculations in a fraction of a second before they even make an eye contact with a pretty sloth from the other sex. This method is so effective that STDs are unable to survive in sloth populations.

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

The sloth's model of the situation is simple: there are male and female sloths. Some males are *compatible* with some females so they can make a couple. They model the worst case (the most chance for STDs) and assume that each night every sloth is randomly paired up with a *compatible* sloth from the other sex in a way that the number of pairs is maximized. Sloths are not particularly monogamous, so the actual pairing may differ from night to night.

The next step in the model is infecting one of the sloths with STD and simulating what happens: how many other sloths have the chance of getting infected in the future. This is the step you are required to compute in this task.

Input

For the same group of sloths, there are multiple questions with the original carrier of the virus being different.

The first line of the input has 4 numbers, *M F C Q*: the number of males (*M*) and females (*F*), the number of compatible connections (*C*), and number of questions (*Q*)

The next *C* lines have 2 numbers each, *m f* indicating that the male sloth *m* and the female sloth *f* are compatible. Both males and females are identified by integers counting from 0.

The next *Q* lines start with an *M* or an *F* and a number *k*, that means that *k*-th male or female is the original carrier for that question.

Output

For each question, output 2 lines. The first line should be 2 numbers, *M* and *F*, the number of males and females in danger. The next line should have *M*+*F* numbers, each number referencing a sloth that may get infected. First list all the *M* males in ascending order of their IDs then all the *F* females also in ascending order. The original carrier should be included in the list.

Example input

```
9 10 19 5
0 0
1 0
2 1
2 2
2 3
2 4
3 4
4 5
4 6
5 5
5 6
5 7
6 6
6 7
7 7
7 8
7 9
8 8
8 9
M 0
M 2
F 4
M 4
M 7
```

Example output

```
2 1
0 1 0
1 3
2 1 2 3
1 1
3 4
3 3
4 5 6 5 6 7
2 2
7 8 8 9
```

F. Swap (1000 points)

There is a reason why sloths do not visit art museums: their sense of art differs from ours. They prefer total assymetry, so they do not find paintings beautiful if they see the same color appear more than once in the same row or the same column (of pixels). Unfortunately they do have the computation power in their brain to spot even a single repetion on the largest paintings within a blink of an eye.

source: http://wugange.com/colorful-cubism-28004-hd-wallpapers.html

If you are ever going to get any sloth to look at your paintings, you will need to fix them first. The easiest way to make it look good for a sloth while keeping the original content is to cut&paste portions of the painting, swapping rectangular areas.

Your job is to come up with a list of rectangle swap instructions, which if executed in order, will result in a sloth-compatible image. Since cutting and pasting is a tedious task, you need to minimize the number of rectangle swaps.

Input

Input is a color png.

Output

The first line must contain one integer *N*, the number of swaps. The next *N* lines contain *N* swap instructions. An instruction consists of 6 integers *x1*, *y1*, *x2*, *y2*, *w*, *h*, separated by arbitrary amount of whitespace, where *x1* and *y1* are the coordinates of the UPPER LEFT corner of the first rectangle to swap, *x2* and *y2* are the coordinates of the UPPER LEFT corner of the second rectangle, and *w* and *h* are the (common) width and height of the rectangles, respectively, in pixels.

Pixels are indexed from zero and pixel (0, 0) refers to the upper left corner of the image. Furthermore *w* and *h* must be at least 1.

The two rectangles must not overlap and both rectangles have to lie entirely within the image. Any swap instruction that violates these conditions is considered invalid. Any submission containing invalid swap instructions is rejected.

Trailing garbage at the end of the lines is ignored. Trailing empty lines after the valid swap instructions are allowed but non-empty lines are not.

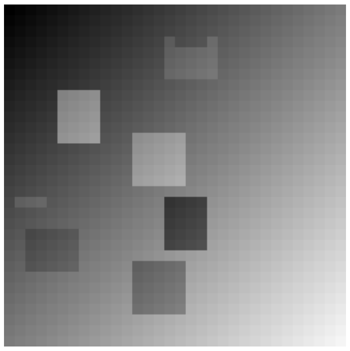
Scoring

The final score is

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

where *N* is the number of swaps in the submission and *BEST* is the number of swaps in the best submission.

Example input



(Note: the image is magnified by 10 times -
the original png is in the input directory as 0.png)

Example output

```
4
12 12 12 24 5 5
1 16 16 1 3 3
15 3 2 21 5 4
5 8 15 18 4 5
```

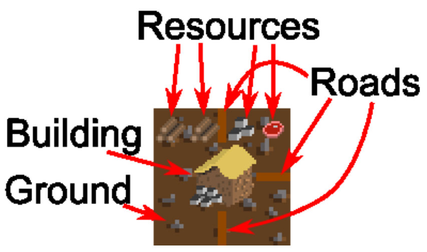
GHI. Slothlers

Sloths often spend their time day-dreaming about the whole world densely populated with a crowd of sloths running a complex economy. Since sloths are very peaceful, they never imagine wars, only building and creating. And when a sloth imagines an utopia, it’s not a set of blurry general thought, but very precise and detailed.

Prof. Scott S. Royknock, Director of Research at the International Sloth Research Foundation believes this utopia will turn into reality some day and the whole world will be governed and operated by sloths. The key for human race to survive under such conditions is to fully understand the new system. In this task we are looking for the team who knows the most about this system; knowledge is proven by utilizing the system better than other teams in one-to-one matches.

Overview

The game is played on a square grid map. In the description, the squares will be referred to as fields. Each field has a ground type, each field can contain a building, some resources, and each field can be connected to some of the 4 adjacent fields with roads. The map has a toroid structure, which means that moving east from the east edge of the map goes to the west edge, and moving north from the north edge goes to the south edge.



Overview of a field

The game is always played with two players. The goal of the game is to get a bigger score than your opponent. This can be achieved through claiming territory, mining resources, processing those resources, then turning those resources into score points.

This task has a single player portion, where the teams play against a passive AI, and certain pre-set objectives must be met. There is also a multi player portion, where the teams are paired up for the games in a Swiss tournament like system, and scoring for the competition will be based on the final standing in this ranking.

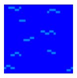
At the beginning of a game, players will already have some basic buildings, resources, and some roads built. The game is played in alternating turns for the two players. Each turn, players can build roads and buildings, use buildings to produce new resources, and move their existing resources around. They send their commands for each turn in a network protocol, and receive a text stream containing their opponent’s actions and feedback on their actions.

Players have a time constraint of 4 seconds for each turn, and a chess clock-like total time limit of 400 seconds for the whole game, that runs while waiting for commands from that player. Each turn takes at least 0.1 seconds on the clock, even if the player responded sooner. If a player’s time limit runs out, the player is considered to have passed the turn, taking no actions. The game is over after 1500 turns, and the player with more score at that point wins.

The rest of the description will start with the overview tables of the building types, ground types and resources. Then the detailed rules of the game, the communication protocol, and finally technical details like how/when servers will be run and so on.

Reference tables

Ground types

letter - ground type	description	image
G - Grass	General terrain with no special properties	
W - Water	Water is an obstacle that can’t be built over	
S - Stone	Stone can only be produced from stone fields. Contains 40 stones	
C - Coal	Coal can only be produced from coal fields. Contains 40 coal pieces	
I - Iron ore	Iron ore can only be produced from iron ore fields. Contains 40 iron ore	

Resources

letter - name	image
W - wood	
S - stone	
G - grain	
M - meat	
F - food	
C - coal	
I - iron	
P - product	

Buildings

The one letter code for the buildings is derived from the resource they produce.

letter - building name	building cost	ground type	production			image
			cost	time	product	
O - Junction foundation	1 wood 1 stone	-	-	-	-	
J - Junction	1 wood 1 stone	-	-	-	-	
W - Lumberjack	4 stone	-	-	4	wood	
S - Quarry	4 wood	Stone	-	2	stone	
G - Farm	4 wood 2 stone	Grass	-	3	wheat	
M - Pig farm	2 wood 4 stone	-	1 wheat	5	meat	
F - Tavern	4 wood 4 stone	-	1 wheat 1 meat	2	food	
C - Coal mine	4 wood 6 stone	Coal	1 food	3	coal	
I - Iron mine	4 wood 6 stone	Iron ore	1 food	3	iron ore	
P - Factory	2 wood 6 stone 2 iron	-	1 iron 1 coal	2	product	
N - Market	4 wood 2 stone 4 iron	-	1 product	1	1 score	

Game rules

Turn structure

Players have separate, alternating turns, that are numbered, so the game starts with the 1st turn of player 0, then the 1st turn of player 1, then the 2nd turn of player 0, and so on. Each turn consists of 2 parts: first part is the player actions, second part is automatic bookkeeping finalizing the turn, like buildings finishing their tasks and producing resources, or increasing the score.

Coordinate system

Fields are referenced by their (x,y) coordinates on the grid map. The map wraps around in both x and y directions (it has toroid topology) so a field always has four adjacent fields in the four main directions:

- north: -y direction
- east: +x direction
- south: +y direction
- west: -x direction

Construction of buildings

A field can be built on in 2 steps. First a junction must be built, which can already be used for transportation. Then, if needed, production buildings can be built on top of a junction. Buildings (including junction foundations) can never be built on adjacent fields, even to the opponent's buildings.

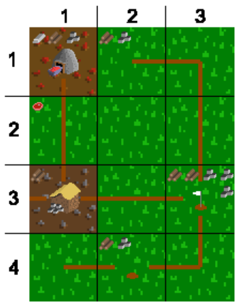
Junctions need to be built in 2 steps. An empty field that only has empty fields adjacent to it, and has at least one road, can be converted to a junction foundation for 1 wood and 1 stone, and then the junction can be built over the foundation for another 1 wood and 1 stone.

Every building other than the junction and junction foundation is a production building. Production buildings are built over junctions. Construction of buildings is instantaneous, if a field has all the necessary resources for a building, it can be built there. The build action removes the necessary wares, and replaces the junction with a building. It's impossible to build a building if more than 4 resources would be left after the building is complete.

Some buildings require a specific ground type, and can't be built on other types, and none of the buildings can be built on water. These requirements can be found in the table of buildings.

Construction of roads

Roads are constructed in road segments. Each road segment connects two adjacent fields. Water fields can't have roads built to them. The cost of each road segment is 1 wood and 1 stone, that has to be in the field from which you build the road segment. On successful building, the resources are removed, and the new road segment is added.



A road network

For example the road segment built from (1, 3) to the north is the same as the road segment going from (1, 2) to the south so it is enough to build a road from one direction, but the direction matters when building because of the placement of the resources and road constraints described below.

Road segments form paths on the map. A path is defined as a series of road segments with no buildings between them, however junction foundations do not count as buildings on the road network. So, for example, on the figure, there are paths between (1, 1) and (1, 3), between (1, 3) and (3, 3), between (3, 3) and (2, 1), and between (3, 3) and (1, 4).

The paths can’t have forks, so it is not allowed to build a road from a field which is empty, but already has more than one road segment going to it. (So it wouldn’t be possible to build a road segment south from (2, 4), since that would create a fork at a junction foundation.)

Paths have to be built one way, it is not allowed to build a path from two sides that meet in the middle. The exact rule is that it’s not allowed to build a road segment to a field with no building but 1 road (a junction foundation does not count as a building here either). It is of course allowed to build *from* such a field. In the figure, it is not allowed to finish the path by building to the east from (1, 1), only by building west from (2, 1).

Transportation

Resources can be moved about the map by the player who owns them. Each resource can only be moved once per turn, and a resource that has already been moved that turn can’t be used for anything else, like as a construction material, or as input to a production building.

A field can only hold a limited amount of resources at the time. For junctions this limit is 12, for fields with production buildings it is 4. On other fields there is a limit of 2, but also, resources can only be at one field along a path at any point. (So, in the example, the player couldn’t move anything north or south from (3, 3), because those paths are already occupied. They could move something west however.)

A path can only be used once per turn to move 1 or 2 resources along it. (In the example, if we moved the meat south from (1, 2), we still couldn’t move anything south from (1, 1), in the same turn, since the path from (1, 1) to (1, 3) was already used.) On the other hand, a road segment that was built that turn can already be used once.

Destroying buildings and roads

Buildings and roads can also be destroyed by their owners. This might be important for exploiting every field, since neighbouring fields can’t be built upon.

When production buildings are dismantled, half of the building materials of each type is reclaimed and the building is changed to a junction. The reclaimed resources can already be moved or used in the same turn.

When junctions are dismantled, they also give back half the building materials (1 wood and 1 stone). They can only be dismantled, if the final state won’t violate any of the other rules, so there can’t be any resources on the field, nor anywhere along the path that the field will be part of after the junction is removed. A junction that is not connected to any roads can’t be dismantled.

Junction foundations are just erased and don’t give back any resources.

Road segments can be erased, which doesn’t give back any resources. The only limitation is that erasing the road isn’t allowed to leave a junction foundation, or any resources on a field with no connection.

Production

The players have to use their buildings to create basic resources, then process those into more advanced resources, and gain score.

The player has to explicitly command their buildings to be used. Some buildings need resources to run, they can only be successfully used if the needed resources are on the field, in which case they are removed. The mining type of buildings (quarry, coal mine, iron mine), can only be used until the field they are on is exhausted. Each stone, coal, and iron ore field only contains 40 of its resource type, once all of that has been mined from the field, it can’t be used for mining.

If the building was successfully used, it will run for a set amount of turns, the number of turns for each building can be found in the building table. After that many turns, the building finishes at the end of the turn. (If the building run time is 1, it means it finishes at the end of the turn it was started in.) When the building finishes, there are 3 possible results. If the building is a market, it gives 1 point to the owner, increasing their score. For other production buildings, it produces the resource, but if the field with the building already has 4 resources, the new resource can’t be stored, and so it is wasted. In any case, the building is free again, and can be used next turn.

Resources are only created this way, and by dismantling buildings. They can only disappear from the game world by being used, either by a production building, or for constructing something. There is no other way to get rid of resources.

Territory ownership

Whenever the player successfully builds a junction, they claim the territory in a square of "radius" 3 from the new junction. (That is, a 7 by 7 square with the junction in the middle.) Any territory in that square that no one owns yet is now theirs until the end of the game.

Buildings and roads can only be built on owned territory. The players already own the territory belonging to their starting buildings at the start of the game. (These territories won’t overlap between the two players.)

Initial state

To start off the players’ economy, at the start of the game, they will each have a quarry (on a stone field), 3 wood, and a road segment connected to it. (Except in the manage subtask, where a complete economy is there already.)

Maps will be symmetric, so the same amount of resources can be reached by both players, and they will have the same amount of starting buildings/resources as well. Still, the starting player has a slight advantage, so matches will be set up in a way where all teams start in about half of their matches.

Communication protocol

The communication protocol is completely text based, through one bidirectional TCP connection. TODO: more TCP if nsz wants

The protocol is based on lines, any line separator works, empty lines are ignored. Each line is separated by spaces into words, and the first word is always one letter. This first one letter word gives the command. Here is a summary table showing all possible commands:

letter	command name	format
Commands only sent by server		
G	new game	G [player number]
Z	map size	Z [width] [height]
L	map ground data	L [one line of ground data]
C	countdown to game start	C [seconds left]
T	start of a players turn	T [turn] [player]
F	end of a players turn	F [turn] [player] [time_left]
X	error executing a player command	X [4 letter error message]
P	building finished producing a resource	P [x] [y] [resource type]
W	building wasted a resource	W [x] [y] [resource type]
S	player score increased	S [x] [y] [new score]
Commands sent only by player		
T	finish sending commands for previous turn and declare next turn to send commands to	T [turn]
Commands sent by both player and server		
M	move resources	M [x] [y] [resource(s)] [direction]
B	build building	B [x] [y] [building type]
D	dismantle building	D [x] [y]
R	build road segment	R [x] [y] [direction]
E	erase road	E [x] [y] [direction]
U	use production building	U [x] [y]

We’ll detail these commands based on the phase of the game it can happen in.

Connection phase

This phase is so that players have some time before the match to set up. When someone connects, they get the initial state of the game. The initial state is in a format like this:

```
G 0
Z 14 7
L GWCGGGGGWCGGGG
L CGGGGGGCGGGGGG
L GWSGCWGGWSGCWG
L WWGSGWWWWGSGWW
L GSCGIGCGSCGIGC
L GICGSGGGICGSGG
L GIGGWGGGIGGWGG
T 0 0
B 3 3 S
R 3 3 E
F 0 0 400
P 3 3 W
P 3 3 W
P 3 3 W
T 0 1
B 10 3 S
R 10 3 E
F 0 1 400
P 10 3 W
P 10 3 W
P 10 3 W
```

The parts of the initial state are:

- "`G 0`" means that you are playing as player 0 in this game. The two players are player 0 and 1.
- "`Z 14 7`" means that the map in this game is 14 by 7.
- The next `height` lines are the ground data of the map, in `L` commands
- After that comes the "0th turn" of the game, which details the starting buildings, roads and resources for each player. It can only contain `B` and `R` commands in the action part of the turn, and only `P` commands in the end of the turn. These `B` and `R` commands work slightly differently from normal turns, because they don't consume resources.

After the initial state, there is a countdown to the beginning of the first turn in the form of "`C <second>`" commands.

During this connection phase the players don't have to send any commands to the server, but they are allowed to send commands to their first turn, as specified in the next phase.

Game phase

During the game, the server broadcasts what is happening in the game to both players (they receive exactly the same messages including error messages). The players send their commands turn by turn. If a player connects during this phase, they won't get the current state of the game, they should connect during the connection phase.

Server broadcast

The protocol was designed in a way that following what is happening should be relatively easy, even if it makes the rules a bit more complicated.

The server broadcast is separated to turns by `T` and `F` commands. For example the start of the 1st turn of player 0 is marked with "`T 1 0`" in the broadcast, and the end of the 5th turn of player 1 is marked with "`F 5 1 15`", where the last number is the remaining time left on the chess clock of player 1 truncated to seconds.

The first part of each turn, marked by the `T` command, is the player actions. These are the same commands the player sent for that turn, however the server broadcast only contains the successfully completed commands.

If the player sent a command that is against some rule, instead of the command, an `X` failure message will be sent. Also, their turn will end, and the rest of the commands sent for that turn will be ignored. So, if an `X` command is sent on the broadcast, it will always be followed by an `F` command.

The first phase of a turn can end when:

- the player sends a new `T` command,
- the player sends a failed command,
- the player ran out of time for the turn or for the game.

The second part of each turn, marked by the `F` command, is when the production buildings' timers tick. If they are finished, the three possible results are marked with the command `P`, `W` or `S`. Either means that the building can be used again next turn.

- "`P 2 2 S`" would mean that a new stone resource was produced at the coordinates (2, 2), and it can be moved or used from next turn. This is the normal way new resources are produced, but the player command `D` can also place resources on the map.
- "`W 2 2`" would mean that a resource is wasted at coordinates (2, 2), because the field is full, and can't store the produced resource. The building can be used in the next turn again. The production inputs and mining capacity used are not given back after a wasted product.
- "`S 2 2 10`" would mean that the market at coordinates (2, 2) has finished selling a product, and so can be used again. It also informs everyone, that the score of the player, whose turn it is, is now 10 points.

Player commands

Before the player sends commands for a turn, they must mark which turn they want to execute those commands in. So, normally, if they want to do any actions in their first turn, the first thing they would send is "`T 1`". The `T` command is also used to mark that they are done sending commands for the previous turn.

The server only caches player commands for 1 turn at a time for each player. This means, that if you sent commands for your turn N, you shouldn't send commands for another turn until you've seen "`F [N] [player number] [time_left]`"

These rules means that the normal flow of game for the players for turn N is:

- At the end of the previous turn, they already sent "`T [N]`".
- They wait until they've seen "`F [N-1] [player number] [time_left]`" in the server broadcast.
- They send their actions for turn N.
- They send "`T [N+1]`" to mark that they are done for the turn.

For the player actions, we'll need designations for the 4 main directions:

- N - north, -y
- E - east, +x
- S - south, +y
- W - west, -x

First we'll list the normal behaviour for each action, then there will be a big table with the possible failures, which are also a quick reference for the ways a command can be against the rules.

- "`M 2 2 WP N`" would mean moving a wood and a product north from (2, 2). The resource part can be one or two letters, depending on whether the player wants to move one or two resources. If successful, it means that a wood and product, that haven't been moved in this turn yet, will be removed from (2, 2), and added to (2, 1), and they can't be moved again this turn.
- "`B 2 2 W`" would mean constructing a lumberjack at (2, 2). If successful, the resources necessary for a lumberjack (4 stone) will be removed from the field, none of which was moved yet. The junction will be changed into a lumberjack. If constructing a junction, another effect is claiming the unclaimed territory around it for the player.
- "`D 2 2`" would mean dismantling the building at (2, 2). This can mean slightly different things if successful, depending on the building there. If it's a junction, or a junction foundation, there will be no building remaining. If it was a junction, there will be 1 wood and 1 stone placed there. If it was a production building, it will be replaced by a junction, and half of the construction resources of each type will be placed there. In any case, the recovered resources are new, and can still be moved once this turn.
- "`R 2 2 N`" would mean constructing a road segment from (2, 2) to (2, 1). If successful, it will remove a wood and a stone from (2, 2), and connects (2, 2) and (2, 1) by a road segment. (The wood and stone can't have been moved that turn.)
- "`E 2 2 N`" would mean erasing the road segment from (2, 2) to (2, 1). If successful, it will just remove that road segment.
- "`U 2 2`" would mean using the production building at (2, 2). If successful, it will remove the necessary resources from the field. It will also render that building unusable for the given amount of turns, until a `P`, `W` or `S` is sent for it.

Error code	**Description**
General errors	
`X SYNT`	Syntax error, that command couldn't make sense at any stage in the game

`X NOWN`	Trying to execute an action in a field that is not owned by the player
`M` errors	
`X MRES`	No resource of the correct type that can be moved is on the field
`X MROD`	The road to be used either isn’t built, or was already used this turn
`X MFUL`	The field you wanted to move to is full, it can’t take the resource(s)
`X MPBS`	Path is busy, there are already resources somewhere along it
`B` errors	
`X BNFR`	When trying to build a junction foundation, the field or one of the neighboring fields isn’t empty.
`X BJTF`	Trying to build a junction on something other than a foundation
`X BBJT`	Trying to build a production building on something other than a junction
`X BRES`	You don’t have all the resources needed to build the building
`X BTMR`	Too many resources would be left on the field after completing the building
`X BGND`	Trying to build mining building on wrong type of ground
`D` errors	
`X DNBD`	There is no building to be destroyed
`D` errors specific to junctions	
`X DDIV`	Junction can’t be destroyed, because path would divide at the spot
`X DORP`	Junction can’t be destroyed if it’s an orphan, that is, if no roads lead there
`X DRES`	Junction can’t be destroyed if there are resources on the spot (because the 2 resources from the dismantling would put it over capacity
`X DPBS`	Path is busy, there are already resources somewhere along it
`R` errors	
`X RDIV`	Road can’t divide (field can’t have more than 2 roads) at a spot with no buildings/junction;
`X RBLT`	Road is already built
`X RWTR`	Can’t build road into water field
`X RTWS`	Can’t build a path from two sides
`X RRES`	Don’t have the resources to build road there
`E` errors	

`X ERNB`	Road isn’t built, so can’t be destroyed
`X ERFO`	Would leave a foundation orphaned on a field with no roads
`X ERRO`	Would leave some resource orphaned on a field with no building and roads
`U` errors	
`X UNPR`	Trying to use a field with no production building
`X URUN`	Trying to use a building that is already running a production
`X URES`	You don’t have the needed resources to use your production building
`X UOUT`	Mining field is out of resources, it is exhausted, so can’t be mined

Input

The graphics created for our public display, which are used throughout this document are in the input files package. They can be used to create graphic GUIs for the game without having to draw something. Of course, those who prefer a command line interface can freely ignore these images.

G. Slothlers - Manage (500 points)

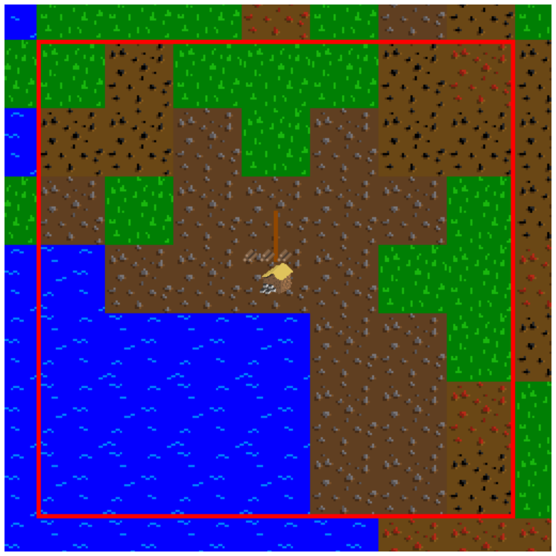
In this task your team is provided with the full infrastructure for a succesful civilization, you just need to run the economy and sell 40 product before the time is up. If the required amount of products are sold, score is awarded.

Scoring is similar to an input of an algorithmic problem: if enough products are sold, you get the 500 score scaled with time passed since the start of the contest.

This is a single player practice server for testing the protocol and the economy part of your AI.

In this subtask, the game ends when your team disconnects. Unlike the multi player servers, whenever the game ends, the server restarts in a few seconds, and if you connect again, the economy will be reset to the same initial state.

H. Slothlers - Produce (500 points)

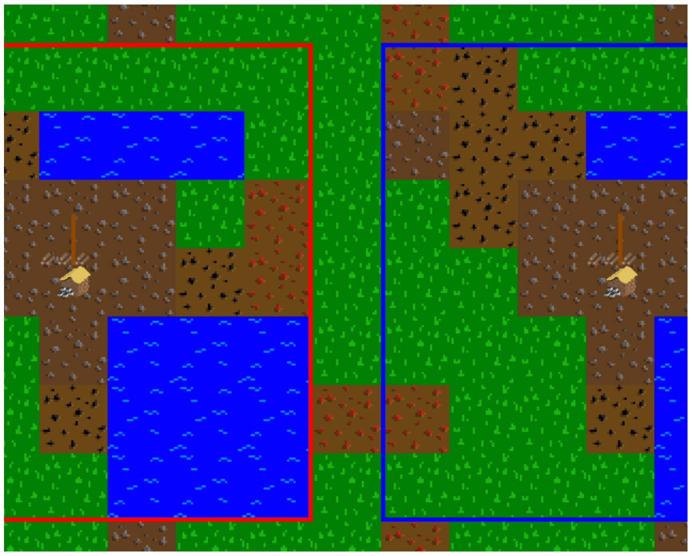


You are alone on a map with the standard startup kit. You have to build your economy from the ground up and sell 50 product within a fixed amount of turns to earn score.

Scoring is similar to an input of an algorithmic problem: if enough products are sold, you get the 500 score scaled with time passed since the start of the contest.

This is a single player practice server for testing the protocol and the building part of your AI. Unlike the tournament servers, these games end when your team disconnects. Whenever the game ends, the server restarts in a few seconds, and if you connect again, you get a new map in starting position.

I. Slothlers - Tournament (3600 points)



In the final task about slothlers, the teams have to compete with each other in running their economy. During the competition, 8 tournaments will be held, where the players can demonstrate their economy skills.

Each tournament will be held in the Swiss tournament system, in 10 rounds. The first tournament starts at the beginning of the competition, each lasts 2 and a half hours, and there are half hour breaks between them. When a tournament is running, a new round of matches starts at 00, 15, 30 and 45 minutes into the hour. There is a connection period for a minute, and the first turn starts at 01, 16, 31 or 46.

The state of the current tournament, and the history of past tournaments can be followed in the public files available on our servers. Your team always needs to connect to the server on the same port, and the matchmaking is done on our side.

At the end of a tournament, scores are distributed based on the final ranking. The 1st prize for the first tournament is 100 points, which increases linearly to 800 points for the last (8th) tournament. Lower ranks get linearly less points, with the last (30th) place getting zero points. In case of a tie, each tied team gets the score corresponding to the lowest rank, so for example all the teams that didn’t win any games in the tournament will tie for last place, and get no points.

J. Sonar (about 5000 points)

Sloth populations are like ice bergs: only a small portion of the sloths live above sea level. At least this is what the new generation of sloth researchers are claiming when they apply for funds for expensive ~~toys~~ field research equipment. They believe underwater sloths live obviously in the deep sea (this why no one has ever seen them). There's a race between these youngsters to find the first deep water sloth.

http://www.flickr.com/photos/primevalnature/3193816366/

Unfortunately they are not very good in sensors, navigation, programming or anything else that'd be required to operate their submarine drones. Senior researchers realized the outstanding importance of this field, so they set up underwater test arenas for the submarine operators to practice in - and you decided to help the juniors with their drone control issues.

You control submarine drones equipped with:

- a 4 microphone array
- an underwater loudspeaker
- north-south and east-west thrusters and velocity sensors
- an underwater mine ejector system, loaded with a single mine

Your task is to navigate arenas based on sound only, avoid the walls, find the treasure, and blow up the opposing team's drone on occasion.

There are two separate arenas with enough place for 15 drones in them. In each 10 minutes, a new round begins with new, unknown walls installed in the arenas, the drones repositioned and loaded with mines, and treasure added. (In each round, the teams are randomly sorted between the arenas.)

Treasure is marked with beepers. In each round, treasure is placed in 8 different positions in each arena. Teams don't take the treasure, just find them (so even if a drone finds a treasure point, it remains available for other teams). If your drone is destroyed, you lose all the treasure you found so far in the round.

Collision with walls will destroy your drone. The drone is immediately moved to a random position in the arena, and you lose all treasure found. There is **no collision between drones** or between a drone and a treasure, or other any source of noise.

Drones **send back 4 channel audio** captured by their microphone arrays, sampled at 5000 Hz. The 4 channels come from 4 microphones, located north, east, south and west from the drone body. Drones keep a **constant orientation** (they don't rotate).

A drone's **loudspeaker** can be used to bounce sounds off surrounding walls, therefore determining their location. The loudspeaker needs a lot of **energy** - this energy is replenished constantly and freely, but can be used up very quickly by sound emission. The amount of used energy **depends on the amplitude** of the emitted signal. (Therefore drones are capable of emitting short, interrupted sounds loudly, or continuous sounds quietly.)

Drones are mostly silent, but the **thrusters** emit some noise when used. **Only walls reflect waves**, small objects (such as drones, mines and treasure) don't.

Each drone carries a single **proximity mine**. After releasing the mine, the mine will stay in the same position and arm itself in a few seconds. Before arming, the mine emits quick beeps (a different sound from treasure); after it's armed, it still beeps, but much slower.

When a mine is armed, it will blow when a drone comes within a certain distance to it, destroying the drone (which will then be moved to a random position, and all treasure is lost). Drones are naturally not immune to mines released by themselves.

Scoring

At the end of each round, your team will receive a number of points for found treasure and other drones blown up (and then the scores are reset before the next round). Teams are scored independently. Only the end of the round is considered for scoring (if you collect treasure, then lose it by hitting a wall, then you don't get points for it; the "kill count" achieved by using mines is however not reset with drone crashes).

Each collected treasure is worth **5 points** at the end of each round. Blowing up an enemy drone is worth **20 points**.

Protocol

Teams control their drones by sending and receiving **UDP packets**. All data is **binary**, and represented in the **little endian** format. Fields listed in the packet descriptions follow each other immediately, **there is no padding**.

All **audio data** is represented using 4 byte floats for samples, clamped between -1.0 and 1.0, sample rate 5000 Hz.

There are only two packet types: the one sent by the team to the control server, and the one sent by the server to the teams.

Used field types

- **u8 -** 8 byte unsigned integer (uint64_t)
- **u4 -** 4 byte unsigned integer (uint32_t)
- **float -** 4 byte IEEE 754 float

Team to Server

- **u8** `seq` - sequence number
- **u4** `mine_release`
- **u4** `samples` - number of samples following in the `sample_data` array

- **float** `accel_x` - west to east acceleration
- **float** `accel_y` - south to north acceleration
- **array of float** `sample_data` - audio to play

Sending this packet to the server automatically subscribes for drone update data for one second. If the team doesn't send a packet for one second, then the updates will stop.

The **sequence number** is expected to monotonously increase. Packets with decreasing or identical sequence numbers are ignored. If the team drops the subscription by not sending a packet for one second, then the expected sequence number is reset. The first packet sent has to have a sequence number of **at least 1**, otherwise it will be ignored.

If **mine_release** is larger than a value received in a previously received valid packet, then the proximity mine is deployed.

accel_x and **accel_y** should be clamped between -1.0 and 1.0. Positive x accelerate the drone towards east, while positive y accelerates towards north.

`samples` should contain the number of samples to be played on the loudspeaker. The sample data itself should be loaded in the `sample_data` array (added to the end of the packet). `samples` may be 0, of course. Audio to be played is added to the end of the drone's outgoing audio buffer. The drone plays audio at a 5000 Hz sample rate.

Server to Team

- **u8** `seq` - sequence number
- **u8** `last_seq` - last seen incoming sequence number
- **u4** `kills` - enemy drones sunk in this round
- **u4** `blown` - times blown up by mines in this round
- **u4** `collisions` - wall collisions in this round
- **u4** `treasure` - treasure collected since the last crash in this round
- **u4** `mines_available` - proximity mines available (1 or 0)
- **u4** `emitter_buffer` - free space in the outgoing audio buffer (sample count)
- **u4** `samples` - number of samples following in the `sample_data` array
- **float** `velocity_x` - west to east velocity
- **float** `velocity_y` - south to north velocity
- **float** `emitter_energy` - available loudspeaker energy
- **array of float** `sample_data` - audio from the microphone array

If the server received a valid packet from the team in the last second, the server will keep sending the packets defined above (at a rate of about 20 Hz).

The **sequence number** will be monotonously increasing (throughout the contest).

If the `blown` or `collisions` values are increased compared to a previously received packet, then the drone had been moved to a random position.

`emitter_buffer` returns the amount of free space in the drone's outgoing audio buffer. This may be used to implement continuous playback. If too much audio is sent, it will silently overwrite old data in the buffer.

`emitter_energy` is decreased by every played sample, and is otherwise continuously replenished up to a set maximum. If emitter energy is depleted during playback, the drone's outgoing audio buffer is cleared.

`velocity_x` is positive when the drone is going east, while `velocity_y` is positive when the drone is going north.

`sample_data` is an array that contains **samples * 4 floats**. The 4 microphones in the microphone array capture the same amount of samples, which are interleaved in the `sample_data` array in **north, east, south, west** order. `samples` indicates the number of groups of floats in the array, so the total amount of data in the array is `samples * 4` floats.

Control Servers

There are **two servers**, one for each arena. The servers are visible through two different ports on `server.ch24.org`. For each round, each team is assigned to one of the two servers randomly. Only the server that the team is assigned to will answer their packets; the other server will (safely) ignore them.

There is **no way for a team to know which server they're assigned to** in the beginning of a round, so **you have to send packets to both servers** (both ports) until one of them answers.

When a round ends, the servers will stop communicating for about 15 seconds. During this time, the scores are recorded, the arenas are rearranged, and teams are resorted between the servers. This means that if you lost the connection for a few seconds, you have to start trying to contact both servers, because you may have been moved to the other one.

When the servers start communicating again, a new round begins in a new environment. Rounds will always begin at "round" times (09:00, 09:10, 09:20 and so on), and end roughly 15 seconds before the beginning of the following round. (In case of a malfunction in the server infrastructure, rounds may be skipped, but following rounds won't be shifted - they will start at the next suitable "round" time.)

KLM. OSM

Sloths don't use GPS to navigate in their forest. They simply remember the map of the whole world and trace how much they move in which direction.

This set of tasks will give you an impression of their storage and processing capabilities. You've received two large files (and a small patch file). These files are a filtered extract of the OpenStreetMap project, containing ways and the corresponding nodes only, for the whole world. The large files are compressed with gzip, which enables decompression while reading, block by block.

source:
http://spaceflight.nasa.gov/gallery/images/apollo/apollo17/html/as17-148-22727.html

Coordinate system

The Earth is a perfect sphere with `R = 6371.0 km` radius. Nodes are positions on the surface of the Earth and they are given by *latitude, longitude* coordinates in degrees (-90 ≤ latitude ≤ 90, -180 ≤ longitude ≤ 180).

A way is a path on the surface and it is given by a sequence of nodes.

Input

Node coordinates are listed in input_nodes.txt.gz. Each line is a node with an integer node ID and a floating point latitude and longitude in decimal format. Ways are described in input_ways.txt.gz, one way per line. The first column is an integer way ID, the next integer is the number of nodes the way consists of, the rest of the line lists the node IDs.

There's a separate nodes_patch_0.txt, in uncompressed form, which has the coordinates of a couple of nodes missing from the original input_nodes.txt.gz; please process these nodes along with input_nodes.txt.gz.

Distance Formula

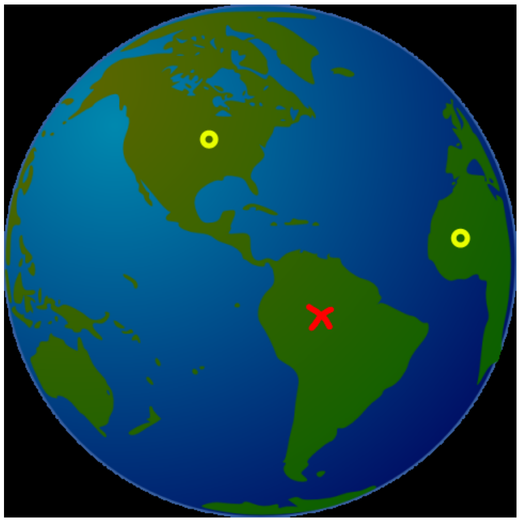
Most calculations can be done approximately, double precision floating-point arithmetics should give more than enough precision. We give a formula for the distance between two nodes, but other methods may be used as well for the calculations:

```
function distance(lat0,lon0, lat1,lon1)
{
        p0 = xyzcoords(lat0, lon0)
        p1 = xyzcoords(lat1, lon1)
        n01 = cross(p0, p1)
        c = dot(p0, p1)
        s = sqrt(dot(n01, n01))
        return R * atan2(s, c)
}

function xyzcoords(lat, lon)
{
        x = cos(lat*pi/180)*cos(lon*pi/180)
        y = cos(lat*pi/180)*sin(lon*pi/180)
        z = sin(lat*pi/180)
        return (x,y,z)
}
```

Where `cross` and `dot` are the cross and dot products of (x,y,z) vectors, `sin`, `cos`, `atan2` and `sqrt` are the usual mathematical functions, R is the radius of the Earth and `pi` is pi.

K. OSM - Search (1000 points)


source: http://www.clker.com/clipart-2123.html

Sloths have a special instinct for sensing gold. If they'd ever be interested in collecting gold, they could hoard up 95% of the world's gold reserves within 24 hours. Of course solving algorithmic problems is much more interesting for sloths so they just ignore this skill of theirs.

However, a friendly sloth has offered her API to this instinct (on a strictly voluntary basis), so she can help you find the treasures.

Treasure is always hidden near a node of the OpenStreetMap database. In a query you shall submit the ID of two nodes and the sloth will return them ordered so that the first returned ID is geographically closer to the gold. If any of the two nodes matches the treasure-node, the treasure is found.

Your task is to find 20 treasures by issuing the smallest amount of queries.

Protocol

The eval server provides a plain text, line based TCP socket you need to connect to. Each line is a message and is terminated by a single newline character ('\n').

The server announces each treasure with a "start *I*", where *I* is the ID of the current treasure. After that the client can start sending queries. Each query consists of two unsigned integer node IDs in a single line, terminated by a '\n'.

In case *N1* or *N2* hits the treasure, the server sends a "success" message.

Otherwise the server answers queries with the message "closer *N1 N2*", where *N1* and *N2* are the two nodes of the query ordered such that *N1* is closer to the treasure.

If the treasure is not found within 200 queries, the server announces the failure with a "fail" message and sends the target node ID using the "solution *ID*" message.

The server keeps the connection open until the first failure or until all the 20 treasures are found successfully or another client connects from the same team.

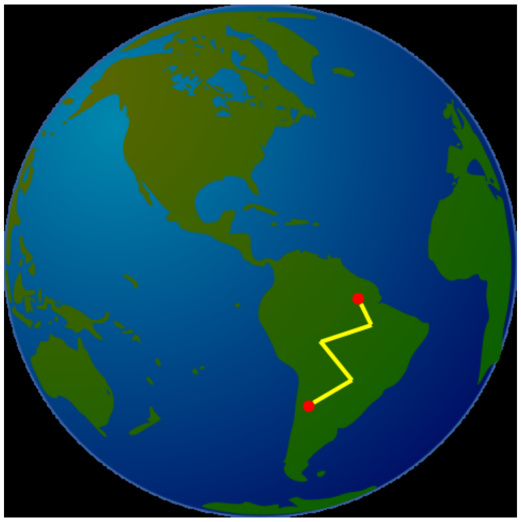
Scoring

If all the treasures are found successfully then the awarded score is

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

where *Q* is the number of queries made during the treasure hunt and *BEST* is the best submission.

L. OSM - Path (1000 points)


source: http://www.clker.com/clipart-2123.html

Of course a path finder algorithm running on a silicon based computer can bever be as fast as the ones in the brains of sloths. Nevertheless, before you start solving Problem M, the race game, it might be useful to play around a bit with path finding in the OSM database.

Given two nodes in the OpenStreetMap database, find any path between them along the given ways.

Input

A source and a target node.

Output

A list of node ids on separate lines each, where consecutive nodes are on the same way and the first node in the list is the given source node and the last node is the given target node.

Example input	Example output
`826166335 257726479`	`826166335 101497069 257726480 257726479`

M. OSM - Race (3000 points)

A random citizen far away has reported something strange: a sloth that can not program in functional programming languages! This is a very rare and exciting opporunity so you decide to abandon even the International Sloth Conference to race there and examine that special sloth.

Of course you are not the only one on the conference who is interested in such oddities - there are potentially 29 competitors starting from the same parking lot with their cars trying to get there faster. This can cause quite a traffic jam that slows cars down.

The sloth who can not program; photo source: http://pixabay.com/en/sloth-mammal-animal-cost-rica-318882/

During the contest a new race will be started at every hour. There will be a 5 minutes connection time after a race is started when clients can connect to the race server at the specified TCP port (see table in the introduction). A race lasts at most 50 minutes, during which teams should get from a given source node of the OpenStreetMap to a given target node stepping along the ways in several rounds. The first team who arrives at the target gets the maximum score for the race.

Protocol

Server sends plain text messages ending in ’\n’. Each meassage is a command word followed by zero or more parameters separated by space.

Connection period

During the connection period the server sends a count down message every second

```
start S
```

where the **S** parameter is the remaining seconds until the end of the connection period.

Rounds

The connection period is followed by **R** rounds of the race. The server announces a round with the

```
round K R G
```

message, where **K** is the index of the round that goes from 0 to **R**-1 and **G** is the target node id (same for the whole race). Then the server sends to each client its id **ID**:

```
yourid **ID**
```

And the positions of each racing teams:

```
pos ID N
```

meaning that team **ID** is at node **N** on the map. (When the race starts all teams will be on the same node).

One round takes 3 seconds during which clients can send their movement for the round. A movement consists of a sequence of steps to adjacent nodes along the ways of the OpenStreetMap. At most 100 steps can be made in a round. The format of the client message is a sequence of node ids, each on a separate line.

After the clients sent their movement the race server evaluates them in the following way: For invalid movements the server sends either

```
error message
```

or

```
warning message
```

to the client, in case of an error all the client steps are discarded for that round, in case of a warning the server continues processing the steps made by the client. The server calculates the positions of all clients after each step. In case two or more teams are on the same node after a step then they "collide" and one step is discarded from their 100 movements (the last not yet discarded step if there is any such step). The server sends a warning message to the collided teams.

If a team with id **ID** arrives at the target node id **G** in a step then the server announces it as

```
arrive ID STEPS
```

where **STEPS** is the number of steps it took for the team to arrive.

The race ends when all teams arrived all there are no more rounds.

Scoring

Each race has a max score

race start	max score
9:00, 10:00, 11:00, 12:00, 13:00, 14:00	50
15:00, 16:00, 17:00, 18:00, 19:00, 20:00	100
21:00, 22:00, 23:00, 00:00, 01:00, 02:00	150
03:00, 04:00, 05:00, 06:00, 07:00, 08:00	200

Every successful team gets at least half of the max score for the race. The team that arrived first gets the max score, the other successful teams get a scaled score based on the order they arrived in.

N. Dog tag (1000 points)

Sloths are born with a dog tag: a special bone that grows in their neck, the *ID bone*. This little piece of bone is more solid and homogeneous than their average bone tissue. Information is conveyed in little bubbles (chambers filled with gas) embedded in this bone. This makes it very easy to identify any dead sloth during an autopsy by slicing the *ID bone*.

source: http://commons.wikimedia.org/wiki/File:Skeleton_of_a_Three-toed_Sloth.jpg

Sloth research has reached a point where it became essential to be able to acquire those IDs. Large amounts of sloths are being slaughtered at this very moment in order to provide *ID bone* samples for researchers who are developing the ultimate software to decode bone slices. You can save a lot of sloths by proving the task can be solved without dissecting thousands of sloths for their ID bones.

Simulated *ID bones* have been sliced up, digitized and vectorized for your experiments. Your task is to extract the information stored in these IDs and return the bits in an ASCII file.

The bone may have a lot of smaller and larger bubbles, some of them encoding bits while others are mere noise (it's very hard to grow 100% solid material!). Looking at the volumes of the bubbles it is easy to decide which one is which:

purpose	volume	remarks
fiducial	~ 8 * V	there are only three of these
bit 0	~ 4 * V	
bit 1	~ 1 * V	
noise	<= 0.1 * V	some input bones won't have noise

NOTE: V is an unknown volumetric unit that may change from input to input.

Slicing happens at a random angle (can't align the saw without seeing into the bone), while center-of-mass of the bit-bubbles are aligned into a 3d *grid*. The three fiducial bubbles are perfect spheres and are always placed on the first plane such a way that they form an isosceles rightangle triangle. The closest bubble to the fidu bubble at the rightangle is the first bit of the first plane. Thus the input slices are given in an X' Y' Z' *slice coordinate system* while output must be read out using an X Y Z *grid coordinate system* (see the figures below for details); the connection between the two systems are determined using the fiducial bubbles.

The solution has to be read out plane by plane, first among increasing X, then among increasing Y and finally by increasing Z (in the *grid coordinate system*). Slices are spaced evenly at an unknown but constant distance from each other (this distance may vary between inputs).

The last 4 bits of every data plane is a simplistic checksum. Checksum state consists of two integers: a sum and an accumulator, initialized to 0 before each plane. Bits are shifted into the lowest bit of the accumulator in order of appearance:

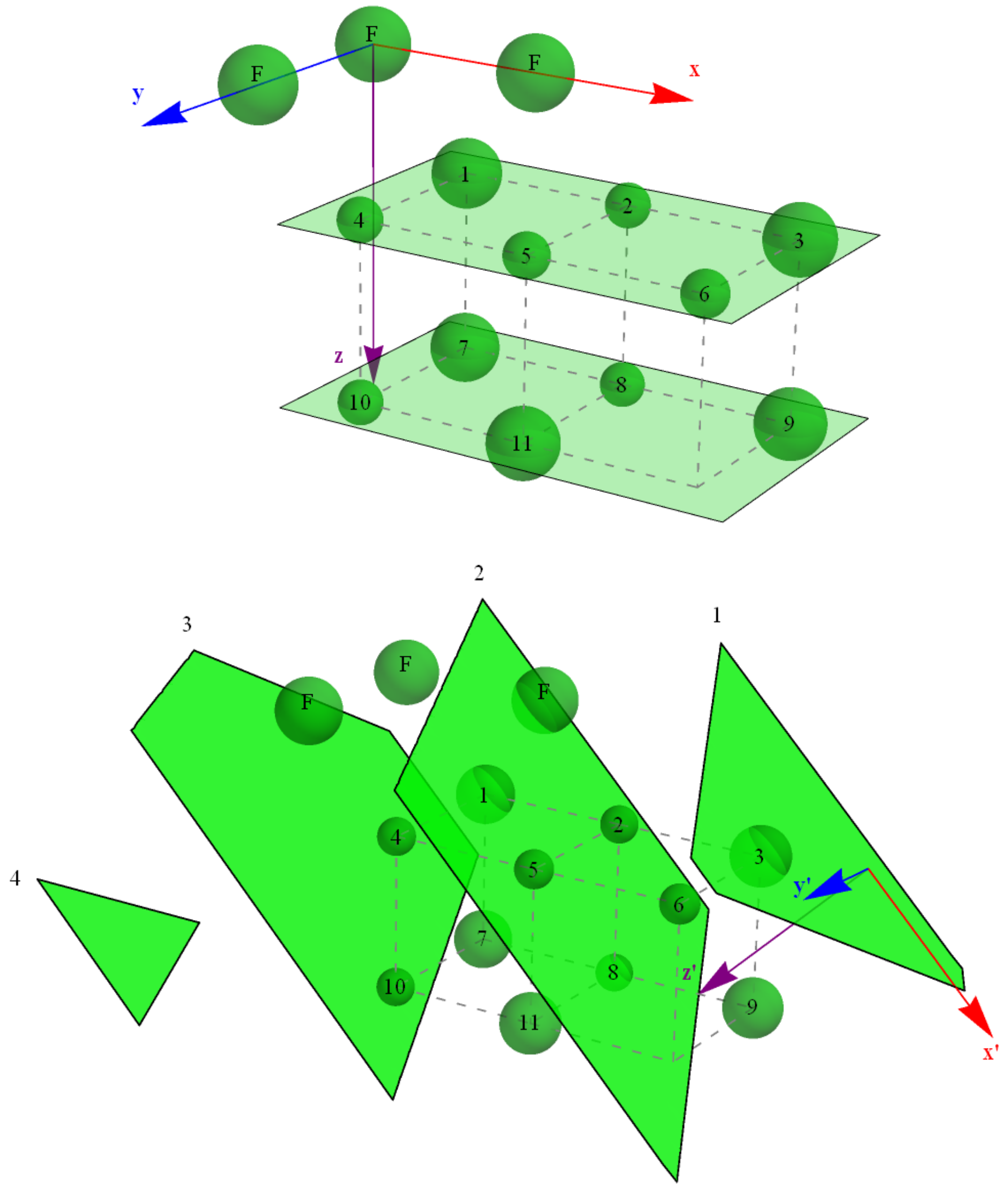
```
    accum := (accum << 1) | input_bit
```

After every 4 bits, the sum is updated using the following formula, and the accumulator is zeroed.

```
    sum := ((sum >> 1) + ((sum & 1) << 3) + accum) & 0xF
```

If the end of the useful data in the plane is reached (before the first bit of the checksum at the end), and we have shifted some bits in the accumulator that we haven't used for a checksum update yet (1 to 3 bits "left over"), then we use the current value in the accumulator to update the checksum (using the above formula) one last time.

The final value of the sum will comprise the checksum, written as 4 bits, from MSB to LSB.



tag (upper) and slice (lower) coordinate systems

Legend: fiducial bubbles are marked with F, data bubbles are marked with an integer to illustrate the order of reading out the bits. There are no noise bubbles on this illustration and data bubbles happen to be perfect spheres. To make the illustration easier to read, only every 10th slice is presented; when all slices are considered it's guaranteed that all data bubbles and fiducial bubbles are crossed by at least one slice. X Y Z is the aligned *grid* system for reading out the data; X' Y' Z' is the *slice* system (inputs are given in that).

Input

First line of the input is an integer *S*, the number of slices. The next *S* blocks describe each slice. The first line of a slice is *L*, the number of loops on that slice. Each loop describes the contour of a bubble as seen on that slice, in the next *L* lines: the first integer *N* is the number of vertices, followed by *N* pairs of x;y coordinates for the vertex. The vertices are given in CW or CCW order.

Input coordinate system: x increases from left to right, y increases from top to bottom on each slice; slices are ordered from lower z to higher z (so that the top slice is described first in the file).

Output

The encoded data is a 3d bit matrix in the original *grid* system. The matrix is a sequence of planes (with different Z coordinates) and each plane is a sequence of rows (with different Y coordinates) and each row is a sequence of bits (with different X coordinates). The output is the plain text sequence of zeros and ones one row per line and planes separated by an empty line. The last plane may be incomplete and may end in an incomplete row.

The ordering and orientation of the bits is shown in the figures above as well. (The decoded bits should appear in the same order as they are indexed on the figure)

Example input

Please refer to 0.in in the input directory.

Example output

```
1010
1011
0000
0000

1010
1011
1000
1
```

O. Ball (4000 points)

Sloths usually do not hurry while moving around. For some researchers this is a great challenge: they need to change the location of a sloth in the cage they keep him in without touching him (to avoid contamination of the fur). The common method is to wait until the sloth curves into a perfect sphere and goes to sleep then tilt the cage so that the sloth rolls where they need him.

Your task is to develop software that can reliably move the sloth around in a cell with an imperfect floor. There is a model of the cell equipped with servos; the sloth is modelled with an orange ball (lab sloths are on strike this week anyway).

Sloth rolled up into a little orange ball. source: http://commons.wikimedia.org/wiki/File:Choloepus_didactylus_2_-_Buffalo_Zoo.jpg

scripting the motor control

There are DC motors driving the mechanism. The output states are speedX and speedY with integers between -127 and +127, proportional to the voltage the motors get (to the speed they change the angle of the table at). Once a script command changes the voltage, it remains the same until another script command changes it again.

The script also has three input channels: X and Y angles and time. Time is a 12 bit unsigned integer increased at about 4 Hz in real time. X and Y are implemented as incremental optical sensors and are represented as 9 bit unsigned integers. The table is horizontal at an arbitrary X and Y angle measured by the sensors - this info will be made available during the contest.

The device has a script memory where 8 statements can be stored in an ordered list of slots. The user may upload a new command into any slot at any time asynchronously. Each statement has the following parts:

- slot ID
- break bit
- a zero (for compatibility reasons)
- condition (cond and cond-arg)
- command
- command argument

The main loop of the device first waits until any of the three inputs change then executes the script. Script execution takes each slot starting from slot 0 and evaluates the condition of it; when the condition doesn't match, evaluation simply continues with the next slot. Upon a match, the command of the statement is executed (so that speedX or speedY can be changed) and then break bit of the statement is evaluated: if br is set, execution stops and no further statements are considered for this event (execution restarts from slot 0 upon the next event)

There are three valid commands:

command	**command arg**	**effect**
report	n/a	send back a time position report; argument is ignored
speedX	S	S is the new speed for the motor driving the X axis; S is an integer between -127 and +127
speedY	S	S is the new speed for the motor driving the Y axis; S is an integer between -127 and +127

It is not recommended to drive the motors below speed 30. For halt, use speed 0. When motors are operated on high speed (typically above 60), there are some overshots due to the inertia of the system: when the motor is stopped, the mechanism will still spin a few positions until the system finally halts.

Sending reports too often may crash the firmware. Sending a report every time tick is safe (see example 1).

Conditions may check time (T), or a coordinate (X or Y). For time the following two checks are available:

condition	**condition arg**	**effect**
T%	divisor	matches whenever time modulo *divisor* is 0 (i.e. every *divisor*-th iteration)
T>=	target	matches when time is greater or equal to target time (in ticks)

Coordinate conditions:

condition	**condition arg**	**effect**
X~	target	X position is close to the target
X>	target	X position is larger than target
X<	target	X position is less than target
Y~	target	Y position is close to the target
Y>	target	Y position is larger than target
Y<	target	Y position is less than target

examples

Example 1: slot 4 sends a report every iteration:

```
4 0 0 T% 1 report 1
```

4 is the slot ID, the first 0 is the *break bit* (0 means do not break when condition is met so that the rest of the script will run). The second 0 is a mandatory constant field; "T% 1" is the condition that matches every iteration (when time modulo 1 is zero). The command to be executed upon match is "report 1", which will send back a status report (1 is a dummy argument).

Example 2: the following three slots (randomly placed in the memory from slot 2) will drive the mechanics to around X position 110 and stop there:

```
2 0 0 X< 100 speedX 55
3 0 0 X~ 110 speedX 0
4 0 0 X> 120 speedX -55
```

While X position is out of the 100..120 range, command the motor to drive the mechanics towards this range at speed 55. When position gets close to 110, stop the motor.

It is possible to write more complex scripts that drive the mechanism faster when it is farther away from the target.

The returned format for the report command is

```
T X Y
```

where *T* is the 12 bit time and X, Y are the axis positions in decimal format.

Input

There is a virtual grid overlaid on the webcam's image. The grid layout will be published during the contest in PNG format (pixel offsets may change from time to time as rigid mounting of the camera is not easy). Grid coordinates are integers, the upper left cell being 0;0. Grid coords are aligned to the camera coordinate system (increasing X is right, increasing Y is down on the image).

For each input there is a different grid setup. There are forbidden cells in the grid, marked with a little red cross in the middle of the cell. If center of the ball enters such a forbidden cell, the solution is not accepted.

For each input there are one or more target cells the ball should visit in order. When reaching a target cell, the ball must stay in that cell for at least 2 seconds. The server will say "# reached" when that condition is met. The input is accepted if all targets are reached in the right order, without touching forbidden cells.

The eval server uses the same webcam stream for calculating the center of the ball. This method can not yield 100% precise ball position info due to camera resolution and noise. Teams will probably have their own image processing which may constantly report slightly different positions.

In any case, whether the ball is in the target cell or crossed the boundary of a forbidden cell is judged by the eval server position. Cells are much larger than the ball, teams are encouraged to play safe: keep the ball around the center of a target cell, still, and avoid going near to any forbidden cell.

communication protocol in the preparation phase

The control server uses a line oriented plain text protocol over TCP/IP. Line terminator is \n. Each line is a new message.

After connecting the server the player gets into a queue. A new connection from the same team replaces the existing connection without changing the place in the queue (the old connection is closed by the server). If the connection dies before a new connection from the same team the position in the queue is lost and a later connection will stand at the end of the line.

Messages sent by the server start with "##" while the connection is in the queue. Messages from the team during this period is considered a protocol error.

When the hardware is avaialble, the first connection in the queue gets redirected. This is indicated by a "## GO" command from the server. After sending this message the server redirects any messages read from the team to the hardware service. The GO message also lets the team know they should start streaming the video from the camera.

The next phase is the hardware control phase. There are in-band messages (status reports from the firmware) and out-of-band messages (eval server reports about errors or progress of solution). Out-of-band messages start with a "#"

The hardware server first says "# input?" for which the team must answer a single integer in one line, naming the input to be prepared. There is a timeout of **2** seconds; if the team fails to name the input within that time frame, the server closes the connection and a new connection gets to the end of the queue.

When preparation is ready (and the hardware calibration is over), the server says "# start *I*" (where *I* is the input number) and the slot timer starts. When the slot is over the server says "# timeup" and closes the connection. A new connection will be appended to the queue. If your control manages to keep the ball in the next target cell for the required period of time, a "# reached X Y" is sent by the server, where X and Y are the cell coordinates. This indicates that the ball should start its journey to the next target. When all target reached (the input is correctly solved) the server sends a "# accepted solution" message and closes the connection.

When the ball visits a forbidden cell, the server sends "# fail: visited forbidden zone at X Y" and closes the connection.