Problem Specification

Overview

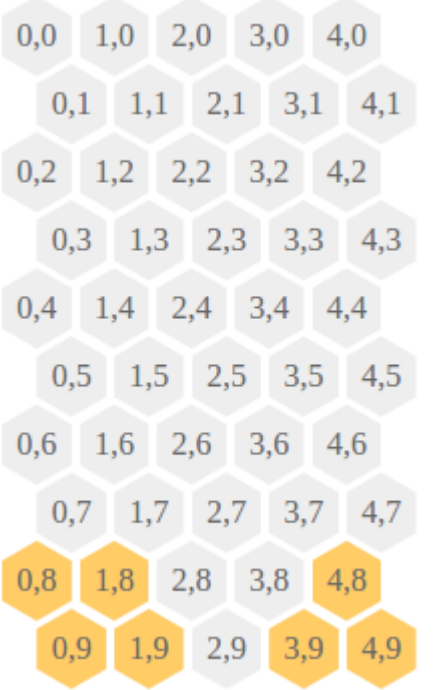
The purpose of the competition is to write a program that can play a game. The objective of the game is to score the most points by ***locking*** units into place while invoking ***phrases of power***. As input, the program will take a description of the initial game state. As output, the program needs to produce an encoding of a sequence of ***commands***. The result of the program will be evaluated by executing the commands and computing a score.

The game state has three parts:

- the ***board***, which is where all the action happens,
- the ***unit*** that is currently under the program's control, and
- the ***source***, which is the sequence of units that will be under the program's control in the future.

The program will control units, one at a time, by issuing them ***commands***. Each command is a single symbol selected from a fixed predefined set. Each unit is controlled until it becomes ***locked***. When a unit is locked, the program is awarded points, the board may get updated, and then it will proceed to control the next unit from the source. The game ends if one of the following conditions occurs:

- all units from the source have been locked, or
- there is no space on the board to place the next unit, or
- the program issues a command that causes an error.



Example board of width 5, height 10. The colored cells are full.

The Board (a.k.a., the Honeycomb)

The game board consists of hexagonal cells arranged in rows, with the first row numbered 0. The ***height*** of the board specifies the number of rows on the board.

Each row contains an equal number of cells, specified by the ***width*** of the board. The cells in a row are oriented so that they have vertices up and down, and edges to the left and right. The first cell in a row is numbered 0, thus each cell may be identified by a pair of coordinates ***(column, row)***.

When we use geographic directions we take "north" to mean "up", or towards smaller row numbers, and "west" to mean "left", or towards smaller column numbers in a row.

Odd-numbered rows are displaced to the right, so that the north-west edge of each cell is adjacent to the south-east edge of the correspondingly numbered cell in the previous row.

Each cell on the board is either ***full*** or ***empty***. Each board specifies an initial set of cells that are full.

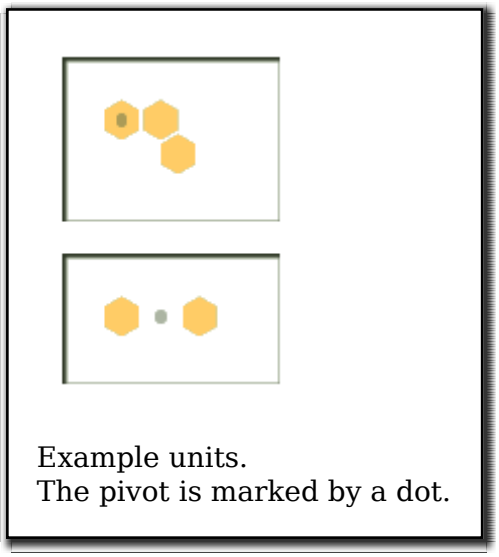
These initial board configurations may contain useful information for contestants.

Units

A unit is a non-empty collection of cells that are manipulated together, and a ***pivot point*** used for some commands. A unit is in a ***valid location*** if all of its cells are on empty board cells. Note that a unit's pivot point need not be on a board cell.

When a new unit is spawned (i.e., it emerges from the ether and comes under player control), it appears centered at the top of the board. More precisely:

- the unit appears in its original orientation, as specified in the unit list;
- the unit's top-most members are in row 0;
- the unit is centered, so that its left-most and right-most members are equidistant from the left and right sides of the board, respectively. If the distance to the sides is not a whole number, then round toward the left (i.e., there will be less space on the left).



Example units.
The pivot is marked by a dot.

If the spawn location is not a valid location for this unit, the game ends.

Unit Commands

Each unit supports the following commands (as abstract syntax):

`move [ E | W | SE | SW ]`
: Move all members of the unit one cell in the given direction.

`turn [ clockwise | counter-clockwise ]`
: Rotate all members of the unit by 60 degrees in the specified direction, around the unit's pivot point.

Issuing a command that would result in an invalid placement for the unit ***locks*** the unit in place without any movement. The members of a locked unit become part of the board, and the corresponding cells on the board become full. Locking units into place is one way to score points.

Important: the same unit cannot visit a location more than once. *(The Old Ones do not take kindly to stuttering computations!)* Issuing a command that would move all members and the pivot to locations that they have previously occupied is an ***error***, and the program will score ***0 points*** for that board.

Updating the Board

When a unit becomes locked, the board is updated as follows:

1. the members of the unit are added to the board, making the corresponding cells full;
2. if all cells in a row became full, then the row is ***cleared***; and
3. if any units remain in the source, the next unit is spawned; otherwise, the game ends.

When a row is cleared all of its cells become empty. Also, the cells from all rows above (i.e., with smaller numbers) are moved one row down. Note that this means that the cells in a row will move either to the SE or SW, depending on the row number.

Phrases of Power

There are ***eighteen*** phrases of power that can be invoked within a sequence of commands. Phrases of power are case insensitive (the fundamental forces of the universe are unperturbed by minor issues like capitalization). Points are earned for invoking and repeating these phrases, as described below.

The shortest phrase of power is three characters long, e.g. "*Ei!*". The longest phrase of power is of length fifty-one. The phrases of power can be found in contest artifacts, tweets, and background literature and media.

The "Power" columns on the leaderboards indicate how many distinct phrases of power were invoked in a team's solution to a particular problem, or over all a team's solutions.

Scoring

The player scores points for each locked unit. There are three factors that contribute to the number of points that are won:

- the number of cells in the unit, ***size***,
- the number of lines cleared with the current unit, ***ls***,
- the number of lines cleared with the previous unit, ***ls_old***,

The points are computed according to this formula:

```
move_score = points + line_bonus
  where
  points = size + 100 * (1 + ls) * ls / 2
  line_bonus  = if ls_old > 1
                then floor ((ls_old - 1) * points / 10)
                else 0
```

The sum of move scores for all locked units is denoted ***move_scores***.

The player also scores points for invoking phrases of power. For each phrase of power ***p***, there are two factors that contribute to the number of points scored:

- the length of the phrase of power, len_p; and
- the number of times the phrase of power appears in the sequence of commands, $reps_p$.

The score for a phrase of power ***p*** is computed according to this formula:

$$power\_score_p = 2 * len_p * reps_p + power\_bonus_p$$

where

$$power\_bonus_p = \begin{cases} 300 & \text{if } reps_p > 0 \\ 0 & \text{otherwise} \end{cases}$$

The sum of power scores for all phrases of power is denoted ***power_scores***.

Consequently, the total number of points earned for a board is:

$$points = move\_scores + power\_scores$$

As previously mentioned, issuing a command that would result in an error immediately ends the game and the player receives ***0 points*** for that board.

Formats

Input

Programs will be given JSON objects describing the game configurations and information on how many games to play using the following JSON schema (names that are in bold correspond to objects described in subsequent sections):

```
{ "id": number              /* A unique number identifying the problem */
```

```
, "units": [ Unit ]
  /* The various unit configurations that may appear in this game.
     There might be multiple entries for the same unit.
     When a unit is spawned, it will start off in the orientation
     specified in this field. */

, "width":  number          /* The number of cells in a row */

, "height": number          /* The number of rows on the board */

, "filled": [ Cell ]        /* Which cells start filled */

, "sourceLength": number    /* How many units in the source */

, "sourceSeeds": [ number ] /* How to generate the source and
                               how many games to play */
}
```

For each such JSON object, the program should play one game for each ***seed*** in the field `sourceSeeds`. Each of these games will be scored separately, and will be played using the same units, board configuration, and number of units. However, the order of the units will differ depending on the seed. Full details on how to order the units in the source are below.

Cell

Identifies a cell, either on the board or within a unit.

```
{ "x": number, "y": number } /* x: column, y: row */
```

Unit

The configuration of a unit. The cells are relative to a local coordinate system, and will be translated to the board when the unit is spawned. The local coordinate system of each cell, like the board's coordinate system, has smaller row numbers in the "up" direction and smaller column numbers in the "left" direction.

```
{ "members":  [ Cell ]  /* The unit members. */
, "pivot":    Cell      /* The rotation point of the unit. */
}
```

Note that the pivot cell does not have to be a member of the unit.

The Source

The order of the units in the source will be determined using a pseudo-random sequence of numbers, starting with a given ***seed***. The unit identified by a random number is obtained by indexing (starting from 0) into the field `units`, after computing the modulo of the number and the length of field `units`. So, for example, if the configuration contains 5 units, then the number 0 will refer to the first unit, while the number 7 will refer the 3rd one (because `7 mod 5` is 2, which refers to the 3rd element).

The pseudo-random sequence of numbers will be computed from a given seed using a linear congruential generator with the following parameters:

modulus: 2^{32}
multiplier: 1103515245
increment: 12345

The random number associated with a seed consists of bits 30..16 of that seed, where bit 0 is the least significant bit. For example, here are the first 10 outputs for the sequence starting with seed 17: `0,24107,16552,12125,9427,13152,21440,3383,6873,16117`.

Command Line Parameters

Submissions should support the following command line parameters, which may be provided in any order:

Flag	Type	Description
`-f`	`FILENAME`	File containing JSON encoded input
`-t`	`NUMBER`	Time limit, in seconds, to produce output
`-m`	`NUMBER`	Memory limit, in megabytes, to produce output
`-c`	`NUMBER`	Number of processor cores available
`-p`	`STRING`	Phrase of power

The parameter `-f` may be provided multiple times, which means that the program should produce results for all of the given inputs.

If provided, the parameter `-t` indicates the execution time to produce the output for ***all problems***. If it has not finished, the program will be killed after that many seconds. If `-t` is not provided, no time limit is imposed on the program.

If provided, the parameter `-m` indicates the maximum amount of memory that the program is allowed to use at any time. Programs that try to allocate more than this amount of memory will be killed. If `-m` is not provided, no memory limit (other than the hard memory limit of the system the judges choose to run it on) is imposed on the program.

If provided, the parameter `-c` indicates the number of processor cores available to the program.

If provided, the parameter `-p` is a phrase of power (for example, ***Ei!*** for the phrase given above). The parameter `-p` may be provided multiple times, once for each phrase of power the program should attempt to invoke in its output. As you discover more phrases of power, you can re-run the program with additional `-p` command line options; during judging (except for the lightning division), the program will be passed all the phrases of power.

Output

The output of the program should be a JSON list, containing one entry per problem and seed, using the following schema:

```
[ { "problemId": number   /* The `id` of the game configuration */
  , "seed":      number   /* The seed for the particular game */
  , "tag":       string   /* A tag for this solution. */
  , "solution":  Commands
  }
]
```

The `tag` field is meant to allow teams to associate scores on the leaderboards with specific submitted solutions. If no `tag` field is supplied, a tag will be generated from the submission time.

Commands

For each game played, the program should compute a sequence of commands. The sequence of commands should be represented as a JSON string, where each character corresponds to a command, using the following encoding:

{p, ', !, ., 0, 3}	`move W`
{b, c, e, f, y, 2}	`move E`
{a, g, h, i, j, 4}	`move SW`
{l, m, n, o, ***space***, 5}	`move SE`
{d, q, r, v, z, 1}	`rotate clockwise`
{k, s, t, u, w, x}	`rotate counter-clockwise`
\t, \n, \r	(ignored)

Exactly one character from each set is used to denote the corresponding command; the sets are comma-separated, and no command is denoted by a comma character. For example, the character sequences "2xjw 4s" and "cthulhu" both correspond to the command sequences `move E`, `rotate counter-clockwise`, `move SW`, `rotate counter-clockwise`, `move SE`, `move SW`, `rotate counter-clockwise`.

(***Note:*** to clarify possibly-ambiguous font rendering, the letter "l" is a command for `move SE`, while the number "1" is a command for `rotate clockwise`)

The multiple possible choices for each command enable the embedding of ***phrases of power*** (described above) into command sequences to earn additional points (and help keep the flesh-eating darkness at bay).

The three white-space characters listed in the table may appear in the command sequence and will be ignored. Any symbol in the command sequence that is not listed in the table will be treated as an ***error***, and the program will be awarded ***0 points*** for this particular execution.

Tournament

Program evaluation will be done in two phases: the ***qualifiers*** and the ***finals***. The qualifiers, which are additionally split into the ***lightning division*** and the ***qualifying division***, will happen live during the programming contest. The finals will be conducted by the judges after the contest ends.

During the qualifiers, teams will submit solutions to a collection of known problems. Based on their submissions, teams will be ranked, and the top-ranking teams will enter the finals. The number of teams to enter the finals will be determined by the judges.

The finals will be conducted after the contest, and will determine the winners. The judges will build the finalists' programs and present them with a collection of problems that may be different from those used during the qualifiers. The resulting solutions will be ranked, and this ranking will determine the overall winners of the contest.

The Qualifiers

The qualifiers will determine the teams that will enter the finals.

Individual Problem Ranking

During the contest, teams will have an opportunity to submit solutions to the qualifier problems. Each

qualifier problem has its own leaderboard, which teams are ranked based on the solutions they have submitted for that problem. The score for a team is the average score of the ***latest*** submitted solutions for each seed of the problem, rounded down to the closest integer. So, for individual problems, ***bigger scores*** are better. Seeds without solutions, or that result in errors, score 0 points. In the event of a tie score, the team whose solution invokes more distinct phrases of power receives the higher ranking. Teams that tie with respect to both score and number of distinct phrases of power invoked receive the same ranking.

Overall Qualifier Ranking

The overall leaderboard will determine the finalists. The overall score for a team is the sum of their ***rankings*** for the individual problems. So, on the overall leaderboard, ***smaller scores*** are better.

Lightning Division

The lightning division includes all solutions ***for which source code is submitted*** during the first 24 hours of the contest. Scoring in the lightning division includes only move-related points, and phrases of power are not considered. That is, scoring for the lightning division is exactly as described above except that the final formula is `points = move_scores` instead of `points = move_scores + power_scores`. Rankings for the lightning division are determined without the phrases of power tiebreaker; in the lightning division, all teams with the same score on the same problem receive the same ranking for that problem. In the unlikely event of an overall tie in the lightning division, the earliest of the tied submissions wins.

There is no separate leaderboard for the lightning division. Lightning division submissions are also qualifying division submissions, and they appear on the leaderboard accordingly.

The Finals

The teams that have the highest overall rank during the qualifiers will enter the finals. The programs of these teams will be evaluated by the judges on another set of problems.

Submitting Solutions

Before submitting solutions, at least one competitor on a team must obtain an API token from the login server (davar.icfpcontest.org). The team must also be created on the login server. It is not possible to join an existing team using the login server; instead, ***one*** team member should create the team and add the email addresses of all the other team members.

Solutions are submitted using an HTTP `POST` to the submission server. One way to accomplish this is to invoke `curl`, as follows:

```
curl --user :$API_TOKEN -X POST -H "Content-Type: application/json" \
    -d $OUTPUT \
    https://davar.icfpcontest.org/teams/$TEAM_ID/solutions
```

The variables referenced in the `curl` invocation are the following:

`API_TOKEN`	The API token obtained from the login server. It must be prefixed with a colon, as shown in the command above.
`OUTPUT`	The output of the program (a JSON list, as described above).
`TEAM_ID`	The numeric team ID of the team submitting the solution.

The URL in the `curl` invocation is the same as the submission URL provided in the web interface.

Submitting Source Code

All teams should submit their source code. The judges may build and evaluate the source code further—in particular, the judges will use it to evaluate the finalists, to determine the Judges' Prize, and to determine the winner of the lightning division.

The source code should be submitted as a single `.tar.gz` file that should containing at least the following:

- A `Makefile` to be used by the GNU `make` utility.
- A text file called `README`.

The `Makefile` will be used the build the program. The program should build by just executing `make`. The result should be an executable file called `play_icfp2015`, which will be used to compute solutions. This executable should support the required command line parameters.

The `README` should describe what you'd like the judges to know about your program—anything you can write here that will help the judges build the program and appreciate its cleverness would be appreciated! In particular, you may want to list the dependencies of your program, in case it fails to build in the judges' build environment. The judges will try to install additional dependencies, within reason.

When the judges choose to build a program, the build will be done on a standard Linux system running a recent Ubuntu flavor, with a standard set of development tools, plus any additional compilers or interpreters that might be needed. If the build requires a platform other than Linux, we will obtain and use it. We have at our disposal and expertise a wide array of systems, including those of Microsoft and Apple.

System Integrity

OGA takes the integrity of our systems very seriously. Any attempts to compromise the integrity of the submission server or any other contest infrastructure will result in disqualification (and more... ***serious*** penalties).