# Hitori

## Assignment 1: User Interface

The objective of this assignment is to start the project: set-up a git repository, create the file hierarchy for your project, write a build system (Makefile) and develop a user interface through basic command line arguments.

## 1. Setting the GIT repository

- Start by reading the git tutorials cited in the lecture (if lecture is not sufficient)
- Set up your global git configuration (name, email, editor)
- Go to the ISTIC gitlab server (https://gitlab.istic.univ-rennes1.fr)
- Create a repository named *hitori.* Invite the professor in charge of your group + the professor in charge of the lecture (Isabelle Puaut)
- Get a local version of your repository on your computer (*git clone*, through *https* or *ssh* if you have set your SSH key on the Gitlab server – strongly recommended)

From now, you are able to pull and push from any place.

## 2. Create file hierarchy

1. Create the file hierarchy (with empty files for now) that contains the following files:

```
Makefile
include/
      \--hitori.h
src/
      |-- Makefile
      \--hitori.c
doc/
tests/
```

The `include/hitori.h` file must contain:

```
#ifndef HITORI_H
#define HITORI_H

#endif /* HITORI_H */
```

**Note**: empty directories (here, doc and tests) are not pushed by `git` by default. The common standard practice you will use such that `git` pushes the empty directories is to create a file named `.gitkeep` in these directories.

2. Write a `main()` function with the following prototype: `int main (int argc, char*argv[])` that will simply write `Hello World!` on `stdout`.
3. Write the `file src/Makefile` with the following targets
    a. all: generate the `hitori` binary file from the source files
    b. clean: remove all temporary files + binary file generated by the compilation
    c. help: display the targets of the `Makefile` with a short description
    d. test: call some shell script (or run a list of commands) to test your software

   The `Makefile` should define variables CFLAGS, CPPFLAGS, LDFLAGS and the target .PHONY. compilation will use flags `-std=c11, -Wextra and -Wall.`

4. Create a file named `Makefile` at the root of the project with targets `all, clean` and `help`. This `Makefile` is expected to be used by users of your project that want to run it and do not want to know the files it contains. This `Makefile` should create the binary of the `hitori` game on the root directory.

## 3. Options and arguments parser

Write an options and arguments parser that supports the following options, even if the code for most of them will be just empty for now:

```
sh> ./hitori -h
Usage:      hitori [-a|-o FILE|-v|-h] FILE
            hitori -gSIZE [-u|-o FILE|-v|-h]
Solve or generate hitori grids of size: 3 to 15
-a, --all                search for all possible solutions
-gN, --generate[=N]      generate a grid of size NxN (default:4)
-o FILE, --output FILE   write output to FILE
-u, --unique             generate a grid with a unique solution
-v, --verbose            verbose output
-h, --help               display this help and exit
```

1. Set up the option parser structure using `getopt_long()` to support the `'-h'` or `'--help'` option and exit with EXIT_SUCCESS
2. Manage option `'-v'` or `'--verbose'` to set up a Boolean variable `verbose` to true if the option is set. In normal mode, the software will only produce basic information (input grid, input grid consistency/validity, output grid(s), number of solutions). When the `verbose` flag is set, more details on the will be printed (details on grid parsing and grid solving). All information in `verbose` mode will be printed on `stdout` by default, unless option `-o` is set.
3. Manage the option `'-o'` or `'--output'` to print the output of the software to a file (by default, `stdout` is used). When the software is in grid generation mode, the output file will contain the generated grid only, and option `verbose` will then be forbidden. Suggested functions to use: `fopen(), fclose().`
4. Manage the options `'-a'` and `'-u'` and relate them to variables `bool all`, and `bool unique`. In general, using global variables is not recommended but using a global structure that stores the software configuration is allowed for the project.
5. Manage the option `'-g'` to store the configuration options. The code will check that N is an integer and is among supported grid sizes (from 3 to 15).
   Suggested functions to use: `strtol(), atoi().`
6. The software will have two modes: solver or generator (solver mode by default, generation mode when option `'-g'` is set). Write the code needed to detect in which mode we are and exit with an error when inconsistent options are found in the command line (`'-u'` on solver mode, `'-a'` in generation mode). Check that all parameters are given (grid to solve in solver

mode, with readable file). If one of these prerequisites is not met, then fail by displaying an error message and return EXIT_FAILURE.

Here is an example of output:

```
sh> ./hitori
hitori: error: no input grid given!
sh> ./hitori -u
hitori: error: option 'unique' conflicts with solver mode, exiting!
```

Suggested functions to use: `perror(), err(), errx(), warn(), warnx().`

**Notes**:

- For all assignments (for automatic testing purposes), error messages will be printed on `stderr`. The software will exit with code `EXIT_SUCCESS` in case of success, and `EXIT_FAILURE` otherwise.
- The software will have a debug mode defined at compile time (#ifdef/#endif DEBUG, or DEBUG macro). In normal operation, the `debug` mode will be turned off.

## 4. Submitting your first assignment

Once you wish to submit your assignment, create a for your first assignment and push your code to the central repository.

```
sh> git commit
sh> git tag assignment-1
sh> git push --set-upstream origin assignment-1
```

Each assignment will be assigned a different tag (`assignment-1, … assignment-6`).