A & DS1 Project

TAKUZU

(Proposed by Mr. Nicolas FLASQUE)

Instructions & General Information :

⇒ This project is to be done exclusively in C language

⇒ Team Organization:
- This project is to be carried out in pairs (**Asingle group of three students could be accepted only in case the number of students of a group is odd**)
- The list of teams is to be given to the teachers at the **latest** at the end of the first project follow-up session

⇒ Key dates:
- Release date: **28/03/2022**
- Project presentation date: **04/04/2022**
- Follow-up date 1: Week of **04/04/2022**
- Follow-up date 2: Week of **09/05/2022**
- Submission date: **15/15/2022** at **11:59 pm**
- Date of defense: Week of **16/05/2022**

⇒ Final output: A **.zip** archive containing
- The project code containing the **.c** and **.h** files
- The report in **.pdf**
- A **README.txt** file listing the programs and how to use them in practice. This file should contain all the instructions needed to run the program; it is very important to explain to the user how to use the tool.

⇒ Project filing:
- Dedicated deposit area on Moodle.

⇒ Evaluation
- Indicative rating scheme
- Detailed rating: to be provided later
- Final project grade = Code grade + Report grade + Defense grade
- Reminder: project grade = 15% of the grade for the course "Algorithmicss and Data Structures 1"
- Members of the same team may receive different grades depending on the effort put into the project.

⇒ Plagiarism
- Any work containing plagiarism will be severely sanctioned

Code Organization:

⇒ Code scoring will mainly take into account:
- Implementing the requested features: move forward as best as possible but NOT out of topic
- The quality of the code provided: organization in modules and functions, **comments**, significant variable names, respect of file names.
- Ease of use of the user interface

Educational concepts covered:

⇒ To help you with this project, you can rely on :
- TI202 course materials (I, B, R)
- Books, various courses on the web. **BE CAREFUL !!** it is not about copying entire programs.
- **Efrei** teachers during the project follow-up sessions

Preamble

Takuzu is a grid game in the spirit of Sudoku, where the numbers used to fill a grid are only 0 and 1.

A game of **Takuzu** is played on a square grid of even dimensions. In this project, you will propose two dimensions:

1. One using a matrix of size 4 x 4 where the rows are numbered from 1 to 4 and the columns from A to D ;
2. The other one using a matrix of size 8 x 8 where the rows are numbered from 1 to 8 and the columns from A to H.

The rules of **Takuzu** are extremely simple:

1. In a row, there must be as many 0s as 1s
2. In a column, there must be as many 0s as 1s
3. There cannot be two identical rows in a grid
4. There cannot be two identical columns in a grid
5. In a row or column, there cannot be more than two 0s or two 1s in a row (there cannot be three 0s or three 1s in a row)

Based on these rules, your menu should offer 3 main features:

1. Let a player solve a grid
2. Solve automatically a **Takuzu** grid
3. Generate a **Takuzu** grid

Part I: Let a player solve a grid (10 pts)

In this step, the grid to be solved is completely known by the machine. However, only a part of the cells is displayed, and it is up to the user to propose the values to fill the grid. The program simply helps the user in two ways:

1. By indicating the validitý of the moves played, i.e. making sure that the rules are respected;
2. By giving, for a grid, clues on correct moves.

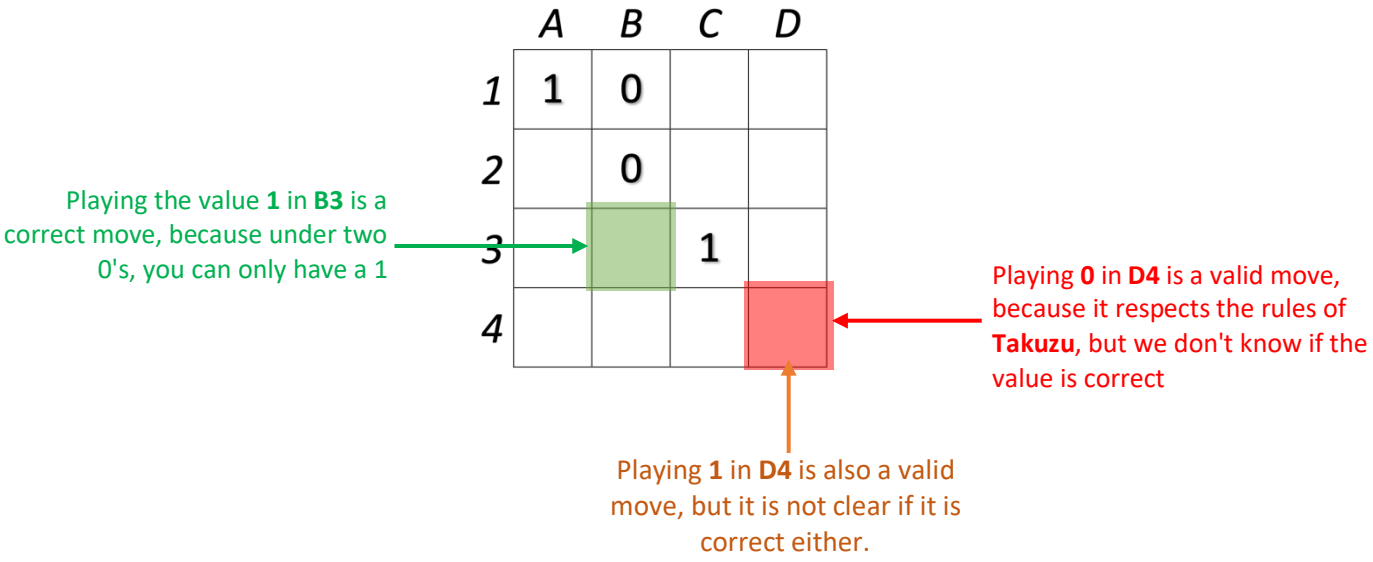
Correct move Vs Valid move

A **valid** move is a move which respects the rules of **Takuzu**, but which is not necessarily the right answer.

A **correct** move is a move that gives the right answer (0 or 1) for a given square of the grid.

Example:

Let's consider the following 4 x 4 grid:



Validity of the played moves

Each time a player proposes a move, the program must indicate whether the move is valid or not. There are several ways to do this. In this project, we will proceed as follows:

1. To enter a move, the user must enter the row number and the column number
2. The user must then insert the desired value (0 or 1)
3. The program displays a text to indicate if the move is valid or not.
4. If the move is correct, the program must indicate that it is correct and explain why it is correct
5. The user has 3 tries to correct his move if it is not valid
6. At the end of these 3 moves, the program offers him two choices:
    - Try to play another cell
    - Get a clue
7. If the user has tried to solve 3 cells without success, the program stops indicating a failure in solving the grid.

Your program will have to test, by functions, the validitý of a move according to the values already discovered by the player.

Implementation of the grid

The grid is first stored in its complete form. The values of the complete grid (the solution to which the player must arrive) are stored in a 2D array, having the values 0 and 1. We call this array the **solution**.

To this array, we will associate a grid of the same dimension called **mask**. The latter will also contain the values 0 and 1 but with a different meaning. Indeed:

1. The value 0 in a **mask[i][j]** box means that its associated **solution[i][j]** box must be **invisible** to the player at the beginning of the game.

2. The value 1 in a **mask[i][j]** box means that its associated **solution[i][j]** box must be **visible** to the player at the beginning of the game.

Finally, the grid that the player sees is partially filled at the beginning of the game with 0's and 1's from the complete grid (in the right place of course); the undiscovered squares are for example represented by the value -1 or any value different from 0 and 1; we will call this **grid_game**.

Illustration:

Let's say the following solution (which is valid of course) and a mask

1	0	0	1
1	0	1	0
0	1	1	0
0	1	0	1

solution grid

1	0	0	0
0	0	1	0
1	0	1	1
0	1	0	0

mask grid

1			
		1	
0		1	0
	1		

game_grid Grid

The **mask is** not a **Takuzu** grid! The 0's of the mask, noted in gray, indicate the values not visible by the player at the beginning of the game.

Some clues to offer to the player

The player can ask for up to 3 hints on a game if he doesn't succeed a move after 3 tries. Your program, from the **game_grid**, must find the correct moves dictated by the **Takuzu** rules:

1. Above, below, to the left, to the right of a series of two 0's, there can only be one 1
2. Above, below, to the left, to the right of a series of two 1's, there can only be one 0
3. Between two 0's, there can only be one 1
4. Between two 1's, there can only be a 0
5. By comparing a row (or column) already filled with a row (or column) missing 2 values, if all values match, then we can fill the row (or column) missing two values.

Example:

If in the grid **grid_game,** we have the following two rows:

0	1	1	0	1	0	1	0

0		1	0	1	0	1	

So, to complete the second row you need a 1 and a 0. Since there cannot be 2 identical rows in the grid, the second row must be completed by:

0	0	1	0	1	0	1	1

Menu

When a user chooses to solve a grid by himself, the program should show him 3 submenus:

1. Choose the size of the grid: 4x4 or 8x8
2. Enter a mask manually
3. Generate a mask automatically
4. Play (if this step is chosen, a mask is automatically generated randomly)

Part II: Automatically solve a grid (6 pts)

This is the easiest part of the project!

In this section, you are asked:

1. To write a function that will solve a **grid_game** by applying the clues progressively according to the possibility of computing them.
2. To display the resolution steps on the screen: display the state of the grid after the application of each hint until the final solution by putting pauses in the program, see the `Sleep()` or `sleep()` functions; or by asking the user to press a key between each move played.

Part III: Generate a grid (4 pts)

In this part, you are asked to generate a **valid** **Takuzu** grid. To achieve this, we must go through the following two steps:

1. Generation of valid rows (or columns) according to the selected grid size (4x4 or 8x8)
2. Construction of a complete valid grid from valid rows and columns.

Valid Rows / Columns

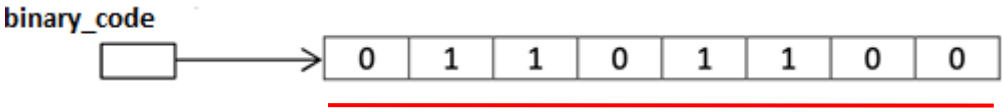
To build a valid row (or column) model, we will rely on binary coding. Indeed, since a row (column) is only made of 0 and 1, we can consider the total combination as a single number.

Thus, if the grid size is:

- 4 x 4: then a row (column) of **Takuzu** represents a binary number between 0 and 15
- 8 x 8: then a row (column) of **Takuzu** represents a binary number between 0 and 255.

Each generated binary number will be stored in a 1D array called **binary_code**. Then, for each generated combination, check if the **Takuzu** rules are respected:

1. Count the number of 0's and 1's in the array **binary_code**
2. Check that there are no more than two 0's or two 1's in the array
3. ...



Binary representation of the number 108: Example of a row of a grid of size 8 x 8

Construction of the complete grid

Based on the valid rows (columns), build the complete grid gradually.

Menu

When a user chooses to generate a **Takuzu** grid automatically, the program must first ask him to choose the size of the grid to generate. After choosing the size, the program should display 2 sub-menus:

1. Display all valid rows (columns)➔ Think about the right data structure to store them as you go along.
2. Generating a **Takuzu** grid➔ The display should include pauses that allow us to follow the detailed steps of the grid generation.

Resources

Here are 4 simple **Takuzu** 8x8 grids, which your program should be able to solve easily. For these grids, we give you the solution. The grey cells correspond to the cells which are not visible at the beginning of the game (**mask**).

For the 4x4 grids, you can create some of them yourself very easily.

1	0	1	1	0	1	0	0
1	0	1	0	1	0	0	1
0	1	0	1	1	0	1	0
0	1	0	1	0	1	1	0
1	0	1	0	0	1	0	1
0	1	0	0	1	0	1	1
0	0	1	1	0	1	1	0
1	1	0	0	1	0	0	1

0	0	1	0	1	0	1	1
1	0	0	1	0	1	0	1
0	1	0	1	1	0	1	0
0	1	1	0	0	1	1	0
1	0	1	0	0	1	0	1
1	0	0	1	1	0	0	1
0	1	1	0	1	0	1	0
1	1	0	1	0	1	0	0

1	0	0	1	0	1	0	1
0	0	1	1	0	0	1	1
1	1	0	0	1	1	0	0
1	1	0	1	0	1	0	0
0	0	1	0	1	0	1	1
0	0	1	1	0	1	0	1
1	1	0	0	1	0	1	0
0	1	1	0	1	0	1	0

1	1	0	1	0	1	0	0
1	0	1	0	1	0	0	1
0	1	0	1	0	0	1	1
0	1	1	0	0	1	1	0
1	0	1	0	1	1	0	0
1	0	0	1	1	0	0	1
0	1	0	1	0	1	1	0
0	0	1	0	1	0	1	1

General remarks:

⇒ Make sure systematically that entries have coherent values even if it is not always explicitly requested
⇒ Do not hesitate to post messages to users when an action is not possible.
⇒ Display the menu at the end of each action to switch to another action.