ICC-P Project: 2048 Game

1. Project Overview

In this project you will implement a simplified version of the famous 2048 game for smartphones and play it in ther terminal. You will implement various parts of the game individually before putting them all together. You will then be able to submit your code to an automatic grader which will run tests on your code awarding you partial points for each test that passes.

2. Game Description

2048 is a single-player sliding puzzle game played on a 4x4 grid. Here is a website where you can play the game: `https://2048game.com/`. Each cell in the grid can either be empty or contain a numbered tile that is a power of two (2, 4, 8, 16, . . .). At the start of the game, two tiles appear randomly on the grid, each showing a number 2.

Objective

The goal of the game is to combine tiles with the same number to create higher-value tiles, ideally reaching the 2048 tile. However, the player can continue playing beyond that if moves are still possible.

How the Game Works

- The player uses the keys **W**, **A**, **S**, **D** to shift all tiles **up**, **left**, **down**, or **right**.
- Each key press triggers a move that updates the entire grid according to the game's movement and merging rules. These rules determine how tiles slide, when merges occur, and how the final state of each row or column is formed. A full, precise description of this procedure is given in the section describing the `move` function.
- After every **valid** move (one that changes the grid by sliding or merging tiles), a new tile with value **2** appears in a randomly chosen empty cell.
- The game continues until no valid moves remain in any direction.

Project Specific Implementation

In this project, several values are provided as **global variables**. A global variable is a variable that is defined once at the top level of the program and can be **read** from inside any function.

- You **may read** global variables inside your functions without doing anything special.
- You **should not modify** these global variables inside your functions. All functions must behave correctly using the values that already exist.

- You **may temporarily change** the values of these variables when testing your code (for example, by using a different `MAX_TILE`), but these changes must not occur inside the functions you submit.

The project uses the following global variables:

- `BLANK = 0` (Represents an empty cell in the grid.)
- `GRID_SIZE = 4` (The game uses a fixed `GRID_SIZE` x `GRID_SIZE` grid.)
- `VALID_KEYS = ['W', 'A', 'S', 'D', 'Q']` (The only allowed user inputs.)
- `MAX_TILE = 64` (The highest tile value the game allows.)

Although the original 2048 game continues until a tile with value 2048 is created, this project uses the `MAX_TILE` variable to define when the game should stop. You may assume that all tile values will always be powers of two between 2 and `MAX_TILE`. `MAX_TILE` is set to 64 in this project by default, but you are free to modify it during your own testing to check whether your code still behaves correctly with larger values. The highest value we may use when testing your submission is 2048.

In the standard game, a new tile added after a move can be either a 2 or a 4. For simplicity, this project only adds a tile with value **2** after each valid move.

Game Win

The game is won when the player obtains at least one tile of value `MAX_TILE`. In that case the game ends and the text "You Win!" is displayed in the terminal.

Game Over

The game is lost when no moves are possible:

- The grid is full (no empty cells), and
- No adjacent tiles have the same value (so no merges can be made).

In that case the game ends and the text "Game Over!" is displayed in the terminal.

3. Game Design

The internal representation of the 2048 grid will use a **2D list (list of lists)** in Python. This structure models the `GRID_SIZE` x `GRID_SIZE` board as a matrix where each sublist represents one **row** of the game.

3.1 Grid Representation

- The grid is a list containing GRID_SIZE sublists (rows).
- Each sublist contains GRID_SIZE integer values (columns).
- Empty cells are represented by the BLANK constant.
- Non-empty cells contain integer powers of two (2, 4, 8, 16, etc.).
- The coordinate $(0, 0)$ refers to the **top-left** cell.
- The coordinate $(0, 3)$ refers to the **top-right** cell.
- The coordinate $(3, 0)$ refers to the **bottom-left** cell.
- The coordinate $(3, 3)$ refers to the **bottom-right** cell.

Example of an internal grid with BLANK = 0, GRID_SIZE = 4 and MAX_TILE = 64:

```
grid = [
    [2, 0, 0, 4],
    [4, 4, 0, 0],
    [0, 32, 8, 0],
    [0, 0, 0, 16]
]

top_left = grid[0][0]    # 2
top_right = grid[0][3]   # 4
bottom_left = grid[3][0] # 0
bottom_right = grid[3][3] # 16
```

3.2 Modules / Functions of the game

For this project you are given a template file game_2048.py where you will need to implement the following functions. The functions are already defined in the template file with their correct parameters and comments describing them. You will need to fill in the code inside each function. Each function has a number of points associated with it, which will be incrementallly awareded for each test that passes for that function.

- check_grid(grid) — (given) — check grid's validity
- display_grid(grid) — (given) — print the grid in a formatted way
- init_grid() — 5 pts — create empty grid
- get_user_input() — 15 pts — read and validate user move (W/A/S/D/Q)
- move(grid, direction) — 25 pts — shift and merge tiles according to the move
- get_empty_positions(grid) — 10 pts — get a list of empty positions in the grid

- `add_new_tile(grid)` — 5 pts — randomly add 2 to an empty cell
- `can_move(grid)` — 10 pts — check if any move is possible
- `game_won(grid)` — 5 pts — check if `MAX_TILE` is present
- `main()` — 25 pts — main game loop

The total number of points for the project is 100.

4. Given functions and functions to implement

4.1 Given functions

`check_grid(grid)` — 0 points

This function checks if the grid is valid for our other functions to work correctly. It checks if the grid is `GRID_SIZE` x `GRID_SIZE` in size, and if all numbers are either `BLANK` or a of a power of 2 between 2 and `MAX_TILE` (included).

This function is already implemented for you. You can take a look at how it works, but you don't need to understand exceptions, as these are a more advanced topic.

Important: you need to call this function at the start of every function that takes a grid as input. This is a standard practice in programming to check input arguments and ensure they are valid. You only need to call the function `check_grid()` at the start of your function. If the input is invalid your program will crash with the correct error message. Take inspiration on how it is done in `display_grid()`.

`display_grid(grid)` — 0 points

This function prints a multiline string that displays the current game board in the terminal in a clear, formatted way. This function already prints a new line at the end, so you will not need to add an extra `print()` after calling it.

Each cell is printed as a box that is **4 characters wide** and **3 lines tall**. BLANK cells are shown as empty spaces. Numbers are centered within their boxes to keep the grid visually aligned.

Example result with `BLANK` = 0, `GRID_SIZE` = 4 and `MAX_TILE` = 64:

```
grid = [
    [2, 4, 0, 8],
    [0, 0, 64, 0],
    [2, 0, 16, 32],
    [0, 8, 0, 0]
]

display_grid(grid) ->
```

```
+----+----+----+----+
|    |    |    |    |
| 2  | 4  |    | 8  |
|    |    |    |    |
+----+----+----+----+
|    |    |    |    |
|    |    | 64 |    |
|    |    |    |    |
+----+----+----+----+
|    |    |    |    |
| 2  |    | 16 | 32 |
|    |    |    |    |
+----+----+----+----+
|    |    |    |    |
|    | 8  |    |    |
|    |    |    |    |
+----+----+----+----+
```

This function is already given to you. Take a look at how it is implemented and how it uses the grid. It will be useful to know how to navigate in the grid for the following functions.

4.2 Functions to implement

`init_grid()` — 5 points

This function creates and returns an empty `GRID_SIZE` x `GRID_SIZE` grid. An empty grid is a grid where all positions are initialized with a `BLANK` value.

- The grid is represented as a **2D list** (a list of lists).
- Each sublist corresponds to one row of the grid.
- Each position in the grid is initialized with `BLANK` to represent an empty cell..

Example result with `BLANK` = 0, `GRID_SIZE` = 4:

```
[
    [0, 0, 0, 0],
    [0, 0, 0, 0],
    [0, 0, 0, 0],
    [0, 0, 0, 0]
]
```

`get_user_input()` — 15 points

This function reads the player's move from the keyboard and validates it. It accepts capital or lowercase letters.

The player must enter one of the following keys to move the tiles:

- **W or w** → Move up
- **A or a** → Move left
- **S or s** → Move down
- **D or d** → Move right

However the player can also decide to stop the game by entering:

- **Q or q** → Quit the game

The function should:

1. Prompt the user for input: `"Enter move (W/A/S/D/Q): "`.
2. Check if the input is one of the valid keys: `['W','A','S','D','Q']`, use the given constant `VALID_KEYS`.
3. If the input is valid, return the chosen direction as a single **capital** character.
4. If it is invalid, display an error message `"Invalid input, try again."`.

The action of quitting the game will be handled by the main function, so if the user inputs `'Q'` the function should simply return `'Q'`.

Attention: Your function should be case insensitive, i.e., it should also accept lowercase version of the valid keys, and it should accept inputs that contain spaces. This means `' a '` is a valid input. **Hint:** Functions `.upper()` and `.strip()` can be helpful.

Example interaction:

```
Enter move (W/A/S/D/Q): x
Invalid input, try again.
Enter move (W/A/S/D/Q):   a

-> get_user_input() returns 'A'
```

`move(grid, direction)` — 25 points

This function must apply the full movement and merging rules of 2048 to the grid in the specified direction. It must return a **new** `GRID_SIZE` x `GRID_SIZE` grid. The **original grid must remain unchanged**.

Inputs:

- `grid`: a `GRID_SIZE` x `GRID_SIZE` list of lists containing integers. Empty cells contain `BLANK`.
- `direction`: one of the capital characters `'W'`, `'A'`, `'S'`, `'D'` representing movement **up**, **left**, **down**, and **right**.

Output: A new `GRID_SIZE` x `GRID_SIZE` grid that reflects the result of applying the move. If the move does not change the grid (no tile moves or merges), the returned new grid must be identical to the input grid.

Precise Rules The function must follow the official movement and merging rules of 2048 exactly. The steps below describe the required behavior of the game mechanics, not the implementation strategy. Your code may be structured however you like, but the resulting grid must always match the rules given here.

1. **The function must not modify the original grid.** It must return a new grid representing the updated state after the move. Use the provided `copy.deepcopy()`.
2. **All tiles move toward the chosen direction as far as possible.** A tile continues sliding until one of the following occurs:
   - it reaches the edge of the grid, or
   - it becomes adjacent to another tile that prevents further movement.

   Tiles never pass through other tiles, regardless of whether those tiles have already moved or will merge later in this move.
3. **After all tiles have slid, adjacent tiles with the same value may merge.** Merging must follow these rules exactly:
   - Only two tiles of equal value that are directly next to each other **in the movement direction** may merge. Merging starts from the side toward which we are moving.
   - A merge produces one tile whose value is the sum of the two originals.
   - Each tile may participate in at most one merge during a single move. This restriction applies to the original tiles involved in the merge as well as the resulting tile: the newly created tile cannot merge again until the next move.
   - After a merge, the resulting tile occupies the position in the movement direction and blocks any tile behind it from merging into that same position during the current move.
4. **After all possible merges are processed, the tiles must be compacted again toward the movement direction.** No gaps may remain. All empty cells must appear on the side opposite the direction of movement.
5. **Every affected row or column must end in a fully resolved state.** Each line must show:

   (a) all tiles shifted as far as possible toward the movement direction,

   (b) all merges correctly applied according to the rules above,
   (c) remaining empty positions filled with BLANK on the opposite side.

6. **The function must return the completed grid.** If the move produces no sliding and no merges, the returned grid must be identical to the input grid.

Examples with BLANK = 0, GRID_SIZE = 4: and MAX_TILE = 64

```
grid = [
    [2, 0, 2, 0],
    [4, 4, 4, 4],
    [0, 0, 0, 0],
    [0, 2, 16, 2]
]

move(grid, 'A') returns ->
[
    [4, 0, 0, 0],
    [8, 8, 0, 0],
    [0, 0, 0, 0],
    [2, 16, 2, 0]
]


grid = [
    [2, 0, 2, 4],
    [2, 0, 2, 4],
    [4, 4, 8, 4],
    [0, 32, 8, 0]
]

move(grid, 'W') returns ->
[
    [4, 4, 4, 8],
    [4, 32, 16, 4],
    [0, 0, 0, 0],
    [0, 0, 0, 0]
]
```

Hint: You are allowed and encouraged to create helper functions if you think this will help you structure your code better.

get_empty_positions(grid) — 10 points

This helper function scans the entire grid and returns a list of all the empty cells. The order in the returned list does not matter.

Each empty cell is represented by a **tuple** (x, y) where:

- x is the **row index** (0 to 3, top to bottom)
- y is the **column index** (0 to 3, left to right)

A cell is considered empty if its value is `BLANK`.

Example with `BLANK` = 0, `GRID_SIZE` = 4 and `MAX_TILE` = 64:

```
grid = [
    [2, 0, 0, 4],
    [0, 2, 2, 16],
    [0, 4, 64, 0],
    [4, 0, 32, 0]
]

print(get_empty_positions(grid)) ->

[(0, 1), (0, 2), (1, 0), (2, 0), (2, 3), (3, 1), (3, 3)]
```

`add_new_tile(grid)` — 5 points

This function should add a new tile in an empty location in the grid. Consider using Python's built-in `random.choice(list)` function to choose a random element from the list. This `add_new_tile()` function should always add a value 2 tile at the chosen empty position. The function is not allowed to modify the non-empty cells.

Note: You do not need to worry about randomness for the automatic grader. Do not set a seed for the random number generator, as this would make your code non-deterministic.

`can_move(grid)` — 10 points

This function checks whether the player can still make a move.

A move is considered possible if **at least one** of the following conditions is true:

1. There is **at least one empty cell** on the grid.
2. There are **two adjacent tiles** (horizontally or vertically) with the **same value**.

If neither condition is met, the game is over.

Examples with `BLANK` = 0, `GRID_SIZE` = 4 and `MAX_TILE` = 64:

```
grid = [
    [2, 4, 2, 8],
    [8, 16, 8, 2],
    [4, 2, 16, 4],
    [2, 8, 4, 2]
]

can_move(grid) # returns False


grid = [
    [2, 4, 2, 8],
    [8, 0, 8, 2],
    [4, 2, 16, 4],
    [2, 8, 4, 2]
]

can_move(grid) # returns True


grid = [
    [2, 4, 2, 8],
    [8, 16, 8, 2],
    [4, 2, 16, 4],
    [4, 8, 4, 2]
]

can_move(grid) # returns True
```

`game_won(grid)` — 5 points

This function checks whether there is a tile with value `MAX_TILE` present in the grid. If such a tile exists, the function returns `True`, indicating that the player has won the game. Otherwise, it returns `False`.

Examples with `BLANK` = 0, `GRID_SIZE` = 4 and `MAX_TILE` = 64:

```
MAX_TILE = 64
grid = [
    [2, 4, 2, 8],
    [8, 0, 8, 2],
    [4, 2, 16, 4],
    [2, 8, 4, 2]
]
```

```
game_won(grid) # returns False

MAX_TILE = 64
grid = [
    [2, 4, 2, 8],
    [8, 16, 8, 2],
    [4, 2, 64, 4],
    [2, 8, 4, 2]
]

game_won(grid) # returns True

MAX_TILE = 128
grid = [
    [2, 4, 2, 8],
    [8, 16, 8, 2],
    [4, 2, 64, 4],
    [2, 8, 4, 2]
]

game_won(grid) # returns False
```

Putting everything together: `main()` — 25 points

The main function is responsible for calling all the other functions in order to be able to play the game. At the start of the function it should create an empty grid and add two 2s at random positions. Then in the main game loop, as long as the player still has a move available, it should display the grid, ask the player for a new move, apply the move, and add a new tile to the grid. If the player enters `'Q'` you should print “Quitting the game.” and end the program.

Important: A new tile should only be added if the move actually changed the grid. If the player tries to move in a direction where no tiles can move (e.g., all tiles are already pushed to the left and they press `'A'`), the grid remains unchanged and no new tile should be added.

If the player reaches a state where at least one `MAX_TILE` tile is present, you should stop the game and print “You Win!” If the player reaches a state where they cannot move anymore, you should stop the game and print “Game Over!” in the terminal. Before showing the final win/lose message, the game should be displayed one last time.

Important: All outputs (e.g., messages, formatting) must match exactly, including capitalization and punctuation, to pass the grader. For this we provied the messages you should use as constants in the template file.

Once this is done you can play the game by starting the program, either by using VS Code’s arrow or by using this command in the folder with the game file present:

```
python game_2048.py
```

5. Instructions and submission details

For this project, you have to implement the functions in the given `game_2048.py` file.
Important: Do not modify the given function names, as the automatic grader would otherwise not recognize the functions and give you 0 points. However, you are allowed to use helper functions if this is useful in your implementation. Use the constants defined in the template file where needed, as your code will be tested with different values for these constants ! Do not hard code any value. For example your code should still be fully functional if `GRID_SIZE` is changed to 5 or if `MAX_TILE` is changed to 2048.

To test your implementation we provide a `tests.py` file that tests the individual functions with the examples given in the handout. Before you start working on the project only the tests of the two already implemented functions (`check_grid()` and `display_grid()`) pass, but when you will have implemented all the functions all tests should pass. Don't hesitate to write additional tests following the file's examples. You will also need to implement the main function and test it yourself by playing the game. All the functions will be tested by the grader and partial points will be awarded for each one.

To use the `tests.py` file, run the file by using the VS Code's arrow or by using this command in the folder with the test and game files present:

```
python tests.py
```

No library other than Python's built-in `random` and `copy` (which are already imported inside the template) is allowed to be used for this project. All functions can and should be coded with the functions you have seen in class or mentioned in this document.

Submission Details

For this project, you can work in groups of 1 or 2 people, which are created in Moodle. To submit your project, only 1 person from the group needs to submit it on Moodle (https://moodle.epfl.ch/). Grading will be done automatically and you will receive partial points for each function, in proportion to tests passed. You are only allowed a limited number of submissions, so please use them wisely.

After your submission Moodle will generate a file `report.txt` with the list of passed and failed tests. The tests will follow a similar pattern as the given `tests.py` file, so don't forget to run this file locally before submitting.

For more details about the submission (grader access, deadline, number of tries, etc.) please check the Moodle page of the class.

Formatting

Readable, consistent formatting is a required part of this project. Correct logic alone is not enough: your code must also follow the formatting and naming conventions described below. If your file is not formatted correctly, you will lose formatting credit even if the program works.

Why Formatting Matters

Consistent formatting ensures that your code is easy to read, easy to debug, and fair to grade. The goal is not to make your code “pretty,” but to ensure it follows standard Python style so that any reader can understand it without effort.

autopep8 will automatically fix indentation, remove inconsistent spacing, and wrap long lines according to the project’s configuration. It will **not** change your logic, rename your variables, or rewrite your functions, so running it is always safe.

Required Formatting Rules

Your submission must meet *both* of the requirements below:

1. **Your file must already match the project’s autopep8 configuration.** Before submitting, format your code using the provided `setup.cfg`. If formatting changes are still suggested after applying autopep8, your file was not correctly formatted.
2. **Your code must use consistent naming and include meaningful comments.**
    - Use **snake_case** for all variables and functions. Examples: `user_input`, `total_count`, `move_line`.
    - Do not use single-letter variable names except for simple loop indices (`i`, `j`, `k`). All other names must be descriptive and at least 2–3 characters long.
    - Add comments inside functions for code that is not immediately obvious, especially multi-step logic.

If your naming is inconsistent, if you rely on one-letter variables, or if you leave out essential comments, you will lose formatting credit.

Using autopep8 in VS Code

The project includes a `setup.cfg` file that defines the formatting rules.

To use it in VS Code, install the autopep8 extension, then inside the file you want to format press:

- Shift + Alt + F (Windows/Linux)
- Shift + Option + F (Mac)

Reminder: to install an extension in VS Code, open the Extensions view by clicking the square icon on the left sidebar or pressing Ctrl + Shift + X (Cmd + Shift + X on Mac). Search for "autopep8" and click "Install" on the appropriate result.

If autopep8 does not change anything after using the keyboard shortcut, your file is correctly formatted.