


MP7

MP 7 - 2048

Introduction

From Wikipedia: 2048 is a single-player puzzle game created in March 2014 by 19-year-old Italian web developer Gabriele Cirulli. The objective is to slide numbered tiles on a grid to combine them and create a tile with the number 2048.

The game can be played online at <https://gabrielecirulli.github.io/2048/>  (<https://gabrielecirulli.github.io/2048/>). or found in the App Store for iPhone or Android. Feel free to try it out...and try not to get too addicted (smile)

More importantly, we provide you with a compiled working copy of the completed MP. You can run `./mp7_gold`, which should be created after running `make`, to play the game and see how your program should function.

Background

The standard game of 2048 is played on a 4×4 grid. The game begins with a random value tile of either 2 or 4 placed at a random location on the board. The player controls the game using the directional keys (we will use keys `w`, `a`, `s`, and `d` for directions where `w` as `up`, `s` as `down`, `a` as `left`, and `d` as `right`). Whenever a direction is pressed, all tiles will first slide in that direction for as far as possible. As they slide, if two numbers of the **same value** collide (for example two 2's), they merge into a single tile whose value is the sum of the two values (two 2's become 4; two 4's become 8...etc). The resulting tile **cannot merge with another tile again in the same move**. So sliding row of (2, 2, 4, 8) to the left would become (4, 4, 8,) **where** is used to denote the empty tile (containing a -1). Additionally, when sliding a row with more than two like terms for example (2, 2, 2,), **merging occurs** first based on the direction of sliding. **For example sliding row (2, 2, 2,) to the left would result in (4, 2, ,)** while sliding to the right would result in (, , 2, 4). Finally, whenever a direction is pressed that results in at least 1 tile changing position, a new random value tile of either 2 or 4 placed at a random empty location on the board. The score is shown at the top of the game. Whenever 2 tiles merge, their sum is added to the total score.

For this MP, you will be implementing the game on a variable sized grid. Upon launching the game, the program will ask the user for dimensions. This input is expected as 2 integers (rows and columns) separated by a space (e.g. '4 4' for the standard 4x4 grid). After configuring board size, the game begins. In addition to the directional controls (w,a,s,d), your version will use `n` to

reset the game, and **q** to quit the game. **n** will recreate the game board with new dimensions, setting all cells to negative one, and randomly adding one tile. **q** will output a "Quitting.." message and terminate the program.

In the documentation for this MP a 'cell' refers to a cell in the game grid, and a 'tile' refers to a cell with a number in it.

Files Provided

- `game.c`, `game.h` - Defines the game data structures, and functions for creating, updating, and destroying game elements. **Modify this code!**
- `main.c`, `main.h` - Calls functions in `game.c` to set up the game and loops through the iterations. You do not need to modify this code. Your implementation **should NOT modify this code**
- `getch_fun.c`, `getch_fun.h` - Implements `getch()` and `getche()` functions, which asks for a single keypress. You will not need to call this (`main.c` calls them for you), and this code **should NOT be modified**.

Details

Within the file `game.c`, you will need to implement the following functions. Descriptions are provided at the beginning of each function.

```
game * make_game(int rows, int cols);
void remake_game(game ** _cur_game_ptr, int new_rows, int new_cols);
cell * get_cell(game * cur_game, int row, int col);
int move_w(game * cur_game);
int move_s(game * cur_game);
int move_a(game * cur_game);
int move_d(game * cur_game);
int legal_move_check(game * cur_game);
```

`game * make_game(int rows, int cols)`: This function returns a pointer to a game structure. Please look at the game struct definition in the game.h header file. We have already allocated the memory associated with the game struct and the cells. You will need to complete the game construction by assigning the correct initial values to rows, cols, and score. You will also need to assign the correct initial values (-1) to each cell within the array.

`void remake_game(game ** _cur_game_ptr, int new_rows, int new_cols)`: As far as what you need to do for the MP, this function is in practice nearly identical to the make_game function. The only significant difference is that the game struct already exists and will be reused, while old game data needs to be reinitialized. We have already freed the memory associated with the previous cells, and have allocated the memory associated the new cells. You will need to complete the game construction by assigning the correct initial values to rows, cols, and score. You will also need to assign the correct initial values (-1) to each cell within the array. Note that while this is not a perfect copy and paste of your code from the make_game function (the variable names have changed) the

algorithm itself almost identical.

`cell * get_cell(game * cur_game, int row, int col)`: Given a game, a row, and a column, return a pointer to the corresponding cell on the game. Your implementation should ensure that position given is within range of the cell dimensions. If the position is out of bounds, you should return NULL.

`int move_w(game * cur_game)`: Slides all of the tiles in `cur_game` upwards. If a tile matches with the one above it, the tiles are merged by adding their values together. Additionally, a tile can not merge twice in one turn. If sliding the tiles up does not cause any cell to change value, `w` is an invalid move and return 0. Otherwise, return 1. The algorithm for accomplishing this is VERY similar to the algorithm devised in the lab, with an addition to accommodate merging tiles.

Overview:

Using the algorithm for sliding from the lab, we will need to decide where we want to merge cells. Since we are sliding upwards, we will want to go through each of the N columns in the $M \times N$ matrix and slide each column up. While traversing down each column, we want to make sure that the row above the target row* has not been combined yet to prevent merging the same tile twice in one turn. In the case of sliding up, we would only merge if the value of the cell in a row above the target row is equal the value of target row cell and that the row above the target row has not been combined.

*We refer to the target row to be the smallest row value in the current column that is empty and less than the current row number (ie. the highest location a tile can be shifted to).

`int move_s(game * cur_game)`: Same as `move_w`, but slide down. Algorithm will need to be modified to accommodate the difference in direction.

`int move_a(game * cur_game)`: Same as `move_w`, but slide left. Algorithm will need to be modified to accommodate the difference in direction.

`int move_d(game * cur_game)`: Same as `move_w`, but slide right. Algorithm will need to be modified to accommodate the difference in direction.

`int legal_move_check(game * cur_game)`: Given the current game, check if there are any legal moves on the board. Return 1 if there are possible legal moves, 0 if there are none. No legal moves remain whenever the board is filled and sliding in any direction will not result a change to the board state. This function can be implemented by returning 1 if any empty spaces remain on the board OR if any 2 adjacent tiles have the same value. Otherwise return 0.

Testing

Compile code by using make command. Running `./mp7_test` will run your source code against the

code and compare the outputs. The tests in `mp7_test` are not comprehensive, but passing all of the tests means that you are on the right track and will get the majority of the points for this MP. **You can write your own tests for this mp7 by modifying** `/test_src/test.c`. Or you can play your implementation of the game for an hour which is admittedly more fun.

Please note: you must implement the `get_cell()` function before running the test code. Otherwise a segmentation fault will occur.

Grading Rubric

Functionality (90%)

- (30%) - Correctly implements the `create_game` and `remake_game` functions
- (40%) - Correctly implements the `move_functions` (10% each).
- (15%) - Correctly implements the `legal_move_check` function.
- (5%) - Correctly implements the `get_cell` function

Style, Comments, Clarity and Writeup (10%)

- (5%) - Introductory paragraph explaining what you did. Even if it is required work.
- (5%) - Code is clear and well commented.

Some point categories in the rubric may depend on other categories. If your code does not compile, you may receive a score close to 0 points.