2048 Game - IP09

Discrete Mathematics Project

CS-1104 | Spring 2020

Project Link - https://github.com/Reminishu/2048_IP-9
Hosted at - https://2048-dm-ip9.netlify.app/

Group Members -

1. Ayushman Ghosh,
2. Gurveen Khanuja,
3. Kanishk Singh, &
4. Nishant P Borah.

Table of Contents:

1. 2048: The Game:
2. Requirements:
3. Main files:
4. Front-end:
5. Back-end:
6. Initialize:
7. Listen for input:
8. Move and Merge:
    a. Algorithm:
    b. Correctness:
    c. Runtime:
9. Random spawn:
10. Moves possible:
11. Display logic:
12. Control flow:
13. Deployment:
14. Source Control:
15. Individual Contributions:
    a. Bibliography:

2048 - The Game:

The 2048 game consists of a 4X4 grid, in which each cell's value is a power of 2. On each turn, the user must perform any of the four actions: up, down, left, or right. All the tiles move in the same direction when an action is performed. Upon this movement, two adjacent tiles with the same value merge to form a single tile whose value is the sum of those numbers. After every move, the board generates a tile with the values 2 or 4 at a random location. The values are also randomly generated with a probability of 90% for 2 and 10% for 4. A move is valid when at least one tile can be moved. The player loses when there are no valid moves possible.

The minimum number of moves required to reach 2048 is 512 and the highest number possible is 131072.

Requirements:

1. Non-greedy movements: The tiles created in one turn can not be combined to form another tile in the same turn. For instance, if there are 4 adjacent tiles with the value 2, they can not combine to give a single tile of value 8 in one turn. They can only combine to give 2 tiles of values 4 each in one turn.
2. Move direction priority: If more than one combination is possible for the same tiles in one move, then the direction in which the move is happening takes priority. For instance, there are 3 tiles with the values 2 each and the player moves the board to the right, then the two tiles on the right will combine to make one tile of value 4.
3. Adding new tile on a blank space: most (90%) of the new tiles will be of value 2, while tiles of value 4 will only be added occasionally (10%).
4. Check for valid moves: The player will not be allowed to make a move that does not change the board in any way.
5. Win condition: the player wins upon reaching the value 2048.
6. Lose condition: the player loses when there are no more valid moves left in the game.

   The game logic meets all the above requirements.

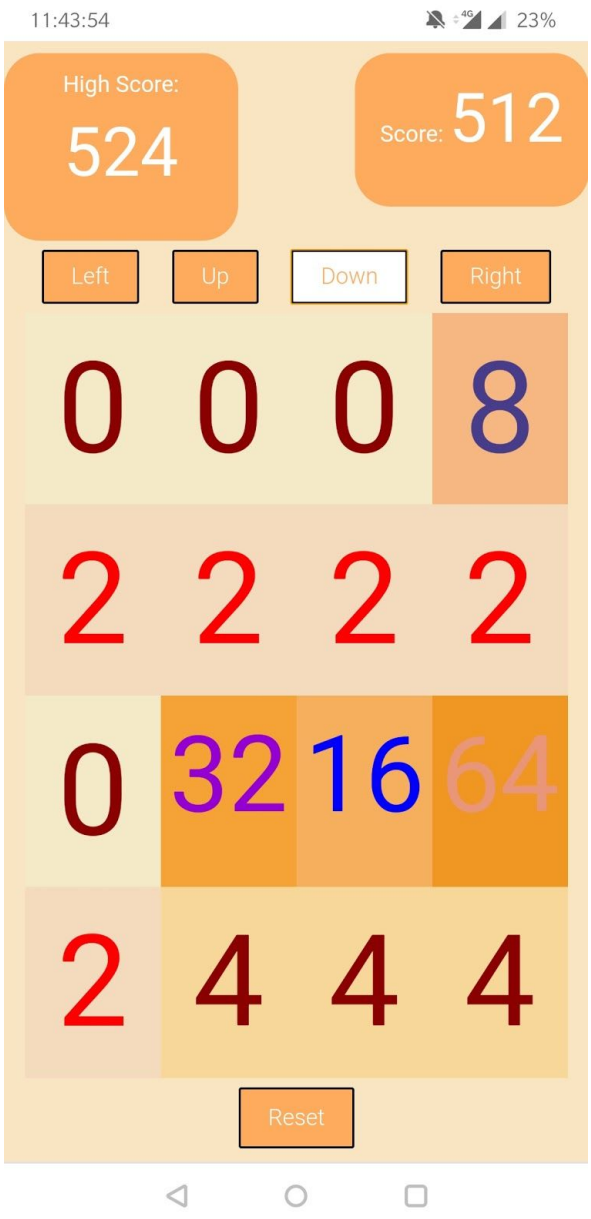
Main files:

Index.html	{Contains main front-end logic}
Index.css	{Contains styling rules}
Index.js	{Contains game and display logic}

Front-end:

The frontend logic is mainly written in HTML and CSS with some JS.

We used Bootstrap to make sure our content was *responsive* and could be viewed on different devices. Bootstrap provides ready-made CSS classes and containers which resize our text as resolutions change. All our content and many other elements such as the buttons are enclosed in bootstrap containers allowing automatic resizing. Our grid, however, was made resizable using custom CSS.



Our HTML code has three main parts:

1. Accessibility buttons(in case the keyboard is unavailable) and Scores (High Score and current score)
2. The Grid itself and accompanying animations.
3. A reset button to restart the game after the game ends.

Our CSS file has custom coloring and animations for when the tiles merge and move around as well as general layout display options. It also displays the information calculated by the backend (Scores, tile merge operations, etc.) and helps users play the game.

Back-end:

The backend logic is completely written in JavaScript and follows the functional programming paradigm. It can be found in the file *index.js*. The breakdown for the core game logic is as follows:

- **Initialize**
- **Listen for input**
- **Move and Merge**
- **Random spawn**
- **Moves possible**

Initialize:

The global variables used are declared in the starting. We store the whole playing grid in a single dimension array *'cell'* initialized with an element 0. Other global variables are '*score'* for keeping track of the score per session, *'highScore'* which keeps track of the highest score achieved on that browser using *'localStorage'* data structure of the browser. Rest of the initialization is done in the function *'reset()'* which is called on the start and every reset. It fills the game array with '0's and then we use JavaScript's inbuilt random number generating function to find two unique locations on the board (in the array) to insert '2's which act as the starting number for the game. At this point the array is ready for the game. Then we reset the value of *'score'* and with that everything is set up for the game to be played. Here we call the function *'keyboard()'* which starts our next step, listening for user input.

Listen for input:

We have two ways to play the game, either the arrow keys of the keyboard or the dedicated buttons displayed on the screen. The keyboard input is taken inside the function *'keyboard()'* which first checks if any valid moves are left on the game grid or not using *'movesLeft()'*. If it returns true, that is valid moves are possible, we listen for arrow key presses and depending on the key pressed we call the function *'move()'* and pass the corresponding key pressed value. The function keeps listening for inputs on a loop till we reach the end of the game. If no valid moves are present, we call the function *'loss()'* which marks the end of a game session.

Move and Merge:

The function *'move()'* forms the major chunk of the game logic.

Algorithm:

As we have a single dimension array as a global object, we operate directly on it by simulating a 4x4 grid through methodical traversal of the game array. In the whole process we keep track of a few things in local variable, namely:

- **headValue** for keeping track of the last non-zero unique value we encountered.
- **headIndex** for keeping track of the position of the said headValue in the game array.
- **zeroIndex** for keeping track of the location of '0' or the first '0' encountered in a consecutive sequence.
- **zeroFlag** for tracking if a '0' was encountered.

Using these variables and loops we simulate the single dimension array as a 4x4 grid and traverse it one row/column at a time depending on the direction from user input. We illustrate the process for a 'right' shift below:

0	**1**	**2**	**3**
4	**5**	**6**	**7**
8	**9**	**10**	**11**
12	**13**	**14**	**15**

The indices of the 1-D array in the simulated matrix

The switch case directs control to *'case 2'* where we start our first loop from index 3 which would be the right edge of the first row and at every iteration we shift the value by 4 thus pointing to the right edge of each row or traversing the rightmost column. After reaching each element in the rightmost column, we reset the values of the above mentioned variables and start another loop for traversing the row. In this loop we start from the index of the previous loop and decrement by 1 at each iteration thus traversing each row from right to left. Inside this loop, we have three conditional statements.

1. First we check if we have encountered a unique non-zero value, if yes we update the *head* variables and then check if any trailing ‘0’s were encountered, if yes we swap the location of the ‘0’ and this unique value therefore shifting it towards right (the direction the user entered). We update the value of *zero* variables accordingly to reflect the swap.
2. If not first, we check if the value is a first zero in sequence, if yes we update the *zero* variables to reflect the encounter and its position.
3. If neither first, nor second, we check if the value is equal to the previous unique value encountered in a consecutive sequence, if yes we call the *‘merge()’* function to handle the merge and reset the *head* variables. We also check if there were any ‘0’s between the two numbers and if not, we update the value of *zero* variables.

After this, we are done with the shifting of the numbers and call the function *‘insertTile()’* to spawn a random tile on the grid and function *‘draw()’* which reflects the updates in the game array in the front-end grid.

Correctness:

As illustrated above, simulate a 4x4 matrix and traverse each row/column in the opposite direction to where we have to shift everything while performing checks and shifting everything one by one.

Runtime:

In each case, we traverse through the whole array only once thus giving a time of O(n).

For *‘merge()’* we take the indices of the two numbers and add them, then mark update the blank location with value ‘0’ and add the sum to the *‘score’*. If the score is higher than the current high score, we update the value in our *localStorage* as well.

Random spawn:

Here in function *‘insertTile()’* we take a new array as a buffer for storing all the locations of ‘0’ in our game array. Then we use the inbuilt random function we first get an index for the buffer array, we use the value stored at that index as the target index for the new tile. We again take a random value and if it's less than 0.9, we insert 2 else we insert 4, simulating a 10-90 split for 4-2 insertion.

Moves possible:

In the function *'movesLeft()'* we traverse the simulated grid first column by column and then row by row, check for '0's or consecutive matching numbers. If found, there are valid moves left.

Display logic:

In function *'draw()'* we update the frontend html element to reflect the value of that tile and change the color depending on the displayed value.

At start, the function*'startGame()'* is called which calls *'reset()'* and *'draw()'* to ready the game array and display it on the frontend.

Control flow:

Starting with a button click,

- *startGame()* is called which calls *reset()* and *draw()* for redying the game array and displaying it.
- Then *keyboard()* is called which keeps listening for key press and checking if valid moves are left. And if moves left it calls
- *move()* which shift the numbers in the desired direction calling
- *merge()* to add matching values and update the array then return back to *move().*
- Then *insertTile()* is called by *move()* which spawns '2' or '4' at a random location and returns back to *move()*.
- Then *draw()* is called by *move()* to display the updated game array on the frontend then returns back to *move()* which then returns back to *keyboard().*
- Then *movesLeft()* is called by keyboard to check if valid moves are present and returns a boolean value to *keyboard()*.
- *loss()* is called by *keyboard()* if no valid moves are left and it updates the frontend to reflect that the game has ended.
- The button press functions are called any time the onscreen buttons are pressed and they call *move()* with the corresponding value.

Deployment:

The project was deployed using Netlify, an all-in-one platform for deploying web projects. It is deployed directly from GitHub and any changes to the repository are automatically reflected on the website.

Deployed Website Link -

https://2048-dm-ip9.netlify.app/

Source Control:

'Git' and 'Github' were used for tracking and managing changes to code. Github proved to be very helpful for collaborating and resolving conflicts.

Github Repository Link -

https://github.com/Reminishu/2048_IP-9

Individual Contributions:

1. Ayushman Ghosh:
    a. Added user input logic in backend.
        i. Tried adding a swipe user input for mobile devices. This feature was not implemented in the final version due to bugs.
    b. Worked on CSS/ HTML in frontend with bootstrap and edited a few CSS classes for user-friendliness.
2. Gurveen Khanuja:
    a. Worked on frontend overseeing site design
    b. Worked on HTML element design and functions as well as key frontend elements such as the starting overlay and accompanying backend code.
3. Kanishk Singh
    a. Implemented complete backend
        i. Core game logic and control (JavaScript)
        ii. Implementing dynamic frontend display control (JavaScript and HTML)
    b. Minor frontend additions
4. Nishant P. Borah:
    a. FrontEnd Logic,
    b. UI Design and CSS Styling,
    c. Animations,
    d. Dynamic Buttons,
    e. Web Deployment,
    f. Compatibility across different devices and screen sizes.

Everyone contributed to doing minor bug fixes and writing documentation.

Bibliography:

https://alligator.io/js/listening-to-keyboard/ [For I/O]
https://2048.love2dev.com/ [Inspiration for the game]
https://www.w3schools.com/ [Very helpful for HTML/CSS]
https://www.netlify.com/ [For Deployment]
https://developer.mozilla.org/ [For JavaScript]
https://xkcd.com/1344/ [For joy in times of sadness]