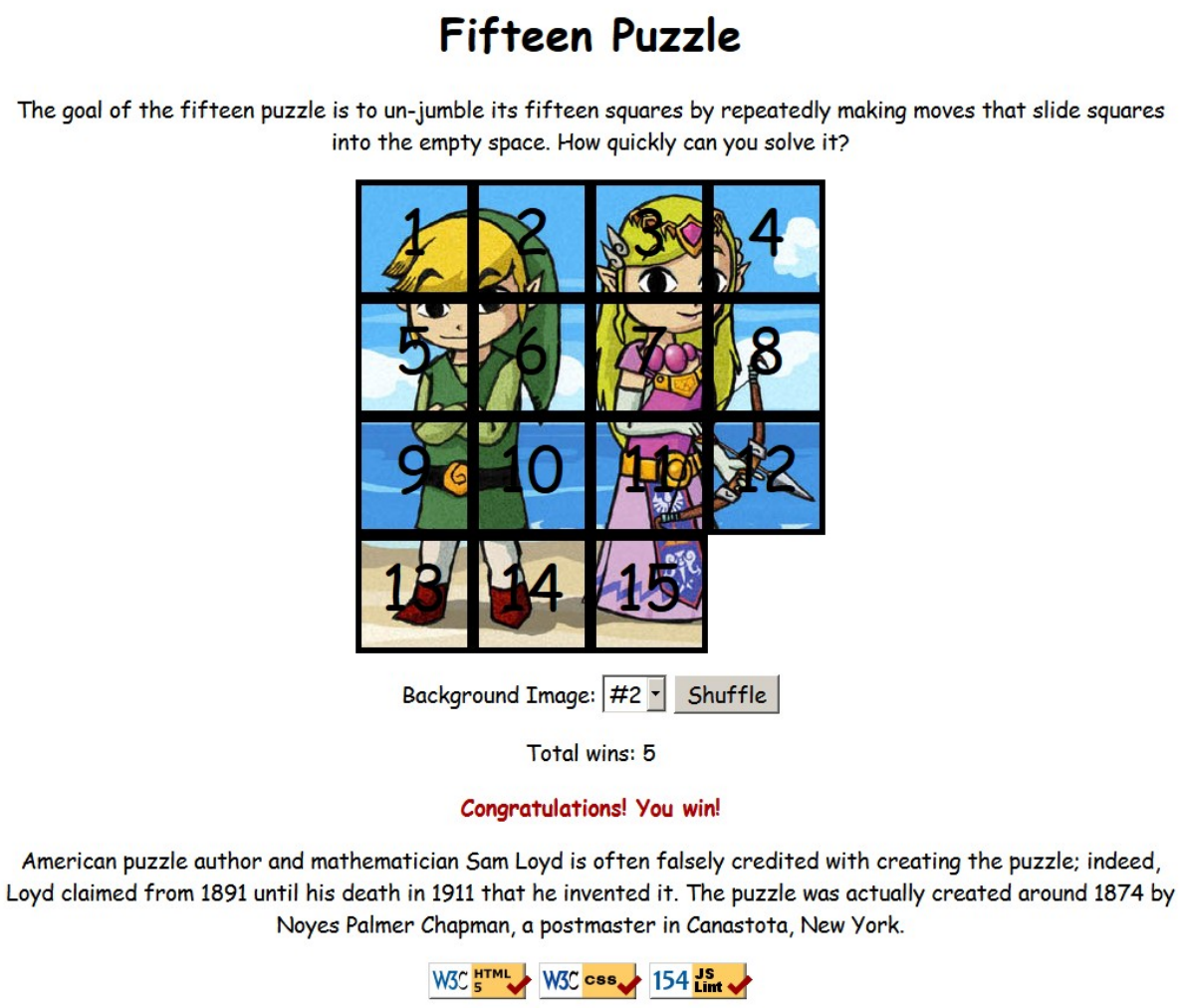
University of Washington, CSE 154
Homework Assignment 8: Fifteen Puzzle (Part B)

This is a continuation of Part A of this program. Part B continues to test your understanding of JavaScript's Document Object Model (DOM) and events, adding some new events to the mix. You'll write the following page:



You will create and turn in the following files:

- **fifteen.html**: the HTML web page
- **fifteen.css**: the CSS styles for your web page
- **fifteen.js**: the JavaScript code for your web page
- **background*.jpg**: your background images, suitable for a puzzle of size 400x400px *(not turned in)*

The appearance and behavior of the page should be as it was previously, except for the changes described in the rest of this document. Note that this part of the assignment will be **graded on internal and external correctness**, so coding style, commenting, etc. are relevant to your grade on Part B.

Some students complete both Part A and B together and are finished with the Part B features before Part A is due. It is okay to submit a functioning solution to Part B as your Part A solution, but you should turn it into both the A and B areas on the class web site so that we know you did submit a Part A solution on time.

In Part A you were not to modify the provided HTML file. In Part B **you are allowed to modify fifteen.html** as needed to help you implement the new features. You can change the page's general appearance, so long as it meets the requirements described in this spec.

1) Shuffle:

Centered under the puzzle tiles is a **Shuffle** button that can be clicked to randomly rearrange the tiles of the puzzle. You should re-enable this button by removing its `disabled` attribute in your HTML page. When the Shuffle button is clicked, the puzzle tiles are rearranged into a random ordering so that the user has a challenging puzzle to solve.

The tiles must be rearranged into a **solvable state**. Some puzzle states are not solvable; for example, the puzzle cannot be solved if you swap only its 14 and 15 tiles. Therefore your algorithm for shuffling cannot simply move each tile to a completely random location. It must generate only solvable puzzle states if you want full credit.

We suggest that you generate a random valid solvable puzzle state by repeatedly choosing a random neighbor of the missing tile and sliding it onto the missing tile's space. Roughly 1000 such random movements should produce a well-shuffled board. Here is a rough pseudo-code of the algorithm we suggest for shuffling.

```
for (~1000 times):
    neighbors = [].
    for each neighbor n that is directly up, down, left, right from empty square:
        if n exists and is movable:
            neighbors.push(n).
    randomly choose an element i from neighbors.
    move neighbors[i] to the location of the empty square.
```

Notice that on each pass of our algorithm, it is guaranteed that one square will move. Some students write an algorithm that randomly chooses any one of the 15 squares and tries to move it; but this is a poor way to shuffle because many of the 15 squares are not neighbors of the empty square. Therefore the loop must repeat many more times in order to shuffle the elements effectively, making it slow and causing the page to lag. This is not acceptable.

Your algorithm should be **efficient**; if it takes more than a second to run or performs a large number of tests and calls unnecessarily, you may lose points. For full credit, your shuffle should have a good random distribution and thoroughly rearrange the tiles and the blank position. You are not required to follow exactly the algorithm above, but if you don't, your algorithm must exhibit the desired properties described in this section to receive full credit.

Your shuffle algorithm will need to incorporate **randomness**. You can generate a random integer from 0 to *N*, or randomly choose between *N* choices, by writing, `parseInt(Math.random() * N)`.

We suggest first implementing code to perform a single random move; that is, when Shuffle is clicked, randomly pick one square near the empty square and move it. Get it to do this once then work on doing it many times in a loop.

In your shuffle algorithm, or elsewhere in your program, you may want access to the DOM object for a square at a particular row/column or x/y position. We suggest you write a function that accepts a row/column as parameters and returns the DOM object for the corresponding square. It may be helpful to you to give an `id` to each square, such as `"square_2_3"` for the square in row 2, column 3, so that you can more easily access the squares later in your JavaScript code. (If any square moves, you will need to update its `id` value to match its new location.)

Use these `id`s appropriately. Don't break apart the string `"square_2_3"` to extract the 2 or 3. Instead, use the `id`s to access the corresponding DOM object: e.g., for the square at row 2, column 3, make an `id` string of `"square_2_3"`.

2) Detect When Puzzle is Solved:

Add code to detect when the user has **solved the puzzle**. When the user moves all of the pieces of the puzzle into their proper places, your page should display a congratulatory message to the user. Any message is fine, but it must contain the word "congratulations" or "win" or "won" as part of its text (so we can find it easily when grading!). It should appear as part of the page, as content that you inject using the DOM; not as an `alert` or pop-up window. For example, you could place an empty `div` in your HTML file and inject text into it when the game is won. The congratulatory message should have some kind of styling to make it stand out on the page, such as colors or font changes. *(Our example screenshot on the first page of this document uses a bold red font saying "Congratulations! You win!")*

Note that the page initially appears with the page in a solved state, but this does not count as having solved the puzzle, and your congratulatory message should not be showing initially. But, for example, if the user moves a single tile from the sorted board out of place and then moves it back, this counts as "solving" the board.

If the "You won!" message is showing and the user makes a move or Shuffles, causing the puzzle to no longer be in its solved state, the message should disappear. The message should show only when the puzzle is in a solved state.

When the user presses the Shuffle button, it is possible (though not very likely) that the shuffle will result in a board that is back in the solved state. The behavior in this case is unspecified. It would probably be better to not count that as having "solved" the puzzle, but we will not test you on that case. But it is also possible that your shuffle code, while it is making its many random moves, will temporarily put the board into an intermediate state where it is back into its proper solved order. Your code should *not* count this as having "solved" the board.

To implement this feature, you will need some way of testing whether the board is currently in its "solved" state after moves. We suggest that you implement this test by examining each puzzle piece's current x/y position relative to what it should be when the board is solved. For example, the top/left corner of the "1" square is x=0, y=0, the "2" square is x=0, y=100, and so on. If all squares are in their expected proper positions, the board is solved.

It can be hard to **test this feature** because it takes a while to actually solve a shuffled board. You can try temporarily reducing the amount of shuffling done by your Shuffle algorithm, such as reducing the loop passes from 1000 to 10, which makes the board closer to its initial state and easier to solve. If you want to try actually solving the puzzle, first get the top/left sides into proper position. That is, put squares number 1, 2, 3, 4, 5, 9, and 13 into their proper places. Now never touch those squares again. Now what's left to be solved is a 3x3 board, which is much easier.

3) Count Wins; Remember Wins and Background Image

Your page should **count the total number of times the player has solved the puzzle**. The page should display a message near the puzzle that initially says, "Total wins: 0". When the user solves the puzzle, you should increase the number displayed by 1 each time.

You should also store the user's progress so that when the user returns to the page at a later date, the number of wins is remembered. That is, if the user solves the puzzle 5 times, closes the browser, and comes back, the page should initially appear with the message, "Total wins: 5". This is a per-browser memory, so if the user accesses the puzzle from another browser or computer, or clears their browser storage settings, the count goes back to 0.

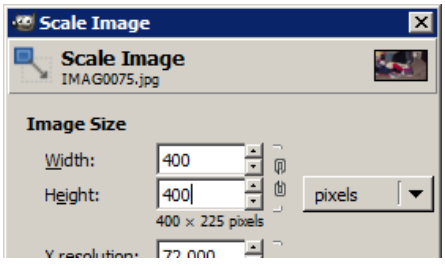
Second, your page should also **remember which background image has been chosen** by the user. If the user has never been to the page before, initially the page shows background image #1. If the user chooses another background, that background should be remembered and should be shown initially when the user comes back to the page later from the same browser. For example, if the user chooses background #2 then leaves the page and comes back, initially background #2 is showing behind the puzzle pieces and #2 is initially selected in the drop-down box.

Implement these two remembered state features using the JavaScript **`localStorage` object**. You can view and modify the state of the local storage in the browser. In Chrome, press F12 to pop up the debug console and then click on the Resources tab. In Firebug, go to the DOM tab, then scroll down until you find `localStorage` in the list.

Please do not overuse/abuse `localStorage`; it should not be used as a hacked form of global variables, and should be used only to store information across page requests, not information used within the current viewing of the page.

4) Provide Your Own Background Images

In Part A you chose from three provided backgrounds. In Part B you should **find three or more images** of your own choosing and use those instead of our images. Store your images on the Webster server in the folder listed on the last page of this document, the same folder as your HTML/JS files. Link to the images using absolute URLs. Provide at least three images; you can provide more than three if you like. If you provide more than three, your drop-down select box should have an `option` for each. If you like, you can change the text in the drop-down box; rather than saying just "#1", "#2", you can show text about your specific images, such as "Mario" or "cute kitten". Your image files do not need to be named **background1.jpg**, **background2.jpg**, etc., but they can be if you like. You will not turn in these images, but you will put them on Webster so that your page can use them.



You can find images online using a service such as Google Image Search. The images you find may not be the right size to use on this page; they should be 400x400 pixels. If you need to resize an image, you can use an image editor such as Photoshop, or download a free image editor such as The Gimp (**www.gimp.org**). To resize an image in The Gimp, click Image, Scale Image... and enter a Width and Height of 400 pixels. (You may need to click the "chain link" icon right of the width/height boxes to allow you to set the width and height individually, since by default it tries to maintain the image's original w/h aspect ratio on a resize operation).

Implementation and Grading:

Unlike on Part A, your Part B will be graded on internal correctness. The entirety of all files you submit are graded on internal correctness, not just the new code you add for Part B. So if you wrote anything with poor style in Part A, you should go back and improve the style of that code before submitting it in your Part B submission.

For full credit, your HTML and CSS code must pass the **W3C validators**. Your JavaScript code should pass our **JSLint** tool with no errors. Your .js file must run in **strict mode** by putting `"use strict";` at the top.

No global variables or functions are allowed on Part B. In order to avoid globals, use the **module pattern** as taught in lecture, wrapping your code in an anonymous function invocation. Even if you use the module pattern, you should limit the amount of "module-global" variables you declare to those that are truly necessary; values should be local as much as possible. If a particular literal value is used frequently throughout your code, declare it as a module-global "constant" variable named `IN_UPPER_CASE` and use the constant throughout your code. Do not store DOM element objects, such as those returned by the `document.getElementById` or `document.querySelectorAll` functions, as module-global variables. As a reference, our own solution has only four module-global variables: the empty square's row and column (initially both are set to 3), the number of rows/cols in the puzzle (4), and the pixel width/height of each tile (100). The last two are essentially constants.

Follow reasonable style guidelines similar to a CSE 14x programming assignment. In particular, minimize global variables, avoid redundant code, and use parameters and return values properly. Make extra effort to minimize **redundant code**. Capture common operations as functions to keep code size and complexity from growing. You can reduce your code size by using the `this` keyword in your event handlers.

Your JavaScript code should have adequate **commenting**. The top of your file should have a descriptive comment header describing the assignment, and each function and complex section of code should be documented.

Format your code similarly to the examples from class. Properly use whitespace and indentation. Use good variable and method names. Avoid lines of code more than 100 characters wide. For reference, our .js file has roughly 160 lines (110 "substantive"), and our CSS file has roughly 50 lines.

Separate content (HTML), presentation (CSS), and behavior (JS). Your JS code should **use styles and classes from the CSS** rather than manually setting each style property in the JS. For example, rather than setting the `.style` of a DOM object, instead, give it a `className` and put the styles for that class in your CSS file. Style properties related to x/y positions of tiles and their backgrounds are impractical to put in the CSS file, so those can be in your JS code.

Use **unobtrusive JavaScript** so that no JavaScript code, `onclick` handlers, etc. are embedded into the HTML code.

You should not use any external **JavaScript frameworks or libraries** such as jQuery to solve this assignment.

Do not place your solution on a public web site. Submit your own work and follow the course misconduct policy.

Put your files on **Webster** at the following URL (note the "b" after "hw8" in the URL, to separate it from Part A):

https://webster.cs.washington.edu/*your_uwnetid*/hw8b/fifteen.html