American University, CSC 435 Web Programming

Homework Assignment 5: Fifteen Puzzle

Total points: 100pts.

Due date: Monday, April 4, 2016, end of the day. No late submissions are acceptable.

Additional features: 3 pts each (but you must finish one additional feature to receive full mark).

Team work: up to 2 people a team.

Please write a read me file for the game that includes your team members name and contributions.

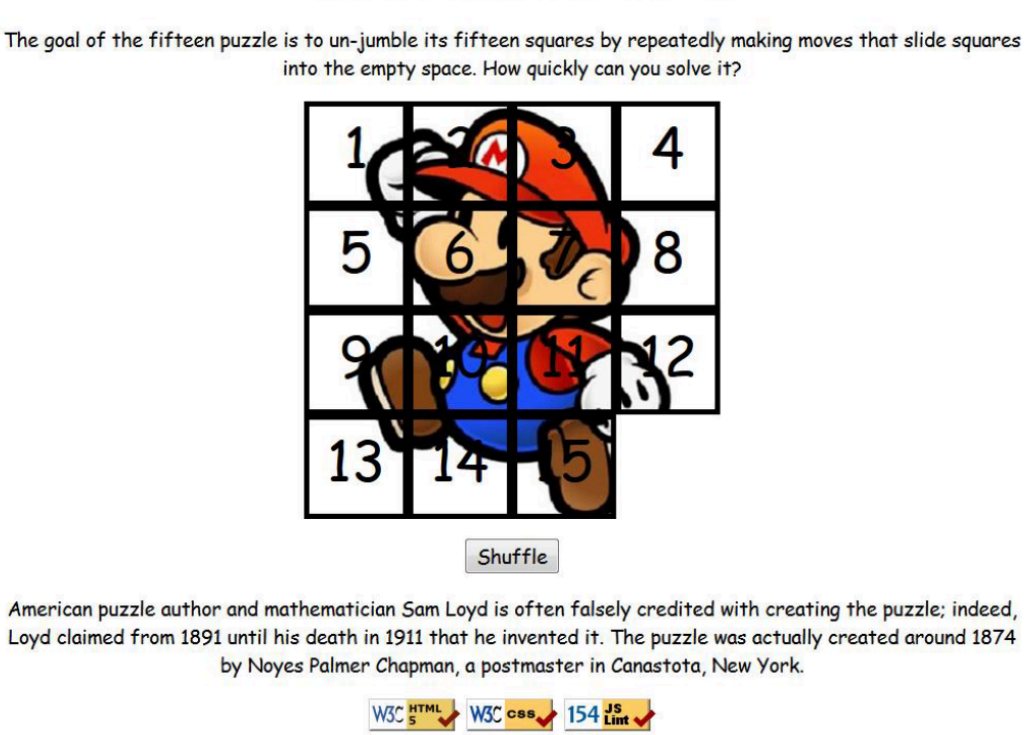
Upload your work onto a web server.

Submission per team: one submission. Please include both of your last names in the submission if you work in a team.

Please do not look it up for solutions on the Internet. Try as hard as you can do solve this.

The goal of the assignment is about JavaScript's Document Object Model (DOM) and events. This is the most comprehensive (and the last) JavaScript exercise. Use the opportunity to get yourself into a Pro.

You will write the following page:



Background Information:

The Fifteen Puzzle (also called the Sliding Puzzle) is a simple classic game consisting of a 4x4 grid of numbered squares with one square missing. The object of the game is to arrange the tiles into numerical order by repeatedly sliding a square that neighbors the missing square into its empty space. Here are some references:

Wikipedia:

https://en.wikipedia.org/wiki/15_puzzle

Mathworld:
http://mathworld.wolfram.com/15Puzzle.html

iPHone app:
https://itunes.apple.com/us/app/15-puzzle-challenge.-free/id416789249?mt=8

You will write the CSS and JavaScript code for a page **fifteen.html** that plays the Fifteen Puzzle. You will also submit a **background image** of your own choosing, displayed underneath the tiles of the board. Choose any image you like, so long as its tiles can be distinguished on the board. Turn in the following files:

- **fifteen.js**, the JavaScript code for your web page
- **fifteen.css**, the CSS styles for your web page
- **background.jpg**, your background image, suitable for a puzzle of size 400x400px
  (go find any image you like, such as by Google Image Search; but don't use the Mario image above)

You will not submit any **.html** file, nor directly write any HTML code. We will provide you with the HTML code to use, which should not be modified. (Download the **.html** file to your machine while writing your JavaScript code, but your code should work with the provided files unmodified.) You will write JavaScript code that interacts with the page using the DOM. To modify the page's appearance, write appropriate DOM code to change styles of on-screen elements by setting classes, IDs, and/or style properties on them.

For full credit, **you must complete one extra feature** listed in a separate document.

Tip for playing the game: First get the entire 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.

Appearance Details:

All text on the page is displayed in a "cursive" font family, at a default font size of 14pt. Everything on the page is centered, including the top heading, paragraphs, the puzzle, the Shuffle button, and the W3C buttons at bottom.

In the center of the page are **fifteen tiles** representing the puzzle. The overall puzzle occupies 400x400 pixels on the page, horizontally centered. Each puzzle tile occupies a total of 100x100 pixels, but 5px on all four sides are occupied by a black border. This leaves 90x90 pixels of area inside each tile. The HTML file given to you does not contain the fifteen div elements to represent the puzzle pieces, and you are not supposed to modify the HTML file; so you will have to create the puzzle pieces and add them to the page yourself using the JavaScript DOM. Initially the page should appear with the puzzle in its properly arranged order like in the screenshot on the first page of this spec, with 1 at top-left, 4 at top-right, 13 at bottom-left, the empty square at bottom-right, and so on.

Each tile displays a number from 1 to 15, in a 40pt font. Each tile displays part of the image **background.jpg**, which you should put in the same folder as your page. Which part of the image is displayed by each tile is related to that tile's number. The "1" tile shows the top-left 100x100 portion of the image. The "2" tile shows the next 100x100px of the background that would be to the right of the part shown under the "1" tile, and so on.

Your **background image** appears on the puzzle pieces when you set it as the background-image of each piece. By adjusting the background-position of each div, you can show a different part of the background on each piece. One confusing thing about background-position is that the x/y values shift the background behind the element, not the element itself. The offsets are the negation of what you may expect. For example, if you wanted a 100x100px div to show the top-right corner of a 400x400px image, set its background-position property to -300px 0px . The following is a complete listing of the exact background-position values each location on the board should have:

0px 0px	-100px 0px	-200px 0px	-300px 0px
0px -100px	-100px -100px	-200px -100px	-300px -100px
0px -200px	-100px -200px	-200px -200px	-300px -200px
0px -300px	-100px -300px	-200px -300px	

Centered under the puzzle tiles is a **Shuffle** button that can be clicked to randomly rearrange the tiles of the puzzle. See the "Shuffle Algorithm" section of this spec for more details about implementing the shuffle behavior.

All other style elements on the page are subject to the preference of the web browser. The screenshots in this document were taken on Windows in Firefox, which may differ from your system.

Playing the Game:

When the mouse button is pressed on a puzzle square, if that square is next to the blank square, it is moved into the blank space. If the square does not neighbor the blank square, no action occurs. Similarly, if the mouse is pressed on the empty square or elsewhere on the page, no action occurs.

When the mouse **hovers** over a square that can be moved (neighbors the blank spot), its border and text color should become red. Also, the mouse cursor should change into a **"hand" cursor** pointing at the square (do this by setting the CSS cursor property to pointer.) Once the cursor is no longer hovering on the square, its appearance should revert to its original state. When the mouse cursor hovers over a square that cannot be moved, it should use the system's standard **arrow cursor** (set the cursor property to default) and it should not have the red text or borders or other effects. (You may find it useful to use the :hover CSS pseudo-class to help deal with hovering.)

The game is not required to take any particular action when the puzzle has been won. You can decide if you'd like to pop up an alert box congratulating the user or add any other optional behavior to handle this event. (See Extra Features spec if you would like to implement end-of-game behavior.)

Shuffle Algorithm:

Centered under the puzzle tiles is a **Shuffle** button that can be clicked to randomly rearrange the tiles of the puzzle. 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.
```

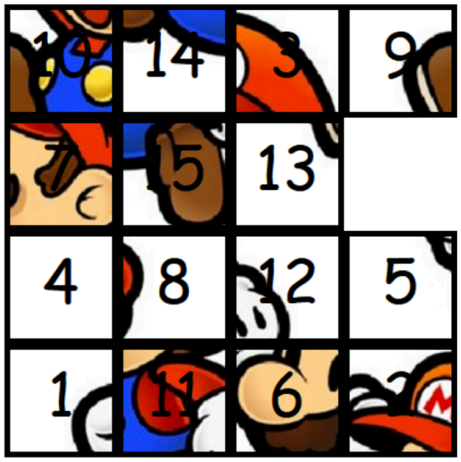
Notice that on each pass of our algorithm, it is guaranteed that one square will move. Some students write an algorithm that randomly chooses any one of the 15 squares and tries to move it; but this is a poor way to shuffle because many of the 15 squares are not neighbors of the empty square. Therefore 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 1 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 *K*, which is helpful to randomly choose between *K* choices, by writing, parseInt(Math.random() * K) .

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.

The goal of the fifteen puzzle is to un-jumble its fifteen squares by repeatedly making moves that slide squares into the empty space. How quickly can you solve it?



American puzzle author and mathematician Sam Loyd is often falsely credited with creating the puzzle; indeed, Loyd claimed from 1891 until his death in 1911 that he invented it. The puzzle was actually created around 1874 by Noyes Palmer Chapman, a postmaster in Canastota, New York.

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

Development Strategy:

Students generally find this to be a tricky assignment. Here is a suggested ordering for tackling the steps:

- Make the fifteen **puzzle pieces appear** in the correct positions without any background behind them.
- Make the correct parts of the **background** show through behind each tile.
- Write the code that **moves a tile** when it is clicked from its current location to the empty square's location. Don't worry initially about whether the clicked tile is "movable" (whether it is next to the empty square).
- Write code to determine whether a **square can move** or not (whether it neighbors the empty square). Implement a highlight when the mouse hovers over movable tiles. Track where the empty square is at all times.

Additional features:

You are required to complete one (1) extra feature for full credit.

Extra Feature #1. End-of-game Notification

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.

Extra Feature #2. Ability to Slide Multiple Squares at Once

Make it so that if you click any square in the empty square's row or column, even ones more than one spot away from the empty square, all squares between it and the empty square slide over. (Much more pleasant to play!) If you do this extra feature, make it so that all movable squares (including ones several rows or columns away from the empty square) highlight on hover as described before.

Extra Feature #3. Game Timer

Keep track of the game time elapsed in seconds and the total number of moves, and when the puzzle has been solved, display them along with the best time and fewest total moves seen so far.
You can implement this feature using a timer and the setInterval function in JavaScript. It's fine to just show the total seconds needed, or if you want to split it into minutes and seconds, that is fine, too.

Extra Feature #4. Animations and/or Transitions

Instead of each tile immediately appearing in its new position, make them animate. You can do any sort of animation or other styling, as long as the game ends up in the proper state after a reasonable amount of time.

It can be tricky to implement animation yourself. You could do animation using a timer such as with the setInterval function. Or you could use a CSS3 transformation. One source of bugs is when the user tries to click pieces quickly before the previous move is done animating. Your puzzle should not have buggy behavior or errors if the user tries to do this.

Extra Feature #5. Different Puzzle Sizes

Place a control on the board to allow the game to be broken apart in other sizes besides 4x4,

such as 3x3 or 6x6. It doesn't need to be possible to change the board size in the middle of a game; changing the size should reset the board back to an initial "solved" state for that board size. You can either use a drop-down box of available board sizes, or allow the user to type in their own board size.

The overall puzzle size should stay at 400x400. This means that each square's size must change if you change the number of rows and columns; for example, if it is a 5x5 game, each square is 80x80 rather than 100x100. Certain sizes don't evenly divide into 400, such as 6x6. In such a case, just choose the nearest integer to the proper size.