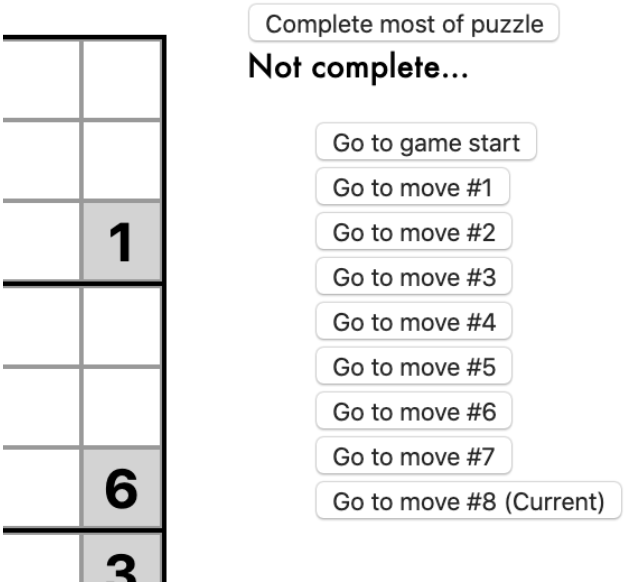
React Sudoku

For this assignment, we will be building a react-based Sudoku web application. For those unfamiliar with the game, here is a video explaining the
concepts: https://www.youtube.com/watch?v=OtKxtvMUahA

This game will also feature a "time-travel" functionality where the user can see a list of every "move" they've made and click on a move to bring the game board back to the state it was at that move.



If you haven't already, watch the introduction video recording to this assignment: Video Recording - Introduction to Homework 2

Do **not** procrastinate this assignment as there is a fair amount of work here! But don't freak out. We will **not** be implementing the logic of Sudoku ourselves, opting to rely on the **sudoku** npm library instead. This project is also based on React's Tutorial where a tic-tac-toe game is built with the same "time-travel" functionality. Going through that tutorial and using that code as a foundation will ensure you can get it done in under 12 days - no sweat.

Unlike a traditional Sudoku game using digits 1-9, the sudoku npm library uses digits 0-8. This is totally fine. We are computer scientists after all, and we know 0-indexed is the way to go. You can change this, but I don't recommend you take on the additional work for yourself until the end.

You will **not** have to handle things like disallowing the user to enter an incorrect number into a cell. The user can enter *whatever* input they would like on the non-hint fields regardless of how wrong it is. Our only concern is to tell the user when they've finally got it *right*.

Requirements

Submission Requirements

The submission should be a ZIP file with a name scheme of "LastnameFirstname-HW2.zip"

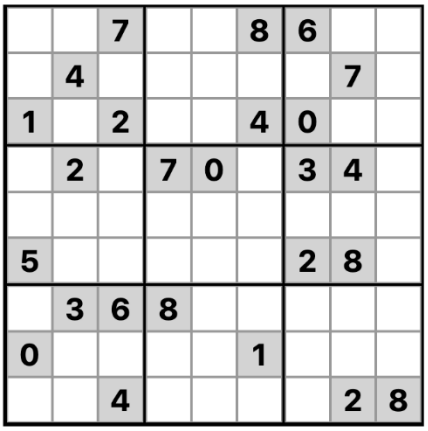
Inside the ZIP, you should have your create-react-app generated project. **You do not need to upload the npm_modules folder (it will be massive)!** Your dependencies are already declared in the `package.json` file, so I can get them locally by just executing **`npm install`**

The way I will launch the app to grade/verify will be executing the **`npm start`** command. If your website does not launch and work correctly after executing this command within your project folder, points will be subtracted.

Grid

The webpage must have a 9 x 9 sudoku grid. How you style it is mostly up to you, but it should have the core structure of 9 x 9 cells to enter numbers into. There should also be clearly visible sub-grids of 3x3 cells. It is very helpful to have borders around each 3 x 3 sub-grid to help

solve the puzzle.



The board should be initialized with a partial solution so the user can start applying the rules of Sudoku to complete the board.

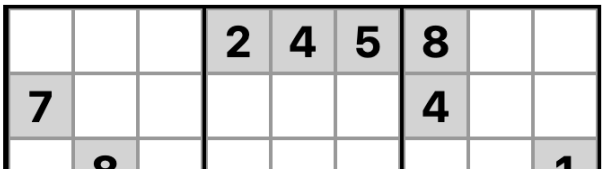
The initialized hint cells should **not** be modifiable, and there should be some styling applied to make it obvious these are the "hints" the game was started with. I used a grey background, but you are open to do things like text styling (perhaps bolded, or a different font).

User input / "moves"

1. The user should only be allowed to enter one number into each cell. This means no letters, punctuation, etc. Just single digit integers. When a digit is entered into a cell, this counts as a "move".
2. The user should also be able to remove their input on a cell with backspace/delete. This also counts as a "move".
3. Somewhere on the web page should be a list of "moves" that dynamically updates as the user plays the game (makes moves.)
4. Clicking on one of the "moves" will set the board state back to how it was after the user conducted that particular move.
5. The "move" that is reflected currently on the game board should be indicated in some way. In my example, I append " (Current)" to the button label. But you can use a different color, etc.

Move history example

For example: when the game first launches there will only be one "move" - game start (empty board.)

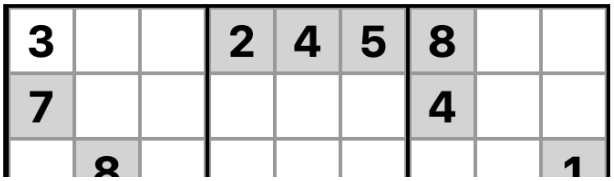


Complete most of puzzle

Not complete...

Go to game start (Current)

When the user creates their *first move* by entering a number into a cell (let's say the first cell all the way top left - let's call it cell #1), there will now be 2 moves - [Game start, Move #1]



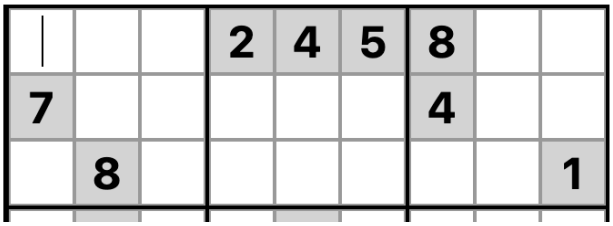
Complete most of puzzle

Not complete...

Go to game start

Go to move #1 (Current)

When the user backspaces/deletes 3 from the first top left cell, this counts as their second "move"



Complete most of puzzle

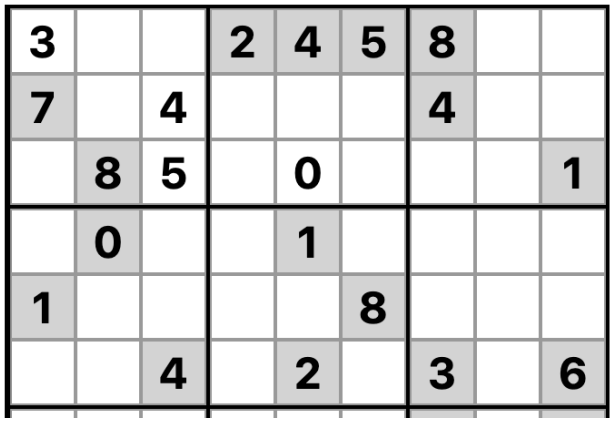
Not complete...

Go to game start

Go to move #1

Go to move #2 (Current)

The user can continue playing the game, adding moves.



Complete most of puzzle

Not complete...

Go to game start

Go to move #1

Go to move #2

Go to move #3

Go to move #4

Go to move #5

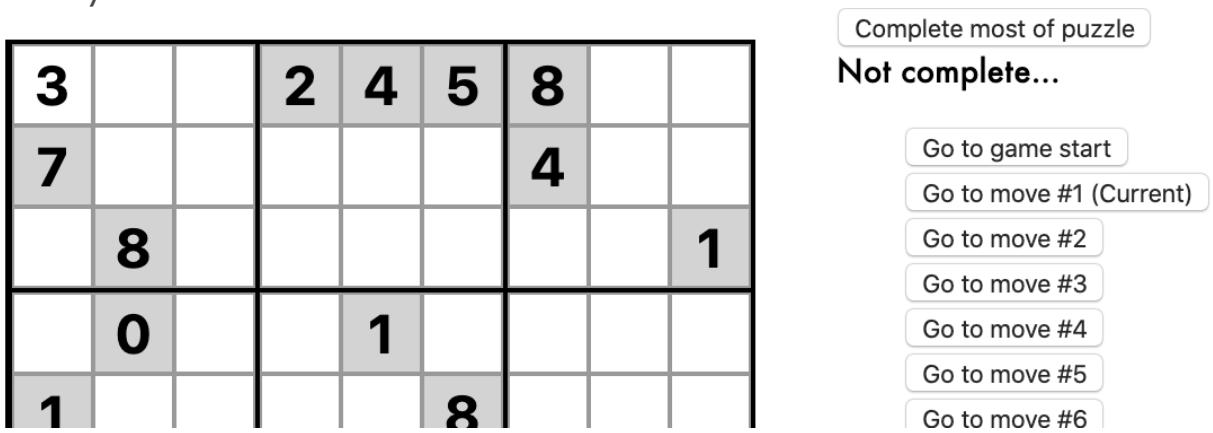
Go to move #6

Go to move #7

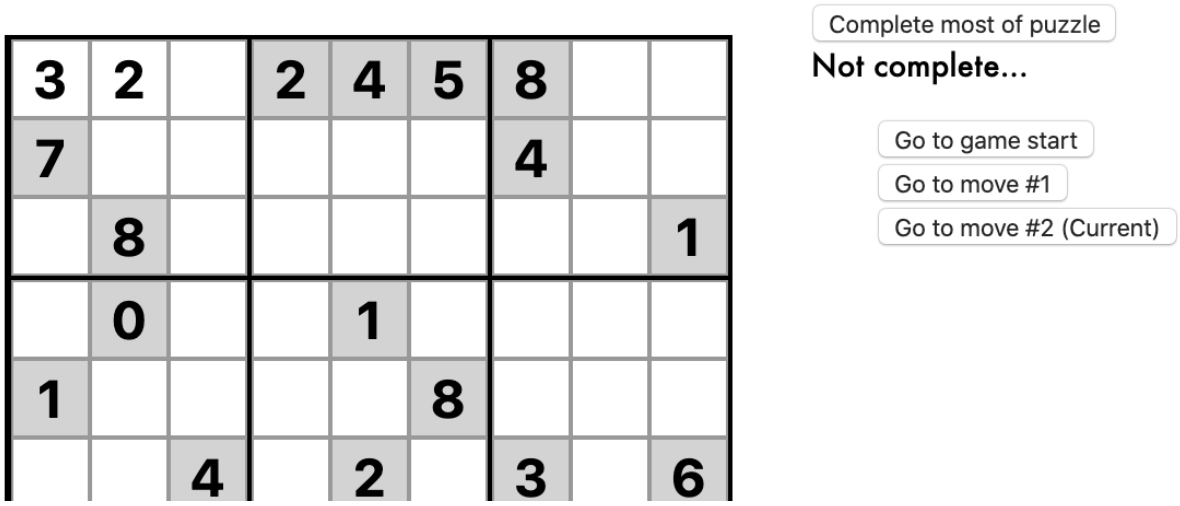
Go to move #8 (Current)

When the user presses the button to jump back to move #1, the game board should reflect the state of the board in that moment of "move"

history



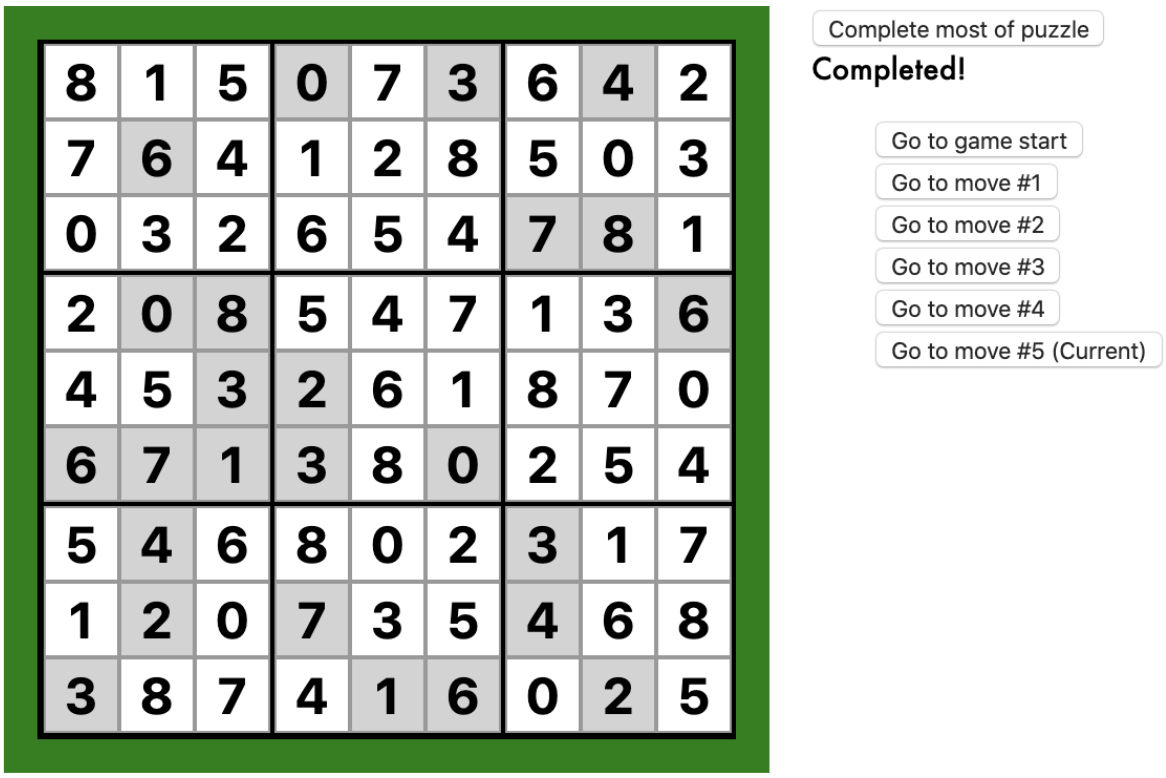
From this point, the user can either press other "moves" to set the board state the desired move, or start making additional "moves" that will start overwriting the history from their current point. Put another way - if the user fills the cell to the right of Cell #1 (let's call it Cell #2), then the history after the current move will be discarded and there will now be a *new* Move #2 to signify the value of "2" being entered into Cell #2.



Successful Game Notification

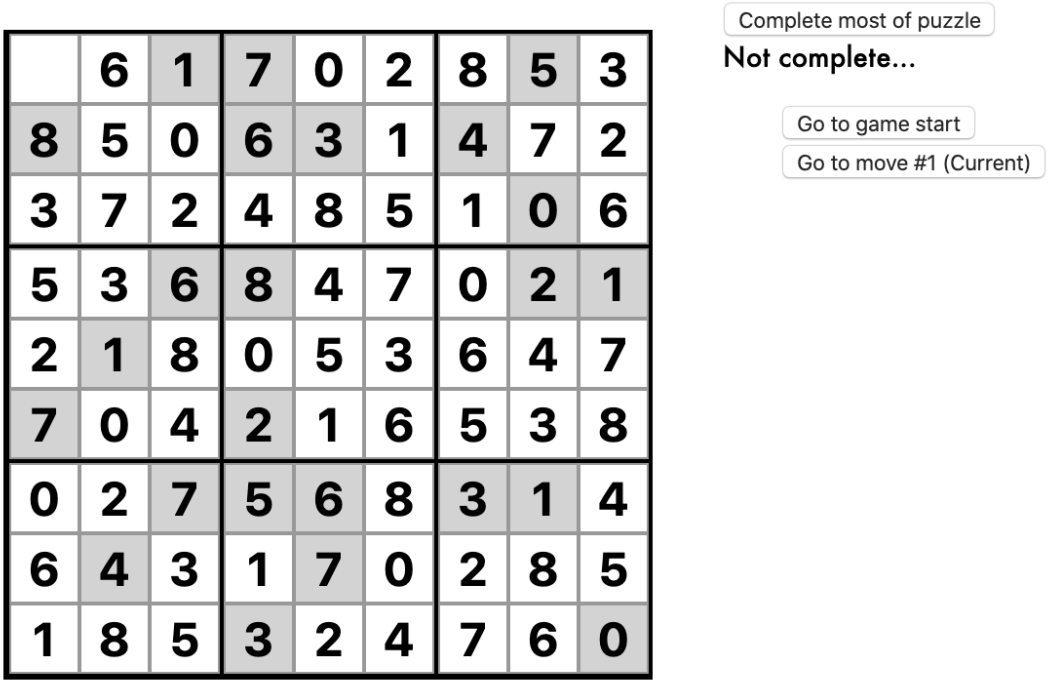
As mentioned, we're not validating and rejecting player input that violates the rules of Sudoku. We're only going to care about notifying the user when the game is completed and satisfies the rules of Sudoku. You can do this *however you like* as long as it's obvious when we "won." Here is my example where I have both a "status" label up on the top

right that always reflects game status (either "Not complete..." or "Completed!"), and a green background on the board padding area to signify you're done. There are no requirements on game behavior once the user has completed it successfully.



Cheat Code Button

This assignment would be impossible to complete if you had to solve a sudoku puzzle to completion every time you wanted to test your successful game notification feature. So there should be a button somewhere on the page that will fill almost the entire solution, letting you (the developer) fill in just one final cell to complete the game. In my example, I've added a button labeled "Complete most of puzzle" where I take the puzzle solution array and overwrite the 0 array index with the value that represents no input. I then set the most recent game state in the history to this "almost solution" state, leaving just Cell #0 without a value. You don't have to worry if this is buggy because a Hint cell is rendered as Cell #0. If Cell #0 is a Hint, then I'll just refresh the assignment until it's not.



	6	1	7	0	2	8	5	3
8	5	0	6	3	1	4	7	2
3	7	2	4	8	5	1	0	6
5	3	6	8	4	7	0	2	1
2	1	8	0	5	3	6	4	7
7	0	4	2	1	6	5	3	8
0	2	7	5	6	8	3	1	4
6	4	3	1	7	0	2	8	5
1	8	5	3	2	4	7	6	0

How to complete the assignment

Complete the React Tutorial

This assignment is essentially a variation of the Tic-Tac-Toe tutorial on React's website. Go through it and skim the documentation pages to familiarize yourself with props, state, JSX, and the render method.

Important note: In the tutorial, there are two setup options. ***Go with Option 2***. Generate yourself a new react app, and launch it in the browser. I recommend also setting up Visual Studio Code with a debugger, so when you run into issues you can set a breakpoint and step through your code in the debugger.

Play with the sudoku library

In the Lecture 2 content area, there is an article on how to set up a basic sandbox to play with an npm library. Set up your sandbox with the sudoku npm library and try creating a puzzle, and then obtaining the solution for the puzzle. Notice how the puzzle is represented as an

array of 81 elements. This represents the 81 cells of the sudoku game (9x9 grid = 81 cells) in a left to right, top to bottom fashion.

Set up your HTML sudoku grid

Google is your friend here - there are some questions & answers on how to create a sudoku grid on the web. If I were to pick, my preference would be the <table> solution that may be found on a StackOverflow thread somewhere...

Start small - wire up each one of your cells to first display their "index", and perhaps `console.log` their own IDs when you click them. The structure should map to the game array like so:

0	1	2	3	4	5	6	7	8
9	10	11	12	13	14	15	16	17
18	19	20	21	22	23	24	25	26
27	28	29	30	31	32	33	34	35
36	37	38	39	40	41	42	43	44
45	46	47	48	49	50	51	52	53
54	55	56	57	58	59	60	61	62
63	64	65	66	67	68	69	70	71
72	73	74	75	76	77	78	79	80

Incorporate Game Initialization

Initialize the game board with the initial state of the puzzle provided by the sudoku library. Indicate the fields as hints, and have them be non-modifiable. **Tip**: a hint cell and an input cell do not necessarily have to a single type of react component. The Hint component can have different CSS classes applied to it, have a "disabled" attribute applied, etc. etc.

Start updating the game board state

When a user inputs a value into a cell, update the global game state & history. Use the React tutorial as a guide, as the mechanics are very similar.

Helpful Resources & Tips

First and foremost - remember why this framework is called "React." Shift your mind away from grabbing individual DOM elements and modifying them, and instead think about modifying *state* and let React do all the hard work for you. Remember, making one "move" is creating an entirely new board and adding it to the history. Let React handle creating the DOM elements for you! Let it re-render everything! The point is that the library is smart enough to manipulate the DOM efficiently on your behalf. All you need to do is tell React how to render a DOM tree from state, and then update that state as needed using `setState`.

React Tutorial - This will serve as the foundation for this assignment. Go through it, download the final code, and start modifying it.

Conditional Rendering - Sometimes we want to render a different component based on a condition. An example could be rendering Hint cells vs regular input cells on our Sudoku grid. This article walks you through conditional rendering strategies.

This is why we need to bind event handlers - You will inevitably run into the following bug: you have a button and you've set an onClick handler to a function defined in your component, e.g. `onClick={this.handleClick}>` And when clicking that button - nothing

happens! And in your console, it seems that the handler is undefined. Why is this? And how do we solve that? This article explains it.

How to manage React state with Arrays - You may be wondering why you see the tutorial code call methods like slice() on arrays to make a copy of them. React's smart virtual DOM uses comparisons between old and new state to determine what to render. Passing in the exact same array may cause React to think nothing has changed, despite the contents of the array being different. So by passing a new array, you are informing React that it's been modified and should be compared. This article goes into a bit more detail on the subject.

Thinking in React - This will help you understand how to structure your React application and components. What type of information should I pass through props? What should I store in state? And in what component should the state live in? This article addresses it all.