Connect-4 Solver

Implement the famous game in JavaScript

Suppose an 8*6 Connect-Four table is given. Each cell of the table is either empty (`null`), or contains the player's number from the possible values `1` and `2`. Determine if any player has won the game by connecting four of their symbols horizontally or vertically. For simplicity, ignore diagonal matches.

Solution:

As there is no example data, we have to model the table ourselves.

```
const createEmptyTable = () =>
    new Array( 8 ).fill( null ).map(
        () => new Array( 6 ).fill( null )
    );
```

This simple arrow function returns an empty 6*8 array:

```
let table = createEmptyTable()
console.log(table)
```

It is evident that we will need a function that checks all elements of the array for four consecutive matches. I encourage you to implement this function yourself. Reading my solution will be more beneficial to you in case you put in the effort to understand what is going on.

```
const checkElements = (
    [head, ...tail],
    matchCount = 0,
    lastElement = null
    lastElement = null
) => {
    if ( matchCount === 3 && head === lastElement ) return true;
    if ( tail.length === 0 ) return false;
    if ( head === null ) return checkElements( tail );
    if ( head === lastElement ) {
        return checkElements( tail, matchCount + 1, head );
    }
    return checkElements( tail, 1, head );
}
```

The solution is based on simple recursion. If we find four matches, the function returns `true`.

If there are no more elements left, and there is no match possible anymore, the function returns `false`. Note that the second `if` is only reachable if the first condition is evaluated to `false`. In general, due to the `return` statements, we know that in each line, all `if` conditions of the lines above are `false`.

In the last two conditions, we check if the head is `null`, or matches the sequence we are looking for. In both cases, our task is to recursively call our function with the correct argument list. Eventually, in the last line, we know that `head` contains a non-null element that is different than the last element. In this case, we have to restart the matching process.

Note that if you know how regular expressions work, you could simply write a regex to perform the same work:

```
const checkElements = arr =>
    /([12]),\1,\1,\1/.test( arr.toString() );
```

`[12]` is an arbitrary character that is either a `1` or a `2`. We capture it using parentheses, then repeat the captured character using the `\1` capture group reference. We insert the commas in-between. If you want to brush up your regex skills, check out my articles on regular expressions.

We can use the `reduce` array method to match columns:

```
const checkColumns = table =>
```

```
    table.reduce(
        (hasMatch, column) => hasMatch || checkElements( column ),
        false
    );
```

If you still have trouble interpreting what is going on, insert a console log inside the arrow function, and study the logged output:

```
const checkColumns = table =>
    table.reduce(
        (hasMatch, column) => {
            console.log( hasMatch, column );
            return hasMatch || checkElements( column );
        },
        false
    );
```

We now need to check the rows. We could google how transposing an array works in JavaScript. However, there is no need to make the solution more complicated than it is. A simple `for` loop will do:

```
const checkRows = table => {
    for ( let i = 0; i < table[0].length; ++i ) {
        let rowArray = table.map( column => column[i] );
        if ( checkElements( rowArray ) ) return true;
    }
    return false;
}
```

The function works as follows: the `for` loop goes through each element of the first column. Note that in a table, each column has the same number of elements. For this reason, we can form `rowArray` by taking the `i` th element from each column using the `map` function. The map function takes each column of the table, and substitutes it with the `i` th element in the column.

Now that we have an array of consecutive elements, we can use our checkElements function to derive the matches. As soon as we find a match, we

can return true. If execution reaches the end of the for loop, we know that none of the rows matched. Therefore, we can safely return false.

Let’s create a function that checks the whole table for matches:

```
const checkTable = table =>
    checkRows( table ) || checkColumns( table );
```