Exercise on Spread Operator and Rest Parameters

Get a hang of the spread operator and rest parameters by trying out these exercises. Remember, the point is to think differently and move away from ES5 conventions.

Exercise 1:

Make a shallow copy of an array of any length in one destructuring assignment! The given array contains random integers, can be of any length and is called `originalArray`. Call the new array you make `clonedArray`. Remember to use this exact spelling or your code won't compile.

If you don't know what a shallow copy is, make sure you read about it, as you will need these concepts during your programming career. I can highly recommend my article on Cloning Objects in JavaScript.

```
// Original array randomly generated
console.log(originalArray);
let clonedArray = []
```

Exercise 2:

Determine the value logged to the console without running it.

index.js

explanation.txt

```
let f = () => [..."12345"];
let A = f().map( f );
console.log( A );
```

Explanation

An array of five vectors of `['1', '2', '3', '4', '5']` is printed out as a table. The mechanism is the exact same as the explanation in the next exercise. The

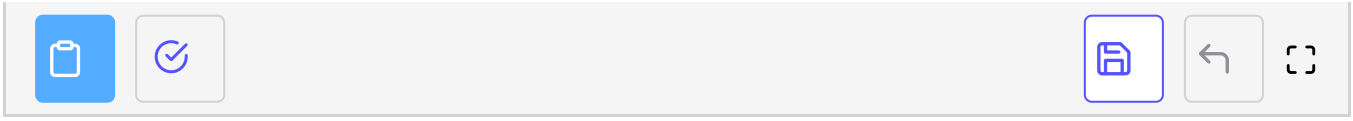
function `f` creates the array `['1', '2', '3', '4', '5']`. In `f().map( f )`,

only the length of `f()` matters, as the values are thrown away by the map function. Each element of the `f()` array is mapped to the array `['1', '2', '3', '4', '5']`, making a 2 dimensional array of vectors `['1', '2', '3', '4', '5']`.

Exercise 3:

Create an 10x10 matrix of `null` values.

```
// fill this array
let nullArray = [];
```



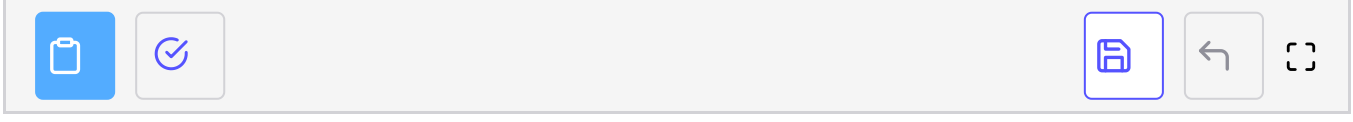
Explanation

Study the fill method for more details here. We create a null vector of size 10 with the `nullVector` function. The values of the first `nullVector()` return value don't matter, as we map each of the ten elements to another value. The `nullVector` mapping function throws each null value away, and inserts an array of ten nulls in its place.

Exercise 4:

Rewrite the `sumArgs` function given in ES2015, using a rest parameter and arrow functions.

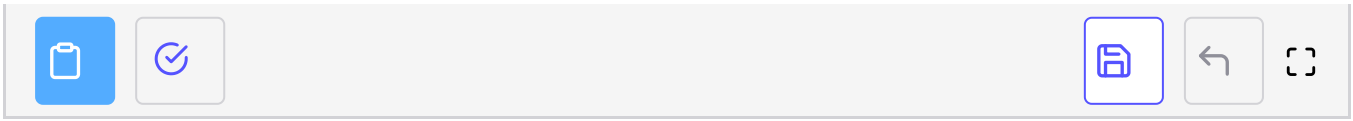
```
function sumArgs() {
    var result = 0;
    for( var i = 0; i < arguments.length; ++i ) {
        result += arguments[i];
    }
    return result;
}
```



Exercise 5:

Complete the following ES2015 function that accepts two String arguments, and returns the length of the longest common substring in the two strings. The algorithmic complexity of the solution does not matter. Remember to keep the name as `maxCommon()` or your code won't compile.

```
let maxCommon = ([head1,...tail1], [head2,...tail2], len = 0) => {
    if ( typeof head1 === 'undefined' ||
         typeof head2 === 'undefined' ) {
         /* Write code here */
    }
    if ( head1 === head2 ){
      /* Write code here */
    }
    let firstBranch = 0 /* Write code here */
    let secondBranch = 0 /* Write code here */
    return Math.max( ...[len, firstBranch, secondBranch ] );
}
```



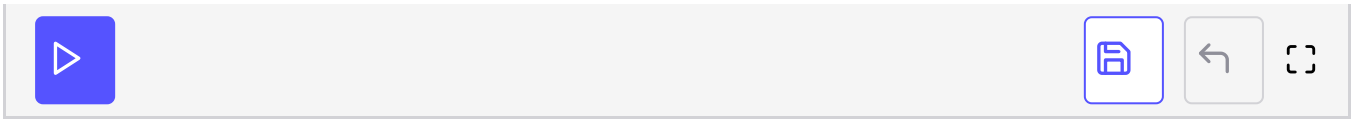
Explanation

We will use an optional `len` argument to store the number of character matches before the current iteration of maxCommon was called. We will use recursion to process the strings. If any of the strings have a length of 0, either `head1`, or `head2` becomes undefined. This is our exit condition for the recursion, and we return `len`, i.e. the number of matching characters right before one of the strings became empty. If both strings are non-empty, and the heads match, we recursively call `maxCommon` on the tails of the strings, and increase the length of the counter of the preceding common substring sequence by 1. If the heads don't match, we remove one character from either the first string or from the second string, and calculate their `maxCommon` score, with `len` initialized to 0 again. The longest string may either be in one of these branches, or it is equal to `len`, counting the matches preceding the current strings `[head1,...tail1]` and `[head2,...tail2]`.

```
maxCommon = ([head1,...tail1], [head2,...tail2], len = 0) =>
{
  if ( typeof head1 === 'undefined' || typeof head2 === 'undefined' )
  {
    return len;
  }
```

```
  if ( head1 === head2 )
    return maxCommon( tail1, tail2, len+1 );
  let firstBranch = maxCommon( tail1, [head2, ...tail2], 0 );

  let secondBranch = maxCommon([head1,...tail1], tail2, 0 );
  return Math.max( ...[len, firstBranch, secondBranch ] );
}
```



Note that this solution is very complex, and requires a magnitude of #(s1)! * #(s2)! steps, where s1 and s2 are the two input strings, and #(...) denotes the length of a string. For those practicing for a Google interview, note that you can solve the same problem in steps of O(#(s1) * #(s2)) magnitude using dynamic programming.