The following is a challenge that I found on CodeWars (a free coding practice site). It hit a nice overlap between my newbie-developer sensibilities and my fond memories of logic puzzles and mathematics challenges in grade-school AG classes, so I decided to record my process through the problem, from start to finish.

Problem author's profile: http://www.codewars.com/users/xcthulhu
Problem location: http://www.codewars.com/kata/sierpinskis-gasket/javascript

Problem statement: "Write a function that takes an integer *n* and returns the *nth* iteration of the fractal known as Sierpinski's Gasket. The fractal is composed entirely of 'L' and white-space characters; each character has one space between it and the next (or a newline)."

The problem then goes on to give examples of the 0th, 1st, 2nd, and 3rd iterations. I went ahead and generated a fourth iteration to help my brain catch patterns, and temporarily replaced the white-space characters with bullets to aid in counting:

0:
```
L
```

1:
```
L
L•L
```

2:
```
L
L•L
L•••L
L•L•L•L
```

3:
```
L
L•L
L•••L
L•L•L•L
L•••••••L
L•L•••••L•L
L•••L•••L•••L
L•L•L•L•L•L•L•L
```

4:
```
L
L•L
L•••L
L•L•L•L
L•••••••L
L•L•••••L•L
L•••L•••L•••L
L•L•L•L•L•L•L•L•L•L•L•L•L•L•L•L
L•••••••••••••••L
L•L•••••••••••••L•L
L•••L•••••••••••L•••L
L•L•L•L•••••••••L•L•L•L
L•••••••L•••••••L•••••••L
L•L•••••L•L•••••L•L•••••L•L
L•••L•••L•••L•••L•••L•••L•••L
L•L•L•L•L•L•L•L•L•L•L•L•L•L•L•L•L•L•L•L•L•L•L•L•L•L•L•L•L•L
```

My first step was to do some basic assessment of the situation, gathering observations without worrying too much what would be useful and what wouldn't:

- Each iteration was 2^n lines long
- Each iteration contained 3^n 'L' symbols

- Each iteration contained three copies of the previous iteration (one on top exactly as in the previous iteration, and then two below 'pasted' side-by-side).
- The bottom line length of each iteration was $2(2^n) - 1$
- The maximum number of spaces pasted in between the lower two copies was $2^n - 1$, and the number of pasted spaces decreased by two for each line. So, for instance, to build the fourth iteration, I could:
    - Copy the third iteration
    - Below that, copy the *first line* of the third iteration, tack on $2^n - 1$ spaces, and then tack on that same first line again.
    - Below that, copy the *second line* of the third iteration, tack on $(2^n - 1) - 2$ spaces, and then tack on that same second line again
    - Below that, copy the *third line* of the third iteration, tack on $(2^n - 1) - 4$ spaces, and then tack on that same third line again, and so on.

After a little consideration, I decided that I was most comfortable storing the output of whatever function I was about to build in an array like the following:

```
var lines = ['L', 'L L', 'L   L', 'L L L L', 'L       L', ... ];
```

… and having the final step be something like:

```
return lines.join('\n');
```

… which would turn the whole thing into one long string that would display correctly in the console (or wherever). The pasting operation described above looked an awful lot like a recursive loop in which (for each element in the previous iteration's array) I would:

- Push it to the current iteration's array
- Concatenate it, some determinable number of spaces, and itself together, and push them to some separate storage array
- Add the contents of the second storage array back to the current iteration's array

So! To work. For the recursion to execute properly, I would need a base case:

```
function addLines(n) {
   if (n === 0) {
      return ['L'];
   }
}
```

There was a good chance I'd want to do something else (like push that 'L' to some other array, rather than returning it already contained), but it was a good starting point. Following my previous thoughts, I worked through n === 1 and n === 2 explicitly before attacking the general case:

n === 1: Take addLines(n === 0), or `['L']`
Pull out the first element and create a new string:
'element + (2^n - 1 white-space characters) + element' or `'L L'`
Add that string back to the array from addLines(n === 0)

n === 2: Take addLines(n === 1), or `['L', 'L L']`
Pull out the first element and create a new string:
'element + (2^n - 1 white-space characters) + element' or `'L L'`
Add that string to a placeholder array (e.g. `appendix`)
Pull out the second element and create a new string:
'element + (2^n - 3 white-space characters) + element' or `'L L L L'`
Add that string to the placeholder array
Add the contents of the placeholder array back to the array from addLines(n === 1)

When I began to type out the general case, I quickly realized that there was no need for a placeholder array—I could simply take the array from the previous iteration and, as I worked through it, keep pushing new entries to it. I'd need to set my *for* loop to terminate based on that array's starting length, rather than simply using array.length as a variable (since that would just keep extending forever), but otherwise, the process was fairly straightforward:

```
function addLines(n) {
   if (n === 0) {
      return ['L'];
   } else {
      var prev = addLines(n - 1);
      var limit = Math.pow(2, n);
      // for each entry (each 'line') in the previous array
      for (var i = 0; i < limit; i++) {
         // make a string equal to prev[i] + <spaces> + prev[i]
      }
      prev.push(that string);
      return prev;
   }
}
```

Now all that remained was to figure out a concise method for concatenating the correct number of spaces into each line between the two copies of prev[i]. I sketched out the following table showing how the number of spaces needed changed as i incremented upward:

n	i	# of spaces
1	0	1
2	0	3
	1	1
3	0	7
	1	5
	2	3
	3	1

For the first row in each for loop (when i = 0), the number of spaces needed was equal to $2^n - 1$. As i incremented by 1, the number of spaces decremented by 2 ($2^n - 1$ where i = 0, then $2^n - 3$ where i = 1, then $2^n - 5$ where i = 2, etc.) yielding the general formula spaces = $2^n - (2i + 1)$. To actually create the string with the right number of spaces and then stick it in between two copies of the current line, I needed a block of code like the following:

```
var howManySpaces = limit - (2*i + 1);
var spaces = ‘’; // an empty string
while (howManySpaces > 0) {
  spaces += ‘ ’;
  howManySpaces--;
var line = prev[i] + spaces + prev[i];
```

… making the first draft of the complete addLines function:

```
function addLines(n) {
   if (n === 0) {
      return [‘L’];
   } else {
      var prev = addLines(n - 1);
      var limit = Math.pow(2, n);
```

```
      for (var i = 0; i < limit; i++) {                    7
         var howManySpaces = limit - (2*i + 1);            8
         var spaces = '';                                  9
         while (howManySpaces > 0) {                       10
           spaces += '-';                                  11
           howManySpaces--;                                12
         }                                                 13
         var line = prev[i] + spaces + prev[i];            14
      }                                                    15
      prev.push(line);                                     16
      return prev;                                         17
   }                                                       18
}                                                          19
```

Note that this isn't a complete Sierpinski function—this is intended to add lines to an array, but that array will have to be flattened into a string for the final desired result. I also chose to use dashes instead of actual spaces, for the sake of readability/countability, but switching that back at the end is just a matter of remembering to change a single character on line 11.

In attempting to run this version of the code, I immediately ran into an error. Calling addLines(0) returned `['L']`, just as one would hope and expect, but calling addLines(1) returned `['L', 'undefined-undefined']`. A quick scan-through of the function revealed that the statement `prev.push(line);` on line 16 was *outside* of the for-loop in which the variable 'line' was defined. Swapping it with line 15, so that it executed inside the for-loop, fixed the 'undefined' problem, but revealed a new hole in my logic:

```
> addLines(1)
> ['L', 'L-L', 'L-L-L-L']

> addLines(2)
> ['L', 'L-L', 'L-L-L-L', 'L---L', 'L-L----L-L', 'L-L-L-L----L-L-L-L',
'L---L----L---L']
```

Manually concatenated, as in the problem statement, those were:

1: L
L•L
L•L•L•L

2:

```
L
L•L
L•L•L•L
L•••L
L•L••••L•L
L•L•L•L••••L•L•L•L
L•••L••••L•••L
```

… which, while interesting, were certainly not the regular Sierpinski triangles I was hoping for.

So what went wrong? I dove back into the code and began working through addLines(1) manually, noting each action that the computer was taking and following all the rabbit holes. The workflow was something like this:

```
addLines(1)
→ Is the argument exactly equal to zero?  NO
→ Okay, then we're in the 'else.'
  → var prev = addLines(n - 1)
             = addLines(1 - 1)
             = addLines(0)
               → Is the argument exactly equal to zero?  YES
             = ['L'];
  → var limit = Math.pow(2, n)
              = Math.pow(2, 1)
              = 2^1
              = 2;
  → FOR LOOP
    → var i = 0; i < limit, i++
          i = 0; i < 2, i++  THIS LOOP WILL RUN TWICE (i = 0, i = 1)
    → i = 0
      → var spaces = '';
      → var howManySpaces = limit - (2*i + 1)
                          = 2 - (2*0 + 1)
                          = 2 - 1
                          = 1;
      → WHILE LOOP
        → Is howManySpaces greater than zero?  YES
        → Okay, then we'll execute the loop code
          → spaces += '-'
                   = spaces + '-'
                   = '' + '-'
```

```
            = '-';
    → howManySpaces--
            = howManySpaces - 1
            = 1 - 1
            = 0
    → Is howManySpaces greater than zero? NO (exit while loop)
→ var line = prev[i] + spaces + prev[i]
           = prev[0] + '-' + prev[0]
           = 'L' + '-' + 'L'
           = 'L-L';
→ prev.push(line);
  → Add line to the end of the array prev
  → prev = ['L', 'L-L']; (end of loop for i = 0)
```

… and right there, I could see the error. At that point, the variable prev contained an array with two elements (`['L', 'L-L']`), which is what I wanted returned for addLines(1). But the program was going to run the for loop *again*, this time with i equal to 1, resulting in an additional, erroneous line being added to the array before it was returned.

Remember that, initially, the 'limit' variable was created so that our for loop would not base its number of iterations on the length of the array that it, itself, was extending. In other words, if the for loop had read `(var i = 0; i < prev.length; i++)`, the function would have run into infinity as each iteration *changed* the value of prev.length, keeping it always one step ahead of i.

My knee-jerk reaction was just to change limit to `Math.pow(2, n) - 1`, but I wanted to think through the ramifications a bit before assuming that was the correct answer. When n = 1, the goal was to extend the array from one element (`['L']`) to two (`['L', 'L-L']`), meaning that the for loop should run one time, adding one new element. When n = 2, the goal was to extend the array from two elements to *four*, meaning that the for loop should run twice, adding two new elements. Continuing in this vein gave me the following table:

n	total elements in array	iterations needed/difference in number of elements from previous iteration
0	1	-
1	2	1
2	4	2
3	8	4

… and in fact I was right to mistrust my initial impulse. In each case, the for loop needs to run, not 2^n - 1 times, but 2^{n-1} times, which is quite a different story. A quick change to the definition of 'limit' (plus the shifting of line 16 as previously noted) left me with the following:

```
function addLines(n) {                                          1
   if (n === 0) {                                               2
      return ['L'];                                             3
   } else {                                                     4
      var prev = addLines(n - 1);                               5
      var limit = Math.pow(2, (n - 1));                         6
      for (var i = 0; i < limit; i++) {                         7
         var howManySpaces = Math.pow(2, n) - (2*i + 1);        8
         var spaces = '';                                       9
         while (howManySpaces > 0) {                            10
           spaces += '-';                                       11
           howManySpaces--;                                     12
         }                                                      13
         var line = prev[i] + spaces + prev[i];                 14
         prev.push(line);                                       15
      }                                                         16
      return prev;                                              17
   }                                                            18
}                                                               19
```

...notice that, on line 8, where I needed 2^n - (2i + 1), I could no longer use the variable 'limit' as shorthand for 2^n. A few quick function calls confirmed that I was now back on track:

```
> addLines(1)
> ['L', 'L-L']

> addLines(2)
> ['L', 'L-L', 'L---L', 'L-L----L-L']

> addLines(3)
> ['L', 'L-L', 'L---L', 'L-L----L-L', 'L-------L',
   'L-L----------L-L', 'L---L--------------L---L',
   'L-L----L-L----------------L-L----L-L']
```

However, another bug had appeared. Notice the fourth element in the array returned from addLines(2). It's `'L-L----L-L'` when it should be `'L-L-L-L'`. My algorithm should have placed a single dash in between two copies of the string `'L-L'`, but instead it placed four.

My first hypothesis was that the variable 'spaces' had not been properly reset. In the previous iteration of the for loop, the function had correctly placed three dashes between two copies of the string `'L'`; if it had then gone on to concatenate a single, additional dash onto those three dashes (instead of onto a new empty string), that would explain the issue.

A quick inspection of the for loop demonstrated that the bug was even more trivial than that. As it turned out, the function written above was perfect; it simply wasn't the function I had actually typed into my console for testing. I had accidentally shifted line 9 (the declaration and assignment of 'spaces') up to line 7, where it was outside the for loop and thus, as suspected, not being reset on each iteration. I corrected the typo, and the results were as hoped-for.

All that remained was to take the results of the addLines function and concatenate them with newline characters (and to switch the dashes I'd been using out for actual whitespace characters). I inserted a few more lines of code:

```
function sierpinski(n) {                                          1
  var lines = addLines(n);                                        2
  return lines.join('\n');                                        3
  function addLines(n) {                                          4
   if (n === 0) {                                                 5
      return ['L'];                                               6
   } else {                                                       7
      var prev = addLines(n - 1);                                 8
      var limit = Math.pow(2, (n - 1));                           9
      for (var i = 0; i < limit; i++) {                           10
         var howManySpaces = Math.pow(2, n) - (2*i + 1);          11
         var spaces = '';                                         12
         while (howManySpaces > 0) {                              13
           spaces += ' ';                                         14
           howManySpaces--;                                       15
         }                                                        16
         var line = prev[i] + spaces + prev[i];                   17
         prev.push(line);                                         18
      }                                                           19
      return prev;                                                20
   }                                                              21
  }                                                               22
}                                                                 23
```

… and voilà! Calling the function sierpinski with arguments 0, 1, 2, 3, and 4 produced exactly the characters shown on the first page, in the problem statement.

The final step was to compare my solution with others on CodeWars, to see if there were any lessons to be learned from more experienced programmers. As usual, there were several shorter solutions that were more compact and implemented methods such as .map, .concat, and .forEach to great advantage. Many of the solutions were dense and difficult to follow, however, raising the question of whether it is better to be clever and efficient, or accessible and understandable. All in all, I found that none of the alternate solutions seemed *strictly* better than my own (though I don't yet know enough about time and space efficiency to judge algorithmic superiority with any confidence). In terms of sheer cleverness and able-to-teach-me-a-lot, my favorite was the one at the top of the page (users borkess and CForero):

```
function sierpinski(n) {
  var m = Math.pow(2,n), s = ['L'];

  for (var i = 1; i < m; i++){
    s.push('L');
    for (var j = 2, l = i * 2; j < l; j +=2 ){
      s[i] += ' ' + (s[i-1][j]===s[i-1][j-2]? ' ': 'L');
    }
    s[i] += ' L';
  }

  return s.join('\n');
}
```

Thanks for reading!