

Recursion I



Overview

```
1  /*
2      - Definition of recursion
3      - The call stack
4      - countdown example
5      - factorial example
6      - Recursion and arrays, strings
7      - Tips for approaching recursion problems
8  */
9
10
11
12
13
14
```



Definition

```
1  /* recursion occurs when a function calls itself!  */
2
3  /* recursion is an alternative to iteration (using a loop) */
4
5  /* in the real world, you may see recursion instead of iteration when a
6     recursive solution is:
7     - easier to reason about (recursion helps break big problems into
8       small chunks
9     - easier to read than an iterative solution
10    - won't negatively affect performance too much (recursion can be
11       a memory hog) */
12
13
14
```



The call stack

```
1  /* before we talk about recursion, we have to talk about the call stack */
2
3  /* JS is "single threaded" - can only run one function at a time */
4
5  /* the call stack is the structure JS uses to figure out which function
6     it should be running at any point in time */
7
8
9
10
11
12
13
14
```



The call stack

```
1  /* whenever we call a function, it's added to the top of the call stack */
2
3  /* JS will execute whatever function is on the top of the stack */
4
5  function first() {
6      console.log('I am first!');
7  }
8
9  function second() {
10     console.log('I am second!');
11 }
12
13 first();
14 second();
```

Callstack



I am first

The call stack

```
1  /* whenever we call a function, it's added to the top of the call stack */
2
3  /* JS will execute whatever function is on the top of the stack */
4
5  function first() {
6      console.log('I am first!');
7  }
8
9  function second() {
10     console.log('I am second!');
11 }
12
13 first();
14 second();
```

Callstack

first()



I am first

The call stack

```
1  /* whenever we call a function, it's added to the top of the call stack */
2
3  /* JS will execute whatever function is on the top of the stack */
4
5  function first() {
6      console.log('I am first!');
7  }
8
9  function second() {
10     console.log('I am second!');
11 }
12
13 first();
14 second();
```

Callstack



I am first
I am second

The call stack

```
1  /* whenever we call a function, it's added to the top of the call stack */
2
3  /* JS will execute whatever function is on the top of the stack */
4
5  function first() {
6      console.log('I am first!');
7  }
8
9  function second() {
10     console.log('I am second!');
11 }
12
13 first();
14 second();
```

Callstack

second()



I am first
I am second

The call stack

```
1  /* whenever we call a function, it's added to the top of the call stack */
2
3  /* JS will execute whatever function is on the top of the stack */
4
5  function first() {
6      console.log('I am first!');
7  }
8
9  function second() {
10     console.log('I am second!');
11 }
12
13 first();
14 second();
```

Callstack



The call stack

```
1  function first() {  
2      console.log('I am first!');  
3      second();  
4      console.log('First is finished');  
5  }  
6  
7  function second() {  
8      console.log('I am second!');  
9  }  
10  
11 first();  
12  
13  
14
```

Callstack



I am first

The call stack

```
1 function first() {
2   console.log('I am first!');
3   second();
4   console.log('First is finished');
5 }
6
7 function second() {
8   console.log('I am second!');
9 }
10
11 first();
12
13
14
```

Callstack

first()



I am first
I am second

The call stack

```
1 function first() {  
2   console.log('I am first!');  
3   second();  
4   console.log('First is finished');  
5 }  
6  
7 function second() {  
8   console.log('I am second!');  
9 }  
10  
11 first();  
12  
13  
14
```

Callstack

second()

first()



I am first
I am second
First is finished

The call stack

```
1 function first() {  
2   console.log('I am first!');  
3   second();  
4   console.log('First is finished'); // first "paused" while second ran  
5 }  
6  
7 function second() {  
8   console.log('I am second!');  
9 }  
10  
11 first();  
12  
13  
14
```

Callstack

first()



I am first
I am second
First is finished

The call stack

```
1 function first() {  
2   console.log('I am first!');  
3   second();  
4   console.log('First is finished'); // first "paused" while second ran  
5 }  
6  
7 function second() {  
8   console.log('I am second!');  
9 }  
10  
11 first();  
12  
13  
14
```

Callstack



5
4
3
2
1

example: countdown

```
1  /* write a function that counts down to 1 */
2
3  function countdown(num) {
4      for (let i = num; i >= 1; i--) {
5          console.log(i);
6      }
7  }
8
9  countdown(5);
10
11
12
13
14
```



example: countdown

```
1  /* let's refactor our solution, writing a function that takes a number and
2     and logs it out */
3
4  function countdown(num) {
5     console.log(num);
6  }
7
8  countdown(5);
9  countdown(4);
10 countdown(3);
11 countdown(2); // notice, no loops!
12 countdown(1); // how do the arguments change between calls?
13
14
```




example: countdown

```
1  /* every time we called countdown, we subtracted one from the previous
2     num */
3
4  /* instead of manually calling countdown over and over, why not have
5     countdown call itself, subtracting one from num each time? */
6
7  function countdown(num) {
8     console.log(num);
9     countdown(num - 1);
10 }
11
12 countdown(5);
13
14
```

Callstack



example: countdown

```
1  /* every time we called countdown, we subtracted one from the previous
2     num */
3
4  /* instead of manually calling countdown over and over, why not have
5     countdown call itself, subtracting one from num each time? */
6
7  function countdown(num) {
8     console.log(num);
9     countdown(num - 1);
10 }
11
12 countdown(5);
13
14
```

Callstack

countdown(5)



example: countdown

```
1  /* every time we called countdown, we subtracted one from the previous
2     num */
3
4  /* instead of manually calling countdown over and over, why not have
5     countdown call itself, subtracting one from num each time? */
6
7  function countdown(num) {
8     console.log(num);
9     countdown(num - 1);
10 }
11
12 countdown(5);
13
14
```

Callstack

countdown(4)

countdown(5)



5
4
3

example: countdown

```
1  /* every time we called countdown, we subtracted one from the previous
2     num */
3
4  /* instead of manually calling countdown over and over, why not have
5     countdown call itself, subtracting one from num each time? */
6
7  function countdown(num) {
8     console.log(num);
9     countdown(num - 1);
10 }
11
12 countdown(5);
13
14
```

Callstack

countdown(3)

countdown(4)

countdown(5)



5
4
3
2

example: countdown

```
1  /* every time we called countdown, we subtracted one from the previous
2     num */
3
4  /* instead of manually calling countdown over and over, why not have
5     countdown call itself, subtracting one from num each time? */
6
7  function countdown(num) {
8      console.log(num);
9      countdown(num - 1);
10 }
11
12 countdown(5);
13
14
```

Callstack

countdown(2)

countdown(3)

countdown(4)

countdown(5)



5
4
3
2
1

example: countdown

```
1  /* every time we called countdown, we subtracted one from the previous
2     num */
3
4  /* instead of manually calling countdown over and over, why not have
5     countdown call itself, subtracting one from num each time? */
6
7  function countdown(num) {
8     console.log(num);
9     countdown(num - 1);
10 }
11
12 countdown(5);
13
14
```

Callstack
countdown(1)
countdown(2)
countdown(3)
countdown(4)
countdown(5)



example: countdown

5
4
3
2
1
0

```
1  /* every time we called countdown, we subtracted one from the pr
2     num */
3
4  /* instead of manually calling countdown over and over, why not have
5     countdown call itself, subtracting one from num each time? */
6
7  function countdown(num) {
8      console.log(num);
9      countdown(num - 1);
10 }
11
12 countdown(5);
13
14
```

Callstack

countdown(0)

countdown(1)

countdown(2)

countdown(3)

countdown(4)

countdown(5)



example: countdown

5
4
3
2
1
0
-1

```
1  /* every time we called countdown, we subtracted one from the pr
2     num */
3
4  /* instead of manually calling countdown over and over, why not have
5     countdown call itself, subtracting one from num each time? */
6
7  function countdown(num) {
8     console.log(num);
9     countdown(num - 1);
10 }
11
12 countdown(5);
13
14
```

Callstack
countdown(-1)
countdown(0)
countdown(1)
countdown(2)
countdown(3)
countdown(4)
countdown(5)



example: countdown

5
4
3
2
1
0
-1
-2

```
1  /* every time we called countdown, we subtracted one from the pr
2     num */
3
4  /* instead of manually calling countdown over and over, why not have
5     countdown call itself, subtracting one from num each time? */
6
7  function countdown(num) {
8      console.log(num);
9      countdown(num - 1);
10 }
11
12 countdown(5);
13
14
```

Callstack
countdown(-2)
countdown(-1)
countdown(0)
countdown(1)
countdown(2)
countdown(3)
countdown(4)
countdown(5)



example: countdown

5
4
3
2
1
0
-1
-2
(and so on)

```
1  /* every time we called countdown, we subtracted one from the pr
2     num */
3
4  /* instead of manually calling countdown over and over, why not have
5     countdown call itself, subtracting one from num each time? */
6
7  function countdown(num) {
8     console.log(num);
9     countdown(num - 1);
10 }
11
12 countdown(5);
13
14
```

Callstack
(and so on)
countdown(-2)
countdown(-1)
countdown(0)
countdown(1)
countdown(2)
countdown(3)
countdown(4)
countdown(5)



example: countdown

```
1  /* every time we called countdown, we subtracted one from the pr
2     num */
3
4  /* instead of manually calling countdown over and over, why not
5     countdown call itself, subtracting one from num each time? */
6
7  function countdown(num) {
8     console.log(num);
9     countdown(num - 1);
10 }
11
12 countdown(5);
13
14
```

5
4
3
2
1
0

-1
-2
-3
-4
-5
-6
-7
-8
-9
-10
-11

RangeError:
Maximum call
stack size
exceeded



example: countdown

```
1  /* that started off so promisingly! */
2
3  /* because our function was instructed to call itself every time, the
4     function ends up calling itself forever until our computer runs out of
5     memory */
6
7  /* let's write in a stop condition so the function eventually stops
8     calling itself */
9
10
11
12
13
14
```



example: countdown

```
1 function countdown(num) {  
2   // here's our stop condition, commonly known as the 'base case'  
3   if (num < 1) {  
4     console.log('done!');  
5   }  
6   // here's our 'recursive case'  
7   else {  
8     console.log(num);  
9     countdown(num - 1);  
10  }  
11 }  
12  
13 countdown(3);  
14
```

Callstack

example: countdown

```
1 function countdown(num) {  
2   // here's our stop condition, commonly known as the 'base case'  
3   if (num < 1) {  
4     console.log('done!');  
5   }  
6   // here's our 'recursive case'  
7   else {  
8     console.log(num);  
9     countdown(num - 1);  
10  }  
11 }  
12  
13 countdown(3);  
14
```

Callstack

countdown(3)



example: countdown

```
1 function countdown(num) {  
2   // here's our stop condition, commonly known as the 'base case'  
3   if (num < 1) {  
4     console.log('done!');  
5   }  
6   // here's our 'recursive case'  
7   else {  
8     console.log(num);  
9     countdown(num - 1);  
10  }  
11 }  
12  
13 countdown(3);  
14
```

Callstack

countdown(2)

countdown(3)



3
2
1

example: countdown

```
1 function countdown(num) {  
2   // here's our stop condition, commonly known as the 'base case'  
3   if (num < 1) {  
4     console.log('done!');  
5   }  
6   // here's our 'recursive case'  
7   else {  
8     console.log(num);  
9     countdown(num - 1);  
10  }  
11 }  
12  
13 countdown(3);  
14
```

Callstack

countdown(1)

countdown(2)

countdown(3)



example: countdown

3
2
1
done!

```
1 function countdown(num) {  
2   // here's our stop condition, commonly known as the 'base case'  
3   if (num < 1) {  
4     console.log('done!');  
5   }  
6   // here's our 'recursive case'  
7   else {  
8     console.log(num);  
9     countdown(num - 1);  
10  }  
11 }  
12  
13 countdown(3);  
14
```

Callstack

countdown(0)

countdown(1)

countdown(2)

countdown(3)



example: countdown

3
2
1
done!

```
1 function countdown(num) {  
2   // here's our stop condition, commonly known as the 'base case'  
3   if (num < 1) {  
4     console.log('done!');  
5   }  
6   // here's our 'recursive case'  
7   else {  
8     console.log(num);  
9     countdown(num - 1);  
10  }  
11 }  
12  
13 countdown(3);  
14
```

Callstack

countdown(1)

countdown(2)

countdown(3)



3
2
1
done!

example: countdown

```
1 function countdown(num) {  
2   // here's our stop condition, commonly known as the 'base case'  
3   if (num < 1) {  
4     console.log('done!');  
5   }  
6   // here's our 'recursive case'  
7   else {  
8     console.log(num);  
9     countdown(num - 1);  
10  }  
11 }  
12  
13 countdown(3);  
14
```

Callstack

countdown(2)

countdown(3)



3
2
1
done!

example: countdown

```
1 function countdown(num) {  
2   // here's our stop condition, commonly known as the 'base case'  
3   if (num < 1) {  
4     console.log('done!');  
5   }  
6   // here's our 'recursive case'  
7   else {  
8     console.log(num);  
9     countdown(num - 1);  
10  }  
11 }  
12  
13 countdown(3);  
14
```

Callstack

countdown(3)



3
2
1
done!

example: countdown

```
1 function countdown(num) {  
2   // here's our stop condition, commonly known as the 'base case'  
3   if (num < 1) {  
4     console.log('done!');  
5   }  
6   // here's our 'recursive case'  
7   else {  
8     console.log(num);  
9     countdown(num - 1);  
10  }  
11 }  
12  
13 countdown(3);  
14
```

Callstack



example: countdown

```
1  /* two takeaways from countdown: */
2
3  /* 1. you need to define a base case! */
4
5  /* 2. your recursive case must change the input to the function so that
6      you will eventually trigger the base case! */
7
8
9
10
11
12
13
14
```



Returning from recursive calls

```
1  /* recursion becomes more complicated when the function must return a
2     value */
3
4  /* good practice is to start by defining a base case */
5
6  /* base cases are often occur when there is a simple input that expects a
7     simple output (e.g., the sum of a single number is that number) */
8
9  /* test that the base case works before working with the recursive
10     case! */
11
12
13
14
```



example: factorial

```
1  /* define a function, factorial, that take a number and returns the
2     factorial of that number */
3
4  /* as a reminder:
5     0! === 1
6     1! === 1
7     2! === 2 (2 * 1)
8     3! === 6 (3 * 2 * 1)
9     4! === 24 (4 * 3 * 2 * 1)
10    5! === 120 (5 * 4 * 3 * 2 * 1) */
11
12 /* what look like simple inputs/outputs we can use to build a base
13    case? */
14
```




example: factorial

```
1 function factorial(num) {  
2   // base case: num is 0 or 1  
3   if (num === 0 || num === 1) {  
4     return 1;  
5   }  
6 }  
7  
8 factorial(0);  
9 factorial(1);  
10  
11  
12  
13  
14
```



example: factorial

```
1  /* ok, base case is set, just need to remember that our recursive case
2     has bring num closer and closer to 1 or 0 so we eventually
3     hit our base case */
4
5  /* notice an interesting pattern!
6
7     0! === 1
8     1! === 1
9     2! === 2 (2 * factorial(1))
10    3! === 6 (3 * factorial(2))
11    4! === 24 (4 * factorial(3))
12    5! === 120 (5 * factorial(4)) */
13
14
```



example: factorial

```
1 function factorial(num) {
2   // base case: num is 0 or 1
3   if (num === 0 || num === 1) {
4     return 1;
5   }
6   // recursive case: num must get closer to 0 or 1
7   // TODO
8 }
9
10 /* it's best to write your recursive case using the simplest possible
11    input that will result in a recursive call */
12 let result = factorial(2);
13 console.log(result);
14
```



example: factorial

```
1 function factorial(num) {
2   // base case: num is 0 or 1
3   if (num === 0 || num === 1) {
4     return 1;
5   }
6   // recursive case: num must get closer to 0 or 1
7   // we know we have to call factorial again in the recursive case
8   // if num === 2, what do we get if we call factorial again with num - 1?
9   console.log(factorial(num - 1));
10 }
11
12 let result = factorial(2);
13 console.log(result);
14
```



example: factorial

```
1 function factorial(num) {
2   // base case: num is 0 or 1
3   if (num === 0 || num === 1) {
4     return 1;
5   }
6   // recursive case: num must get closer to 0 or 1
7   // from that pattern we noticed earlier, we know 2! === 2 * 1!
8   console.log(num * factorial(num - 1));
9 }
10
11 let result = factorial(2);
12 console.log(result);
13
14
```



example: factorial

```
1 function factorial(num) {
2   // base case: num is 0 or 1
3   if (num === 0 || num === 1) {
4     return 1;
5   }
6   // recursive case: num must get closer to 0 or 1
7   // just have to return the result now
8   let result = num * factorial(num - 1);
9   return result;
10 }
11
12 let result = factorial(2);
13 console.log(result);
14
```



example: factorial

```
1 function factorial(num) {  
2   // base case: num is 0 or 1  
3   if (num === 0 || num === 1) {  
4     return 1;  
5   }  
6   // recursive case: num must get closer to 0 or 1  
7   let result = num * factorial(num - 1);  
8   return result;  
9 }  
10  
11 let result = factorial(5);  
12 console.log(result);  
13  
14
```

call stack

return value



example: factorial

```
1 function factorial(num) {  
2   // base case: num is 0 or 1  
3   if (num === 0 || num === 1) {  
4     return 1;  
5   }  
6   // recursive case: num must get closer to 0 or 1  
7   let result = num * factorial(num - 1);  
8   return result;  
9 }  
10  
11 let result = factorial(5);  
12 console.log(result);  
13  
14
```

call stack

return value

factorial(5)

5 * factorial(4)



example: factorial

```
1 function factorial(num) {  
2   // base case: num is 0 or 1  
3   if (num === 0 || num === 1) {  
4     return 1;  
5   }  
6   // recursive case: num must get closer to 0 or 1  
7   let result = num * factorial(num - 1);  
8   return result;  
9 }  
10  
11 let result = factorial(5);  
12 console.log(result);  
13  
14
```

call stack	return value
factorial(4)	4 * factorial(3)
factorial(5)	5 * factorial(4)



example: factorial

```
1 function factorial(num) {
2   // base case: num is 0 or 1
3   if (num === 0 || num === 1) {
4     return 1;
5   }
6   // recursive case: num must get closer to 0 or 1
7   let result = num * factorial(num - 1);
8   return result;
9 }
10
11 let result = factorial(5);
12 console.log(result);
13
14
```

call stack

return value

factorial(3) 3 * factorial(2)

factorial(4) 4 * factorial(3)

factorial(5) 5 * factorial(4)



example: factorial

```
1 function factorial(num) {
2   // base case: num is 0 or 1
3   if (num === 0 || num === 1) {
4     return 1;
5   }
6   // recursive case: num must get closer to 0 or 1
7   let result = num * factorial(num - 1);
8   return result;
9 }
10
11 let result = factorial(5);
12 console.log(result);
13
14
```

call stack

return value

factorial(2) 2 * factorial(1)

factorial(3) 3 * factorial(2)

factorial(4) 4 * factorial(3)

factorial(5) 5 * factorial(4)



example: factorial

```
1 function factorial(num) {
2   // base case: num is 0 or 1
3   if (num === 0 || num === 1) {
4     return 1;
5   }
6   // recursive case: num must get closer to 0 or 1
7   let result = num * factorial(num - 1);
8   return result;
9 }
10
11 let result = factorial(5);
12 console.log(result);
13
14
```

call stack	return value
factorial(1)	=> 1
factorial(2)	2 * factorial(1)
factorial(3)	3 * factorial(2)
factorial(4)	4 * factorial(3)
factorial(5)	5 * factorial(4)



example: factorial

```
1 function factorial(num) {
2   // base case: num is 0 or 1
3   if (num === 0 || num === 1) {
4     return 1;
5   }
6   // recursive case: num must get closer to 0 or 1
7   let result = num * factorial(num - 1);
8   return result;
9 }
10
11 let result = factorial(5);
12 console.log(result);
13
14
```

call stack	return value
factorial(2)	2 * 1
factorial(3)	3 * factorial(2)
factorial(4)	4 * factorial(3)
factorial(5)	5 * factorial(4)



example: factorial

```
1 function factorial(num) {
2   // base case: num is 0 or 1
3   if (num === 0 || num === 1) {
4     return 1;
5   }
6   // recursive case: num must get closer to 0 or 1
7   let result = num * factorial(num - 1);
8   return result;
9 }
10
11 let result = factorial(5);
12 console.log(result);
13
14
```

call stack

return value

factorial(2)

=> 2

factorial(3)

3 * factorial(2)

factorial(4)

4 * factorial(3)

factorial(5)

5 * factorial(4)



example: factorial

```
1 function factorial(num) {
2   // base case: num is 0 or 1
3   if (num === 0 || num === 1) {
4     return 1;
5   }
6   // recursive case: num must get closer to 0 or 1
7   let result = num * factorial(num - 1);
8   return result;
9 }
10
11 let result = factorial(5);
12 console.log(result);
13
14
```

call stack

return value

factorial(3)

3 * 2

factorial(4)

4 * factorial(3)

factorial(5)

5 * factorial(4)



example: factorial

```
1 function factorial(num) {
2   // base case: num is 0 or 1
3   if (num === 0 || num === 1) {
4     return 1;
5   }
6   // recursive case: num must get closer to 0 or 1
7   let result = num * factorial(num - 1);
8   return result;
9 }
10
11 let result = factorial(5);
12 console.log(result);
13
14
```

call stack

return value

factorial(3)

=> 6

factorial(4)

4 * factorial(3)

factorial(5)

5 * factorial(4)



example: factorial

```
1 function factorial(num) {
2   // base case: num is 0 or 1
3   if (num === 0 || num === 1) {
4     return 1;
5   }
6   // recursive case: num must get closer to 0 or 1
7   let result = num * factorial(num - 1);
8   return result;
9 }
10
11 let result = factorial(5);
12 console.log(result);
13
14
```

call stack

return value

factorial(4)

4 * 6

factorial(5)

5 * factorial(4)



example: factorial

```
1 function factorial(num) {
2   // base case: num is 0 or 1
3   if (num === 0 || num === 1) {
4     return 1;
5   }
6   // recursive case: num must get closer to 0 or 1
7   let result = num * factorial(num - 1);
8   return result;
9 }
10
11 let result = factorial(5);
12 console.log(result);
13
14
```

call stack

return value

factorial(4)

=> 24

factorial(5)

5 * factorial(4)



example: factorial

```
1 function factorial(num) {  
2   // base case: num is 0 or 1  
3   if (num === 0 || num === 1) {  
4     return 1;  
5   }  
6   // recursive case: num must get closer to 0 or 1  
7   let result = num * factorial(num - 1);  
8   return result;  
9 }  
10  
11 let result = factorial(5);  
12 console.log(result);  
13  
14
```

call stack

return value

factorial(5)

5 * 24



example: factorial

```
1 function factorial(num) {  
2   // base case: num is 0 or 1  
3   if (num === 0 || num === 1) {  
4     return 1;  
5   }  
6   // recursive case: num must get closer to 0 or 1  
7   let result = num * factorial(num - 1);  
8   return result;  
9 }  
10  
11 let result = factorial(5);  
12 console.log(result);  
13  
14
```

call stack

return value

factorial(5)

=> 120



example: factorial

```
1 function factorial(num) {
2   // base case: num is 0 or 1
3   if (num === 0 || num === 1) {
4     return 1;
5   }
6   // recursive case: num must get closer to 0 or 1
7   let result = num * factorial(num - 1);
8   return result;
9 }
10
11 let result = factorial(5);
12 console.log(result);
13
14
```

call stack	return value



example: factorial

```
1  /* three takeaways from factorial: */
2
3  /* 1. define your base case first, using simple inputs/outputs */
4
5  /* 2. define your base case, and test it using the simplest possible
6      input that results in one recursive call to the base case */
7
8  /* 3. test your function against more-complex inputs */
9
10
11
12
13
14
```



recursion and iterables

```
1  /* you can use recursion with any data type in JS */
2
3  /* if you're asked to recurse through arrays or strings, the base case
4     often occurs when the iterable is empty or has a length of one */
5
6  /* imagine finding the sum of numbers in an array */
7
8  sumArray([4]); // if array.length === 1, the sum is easy to calculate
9
10 /* if the base case required the iterable to have a length of 1 or 0, it
11     must mean that the recursive case has to reduce the length of the
12     iterable with every recursive call */
13
14 /* note: nested arrays can be approached differently; see next unit! */
```



other recursion hints

```
1  /* cannot emphasize enough: start with the base case! */
2
3  /* cannot emphasize enough: test recursive case with simplest possible
4     input that will result in one recursive call to the base case */
5
6  /* ask yourself: what type of thing should my function return? base case
7     and recursive case should return the same type of thing! */
8
9  /* use console.logs or debugger to debug */
10
11
12
13
14
```




Recap

```
1  /*
2      - Definition of recursion
3      - The call stack
4      - countdown example
5      - factorial example
6      - Recursion and arrays, strings
7      - Tips for approaching recursion problems
8  */
9
10
11
12
13
14
```