Recursion

Springboard

Download Demo Code

Having a function call itself

Also: a very powerful programming technique

Also: a popular interview question topic

The Tiniest Review

Functions Calling Functions

```
function a() {
  console.log("hello");
  b();
  console.log("coding");
}

function b() {
  console.log("world");
  c();
  console.log("love");
}

function c() {
  console.log("i");
}
```

```
| → a()
| hello
|
| | → b()
| | world
| |
| | | → c()
| | | i
| | | ← undefined from c()
| |
| | love
| | ← undefined from b()
|
| coding
| ← undefined from a()
```

→ "hello world i love coding"

Remember, when you call a function, you "freeze" where you are until that function returns, and then continue where you left off.

So ***a*** prints `hello`, calls ***b*** which prints `world`, calls ***c*** which prints `i` and returns back to ***b*** which then prints `love` which then returns back to ***a*** which prints `coding`.

Loops and Recursion

Loops versus Recursion

Any loop can be written instead with recursion

Any recursion can be written instead with a loop

… but often, one way is easier for a problem

Count to 3

Using a ***while*** loop:

```
function count() {
  let n = 1;

  while (n <= 3) {
    console.log(n);
    n += 1;
  }
}

count();
```

Using recursion:

```
function count(n=1) {
  if (n > 3) return;

  console.log(n);
  count(n + 1);
}

count();
```

Call Frames / Stack

```
function count(n=1) {
  if (n > 3) return;

  console.log(n);
  count(n + 1);
}

count();
```

```
| → count(n=1)
| 1
|
| | → count(n=2)
| | 2
| |
| | | → count(n=3)
| | | 3
| | |
| | | | → count(n=4)
| | | | ← undefined from count(n=4)
| | |
| | | ← undefined from count(n=3)
| |
| | ← undefined from count(n=2)
|
| ← undefined from count(n=1)
```

More Counting

```
function count(n=1) {
  if (n > 3) return;

  console.log(n);
  count(n + 1);
  console.log(n);
}

count();
```

```
| → count(n=1)
| 1
|
| | → count(n=2)
| | 2
| |
| | | → count(n=3)
| | | 3
| | |
| | | | → count(n=4)
| | | | ← undefined from count(n=4)
| | |
| | | 3
| | | ← undefined from count(n=3)
| |
| | 2
| | ← undefined from count(n=2)
|
| 1
| ← undefined from count(n=1)
```

Loops versus Recursion

Using a ***while*** loop:

```
function count() {
  let n = 1;

  while (n <= 3) {
    console.log(n);
    n += 1;
  }
}

count();
```

Using recursion:

```
function count(n=1) {
  if (n > 3) return;

  console.log(n);
  count(n + 1);
}

count();
```

Which do you prefer?

Requirements

Base Case

```
function count(n=1) {
  if (n > 3) return;

  console.log(n);
  count(n + 1);
}
```

- Every recursive function needs a ***base case***
  - How do we know when we're done?

Often a base case is a "degenerate case".

- concat([1, 2, 3]) →
- "1" + concat([2, 3]) →
- "1" + "2" + concat([3]) →
- "1" + "2" + "3" + concat([]) ← **degenerate: empty array**

Note: Degenerate Cases

A "degenerate case" is one that is so reduced that it's fundamentally different from the others and would need to be treated differently.

Consider counting up to 3 recursively:

```
function count(n=1) {
  if (n > 3) return;

  console.log(n);
  count(n + 1);
}
```

Here, our base case is "when we hit 3, don't keep recursing". This is a base case, but it's not "degenerate" — we *could* keep counting up after 3; there's nothing preventing us from doing so besides our goal to stop.

Compare this with finding the length of a list recursively:

```
function lenlist(nums) {
  if (nums[0] === undefined) return 0

  return 1 + lenlist(nums.slice(1));
}
```

Here, our base case is "the length of an empty list is 0, so return that and don't recurse". This base is "degenerate" — there's no possible way for us to find the length of a list with -1 items in it! It wouldn't even be possible for us to keep recursing; this base case is a hard limit on what's possible.

Not all recursive problems have a degenerate base case, but thinking about if one is possible is often helpful in figuring what your base case is and how the recursion should work.

No Base Case

```
function count(n=1) {
  console.log(n);
  count(n + 1);
}

count();
```

Stack Overflow!

Explicit vs. Hidden Base Cases

```
function count(n=1) {
  if (n > 3) return;

  console.log(n);
  count(n + 1);
}
```

```
function count(n=1) {
  if (n <= 3) {
    console.log(n);
    count(n + 1);
  }
}
```

Which do you prefer?

Progress

```
function count(n=1) {
  if (n > 3) return;

  console.log(n);
  count(n + 1);
}
```

Returning Data

Finding Sum of List

"Return sum of list using recursion"

- What's our base case?
  - An empty list has sum = 0!

```
function sum(nums) {
  if (nums.length === 0) return 0;

  return nums[0] + sum(nums.slice(1));
}

sum([1, 2, 4, 5]);
```

List Doubler

The Problem

"For every number in array, print the value, doubled"

```
data = [ 1, 2, 3 ]  //  => 2 4 6
```

```
function doubler(nums) {
  for (let n of nums) {
    console.log(n * 2);
  }
}
```

The Challenge

- Some items can be lists themselves
- We want to "flatten" them and still print doubled

```
data = [ 1, [2, 3], 4 ]  //   => 2 4 6 8
```

```
function doubler(nums) {
  for (let n of nums) {
    if Array.isArray(n) {
      for (let o of n) console.log(o * 2);
    } else {
      console.log(n * 2);
    }
  }
}
```

Oh No!

Some of *those* items can be lists!

```
data = [ 1, [2, [3], 4], 5 ]  //   => 2 4 6 8 10
```

```
function doubler(nums) {
  for (let n of nums) {
    if Array.isArray(n) {
      for (let o of nums) {
        if Array.isArray(o) {
          for (let p of o) console.log(p * 2);
        } else {
          console.log(o);
        }
      }
    } else {
      console.log(n * 2);
    }
  }
}
```

Arbitrary Depth with Loop

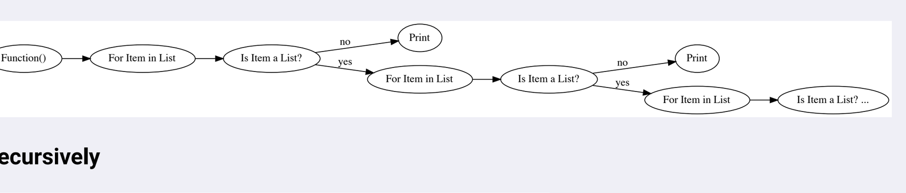
```
data = [ 1, [2, [3], 4], 5 ]  //   => 2 4 6 8 10

function doubler(nums) {
  stack = nums.reverse();

  while (stack.length > 0) {
    let n = stack.pop();
    if Array.isArray(n) {
      // If array, add it to stack, reversed
      for (let inner of n.reverse() {
        stack.append(inner);
      }
    } else {
      console.log(n * 2);
    }
  }
}
```

It works, but it's pretty hairy!

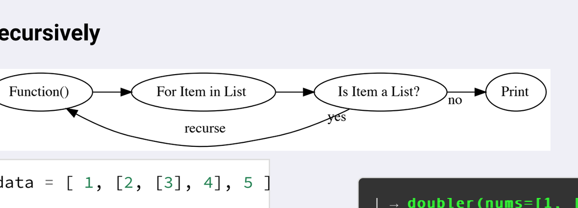
This solution uses a data structure called a "stack", adding new work to the end and popping them off the end.

This code may be worth study, even though this problem is more easily solved with recursion.

Non-Recursively



Recursively

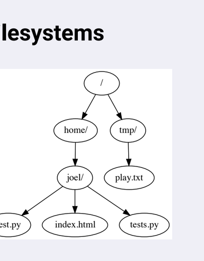



```
data = [ 1, [2, [3], 4], 5 ]

function doubler(nums) {
  for (let n of nums) {
    if Array.isArray(n) {
      doubler(n);
    } else {
      console.log(n * 2);
    }
  }
}
```

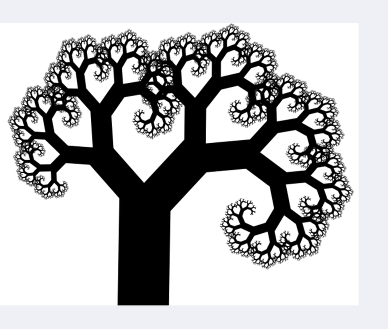
```
| → doubler(nums=[1, [2, [3], 4], 5])
| 2
|
| | → doubler(nums=[2, [3], 4])
| | 4
| |
| | | → doubler(nums=[3])
| | | 6
| | |
| | | ← undefined from doubler(nums=[3])
| |
| | 8
| | ← undefined from doubler(nums=[2, [3], 4])
|
| 10
| ← undefined from
|      doubler(nums=[1, [2, [3], 4], 5])
```

Recognizing Recursion

Filesystems



Fractals



Parsing

1 × (2 + 3 × (4 + 5 × 6) + 7)

This is a particularly good, hard exercise to give yourself.

Nested Data

```
<html>
  <head>
    <title>Title</title>
  </head>
  <body>
    <h1>Body</h1>
    <ul>
      <li>One</li>
      <li>Two
        <ul>
          <li>Two A</li>
          <li>Two B</li>
        </ul>
      </li>
    </ul>
  </body>
</html>
```

Runtime

What's the runtime?

```
function sum(nums) {
  if (nums.length === 0) return 0;

  return nums[0] + sum(nums.slice(1));
}
```

O(n ^ 2) — we keep making new lists!

It also has ***O(n ^ 2)*** *runspace* — keeping all lists in memory!

Improving Runtime

Often, you can keep track of position in array, rather than slice:

```
function sum(nums, i=0) {
  if (i === nums.length) return 0;

  return nums[i] + sum(nums, i + 1);
}
```

Now runtime and runspace are ***O(n)***

Accumulating Output

> *Given array of numbers, return even numbers*

```
function evens(nums, i=0) {
  if (nums.length === i) return [];

  if (nums[i] % 2 === 0) {
    return [nums[i], ...evens(nums, i +1)];
  }

  return evens(nums, i + 1);
}
```

Back to ***O(n ^ 2)*** — making all those lists!

Can solve with "helper recursion":

```
function evens(nums) {
  let out = [];

  function _evens(nums, i) {
    if (nums.length === i) return;
    if (nums[i] % 2 === 0) out.push(nums[i]);
    _evens(nums, i + 1);
  }

  _evens(nums, 0);
  return out;
}
```

Back to ***O(n)***

Accumulators

Often, can also solve with "accumulator":

```
function evens(nums, out=[], i=0) {
  if (nums.length === i) return out;

  if (nums[i] % 2 === 0) out.push(nums[i]);

  return evens(nums, out, i + 1);
}
```

Back to ***O(n)***

In some browsers, this can be "tail-call optimized"

Resources

How Recursion Works

Advanced: Tail Call Optimization