Grade 7/8 Math Circles
March 6/7/8/9, 2023
Recursion and Stack ADTs

What Is Recursion?

What is recursion? If you've heard of recursive sequences in math, you might think of a famous sequence called the Fibonacci Sequence. But what makes this sequence recursive? Let's take a look.

Example A

The Fibonacci sequence:

Place in Sequence	1	2	3	4	5	6	7	8	...
Value	1	1	2	3	5	8	13	21	...

What is the pattern here? We can see that the first two numbers are 1 and 1, but after that the value generated is the sum of the previous two numbers. For example, the third number in the sequence (2) is the sum of the two previous numbers, 1 and 1. The eighth number in the sequence (21) is the sum of the two previous numbers, 8 and 13.

So from this example, we can see that a recursive sequence is a sequence that generates the next value based on previous values.

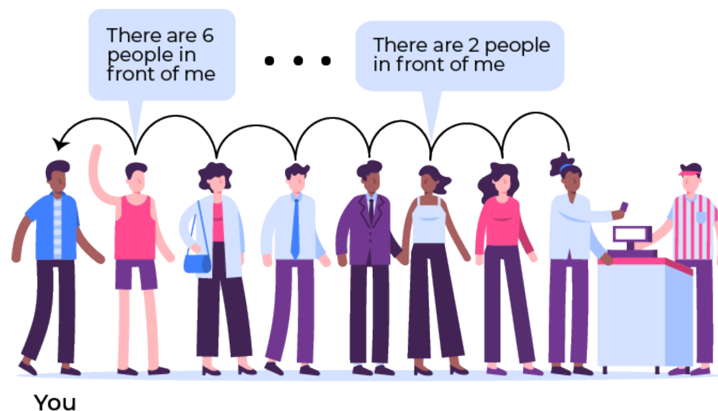
Let's look at an example that might help you think about recursion in general. (This example is taken from this YouTube video on recursion.) Suppose you're waiting in line to pay for your groceries. You just got there and wanted to know how long the line was. One way you could find out would be to go and count every person in the line. Or you could find out how many people are in a line through recursion.

Designed by pikisuperstar / Freepik

It works like this. First, you ask the person in front of you how many people are in front of them.

Then the person in front of you asks how many people are in front of them and so on. Each time someone asks the person in from of them how many people are ahead in line, we call this an iteration. This process would only stop until the second person in line asks the first person.

Then the first person would know that they are the first person in line, so they would tell the second person that there was no one in front of them. The second person would then tell the person behind them there was one person in front of them. This process would repeat until the person in front of you told you how many people were in front of them.



In this scenario, you would know that there are 7 people in front of you, because the person in front of you told you that there were 6 people in front of them. So we can summarize recursion as the idea of making the problem a smaller version of itself after each iteration.

Recursion In Computer Science

That was recursion in a real life scenario, but how about recursion in computer science? It is a topic that is heavily studied in the world of computer science, and to study it, we will have to use **pseudo-code** (simplified code). Let us define some computer science ideas that we will use.

Definitions

Variable: the name for an object. Just like how x and y are variables in math and can take on values, the same can happen for variables in computer science.
Function: a set of instructions that takes an input and produces an output. You can think of it almost like a function in math. When we use a function, we say that we *call* the function.
Argument: an input variable that a function can manipulate. A function can have no arguments, one argument, or more than one argument.
Return statement: a statement that determines the output of a function.
If statement: a decision statement that is based on whether something is true or false. Sometimes this is called a **conditional statement**.
Base case: the case where the answer is "obvious" and stops the recursion. In our line example, the base case was when the first person was asked how many people were in front of them.
Recursive case: the case where the answer requires another function call.

Example B

Functions, arguments, and return statements:

An example of a function with an argument and return statement is the following:

```
1 plus_one(number) {
2     return number + 1
3 }
```

The function called `plus_one` takes the argument, `number`, and returns the same number but one larger. The *function call* `plus_one(1)` gives us 2 as our value.

Notice that if we have an open bracket, we must also have an end bracket. Here, our open and close curly brackets on lines 1 and 3 show us that line 2 is the body of the function. In general, brackets are used to "group" different sections of code.

`if` statement:

An examples of `if` statements:

```
1  if(a=b) {
2      print("equal")
3  } else if(a>b) {
4      print("greater than")
5  } else {
6      print("less than")
7  }
```

```
1  if(a=b) {
2      print("equal")
3  }
4
5
6
7
```

The quotation marks show that the print statement gives us a word instead of just a number or another variable. Both sets of code have variables `a` and `b`. The code on the left compares their value before printing whether `a` is equal to `b` (on lines 1-2), greater than `b` (on lines 3-4), or less than `b` (on lines 5-6). The code on the right just compares `a` and `b`, prints whether they are equal or not, and does nothing otherwise.

Recursive functions, base cases, and recursive cases:

An example of a recursive function with a base case and recursive case:

```
1  two_power(exp) {
2      if(exp=0) {
3          return 1
4      } else {
5          return 2*two_power(exp-1)
6      }
7  }
```

Our function `two_power` takes a (non-negative) exponent, `exp`, and calculates the 2^{exp}. The base case (on lines 2-3) is that if our exponent is 0, then our value is automatically 1. The recursive case (on line 5) happens otherwise. Note that the recursive case always calls the function.

Let's trace our code to see how this happens. If we call the function with `two_power(3)` then we want to see whether we have our base case or whether we have to go into our recursive case. Since `exp=3`, the first iteration of `two_power` is not our base case and returns `two_power(3) => 2*two_power(3-1) => 2*two_power(2)`. This gives the following:

```
two_power(3) => 2*two_power(2)
```

So we go to our second iteration to find our answer. Since `exp=2` in our second iteration, this is also not our base case and returns `two_power(2) => 2*two_power(2-1) => 2*two_power(1)`. This gives the following:

```
two_power(3) => 2*two_power(2)
             => 2*2*two_power(1)
```

So we go to our third iteration to find our answer. Since `exp=1` in our third iteration, this is also not our base case and returns `two_power(1) => 2*two_power(1-1) => 2*two_power(0)`. This gives the following:

```
two_power(3) => 2*two_power(2)
             => 2*2*two_power(1)
             => 2*2*2*two_power(0)
```

So we go to our fourth iteration to find our answer. Since `exp=0` in our fourth iteration, this is finally our base case and returns `two_power(0) => 1`. This gives the following:

```
two_power(3) => 2*two_power(2)
             => 2*2*two_power(1)
             => 2*2*2*two_power(0)
             => 2*2*2*1
             => 8
```

This makes sense since $2^3 = 8$.

Some tips and tricks

If that was confusing for you, that's okay, because recursion is confusing for many other people too. That includes university students! Here are some tips to help you think about recursion:

1. Always think about your base case first. The base case is the scenario in your problem that is at its smallest point and can be solved easily. It should be almost trivial to solve a base case.
2. When thinking about your recursive case, you should always try to make your return value a smaller version of the bigger problem. You want to get closer to your base case at every step. Often you'll add/subtract a number or add/remove something from your variable to get closer to your base case.
3. Don't try to track your code through every step when it comes to big cases! For our example above, it would be pointless to go through `two_power(50)` step by step. It will only hurt your brain and subject yourself to human error.

Example C

Suppose you wanted to write a function called `reverse` that took a number called `num` and flipped all the digits. How would you do that?

First, let's think about our base case. What's the smallest case that we can be given? The "most obvious" case would be if the number is a single digit. This is the smallest case because no work is needed to flip a number that is just one digit; a single digit number flipped is just itself.

So we know that our function, which we call `reverse`, looks something like

```
1 reverse(num) {
2     if(len(num)=1) {
3         return num
4     } else {
5         recursive case
6     }
7 }
```

where `len(num)` gives the length of `num` and `num` is a positive number.

Then let's think about our recursive case. How do we make our problem smaller at every iteration? Given a number that has n digits (meaning an arbitrary number of digits), we would be able to make our problem smaller by reducing the number of digits in the number. So how would we do this? If we think about a number with two digits, we would flip this number by taking the first digit and putting it at the end. So applying this idea generally, we can write our function to be the following.

```
1 reverse(num) {
2     if(len(num)=1) {
3         return num
4     } else {
5         return reverse(rest(num)) & first_dig(num)
6     }
7 }
```

`rest(num)` removes the first digit of `num` and gives the rest of the digits of the numbers and `first_dig(num)` gives the first digit of the number. Here, we just use "&" to combine the digits. If we were to use "+", then we would be doing math addition.

Exercise 1

Write a recursive function called `fibonacci` that takes a number `n` and gives the nth term of the Fibonacci sequence. Keep in mind that the first two digits of the sequences are 1 and 1. If

you need a reminder of how the Fibonacci sequence works, refer back to Example A.

Note: n^{th} means counting an arbitrary number, like $1^{st}, 2^{nd}, 3^{rd}, 4^{th}, ..., n^{th}$.

Solution

Since the first two digits of the sequence are 1 and 1, we actually have two base cases. If `n=1`, then we return 1, and if `n=2`, then we also return 1. So this means our function would look something like this.

```
1 fibonacci(n) {
2     if(n=1) {
3         return 1
4     } else if(n=2) {
5         return 1
6     } else {
7         recursive case
8     }
9 }
```

```
1 fibonacci(n) {
2     if(n=1 or n=2) {
3         return 1
4     } else {
5         recursive case
6     }
7 }
8
9
```

Note that these two ways of writing the function are the same, since in both base cases the return value is 1. The second way is a little more efficient to write since it uses less code. This would not work if the return values were different for the base cases.

Next, we look at our recursive case. Since our n^{th} Fibonacci number is calculated by adding the previous two Fibonacci numbers, we can deduce that our recursive case must be `return fibonacci(num-1) + fibonacci(num-2)`. So we get the following function

```
1 fibonacci(n) {
2     if (n=1 or n=2) {
3         return 1
4     } else {
5         return fibonacci(n-1) + fibonacci(n-2)
6     }
7 }
```

Abstract Data Types

Now we're going to look at something called an abstract data type. What is a data type? If you learned about computer science, you might tell me that something like an integer or a string is a data type. That's correct, but when we're looking at an abstract data type, we're looking more at a *data structure*. These data structures include stacks, arrays, queues, linked lists, trees, and many more! Don't worry if you don't know what these are.

But what makes these data structures abstract? You can think of abstract data types (or ADTs for short) as machines that have certain functions. For example, we can look at a TV and say that it at least has the functions to turn on/off, to turn up/down volume, and to change input. While we know that a TV can do these things, we might not know exactly how the remote control connects to the TV and how the circuits and lights work in it. So the TV is abstract to the user. ADTs work very similarly. They have functions that a user can use, but as the user we won't be able to see exactly how everything inside the ADT works. In this lesson, we will focus on Stack ADTs.

Stack ADTs

Stacks appear in the real world all the time. When do you see stacks? Maybe you see stacks of pancakes, stacks of paper, or even stacks of shipping containers. One of the most classic stack problems is the Tower of Hanoi. (You will see this in your problem set later.)

> **Stop and Think**
>
> What other real life scenarios can be modelled by a Stack ADT?
>
> You might be more familiar with a phone game advertisement for Sand Sort Puzzle. Believe it or not, that is a game about stacks! Some other examples of stacks are stacks of plates, stacks of chairs, stacks of trays, or even burgers. Anything that is stacked on top of each other can be modelled by a Stack ADT.

A Stack ADT is basically the same thing. Let's look its functions (operations):

- `top()`: returns what the item at the top is. Notice that this does not require any arguments.
- `is_empty()`: checks whether the stack is empty. It returns `true` if it is empty and `false`

otherwise.

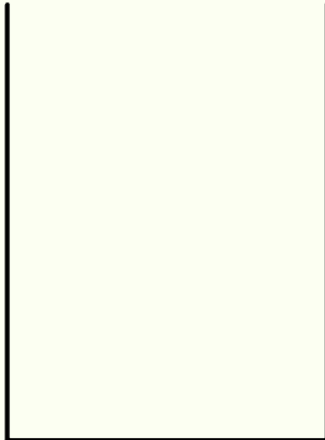
- `is_full()`: checks whether the stack is full. It returns `true` if it is full and `false` otherwise. Theoretically, we don't have to have this function since we can imagine a stack to be infinitely large. But practically, this is needed because computers can only hold so much information.
- `push(item)`: puts `item` at the top of the stack.
- `pop()`: takes off `item` from the top of the stack and returns it (if the stack has anything).

Items can be anything you want! To use a function (or operation), we code `stack.top()`, `stack.is_empty()`, `stack.is_full()`, `stack.push(item)`, or `stack.pop()`, where `stack` is the name of our stack.

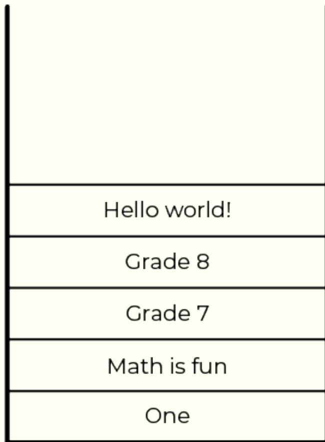
Note that our stack is abstract to us (the user) because we know how to interact with it using our five stack operations, but we don't exactly know how each operation works. We don't even know how the information is kept track of within the computer!

Exercise 2

Given the following pictures of the stack, fill out the table based on what would happen if you called the stack function from what the picture is showing.


empty stack

function call	return value	modifications
`stack.top()`		
`stack.is_empty()`		
`stack.is_full()`		
`stack.push(item)`		
`stack.pop()`		



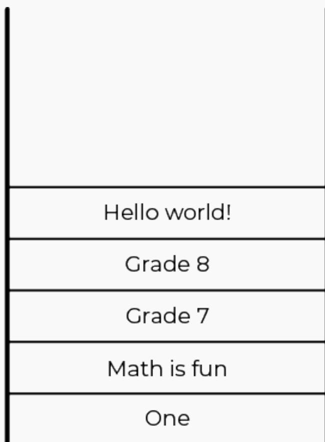
non-empty stack

function call	return value	modifications
`stack.top()`		
`stack.is_empty()`		
`stack.is_full()`		
`stack.push(item)`		
`stack.pop()`		

Solution

empty stack

function call	return value	modifications
`stack.top()`	none	none
`stack.is_empty()`	`true`	none
`stack.is_full()`	`false`	none
`stack.push(item)`	none	puts `item` at the top of the stack
`stack.pop()`	none	none



non-empty stack

function call	return value	modifications
`stack.top()`	`"Hello World!"`	none
`stack.is_empty()`	`false`	none
`stack.is_full()`	?	none
`stack.push(item)`	none	puts `item` at the top of the stack
`stack.pop()`	`"Hello World!"`	removes `"Hello World!"` from the top of the stack

Note that:

- `top()`, `is_empty`, and `is_full` will never modify your stack.
- `push(item)` will always work.
- `pop()` can only return something if there is something on the stack.
- An item that is popped can be directly used to push onto a stack (ie. `stack.push(stack.pop())`).

Example D

Lettuce
Tomato
Patty
Bun
Pickle
Bun

Supposed you asked your younger sibling to make a burger for you. You asked that in between the buns, the order of the ingredients from bottom to top would be lettuce, tomato, pickle, and then patty. Your sibling misheard you and made the following burger for you, represented in this stack called `burger`.

1. What does `burger.top()` return? What does that tell you about the burger?
2. Use stack functions (operations) to fix your burger. Assume that when you `pop` an item, it goes onto a plate.

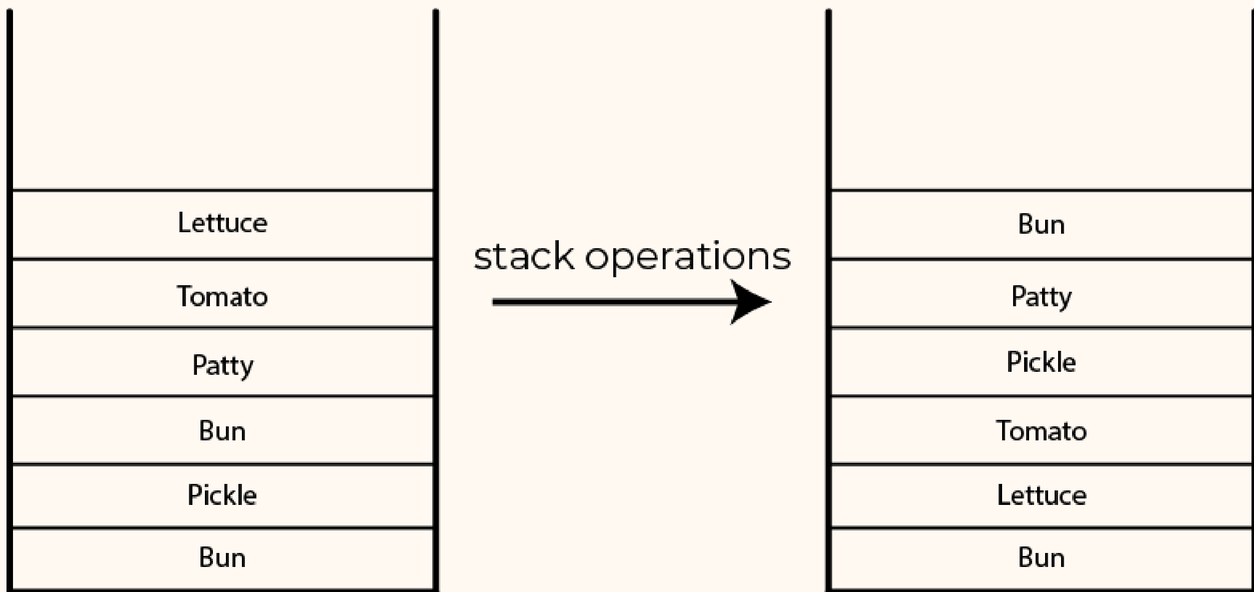
Solution

1. Calling `burger.top()` returns "Lettuce". Since the top of the burger is not a bun, you could reason just from looking at the top that this is not a burger (or maybe a lettuce burger at best).

2. We want our stack from bottom to top to be: bun, lettuce, tomato, pickle, patty, bun. Since we already have a bun on the bottom, we just have to remove everything above. This means that we would have to call `stack.pop()` 5 times before pushing the ingredients back on the stack in order. So we would have:

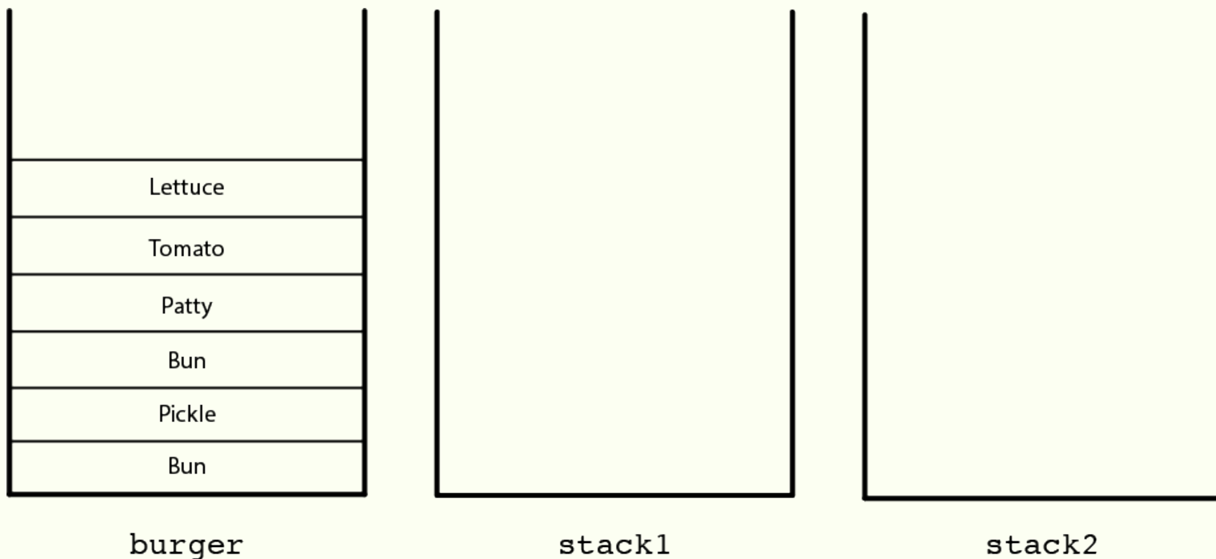
```
1  * Pop the ingredients on top of the bottom bun *
2  burger.pop()
3  burger.pop()
4  burger.pop()
5  burger.pop()
6  burger.pop()
7
8  * Push the ingredients back in order *
9  burger.push("Lettuce")
10 burger.push("Tomato")
11 burger.push("Pickle")
12 burger.push("Patty")
13 burger.push("Bun")
```

This gives the following before and after picture:



Exercise 3

Suppose you had the same burger as in Example E, but instead of popping an ingredient onto the plate, you popped the ingredient to another stack. In this setup, other than your `burger` stack, you have `stack1` and `stack2`. Use stack functions (operations) to fix your burger.



Solution

Similar to Example E, we need to pop every ingredient on top of the bottom bun. But what is different is that since we are popping ingredients onto stacks, when we rebuild our burger, we can only use the ingredient on top. This means that if we pop all our ingredients onto one stack, it will be less efficient when we need the ingredient that is at the bottom of the stack. Note that there are many ways to solve this problem, some being more efficient than others.

Let's start by popping off the ingredients from `burger` and pushing what we can onto `stack1`.

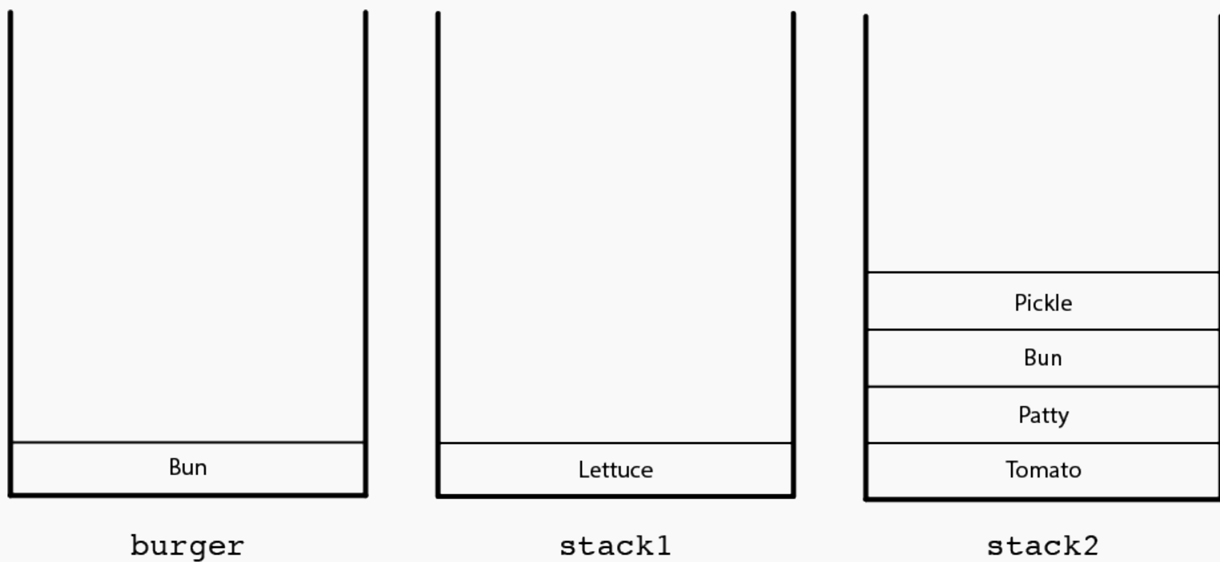
```
1 stack1.push(burger.pop())
```

This pops "Lettuce" off of `burger` and pushes it onto `stack1`. Since we want "Lettuce" to be on top of the last "Bun", we can reason that we should not push anything else onto `stack1` so that we can easily access "Lettuce" after we've popped everything we need to off

of burger. So we push everything else onto stack2 instead.

```
2  stack2.push(burger.pop())
3  stack2.push(burger.pop())
4  stack2.push(burger.pop())
5  stack2.push(burger.pop())
```

Now we get the following picture:



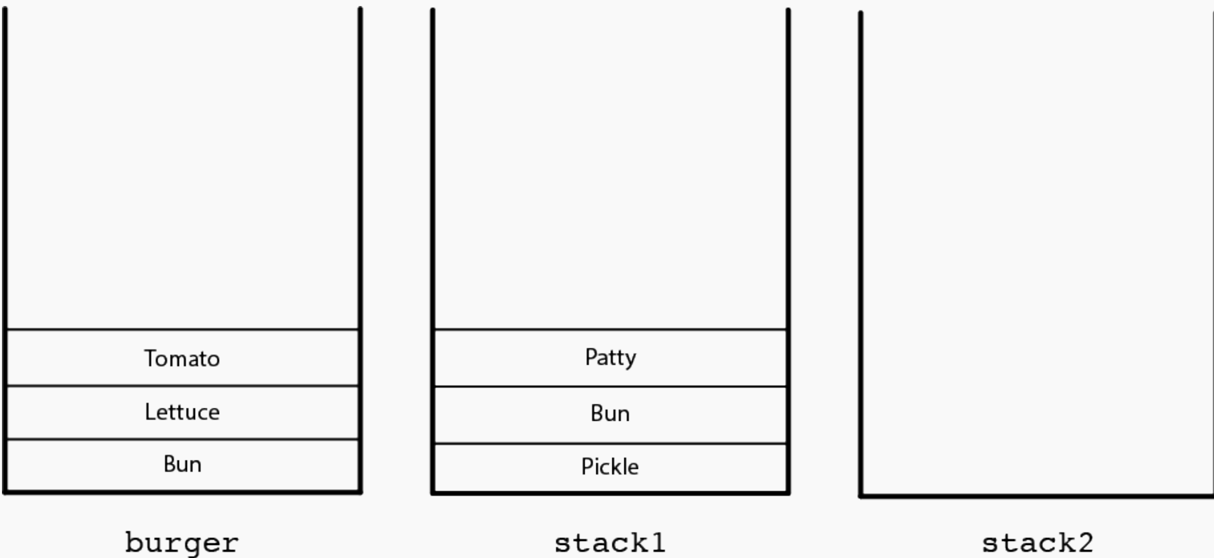
Now we start stacking the burger again. Since “Lettuce” is easily accessible, all we have to do is pop it off of stack1 and push it onto burger.

```
6  burger.push(stack1.pop())    * Put the lettuce onto burger *
```

Since we need “Tomato” to be pushed onto burger, we need to get to the bottom of stack2. But we also don’t want to push these other ingredients onto burger either, so we push it onto stack1 since it is currently empty. So we have the following code.

```
7  stack1.push(stack2.pop())
8  stack1.push(stack2.pop())
9  stack1.push(stack2.pop())
10 burger.push(stack2.pop())   * Put the tomato onto burger *
```

Now we have the following picture:



Now we finish off our burger with the following stack operations.

```
11 stack2.push(stack1.pop())
12 stack2.push(stack1.pop())
13 burger.push(stack1.pop())   * Put the pickle onto burger *
14 stack1.push(stack2.pop())
15 burger.push(stack2.pop())   * Put the patty onto the burger *
16 burger.push(stack1.pop())   * Put the bun onto the burger *
```

Combining Recursion and Stack ADTs

We've looked at two separate ideas in this lesson: recursion and Stack ADTs. But how can we put these two ideas together? Even looking at our burger example, there we saw a lot of repeated actions where our stack got smaller (or bigger). Well, what is recursion good for? It's good for doing repeated actions. Let's take a look at the following example.

Example E

What does this function (called `function`) do?

```
1 function(stack1, stack2) {
2     if(stack1.is_empty()) {
3         return "Done"
4     } else {
5         stack2.push(stack1.pop())
6         return function(stack1, stack2)
7     }
8 }
```

Solution

Let's use our recursion skills to analyze `function`.

Base case: Notice that the base case is to return `"Done"` if `stack1` is empty, since `stack1.is_empty()` only returns `true` if that is the case.

Recursive case: Notice that the recursive case is to first pop an item from `stack1` and push it onto `stack2`. This *modifies* both `stack1` and `stack2` before `function` recursively calls itself with the modified stacks. Note that the top item in `stack1` will become the bottom item in `stack2`, and the bottom item in `stack1` will become the top item in `stack2`.

So we can summarize `function` as a function that takes two stacks, `stack1` and `stack2`, and "flips" `stack1` onto `stack2`.

Example F

What does this function (called other_function) do?

```
1  other_function(stack, count) {
2      if(stack.is_full()) {
3          return "Done"
4      } else {
5          stack.push(count)
6          other_function(stack, count+1)
7      }
8  }
```

Solution

Let's use our recursion skills to analyze function.

Base case: Notice that the base case is to return "Done" if stack1 is full, since stack1.is_full() only returns true if that is the case.

Recursive case: Notice that the recursive case is to push count onto the stack and then to call other_function with an increased count by one.

So we can summarize other_function as a function that takes a stack and keeps on adding count to the stack increasingly until stack is full.

Exercise 4

Write a function called search that takes stack and item as arguments and searches for item within stack. If the item is found, then return true. If it cannot be found, then return "Not found".

Solution

First, we want to look at the structure of our function. Our function is called search and have our two arguments, stack and item.

```
1 search(stack, item) {
2     if(base case condition) {
3         base case
4     } else {
5         recursive case
6     }
7 }
```

Base case: Then we look at our base case, or in other words, when is it "obvious" that we have our answer? It's "obvious" when we either have `item` at the top of the `stack` or if the `stack` is empty. So we have our following code.

```
1 search(stack, item) {
2     if(stack.is_empty()) {
3         return "Not found"
4     } else if(stack.top()=item) {
5         return true
6     } else {
7         recursive case
8     }
9 }
```

Recursive case: Lastly, we look at our recursive case, or in other words, how can we make our problem smaller.

```
1  search(stack, item) {
2      if(stack.is_empty()) {
3          return "Not found"
4      } else if(stack.top()=item) {
5          return true
6      } else {
7          stack.pop()
8          return search(stack, item)
9      }
10 }
```

Note that since `stack.pop()` modifies the `stack`, our problem gets smaller every function call.