

# Recursion

15-110 - Friday 10/02

# Learning Objectives

- Define and recognize **base cases** and **recursive cases** in recursive code
- Read and write basic **recursive code**

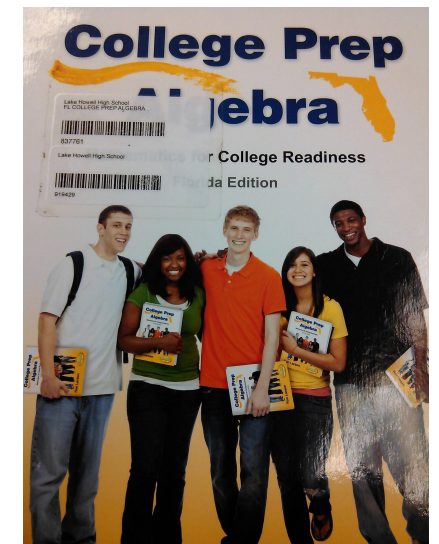
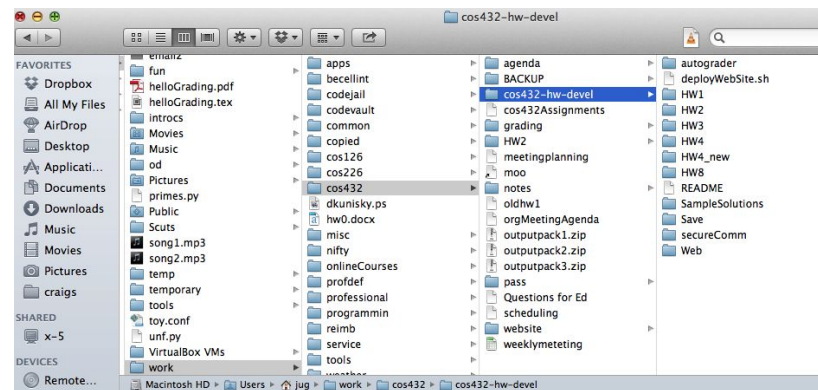
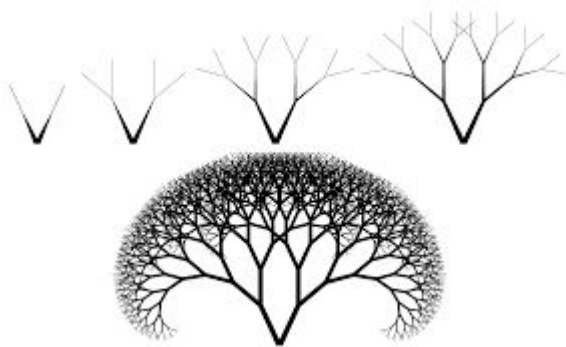
# Concept of Recursion

# Concept of Recursion

Recursion is a concept that shows up commonly in computing, and in the world.

Core idea: an idea  $X$  is recursive if  $X$  is used **in its own definition**.

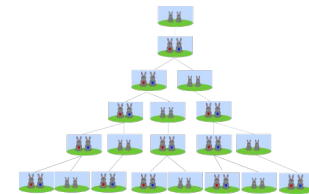
Example: fractals; nesting dolls; your computer's file system



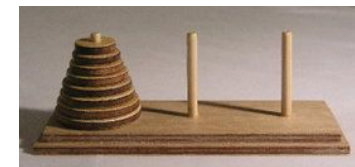
# Why Use Recursion?

Recursion is a hard concept to master, as it is different from how we typically approach problem-solving.

But recursion also makes it possible for us to solve some problems with simple, elegant algorithms. It also lets us think about how to structure data in new ways.



We'll start by using recursion to solve very simple problems, then show how it applies more naturally to complex problems next time.



# Recursion in Algorithms

When we use recursion in algorithms, it's generally used to implement **delegation** in problem solving, sometimes as an alternative to iteration.

To solve a problem recursively:

1. Find a way to make the problem **slightly smaller**
2. **Delegate** solving that problem to someone else
3. When you get the smaller-solution, **combine it** with the remaining part of the problem to get the answer

# Example: Iteration vs. Recursion

How do we add the numbers on a deck of cards?

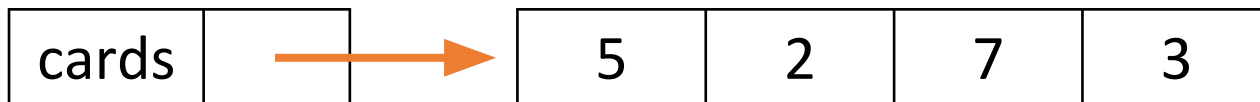
Iterative approach: keep track of the total so far, iterate over the cards, add each to the total.

Recursive approach: take a card off the deck, **delegate adding the rest of the deck to someone else**, then when they give you the answer, add the remaining card to their sum.

# Implementing Iteration

Let's look at how we'd add the deck of four cards using **iteration**.

**Pre-Loop:**

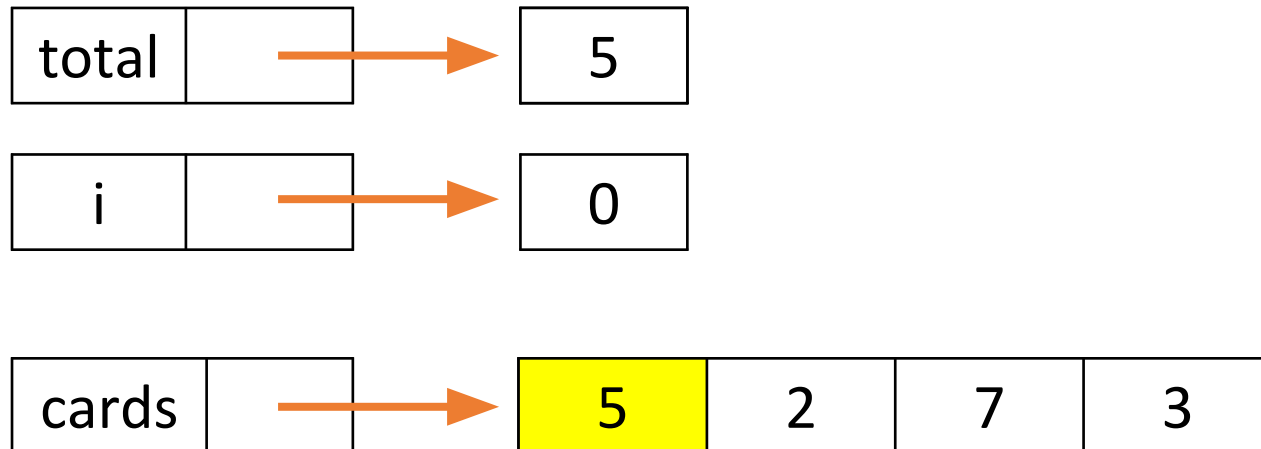




# Implementing Iteration

Let's look at how we'd add the deck of four cards using **iteration**.

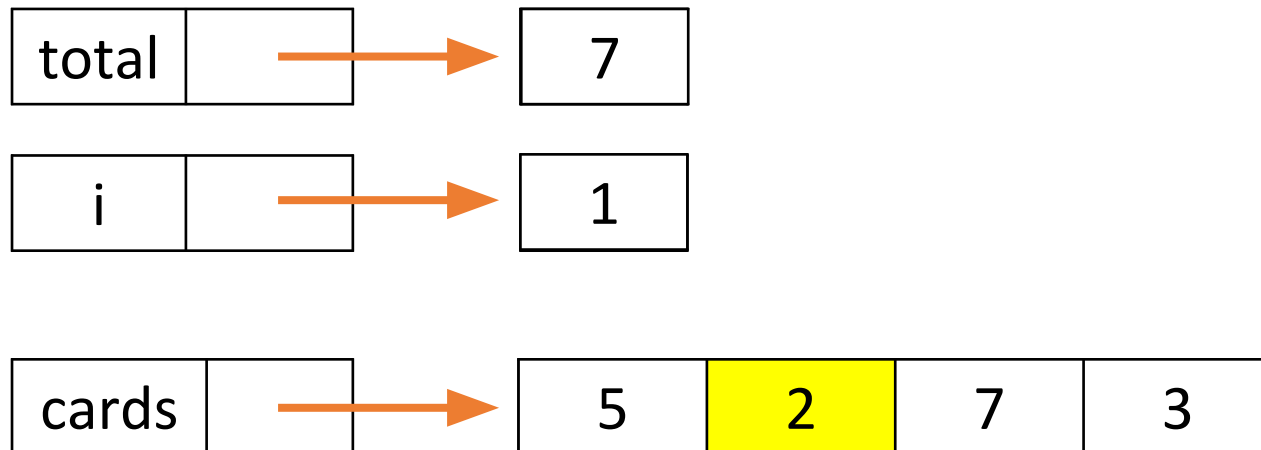
**First iteration:**



# Implementing Iteration

Let's look at how we'd add the deck of four cards using **iteration**.

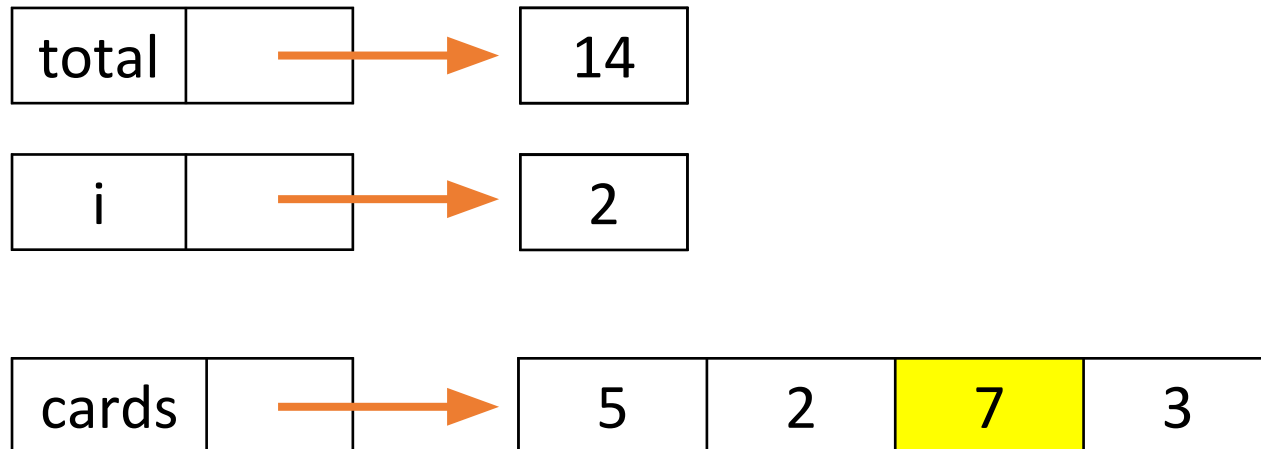
**Second iteration:**



# Implementing Iteration

Let's look at how we'd add the deck of four cards using **iteration**.

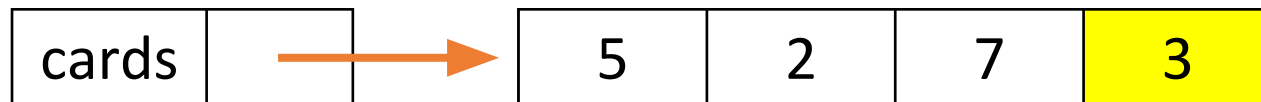
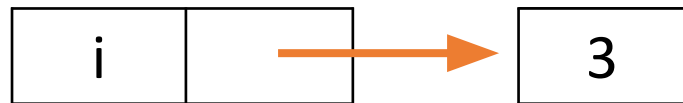
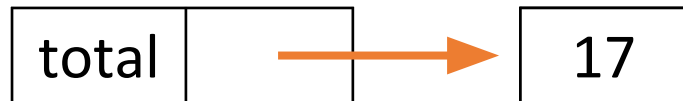
**Third iteration:**



# Implementing Iteration

Let's look at how we'd add the deck of four cards using **iteration**.

**Fourth iteration:**



And we're done!

# Iteration in Code

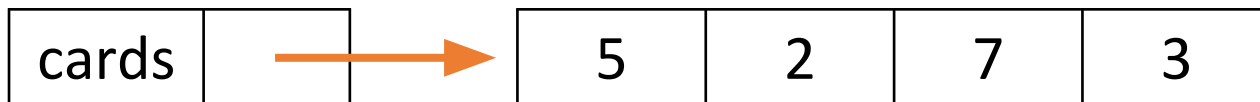
We could implement this in code with the following function:

```
def iterativeAddCards(cards):  
    total = 0  
    for card in cards:  
        total = total + card  
    return total
```

# Implementing Recursion

Now let's add the same deck of cards using **recursion**.

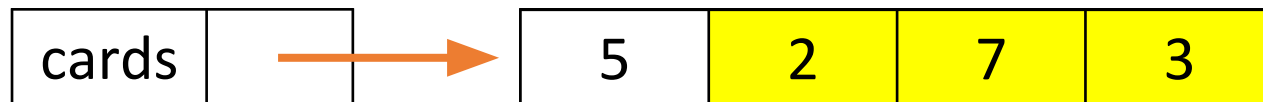
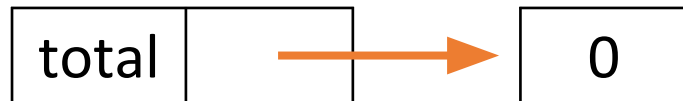
**Start State:**



# Implementing Recursion

Now let's add the same deck of cards using **recursion**.

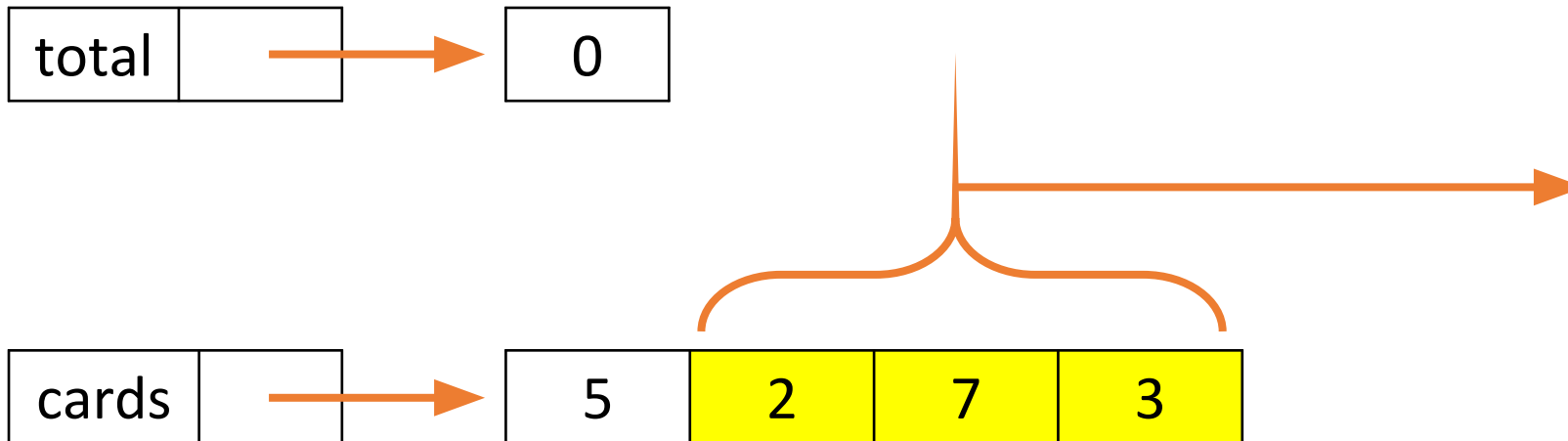
**Make the problem smaller:**



# Implementing Recursion

Now let's add the same deck of cards using **recursion**.

**Delegate that smaller problem:**



This is the Recursion Genie. They can solve problems, but only if the problem has been made slightly smaller than the start state.





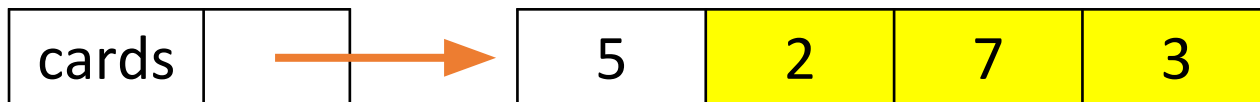
# Implementing Recursion

Now let's add the same deck of cards using **recursion**.

**Get the smaller problem's solution:**



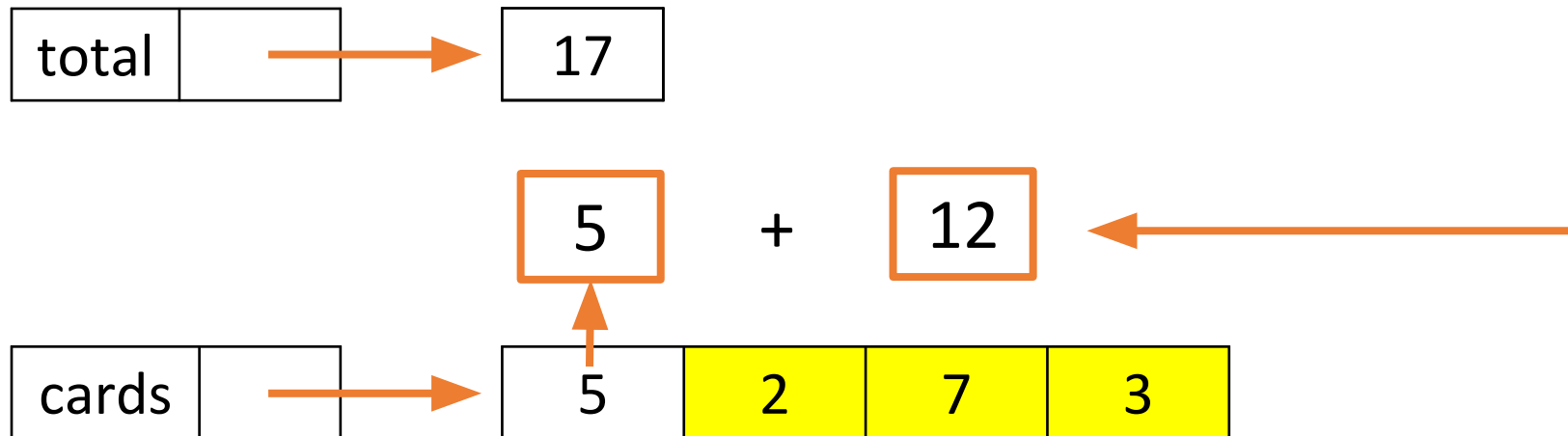
12



# Implementing Recursion

Now let's add the same deck of cards using **recursion**.

**Combine the leftover part with the smaller solution:**



And we're done!

# Recursion in Code

Now let's implement the recursive approach in code.

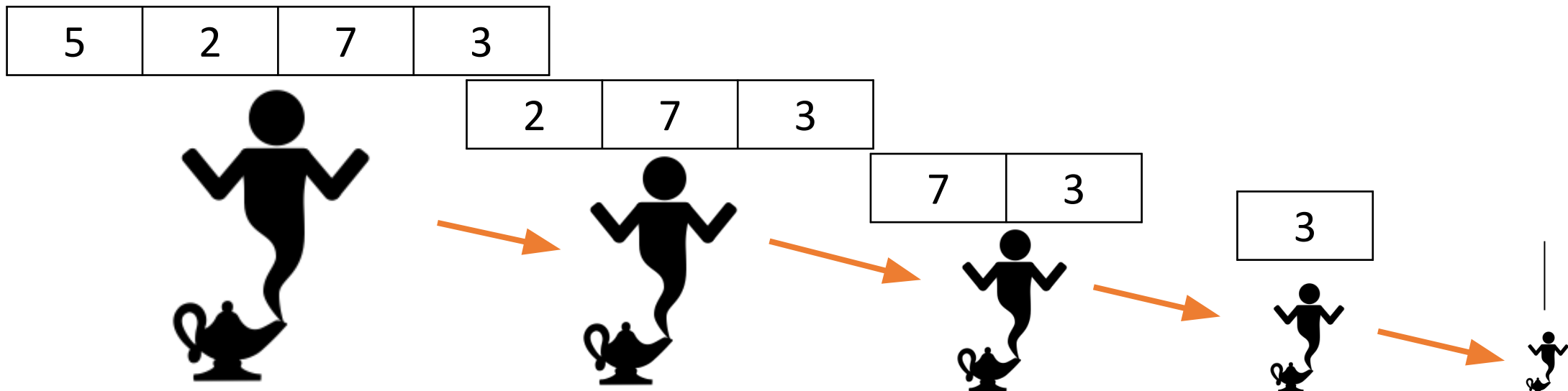
```
def recursiveAddCards(cards):  
    smallerProblem = cards[1:]  
    smallerResult = ??? # how to call the genie?  
    return cards[0] + smallerResult
```

# Base Cases and Recursive Cases

# Big Idea #1: The Genie is the Algorithm Again!

We don't need to make a new algorithm to implement the Recursion Genie. Instead, we can just **call the function itself** on the slightly-smaller problem.

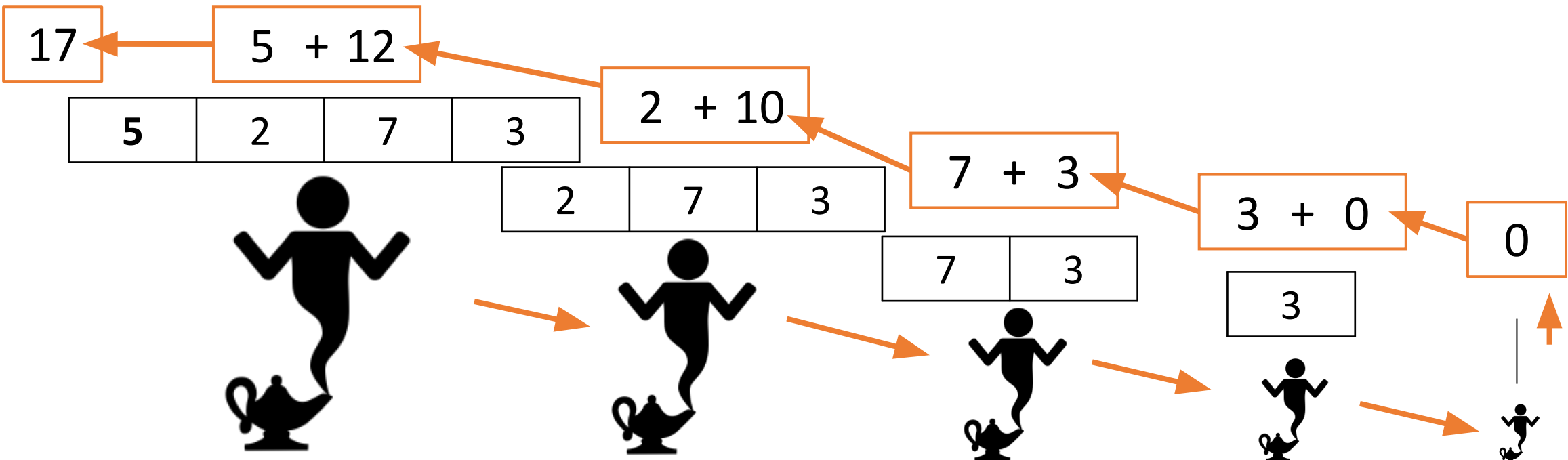
Every time the function is called, the problem gets smaller again. Eventually, the problem reaches a state where we can't make it smaller. We'll call that the **base case**.



# Big Idea #2: Base Case Builds the Answer

When the problem gets to the base case, the answer is immediately known. For example, in adding a deck of cards, the sum of an empty deck is 0.

That means the base case can solve the problem **without delegating**. Then it can pass the solution back to the prior problem and start the chain of solutions.



# Recursion in Code – Recursive Call

To update our recursion code, we first need to add the call to the function itself.

```
def recursiveAddCards(cards):  
    smallerProblem = cards[1:]  
    smallerResult = recursiveAddCards(smallerProblem)  
    return cards[0] + smallerResult
```

Incomplete solution!

# Recursion in Code – Base Case

We also need to add in the **base case**, as an explicit instruction about what to do when the problem cannot be made any smaller.

```
def recursiveAddCards(cards):  
    if cards == []:  
        return 0  
    else:  
        smallerProblem = cards[1:]  
        smallerResult = recursiveAddCards(smallerProblem)  
        return cards[0] + smallerResult
```



# Every Recursive Function Includes Two Parts

The two big ideas we just saw are used in **all** recursive algorithms.

**1. Base case(s) (non-recursive):**

One or more simple cases that can be solved directly (with no further work).

**2. Recursive case(s):**

One or more cases that require solving "simpler" version(s) of the original problem. By "simpler" we mean smaller/shorter/closer to the base case.

# Identifying Cases in addCards(cards)

Let's locate the base case and recursive case in our example.

```
def recursiveAddCards(cards):  
    if cards == []:  
        return 0  
    else:  
        smallerProblem = cards[1:]  
        smallerResult = recursiveAddCards(smallerProblem)  
        return cards[0] + smallerResult
```

**base case**

**recursive case**

# Activity: Recursive Binary Conversion

Let's look at a different example. Let's write an algorithm that takes a binary number as a string (like "10110") and converts it to decimal (21). Our algorithm will only use **recursion**; no loops allowed.

**You do:** in general terms, what is the base case for this problem? And in the recursive case, how do we make the problem smaller?

**Hint:** consider how you converted binary to decimal on Hw1.

# Python Tracks Recursion with the Call Stack

Recall back when we learned about functions and bytecode, how we used the **Call Stack** to keep track of nested function calls.

Python also uses the Call Stack to track recursive calls!

Because each function call has its own set of **local variables**, which includes function parameters, the values across functions don't get confused.

# Trace the Stack

recursiveAddCards([5, 2, 7, 3])

```
def recursiveAddCards(cards):  
    if cards == [ ]:  
        return 0  
    else:  
        return cards[0] + recursiveAddCards(cards[1:])
```

cards=[5,2,7,3] ; return 5 + recursiveAddCards([2,7,3])

Call Stack

# Trace the Stack

recursiveAddCards([5, 2, 7, 3])



recursiveAddCards([2, 7, 3])

```
def recursiveAddCards(cards):  
    if cards == [ ]:  
        return 0  
    else:  
        return cards[0] + recursiveAddCards(cards[1:])
```

cards=[2,7,3] ; return 2 + recursiveAddCards([7,3])

cards=[5,2,7,3] ; return 5 + recursiveAddCards([2,7,3])

Call Stack

# Trace the Stack

`recursiveAddCards([5, 2, 7, 3])`



`recursiveAddCards([2, 7, 3])`



`recursiveAddCards([7, 3])`

```
def recursiveAddCards(cards):  
    if cards == [ ]:  
        return 0  
    else:  
        return cards[0] + recursiveAddCards(cards[1:])
```

`cards=[7,3] ; return 7 + recursiveAddCards([3])`

`cards=[2,7,3] ; return 2 + recursiveAddCards([7,3])`

`cards=[5,2,7,3] ; return 5 + recursiveAddCards([2,7,3])`

Call Stack

# Trace the Stack

recursiveAddCards([5, 2, 7, 3])



recursiveAddCards([2, 7, 3])



recursiveAddCards([7, 3])



recursiveAddCards([3])

```
def recursiveAddCards(cards):  
    if cards == [ ]:  
        return 0  
    else:  
        return cards[0] + recursiveAddCards(cards[1:])
```

cards=[3] ; return 3 + recursiveAddCards([ ])

cards=[7,3] ; return 7 + recursiveAddCards([3])

cards=[2,7,3] ; return 2 + recursiveAddCards([7,3])

cards=[5,2,7,3] ; return 5 + recursiveAddCards([2,7,3])

Call Stack



# Trace the Stack

recursiveAddCards([5, 2, 7, 3])



recursiveAddCards([2, 7, 3])



recursiveAddCards([7, 3])



recursiveAddCards([3])



recursiveAddCards([])

```
def recursiveAddCards(cards):  
    if cards == []:  
        return 0  
    else:  
        return cards[0] + recursiveAddCards(cards[1:])
```

cards=[] ; return 0

cards=[3] ; return 3 + recursiveAddCards([])

cards=[7,3] ; return 7 + recursiveAddCards([3])

cards=[2,7,3] ; return 2 + recursiveAddCards([7,3])

cards=[5,2,7,3] ; return 5 + recursiveAddCards([2,7,3])

Call Stack

# Trace the Stack

`recursiveAddCards([5, 2, 7, 3])`



`recursiveAddCards([2, 7, 3])`



`recursiveAddCards([7, 3])`



`recursiveAddCards([3])`

```
def recursiveAddCards(cards):  
    if cards == []:  
        return 0  
    else:  
        return cards[0] + recursiveAddCards(cards[1:])
```

`cards=[3] ; return 3 + 0`

`cards=[7,3] ; return 7 + recursiveAddCards([3])`

`cards=[2,7,3] ; return 2 + recursiveAddCards([7,3])`

`cards=[5,2,7,3] ; return 5 + recursiveAddCards([2,7,3])`

Call Stack

# Trace the Stack

recursiveAddCards([5, 2, 7, 3])



recursiveAddCards([2, 7, 3])



recursiveAddCards([7, 3])

```
def recursiveAddCards(cards):  
    if cards == [ ]:  
        return 0  
    else:  
        return cards[0] + recursiveAddCards(cards[1:])
```

cards=[7,3] ; return 7 + 3

cards=[2,7,3] ; return 2 + recursiveAddCards([7,3])

cards=[5,2,7,3] ; return 5 + recursiveAddCards([2,7,3])

Call Stack

# Trace the Stack

recursiveAddCards([5, 2, 7, 3])



recursiveAddCards([2, 7, 3])

```
def recursiveAddCards(cards):  
    if cards == [ ]:  
        return 0  
    else:  
        return cards[0] + recursiveAddCards(cards[1:])
```

cards=[2,7,3] ; return 2 + 10

cards=[5,2,7,3] ; return 5 + recursiveAddCards([2,7,3])

Call Stack

# Trace the Stack

`recursiveAddCards([5, 2, 7, 3])`

```
def recursiveAddCards(cards):  
    if cards == [ ]:  
        return 0  
    else:  
        return cards[0] + recursiveAddCards(cards[1:])
```

`cards=[5,2,7,3] ; return 5 + 12`

Call Stack

# Trace the Stack

17

```
def recursiveAddCards(cards):  
    if cards == [ ]:  
        return 0  
    else:  
        return cards[0] + recursiveAddCards(cards[1:])
```

Call Stack

# Programming with Recursion

# Recipe for Writing Recursive Functions

Thinking of recursive algorithms can be tricky at first. Here's a recipe you can follow that might help.

1. Write an **if** statement  
**if** is the base case; **else** is the recursive case
2. Handle simplest case - the base case(s)  
No recursive call needed (that's why it is the **base case!**)
3. Write the recursive call  
Input to call must be slightly simpler/smaller to move towards the base case
4. **Be optimistic:** Assume the recursive call works!  
Ask yourself: What does it do?  
Ask yourself: How does it help?



# General Recursive Form

In fact, most of the simple recursive functions you write can take the following form:

```
def recursiveFunction(problem):  
    if problem == ???: # base case is the smallest value  
        return ____ # something that isn't recursive  
    else:  
        smallerProblem = ??? # make the problem smaller  
        smallerResult = recursiveFunction(smallerProblem)  
        return ____ # combine with the leftover part
```

# Important: Return Types Must Match!

When you write a recursive function, always remember that the base case must return the **same type** as the recursive case.

If the types are different, you'll have a problem combining the next step with the smaller-result, because the type of the smaller-result will be **inconsistent**.

Also make sure that you always provide the correct type in the **argument** given to the recursive function call. It must match the type of the function's parameter.

# Example: factorial

Assume we want to implement factorial recursively. Recall that:

$$x! = x * (x-1) * (x-2) * \dots * 2 * 1$$

We could rewrite that as...

$$x! = x * (x-1)!$$

What's the **base case**?

$$x == 1$$

What's the **smaller problem**?

$$(x - 1)!$$

How to **combine it**?

Multiply result of  $(x-1)!$  by  $x$

# Writing Factorial Recursively

We can take these algorithmic components and combine them with the general recursive form to get a solution.

```
def factorial(x):  
    if x == 1: # base case  
        return 1 # something not recursive  
    else:  
        smaller = factorial(x - 1) # recursive call  
        return x * smaller # combination
```

# Sidebar: Infinite Recursion Causes RecursionError

What happens if you call a function on an input that will never reach the base case? **It will keep calling the function forever!**

Example: `factorial(5.5)`

Python keeps track of how many function calls have been added to the stack. If it sees there are too many calls, it raises a `RecursionError` to stop your code from repeating forever.

If you encounter a `RecursionError`, check a) whether you're making the problem smaller each time, and b) whether the input you're using will ever reach the base case.

# Example: binToDec(x)

Let's go back to our earlier example, converting binary to decimal.

```
def binToDec(s):  
    if _____: # base case  
        return _____  
    else: # recursive case  
        result = binToDec(_____)  
        return _____
```

# Example: binToDec(x)

We make the problem smaller by chopping off the last digit.

```
def binToDec(s):
    if len(s) == 1: # base case
        return int(s)
    else: # recursive case
        lastDigit = s[len(s)-1]
        smaller = binToDec(s[:len(s)-1])
        # multiply result by 2, as it was in the 2s place
        return 2 * smaller + int(lastDigit)
```

# Example: countVowels(s)

Let's do one last example. Write the function `countVowels(s)` that takes a string and recursively counts the number of vowels in that string.

```
def countVowels(s):  
    if _____: # base case  
        return _____  
    else: # recursive case  
        smaller = countVowels(_____)  
        return _____
```



# Example: countVowels(s)

We make the string smaller by removing one letter. Change the code's behavior based on whether the letter is a vowel or not.

```
def countVowels(s):  
    if s == "": # base case  
        return 0  
    else: # recursive case  
        smaller = countVowels(s[1:])  
        if s[0] in "AEIOU":  
            return 1 + smaller  
        else:  
            return smaller
```

# Example: countVowels(s)

An alternative approach is to make **multiple recursive cases** based on the smaller part.

```
def countVowels(s):  
    if s == "": # base case  
        return 0  
    elif s[0] in "AEIOU": # recursive case  
        smaller = countVowels(s[1:])  
        return 1 + smaller  
    else:  
        smaller = countVowels(s[1:])  
        return smaller
```

## Activity: recursiveMatch(lst1, lst2)

**You do:** Write `recursiveMatch(lst1, lst2)`, which takes two lists of equal length and returns the number of indexes where `lst1` has the same value as `lst2`.

For example, `recursiveMatch([4, 2, 1, 6], [4, 3, 7, 6])` should return `2`.

# Learning Objectives

- Define and recognize **base cases** and **recursive cases** in recursive code
- Read and write basic **recursive code**
- **Feedback:** <https://bit.ly/110-feedback>