Advanced Programming #2: Recursion

CS SCHOLARS – PROGRAMMING

Content Note

This lecture will reference **indexing and slicing** with **lists and strings**, as it is hard to motivate recursion without data structures.

If you haven't yet learned about indexing and slicing, you can still attempt to follow along- there are some examples that just use numbers as well.

Learning Objectives

Define and recognize **base cases** and **recursive cases** in recursive code

Read and write basic **recursive code**

Trace over recursive functions that use **multiple recursive calls** with Towers of Hanoi

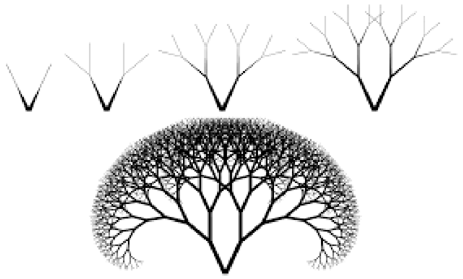
Concept of Recursion

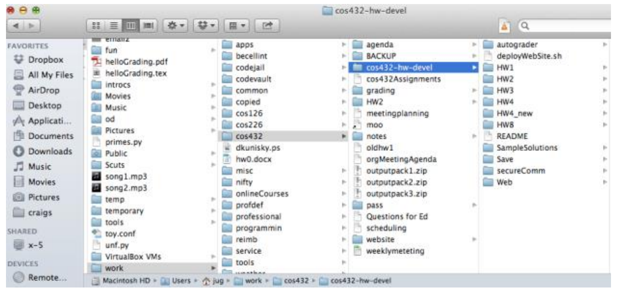
Concept of Recursion

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





cos432-hw-devel

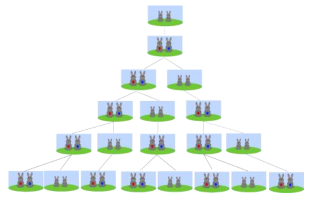
```
FAVORITES       fun                 apps            agenda               autograder
Dropbox         helloGrading.pdf    becellint       BACKUP               deployWebSite.sh
All My Files    helloGrading.tex    codejail        cos432-hw-devel      HW1
AirDrop         introcs             codevault       cos432Assignments    HW2
Desktop         Movies              common          grading              HW3
Applicati...    Music               copied          HW2                  HW4
Documents       od                  cos126          meetingplanning      HW4_new
Downloads       Pictures            cos226          moo                  HW8
Music           primes.py           cos432          notes                README
Movies          Public              dkunisky.ps     oldhw1               SampleSolutions
Pictures        Scuts               hw0.docx        orgMeetingAgenda     Save
craigs          song1.mp3           misc            outputpack1.zip      secureComm
SHARED          song2.mp3           nifty           outputpack2.zip      Web
x-5             temp                onlineCourses   outputpack3.zip
DEVICES         temporary           profdef         pass
Remote...       tools               professional    Questions for Ed
                toy.conf            programmin      scheduling
                unf.py              reimb           website
                VirtualBox VMs      service         weeklymeteting
                work                tools

Macintosh HD ▸ Users ▸ jug ▸ work ▸ cos432 ▸ cos432-hw-devel
```

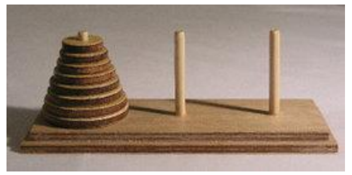
Why Use Recursion?

Recursion is a hard concept to master because 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.



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 solution to 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?

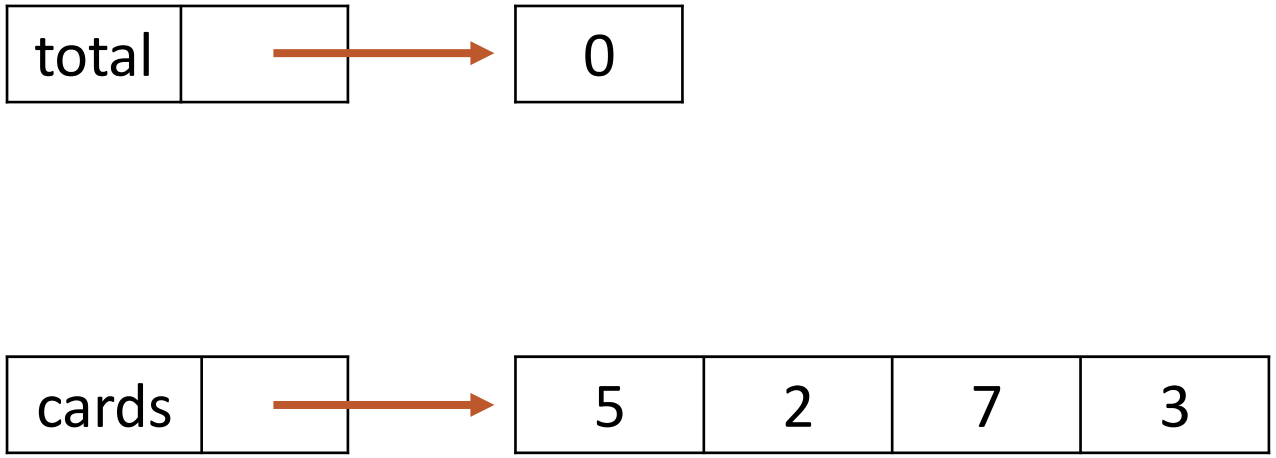
Iterative approach: keep track of the total so far, iterate over the cards, add each to the total.

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**.

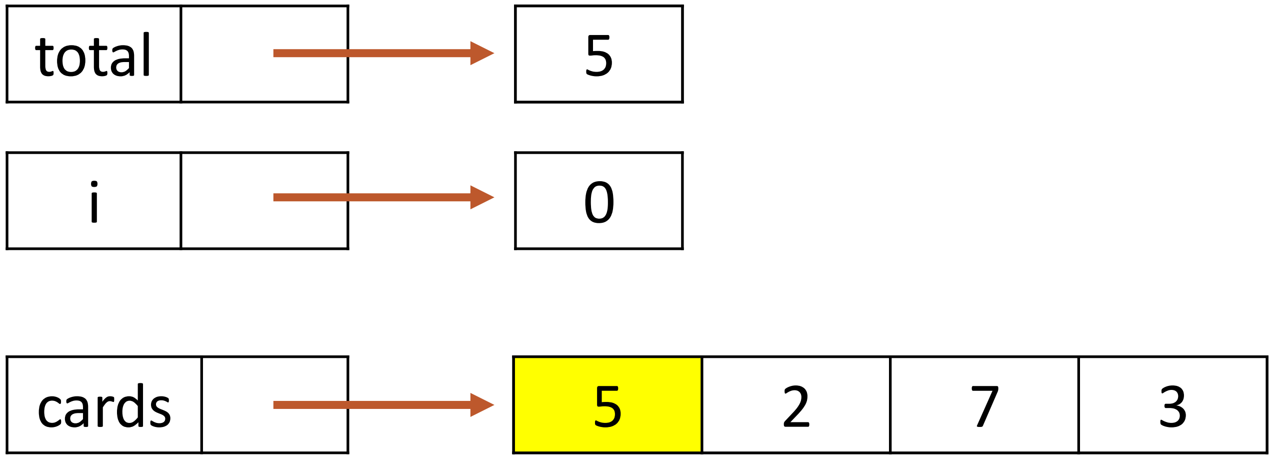
Pre-Loop:



Implementing Iteration

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

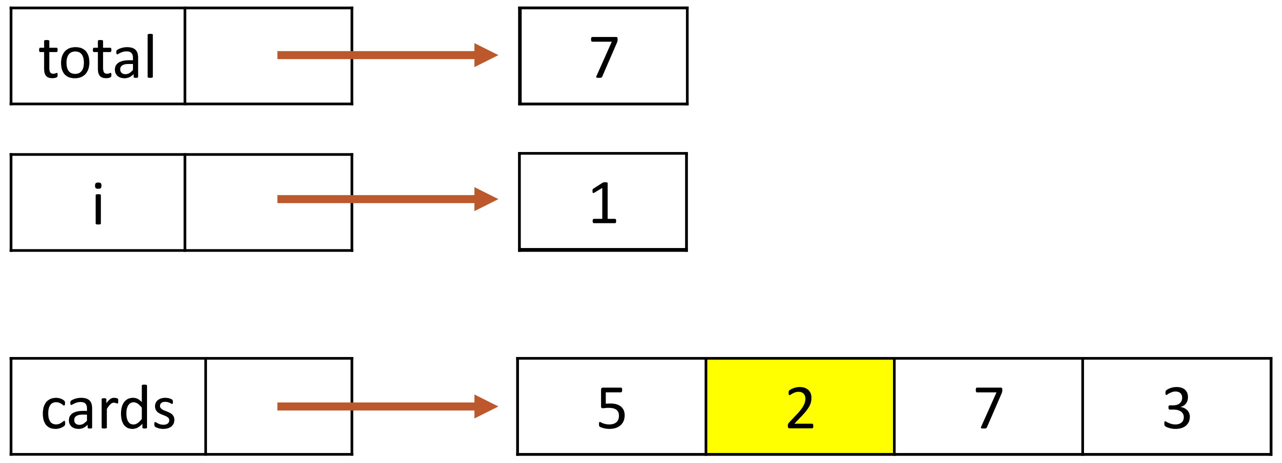
First iteration:



Implementing Iteration

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

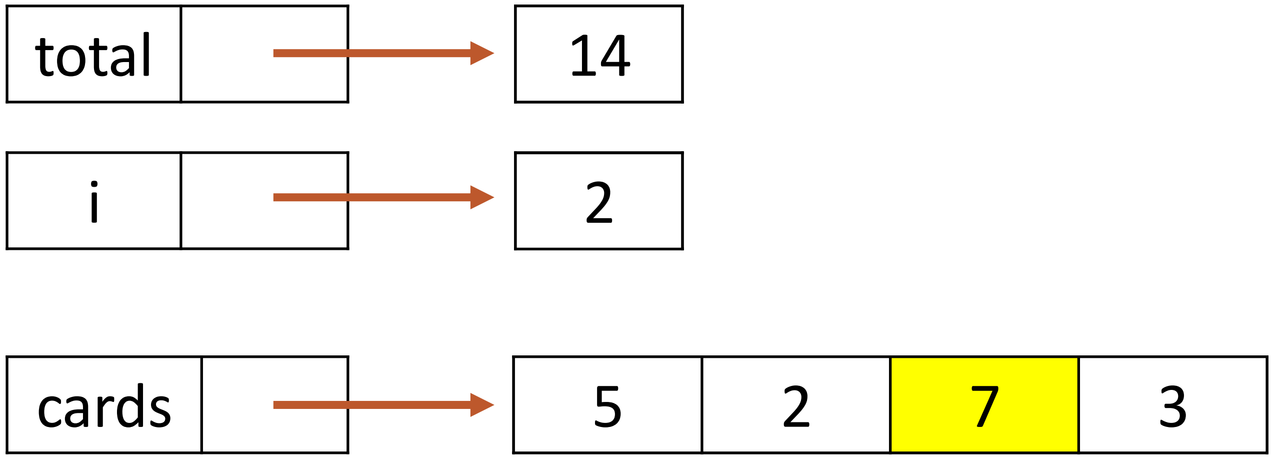
Second iteration:



Implementing Iteration

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

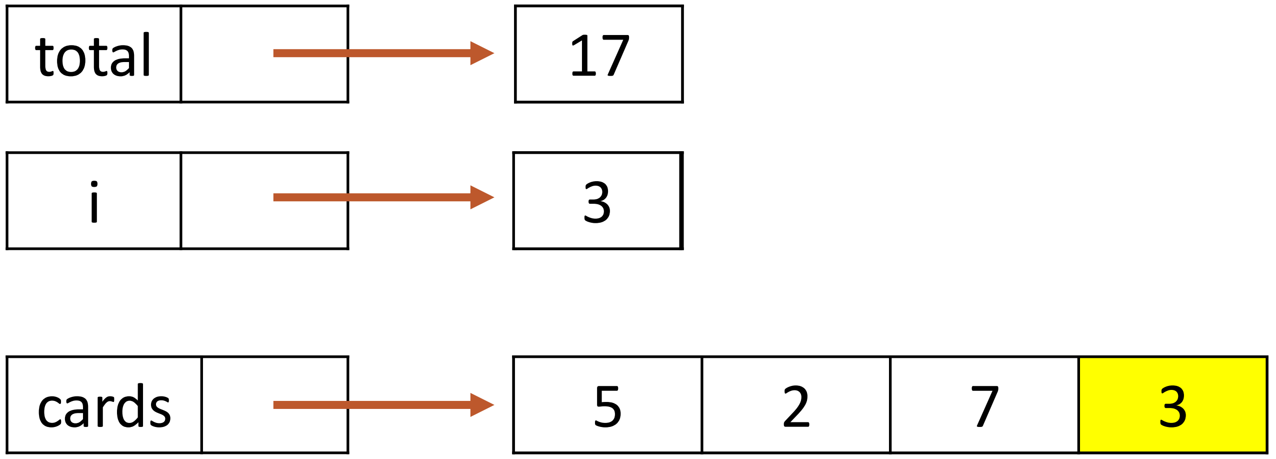
Third iteration:



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

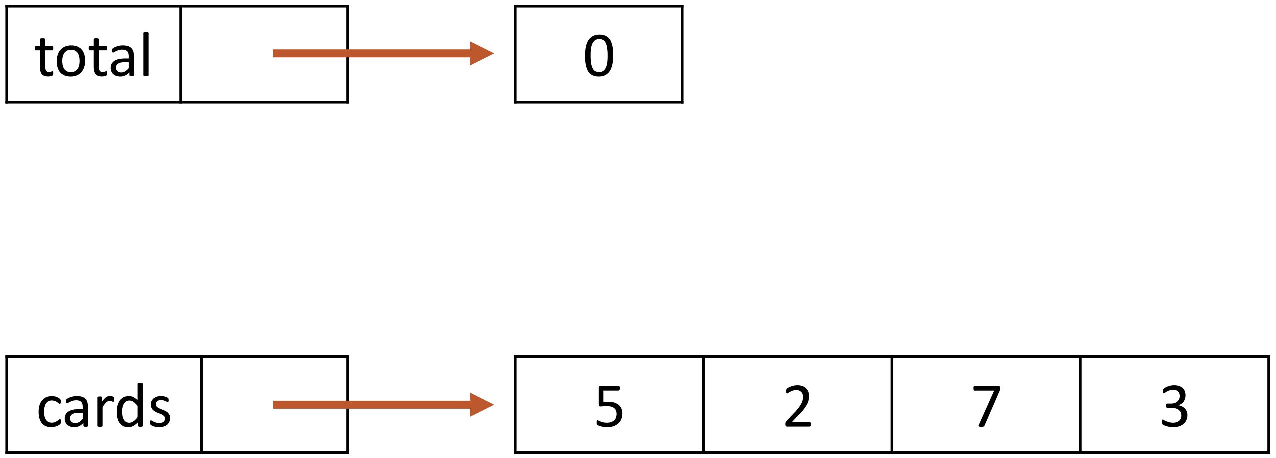
We could implement this in code with the following function:

```
def iterativeAddCards(cards):
    total = 0
    for i in range(len(cards)):
        total = total + cards[i]
    return total
```

Implementing Recursion

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

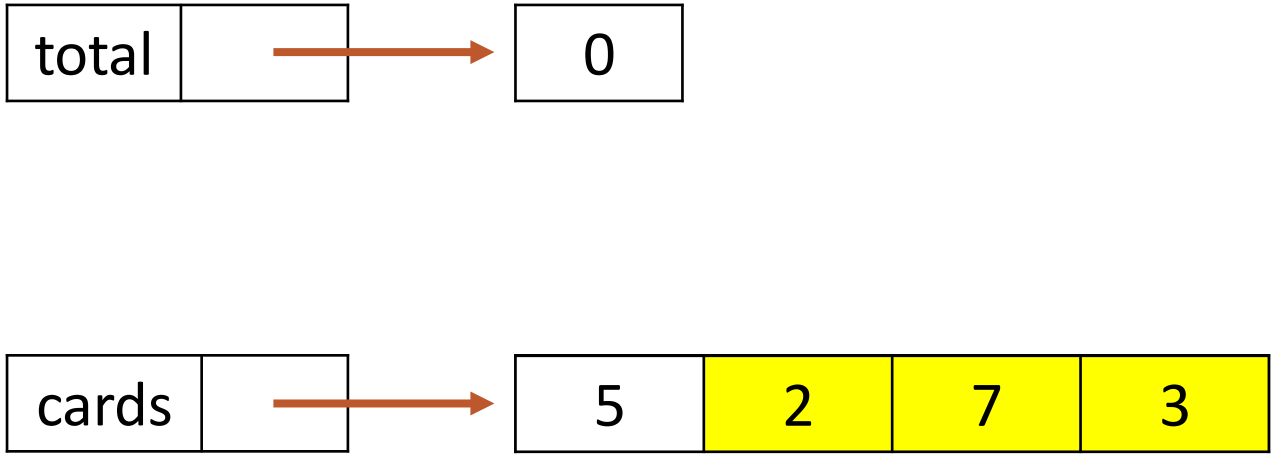
Start State:



Implementing Recursion

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

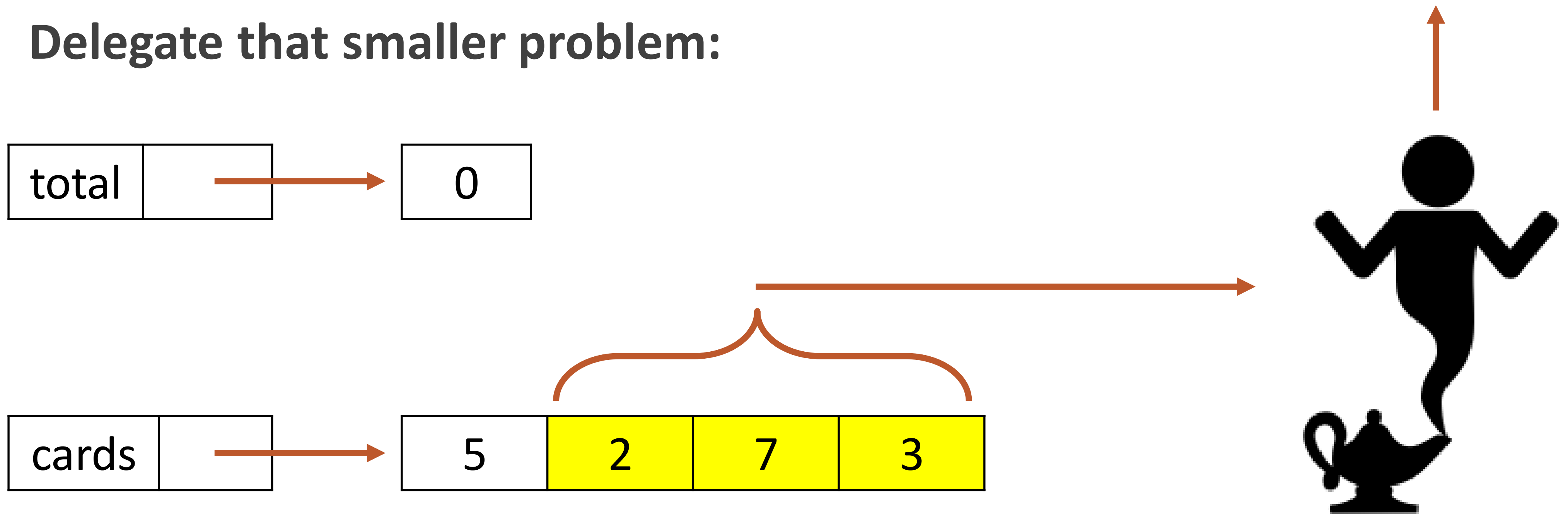
Make the problem smaller:



Implementing Recursion

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

Delegate that smaller problem:

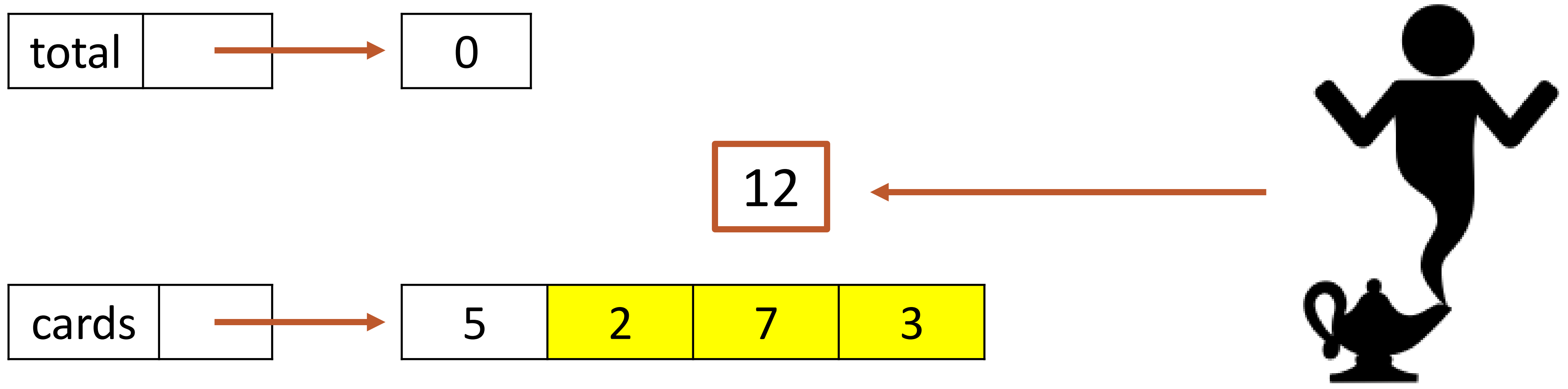


This is the Recursion Genie. They can solve problems, but only if the problem has been made slightly smaller than the start state.

Implementing Recursion

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

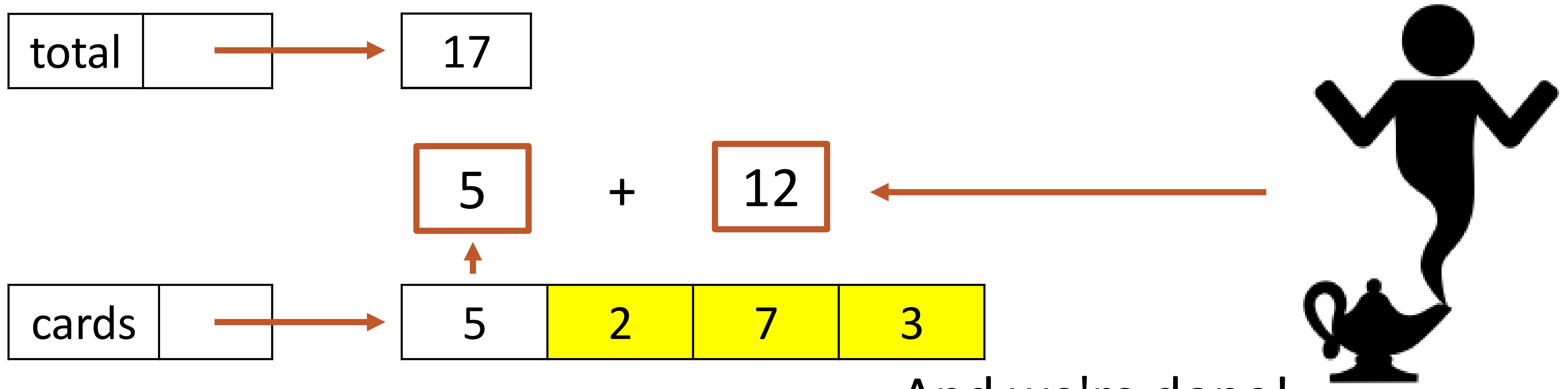
Get the smaller problem's solution:



Implementing Recursion

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

Combine the leftover solution with the smaller solution:



And we're done!

Recursion in Code

Now let's implement the recursive approach in code.

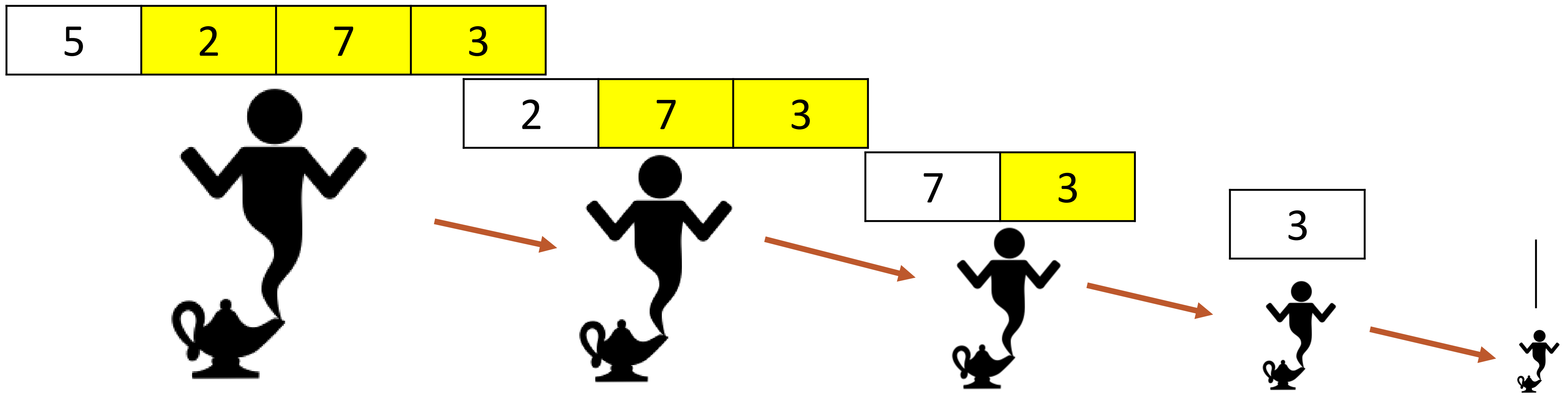
```
def recursiveAddCards(cards):
    smallerProblem = cards[1:]
    smallerResult = ??? # how to call the genie?
    return cards[0] + smallerResult
```

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.

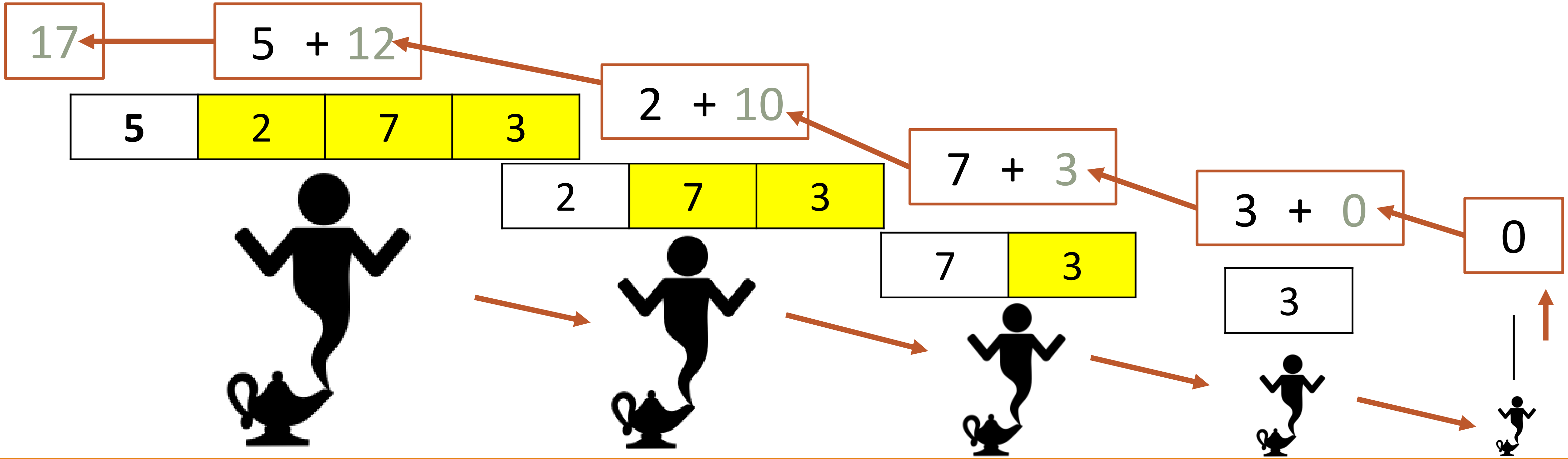
Every time the function is called, the problem gets smaller again. Eventually, the problem reaches a state where we can't make it smaller. We'll call that the **base case**.



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-solver and start the chain of solutions.



Recursion in Code – Recursive Call

To update our recursion code, we'll take two steps. First, we need to add the call to the function itself.

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

Recursion in Code – Base Case

Second, we 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

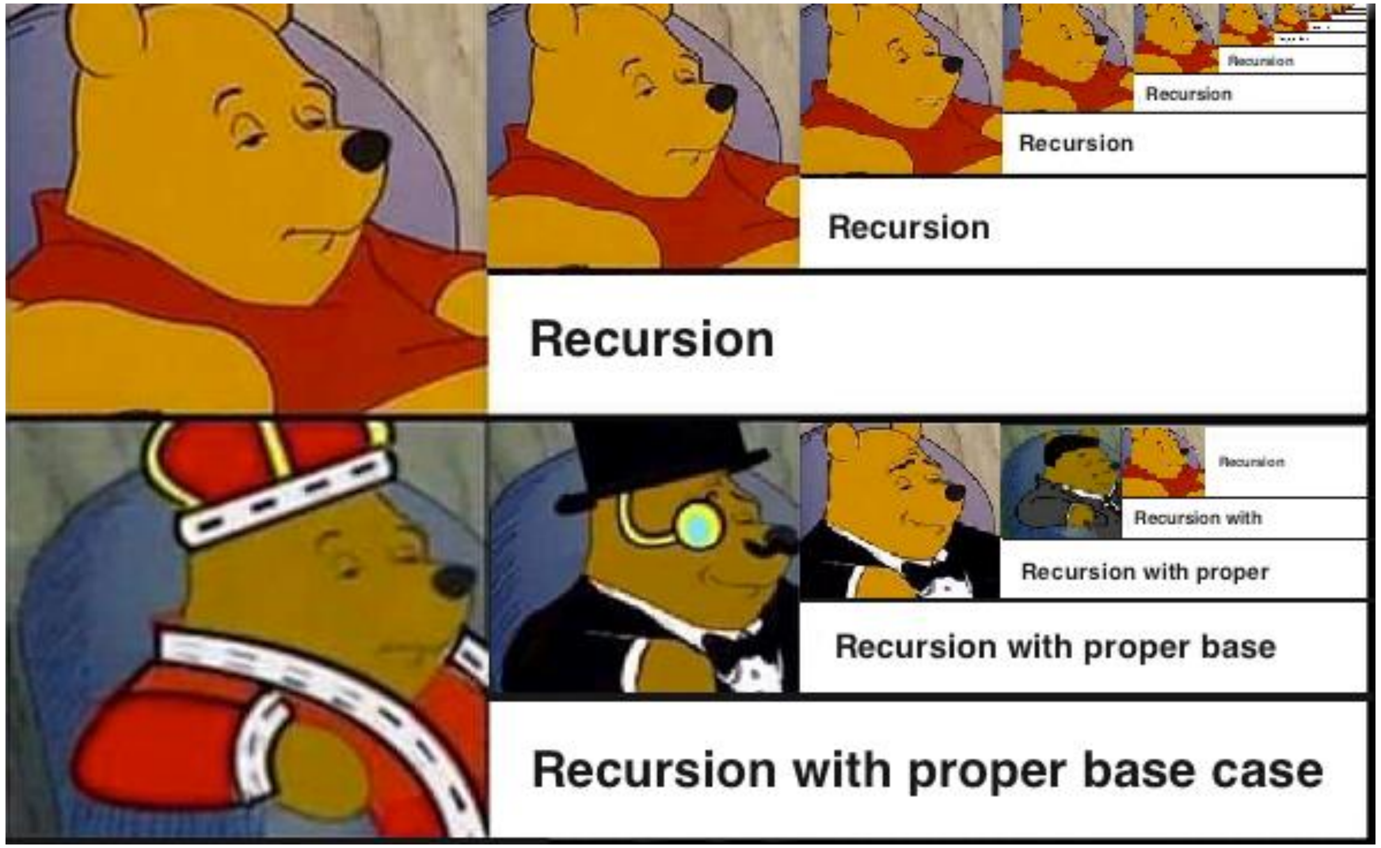
Python Tracks Recursion Calls

Recall how we visualized how Python keeps track of nested function calls with 'bookmarks'.

Python also uses this approach 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.

Simplified Call Stack Representation



Activity: Base Case/Recursive Case

Let's write an algorithm that takes a binary number (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? You don't need to write code, just consider the algorithmic cases.

Hint: consider how you converted binary to decimal in the advanced content for week1.

Activity Solution

Base case: when there's a single digit, you can convert that digit to an integer and return it.

Recursive case: slice the first number off, turn it into an int, and multiply it by $2^{len(num)-1}$. Then recurse over the rest of the number and add the two values together.

Alternate Recursive case: slice the last number off. Recurse over the rest of the number and multiply the result by 2. Cast the last number to an int and add it to the result.

Programming with Recursion

General Recursive Form

Thinking of recursive algorithms can be tricky at first. However, 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 (takes an int, returns an int). Recall that:

```
x! = x*(x-1)*(x-2)*...*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.

Simplified Infinite Recursion

Learn to program
Make recursive function
No exit condition
Learn to program
Make recursive function
No exit condition

Example: countVowels(s)

Let's do another example. Write the function countVowels(s) that takes a string and recursively counts the number of vowels in that string, returning an int. For example, countVowels("apple") would return 2.

```
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
```

Example: removeDuplicates(lst)

Let's do one final example. Write the function `removeDuplicates(lst)` that takes a list of items and recursively generates a new list that contains only one of each unique item from the original list. For example, `removeDuplicates([1, 2, 1, 2, 3, 4, 3, 3])` might return `[1, 2, 3, 4]`.

```
def removeDuplicates(lst):
    if ____________: # base case
        return ________
    else: # recursive case
        smaller = removeDuplicates(_______)
        return ______________
```

Example: `removeDuplicates(lst)`

The recursive case generates a list that holds only unique elements.

Just check whether the remaining element is already in that list or not!

```
def removeDuplicates(lst):
    if lst == []: # base case
        return []
    else: # recursive case
        smaller = removeDuplicates(lst[1:])
        if lst[0] in smaller:
            return smaller
        else:
            return [lst[0]] + 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`.

Note: you can index into and slice both lists at the same time!

Another note: when it comes to writing recursive code, **be optimistic**. Write a solution that should work *assuming the recursive call gives the proper result*.

Activity Solution

```
def recursiveMatch(lst1, lst2):
    if len(lst1) == 0:
        return 0
    else:
        partialResult = recursiveMatch(lst1[1:], lst2[1:])
        if lst1[0] == lst2[0]:
            return partialResult + 1
        else:
            return partialResult
```

Multiple Recursive Calls

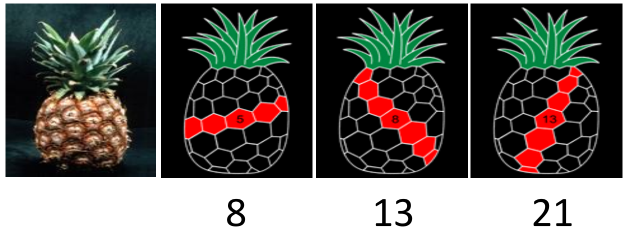
Multiple Recursive Calls

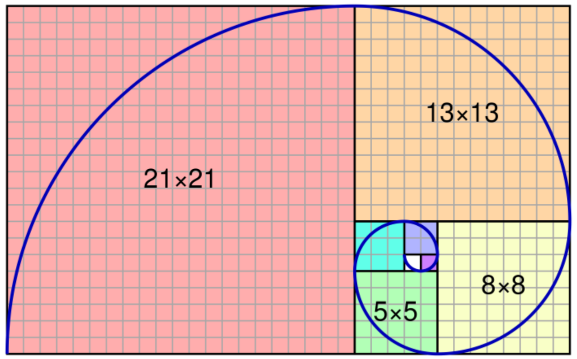
So far, we've used just one recursive call to build up a recursive answer.

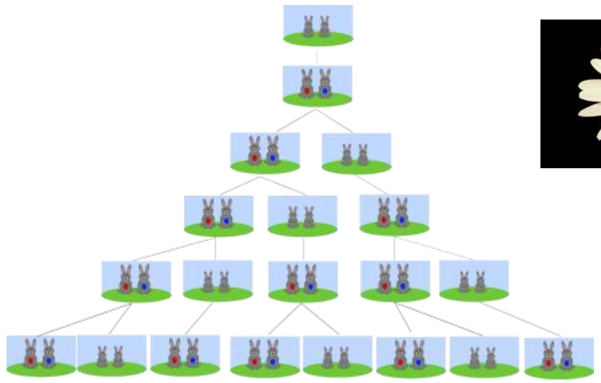
The real **conceptual** power of recursion happens when we need more than one recursive call!

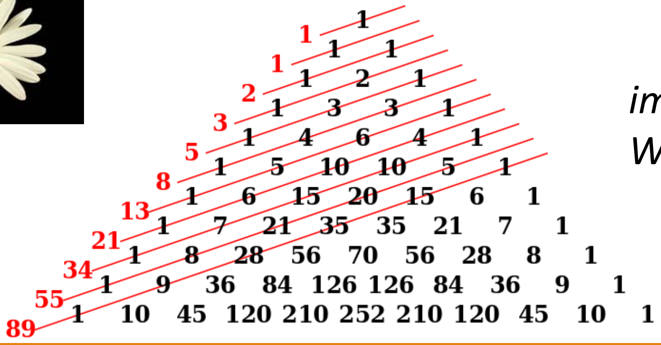
Example: Fibonacci numbers

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, etc.









images from Wikipedia

Code for Fibonacci Numbers

The Fibonacci number pattern goes as follows:

F(0) = 0

F(1) = 1

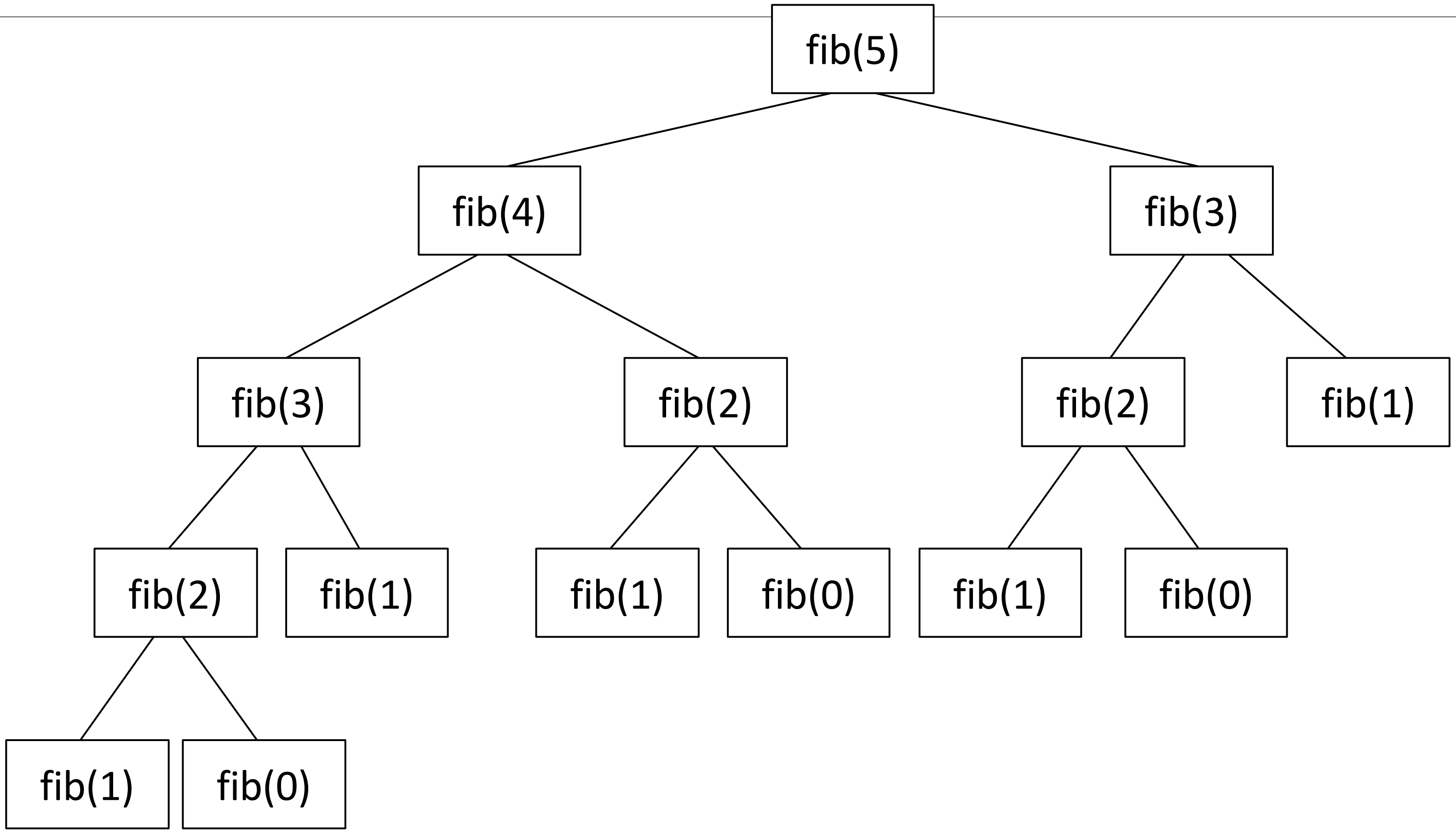
F(n) = F(n-1) + F(n-2), n > 1

```
def fib(n):
    if n == 0 or n == 1:
        return n
    else:
        return fib(n-1) + fib(n-2)
```

Two recursive calls!

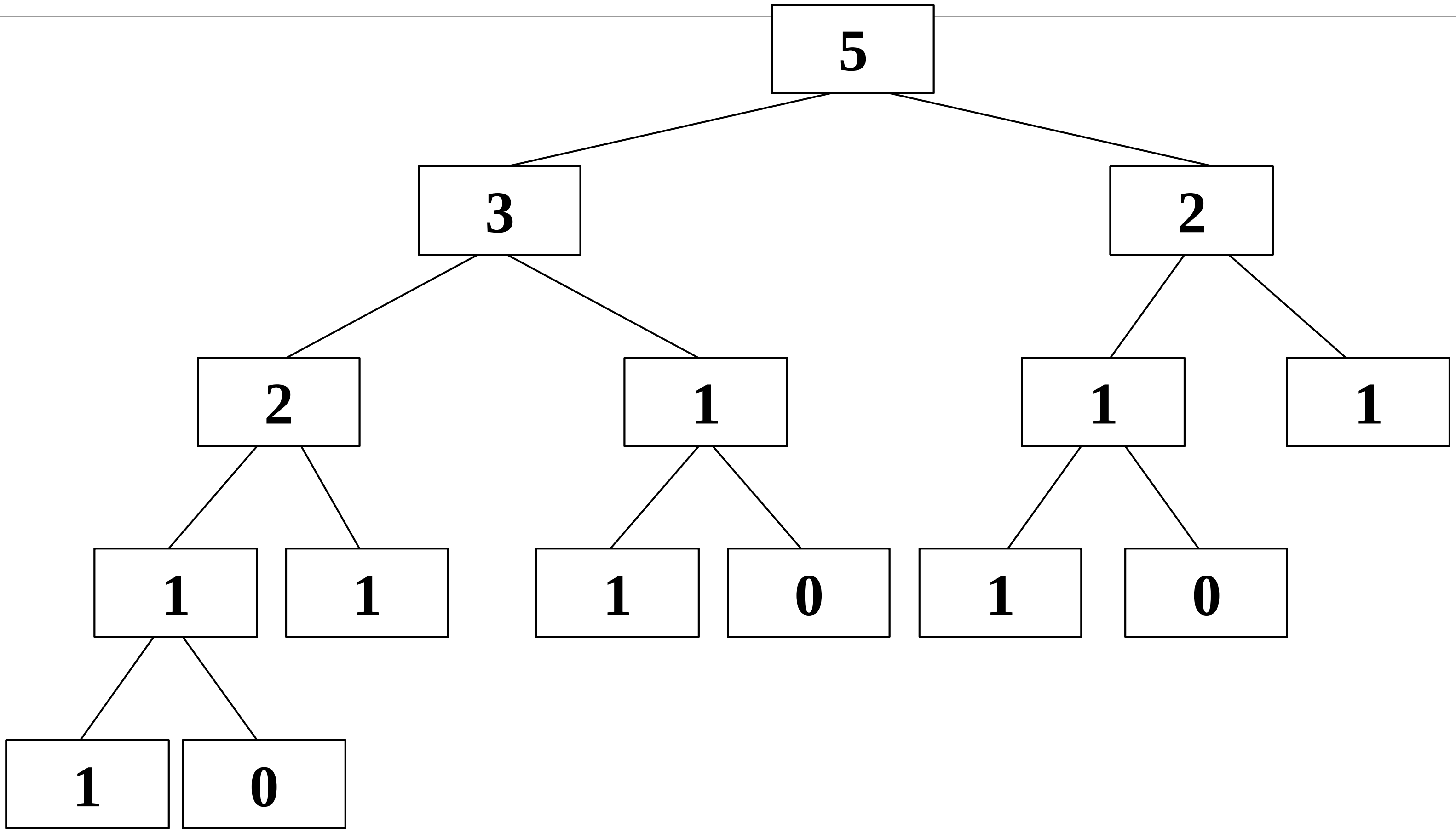
Fibonacci Recursive Call Tree

fib(0) = 0
fib(1) = 1
fib(n) = fib(n-1) + fib(n-2), n > 1



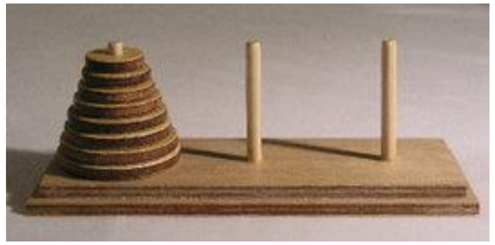
Fibonacci Recursive Call Tree

fib(0) = 0
fib(1) = 1
fib(n) = fib(n-1) + fib(n-2), n > 1



Another Example: Towers of Hanoi

Legend has it that long ago at a temple far away, a priest was led to a courtyard with 64 discs stacked in size order on a sacred platform.



The priest needed to move all 64 discs from this sacred platform to the second sacred platform, but there was only one other place (let's say a sacred table) on which they could temporarily place the discs.

The priest could move only one disc at a time, because they're heavy. And they could not put a larger disc on top of a smaller disc at any time, because the discs were fragile.

According to the legend, the world would end when the priest finished their work.

How long will this task take?

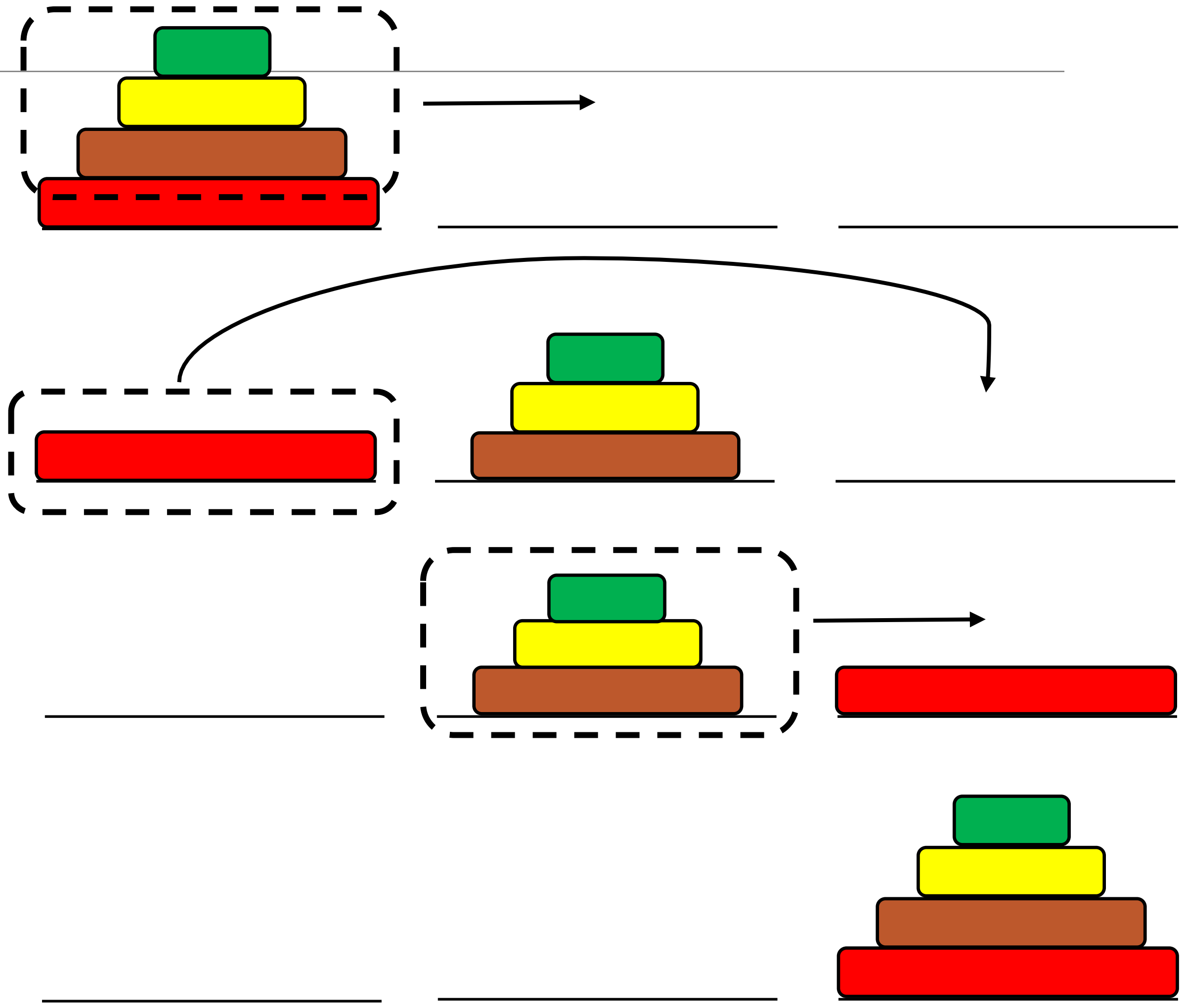
Solving Hanoi – Use Recursion!

It's difficult to think of an iterative strategy to solve the Towers of Hanoi problem. Thinking recursively makes the task easier.

The base case is when you need to move one disc. Just move it directly to the end platform.

Then, given N discs:

1. Delegate moving **all but one** of the discs to the temporary platform.
2. Move the remaining disc to the end platform.
3. Delegate moving the **all but one** pile to the end platform.



Solving Hanoi - Code

```
# Prints instructions to solve Towers of Hanoi and
# returns the number of moves needed to do so.
def moveDiscs(start, tmp, end, discs):
    if discs == 1: # 1 disc - move it directly
        print("Move one disc from", start, "to", end)
        return 1
    else: # 2+ discs - move N-1 discs, then 1, then N-1
        moves = 0
        moves = moves + moveDiscs(start, end, tmp, discs - 1)
        moves = moves + moveDiscs(start, tmp, end, 1)
        moves = moves + moveDiscs(tmp, start, end, discs - 1)
        return moves

result = moveDiscs("left", "middle", "right", 3)
print("Number of discs moved:", result)
```

Activity: Towers of Hanoi Steps

Our original question was: how many steps will it take to move 64 discs?

We can calculate this by asking a different question: if we add one disc to a Towers of Hanoi set, how does that affect the total number of steps that need to be taken?

Number of Moves in Towers of Hanoi

Every time we add another disc to the tower, it **doubles** the number of moves we make.

It doubles because moving N discs takes moves(N-1) + 1 + moves(N-1) total moves.

We can approximate the number of moves needed for the 64 discs in the story with 2^{64}. That's 1.84×10^{19} moves!

If we estimate each move takes one second, then that's (1.84×10^{19}) / (60*60*24*365) = 5.85×10^{11} years, or **585 billion years!** We're safe for now.

Learning Objectives

Define and recognize **base cases** and **recursive cases** in recursive code

Read and write basic **recursive code**

Trace over recursive functions that use **multiple recursive calls** with Towers of Hanoi