Recursion

15-110 – Wednesday 09/24

Quizlet3

Learning Objectives

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

- Read and write basic **recursive code**

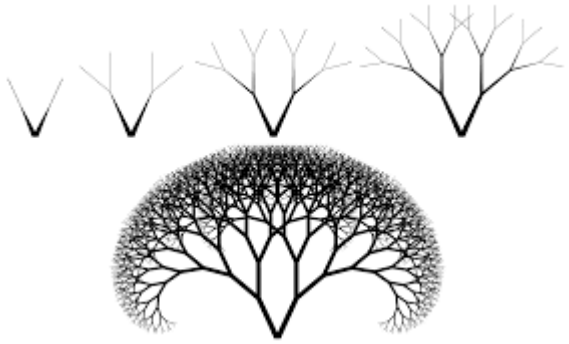
Concept of Recursion

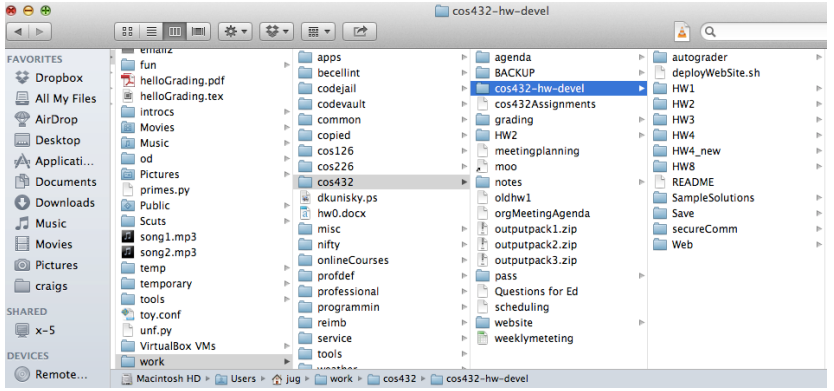
Concept of Recursion

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

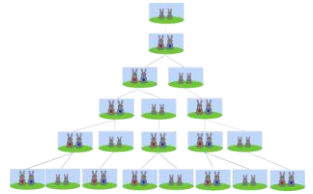




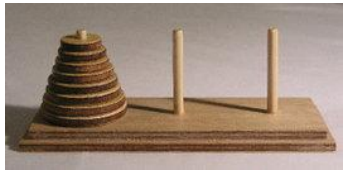
Why Use Recursion?

Recursion is a hard concept to master 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 in the future.



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 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?

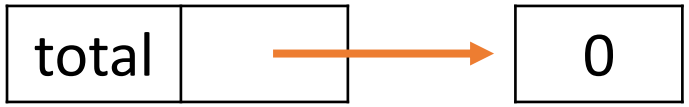
Iterative approach: keep track of the total so far, iterate over the cards, add each to the total.

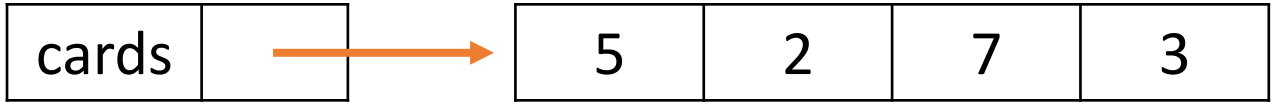
Recursive approach: take a card off the deck, **delegate adding the rest of the deck to someone else**, then when they give you the answer, add the remaining card to their sum.

Implementing Iteration

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

Pre-Loop:

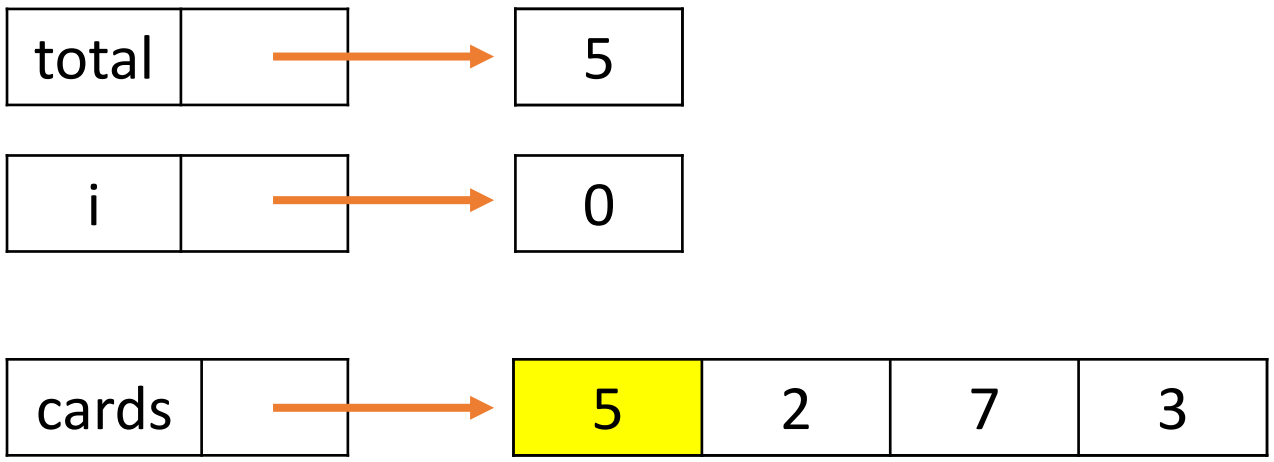




Implementing Iteration

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

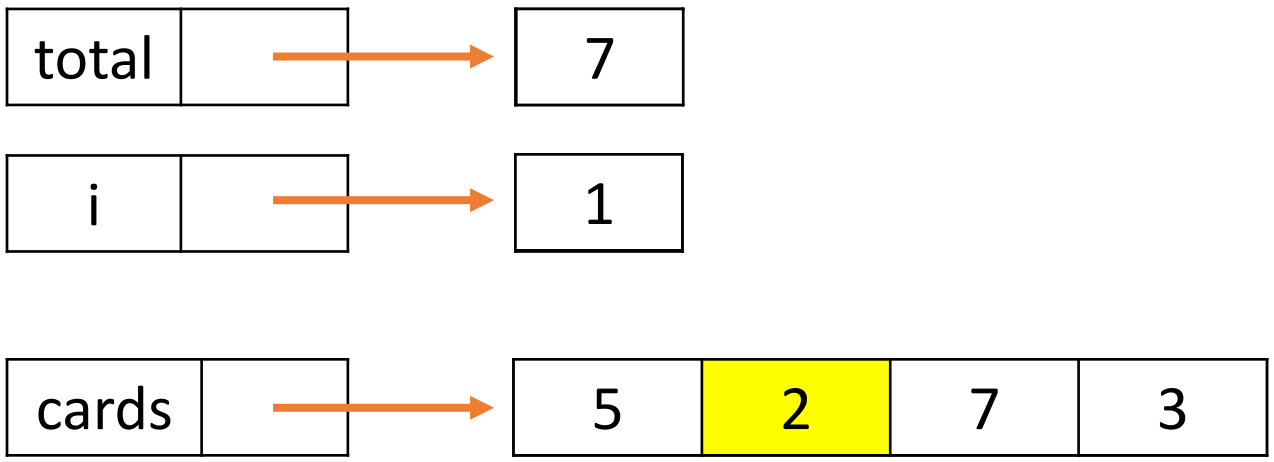
First iteration:



Implementing Iteration

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

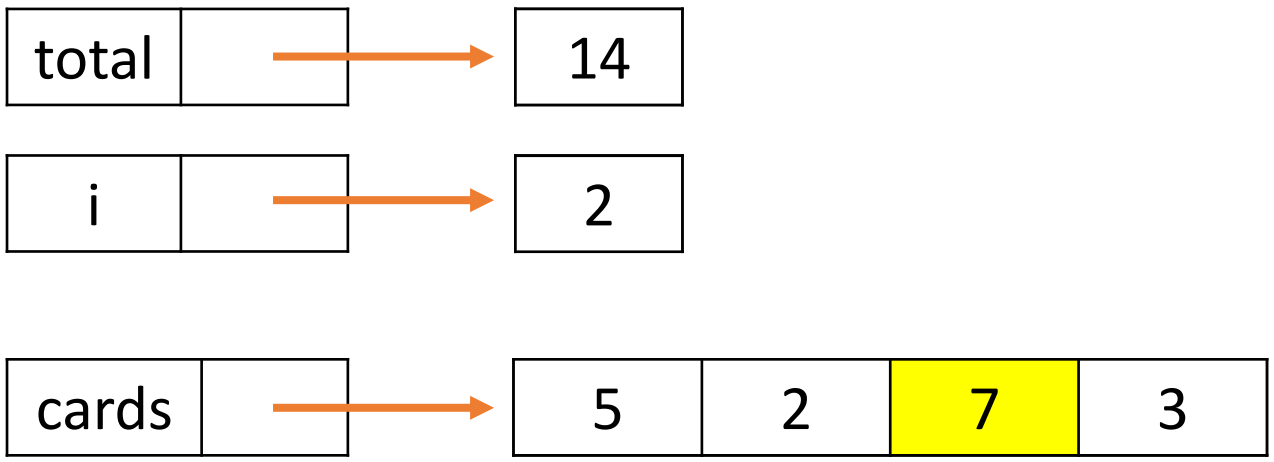
Second iteration:



Implementing Iteration

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

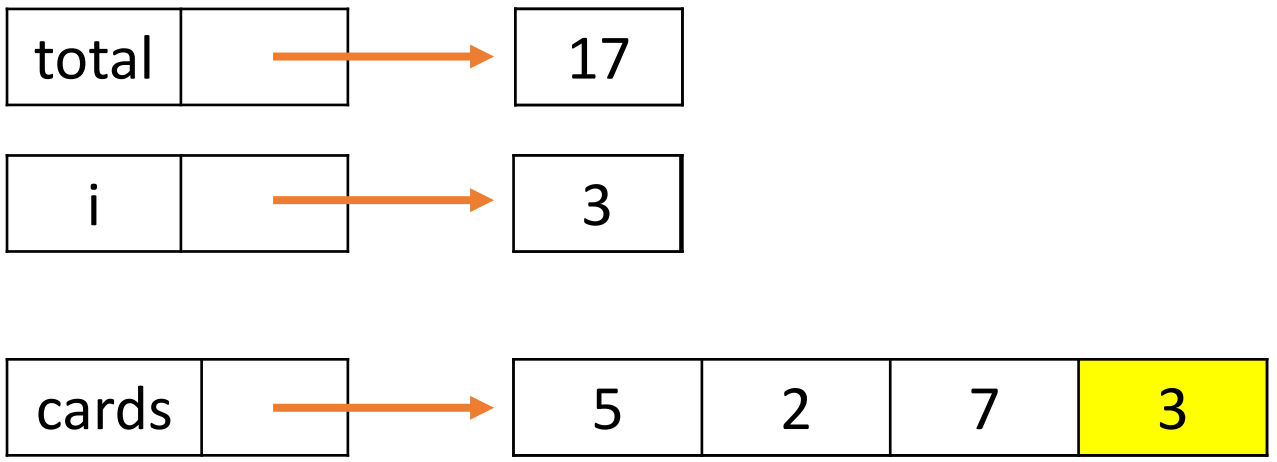
Third iteration:



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

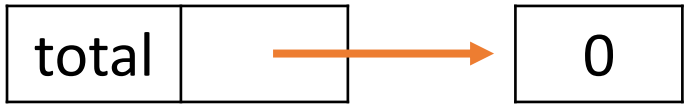
We could implement this in code with the following function:

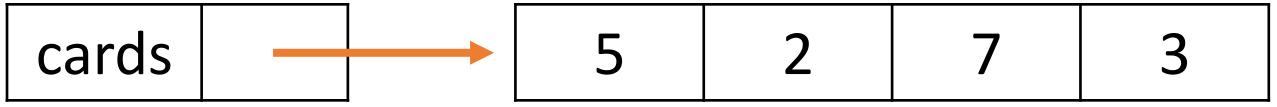
```
def iterativeAddCards(cards):
    total = 0
    for i in range(len(cards)):
        total = total + cards[i]
    return total
```

Implementing Recursion

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

Start State:

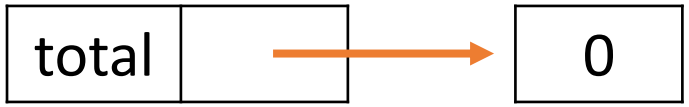




Implementing Recursion

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

Make the problem smaller:

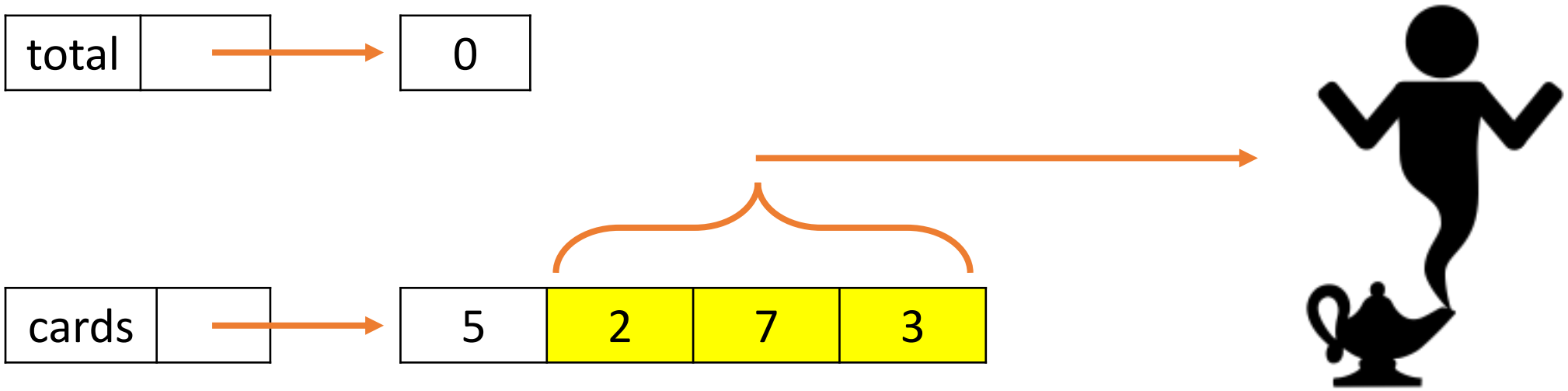


Implementing Recursion

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

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

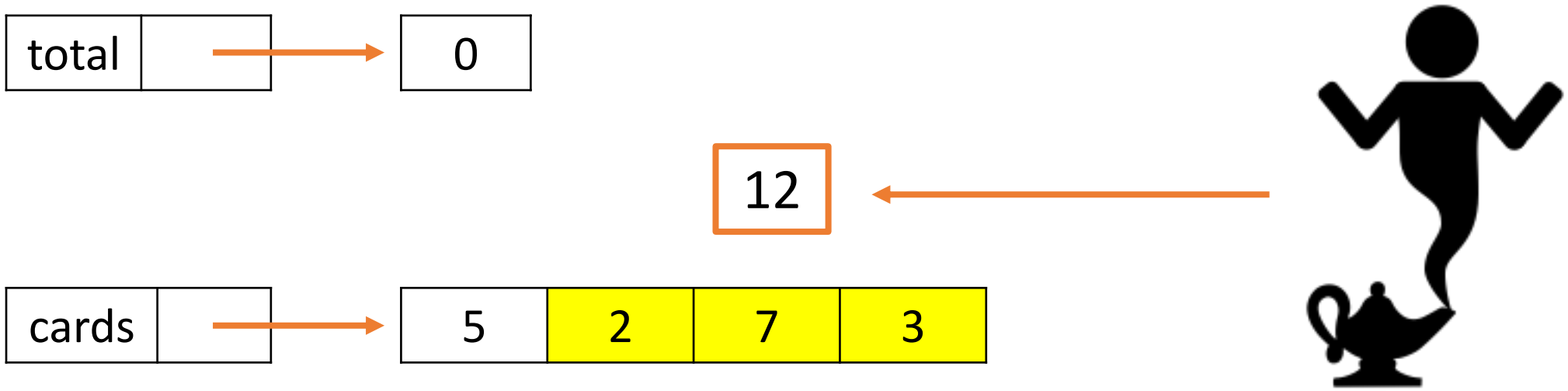
Delegate that smaller problem:



Implementing Recursion

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

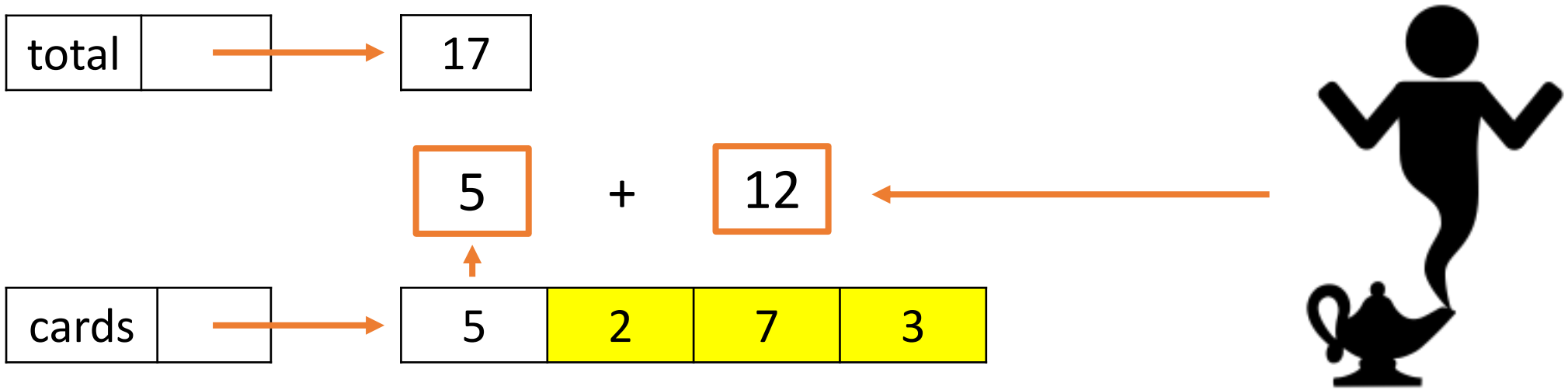
Get the smaller problem's solution:



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.

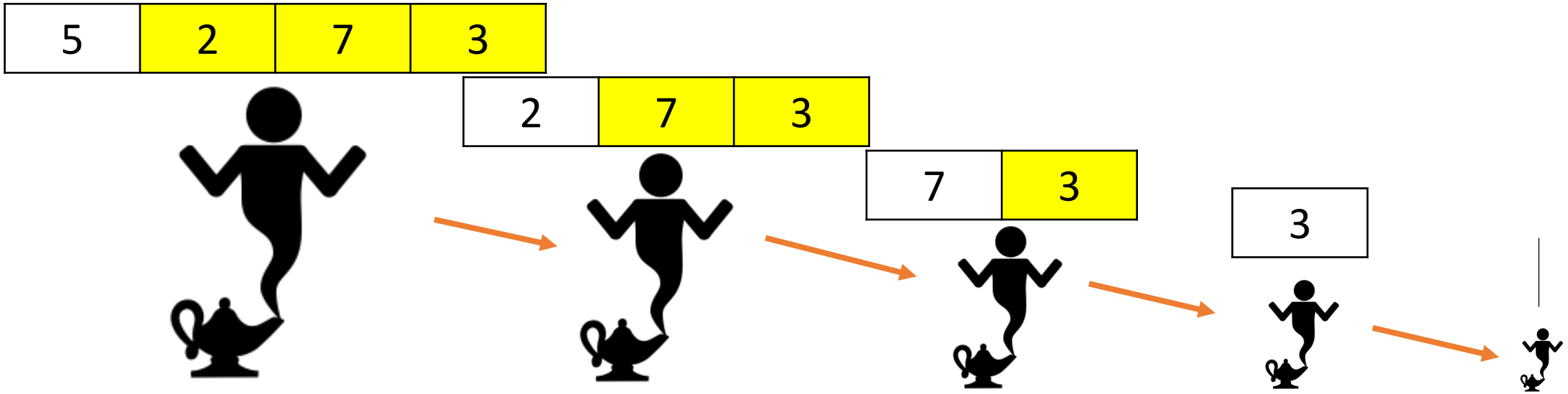
```
def recursiveAddCards(cards):
    smallerProblem = cards[1:]
    smallerResult = ??? # how to call the genie?
    return cards[0] + smallerResult
```

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.

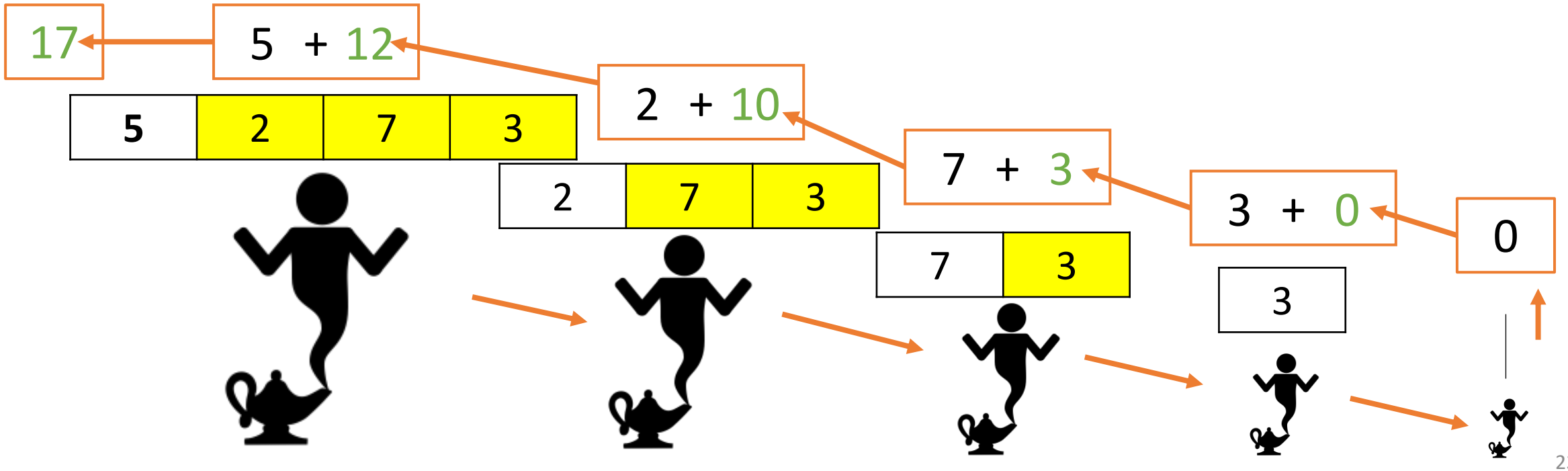
Every time the function is called, the problem gets smaller again. Eventually, the problem reaches a state where we can't make it smaller. We'll call that the **base case**.



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 the numbers on 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

These two big ideas are used in **all** recursive algorithms.

- **Base case**(s): One or more simple cases that can be solved with no further work
- **Recursive case**(s): One or more cases that require solving "simpler" (smaller/shorter/closer to the base case) version(s) of the original problem

```
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 with Code Tracing!

Recall how we used **tracing with bookmarks** to keep track of nested function calls. 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.

Let's switch to a different slide deck for an example.

Activity: Base Case/Recursive Case

Let's design a non-code algorithm for baking a multi-layer cake. You want to bake a cake that has n layers, but you can only bake one layer at a time. You want to use recursion to solve this problem.

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 and combine the results? You don't need to write code, just consider the algorithmic cases.

Programming with Recursion

General Recursive Form

Thinking of recursive algorithms can be tricky at first. However, most of the simple recursive functions we 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 ____ # solve using leftover part & smallerResult
```

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 leftover part 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 the 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: `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*.

Learning Objectives

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

- Read and write basic **recursive code**