Do this Piazza Poll!

https://tinyurl.com/yjk2wxvn

Recursion II & Search Algorithms

15-110 – Friday 09/27

Announcements

- Fill out Piazza Poll for topics to review during Monday's exam review lecture by tonight at 11:59pm
- Exam notes sheet posted on the course website will be given to you the day of the exam so you can refer to it during the exam
- TA-led Exam1 Review Session:
  - **TBD**
- Check3 due **Monday at noon**

Learning Objectives

- Trace over recursive functions that use **multiple recursive calls**.
- Recognize **linear search** on lists and in recursive contexts
- Use **binary search** when reading and writing code to search for items in sorted lists

Programming with Recursion

Example: `countVowels(s)`

Problem: 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.

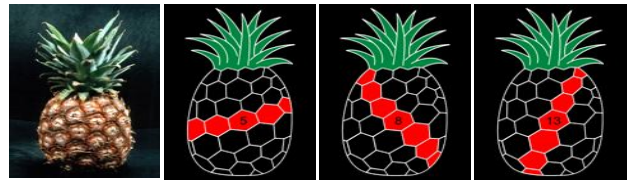
```
def countVowels(s):
    if s == "": # base case
        return 0
    else: # recursive case
        smaller = countVowels(s[1:])
        if s[0] in "AEIOU":
            return 1 + smaller
        else:
            return smaller
```

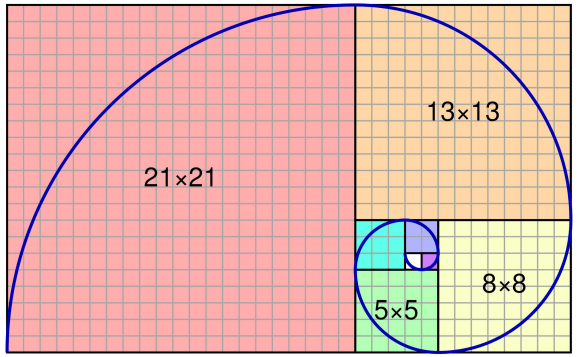
Multiple Recursive Calls

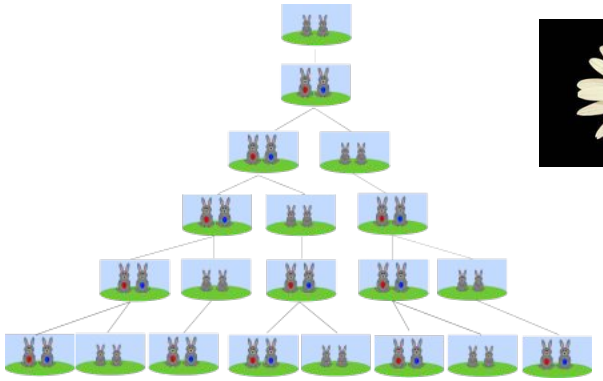
The real conceptual power of recursion happens when we need **more than one recursive call**!

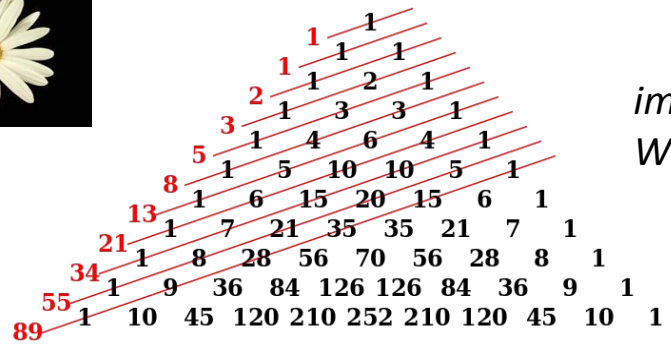
Example: Fibonacci numbers

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









images from Wikipedia

The **Fibonacci sequence** is a sequence in which each number is the sum of the two preceding ones.

F(0) = 0

F(1) = 1

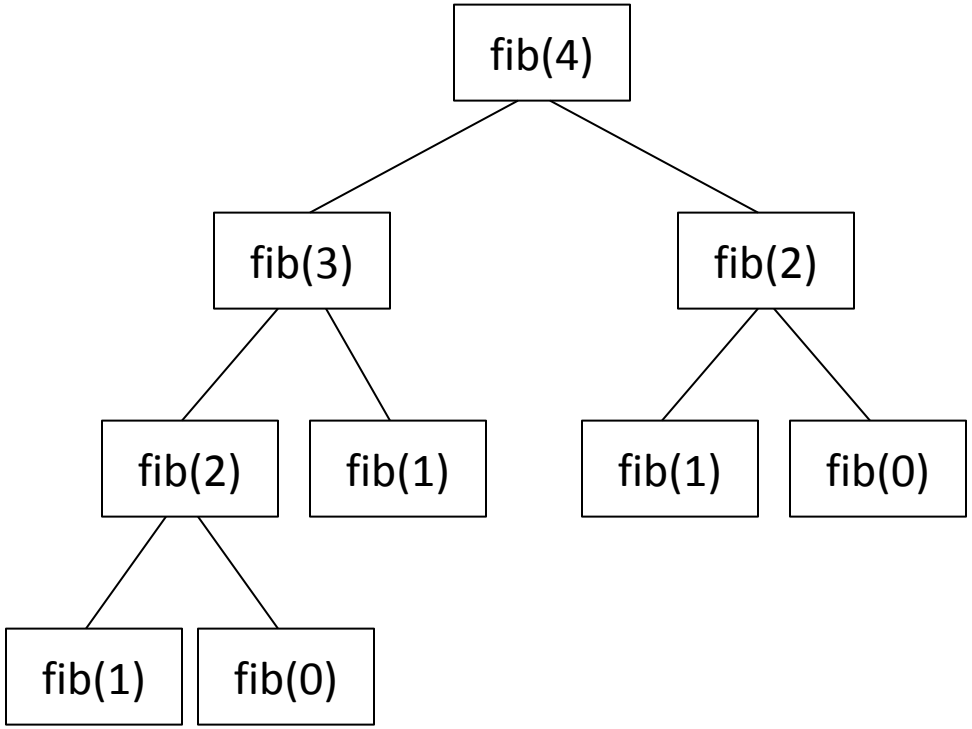
F(n) = F(n-1) + F(n-2), n > 1

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

Two recursive calls!

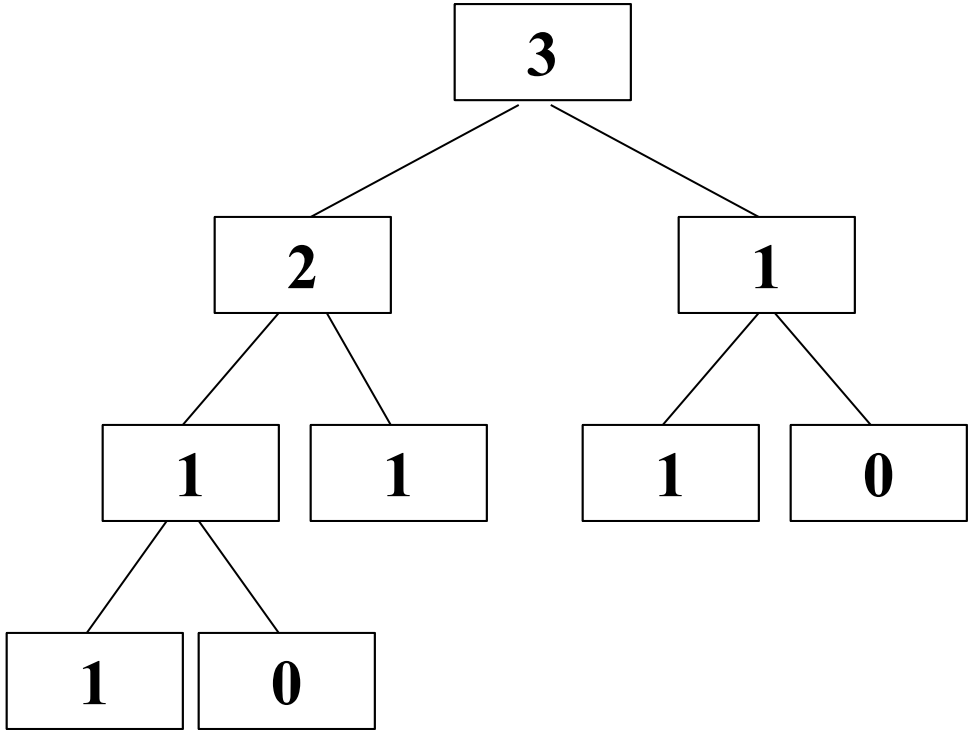
Fibonacci Recursive Call Tree

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

Goal: Move the entire stack of disks from one rod to another with the following restrictions:

1. Only **one disk may be moved** at a time.
2. Each move consists of taking the upper disk from one of the stacks and placing it on top of another stack or on an empty rod.
3. **No disk may be placed on top of a disk that is smaller than it.**

Solving Hanoi – Use Recursion!

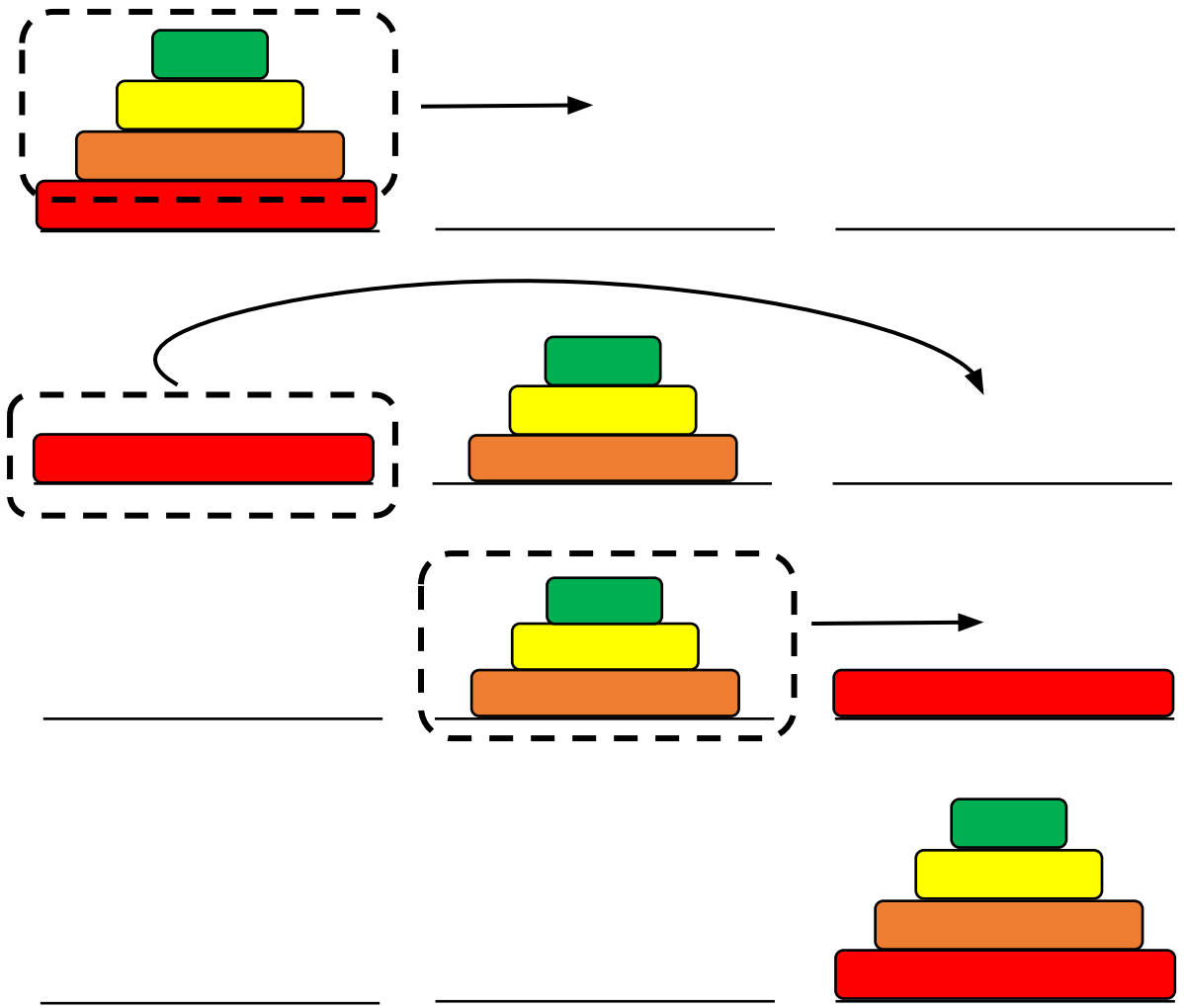
Online version:
https://www.mathsisfun.com/games/towerofhanoi.html

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. Directly 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
def moveDiscs(start, tmp, end, discs):
    if discs == 1: # 1 disc - move it directly
        print("Move one disc from", start, "to", end)
    else:
        # Move all but largest disc out of the way
        moveDiscs(start, end, tmp, discs - 1)

        # Move the largest disc to the goal
        moveDiscs(start, tmp, end, 1)

        # Move all but largest disc to the goal
        moveDiscs(tmp, start, end, discs - 1)

moveDiscs("left", "middle", "right", 3)
```

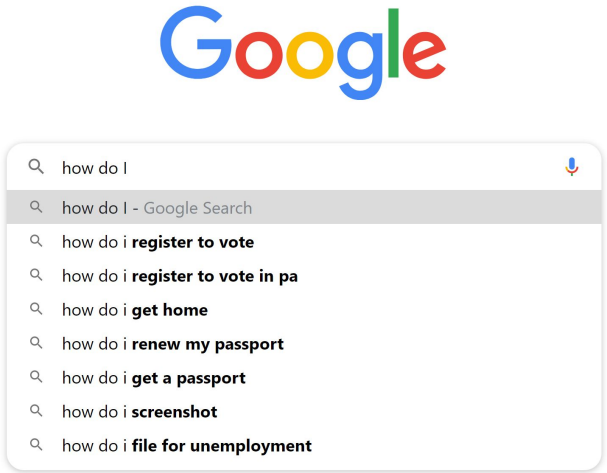
Solving Hanoi - Alternate version to get number of moves

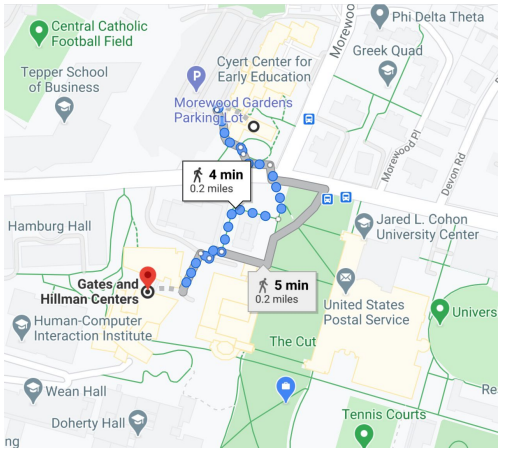
```
# Prints instructions to solve Towers of Hanoi and
# returns the number of moves needed to do so.
def moveDiscs(start, tmp, end, discs):
    if discs == 1: # 1 disc - move it directly
        print("Move one disc from", start, "to", end)
        return 1
    else: # 2+ discs - move N-1 discs, then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

Linear Search

Search is one of the most common tasks a computer needs to do.





You use search in real life all the time too! Every time you manually look through papers or other physical documents for information, you conduct a search algorithm of your own.

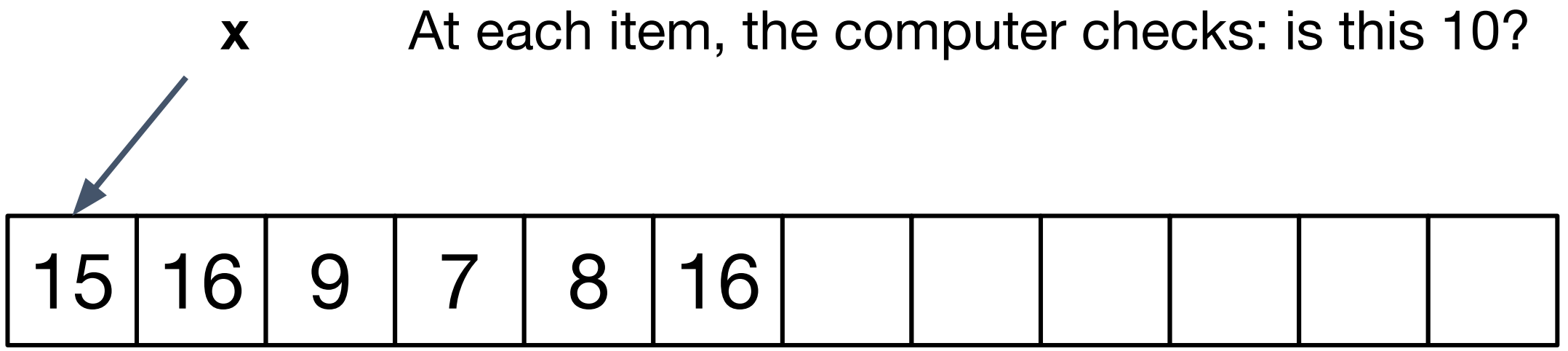
When we want to check if a list contains a specific value we use the `in` operator.

```
x = [15, 16, 9, 7, 8, 16]
if 10 in x:
   print("Found it!")
```

What algorithm does `in` implement?

We'll need to think about this from a computer's perspective...

A computer sees the list as a series of not-yet-known values and **needs to check each item**.



Analogy: Searching a Stack of Books

This is like looking for a particular book in a stack of books.

You need to repeatedly check each book until you find the right title, or until you've checked them all.

This is **linear search**: check each item in order.

We can implement this with a `for` loop:

```
def linearSearch(L, target):
    for i in range(len(L)):
        if L[i] == target:
            return True
    return False
```

You do: If `target` appears more than once in `lst`, which value will cause the function to return?

Search follows **common patterns for functions** that use a loop to return a Boolean: check-any and check-all

A **check-any** pattern returns `True` if any of the items in the list meet a condition, and `False` otherwise.

```
def checkAny(L, target):
    for i in range(len(L)):
        if L[i] == target:
            return True
    return False
```

A **check-all** pattern returns `True` if all of the items in the list meet a condition, and `False` otherwise.

```
def checkAll(L, target):
    for i in range(len(L)):
        if L[i] != target:
            return False
    return True
```

For practice, how can we implement linear search recursively?

How do we make the problem smaller?

Answer: Call the linear search on all but the first element of the list.

What's the base case for linear search?

Answer: an empty list. The item can't possibly be in an empty list, so the result is `False`.

Also: a list where the first element is what we're searching for, so the result is `True`.

How do we combine the solutions?

Answer: no combination necessary. The recursive call returns whether the item occurs in the rest of the list; just return that result unmodified.

Recursive Linear Search Code

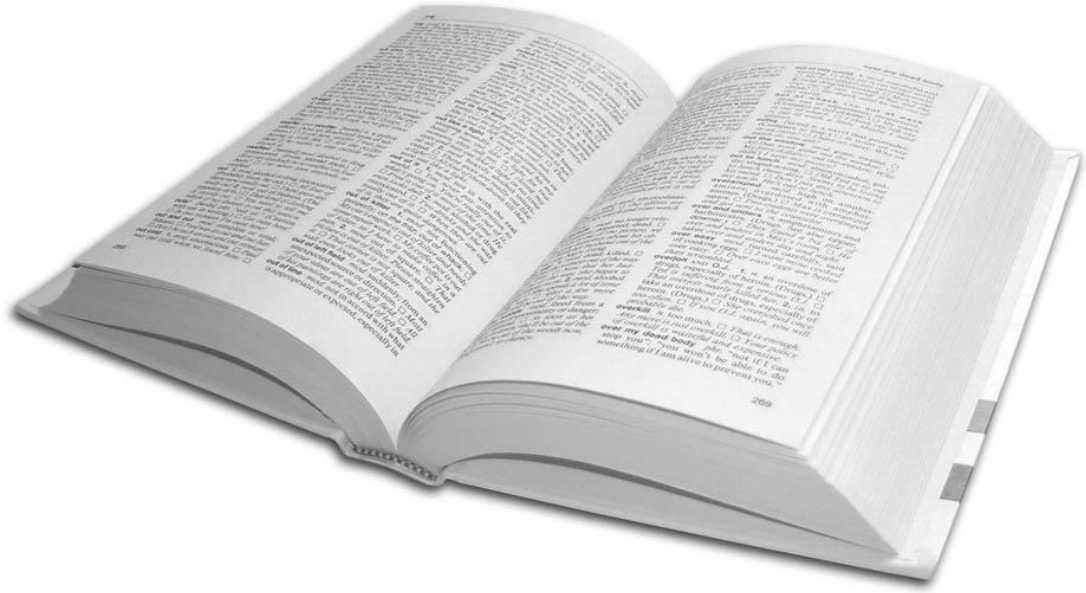
```
def recursiveLinearSearch(L, target):
    if L == [ ]:
        return False
    else:
        if L[0] == target:
            return True
        else:
            return recursiveLinearSearch(L[1:], target)

print(recursiveLinearSearch(["dog", "cat", "rabbit", "mouse"], "rabbit"))
print(recursiveLinearSearch(["dog", "cat", "rabbit", "mouse"], "horse"))
```

Alternative to Linear Search

Linear Search is a nice, straightforward approach to searching a set of items. But that doesn't mean it's the only way to search.

Assume you want to search a dictionary to find the definition of a word you just read. Would you use linear search, or a different algorithm?

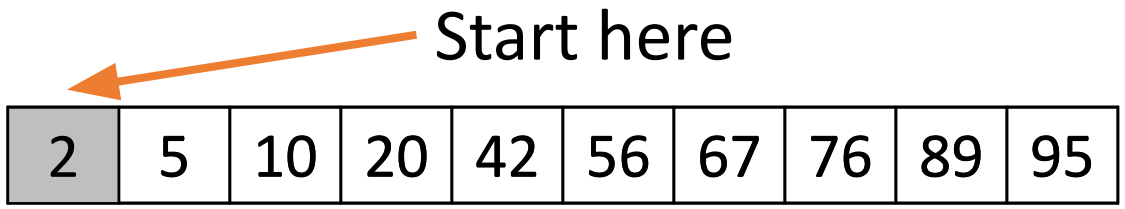


Can we take advantage of dictionaries being **sorted**?

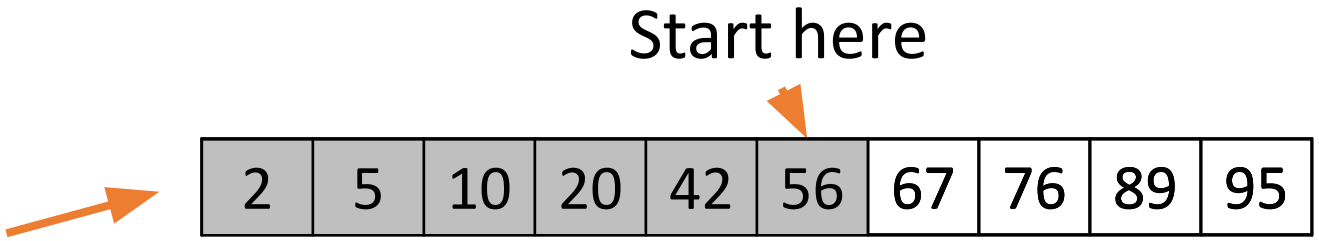
Binary Search

Binary Search Divides the List Repeatedly

In **Linear Search**, we start at the beginning of a list and check each element in order. So if we search for 98 and do one comparison...



In **Binary Search** on a **sorted list**, we'll start at the **middle** of the list and **eliminate** half the list based on the comparison we do. When we search for 98 again...



Analogy: Searching in a Library

If you're looking for a particular book in a library, you don't have to check every single book!

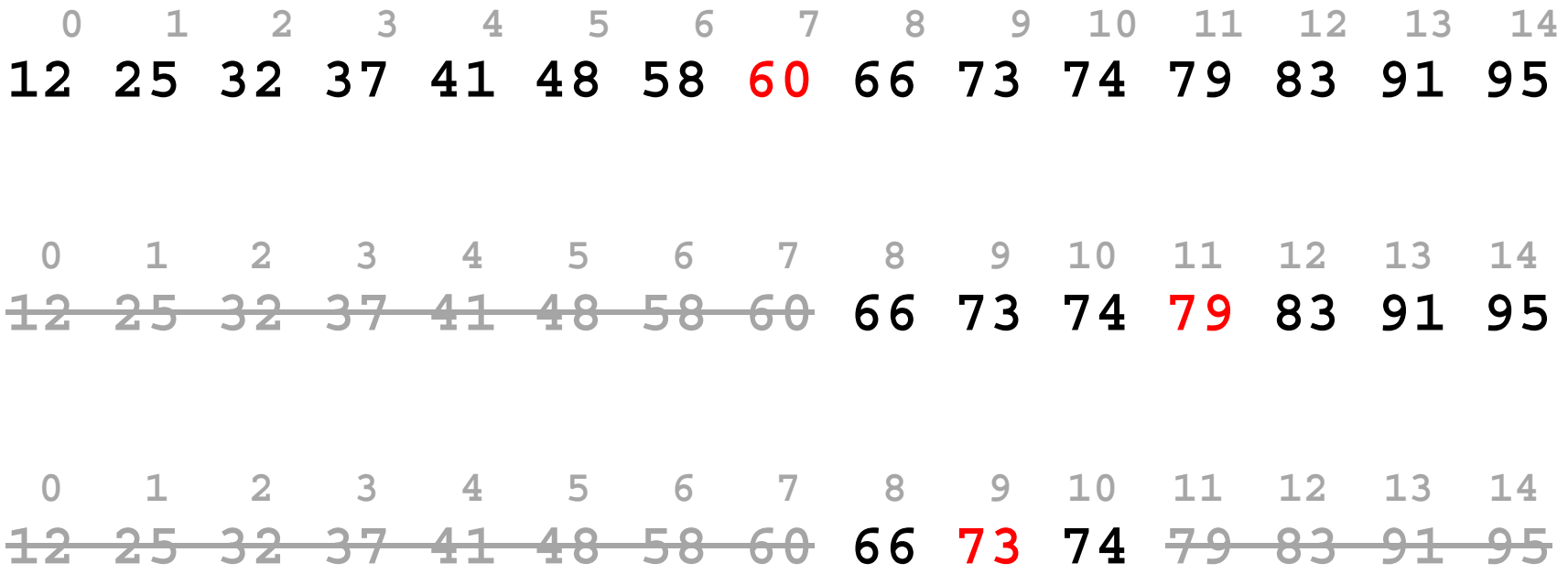
You can navigate to the right location because the books are **sorted** and you know your book's **author** already.

You can use existing information to speed up your algorithm!

We can implement **binary search** using recursion.

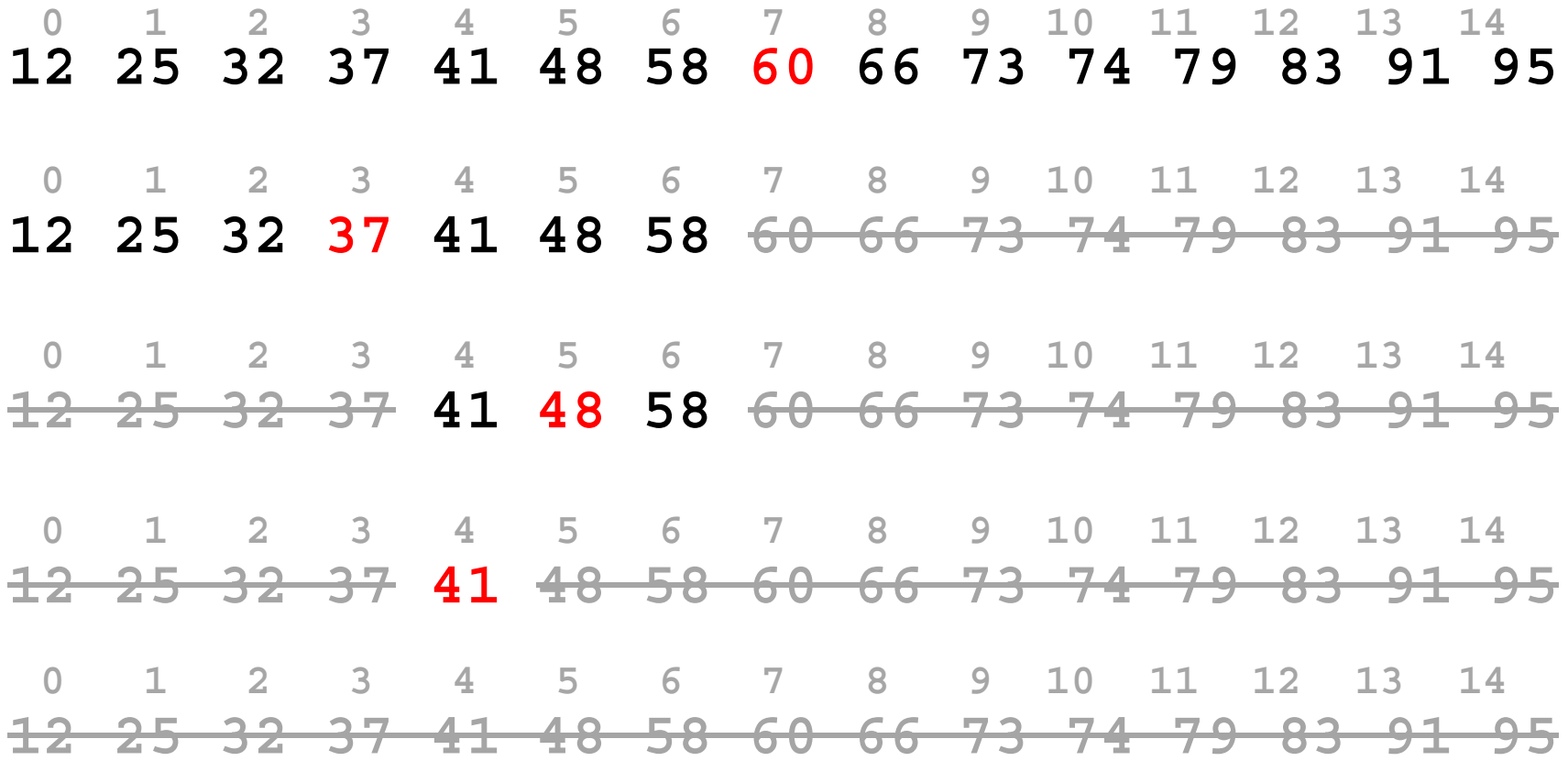
1. Find the middle element of the list.
2. Compare the middle element to the target.
3. If they're equal – you're done!
4. If the item is smaller – recursively search to the left of the middle.
5. If the item is bigger – recursively search to the right of the middle.

Example 1: Search for 73



Found: return True

Example 2: Search for 42



Not found: return False

Activity: Trace Binary Search

You do: determine the correct trace for the following call to binary search. Which numbers are visited?

```
binarySearch([2, 7, 11, 18, 19, 32, 45, 63, 84, 95, 97], 95)
```

What are the **base case and recursive case** of binary search?

How do we make the problem smaller?

Answer: get rid of the half of the list we know the target isn't in (which half?).

What are the base cases for binary search?

Answer: an empty list. The target can't possibly be in an empty list, so the result is False.

Also: a list where the target is the middle element. Then we can stop searching and immediately return True.

How do we combine the solutions?

Answer: no need to combine anything. Simply return the result of the recursive function call.

Binary Search in Code

Now we just need to translate the algorithm to Python.

```
def binarySearch(lst, target):
    if ____ # base case
        return _____
    else:
        # Find the middle element of the list.
        # Compare middle element to the target.
            # If they're equal – you're done!
            # If the item is smaller, recursively search
            #    to the left of the middle.
            # If the item is bigger, recursively search
            #    to the right of the middle.
```

Binary Search in Code – Base Case

The first **base case** is the empty list, and `return False`

```
def binarySearch(lst, target):
    if lst == [ ]:
        return False
    else:
        # Find the middle element of the list.
        # Compare middle element to the target.
            # If they're equal – you're done!
            # If the item is smaller, recursively search
            #    to the left of the middle.
            # If the item is bigger, recursively search
            #    to the right of the middle.
```

Binary Search – Middle Element

To get the middle element, use **indexing** with half the length of the list.

```
def binarySearch(lst, target):
    if lst == [ ]:
        return False
    else:
        midIndex = len(lst) // 2
        # Compare middle element to the target.
            # If they're equal – you're done!
            # If the item is smaller, recursively search
            #    to the left of the middle.
            # If the item is bigger, recursively search
            #    to the right of the middle.
```

Use integer division in case the list has an odd length

Binary Search – Base Case

The second **base case** occurs when we find the target. Return `True`.

```
def binarySearch(lst, target):
    if lst == [ ]:
        return False
    else:
        midIndex = len(lst) // 2
        if lst[midIndex] == target:
            return True
        # If the item is smaller, recursively search
        #    to the left of the middle.
        # If the item is bigger, recursively search
        #    to the right of the middle.
```

Binary Search – Comparison

Use an `if`/`elif`/`else` statement to decide which side to use for the smaller problem.

```
def binarySearch(lst, target):
    if lst == [ ]:
        return False
    else:
        midIndex = len(lst) // 2
        if lst[midIndex] == target:
            return True
        elif target < lst[midIndex]:
            ________ # recursively search to the left of the middle
        else: # lst[midIndex] < target
            ________ # recursively search to the right of the middle
```

Binary Search – Recursive Calls

Use **slicing** to make the recursive call and return the result immediately.

```
def binarySearch(lst, target):
    if lst == [ ]:
        return False
    else:
        midIndex = len(lst) // 2
        if lst[midIndex] == target:
            return True
        elif target < lst[midIndex]:
            return binarySearch(lst[:midIndex], target)
        else: # lst[midIndex] < target
            return binarySearch(lst[midIndex+1:], target)
```

Linear Search vs. Binary Search

Why should we go through the effort of writing this more-complicated search method?

Answer: **efficiency**. Binary search is **vastly** more efficient than linear search, as it performs a lot fewer comparisons to find the same item (as long as the list is already sorted).

This makes sense intuitively, but we don't yet have a way to **prove** that binary search is more efficient. We'll introduce a way to do this soon.

Learning Goals

- Trace over recursive functions that use **multiple recursive calls**
- Recognize **linear search** on lists and in recursive contexts
- Use **binary search** when reading and writing code to search for items in **sorted** lists