Search Algorithms

15-110 - Monday 10/05

Learning Objectives

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

- Recognize **linear search** on lists and in recursive contexts

- Use **binary search** when reading and writing code to search for items in sorted lists

Multiple Recursive Calls

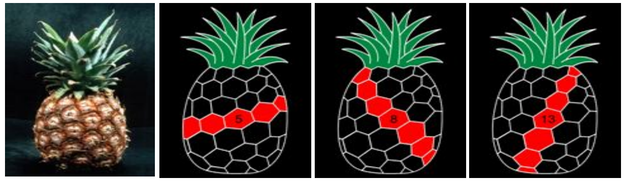
Multiple Recursive Calls

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

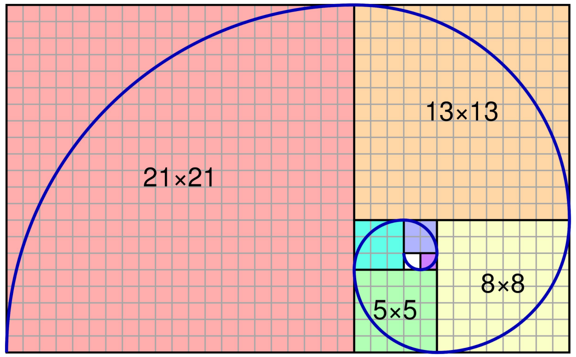
The real **conceptual** power of recursion happens when we need more than one recursive call!

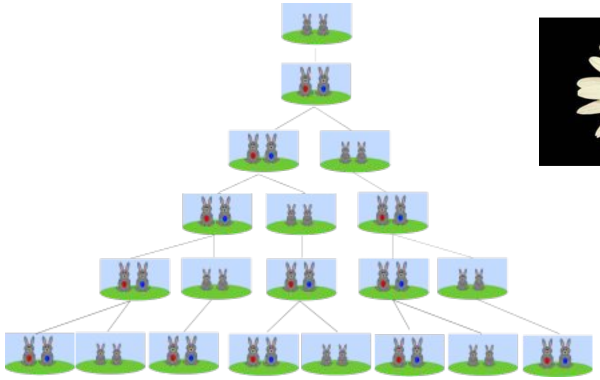
Example: Fibonacci numbers

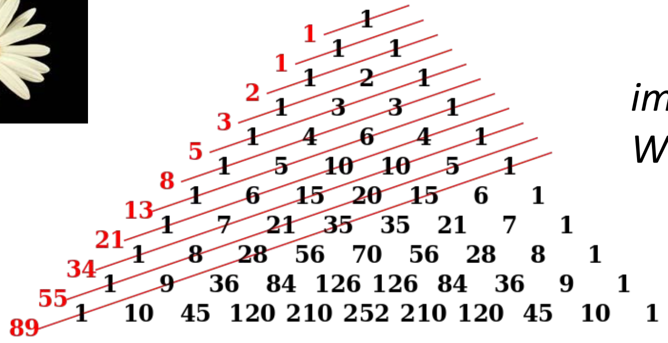
0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, etc.



8 13 21







images from Wikipedia

Code for Fibonacci Numbers

The Fibonacci number pattern goes as follows:

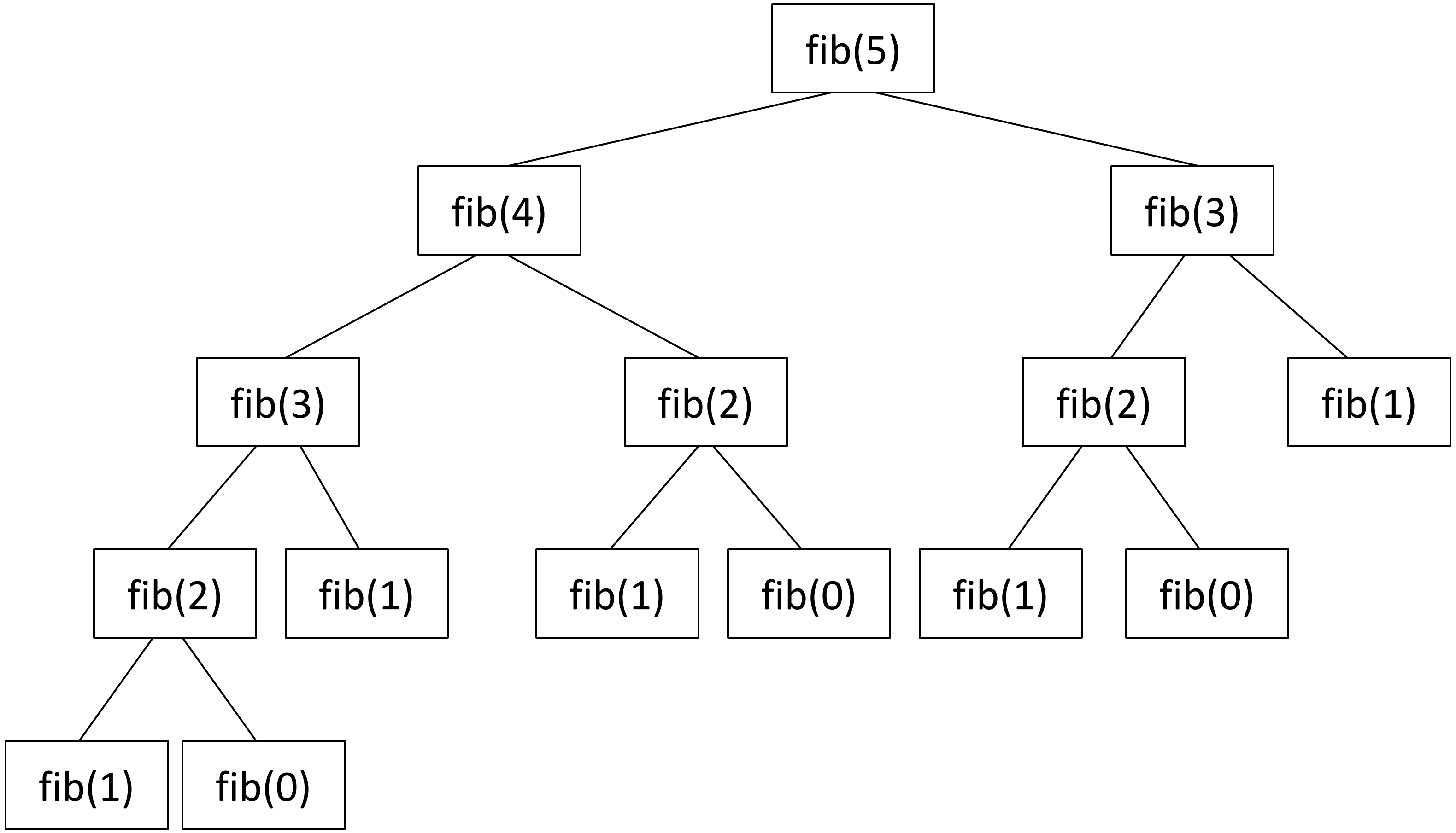
F(0) = 0
F(1) = 1
F(n) = F(n-1) + F(n-2), n > 1

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

Two recursive calls!

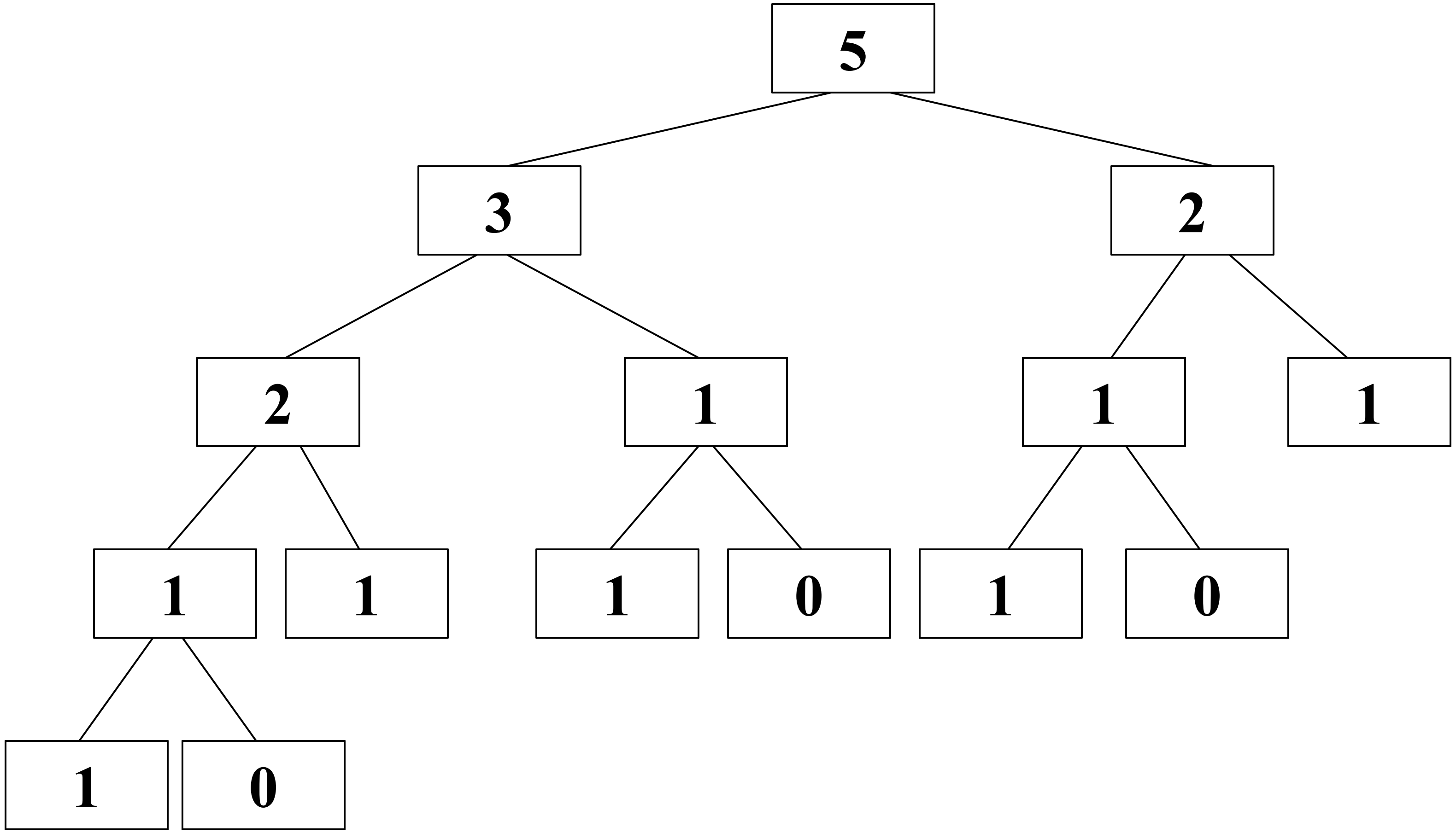
Fibonacci Recursive Call Tree

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, at a temple far away, priests were led to a courtyard with 64 discs stacked in size order on a sacred platform.

The priests need to move all 64 discs from this sacred platform to a second sacred platform, but there is only one other place (a third sacred platform) on which they can temporarily place the discs.

Priests can move only one disc at a time, because they’re heavy. And they may not put a larger disc on top of a smaller disc at any time, because the discs are fragile.

According to the legend, the world will end when the priests finish their work.

How long will this task take?

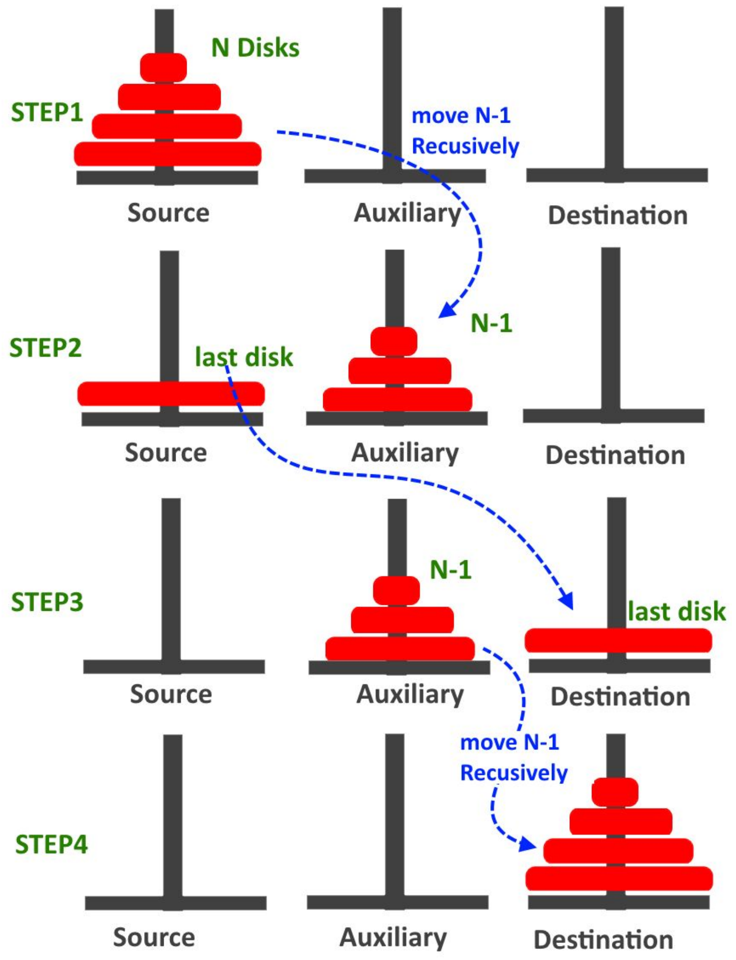
Solving Hanoi – Use Recursion!

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 peg.

Then, given N discs:

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



Solving Hanoi - Code

```python
# 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?

Discuss with your breakout group.

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.

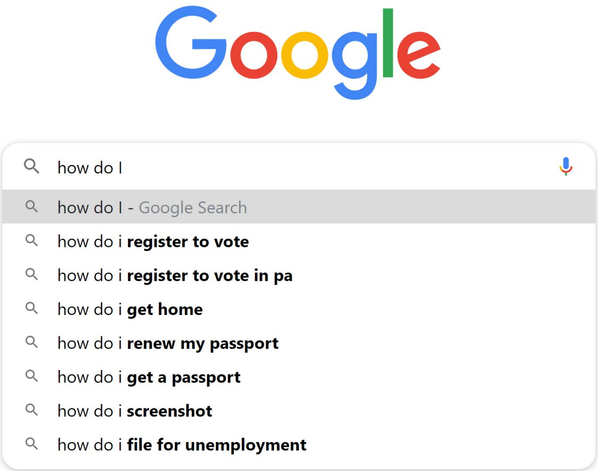
We can approximate the number of moves needed for the 64 discs in the story with 2^{64}. That's 1.84×10^{19} moves!

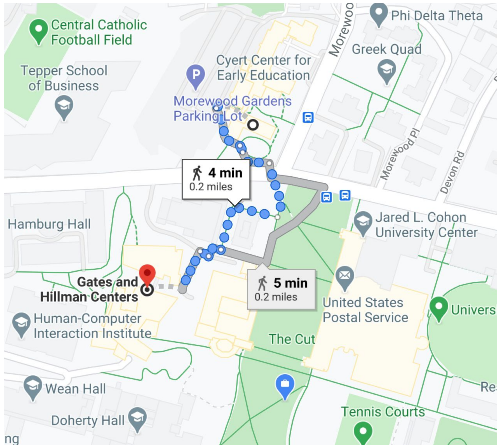
If we estimate each move takes one second, then that's $(1.84 \times 10^{19}) / (60*60*24*365) = 5.85 \times 10^{11}$ years, or **585 billion years!** We're safe for now.

Linear Search

Searching for Items

Search is one of the most common tasks a computer needs to do. We'll discuss it in depth this week and will revisit the concept several more times in this unit.





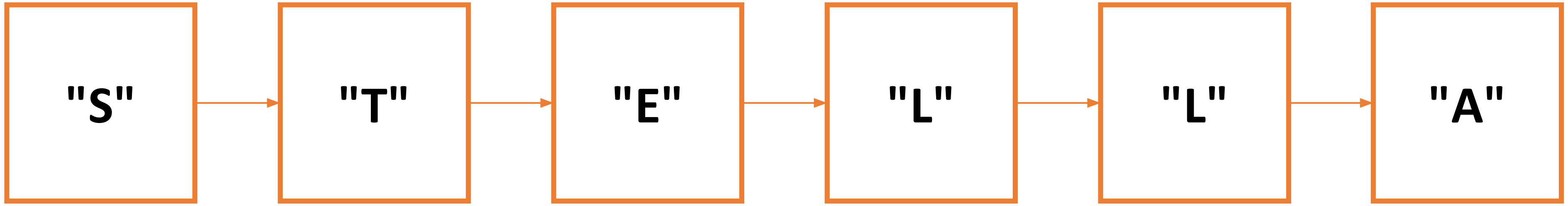
Suppose we want to determine whether a list contains a specific value. We know that the `in` operator can check this for us, but what algorithm does `in` implement?

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

How Computers See Lists

If we ask a computer to check if a value is in a list, it sees the whole list as a series of not-yet-known values:

In order to determine if the value is one of them, it needs to check each item in turn.



For-Each Search Function

We can use a for loop to implement this approach as code. We call this **linear search**, because it searches all items in a linear order.

```
def linearSearch(lst, target):
    for item in lst:
        if item == target:
            return True
    return False
```

Note that we can return `True` as soon as we find the target value, but we can't return `False` until we've examined all the values.

Question: If `target` appears more than once in `lst`, which instance will cause the function to return?

Sidebar: Check-Any and Check-All Patterns

Search follows a common pattern for functions that use a loop to return a Boolean.

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

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

```
def checkAny(lst, target):
    for item in lst:
        if item == target:
            return True
    return False
```

```
def checkAll(lst, target):
    for item in lst:
        if item != target:
            return False
    return True
```

Recursive Linear Search Algorithm

What are the **base cases** 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 make the problem **smaller**?

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

How do we **combine** the solutions?

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

Recursive Linear Search Code

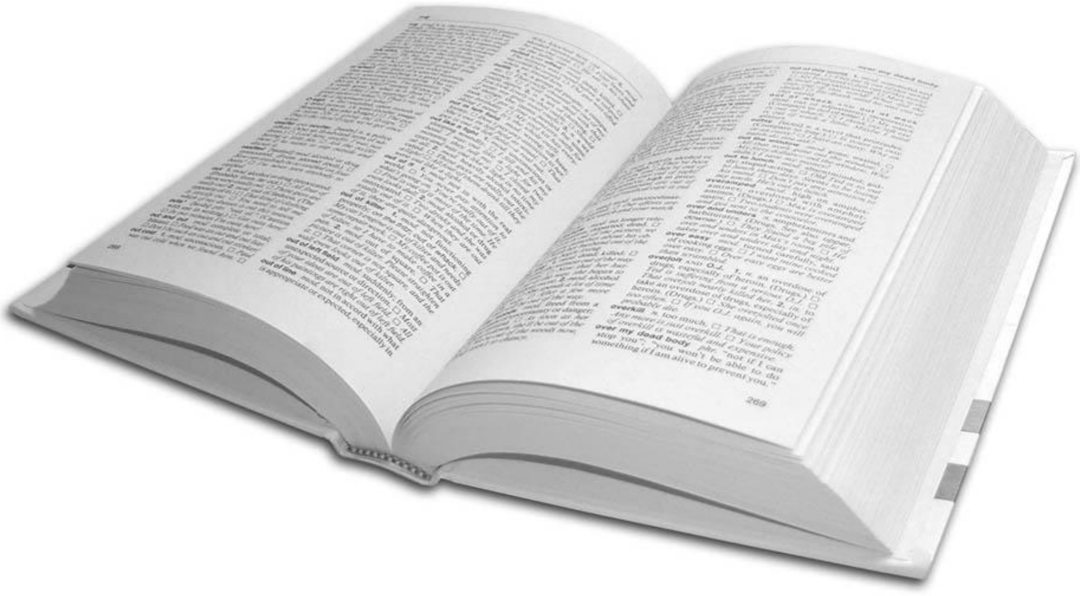
```
def recursiveLinearSearch(lst, target):
    if lst == [ ]:
        return False
    elif lst[0] == target:
        return True
    else:
        return recursiveLinearSearch(lst[1:], target)

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...

Start here

2	5	10	20	42	56	67	76	89	95

In **Binary Search** on a <u>**sorted list**</u>, 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...

Start here

2	5	10	20	42	56	67	76	89	95

Many more #s have been eliminated!

Algorithm for Binary Search

Algorithm for Binary Search:

1. Find the middle element of the list.
2. Compare the middle element to the target.
   a) If they're equal – you're done!
   b) If the item is **smaller** – recursively search to the **left** of the middle.
   c) If the item is **bigger** – recursively search to the **right** of the middle.

Example 1: Search for 73

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
12	25	32	37	41	48	58	60	66	73	74	79	83	91	95

Example 1: Search for 73

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
12	25	32	37	41	48	58	60	66	73	74	79	83	91	95

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
~~12~~	~~25~~	~~32~~	~~37~~	~~41~~	~~48~~	~~58~~	~~60~~	66	73	74	79	83	91	95

Example 1: Search for 73

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
12	25	32	37	41	48	58	60	66	73	74	79	83	91	95

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
~~12~~	~~25~~	~~32~~	~~37~~	~~41~~	~~48~~	~~58~~	~~60~~	66	73	74	79	83	91	95

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
~~12~~	~~25~~	~~32~~	~~37~~	~~41~~	~~48~~	~~58~~	~~60~~	66	73	74	~~79~~	~~83~~	~~91~~	~~95~~

Found: return True

Example 2: Search for 42

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
12	25	32	37	41	48	58	60	66	73	74	79	83	91	95

Example 2: Search for 42

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
12	25	32	37	41	48	58	60	66	73	74	79	83	91	95

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
12	25	32	37	41	48	58	~~60~~	~~66~~	~~73~~	~~74~~	~~79~~	~~83~~	~~91~~	~~95~~

Example 2: Search for 42

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
12	25	32	37	41	48	58	60	66	73	74	79	83	91	95

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
12	25	32	37	41	48	58	~~60~~	~~66~~	~~73~~	~~74~~	~~79~~	~~83~~	~~91~~	~~95~~

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
~~12~~	~~25~~	~~32~~	~~37~~	41	48	58	~~60~~	~~66~~	~~73~~	~~74~~	~~79~~	~~83~~	~~91~~	~~95~~

Example 2: Search for 42

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
12	25	32	37	41	48	58	60	66	73	74	79	83	91	95

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
12	25	32	37	41	48	58	~~60~~	~~66~~	~~73~~	~~74~~	~~79~~	~~83~~	~~91~~	~~95~~

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
~~12~~	~~25~~	~~32~~	~~37~~	41	48	58	~~60~~	~~66~~	~~73~~	~~74~~	~~79~~	~~83~~	~~91~~	~~95~~

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
~~12~~	~~25~~	~~32~~	~~37~~	41	~~48~~	~~58~~	~~60~~	~~66~~	~~73~~	~~74~~	~~79~~	~~83~~	~~91~~	~~95~~

Example 2: Search for 42

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
12	25	32	37	41	48	58	60	66	73	74	79	83	91	95

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
12	25	32	37	41	48	58	~~60~~	~~66~~	~~73~~	~~74~~	~~79~~	~~83~~	~~91~~	~~95~~

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
~~12~~	~~25~~	~~32~~	~~37~~	41	48	58	~~60~~	~~66~~	~~73~~	~~74~~	~~79~~	~~83~~	~~91~~	~~95~~

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
~~12~~	~~25~~	~~32~~	~~37~~	41	~~48~~	~~58~~	~~60~~	~~66~~	~~73~~	~~74~~	~~79~~	~~83~~	~~91~~	~~95~~

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
~~12~~	~~25~~	~~32~~	~~37~~	~~41~~	~~48~~	~~58~~	~~60~~	~~66~~	~~73~~	~~74~~	~~79~~	~~83~~	~~91~~	~~95~~

Not found: return False

Activity: Trace Binary Search

You do: work with your breakout group to determine the correct trace for the following call to binary search:

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

Base and Recursive Cases for Binary Search

What's are the **base cases** for binary search?

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

Answer: if we find the target! Then we can stop searching and immediately return `True`.

How do we make the problem **smaller**? What are the **recursive cases**?

Answer: call the binary search on one half of the list. Which half?

How do we **combine** the solutions?

Answer: We don’t have 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 ____: # first base case
        return _____
    mid = _____ # Calculate index of the middle element of the list.
    if ____:    # second base case
        return _____
    else:
        # Compare middle element to the target.
            # If the middle is smaller, recursively search
            #    to the left of the middle.
            # Else the middle is larger, so recursively search
            #    to the right of the middle.
```

Binary Search in Code – Base Case

The first **base case** is the empty list, for which we should `return False`

```
def binarySearch(lst, target):
    if lst == [ ]:
        return False
    mid = _____ # Calculate index of the middle element of the list.
    if _____:   # second base case
        return _____
    else:
        # Compare middle element to the target.
            # If the middle is smaller, recursively search
            #    to the left of the middle.
            # Else the middle is larger, so recursively search
            #    to the right of the middle.
```

Binary Search – Middle Element

To get the index of the middle element, use take half the length of the list.

```
def binarySearch(lst, target):
    if lst == [ ]:
        return False
    mid = len(lst) // 2
    if _____:   # second base case
        return _____
    else:
        # Compare middle element to the target.
            # If the middle is smaller, recursively search
            #    to the left of the middle.
            # Else the middle is larger, so recursively search
            #    to the right of the middle.
```

Use integer division, in case the list has an odd length

Binary Search – Comparison

Second base case: the middle element matches the target.

```
def binarySearch(lst, target):
    if lst == [ ]:
        return False
    mid = len(lst) // 2
    if lst[mid] == target:
        return _______
    else:
        # Compare middle element to the target.
            # If the middle is smaller, recursively search
            #    to the left of the middle.
            # Else the middle is larger, so recursively search
            #    to the right of the middle.
```

Binary Search – Comparison

For the second base case, return `True`.

```
def binarySearch(lst, target):
    if lst == [ ]:
        return False
    mid = len(lst) // 2
    if lst[mid] == target:
        return True
    else:
        # Compare middle element to the target.
            # If the middle is smaller, recursively search
            #    to the left of the middle.
            # Else the middle is larger, so recursively search
            #    to the right of the middle.
```

Binary Search – Comparison

Use an `elif` and `else` statements to do the larger/smaller comparison.

```
def binarySearch(lst, target):
    if lst == [ ]:
        return False
    mid = len(lst) // 2
    if lst[mid] == target:
            return True
    elif target < lst[mid]:

            ________
    else: # lst[mid] < target

            ________
```

Binary Search – Results

Use **slicing** to write the two recursive calls. We will only make one of them.

```
def binarySearch(lst, target):
    if lst == [ ]:
        return False
    mid = len(lst) // 2
    if lst[mid] == target:
        return True
    elif target < lst[mid]:
        return binarySearch(lst[:mid], target) # recurse on left half
    else: # lst[mid] < target
        return binarySearch(lst[mid+1:], target) # recurse on right half
```

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.

In the next class, we'll introduce a way to compare the efficiency of algorithms more formally.

But note, binary search only works on **sorted** lists!

Case Study: Facebook

When you login to Facebook, it has to look up your username.

How long does that take?

- Facebook has 2.7 billion active users.
- Assume it can compare a username to a target string in 1 microsecond (one millionth of a second).
- Assume, on average, that linear search has to go half-way down the list to find a user name.
- How long will it take to log in to Facebook?

Linear search: 2.7e+09 / (60*1,000,000) / 2 = 22.5 minutes

Binary search: $\log_2$(2.7e+09) / 1,000,000 = 31 microseconds

Learning Objectives

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

- Recognize **linear search** on lists and in recursive contexts

- Use **binary search** when reading and writing code to search for items in **sorted** lists

- **Feedback: https://bit.ly/110-feedback**