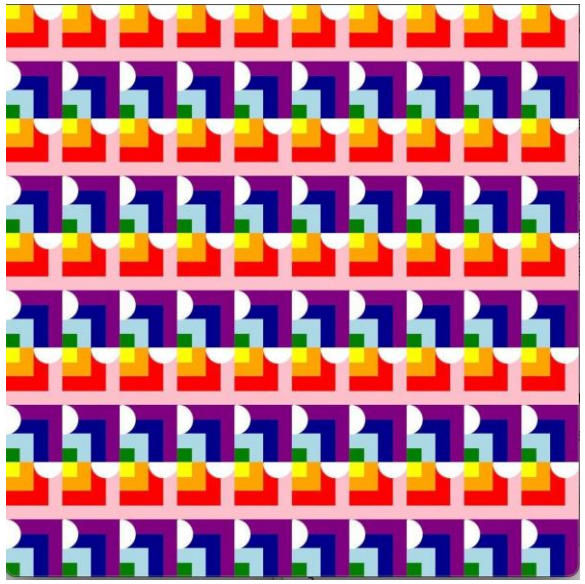
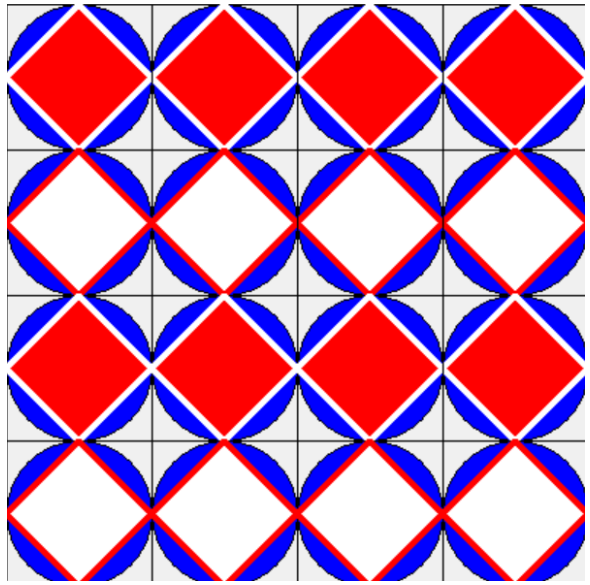
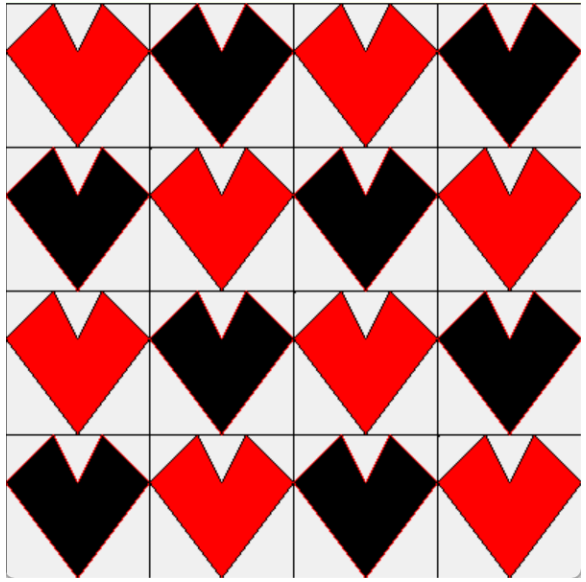
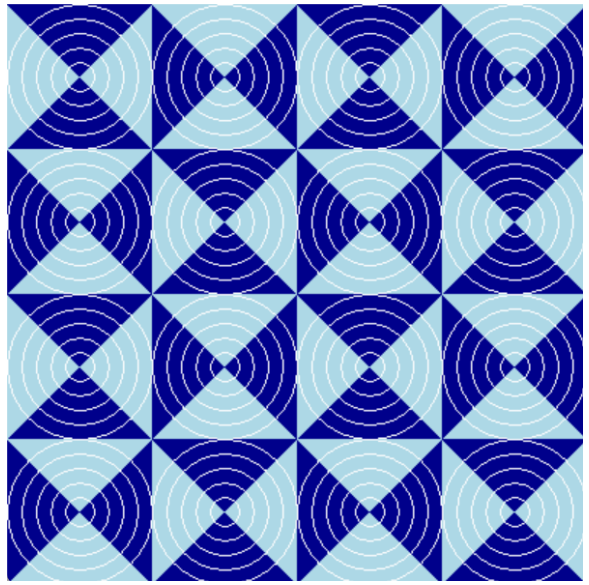
Awesome Patterns!







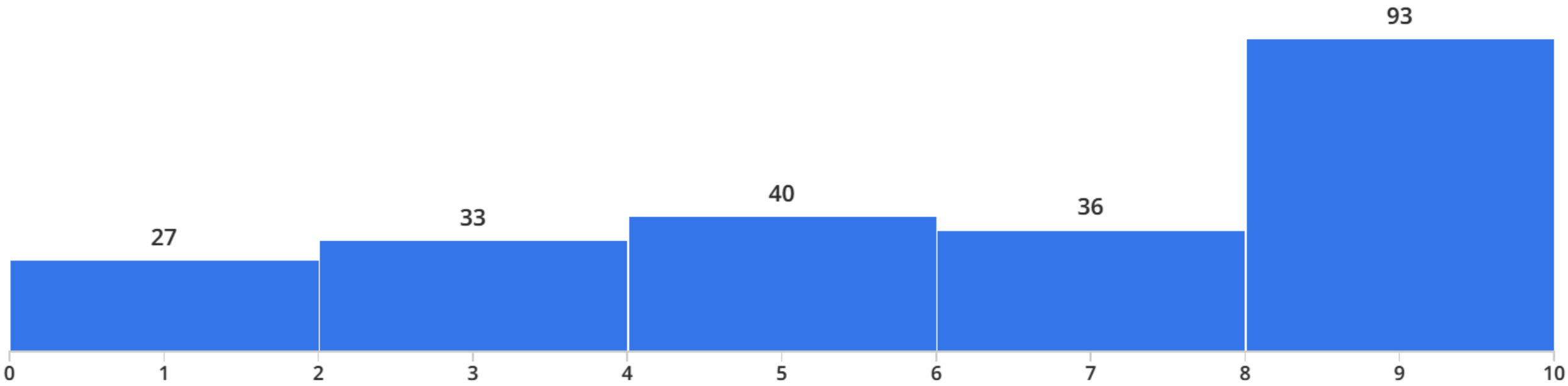


Recursion II & Search Algorithms

15-110 – Friday 09/26

Announcements

- Check3 due **Monday at noon**
- Quizlet2 average: 6.75/10.



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

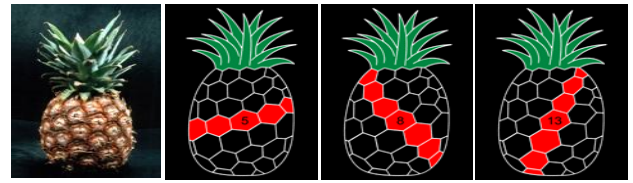
Multiple Recursive Calls

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

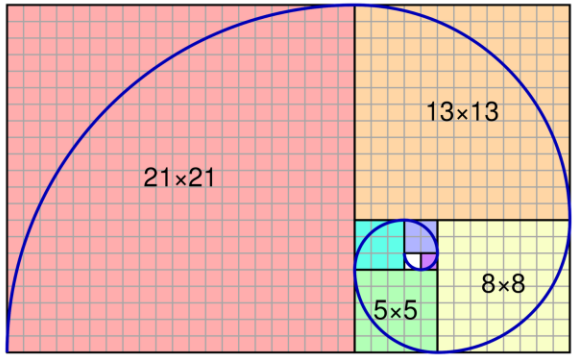
The real **conceptual** power of recursion happens when we need more than one recursive call!

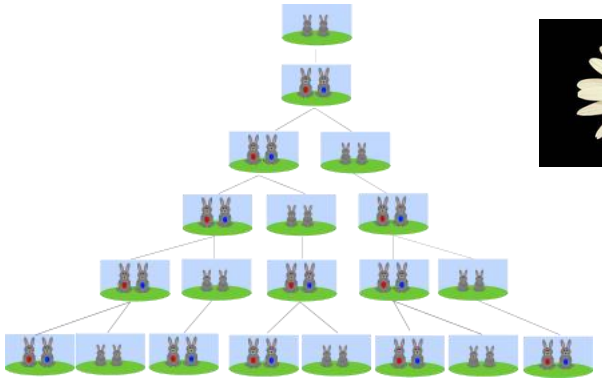
Example: Fibonacci numbers

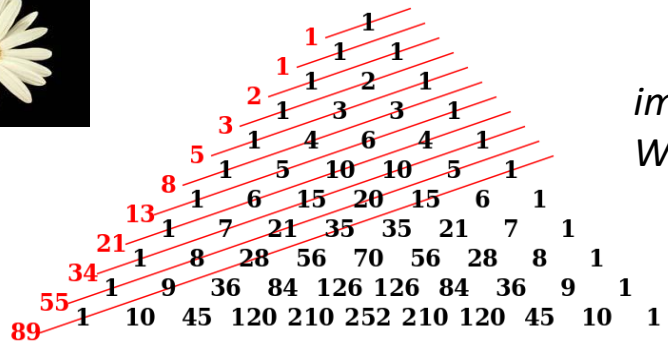
0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, etc.



8 13 21







images from Wikipedia

Code for Fibonacci Numbers

The Fibonacci number pattern goes as follows:

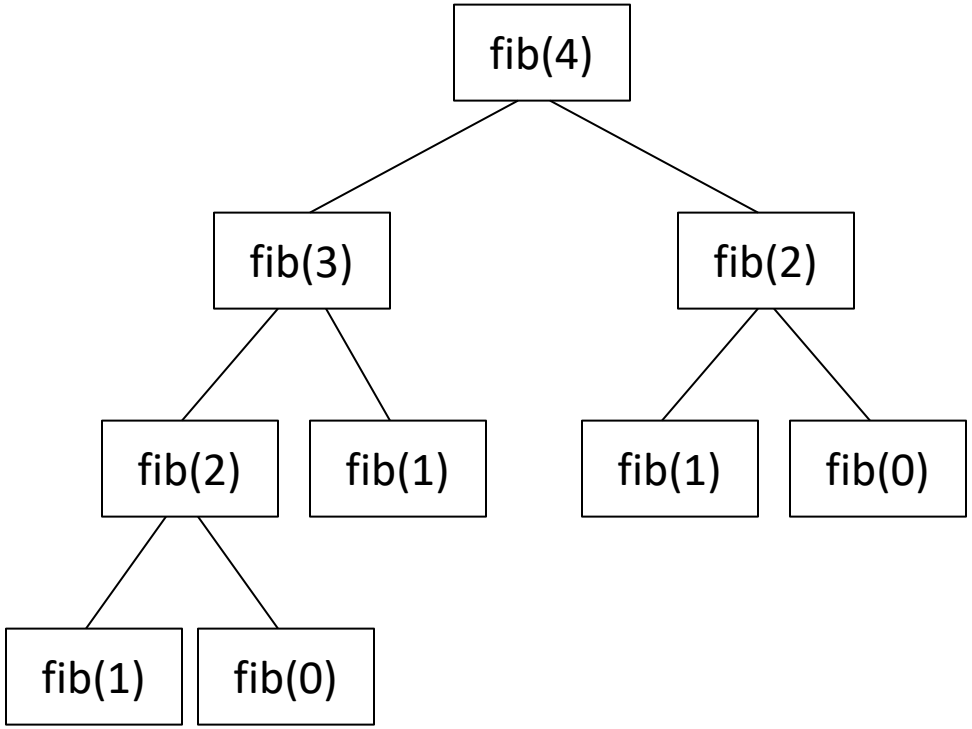
F(0) = 0
F(1) = 1
F(n) = F(n-1) + F(n-2), n > 1

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

Two recursive calls!

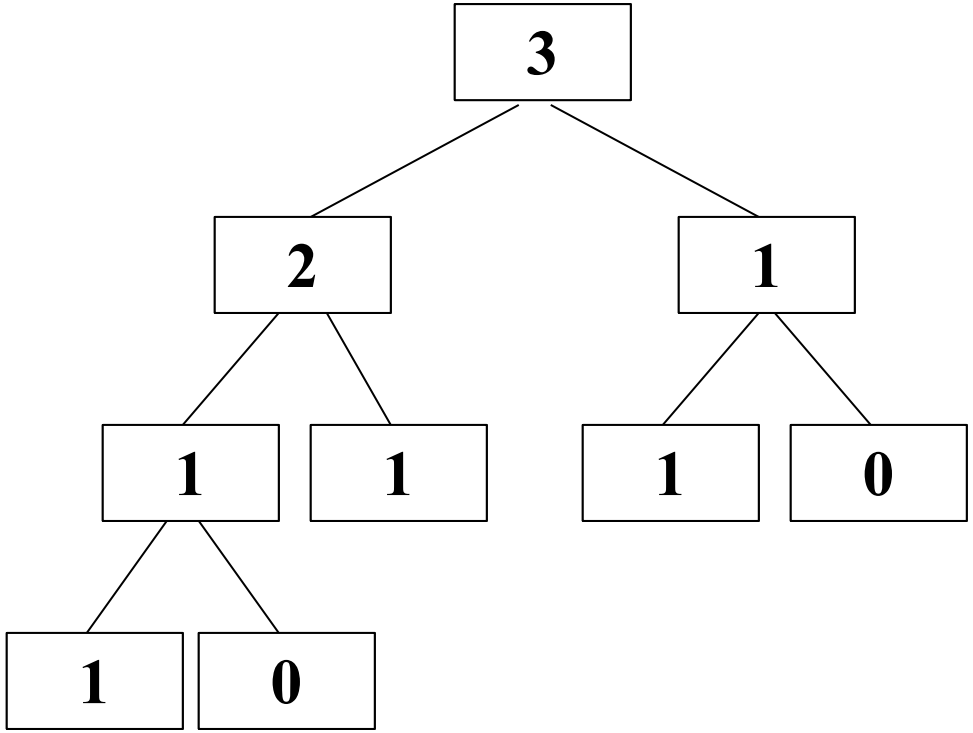
Fibonacci Recursive Call Tree

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



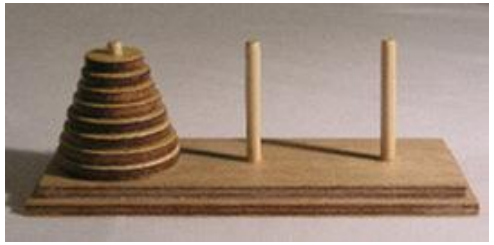
Fibonacci Recursive Call Tree

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



Another Example: Towers of Hanoi

Legend has it 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 the sacred platform to a sacred stage**, 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?

Try it yourself: https://www.mathsisfun.com/games/towerofhanoi.html

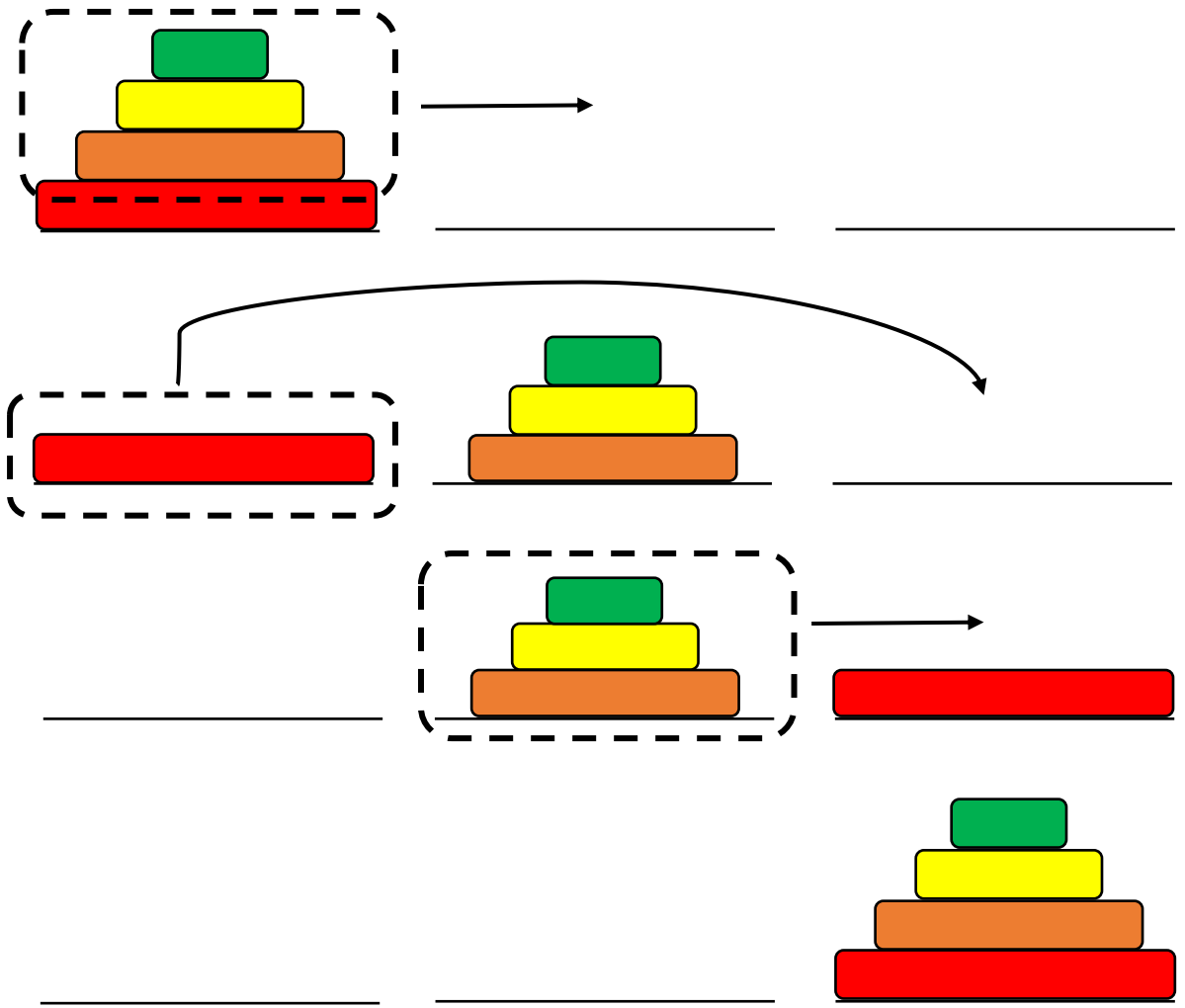
Solving Hanoi – Use Recursion!

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. 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: # 2+ discs – use recursion
        # Move all but the largest to tmp
        moveDiscs(start, end, tmp, discs - 1)

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

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

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

Note that the roles of the left, middle, right positions **change** as we move discs around. A peg that is initially 'start' might become 'tmp' later on.

Activity: Towers of Hanoi Steps

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

Thinking about 64 discs is hard! Let's rephrase the question to make it easier to answer.

You do: 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 approximately **doubles** the number of moves we make.

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

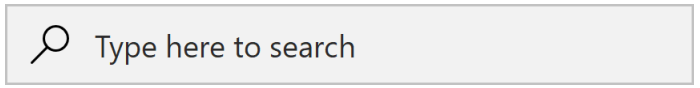
We can approximate the number of moves needed for the 64 discs in the story with 2^{64}. That's 1.84×10^{19} moves!

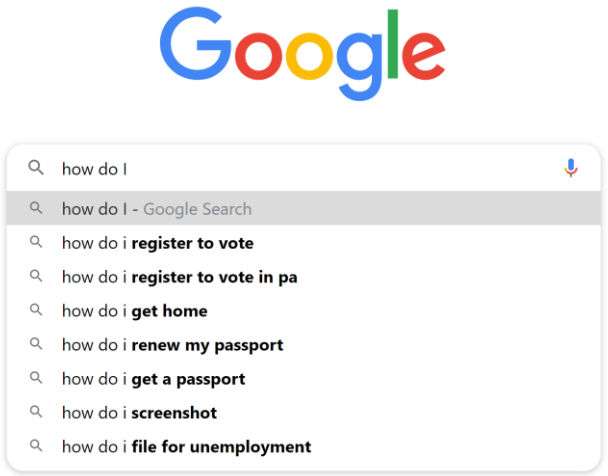
If we estimate each move takes one second, then that's $(1.84 \times 10^{19}) / (60*60*24*365) = 5.85 \times 10^{11}$ years, or **585 billion years!** We're safe for now.

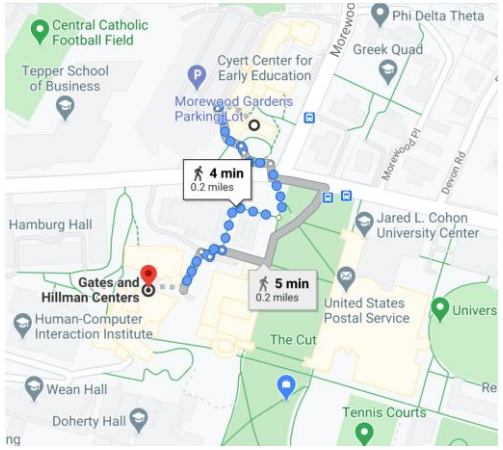
Linear Search

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.







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.

Implementing Search

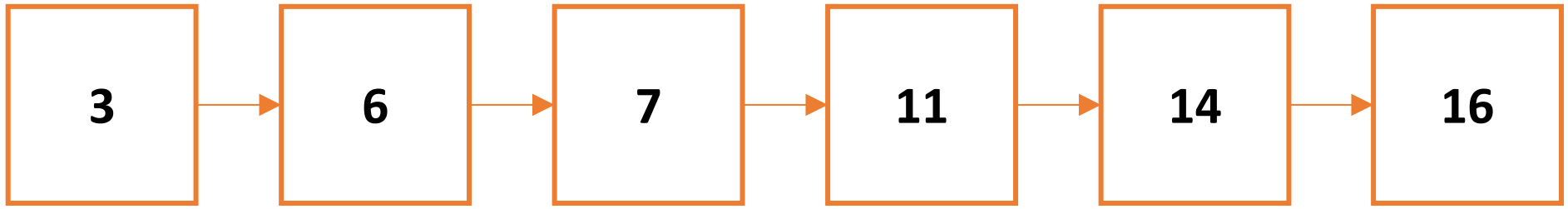
Suppose we want to determine whether a list contains a specific value. We know that the `in` operator can check this for us, but what algorithm does `in` implement?

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.



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.

For Loop 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 i in range(len(lst)):
        if lst[i] == 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.

You do: If `target` appears more than once in `lst`, which value 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 i in range(len(lst)):
        if lst[i] == target:
            return True
    return False
```

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

Recursive Linear Search Algorithm

Let's implement linear search recursively, just to practice.

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 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 necessary. The recursive call returns whether the item occurs in the rest of the list; just return that result unmodified.

Recursive Linear Search Code

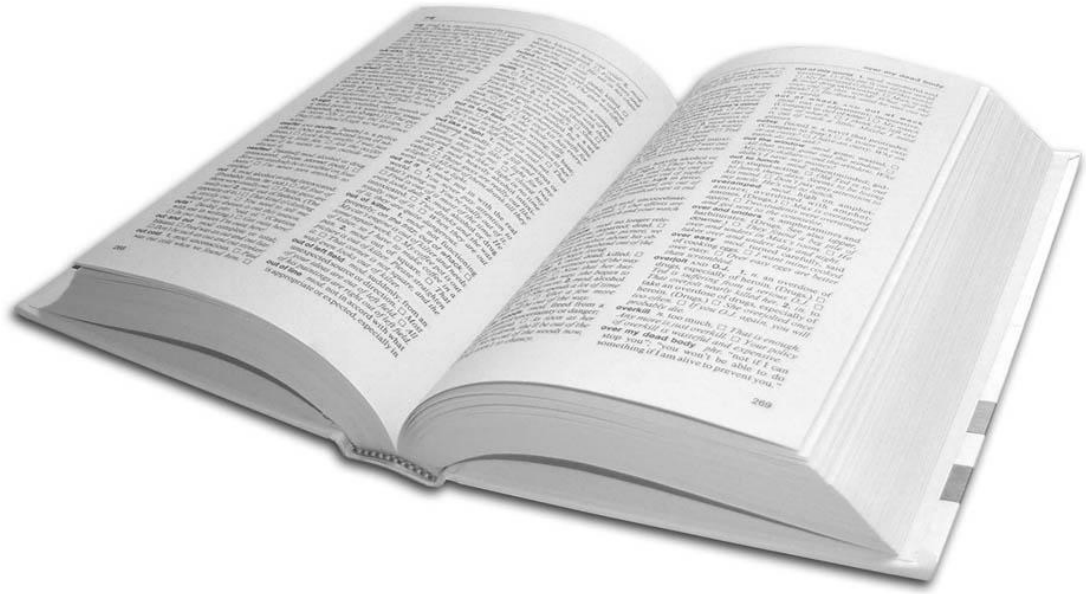
```
def recursiveLinearSearch(lst, target):
    if lst == [ ]:
        return False
    elif lst[0] == target:
        return True
    else:
        return recursiveLinearSearch(lst[1:], target)

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?

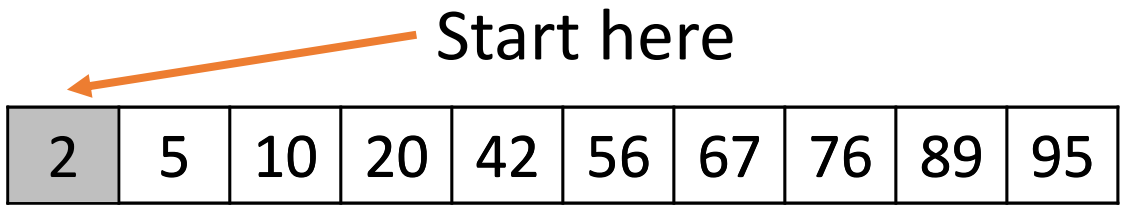


Can we take advantage of dictionaries being **sorted**?

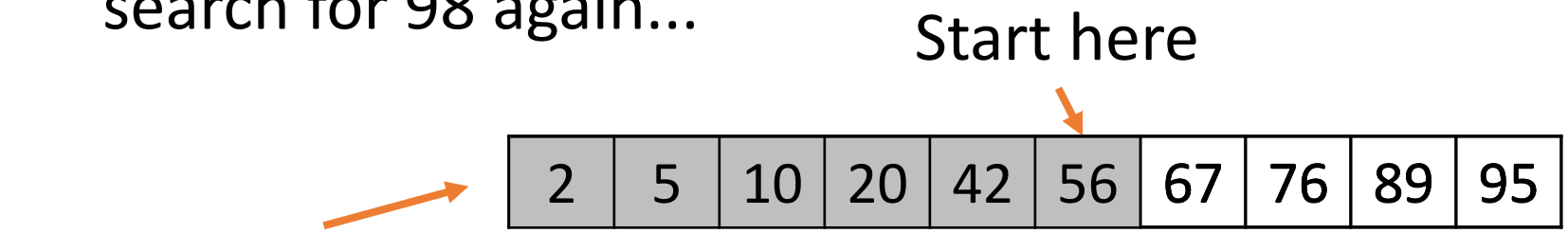
Binary Search

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...



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!

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 target is **smaller** – recursively search to the **left** of the middle.
   c) If the target is **bigger** – recursively search to the **right** of the middle.

What if there are an even number of elements? We'll break ties to the right.

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

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: 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)
```

Base Case and Recursive Case of Binary Search

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 make the problem **smaller**?

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

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 target is smaller, recursively search
            #    to the left of the middle.
            # If the target 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 target is smaller, recursively search
            #    to the left of the middle.
            # If the target 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 target is smaller, recursively search
            #    to the left of the middle.
            # If the target 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 target is smaller, recursively search
        #    to the left of the middle.
        # If the target 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 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