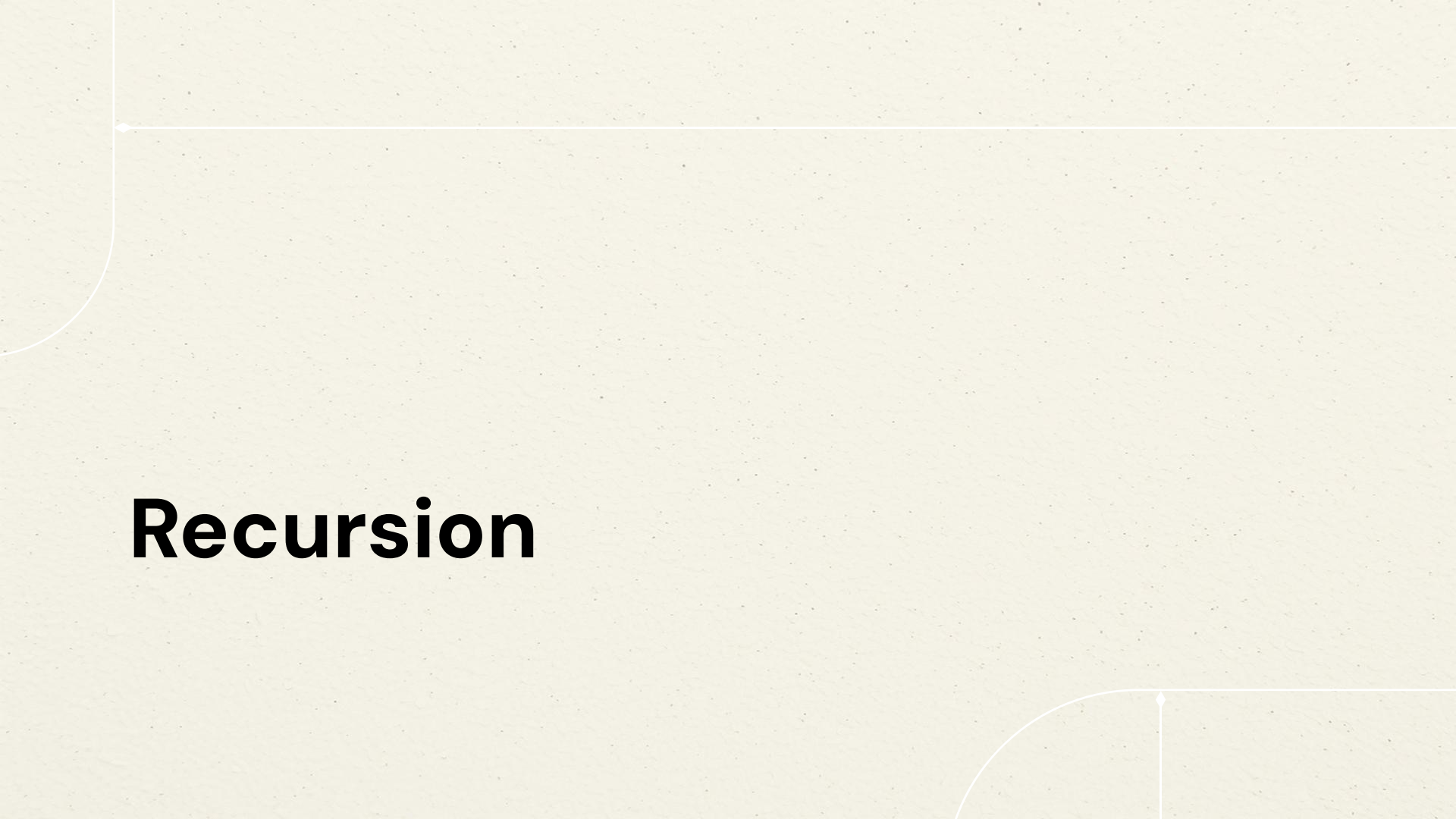


15-110

Final Exam Review



Recursion

How Does Recursion Work

- Make the problem smaller
- Get the smaller problem's solution
 - Pretend that recursion automatically solves the problem correctly!
- Combine leftover solution with smaller problem's solution

Try it: Finding the sum of a list of ints

8	2	6	9	16	35	21
---	---	---	---	----	----	----

How Does Recursion Work

- **Make the problem smaller**
- Get the smaller problem's solution
 - Pretend that recursion automatically solves the problem correctly!
- Combine leftover solution with smaller problem's solution

Try it: Finding the sum of a list of ints

8	2	6	9	16	35	21
---	---	---	---	----	----	----

1. Smaller problem: `lst[1:]`

8	2	6	9	16	35	21
---	---	---	---	----	----	----

How Does Recursion Work

- Make the problem smaller
- Get the smaller problem's solution
 - Pretend that recursion automatically solves the problem correctly!
- **Combine leftover solution with smaller problem's solution**

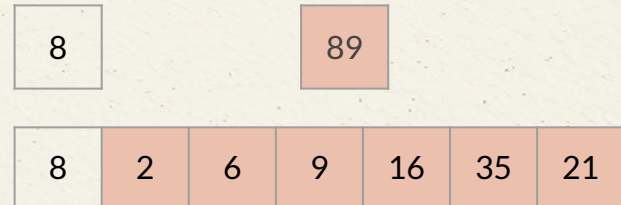
Try it: Finding the sum of a list of ints

1. Smaller problem: `lst[1:]`

2. $2 + 6 + 9 + 16 + 35 + 21 = 89$

`getSum(lst[1:])` #Smaller Result

3. $8 + 89 = 97 \rightarrow \text{lst}[0] + \text{getSum}(\text{lst}[1:])$



Things to Remember

Things To Remember:

1. You have to call the function within its own body for it to be recursion
 - a. Make sure you're making the problem smaller when you're calling it
 - i. Otherwise it'll go on forever
2. Your return types have to match!
 - a. Your base can't return an int while your recursive case returns a list

Recursion Practice

Write the function **fixPalindrome** that takes in a string **s** and returns **s** if **s** is a palindrome or returns **s** with all of its mismatching letters converted into 0s to make it into a palindrome.

Examples

fixPalindrome("arra") returns "arra"

fixPalindrome("5660") returns "0660"

fixPalindrome("iuprk") returns "00p00"

Recursion Practice SOLUTION

Write the function **fixPalindrome** that takes in a string **s** and returns **s** if **s** is a palindrome or returns **s** with all of its mismatching letters converted into 0s to make it into a palindrome.

```
def fixPalindrome(s):  
    if len(s) <= 1:  
        return s  
    else:  
        smallerProblem = s[1:len(s)-1]  
        smallerResult = fixPalindrome(smallerProblem)  
        if s[0] == s[len(s)-1]:  
            return s[0] + smallerResult + s[len(s)-1]  
        else:  
            return "0" + smallerResult + "0"
```

Recursion Code Tracing Practice

What will this function return when given `print(f([1,2,3], [4,5,6]))` ?

```
def f(a, b):
```

```
    if a == []:
```

```
        return []
```

```
    else:
```

```
        return [a[0]] + [b[0]] + f(a[1:], b[1:])
```

Recursion Code Tracing SOLUTION

What will this function return when given `print(f([1,2,3], [4,5,6]))` ?

```
def f(a, b):
```

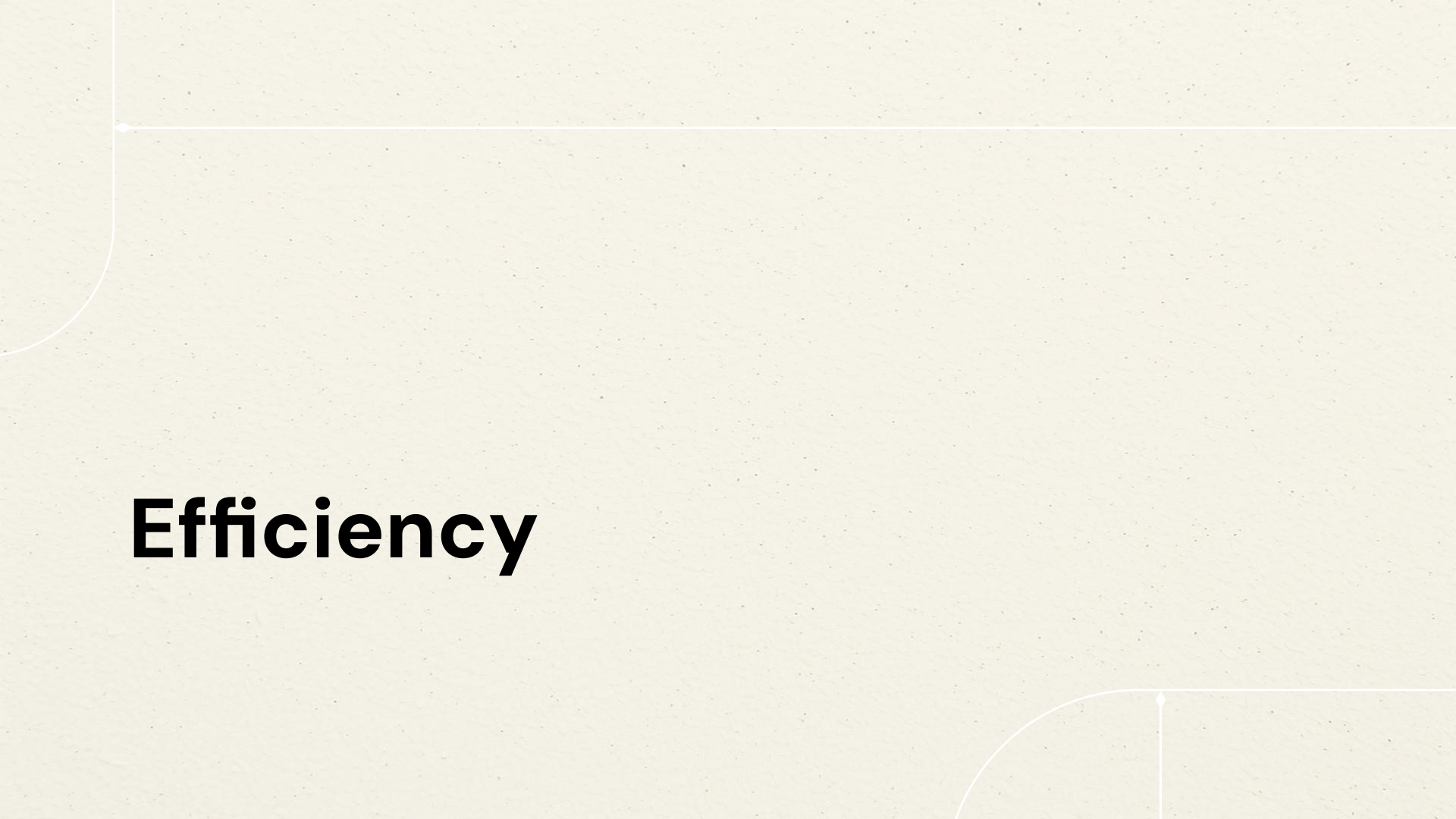
```
    if a == []:
```

```
        return []
```

```
    else:
```

```
        return [a[0]] + [b[0]] + f(a[1:], b[1:])
```

[1,4,2,5,3,6]

The background is a light beige, textured surface. A thin white line starts from the left edge, curves down, and then extends horizontally across the top. A small white arrowhead points to the right at the end of this horizontal line. Another thin white line starts from the bottom right, curves up, and then extends horizontally across the bottom. A small white arrowhead points to the left at the end of this horizontal line.

Efficiency

What is Big-O

- Runtime it takes to execute a program based on the input
- Simplest and tightest bound
 - $O(2n + 6) \rightarrow O(n)$
 - $O(n^2 + n) \rightarrow O(n^2)$
 - $O(n^2 \log n + 4n + 3) \rightarrow ?$
- Some polynomial Big Os: $O(1)$, $O(\log n)$, $O(n)$, $O(n^2)$, $O(n^k)$
- Some non-polynomial Big Os: $O(n!)$, $O(2^n)$, $O(k^n)$

Determining Big-O

- Determine the runtime for **each line** of the function
 - Common $O(1)$ lines: print, return, comparing with $<$ or $>$, initializing or updating variables
 - Common $O(n)$: in, .index()
 - Be careful of runtimes of built-in functions or methods like L.pop, L.remove...
 - For **loops**:
 - Identify how many the times the loop iterates
 - Identify the runtime for each line inside the loop
 - **Multiply** the number of times the loop iterates by the longest time in the loop
- Determine the runtime of the entire function... by **adding consecutive steps** and **simplifying** to the simplest and tightest bound

Big-O Practice

```
def mystery(num):  
    i = 0  
    result = 0  
    while i <= num: #iterates n times  
        j = 0  
        while j <= i: # 0, 1, 2, 3 ,4,...n  
            result += 1  
            j = j + 1  
        k = 0  
        while k <= 10: O(1)  
            k = k+1  
            result += 1  
        i += 1  
    return result
```

Big-O Practice SOLUTION

def mystery(num):	#n = num	
i = 0	O(1)	
result = 0	O(1)	
while i <= num:	O(n)	
j = 0	O(1)	
while j <= i:	O(n)	
result += 1	O(1)	
j = j + 1	O(1)	
k = 0	O(1)	
while k <= 10:	O(1)	
result += 1	O(1)	
i += 1	O(1)	
return result	O(1)	

Overall: ?

Big-O Practice SOLUTION

def mystery(num):	#n = num	
i = 0	O(1)	
result = 0	O(1)	
while i <= num:	O(n)	
j = 0	O(1)	
while j <= i:	O(n)	
result += 1	O(1)	
j = j + 1	O(1)	
k = 0	O(1)	
while k <= 10:	O(1)	
result += 1	O(1)	
i += 1	O(1)	
return result	O(1)	
		Overall: $O(n^2)$

Big-O Practice

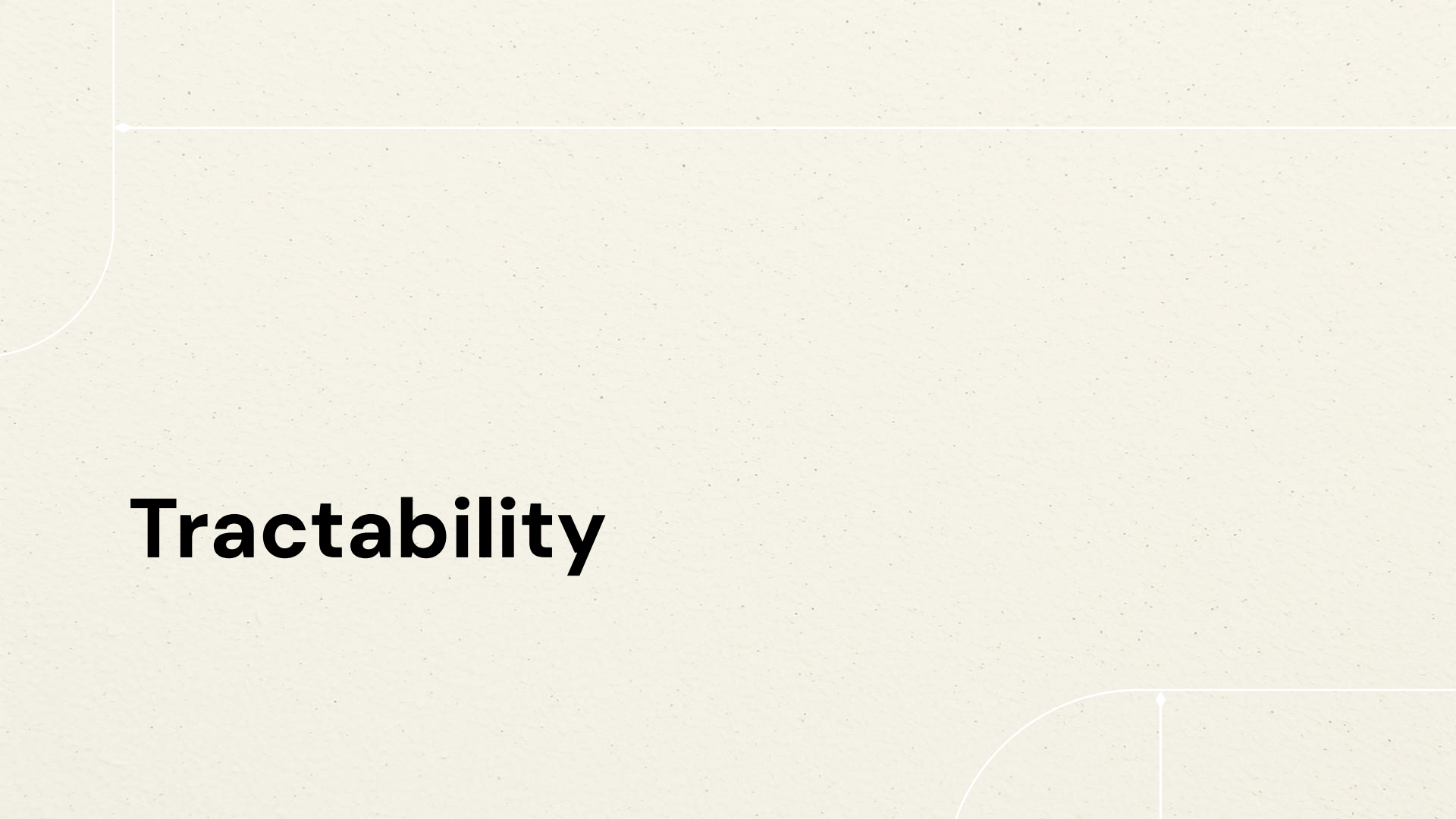
```
# input size = n
def powersOfTwo(n):
    m = 1
    while m <= n:
        print(m)
        m = m * 2
```

Big-O Practice SOLUTION

```
# input size = n
def powersOfTwo(n):
    m = 1
    while m <= n:
        print(m)
        m = m * 2
```

$O(\log n)$

Starting at 1, we multiply our current power by 2 and output until we exceed n - we only iterate $\log_2 n$ times

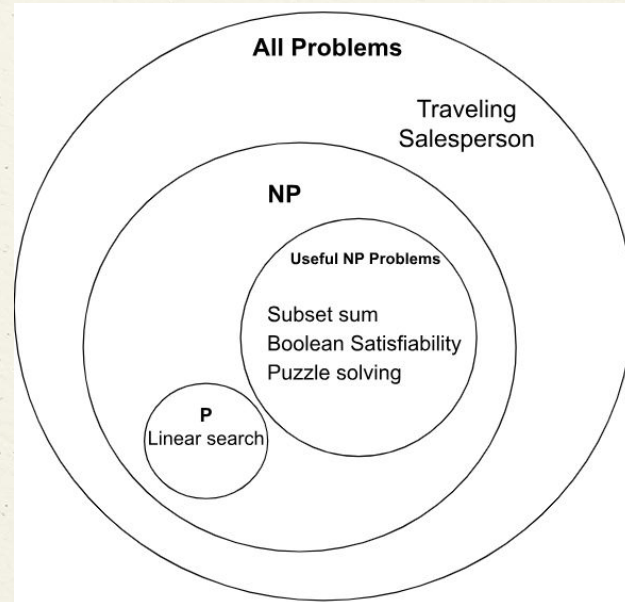


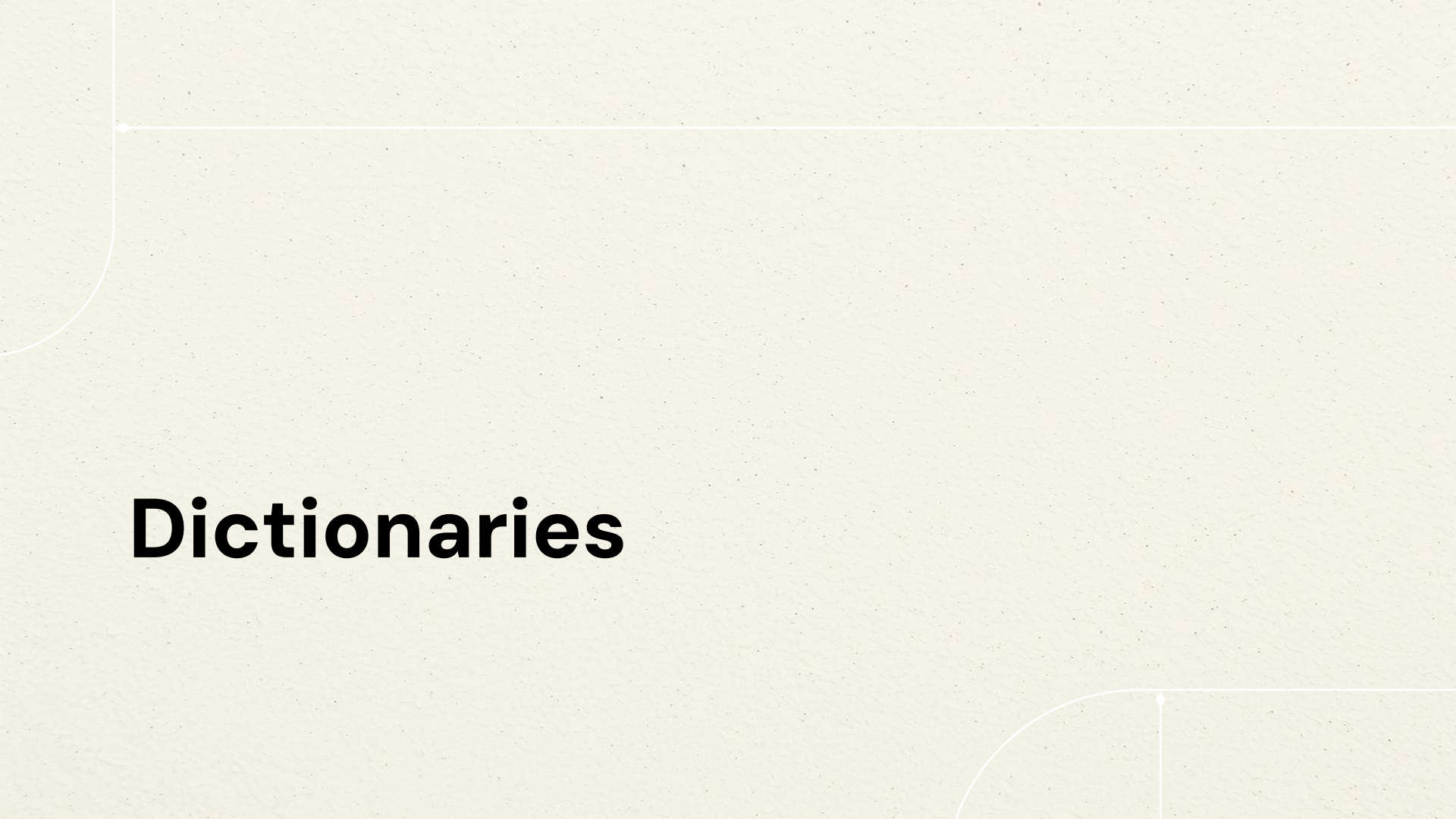
Tractability

P vs NP

- Tractable functions can be run in a reasonably efficient amount of time
 - Polynomial time - $O(n^k)$ or less!
- Intractable functions can not be run in a reasonably efficient amount of time
 - $O(2^n)$, $O(n!)$
- Problems are classified as P or NP based on their tractability in solving and verifying them
 - Proving any useful NP problem has a tractable solution proves all useful NP problems are tractable (and \$\$\$)

	P	NP
Verifying	Tractable	Tractable
Solving	Tractable	?



The background is a light beige, textured surface. A thin white line runs horizontally across the top, with a small white arrow pointing left at its left end. Another thin white line runs horizontally across the bottom, with a small white arrow pointing up at its right end. A curved white line is visible on the left side, and another curved white line is visible on the bottom right side.

Dictionary

Dictionaries

- Data structure storing data by **mapping keys to values**
 - **Keys** are unordered and must be unique and immutable
 - Values can be any type
- Index with the key to access the value
- Dictionaries are mutable
 - Add a key-value pair: Use index assignment
 - Remove a key-value pair: Use `d.pop(key)`
- Looping over a dictionary
 - for key in d:

```
d = {'abc' : 1, 'hello' : [2, 4, 7], 26 : '15-110'}
```

Dictionaries

Write the function **getFavoriteCereal** that takes in **cerealDict**, a dictionary of cereal brands and the number of people that said it was their favorite cereal, and returns the most popular cereal.

We can assume that there will not be ties.

```
assert(getFavoriteCereal(cereals) == "lucky charms")
```

```
cereals = {"cheerios" : 17, "fruit loops" : 25, "grape nuts" : 1,  
          "lucky charms" : 28, "corn flakes" : 23}
```

Dictionaries - Solution

```
def getFavoriteCereal(cerealDict):  
    currentFave = ""  
    currentHighest = -1  
    for cereal in cerealDict:  
        if cerealDict[cereal] > currentHighest:  
            currentFave = cereal  
            currentHighest = cerealDict[cereal]  
    return currentFave
```

Dictionaries

Write a function **studyHours(subjects, hours)** that takes a list of subjects (subjects) and a list of hours spent studying that subject (hours), and returns a dictionary (hoursPerSubject) where the subject names are keys and the hours are values.

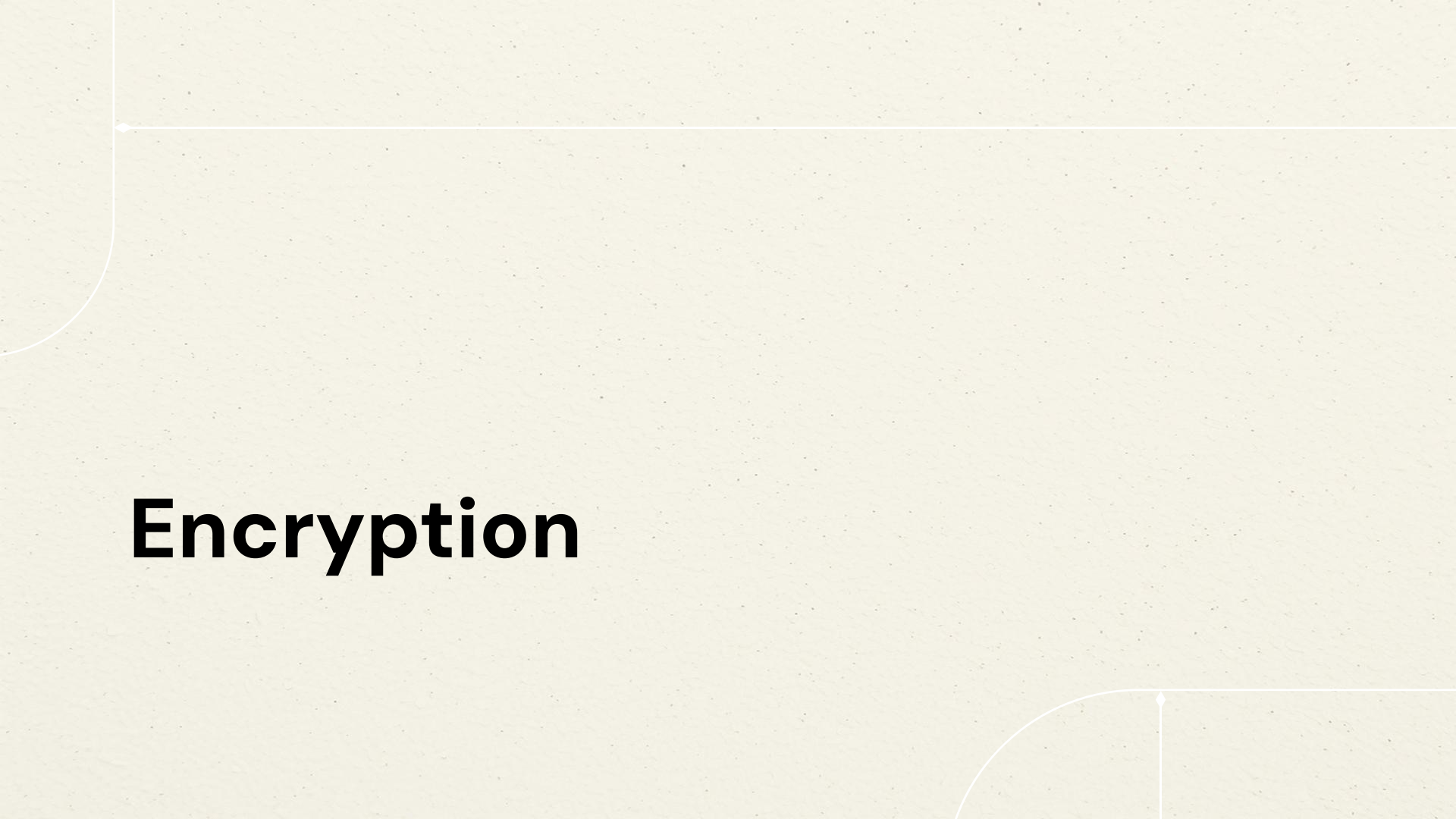
You're given that the indexes of the strings in subjects and integers in hours correspond to each other.

ex:

```
subject = ["15-110", "Black Holes", "Physics II", "Chemistry", "Business"]  
hours = [3, 1, 2, 2, 3]  
returns {"15-110" : 3, "Black Holes" : 1, "Physics II" : 2, "Chemistry" : 2, "Business" : 3}
```

Dictionaries - Solution

```
def studyHours(subjects, hours):  
    hoursPerSubject = {}  
    for i in range(len(subjects)):  
        subject = subjects[i]  
        timeSpent = hours[i]  
        hoursPerSubject[subject] = timeSpent  
    return hoursPerSubject
```

The background is a light beige, textured surface. A thin white line runs horizontally across the top, with a small white arrow pointing left at its left end. Another thin white line runs horizontally across the bottom, with a small white arrow pointing up at its right end. A curved white line is visible on the left side, and another curved white line is visible on the bottom right side.

Encryption

What is encryption? + How does it work?

- “Encoding” data so only the people involved (sender and receiver) can see its message
- This is a way we can protect data from people we don’t want to have access to it
- Need a **secret key** to prevent other people from finding out the message - tells the people involved how to encode and decode

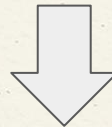
Two parts of encryption:

- Encoding
- Decoding

To Kate

From Alice

“Good luck on the final”



Encoding

Nvvk sbjr vu aol mpuhs

Nvvk sbjr vu aol mpuhs



Decoding

“Good luck on the final”

Types of Encryption

- Symmetric: the sender and receiver (ex. Alice and Kate) both share a key
 - Caesar Cipher is symmetric - shifts each letter in the message by a number between 0 and 25
 - Ex. The key tells us to shift by 2 letters. Alice shifts the original message by 2 letters and Kate receives this encoded message and shifts it by 2 to get the original message from Alice. They both use the same key that tells them to shift by 2.
- Asymmetric: the sender has their own key and the receiver has their own key
 - RSA encryption is asymmetric (**and super hard to break**)
 - Ex. We use the public key for encoding and private key for decoding. Alice uses the Alice's public key to encode her message. Kate uses the Kate's private key to decode the message.

Exercises

You've been tasked by a special task force to encode this message with the Caesar Cipher.

The key is shifting each letter by 1.

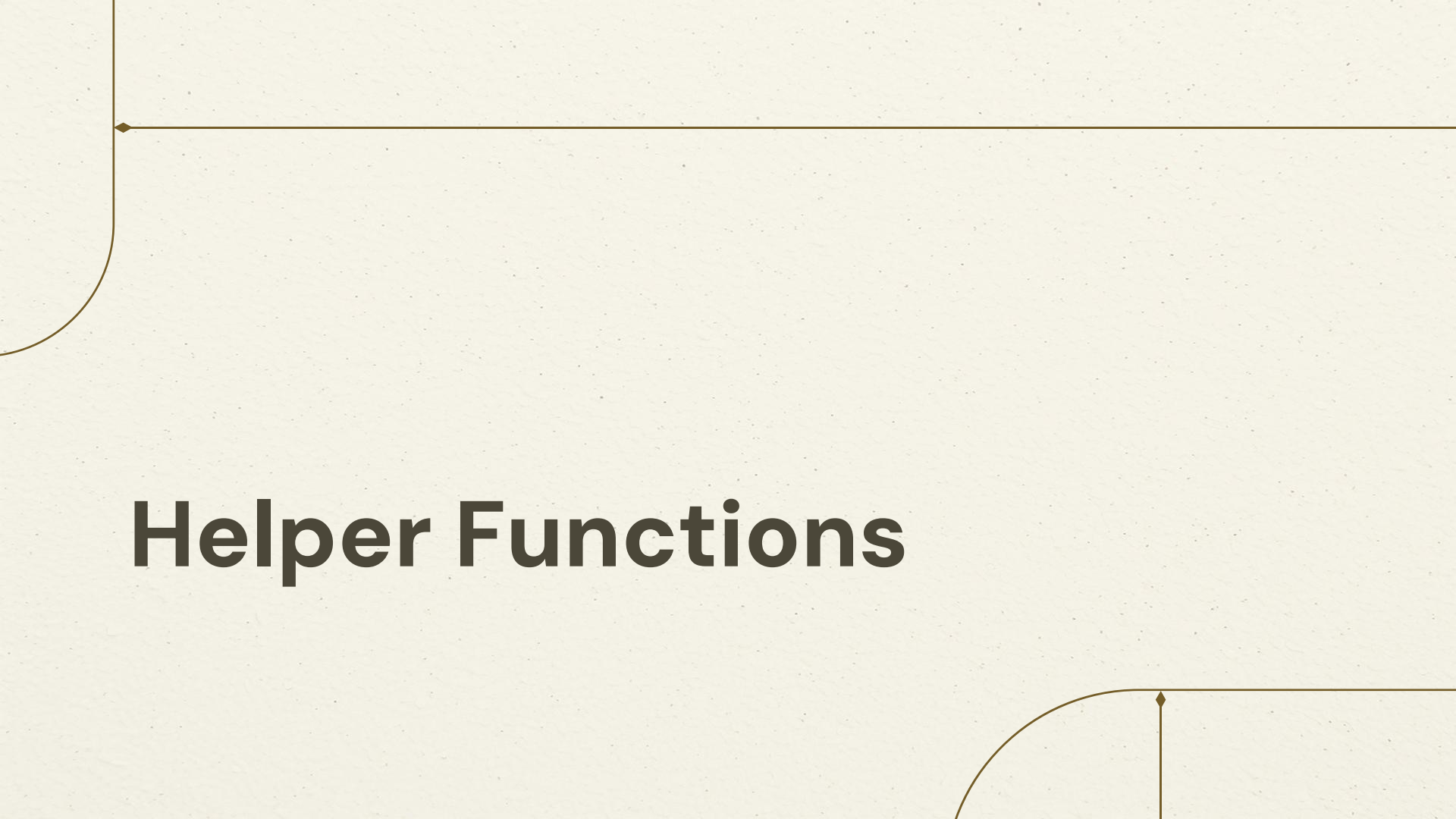
"the final is so easy"

The task force will then decode your message with this same key.

1. Encode this message. (Ex. a->b, e->f, x->y)
2. What is the plaintext in this scenario? What is the ciphertext? Is this asymmetric or symmetric encryption?
3. What will the task force do to decode your message?

Exercises

1. Is the Caesar Cipher easy to break or hard to break? Explain why.
2. Is RSA encryption easy to break or hard to break? Explain why.
3. Is guessing a key in RSA encryption NP or P?



Helper Functions

Helper Functions

What is a helper function:

- Helper functions break down the problem into **smaller subtasks**
 - These subtasks are often essential and/or repetitive
- They are **called in the main function** to help solve the problem

Helper Functions

What is a helper function:

- Helper functions break down the problem into **smaller subtasks**
 - These subtasks are often essential and/or repetitive
- They are **called in the main function** to help solve the problem

Things to consider:

- What data needs to be **passed into** the helper functions
- What data needs to be **returned** from the helper functions
- When we call helper functions and need to use its returned value, we should set the function to a variable

Helper Functions

An example where we use helper functions are Monte Carlo problems

Helper Functions

An example where we use helper functions are Monte Carlo problems

```
import random
def runTrial(lst):
    sum = 0
    for i in range(len(lst)):
        sum += lst[i] * random.randint(1, 10)
    return sum

def getExpectedValue(numTrials):
    total = 0
    lst = [1, 2, 3, 4]
    for trial in range(numTrials):
        total += runTrial(lst) #call our helper function
                               which takes in lst and returns a number
    return total/numTrials
```



Helper Functions Thought Exercise

Consider the game tic-tac-toe. What are some of the subtasks that we need for this game?



Helper Functions Thought Exercise

Subtasks for tic-tac-toe:

makeBoard(): construct and return a starter board (2D list)

Helper Functions Thought Exercise

Subtasks for tic-tac-toe:

makeBoard(): construct and return a starter board (2D list)

showBoard(board): displays the given board

Helper Functions Thought Exercise

Subtasks for tic-tac-toe:

makeBoard(): construct and return a starter board (2D list)

showBoard(board): displays the given board

takeTurn(board, player): let the given player (where player is either “X” or “O”)
make a move and return the updated board

Helper Functions Thought Exercise

Subtasks for tic-tac-toe:

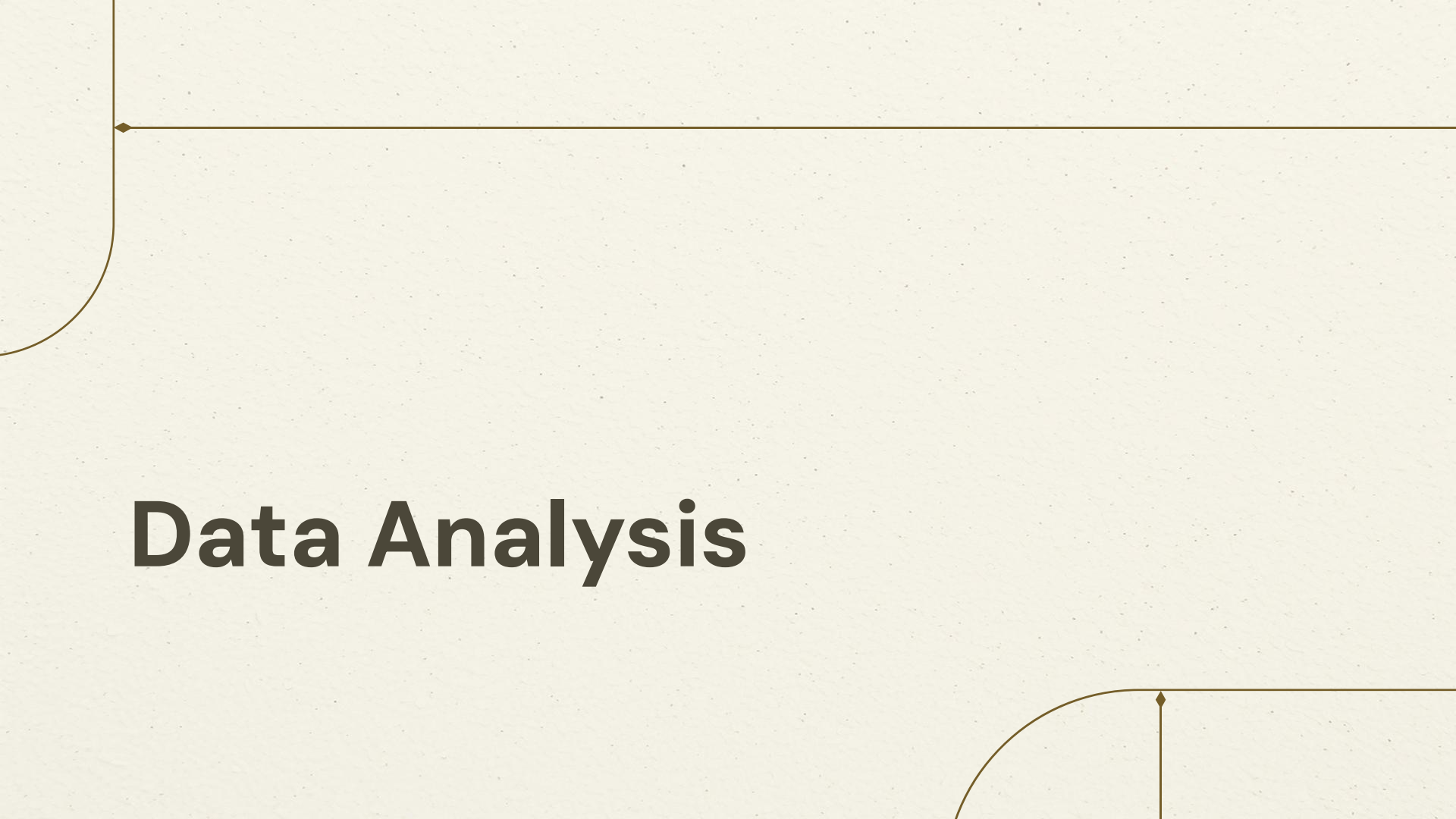
makeBoard(): construct and return a starter board (2D list)

showBoard(board): displays the given board

takeTurn(board, player): let the given player (where player is either “X” or “O”) make a move and return the updated board

isGameOver(board): return True or False based on whether game is over or not

We'll then call each of these helper functions in our main function. (See 11/01 lecture slides for more detail!)



Data Analysis

Data Types

- **Categorical:** data fall into separate, uncomparable categories
- **Ordinal:** data falls into separate, comparable categories (categories have an order!)
- **Numerical:** data are numbers

Data Types

- **Categorical:** data fall into separate, uncomparable categories
- **Ordinal:** data falls into separate, comparable categories (categories have an order!)
- **Numerical:** data are numbers

Practice: What type of data are...

- letter grades assigned for a class? (eg. A, B, C...)?
- cookie flavors?

Data Types

- **Categorical:** data fall into separate, uncomparable categories
- **Ordinal:** data falls into separate, comparable categories (categories have an order!)
- **Numerical:** data are numbers

Practice: What type of data are...

- letter grades assigned for a class? (eg. A, B, C...)? **ordinal**
- cookie flavors? **categorical**

Reading Files

```
f = open(filename, "r")  # open a file to read
```

```
text = f.read()  # reads the entire file as a string
```

```
f = open(filename, "w")  # opens the file to write
```

```
f.write(text)  # writes a string to the file
```

```
f.close()  # don't forget to close a file when you're done!!
```

Data Formats

Comma-Separated Values (CSV) - store data in two dimensions (spreadsheets)

- Use the csv library to read csv files

```
f = open("testfile.csv", "r")
```

```
data = list(csv.reader(f))
```

JavaScript Object Notation (JSON) - stores nested data like trees

- Use the json library

```
f = open("testfile.json", "r")
```

```
data = json.load(f)
```

Extracting Data from Plaintext

Plaintext data - data that can be read in a simple text editor but doesn't fit into any standard format

Look for patterns! Once you know where the info you need is located, consider using techniques like...

- **String slicing** can help you choose what parts of the data to use
- **Split method** breaks up data by a delimiter `# s.split(",")`
- **Index method** finds the location `# s.index(",")`
- **Strip method** removes whitespace `# s.strip()`

Changing Data

- **Updating values**
 - Access a column and use indexing to modify a value
- **Removing values**
 - Use **pop** to remove elements
- **Adding values**
 - Use **append** or **insert** to add values to rows
- **Headers**
 - Don't forget to skip the 0th row when necessary!

Statistical Analysis

- **statistics** library helps with descriptive statistics
 - eg `statistics.mean(data)`, `statistics.median(data)`, `statistics.mode(data)`

Practice: You're given `employees.csv`, where each row represents an employee and the 4th column in each row contains their age. Write code to find the most common age.

Statistical Analysis: Practice

Practice: You're given `employees.csv`, where each row represents an employee and the 4th column in each row contains their age. Write code to find the most common age.

```
import csv
f = open("employees.csv", "r")
employeeData = list(csv.reader(f))
ages = []
for row in employeeData:
    ages.append(employeeData[3])
f.close()
return statistics.mode(ages)
```

Data visualization & Matplotlib

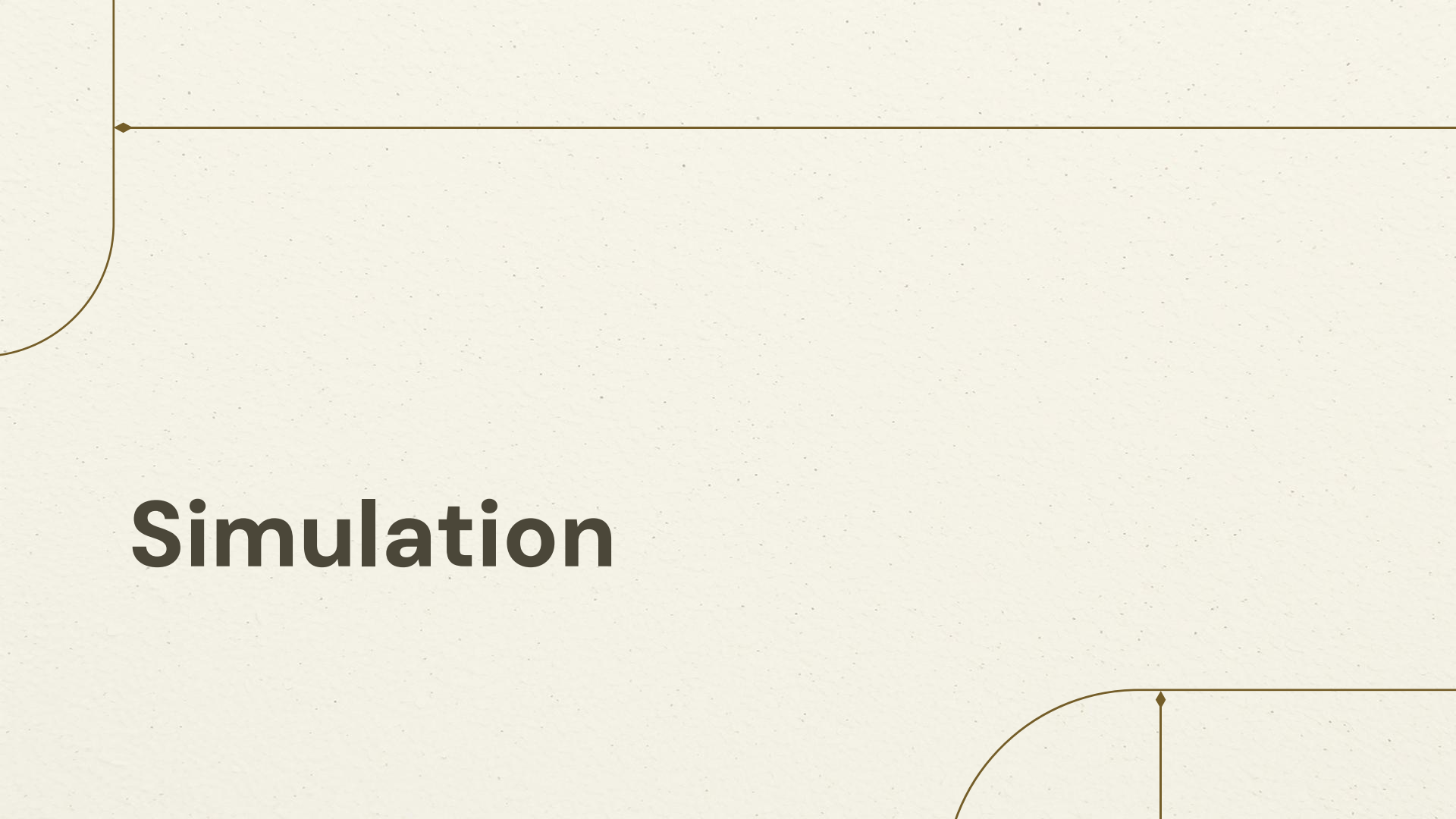
- One-dimensional - one feature of the data
 - **Histogram:** 1D, numerical data
 - **Bar graph:** 1D, ordinal data
 - **Pie chart:** 1D categorical data
- Two-dimensional - relationship between two features of data
 - **Scatter plot:** numerical v numerical
 - **Bar plots of averages/box plots:** numerical vs ordinal/categorical
 - **Table:** (sometimes) ordinal/categorical v ordinal/categorical
- Customize graphs using keyword arguments (eg color='blue')

Practice: What plot do you want to create if you want to show how people's ages relate to their favorite genre of movies?

Data visualization & Matplotlib

- One-dimensional - one feature of the data
 - **Histogram:** 1D, numerical data
 - **Bar graph:** 1D, ordinal data
 - **Pie chart:** 1D categorical data
- Two-dimensional - relationship between two features of data
 - **Scatter plot:** numerical v numerical
 - **Bar plots of averages/box plots:** numerical vs ordinal/categorical
 - **Table:** (sometimes) ordinal/categorical v ordinal/categorical
- Customize graphs using keyword arguments (eg color='blue')

Practice: What plot do you want to create if you want to show how people's ages relate to their favorite genre of movies? **numerical v categorical → bar plot of averages**



Simulation

MVC – Division of Labor

Model

- Stores the core components of the simulation (i.e. important information and data structures)
- `makeModel(data)`
- Usually where we **initialize** values

MVC – Division of Labor

Model

- Stores the core components of the simulation (i.e. important information and data structures)
- `makeModel(data)`
- Usually where we **initialize** values

View

- Display the current state of the model (aka, where you make what you *view*)
- `makeView(data, canvas)`
- Usually where we **call** `canvas.create_SHAPE`

MVC – Division of Labor

Model

- Stores the core components of the simulation (i.e. important information and data structures)
- `makeModel(data)`
- Usually where we **initialize** values

View

- Display the current state of the model (aka, where you make what you *view*)
- `makeView(data, canvas)`
- Usually where we **call** `canvas.create_SHAPE`

Controller

- Where we update (or *control*) components of the model
- Where we run rules
- `mousePressed(data, event) ...` and many more
- Usually where we **change** model values

Simulation Example

Let's make an animation where:

- We start with a blank canvas
- Every time the canvas is pressed, a orange circle is created with its center at the click location
- By default, each circle that's created will have a radius of 20
- When the up arrow is pressed, the radius for any new circles should increase by 5
- When the down arrow is pressed, the radius for any new circles should decrease by 5



What will our model need to have?



What will our model need to have?

- Radius for new circles
- Location and color of existing circles

What will our model need to have?

- **Radius for new circles**

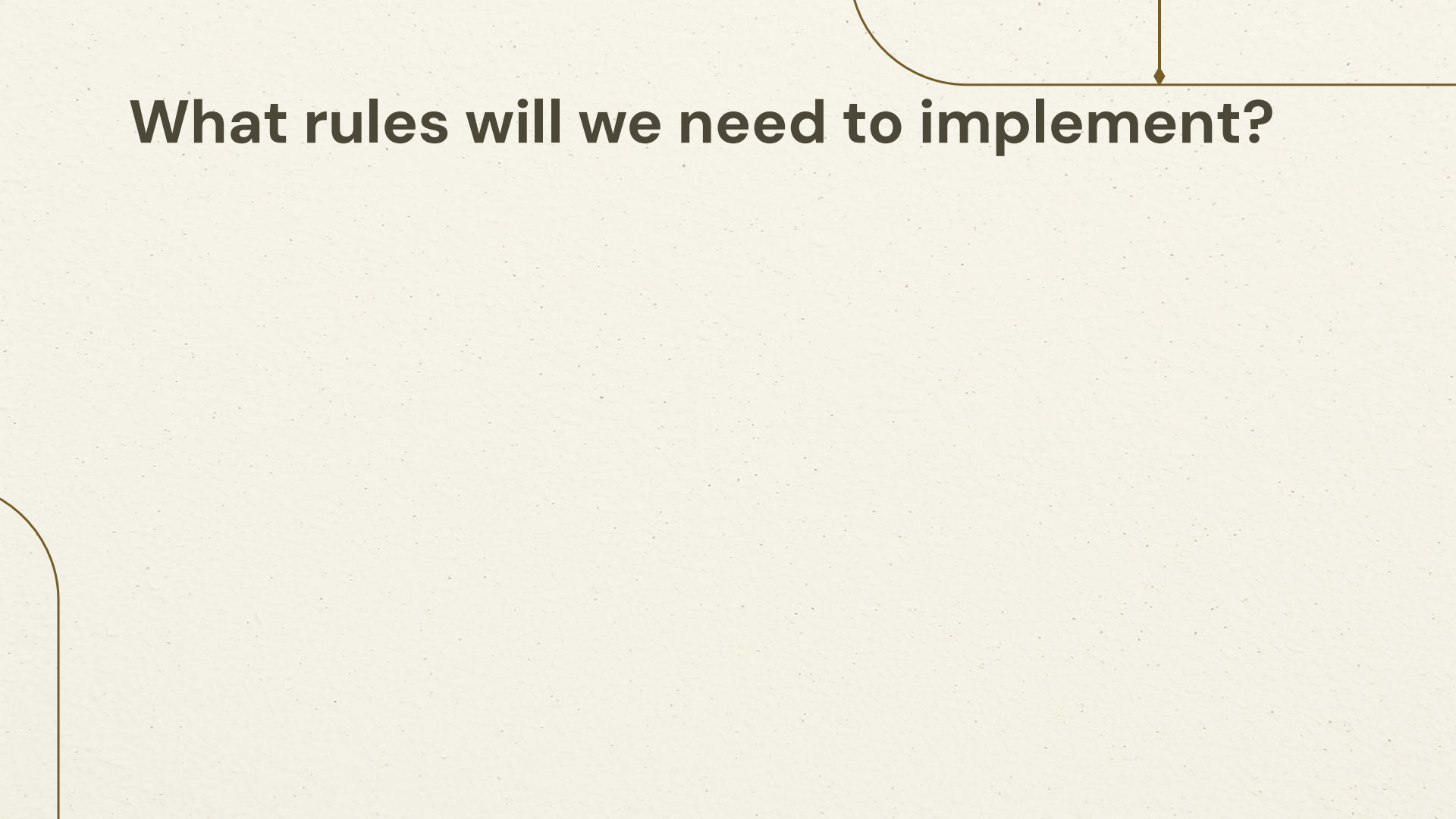
Radius of 20 at the start

```
data["radius"] = 20
```

- **Location and color of existing circles**

No circles in the beginning :(

```
data["circles"] = [ ] # will be [ [x1, y1, color1], [x2, y2, color2] ]
```



What rules will we need to implement?



What rules will we need to implement?

- Add a circle on mouse click
 - Update radius on arrow presses
- 

What rules will we need to implement?

- Add an orange circle on mouse click

```
def mousePressed(data, event):  
    newCircle = [event.x, event.y, 'orange']  
    data['circles'].append(newCircle)
```

- Update radius on arrow presses

```
def keyPressed(data, event):  
    if event.keysym == 'Up':    data['radius'] += 10  
    if event.keysym == 'Down': data['radius'] -= 10
```

Where are the circles???



```
def makeView(data, canvas):
```

```
    r = data["radius"]
```

```
    for circle in data["circles"]:
```

```
        x = circle[0]
```

```
        y = circle[1]
```

```
        color = circle[2]
```

```
        canvas.create_oval(x - r, y - r, x + r, y + r, fill=color)
```



How can we make our circles change color?

- Make a list of colors in makeModel
 - Make changes to runRules
- 

How can we make our circles change color?

- Make a list of colors in makeModel

```
data['colors'] = ['red', 'orange', 'yellow', 'green', 'blue', 'purple']
```

- Make changes to runRules

```
def runRules(data, call):  
    for circle in data['circles']:  
        newColor = random.choice(data['colors'])  
        circle[2] = newColor
```

Dice Simulation

Let's investigate the odds of rolling a 6 with normal dice!

How can we simulate rolling one die?



Dice Simulation

Let's investigate the odds of rolling a 6 with normal dice!

How can we simulate rolling one die?

```
def rollDice():  
    return random.randint(1, 6)
```

Dice Simulation

```
def rollDice():  
    return random.randint(1, 6)
```

Now, we want to create a simulation that let's us roll a die a bunch of times (n times) for quick data collection!

How many sixes, on average, do we roll?



Simulation

Now, we want to create a simulation that let's us roll a die a bunch of times (n times) for quick data collection!

```
def simulateDiceRolling(trials):  
    sixesRolled = 0  
    for i in range(trials):  
        if (rollDice() == 6):  
            sixesRolled += 1  
    print("The odds of rolling a 6 are: ", sixesRolled / trials)
```

Law of Large Numbers

Last, let's look at why we care about how many trials we do!

Say we do 2 trials:

Trial 1: we roll a 3

Trial 2: we roll a 6

Conclusion → there's a 50% chance that we roll a 6!!

Law of Large Numbers

Last, let's look at why we care about how many trials we do!

Say we do 15 trials:

2 6 5 2 2 1 5 4 2 3 4 2 1 3 3

Conclusion → there's a 6% chance that we roll a 6!!

Law of Large Numbers

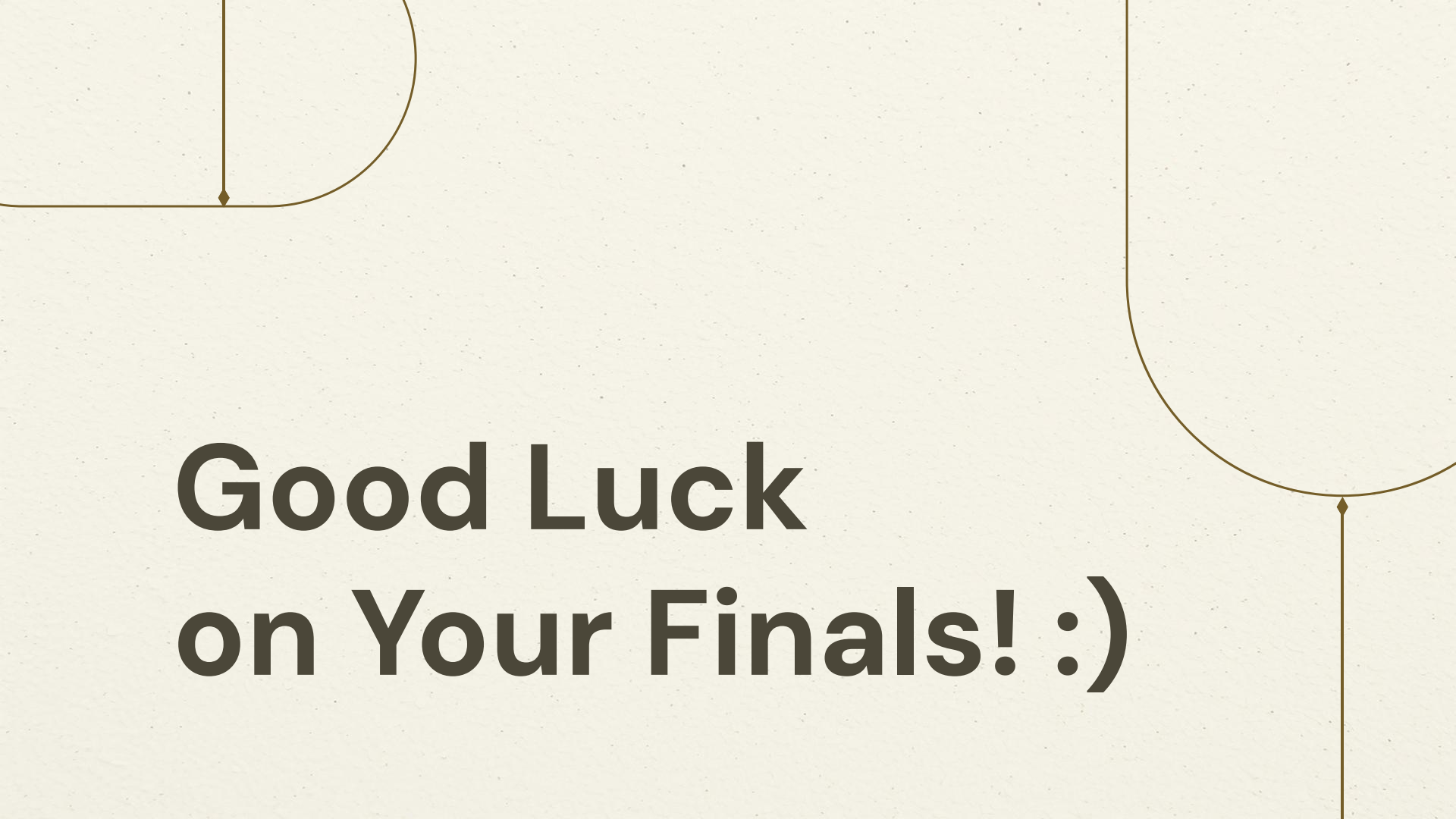
Last, let's look at why we care about how many trials we do!

Say we do 300 trials: (total 59 6's)

5154125425621243154654613365351156666223356113626252243214364
1663342635562443635214555412565156452431135342334156545153556
1455112335151313511515643145534522462264436616313531641414241
2662663111253554135153323565532451154314225456345411614115165
45353465636325614346353112333161412116211466541312664246

Conclusion → there's a 19.6% chance that we roll a 6!!

(wayyyyy closer to the true chance of $\frac{1}{6}$)



**Good Luck
on Your Finals! :)**