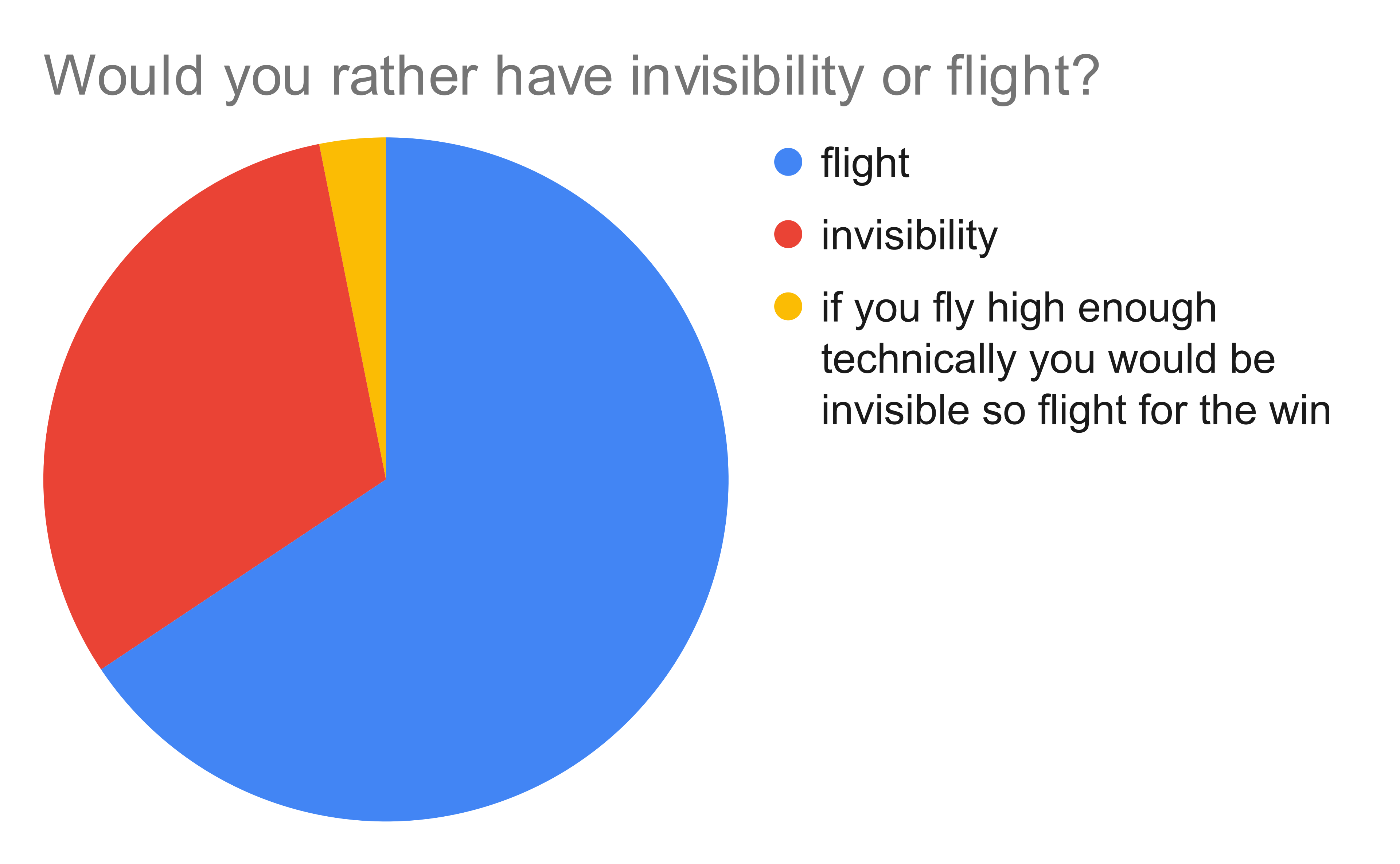
DISCUSSION 04

Tree Recursion, Python Lists

Mingxiao Wei
mingxiaowei@berkeley.edu

Feb 16, 2023

FROM LAST TIME... 👀



LOGISTICS 🏡

- Homework 04 due tomorrow 02/17
- CATS is released 😽😽😽
  - Try out the game here: cats.cs61a.org (not now LOL)
  - Checkpoint 1 due next Tue 02/21
  - The whole project due next Fri 02/24
  - Submit everything by next Thu 02/23 for one extra credit
- I have OH today 5-6 pm @ Warren 101B - OH is queue-based and staffed by multiple TAs/tutors, so you may not necessarily helped by me, but I'll be there :)

FEEDBACK FROM LAST TIME

- For labs
    - Sometimes the WWPD questions can be a bit confusing
        - I knew this as a 61A student 2 years ago (yeah I'm old...) so I'll walk through the WWPD question in the future if time allows
- For my slides
    - Less words, more pictures and diagrams
        - More pics for sure; and I'll try my best to keep the wording concise
    - More examples related to lab/hw or maybe walking through one of the lab problems
        - I can definitely walk through some lab problems (e.g., the WWPD one)
        - For discussions, the worksheet is provided, but if any of the worksheet problems is similar to a homework problem, I can give a verbal hint on that :)

FEEDBACK FROM LAST TIME 🪶

- If you have any feedback, please do let me know!
  - Leave anonymous feedback here: go.cs61a.org/mingxiao-anon (also linked on our section website)
  - Or just say it at the end of the attendance form

TREE RECURSION 🌴

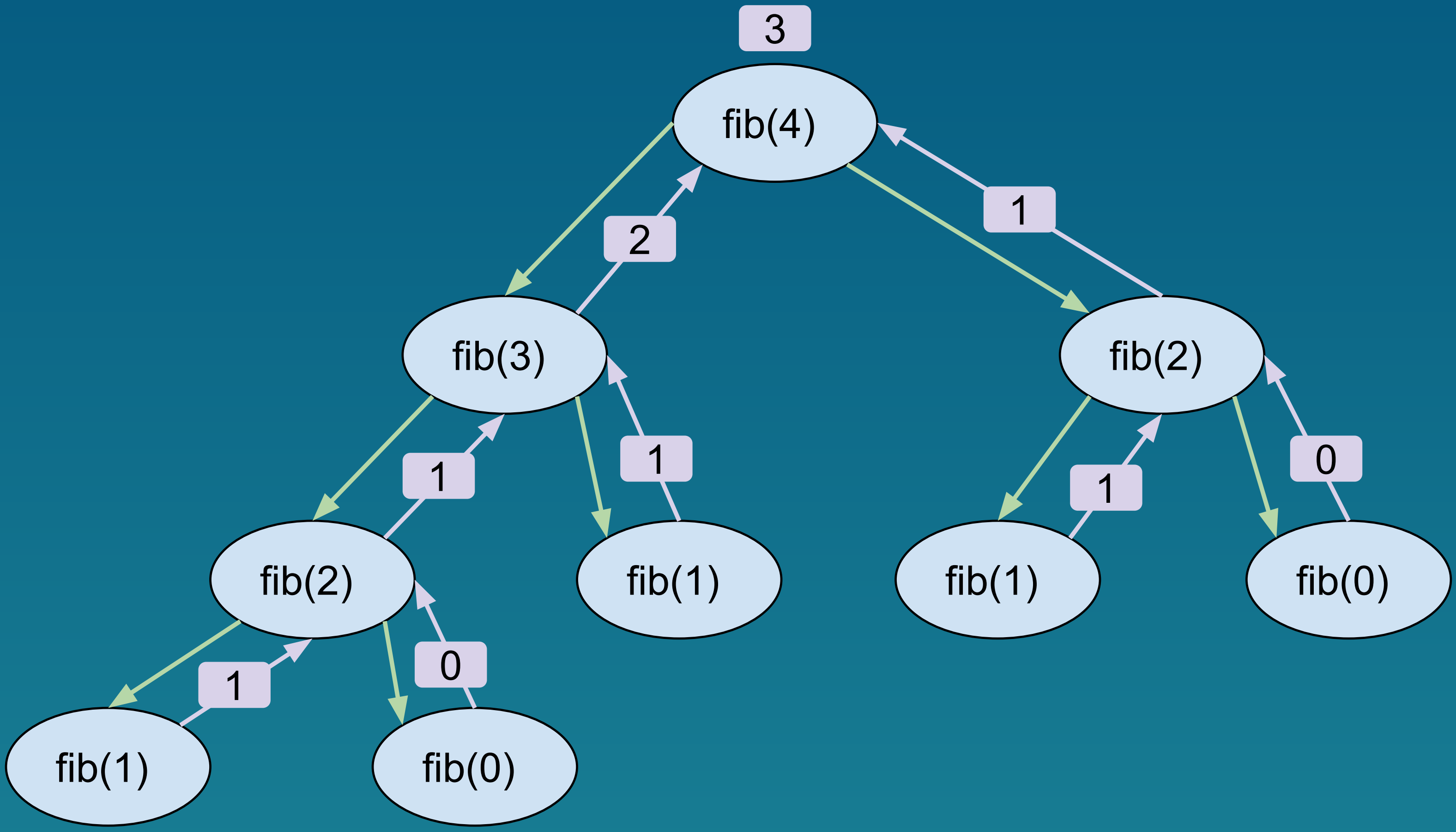
TREE RECURSION

- A tree recursive function makes more than one call to itself
- For example, to recursively calculate the n^{th} Fibonacci number:

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

- Now, what happens when we call `fib(4)`?
  - Each `fib(i)` node represents a recursive call to `fib`.
  - For `i >= 2`, each recursive call `fib(i)` makes another two recursive calls, `fib(i - 1)` and `fib(i - 2)`.
  - Whenever we reach a `fib(0)` or `fib(1)` node, directly return 0 or 1 - base cases.

TREE RECURSION



COUNT PARTITIONS REVISIT

Given two positive integers `n` and `m`, return the number of ways in which `n` can be expressed as the sum of positive integer parts up to `m` in increasing order.

```
def count_partitions(n, m):
    if n == 0:
        return 1
    elif n < 0 or m == 0:
        return 0
    else:
        with_m = count_partitions(n-m, m)
        without_m = count_partitions(n, m-1)
        return with_m + without_m
```

COUNT PARTITIONS REVISIT

Given two positive integers `n` and `m`, return the number of ways in which `n` can be expressed as the sum of positive integer parts up to `m` in increasing order.

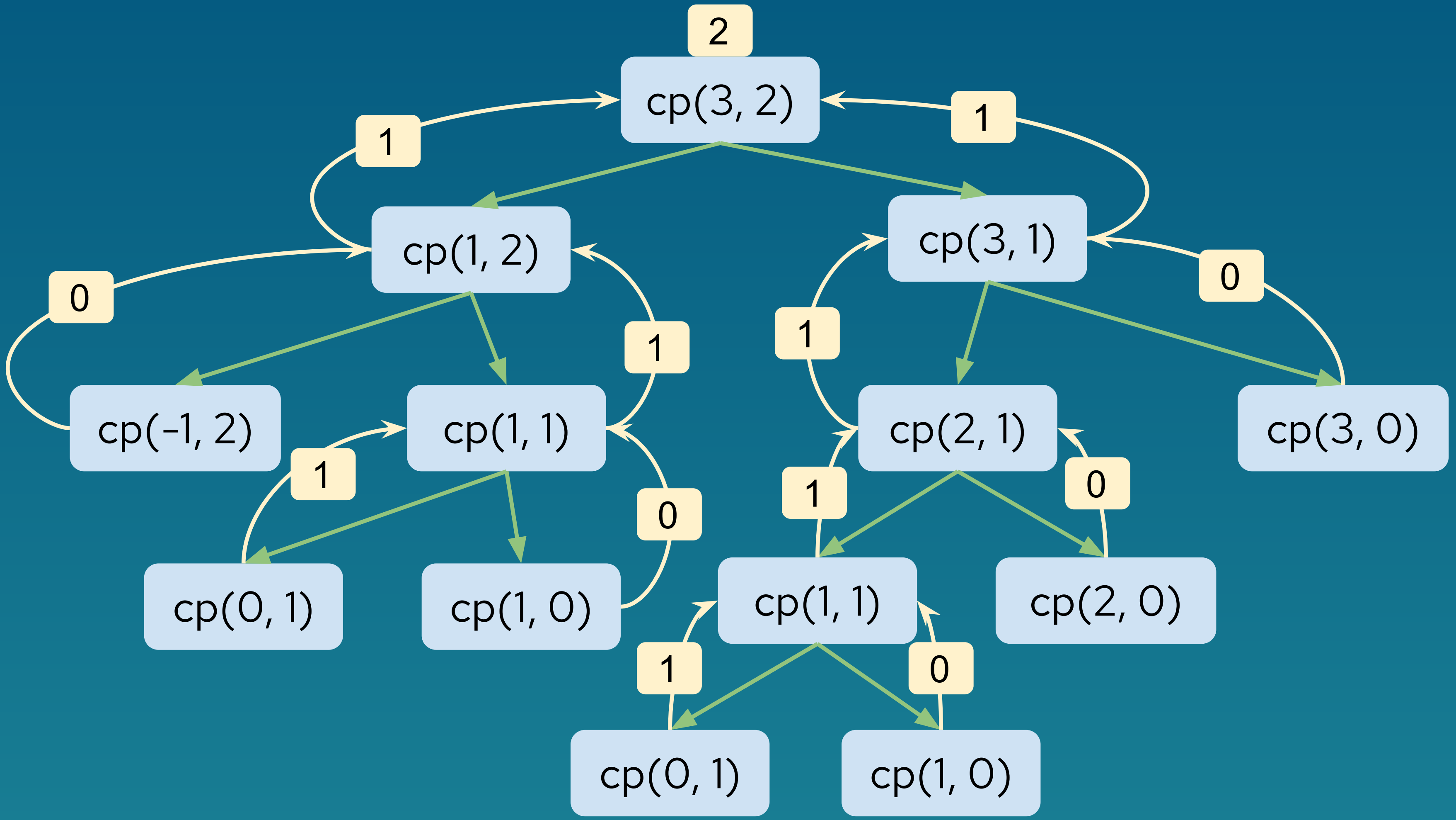
- Recursive case: Since each integer part is up to `m`, at each step, where each step generates one number in the partition, we have two choices:
    1. Use `m` to partition `n`, so that at the next step, `n` becomes `n - m`, and the largest possible part is still `m`
    2. Don't use `m`. So at the next step, `n` remains unchanged, but `m` becomes `m - 1` (we choose not to use the largest possible part, `m`, so the next largest possible one is `m - 1`)
- The two choices will result in two distinct sets of results, since in the first one we use `m` to partition, while in the second one we use at most `m - 1`
- Therefore, the total # of partitions = # partitions from choice 1 + # partitions from choice 2

COUNT PARTITIONS REVISIT

Given two positive integers `n` and `m`, return the number of ways in which `n` can be expressed as the sum of positive integer parts up to `m` in increasing order.

- Base case:
  - `n == 0` - note that since `n` and `m` are positive integers according to the problem description, when `n` is 0, it could only be the case where `n - m` results in 0 from the previous recursive call. In other words, when `n` is 0, it menas that we've successfully partitioned `n` so that there's nothing left to partition. In this case, return 1, since we found one valid parition.
  - `n < 0` - similarly, since the original input to the function must be positive integers, a negative `n` can only result from `n - m` from the last step. In this case, `m` was greater than `n` from the last step, indicating that the partition was not successful.
  - `m == 0` - also similarly, a negative `m` can only result from `m - 1` from the last step. Since the question requires that all parts of a partition are positive integers, such an partition is invalid.

COUNT PARTITIONS REVISIT



PROBLEM SOLVING STRATEGIES

- In tree recursion, often times each recursive call represents one "choice" we have
    - Sometimes there may be so many choices that we'll need to use a loop to iterate through them
- Think about what choices you have and each step, and how to combine these choices to get the final result
    - This could be sum, max, min, etc.

WORKSHEET Q1, 2

PYTHON LISTS

[🐍, 🐍]

LIST SLICING

- List slicing creates a copy of part or all of the list.

```
lst[ <start index> : <end index> : <step size> ]
```

- `start index`
  - index to start at, *inclusive*, default to 0[*]
- `end index`
  - index to end by, *exclusive*, default to `len(lst)`[*]
- `step size`
  - difference between indices of elements to include , default to 1
  - negative steps means stepping backwards

* when step is positive (when step is negative, start index defaults to the end of the list and end index defaults to the start of the list)

** whenever you see a negative number `-i`, think of it as `len(lst) - i`

LIST SLICING

```
>>>  lst = [1, 2, 3, 4, 5]
>>>  lst[1: ]
[2, 3, 4, 5]
>>>  lst[:-2]
[1, 2, 3]
>>>  lst[1::2]
[2, 4]
>>>  lst[::-1] # reverse the list
[5, 4, 3, 2, 1]
>>>  lst[5:9] # list slicing won't cause an index error
[]
```

Takeaway: list slicing picks elements at indices `start`, `start + step`, `start + 2 * step`, … and stops before `end`, and makes those selected elements into a new list

LIST COMPREHENSION

- List comprehensions are a compact and powerful way of creating new lists out of sequences.

```
[<expr> for <var> in <seq> if <cond>]
```

- In English, this translates to:
  - Compute the expression for each element in the sequence if the condition is true for that element (or skip this check if there's no condition)
- Note:
  - `if <cond>`is optional.
  - `<expr>` and `<cond>` may refer to `<var>`, which is essentially every element in the sequence

LIST COMPREHENSION

- List comprehensions are a compact and powerful way of creating new lists out of sequences.

```
[<expr> for <var> in <seq> if <cond>]
```

- In Python, this translates to:

```
lst = []
for <var> in <seq>:
    if <cond>:
        lst.append(<expr>)
```

- Note:
  - `if <cond>` is optional.
  - `<expr>` and `<cond>` may refer to `<var>`, which is essentially every element in the sequence

EXAMPLES

```
>>> lst = [1, 2, 3]
>>> [i ** 2 for i in lst if i % 2 == 0]
[1, 9]
>>> [[i, j] for i in lst for j in lst if i != j]
[[1, 2], [1, 3], [2, 1], [2, 3], [3, 1], [3, 2]]
>>> # here [i, j] for i in lst is considered as
>>> # the expression for the second list comprehension
```

WORKSHEET Q3,4,5

DICTIONARIES 📚

DICTIONARIES

- Dictionaries maps keys to their corresponding values
- Create a dictionary
    - `{key1: val1, key2: val2, key3: val3}`
- Access values from a dictionary
    - `dict[key]`
    - If `key` does not exist in `dict`, this will error
- Add to / Modify a dictionary:
    - `dict[key] = val`
    - If `key` does not exist in `dict`, this will create a new entry. Otherwise it updates the value corresponding to `key` to be `val`
- `len(dict)` returns the number of entries (key-value pairs) in the dictionary
- The key of a dictionary must be immutable (numbers, strings, tuples, but NOT lists)

WORKSHEET Q6

ATTENDANCE! 🤠

go.cs61a.org/mingxiao-att

- The attendance form and slides are both linked on our section website!
- Please leave any anonymous feedback here go.cs61a.org/mingxiao-anon
- Once again, please do remember to fill out the form by midnight today!!