Discussion 7

Mutable Trees, Linked Lists, and Efficiency

Antonio Kam

anto [at] berkeley [dot] edu

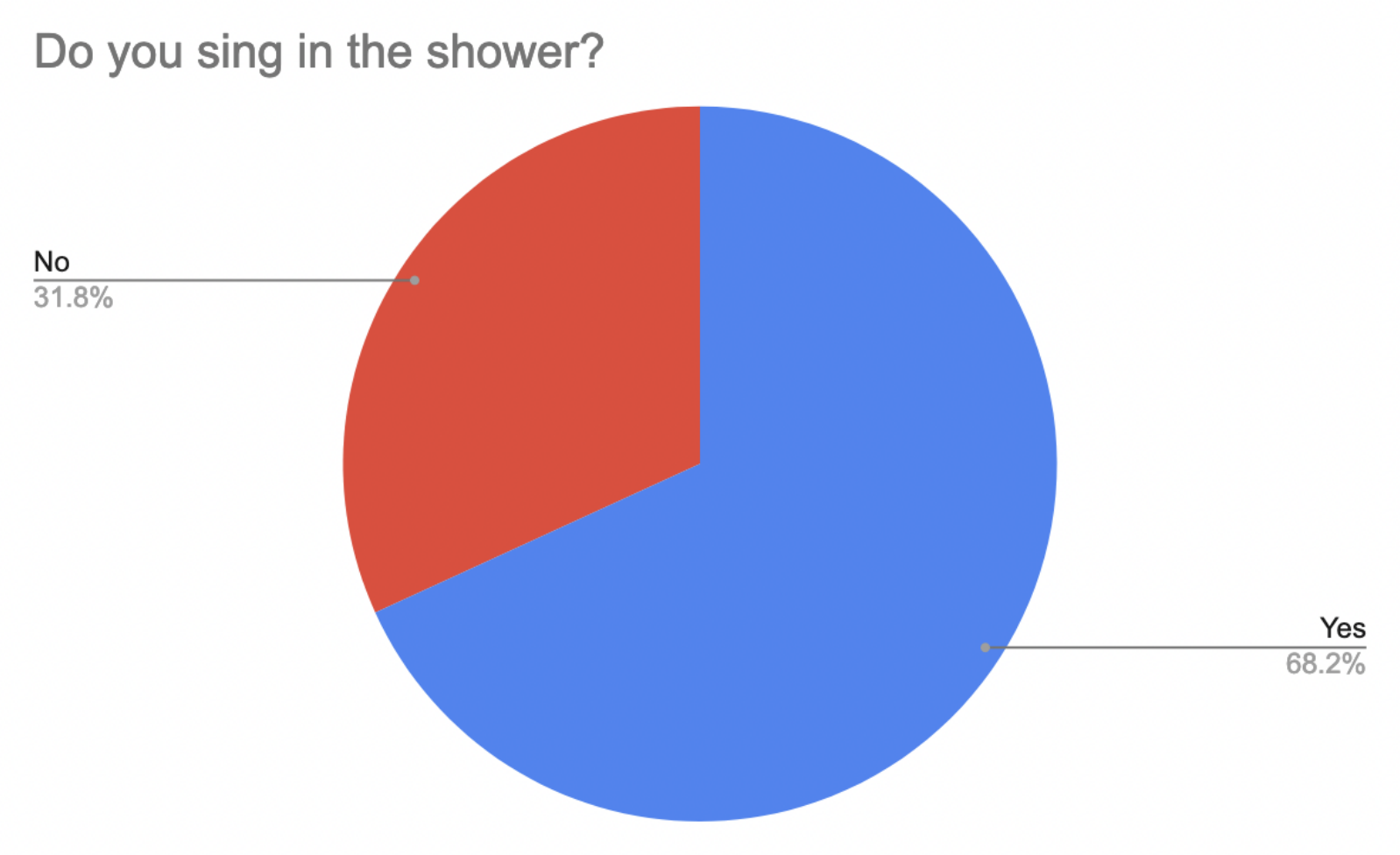
All slides can be found on

`teaching.rouxl.es`

Announcements

- New CSM Sections Open! (Would highly recommend joining even now; these sections are super helpful)
- Ants due in today! Please remember to submit!
  - As a reminder, you can request an extension on the contact page of CS61A's website
- No Homework this week 👀
- Lab next week is optional (I highly recommend doing all the questions - they're fairly difficult and will be somewhat similar to the difficulty of exam problems)
- Good luck for the midterm 🥲

Results from last section



Notes from last section

- `repr` and `str`
  - Going to briefly cover this at the start
- "Practice problems for midterm"
  - I can't do this during the actual discussion, but there's some stuff on my teaching website and my course notes website (the content for the second link may not match this semester - I wrote it when I was taking the course last fall)
- "Are there any quirks or unique elements of inheritance in Python than you wouldn't normally see in other programming languages?"
  - Not really, other than that instance variables can be created whenever rather than always needing to be created at the start

Temperature Check 🌡️

- Linked lists (have not been covered in lecture yet)
- Mutable Trees
- Tree Recursion (they're pretty related with trees)

Quick `repr`, `str` mini-lecture

Linked Lists

Linked Lists

- You can use linked lists to create your own version of a sequence
- They are generally useful when you want to have an infinitely-sized list, or want to dynamically change the size of the list (more useful in 61B if you do end up taking it)
- In general, linked lists problems can be solved using both iteration and recursion
  - Like trees, they are recursive data structures, but unlike trees, you can use both recursion and iteration to solve them.

Linked Lists (Anatomy)

```
class Link:
    empty = ()

    def __init__(self, first, rest = Link.empty):
        assert rest is Link.empty or isinstance(rest, Link)
        self.first = first
        self.rest = rest
```

- Linked lists have a `first` (similar to `label`) attribute and a `rest` (similar-ish to `branches`) attribute.

Linked Lists (Construction)

Note to Anto: Draw box-and-pointer diagram

```
s = Link(1, Link(2, Link(3)))
>>> s.first
1
>>> s.rest
Link(2, Link(3))
>>> s.rest.first
2
s2 = Link(1, 2) # This will error because rest is not a linked list
>>> s.rest.rest.first
3
```

Worksheet!

Attendance

`links.rouxl.es/disc`

Trees

Trees (construction)

- The `Tree` constructor takes in a `label`, and an *optional* `list` of `branches`. If `branches` isn't given, it defaults to an empty list `[]`

```
class Tree: # simplified version compared to the 61A implementation
    def __init__(self, label, branches = []):
        assert type(branches) == list
        for item in branches:
            assert isinstance(item, Tree)
        self.label = label
        self.branches = branches
```

Construction: `Tree(2)`; `Tree(2, [Tree(3), Tree(4)])`

Note that the branches **must** be in a list.

Trees (access)

```
t = Tree(3, [Tree(4, [Tree(5)]), Tree(6)])
>>> t.label
3
>>> t.branches
[Tree(4, [Tree(5)]), Tree(6)]
```

Worksheet!

Efficiency

Efficiency

- In general, your programs take time to run, but some perform better than others.
  - While at a small scale, you can't really tell the difference in terms of time, when your inputs become really big, the 'runtime factor' of your algorithm starts to really matter in the amount of time it takes
- There are different types of runtime, but the ones we'll be focusing on are:
  - Constant
  - Logarithmic
  - Linear
  - Quadratic
  - Exponential

Efficiency

- In general, your programs take time to run, but some perform better than others.
  - While at a small scale, you can't really tell the difference in terms of time, when your inputs become really big, the 'runtime factor' of your algorithm starts to really matter in the amount of time it takes
- There are different types of runtime, but the ones we'll be focusing on are:
  - Constant
  - Logarithmic
  - Linear
  - Quadratic
  - Exponential

Efficiency - Constant

```python
def constant(n):
    for i in range(50):
        print(50)
```

- Notice how no matter what you do to the input of `n`, you'll only be running through the for loop `50` times. This is therefore **constant** in runtime.

Efficiency - Logarithmic

```
def log(n):
    while n > 0:
        print(n % 10)
        n //= 10
```

- For this one, when we go through the while loop, we're dividing by 10 each time.

Efficiency - Linear

```
def linear(n):
    while n > 0:
        print(n)
        n -= 1
```

- In this case, the input size scales linearly to the input that you give

Efficiency - Quadratic

```
def quadratic(n):
    for i in range(n):
        for j in range(n):
            print(i + j)
```

- In this case, the amount of steps needed scales quadratically

Efficiency - Exponential

```
def exponential(n):
    assert n >= 0 and type(n) == int
    if n == 0 or n == 1:
        return n
    else:
        return exponential(n - 1) + exponential(n - 2)
```

- As the input size gets larger, we have to make significantly more calls
- This is very inefficient and should be avoided where possible (sometimes it is not possible to avoid exponential time)

Worksheet!

Mental Health Resources

- CAPS:
  - If you need to talk to a professional, please call CAPS at 510-642-9494.
- After Hours Assistance
  - For any assistance after hours, details on what to do can be found at this link

Anonymous Feedback Form

`links.rouxl.es/feedback`

Thanks for coming! 🥳

Please give me feedback on what to improve!