# CPSC 231 Tutorial #22

michael-hung.ca/teaching

# Reminders

---

**TODAY**

Quiz 11 Review

**TOMORROW**

Assignment 6 Individual Component Due

# Quiz 11 Review

---

1. Best practices for designing data types

   - **Wide Interface** (many operations) vs **Narrow Interface** (few operations)
   - Describing data-type operations with **computer code** vs **natural language** (e.g. English)
   - Writing **client code** first vs **implementation code** (of the data type) first
   - Focusing on the **representation** (instance variables) vs its **behaviour** (methods)

# Quiz 11 Review

---

1. Best practices for designing data types

- **Wide Interface** (many operations) vs **Narrow Interface** (few operations)
- Describing data-type operations with **computer code** vs **natural language** (e.g. English)
- Writing **client code** first vs **implementation code** (of the data type) first
- Focusing on the **representation** (instance variables) vs its **behaviour** (methods)

# Quiz 11 Review

---

2. Design an API for a Fraction data type

| Operation | Description |
|---|---|
| Fraction(num, den) | makes a new Fraction with given numerator and denominator |
| num | variable holding numerator of fraction |
| den | variable holding denominator of fraction |
| add(f) | returns Fraction sum of this and given Fraction |
| mul(f) | returns Fraction product of this and given Fraction |
| div(d) | returns Fraction divided by d |
| val() | returns num / den |
| toString() | returns String representation of this Fraction |

# Quiz 11 Review

---

3. Rewrite code using Fraction API.

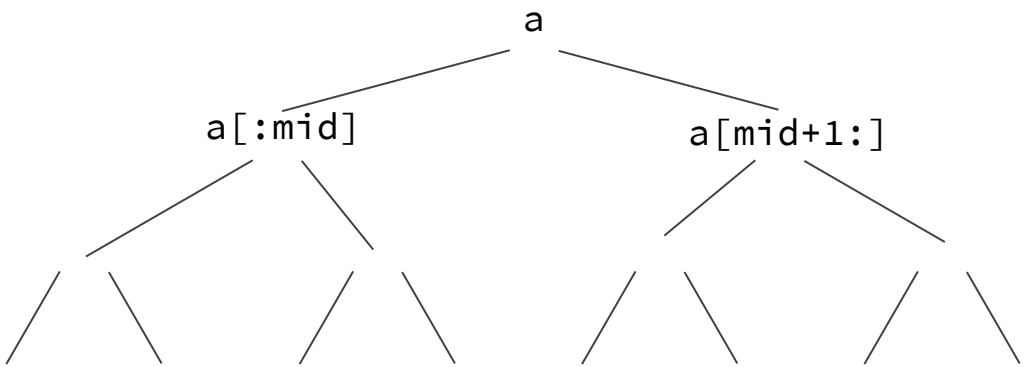
```
# gcd function does not change
a = Fraction(5, 8)
b = Fraction(2, 3)
c = Fraction(1, 12)

# calculate d = a * b + c
# first calculate p = a * b
p = a.mul(b)
# then calculate s = p + c
s = p.add(c)
# reduce fraction
divisor = gcd(s.num, s.den)
d = s.div(divisor)
#convert and print
print('5/8 * 2/3 + 1/12 = ' + d.toString())
```

# Recursion

---

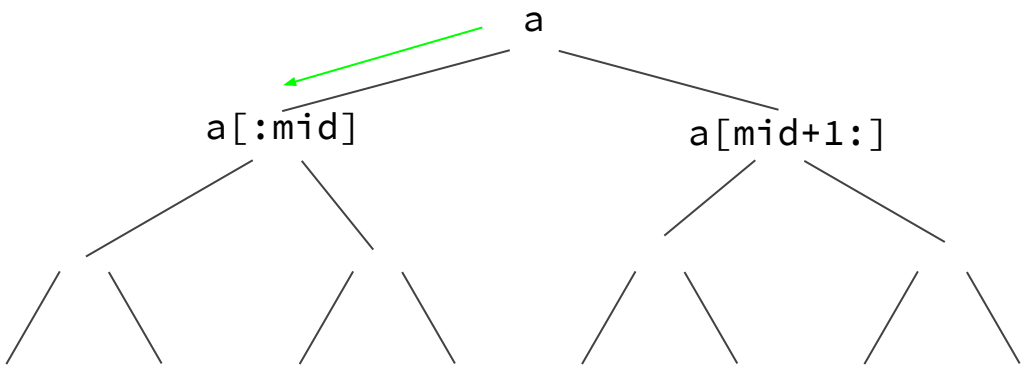
Think of recursion as a tree.



# Recursion

---

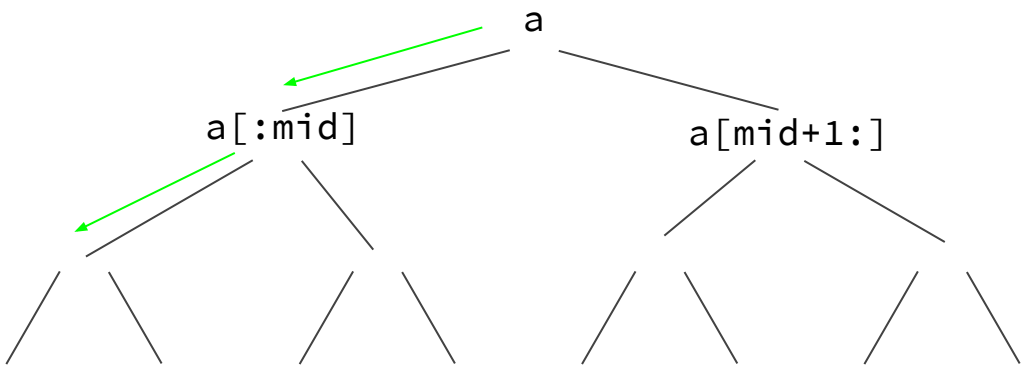
Think of recursion as a tree.



# Recursion

---

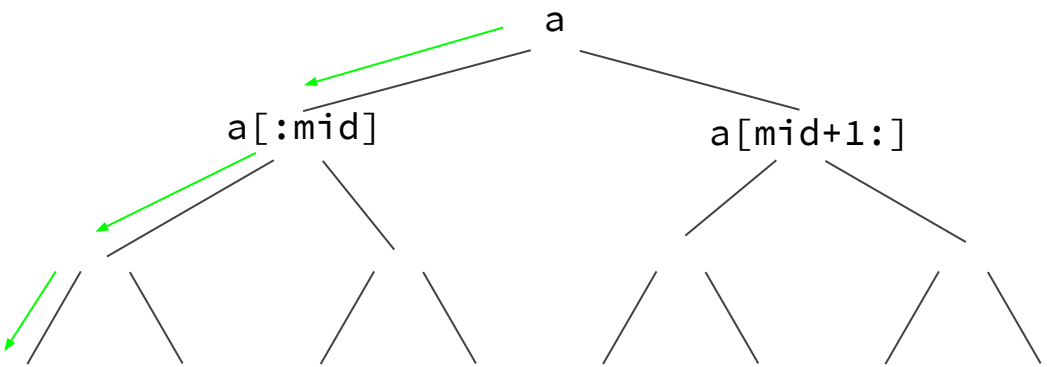
Think of recursion as a tree.



# Recursion

---

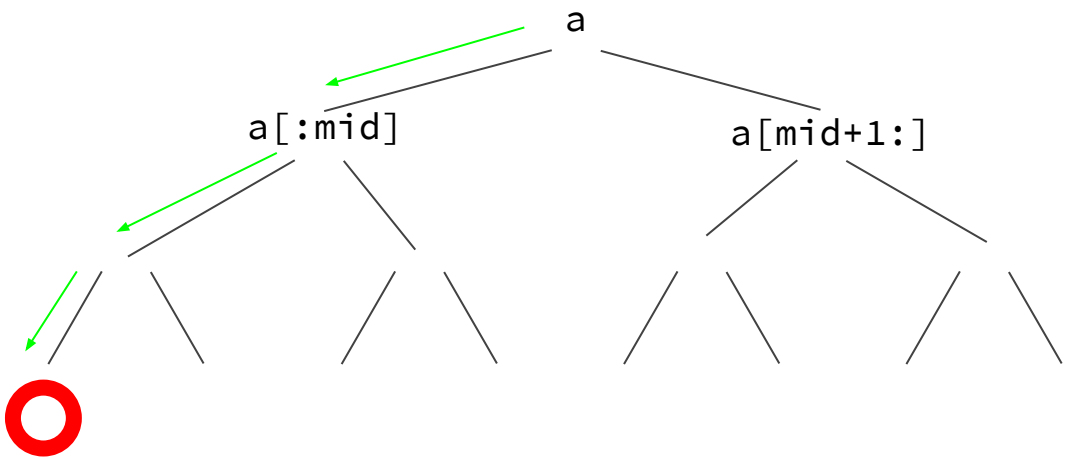
Think of recursion as a tree.



# Recursion

---

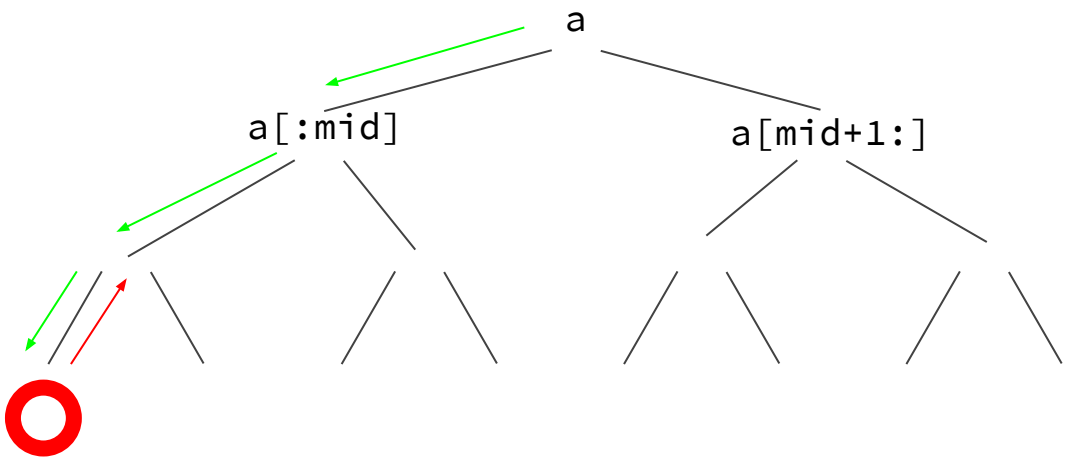
Think of recursion as a tree.



# Recursion

---

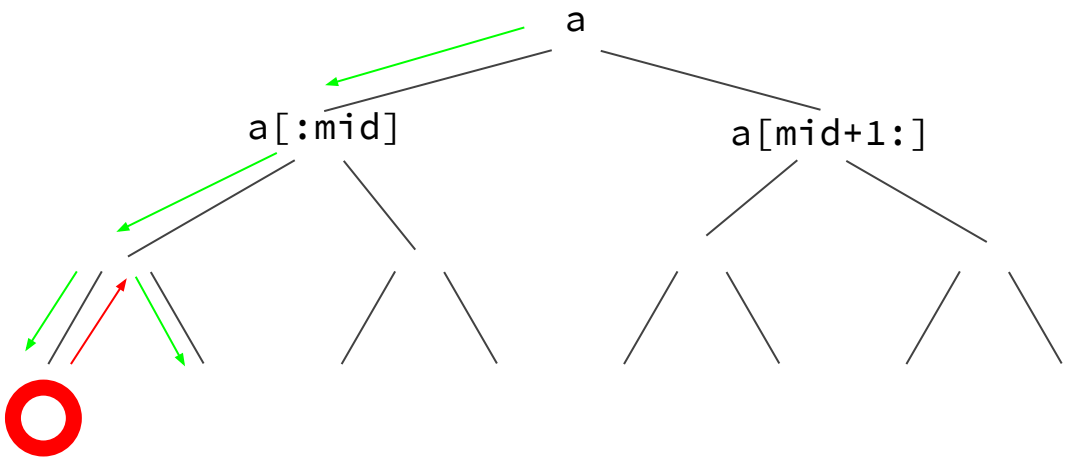
Think of recursion as a tree.



# Recursion

---

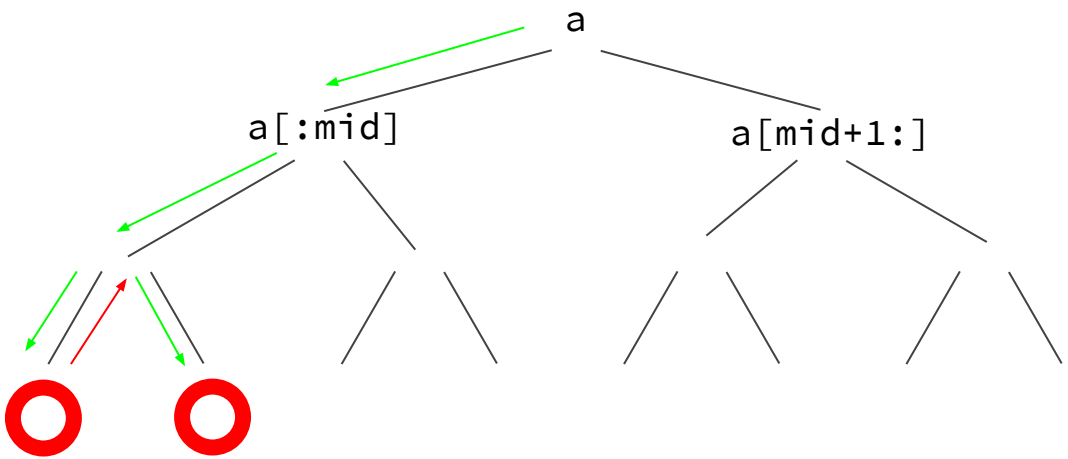
Think of recursion as a tree.



# Recursion

---

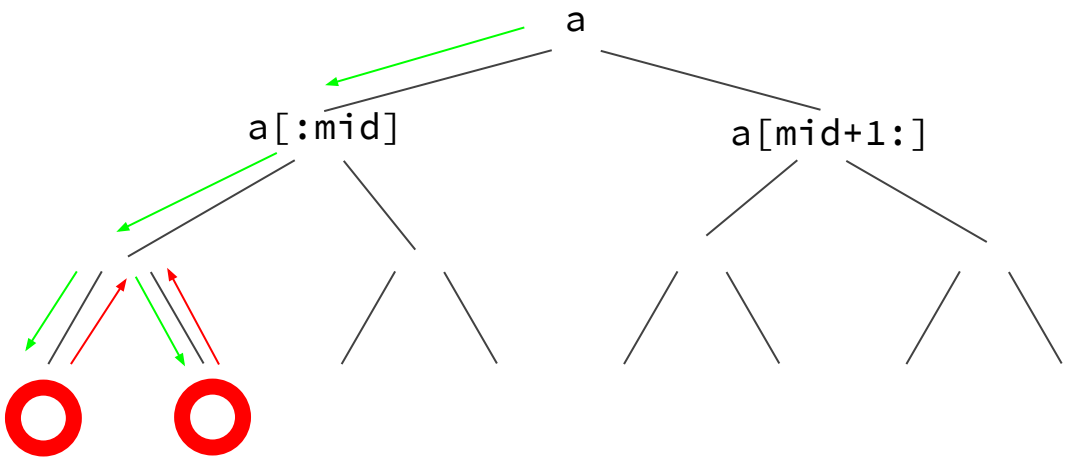
Think of recursion as a tree.



# Recursion

---

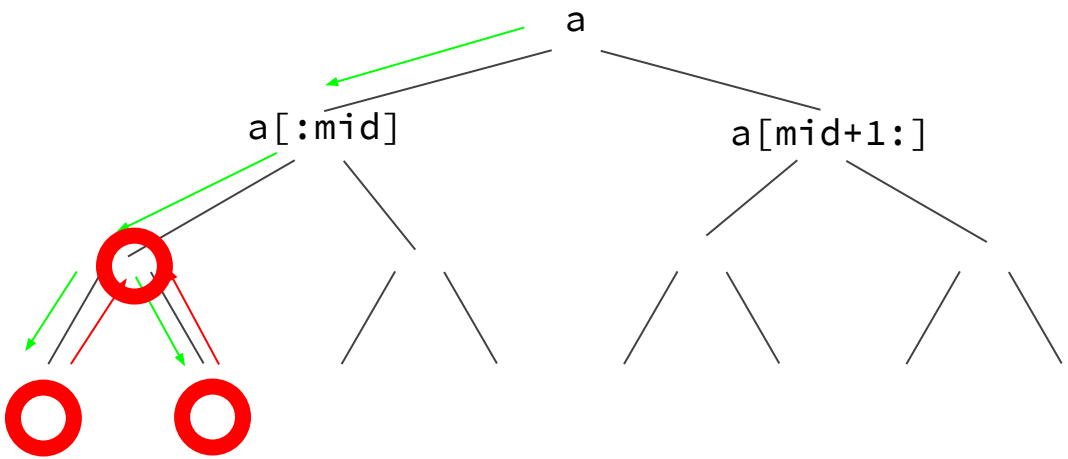
Think of recursion as a tree.



# Recursion

---

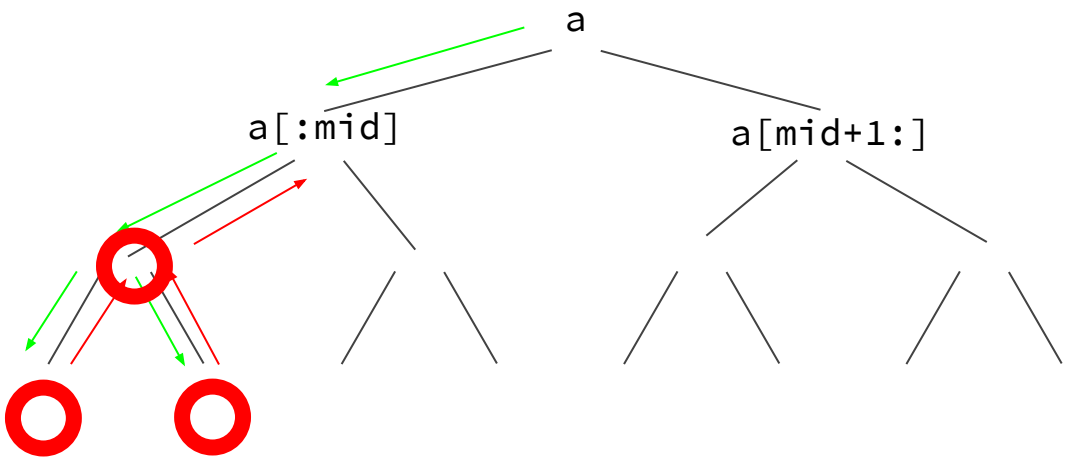
Think of recursion as a tree.



# Recursion

---

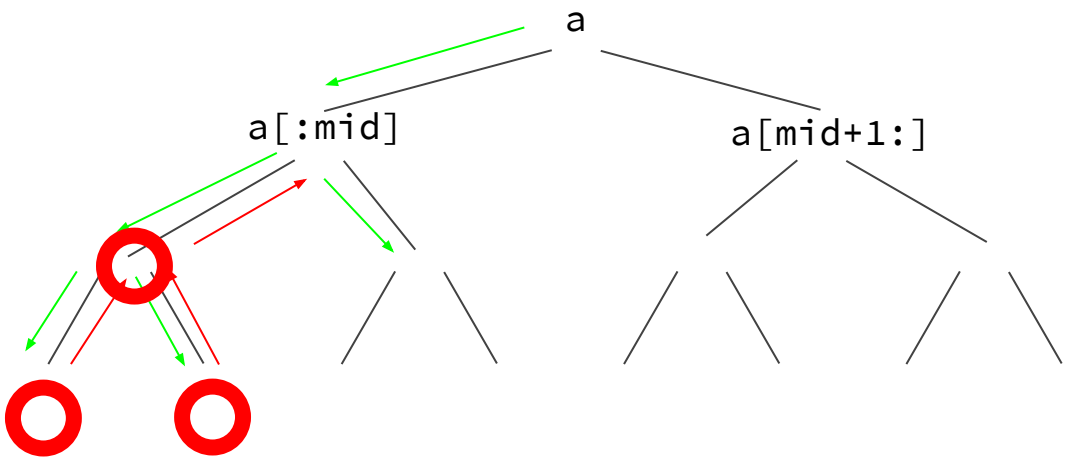
Think of recursion as a tree.



# Recursion

---

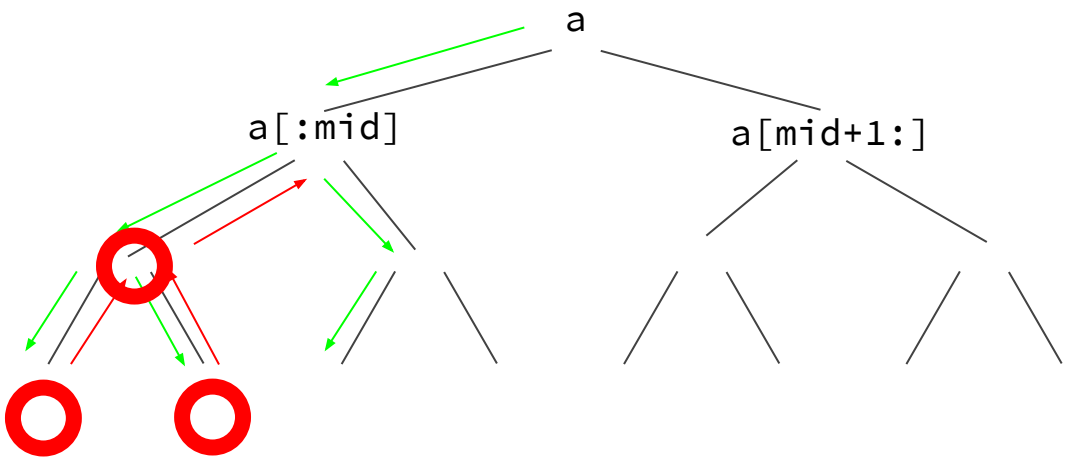
Think of recursion as a tree.



# Recursion

---

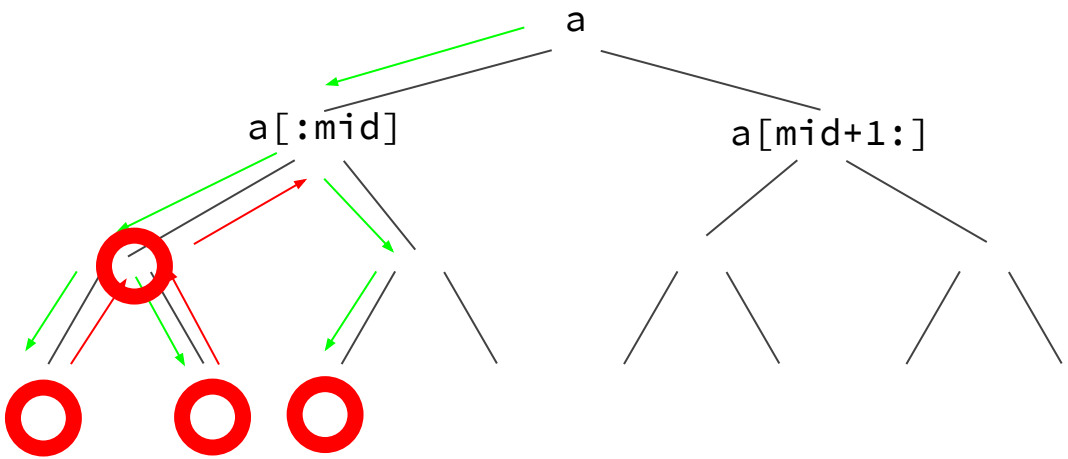
Think of recursion as a tree.



# Recursion

---

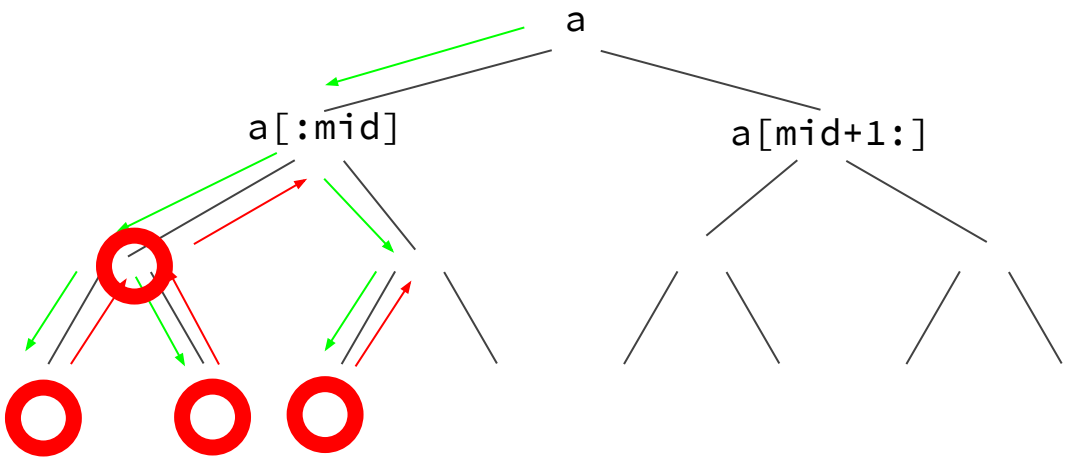
Think of recursion as a tree.



# Recursion

---

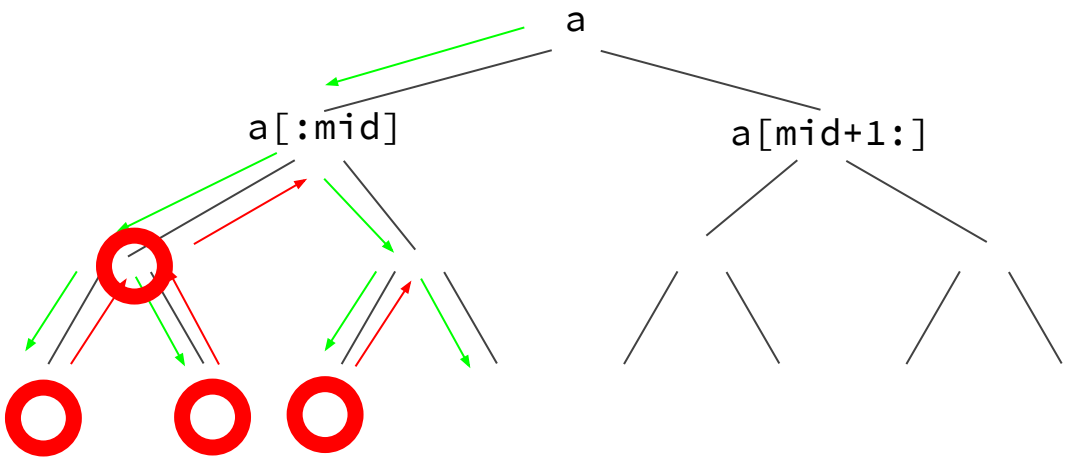
Think of recursion as a tree.



# Recursion

---

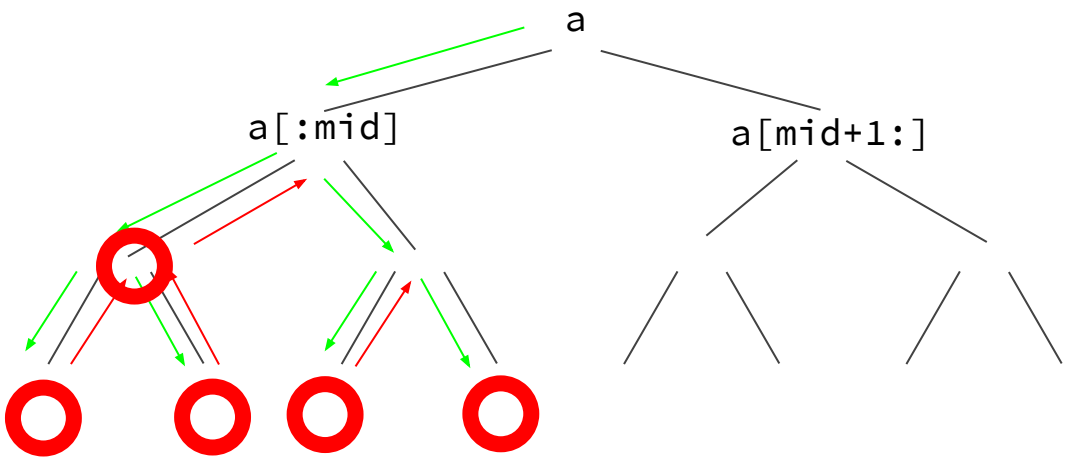
Think of recursion as a tree.



# Recursion

---

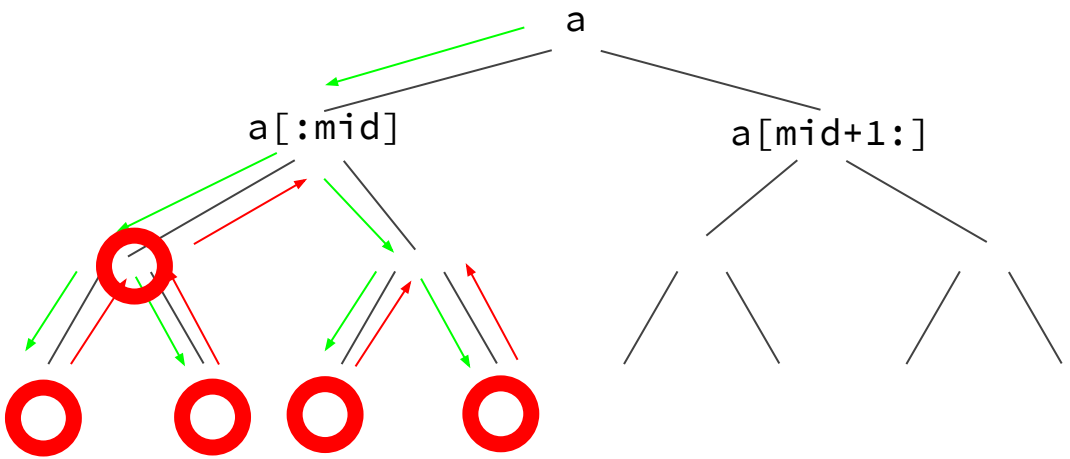
Think of recursion as a tree.



# Recursion

---

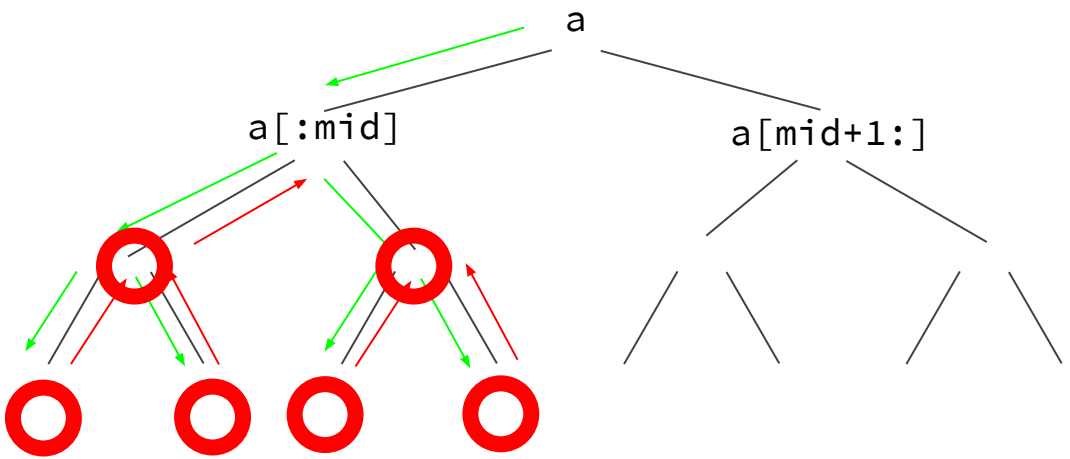
Think of recursion as a tree.



# Recursion

---

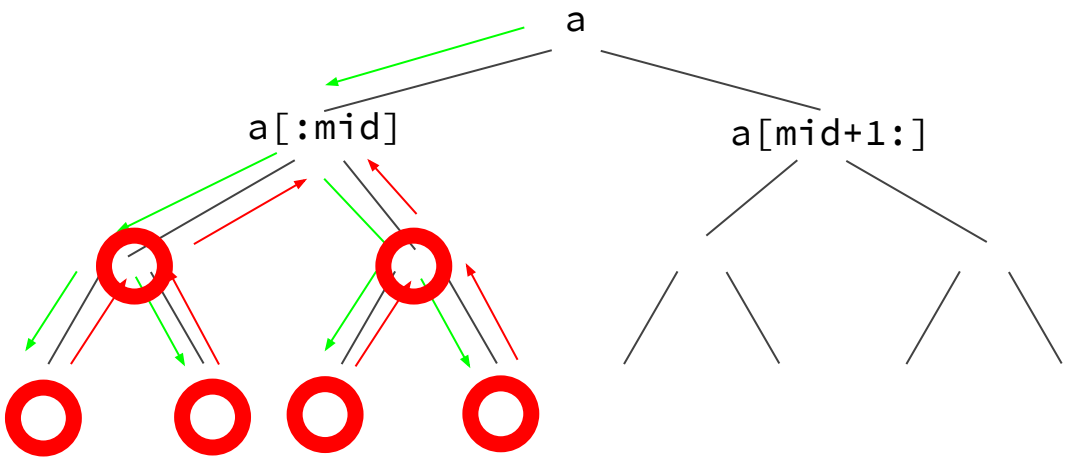
Think of recursion as a tree.



# Recursion

---

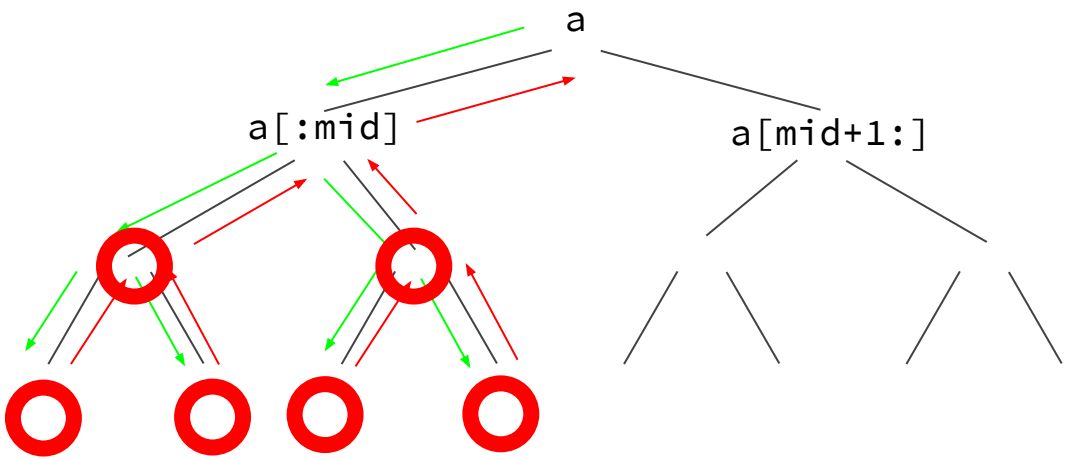
Think of recursion as a tree.



# Recursion

---

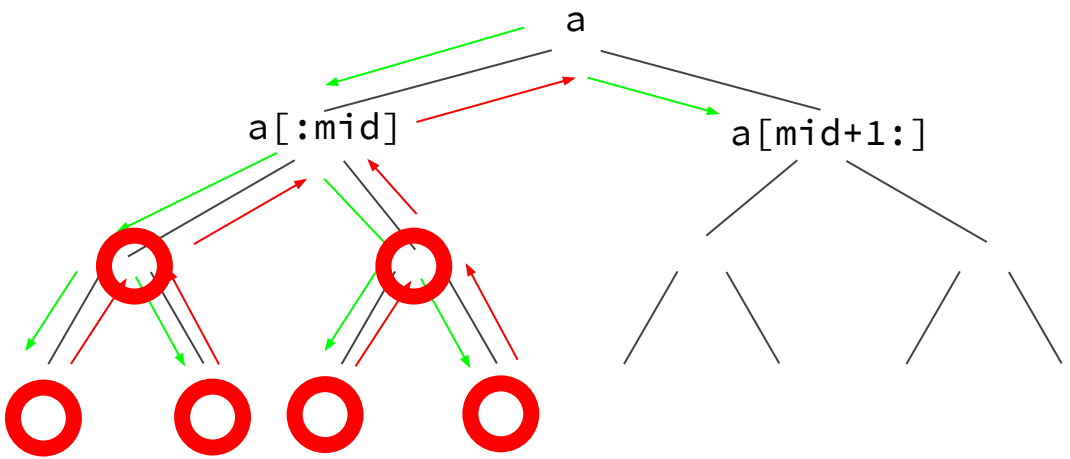
Think of recursion as a tree.



# Recursion

---

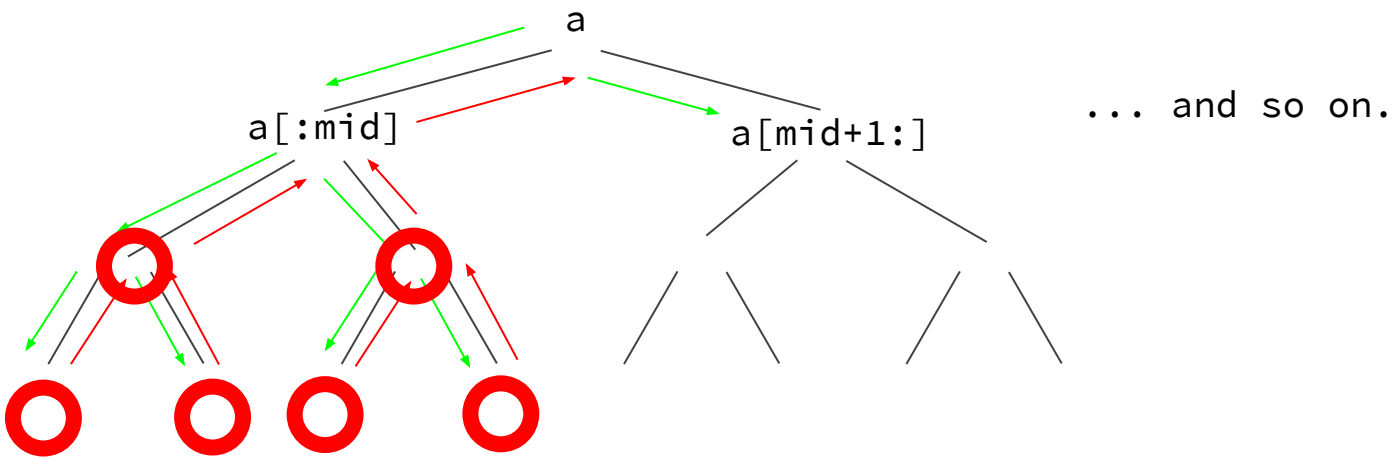
Think of recursion as a tree.



# Recursion

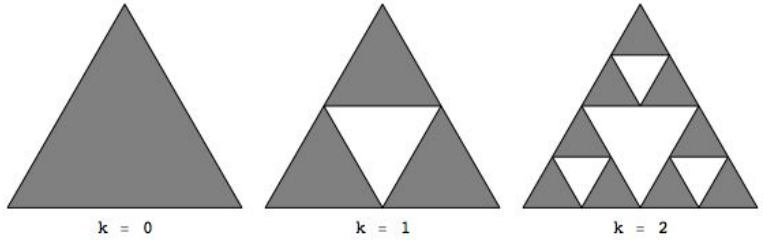
---

Think of recursion as a tree.



# Sierpinski Triangle

---




The Sierpinski Triangle is a fractal that demonstrates a three-way recursive algorithm. Draw a triangle, then divide it into four triangles by connecting the midpoints of each line. Repeat for all triangles *except the middle*.

**INPUT**

| | |
|---|---|
| points | an array of the three vertices' coordinates |
| degree | a positive integer, how many times to recur |

**OUTPUT**

Draw a Sierpinski Triangle

# Assignment 6 Problem 1

---

```
def fill_brownian(a, i0, i1, variance, scale):
    # Calculate midpoint index between a[i0] and a[i1]
    # Add random number from normal distribution to midpoint value
    # Set new variance
    # Process array slices
```

# Midpoint

---

Given indices i0 and i1, the index halfway between them is:

i0 + i1 // 2 ← **This needs to be an integer**

Given values a[i0] and a[i1], the midpoint between them is:

a[i0] + a[i1] / 2 ← **This needs to be a float**

# Normal Distribution

---

`sigma = random.gauss(mean, standard_deviation)`

Normalize using the `i0` and `i1` of this segment, i.e. add sigma to the average of `a[i0]` and `a[i1]`

Standard deviation is equal to the **square root** of the variance.

# Setting the New Variance

---

Given the Hurst exponent, H, on the command line:

Divide the original variance by $2^{2H}$

**H should never change.**

# Process Both Array Slices

---

Call fill_brownian on the array slice:

1. *before* the midpoint (inclusive)
2. *after* the midpoint (exclusive)

*Note: You don't need to change the array itself, just specify the i0 and i1*, e.g.

```
fill_brownian(a, i0, midpoint, variance, scale)
fill_brownian(a, midpoint, i1, variance, scale)
```

# Exercise: Binary Conversion

---

A binary digit, i.e. a number in base 2, can have one of two values: 0 or 1.

```
000 = 0
001 = 1
010 = 2
011 = 3
100 = 4
101 = 5
... etc
```

bin

| 1 | 0 | 1 |
|---|---|---|
| $2^2$ | $2^1$ | $2^0$ |

$4 + 0 + 1 = 5$

dec

| 5 | 3 | 2 |
|---|---|---|
| $10^2$ | $10^1$ | $10^0$ |

$500 + 30 + 2 = 532$

Given a positive integer **n** in base 10 (digit 0-9), use a recursive function to convert the decimal base10 form into a binary base2 form.
**HINT: USE STRINGS TO REPRESENT THE BINARY FORM**