INTRO TO DATA STRUCTURES / FUNCTIONAL DATA STRUCTURES IN OO-LAND

CALIBRATION (0)

Data structures?

CALIBRATION (1)

- Big-O and complexity?
- Interfaces vs. Implementation

CALIBRATION (2)

- Arrays?
- Linked Lists?
- Stacks and Queues?
- Hashing? Maps?
- Sets?

CALIBRATION (3)

- Immutability? Immutable data structures?
- Persistent data structures?
- Garbage collection?

PREFACE

- Language agnostic, so I used python
- Trigger warning for folks with Mathematics / CS Theory background (why are you here anyway?)

PRIMERS

Data Structures, Complexity, Interfaces

DATA STRUCTURES

- Data lives in memory
- How we store that data in memory is important
- Computers are fast...but not that fast

BIG-O

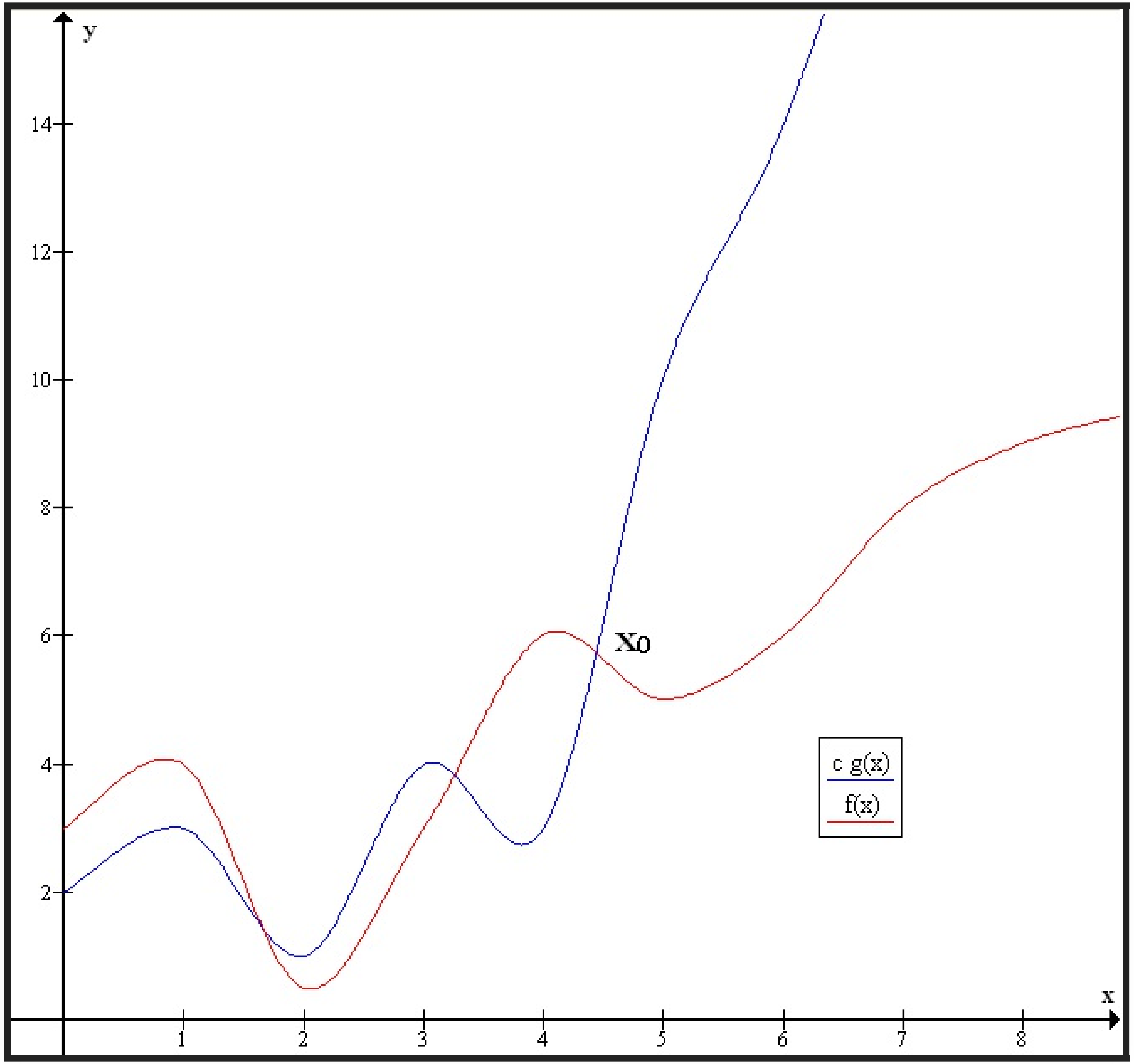
$$f(x) = O(g(x))$$

if and only if

$$|f(x)| \le M|g(x)|$$

for all

$$x \ge x_0$$


y
14
12
10
8
6
4
2
x
1
2
3
4
5
6
7
8
X0
c g(x)
f(x)

COMPLEXITY

$O(1)$

```
x = 5
```

$O(n)$

```
x = [1, 2, 3, 4, 5]
for i in x:
    print(i * 2)
```

$O(n^2)$

```
x = [1, 2, 3, 4, 5]
for i in x:
    for j in x:
        print(i + j)
```

$O(\log n)$

Binary search

$O(n \log n)$

Most (practical) sorting algorithms

INTERFACES VS IMPLEMENTATIONS

- Interfaces: what a data structure does.
- Implementation: how a data structure does it

COMMON DATA STRUCTURES

LIST INTERFACE

- Support add(item, i), remove(i), get(i), set(item, i)
- Could use an Array, but what happens when you (1) add items (2) run out of space in the Array?
- Array backed lists

```
[ ][ ][ ][ ][ ][ ][ ][ ][ ][ ]
[a][b][c][ ][ ][ ][ ][ ][ ][ ]
[a][b][c][d][ ][ ][ ][ ][ ][ ]
[e][a][b][c][d][ ][ ][ ][ ][ ]
```

LINKED LIST

```
class Node:
    def __init__(self, data, next):
        self.data = data
        self.next = next
```

```python
class LinkedList:
    def __init__(self):
        self.head = Node(None, None)
        self.size = 0

    def add(self, item, index):
        current = self.head
        for i in range(0, index):
            current = current.next
        node = Node(item, current.next)
        current.next = node
        self.size = self.size + 1
```

DOUBLY LINKED LIST

```
class Node:
    def __init__(self, data, prev, next):
        self.data = data
        self.prev = prev
        self.next = next
```

```python
class LinkedList:
    def __init__(self):
        self.head = Node(None, None, None)
        self.tail = Node(None, self.head, None)
        self.head = self.tail
        self.size = 0

    def add(self, item, index):
        current = self.head
        for i in range(0, index):
            current = current.next
        node = Node(item, current.prev, current)
        node.prev.next = node
        current.prev = node
        size += 1
```

STACK AND QUEUE INTERFACES

- Stack: Last In First Out (LIFO)
  - Can be implemented using a Linked List
- Queues: First In First Out (FIFO)
  - Can be implemented using a Doubly Linked List

MAP INTERFACE

- A dictionary! Supports `get(key)` and `set(key, value)`
- Sketch of an implementation of a hash map
- Hashing and keys

SET INTERFACE

- A dictionary (without values). Supports `contains(key)` and `add(key)`
- HashSet is a common implementation

FUN STUFF

Functional Data Structures!

> *You can have any data structure you want as long as it's imperative - Not Henry Ford*

MOTIVATING 'WAT'S

```
public class Rectangle implements Comparable<Rectangle> {
    private int width, height;
    public Rectangle(int width, int height);

    public int getWidth();
    public int getHeight();
    public void setWidth(int width);
    public void setHeight(int height);

    public int hashCode();
    public boolean equals(Object o);
    public int compareTo(Rectangle r);
}
```

```java
Set<Rectangle> s = new HashSet<Rectangle>();
Rectangle r = new Rectangle(2, 3);
s.add(r);

r.setWidth(5);
s.size(); // 1
s.contains(r) // AHA false

// even more fun...
s.add(r); // s = [Rectangle(5, 3), Rectangle(5, 3)]
```

JAVA'S SOLUTION

```
Set<Rectangle> s = new HashSet<Rectangle>();
Rectangle r = new Rectangle(2, 3);
Set<Rectangle> sUnmodifiable = Collections.unmodifiableSet(r);
sUnmodifiable.add(new Rectangle(3, 4)); // throws
```

You could also always create new sets, do a deep-copy. Still icky.

FUNCTIONAL DATA STRUCTURES

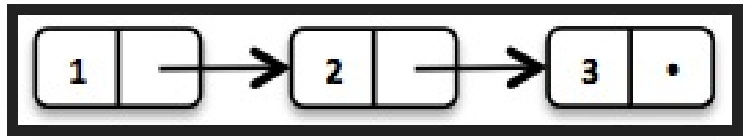
WHAT'S GOOD?

- Immutability (no assignments)
- Persistence

HOW TO BUILD A LIST

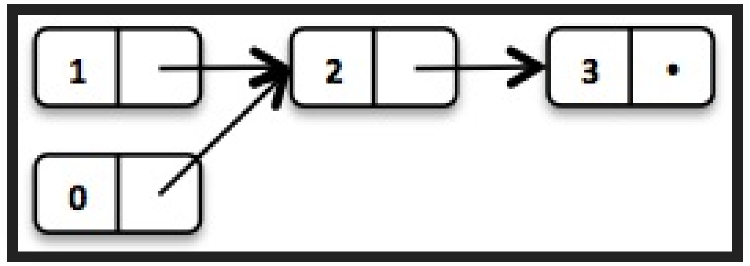
```
final class Cons<T> implements List<T>{
    private final T head;
    private final List<T> tail;
    private final int length;

    public static List<T> of(int... items) {
        // factory method that supports e.g.
        // List<Integer> list1 = List.of(1, 2, 3);
    }
}
```



HOW TO PREPEND A LIST

```
@Override
default List<T> prepend(T element) {
    return new Cons<>(element, this);
}
```



HOW TO FOLD A LIST

```
@Override
default <U> U foldLeft(U zero, BiFunction<? super U, ? super T, ? extends U> f) {
    U xs = zero;
    for (T x : this) {
        xs = f.apply(xs, x);
    }
    return xs;
}
```

HOW TO REVERSE A LIST

```
@Override
default List<T> reverse() {
    return (length() <= 1)
        ? this
        : foldLeft(empty(), List::prepend);
}
```

HOW TO FOLD (RIGHT) A LIST

```
@Override
default <U> U foldRight(U zero, BiFunction<? super U, ? super T, ? extends U> f) {
    return reverse().foldLeft(zero, f);
}
```

HOW TO APPEND TO A LIST

```
@Override
default List<T> append(T element) {
    return foldRight(Cons.of(element), (x, xs) -> xs.prepend(x));
}
```

COMPLEXITY?

- prepending is cheap
- foldLeft (as with all foldLefts) are linear
- reverse is expensive, linear
- foldRight is expensive (linear + linear = linear)
- append is expensive linear

How do we think about complexity in this case?

COMPLEXITY?

```
Cons cons = Cons.of(1);
cons.prepend(2);
cons.prepend(3);
...
cons.prepend(n);
cons.reverse();
```

COMPLEXITY?

```
Cons cons = Cons.of(1); // O(1)
cons.prepend(2); // O(1)
cons.prepend(3); // O(1)
...
cons.prepend(n); // O(1)
cons.reverse(); // O(n)
```

BANKER'S METHOD

```
Cons cons = Cons.of(1); // O(1 + x)
cons.prepend(2); // O(1 + x)
cons.prepend(3); // O(1 + x)
...
cons.prepend(n); // O(1 + x)
cons.reverse(); // O(n)
```

Total:

$$O(2N + Nx) = O(N(2 + x))$$

Per Operation:

$$O(1 + x)$$

HOWEVER...

You can't stop people from abusing foldRight, reverse and prepend. e.g. Using the Cons List as a Queue. What can you do?

- Strict evaluation
- Lazy evaluation without memoization
- Lazy evaluation with memoization

OKAY...BUT CAN I HAZ DATA STRUCTURE?

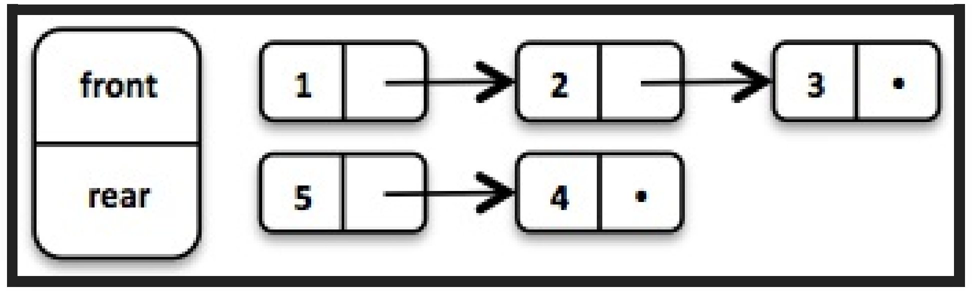
So how do we actually build a Queue?

HOW TO BUILD A QUEUE (BASELINE)

```
public final class Queue<T> {
    private final List<T> queue;
}
```

HOW TO BUILD A (BETTER) QUEUE

```
public final class Queue<T> {
    private static final Queue<?> EMPTY = new Queue<>(List.empty(), List.empty());
    private final List<T> front; // or front
    private final List<T> rear; // or rear
}
```



HOW TO ENQUEUE

```
@Override
public Queue<T> enqueue(T element) {
    return new Queue<>(front,rear.prepend(element));
}
```

HOW TO TAIL

```
@Override
public Queue<T> tail() {
    return new Queue<>(front.tail(), rear);
}
```

HOW TO PEEK

```
public T peek() {
    if (isEmpty()) {
        throw new NoSuchElementException("empty");
    } else {
        return front.head();
    }
}
```

HOW TO DEQUEUE

```
Queue queue = Queue.of(1, 2, 3);
// = (1, Queue(2, 3))
Tuple2<Integer, Queue> dequeued = queue.dequeue();
```

```
@Override
public Tuple2<T, Q> dequeue() {
    if (isEmpty()) {
        throw new NoSuchElementException("empty");
    } else {
        return Tuple.of(head(), tail());
    }
}
```

WHAT IF FRONT IS EMPTY?

```
private Queue(List<T> front, List<T> rear) {
    final boolean frontIsEmpty = front.isEmpty();
    this.front = frontIsEmpty ? rear.reverse() : front;
    this.rear = frontIsEmpty ? front : rear;
}
```

Isn't reverse() expensive? (hint: amortized!)

WHY'S GARBAGE COLLECTION IMPORTANT

```
public Queue<Integer> someOperation(Queue<Integer> a) {
    // a = 1 2 3
    b = a.enqueue(4); // b = 1 2 3 4
    c = b.enqueue(5); // c = 1 2 3 4 5
    z = a.enqueue(6); // d = 1 2 3 6

    return z;

    // 4 and 5 are no longer being used. need to GC!!!
}
```

WHAT'S THE POINT OF THIS?

- Common perils of mutable data structures
- Immutable data structures as an unicorn
- Functional programming is not a cult -- it's a way of being