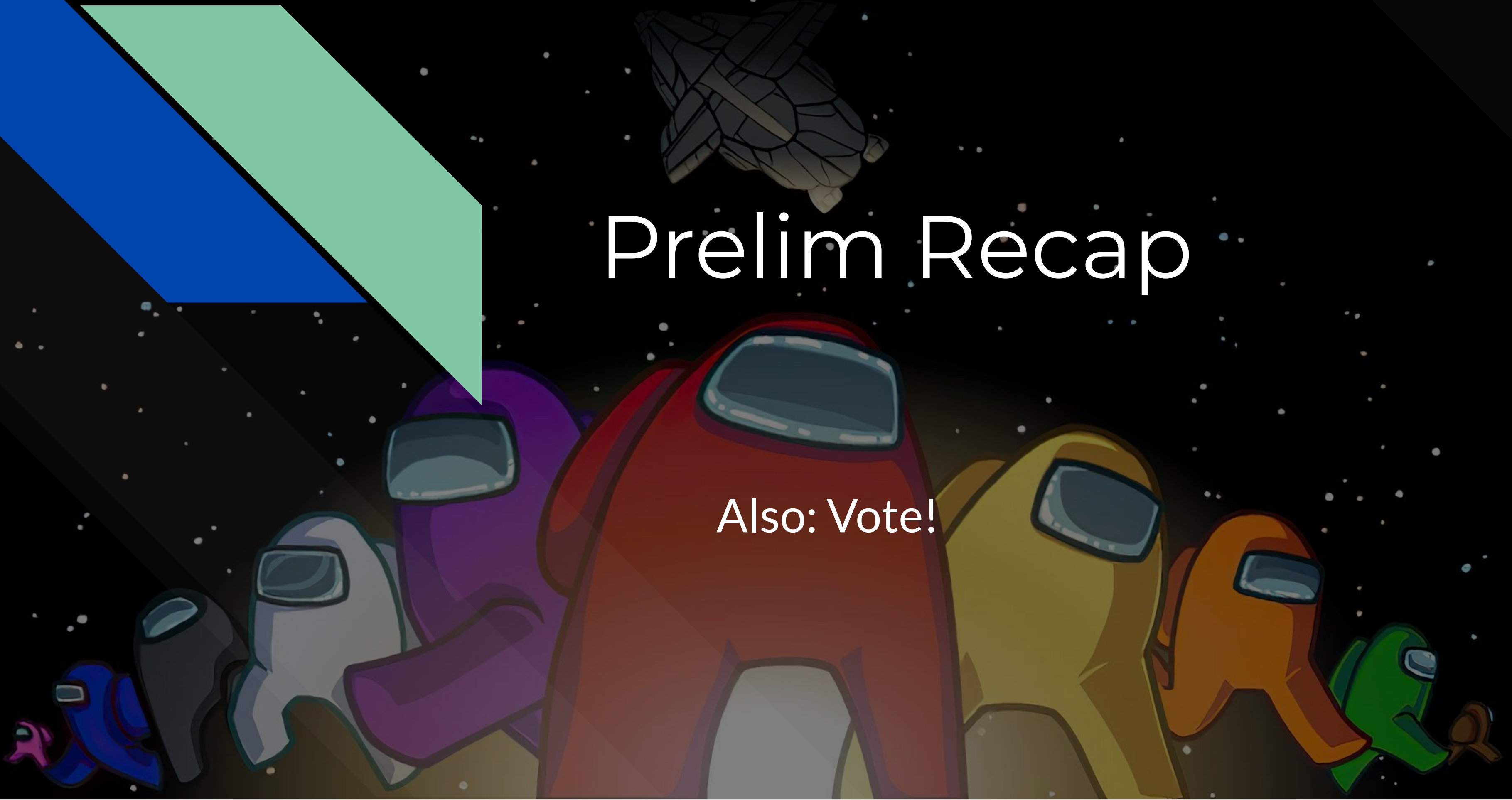
Prelim Recap

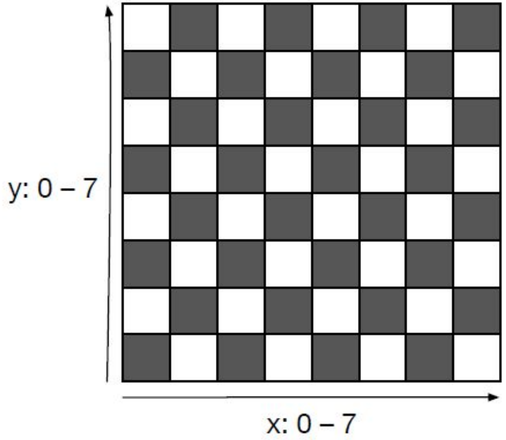
Also: Vote!



Problem 2

2. **Designing and implementing abstractions** [31 pts] (parts a–g)

The game of checkers is played on an 8×8 board in which only the black squares, those whose coordinates sum to an even number, can hold a piece. Ignoring the possibility of kings for simplicity, a black square can either be empty, contain a black piece, or contain a white piece. All pieces of a given color are indistinguishable from each other.



Problem 2

```java
public class Board {
    /** cells is an 8x8 array that may contain nulls to represent
     *  empty squares.
     */
    private Piece[][] cells = new Piece[8][8];

    public Piece get(int x, int y) {
        return cells[x][y];
    }
    public Piece[][] getCells() {
        return cells;
    }
}
```

```java
/** An immutable checkers piece. */
public class Piece {
    private boolean black;
    public Piece(boolean isBlack) { black = isBlack; }
    ...
}
```

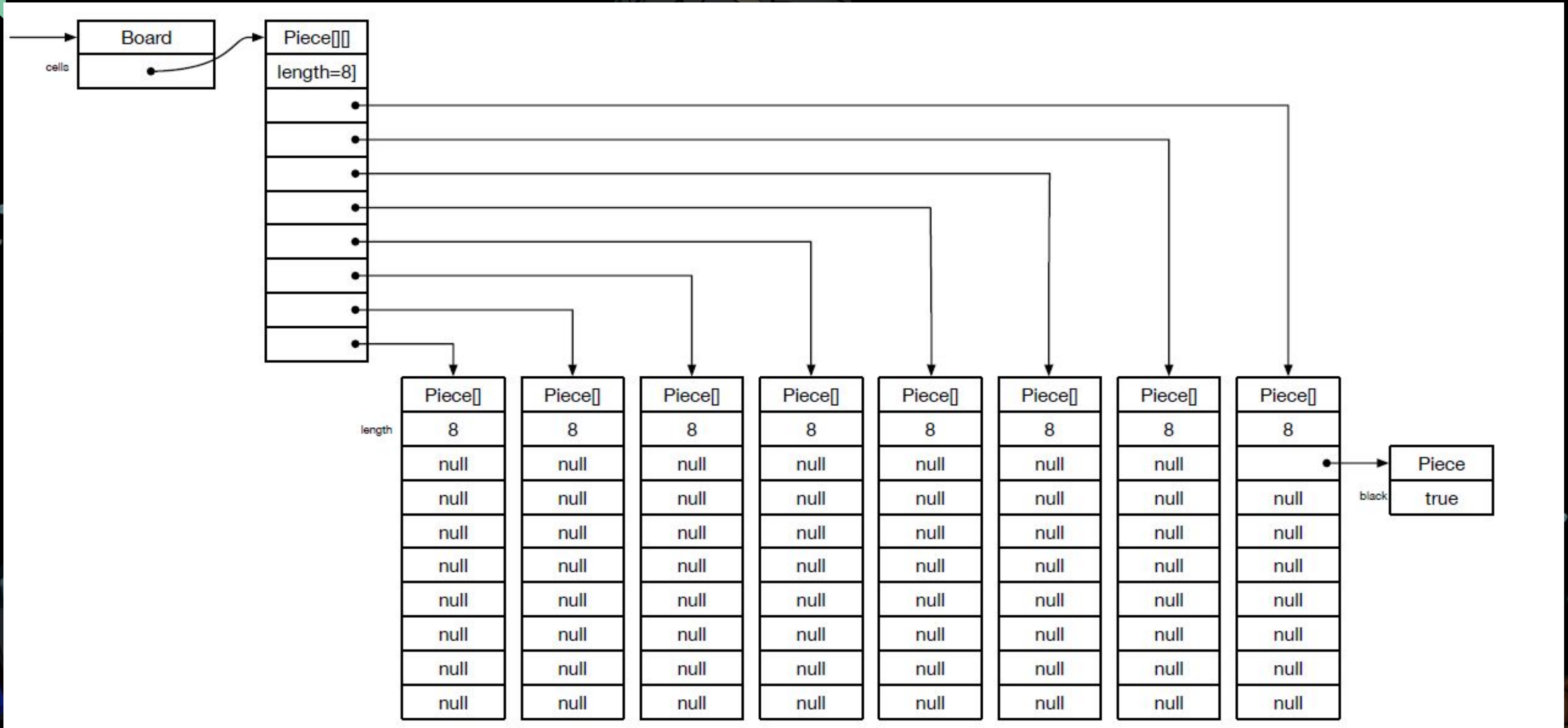
Problem 2, part a)

(a) [5 pts] Draw an object diagram for the data structure referenced by variable b after the statements `b = new Board(); b.getCells()[0][0] = new Piece(true);`. You can abbreviate the diagram as long as it is clear.

Things we need to represent

- Board
- An array of arrays
- Empty array vs. null elements
- One Piece plus its instance variable

Problem 2, part a)



Problem 2, part b)

(b) [2 pts] About how many bytes of memory space does this particular data structure take up? Show any calculations; we will not require you to be perfectly accurate or to consider all possible data structures that a Board could refer to.

What types of things take up memory?

- Primitives
- Pointers
- Space for arrays
- Object headers

Problem 2, part b)

How much memory does a pointer take up?

It depends if you use a 32-bit or 64-bit machine.

Let’s assume we have a 64-bit machine.

Problem 2, part b)

Each array needs

Recall: a word is 4 bytes

2 + 1 + 8 x 2 = 19 **words**

Object Header

Length

Space for each of the 8 pointers

Problem 2, part b)

Now for the objects

Piece:

2 + 1 = 3 **words**

Object Header

boolean

Board:

2 + 2 = 4 **words**

Object Header

Pointer to cells

Problem 2, part b)

In total,

Recall: a word is 4 bytes

$$\underbrace{19}_{\text{All 9 arrays}} \times 9 + \underbrace{3}_{\text{Piece}} + \underbrace{4}_{\text{Board}} = 178 \text{ words} = 712 \text{ bytes}$$

Problem 2, part c)

```java
public Piece get(int x, int y) {
    return cells[x][y];
}
```

What should our specification tell the user?

- Input, output
- How we handle edge cases
- Requirements, if any

Problem 2, part c)

Returns the piece stored at location (x, y) on the board, or null if the location is empty. Requires: x and y are both in range (0–7) and their sum is even.

There's more than one way to write this!

Problem 2, part d)

(d) [4 pts] Currently the client code modifies the board by calling `getCells()` and then assigning into elements of the underlying array. Briefly discuss the problems with this design.

- Straightforward example of rep exposure
  - The user could ruin our representation!

Problem 2, part e)

How do we fix part d)?

- ~~Make a deep copy?~~
  - Unnecessary; there is already a get method!
  - Won't reflect future updates to the board
- Setter method or add and remove methods
  - Allows user to modify the board without revealing the underlying representation.

Problem 2, part f)

What is a more space efficient way to represent our board?

What takes up space unnecessarily?

- Empty tiles
- Piece objects
- Booleans!

Problem 2, part f)

How many tiles do we actually need to represent?

- 32

How many states do we need to encode for any given tile?

- 3 - empty, white piece, or black piece

Problem 2, part f)

Idea: use a set of bits to represent the state for each tile.

How do we represent our bits?

- Byte array
- Bitset
- long

Problem 2, part f)

```
// Each consecutive pair of bits in cellBits represents a black
// square on the board, containing either 0 (empty), 1 (black
// piece), or 2 (white piece).
//
//          ...
// -- 89 -- ab -- cd --  ef
// 01 -- 23 -- 45 -- 67 --
private long cellBits;
```

Note: we are encoding 4 states per tile. We could improve on this by using base 3, but that would be pretty hard.

Problem 2, part g)

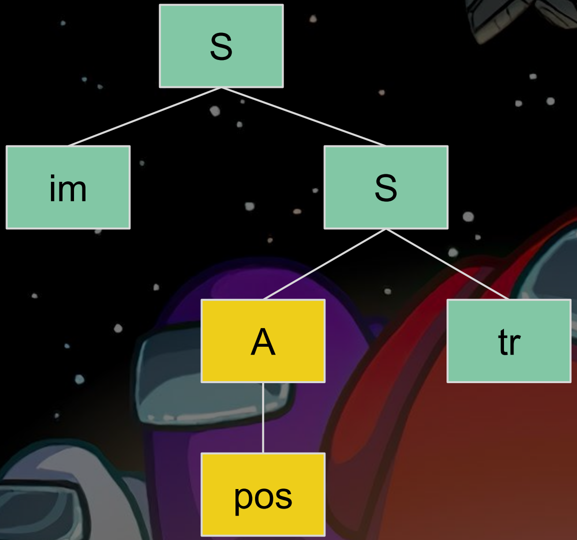
```java
// Returns: bit position of cell (x,y)
private int bitPos(int x, int y) {
    assert ((x + y) % 2 == 0);
    int i = y * 8 + (x & -2);
    return  i;
}

/** Returns: the piece at location (x, y) on the board, or null
 *  if that location is empty.
 *  Checks: (x,y) is a legal location for a piece.
 */
public Piece get(int x, int y) {
    switch ((int)(cellBits >> bitPos(x, y)) % 4) {
        case 0: return null;
        case 1: return new Piece(true);
        case 2: return new Piece(false);
        default: assert false;
    }
}
```

Problem 3

(a) im pos tr

S -> A tr | im S

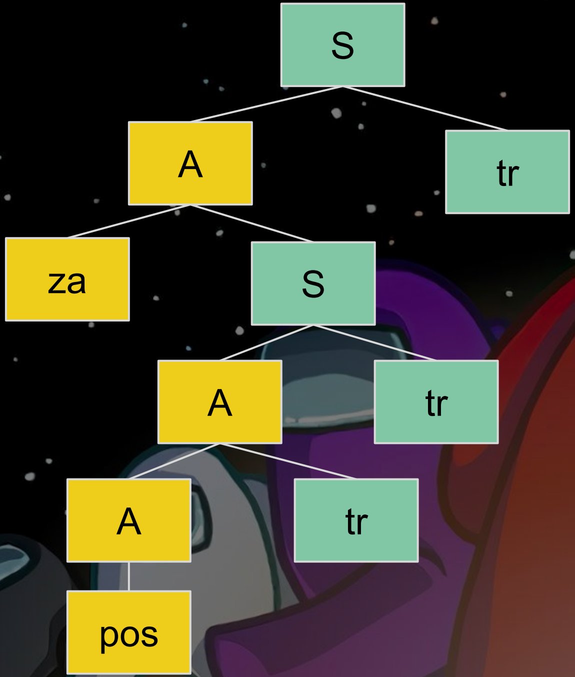
A -> A tr | za S | pos

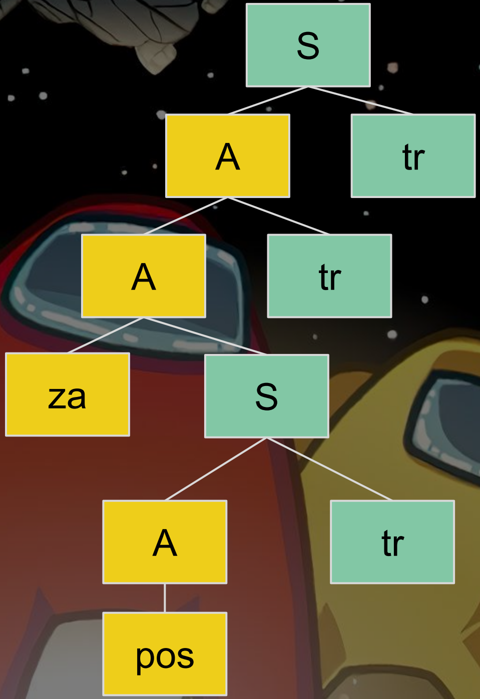


(b) za pos tr tr tr

S -> A tr | im S

A -> A tr | za S | pos





The grammar is *suspiciously* ambiguous!

(c)
Issue with the grammar

S -> A tr | im S

A -> A tr | za S | pos

A -> A tr is sus

```
A parseA() {
    parseA(); // uh oh, infinite recursion
    consume(); // should consume 'tr'
}
```

(d) Grammar fix

S -> A tr | im S

A -> A tr | za S | pos

We want to get rid of A -> A tr **without changing the language**.

The best way to avoid changing the language is to change as little as possible.

(d) Grammar fix

S -> A `tr` | `im` S

A -> A `tr` | `za` S | `pos`

A -> `za` S | `pos` behaves like base case. (no self-reference)

The A -> A `tr` rule allows you to append 0 or more `tr` at the end, like some suffix

(d) Grammar fix

S -> A `tr` | `im` S

A -> A `tr` | `za` S | `pos`

Write a rule that let you have 0 or more `tr` at the end.

T -> tr T | ε

(d) Grammar fix

Append T!

A -> za S T | pos T

S -> A tr | im S

A -> za S T | pos T

T -> tr T | ε

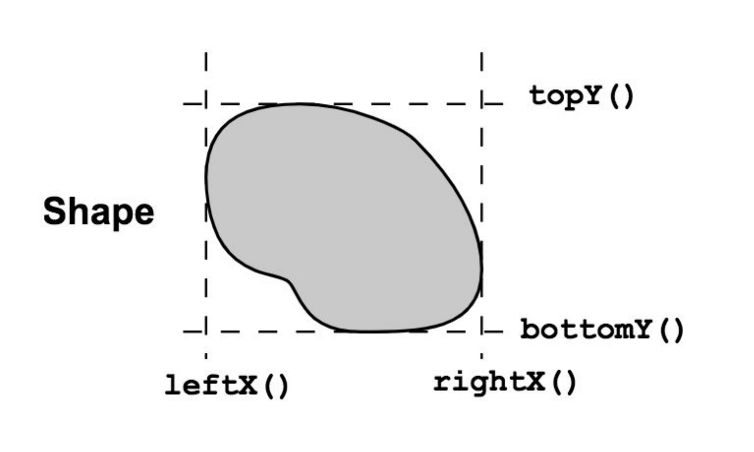
Problem 3 Common Mistakes

- (b) Start with A rule. We specified that S is the start symbol.
- (c) Thinking that S -> A tr and A -> A tr introduced ambiguity.
  - We always know which production rule we are current in!
- (d) Trying to develop some intuition on the grammar, then try to create a new grammar that matches the intuition.
  - It’s very likely that your intuition is slightly wrong.

Problem 4: Inheritance

Suppose we have an interface `Shape` that is part of a geometry package. It has the following operations:

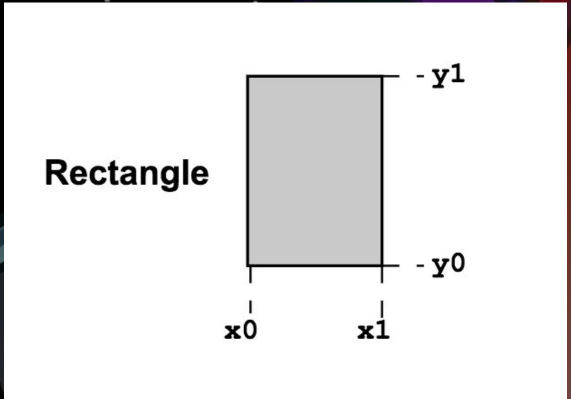
```
/** A two-dimensional shape on the plane, with Cartesian coordinates. */
interface Shape {
  // These four methods give coordinates that bound the shape in a
  // rectangle.
  double leftX();
  double rightX();
  double bottomY();
  double topY();
  /** The area of the shape. */
  double area();
}
```

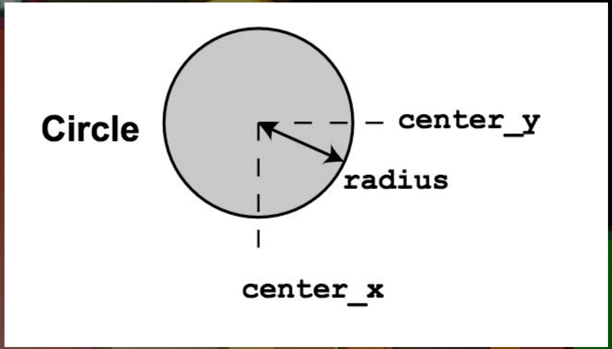


Problem 4: Inheritance

```java
class Rectangle implements Shape {
  private double x0, x1, y0, y1;
  public Rectangle(double x0, double x1, double y0, double y1) {
    this.x0 = x0; this.x1 = x1; this.y0 = y0; this.y1 = y1;
  }
  public double leftX() { return x0; }
  public double rightX() { return x1; }
  public double bottomY() { return y0; }
  public double topY() { return y1; }
  public double area() { return (x1 - x0) * (y1 - y0); }
}
```

```java
class Circle implements Shape {
  private double center_x, center_y, radius;
  public Circle(double cx, double cy, double r) {
    center_x = cx; center_y = cy; radius = r;
  }
  public double leftX() { return center_x - radius; }
  public double rightX() { return center_x + radius; }
  public double bottomY() { return center_y - radius; }
  public double topY() { ... }
  public double area() { return Math.PI*radius*radius; }
}
```



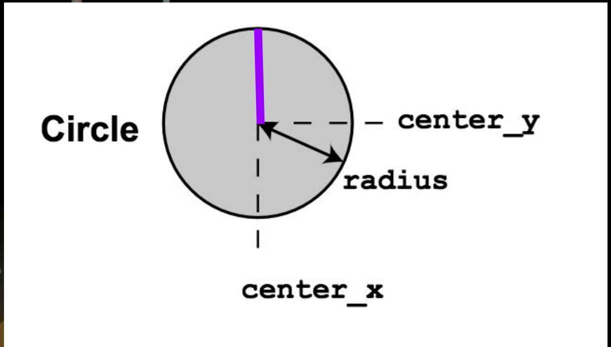


Problem 4: Inheritance

Implement `topY` for `Circle`

```
class Circle implements Shape {
  private double center_x, center_y, radius;
  public Circle(double cx, double cy, double r) {
    center_x = cx; center_y = cy; radius = r;
  }
  public double leftX() { return center_x - radius; }
  public double rightX() { return center_x + radius; }
  public double bottomY() { return center_y - radius; }
  public double topY() { ... }
  public double area() { return Math.PI*radius*radius; }
}
```

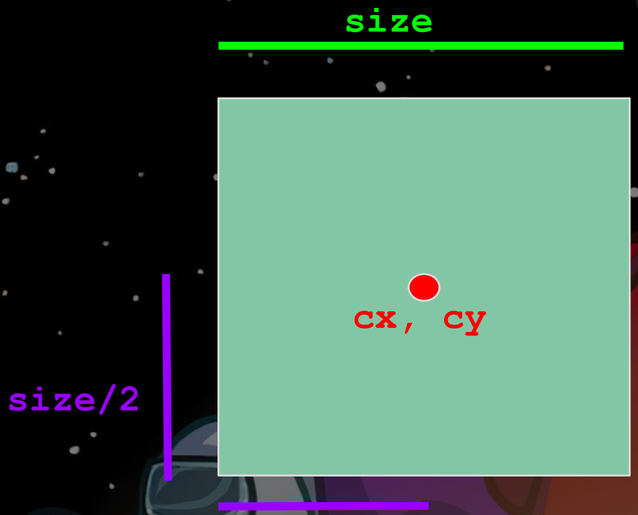
Top y: center_y + radius




```
public double topY() { return center_y + radius; }
```

Problem 4: Inheritance

Create a constructor for a class `Square` with constructor `Square(double cx, double cy, double size)`



- `rightX = cx + size / 2`
- `leftX = cx - size / 2`
- `topY = cy + size / 2`
- `bottomY = cy - size / 2`

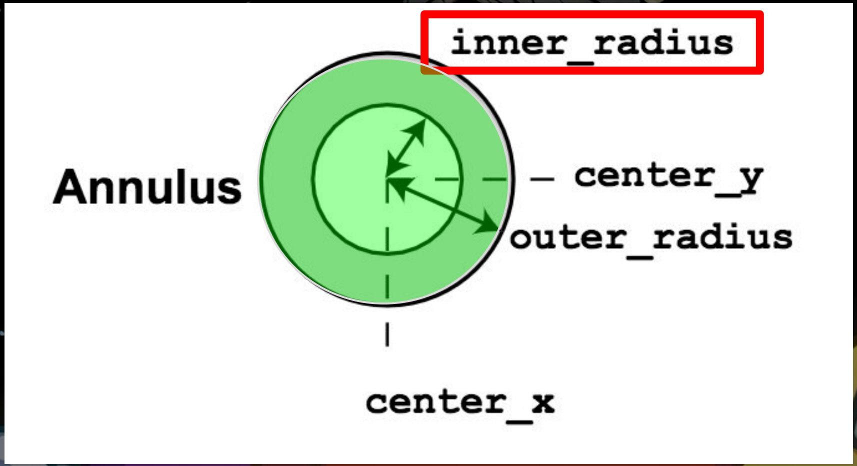
```
Square(double cx, double cy, double size) {
  super(cx - size / 2, cx + size / 2, cy - size / 2, cy + size / 2)
}
```

Problem 4: Inheritance

[6 pts] Now, suppose we also want to implement an annulus, the doughnut-shaped region lying between two concentric circles. Complete the implementation below by giving code for $\boxed{1}$–$\boxed{3}$. Be sure to consider what methods, if any, of `Circle` need to be overridden.

```
class Annulus extends Circle {
[1]
    Annulus(float cx, float cy,
            float outer_radius, float inner_radius) {
        super(cx, cy, outer_radius);
        [2]
    }
    [3]
}
```

Problem 4: Inheritance



Annulus is like Circle, but needs more information: an inner_radius, and a different area() calculation

Problem 4: Inheritance

```java
class Circle implements Shape {
  private double center_x, center_y, radius;
  public Circle(double cx, double cy, double r) {
    center_x = cx; center_y = cy; radius = r;
  }
  public double leftX() { return center_x - radius; }
  public double rightX() { return center_x + radius; }
  public double bottomY() { return center_y - radius; }
  public double topY() { ... }
  public double area() { return Math.PI*radius*radius; }
}
```

```java
class Annulus extends Circle {
    double inner_radius;
    Annulus(double cx, double cy,
            double outer_radius, double inner_radius) {
        super(cx, cy, outer_radius);
        this.inner_radius = inner_radius;
    }
    double area() {
        return super.area() - PI*inner_radius*inner_radius;
    }
}
```

Annulus is like Circle, but needs more information: an inner_radius, and a different area() calculation (also, Circle's radius is private/inaccessible to subclasses!)

Problem 5: Hash Tables
Part (a)

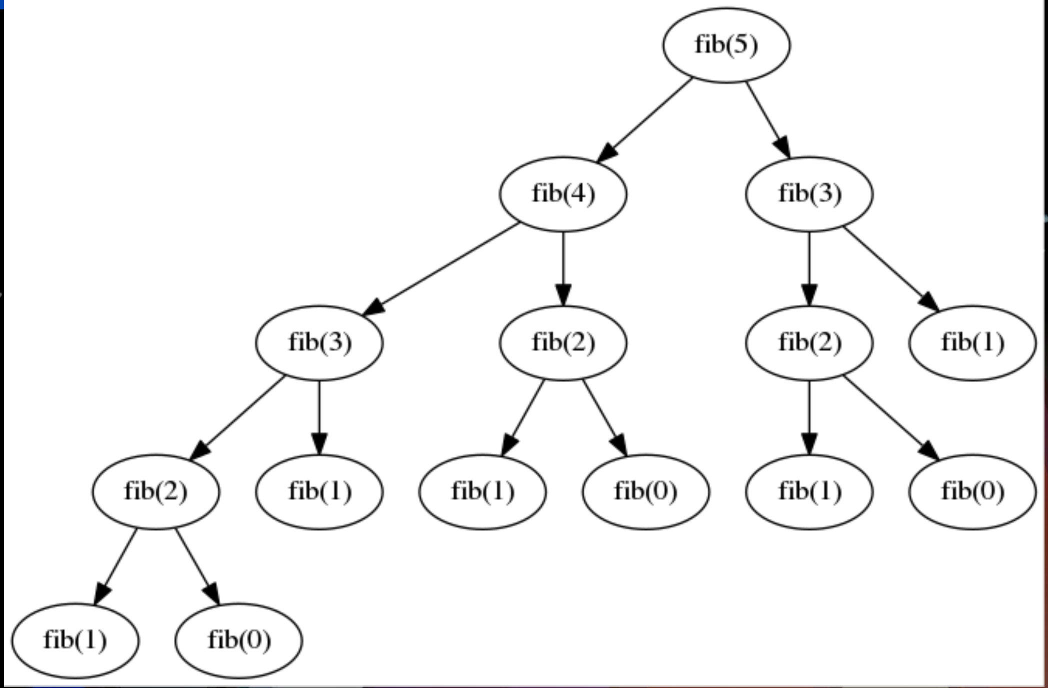
```java
MemoTable<Integer, Integer> memo = new MemoTable<>();
int fibo(int n) {
    return memo.get(n) // do we already know it?
        .orElseGet(() -> { // otherwise compute the slow way
            int result;
            if (n < 2) result = n;
            else result = fibo(n - 1) + fibo(n - 2);
            memo.put(n, result); // remember for next time
            return result;
        });
}
```

With perfect memoization, each call to fibo(...) will only need to be computed once, before becoming an O(1) lookup.

Thus, fibo(n) is

O(n)

To see this:



For fib(5), we only need to compute fib(5), since we can look up fib(4), fib(3), fib(2), fib(1), each of which only takes O(1)

Problem 5: Hash Tables
Part (b) bucket()

The spec for bucket() required the bucket an object ***would*** be put into, meaning it must be a hash function, not a search for a key that already exists.

We accepted any reasonable hash function (default, Message Digest, multiplicative, etc.)

Common Mistake: k.hashCode() could potentially be negative!

```java
private int bucket(K k) {
    return Math.abs(k.hashCode()) % MEMO_TABLE_SIZE;
}
```

Problem 5: Hash Tables
Part (b) `put()`

Since we forget previous bindings, chaining and probing are both unnecessary.
Just update the new key and value into the right bucket.

```
public void put(K k, V v){
    int i = bucket(k);
    keys[i] = k;
    values[i] = v;
}
```

Problem 5: Hash Tables
Part (b) `get()`

```java
public Optional<V> get(K k) {
    int i = bucket(k);
    if (k.equals(keys[i])) {
        return Optional.of(values[i]);
    } else {
        return Optional.empty();
    }
}
```

Get the binding if it exists.

Common Mistakes:

- You must remember to check if the key in the bucket is the correct one before returning
- keys[i] is null for empty buckets; calling keys[i].equals(...) would throw a null pointer exception

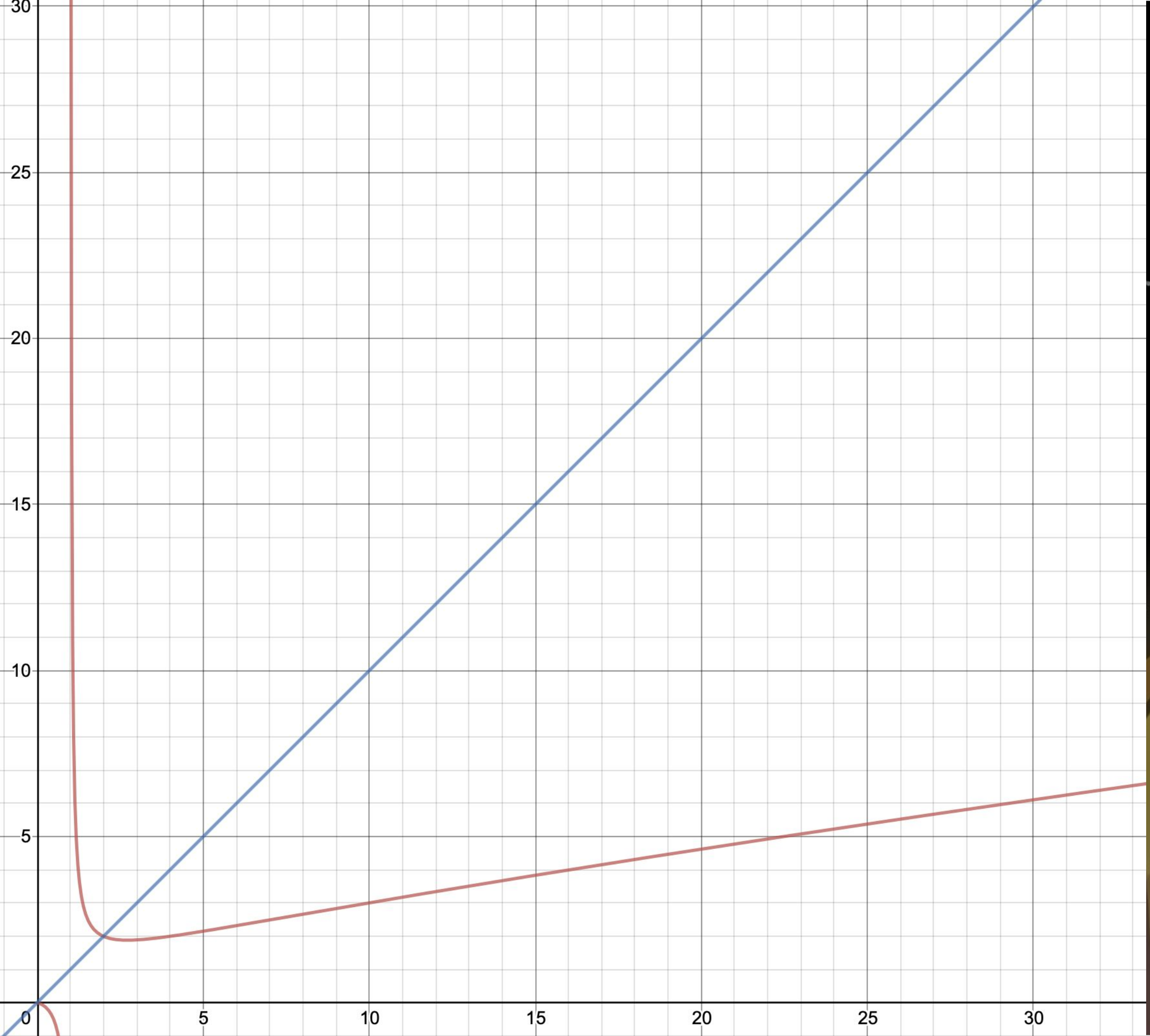
Problem 7a: Big O

(a) Show that $\frac{n}{lgn}$ is $O(n)$ by providing a witness pair

Recall the definition of Big O:

$$O(g) = \{f | \exists c > 0, \exists n_0 > 0 \text{ s.t.} \forall n \geq n_0, f(n) \leq cg(n)\}$$

Note the way the logic of the statement works: there are 2 there exist statements, and one for all statement.
For exist clauses, all we need to do is show that one of them is correct.
These 2 exist statements are our witness pairs.



Looking at the graph: choose (n_0 = 2, c=1)

Proving for all n:

We want to show that $\forall n \geq n_0, f(n) \leq cg(n)$

Substitute our values:

We want to Show: $\forall n \geq 2, \frac{n}{lgn} \leq n$

The key part of this argument is that we need to **show (i.e. prove)** that this statement is true for all possible values of n >= 2

Finishing the Proof

Consider the function lg(n).
Note that lg(n) is strictly increasing and that $\forall n \geq 2, \lg(n) \geq 1$
Therefore, $\forall n \geq 2,\ \frac{n}{lgn} \leq n$, since $\lg(n) \geq 1$
We can therefore conclude that $\frac{n}{lgn}$ is $O(n)$

By showing the for all statement to be true for all n >= 2, we have satisfied the definitions
Of Big O and therefore we can conclude our proof.

Note that simply stating the witness pairs is not enough, because Big O requires the relation
to hold for all n sufficiently large.

You also need to **show**, not just state that the statement is true.

Question 8: Loop Invariants

```
int checkOrdering(int[] a) {
    boolean ascending = true, descending = true;
    int previous = 0;
    for (int i = 0; i < a.length; i++) {
        if (i == 0) {
            previous = a[0];
        } else {
            if (a[i] < previous) ascending = false;
            if (a[i] > previous) descending = false;
            previous = a[i];


        }
    }
    int result = 0;
    if (ascending) result += 1;
    if (descending) result += 2;
    return result;
}
```

The key here is to understand how the algorithm works.

Loop invariants represent core reasons why the algorithm is correct.

Loop Invariants A

(b) Either ascending or descending is true.

F

(c) $\mathtt{i} = 0$

F

(d) $\mathtt{i} < \mathtt{a.length}$

F

(e) $\mathtt{i} \geq 0$

T

(f) `previous` = `a[i-1]` or `previous = 0`.

T

(g) `previous` = `a[i-1]` or `i = 0`.

T

B. If we have elements that are not sorted, then both ascending, descending are false

C. i is in the loop, will increase

D. Note that when the loop terminates, i = a.length

E. Clearly true

F / G: Both true

(h) If `ascending = true`, then all elements in a are sorted in ascending order.

F

(i) If `ascending = true`, then the elements `a[0..i-1]` (if any) are sorted in ascending order.

T

(j) If `descending = true`, then elements `a[0..min(i, a.length-1)]` (if any) are sorted in descending order.

T

(k) If `descending = true`, then elements `a[i..a.length-1]` (if any) are sorted in descending order.

F

H: ascending is true iff. all previous elements are sorted. If i=2 and array has length 5, we have not yet checked full array
I: Key part of the algorithm
J: This is false; needs to be i-1.
K: Wrong for same reason as H.

Problem 1: True / False

(a) The type Set<Integer> is a subtype of Set<? extends Object>

TRUE: You can use Set<Integer> as a Set<? extends Object> since Integer extends Object

(b) An assignment o.v = x; where the declared type of v is int, can throw an exception

TRUE: x could be null if it is of type Integer, since Java will typecast Integer to int

(c) For a hash table to find keys correctly, the hash function must guarantee that two keys have the same hash code whenever the keys are equal.

TRUE: Otherwise the hash table wouldn't find the correct bucket

Problem 1: True / False

(d) The Integer class defines a mutable data abstraction.

FALSE: No autoboxed type is mutable

(e) For large collections, binary search trees are usually faster than hash tables when searching for a key.

FALSE: BSTs are usually O(log n) while hash tables are usually O(1)

(f) A node in a binary tree can have 0, 1, or 2 children, and 0 or 1 parents.

TRUE: All nodes in trees have 0 or 1 parents, and the definition of a BST caps 2 children