MIT
6.031 Software Construction
Prof. Rob Miller and Max Goldman

6.031 Fall 2021 Quiz 1

You have **50** minutes to complete this quiz. There are **5** problems.

The quiz is **closed-book** and closed-notes, but you are allowed one two-sided page of notes on paper. You may not open or use anything else on your computer: no 6.031 website or readings; no VS Code, TypeScript compiler, or programming tools; no web search or discussion with other people.

Before you begin: you must enter a *check-in code*, provided when you arrive, in the green box above. By checking in, you indicate that you will take this quiz under closed-book conditions, and you will not discuss this quiz with anyone other than 6.031 staff members until the solutions are officially released.

To leave the quiz early: click the *done* button at the very bottom of the page and follow instructions to give the provided *check-out* code to a staff member.

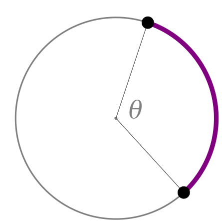
This page automatically saves your answers as you work. Saved answers are marked with a green cloud-with-up-arrow icon. If you see a stuck yellow spinner, red exclamation mark, or a red notification that you are disconnected, your answers are not being saved: try reloading the page right away, before continuing to work on the quiz. There is no *save* or *submit* button.

Only the answers you write in boxes or select from choices will be graded. Purple margin notes are for your use only during the quiz, and will not be seen by the graders. To ask a question, please raise your hand.

Good luck!

Problem	Points
1	20
2	16
3	22
4	20
5	22

Archimedes is very excited to start programming his new laser cutter. The first abstraction he wants is a way to represent *circular arcs*.



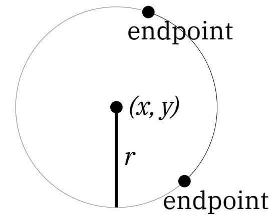
A *circular arc* is a curve between two different points on a circle, as shown on the right.

An arc is longer than a single point but shorter than a complete circle. That is, it subtends an angle θ radians where $0 < \theta < 2\pi$.

An arc of a circle of radius r that subtends angle θ radians will have length $\theta \times r$. A *major* arc curves more than half way around (subtends an angle $\theta > \pi$) and a *minor* arc curves less than half way ($\theta < \pi$). A *circular sector* is the slice-of-pie-shaped region bounded by an arc and two radii.

Archimedes writes the interface **`Arc`** in the provided code to represent circular arcs positioned in the 2D plane. You can read the code at the bottom of this page, or **open the provided code in a separate tab**.

Problem ⋊1.



Natasha points out a problem with `Arc`: there are pairs of arcs that share the same endpoints, and they cannot be distinguished given only the operations Archimedes has included. They brainstorm several possible solutions:

(⋊a) Maybe add a method that returns all the points, not just the ends?

```
/** @returns all the points on this arc */
allPoints(): Array<Point>
```

Using the taxonomy of four kinds of ADT ops, what kind of op is this?

observer

And in one sentence, why is it not feasible to add this op?

The number of points is infinite, and the number of bounded-precision Points is infeasibly large

-Question about either `isMajor()` or `length()`-

(⋊b) Maybe say if the arc is *major* or not?

```
/** @returns true iff this arc subtends an angle greater than 180 degrees, false otherwise */
isMajor(): boolean
```

In one sentence, why does this op not *quite* solve the problem?

If the endpoints are on a diameter, both arcs subtend 180 degrees and isMajor returns false

(⋊c) Maybe give the length of the arc?

```
/** @returns positive length of this arc */
length(): number;
```

In one sentence, why does this op not *quite* solve the problem?

If the endpoints are on a diameter, both arcs subtend 180 degrees and length returns the same value

(⋊d) Ah! Give a third point that shows where the arc is:

```
/** @returns a point that is on this arc and is not either of the endpoints */
thirdPoint(): Point
```

Now, write a valid **deterministic** spec that is **stronger** than the provided spec for `thirdPoint()`:

e.g. @returns the midpoint of this arc

And write a valid spec that is **weaker** than the provided spec, but which **still distinguishes** the pairs of arcs that share endpoints:

e.g. @returns a point on this arc or in the sector bounded by this arc that is not either of the endpoints

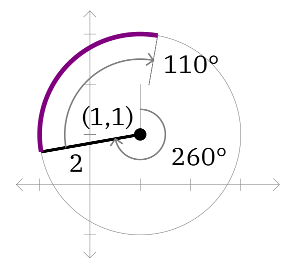
Problem ⋊2.

Archimedes starts writing an implementation:

```
class RepCRHSArc implements Arc {
    private readonly circCen: Point;
    private readonly circRad: number;
    private readonly heading: number;
    private readonly subtend: number;

    // ...
```

The example on the right shows an arc of the circle centered at (1,1) with radius 2, curving 110° clockwise from approximately (-0.970,0.653). Archimedes wants every arc to have **only one valid rep**, and the only valid rep for this arc should be:




```
circCen = (1,1)
circRad = 2
heading = 260
subtend = 110
```

Write a rep invariant and abstraction function for `RepCRHSArc` to satisfy Archimedes' requirements:

(⋊a) Rep invariant:

circRad > 0
0 <= heading < 360
0 < subtend < 360

(⋊b) Abstraction function: `AF(circCen,circRad,heading,subtend) =`

e.g. the circular arc of the circle centered at `circCen` with radius `circRad`, with one endpoint at `heading` degrees clockwise rotation from the maximum-y-coordinate point of the circle, subtending an angle `subtend` degrees clockwise rotation

Problem ⤫3.

Natasha proposes a different rep, and starts implementing it:

```
class Rep3PArc implements Arc {

    // RI: points has length 3, none of the points[i] are equal,
    //        points[1] is not on the line with points[0] and points[2]

    /**
     * Make a new circular arc of the circle that passes through the given points,
     * starting at points[0], going through points[1], ending at points[2].
     * @param points array of three distinct points where points[1] is not colinear
     *               with points[0] and points[2]
     */
    public constructor(
        private readonly points: Array<Point>,
    ) { }

    // ...
```

(⤫a) In a short phrase, what dangerous issue for `Rep3PArc` exists with its **constructor implementation**?

rep exposure

(⤫b) Explain a sequence of actions that demonstrates this problem with the constructor:

1. The client constructs a Rep3PArc with an array that meets the precondition but retains a reference to that array
2. The client mutates that array, e.g. to remove or add an element
3. The rep invariant of Rep3PArc is now violated

(⤫c) How should we fix this problem?

copy the array before assigning to the rep

Boris reviews Natasha's implementations for `circleCenter()` and `circleRadius()`. Without seeing that code (which is not provided on the quiz), for each of the suggestions below:

- say whether it sounds **reasonable**,
- and if it is, state **one benefit and one drawback** to making the change;
- and if it is not reasonable, state **why not**.

(⤫d) *Maybe instead of computing these every time, compute in the constructor and store in the rep?*

● reasonable suggestion (state one benefit and one drawback)
○ unreasonable (state why)

Benefit: improved performance if clients call those operations many times
Drawback: more complicated rep and rep invariant

(⤫e) *There's a division by zero when first and last elements of `points` are the same, wrap the computation in an `if`...`else` that checks for that.*

○ reasonable suggestion (state one benefit and one drawback)
● unreasonable (state why)

This situation is prohibited by the rep invariant, so it would be better to rely on calls to `checkRep` to detect this condition, and not make the implementation code more complex

Problem ⤫4.

Both of the `Arc` implementations rely on the provided **`Point`** class, which provides an `equalsWithin(..)` method.

(⤫a) Suppose we will use `epsilon = 0.05` to match the tolerances of the laser cutter. For each of the three properties of an equivalence relation:

- state the property (by name, or by mathematical statement),
- state whether `Point.equalsWithin(..)` with `epsilon = 0.05` satisfies that property or not,
- and if not, give a counterexample (don't write code, just use *(x, y)* points)

Reflexive: satisfied

Symmetric: satisfied

Transitive: not satisfied
(0,0) is equalsWithin to (0.04,0.04) which is equalsWithin to (0.08,0.08), but (0,0) is not equalsWithin to (0.08,0.08)

(⤫b) Compare the original spec to this alternative, identical except for one line:

```
/** ... unchanged ...
 * @param epsilon tolerance for comparison, 0 ≤ tolerance < 1
 *  ... unchanged ...
```

This new precondition is:	Given the preconditions, this new postcondition is:	This new specification is:
● stronger	○ stronger	○ stronger
○ weaker	○ weaker	● weaker
○ identical	● identical	○ identical
○ incomparable	○ incomparable	○ incomparable

(⤫c) Compare the original spec to this alternative, identical except for one line:

```
/** ... unchanged ...
 * @returns true iff the Euclidean distance between this and that is less than epsilon
 *  ... unchanged ...
```

This new precondition is:	Given the preconditions, this new postcondition is:	This new specification is:
○ stronger	○ stronger	○ stronger
○ weaker	○ weaker	○ weaker
● identical	○ identical	○ identical
○ incomparable	● incomparable	● incomparable

Problem ⤫5.

Natasha wants to build longer paths out of arcs, to make shapes for the laser cutter to cut out. She writes the **`ConnectedPath`** ADT in the provided code. *Connected* means that we can follow the arcs in the path one after another because their endpoints match up (within parameter `epsilon`).

(⤫a) Argue that `ConnectedPath` is safe from rep exposure:

epsilon is unreassignable and an immutable number
arcsArr is unreassignable and contains immutable Arc instances; it is not assigned to a reference provided as an argument to any operation (e.g. add), nor is it returned from any operation (none possible)

Start testing `ConnectedPath`.**`add(..)`**...

(⤫b) Write an excellent partition, with at least three subdomains, on input `this` only (do not use other inputs/output, and be sure to partition on the abstract value, not the rep):

count() =0, =1, >1

(⤫c) Write an excellent partition, with at least three subdomains, that relates `this` and `idx` only:

count()=0 and idx=0, count()>0 and idx<count(), count()>0 and idx=count()

(⤫d) This **`add(..)`** operation has preconditions on both `idx` and `arcs`. One of these would be especially helpful to transform into a postcondition, allowing us to test for that postcondition behavior. What precondition would you like to remove, and what new part of the postcondition would you like to have instead?

Remove the precondition "requires that resulting path be connected" and add the postcondition "@throws error if the resulting path is not connected"

Compared to the original spec, is your new spec:

- ● stronger
- ○ weaker
- ○ incomparable

Explain why:

It has a weaker precondition and identical postcondition for clients following the stronger precondition, so it is a stronger spec

---

You can **open the code below in a separate tab**.

Code for `Point`, `Arc`, and `ConnectedPath`.

```
/** Immutable point in the 2D plane. */
export class Point {
    /**
     * Make a new point (x, y).
     * @param x x-coordinate
     * @param y y-coordinate
     */
    public constructor(
        public readonly x: number,
        public readonly y: number,
    ) { }

    /**
     * @param that point to compare to this point
     * @param epsilon nonnegative tolerance for comparison
     * @returns true iff for both x and y, the difference in coordinate values is less than epsilon
     */
    public equalsWithin(that: Point, epsilon: number): boolean {
        return (Math.abs(this.x-that.x) < epsilon) &&
               (Math.abs(this.y-that.y) < epsilon);
    }
}

/** Immutable circular arc. */
export interface Arc {

    /** @returns center point of the circle of which this is an arc */
    circleCenter(): Point;

    /** @returns radius of the circle of which this is an arc */
    circleRadius(): number;

    /** @returns array of the two distinct end points of this arc, in arbitrary order */
    endpoints(): Array<Point>;
}

/** Mutable path in the 2D plane, made of a connected sequence of zero or more circular arcs. */
export class ConnectedPath {

    private readonly arcsArr: Array<Arc> = [];

    /**
     * Make an empty path that has no arcs.
     * @param epsilon nonnegative point equality tolerance for determining if this path is connected
     */
    public constructor(
        public readonly epsilon: number,
    ) { }

    /** @returns number of arcs in this connected path */
    public count(): number { return this.arcsArr.length; }

    /**
```

```
     * @param idx nonnegative index less than count()
     * @returns the idx^th arc in this path
     */
    public arc(idx: number): Arc { return this.arcsArr[idx] ?? assert.fail(); }

    /**
     * Insert arcs at the given place in this path.
     * @param idx nonnegative index at which to insert, less than or equal to count()
     * @param arcs arcs to insert, requires that the resulting path be connected
     */
    public add(idx: number, arcs: Array<Arc>): void {
        // insert all the items from arcs into arcsArr at index idx, see notes below
        this.arcsArr.splice(idx, 0, ...arcs);
        // TODO: call checkRep!
    }
}
```

`Array splice(start, deleteCount, item1, item2, …)` — the `splice(..)` method changes the contents of an array by removing or replacing existing elements and/or adding new elements in place.

`start` the index at which to start changing the array
`deleteCount` integer number of elements to remove from the array, starting at index `start`
`item1, item2, …` zero or more elements to add to the array, starting at index `start`

spread syntax (**`...`**) — allows an iterable (such as an array) to be expanded in places where zero or more arguments are expected (such as a function call). Given:

```
function sum(x, y, z) { return x + y + z; }
const numbers = [ 1, 2, 3 ];
```

Calling `sum(...numbers)` returns 6.