MIT
6.031 Software Construction
Prof. Rob Miller and Max Goldman

6.031 Spring 2022 Quiz 1

You have **50** minutes to complete this quiz. There are **5** problems, each worth approximately an equal share of points.

The quiz is **closed-book** and closed-notes, but you are allowed one two-sided page of notes on paper, and you may use blank scratch 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 *check in* by having the course staff scan the QR code at the top of the page.

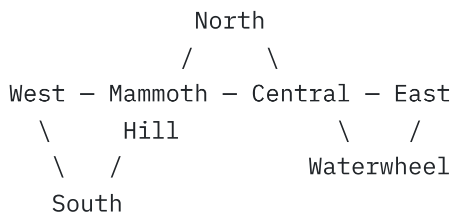
To leave the quiz early: click the *done* button at the very bottom of the page and show your screen with the check-out code to a staff member.

This page automatically saves your answers as you work. 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.

Good luck!

The questions on this quiz use the code at the bottom of this page. You can **open this introduction and the provided code in a separate tab**.

Battery magnate Elron Tusk has decided to donate a scooter-sharing system to the town of Riverdale. Unfortunately, Tusk only wants to donate tiny batteries, so the scooters will use **docking stations** to charge, and they will only be able to scoot to **nearby** charging stations before they run out of juice. Here's a map showing which stations are reachable from one another:



Despite its name, Riverdale's Mammoth Hill is not very tall, and the whole town is quite flat. Therefore, scooting range is just a function of distance, and all the connections are bidirectional. Mathematically speaking, the system is an undirected graph of docking stations with no self-loops: if and only if you can scoot from station A to station B, then you can scoot from B to A. Stations are never considered to be connected to themselves.

Each station has a limited number of **charging docks** where scooters can be plugged in. Each dock at a station is either empty, has a charging scooter docked there, or has a fully-charged scooter.

Problem ⋊1.

The provided class **`Station`** represents a mutable docking station with some number of docks where scooters can be charged.

(⋊a) For each of the following changes, does it *break* the code, *fix* the code, or have *no effect* on the correctness? Ignore changes to ETU, consider only SFB.

In `status(..)`, use:

```
return this.docks.map(s => s);
```

Pick one: ○ breaks ○ fixes ○ no effect ... and say why:

In `dock(..)`, use:

```
this.docks.map(s => s)[dock] = Status.Charging;
```

Pick one: ○ breaks ○ fixes ○ no effect ... and say why:

(⋊b) Complete the rep invariant for `Station`:

RI(`name`, `size`, `docks`) = `size` is a nonnegative integer and...

... and complete the abstraction function, such that they work with provided code after any necessary fixes:

AF(`name`, `size`, `docks`) = the scooter docking station named `name`...

Problem ⋊2.

Evalute the following potential changes to specs in `Station`.

(⋊a) Compare the original spec for `dock(..)` to this alternative:

```
// Modify this station to have a charging scooter in the given dock.
// Requires 0 ≤ dock < size. Throws an error and makes no modification
//   if the given dock is not currently Empty.
```

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

(⋊b) Compare the original spec for `dock(..)` to this alternative:

```
// Modify this station to have some scooter in the given dock.
// Requires 0 ≤ dock < size, and the dock must currently be Empty.
```

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

Problem ⋊3.

The provided class **`Network`** represents a mutable network using the rep `Map<Station, Set<Station>>`.

(⋊a) Which of the following changes should we make to prevent rep exposure?

- Evaluate just the suggested change: assume we fix other code to work with the change, but *don't* assume what the fix is.
- Keep in mind that TypeScript considers `Set<E>` to be a subtype of `ReadonlySet<E>` because TypeScript uses structural subtyping, even though it is *not* a subtype judging by its specifications.

Change the type of the `nearbyStations` argument of `add(..)` to `ReadonlySet<Station>` to prevent rep exposure:

Pick one: ○ yes ○ no ... and say why or why not:

Change the type of the return value for `getNearby(..)` to `ReadonlySet<Station>` to prevent rep exposure:

Pick one: ○ yes ○ no ... and say why or why not:

(⋊b) Is it safe to use `Station` as the type of `Map` keys and `Set` elements, given we cannot override the equality comparisons used by those data structures?

Pick one: ○ yes ○ no ... and say why or why not:

Problem ⋊4.

There are still more bugs to fix in `Network`, because `getNearby(..)` requires a rep invariant that `add(..)` does not preserve.

(⋊a) One partial fix for the problem is to add another precondition to the `add(..)` method:

```
// Requires every station in nearbyStations is in this network.
```

If we do not add this precondition, what rep invariant needed by `getNearby(..)` would the client easily be able to break?

Not sure? Start by writing down any *rep invariant needed in* `Network`*.*

(⋊b) After adding this precondition and fixing any rep exposure, there is *still* a way for a client to break this necessary invariant using `add(..)`.

Say (without writing code) how a client can call `add(..)` and break the rep invariant *without* the client doing its own mutation of the rep, neither directly nor through aliases:

(⋊c) Explain (without writing the code) a change to the implementation of `add(..)` that fixes this problem:

Not sure? Start by writing down any *bug fix needed in add.*

Problem ⋊5.

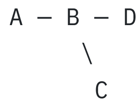
Instead of the rep used in `Network`, let's build an ADT without using maps or sets. Our new rep will be:

```
private readonly netarr: Array<Array<Station>>;
```

With the abstraction function:

AF(`netarr`) = the network with stations { `s` | `s` appears in a subarray of `netarr` }
where two stations `s1` and `s2` are connected if and only if `s1≠s2` and `netarr` contains at least one of:
a subarray whose first element is `s1` that also contains `s2`, and/or
a subarray whose first element is `s2` that also contains `s1`

For example, suppose ABC Town has four stations:



Without knowing the rep invariant, we might imagine many possible reps for ABC Town, including:

```
netarr = [ [ A, B ], [ B, C, D ] ]
netarr = [ [ B, C, D, A ] ]
netarr = [ [], [A], [B], [C], [D], [ B, A, A, C, C, D, D ] ]
```

Assuming `A`, `B`, `C`, and `D` are appropriate `Station` objects, all of these reps for ABC Town work with the AF above.

(⋊a) A weak rep invariant allows more possible rep values. Could we use the following rep invariant, or is it too weak?

RI(`netarr`) = true

Pick one: ◯ yes (leave the box blank)
◯ no ... and say why not:

(⋊b) A strong rep invariant allows fewer possible rep values. Could we use the following rep invariant, or is it too strong?

RI(`netarr`) = every station that appears in `netarr` appears exactly once

Pick one: ◯ yes (leave the box blank)
◯ no ... and say why not:

(⋊c) For each of the following subdomains, say whether we should include it as part of a partition in our documented testing strategy for `getNearby(..)`:

Want to cover: `station` has no nearby stations

Pick one: ◯ yes
◯ no
... and say why or why not:

Want to cover: `station` is nearby itself

Pick one: ◯ yes
◯ no
... and say why or why not:

Want to cover: `netarr` has no subarrays

Pick one: ◯ yes
◯ no
... and say why or why not:

```
/** Station dock status: no scooter, a charging scooter, or a fully-charged scooter. */
enum Status {
    Empty, Charging, Charged
}
```

```
/** Mutable scooter-sharing network station with docks numbered 0 through size-1. */
class Station {
    private readonly docks: Array<Status>;

    // Make a new station with the given name and the given number of scooter docks.
    // Requires nonnegative integer size.
    public constructor(public readonly name: string,
                       public readonly size: number) {
        this.docks = new Array(size).fill(Status.Empty);
    }

    // Get the status of all docks at this station.
    public status(): Array<Status> {
        return this.docks;
    }

    // Modify this station to have a charging scooter in the given dock.
    // Requires 0 ≤ dock < size, and the dock must currently be Empty.
    public dock(dock: number): void {
        this.docks[dock] = Status.Charging;
    }

    // ... other operations, e.g. taking out a charged scooter, etc. ...
}

/** Mutable scooter-sharing network, an undirected graph of mutable stations where
 *  each station is only nearby (within scooting range of) certain other stations. */
class Network {
    private readonly netmap: Map<Station, Set<Station>> = new Map();

    // Make an empty network.
    public constructor() { }

    // Add the given station to this network if it is not already present,
    //   and set its nearby stations to the given set of stations.
    // Requires station not a member of nearbyStations.
    public add(station: Station, nearbyStations: Set<Station>): void {
        this.netmap.set(station, nearbyStations);
        for (const other of nearbyStations) {
            const nearbyOther = this.netmap.get(other);
            if (nearbyOther) { nearbyOther.add(station); }
        }
    }

    // Get the stations in this network nearby the given station.
    public getNearby(station: Station): Set<Station> {
        const nearby = this.netmap.get(station);
        if (nearby) { return nearby; } else { return new Set(); }
    }

    // ... other operations, e.g. removing a station, etc. ...
}
```

`Array(length)` constructor — create an `Array` object.

`length` nonnegative integer length of the new array. Note: this is an array of `length` empty slots, not slots with `undefined` values. Use *e.g.* `fill(..)` to initialize the values in the array.

`Array fill(value)` — change all the elements in the array to a given value, and return the modified array.

`value` the value to fill the array with. Note: all elements in the array will be this exact value.