Question 1: Specification Strength

Consider four possible specifications for a method `static void storeLargest(List<Integer> list)`. Every call passes a non-null `list` with at least one element. In the specifications, `list0` is `list` as it was when the method was called, and a list is in *non-decreasing order* when each of its elements is at most the next.

```
Spec A
  @modifies list
  @effects  stores the largest element of list0 at index 0 of list;
            the other elements of list are unspecified

Spec B
  @requires list is in non-decreasing order
  @modifies list
  @effects  stores the largest element of list0 at index 0 of list;
            the other elements of list are unspecified

Spec C
  @modifies list
  @effects  stores the largest element of list0 at index 0 of list,
            leaving the other elements unchanged

Spec D
  @requires no two elements of list are equal
  @modifies list
  @effects  stores the largest element of list0 at index 0 of list,
            leaving the other elements unchanged
```

For each pair, write in its box:

- **S** if the first specification is stronger than the second
- **W** if the first is weaker than the second
- **I** if the two are incomparable

Write only your answer in each box, clearly; work anywhere outside them.	**Answer** **(S / W / I)**
a. A and B	**S**
b. A and C	**W**
c. B and C	**W**

Question 1: Specification Strength, continued

Repeated for your convenience:

> For each pair, write in its box:
>
> - **S** if the first specification is stronger than the second
> - **W** if the first is weaker than the second
> - **I** if the two are incomparable

Write only your answer in each box, clearly; work anywhere outside them.	**Answer** (S / W / I)
d. A and D	**I**
e. C and D	**S**
f. B and D	**I**

a.

Answer
S

Stronger: A asks less than B and promises the same.

A's precondition is weaker.

- A asks nothing; B asks that `list` be in non-decreasing order.

The postconditions are the same.

- Both allow exactly the lists whose first element is the largest element of `list0`.

An implementation that shows it:

- One that moves the last element of `list` to the front, the others keeping their order, meets B and not A.
  - Where `list` is in non-decreasing order, its last element is its largest, so `list.get(0)` ends as the largest of `list0`, as B asks.
  - On `[3, 1, 2]` it leaves `[2, 3, 1]`, and A asks that `list.get(0)` be 3.

The rule: Lecture 2's Stronger and weaker (https://courses.cs.washington.edu/courses/cse331/26au/lectures/lecture02-specifications/notes.html#stronger-and-weaker).

b.

Answer
W

Weaker: A asks the same as C and promises less.

The preconditions are the same.

- Neither asks anything.

C's postcondition is stronger.

- C allows one result: `list0` with its first element replaced by the largest element of `list0`.
- Its `list.get(0)` is the largest of `list0`, so A allows it too.
- A allows more: A leaves the rest of `list` unspecified, so on `[1, 3, 2]` it allows `[3, 3, 3]`, and C does not.

An implementation that shows it:

- One that sets every element of `list` to its largest meets A and not C.
  - It makes `list.get(0)` the largest of `list0`, as A asks.
  - On `[1, 3, 2]` it leaves `[3, 3, 3]`, and C asks for `[3, 3, 2]`.

The rule: Lecture 2's Stronger and weaker (https://courses.cs.washington.edu/courses/cse331/26au/lectures/lecture02-specifications/notes.html#stronger-and-weaker).

c.

Answer
W

Weaker: B asks more than C and promises less.

C's precondition is weaker.

- C asks nothing; B asks that `list` be in non-decreasing order.

C's postcondition is stronger.

- B allows only calls where `list` is in non-decreasing order, so compare them on those calls.
- C allows one result: `list0` with its first element replaced by the largest element of `list0`.
- Its `list.get(0)` is the largest of `list0`, so B allows it too.
- B allows more: on `[1, 2, 3]`, B allows `[3, 1, 2]`, and C does not.

An implementation that shows it:

- One that moves the last element of `list` to the front, the others keeping their order, meets B and not C.
  - Where `list` is in non-decreasing order, its last element is its largest, so it does what B asks.
  - On `[1, 2, 3]` it leaves `[3, 1, 2]`, and C asks for `[3, 2, 3]`.

The rule: Lecture 2's Stronger and weaker (https://courses.cs.washington.edu/courses/cse331/26au/lectures/lecture02-specifications/notes.html#stronger-and-weaker).

d.

Answer
I

Incomparable: A asks less than D, but promises less on the calls D allows.

A's precondition is weaker.

- A asks nothing; D asks that no two elements of `list` be equal.
- So D cannot be stronger than A.

D's postcondition is stronger.

- D allows only calls where no two elements are equal, so compare them on those calls.
- D allows one result: `list0` with its first element replaced by the largest element of `list0`.
- Its `list.get(0)` is the largest of `list0`, so A allows it too.
- A allows more: on `[1, 3, 2]`, A allows `[3, 3, 3]`, and D does not.
- So A cannot be stronger than D either.

Two implementations that show it:

- One that sets every element of `list` to its largest meets A and not D.
  - It makes `list.get(0)` the largest of `list0`, as A asks.
  - On `[1, 3, 2]` it leaves `[3, 3, 3]`, and D asks for `[3, 3, 2]`.
- One that sets `list.get(0)` to the largest element when no two elements are equal, and otherwise leaves `list` as it is, meets D and not A.
  - Where no two elements are equal, it does what D asks.
  - On `[1, 2, 2]` it leaves `[1, 2, 2]`, and A asks that `list.get(0)` be 2.

The rule: Lecture 2's Stronger and weaker (https://courses.cs.washington.edu/courses/cse331/26au/lectures/lecture02-specifications/notes.html#stronger-and-weaker).

e.

Answer
S

Stronger: C asks less than D and promises the same.

C's precondition is weaker.

- C asks nothing; D asks that no two elements of `list` be equal.

The postconditions are the same.

- Both allow one result: `list0` with its first element replaced by the largest element of `list0`.

An implementation that shows it:

- One that sets `list.get(0)` to the largest element when no two elements are equal, and otherwise leaves `list` as it is, meets D and not C.
  - Where no two elements are equal, it does what D asks.
  - On `[1, 2, 2]` it leaves `[1, 2, 2]`, and C asks for `[2, 2, 2]`.

The rule: Lecture 2's Stronger and weaker (https://courses.cs.washington.edu/courses/cse331/26au/lectures/lecture02-specifications/notes.html#stronger-and-weaker).

f.

Answer
I

Incomparable: each allows a call the other forbids.

Neither precondition is weaker.

- B allows `[1, 2, 2]`, in non-decreasing order with two equal elements; D forbids it. So D cannot be stronger than B.
- D allows `[3, 1, 2]`, out of order with no two elements equal; B forbids it. So B cannot be stronger than D.

Two implementations that show it:

- One that moves the last element of `list` to the front, the others keeping their order, meets B and not D.
  - Where `list` is in non-decreasing order, its last element is its largest, so it does what B asks.
  - On `[3, 1, 2]` it leaves `[2, 3, 1]`, and D asks for `[3, 1, 2]`.
- One that sets `list.get(0)` to the largest element when no two elements are equal, and otherwise leaves `list` as it is, meets D and not B.
  - Where no two elements are equal, it does what D asks.
  - On `[1, 2, 2]` it leaves `[1, 2, 2]`, and B asks that `list.get(0)` be 2.

The rule: Lecture 2's Stronger and weaker (https://courses.cs.washington.edu/courses/cse331/26au/lectures/lecture02-specifications/notes.html#stronger-and-weaker).

Question 2: Test Coverage: 3n + 1

Here is a method, with its specification in Javadoc comments. The code correctly implements the specification. You may assume no arithmetic here ever overflows.

```
/**
 * Applies exactly steps updates to n, even if n reaches 1.
 * Each update divides an even n by 2, or replaces an odd n
 * with 3 * n + 1.
 *
 * @requires 0 < n and 0 <= steps
 * @return the value after exactly steps updates
 */
static long collatz(long n, int steps) {
    for (int i = 0; i < steps; i++) {
        if (n % 2 == 0) {
            n = n / 2;
        } else {
            n = 3 * n + 1;
        }
    }
    return n;
}
```

Each row lists a set of test calls. For each row, answer each coverage question about its tests:

- **Y** for yes
- **N** for no

Write only your answer in each box, clearly; work anywhere outside them.	**Statement coverage?** (Y / N)	**Branch coverage?** (Y / N)	**Loop coverage?** (Y / N)

a. `collatz(1, 0)`
`collatz(2, 1)`
`collatz(4, 2)`
	N	**N**	**Y**

b. `collatz(1, 0)`
`collatz(2, 1)`
`collatz(1, 5)`
	Y	**Y**	**Y**

c. `collatz(1, 5)`	**Y**	**Y**	**N**

d. `collatz(3, 2)`
`collatz(3, 0)`
`collatz(3, 1)`
	Y	**Y**	**Y**

a.

Statement coverage?	Branch coverage?	Loop coverage?
N	**N**	**Y**

These tests have loop coverage but never take the odd branch.

Statement coverage: no.

- `(1, 0)` performs no updates.
- The other calls follow `2 -> 1` and `4 -> 2 -> 1`.
- The statement `n = 3 * n + 1` never runs.

Branch coverage: no.

- Whenever the `if` runs, `n` is even.
- No test takes its odd branch.
- Reaching 1 does not take that branch: each call stops before another iteration can test its parity.

Loop coverage: yes.

- The body runs zero times on `(1, 0)`, once on `(2, 1)`, and twice on `(4, 2)`.

The definitions: Lecture 5's Statement coverage (https://courses.cs.washington.edu/courses/cse331/26au/lectures/lecture05-coverage/notes.html#statement-coverage), Branch coverage (https://courses.cs.washington.edu/courses/cse331/26au/lectures/lecture05-coverage/notes.html#branch-coverage), and Loop coverage (https://courses.cs.washington.edu/courses/cse331/26au/lectures/lecture05-coverage/notes.html#loop-coverage).

b.

Statement coverage?	Branch coverage?	Loop coverage?
Y	Y	Y

These tests have all three kinds of coverage.

Statement coverage: yes.

- `(1, 5)` follows `1 -> 4 -> 2 -> 1 -> 4 -> 2`.
- That call runs both updates and every other statement.

Branch coverage: yes.

- `(1, 5)` takes both branches of the `if`.
- The loop condition is true during the nonempty runs and false when each call finishes.

Loop coverage: yes.

- The body runs zero times on `(1, 0)`, once on `(2, 1)`, and five times on `(1, 5)`.

The definitions: Lecture 5's Statement coverage (https://courses.cs.washington.edu/courses/cse331/26au/lectures/lecture05-coverage/notes.html#statement-coverage), Branch coverage (https://courses.cs.washington.edu/courses/cse331/26au/lectures/lecture05-coverage/notes.html#branch-coverage), and Loop coverage (https://courses.cs.washington.edu/courses/cse331/26au/lectures/lecture05-coverage/notes.html#loop-coverage).

c.

Statement coverage?	Branch coverage?	Loop coverage?
Y	**Y**	**N**

The test runs both updates, but only covers a loop run of more than one iteration.

Statement coverage: yes.

- The code follows `1 -> 4 -> 2 -> 1 -> 4 -> 2`.
- Both assignments to `n` run, as does every other statement.

Branch coverage: yes.

- The odd and even branches both run.
- The loop condition is true for five iterations, then false.

Loop coverage: no.

- The body runs five times.
- No test runs it zero times or once.

The definitions: Lecture 5's Statement coverage (https://courses.cs.washington.edu/courses/cse331/26au/lectures/lecture05-coverage/notes.html#statement-coverage), Branch coverage (https://courses.cs.washington.edu/courses/cse331/26au/lectures/lecture05-coverage/notes.html#branch-coverage), and Loop coverage (https://courses.cs.washington.edu/courses/cse331/26au/lectures/lecture05-coverage/notes.html#loop-coverage).

d.

Statement coverage?	Branch coverage?	Loop coverage?
Y	Y	Y

These tests run both updates and cover all three loop counts.

Statement coverage: yes.

- `(3, 2)` follows `3 -> 10 -> 5`.
- The first iteration runs the odd update; the second runs the even update.
- Every statement runs.

Branch coverage: yes.

- The `if` takes its odd branch on 3 and its even branch on 10.
- On `(3, 2)`, the loop condition is true twice, then false.

Loop coverage: yes.

- The body runs twice on `(3, 2)`, zero times on `(3, 0)`, and once on `(3, 1)`.

The definitions: Lecture 5's Statement coverage (https://courses.cs.washington.edu/courses/cse331/26au/lectures/lecture05-coverage/notes.html#statement-coverage), Branch coverage (https://courses.cs.washington.edu/courses/cse331/26au/lectures/lecture05-coverage/notes.html#branch-coverage), and Loop coverage (https://courses.cs.washington.edu/courses/cse331/26au/lectures/lecture05-coverage/notes.html#loop-coverage).

Question 3: Hoare Triples

Each row is a Hoare triple. All variables have type `int`. You may assume no arithmetic here ever overflows.

For each triple, write in its box:

- **T** if the triple is valid
- **F** if it is not

Write only your answer in each box, clearly; work anywhere outside them.	**Valid?** **(T / F)**

a. `{{ x + y = 10 }}`
`x = 0;`
`{{ y = 10 }}`
	F

b. `{{ 0 <= x }}`
`if (x < 0) {`
`  y = -1;`
`} else {`
`  y = x + 1;`
`}`
`{{ 0 < y }}`
	T

c. `{{ x + y = n }}`
`x = x - y;`
`y = 2 * y;`
`{{ x + y = n }}`
	T

d. `{{ true }}`
`if (x == 0) {`
`  y = 0;`
`} else {`
`  y = 1;`
`}`
`x = 0;`
`{{ y = 0 }}`
	F

Question 3: Hoare Triples, continued

Repeated for your convenience:

> Each row is a Hoare triple. All variables have type `int`. You may assume no arithmetic here ever overflows.
>
> For each triple, write in its box:
>
> - **T** if the triple is valid
> - **F** if it is not

Write only your answer in each box, clearly; work anywhere outside them.	**Valid?** **(T / F)**

e. `{{ 0 <= x }}`
`if (0 < x) {`
`  y = x;`
`}`
`{{ 0 <= y }}`
	F

f. `{{ x = 0 or y = 0 }}`
`if (x != 0) {`
`  if (y != 0) {`
`  x = 1;`
`  y = 1;`
`  }`
`}`
`{{ x = 0 or y = 0 }}`
	T

g. `{{ 0 <= x }}`
`while (0 < x) {`
`  x = x - 2;`
`}`
`{{ x = 0 }}`
	F

h. `{{ x != y }}`
`while (x != y) {`
`  int t = x;`
`  x = y;`
`  y = t;`
`}`
`{{ false }}`
	T

a.

Valid?
F

Not valid: assigning to `x` does not change `y`.

A counterexample:

- Start from `x = 3` and `y = 7`, so `x + y = 10`.
- The assignment makes `x` equal to 0.
- `y` stays 7, so `y = 10` is false.

The rule: Lecture 7's The rules for an assignment (https://courses.cs.washington.edu/courses/cse331/26au/lectures/lecture07-reasoning/notes.html#the-rules-for-an-assignment).

b.

Valid?
T

Valid: every input the precondition allows takes the false branch.

Forward, from the precondition:

```
{{ 0 <= x }}
if (x < 0) {
    {{ 0 <= x and x < 0 }}
    {{ false }}
    y = -1;
    {{ false }}
} else {
    {{ 0 <= x }}
    y = x + 1;
    {{ 0 <= x and y = x + 1 }}
}
{{ false or (0 <= x and y = x + 1) }}
{{ 0 < y }}
```

1. The conditional rule adds `x < 0` in the true branch.
2. No state has both `0 <= x` and `x < 0`.
3. The assignment preserves `false`, since no run reaches it.
4. The false branch starts with `0 <= x`, as the condition is false there.
5. The assignment preserves `0 <= x` and adds `y = x + 1`.
6. The conditional rule joins the branch claims with `or`.
7. Dropping `false or` leaves `0 <= x and y = x + 1`, which gives `1 <= y`, so `0 < y`.

The rules: Lecture 7's The rule for a conditional (https://courses.cs.washington.edu/courses/cse331/26au/lectures/lecture07-reasoning/notes.html#the-rule-for-a-conditional) and The rules for an assignment (https://courses.cs.washington.edu/courses/cse331/26au/lectures/lecture07-reasoning/notes.html#the-rules-for-an-assignment).

c.

Valid?
T

Valid: the two assignments preserve the sum.

Backward, from the postcondition:

```
{{ x + y = n }}
{{ (x - y) + 2 * y = n }}
x = x - y;
{{ x + 2 * y = n }}
y = 2 * y;
{{ x + y = n }}
```

1. Before `y = 2 * y`, put `2 * y` for `y` in the postcondition.
2. Before `x = x - y`, put `x - y` for `x` in the claim below it.
3. `(x - y) + 2 * y` simplifies to `x + y`, so the precondition implies the weakest precondition and the triple is valid.

The rules: Lecture 7's The rules for an assignment (https://courses.cs.washington.edu/courses/cse331/26au/lectures/lecture07-reasoning/notes.html#the-rules-for-an-assignment) and The rule for a sequence (https://courses.cs.washington.edu/courses/cse331/26au/lectures/lecture07-reasoning/notes.html#the-rule-for-a-sequence).

d.

Valid?
F

Not valid: when `x` starts nonzero, `y` ends as 1.

A counterexample:

- Start from `x = 1` and `y = 0`.
- The condition is false, so the false branch makes `y = 1`.
- The final assignment makes `x = 0`.
- That assignment leaves `y` unchanged.
- The final `y` is 1, so `y = 0` is false.

The rules: Lecture 7's The rule for a conditional (https://courses.cs.washington.edu/courses/cse331/26au/lectures/lecture07-reasoning/notes.html#the-rule-for-a-conditional) and The rule for a sequence (https://courses.cs.washington.edu/courses/cse331/26au/lectures/lecture07-reasoning/notes.html#the-rule-for-a-sequence).

e.

Valid?
F

Not valid: the empty false branch need not make `0 <= y` true.

A counterexample:

- Start from `x = 0` and `y = -1`, which the precondition allows.
- `0 < x` is false, so the assignment is skipped.
- `y` stays -1, so `0 <= y` is false.

The rule: Lecture 7's The rule for a conditional (https://courses.cs.washington.edu/courses/cse331/26au/lectures/lecture07-reasoning/notes.html#the-rule-for-a-conditional).

f.

Valid?
T

Valid: the inner true branch cannot execute.

Forward, from the precondition:

```
{{ x = 0 or y = 0 }}
if (x != 0) {
    {{ x != 0 and y = 0 }}
    if (y != 0) {
        {{ x != 0 and y = 0 and y != 0 }}
        {{ false }}
        x = 1;
        {{ false }}
        y = 1;
        {{ false }}
    }
    {{ x != 0 and y = 0 }}
}
{{ x = 0 or y = 0 }}
```

1. The outer true branch adds `x != 0`, so the precondition requires `y = 0`.
2. The inner true branch adds `y != 0`.
3. `y = 0` and `y != 0` cannot both be true.
4. The first assignment preserves `false`.
5. The second assignment preserves `false`.
6. The inner conditional joins `false` with the empty false branch's `x != 0 and y = 0`, leaving that claim.
7. The outer conditional joins that claim with its empty false branch's `x = 0`, giving the postcondition.

The rules: Lecture 7's The rule for a conditional (https://courses.cs.washington.edu/courses/cse331/26au/lectures/lecture07-reasoning/notes.html#the-rule-for-a-conditional) and The rules for an assignment (https://courses.cs.washington.edu/courses/cse331/26au/lectures/lecture07-reasoning/notes.html#the-rules-for-an-assignment).

g.

Valid?
F

Not valid: the loop can end with a negative `x`.

A counterexample:

- Start from `x = 1`, which the precondition allows.
- The condition is true, so the body makes `x = -1`.
- The condition is now false, so the loop ends.
- The final `x` is -1, so `x = 0` is false.

The rules: Lecture 7's A triple (https://courses.cs.washington.edu/courses/cse331/26au/lectures/lecture07-reasoning/notes.html#a-triple) and Lecture 8's The rule for a while loop (https://courses.cs.washington.edu/courses/cse331/26au/lectures/lecture08-loops/notes.html#the-rule-for-a-while-loop).

h.

Valid?
T

Valid: no run from the precondition terminates.

Forward, with invariant `x != y`:

```
{{ x != y }}
{{ Inv: x != y }}
while (x != y) {
    {{ x != y }}
    int t = x;
    {{ x != y and t = x }}
    x = y;
    {{ t != y and x = y }}
    y = t;
    {{ x != y }}
}
{{ x != y and x = y }}
{{ false }}
```

1. The precondition is the invariant, so check (1) passes.
2. The while rule adds `x != y`, which is already the invariant.
3. The assignment saves `x` in `t` while preserving `x != y`.
4. The assignment sets `x = y` while `t != y` remains true.
5. Since `t` differs from `x`, assigning `t` to `y` restores `x != y`, so check (2) passes.
6. At exit, the while rule gives `x != y` and the false condition, `x = y`.
7. No state makes both claims true, so they imply `false` and check (3) passes.

The rules: Lecture 7's The rules for an assignment (https://courses.cs.washington.edu/courses/cse331/26au/lectures/lecture07-reasoning/notes.html#the-rules-for-an-assignment) and Lecture 8's The rule for a while loop (https://courses.cs.washington.edu/courses/cse331/26au/lectures/lecture08-loops/notes.html#the-rule-for-a-while-loop).

Question 4: Loop Checks: Count Positive Elements

Arrays are non-null. `n` is the length of `a` and does not change. `a[l..r]` is the slice starting at `l` and ending just before `r`. Every numeric variable and array element is an `int`. You may assume no arithmetic here ever overflows. The number of positive values in an empty slice is 0.

```
{{ i = 0 and count = 0 }}
{{ Inv: I }}
while (i < n) {
    if (0 < a[i]) {
        count = count + 1;
    }
    i = i + 1;
}
{{ count = number of positive values in a }}
```

Each row is a candidate loop invariant `I`. With `P` the precondition on the line above `Inv`, `c` the loop's condition, and `Q` the postcondition just after the loop, the rule for a while loop asks three checks of `I`:

- **Check (1):** `P` implies `I`.
- **Check (2):** `{{ I and c }}` body `{{ I }}` is valid.
- **Check (3):** `I and not c` implies `Q`.

Every check may use the stated array shapes and unmodified inputs; only check (1) may use `P`. Checks (2) and (3) start from any state where their claims are true, even one the loop never reaches.

For each candidate `I`, write in each check's box:

- **Y** if the check passes
- **N** if it fails

Write only your answer in each box, clearly; work anywhere outside them.	**Check (1)** (Y / N)	**Check (2)** (Y / N)	**Check (3)** (Y / N)
a. `{{ 0 <= i <= n and 0 <= count <= i }}`	**Y**	**Y**	**N**
b. `{{ 0 <= i <= n and count = number of positive values in a[0..i] }}`	**Y**	**Y**	**Y**
c. `{{ 0 <= i <= n and count = 1 + number of positive values in a[0..i] }}`	**N**	**Y**	**N**
d. `{{ i = 0 and count = 0 }}`	**Y**	**N**	**Y**

a.

Check (1)	Check (2)	Check (3)
Y	Y	N

Check (1) passes.

- Initially `i = count = 0`, so both bounds hold.

Check (2) passes.

- `count` increases by either 0 or 1 while `i` increases by 1.
- Thus `0 <= count <= i` is still true.
- The condition `i < n` keeps the new index at most `n`.

Check (3) fails.

- A bound on the count does not connect it to the positive elements of the array.
- **A counterexample:**
  - Before: `n = 1; a = [0]; i = 1; count = 1`.
  - The candidate is true.
  - The condition is false.
  - The postcondition is false.

The rule: Lecture 8's The rule for a while loop (https://courses.cs.washington.edu/courses/cse331/26au/lectures/lecture08-loops/notes.html#the-rule-for-a-while-loop).

b.

Check (1)	Check (2)	Check (3)
Y	Y	Y

Check (1) passes.

- Initially the prefix is empty and its positive count is 0, which equals `count`.

Check (2) passes.

- A positive next element increases both counts by 1.
- Any other element increases neither count.
- Incrementing `i` includes exactly that element in the prefix and preserves the bounds.

Check (3) passes.

- The bounds and the false condition give `i = n`.
- The prefix is the entire array, so its positive count is the required answer.

The rule: Lecture 8's The rule for a while loop (https://courses.cs.washington.edu/courses/cse331/26au/lectures/lecture08-loops/notes.html#the-rule-for-a-while-loop).

c.

Check (1)	Check (2)	Check (3)
N	Y	N

Check (1) fails.

- The empty prefix has no positive values, so this candidate requires an initial count of 1.
- **A counterexample:**
  - Before: `n = 0; a = []; i = 0; count = 0` .
  - The precondition is true.
  - The candidate is false.

Check (2) passes.

- The body adds the same amount to the stored count and the prefix count.
- Their difference remains 1.
- The condition `i < n` keeps the incremented index in range.

Check (3) fails.

- At exit the stored count is one greater than the required answer.
- **A counterexample:**
  - Before: `n = 0; a = []; i = 0; count = 1` .
  - The candidate is true.
  - The condition is false.
  - The postcondition is false.

The rule: Lecture 8's The rule for a while loop (https://courses.cs.washington.edu/courses/cse331/26au/lectures/lecture08-loops/notes.html#the-rule-for-a-while-loop).

d.

Check (1)	Check (2)	Check (3)
Y	N	Y

Check (1) passes.

- This candidate is the given precondition.

Check (2) fails.

- Every execution of the body increases `i` to 1, breaking `i = 0`.
- **A counterexample:**
  - Before: `n = 1; a = [0]; i = 0; count = 0`.
  - The candidate is true before the body.
  - The condition is true before the body.
  - After: `n = 1; a = [0]; i = 1; count = 0`.
  - The candidate is now false.

Check (3) passes.

- `i = 0` and `not (i < n)` force `n = 0`, since array lengths are nonnegative.
- The empty array contains zero positive values, which equals `count`.

The rule: Lecture 8's The rule for a while loop (https://courses.cs.washington.edu/courses/cse331/26au/lectures/lecture08-loops/notes.html#the-rule-for-a-while-loop).

Question 5: Representation Invariants and Abstraction Functions

Here is a class `Line`, with its specification in Javadoc comments. You may assume no arithmetic here ever overflows.

```
/**
 * An ordered line of numbered tickets at a deli counter, as well as
 * peak, which tracks the most tickets ever waiting at once.
 *
 * Example sequence, showing abstract state (tickets front first):
 *   new Line()                 -> tickets [],     peak 0
 *   take() -> 1; take() -> 2   -> tickets [1, 2], peak 2
 *   serve()                    -> tickets [2],    peak 2
 *   serve(); serve()           -> tickets [],     peak 2
 *   take() -> 3                -> tickets [3],    peak 2
 */
class Line {
    private int issued;   // RI: ...
    private int served;   // AF(this) = ...
    private int peak;

    /** @effects makes a line with no tickets waiting and peak 0 */
    Line() { issued = 0; served = 0; peak = 0; }

    /**
     * @modifies this
     * @effects adds the next ticket at the back:
     *     ticket 1 first, then ticket 2, and so on
     * @return the number of the ticket added
     */
    int take() {
        issued = issued + 1;
        peak = Math.max(peak, issued - served);
        return issued;
    }

    /**
     * @modifies this
     * @effects removes the ticket at the front,
     *     or does nothing if no ticket is waiting
     */
    void serve() { if (served < issued) { served = served + 1; } }

    /** @return the number of tickets waiting */
    int waiting() { return issued - served; }

    /** @return the peak */
    int peak() { return peak; }
}
```

Part 1

For each claim, write in its box:

- **T** if it is true of every `Line` whenever no method of the class is running
- **F** if not

Write only your answer in each box, clearly; work anywhere outside them.	**Answer** (**T** / **F**)
a. `served <= issued`	**T**
b. `peak = issued - served`	**F**
c. `issued - served <= peak`	**T**
d. `peak = issued`	**F**
e. `peak <= issued`	**T**

Part 2

Which is the abstraction function? Write its letter in the box. In each option the peak is `peak`, and the tickets waiting, front first, are those numbered:

- **A.** `1` to `issued - served`
- **B.** `served` to `issued - 1`
- **C.** `served + 1` to `issued`
- **D.** `issued` down to `served + 1`

When `served = issued`, each option lists no tickets.

Write only your answer in each box, clearly; work anywhere outside them.	**Answer** (**A** / **B** / **C** / **D**)
f. `AF(this)`	**C**

a.

Answer
T

True: the constructor makes it true, and each method keeps it true.

The constructor makes it true.

- It sets `issued` and `served` to 0, and `0 <= 0`.

Each method keeps it true.

- `take()` adds 1 to `issued` and leaves `served` alone, so `served` stays at most `issued`.
- `serve()` adds 1 to `served` only when `served < issued`, so the new `served` is at most `issued`.
  - Without its `if`, a `serve()` on a new `Line` would make `served` 1 while `issued` is 0.
- `waiting()` and `peak()` change no field.

The definition: Lecture 11's Representation invariants (https://courses.cs.washington.edu/courses/cse331/26au/lectures/lecture11-reps/notes.html#representation-invariants).

b.

Answer
F

False: `peak` is the most tickets ever waiting at once, not the number waiting now.

A counterexample:

- `new Line()` sets all three fields to 0.
- `take()` makes `issued` 1 and `peak` 1.
- `serve()` then makes `served` 1.
- No method is running, and `peak` is 1, but `issued - served` is 0.

The definition: Lecture 11's Representation invariants (https://courses.cs.washington.edu/courses/cse331/26au/lectures/lecture11-reps/notes.html#representation-invariants).

c.

Answer
T

True: the constructor makes it true, and each method keeps it true, though it is false for a moment inside `take()`.

The constructor makes it true.

- It sets all three fields to 0, and `0 - 0 <= 0`.

Each method keeps it true.

- `take()` first adds 1 to `issued`, so `issued - served` grows by 1, and may now be greater than `peak`.
- Its next line sets `peak` to the larger of `peak` and `issued - served`, so the claim holds again before `take()` returns.
- `serve()` adds 1 to `served` or changes nothing, so `issued - served` shrinks or stays the same, and `peak` stays the same.
- `waiting()` and `peak()` change no field.

False inside a method is allowed.

- On a new `Line`, after the first line of `take()`, `issued` is 1 and `peak` is still 0, so the claim is false there.
- The claim need only be true between calls to the class's methods, and it may be false inside one, as Lecture 11 shows inside `FastQueue`'s `next`.

The definition: Lecture 11's Representation invariants (https://courses.cs.washington.edu/courses/cse331/26au/lectures/lecture11-reps/notes.html#representation-invariants).

d.

Answer
F

False: if `serve()` removes the first ticket before the second is taken, the two are never waiting at once, so `peak` is 1 but `issued` is 2.

A counterexample:

- `new Line()` sets all three fields to 0.
- `take()` makes `issued` 1 and `peak` 1.
- `serve()` makes `served` 1.
- `take()` makes `issued` 2, and `issued - served` is 1, so `peak` stays 1.
- No method is running, and `peak` is 1, but `issued` is 2.
- No shorter run breaks the claim: after any one or two calls on a new `Line`, `peak = issued` holds.

The definition: Lecture 11's Representation invariants (https://courses.cs.washington.edu/courses/cse331/26au/lectures/lecture11-reps/notes.html#representation-invariants).

e.

Answer
T

True: the constructor makes it true, and each method keeps it true. The step for `take()` uses one more fact, `0 <= served`.

The constructor makes it true.

- It sets `peak` and `issued` to 0, and `0 <= 0`.

Each method keeps it true.

- `take()` adds 1 to `issued`, then sets `peak` to the larger of `peak` and `issued - served`. Both are at most the new `issued`:
  - `peak`: the claim held when `take()` began, so `peak` was at most the old `issued`, which is less than the new one.
  - `issued - served` is at most `issued`, since `0 <= served`.
- `serve()` changes neither `peak` nor `issued`.
- `waiting()` and `peak()` change no field.

Why `0 <= served`:

- The constructor sets `served` to 0, and the only method that changes it, `serve()`, adds 1, so it never drops below 0.

The definition: Lecture 11's Representation invariants (https://courses.cs.washington.edu/courses/cse331/26au/lectures/lecture11-reps/notes.html#representation-invariants).

f.

Answer
C

C: tickets 1 to `issued` have been taken, the first `served` of them have been removed, and the rest are waiting.

What the fields hold:

- Each `take()` adds 1 to `issued` and returns it, and its specification says it returns the new ticket's number, so the tickets taken are 1 to `issued`.
- Tickets join at the back in number order and leave from the front, so they leave in number order too.
- Each `serve()` that removes a ticket adds 1 to `served`, so the tickets removed are 1 to `served`.
- So the tickets waiting are `served + 1` to `issued`, with the oldest, `served + 1`, in front. That is C.

The count alone cannot decide:

- Every option lists `issued - served` tickets, the number `waiting()` returns.
- They differ in which tickets they list, and in which is in front.

One run tells them apart:

- On a new `Line`, `take()` three times and `serve()` once: ticket 1 was removed, and tickets 2 and 3 are waiting, 2 in front.
- The fields are then `issued` 3, `served` 1, and `peak` 3.
  - A gives tickets 1 and 2: the right count, but numbered from 1.
  - B gives tickets 1 and 2: each one too low.
  - C gives tickets 2 and 3, with 2 in front.
  - D gives tickets 3 and 2: the right tickets, but the newest in front.

The definition: Lecture 11's Abstraction functions (https://courses.cs.washington.edu/courses/cse331/26au/lectures/lecture11-reps/notes.html#abstraction-functions).

Question 6: Subtypes: Polygons

For each class below, decide whether it is a **behavioral subtype** of the interface it implements: every object must satisfy the interface's specification, starting when its constructor returns. All six classes are Java subtypes of their declared interfaces.

Assume all lengths are positive, finite numbers. Clients call methods and do not access fields directly. Treat `double` calculations as exact real arithmetic. You may assume no arithmetic here ever overflows.

Part 1

Consider this interface:

```
/** A convex polygon with equal sides and equal interior angles. */
interface RegularPolygon {
    /** @return this polygon's area */
    double getArea();
    /** @return this polygon's perimeter */
    double getPerimeter();
}
```

For each class, write in its box:

- **T** if it is a behavioral subtype of `RegularPolygon`
- **F** if it is not

Write only your answer in each box, clearly; work anywhere outside them.	**Subtype? (T / F)**

a.
`class Square implements RegularPolygon {`
`  private double s;`
`  Square(double s) { this.s = s; }`
`  public double getArea() { return s * s; }`
`  public double getPerimeter() { return s * 4; }`
`}`
	T

b.
`class RightTriangle implements RegularPolygon {`
`  private double a, b; // the two legs meeting at the right angle`
`  RightTriangle(double a, double b) {`
`  this.a = a; this.b = b;`
`  }`
`  public double getArea() { return a * b / 2; }`
`  public double getPerimeter() {`
`  return a + b + Math.sqrt(a * a + b * b);`
`  }`
`}`
	F

Question 6: Subtypes: Polygons, continued

Part 2

Now consider this interface.

```
/** A convex polygon whose side lengths can be changed. */
interface ResizablePolygon {
    /** @return this polygon's area */
    double getArea();
    /** @return this polygon's perimeter */
    double getPerimeter();
    /**
     * @requires sides is non-null and lists, in order, the positive
     *     side lengths of some convex polygon
     * @modifies this
     * @effects changes this to a convex polygon with those sides,
     *     even if the number of sides differs from before
     */
    void setSides(double[] sides);
}
```

For each class, write in its box:

- **T** if it is a behavioral subtype of `ResizablePolygon`
- **F** if it is not

Write only your answer in each box, clearly; work anywhere outside them.	**Subtype?** (T / F)

c. `class ResizableSquare implements ResizablePolygon {`
`  private double s;`
`  ResizableSquare(double s) { this.s = s; }`
`  public double getArea() { return s * s; }`
`  public double getPerimeter() { return s * 4; }`
`  public void setSides(double[] sides) { s = sides[0]; }`
`}`
	F

Question 6: Subtypes: Polygons, continued

Repeated for your convenience:

Part 2

Now consider this interface.

```
/** A convex polygon whose side lengths can be changed. */
interface ResizablePolygon {
    /** @return this polygon's area */
    double getArea();
    /** @return this polygon's perimeter */
    double getPerimeter();
    /**
     * @requires sides is non-null and lists, in order, the positive
     *     side lengths of some convex polygon
     * @modifies this
     * @effects changes this to a convex polygon with those sides,
     *     even if the number of sides differs from before
     */
    void setSides(double[] sides);
}
```

For each class, write in its box:

- **T** if it is a behavioral subtype of `ResizablePolygon`
- **F** if it is not

Write only your answer in each box, clearly; work anywhere outside them.	**Subtype? (T / F)**

d.
`class ResizableRightTriangle implements ResizablePolygon {`
`  private double a, b; // the two legs meeting at the right angle`
`  ResizableRightTriangle(double a, double b) {`
`  this.a = a; this.b = b;`
`  }`
`  public double getArea() { return a * b / 2; }`
`  public double getPerimeter() {`
`  return a + b + Math.sqrt(a * a + b * b);`
`  }`
`  public void setSides(double[] sides) {`
`  if (sides.length != 3 ||`
`  sides[0] * sides[0] + sides[1] * sides[1]`
`  != sides[2] * sides[2]) {`
`  throw new IllegalArgumentException("invalid");`
`  }`
`  a = sides[0]; b = sides[1];`
`  }`
`}`
	F

Question 6: Subtypes: Polygons, continued

Part 3

Finally, consider this interface:

```
/** A convex polygon with equal sides and equal interior angles. */
interface ResizableRegularPolygon {
    /** @return this polygon's area */
    double getArea();
    /** @return this polygon's perimeter */
    double getPerimeter();
    /**
     * @requires 0 < len
     * @modifies this
     * @effects sets every side's length to len, keeping the number
     *     of sides and keeping the polygon regular
     */
    void setSideLength(double len);
}
```

For each class, write in its box:

- **T** if it is a behavioral subtype of `ResizableRegularPolygon`
- **F** if it is not

Write only your answer in each box, clearly; work anywhere outside them.	**Subtype?** (**T** / **F**)
e. `class ResizableSquare2 implements ResizableRegularPolygon { private double s; ResizableSquare2(double s) { this.s = s; } public double getArea() { return s * s; } public double getPerimeter() { return s * 4; } public void setSideLength(double sideLength) { s = sideLength; } }`	**T**
f. `class ResizableRightTriangle2 implements ResizableRegularPolygon { private double a, b; // the two legs meeting at the right angle ResizableRightTriangle2(double a, double b) { this.a = a; this.b = b; } public double getArea() { return a * b / 2; } public double getPerimeter() { return a + b + Math.sqrt(a * a + b * b); } public void setSideLength(double len) { a = len; b = len; } }`	**F**

a.

Subtype?
T

True: every `Square` satisfies the interface's promises.

- A square has four equal sides and four equal interior angles.
- For side length `s`, its area is `s * s` and its perimeter is `4 * s`, exactly what the methods return.
- Neither method changes the square.

The rule: a behavioral subtype keeps every promise (https://courses.cs.washington.edu/courses/cse331/26au/lectures/lecture14-subtyping/notes.html#a-behavioral-subtype-keeps-every-promise).

b.

Subtype?
F

False: a `RightTriangle` is not regular.

- `new RightTriangle(3, 4)` meets the constructor's assumptions.
- Its sides are unequal, so it violates `RegularPolygon`'s promise immediately after construction.
- Correct area and perimeter methods do not repair that violation.

The rule: a behavioral subtype keeps every promise (https://courses.cs.washington.edu/courses/cse331/26au/lectures/lecture14-subtyping/notes.html#a-behavioral-subtype-keeps-every-promise).

c.

Subtype?
F

False: `setSides` does not install the requested sides.

- Start with `new ResizableSquare(2)`.
- Call `setSides(new double[] {2, 3, 2, 3})`.
- These are the sides of a convex rectangle, so the call meets the interface's precondition and should leave a perimeter of 10.
- The method keeps only the first length, 2, leaving a square with perimeter 8. It breaks the postcondition.

The rule: a behavioral subtype keeps every promise (https://courses.cs.washington.edu/courses/cse331/26au/lectures/lecture14-subtyping/notes.html#a-behavioral-subtype-keeps-every-promise).

d.

Subtype?
F

False: `setSides` rejects an input the interface permits.

- Start with `new ResizableRightTriangle(3, 4)`.
- Call `setSides(new double[] {2, 2, 2, 2})`.
- These are the sides of a convex square, so the call meets the interface's precondition.
- The method throws instead of changing the polygon as promised. It accepts only right triangles with the hypotenuse last, a stronger precondition.

The rule: a behavioral subtype keeps every promise (https://courses.cs.washington.edu/courses/cse331/26au/lectures/lecture14-subtyping/notes.html#a-behavioral-subtype-keeps-every-promise).

e.

Subtype?
T

True: construction and every method keep all the interface's promises.

- The constructor creates a square, which is a regular polygon.
- `setSideLength` accepts every positive length and changes all four sides to that length, preserving regularity and the number of sides.
- The area and perimeter methods return `s * s` and `4 * s` for the current side length and change no state.

The rule: a behavioral subtype keeps every promise (https://courses.cs.washington.edu/courses/cse331/26au/lectures/lecture14-subtyping/notes.html#a-behavioral-subtype-keeps-every-promise).

f.

Subtype?
F

False: the right triangle is not regular, even after resizing.

- `new ResizableRightTriangle2(3, 4)` meets the constructor's assumptions.
- Its sides are unequal, violating the interface's promise before any call to `setSideLength`.
- `setSideLength(len)` sets the two legs to `len`, but the hypotenuse is `Math.sqrt(2) * len`. It neither sets every side to `len` nor makes the triangle regular.

The rule: a behavioral subtype keeps every promise (https://courses.cs.washington.edu/courses/cse331/26au/lectures/lecture14-subtyping/notes.html#a-behavioral-subtype-keeps-every-promise).

Question 7: Implementing Equality for a Polyline

A final class `Polyline` represents an immutable, ordered sequence of points. Two polylines are equal exactly when their point sequences have the same size and equal points at every index. Adding a repeated point changes the abstract value, even if the line traces the same path.

The class has two fields:

- `List<Point> points` : at least two non-null, immutable points. Repeated points are allowed. Each point has integer coordinates; `Point.equals` compares both coordinates.
- `double total_length` : the sum of the Euclidean distances between consecutive points. The constructor always computes this finite value from the list, starting at 0.0 and adding distances from left to right. It is never stale.

The constructor copies the list. No method exposes or changes it. You may assume no arithmetic here ever overflows.

`List.equals` compares sizes and then corresponding elements with `equals` . `get(i)` throws `IndexOutOfBoundsException` unless `0 <= i` and `i < size()` . The class uses this hash function:

```
    @Override
    public int hashCode() {
        return points.size();
    }
```

Each row replaces the comment in this method:

```
@Override
public boolean equals(Object o) {
    if (!(o instanceof Polyline)) return false;
    Polyline other = (Polyline) o;
    // Insert the proposed body here.
}
```

Consider proposals independently. A good method must terminate normally, return true exactly for a `Polyline` with the same abstract value, satisfy Object's equals contract, and leave both objects unchanged.

For each proposed implementation, write in its box:

- **T** if it is a good equals method for this Polyline specification
- **F** if it is not

Write only your answer in each box, clearly; work anywhere outside them.	**Good equals? (T / F)**
a. `return points.equals(other.points);`	**T**

b. `int n = Math.min(points.size(), other.points.size());`
`for (int i = 0; i < n; i++) {`
`  if (!points.get(i).equals(other.points.get(i))) return false;`
`}`
`return true;`
	F

c. `return hashCode() == other.hashCode();`	**F**
d. `return points.equals(other.points)` `  && total_length == other.total_length;`	**T**

a.

Good equals?
T

True: the list determines the abstract value.

- `List.equals` requires the same size and equal points in order.
- Point equality compares the two coordinates.
- These are exactly the requirements for equal polylines here.
- The derived length contains no additional information needed to decide equality.

The definition: Lecture 16 on equality of abstract values (https://courses.cs.washington.edu/courses/cse331/26au/lectures/lecture16-equality/notes.html#equal-means-the-same-abstract-value).

b.

Good equals?
F

False: a proper prefix compares equal to the longer list.

- Let `x` hold `[(0, 0), (1, 0)]` and `y` hold `[(0, 0), (1, 0), (1, 1)]`.
- The minimum size is 2, so the loop checks only indices 0 and 1.
- Both comparisons succeed, and the method returns true.
- The extra point in `y` makes these different abstract values.

The definition: Lecture 16 on equality of abstract values (https://courses.cs.washington.edu/courses/cse331/26au/lectures/lecture16-equality/notes.html#equal-means-the-same-abstract-value).

c.

Good equals?
F

False: unequal polylines can have equal hash codes.

- Let `x` hold `[(0, 0), (1, 0)]` and `y` hold `[(0, 0), (0, 1)]`.
- Both lists have size 2, so both hash codes are 2.
- The method returns true, but the second points differ.
- Hash compatibility requires equal values to have equal hash codes. It does not require the converse.

The rule: Lecture 16 on equals and hashCode (https://courses.cs.washington.edu/courses/cse331/26au/lectures/lecture16-equality/notes.html#the-override-and-the-hash-code).

d.

Good equals?
T

True: the length comparison is redundant.

- `List.equals` compares list sizes and corresponding points.
- If the lists differ, the method returns false.
- If the lists agree, their segment distances agree in order.
- The constructor performs the same sum in the same order, so the finite cached lengths agree too, even if rounding occurs.
- Thus the extra comparison never rejects equal abstract values.

The definition: Lecture 16 on equality of abstract values (https://courses.cs.washington.edu/courses/cse331/26au/lectures/lecture16-equality/notes.html#equal-means-the-same-abstract-value).