24wi Final Review SOLUTIONS

These practice questions are from CSE 331 2024 Winter. Questions relating to content that was not covered in 2025 Summer were removed. The remaining questions are similar to the style of questions you can expect to see on the 25SU exam and cover topics that are fair game, but it is not a perfect representation of what topics will be emphasized in the 25SU exam or the length of a complete exam.

We recommend that you attempt these questions in the non-solutions version first before looking at the answers. Please read the explanations (if included) to make sure you understand the answers, and ask questions on Ed for further clarification.

Question 1 : Testing Subdomains

a. For the following function, select one input that is **not** required to test using the 331 Testing requirements (assuming all other options would be tested)? [multiple correct answers]

```
const foo = (x: number) : number => {
  if (x < 0) {
    const a = x * 2;
    const b = a - 2;
    return b * 3 + 1;
  } else if (x > 10) {
    return 100 * x  - x / 2
  } else {
    return 55;
  }
}
```

A. 0
B. 13
C. 1
D. -1

Explanation: To reach statement and branch coverage, we must include test -1 for coverage of the first branch and 13 for coverage of the second branch. Then, for coverage of the 3rd branch 0 and 1 would each be valid inputs, so if either is tested, the other wouldn't be necessary under the **minimum** 331 testing requirements. Of course, testing both is perfectly fine, and doing so also gets us closer to meeting the boundary testing heuristic which is an optional (in 331) but recommended heuristic.

b. For the following function, what is the **minimum** number of tests needed?

```
const bar = (x: number) : number => {
  if (x === 0) {
    return 0;
  } else {
    if (x % 2 === 0) {        // x is even
      return bar(x / 2);
    } else {                  // x is odd
      return bar(x - 1);
    }
  }
}
```

A. 1
B. 2
C. 3
D. 4
E. 5
F. 6
G. 7
H. 8+

Explanation: This is a recursive function, so we know we need to test 0, 1, and 2+ recursive calls (before hitting the base case). Since there are recursive calls in 2 branches, your first instinct should be to satisfy 1 and 2+ recursive calls for both, however, in the “x is even” branch, it is not possible to enter that branch, make 1 recursive call, and then hit the base case (because no non-zero, even number / 2 = 0). So, we only need a test for 2+ recursive calls in the “even” branch (e.g. 6), a test for 1 recursive call in the “odd” branch (1), a test for 2+ recursive calls in the “odd” branch (e.g. 3), and a base case test (0).

Satisfying this 0-1-2+ heuristic will also satisfy statement and branch coverage.

Question 2: Types

Pick the best, most precise definition for what this inductive data type represents.

$\text{type T} = \text{A(x: } \mathbb{Z}\text{)} \mid \text{B(x: } \mathbb{Z}\text{, y: T, z: T)}$

A. Lists of integers
B. Non-empty lists of integers
C. Binary trees of integers
D. Non-empty binary trees of integers

Explanation: The base constructor has an integer, so we know it represents something non-empty. The recursive constructor has two T fields, so we can think of it as a node with two children. And it has one integer field. So in our data type, we have **integers**, and we have **integer nodes with two children**. That is a non-empty binary tree of integers.

Question 3: Proof By Calculation

For each of the proofs below, select the correct option to fill in the blank (____) space in order for the proof to be deemed correct.

Prove that $\text{sum}(\text{twice}(L)) = 2\text{sum}(L)$ where $L = \text{cons}(a, \text{cons}(b, \text{nil}))$

$\text{sum}(\text{twice}(L)) = \text{sum}(\text{twice}(\text{cons}(a, \text{cons}(b, \text{nil}))))$	__ P1 __.
$= \text{sum}(\text{cons}(2a, \text{twice}(\text{cons}(b, \text{nil}))))$	def of twice
$= \text{sum}(\text{cons}(2a, \text{cons}(2b, \text{twice}(\text{nil}))))$	def of twice
$= \text{sum}(\text{cons}(2a, \text{cons}(2b, \_\_\text{P2}\_\_)))$	def of twice
$= 2a + \text{sum}(\text{cons}(2b, \text{nil}))$	def of sum
$= 2a + 2b + \text{sum}(\text{nil})$	def of sum
$= 2a + 2b + 0$	def of sum
$= 2(a + b + 0)$	algebra
$= 2(a + b + \_\_\text{P3}\_\_)$	def of sum
$= 2(a + \text{sum}(\text{cons}(b, \text{nil})))$	def of sum
$= 2(\text{sum}(\text{cons}(a, \text{cons}(b, \text{nil}))))$	def of sum
$= 2(\text{sum}(L))$	def of L

Part 1:
What should go in the blank *reasoning* space on the 1st line of the proof:

A. def of List
B. def of cons
C. def of L
D. def of sum

Explanation: Here, we're substituting in the value of L from its definition: $L = \text{cons}(a, \text{cons}(b, \text{nil}))$

Part 2:
What should go in the blank space on the 4th line of the proof: $\text{sum}(\text{cons}(2a, \text{cons}(2b, \_\_\text{P2}\_\_)))$

A. $\text{twice}(0)$
B. 0
C. nil
D. $\text{twice}(\text{nil})$

Explanation: From the definition of twice, we know that twice(nil) = nil.

Part 3:
What should go in the blank space on the 9th line of the proof: $2(a + b + \_\_\text{P3}\_\_)$

A. $\text{sum}(\text{nil})$
B. nil
C. 0
D. $\text{sum}(0)$

Explanation: From the definition of sum, we know that sum(nil) = 0.

Question 5

For each of the programs below, determine whether there will be a type error using both structural (as in TypeScript) and nominal (as in Java) typing. Assume the following types and variables are in scope for all of the programs.

```
type A = {x: bigint}
type B = {x: bigint}
type C = {x: number}
type D = {x: bigint, y: bigint}
type E = {y: bigint, x: bigint}

const a : A = {x: 123n}
const b : B = {x: 456n}
const c : C = {x: 789}
const d : D = {x: 1n, y: 2n}
const e : E = {y: 3n, x: 4n}
```

Program	Structural Typing Error (Yes/No)	Nominal Typing Error (Yes/No)
`v0: E = {x: 12n, y: 34n}`	**No**	**No**
`v1 : B = a`	**No**	**Yes**
`v2 : C = b`	**Yes**	**Yes**
`v3 : D = e` `v4 : E = d`	**No**	**Yes**
`v5 : A = e`	**No**	**Yes**
`v6 : E = a`	**Yes**	**Yes**

`foo = (x: E) : D => {`
`  return {x: 123n, y: 456n}`
`}`
`v7 : D = foo(e)`
	No	**No**

`foo = (x : A) : B => {`
`  return (x.x > 0n) ? {x: 1n} : {x: -1n}`
`}`
`bar = (x : B) : C => {`
`  return (x.x === 0n) ? {x: -1.5} : {x: 2 ** 32}`
`}`
`v8 : C = bar(foo({x: 0n}))`
	No	**No**

`v9 : A = {x: null}`	**Yes**	**Yes**
`v10 : B = {x : 0n}` `v11 : A = {x : v10.x}`	**No**	**No**

Question 6: Equality

For each of the expressions below, write the boolean value (T/F) that the expression will evaluate to.

Code	Eval (T/F)
`3n === 3`	**F**

Explanation: === doesn't do type conversion. 3n and 3 are not the same type, so they will never be equal.

`"I love 331" === "I love 331"`	**T**

Explanation: Strings are primitives in TypeScript, so we can compare them for equality using == and ===

`5n == 5`	**T**

Explanation: == converts the arguments to be the same type first, and 5n and 5 represent the same number, so they are equal.

`10.0 === 10`	**T**

Explanation: 10.0 is equal to 10

`false === 5 < 4`	**T**

Explanation: 5 < 4 evaluates to false, so it is equal to false

`const a: {x: bigint, s: string} = {x: 4n, s: "hello"};`
`const b: {x: bigint, s: string} = {x: 5n, s: "hello"};`
 	
`a === b`
	F

Explanation: When applied to objects, === compares by *reference*, so two objects will never be equal unless they point to literally the same point in memory. a and b are not equal because they refer to two different objects

`const a: {x: bigint, s: string} = {x: 5n, s: "hello"};`
`const b: {x: bigint, s: string} = {x: 5n, s: "hello"};`
 	
`a === b`
	F

Explanation: When applied to objects, === compares by *reference*, so two objects will never be equal unless they point to literally the same point in memory. a and b are not equal because they refer to two different objects

`const a : {b: boolean, s: string,  x: bigint}`
`  = {b: true,  s: "okapi", x: 1n};`
 	
`const b : {a: bigint, b: string,  c: boolean}`
`  = {a: 1n,  b: "giraffe", c: false};`
 	
`b.a === a.x`
	T

Explanation: a.x is the bigint 1n, and b.a is also 1n, so they are equal

`"101" === 101`	**F**

Explanation: === does not do type conversion, so a string and a number will never be equal

`"horses" === "horse" + "s"`	**T**

Explanation: The right-hand side evaluates to "horses", so it is equal to the string literal "horses"

`const x = 5/2;`
`const y = 2;`
 	
`x === y`
	F

Explanation: Since these are floating point numbers, 5/2 evaluates to 2.5, which is not equal to 2.

`undefined === null`	**F**

Explanation: === does not do type conversion. undefined and null are different types, so they are not equal. Note that `undefined == null` returns true, which is potentially confusing and may be a source for bugs. This is an example of why you should prefer === over == whenever possible.

Question 8: Inductive Data Types in TypeScript

For each of the following, select True if the TypeScript type is a correct encoding of the mathematical inductive data definition.

Math Definition	TypeScript Encoding	Match? T/F

$\text{type X} = \text{A} \mid \text{B} \mid \text{C}(\text{x}: \text{X})$	`type X =`
	`  {kind: "A"}`
	`| {kind: "B"}`
	`| {kind: "C"}`
		F

Explanation: The record type for the C constructor is missing the x field.

$\text{type Y} = \text{A}(\text{n}: \mathbb{Z}) \mid \text{B}(\text{n}: \mathbb{Z})$	`type Y =`
	`  A(n: bigint)`
	`| B(n: bigint)`
		F

Explanation: This is not syntactically-valid TypeScript

$\text{type Z} = \text{A} \mid \text{B}(\text{z} : \text{Z})$	`type Z =`
	`  {kind: "A"}`
	`| {kind: "B", z: Z}`
		T

Explanation: This is the correct encoding. Each constructor corresponds to a record type, and the inductive data type is the union of these record types.

$\text{type U} =$
$\text{A}(\text{x}: \mathbb{Z}) \mid \text{B}(\text{x}: \mathbb{Z}, \text{y}: \text{U}, \text{z}: \text{U})$
	`type U =`
	`  {kind: "A", x: bigint}`
	`| {kind: "B", x: bigint, y: U, z: U }`
		T

Explanation: This is the correct encoding. Each constructor corresponds to a record type, and the inductive data type is the union of these record types.

$\text{type V} = \text{A}(\text{r}: \mathbb{R}) \mid \text{B}(\text{n}: \mathbb{Z})$	`type V =`
	`  {r: number}`
	`| {n: integer}`
		F

Explanation: The record types are missing the kind field, so it doesn't properly encode the constructors.

Question 9: Implications

For each of the following, answer T or F based on if the facts imply the obligation.

Facts	Obligation	Do the facts imply the obligation? T/F
For $x, y, z : \mathbb{N}$, $x = y$ and $2y > z$	$x > z/3$	**T**

Explanation: The obligation is claiming something about x, so let's start a proof by calculation from there.

$x = y$ [given]
$> z/2$ [dividing 2 from both sides from $2y > z$]

We've shown $x > z/2$, so $x > z/3$ must be true (given that $x, y,$ and z are natural numbers.)

For x, y: $\mathbb{Z}$, $x \geq 9$ and $y \geq -10$	`return x + y` returns a positive integer	**F**

Explanation: The obligation is claiming something about $x + y$, so let's start a proof by calculation from there.

$x + y \geq x - 10$ [since $y \geq -10$]
$\geq 9 - 10$ [since $x \geq 9$]
$= -1$

We've shown $x + y \geq -1$, so `return x + y` will not always return a positive integer.

For x, y: $\mathbb{Z}$, $x + y \leq -5$ and $y = 2$	$-x \leq 7$	**F**

Explanation: The obligation is claiming something about $-x$, so let's start a proof by calculation from there.

$$\begin{aligned} -x &\geq 5 + y && [\text{since } x + y \leq -5] \\ &\geq 5 + 2 && [\text{since } y = 2] \\ &= 7 \end{aligned}$$

We've shown $-x \geq 7$, so $-x \leq 7$ is false.

For x, y, z : $\mathbb{Z}$, $x = z - y$ and $y \geq 0$	$x \leq z$	**T**

Explanation: The obligation is claiming something about x, so let's start a proof by calculation from there.

$$\begin{aligned} x &= z - y && [\text{given}] \\ &\leq z - 0 && [\text{since } y \geq 0 \text{ and we are subtracting y from z}] \\ &= z \end{aligned}$$

We've exactly shown $x \leq z$, it is true.

Question 10: Structural Induction

The following proof is incorrect. In one sentence, identify an error in the proof. In another single sentence, explain how to fix the proof either by adding, changing, or removing steps.

$$\begin{aligned}
&\textbf{func}\ \text{concat}(\text{nil}, R) && := R && \text{for any } R : \text{List}\\
&\quad \text{concat}(\text{cons}(x, L), R) && := \text{cons}(x, \text{concat}(L, R)) && \text{for any } x : \mathbb{Z} \text{ and any } L, R : \text{List}
\end{aligned}$$

Prove that $\text{len}(\text{concat}(S, R)) = \text{len}(S) + \text{len}(R)$
This proof is by structural induction on S.

Base Case: $S = \text{nil}$.
Goal is to show $\text{len}(\text{concat}(\text{nil}, R)) = \text{len}(\text{nil}) + \text{len}(R)$

$$\begin{aligned}
\text{len}(\text{concat}(\text{nil}, R)) &= \text{len}(R) && \text{def of concat}\\
&= 0 + \text{len}(R)\\
&= \text{len}(\text{nil}) + \text{len}(R) && \text{def of len}
\end{aligned}$$

Inductive Hypothesis: assume that $\text{len}(\text{concat}(L, R)) = \text{len}(L) + \text{len}(R)$

Inductive Step $S = (\text{cons}(x, L))$.
Goal is to show $\text{len}(\text{concat}(\text{cons}(x, L), R) = \text{len}(\text{cons}(x, L)) + \text{len}(R)$

$$\begin{aligned}
\text{len}(\text{concat}(\text{cons}(x, L), R)) &= \text{len}(\text{cons}(x, \text{concat}(L, R))) && \text{def of concat}\\
&= 1 + \text{len}(L) + \text{len}(R) && \text{Ind. Hyp.}\\
&= \text{len}(\text{cons}(x, L)) + \text{len}(R) && \text{def of len}
\end{aligned}$$

Explanation:
In the inductive step, there is a skipped step. Before applying the Inductive Hypothesis, we need to use the definition of len:
len(cons(x, concat(L, R))) = 1 + len(concat(L, R))

We cannot apply the Inductive Hypothesis to the term len(cons(x, concat(L, R))) directly.

Question 11: Induction Hypothesis

Recall the definition of Trees: $\text{type Tree} := \text{leaf} \mid \text{node}(x : \mathbb{Z}, L : \text{Tree}, R : \text{Tree})$
Suppose we are doing a proof of some property P by structural induction on S : Tree.
What induction hypothesis/es can we assume in the inductive case of the proof, where we have $S = \text{node}(x, L, R)$?

A. P(S)
B. P(L)
C. P(R)
D. P(L) and P(R)
E. P(L) and P(S)
F. P(R) and P(S)

Explanation:
The correct answer is D: P(L) and P(R).
In the inductive case, you can assume that the property holds for the smaller elements that are arguments to the constructor. In the case of trees, that is the left subtree and the right subtree.
It is *not* okay to assume P(S)-- that's what we're trying to prove!

Question 12: Generic Types

Consider these types:

How many distinct values are there of each type (e.g., “zero”, “one”, “two”, ..., “infinity”)?

```
type A =
    | { kind: "M", value: A };
```

Explanation: To use the M constructor, you need to already have an A, and there’s no base constructor, so there’s no way to actually make a value of type A. Therefore, there are zero values in this type.

```
type B =
    | { kind: "P" }
    | { kind: "Q" };
```

Explanation: There are two values of type B: P and Q. There are two constructors, and neither of them take any arguments, so there are only two values.

```
type C =
    | { kind: "CA", value: bool }
    | { kind: "CB", value: B };
```

Explanation: There are 4 values of type C. There are two constructors, so we consider how many values there are of each constructor, and add these. There are two values that are possible to build with the CA constructor: CA(true) and CA(false). There are two values that are possible to build with the CB constructor: CB(P) and CB(Q). In total, that gives 4 possible values.

```
type D =
    | { kind: "DA", value: ℤ }
    | { kind: "DB", value: bool };
```

Explanation: There are infinite values of type D. Again, with two constructors, we add the number of values from each constructor. There are two values from constructor DB: DB(true) and DB(false). There are infinite values that can be built with the DA constructor. Since it takes an integer value, and there are infinite integers, there are infinite values.

```
type E =
    | { kind: "EA" }
    | { kind: "EB", v1: [E, B] };
```

Explanation: There are infinite values of type E. Here, we have one base constructor and one recursive constructor. We can apply the recursive constructor an arbitrary number of times, so there are infinite values in this type. This type represents lists of B elements. There is a nil case (the EA constructor) and a cons case (the EB constructor takes a B and another element of type E).

```
type F =
    | { kind: "FA", v1: F, v2: F] }
    | { kind: "FB" };
```

Explanation: There are infinite values of type F. This is pretty similar to the previous type. There is one base constructor (FB) and one recursive constructor (FA), which can be applied arbitrarily many times to construct values of type F. This type represents trees with no data.

Question 13. Comparing Specifications

Here are four different specifications.

Spec A
@param x, a natural number
@returns a natural number between 1 and 10

Spec B
@param x, a natural number
@requires x is even
@returns a natural number between 1 and 10

Spec C
@param x, an integer between 0 and 10
@returns an integer between 0 and 20

Spec D
@param x, a natural number
@returns an integer between -10 and 10

For each pair of specs, fill in the blank if the first spec is stronger than, weaker than, or incomparable to the second.

Explanations (In general):
Strengthening cannot break the clients

- A stronger spec accepts the original inputs (or more inputs)
- A stronger spec makes the original promises about outputs (or more)

Weakening cannot break the implementation

- A weaker spec does not allow more (new) inputs
- A weaker spec does not add more promises about outputs

A is _____**stronger**_____ than/to B
Explanation: A is stronger than B. They both make the same guarantees on the output (an integer between 1 and 10), but A makes fewer assumptions about the inputs (accepts more inputs than B). A client using B can never break if they switch to A, but a client using A might break if they switch to B (for example, if they call the function with an odd number, which is allowed by A but not allowed by B).

A is _____**stronger**_____ than/to C
Explanation: A returns fewer outputs than C. A accepts more inputs than C. Therefore, A is stronger than C.

B is _____**incomparable**_____ than/to C
Explanation:: B and C are incomparable because they have different requirements on their inputs. B accepts any even numbers while C accepts even or odd inputs but the input must be between 0 and 10.

B is _____**incomparable**_____ than/to D
Explanation: B accepts fewer inputs (only even nats compared to any nat). However, B also returns fewer outputs (1 - 10 compared to -10 - 10). So they are incomparable. A client could break switching from B to D (for example, the client could be relying on the fact that B returns a positive number, which D does not). A client could also break switching from D to B (for example, it could be calling the function with an odd input).

C is _____**incomparable**_____ than/to D
Explanation: The specs make incomparable guarantees about the output range. A client could break in either direction.

D is _____**weaker**_____ than/to A
Explanation: A and D make the same assumptions about the inputs (accept the same inputs). However, A makes more guarantees about the outputs (produces fewer outputs). So A is stronger than D (D is weaker than A).

Question 14. ADTs

Consider the following public specification for an ADT representing a Rectangle:

```
// A Rectangle is represented by a triple (p, l, w) where:
//   p is the location of the top-left corner, l is the length, and w is the width
export interface Rectangle {
  length: () => bigint;
  width: () => bigint;
  topLeft: () => {x: bigint, y: bigint};
}
```

For each of the following concrete representations of the Rectangle abstract data type, write down the abstraction function (AF) and representation invariant (RI), if any.

```
class TopLeftRectangle implements Rectangle {
  // AF: obj = (p = (this.topLeft.x, this.topLeft.y), l = this.length,
  //           w = this.width)
  // RI: this.length > 0 & this.width > 0
  readonly topLeft: {x: bigint, y: bigint};
  readonly length: bigint;
  readonly width: bigint;
}
```

Explanation: In this implementation, we represent the rectangle using p, l, and w, so the abstraction function just needs to map from the record field names to the mathematical tuple (p, l, w).

The representation invariant is that length and width are positive, because negative values don't make sense for a rectangle.

```
class OppositeCornersRectangle implements Rectangle {
  // AF: obj = (p = (this.topLeft.x, this.topLeft.y),
  //            l = this.bottomRight.x - this.topLeft.x,
  //            w: this.topLeft.y - this.bottomRight.y)
  // RI: this.bottomRight.x > this.topLeft.x & this.topLeft.y > this.bottomRight.x
  readonly topLeft: {x: bigint, y: bigint};
  readonly bottomRight: {x: bigint, y: bigint};
}
```

Explanation: In this implementation, we represent the rectangle using two corners. So to map from these two points to the mathematical tuple (p, l, w), we need to compute the length and width.

The representation invariant constrains that the topLeft point is actually above and to the left of

the bottomRight point.

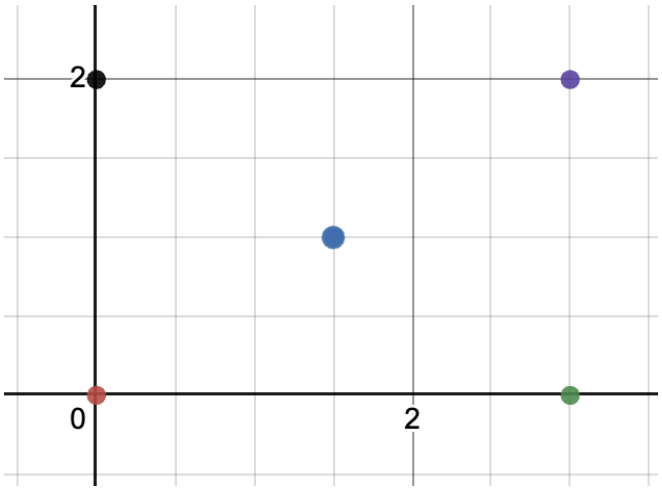
```
class CenterRectangle implements Rectangle {
  // AF: obj = (p = (this.center.x - this.length / 2, this.center.y +
  //      (this.area/this.length/2)), l = this.length, w = this.area / this.length)
  // RI: this.length > 0 and this.area > 0 and this.area % this.length = 0
  readonly center: {x: bigint, y: bigint};
  readonly length: bigint;
  readonly area: bigint;
}

Bonus: Give an example of a Rectangle you can represent in the other two
implementations but not this one
```

Explanation: In this implementation, we represent the rectangle by its midpoint, length, and area. To map from these values to the (p, l, w) mathematical representation, we need to compute the width as area / length and the location of the top-left corner by subtracting half the length and half the width from the midpoint's x and y coordinates, respectively.

The representation invariant requires that the length and area are both positive. We also require that area / length is an integer, so that the width is representable by an integer.

Bonus: We cannot represent the rectangle whose top left corner is at (0, 2) with a length 3 and width 2. Its center point would be at (1.5, 1), but that's not representable using bigints.



Question 15: Hoare Triples

For each of the following Hoare Triples, determine whether the triple is valid.

{{ $x < 0$ }} `y = 2n * x;` {{ $y \leq 0$ }}	**Valid**

Explanation: After the assignment, we know x < 0 and y = 2x. Thus, we know y ≤ 0.

{{ $x \geq y$ }} `z = x - y;` {{ $z > 0$ }}	**Invalid**

Explanation: If x = y, then z = 0, and z > 0 will not hold.

{{ True }}
`if ( x >= 10n) {`
`  y = x % 7n;`
`} else {`
`  y = x - 1n;`
`}`
{{ $y < 9$ }}
	Valid

Explanation: If x ≥ 10, then y will be x % 7. In this case 0 ≤ y < 7. If x < 10, then y will be x - 1. So after the conditional, we'll have {{ x ≥ 10 and 0 ≤ y < 7 or x < 10 and y = x - 1 }}. From here, we can prove by cases that y < 9 (in the first case, it follows directly from 0 ≤ y < 7. in the second case, we can show y < 10 - 1 (since x < 10).)

{{ $x < 0$ }}
`if (x < 100n) {`
`  x = -1n;`
`} else {`
`  x = 1n;`
`}`
{{ $x < 0$ }}
	Valid

Explanation: The postcondition here would not hold if the else branch executed (it is not true that 1 < 0). However, that branch is impossible to reach since x < 0 and the condition is x < 100. So the postcondition we have after the conditional is {{ $x_0 < 0 \wedge x_0 < 100 \wedge x = -1 \vee (x_0 < 0 \wedge x_0 \geq 100) \wedge x = 1$ }}. Since the second clause of the or is always False, we just have to show that the first implies the postcondition, which it does because -1 < 0.

`/**`
`* @param n: a natural number`
`* @returns a natural number`
`*/`
	Invalid

`const foo = (n: bigint): bigint => {`
`  …`
`}`
 	
// some other code that calls foo
{{ x ≥ 0 /\ y ≥ 0 }}
`if (x < y) {`
`  y = -1n * x`
`}`
`z = foo(y)`
{{ z ≥ 0 }}	

Explanation: This one is invalid because it might not be valid to call `foo` on argument `y`. If x < y, then we set y to -x. In this case, we'd then know that y ≤ 0 (since x ≥ 0). However, foo requires the input to be a natural number, so calling it with y ≤ 0 is not valid. If we call the function with an invalid input, we cannot make any claims about what it returns.

Question 17: Weakest Assertion

Select the weakest assertion in each set. The first row is filled in as an example.

Assertions	Weakest
A. {{ True }} B. {{ False }} C. {{ z = 42 }}	A
A. {{ x = 20 }} B. {{ x > 10 }} C. {{ x ≥ 10 }}	**C**
A. {{ t = 2 }} B. {{ t ≠ 0 }} C. {{ t > 0 }}	**B**
A. {{ x > 0 /\ y > 0 }} B. {{ x > 0 \/ y > 0 }}	**B**
A. {{ |x + y| > w }} B. {{ x + y > w }}	**A**

Question 18: Array Loop Invariants

```
/**
* Swaps two elements in an array
* @param A The array in which to swap elements
* @modifies A
* @param i The index of the first element
* @param j The index of the second element
* @requires 0 <= i < A.length /\ 0 <= j < A.length
*/
const swap = (A: bigint[], i: number, j: number): void => {
 const tmp = A[i];
 A[i] = A[j];
 A[j] = tmp;
}
/**
* Sorts an array in place
* @param A
* @modifies A
*/
const sort = (A: bigint[]): void => {
 let i = 0;
  // Inv: TODO
 while (i < A.length) {
   let j = i;
   // Inv: TODO
   while (j < A.length) {
     if (A[i] > A[j]) {
       swap(A, i, j);
     }
     j = j + 1;
   }
   i = i + 1;
 }
// Postcondition: A[x] ≤ A[y] for all 0 ≤ x < A.length and x ≤ y < A.length
```

What is the correct invariant for the outer loop?

A. $A[x] \leq A[y]$ for all $0 \leq x \leq i$ and $x \leq y \leq i$ and $0 \leq i < A.length$

B. $A[x] \leq A[y]$ for all $0 \leq x \leq i$ and $x \leq y \leq A.length$ and $0 \leq i < A.length$

C. $A[x] < A[y]$ for all $0 \leq x < i$ and $x \leq y < i$ and $0 \leq i < A.length$

D. $A[x] < A[y]$ for all $0 \leq x < i$ and $x \leq y < A.length$ and $0 \leq i < A.length$

E. $A[x] \leq A[y]$ for all $0 \leq x < i$ and $0 \leq i < A.length$ and $x \leq y < A.length$

Explanation: The intuition for this invariant is that it's saying "the array is sorted up to i". We start i at 0 (nothing is sorted) and we end when i = arr.length (everything is sorted). It's also a **weakening of the postcondition** which is a common pattern for invariants and a good clue here.

What is the correct invariant for the inner loop?

A. $A[x] \leq A[j]$ for all $i < x < j$ and $0 \leq i < j < A.length$

B. $A[x] \leq A[j]$ for all $i \leq x \leq j$ and $0 \leq i < j < A.length$

C. $A[x] \leq A[y]$ for all $i \leq x < j$ and $x \leq y < j$ and $0 \leq i < j < A.length$

D. $A[x] \leq A[j]$ for all $i \leq x < A.length$ and $0 \leq i < j < A.length$

E. $A[i] \leq A[j]$ for all $0 \leq i < j < A.length$

Explanation: The intuition for this invariant is: “A[i] is smaller than everything between A[i] and A[j]”.

Question 19: Servers and Routes

For each of the following, mark whether it applies to client/server interactions, normal function calls, both, or neither

	Client/Server Interaction	Normal Function Call
The public specification should be documented	✅	✅

Explanation: Both public functions and public server APIs need to be documented so that callers know how to call them correctly.

It is asynchronous	✅	❌

Explanation: Client/Server interactions are *always* asynchronous. Requests are sent over the network (which takes time). The server listens for requests and then sends a response back over the network (which also takes time). In normal function calls, when there's code that calls a function, it doesn't proceed to the next line until it gets the return value from the function (this is called "blocking").

The arguments must be serialized to JSON or text	✅	❌

Explanation: Normal function calls can take any kind of arguments (records, inductive data types, functions, etc). Requests and responses sent over the network must be serialized as either JSON or text.

It can throw an error	✅	✅

Explanation: Errors can be thrown in either :)

The function and the code that calls it are implemented in different languages	❌	❌

Explanation: While it is possible to have a client and server implemented in different languages, it's not necessarily true. For example, in this class, we will implement both client-side and server-side code in TypeScript. On the flip side, while it's often true for normal function calls that the caller and the function are implemented in the same language, it is possible to call a function written in another language using something called a Foreign Function Interface

Question 20: Stateful UI

```
type FooState = { items: string[] };

class Foo extends Component<{}, TodoState> {
...
// Called when the user clicks on the button to clear all items.
doClearClick = (_: MouseEvent<HTMLButtonElement>): void => {
       const name = this.state.input.trim();
       if (name.length === 0) {
              return;
       }

       // Now set the state to empty array
       ____________________________________
...
}
```

Which of the following options should go on the line in the method above?

A. this.state = {items: []};
B. this.setState({items: []});
C. Either A or B will work
D. Neither A nor B is correct.

Explanation: Directly modifying React state does not work and will lead to very subtle bugs. Instead, you should always use `this.setState()` to make changes to the state. This way, React can detect these changes and re-render the component.

Question 21: Status Codes

For each of the following scenarios, indicate the most appropriate status code for the response

Scenario	Error Code (200, 400, 500)
The request had all of the required parameters and the response contains the information the user was looking for about New Zealand	A. 200 B. 400 C. 500

Explanation: The status code 200 means OK. This status code is used when everything went smoothly with the request and response, as in this case.

The request is missing the country name (a required parameter), so the search cannot be completed	A. 200 B. 400 C. 500

Explanation: 4xx status codes mean there was something wrong with the request, and so it could not be fulfilled. They indicate that the problem was with the client, not a problem with the server. In this case, we use 400, which means “bad request” to indicate that the client did not provide all of the required parameters.

The server receives the request and it has the required parameters, but there is a problem with the database, so the server is unable to complete the search	A. 200 B. 400 C. 500

Explanation: 5xx status codes mean there was something wrong with the server. These errors indicate that the request appeared valid, but there was something on the server side that prevented the request from being fulfilled.