CSE 331
Software Design & Implementation

Spring 2023

Section 1 – HW1, Correctness, and Testing

Administrivia

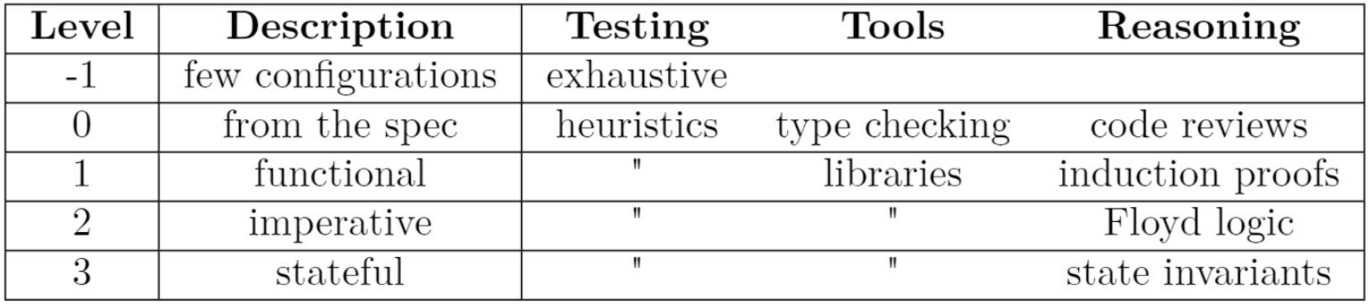
- HW1 released today, due next Wednesday (4/5)
- No more than **one** late day per assignment
- 4 late days in total
- Section attendance is required
  - Please fill out the section attendance form before the end of section!
  - Talk to us if you can’t make it
- If you haven't done the software setup yet, please take a look at the email sent last night!

Welcome

- Lets all introduce ourselves:
  - Name and pronouns
  - Year
  - What other classes you are taking this quarter
  - Something fun you did over spring break

Review – Correctness

- Levels of correctness:
  - (-1): We can manually check the output of all possible cases
  - (0): Program comes directly from spec. Can no longer test all possible cases
    - Ex: factorial, unit conversion, etc.
  - (1): Code that differs from the spec.
  - (2): Implementation of spec using imperative programming language constructs (ex: local variable mutation, *for* loops)
  - (3): Code maintains and mutates heap-allocated data structures



Level	Description	Testing	Tools	Reasoning
-1	few configurations	exhaustive		
0	from the spec	heuristics	type checking	code reviews
1	functional	"	libraries	induction proofs
2	imperative	"	"	Floyd logic
3	stateful	"	"	state invariants

Question 1

(a) Consider the following mathematical function defined on the integers 1, 2, 3, and 4:

$$\begin{aligned} \textbf{func } & f(1) := 2 \\ & f(2) := 3 \\ & f(3) := 4 \\ & f(4) := 1 \end{aligned}$$

If we implement this directly in TypeScript using a `switch` statement, what level of correctness is required?

(b) Consider the following mathematical function defined on the inputs n and b, where n is 1, 2, 3, or 4 and b is true or false. It is defined in terms of the function f defined in part (a).

$$\begin{aligned} \textbf{func } & g(n, \mathsf{T}) := f(n) \\ & g(n, \mathsf{F}) := f(n) \end{aligned}$$

If we implement this in TypeScript using an `if` statement (on b), what level of correctness is required?

Question 1

(a) Consider the following mathematical function defined on the integers 1, 2, 3, and 4:

$$\begin{aligned} \textbf{func } f(1) &:= 2 \\ f(2) &:= 3 \\ f(3) &:= 4 \\ f(4) &:= 1 \end{aligned}$$

If we implement this directly in TypeScript using a `switch` statement, what level of correctness is required?

This is level -1. There are only 4 valid inputs

(b) Consider the following mathematical function defined on the inputs n and b, where n is 1, 2, 3, or 4 and b is true or false. It is defined in terms of the function f defined in part (a).

$$\begin{aligned} \textbf{func } g(n, \mathsf{T}) &:= f(n) \\ g(n, \mathsf{F}) &:= f(n) \end{aligned}$$

If we implement this in TypeScript using an `if` statement (on b), what level of correctness is required?

Question 1

(a) Consider the following mathematical function defined on the integers 1, 2, 3, and 4:

$$\begin{aligned} \textbf{func } f(1) &:= 2 \\ f(2) &:= 3 \\ f(3) &:= 4 \\ f(4) &:= 1 \end{aligned}$$

If we implement this directly in TypeScript using a `switch` statement, what level of correctness is required?

This is level -1. There are only 4 valid inputs

(b) Consider the following mathematical function defined on the inputs n and b, where n is 1, 2, 3, or 4 and b is true or false. It is defined in terms of the function f defined in part (a).

$$\begin{aligned} \textbf{func } g(n, \mathsf{T}) &:= f(n) \\ g(n, \mathsf{F}) &:= f(n) \end{aligned}$$

If we implement this in TypeScript using an `if` statement (on b), what level of correctness is required?

This is level -1. There are only 8 valid inputs

Question 1

(c) Consider the following mathematical function defined on the inputs n and x, where n is 1, 2, 3, or 4 and x is any integer. It is defined in terms of the function f defined in part (a).

$$\textbf{func } h(n, x) := f(n) + x$$

If we implement this in TypeScript using a single `return` statement, what level of correctness is required?

(d) Suppose that we implement the function h with the following TypeScript code. (It calls f, which we will assume is implemented in TypeScript with a simple `switch` statement.)

```
function h(n: 1|2|3|4, x: number): number {
  let y = f(n);
  while (x > 0) {
    y = y + 1;
    x = x - 1;
  }
  return y;
}
```

What level of correctness is required now?

Question 1

(c) Consider the following mathematical function defined on the inputs n and x, where n is 1, 2, 3, or 4 and x is any integer. It is defined in terms of the function f defined in part (a).

$$\textbf{func } h(n, x) := f(n) + x$$

If we implement this in TypeScript using a single `return` statement, what level of correctness is required?

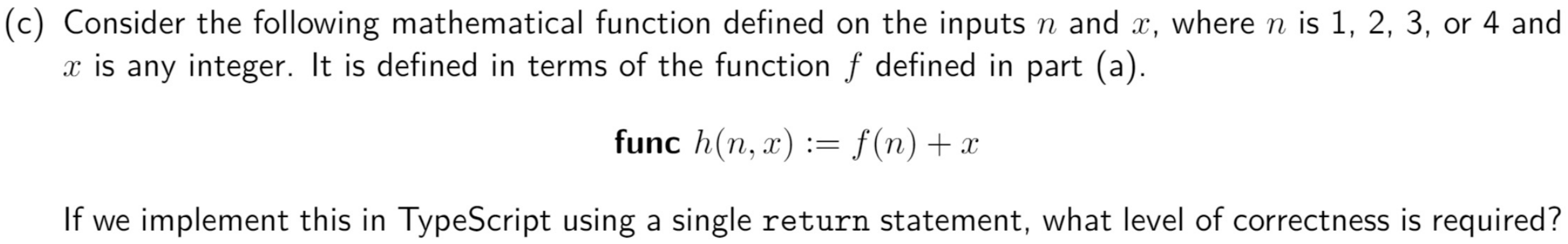
This is level 0. Infinitely many inputs but straight from the specification

(d) Suppose that we implement the function h with the following TypeScript code. (It calls f, which we will assume is implemented in TypeScript with a simple `switch` statement.)

```
function h(n: 1|2|3|4, x: number): number {
  let y = f(n);
  while (x > 0) {
    y = y + 1;
    x = x - 1;
  }
  return y;
}
```

What level of correctness is required now?

Question 1

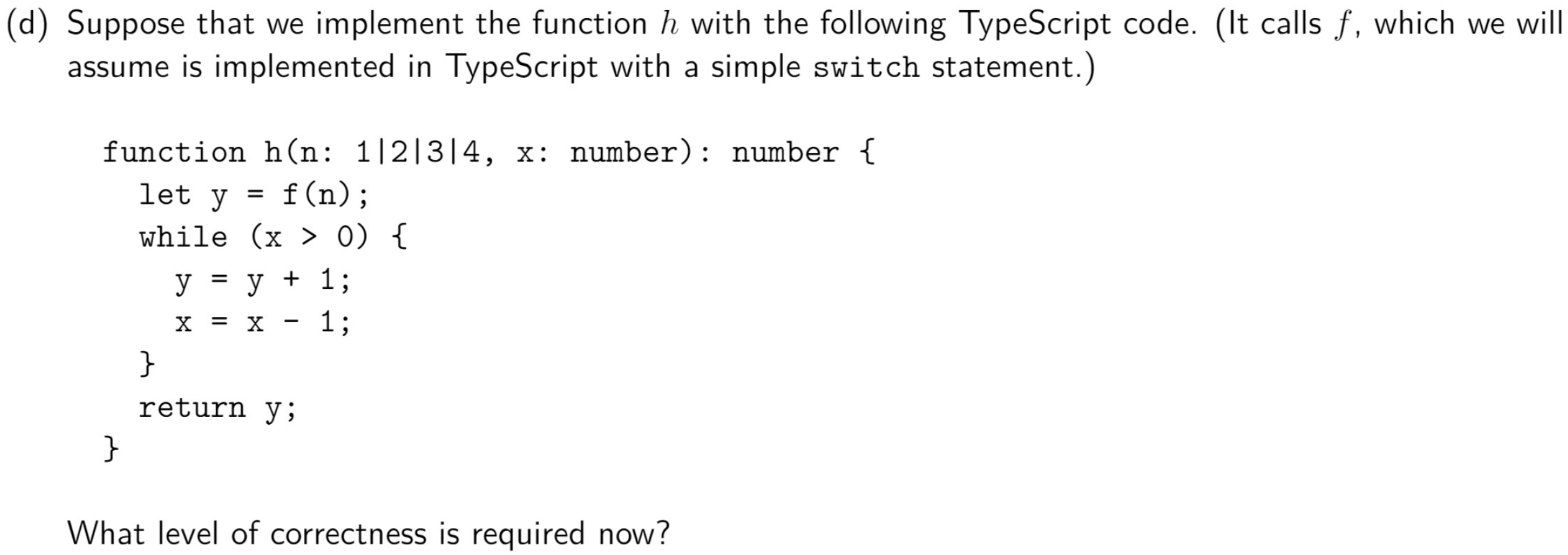


(c) Consider the following mathematical function defined on the inputs n and x, where n is 1, 2, 3, or 4 and x is any integer. It is defined in terms of the function f defined in part (a).

$$\textbf{func } h(n, x) := f(n) + x$$

If we implement this in TypeScript using a single `return` statement, what level of correctness is required?

This is level 0. Infinitely many inputs but straight from the specification



(d) Suppose that we implement the function h with the following TypeScript code. (It calls f, which we will assume is implemented in TypeScript with a simple `switch` statement.)

```
function h(n: 1|2|3|4, x: number): number {
  let y = f(n);
  while (x > 0) {
    y = y + 1;
    x = x - 1;
  }
  return y;
}
```

What level of correctness is required now?

This is level 2 since it mutates local variables (x and y are changed)

Coding Setup

Software we will use

- **Bash**: command-line shell (built-in on Mac, see course website to download Windows version)
    - Run `echo "${BASH_VERSION}”` to check for download
- **Git**: version control system (built-in on Mac, Windows version comes with Bash, above)
- **Node**: executes JavaScript code on the command-line (see link on course website to install)
    - Run `node -v` to check for download
- **NPM**: package manager (comes with Node, above)
- **VS Code** or the editor of your choice

Review – Testing

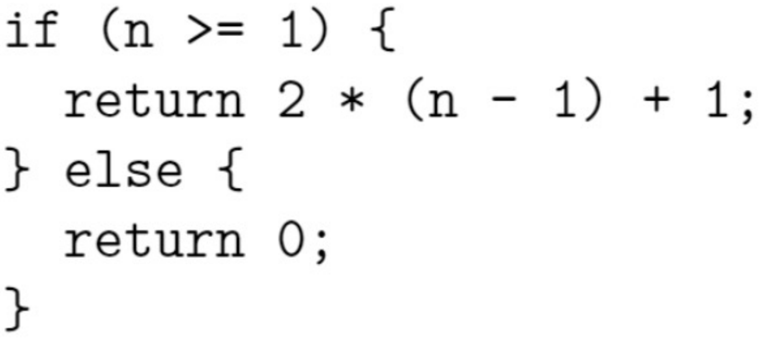
- Opaque-Box Testing
  - We look solely at the specs of the code and the description to determine test cases
  - Come up with tests before seeing the actual code
- Clear-Box Testing
  - Determines tests by looking at the actual code
  - 3 basic elements:
    - Straight-Line Calculation
    - Conditionals
    - Recursive Calls

Review – Testing

- **Straight-Line Calculation**:
  - Simplest type of code. Performs calculation without any recursive calls or *if* statements
  - Need a minimum of 2 test cases (to ensure that it is not just returning a constant)
  - Ex: `return 2 * (x - 1);`

Review – Testing

- **Conditionals**:
  - Functional code contains conditionals (if/else)
  - Code behaves differently on inputs that fall into the “if” part vs the “else” part
  - Needs test cases for each subdomain
  - Also needs to test boundary cases (where the code switches from the “if” branch to the “else” branch
  - Ex:

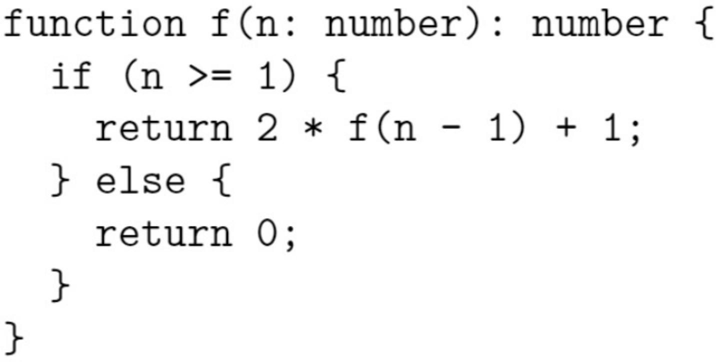


```
if (n >= 1) {
  return 2 * (n - 1) + 1;
} else {
  return 0;
}
```

In this example, we would need 4 test cases. There are 2 subdomains and each subdomain performs a straight-line calculation (which needs 2 tests each)

Review – Testing

- **Recursive Calls:**
  - Functional code contains *recursive* calls
  - Base case contains straight-line calculation
  - Needs a minimum of 2 test cases for “recursive calls”
    - One that recursively calls the base case
    - One that recursively calls the recursive call
  - Ex:



```
function f(n: number): number {
  if (n >= 1) {
    return 2 * f(n - 1) + 1;
  } else {
    return 0;
  }
}
```

In this example, we would need 3 test cases. 1 for the base case and 2 for the recursive calls

Question 3

(b) How many tests should we write for the following function? Why?

```
function quadrant(x: number, y: number): 1|2|3|4 {
  if (x >= 0) {
    return (y >= 0) ? 1 : 2;
  } else {
    return (y <= 0) ? 3 : 4;
  }
}
```

(c) How many tests should we write for the following function, defined only on the *non-negative* integers? Why? What are the tests that we should use?

```
function f(n: number): number {
  if (n === 0) {
    return 0;
  } else if (n === 1) {
    return 1;
  } else if (n % 2 === 1) {  // n is > 1 and odd
    return f(n - 2) + 1;
  } else {                   // n is > 1 and even
    return f(n - 2) + 1;
  }
}
```

Question 3

(b) How many tests should we write for the following function? Why?

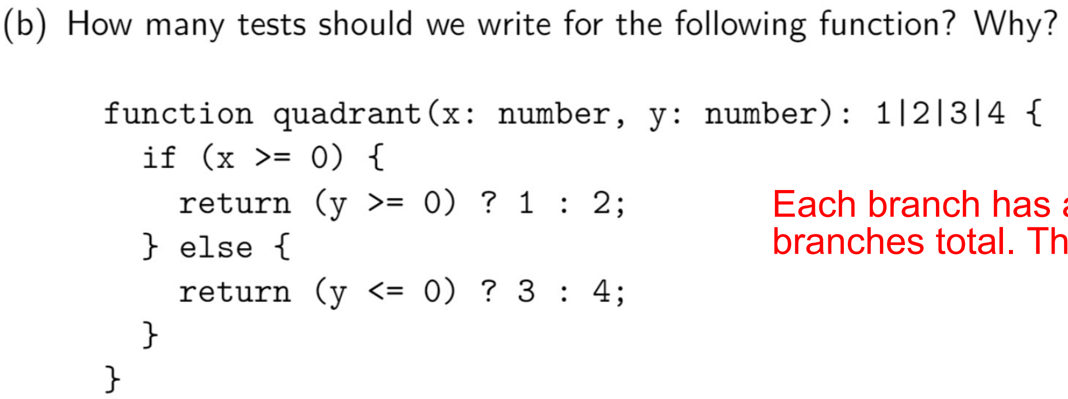
```
function quadrant(x: number, y: number): 1|2|3|4 {
  if (x >= 0) {
    return (y >= 0) ? 1 : 2;
  } else {
    return (y <= 0) ? 3 : 4;
  }
}
```

Each branch has an if statement, so 4 branches total. Thus we need 8 tests

(c) How many tests should we write for the following function, defined only on the *non-negative* integers? Why? What are the tests that we should use?

```
function f(n: number): number {
  if (n === 0) {
    return 0;
  } else if (n === 1) {
    return 1;
  } else if (n % 2 === 1) {  // n is > 1 and odd
    return f(n - 2) + 1;
  } else {                   // n is > 1 and even
    return f(n - 2) + 1;
  }
}
```

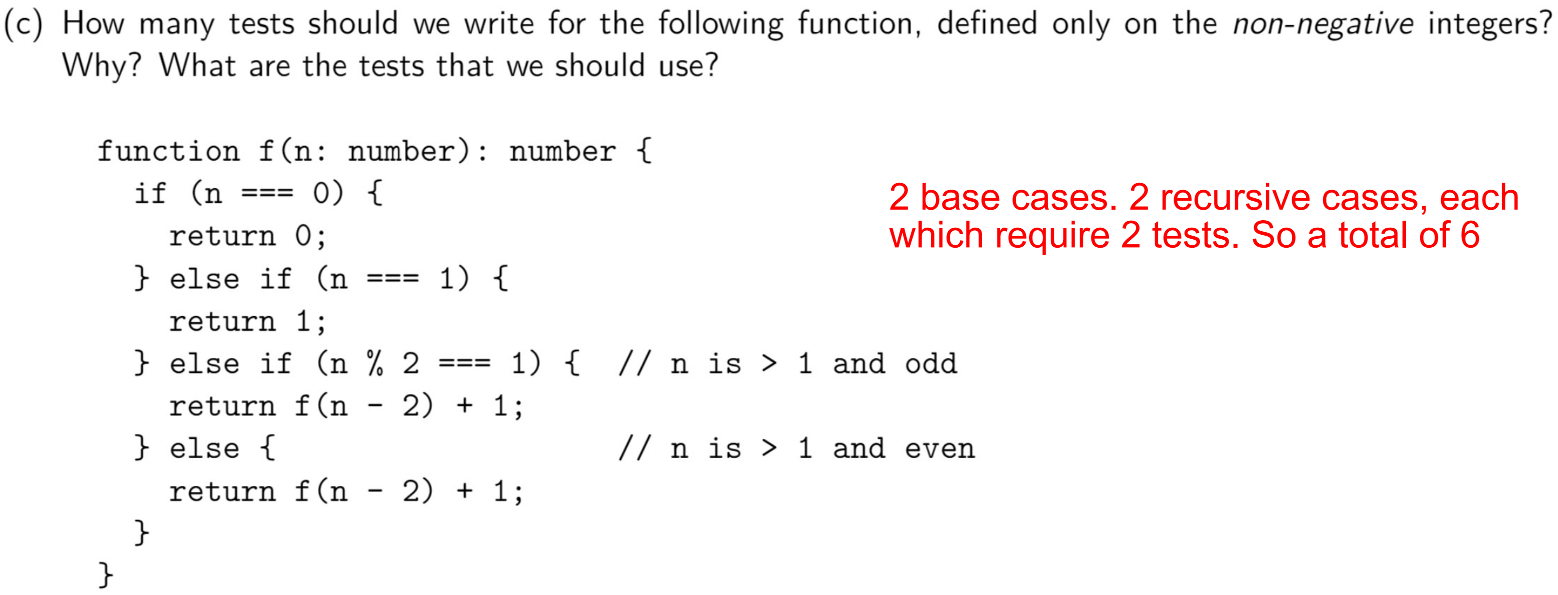
Question 3



(b) How many tests should we write for the following function? Why?

```
function quadrant(x: number, y: number): 1|2|3|4 {
  if (x >= 0) {
    return (y >= 0) ? 1 : 2;
  } else {
    return (y <= 0) ? 3 : 4;
  }
}
```

Each branch has an if statement, so 4 branches total. Thus we need 8 tests



(c) How many tests should we write for the following function, defined only on the *non-negative* integers? Why? What are the tests that we should use?

```
function f(n: number): number {
  if (n === 0) {
    return 0;
  } else if (n === 1) {
    return 1;
  } else if (n % 2 === 1) {  // n is > 1 and odd
    return f(n - 2) + 1;
  } else {                   // n is > 1 and even
    return f(n - 2) + 1;
  }
}
```

2 base cases. 2 recursive cases, each which require 2 tests. So a total of 6

Question 4

```
function f(n: number): number {
  if (n == 0) {
    return 0;
  } else {
    return A * f(n - 1) + B;
  }
}
```

(a) What are the values of $f(0)$, $f(1)$, $f(2)$, and $f(3)$ in terms of A and B?

(b) Suppose that we typed in the wrong value for `A` in the code. If we test the output of f for each input starting from 0, how far do we have to go before we could notice that `A` was wrong?

Question 4

```
function f(n: number): number {
  if (n == 0) {
    return 0;
  } else {
    return A * f(n - 1) + B;
  }
}
```

(a) What are the values of $f(0)$, $f(1)$, $f(2)$, and $f(3)$ in terms of A and B?

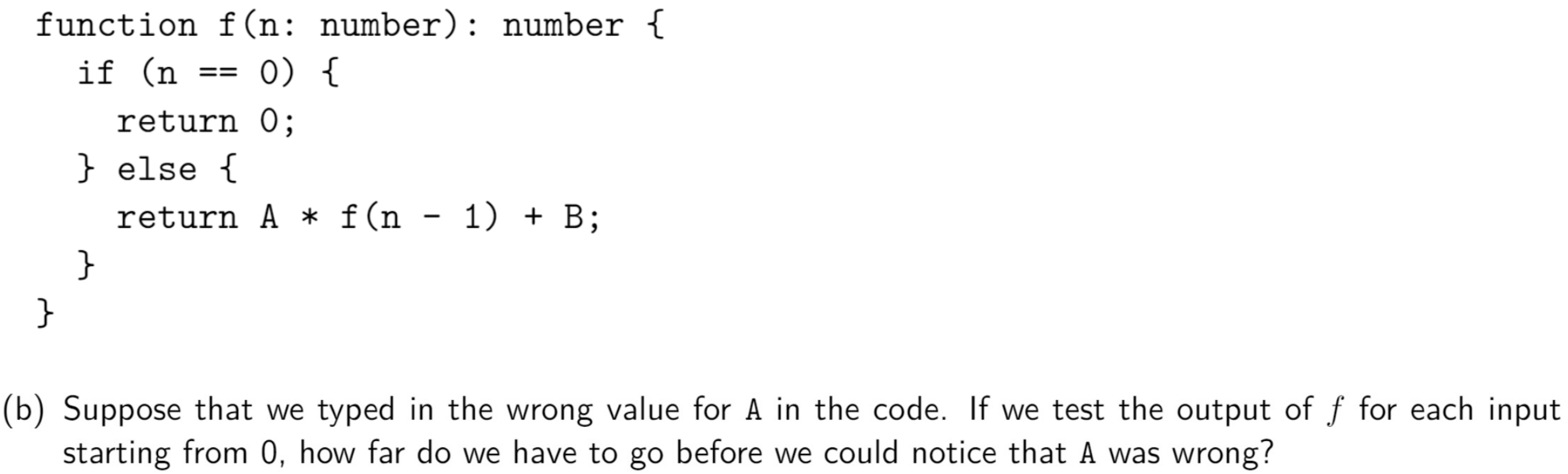
f(0) = 0

f(1) = A * f(0) + B = A * 0 + B = B

f(2) = A * f(1) + B = A * B + B = AB + B

f(3) = A * f(2) + B = A * (AB + B) = AB^2 + AB + B

Question 4



```
function f(n: number): number {
  if (n == 0) {
    return 0;
  } else {
    return A * f(n - 1) + B;
  }
}
```

(b) Suppose that we typed in the wrong value for A in the code. If we test the output of f for each input starting from 0, how far do we have to go before we could notice that A was wrong?

A does not show up in the answer until *n = 2* , so we need to test at least 0, 1, 2 before we could notice A is wrong

Attendance

Please fill out the Google Form at the following link: