CSE 331 Levels

Rohan Mukherjee

July 2, 2024

1. (a) This is correctness level 0. There are only a small number of inputs. The color type can only be one of three strings, and adding the question mark to the second input just adds one more type of undefined. So, there are $3 \times 4 = 12$ total inputs and we can just test all of them.

   (b) This one is not quite straight from the spec. Straight from the spec would be making the function look almost exactly the same, with the recursive calls, but the implementation does not have those recursive calls. However, there is also no mutation whatsover, so this would be correctness level 2.

   (c) They should use testing heuristics, testing all of their subdomains, and Floyd's logic to reason that their code is correct.

   (d) If you cannot understand the code then it cannot be straight from the spec. There is als only an English description and no formal spec. There is nothing else known about the program that ChatGPT returns. It could have any number of complicated elements, such as local variable mutation, or even heap mutation. Since it is unspecified, the prgoram it gives could be correctness level 5.

   (e) Since the function with the imperative specification is by definition straight from the spec, it would only have a correctness level of 1. However, the second function with a declarative spec, does not come straight from the spec because such an imperative spec was not given, and on top of that it edits local variables. This puts it decisively in correctness level 2, so the function with the declarative spec has a higher correctness level.

2. (a) If the function is taking a list of integers and returning a list of integers, there is no way to guarantee that the function is correct by just testing. This is because there are countably infinite-many inputs that the function could take, and we cannot test an infinite amount of inputs. We would need to formally prove that the function is correct to guarantee that it is correct.

   (b) One circumstance where we would have to change the inputs we test is if we for example changed the boundary case, by switching a $\leq$ to a $<$. We could have also decided we were just testing the wrong type of numbers, like we were testing even vs odd but we instead wanted to test $1 \mod 4$ vs $3 \mod 4$ or something like that. Perhaps we realized that we need to work with fractions instead of just integers. Then we would need to also test some fractions instead of just bigints. If we decided that one of the function inputs could be made optional, by adding a question mark to one of them, then we would also have to test that leaving out that input would still have the function return the right value.

   (c) This is basically the dual of the last case. If we instead just made it so that the arithmetic line calculates something different, with the exact same cases (basically, not changing the if statement at all), then we would be fine testing the same inputs. Similarly, if we casted the final return value to a number instead of a bigint and then returned, we would be fine testing the same cases because we haven't changed the subdomains, just what the subdomains calculate. Another thing like renaming a local variable would not cause us to test different inputs.

   (d) We can always assume that the other function has already been tested and is correct. Under this assupmtion, we haven't changed any of the subdomains, since they are all going to be the same but instead this time they are returning something else. So in this case we would not have to change our test inputs.

   (e) In part (b), if we changed an arithmetic expression to calculate something different, such as adding 1 instead of adding 2 or something like that, then we would definitely need to change the outputs as well since we would be returning something different. The same logic applies to the first case of part (c) that I talked about, just changing one arithmetic line Finally, if we changed a call from one function to another, chances are those functions are going to be used in different ways and return something different, so in this case the outputs would change as well. So it could possibly change in all 3 parts: (b), (c), (d).

3. (a) The function says that we should just be multiplying the first input by 200 and leaving the second input unchanged. A potential bug that we could have is negating the second variable on accident: such as returning $(200n, !b)$ instead of the actual output. We could also hardcode it to always return true or false as well.

   (b) This feels like more an exercise in finding patterns in numbers, but we could instead return $m^2 - n$ always. A short calculation shows that for the inputs $(n, m) = (0, -1)$ and $(-4, 3)$, $m^2 - n = n^2 - m$, so the test cases would not have caught this bug.

   (c) This type of bug could happen if the if statement had $x < -1n$ instead of $x < 0n$. In this case, the function becomes:

$$f(x) = \begin{cases} -x, & x \leq -2 \\ x, & x \geq -1 \end{cases}$$

   Plotting this graph shows that the only time that this does not return the absolute value is for $x = -1$. The testing heuristic that would have caught this bug is the one that checks boundary cases: since the first loop has -1 has it's boundary value, we would've checked that and noticed it was wrong.

   (d) The bug this time is that these cases are not exhaustive and exclusive. In this case they are not exhaustive: the input $m = 0$ does not fall into any of the conditionals. The type checker can easily catch a bug like this. A bug in this code that the type checker could not catch is noticing that we are not actually returning $|m| + 5m$, instead we are returning $|m| + 5$. This is where the testing would be helpful.

   (e) An off-by-one error for all odd inputs could happen if we for example made the final return statement return count_pairs$(n - 1n) + 1$; instead of what it actually is. If an integer n is even then so is $n - 2k$ for every $k \in \mathbb{Z}$, so if the function is passed an even number then all recursive calls will also have even inputs (since if n is even then it calls the second return statement, which is the function on $n - 2$ and then adds one). So, if n is even then we do not ever even look at the final return statement: so changing the final return statement in any way does not affect the even inputs. However, let $n = 2k + 1$ be odd and suppose that we have called the function on this input n. Then we will go into the last return statement, with our change, we will call count_pairs$(2k)$ and add 1 at the end. In particular, we are only ever adding 1 one single time, since then we call the function on an even input and as discussed above we will never see the final return statement again. This will yield an off-by-one error on all odd inputs. This could've happened if we copy-pasted the line!

(f) If the base case is wrong, then all inputs will be wrong as well. This is because the function is recursive: relying on the fact that the function works for smaller inputs, and applies a simple logic to calculate the number of pairs: if $n = 2k$ is even, then we can pair $2k$ with $2k - 1$ and then we just have to find the number of pairs on $n - 2 = 2(k - 1)$ instead. If the base case is wrong, then the final number returned will go up/down by whatever the base case actually is. Thus, testing the base case is important because for recursive inputs, the entire function depends on it being correct.

4. (a) We define $u : \mathbb{B} \times \mathbb{N} \to \mathbb{N}$ as follows:

$$\begin{array}{lllll} \textbf{func} & u((b,0)) & := 1 & & \text{for any } b : \mathbb{B} \\ & u((b,1)) & := 2 & & \text{for any } b : \mathbb{B} \\ & u((b,i+2)) & := 3 + u(b,i+1) & & \text{for any } b : \mathbb{B},\ i : \mathbb{N} \end{array}$$

(b) We define $v : \mathbb{Z} \times \mathbb{B} \times \mathbb{Z} \to \mathbb{Z} \times \mathbb{Z}$ as follows:

$$\begin{array}{lllll} \textbf{func} & v((i,(T,j))) & := (j,i) & \text{for any } i : \mathbb{Z},\ j : \mathbb{Z} \\ & v((i,(F,j))) & := (i,j) & \text{for any } i : \mathbb{Z},\ j : \mathbb{Z} \end{array}$$

(c) We define $w : \{s : \mathbb{B} \times \mathbb{R},\ b : \mathbb{B}\} \to \mathbb{R}$ as follows:

$$\begin{array}{lllll} \textbf{func} & w(\{s : (T,n),\ b : T\}) & := n & \text{for any } n \in \mathbb{R} \\ \textbf{func} & w(\{s : (F,n),\ b : F\}) & := n & \text{for any } n \in \mathbb{R} \\ \textbf{func} & w(\{s : (T,n),\ b : F\}) & := -n & \text{for any } n \in \mathbb{R} \\ \textbf{func} & w(\{s : (F,n),\ b : T\}) & := -n & \text{for any } n \in \mathbb{R} \end{array}$$

5. (a) Notice that the type color can be one of 3 strings. Also, since there is a question mark next to the input c, c can also take on the value undefined. Thus c can take on 4 total inputs: the 3 colors that Color can be and undefined. This is a small number of inputs, so we can just test all 4 inputs with 4 test cases. A specific test set is ”red”, ”yellow”, ”blue” and undefined.

   (b) Now this function has infinitely many inputs: the 3 colors and the infinitely many integers. Thinking of a switch statement as just if/else blocks, all 3 cases have infinitely many inputs, since there are infinitely many integers. Thus, per the testing guidelines, we would need 2 test cases for each case. Since there are 3 colors, this comes out to 6 total cases. We could use the tests: (”red”, 1), (”red”, -3), (”blue”, 2), (”blue”, -6), (”yellow”, 3), (”yellow”, -5).

   (c) We can assume that the f2 function has already been tested. The color option can only be one of three strings, thus there is a small number of inputs to this function, namely, $3 \cdot 3 = 9$, so we can simply test all of the inputs. The test set would be the pairs in $\{\text{”red”}, \text{”yellow”}, \text{”blue”}\}^2$.

   (d) This is a recursive function. The first if statement has only one input, and it is a base case, so we would test that with the empty (length 0) list. Then for the recursive part, we would want to use the 1-many part of the rule. Notice that we simply return the first value of the list plus a recursive call to the list but with the first element removed (that is what the slice is doing). So if the list has length n then there will be n recursive calls. The subdomain consisting of length 1 lists is infinitely large, so we would want to test at least 2 lists for this subdomain, such as [1] and [45]. Finally, for the many case, we could also test two lists such as $[3, 19, 45]$ and $[6, 22]$, since the subdomain consisting of more than one recursive call is infinitely large we need at least 2 cases. This comes to $1 + 2 + 2 = 5$ total test cases.

   (e) For the first if statement, since it has infinitely many inputs (all non-positive numbers), we would need two test cases for it, such as 0 (the boundary case), and -5. Then since we are recursing, we need to use the 1-many rule. Since the base case is when $n \leq 0$, and we are recursing on $n - 1$, we see that there is one recursive call iff $n = 1$. So we test that single case. Finally, we would want to test the many case, and since this subdomain is infinitely large, we would want at least two test cases for this. So, we could perhaps test 17 and 42, for a total of $2 + 1 + 2 = 5$ test cases.

   (f) First, we test the base case of the recursive function, which has only one input, namely 0. Then we have three if/else blocks that we need to test, based on if n is $0, 1$ or 2 mod 3. Since there is a recursive call in each of the if/else blocks, we have to use

the 1-many rule again. The first one is simple: the only single recursive call is just $n = 3$, and similarly for the second the only single recursive call is $n = 1$ and for the last its $n = 2$. Now, for the first if statement, notice that we always recurse on $n - 3$: which is going to be another multiple of 3. Thus once we have called the function on a multiple of 3 it will always call itself on smaller multiples of 3. The other thing to notice is that the other two if statements make one recursive call on a multiple of three: so every path to the base case starts with a possibly non-multiple of three followed by only multiples of three. So there are three subdomains: multiples of three, 1 mod 3 and 2 mod 3. For the many case, each of these are infinitely large. For the first one, we could use for example 6 and 12. For the second, we could use 4 and 7, and for the last we could test 5 and 8. We need two tests per each subdomain since each of their input spaces are infinitely large (have more than 2 inputs). This gives a total of $2 + 2 + 2 + 1 + 1 + 1 + 1 = 10$ test cases.

6. (a)

$$\begin{array}{lll} \textbf{func} & \text{fact}(0) & := 1 \\ & \text{fact}(n+1) & := (n+1) \cdot \text{fact}(n) \quad \text{for any } n : \mathbb{N} \end{array}$$

(b) The recursive calls would go like this:

$$\begin{array}{c} \text{fact}(4) = 4 \cdot \text{fact}(3) \\ \downarrow \\ 4 \cdot \text{fact}(3) = 4 \cdot 3 \cdot \text{fact}(2) \\ \downarrow \\ 4 \cdot 3 \cdot \text{fact}(2) = 4 \cdot 3 \cdot 2 \cdot \text{fact}(1) \\ \downarrow \\ 4 \cdot 3 \cdot 2 \cdot \text{fact}(1) = 4 \cdot 3 \cdot 2 \cdot 2 \cdot 1 \cdot \text{fact}(0) \\ \downarrow \\ 4 \cdot 3 \cdot 2 \cdot 2 \cdot 1 \cdot \text{fact}(0) = 4 \cdot 3 \cdot 2 \cdot 1 \cdot 1 = 24 \end{array}$$

Here we use the notation that the left hand side is the same as the right hand side of the previous step, and the right hand side is what you get once you use the definition of fact. For example, $4 \cdot 3 \cdot \text{fact}(2) = 4 \cdot 3 \cdot 2 \cdot \text{fact}(1)$, since $\text{fact}(2) = 2 \cdot \text{fact}(1)$.

7. (a) First, notice that if $f \in \mathbb{Z}[x]$ is a polynomial with integer coefficients, then by definition $f(n)$ is a polynomial in n. If $g \in \mathbb{Z}[x]$ is a polynomial in x, I claim that $g(f(n))$ is also a polynomial in n. This is because, writing $g(x) = \sum_{i=0}^{m} g_i x^i$ and $f(x) = \sum_{j=0}^{l} f_j x^j$, we have that:

$$g(f(n)) = \sum_{i=0}^{m} g_i \left(\sum_{j=0}^{l} f_j x^j \right)^i$$

Since powers of polynomials are polynomials, and linear combinations of polynomials are polynomials, we have that $g(f(n))$ is a polynomial in n. Repeating this argument shows that the return value of the function is a polynomial in n.

(b) Given $k+1$ points with distinct x coordinates $(x_1, y_1), \ldots, (x_{k+1}, y_{k+1})$ there exists a unique polynomial $f(x)$ satisfying $f(x_i) = y_i$. This follows by a very nice argument involving the Vandermonde matrix and using the factor theorem: that a degree k polynomial has at most k roots. So, to ensure that our function is giving us the unique, correct value, we would need greater than or equal to $k+1$ test cases with different inputs.

(c) By the above argument, if 2 can detect any bug, then we must have $k + 1 \leq 2$. This is the same as $k \leq 1$, so we would need the function to be linear or constant in n.