CSE 331: Software Design & Implementation Fall 2025

Homework 1

Due: Friday, October 3rd, 6pm

Task 1 – Every Log Has Its Day [8 pts]

We plan to provide the following method:

```
/** Calculates the integer, base-2 logarithm of n.
 * ...
 * @return the integer k such that 2^k <= n < 2^(k+1)
 */
public static int log2(int n);
```

a) When n is a power of 2, the integer k from the spec will satisfy $2^k = n$. Taking logarithms on both sides, we see that $k = \log_2(n)$. But how does k relate to $\log_2(n)$ *when n is not a power of 2*?

b) Keeping in mind your answer to (a), what is an another, equally reasonable but incomparable, way to specify this function? Your answer should leave the precondition unchanged and not change the postcondition in any case where n is a power of 2.

How would you write the `@return` statement for this new specification?

c) This specification precisely defines the return value for valid inputs, but it does not make sense when `n` not positive. Give **two** distinct ways of turning this into a specification that fully defines the all integer inputs, now including non-positive values.

Your two specifications must still return the value described in the original specification when `n` is positive but they should be incomparable specifications overall.

d) Give a **third** distinct specification that is weaker than both of your specifications from part (c).

Again, it must still return the value described in the original specification when `n` is positive.

Task 2 – Extra Credit: The Log Ate My Homework [8 pts]

The original specification is *declarative*. Write an imperative specification for positive n using our mathematical notation (inside the Javadoc `@return`). Do not worry about efficiency.

Assume that integer addition, multiplication, and comparison operators are already defined.

Hint: Define a function with two parameters, where the second parameter is used to count upward until we find the correct value of k. You can then specify the return value, in the `@return`, as a call to your function with appropriate choices for its two parameter values. For this problem, you can assume that our math notation includes exponentiation when the base is 2 (e.g., 2^n).

Task 3 – Optional: A-I, Captain [10 pts]

In this problem, we will use AI to implement the `log2` method from Task 1 with different specifications.

a) Which AI are you using?

b) Prompt the AI to fill in the body of `log2` for your incomparable specifications from Task 1(c). Show the code that it produces.

c) Now, prompt it using your weaker specification from Task 1(d). Which code does it give you? Is that guaranteed to be the one that you wanted?

d) Change back to one of your specifications from Task 1(c), but now, change the postcondition for positive values of `n` to your alternative, incomparable `@return` from Task 1(b).

Prompt the AI to fill in the body. Show the code that it produces.

e) Now, change the `@return` to this less formal, English specification:

```
@return the base-2 logarithm of n
```

Prompt the AI to fill in this method body. Which code does it give you? Is that guaranteed to be the one that you wanted?

Task 4 – The Test-Laid Plans [10 pts]

For each of the following functions, state the number of tests required to meet our coverage requirements and explain why that is the required number.

Then, describe a specific set of tests to use (with the same number of tests you as said before). Describe each test by giving the input (identify a specific input rather than saying, e.g., "some positive number"), saying what portion of the function it tests, and explaining why our rules require that test.

a) The function j : (List) $\to$ List defined by

$$\begin{aligned} j(\mathsf{nil}) &= \mathsf{nil} \\ j(a :: \mathsf{nil}) &= a :: \mathsf{nil} \\ j(a :: b :: L) &= a :: j(L) \end{aligned}$$

b)
```
/** @requires n >= 1 */
public static int l(int n) {
  int k = 0;
  while (n > 1) {
    n = n / 2;
    k++;
  }
  return k;
}
```

Task 5 – Under Formal Circumstances [12 pts]

In this problem, we will formalize the following English description: "the function `mult` multiplies together the values in a given list".

a) Calculate the value of mult on $2 :: 3 :: 4 ::$ nil and each shorter list: $3 :: 4 ::$ nil, and $4 ::$ nil.

b) Looking at these examples, can we calculate mult$(x :: L)$ in terms of x and mult(L). Show how this works on $3 :: 4 ::$ nil and $2 :: 3 :: 4 ::$ nil.

c) It is not clear from the English that mult(nil) makes any sense. It would be reasonable to disallow this case by requiring that the input is not an empty list.

Suppose that we want to support empty lists. How do we define mult(nil) so that the recursive pattern you identified in part (b) applies to lists containing one element also?

d) Write a formal definition of mult, both its type and definition, using our mathematical notation.

e) State the number of tests required to meet our coverage requirements for your function mult and explain why that is the required number. Then, describe a specific set of tests to use (with the same number of tests you as said before). Describe each test by giving the input (identify a specific input rather than saying, e.g., "some positive number"), saying what portion of the function it tests, and explaining why our rules require that test.