Problem 1a

**Summary**

The given loop invariants are correct. Though we can be more accurate in the description of the state of the array during the sorting process. Clarifying them will provide a better understanding of the algorithm's progress and ensures they align more precisely with the actual operations and goals of the Bubble Sort algorithm.

**Outer Loop Improved Invariant**

After each iteration of the outer loop, the subarray A[bound,..,n-1] is sorted in non-decreasing order and contains elements that are greater than or equal to all elements in A[0,..,bound-1]. Each pass of the bubble sort ensures the next largest element is placed at its correct position, reducing the unsorted portion of the array by one.

**Inner Loop Improved Invariant**

The variable t marks the position of the last swap in the array A[0,..,bound-1] during each iteration of the inner loop. This implies that all elements after the last swap position t (up to bound-1) are in correct order relative to each other and do not require further swaps in this pass.

Problem 1b

1. Weakest Precondition: True
2. Weakest Precondition:
3. Weakest Precondition:
4. Weakest Precondition:
5. Weakest Precondition:
6. Weakest Precondition:
7. Weakest Precondition:
8. Weakest Precondition:
9. Weakest Precondition:
10. Weakest Precondition:
11. Weakest Precondition:
12. Weakest Precondition: must be the largest element of the subarray for the update to make sense.

Additional context …

1. bound = n;
   1. Weakest Precondition: n >= 0 and array A is defined with n elements.
   2. This is necessary for the array to be properly initialized for sorting.
2. while (bound > 0) {
   1. Weakest Precondition: The array A[0,…,n-1] exists , and n >= 0.
   2. This loop iterates until the array is sorted, so the initial condition is simply that the array exists, and n is defined.
3. t = 0;
   1. Weakest Precondition: bound > 0
   2. Ensures that there's at least one element to consider for sorting.
4. i = 0;
   1. Weakest Precondition: bound > 0
   2. Indicates the start of a new iteration over the array elements up to *bound*.
5. while (i < bound-1) {
   1. Weakest Precondition: i < bound and bound > 1
   2. Ensures there are at least two elements to compare and possibly swap.
6. if (A[i] > A[i+1])
   1. Weakest Precondition: A[i] and A[i+1] exist (i.e., 0 ≤ i < n-1).
   2. Checks if two adjacent elements are out of order and need swapping.
7. swap = A[i];
8. i] = A[i+1];
9. A[i+1] = swap;
   1. Weakest Precondition for 7-9: A[i] > A[i+1]
   2. Ensures that a swap is needed to correct the order of two adjacent elements.
10. t = i+1;
    1. Weakest Precondition: A swap has occurred, indicating the new boundary for the next iteration.
11. i++;
    1. Weakest Precondition: i < bound - 1.
    2. Ensures the loop continues to iterate over the array elements within the current boundary.
12. bound = t;
    1. Weakest Precondition: The loop has completed an iteration, and t marks the new boundary up to which the array may not be sorted.

Problem 1c

**Precondition for the Entire Code**

n >= 0 and A contains n elements indexed from 0.

**Statement 1**

bound = n;

**Weakest Precondition for Statement 1**

The precondition for the entire code directly leads to the weakest precondition for statement 1. Since the code's precondition specifies that n >= 0 and A contains n elements, this inherently supports the action of setting bound to n. Therefore, the Weakest Precondition for statement 1, ensuring bound = n; is a valid operation, is logically inferred from the precondition of the code because it relies on the existence and size of the array A, as well as n being a non-negative integer indicating the number of elements in A.

**Conclusion**

The weakest precondition for statement 1 is logically inferred from the stated precondition of the code, which is and contains elements indexed from

Problem 2

<assign> ::= <var> = <expression>

Updates the state by evaluating the right-hand side expression and assigning its value to the variable on the left-hand side.

Mstate(<var> = <expression>, state) = Add(Mname(<var>), Minteger(<expression>, state), state)

<if> ::= if <condition> then <statement> else <statement2>

Branches execution based on the evaluation of a condition, leading to different state changes depending on whether the **then** or **else** statement is executed.

Mstate(if <condition> then <statement1> else <statement2>, state) =

if Mboolean(<condition>, state) then Mstate(<statement1>, state)

else Mstate(<statement2>, state)

<while> ::= while <condition> <statement>

Repeatedly updates the state by executing a statement as long as a condition remains true.

Mstate(while <condition> <statement>, state) =

if Mboolean(<condition>, state) then Mstate(while <condition> <statement>, Mstate(<statement>, state))

else state

Summary

Denotational semantics is a mathematical approach to defining programming language syntax and its effects on program state, utilizing mappings such as **Minteger** and **Mboolean** for evaluating expressions and conditions, and **Mname** for variable identification. It employs state manipulation functions like **Add**, which updates or adds variable values, and **Remove**, which deletes variable associations, albeit less commonly due to implicit variable management in many languages. This framework provides a robust method for handling normal and error conditions, offering a precise, mathematical understanding of program behavior. While denotational semantics is broadly applicable, its effectiveness and challenges vary across different programming paradigms, emphasizing its foundational role in the theoretical underpinnings of computer science.