3.3 Bubble Sort Program

3.3.1 Problem Definition

1. We define the problem as a function BubbleSort: array<Z> → array<Z>, where array<Z> is the ordered and finite set of integers.
2. Input Space as well as Output Space is array<Z>.

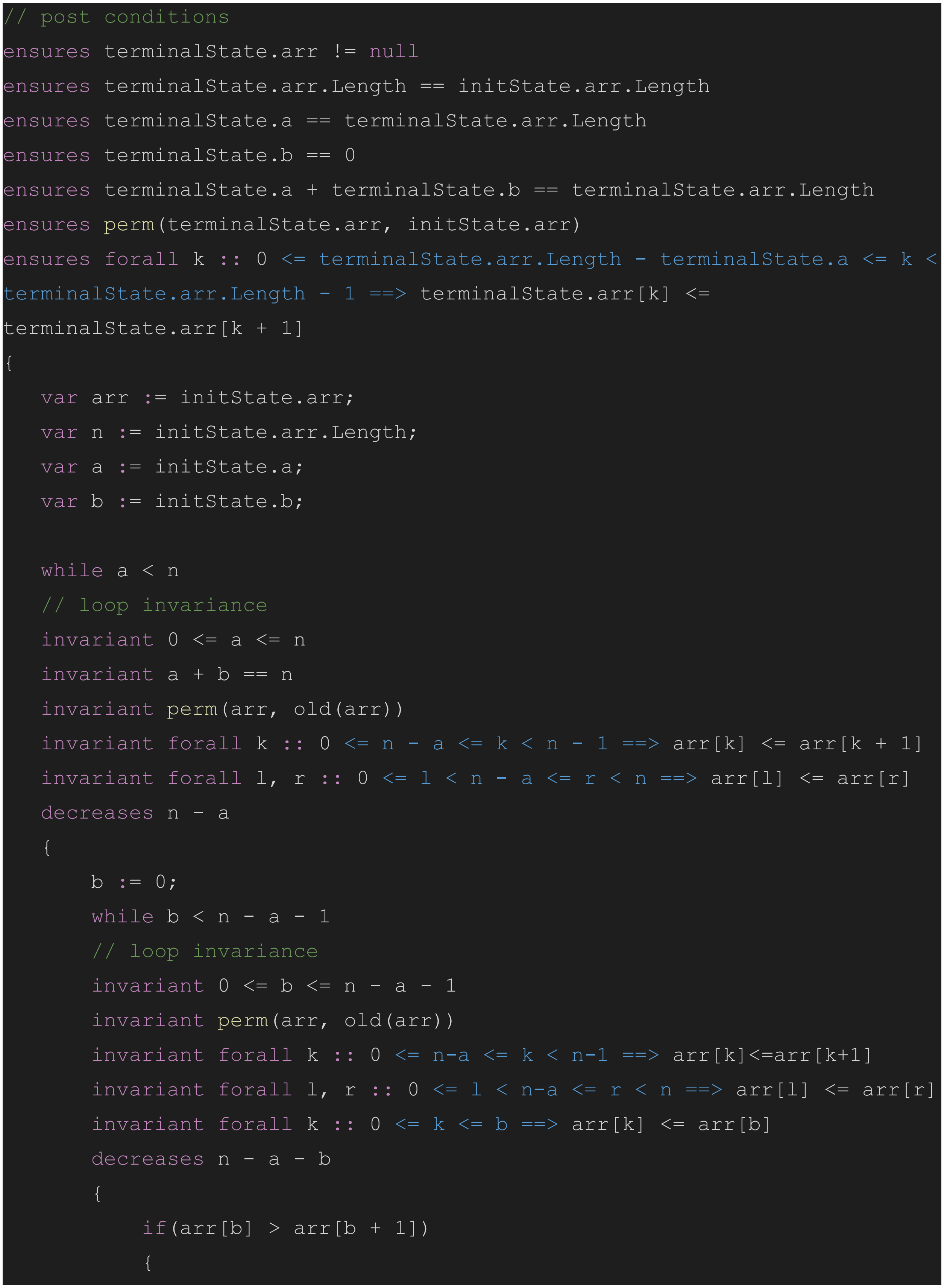
3.3.2 Transition System Definition

1. S_{sort} = < X, X^0, U, →, Y, h>
2. The state space of the system X = array<Z> x Z x Z.
3. We define a function ρ: array<Z> → X, which converts the input space of the problem to the state space of the system.
4. ρ(arr) = (arr, 0, arr.Length), such that arr ∈ array<Z> is the case for the initial state. Hence X^0 = ρ(arr) = (arr, 0, arr.Length), where arr.Length is the number of integers in the arr.
5. U = {next}
6. Transition Relation (arr, a, b) $^{next}\rightarrow$ (arr, a + 1, 0) if a + b == n else (arr', a, b + 1) where if arr[b] > arr[b + 1] then arr'[b] = arr[b + 1] and arr'[b + 1] == arr[b], such that a, b ∈ Z ∧ arr, arr' ∈ array<Z> ∧ a, b >= 0.
7. Let X_f be the final state of the system, defined as X_f = (arr, a_f, b_f) iff a_f = arr.Length and b_f == 0.
8. Y = array<Z>, as the view space of the system is equal to the output space of the problem.
9. h: X → Y, where h: X → array<Z>
10. h(x) = x[0], where x ∈ X and x[0] is the 1^{st} element from the 3 tuple state vector.

3.3.3 Program

```
// Input Space
datatype InputSpace = InputSpace(arr: array?<int>)

// State Space
datatype StateSpace = StateSpace(arr: array?<int>, a: int, b: int)

// rho function
function method rho(tup:InputSpace): StateSpace
requires tup.arr != null
{
   StateSpace(tup.arr, 0, tup.arr.Length)
}
```

```
// view function h
function method pi(trip:StateSpace): array?<int>
{
   (trip.arr)
}

function count(k: int, a: array?<int>, i: int): int
requires a != null
requires 0 <= i <= a.Length
reads a
decreases a.Length - i
{
   if i == a.Length then 0
   else if k == a[i] then 1 + count(k, a, i + 1)
   else count(k, a, i + 1)
}

predicate perm(a: array?<int>, b: array?<int>)
requires a != null
requires b != null
reads a
reads b
requires a.Length == b.Length
{
   forall k :: count(k, a, 0) == count(k, b, 0)
}

// Transition System
method TransitionSystem(initState: StateSpace) returns
(terminalState:StateSpace)
// pre conditions
requires initState.arr != null
modifies initState.arr
requires initState.a == 0
requires initState.b == initState.arr.Length
requires initState.a + initState.b == initState.arr.Length
```



```
// post conditions
ensures terminalState.arr != null
ensures terminalState.arr.Length == initState.arr.Length
ensures terminalState.a == terminalState.arr.Length
ensures terminalState.b == 0
ensures terminalState.a + terminalState.b == terminalState.arr.Length
ensures perm(terminalState.arr, initState.arr)
ensures forall k :: 0 <= terminalState.arr.Length - terminalState.a <= k <
terminalState.arr.Length - 1 ==> terminalState.arr[k] <=
terminalState.arr[k + 1]
{
    var arr := initState.arr;
    var n := initState.arr.Length;
    var a := initState.a;
    var b := initState.b;

    while a < n
    // loop invariance
    invariant 0 <= a <= n
    invariant a + b == n
    invariant perm(arr, old(arr))
    invariant forall k :: 0 <= n - a <= k < n - 1 ==> arr[k] <= arr[k + 1]
    invariant forall l, r :: 0 <= l < n - a <= r < n ==> arr[l] <= arr[r]
    decreases n - a
    {
        b := 0;
        while b < n - a - 1
        // loop invariance
        invariant 0 <= b <= n - a - 1
        invariant perm(arr, old(arr))
        invariant forall k :: 0 <= n-a <= k < n-1 ==> arr[k]<=arr[k+1]
        invariant forall l, r :: 0 <= l < n-a <= r < n ==> arr[l] <= arr[r]
        invariant forall k :: 0 <= k <= b ==> arr[k] <= arr[b]
        decreases n - a - b
        {
            if(arr[b] > arr[b + 1])
            {
                var tmp := arr[b + 1];
                arr[b + 1] := arr[b];
                arr[b] := tmp;
            }
            b := b + 1;
        }
        a := a + 1;
    }
    terminalState := StateSpace(arr, a, b);
}

method Main()
{
    var n: int := 4;
    var arr := new int[n];
    arr[0] := 2;
    arr[1] := 3;
    arr[2] := 4;
    arr[3] := 1;

    var input := InputSpace(arr);
    var initState := rho(input);
    var terminalState := TransitionSystem(initState);
    var output := pi(terminalState);

    n := output.Length;
    assert perm(output, arr);
    assert forall k :: 0 <= k < n - 1 ==> output[k] <= output[k + 1];
}
```

3.3.4 Pre Condition

1. requires input arr not to be null
   requires initState.arr != null.
2. requires input a equal to 0
   requires initState.a == 0.
3. requires input b to be equal to arr's length
   requires initState.b == initState.arr.Length.
4. requires input a + b to be equal to arr's length

requires initState.a + initState.b == initState.arr.Length.

3.3.5 Post Condition

1. ensures that the output arr is not null
   ensures terminalState.arr != null.
2. ensures that the output arr's length equals to the input arr's length
   ensures terminalState.arr.Length == initState.arr.Length.
3. ensures that the output a equals arr's length which ensures successful termination
   ensures terminalState.a == terminalState.arr.Length.
4. ensures that the output b equals 0 which ensures successful termination
   ensures terminalState.b == 0.
5. ensures that the output a + b equals arr's length which ensures successful termination
   ensures terminalState.a + terminalState.b == terminalState.arr.Length.
6. ensures that the output arr has same set of elements as input arr so the sorted array is the required array
   ensures perm(terminalState.arr, initState.arr).
7. ensures that the arr is sorted
   ensures forall k :: 0 <= terminalState.arr.Length - terminalState.a <= k < terminalState.arr.Length - 1 ==> terminalState.arr[k] <= terminalState.arr[k + 1].