3.1 Recursive Fibonacci Program

3.1.1 Problem Definition

1. We define the problem as a function Fibo: Z → Z
2. Input Space as well as Output Space is Z

3.1.2 Transition System Definition

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

3.1.3 Program

```
// Input Space
datatype InputSpace = InputSpace(n: int)

// State Space
datatype StateSpace = StateSpace(n: int, a: int, b: int, c: int)

// rho function
function method rho(tup:InputSpace): StateSpace
{
   StateSpace(tup.n, 0, 0, 1)
}

// view function h
function method pi(trip:StateSpace): int
{
   (trip.b)
}
```

```
// test function
function fibo(n: int): int
requires n >= 0
decreases n
{
   if n == 0 then 0
   else if n == 1 then 1
   else fibo(n - 1) + fibo(n - 2)
}

// Transition System
method TransitionSystem(initState: StateSpace) returns
(terminalState:StateSpace)
// pre conditions
requires initState.n >= 0
requires initState.a == 0
requires initState.b == 0
requires initState.c == 1
// post conditions
ensures terminalState.n == initState.n
ensures terminalState.a == terminalState.n
ensures terminalState.b == fibo(terminalState.n)
ensures terminalState.c == fibo(terminalState.n + 1)
{
   var n := initState.n;
   var a := initState.a;
   var b := initState.b;
   var c := initState.c;

   while a < n
       // loop invariance
       invariant 0 <= a <= n
       invariant b == fibo(a)
       invariant c == fibo(a + 1)
       decreases n - a
   {
       var cur := c;
```

```
        c := b + c;
        b := cur;
        a := a + 1;
    }
    terminalState := StateSpace(n, a, b, c);
}

method Main()
{
    var input := InputSpace(4);
    var initState := rho(input);
    var terminalState := TransitionSystem(initState);
    var output := pi(terminalState);
    assert output == 3;
}
```

3.1.4 Pre Condition

1. requires input integer to be always greater than or equal to 0
   requires initState.n >= 0
2. requires initState.a == 0, as a == $fibo^{-1}$(b).
3. requires initState.b == 0, as initState.b represents fibo(0) = fibo(a).
4. requires initState.c == 1, as initState.c represents fibo(1) = fibo(a + 1).

3.1.5 Post Condition

1. ensures that the output is for the given input.
   ensures terminalState.n == initState.n.
2. ensures that the output is the n^{th} fibonacci number
   ensures terminalState.a == terminalState.n.
3. ensures that b which is the final output is the correct n^{th} fibonacci number
   ensures terminalState.b == fibo(terminalState.n).
4. ensures that c is the correct $(n+1)^{th}$ fibonacci number
   ensures terminalState.c == fibo(terminalState.n + 1).