```
// Part A.

// Weakest Precondition Proofs
// In the following weakest precondition proofs, [x.xx] references a law from Appendix A of Programming from Specifications by Carroll Morgan.

// Weakest Precondition and Termination Proof
method ComputeFusc(N: int) returns (b: int)
    requires N >= 0
    ensures b == fusc(N)
{
    // This method is totally correct as the given precondition implies the weakest precondition and termination has been proven.
    // Through the ghost var d and the invariant n >= 0 the loop is shown to be complete (as n decreases through each iteration
    // approaching n == 0) and thus the specified program terminates.
    { N >= 0 } // By law [A.09]
    { (true && (N >= 0) }
    { (fusc(N) == fusc(N)) && (N >= 0) } // Multiplicative Identity
    { (fusc(N) == fusc(N) + 0 * fusc(N + 1)) && (N >= 0) }
    b := 0;
    { (fusc(N) == 1 * fusc(N) + b * fusc(N + 1)) && b == fusc(N) && (N >= 0) } // By law [A.74]
    { forall n, a :: (fusc(N) == 1 * fusc(N) + b * fusc(N + 1)) && b == fusc(N) && (N >= 0) }
    var n, a := N, 1;
    { (fusc(N) == a * fusc(n) + b * fusc(n + 1)) && (n >= 0) }
    while (n != 0)
        invariant fusc(N) == a * fusc(n) + b * fusc(n + 1)
        invariant n >= 0 // For termination
        decreases n
    {
        { (n >= 0) && (fusc(N) == (a * fusc(n) + b * fusc(n + 1)) } // By law [A.09]
        { ((n >= 0) && (fusc(N) == (a * fusc(n) + b * fusc(n + 1)))) && (true) } // By law [A.16]
        { (((n >= 0) && (fusc(N) == (a * fusc(n) + b * fusc(n + 1)))) && (n % 2 == 0 || n % 2 == 1) } // By law [A.07]
        { (n % 2 == 0 && ((n>= 0) && (fusc(N) == (a * fusc(n) + b * fusc(n + 1))))) || (n % 2 == 1 && ((n >= 0) && (fusc(N) == a * fusc(n) + b * fusc(n + 1))) } // Since n == 0 => n % 2 != 1 (n = 0 can be included within the range for simplification)
        { (n % 2 == 0 && ((n>= 0) && (fusc(N) == (a * fusc(n) + b * fusc(n + 1))))) || (n % 2 == 1 && ((n >= 1) && (fusc(N) == a * fusc(n) + b * fusc(n + 1))) } // By law [A.38]
        { (n % 2 == 0 => ((n >= 0) && (fusc(N) == (a * fusc(n) + b * fusc(n + 1))))) && (n % 2 == 1 => ((n >= 0) && (fusc(N) == a * fusc(n) + b * fusc(n + 1)))) } // By law [A.09]
        { (n % 2 == 0 => ((n >= 0) && true && (fusc(N) == (a * fusc(n) + b * fusc(n + 1))))) && (n % 2 == 1 => ((n >= 0) && true && (fusc(N) == a * fusc(n) + b * fusc(n + 1)))) }
        { (n % 2 == 0 => ((n >= 0) && (n > n/2) && (fusc(N) == (a * fusc(n) + b * fusc(n + 1))))) && (n % 2 == 1 => ((n >= 0) && (n > (n - 1)/2) && (fusc(N) == a * fusc(n) + b * fusc(n + 1)))) } // By law [A.74]
        { forall d :: (n % 2 == 0 => ((n >= 0) && (n > n/2) && (fusc(N) == (a * fusc(n) + b * fusc(n + 1))))) && (n % 2 == 1 => ((n >= 0) && (n > (n - 1)/2) && (fusc(N) == a * fusc(n) + b * fusc(n + 1)))) }
        ghost var d := n;
        { (n % 2 == 0 => ((d >= 0) && (d > n/2) && (fusc(N) == (a * fusc(n) + b * fusc(n + 1))))) && (n % 2 == 1 => ((d >= 0) && (d > (n - 1)/2) && (fusc(N) == a * fusc(n) + b * fusc(n + 1)))) }
        if (n % 2 == 0) {
            { (d >= 0) && (d > n/2) && (fusc(N) == (a * fusc(n) + b * fusc(n + 1)) } // By fusc properties (iv and iii) with (n' == (n/2))
            { (d >= 0) && (d > n/2) && (fusc(N) == (a * fusc(n/2) + b * fusc(n/2) + b * fusc(n/2 + 1))} // Multiplicative Distribution
            { (d >= 0) && (d > n/2) && (fusc(N) == (a + b) * fusc(n/2) + b * fusc(n/2 + 1))}
            a := a + b;
            { (d >= 0) && (d > n/2) && (fusc(N) == a * fusc(n/2) + b * fusc(n/2 + 1))}
            n := n / 2;
            { (d >= 0) && (d > n) && (fusc(N) == a * fusc(n) + b * fusc(n + 1))}
        } else {
            { (d >= 0) && (d > (n - 1)/2) && (fusc(N) == a * fusc(n) + b * fusc(n + 1) }
            { (d >= 0) && (d > (n - 1)/2) && (fusc(N) == a * fusc((n - 1 + 1)) + (b * fusc((n - 1 + 2) } // By fusc properties (iv and iii) with (n' == (n-1)/2)
```

```
            { (d >= 0) && (d > (n - 1)/2) && (fusc(N) == a * fusc(((n - 1) / 2)) + (b * fusc(((n - 1) / 2) + 1) + (a * fusc(((n - 1) / 2) + 1)) } // Multiplicative Distribution
            { (d >= 0) && (d > (n - 1)/2) && (fusc(N) == a * fusc(((n - 1) / 2)) + (b + a) * fusc(((n - 1) / 2) + 1)) }
            b := b + a;
            { (d >= 0) && (d > (n - 1)/2) && (fusc(N) == a * fusc(((n - 1) / 2)) + b * fusc(((n - 1) / 2) + 1)) }
            n := ((n - 1)) / 2;
            { (d >= 0) && (d > n) && (fusc(N) == a * fusc(n) + b * fusc(n + 1))}
        }
        { (d >= 0) && (d > n) && (fusc(N) == a * fusc(n) + b * fusc(n + 1))}
    }
    { (n == 0) && (n >= 0) && (fusc(N) == a * fusc(n) + b * fusc(n + 1)) } // Strengthing with loop invariants and loop condition
    { b == fusc(N) }
}

// Part B.

// Derived Code (By the replace a constant by a variable method)
method ComputePos(num: int, den: int) returns (n: int)
{
    var n := 2;
    var x := 1;
    var prevX := 1;
    while (x != den && prevX != num) {
        prevX := x;
        n := n + 1;
        x := ComputeFusc(n);
    }
    n := n - 1;
    return n;
}

// Specification
method ComputePos(num: int, den: int) returns (n: int)
    requires num > 0 && den > 0
    ensures n > 0 && num == fusc(n) && den == fusc(n + 1)
{
    n := 2;
    var x := 1;
    var prevX := 1;
    while (x != den && prevX != num)
        invariant n > 1 && prevX == fusc(n - 1) && x == fusc(n)
    {
        prevX := x;
        n := n + 1;
        x := ComputeFusc(n);
    }
    n := n - 1;
}

// Weakest Precondition Proof (Partial)
method ComputePos(num: int, den: int) returns (n: int)
    requires num > 0 && den > 0
    ensures n > 0 && num == fusc(n) && den == fusc(n + 1)
{
    // This method is partially correct as the given precondition implies the weakest precondition, however termination has not been proven. (num > 0 && den > 0 ==> true) // By law [A.27]
    { true } // By law [A.09]
```

```
    { true && true && true } // By fusc properties ii. and iii. (fusc(2) == fusc(2 * 1) == fusc(1) == 1)
    { 2 > 1 && 1 == fusc(1) && 1 == fusc(2) }
    { 2 > 1 && 1 == fusc(2 - 1) && 1 == fusc(2) }
    n := 2;
    { n > 1 && 1 == fusc(n - 1) && 1 == fusc(n) } // By law [A.74]
    { forall x :: n > 1 && 1 == fusc(n - 1) && 1 == fusc(n) }
    var x := 1;
    { n > 1 && 1 == fusc(n - 1) && x == fusc(n) } // By law [A.74]
    { forall prevX :: n > 1 && 1 == fusc(n - 1) && x == fusc(n) }
    var prevX := 1;
    { n > 1 && prevX == fusc(n - 1) && x == fusc(n) } // By the invariant (and since n > 1 => n > 0)
    while (x != den && prevX != num)
        invariant n > 1 && prevX == fusc(n - 1) && x == fusc(n)
    {
        { n > 0 && x != den && prevX != num} // By law [A.09] and strengthing with loop condition
        { n > 0 && true && true }
        { n > 0 && fusc(n) == fusc(n) && true }
        { n > 0 && x == fusc(n) && true }
        prevX := x;
        { n > 0 && prevX == fusc(n) && true }
        { n + 1 > 1 && prevX == fusc(n - 1) && true }
        n := n + 1;
        { n > 1 && prevX == fusc(n - 1) && true }
        { n > 1 && prevX == fusc(n - 1) && fusc(n) == fusc(n) }
        x := fusc(n);
        { n > 1 && prevX == fusc(n - 1) && x == fusc(n) }
    }
    { x == den && prevX == num && n > 1 && num == fusc(n - 1) && den == fusc(n) } // Strengthening with loop condition
    { n > 1 && num == fusc(n - 1) && den == fusc(n) }
    { n - 1 > 0 && num == fusc(n - 1) && den == fusc(n) }
    n := n - 1;
    { n > 0 && num == fusc(n) && den == fusc(n + 1) }
}
```