```
method ComputeFusc(N: int) returns (b: int)
    requires N >= 0
    ensures b == fusc(N)
{
    {true}

    {fusc(N) == fusc(N)}

    {fusc(N) == fusc(N) + 0 * fusc(N + 1)}

    b := 0;

    {fusc(N) == fusc(N) + b * fusc(N + 1)}

    {forall n :: fusc(N) == fusc(N) + b * fusc(N + 1)}

    var n := N;

    {fusc(N) == fusc(n) + b * fusc(n + 1)}

    {forall a :: 1 * fusc(N) == fusc(n) + b * fusc(n + 1)}

    var a := 1

    {fusc(N) == a * fusc(n) + b * fusc(n + 1)}

    while (n != 0)
        invariant fusc(N) == a * fusc(n) + b * fusc(n + 1)
        decreases n
    {
        {fusc(N) == a * fusc(n) + b * fusc(n + 1)} // strengthen (A || A == B -> A
as this reduces sample space of A)

        {fusc(N) == a * fusc(n) + b * fusc(n + 1) || a * fusc(n) + b * fusc(n + 1)
== a * fusc(n) + b * fusc((n + 1) / 2)}

        {(fusc(N) == a * fusc(n) + b * fusc(n + 1) && (n % 2 == 0 || n % 2 == 1))
||
                                    (fusc(N) == a * fusc(n) + b * fusc(n + 1) &&
fusc(N) == a * fusc(n) + b * fusc((n + 1) / 2)}

        {false || (n % 2 == 1 && fusc(N) == a * fusc(n) + b * fusc(n + 1) ||
(fusc(N) == a * fusc(n) + b * fusc(n + 1) && n % 2 == 0) ||
                                                                  (fusc(N) == a * fusc(n) + b
* fusc(n + 1) && fusc(N) == a * fusc(n) + b * fusc((n + 1) / 2)}

        {(n % 2 == 1 && n % 2 == 0) || (n % 2 == 1 && fusc(N) == a * fusc(n) + b *
fusc(n + 1) / 2) ||
                            (fusc(N) == a * fusc(n) + b * fusc(n + 1) && n % 2 ==
0) || (fusc(N) == a * fusc(n) + b * fusc(n + 1) && fusc(N) == a * fusc(n) + b *
```

```
fusc(n + 1) / 2)}

        {(n % 2 == 1 || fusc(N) == a * fusc(n) + b * fusc(n + 1)) && (n % 2 == 0 ||
fusc(N) == a * fusc(n) + b * fusc(n + 1) / 2)}

        {n % 2 == 0 ==> fusc(N) == a * fusc(n) + b * fusc(n + 1) && n % 2 == 1 ==>
fusc(N) == a * fusc(n) + b * fusc(n + 1) / 2)}

        if (n % 2 == 0) {

            {fusc(N) == a * fusc(n) + b * fusc(n + 1)}          -- as an input of a
number (n) with a multiple of 2 is equal to an input of itself (n) {rule iii}

            {fusc(N) == a * fusc(n / 2) + b * fusc(n + 1)}

            {fusc(N) == a * fusc(n / 2) + b * fusc(2 * (n / 2) + 1)}

            {fusc(N) == a * fusc(n / 2) + b * fusc(n / 2) + b * fusc((n / 2) + 1)}

            {fusc(N) == (a + b) * fusc(n / 2) + b * fusc((n / 2) + 1)}

            a := a + b;

            {fusc(N) == a * fusc(n / 2) + b * fusc((n / 2) + 1)}

            n := n / 2;

            {fusc(N) == a * fusc(n) + b * fusc(n + 1)}

        } else {

            {fusc(N) == a * fusc(n) + b * fusc((n + 1) / 2)}

            {fusc(N) == a * (fusc(2 * ((n - 1) / 2) + 1)) + b * fusc((n + 1) / 2)}

            {fusc(N) == a * (fusc((n - 1) / 2) + fusc(((n - 1) / 2) + 1)) + b *
fusc((n + 1) / 2)}

            {fusc(N) == a * fusc((n - 1) / 2) + a * fusc(((n - 1) / 2) + 1) + b *
fusc((n + 1) / 2)}

            {fusc(N) == a * fusc((n - 1) / 2) + a * (fusc(((n - 1) / 2) + 1) + b *
fusc(((n - 1) / 2) + 1)}

            {fusc(N) == a * fusc((n - 1) / 2) + (b + a) * fusc(((n - 1) / 2) + 1)}

            b := b + a;

            {fusc(N) == a * fusc((n - 1) / 2) + b * fusc(((n - 1) / 2) + 1)}
```

```
            {fusc(N) == a * fusc((n - 1) / 2) + b * fusc(((n - 1) / 2) + 1)}

            n := (n - 1) / 2;

            {fusc(N) == a * fusc(n) + b * fusc(n + 1)}

        }

        {fusc(N) == a * fusc(n) + b * fusc(n + 1)}

    }

    {fusc(N) == a * fusc(n) + b * fusc(n + 1) && n == 0}
}
```

```
method ComputePos(num: int, den: int) returns (n: int)
    requires num > 0 && den > 0
    ensures n > 0 && num == fusc(n) && den == fusc(n + 1)
{
    {true}
    {1 == 1}
    {forall x :: 1 == 1}
    var x := 1;
    {x == 1}
    {x == 1 && true}
    {forall y :: x == 1 && 1 == 1}
    var y := 1;
    {x == 1 && y == 1}
    {forall s :: x == 1 && y == 1}
    var s := 0;
    {x == 1 && y == 1}
    {x == 1 && y == fusc(1)}
    {x == 1 && y == fusc(2 * 1)}
    {true && x == 1 && y == fusc(2)}
    {1 > 0 && x == fusc(1) && y == fusc(1 + 1)}
    n := 1;
    {n > 0 && x == fusc(n) && y == fusc(n + 1)}
    while (num != x && den != y)
        invariant n > 0 && x == fusc(n) && y == fusc(n + 1)
        decreases -n
    {
        {n > 0 && x == fusc(n) && y == fusc(n + 1)}        --  strengthen with
invariants
        {n > 0}
        {(n + 1) >= 0}
        n := n + 1;
        {n >= 0}
        {n >= 0 && true}
        {n >= 0 fusc(n) == fusc(n)}
        {n >= 0 && forall b' :: b' == fusc(n) == fusc(n)}
        x := ComputeFusc(n);
        {n >= 0 && x == fusc(n)}
        {n >= 0 && x == fusc(n) && true}
        {n >= 0 && x == fusc(n) && fusc(n + 1) == fusc(n + 1)}
        s := n + 1;
        {n >= 0 && x == fusc(n) && fusc(s) == fusc(n + 1)}
        {n >= 0 && x == fusc(n) && forall b' :: b' == fusc(s) ==> b' == fusc(n +
1)}
        y := ComputeFusc(s);
        {n > 0 && x == fusc(n) && y == fusc(n + 1)}
    }
}
```