```
/**
ensures forall i :: 0 ⩽ i < a.Length ⟹ a[i] ⩽ a[i+1] this is not good on its own because we could have changed array's values
and not remain with the previous ones. for example array[i] = 7
we need to add a precondition so that we'll remain with the same.
ensures forall x :: x in a[..] ⟹ x in old (a[..]). this is not good because we need to sum the amount of times it's in old_a and a.
ensures (set x | x in a[..]) == (set x | x in old(a[..])) - more elegant to the ensures above, but the same problem. we need multiset!
multiset is both a function and a type
 */


predicate Sorted(q: seq<int>)
{
    forall i,j :: 0⩽ i ⩽ j < |q| ==> q[i] ⩽ q[j]
}
method InsertionSort(a: array<int>)
    requires true // pre
    ensures forall i :: 0 ⩽ i < a.Length -1  ==> a[i] ⩽ a[i+1]
    ensures multiset(a[..]) == multiset(old(a[..])) // old a is the value of a in the paramters of the function.
    modifies a // we can change a, not as in previous functions when they were not part of the frame (can't change local variables). modifies adds to frame.
    {
        //introduce logical constants
        // ghost var A:= multiset(a[..]);
        ghost var A :| A == multiset(a[..]); // :| is a such that, give me a value of A, such that A == multiset... the same as the assignment above. this is what the law expect (such that)
        InsertionSort1(a,A);

    }
    /**
    Adding ghost to function parameters is only for proving correctness. the compiled version will not show up in running time
     */
method InsertionSort1(a:array<int>, ghost A: multiset<int>)
    requires A == multiset(a[..]) //pre'
    ensures Sorted(a[..])
    ensures multiset(a[..]) == multiset(old(a[..])) // == A. the same post condition as inserstion sort.
    modifies a
    {
        //introduce local variable + strengthen postcondition
        var i;
        //the guard will not be satisfied and inv will
        i:= InsertionSort2(a,A);
        L2(a,i,A);
    }
method InsertionSort2(a:array<int>, ghost A: multiset<int>) returns (i: nat)
    requires A == multiset(a[..])
    ensures Inv1(a[..],i,A) && !Guard1(a,i,A) // the L2 lemma says it's okay.
    modifies a
    {
        i:=0;
        while Guard1(a,i,A)
            invariant Inv1(a[..],i,A)
            decreases a.Length - i
            {
                Insert(a,i,A);
                i:=i+1;

            }
    }

method {:verify false} Insert(a:array<int>, i:nat, ghost A: multiset<int>)
    requires Inv1(a[..],i,A) && Guard1(a,i,A)
    ensures Inv1(a[..], i+1,A) // for the incrementing of i.
    modifies a
/**
We have began with i <= |q| && Sorted(q[..i]) in Inv1, but there was a problem for Dfny to prove ensures multiset(a[..]) == A in L2
So we're adding it.
 */
predicate Inv1(q: seq<int>, i:nat, A: multiset<int>)
{

    i <= |q| && Sorted(q[..i]) && multiset(q) == A
}
predicate method Guard1(a:array<int>, i:nat, ghost A: multiset<int>)
{
    i < a.Length
}
/**
All lemmas's parameters are ghost!
we need to think about the main loop in the strength post condition.
 */
lemma L2(a:array<int>, i:nat, A: multiset<int>)
    requires Inv1(a[..],i,A) && !Guard1(a,i,A)
    ensures Sorted(a[..])
    ensures multiset(a[..]) == A


method Main() {
    var a: array<int> := new int[4];
    a[0],a[1],a[2],a[3] := 3,8,5,-1;
    print a[..];
    ghost var q := a[..]; //ghost variables allows in specification contexts (in asserts and such). they are not real variables (not taking place in memory)
    InsertionSort(a);
    assert Sorted(a[..]);
    assert multiset(a[..]) == multiset(q); // this is why we needed the ghost - instead of old(a) which is undefined in current context.
}
```