Exercise 3

Behavioral Subtyping

1.

```
class sortedArray{

 int[] A;
 invariant A ≠ null;
 invariant ∀ i:int | 0 ≤ i ∧ i < A.length -1
                  ⇒ A[i] < A[i+1];

 requires ∀ i:int | 0 ≤ i ∧ i < A.length
                  ⇒ x ≠ A[i];
 ensures A.length = old(A.length) + 1;
 ensures ∃ i0:int | (0 ≤ i0 ∧ i0 < A.length)
 ∧ (∀ i:int | 0 ≤ i  ∧ i < i0 ⇒
      A[i] = old(A[i]))
 ∧ (∀ i:int | i0 < i ∧ i < A.length
       ⇒ A[i] = old(A[i-1]))
 ∧ A[i0] = x;
 void insert (int x){...}
}
```

Here is another way to express the last ensures clause. First of all we need to introduce an auxiliary predicate `contains`:

```
contains (L, x) = ∃ j:int | 0≤j ∧ j<L.length ∧ L[j]=x
```

Using this predicate we can express the desired property as:

```
ensures ∀ i:int |  contains (A, i)  ⇔
     i=x ∨ contains (old(A), i)
```

2.

	$Pre_{super} \Rightarrow Pre_{sub}$	$Post_{sub} \Rightarrow Post_{super}$	Behavioral subtyping
(a)	Yes	Yes	Yes
(b)	Yes	No	No
(c)	Yes	Yes	Yes
(d)	No	Yes	No
(e)	Yes	Yes	Yes
(f)	Yes	Yes	Yes

3. The proposed example violates the behavioral subtyping rules that we currently have. Nevertheless class `B` can be used in any context where class `A` can be used. The source of this mismatch is that we ignore preconditions when checking post-conditions. So if

we want to check that a class `Sub` is a behavioral subtype of a class `Super` it is enough to check that for each inherited method `m`:

- $Pre_{super} \Rightarrow Pre_{sub}$
- $old(Pre_{sub}) \wedge Post_{sub} \Rightarrow Post_{super}$

We can see that the new rules are satisfied for classes A and B:

- `p==p*p  ⇒  p==0 || p==1`
- `result==2 && (p==0 || p==1) ⇒  p<result`

4.

(a) All of the classes have the invariant `content ≠ null`, and in addition the following specific invariants:

- `ArrayNonDecreasing`

```
invariant ∀ i:int | 0 ≤ i ∧ i < content.length -1
          ⇒ content[i] ≤ content[i+1];
```

- `ArrayIncreasing`

```
invariant ∀ i:int | 0 ≤ i ∧ i < content.length -1
          ⇒ content[i] < content[i+1];
```

- `ArrayNoDuplicates`

```
invariant ∀ i,j:int |
         (0 ≤ i ∧ i < content.length)
       ∧ (0 ≤ j ∧ j < content.length)
       ∧ i ≠ j
          ⇒ content[i] ≠ content[j];
```

(b) `ArrayIncreasing` is a behavioral subtype of `ArrayNonDecreasing` and `ArrayNoDuplicates`.

(c) An example of such a method is an `addToFront(int x)` method. The appropriate preconditions for this method are the following:

- `ArrayNonDecreasing`

```
requires content.length > 0 ⇒ x ≤ content[0];
```

- `ArrayIncreasing`

```
requires content.length > 0 ⇒ x < content[0];
```

- `ArrayNoDuplicates`

```
requires ∀ i:int | 0 ≤ i ∧ i < content.length
          ⇒ x ≠ content[i];
```

We can see that the precondition of the method of class `ArrayIncreasing` is not implied by the preconditions of the methods of the other two classes, which violates the previous behavioral subtype relations.

5.

(a)

```
class IncCounter
{
```

```
    int key;
    IncCounter () { key = 0; }

    //ensures key == old(key)+1 && result == old(key)
    int generate () { return key++; }
}
```

(b) The postcondition for `generate` is

```
key == old(key)-1 && result == old(key)
```

and it is easy to see that it does not refine the postcondition of `IncCounter.generate`.

(c) The abstract parent class can be declared using a helper pure method `boolean used(int)`. Informally, the meaning of the helper method is:

`x` has been used as a key before ⇒ `used(x)`

Furthermore, the correctness of the class relies on the property that once a number is used, it never becomes unused again. This can be expressed with a two-state history constraint.

The definitions of the classes follow:

```
abstract class GenerateUniqueKey
{
  abstract boolean used(int);

  //constraint ∀x:int | old(used(x)) ⇒ used(x)

  //ensures !old(used(result)) && used(result)
  abstract int generate ();
}

class IncCounter  // … and similarly for DecCounter
{
  int key;
  IncCounter () { key = 0; }

  boolean used (int x)
  { return x < key; }

  //ensures key == old(key)+1 && result == old(key)
  int generate () { return key++; }
}
```