Exercise 3

Behavioral Subtyping

1.

```
class sortedArray{

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

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

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 (content, i)  ⇔
     i=newElement ∨ contains (old(content), i)
```

2.

```
class A{
  int x;

  ensures result = this.x;
  ensures ∀ o:object, f:field | o.f = old(o.f)
  public int getX(){return x;}

  ensures x = this.x;
  ensures ∀ o:object, f:field | (o ≠ this ∨ f ≠ x)
          ⇒ o.f = old(o.f)
  public void setX(int x){this.x = x;}
}
```

It is possible, in principle, that `getX` and `setX` could affect not only the receiver but other objects as well. In such a case, execution of `a2.setX (x2);` could potentially change the value of the field x of the object `a1`. This can result in violation of the post-condition. To prevent it we need to precisely specify the write effects of the methods `getX` and `setX`.

3.

	$Pre_{super} \Rightarrow Pre_{sub}$	$Post_{sub} \Rightarrow Post_{super}$	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

4. 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 the invariant of the object when we check properties of preconditions, and that we ignore invariants and 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:
   - $Inv_{sub} \Rightarrow Inv_{super}$
   - For each inherited method `m`:
     - $Inv_{sub} \wedge Pre_{super} \Rightarrow Pre_{sub}$
     - $old(Inv_{Sub}) \wedge Inv_{Sub} \wedge old(Pre_{sub}) \wedge Post_{sub} \Rightarrow Post_{super}$

   Note: Inv_{sub} (which is in general stronger than Inv_{super}) can be assumed in both cases, since we are considering the behavior of an object which is actually of the subclass (although it may be accessed via the specification of the superclass).

   We can see that the new rules are satisfied for classes A and B:
   - `f>0 ⇒ f>0`
   - `f>0 ∧ p> f ⇒ p>0`
   - `old(f)>0 ∧ f>0 ∧ old(p)>0 ∧ result=old(p)+f ⇒ result > 0`

5.

(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`

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

- `ArrayIncreasing`

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

- `ArrayNoDuplicates`

```
invariant ∀ 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.

6.

- The intended behaviour is that a `Stack` is first-in-first-out, while a `Queue` is last-in-first-out. Therefore, it is impossible that both the `pop` and `push` methods can have similar behaviours across the two classes, and so neither class can be a behavioural subtype of the other.
- Depending on the internal representation, either the `pop()` or the `push()` method (but not both) could be re-used, from one implementation to the other. For example, if one implements a `Queue` by pushing to the end of a linked list, and popping from the beginning, then a `Stack` could be implemented either by pushing on the beginning of the list and reusing the `pop()` method, or by reusing the `push()` method and popping from the end of the list. Furthermore, it's likely that the `isEmpty()`, `size()` and `reverse()` methods could all be reused.
- Any mechanism which allows code reuse without subtyping, e.g., private inheritance in C++. In principle, aggregation could be employed, but the "common class" would be rather strange (e.g., a list which could only grow, and only at one end). Traits might also provide a solution to this problem, but again, identifying a fragment of the implementation to abstract out might not be natural. One could argue that this kind of code reuse binds the implementations too closely together, when it might be that one or other class wants to evolve independently (e.g., given some other desired methods, we want to change the underlying implementation of one class in a way which isn't helpful for the other). However, the ability to reuse a large number of common methods seems tempting.