Question 1 (24 points)
a) (4 points)

```
FiniteStringBag represents a bag (or multiset) with max size ==
items.length. If size==0, this FiniteStringBag represents the
empty bag { }, otherwise the strings in items[0..size-1]
represent the FiniteStringBag{items[0], ..., items[size-1]} with
size <= items.length.
```

The part about max size == items.length is redundant if they say size <= items.length. If they leave off the part about size == 0, give full points, it's redundant if the rest is there. It's OK to use capacity() instead of items.length. If they use the variable `capacity` instead, deduct 0.5 points. Capacity isn't a field in the object. They must include something about the fixed size, if not deduct a point. They can use the capacity() method.

b)(5 points)
Should check for nulls and check size is <= items.length (capacity). Deduct a point if they use capacity as a variable. They can use the capacity() method.

```
private void checkRep() throws RuntimeException {
     if (size < 0 || size > items.length)
         throw new RuntimeException("size out of bounds");
     for (int k = 0; k < size; k++)
         if (items[k] == null)
          throw new RuntimeException("null item");
    }
}
```

c) (9 points) (3 points each)
There are, of course, many possible tests. Here are some. Give points for reasonable attempts. They need 3 different tests.

```
(a)
Initialize bag b to { "a", "b", "c" }
b.deleteLongStrings(3)
Verify that b contains {"a", "b", "c"}

(b)
Initialize bag b to {
"xyzzy", "", "ab", "abcd", "abcdefg"}
b.deleteLongStrings(4)

Verify that b contains {"", "ab", "abcd"}

(c)
Initialize bag b to the empty bag { }
b.deleteLongStrings(1)
Verify that b is still the empty bag { }
```

```
(d)
Initialize bag b to { "abcd", "pqrstuv", "wxyz"
}b.deleteLongStrings(3)
Verify that b is the empty bag { }

(e)
Initialize bag b to { "abc", "abc", "abc" }
Verify that b contains 3 copies of "abc"
```

d) (2 points) No. All instance variables are private and the only data that is shared by the client code are references to immutable String objects, which the client cannot change.
Give 1 point if they just say No without an explanation.

e) (4 points, 2 points each)
It returns the whole array, but only 0…size-1 are valid entries.
Returning `items` is a rep exposure.

Question 2 (18 points)

	a1	a2	a3	b1	b2	c1
a1	AA	AA	AA	AB	AB	AC
a2	BA	BA	BA	BB	BB	BC
a3	CA	CA	CA	CB	CB	CC
b1	BA	BA	BA	BB	BB	BC
b2	CA	CA	CA	CB	CB	CC
c1	CA	CA	CA	CB	CB	CC

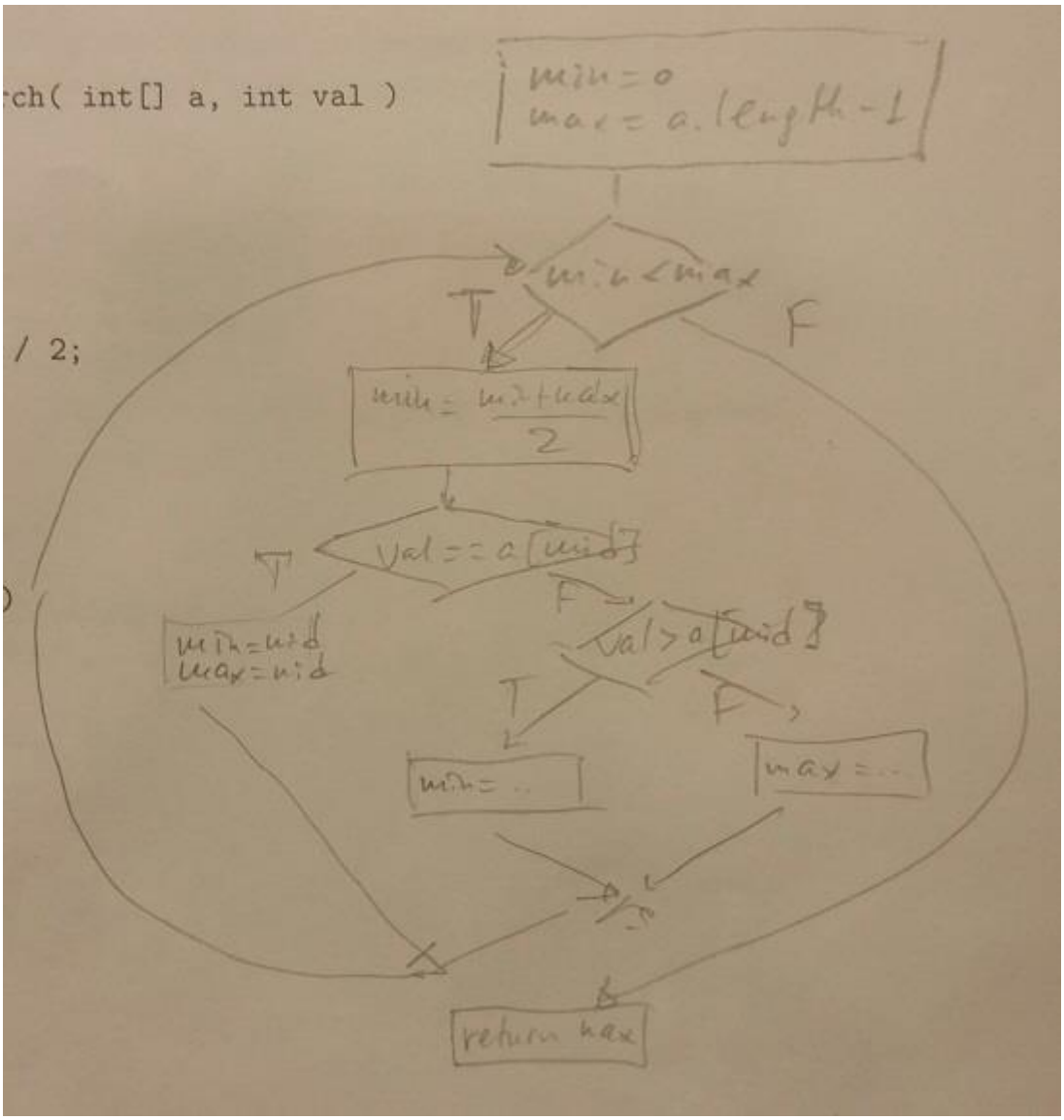
Question 3 (14 points)

b) False
f)) (2 points, 1 point for each condition)
Both of these must hold:
T1 is the same as, or is a subtype of T, can also say return type are covariant
S1 is the same as, or is a supertype of S, can also say arguments are contravariant
They don't have to say same, subtype or supertype is enough.
g) (2 points, 1 point each)

```
static void removeDuplicates(Collection<? extends T> src,
                             Collection <? super T> dst);
```

Question 4 (14 points, 2 points each)
a) ERROR
b) OK
c) ERROR
d) OK
e) ERROR
f) OK
g) OK

Question 5 (8 points) Apologies for the picture quality.



Question 6

a) (5 points)
Must have the following (or an equivalent variation):
stackArray.length = maxSize && -1 <= top < maxSize.
For points, 2pts for stackArray.length = maxSize
and 3pts for -1 <= top < maxsize.
Could also specify && maxSize >= 0, though this is implied.

b) (6 points)

The aim of this question was to correct the push() and pop() methods to avoid overow and underow, respectively, though other valid answers could exist here(i.e., use your own judgement in grading).
3pts for each of the two corrected methods below.

```
public void push( long j ) {
        if ( ! this.isFull() ) // use of "this" here is optional
                stackArray[++top] = j;
                // could be an else here that throws an exception (optional)
}

public long pop() {
l       f ( ! this.isEmpty() ) // use of "this" here is optional
                return stackArray[top--];
        else
                throw SomeException; // Any exception here should be fine
}
```

Question 7 (10 points)

The rep invariant does not hold in the following specific ways:
(6pts) representation exposure via getStart() and getStop() methods
(4pts) duration not recalculated in setStart() and setStop() methods

Question 8 (10 points, 1 point each)
c,e,h are true.

Question 9 (10 points, 5 points each)
Rep invariant: doesn't have to be exactly the same, but must make sense

```
  // Rep invariant: degree = coeffs.length-1
  //            coeffs[degree] != 0
```

AF: must be consistent with rep invariant

```
// Abstraction function: coeffs [a0,a1,…adegree]
// represents polynomial
// adegree*x^adegree + … + a1*x + a0
// E.g., array [-2,1,3] is equivalent to 3x^2 + x - 2
// Empty array represents the 0 polynomial
```

Question 10 (8 points, 4 points each)

```
A: X m(Z z)
C: Y m(W w)
```

Question 11 (6 points)

a) All of these are trivially true if equivalence is defined as object equality:
   1. a==a always.
   2. if a==b then a and b refer to the same object, so b==a.
   3. If a==b and b==c, then all three refer to the same object, so a==c.

b) The requirement on hashCodeis that if a.equals(b) then it must be true that a.hashCode()==b.hashCode(). If a.equals(b) according to the equals method in Object, then we know that a==b. That means a and b have the same memory address, so they both have the same hashCode() value using Object's implementation of hashCode.

Question 12 (6 points)

(iii) is not a valid hashCode. a.equal(b) => hashCode(a) == hashCode(b). Math.Random will return a different value each time, so if two Property objects are equal, their hashCodes will not be.