CSE 331 SECTION 6: MIDTERM REVIEW

Reasoning about Code

1. *Using backwards reasoning, find the weakest precondition for each sequence of statements and postcondition below. Insert appropriate assertions in each blank line. You should simplify your answers if possible.*

   A.

```
{{ ____________________________________________________________ }}
z = x + y;
{{ ____________________________________________________________ }}
y = z - 3;
{{ x > y }}
```

   B.

```
{{ ____________________________________________________________ }}
p = a + b;
{{ ____________________________________________________________ }}
q = a - b;
{{ p + q = 42 }}
```

2. *Given two strings a and b where a.length > 0 and b.length > 0 that are only comprised of alphabetic characters a-z, fill in the implementations for the method* `arePermutations` *which returns true if a and b are permutations of each other and false otherwise. You do not need to turn in a complete proof of correctness, but you should complete one to ensure your code is correct.*

Implementation 1:

```
public boolean arePermutations(String a, String b) {

    {inv: sortedA = sorted(a[0] ... a[k-1]) && sortedB = sorted(b[0] ... b[k-1]) && a.length == b.length}
    while (                                    ) {

    }

}
```

Implementation 2:

```
public boolean arePermutations(String a, String b) {

    {inv: counts[0] = # of a's in a[0], ..., a[i-1], ..., counts[25] = # of z's in a[0], ..., a[i-1] && a.length == b.length}
    while (                    ) {



    }

    {inv: counts[0] >= 0, ... , counts[25] >= 0 && a.length == b.length}
    while (                    ) {



    }

}
```

Testing

3. *For the previous implementations of* `arePermutations`*, write two test cases where the inputs result in expected/actual behavior that is fundamentally different from each other. Write a brief explanation convincing someone else why your test cases test different behavior.*

`Input: a =` _____ `and b =` _____
`Returns:`

Explanation:

`Input: a =` _____ `and b =` _____
`Returns:`

Explanation:

4. *Suppose we have a class* `IntPoly` *(similar to RatPoly in hw4). We would like to add a method to this class that evaluates the* IntPoly *at a particular value x. In other words, given a value x, the method* `valueAt(x)` *should return* $a_0 + a_1x + a_2x^2 + \ldots + a_nx^n$*, where* a_0 *through* a_n *are the coefficients of this* `IntPoly`*. Suppose you have developed the following implementation of this method. Prove that your implementation is correct by filling in the reasoning.*

```
/** Return the value of this IntPoly at point x */
public int valueAt(int x) {
   int val = a[0];
   int xk = 1;
```

```
    int k = 0;
    int n = a.length-1; // degree of this, n >=0
    {{ Inv: xk = x^k && val = a[0] + a[1]*x + ... + a[k]*x^k }}
    while (k != n) {
       {{ _____________________________________________________ }}
       xk = xk * x;
       {{ _____________________________________________________ }}
       val = val + a[k+1]*xk;
       {{ _____________________________________________________ }}
       k = k + 1;
       {{ _____________________________________________________ }}
    }
    {{ _____________________________________________________ }}
    return val;
}
```

Specifications

5. *Suppose we have a* `BankAccount` *class with instance variable* `balance`*. Consider the following three specifications:*

   A. `@effects decreases balance by amount`

   B.
   ```
   @requires amount >= 0 and amount <= balance
   @effects decreases balance by amount
   ```

   C.
   ```
   @throws InsufficientFundsException
      if balance < amount
   @effects decreases balance by amount
   ```

   Which specifications do each of these implementations meet? Write A, B, *and/or* C *for each implementation.*

   Example:
   ```
   void withdraw(int amount) {
        balance -= amount;
   }
   ```
   Specifications: A, B

I.
```
void withdraw(int amount) {
   if (balance >= amount) balance -= amount;
}
```
Specifications: _____

II.
```
void withdraw(int amount) {
   if (amount < 0) throw new IllegalArgumentException();
   balance -= amount;
}
```
Specifications: _____

III.
```
void withdraw(int amount) throws InsufficientFundsException {
   if (balance < amount) throw new InsufficientFundsException();
   balance -= amount;
}
```
Specifications: _____

6. *Fill out the specification for following* `add` *method*:

```
/**
 * Return a new IntPoly that is the sum of this and other
 *
 * @requires _____
 *
 * @modifies _____
 *
 * @effects  _____
 *
 * @return   _____
 *
 * @throws   _____
 */
public IntPoly add(IntPoly other)
```

Defensive Programming

One of your colleagues is worried that the implementation below creates a potential representation exposure problem. Another colleague says there's no problem since an IntPoly *is immutable.*

```
public class IntPoly {
   int a[];
   // AF(this) = a has n+1 entries, and for each entry,
   // a[i] = coefficient a_i of the polynomial.

   // Return the coefficients of this IntPoly
   public int[] getCoeffs() {
      return a;
   }
}
```

7. *Explain how representation exposure would be a problem here and change the implementation of getCoeffs to fix the problem, but still return the coefficients of the IntPoly to the client. You may give a description of the changes instead of writing Java code.*

Object Equality

Suppose we are defining a class StockItem to represent items stocked by an online grocery store. Here is the start of the class definition, including the class name and instance variables:

```
public class StockItem {
   String name;
   String size;
   String description;
   int quantity;

   /* Construct a new StockItem */
   public StockItem(...);
}
```

A summer intern was asked to implement an equals *function for this class that treats two* StockItem *objects as equal if their* name *and* size *fields match. Here's the result:*

```
/** return true if the name and size fields match */
public boolean equals(StockItem other) {
   return name.equals(other.name) && size.equals(other.size);
}
```

8. *This* equals *method seems to work sometimes but not always. Give an example showing a situation when it fails. (**Hint**: Instantiate two objects that should be equal but are not.)*