Unless otherwise stated all variables are ints.

Question 1. (10 pts) Draw the contents of the stack and heap after the execution of the following Java code. If a heap item becomes "garbage", do not erase it, place an x over the reference line that pointed to it.

```
int x = 27;
int y;
double [] a = {1.0,2.0,3.14,4.0};
double [] b = new double[3];
String s = new String("cat");
String t = null;

y = x;
t = s;
s = new String("dog");
t = t + "2";
```

Question 2. (6 points)

Order the conditions from strongest to weakest. Note: All variables are ints.

(a) y = 11
(b) y > -1
(c) y > 0
(d) 0 < y <= 11

Answer: _ , _ , _ , _

(a) abs(result*result - x) ≤ 0.0001 assume result is a double
(b) abs(result*result - x) ≤ 0.000001

Answer: _ , _ ,

Assume the following class hierarchy: C is a subclass of B and B is a subclass of A.

(a) { x is an object of type A }
(b) { x is an object of type C }
(c) { x is an object of type B }

Answer: _ , _ , _

Question 3. (6 pts) For each of the following Hoare Triples, indicate which are valid and which are not necessarily true. If the triple is not valid, explain why or give a counter example. All variables are ints.

a) (VALID / NOT VALID) {true} x = 1; x = x + 1 {x = 1}

b) (VALID / NOT VALID) {x >= -1} x = x + 1 {x > 0}

c) (VALID / NOT VALID) {z is odd && z > 0} w = z + 3 {w is even && z % 2 == 1}

Question 4. (6 pts) Assume that the following are true. Indicate which Hoare triples are valid

{b} code {y}
a ➔ b (a implies b)
b ➔ c
x ➔ y
y ➔ z

a) {a} code {y}

b) {b} code {x}

c) {b} code {z}

Question 5. (12 pts) Compute the weakest precondition using **backward** reasoning. Fill in all intermediate conditions at the designated places. Simplify your weakest precondition, Assume all variables are ints.

a)

```
P: {_________________________________}
if (x >=0 ) {
          {_________________________________}
     z = x;
}
else {
          {_________________________________}
     z = x+1;
}
{z != 0}
```

b)

```
P: {_______________________________________}
b = b+1;
          {_________________________________}
a = b + 1;
{a > 0 && b > 0}
```

c)

```
P:{_____________________________}
t=x;
     {_____________________________}
x=y;
     {_____________________________}
y=t;;
{ x > y }
```

Question 6. (8 pts) Compute the strongest postcondition using **forward** reasoning. Fill in all intermediate conditions at the designated places. Simplify your final postcondition, Assume all variables are ints.

a)
```
        { { x < 0 && y > 0 }
y = 2;
        { _________________________ }
x = x + y;
        { _________________________ }
```

b)
```
        { |x| > 2 }
x = x * 2;
        { _________________________ }
x = x - 1;
        { _________________________ }
```

Question 7. (12 pts) TRUE/FALSE.

a) (TRUE/FALSE) If specification A is stronger than specification B, then any implementation that satisfies B satisfies A as well.

b) (TRUE/FALSE) There may exist two logically distinct weakest preconditions A and B for a given bit of Java code. (Logically distinct means that A and B are not just different ways of writing exactly the same logical formula.)

c) (TRUE/FALSE) Keeping the precondition the same, you can strength a specification by weakening the postcondition.

Consider the following method declaration for binary search:

public static int binarySearch(int[] a, int key)

For each of the following specifications, indicate whether it is a valid specification for this code.

d) (TRUE/FALSE)

@requires a is sorted and key is contained in a
@modifies nothing
@returns i such that a[i] = key

e) (TRUE/FALSE)

@requires a is sorted
@modifies nothing
@returns i such that a[i] = key if such an i exists and a negative value otherwise
@throws NullPointerException

f) (TRUE/FALSE)

@requires a is not null
@modifies nothing
@returns returns i such that a[i] = key if such an i exists and a negative value otherwise

g) (TRUE/FALSE) The rep invariant must hold before and after every statement in every method.

h) (TRUE/FALSE) The abstraction function maps valid objects to a boolean.

Question 8. (12 pts)
Consider the following specifications for a method that has one int argument:
(a) Returns an integer ≥ the argument
(b) Returns a non-negative integer ≥ the argument
(c) Returns argument – 1
(d) Returns argument^2 (i.e., the square of the argument)
(e) Returns a non - negative number

Consider these implementations, where arg is the function argument value:
(i) return arg + 5;
(ii) return arg * arg;
(iii) return arg % 10;
(iv) return Math.abs(arg);
(v) return Integer.MAX_VALUE;

Place a check mark in each box for which the implementation satisfies the specification. If the implementation does not satisfy the specification, leave the box blank

	(a)	(b)	(c)	(d)	(e)
(i)					
(ii)					
(iii)					
(iv)					
(v)					

Question 9 (14 pts)
Prove that the given code below computes the correct result if the code terminates. To do so, identify the loop invariant and decrementing function. Next, prove that the loop invariant holds for the base case and the general case, then prove that the decrementing function is indeed a decrementing function that decreases to zero. Show that at exit, the negation of the loop condition and the loop invariant imply the postcondition.

Precondition: arr is not null and arr is not empty
```
int product(int [] arr) {
    n = 1;
    p = arr[0];
    while (n < arr.length) {
        p = p * arr[n];
        n = n + 1;
    }

    return p;
}
Postcondition: p = arr[0] * … arr[arr.length-1]
```

Question10. (10 pts) Below is the (simplified) Javadoc specification of a **get** method from class ArrayList.

public E get(int index)

Returns the element at the specified position in this list.

Parameters:
index - index of the element to return

Returns:
The element at the specified position

Throws:
IndexOutOfBoundsException - if the index is out of range (index < 0 || index >= size())

a) Convert the Javadoc specification into a PoS specification:

requires:

modifies:

effects:

returns:

throws:

b) Now, convert your PoS specification into a logical formula:

Question 11 (16 pts)

Our friend Willie Wazoo has created the method shown below. Willie thinks this code works properly. That is, he thinks it calculates a * b when the preconditions are met, but is not completely sure it works in general. We will help Willie out by proving that given that the preconditions hold, the postcondition will hold.

```
// precondition a >= 0
static int mult(int a, int b) {
    int x = a;
    int y = b;
    int total = 0;
    System.out.println("x" + "\t" + "y" + "\t" + "total");
    System.out.println(x + "\t" + y + "\t" + total);
    while( x > 0 ) {
        if(x % 2 == 1) {
                x = (x - 1)/2;  // integer division
                total = total + y;
        } else {
            x = x / 2;  // integer division
        }
        y = y * 2;
        System.out.println(x + "\t" + y + "\t" + total);
    }

    return total;
}
// postcondition: total == a*b
```

To help understand the code, Willie tested the code with mult(18, 3) and generated this table:

x	y	total
18	3	0
9	6	0
4	12	6
2	24	6
1	48	6
0	96	54

The method above returned 54 for mult(18,3).

a) (2 points) Identify the loop invariant. Willie's table may give a hint.

b) (2 points) Using the loop invariant you defined above, show that the loop invariant holds before the first iteration of the loop, i.e. show that the base case holds.

c) (8 points) Assume that the invariant is true at some iteration k and show by **induction** that it is true at the next iteration.

d) (2 points) Show that the negation of the loop condition and the loop invariant imply the postcondition. That is, `!(x > 0) && LI => total == a*b.`

e) (2 points)
Choose a decrementing function D and show that it decreases with each iteration. Show that when D is at its minimum, the loop exit condition is implied.

Question 12 (24 points)

Our friend Willie Wazoo has created the code below to implement a finite string bag (FSB). A FiniteStringBag is a set that allows duplicate values, also called a multiset. The representation uses a fixed length array to hold the elements and has an associated integer variable to record the size of the bag (the number of elements being used in the array).

```
class FiniteStringBag {
       // Rep: items is a fixed length array
        // items[0..size-1] contains Strings
        // size <= items.length
       // Entries cannot be null. There may be multiple copies
       // of the same string in the FiniteStringBag (FSB).
       private String[] items;
       private int size;

       //Construct new FSB with given capacity.
       // Requires: capacity > 0
       public FiniteStringBag(int capacity) {
              this.items = new String[capacity];
              this.size = 0;
       }

       /** Return capacity of this FSB */
       public int capacity() { return items.length; }
       /** Return current size of this FSB*/
       public int size() { return size; }
```

```
    /** Return item at position i of this FSB */
    public String get(int i) {
            if(i >= size)
                    throw new IndexOutOfBoundsException();
            return items[i];
    }
    public void add(String s) {
            if(size == items.length)
                    throw new BufferOverflowException();
            items[size] = s;
            size = size + 1;
    }
    /** Return whether s is located in this FSB */
    public boolean contains(String s) {
            if (size == 0) return false;
            for(int i = 0; i < size; i++) {
                    if(items[i].equals(s))
                            return true;
            return false;
    }
    // delete strings with length > n
    public void deleteLongStrings(int n) {
            int k = 0;
            while (k < size) {
                    if (items[k].length() > n) {
                            items[k] = items[size-1];
                            size = size-1;
                    } else {
                            k = k + 1;
                    }
            }
    }

    // additional methods to be added later....…
```

a) (4 points)
Give a suitable abstraction function (AF) for this class relating the representation to the abstract value of a finite string bag.

b) (5 points)

`FiniteStringBag` lacks a `checkRep()` method. Write a `checkRep()` method for this class. The method should check the representation as described in the comments at the start of the class definition.

c) (9 points)
Describe three separate, distinct "black box" tests for the deleteLongStrings method. You don't need to write Java or Junit code. Just give a brief clear description of the test.

d) (2 points)

Are there any potential problem from representation exposure with the FiniteStringBag as it written above? Why or why not? Be brief.

e) (4 points)

We would like to add an observer method to FiniteStringBag. Willie proposes the following method

```
// return the current strings in this FSB to the caller
public String[] getItems() {
      return items;
}
```

Willie's colleague Ima Hacker points out that there are two problems with this method. What are they?

Question 13 (8 points)

Consider this implementation of a binary search. Draw the control-flow graph (CFG) for this implementation.

```
public static int binarySearch( int[] a, int val ) {
        int min = 0;
        int max = a.length - 1;
        while ( min < max ) {
                int mid = ( min + max ) / 2;
                if ( val == a[mid] ) {
                        min = mid;
                        max = mid;
                }
                else if ( val > a[mid] ) {
                        min = mid + 1;
                }
                else {
                        max = mid - 1;
                }
        }
        return max;
}
```

Question 14 (11 points)

Consider the following implementation of a stack data structure.
In a stack, elements are added to the top of the stack via push() and removed from the top of the stack via pop(). Note that there are a few bugs in the given implementation.

```
public class LongStack {
	private int maxSize;
	private long[] stackArray;
	private int top;

	public LongStack( int maxSize ) {
		this.maxSize = maxSize;
		this.stackArray = new long[maxSize];
		top = -1;
	}

	public void push( long j ) { stackArray[++top] = j; }
	public long pop() { return stackArray[top--]; }
	public boolean isEmpty() { return ( top == -1 ); }
	public boolean isFull() { return ( top == maxSize - 1 ); }
}
```

a) (5 points) Write a suitable representation invariant for the LongStack class.

b) (6 points) Make minimal corrections to the implementation of the LongStack class to fix any bugs. Note that you are only allowed to make changes to the method implementations (i.e., the representation fields and method signatures cannot be changed).

Rewrite methods in the space below, as necessary.

Question 15 (10 points)

Consider the Interval class shown below, which represents an interval of time between two Date objects (i.e., start and stop).

```
public class Interval {
        private Date start;
        private Date stop;
        private long duration;

        // Rep invariant: duration = stop.getTime() - start.getTime();

        public Interval( Date start, Date stop ) {
                this.start = start;
                this.stop = stop;
                duration = stop.getTime() - start.getTime();
        }

        public Date getStart() { return start; }
        public Date getStop() { return stop; }
        public long getDuration() { return duration; }
        public void setStart( Date start ) { this.start = start; }
        public void setStop( Date stop ) { this.stop = stop; }
}
```

List all ways in which the representation invariant does not hold.

Question 16 (10 points)

The following partial class definition is for an implementation of a polynomial with integer coefficients.

```
    class IntPoly {
        private int [] coeffs;  // the integer coefficients
        private int degree;    // the degree of the polynomial

        // the rest of the class definition follows….
    }
```

Write a suitable rep invariant for this class.

Next, write an abstraction function.