IT314: Software Engineering
Lab Session IX – Functional and Structural Testing

Prerequisites: Before the session you should review the lecture slides on black and white-box (or structural) testing.
All the exercise to be performed in Junit Testing Framework.

Section A:
Consider a program for determining the previous date. Its input is triple of day, month and year with the following ranges 1 <= month <= 12, 1 <= day <= 31, 1900 <= year <= 2015.The possible output dates would be previous date or invalid date. Design the equivalence class test cases?

Write a set of test cases (i.e., test suite) – specific set of data – to properly test the programs. Your test suite should include both correct and incorrect inputs.

1. Enlist which set of test cases have been identified using Equivalence Partitioning and Boundary Value Analysis separately.
2. Modify your programs such that it runs on `eclipse IDE`, and then execute your test suites on the program. While executing your input data in a program, check whether the identified expected outcome (mentioned by you) is correct or not.

The solution of each problem must be given in the format as follows:

Tester Action and Input Data	Expected Outcome
Equivalence Partitioning	
a, b, c	*An Error message*
a-1, b, c	*Yes*
Boundary Value Analysis	
a, b, c-1	*Yes*

For details you may also refer to an **example:** Testing a simple program [Appendix]

Programs:

P1. The function `linearSearch` searches for a value `v` in an array of integers `a`. If `v` appears in the array `a`, then the function returns the first index `i`, such that `a[i] == v`; otherwise, -1 is returned.

```
int linearSearch(int v, int a[])
{
	int i = 0;
	while (i < a.length)
	{
		if (a[i] == v)
			return(i);
		i++;
	}
	return (-1);
}
```

P2. The function `countItem` returns the number of times a value `v` appears in an array of integers `a`.

```
int countItem(int v, int a[])
```

```
{
      int count = 0;
      for (int i = 0; i < a.length; i++)
      {
            if (a[i] == v)
                  count++;
      }
      return (count);
}
```

P3. The function `binarySearch` searches for a value v in an ordered array of integers `a`. If `v` appears in the array `a`, then the function returns an index `i`, such that `a[i] == v`; otherwise, -1 is returned.

Assumption: the elements in the array `a` are sorted in non-decreasing order.

```
int binarySearch(int v, int a[])
{
      int lo,mid,hi;
      lo = 0;
      hi = a.length-1;
      while (lo <= hi)
      {
            mid = (lo+hi)/2;
            if (v == a[mid])
                  return (mid);
            else if (v < a[mid])
                  hi = mid-1;
            else
                  lo = mid+1;
      }
      return(-1);
}
```

P4. The following problem has been adapted from The Art of Software Testing, by G. Myers (1979). The function `triangle` takes three integer parameters that are interpreted as the lengths of the sides of a triangle. It returns whether the triangle is equilateral (three lengths equal), isosceles (two lengths equal), scalene (no lengths equal), or invalid (impossible lengths).

```
final int EQUILATERAL = 0;
final int ISOSCELES = 1;
final int SCALENE = 2;
final int INVALID = 3;
int triangle(int a, int b, int c)
{
      if (a >= b+c || b >= a+c || c >= a+b)
            return(INVALID);
      if (a == b && b == c)
            return(EQUILATERAL);
      if (a == b || a == c || b == c)
            return(ISOSCELES);
      return(SCALENE);
```

```
}
```

P5. The function `prefix (String s1, String s2)` returns whether or not the string `s1` is a prefix of string `s2` (you may assume that neither `s1` nor `s2` is null).

```
public static boolean prefix(String s1, String s2)
{
      if (s1.length() > s2.length())
      {
            return false;
      }
      for (int i = 0; i < s1.length(); i++)
      {
            if (s1.charAt(i) != s2.charAt(i))
            {
                  return false;
            }
      }
      return true;
}
```

P6: Consider again the triangle classification program *(P4)* with a slightly different specification: The program reads floating values from the standard input. The three values `A`, `B`, and `C` are interpreted as representing the lengths of the sides of a triangle. The program then prints a message to the standard output that states whether the triangle, if it can be formed, is scalene, isosceles, equilateral, or right angled. Determine the following for the above program:

a) Identify the equivalence classes for the system
b) Identify test cases to cover the identified equivalence classes. Also, explicitly mention which test case would cover which equivalence class.
   (Hint: you must need to be ensure that the identified set of test cases cover all identified equivalence classes)
c) For the boundary condition $A + B > C$ case (scalene triangle), identify test cases to verify the boundary.
d) For the boundary condition $A = C$ case (isosceles triangle), identify test cases to verify the boundary.
e) For the boundary condition $A = B = C$ case (equilateral triangle), identify test cases to verify the boundary.
f) For the boundary condition $A^2 + B^2 = C^2$ case (right-angle triangle), identify test cases to verify the boundary.
g) For the non-triangle case, identify test cases to explore the boundary.
h) For non-positive input, identify test points.

Lab Execution (how to perform the exercises):

1. Identify test cases for each program, document them in a separate file **(submit).**
2. Write source code in eclipse IDE, write so obtained test cases in Junit testing framework and run the test cases.
3. Submit the snapshot of test runner, source code of each programs and corresponding test cases.

Section B:

The code below is part of a method in the ConvexHull class in the VMAP system. The following is a small fragment of a method in the ConvexHull class. For the purposes of this exercise you do not need to know the intended function of the method. The parameter p is a Vector of Point objects, p.size() is the size of the vector p, (p.get(i)).x is the x component of the i^{th} point appearing in p, similarly for (p.get(i)).y. This exercise is concerned with structural testing of code and so the focus is on creating test sets that satisfy some particular coverage criterion.

```
Vector doGraham(Vector p) {
        int i,j,min,M;

        Point t;
        min = 0;

        // search for minimum:
        for(i=1; i < p.size(); ++i) {
            if( ((Point) p.get(i)).y <
                         ((Point) p.get(min)).y )
            {
                min = i;
            }
        }

        // continue along the values with same y component
        for(i=0; i < p.size(); ++i) {
            if(( ((Point) p.get(i)).y ==
                          ((Point) p.get(min)).y ) &&
                 (((Point) p.get(i)).x >
                          ((Point) p.get(min)).x ))
            {
                 min = i;
            }
        }
```

For the given code fragment you should carry out the following activities.

1. Convert the Java code comprising the beginning of the doGraham method into a control flow graph (CFG).
2. Construct test sets for your flow graph that are adequate for the following criteria:
    a. Statement Coverage.
    b. Branch Coverage.
    c. Basic Condition Coverage.
3. For the test set you have just checked can you find a **mutation** of the code (i.e. the deletion, change or insertion of some code) that will result in failure but is not detected by your test set.
4. Create a test set that satisfies the path coverage criterion where every loop is explored at least zero, one or two times.

Lab Execution (how to perform the exercises):

4. After generating the control flow graph, check whether your CFG match with the CFG generated by ***Control Flow Graph Factory Tool*** and ***Eclipse flow graph generator***. (In your submission document, mention only "Yes" or "No" for each tool).

5. Devise minimum number of test cases required to cover the code using the aforementioned criteria.

6. This part of the exercise is very tricky and interesting. The test cases that you have derived in Step 2 are then used to identify the fault when you make some modifications in the code.

   Here, you need to insert/delete/modify a piece of code that will result in failure but it is not detected by your test set – derived in Step 2.

   Write/identify a mutation code for each of the three operation separately, i.e., by deleting the code, by inserting the code, by modifying the code.

7. Write all test cases that can be derived using path coverage criterion for the Java code.
