IT559
Assignment – 3

Aviraj Karangiya 202001185

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?

  Based on the input ranges we can identify the following Equivalence classes.

  Day:

Equivalence Classed	Range	Status
E1	Between 1 and 28	Valid
E2	Less than 1	Invalid
E3	Greater than 31	Invalid
E4	Equals 30	Valid
E5	Equals 29	Valid for leap year
E6	Equals 31	Valid

  Month:

Equivalence Classed	Range	Status
E7	Between 1 and 12	Valid
E8	Less than 1	Invalid
E9	Greater than 12	Invalid

Year:

Equivalence Classed	Range	Status
E10	Between 1900 and 2015	Valid
E11	Less than 1	Invalid
E12	Greater than 2015	Invalid

Equivalence Partitioning Test Cases:

Test Action & Input Data	Expected Outcome
Valid input: day=1, month=1, year=1900	Invalid Date
Valid input: day=31, month=12, year=2015	Previous Date
Invalid input: day=0, month=6, year=2000	An error message
Invalid input: day=32, month=6, year=2000	An error message
Invalid input: day=29, month=2, year=2001	An error message

Boundary Value Analysis:

1. The earliest possible date: (1, 1, 1900)
2. The latest possible date: (31, 12, 2015)
3. The earliest day of each month: (1, 1, 2000), (1, 2, 2000), (1, 3, 2000),..., (1, 12, 2000)
4. The latest day of each month: (31, 1, 2000), (28, 2, 2000), (31, 3, 2000),..., (31, 12, 2000)
5. Leap year day: (29, 2, 2000)

6. Invalid leap year day: (29, 2, 1900)
7. One day before earliest date: (31, 12, 1899)
8. One day after latest date: (1, 1, 2016)

Based on the Boundary Value Analysis we can generate following test cases,

Tester Action and Input Data	Expected Outcome
Valid input: day=1, month=1, year=1900	Invalid date
Valid input: day=31, month=12, year=2015	Previous date
Invalid input: day=0, month=6, year=2000	An error message
Invalid input: day=32, month=6, year=2000	An error message
Invalid input: day=29, month=2, year=2000	An error message
Valid input: day=1, month=6, year=2000	Previous date
Valid input: day=31, month=5, year=2000	Previous date
Valid input: day=15, month=6, year=2000	Previous date
Invalid input: day=31, month=4, year=2000	An error message

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.

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);
}
```

```
@Test
public void test() {
   unittesting obj = new unittesting();
   int[] arr1 = {2, 4, 6, 8, 10};
   int[] arr2 = {-3, 0, 3, 7, 11};
   int[] arr3 = {1, 3, 5, 7, 9};
   int[] arr4 = {};
```

```
    assertEquals(0, obj.linearSearch(2, arr1));
    assertEquals(4, obj.linearSearch(10, arr1));
    assertEquals(-1, obj.linearSearch(3, arr2));
    assertEquals(4, obj.linearSearch(9, arr3));
    assertEquals(-1, obj.linearSearch(2, arr4));
}
```

Equivalence Partitioning:

Tester Action and Input Data	Expected Outcome
v is present in a	Index of v
v is not present in a	-1

Boundary Value Analysis:

Tester Action and Input Data	Expected Outcome
Empty array a	-1
v is present at the first index of a	0
v is present at the last index of a length of a	a-1
v is not present in a	-1

Test Suites:

Tester Action and Input Data	Value to be found	Expected Outcome
Valid Partition:		
[1, 2, 3, 4, 5]	3	2

[5, 10, 15, 20, 25]	5	0
[2, 4, 6, 8]	5	-1
[1, 3, 5, 7]	4	-1
Boundary Value Analysis:		
[]	5	-1
[5]	5	0
[15]	5	-1
[5, 10, 15, 20, 25]	5	0
[5, 10, 15, 20, 25]	25	4
[2, 4, 6, 8]	2.2	Invalid Input
[2, 4, 6, 8]	a	Invalid Input
[1.1, c, 5, 7]	2	Invalid Input

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);
}
```

```
public void testCountItem() {
   CountItems counter = new CountItems();
   int[] arr1 = {1, 2, 3, 4, 5};
   int[] arr2 = {1, 2, 3, 4, 5, 6, 7, 8, 9};
   int[] arr3 = {1, 2, 3, 4, 4, 4, 5, 6, 7, 8, 9};
   int[] arr4 = {};
   int v1 = 3;
   int v2 = 10;
   assertEquals(1, counter.countItem(v1, arr1));
   assertEquals(0, counter.countItem(v2, arr1));
   assertEquals(9, counter.countItem(v1, arr2));
```

```
    assertEquals(0, counter.countItem(v2, arr2));
    assertEquals(1, counter.countItem(v1, arr3));
    assertEquals(0, counter.countItem(v2, arr3));
    assertEquals(0, counter.countItem(v2, arr4));
}
```

Equivalence Partitioning:

Tester Action and Input Data	Expected Outcome
v is present in a	Number of times v appears in a
v is not present in a	0

Boundary Value Analysis:

Tester Action and Input Data	Expected Outcome
Empty array a	0
v is present once in a	1
v is present multiple times in a	Number of times v appears in a
v is not present in a	0

Test Suites:

Tester Action and Input Data	Value to be found	Expected Outcome
Valid Partition:		
[1, 2, 3, 4, 5]	3	1
[5, 10, 15, 20, 25]	11	0

Boundary Value Analysis:		
[]	5	0
[5]	5	1
[15]	5	0
[5, 10, 15, 20, 25]	5	2
[2, 4, 6, 8]	2.2	Invalid Input
[2, 4, 6, 8]	a	Invalid Input
[1.1, c, 5, 7]	2	Invalid Input

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);
}
```

```
import static org.junit.Assert.assertEquals;
```

```
import org.junit.Test;


public class BinarySearchTest {
   @Test
   public void testBinarySearch() {
      BinarySearch bs = new BinarySearch();


      int[] arr1 = {1, 3, 5, 7, 9};
      assertEquals(0, bs.binarySearch(1, arr1)); // search for 1 in {1, 3,
5, 7, 9}
      assertEquals(2, bs.binarySearch(5, arr1)); // search for 5 in {1, 3,
5, 7, 9}
      assertEquals(4, bs.binarySearch(9, arr1)); // search for 9 in {1, 3,
5, 7, 9}
      assertEquals(-1, bs.binarySearch(4, arr1)); // search for 4 in {1, 3,
5, 7, 9}


      int[] arr2 = {2, 4, 6, 8, 10, 12};
      assertEquals(-1, bs.binarySearch(1, arr2)); // search for 1 in {2, 4,
6, 8, 10, 12}
      assertEquals(2, bs.binarySearch(6, arr2)); // search for 6 in {2, 4,
6, 8, 10, 12}
      assertEquals(5, bs.binarySearch(12, arr2)); // search for 12 in {2,
4, 6, 8, 10, 12}
      assertEquals(-1, bs.binarySearch(7, arr2)); // search for 7 in {2, 4,
6, 8, 10, 12}
```

```
    }
}
```

Equivalence Partitioning:

Tester Action and Input Data	Expected Outcome
v is present in a	Index of v
v is not present in a	-1

Boundary Value Analysis:

Tester Action and Input Data	Expected Outcome
Empty array a	-1
v is present at the first index of a	0
v is present at the last index of a length of a	a-1
v is not present in a	-1

Test Suites:

Tester Action and Input Data	Value to be found	Expected Outcome
Valid Partition:		
[1, 2, 3, 4, 5]	3	2
[5, 10, 15, 20, 25]	5	0
[2, 4, 6, 8]	5	-1
[1, 3, 5, 7]	4	-1
Boundary Value Analysis:		
[]	5	-1

[5]	5	0
[15]	5	-1
[1, 1, 1, 1, 1]	1	0,1,2,3,4
[5, 10, 15, 20, 25]	5	0
[5, 10, 15, 20, 25]	25	4
[2, 4, 6, 8]	2.2	Invalid Input
[2, 4, 6, 8]	a	Invalid Input
[1.1, c, 5, 7]	2	Invalid Input

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);

}


@Test

public void testEquilateral() {

  assertEquals("Equal", unittesting.triangle(3, 3, 3));
```

```
}

@Test
public void testIsosceles() {
  assertEquals("Isosceles", unittesting.triangle(5, 5, 6));

}

@Test
public void testScalene() {
  assertEquals("Scalene", unittesting.triangle(3, 4, 5));
}

@Test
public void testIncorrectInput() {
  assertEquals("Incorrect input", unittesting.triangle(1, 2, 3));

}
```

Equivalence Partitioning:

Tester Action and Input Data	Expected Outcome
Valid input: a=3, b=3, c=3	EQUILATERAL
Valid input: a=4, b=4, c=5	ISOSCELES
Valid input: a=5, b=4, c=3	SCALENE
InValid input: a=0, b=0, c=0	INVALID

InValid input: a=-1, b=2, c=3	INVALID
Valid input: a=1, b=1, c=1	EQUILATERAL
Valid input: a=2, b=2, c=1	ISOSCELES
Valid input: a=3, b=4, c=5	SCALENE
InValid input: a=0, b=1, c=1	INVALID
InValid input: a=1, b=0, c=1	INVALID
InValid input: a=1, b=1, c=0	INVALID

Boundary Value Analysis:

Tester Action and Input Data	Expected Outcome
Invalid inputs: a = 0, b = 0, c = 0	INVALID
Invalid inputs: a + b = c or b + c = a or c + a = b (a=3, b=4, c=8)	INVALID
Equilateral triangles: a = b = c = 1	EQUILATERAL
Equilateral triangles: a = b = c = 100	EQUILATERAL
Isosceles triangles: a = b ≠ c = 10	ISOSCELES
Isosceles triangles: a ≠ b = c = 10	ISOSCELES
Isosceles triangles: a = c ≠ b = 10	ISOSCELES
Scalene triangles: a = b + c - 1	SCALENE
Scalene triangles: b = a + c - 1	SCALENE
Scalene triangles: c = a + b - 1	SCALENE
Maximum values: a, b, c = Integer.MAX_VALUE	INVALID
Minimum values: a, b, c = Integer.MIN_VALUE	INVALID

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;
}

public void testPrefix() {
   String s1 = "hello";
   String s2 = "hello world";
   assertTrue(unittesting.prefix(s1, s2));
```

```
    s1 = "abc";
    s2 = "abcd";
    assertTrue(unittesting.prefix(s1, s2));

    s1 = "";
    s2 = "hello";
    assertTrue(unittesting.prefix(s1, s2));

    s1 = "hello";
    s2 = "hi";
    assertFalse(unittesting.prefix(s1, s2));

    s1 = "abc";
    s2 = "def";
    assertFalse(unittesting.prefix(s1, s2));
}
```

Equivalence Partitioning:

Tester Action and Input Data	Expected Outcome
Empty string s1 and s2	True
Empty string s1 and non-empty s2	True
Non-empty s1 is a prefix of non-empty s2	True

Non-empty s1 is not a prefix of s2	False
Non-empty s1 is longer than s2	False

Boundary Value Analysis:

Tester Action and Input Data	Expected Outcome
Empty string s1 and s2	True
Empty string s1 and non-empty s2	True
Non-empty s1 is not a prefix of s2	False
Non-empty s1 is longer than s2	False

Test Suites:

Tester Action and Input Data	Expected Outcome	Tester Action and Input Data
Equivalence Partitioning		Equivalence Partitioning
s1= "abcd",s2 = "abcd"	true	s1= "abcd",s2 = "abcd"
s1 = "",s2 = ""	true	s1 = "",s2 = ""
s1 = "ha",s2 = "hasrh"	true	s1 = "ha",s2 = "hasrh"
s1 = "hcp",s2 = "hc"	false	s1 = "hcp",s2 = "hc"
s1 = "abc",s2 = ""	false	s1 = "abc",s2 = ""
s1 = "",s2 = "abc"	true	s1 = "",s2 = "abc"
s1 = "o",s2 = "ott"	true	s1 = "o",s2 = "ott"
s1 = "abc",s2 = "def"	false	s1 = "abc",s2 = "def"

s1 = "deg",s2 = "def"	false	s1 = "deg",s2 = "def"
Boundary value analysis		Boundary value analysis
s1= "abcd",s2 = "abcd"	true	s1= "abcd",s2 = "abcd"
s1= "",s2 = ""	true	s1= "",s2 = ""
s1= "abcd",s2 = ""	false	s1= "abcd",s2 = ""

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

The following are the equivalence classes for different types of triangles

Invalid case:

E1 : a+b<=c

E2 : a+c<=b

E3: b+c <=a

Equilateral case:

E4 : a=b,b=c,c=a

Isosceles case:

E5 : a=b , a!=c

E6: a= c, a!=b

E7: b=c, b!=a

Scalene case:

E8 : a!=b ,b!=c, c!=a

Right-angled triangle case:

E9 : a^2 + b^2 = c^2

E10: b^2+c^2 =a^2

E11: a^2 +c^2=b^2

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)

Test case Output Equivalent class covered

a=1.5, b=2.6 , c=4.1 Not a triangle E1

a = -1.6, b=5, c=6 Not a triangle E2

a=7.1, b=6.1, c=1 Not a triangle E3

a=5.5, b= 5.5, c=5.5 Equilateral E4

a=4.5, b=4.5, c=5 isosceles E5

a=6, b=4, c=6 isosceles E6

a=8, b=5, c=5 isosceles E7

a=6,b=7,c=8 scalene E8

a=3,b=4,c=5 Right-angled triangle E9

a=0.13,b=0.12,c=0.05 Right-angled triangle E10

a=7,b=25,c=23 Right-angled triangle E11

All of the equivalent classes are covered with the above test cases

c) For the boundary condition A + B > C case (scalene triangle), identify test cases

to verify the boundary.

Test cases to verify the boundary condition:

1) a=5 b=5 c=9 (a+b=c)

2) a=5.5 b=5.5 c=10.9 (a+b just greater than c)

3) a=5.5 b=5 c=9.6 (a+b just less than c)

d) For the boundary condition A = C case (isosceles triangle), identify test cases to

verify the boundary.

Test cases to verify the boundary condition:

1) a=5 b=5 c=5 (a=c)

2) a=5.5 b=5.5 c=5.6 (a just less than c)

3) a=5.5 b=5 c=5.4 (a just greater than c)

e) For the boundary condition A = B = C case (equilateral triangle), identify test

cases to verify the boundary.

Test cases to verify the boundary condition:

1) a=5 b=5 c=5 (a=b=c)

2) a=10 b=10 c=9 (a= b but a≠c)

3) a=10 b=11 c=10 (a=c but a≠b)

f) For the boundary condition A2 + B2 = C2 case (right-angle triangle), identify test

cases to verify the boundary.

Test cases to verify the boundary condition:

1) a=3, b=4, c=5 (a2+b2=c2)

2) a=0.12, b=0.5 , c=0.14 (a2+b2 just less than c2)

3) a=7 , b=23 , c=24 (a2+b2 just greater than c2)

g) For the non-triangle case, identify test cases to explore the boundary.

Test cases to verify the boundary condition:

1) a=1,b=2,c=3

2) a=5,b=5,c=10

3) a=0,b=0,c=0

h) For non-positive input, identify test points.

Test points for non-positive input:

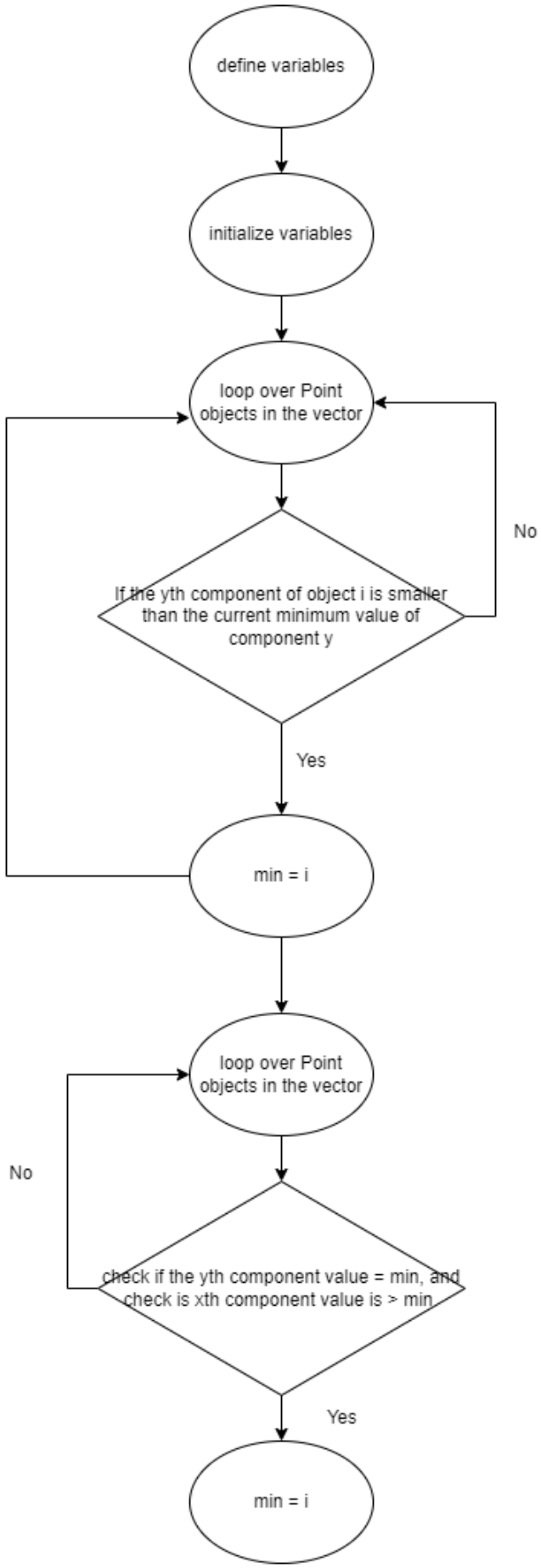
1) a=-4.0 b=3.2 c=4.5

2) a=5,b=-4.2,c=-3.2

3) a=4, b=5, c=-10

Section – B

1. Control Flow Diagram


define variables
initialize variables
loop over Point objects in the vector
If the yth component of object i is smaller than the current minimum value of component y
No
Yes
min = i
loop over Point objects in the vector
No
check if the yth component value = min, and check is xth component value is > min
Yes
min = i

2. Test sets

Statement coverage test sets: To achieve statement coverage, we need to make sure that every statement in the code is executed at least once.
Test 1: p = empty vector
Test 2: p = vector with one point
Test 3: p = vector with two points with the same y component
Test 4: p = vector with two points with different y components
Test 5: p = vector with three or more points with different y components
Test 6: p = vector with three or more points with the same y component

Branch coverage test sets: To achieve branch coverage, we need to make sure that every possible branch in the code is taken at least once
Test 1: p = empty vector
Test 2: p = vector with one point
Test 3: p = vector with two points with the same y component
Test 4: p = vector with two points with different y components
Test 5: p = vector with three or more points with different y components, and none of them have the same x component
Test 6: p = vector with three or more points with the same y component, and some of them have the same x component
Test 7: p = vector with three or more points with the same y component, and all of them have the same x component

Basic condition coverage test sets: To achieve basic condition coverage, we need to make sure that every basic condition in the code (i.e., every Boolean subexpression) is evaluated as both true and false at least once
Test 1: p = empty vector

Test 2: p = vector with one point
Test 3: p = vector with two points with the same y component, and the first point has a smaller x component
Test 4: p = vector with two points with the same y component, and the second point has a smaller x component
Test 5: p = vector with two points with different y components
Test 6: p = vector with three or more points with different y components, and none of them have the same x component
Test 7: p = vector with three or more points with the same y component, and some of them have the same x component
Test 8: p = vector with three or more points with the same y component, and all of them have the same x component