IT314
LAB - 7

202001268
Fenil M

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.
Enlist which set of test cases have been identified using Equivalence Partitioning and Boundary Value Analysis separately.

Equivalence Class :

Day:

ID	**Class**	**Validity**
E1	date<1	Invalid
E2	1<=date<=28	Valid
E3	date=31	Valid
E4	date=29	Valid
E5	date=30	Valid
E6	date > 31	Invalid

Month:

ID	Class	Validity
E7	month<1	Invalid
E8	month = 1,3,5,7,8,10,12(months with 31 days)	Valid
E9	month = 4,6,9,11(months with 30 days)	Valid
E10	month = 2(month with either 28 or 29 days)	Valid
E11	month > 12	Invalid

Year:

ID	Class	Validity
E12	year<1900	Invalid
E13	leap year 1900<=year<=2015	Valid
E14	non leap year 1900<=year<=2015	Valid
E15	year > 2015	Invalid

Test Cases:

ID	Day	Month	Year	Expected Output
1	15	5	2005	Previous date
2	25	2	2007	Previous date
3	29	2	2003	Invalid
4	31	4	2012	Invalid
5	1	3	2000	Previous date
6	30	4	2016	Invalid
7	1	1	2000	Previous date
8	1	1	1900	Invalid
9	31	12	2015	Previous date
10	5	15	2001	Invalid

Boundary Value Analysis: Using boundary value analysis, we can identify the following boundary test cases:

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 these boundary test cases, we can design the 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

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.

Code:

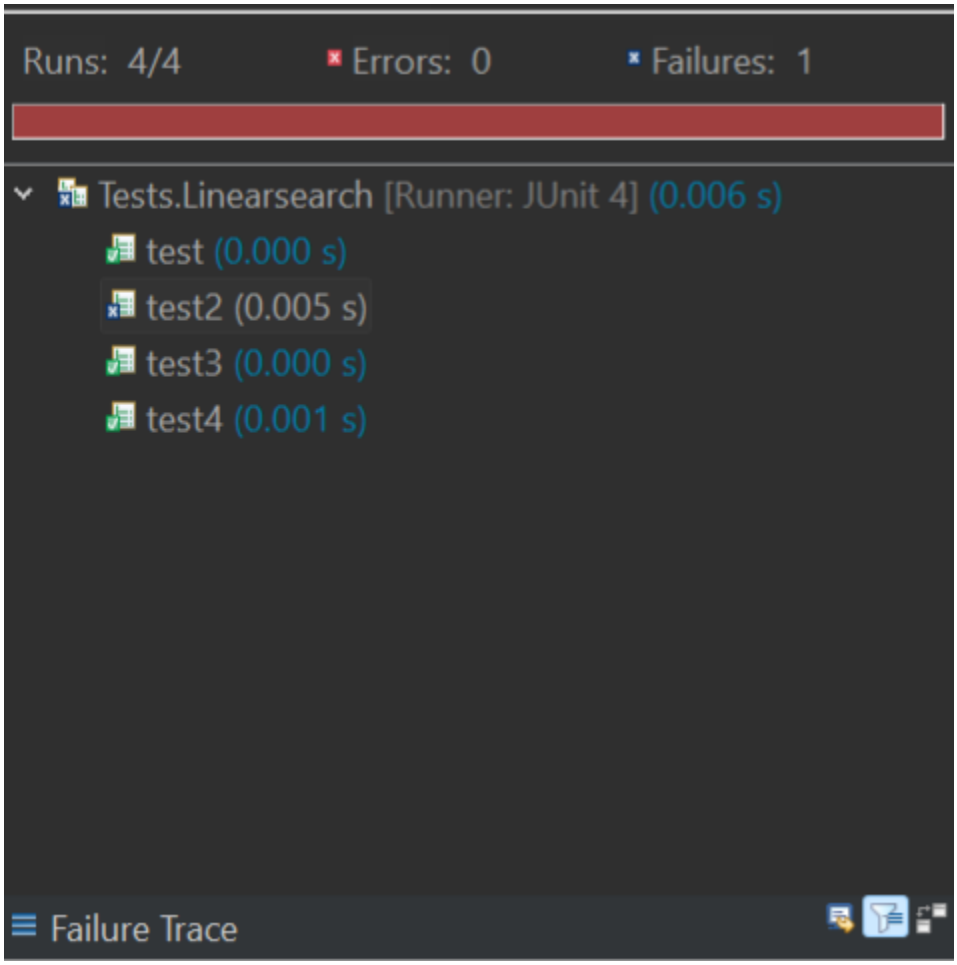
```
int linearSearch(int v, int a[])

{

int i = 0;

while (i < a.length)

{

if (a[i] == v)

return(i);

i++;

}

return (-1);

}
```

Testing Code:

```
package Tests;

import static org.junit.Assert.*;

import org.junit.Test;

public class Linearsearch {

  @Test

  public void test() {
```

```
  UnitTesting obj = new UnitTesting();
  int[] arr1 = { 2, 4, 6, 8, 10 };
  assertEquals(0, obj.linearSearch(2, arr1));
  assertEquals(4, obj.linearSearch(10, arr1));
}
@Test
public void test2() {
  UnitTesting obj = new UnitTesting();
  int[] arr2 = { -3, 0, 3, 7, 11 };
  assertEquals(-1, obj.linearSearch(3, arr2));
}
@Test
public void test3() {
  UnitTesting obj = new UnitTesting();
  int[] arr3 = { 1, 3, 5, 7, 9 };
  assertEquals(4, obj.linearSearch(9, arr3));
}
@Test
public void test4() {
  UnitTesting obj = new UnitTesting();
  int[] arr4 = {};
  assertEquals(-1, obj.linearSearch(2, arr4));
}
```

```
}
```



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

Code :

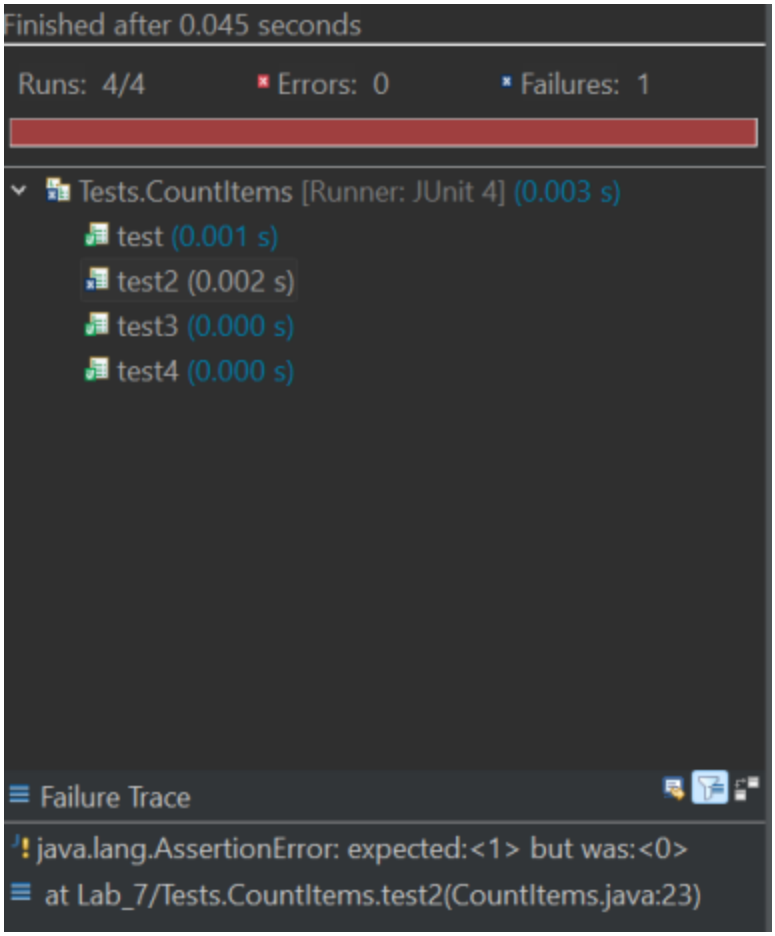
```
int countItem(int v, int a[]) {
    int count = 0;
    for (int i = 0; i < a.length; i++) {
      if (a[i] == v) count++;
```

```
    }
    return (count);
  }
```

Testing Code:

```
package Tests;
import static org.junit.Assert.*;
import org.junit.Test;
public class CountItems {
  @Test
  public void test() {
    UnitTesting obj = new UnitTesting();
    int[] arr1 = { 1, 2, 3, 4, 5 };
    int v1 = 3;
    int v2 = 10;
    assertEquals(1, obj.countItem(v1, arr1));
  }
  @Test
  public void test2() {
    UnitTesting obj = new UnitTesting();
    int[] arr2 = { 1, 2, 3, 4, 5, 6, 7, 8, 9 };
    int v1 = 3;
    int v2 = 10;
    assertEquals(1, obj.countItem(v2, arr2));
```

```
  }

  @Test
  public void test3() {
    UnitTesting obj = new UnitTesting();
    int[] arr3 = { 1, 2, 3, 4, 4, 4, 5, 6, 7, 8, 9 };
    int v1 = 3;
    int v2 = 10;
    assertEquals(1, obj.countItem(v1, arr3));
  }
  @Test
  public void test4() {
    UnitTesting obj = new UnitTesting();
    int[] arr4 = {};
    int v1 = 3;
    int v2 = 10;
    assertEquals(0, obj.countItem(v2, arr4));
  }
}
```



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.

Code :

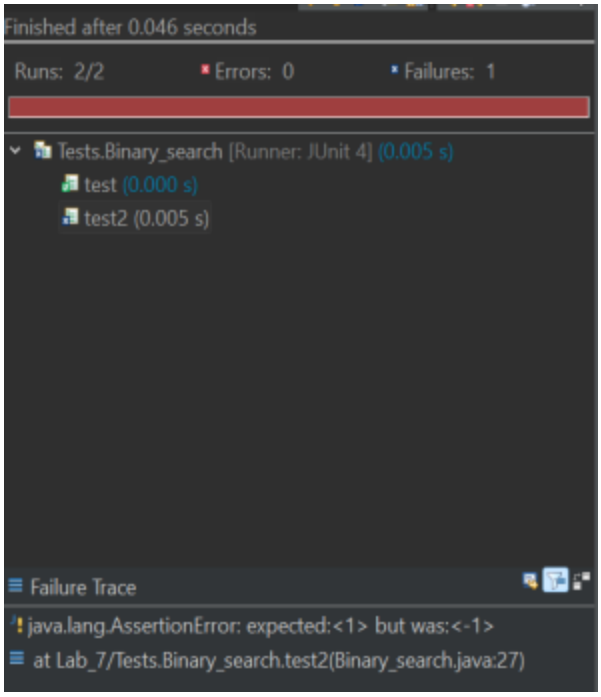
```
int binarySearch(int v, int a[]) {

    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);
  }
```

Testing code:

```
package Tests;
import static org.junit.Assert.*;
import org.junit.Test;
public class Binary_search {
@Test
public void test() {
 UnitTesting obj = new UnitTesting();
 int[] arr1 = {1, 3, 5, 7, 9};
 assertEquals(0, obj.binarySearch(1, arr1)); // search
for 1 in {1, 3, 5, 7, 9}
 assertEquals(2, obj.binarySearch(5, arr1)); // search
for 5 in {1, 3, 5, 7, 9}
 assertEquals(4, obj.binarySearch(9, arr1)); // search
for 9 in {1, 3, 5, 7, 9}
 assertEquals(-1, obj.binarySearch(4, arr1)); // search
for 4 in {1, 3, 5, 7, 9}
```

```
}

@Test

public void test2() {

 UnitTesting obj = new UnitTesting();

 int[] arr2 = {2, 4, 6, 8, 10, 12};

 assertEquals(-1, obj.binarySearch(1, arr2)); // search

for 1 in {2, 4, 6, 8, 10, 12}

 assertEquals(2, obj.binarySearch(6, arr2)); // search

for 6 in {2, 4, 6, 8, 10, 12}

 assertEquals(5, obj.binarySearch(12, arr2)); // search

for 12 in {2, 4, 6, 8, 10, 12}

 assertEquals(1, obj.binarySearch(7, arr2)); // search

for 7 in {2, 4, 6, 8, 10, 12}

}

}
```



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).

Code :

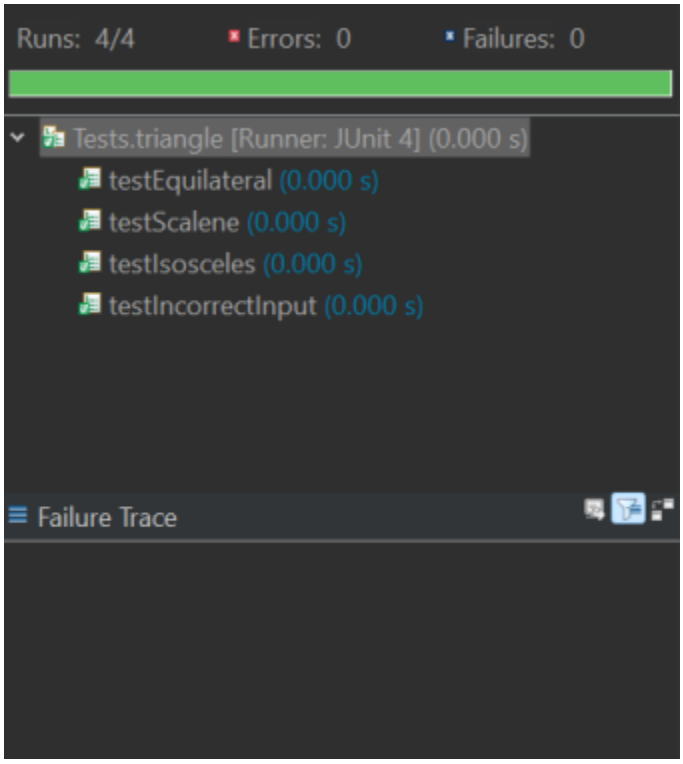
```
final int EQUILATERAL = 0;

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);
  }
```

Testing Code:

```
package Tests;

import static org.junit.Assert.*;

import org.junit.Test;

public class triangle {

  @Test
  public void testEquilateral() {
    UnitTesting obj = new UnitTesting();
    assertEquals(0, obj.triangle(3, 3, 3));
  }

  @Test
  public void testIsosceles() {
    UnitTesting obj = new UnitTesting();
    assertEquals(1, obj.triangle(5, 5, 6));
  }

  @Test
  public void testScalene() {
    UnitTesting obj = new UnitTesting();
    assertEquals(2, obj.triangle(3, 4, 5));
  }

  @Test
```

```
  public void testIncorrectInput() {
    UnitTesting obj = new UnitTesting();
    assertEquals(3, obj.triangle(1, 2, 3));
  }
}
```



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).

Code :

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

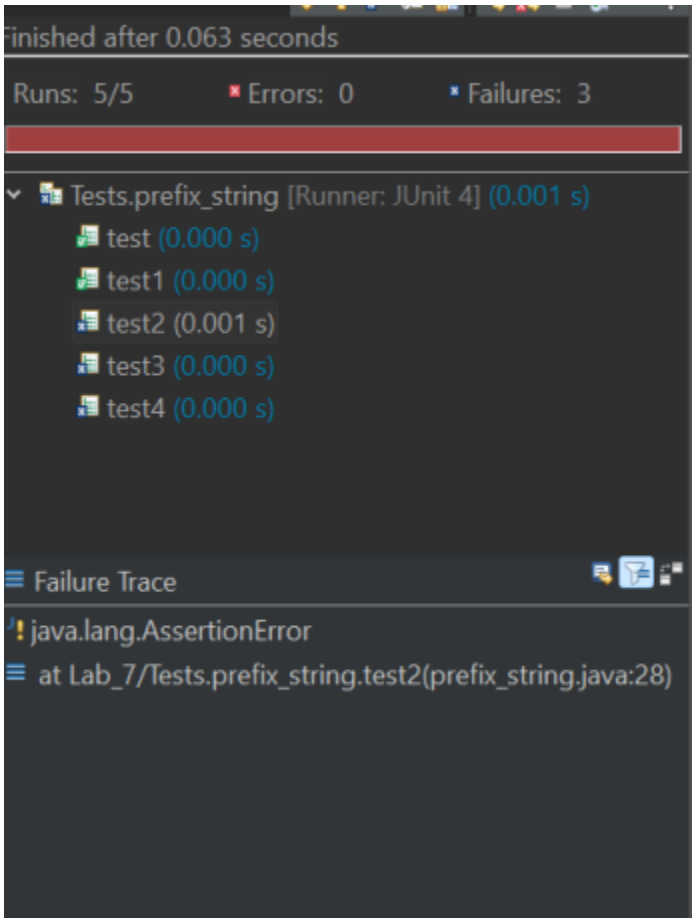
```
      if (s1.charAt(i) != s2.charAt(i)) {
        return false;
      }
    }
    return true;
  }
```

Testing Code:

```
package Tests;
import static org.junit.Assert.*;
import org.junit.Test;
public class prefix_string {
  @Test
  public void test() {
    UnitTesting obj = new UnitTesting();
    String s1 = "hello";
    String s2 = "hello world";
    assertTrue(UnitTesting.prefix(s1, s2));
  }
  @Test
  public void test1() {
    UnitTesting obj = new UnitTesting();
    String s1 = "abc";
    String s2 = "abcd";
```

```
  assertTrue(UnitTesting.prefix(s1, s2));
}
@Test
public void test2() {
  UnitTesting obj = new UnitTesting();
  String s1 = "hello";
  String s2 = "";
  assertTrue(UnitTesting.prefix(s1, s2));
}
@Test
public void test3() {
  UnitTesting obj = new UnitTesting();
  String s1 = "hello";
  String s2 = "hi";
  assertTrue(UnitTesting.prefix(s1, s2));
}
@Test
public void test4() {
  UnitTesting obj = new UnitTesting();
  String s1 = "abc";
  String s2 = "def";
  assertTrue(UnitTesting.prefix(s1, 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

a) Identify the equivalence classes for the system Equivalence classes:

EC1: Invalid inputs (negative or zero values)

EC2: Non-triangle (sum of the two shorter sides is not greater than the longest side)

EC3: Scalene triangle (no sides are equal)

EC4: Isosceles triangle (two sides are equal)

EC5: Equilateral triangle (all sides are equal)

EC6: Right-angled triangle (satisfies the Pythagorean theorem)

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 cases:

TC1: -1, 0

TC2: 1, 2, 5

TC3: 3, 4, 5

TC4: 5, 5, 7

TC5: 6, 6, 6

TC6: 3, 4, 5

Test case 1 covers class 1, test case 2 covers class 2, test case 3 covers class 3, test

case 4 covers class 4, test case 5 covers class 5, and test case 6 covers class 6

c) For the boundary condition A + B > C case (scalene triangle), Identify test cases to verify the boundary.

2, 3, 6

3, 4, 8

Both test cases have two sides that are shorter than the third side, and should not form a triangle

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

1, 2, 1

0, 2, 0

5, 6, 5

Both test cases have two sides that are equal, but only test case 2 should form an isosceles triangle, other input are invalid.

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

5, 5, 5

0, 0, 0

Both test cases have all sides equal, but only test case 1 should form an equilateral triangle, other input are invalid.

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

3, 4, 5 0, 0, 0 -3, -4, -5 Both test cases satisfy the Pythagorean theorem, but only test

case 1 should form right-angled triangle, other input are invalid. triangle

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

Test cases for the non-triangle case:

TC11 (EC3): A=2, B=2, C=4 (sum of A and B is less than C)

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

Test points for non-positive input:

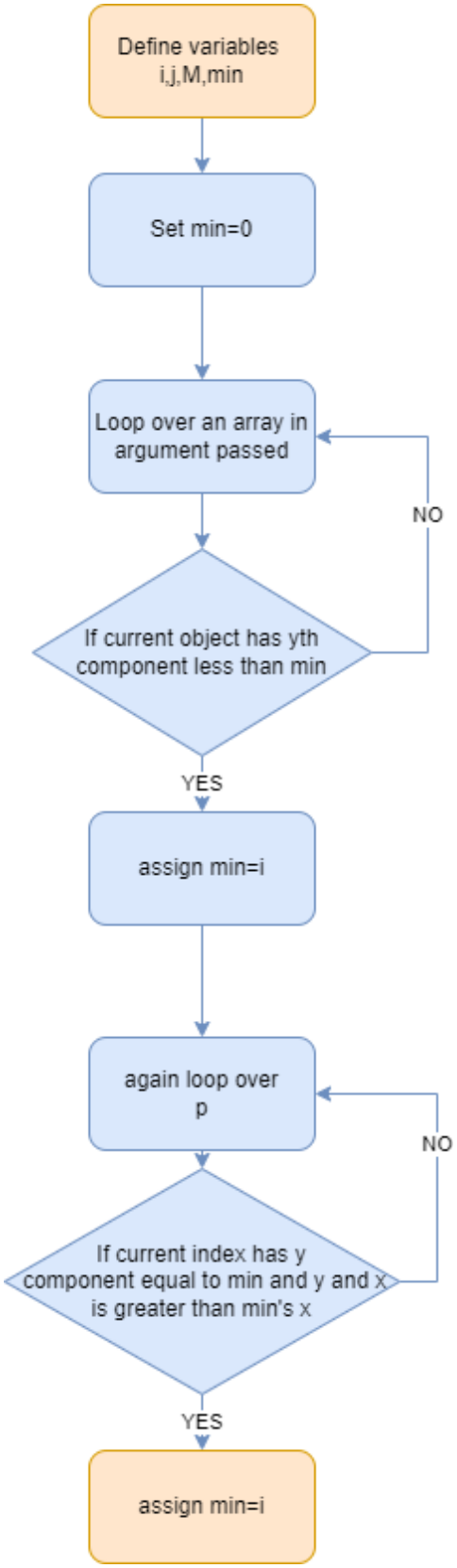
TP1 (EC2): A=0, B=4, C=5 (invalid input)

TP2 (EC2): A=-2, B=4, C=5 (invalid input)

[Test cases TC1 to TC10 covers all identified equivalence classes.]

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.


Define variables i,j,M,min
Set min=0
Loop over an array in argument passed
NO
If current object has yth component less than min
YES
assign min=i
again loop over p
NO
If current index has y component equal to min and y and x is greater than min's x
YES
assign min=i

2. Construct test sets for your flow graph that are adequate for the following criteria:

A. Statement Coverage :

1 p is empty array

2 p has one point object

3 p has two points object with different y component

4 p has two points object with different x component

5 p has three or more point object with different y component

B. Branch Coverage test set [In this all branch are taken atleast once] :

1 p is empty array

2 p has one point object

3 p has two points object with different y component

4 p has two points object with different x component

5 p has three or more point object with different y component

6 p has three or more point object with same y component

7 p has three or more point object with all same x component

8 p has three or more point object with all different x component

9 p has three or more point object with some same and some different x component

C. Basic condition coverage test set [Each boolean expression has been evaluated to both true and false] :

1 p is empty array

2 p has one point object

3 p has two points object with different y component

4 p has two points object with different x component

5 p has three or more point object with different y component

6 p has three or more point object with same y component

7 p has three or more point object with all same x component

8 p has three or more point object with all different x component

9 p has three or more point object with some same and some different x component

10 p has three or more point object with some same and some different y component

11 p has three or more point object with all different y component

12 p has three or more point object with all same y component