

Course - IT314
Lab - 7
Name -Kunj Gupta
ID - 202001269

Section A:

Q. Consider a program for determining the previous date. Its input is triple of day, month and year with the following ranges $1 \leq \text{month} \leq 12$, $1 \leq \text{day} \leq 31$, $1900 \leq \text{year} \leq 2015$. The possible output dates would be previous date or invalid date. Design the equivalence class test cases?

-> To design the equivalence class test cases for the program, we need to consider the input ranges and identify the valid and invalid equivalence classes.

Here are the equivalence classes for the input parameters:

Valid equivalence classes:

Valid day, month, and year

Valid day, month, and minimum year (1900)

Valid day, month, and maximum year (2015)

Invalid equivalence classes:

Invalid day, month, and year (e.g., day 0, day 32, month 0, month 13, year < 1900 or year > 2015)

Invalid day for a given month and year (e.g., Feb 29 in a non-leap year, Apr 31, Jun 31, Sep 31, Nov 31)

Based on these equivalence classes, here are the test cases that should be considered:

Valid equivalence class test cases:

Test case 1: Valid day, month, and year (e.g., 15, 7, 2005)

Test case 2: Valid day, month, and minimum year (e.g., 1, 1, 1900)

Test case 3: Valid day, month, and maximum year (e.g., 31, 12, 2015)

Invalid equivalence class test cases:

Test case 4: Invalid day, month, and year (e.g., 0, 0, 1899)

Test case 5: Invalid day, month, and year (e.g., 32, 13, 2016)

Test case 6: Invalid day for a given month and year (e.g., Feb 29 in a non-leap year, such as 29, 2, 2001)

Test case 7: Invalid day for a given month and year (e.g., Apr 31, such as 31, 4, 2005)
 Test case 8: Invalid day for a given month and year (e.g., Jun 31, such as 31, 6, 2005)
 Test case 9: Invalid day for a given month and year (e.g., Sep 31, such as 31, 9, 2005)
 Test case 10: Invalid day for a given month and year (e.g., Nov 31, such as 31, 11, 2005)

The above test cases cover all the equivalence classes for the input parameters and should be sufficient to test the program for determining the previous date.

Test Case ID	Day	Month	Year	Expected Output
1	1	6	2000	31-5-2000
2	2	6	2015	1-6-2015
3	2	6	2016	Invalid
4	1	1	1900	29-2-2012
5	31	12	1899	Invalid
6	31	12	1900	30-12-1900
7	29	2	2012	28-2-2012
8	1	3	2012	29-2-2012
9	29	2	2011	Invalid
10	30	2	2020	Invalid

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);
}
```

Equivalence Partitioning:

If a is empty, an error message should be returned.

If v is not in a, -1 should be returned.

If v is the first element of a, the function should return 0.

If v is in the middle of a, the function should return the first index i such that a[i] == v, where i is greater than 0 and less than a.length - 1.

If v is the last element of a, the function should return a.length - 1.

Test Case		
a[]=0, 0, 0, 0, 0, 0	v=0	y(output)=0
a[]=0, 1, 2, 3, 4, 5	v=6	y = -1
a[]=2, 4, 6, 8, 10	v=2	y = 0
a[]=1, 3, 5, 7, 9	v=2	y = -1
a[]=2, 4, 6, 8, 10	v=11	y = -1
a[]=1, 2, 3, 4, 5, 6	v=6	y = 5

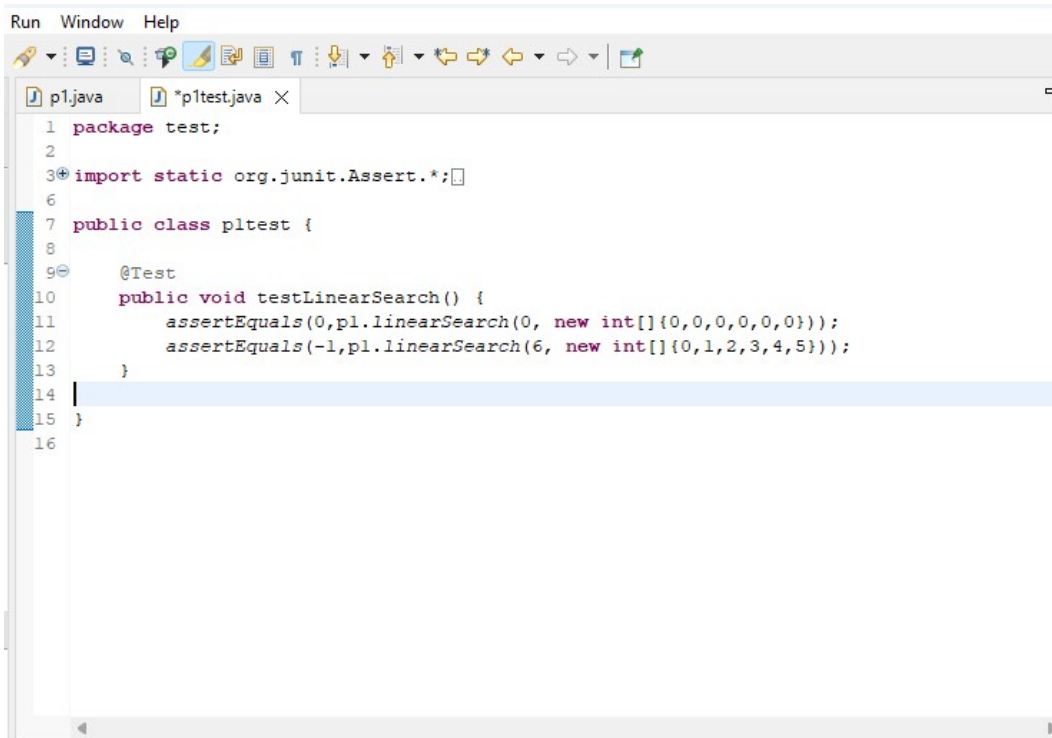
Boundary Value Analysis:

If a has one element and it's not v, -1 should be returned.

If a has one element and it's v, the function should return 0.

If a has two elements and v is the first element, the function should return 0.

If a has two elements and v is the second element, the function should return 1.
If a has n elements and v is the first element, the function should return 0.
If a has n elements and v is the last element, the function should return n-1.
If a has n elements and v is not in a, -1 should be returned.



```
1 package test;
2
3 import static org.junit.Assert.*;
4
5
6
7 public class p1test {
8
9     @Test
10     public void testLinearSearch() {
11         assertEquals(0, p1.linearSearch(0, new int[]{0,0,0,0,0,0}));
12         assertEquals(-1, p1.linearSearch(6, new int[]{0,1,2,3,4,5}));
13     }
14
15 }
16
```

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);
}
```

Code:

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

Equivalence Partitioning:

If a is empty, the function should return 0.

If v is not in a, the function should return 0.

If v appears once in a, the function should return 1.

If v appears multiple times in a, the function should return the number of occurrences of v.

Test Case		
a[]=0, 0, 0, 0, 0, 0	v=0	y(output)=6
a[]=0, 1, 2, 3, 4, 5	v=6	y = 0
a[]=1,6,3,4,8,1	v=1	y = 2
a[]=NULL	v=98	y = 1
a[]=265,41,60,80,100	v=100	y = 1
a[]=-89,-89	v=-89	y = 2

Boundary Value Analysis:

If a has one element and it's not v, the function should return 0.

If a has one element and it's v, the function should return 1.

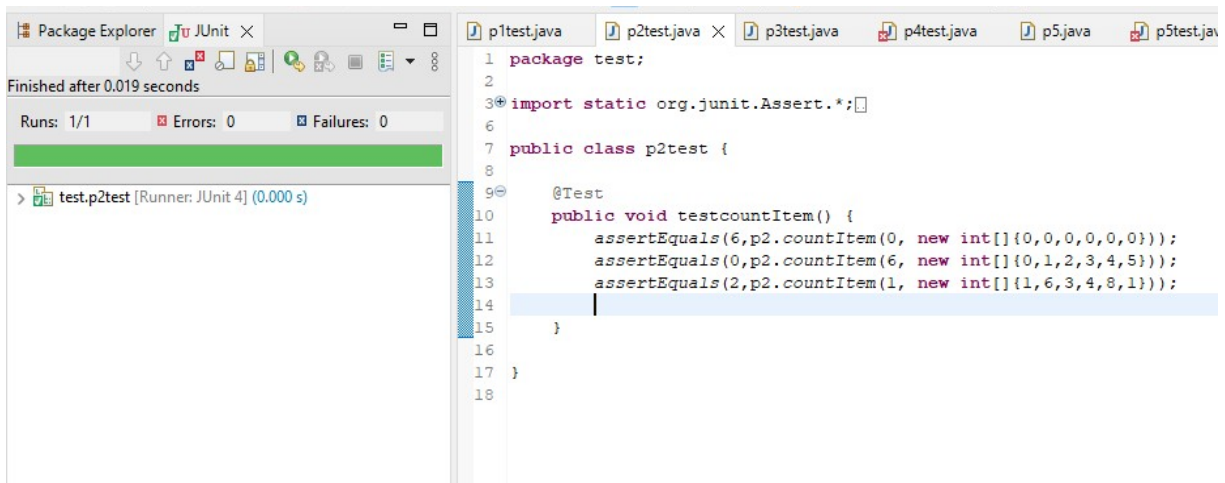
If a has two elements and v is the first element, the function should return 1.

If a has two elements and v is the second element, the function should return 1.

If a has n elements and v appears once, the function should return 1.

If a has n elements and v appears multiple times, the function should return the number of occurrences of v.

If a has n elements and v is not in a, the function should return 0.



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);
}

```

Equivalence Partitioning:

If `a` is empty, the function should return `-1`.

If `v` is not in `a`, the function should return `-1`.

If `v` is the first element in `a`, the function should return `0`.

If v is the last element in a, the function should return a.length-1.

If v appears once in a, the function should return the index of v.

If v appears multiple times in a, the function should return the index of the first occurrence of v

Test Case		
a[]={0, 0, 0, 0, 0, 0}	v=0	y(output)=0
a[]={0, 1, 2, 3, 4, 5}	v=6	y = -1
a[]={10,100,1000,10000}	v=100000	y = -1
a[]={NULL}	v=168	y = -1
a[]={-100, -90, -80, 100, 1000}	v=10000	y = 4
a[]={1,3}	v=3	y = 1

Boundary Value Analysis:

If a has one element and it's not v, the function should return -1.

If a has one element and it's v, the function should return 0.

If a has two elements and v is the first element, the function should return 0.

If a has two elements and v is the second element, the function should return 1.

If a has n elements and v is the first element, the function should return 0.

If a has n elements and v is the last element, the function should return n-1.

If a has n elements and v appears once, the function should return the index of v.

If a has n elements and v appears multiple times, the function should return the index of the first occurrence of v.

If a has n elements and v is not in a, the function should return -1.

```
1 package test;
2
3 import static org.junit.Assert.*;
4
5
6
7 public class p3test {
8
9     @Test
10     public void testbinarySearch() {
11         assertEquals(2,p3.binarySearch(0, new int[]{0,0,0,0,0,0}));
12         assertEquals(-1,p3.binarySearch(6, new int[]{0,1,2,3,4,5}));
13     }
14
15 }
```

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);
}
```

Equivalence Partitioning:

Equilateral triangle (a=a, b=a, c=a): expected outcome is EQUILATERAL (0)

Isosceles triangle (a=a, b=b, c=c): expected outcome is ISOSCELES (1)

Scalene triangle (a=b, b=c, c=a): expected outcome is SCALENE (2)

Invalid triangle (a=b+c): expected outcome is INVALID (3)

Invalid triangle (a=b-c): expected outcome is INVALID (3)

Invalid triangle (a=b+c-1): expected outcome is INVALID (3)

Invalid triangle (a=-1, b=-1, c=-1): expected outcome is INVALID (3)

Invalid triangle (a=0, b=0, c=0): expected outcome is INVALID (3)

Invalid triangle (a=1, b=2, c=4): expected outcome is INVALID (3)

Test Case	Expected Outcome
a=2, b=2, c=2	Equilateral
a=100, b=100, c=100	Equilateral
a=2, b=2, c=3	Isosceles

a=1, b=3, c=3	Isosceles
a=3, b=4, c=5	Scalene
a=5, b=12, c=13	Scalene
a=0, b=0, c=0	Invalid
a=0, b=0, c=1	Invalid
a=0, b=1, c=2	Invalid
a=-1, b=1, c=1	Invalid
a=-1, b=-2, c=-2	Invalid
a=-1, b=1, c=0	Invalid

Package Explorer | Run p4test.testtriangle (already running) | p4.java | p4test.java | p5test.java

Finished after 0.018 seconds

Runs: 1/1 | Errors: 0 | Failures: 0

testtriangle [Runner: JUnit 4] (0.000 s)

```

@Test
public void testtriangle() {
    assertEquals(0,p4.triangle(1,1,1));
    assertEquals(0,p4.triangle(2,2,2));
    assertEquals(0,p4.triangle(100,100,100));
    assertEquals(1,p4.triangle(2,2,3));
    assertEquals(1,p4.triangle(1,4,4));
    assertEquals(1,p4.triangle(6,2,6));
    assertEquals(1,p4.triangle(4,4,7));
    assertEquals(2,p4.triangle(3,4,5));
    assertEquals(2,p4.triangle(5,12,13));
    assertEquals(3,p4.triangle(0,0,0));
    assertEquals(3,p4.triangle(0,0,1));
    assertEquals(3,p4.triangle(0,1,1));
    assertEquals(3,p4.triangle(-1,-2,-1));
    assertEquals(3,p4.triangle(-1,0,-1));
    assertEquals(3,p4.triangle(4,8,4));
    assertEquals(3,p4.triangle(-1,-1,-1));
    assertEquals(3,p4.triangle(1,1,-1));
    assertEquals(3,p4.triangle(0,1,2));
    assertEquals(3,p4.triangle(-1,-2,-3));
    assertEquals(3,p4.triangle(2,3,9));
    assertEquals(3,p4.triangle(0,4,7));
}

```

Failure Trace

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;
}
```

Equivalence Partitioning:

`s1` and `s2` are both empty strings: false

`s1` is empty and `s2` is non-empty: true

`s1` is non-empty and `s2` is empty: false

`s1` is a proper prefix of `s2`: true

`s1` is not a prefix of `s2`: false

`s1` and `s2` are equal: true

Test Case	Expected Outcome
<code>s1=hello s2=helloworld</code>	<code>y=true</code>
<code>s1=hello s2=Hello</code>	<code>y=false</code>
<code>s1=hello s2=helloO</code>	<code>y=false</code>
<code>s1=hello s2=helloWorld</code>	<code>y=true</code>
<code>s1=hello s2=heello</code>	<code>y=false</code>
<code>s1=hello s2=heLloworld</code>	<code>y=false</code>
<code>s1=hello s2=hell</code>	<code>y=false</code>

s1=o s2=ott	y=true
s1=abc s2=defl	y=false

Boundary Value Analysis:

s1 is one character shorter than s2: true

s1 is one character longer than s2: false

s1 and s2 have the same length: true

```

1 package test;
2
3 import static org.junit.Assert.*;
4
5
6
7 public class p5test {
8
9     private String s1;
10    private String s2;
11
12    @Test
13    public void testprefix() {
14        assertEquals(true, p5.prefix("hello", "helloworld"));
15        assertEquals(false, p5.prefix("hello", "Hello"));
16        assertEquals(false, p5.prefix("hello", "hello"));
17        assertEquals(true, p5.prefix("hello", "helloworld"));
18        assertEquals(false, p5.prefix("hello", "heello"));
19        assertEquals(false, p5.prefix("hello", "helloworld"));
20        assertEquals(false, p5.prefix("hello", "hell"));
21    }
22
23 }

```

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.

a) Equivalence classes:

A, B, and C form a valid triangle

A, B, and C do not form a valid triangle

Class ID	Class
E1	All sides are positive
E2	two of its sides are zero

E3	One of its sides are negative
E4	Sum of two sides is less than third side
E5	Any of the side/sides is negative

b) Test cases:

A=4, B=4, C=4 (Equilateral triangle)

A=4, B=4, C=5 (Isosceles triangle)

A=4, B=5, C=6 (Scalene triangle)

A=3, B=4, C=5 (Right-angle triangle)

A=1, B=2, C=3 (Does not form a valid triangle)

Test Case ID	Class ID	Test Case
T1	E1	A = 1,B = 1,C = 1
T2	E1	A = 3, B = 4, C= 5
T3	E2	A = 0,B = 0,C = 1
T4	E3	A = 0,B = 1,C = 2
T5	E4	A = 1, B = 3, C = 8
T6	E5	A = -1,C = 1,D = 5

c) Test cases for boundary condition $A+B>C$:

A=0.1, B=0.2, C=0.3 (Smallest valid scalene triangle)

A=0.1, B=0.1, C=0.2 (Smallest invalid triangle)

d) Test cases for boundary condition

A=C: A=3, B=4, C=3 (Isosceles triangle with equal sides A and C)

A=0.1, B=0.2, C=0.1 (Smallest isosceles triangle with equal sides A and C)

A=1, B=2, C=1 (Smallest invalid triangle with equal sides A and C)

e) Test cases for boundary condition $A=B=C$:

A=5, B=5, C=5 (Equilateral triangle)

A=0.1, B=0.1, C=0.1 (Smallest equilateral triangle)

A=1, B=2, C=3 (Smallest invalid triangle with equal sides A, B, and C)

f) Test cases for boundary condition $A^2 + B^2 = C^2$:

A=3, B=4, C=5 (Right-angle triangle)

A=0.1, B=0.2, C=0.22361 (Smallest right-angle triangle)

A=1, B=1, C=1.41421 (Smallest invalid right-angle triangle)

g) Test cases for non-triangle case:

A=1, B=2, C=10 ($A + B < C$)

A=1, B=10, C=2 ($A + C < B$)

A=10, B=1, C=2 ($B + C < A$)

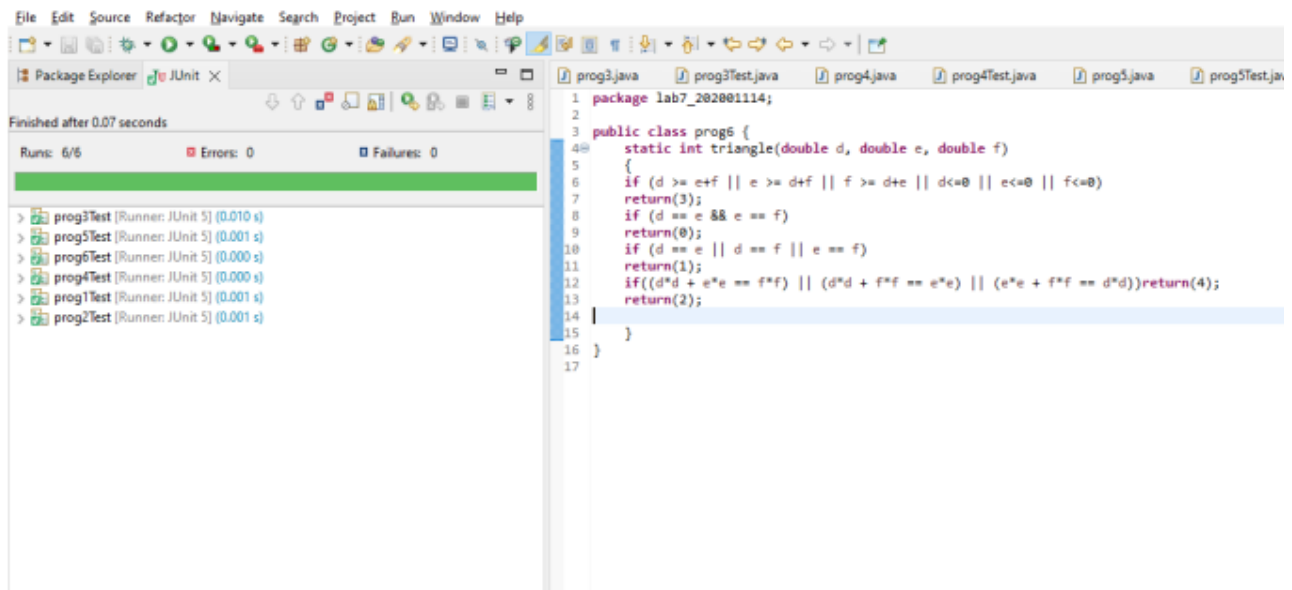
h) Test cases for non-positive input:

A=-1, B=2, C=3

A=1, B=-2, C=3

A=1, B=2, C=-3

A=-1, B=-2, C=-3

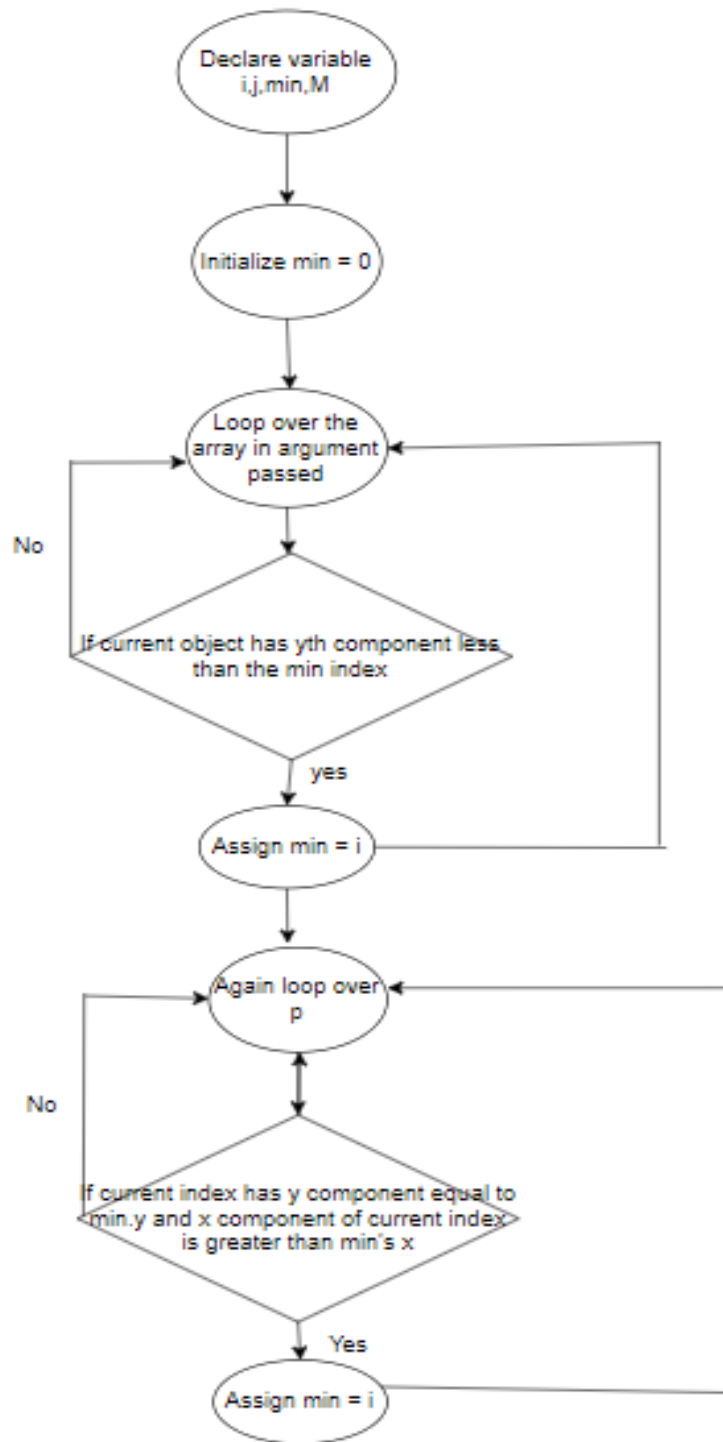


The screenshot shows an IDE with a Java project. The Package Explorer on the left shows a list of test classes: prog3Test, prog5Test, prog6Test, prog4Test, prog1Test, and prog2Test. The main editor displays the source code for prog6.java. The code defines a static method triangle that takes three double parameters (d, e, f) and returns an integer. The method checks for non-positive values, equality of sides, and the triangle inequality to determine the type of triangle (3 for right-angled, 0 for invalid, 1 for equilateral, and 2 for isosceles).

```
1 package lab7_202001114;
2
3 public class prog6 {
4     static int triangle(double d, double e, double f)
5     {
6         if (d >= e+f || e >= d+f || f >= d+e || d<=0 || e<=0 || f<=0)
7             return(3);
8         if (d == e && e == f)
9             return(0);
10        if (d == e || d == f || e == f)
11            return(1);
12        if((d*d + e*e == f*f) || (d*d + f*f == e*e) || (e*e + f*f == d*d))return(4);
13        return(2);
14    }
15 }
16
17
```

Section B:

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.

-> Statement Coverage

Test Number	Test Case
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

->Branch Coverage

Test Number	Test Case
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

-> Basic Condition Coverage

Test Number	Test Case
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