IT-314: Software Engineering

Name: Shah Poojan Dikeshkumar
Student ID: 202001428
Lab 7

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?

Solution:

From the given input constraints, the following are the equivalence classes obtained.

Equivalent classes:

E1 = {1<=date<=31}
E2 = { date<1}
E3= {date >31 }
E4 = {1<=month <=12}
E5 = { month <1}
E6 = {month >12}
E7 = {1900<=year<=2015}
E8 = {year <1900}
E9 = {year>2015 }

There are 9 equivalent classes

The following are the weak normal equivalence class test cases:

Equivalent class	day	month	year	output
E1	2	3	2011	1/3/2021
E2	0	4	2022	Invalid date
E3	34	5	2000	Invalid date
E4	1	1	1980	31/12/1989
E5	21	-4	1970	Invalid
E6	20	15	1943	Invalid
E7	4	5	1980	3/5/1980
E8	5	6	1899	Invalid
E9	4	3	2016	Invalid

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.

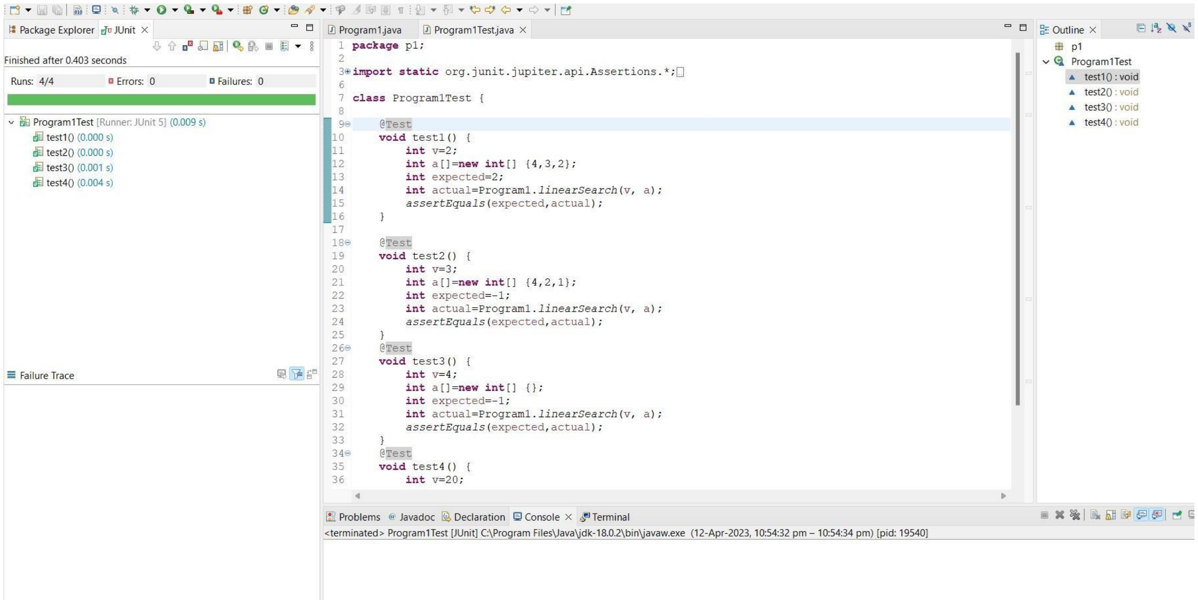
```
int linearSearch(int v, int a[])
{
        int i = 0;
        while (i < a.length)
        {
            if (a[i] == v)
                return(i);
            i++;
        }
        return (-1);
}
```

Test cases:

1) v=2 , a={4,3,2} expected output = 2
2) v=3 , a={4,2,1} expected output = -1
3) v=4 , a={} expected output = -1
4) v=20 , a= {10,20,30,20,40} expected output = 1

Tester Action and Input Data	**Expected Outcome**
Equivalence Partitioning	
v=2 , a={4,3,2}	2
v=3 , a={4,2,1}	-1
v=20 , a= {10,20,30,20,40}	1
Boundary Value Analysis	
v=4 , a={}	-1

Junit Testing:



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);
}
```

Test cases :

1)v=2 , a={4,2,3,2,1} expected output = 2

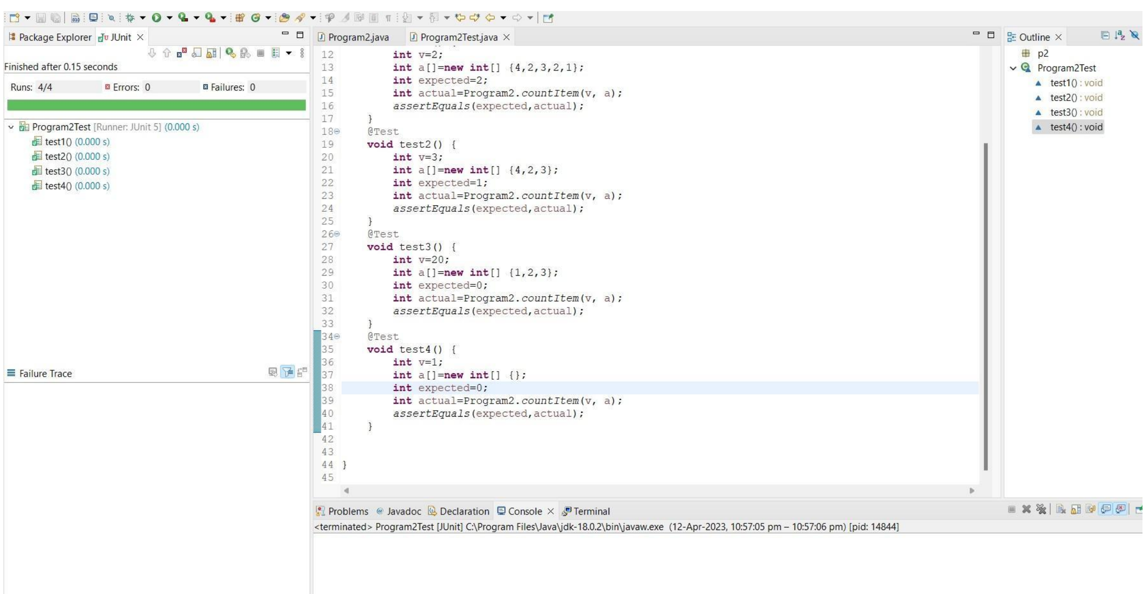
2) v=3 , a={4,2,3} expected output =1

3) v= 20 , a= {1,2,3} expected output =0

4) v=1 , a ={} , expected output = 0

Tester Action and Input Data	Expected Outcome
Equivalence Partitioning	
v=2 , a={4,2,3,2,1}	2
v=3 , a={4,2,3}	1
v= 20 , a= {1,2,3}	0
Boundary Value Analysis	
v=1 , a ={}	0

Junit Testing:



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 is sorted in non-decreasing order.

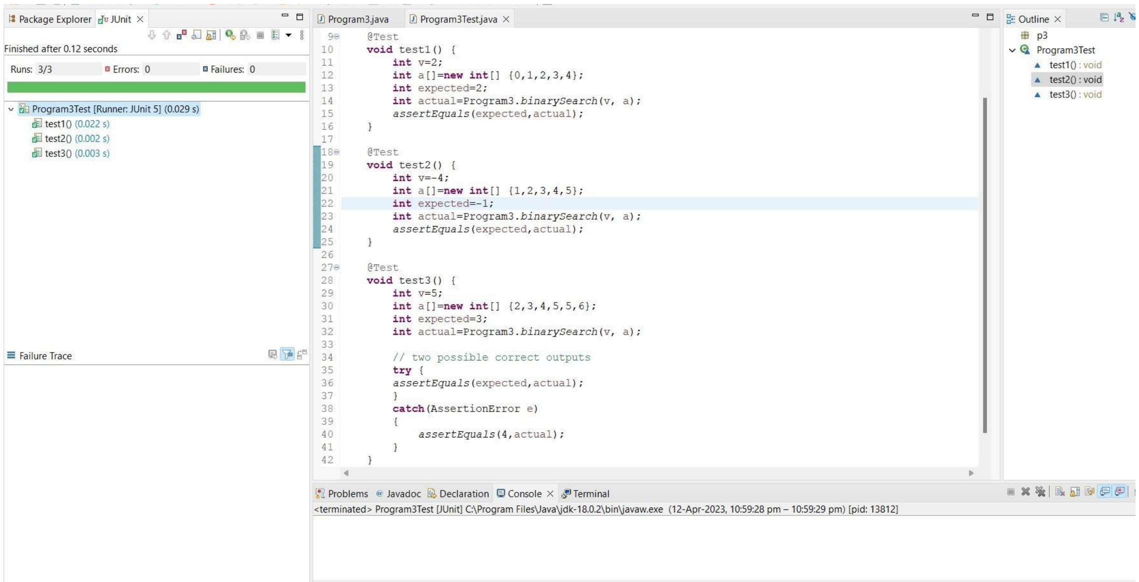
```
int binarySearch(int v, int a[])
{
        int lo,mid,hi;
        lo = 0;
        hi = a.length-1;
        while (lo <= hi)
        {
          mid = (lo+hi)/2;
          if (v == a[mid])
            return (mid);
          else if (v < a[mid])
              hi = mid-1;
          else
          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

Test cases:

1) v=2 , a= { 0, 1,2,3,4} expected output = 2
2) v= -4 , a= {1,2,3,4,5 } expected output = -1
3) v=5 , a= {2,3,4,5,5,6}   expected output = 3 or 4

Tester Action and Input Data	**Expected Outcome**
Equivalence Partitioning	
v=2 , a= { 0, 1,2,3,4}	2
v=5 , a= {2,3,4,5,5,6}	3 or 4
Boundary Value Analysis	
v= -4 , a= {1,2,3,4,5 }	-1

Junit Testing:



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 cases:

1)a=4,b=4,c=4   expected output = EQUILATERAL

2)a=1,b=2,c=3   expected output= INVALID

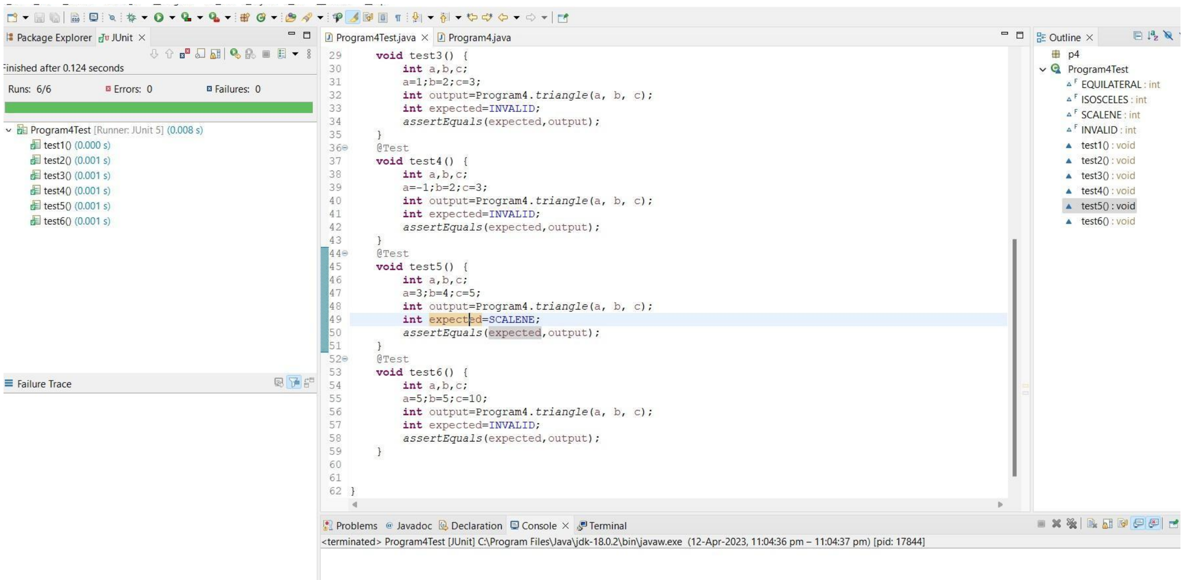
3) a=-1,b=2,c=3 expected output = INVALID 4)a=3,b=4,c=5
expected output = SCALENE

5) a=5,b=5,c=9 expected output = ISOSCELES

6) a=5, b=5, c=10 expected output = INVALID

Tester Action and Input Data	**Expected Outcome**
Equivalence Partitioning	
a=4,b=4,c=4	EQUILATERAL
a=5,b=5,c=9	ISOSCELES
a=5, b=5, c=10	INVALID
a=3,b=4,c=5	SCALENE
Boundary Value Analysis	
a=-1,b=2,c=3	INVALID
a=1,b=2,c=3	INVALID

Junit Testing:



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;
}
```

Test cases:

1)s1=”soft”, s2=”software” , expected output =true

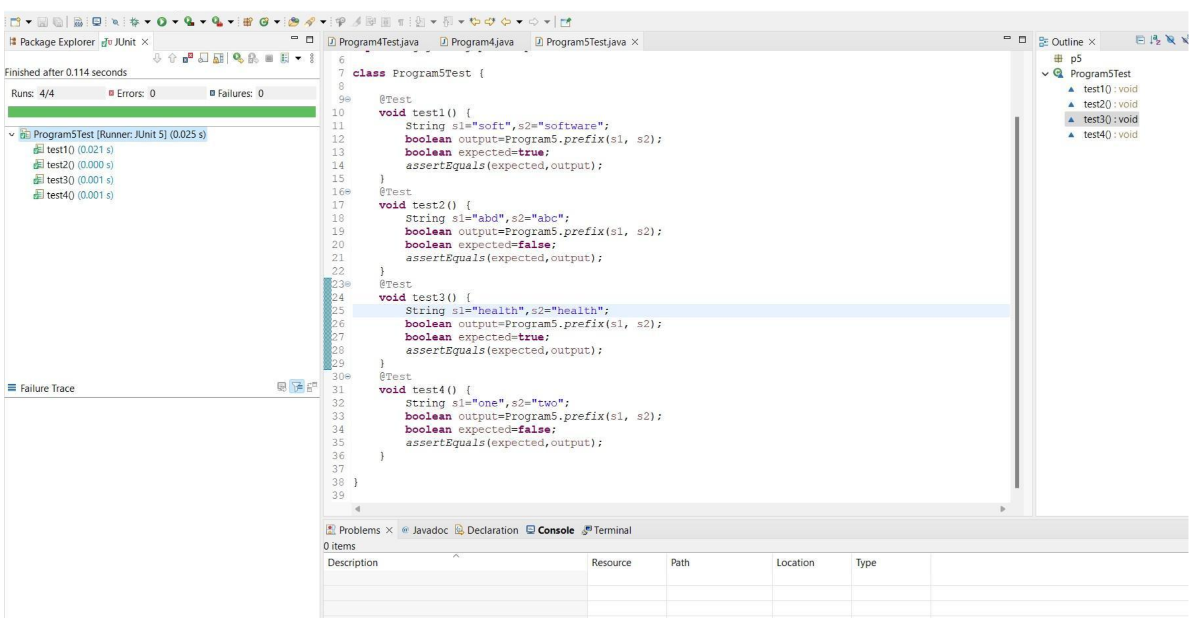
2)s1=”abd” , s2=”abc” , expected output = false

3) s1=”health” , s2=”health”, expected output=true

4) s1=”one” , s2=”two”, expected output=false 5)s1=””,s2=”sdf” , expected output=true

Tester Action and Input Data	**Expected Outcome**
Equivalence Partitioning	
s1=”soft”, s2=”software”	true
s1=”abd” , s2=”abc”	false
s1=”health” , s2=”health”	true
s1=”one” , s2=”two”	true
Boundary Value Analysis	
s1=””,s2=”sdf”	true

Junit Testing:



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 $A^2 + B^2 = C^2$ 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 ($a^2+b^2=c^2$)
2) a=0.12, b=0.5 , c=0.14 (a^2+b^2 just less than c^2)
3) a=7 , b=23 , c=24 (a^2+b^2 just greater than c^2)

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

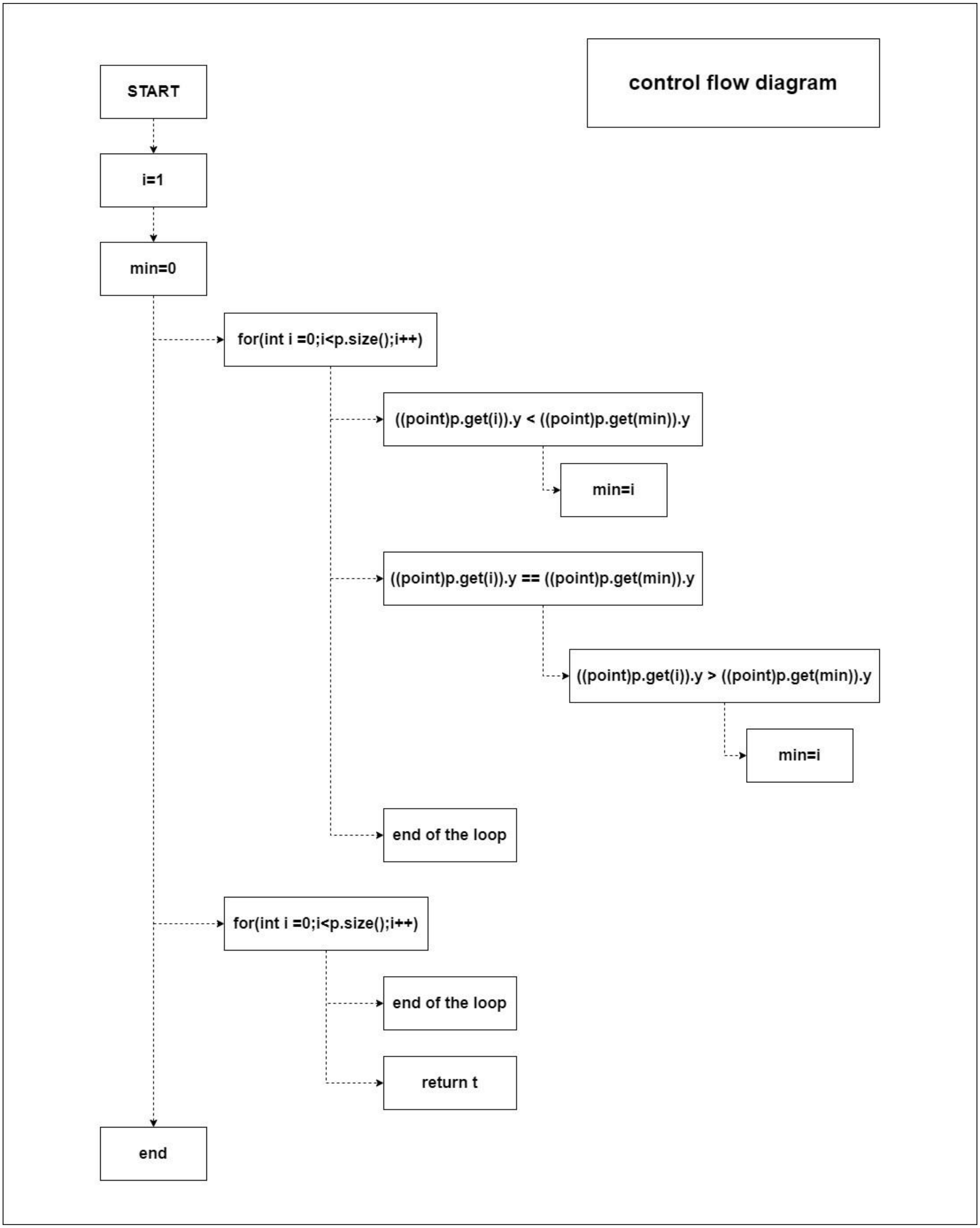
The code below is part of a method in the ConvexHull class in the VMAP system. The following is a small fragment of a method in the ConvexHull class. For the purposes of this exercise you do not need to know the intended function of the method. The parameter p is a Vector of Point objects, p.size() is the size of the vector p, (p.get(i)).x is the x component of the ith 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).

Control Flow graph (CFG):



The above control graph represents the structural flow of the given function. For decision nodes, the left branch is when the node evaluates to True and the right part is when the node evaluates to False.

2. Construct test sets for your flow graph that are adequate for the following criteria:
Statement Coverage.
Branch Coverage.
Basic Condition Coverage.

Solution:
a. Statement Coverage:
To satisfy statement coverage, we need to ensure that each statement in the CFG is executed at least once. We can achieve this by providing a test case with a single point in the vector. In this case, both loops will not execute, and the return statement will be executed

Test 1: p = {new Point(0, 0), new Point(1, 1)}
Test 2: p = {new Point(0, 0), new Point(1, 0), new Point(2, 0)}

b. Branch Coverage:
To satisfy branch coverage, we need to ensure that each branch in the CFG is executed at least once. We can achieve this by providing a test case with two points such that one of the points has the minimum y-coordinate, and the other has a greater x-coordinate than the minimum. In this case, both loops will execute, and the second branch in the second loop will be taken.

Test 1: p = {new Point(0, 0), new Point(1, 1)}
Test 2: p = {new Point(0, 0), new Point(1, 0), new Point(2, 0)}
Test 3: p = {new Point(0, 0), new Point(1, 0), new Point(1, 1)}

c. Basic Condition Coverage:
To satisfy basic condition coverage, we need to ensure that each condition in the CFG is evaluated to both true and false at least once. We can achieve this by providing a

test case with three points such that two of the points have the same y-coordinate, and the other has a greater x-coordinate than the minimum. In this case, both loops will execute, and the second condition in the second loop will be evaluated to true and false.

Test 1: p = {new Point(0, 0), new Point(1, 1)}
Test 2: p = {new Point(0, 0), new Point(1, 0), new Point(2, 0)}
Test 3: p = {new Point(0, 0), new Point(1, 0), new Point(1, 1)}
Test 4: p = {new Point(0, 0), new Point(1, 0), new Point(0, 1)}
Test 5: p = {new Point(0, 0), new Point(0, 1), new Point(1, 1)}