

IT 314 Software engineering

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 $1 \leq \text{month} \leq 12$, $1 \leq \text{day} \leq 31$, $1900 \leq \text{year} \leq 2015$. The possible output dates would be previous date or invalid date. Design the equivalence class test cases?

Ans:

Equivalent classes:

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

There are 9 equivalent classes

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.

Code:

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

Test cases:

`v=2` , `a={4,3,2}` expected output = 2

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

`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:

The screenshot displays an IDE with the following components:

- Package Explorer:** Shows the project structure with 'Program1Test' and its methods: 'test1(): void', 'test2(): void', 'test3(): void', and 'test4(): void'.
- JUnit Runner:** Shows the test results for 'Program1Test' (Runner: JUnit 5) with a duration of 0.009 s. It lists four tests: 'test1() (0.000 s)', 'test2() (0.000 s)', 'test3() (0.001 s)', and 'test4() (0.004 s)'. All tests passed, with 4 runs, 0 errors, and 0 failures.
- Source Code:** The 'Program1Test.java' file is open, showing the implementation of the four test methods. Each method calls 'Program1.linearSearch(v, a)' and asserts the result against an expected value.

```
1 package p1;
2
3 import static org.junit.jupiter.api.Assertions.*;
4
5
6
7 class Program1Test {
8
9     @Test
10    void test1() {
11        int v=2;
12        int a[]=new int[] {4,3,2};
13        int expected=2;
14        int actual=Program1.linearSearch(v, a);
15        assertEquals(expected,actual);
16    }
17
18    @Test
19    void test2() {
20        int v=3;
21        int a[]=new int[] {4,2,1};
22        int expected=-1;
23        int actual=Program1.linearSearch(v, a);
24        assertEquals(expected,actual);
25    }
26
27    @Test
28    void test3() {
29        int v=4;
30        int a[]=new int[] {};
31        int expected=-1;
32        int actual=Program1.linearSearch(v, a);
33        assertEquals(expected,actual);
34    }
35
36    @Test
37    void test4() {
38        int v=20;
```

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);
}
```

Test cases :

- 1)v=2 , a={4,2,3,2,1} expected output = 2
- 2)v=3 , a={4,2,3} expected output =1
- 2) v= 20 , a= {1,2,3} expected output =0
- 3) v=1 , a ={} , expected output = 0

Tester Action and Input Data	Expected
-------------------------------------	-----------------

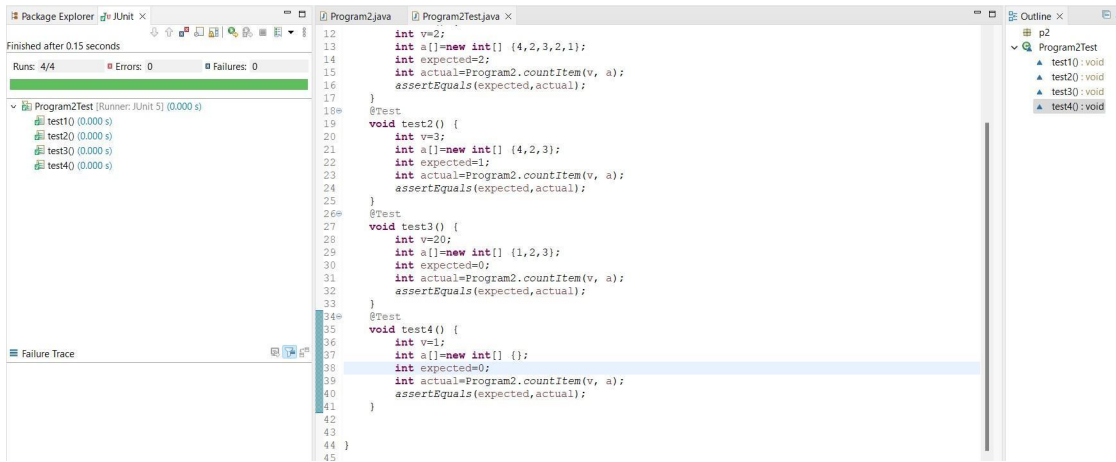
Outcome Equivalence Partitioning

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.

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);
}
```

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

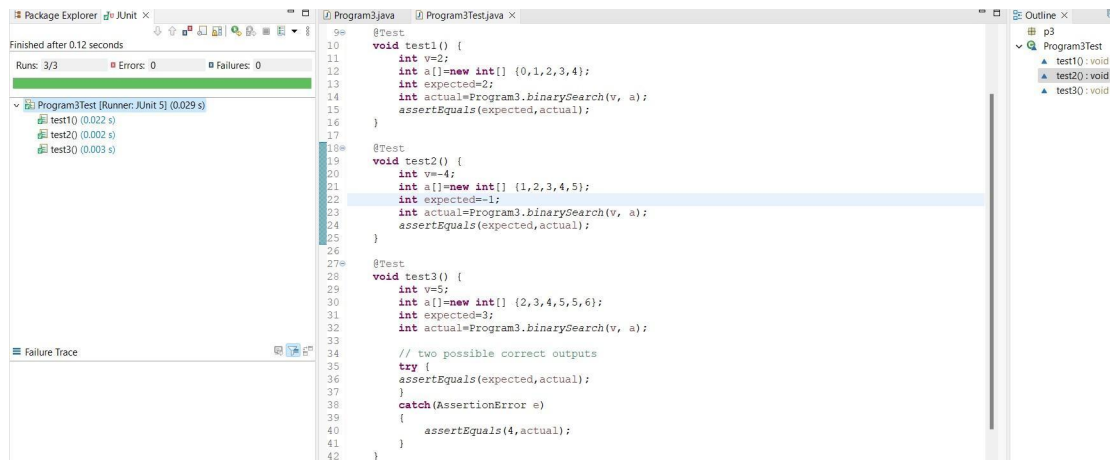
v=5 , a= {2,3,4,5,5,6}

3 or 4

Boundary Value Analysis

v= -4 , a= {1,2,3,4,5 }

-1



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);
}
```

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:

The screenshot displays the JUnit test results for Program4Test. The test results pane on the left shows that all tests passed, including test10, test20, test30, test40, test50, and test60. The source code pane in the center shows the test50 method, which is highlighted. The test50 method is a JUnit test that calls Program4.triangle(a, b, c) with a=3, b=4, c=5 and expects the output to be SCALENE. The right pane shows the package structure for p4, including Program4Test and its sub-packages.

```
29 void test3() {
30     int a,b,c;
31     a=1;b=2;c=3;
32     int output=Program4.triangle(a, b, c);
33     int expected=INVALID;
34     assertEquals(expected,output);
35 }
36 @Test
37 void test4() {
38     int a,b,c;
39     a=-1;b=2;c=3;
40     int output=Program4.triangle(a, b, c);
41     int expected=INVALID;
42     assertEquals(expected,output);
43 }
44 @Test
45 void test5() {
46     int a,b,c;
47     a=3;b=4;c=5;
48     int output=Program4.triangle(a, b, c);
49     int expected=SCALENE;
50     assertEquals(expected,output);
51 }
52 @Test
53 void test6() {
54     int a,b,c;
55     a=5;b=5;c=10;
56     int output=Program4.triangle(a, b, c);
57     int expected=INVALID;
58     assertEquals(expected,output);
59 }
60
61
```

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;
}
```

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:

The screenshot displays an IDE interface with three main panels. The left panel, titled 'Package Explorer', shows a project named 'JUnit' with a sub-project 'Program5Test'. Under 'Program5Test', there are four test methods: 'test1() (0.021 s)', 'test2() (0.000 s)', 'test3() (0.001 s)', and 'test4() (0.001 s)'. The status bar indicates 'Runs: 4/4', 'Errors: 0', and 'Failures: 0'. The middle panel shows the source code for 'Program5Test.java'. The code defines a class 'Program5Test' with four test methods: 'test1()', 'test2()', 'test3()', and 'test4()'. Each method calls 'Program5.prefix()' with specific string arguments and asserts the result. The right panel, titled 'Outline', shows the project structure with 'Program5Test' expanded, listing the four test methods.

```
6
7 class Program5Test {
8
9     @Test
10    void test1() {
11        String s1="soft",s2="software";
12        boolean output=Program5.prefix(s1, s2);
13        boolean expected=true;
14        assertEquals(expected,output);
15    }
16    @Test
17    void test2() {
18        String s1="abd",s2="abc";
19        boolean output=Program5.prefix(s1, s2);
20        boolean expected=false;
21        assertEquals(expected,output);
22    }
23    @Test
24    void test3() {
25        String s1="health",s2="health";
26        boolean output=Program5.prefix(s1, s2);
27        boolean expected=true;
28        assertEquals(expected,output);
29    }
30    @Test
31    void test4() {
32        String s1="one",s2="two";
33        boolean output=Program5.prefix(s1, s2);
34        boolean expected=false;
35        assertEquals(expected,output);
36    }
37
38 }
39
```

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

Invalid case:

$$E1 : a+b \leq c$$

$$E2 : a+c \leq b$$

$$E3: b+c \leq a$$

Equilateral case:

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

Isosceles case:

$$E5 : a=b ,$$

$$a \neq c \quad E6: a = c,$$

$$a \neq b \quad E7: b = c,$$

$$b \neq a$$

Scalene case:

$$E8 : a \neq b , b \neq c, c \neq a$$

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 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

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 $a \neq c$)
- 3) $a=10$ $b=11$ $c=10$ ($a=c$ but $a \neq b$)

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 `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.

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.

Ans:

```
1.  int i,j,min,M;
2.  Point t;
3.  min=0;
4.  for(i=1;i<p.size();++i)
    {
5.      if(((Point)P.get(i)).y<((Point)P.get(min)).y)
6.          min=i;
    }

7.  for(i=0;i<p.size();++i)
    {
8.      if(((Point)P.get(i)).y==((Point)P.get(min)).y
        &&
        ((Point)P.get(i)).x>((Point)P.get(min)).x)
9.          min=i;
    }
```

Test cases:

1) p=[(x=2,y=2),(x=2,y=3),(x=1,y=3),(x=1,y=4)]

Statements covered = { 1,2,3,4,5,7,8}

Branches covered = {5,8}

Basic conditions covered = {5-false, 8-false}

2) p=[(x=2,y=3),(x=3,y=4),(x=1,y=2),(x=5,y=6)]

Statements covered = { 1,2,3,4,5,6,7}

Branches covered = {5,8}

Basic conditions covered = {5-false,true, 8-false}

3) p=[(x=1,y=5),(x=2,y=7),(x=3,y=5),(x=4,y=5),(x=5,y=6)]

Statements covered = { 1,2,3,4,5,6,7,8,9}

Branches covered = {5,8}

Basic conditions covered = {5-false,true, 8-false,true}

4) p=[(x=1,y=2)]

Statements covered = { 1,2,3,7,8}

Branches covered = {8}

Basic conditions covered = {}

5) p=[]

Statements covered = { 1,2,3}

Branches covered = {}

Basic conditions covered = {}