IT314 - Software Engineering

Lab - 7

Testing

Name - Harsh Vadi

Student ID - 202001143

Section A

Based on the input ranges, we can identify the following equivalence classes:

1. **Valid dates**: The input triple (day, month, year) representing a valid date in the Gregorian calendar, such as (3, 4, 1995).
2. **Invalid dates**: The input triple (day, month, year) that represent an invalid date, such as (31, 2, 2022) or (29, 2, 1900).
3. **Out of range dates**: The input triple (day, month, year) that are outside the allowed ranges, such as (0, 5, 2010) or (15, 13, 2005). Based on these equivalence classes, we can design the following test cases:

Tester Action and Input Data Expected Outcome

Valid dates:

1. Calculate previous date for (15, 10, 2022) 14, 10, 2022
2. Calculate previous date for (1, 1, 2015) 31, 12, 2014
3. Calculate previous date for (31, 3, 2000) 30, 3, 2000

Invalid dates:

1. Calculate previous date for (29, 2, 2022) Invalid date
2. Calculate previous date for (31, 4, 2010) Invalid date
3. Calculate previous date for (30, 2, 2000) Invalid date

Out-of-range dates:

1. Calculate previous date for (0, 5, 2010) Invalid date
2. Calculate previous date for (15, 13, 2005) Invalid date
3. Calculate previous date for (31, 12, 1899) Invalid date

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, 12, 2000)
4. The latest day of each month: (31, 1, 2000), (28, 2, 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

Boundary Test Cases:

1. Calculate previous date for (1, 1, 1900) Invalid date
2. Calculate previous date for (31, 12, 2015) 30, 12, 2015
3. Calculate previous date for (1, 1, 2000) 31, 12, 1999
4. Calculate previous date for (31, 1, 2000) 30, 1, 2000
5. Calculate previous date for (29, 2, 2000) 28, 2, 2000
6. Calculate previous date for (29, 2, 1900) Invalid date
7. Calculate previous

Program 1:

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:

Tester Action and Input Data	Expected Outcome
v is an invalid number, and a is empty v = NULL, a = [ ]	-1
v is a valid number, and a is empty v = 8, a = [ ]	-1
v is an invalid number, and a is non-empty v = NULL, a = [2, 4, 7]	-1
v is a valid number, a is non-empty, and v is not present v = 2, a = [1, 3, 6]	-1
v is a valid number, a is non-empty, and v is present v = 6, a = [2, 4, 6, 7]	Index of v 2

Boundary value analysis

Tester Action and Input Data	Expected Outcome
v is a valid number, a is non-empty, and v is present at 1st index v = 7, a = [7, 6, 4]	0
v is a valid number, a is non-empty, and v is present at the last index v = 4, a = [7, 6, 4]	a.size() - 1
v is a valid number, a is non-empty, and v is not present v = 9, a = [7, 6, 4]	-1

Program 2:

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);
}
```

Equivalence Partitioning:

Tester Action and Input Data	Expected Outcome
v is an invalid number, and a is empty v = NULL, a = [ ]	0
v is a valid number, and a is empty v = 7, a = [ ]	0
v is an invalid number, and a is non-empty v = NULL, a = [7, 6, 4]	0
v is a valid number, a is non-empty, and v is present one time v = 7, a = [7, 6, 4]	1
v is a valid number, a is non-empty, and v is present multiple times v = 7, a = [7, 7, 4]	Number of occurrences of v 2

Boundary value analysis

Tester Action and Input Data	Expected Outcome
v is a valid number, a is non-empty, and v is present at 1st index v = 7, a = [7, 6, 4]	1
v is a valid number, a is non-empty, and v is present at the last index v = 4, a = [7, 6, 4]	1
v is a valid number, a is non-empty, and v is not present v = 5, a = [7, 6, 4]	0
v is a valid number, a is non-empty, and v is present once v = 6, a = [7, 6, 4]	1
v is a valid number, a is non-empty, and v is present multiple times v = 6, a = [6, 6, 6, 4]	Number of occurrences of v 3

Program 3:

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:

Tester Action and Input Data	Expected Outcome
v is an invalid number, and a is empty v = NULL, a = [ ]	-1
v is a valid number, and a is empty v = 7, a = [ ]	-1
v is an invalid number, and a is non-empty v = NULL, a = [7, 6, 4]	-1
v is a valid number, a is non-empty, and v is not present v = 9, a = [7, 6, 4]	-1

v is a valid number, a is non-empty, and v is present v = 7, a = [7, 6, 4]	Index of v 0

Boundary value analysis

Tester Action and Input Data	Expected Outcome
v is a valid number, a is non-empty, and v is present at 1st index v = 7, a = [7, 6, 4]	0
v is a valid number, a is non-empty, and v is present at the last index v = 4, a = [7, 6, 4]	a.size() - 1 2
v is a valid number, a is non-empty, and v is not present v = 10, a = [7, 6, 4]	-1

Program 4:

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:

Tester Action and Input Data	Expected Outcome
Invalid triangle (a+b<=c) a=3, b=4, c=11	INVALID
Valid equilateral triangle (a=b=c) a=4, b=4, c=4	EQUILATERAL
Valid isosceles triangle (a=b<c) a=4, b=4, c=5	ISOSCELES
Valid scalene triangle (a<b<c) a=3, b=4, c=6	SCALENE

Boundary value analysis

Tester Action and Input Data	Expected Outcome
Invalid triangle (a+b<=c) a=3, b=4, c=10	INVALID
Invalid triangle (a+c<b) a=3, b=9, c=5	INVALID
Invalid triangle (b+c<a) a=12, b=4, c=6	INVALID
Valid equilateral triangle (a=b=c) a=3, b=3, c=3	EQUILATERAL
Valid isosceles triangle (a=b<c) a=3, b=3, c=4	ISOSCELES
Valid isosceles triangle (a=c<b) a=3, b=4, c=3	ISOSCELES
Valid isosceles triangle (b=c<a) a=4, b=3, c=3	ISOSCELES
Valid scalene triangle (a<b<c) a=3, b=4, c=6	SCALENE

Program 5:

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:

Tester Action and Input Data	Expected Outcome
String s1 empty and String s2 empty s1="", s2=""	true
String s1 empty and String s2 non-empty s1="", s2="abc"	true
String s1 non-empty and String s2 empty s1="abc", s2=""	false
String s1 non-empty, String s2 non-empty, and s1.size() > s2.size() s1="abc", s2="a"	false
String s1 non-empty, String s2 non-empty, s1.size() <= s2.size() and s1 is prefix of s2 s1="abc", s2="abcd"	true

String s1 non-empty, String s2 non-empty, s1.size() <= s2.size() and s1 is not a prefix of s2 s1=”abc”, s2=”defg”	false

Boundary value analysis

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

Program 6:

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

Equivalence Classes:
EC1: All sides are positive, real numbers.
EC2: One or more sides are negative or zero.
EC3: The sum of the lengths of any two sides is less than or equal to the length of the remaining side (impossible lengths).

EC4: The sum of the lengths of any two sides is greater than the length of the remaining side (possible lengths).

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 (EC1): A=3, B=4, C=5 (right-angled triangle)
TC2 (EC1): A=5, B=5, C=5 (equilateral triangle)
TC3 (EC1): A=5, B=6, C=7 (scalene triangle)
TC4 (EC1): A=5, B=5, C=7 (isosceles triangle)
TC5 (EC2): A=-2, B=4, C=5 (invalid input)
TC6 (EC2): A=0, B=4, C=5 (invalid input)

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

Test cases for the boundary condition A + B > C:
TC7 (EC4): A=2, B=3, C=6 (sum of A and B is equal to C)

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

Test cases for the boundary condition A = C:
TC8 (EC4): A=5, B=6, C=5 (A equals to C)

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

Test cases for the boundary condition A = B = C:
TC9 (EC4): A=1, B=1, C=1 (all sides are equal)

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

Test cases for the boundary condition A^2 + B^2 = C^2:
TC10 (EC4): A=3, B=4, C=5 (right-angled triangle)

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

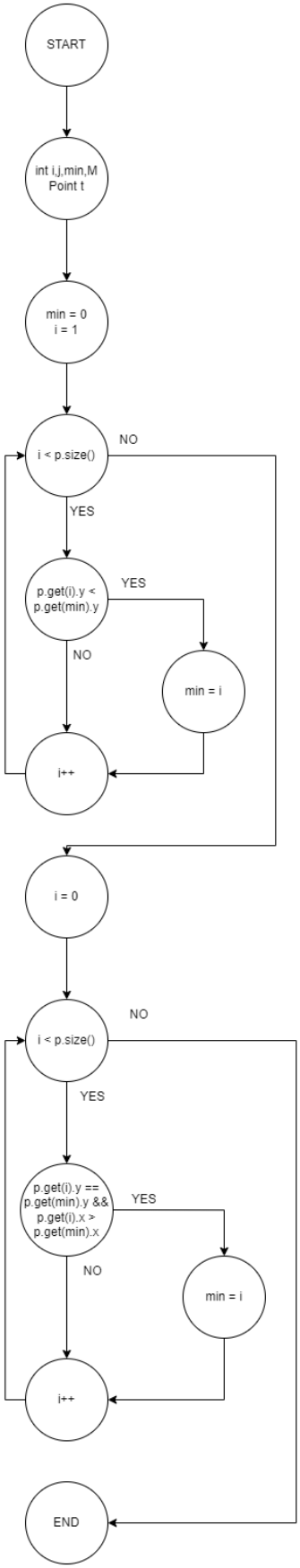
Test cases for the non-triangle case:
TC11 (EC3): A=2, B=2, C=4 (sum of A and B is less than C)

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)
Note: Test cases TC1 to TC10 covers all identified equivalence classes.

Section B

Control flow diagram:


START
int i,j,min,M
Point t
min = 0
i = 1
i < p.size()
NO
YES
p.get(i).y <
p.get(min).y
YES
NO
min = i
i++
i = 0
i < p.size()
NO
YES
p.get(i).y ==
p.get(min).y &&
p.get(i).x >
p.get(min).x
YES
NO
min = i
i++
END

Test sets:

a) **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

b) **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

c) **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.