# Name = Sangani Hemil Sharadbhai
# Id = 202001144
# Lab = 7
# Course = IT 314

## Section A

*Equivalence Partitioning:*

| Tester Action and Input Data | Expected Outcome |
|---|---|
| Valid Date | Previous Date |
| Invalid Date | Invalid Date |
| Out of Range Date | Invalid Date |

*Boundary Value Analysis:*

| Tester Action and Input Data | Expected Outcome |
|---|---|
| First possible date (1/1/1900) | Invalid Date |
| Last possible date (31/12/2015) | Previous Date (30/12/2015) |
| First day of a month (1/1/2003) | Previous Date (31/12/2002) |

| | |
|---|---|
| Last day of a month (31/1/2003) | Previous Date (30/1/2003) |
| Leap year day (29/2/2004) | Previous Date (28/2/2004) |
| Invalid Leap year day (29/2/1901) | Invalid Date |
| Day before first date (31/12/1899) | Invalid Date |
| Day after last day (1/1/2016) | Invalid Date |

# 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 |
|---|---|
| invalid number v, and empty a | -1 |
| valid number v, and empty a | -1 |
| invalid number v, and non-empty a | -1 |
| valid number v, non-empty a, and v is not present | -1 |

| valid number v, non-empty a, and v is present | Index of v |
|---|---|

## *Boundary value analysis:*

| Tester Action and Input Data | Expected Outcome |
|---|---|
| valid number v, non-empty a, and v is at 1st index | 0 |
| valid number v, non-empty a, and v is at the last index | a.size() - 1 |
| valid number v, non-empty a, and v is not present | -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 |
|---|---|
| invalid number v, and empty a | 0 |
| valid number v, and empty a | 0 |
| invalid number v, and non-empty a | 0 |
| valid number v, non-empty a, and v is present one time | 1 |
| valid number v, non-empty a, and v is present multiple times | Number of occurrences of v |

## *Boundary value analysis:*

| Tester Action and Input Data | Expected Outcome |
|---|---|
| valid number v, non-empty a, and v is at 1st index | 1 |
| valid number v, non-empty a, and v is at the last index | 1 |
| valid number v, non-empty a, and v is not present | 0 |
| valid number v, non-empty a, and v is present once | 1 |
| valid number v, non-empty a, and v is present multiple times | Number of occurrences of v |

# 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 |
|---|---|
| invalid number v, and empty a | -1 |
| valid number v, and empty a | -1 |
| invalid number v, and non-empty a | -1 |
| valid number v, non-empty a, and v is not present | -1 |
| valid number v, non-empty a, and v is present | Index of v |

## *Boundary value analysis:*

| Tester Action and Input Data | Expected Outcome |
|---|---|
| valid number v, non-empty a, and v is at 1st index | 0 |
| valid number v, non-empty a, and v is at the last index | a.size() - 1 |

| | |
|---|---|
| valid number v, non-empty a, and v is not present | -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) | INVALID |
| Valid equilateral triangle (a=b=c) | EQUILATERAL |
| Valid isosceles triangle (a=b<c) | ISOSCELES |
| Valid scalene triangle (a<b<c) | SCALENE |

## Boundary value analysis:

| Tester Action and Input Data | Expected Outcome |
|---|---|
| Invalid triangle (a+b<=c) | INVALID |
| Invalid triangle (a+c<b) | INVALID |
| Invalid triangle (b+c<a) | INVALID |
| Valid equilateral triangle (a=b=c) | EQUILATERAL |
| Valid isosceles triangle (a=b<c) | ISOSCELES |
| Valid isosceles triangle (a=c<b) | ISOSCELES |
| Valid isosceles triangle (b=c<a) | ISOSCELES |
| Valid scalene triangle (a<b<c) | 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 |
|---|---|
| s1 empty and s2 empty | True |
| s1 empty and s2 non-empty | True |
| s1 non-empty and s2 empty | False |
| s1 non-empty, s2 non-empty, and s1.size() > s2.size() | False |
| s1 non-empty, s2 non-empty, s1.size() <= s2.size() and s1 is prefix of s2 | True |
| s1 non-empty, s2 non-empty, s1.size() <= s2.size() and s1 is not a prefix of s2 | False |

## *Boundary value analysis:*

| Tester Action and Input Data | Expected Outcome |
|---|---|
| Empty s1 and s2 | True |
| Empty s1 and non-empty s2 | True |
| Non-empty s1 is not a prefix of s2 | False |
| Non-empty s1 is longer than s2 | 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=6, B=8, C=10 (right-angled triangle)
TC2 (EC1): A=7, B=7, C=7 (equilateral triangle)
TC3 (EC1): A=5, B=6, C=8 (scalene triangle)
TC4 (EC1): A=8, B=8, C=7 (isosceles triangle)
TC5 (EC2): A=-2, B=8, C=5 (invalid input)
TC6 (EC2): A=8, B=0, 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=4, C=7 (sum of A and B is less than 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=7, 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=6, B=8, C=10 (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=4, B=4, C=8 (sum of A and B is equal to C)

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

Test points for non-positive input:
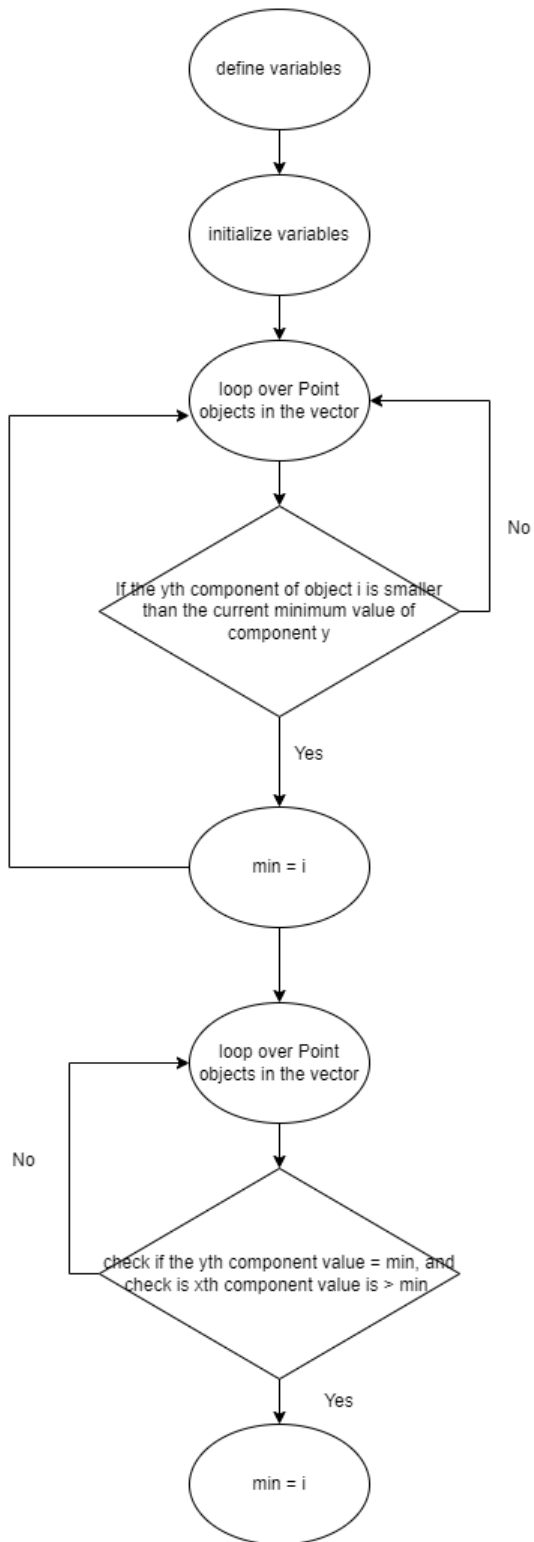TP1 (EC2): A=8, B=0, C=5 (invalid input)
TP2 (EC2): A=-2, B=8, C=5 (invalid input)
Note: Test cases TC1 to TC10 covers all identified equivalence classes.

# Section B

*Control flow diagram:*

define variables

initialize variables

loop over Point objects in the vector

If the yth component of object i is smaller than the current minimum value of component y

No

Yes

min = i

loop over Point objects in the vector

No

check if the yth component value = min, and check is xth component value is > min

Yes

min = i

**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.