Perform the following tasks:

a. Write one simple program that prints out the value of the -11 modulo 3 (e.g., -11%3 in Python) in each of the following three programming languages: C/C++, Java, and Python. You will write three programs in total. Report what you have found. (4 points)
b. Add to each of the previously written program codes to print out the value of the following integer division -11/3 (e.g., -11//3 in Python). Report what you have found and if they are related to the findings in task 1. (4 points)
c. Write the same program in JavaScript and see if you can replicate the integer division results from all other three languages by making different design choices. Explain the choices and how that is related to the findings in task 1. (2 points)
d. Summarize the problem and explain why we see such differences in results. (4 points)
e. Discuss at least 2 different strategies to cope with such a problem (Hint: think about what can change: developers, compilers, standards, etc) (6 points)

For this question, submit the following files:

- Result, explanation, and discussion:
    - firstname_lastname_q1.pdf
- Source files:
    - firstname_lastname_q1.java
    - firstname_lastname_q1.c or *.cpp
    - firstname_lastname_q1.py
    - firstname_lastname_q1.js

Below is a faulty Java program, which includes a test case that results in a failure. The if-statement needs to take into account negative values. A possible fix is:

```
if (x[i]%2 != 0)
```

Answer the following questions for this program.

a. If possible, identify a test case that does not execute the fault.
b. If possible, identify a test case that executes the fault, but does not result in an error state.
c. If possible, identify a test case that results in an error, but not a failure.
d. For the given test case identify the first error state. Describe the complete state.

```
public static int even(int[] x) {
// Effects: if x == null throw NullPointerException ,
// else return the number of elements in x that are even
  int count = 0;
  for (int i = 0; i < x.length; i++) {
    if (x[i]%2 == 1) {
      count++;
    }
  }
  return x.length - count;
}
// test: x = [-18, 0, -99, 17, 102, 16]
// Expected: 4
```

Consider the following program:

```
void DonNothingWithManyIfs(boolean[] x){

        if (x == null)

                return;

        if (x.length == 1)

                if (x[0])

                        // bug 1

        if (x.length > 1)

                if (!x[0])

                        // bug 2

        // bug 3

}
```

Answer the following questions:

a. Draw the control flow graph of this program where each node is one statement. Using any computer diagrams or drawing tools such as Paint, Google Draw, MS Visio, Photoshop, etc. Be sure to label all nodes and edges, and include proper code sections in each of the nodes.
b. Is static symbolic execution viable for this program? Explain your answer.
c. Perform dynamic symbolic execution to find the condition to reach all statements for this program. Include the table as shown in the lecture slides as well as clearly state which test data is needed to reach which statement.

Consider the following (contrived) program:

```
class M {
      public static void main(String [] argv){
            M obj = new M();
            if (argv.length > 0)
                  obj.m(argv[0], argv.length);
      }

      public void m(String arg, int i) {
            int q = 1;
            A o = null;
            Impossible none = new Impossible();
            if (i == 0)
                  q = 4;
            q++;
            switch (arg.length()) {
                  case 0: q /= 2;
                  case 1: o = new A(); new B(); q *= 3; break;
                  case 2: o = new A(); q = q * 100;
                  default: o = new B(); break;
            }
            if (arg.length() > 0) {
                  o.m();
            } else {
                  System.out.println("zero");
            }
            none.happened();
      }
}

class A {
      public void m() {
            System.out.println("a");
      }
}

class B extends A {
      public void m() {
            System.out.println("b");
      }
}

class Impossible{
      public void happened() {
            // "2b||!2b?", whatever the answer nothing happens here
      }
}
```

a. Using the **minimal number of nodes** (hint: 11), draw a Control Flow Graph (CFG) for method M.m() and include it in your submission. The CFG should be at the level of basic blocks. Using any computer diagrams or drawing tools such as Paint, Google Draw, MS Visio, Photoshop, etc. Be sure to label all nodes and edges, and include proper code sections in each of the nodes. Submit it as a PDF image. See the lecture notes on Structural Coverage and CFG for examples

b. List the sets of Test Requirements (TRs) with respect to the CFG you drew in part (a) for each of the following coverages: node coverage; edge coverage; edge-pair coverage; and prime path coverage. In other words, write four sets: TR_{NC}, TR_{EC}, TR_{EPC}, and TR_{PPC}. If there are infeasible test requirements, list them separately and explain why they are infeasible

c. Using q3/TestM-skeleton.java as a starting point, write one JUnit Test Class that achieves, for method M.m(), each of the following coverages:
   i. node coverage but not edge coverage;
   ii. edge coverage but not edge-pair coverage;
   iii. edge-pair coverage but not prime path coverage; and
   iv. prime path coverage.

   In other words, you will write four test sets (groups of JUnit test functions) in total. One test set satisfies (i), one satisfies (ii), one satisfies (iii), and the last satisfies (iv), if possible. If it is not possible to write a test set to satisfy (i), (ii), (iii), or (iv), explain why. For each test written, provide simple documentation in the form of a few comment lines above the test function, listing which TRs are satisfied by that test. Consider feasible test requirements only for this part