Exercise 1

```
1. void f() {
2.     float x;
3.     float y;
4.     x = read();
5.     y = read();
6.     if (x > 0)
7.       x += 10;
8.     y = y / x;
9.     write(x);
10.    write(y);
11. }
```

- Create the control flow graph of this program
- Write the test cases using the following coverage criteria
  - Node coverage
  - Edge coverage
- Do they identify the fault ?

Note well:

- Create one node for each atomic statement
- Compound statements must be divided into atomic statements
- Create an edge from node N1 to node N2 if during the execution the corresponding statements can be executed one immediately after the other

Exercise 2

The following Java function returns the highest absolute value in an array of maximum 5 integers (and minimum 1), -1 if bad input (too large array, empty array, etc..).

```
1.  public int max_absolute(int[] numbers){
2.          if(numbers.length > 5)
3.                  return -1;
4.          int max_value = 0;
5.          for(int i = 0; i<numbers.length; i++){
6.                  if (numbers[i] < 0 )
7.                          max_value = Math.max(max_value,Math.abs(numbers[i]));
8.                  else    max_value = Math.max(max_value, numbers[i]);
9.          }
10.         return max_value;
11. }
```

- Draw the control graph;
- The function is tested with the following test cases (separately):
  for each or them, define statement, branch (edge) and loop ratio coverage.

```
int[] all_equals = {0,0,0,0,0};    // T1 - (0)
int[] all_positive = {1,2,3,4,5};   // T2- (5)
int[] all_negative = {-1,-2,-3,-4,-5};  // T3- (5)
int[] out_of_size = {1,2,3,4,5,6};    // T4- (-1)
int[] mixed = {-10,10,3,5,-6};  // T5 - (10)
int[] empty = {};    // T6 - (-1)
```

- Which test case will fail?
- Which combination of tests can assure 100% coverage of (1) statements, (2) branches, (3) loops, and (4) all?

Exercise 3

Assume the following specification for some piece of code which could be part of a bisection algorithm to find π/2:

- Input parameters are the float values *a* and *b*.
- Swap *a* and *b* unless *a* <= *b*.
- Set *a* and *b* to 1 and 3 unless *cos*(*a*) >= 0 or *cos*(*b*) <= 0.
- Set *x* to the arithmetic mean of *a* and *b*.
- Set *a* to *x* if *cos*(*x*) > 0 and *b* to *x* otherwise.
- Print *a* and *b*.

This is the code:

```
1.  if (a > b) {
2.                          float tmp=b; b = a; a = tmp;
3.  }
4.  if (cos(a) < 0 || cos(b) > 0) {
5.                          a = 1; b = 3;
6.  }
7.  x = (a + b) / 2;
8.  if (cos(x) > 0) {
9.                          a = x;
10. } else {
11.                         b = x;
12. }
```

- Write test cases that are able to cover the following situations:

  (A) swap code at line 2

  (B) line 2 is not executed

  (C) line 5 is executed

  (D) line 5 is not executed

  (E) `line 9 is executed`

  (F) Line 11 is executed

Exercise 4

Consider the following requirements and code:

The program shall take as input an array of three integer numbers.

The program shall output the greatest number among the elements of the array.

The program shall order the elements of the array in decreasing order.

```
1.  int[] order(int v[]) {
2.          int tmp;
3.          if (v[0]<v[1]) {
4.                  tmp = v[0];
5.                  v[1] = v[1];
6.                  v[1] = tmp;
7.          }
8.          if (v[1]<v[2]) {
9.                  tmp = v[0];
10.                 v[1] = v[2];
11.                 v[2] = tmp;
12.         }
13.         return v;
14. }
```

- Create the control flow graph
- Write the test cases using the following coverage criteria:
  Node (statement) coverage, Edge (branch) coverage, Path coverage

Exercise 5

Consider the following code:

```
double elbonianTaxCalculator(double income, int nDependents) {
        double TaxSubTotal,Exemption,TaxTotal;

        // first if - check income
        if (income < 0) {
                System.out.println("You cannot have a negative income.\n");
                return -1;
        }

        // second if - check dependents
        if (nDependents <= 0) {
                System.out.println("You must have at least one dependent.\n");
                return -2;
        }

        // third if (else-if) - compute tax subtotal
        if (income < 10000)
                TaxSubTotal = .02 * income;
        else if (income < 50000)
                TaxSubTotal = 200 + .03 * (income - 10000);
        else
                TaxSubTotal = 1400 + .04 * (income - 50000);
        Exemption= nDependents * 50;
        TaxTotal=TaxSubTotal - Exemption;

        // last if - check negative tax
        if (TaxTotal<0) //In case of negative tax
                TaxTotal=0;

        System.out.println( "$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$ \n");
        System.out.println( "Elbonian Tax Collection Agency \n");
        System.out.println( "Tax Bill \n");
        System.out.println( "Citizen's Income: " + income  +'\n');
        System.out.println( "Tax Subtotal: "  + TaxSubTotal +'\n');
        System.out.println( "Number of Dependents: "+ nDependents + '\n');
        System.out.println( "Tax Exepmtion: " + Exemption +'\n');
        System.out.println( "Final Tax Bill: " + TaxTotal + '\n');
        System.out.println( "$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$ \n");

        return TaxTotal;
}
```

Build a test-suite for complete path coverage.

Exercise 6

Consider the following code:

```
1.  float foo (int a, int b, int c, int d, float e) {
2.                                  float e;
3.                                  if (a == 0) {
4.                                          return 0;
5.                                  }
6.                                  int x = 0;
7.                                  if ((a==b) || ((c == d) && bug(a) )) {
8.                                          x=1;
9.                                  }
10.                                 e = 1/x;
11.                                 return e;
12.  }
```

Function bug(a) should return a value of true when passed a value of a=1.

Build:

- a test suite for 100% statement coverage
- a test suite for 100% branch coverage
- a test suite for 100% condition coverage

Exercise 7

A function converts a sequence of chars in an integer number. The sequence can start with a '-' (negative number). If the sequence is shorter than 6 chars, it is filled with blanks (to the left side).
The integer number must be in the range minint = -32768 to maxint = 32767.
The function signals an error if the sequence of chars is not allowed.

```
1. public class ConvertInt {
2.      public int convert(char[] str) throws Exception{
3.            if (str.length > 6)
4.                 throw new Exception();
5.            int number=0;int digit; int i=0;
6.            if (str[0]=='-')
7.                 i=1;
8.            for(; i<str.length; i++){
9.                 digit = str[i] - '\0';
10.                      number = number * 10 + digit;
11.                }
12.                      if (str[0]=='-')
13.                           number =  -number;
14.                      if (number > 32767 || number < -32768)
15.                                          throw new Exception();
16.                return number;
17.                }
18.           }
```

Write tests in order to achieve:

- node coverage
- edge coverage
- condition coverage for all if statements
- multiple condition coverage of if statement at line 14
- path coverage

Exercise 8

A queue of events in a simulation system receives events. Each event has a time tag.
It is possible to extract events from the queue, the extraction must return the event with lower time tag.
The queue discards events with negative or null time tag.
The queue must accept at least 100.000 events.
Events with the same time tag must be merged (i.e. the second received is discarded)
Define test cases to achieve

- Node coverage
- Edge coverage

```
1. import java.util.Iterator;
2. import java.util.LinkedList;
3.
4. public class EventsQueue {
5.             private LinkedList queue;
6.
7.     public EventsQueue(){
8.             queue = new LinkedList();
9.     }
10.                public void insert(int event){
11.                 int index = 0;
12.                 int size = queue.size();
13.                 while (index < size &&
14.                             ((Integer)queue.get(index)).intValue() <
   event){
15.                       index++;
16.                 }
17.                 queue.add(index, new Integer(event));
18.                }
19.                public int pop(){
20.                 Object o = queue.getFirst();
21.                 if (o != null)
22.                 return ((Integer) o).intValue();
23.                 else return -1;
24.                }
25.                public void print(){
26.                 Iterator i = queue.iterator();
27.                 int event;
28.                 while (i.hasNext()){
29.                       event = ((Integer) i.next()).intValue();
30.                            System.out.println(event + " ");
31.                 }
32.                }}
```