# ST 2018 Past Paper

1. An array with an odd number of elements is said to be centered if all elements, other than the middle element, are strictly greater than the value of the middle element. Only arrays with an odd number of elements have a middle element. The function should accept an integer array as input and return 1 if it is centered array, return 0 otherwise.

```
static int centered(int[] a)                          //Line 1
{      int midIndex, middleItem;                      //Line 2
     if (a == null || a.length % 2 == 0) return 0;    //Line 3
     midIndex = a.length / 2 ;                        //Line 4
     middleItem = a[midIndex + 1];                    //Line 5
     for (int i=0;                                    //Line 6
          i<a.length;                                 //Line 7
          i++){                                       //Line 8
          if (i != midIndex && middleItem >= a[i])    //Line 9
             return 0;                                //Line 10
     }
     return 1;                                        //Line 11
}
```

i = 0 1 2 3 4 5 6
a = [3, 2, 1, 0, 1, 2, 3]

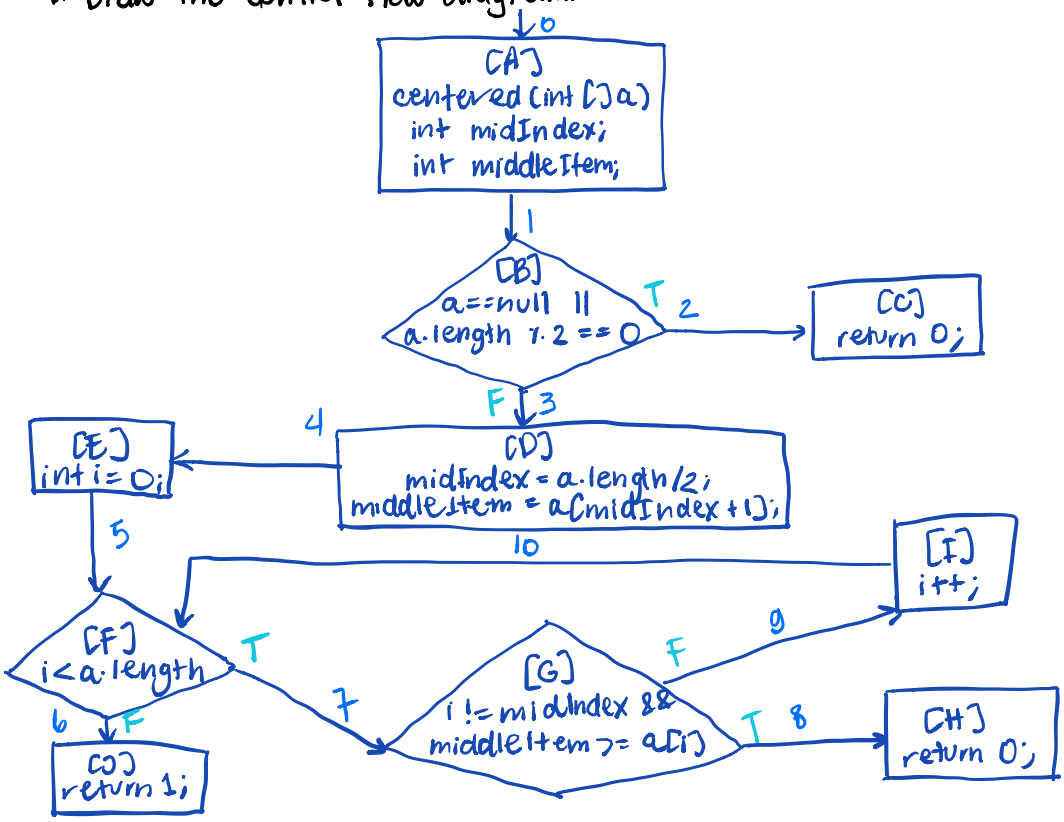
len(a) = 7/2 = 3

↑ midIndex

a. Identify the fault and suggest correct solution.

Line 5 should be middleItem = a[midIndex].

b. Draw the control flow diagram.



c. Write 3 tests with input, expected output, and sequence of edges.

| Test | Input | Expected output | Sequence of edges |
|---|---|---|---|
| a | [3, 2, 2, 3] | 0 | 0, 1, 2 |
| b | [3, 2, 1, 2, 3] | 1 | 0, 1, 3, 4, 5, 7, 9, 10, ..., 6 |
| c | [2, 3, 2] | 0 | 0, 1, 3, 4, 5, 7, 8 |

d. Evaluate the fraction of edges covered by the tests (covered / total).

coverage = 11/11 = 100%. → No additional tests needed.

e. Change Line 9 &&  to || → if (i != midIndex || middleItem >= a[i])
Will the tests reveal this fault? Which coverage criteria is effective in revealing faults in a complex boolean expression?
Develop tests that achieve 100% coverage for this criterria for the faulty expressions shown.
You can reuse tests.

Yes, test (b) will reveal this fault as the expected output is 1 but the actual output is 0.

MC/DC is effective in revealing faults in a complex boolean expression.

| i != midIndex | middleItem >= a[i] | i != midIndex \|\| middleItem >= a[i] |
|---|---|---|
| 1 | 1 | 1 |
| 1 | 0 | 1 |
| 0 | 1 | 1 |
| 0 | 0 | 0 |

f. Name the coverage criteria that is practical for testing paths through loops? For one of these criteria, compute the coverage achieved using the tests developed in c) and d).
If full coverage is not achieved, develop tests to achieve full coverage.

Basic condition coverage → a = {1, 2, 0, 4, 5}

(CSA) adequacy, Loop boundary, boundary interior
every loop is executed ← zero times
exactly once
more than once

Current tests → a → 0 times
b → more than once
c → zero times
Additional → d : a = {3, 2, 3} → exactly once
} 100% loop adequacy

g. Write tests with input and expected output that achieves 'all DU pairs' coverage and write down <D,U> pairs for all the variables in the function (using line numbers) and show coverage achieved.

| Variable | Definitions | Uses | <D,U> pairs |
|---|---|---|---|
| a | 1 | 3, 4, 5, 7, 9 | <1,3>, <1,4>, <1,5>, <1,7>, <1,9> |
| midIndex | 2, 4 | 5, 9 | <4,5>, <4,9> |
| middleItem | 2, 5 | 9 | <5,8> |
| i | 6, 8 | 7, 9 | <6,7>, <6,8>, <6,9>, <8,7>, <8,8>, <8,9> |

The tests from part c achieves 100% coverage.

2. GUI with control elements: 2 checkboxes (unchecked/checked), 1 radio buttons (on/off), 1 text box (1<number<100)

a. How many tests to exhaustively test the input space?

$2 \times 2 \times 2 \times 100 = 800$

b. Define partitions/value classes for the inputs. How many test specs do you get covering all combinations of the value classes of all inputs?

| Characteristic | Partition | Value classes |
|---|---|---|
| Check box | P1 | Checked |
| | P2 | Unchecked |
| Radio buttons | P3 | On |
| | P4 | Off |
| Text box | P5 | Number between 1 and 100 |
| | P6 | Number on the edges of the range |
| | P7 | Number out of range |
| | P8 | Not a Number (NaN) |
| | P9 | Number just out of range |

Total no. of specs = $2 \times 2 \times 5 = 40$

c. Generate pairwise testing specs.

| First checkbox | Second checkbox | Radio button | Text box |
|---|---|---|---|
| checked | unchecked | on | 1-100 |
| unchecked | checked | off | NaN |
| unchecked | unchecked | on | 0/101 |
| checked | unchecked | off | 1/100 |
| unchecked | unchecked | on | 1-100 |
| checked | unchecked | on | NaN |
| checked | checked | off | 0/101 |
| unchecked | checked | on | 1/100 |

d. Give concrete test cases.

| First checkbox | Second checkbox | Radio button | Text box |
|---|---|---|---|
| checked | unchecked | on | 8 |
| unchecked | checked | off | 'a' |
| unchecked | unchecked | on | 0 |
| checked | unchecked | off | 100 |
| unchecked | unchecked | on | 54 |
| checked | unchecked | on | '!' |
| checked | checked | off | 101 |
| unchecked | checked | on | -70 |

e. What is an equivalent mutant? Create 3 mutants using three diff. types of mutation operators, one of them equivalent. Name type of mutation operator. Write a single test and compute the mutation score achieved when it is executed.

```
public static void bubblesort(int[] list){
      int temp;
      boolean changed = true;
      if (list == null || list.length == 0)
         System.out.println("Invalid list of numbers.")
      for(int i = 0; i < list.length && changed; i++)
      {       changed = false;
              for(int j = 0; j < list.length - i - 1; j++)
              {
                      if(list[j] > list[j+1])
                      {       changed = true;
                              temp = list[j + 1];
                              list[j + 1] = list[j];
                              list[j] = temp;
                      }
               }
       }
}
```

Equivalent mutant → a change of code that cannot be detected by any tests, because it does not change the behaviour of the system in any way.

- boolean changed = false
  → constant value replacement
- for (int i = 1; i < list.length && changed...
  → scalar variable replacement
- if ( ! (list[j] <= list[j+1]) )
  → expression modification
  → equivalent mutant

Test = [3, 2, 4, 1] → kills 2/3 mutants = 66.7% coverage

f. Why is regression testing time consuming? Describe techniques available to optimise regression testing.

After every new feature is added, the test suite has to be expanded to be a combination of all tasks suitable for the previous version of the system with the tests added to verify the new feature.

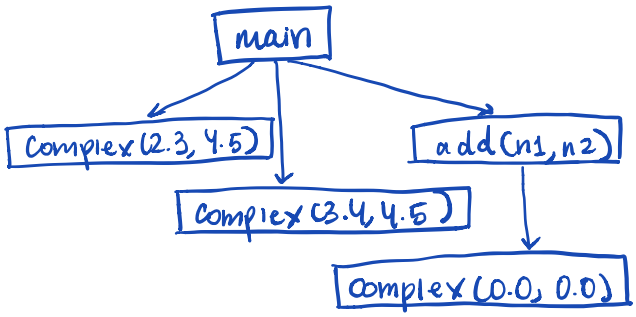
- Regression test selection
  → from entire test suite, only select subset of test cases whose execution is important / relevant to changes.

- Regression test set minimisation
  → identify redundant test cases and remove them from the test suite to reduce its size.

- Regression test set prioritisation
  → sort test cases in order of increasing cost per additional coverage

3. a. Draw a context-sensitive call graph for the following class.

```
public class Complex {

    double real;
    double imag;

    public Complex(double real, double imag) {
        this.real = real;
        this.imag = imag;
    }

    public static void main(String[] args) {
        Complex n1 = new Complex(2.3, 4.5),
                n2 = new Complex(3.4, 5.0),
                temp;

        temp = add(n1, n2);

        System.out.printf("Sum = %.1f + %.1fi", temp.real, temp.imag);
    }

    public static Complex add(Complex n1, Complex n2)
    {
        Complex temp = new Complex(0.0, 0.0);

        temp.real = n1.real + n2.real;
        temp.imag = n1.imag + n2.imag;

        return(temp);
    }
}
```




b. A system has a failure rate of $4 \times 10^{-3}$ failures/hours. What is the Mean Time to Failure (MTTF) or Expected Life?

$$MTTF = E(t_f) = \int_0^\infty R(t)dt = \frac{1}{\lambda} \quad \text{where } \lambda \text{ is the failure rate}$$

$$= 1/4 \times 10^{-3} = \frac{1}{4} \times 10^3 = 250 \text{ hours}$$

c. A component has a failure rate of 5 failures/$10^6$ hours. What is the reliability of the component after 10000 hours of operation?

$$R_K(t) = e^{-\lambda_K t}$$

$$= e^{-5/10^6 \cdot 10000} \approx 0.951$$

d. Reach equations for line 16 using reach and reachout equations for lines 15, 14, and 13 for the following method:

```
  public class Sample {
1    public static void main(String[] args) {
2        int m = Integer.parseInt(args[0]);    // choose this many elements
3        int n = Integer.parseInt(args[1]);    // from 0, 1, ..., n-1
4
        // create permutation 0, 1, ..., n-1
5        int[] perm = new int[n];
6        for (int i = 0;
7             i < n;
8             i++)
9               perm[i] = i;

        // create random sample in perm[0], perm[1], ..., perm[m-1]
10       for (int i = 0;
11            i< m;
12            i++)  {

            // random integer between i and n-1
13            int r = i + (int) (Math.random() * (n-i));

            // swap elements at indices i and r
14            int t = perm[r];
15            perm[r] = perm[i];
16            perm[i] = t;
17       }
18       for (int i = 0;
19            i < m;
20            i++)
21              System.out.print(perm[i] + " ");
22       System.out.println();
23   }
24 }
```

Reach(16) = ReachOut(15)

ReachOut(15) = (Reach(15) - perm[r] g) ∪ (perm[r] 15)

Reach(15) = ReachOut(14)

ReachOut(14) = (Reach(14) - t 14) ∪ (t 14)

Reach(14) = ReachOut(13)

ReachOut(13) = (Reach(13) - r 13) ∪ (r 13)

?

e. Discusss 3 security vulnerabilities + why traditional testing techniques are inadequate for ensuring software security + 2 testing techniques for security requirements.

Security vulnerabilities

1. Buffer overflows - overwriting return address, hijacking control
2. Race conditions - execution of program depends on which process finishes first
3. Input validation errors - email injection, HTTP header injection

* Traditional testing do not assume that the user will be a malicious user who will attempt to break a program on purpose.

Testing techniques for security requirements

1. Penetration testing - using tester's expertise to test the system against common malicious behaviours.
2. Fuzz-testing - sending semi-valid input to the program and observing its behaviour.

f. For the following specification, illustrate steps involved in TDD. As part of the steps, the code that you write can be in the form of pseudo-code or a programming language of your choice. For the purpose of the exam, do not worry about the syntactical correctness of the code. You only need to focus on program logic and its correspondence to tests developed.

```
Take an integer number as input.
Spec 1: If input is negative print "error".
Spec 2: Print the word "fizz" if the number is a multiple of 5.
Spec 3: Print the word "buzz" if the number is a multiple of 7.
Spec 4: Print the word "fizzbuzz" if the number is a multiple
        of both 5 and 7.
Spec 5: Print the number if it is not a multiple of 5 or 7.
```

Test 1: If input is negative, print "error".
Input = -3 Expected output = "error"

Code Stage 1:
```
def fizzbuzz(n):
    if n<0:
        print("error")
```

Test 2: If input is a multiple of 5, print "fizz"
Input = 10 Expected output = "fizz"

Code Stage 2:
```
def fizzbuzz(n):
    if n<0:
        print("error")
    elif (n%5==0):
        print("fizz")
```

Test 3: If input is a multiple of 7, print "buzz"
Input = 21 Expected output = "buzz"

Code Stage 3:
```
def fizzbuzz(n):
    if n<0:
        print("error")
    elif (n%5==0):
        print("fizz")
    elif (n%7==0):
        print("buzz")
```

Test 4: If input is a multiple of 5 AND 7, print "fizzbuzz"
Input = 35 Expected output = "fizzbuzz"

Code Stage 4:
```
def fizzbuzz(n):
    if n<0:
        print("error")
    elif (n%5==0 && n%7==0):
        print("fizzbuzz")
    elif (n%5==0):
        print("fizz")
    elif (n%7==0):
        print("buzz")
```

Test 5: If input is not a multiple of 5 or 7, print the number.
Input = 6 Expected output = 6

Code Stage 5:
```
def fizzbuzz(n):
    if n<0:
        print("error")
    elif (n%5==0 && n%7==0):
        print("fizzbuzz")
    elif (n%5==0):
        print("fizz")
    elif (n%7==0):
        print("buzz")
    else:
        print(n)
```