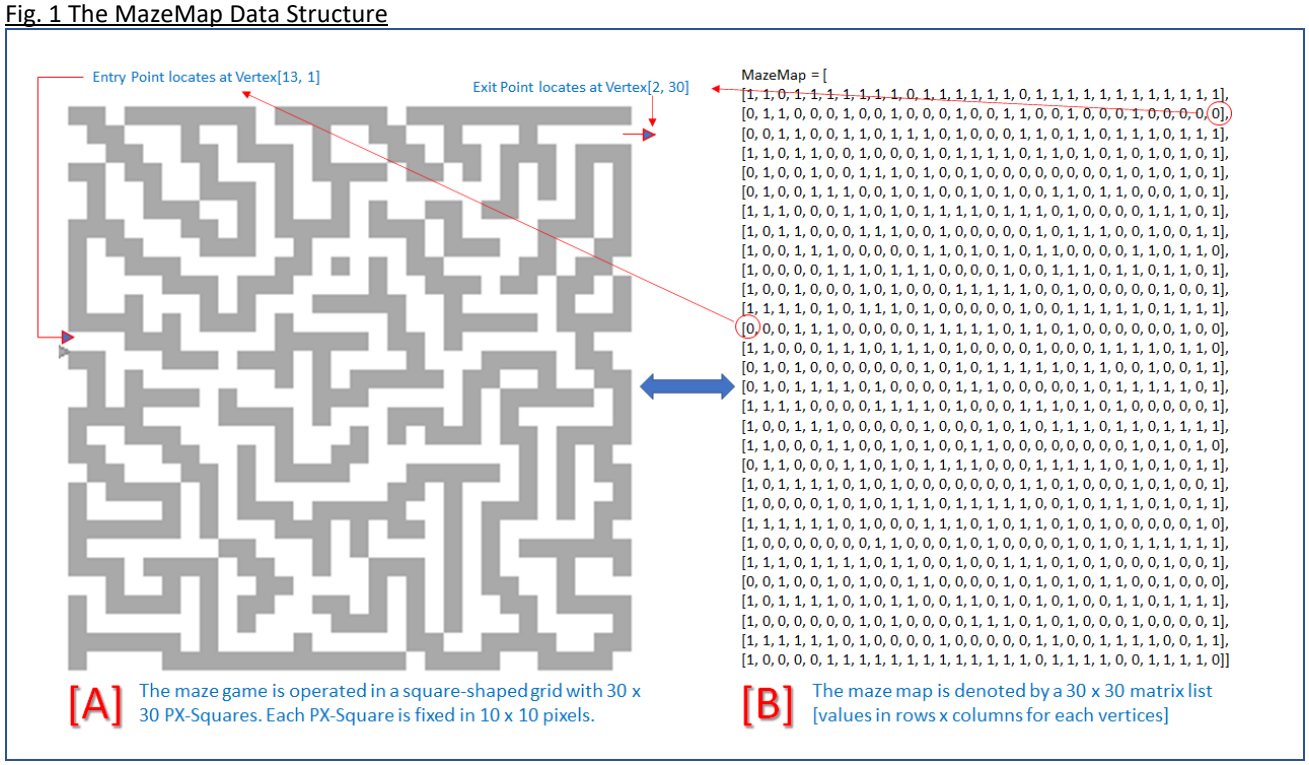
Project Title: Tom and Jerry in maze game
Project ID: TnJ1

Story: “Tom and Jerry” is an American cartoon film about a halpless cat’s never-ending pursuit of a clever mouse. Tom is the scheming cat, and Jerry is the spunky mouse.

Introduction to the TnJ1 Maze Game

1. The dimension of the electronic play board is 300 x 300 = 90,000 pixels.
2. The maze game is displayed in a square-shaped grid with 30 x 30 PX-Squares filled by dark grey colour.
3. The maze graph is translated in a 30 x 30 matrix list (Fig.1 [B] – MazeMap).
4. PX-Square is a vertex of the maze graph, where vertex size = 10 x 10 pixels.
5. Each vertex address and value are expressed in {Boolean Vertex[r,c] where r=row, c=column. If Vertex[r,c]=1, it is a barrier filled by dark grey colour, else Vertex[r,c]=0, it is a clear path filled by white colour.
6. There will have only one Entry-Point e.g. Vertex[13,1] located at the left-side border of the maze.
7. There will have only one Exit-Point e.g. Vertex[2,30] located at the right-side border of the maze.
8. For each maze, it should have at least 2 possible paths that Tom and Jerry can traverse between Entry-Point and Exit-Point along all possible paths of the maze.

Fig. 1 The MazeMap Data Structure



The maze game is divided into three functions that requires each project group to complete the implementation.

Function A: Build a maze

1. For COMP3111 students, choose either one of below tasks:

   i) Write a program of Maze-Editor that allows user to build a maze manually. See Appendix 2.1 for a reference, or, you may feel free to use your better idea to complete this task.

   ii) Write a program of Maze-Generator based on MST (Minimum Spanning Tree) to automatically generate a maze map data file at random time (Fig.1 [B]). A sample skeleton code that used DFS (Depth First Search) algorithm is provided. Note that the output of maze graph is displayed in text format, denoted by a text character(*). You need to translate to GUI type using PX-Squares as a sample output shown in Fig 1 [A].

2. For COMP3111H students, choose either one of below tasks:

   i) Write a program of Maze-Editor that allows user to build a maze manually. See Appendix 2.1 for reference as a minimum requirement. You should design your own way to enhance the program higher than standard.

   ii) Write a program of Maze Generator based on MST (Minimum Spanning Tree) to automatically generate a maze map data file at random time (Fig.1 [B]). Sample skeleton code is not provided.

3. Whichever task i) or ii) is chosen, finally you should produce MazeMap file saving as csv file format delimited with "comma" likes Fig.3 [B] and sketch the GUI maze graph likes Fig.1 [A] on your laptop.

4. A sample skeleton code on how to create the GUI interface for an object moving around a 20 x 20 PX-Squares grid play board – The Snake Game (See Appendix 1.1) is provided. You may clone the git repository onto your laptop and run it for a trial, and copy part of the code into your group project.

Function B: Shortest Path

To write a smart maze game program, the use of tree traversal algorithms to find possible paths and shortest path(s) between two vertices is indispensable. When a maze map file is created, see Fig. 2, two vertex-locations of Entry-Point.Vertex[13,1] and Exit-Point.Vertex[2,30] are formed.

1. Fig.2 [A] : Show the GUI maze graph with green Shortest Path(s) and yellow Other Possible Paths.
2. Fig.2 [B] : Export all path data in the format of {P(n), v1, v2, v3, ……, v(pl)}

Where P=path, n=path number, v=vertex, pl=path length=number of vertices for the path, P(n) is string, v1 to v(pl) are written as (r,c) where r=row and c=column of the maze grid. E.g. [s1, (15,9), (14,9), (13,9), ……, (2,30)]

Fig.2 Illustration of Shortest Path (This maze map data file can be found on Canvas for reference only.)

[A] Screen layout of finding Possible Paths & Shortest Path in a Maze Game

[B] Examples of data output for 2 Shortest Paths and 3 Other Possible Paths.

Paths from MathFile[13,1] to MathFile[2,30]

Shortest Paths	s1 = {	13,1	13,2	13,3	14,3	14,4	14,5	15,5	15,6	15,7	15,8
	Length (Vertices) = 39	15,9	14,9	13,9	13,10	13,11	12,11	11,11	11,12	11,13	10,13
		10,14	9,14	8,14	8,15	8,16	7,16	6,16	5,16	5,17	5,18
		5,19	5,20	5,21	5,22	6,22	7,22	7,23	7,24	7,25	6,25
		6,26	6,27	5,27	4,27	3,27	2,27	2,28	2,29	2,30	}
	s2 = {	13,1	13,2	13,3	14,3	14,4	14,5	15,5	15,6	15,7	15,8
	Length (Vertices) = 39	15,9	14,9	13,9	13,10	13,11	12,11	11,11	11,12	11,13	10,13
		10,14	10,15	10,16	9,16	8,16	7,16	6,16	5,16	5,17	5,18
		5,19	5,20	5,21	5,22	6,22	7,22	7,23	7,24	7,25	6,25
		6,26	6,27	5,27	4,27	3,27	2,27	2,28	2,29	2,30	}
Other Possible Paths	p1 = {	13,1	13,2	13,3	14,3	14,4	14,5	15,5	15,6	15,7	15,8
	Length (Vertices) = 55	15,9	15,10	16,10	16,11	16,12	16,13	17,13	18,13	18,14	18,15
		17,15	17,16	17,17	16,17	16,18	16,19	16,20	16,21	15,21	14,21
		14,20	13,20	12,20	11,20	11,19	10,19	10,18	9,18	8,18	8,17
		8,16	7,16	6,16	5,16	5,17	5,18	5,19	5,20	5,21	5,22
		6,22	7,22	7,23	7,24	7,25	6,25	6,26	6,27	5,27	4,27
		3,27	2,27	2,28	2,29	2,30	}				
	p2 = {	13,1	13,2	13,3	14,3	14,4	14,5	15,5	15,6	15,7	15,8
	Length (Vertices) = 57	15,9	15,10	16,10	16,11	16,12	16,13	17,13	18,13	18,14	18,15
		17,15	17,16	17,17	16,17	16,18	16,19	16,20	16,21	15,21	14,21
		14,22	13,22	13,23	13,24	13,25	13,26	12,26	11,26	11,25	11,24
		11,23	10,23	9,23	9,24	8,24	7,24	7,25	6,25	6,26	6,27
		5,27	4,27	3,27	2,27	2,28	2,29	2,30	}		
	p3 = {	13,1	13,2	13,3	14,3	14,4	14,5	15,5	15,6	15,7	15,8
	Length (Vertices) = 63	15,9	14,9	13,9	13,10	13,11	12,11	11,11	11,12	11,13	10,13
		10,14	10,15	10,16	9,16	8,16	8,17	8,18	9,18	10,18	10,19
		11,19	11,20	12,20	13,20	14,20	14,21	14,22	13,22	13,23	13,24
		13,25	13,26	12,26	11,26	11,25	11,24	11,23	10,23	9,23	9,24
		8,24	7,24	7,25	6,25	6,26	6,27	5,27	4,27	3,27	2,27
		2,28	2,29	2,30	}						

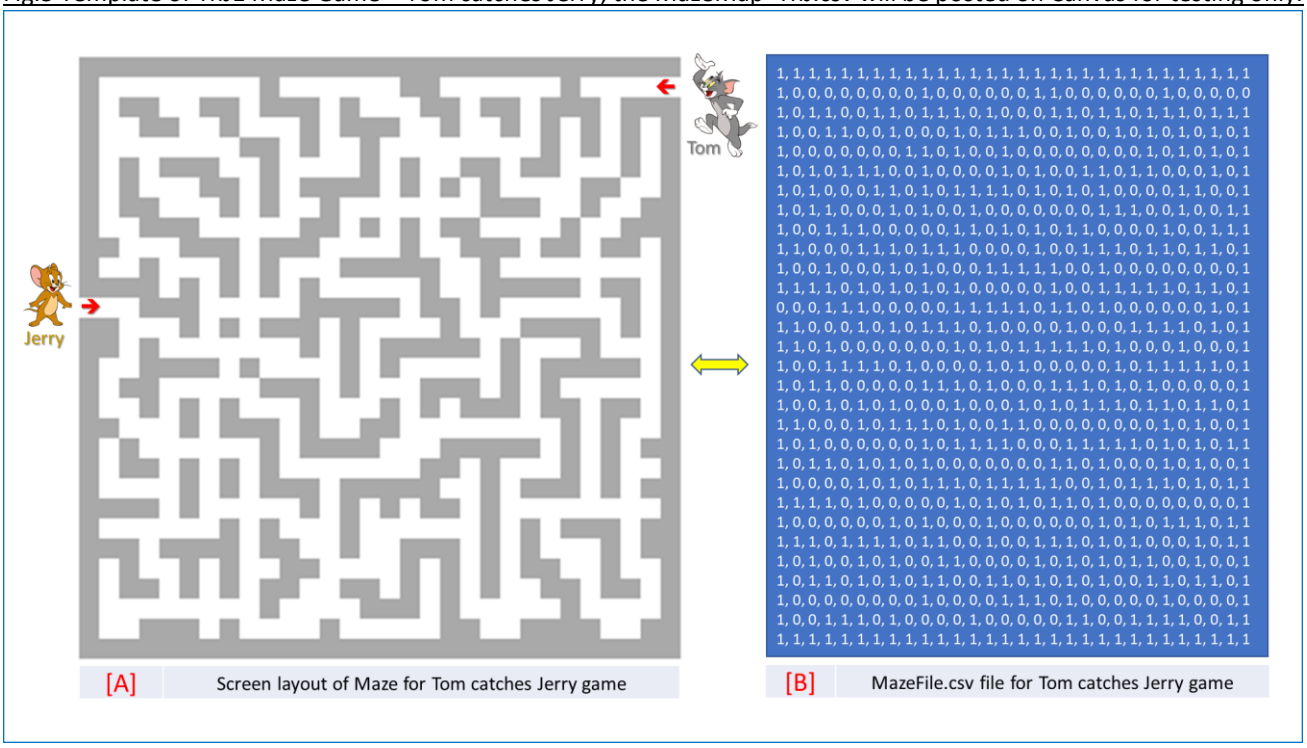
1. For COMP3111 students, do shortest path only, show and output one of the shortest path.
2. For COMP3111H students, do & show all possible paths. Additionally, use 2 algorithms (e.g. DFS vs BFS) to run ONE of the possible paths, e.g. p3 in Fig.2. To compare 2 outputs from 2 algorithms, write your comments for any findings of searching results.

Function C – Tom catches Jerry

This is an interactive game between user and computer with the following rules of operation:

1. Jerry is controlled by the computer that solution from Function B – Shortest Path must be reused in this part of coding for below purposes:
    a. To find the fastest speed reaching the Exit-Point of the maze.
    b. To avoid catching by Tom along paths in the maze.
2. Tom is controlled by user.
3. When the game begins, Jerry starts at Entry-Point and runs along the shortest path to reach the Exit-Point, while Tom starts at Exit-Point and runs approaching any paths to catch Jerry.
4. If Tom caught Jerry successfully, user wins the game and program stops.
5. If Jerry reached Exit-Point, user loses, the game is over.
6. In normal circumstance, Tom's running speed is always a bit faster than Jerry.

Fig.3 Template of TnJ1 Maze Game – Tom catches Jerry, the MazeMap_TnJ.csv will be posted on Canvas for testing only.



1. For COMP3111 students, they can use Fig.3 template of maze map data and to simply follow the rules of operation from item 1 to 6 listed in the first paragraph, to implement the interactive Tom catches Jerry game.
2. The COMP3111H students:
    a. They can also use the above template of maze map data or design their own maze map, in order to make it more sophisticated with funs.
    b. Must follow the rules of operation from item 1 to 6 listed in the first paragraph, plus below enhanced features:
        i. Add an object "Crystal" at a fixed location inside the maze map.
        ii. If Jerry went to get the Crystal successfully, the role-play including running speed between Tom & Jerry will be swapped within a fixed time stream, for instance 10 seconds or a traversal of 20. That is, Jerry catches Tom!
        iii. When Jerry gets the crystal, Jerry will choose whether goes to catch Tom or goes toward the Exit-Point, subject to which way has the highest opportunity to win the game.

Function D – Unit Testing

You are required to hand-craft unit test cases using JUnit to test your implemented code. Your unit test cases should cover as many lines as possible.

The grading metric for the submitted test cases is related to line coverage. Specifically, the submitted test cases that have the least number of uncovered lines get the highest score. Specifically, the grading for Function D consists of two parts.

Part A: the submitted test cases that achieve more than 20% line coverage get 50% of the total score of Function D.

Part B: For the remaining 50% total score, the grade that a submission can get is $50\% * Total\ Score\ of\ Function\ D * (1 - \frac{rank - 1}{Total\ number\ of\ student\ groups})$. Particularly, the submitted test cases that have the least number of uncovered lines get the best rank (i.e., 1). Similarly, submitted test cases that have the greatest number of uncovered lines ranked the last.

If there are multiple submitted test cases having the same number of uncovered lines, we further rank these submitted test cases according to *Test-per-function ratio* = $\frac{number\ of\ test\ cases}{max(160 - max(number\ of\ functions\ in\ the\ code, 80), 1)}$. Submissions with a lower *Test-per-function ratio* get better rank. In short, the grading scheme encourages you to use fewer test cases to cover as many lines of code as possible (Test minimization is always a good practice). Also, the denominator encourages you to implement at most 80 functions for your project.

We have three requirements for the test cases and project code:

- Only lines of one function called by a test case will be considered covered (see Illustration 1)
- Only lines of function directly called by a test case will be considered covered (see Illustration 2)
- The implemented code should have at most one statement in each line (see Illustration 3)

Failure to achieve the above requirements will result in marks deduction.

Illustrations:

1. Only lines of one function called by a test case will be considered covered. If multiple functions (including constructors) are called in a test case, only one function's covered lines can be considered, and students are required mark which function is considered. For instance, testmultiple_legitimate() in Figure 2.2.2 calls multiple functions: *findMin()* and *finMax().* The function *findMax()* has been marked with "*target function"*, meaning that students want the covered line of *finMax()* to be considered. On the other hand, testmultiple_illegitimate() also calls *findMin()* and *finMax()*, yet no function is marked with "*target function"*. There will be mark deduction and the lines covered by testmultiple_illegitimate() will not be considered.

2. "Uncovered lines" refers to lines explicitly marked "uncovered" by JUnit (i.e., lines with red bars nearly their line number). Lines that are marked "covered" (i.e., lines with green bars nearly their line number in Figure 2.2.3 and Figure 2.2.4) are not considered "uncovered", except that the "covered lines" are from a function not directly called by a test case. For instance, Figure 2.2.3 shows that after exercising testfindMax(), lines of findMax() and findMin() are marked "covered" by JUnit. However, we only consider the lines of findMax() are covered, and all lines of findMin() are not covered.

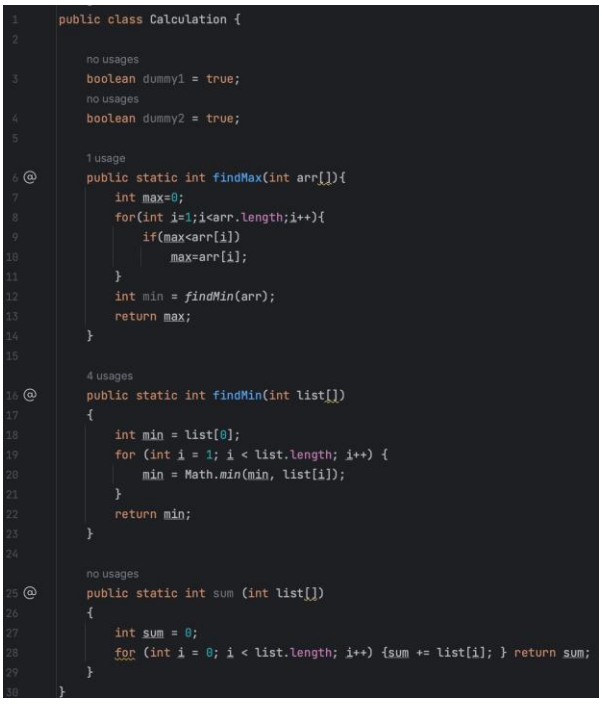
Lastly, lines that do not have any marking (e.g., line 11 or 21 in Figure 2.2.3 and Figure 2.2.4) are not considered uncovered.

3. To objectively evaluate the test cases' effectiveness, the implemented code should have at most one statement in one line. For instance, for loop or if condition predicates should be separated from other statements. A counterexample is shown in Figure 2.2.1, where line 28 consists of both a loop predicate and the rest of the statements; a correct practice is shown in line 8-10. Moreover, in line 28, *sum += list[i]* and *return sum* should be put in two different lines.

Task:

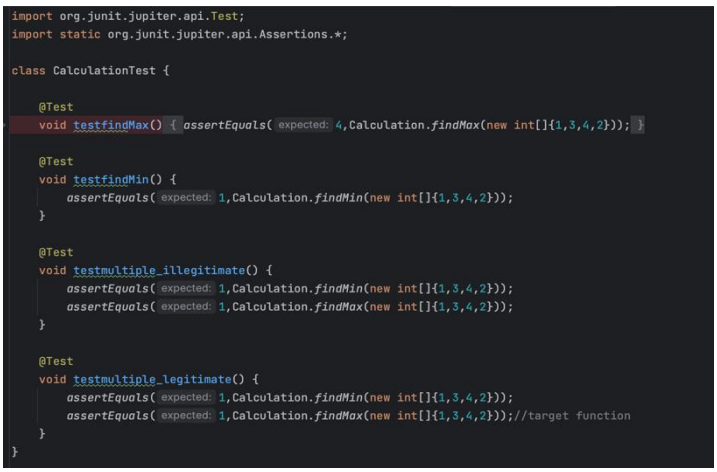
Write a brief report to 1) show the line coverage report generated by JUnit, 2) put down which lines are not covered by each test case (e.g., provide the line numbers), 3) describe which lines are never covered by any test case, the total number of these uncovered lines, as well as the Test-per-function ratio (shows how it is computed). We will reproduce the reported result, and any inconsistency found will result in significant mark deduction. Please refer to our presentation video about Function D for more details.

Figure 2.2.1: Implemented code



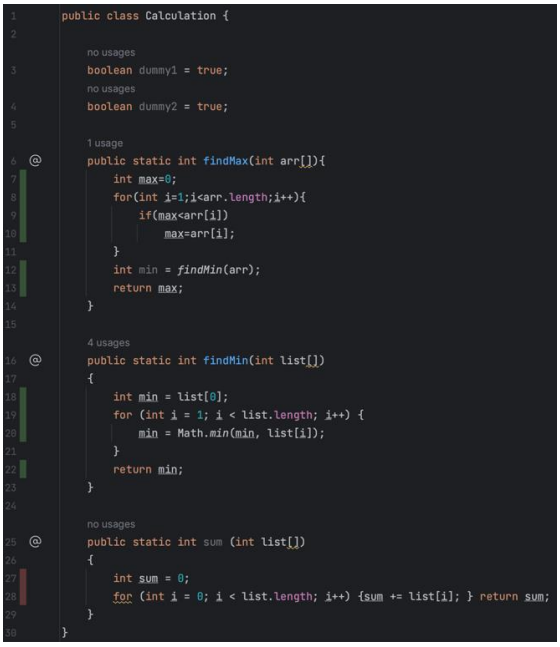
```
public class Calculation {

    no usages
    boolean dummy1 = true;
    no usages
    boolean dummy2 = true;

    1 usage
    public static int findMax(int arr[]){
        int max=0;
        for(int i=1;i<arr.length;i++){
            if(max<arr[i])
                max=arr[i];
        }
        int min = findMin(arr);
        return max;
    }

    4 usages
    public static int findMin(int list[])
    {
        int min = list[0];
        for (int i = 1; i < list.length; i++) {
            min = Math.min(min, list[i]);
        }
        return min;
    }

    no usages
    public static int sum (int list[])
    {
        int sum = 0;
        for (int i = 0; i < list.length; i++) {sum += list[i]; } return sum;
    }
}
```

Figure 2.2.2: Implemented test cases



```
import org.junit.jupiter.api.Test;
import static org.junit.jupiter.api.Assertions.*;

class CalculationTest {

    @Test
    void testfindMax() { assertEquals( expected: 4,Calculation.findMax(new int[]{1,3,4,2})); }

    @Test
    void testfindMin() {
        assertEquals( expected: 1,Calculation.findMin(new int[]{1,3,4,2}));
    }

    @Test
    void testmultiple_illegitimate() {
        assertEquals( expected: 1,Calculation.findMin(new int[]{1,3,4,2}));
        assertEquals( expected: 1,Calculation.findMax(new int[]{1,3,4,2}));
    }

    @Test
    void testmultiple_legitimate() {
        assertEquals( expected: 1,Calculation.findMin(new int[]{1,3,4,2}));
        assertEquals( expected: 1,Calculation.findMax(new int[]{1,3,4,2}));//target function
    }
}
```

Figure 2.2.3: Coverage result A of testfindMax()



```
public class Calculation {

    no usages
    boolean dummy1 = true;
    no usages
    boolean dummy2 = true;

    1 usage
    public static int findMax(int arr[]){
        int max=0;
        for(int i=1;i<arr.length;i++){
            if(max<arr[i])
                max=arr[i];
        }
        int min = findMin(arr);
        return max;
    }

    4 usages
    public static int findMin(int list[])
    {
        int min = list[0];
        for (int i = 1; i < list.length; i++) {
            min = Math.min(min, list[i]);
        }
        return min;
    }

    no usages
    public static int sum (int list[])
    {
        int sum = 0;
        for (int i = 0; i < list.length; i++) {sum += list[i]; } return sum;
    }
}
```

Figure 2.2.4: Coverage result B of testfindMax()

```
public class Calculation {

    no usages
    boolean dummy1 = true;
    no usages
    boolean dummy2 = true;

    1 usage
    public static int findMax(int arr[]){
        int max=0;
        for(int i=1;i<arr.length;i++){
            if(max<arr[i])
                max=arr[i];
        }
        int min = findMin(arr);
        return max;
    }

    4 usages
    public static int findMin(int list[])
    {
        int min = list[0];
        for (int i = 1; i < list.length; i++) {
            min = Math.min(min, list[i]);
        }
        return min;
    }

    no usages
    public static int sum (int list[])
    {
        int sum = 0;
        for (int i = 0; i < list.length; i++) {sum += list[i]; } return sum;
    }
}
```

Appendix 1.1 – Snake Game

You are given a complete skeleton code of Snake Game. The Snake Game program is an open source developed by Hexadeciman Copyright © 2017 with MIT license.

The Game is simply constructed with 3 Java API packages. It works in the environment of 2D GUI.

API Package	Description	Usages in the game program
java.awt	Java Abstract Window Toolkit is an API to develop Graphical User Interface (GUI) or windows-based applications in Java.	Control keypressed, set colour.
java.util	Java Utility contains the collections framework, legacy collection classes, event model, date and time facilities, internationalization, and miscellaneous utility classes.	Basic function of ArrayList, Random-number generator.
javax.swing	Java Swing is a part of Java Foundation Classes (JFC) that is used to create window-based applications. It is built on the top of AWT (Abstract Windowing Toolkit) API and entirely written in java.	Setup of SquarePanel for 1) Playing Board; 2) Snake; 3) Apple.

These API packages are built-in tools of Java 17 and are lightweight components. The programs are developed with fully object-oriented classes.

Click here to clone the GitHub skeleton code into your local IntelliJ Java Project.
More URL for learning Java APIs: java.awt, java.util, javax.swing

Introduction to Snake Game

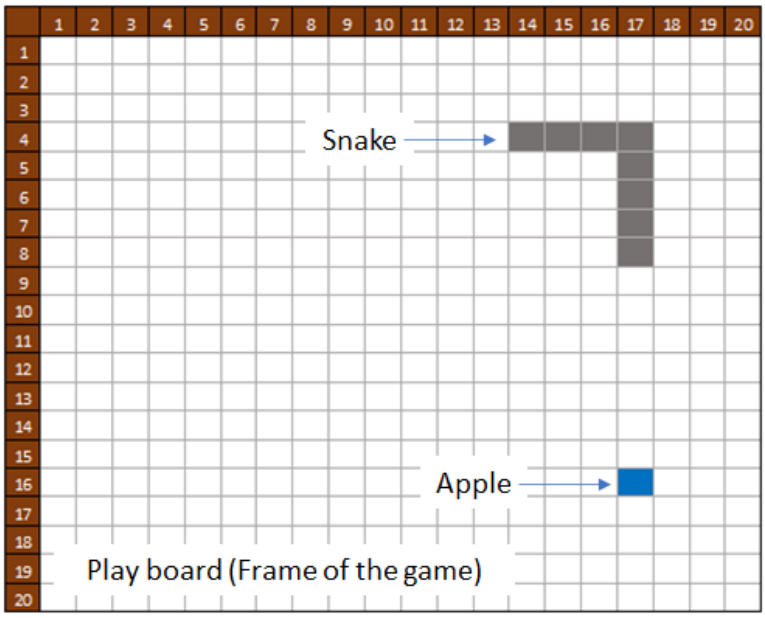
The Snake Game was first created in late 70s. In this game the player controls a snake with 4 cursor keys for 4 moving directions:

1. Pressing Up-Arrow to turn the snake moving upward;
2. Pressing Down-Arrow to turn the snake moving downward;
3. Pressing Left-Arrow to control the snake to make a left-turn;
4. Pressing Right-Arrow to control the snake to make a Right-turn;

The objective of the game is to control the snake to eat as many apples as possible. Each time the snake eats an apple its body grows. The size of each joint of a snake is 10x10 pixels. Initially, the snake has three joints. The snake must avoid the touch of its own body. If a touch is made, the game is over.

There are 3 objects running in the game. Each object is constructed with a standard component of a PX-Square (10x10 pixels)

1. The electronic play board is constructed by 20 x 20 PX-Squares in white colour.
2. The snake is constructed by 3 to many joints in dark grey colour. Each joint is equal to exactly 1 PX-Square.
3. The apple is constructed by exactly 1 PX-Square in blue colour.



Appendix 2.1 – Maze Editor (Example)

One of the idea to create a Maze Editor:

1. Build a grid editor with 30 x 30 PX-Squares, R (row) = 30, C (column) = 30
2. Initiate a plain grid:
   { Boolean MazeMap[int R, int C]
   for i in (1, 30)
   for j in (1, 30)
   MazeMap[i, j] := 0 }
3. When user moves the mouse and clicks on a particular cell e.g. MazeMap[6,10]:
   { if the cell is not filled with dark grey colour, i.e. MazeMap[6,10]=0;
   fill the cell with dark grey colour and set MazeMap[6,10]=1;
   else
   fill the cell with dark white colour and set MazeMap[6,10]=0 }

Here we introduce the most simply algorithm: "Turn On and Turn Off" approach for user's operation. Where On=1=Grey colour filled, Off=0=Colour unfilled. For every click, user turns on to fill colour or turns off to clear the filled colour.

Java API tools for reference: Java GridLayout or Java Swing.