**CS 1120 Computer Science II (with Java),** *Spring 2019*

Department of Computer Science, Western Michigan University

Instructor: *Fola Olagbe* Lab TA: *Hisham Saleh*

Ryan Hopkins

**SOFTWARE LIFE CYCLE REPORT – FOR LAB ASSIGNMENT 2**

# PHASE 1: SPECIFICATION

Create an application Maze Solver that can:

1. Generate a maze (a 2d array of strings)
   1. Number of rows and columns are given by the user and must be in the range [5,10]. Number of rows cannot equal number of columns.
   2. Must validate input. If input is invalid, repeatedly ask for valid input.
2. Randomly fill the maze with numbers in the range [1,100] and #’s.
   1. The number is the amount of coins the robot can pick up by passing through that cell.
   2. A ‘#’ represents a blocked position the robot cannot pass through.
   3. ‘S’ is the starting point for the robot and is at the top left corner of the maze.
   4. ‘D’ is the robot’s final destination (the end of the maze) and is at the bottom right corner of the maze.
   5. **The total number of #’s cannot be greater than a third of the total number of positions in the maze.**
3. Print out the maze to the screen once it is generated.
4. Attempt to find an available path from S to D.
   1. The robot can move up, right, left, or down.
   2. Use A\* pathfinding algorithm.
   3. **This program must use recursion.**
5. If the robot is not able to find a path, print a corresponding message to the screen.
6. If the robot finds a path, then:
   1. Print out the maze with the path marked by +’s.
   2. Sum up all the numbers on this path and print out the result as the number of coins found.
7. **This program is required to run steps 2 – 6 above 3 times.**

**Example output:**

Enter number of rows in range [5, 10]:

1

Invalid input!

Enter number of rows in range [5, 10]:

a

Input must be an integer. Re-enter:

5

Enter number of columns in range [5, 10].

This must be different from number of rows:

5

Number of columns is the same as number of rows!

Enter number of columns in range [5, 10].

This must be different from number of rows:

6

\*\*\*\*\*\*Maze #1\*\*\*\*\*\*

Start drawing the maze...

The maze is as below:

S # 6 # 4 100

27 78 # # 51 #

27 # # # 3 #

47 29 47 # 53 #

32 29 48 56 47 D

Congratulations! I found a solution for this maze as below:

S # 6 # 4 100

+ 78 # # 51 #

+ # # # 3 #

+ 29 47 # 53 #

+ + + + + D

The amount of coins collected: 313

\*\*\*\*\*\*Maze #2\*\*\*\*\*\*

Start drawing the maze...

The maze is as below:

S # # # 86 47

83 # 22 # 34 #

# 24 48 # # 17

# 31 # 24 67 1

21 77 99 87 95 D

Sorry, no solution can be found for this maze!

\*\*\*\*\*\*Maze #3\*\*\*\*\*\*

Start drawing the maze...

The maze is as below:

S # 57 71 5 37

88 # # 72 # #

# 68 79 # # #

88 69 47 # # #

55 35 89 99 26 D

Sorry, no solution can be found for this maze!

Test the application by performing the following steps:

1. Enter invalid inputs for number of rows / columns and ensure program asks user to re-enter inputs if they are invalid.
2. Ensure application generates correct output for mazes based on example output above.
   1. Ensure application finds path if one exists.
   2. Ensure program outputs “Sorry, no solution can be found for this maze!” if one does not.

# PHASE 2: DESIGN

**2.1 Modules and Their Structure**

The program will have two classes: the class Maze acts as a template for Maze type objects and the class MazeSolver instantiates these objects. These two classes are not related by the class hierarchy (none is derived from the other).

The two classes are specified below by the **UML diagram**.

|  |  |  |
| --- | --- | --- |
| MazeInput |  | LA2Main |
|  |  |  |
| + initializeMazeSolver(solver: MazeSolver): void  + getNumRows(): int  + getNumCols(numRows: int): int |  | + main(args: String[]): void |

|  |
| --- |
| MazeSolver |
| String[][] maze  Int[][] pathScoring  boolean[][] visited  boolean[][] solution  int numRows  int numCols |
| + drawMaze(): void  + printMaze(): void  + isSolvable(): Boolean  + solveMazeRecursively(row: int, col: int): Boolean  + printResult(): void  + setNumRows(numRows: int): void  + getNumRows(): int  + setNumCols(numCols: int): void  + getNumCols(): int |

**2.2 Pseudocode for the Modules**

**2.2.1 Pseudocode for** LA2Main

1a) LA2Main Pseudocode Refinement #1:

// instantiate MazeSolver object

// initialize MazeSolver object using MazeInput.initializeMazeSolver

// ask user for numRows, set respective field in MazeSolver object

// ask user for numCols, set respective field in MazeSolver object

// create maze, set respective fields in MazeSolver object

// print maze

// determine if maze is solvable

// if maze is solvable

// solve maze

// print result

// if maze is not solvable

// print appropriate message

**2.2.2 Pseudocode for** MazeInput

2a) MazeInput Pseudocode Refinement #1:

// --METHODS--

// initializeMazeSolver(solver: MazeSolver): void

// getNumRows(): int

// getNumCols(numRows: int): int

2b) MazeInput Pseudocode Refinement #2:

// --METHODS--

// initializeMazeSolver(solver: MazeSolver): void

// getNumRows(): int

// getNumCols(numRows: int): int

**2.2.3 Pseudocode for** MazeSolver

3a) MazeSolver Pseudocode Refinement #1:

// --FIELDS--

// String[][] maze

// Int[][] pathScoring

// boolean[][] visited

// boolean[][] solution

// int numRows

// int numCols

// --METHODS--

// drawMaze(): void

// printMaze(): void

// isSolvable(): Boolean

// solveMazeRecursively(row: int, col: int): Boolean

// printResult(): void

// setNumRows(numRows: int): void

// getNumRows(): int

// setNumCols(numCols: int): void

// getNumCols(): int

3b) MazeSolver Pseudocode Refinement #2:

// --FIELDS--

// String[][] maze <- array with coins and obstacles

// Int[][] pathScoring <- parallel array to maze with path scoring for A\*

// boolean[][] visited <- parallel array to maze stores visited cells

// boolean[][] solution <- parallel array to maze stores solution

// int numRows

// int numCols

// --METHODS--

//

3b) MazeSolver Pseudocode Refinement #3:

// --FIELDS--

// String[][] maze <- array with coins and obstacles

// Int[][] pathScoring <- parallel array to maze with path scoring for A\*

// boolean[][] visited <- parallel array to maze stores visited cells

// boolean[][] solution <- parallel array to maze stores solution

// int numRows

// int numCols

// --METHODS--

//

// Got to here

**PHASE 3: RISK ANALYSIS**

No risks (to timetable, cost, human health, etc.) are identified by me.

# PHASE 4: VERIFICATION

The algorithm has a recursive execution path. Correctness of this path has been verified by me by analyzing its steps, and their completeness w.r.t. the Specification.

# PHASE 5: CODING

## 5a) Code Refinement #1

**File GradeBook.Java:**

package sampleslc;

// The GradeBook class.

public class GradeBook

{

// Fields: courseName, grades[]

// Constructor: Initialize fields courseName and grades[] (with

// values from parameters).

// Set courseName and grades[]

// Methods: ------------------------------------------------------

// Method findMinimum: Find the minimum value in grades[].

// Initialize lowest to first element in grades[]

// Loop through each grade in grades[]

// If grade is lower than lowest, assign it to lowest

// Return lowest

// Method findMaximum: Find the maximum value in grades[].

// Initialize highest to first element in grades[]

// Loop through each grade in grades[]

// If grade is higher than highest, assign it to highest

// Return highest

// Method computeAverage: Compute the average of all the values in grades[]. // Initialize total to 0

// Loop through each grade in grades[]

// Add grade to total

// Compute and return average by dividing total by number of grades

// Method displayDistribution: Display the counts of grades in each grade range

// as a bar graph.

// Calculate number of grades for each grade range

// Create counts[] to hold counts of grades in each grade

// range; initialize it to 0’s

// Loop through each grade in grades[]

// If the grade falls into i-th grade range, increment counts[i]

// Display header

// Display the bar graph

// Loop through each count (grade range) in counts[]

// Print range label and bar for counts[i]

// 1) Print range label for counts[i]

// 2) Print bar for counts[i]

// Print new line

} // end class GradeBook

**File GradeBookTest.Java:**

package sampleslc;

// The GradeBookTest class for running (thus testing) the GradeBook class.

public class GradeBookTest

{

// Create a new GradeBook object for first set of data.

// Display course name, lowest, highest, and average grades.

// Display grade distribution as a bar graph.

// Create another new GradeBook object for second set of data.

// Display course name, lowest, highest, and average grades.

// Display grade distribution as a bar graph.

} // end class GradeBookTest

## 5b) Code Refinement #2

**File GradeBook.Java:**

package sampleslc;

// The GradeBook class. public class GradeBook {

// Fields: courseName, grades[]

// Constructor: Initialize fields courseName and grades[] (with

// values from parameters).

public GradeBook(/\* name, gradesArray \*/) {

// Set courseName and grades[]

}// end constructor

// Methods: ------------------------------------------------------

// Method findMinimum: Find the minimum value in grades[].

public int findMinimum() {

// Initialize lowest to first element in grades[]

// Loop through each grade in grades[]

// If grade is lower than lowest, assign it to lowest

// Return lowest

}// end method GetMinimum

// Method findMaximum: Find the maximum value in grades[].

public int findMaximum() {

// Initialize highest grade to first element in grades[]

// Loop through each grade in grades[]

// If grade is higher than highest, assign it to highest

// Return highest

}// end method findMaximum

// Method computeAverage: Compute the average of all the values in grades[].

public double computeAverage() {

// Initialize total to 0

// Loop through each grade in grades[]

/ Add grade to total

// Compute and return average by dividing total by number of grades

}// end method computeAverage

// Method DisplayDistribution: Display the counts of grades in each grade range

// as a bar graph.

public void displayDistribution() {

// Calculate number of grades for each grade range

// Create counts[] to hold counts of grades in each grade

// range; initialize it to 0’s

// Loop through each grade in grades[]

// If the grade falls into i-th grade range, increment counts[i]

// Display header

// Display the bar graph

// Loop through each count (grade range) in counts[]

// Print range label and bar for counts[i]:

// 1) Print range label for counts[i]

// 2) Print bar for counts[i]

// Print new line

}// end method displayDistribution

}// end class GradeBook

**File GradeBookTest.Java:**

package sampleslc;

// The GradeBookTest class for running (thus testing) the GradeBook class. public class GradeBookTest

{

// Main method begins program execution

public static void main(String[] args) {

// Create a new GradeBook object for first set of data.

// Display course name, lowest, highest, and average grades.

// Display grade distribution as a bar graph.

// Create another new GradeBook object for second set of data.

// Display course name, lowest, highest, and average grades.

// Display grade distribution as a bar graph.

}// end Main

} // end class GradeBookTest

## 5c) Code Refinement #3

**File GradeBook.Java:**

package sampleslc;

// The GradeBook class.

public class GradeBook

{

// Fields: courseName, grades[]

private String courseName;

private int[] grades;

// getter for courseName

public String getCourseName() {

return courseName;

}

// Constructor: Initialize fields courseName and grades[] (with

// values from parameters).

public GradeBook(String name, int[] gradesArray) {

// Set courseName and grades[]

courseName = name;

grades = gradesArray;

} // end constructor

// Methods: ------------------------------------------------------

// Method findMinimum: Find the minimum value in grades[].

public int findMinimum() {

// Initialize lowest to first element in grades[]

int lowest = grades[0];

// Loop through each grade in grades[]

// If grade is lower than lowest, assign it to lowest

for (int grade : grades) {

if (grade < lowest) lowest = grade;

}

// Return lowest

return lowest;

} // end method findMinimum

// Method findMaximum: Find the maximum value in grades[].

public int findMaximum() {

// Initialize highest to first element in grades[]

int highest = grades[0];

// Loop through each grade in grades[]

// If grade is higher than highest, assign it to highest

for (int grade : grades) {

if (grade > highest) highest = grade;

}

// Return highest

return highest;

}// end method findMaximum

// Method computeAverage: Compute average of all values in grades[].

public double computeAverage() {

// Initialize total to 0

int total = 0;

// Loop through each grade in grades[]

// Add grade to total

for (int grade : grades) {

total += grade;

}

// Compute and return average by dividing total by number of grades

return (double)total / grades.length;

}// end method computeAverage

// Method DisplayDistribution: Display the counts of grades in each grade range

// as a bar graph.

public void displayDistribution() {

// Calculate number of grades for each grade range

// Create counts[] to hold counts of grades in each grade

// range; initialize it to 0’s

int[] counts = new int[11];

// Loop through each grade in grades[]

for (int grade : grades) {

// If the grade falls into i-th grade range, increment counts[i]

counts[grade / 10]++;

}

// Display header

System.out.println(“Grade Distribution”);

// Display the bar graph

// Loop through each count (grade range) in counts[]

for (int i = 0; i < counts.length; i++) {

// Print range label and bar for counts[i]:

// 1) Print range label for counts[i]

if (i == 10)

System.out.print(“ 100: “);

else

System.out.printf(“%02d-%02d: “, i \* 10, i \* 10 + 9);

}

// 2) Print bar for counts[i]

for (int numStars = 0; numStars < counts[i]; numStars++)

System.out.print(“\*”);

// Print new line

System.out.println();

}

}// end method displayDistribution

}// end class GradeBook

**File GradeBookTest.Java:**

package sampleslc;

// The GradeBookTest class for running (thus testing) the GradeBook class. public class GradeBookTest

{

// Main method begins program execution static public static void main(String[] args) {

// Create a new GradeBook object for first set of data.

int[] gradesArray1 =

{ 76, 88, 89, 82, 96, 100, 84, 77, 75, 32, 85, 62, 100, 92, 85 };

GradeBook gb1 = new GradeBook(“CS1110 Computer Science I\n”, gradesArray1);

// Display course name, lowest, highest, and average grades.

System.out.println(gb1.getCourseName());

System.out.printf(“Minimum: %s%n”, gb1.findMinimum());

System.out.printf(“Maximum: %s%n”, gb1.findMaximum());

System.out.printf(“Average: %s%n%n”, gb1.computeAverage());

// Display grade distribution as a bar graph.

gb1.displayDistribution();

System.out.println(“\n=================================\n”); //

Create another new GradeBook object for second set of data. int[] gradesArray2 =

{ 68, 88, 98, 29, 61, 100, 87, 77, 53, 28, 56, 62, 95, 85, 75, 76 };

GradeBook gb2 = new GradeBook(“CS1120 Computer Science II\n”, gradesArray2);

// Display course name, lowest, highest, and average grades.

System.out.println(gb2.getCourseName());

System.out.printf(“Minimum: %s%n”, gb2.findMinimum());

System.out.printf(“Maximum: %s%n”, gb2.findMaximum());

System.out.printf(“Average: %s%n%n”, gb2.computeAverage());

// Display grade distribution as a bar graph.

gb2.displayDistribution();

}// end Main

} // end class GradeBookTest

# PHASE 6: TESTING

This program has only a single execution path. Therefore a single run tests it completely. The single test produces the following output of the program:

CS1110 Computer Science I

Minimum: 32

Maximum: 100

Average: 81.5333333333333

Grade Distribution:

00-09:

10-19:

20-29:

30-39: \*

40-49:

50-59:

60-69: \*

70-79: \*\*\*

80-89: \*\*\*\*\*\*

90-99: \*\*

100: \*\*

=================================

CS1120 Computer Science II

Minimum: 28

Maximum: 100

Average: 71.125

Grade Distribution:

00-09:

10-19:

20-29: \*\*

30-39:

40-49:

50-59: \*\*

60-69: \*\*\*

70-79: \*\*\*

80-89: \*\*\*

90-99: \*\*

100: \*

I verified that this output satisfies program requirements.

**PHASE 7: REFINING THE PROGRAM**

No refinements are needed. In this program, I have already included all required features.

# PHASE 8: PRODUCTION

I prepared a copy of the entire program for Lab TA’s evaluation, as specified by the TA. Then, I sent electronically the copy to the Lab TA.

# PHASE 9: MAINTENANCE

To fully benefit from the program evaluation feedback received from the Lab TA, I will perform program maintenance. This means that I should use all TAs feedback to improve my program.

© 2012-2019 by Leszek T. Lilien and Jeremy Doornbos