CMPE 150 Introduction to Computing **CMPE150.06**

Fall 2018 – Assignment 2 **08.12.2018**

Project Report

▪ Problem Description

The project requires us to write a program that reads a table to an array, then process each cell according to their labels. We had two options. Option A allowed us to use *for* and *while* loops, while Option B required us to write the program without using any. I chose Option B.

The cells can have 5 different labels: F, R, C, N, D.

·F label means the value is fixed, it will not be changed.

·R label means the cell should be replaced with the maximum value in the row.

·C label means the cell should be replaced with the median of the column of the cell.

·N label means the cell should replace all the neighbors and the neighbors of the neighbors and so on, with the value of itself. Neighbors mean the cells above, below, to the left, and to the right of the cell.

·D label means the cell should be replaced with the mean of the cells that are diagonal to the cells in two directions.

My program is able to handle all of the labels.

Implementation

```
import java.util.*;
import java.io.*;

public class ENA2018400312 {

	// In the main method, a file is taken and read in a Scanner. A new 2D array of
	// Strings is created by the method setTable, which reads the first line of the
	// file. After that, the entire table is read from table using readTable method.
	// The original table is printed. After that, two new arrays are created, one is
	// of Strings, one is of integers. The original table is split into these two
	// arrays. One holds only the labels and the other holds only the integer
	// values, so it becomes more convenient to process the table. After that, the
	// method process is called and the entire table of cells is processed one by
	// one. In the end, the final table of integers is printed.
	public static void main(String[] args) throws FileNotFoundException {
		File file = new File("input.txt");
		Scanner s = new Scanner(file);
		String[][] table = setTable(s);
		table = readTable(s, table, 0);

		printTable(table, 0);
		System.out.println();

		int[][] tableI = new int[table.length][table[0].length];
		tableI = tableToInt(table, tableI, 0);

		String[][] tableS = new String[table.length][table[0].length];
		tableS = tableToS(table, tableS, 0);

		process(tableS, tableI, 0);
		printTable(tableI, 0);

	}

	// This method takes 2 2D arrays, one of Strings, one of integers, which is
	// empty.
	// It examines the table and takes the integer value of each cell, and assigns
	// it to the same cell of the empty array. That way, the empty array is filled
	// only the integers of the original table. Starting from 0, i is increased in
	// every iteration until it reaches the end of the table.
	public static int[][] tableToInt(String[][] table, int[][] output, int i) {
		if (i < table.length * table[0].length) {
			output[i / table[0].length][i % table[0].length] = Integer
					.parseInt(table[i / table[0].length][i % table[0].length].substring(1));
			tableToInt(table, output, i + 1);
		}
		return output;
	}

	// This method takes 2 2D arrays of Strings of same size, one filled, one empty.
	// It examines the table and takes the first character of each cell, and assigns
	// it to the same cell of the empty array. That way, the empty array is filled
	// only the labels of the original table. Starting from 0, i is increased in
	// every iteration until it reaches the end of the table.
	public static String[][] tableToS(String[][] table, String[][] output, int i) {
		if (i < table.length * table[0].length) {
			output[i / table[0].length][i % table[0].length] = (table[i / table[0].length][i %
table[0].length].substring(0, 1));
			tableToS(table, output, i + 1);
		}
		return output;
	}

	// This method reads the file in a Scanner to a 2D array of Strings, and returns
	// the array. i is the coordinate value. Starting from 0, i is increased in each
	// iteration until it reaches the end of the table. That way, the table is
	// filled.
	public static String[][] readTable(Scanner s, String[][] table, int i) {
		if (i < table.length * table[0].length) {
			table[i / table[0].length][i % table[0].length] = s.next();
			readTable(s, table, i + 1);
		}
		return table;
	}

	// This method creates a new table and returns it, by taking the first line of a
	// scanner and reads the value in it. The numbers until the letter "x" is the a
```

```
	// coordinate of the table, and the numbers after it are the b coordinate. A new
	// 2D array of Strings is created, that has a size of a and b.
	public static String[][] setTable(Scanner s) {
		String dimension = s.nextLine();
		int a = Integer.parseInt(dimension.substring(0, dimension.indexOf("x")));
		int b = Integer.parseInt(dimension.substring(dimension.indexOf("x") + 1));

		String[][] table = new String[a][b];

		return table;
	}

	// This method takes an array of Strings and prints it. It also takes an integer
	// i, that holds the x and y coordinate of the table. In every iteration, i is
	// increased until it reaches the end of the table. If the current cell is the
	// last cell of a row but not the end of the file, the method goes to the next
	// line. The value in each cell is printed, by finding the x and y coordinates
	// from i.
	public static void printTable(String[][] table, int i) {
		if (i % table[0].length == 0 && i != table.length * table[0].length && i != 0) {
			System.out.println();
		}
		if (i < table.length * table[0].length) {
			System.out.print(table[i / table[0].length][i % table[0].length] + " ");
			printTable(table, i + 1);
		}

	}

	// This method takes an array of integers, and does the same thing the method
	// above does.
	public static void printTable(int[][] table, int i) {
		if (i % table[0].length == 0 && i != table.length * table[0].length) {
			System.out.println();
		}
		if (i < table.length * table[0].length) {
			System.out.print(table[i / table[0].length][i % table[0].length] + " ");
			printTable(table, i + 1);
		}

	}

	// This method takes two 2D arrays, one String, one integer, and takes an
	// integer, it holds the coordinate value. In the method, each cell is checked.
	// In every iteration, i is increased by 1, until the method reaches the end of
	// the table. In every cell, the method checks which label it has. If the label
	// is F, the method does nothing, so I didn't write that part. If the label is
	// R, the method calls processR, and it returns the largest number in the
	// specific row. If the label is D, processD is called. If the label is C, a new
	// array called column is made, and it is filled by getColumn. After that,
	// processC is called, and it returns the median of the column. After every cell
	// is reevaluated, the method returns the new array of integers, tableI.
	public static int[][] process(String[][] tableS, int[][] tableI, int i) {

		if (i < tableS.length * tableS[0].length) {
			String letter = tableS[i / tableS[0].length][i % tableS[0].length];
			switch (letter) {
			case "N":
				processN(tableS, tableI, i);
				break;
			case "R":
				tableI[i / tableI[0].length][i % tableI[0].length] = processR(tableI[i /
tableS[0].length]);
				break;
			case "D":
				processD(tableI, i);
				break;
			case "C":
				int[] column = new int[tableI.length];
				column = getColumn(tableI, column, 0, i % tableS[0].length);
				tableI[i / tableI[0].length][i % tableI[0].length] = processC(tableI, column);

			}
			process(tableS, tableI, i + 1);
		}
		return tableI;
	}

	// This method processes R label. It returns the largest number in the row. If
	// the array has a length of 1, it returns the only number in it. If there is
	// more than 1 item, the method is called again, taking the largest of the
```

```java
// values except the first one.
public static int processR(int[] row) {
	if (row.length == 1)
		return row[0];

	else
		return Math.max(row[0], processR(Arrays.copyOfRange(row, 1, row.length)));

}

// This method processes N label. It takes an i number, which holds the x and y
// coordinates of the specific cell. The method checks if the cells next to it,
// above and below are labeled N. If they are, the value they hold are changed
// the the number in the original cell. After that, the same process is done in
// the cells surrounding the original cell. In the end, all the N labeled cells
// that are neighbors have the same value. The changed table is returned.
public static int[][] processN(String[][] tableS, int[][] tableI, int i) {

	int r = tableS.length;
	int c = tableS[0].length;
	if (i < r * c) {

		int y = i / c;
		int x = i % c;
		int co = tableI[y][x];

		if (y > 0) {
			if (tableS[y - 1][x].equals("N")) {
				tableI[y - 1][x] = co;
				tableS[y][x] = "n";
				processN(tableS, tableI, i - c);
			}
		}
		if (y + 1 != r) {
			if (tableS[y + 1][x].equals("N")) {
				tableI[y + 1][x] = co;
				tableS[y][x] = "n";
				processN(tableS, tableI, i + c);
			}
		}
		if (x + 1 != c) {
			if (tableS[y][x + 1].equals("N")) {
				tableI[y][x + 1] = co;
				tableS[y][x] = "n";
				tableI = processN(tableS, tableI, i + 1);
			}
		}
		if (x > 0) {
			if (tableS[y][x - 1].equals("N")) {
				tableI[y][x - 1] = co;
				tableS[y][x] = "n";
				tableI = processN(tableS, tableI, i - 1);
			}
		}

	}
	return tableI;
}

// This method processes C label by taking an array called column is sorted from
// lower to higher value. processC returns the value in the middle.
public static int processC(int[][] table, int[] column) {
	Arrays.sort(column);
	return column[(column.length + 1) / 2 - 1];
}

// This method is called by processC, it returns the specific column wanted.
// Starting from 0, y coordinate is increased in every iteration and x
// coordinate is held constant. Each value is stored in an array called column.
// After y coordinate reaches the end of the table, column is returned.
public static int[] getColumn(int[][] table, int[] column, int y, int x) {
	if (y < table.length) {
		column[y] = table[y][x];
		getColumn(table, column, y + 1, x);
	}
	return column;

}

// This method processes D label, by calling 4 other methods, and they find the
// elements in 4 diagonal directions. The sum of the elements in diagonal
```

```java
	// directions are stored in the array sum. The number of them are stored in the
	// array count. Since there are 4 directions in a 2D table, sum and count have a
	// length of 4. After that, the integer in the corresponding cell of the table
	// is changed to the average of these values.
	public static int[][] processD(int[][] tableI, int i) {
		int x = i % tableI[0].length;
		int y = i / tableI[0].length;
		int[] sum = new int[4];
		int[] count = new int[4];
		leftDownD(tableI, sum, y, x, count);
		rightDownD(tableI, sum, y, x, count);
		leftUpD(tableI, sum, y, x, count);
		rightUpD(tableI, sum, y, x, count);

		tableI[y][x] = (sum[0] + sum[1] + sum[2] + sum[3] - 3 * tableI[y][x])
				/ (count[0] + count[1] + count[2] + count[3] - 3);
		return tableI;
	}

	// This method is one of the 4 methods that are called by processD, it checks
	// the left down direction of the specific cell, and returns the sum of them in
	// the array sum, and the number of them in the array count.
	public static int leftDownD(int[][] tableI, int[] sum, int y, int x, int[] count) {
		if (x < 0) {
			return 0;
		}
		if (y < tableI.length) {
			sum[0] += tableI[y][x];
			count[0]++;

		}
		return leftDownD(tableI, sum, y + 1, x - 1, count);
	}

	// This method is one of the 4 methods that are called by processD, it checks
	// the left up direction of the specific cell, and returns the sum of them in
	// the array sum, and the number of them in the array count.
	public static int leftUpD(int[][] tableI, int[] sum, int y, int x, int[] count) {
		if (x < 0) {
			return 0;
		}
		if (y > -1) {
			sum[1] += tableI[y][x];
			count[1]++;

		}
		return leftUpD(tableI, sum, y - 1, x - 1, count);
	}

	// This method is one of the 4 methods that are called by processD, it checks
	// the right down direction of the specific cell, and returns the sum of them in
	// the array sum, and the number of them in the array count.
	public static int rightDownD(int[][] tableI, int[] sum, int y, int x, int[] count) {
		if (x > tableI[0].length - 1) {
			return 0;
		}
		if (y < tableI.length) {
			sum[2] += tableI[y][x];
			count[2]++;

		}
		return rightDownD(tableI, sum, y + 1, x + 1, count);
	}

	// This method is one of the 4 methods that are called by processD, it checks
	// the right up direction of the specific cell, and returns the sum of them in
	// the array sum, and the number of them in the array count.
	public static int rightUpD(int[][] tableI, int[] sum, int y, int x, int[] count) {
		if (x > tableI[0].length - 1) {
			return 0;
		}
		if (y > -1) {
			sum[3] += tableI[y][x];
			count[3]++;

		}
		return rightUpD(tableI, sum, y - 1, x + 1, count);
	}
}
```

Output of the Program

```
F0 N5 N0 N0 N4 F3 R9 R4 N9 R3 C1 F8 C8
F3 R9 R4 N9 R3 C1 F8 C8 C3 C7 D6 D1 F8
C1 F8 C8 C3 C7 D6 D1 F8 F8 R3 R3 D9 N8
D6 D1 F8 F8 R3 R3 D9 N8 D5 C9 D6 F6 D2
R3 D9 N8 D5 C9 D6 F6 D2 F1 D7 R4 D6 C3

0 5 5 5 5 3 9 9 9 9 4 8 8
3 9 9 5 9 3 8 8 5 7 5 4 8
3 8 8 5 7 6 5 8 8 9 9 6 8
7 4 8 8 9 9 6 8 6 9 7 6 5
9 6 8 7 9 6 6 6 1 6 9 6 8
```

```
F9 N9 F7 N7 N7 C0 C4
D0 R7 F6 R8 C4 R0 R2
N4 R4 C9 C5 D9 R4 D5
N6 N1 D3 R6 F8 R3 C7
R9 N4 N0 D3 C3 N6 D3
R7 N6 N6 R0 C5 N9 C5
C3 N4 R7 D3 R2 D2 C2
N6 F4 N5 D1 R8 D9 C2
R2 C6 F2 R7 R9 R4 F6
N7 F5 F5 N9 R4 N2 D2
C1 R8 C8 R5 D7 F1 R0
F9 R1 F0 N4 D0 N9 N0
C9 C4 C8 N3 N3 N9 R2
D3 F2 N4 N4 N2 R2 D7

9 9 7 7 7 3 2
3 8 6 8 4 8 8
4 9 5 4 5 9 4
4 4 4 8 8 8 2
9 4 4 5 4 6 4
7 4 4 7 4 6 2
4 4 7 4 7 5 2
6 4 5 4 9 6 2
9 4 2 9 9 9 6
7 5 5 9 9 2 4
6 8 5 8 4 1 8
```