ITI 1120 Fall 2012 - Assignment 4

Available: Sunday, November 11, 2012
Due: Wednesday, November 21, 2012, 11:59 pm

Instructions

This assignment is to be done INDIVIDUALLY. Follow the instructions in the lab manual for submitting assignments through the Virtual campus. The following are specific instructions for this assignment:

1) For question 1, provide a Word file A4Q1.doc containing the algorithm developed for the question as well as the programming model that shows the results of tracing the algorithm. A Visio file is provided for completing the programming model or you may use the programming model provided in the Appendix of this document.
2) For question 2, submit the Java file A4Q2.java. The Java code should also be inserted in the files A4Q2.doc for commenting
3) For question 3, submit the Java file A4Q3.java. The Java code should also be inserted in the files A4Q3.doc for commenting.
4) Zip all the .doc, .java, and .class files in A4_xxxxxx.zip, where xxxxxx is your student number, and submit it through the Virtual Campus. Please DO NOT return the JUnit test files that have been provided.

Your algorithms should be developed using the format used in class. In the Java code, use the coding structures presented in class. Do not use other structures, such as the switch and break statements.

Marking Scheme (total 100 marks)

- **Regulations and Standards: 5 marks**
- Question 1: 25 marks
- Question 2: 20 marks
- Question 3: 50 marks

Question 1 (25 marks)

a) Develop a recursive algorithm, *reverseChArray* that reverses the characters in a character array containing *n* characters. For example, the contents of the character array {‘a’, ‘ ’, ‘s’, ‘t’, ‘r’, ‘i’, ‘n’, ‘g’} becomes {‘g’, ‘n’, ‘i’, ‘r’, ‘t’, ‘s’, ‘ ’ , ‘a’ } (hint: note that characters in positions 0 and *n*-1 are exchanged, as are characters in positions 1 and *n*-2, 2 and *n*-3, …..).

The algorithm is given a reference to a character array, *cArr*, as well as the index of the first array element under consideration, *ixStart*, and the index of the last array element under consideration, *ixEnd*.

b) Complete the programming model in Appendix A to illustrate how memory is changed by tracing the recursive algorithm when the contents of the array is {‘a’, ‘b’, ‘c’, ‘d’, ‘e’ } (*n*=5). Show how the contents of the variables (and any changes) in the working memory of each recursive call.

Question 2 (20 marks)

Translate the algorithm developed in question 1 to a **recursive Java method** and put it into the class A4Q2, stored in the file A4Q2.java. Class A4Q2 should not contain a main algorithm. Test your method using the JUnit class A4Q2Test.java provided. Do not submit the A4Q2Test.java file as part of your assignment.

Question 3 (50 marks)

Consider an *n* x *m* matrix, the grid matrix, containing Boolean values in which a path is defined by “true” values. For example:

false	*true*	*false*	*false*	*false*
false	*true*	*false*	*false*	*false*
false	*true*	*true*	*false*	*false*
false	*false*	*true*	*false*	*false*

Note that the path can move in horizontal or vertical directions (not diagonally) and always will start on a border and terminate on a border (i.e. first/last row or first/last column). Also a path can never be adjacent to itself, that is, any point in the path cannot be adjacent to more than two points of the path, the previous point and the next point in the path. That is, with the exception of the starting and ending points in the path, only two of the elements surrounding a point in the path can have true values in the vertical and horizontal directions.

Your task is to complete the file A4Q3.java that has been provided. You do not have to supply algorithms for this question. But do take the time to think about the logic of the methods before you start coding. Make notes and even draft out algorithms – it will save you time.

The principle Java method in the class A4Q3 is named *findpath* and has the header:

```
public static int [][] findPath(boolean [] [] grid, int row, int col)
```

The method will return a reference to an integer matrix, the path matrix that defines the path defined in the grid matrix of Boolean values. The method parameters are defined as follows:

- *grid*: a reference to the *n* x *m* Boolean matrix containing a path of "true" values (hint: dimensions of the matrix is given by *grid.length* and *grid[0].length*).
- *row*, *col*: row and column index values of the first point in the path of the matrix; these values define a position on a border (i.e. an element in either row 0 or *n*-1, or in column 0 or *m*-1).

The *findPath* method shall produce a list of points (row, col) in a path matrix with dimension *l* x 2 where *l* corresponds to the number of points in the path, i.e. the length of the path. For the above grid matrix, the method produces a 5 x 2 path matrix that contains:

```
0  1
1  1
2  1
2  2
3  2
```

The method *findPath* first creates a path matrix with a single row (and 2 columns) that will contain the first point of the path. It then will call the recursive method *doPath* to add the other points to the matrix. The task of the method *doPath* is subdivided into smaller tasks implemented within other methods (you will also find the header of these methods in the provided file A4Q3.java).

- *doPath*: this recursive method is responsible for finding the path through the matrix and building the result matrix using other methods. It uses the method *findNextPoint* to obtain the next point in a path. Also, for checking if a point is on the matrix border, it calls method *isOnBorder*. The end of a path is found when either the following conditions occur:
  - The point returned by *findNextPoint* is on the border of the matrix (first/last row or first/last column).
  - A null value is returned by *findNextPoint* (an error has been encountered).
- *findNextPoint*: this method finds the next point in the path given the current path matrix (hint: you need to consider the last two points in the current path so that the next point selected is not the before last point). The next point is returned in an integer array (you must create the array in *findNextPoint)* with two elements. The first element contains the row number of the next point and the second element contains the column number of the next point. If the next point cannot be found or more than one 'next' point is found, a *null* address is returned (note that you must test to see if more than one 'next' point exists).
- *addPoint*: this method shall add a row and column number to the end of the path matrix. It shall increment its length of the row dimension by one and add the given row and column numbers (*row* and *col* parameters). A new matrix must be created and its reference will be returned by the method. This way the path matrix to be produced by *findPath* will have the exact dimension required to contain the path points. The parameter *path* will contain *null* when the first point is added to the path matrix.

  Consider the following call:

  ```
  int [][] path = addPoint(path, 3, 2)
  ```

  The intention of the above call is to increase the dimensions of the matrix referenced by the *path* variable by one row to then add 3 and 2 into the columns of the new row. The only way to do this is to create a new matrix, copy the contents of the shorter matrix and add the new point. Thus, *addPoint*, must return the reference of the new matrix created. This approach will be necessary for other methods. That is, the reference stored in *path* is given to *addPoint* that returns a new reference (to the new matrix) which replace the previous reference in *path*.

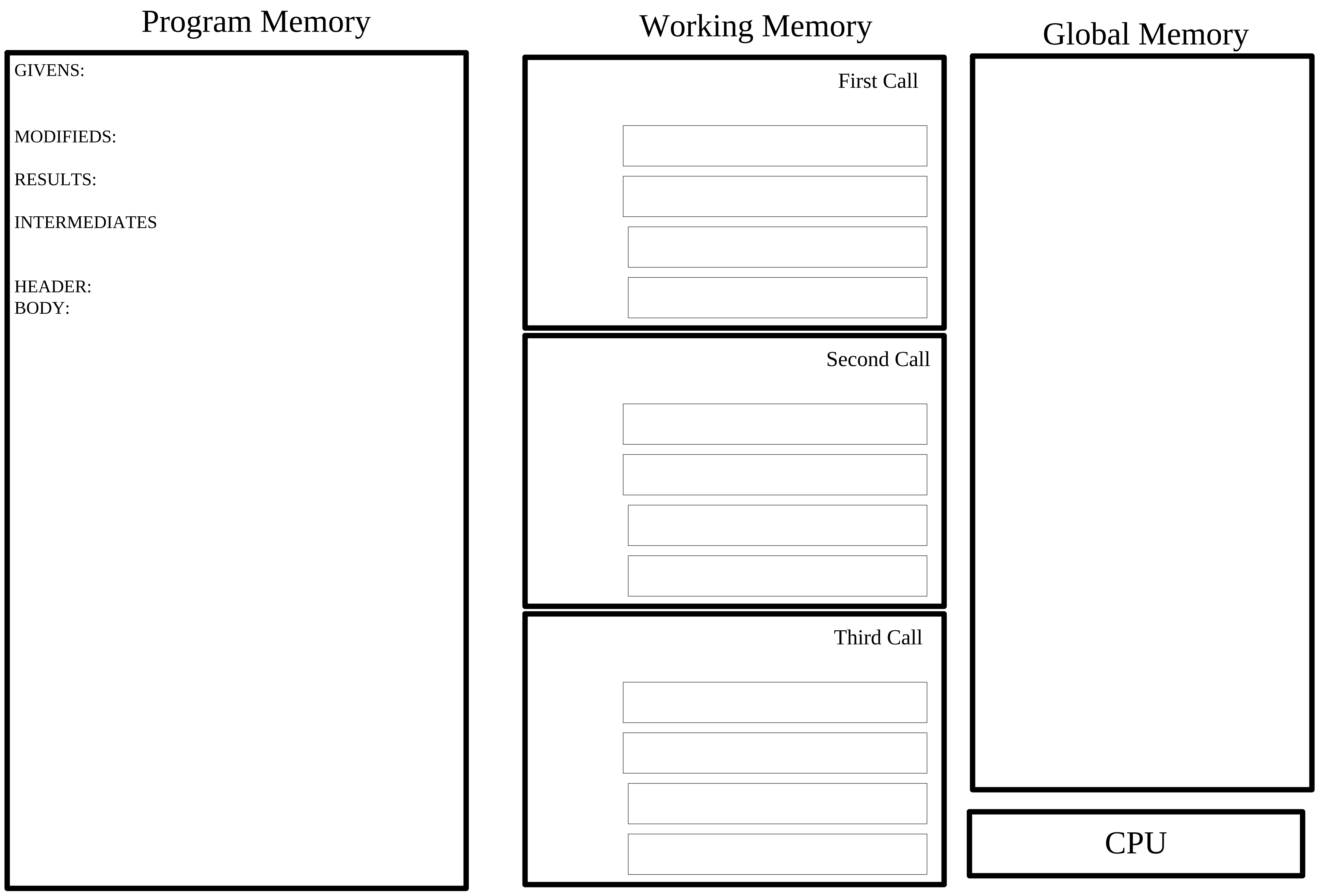
- *isOnBorder*: this method returns true if a given point is found on the matrix border (first row or last row or first column or last column), and false otherwise. Note that the dimension of the matrix can be obtained with *matrix.length* (number of rows) and *matrix[0].length* (number of columns).

Hints:

- The dimension of matrix referenced by the reference variable *mat*, is defined by the *length* variables defined for rows and columns. That is, *mat.length* gives the number of rows in the matrix, and *mat[0].length* gives the number of columns of the matrix.
- File A4Q3Test.java has been provided to test all methods. Each of the methods provided in the A4Q3.java file contain a single return statement. These return statements allow the compilation of the class as provided, and all JUnit but one of the tests defined in A4Q3Ttest.java will fail. You can develop each method individually (start with the simple methods) and run the JUnit tests after you complete one method and before you start the next. Note that the tests in A4Q3Test.java are ordered to test the simpler methods first.
- The design of the *findNextPoint* shall be the most challenging. Think about using a separate method to check if a given point can be considered as a next point. The following list gives the parameters (givens), value returned (results), and the logic of such a method.
    - Parameters: A reference to the grid matrix, the indexes (row and column) of the before last point in the current path, and the indexes (row and column) of the point to consider.
    - Value returned: If the point to consider is valid, then an integer array is created, the indexes of the point stored in the array, and the array address is returned; otherwise the *null* address is returned.
    - Method logic: Ensure that the point does not correspond to the before last point and that the indexes of the point to be considered are valid (compare to the dimensions of the grid matrix). Then if the element of the point in the grid matrix is true, then the point can be considered a next point. Note that this method shall not be tested by the JUnit tests.

This method can then be called by the *findNextPoint* method to check all of the four possible points around the last point of a given path. Only one next point may exist – it is thus necessary to check all four points with four calls. If at least two of the calls indicate a next point, then *findNextPoint* must return a *null* value. If only one of the calls returns a non-null address, then the address can be returned by *findNextPoint.* Finally, when the current point is on the border (first point in the path) set the indexes of the before last point to -1.

Appendix A – Programming Model


Program Memory
GIVENS:
MODIFIEDS:
RESULTS:
INTERMEDIATES
HEADER:
BODY:
Working Memory
First Call
Second Call
Third Call
Global Memory
CPU