Running the code

The algorithm that I have implemented is Kosaraju's algorithm in Java. The resources that I took help of for this have been provided as references. The algorithm itself is in the file Graph.java which contains the "main" function. Since I have used the datastructures of stack and list, I have implemented my own version of them. I have not made use of the "Collections Framework" in java.

As for running the code, we go to the "main" function in Graph.java and provide the path of the input file as the parameter to the function "readFile". The input file would be used for creating the integer array named "dataArray". Each integer provided in the file will be stored, in order, in the "dataArray". Once the path for the input file is provided, we must provide the path for the output file. To do this, we must go to "FW.java", located in the default package (there is only one package in my code). In the following line of code, we provide the path:

```
File file = new File("Enter path for output file");
```

Now that we have the paths for the input and output files typed in, we can run the code. The result will be stored in the output file in the directory we specified which would be a text file as per the requirements of the problem.

Classes for reading the input file and printing to the output file

The class "readFile.java" handles the reading of the input text file. The function "readFile" uses a scanner object to read the input file line-by-line and store the integer values in an array and this array is then returned.

The class "FW.java" handles printing to the output file. It has one function "CreateAndWriteToFile" that takes the data to be printed (only string allowed) as parameter. The path for the output file is given when a new "File" object is created. This function will be called multiple times during the run of the DFS.

Explanation of the algorithms used

The steps of Kosaraju's algorithm are given as follows:

- Create an empty stack S.
- Run the DFS algorithm on the graph and store the vertices in the stack based on their "finish time".
- After the DFS terminates, reverse the edges of the given graph to obtain the transpose graph.

- While the stack is not empty, pop a vertex from S one by one. Take every popped vertex as the source and do a DFS call on the source. The DFS starting from the source will give a strongly connected component.

The code follows the above steps. The different aspects of the code have been divided into subsections as follows:

Graph Creation

The code makes use of adjacency list to represent the graph. Technically, a graph is represented by an array of linked lists. In the class "Graph", we can look at the constructor which takes an input parameter which is the number of vertices in the graph. The size of the array would be the number of vertices plus one since we are working with a 1-based index here. The function iterates through the array and creates a new linked list at every index of the array starting with index 1.

The function for adding an edge adds a vertex 'u' to the linked list (neighbour list) of 'v'. Now we look at the main function where the graph is actually constructed for the given input. Starting from the second line of the input file, we start reading a vertex and its neighbour on the same line and add an edge in the graph.

The DFS algorithm

The main function calls the print SCC function and here's where the DFS begins. Initially, all vertices of the graph are unvisited by DFS. So in the boolean array "visited[i]", the boolean for every vertex is false. As we visit a vertex one by one, we call a helper function "fillOrder" on the vertex which performs a recursive DFS on the children of the vertex 'v' in consideration and also keeps track of the finish time.

The fillOrder function takes a vertex 'v' and a stack 'S' as input. The function looks at each of its neighbour and checks if the neighbour has been visited or not. If the neighbour is unvisited, then fillOrder is called recurisively on the neighbour. When all the neighbouring vertices are visited, 'v' is pushed into 'S'.

Once the fillOrder function has been called on each vertex in the printSCC function, the DFS is complete. Now we obtain the transpose of our graph. This is done by calling the "getTranspose" function. This function checks the adjacency list for each vertex, and if there was a directed edge from a vertex 'v' to 'u' in the original list, the function reverses the edge by adding 'v' to the adjacency list of 'u'. This is done for all edges. As a result, we get the transpose of the original graph.

Now we work on the transpose graph. We set all vertices as unvisited again. We pop a vertex from S and check if it was unvisited or not. If it was unvisited, then a variant of the DFS function used earlier is called on it. This function does recursive DFS as well but it does not compute finish times and it also prints the result to the output file. Calling “dfsUtil” on a vertex ‘v’ prints all the vertices reachable from ‘v’ in the transpose graph. Once ‘v’ is done, the control returns to “printSCC” function and the same thing is done for each vertex popped from S. The while loop runs until there is no element left in the Stack. The final result is an output file with the strongly connected components printed in the desired pattern.

Implementation of datastructures

Linked List

The code for my implementation of the linked list class is located in “MyLinkedList3.java” in the default package. The standard linked list functions for creating a linked list, adding a node to a linked list, deleting the node from a linked list have been implemented. The code is well-commented and every function has comments explaining what the function does. An iterator for the linked list has also been implemented.

Stack

The code for my implementation of stack class is located in “MyStack.java” in the default package. The stack datastructure has been implemented using the concept of linked list. The standard stack functions of push, pop and returning the size of the stack have been implemented. The push operation here always adds an element to the front (top) of the stack. The code is well-commented and every function has comments explaining what the function does. An iterator for the stack has also been implemented.

References

- **Kosaraju's algorithm:**
  http://www.geeksforgeeks.org/strongly-connected-components/
  https://www.youtube.com/watch?v=5wFyZJ8yH9Q

- **Linked List Datastructure:**
  https://www.cs.cmu.edu/~adamchik/15-121/lectures/Linked%20Lists/code/LinkedList.java

- **Stack Datastructure:**
  http://algs4.cs.princeton.edu/13stacks/LinkedStack.java.html

- **Reading an input file**
  https://www.youtube.com/watch?v=k9pf8KKPcwI&index=7&list=LLDEyoU8jkNnvBGIYPU2vuTg

- **Printing to an output file**
  https://www.youtube.com/watch?v=ymvFVkJ_SDI