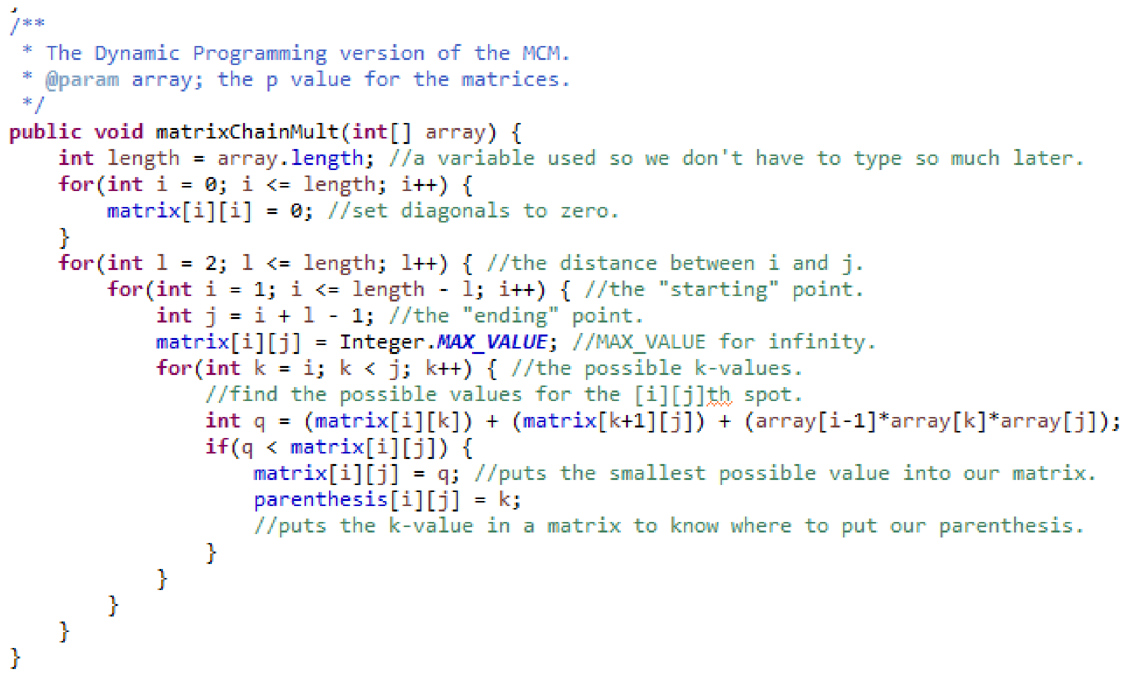
Tiffany Jansen
CS 361
Lab 3

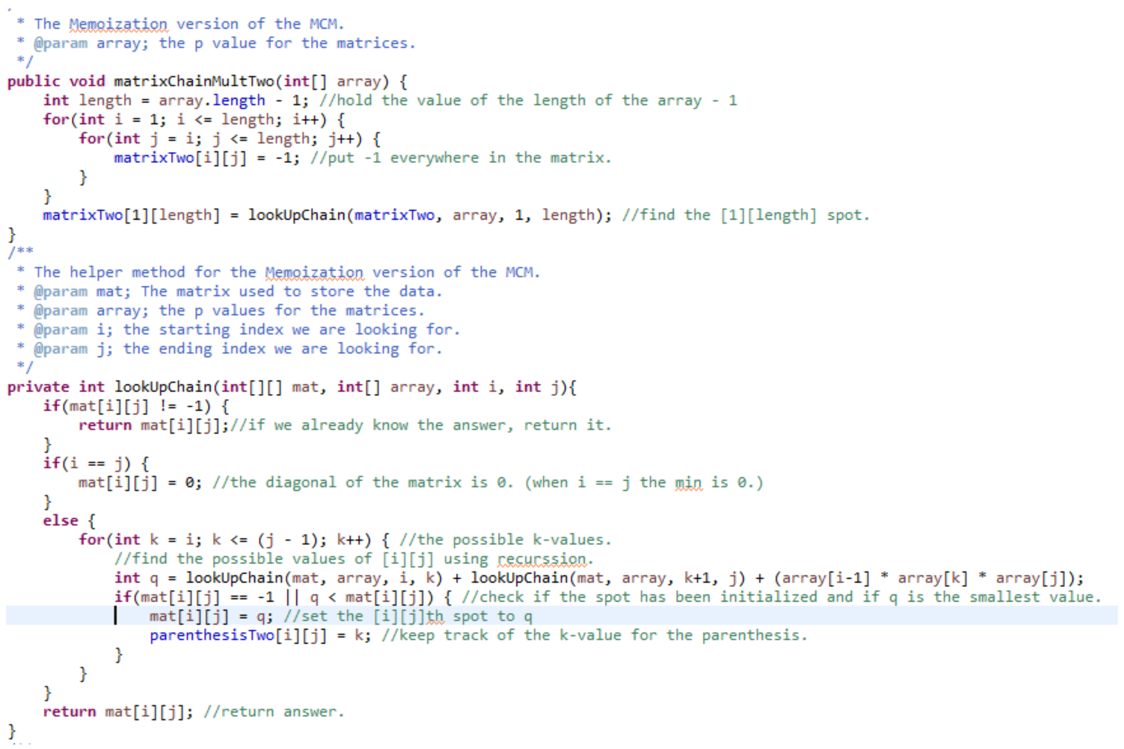
Lab Report

1. *DP version of MCM algorithm*



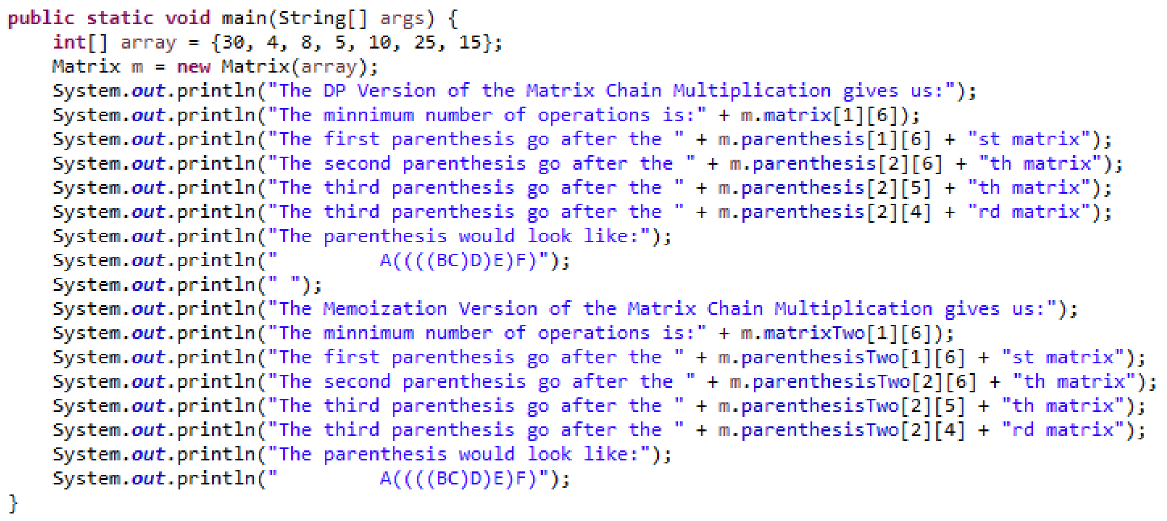
```
/**
 * The Dynamic Programming version of the MCM.
 * @param array; the p value for the matrices.
 */
public void matrixChainMult(int[] array) {
    int length = array.length; //a variable used so we don't have to type so much later.
    for(int i = 0; i <= length; i++) {
        matrix[i][i] = 0; //set diagonals to zero.
    }
    for(int l = 2; l <= length; l++) { //the distance between i and j.
        for(int i = 1; i <= length - l; i++) { //the "starting" point.
            int j = i + l - 1; //the "ending" point.
            matrix[i][j] = Integer.MAX_VALUE; //MAX_VALUE for infinity.
            for(int k = i; k < j; k++) { //the possible k-values.
                //find the possible values for the [i][j]th spot.
                int q = (matrix[i][k]) + (matrix[k+1][j]) + (array[i-1]*array[k]*array[j]);
                if(q < matrix[i][j]) {
                    matrix[i][j] = q; //puts the smallest possible value into our matrix.
                    parenthesis[i][j] = k;
                    //puts the k-value in a matrix to know where to put our parenthesis.
                }
            }
        }
    }
}
```

2. *Memoization version of MCM algorithm*



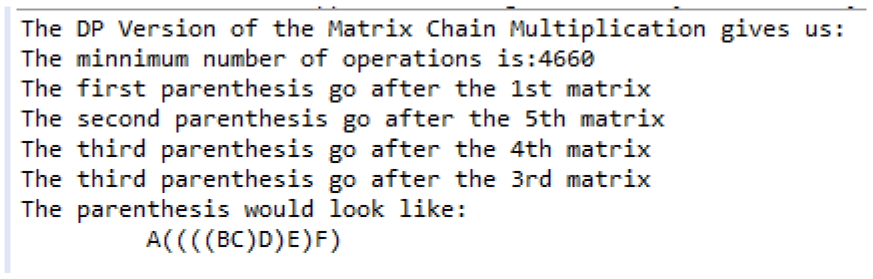
```
 * The Memoization version of the MCM.
 * @param array; the p value for the matrices.
 */
public void matrixChainMultTwo(int[] array) {
    int length = array.length - 1; //hold the value of the length of the array - 1
    for(int i = 1; i <= length; i++) {
        for(int j = i; j <= length; j++) {
            matrixTwo[i][j] = -1; //put -1 everywhere in the matrix.
        }
    }
    matrixTwo[1][length] = lookUpChain(matrixTwo, array, 1, length); //find the [1][length] spot.
}
/**
 * The helper method for the Memoization version of the MCM.
 * @param mat; The matrix used to store the data.
 * @param array; the p values for the matrices.
 * @param i; the starting index we are looking for.
 * @param j; the ending index we are looking for.
 */
private int lookUpChain(int[][] mat, int[] array, int i, int j){
    if(mat[i][j] != -1) {
        return mat[i][j];//if we already know the answer, return it.
    }
    if(i == j) {
        mat[i][j] = 0; //the diagonal of the matrix is 0. (when i == j the min is 0.)
    }
    else {
        for(int k = i; k <= (j - 1); k++) { //the possible k-values.
            //find the possible values of [i][j] using recurssion.
            int q = lookUpChain(mat, array, i, k) + lookUpChain(mat, array, k+1, j) + (array[i-1] * array[k] * array[j]);
            if(mat[i][j] == -1 || q < mat[i][j]) { //check if the spot has been initialized and if q is the smallest value.
                mat[i][j] = q; //set the [i][j]th spot to q
                parenthesisTwo[i][j] = k; //keep track of the k-value for the parenthesis.
            }
        }
    }
    return mat[i][j]; //return answer.
}
```

Main Method Code:



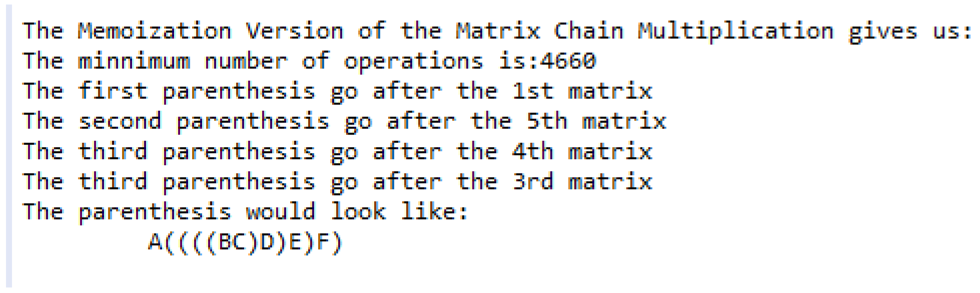
```
public static void main(String[] args) {
    int[] array = {30, 4, 8, 5, 10, 25, 15};
    Matrix m = new Matrix(array);
    System.out.println("The DP Version of the Matrix Chain Multiplication gives us:");
    System.out.println("The minnimum number of operations is:" + m.matrix[1][6]);
    System.out.println("The first parenthesis go after the " + m.parenthesis[1][6] + "st matrix");
    System.out.println("The second parenthesis go after the " + m.parenthesis[2][6] + "th matrix");
    System.out.println("The third parenthesis go after the " + m.parenthesis[2][5] + "th matrix");
    System.out.println("The third parenthesis go after the " + m.parenthesis[2][4] + "rd matrix");
    System.out.println("The parenthesis would look like:");
    System.out.println("         A((((BC)D)E)F)");
    System.out.println(" ");
    System.out.println("The Memoization Version of the Matrix Chain Multiplication gives us:");
    System.out.println("The minnimum number of operations is:" + m.matrixTwo[1][6]);
    System.out.println("The first parenthesis go after the " + m.parenthesisTwo[1][6] + "st matrix");
    System.out.println("The second parenthesis go after the " + m.parenthesisTwo[2][6] + "th matrix");
    System.out.println("The third parenthesis go after the " + m.parenthesisTwo[2][5] + "th matrix");
    System.out.println("The third parenthesis go after the " + m.parenthesisTwo[2][4] + "rd matrix");
    System.out.println("The parenthesis would look like:");
    System.out.println("         A((((BC)D)E)F)");
}
```

3. *The output for the DP version of MCM algorithm*



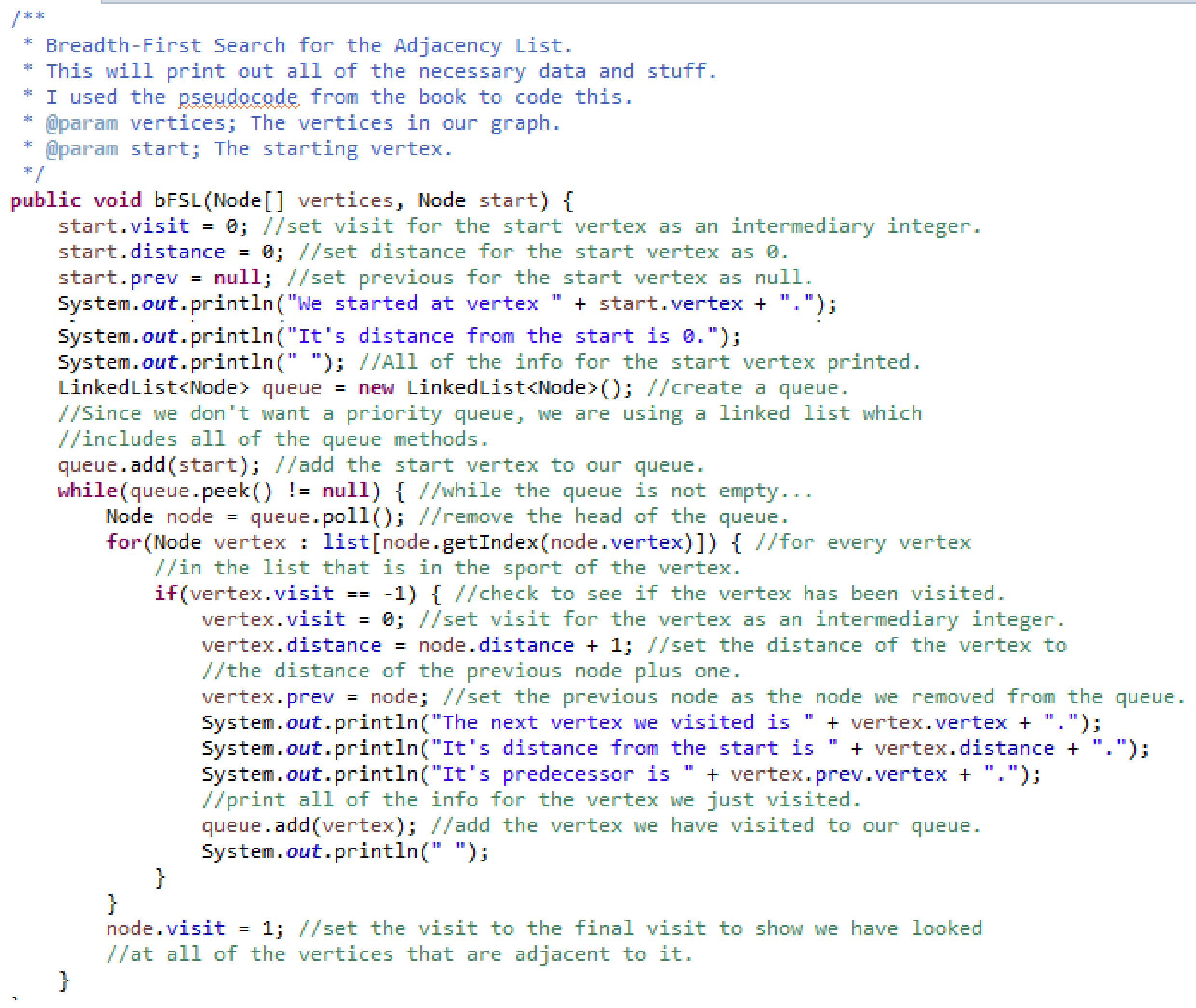
```
The DP Version of the Matrix Chain Multiplication gives us:
The minnimum number of operations is:4660
The first parenthesis go after the 1st matrix
The second parenthesis go after the 5th matrix
The third parenthesis go after the 4th matrix
The third parenthesis go after the 3rd matrix
The parenthesis would look like:
         A((((BC)D)E)F)
```

4. *The output for the Memoization version of MCM algorithm*



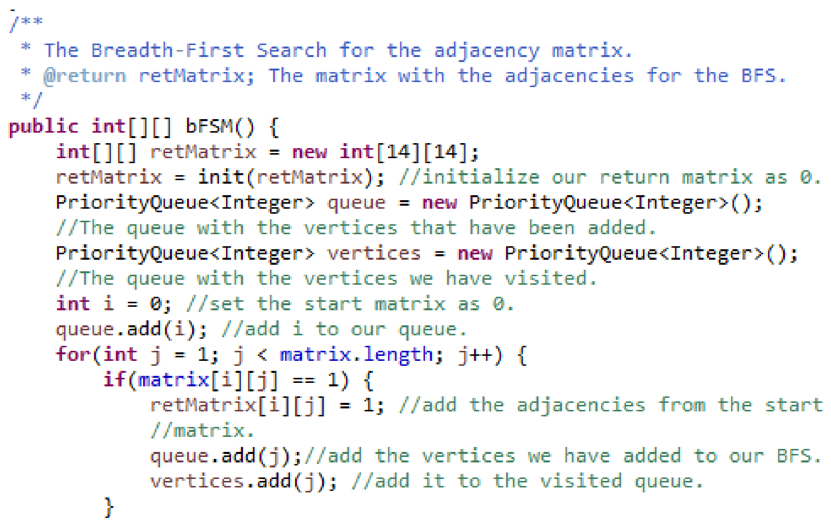
```
The Memoization Version of the Matrix Chain Multiplication gives us:
The minnimum number of operations is:4660
The first parenthesis go after the 1st matrix
The second parenthesis go after the 5th matrix
The third parenthesis go after the 4th matrix
The third parenthesis go after the 3rd matrix
The parenthesis would look like:
         A((((BC)D)E)F)
```

5. Implement BFS on Adjacency List

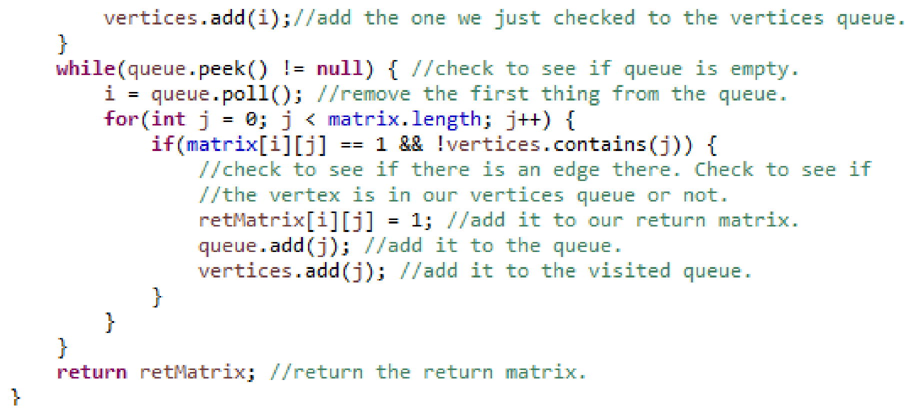


```
/**
 * Breadth-First Search for the Adjacency List.
 * This will print out all of the necessary data and stuff.
 * I used the pseudocode from the book to code this.
 * @param vertices; The vertices in our graph.
 * @param start; The starting vertex.
 */
public void bFSL(Node[] vertices, Node start) {
    start.visit = 0; //set visit for the start vertex as an intermediary integer.
    start.distance = 0; //set distance for the start vertex as 0.
    start.prev = null; //set previous for the start vertex as null.
    System.out.println("We started at vertex " + start.vertex + ".");
    System.out.println("It's distance from the start is 0.");
    System.out.println(" "); //All of the info for the start vertex printed.
    LinkedList<Node> queue = new LinkedList<Node>(); //create a queue.
    //Since we don't want a priority queue, we are using a linked list which
    //includes all of the queue methods.
    queue.add(start); //add the start vertex to our queue.
    while(queue.peek() != null) { //while the queue is not empty...
        Node node = queue.poll(); //remove the head of the queue.
        for(Node vertex : list[node.getIndex(node.vertex)]) { //for every vertex
            //in the list that is in the sport of the vertex.
            if(vertex.visit == -1) { //check to see if the vertex has been visited.
                vertex.visit = 0; //set visit for the vertex as an intermediary integer.
                vertex.distance = node.distance + 1; //set the distance of the vertex to
                //the distance of the previous node plus one.
                vertex.prev = node; //set the previous node as the node we removed from the queue.
                System.out.println("The next vertex we visited is " + vertex.vertex + ".");
                System.out.println("It's distance from the start is " + vertex.distance + ".");
                System.out.println("It's predecessor is " + vertex.prev.vertex + ".");
                //print all of the info for the vertex we just visited.
                queue.add(vertex); //add the vertex we have visited to our queue.
                System.out.println(" ");
            }
        }
        node.visit = 1; //set the visit to the final visit to show we have looked
        //at all of the vertices that are adjacent to it.
    }
```

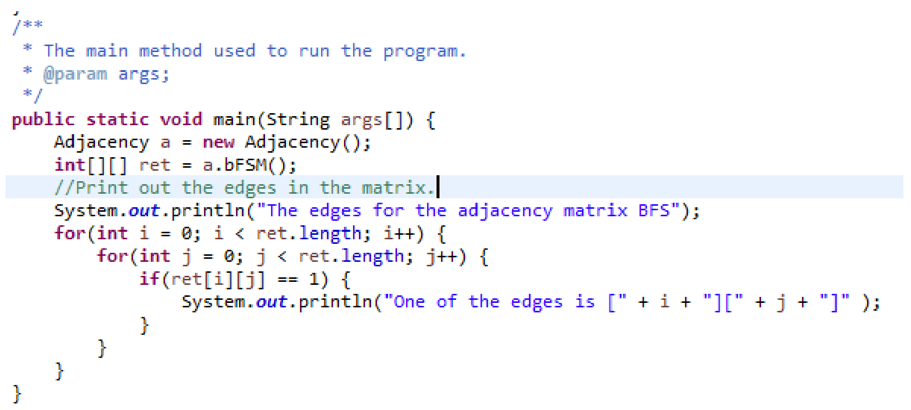
6. *Implement BFS on Adjacency Matrix*



```
/**
 * The Breadth-First Search for the adjacency matrix.
 * @return retMatrix; The matrix with the adjacencies for the BFS.
 */
public int[][] bFSM() {
    int[][] retMatrix = new int[14][14];
    retMatrix = init(retMatrix); //initialize our return matrix as 0.
    PriorityQueue<Integer> queue = new PriorityQueue<Integer>();
    //The queue with the vertices that have been added.
    PriorityQueue<Integer> vertices = new PriorityQueue<Integer>();
    //The queue with the vertices we have visited.
    int i = 0; //set the start matrix as 0.
    queue.add(i); //add i to our queue.
    for(int j = 1; j < matrix.length; j++) {
        if(matrix[i][j] == 1) {
            retMatrix[i][j] = 1; //add the adjacencies from the start
            //matrix.
            queue.add(j);//add the vertices we have added to our BFS.
            vertices.add(j); //add it to the visited queue.
        }
```

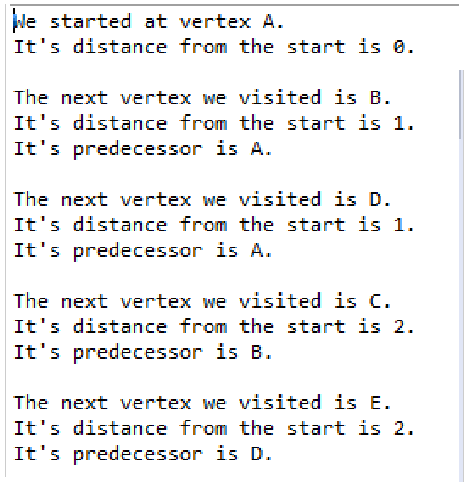


```java
            vertices.add(i);//add the one we just checked to the vertices queue.
        }
        while(queue.peek() != null) { //check to see if queue is empty.
            i = queue.poll(); //remove the first thing from the queue.
            for(int j = 0; j < matrix.length; j++) {
                if(matrix[i][j] == 1 && !vertices.contains(j)) {
                    //check to see if there is an edge there. Check to see if
                    //the vertex is in our vertices queue or not.
                    retMatrix[i][j] = 1; //add it to our return matrix.
                    queue.add(j); //add it to the queue.
                    vertices.add(j); //add it to the visited queue.
                }
            }
        }
        return retMatrix; //return the return matrix.
    }
```



```java
    /**
     * The main method used to run the program.
     * @param args;
     */
    public static void main(String args[]) {
        Adjacency a = new Adjacency();
        int[][] ret = a.bFSM();
        //Print out the edges in the matrix.
        System.out.println("The edges for the adjacency matrix BFS");
        for(int i = 0; i < ret.length; i++) {
            for(int j = 0; j < ret.length; j++) {
                if(ret[i][j] == 1) {
                    System.out.println("One of the edges is [" + i + "][" + j + "]" );
                }
            }
        }
    }
```

7. *Output for BFS on Adjacency List*



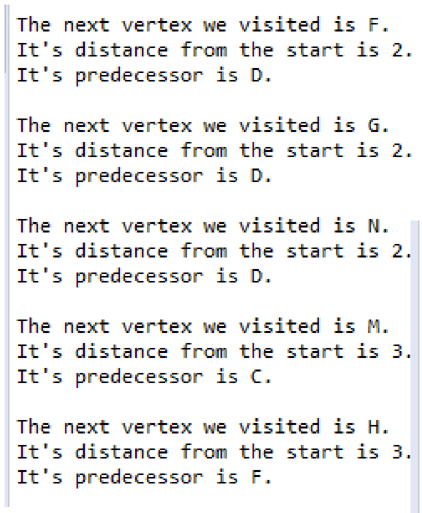
```
We started at vertex A.
It's distance from the start is 0.

The next vertex we visited is B.
It's distance from the start is 1.
It's predecessor is A.

The next vertex we visited is D.
It's distance from the start is 1.
It's predecessor is A.

The next vertex we visited is C.
It's distance from the start is 2.
It's predecessor is B.

The next vertex we visited is E.
It's distance from the start is 2.
It's predecessor is D.
```



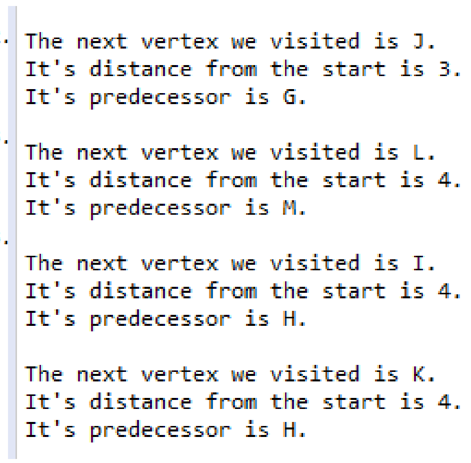
```
The next vertex we visited is F.
It's distance from the start is 2.
It's predecessor is D.

The next vertex we visited is G.
It's distance from the start is 2.
It's predecessor is D.

The next vertex we visited is N.
It's distance from the start is 2.
It's predecessor is D.

The next vertex we visited is M.
It's distance from the start is 3.
It's predecessor is C.

The next vertex we visited is H.
It's distance from the start is 3.
It's predecessor is F.
```



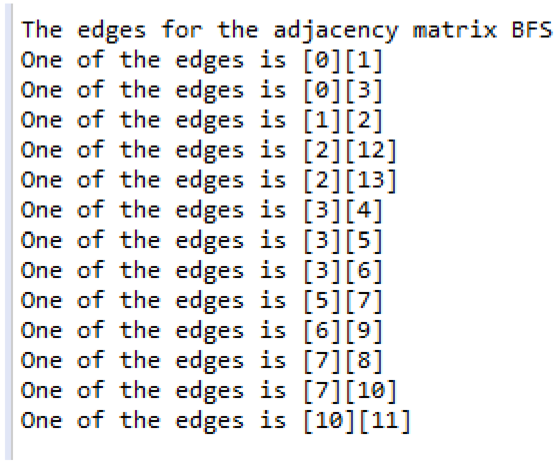
```
The next vertex we visited is J.
It's distance from the start is 3.
It's predecessor is G.

The next vertex we visited is L.
It's distance from the start is 4.
It's predecessor is M.

The next vertex we visited is I.
It's distance from the start is 4.
It's predecessor is H.

The next vertex we visited is K.
It's distance from the start is 4.
It's predecessor is H.
```

8. *Output for BFS an Adjacency Matrix*



```
The edges for the adjacency matrix BFS
One of the edges is [0][1]
One of the edges is [0][3]
One of the edges is [1][2]
One of the edges is [2][12]
One of the edges is [2][13]
One of the edges is [3][4]
One of the edges is [3][5]
One of the edges is [3][6]
One of the edges is [5][7]
One of the edges is [6][9]
One of the edges is [7][8]
One of the edges is [7][10]
One of the edges is [10][11]
```

Analysis:

(1-4): For the Matrix Chain Multiplication Algorithms I used the pseudocode from the book to code it. I didn't think it was too bad. For the matrix given I did the math to make sure that the answer was correct. I did do it for the one on the worksheet but I wanted to be 100% sure that it was going to work on other ones too.

(5 and 7): For the Breadth-First Search algorithm on the adjacency list I used the pseudocode from the book and just changed the .color part in the node to an integer field that I can change from -1 to 1 as we visit them. The output is much nicer that the other breadth-first search because this one contained Nodes so I could more easily store everything. I didn't think it was too bad since I had the pseudocode… but yeah.

(6 and 8): For the Breadth-First Search algorithm on the adjacency matrix I used my brain to come up with a way to find the specific edges necessary for the search. I had to use 2 different priority queues to make sure that we didn't "visit" the same nodes twice or make a cycle. I'm not really sure if this was a very "good" way to do it, but I really couldn't think of any other way and I really didn't want to Google it. I think this was one of the hardest parts of the lab because the pseudocode wasn't given to us.

For both the BFS algorithms I checked my answer by doing the problem. This way when I got the return values I could make sure they were actually correct. 😊