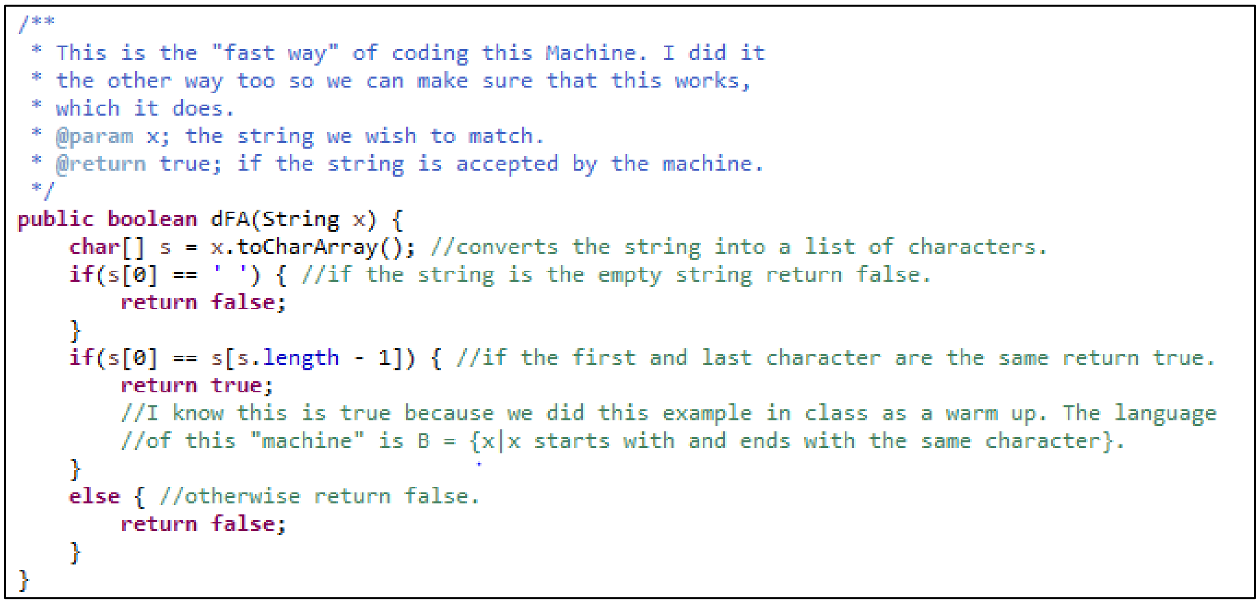
Tiffany Jansen
CS 361
Lab 4

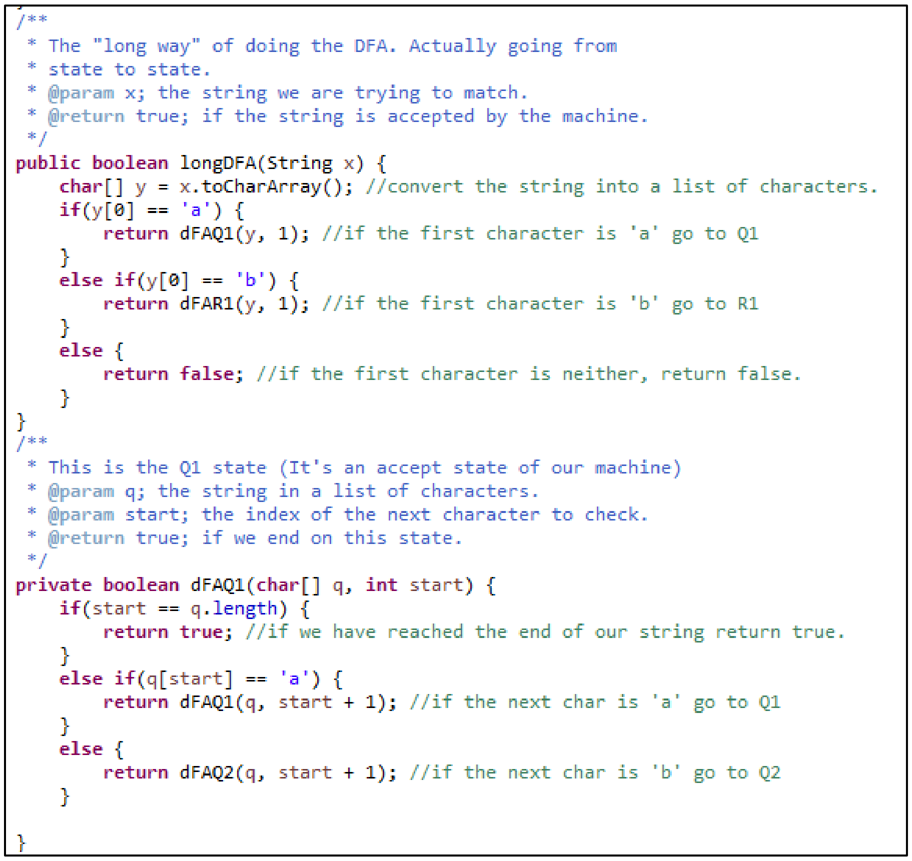
Lab Report

DFA:



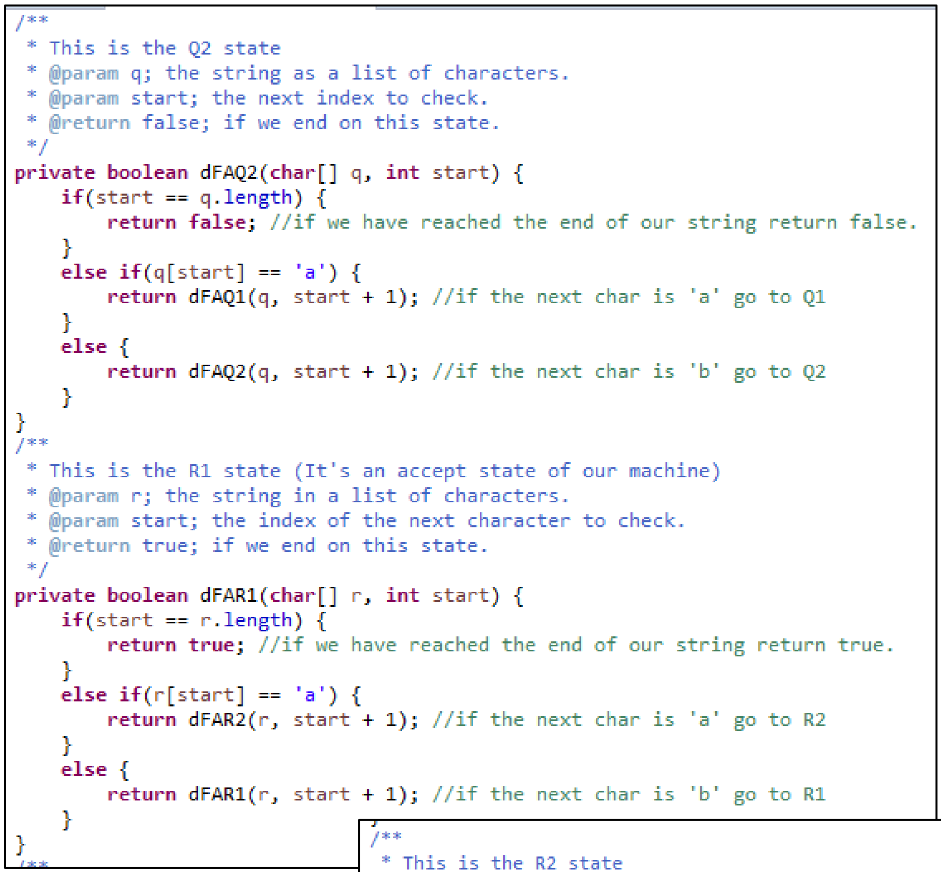
```java
/**
 * This is the "fast way" of coding this Machine. I did it
 * the other way too so we can make sure that this works,
 * which it does.
 * @param x; the string we wish to match.
 * @return true; if the string is accepted by the machine.
 */
public boolean dFA(String x) {
    char[] s = x.toCharArray(); //converts the string into a list of characters.
    if(s[0] == ' ') { //if the string is the empty string return false.
        return false;
    }
    if(s[0] == s[s.length - 1]) { //if the first and last character are the same return true.
        return true;
        //I know this is true because we did this example in class as a warm up. The language
        //of this "machine" is B = {x|x starts with and ends with the same character}.
    }
    else { //otherwise return false.
        return false;
    }
}
```

**I coded this in 2 ways… so here is the 2nd way:

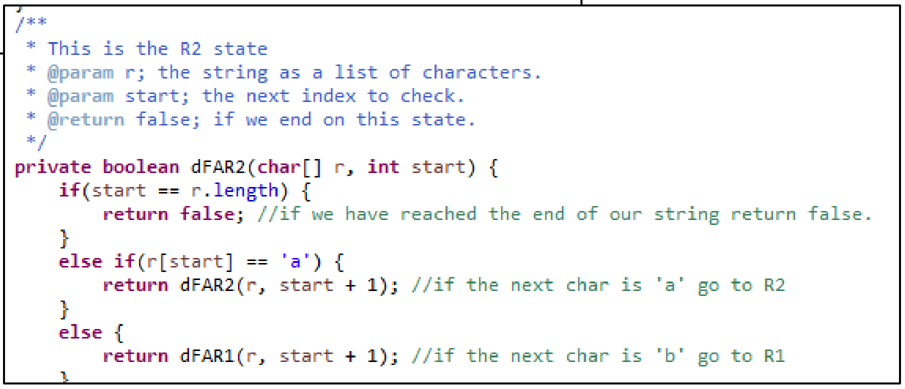


```java
/**
 * The "long way" of doing the DFA. Actually going from
 * state to state.
 * @param x; the string we are trying to match.
 * @return true; if the string is accepted by the machine.
 */
public boolean longDFA(String x) {
    char[] y = x.toCharArray(); //convert the string into a list of characters.
    if(y[0] == 'a') {
        return dFAQ1(y, 1); //if the first character is 'a' go to Q1
    }
    else if(y[0] == 'b') {
        return dFAR1(y, 1); //if the first character is 'b' go to R1
    }
    else {
        return false; //if the first character is neither, return false.
    }
}
/**
 * This is the Q1 state (It's an accept state of our machine)
 * @param q; the string in a list of characters.
 * @param start; the index of the next character to check.
 * @return true; if we end on this state.
 */
private boolean dFAQ1(char[] q, int start) {
    if(start == q.length) {
        return true; //if we have reached the end of our string return true.
    }
    else if(q[start] == 'a') {
        return dFAQ1(q, start + 1); //if the next char is 'a' go to Q1
    }
    else {
        return dFAQ2(q, start + 1); //if the next char is 'b' go to Q2
    }

}
```

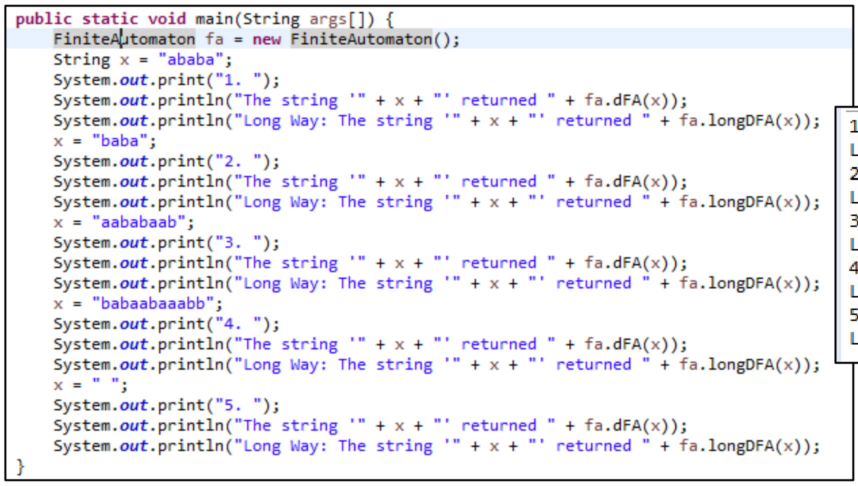


```java
/**
 * This is the Q2 state
 * @param q; the string as a list of characters.
 * @param start; the next index to check.
 * @return false; if we end on this state.
 */
private boolean dFAQ2(char[] q, int start) {
    if(start == q.length) {
        return false; //if we have reached the end of our string return false.
    }
    else if(q[start] == 'a') {
        return dFAQ1(q, start + 1); //if the next char is 'a' go to Q1
    }
    else {
        return dFAQ2(q, start + 1); //if the next char is 'b' go to Q2
    }
}
/**
 * This is the R1 state (It's an accept state of our machine)
 * @param r; the string in a list of characters.
 * @param start; the index of the next character to check.
 * @return true; if we end on this state.
 */
private boolean dFAR1(char[] r, int start) {
    if(start == r.length) {
        return true; //if we have reached the end of our string return true.
    }
    else if(r[start] == 'a') {
        return dFAR2(r, start + 1); //if the next char is 'a' go to R2
    }
    else {
        return dFAR1(r, start + 1); //if the next char is 'b' go to R1
    }
}
```

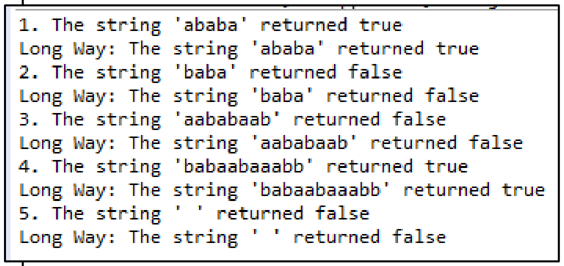


```java
/**
 * This is the R2 state
 * @param r; the string as a list of characters.
 * @param start; the next index to check.
 * @return false; if we end on this state.
 */
private boolean dFAR2(char[] r, int start) {
    if(start == r.length) {
        return false; //if we have reached the end of our string return false.
    }
    else if(r[start] == 'a') {
        return dFAR2(r, start + 1); //if the next char is 'a' go to R2
    }
    else {
        return dFAR1(r, start + 1); //if the next char is 'b' go to R1
```

Main Method/Output:

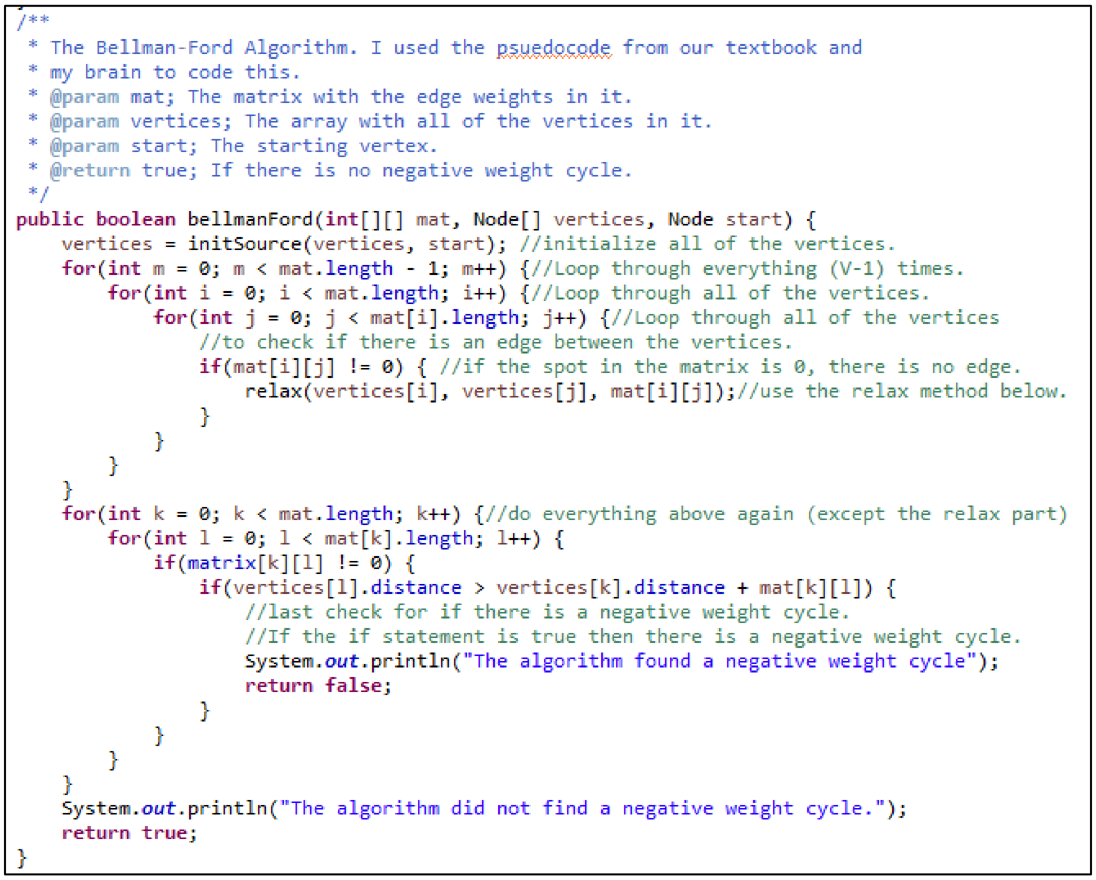


```java
public static void main(String args[]) {
    FiniteAutomaton fa = new FiniteAutomaton();
    String x = "ababa";
    System.out.print("1. ");
    System.out.println("The string '" + x + "' returned " + fa.dFA(x));
    System.out.println("Long Way: The string '" + x + "' returned " + fa.longDFA(x));
    x = "baba";
    System.out.print("2. ");
    System.out.println("The string '" + x + "' returned " + fa.dFA(x));
    System.out.println("Long Way: The string '" + x + "' returned " + fa.longDFA(x));
    x = "aababaab";
    System.out.print("3. ");
    System.out.println("The string '" + x + "' returned " + fa.dFA(x));
    System.out.println("Long Way: The string '" + x + "' returned " + fa.longDFA(x));
    x = "babaabaaabb";
    System.out.print("4. ");
    System.out.println("The string '" + x + "' returned " + fa.dFA(x));
    System.out.println("Long Way: The string '" + x + "' returned " + fa.longDFA(x));
    x = " ";
    System.out.print("5. ");
    System.out.println("The string '" + x + "' returned " + fa.dFA(x));
    System.out.println("Long Way: The string '" + x + "' returned " + fa.longDFA(x));
}
```

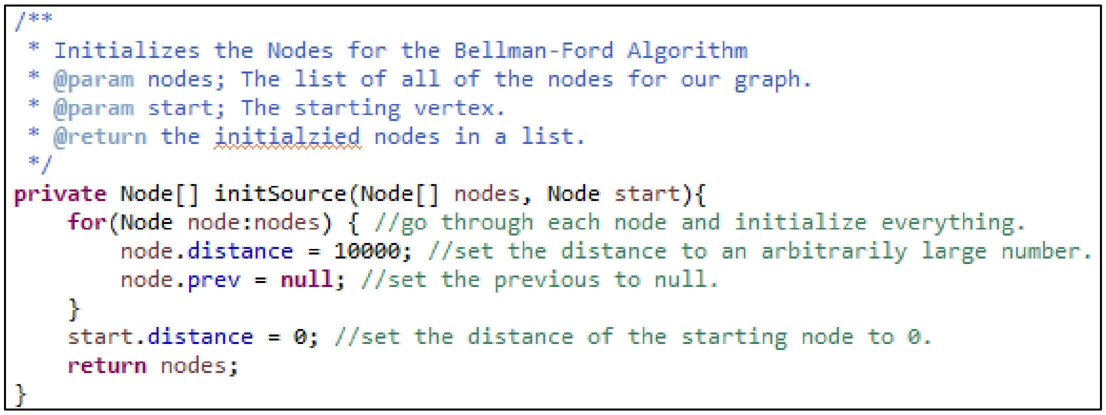


```
1. The string 'ababa' returned true
Long Way: The string 'ababa' returned true
2. The string 'baba' returned false
Long Way: The string 'baba' returned false
3. The string 'aababaab' returned false
Long Way: The string 'aababaab' returned false
4. The string 'babaabaaabb' returned true
Long Way: The string 'babaabaaabb' returned true
5. The string ' ' returned false
Long Way: The string ' ' returned false
```

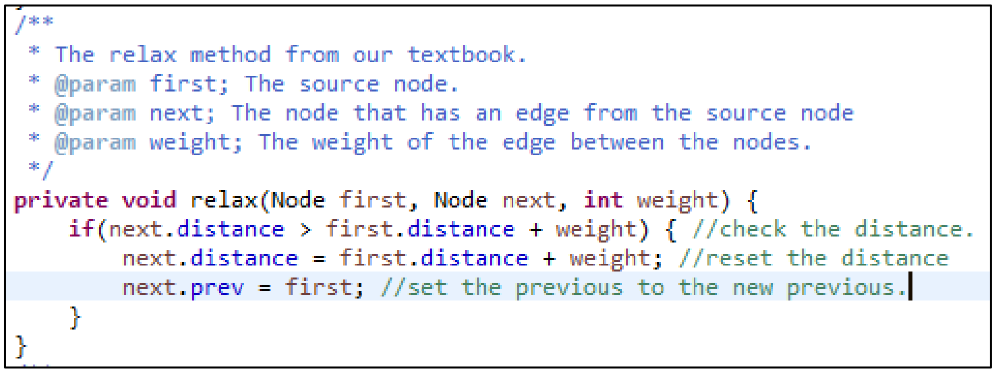
Bellman-Ford Algorithm:



```java
/**
 * The Bellman-Ford Algorithm. I used the psuedocode from our textbook and
 * my brain to code this.
 * @param mat; The matrix with the edge weights in it.
 * @param vertices; The array with all of the vertices in it.
 * @param start; The starting vertex.
 * @return true; If there is no negative weight cycle.
 */
public boolean bellmanFord(int[][] mat, Node[] vertices, Node start) {
    vertices = initSource(vertices, start); //initialize all of the vertices.
    for(int m = 0; m < mat.length - 1; m++) {//Loop through everything (V-1) times.
        for(int i = 0; i < mat.length; i++) {//Loop through all of the vertices.
            for(int j = 0; j < mat[i].length; j++) {//Loop through all of the vertices
                //to check if there is an edge between the vertices.
                if(mat[i][j] != 0) { //if the spot in the matrix is 0, there is no edge.
                    relax(vertices[i], vertices[j], mat[i][j]);//use the relax method below.
                }
            }
        }
    }
    for(int k = 0; k < mat.length; k++) {//do everything above again (except the relax part)
        for(int l = 0; l < mat[k].length; l++) {
            if(matrix[k][l] != 0) {
                if(vertices[l].distance > vertices[k].distance + mat[k][l]) {
                    //last check for if there is a negative weight cycle.
                    //If the if statement is true then there is a negative weight cycle.
                    System.out.println("The algorithm found a negative weight cycle");
                    return false;
                }
            }
        }
    }
    System.out.println("The algorithm did not find a negative weight cycle.");
    return true;
}
```

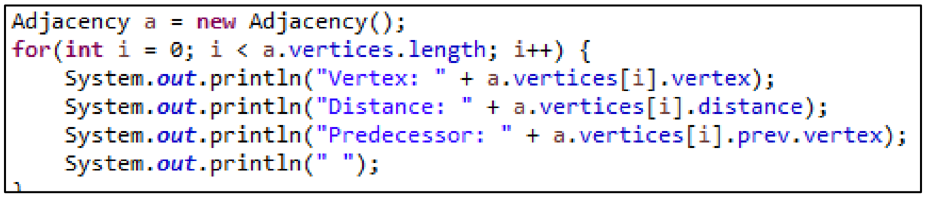


```java
/**
 * Initializes the Nodes for the Bellman-Ford Algorithm
 * @param nodes; The list of all of the nodes for our graph.
 * @param start; The starting vertex.
 * @return the initialzied nodes in a list.
 */
private Node[] initSource(Node[] nodes, Node start){
    for(Node node:nodes) { //go through each node and initialize everything.
        node.distance = 10000; //set the distance to an arbitrarily large number.
        node.prev = null; //set the previous to null.
    }
    start.distance = 0; //set the distance of the starting node to 0.
    return nodes;
}
```

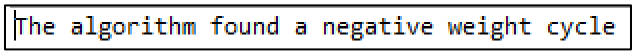


```java
/**
 * The relax method from our textbook.
 * @param first; The source node.
 * @param next; The node that has an edge from the source node
 * @param weight; The weight of the edge between the nodes.
 */
private void relax(Node first, Node next, int weight) {
    if(next.distance > first.distance + weight) { //check the distance.
        next.distance = first.distance + weight; //reset the distance
        next.prev = first; //set the previous to the new previous.
    }
}
```

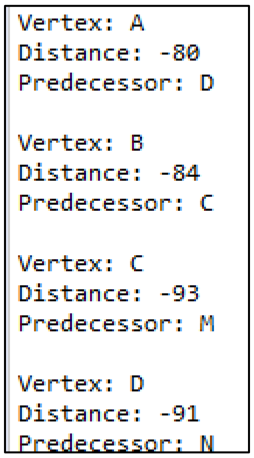
Main Method/Output:



```
Adjacency a = new Adjacency();
for(int i = 0; i < a.vertices.length; i++) {
    System.out.println("Vertex: " + a.vertices[i].vertex);
    System.out.println("Distance: " + a.vertices[i].distance);
    System.out.println("Predecessor: " + a.vertices[i].prev.vertex);
    System.out.println(" ");
}
```



```
The algorithm found a negative weight cycle
```

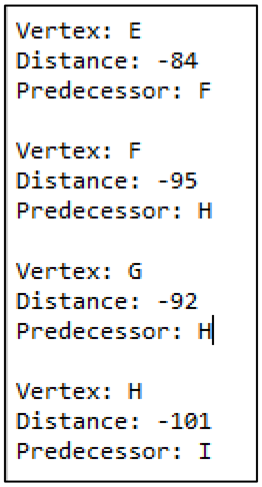


```
Vertex: A
Distance: -80
Predecessor: D

Vertex: B
Distance: -84
Predecessor: C

Vertex: C
Distance: -93
Predecessor: M

Vertex: D
Distance: -91
Predecessor: N
```

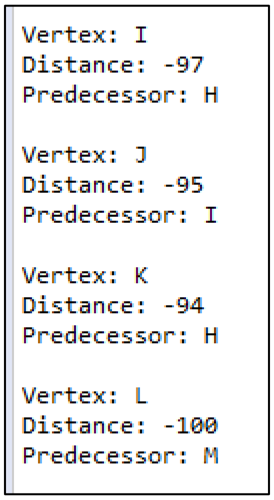


```
Vertex: E
Distance: -84
Predecessor: F

Vertex: F
Distance: -95
Predecessor: H

Vertex: G
Distance: -92
Predecessor: H

Vertex: H
Distance: -101
Predecessor: I
```

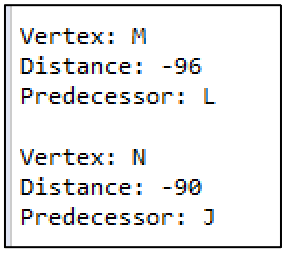


```
Vertex: I
Distance: -97
Predecessor: H

Vertex: J
Distance: -95
Predecessor: I

Vertex: K
Distance: -94
Predecessor: H

Vertex: L
Distance: -100
Predecessor: M
```



```
Vertex: M
Distance: -96
Predecessor: L

Vertex: N
Distance: -90
Predecessor: J
```

Notes:

I wasn't sure if the first way I did the DFA was good enough… so I did it a second way. Other than that everything is commented so… yeah. If you have any questions let me know. ☺