```
// runs Fleury's algorithm to find an Euler
circuit starting at a specified node

Fleurify (int start) {

  // stores a copy of matrix[][] to keep track
  // of the original graph adjacency matrix

  int temp[][];

  // keeps track of the current node that
is traveled to

    int current;

  // stores the endpoint of the edge
    that was taken
                          ↙ initialize to random
                            variable to suppress
                            compiler warning
    int endpoint = -1;

  // stores the Euler circuit as it is
    built up

    ArrayList<Integer> circuit;
```

// Allocating space to store the copy of the original adjacency matrix

```
temp= new int [N][N];
```

N is number of Nodes in graph

```
// Store copy
temp= matrix;
```

// Gets the number of edges in the graph and stores it (uses The First Theorem of Graph Theory)

```
graph_size = size_of_graph();
```

// Initializes some of the structures used to keep track of Fleury and prim's algo.

```
init_structures_fleury();
```

Stores all nodes in a list to keep track of visited nodes in Visited arraylist

↑ Initializes V-S_master array list which keeps track of nodes that should not be visited any more if they have a degree of 0

// Start the circuit at the specified start node

```
        current = start;
// Add the start node to the circuit

        circuit.add(current);

// Implementation of Fleury's algorithm.
// Continues looping as long as there are
  edges to visit

    while( graph_size > 0) {

    // Choose an edge that doesn't
    disconnect the graph (run's prim's algo)

for(int del_vert =0; del_vert < N; del_vert++){
    if((temp[current][del_vert] == 1) &&
      ( prims_algo_check(current, del_vert, start) ) {

          endPoint = del_vert;
          break;
      }
}

  // Add valid endpoint in edge
that does not disconnect graph
```

```
        circuit.add(endPoint);

        // update copy of adjacency matrix
        of available edges in graph

        remove_edge_in_matrix(current,
                              endPoint,
                              temp);

        // update the current node in circuit

        Current = endpoint

        // update number of edges left to travel

        graph_size--;

    // Writes Euler circuit to text file: output.txt
    writeArrayListToFile("output.txt", circuit);

}
```

// Performs prim's algorithm to ensure that choosing a certain edge in Fleury's algorithm doesn't disconnect the graph.

// The parameters x and y refer to the edge chosen of the form [x, y] where x is the left endpoint and y is the right endpoint of the edge

```
boolean prims_algo_check(int x, int y, int start)
{
```

// flag to keep track when a valid connection is still present in the edge being checked if it continues to connect the graph upon its removal ← initially false

```
int flag = 0;
```

// initialize structures for prim's algorithm

1) initializes an empty arraylist called S
   As prim's algo runs, S is populated with vertices that are visited
2) initializes an arraylist called V-S which represents the set of vertices that have not yet been visited in prim's algo.

V_S is initialized to store all valid nodes in a graph

init_structures_prims();

// Adds the endpoint into the set of visited nodes S and removes the endpoint from the set of non-visited nodes V_S

update_lists_prims(y)

// Removes the considered edge to be deleted from the set of all adjacent vertices "edges" which is defined as:
ArrayList<Hashset<Int>> edges.

remore_edge_in_set(x,y);

// Checks if the degree of the left endpoint X is ∅ after deleting the edge from the edge set edges.
If the degree is ∅, that node

is not considered in prim's algo since it can no longer be traveled to. V_S_Master and V_S are updated accordingly by removing that node from each set.

```
isDegreeZero(x);
```

// Checks if the current edge wraps back to the start node and if all other nodes have been visited. If so, then return true

```
if( y_is_start(y, start) && all_verts_visited())
        return True;
else if( y_is_start(y,start) && getDegree(y) == 1){
```

// If the edge leads back to the start but this node still has a degree of 1 then this node can not be visited. add Back the edge and return false

// adds the edgeback to the matrix and ArrayList<Hashset<Int>> edges

```
            addBack(x,y)
            return false;
        }

// Loop as long as there are vertices
 in the graph to visit

while ( !V_S.isEmpty() ) {

    // Loops through the visited nodes
    for(int i=0; i < S.size(); i++) {

        // Reset flag each loop to
        catch if no visited vertices can
        reach any unvisited vertices

        flag=0;

// Loops through all unvisited vertices

for( int j=0; j < V_S.size(); j++ ) {

    // If a single adjacent vertex is
    found in the unvisited vertices V_S
```

that is adjacent to a visited vertex in the set S then we can continue with prim's

```
if( foundLink(i,j)) {

    // Add the adjacent unvisited vertex to S , remove it from the unvisited set V-S and mark that node as visited in the visited arrayList.

    update_lists_prims(V-S.get(j))j

    // set flag to one indicating that an unvisited vertex is adjacent to one of the visited vertices in

        flag = 1;
        breakj
    }
}

    // keep breaking out until we get to the while loop
if(flag == 1 )
    breakj
```

```
}

// At this point, we looked through all
adjacent vertices and no unvisited nodes
can be reached. Thus, the deletion of
the edge in question disconnects the graph.

    if( flag == 0) {

        // add the edge back

            addBack(x,y);
            return false;
        }
    }

return true;
}
```