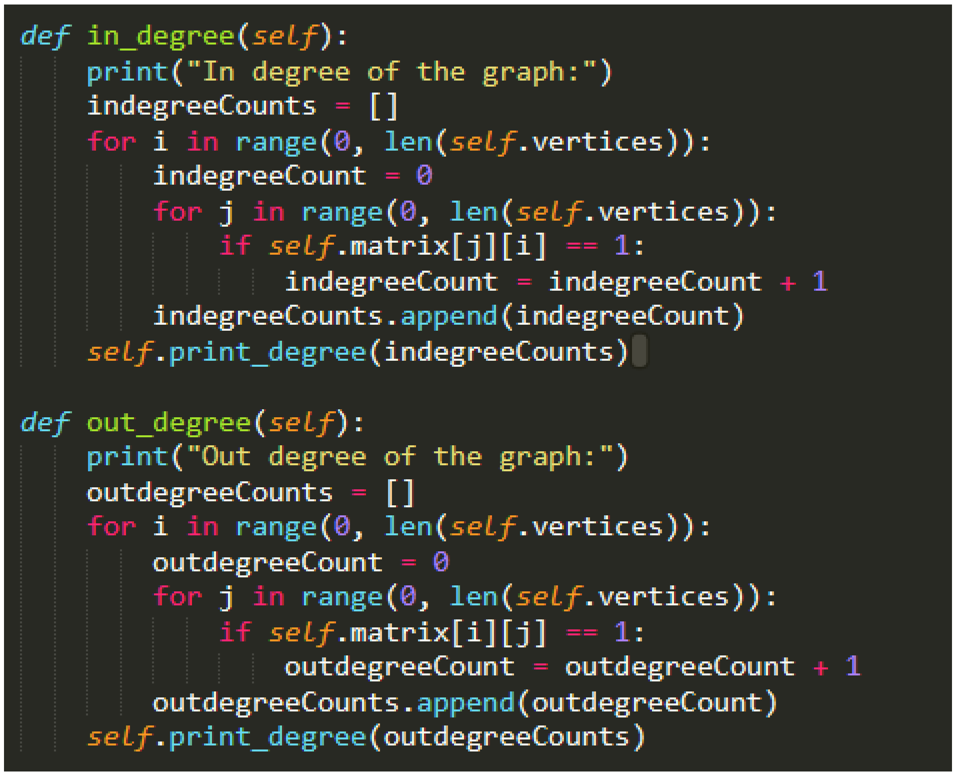
Problem 1:

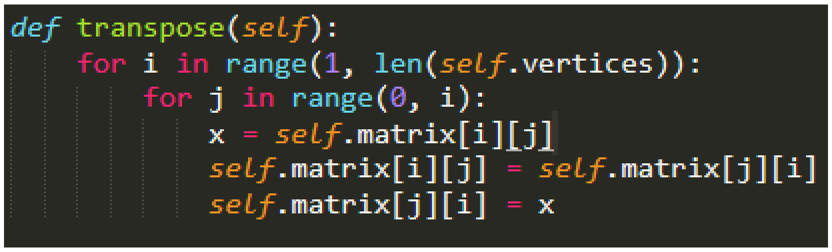


```python
def in_degree(self):
    print("In degree of the graph:")
    indegreeCounts = []
    for i in range(0, len(self.vertices)):
        indegreeCount = 0
        for j in range(0, len(self.vertices)):
            if self.matrix[j][i] == 1:
                indegreeCount = indegreeCount + 1
        indegreeCounts.append(indegreeCount)
    self.print_degree(indegreeCounts)

def out_degree(self):
    print("Out degree of the graph:")
    outdegreeCounts = []
    for i in range(0, len(self.vertices)):
        outdegreeCount = 0
        for j in range(0, len(self.vertices)):
            if self.matrix[i][j] == 1:
                outdegreeCount = outdegreeCount + 1
        outdegreeCounts.append(outdegreeCount)
    self.print_degree(outdegreeCounts)
```

The above shows the code for determining the indegrees and outdegrees of vertices in the graph represented by an adjacency matrix. The out_degree function works by considering how connections from a vertex in a row of the adjacency matrix to another vertex in a corresponding column of the adjacency matrix are represented by a 1. With this consideration, the algorithm simply needs to iterate through the adjacency matrix and record the number of 1's for each row in the matrix, thereby calculating the outdegree of each vertex in the matrix. The in_degree function is nearly identical with one key difference. Instead of comparing the rows of the adjacency matrix with its columns, the in_degree function compares the columns of the adjacency matrix with its rows, which determines not what a vertex connects to, but what a vertex is connected to. Each function would have a complexity of O(V^2) where V represents the number of vertices in the matrix. This is because it is necessary to traverse the size of the number of vertices for each vertex in the graph in order to completely explore the adjacency matrix.

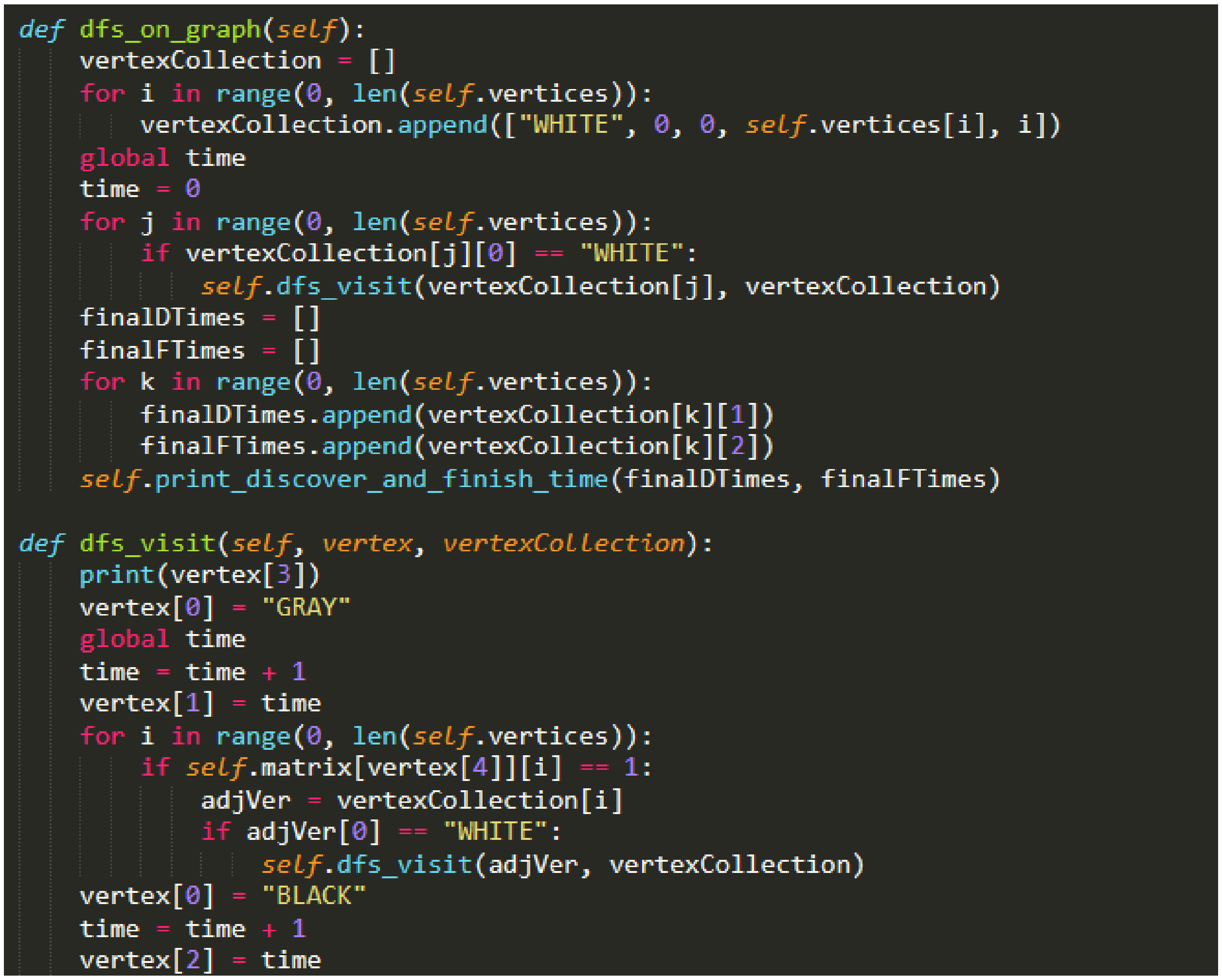
Problem 2:


```python
def transpose(self):
    for i in range(1, len(self.vertices)):
        for j in range(0, i):
            x = self.matrix[i][j]
            self.matrix[i][j] = self.matrix[j][i]
            self.matrix[j][i] = x
```

The above shows the code for producing the transpose of the graph. In order to reverse the direction of the connections between vertices, the values must be swapped across the diagonal of the adjacency matrix. To do so, we iterate through the list of vertices starting at index 1, since the encoding (0, 0) represents a connection of vertex 1 to itself, which cannot happen in a digraph. Even if allowed, swapping at the same position would be pointless since the starting value of that portion of the diagonal would not change. Then, at each vertex, it finds the values of elements on both sides of the diagonal along the same connection between vertices and swaps those values. The distance between the starting value and the diagonal is determined by the index of the outer loop. The complexity of the algorithm would likely be O(V + E) since, in addition to exploring each of the vertices (except one at the start), the algorithm also attempts to find any possible connections, or edges, between those vertices. The size of V + E would be the same size as the number of elements above (or below if looking at elements below the diagonal) the diagonal in the adjacency matrix.d

Problem 3:
Edge classifications

Q to S: Tree Edge
S to V: Tree Edge
V to W: Tree Edge
Q to T: Tree Edge
T to X: Tree Edge
X to Z: Tree Edge
T to Y: Tree Edge
R to U: Tree Edge
W to S: Back Edge
Z to X: Back Edge
Y to Q: Back Edge
Q to W: Forward Edge
R to Y: Cross Edge
U to Y: Cross Edge

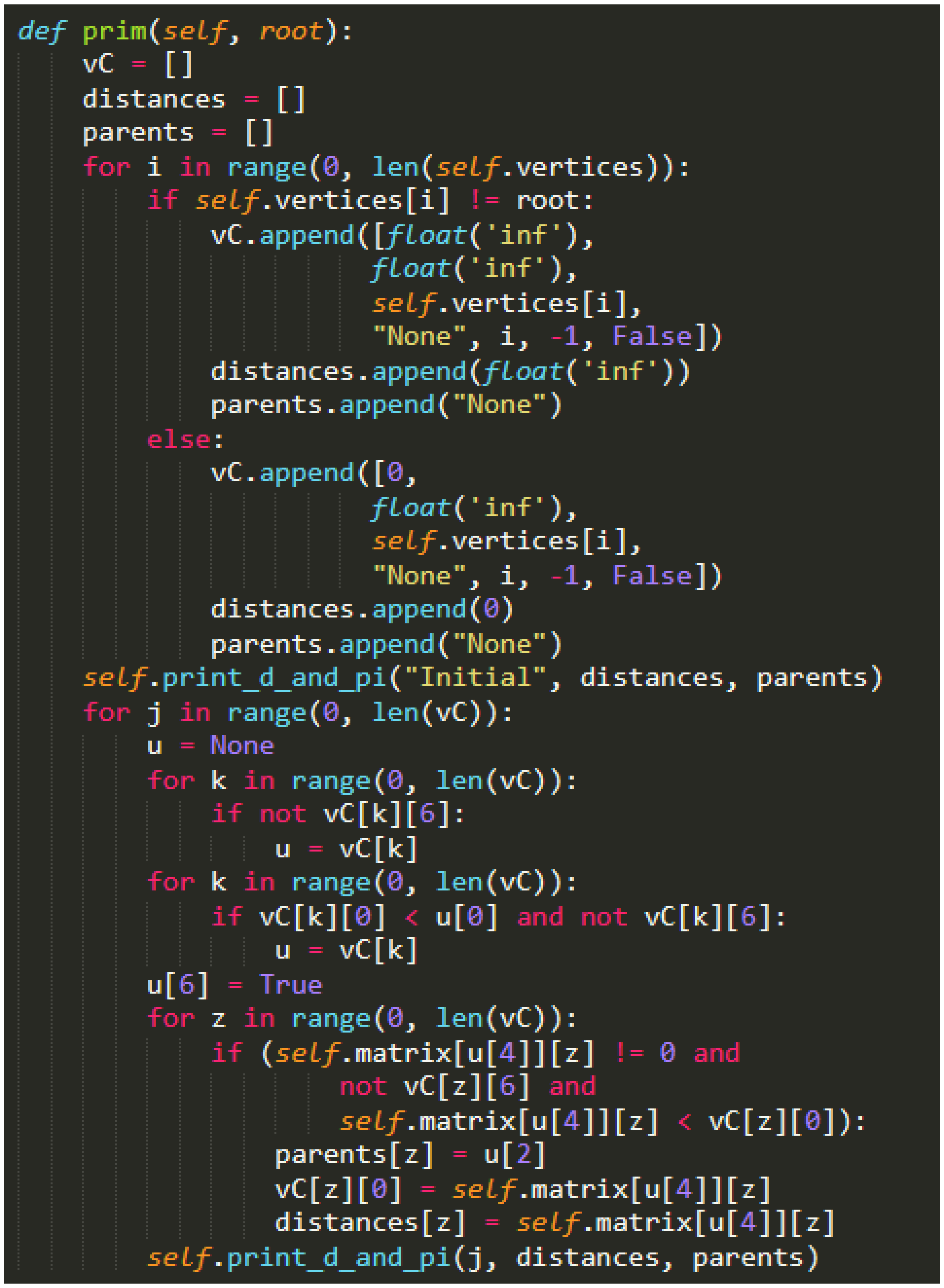


```python
def dfs_on_graph(self):
    vertexCollection = []
    for i in range(0, len(self.vertices)):
        vertexCollection.append(["WHITE", 0, 0, self.vertices[i], i])
    global time
    time = 0
    for j in range(0, len(self.vertices)):
        if vertexCollection[j][0] == "WHITE":
            self.dfs_visit(vertexCollection[j], vertexCollection)
    finalDTimes = []
    finalFTimes = []
    for k in range(0, len(self.vertices)):
        finalDTimes.append(vertexCollection[k][1])
        finalFTimes.append(vertexCollection[k][2])
    self.print_discover_and_finish_time(finalDTimes, finalFTimes)

def dfs_visit(self, vertex, vertexCollection):
    print(vertex[3])
    vertex[0] = "GRAY"
    global time
    time = time + 1
    vertex[1] = time
    for i in range(0, len(self.vertices)):
        if self.matrix[vertex[4]][i] == 1:
            adjVer = vertexCollection[i]
            if adjVer[0] == "WHITE":
                self.dfs_visit(adjVer, vertexCollection)
    vertex[0] = "BLACK"
    time = time + 1
    vertex[2] = time
```

The above is the implementation of the dfs algorithm. This implementation closely represents its depiction in the slides. Worth noting is that this algorithm assumes that the vertices provided in the adjacency matrix are already ordered. Additionally, in order to determine vertices adjacent to the visited vertex, an iteration over all vertices in the adjacency matrix occurs. This is appropriate since, strictly speaking, the complexity of the algorithm is O(V^2).

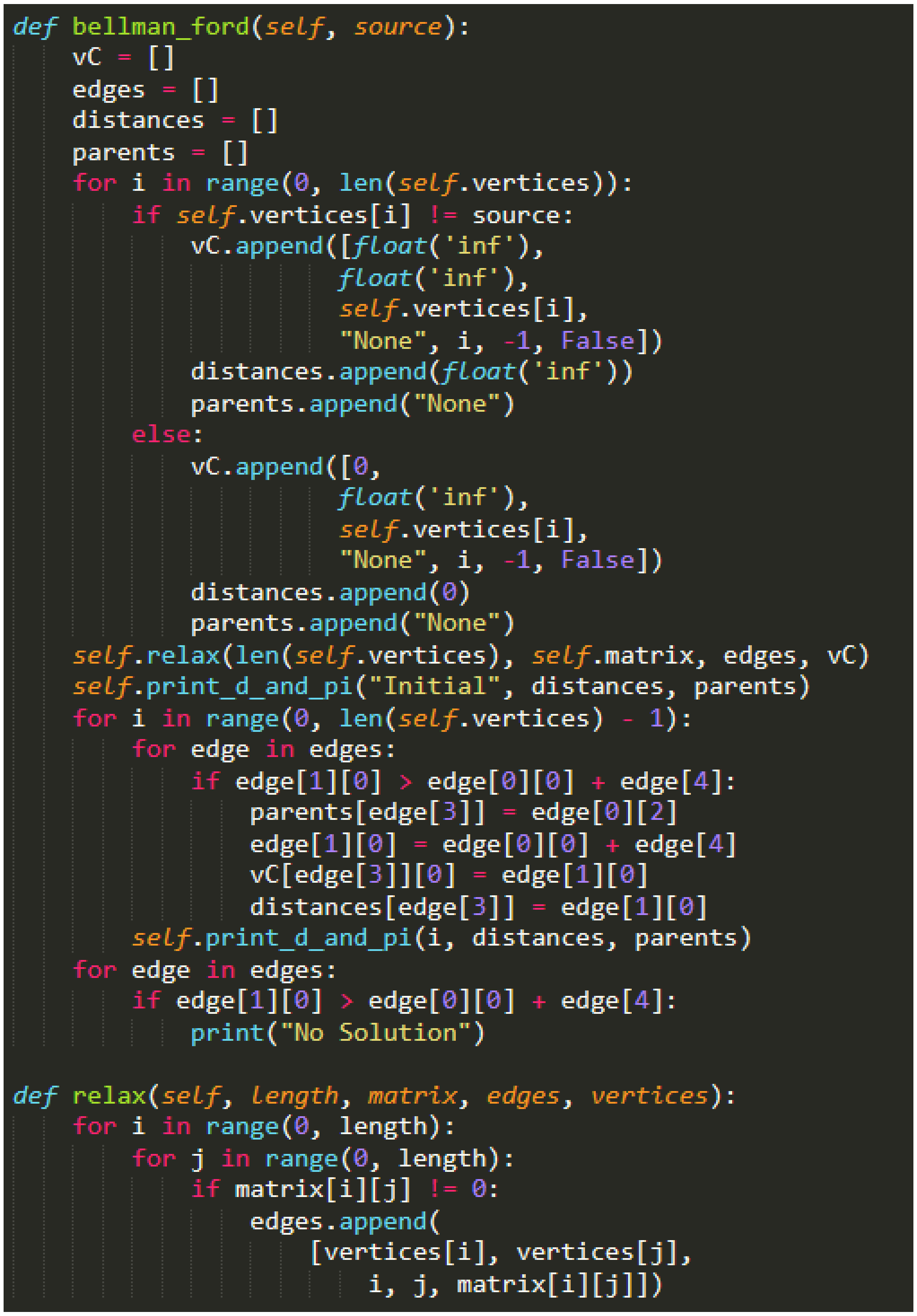
Problem 4:



```
def prim(self, root):
    vC = []
    distances = []
    parents = []
    for i in range(0, len(self.vertices)):
        if self.vertices[i] != root:
            vC.append([float('inf'),
                       float('inf'),
                       self.vertices[i],
                       "None", i, -1, False])
            distances.append(float('inf'))
            parents.append("None")
        else:
            vC.append([0,
                       float('inf'),
                       self.vertices[i],
                       "None", i, -1, False])
            distances.append(0)
            parents.append("None")
    self.print_d_and_pi("Initial", distances, parents)
    for j in range(0, len(vC)):
        u = None
        for k in range(0, len(vC)):
            if not vC[k][6]:
                u = vC[k]
        for k in range(0, len(vC)):
            if vC[k][0] < u[0] and not vC[k][6]:
                u = vC[k]
        u[6] = True
        for z in range(0, len(vC)):
            if (self.matrix[u[4]][z] != 0 and
                    not vC[z][6] and
                    self.matrix[u[4]][z] < vC[z][0]):
                parents[z] = u[2]
                vC[z][0] = self.matrix[u[4]][z]
                distances[z] = self.matrix[u[4]][z]
        self.print_d_and_pi(j, distances, parents)
```

The above is a simplified implementation of Prim’s algorithm for determining what graph edges form an MST. Instead of using a priority queue for storing vertices, this algorithm simply uses an array for storing those vertices. Each vertex contains data of its current distance from its parent, its value (e.g. “G” or “H”), and the value of its parent. In initializing this array, it immediately sets the distance of the provided root node to 0. To simulate popping, each vertex also has a boolean indicating if it has been examined yet. After printing the initial d’s and pi’s, the algorithm iterates over the length of the array to simulate iterating over the vertex queue while the queue is not empty. Then, three loops run inside this loop, each iterating as many times as the length of the vertex array. This will not change the complexity of the algorithm as a whole and it will remain $O(V^2)$. The first inner loop finds a vertex in the collection that has not yet been examined and “popped” from the list. The second inner loop compares the key value of the found vertex with non-popped vertices in the vertex collection and assigns found vertex to the vertex from the vertex collection with a key value smaller than the originally found vertex. The found vertex is then “popped” from the list by setting the appropriate value to True. After this, the final loop searches for vertices adjacent to the found vertex that have not been popped from the list. These vertices must also have weights.

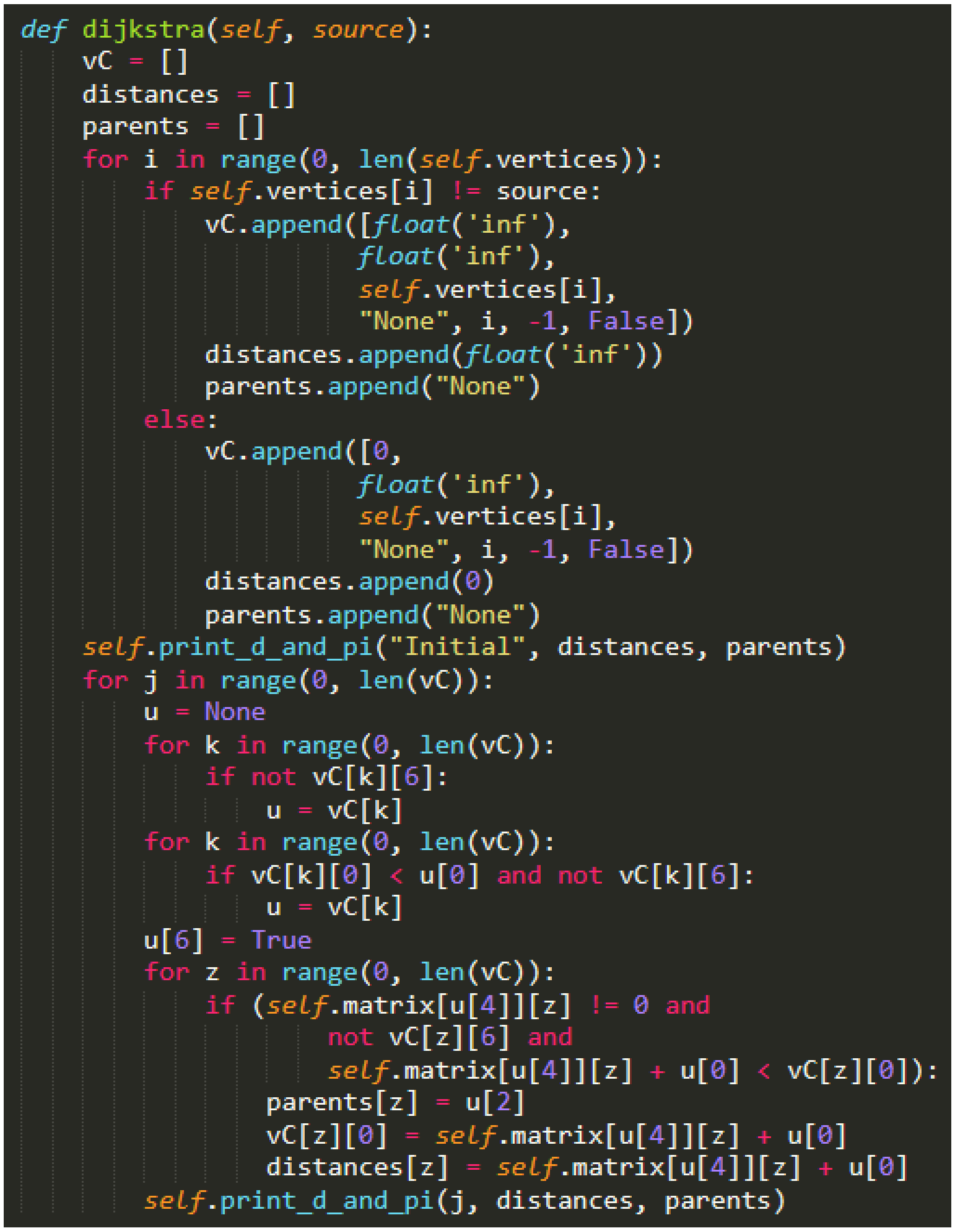
Problem 5:


```
def bellman_ford(self, source):
    vC = []
    edges = []
    distances = []
    parents = []
    for i in range(0, len(self.vertices)):
        if self.vertices[i] != source:
            vC.append([float('inf'),
                       float('inf'),
                       self.vertices[i],
                       "None", i, -1, False])
            distances.append(float('inf'))
            parents.append("None")
        else:
            vC.append([0,
                       float('inf'),
                       self.vertices[i],
                       "None", i, -1, False])
            distances.append(0)
            parents.append("None")
    self.relax(len(self.vertices), self.matrix, edges, vC)
    self.print_d_and_pi("Initial", distances, parents)
    for i in range(0, len(self.vertices) - 1):
        for edge in edges:
            if edge[1][0] > edge[0][0] + edge[4]:
                parents[edge[3]] = edge[0][2]
                edge[1][0] = edge[0][0] + edge[4]
                vC[edge[3]][0] = edge[1][0]
                distances[edge[3]] = edge[1][0]
        self.print_d_and_pi(i, distances, parents)
    for edge in edges:
        if edge[1][0] > edge[0][0] + edge[4]:
            print("No Solution")

def relax(self, length, matrix, edges, vertices):
    for i in range(0, length):
        for j in range(0, length):
            if matrix[i][j] != 0:
                edges.append(
                    [vertices[i], vertices[j],
                     i, j, matrix[i][j]])
```

This code represents the Bellman-Ford algorithm. Slightly misleading is the naming of the function responsible for gathering the edges of the graph such that they can be useable in relaxing the edges. The actual relaxing step occurs in the first instance of the loop over the edges list. Theoretically a O(VE) algorithm, this O(V^2) implementation is still appropriate.

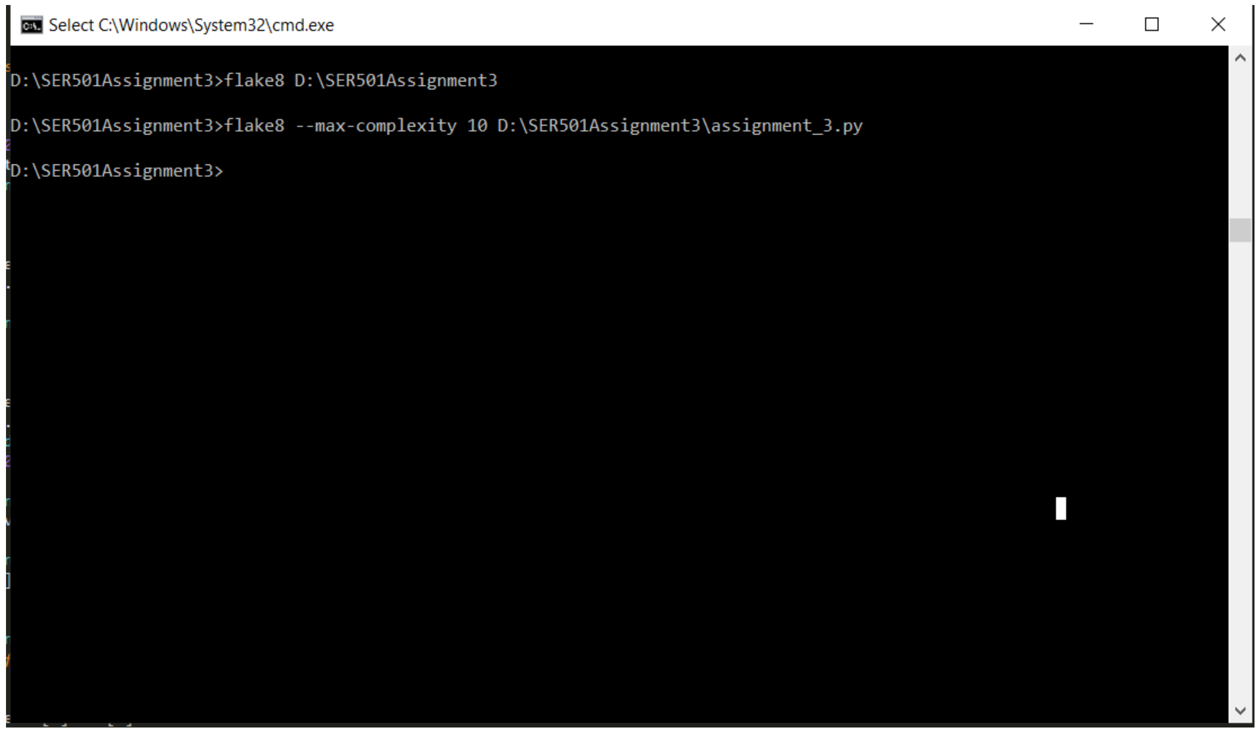
Problem 6:



```python
def dijkstra(self, source):
    vC = []
    distances = []
    parents = []
    for i in range(0, len(self.vertices)):
        if self.vertices[i] != source:
            vC.append([float('inf'),
                       float('inf'),
                       self.vertices[i],
                       "None", i, -1, False])
            distances.append(float('inf'))
            parents.append("None")
        else:
            vC.append([0,
                       float('inf'),
                       self.vertices[i],
                       "None", i, -1, False])
            distances.append(0)
            parents.append("None")
    self.print_d_and_pi("Initial", distances, parents)
    for j in range(0, len(vC)):
        u = None
        for k in range(0, len(vC)):
            if not vC[k][6]:
                u = vC[k]
        for k in range(0, len(vC)):
            if vC[k][0] < u[0] and not vC[k][6]:
                u = vC[k]
        u[6] = True
        for z in range(0, len(vC)):
            if (self.matrix[u[4]][z] != 0 and
                    not vC[z][6] and
                    self.matrix[u[4]][z] + u[0] < vC[z][0]):
                parents[z] = u[2]
                vC[z][0] = self.matrix[u[4]][z] + u[0]
                distances[z] = self.matrix[u[4]][z] + u[0]
        self.print_d_and_pi(j, distances, parents)
```

This simple implementation of Dijkstra’s algorithm is nearly identical to Prim’s algorithm. The only change is the modified relaxing step. The runtime is the same as Prim’s.

Running Flake8:



Select C:\Windows\System32\cmd.exe

```
D:\SER501Assignment3>flake8 D:\SER501Assignment3

D:\SER501Assignment3>flake8 --max-complexity 10 D:\SER501Assignment3\assignment_3.py

D:\SER501Assignment3>
```

This console output demonstrates that the appropriate complexity is maintained and that there are no warnings in the code.