Question 1(Zhenjie Xiong):

//Verified by Xiaoxue Xing

According to textbook, 'an articulation vertex of a graph G is a vertex whose deletion disconnects G'. Then a non-articulation vertex would be what we are looking for.

We will use DFS to find such a vertex in any connected undirected graph. The key is to consider the first vertex in DFS that finishes. So I will do a DFS, but not a complete one. The first vertex which stops being processed will be a non-articulation vertex. Because it is either a leaf or a leaf with back edge pointing back to its parent.

It is correct because it deletes any node with maximum depth, and each other nodes can be reached by a node with less depth. The running time would be O(|V|+|E|) because it's using DFS which has the worst case running time O(|v|+|E|). Specifically, we loop through all of a vertex's adjacent list, calling DFS(v) if it has not been visited. It means that we incur |V| time steps, plus the time incurred to visit adjacent nodes. We have a total of |E| edges, which is O(|V|+|E|).

Question 2(Zhenjie Xiong):

//Verified by Xiaoxue Xing

a) To prove with contradiction, we first assume that G is bipartite and it HAS a cycle of odd length, with vertices $v_1, \dots, v_m$ where m is odd. Then because G is bipartite, we can separate the vertices into V_0 and V_1 where no edges exist in either V_0 or V_1.

   Then $v_1, v_3, v_5, v_7, \dots, v_m$ will be in one of the partition groups because the graph is bipartite. Now, for the cycle to be complete, we need to use the edge $v_m v_1$ at the end of this cycle, which is an edge within a partition group.

   However, this contradict that G is bipartite and then has no such edges within the group. So, the assumption is incorrect. The correct one should be 'if G is bipartite, it has NO simple cycle of odd length'.

b) If G has no simple cycle of odd length, then we can color the vertices in a way that no two

colors are adjacent to each other. These two colors will become our two partition sets.

Think about the DFS tree starting at any node s in the graph. If we color the root with a color, then color its every child the opposite color of its parent. Then, all tree edges will follow the bipartite feature.

Now, back edges exist in an undirected DFS graph. We need to argue that back edges can't exist in a way that connects two vertices of same color. Why is this the case?

Because if a back edge connects two vertices of same color, it implies that the cycle containing this back edge must also be odd, according to the color alternating pattern. But G doesn't have any odd cycles, so this assumption is incorrect. All the edges should have end points that have different color from this structure. Two bipartite sets from the vertices are made and the statement is proved.

c) **Coloring**(G):

Starting at any node s and color it white(opposite: black). Then put s in a queue named A.

While A is not empty:

```
    s = pop(Q);
    For every neighbor x of s:
        If x is not colored:
            color it the opposite color of s and put it into the queue
        if x is colored and have same color as s:
            return 'Not bipartite'
return 'Bipartite'
```

Basically, I first color each node a color that is different from its parent by running a modified BFS. Then in the for loop I check whether its neighbor has the same color as it. If yes then the graph is not bipartite.

If we color through the whole graph with this alternating pattern with respect to its parents. And not a single neighbor it encounters has the same color. Then the graph is bipartite.

<u>Why it's correct:</u>

We try to prove that if G is bipartite, our algorithm returns it. Accordingly, the starting vertex will be either in V_0 or V_1. All the neighbors of s must be the opposite color, with their neighbors having the same property, just the same way as our algorithm which recovers V_0 and V_1.

We need to prove that our algorithm's output is the same as the fact. Here's how we do it, we let G be a not bipartite graph. Then how I partition the vertices into two groups should not intervene with the fact that there will be an edge in one of the groups. Also, looking at our algorithm's coloring result, either the black or white group should all have an edge that points to two of the same colors. The algorithm returns the same result as fact.

Justify running time:

To put things into and out of the queue takes O(1) times |V| in total. For each color check of the neighbor, it takes O(1) work each for 2|E| times (needed for both sides of an edge). Therefore the running time of my algorithm is O(n+m), where n = |V| and m = |E|.

Question (Xiaoxue Xing):

//Verified by Zhenjie Xiong

(a) (i) The vertices is every bike pump stations. Each vertices has its location (x,y)

The edge is the straight bikeway that goes directly between any two bike pump stations

The edge weight is the length between two bike stations. That is √((x1-x2)^2+(y1-y2)^2)

(a) (ii) This question want to find a algorithm that find the route from s to t that the path has minimum maximum edge weight between any two nodes in the path. (For example, one path is 2,3,5 and another path is 4,4,4, we will choose 4,4,4, because 4 is smaller than 5.) This algorithm should be O(n^2logn).

(b) We will use Kruskal's algorithm and have slightly changes in this algorithm. We will stop once s and t are connected (that is both s and t are in the set). Since every time, the edge connected is the smallest weight edge, so once t and s are connected, the largest edge is as smallest as possible.

(c) Firstly, we will rank the edge with its weight from the smallest to the largest. Then, we will union the vertices (put vertices in a set) with the smallest edge, then the vertices with second smallest edge, and so on. If the vertices of that edge are already in the set, we will do noting for the set. When the first time the s and t are both in the set, s and t are connected and the minimum maximum edge is the last edge we use to put s or t in the set.

(d) Because the Kruskal's algorithm running time is O(E logV) V is the number of vertices, which is n. E is the number of edges. The number of edges has at the most n*(n-1). So the time complexity for the Kruskal's algorithm is O(n^2logn). For the s and t, the worst case is that the s or t is the last one added to the set. In this case, It is the same as the Kruskal's algorithm to find the Minimum Spanning Trees. So the worst case running time is O(n^2logn).