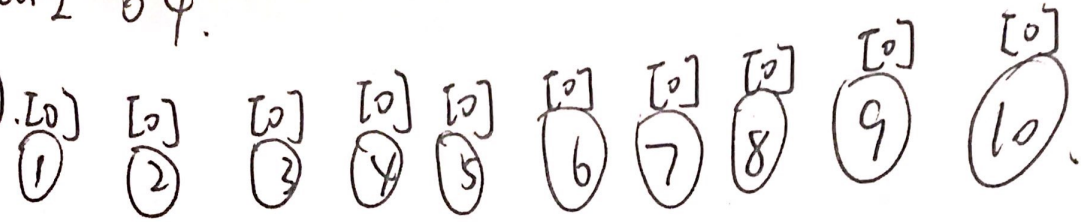
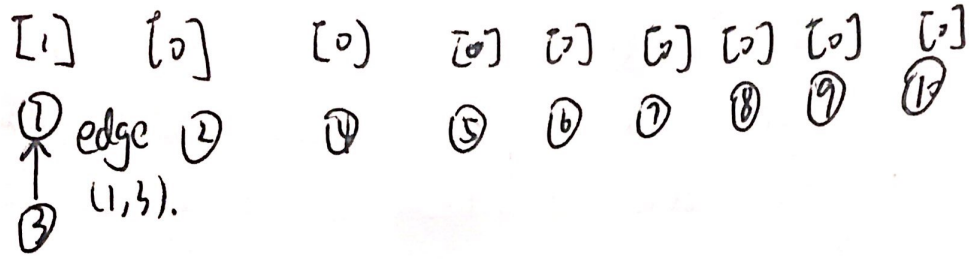
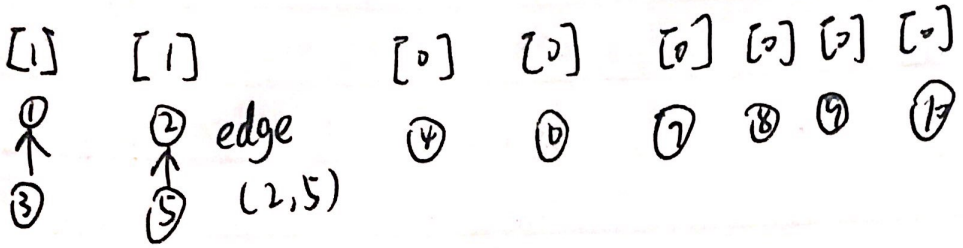
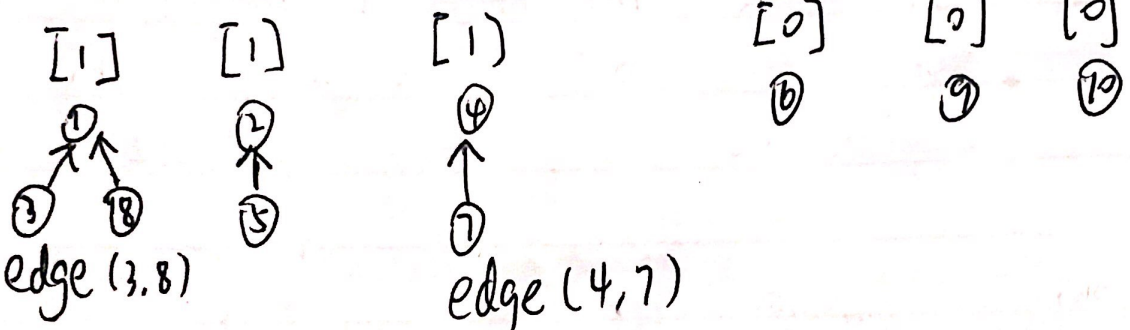
quiz 04.

1.(9).

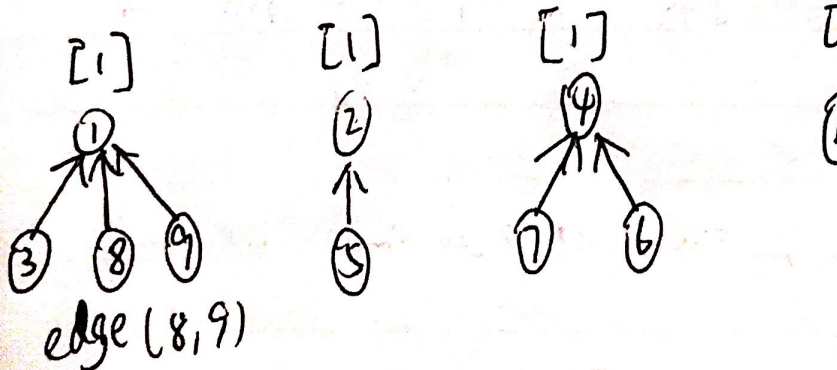

[0] [0] [0] [0] [0] [0] [0] [0] [0] [0]
1 2 3 4 5 6 7 8 9 10

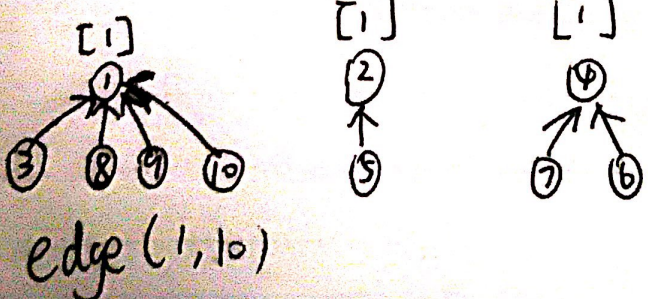

[1] [0] [0] [0] [0] [0] [0] [0] [0]
edge (1,3).


[1] [1] [0] [0] [0] [0] [0] [0]
edge (2,5)


[1] [1] [1] [0] [0] [0]
edge (3,8)
edge (4,7)


[1] [1] [1] [0] [0]
edge (6,7)


[1] [1] [1] [0]
edge (8,9)


[1] [1] [1]
edge (1,10)

(b) For the first node:

When inserting the first node, its color will be black as it is the tree root. And then inserting the second one, its color will be red and no other red nodes.

For the ith node:

Firstly, its color is red. We assume the ith node is x. Thus, there exists 2 situations:

①: x has red uncle node and red father node. So we need to change uncle and father's color to black, and change x's grandfather node to red. Then the grandfather node becomes a new x and the original x is red. ✓

②. x has either a red uncle node and black father node, or a black uncle node and red father node. In this case, x is either left or right child node of its father node. Thus, a left-right rotation can transform one case to another case, and after applied the rotation, x's color will not change.

To sum up, the tree has at least one red node. The claim is true

(c)

When using DFS search, we first start from a node in $s \in V$, when there exists a back edge. Then, there is a cycle in G.

Because there are only $|V|$ nodes in the graph, the back edge doesn't exist, otherwise it will be detected by searching less than $|V|$ edges. If G is disconnected, a node from each connected component is chosen as the starting node to perform DFS search.

Thus, checking whether it exists a cycle takes $O(|V|)$ time, it satisfies with the question.

(d).

We use BFS in G and obtain a BFS tree rooted at the node r. We assume $d(u)$ is the depth of node u in the BFS tree and $d(r)=1$. Then, there exists after we check each non-tree edge $(u,v) \in E$;

① when $d(u) = d(v)$.

In this case, G has an odd cycle.

② when $d(u) \neq d(v)$

In this case, both of them are either even or odd. So G still has an odd cycle.

Otherwise, G will not contain any odd cycles.

For time complexity:

① The construction of the BFS tree takes $O(|V|+|E|)$ time,

② Checking whether there is a non-tree edge $(u,v) \in E$ takes

$O(1)$ time for each non-tree edge and there are no more than $O(|E|)$ non-tree edges.

Thus, sum up above, the algorithm takes $O(|V|+|E|)$ time, which satisfies with the question.

2. (a)

When there exists an edge with negative weight in the directed graph $G(V, E)$, we should use Bellman-ford algorithm instead of Dijkstra's algorithm.

(b)

We assume the edge is with the minimum weight e_{min} is edge (u, v), and T is an MST which contains e_{min}. Then T has a path with different edges from u to v other than edge (u, v).

We now remove e_{min} from T and divide T into 2 disconnected components, one is with u and the other is with v. Then connect these two subset with edge (u,v). Then we have another MST T'.

We assume the cost of T is $w(T) = \sum w(x, y)$ $(x, y \in T)$. Thus, $w(T') = w(T) - w(e_{min}) + w(u, v) \leq w(T)$.

Therefore, e_{min} will be contained by any MST in G because it is the only edge with the minimum weight.

(c) [i]

Since we can only take a route if everyone of its edge has a length less than or equal to L, we can remove edges of a bigger value and check whether city t is still reachable from city s. Thus, a suitable BFS should be able to verify this case, which takes the time $O(|V|+|E|)$.

[ii]

We need to find a path from s to t, such that the largest edge-weight on the path is smallest. To get a such path, we need to modify the Dijkstra's algorithm.

In original Dijkstra's algorithm, for every vertex we have a field d, which is also the key field of the priority queue, Q. We replace the d field as d_2, which is to store the shortest path as mentioned above. In this case, $t.d_2$ describes the length of the shortest path from city s to city t. On initialization, $s.d_2 = 0$ and $t.d_2 = \infty$ for all $s, t \in V$. The detailed algorithm is shown below:

the length.

```
for any edge (s,t) ∈ E:
    t.d2 = ∞
    previous(t) = null.
s.d2 = 0.
if max(t.d2, w(s,t)) < t.d2
    t.d2 = max(t.d2, w(s,t)).
    previous(t) = s.
```

And, the time complexity is $O((|V|+|E|)\log|V|)$.