Problem- LeetCode #899: *Orderly Queue*

You are given a string s and an integer k. You can choose one of the first k letters of s and append it at the end of the string.

Return *the lexicographically smallest string you could have after applying the mentioned step any number of moves*.

Example 1:

Input: s = "cba", k = 1
Output: "acb"
Explanation:
In the first move, we move the 1st character 'c' to the end, obtaining the string "bac".
In the second move, we move the 1st character 'b' to the end, obtaining the final result "acb".

Example 2:

Input: s = "baaca", k = 3
Output: "aaabc"
Explanation:
In the first move, we move the 1st character 'b' to the end, obtaining the string "aacab".
In the second move, we move the 3rd character 'c' to the end, obtaining the final result "aaabc".

Code-

```
class Solution {
public:
    string orderlyQueue(string s, int k) {
        if (k == 1) {

            queue<char> q;

            for (int i = 0; i < s.size(); i++) {
                q.push(s[i]);
```

```
    }

            string smallest = s;

            for (int step = 0; step < s.length(); step++) {
                char first = q.front();
                q.pop();
                q.push(first);

                queue<char> copy= q;
                 string current="";
                 while(!copy.empty()){
                    current+=copy.front();
                    copy.pop();
                 }

                 if(current<smallest){
                    smallest=current;
                 }
            }
            return smallest;

        }else{
            sort(s.begin(),s.end());
            return s;
        }
    }
};
```

Problem- LeetCode #134: *Gas Station*

There are n gas stations along a circular route, where the amount of gas at the i_{th} station is gas[i].

You have a car with an unlimited gas tank and it costs cost[i] of gas to travel from the i_{th} station to its next $(i + 1)_{th}$ station. You begin the journey with an empty tank at one of the gas stations.

Given two integer arrays gas and cost, return *the starting gas station's index if you can travel around the circuit once in the clockwise direction, otherwise return* -1. If there exists a solution, it is **guaranteed** to be **unique**.

Example 1:

Input: gas = [1,2,3,4,5], cost = [3,4,5,1,2]
Output: 3
Explanation:
Start at station 3 (index 3) and fill up with 4 unit of gas. Your tank = 0 + 4 = 4
Travel to station 4. Your tank = 4 - 1 + 5 = 8
Travel to station 0. Your tank = 8 - 2 + 1 = 7
Travel to station 1. Your tank = 7 - 3 + 2 = 6
Travel to station 2. Your tank = 6 - 4 + 3 = 5
Travel to station 3. The cost is 5. Your gas is just enough to travel back to station 3.
Therefore, return 3 as the starting index.

Example 2:

Input: gas = [2,3,4], cost = [3,4,3]
Output: -1
Explanation:
You can't start at station 0 or 1, as there is not enough gas to travel to the next station.
Let's start at station 2 and fill up with 4 unit of gas. Your tank = 0 + 4 = 4
Travel to station 0. Your tank = 4 - 3 + 2 = 3
Travel to station 1. Your tank = 3 - 3 + 3 = 3
You cannot travel back to station 2, as it requires 4 unit of gas but you only have 3.
Therefore, you can't travel around the circuit once no matter where you start.

Code-

```
class Solution {
public:
    int canCompleteCircuit(vector<int>& gas, vector<int>& cost) {

        int total = 0;
        int current = 0;
        int start = 0;

        for (int i = 0;i < gas.size(); i++) {
            int difference = gas[i] - cost[i];
            total += difference;
            current += difference;

            if (current < 0) {
                start = i + 1;
                current = 0;
            }
        }
        if(total<0) return-1;
        return start;
    }
};
```

Problem- LeetCode #994: *Rotting Oranges*

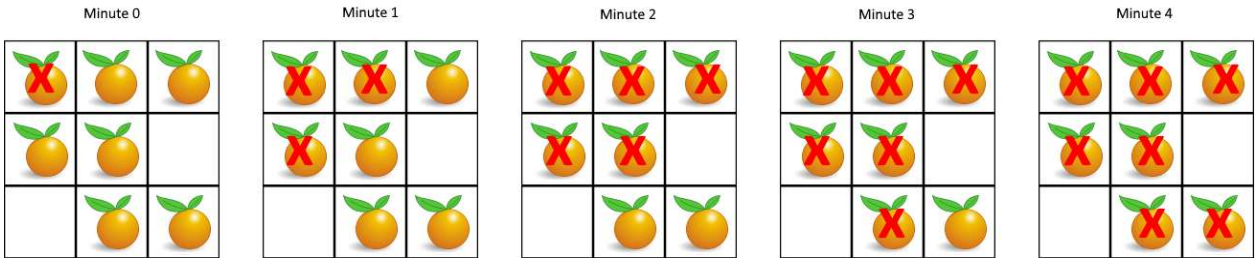
You are given an m x n grid where each cell can have one of three values:

- 0 representing an empty cell,
- 1 representing a fresh orange, or
- 2 representing a rotten orange.

Every minute, any fresh orange that is **4-directionally adjacent** to a rotten orange becomes rotten.

Return *the minimum number of minutes that must elapse until no cell has a fresh orange*. If *this is impossible, return* -1.

Example 1:



Input: grid = [[2,1,1],[1,1,0],[0,1,1]]
Output: 4

Example 2:

Input: grid = [[2,1,1],[0,1,1],[1,0,1]]
Output: -1
Explanation: The orange in the bottom left corner (row 2, column 0) is never rotten, because rotting only happens 4-directionally.

Example 3:

Input: grid = [[0,2]]
Output: 0
Explanation: Since there are already no fresh oranges at minute 0, the answer is just 0.

Code-

```
class Solution {
public:
   int orangesRotting(vector<vector<int>>& grid) {
    int rows = grid.size();
    int cols = grid[0].size();

    queue<pair<int, int>> q; // This queue will store positions of rotten
oranges
    int fresh = 0;           // Count of fresh oranges
    int minutes = 0;         // Time taken to rot all oranges

    // Step 1: Count fresh oranges and add rotten ones to queue
    for (int i = 0; i < rows; i++) {
        for (int j = 0; j < cols; j++) {
            if (grid[i][j] == 2) {
                q.push({i, j}); // Rotten orange found, add to queue
            } else if (grid[i][j] == 1) {
                fresh++; // Fresh orange found, count it
            }
        }
    }

    // These are the directions: -         up,    down,   left,  right
    vector<pair<int, int>> directions = {{-1,0}, {1,0}, {0,-1}, {0,1}};

    // Step 2: Use BFS to spread rot to nearby fresh oranges
    while (!q.empty() && fresh > 0) {
        int size = q.size(); // Number of rotten oranges at this minute

        for (int i = 0; i < size; i++) {
            pair<int, int> current = q.front(); // Using pair<int, int>
explicitly
            q.pop();

            int x = current.first;
            int y = current.second;
```

```
            // Check all 4 directions
            for (int j = 0; j < 4; j++) { // Loop over the directions using
index
                int newX = x + directions[j].first;
                int newY = y + directions[j].second;

                // If the new cell is inside the grid and has a fresh orange
                if (newX >= 0 && newY >= 0 && newX < rows && newY < cols
                    && grid[newX][newY] == 1) {

                    grid[newX][newY] = 2;      // Make the orange rotten
                    q.push({newX, newY});      // Add to queue to process later
                    fresh--;                   // One less fresh orange
                }
            }
        }

        minutes++; // One round of BFS done, so one minute passed
    }

    // If no fresh orange left, return time taken, otherwise return -1
    return (fresh == 0) ? minutes : -1;
}
};
```