STACK AND QUEUE STANDARD QUESTION

(DAY :- 4)

VERY EASY

QUESTION 1:- Design a stack that supports push, pop, top, and retrieving the minimum element in constant time.

Implement the MinStack class:

- MinStack() initializes the stack object.
- void push(int val) pushes the element val onto the stack.
- void pop() removes the element on the top of the stack.
- int top() gets the top element of the stack.
- int getMin() retrieves the minimum element in the stack.

You must implement a solution with O(1) time complexity for each function.

Example 1:

Input

["MinStack","push","push","push","getMin","pop","top","getMin"]

[[],[-2],[0],[-3],[],[],[],[]]

Output

[null,null,null,null,-3,null,0,-2]

Explanation

MinStack minStack = new MinStack();

minStack.push(-2);

minStack.push(0);

minStack.push(-3);

minStack.getMin(); // return -3

minStack.pop();

minStack.top();   // return 0

minStack.getMin(); // return -2

Example 2:

Input:

["MinStack", "push", "push", "push", "push", "getMin", "pop", "getMin", "top", "getMin"]

[[], [5], [3], [7], [3], [], [], [], [], []]

Output

[null, null, null, null, null, 3, null, 3, 7, 3]

Explanation:

MinStack minStack = new MinStack();

minStack.push(5);   # Stack: [5], MinStack: [5]

minStack.push(3);   # Stack: [5, 3], MinStack: [5, 3]

minStack.push(7);   # Stack: [5, 3, 7], MinStack: [5, 3]

minStack.push(3);   # Stack: [5, 3, 7, 3], MinStack: [5, 3, 3]

minStack.getMin();   # Returns 3

minStack.pop();   # Removes 3; Stack: [5, 3, 7], MinStack: [5, 3]

minStack.getMin();   # Returns 3

minStack.top();   # Returns 7

minStack.getMin();   # Returns 3

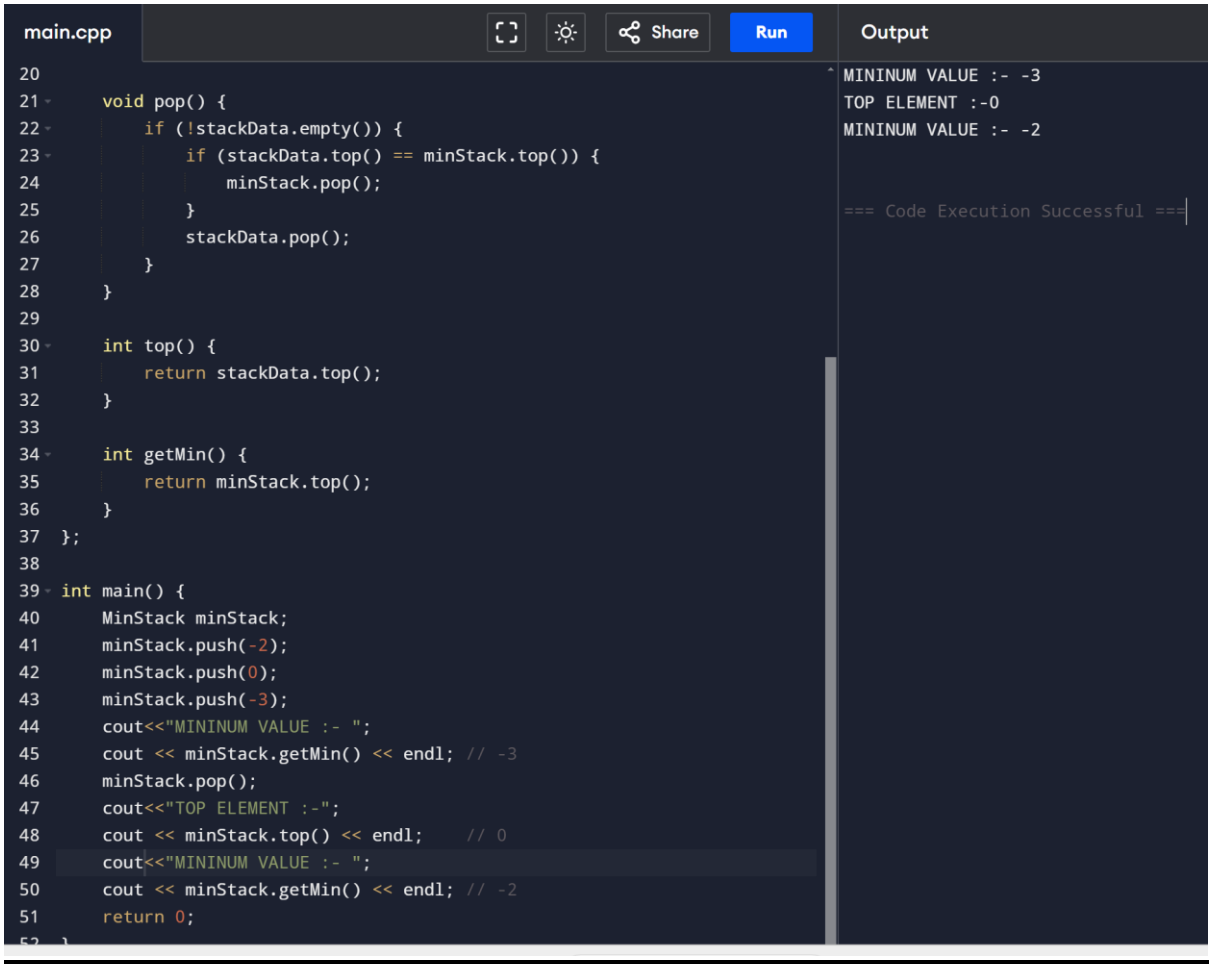
- Minimum values are maintained as: [5] → [5, 3] → [5, 3] → [5, 3]
- After pops, the minimum values update accordingly.

CODE:-

```
#include <stack>

#include <iostream>

using namespace std;
```

```
class MinStack {
private:
    stack<int> stackData;
    stack<int> minStack;
public:
    MinStack() {}
    void push(int val) {
        stackData.push(val);
        if (minStack.empty() || val <= minStack.top()) {
            minStack.push(val);
        }
    }
    void pop() {
        if (!stackData.empty()) {
            if (stackData.top() == minStack.top()) {
                minStack.pop();
            }
            stackData.pop();
        }
    }
    int top() {
        return stackData.top();
    }
    int getMin() {
        return minStack.top();   }};
```

```
int main() {
    MinStack minStack;
    minStack.push(-2);
    minStack.push(0);
    minStack.push(-3);
    cout << minStack.getMin() << endl;
    minStack.pop();
    cout << minStack.top() << endl;
    cout << minStack.getMin() << endl;
    return 0;
}
```



QUESTION 2:- Given a string s, find the first non-repeating character in it and return its index. If it does not exist, return -1.

Example 1:

Input: s = "leetcode"

Output: 0

Explanation:

The character 'l' at index 0 is the first character that does not occur at any other index.

Example 2:

Input: s = "loveleetcode"

Output: 2

Example 3:

Input: s = "aabb"

Output: -1

Constraints:

- 1 <= s.length <= 105
- s consists of only lowercase English letters.

Approach:

- Use a hash map or array of size 26 to store the frequency of each character (since the input consists of lowercase English letters).
- Traverse the string to count the frequency of each character.
- Traverse the string again to find the first character with a frequency of 1. Return its index.
- If no such character is found, return -1.
- **Time Complexity:** (O(n)), where (n) is the length of the string. The string is traversed twice.
- **Space Complexity:** (O(1)), as the frequency array has a fixed size of 26.

CODE:-

```
#include <iostream>

#include <string>

#include <vector>
```

```
using namespace std;
int firstUniqChar(string s) {
    vector<int> freq(26, 0);
    for (char c : s) {
        freq[c - 'a']++;    }
    for (int i = 0; i < s.length(); i++) {
        if (freq[s[i] - 'a'] == 1) {
            return i;   }    }
    return -1; }
int main() {
   string s1 = "leetcode";
    cout << firstUniqChar(s1) << endl;
    string s2 = "loveleetcode";
    cout << firstUniqChar(s2) << endl;
    string s3 = "aabb";
    cout << firstUniqChar(s3) << endl;
    return 0;}
```

<u>QUESTION 3:-</u> You are given an array of strings tokens that represents an arithmetic expression in a Reverse Polish Notation.

Evaluate the expression. Return an integer that represents the value of the expression.

Note that:

- The valid operators are '+', '-', '*', and '/'.
- Each operand may be an integer or another expression.
- The division between two integers always truncates toward zero.
- There will not be any division by zero.
- The input represents a valid arithmetic expression in a reverse polish notation.
- The answer and all the intermediate calculations can be represented in a 32-bit integer.

Example 1:

Input: tokens = ["2","1","+","3","*"]

Output: 9

Explanation: ((2 + 1) * 3) = 9

Example 2:

Input: tokens = ["4","13","5","/","+"]

Output: 6

Explanation: (4 + (13 / 5)) = 6

Example 3:

Input: tokens = ["10","6","9","3","+","-11","*","/","*","17","+","5","+"]

Output: 22

Explanation: ((10 * (6 / ((9 + 3) * -11))) + 17) + 5

= ((10 * (6 / (12 * -11))) + 17) + 5

= ((10 * (6 / -132)) + 17) + 5

= ((10 * 0) + 17) + 5

= (0 + 17) + 5

= 17 + 5

= 22

Constraints:

- 1 <= tokens.length <= 104
- tokens[i] is either an operator: "+", "-", "*", or "/", or an integer in the range [-200, 200].

Approach

Using Fundamentals of STACK && LAMBDA...

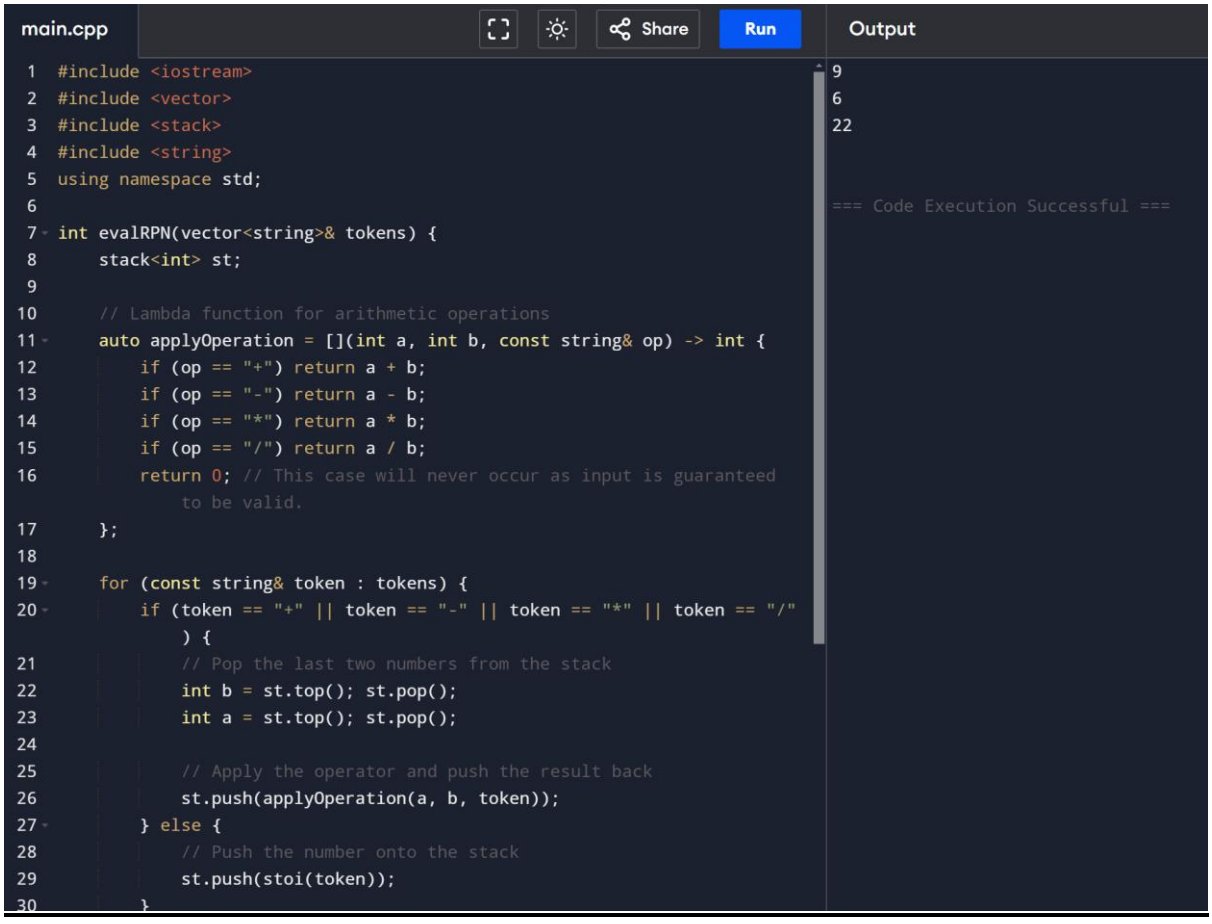
Time complexity: O(n)

Space complexity: O(n)

CODE:-

```
#include <iostream>
#include <vector>
#include <stack>
#include <string>
using namespace std;
int evalRPN(vector<string>& tokens) {
    stack<int> st;
    auto applyOperation = [](int a, int b, const string& op) -> int {
        if (op == "+") return a + b;
        if (op == "-") return a - b;
        if (op == "*") return a * b;
        if (op == "/") return a / b;
        return 0;    };
    for (const string& token : tokens) {
        if (token == "+" || token == "-" || token == "*" || token == "/") {
            int b = st.top(); st.pop();
            int a = st.top(); st.pop();
            st.push(applyOperation(a, b, token));
        } else {
            st.push(stoi(token));
```

```
    } }
    return st.top();
}
int main()
 {
    vector<string> tokens1 = {"2", "1", "+", "3", "*"};
    cout << evalRPN(tokens1) << endl; // Output: 9
    vector<string> tokens2 = {"4", "13", "5", "/", "+"};
    cout << evalRPN(tokens2) << endl; // Output: 6
    vector<string> tokens3 = {"10", "6", "9", "3", "+", "-11", "*", "/", "*", "17", "+", "5", "+"};
    cout << evalRPN(tokens3) << endl; // Output: 22
    return 0;
}
```



EASY LEVEL

QUESTION 1:- The school cafeteria offers circular and square sandwiches at lunch break, referred to by numbers 0 and 1 respectively. All students stand in a queue. Each student either prefers square or circular sandwiches.The number of sandwiches in the cafeteria is equal to the number of students. The sandwiches are placed in a stack. At each step:

If the student at the front of the queue prefers the sandwich on the top of the stack, they will take it and leave the queue.
Otherwise, they will leave it and go to the queue's end.
This continues until none of the queue students want to take the top sandwich and are thus unable to eat.
You are given two integer arrays students and sandwiches where sandwiches[i] is the type of the ith sandwich in the stack (i = 0 is the top of the stack) and students[j] is the preference of the jth student in the initial queue (j = 0 is the front of the queue). Return the number of students that are unable to eat.

Example 1:

Input: students = [1,1,0,0], sandwiches = [0,1,0,1]
Output: 0
Explanation:
- Front student leaves the top sandwich and returns to the end of the line making students = [1,0,0,1].
- Front student leaves the top sandwich and returns to the end of the line making students = [0,0,1,1].
- Front student takes the top sandwich and leaves the line making students = [0,1,1] and sandwiches = [1,0,1].
- Front student leaves the top sandwich and returns to the end of the line making students = [1,1,0].
- Front student takes the top sandwich and leaves the line making students = [1,0] and sandwiches = [0,1].
- Front student leaves the top sandwich and returns to the end of the line making students = [0,1].
- Front student takes the top sandwich and leaves the line making students = [1] and sandwiches = [1].
- Front student takes the top sandwich and leaves the line making students = [] and sandwiches = [].
Hence all students are able to eat.
Example 2:
Input: students = [1,1,1,0,0,1], sandwiches = [1,0,0,0,1,1]
Output: 3
Constraints:
- 1 <= students.length, sandwiches.length <= 100
- students.length == sandwiches.length
- sandwiches[i] is 0 or 1.
- students[i] is 0 or 1.

Approach

- Create two queues of students and sandwiches
- And a count variable to check if is loop in left student
- If students in the queue cannot have their ordered sandwiches, it makes a loop. If it is a loop, just break and return the result
- Then implement the program like the given rules.
  Time complexity: O(n)
  Space complexity: O(n)

CODE:-

```
#include <iostream>
#include <queue>
#include <vector>
using namespace std;

int countStudents(vector<int>& students, vector<int>& sandwiches) {
   queue<int> studentQueue;
   queue<int> sandwichQueue;

   for (int s : students) studentQueue.push(s);
   for (int s : sandwiches) sandwichQueue.push(s);

   int count = 0;
   while (!studentQueue.empty() && count < studentQueue.size()) {
      if (studentQueue.front() == sandwichQueue.front()) {
         // Student takes the sandwich
         studentQueue.pop();
         sandwichQueue.pop();
         count = 0;
      } else {
         studentQueue.push(studentQueue.front());
         studentQueue.pop();
         count++;
      }
   }
   return studentQueue.size(); }
 int main() {
   vector<int> students1 = {1, 1, 0, 0};
   vector<int> sandwiches1 = {0, 1, 0, 1};
   cout << countStudents(students1, sandwiches1) << endl;

   vector<int> students2 = {1, 1, 1, 0, 0, 1};
   vector<int> sandwiches2 = {1, 0, 0, 0, 1, 1};
   cout << countStudents(students2, sandwiches2) << endl;

   return 0; }
```

main.cpp

Share Run

Output

```cpp
#include <iostream>
#include <queue>
#include <vector>
using namespace std;

int countStudents(vector<int>& students, vector<int>& sandwiches) {
    queue<int> studentQueue;
    queue<int> sandwichQueue;

    // Populate the queues
    for (int s : students) studentQueue.push(s);
    for (int s : sandwiches) sandwichQueue.push(s);

    int count = 0; // Counter to detect a loop

    while (!studentQueue.empty() && count < studentQueue.size()) {
        if (studentQueue.front() == sandwichQueue.front()) {
            // Student takes the sandwich
            studentQueue.pop();
            sandwichQueue.pop();
            count = 0; // Reset the loop counter
        } else {
            // Student goes to the end of the queue
            studentQueue.push(studentQueue.front());
            studentQueue.pop();
            count++;
        }
    }

    // Remaining students in the queue cannot eat
    return studentQueue.size();
}
```

```
0
3


=== Code Execution Successful ===
```

<u>QUESTION 2:-</u> We are given an array asteroids of integers representing asteroids in a row.For each asteroid, the absolute value represents its size, and the sign represents its direction (positive meaning right, negative meaning left). Each asteroid moves at the same speed.

Find out the state of the asteroids after all collisions. If two asteroids meet, the smaller one will explode. If both are the same size, both will explode. Two asteroids moving in the same direction will never meet.

Example 1:

Input: asteroids = [5,10,-5]

Output: [5,10]

Explanation: The 10 and -5 collide resulting in 10. The 5 and 10 never collide.

Example 2:

Input: asteroids = [8,-8]

Output: []

Explanation: The 8 and -8 collide exploding each other.

Example 3:

Input: asteroids = [10,2,-5]

Output: [10]

Explanation: The 2 and -5 collide resulting in -5. The 10 and -5 collide resulting in 10.

Constraints:

- 2 <= asteroids.length <= 104
- -1000 <= asteroids[i] <= 1000
- asteroids[i] != 0

Complexity

- **Time complexity:** O(N), since each asteroid is processed at most once.
- **Space complexity:** O(N), to account for the stack storage in the worst case where no collisions occur.

CODE:-

```
#include <vector>
#include <stack>
#include <iostream>
using namespace std;


vector<int> asteroidCollision(vector<int>& asteroids) {
    stack<int> st;
   for (int asteroid : asteroids) {
      while (!st.empty() && asteroid < 0 && st.top() > 0) {
          if (st.top() < -asteroid) {
              st.pop();
              continue;
          }
          else if (st.top() == -asteroid) {
              st.pop();
          }
          break;        }
```

```
        if (asteroid > 0 || st.empty() || st.top() < 0) {
            st.push(asteroid);
        }    }
    vector<int> result;
    while (!st.empty()) {
        result.push_back(st.top());
        st.pop();
    }
    reverse(result.begin(), result.end());
    return result;
}
int main() {
    vector<int> asteroids1 = {5, 10, -5};
    vector<int> result1 = asteroidCollision(asteroids1);
    for (int asteroid : result1) {
    cout << asteroid << " ";    }
    cout << endl;  // Output: 5 10
    vector<int> asteroids2 = {8, -8};
    vector<int> result2 = asteroidCollision(asteroids2);
    for (int asteroid : result2) {
    cout << asteroid << " ";    }
    cout << endl;  // Output: (empty)
    vector<int> asteroids3 = {10, 2, -5};
    vector<int> result3 = asteroidCollision(asteroids3);
    for (int asteroid : result3) {
    cout << asteroid << " ";    }
    cout << endl;  // Output: 10
    return 0;
}
```

main.cpp Share Run Output

```cpp
#include <vector>
#include <stack>
#include <iostream>
#include<algorithm>
using namespace std;

vector<int> asteroidCollision(vector<int>& asteroids) {
    stack<int> st;
    for (int asteroid : asteroids) {

        while (!st.empty() && asteroid < 0 && st.top() > 0) {

            if (st.top() < -asteroid) {
                st.pop();
                continue;
            }

            else if (st.top() == -asteroid) {
                st.pop();
            }

            break;
        }

        if (asteroid > 0 || st.empty() || st.top() < 0) {
            st.push(asteroid);
        }
    }
```

```
5 10
-8
10


=== Code Execution Success
```

<u>QUESTION 3:-</u> Given an integer array nums, handle multiple queries of the following type:
Calculate the sum of the elements of nums between indices left and right inclusive where left <= right.
Implement the NumArray class:
NumArray(int[] nums) Initializes the object with the integer array nums.
int sumRange(int left, int right) Returns the sum of the elements of nums between indices left and right inclusive (i.e. nums[left] + nums[left + 1] + ... + nums[right]).

Example 1:
Input :["NumArray", "sumRange", "sumRange", "sumRange"]
[[[-2, 0, 3, -5, 2, -1]], [0, 2], [2, 5], [0, 5]]
Output:[null, 1, -1, -3]

Explanation
NumArray numArray = new NumArray([-2, 0, 3, -5, 2, -1]);
numArray.sumRange(0, 2); // return (-2) + 0 + 3 = 1
numArray.sumRange(2, 5); // return 3 + (-5) + 2 + (-1) = -1
numArray.sumRange(0, 5); // return (-2) + 0 + 3 + (-5) + 2 + (-1) = -3

Constraints:

- 1 <= nums.length <= 104
- -105 <= nums[i] <= 105
- 0 <= left <= right < nums.length
- At most 104 calls will be made to sumRange.

Approach

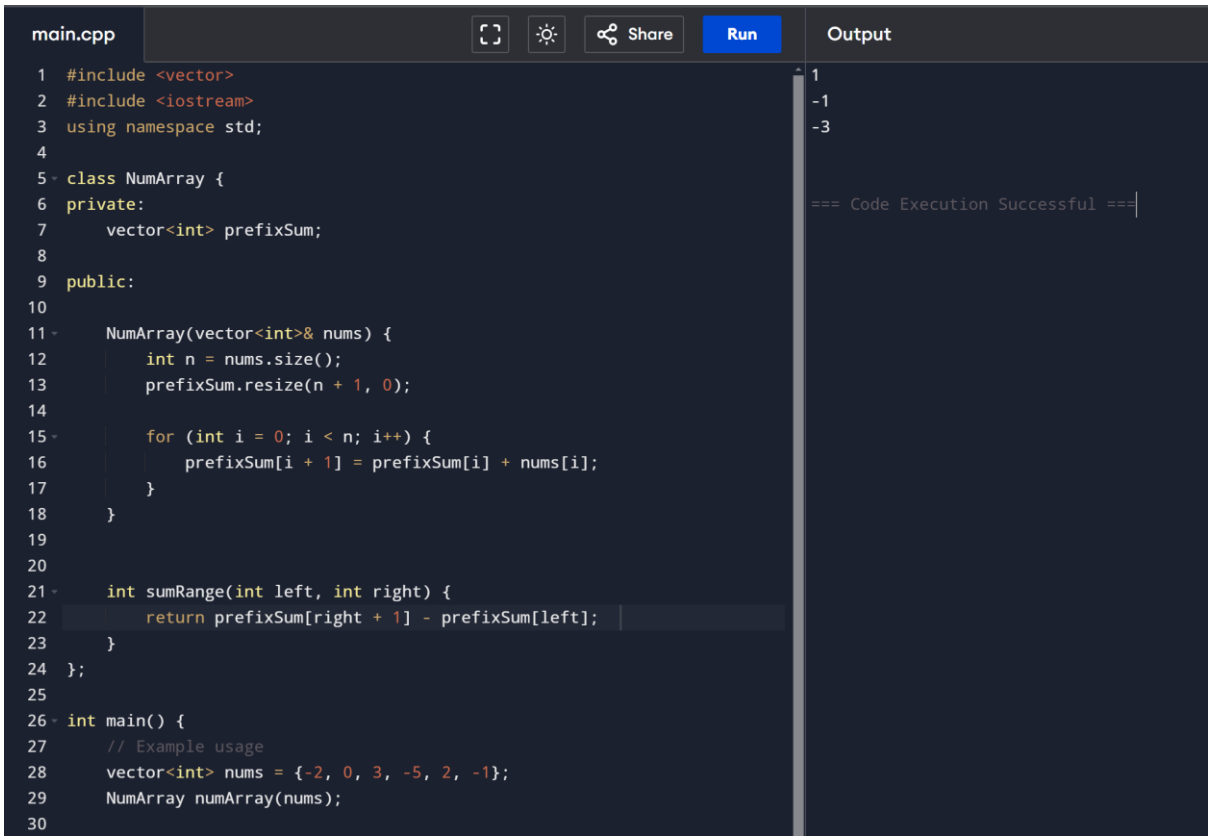
Let's Say Array Is : [-2, 0, 3, -5, 2, -1]
I Create Prefix Sum Array A: [0, -2, -2, 1, -4, -2, -3]
How ?
Prefix Sum For Index i : Sum Of All Elements Before i

So For Sum from l to r : A[r+1] - A[l]
A[r+1] Sum of All Element from `0 To r`
A[l] Sum of All Element from `0 To l-1`

CODE:-

```
#include <vector>
#include <iostream>
using namespace std;


class NumArray {
private:
    vector<int> prefixSum;
public:
   NumArray(vector<int>& nums) {
        int n = nums.size();
        prefixSum.resize(n + 1, 0);
        for (int i = 0; i < n; i++) {
            prefixSum[i + 1] = prefixSum[i] + nums[i];
        }
    }
    int sumRange(int left, int right) {
        return prefixSum[right + 1] - prefixSum[left];
   }
};
```

```
int main() {
      vector<int> nums = {-2, 0, 3, -5, 2, -1};
    NumArray numArray(nums);
    cout << numArray.sumRange(0, 2) << endl;
    cout << numArray.sumRange(2, 5) << endl;
    cout << numArray.sumRange(0, 5) << endl;
    return 0;
}
```



MEDIUM LEVEL:-

QUESTION 1:- Given a circular integer array nums (i.e., the next element of nums[nums.length - 1] is nums[0]), return the next greater number for every element in nums.The next greater number of a number x is the first greater number to its traversing-order next in the array, which means you could search circularly to find its next greater number. If it doesn't exist, return -1 for this number.

Example 1:

Input: nums = [1,2,1]

Output: [2,-1,2]

Explanation:

- The first 1's next greater number is 2;
- The number 2 can't find next greater number.
- The second 1's next greater number needs to search circularly, which is also 2.

Example 2:

Input: nums = [1,2,3,4,3]

Output: [2,3,4,-1,4]

Constraints:

- 1 <= nums.length <= 104
- -109 <= nums[i] <= 109

CODE:-

```
#include <vector>

#include <stack>

#include <iostream>

using namespace std;

vector<int> nextGreaterElements(vector<int>& nums)

{

    int n = nums.size();

    vector<int> result(n, -1);

    stack<int> st;
```

```
    for (int i = 0; i < 2 * n; ++i)
  {
        int num = nums[i % n];
        while (!st.empty() && nums[st.top()] < num) {
            result[st.top()] = num;
            st.pop();      }
        if (i < n) {
            st.push(i);   }    }
        return result;
  }
  int main() {
  vector<int> nums1 = {1, 2, 1};
  vector<int> result1 = nextGreaterElements(nums1);
  for (int val : result1) {
  cout << val << " ";    }
  cout << endl;  // Output: 2 -1 2
  vector<int> nums2 = {1, 2, 3, 4, 3};
  vector<int> result2 = nextGreaterElements(nums2);
  for (int val : result2) {
  cout << val << " ";    }
  cout << endl;  // Output: 2 3 4 -1 4
  return 0;
}
```

main.cpp

Share Run Output

```cpp
#include <vector>
#include <stack>
#include <iostream>
using namespace std;

vector<int> nextGreaterElements(vector<int>& nums) {
    int n = nums.size();
    vector<int> result(n, -1);
    stack<int> st;
    for (int i = 0; i < 2 * n; ++i) {
        int num = nums[i % n];
        while (!st.empty() && nums[st.top()] < num) {
            result[st.top()] = num;
            st.pop();
        }

        if (i < n) {
            st.push(i);
        }
    }

    return result;
}
```

```
2 -1 2
2 3 4 -1 4


=== Code Execution Successful ===
```

<u>QUESTION 2:-</u> Given a balanced parentheses string s, return the score of the string.

The score of a balanced parentheses string is based on the following rule:

"()" has score 1.

AB has score A + B, where A and B are balanced parentheses strings.

(A) has score 2 * A, where A is a balanced parentheses string.

Example 1:

Input: s = "()"

Output: 1

Example 2:

Input: s = "(())"

Output: 2

Example 3:

Input: s = "()()"

Output: 2

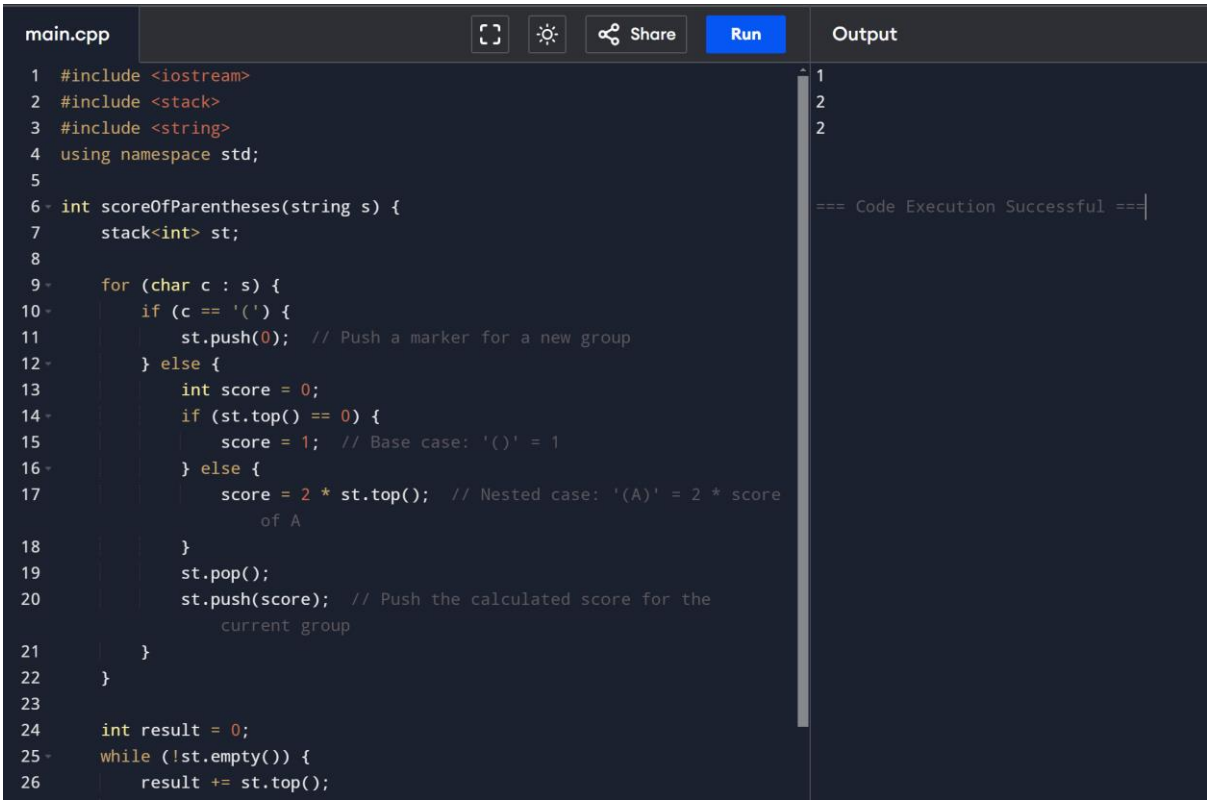
Constraints:

- 2 <= s.length <= 50
- s consists of only '(' and ')'.
- s is a balanced parentheses string.

CODE:-

```
#include <iostream>

#include <stack>

#include <string>

using namespace std;

int scoreOfParentheses(string s) {

stack<int> st;

for (char c : s) {

   if (c == '(') {

      st.push(0);   }

   else {

      int score = 0;

      if (st.top() == 0) {

         score = 1; }

      else {

          score = 2 * st.top();  }

          st.pop();

          st.push(score);

       }    }

    int result = 0;

   while (!st.empty()) {

       result += st.top();

       st.pop();     }
```

```
    return result; }

int main() {

string s1 = "()";

cout << scoreOfParentheses(s1) << endl;

string s2 = "(())";

cout << scoreOfParentheses(s2) << endl;

string s3 = "()()";

cout << scoreOfParentheses(s3) << endl;

return 0;

}
```



QUESTION 3:- You are given a 0-indexed string pattern of length n consisting of the characters 'I' meaning increasing and 'D' meaning decreasing.A 0-indexed string num of length n + 1 is created using the following conditions:

- **num consists of the digits '1' to '9', where each digit is used at most once.**

- **If pattern[i] == 'I', then num[i] < num[i + 1].**
- **If pattern[i] == 'D', then num[i] > num[i + 1].**
- **Return the lexicographically smallest possible string num that meets the conditions.**

Example 1:

Input: pattern = "IIIDIDDD"

Output: "123549876"

Explanation:

- At indices 0, 1, 2, and 4 we must have that num[i] < num[i+1].
- At indices 3, 5, 6, and 7 we must have that num[i] > num[i+1].
- Some possible values of num are "245639871", "135749862", and "123849765".
- It can be proven that "123549876" is the smallest possible num that meets the conditions.
- Note that "123414321" is not possible because the digit '1' is used more than once.

Example 2:

Input: pattern = "DDD"

Output: "4321"

Explanation:

- Some possible values of num are "9876", "7321", and "8742".
- It can be proven that "4321" is the smallest possible num that meets the conditions.

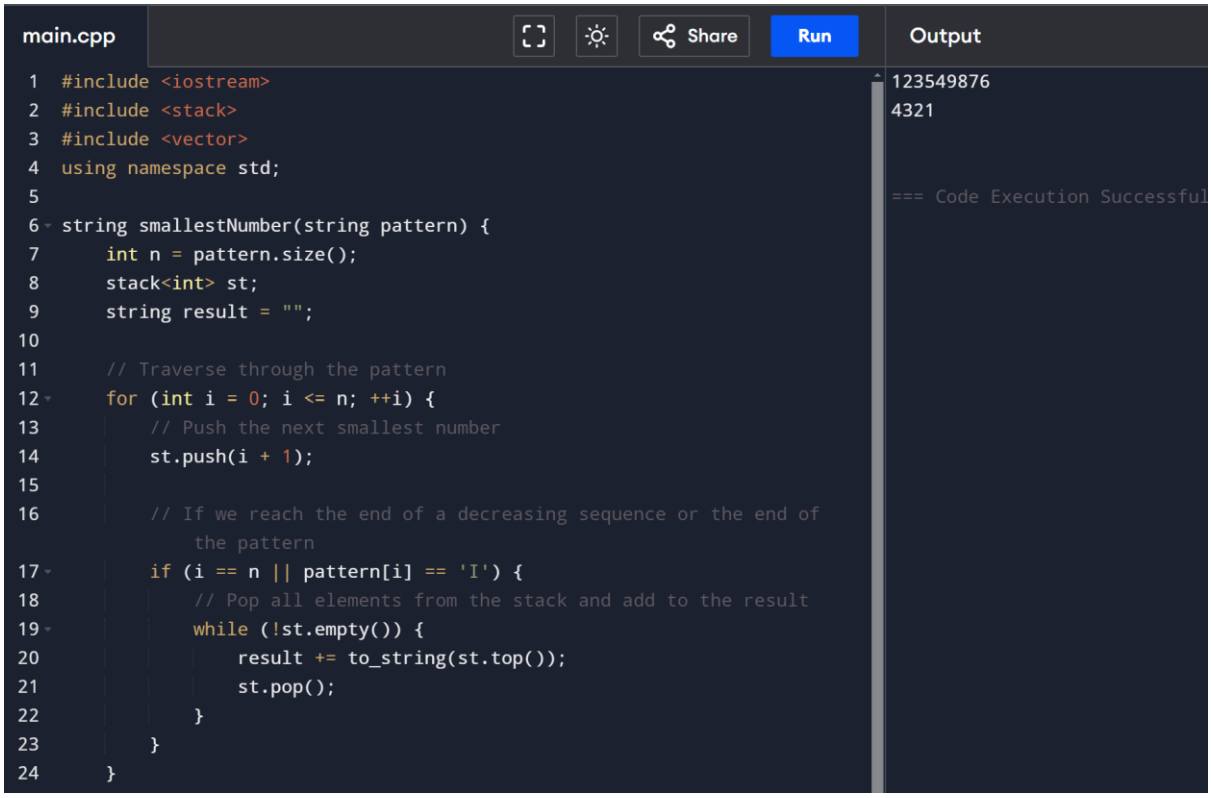
Constraints:

- 1 <= pattern.length <= 8
- pattern consists of only the letters 'I' and 'D'.

CODE:-

```
#include <iostream>

#include <stack>

#include <vector>

using namespace std;

string smallestNumber(string pattern) {

    int n = pattern.size();

    stack<int> st;
```

```
    string result = "";
    for (int i = 0; i <= n; ++i) {
        st.push(i + 1);
        if (i == n || pattern[i] == 'I') {
            while (!st.empty()) {
                result += to_string(st.top());
                st.pop(); } } }
     return result; }
int main() {
    string pattern1 = "IIIDIDDD";
    cout << smallestNumber(pattern1) << endl;  // Output: "123549876"
    string pattern2 = "DDD";
    cout << smallestNumber(pattern2) << endl;  // Output: "4321"
    return 0; }
```



HARD LEVEL

QUESTION 1:- You are given an array of integers nums, there is a sliding window of size k which is moving from the very left of the array to the very right. You can only see the k numbers in the window. Each time the sliding window moves right by one position. Return the max sliding window.

Example 1:

Input: nums = [1,3,-1,-3,5,3,6,7], k = 3

Output: [3,3,5,5,6,7]

Explanation:

```
Window position                Max
---------------               -----
[1  3  -1] -3  5  3  6  7       3
 1 [3  -1  -3] 5  3  6  7       3
 1  3 [-1  -3  5] 3  6  7       5
 1  3  -1 [-3  5  3] 6  7       5
 1  3  -1  -3 [5  3  6] 7       6
 1  3  -1  -3  5 [3  6  7]      7
```

Example 2:

Input: nums = [1], k = 1

Output: [1]

Constraints:

- 1 <= nums.length <= 105
- -104 <= nums[i] <= 104
- 1 <= k <= nums.length

CODE:-

```
#include <iostream>

#include <deque>

#include <vector>

using namespace std;

vector<int> maxSlidingWindow(vector<int>& nums, int k) {

    vector<int> result;

    deque<int> dq;

    for (int i = 0; i < nums.size(); i++) {

        if (!dq.empty() && dq.front() < i - k + 1) {

            dq.pop_front();  }
```

```
        while (!dq.empty() && nums[dq.back()] < nums[i]) {
         dq.pop_back(); }
         dq.push_back(i);
        if (i >= k - 1) {
         result.push_back(nums[dq.front()]);
        }
  }
 return result; }
 int main() {
    vector<int> nums1 = {1, 3, -1, -3, 5, 3, 6, 7};
    int k1 = 3;
    vector<int> result1 = maxSlidingWindow(nums1, k1);
    for (int val : result1) {
        cout << val << " ";
}
    cout << endl;
    vector<int> nums2 = {1};
    int k2 = 1;
    vector<int> result2 = maxSlidingWindow(nums2, k2);
    for (int val : result2) {
        cout << val << " ";
 }
    cout << endl;
    return 0;
 }
```

main.cpp

Share Run

```cpp
#include <iostream>
#include <deque>
#include <vector>
using namespace std;

vector<int> maxSlidingWindow(vector<int>& nums, int k) {
    vector<int> result;
    deque<int> dq;  // This will store indices of nums

    for (int i = 0; i < nums.size(); i++) {
        // Remove elements out of the current window
        if (!dq.empty() && dq.front() < i - k + 1) {
            dq.pop_front();
        }

        // Remove all elements smaller than the current element from the
            deque
        while (!dq.empty() && nums[dq.back()] < nums[i]) {
            dq.pop_back();
        }

        // Add the current element at the back of the deque
        dq.push_back(i);

```

Output

```
3 3 5 5 6 7
1


=== Code Execution Succes
```

<u>QUESTION 2:-</u> Suppose there is a circle. There are N petrol pumps on that circle. Petrol pumps are numbered 0 to (N-1) (both inclusive). You have two pieces of information corresponding to each of the petrol pump: (1) the amount of petrol that particular petrol pump will give, and (2) the distance from that petrol pump to the next petrol pump.

Initially, you have a tank of infinite capacity carrying no petrol. You can start the tour at any of the petrol pumps. Calculate the first point from where the truck will be able to complete the circle. Consider that the truck will stop at each of the petrol pumps. The truck will move one kilometer for each litre of the petrol.

Input Format

The first line will contain the value of N.
The next N lines will contain a pair of integers each, i.e. the amount of petrol that petrol pump will give and the distance between that petrol pump and the next petrol pump.

Constraints:

$1 <= N <= 10^5$

$1 <=$ amount of petrol,distance $<= 10^9$

Output Format

An integer which will be the smallest index of the petrol pump from which we can start the tour.

Example 1:

Sample Input

3
1 5
10 3
3 4

Sample Output

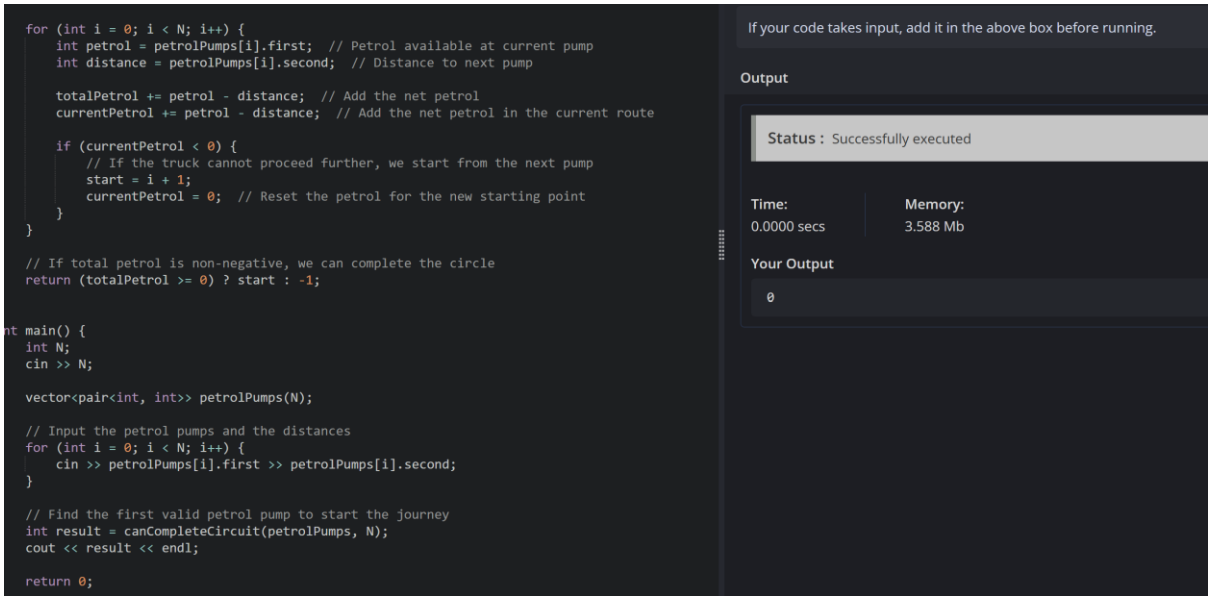
1

Explanation

We can start the tour from the second petrol pump.

CODE:-

```
#include <iostream>
#include <vector>
using namespace std;
int canCompleteCircuit(vector<pair<int, int>>& petrolPumps, int N) {
    int totalPetrol = 0, currentPetrol = 0;
    int start = 0;
    for (int i = 0; i < N; i++) {
        int petrol = petrolPumps[i].first;
        int distance = petrolPumps[i].second;

        totalPetrol += petrol - distance;
        currentPetrol += petrol - distance;
        if (currentPetrol < 0) {
            start = i + 1;
            currentPetrol = 0;
        }
    }
    return (totalPetrol >= 0) ? start : -1;
}
int main() {
    int N;
    cin >> N;
    vector<pair<int, int>> petrolPumps(N);
    for (int i = 0; i < N; i++)
   {
        cin >> petrolPumps[i].first >> petrolPumps[i].second;
    }
    int result = canCompleteCircuit(petrolPumps, N);
    cout << result << endl;
    return 0;
}
```



```
    for (int i = 0; i < N; i++) {
        int petrol = petrolPumps[i].first;  // Petrol available at current pump
        int distance = petrolPumps[i].second;  // Distance to next pump

        totalPetrol += petrol - distance;  // Add the net petrol
        currentPetrol += petrol - distance;  // Add the net petrol in the current route

        if (currentPetrol < 0) {
            // If the truck cannot proceed further, we start from the next pump
            start = i + 1;
            currentPetrol = 0;  // Reset the petrol for the new starting point
        }
    }

    // If total petrol is non-negative, we can complete the circle
    return (totalPetrol >= 0) ? start : -1;


nt main() {
    int N;
    cin >> N;

    vector<pair<int, int>> petrolPumps(N);

    // Input the petrol pumps and the distances
    for (int i = 0; i < N; i++) {
        cin >> petrolPumps[i].first >> petrolPumps[i].second;
    }

    // Find the first valid petrol pump to start the journey
    int result = canCompleteCircuit(petrolPumps, N);
    cout << result << endl;

    return 0;
```

If your code takes input, add it in the above box before running.

Output

Status : Successfully executed

Time: 0.0000 secs

Memory: 3.588 Mb

Your Output

0

<u>QUESTION 3:-</u> You are playing a variation of the game Zuma.

In this variation of Zuma, there is a single row of colored balls on a board, where each ball can be colored red 'R', yellow 'Y', blue 'B', green 'G', or white 'W'. You also have several colored balls in your hand.Your goal is to clear all of the balls from the board. On each turn:Pick any ball from your hand and insert it in between two balls in the row or on either end of the row.If there is a group of three or more consecutive balls of the same color, remove the group of balls from the board.If this removal causes more groups of three or more of the same color to form, then continue removing each group until there are none left.If there are no more balls on the board, then you win the game. Repeat this process until you either win or do not have any more balls in your hand.Given a string board, representing the row of balls on the board, and a string hand, representing the balls in your hand, return the minimum number of balls you have to insert to clear all the balls from the board. If you cannot clear all the balls from the board using the balls in your hand, return -1.

Example 1:

Input: board = "WRRBBW", hand = "RB"

Output: -1

Explanation: It is impossible to clear all the balls. The best you can do is:

- Insert 'R' so the board becomes WRRRBBW. WRRRBBW -> WBBW.

- Insert 'B' so the board becomes WBBBW. WBBBW -> WW.

There are still balls remaining on the board, and you are out of balls to insert.

Example 2:

Input: board = "WWRRBBWW", hand = "WRBRW"

Output: 2

Explanation: To make the board empty:

- Insert 'R' so the board becomes WWRRRBBWW. WWRRRBBWW -> WWBBWW.

- Insert 'B' so the board becomes WWBBBWW. WWBBBWW -> WWWW -> empty.

2 balls from your hand were needed to clear the board.

Example 3:

Input: board = "G", hand = "GGGGG"

Output: 2

Explanation: To make the board empty:

- Insert 'G' so the board becomes GG.

- Insert 'G' so the board becomes GGG. GGG -> empty.

2 balls from your hand were needed to clear the board.

Constraints:

- 1 <= board.length <= 16
- 1 <= hand.length <= 5
- board and hand consist of the characters 'R', 'Y', 'B', 'G', and 'W'.
- The initial row of balls on the board will not have any groups of three or m

CODE:-

```
#include <iostream>

#include <string>

#include <climits>

using namespace std;

class ZumaGame {

public:
```

```
    int findMinStep(string board, string hand) {
        int result = dfs(board, hand);
        return result == INT_MAX ? -1 : result;  }
private:
    string removeConsecutive(string board) {
    int i = 0;
    while (i < board.size()) {
    int j = i;
    while (j < board.size() && board[j] == board[i]) {
     j++;            }
     if (j - i >= 3) {
     return removeConsecutive(board.substr(0, i) + board.substr(j));          }
     i = j;         }
     return board;    }
    int dfs(string board, string hand) {
    if (board.empty()) return 0; // If the board is cleared
    if (hand.empty()) return INT_MAX; // No more balls in hand
     int minSteps = INT_MAX;
     for (int i = 0; i < hand.size(); ++i) {
     for (int j = 0; j <= board.size(); ++j) {
     string newBoard = board.substr(0, j) + hand[i] + board.substr(j);
     newBoard = removeConsecutive(newBoard);
     string newHand = hand.substr(0, i) + hand.substr(i + 1);
     int result = dfs(newBoard, newHand);
```

```
        if (result != INT_MAX) {
        minSteps = min(minSteps, result + 1); }  }  }
       return minSteps;   }
};
int main() {
    ZumaGame game;
    cout << game.findMinStep("WRRBBW", "RB") << endl;  // Output: -1
    cout << game.findMinStep("WWRRBBWW", "WRBRW") << endl;  // Output: 2
    cout << game.findMinStep("G", "GGGGG") << endl;  // Output: 2
    return 0;
}
```