1. Given a string s and an integer k, find the length of the **longest substring** that contains **exactly k unique characters**. If no such substring exists, return -1.

```
#include <iostream>
#include <unordered_map>
#include <deque>
#include <algorithm>
using namespace std;

int longestSubstringWithKUnique(string s, int k) {
    unordered_map<char, int> m; // character frequency
    deque<char> window; // simulate a sliding window
    int maxLen = -1, start = 0; // result and left index

    // expand the window
    for (int end = 0; end < s.length(); ++end) {
        m[s[end]]++; // update count
        window.push_back(s[end]); // add to window

        // reduce size if too many unique chars
        while (m.size() > k) {
            m[window.front()]--; // reduce frequency
            if (m[window.front()] == 0) m.erase(window.front()); // remove if count zero
            window.pop_front(); // move left
            start++; // increment start pointer
        }

        // check if condition is satisfied
        if (m.size() == k)
            maxLen = max(maxLen, end - start + 1); // update max length
    }
```

```
    return maxLen; // final result
}

int main() {
    string s = "aabacbebebe";
    int k = 3;
    cout << longestSubstringWithKUnique(s, k) << endl;
    return 0;
}
```

2. Given a 2D matrix of size n x m, return the **boundary traversal** of the matrix in **clockwise direction**, starting from the top-left element.

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

void traverseBoundary(const vector<vector<int>>& mat, int n, int m, vector<int>& res) {
    for (int i = 0; i < m; i++) res.push_back(mat[0][i]);
    for (int i = 1; i < n; i++) res.push_back(mat[i][m - 1]);
    if (n > 1)
        for (int i = m - 2; i >= 0; i--) res.push_back(mat[n - 1][i]);
    if (m > 1)
        for (int i = n - 2; i > 0; i--) res.push_back(mat[i][0]);
}

vector<int> boundaryTraversal(vector<vector<int>>& matrix) {
    vector<int> res;
    traverseBoundary(matrix, matrix.size(), matrix[0].size(), res);
    return res;
}

int main() {
```

```
    vector<vector<int>> matrix = {
        {1, 2, 3, 4},
        {5, 6, 7, 8},
        {9, 10, 11, 12}
    };
    vector<int> result = boundaryTraversal(matrix);
    for (int val : result) cout << val << " ";
    cout << endl;
    return 0;
}
```

3. Write a function that evaluates a simple arithmetic expression string containing only non-negative integers, +, -, and parentheses (). The expression can have any valid nesting of parentheses.

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

int evaluateExpression(string expression) {
    stack<int> numStack;
    int currentNum = 0;
    char currentOp = '+';
    expression += '+'; // Append '+' to ensure last number is processed

    for (int i = 0; i < expression.size(); i++) {
        char c = expression[i];

        if (isdigit(c)) {
            currentNum = currentNum * 10 + (c - '0');
        }
```

```
        if ((c == '+' || c == '-' || c == '(' || c == ')') || i == expression.size() - 1) {
            if (currentOp == '+') numStack.push(currentNum);
            else if (currentOp == '-') numStack.push(-currentNum);

            if (c == '(') {
                numStack.push(currentNum);  // Temporarily store
            } else if (c == ')') {
                int sum = 0;
                while (!numStack.empty()) {
                    sum += numStack.top();
                    numStack.pop();
                }
                numStack.push(sum);  // Calculate the sum of numbers inside parentheses
            }

            if (c == '+' || c == '-') currentOp = c;
            currentNum = 0;
        }
    }

    int total = 0;
    while (!numStack.empty()) {
        total += numStack.top();
        numStack.pop();
    }

    return total;
}

int main() {
    string expr = "2+(3-1)+4";
```

```
    cout << evaluateExpression(expr) << endl;
    return 0;
}
```

4. You are given a polygon NP defined by its vertices (npVertices) and a set of rectangular plots defined by their bottom-left and top-right coordinates. Determine whether a **subset of the given plots can exactly cover** the polygon without overlaps or gaps. The function isExactCover (currently a placeholder) should check whether the area covered by selected plots **exactly matches** the polygon NP.

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

// Placeholder for actual convex hull and coverage logic
bool canCoverNPWithPlots(vector<pair<int, int>>& npVertices, vector<pair<pair<int, int>, pair<int, int>>>& plots) {
    // Compute convex hull of npVertices
    // Attempt to cover convex hull with plots
    return false; // Placeholder
}

int main() {
    // Example for npVertices and plots can be provided here
    vector<pair<int, int>> npVertices = {{0, 0}, {1, 0}, {1, 1}, {0, 1}};
    vector<pair<pair<int, int>, pair<int, int>>> plots = {
        {{0, 0}, {1, 0}},
        {{1, 0}, {1, 1}},
        {{1, 1}, {0, 1}},
        {{0, 1}, {0, 0}}
    };
    cout << (canCoverNPWithPlots(npVertices, plots) ? "Yes" : "No") << endl;
    return 0;
}
```