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 <algorithm>
using namespace std;

int helper(string& s, int k, int start, unordered_map<string, int>& memo) {
    if (start >= s.length()) return -1;
    string key = to_string(start) + "," + to_string(k);
    if (memo.count(key)) return memo[key];

    unordered_map<char, int> freq;
    int maxLen = -1;

    for (int end = start; end < s.length(); ++end) {
        freq[s[end]]++;
        if (freq.size() == k) maxLen = max(maxLen, end - start + 1);
        if (freq.size() > k) break;
    }

    return memo[key] = max(maxLen, helper(s, k, start + 1, memo));
}

int longestSubstringWithKUnique(string s, int k) {
    unordered_map<string, int> memo;
    return helper(s, k, 0, memo);
}

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;

vector<int> boundaryTraversal(vector<vector<int>>& matrix) {
    vector<int> res;
    int n = matrix.size(), m = matrix[0].size();

    auto push = [&](int i, int j) {
        res.push_back(matrix[i][j]);
    };

    for (int j = 0; j < m; ++j) push(0, j);
    for (int i = 1; i < n; ++i) push(i, m - 1);
    if (n > 1)
        for (int j = m - 2; j >= 0; --j) push(n - 1, j);
    if (m > 1)
        for (int i = n - 2; i > 0; --i) push(i, 0);

    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> nums;
    char lastOp = '+';
    int num = 0;
    expression += "+";  // To handle the last number

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

        if (isdigit(ch)) {
            num = num * 10 + (ch - '0');
        }
        if ((ch == '+' || ch == '-' || ch == '(' || ch == ')') || i == expression.size() - 1) {
            if (lastOp == '+') nums.push(num);
            else if (lastOp == '-') nums.push(-num);

            if (ch == '(') {
                nums.push(-1); // Placeholder for opening parenthesis
            } else if (ch == ')') {
                int sum = 0;
                while (nums.top() != -1) {
                    sum += nums.top();
                    nums.pop();
                }
                nums.pop();  // Remove the placeholder
                nums.push(sum);
            }

            if (ch == '+' || ch == '-') lastOp = ch;
            num = 0;
        }
    }

    int result = 0;
    while (!nums.empty()) {
        result += nums.top();
        nums.pop();
    }

    return result;
}

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 function for checking plot coverage
bool canCoverNPWithPlots(vector<pair<int, int>>& npVertices, vector<pair<pair<int, int>, pair<int, int>>>& plots) {
    // Randomly select subsets of plots and check coverage
    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;
}
```