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 <set>
#include <string>
using namespace std;

int longestSubstringWithKUnique(string s, int k) {
    multiset<char> window;
    unordered_map<char, int> freq;
    int start = 0, ans = -1;

    for (int end = 0; end < s.length(); end++) {
        window.insert(s[end]);
        freq[s[end]]++;

        while (freq.size() > k) {
            char ch = s[start];
            window.erase(window.find(ch));
            if (--freq[ch] == 0) freq.erase(ch);
            start++;
        }

        if (freq.size() == k)
            ans = max(ans, end - start + 1);
    }
    return ans;
}

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

    for (int i = 0; i < m; ++i) result.push_back(matrix[0][i]);
    for (int i = 1; i < n - 1; ++i) result.push_back(matrix[i][m - 1]);
    if (n > 1)
        for (int i = m - 1; i >= 0; --i) result.push_back(matrix[n - 1][i]);
    if (m > 1)
        for (int i = n - 2; i > 0; --i) result.push_back(matrix[i][0]);

    return result;
}

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> values;
    char operatorChar = '+';
    int num = 0;
    expression += "+";

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

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

        if ((c == '+' || c == '-' || c == '(' || c == ')') || i == expression.size() - 1) {
            if (operatorChar == '+') values.push(num);
            else if (operatorChar == '-') values.push(-num);

            if (c == '(') {
```

```
            operatorChar = '(';
        } else if (c == ')') {
            int resultInParentheses = 0;
            while (!values.empty()) {
                resultInParentheses += values.top();
                values.pop();
            }
            values.push(resultInParentheses);
        }

        if (c == '+' || c == '-') operatorChar = c;
        num = 0;
      }
   }

   int finalResult = 0;
   while (!values.empty()) {
      finalResult += values.top();
      values.pop();
   }

   return finalResult;
}

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>
#include <algorithm>
using namespace std;

bool canCoverNPWithPlots(vector<pair<int, int>>& npVertices, vector<pair<pair<int, int>, pair<int, int>>>& plots) {
    sort(plots.begin(), plots.end(), [](const auto& a, const auto& b) {
        return area(a) > area(b); // Sort plots by area in descending order
    });
    vector<pair<int, int>> coveredArea;
    for (const auto& plot : plots) {
        // Try to add plot to coveredArea
        if (isExactCover(npVertices, coveredArea)) {
            return true;
        }
    }
    return false;
}

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;
}
```