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

class CharWindow {
   unordered_map<char, int> counts;
public:
   void add(char c) { counts[c]++; }
   void remove(char c) {
      counts[c]--;
      if (counts[c] == 0) counts.erase(c);
   }
   int size() { return counts.size(); }
};

int longestSubstringWithKUnique(string s, int k) {
   CharWindow window;
   int i = 0, j = 0, maxLen = -1;

   while (j < s.length()) {
      window.add(s[j]);

      while (window.size() > k) {
         window.remove(s[i]);
         i++;
      }
```

```
        if (window.size() == k)
            maxLen = max(maxLen, j - i + 1);
        j++;
    }
    return maxLen;
}

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 i = 0, j = 0, n = matrix.size(), m = matrix[0].size();

    while (j < m) res.push_back(matrix[0][j++]);
    i = 1; j = m - 1;
    while (i < n) res.push_back(matrix[i++][j]);
    i = n - 1; j = m - 2;
    if (n > 1)
        while (j >= 0) res.push_back(matrix[i][j--]);
    i = n - 2; j = 0;
    if (m > 1)
```

```
        while (i > 0) res.push_back(matrix[i--][j]);

    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;
    stack<char> opStack;
    int currentNum = 0;
    char currentOp = '+';
    expression += '+'; // 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');
    }

    // Handle operator and parentheses logic
    if ((c == '+' || c == '-' || c == '(' || c == ')') || i == expression.size() - 1) {
        if (currentOp == '+') numStack.push(currentNum);
        else if (currentOp == '-') numStack.push(-currentNum);

        if (c == '(') opStack.push(currentOp);
        else if (c == ')') {
            int tempSum = 0;
            while (!numStack.empty()) {
                tempSum += numStack.top();
                numStack.pop();
            }
            numStack.push(tempSum); // Push the result of parenthesis evaluated
        }

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

int result = 0;
while (!numStack.empty()) {
    result += numStack.top();
    numStack.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 for actual logic that involves building quadtree
bool canCoverNPWithPlots(vector<pair<int, int>>& npVertices, vector<pair<pair<int, int>, pair<int, int>>>& plots) {
    // Build quadtree from npVertices
    // Recursively divide NP and attempt to cover each quadrant
    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;
}
```