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 longestSubstringWithKUnique(string s, int k) {
   int maxLen = -1;
   for (int i = 0; i < s.length(); ++i) {
      unordered_map<char, int> freq;
      for (int j = i; j < s.length(); ++j) {
         freq[s[j]]++;
         if (freq.size() == k) maxLen = max(maxLen, j - i + 1);
         if (freq.size() > k) break;
      }
   }
   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) {
    int n = matrix.size(), m = matrix[0].size();
    vector<vector<bool>> visited(n, vector<bool>(m, false));
    vector<int> res;

    // Traverse top row
    for (int j = 0; j < m; j++) {
        res.push_back(matrix[0][j]);
        visited[0][j] = true;
    }

    // Traverse right column
    for (int i = 1; i < n; i++) {
        res.push_back(matrix[i][m - 1]);
        visited[i][m - 1] = true;
    }

    // Traverse bottom row
    for (int j = m - 2; j >= 0; j--) {
        if (!visited[n - 1][j]) res.push_back(matrix[n - 1][j]);
    }

    // Traverse left column
    for (int i = n - 2; i > 0; i--) {
        if (!visited[i][0]) res.push_back(matrix[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> valueStack;
    stack<char> operatorStack;
    int currentNum = 0;
    char lastOperator = '+';
    expression += '+';  // To handle the last number

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

        if (isdigit(currentChar)) {
```

```
            currentNum = currentNum * 10 + (currentChar - '0');
        }

        if ((currentChar == '+' || currentChar == '-' || currentChar == '(' || currentChar == ')') || i ==
expression.size() - 1) {
            if (lastOperator == '+') valueStack.push(currentNum);
            else if (lastOperator == '-') valueStack.push(-currentNum);

            if (currentChar == '(') operatorStack.push(lastOperator);
            else if (currentChar == ')') {
                int insideParentheses = 0;
                while (!valueStack.empty()) {
                    insideParentheses += valueStack.top();
                    valueStack.pop();
                }
                valueStack.push(insideParentheses);
            }

            if (currentChar == '+' || currentChar == '-') lastOperator = currentChar;
            currentNum = 0;
        }
    }

    int result = 0;
    while (!valueStack.empty()) {
        result += valueStack.top();
        valueStack.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 ILP solver for NP and plots coverage
bool canCoverNPWithPlots(vector<pair<int, int>>& npVertices, vector<pair<pair<int, int>, pair<int, int>>>& plots) {
   // Formulate ILP to minimize uncovered area
   // Solve ILP using a solver
   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;
}
```