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

int longestSubstringWithKUnique(string s, int k) {
    int freq[26] = {0};
    int unique = 0, left = 0, maxLen = -1;

    for (int right = 0; right < s.size(); ++right) {
        if (freq[s[right] - 'a']++ == 0) unique++;

        while (unique > k) {
            if (--freq[s[left] - 'a'] == 0) unique--;
            left++;
        }

        if (unique == k)
            maxLen = max(maxLen, right - left + 1);
    }

    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 rows = matrix.size();
  int cols = matrix[0].size();
  vector<int> boundary;

  for (int col = 0; col < cols; col++)
    boundary.push_back(matrix[0][col]);

  for (int row = 1; row < rows; row++)
    boundary.push_back(matrix[row][cols - 1]);

  if (rows > 1)
    for (int col = cols - 2; col >= 0; col--)
      boundary.push_back(matrix[rows - 1][col]);

  if (cols > 1)
    for (int row = rows - 2; row > 0; row--)
      boundary.push_back(matrix[row][0]);

  return boundary;
}

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> numbers;
    char op = '+';
    int currentNumber = 0;
    expression += "+";

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

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

        if ((c == '+' || c == '-' || c == '(' || c == ')') || i == expression.size() - 1) {
            if (op == '+') numbers.push(currentNumber);
            else if (op == '-') numbers.push(-currentNumber);

            if (c == '(') op = '(';
            else if (c == ')') {
                int sum = 0;
                while (!numbers.empty()) {
                    sum += numbers.top();
                    numbers.pop();
                }
                numbers.push(sum);
            }

            if (c == '+' || c == '-') op = c;
            currentNumber = 0;
        }
    }

    int total = 0;
    while (!numbers.empty()) {
        total += numbers.top();
        numbers.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;


bool isExactCover(vector<pair<int, int>>& npVertices, vector<pair<int, int>>& coveredArea) {
  return false; // placeholder
}


bool backtrack(int idx, vector<pair<int, int>>& npVertices, vector<pair<pair<int, int>, pair<int,
int>>>& plots, vector<pair<int, int>>& currentCover) {
  if (idx == plots.size()) {
    return isExactCover(npVertices, currentCover);
  }
  currentCover.push_back(plots[idx].first);
  currentCover.push_back(plots[idx].second);
  if (backtrack(idx + 1, npVertices, plots, currentCover)) return true;
  currentCover.pop_back();
  currentCover.pop_back();
  if (backtrack(idx + 1, npVertices, plots, currentCover)) return true;
  return false;
}


bool canCoverNPWithPlots(vector<pair<int, int>>& npVertices, vector<pair<pair<int, int>, pair<int,
int>>>& plots) {
  vector<pair<int, int>> currentCover;
  return backtrack(0, npVertices, plots, currentCover);
}


int main() {
    vector<pair<int, int>> np = {{0,0}, {0,2}, {2,2}, {2,0}};
    vector<pair<pair<int, int>, pair<int, int>>> plots = {
        {{0,0}, {1,1}}, {{1,0}, {2,1}}, {{0,1}, {1,2}}, {{1,1}, {2,2}}
    };
    cout << (canCoverNPWithPlots(np, plots) ? "Yes" : "No") << endl;
    return 0;
}
```