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>
using namespace std;

int longestSubstringWithKUnique(string s, int k) {
   unordered_map<char, int> count; // map to store frequency of characters in current window
   int left = 0, right = 0, maxLen = -1; // window pointers and result variable

   // iterate through the string with right pointer
   while (right < s.length()) {
      count[s[right]]++; // include character at right in map

      // shrink window from left if unique character count exceeds k
      while (count.size() > k) {
         count[s[left]]--;
         if (count[s[left]] == 0) count.erase(s[left]); // remove character if frequency is zero
         left++; // move left pointer forward
      }

      // check if current window has exactly k unique characters
      if (count.size() == k)
         maxLen = max(maxLen, right - left + 1); // update maximum length
      right++; // move right pointer
   }
   return maxLen; // return the maximum length found
}

int main() {
```

```
    string s = "abcba";
    int k = 2;
    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> res;

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

    return res;
}

int main() {
    vector<vector<int>> matrix = {
        {1, 2, 3},
        {4, 5, 6},
        {7, 8, 9}
    };
```

```
    vector<int> result = boundaryTraversal(matrix);
    for (int x : result) cout << x << " ";
    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, ops;
    int num = 0;
    char op = '+';
    expression += "+"; // To ensure the last number is processed.

    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 (op == '+') nums.push(num);
            else if (op == '-') nums.push(-num);
```

```
            if (c == '(') ops.push(op);
            else if (c == ')') {
                int temp = 0;
                while (!nums.empty()) {
                    temp += nums.top();
                    nums.pop();
                }
                nums.push(temp);
            }

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

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

    return result;
}

int main() {
    string expr = "1+(2-3)+(4+5)";
    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(const vector<pair<int, int>>& npVertices, const vector<pair<int, int>>&
coveredArea) {
    return false; // Placeholder for actual polygon coverage logic
}

bool canCoverNPWithPlots(vector<pair<int, int>>& npVertices, vector<pair<pair<int, int>, pair<int,
int>>>& plots) {
    int n = plots.size();
    for (int mask = 1; mask < (1 << n); ++mask) {
        vector<pair<int, int>> coveredArea;
        for (int i = 0; i < n; ++i) {
            if (mask & (1 << i)) {
                // Add plot[i] to coveredArea
                coveredArea.push_back(plots[i].first);
                coveredArea.push_back(plots[i].second);
            }
        }
        if (isExactCover(npVertices, coveredArea)) {
            return true;
        }
    }
    return false;
}
```

```
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;
}
```