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> m;
  int left = 0, right = 0, len = -1, uniqueCount = 0;

  while (right < s.size()) {
    if (m[s[right]] == 0) uniqueCount++;
    m[s[right]]++;

    while (uniqueCount > k) {
      m[s[left]]--;
      if (m[s[left]] == 0) {
        m.erase(s[left]);
        uniqueCount--;
      }
      left++;
    }

    if (uniqueCount == k)
      len = max(len, right - left + 1);
    right++;
  }
  return len;
}
```

```
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> result;
   int n = matrix.size();
   int m = matrix[0].size();

   for (int j = 0; j < m; j++) {
      result.push_back(matrix[0][j]);
   }

   for (int i = 1; i < n; i++) {
      result.push_back(matrix[i][m - 1]);
   }

   if (n > 1) {
      for (int j = m - 2; j >= 0; j--) {
         result.push_back(matrix[n - 1][j]);
      }
   }
```

```
    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> numberStack, operatorStack;
    int currentNumber = 0;
    char lastOperator = '+';
    expression += "+";

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

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

        if (currentChar == '+' || currentChar == '-' || currentChar == '(' || currentChar == ')' || i ==
expression.size() - 1) {
            if (lastOperator == '+') {
                numberStack.push(currentNumber);
            } else if (lastOperator == '-') {
                numberStack.push(-currentNumber);
            }

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

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

            currentNumber = 0;
        }
    }

    int result = 0;
    while (!numberStack.empty()) {
        result += numberStack.top();
        numberStack.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;
```

```
bool canCoverNPWithPlots(vector<pair<int, int>>& npVertices, vector<pair<pair<int, int>, pair<int, int>>>& plots) {
    int n = plots.size();
    vector<vector<bool>> dp(1 << n, vector<bool>(npVertices.size(), false));
    dp[0][0] = true;

    for (int mask = 0; mask < (1 << n); ++mask) {
        for (int i = 0; i < npVertices.size(); ++i) {
            if (dp[mask][i]) {
                for (int j = 0; j < n; ++j) {
                    if (!(mask & (1 << j))) {
                        // Update dp[mask | (1 << j)][newIndex]
                    }
                }
            }
        }
    }
    return dp[(1 << n) - 1][npVertices.size() - 1];
}

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;
}
```