```
#include <iostream>
#include <vector>
#include <algorithm>
#include <random>
#include <chrono>
#include <cmath>
#include <stack>
#include <string>
#include <sstream>
#include <functional>
#include <stdexcept>

using namespace std;

// ====================== 第一部分：复数类实现 ======================

class Complex {
private:
    double real;
    double imag;

public:
    Complex(double r = 0, double i = 0) : real(r), imag(i) {}

    // 获取模
    double modulus() const {
```

```
        return sqrt(real * real + imag * imag);
}

// 重载运算符
bool operator==(const Complex& other) const {
        return abs(real - other.real) < 1e-9 && abs(imag - other.imag) < 1e-9;
}

bool operator<(const Complex& other) const {
        double mod1 = modulus();
        double mod2 = other.modulus();
        if (abs(mod1 - mod2) < 1e-9) {
                return real < other.real;
        }
        return mod1 < mod2;
}

bool operator>(const Complex& other) const {
        return other < *this;
}

bool operator<=(const Complex& other) const {
        return !(other < *this);
}

bool operator>=(const Complex& other) const {
        return !(*this < other);
```

```
    }

    // 友元函数用于输出
    friend ostream& operator<<(ostream& os, const Complex& c) {
        os << "(" << c.real << (c.imag >= 0 ? "+" : "") << c.imag << "i)";
        return os;
    }

    // getter 方法
    double getReal() const { return real; }
    double getImag() const { return imag; }
};

// 置乱函数
template<typename T>
void shuffleVector(vector<T>& vec) {
    random_device rd;
    mt19937 g(rd());
    shuffle(vec.begin(), vec.end(), g);
}

// 查找函数
template<typename T>
int findVector(const vector<T>& vec, const T& target) {
    auto it = find(vec.begin(), vec.end(), target);
    return it != vec.end() ? distance(vec.begin(), it) : -1;
}
```

```
// 唯一化函数
template<typename T>
void uniqueVector(vector<T>& vec) {
    sort(vec.begin(), vec.end());
    auto last = unique(vec.begin(), vec.end());
    vec.erase(last, vec.end());
}

// 起泡排序
template<typename T>
void bubbleSort(vector<T>& vec) {
    int n = vec.size();
    for (int i = 0; i < n - 1; i++) {
        for (int j = 0; j < n - i - 1; j++) {
            if (vec[j + 1] < vec[j]) {
                swap(vec[j], vec[j + 1]);
            }
        }
    }
}

// 归并排序辅助函数
template<typename T>
void merge(vector<T>& vec, int left, int mid, int right) {
    vector<T> temp(right - left + 1);
    int i = left, j = mid + 1, k = 0;
```

```
        while (i <= mid && j <= right) {
            if (vec[i] < vec[j]) {
                temp[k++] = vec[i++];
            } else {
                temp[k++] = vec[j++];
            }
        }

        while (i <= mid) temp[k++] = vec[i++];
        while (j <= right) temp[k++] = vec[j++];

        for (int p = 0; p < k; p++) {
            vec[left + p] = temp[p];
        }
}

template<typename T>
void mergeSort(vector<T>& vec, int left, int right) {
        if (left >= right) return;

        int mid = left + (right - left) / 2;
        mergeSort(vec, left, mid);
        mergeSort(vec, mid + 1, right);
        merge(vec, left, mid, right);
}
```

```
template<typename T>
void mergeSort(vector<T>& vec) {
    if (vec.empty()) return;
    mergeSort(vec, 0, vec.size() - 1);
}

// 区间查找
vector<Complex> rangeSearch(const vector<Complex>& vec, double m1, double m2) {
    vector<Complex> result;
    for (const auto& c : vec) {
        double mod = c.modulus();
        if (mod >= m1 && mod <= m2) {
            result.push_back(c);
        }
    }
    // 保持原顺序
    return result;
}

// 生成随机复数向量
vector<Complex> generateRandomComplexVector(int size) {
    vector<Complex> complexes;
    random_device rd;
    mt19937 gen(rd());
    uniform_real_distribution<> dis(-10, 10);

    for (int i = 0; i < size; i++) {
```

```
        complexes.push_back(Complex(dis(gen), dis(gen)));
    }

    // 添加一些重复项
    if (size >= 3) {
        complexes.push_back(complexes[0]);
        complexes.push_back(complexes[1]);
    }

    return complexes;
}

// 测试复数类功能
void testComplexClass() {
    cout << "========== 复数类测试 ==========" << endl;

    // 生成随机复数向量
    vector<Complex> complexes = generateRandomComplexVector(8);

    cout << "1. 原始向量(" << complexes.size() << "个): ";
    for (const auto& c : complexes) cout << c << " ";
    cout << endl;

    // (1) 置乱测试
    shuffleVector(complexes);
    cout << "2. 置乱后: ";
    for (const auto& c : complexes) cout << c << " ";
```

```
cout << endl;

// 查找测试
if (!complexes.empty()) {
    Complex target = complexes[complexes.size() / 2];
    int pos = findVector(complexes, target);
    cout << "3. 查找 " << target << " 的位置: " << pos << endl;
}

// 插入测试
complexes.insert(complexes.begin() + 2, Complex(7.5, 8.5));
cout << "4. 插入(7.5+8.5i)后: ";
for (const auto& c : complexes) cout << c << " ";
cout << endl;

// 删除测试
complexes.erase(complexes.begin() + 2);
cout << "5. 删除第三个元素后: ";
for (const auto& c : complexes) cout << c << " ";
cout << endl;

// 唯一化测试
cout << "6. 唯一化前大小: " << complexes.size() << endl;
uniqueVector(complexes);
cout << "   唯一化后大小: " << complexes.size() << endl;
cout << "   唯一化后向量: ";
for (const auto& c : complexes) cout << c << " ";
```

```
cout << endl;

// (2) 排序效率比较
cout << "\n7. 排序效率比较:" << endl;

// 准备测试数据
vector<Complex> ordered = complexes;
sort(ordered.begin(), ordered.end());

vector<Complex> reversed = ordered;
reverse(reversed.begin(), reversed.end());

vector<Complex> random = complexes;

// 测试函数
auto testSorting = [](const string& name, vector<Complex> data,
                      function<void(vector<Complex>&)> sortFunc) {
    auto start = chrono::high_resolution_clock::now();
    sortFunc(data);
    auto end = chrono::high_resolution_clock::now();
    auto duration = chrono::duration_cast<chrono::microseconds>(end - start);
    cout << "   " << name << ": " << duration.count() << " μs" << endl;
    return duration.count();
};

cout << "   --- 起泡排序 ---" << endl;
testSorting("顺序情况", ordered, bubbleSort<Complex>);
```

```
    testSorting("乱序情况", random, bubbleSort<Complex>);
    testSorting("逆序情况", reversed, bubbleSort<Complex>);

    cout << "      ---  归并排序  ---" << endl;
    testSorting("顺序情况", ordered, mergeSort<Complex>);
    testSorting("乱序情况", random, mergeSort<Complex>);
    testSorting("逆序情况", reversed, mergeSort<Complex>);

    // (3)  区间查找测试
    vector<Complex> sorted = complexes;
    sort(sorted.begin(), sorted.end());
    vector<Complex> rangeResult = rangeSearch(sorted, 3.0, 8.0);
    cout << "\n8.  模在[3,8]区间的复数(" << rangeResult.size() << "个): ";
    for (const auto& c : rangeResult) cout << c << " ";
    cout << endl;
}

// ====================  第二部分:  栈计算器实现
====================

//  栈模板类
template<typename T>
class Stack {
private:
    vector<T> data;

public:
```

```
void push(const T& value) {
    data.push_back(value);
}

T pop() {
    if (empty()) {
        throw runtime_error("栈为空");
    }
    T value = data.back();
    data.pop_back();
    return value;
}

T top() const {
    if (empty()) {
        throw runtime_error("栈为空");
    }
    return data.back();
}

bool empty() const {
    return data.empty();
}

size_t size() const {
    return data.size();
}
```

```
};

class Calculator {
private:
    Stack<char> opStack;
    Stack<double> numStack;

    // 优先级表
    int getPriority(char op) {
        switch (op) {
            case '+': case '-': return 1;
            case '*': case '/': return 2;
            case '^': return 3;
            case 's': case 'c': case 't': case 'l': return 4; // sin, cos, tan, log
            default: return 0;
        }
    }

    bool isOperator(char c) {
        return c == '+' || c == '-' || c == '*' || c == '/' || c == '^' ||
               c == 's' || c == 'c' || c == 't' || c == 'l';
    }

    bool isDigit(char c) {
        return (c >= '0' && c <= '9') || c == '.';
    }
```

```
double calculate(double a, double b, char op) {
    switch (op) {
        case '+': return a + b;
        case '-': return a - b;
        case '*': return a * b;
        case '/':
            if (abs(b) < 1e-9) throw runtime_error("除零错误");
            return a / b;
        case '^': return pow(a, b);
        default: throw runtime_error("未知运算符");
    }
}

double calculateFunction(double a, char func) {
    switch (func) {
        case 's': return sin(a); // sin
        case 'c': return cos(a); // cos
        case 't': return tan(a); // tan
        case 'l':
            if (a <= 0) throw runtime_error("对数参数必须大于 0");
            return log(a); // log
        default: throw runtime_error("未知函数");
    }
}

void processOperator(char op) {
    if (op == 's' || op == 'c' || op == 't' || op == 'l') {
```

```
            // 单目运算符
            if (numStack.empty()) throw runtime_error("表达式错误");
            double a = numStack.pop();
            numStack.push(calculateFunction(a, op));
        } else {
            // 双目运算符
            if (numStack.size() < 2) throw runtime_error("表达式错误");
            double b = numStack.pop();
            double a = numStack.pop();
            numStack.push(calculate(a, b, op));
        }
    }

public:
    double evaluate(const string& expression) {
        // 清空栈
        while (!opStack.empty()) opStack.pop();
        while (!numStack.empty()) numStack.pop();

        string expr = expression + "#"; // 结束标志
        string currentNum = "";

        for (size_t i = 0; i < expr.length(); i++) {
            char c = expr[i];

            if (c == ' ') continue; // 跳过空格
```

```
            if (isDigit(c)) {
                currentNum += c;
            } else {
                if (!currentNum.empty()) {
                    try {
                        numStack.push(stod(currentNum));
                        currentNum = "";
                    } catch (const exception& e) {
                        throw runtime_error("数字格式错误: " + currentNum);
                    }
                }

                if (c == '(') {
                    opStack.push(c);
                } else if (c == ')') {
                    while (!opStack.empty() && opStack.top() != '(') {
                        char op = opStack.pop();
                        processOperator(op);
                    }
                    if (opStack.empty()) throw runtime_error("括号不匹配");
                    opStack.pop(); //  弹出  '('

                    //  检查函数调用
                    if (!opStack.empty() && (opStack.top() == 's' || opStack.top() ==
'c' ||
                                             opStack.top() == 't' || opStack.top()
== 'l')) {
```

```
                    char func = opStack.pop();
                    processOperator(func);
                }
            } else if (c == '#') {
                while (!opStack.empty()) {
                    char op = opStack.pop();
                    processOperator(op);
                }
            } else {
                // 处理函数名
                if (c == 's' && i + 2 < expr.length() && expr.substr(i, 3) == "sin")
{
                    opStack.push('s');
                    i += 2; // 跳过"in"
                } else if (c == 'c' && i + 2 < expr.length() && expr.substr(i, 3) ==
"cos") {
                    opStack.push('c');
                    i += 2; // 跳过"os"
                } else if (c == 't' && i + 2 < expr.length() && expr.substr(i, 3) ==
"tan") {
                    opStack.push('t');
                    i += 2; // 跳过"an"
                } else if (c == 'l' && i + 2 < expr.length() && expr.substr(i, 3) ==
"log") {
                    opStack.push('l');
                    i += 2; // 跳过"og"
                } else if (isOperator(c)) {
                    while (!opStack.empty() && getPriority(opStack.top()) >=
```

```
getPriority(c)) {
                                    char op = opStack.pop();
                                    processOperator(op);
                                }
                                opStack.push(c);
                            } else if (c != '#') {
                                throw runtime_error("无效字符: " + string(1, c));
                            }
                        }
                    }
                }

                if (numStack.size() != 1) {
                    throw runtime_error("表达式不完整");
                }
                return numStack.pop();
            }
};

// 测试计算器
void testCalculator() {
    cout << "\n========== 栈计算器测试 ==========" << endl;
    Calculator calc;

    vector<pair<string, double>> testCases = {
        {"2 + 3", 5},
        {"3 * 4", 12},
```

```
        {"10 - 5", 5},
        {"15 / 3", 5},
        {"2 + 3 * 4", 14},
        {"(2 + 3) * 4", 20},
        {"10 / 2 - 1", 4},
        {"2 ^ 3", 8},
        {"3 + 4 * 2 / (1 - 5) ^ 2", 3.5},
        {"sin(0)", 0},
        {"cos(0)", 1},
        {"log(1)", 0},
        {"sin(3.1415926/2)", 1}   // 近似值
};

for (const auto& testCase : testCases) {
        try {
                double result = calc.evaluate(testCase.first);
                double expected = testCase.second;
                double error = abs(result - expected);
                cout << "测试: " << testCase.first << " = " << result;
                if (error < 1e-6) {
                        cout << " ✓" << endl;
                } else {
                        cout << " ✗ (期望: " << expected << ")" << endl;
                }
        } catch (const exception& e) {
                cout << "测试: " << testCase.first << " -> 错误: " << e.what() << endl;
```

```
            }
        }
    }

// ====================== 第三部分：柱状图最大面积实现 ======================

int largestRectangleArea(vector<int>& heights) {
        int n = heights.size();
        Stack<int> st; //  存储索引的栈
        int maxArea = 0;

        for (int i = 0; i <= n; i++) {
            int currentHeight = (i == n) ? 0 : heights[i];

            while (!st.empty() && currentHeight < heights[st.top()]) {
                int height = heights[st.pop()];
                int width = st.empty() ? i : i - st.top() - 1;
                maxArea = max(maxArea, height * width);
            }

            st.push(i);
        }

        return maxArea;
}
```

```
// 生成随机测试数据
vector<vector<int>> generateTestData(int numTests) {
    vector<vector<int>> testData;
    random_device rd;
    mt19937 gen(rd());

    for (int i = 0; i < numTests; i++) {
        uniform_int_distribution<> sizeDis(1, 20); // 为了演示使用较小尺寸
        uniform_int_distribution<> heightDis(0, 50);

        int size = sizeDis(gen);
        vector<int> heights(size);
        for (int j = 0; j < size; j++) {
            heights[j] = heightDis(gen);
        }
        testData.push_back(heights);
    }

    return testData;
}

// 测试柱状图最大面积
void testLargestRectangleArea() {
    cout << "\n========== 柱状图最大面积测试 ==========" << endl;

    // 示例测试
    vector<int> heights1 = {2, 1, 5, 6, 2, 3};
```

```
    vector<int> heights2 = {2, 4};
    vector<int> heights3 = {1, 1, 1, 1, 1};
    vector<int> heights4 = {4, 2, 0, 3, 2, 5};

    cout << "示例测试:" << endl;
    cout << "1. [2,1,5,6,2,3] = " << largestRectangleArea(heights1) << " (期望: 10)" << endl;
    cout << "2. [2,4] = " << largestRectangleArea(heights2) << " (期望: 4)" << endl;
    cout << "3. [1,1,1,1,1] = " << largestRectangleArea(heights3) << " (期望: 5)" << endl;
    cout << "4. [4,2,0,3,2,5] = " << largestRectangleArea(heights4) << " (期望: 6)" << endl;

    // 随机测试
    cout << "\n 随机测试:" << endl;
    vector<vector<int>> randomTests = generateTestData(10);
    for (size_t i = 0; i < randomTests.size(); i++) {
        int area = largestRectangleArea(randomTests[i]);
        cout << i + 1 << ". [";
        for (size_t j = 0; j < randomTests[i].size(); j++) {
            cout << randomTests[i][j] << (j < randomTests[i].size() - 1 ? "," : "");
        }
        cout << "] = " << area << endl;
    }
}

// ==================== 主函数 ====================
```

```
int main() {
    cout << "第一次代码作业：线性数据结构" << endl;
    cout << "====================================" << endl;

    // 测试复数类
    testComplexClass();

    // 测试计算器
    testCalculator();

    // 测试柱状图最大面积
    testLargestRectangleArea();

    cout << "\n====================================" << endl;
    cout << "所有测试完成！" << endl;

    return 0;
}
```