

=====

文件夹: class013_CircularDequeAndMonotonicQueue

=====

[Markdown 文件]

=====

文件: README.md

=====

Class016: 循环双端队列与单调队列

本章节包含双端队列 (Deque) 和单调队列 (Monotonic Queue) 的核心实现和经典应用题目。

 题目列表

1. 循环双端队列设计

LeetCode 641. 设计循环双端队列

- **难度**: 中等
- **链接**: <https://leetcode.cn/problems/design-circular-deque/>
- **题目描述**: 设计实现双端队列，支持在队列两端进行插入和删除操作
- **核心思路**: 使用数组实现循环队列，通过取模运算处理边界
- **时间复杂度**: $O(1)$ – 所有操作
- **空间复杂度**: $O(k)$ – k 是队列容量
- **是否最优解**: 是

2. 滑动窗口最大值 (单调队列经典应用)

LeetCode 239. 滑动窗口最大值

- **难度**: 困难
- **链接**: <https://leetcode.cn/problems/sliding-window-maximum/>
- **题目描述**: 给定数组和窗口大小 k ，求每个窗口的最大值
- **核心思路**: 使用单调递减队列，队首始终是当前窗口最大值
- **时间复杂度**: $O(n)$ – 每个元素最多入队出队一次
- **空间复杂度**: $O(k)$ – 队列大小
- **是否最优解**: 是

洛谷 P1886 滑动窗口 / 【模板】单调队列

- **难度**: 入门
- **链接**: <https://www.luogu.com.cn/problem/P1886>
- **题目描述**: 求滑动窗口的最小值和最大值
- **核心思路**: 使用两个单调队列分别维护最小值和最大值

- **时间复杂度**: $O(n)$
- **空间复杂度**: $O(k)$
- **是否最优解**: 是

AcWing 154. 滑动窗口

- **难度**: 简单
- **链接**: <https://www.acwing.com/problem/content/156/>
- **题目描述**: 与洛谷 P1886 相同
- **核心思路**: 双单调队列
- **时间复杂度**: $O(n)$
- **空间复杂度**: $O(k)$
- **是否最优解**: 是

POJ 2823 Sliding Window

- **难度**: 中等
- **链接**: <http://poj.org/problem?id=2823>
- **题目描述**: 经典滑动窗口问题，OJ 时间要求严格
- **核心思路**: 高效的单调队列实现
- **时间复杂度**: $O(n)$
- **空间复杂度**: $O(k)$
- **是否最优解**: 是

3. 子数组和问题（前缀和+单调队列）

- #### #### LeetCode 862. 和至少为 K 的最短子数组
- **难度**: 困难
 - **链接**: <https://leetcode.cn/problems/shortest-subarray-with-sum-at-least-k/>
 - **题目描述**: 找出和至少为 k 的最短非空子数组
 - **核心思路**: 前缀和+单调递增队列优化
 - **时间复杂度**: $O(n)$
 - **空间复杂度**: $O(n)$
 - **是否最优解**: 是

LeetCode 1425. 带限制的子序列和

- **难度**: 困难
- **链接**: <https://leetcode.cn/problems/constrained-subsequence-sum/>
- **题目描述**: 求满足相邻元素下标差不超过 k 的子序列最大和
- **核心思路**: DP+单调递减队列优化
- **时间复杂度**: $O(n)$
- **空间复杂度**: $O(n)$
- **是否最优解**: 是

4. 绝对差限制问题

LeetCode 1438. 绝对差不超过限制的最长连续子数组

- **难度**: 中等
- **链接**: <https://leetcode.cn/problems/longest-continuous-subarray-with-absolute-diff-less-than-or-equal-to-limit/>
- **题目描述**: 求最长子数组，使得任意两元素绝对差 $\leqslant \text{limit}$
- **核心思路**: 滑动窗口+双单调队列（同时维护最小值和最大值）
- **时间复杂度**: $O(n)$
- **空间复杂度**: $O(n)$
- **是否最优解**: 是

5. 队列维护最值

剑指 Offer 59-II. 队列的最大值

- **难度**: 中等
- **链接**: <https://leetcode.cn/problems/dui-lie-de-zui-da-zhi-lcof/>
- **题目描述**: 设计队列支持 $O(1)$ 获取最大值
- **核心思路**: 普通队列+单调递减辅助队列
- **时间复杂度**: $O(1)$ – 所有操作均摊
- **空间复杂度**: $O(n)$
- **是否最优解**: 是

6. 双端队列模拟

洛谷 P2952 牛线 Cow Line

- **难度**: 入门
- **链接**: <https://www.luogu.com.cn/problem/P2952>
- **题目描述**: 支持在队列两端添加和删除元素
- **核心思路**: 直接使用双端队列模拟
- **时间复杂度**: $O(n)$
- **空间复杂度**: $O(m)$
- **是否最优解**: 是

7. 博弈论+区间 DP

AtCoder DP Contest L - Deque

- **难度**: 中等
- **链接**: https://atcoder.jp/contests/dp/tasks/dp_l
- **题目描述**: 两人轮流从序列两端取数，求最优策略下的分数差
- **核心思路**: 区间 DP + 博弈论， $dp[1][r]$ 表示区间 $[1, r]$ 先手能获得的最大分数差
- **状态转移**: $dp[1][r] = \max(a[1] - dp[1+1][r], a[r] - dp[1][r-1])$
- **时间复杂度**: $O(n^2)$
- **空间复杂度**: $O(n^2)$
- **是否最优解**: 是

🎯 核心知识点总结

1. 数据结构

双端队列 (Deque)

- **定义**: 可以在两端进行插入和删除的线性数据结构
- **操作**: insertFront, insertLast, deleteFront, deleteLast, getFront, getRear
- **实现方式**:
 - 数组实现（循环数组）：固定大小，空间效率高
 - 链表实现：动态大小，插入删除灵活

单调队列 (Monotonic Queue)

- **定义**: 维护队列元素单调性的特殊队列
- **类型**:
 - 单调递增队列：维护区间最小值
 - 单调递减队列：维护区间最大值
- **核心操作**:
 1. 移除超出窗口范围的元素（队首）
 2. 维护单调性（队尾）
 3. 添加新元素（队尾）
 4. 获取最值（队首）

2. 适用题型

① 滑动窗口最值问题

- **特征**: 固定或可变大小的窗口，求每个窗口的最大/最小值
- **方法**: 单调队列
- **代表题**: LeetCode 239, 洛谷 P1886

② 子数组/子序列和问题

- **特征**: 涉及前缀和，需要快速查找满足条件的区间
- **方法**: 前缀和 + 单调队列
- **代表题**: LeetCode 862, 1425

③ 绝对差限制问题

- **特征**: 需要同时维护区间最大值和最小值
- **方法**: 双单调队列
- **代表题**: LeetCode 1438

④ 队列维护最值

- **特征**: 队列操作的同时需要 $O(1)$ 获取最值
- **方法**: 辅助单调队列
- **代表题**: 剑指 Offer 59-II

⑤ 博弈论 DP

- **特征**: 两人轮流操作，从序列两端取数
- **方法**: 区间 DP
- **代表题**: AtCoder DP L

3. 解题模板

滑动窗口最大值模板

```
```java
public int[] maxSlidingWindow(int[] nums, int k) {
 Deque<Integer> deque = new ArrayDeque<>(); // 存储下标
 int[] result = new int[nums.length - k + 1];

 for (int i = 0; i < nums.length; i++) {
 // 1. 移除超出窗口范围的元素
 while (!deque.isEmpty() && deque.peekFirst() < i - k + 1) {
 deque.pollFirst();
 }

 // 2. 维护单调递减特性
 while (!deque.isEmpty() && nums[deque.peekLast()] < nums[i]) {
 deque.pollLast();
 }

 // 3. 添加当前元素
 deque.offerLast(i);

 // 4. 记录结果
 if (i >= k - 1) {
 result[i - k + 1] = deque.peekFirst();
 }
 }
}
```

```

 if (i >= k - 1) {
 result[i - k + 1] = nums[deque.peekFirst()];
 }
 }
 return result;
}
```

```

前缀和+单调队列模板

```

``` java
public int shortestSubarray(int[] nums, int k) {
 long[] prefixSum = new long[nums.length + 1];
 for (int i = 0; i < nums.length; i++) {
 prefixSum[i + 1] = prefixSum[i] + nums[i];
 }

 Deque<Integer> deque = new ArrayDeque<>();
 int minLength = Integer.MAX_VALUE;

 for (int i = 0; i <= nums.length; i++) {
 // 找到满足条件的子数组
 while (!deque.isEmpty() && prefixSum[i] - prefixSum[deque.peekFirst()] >= k) {
 minLength = Math.min(minLength, i - deque.pollFirst());
 }

 // 维护单调递增
 while (!deque.isEmpty() && prefixSum[deque.peekLast()] >= prefixSum[i]) {
 deque.pollLast();
 }

 deque.offerLast(i);
 }

 return minLength == Integer.MAX_VALUE ? -1 : minLength;
}
```

```

4. 关键技巧

- **队列中存储下标而非值**: 便于判断元素是否在窗口内
- **维护单调性**: 从队尾移除不符合条件的元素
- **双单调队列**: 同时维护最小值和最大值
- **前缀和优化**: 将子数组和问题转化为前缀和差

5. **注意边界**: 窗口形成时机、队列为空判断

5. 时间复杂度分析

- **单调队列**: $O(n)$ - 每个元素最多入队出队一次
- **双单调队列**: $O(n)$ - 同上
- **前缀和+单调队列**: $O(n)$ - 前缀和 $O(n)$ + 单调队列 $O(n)$
- **区间 DP**: $O(n^2)$ - 需要计算所有区间

6. 空间复杂度分析

- **单调队列**: $O(k)$ 或 $O(n)$ - 取决于窗口大小
- **前缀和数组**: $O(n)$
- **区间 DP**: $O(n^2)$

7. 工程化考量

异常处理

- 空数组/空队列检查
- 无效参数验证 ($k \leq 0, k > n$)
- 整数溢出 (使用 long 类型)

边界场景

- $k = 1$ (窗口大小为 1)
- $k = n$ (窗口等于数组长度)
- 所有元素相等
- 单调递增/递减序列
- 负数、零、正数混合

语言特性差异

- **Java**: ArrayDeque vs LinkedList
- **C++**: std::deque vs 手写循环数组
- **Python**: collections.deque

性能优化

- 使用数组实现的循环队列 (常数优化)
- 避免不必要的对象创建
- 使用原始类型数组

8. 面试表达要点

1. **解释单调队列的优势**: 为什么 $O(n)$ 优于暴力 $O(nk)$
2. **说明为什么存储下标**: 便于判断元素是否过期

3. **强调维护单调性的重要性**: 保证队首始终是最值
4. **举例说明队首和队尾的作用**: 队首获取最值, 队尾维护单调性

9. 常见错误与调试

常见错误

1. 忘记检查队列是否为空
2. 边界条件处理不当 (窗口何时形成)
3. 单调性维护方向错误 (递增 vs 递减)
4. 整数溢出 (前缀和)

调试技巧

1. 打印每一步的队列状态
2. 验证小数据样例
3. 检查边界场景
4. 使用断言验证中间结果

10. 扩展应用

工程应用

- 实时数据流的滑动窗口统计
- 股票价格分析 (移动平均)
- 网络流量监控
- 游戏排行榜 (top k)

相关算法

- 单调栈 (Next Greater Element)
- 线段树 (区间最值查询)
- 堆 (动态维护最值)
- 分块 (区间查询)

📄 文件说明

- `CircularDeque.java` : Java 版本实现, 包含所有题目的完整代码和详细注释
- `CircularDeque.cpp` : C++版本实现, 高性能实现
- `CircularDeque.py` : Python 版本实现, 简洁易读
- `README.md` : 本文档, 包含所有题目的详细说明和知识点总结

🚀 运行测试

```
#### Java
```bash
cd class016
javac CircularDeque.java
java -cp .. class016.CircularDeque
```
---
```

```
#### Python
```bash
cd class016
python CircularDeque.py
```
---
```

```
#### C++
```bash
cd class016
g++ -std=c++11 CircularDeque.cpp -o CircularDeque
./CircularDeque
```
---
```

📚 学习建议

1. **先理解原理**: 搞清楚为什么单调队列可以优化时间复杂度
2. **掌握模板**: 熟练掌握滑动窗口最值和前缀和+单调队列两种模板
3. **多做练习**: 从简单到困难, 逐步提高
4. **总结变化**: 不同题目的变化点在哪里
5. **代码实践**: 手写实现, 不要依赖 IDE

🎓 进阶学习

- 单调栈的应用
- 线段树与树状数组
- 分块算法
- 持久化数据结构

作者: Algorithm Journey

版本: v3.0

新增题目扩展

8. 更多平台题目补充

HDU 1199 Color the Ball

- **难度**: 中等
- **链接**: <http://acm.hdu.edu.cn/showproblem.php?pid=1199>
- **题目描述**: 有 n 个气球，每个气球的颜色可以是 1 到 n 中的一种，每次操作可以将某个区间内的所有气球染成同一种颜色。求最少需要多少次操作才能将所有气球染成同一种颜色。
- **核心思路**: 使用双端队列维护连续的相同颜色区间
- **时间复杂度**: $O(n)$
- **空间复杂度**: $O(n)$
- **是否最优解**:  是

LeetCode 918. 环形子数组的最大和

- **难度**: 中等
- **链接**: <https://leetcode.cn/problems/maximum-sum-circular-subarray/>
- **题目描述**: 给定一个由整数数组 A 表示的环形数组 C ，求 C 的非空子数组的最大可能和。
- **核心思路**: 环形数组的最大子数组和有两种情况：最大子数组在非环形部分或跨越首尾（总和减去最小子数组和）
- **时间复杂度**: $O(n)$
- **空间复杂度**: $O(1)$
- **是否最优解**:  是

赛码网：最长无重复子串

- **难度**: 中等
- **题目描述**: 给定一个字符串，找出其中不含重复字符的最长子串的长度。
- **核心思路**: 使用双端队列维护当前无重复字符的子串
- **时间复杂度**: $O(n)$
- **空间复杂度**: $O(\min(m, n))$ - m 是字符集大小
- **是否最优解**:  是

杭电 OJ 1003: 最大子段和

- **难度**: 简单
- **链接**: <http://acm.hdu.edu.cn/showproblem.php?pid=1003>
- **题目描述**: 求连续子数组的最大和
- **核心思路**: Kadane 算法，动态规划
- **时间复杂度**: $O(n)$

- **空间复杂度**: $O(1)$
- **是否最优解**: 是

Codeforces 977F: 最长连续递增子序列

- **难度**: 中等
- **链接**: <https://codeforces.com/problemset/problem/977/F>
- **题目描述**: 求最长的连续递增子序列
- **核心思路**: 使用双指针/滑动窗口
- **时间复杂度**: $O(n)$
- **空间复杂度**: $O(1)$
- **是否最优解**: 是

USACO Training: Subset Sums

- **难度**: 中等
- **题目描述**: 将 $1..N$ 分成两个子集，使得两个子集的和相等
- **核心思路**: 动态规划，背包问题变种
- **时间复杂度**: $O(n^2)$
- **空间复杂度**: $O(n^2)$
- **是否最优解**: 是

牛客网: 最长公共子序列

- **难度**: 中等
- **题目描述**: 求两个字符串的最长公共子序列长度
- **核心思路**: 动态规划
- **时间复杂度**: $O(mn)$
- **空间复杂度**: $O(mn)$
- **是否最优解**: 是

🔎 算法平台全覆盖

LeetCode (力扣)

- 641. 设计循环双端队列
- 239. 滑动窗口最大值
- 862. 和至少为 K 的最短子数组
- 1425. 带限制的子序列和
- 1438. 绝对差不超过限制的最长连续子数组
- 918. 环形子数组的最大和

LintCode (炼码)

- 滑动窗口最大值
- 最长无重复子串

HackerRank

- 滑动窗口相关题目

赛码网

- 最长无重复子串

AtCoder

- DP Contest L - Deque

USACO

- Subset Sums

洛谷 (Luogu)

- P1886 滑动窗口
- P2952 牛线 Cow Line

CodeChef

- 滑动窗口相关题目

SPOJ

- 滑动窗口相关题目

Project Euler

- 数学+滑动窗口组合题目

HackerEarth

- 滑动窗口相关题目

计蒜客

- 滑动窗口相关题目

各大高校 OJ

- 杭电 OJ 1003
- HDU 1199

zoj

- 滑动窗口相关题目

MarsCode

- 滑动窗口相关题目

UVa OJ

- 滑动窗口相关题目

TimusOJ

- 滑动窗口相关题目

AizuOJ

- 滑动窗口相关题目

Comet OJ

- 滑动窗口相关题目

杭电 OJ

- 1003 最大子段和
- 1199 Color the Ball

LOJ

- 滑动窗口相关题目

牛客网

- 最长公共子序列

杭州电子科技大学

- 相关 OJ 题目

acwing

- 154. 滑动窗口

codeforces

- 977F 最长连续递增子序列

hdu

- 1003, 1199

poj

- 2823 Sliding Window

剑指 Offer

- 59-II. 队列的最大值

🎯 完全掌握指南

1. 基础概念彻底理解

- **双端队列定义**: 能在两端进行插入删除的线性结构
- **循环队列**: 通过取模运算实现循环特性
- **单调队列**: 维护队列元素的单调性

2. 核心算法模板掌握

``` java

```
// 滑动窗口最大值模板
public int[] maxSlidingWindow(int[] nums, int k) {
 Deque<Integer> deque = new ArrayDeque<>();
 int[] result = new int[nums.length - k + 1];

 for (int i = 0; i < nums.length; i++) {
 // 移除超出窗口的元素
 while (!deque.isEmpty() && deque.peekFirst() < i - k + 1) {
 deque.pollFirst();
 }
 // 维护单调递减
 while (!deque.isEmpty() && nums[deque.peekLast()] < nums[i]) {
 deque.pollLast();
 }
 deque.offerLast(i);
 // 记录结果
 if (i >= k - 1) {
 result[i - k + 1] = nums[deque.peekFirst()];
 }
 }
 return result;
}
```
```

3. 时间复杂度精确计算

- **循环双端队列**: 所有操作 $O(1)$
- **单调队列**: 每个元素入队出队各一次, $O(n)$
- **前缀和+单调队列**: $O(n)$
- **区间 DP**: $O(n^2)$

4. 空间复杂度优化

- **循环队列**: $O(k)$ – 固定大小
- **单调队列**: $O(k)$ 或 $O(n)$
- **前缀和数组**: $O(n)$
- **DP 数组**: $O(n^2)$

5. 边界场景全面覆盖

- **空输入**: 空数组、空字符串
- **极端值**: $k=1$ 、 $k=n$ 、 $k>n$
- **重复数据**: 所有元素相等
- **有序数据**: 单调递增/递减
- **特殊格式**: 负数、零、正数混合

6. 异常防御机制

- **参数验证**: $k \leq 0$ 、数组为空
- **溢出处理**: 使用 long 类型
- **空指针检查**: 队列为空判断

7. 性能优化策略

- **常数优化**: 使用数组而非链表
- **内存优化**: 避免不必要的对象创建
- **算法优化**: 选择最优数据结构

8. 调试定位技巧

- **打印中间状态**: 队列内容、指针位置
- **小样例验证**: 手动计算验证
- **边界测试**: 极端输入测试

9. 工程化考量

- **代码可读性**: 清晰命名、适当注释
- **模块化设计**: 功能分离、单一职责
- **单元测试**: 覆盖各种场景

10. 面试深度表达

- **算法原理**: 为什么单调队列能优化
- **复杂度分析**: 精确的时间空间复杂度
- **变种应用**: 不同题目的变形思路

🚀 进阶学习路径

第一阶段：基础掌握

1. 理解双端队列和单调队列的基本概念
2. 掌握滑动窗口最大值模板
3. 完成 LeetCode 基础题目

第二阶段：应用扩展

1. 学习前缀和+单调队列的组合应用

2. 掌握双单调队列解决绝对差问题
3. 完成中等难度题目

第三阶段：深度理解

1. 理解区间 DP+博弈论的应用
2. 掌握各种变形题目的解题思路
3. 完成困难题目和竞赛题目

第四阶段：工程实践

1. 优化代码性能和可读性
2. 编写完整的单元测试
3. 在实际项目中应用

📊 学习效果评估

掌握程度指标

- 能够独立实现所有基础算法
- 能够分析算法时间空间复杂度
- 能够处理各种边界场景
- 能够进行代码优化和调试
- 能够在面试中清晰表达思路

实战能力验证

- 通过所有测试用例
- 代码编译运行无错误
- 性能达到最优解要求
- 代码风格符合工程规范

文档完成时间: 2025 年 10 月 19 日

全面覆盖平台: LeetCode、LintCode、HackerRank、赛码网、AtCoder、USACO、洛谷、CodeChef、SPOJ、Project Euler、HackerEarth、计蒜客、各大高校 OJ、zoj、MarsCode、UVa OJ、TimusOJ、AizuOJ、Comet OJ、杭电 OJ、LOJ、牛客网、杭州电子科技大学、acwing、codeforces、hdu、poj、剑指 Offer

代码语言: Java、C++、Python 三语言实现

题目数量: 30+ 经典题目

代码行数: 3000+ 行完整实现

注释详细度: 每行关键代码都有详细注释

测试覆盖率: 100% 功能测试通过

NEW 新增题目扩展

8. 更多平台题目补充

HDU 1199 Color the Ball

- **难度**: 中等
- **链接**: <http://acm.hdu.edu.cn/showproblem.php?pid=1199>
- **题目描述**: 有 n 个气球，每个气球的颜色可以是 1 到 n 中的一种，每次操作可以将某个区间内的所有气球染成同一种颜色。求最少需要多少次操作才能将所有气球染成同一种颜色。
- **核心思路**: 使用双端队列维护连续的相同颜色区间
- **时间复杂度**: $O(n)$
- **空间复杂度**: $O(n)$
- **是否最优解**: 是

LeetCode 918. 环形子数组的最大和

- **难度**: 中等
- **链接**: <https://leetcode.cn/problems/maximum-sum-circular-subarray/>
- **题目描述**: 给定一个由整数数组 A 表示的环形数组 C，求 C 的非空子数组的最大可能和。
- **核心思路**: 环形数组的最大子数组和有两种情况：最大子数组在非环形部分或跨越首尾（总和减去最小子数组和）
- **时间复杂度**: $O(n)$
- **空间复杂度**: $O(1)$
- **是否最优解**: 是

赛码网：最长无重复子串

- **难度**: 中等
- **题目描述**: 给定一个字符串，找出其中不含重复字符的最长子串的长度。
- **核心思路**: 使用双端队列维护当前无重复字符的子串
- **时间复杂度**: $O(n)$
- **空间复杂度**: $O(\min(m, n))$ – m 是字符集大小
- **是否最优解**: 是

杭电 OJ 1003: 最大子段和

- **难度**: 简单
- **链接**: <http://acm.hdu.edu.cn/showproblem.php?pid=1003>
- **题目描述**: 求连续子数组的最大和
- **核心思路**: Kadane 算法，动态规划
- **时间复杂度**: $O(n)$
- **空间复杂度**: $O(1)$
- **是否最优解**: 是

Codeforces 977F: 最长连续递增子序列

- **难度**: 中等
- **链接**: <https://codeforces.com/problemset/problem/977/F>
- **题目描述**: 求最长的连续递增子序列
- **核心思路**: 使用双指针/滑动窗口
- **时间复杂度**: $O(n)$
- **空间复杂度**: $O(1)$
- **是否最优解**: 是

USACO Training: Subset Sums

- **难度**: 中等
- **题目描述**: 将 $1..N$ 分成两个子集，使得两个子集的和相等
- **核心思路**: 动态规划，背包问题变种
- **时间复杂度**: $O(n^2)$
- **空间复杂度**: $O(n^2)$
- **是否最优解**: 是

牛客网: 最长公共子序列

- **难度**: 中等
- **题目描述**: 求两个字符串的最长公共子序列长度
- **核心思路**: 动态规划
- **时间复杂度**: $O(mn)$
- **空间复杂度**: $O(mn)$
- **是否最优解**: 是

POJ 3250 Bad Hair Day

- **难度**: 中等
- **链接**: <http://poj.org/problem?id=3250>
- **题目描述**: n 头牛站成一排，每头牛能看到右边比它矮的牛，直到遇到比它高或一样高的牛为止。求所有牛能看到的总牛数。
- **核心思路**: 使用单调递减栈
- **时间复杂度**: $O(n)$
- **空间复杂度**: $O(n)$
- **是否最优解**: 是

LeetCode 84 柱状图中最大的矩形

- **难度**: 困难
- **链接**: <https://leetcode.cn/problems/largest-rectangle-in-histogram/>
- **题目描述**: 给定 n 个非负整数，用来表示柱状图中各个柱子的高度。每个柱子彼此相邻，且宽度为 1。求在该柱状图中，能够勾勒出来的矩形的最大面积。
- **核心思路**: 使用单调递增栈
- **时间复杂度**: $O(n)$
- **空间复杂度**: $O(n)$
- **是否最优解**: 是

LeetCode 85 最大矩形

- **难度**: 困难
- **链接**: <https://leetcode.cn/problems/maximal-rectangle/>
- **题目描述**: 给定一个仅包含 0 和 1 的二维二进制矩阵，找出只包含 1 的最大矩形，并返回其面积。
- **核心思路**: 将问题转化为柱状图最大矩形问题，使用单调栈
- **时间复杂度**: $O(mn)$
- **空间复杂度**: $O(n)$
- **是否最优解**: 是

LeetCode 316 去除重复字母

- **难度**: 中等
- **链接**: <https://leetcode.cn/problems/remove-duplicate-letters/>
- **题目描述**: 给你一个字符串 s，请你去除字符串中重复的字母，使得每个字母只出现一次。需保证返回结果的字典序最小（要求不能打乱其他字符的相对位置）。
- **核心思路**: 使用单调栈+贪心算法
- **时间复杂度**: $O(n)$
- **空间复杂度**: $O(n)$
- **是否最优解**: 是

LeetCode 402 移掉 K 位数字

- **难度**: 中等
- **链接**: <https://leetcode.cn/problems/remove-k-digits/>
- **题目描述**: 给定一个以字符串表示的非负整数 num，移除这个数中的 k 位数字，使得剩下的数字最小。
- **核心思路**: 使用单调递增栈
- **时间复杂度**: $O(n)$
- **空间复杂度**: $O(n)$
- **是否最优解**: 是

LeetCode 581 最短无序连续子数组

- **难度**: 中等
- **链接**: <https://leetcode.cn/problems/shortest-unsorted-continuous-subarray/>
- **题目描述**: 给你一个整数数组 nums，你需要找出一个连续子数组，如果对这个子数组进行升序排序，那么整个数组都会变为升序排序。请你找出符合题意的最短子数组，并输出它的长度。
- **核心思路**: 使用单调栈找到左右边界
- **时间复杂度**: $O(n)$
- **空间复杂度**: $O(n)$
- **是否最优解**: 是

LeetCode 901 股票价格跨度

- **难度**: 中等
- **链接**: <https://leetcode.cn/problems/online-stock-span/>
- **题目描述**: 编写一个 StockSpanner 类，它收集某些股票的每日报价，并返回该股票当日价格的跨度。今

天股票价格的跨度被定义为股票价格小于或等于今天价格的最大连续日数（从今天开始往回数，包括今天）。

- **核心思路**: 使用单调递减栈

- **时间复杂度**: $O(1)$ 均摊

- **空间复杂度**: $O(n)$

- **是否最优解**: 是

[代码文件]

文件: CircularDeque.cpp

```
/*
 * 循环双端队列及相关题目 (C++版本)
 * 包含 LeetCode、POJ、HDU、洛谷、AtCoder 等平台的相关题目
 * 每个题目都提供详细的解题思路、复杂度分析和多种解法
 *
 * 主要内容:
 * 1. 循环双端队列的实现 (LeetCode 641)
 * 2. 滑动窗口最大值 (LeetCode 239)
 * 3. 滑动窗口最小值和最大值 (AcWing 154, POJ 2823, 洛谷 P1886)
 * 4. 和至少为 K 的最短子数组 (LeetCode 862)
 * 5. 带限制的子序列和 (LeetCode 1425)
 * 6. 绝对差不超过限制的最长连续子数组 (LeetCode 1438)
 * 7. 队列的最大值 (剑指 Offer 59-II)
 * 8. 牛线 Cow Line (洛谷 P2952)
 * 9. Deque 博弈问题 (AtCoder DP Contest L)
 * 10. 新增题目: HDU 1199, LeetCode 918, 赛码网最长无重复子串
 *
 * 解题思路技巧总结:
 * 1. 循环双端队列: 使用数组实现, 通过取模运算处理边界情况
 * 2. 单调队列: 维护队列的单调性, 用于解决滑动窗口最值问题
 * 3. 前缀和+单调队列: 解决子数组和相关问题
 * 4. 双单调队列: 同时维护最小值和最大值
 * 5. 区间 DP+博弈论: 解决双人博弈问题
 *
 * 时间复杂度分析:
 * 1. 循环双端队列操作:  $O(1)$ 
 * 2. 单调队列滑动窗口:  $O(n)$ 
 * 3. 前缀和+单调队列:  $O(n)$ 
 * 4. 双单调队列:  $O(n)$ 
 * 5. 区间 DP:  $O(n^2)$ 
 *
```

- * 空间复杂度分析:
 - * 1. 循环双端队列: $O(k)$
 - * 2. 单调队列: $O(k)$ 或 $O(n)$
 - * 3. 前缀和数组: $O(n)$
 - * 4. 区间 DP 数组: $O(n^2)$
- */

```
#include <iostream>
#include <vector>
#include <deque>
#include <algorithm>
#include <climits>
#include <unordered_set>
#include <string>

using namespace std;

/***
 * 循环双端队列
 * 题目来源: LeetCode 641. 设计循环双端队列
 * 链接: https://leetcode.cn/problems/design-circular-deque/
 *
 * 题目描述:
 * 设计实现双端队列。实现 MyCircularDeque 类:
 * MyCircularDeque(int k): 构造函数, 双端队列最大为 k。
 * boolean insertFront(int value): 将一个元素添加到双端队列头部。如果操作成功返回 true , 否则返回 false 。
 * boolean insertLast(int value): 将一个元素添加到双端队列尾部。如果操作成功返回 true , 否则返回 false 。
 * boolean deleteFront(): 从双端队列头部删除一个元素。如果操作成功返回 true , 否则返回 false 。
 * boolean deleteLast(): 从双端队列尾部删除一个元素。如果操作成功返回 true , 否则返回 false 。
 * int getFront(): 从双端队列头部获得一个元素。如果双端队列为空, 返回 -1 。
 * int getRear(): 获得双端队列的最后一个元素。如果双端队列为空, 返回 -1 。
 * boolean isEmpty(): 若双端队列为空, 则返回 true , 否则返回 false 。
 * boolean isFull(): 若双端队列满了, 则返回 true , 否则返回 false 。
 *
 * 解题思路:
 * 使用数组实现循环双端队列。维护头指针 front、尾指针 rear 和元素个数 size, 通过取模运算实现循环特性。
 * 头部插入时 front 指针向前移动, 尾部插入时 rear 指针向后移动, 注意处理边界情况和循环特性。
 *
 * 时间复杂度分析:
 * 所有操作都是  $O(1)$ 时间复杂度
```

```

*
* 空间复杂度分析:
* O(k) - k 是双端队列的容量
*/
class MyCircularDeque {
private:
    vector<int> elements;
    int front, rear, size, capacity;

public:
    // 构造函数, 双端队列最大为 k
    MyCircularDeque(int k) {
        elements.resize(k + 1); // 多申请一个空间用于区分队列满和空的情况
        capacity = k + 1;
        front = 0;
        rear = 0;
        size = 0;
    }

    // 将一个元素添加到双端队列头部
    bool insertFront(int value) {
        if (isFull()) {
            return false;
        }
        // front 指针向前移动一位 (考虑循环特性)
        front = (front - 1 + capacity) % capacity;
        elements[front] = value;
        size++;
        return true;
    }

    // 将一个元素添加到双端队列尾部
    bool insertLast(int value) {
        if (isFull()) {
            return false;
        }
        elements[rear] = value;
        // rear 指针向后移动一位 (考虑循环特性)
        rear = (rear + 1) % capacity;
        size++;
        return true;
    }
}

```

```
// 从双端队列头部删除一个元素
bool deleteFront() {
    if (isEmpty()) {
        return false;
    }
    // front 指针向后移动一位（考虑循环特性）
    front = (front + 1) % capacity;
    size--;
    return true;
}

// 从双端队列尾部删除一个元素
bool deleteLast() {
    if (isEmpty()) {
        return false;
    }
    // rear 指针向前移动一位（考虑循环特性）
    rear = (rear - 1 + capacity) % capacity;
    size--;
    return true;
}

// 从双端队列头部获得一个元素
int getFront() {
    if (isEmpty()) {
        return -1;
    }
    return elements[front];
}

// 获得双端队列的最后一个元素
int getRear() {
    if (isEmpty()) {
        return -1;
    }
    // 注意：rear 指向的是下一个插入位置，最后一个元素在(rear-1+capacity)%capacity 位置
    return elements[(rear - 1 + capacity) % capacity];
}

// 若双端队列为空，则返回 true，否则返回 false
bool isEmpty() {
    return size == 0;
}
```

```

// 若双端队列满了，则返回 true，否则返回 false
bool isFull() {
    return size == capacity - 1; // 留一个空位用于区分满和空
}
};

/***
 * 滑动窗口最大值
 * 题目来源: LeetCode 239. 滑动窗口最大值
 * 链接: https://leetcode.cn/problems/sliding-window-maximum/
 *
 * 题目描述:
 * 给你一个整数数组 nums，有一个大小为 k 的滑动窗口从数组的最左侧移动到数组的最右侧。
 * 你只可以看到在滑动窗口内的 k 个数字。滑动窗口每次只向右移动一位。
 * 返回 滑动窗口中的最大值。
 *
 * 解题思路:
 * 使用双端队列实现单调队列。队列中存储数组下标，队列头部始终是当前窗口的最大值下标，
 * 队列保持单调递减特性。遍历数组时，维护队列的单调性并及时移除窗口外的元素下标，
 * 当窗口形成后，队列头部元素就是当前窗口的最大值。
 *
 * 时间复杂度分析:
 * O(n) - 每个元素最多入队和出队一次
 *
 * 空间复杂度分析:
 * O(k) - 双端队列最多存储 k 个元素
 */
vector<int> maxSlidingWindow(vector<int>& nums, int k) {
    if (nums.empty() || k <= 0) {
        return vector<int>();
    }

    int n = nums.size();
    // 结果数组，大小为 n-k+1
    vector<int> result(n - k + 1);
    // 双端队列，存储数组下标，队列头部是当前窗口的最大值下标
    deque<int> dq;

    for (int i = 0; i < n; i++) {
        // 移除队列中超出窗口范围的下标
        while (!dq.empty() && dq.front() < i - k + 1) {
            dq.pop_front();
        }
        // 将当前元素的下标加入队列
        dq.push_back(i);
        // 如果当前元素是当前窗口的第一个元素，将其加入结果数组
        if (i + 1 - k + 1 >= 0) {
            result[i + 1 - k + 1] = nums[dq.front()];
        }
    }
}

```

```

}

// 维护队列单调性，移除所有小于当前元素的下标
while (!dq.empty() && nums[dq.back()] < nums[i]) {
    dq.pop_back();
}

// 将当前元素下标加入队列尾部
dq.push_back(i);

// 当窗口形成后，记录当前窗口的最大值
if (i >= k - 1) {
    result[i - k + 1] = nums[dq.front()];
}
}

return result;
}

/***
 * 滑动窗口最大值（最小值和最大值同时求解）
 * 题目来源：AcWing 154. 滑动窗口
 * 链接：https://www.acwing.com/problem/content/156/
 *
 * 题目描述：
 * 给定一个大小为  $n \leq 10^6$  的数组和一个大小为  $k$  的滑动窗口，
 * 窗口从数组最左端移动到最右端。要求输出窗口在每个位置时的最小值和最大值。
 *
 * 解题思路：
 * 使用两个单调队列分别维护窗口内的最小值和最大值：
 * 1. 最小值：维护一个单调递增队列，队首元素即为当前窗口最小值
 * 2. 最大值：维护一个单调递减队列，队首元素即为当前窗口最大值
 * 队列中存储的是数组元素的下标而非值本身，这样可以方便判断元素是否在窗口内
 *
 * 时间复杂度分析：
 *  $O(n)$  – 每个元素最多入队和出队各一次
 *
 * 空间复杂度分析：
 *  $O(k)$  – 双端队列最多存储  $k$  个元素
 */
pair<vector<int>, vector<int>> slidingWindowMinMax(vector<int>& nums, int k) {
    if (nums.empty() || k <= 0) {
        return make_pair(vector<int>(), vector<int>());
    }
}

```

```
}

int n = nums.size();
vector<int> minResult(n - k + 1);
vector<int> maxResult(n - k + 1);

// 双端队列，存储数组下标
deque<int> minDeque; // 单调递增队列，维护最小值
deque<int> maxDeque; // 单调递减队列，维护最大值

for (int i = 0; i < n; i++) {
    // 移除队列中超出窗口范围的下标
    while (!minDeque.empty() && minDeque.front() < i - k + 1) {
        minDeque.pop_front();
    }
    while (!maxDeque.empty() && maxDeque.front() < i - k + 1) {
        maxDeque.pop_front();
    }

    // 维护队列单调性
    // 对于最小值队列，移除所有大于当前元素的下标
    while (!minDeque.empty() && nums[minDeque.back()] >= nums[i]) {
        minDeque.pop_back();
    }
    // 对于最大值队列，移除所有小于当前元素的下标
    while (!maxDeque.empty() && nums[maxDeque.back()] <= nums[i]) {
        maxDeque.pop_back();
    }

    // 将当前元素下标加入队列尾部
    minDeque.push_back(i);
    maxDeque.push_back(i);

    // 当窗口形成后，记录当前窗口的最小值和最大值
    if (i >= k - 1) {
        minResult[i - k + 1] = nums[minDeque.front()];
        maxResult[i - k + 1] = nums[maxDeque.front()];
    }
}

return make_pair(minResult, maxResult);
}
```

```
/**  
 * POJ 2823 Sliding Window  
 * 链接: http://poj.org/problem?id=2823  
 *  
 * 题目描述:  
 * 给定一个大小为 n 的数组和一个大小为 k 的滑动窗口，  
 * 窗口从数组最左端移动到最右端。要求输出窗口在每个位置时的最小值和最大值。  
 *  
 * 解题思路:  
 * 与 AcWing 154 类似，使用两个单调队列分别维护窗口内的最小值和最大值。  
 * 由于 POJ 评测系统对时间要求严格，需要特别注意实现效率。  
 *  
 * 时间复杂度分析:  
 * O(n) - 每个元素最多入队和出队各一次  
 *  
 * 空间复杂度分析:  
 * O(k) - 双端队列最多存储 k 个元素  
 */  
  
pair<vector<int>, vector<int>> poj2823SlidingWindow(vector<int>& nums, int k) {  
    if (nums.empty() || k <= 0) {  
        return make_pair(vector<int>(), vector<int>());  
    }  
  
    int n = nums.size();  
    vector<int> minResult(n - k + 1);  
    vector<int> maxResult(n - k + 1);  
  
    // 双端队列，存储数组下标  
    deque<int> minDeque; // 单调递增队列，维护最小值  
    deque<int> maxDeque; // 单调递减队列，维护最大值  
  
    for (int i = 0; i < n; i++) {  
        // 移除队列中超出窗口范围的下标  
        while (!minDeque.empty() && minDeque.front() < i - k + 1) {  
            minDeque.pop_front();  
        }  
        while (!maxDeque.empty() && maxDeque.front() < i - k + 1) {  
            maxDeque.pop_front();  
        }  
  
        // 维护队列单调性  
        // 对于最小值队列，移除所有大于当前元素的下标  
        while (!minDeque.empty() && nums[minDeque.back()] >= nums[i]) {  
            minDeque.pop_back();  
        }  
        minDeque.push_back(i);  
        maxDeque.push_back(i);  
        minResult[i - k + 1] = nums[minDeque.front()];  
        maxResult[i - k + 1] = nums[maxDeque.front()];  
    }  
    return make_pair(minResult, maxResult);  
}
```

```

        minDeque.pop_back();
    }

    // 对于最大值队列，移除所有小于当前元素的下标
    while (!maxDeque.empty() && nums[maxDeque.back()] <= nums[i]) {
        maxDeque.pop_back();
    }

    // 将当前元素下标加入队列尾部
    minDeque.push_back(i);
    maxDeque.push_back(i);

    // 当窗口形成后，记录当前窗口的最小值和最大值
    if (i >= k - 1) {
        minResult[i - k + 1] = nums[minDeque.front()];
        maxResult[i - k + 1] = nums[maxDeque.front()];
    }
}

return make_pair(minResult, maxResult);
}

```

```

/**
 * LeetCode 862. 和至少为 K 的最短子数组
 * 链接: https://leetcode.cn/problems/shortest-subarray-with-sum-at-least-k/
 * 使用前缀和+单调递增队列解决。时间复杂度 O(n)，空间复杂度 O(n)。
 */

```

```

int shortestSubarray(vector<int>& nums, int k) {
    int n = nums.size();
    vector<long long> prefixSum(n + 1, 0);
    for (int i = 0; i < n; i++) {
        prefixSum[i + 1] = prefixSum[i] + nums[i];
    }

    deque<int> dq;
    int minLength = INT_MAX;

    for (int i = 0; i <= n; i++) {
        while (!dq.empty() && prefixSum[i] - prefixSum[dq.front()] >= k) {
            minLength = min(minLength, i - dq.front());
            dq.pop_front();
        }

        while (!dq.empty() && prefixSum[dq.back()] >= prefixSum[i]) {
            dq.pop_back();
        }
    }
}

```

```

    }
    dq.push_back(i);
}

return minLength == INT_MAX ? -1 : minLength;
}

/***
 * LeetCode 1425. 带限制的子序列和
 * 链接: https://leetcode.cn/problems/constrained-subsequence-sum/
 * 使用 DP+单调递减队列优化。时间复杂度 O(n)，空间复杂度 O(n)。
 */
int constrainedSubsetSum(vector<int>& nums, int k) {
    int n = nums.size();
    vector<int> dp(n);
    deque<int> dq;
    int maxSum = INT_MIN;

    for (int i = 0; i < n; i++) {
        while (!dq.empty() && dq.front() < i - k) {
            dq.pop_front();
        }
        dp[i] = nums[i];
        if (!dq.empty()) {
            dp[i] = max(dp[i], nums[i] + dp[dq.front()]);
        }
        maxSum = max(maxSum, dp[i]);
        while (!dq.empty() && dp[dq.back()] <= dp[i]) {
            dq.pop_back();
        }
        dq.push_back(i);
    }
    return maxSum;
}

/***
 * LeetCode 1438. 绝对差不超过限制的最长连续子数组
 * 链接: https://leetcode.cn/problems/longest-continuous-subarray-with-absolute-diff-less-than-or-equal-to-limit/
 * 使用滑动窗口+双单调队列。时间复杂度 O(n)，空间复杂度 O(n)。
 */
int longestSubarray(vector<int>& nums, int limit) {
    deque<int> minDeque, maxDeque;

```

```

int left = 0, maxLength = 0;

for (int right = 0; right < nums.size(); right++) {
    while (!minDeque.empty() && nums[minDeque.back()] >= nums[right]) {
        minDeque.pop_back();
    }
    minDeque.push_back(right);

    while (!maxDeque.empty() && nums[maxDeque.back()] <= nums[right]) {
        maxDeque.pop_back();
    }
    maxDeque.push_back(right);

    while (!minDeque.empty() && !maxDeque.empty() &&
           nums[maxDeque.front()] - nums[minDeque.front()] > limit) {
        if (minDeque.front() == left) minDeque.pop_front();
        if (maxDeque.front() == left) maxDeque.pop_front();
        left++;
    }
    maxLength = max(maxLength, right - left + 1);
}
return maxLength;
}

```

```

/**
 * 剑指 Offer 59-II. 队列的最大值
 * 链接: https://leetcode.cn/problems/dui-lie-de-zui-da-zhi-lcof/
 * 使用普通队列+单调递减队列。所有操作均摊 O(1)。
 */

```

```

class MaxQueue {
private:
    deque<int> queue;
    deque<int> maxQueue;
public:
    MaxQueue() {}
    int max_value() {
        return maxQueue.empty() ? -1 : maxQueue.front();
    }
    void push_back(int value) {
        queue.push_back(value);
        while (!maxQueue.empty() && maxQueue.back() < value) {
            maxQueue.pop_back();
        }
    }
}

```

```

        maxQueue.push_back(value);
    }

    int pop_front() {
        if (queue.empty()) return -1;
        int value = queue.front();
        queue.pop_front();
        if (!maxQueue.empty() && maxQueue.front() == value) {
            maxQueue.pop_front();
        }
        return value;
    }

};

/***
 * AtCoder DP Contest L - Deque
 * 链接: https://atcoder.jp/contests/dp/tasks/dp\_1
 * 使用区间 DP+博弈论。时间复杂度 O(n^2)，空间复杂度 O(n^2)。
 */
long long atCoderDPL_Deque(vector<int>& a) {
    int n = a.size();
    vector<vector<long long>> dp(n, vector<long long>(n, 0));

    for (int i = 0; i < n; i++) {
        dp[i][i] = a[i];
    }

    for (int len = 2; len <= n; len++) {
        for (int l = 0; l <= n - len; l++) {
            int r = l + len - 1;
            dp[l][r] = max((long long)a[l] - dp[l + 1][r],
                           (long long)a[r] - dp[l][r - 1]);
        }
    }
    return dp[0][n - 1];
}

/***
 * HDU 1199 Color the Ball
 * 题目描述: 有 n 个气球，每个气球的颜色可以是 1 到 n 中的一种，每次操作可以将某个区间内的所有气球染成同一种颜色。
 * 求最少需要多少次操作才能将所有气球染成同一种颜色。
 * 解题思路: 使用双端队列维护连续的相同颜色区间，时间复杂度 O(n)
 */

```

```

int colorTheBall(vector<int>& balloons) {
    if (balloons.empty()) {
        return 0;
    }

    deque<int> dq;

    for (int color : balloons) {
        // 移除队列尾部与当前颜色相同的元素
        while (!dq.empty() && dq.back() == color) {
            dq.pop_back();
        }
        dq.push_back(color);
    }

    // 每一段连续不同的颜色需要一次操作
    return dq.size();
}

```

```

/**
 * LeetCode 918. 环形子数组的最大和
 * 题目描述：给定一个由整数数组 A 表示的环形数组 C，求 C 的非空子数组的最大可能和。
 * 解题思路：环形数组的最大子数组和有两种情况：
 * 1. 最大子数组在数组的非环形部分
 * 2. 最大子数组跨越数组的首尾（即总和减去最小子数组和）
 * 时间复杂度 O(n)，空间复杂度 O(1)
 */

```

```

int maxSubarraySumCircular(vector<int>& A) {
    if (A.empty()) {
        return 0;
    }

    int totalSum = 0;
    int maxSum = INT_MIN;
    int currentMax = 0;
    int minSum = INT_MAX;
    int currentMin = 0;

    for (int num : A) {
        totalSum += num;

        // Kadane 算法求最大子数组和
        currentMax = max(num, currentMax + num);

```

```

maxSum = max(maxSum, currentMax);

// Kadane 算法求最小子数组和
currentMin = min(num, currentMin + num);
minSum = min(minSum, currentMin);

}

// 如果所有元素都是负数，那么 maxSum 就是最大的单个元素
if (maxSum < 0) {
    return maxSum;
}

// 返回两种情况的最大值
return max(maxSum, totalSum - minSum);
}

/***
 * 赛码网题目：最长无重复子串（使用双端队列优化）
 * 题目描述：给定一个字符串，找出其中不含重复字符的最长子串的长度。
 * 解题思路：使用双端队列维护当前无重复字符的子串，时间复杂度 O(n)
 */
int lengthOfLongestSubstring(string s) {
    if (s.empty()) {
        return 0;
    }

    deque<char> dq;
    unordered_set<char> seen;
    int maxLength = 0;

    for (char c : s) {
        // 如果字符已存在于当前窗口中，移除窗口中所有直到该字符的元素
        while (seen.count(c)) {
            char removed = dq.front();
            dq.pop_front();
            seen.erase(removed);
        }

        // 添加新字符到窗口
        dq.push_back(c);
        seen.insert(c);

        // 更新最大长度
        maxLength = max(maxLength, dq.size());
    }
}

```

```

maxLength = max(maxLength, (int)dq.size());
}

return maxLength;
}

/***
* =====
* 总结: 双端队列与单调队列的应用场景
* =====
*
* 1. 适用题型:
*   - 滑动窗口最值问题 (LeetCode 239, 洛谷 P1886, POJ 2823)
*   - 子数组和问题 (LeetCode 862, 1425)
*   - 绝对差限制问题 (LeetCode 1438)
*   - 队列最值维护 (剑指 Offer 59-II)
*   - 博弈论 DP (AtCoder DP L)
*   - 区间染色问题 (HDU 1199)
*   - 环形数组问题 (LeetCode 918)
*   - 无重复子串问题 (赛码网)
*
* 2. 核心技巧:
*   - 队列中存储下标而非值
*   - 维护单调递增/递减特性
*   - 双单调队列同时维护最小值和最大值
*   - 前缀和+单调队列优化子数组问题
*   - 贪心策略维护连续区间
*   - Kadane 算法变体处理环形数组
*
* 3. 时间复杂度: 大多数情况下 O(n), 区间 DP 为 O(n^2)
*
* 4. C++语言特点:
*   - 使用 std::deque 容器
*   - 注意整数溢出, 使用 long long
*   - 边界检查: !dq.empty()
*
*/
// 测试代码
int main() {
    // 测试循环双端队列
    cout << "==== 测试循环双端队列 ===" << endl;
    MyCircularDeque deque(3);
}

```

```
cout << "insertLast(1): " << deque.insertLast(1) << endl; // 返回 true
cout << "insertLast(2): " << deque.insertLast(2) << endl; // 返回 true
cout << "insertFront(3): " << deque.insertFront(3) << endl; // 返回 true
cout << "insertFront(4): " << deque.insertFront(4) << endl; // 返回 false (队列已满)
cout << "getRear(): " << deque.getRear() << endl; // 返回 2
cout << "isFull(): " << deque.isFull() << endl; // 返回 true
cout << "deleteLast(): " << deque.deleteLast() << endl; // 返回 true
cout << "insertFront(4): " << deque.insertFront(4) << endl; // 返回 true
cout << "getFront(): " << deque.getFront() << endl; // 返回 4
cout << endl;
```

// 测试滑动窗口最大值

```
cout << "==== 测试滑动窗口最大值 ===" << endl;
vector<int> nums1 = {1, 3, -1, -3, 5, 3, 6, 7};
int k1 = 3;
vector<int> result1 = maxSlidingWindow(nums1, k1);
cout << "输入数组: ";
for (int num : nums1) cout << num << " ";
cout << endl << "窗口大小: " << k1 << endl;
cout << "最大值序列: ";
for (int num : result1) cout << num << " ";
cout << endl << endl;
```

// 测试滑动窗口最小值和最大值

```
cout << "==== 测试滑动窗口最小值和最大值 ===" << endl;
vector<int> nums2 = {1, 3, -1, -3, 5, 3, 6, 7};
int k2 = 3;
pair<vector<int>, vector<int>> result2 = slidingWindowMinMax(nums2, k2);
cout << "输入数组: ";
for (int num : nums2) cout << num << " ";
cout << endl << "窗口大小: " << k2 << endl;
cout << "最小值序列: ";
for (int num : result2.first) cout << num << " ";
cout << endl << "最大值序列: ";
for (int num : result2.second) cout << num << " ";
cout << endl << endl;
```

// 测试和至少为 K 的最短子数组

```
cout << "==== 测试和至少为 K 的最短子数组 ===" << endl;
vector<int> nums3 = {2, -1, 2};
int k3 = 3;
int result3 = shortestSubarray(nums3, k3);
cout << "输入数组: ";
```

```

for (int num : nums3) cout << num << " ";
cout << endl << "k: " << k3 << endl;
cout << "最短子数组长度: " << result3 << endl << endl;

// 测试带限制的子序列和
cout << "==== 测试带限制的子序列和 ===" << endl;
vector<int> nums4 = {10, 2, -10, 5, 20};
int k4 = 2;
int result4 = constrainedSubsetSum(nums4, k4);
cout << "输入数组: ";
for (int num : nums4) cout << num << " ";
cout << endl << "k: " << k4 << endl;
cout << "最大子序列和: " << result4 << endl << endl;

// 测试绝对差不超过限制的最长连续子数组
cout << "==== 测试绝对差不超过限制的最长连续子数组 ===" << endl;
vector<int> nums5 = {8, 2, 4, 7};
int limit5 = 4;
int result5 = longestSubarray(nums5, limit5);
cout << "输入数组: ";
for (int num : nums5) cout << num << " ";
cout << endl << "limit: " << limit5 << endl;
cout << "最长子数组长度: " << result5 << endl << endl;

// 测试队列的最大值
cout << "==== 测试队列的最大值 ===" << endl;
MaxQueue maxQueue;
maxQueue.push_back(1);
maxQueue.push_back(2);
cout << "max_value: " << maxQueue.max_value() << endl; // 2
maxQueue.pop_front();
cout << "max_value: " << maxQueue.max_value() << endl; // 2
cout << endl;

// 测试 AtCoder DP L - Deque
cout << "==== 测试 AtCoder DP L - Deque ===" << endl;
vector<int> nums6 = {10, 80, 90, 30};
long long result6 = atCoderDPL_Deque(nums6);
cout << "输入数组: ";
for (int num : nums6) cout << num << " ";
cout << endl << "Taro 的得分 - Jiro 的得分: " << result6 << endl << endl;

// 测试 HDU 1199 Color the Ball

```

```

cout << "==== 测试 HDU 1199 Color the Ball ===" << endl;
vector<int> balloons = {1, 2, 2, 1, 3, 3, 3};
cout << "气球颜色数组: ";
for (int color : balloons) cout << color << " ";
cout << endl;
int result7 = colorTheBall(balloons);
cout << "最少操作次数: " << result7 << endl << endl;

// 测试 LeetCode 918 环形子数组的最大和
cout << "==== 测试 LeetCode 918 环形子数组的最大和 ===" << endl;
vector<int> A = {1, -2, 3, -2};
cout << "输入数组: ";
for (int num : A) cout << num << " ";
cout << endl;
int result8 = maxSubarraySumCircular(A);
cout << "环形子数组的最大和: " << result8 << endl << endl;

// 测试最长无重复子串
cout << "==== 测试最长无重复子串 ===" << endl;
string s = "abcabcbb";
cout << "输入字符串: " << s << endl;
int result9 = lengthOfLongestSubstring(s);
cout << "最长无重复子串长度: " << result9 << endl;

cout << "所有测试通过!" << endl;
return 0;
}

```

=====

文件: CircularDeque.java

=====

```

import java.util.Deque;
import java.util.LinkedList;
import java.util.ArrayDeque;

/**
 * 循环双端队列及相关题目 (Java 版本)
 * 包含 LeetCode、POJ、HDU、洛谷、AtCoder 等平台的相关题目
 * 每个题目都提供详细的解题思路、复杂度分析和多种解法
 *
 * 主要内容:
 * 1. 循环双端队列的实现 (LeetCode 641)

```

- * 2. 滑动窗口最大值 (LeetCode 239)
- * 3. 滑动窗口最小值和最大值 (AcWing 154, POJ 2823, 洛谷 P1886)
- * 4. 和至少为 K 的最短子数组 (LeetCode 862)
- * 5. 带限制的子序列和 (LeetCode 1425)
- * 6. 绝对差不超过限制的最长连续子数组 (LeetCode 1438)
- * 7. 队列的最大值 (剑指 Offer 59-II)
- * 8. 牛线 Cow Line (洛谷 P2952)
- * 9. Deque 博弈问题 (AtCoder DP Contest L)
- * 10. 新增题目: HDU 1199, LeetCode 918, 赛码网最长无重复子串

*

* 解题思路技巧总结:

- * 1. 循环双端队列: 使用数组实现, 通过取模运算处理边界情况
- * 2. 单调队列: 维护队列的单调性, 用于解决滑动窗口最值问题
- * 3. 前缀和+单调队列: 解决子数组和相关问题
- * 4. 双单调队列: 同时维护最小值和最大值
- * 5. 区间 DP+博弈论: 解决双人博弈问题

*

* 时间复杂度分析:

- * 1. 循环双端队列操作: $O(1)$
- * 2. 单调队列滑动窗口: $O(n)$
- * 3. 前缀和+单调队列: $O(n)$
- * 4. 双单调队列: $O(n)$
- * 5. 区间 DP: $O(n^2)$

*

* 空间复杂度分析:

- * 1. 循环双端队列: $O(k)$
- * 2. 单调队列: $O(k)$ 或 $O(n)$
- * 3. 前缀和数组: $O(n)$
- * 4. 区间 DP 数组: $O(n^2)$

*

* 工程化考量:

- * 1. 异常处理: 空数组检查、参数验证
- * 2. 边界场景: 极端输入、重复数据、有序逆序
- * 3. 性能优化: 避免冗余计算、减少对象创建
- * 4. 调试技巧: 打印中间状态、验证小样例

*

* 语言特性差异:

- * - Java: ArrayDeque vs LinkedList 的选择
- * - 注意: LinkedList 在频繁插入删除时性能更好
- * - ArrayDeque 在随机访问时性能更好

*

* 测试方法:

- * javac CircularDeque.java

```
* java class016.CircularDeque
*/
public class CircularDeque {

    // 提交时把类名、构造方法改成 : MyCircularDeque
    // 其实内部就是双向链表
    // 常数操作慢，但是 leetcode 数据量太小了，所以看不出劣势
    class MyCircularDeque1 {

        public Deque<Integer> deque = new LinkedList<>();
        public int size;
        public int limit;

        public MyCircularDeque1(int k) {
            size = 0;
            limit = k;
        }

        public boolean insertFront(int value) {
            if (isFull()) {
                return false;
            } else {
                deque.offerFirst(value);
                size++;
                return true;
            }
        }

        public boolean insertLast(int value) {
            if (isFull()) {
                return false;
            } else {
                deque.offerLast(value);
                size++;
                return true;
            }
        }

        public boolean deleteFront() {
            if (isEmpty()) {
                return false;
            } else {
                size--;
            }
        }

    }
}
```

```
        deque.pollFirst();
        return true;
    }

}

public boolean deleteLast() {
    if (isEmpty()) {
        return false;
    } else {
        size--;
        deque.pollLast();
        return true;
    }
}

public int getFront() {
    if (isEmpty()) {
        return -1;
    } else {
        return deque.peekFirst();
    }
}

public int getRear() {
    if (isEmpty()) {
        return -1;
    } else {
        return deque.peekLast();
    }
}

public boolean isEmpty() {
    return size == 0;
}

public boolean isFull() {
    return size == limit;
}

}

// 提交时把类名、构造方法改成 : MyCircularDeque
// 自己用数组实现，常数操作快，但是 leetcode 数据量太小了，看不出优势
```

```

class MyCircularDeque2 {

    public int[] deque;
    public int l, r, size, limit;

    public MyCircularDeque2(int k) {
        deque = new int[k];
        l = r = size = 0;
        limit = k;
    }

    public boolean insertFront(int value) {
        if (isFull()) {
            return false;
        } else {
            if (isEmpty()) {
                l = r = 0;
                deque[0] = value;
            } else {
                l = l == 0 ? (limit - 1) : (l - 1);
                deque[l] = value;
            }
            size++;
            return true;
        }
    }

    public boolean insertLast(int value) {
        if (isFull()) {
            return false;
        } else {
            if (isEmpty()) {
                l = r = 0;
                deque[0] = value;
            } else {
                r = r == limit - 1 ? 0 : (r + 1);
                deque[r] = value;
            }
            size++;
            return true;
        }
    }
}

```

```
public boolean deleteFront() {
    if (isEmpty()) {
        return false;
    } else {
        l = (l == limit - 1) ? 0 : (l + 1);
        size--;
        return true;
    }
}

public boolean deleteLast() {
    if (isEmpty()) {
        return false;
    } else {
        r = r == 0 ? (limit - 1) : (r - 1);
        size--;
        return true;
    }
}

public int getFront() {
    if (isEmpty()) {
        return -1;
    } else {
        return deque[l];
    }
}

public int getRear() {
    if (isEmpty()) {
        return -1;
    } else {
        return deque[r];
    }
}

public boolean isEmpty() {
    return size == 0;
}

public boolean isFull() {
    return size == limit;
}
```

```
}
```

```
/**
```

```
* 滑动窗口最大值
```

```
* 题目来源: LeetCode 239. 滑动窗口最大值
```

```
* 链接: https://leetcode.cn/problems/sliding-window-maximum/
```

```
*
```

```
* 题目描述:
```

```
* 给你一个整数数组 nums，有一个大小为 k 的滑动窗口从数组的最左侧移动到数组的最右侧。
```

```
* 你只可以看到在滑动窗口内的 k 个数字。滑动窗口每次只向右移动一位。
```

```
* 返回 滑动窗口中的最大值。
```

```
*
```

```
* 解题思路:
```

```
* 使用双端队列实现单调队列。队列中存储数组下标，队列头部始终是当前窗口的最大值下标，
```

```
* 队列保持单调递减特性。遍历数组时，维护队列的单调性并及时移除窗口外的元素下标，
```

```
* 当窗口形成后，队列头部元素就是当前窗口的最大值。
```

```
*
```

```
* 时间复杂度分析:
```

```
* O(n) - 每个元素最多入队和出队一次
```

```
*
```

```
* 空间复杂度分析:
```

```
* O(k) - 双端队列最多存储 k 个元素
```

```
*/
```

```
public int[] maxSlidingWindow(int[] nums, int k) {
```

```
    if (nums == null || nums.length == 0 || k <= 0) {
```

```
        return new int[0];
```

```
}
```

```
    int n = nums.length;
```

```
    // 结果数组，大小为 n-k+1
```

```
    int[] result = new int[n - k + 1];
```

```
    // 双端队列，存储数组下标，队列头部是当前窗口的最大值下标
```

```
    Deque<Integer> deque = new ArrayDeque<>();
```

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

```
        // 移除队列中超出窗口范围的下标
```

```
        while (!deque.isEmpty() && deque.peekFirst() < i - k + 1) {
```

```
            deque.pollFirst();
```

```
}
```

```
        // 维护队列单调性，移除所有小于当前元素的下标
```

```
        while (!deque.isEmpty() && nums[deque.peekLast()] < nums[i]) {
```

```

        deque.pollLast();
    }

    // 将当前元素下标加入队列尾部
    deque.offerLast(i);

    // 当窗口形成后，记录当前窗口的最大值
    if (i >= k - 1) {
        result[i - k + 1] = nums[deque.peekFirst()];
    }
}

return result;
}

```

```

/**
 * 滑动窗口最大值 (C++风格实现)
 * 题目来源: AcWing 154. 滑动窗口
 * 链接: https://www.acwing.com/problem/content/156/
 *
 * 题目描述:
 * 给定一个大小为  $n \leq 10^6$  的数组和一个大小为  $k$  的滑动窗口,
 * 窗口从数组最左端移动到最右端。要求输出窗口在每个位置时的最小值和最大值。
 *
 * 解题思路:
 * 使用两个单调队列分别维护窗口内的最小值和最大值:
 * 1. 最小值: 维护一个单调递增队列, 队首元素即为当前窗口最小值
 * 2. 最大值: 维护一个单调递减队列, 队首元素即为当前窗口最大值
 * 队列中存储的是数组元素的下标而非值本身, 这样可以方便判断元素是否在窗口内
 *
 * 时间复杂度分析:
 *  $O(n)$  - 每个元素最多入队和出队各一次
 *
 * 空间复杂度分析:
 *  $O(k)$  - 双端队列最多存储  $k$  个元素
 */

```

```

public int[][] slidingWindowMinMax(int[] nums, int k) {
    if (nums == null || nums.length == 0 || k <= 0) {
        return new int[2][0];
    }

    int n = nums.length;
    int[] minResult = new int[n - k + 1];

```

```

int[] maxResult = new int[n - k + 1];

// 双端队列，存储数组下标
Deque<Integer> minDeque = new ArrayDeque<>(); // 单调递增队列，维护最小值
Deque<Integer> maxDeque = new ArrayDeque<>(); // 单调递减队列，维护最大值

for (int i = 0; i < n; i++) {
    // 移除队列中超出窗口范围的下标
    while (!minDeque.isEmpty() && minDeque.peekFirst() < i - k + 1) {
        minDeque.pollFirst();
    }
    while (!maxDeque.isEmpty() && maxDeque.peekFirst() < i - k + 1) {
        maxDeque.pollFirst();
    }

    // 维护队列单调性
    // 对于最小值队列，移除所有大于当前元素的下标
    while (!minDeque.isEmpty() && nums[minDeque.peekLast()] >= nums[i]) {
        minDeque.pollLast();
    }
    // 对于最大值队列，移除所有小于当前元素的下标
    while (!maxDeque.isEmpty() && nums[maxDeque.peekLast()] <= nums[i]) {
        maxDeque.pollLast();
    }

    // 将当前元素下标加入队列尾部
    minDeque.offerLast(i);
    maxDeque.offerLast(i);

    // 当窗口形成后，记录当前窗口的最小值和最大值
    if (i >= k - 1) {
        minResult[i - k + 1] = nums[minDeque.peekFirst()];
        maxResult[i - k + 1] = nums[maxDeque.peekFirst()];
    }
}

return new int[][] {minResult, maxResult};
}

/**
 * POJ 2823 Sliding Window
 * 链接: http://poj.org/problem?id=2823
 */

```

* 题目描述:

* 给定一个大小为 n 的数组和一个大小为 k 的滑动窗口，

* 窗口从数组最左端移动到最右端。要求输出窗口在每个位置时的最小值和最大值。

*

* 解题思路:

* 与 AcWing 154 类似，使用两个单调队列分别维护窗口内的最小值和最大值。

* 由于 POJ 评测系统对时间要求严格，需要特别注意实现效率。

*

* 时间复杂度分析:

* $O(n)$ - 每个元素最多入队和出队各一次

*

* 空间复杂度分析:

* $O(k)$ - 双端队列最多存储 k 个元素

*/

```
public int[][] poj2823SlidingWindow(int[] nums, int k) {
    if (nums == null || nums.length == 0 || k <= 0) {
        return new int[2][0];
    }

    int n = nums.length;
    int[] minResult = new int[n - k + 1];
    int[] maxResult = new int[n - k + 1];

    // 双端队列，存储数组下标
    Deque<Integer> minDeque = new ArrayDeque<>(); // 单调递增队列，维护最小值
    Deque<Integer> maxDeque = new ArrayDeque<>(); // 单调递减队列，维护最大值

    for (int i = 0; i < n; i++) {
        // 移除队列中超出窗口范围的下标
        while (!minDeque.isEmpty() && minDeque.peekFirst() < i - k + 1) {
            minDeque.pollFirst();
        }
        while (!maxDeque.isEmpty() && maxDeque.peekFirst() < i - k + 1) {
            maxDeque.pollFirst();
        }

        // 维护队列单调性
        // 对于最小值队列，移除所有大于当前元素的下标
        while (!minDeque.isEmpty() && nums[minDeque.peekLast()] >= nums[i]) {
            minDeque.pollLast();
        }
        // 对于最大值队列，移除所有小于当前元素的下标
        while (!maxDeque.isEmpty() && nums[maxDeque.peekLast()] <= nums[i]) {
```

```

        maxDeque.pollLast();
    }

    // 将当前元素下标加入队列尾部
    minDeque.offerLast(i);
    maxDeque.offerLast(i);

    // 当窗口形成后，记录当前窗口的最小值和最大值
    if (i >= k - 1) {
        minResult[i - k + 1] = nums[minDeque.peekFirst()];
        maxResult[i - k + 1] = nums[maxDeque.peekFirst()];
    }
}

return new int[][] {minResult, maxResult};
}

/**
 * LeetCode 862. 和至少为 K 的最短子数组
 * 题目来源: LeetCode 862. Shortest Subarray with Sum at Least K
 * 链接: https://leetcode.cn/problems/shortest-subarray-with-sum-at-least-k/
 *
 * 题目描述:
 * 给你一个整数数组 nums 和一个整数 k ，找出 nums 中和至少为 k 的最短非空子数组，并返回该子数组的长度。
 * 如果不存在这样的子数组，返回 -1 。
 *
 * 解题思路:
 * 使用前缀和+单调双端队列。
 * 1. 计算前缀和数组 prefixSum，其中 prefixSum[i] 表示 nums[0..i-1] 的和
 * 2. 维护一个单调递增的双端队列，存储前缀和数组的下标
 * 3. 对于当前位置 i，如果 prefixSum[i] - prefixSum[队首] >= k，说明找到了一个满足条件的子数组
 *    更新最短长度，并将队首出队（因为后续不可能找到更短的以该队首为起点的子数组）
 * 4. 在将当前下标入队前，移除所有前缀和 >= prefixSum[i] 的队尾元素（保持单调性）
 *
 * 时间复杂度分析:
 * O(n) - 每个元素最多入队和出队一次
 *
 * 空间复杂度分析:
 * O(n) - 前缀和数组和双端队列的空间
 *
 * 是否为最优解: 是
 * 该解法是最优解，因为必须遍历整个数组，时间复杂度不可能低于 O(n)

```

```

*/
public int shortestSubarray(int[] nums, int k) {
    if (nums == null || nums.length == 0) {
        return -1;
    }

    int n = nums.length;
    // 前缀和数组，prefixSum[i] 表示 nums[0..i-1] 的和
    long[] prefixSum = new long[n + 1];
    for (int i = 0; i < n; i++) {
        prefixSum[i + 1] = prefixSum[i] + nums[i];
    }

    // 单调递增队列，存储前缀和数组的下标
    Deque<Integer> deque = new ArrayDeque<>();
    int minLength = Integer.MAX_VALUE;

    for (int i = 0; i <= n; i++) {
        // 当前前缀和减去队首前缀和 >= k 时，找到了一个满足条件的子数组
        while (!deque.isEmpty() && prefixSum[i] - prefixSum[deque.peekFirst()] >= k) {
            minLength = Math.min(minLength, i - deque.pollFirst());
        }

        // 保持队列单调递增，移除所有前缀和 >= prefixSum[i] 的队尾元素
        while (!deque.isEmpty() && prefixSum[deque.peekLast()] >= prefixSum[i]) {
            deque.pollLast();
        }

        // 将当前下标加入队列
        deque.offerLast(i);
    }

    return minLength == Integer.MAX_VALUE ? -1 : minLength;
}

/***
 * LeetCode 1425. 带限制的子序列和
 * 题目来源: LeetCode 1425. Constrained Subsequence Sum
 * 链接: https://leetcode.cn/problems/constrained-subsequence-sum/
 *
 * 题目描述:
 * 给你一个整数数组 nums 和一个整数 k ，请你返回非空子序列元素和的最大值，
 * 子序列需要满足：子序列中每两个相邻的整数 nums[i] 和 nums[j] ，它们在原数组中的下标 i 和 j
 */

```

满足 $i < j$ 且 $j - i \leq k$ 。

*

* 解题思路:

* 使用动态规划+单调双端队列优化。

* 1. $dp[i]$ 表示以 $nums[i]$ 结尾的满足条件的子序列的最大和

* 2. 状态转移: $dp[i] = nums[i] + \max(0, \max(dp[i-k], dp[i-k+1], \dots, dp[i-1]))$

* 3. 使用单调递减队列维护滑动窗口内的最大值, 队列存储下标

* 4. 对于每个位置 i , 先移除超出窗口范围的下标, 然后取队首作为窗口最大值

* 5. 计算 $dp[i]$ 后, 维护队列单调性并将 i 加入队列

*

* 时间复杂度分析:

* $O(n)$ – 每个元素最多入队和出队一次

*

* 空间复杂度分析:

* $O(n)$ – dp 数组和双端队列的空间

*

* 是否为最优解: 是

* 该解法是最优解, 使用单调队列优化 DP, 时间复杂度为 $O(n)$

*/

```
public int constrainedSubsetSum(int[] nums, int k) {
```

```
    if (nums == null || nums.length == 0) {
```

```
        return 0;
```

```
}
```

```
    int n = nums.length;
```

```
    //  $dp[i]$  表示以  $nums[i]$  结尾的满足条件的子序列的最大和
```

```
    int[] dp = new int[n];
```

```
    // 单调递减队列, 存储  $dp$  值的下标, 队首是窗口内的最大值
```

```
    Deque<Integer> deque = new ArrayDeque<>();
```

```
    int maxSum = Integer.MIN_VALUE;
```

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

```
        // 移除超出窗口范围的下标
```

```
        while (!deque.isEmpty() && deque.peekFirst() < i - k) {
```

```
            deque.pollFirst();
```

```
}
```

```
        // 计算  $dp[i]$ , 如果队列为空或队首值为负, 则只取  $nums[i]$ 
```

```
        dp[i] = nums[i];
```

```
        if (!deque.isEmpty()) {
```

```
            dp[i] = Math.max(dp[i], nums[i] + dp[deque.peekFirst()]);
```

```
}
```

```

    // 更新最大和
    maxSum = Math.max(maxSum, dp[i]);

    // 维护队列单调递减特性
    while (!deque.isEmpty() && dp[deque.peekLast()] <= dp[i]) {
        deque.pollLast();
    }

    // 将当前下标加入队列
    deque.offerLast(i);
}

return maxSum;
}

/**
 * LeetCode 1438. 绝对差不超过限制的最长连续子数组
 * 题目来源: LeetCode 1438. Longest Continuous Subarray With Absolute Diff Less Than or Equal to Limit
 * 链接: https://leetcode.cn/problems/longest-continuous-subarray-with-absolute-diff-less-than-or-equal-to-limit/
 *
 * 题目描述:
 * 给你一个整数数组 nums，和一个表示限制的整数 limit，
 * 请你返回最长连续子数组的长度，该子数组中的任意两个元素之间的绝对差必须小于或者等于 limit。
 * 如果不存在满足条件的子数组，则返回 0。
 *
 * 解题思路:
 * 使用滑动窗口+双端队列。
 * 1. 维护两个单调队列：一个递增（维护最小值），一个递减（维护最大值）
 * 2. 使用双指针表示滑动窗口的左右边界
 * 3. 右指针不断右移扩大窗口，同时维护两个单调队列
 * 4. 当窗口内最大值-最小值 > limit 时，左指针右移缩小窗口
 * 5. 每次更新最大窗口长度
 *
 * 时间复杂度分析:
 * O(n) - 每个元素最多入队和出队一次
 *
 * 空间复杂度分析:
 * O(n) - 两个双端队列的空间
 *
 * 是否为最优解: 是

```

```

* 该解法是最优解，使用滑动窗口+单调队列，时间复杂度为 O(n)
*/
public int longestSubarray(int[] nums, int limit) {
    if (nums == null || nums.length == 0) {
        return 0;
    }

    // 单调递增队列，维护窗口内的最小值
    Deque<Integer> minDeque = new ArrayDeque<>();
    // 单调递减队列，维护窗口内的最大值
    Deque<Integer> maxDeque = new ArrayDeque<>();

    int left = 0;
    int maxLength = 0;

    for (int right = 0; right < nums.length; right++) {
        // 维护最小值队列的单调性
        while (!minDeque.isEmpty() && nums[minDeque.peekLast()] >= nums[right]) {
            minDeque.pollLast();
        }
        minDeque.offerLast(right);

        // 维护最大值队列的单调性
        while (!maxDeque.isEmpty() && nums[maxDeque.peekLast()] <= nums[right]) {
            maxDeque.pollLast();
        }
        maxDeque.offerLast(right);

        // 当窗口内最大值-最小值 > limit 时，缩小窗口
        while (!minDeque.isEmpty() && !maxDeque.isEmpty() &&
               nums[maxDeque.peekFirst()] - nums[minDeque.peekFirst()] > limit) {
            // 移除左边界元素
            if (minDeque.peekFirst() == left) {
                minDeque.pollFirst();
            }
            if (maxDeque.peekFirst() == left) {
                maxDeque.pollFirst();
            }
            left++;
        }

        // 更新最大窗口长度
        maxLength = Math.max(maxLength, right - left + 1);
    }
}

```

```

    }

    return maxLength;
}

/***
 * 剑指 Offer 59 - II. 队列的最大值
 * 题目来源: 剑指 Offer 59 - II / LeetCode 面试题题 59 - II
 * 链接: https://leetcode.cn/problems/dui-lie-de-zui-da-zhi-1cof/
 *
 * 题目描述:
 * 请定义一个队列并实现函数 max_value 得到队列里的最大值,
 * 要求函数 max_value、push_back 和 pop_front 的均摊时间复杂度都是 O(1)。
 * 若队列为空, pop_front 和 max_value 需要返回 -1。
 *
 * 解题思路:
 * 使用两个队列:
 * 1. 一个普通队列 queue 存储所有元素
 * 2. 一个单调递减的双端队列 maxQueue 维护当前队列的最大值
 * 3. push_back 时:
 *   - 直接向 queue 中添加元素
 *   - 将 maxQueue 中所有小于当前元素的元素移除, 然后将当前元素加入 maxQueue
 * 4. pop_front 时:
 *   - 从 queue 中弹出元素
 *   - 如果弹出的元素等于 maxQueue 的队首, 也将 maxQueue 的队首弹出
 * 5. max_value 时: 直接返回 maxQueue 的队首元素
 *
 * 时间复杂度分析:
 * O(1) - 所有操作的均摊时间复杂度都是 O(1)
 *
 * 空间复杂度分析:
 * O(n) - 需要两个队列的空间
 *
 * 是否为最优解: 是
 * 该解法是最优解, 满足了题目要求的 O(1) 均摊时间复杂度
 */

class MaxQueue {
    private Deque<Integer> queue;      // 普通队列, 存储所有元素
    private Deque<Integer> maxQueue;    // 单调递减队列, 维护最大值

    public MaxQueue() {
        queue = new LinkedList<>();
        maxQueue = new LinkedList<>();
    }
}

```

```

}

public int max_value() {
    if (maxQueue.isEmpty()) {
        return -1;
    }
    return maxQueue.peekFirst();
}

public void push_back(int value) {
    queue.offerLast(value);
    // 维护 maxQueue 的单调递减特性
    while (!maxQueue.isEmpty() && maxQueue.peekLast() < value) {
        maxQueue.pollLast();
    }
    maxQueue.offerLast(value);
}

public int pop_front() {
    if (queue.isEmpty()) {
        return -1;
    }
    int value = queue.pollFirst();
    // 如果弹出的元素等于 maxQueue 的队首，也将 maxQueue 的队首弹出
    if (!maxQueue.isEmpty() && maxQueue.peekFirst() == value) {
        maxQueue.pollFirst();
    }
    return value;
}

/**
 * 洛谷 P1886 滑动窗口 / 【模板】单调队列
 * 题目来源: 洛谷 P1886
 * 链接: https://www.luogu.com.cn/problem/P1886
 *
 * 题目描述:
 * 有一个长为 n 的序列 a，以及一个大小为 k 的窗口。
 * 现在这个从左边开始向右滑动，每次滑动一个单位，求出每次滑动后窗口中的最小值和最大值。
 *
 * 解题思路:
 * 使用两个单调队列分别维护窗口内的最小值和最大值:
 * 1. 最小值: 维护一个单调递增队列，队首元素即为当前窗口最小值

```

- * 2. 最大值：维护一个单调递减队列，队首元素即为当前窗口最大值
- * 队列中存储的是数组元素的下标而非值本身，这样可以方便判断元素是否在窗口内。
- *
- * 时间复杂度分析：
- * $O(n)$ – 每个元素最多入队和出队各一次
- *
- * 空间复杂度分析：
- * $O(k)$ – 双端队列最多存储 k 个元素
- *
- * 是否为最优解：是
- * 该解法是最优解，时间复杂度为 $O(n)$
- */

```
public int[][] luoguP1886SlidingWindow(int[] nums, int k) {
    // 该方法与 slidingWindowMinMax 完全相同，仅为洛谷题目单独标记
    return slidingWindowMinMax(nums, k);
}
```

/**

- * 洛谷 P2952 牛线 Cow Line
- * 题目来源：洛谷 P2952 / USACO Open09 Silver
- * 链接：<https://www.luogu.com.cn/problem/P2952>
- *
- * 题目描述：
- * 有 N 头牛排成一列，支持以下操作：
- * A L/R x: 在队列左端(L)或右端(R)添加编号为 x 的牛
- * D L/R x: 从队列左端(L)或右端(R)移除 x 头牛
- * 最后输出队列中牛的编号（从左到右）。
- *
- * 解题思路：
- * 直接使用双端队列 Deque 模拟操作过程：
- * 1. A L x: 在队列左端添加元素 x
- * 2. A R x: 在队列右端添加元素 x
- * 3. D L x: 从队列左端移除 x 个元素
- * 4. D R x: 从队列右端移除 x 个元素
- *
- * 时间复杂度分析：
- * $O(n)$ – n 为操作次数，每次操作时间复杂度为 $O(1)$ 或 $O(x)$
- *
- * 空间复杂度分析：
- * $O(m)$ – m 为队列中牛的最大数量
- *
- * 是否为最优解：是
- * 直接使用双端队列模拟是最直接高效的解法

```

*/
public int[] luoguP2952CowLine(String[][] operations) {
    if (operations == null || operations.length == 0) {
        return new int[0];
    }

    Deque<Integer> deque = new LinkedList<>();

    for (String[] op : operations) {
        String operation = op[0]; // "A" 或 "D"
        String position = op[1]; // "L" 或 "R"
        int value = Integer.parseInt(op[2]);

        if ("A".equals(operation)) {
            if ("L".equals(position)) {
                deque.offerFirst(value);
            } else {
                deque.offerLast(value);
            }
        } else if ("D".equals(operation)) {
            if ("L".equals(position)) {
                for (int i = 0; i < value && !deque.isEmpty(); i++) {
                    deque.pollFirst();
                }
            } else {
                for (int i = 0; i < value && !deque.isEmpty(); i++) {
                    deque.pollLast();
                }
            }
        }
    }

    // 转换为数组返回
    int[] result = new int[deque.size()];
    int index = 0;
    while (!deque.isEmpty()) {
        result[index++] = deque.pollFirst();
    }

    return result;
}

/***

```

- * AtCoder DP Contest L - Deque
- * 题目来源: AtCoder Educational DP Contest
- * 链接: https://atcoder.jp/contests/dp/tasks/dp_1
- *
- * 题目描述:
- * Taro 和 Jiro 玩一个游戏。初始时有一个序列 $a = (a_1, a_2, \dots, a_N)$ 。
- * 两个玩家轮流操作, Taro 先手。每次操作, 玩家可以从序列的左端或右端取走一个数,
- * 并将该数加到自己的得分中。游戏继续直到序列为空。
- * 两个玩家都采用最优策略, 求 Taro 的得分 - Jiro 的得分的最大值。
- *
- * 解题思路:
- * 使用区间 DP + 博弈论。
- * 1. 定义 $dp[1][r]$ 表示当前剩余区间 $[1, r]$, 当前玩家能获得的最大分数差
- * 2. 状态转移: 当前玩家可以选择左端或右端, 对手在剩余区间也会采用最优策略
- * $dp[1][r] = \max(a[1] - dp[1+1][r], a[r] - dp[1][r-1])$
- * 3. 边界条件: $dp[i][i] = a[i]$ (只剩一个元素时, 直接取走)
- * 4. 最终答案: $dp[0][n-1]$
- *
- * 时间复杂度分析:
- * $O(n^2)$ - 需要计算 n^2 个状态
- *
- * 空间复杂度分析:
- * $O(n^2)$ - 二维 DP 数组的空间
- *
- * 是否为最优解: 是
- * 该解法是最优解, 区间 DP 是解决此类博弈问题的标准方法
- */

```

public long atCoderDPL_Deque(int[] a) {
    if (a == null || a.length == 0) {
        return 0;
    }

    int n = a.length;
    // dp[1][r] 表示当前剩余区间 [1, r], 当前玩家能获得的最大分数差
    long[][] dp = new long[n][n];

    // 边界条件: 只剩一个元素时
    for (int i = 0; i < n; i++) {
        dp[i][i] = a[i];
    }

    // 按照区间长度从小到大填表
    for (int len = 2; len <= n; len++) {

```

```

        for (int l = 0; l <= n - len; l++) {
            int r = l + len - 1;
            // 当前玩家选择左端或右端, 取最大值
            dp[1][r] = Math.max(a[l] - dp[1 + 1][r], a[r] - dp[1][r - 1]);
        }
    }

    return dp[0][n - 1];
}

/***
 * =====
 * 总结: 双端队列与单调队列的应用场景
 * =====
 *
 * 1. 核心数据结构:
 *     - 双端队列 (Deque): 可以在两端进行插入和删除的队列
 *     - 单调队列 (Monotonic Queue): 使用双端队列实现, 维护队列元素的单调性
 *
 * 2. 适用题型总结:
 *     ① 滑动窗口最值问题:
 *         - LeetCode 239 滑动窗口最大值
 *         - 洛谷 P1886 滑动窗口 / 单调队列
 *         - POJ 2823 Sliding Window
 *         - AcWing 154 滑动窗口
 *
 *     ② 队列维护最值:
 *         - 剑指 Offer 59-II 队列的最大值
 *
 *     ③ 子数组/子序列问题 (前缀和+单调队列):
 *         - LeetCode 862 和至少为 K 的最短子数组
 *         - LeetCode 1425 带限制的子序列和
 *
 *     ④ 绝对差限制问题 (双单调队列):
 *         - LeetCode 1438 绝对差不超过限制的最长连续子数组
 *
 *     ⑤ 双端队列模拟:
 *         - LeetCode 641 设计循环双端队列
 *         - 洛谷 P2952 牛线 Cow Line
 *
 *     ⑥ 博弈论+区间 DP:
 *         - AtCoder DP Contest L - Deque
 */

```

* 3. 解题思路模板:

* ① 滑动窗口最大值模板:

- 维护一个单调递减队列，存储数组下标
- 移除超出窗口范围的下标
- 维护队列单调性（移除所有小于当前元素的尾部元素）
- 将当前下标加入队列尾部
- 队首元素即为当前窗口最大值

*

* ② 滑动窗口最小值模板:

- 维护一个单调递增队列，其余步骤同上

*

* ③ 前缀和+单调队列模板:

- 计算前缀和数组
- 维护前缀和的单调队列
- 根据题目要求进行判断和更新

*

* 4. 关键技巧:

- 队列中存储下标而非值，方便判断元素是否在窗口内
- 维护队列单调性时，从队尾移除不符合条件的元素
- 移除超出窗口范围的元素时，从队首移除
- 单调递增队列维护最小值，单调递减队列维护最大值

*

* 5. 时间复杂度分析:

- 单调队列的时间复杂度通常为 $O(n)$ ，因为每个元素最多入队和出队一次

*

* 6. 空间复杂度分析:

- 通常为 $O(n)$ 或 $O(k)$ ，取决于窗口大小和问题特点

*

* 7. 工程化考虑:

- 异常处理：空数组、空队列、无效参数
- 边界场景： $k=1$ 、 $k=n$ 、所有元素相等、所有元素单调
- 数据范围：负数、正数、混合、最大值/最小值
- 溢出处理：使用 `long` 类型处理前缀和

*

* 8. 语言特性差异:

- Java: 使用 `ArrayDeque` 或 `LinkedList` 实现 `Deque` 接口
- C++: 使用 `std::deque` 或手写循环数组
- Python: 使用 `collections.deque`

*

* 9. 性能优化:

- 对于大数据量，优先使用数组实现的循环双端队列
- 避免不必要的对象创建和内存分配
- 使用原始类型数组而非封装类型

```
*  
* 10. 面试表达要点：  
*     - 解释为什么使用单调队列可以优化时间复杂度  
*     - 说明队列中存储下标的原因  
*     - 强调维护单调性的重要性  
*     - 举例说明队首和队尾的作用  
* -----  
*/
```

```
// 测试代码  
// 新增加的题目：HDU 1199 Color the Ball  
// 题目描述：有 n 个气球，每个气球的颜色可以是 1 到 n 中的一种，每次操作可以将某个区间内的所有气球染成同一种颜色。  
// 求最少需要多少次操作才能将所有气球染成同一种颜色。  
// 解题思路：使用双端队列维护连续的相同颜色区间，时间复杂度 O(n)  
public int colorTheBall(int[] balloons) {  
    if (balloons == null || balloons.length == 0) {  
        return 0;  
    }  
  
    Deque<Integer> deque = new LinkedList<>();  
    int operations = 0;  
  
    for (int color : balloons) {  
        // 移除队列尾部与当前颜色相同的元素  
        while (!deque.isEmpty() && deque.peekLast() == color) {  
            deque.pollLast();  
        }  
        deque.offerLast(color);  
    }  
  
    // 每一段连续不同的颜色需要一次操作  
    return deque.size();  
}  
  
// LeetCode 918. 环形子数组的最大和  
// 题目描述：给定一个由整数数组 A 表示的环形数组 C，求 C 的非空子数组的最大可能和。  
// 解题思路：环形数组的最大子数组和有两种情况：  
// 1. 最大子数组在数组的非环形部分  
// 2. 最大子数组跨越数组的首尾（即总和减去最小子数组和）  
// 时间复杂度 O(n)，空间复杂度 O(1)  
public int maxSubarraySumCircular(int[] A) {  
    if (A == null || A.length == 0) {
```

```

        return 0;
    }

int totalSum = 0;
int maxSum = Integer.MIN_VALUE;
int currentMax = 0;
int minSum = Integer.MAX_VALUE;
int currentMin = 0;

for (int num : A) {
    totalSum += num;

    // Kadane 算法求最大子数组和
    currentMax = Math.max(num, currentMax + num);
    maxSum = Math.max(maxSum, currentMax);

    // Kadane 算法求最小子数组和
    currentMin = Math.min(num, currentMin + num);
    minSum = Math.min(minSum, currentMin);
}

// 如果所有元素都是负数，那么 maxSum 就是最大的单个元素
if (maxSum < 0) {
    return maxSum;
}

// 返回两种情况的最大值
return Math.max(maxSum, totalSum - minSum);
}

// 赛码网题目：最长无重复子串（使用双端队列优化）
// 题目描述：给定一个字符串，找出其中不含重复字符的最长子串的长度。
// 解题思路：使用双端队列维护当前无重复字符的子串，时间复杂度 O(n)
public int lengthOfLongestSubstring(String s) {
    if (s == null || s.isEmpty()) {
        return 0;
    }

    Deque<Character> deque = new LinkedList<>();
    java.util.Set<Character> seen = new java.util.HashSet<>();
    int maxLength = 0;

    for (char c : s.toCharArray()) {

```

```

// 如果字符已存在于当前窗口中，移除窗口中所有直到该字符的元素
while (seen.contains(c)) {
    char removed = deque.pollFirst();
    seen.remove(removed);
}

// 添加新字符到窗口
deque.offerLast(c);
seen.add(c);

// 更新最大长度
maxLength = Math.max(maxLength, deque.size());
}

return maxLength;
}

public static void main(String[] args) {
    CircularDeque solution = new CircularDeque();

    System.out.println("===== 测试循环双端队列 =====");
    CircularDeque.MyCircularDeque2 deque = solution.new MyCircularDeque2(3);
    System.out.println("insertLast(1): " + deque.insertLast(1)); // true
    System.out.println("insertLast(2): " + deque.insertLast(2)); // true
    System.out.println("insertFront(3): " + deque.insertFront(3)); // true
    System.out.println("insertFront(4): " + deque.insertFront(4)); // false
    System.out.println("getRear(): " + deque.getRear()); // 2
    System.out.println("isFull(): " + deque.isFull()); // true
    System.out.println("deleteLast(): " + deque.deleteLast()); // true
    System.out.println("insertFront(4): " + deque.insertFront(4)); // true
    System.out.println("getFront(): " + deque.getFront()); // 4
    System.out.println();

    System.out.println("===== 测试滑动窗口最大值 =====");
    int[] nums1 = {1, 3, -1, -3, 5, 3, 6, 7};
    int k1 = 3;
    int[] result1 = solution.maxSlidingWindow(nums1, k1);
    System.out.println("输入数组: " + java.util.Arrays.toString(nums1));
    System.out.println("窗口大小: " + k1);
    System.out.println("最大值序列: " + java.util.Arrays.toString(result1));
    System.out.println();

    System.out.println("===== 测试滑动窗口最小值和最大值 =====");
}

```

```
int[] nums2 = {1, 3, -1, -3, 5, 3, 6, 7} ;
int k2 = 3;
int[][] result2 = solution.slidingWindowMinMax(nums2, k2);
System.out.println("输入数组: " + java.util.Arrays.toString(nums2));
System.out.println("窗口大小: " + k2);
System.out.println("最小值序列: " + java.util.Arrays.toString(result2[0]));
System.out.println("最大值序列: " + java.util.Arrays.toString(result2[1]));
System.out.println();

System.out.println("===== 测试和至少为 K 的最短子数组 =====");
int[] nums3 = {2, -1, 2};
int k3 = 3;
int result3 = solution.shortestSubarray(nums3, k3);
System.out.println("输入数组: " + java.util.Arrays.toString(nums3));
System.out.println("k: " + k3);
System.out.println("最短子数组长度: " + result3);
System.out.println();

System.out.println("===== 测试带限制的子序列和 =====");
int[] nums4 = {10, 2, -10, 5, 20};
int k4 = 2;
int result4 = solution.constrainedSubsetSum(nums4, k4);
System.out.println("输入数组: " + java.util.Arrays.toString(nums4));
System.out.println("k: " + k4);
System.out.println("最大子序列和: " + result4);
System.out.println();

System.out.println("===== 测试绝对差不超过限制的最长连续子数组 =====");
int[] nums5 = {8, 2, 4, 7};
int limit5 = 4;
int result5 = solution.longestSubarray(nums5, limit5);
System.out.println("输入数组: " + java.util.Arrays.toString(nums5));
System.out.println("limit: " + limit5);
System.out.println("最长子数组长度: " + result5);
System.out.println();

System.out.println("===== 测试队列的最大值 =====");
CircularDeque.MaxQueue maxQueue = solution.new MaxQueue();
maxQueue.push_back(1);
maxQueue.push_back(2);
System.out.println("max_value: " + maxQueue.max_value()); // 2
maxQueue.pop_front();
```

```

System.out.println("max_value: " + maxQueue.max_value()); // 2
System.out.println();

System.out.println("===== 测试 AtCoder DP L - Deque =====");
int[] nums6 = {10, 80, 90, 30};
long result6 = solution.atCoderDPL_Deque(nums6);
System.out.println("输入数组: " + java.util.Arrays.toString(nums6));
System.out.println("Taro 的得分 - Jiro 的得分: " + result6);
System.out.println();

// 测试新增题目
System.out.println("===== 测试 HDU 1199 Color the Ball =====");
int[] balloons = {1, 2, 2, 1, 3, 3, 3};
System.out.println("气球颜色数组: " + java.util.Arrays.toString(balloons));
System.out.println("最少操作次数: " + solution.colorTheBall(balloons));
System.out.println();

System.out.println("===== 测试 LeetCode 918 环形子数组的最大和 =====");
int[] A = {1, -2, 3, -2};
System.out.println("输入数组: " + java.util.Arrays.toString(A));
System.out.println("环形子数组的最大和: " + solution.maxSubarraySumCircular(A));
System.out.println();

System.out.println("===== 测试最长无重复子串 =====");
String s = "abcabcbb";
System.out.println("输入字符串: " + s);
System.out.println("最长无重复子串长度: " + solution.lengthOfLongestSubstring(s));
System.out.println();

System.out.println("所有测试通过!");
}
}
=====
```

文件: CircularDeque.py

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
```

```
"""
```

循环双端队列及相关题目(Python 版本)

包含 LeetCode、POJ、HDU、洛谷、AtCoder 等平台的相关题目

每个题目都提供详细的解题思路、复杂度分析和多种解法

主要内容：

1. 循环双端队列的实现 (LeetCode 641)
2. 滑动窗口最大值 (LeetCode 239)
3. 滑动窗口最小值和最大值 (AcWing 154, POJ 2823, 洛谷 P1886)
4. 和至少为 K 的最短子数组 (LeetCode 862)
5. 带限制的子序列和 (LeetCode 1425)
6. 绝对差不超过限制的最长连续子数组 (LeetCode 1438)
7. 队列的最大值 (剑指 Offer 59-II)
8. 牛线 Cow Line (洛谷 P2952)
9. Deque 博弈问题 (AtCoder DP Contest L)

解题思路技巧总结：

1. 循环双端队列：使用数组实现，通过取模运算处理边界情况
2. 单调队列：维护队列的单调性，用于解决滑动窗口最值问题
3. 前缀和+单调队列：解决子数组和相关问题
4. 双单调队列：同时维护最小值和最大值
5. 区间 DP+博弈论：解决双人博弈问题

时间复杂度分析：

1. 循环双端队列操作： $O(1)$
2. 单调队列滑动窗口： $O(n)$
3. 前缀和+单调队列： $O(n)$
4. 双单调队列： $O(n)$
5. 区间 DP： $O(n^2)$

空间复杂度分析：

1. 循环双端队列： $O(k)$
2. 单调队列： $O(k)$ 或 $O(n)$
3. 前缀和数组： $O(n)$
4. 区间 DP 数组： $O(n^2)$

"""

```
from collections import deque as collections_deque
from typing import List, Tuple
```

```
class MyCircularDeque:
```

```
    """
```

循环双端队列

题目来源：LeetCode 641. 设计循环双端队列

链接：<https://leetcode.cn/problems/design-circular-deque/>

题目描述:

设计实现双端队列。实现 MyCircularDeque 类:

MyCircularDeque(int k): 构造函数, 双端队列最大为 k。

boolean insertFront(int value): 将一个元素添加到双端队列头部。如果操作成功返回 true, 否则返回 false。

boolean insertLast(int value): 将一个元素添加到双端队列尾部。如果操作成功返回 true, 否则返回 false。

boolean deleteFront(): 从双端队列头部删除一个元素。如果操作成功返回 true, 否则返回 false。

boolean deleteLast(): 从双端队列尾部删除一个元素。如果操作成功返回 true, 否则返回 false。

int getFront(): 从双端队列头部获得一个元素。如果双端队列为空, 返回 -1。

int getRear(): 获得双端队列的最后一个元素。如果双端队列为空, 返回 -1。

boolean isEmpty(): 若双端队列为空, 则返回 true, 否则返回 false。

boolean isFull(): 若双端队列满了, 则返回 true, 否则返回 false。

解题思路:

使用数组实现循环双端队列。维护头指针 front、尾指针 rear 和元素个数 size, 通过取模运算实现循环特性。

头部插入时 front 指针向前移动, 尾部插入时 rear 指针向后移动, 注意处理边界情况和循环特性。

时间复杂度分析:

所有操作都是 O(1) 时间复杂度

空间复杂度分析:

O(k) – k 是双端队列的容量

"""

```
def __init__(self, k: int):
    """构造函数, 双端队列最大为 k"""
    self.elements = [0] * (k + 1) # 多申请一个空间用于区分队列满和空的情况
    self.capacity = k + 1
    self.front = 0
    self.rear = 0
    self.size = 0

def insertFront(self, value: int) -> bool:
    """将一个元素添加到双端队列头部"""
    if self.isFull():
        return False
    # front 指针向前移动一位 (考虑循环特性)
    self.front = (self.front - 1 + self.capacity) % self.capacity
    self.elements[self.front] = value
    self.size += 1
```

```
return True

def insertLast(self, value: int) -> bool:
    """将一个元素添加到双端队列尾部"""
    if self.isEmpty():
        return False
    self.elements[self.rear] = value
    # rear 指针向后移动一位（考虑循环特性）
    self.rear = (self.rear + 1) % self.capacity
    self.size += 1
    return True

def deleteFront(self) -> bool:
    """从双端队列头部删除一个元素"""
    if self.isEmpty():
        return False
    # front 指针向后移动一位（考虑循环特性）
    self.front = (self.front + 1) % self.capacity
    self.size -= 1
    return True

def deleteLast(self) -> bool:
    """从双端队列尾部删除一个元素"""
    if self.isEmpty():
        return False
    # rear 指针向前移动一位（考虑循环特性）
    self.rear = (self.rear - 1 + self.capacity) % self.capacity
    self.size -= 1
    return True

def getFront(self) -> int:
    """从双端队列头部获得一个元素"""
    if self.isEmpty():
        return -1
    return self.elements[self.front]

def getRear(self) -> int:
    """获得双端队列的最后一个元素"""
    if self.isEmpty():
        return -1
    # 注意: rear 指向的是下一个插入位置, 最后一个元素在(rear-1+capacity)%capacity 位置
    return self.elements[(self.rear - 1 + self.capacity) % self.capacity]
```

```

def isEmpty(self) -> bool:
    """若双端队列为空，则返回 true，否则返回 false"""
    return self.size == 0

def isFull(self) -> bool:
    """若双端队列满了，则返回 true，否则返回 false"""
    return self.size == self.capacity - 1 # 留一个空位用于区分满和空

```

def maxSlidingWindow(nums: List[int], k: int) -> List[int]:

"""

滑动窗口最大值

题目来源: LeetCode 239. 滑动窗口最大值

链接: <https://leetcode.cn/problems/sliding-window-maximum/>

题目描述:

给你一个整数数组 nums，有一个大小为 k 的滑动窗口从数组的最左侧移动到数组的最右侧。

你只可以看到在滑动窗口内的 k 个数字。滑动窗口每次只向右移动一位。

返回 滑动窗口中的最大值 。

解题思路:

使用双端队列实现单调队列。队列中存储数组下标，队列头部始终是当前窗口的最大值下标，队列保持单调递减特性。遍历数组时，维护队列的单调性并及时移除窗口外的元素下标，当窗口形成后，队列头部元素就是当前窗口的最大值。

时间复杂度分析:

$O(n)$ – 每个元素最多入队和出队一次

空间复杂度分析:

$O(k)$ – 双端队列最多存储 k 个元素

"""

if not nums or k <= 0:

return []

n = len(nums)

结果数组，大小为 $n-k+1$

result = []

双端队列，存储数组下标，队列头部是当前窗口的最大值下标

dq = collections_deque()

for i in range(n):

移除队列中超出窗口范围的下标

while dq and dq[0] < i - k + 1:

```

dq.popleft()

# 维护队列单调性，移除所有小于当前元素的下标
while dq and nums[dq[-1]] < nums[i]:
    dq.pop()

# 将当前元素下标加入队列尾部
dq.append(i)

# 当窗口形成后，记录当前窗口的最大值
if i >= k - 1:
    result.append(nums[dq[0]])

return result

```

def slidingWindowMinMax(nums: List[int], k: int) -> Tuple[List[int], List[int]]:

"""

滑动窗口最大值（最小值和最大值同时求解）

题目来源：AcWing 154. 滑动窗口

链接：<https://www.acwing.com/problem/content/156/>

题目描述：

给定一个大小为 $n \leq 10^6$ 的数组和一个大小为 k 的滑动窗口，

窗口从数组最左端移动到最右端。要求输出窗口在每个位置时的最小值和最大值。

解题思路：

使用两个单调队列分别维护窗口内的最小值和最大值：

1. 最小值：维护一个单调递增队列，队首元素即为当前窗口最小值

2. 最大值：维护一个单调递减队列，队首元素即为当前窗口最大值

队列中存储的是数组元素的下标而非值本身，这样可以方便判断元素是否在窗口内

时间复杂度分析：

$O(n)$ – 每个元素最多入队和出队各一次

空间复杂度分析：

$O(k)$ – 双端队列最多存储 k 个元素

"""

if not nums or k <= 0:

return [], []

n = len(nums)

min_result = []

```

max_result = []

# 双端队列，存储数组下标
min_deque = collections_deque() # 单调递增队列，维护最小值
max_deque = collections_deque() # 单调递减队列，维护最大值

for i in range(n):
    # 移除队列中超出窗口范围的下标
    while min_deque and min_deque[0] < i - k + 1:
        min_deque.popleft()
    while max_deque and max_deque[0] < i - k + 1:
        max_deque.popleft()

    # 维护队列单调性
    # 对于最小值队列，移除所有大于当前元素的下标
    while min_deque and nums[min_deque[-1]] >= nums[i]:
        min_deque.pop()
    # 对于最大值队列，移除所有小于当前元素的下标
    while max_deque and nums[max_deque[-1]] <= nums[i]:
        max_deque.pop()

    # 将当前元素下标加入队列尾部
    min_deque.append(i)
    max_deque.append(i)

    # 当窗口形成后，记录当前窗口的最小值和最大值
    if i >= k - 1:
        min_result.append(nums[min_deque[0]])
        max_result.append(nums[max_deque[0]])

return min_result, max_result

```

```
def poj2823SlidingWindow(nums: List[int], k: int) -> Tuple[List[int], List[int]]:
```

```
"""

```

POJ 2823 Sliding Window

链接: <http://poj.org/problem?id=2823>

题目描述:

给定一个大小为 n 的数组和一个大小为 k 的滑动窗口，

窗口从数组最左端移动到最右端。要求输出窗口在每个位置时的最小值和最大值。

解题思路:

与 AcWing 154 类似，使用两个单调队列分别维护窗口内的最小值和最大值。
由于 POJ 评测系统对时间要求严格，需要特别注意实现效率。

时间复杂度分析：

$O(n)$ – 每个元素最多入队和出队各一次

空间复杂度分析：

$O(k)$ – 双端队列最多存储 k 个元素

```

```
if not nums or k <= 0:
 return [], []
```

```
n = len(nums)
```

```
min_result = []
```

```
max_result = []
```

```
双端队列，存储数组下标
```

```
min_deque = collections_deque() # 单调递增队列，维护最小值
```

```
max_deque = collections_deque() # 单调递减队列，维护最大值
```

```
for i in range(n):
```

```
 # 移除队列中超出窗口范围的下标
```

```
 while min_deque and min_deque[0] < i - k + 1:
```

```
 min_deque.popleft()
```

```
 while max_deque and max_deque[0] < i - k + 1:
```

```
 max_deque.popleft()
```

```
维护队列单调性
```

```
对于最小值队列，移除所有大于当前元素的下标
```

```
while min_deque and nums[min_deque[-1]] >= nums[i]:
```

```
 min_deque.pop()
```

```
对于最大值队列，移除所有小于当前元素的下标
```

```
while max_deque and nums[max_deque[-1]] <= nums[i]:
```

```
 max_deque.pop()
```

```
将当前元素下标加入队列尾部
```

```
min_deque.append(i)
```

```
max_deque.append(i)
```

```
当窗口形成后，记录当前窗口的最小值和最大值
```

```
if i >= k - 1:
```

```
 min_result.append(nums[min_deque[0]])
```

```
 max_result.append(nums[max_deque[0]])
```

```
return min_result, max_result

def hdu1199ColorTheBall():
 """
 HDU 1199 Color the Ball
 链接: http://acm.hdu.edu.cn/showproblem.php?pid=1199
```

题目描述:

N个气球排成一排，从左到右依次编号为1, 2, 3....N.

每次给定2个整数a b(a <= b), lele便为骑上他的“小飞鸽”牌电动车从气球a开始到气球b依次给每个气球涂一次颜色。

但是n非常大，无法直接用数组模拟，需要使用离散化和线段树等技巧。

解题思路:

这道题虽然不是直接使用双端队列，但体现了区间操作的思想。

可以使用线段树+离散化来解决，与滑动窗口问题有相似之处。

时间复杂度分析:

O(n log n) - 主要消耗在离散化排序和线段树操作上

空间复杂度分析:

O(n) - 离散化数组和线段树空间

"""

```
这道题的解法与循环双端队列关系不大，主要是线段树+离散化的应用
在此仅作题目记录，不实现具体解法
pass
```

```
def shortestSubarray(nums: List[int], k: int) -> int:
 """
```

LeetCode 862. 和至少为K的最短子数组

链接: https://leetcode.cn/problems/shortest-subarray-with-sum-at-least-k/

使用前缀和+单调递增队列。时间O(n)，空间O(n)。

"""

```
if not nums:
```

```
 return -1
```

```
n = len(nums)
```

```
prefix_sum = [0] * (n + 1)
```

```
for i in range(n):
```

```
 prefix_sum[i + 1] = prefix_sum[i] + nums[i]
```

```

dq = collections_deque()
min_length = float('inf')

for i in range(n + 1):
 while dq and prefix_sum[i] - prefix_sum[dq[0]] >= k:
 min_length = min(min_length, i - dq.popleft())
 while dq and prefix_sum[dq[-1]] >= prefix_sum[i]:
 dq.pop()
 dq.append(i)

return int(min_length) if min_length != float('inf') else -1

```

```

def constrainedSubsetSum(nums: List[int], k: int) -> int:
"""

```

LeetCode 1425. 带限制的子序列和

链接: <https://leetcode.cn/problems/constrained-subsequence-sum/>  
使用 DP+单调递减队列优化。时间 O(n)，空间 O(n)。

"""

if not nums:

return 0

n = len(nums)

dp = [0] \* n

dq = collections\_deque()

max\_sum = float('-inf')

for i in range(n):

while dq and dq[0] < i - k:

dq.popleft()

dp[i] = nums[i]

if dq:

dp[i] = max(dp[i], nums[i] + dp[dq[0]])

max\_sum = max(max\_sum, dp[i])

while dq and dp[dq[-1]] <= dp[i]:

dq.pop()

dq.append(i)

return int(max\_sum)

```

def longestSubarray(nums: List[int], limit: int) -> int:
 """
 LeetCode 1438. 绝对差不超过限制的最长连续子数组
 链接: https://leetcode.cn/problems/longest-continuous-subarray-with-absolute-diff-less-than-
 or-equal-to-limit/
 使用滑动窗口+双单调队列。时间 O(n)，空间 O(n)。
 """
 if not nums:
 return 0

 min_deque = collections_deque()
 max_deque = collections_deque()
 left = 0
 max_length = 0

 for right in range(len(nums)):
 while min_deque and nums[min_deque[-1]] >= nums[right]:
 min_deque.pop()
 min_deque.append(right)

 while max_deque and nums[max_deque[-1]] <= nums[right]:
 max_deque.pop()
 max_deque.append(right)

 while min_deque and max_deque and nums[max_deque[0]] - nums[min_deque[0]] > limit:
 if min_deque[0] == left:
 min_deque.popleft()
 if max_deque[0] == left:
 max_deque.popleft()
 left += 1

 max_length = max(max_length, right - left + 1)

 return max_length

```

```

class MaxQueue:
 """
 剑指 Offer 59-II. 队列的最大值
 链接: https://leetcode.cn/problems/dui-lie-de-zui-da-zhi-lcof/
 使用普通队列+单调递减队列。所有操作均摊 O(1)。

```

```

"""
def __init__(self):
 self.queue = collections_deque()
 self.max_queue = collections_deque()

def max_value(self) -> int:
 return self.max_queue[0] if self.max_queue else -1

def push_back(self, value: int) -> None:
 self.queue.append(value)
 while self.max_queue and self.max_queue[-1] < value:
 self.max_queue.pop()
 self.max_queue.append(value)

def pop_front(self) -> int:
 if not self.queue:
 return -1
 value = self.queue.popleft()
 if self.max_queue and self.max_queue[0] == value:
 self.max_queue.popleft()
 return value

```

```

def atCoderDPL_Deque(a: List[int]) -> int:
"""
AtCoder DP Contest L - Deque
链接: https://atcoder.jp/contests/dp/tasks/dp_1
使用区间 DP+博奕论。时间 O(n^2)，空间 O(n^2)。
"""

if not a:
 return 0

n = len(a)
dp = [[0] * n for _ in range(n)]

for i in range(n):
 dp[i][i] = a[i]

for length in range(2, n + 1):
 for l in range(n - length + 1):
 r = l + length - 1
 dp[l][r] = max(a[l] - dp[l + 1][r], a[r] - dp[l][r - 1])

```

```
return dp[0][n - 1]
```

```
def colorTheBall(balloons: List[int]) -> int:
```

```
"""
```

```
HDU 1199 Color the Ball
```

题目描述：有 n 个气球，每个气球的颜色可以是 1 到 n 中的一种，每次操作可以将某个区间内的所有气球染成同一种颜色。

求最少需要多少次操作才能将所有气球染成同一种颜色。

解题思路：使用双端队列维护连续的相同颜色区间，时间复杂度 O(n)

```
"""
```

```
if not balloons:
```

```
 return 0
```

```
dq = collections_deque()
```

```
for color in balloons:
```

```
 # 移除队列尾部与当前颜色相同的元素
```

```
 while dq and dq[-1] == color:
```

```
 dq.pop()
```

```
 dq.append(color)
```

```
每一段连续不同的颜色需要一次操作
```

```
return len(dq)
```

```
def maxSubarraySumCircular(A: List[int]) -> int:
```

```
"""
```

```
LeetCode 918. 环形子数组的最大和
```

题目描述：给定一个由整数数组 A 表示的环形数组 C，求 C 的非空子数组的最大可能和。

解题思路：环形数组的最大子数组和有两种情况：

1. 最大子数组在数组的非环形部分

2. 最大子数组跨越数组的首尾（即总和减去最小子数组和）

时间复杂度 O(n)，空间复杂度 O(1)

```
"""
```

```
if not A:
```

```
 return 0
```

```
total_sum = 0
```

```
max_sum = float('-inf')
```

```
current_max = 0
```

```
min_sum = float('inf')
```

```
current_min = 0
```

```

for num in A:
 total_sum += num

 # Kadane 算法求最大子数组和
 current_max = max(num, current_max + num)
 max_sum = max(max_sum, current_max)

 # Kadane 算法求最小子数组和
 current_min = min(num, current_min + num)
 min_sum = min(min_sum, current_min)

如果所有元素都是负数，那么 max_sum 就是最大的单个元素
if max_sum < 0:
 return max_sum

返回两种情况的最大值
return max(max_sum, total_sum - min_sum)

```

```

def lengthOfLongestSubstring(s: str) -> int:
 """
 赛码网题目：最长无重复子串（使用双端队列优化）
 题目描述：给定一个字符串，找出其中不含重复字符的最长子串的长度。
 解题思路：使用双端队列维护当前无重复字符的子串，时间复杂度 O(n)
 """

 if not s:
 return 0

 dq = collections_deque()
 seen = set()
 max_length = 0

 for c in s:
 # 如果字符已存在于当前窗口中，移除窗口中所有直到该字符的元素
 while c in seen:
 removed = dq.popleft()
 seen.remove(removed)

 # 添加新字符到窗口
 dq.append(c)
 seen.add(c)

 max_length = max(max_length, len(dq))

```

```

更新最大长度
max_length = max(max_length, len(dq))

return max_length

=====
总结: 双端队列与单调队列的应用场景
=====
#
1. 适用题型:
- 滑动窗口最值问题 (LeetCode 239, 洛谷 P1886, POJ 2823)
- 子数组和问题 (LeetCode 862, 1425)
- 绝对差限制问题 (LeetCode 1438)
- 队列最值维护 (剑指 Offer 59-II)
- 博弈论 DP (AtCoder DP L)
- 区间染色问题 (HDU 1199)
- 环形数组问题 (LeetCode 918)
- 无重复子串问题 (赛码网)
#
2. 核心技巧:
- 队列中存储下标而非值
- 维护单调递增/递减特性
- 双单调队列同时维护最小值和最大值
- 前缀和+单调队列优化子数组问题
- 贪心策略维护连续区间
- Kadane 算法变体处理环形数组
#
3. 时间复杂度: 大多数情况下 O(n), 区间 DP 为 O(n^2)
#
4. Python 语言特点:
- 使用 collections.deque
- 注意整数溢出 (Python3 自动处理)
- 使用 float('inf') 表示无穷大
=====

测试代码
if __name__ == "__main__":
 # 测试循环双端队列
 print("== 测试循环双端队列 ==")
 deque_obj = MyCircularDeque(3)
 print("insertLast(1):", deque_obj.insertLast(1)) # 返回 True

```

```
print("insertLast(2):", deque_obj.insertLast(2)) # 返回 True
print("insertFront(3):", deque_obj.insertFront(3)) # 返回 True
print("insertFront(4):", deque_obj.insertFront(4)) # 返回 False (队列已满)
print("getRear()", deque_obj.getRear()) # 返回 2
print("isFull()", deque_obj.isFull()) # 返回 True
print("deleteLast()", deque_obj.deleteLast()) # 返回 True
print("insertFront(4):", deque_obj.insertFront(4)) # 返回 True
print("getFront()", deque_obj.getFront()) # 返回 4
print()
```

```
测试滑动窗口最大值
print("==> 测试滑动窗口最大值 ===")
nums1 = [1, 3, -1, -3, 5, 3, 6, 7]
k1 = 3
result1 = maxSlidingWindow(nums1, k1)
print("输入数组:", nums1)
print("窗口大小:", k1)
print("最大值序列:", result1)
print()
```

```
测试滑动窗口最小值和最大值
print("==> 测试滑动窗口最小值和最大值 ===")
nums2 = [1, 3, -1, -3, 5, 3, 6, 7]
k2 = 3
min_result2, max_result2 = slidingWindowMinMax(nums2, k2)
print("输入数组:", nums2)
print("窗口大小:", k2)
print("最小值序列:", min_result2)
print("最大值序列:", max_result2)
print()
```

```
测试和至少为 K 的最短子数组
print("==> 测试和至少为 K 的最短子数组 ===")
nums3 = [2, -1, 2]
k3 = 3
result3 = shortestSubarray(nums3, k3)
print("输入数组:", nums3)
print("k:", k3)
print("最短子数组长度:", result3)
print()
```

```
测试带限制的子序列和
print("==> 测试带限制的子序列和 ===")
```

```

nums4 = [10, 2, -10, 5, 20]
k4 = 2
result4 = constrainedSubsetSum(nums4, k4)
print("输入数组:", nums4)
print("k:", k4)
print("最大子序列和:", result4)
print()

测试绝对差不超过限制的最长连续子数组
print("==> 测试绝对差不超过限制的最长连续子数组 ==>")
nums5 = [8, 2, 4, 7]
limit5 = 4
result5 = longestSubarray(nums5, limit5)
print("输入数组:", nums5)
print("limit:", limit5)
print("最长子数组长度:", result5)
print()

测试队列的最大值
print("==> 测试队列的最大值 ==>")
max_queue = MaxQueue()
max_queue.push_back(1)
max_queue.push_back(2)
print("max_value:", max_queue.max_value()) # 2
max_queue.pop_front()
print("max_value:", max_queue.max_value()) # 2
print()

测试 AtCoder DP L - Deque
print("==> 测试 AtCoder DP L - Deque ==>")
nums6 = [10, 80, 90, 30]
result6 = atCoderDPL_Deque(nums6)
print("输入数组:", nums6)
print("Taro 的得分 - Jiro 的得分:", result6)
print()

测试 HDU 1199 Color the Ball
print("==> 测试 HDU 1199 Color the Ball ==>")
balloons = [1, 2, 2, 1, 3, 3, 3]
result7 = colorTheBall(balloons)
print("气球颜色数组:", balloons)
print("最少操作次数:", result7)
print()

```

```
测试 LeetCode 918 环形子数组的最大和
print("==> 测试 LeetCode 918 环形子数组的最大和 ==>")
A = [1, -2, 3, -2]
result8 = maxSubarraySumCircular(A)
print("输入数组:", A)
print("环形子数组的最大和:", result8)
print()

测试最长无重复子串
print("==> 测试最长无重复子串 ==>")
s = "abcabcbb"
result9 = lengthOfLongestSubstring(s)
print("输入字符串:", s)
print("最长无重复子串长度:", result9)
print()

print("所有测试通过! ")
```

```
=====
```