算法Project4 318104584 毛雨帆

Part1：Leetcode 题目（Hard）

滑动窗口系列问题（最大值，中位数，线性时间解决最大值问题）

注：笔者在做题过程中，发现了这一个系列的滑动窗口问题，彼此之间极为相似，解法也可互相借鉴，故作为一个完整系列来解。

（1）滑动窗口最大值

给定一个数组 nums 和滑动窗口的大小 k，请找出所有滑动窗口里的最大值。

示例:

输入: nums = [1,3,-1,-3,5,3,6,7], 和 k = 3
输出: [3,3,5,5,6,7] **示例：**

给出 *nums* = `[1,3,-1,-3,5,3,6,7]`，以及 *k* = 3。

```
窗口位置                      最大值
---------------               -----
[1  3  -1] -3  5  3  6  7       3
 1 [3  -1  -3] 5  3  6  7       3
 1  3 [-1  -3  5] 3  6  7       5
 1  3  -1 [-3  5  3] 6  7       5
 1  3  -1  -3 [5  3  6] 7       6
 1  3  -1  -3  5 [3  6  7]      7
```

因此，返回该滑动窗口的中位数数组 `[1,-1,-1,3,5,6]`。

(2)滑动窗口中位数

中位数是有序序列最中间的那个数。如果序列的大小是偶数，则没有最中间的数；此时中位数是最中间的两个数的平均数。

例如：

- `[2,3,4]`，中位数是 `3`
- `[2,3]`，中位数是 `(2 + 3) / 2 = 2.5`

给你一个数组 *nums*，有一个大小为 *k* 的窗口从最左端滑动到最右端。窗口中有 *k* 个数，每次窗口向右移动 *1* 位。你的任务是找出每次窗口移动后得到的新窗口中元素的中位数，并输出由它们组成的数组。

示例：

给出 *nums* = `[1,3,-1,-3,5,3,6,7]`，以及 *k* = 3。

```
窗口位置                      中位数
---------------               -----
[1  3  -1] -3  5  3  6  7       1
 1 [3  -1  -3] 5  3  6  7      -1
 1  3 [-1  -3  5] 3  6  7      -1
 1  3  -1 [-3  5  3] 6  7       3
 1  3  -1  -3 [5  3  6] 7       5
 1  3  -1  -3  5 [3  6  7]      6
```

因此，返回该滑动窗口的中位数数组 `[1,-1,-1,3,5,6]`。

（3）滑动窗口最大值进阶

设计一个在线性时间复杂度的算法，完成问题（1）

为方便阅读，将算法思路的描述与代码实现放在一个part

Part2：算法设计与代码实现（C++）

问题一：滑动窗口最大值

方法一：暴力搜索法

设窗口区间为 [i, j]，最大值为x{j}。当窗口向前移动一格，则区间变为 [i+1,j+1]，即添加了nums[j+1]，删除了nums[i]。

若只向窗口 [i, j] 右边添加数字nums[j+1]，则新窗口最大值可以通过一次对比，使用O(1) 时间得到，即：

x{j+1} = max(x{j}, nums[j + 1]).

而由于删除的nums[i]可能恰好是窗口内唯一的最大值x{j}，因此不能通过以上方法计算x{j+1}，而必须使用 O(j-i)时间， 遍历整个窗口区间 获取最大值，即：

x{j+1} = max(nums(i+1),..... , num(j+1))

设数组nums 的长度为n，则共有(n−k+1) 个窗口；获取每个窗口最大值需线性遍历，时间复杂度为O(k)。

前一个窗口的最大值和现在窗口的最大值其实是有关系的，关系如下：其实前一个窗口和现在窗口只相差了 左指针指向的数字 和 右指针指向的数字 其他数字都是相等的所以只需要根据左指针指向的数字 和 新的右指针指向的数字来判断一下是否需要更新最大值即可
分为以下两种情况：
1、首先让右指针+1，如果这时候新右指针指向的值大于了原来的最大值，直接更新，跳到下次循环即可
2、否则，判断左指针是否等于原来的最大值。如果否，很明显最大值位于新窗口和老窗口重叠部分前一次的最大值也是当前的最大值，不需要更新。否则跳至第三步
3、这时候不知道新窗口的最大值是什么，所以需要遍历一次，保存后继续循环即可

根据以上分析，可得暴力法的时间复杂度为 O((n-k+1)*k) ≈O(nk)。

```
class Solution {
public:
    int FindMax(const vector<int>& nums, const int &i, const int &j)
    {
    int maxval = nums[i];
    for (int k = i + 1; k <= j; ++k)
    {
        maxval = max(maxval, nums[k]);
    }
    return maxval;
    }

    vector<int> maxSlidingWindow(vector<int>& nums, int k) {
       vector<int> ans;
```

```
    if(nums.size()==0)  return ans;
    int maxval = FindMax(nums, 0, k - 1);
    int i = 0;
    int j = k-1;
    ans.push_back(maxval);
    while ( (++j) != nums.size())
    {
        ++i;
        if (nums[j] >= maxval)
        //先判断新加入的值是否为最大值在某些情况下比直接FindMax的效率要高一点
            maxval = nums[j];
        else if (nums[i-1] == maxval)
        //新加入的值不是最大值 且除去的值是最大值  则需要在范围内寻找最大值
            maxval = FindMax(nums, i, j);
        ans.push_back(maxval);
    }
    return ans;
    }
};
```

问题二：滑动窗口中位数

方法一：插入排序二分查找
要想找中位数，那最基本的方法当然是在排序好了的数组中然后找中位数。

eg：数组是偶数的情况下[1 2 3 4]中位数是(2+3)/2=2.5；数组是奇数的情况下[1 2 3]中位数是2
如何排序？而且是不断插入和删除的过程中？就想到了比较基础的排序方法：插入排序。那么优化一下插排，结合二分查找，采用lower_bound或upper_bound来查找插入和删除的位置.

再考虑容器的选择问题

vector：支持随机访问，但是我们要插入和删除，数组的插入删除复杂度最坏是可以到达O(n)的，不推荐；list：十分方便插入和删除，但是不支持随机访问，不推荐；deque：底层十分复杂，但是插入删除复杂度都是与插入删除的个数呈线性关系，且支持随机访问，因此采用deque。

```
class Solution {
public:
    vector<double> medianSlidingWindow(vector<int>& nums, int k) {
        int nums_size=nums.size();
        deque<int> window;
        vector<double> ans;
        ans.reserve(nums_size-k+1);
        for (int i=0;i<k;++i)
        {
            deque<int>::iterator
insert_p=upper_bound(window.begin(),window.end(),nums.at(i));
            window.insert(insert_p,nums.at(i));
        }
        if (k%2==0)
            ans.push_back(((double)window.at(k/2-1)+window.at(k/2))/2.0);
        else
            ans.push_back(window.at(k/2));
        for (int i=k;i<nums_size;++i)
        {
            deque<int>::iterator
delete_p=lower_bound(window.begin(),window.end(),nums.at(i-k));
```

```
            window.erase(delete_p);
            deque<int>::iterator
insert_p=upper_bound(window.begin(),window.end(),nums.at(i));
            window.insert(insert_p,nums.at(i));
            if (k%2==0)
                ans.push_back(((double)window.at(k/2-1)+window.at(k/2))/2.0);
            else
                ans.push_back(window.at(k/2));
        }
        return ans;
    }
};
```

方法二：多重集合 + 迭代器

这种方法基于特定的语言，即 C++ 中的 multiset（多重集合）数据结构。我们使用一个多重集合和一个迭代器（iterator），其中迭代器指向集合中的中位数。当我们添加或删除元素时，我们修改迭代器的指向，保证其仍然指向中位数。下面给出了我们的算法：

我们维护多重集合 window 的迭代器 mid；

首先我们在 window 中加入前 k 个元素，并让 mid 指向 window 中的第 \lfloor k/2 \rfloor⌊k/2⌋ 个元素（从 0 开始计数）；

当我们在 window 中加入数 num 时：

如果 num < *mid，那么我们需要将 mid 往前移；

如果 num >= mid，我们不需要对 mid 进行任何操作。

当我们在 windows 中删除数 num 时：

如果 num < *mid，我们需要将 mid 先往后移，再删除 num；

如果 num > *mid，我们不需要对 mid 进行任何操作；

如果 num == *mid，我们需要找到 num 第一次出现的位置对应的迭代器（使用 lower_bound()）并删除，

而不是删除 mid 对应的　　数。随后和 num < *mid 的处理方式相同。

复杂度分析

- 时间复杂度O(*N*log*k*)，其中 N 是数组的长度。
- 空间复杂度：O(N)。

```
vector<double> medianSlidingWindow(vector<int>& nums, int k)
{
    vector<double> medians;
    multiset<int> window(nums.begin(), nums.begin() + k);
    auto mid = next(window.begin(), k / 2);

for (int i = k;; i++) {

    // Push the current median
    medians.push_back(((double)(*mid) + *next(mid, k % 2 - 1)) * 0.5);

    // If all done, break
    if (i == nums.size())
```

```
            break;

        // Insert incoming element
        window.insert(nums[i]);
        if (nums[i] < *mid)
            mid--;                  // same as mid = prev(mid)

        // Remove outgoing element
        if (nums[i - k] <= *mid)
            mid++;                  // same as mid = next(mid)

        window.erase(window.lower_bound(nums[i - k]));
    }

    return medians;
    }
```

问题三：滑动窗口最大值进阶（线性时间复杂度）

方法一：单调队列

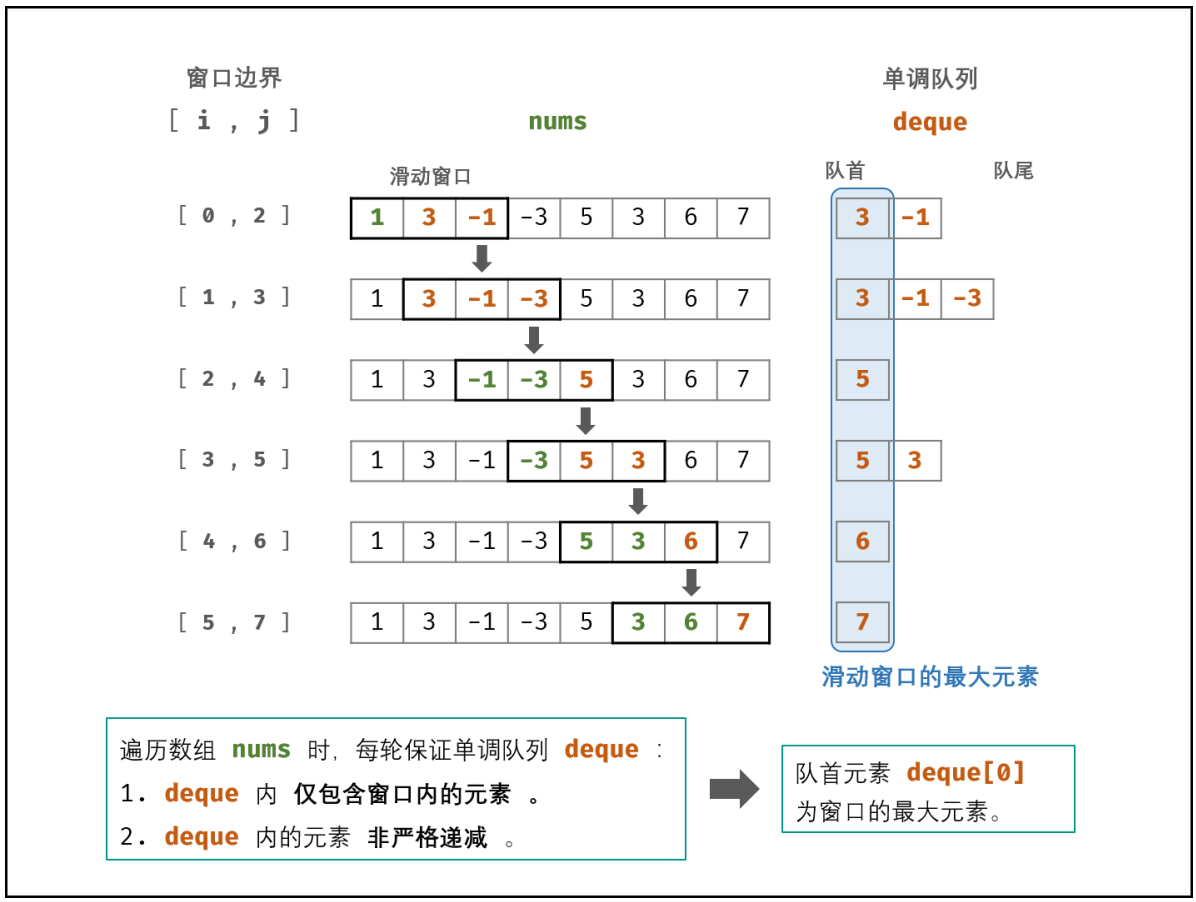
为了降低时间复杂度，我们观察一下窗口移动的过程类似于队列出队入队的过程，每次队尾出一个元素，然后队头插入一个元素，求该队列中的最大值。

我们可以维护一个递减队列，队列用来保存可能是最大值的数字的index。当前窗口最大值的index在队首，当窗口滑动时，会进入一个新值，出去一个旧值，我们需要给出当前窗口的最大值。

这个队列应该长这个样子：

```
class MyQueue {
public:
    void pop(int value) {
    }
    void push(int value) {
    }
    int front() {C
        return que.front();
    }
};
```

需要先检查队首（上一窗口的最大值）的index是否还在当前窗口内，如果不在的话需要淘汰。然后新进入的值要和队尾元素做比较，如果比队尾元素大，那么队尾元素出队（用到双端队列特性的地方），直到队列为空或者前面的值不比他小为止。



队列的设计思路如下：

每次窗口移动的时候，调用que.pop(滑动窗口中移除元素的数值)，que.push(滑动窗口添加元素的数值)，然后que.front()就返回我们要的最大值。其实队列没有必要维护窗口里的所有元素，只需要维护有可能成为窗口里最大值的元素就可以了，同时保证队里里的元素数值是由大到小的。那么这个维护元素单调递减的队列就叫做单调队列，即单调递减或单调递增的队列。

C++中没有直接支持单调队列，需要我们自己实现一个单调队列，此处用C++中的deque容器来实现，

具体方法如下：

设计单调队列的时候，pop，和push操作要保持如下规则：1.pop(value)：如果窗口移除的元素value等于单调队列的出口元素，那么队列弹出元素，否则不用任何操作。2.push(value)：如果push的元素value大于入口元素的数值，那么就将队列出口的元素弹出，直到push元素的数值小于等于队列入口元素的数值为止。

保持如上规则，每次窗口移动的时候，只要问que.front()就可以返回当前窗口的最大值。

使用单调队列的时间复杂度是 O(n)

```
class Solution {
public:
    class MyQueue { //单调队列（从大到小）
    public:
        deque<int> que; // 使用deque来实现单调队列
        void pop(int value) {
            if (!que.empty() && value == que.front()) {
                que.pop_front();
            }
        }
        void push(int value) {
            while (!que.empty() && value > que.back()) {
                que.pop_back();
```

```
            }
            que.push_back(value);

        }
        int front() {
            return que.front();
        }
    };
    vector<int> maxSlidingWindow(vector<int>& nums, int k) {
        MyQueue que;
        vector<int> result;
        if (nums.empty()) {
            return result;
        }
        for (int i = 0; i < k; i++) { // 先将前k的元素放进队列
            que.push(nums[i]);
        }
        result.push_back(que.front()); // result 记录前k的元素的最大值
        for (int i = k; i < nums.size(); i++) {
            que.pop(nums[i - k]); // 模拟滑动窗口的移动
            que.push(nums[i]); // 模拟滑动窗口的移动
            result.push_back(que.front()); // 记录对应的最大值
        }
        return result;
    }
};
```

方法二：变治法+线性递归

将数组分成大小相等的块，每个块都可以理解为有两个数组 left 和 right。left 方向从左到右，right 相反。left[i]是指块从开始到下标 i 的最大元素，right[j]是指块从开始到下标 j 的最大元素。假设滑动窗口的范围是[i, j]，很容易看出来，滑动窗口中的最大值就是max(right[i], left[j])。这里问题进行了变治。接下来用线性递归（减治）思想解决。

```
var maxSlidingWindow = function(nums, k) {
    if (k === 1) return nums;
    const length = nums.length;
    if (!length) return [];

    const left = new Array(length);
    const right = new Array(length);

    left[0] = nums[0];
    right[length - 1] = nums[length - 1];
    for (let i = 1; i < length; ++i) {
        if (i % k) {
            left[i] = Math.max(nums[i], left[i - 1]);
        } else {
            left[i] = nums[i];
        }

        let j = length - i - 1;
        if ((j + 1) % k) {
            right[j] = Math.max(nums[j], right[j + 1]);
        } else {
```

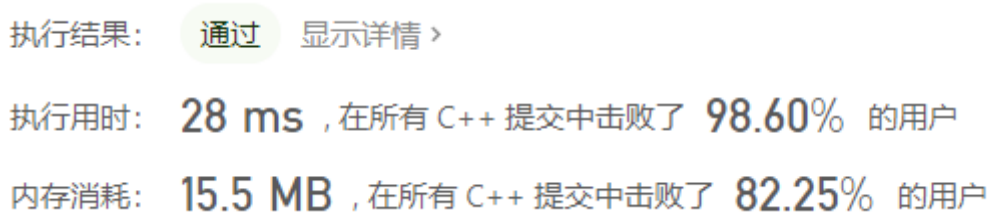
```
            right[j] = nums[j];
        }
    }

    const res = [];
    for (let i = 0; i < length - k + 1; i++) {
        res.push(Math.max(right[i], left[i + k - 1]));
    }
    return res;
};
```

Part3:实验结果与分析

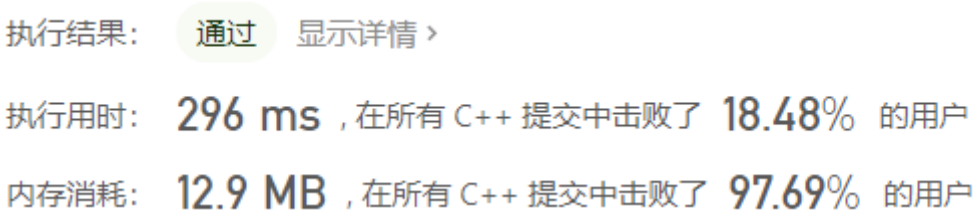
问题一

方法一：暴力搜索法

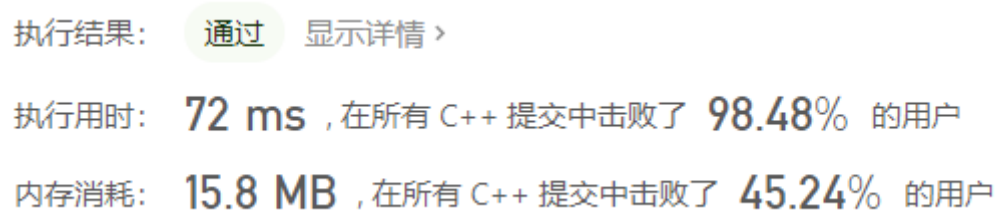


问题二

方法一：插入排序（二分查找优化）

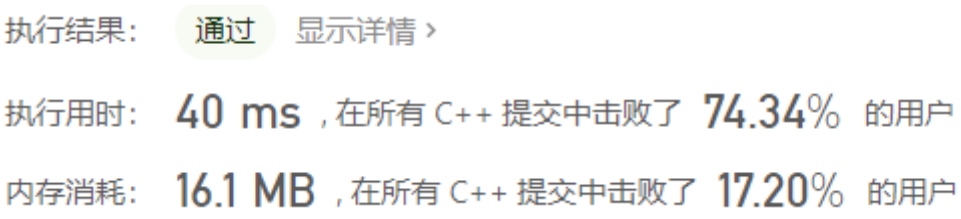


方法二：多重集合 + 迭代器

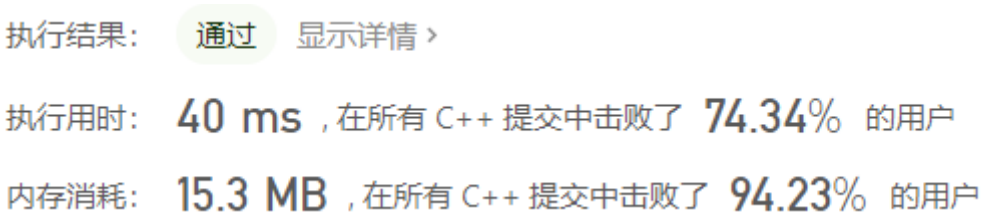


问题三

方法一：单调队列



方法二：变治法+线性递归


执行结果：
通过
显示详情 >
执行用时：
40 ms
，在所有 C++ 提交中击败了
74.34%
的用户
内存消耗：
15.3 MB
，在所有 C++ 提交中击败了
94.23%
的用户