

master ▾

...

leetcode-master / problems / 0239.滑动窗口最大值.md



KailokFung fix(0239): 新增一种java写法

History

6 contributors



460 lines (364 sloc) | 17 KB

...

欢迎大家[参与本项目](#)，贡献其他语言版本的代码，拥抱开源，让更多学习算法的小伙伴们收益！

要用啥数据结构呢？


239. 滑动窗口最大值

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

给定一个数组 `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。

进阶：

你能在线性时间复杂度内解决此题吗？ 

提示：

$1 \leq \text{nums.length} \leq 10^5$ $10^4 \leq \text{nums}[i] \leq 10^4$ $1 \leq k \leq \text{nums.length}$

思路

这是使用单调队列的经典题目。

难点是如何求一个区间里的最大值呢？（这好像是废话），暴力一下不就得了。

暴力方法，遍历一遍的过程中每次从窗口中找到最大的数值，这样很明显是 $O(n * k)$ 的算法。

有的同学可能会想用一个大顶堆（优先级队列）来存放这个窗口里的k个数字，这样就可以知道最大的最大值是多少了，但是问题是这个窗口是移动的，而大顶堆每次只能弹出最大值，我们无法移除其他数值，这样就造成大顶堆维护的不是滑动窗口里面的数值了。所以不能用大顶堆。

此时我们需要一个队列，这个队列呢，放进去窗口里的元素，然后随着窗口的移动，队列也一进一出，每次移动之后，队列告诉我们里面的最大值是什么。

这个队列应该长这个样子：

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

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。

这么个队列香不香，要是现成的这种数据结构是不是更香了！

可惜了，没有！我们需要自己实现这么个队列。

然后在分析一下，队列里的元素一定是要排序的，而且要最大值放在出队口，要不然怎么知道最大值呢。

但如果把窗口里的元素都放进队列里，窗口移动的时候，队列需要弹出元素。

那么问题来了，已经排序之后的队列 怎么能把窗口要移除的元素（这个元素可不一定是最大值）弹出呢。

大家此时应该陷入深思.....

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，需要我们自己来一个单调队列

不要以为实现的单调队列就是对窗口里面的数进行排序，如果排序的话，那和优先级队列又有什么区别了呢。

来看一下单调队列如何维护队列里的元素。

动画如下：

 239.滑动窗口最大值

对于窗口里的元素{2, 3, 5, 1, 4}，单调队列里只维护{5, 4}就够了，保持单调队列里单调递减，此时队列出口元素就是窗口里最大元素。

此时大家应该怀疑单调队列里维护着{5, 4}怎么配合窗口经行滑动呢？

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，那么就将队列入口的元素弹出，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。

为了更直观的感受单调队列的工作过程，以题目示例为例，输入: nums = [1,3,-1,-3,5,3,6,7], 和 k = 3，动画如下：

239.滑动窗口最大值-2

那么我们用什么数据结构来实现这个单调队列呢？

使用deque最为合适，在文章[栈与队列：来看看栈和队列不为人知的一面](#)中，我们就提到了常用的queue在没有指定容器的情况下，deque就是默认底层容器。

基于刚刚说过的单调队列pop和push的规则，代码不难实现，如下：

```
class MyQueue { //单调队列（从大到小）
public:
    deque<int> que; // 使用deque来实现单调队列
    // 每次弹出的时候，比较当前要弹出的数值是否等于队列出口元素的数值，如果相等则弹出。
    // 同时pop之前判断队列当前是否为空。
    void pop(int value) {
        if (!que.empty() && value == que.front()) {
            que.pop_front();
        }
    }
    // 如果push的数值大于入口元素的数值，那么就将队列后端的数值弹出，直到push的数值小于等于队列入口元素的数值为止。
    // 这样就保持了队列里的数值是单调从大到小的了。
    void push(int value) {
        while (!que.empty() && value > que.back()) {
            que.pop_back();
        }
        que.push_back(value);
    }
    // 查询当前队列里的最大值 直接返回队列前端也就是front就可以了。
    int front() {
        return que.front();
    }
};
```

这样我们就用deque实现了一个单调队列，接下来解决滑动窗口最大值的问题就很简单了，直接看代码吧。

C++代码如下：

```
class Solution {
private:
    class MyQueue { //单调队列（从大到小）
    public:
        deque<int> que; // 使用deque来实现单调队列
        // 每次弹出的时候，比较当前要弹出的数值是否等于队列出口元素的数值，如果相等则弹出。
        // 同时pop之前判断队列当前是否为空。
        void pop(int value) {
            if (!que.empty() && value == que.front()) {
                que.pop_front();
            }
        }
        // 如果push的数值大于入口元素的数值，那么就将队列后端的数值弹出，直到push的数值小于等于队列
        // 这样就保持了队列里的数值是单调从大到小的了。
        void push(int value) {
            while (!que.empty() && value > que.back()) {
                que.pop_back();
            }
            que.push_back(value);
        }
        // 查询当前队列里的最大值 直接返回队列前端也就是front就可以了。
        int front() {
            return que.front();
        }
    };
public:
    vector<int> maxSlidingWindow(vector<int>& nums, int k) {
        MyQueue que;
        vector<int> 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]); // 滑动窗口移除最前面元素
            que.push(nums[i]); // 滑动窗口前加入最后面的元素
            result.push_back(que.front()); // 记录对应的最大值
        }
        return result;
    }
};
```

在来看一下时间复杂度，使用单调队列的时间复杂度是 $O(n)$ 。

有的同学可能想了，在队列中 push元素的过程中，还有pop操作呢，感觉不是纯粹的 $O(n)$ 。

其实，大家可以自己观察一下单调队列的实现，nums 中的每个元素最多也就被 push_back 和 pop_back 各一次，没有任何多余操作，所以整体的复杂度还是 $O(n)$ 。 pop_front

空间复杂度因为我们定义一个辅助队列，所以是 $O(k)$ 。

扩展

大家貌似对单调队列 都有一些疑惑，首先要明确的是，题解中单调队列里的pop和push接口，仅适用于本题哈。单调队列不是一成不变的，而是不同场景不同写法，总之要保证队列里单调递减或递增的原则，所以叫做单调队列。不要以为本地中的单调队列实现就是固定的写法哈。

大家貌似对deque也有一些疑惑，C++中deque是stack和queue默认的底层实现容器（这个我们之前已经讲过啦），**deque是可以两边扩展的，而且deque里元素并不是严格的连续分布的。**

其他语言版本

Java:

```
//解法一
//自定义数组
class MyQueue {
    Deque<Integer> deque = new LinkedList<>();
    //弹出元素时，比较当前要弹出的数值是否等于队列出口的数值，如果相等则弹出
    //同时判断队列当前是否为空
    void poll(int val) {
        if (!deque.isEmpty() && val == deque.peek()) {
            deque.poll();
        }
    }
    //添加元素时，如果要添加的元素大于入口处的元素，就将入口元素弹出
    //保证队列元素单调递减
    //比如此时队列元素3,1, 2将要入队，比1大，所以1弹出，此时队列：3,2
    void add(int val) {
        while (!deque.isEmpty() && val > deque.getLast()) {
            deque.removeLast();
        }
        deque.add(val);
    }
    //队列队顶元素始终为最大值
    int peek() {
        return deque.peek();
    }
}

class Solution {
    public int[] maxSlidingWindow(int[] nums, int k) {
        if (nums.length == 1) {
            return nums;
        }
        int len = nums.length - k + 1;
        //存放结果元素的数组
        int[] res = new int[len];
        int num = 0;
        //自定义队列
        MyQueue myQueue = new MyQueue();
        //先将前k的元素放入队列
        for (int i = 0; i < k; i++) {
            myQueue.add(nums[i]);
        }
    }
}
```

```

        res[num++] = myQueue.peek();
    }
    for (int i = k; i < nums.length; i++) {
        //滑动窗口移除最前面的元素，移除是判断该元素是否放入队列
        myQueue.poll(nums[i - k]);
        //滑动窗口加入最后面的元素
        myQueue.add(nums[i]);
        //记录对应的最大值
        res[num++] = myQueue.peek();
    }
    return res;
}
}

//解法二
//利用双端队列手动实现单调队列
/**
 * 用一个单调队列来存储对应的下标，每当窗口滑动的时候，直接取队列的头指针对应的值放入结果集即可
 * 单调队列类似 (tail -->) 3 --> 2 --> 1 --> 0 (--> head) (右边为头结点，元素存的是下标)
 */
class Solution {
    public int[] maxSlidingWindow(int[] nums, int k) {
        ArrayDeque<Integer> deque = new ArrayDeque<>();
        int n = nums.length;
        int[] res = new int[n - k + 1];
        int idx = 0;
        for(int i = 0; i < n; i++) {
            // 根据题意，i为nums下标，是要在[i - k + 1, k] 中选到最大值，只需要保证两点
            // 1.队列头结点需要在[i - k + 1, k]范围内，不符合则要弹出
            while(!deque.isEmpty() && deque.peek() < i - k + 1){
                deque.poll();
            }
            // 2.既然是单调，就要保证每次放进去的数字要比末尾的都大，否则也弹出
            while(!deque.isEmpty() && nums[deque.peekLast()] < nums[i]) {
                deque.pollLast();
            }

            deque.offer(i);

            // 因为单调，当i增长到符合第一个k范围的时候，每滑动一步都将队列头节点放入结果就行了
            if(i >= k - 1){
                res[idx++] = nums[deque.peek()];
            }
        }
        return res;
    }
}

```

Python:

```

class MyQueue: #单调队列 (从大到小
    def __init__(self):
        self.queue = [] #使用list来实现单调队列

    #每次弹出的时候，比较当前要弹出的数值是否等于队列出口元素的数值，如果相等则弹出。
    #同时pop之前判断队列当前是否为空。
    def pop(self, value):

```

```
if self.queue and value == self.queue[0]:
```

```
    self.queue.pop(0)#list.pop()时间复杂度为O(n),这里可以使用collections.deque()
```

#如果push的数值大于入口元素的数值，那么就将队列后端的数值弹出，直到push的数值小于等于队列入口
#这样就保持了队列里的数值是单调从大到小的了。

```
def push(self, value):
```

```
    while self.queue and value > self.queue[-1]:
```

```
        self.queue.pop()
```

```
    self.queue.append(value)
```

#查询当前队列里的最大值 直接返回队列前端也就是front就可以了。

```
def front(self):
```

```
    return self.queue[0]
```

```
class Solution:
```

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

```
        que = MyQueue()
```

```
        result = []
```

```
        for i in range(k): #先将前k的元素放进队列
```

```
            que.push(nums[i])
```

```
        result.append(que.front()) #result 记录前k的元素的最大值
```

```
        for i in range(k, len(nums)):
```

```
            que.pop(nums[i - k]) #滑动窗口移除最前面元素
```

```
            que.push(nums[i]) #滑动窗口前加入最后面的元素
```

```
            result.append(que.front()) #记录对应的最大值
```

```
        return result
```

Go:

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

```
    var queue []int
```

```
    var rtn []int
```

```
    for f := 0; f < len(nums); f++ {
```

```
        //维持队列递减，将 k 插入合适的位置，queue中 <=k 的元素都不可能是窗口中的最大值，直接弹出
```

```
        for len(queue) > 0 && nums[f] > nums[queue[len(queue)-1]] {
```

```
            queue = queue[:len(queue)-1]
```

```
        }
```

```
        // 等大的后来者也应入队
```

```
        if len(queue) == 0 || nums[f] <= nums[queue[len(queue)-1]] {
```

```
            queue = append(queue, f)
```

```
        }
```

```
        if f >= k - 1 {
```

```
            rtn = append(rtn, nums[queue[0]])
```

```
            //弹出离开窗口的队首
```

```
            if f - k + 1 == queue[0] {
```

```
                queue = queue[1:]
```

```
            }
```

```
        }
```

```
    }
```

```
    return rtn
```

```
}
```

```

// 封装单调队列的方式解题
type MyQueue struct {
    queue []int
}

func NewMyQueue() *MyQueue {
    return &MyQueue{
        queue: make([]int, 0),
    }
}

func (m *MyQueue) Front() int {
    return m.queue[0]
}

func (m *MyQueue) Back() int {
    return m.queue[len(m.queue)-1]
}

func (m *MyQueue) Empty() bool {
    return len(m.queue) == 0
}

func (m *MyQueue) Push(val int) {
    for !m.Empty() && val > m.Back() {
        m.queue = m.queue[:len(m.queue)-1]
    }
    m.queue = append(m.queue, val)
}

func (m *MyQueue) Pop(val int) {
    if !m.Empty() && val == m.Front() {
        m.queue = m.queue[1:]
    }
}

func maxSlidingWindow(nums []int, k int) []int {
    queue := NewMyQueue()
    length := len(nums)
    res := make([]int, 0)
    // 先将前k个元素放入队列
    for i := 0; i < k; i++ {
        queue.Push(nums[i])
    }
    // 记录前k个元素的最大值
    res = append(res, queue.Front())

    for i := k; i < length; i++ {
        // 滑动窗口移除最前面的元素
        queue.Pop(nums[i-k])
        // 滑动窗口添加最后面的元素
        queue.Push(nums[i])
        // 记录最大值
        res = append(res, queue.Front())
    }
}

```



```
    return res  
}
```

Javascript:

```
var maxSlidingWindow = function (nums, k) {  
    // 队列数组（存放的是元素下标，为了取值方便）  
    const q = [];  
    // 结果数组  
    const ans = [];  
    for (let i = 0; i < nums.length; i++) {  
        // 若队列不为空，且当前元素大于等于队尾所存下标的元素，则弹出队尾  
        while (q.length && nums[i] >= nums[q[q.length - 1]]) {  
            q.pop();  
        }  
        // 入队当前元素下标  
        q.push(i);  
        // 判断当前最大值（即队首元素）是否在窗口中，若不在便将其出队  
        while (q[0] <= i - k) {  
            q.shift();  
        }  
        // 当达到窗口大小时便开始向结果中添加数据  
        if (i >= k - 1) ans.push(nums[q[0]]);  
    }  
    return ans;  
};
```

-
- 作者微信: [程序员Carl](#)
 - B站视频: [代码随想录](#)
 - 知识星球: [代码随想录](#)