题目地址(239. 滑动窗口最大值)

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

入选理由

1. 双指针最后一种类型，滑动窗口。由于专题篇会继续，因此这里就只整一道

标签

- 双指针
- 滑动窗口

难度

- 困难

题目描述

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?

示例：

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

  滑动窗口的位置                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
```

提示：

```
1 <= nums.length <= 10^5
-10^4 <= nums[i] <= 10^4
1 <= k <= nums.length
```

前置知识

- 队列
- 滑动窗口

公司

- 阿里
- 腾讯
- 百度
- 字节

暴力

思路

题目很好理解，简单来说就是寻找所有窗口大小固定为 k 的滑动窗口内的最大值。

问题其实就是维护一个滑动窗口，每次获取滑动窗口最大值即可。

算法整体框架为：

```
function solution(nums, k) {
  const res = [];
  for (let i = 0; i <= nums.length - k; i++) {
    // maxInSlidingWindow 的功能是求数组 nums 中从索引 i 到 索引 i + k（两端包含）的最大值
    let curMax = maxInSlidingWindow(nums, i, i + k);
    res.push(curMax);
  }
  return res;
}
```

接下来就是考虑如何实现 `maxInSlidingWindow` 。不妨先从暴力解开始，我们线性枚举 i 到 i + k 的值找出最大的。

```
function maxInSlidingWindow(nums, start, end) {
  let max = -Infinity;
  for (let i = start; i < end; i++) {
    max = Math.max(nums[i], max);
  }
  return max;
}
```

我们来看下完整代码。

代码

代码支持：JS,Python3

JS Code：

```
var maxSlidingWindow = function (nums, k) {
  const res = [];
  for (let i = 0; i <= nums.length - k; i++) {
    let cur = maxInSlidingWindow(nums, i, i + k);
    res.push(cur);
  }
  return res;
};

function maxInSlidingWindow(nums, start, end) {
  let max = -Infinity;
  for (let i = start; i < end; i++) {
    max = Math.max(nums[i], max);
  }
  return max;
}
```

Python3:

```
class Solution:
    def maxSlidingWindow(self, nums: List[int], k: int) -> List[int]:
        if k == 0: return []
        res = []
        for r in range(k - 1, len(nums)):
            res.append(max(nums[r - k + 1:r + 1]))
        return res
```

复杂度分析

令 n 为数组长度

- 时间复杂度: $O(n * k)$
- 空间复杂度: $O(1)$

堆/优先队列

显然上面的解法不是题目要求的 O(n)，并且代入题目的数据范围多半是超时的，因此我们必须进行优化。

思路

求极值，特别是待求队列内容变动的场景下，用堆/优先队列是一种常见的方案。

这里可以对滑动窗口建立一个大小为 k 的**大顶堆**。窗口滑动时，从堆中去除一个滑动窗口**最前的一个数，添加滑动窗口后一个数**。取得窗口最大值，每次堆操作时间复杂度 $O(\log K)$

> 关于堆的具体内容，我们会在专题篇进行详细讲解。

代码

代码支持：Java,Python,JS

Java Code：

```
public ArrayList<Integer> maxInWindows2(int[] num, int size) {
        if (num == null || num.length == 0 || size <= 0 || num.length < size) {
            return new ArrayList<>();
        }
        ArrayList<Integer> result = new ArrayList<>();
        PriorityQueue<Integer> q = new PriorityQueue(size, Comparator.reverseOrder());
        for (int i = 0; i < num.length; i++) {
            if (q.size() == size) {
                q.remove(num[i - size]);
            }
            q.add(num[i]);
            if (i >= size - 1) {
                result.add(q.peek());
            }
        }
        int[] arr = new int[result.size()];
        for (int i = 0; i < result.size(); i++) {
            arr[i] = result.get(i);
```

```
        }
        return result;
    }
```

Python Code:

```python
class Solution:
    def maxSlidingWindow(self, nums: List[int], k: int) -> List[int]:
        h=[]
        ans=[]
        for i in range(k):
            heapq.heappush(h,(-nums[i],i))
        ans.append(-h[0][0])
        for i in range(k,len(nums)):
            heapq.heappush(h,(-nums[i],i))
            while h[0][1]<i-k+1:
                heapq.heappop(h)
            ans.append((-h[0][0]))
        return ans
```

JS Code:

```javascript
/**
 * @param {number[]} nums
 * @param {number} k
 * @return {number[]}
 */
var maxSlidingWindow = function (nums, k) {
  let ans = [];
  const pQ = new MaxPriorityQueue({ priority: (idx) => nums[idx] });
  let l = 0;
  for (let r = 0; r < nums.length; r++) {
    pQ.enqueue(r);
    if (r >= k - 1) {
      // console.log(pQ.toArray());
      while (pQ.front().element < l) {
        // not in window
        pQ.dequeue();
      }
      ans.push(nums[pQ.front().element]);
      l++;
    }
  }

  return ans;
};
```

复杂度分析

令 n 为数组长度

- 时间复杂度: $O(nlogk)$
- 空间复杂度: $O(k)$

单调队列

然而前面的两种解法都不是时间复杂度 $O(n)$ 的解法。接下来，我们考虑使用 $O(n)$ 的解法。

思路

其实，我们没必须存储窗口内的所有元素。 如果新进入的元素比前面的大，那么前面的元素就**不再有利用价值，可以直接移除**。这提示我们使用一个**单调递增栈**[1]来完成。

但由于窗口每次向右移动的时候，位于窗口最左侧的元素是需要被擦除的，而栈只能在一端进行操作。而如果你使用普通的数组实现，就是可以在另一端操作了，但是时间复杂度仍然是 $O(k)$，和上面的暴力算法时间复杂度一样。

因此，我们考虑使用链表来实现，维护两个指针分别指向头部和尾部即可，这样做的时间复杂度是 $O(1)$，这就是双端队列。

因此思路就是用一个双端队列来保存 接下来的滑动窗口可能成为最大值的数 。

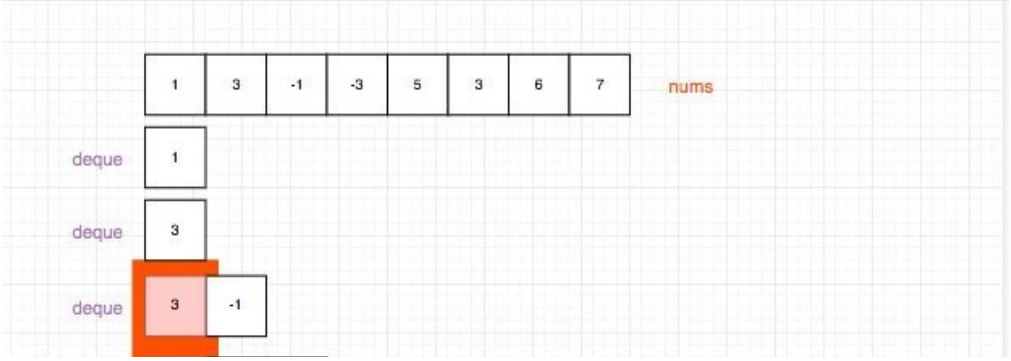
具体做法：

- 入队列
- 移除失效元素，失效元素有两种

1. 一种是已经超出窗口范围了，比如我遍历到第 4 个元素，k = 3，那么 i = 0 的元素就不应该出现在双端队列中了。具体就是 索引大于 `i - k + 1`的元素都应该被清除

2. 小于当前元素都没有利用价值了，具体就是 从后往前遍历（双端队列是一个递减队列）双端队列，如果小于当前元素就出队列

经过上面的分析，不难知道双端队列其实是一个递减的一个队列，因此队首的元素一定是最大的。用图来表示就是：



代码

代码支持：JS，Python3， CPP, Java

JS Code：

```
var maxSlidingWindow = function (nums, k) {
  const res = [];
  const dequeue = new Dequeue([]);
  // 前 k - 1 个数入队
  for (let i = 0; i < k - 1; i++) {
    dequeue.push(nums[i]);
  }

  // 滑动窗口
  for (let i = k - 1; i < nums.length; i++) {
    dequeue.push(nums[i]);
    res.push(dequeue.max());
    dequeue.shift(nums[i - k + 1]);
  }
  return res;
};

class Dequeue {
  constructor(nums) {
    this.list = nums;
  }

  push(val) {
    const nums = this.list;
    // 保证数据从队头到队尾递减
    while (nums[nums.length - 1] < val) {
      nums.pop();
    }
    nums.push(val);
  }

  // 队头出队
  shift(val) {
```

```
    let nums = this.list;
    if (nums[0] === val) {
      // 这里的js实现shift()理论上复杂度应该是O(k)，就不去真实实现一个O(1)出队的队列了，意思到位即可
      nums.shift();
    }
  }

  max() {
    return this.list[0];
  }
}
```

Python3:

```
class Solution:
    def maxSlidingWindow(self, nums: List[int], k: int) -> List[int]:
        q = collections.deque() # 本质就是单调队列
        ans = []
        for i in range(len(nums)):
            while q and nums[q[-1]] <= nums[i]: q.pop() # 维持单调性
            while q and i - q[0] >= k: q.popleft() # 移除失效元素
            q.append(i)
            if i >= k - 1: ans.append(nums[q[0]])
        return ans
```

CPP Code(by @yanglr):

```
class Solution {
public:
    vector<int> res;
    vector<int> maxSlidingWindow(vector<int>& nums, int k) {
        deque<int> q; // 双端队列，存储的是当前窗口最大值的索引，维护操作：保持单调递减，队头是最大值

        for (int i = 0; i < nums.size(); i++) {
            // 依次地将数组元素加入到队列中
            // 注意：确保队列元素间的距离都在k以内
            if (!q.empty() && i - k + 1 > q.front()) /* 倒着数第k个与队列开头数的index比较。窗口长度>k时，从队列中删掉最前面的数 */
                q.pop_front();
            while (!q.empty() && nums[i] >= nums[q.back()])
                q.pop_back();  /* 不断地把左侧比自己小的数从队列中删掉，遇到下一个比自己大的数时自己会被删掉。遇到比自己小的数得留着，最

            q.push_back(i);
            if (i >= k - 1)      // 只要窗口大小 ≥ k 时，窗口就会有最大值，将其放进res(结果vector)中
            {
                res.push_back(nums[q.front()]);
            }
        }
```

```
        return res;
    }
};
```

Java Code:

```
class Solution {
    public int[] maxSlidingWindow(int[] nums, int k) {
        int[] ans = new int[nums.length - k + 1];
        Deque<Integer> deque = new ArrayDeque<>();

        for (int i = 0; i < nums.length; i++){
            if (!deque.isEmpty() && deque.peekFirst() + k <= i) deque.pollFirst();
            while (!deque.isEmpty() && nums[deque.peekLast()] <= nums[i]){
                deque.pollLast();
            }
            deque.offerLast(i);
            if (i - k + 1 >= 0) ans[i - k + 1] = nums[deque.peekFirst()];
        }
        return ans;
    }
}
```

复杂度分析

令 n 为数组长度

- 时间复杂度: $O(n)$
- 空间复杂度: $O(k)$

参考

JS 的 deque 没有使用链表模拟，而是使用了数组。具体的链表实现可以参考**deque**

另外可以参考 **leetcode 仓库的题解**

参考资料

[1] 单调栈专题: *https://github.com/azl397985856/leetcode/blob/master/thinkings/monotone-stack.md*

上一页 下一页