- 一、随想录（共175道题）
  - 一、哈希表（8道题）
  - 二、链表（7道题）
  - 三、栈与队列（7道题）
    - 1、用栈实现队列（20230719，232题，简单）
    - 2、用队列实现栈（20230720，225题，简单）
    - 3、有效的括号（20230724，20题，简单）
    - 4、删除字符串中的所有相邻重复项（20230725，1047题，简单）
    - 5、逆波兰表达式求值（20230726，150题，中等）
    - 6、滑动窗口最大值（20230731，239题，困难）
    - 6、前K个高频元素（20230802，347题，中等）
  - 四、数组（5道题）
  - 五、字符串（7道题）
  - 六、双指针法（10道题）
  - 七、二叉树（32道题）
  - 八、动态规划（52道题）
  - 九、贪心算法（22道题）
  - 十、回溯算法（20道题）
  - 十一、单调栈（5道题）

一、随想录（共175道题）

一、哈希表（8道题）

二、链表（7道题）

三、栈与队列（7道题）

1、用栈实现队列（20230719，232题，简单）

请你仅使用两个栈实现先入先出队列。队列应当支持一般队列支持的所有操作（push、pop、peek、empty）：

实现 MyQueue 类：

- void push(int x) 将元素 x 推到队列的末尾
- int pop() 从队列的开头移除并返回元素
- int peek() 返回队列开头的元素
- boolean empty() 如果队列为空，返回 true ； 否则，返回 false

说明：

- 你 只能 使用标准的栈操作 —— 也就是只有 push to top, peek/pop from top, size, 和 is empty 操作是合法的。
- 你所使用的语言也许不支持栈。你可以使用 list 或者 deque（双端队列）来模拟一个栈，只要是标准的栈操作即可。

进阶：

- 你能否实现每个操作均摊时间复杂度为 O(1) 的队列？换句话说，执行 n 个操作的总时间复杂度为 O(n) ，即使其中一个操作可能花费较长时间。

> 自己看答案版本，对C++栈声明和函数不熟悉

```
class MyQueue {
public:

    stack<int> stackIn,stackOut;  //用两个栈实现，两个栈同等地位

    void Use2Tmp()   //不用参数，类中的函数可以调用成员变量
    {
        while(!stackIn.empty())
        {
            stackOut.push(stackIn.top());
            stackIn.pop();
        }
    }

    MyQueue() {

    }

    void push(int x) {        /*正常入栈*/
        stackIn.push(x);
    }

    int pop() {        /*使用两个栈A/B，A出现放入B，则B的栈顶即为队列头元素。B出栈，再入栈A*/
        if(stackOut.empty())
        {
            Use2Tmp();
        }
        int val = stackOut.top();
        stackOut.pop();          //栈pop返回的是什么，有返回值?
       return val;
    }

    int peek() { /*使用两个栈A/B，A出现放入B，则B的栈顶即为队列头元素*/
        if(stackOut.empty())
        {
            Use2Tmp();
        }
        return stackOut.top();
    }

    bool empty() {
        return (stackIn.empty() && stackOut.empty());
    }
};

/**
```

```
 * Your MyQueue object will be instantiated and called as such:
 * MyQueue* obj = new MyQueue();
 * obj->push(x);
 * int param_2 = obj->pop();
 * int param_3 = obj->peek();
 * bool param_4 = obj->empty();
 */
```

执行用时： **4 ms** ，在所有 C++ 提交中击败了 **33.48%** 的用户

内存消耗： **6.7 MB** ，在所有 C++ 提交中击败了 **91.89%** 的用户

通过测试用例： **22 / 22**

复杂度分析

- 时间复杂度：`push` 和 `empty` 为 $O(1)$，`pop` 和 `peek` 为均摊 $O(1)$。对于每个元素，至多入栈和出栈各两次，故均摊复杂度为 $O(1)$。
- 空间复杂度：$O(n)$。其中 n 是操作总数。对于有 n 次 `push` 操作的情况，队列中会有 n 个元素，故空间复杂度为 $O(n)$。

2、用队列实现栈（20230720，225题，简单）

请你仅使用两个队列实现一个后入先出（LIFO）的栈，并支持普通栈的全部四种操作（push、top、pop 和 empty）。

实现 MyStack 类：

- void push(int x) 将元素 x 压入栈顶。
- int pop() 移除并返回栈顶元素。
- int top() 返回栈顶元素。
- boolean empty() 如果栈是空的，返回 true ； 否则，返回 false 。

注意：

- 你只能使用队列的基本操作 —— 也就是 push to back、peek/pop from front、size 和 is empty 这些操作。
- 你所使用的语言也许不支持队列。 你可以使用 list （列表）或者 deque（双端队列）来模拟一个队列 , 只要是标准的队列操作即可。

进阶：你能否仅用一个队列来实现栈。

> 自己看答案版本，对C++队列声明和函数不熟悉

```
/*思路，双队列方法：
队列1为主队列，用于实现各种入栈出栈操作，队列2辅助队列。
每次入队，从队列2入，队列1依次出队列进入队列2。入队完成后，队列2即满足越靠前的是越后面进
```

```
来的。
最后将队列1和2互换名字，等待相应的操作指令。
*/
class MyStack {
public:
    queue<int> que1;
    queue<int> que2;

    MyStack() {

    }
    void push(int x) {        //从队列2入队列，然后将队列1依次入队列2
        que2.push(x);          /*que2.push() push函数需要入参*/
        while(!que1.empty())  /*que1.empty  empty是函数，别忘记加括号*/
        {
            que2.push(que1.front());   /*que1.top()  队列不是top，是front函数*/
            que1.pop();
        }
        swap(que1,que2);    /*swap()直接互换队列*/
    }
    int pop() {
        int tmpfront = que1.front();
        que1.pop();
        return tmpfront;
    }

    int top() {
        return que1.front();
    }

    bool empty() {
        return que1.empty();
    }
};

/**
 * Your MyStack object will be instantiated and called as such:
 * MyStack* obj = new MyStack();
 * obj->push(x);
 * int param_2 = obj->pop();
 * int param_3 = obj->top();
 * bool param_4 = obj->empty();
 */
```

执行用时： 0 ms ，在所有 C++ 提交中击败了 100.00% 的用户

内存消耗： 6.8 MB ，在所有 C++ 提交中击败了 11.46% 的用户

通过测试用例： 17 / 17

复杂度分析

- 时间复杂度：入栈操作 $O(n)$，其余操作都是 $O(1)$，其中 n 是栈内的元素个数。
  入栈操作需要将 $queue_1$ 中的 n 个元素出队，并入队 $n+1$ 个元素到 $queue_2$，共有 $2n+1$ 次操作，每次出队和入队操作的时间复杂度都是 $O(1)$，因此入栈操作的时间复杂度是 $O(n)$。
  出栈操作对应将 $queue_1$ 的前端元素出队，时间复杂度是 $O(1)$。
  获得栈顶元素操作对应获得 $queue_1$ 的前端元素，时间复杂度是 $O(1)$。
  判断栈是否为空操作只需要判断 $queue_1$ 是否为空，时间复杂度是 $O(1)$。
- 空间复杂度：$O(n)$，其中 n 是栈内的元素个数。需要使用两个队列存储栈内的元素。

3、有效的括号（20230724，20题，简单）

给定一个只包括 '('，')'，'{'，'}'，'['，']' 的字符串 s ，判断字符串是否有效。

有效字符串需满足：

- 左括号必须用相同类型的右括号闭合。
- 左括号必须以正确的顺序闭合。
- 每个右括号都有一个对应的相同类型的左括号。

> 自己写版本

```
/*思路：
1、使用栈来做，按顺序入栈，以各类左括号为基准，每次入栈，若是左括号，计数countA，
countB,countC，右边括号（思路不行）
2、遇到有括号，必须和上一个入栈的括号是匹配的，并且一起出栈，否则报错。
*/

class Solution {
public:
    stack<char> stk;
    bool isValid(string s) {
        if(s.empty())
        {
            return false;
        }
        int len = s.size();
        for(int i = 0; i < len; i++)
        {
            stk.push(s[i]);
            switch(s[i])      //Switch-case语句格式，Switch用花括号括起来{case 'A':
执行语句;  break; }
```

```
            {
                case '(':
                case '[':
                case '{':
                    continue;
                case ')':        //出栈，并判断下一个字符是否匹配此括号，匹配也出栈，否则false
                    stk.pop();
                    if(stk.empty())   //为什么一定要加这一句？下面的if判断不是已经保证top不为空了吗？？？
                    {
                        return false;
                    }
                    if(stk.top() != '(')  //包含了第一个字符就是右括号的情况
                    {
                        return false;
                    }
                    else
                    {
                        stk.pop();
                    }
                    break;              //每个case代码块最后要有break语句！！！
                case ']':
                    stk.pop();
                    if(stk.empty())
                    {
                        return false;
                    }
                    if(stk.top() != '[')
                    {
                        return false;
                    }
                    else
                    {
                        stk.pop();
                    }
                    break;
                case '}':
                    stk.pop();
                    if(stk.empty())
                    {
                        return false;
                    }
                    if(stk.top() != '{')
                    {
                        return false;
                    }
                    else
                    {
                        stk.pop();
                    }
                    break;
                default:
                    return false;
```

```
            }
        }
        if(stk.empty())
        {
            return true;
        }
        else
        {
            return false;
        }
    }
};
```

执行用时： **0 ms** ，在所有 C++ 提交中击败了 **100.00%** 的用户

内存消耗： **6.1 MB** ，在所有 C++ 提交中击败了 **83.91%** 的用户

通过测试用例： **93 / 93**

复杂度分析

- 时间复杂度：$O(n)$，其中 n 是字符串 s 的长度。
- 空间复杂度：$O(n+|\Sigma|)$，其中 Σ 表示字符集，本题中字符串只包含 6 种括号，$|\Sigma|=6$。栈中的字符数量为 $O(n)$，而哈希表使用的空间为 $O(|\Sigma|)$，相加即可得到总空间复杂度。

4、删除字符串中的所有相邻重复项（20230725，1047题，简单）

给出由小写字母组成的字符串 S，重复项删除操作会选择两个相邻且相同的字母，并删除它们。 在 S 上反复执行重复项删除操作，直到无法继续删除。 在完成所有重复项删除操作后返回最终的字符串。答案保证唯一。

示例：

- 输入："abbaca"
  输出："ca"
  解释： 例如，在 "abbaca" 中，我们可以删除 "bb" 由于两字母相邻且相同，这是此时唯一可以执行删除操作的重复项。之后我们得到字符串 "aaca"，其中又只有 "aa" 可以执行重复项删除操作，所以最后的字符串为 "ca"。

> 自己版本，没通过！！！

```
/*
思路：
入栈依次删除
一直重复，直到一次遍历中没有删除操作，用计数器标记。
先写一个处理一次遍历的函数，再多次调用。
入栈后判断相同，两个出栈；tmp字符重复赋值栈顶，再入栈并判断。
*/
class Solution {
```

```
public:
    stack<char> stk;
    string removeDuplicates(string s) {
        string tmpStr = s;
        char chPre = '\0';
        char chCur = '\0';
        int len = 0;

        for(int i = 0; i < len; i++)   //不能while(!s.empty())
        {
            chPre = stk.top();
            stk.push(tmpStr[i]);
            i++;
            chCur = stk.top();

            if(chCur == chPre && (stk.size() > 1))
            {
                stk.pop();
                stk.pop();
            }
        }
        string ret;
        int j = 0;
        while(!stk.empty())
        {
            ret[j] = stk.top();
            j++;                //别忘了累加j
            stk.pop();
        }
        reverse(ret.begin(),ret.end());
        return ret;
    }
};
```

根据答案修改后的自己版本

```
class Solution {
public:
    stack<char> stk;
    string removeDuplicates(string s) {
        string tmpStr = s;
        char chPre = '\0';
        char chCur = '\0';
        int len = 0;

        len = tmpStr.size(); /*len开始忘记了赋值*/
        for(int i = 0; i < len; i++)   //不能while(!s.empty())
        {
            if(!stk.empty())
            {
                chPre = stk.top();    //必须先判空，不然可能top是空的
            }
```

```
            if((tmpStr[i] == chPre) && (stk.size() > 0))
            {
                stk.pop();
            }
            else
            {
                stk.push(tmpStr[i]);
            }
        }
        string ret;
        while(!stk.empty())
        {
            ret += stk.top(); /*string初始化后只能通过拼接来改变字符串*/
            stk.pop();
        }
        reverse(ret.begin(),ret.end()); /*熟悉string类的接口*/
        return ret;
    }
};
```

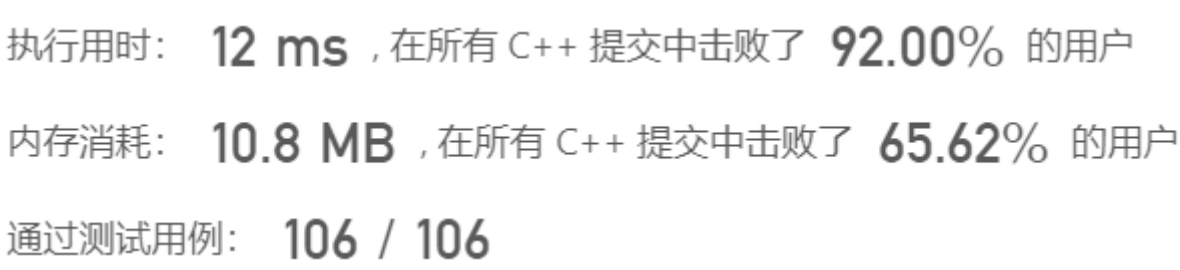
执行用时： 24 ms ，在所有 C++ 提交中击败了 45.62% 的用户

内存消耗： 11.8 MB ，在所有 C++ 提交中击败了 10.05% 的用户

通过测试用例： 106 / 106

> 默写答案版本

```
class Solution {
public:
    string removeDuplicates(string s) {
        string tmpStr;
        for(char ch : s)
        {
            if(!tmpStr.empty() && (ch == tmpStr.back()))  //tmpStr.back()取string
的栈顶元素!
            {
                tmpStr.pop_back();
            }
            else
            {
                tmpStr.push_back(ch);
            }
        }
        return tmpStr;
    }
};
```



5、逆波兰表达式求值（20230726，150题，中等）

给你一个字符串数组 tokens ， 表示一个根据 逆波兰表示法 表示的算术表达式。 请你计算该表达式。返回一个表示表达式值的整数。

注意：

- 有效的算符为 '+'、'-'、'*' 和 '/' 。
- 每个操作数（运算对象）都可以是一个整数或者另一个表达式。
- 两个整数之间的除法总是 向零截断 。
- 表达式中不含除零运算。
- 输入是一个根据逆波兰表示法表示的算术表达式。
- 答案及所有中间计算结果可以用 32 位 整数表示。

示例 1：

- 输入：tokens = ["2","1","+","3","*"]
  输出：9
  解释：该算式转化为常见的中缀算术表达式为：((2 + 1) * 3) = 9

示例 2：

- 输入：tokens = ["4","13","5","/","+"]
  输出：6
  解释：该算式转化为常见的中缀算术表达式为：(4 + (13 / 5)) = 6

> 自己写，漏洞百出版本

```
/*
思路：
依次入栈，直到遇到运算符，出栈两个字符串，并运算。
将结果入栈，继续等待下一个运算符
*/
class Solution {
public:
    int evalRPN(vector<string>& tokens) {
        //stack<string> tmp;
        vector<string> tmp;
        int firstNum = 0;
        int SecNum = 0;
        int Sum = 0;
        int len = tokens.size();
```

```
        for(int i = 0; i < len; i++)
        {
            tmp.push_back(tokens[i]);    /*这个元素已经入栈，栈中至少保证有三个字符串，
tmp.size() < 3才对*/
            if(tmp.size() < 3 )          //栈中至少有两个字符串，第三个是运算符，才能进
行运算
            {
                continue;
            }
            switch(tmp.back())   /*switc语句不能用于字符串，只能用整数、枚举或字符。但
是可以用str[0] == "+"来判断*/
            {
                case '+':                      //如果是字符串，则出栈两个
                firstNum = stoi(tmp.back());
                tmp.erase(tmp.end() - 1);   /*tmp.end() - 1才是指向最后一个元素，
tmp.end()指向的最后元素之后*/
                SecNum = stoi(tmp.back());
                tmp.erase(tmp.end() - 1);
                Sum = firstNum + SecNum;   /*sum不需要累加，运算完入栈就行*/
                tmp.push_back(atoi(Sum));  /*sum忘记入栈，这里就能看出栈应该用int类型
的*/
                break;

                case '-':
                firstNum = stoi(tmp.back());
                tmp.erase(tmp.end() - 1);
                SecNum = stoi(tmp.back());
                tmp.erase(tmp.end() - 1);
                Sum = firstNum - SecNum;
                tmp.push_back(atoi(Sum));
                break;

                case '*':
                firstNum = stoi(tmp.back());
                tmp.erase(tmp.end() - 1);
                SecNum = stoi(tmp.back());
                tmp.erase(tmp.end() - 1);
                Sum = firstNum * SecNum;
                tmp.push_back(atoi(Sum));
                break;

                case '/':
                firstNum = stoi(tmp.back());
                tmp.erase(tmp.end() - 1);
                SecNum = stoi(tmp.back());
                tmp.erase(tmp.end() - 1);
                Sum = firstNum / SecNum;
                tmp.push_back(atoi(Sum));
                break;

                default:
                continue;
            }
```

```
        }
        return tmp.front();
    }
};
```

根据ChatGPT4改写答案

```
/*
思路:
依次入栈，直到遇到运算符，出栈两个字符串，并运算。
将结果入栈，继续等待下一个运算符
临时的栈使用int类型的数据接收。
*/
class Solution {
public:
    int evalRPN(vector<string>& tokens) {
        vector<int> tmp;    /*注意定义的tmp数据类型，后续代码要匹配*/
        int firstNum = 0;
        int SecNum = 0;
        int Sum = 0;

        for(auto &str :  tokens) /*str使用引用，减少内存*/
        {
            if((str == "+") || (str == "-") || (str == "*") || (str == "/"))
            {
                SecNum = tmp.back();   /*第二个数字，即运算符右边的数字，是先出栈的!
*/
                tmp.pop_back();
                firstNum = tmp.back(); /*第一个数字，即运算符左边的数字，是后出栈的!
*/
                tmp.pop_back();
                if(str == "+")      Sum = firstNum + SecNum;
                else if(str == "-") Sum = firstNum - SecNum;
                else if(str == "*") Sum = firstNum * SecNum;
                else(str == "/")    Sum = firstNum / SecNum;
                tmp.push_back(Sum);
            }
            else
            {
                tmp.push_back(stoi(str));
            }
        }
        return tmp.front();
    }
};
```

执行用时： 8 ms ，在所有 C++ 提交中击败了 89.18% 的用户

内存消耗： 11.5 MB ，在所有 C++ 提交中击败了 97.85% 的用户

6、滑动窗口最大值（20230731，239题，困难）

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**

> 自己写，漏洞百出版本

```
/*
思路1:
依次获取nums中的数据，收集到K个后进行排序。
取最大值，并剔除第一个数据，随后取下一个数据，进行重复操作。
思路2:
依次取数值后，记录当前最大值a。
```

```
取到K个数值为止，当前最大值a1在索引i处，继续往后去若干，直至i+k处。
    1）此前若未遇到大于a1的数值，则输出K个a1；
    2）若遇到更大的数值，在i+m处遇到（m<k），则输出i+m-k或者m的较小值（因为a1可能在索引小于K处），一直检索至最后数据
    情况1中，需要取a1后的第K个数字，继续找其后最大的K个数字内的最大值。
    情况2中，依次处理就行，注意索引值的边界。
使用栈stk存储遍历的nums，临时变量count记录最大值应该出现的次数。
    从第一个数开始，就可以以这个数为基准去找。只是注意初始的前K个的特殊点，要满足框柱K个开始计数。
    当索引大于K时，只要找到大于当前最大数时即可停止，计算相隔多少个数字，然后将其入栈stk2。
思路3：
    细分情况较多，a在K内或K外；找到或没找到b；找到的b在K内或K外；没找到b。
    分析后可结合为找到或者没找到b两种情况。用较小的索引来控制a入栈次数。
给ChatGPT的：我的思路是以一个数字a为基准，往后找更大的数b（只会往后找k个数），找到b，就将一定数量的a存入容器vec2中；没找到b，就将a放入容器vec2 k次，然后将后面那个数作为新的基准。
*/

class Solution {
public:
    vector<int> maxSlidingWindow(vector<int>& nums, int k) {
        //vector<int> vec1;
        vector<int> vec2;   //用于存输出数据
        int indexMaxNum = 0;
        int MaxNew = 0;
        int isFirst = 1;
        int len = nums.size();
        int j = 0;
        int tmp = 0;

        for(int i = 0; i < len; i++)
        {
            j++;                //判断j等于6，即在没找到b时情况时，做相应处理
            tmp = nums[i];
            if(isFirst)         //以第一个数作为基点
            {
                MaxNew = tmp;
                indexMaxNum = i;
                isFirst = 0;
            }
            if(1 == len)  //只有一个数字的情况
            {
                vec2.push_back(nums[indexMaxNum]);
                return vec2;
            }
            if(tmp >= MaxNew )    //找到一个大于当前的数字.需要考虑最后的数字最大且需要输出的情况
            {

                int count1 = (i - k) > 0 ? (i - k): 0;      //负数-4也算true。。
                int count2 = (i - indexMaxNum) > 0 ? (i - indexMaxNum): 0;
                while(count1 && count2)             //包括了b在K内或者外的情况，有一种为零，则之前的较大数不能入栈vec2
```

```
                {
                    vec2.push_back(nums[indexMaxNum]);
                    count1--;
                    count2--;
                }
                MaxNew = tmp;      //要先把之前的大数存近栈，然后再重新赋值
                indexMaxNum = i;  //要先把之前的大数存近栈，然后再重新赋值
                j = 0;       //找到新的大数，则重新计数
            }
            //|| (i == len - 1)
            if(j == k - 1 )   //往后找k个数都没找到，则入栈K次，将当前数作为新基点。考虑到最后但没慢K个数，切没找到b
            {
                int count3 = (i - k + 2) > 0 ? (i - k + 2): 0;    //得加上1，新大数本身位置也要算一次
                int count4 = (i - indexMaxNum + 2) > 0 ? (i - indexMaxNum + 2): 0;
                while(count3 && count4)
                {
                    vec2.push_back(nums[indexMaxNum]);
                    count3--;
                    count4--;
                }

                isFirst = 1;  //让下次入栈的数据重新当做新的基点
                j = 0;       //找到后面第k个数还没找到新的大数，则重新计数
            }
           if(i == len - 1)   //往后找k个数都没找到，则入栈K次，将当前数作为新基点。考虑到最后但没慢K个数，切没找到b
            {
                //给最后的数字打上补丁
                vec2.push_back(nums[i]);
            }
        }
        return vec2;
    }
};
```

根据答案修改的版本，使用双端队列deque

```
/*
思路：
使用双端队列deque，存放nums[]的索引。
想要每次滑动窗口，则确定一个数字。
首先for循环确认前K个数中数据，定制规则如下：
    依次比对nums[i]，入队列的数只能比
    之后从K开始，遍历nums[i]，nums[i]大于队尾索引对应的值时，令队列依次后端出列。
    这样保持了队列始终是右大到小的，队头索引x最大，其后的预备队员也不会被舍弃，将在x小于i-k后，依次充当大王。
之后从K开始循环遍历nums[i]，保持队列的由大到小的性质，依次将队头的数存储在vec中，最后返回vec。
理解：
deque要存放nums[i]的索引，因为索引不仅能够唯一确定它的值，还能由索引确认数值的位置。
```

```
首次for循环K个后，双端队列已经是个窗口了，只是我们可以给这个窗口中的元素定制规则，方便找到所需数据。
*/
class Solution {
public:
    vector<int> maxSlidingWindow(vector<int>& nums, int k) {
        deque<int> dq;
        vector<int> ret;
        int len = nums.size();
        for(int i = 0; i < k; i++)
        {
            /*if(!deque.empty() && nums[i] >= nums[deque.back()]),直接while就行，不用这句if，画蛇添足了！ */
            while(!dq.empty() && nums[i] >= nums[dq.back()]) /*deque是数据类型，用的时候用它的对象dq*/
            {
                dq.pop_back();
            }
            dq.push_back(i);  /*别忘记结束的分号";"*/
        }
        ret.push_back(nums[dq.front()]);
        for(int i = k; i < len; i++)
        {
            while(!dq.empty() && nums[i] >= nums[dq.back()])
            {
                dq.pop_back();
            }
            /*新的大数来了后，deque所有索引都会被清除，所以得先push，不然后面while会访问front，导致访问空，而出错*/
            dq.push_back(i);  /*别忘记结束的分号";"*/
            /*if(dq.front() <= (i - k)),使用if也可以？！！ */
            while(dq.front() <= (i - k)) /*不在窗口的最大值要从队列头出列*/
            {
                dq.pop_front();
            }

            ret.push_back(nums[dq.front()]);
        }
        return ret;
    }
};
```

执行用时： **240 ms** ，在所有 C++ 提交中击败了 **45.85%** 的用户

内存消耗： **131.5 MB** ，在所有 C++ 提交中击败了 **57.12%** 的用户

复杂度分析

- 时间复杂度：$O(n)$，其中 n 是数组 $nums$ 的长度。每一个下标恰好被放入队列一次，并且最多被弹出队列一次，因此时间复杂度为 $O(n)$。
- 空间复杂度：$O(k)$。与方法一不同的是，在方法二中我们使用的数据结构是双向的，因此「不断从队首弹出元素」保证了队列中最多不会有超过 $k+1$ 个元素，因此队列使用的空间为 $O(k)$。

> 根据答案修改的版本，使用优先队列priority_queue<pair<int,int>>

```
/*
使用优先队列解题：
优先队列存储窗口中的数字，最大的数字在队头。
遍历nums时，将nums[i]依次放入队列，读取队列头数字即可。
但队列头可能已经不在时间窗口，于是可以令队列存储一个元组pair(value,index)。
出列时while判断是否将次数值读取。
*/
class Solution {
public:
    vector<int> maxSlidingWindow(vector<int>& nums, int k) {
        priority_queue<pair<int, int>> prDq;   /*不是deque，是queue队列，
priority_deque<pair<int, int>>*/
        vector<int> ret;
        int len = nums.size();
        for(int i = 0; i < k; i++)
        {
            prDq.emplace(nums[i],i);   /*不是prDq.push_back(nums[i],i)*/
        }
        ret.push_back(prDq.top().first); /*不是ret.push_back(prDq.front().first)*/
        for(int i = k; i < len; i++)
        {
            prDq.emplace(nums[i],i);
            while(prDq.top().second <= (i - k))
            {
                prDq.pop();  /*不是prDq.pop_front()*/
            }
            ret.push_back(prDq.top().first);
        }
        return ret;
    }
};
```

执行用时： **276 ms** ，在所有 C++ 提交中击败了 **22.54%** 的用户

内存消耗： **145.3 MB** ，在所有 C++ 提交中击败了 **13.49%** 的用户

复杂度分析

- 时间复杂度：$O(n \log n)$，其中 n 是数组 $nums$ 的长度。在最坏情况下，数组 $nums$ 中的元素单调递增，那么最终优先队列中包含了所有元素，没有元素被移除。由于将一个元素放入优先队列的时间复杂度为 $O(\log n)$，因此总时间复杂度为 $O(n \log n)$。
- 空间复杂度：$O(n)$，即为优先队列需要使用的空间。**这里所有的空间复杂度分析都不考虑返回的答案需要的 $O(n)$ 空间，只计算额外的空间使用。**

> chatgpt版本

```
/*
解法思路:
使用一个双端队列，队列中存储的是元素的索引，队头元素是当前窗口的最大值。
每次向右移动窗口时，如果新的元素大于队尾元素，那么就将队尾元素弹出，直到队列为空或者新的元素小于等于队尾元素，然后将新元素的索引压入队尾。
如果队头元素已经滑出窗口，那么就将队头元素弹出。
如果窗口已经形成，那么就将队头元素（窗口最大值）放入结果数组。
这个解法的时间复杂度是O(n)，空间复杂度是O(k)，其中n是数组的长度，k是滑动窗口的大小。
*/
class Solution {
public:
    vector<int> maxSlidingWindow(vector<int>& nums, int k) {
        deque<int> deq;
        vector<int> res;
        for (int i = 0; i < nums.size(); i++) {
            // 如果队列不为空，且当前数字大于队尾，那么就将队尾弹出，直到队列为空或者当前数字小于等于队尾
            while (!deq.empty() && nums[i] > nums[deq.back()]) {
                deq.pop_back();
            }
            // 将当前数字压入队尾
            deq.push_back(i);
            // 如果队头元素已经滑出窗口，那么就将队头弹出
            if (i - deq.front() + 1 > k) {
                deq.pop_front();
            }
            // 如果窗口已经形成，那么就将队头（窗口最大值）放入结果数组
            if (i + 1 >= k) {
                res.push_back(nums[deq.front()]);
            }
        }
        return res;
    }
};
```

6、前K个高频元素（20230802，347题，中等）

给你一个整数数组 nums 和一个整数 k ，请你返回其中出现频率前 k 高的元素。你可以按 任意顺序 返回答案。

示例 1:

- 输入: nums = [1,1,1,2,2,3], k = 2
- 输出: [1,2]

进阶：你所设计算法的时间复杂度必须优于O(nlogn),其中n 数组大小。

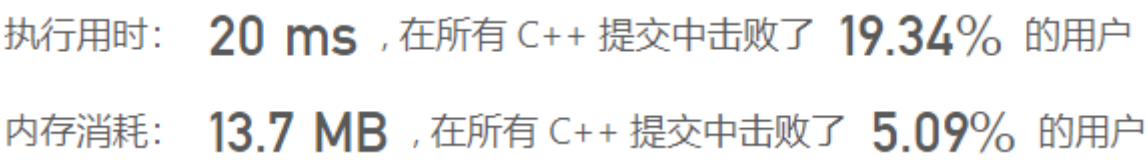
> 自己版本，与ChatGPT交流修改版本

```
/*
思路:
先循环遍历nums，存入优先队列priority_queue<int>;
再依次队列出列，相同的数存入vector<pair<int,int>>，（计数，值）
第三次将vector放入优先队列priority_queue<int,int>，依次取出前k个值存入vector，最后返回。
*/
class Solution {
public:
    vector<int> topKFrequent(vector<int>& nums, int k) {
        int len = nums.size();
        priority_queue<int> pq;
        vector<pair<int, int>> vec;  //1
        vector<int> ret;
        priority_queue<pair<int,int>> pq2;

        for(int i = 0; i < len; i++)
        {
            pq.emplace(nums[i]);
        }
        for(int i = 0; i < len; i++)
        {
            if(!vec.empty() && pq.top() == vec.back().second) //2,3
            {
                vec.back().first = vec.back().first + 1;  //4
                pq.pop();   //5
            }
            else
            {
                vec.push_back(make_pair(1, pq.top()));  //6,7
                pq.pop();
            }
        }
        int lenVec = vec.size();
        for(int i = 0; i < lenVec; i++)
        {
            pq2.emplace(vec[i].first, vec[i].second);
        }
        for(int i = 0; i < k; i++)
```

```
            {
                ret.push_back(pq2.top().second);
                pq2.pop();
            }
            return ret;
        }
    };
```

> 本题出现问题：
> 1、vector<pair<int, int>> vec(10)，可能会溢出，如果nums个数大于10
> 2、和容器vector的最后一个数比较，直接vec.back()，别用vec[j]
> 3、pq.top要加括号pq.top()。相同的数直接自增计数
> 4、vector[j].first要改为vec[j].first，要用数据类型的对象
> 5、别忘了要依次弹出队列头部的数
> 6、vec.push_back(1, pq.top())，vector放入元组pair要用make_pair
> 7、vec[j].first,不能直接访问vec不存在的索引，使用push_back加入新元素



四、数组（5道题）

五、字符串（7道题）

六、双指针法（10道题）

七、二叉树（32道题）

八、动态规划（52道题）

九、贪心算法（22道题）

十、回溯算法（20道题）

十一、单调栈（5道题）