- 数组

# 数组

## 704. 二分查找 - 力扣（LeetCode）

```
class Solution
{
public:
    int search(vector<int> &nums,int target)
    {
        int left=0;
        int right=nums.size()-1;
        int mid=(left+right)/2;

        while(left<=right)
        {
            if(target==nums[mid])
            {
                return mid;
            }else if(target>nums[mid])
            {
                left=mid+1;
            }else
            {
                right=mid-1;
            }
            mid=(left+right)/2;
        }
        return -1;
    }
};
```

## 35. 搜索插入位置 - 力扣（LeetCode）

利用暴力方法求解：

```
#include <vector>

class Solution
{
public:
    int searchInsert(std::vector<int> &nums,int target)
    {
        //利用暴力解法进行求解 （其实本质上与插入排序中寻找插入位置有一定的相似之处）
        for(int i=0;i<nums.size();i++)
        {
            //不断进行右移，然后找到最佳位置
            if(nums[i]>=target)
```

```
            {
                return i;
            }
        }
        return nums.size();
    }
};
```

利用二分法求解：

```
class Solution
{
public:
    int searchInsert(std::vector<int> &nums,int target)
    {
        //利用二分法进行求解
        //当前需要查询的数字在 [left,right]这个区间内部

        int n=nums.size();
        int left=0;
        int right=n-1;
        while(left<=right)
        {
            int middle=left+((right-left)>>1);

            if(nums[middle]>target)
            {
                right=middle-1;
            }else if(nums[middle]<target)
            {
                left=middle+1;
            }else
            {
                return middle;
            }
        }
        return right+1;
    }
};
```

## 34. 在排序数组中查找元素的第一个和最后一个位置 - 力扣（LeetCode）

第一种方法当然是遍历了，暴力算法永远都不会过时

```
class Solution {
public:
    vector<int> searchRange(vector<int>& nums, int target) {
        int left=-1;
        int right=-1;
        int state=0;
        for (size_t i=0;i<nums.size();i++)
        {
            if(nums[i]==target && !state)
            {
                left=i;
```

```
                right=i;
                state=1;
            }
            if(nums[i]==target && state)
            {
                right=i;
            }
        }
        vector<int> RetValue;
        RetValue.push_back(left);
        RetValue.push_back(right);
        return RetValue;
    }
};
```

第二种打算使用二分查找来进行。但是发现效率并不高，思路主要是找到值之后利用两个指针，不断向左遍历，然后再不断向右遍历，找到自己想要的位置，但是并没有达到我想要的效果，所以这个方法不建议使用。

```
class Solution {
public:
    vector<int> searchRange(vector<int>& nums, int target) {
        int left=-1;
        int right=-1;
        int state=0;
        int L=0;
        int R=nums.size()-1;
        while(L<=R)
        {
            int mid=L+((R-L)>>1);//防止溢出
            if(nums[mid]>target)
            {
                L=mid+1;
            }else if(nums[mid]<target)
            {
                R=mid-1;
            }else
            {
                left=mid;
                right=mid;
                break;
            }
        }
        while(left!=-1 && left >0)
        {
            if(nums[left-1]==target)
            {
                left--;
            }else
            {
                break;
            }
        }
        while(right!=-1 && right<nums.size())
        {
            if(nums[right+1]==target)
            {
                right++;
            }else
            {
                break;
```

```
            }
        }
        vector<int> RetValue;
        RetValue.push_back(left);
        RetValue.push_back(right);
        return RetValue;
    }
};
```

第三种是在代码随想录中的代码，这里自己使用复现一下。

其实这里的思路，可以说是一种“超量”的思想，就是不断查找然后最后再进行补足就可以了

```
class Solution {
public:
    vector<int> searchRange(vector<int>& nums, int target)
    {
        int leftborder=getLeftBorder(nums,target);
        int rightborder=getRightBorder(nums,target);
        if(leftborder ==-2 || rightborder==-2) return {-1,-1};
        if(rightborder-leftborder>1)
        {
            return {leftborder+1,rightborder-1};
        }
        return {-1,-1};
    }

    int getLeftBorder(vector<int>& nums,int target)
    {
        int leftborder=-2;//这里记录左侧没有被赋值的情况
        int left=0;
        int right=nums.size()-1;
        while(left<=right)
        {
            int mid=left+((right-left)>>1);

            if(nums[mid]<target)
            {
                left=mid+1;
            }else
            {
                right=mid-1;
                leftborder=right;
            }
        }
        return leftborder;
    }

    int getRightBorder(vector<int> &nums,int target)
    {
        int rightborder=-2;//这里记录没有被赋值的情况
        int left=0;
        int right=nums.size()-1;
        while(left<=right)
        {
            int mid=left+((right-left)>>1);

            if(nums[mid]>target)
            {
                right=mid-1;
            }else
            {
```

```
                left=mid+1;
                rightborder=left;
            }
        }
        return rightborder;
    }
};
```

### 69. x 的平方根 - 力扣（LeetCode）

有牛顿迭代法

### 牛顿迭代法 - x 的平方根 - 力扣（LeetCode）

二分法

### 二分查找（Java） - x 的平方根 - 力扣（LeetCode）

这个看起来还是蛮简单的，直接进行一次尝试
这个看起来利用暴力方法可以进行求解，那么我们就使用暴力法就可以了

```
func mySqrt(x int) int {
    tmp:=1
    for tmp>=0 && tmp*tmp<=x{
        tmp++
    }
    return tmp-1
}
```

当然我们也可以使用二分查找的方法来解决这个问题，还是很简单的。

这里小用一下go，还是看着非常亲切，总归是学习了半个多月的东西

### 27. 移除元素 - 力扣（LeetCode）

```
#include <iostream>
#include <vector>
using namespace std;
//这样也蛮不错的，这个方法是一种暴力的方法，但是效率较低
class Solution {
public:
    int removeElement(vector<int>& nums, int val) {
        int Len=nums.size();

        for(int i=0;i<Len;i++)
        {
            if(nums[i]==val)
            {
                for(int j=i+1;j<Len;j++)
                {
```

```
                    nums[j-1]=nums[j];
                }
                Len--;
                i--;
            }
        }
        return Len;
    }
};
```

利用暴力方法当然还是很好解决的，现在的问题是如何用更简单的方法解决这些问题：
利用双指针的方法，可以解决这个问题，我当然感觉这个部分实在是太巧妙了，当然肯定是我想不到，其实就是选择一个快速指针，选择一个慢速指针，然后不断指向，如果不是需要寻找的值，则慢速指针右移，如果是我们需要寻找的值，
慢速指针则不动，同时快速指针不断向右，覆盖掉慢速指针的值，也就是说，慢速指针指向的永远是我们需要覆盖的值，
可以很好的删除掉连续的同样的val，这样可以很好的解决。

```
class Solution {
public:
    int removeElement(vector<int>& nums, int val) {
        int slowPtr=0;
        int quickPtr=0;

        for(;quickPtr<nums.size();quickPtr++)
        {
            if(nums[quickPtr]!=val)
            {
                nums[slowPtr++]=nums[quickPtr];
            }
        }
        return slowPtr;
    }
};
```

## 844. 比较含退格的字符串 - 力扣（LeetCode）

本人比较笨，开始还是采用暴力的方法求解，当然这里还是需要考虑一些东西，例如我设置的缓冲区现在空了，就不能再弹出了，直接上代码了。

```
class Solution {
public:
    bool backspaceCompare(string s, string t) {
        int lens=s.size();
        int lent=t.size();

        int indexs=0;
        int indext=0;

        string strs="";
        string strt="";
        for(int i=0;i<lens;i++)
        {
            if(s[indexs]!='#')
            {
```

```
                strs+=s[indexs];
            }else
            {
                if(strs.size()>0) strs.pop_back();
            }
            indexs++;
        }
        for(int i=0;i<lent;i++)
        {
            if(t[indext]!='#')
            {
                strt+=t[indext];
            }else
            {
                if(strt.size()>0) strt.pop_back();
            }
            indext++;
        }
        return strt==strs;
    }
};
```

最好的方法还是利用双指针来进行，这个思路的问题在于没有想到如何让两个指针同时移动，这里采用别人的思路重新写一遍，如下：

```
class Solution {
public:
    bool backspaceCompare(string s, string t) {
        int ptrS=s.size()-1;
        int ptrT=t.size()-1;
        int skipS=0,skipT=0;
        while (ptrS>=0 || ptrT>=0)
        {
            while(ptrS>=0)
            {
                if(s[ptrS]=='#')
                {
                    skipS++;
                    ptrS--;
                }else if(skipS>0)
                {
                    skipS--;
                    ptrS--;
                }
                else
                {
                    break;//此时直接结束循环，然后去判断是否两项是相同的
                }
            }
            while(ptrT>=0)
            {
                if(t[ptrT]=='#')
                {
                    skipT++;
                    ptrT--;
                }
                else if(skipT>0)
                {
                    skipT--;
                    ptrT--;
```

```
                }else
                {
                    break;
                }
            }
            if(ptrS>=0 && ptrT>=0)
            {
                if(s[ptrS]!=t[ptrT])
                {
                    return false;
                }
            }else
            {
                if(ptrS>=0 || ptrT>=0) //也就是此时有一个没有运行到结尾
                {
                    return false;
                }
            }
            ptrS--;
            ptrT--;
        }
        return true;
    }
};
```

## 977. 有序数组的平方 - 力扣（LeetCode）

利用双指针的方法解题，极大的提高了算法的速度

```
class Solution {
public:
    vector<int> sortedSquares(vector<int>& nums) {
        vector<int> tmp(nums.size(),0);
        int k=nums.size()-1;
        for(int i=0,j=nums.size()-1;i<=j;)
        {
            if(nums[i]*nums[i]<nums[j]*nums[j])
            {
                tmp[k--]=nums[j]*nums[j];
                j--;
            }else
            {
                tmp[k--]=nums[i]*nums[i];
                i++;
            }
        }
        return tmp;
    }
};
```

## 209. 长度最小的子数组 - 力扣（LeetCode）

这里首先还是采用了暴力方法来解决这些个问题，当然这里出现了个问题，就是执行时间超限了，那这里还是采用稍微高级一点的办法来解决了，叫做滑动窗口的方法。

```
class Solution {
public:
    int minSubArrayLen(int target, vector<int>& nums) {
        //这里先使用暴力方法求一下解
        int result=INT32_MAX;//来记录最终的结果
        int sum=0;
        int subLength=0;
        for(int i=0;i<nums.size();i++)
        {
            sum=0;
            for(int j=i;j<nums.size();j++)
            {
                sum+=nums[j];
                if(sum>=target)
                {
                    subLength=j-i+1;
                    result=result<subLength ? result:subLength;//这里利用一个三元运算符来进行
                    break;//此时查询到了 那么就结束这一轮的循环
                }
            }
        }
        return result;
    }
};
```

这个方法在我去年刷题的过程中，一直没有理解透彻，看看今年我有没有长进力。
这里对于求子序列的问题，大都用的是滑动窗口的方式，这一点还是一直没有改变的。
https://programmercarl.com/0209.%E9%95%BF%E5%BA%A6%E6%9C%80%E5%B0%8F%E7%9A%84%E5%AD%90%E6%95%B0%E7%BB%84.html#%E6%BB%91%E5%8A%A8%E7%AA%97%E5%8F%A3
上面是代码随想录的链接了。
这里的思路其实很明确，但是要是去年的我确实理解不了，
首先，窗口里面的内容是什么，窗口里面的内容是和大于target的子序列；
其次，我们什么时候移动窗口，也就是当窗口满足条件的时候，我们此时需要缩小窗口了；
再次，我们该怎样移动窗口，当然也就是我们移动的是哪一个指针，我们遍历窗口的末尾指针，让值不断扩大，内层循环也就是当我们的值到达大于等于target这个条件的情况时，我们需要判断当前的窗口大小是不是小于我们刚才求得的大小，然后移动前一个指针，完成我们想要的操作，同时sum要减去当前开始窗口的值，那么我们就可以开始完成我们的代码了。

```
class Solution {
public:
    int minSubArrayLen(int target, vector<int>& nums) {
        //利用滑动窗口求解
        int result=INT32_MAX;//最大值
        int sum=0;
        int left=0;//滑动窗口的初始位置

        for(int right=0;right<nums.size();right++)
        {
            sum+=nums[right];//此时向后遍历，加上我们当前的最右端窗口的值
            while (sum>=target)
            {
                int tmp=right-left+1;
                result=result<tmp ? result : tmp;
                sum-=nums[left++];//减去左窗口的值
            }
        }
```

```
        return result==INT32_MAX ? 0:result;
    }
};
```

和上面类似的题目，我们还需要多加练习，下面还是题解

## 904. 水果成篮 - 力扣（LeetCode）

利用滑动窗口来解决问题，
还是一样的思路
我们需要知道，窗口里面的是什么
窗口里面装的是最多两种水果
那每次移动的条件是什么，也就是窗口里面水果的种类数大于两种，此时向右移动，正确的，滑动窗口
yes

```
class Solution {
public:
    int totalFruit(vector<int>& fruits) {
        //这里打算采用map结构进行哈希计算
        map<int, int> cnt;
        int left=0,ans=0;

        for(int right=0;right<fruits.size();right++)
        {
            ++cnt[fruits[right]];
            while(cnt.size()>2) //当前哈希表中有项
            {
                auto it=cnt.find(fruits[left]);
                it->second--;//这里其实是值
                if(it->second==0)
                {
                    cnt.erase(it);//删除当前这个迭代器指向的一项
                }
                left++; //向右移动滑动窗口
            }
            ans=ans>(right-left+1) ? ans:(right-left+1);
        }
        return ans;
    }
};
```

## 76. 最小覆盖子串 - 力扣（LeetCode）

还是利用滑动窗口来解决，
窗口里面装的东西：包含t中字符的最小子串
移动条件：当包含了t中的所有字符的时候
那么我们思考，如何判断我们已经具有了所有的字符
这里留到明天再解决