LeetCode解题报告1（1-10）

刘国强*

哈尔滨工业大学

2016年7月

Contents

*email: 15114576057@163.com

1 Two Sum

1.1 Question

Given an array of integers, return `indices` of the two numbers such that they add up to a specific target.You may assume that each input would have ***exactly*** one solution.

例子：

```
Given nums = [2, 7, 11, 15], target = 9,
Because nums[0] + nums[1] = 2 + 7 = 9,
return [0, 1].
```

标签：Array;Hash Table 难度：Easy

1.2 Describtion

题目的含义是，给定一个整数数组，还有一个数作为目标（target），在数组中找到两个数，使得这两个数的和正好等于目标，并返回这两个数在数组中的下标。在给定的例子中，目标是9，数组中的2和7的和正好是9，那么就返回2和7的下标，即0,1。这里假设每个数组中只有一对数的和与目标相等，也就是说，找到一对数返回下标就好。

1.3 Solution #1

暴力搜索。这是最简单的一个方法。遍历数组中的每一个元素，当访问都一个元素x时，从这个元素x的下一个元素再次遍历，检查数组中是否含有元素$target-x$，如果有，就返回这两个元素的下标；如果遍历完整个数组后还没有，就返回空。C语言的实现代码如下：

```
/**
 * Note: The returned array must be malloced, assume caller calls free().
 */
int* twoSum(int* nums, int numsSize, int target) {
    int* indices=malloc(2*sizeof(int));
    for(int i=0;i<numsSize;i++){
        for(int j=i+1;j<numsSize;j++){
            if(nums[j]==target-nums[i]){
                indices[0]=i;
                indices[1]=j;
                return indices;
            }
        }
```

```
    }
    return NULL;
}
```

复杂度分析

- 时间复杂度:$O(n^2)$。对于每个元素，我们都对数组中剩余的元素进行遍历来找合适的元素，这样就花费了$O(n)$的时间。因此，整个过程的时间复杂度是$O(n^2)$。
- 空间复杂度:$O(1)$。这里我们仅仅声明了一个int类型的数组，所以空间复杂度是$O(1)$。

1.4 Solution #2

现在考虑如何降低时间复杂度，毕竟$O(n^2)$的时间复杂度对于这个问题来说还是有点大。题目要求找到数组中的两个数，使得这两个数的和等于给定的*target*。那么也就是说，对于当前访问的元素x，如果x是结果的一部分，那么*target*-x一定也在这个数组中。这样，我们可以从头遍历这个数组，对于每一个元素x，查找*target*-x是否也在这个数组中。这就把问题归结为，如何快速地判断一个元素是否在数组中。这就用到了*Hash Table*。一个*Hash Table*查找元素的时间近似为常量（因为有时候会发生碰撞），所以在*Hash Table*中查找一个元素的时间大致为$O(1)$。不过，由于创建了一个*Hash Table*，所以空间复杂度变大了，是$O(n)$，但这样做降低了整个过程的时间复杂度，是一个以空间换时间的策略。在这个版本的方案中，一共进行了两次数组的遍历。第一次是创建一个*Hash Table*，第二次遍历检查是否有答案存在。Java的实现代码如下:

```
public class Solution {
    public int[] twoSum(int[] nums, int target) {
        Map<Integer,Integer> map=new HashMap<>();
        for(int i=0;i<nums.length;i++){
            map.put(nums[i],i);
        }

        for(int i=0;i<nums.length;i++){
            int left=target-nums[i];
            if(map.containsKey(left)&&map.get(left)!=i)
                return new int[]{i,map.get(left)};
        }
        throw new IllegalArgumentException("No Solution");
    }
}
```

复杂度分析

- 时间复杂度:$O(n)$。虽然遍历数组两次，但是由于*Hash Table*将元素的查找时间降到了$O(1)$，所以整个过程的时间复杂度是$O(n^2)$。
- 空间复杂度:$O(n)$。由于需要将数组中的n个元素存储在*Hash Table*中，所以需要额外的$O(n)$的空间。

1.5 Solution #3

在方法2中，我们对数组进行了两次遍历，这里考虑能否仅仅进行一次遍历。在上面中，第一次遍历将数组中的元素添加到*Hash Table*中，第二次遍历是查找元素。我们可以将两次遍历合并在一起，一边讲元素添加到*Hash Table*中，一边进行查找。Java实现代码如下:

```
public class Solution {
    public int[] twoSum(int[] nums, int target) {
        Map<Integer,Integer> map=new HashMap<>();
        for(int i=0;i<nums.length;i++){
            int left=target-nums[i];
            if(map.containsKey(left)){
                return new int[]{map.get(left),i};
            }
            map.put(nums[i],i);
        }
        throw new IllegalArgumentException("No Solution");
    }
}
```

复杂度分析

- 时间复杂度:$O(n)$。和方法2一样。
- 空间复杂度:$O(n)$。和方法2一样。

1.6 Solution #4

方法2和方法3都使用了*Hash Table*，这可以降低查找元素的时间。现在考虑如果不用*Hash Table*呢? 这就需要数组具有一定的特点了，也就是要求数组是有序的。当数组是有序的时候，我们就可以使用两个游标在数组的左右两端分别进行遍历，这样，也仅仅需要一次遍历就可以。

在C/C++中的algorithm头文件中，有一个sort函数，能够根据提供的排序方法对数组进行排序。函数原型如下：

```
template< class RandomAccessIterator > void sort( RandomAccessIterator
    first , RandomAccessIterator last );
template< class RandomAccessIterator, class Compare > void sort(
    RandomAccessIterator first , RandomAccessIterator last , Compare comp );
```

这两个函数都有*first*和*last*两个参数，表示了需要排序的数组的左右界限。第二个函数还有一个*comp*参数，是用户自定义的比较函数。
由于题目要求返回的是元素的下标，所以需要对每个元素保存下标的信息，这样可以自定义一个结构体，代码如下：

```
struct Node
{
    int val;
    int idx;
};
```

其中*val*存储元素的值，*idx*存储元素的下标。接下来就需要一个自定义的比较函数，代码如下：

```
bool cmp(Node a,Node b)
{
    return a.val<b.val;
}
```

我们仅仅需要比较元素的值。最终的C++实现代码如下：

```
struct Node
{
    int val;
    int idx;
};
bool cmp(Node a,Node b)
{
    return a.val<b.val;
}
class Solution {
public:
    vector<int> twoSum(vector<int>& nums, int target) {
        vector<int> ret(2);
        vector<Node> nodes;
        for(int i=0;i<nums.size();i++)
        {
```

```
            Node node;
            node.val=nums[i];
            node.idx=i;
            nodes.push_back(node);
        }
        sort(nodes.begin(),nodes.end(),cmp);
        for(int i=0,j=nodes.size()-1;i!=j;)
        {
            int sum=nodes[i].val+nodes[j].val;
            if(sum==target)
            {
                ret[0]=nodes[i].idx<nodes[j].idx?nodes[i].idx:nodes[j].idx;
                ret[1]=nodes[i].idx+nodes[j].idx-ret[0];
                break;
            }
            else if(sum<target)
                i++;
            else
                j--;
        }
        return ret;
    }
};
```

2 Add Two Numbers

2.1 Question

You are given two linked lists representing two non-negative numbers. The digits are stored in reverse order and each of their nodes contain a single digit. Add the two numbers and return it as a linked list.

Input: (2 -> 4 -> 3) + (5 -> 6 -> 4)

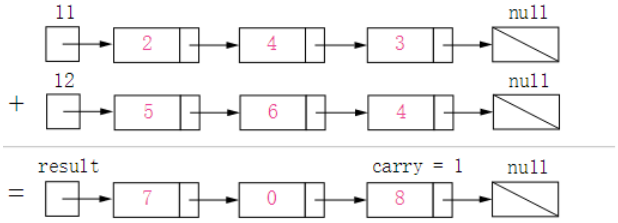
Output: 7 -> 0 -> 8

标签：Linked List;Math 难度：Medium

2.2 Describtion

这是一个链表题，链表中的每一个节点存储一个一位数字，整个链表表示一个十进制的数。给出两个链表代表两个数，然后模拟数的加法把这两个链表加起来得到一个新的链表。和平时的加法不同的是，这里链表中的数字是倒序存储的，也就是如果产生进

位，这个进位加在下一个节点上。下图是上面那个例子的示意图：



2.3 Solution

简单的思路就是从头开始遍历两个链表，将对应的节点上的数字相加作为当前节点的值，如果产生进位就在加在下一个节点上。需要考虑的问题有：

- 如果两个链表的长度不同怎么办；
- 如果两个链表的最后一个节点相加后又产生了进位怎么办；

显然在进行计算的时候需要一个循环，循环结束的条件是到达了两个链表的结尾。也就是说，如果一个链表还没结束而另一个链表结束了，那么循环也应该继续。即循环条件为while(p!=null || q!=null)。在每次循环过程中，首先要判断当前节点是否为*null*，如果是*null*，则对应的数字应该是0。下表是一些需要注意的特殊情况：

例子	解释
l_1=[0,1],l_2=[0,1,2]	一个链表比另一个长。
l_1=[],l_2=[0,1]	其中一个链表为空。
l_1=[9,9],l_2=[1]	相加的结果会有额外的进位，使得最终的链表需要在结尾增加一个节点。

其中最后一种情况最容易忽略。Java实现代码如下：

```
/**
 * Definition for singly-linked list.
 * public class ListNode {
 *     int val;
 *     ListNode next;
 *     ListNode(int x) { val = x; }
 * }
 */
public class Solution {
    public ListNode addTwoNumbers(ListNode l1, ListNode l2) {
        ListNode dummyHead=new ListNode(0);
```

```
        ListNode p=l1,q=l2,curr=dummyHead;
        int carry=0;
        while(p!=null || q!=null){
            int x=p==null?0:p.val;
            int y=q==null?0:q.val;
            int sum=x+y+carry;
            carry=sum/10;
            curr.next=new ListNode(sum%10);
            curr=curr.next;
            if(p!=null)p=p.next;
            if(q!=null)q=q.next;
        }
        if(carry>0){
            curr.next=new ListNode(carry);
        }
        return dummyHead.next;
    }
}
```

上面的代码就是按照基本思路运行的。这里有个关于*dummyHead*的小技巧。如果需要返回的结果是一个链表，那么这个链表的头结点就需要在计算之前创建，那么这个节点的值就不确定。这时，如果使用一个额外的节点作为头结点就避免了这个问题。当计算结束后，就可以直接返回这个*dummyHead*节点的*next*节点。
复杂度分析

- 时间复杂度:$O(max(m,n))$。只需要一次遍历即可。
- 空间复杂度:$O(max(m,n))$。相加后得到的链表的最大长度就是$max(m,n)+1$。

3 Longest Substring Without Repeating Characters

3.1 Question

Given a string, find the length of the longest substring without repeating characters. Examples: Given "abcabcbb", the answer is "abc", which the length is 3. Given "bbbbb", the answer is "b", with the length of 1. Given "pwwkew", the answer is "wke", with the length of 3. Note that the answer must be a substring, "pwke" is a subsequence and not a substring.

标签：Hash Table;Two Pointers;String 难度：Medium

3.2 Description

给定一个字符串s，找出这个字符串中所有子串中没有重复字符的最长子串，并返回这个子串的长度。比如给出的例子中，字符串*"abcabcbb"*中没有重复字符的最长子串有*"abc"*，那么结果就是3。

3.3 Solution #1

最简单的思路：暴力搜索。那就需要得到一个字符串的所有子串，然后依次判断每个子串是否包含重复的字符，每次判断的时候更新需要返回的值。那么，就有两个问题需要解决：

- 如何找到一个字符串中的所有子串
- 如何判断一个字符串是否包含重复的字符

对于第一个问题，我们需要两个游标*begin*和*end*，分别指示一个子串左右的下标。需要注意的是，在这里我们统一规定一个子串是指原字符串中范围是[*begin*,*end*)，即不包含下标是*end*的字符。这样*begin*的取值范围就是$0 \leq begin \leq n-1$，*end*的取值范围就是$1 \leq end \leq n$，其中n是字符串的长度。有了这两个游标之后，我们只需要一个双层循环就可以得到所有的子串。在这个双层循环中，两个游标的关系是：$0 \leq begin < end \leq n$。

对于第二个问题，要想判断一个字符串中是否有重复的字符，就需要存储已经存在的字符，正好Java中的set可以满足需求。给定一个字符串之后，遍历这个字符串并得到每个位置上的字符，每访问一个字符时，判断这个字符是否在set中已经存在，如果存在直接返回false；如果不存在，将这个字符加入set中继续判断，直到遍历完所有的字符。

下面的代码就是上面思路的Java实现。

```
public class Solution {
    public int lengthOfLongestSubstring(String s) {
        int ret=0;
        int length=s.length();
        for(int i=0;i<length;i++){
            for(int j=i+1;j<=length;j++){
                if(isUnique(s,i,j)){
                    ret=ret>(j-i)?ret:(j-i);
                }
            }
        }
```

```
        return ret;
    }
    public boolean isUnique(String s,int begin,int end){
        Set<Character> set=new HashSet<>();
        for(int i=begin;i<end;i++){
            Character ch=s.charAt(i);
            if(set.contains(ch))
                return false;
            set.add(ch);
        }
        return true;
    }
}
```

复杂度分析

- 时间复杂度：$O(n^3)$。在查找所有子串的过程中，复杂度是$O(n^2)$，对于每个子串，都要检查是否存在重复字符，因此整个过程的复杂度是$O(n^3)$。
- 空间复杂度：$O(min(n,26))$。这个过程中唯一需要分配空间的地方在于检查字符串是否含有重复字符的地方，需要创建一个set，由于最多只有26个字母，所以最多需要$O(min(n,26))$的空间，其中n是字符串的长度。

3.4 Solution #2

方法1虽然简单，但是太过粗暴，消耗的时间太长了。那么怎么优化呢？ 在这个问题中，有两个主要问题需要解决，在方法1中已经介绍了。那么是否有必要检查所有的子串呢？当然不是，如果一个字符串s_{ij}，即下标为i到下标j-1的字符串中没有重复的字符，那么我们只需要检查下标为j的字符是否在字符串s_{ij}中，而不是重新检查字符串s_{ij+1}。同时，也不需要对每个待检测子串进行是否含有重复字符的检查，我们可以维护一个*HashSet*，这里保存当前子串中所有出现的字符。这样，对于下一个字符，检查的时间就降到了$O(1)$。

上面的思路其实就是活动窗口的思想，滑动窗口经常用在数组的问题中。一个滑动窗口由两个游标组成，就像上面讨论的*begin*和*end*那样，组成的范围就是[*begin*,*end*)。滑动的含义就是，两个坐标会向一个特定的方向移动。如果窗口向右移动一个位置，那么这个窗口的范围就是[*begin*+1,*end*+1)。

有了滑动窗口，这个问题就好解决了。首先我们使用*HashSet*来存储滑动窗口[*begin*,*end*)中出现的字符（最开始的时候*begin*=*end*）。然后我们将*end*向右移动一个位置并检查字符s_{end+1}是否已经在*HashSet*中存在。如果已经存在，那就需要将*begin*向

右移动一个位置了，同时将s_{begin}从*HashSet*中去除；如果不存在，说明找到了一个更长的符合条件的子串，然后计算这个子串的长度并更新返回值。继续这个动作，直到检查完所有的字符。下面是Java的实现代码。

```
public class Solution {
    public int lengthOfLongestSubstring(String s) {
        int ret=0;
        int n=s.length();
        Set<Character> set=new HashSet<>();
        int i=0,j=0;
        while(i<n && j<n){
            if(!set.contains(s.charAt(j))){
                set.add(s.charAt(j));
                j++;
                ret=ret>(j-i)?ret:(j-i);
            }else{
                set.remove(s.charAt(i));
                i++;
            }
        }
        return ret;
    }
}
```

复杂度分析

- 时间复杂度：$O(2n) = O(n)$。在最坏的情况下，字符串中的每个字符都需要访问两次（分别被*begin*和*end*访问，比如字符串"bbbbb"）；最好的情况下，字符串只被访问一次（比如字符串"abcdef"）。
- 空间复杂度：$O(min(n, 26))$。和方法1一样。

3.5 Solution #3

方法2中使用了*HashSet*来存储正在访问的子串中已经出现的字符。不过，这个滑动窗口在字符s_{end}出现的时候，下标*begin*是一个一个向右移动的。这没有必要，如果我们知道已经出现的这个字符的下标，比如是j'，那么可以直接将*begin*右移到$j' + 1$。下面是Java的实现。

```
public class Solution {
    public int lengthOfLongestSubstring(String s) {
        int ret=0;
        int n=s.length();
```

```
        Map<Character,Integer> map=new HashMap<>();
        for(int i=0,j=0;j<n;j++){
            if(map.containsKey(s.charAt(j))){
                i=i>map.get(s.charAt(j))?i:map.get(s.charAt(j));
            }
            ret=ret>(j-i+1)?ret:(j-i+1);
            map.put(s.charAt(j),j+1);
        }
        return ret;
    }
}
```

复杂度分析

- 时间复杂度：$O(n)$。下标j访问整个字符串一次。
- 空间复杂度：$O(min(n,26))$。和方法2一样。

4 Median of Two Sorted Arrays

4.1 Question

There are two sorted arrays *nums1* and *nums2* of size m and n respectively.Find the median of the two sorted arrays. The overall run time complexity should be $O(log(m+n))$.

Example 1: nums1 = [1, 3] nums2 = [2] The median is 2.0

Example 2: nums1 = [1, 2] nums2 = [3, 4] The median is (2 + 3)/2 = 2.5

标签：Binary Search;Array;Divide and Conquer 难度：Hard

4.2 Description

这道题给了两个已经排好序的数组*nums1*和*nums2*，它们的大小分别是m和n。求出这两个数组的中位数。中位数就是排好序的数组中最中间的那个数。如果元素个数是两个，那么中位数就是中间那两个数字的平均数。

4.3 Solution #1

最简单的方法就是将两个数组合并为一个有序的数组，然后就可以直接得到这个数组的中位数了。这个方法的思路很简单，就是重新构造一个长度为$m+n$的新数组。Java实现的代码如下：

```
public class Solution {
    public double medianOfTwoSortedArrays(int[] nums1,int[] nums2) {
        int m=nums1.length;
        int n=nums2.length;
        int[] nums=new int[m+n];
        int i=0,j=0,k=0;
        while(i<m && j<n){
            if(nums1[i]<nums2[j])nums[k++]=nums1[i++];
            else nums[k++]=nums2[j++];
        }
        while(i<m)nums[k++]=nums1[i++];
        while(j<n)nums[k++]=nums2[j++];
        return (nums[m+n]+nums[m+n-1])/2.0;
    }
}
```

复杂度分析

- 时间复杂度：$O(m+n)$。为了构造新数组，需要将两个数组都遍历一遍，所以时间复杂度是$O(m+n)$。
- 空间复杂度：$O(m+n)$。代码里声明了一个新数组，这个数组的大小是两个数组的长度之和，所以空间复杂度是$O(m+n)$。

4.4 Solution #2

上面的方法1通过构造一个新的数组来得到中位数。但是时间复杂度是$O(m+n)$，题目中要求的时间复杂度是$O(log(m+n))$，虽然提交后能够*Accept*。那么如何进行优化呢？这就需要二分查找了。

首先，我们再一次明确中位数的定义和特点。中位数将一个有序数组分为了长度相等的两个部分，并且其中一个部分中的任何一个元素都比另一个部分的任何元素大。为了将一个数组分为两个部分，可以随意选择一个分割点i：

nums[0],nums[1],...,nums[i-1] | nums[i],nums[i+1],...,nums[m-1]

既然数组有m个元素，那么i就有$m+1$种取值可能(0-m)。当$i=0$时，说明左边的那个部分为空；当$i=m$时，说明右边那个部分为空。

回到题目。为了得到两个数组的中位数，需要将这两个数组都分为两个部分，然后分得到的左面两个部分组合，右面两个部分组合。如下：

num1s[0],nums1[1],...,nums1[i-1] | nums1[i],nums1[i+1],...,nums1[m-1]

num2s[0],nums2[1],...,nums2[j-1] | nums2[j],nums2[j+1],...,nums2[n-1]

其中，数组*nums2*的长度是n，分割点使用j表示。为了找到中位数，这个分隔方法应该满足下面的两个条件：

- left.length=right.length
- max(left)≤min(right)

那么我们就可以得到中位数了，中位数就是$(max(left)+min(right))/2.0$。（这里先放下边界条件。边界条件就是当i=0或i=m或j=0或j=n时的情况）

也就是说，我们只需要寻找合适的i（左边部分的长度是$i+j=(m+n+1)/2$，即$j=(m+n+1)/2$）就可以了。i应该满足的条件是：

- $nums2[j-1] \leq nums1[i]$
- $nums1[i-1] \leq nums2[j]$

可以使用二分查找来找到这个合适的i。由于$i+j=(m+n+1)/2$，所以为了方便，我们从两个数组中长度较小的那个数组查找，即令nums1是两个数组中元素个数少的那个数组。这样，i的取值范围就是[0,m]。下面给出了具体的查找方法：

- 令二分查找的左右两个端点是*imin*和*imax*，初始化为*imin=0,imax=m*。然后开始在[imin,imax]中进行查找；
- 设置*i=(imin+imax)/2,j=(m+n+1)/2-i*；
- 由于设置了*j=(m+n+1)/2-i*，我们保证了left.length=right.length。接下来，会有三种情况：
  - $nums2[j-1] \leq nums1[i]$并且$nums1[i-1] \leq nums2[j]$：意味着我们找到了合适的i，停止搜索；
  - $nums2[j-1] < nums1[i]$：说明nums1[i]的值过大，需要减少nums1[i]的值，也就是说要减少i的值。可以令$imax=i-1$，并回到第二步继续搜索；
  - $nums1[i-1] < nums2[j]$：说明nums1[i-1]的值过小，需要增加nums1[i-1]的值也就是说增加i的值。可以令$imin=i+1$，并返回第二步继续搜索；

当找到合适的i时，中位数就是：

$$ans=\begin{cases}\max(\text{nums1[i-1],nums2[j-1]}) & \text{当}m+n\text{是奇数时}\\ (\max(\text{nums1[i-1],nums2[j-1]})+\min(\text{nums1[i],nums2[j]}))/2.0 & \text{当}m+n\text{是偶数时}\end{cases}$$

接下来看看边界情况。其实，当$i = 0^{-}i = m^{-}j = 0^{-}j = n$，情况比较简单，这说明某个数组被完全分到了其中某一个部分中。比如，当$i = 0$时，说明数组*nums1*被完全分到了*right*那个部分。其它情况也是类似。下面就是Java的实现代码：

```
public class Solution {
    public double findMedianSortedArrays(int[] nums1, int[] nums2) {
        int m=nums1.length,n=nums2.length;
        //change nums1 and nums2 so that m<n
        if(m>n){
            int[] temp=nums1;nums1=nums2;nums2=temp;
            int t=m;m=n;n=t;
        }

        int imin=0,imax=m;
        int i,j,half=(m+n+1)/2;

        int leftMax=0,rightMin=0;

        //begin binary search
        while(imin<=imax){
            i=(imin+imax)/2;
            j=half-i;

            if(i>0 && j<n && nums1[i-1]>nums2[j]){
                //means nums1[i-1] is too big
                imax=i-1;
            }else if(j>0 && i<m && nums2[j-1]>nums1[i]){
                //means nums1[i] is too small
                imin=i+1;
            }else{
                //means we found the right i
                //discuss the eage situation
                if(i==0){
                    leftMax=nums2[j-1];
                }else if(j==0){
                    leftMax=nums1[i-1];
                }else{
                    leftMax=Math.max(nums1[i-1],nums2[j-1]);
                }

                if((m+n)%2==1){
                    return leftMax;
                }
```

```
                if(i==m){
                    rightMin=nums2[j];
                }else if(j==n){
                    rightMin=nums1[i];
                }else{
                    rightMin=Math.min(nums1[i],nums2[j]);
                }
                return (leftMax+rightMin)/2.0;
            }
        }
        return 0;
    }
}
```

复杂度分析

- 时间复杂度：$O(log(min(m,n)))$。程序相当于进行了一次二分查找，查找的范围就是[0,m]，所以时间复杂度是$O(log(min(m,n)))$。
- 空间复杂度：$O(1)$。

5 Longest Palindromic Substring

5.1 Question

Given a string S, find the longest palindromic substring in S. You may assume that the maximum length of S is 1000, and there exists one unique longest palindromic substring.

标签：String 难度：Medium

5.2 Description

给定一个字符串s，返回s中的最长回文子串。这里假设每一个字符串都有一个唯一的最长回文子串，并且最大的长度不超过1000。所谓的回文，就是正反两个方向阅读是一样的，比如*"aba"*就是一个回文字符串，而*"abc"*就不是一个回文字符串。

要注意一个常见的错误。最可能想到的就是，首先将s逆序得到s'，然后求s和s'的最长公共子串就是结果。这是不正确的，虽然有的时候会得到正确的答案。比如，当s=*"caba"*，则s'=*"abac"*，最长公共子串就是*"aba"*，这是正确答案。但是，如果s=*"abcdfgdcba"*，则s'=*"abcdgfdcba"*，最长公共子串就是*"abcd"*，显然不是回文的。所以不能简单使用最长公共子串来计算。

5.3 Solution #1

最简单的还是暴力搜索。首先找到所有的子串，判断每个子串是否是回文的，然后更新最长的回文子串即可。Java实现代码如下：

```
public class Solution{
    public String longestPalindrome(String s){
        int n=s.length();
        if(n==0)return '''';
        String ans=s.substring(0,1);
        for(int i=0;i<n;i++){
            for(int j=0;j<n;j++){
                String s1=s.substring(i,j-i+1);
                if(isPalindrome(s1) && s1.length()>ans.length()){
                    ans=s1;
                }
            }
        }
        return ans;
    }
    private boolean isPalindrome(String s){
        int n=s.length();
        int i=0,j=n;
        while(i<=j){
            if(s.charAt(i++)!=s.charAt(j--)){
                return false;
            }
        }
        return true;
    }
}
```

复杂度分析

- 时间复杂度：$O(n^3)$。寻找一个字符串的所有子串需要$O(n^2)$，检查一个子串是否是回文的需要$O(n)$，所以整个过程的时间复杂度是$O(n^3)$。
- 空间复杂度：$O(1)$。

5.4 Solution #2

在绝大多数的问题中暴力搜索都不是一个好的方法。为了降低整个过程的时间，需要使用特殊的方法。对于这个问题，可以使用动态规划，以空间换时间来降低时间复杂

度。首先创建一个二维表P，表的大小就是字符串s的长度n。表中的某一个位置P_{ij}中的值表示子串s_{ij}是否是回文的。P的定义如下：

$$P_{ij} = \begin{cases} \text{true} & \text{当子串} s_{ij} \text{是回文字符串时} \\ \text{false} & \text{其它情况} \end{cases}$$

因此，就有下面的公式：

$$P_{ij} = (P_{i+1,j-1} \&\& s_i == s_j)$$

对于基本情况，有：

$$P_{i,i} = true$$

$$P_{i,i+1} = (s_i == s_{i+1})$$

这样就得到了当子串长度是1和2时的情况。然后根据这个基本情况，依次增加子串的长度进行循环检测。下面是Java的实现代码：

```
public class Solution{
    public String longestPalindrome(String s){
        int n=s.length();
        if(n==0)return "";
        int maxLen=1;
        int longestBegin=0;
        boolean[][] table=new boolean[n][n];
        for(int i=0;i<n;i++){
            table[i][i]=true;
        }
        for(int i=0;i<n-1;i++){
            table[i][i+1]=(s.charAt(i)==s.charAt(i+1));
        }

        for(int len=3;len<=n;len++){
            for(int i=0;i<n-len+1;i++){
                int j=len+i-1;
                if(table[i+1][j-1] && s.charAt(i)==s.charAt(j)){
                    table[i][j]=true;
                    longestBegin=i;
                    maxLen=len;
                }
            }
        }
        return s.substring(longestBegin,longestBegin+maxLen);
    }
}
```

复杂度分析：

- 时间复杂度：$O(n^2)$。
- 空间复杂度：$O(n^2)$。

5.5 Solution #3

方法2中使用一个二维数组*table*来记录所有子串是否是回文字符串的信息，导致空间复杂度是$O(n^2)$。其实，我们可以将这个空间复杂度降到$O(1)$。这就需要从中间扩展的方法。

一个回文的字符串的长度可能是奇数也可能是偶数。当长度是奇数时，这个字符串以中间那个字符为中心，左右两边对称；当长度是偶数时，这个字符串以最中间的空格为中心，左右两边对称。这样，对于一个长度是n的字符串来说，就有*2n-1*个中心可能存在回文子串（n个字符加上*n-1*个空格）。那么，如果子串s_{ij}是回文的，并且字符$s_{i-1} == s_{j+1}$，那么子串$s_{i-1,j+1}$也是回文的。这就是从中间进行扩展。下面是Java的实现代码：

```
public class Solution{
    public String longestPalindrome(String s){
        int n=s.length();
        if(n==0)return "";
        String ans=s.substring(0,1);
        for(int i=0;i<n-1;i++){
            String s1=expandAroundCenter(s,i,i);
            String s2=expandAroundCenter(s,i,i+1);
            if(s1.length()>ans.length()){
                ans=s1;
            }
            if(s2.length()>ans.length()){
                ans=s2;
            }
        }
        return ans;
    }
    private String expandAroundCenter(String s,int left,int right){
        int i=left,j=right;
        int n=s.length();
        while(i>=0 && j<n && s.charAt(i)==s.charAt(j)){
            i--;
            j++;
        }
```

```
            return s.substring(i+1,j);
        }
    }
```

复杂度分析：

- 时间复杂度：$O(n^2)$。和方法2一样。
- 空间复杂度：$O(1)$。

6 ZigZag Conversion

6.1 Question

The string "PAYPALISHIRING" is written in a zigzag pattern on a given number of rows like this: (you may want to display this pattern in a fixed font for better legibility)

```
P   A   H   N
A P L S I I G
Y   I   R
```

And then read line by line: "PAHNAPLSIIGYIR" Write the code that will take a string and make this conversion given a number of rows:

string convert(string text, int nRows); convert("PAYPALISHIRING", 3) should return "PAHNAPLSIIGYIR".

标签：String 难度：Easy

6.2 Description

这其实就是一个字符串的转换，将一个字符串变换成如下图的形式：

6.3 Solution

没什么难度，相当于对字符数组的操作，下面是Java的实现代码：

```
public class Solution{
    public String convert(String s,int nRows){
        int n=s.length();
        StringBuffer[] sb=new StringBuffer[nRows]();
        int i;
        for(i=0;i<nRows;i++){
            sb[i]=new StringBuffer();
        }
        i=0;
        while(i<n){
            for(int d=0;d<nRows && i<n;d++){
                sb[d].append(s.charAt(i++));
            }
            for(int u=nRows-2;u>0 && i<n;u--){
                sb[u].append(s.charAt(i++));
            }
        }
        for(i=1;i<nRows;i++){
            sb[0].append(sb[i]);
        }
        return sb[0].toString();
    }
}
```

复杂度分析

- 时间复杂度：$O(n)$。就是对字符串进行一次遍历即可。
- 空间复杂度：$O(n)$。就需要构造一个与原来字符串长度相同的字符串。

7 Reverse Integer

7.1 Question

Reverse digits of an integer.

Example1: x = 123, return 321

Example2: x = -123, return -321

标签：Math 难度：Easy

7.2 Description

这道题就是将一个整数反转过来，没什么难度，需要注意的有两点：

- 注意后几位是0的情况，比如1000反转后是1；
- 注意溢出。对于整数只有32位的机器来说，如果原来的数是1000000003，那么反转后就会溢出。如果出现这种情况，返回0。

7.3 Solution

首先得到原数的标志，是正还是负。之后一个循环就可以完成。在检查是否溢出时，首先进行反转，然后用反转的变回反转前的数，如果两个数相等说明没有发生溢出；如果不等，说明发生了溢出。下面是Java的实现代码：

```
public class Solution{
    public int reverse(int x){
        int sign=x>0?1:-1;
        int ans=0;
        x*=sign;
        while(x>0){
            int least=x%10;
            int t=ans;
            ans=ans*10+least;
            int test=(ans-least)/10;
            if(test!=t)return 0;
            x/=10;
        }
        return ans*sign;
    }
}
```

复杂度分析：

- 时间复杂度：$O(log_{10}n)$。只需要一个循环即可，循环的次数就是这个整数的位数。
- 空间复杂度：$O(1)$。

8 String to Integer (atoi)

8.1 Question

Implement atoi to convert a string to an integer.

Hint: Carefully consider all possible input cases. If you want a challenge, please do not see below and ask yourself what are the possible input cases.

Notes: It is intended for this problem to be specified vaguely (ie, no given input specs). You are responsible to gather all the input requirements up front.

标签：Math,String 难度：Easy

8.2 Description

这道题就是实现一个将字符串转化为数字的函数。虽然简单，但还是要注意一些特殊情况：

- 忽略字符串开始的空格：比如字符串" 123"对应的数字是123；
- 字符串的数字前的符号"+"和"-"是可选的：比如"123"和"+123"都对应数字123，"-123"对应数字-123；但是只能有一个；
- 在"+"或"-"符号后面紧跟着一系列数字字符，如果期间出现非数字字符，就停止转换。比如"123f456"对应的数字是123；
- 如果转换后发生溢出，返回int类型的边界值。比如(2147483647)或(-2147483648)。这就需要检查是否溢出；

考虑了这些情况之后，就应该没有问题了。

8.3 Solution

下面是Java的实现代码：

```
public class Solution{
    public int myAtoi(String str){
        String s=str.trim();
        int n=s.length();
        int ans=0;
        int sign=1;
        for(int i=0;i<n;i++){
            if(s.charAt(i)=='+')
                i++;
            else if(s.charAt(i)=='-'){
                i++;
                sign=-1;
            }
            int digit=s.charAt(i)-'0';
```

```
            if(Integer.MAX_VALUE/10<ans || Integer.MAX_VALUE/10==ans &&
                Integer.MAX_VALUE%10<digit)
                return sign==1?Integer.MAX_VALUE:Integer.MIN_VALUE;
            ans=ans*10+digit;
        }
        ans*=sign;
        return ans;
    }
}
```

复杂度分析：

- 时间复杂度：$O(n)$。一个循环就可以，循环的次数就是字符串的长度。
- 空间复杂度：$O(1)$。

9 Palindrome Number

9.1 Question

Determine whether an integer is a palindrome. Do this without extra space.

标签：Math 难度：East

9.2 Description

这道题就是判断一个数字是否是回文的。方法有很多，首先可以转换为字符串，然后判断字符串是否是回文的。不过题目中要求不能使用额外的空间。然后还有可以将数字反转，然后判断两个数字是否相同，不过要注意反转的时候不要溢出。最后，就是数字的第一位和最后一位判断是否相等，然后从两边向中间移动，依次判断。

9.3 Solution

下面是Java的实现代码：

```
public class Solution{
    public boolean isPalindrome(int x){
        if(x<0)return false;
        int div=1;
        while(x/div>=10)div*=10;
        while(x>0){
            int l=x/div;
            int r=x%10;
```

```
            if(l!=r)return false;
            x=(x%div)/10;
            div/=100;
        }
        return true;
    }
}
```

复杂度分析：

- 时间复杂度：$O(log_{10}n)$。循环的次数就是数字的位数的一半。
- 空间复杂度：$O(1)$。