
Table of Contents

前言	1.1
目前收录的题目	1.2
双指针	1.3
两数求和问题[E]	1.3.1
三数组求和问题[M]	1.3.2
三数组问题变种[M]	1.3.3
四数组问题[M]	1.3.4
两个指针解决装水问题[M]	1.3.5
删除有序数组重复元素[E]	1.3.6
删除重复元素[E]	1.3.7
二分查找	1.4
搜索范围[M]	1.4.1
搜索插入位置[M]	1.4.2
猜数问题[E]	1.4.3
*旋转后的二分查找[H]	1.4.4
字符串	1.5
不重复子串问题[M]	1.5.1
最长回文串[M]	1.5.2
链表	1.6
两个大数相加[M]	1.6.1
*单链表求倒数问题[E]	1.6.2
合并两个链表[E]	1.6.3
合并多个链表[H]	1.6.4
交换节点对[E]	1.6.5
队列和堆栈	1.7
数字键盘字母组合问题[M]	1.7.1
数学	1.8
转置数字——解决溢出的思路[E]	1.8.1
atoi——培养严谨的思路，正负号的处理技巧[E]	1.8.2
回文数字巧解[E]	1.8.3

位运算实现除法[M]	1.8.4
找规律	1.9
ZigZag解码[E]	1.9.1
Nim的游戏[E]	1.9.2
查表	1.10
拉丁数字转罗马数字[M]	1.10.1
罗马数字转拉丁数字[E]	1.10.2
位操作	1.11
求子集[M]	1.11.1
分治 (Divide & Conquer)	1.12
*两个有序数组中的中位数和Top K问题[H]	1.12.1
动态规划 (DP)	1.13
*正则匹配问题[H]	1.13.1
三角形问题[M]	1.13.2
计算二进制数中1的个数[M]	1.13.3
*括号匹配问题[M]	1.13.4
最短路径和[M]	1.13.5
贪心(Greedy)	1.14
未在上面列出的题目	1.15
014. Longest Common Prefix[E]	1.15.1
020. Valid Parentheses[E]	1.15.2
028. Implement strStr()[E]	1.15.3

LeetBook

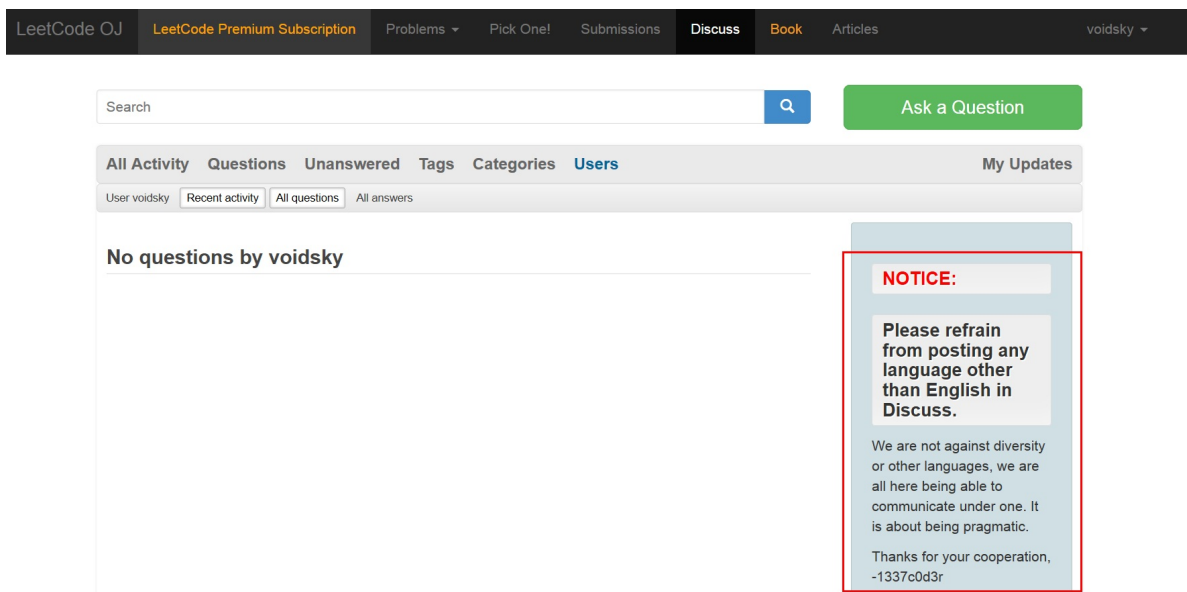
By [Voidsky](#)(黄锴)

写在前面

本人一直有写书的“小梦想”，机缘巧合碰到gitbook，也算看缘分，虽然我知道这本“书”远达不到想象中的书的概念，但是，我仍然视其我第一本书。尽管它不能出版，也不会被人广泛阅读，但却值得我小心呵护。我希望通过自己的努力能慢慢的让这本“不是书的书”变得丰满起来。也算是给我最后这段校园时光留下点什么印记。

2015-05-05:

今天，突然发现leetcode上我写的所有的中文思路贴都被删了，心情还是有些低落，我看见了站长对我的“善意提醒”，他说的也没错，毕竟是在别人的地盘。



不过，在短短几天，我看见了大家对我的支持，我很感动，也许没机会再在leetcode上用中文发思路了，但是我还是想坚持把这本书完成，虽然不一定会有很多人看了，怎么说呢，坚持吧！

Activity by voidsky

Score: **7,300** points (ranked #142)

Questions: **0**

Answers: **2**

Comments: **18**

Voted on: **4** questions, **0** answers

Gave out: **4** up votes, **0** down votes

Received: **144** up votes, **1** down vote

LeetBook介绍

LeetBook电子书的地址：<https://www.gitbook.com/book/hk029/leetbook/details>

LeetBook理论上不是一本书，它是我做leetcode已来的一些笔记，总结。

本“书”不是一蹴而就的，是本人在刷题过程中做的笔记逐步积累起来的，并会随着我的刷题而变得丰富。

每道题后面的[E][M][H]代表了这个题的难度等级，分别是轻松，略有难度，很难

强调：本人并不是什么大神，只是喜欢整理归纳，喜欢把别人的思维转换成自己更易懂的形式。在刷题过程中，会有很多自己的解题思路和感想，也会有很多灵光一现的瞬间，我习惯用马克飞象（一个支持印象笔记的第三方markdown编辑器）把这些瞬间捕捉下来，所以我很多笔记都存在印象笔记中。而偶然间遇到gitbook，又刚好支持markdown，这也促成就是这本书的诞生。

LeetBook怎么看

这本“书”你可以横着看，竖着看，躺着看（开玩笑），怎么看取决于你了，我讲讲我是怎么组织这本书的：

- 我在第二页写了一个目录，包含了所有收录的题目，你可以从这里入手，查找你需要的题目找踢解
- 我把题型分类了，我会选择一部分我觉得能学到点东西的题目放在这里面，你可以通过侧栏寻找感兴趣的。

为什么要写做这个？

我相信很多为了找工作都开始刷LeetCode了，我也不例外，LeetCode目前应该是找工作领域最权威的刷题网站了，上面的题目都很有代表性，也是很多公司喜欢问的，所以也成就了现在刷LeetCode的狂热。

但是，很多人都发现了一个问题，只是一味的刷题，却没有机会对每个题目做一个整理或总结。导致题目做过一遍，但是没什么印象，很多题目虽然A过了，但是实际上用的是比较笨，或者比较费时的思路，有更多更精彩，更简单的思路没有考虑到。导致一遍下来收获并不大。

所以这里推出这本书，一方面，方便大家A题后来复习看看，也方便大家来这里找每道题新的思路。也督促自己对每道题尽可能的想全面点。

这本书力求做到

1. 覆盖尽可能多的题目
2. 每道题力求多种思路解决
3. 解答思路尽可能清晰，易懂

这也算对自己的一个鞭策，每道题让自己考虑出更加容易理解的思路，也方便日后的复习。

声明

1. 它不是一本算法或数据结构书，这毕竟是一本有关解题思路的“书”，它默认你是有一定的算法和数据结构基础的，如果你在这方面还十分欠缺，推荐你看下下面相关推荐的书（我以后有时间可能会结合自己的能力加上每类题型的讲解，相关算法等）
2. 关于语言，这里不推荐任何语言，我觉得语言只是工具，思路才是最重要的，所有的思路其实都可以快速的用特定的语言实现，大多就是语法上的细微区别，我觉得大家还是不要把注意力放在特定的实现上。我也不会在这道题的时候拘泥于特定的语言，之前我大部分是使用c++，但是有些题目用python确实方便我也会用python，之后我A题主要使用java(考虑以后工作可能会用java),
3. 这本“书”目前是我一个人闲暇时间完成，所以更新速度上不会特别快，大概每周4-6篇，跟我这周做题时间有关。

相关推荐

- 书籍推荐(以下链接都是豆瓣读书的链接，可以方便你了解详情)
 1. 《算法》（红宝书，书内有能运行的代码，大量图示，而且讲解的通俗易懂，作者是普林斯顿大学计算机学院院长，Kuth的徒弟，算法界的牛人！这里我不打算推荐算法导论，因为它太深了，不适合以找工作为目的的同学）
 2. 《大话数据结构》（数据结构就推荐这本，国内教材都过于晦涩，这个适合自学）
 3. 《编程珠玑》（你会很明确知道什么叫精华）
 4. 《编程之美》（微软的面试宝典）
 5. 待续
- 博客推荐
 1. blog.csdn.net/hk221976（不要脸的先推荐了自己的博客 -- 捂脸）
 2. <http://www.nowamagic.net/> 简明现代魔法原本是一个个人博客，现在已经发展成一个清新的、动漫风的、专注于将Web开发技术尽可能简明易懂地描述出来的IT社区，在这里你可以轻松愉快地学习Web开发技术。

关于评论

目前已经加上了评论功能，可以在每一页的底部发现输入评论。可以用自己的社交账号登录，评论功能由多说提供支持。



联系我

毕竟是自己一个人的工作，虽然每道题我都力求做到完善。但是可能还是会出现一些不足，如果有什么问题或建议，可以联系我：

- 黄锴
 - 邮箱 hk2291976@hotmail.com
 - 简书 http://www.jianshu.com/users/30f737ee0051/latest_articles
 - 个人主页 [Voidsky](#)
 - CSDN博客 blog.csdn.net/hk221976
 - qq 263791865

更新情况

1.2版本

1. 更新题目到30多道
2. 新增章节“双指针”

1.1版本

1. 加入katex,支持Latex算数表达式
2. 加入了多说评论插件，可以支持评论
3. 目前题目更新到21题

1.0版本

这是最原始的版本，完善了首页介绍和书的框架，加入了目前整理的解题思路

目前收录的题目

为什么开这样一页，是因为分类的情况下，大家找题目可能不方便，为了方便大家找题目开了这一页，所有题目按顺序排列，大家可以用`ctrl+f`找到自己需要的题目

- [001. Two Sum\[E\]](#)
- [002. Add Two Numbers \[M\]](#)
- [003. Longest Substring Without Repeating Characters\[M\].](#)
- [004. Median of Two Sorted Arrays\[H\]](#)
- [005. Longest Palindromic \[M\]](#)
- [006. ZigZag Conversion\[E\]](#)
- [007. Reverse Integer\[E\]](#)
- [008. String to Integer \(atoi\) \[E\]](#)
- [009. Palindrome Number\[E\]](#)
- [010. Regular Expression Matching\[\[H\]](#)
- [011. Container With Most Water\[M\]](#)
- [012. Integer to Roman\[M\]](#)
- [013. Roman to Integer\[E\]](#)
- [014. Longest Common Prefix\[E\]](#)
- [015. 3Sum\[M\]](#)
- [016. 3Sum Closest \[M\]](#)
- [017. Letter Combinations of a Phone Number\[M\]](#)
- [018. 4Sum\[M\]](#)
- [019. Remove Nth Node From End of List\[E\]](#)
- [020. Valid Parentheses\[E\]](#)
- [021. Merge Two Sorted Lists\[E\]](#)
- [022. Generate Parentheses\[M\]](#)
- [023. Merge k Sorted Lists \[H\]](#)
- [024. Swap Nodes in Pairs\[E\]](#)
- [026. Remove Duplicates from Sorted Array\[E\]](#)
- [027. Remove Element\[E\]](#)
- [028. Implement strStr\(\)\[E\]](#)
- [029. Divide Two Integers\[M\]](#)
- [033. Search in Rotated Sorted Array\[H\]](#)
- [034. Search for a Range\[M\]](#)
- [035. Search Insert Position\[M\]](#)
- [064. Minimum Path Sum\[M\]](#)
- [120. Triangle\[M\]](#)
- [292. Nim Game\[E\]](#)

- [338. Counting Bits \[M\]](#)
- [374. Guess Number Higher or Lower\[E\]](#)

双指针

介绍

开始，这章的题目是叫“数组”，但是，目前更名为双指针，这是因为我发现凡是数组的题目，大部分都是利用双指针去解决问题。

双指针，顾名思义，就是利用两个指针去遍历数组，一般来说，遍历数组采用的是单指针（index）去遍历，两个指针一般是在有序数组中使用，一个放首，一个放尾，同时向中间遍历，直到两个指针相交，完成遍历，时间复杂度也是 $O(n)$ 。

用法

一般会有两个指针 `front` , `tail` 。分别指向开始和结束位置。

```
front = 0;
tail = A.length()-1
```

一般循环结束条件采用的是判断两指针是否相遇

```
while(fron < tail)
{
    .....
}
```

对于in place交换的问题，循环结束条件一般就是其中一个指针遍历完成。

使用范围

一般双指针在有序数组中使用的特别多。（部分情况下，未排序数组也有应用） 一般用来解决下列问题（陆续补充中）：

1. 两数求和

一般这种问题是问，寻找两个数的和为一个特定的值（比如后面的N SUM问题），这时候，如果数组有序，我们采用两个指针，分别从前和后往中间遍历，`front`移动和增大，`tail`移动和减小，通过特定的判断，可以求出特定的和。

时间复杂度为 $O(n)$,如果用双重循环则要 $O(n^2)$ 。

2. in place 交换

数组的in place(就地)交换一般得用双指针，不然数组中添加或删除一个元素，需要移动大量元素。

这时候，一般是一个指针遍历，一个指针去找可以用来交换的元素。

001.Two Sum[E]

1. 题目

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.

Example:

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

2. 思路

作为我们可能进入leetcode遇到的第一题，大家对它可谓是又爱又恨，而且它确实又有很多解法，总结起来可以有以下3种，3种还都能A过....

2.1 双重循环

最简单的思路，两重循环，没啥技术含量，速度很慢，毕竟是 $O(N^2)$

```
class Solution {
public:
    vector<int> twoSum(vector<int>& nums, int target) {
        vector<int> result;
        for(int i = 0; i < nums.size(); i++)
        {
            for(int j = i+1; j < nums.size(); j++)
                if(nums[i] + nums[j] == target)
                {
                    result.push_back(i);
                    result.push_back(j);
                    return result;
                }
        }
    }
};
```

2.2 排序

其实简单的想，用一个排序都能把复杂度降到 $O(N\log N)$ ，通过排序，然后用两个指针从前扫描逼近真值(注意这个思想，可以让 $O(N*N)$ 的复杂度降为 $O(N)$ ，充分利用排序，因为一定会有一个值满足，然后通过值去原数组里找对应的下标（这里其实就可以考虑，如果当初就用一个数据结构存好键值对应关系就好了，其实就是hashmap，下面的做法就用到了）（下面代码是dugu9sword写的java代码，我就没写成c++的，主要看思路，还是比较好理解的）

```
public class Solution {
    public int[] twoSum(int[] nums, int target) {
        int[] nums_sorted=new int[nums.length];
        System.arraycopy(nums,0,nums_sorted,0,nums.length);
        //Quicksort.
        Arrays.sort(nums_sorted);

        //Find the two numbers. O(n)
        int start=0;
        int end=nums_sorted.length;
        while(start<end){
            while(nums_sorted[start]+nums_sorted[--end]>target);
            if(nums_sorted[end]+nums_sorted[start]==target)
                break;
            while(nums_sorted[++start]+nums_sorted[end]<target);
            if(nums_sorted[end]+nums_sorted[start]==target)
                break;
        }

        //find the indices of the two numbers
        int[] ret=new int[2];
        int index=0;
        int a=nums_sorted[start];
        int b=nums_sorted[end];
        for(int i=0;i<nums.length;i++)
            if(nums[i]==a || nums[i]==b)
                ret[index++]=i;
        return ret;
    }
}
```

2.3 Hashmap

最后一种是比较聪明的做法，用hashmap，hashmap是内部存储方式为哈希表的map结构，哈希表可以达到查找O(1)，哈希表的介绍可以[看这里](#)，C++实现Hashmap的方式，这里用[unordered_map](#)关联容器，可以实现键值对应。

```
class Solution {
public:
    vector<int> twoSum(vector<int>& nums, int target) {
        vector<int> result;
        unordered_map<int,int> mymap;
        int res;
        for(int i = 0;i < nums.size();i++)
        {
            res = target - nums[i];
            unordered_map<int,int>::iterator it = mymap.find(res);
            if(it != mymap.end())
            {
                return vector<int>({it->second,i});
            }
            mymap[nums[i]] = i;
        }
    }
};
```


015. 3Sum

问题

Given an array S of n integers, are there elements a, b, c in S such that $a + b + c = 0$? Find all unique triplets in the array which gives the sum of zero.

Note: Elements in a triplet (a,b,c) must be in non-descending order. (ie, $a \leq b \leq c$) The solution set must not contain duplicate triplets. For example, given array $S = \{-1\ 0\ 1\ 2\ -1\ -4\}$,

```
A solution set is:  
(-1, 0, 1)  
(-1, -1, 2)
```

思路

这个问题其实就是2 SUM的变种问题，和这个问题类似的还有4SUM，解题思路也可以参考2SUM。我们知道2SUM还可以用暴力两重循环解决，3SUM如果暴力就要三重循环，想想也可怕。

考虑一下如何将3SUM问题转变一下：如果我们随机确定了一个数 a ，问题是不是就变成了，在剩下的数里面找到2个数和为 $0-a$ ，是不是就和2SUM问题一样了？

其实这题相比2SUM多了几个难点：

1. 数组里允许重复的数
2. 结果要按升序排列
3. 结果中不能出现重复的结果

当然，我们可以通过写很多条判断语句解决这些问题，但是其实稍微想一下，可以发现，只要保证数组一开始就有序就好办很多了。

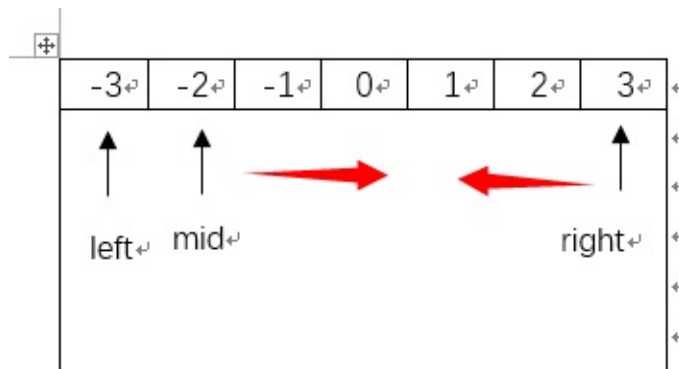
我们可以选择3个变量， $left$ ， mid ， $right$ 。在循环的时候，永远保证相对顺序就行了。这样在插入结果的时候，就自然是升序的。

我们可以参考2SUM的思路2解决这道题。

首先，我们考虑如何确定第一个数left，这肯定是我们第一层循环。第一个数可不能无限制的随便选，因为我们要保证上面的几个条件都满足，我们要保证它时刻是最小的数，那么我们可以考虑left取到全部非正数就行了。（如果要和为0，至少要有1个非正数）

```
for (int left = 0; left < nums.length && nums[left] <= 0; left++)
```

然后就是mid和right的确定了，我们采用思路2的方案，mid和right分别从两端往中央扫描，如果mid+right还比较小，那就需要mid右移，反之right左移



我们可以写出如下的代码:

```
mid = left+1; right = nums.length-1;
while(mid < right)
{
    int tmp = 0-nums[left];
    if(nums[mid] + nums[right] == tmp)
        addtolist;
    else if(nums[mid] + nums[right] < tmp)
        mid++;
    else
        right--;
}
```

一切看起来特别美好了，可以当你提交的时候，你会发现，还是会报错，因为它虽然能解决问题2，但是不能处理重复结果。举个最简单的例子：-2 -2 -1 -1 0 1 1 2 2 这个代码会输出数个[-2 0 2] [-1 0 1]，解决方案也很简单，如果一个left指向的数是之前判断过的，跳过，如果mid和right往中间移动的时候，是刚才的数，也跳过。

```
mid = left+1; right = nums.length-1;
while(mid < right)
{
    int tmp = 0-nums[left];
    //跳过left重复匹配
    if(left > 0 && nums[left] == nums[left-1])
        continue;

    if(nums[mid] + nums[right] == tmp)
    {
        int tmp_mid = nums[mid],tmp_right= nums[right];
        list.add(Arrays.asList(nums[left], nums[mid], nums[right]));
        //跳过right和mid的重复匹配
        while(mid < right && nums[++mid] == tmp_mid);
        while(mid < right && nums[--right] == tmp_right);
    }
    else if(nums[mid] + nums[right] < tmp)
        mid++;
    else
        right--;
}
```

代码

```
public class Solution {
    public List<List<Integer>> threeSum(int[] nums) {
        Arrays.sort(nums);
        List<List<Integer>> list;
        list = new ArrayList<List<Integer>>();
        int mid, right;
        //left只用循环所有的非正数就行了（不是负数是因为还要考虑[0 0 0]的情况所以是非正数）
        for (int left = 0; left < nums.length && nums[left] <= 0; left++) {
            mid = left+1; right = nums.length-1;
            int tmp = 0-nums[left];
            //跳过left重复匹配
            if(left > 0 && nums[left] == nums[left-1])
                continue;
            while(mid < right)
            {
                if(nums[mid] + nums[right] == tmp)
                {
                    int tmp_mid = nums[mid], tmp_right = nums[right];
                    list.add(Arrays.asList(nums[left], nums[mid], nums[right]));
                    //跳过right和mid的重复匹配
                    while(mid < right && nums[++mid] == tmp_mid);
                    while(mid < right && nums[--right] == tmp_right);
                }
                else if(nums[mid] + nums[right] < tmp)
                    mid++;
                else
                    right--;
            }
        }
        return list;
    }
}
```

16. 3Sum Closest [M]

Given an array S of n integers, find three integers in S such that the sum is closest to a given number, $target$. Return the sum of the three integers. You may assume that each input would have exactly one solution.

For example, given array $S = \{-1, 2, 1, -4\}$, and $target = 1$. The sum that is closest to the target is 2. $(-1 + 2 + 1 = 2)$.

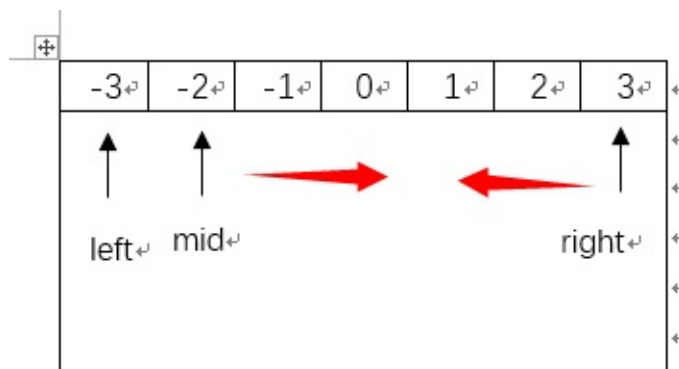
思路

这个问题等于是3SUM又升级了，因为这里现在是最接近的结果，所以Hashmap的思路基本没用了，因为必须得要循环找到所有可能。当然，这个题目有个限制就是只有唯一的结果，所以，如果发现完全相等的，也就可以停下来了。

但是解题思路还是可以参考3SUM，算法基本一样，有些地方需要修改一下。首先，因为是最接近的，所以要考虑所有情况， $left$ 循环到 $length-2$

```
for (int left = 0; left < nums.length-2; left++)
```

然后还是 mid 和 $right$ 分别从两端往中央扫描，如果 $mid+right$ 还比较小，那就需要 mid 右移，反之 $right$ 左移（每次如果有最小的就存下来）



我们可以写出如下的代码:

```

mid = left+1; right = nums.length-1;
while(mid < right)
{
    int tmp = target-nums[left];
    if(abs(tmp - nums[mid] + nums[right]) < abs(target-Min))
        Min = nums[left] + nums[mid] + nums[right];
    if(nums[mid] + nums[right] == tmp)
        return Min;
    else if(nums[mid] + nums[right] < tmp)
        mid++;
    else
        right--;
}

```

代码

```

public class Solution {
    public int threeSumClosest(int[] nums, int target) {
        Arrays.sort(nums);
        int mid, right;
        if(nums.length < 3)
            return 0;
        int Min = nums[0]+nums[1]+nums[2];
        //left要循环全部
        for (int left = 0; left < nums.length-2; left++) {
            mid = left+1; right = nums.length-1;
            int tmp = target-nums[left];
            while(mid < right)
            {
                if(Math.abs(tmp - nums[mid] - nums[right]) < Math.abs(target - Min)) /
                /每次查看是不是最小的情况
                Min = nums[left]+ nums[mid] + nums[right];
                if(nums[mid] + nums[right] == tmp)
                {
                    return target;    //因为只有一种答案所以可以直接返回
                }
                else if(nums[mid] + nums[right] < tmp)
                    mid++;
                else
                    right--;
            }
        }
        return Min;
    }
}

```


018. 4Sum

问题

Given an array S of n integers, are there elements a, b, c, and d in S such that $a + b + c + d = \text{target}$? Find all unique quadruplets in the array which gives the sum of target.

Note: Elements in a quadruplet (a,b,c,d) must be in non-descending order. (ie, $a \leq b \leq c \leq d$)
The solution set must not contain duplicate quadruplets.

For example, given array S = {1 0 -1 0 -2 2}, and target = 0.

A solution set is: (-1, 0, 0, 1) (-2, -1, 1, 2) (-2, 0, 0, 2)

思路

好了，2SUM，3SUM，3SUM Closet 都解决了，我们顺利迎来了最后一个boss，4SUM，但是我们已经见怪不怪了。老思路，把4SUM问题变成3SUM问题。

- 先排序
- 确定一个数，然后文件顺利变成3SUM问题，然后就是完完全全3SUM的解决思路了。

```
for(int i = 0; i < nums.length-3; i++)
{
    int target_i = target - nums[i];
    if(i > 0 && nums[i] == nums[i-1]) //排除一样的数
        continue;
    .....//3SUM问题
}
```

整体代码


```

public class Solution {
    public List<List<Integer>> fourSum(int[] nums, int target) {

        Arrays.sort(nums);
        List<List<Integer>> list;
        list = new ArrayList<List<Integer>>();
        int mid, right;
        for(int i = 0; i < nums.length-3; i++)
        {
            int target_i = target - nums[i];
            if(i > 0 && nums[i] == nums[i-1])
                continue;
            for (int left = i+1; left < nums.length-2; left++)
            {
                mid = left+1;
                right = nums.length-1;
                int tmp = target_i-nums[left];
                if(left > i+1 && nums[left] == nums[left-1])
                    continue;
                while(mid < right)
                {
                    if(nums[mid] + nums[right] == tmp)
                    {
                        int tmp_mid = nums[mid], tmp_right = nums[right];
                        list.add(Arrays.asList(nums[i], nums[left], nums[mid], nums[right]));

                        while(mid < right && nums[++mid] == tmp_mid);
                        while(mid < right && nums[--right] == tmp_right);
                    }
                    else if(nums[mid] + nums[right] < tmp)
                        mid++;
                    else
                        right--;
                }
            }
        }
        return list;
    }
}

```

思路2：排除不可能情况

讨论区rikimberley有个高分的答案，具体思路还是和上面的思路一样的，但是它的运行速度超过了100%的人，是因为它在运行的时候，排除了很多不可能的情况。假设我们考虑4个数分别为A B C D（有序），最大值MAX。

1. A太大，退出：（如果 $4*A > target$ ）
2. A太小，跳过：（ $A+4*MAX < target$ ）

3. 确定A后求BCD的3SUM问题
4. B太大，退出：（如果 $3*B > target$ ）
5. B太小，跳过：（ $B+3*MAX < target$ ）

```

public List<List<Integer>> fourSum(int[] nums, int target) {
    ArrayList<List<Integer>> res = new ArrayList<List<Integer>>();
    int len = nums.length;
    if (nums == null || len < 4)
        return res;

    Arrays.sort(nums);

    int max = nums[len - 1];
    if (4 * nums[0] > target || 4 * max < target)
        return res;

    int i, z;
    for (i = 0; i < len; i++) {
        z = nums[i];
        if (i > 0 && z == nums[i - 1]) // avoid duplicate
            continue;
        if (z + 3 * max < target) // z is too small
            continue;
        if (4 * z > target) // z is too large
            break;
        if (4 * z == target) { // z is the boundary
            if (i + 3 < len && nums[i + 3] == z)
                res.add(Arrays.asList(z, z, z, z));
            break;
        }

        threeSumForFourSum(nums, target - z, i + 1, len - 1, res, z);
    }

    return res;
}

/*
 * Find all possible distinguished three numbers adding up to the target
 * in sorted array nums[] between indices low and high. If there are,
 * add all of them into the ArrayList fourSumList, using
 * fourSumList.add(Arrays.asList(z1, the three numbers))
 */
public void threeSumForFourSum(int[] nums, int target, int low, int high, ArrayList<List<Integer>> fourSumList,
    int z1) {
    if (low + 1 >= high)
        return;

    int max = nums[high];
    if (3 * nums[low] > target || 3 * max < target)
        return;

```

```

    int i, z;
    for (i = low; i < high - 1; i++) {
        z = nums[i];
        if (i > low && z == nums[i - 1]) // avoid duplicate
            continue;
        if (z + 2 * max < target) // z is too small
            continue;

        if (3 * z > target) // z is too large
            break;

        if (3 * z == target) { // z is the boundary
            if (i + 1 < high && nums[i + 2] == z)
                fourSumList.add(Arrays.asList(z1, z, z, z));
            break;
        }

        twoSumForFourSum(nums, target - z, i + 1, high, fourSumList, z1, z);
    }
}

/*
 * Find all possible distinguished two numbers adding up to the target
 * in sorted array nums[] between indices low and high. If there are,
 * add all of them into the ArrayList fourSumList, using
 * fourSumList.add(Arrays.asList(z1, z2, the two numbers))
 */
public void twoSumForFourSum(int[] nums, int target, int low, int high, ArrayList<
List<Integer>> fourSumList,
    int z1, int z2) {

    if (low >= high)
        return;

    if (2 * nums[low] > target || 2 * nums[high] < target)
        return;

    int i = low, j = high, sum, x;
    while (i < j) {
        sum = nums[i] + nums[j];
        if (sum == target) {
            fourSumList.add(Arrays.asList(z1, z2, nums[i], nums[j]));

            x = nums[i];
            while (++i < j && x == nums[i]) // avoid duplicate
                ;
            x = nums[j];
            while (i < --j && x == nums[j]) // avoid duplicate
                ;
        }
        if (sum < target)

```

```
        i++;  
        if (sum > target)  
            j--;  
    }  
    return;  
}
```

011. Container With Most Water

问题

Given n non-negative integers a_1, a_2, \dots, a_n , where each represents a point at coordinate (i, a_i) . n vertical lines are drawn such that the two endpoints of line i is at (i, a_i) and $(i, 0)$. Find two lines, which together with x -axis forms a container, such that the container contains the most water.

Note: You may not slant the container.

思路

首先要明确一个我之前一直理解错的，它的题目是随意找2个木板构成木桶，容量最大，我之前以为是所有的木板已经在它的位置上了，然后找其中容量最大的——你加的水是可以漫过中间比较短的板子的。

如果按题意来，就简单了。

我们用两个指针来限定一个水桶， $left, right$ 。 $h[i]$ 表示 i 木板高度。 Vol_max 表示木桶容量最大值

由于桶的容量由最短的那个木板决定： $Vol = \min(h[left], h[right]) * (right - left)$

- $left$ 和 $right$ 分别指向两端的木板。
- $left$ 和 $right$ 都向中央移动，每次移动 $left$ 和 $Right$ 中间高度较小的（因为反正都是移动一次，宽度肯定缩小1，这时候只能指望高度增加来增加容量，肯定是替换掉高度较小的，才有可能找到更大的容量。）
- 看新桶子的容量是不是大于 Vol_max ，直到 $left$ 和 $right$ 相交。

代码如下：

```
public class Solution {  
    public int maxArea(int[] height) {  
        int l = 0, r = height.length-1;  
        int i = height[l] > height[r] ? r:l;  
        int vol, vol_max = height[i]*(r-l);  
        while(l < r)  
        {  
            if(height[l] < height[r]) l++;  
            else r--;  
            vol = Math.min(height[l],height[r]) * (r - l);  
            if(vol > vol_max) vol_max = vol;  
        }  
        return vol_max;  
    }  
}
```

26. Remove Duplicates from Sorted Array[E]

问题

Given a sorted array, remove the duplicates in place such that each element appear only once and return the new length.

Do not allocate extra space for another array, you must do this in place with constant memory.

For example, Given input array nums = [1,1,2],

Your function should return length = 2, with the first two elements of nums being 1 and 2 respectively. It doesn't matter what you leave beyond the new length.

思路

这题因为是排序好的，所以很容易做。这里关键点是"删除多余元素"这句话，这里其实不是真的删除，因为题目要求不能用额外的空间，所以最好的思路是替换。题目也说了"It doesn't matter what you leave beyond the new length."，所以我们只要把后面的元素放到前面来就行了。

这里用2个指针:

- 1个指针i从1移动到length
- 1个指针newlen记录不重复的个数，只有出现不重复的时候它才能+1

```
public class Solution {  
    public int removeDuplicates(int[] nums) {  
        if(nums.length < 2) return nums.length;  
        int newlen = 1;  
        for(int i = 1; i < nums.length; i++)  
        {
```

```
            if(nums[i] != nums[i-1]) nums[newlen++] = nums[i];  
        }  
        return newlen;  
    }  
}
```

} ``

027. Remove Element[E]

问题

Given an array and a value, remove all instances of that value in place and return the new length.

Do not allocate extra space for another array, you must do this in place with constant memory.

The order of elements can be changed. It doesn't matter what you leave beyond the new length.

Example: Given input array `nums = [3,2,2,3]`, `val = 3`

Your function should return `length = 2`, with the first two elements of `nums` being 2.

思路

和之前的那道题很像，凡是涉及in place交换的都可以考虑用两指针，一次扫描完成。由于这里说了可以交换元素的位置，那么就方便了，我们可以用后面的元素来替换前面的元素，这样就不用交换整个数组。

两个指针`front`和`tail`，分别从前后向中间扫描，当两个指针相遇则结束。

- 如果`front < tail`：
 - 移动`front`，如果遇到了`A[front] == val`的，暂停
 - 移动`tail`，如果遇到了`A[tail] != val`的，把值复制给`A[front]`

```
public class Solution {
    public int removeElement(int[] nums, int val) {
        int front = 0, tail = nums.length-1;
        while(front <= tail)
        {
            if(nums[front] == val && nums[tail] != val)
            {
                nums[front] = nums[tail];
                nums[tail] = val;
            }
            if(nums[front] != val) front++;
            if(nums[tail] == val) tail--;
        }
        return tail+1;
    }
}
```

可以参考 [daxianji007](#) 更加巧妙的代码：这里用一个变量存储不一样的个数，由于题目里只用前index个与val不一样就行，后面的不用考虑，所以每次遇到一个不一样的值，就把它插到前面就好了。

```
int removeElement(int A[], int n, int elem)
{
    int begin=0;
    for(int i=0;i<n;i++)
        if(A[i]!=elem)
            A[begin++]=A[i];
    return begin;
}
```

标准二分的框架

```
while(st < ed)
{
    int mid= st + (ed-st)/2;

    if( r[mid] < target)

        st = mid+1;

    else

        ed = mid;

}

return st;
```

二分有几个关键点需要注意：

1. 二分的关键就是`st`和`ed`两个指针如何移动。需要记住的是，`st`只会往大的方向移动，`ed`只会往小方向移动。
2. `mid= st + (ed-st)/2` 而不用 `mid = (st + ed) / 2` 是因为后面的情况当`st+ed`很大时可能会产生溢出。
3. 注意避免死循环(一般就考虑最后两个数的极端情况):
 - 当 `mid = st + (ed-st)/2` 时，`mid`偏向左边（一般求小于等于`target`的数），所以`st`必须移动 `st = mid+1`，否则就可能会死循环。
 - 当 `mid = st + (ed-st)/2 + 1`，`mid`偏向右边（一般是求大于等于`target`的数），所以`ed`必须移动 `ed = mid -1`，否则可能死循环
1. 一般来说，如果是求恰好找到的，可以多加一个等于的时候退出
2. 有时候，我们不好判断到底判断条件里 `<` 还是 `<=` 的时候，可以把条件限定在2个数A，B（一般来说，此时的`mid = A`）和`target`：
 - `A (mid) < target`，我们需要`st`还是`ed`移动
 - `A (mid) = target`，我们需要`st`还是`ed`移动

二分相关的题目

注意一般二分查找前提是有序，要不是题目已经确保有序了，要不就是自己先做一个排序。

一般有几种：

猜数问题（374题）

这种题目一般是说找到里面特定的数字，这个数字是确定存在的，所以肯定会有一个一定匹配的问题,所以一般我们在判断条件中把 `if(r[mid] == target)` 单独提出来。

找位置（35题）

这类题目一般是说找到一个数它应该在数组中的位置，就是完整的套用框架。

找边界（34题）

这类题目有点麻烦，它需要用两次分别找左右边界。这里就需要考虑里面可能有和目标数相同的数。这时候就要特别注意是使用 `<=` 还是 `<`，而在找右边界的时候，需要移动 `ed`，这时候 `mid` 在原来的基础要+1（看注意3）

问题

Given a sorted array of integers, find the starting and ending position of a given target value. Your algorithm's runtime complexity must be in the order of $O(\log n)$. If the target is not found in the array, return $[-1, -1]$.

For example, Given $[5, 7, 7, 8, 8, 10]$ and target value 8, return $[3, 4]$.

思路

题目里给了提示要求时间上必须是 $O(\log N)$ ，那么暗示了这要用二分去做。范围搜索也就是要确定一个上确界，一个下确界。

分别都可以用二分去找。

左边界

二分的话主要有以下几种情况，我们定义成3个基本规则：

Rule 1. $A[mid] < target$ ，这时候说明还可以向右， $st = mid + 1$ Rule 2. $A[mid] > target$ ，这时候说明应该在左边， $ed = mid - 1$ Rule 3. $A[mid] == target$ ，这时候既可以左移，也可以不移动， $ed = mid$

因为作为二分，最后肯定是在2个数中选一个然后停止循环（ $while(st < ed)$ ）。考虑下面几种情况，假设 $target=5$ ：（在两个数的情况下 $mid = st$ ）

```
case 1: [3 5] ( $A[st] = target > A[ed]$ )
case 2: [5 7] ( $A[st] = target < A[ed]$ )
case 3: [5 5] ( $A[st] = target = A[ed]$ )
case 4: [3 7] ( $A[st] < target < A[ed]$ )
case 5: [3 4] ( $A[st] < A[ed] < target$ )
case 6: [6 7] ( $target < A[st] < A[ed]$ )
```

情况3就是Rule1的情况， $st = mid + 1$ 就行。对于情况2-3这时候，我们只要 $ed = mid$ 就行了，其实也就是把Rule 2 和Rule 3合并成了

Rule 2*. $A[mid] \geq target$ ， $ed = mid$

所以case 1-3 最后停止条件都是 $A[st] = A[ed] = target$

对于case 4-6 都是 $A[st] \neq target$ ，可以直接退出

右边界

找到了左边的边界后就好办了，右边的边界只用同理操作就行了。而且因为找到了左边的边界，实际上就不存在比target小的数了。

这里，我们让mid偏向右边

```
mid = (st + ed)/2 + 1;
```

然后主要控制ed移动就可以了。

代码

```
public class Solution {
    public int[] searchRange(int[] nums, int target) {
        int [] result = new int [2];
        result[0] = result[1] = -1;
        if(nums.length == 0)
            return result;

        int st = 0, ed = nums.length-1, mid;
        while(st < ed) {
            // 寻找左边界
            mid = (st + ed) / 2;
            if (nums[mid] < target) //保证st不会越过任何等于target的数
            {
                st = mid+1; //如果当前的数小于我们需要查的数，继续逼近
            } else { //如果大于等于
                ed = mid;
            }
        }
        if(nums[st] != target)
            return result;
        result[0] = st;
        // 寻找右边界
        ed = nums.length-1;
        while(st < ed)
        {
            mid = (st + ed)/2 + 1; //这里用+1是为了让mid偏向右边
            if(nums[mid] > target)
            {
                ed = mid-1;
            }
            else{ //因为st已经确定了，所以这里实际上不会出现比target小的数了，这里实际就是A[mi
d] == target
                st = mid;
            }
        }
        result[1] = ed;
        return result;
    }
}
```

问题：

We are playing the Guess Game. The game is as follows: I pick a number from 1 to n. You have to guess which number I picked. Every time you guess wrong, I'll tell you whether the number is higher or lower. You call a pre-defined API guess(int num) which returns 3 possible results (-1, 1, or 0): -1 : My number is lower 1 : My number is higher 0 : Congrats! You got it!

Example: n = 10, I pick 6. Return 6.

思路

注意，这里有几个陷阱，一是你要理解 my 的含义，它的 my 是指的 出题人，而我们是解题人，所以如果返回-1，说明它的数小，我们猜的数大，这里一定要理解！

这是一个经典的二分题目，很容易套用二分查找的公式写出来：

```
public class Solution extends GuessGame {
    public int guessNumber(int n) {
        int st = 1, ed = n;
        while(st < ed)
        {
            int mid = (st + ed) / 2;
            if(guess(mid) == 0)
                return mid;
            else if(guess(mid) < 0) // the number is too large
                ed = mid-1;
            else
                st = mid+1;
        }
        return st;
    }
}
```

看上去没问题，但是实际上有潜在的危险，比如这个例子就A不过去：

Status: Time Limit Exceeded	
Submitted: 9 minutes ago	
Last executed input:	2126753390 1702766719

为什么，因为 `int mid = (st + ed) / 2;` 可能会由于 (st+ed) 产生溢出！！

所以，为了解决这个问题，推荐把mid的写法写成下面的格式：

```
int mid = st + (ed - st) / 2;
```

然后再A就能通过

问题

Suppose a sorted array is rotated at some pivot unknown to you beforehand.

(i.e., `0 1 2 4 5 6 7` might become `4 5 6 7 0 1 2`).

You are given a target value to search. If found in the array return its index, otherwise return -1.

You may assume no duplicate exists in the array.

思路

这个问题是二分查找的变种，它这里唯一的一个小trick就是把本来有序的数组做了一个旋转。但是注意，旋转的2部分是分别有序的。

还有一个很重要的特点：后半部分小于前半部分，所以其实很容易判断出mid在哪一部分。

（如果`num[mid] > num[0]`，我们就知道mid在左部分，否则在右部分。）

这里不能直接用二分的问题在于：如果target和mid不在同一部，就会出错。

举例`nums: [4,5,6,1,2,3,4]`，`st = 0`，`ed = 6`，`mid = 3`，`target = 6`

此时：mid指向1，而target是6，说明mid和target不在同一部，会出现什么问题呢？

按照算法：

```
if (nums[mid] < target)
    st = mid+1
```

这里，明明应该往左搜索，但是却往右边搜索了。所以，为了解决这个问题。我们可以引入 `inf` 和 `-inf`

引入inf和-inf

为了更加方便理解，我们举例：

我们随便看一个数组`num:[4,5,6,1,2,3,4]`

- 假设我们查询的目标是3：那我们实际查询的是 右一段：`[4,5,6,1,2,3,4]`，实际上我们可以把前面的数都看作 `-inf`：`[-inf,-inf,-inf,1,2,3,4]`

- 假设我们查询的目标是5：那我们实际查询的是 左一段 ：[4,5,6,7,1,2,3,4]，实际上我们可以把后面的数都看作 `inf` ：[4,5,6,inf,inf,inf,inf]

这样其实就和普通的二分一样了。

如何知道在哪一段？

1. 如果 `target < nums[0]` 和 `nums[mid] < nums[0]` 同时成立，那么我们按照普通的二分查找操作就行。
2. 如果 `target < nums[0]` 和 `nums[mid] < nums[0]` 不同时成立，那么说明`target`和`mid`在不同部分。我们需要根据`target`的情况引入 `inf` 和 `-inf`
 - 如果 `target < nums[0]`，说明在右部，我们引入 `-inf`，让`st`移动
 - 否则，说明在左部，我们引入 `inf`，让`ed`移动

代码

```
public class Solution {
    public int search(int[] nums, int target) {
        int st, ed, mid;
        st = 0;
        ed = nums.length-1;
        while(st < ed)
        {
            mid = st + (ed - st)/2;
            //增加部分
            int tmp = (nums[mid] < nums[0]) == (target < nums[0])
                ? nums[mid] //如果在同一部，则用原值
                : target < nums[0] ? Integer.MIN_VALUE : Integer.MAX_VALUE; //否则根据情况引入-inf或inf

            if(tmp < target)
                st = mid+1;
            else if(tmp > target)
                ed = mid;
            else
                return mid;
        }
        return nums[st] == target?st:-1;
    }
}
```

003. Longest Substring Without Repeating Characters[M]

题目：

Given a string, find the length of the longest substring without repeating characters. For example, the longest substring without repeating letters for "abcabcbb" is "abc", which the length is 3. For "bbbbbb" the longest substring is "b", with the length of 1.

分析

题目意思是在一个字符串中找一个最长的子串（没有重复的字母） 在简单的思路：从左往右扩展子串，维持2个变量*i*和*j*来维持一个新的子串，*j*不断移动，每加入一个新的字符，判断是否有重复的，如果有重复的，移动*i*，生成新子串.....

```
len = max(len, j-i+1);
```

abcad**f**

↑ ↑
i j

$max=3$

abcad**f**

↑ ↑
i j

$max=3$

abcad**f**

↑ ↑
i j

$max=3$

abcad**f**

↑ ↑
i j

$max=4$

abcad**f**

↑ ↑
i j

$max=5$

这里有个问题就是，如何判断子串中是否有重复的字符，传统思路就是循环，这样每次查找重复的时间复杂度为 $O(n)$ ，导致整体时间复杂度为 $O(N^2)$ ，其实我们可以使用hashmap来存，这样可以保证每次查找的效率为 $O(1)$ 。

代码

```
class Solution {
public:
    int lengthOfLongestSubstring(string s) {
        unordered_map<char, int> mymap;
        unordered_map<char, int>::iterator it;
        int len = 0, i = -1;
        for(int j=0; j < s.length(); j++)
        {
            /**是否有重复***/
            it = mymap.find(s.at(j));
            if(it != mymap.end())
                /**有重复的时候，移动i***/
                i = std::max(it->second, i);
            /**把新的字符加入***/
            mymap[s.at(j)] = j;
            len = std::max(len, (j-i));
        }
        return len;
    }
};
```

但是实际上，对于这个题目，不需要用hashmap，因为所有的字符ASCII码加起来也就最多255个，可以直接用数组来代替hashmap，效率更高。

```
class Solution {
public:
    int lengthOfLongestSubstring(string s) {
        vector<int> mymap(255, -1);
        int len = 0, i = -1, tmp;
        for(int j=0; j < s.length(); j++)
        {
            tmp = mymap[s.at(j)];
            i = std::max(tmp, i);
            mymap[s.at(j)] = j;
            len = std::max(len, (j-i));
        }
        return len;
    }
};
```

上面用了一个trick就是每个数组的初始化为-1表示没有出现重复，它不可能比i的初始值大，如果有重复的，直接覆盖，这样可以不用额外的语句判断是否出现重复。

005.Longest Palindromic [M]

题目

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.

思路

可以用的算法有改良KMP还有manacher（马拉车）算法，毫无疑问，manacher算法是专门用来解决最长子串问题的，也是最简便的。关于这个算法可以看: [Manacher算法](#)


```

class Solution {
public:
    string longestPalindrome(string s) {
        //manacher
        int bound = 0;
        int id,max_pos=0;
        int new_len = 2*s.length()+2;
        vector<int> P(new_len,1);
        string new_str(new_len-1,'#');
        //生成新串，把所有的字符串通过'#'扩展成奇数
        for(int i = 0;i < s.length();i++)
        {
            new_str[2*i+1] = s[i];
        }
        new_str = '$'+new_str +='\0'; //防止越界
        //manacher算法
        for(int i=1;i < new_len; i++)
        {
            if(i < bound)
            {
                P[i] = min(bound-i,P[2*id-i]); //如果在范围内，找对称面的P[id-(i-id)]
和max_pos-i的最小值
            }
            while(new_str[i-P[i]] == new_str[i+P[i]])//查找以这个字符为中心的回文串
            {
                P[i]++;
            }
            //更新id和bound
            if(i+P[i] > bound)
            {
                bound = i+P[i];
                id = i;
            }

            max_pos = P[i] > P[max_pos]?i:max_pos;
        }
        int len = P[max_pos]-1;
        int start = (max_pos-P[max_pos])/2;
        return s.substr(start,len);
    }
};

```

2.Add Two Numbers [M]

题目：

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

思路：

题目的大意是两个链表求和，和也是一个链表（链表是逆序存的数字）这和大数的加减很像，不过这个进位是从后向前。所以可以同时从前往后加，然后保留进位。

其实做这种大数加减的题目不管是用的大数组还是链表，都是一样的：

- 首先做个大循环，对每一位进行操作：
 - 当前位： $(A[i]+B[i])\%10$
 - 进位： $(A[i]+B[i])/10$

这里要注意一点，就是可能两个数组不是同样的长度，需要考虑一个数组已经到头，另一个数组还没结束的情况

代码

我的这个代码写的不够好，考虑的过于复杂，可以直接看下面更精巧的代码。

```
/**
 * Definition for singly-linked list->
 * struct ListNode {
 *     int val;
 *     ListNode *next;
 *     ListNode(int x) : val(x), next(NULL) {}
 * };
 */
class Solution {
public:
    ListNode* addTwoNumbers(ListNode* l1, ListNode* l2) {
        //异常判断
        if(l1 == NULL && l2 == NULL)
            return NULL;
        if(l1 == NULL)
            return l2;
        if(l2 == NULL)
            return l1;
        //初始化
        ListNode * result = new ListNode((l1->val + l2->val)%10);
        int carry = (l1->val + l2->val)/10; //表示进位
        ListNode* templ1 = l1;
        ListNode* templ2 = l2;
        ListNode* tempresult = result;
        //这里弄一个结束节点，有利于当一个节点到结尾后做操作
        ListNode* EndNode = new ListNode(0);
        while(templ1->next != NULL || templ2->next != NULL)
        {
            //如果某个链表已经到结尾了，那么就把它变成EndNode,特点是值为0,next = NULL
            if(templ1->next == NULL)
                templ1 = EndNode;
            else
                templ1 = templ1->next;
            if(templ2->next == NULL)
                templ2 = EndNode;
            else
                templ2 = templ2->next;
            tempresult->next = new ListNode((templ1->val + templ2->val + carry)%10);

            carry = (templ1->val + templ2->val + carry)/10;

            tempresult = tempresult->next;
        }
        if(carry == 1)
            tempresult->next = new ListNode(1);
        return result;
    }
};
```

更精巧的代码

不得不说，potpie的这个代码比我上面的代码精简了不少，我上面的代码考虑的太多了，因为我通式用的`templ1->val + templ2->val + carry`，导致每次需要对先结束的链表进行尾部填充，而且开头多了很多额外的代码。

他的代码，2个链表是独立操作的，而且代码写的很精简，基本没废话！赞

```
public class Solution {
    public ListNode addTwoNumbers(ListNode l1, ListNode l2) {
        ListNode c1 = l1;
        ListNode c2 = l2;
        ListNode sentinel = new ListNode(0);
        ListNode d = sentinel;
        int sum = 0;
        while (c1 != null || c2 != null) {
            sum /= 10;
            if (c1 != null) {
                sum += c1.val;
                c1 = c1.next;
            }
            if (c2 != null) {
                sum += c2.val;
                c2 = c2.next;
            }
            d.next = new ListNode(sum % 10);
            d = d.next;
        }
        if (sum / 10 == 1)
            d.next = new ListNode(1);
        return sentinel.next;
    }
}
```

19. Remove Nth Node From End of List

问题

Given a linked list, remove the n th node from the end of list and return its head.

For example,

Given linked list: 1->2->3->4->5, and $n = 2$.

After removing the second node from the end, the linked list becomes 1->2->3->5.

思路

这是链表中非常常见的问题，众所周知，链表慢就慢在遍历查找，而对于单链表来说，每次必须从头开始搜索，这样使得链表在处理“倒数”这个概念的时候，特别无力。常规的做法必须要2遍遍历：1遍计算链表长度 len ，1遍搜索倒数的元素 $len-n$ 。(当然，你可以通过加入链表长度变量或者使用双向链表解决这个问题。)

但是题目要求是一遍完成，有没有一遍的思路呢？其实是有的，那就是双指针或递归。

思路一 —— 双指针

这里有个常用的做法去解决“倒数问题”：双指针。

双指针常用做法是：

- 一个指针用来作为参考，控制长度（作为循环停止条件）
- 一个指针延迟启动用来跑“倒数”。

第一个指针先运行 n 个数，然后打开第二个指针，这样，当第一个指针跑完时，第二个指针刚好跑过 $Len-N$ 数，这样就找到了倒数第 n 个数。

注意：由于删除操作比较特殊，必须找到前一个节点才能删除下个节点，所以一般删除操作我们会构造一个虚拟节点作为开头，以防开头节点被删除。

```
public class Solution {
    public ListNode removeNthFromEnd(ListNode head, int n) {
        ListNode newhead = new ListNode(-1); //防止头被删除
        newhead.next = head;
        ListNode point1 = newhead;
        ListNode point2 = newhead;
        for(;point1 != null;point1 = point1.next,n--) //point1 控制长度
        {
            if(n < 0)
                point2 = point2.next; //point2延迟启动
        }
        point2.next = point2.next.next;
        return newhead.next;
    }
}
```

思路二 —— 递归

除了用双指针外，还可以考虑用递归，凡是这种涉及单链表插入删除操作的时候，都可以考虑用递归，因为插入和删除都需要涉及它的父亲操作。我们考虑最后一个元素是第一层，然后逐级返回，当返回到第N+1层（也就是父亲节点所在层数）就开始删除操作。

```
public class Solution {
    public ListNode removeNthFromEnd(ListNode head, int n) {
        ListNode newhead = new ListNode(-1);
        newhead.next = head;
        remove(newhead,n);
        return newhead.next;
    }

    private int remove(ListNode node, int n) {
        if(node.next == null) return 1;
        int level = remove(node.next,n)+1; //层数+1
        if(level == n+1) //找到了父亲
            node.next = node.next.next;
        return level;
    }
}
```

21. Merge Two Sorted Lists

问题

Merge two sorted linked lists and return it as a new list.

思路

这个题目很简单也有几个可以考虑的思路，一个是比较直接的方式，重新构造链表，一种是利用递归

思路1：用新的链表

这里用了一个新的节点保存结果的链表，这里为了方便链表的扩充，增加一个临时的节点变量（否则每次加入都要遍历到尾部）

```
public class Solution {
    public ListNode mergeTwoLists(ListNode l1, ListNode l2) {
        ListNode result = new ListNode(0);
        ListNode tmp = result;
        while(l1 != null || l2 != null)
        {
            if(l2 == null || (l1 != null && l1.val <= l2.val))
            {
                tmp.next = l1;
                l1 = l1.next;
            }
            else
            {
                tmp.next = l2;
                l2 = l2.next;
            }
            tmp = tmp.next;
        }
        return result.next;
    }
}
```

思路2：递归方案

在很多时候，递归的方案可以提供一种清晰简单的解决方案，代码也更精炼。但是递归有个问题就是容易出现堆栈溢出的问题。

```
public:
    ListNode *mergeTwoLists(ListNode *l1, ListNode *l2) {
        if(l1 == NULL) return l2;
        if(l2 == NULL) return l1;

        if(l1->val < l2->val) {
            l1->next = mergeTwoLists(l1->next, l2);
            return l1;
        } else {
            l2->next = mergeTwoLists(l2->next, l1);
            return l2;
        }
    }
};
```


023. Merge k Sorted Lists [H]

问题

Merge k sorted linked lists and return it as one sorted list. Analyze and describe its complexity.

Subscribe to see which companies asked this question

思路

这题明显就是Merge two sorted list的升级版，我们知道，那题我们可以用2个指针完成，但是这题，难道要用K个指针？

思路1：每次循环K下，找到每个链表的头，然后把最小的插入新链表中，把这个链表头弃掉，然后重复这个操作。这个最多需要循环KKn次

思路2：二路归并，每次合并两个链表最多合并 $\log_2 N$ 次

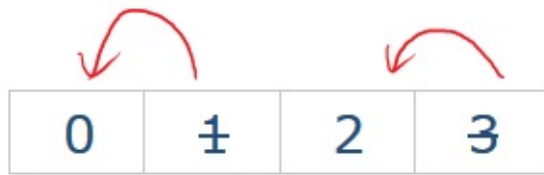
流程：

- 先每隔2个合并，结果放到合并的第一个链表处
- 再每隔4个合并，结果放到合并的第一个链表处
- 每隔K/2个合并，结果放到合并的第一个链表处
- 返回第一个链表

```
//控制每隔多少
for(int step = 1; step < lists.length; step*=2)
//遍历所有可以合并的选项
    for(int i = 0; i < lists.length; i+=step*2)
```

假设K=4：（偶数相对好处理）

- 先合并L[0],L[1]放到L[0]
- 然后合并L[2],L[3]放到L[2]
- 最后合并L[0],L[2]放到L[0]
- 返回L[0]就好了

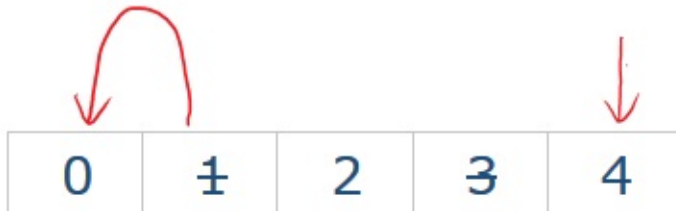


step=1

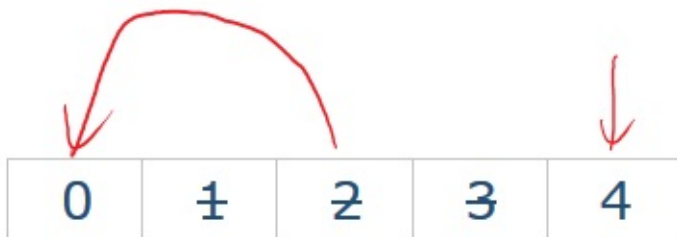


step=2

假设K=5：(麻烦点，当运行到L[4]的时候它没有下个可以和它合并的元素了，所以应该跳出循环)



step=1



step=2



step=4

```
//控制每隔多少
for(int step = 1; step < lists.length; step*=2)
//遍历所有可以合并的选项
for(int i = 0; i < lists.length; i+=step*2)
{
    //如果某个元素没有下一个元素就退出循环
    if(i+step >= lists.length) break;
    lists[i] = mergeTwoLists(lists[i], lists[i+step]);
}
```

所以代码涉及2层循环，外面是控制每隔多少合并一次，里面是遍历所有可以合并的链表，进行合并操作。

```
public ListNode mergeKLists(ListNode[] lists) {
    if(lists.length == 0)
        return null;
    //控制每隔多少
    for(int step = 1; step < lists.length; step*=2)
        //遍历所有可以合并的选项
        for(int i = 0; i < lists.length; i+=step*2)
        {
            //如果某个元素没有下一个元素就退出循环
            if(i+step >= lists.length) break;
            lists[i] = mergeTwoLists(lists[i], lists[i+step]);
        }
    return lists[0];
}
//完全是之前的代码
public ListNode mergeTwoLists(ListNode l1, ListNode l2) {
    ListNode result = new ListNode(0);
    ListNode tmp = result;
    while(l1 != null || l2 != null)
    {
        if(l2 == null || (l1 != null && l1.val <= l2.val))
        {
            tmp.next = l1;
            l1 = l1.next;
        }
        else
        {
            tmp.next = l2;
            l2 = l2.next;
        }
        tmp = tmp.next;
    }
    return result.next;
}
```


024. Swap Nodes in Pairs[E]

题目

Given a linked list, swap every two adjacent nodes and return its head.

For example, Given 1->2->3->4, you should return the list as 2->1->4->3.

Your algorithm should use only constant space. You may not modify the values in the list, only nodes itself can be changed.

思路

这个题目的意思是每2个元素要交换一下，我们已经做了很多有关链表的问题，知道凡是涉及链表的操作，比较麻烦的地方都在与链表的操作都必须涉及到它的父亲。

这题也不例外，需要考虑很多问题：

1. 如果链表头要交换怎么办
2. 如何方便的交换两个节点
3. 如果长度不为偶数怎么办

问题1我们已经说过很多遍了，直接用一个新节点去解决这个问题。

```
ListNode newhead = new ListNode(0);
```

问题2，这里得知道，如果涉及到链表交换操作，那么它至少涉及3个节点：当前节点，当前节点的父亲，当前节点的孩子这里由于我们循环是从前往后走的，所以我们这里使用：当前节点，当前节点的孩子，当前节点的孙子，可能更方便。

```
ListNode first = current.next;  
ListNode second = current.next.next;
```

交换操作就很简单了

```
first.next = second.next;  
current.next = second;  
current.next.next = first;  
current = current.next.next;
```

问题3，我们得需要加入判断，当前节点的孩子和孙子都要有才能继续

```
while (current.next != null && current.next.next != null)
```

所以这样一分析，代码就很容易写出来了

代码

```
public class Solution {  
    public ListNode swapPairs(ListNode head) {  
        if(head == null)  
            return null;  
        ListNode newhead = new ListNode(0);  
        newhead.next = head;  
        ListNode current = newhead;  
        while (current.next != null && current.next.next != null) {  
            ListNode first = current.next;  
            ListNode second = current.next.next;  
            first.next = second.next;  
            current.next = second;  
            current.next.next = first;  
            current = current.next.next;  
        }  
        return newhead.next;  
    }  
}
```

017. Letter Combinations of a Phone Number[M]

问题

Given a digit string, return all possible letter combinations that the number could represent.

A mapping of digit to letters (just like on the telephone buttons) is given below.



Input: Digit string "23" Output: ["ad", "ae", "af", "bd", "be", "bf", "cd", "ce", "cf"].

分析

这题其实含义是枚举所有可能的组合方式。如果题目要求是枚举“23”的所有可能字母组合，我们很好做，2重循环对吧？但是现在难就难在，一开始你不知道输入是什么，你没办法确定组合长度，组合个数，也没办法确定循环层数，这时候怎么办？？

这里有两个思路，也是很常用的思路：递归，队列。

思路1：递归

递归一般是解决一些整体不好求的问题。它通过把大问题划小，然后找到一种特定的规律，然后求解。

递归的思路我们很好理解，我们没办法确定整体，我可以先从入手。

假定有个数字串“23456”

- 假定除了数字'2'的组合已经求出来了，准备求'3'，那我只要把'3'所代表的'def'加到之前字符串他们每一个的后面就好。
- 假定除了数字'2'的组合已经求出来了，准备求'4'，那我只要把'4'所代表的'hij'加到之前字符串他们每一个的后面就好。
- 一直这样推下去，直到发现'6'后面是空的了，那将当前这个字符串加入列表就好了。

```
public class Solution {
    private char[][] dict = { {}, { 'a', 'b', 'c' }, { 'd', 'e', 'f' }, { 'g', 'h', 'i' }, { 'j', 'k', 'l' }, { 'm', 'n', 'o' },
        { 'p', 'q', 'r', 's' }, { 't', 'u', 'v' }, { 'w', 'x', 'y', 'z' } };
    public List<String> letterCombinations(String digits) {
        LinkedList<String> result = new LinkedList<String>();
        if(digits.length() == 0)
            return result;
        addNumber(result, 0, "", digits);
        return result;
    }
    public void addNumber(List<String> list, int i, String curstr, String digits)
    {
        if (i == digits.length()) {
            list.add(curstr);
            return;
        }
        char[] candidates = dict[digits.charAt(i) - '1'];
        for (char c : candidates)
            addNumber(list, i+1, curstr + c, digits);
    }
}
```

思路2：用队列

队列的思路也不算太难理解。如果递归算纵向求解的话，队列就是横向求解。每加入一个新的数字的时候，就把当前队列的元素全都扩充一遍。使得队列不仅在长度上，也在宽度上增加了。这就像一个装配流水线。半成品每流过一个工人，工人就把之前的产品拿出来，往上安装一个零件，然后放到传送带上，让它继续传到下个工人那。

1. 一共需要的工人数，就是数字串长度，它决定了产品需要经过几道加工

```
for(int i = 0; i < digits.length(); i++)
{
}
```

2. 然后我们看目前有多少个不同的半成品需要加工


```
int pos = digits.charAt(i) - '0';
int size = result.size();
for(int j = 0;j < size;j++)
{
}
```

3. 然后就开始加工了，我们获取每个数字对应的字符串长度，这就是工人需要加工的零件个数。这里加工是把每个半成品拿出来，复制多份，然后按上新的零件

```
String tmp = result.remove();
for(int k = 0;k < map[pos].length();k++)
    result.add(tmp+map[pos].charAt(k));
```

整体代码

```
public class Solution {
    public List<String> letterCombinations(String digits) {
        LinkedList<String> result = new LinkedList<String>();
        if(digits.length() == 0)
            return result;
        String[] map = new String[] { "0", "1", "abc", "def", "ghi", "jkl", "mno", "pqrs", "tuv", "wxyz" };
        result.add("");
        for(int i = 0;i < digits.length(); i++)
        {
            int pos = digits.charAt(i) - '0';
            String s = map[pos];
            int size = result.size();
            for(int j = 0;j < size;j++)
            {
                String tmp = result.poll();
                for(int k = 0;k < s.length();k++)
                    result.add(tmp+s.charAt(k));
            }
        }
        return result;
    }
}
```

这里，我用了size变量来存之前加工好的半成品个数（因为队列会在加工后扩充，size会变化），

```
int size = result.size();
for(int j = 0;j < size;j++)
```

但是，高分答案中有一个思路，我觉得很赞。（要是想不到这个，就用我上面的写就好了，多一行代码而已）

```
while(ans.peek().length()==i)
```

这里`ans.peek().length()`是取出第一个元素的长度，当长度等于`i`的时候，说明是当前需要加工的半成品，而加工完后，队列中的每个元素长度都会增加1，所以，这时候循环就会停止。

007. Reverse Integer[E]——处理溢出的技巧

题目

Reverse digits of an integer.

Example1: x = 123, return 321 Example2: x = -123, return -321

思路

这题完全没丝毫的难度，任何人几分钟都可以写出来，但是，这题修改后，加入了一个新的测试，瞬间大家讨论的就多了，就是——溢出测试

因为整数只有32位，可能原数不会溢出，但是转置后就不一定了，所以必须要考虑溢出的情况。

思路1——用long

一个比较讨巧的方案，直接用long不会溢出再和INT_MAX比较就好了

```
class Solution {
public:
    int reverse(int x) {
        long tmp=0;
        while(x != 0)
        {
            tmp *=10;
            tmp += x%10;
            if(tmp > INT_MAX || tmp < INT_MIN)
                return 0;
            x /= 10;
        }
        return tmp;
    }
};
```

思路2——变化前后对比

bitzhuwei的代码。不用任何flag和INT_MAX宏或者任何硬编码0x7fffffff 这个思路也是很容易理解的，做一些操作，如果溢出了，那溢出后的值做反向操作会和之前的值不一样。

这里就用一个变量存储变化后的值，每次做反向操作，如果和之前的值一样就更新，不一样，说明溢出了。``c++ public int reverse(int x) { int result = 0;

```
while (x != 0)
{
    int tail = x % 10;
    int newResult = result * 10 + tail;
    if ((newResult - tail) / 10 != result)
    { return 0; }
    result = newResult;
    x = x / 10;
}

return result;
```

```
}
```

#####思路3——提前停止操作**

如果当前的数已经 $>INT_MAX/10$ 那么再做一次操作，必然溢出。

```c++

class Solution

{

public:

int reverse(int n)

{

int result = 0;

while (n != 0)

{

if (result > INT\_MAX / 10

|| ((result == INT\_MAX / 10) && (n % 10 > INT\_MAX % 10)))

{

result = 0;

break;

}

if (result < INT\_MIN / 10

|| ((result == INT\_MIN / 10) && (n % 10 < INT\_MIN % 10)))

{

result = 0;

break;

}

result = result \* 10 + n % 10;

n = n / 10;

}

return result;

}

};

## 008. String to Integer (atoi) [E]

[TOC]

### 题目

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.

### 思路

这题也比较好做，关键是要考虑挺多东西，我也是提交了好多次才发现有这么多要考虑的地方。

- 开头的空格
- 正负符号的处理
- 溢出处理
- 非法输入

开头空格处理：

```
while(str[i] == " ") i++;
```

正负号的处理：我觉得yuruofeifei这个解决方案简直赞

```
if (str[i] == '-' || str[i] == '+') {
 sign = 1 - 2 * (str[i++] == '-');
}
.....
return base * sign;
```

溢出处理（可以参考上一道题）：

```

if (base > INT_MAX / 10 || (base == INT_MAX / 10 && str[i] - '0' > INT_MAX%10)) {
 if (sign == 1) return INT_MAX;
 else return INT_MIN;
}

```

非法输入：其实只用过滤就行了

```

while (str[i] >= '0' && str[i] <= '9') {

}

```

## 代码

我的代码，不够简洁，可以参考yuruofeifei的代码，在下面

```

class Solution {
public:
 int myAtoi(string str) {
 long tmp=0;
 bool neg;
 int i = 0;
 while(str[i] == ' ') i++; //读掉空格
 neg = str[i] == '-'?1:0;
 for(i = i+ (neg || str[i] == '+'); i < str.length(); i++) //如果是- 或 + i+1跳过
 符号
 {
 if(str[i] - '0' >= 0 && str[i] - '0' < 10) //过滤非法输入
 {
 tmp *= 10;
 tmp += (str[i] - '0');
 if(tmp >= INT_MAX && !neg) //溢出判断
 {
 tmp = INT_MAX;
 break;
 }
 if(tmp - 1 >= INT_MAX && neg) //除了符号，INT_MAX和INT_MIN只差1
 {
 tmp = INT_MIN;
 break;
 }
 }
 else break;
 }
 if(neg) return -tmp;
 return tmp;
 }
};

```

yuruofeifei的代码

```
int myAtoi(string str) {
 int sign = 1, base = 0, i = 0;
 while (str[i] == ' ') { i++; }
 if (str[i] == '-' || str[i] == '+') {
 sign = 1 - 2 * (str[i++] == '-');
 }
 while (str[i] >= '0' && str[i] <= '9') {
 if (base > INT_MAX / 10 || (base == INT_MAX / 10 && str[i] - '0' > 7)) {
 if (sign == 1) return INT_MAX;
 else return INT_MIN;
 }
 base = 10 * base + (str[i++] - '0');
 }
 return base * sign;
}
```



## 009. Palindrome Number[E]

问题：

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

思路

这里说不用额外的空间意思是不用 $O(n)$ 的空间， $O(1)$ 的还是可以用的，不然循环都不好写。。

思路1

简单的思路 就是把数字逆转，然后判断逆转后的数字跟原来数字是不是一样的。

```
class Solution {
public:
 bool isPalindrome(int x) {
 if(x < 0) return false;
 int r=0,t;
 t = x;
 while(t != 0)
 {
 r = r*10 + t%10;
 t /=10;
 }
 return r == x;
 }
};
```

思路2

但是其实，不用把数字逆转完再判断，因为如果是回文数字，那么只要逆转一半看是否满足回文条件就行了。

设新数为 $r$ ，原来数为 $x$ ，每次：

```
x = x/10, r = r*10 + x%10;
```

- 如何到一半停止？
  - 如果 $x \leq r$ 时可以停止了（至少到了一半）
  - 如果是偶数长度，并且是回文，那么刚好可以到 $x == r$
  - 如果是奇数长度，并且是回文，那么 $x < r$ ，这时候 $r$ 刚好比 $x$ 多一位数
- 停止后判断是否是回文
  - 如果是偶数长度，很简单判断 $r == x$
  - 如果是奇数长度，要判断 $r/10 == x$

注意：这里有一些小问题。

- 当尾数为0的情况。尾数为0会导致 $r$ 的增长少1位数（因为 $0*10 = 0$ ）。比如110不是回文，最后停止 $r = 1, x = 1$ 但是 $r == x$
- 当数字小于0的时候，也是不满足回文的条件的。

```
class Solution {
public:
 bool isPalindrome(int x) {
 if(x < 0 || (x != 0 && x % 10 == 0)) return false;
 int r = 0;
 while(x > r)
 {
 r = r*10 + x%10;
 x /= 10;
 }
 return (r == x) || (r/10 == x);
 }
};
```

## 029. Divide Two Integers[M]

### 问题

Divide two integers without using multiplication, division and mod operator.

If it is overflow, return MAX\_INT.

### 思路

这道题难在不能使用乘除取余操作，所以我们只能手动的实现除法，我们来看看如何实现。

我们假设被除数为 `D`，除数为 `d`

### 手动实现除法

首先，本能想到  $10 = 2 * 5$ ， $10/2 = 5$ ，说明10中有5个2，这里可以用循环来做，看 `D` 中间有多少个 `d` 就行了。

本能写出一个循环应该不难。

```
for(i = 0; dividend - divisor >= 0 ; i++)
{
 dividend = dividend - divisor;
}
```

### 符号

现在有另一个问题，正负数怎么办？我们知道，除法里，异号为负，同号为正，既然不能使用乘除判断，那我们就只能写个判断，这个flag到时候就充当标志。

```
boolean flag = (dividend > 0 && divisor < 0 || dividend < 0 && divisor > 0);
```

当然还有更好的实现方式，既然不能用乘除，可以用位运算呀，这和异或操作的含义刚好一样。

```
boolean sign = ((dividend < 0) ^ (divisor < 0));
```

然后，把 `D` 和 `d` 同时取绝对值就好了。

```
long did = Math.abs((long)dividend);
long dis = Math.abs((long)divisor);
```

注意：这里除数一定要用`long`，因为如果最小值取绝对值会溢出

## 溢出

我们知道，除法可能产生溢出的情况就是 `D` 为0，或者 `D` 为 最小值 ， `d` 为-1

```
if (!divisor || (dividend == Integer.MIN_VALUE && divisor == -1))
 return Integer.MAX_VALUE;
```

## 更快的方案

好了，我们可以测试下代码。发现超时，想想也是，上面的循环太慢了。其实我们稍微想一下就可以提速，我们可以采用逐渐逼近的思路：

- 我们先看 `i = N` 个 `d` 可不可以
- 如果可以，我们看看 `i = 2N` 个 `d` 可不可以
  - 如果还可以就继续看 `i = 4N` 个 `d` 可不可以
    - 直到不可以减，我们让 `D` 减去 `i` 个 `d`

这里为什么使用2的倍数，因为可以用位运算呀~~

```
d << 1 就相当与d*2
d << 2 就相当与d*4
```

我们再用一个`i`来就记数 `i = 0` 开始

```
1 << 0 就相当与1
1 << 1 就相当与2
1 << 2 就相当与4
```

这里我们用一个临时变量 `mul_d` 存 `d` 的左移操作，注意：`mul_d` 必须为`long`，因为左移操作很可能溢出！！

现在我们可以写出以下代码了。

```
public class Solution {
 public int divide(int dividend, int divisor) {
 if(divisor == 0 || (dividend == Integer.MIN_VALUE && divisor == -1))
 return Integer.MAX_VALUE;
 int i, total = 0;
 //判断正负号
 boolean sign = ((dividend < 0) ^ (divisor < 0));
 //这里必须要long, 因为如果最小值取绝对值会溢出
 long did = Math.abs((long)dividend);
 long dis = Math.abs((long)divisor);
 while(did >= dis)
 {
 long mul_dis = dis;
 i = 0;
 //每次左移乘2, 记录下来, 直到不能减
 while(did >= (mul_dis<<1))
 {
 i++;
 mul_dis <= 1;
 }
 did -= mul_dis;
 total += 1<<i;
 }
 //根据符号返回
 return sign?-total:total;
 }
}
```

## 006.ZigZag Conversion[E]

### 题目

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)



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".

### 思路1——用字符串数组

我能说我一开始完全没看懂吗？我是根据**Custom Testcase**自己慢慢测试摸索出来的。其实，应该是这样的 2行：A C E B D F

3行：A E I B D F H J C G \_ K

所以有个简单的思路：

- 每行弄个string。
- 对原始字符串进行扫描，从上往下，从下往上，依次加入每行的string
- 最后把所有的string拼接起来

```

class Solution {
public:
 string convert(string s, int numRows) {
 string str[numRows], tmp;
 if(numRows == 1)
 return s;
 int flag;
 for(int i = 0, j = 0; i < s.length(); i++)
 {
 if(j == 0)
 flag = 1;
 if(j == numRows-1)
 flag = -1;
 str[j] += s[i];
 j += flag;
 }
 for(int i = 0; i < numRows; i++){
 tmp += str[i];
 }
 return tmp;
 }
};

```

## 思路2——观察规律

2行：A C E B D F

3行：

A E I B D F H J C G \_ K

4行：

A G B F H C E I K D \_ J

观察规律后，以每行的元素作为轴，可以发现，下面的字母都是对称排列的 换成对应的index后，规律更明显

0 4 8 1 3 5 7 9 2 6 \_ 10

第2层的元素就是以第一行的元素为轴，+1,-1 第三层的元素就是以第一行的元素为轴，+2,-2 ..... 但是最后一层的元素，由于其特殊性，我们可以只考虑+k

Ps.所有过界的元素都不考虑

轴也有规律：除了首尾两层，其他都是2个，所以第一层每隔 $2n-2$ 出现一次。

```
class Solution {
public:
 string convert(string s, int numRows)
 {
 string tmp;
 if(numRows == 1)
 return s;
 int inc = 2*numRows-2; //每次轴增加的步长
 int len = s.length();
 for(int i = 0; i < numRows;i++)
 {
 for(int j = 0;j < s.length()+numRows; j += inc)
 {
 if(j - i > 0 && j - i < s.length()
 && i != 0 && i != numRows -1) //首，尾只考虑+不考虑-
 tmp+= s[j-i];
 if(j + i < s.length())
 tmp += s[j+i];
 }
 }
 return tmp;
 }
};
```



## 292. Nim Game[E]

### 题目

You are playing the following Nim Game with your friend: There is a heap of stones on the table, each time one of you take turns to remove 1 to 3 stones. The one who removes the last stone will be the winner. You will take the first turn to remove the stones.

Both of you are very clever and have optimal strategies for the game. Write a function to determine whether you can win the game given the number of stones in the heap.

For example, if there are 4 stones in the heap, then you will never win the game: no matter 1, 2, or 3 stones you remove, the last stone will always be removed by your friend.

### 分析

1. 只要能被4整除，你就输了。因为不论你怎么拿，对方都会凑到4。
2. 如果不能被4整除，那你第一次把余数拿了，然后你模仿上面的策略，你就赢了。

### 代码

```
return bool(n%4)
```

## 012. Integer to Roman[M]

### 问题

Given an integer, convert it to a roman numeral.

Input is guaranteed to be within the range from 1 to 3999.

### 思路

分析罗马数字的规律：

| Symbol | Value |
|--------|-------|
| I      | 1     |
| V      | 5     |
| X      | 10    |
| L      | 50    |
| C      | 100   |
| D      | 500   |
| M      | 1,000 |

上面是罗马数字所有的符号。罗马数字的规则：一般情况下，从左到右从大到小排，字母代表的数字累加。比如：

XII = 12

MDCCLXVI = 1000+500+100+100+50+10+5+1

但是有特殊情况，就是，如果数字的范围在大数减小数的范围内，则会出现小数在大数前面的情况，代表（大数-小数）

IV = 5-1 IX = 10 - 1 = 9 XL = 50-10 = 40

| Symbol | Value |
|--------|-------|
| IV     | 4     |
| IX     | 9     |
| XL     | 40    |
| XC     | 90    |
| CD     | 400   |
| CM     | 900   |

## 思路1——循环

一旦把所有可能的情况符号情况都列举出来了，就好做了。我们现在拿到一个数N

1. 我们就去表里面找不超过它的最大的数x，
2. 然后把它入我们的输出字符串中，然后将数 $N -= x$ ，
3. 继续执行这个操作，直到 $N = 0$

```
public class Solution {
 public String intToRoman(int num) {
 int list[] = {1000, 900, 500, 400, 100, 90, 50, 40, 10, 9, 5, 4, 1};
 String chars[] = {"M", "CM", "D", "CD", "C", "XC", "L", "XL", "X", "IX", "V", "IV", "I"};
 int i = 0;
 String out = "";
 while(num > 0)
 {
 for(; i < list.length; i++)
 if(num >= list[i])
 break;
 out += chars[i];
 num -= list[i];
 }
 return out;
 }
}
```

## 思路2——查表

还有个更极端的方案，就是，把每位上可能出现的情况都列举出来，剩下的，只用查表就行了。

```
public class Solution {
 public static String intToRoman(int num) {
 String M[] = {"", "M", "MM", "MMM"};
 String C[] = {"", "C", "CC", "CCC", "CD", "D", "DC", "DCC", "DCCC", "CM"};
 String X[] = {"", "X", "XX", "XXX", "XL", "L", "LX", "LXX", "LXXX", "XC"};
 String I[] = {"", "I", "II", "III", "IV", "V", "VI", "VII", "VIII", "IX"};
 return M[num/1000] + C[(num%1000)/100] + X[(num%100)/10] + I[num%10];
 }
}
```

## 013. Roman to Integer

### 问题

Given a roman numeral, convert it to an integer.

Input is guaranteed to be within the range from 1 to 3999.

Subscribe to see which companies asked this question

### 思路

首先要知道罗马数字的规律：

| Symbol | Value |
|--------|-------|
| I      | 1     |
| V      | 5     |
| X      | 10    |
| L      | 50    |
| C      | 100   |
| D      | 500   |
| M      | 1,000 |

然后还有一个规则是，罗马数字从左自右相加，但是如果小数字A在大数字B之前，表示B-A

$VI = 5+1 = 6$   $IV = 5-1 = 4$

因此，利用2个变量保存当前数字和之前的数字就行了。因为这题很简单，用python比较方便，我就用python做的

```
class Solution(object):
 def romanToInt(self, s):
 sum=0
 pre = 2000
 cur = 0
 Map = {'I':1, 'V':5, 'X':10, 'L':50, 'C':100, 'D':500, 'M':1000}
 for i in range(len(s)):
 cur = Map[s[i]]
 sum = sum+Map[s[i]]
 if cur > pre :
 sum = sum-2*pre
 pre = cur
 return sum
```

## 题目

Given a set of distinct integers, `nums`, return all possible subsets.

Note: The solution set must not contain duplicate subsets.

For example, If `nums` = [1,2,3], a solution is:

```
[
 [3],
 [1],
 [2],
 [1,2,3],
 [1,3],
 [2,3],
 [1,2],
 []
]
```

## 思路

### 1. 利用bitmap

这是一个非常巧妙的思路，因为对于子集来说，每个元素只有2种状态：在子集中，不在子集中

这刚好符合2进制，可以考虑使用bitmap的方法。

我们对每个元素一个bit：

- 0：在当前子集中
- 1：不在当前子集中

对于n个元素，一共有 $2^n$ 种取法，这和子集数也是一致的。

拿2个元素{1,2}举例：

- 00 ——> [] //1不取，2不取
- 01 ——> [1] //取1
- 10 ——> [2] //取2
- 11 ——> [1,2] //取1，2

刚好可以取尽所有元素。

这个算法的复杂度是 $O(N^2)$ ，一层循环从0到 $2^n$ 种取法，二层循环看每个取法对应的2进制中的1的个数

## 代码

```
public class Solution {

 public List<List<Integer>> subsets(int[] nums) {

 List<List<Integer>> mylist = new ArrayList<List<Integer>>();

 int len = 1<<nums.length;

 //System.out.println(len);

 for(int i = 0;i < len;i++)

 {

 List<Integer> tmpList = new ArrayList<Integer>();

 for(int j = 0;j < nums.length;j++)

 {

 if(((1<<j) & i) != 0)

 {

 tmpList.add(nums[j]);

 }

 }

 mylist.add(tmpList);

 }

 return mylist;

 }

}
```

## 递归，回溯



用回溯法，因为是子集，[1,2]和[2,1]一样，所以我们的每次递归的停止条件都是到nums[]中的最后一个元素。

## 代码

```
public class Solution {

 public List<List<Integer>> subsets(int[] nums) {

 List<List<Integer>> mylist = new ArrayList<List<Integer>>();

 getsubset(mylist,0,nums,new ArrayList<Integer>());

 return mylist;

 }

 private void getsubset(List<List<Integer>> mylist, int cur, int[] nums, List<Integer> tplist){

 mylist.add(tplist);

 for(;cur < nums.length;cur++) {

 List<Integer> newlist = new ArrayList<Integer>(tplist);

 newlist.add(nums[cur]);

 getsubset(mylist, cur + 1,nums,newlist);

 }

 }

}
```

## 004. Median of Two Sorted Arrays[H]

---

### 题目

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  $O(\log(m+n))$ .

### 分析

这个题目是非常的常见，而且有特别多的变形。特别是在当前大数据的环境下，如何快速查找第*i*个元素有很现实的意义。

关注D&C方法的，[直接看思路2](#)

### 思路1

很简单的思路：就是遍历两个数组，在里面找到第*i*个大元素，这个应该还是比较简单的，时间复杂度 $O(m+n)$ 。

用2个变量分别指向两个数组，每次取较小的一个，然后将其指针后移动。但是这里有个问题，就是奇偶判断，如果是奇数，中位数是`num[mid]`，但是如果偶数，是 $(\text{num}[\text{mid}] + \text{num}[\text{mid}-1]) / 2$ 。这里我的做法是把`num[mid]`看作 $(\text{num}[\text{mid}] + \text{num}[\text{mid}]) / 2$ 。如果是偶数-1, 奇数-0。

```
class Solution {
public:
 double findMedianSortedArrays(vector<int>& nums1, vector<int>& nums2) {
 if(nums1.size() == 0)
 return MedofArray(nums2);
 if(nums2.size() == 0)
 return MedofArray(nums1);
 vector<int> num3;
 int size = (nums1.size()+nums2.size());
 int mid = size/2;
 int flag = !(size%2);
 int i,m1,m2,cur;
 double a,b;
 for(i = m1 = m2 = 0;i < size;i++)
 {
 a = m1 < nums1.size()?nums1[m1]:INT_MAX;//过界处理
 b = m2 < nums2.size()?nums2[m2]:INT_MAX;//过界处理
 //cout<<i<<" a "<<a<<" b "<<b<<endl;
 if(a < b)
 {
 num3.push_back(nums1[m1]);
 m1++;
 }
 else
 {
 num3.push_back(nums2[m2]);
 m2++;
 }
 if(i == mid)
 break;
 }
 return (num3[mid]+num3[mid-flag])/2.0;
 }
 double MedofArray(vector<int>& nums)
 {
 int mid = nums.size()/2;
 int flag = !(nums.size()%2);
 return (nums[mid]+nums[mid-flag])/2.0;
 }
};
```

## 思路2

### 重点来了!!

这是一个很经典的Divide & Conquer的题目，关键就在如何划分。这里引用stellari 的高分答案，觉得他这个讲的特别好：

## 预备知识

### 先解释下“割”

我们通过切一刀，能够把有序数组分成左右两个部分，切的那一刀就被称为割(Cut)，割的左右会有两个元素，分别是左边最大值和右边最小值。

我们定义  $L = \text{Max}(\text{LeftPart})$ ， $R = \text{Min}(\text{RightPart})$

Ps. 割可以割在两个数中间，也可以割在1个数上，如果割在一个数上，那么这个数即属于左边，也属于右边。（后面讲单数组中值问题的时候会讲）

比如说[2 3 5 7]这个序列，割就在3和5之间

[2 3 / 5 7]

中值就是  $(3+5) / 2 = 4$

如果[2 3 4 5 6]这个序列，割在4上，我们可以把4分成2个

[2 3 (4/4) 5 7]

中值就是  $(4+4) / 2 = 4$

这样可以保证不管中值是1个数还是2个数都能统一运算。

### 割和第k个元素

对于单数组，找其中的第k个元素特别好做，我们用割的思想就是：

常识1：如果在k的位置割一下，然后A[k]就是L。换言之，就是如果左侧有k个元素，A[k]属于左边部分的最大值。（都是明显的事情，这个不用解释吧！）

### 双数组

我们设:  $C_i$  为第  $i$  个数组的割。  $L_i$  为第  $i$  个数组割后的左元素.  $R_i$  为第  $i$  个数组割后的右元素。

| Leftpart               | $C_i$ | Righthpart                     |
|------------------------|-------|--------------------------------|
| $a_1, a_2, \dots, a_i$ | /     | $a_{i+1}, a_{i+2}, \dots, a_m$ |
| $b_1, b_2, \dots, b_j$ | /     | $b_{j+1}, b_{j+2}, \dots, b_n$ |
| $L_i$                  |       | $R_i$                          |

我们看如何从双数组里取出第  $k$  个元素

1. 首先  $L_i \leq R_i$  是肯定的（因为数组有序，左边肯定小于右边）
2. 如果我们让  $L_1 \leq R_2$  &&  $L_2 \leq R_1$

| Leftpart               | $C_i$ | Righthpart                     |
|------------------------|-------|--------------------------------|
| $a_1, a_2, \dots, a_i$ | /     | $a_{i+1}, a_{i+2}, \dots, a_m$ |
| $b_1, b_2, \dots, b_j$ | /     | $b_{j+1}, b_{j+2}, \dots, b_n$ |
| $L_i$                  |       | $R_i$                          |

3. 那么左半边 全小于右半边，如果左边的元素个数相加刚好等于  $k$ ，那么第  $k$  个元素就是  $\text{Max}(L_1, L_2)$ ，参考上面常识1。
4. 如果  $L_1 > R_2$ ，说明数组1的左边元素太大（多），我们把  $C_1$  减小，把  $C_2$  增大。  $L_2 > R_1$  同理，把  $C_1$  增大，  $C_2$  减小。

假设  $k=3$

对于

[1 4 7 9]

[2 3 5]

设 $C1 = 2$ ，那么 $C2 = k - C1 = 1$

[1 4/7 9]

[2/3 5]

这时候， $L1(4) > R2(3)$ ，说明 $C1$ 要减小， $C2$ 要增大， $C1 = 1$ ， $C2 = k - C1 = 2$

[1/4 7 9]

[2 3/5]

这时候，满足了 $L_1 \leq R_2$  &  $L_2 \leq R_1$ ，第3个元素就是 $\text{Max}(1, 3) = 3$ 。

如果对于上面的例子，把 $k$ 改成4就恰好是中值。

下面具体来看特殊情况的中值问题。

## 双数组的奇偶

中值的关键在于，如何处理奇偶性，单数组的情况，我们已经讨论过了，那双数组的奇偶问题怎么办， $m+n$ 为奇偶处理方案都不同，

## 让数组恒为奇数

有没有办法让两个数组长度相加一定为奇数或偶数呢？

其实有的，虚拟加入'#'(这个trick在[manacher算法](#)中也有应用)，让数组长度恒为奇数（ $2n+1$ 恒为奇数）。

Ps.注意是虚拟加，其实根本没这一步，因为通过下面的转换，我们可以保证虚拟加后每个元素跟原来的元素一一对应

| 之前        | len | 之后                  | len |
|-----------|-----|---------------------|-----|
| [1 4 7 9] | 4   | [# 1 # 4 # 7 # 9 #] | 9   |
| [2 3 5]   | 3   | [# 2 # 3 # 5 #]     | 7   |

## 映射关系

这有什么好处呢，为什么这么加？因为这么加完之后，每个位置可以通过 $/2$ 得到原来元素的位置。

| i | 原位置 | 新位置 | 除2后 |
|---|-----|-----|-----|
| 0 | 1   | 0   | 1   |
| 5 | 2   | 5   | 2   |

## 在虚拟数组里表示“割”

不仅如此，割更容易，如果割在'#'上等于割在2个元素之间，割在数字上等于把数字划到2个部分。

奇妙的是不管哪种情况：

$$L_i = (C_i - 1) / 2$$

$$R_i = C_i / 2$$

例：

1. 割在4/7之间'#'， $C = 4$ ， $L = (4 - 1) / 2 = 1$ ， $R = 4 / 2 = 2$  刚好是4和7的原来位置！
2. 割在3上， $C = 3$ ， $L = (3 - 1) / 2 = 1$ ， $R = 3 / 2 = 1$ ，刚好都是3的位置！

剩下的事情就好办了，把2个数组看做一个虚拟的数组A，目前有 $2m + 2n + 2$ 个元素，割在 $m + n + 1$ 处，所以我们只需找到 $m + n + 1$ 位置的元素和 $m + n + 2$ 位置的元素就行了。(在数组中是 $[m + n]$ 和 $[m + n + 1]$ )

左边： $A[m + n] = \text{Max}(L_1 + L_2)$

右边： $A[m + n + 1] = \text{Min}(R_1 + R_2)$

$$\text{Mid} = (A[m + n] + A[m + n + 1]) / 2 = (\text{Max}(L_1 + L_2) + \text{Min}(R_1 + R_2)) / 2$$

至于在两个数组里找割的方案，就是上面的方案。

## 分治的思路

有了上面的知识后，现在的问题就是如何利用分治的思想。

### 怎么分？

最快的分的方案是二分，有2个数组，我们对哪个做二分呢？根据之前的分析，我们知道了，只要 $C_1$ 或 $C_2$ 确定，另外一个也就确定了。这里，为了效率，我们肯定是选长度较短的做二分，假设为 $C_1$ 。

### 怎么治？

也比较简单，我们之前分析了：就是比较 $L1, L2$ 和 $R1, R2$ 。

- $L1 > R2$ ，把 $C1$ 减小， $C2$ 增大。→  $C1$ 向左二分
- $L2 > R1$ ，把 $C1$ 增大， $C2$ 减小。→  $C1$ 向右二分

## 越界问题

如果 $C1$ 或 $C2$ 已经到头了怎么办？这种情况出现在：如果有个数组完全小于或大于中值。可能有4种情况：

- $C1 = 0$  —— 数组1整体都比中值大， $L1 > R2$ ，中值在2中
- $C2 = 0$  —— 数组1整体都比中值小， $L1 < R2$ ，中值在1中
- $C1 = n*2$  —— 数组1整体都比中值小， $L1 < R2$ ，中位数在2中
- $C2 = m*2$  —— 数组1整体都比中值大， $L1 > R2$ ，中位数在1中

其实，如果我已经确定了数组1是最短的数组，那只有两种情况了，比较好处理：

- 如果 $C1 = 0$  → 那么我们缩小 $L1$ ， $L1 = INT\_MIN$ ，保证判断正确。
- 如果 $C1 = n*2$  → 那么我们增大 $R1$ ， $R1 = INT\_MAX$ ，保证判断正确。

## 代码



```
class Solution {
public:
 double findMedianSortedArrays(vector<int>& nums1, vector<int>& nums2) {
 if(nums1.size() == 0)
 return MedofArray(nums2);
 if(nums2.size() == 0)
 return MedofArray(nums1);
 int n = nums1.size();
 int m = nums2.size();
 if(n > m) //保证数组1一定最短
 return findMedianSortedArrays(nums2,nums1);
 int L1,L2,R1,R2,c1,c2,lo = 0, hi = 2*n; //我们目前是虚拟加了'#'所以数组1是2*n+1长度

 while(lo <= hi) //二分
 {
 c1 = (lo+hi)/2; //c1是二分的结果
 c2 = m+n- c1;
 L1 = (c1 == 0)?INT_MIN:nums1[(c1-1)/2]; //map to original element
 R1 = (c1 == 2*n)?INT_MAX:nums1[c1/2];
 L2 = (c2 == 0)?INT_MIN:nums2[(c2-1)/2];
 R2 = (c2 == 2*m)?INT_MAX:nums2[c2/2];

 if(L1 > R2)
 hi = c1-1;
 else if(L2 > R1)
 lo = c1+1;
 else
 break;
 }
 return (max(L1,L2)+ min(R1,R2))/2.0;
 }
 double MedofArray(vector<int>& nums)
 {
 if(nums.size() == 0) return -1;
 return (nums[nums.size()/2]+nums[(nums.size()-1)/2])/2.0;
 }
};
```

## 010. Regular Expression Matching

@(leetcode解题思路)[DP]

### 问题

Implement regular expression matching with support for '.' and '\*'.

'.' Matches any single character. '\*' Matches zero or more of the preceding element.

The matching should cover the entire input string (not partial).

The function prototype should be: `bool isMatch(const char s, const char p)`

Some examples: `isMatch("aa","a") → false` `isMatch("aa","aa") → true`

`isMatch("aaa","aa") → false` `isMatch("aa","a") → true` `isMatch("aa",".") → true`

`isMatch("ab",".") → true` `isMatch("aab","ca*b") → true`

### 思路

这里面最复杂的操作是"**\***"，这是个很可恶的操作，因为你永远不知道它多长。但是有一点，"**\***"不会单独出现，它一定是和前面一个字母或"."配成一对。看成一对后"**X\***"，它的性质就是：要不匹配0个，要不匹配连续的"**X**"

题目的关键就是如何把这一对放到适合的位置。

考虑一个特殊的问题：情况1：“aaaaaaaaaaaaaaaaa"

"aaa"

情况2：“aaaaaaaaaaaaaaaaa"

"aab"

在不知道后面的情况的时候，我如何匹配**a\***？

- 最长匹配 显然不合适，这样后面的**a**就无法匹配上了
- 匹配到和后面长度一样的位置，比如情况1，就是留3个**a**不匹配，让后面3个字母尝试去匹配。这样看似合适，但是遇到情况2就不行了。
- 回溯，每种"**\***"的情况我都匹配一次，看哪种情况能成功，如果其中出现了问题，马上回溯，换下一种情况

## 思路1——回溯

如果“\*”不好判断，那我大不了就来个暴力的算法，把“\*”的所有可能性都测试一遍看是否有满足的，用两个指针*i,j*来表明当前*s*和*p*的字符。我们采用**从后往前匹配**，为什么这么匹配，**因为如果我们从前往后匹配，每个字符我们都得判断是否后面跟着“\*”，而且还要考虑越界的问题。**但是从后往前没这个问题，一旦遇到“\*”，前面必然有个字符。</font>

- 如果*j*遇到“\*”，我们判断*s[i]* 和 *p[j-1]*是否相同，
  - 如果相同我们可以先尝试匹配掉*s*的这个字符，*i--*，然后看之后能不能满足条件，满足条件，太棒了！我们就结束了，如果中间出现了一个不满足的情况，马上回溯到不匹配这个字符的状态。
  - 不管相同不相同，都不匹配*s*的这个字符，*j-=2* (跳过“\*”前面的字符)

```
if(p[j-1] == '.' || p[j-1] == s[i])
 if(myMatch(s,i-1,p,j))
 return true;
 return myMatch(s,i,p,j-2);
```

- 如果*j*遇到的不是“\*”，那么我们就直接看*s[i]*和*p[j]*是否相等，不相等就说明错了，返回。

```
if(p[j] == '.' || p[j] == s[i])
 return myMatch(s,i-1,p,j-1);
else return false;
```

- 再考虑退出的情况
  - 如果*j*已经<0了说明*p*已经匹配完了，这时候，如果*s*匹配完了，说明正确，如果*s*没匹配完，说明错误。
  - 如果*i*已经<0了说明*s*已经匹配完，这时候，*s*可以没匹配完，只要它还有“\*”存在，我们继续执行代码。

所以代码应该是这样的：

```
class Solution {
public:
 static const int FRONT=-1;
 bool isMatch(string s, string p) {
 return myMatch(s,s.length()-1,p,p.length()-1);
 }
 bool myMatch(string s, int i, string p,int j)
 {
 if(j == FRONT)
 if(i == FRONT) return true;
 else return false;
 if(p[j] == '*')
 {
 if(i > FRONT && (p[j-1] == '.' || p[j-1] == s[i]))
 if(myMatch(s,i-1,p,j))
 return true;
 return myMatch(s,i,p,j-2);
 }
 if(p[j] == '.' || p[j] == s[i])
 return myMatch(s,i-1,p,j-1);
 return false;
 }
};
```

## 思路2——DP

DP的话，肯定要用空间换时间了，这里用 monkeyGoCrazy 的思路：用2维布尔数组，dp[i][j] 的含义是s[0-i] 与 s[0-j]是否匹配。

1. p.charAt(j) == s.charAt(i) : dp[i][j] = dp[i-1][j-1]
2. If p.charAt(j) == '.': dp[i][j] = dp[i-1][j-1];
3. If p.charAt(j) == '\*': here are two sub conditions:

```
- if p.charAt(j-1) != s.charAt(i) : dp[i][j] = dp[i][j-2] //in this case, a*
 only counts as empty
- if p.charAt(j-1) == s.charAt(i) or p.charAt(i-1) == '.':
 dp[i][j] = dp[i-1][j] //in this case, a* counts as multiple a
 dp[i][j] = dp[i][j-1] // in this case, a* counts as single a
 dp[i][j] = dp[i][j-2] // in this case, a* counts as empty
```

这里用的bool数组比较巧妙，初始化为true。前两种情况好理解，如果匹配成功就维持之前的真假值。程序的目的是看真值能不能传递下去。如果遇到三种情况，我们就看哪种情况有真值可以传递，就继续传递下去。

## 初始化

```
dp[0][0] = true;
//初始化第0行,除了[0][0]全为false,毋庸置疑,因为空串p只能匹配空串,其他都无能匹配
for (int i = 1; i <= m; i++)
 dp[i][0] = false;
//初始化第0列,只有X*能匹配空串,如果有*,它的真值一定和p[0][j-2]的相同(略过它之前的符号)
for (int j = 1; j <= n; j++)
 dp[0][j] = j > 1 && '*' == p[j - 1] && dp[0][j - 2];
```

## 图示

我用excel自己跑了下代码,画了一下示意图,下面橘黄色表示正常匹配了,蓝色表示“\*”匹配空串。可以看出真值是如何传递下去的。

例1: "aaa" 和 正则式"aa"

| p \ s |    | "" | a | a |
|-------|----|----|---|---|
|       |    |    |   |   |
| s     | "" | 1  | 0 | 0 |
|       | a  | 0  | 1 | 0 |
|       | a  | 0  |   |   |
|       | a  | 0  |   |   |

| p \ s |    | "" | a | a |
|-------|----|----|---|---|
|       |    |    |   |   |
| s     | "" | 1  | 0 | 0 |
|       | a  | 0  | 1 | 0 |
|       | a  | 0  | 0 | 1 |
|       | a  | 0  | 0 | 0 |

例2: "aabc" 和 正则式"a\*bc"

| <div><div>p</div><div>s</div></div> | ''' | a | * | b | c |
|-------------------------------------|-----|---|---|---|---|
| '''                                 | 1   | 0 | 1 | 0 | 0 |
| a                                   | 0   | 1 | 1 | 0 | 0 |
| a                                   | 0   |   |   |   |   |
| b                                   | 0   |   |   |   |   |
| c                                   | 0   |   |   |   |   |

| <div><div>p</div><div>s</div></div> | ''' | a | * | b | c |
|-------------------------------------|-----|---|---|---|---|
| '''                                 | 1   | 0 | 1 | 0 | 0 |
| a                                   | 0   | 1 | 1 | 0 | 0 |
| a                                   | 0   | 0 | 1 | 0 | 0 |
| b                                   | 0   | 0 | 0 | 1 | 0 |
| c                                   | 0   | 0 | 0 | 0 | 1 |

例3：“aaaa”和正则式“a\*b\*”

| <div><div>p</div><div>s</div></div> | ''' | a | * | b | * |
|-------------------------------------|-----|---|---|---|---|
| '''                                 | 1   | 0 | 1 | 0 | 1 |
| a                                   | 0   | 1 | 1 | 0 | 1 |
| a                                   | 0   |   |   |   |   |
| a                                   | 0   |   |   |   |   |
| a                                   | 0   |   |   |   |   |

| s \ p | "" | a | * | b | * |
|-------|----|---|---|---|---|
| ""    | 1  | 0 | 1 | 0 | 1 |
| a     | 0  | 1 | 1 | 0 | 1 |
| a     | 0  | 0 | 1 | 0 | 1 |
| a     | 0  | 0 | 1 | 0 | 1 |
| a     | 0  | 0 | 1 | 0 | 1 |

## 代码执行

```

for(int i = 1; i <= n; i++)
{
 for(int j = 1; j <= m; j++)
 {
 //这里j-1才是正常字符串中的字符位置
 //要不*当空，要不就只有当前字符匹配了*之前的字符，才有资格传递dp[i-1][j]真值
 if(p[j-1] == '*')
 dp[i][j] = dp[i][j-2] || (s[i-1] == p[j-2] || p[j-2] == '.') && dp[i-1][j];
 else
 //只有当前字符完全匹配，才有资格传递dp[i-1][j-1] 真值
 dp[i][j] = (p[j-1] == '.' || s[i-1] == p[j-1]) && dp[i-1][j-1];
 }
}

```

## 返回值

```
return dp[n][m]
```

## 完整代码

```
class Solution
{
public:
 static const int FRONT=-1;
 bool isMatch(string s, string p)
 {
 int m = s.length(), n = p.length();
 bool dp[m+1][n+1];
 dp[0][0] = true;
 //初始化第0行,除了[0][0]全为false,毋庸置疑,因为空串p只能匹配空串,其他都无能匹配
 for (int i = 1; i <= m; i++)
 dp[i][0] = false;
 //初始化第0列,只有X*能匹配空串,如果有*,它的真值一定和p[0][j-2]的相同(略过它之前的符号)
 for (int j = 1; j <= n; j++)
 dp[0][j] = j > 1 && '*' == p[j - 1] && dp[0][j - 2];

 for (int i = 1; i <= m; i++)
 {
 for (int j = 1; j <= n; j++)
 {
 //由于表格中是从1开始的,而字符串中是以0开始的,所以i-1和j-1才对应字符串中的字符。
 if (p[j - 1] == '*')
 {
 dp[i][j] = dp[i][j - 2] || (s[i - 1] == p[j - 2] || p[j - 2] == '.'
) && dp[i - 1][j];
 }
 else //只有当前字符完全匹配,才有资格传递dp[i-1][j-1] 真值
 {
 dp[i][j] = (p[j - 1] == '.' || s[i - 1] == p[j - 1]) && dp[i - 1][
j - 1];
 }
 }
 }
 return dp[m][n];
 }
};
```



## 120. Triangle[M]

### 题目

Given a triangle, find the minimum path sum from top to bottom. Each step you may move to adjacent numbers on the row below.

For example, given the following triangle [ [2], [3,4], [6,5,7], [4,1,8,3] ] The minimum path sum from top to bottom is 11 (i.e.,  $2 + 3 + 5 + 1 = 11$ ).

Note: Bonus point if you are able to do this using only  $O(n)$  extra space, where  $n$  is the total number of rows in the triangle.

### 思路

如图分析： $d(0,1) = \text{Path}[0,1] + \min(d(1,1), d(1,2))$   
 $d(1,1) = \text{Path}[1,1] + \min(d(2,1), d(2,2))$   
 $d(1,2) = \text{Path}[1,2] + \min(d(2,2), d(2,3))$   
 $\dots$   
 $d(3,1) = 4$   
 $d(3,2) = 1$   
 $d(3,3) = 8$   
 $d(3,4) = 3$   
 我们从底层往上走，每走一层，都是到这一层的最短路径距离。所以，我们每次只用保存到这一层每个元素的最短路径距离即可。也就是说，对于 $n$ 层，我们最多只需要 $n$ 个额外空间。

- 最底层:  $[4, 1, 8, 3]$
- 向上走:  $[6+1, 5+1, 7+3] = [7, 6, 10]$
- 继续:  $[3+6, 4+6] = [9, 10]$
- 最后:  $[2+9] = [11]$

### 代码

```
class Solution {
public:
 int minimumTotal(vector<vector<int>>& triangle) {
 vector<int> min_path(triangle.back());
 for(int i = triangle.size()-2; i>=0; i--)
 {
 for(int j = 0; j < triangle[i].size(); j++)
 {
 min_path[j] = min(min_path[j], min_path[j+1]) + triangle[i][j];
 }
 }
 return min_path[0];
 }
};
```

## 338. Counting Bits [M]

### 题目

Given a non negative integer number num. For every numbers i in the range  $0 \leq i \leq \text{num}$  calculate the number of 1's in their binary representation and return them as an array.

Example: For num = 5 you should return [0,1,1,2,1,2].

Follow up:

1. It is very easy to come up with a solution with run time  $O(n * \text{sizeof}(\text{integer}))$ . But can you do it in linear time  $O(n)$  /possibly in a single pass
2. Space complexity should be  $O(n)$ .
3. Can you do it like a boss? Do it without using any builtin function like `__builtin_popcount` in c++ or in any other language.

### 思路

这是一个比较简单的动规题目，就是统计二进制数的1的个数，通过观察可以发现二进制数有很多规律在里面

| 十进制 | 二进制   |
|-----|-------|
| 1   | 1     |
| 2   | 10    |
| 3   | 11    |
| 4   | 100   |
| 8   | 1000  |
| 9   | 10001 |

可以看见，当二进制位数多1位的时候，高位为1，剩下的和之前的是一样的。

```
list[i] = list[i%n]+1 //n = floor(log(i))

list[2] = list[2%2]+1

list[5] = list[5%4]+1

list[12] =list[12%8]+1
```

## 代码

```
class Solution {
public:
 vector<int> countBits(int num) {
 vector<int> ones(num+1,0);
 int base = 1;
 for(int i = 1; i <= num; i++)
 {
 if(i < base * 2)
 ones[i] = ones[i%base]+1;
 else{
 base *= 2;
 ones[i] = ones[i%base]+1;
 }
 }
 return ones;
 }
};
```

## 更巧妙的代码

最高分代码： $f[i] = f[i/2] + i\%2$

思想：和我的差不多，我是不看最高位，**这个巧妙巧妙在不看最低位，直接右移1位**，这个数在数组中肯定已经存在了，然后加上移走那位是0还是1 ( $i\%2$  或者  $i \& 1$ )

其实整体思想都是把新的二进制模式映射到之前出现的二进制模式上去。

```
public int[] countBits(int num) {
 int[] f = new int[num + 1];
 for (int i=1; i<=num; i++) f[i] = f[i >> 1] + (i & 1);
 return f;
}
```



## 22. Generate Parentheses[M]

### 问题

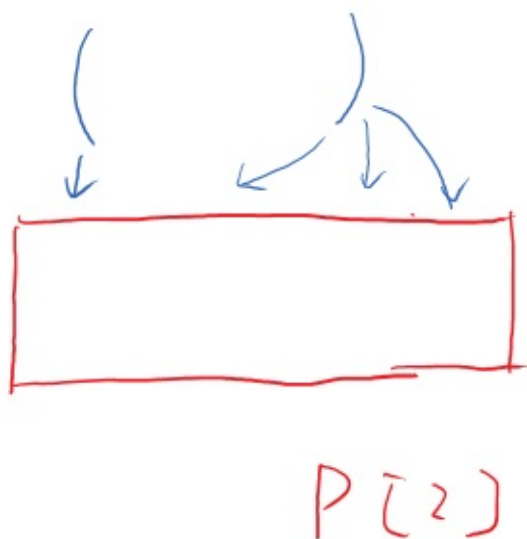
Given  $n$  pairs of parentheses, write a function to generate all combinations of well-formed parentheses.

For example, given  $n = 3$ , a solution set is:

"((()))", "(()())", "(())()", "()()()", "()(())"

### 思路1——DP

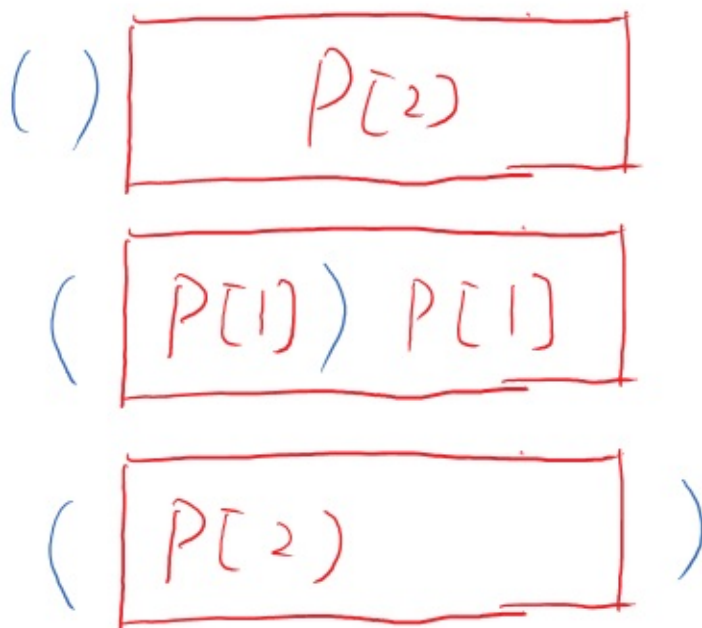
设： $P[i]$ 表示当 $n=i$ 的时候括号组合串。观察规律：我们知道，要形成一个括号的组合，肯定不是凭空产生的，产生一个 $P[3]$ 的组合，那肯定是把"("和")"分别插在 $P[2]$ 中间的。



我们假设产生 $P[3]$ 组合的时候，之前的组合都是正确的，那么通过插入"("，")"肯定会把 $P[2]$ 分成两个部分（括号内一个，括号外一个）

看似好像有很多插入的方法，但是，其实仔细想想，反正"("得增加一个，由于括号组合的第一一定是"("，为什么不把新增的"("放在开头呢？这样我们就只用考虑")"了

会怎么把 $P[2]$ 切割就好了，我们知道 $P[2]$ 的组合有 $P[0]+P[2]$ ， $P[1]+P[1]$ ， $P[2]+P[0]$ ，



通过写出前几个可以观察到下面的规律  $P[0] = [\"\"]$   $P[1] = [()] = \text{"("} + P[0] + \text{"")}$   $P[2] = [()(), ()()] = \text{"("} + P[0] + \text{"")}$   $P[1] + P[0]$ ,  $\text{"("} + P[1] + \text{"")}$   $+ P[0]$   $P[3] = [()()(), ()()(), ()()(), ()()(), ()()()] = \text{"("} + P[0] + \text{"")}$   $+ P[2]$ ,  $\text{"("} + P[1] + \text{"")}$   $+ P[1]$ ,  $\text{"("} + P[2] + \text{"")}$   $+ P[0]$

我们可以知道了组合方式：

- $P[i] = \text{"("} + P[i-j-1] + \text{"")}$   $+ P[j]$  ( $j \in [0, n-1]$ )

```
public class Solution {
 public List<String> generateParenthesis(int n) {
 List<List<String>> result = new ArrayList<List<String>>();
 //初始化P[0] = ""
 result.add((List<String>)Arrays.asList(new String []{""}));
 for(int i=1;i <= n;i++)
 {
 result.add(new ArrayList<>());
 for(int j = 0; j < i;j++)
 {
 //获取P[k]
 for (String s1 : result.get(j)) {
 //获取P[i-j-1]
 for(String s2 : result.get(i-j-1))
 {
 result.get(i).add("(" + s1 + ")" + s2);
 }
 }
 }
 }
 return result.get(n);
 }
}
```

## 思路2：回溯

假设我能枚举所有的情况，我们考虑合理的括号组合是什么样的：

1. 左括号数==右括号数
2. 左括号一定要先于右括号

所以我们可以用一个大数组来表示字符串，2个指针left，right来表示左右括号，我们递归遍历所有情况，把满足条件的情况加入list就行了。



```
public class Solution {
 public List<String> generateParenthesis(int n) {
 LinkedList<String> result = new LinkedList<String>();
 if(n== 0) return result;
 backtracing(result,0,0,"",n);
 return result;
 }
 void backtracing(LinkedList<String> result, int left, int right, String par, int max)
 {
 if(par.length() == 2*max)
 {
 result.add(par);
 return;
 }
 if(left < max)
 backtracing(result, left+1, right, par+'(', max);
 if(right < left)
 backtracing(result, left, right+1, par+')', max);
 }
}
```

## 064. Minimum Path Sum[M]

### 题目

Given a  $m \times n$  grid filled with non-negative numbers, find a path from top left to bottom right which minimizes the sum of all numbers along its path. Note: You can only move either down or right at any point in time.

### 思路

#### DP

这应该是很容易想到的，因为每次只能往下或右走，所以对于每个格，就两种可能：

- 从上面走下来
- 从左边走过来 那么很容易写出表达式： $dp(i,j) = \min(dp(i-1,j), dp(i,j-1)) + path(i,j)$  然后要做一个处理，就是对于第0行，和第0列只能从左边或上面走。这样判断很麻烦，其实有一个很简单的方法：增加一行和一列，让这一行和这一列都为一个很大的值，这样，不管是第0行还是0列，都可以用统一的算法。（这里有个小注意点，就是我对minpath[0][1]赋值为0，这是为了更改原来minpath[0][0]位置的值）

```
public class Solution {
 public int minPathSum(int[][] grid) {
 int m = grid.length;
 if(0 == m)
 return 0;
 int n = grid[0].length;
 int [][] minpath = new int[m+1][n+1];
 //init 0th row and 0th column to Max
 for(int i = 0;i <= m;i ++){
 minpath[i][0] = Integer.MAX_VALUE;
 }
 for(int j = 0;j <= n;j ++){
 minpath[0][j] = Integer.MAX_VALUE;
 }

 minpath[0][1] = 0;

 for(int i = 1;i <= m;i ++){
 for(int j = 1;j <= n;j++){
 minpath[i][j] = Math.min(minpath[i-1][j],minpath[i][j-1]) + grid[i-1][j-1];
 }
 }

 return minpath[m][n];
 }
}
```

## 一维数组

当然，上面的算法还可以改进，就是把二维数组换成一维数组，因为我们实际运行的时候发现，我们每次计算只和当前行的grid和之前行的minpath有关。这个一维数组一直会更新，而更新条件就是： $dp[j] = \min(dp[j], dp[j-1]) + grid(i, j)$  这里同样为了避免第0列出问题（j-1越界），加入了1格，实际j从1开始

```
public class Solution {
 public int minPathSum(int[][] grid) {
 int m = grid.length;
 if(0 == m)
 return 0;
 int n = grid[0].length;
 int [] minpath = new int[n+1];
 for(int i = 0;i <= n;i ++){
 minpath[i] = Integer.MAX_VALUE;
 }

 minpath[1] = 0;

 for(int i = 0;i < m;i ++){
 for(int j = 1;j <= n;j++){
 minpath[j] = Math.min(minpath[j-1],minpath[j]) + grid[i][j-1];
 }
 }

 return minpath[n];
 }
}
```

## 014. Longest Common Prefix[E]

### 问题

Write a function to find the longest common prefix string amongst an array of strings.

Subscribe to see which companies asked this question

### 思路

这个没啥思路的，怎么都要两重循环，因为是最长公共子串，随便找一个(一般是第一个作为基准)，然后拿拿的首部慢慢去匹配后面的字符串就行了。

```
public class Solution {
 public String longestCommonPrefix(String[] strs) {
 String s = "";
 if(strs.length == 0)
 return "";
 for(int i = 0; i < strs[0].length();i++)
 {
 char c = strs[0].charAt(i);
 for(int j = 1;j < strs.length;j++)
 {
 if(i >= strs[j].length() || strs[j].charAt(i) != c)
 return s;
 }
 s += c;
 }
 return s;
 }
}
```

## 20. Valid Parentheses

### 问题

Given a string containing just the characters '(', ')', '{', '}', '[' and ']', determine if the input string is valid.

The brackets must close in the correct order, "()" and "()[]{}" are all valid but "(]" and "([)]" are not.

### 思路

这道题很简单，就是一个经典的栈的例子——表达式中的括号符号匹配。

- 遇见了左括号就进栈
- 遇见了右括号就出栈
  - 如果栈为空，出错
  - 如果出栈元素不是匹配的括号，出错

这里解决出括号匹配用了一个小tick，就是利用ASCII码，匹配的括号的ascii码都不会相差太远

- '(' ')' 相差1
- '[' ']' '{' '}' 相差2

```
public class Solution {
 public boolean isValid(String s) {
 if(s.length() == 0)
 return false;
 Stack<Character> stack = new Stack<Character>(); // 创建堆栈对象
 for(int i = 0; i < s.length(); i++)
 {
 if(s.charAt(i) == '(' || s.charAt(i) == '[' || s.charAt(i) == '{')
 stack.push(s.charAt(i));
 if(s.charAt(i) == ')' || s.charAt(i) == ']' || s.charAt(i) == '}')
 {
 if(stack.empty()) return false;
 char out = stack.pop();
 if(out - s.charAt(i) > 2)
 return false;
 }
 }
 if(stack.empty())
 return true;
 return false;
 }
}
```