1.最长无重复子串：

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 "bbbbb" the longest substring is "b", with the length of 1.

这道求最长无重复子串的题和之前那道[Isomorphic Strings 同构字符串](http://www.cnblogs.com/grandyang/p/4465779.html)很类似，我们还是建立一个256位大小的整型数组来代替哈希表，这样做的原因是ASCII表共能表示256个字符，所以可以记录所有字符，然后我们需要定义两个变量res和left，其中res用来记录最长无重复子串的长度，left指向该无重复子串左边的起始位置，然后我们遍历整个字符串，对于每一个遍历到的字符，如果哈希表中该字符串对应的值为0，说明没有遇到过该字符，则此时计算最长无重复子串，i - left +１，其中ｉ是最长无重复子串最右边的位置，left是最左边的位置，还有一种情况也需要计算最长无重复子串，就是当哈希表中的值小于left，这是由于此时出现过重复的字符，left的位置更新了，如果又遇到了新的字符，就要重新计算最长无重复子串。最后每次都要在哈希表中将当前字符对应的值赋值为i+1。代码如下：

class Solution {

public:

int lengthOfLongestSubstring(string s) {

int m[256] = {0}, res = 0, left = 0;

for (int i = 0; i < s.size(); ++i) {

if (m[s[i]] == 0 || m[s[i]] < left) {

res = max(res, i - left + 1);

} else {

left = m[s[i]];

}

m[s[i]] = i + 1;

}

return res;

}

};

这里解释下程序中那个if条件语句中为啥要有个m[s[i]] < left，我们用一个例子来说明，当输入字符串为"abbca"的时候，当i=4时，也就是即将要开始遍历最后一个字母a时，此时哈希表表中a对应1，b对应3，c对应4，left为2，即当前最长的子字符串的左边界为第二个b的位置，而第一个a已经不在当前最长的字符串的范围内了，那么对于i=4这个新进来的a，应该要加入结果中，而此时未被更新的哈希表中a为1，不是0，如果不判断它和left的关系的话，就无法更新结果，那么答案就会少一位，所以需要加m[s[i]] < left。

2.两个有序数组的中位数

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)).

这道题让我们求两个有序数组的中位数，而且限制了时间复杂度为O(log (m+n))，看到这个时间复杂度，自然而然的想到了应该使用二分查找法来求解。但是这道题被定义为Hard也是有其原因的，难就难在要在两个未合并的有序数组之间使用二分法，这里我们需要定义一个函数来找到第K个元素，由于两个数组长度之和的奇偶不确定，因此需要分情况来讨论，对于奇数的情况，直接找到最中间的数即可，偶数的话需要求最中间两个数的平均值。下面重点来看如何实现找到第K个元素，首先我们需要让数组1的长度小于或等于数组2的长度，那么我们只需判断如果数组1的长度大于数组2的长度的话，交换两个数组即可，然后我们要判断小的数组是否为空，为空的话，直接在另一个数组找第K个即可。还有一种情况是当K = 1时，表示我们要找第一个元素，只要比较两个数组的第一个元素，返回较小的那个即可。

class Solution {

public:

double findMedianSortedArrays(vector<int>& nums1, vector<int>& nums2) {

int m = nums1.size(), n = nums2.size();

return (findKth(nums1, nums2, (m + n + 1) / 2) + findKth(nums1, nums2, (m + n + 2) / 2)) / 2.0;

}

int findKth(vector<int> nums1, vector<int> nums2, int k) {

int m = nums1.size(), n = nums2.size();

if (m > n) return findKth(nums2, nums1, k);

if (m == 0) return nums2[k - 1];

if (k == 1) return min(nums1[0], nums2[0]);

int i = min(m, k / 2), j = min(n, k / 2);

if (nums1[i - 1] > nums2[j - 1]) {

return findKth(nums1, vector<int>(nums2.begin() + j, nums2.end()), k - j);

} else {

return findKth(vector<int>(nums1.begin() + i, nums1.end()), nums2, k - i);

}

return 0;

}

};

3.Z字形转换字符串：

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".

这道题刚开始看了半天没看懂是咋样变换的，上网查了些资料，终于搞懂了，就是要把字符串摆成一个之字型的，参考了网上这位仁兄的解法 ([http://www.cnblogs.com/springfor/p/3889414.html](http://www.cnblogs.com/springfor/p/3889414.html" \t "http://www.cnblogs.com/grandyang/p/_blank))。

比如有一个字符串 “0123456789ABCDEF”，转为zigzag

当 n = 2 时：

0 2 4 6 8 A C E

1 3 5 7 9 B D F

当 n = 3 时：

0   4    8     C

1 3 5 7 9 B D F

2    6   A     E

当 n = 4 时：

0     6        C

1   5 7   B  D

2 4   8 A    E

3      9       F

我们发现，除了第一行和最后一行没有中间形成之字型的数字外，其他都有，而首尾两行中相邻两个元素的index之差跟行数是相关的，为 2\*nRows - 2, 根据这个特点，我们可以按顺序找到所有的黑色元素在元字符串的位置，将他们按顺序加到新字符串里面。对于红色元素出现的位置也是有规律的，每个红色元素的位置为 j + 2\*nRows-2 - 2\*i, 其中，j为前一个黑色元素的列数，i为当前行数。 比如当n = 4中的那个红色5，它的位置为 1 + 2\*4-2 - 2\*1 = 5，为原字符串的正确位置。当我们知道所有黑色元素和红色元素位置的正确算法，我们就可以一次性的把它们按顺序都加到新的字符串里面。代码如下：

class Solution {

public:

string convert(string s, int nRows) {

if (nRows <= 1) return s;

string res = "";

int size = 2 \* nRows - 2;

for (int i = 0; i < nRows; ++i) {

for (int j = i; j < s.size(); j += size) {

res += s[j];

int tmp = j + size - 2 \* i;

if (i != 0 && i != nRows - 1 && tmp < s.size()) res += s[tmp];

}

}

return res;

}

};

4.翻转整数

Reverse digits of an integer.

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

翻转数字问题需要注意的就是溢出问题，看了许多网上的解法，由于之前的OJ没有对溢出进行测试，所以网上很多人的解法没有处理溢出问题也能通过OJ。现在OJ更新了溢出测试，所以还是要考虑到。为什么会存在溢出问题呢，我们知道int型的数值范围是 -2147483648～2147483647， 那么如果我们要翻转 1000000009 这个在范围内的数得到 9000000001，而翻转后的数就超过了范围

class Solution {

public:

int reverse(int x) {

int res = 0;

while (x != 0) {

if (abs(res) > INT\_MAX / 10) return 0;

res = res \* 10 + x % 10;

x /= 10;

}

return res;

}

};

5.验证数字是否是回文的

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

 这道验证回文数字的题不能使用额外空间，意味着不能把整数变成字符，然后来验证回文字符串。而是直接对整数进行操作，我们可以利用取整和取余来获得我们想要的数字，比如 1221 这个数字，如果 计算 1221 / 1000， 则可得首位1， 如果 1221 % 10， 则可得到末尾1，进行比较，然后把中间的22取出继续比较。代码如下：

class Solution {

public:

bool isPalindrome(int x) {

if (x < 0) return false;

int div = 1;

while (x / div >= 10) div \*= 10;

while (x > 0) {

int left = x / div;

int right = x % 10;

if (left != right) return false;

x = (x % div) / 10;

div /= 100;

}

return true;

}

};

6.正则表达式匹配：

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", "c\*a\*b") → true

这道求正则表达式匹配的题和那道[Wildcard Matching 通配符匹配](http://www.cnblogs.com/grandyang/p/4401196.html)的题很类似，不同点在于\*的意义不同，在之前那道题中，\*表示可以代替任意个数的字符，而这道题中的\*表示之前那个字符可以有0个，1个或是多个，就是说，字符串a\*b，可以表示b或是aaab，即a的个数任意，这道题的难度要相对之前那一道大一些，分的情况的要复杂一些，需要用递归Recursion来解，大概思路如下：

- 若p为空，若s也为空，返回true，反之返回false

- 若p的长度为1，若s长度也为1，且相同或是p为'.'则返回true，反之返回false

- 若p的第二个字符不为\*，若此时s为空返回false，否则判断首字符是否匹配，且从各自的第二个字符开始调用递归函数匹配

- 若p的第二个字符为\*，若s不为空且字符匹配，调用递归函数匹配s和去掉前两个字符的p，若匹配返回true，否则s去掉首字母

- 返回调用递归函数匹配s和去掉前两个字符的p的结果

class Solution {

public:

bool isMatch(string s, string p) {

if (p.empty()) return s.empty();

if (p.size() == 1) {

return (s.size() == 1 && (s[0] == p[0] || p[0] == '.'));

}

if (p[1] != '\*') {

if (s.empty()) return false;

return (s[0] == p[0] || p[0] == '.') && isMatch(s.substr(1), p.substr(1));

}

while (!s.empty() && (s[0] == p[0] || p[0] == '.')) {

if (isMatch(s, p.substr(2))) return true;

s = s.substr(1);

}

return isMatch(s, p.substr(2));

}

};

7.三数之和等于0

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 ≤ b ≤ 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)

这道题让我们求三数之和，比之前那道[Two Sum 两数之和](http://www.cnblogs.com/grandyang/p/4130379.html)要复杂一些，我们还是要首先对原数组进行排序，然后开始遍历排序后的数组，这里注意不是遍历到最后一个停止，而是到倒数第三个就可以了，然后我们还要加上重复就跳过的处理，对于遍历到的数，我们用0减去这个数得到一个sum，我们只需要再之后找到两个数之和等于sum即可，这样一来问题又转化为了求two sum，这时候我们一次扫描，找到了等于sum的两数后，加上当前遍历到的数字，按顺序存入结果中即可，然后还要注意跳过重复数字。代码如下：

 class Solution {

public:

vector<vector<int>> threeSum(vector<int>& nums) {

vector<vector<int>> res;

sort(nums.begin(), nums.end());

for (int k = 0; k < nums.size(); ++k) {

if (nums[k] > 0) break;//num[k]为主元素，只计算排序后它后面，所以必须小于0

if (k > 0 && nums[k] == nums[k - 1]) continue;//主元素重复跳过

int target = 0 - nums[k];

int i = k + 1, j = nums.size() - 1;

while (i < j) {

if (nums[i] + nums[j] == target) {

res.push\_back({nums[k], nums[i], nums[j]});

while (i < j && nums[i] == nums[i + 1]) ++i;//相同的跳过

while (i < j && nums[j] == nums[j - 1]) --j;///相同的跳过

++i; --j;

} else if (nums[i] + nums[j] < target) ++i;

else --j;

}

}

return res;

}

};

7.2最近的三数之和

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).

这道题让我们求最接近给定值的三数之和，是在之前那道[3Sum 三数之和](http://www.cnblogs.com/grandyang/p/4481576.html)的基础上又增加了些许难度，那么这道题让我们返回这个最接近于给定值的值，即我们要保证当前三数和跟给定值之间的差的绝对值最小，所以我们需要定义一个变量diff用来记录差的绝对值，然后我们还是要先将数组排个序，然后开始遍历数组，思路跟那道三数之和很相似，都是先确定一个数，然后用两个指针left和right来滑动寻找另外两个数，每确定两个数，我们求出此三数之和，然后算和给定值的差的绝对值存在newDiff中，然后和diff比较并更新diff和结果closest即可，代码如下：

class Solution {

public:

int threeSumClosest(vector<int>& nums, int target) {

int closest = nums[0] + nums[1] + nums[2];

int diff = abs(closest - target);

sort(nums.begin(), nums.end());

for (int i = 0; i < nums.size() - 2; ++i) {

int left = i + 1, right = nums.size() - 1;

while (left < right) {

int sum = nums[i] + nums[left] + nums[right];

int newDiff = abs(sum - target);

if (diff > newDiff) {

diff = newDiff;

closest = sum;

}

if (sum < target) ++left;

else --right;

}

}

return closest;

}

};

7.3 四数之和

Given an array S of n integers, are there elements a, b, c, and d in S such that a + b + c + d = 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 ≤ b ≤ c ≤ 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)

LeetCode中关于数字之和还有其他几道，分别是[Two Sum 两数之和](http://www.cnblogs.com/grandyang/p/4130379.html)，[3Sum 三数之和](http://www.cnblogs.com/grandyang/p/4481576.html)，[3Sum Closest 最近三数之和](http://www.cnblogs.com/grandyang/p/4510984.html)，虽然难度在递增，但是整体的套路都是一样的，在这里为了避免重复项，我们使用了STL中的set，其特点是不能有重复，如果新加入的数在set中原本就存在的话，插入操作就会失败，这样能很好的避免的重复项的存在。此题的O(n^3)解法的思路跟[3Sum 三数之和](http://www.cnblogs.com/grandyang/p/4481576.html)基本没啥区别，就是多加了一层for循环，其他的都一样，代码如下：

// O(n^3)

class Solution {

public:

vector<vector<int> > fourSum(vector<int> &nums, int target) {

set<vector<int> > res;

sort(nums.begin(), nums.end());

for (int i = 0; i < int(nums.size() - 3); ++i) {

for (int j = i + 1; j < int(nums.size() - 2); ++j) {

int left = j + 1, right = nums.size() - 1;

while (left < right) {

int sum = nums[i] + nums[j] + nums[left] + nums[right];

if (sum == target) {

vector<int> out;

out.push\_back(nums[i]);

out.push\_back(nums[j]);

out.push\_back(nums[left]);

out.push\_back(nums[right]);

res.insert(out);

++left; --right;

} else if (sum < target) ++left;

else --right;

}

}

}

return vector<vector<int> > (res.begin(), res.end());

}

};

8.字符串系列的最长公共前缀

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

这道题让我们求一系列字符串的共同前缀，没有什么特别的技巧，无脑查找即可，我们定义两个变量i和j，其中i是遍历搜索字符串中的每个字符串，即行数，j是遍历字符串集中的每个字符，即列数。这里将单词上下排好，则相当于一个各行长度有可能不相等的二维数组，我们遍历顺序和一般的横向逐行遍历不同，而是采用纵向逐列遍历，在遍历的过程中，如果某一行没有了，说明其为最短的单词，因为共同前缀的长度不能长于最短单词，所以此时返回已经找出的共同前缀。我们每次取出第一个字符串的某一个位置的单词，然后遍历其他所有字符串的对应位置看是否相等，如果有不满足的直接返回res，如果都相同，则将当前字符存入结果，继续检查下一个位置 的字符，参见代码如下：

class Solution {

public:

string longestCommonPrefix(vector<string>& strs) {

if (strs.empty()) return "";

string res = "";

for (int j = 0; j < strs[0].size(); ++j) {

char c = strs[0][j];

for (int i = 1; i < strs.size(); ++i) {

if (j >= strs[i].size() || strs[i][j] != c) {

return res;

}

}

res.push\_back(c);

}

return res;

}

};

9.罗马数字转换为整数

Given a roman numeral, convert it to an integer.

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

[罗马数](http://zh.wikipedia.org/wiki/%E7%BD%97%E9%A9%AC%E6%95%B0%E5%AD%97" \t "http://www.cnblogs.com/grandyang/p/_blank)转化成数字问题，我们需要对于[罗马数字](http://baike.baidu.com/view/42061.htm?fr=aladdin" \t "http://www.cnblogs.com/grandyang/p/_blank)很熟悉才能完成转换。以下截自百度百科：

[罗马](http://baike.baidu.com/view/2245.htm" \t "http://www.cnblogs.com/grandyang/p/_blank)**数字是最早的数字表示方式，比阿拉伯数字早2000多年，起源于罗马。**

如今我们最常见的罗马数字就是钟表的表盘符号：**Ⅰ，Ⅱ，Ⅲ，Ⅳ（IIII），Ⅴ，Ⅵ，Ⅶ，Ⅷ，Ⅸ，Ⅹ，Ⅺ，Ⅻ……**

对应阿拉伯数字（就是现在国际通用的数字），就是1，2，3，4，5，6，7，8，9，10，11，12。（注：**阿拉伯数字**其实是古代[印度](http://baike.baidu.com/view/2174.htm" \t "http://www.cnblogs.com/grandyang/p/_blank)人发明的，后来由阿拉伯人传入[欧洲](http://baike.baidu.com/view/3622.htm" \t "http://www.cnblogs.com/grandyang/p/_blank)，被欧洲人误称为阿拉伯数字。）

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 基本字符 | I | V | X | L | C | D | M |
| 相应的阿拉伯数字表示为 | 1 | 5 | 10 | 50 | 100 | 500 | 1000 |

1、相同的数字连写，所表示的数等于这些数字相加得到的数，如：Ⅲ = 3；

2、小的数字在大的数字的右边，所表示的数等于这些数字相加得到的数， 如：Ⅷ = 8；Ⅻ = 12；

3、小的数字，（限于Ⅰ、X 和C）在大的数字的左边，所表示的数等于大数减小数得到的数，如：Ⅳ= 4；Ⅸ= 9；

4、正常使用时，连写的数字重复不得超过三次。（表盘上的四点钟“IIII”例外）

5、在一个数的上面画一条横线，表示这个数扩大1000倍。

**有几条须注意掌握：**

1、基本数字Ⅰ、X 、C 中的任何一个，自身连用构成数目，或者放在大数的右边连用构成数目，都不能超过三个；放在大数的左边只能用一个。

2、不能把基本数字V 、L 、D 中的任何一个作为小数放在大数的左边采用相减的方法构成数目；放在大数的右边采用相加的方式构成数目，只能使用一个。

3、V 和X 左边的小数字只能用Ⅰ。

4、L 和C 左边的小数字只能用X。

5、D 和M 左边的小数字只能用C。

而这道题好就好在没有让我们来验证输入字符串是不是罗马数字，这样省掉不少功夫。我们需要用到map数据结构，来将罗马数字的字母转化为对应的整数值，因为输入的一定是罗马数字，那么我们只要考虑两种情况即可：

第一，如果当前数字是最后一个数字，或者之后的数字比它小的话，则加上当前数字

第二，其他情况则减去这个数字

class Solution {

public:

int romanToInt(string s) {

int res = 0;

unordered\_map<char, int> m{{'I', 1}, {'V', 5}, {'X', 10}, {'L', 50}, {'C', 100}, {'D', 500}, {'M', 1000}};

for (int i = 0; i < s.size(); ++i) {

int val = m[s[i]];

if (i == s.size() - 1 || m[s[i+1]] <= m[s[i]]) res += val;

else res -= val;

}

return res;

}

};

10.整数转化为罗马数字

Given an integer, convert it to a roman numeral.

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

之前那篇文章写的是罗马数字转化成整数([http://www.cnblogs.com/grandyang/p/4120857.html](http://www.cnblogs.com/grandyang/p/4120857.html" \t "http://www.cnblogs.com/grandyang/p/_blank))， 这次变成了整数转化成罗马数字，基本算法还是一样。由于题目中限定了输入数字的范围(1 - 3999), 使得题目变得简单了不少。

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 基本字符 | I | V | X | L | C | D | M |
| 相应的阿拉伯数字表示为 | 1 | 5 | 10 | 50 | 100 | 500 | 1000 |

例如整数 1437 的罗马数字为 MCDXXXVII， 我们不难发现，千位，百位，十位和个位上的数分别用罗马数字表示了。 1000 - M, 400 - CD, 30 - XXX, 7 - VII。所以我们要做的就是用取商法分别提取各个位上的数字，然后分别表示出来：

100 - C

200 - CC

300 - CCC

400 - CD

500 - D

600 - DC

700 - DCC

800 - DCCC

900 - CM

我们可以分为四类，100到300一类，400一类，500到800一类，900最后一类。每一位上的情况都是类似的，代码如下：

class Solution {

public:

string intToRoman(int num) {

string res = "";

char roman[] = {'M', 'D', 'C', 'L', 'X', 'V', 'I'};

int value[] = {1000, 500, 100, 50, 10, 5, 1};

for (int n = 0; n < 7; n += 2) {

int x = num / value[n];

if (x < 4) {

for (int i = 1; i <= x; ++i) res += roman[n-1];

} else if (x == 4) res = res + roman[n] + roman[n - 1];

else if (x > 4 && x < 9) {

res += roman[n - 1];

for (int i = 6; i <= x; ++i) res += roman[n];

}

else if (x == 9) res = res + roman[n] + roman[n - 2];

num %= value[n];

}

return res;

}

};

11.装水最大的容器

Given n non-negative integers a1, a2, ..., an, where each represents a point at coordinate (i, ai). n vertical lines are drawn such that the two endpoints of line i is at (i, ai) 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.

这道求装最多水的容器的题和那道[Trapping Rain Water 收集雨水](http://www.cnblogs.com/grandyang/p/4402392.html)很类似，但又有些不同，那道题让求整个能收集雨水的量，这道只是让求最大的一个的装水量，而且还有一点不同的是，那道题容器边缘不能算在里面，而这道题却可以算，相比较来说还是这道题容易一些，我们需要定义i和j两个指针分别指向数组的左右两端，然后两个指针向中间搜索，每移动一次算一个值和结果比较取较大的，容器装水量的算法是找出左右两个边缘中较小的那个乘以两边缘的距离，代码如下

class Solution {

public:

int maxArea(vector<int>& height) {

int res = 0, i = 0, j = height.size() - 1;

while (i < j) {

res = max(res, min(height[i], height[j]) \* (j - i));

If(height[i] < height[j]) ++i ;

else

--j;

}

return res;

}

};

1. 移除链表倒数第n个节点

Given a linked list, remove the nth 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**.

**Note:**  
Given n will always be valid.  
Try to do this in one pass.

这道题让我们移除链表倒数第N个节点，限定n一定是有效的，即n不会大于链表中的元素总数。还有题目要求我们一次遍历解决问题，那么就得想些比较巧妙的方法了。比如我们首先要考虑的时，如何找到倒数第N个节点，由于只允许一次遍历，所以我们不能用一次完整的遍历来统计链表中元素的个数，而是遍历到对应位置就应该移除了。那么我们需要用两个指针来帮助我们解题，pre和cur指针。首先cur指针先向前走n步，如果此时cur指向空，说明n为链表的长度，则需要移除的为首元素，那么此时我们返回head->next即可，如果cur存在，我们再继续往下走，此时pre指针也跟着走，直到cur为最后一个元素时停止，此时pre指向要移除元素的前一个元素，我们再修改指针跳过需要移除的元素即可。代码如下：

class Solution {

public:

ListNode\* removeNthFromEnd(ListNode\* head, int n) {

if (!head->next) return NULL;

ListNode \*pre = head, \*cur = head;

for (int i = 0; i < n; ++i) cur = cur->next;

if (!cur) return head->next;

while (cur->next) {

cur = cur->next;

pre = pre->next;

}

pre->next = pre->next->next;

return head;

}

};

1. 1验证括号

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.

这道题让我们验证输入的字符串是否为括号字符串，包括大括号，中括号和小括号。这里我们需要用一个栈，我们开始遍历输入字符串，如果当前字符为左半边括号时，则将其压入栈中，如果遇到右半边括号时，若此时栈为空，则直接返回false，如不为空，则取出栈顶元素，若为对应的左半边括号，则继续循环，反之返回false，代码如下：

class Solution {

public:

bool isValid(string s) {

stack<char> parentheses;

for (int i = 0; i < s.size(); ++i) {

if (s[i] == '(' || s[i] == '[' || s[i] == '{') parentheses.push(s[i]);

else {

if (parentheses.empty()) return false;

if (s[i] == ')' && parentheses.top() != '(') return false;

if (s[i] == ']' && parentheses.top() != '[') return false;

if (s[i] == '}' && parentheses.top() != '{') return false;

parentheses.pop();

}

}

return parentheses.empty();

}

};

13.2 生成括号

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:

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

在LeetCode中有关括号的题共有三道，除了这一道的另外两道是[Valid Parentheses 验证括号](http://www.cnblogs.com/grandyang/p/4424587.html)和[Longest Valid Parentheses 最长有效括号](http://www.cnblogs.com/grandyang/p/4424731.html)，这道题给定一个数字n，让生成共有n个括号的所有正确的形式，对于这种列出所有结果的题首先还是考虑用递归Recursion来解，由于字符串只有左括号和右括号两种字符，而且最终结果必定是左括号3个，右括号3个，所以我们定义两个变量left和right分别表示剩余左右括号的个数，如果在某次递归时，左括号的个数大于右括号的个数，说明此时生成的字符串中右括号的个数大于左括号的个数，即会出现')('这样的非法串，所以这种情况直接返回，不继续处理。如果left和right都为0，则说明此时生成的字符串已有3个左括号和3个右括号，且字符串合法，则存入结果中后返回。如果以上两种情况都不满足，若此时left大于0，则调用递归函数，注意参数的更新，若right大于0，则调用递归函数，同样要更新参数。代码如下：

class Solution {

public:

vector<string> generateParenthesis(int n) {

vector<string> res;

generateParenthesisDFS(n, n, "", res);

return res;

}

void generateParenthesisDFS(int left, int right, string out, vector<string> &res) {

if (left > right) return;

if (left == 0 && right == 0) res.push\_back(out);

else {

if (left > 0) generateParenthesisDFS(left - 1, right, out + '(', res);

if (right > 0) generateParenthesisDFS(left, right - 1, out + ')', res);

}

}

};

13.3 最长有效括号

Given a string containing just the characters '(' and ')', find the length of the longest valid (well-formed) parentheses substring.

For "(()", the longest valid parentheses substring is "()", which has length = 2.

Another example is ")()())", where the longest valid parentheses substring is "()()", which has length = 4.

这道求最长有效括号比之前那道[Valid Parentheses 验证括号](http://www.cnblogs.com/grandyang/p/4424587.html)难度要大一些，这里我们还是借助栈来求解，需要定义个start变量来记录合法括号串的起始位置，我们遍历字符串，如果遇到左括号，则将当前下标压入栈，如果遇到右括号，如果当前栈为空，则将下一个坐标位置记录到start，如果栈不为空，则将栈顶元素取出，此时若栈为空，则更新结果和i - start + 1中的较大值，否则更新结果和i - 栈顶元素中的较大值，代码如下：

class Solution {

public:

int longestValidParentheses(string s) {

int res = 0, start = 0;

stack<int> m;

for (int i = 0; i < s.size(); ++i) {

if (s[i] == '(') m.push(i);

else if (s[i] == ')') {

if (m.empty()) start = i + 1;

else {

m.pop();

res = m.empty() ? max(res, i - start + 1) : max(res, i - m.top());

}

}

}

return res;

}

};

14. 合并K个有序链表

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

这道题让我们合并k个有序链表，之前我们做过一道[Merge Two Sorted Lists 混合插入有序链表](http://www.cnblogs.com/grandyang/p/4086297.html)，是混合插入两个有序链表。这道题增加了难度，变成合并k个有序链表了，但是不管合并几个，基本还是要两两合并。那么我们首先考虑的方法是能不能利用之前那道题的解法来解答此题。答案是肯定的，但是需要修改，怎么修改呢，最先想到的就是两两合并，就是前两个先合并，合并好了再跟第三个，然后第四个直到第k个。这样的思路是对的，但是效率不高，没法通过OJ，所以我们只能换一种思路，这里就需要用到分治法 Divide and Conquer Approach。简单来说就是不停的对半划分，比如k个链表先划分为合并两个k/2个链表的任务，再不停的往下划分，直到划分成只有一个或两个链表的任务，开始合并。举个例子来说比如合并6个链表，那么按照分治法，我们首先分别合并1和4,2和5,3和6。这样下一次只需合并3个链表，我们再合并1和3，最后和2合并就可以了。参见代码如下：

解法一：分治思想

class Solution {

public:

ListNode \*mergeKLists(vector<ListNode \*> &lists) {

if (lists.size() == 0) return NULL;

int n = lists.size();

while (n > 1) {

int k = (n + 1) / 2;

for (int i = 0; i < n / 2; ++i) {

lists[i] = mergeTwoLists(lists[i], lists[i + k]);

}

n = k;

}

return lists[0];

}

ListNode \*mergeTwoLists(ListNode \*l1, ListNode \*l2) {

ListNode \*head = new ListNode(-1);

ListNode \*cur = head;

while (l1 && l2) {

if (l1->val < l2->val) {

cur->next = l1;

l1 = l1->next;

} else {

cur->next = l2;

l2 = l2->next;

}

cur = cur->next;

}

if (l1) cur->next = l1;

if (l2) cur->next = l2;

return head->next;

}

};

我们再来看另一种解法，这种解法利用了[最小堆](https://zh.wikipedia.org/wiki/%E6%9C%80%E5%A4%A7%E2%80%94%E6%9C%80%E5%B0%8F%E5%A0%86" \t "http://www.cnblogs.com/grandyang/p/_blank)这种数据结构，我们首先把k个链表的首元素都加入最小堆中，它们会自动排好序。然后我们每次取出最小的那个元素加入我们最终结果的链表中，然后把取出元素的下一个元素再加入堆中，下次仍从堆中取出最小的元素做相同的操作，以此类推，直到堆中没有元素了，此时k个链表也合并为了一个链表，返回首节点即可，代码如下：

归并思想：

struct cmp {

bool operator () (ListNode \*a, ListNode \*b) {

return a->val > b->val;

}

};

class Solution {

public:

ListNode \*mergeKLists(vector<ListNode \*> &lists) {

priority\_queue<ListNode\*, vector<ListNode\*>, cmp> q;

for (int i = 0; i < lists.size(); ++i) {

if (lists[i]) q.push(lists[i]);

}

ListNode \*head = NULL, \*pre = NULL, \*tmp = NULL;

while (!q.empty()) {

tmp = q.top();

q.pop();

if (!pre) head = tmp;

else pre->next = tmp;

pre = tmp;

if (tmp->next) q.push(tmp->next);

}

return head;

}

};

15.找出短字符串在长字符串中第一次出现的位置

Implement strStr().

Returns the index of the first occurrence of needle in haystack, or -1 if needle is not part of haystack.

**Update (2014-11-02):**  
The signature of the function had been updated to return the index instead of the pointer. If you still see your function signature returns a char \* or String, please click the reload button to reset your code definition.

这道题让我们在一个字符串中找另一个字符串第一次出现的位置，那我们首先要做一些判断，如果子字符串为空，则返回0，如果子字符串长度大于母字符串长度，则返回-1。然后我们开始遍历母字符串，我们并不需要遍历整个母字符串，而是遍历到剩下的长度和子字符串相等的位置即可，这样可以提高运算效率。然后对于每一个字符，我们都遍历一遍子字符串，一个一个字符的对应比较，如果对应位置有不等的，则跳出循环，如果一直都没有跳出循环，则说明子字符串出现了，则返回起始位置即可，代码如下：

class Solution {

public:

int strStr(string haystack, string needle) {

if (needle.empty()) return 0;

int m = haystack.size(), n = needle.size();

if (m < n) return -1;

for (int i = 0; i <= m - n; ++i) {

int j = 0;

for (j = 0; j < n; ++j) {

if (haystack[i + j] != needle[j]) break;

}

if (j == n) return i;

}

return -1;

}

};

16.两数相除

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

If it is overflow, return MAX\_INT.

这道题让我们求两数相除，而且规定我们不能用乘法，除法和取余操作，那么我们还可以用另一神器位操作Bit Operation，思路是，如果被除数大于或等于除数，则进行如下循环，定义变量t等于除数，定义计数p，当t的两倍小于等于被除数时，进行如下循环，t扩大一倍，p扩大一倍，然后更新res和m。这道题的OJ给的一些test case非常的讨厌，因为输入的都是int型，比如被除数是-2147483648，在int范围内，当除数是-1时，结果就超出了int范围，需要返回INT\_MAX，所以对于这种情况我们就在开始用if判定，将其和除数为0的情况放一起判定，返回INT\_MAX。然后我们还要根据被除数和除数的正负来确定返回值的正负，这里我们采用长整型long来完成所有的计算，最后返回值乘以符号即可，代码如下：

class Solution {

public:

int divide(int dividend, int divisor) {

if (divisor == 0 || (dividend == INT\_MIN && divisor == -1)) return INT\_MAX;

long long m = abs((long long)dividend), n = abs((long long)divisor), res = 0;

int sign = ((dividend < 0) ^ (divisor < 0)) ? -1 : 1;

if (n == 1) return sign == 1 ? m : -m;

while (m >= n) {

long long t = n, p = 1;

while (m >= (t << 1)) {

t <<= 1;

p <<= 1;

}

res += p;

m -= t;

}

return sign == 1 ? res : -res;

}

};

17.下一个排列

Implement next permutation, which rearranges numbers into the lexicographically next greater permutation of numbers.

If such arrangement is not possible, it must rearrange it as the lowest possible order (ie, sorted in ascending order).

The replacement must be in-place, do not allocate extra memory.

Here are some examples. Inputs are in the left-hand column and its corresponding outputs are in the right-hand column.  
1,2,3 → 1,3,2  
3,2,1 → 1,2,3  
1,1,5 → 1,5,1

这道题让我们求下一个排列顺序，有题目中给的例子可以看出来，如果给定数组是降序，则说明是全排列的最后一种情况，则下一个排列就是最初始情况，可以参见之前的博客[Permutations 全排列](http://www.cnblogs.com/grandyang/p/4358848.html)。我们再来看下面一个例子，有如下的一个数组

1　　2　　7　　4　　3　　1

下一个排列为：

1　　3　　1　　2　　4　　7

那么是如何得到的呢，我们通过观察原数组可以发现，如果从末尾往前看，数字逐渐变大，到了2时才减小的，然后我们再从后往前找第一个比2大的数字，是3，那么我们交换2和3，再把此时3后面的所有数字转置一下即可，步骤如下：

1　　2　　7　　4　　3　　1

1　　2　　7　　4　　3　　1

1　　3　　7　　4　　2　　1

1　　3　　1　　2　　4　　7

class Solution {

public:

void nextPermutation(vector<int> &num) {

int i, j, n = num.size();

for (i = n - 2; i >= 0; --i) {

if (num[i + 1] > num[i]) {

for (j = n - 1; j >= i; --j) {

if (num[j] > num[i]) break;

}

swap(num[i], num[j]);

reverse(num.begin() + i + 1, num.end());

return;

}

}

reverse(num.begin(), num.end());

}

};

1. 搜索插入位置

Given a sorted array and a target value, return the index if the target is found. If not, return the index where it would be if it were inserted in order.

You may assume no duplicates in the array.

Here are few examples.  
[1,3,5,6], 5 → 2  
[1,3,5,6], 2 → 1  
[1,3,5,6], 7 → 4  
[1,3,5,6], 0 → 0

这道题基本没有什么难度，实在不理解为啥还是Medium难度的，完完全全的应该是Easy啊，三行代码搞定的题，只需要遍历一遍原数组，若当前数字大于或等于目标值，则返回当前坐标，如果遍历结束了，说明目标值比数组中任何一个数都要大，则返回数组长度n即可，代码如下：

class Solution {

public:

int searchInsert(vector<int>& nums, int target) {

for (int i = 0; i < nums.size(); ++i) {

if (nums[i] >= target) return i;

}

return nums.size();

}

};

当然，我们还可以用二分搜索法来优化我们的时间复杂度，而且个人认为这种方法应该是面试官们想要考察的算法吧，参见代码如下：

 class Solution {

public:

int searchInsert(vector<int>& nums, int target) {

if (nums.back() < target) return nums.size();

int left = 0, right = nums.size() - 1;

while (left < =right) {

int mid = left + (right - left) / 2;

if (nums[mid] == target) return mid;

else if (nums[mid] < target) left = mid + 1;

else right = mid;

}

return right;

}

};

19.1 组合之和

Given a set of candidate numbers (C) and a target number (T), find all unique combinations in C where the candidate numbers sums to T.

The same repeated number may be chosen from C unlimited number of times.

Note:

* All numbers (including target) will be positive integers.
* Elements in a combination (a1, a2, … , ak) must be in non-descending order. (ie, a1 ≤ a2 ≤ … ≤ ak).
* The solution set must not contain duplicate combinations.

For example, given candidate set 2,3,6,7 and target 7,   
A solution set is:   
[7]   
[2, 2, 3]

像这种结果要求返回所有符合要求解的题十有八九都是要利用到递归，而且解题的思路都大同小异，相类似的题目有[Path Sum II 二叉树路径之和之二](http://www.cnblogs.com/grandyang/p/4042156.html)，[Subsets II 子集合之二](http://www.cnblogs.com/grandyang/p/4310964.html)，[Permutations 全排列](http://www.cnblogs.com/grandyang/p/4358848.html)，[Permutations II 全排列之二](http://www.cnblogs.com/grandyang/p/4359825.html)，[Combinations 组合项](http://www.cnblogs.com/grandyang/p/4332522.html)等等，如果仔细研究这些题目发现都是一个套路，都是需要另写一个递归函数，这里我们新加入三个变量，start记录当前的递归到的下标，out为一个解，res保存所有已经得到的解，每次调用新的递归函数时，此时的target要减去当前数组的的数，具体看代码如下：

 class Solution {

public:

vector<vector<int> > combinationSum(vector<int> &candidates, int target) {

vector<vector<int> > res;

vector<int> out;

sort(candidates.begin(), candidates.end());

combinationSumDFS(candidates, target, 0, out, res);

return res;

}

void combinationSumDFS(vector<int> &candidates, int target, int start, vector<int> &out, vector<vector<int> > &res) {

if (target < 0) return;

else if (target == 0) res.push\_back(out);

else {

for (int i = start; i < candidates.size(); ++i) {

out.push\_back(candidates[i]);

combinationSumDFS(candidates, target - candidates[i], i, out, res);

out.pop\_back();

}

}

}

};

19.2组合之和（不可重复使用数组数字）

Given a collection of candidate numbers (C) and a target number (T), find all unique combinations in C where the candidate numbers sums to T.

Each number in C may only be used once in the combination.

Note:

* All numbers (including target) will be positive integers.
* Elements in a combination (a1, a2, … , ak) must be in non-descending order. (ie, a1 ≤ a2 ≤ … ≤ ak).
* The solution set must not contain duplicate combinations.

For example, given candidate set 10,1,2,7,6,1,5 and target 8,   
A solution set is:   
[1, 7]   
[1, 2, 5]   
[2, 6]   
[1, 1, 6]

这道题跟之前那道[Combination Sum 组合之和](http://www.cnblogs.com/grandyang/p/4419259.html) 本质没有区别，只需要改动一点点即可，上道题给定数组中的数字可以重复使用，而这道题不能重复使用，只需要在之前的基础上修改两个地方即可，首先在递归的for循环里加上if (i > start && num[i] == num[i - 1]) continue; 这样可以防止res中出现重复项，然后就在递归调用combinationSum2DFS里面的参数换成i+1，这样就不会重复使用数组中的数字了，代码如下：

class Solution {

public:

vector<vector<int> > combinationSum2(vector<int> &num, int target) {

vector<vector<int> > res;

vector<int> out;

sort(num.begin(), num.end());

combinationSum2DFS(num, target, 0, out, res);

return res;

}

void combinationSum2DFS(vector<int> &num, int target, int start, vector<int> &out, vector<vector<int> > &res) {

if (target < 0) return;

else if (target == 0) res.push\_back(out);

else {

for (int i = start; i < num.size(); ++i) {

if (i > start && num[i] == num[i - 1]) continue;

out.push\_back(num[i]);

combinationSum2DFS(num, target - num[i], i + 1, out, res);

out.pop\_back();

}

}

}

};

20.首个缺失的正数

Given an unsorted integer array, find the first missing positive integer.

For example,  
Given [1,2,0] return 3,  
and [3,4,-1,1] return 2.

Your algorithm should run in O(n) time and uses constant space.

这道题让我们找缺失的首个正数，由于限定了O(n)的时间，所以一般的排序方法都不能用，最开始我没有看到还限制了空间复杂度，所以想到了用哈希表来解，这个思路很简单，第一遍遍历数组把所有的数都存入哈希表中，并且找出数组的最大值，下次循环从1开始递增找数字，哪个数字找不到就返回哪个数字，如果一直找到了最大的数字，则返回最大值+1，代码如下：

// NOT constant space

class Solution {

public:

int firstMissingPositive(int A[], int n) {

if (n <= 0) return 1;

unordered\_map<int, int> m;

int mx = A[0];

for (int i = 0; i < n; ++i) {

if (A[i] > 0) {

m[A[i]] = 1;

mx = max(mx, A[i]);

}

}

for (int i = 1; i <= mx; ++i) {

if (m.find(i) == m.end()) return i;

}

return mx + 1;

}

};

但是上面的解法不是O(1)的空间复杂度，所以我们需要另想一种解法，既然不能建立新的数组，那么我们只能覆盖原有数组，我们的思路是把1放在数组第一个位置A[0]，2放在第二个位置A[1]，即需要把A[i]放在A[A[i] - 1]上，那么我们遍历整个数组，如果A[i] != i + 1, 而A[i]为整数且不大于n，另外A[i]不等于A[A[i] - 1]的话，我们将两者位置调换，如果不满足上述条件直接跳过，最后我们再遍历一遍数组，如果对应位置上的数不正确则返回正确的数，代码如下：

 class Solution {

public:

int firstMissingPositive(int A[], int n) {

int i = 0;

while (i < n) {

if (A[i] != i + 1 && A[i] > 0 && A[i] <= n && A[i] != A[A[i] - 1]) {

swap(A[i], A[A[i] - 1]);

} else {

++ i;

}

}

for (i = 0; i < n; ++i) {

if (A[i]>0&&A[i] != i + 1) return i + 1;

}

return n + 1;

}

};

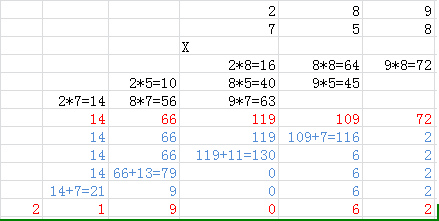
21.字符串相乘

Given two numbers represented as strings, return multiplication of the numbers as a string.

Note: The numbers can be arbitrarily large and are non-negative.

这道题让我们求两个字符串数字的相乘，输入的两个数和返回的数都是以字符串格式储存的，这样做的原因可能是这样可以计算超大数相乘，可以不受int或long的数值范围的约束，那么我们该如何来计算乘法呢，我们小时候都学过多位数的乘法过程，都是每位相乘然后错位相加，那么这里就是用到这种方法，把错位相加后的结果保存到一个一维数组中，然后分别每位上算进位，最后每个数字都变成一位，然后要做的是去除掉首位0，最后把每位上的数字按顺序保存到结果中即可，代码如下：

我们以289\*758为例



首先我们把每一位相乘，得到一个没有进位的临时结果，如图中中间的一行红色数字就是临时结果，然后把临时结果从低位起依次进位。对于一个m位整数乘以n位整数的结果，最多只有m+n位.

class Solution {

public:

    string multiply(string num1, string num2) {

**int** n1 = num1.size(), n2 = num2.size();

        vector<**int**> tmpres(n1+n2, 0);

**int** k = n1 + n2 - 2;

        for(**int** i = 0; i < n1; i++)

            for(**int** j = 0; j < n2; j++)

                tmpres[k-i-j] += (num1[i]-'0')\*(num2[j]-'0');

**int** carryBit = 0;

        for(**int** i = 0; i < n1+n2; i++)//处理进位

        {

            tmpres[i] += carryBit;

            carryBit = tmpres[i] / 10;

            tmpres[i] %= 10;

        }

**int** i = k+1;

        while(tmpres[i] == 0)i--;//如果没有m+n位，去掉乘积的前导0

        if(i < 0)return "0"; //注意乘积为0的特殊情况

        string res;

        for(; i >= 0; i--)

            res.push\_back(tmpres[i] + '0');

        return res;

    }

};

22.1到N个数中的K个数的组合

Given two integers n and k, return all possible combinations of k numbers out of 1 ... n.

For example,  
If n = 4 and k = 2, a solution is:

[

[2,4],

[3,4],

[2,3],

[1,2],

[1,3],

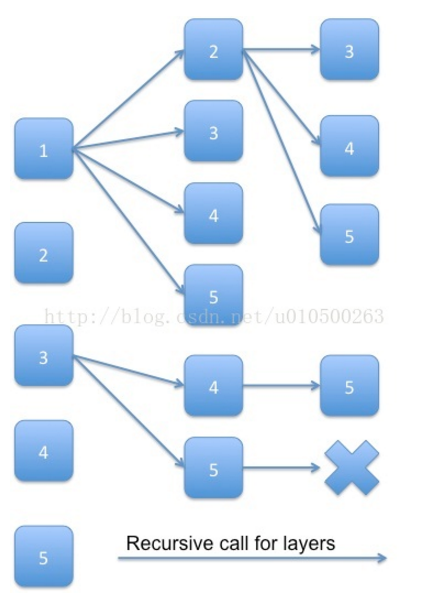
[1,4],

]

这道题让求1到n共n个数字里k个数的组合数的所有情况，还是要用深度优先搜索DFS来解，根据以往的经验，像这种要求出所有结果的集合，一般都是用DFS调用递归来解。那么我们建立一个保存最终结果的大集合res，还要定义一个保存每一个组合的小集合out，每次放一个数到out里，如果out里数个数到了k个，则把out保存到最终结果中，否则在下一层中继续调用递归。网友[u010500263的博客](http://blog.csdn.net/u010500263/article/details/18435495" \t "http://www.cnblogs.com/grandyang/p/_blank)里有一张图很好的说明了递归调用的顺序，请点击[这里](http://blog.csdn.net/u010500263/article/details/18435495" \t "http://www.cnblogs.com/grandyang/p/_blank)。根据上面分析，可写出代码如下：

对于n = 5, k = 3, 处理的结果如下：

1 2 3   
1 2 4  
1 2 5  
1 3 4  
1 3 5  
1 4 5  
2 3 4  
2 3 5  
2 4 5  
3 4 5



class Solution {

public:

vector<vector<int> > combine(int n, int k) {

vector<vector<int> > res;

vector<int> out;

combineDFS(n, k, 1, out, res);

return res;

}

void combineDFS(int n, int k, int level, vector<int> &out, vector<vector<int> > &res) {

if (out.size() == k) res.push\_back(out);

else {

for (int i = level; i <= n; ++i) {//level用来控制不重复

out.push\_back(i);

combineDFS(n, k, i + 1, out, res);

out.pop\_back();

}

}

}

};

1. 数组的全排列

Given a collection of numbers, return all possible permutations.

For example,  
[1,2,3] have the following permutations:  
[1,2,3], [1,3,2], [2,1,3], [2,3,1], [3,1,2], and [3,2,1].

这道题是求全排列问题，给的输入数组没有重复项，这跟之前的那道[Combinations 组合项](http://www.cnblogs.com/grandyang/p/4332522.html) 和类似，解法基本相同，但是不同点在于那道不同的数字顺序只算一种，是一道典型的组合题，而此题是求全排列问题，还是用递归DFS来求解。这里我们需要用到一个visited数组来标记某个数字是否访问过，然后在DFS递归函数从的循环应从头开始，而不是从level开始，这是和[Combinations 组合项](http://www.cnblogs.com/grandyang/p/4332522.html) 不同的地方，其余思路大体相同，代码如下：

class Solution {

public:

vector<vector<int> > permute(vector<int> &num) {

vector<vector<int> > res;

vector<int> out;

vector<int> visited(num.size(), 0);

permuteDFS(num, 0, visited, out, res);

return res;

}

void permuteDFS(vector<int> &num, int level, vector<int> &visited, vector<int> &out, vector<vector<int> > &res) {

if (level == num.size()) res.push\_back(out);

else {

for (int i = 0; i < num.size(); ++i) {

if (visited[i] == 0) {

visited[i] = 1;

out.push\_back(num[i]);

permuteDFS(num, level + 1, visited, out, res);

out.pop\_back();

visited[i] = 0;

}

}

}

}

};

24.顺时针旋转数组90度

You are given an n x n 2D matrix representing an image.

Rotate the image by 90 degrees (clockwise).

Follow up:  
Could you do this in-place?

首先对原数组取其转置矩阵，然后把每行的数字翻转可得到结果，如下所示(其中蓝色数字表示翻转轴)：

1  2  3　　　 　　 1  4  7　　　　　  7  4  1

4  5  6　　-->　　 2  5  8　　 -->  　  8  5  2

7  8  9 　　　 　　3  6  9　　　　      9  6  3

class Solution {

public:

void rotate(vector<vector<int> > &matrix) {

int n = matrix.size();

for (int i = 0; i < n; ++i) {

for (int j = i + 1; j < n; ++j) {

swap(matrix[i][j], matrix[j][i]);

}

reverse(matrix[i].begin(), matrix[i].end());

}

}

};

25.错位词

Given an array of strings, return all groups of strings that are anagrams.

Note: All inputs will be in lower-case.

For example:

Input:　　["tea","and","ate","eat","den"]

Output:   ["tea","ate","eat"]

- 遍历整个字符串集，将当前取出的字符串重新排序，在哈希表中查找排序后的字符串  
- 如果不存在该字符串，则新建一个该字符串和其下标的映射  
- 如果存在该字符串，则看其映射的下标是否大于等于0  
- 如果大于等于0，则将该下标的原字符串存入结果中，并将下标赋为-1，防止下次再存  
- 将当前字符串存入结果

class Solution {

public:

vector<string> anagrams(vector<string> &strs) {

vector<string> res;

unordered\_map<string, int> m;

for (int i = 0; i < strs.size(); ++i) {

string s = strs[i];

sort(s.begin(), s.end());

if (m.find(s) == m.end()) {//不存在该字符串

m[s] = i;

} else {

if (m[s] >= 0) {

res.push\_back(strs[m[s]]);

m[s] = -1;//防止下次再存

}

res.push\_back(strs[i]);

}

}

return res;

}

};

1. 求X的N次方

Implement pow(x, n).

这道题让我们求x的n次方，如果我们只是简单的用个for循环让x乘以自己n次的话，未免也把LeetCode上的想的太简单了，一句话形容图样图森破啊。OJ因超时无法通过，所以我们需要优化我们的算法，使其在更有效的算出结果来。我们可以用递归来折半计算，每次把n缩小一半，这样n最终会缩小到0，任何数的0次方都为1，这时候我们再往回乘，如果此时n是偶数，直接把上次递归得到的值算个平方返回即可，如果是奇数，则还需要乘上个x的值。还有一点需要引起我们的注意的是n有可能为负数，对于n是负数的情况，我们可以先用其绝对值计算出一个结果再取其倒数即可，代码如下：

class Solution {

public:

double myPow(double x, int n) {

if (n < 0) return 1 / power(x, -n);

return power(x, n);

}

double power(double x, int n) {

if (n == 0) return 1;

double half = power(x, n / 2);

if (n % 2 == 0) return half \* half;

return x \* half \* half;

}

};

1. 跳跃游戏

Given an array of non-negative integers, you are initially positioned at the first index of the array.

Each element in the array represents your maximum jump length at that position.

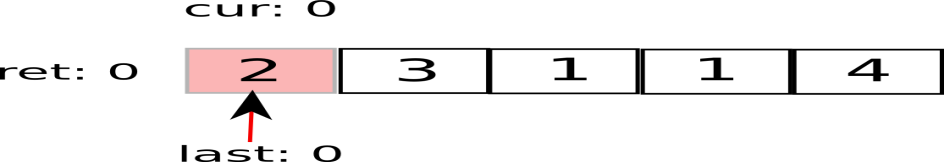
Your goal is to reach the last index in the minimum number of jumps.

For example:  
Given array A = [2,3,1,1,4]

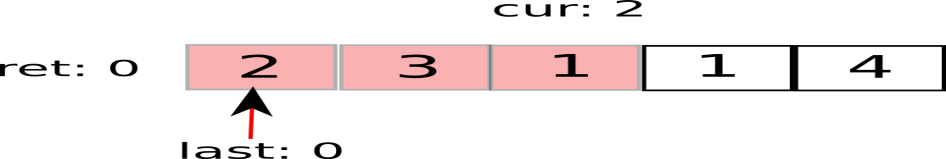
The minimum number of jumps to reach the last index is 2. (Jump 1 step from index 0 to 1, then 3 steps to the last index.)

思想主要是，扫描数组（废话。。。），以确定当前最远能覆盖的节点，放入curr。然后继续扫描，直到当前的路程超过了上一次算出的覆盖范围，那么更新覆盖范围，同时更新条数，因为我们是经过了多一跳才能继续前进的。形象地说，这个是在争取每跳最远的greedy（贪婪算法）

比如就是我们题目中的[2,3,1,1,4]。初始状态是这样的：cur表示最远能覆盖到的地方，用红色表示。last表示已经覆盖的地方，用箭头表示。它们都指在第一个元素上。



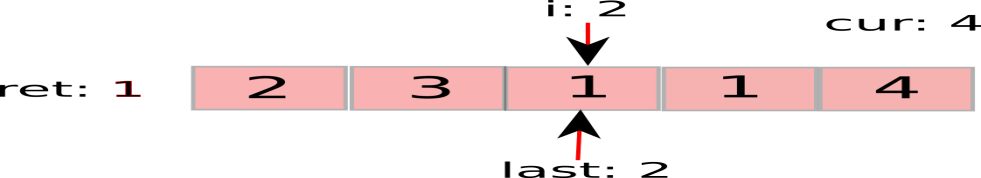
接下来，第一元素告诉cur，最远咱可以走2步。于是：



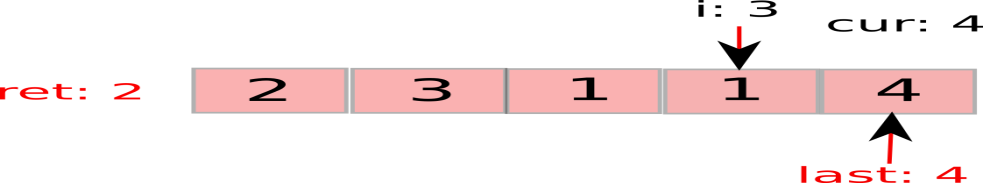
下一循环中，i指向1（图中的元素3），发现，哦，i小于last能到的范围，于是更新last（相当于说，进入了新的势力范围），步数ret加1.同时要更新cur。因为最远距离发现了。



接下来，i继续前进，发现i在当前的势力范围内，无需更新last和步数ret。更新cur。



i继续前进，接下来发现**超过**当前势力范围，更新last和步数。cur已然最大了。



最后，i到最后一个元素。依然在势力范围内，遍历完成，返回ret。



为啥要用last，直接用curr跳不就行了。直接用curr跳那每次都是跳最远的，但是最优路径不不一定是这样

class Solution {

public:

int jump(int A[], int n) {

int res = 0, last = 0, cur = 0;

for (int i = 0; i < n - 1; ++i) {

if (i > last) {

last = cur;

++res;

if (cur >= n - 1) break;

}

cur = max(cur, i+A[i]);

}

return res;

}

};

1. 螺旋打映数组

我们定义p，q为当前环的高度和宽度，当p或者q为1时，表示最后一个环只有一行或者一列，可以跳出循环。此题的难点在于下标的转换，如何正确的转换下标是解此题的关键，我们可以对照着上面的3x3的例子来完成下标的填写，代码如下：

class Solution {

public:

vector<int> spiralOrder(vector<vector<int> > &matrix) {

vector<int> res;

if (matrix.empty() || matrix[0].empty()) return res;

int m = matrix.size(), n = matrix[0].size();

int c = m > n ? (n + 1) / 2 : (m + 1) / 2;

int p = m, q = n;

for (int i = 0; i < c; ++i, p -= 2, q -= 2) {

for (int col = i; col < i + q; ++col)

res.push\_back(matrix[i][col]);

for (int row = i + 1; row < i + p; ++row)

res.push\_back(matrix[row][i + q - 1]);

if (p == 1 || q == 1) break;

for (int col = i + q - 2; col >= i; --col)

res.push\_back(matrix[i + p - 1][col]);

for (int row = i + p - 2; row > i; --row)

res.push\_back(matrix[row][i]);

}

return res;

}

};

29，插入区间，融合区间

Given a set of non-overlapping intervals, insert a new interval into the intervals (merge if necessary).

You may assume that the intervals were initially sorted according to their start times.

**Example 1:**  
Given intervals [1,3],[6,9], insert and merge [2,5] in as [1,5],[6,9].

**Example 2:**  
Given [1,2],[3,5],[6,7],[8,10],[12,16], insert and merge [4,9] in as [1,2],[3,10],[12,16].

This is because the new interval [4,9] overlaps with [3,5],[6,7],[8,10].

这道题让我们在一系列非重叠的区间中插入一个新的区间，可能还需要和原有的区间合并，那么我们需要对给区间集一个一个的遍历比较，那么会有两种情况，重叠或是不重叠，不重叠的情况最好，直接将新区间插入到对应的位置即可，重叠的情况比较复杂，有时候会有多个重叠，我们需要更新新区间的范围以便包含所有重叠，而且最后处理的时候还需要删除原区间集中所有和新区间重叠的区间，然后插入新区间即可。具体思路如下：

- 对区间集中每个区间进行遍历

　　- 如果新区间的末尾小于当前区间的开头，则跳出循环

　　- 如果新区间的开头大于当前区间的末尾，不作处理

　　- 如果新区间和当前区间有重叠，则更新新区间的开头为两者最小值，新区间的末尾为两者最大值，重叠数加一

　　- 指针移向下一个区间

- 如果重叠数大于0，则删除掉所有的重叠区间

- 插入新区间到对应的位置

代码如下：

// Using Index

class Solution {

public:

vector<Interval> insert(vector<Interval> &intervals, Interval newInterval) {

vector<Interval> res = intervals;

int i = 0, overlap = 0, n = res.size();

while (i < n) {

if (newInterval.end < res[i].start) break;

else if (newInterval.start > res[i].end) {}

else {

newInterval.start = min(newInterval.start, res[i].start);

newInterval.end = max(newInterval.end, res[i].end);

++overlap;

}

++i;

}

if (overlap > 0) res.erase(res.begin() + i - overlap, res.begin() + i);

res.insert(res.begin() + i - overlap, newInterval);

return res;

}

};

30.

Given a string s consists of upper/lower-case alphabets and empty space characters ' ', return the length of last word in the string.

If the last word does not exist, return 0.

Note: A word is defined as a character sequence consists of non-space characters only.

For example,   
Given s = "Hello World",  
return 5.

这道题难度不是很大。先对输入字符串做预处理，去掉开头和结尾的空格，然后用一个计数器来累计非空格的字符串的长度，遇到空格则将计数器清零。代码如下：

 class Solution {

public:

int lengthOfLastWord(const char \*s) {

int len = strlen(s);

int left = 0;

int right = len - 1;

int count = 0;

while (s[left] == ' ') ++left;

while (s[right] == ' ') --right;

for (int i = left; i <= right; ++i) {

if (s[i] == ' ') count = 0;

else ++count;

}

return count;

}

};

由于我们只关于最后一个单词的长度，所以开头有多少个空格起始我们并不在意，我们从字符串末尾开始，先将末尾的空格都去掉，然后开始找非空格的字符的长度即可，参见代码如下：

class Solution {

public:

int lengthOfLastWord(string s) {

int right = s.size() - 1, res = 0;

while (right >= 0 && s[right] == ' ') --right;

while (right >= 0 && s[right] != ' ' ) {

--right;

++res;

}

return res;

}

};

31.矩阵中斜对角之间的不同路径

A robot is located at the top-left corner of a m x n grid (marked 'Start' in the diagram below).

The robot can only move either down or right at any point in time. The robot is trying to reach the bottom-right corner of the grid (marked 'Finish' in the diagram below).

How many possible unique paths are there?

我们需要用动态规划Dynamic Programming来解，我们可以维护一个二维数组dp，其中dp[i][j]表示到当前位置不同的走法的个数，然后可以得到递推式为: dp[i][j] = dp[i - 1][j] + dp[i][j - 1]，这里为了节省空间，我们使用一维数组dp，一行一行的刷新也可以，代码如下：

// DP

class Solution {

public:

int uniquePaths(int m, int n) {

vector<int> dp(n, 1);

for (int i = 1; i < m; ++i) {

for (int j = 1; j < n; ++j) {

dp[j] += dp[j - 1];

}

}

return dp[n - 1];

}

};

这道题其实还有另一种很数学的解法，参见网友[Code Ganker的博客](http://blog.csdn.net/linhuanmars/article/details/22126357" \t "http://www.cnblogs.com/grandyang/p/_blank)，实际相当于机器人总共走了m + n - 2步，其中m - 1步向下走，n - 1步向右走，那么总共不同的方法个数就相当于在步数里面m - 1和n - 1中较小的那个数的取法，实际上是一道组合数的问题，写出代码如下:

class Solution {

public:

int uniquePaths(int m, int n) {

double num = 1, denom = 1;

int small = m > n ? n : m;

for (int i = 1; i <= small - 1; ++i) {

num \*= m + n - 1 - i;

denom \*= i;

}

return (int)(num / denom);

}

};

矩阵中有障碍物，求不同路径

Follow up for "Unique Paths":

Now consider if some obstacles are added to the grids. How many unique paths would there be?

An obstacle and empty space is marked as 1 and 0 respectively in the grid.

For example,

There is one obstacle in the middle of a 3x3 grid as illustrated below.

[

[0,0,0],

[0,1,0],

[0,0,0]

]

The total number of unique paths is 2.

Note: m and n will be at most 100.

这道题是之前那道[Unique Paths 不同的路径](http://www.cnblogs.com/grandyang/p/4353555.html) 的延伸，在路径中加了一些障碍物，还是用动态规划Dynamic Programming来解，不同的是当遇到为1的点，将该位置的dp数组中的值清零，其余和之前那道题并没有什么区别，代码如下：

class Solution {

public:

int uniquePathsWithObstacles(vector<vector<int>>& obstacleGrid) {

if (obstacleGrid.empty() || obstacleGrid[0].empty() || obstacleGrid[0][0] == 1) return 0;

vector<vector<int> > dp(obstacleGrid.size(), vector<int>(obstacleGrid[0].size(), 0));

for (int i = 0; i < obstacleGrid.size(); ++i) {

for (int j = 0; j < obstacleGrid[i].size(); ++j) {

if (obstacleGrid[i][j] == 1) dp[i][j] = 0;

else if (i == 0 && j == 0) dp[i][j] = 1;

else if (i == 0 && j > 0) dp[i][j] = dp[i][j - 1];

else if (i > 0 && j == 0) dp[i][j] = dp[i - 1][j];

else dp[i][j] = dp[i - 1][j] + dp[i][j - 1];

}

}

return dp.back().back();

}

};

或者我们也可以使用一维dp数组来解，省一些空间，参见代码如下：

// DP

class Solution {

public:

int uniquePathsWithObstacles(vector<vector<int> > &obstacleGrid) {

if (obstacleGrid.empty() || obstacleGrid[0].empty()) return 0;

int m = obstacleGrid.size(), n = obstacleGrid[0].size();

if (obstacleGrid[0][0] == 1) return 0;

vector<int> dp(n, 0);

dp[0] = 1;

for (int i = 0; i < m; ++i) {

for (int j = 0; j < n; ++j) {

if (obstacleGrid[i][j] == 1) dp[j] = 0;

else if (j > 0) dp[j] += dp[j - 1];

}

}

return dp[n - 1];

}

};

最小路径和

Given a m x 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.

这道题跟之前那道[Dungeon Game 地牢游戏](http://www.cnblogs.com/grandyang/p/4233035.html) 没有什么太大的区别，都需要用动态规划Dynamic Programming来做，这应该算是DP问题中比较简单的一类，我们维护一个二维的dp数组，其中dp[i][j]表示当前位置的最小路径和，递推式也容易写出来 dp[i][j] = grid[i][j] + min(dp[i - 1][j], 反正难度不算大，代码如下：

class Solution {

public:

int minPathSum(vector<vector<int> > &grid) {

int m = grid.size(), n = grid[0].size();

int dp[m][n];

dp[0][0] = grid[0][0];

for (int i = 1; i < m; ++i) dp[i][0] = grid[i][0] + dp[i - 1][0];

for (int i = 1; i < n; ++i) dp[0][i] = grid[0][i] + dp[0][i - 1];

for (int i = 1; i < m; ++i) {

for (int j = 1; j < n; ++j) {

dp[i][j] = grid[i][j] + min(dp[i - 1][j], dp[i][j - 1]);

}

}

return dp[m - 1][n - 1];

}

};

验证数字

Validate if a given string is numeric.

Some examples:  
"0" => true  
" 0.1 " => true  
"abc" => false  
"1 a" => false  
"2e10" => true

Note: It is intended for the problem statement to be ambiguous. You should gather all requirements up front before implementing one.

所有的字符可以分为六大类，空格，符号，数字，小数点，自然底数和其他字符，我们需要五个标志变量，num, dot, exp, sign分别表示数字，小数点，自然底数和符号是否出现，numAfterE表示自然底数后面是否有数字，那么我们分别来看各种情况：

- 空格： 我们需要排除的情况是，当前位置是空格而后面一位不为空格，但是之前有数字，小数点，自然底数或者符号出现时返回false。

- 符号：符号前面如果有字符的话必须是空格或者是自然底数，标记sign为true。

- 数字：标记num和numAfterE为true。

- 小数点：如果之前出现过小数点或者自然底数，返回false，否则标记dot为true。

- 自然底数：如果之前出现过自然底数或者之前从未出现过数字，返回false，否则标记exp为true，numAfterE为false。

- 其他字符：返回false。

最后返回num && numAfterE即可。

class Solution {

public:

bool isNumber(string s) {

bool num = false, numAfterE = true, dot = false, exp = false, sign = false;

int n = s.size();

for (int i = 0; i < n; ++i) {

if (s[i] == ' ') {

if (i < n - 1 && s[i + 1] != ' ' && (num || dot || exp || sign)) return false;

} else if (s[i] == '+' || s[i] == '-') {

if (i > 0 && s[i - 1] != 'e' && s[i - 1] != ' ') return false;

sign = true;

} else if (s[i] >= '0' && s[i] <= '9') {

num = true;

numAfterE = true;

} else if (s[i] == '.') {

if (dot || exp) return false;

dot = true;

} else if (s[i] == 'e') {

if (exp || !num) return false;

exp = true;

numAfterE = false;

} else return false;

}

return num && numAfterE;

}

};

加1运算：

Given a non-negative number represented as an array of digits, plus one to the number.

The digits are stored such that the most significant digit is at the head of the list.

将一个数字的每个位上的数字分别存到一个一维向量中，最高位在最开头，我们需要给这个数字加一，即在末尾数字加一，如果末尾数字是9，那么则会有进位问题，而如果前面位上的数字仍为9，则需要继续向前进位。具体算法如下：首先判断最后一位是否为9，若不是，直接加一返回，若是，则该位赋0，再继续查前一位，同样的方法，知道查完第一位。如果第一位原本为9，加一后会产生新的一位，那么最后要做的是，查运算完的第一位是否为0，如果是，则在最前头加一个1。代码如下：

class Solution {

public:

vector<int> plusOne(vector<int> &digits) {

int n = digits.size();

for (int i = n - 1; i >= 0; --i) {

if (digits[i] == 9) digits[i] = 0;

else {

digits[i] += 1;

return digits;

}

}

if (digits.front() == 0) digits.insert(digits.begin(), 1);

return digits;

}

};

二进制数相加

Given two binary strings, return their sum (also a binary string).

For example,  
a = "11"  
b = "1"  
Return "100".

用两个指针分别指向a和b的末尾，然后每次取出一个字符，转为数字，若无法取出字符则按0处理，然后定义进位carry，初始化为0，将三者加起来，对2取余即为当前位的数字，对2取商即为当前进位的值，记得最后还要判断下carry，如果为1的话，要在结果最前面加上一个1，参见代码如下：

class Solution {

public:

string addBinary(string a, string b) {

string res = "";

int m = a.size() - 1, n = b.size() - 1, carry = 0;

while (m >= 0 || n >= 0) {

int p = m >= 0 ? a[m--] - '0' : 0;

int q = n >= 0 ? b[n--] - '0' : 0;

int sum = p + q + carry;

res = to\_string(sum % 2) + res;

carry = sum / 2;

}

return carry == 1 ? "1" + res : res;

}

};

求平方根

Implement int sqrt(int x).

Compute and return the square root of x.

这道题要求我们求平方根，我们能想到的方法就是算一个候选值的平方，然后和x比较大小，为了缩短查找时间，我们采用二分搜索法来找平方根，由于求平方的结果会很大，可能会超过int的取值范围，所以我们都用long long来定义变量，这样就不会越界，代码如下：

// Binary Search

class Solution {

public:

int sqrt(int x) {

long long left = 0, right = (x / 2) + 1;

while (left <= right) {

long long mid = (left + right) / 2;

long long sq = mid \* mid;

if (sq == x) return mid;

else if (sq < x) left = mid + 1;

else right = mid - 1;

}

return right;

}

};