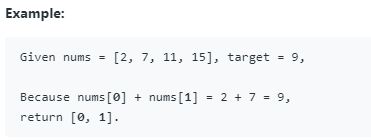
①想问题的时候首先从最简单的方法开始，如判断一个数字是否是回文数（第9题），应该想办法直接获得后面数字的回文数，比中间分割要精简且易懂；

②注意了！！！如果使用了STL，那么x.size()的类型是unsigned int，如果与int作比较，会将int升级为unsigned int，会导致-1>=0的结果出现，因为-1升级变成了1，因此要将x.size()强制类型转换为int；

③是的，我们函数可以传值也可以传引，其实指针本身也是传值或者传引的。例如，func（int \*p），其实传入的是p的副本，所以你在func内将p=nullptr，返回之后判断p==nullptr为false；如果要同步修改，需要传入指针的引用：func（int \*&p），此时返回之后判断p==nullptr为true；或者func（int \*\*p），调用的时候传入地址func（&p），这样的话，在func里面就要当成指针的指针来使用了，比较麻烦；

第一题：

给定一个数组nums和一个target，返回nums里面两个加起来等于target的元素的索引。



思路很简单，但是使用find会很慢，所以这里使用map。思路是检查每一个target-nums[i]，如果map里面有这个数字，则返回两个索引；否则把这个元素入map，检查下一个。map的搜索时间为O（1），最坏情况是最后两个数字是结果，所以时间复杂度为O（n）。

代码：

class Solution {

public:

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

map<int,int> temp;

vector<int> result;

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

{

int remain=target-nums[i];

if(temp.count(remain)==0)

temp[nums[i]]=i;

else

{

result.push\_back(temp[remain]);

result.push\_back(i);

return result;

}

}

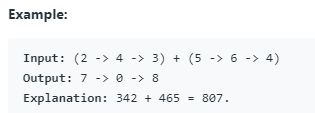
return result;

}

};

第二题：

给定两个链表表示两个整数，注意这个整数是倒序的，返回一个链表表示他们的和



思路很简单，就是加法模拟。首先这里利用一个dummy作为答案的链表头，这样操作会方便很多。首先，当l1和l2都不空的时候，用一个up表示进位，然后计算temp=up+l1+l2，其结果的个位数就是这个位的答案，然后更新up，以此类推。最后检查l1或者l2不空的时候按照同样方法计算，这里的计算还是有可能有多次进位，比如1+99；最后检查up，如果不为0则插入，否则返回。遍历两个链表一次，所以时间复杂度为O（n）。

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) {

int up=0;

ListNode \*res=new ListNode(-1),\*w=res;

while(l1&&l2){

int temp=up+l1->val+l2->val;

ListNode \*t=new ListNode(temp%10);

w->next=t,w=w->next;

up=temp/10;

l1=l1->next,l2=l2->next;

}

while(l1){

int temp=up+l1->val;

ListNode \*t=new ListNode(temp%10);

w->next=t,w=w->next;

up=temp/10;

l1=l1->next;

}

while(l2){

int temp=up+l2->val;

ListNode \*t=new ListNode(temp%10);

w->next=t,w=w->next;

up=temp/10;

l2=l2->next;

}

if(up){

ListNode \*t=new ListNode(up);

w->next=t;

}

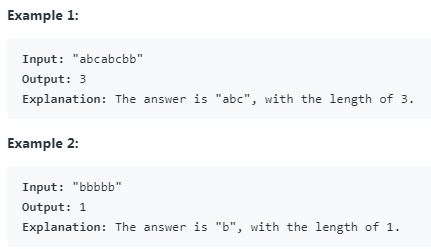
return res->next;

}

};

第三题：

给定一个字符串s，返回其不含重复字符的最长子串的长度。



思路比较独特。利用两个指针i，j来表示子串[i，j]，利用一个哈希表来记录该字符已经出现的次数。首先wh[s[j]]++，表示该字符出现次数+1，然后检查j这个字符是不是重复字符，然后不断将i往前移，直到wh[s[j]]==1，此时子串[i，j]里面的字符又是不重复的了；每次都更新res的值。哈希表查找时间O（1），然后最坏情况i，j都遍历了一遍字符串，所以时间复杂度为O（n）。

代码：

class Solution {

public:

int lengthOfLongestSubstring(string s) {

unordered\_map<char,int> wh;

int res=0;

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

wh[s[j]]++;

while(wh[s[j]]>1)

wh[s[i++]]--;

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

}

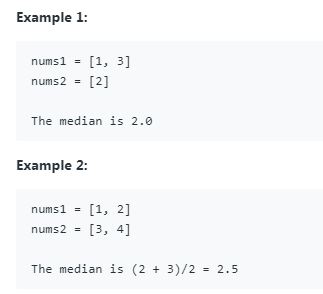
return res;

}

};

第四题：

**这题重要。**给定两个有序数组nums1和nums2，返回这两个数组整体来说的中位数，如果总长度为偶数，则返回中间两个数的平均数。



这题的思路很难想，但是这个方法可以用于寻找这其中的第k小的数！首先假设nums1的长度为m，nums2的长度为n，要寻找第k小的数，如果m，n均大于k/2，则比较nums1[k/2-1]和nums2[k/2-1]，如果前者大，说明nums2中前k/2个数绝对不是第k大的数，因为即使是最优情况下，nums2[k/2-1]也只是第k-1小的数，但是有可能第k大的数在nums2的后面，所以这里只去掉nums2的前k/2个数，在剩余的数里面寻找第k-k/2小的数。如果后者大思路相同；然后如果m的长度小于k/2，则直接比较nums1[m-1]和nums2[k/2-1]，不用担心越界，因为m+n大于等于k。如果前者小，那么答案肯定是nums2[k-m-1]；如果前者大，那就去掉后者前k/2个数然后继续。注意下面代码中，findKthNumber里面的i和j分别表示nums在这个递归的范围为[i,m]和[j,n]。因为每次都去掉k/2个不符合题意的解，所以每次都缩小问题一半的规模，所以时间复杂度为O（log（m+n））。

代码：

class Solution {

public:

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

int total = nums1.size() + nums2.size();

if (total % 2 == 0)

{

int left = findKthNumber(nums1, 0, nums2, 0, total / 2);

int right = findKthNumber(nums1, 0, nums2, 0, total / 2 + 1);

return (left + right) / 2.0;

}

else

{

return findKthNumber(nums1, 0, nums2, 0, total / 2 + 1);

}

}

int findKthNumber(vector<int> &nums1, int i, vector<int> &nums2, int j, int k)

{

if (nums1.size() - i > nums2.size() - j) return findKthNumber(nums2, j, nums1, i, k);

if (nums1.size() == i) return nums2[j + k - 1];

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

int si = min(i + k / 2, int(nums1.size())), sj = j + k / 2;

if (nums1[si - 1] > nums2[sj - 1])

{

return findKthNumber(nums1, i, nums2, j + k / 2, k - k / 2);

}

else

{

return findKthNumber(nums1, si, nums2, j, k - (si - i));

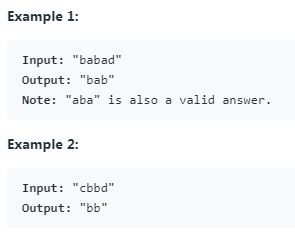
}

}

};

第五题：

给定一个字符串s，返回其最长的回文子串。



这题的思路很简单，只是有些地方需要注意，对于查找回文子串，最厉害的方法应该是Manacher（马拉车）算法，时间复杂度为O（n）。依次以i为中心然后两边往外扩，如果两边的值一样，就更新答案str，否则就到达边界，就是下面j=0的那个循环，这里的j表示回文子串一边的长度；还有一种可能是baab这种，所以首先检查i，i+1是否一样，一样就更新答案，然后往外扩，下面j=i，k=i+1就是检查这种情况，注意不要越界！因为两重循环，所以时间复杂度为O（n^2）。

代码：

class Solution {

public:

string longestPalindrome(string s) {

int res = 0;

string str;

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

{

for (int j = 0; i - j >= 0 && i + j < s.size(); j ++ )

if (s[i - j] == s[i + j])

{

if (j \* 2 + 1 > res)

{

res = j \* 2 + 1;

str = s.substr(i - j, j \* 2 + 1);

}

}

else break;

for (int j = i, k = i + 1; j >= 0 && k < s.size(); j -- , k ++ )

if (s[j] == s[k])

{

if (k - j + 1 > res)

{

res = k - j + 1;

str = s.substr(j, k - j + 1);

}

}

else break;

}

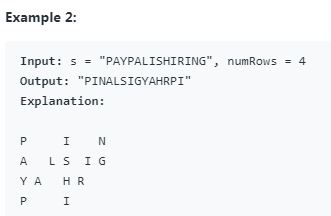
return str;

}

};

第六题：

给定一个字符串s和一个行数numRows，将给定的s排成numRows行，N字形读取的形状，然后返回此时每一行依次读取的字符串。



思路就是找到规律。可以发现，上图中RAYPA为一组、ISHIRI为一组、NG为一组；每一组的元素个数为2\*（numRows-1）个，可以发现，每一组的第一行和最后一行都是只有一个元素，其他各行有两个元素；因此我们可以按行遍历，间隔为2\*（numRows-1），如果是第一行或者最后一行，就直接每一组放入一个元素；如果是中间行，可以发现，当A下标是1，L的下标就是5，也就是2\*（numRows-1）-2\*i，i为A的下标；这样就可以先将第一行入队，然后第二行，以此类推直到最后一行，期间注意判断不要越界。遍历整个s一次，所以时间复杂度为O（n）。

代码：

class Solution {

public:

string convert(string s, int numRows) {

int judge=2\*numRows-2;

if(numRows==1)

return s;

string ans;

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

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

if(i==0||i==numRows-1)

ans+=s[j];

else{

ans+=s[j];

if(j+judge-2\*i<s.size())

ans+=s[j+judge-2\*i];

}

}

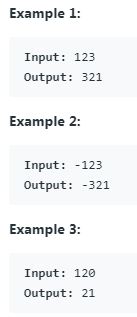
return ans;

}

};

第七题：

给定一个32位有符号整数x，将其转置返回，注意如果转置之后越界了，就返回0。



思路很简单。先初始化一个longlong变量res，然后每次res\*10然后加上x的最后一位的数字，x去掉最后一位数字。这里的x如果是负数，那么每一次%10得到的都是负数，所以最终结果也是负数，所以不用判断是正数还是负数。以此类推，最后判断是否越界，越界返回0，否则返回res。遍历x所有位数一次，所以时间复杂度为O（n）。

代码：

class Solution {

public:

int reverse(int x) {

long long res = 0;

while(x) {

res = res\*10 + x%10;

x /= 10;

}

return (res<INT\_MIN || res>INT\_MAX) ? 0 : res;

}

};

第八题：

自己实现atoi这个函数，这个函数将string转换为int，先找到第一个非空字符，检查如果是‘+’或‘-’就表示正负，不存在这俩符号之一的就表示正。然后后面的所有数字表示这个数，如果遇到非数字字符就表示后面的全部舍去。如果没有数字就返回0。



思路很简单。先去掉所有的空格，然后检查第一个非空字符是否为‘+’或者‘-’，如果是+则往前一步，如果是-将flag置0并往前一步。然后不断前进直到遇到一个非数字字符。最后返回的时候判断范围，越界就修正，没有越界就直接返回。最坏情况下遍历整个s一遍，所以时间复杂度为O（n）。

代码：

class Solution {

public:

int myAtoi(string str) {

if(str.size())

{

long long result=0,i=0,max=str.size();

int flag=1;

while(str[i]==' '&&i<max)i++;

if (str[i]=='-'&&i<max)

{

flag=0;

++i;

}

else if(str[i]=='+'&&i<max)

++i;

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

{

result=result\*10+int(str[i++]-'0');

if(result>=INT\_MAX)

break;

}

if(flag)

return result<INT\_MAX?result:INT\_MAX;

else

return (-result)>INT\_MIN?(-result):INT\_MIN;

}

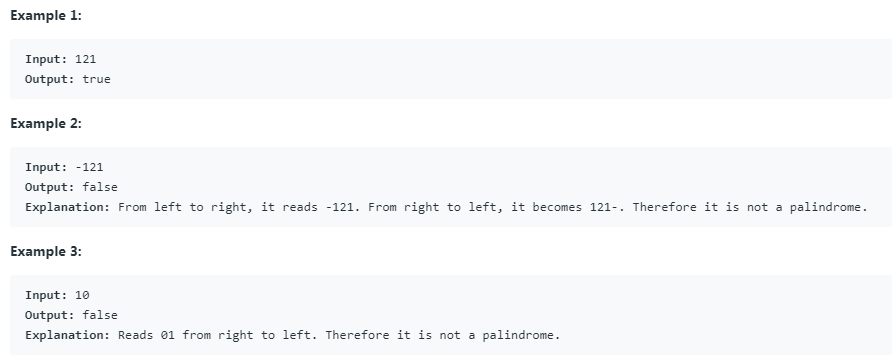
return 0;

}

};

第九题：

给定一个整数x，判断是否是一个回文数。



思路很简单。可以转换为string再判断，也可以直接reverse然后判断相等，但是可能会越界，所以最好的做法应该是从最后面开始取数，取到一半然后判断是否跟前一半相等。如果是负数或者最后一位是0就直接是false。而且要注意如果长度为奇数，那么最中间的数字是无所谓的，所以要注意true的判断。因为最大的有符号int数在十进制下最大是21亿多，也就是说10位，最多判定5次。所以时间复杂度是O（1）。

代码：

class Solution {

public:

bool isPalindrome(int x) {

if (x < 0 || x && x % 10 == 0) return false;

int s = 0;

while (s <= x)

{

s = s \* 10 + x % 10;

if (s == x || s == x / 10) return true; // 分别处理整数长度是奇数或者偶数的情况

x /= 10;

}

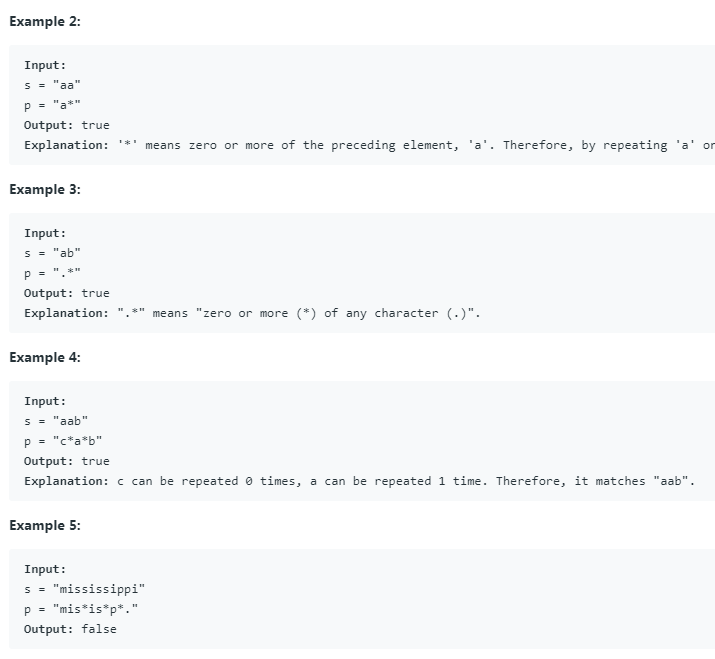
return false;

}

};

第十题：

**这题重要。**给定一个字符串s和一个带有‘\*’和‘.’的字符串p，‘.’可以代表任意一个字符，‘\*’可以表示0-无穷个其前面字符，判断s和p是否匹配。



字符串的问题可以多想想动态规划，这里要比较s和p，不妨分解为很多小问题：当s的长度为i时，p的长度为j，是否匹配？很明显，如果s[i]==p[j]的时候，只要s的0~i-1字符都匹配与p的0~j-1的字符，那么s的0~i的字符就会匹配于p的0~j的字符。所以我们设dp（i，j）表示s的0~i的子串与p的0~j的子串匹配，这里的i和j表示长度；首先，当两个子串都长度均为0的时候为true；并且我们要计算当s的长度为0的时候，p跟s的匹配情况。所以枚举的范围是s为0~s.length，p为1~p.length，p==0肯定是false，没有意义。当i>0,如果s[i-1]==p[j-1]||p[j-1]==’.’，那么dp（i，j）=dp（i-1，j-1）。当p[j-1]==’\*’，首先可以把这个字符当成0个字符，所以dp（i，j）=dp（i，j-2）；然后如果i>0，当\*表示的字符与s[i-1]相同，可以表示1个这个字符，即dp（i，j）=dp（i-1，j）；这里不用在意表示多个字符的情况，因为前一次循环计算过了，所以只需要表示为1个字符就好。动态规划要填一个mn的表，所以时间复杂度为O（mn）。

代码（动态规划）：

class Solution {

public:

bool isMatch(string s, string p) {

int slen=s.length(),plen=p.length();

vector<vector<bool>> dp(slen+1,vector<bool>(plen+1,false));

dp[0][0]=true;

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

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

if(i>0&&(s[i-1]==p[j-1]||p[j-1]=='.'))

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

else if(j>1&&p[j-1]=='\*'){

dp[i][j]=dp[i][j-2];

if(i>0&&(s[i-1]==p[j-2]||p[j-2]=='.'))

dp[i][j]=dp[i][j]|dp[i-1][j];

}

}

return dp[slen][plen];

}

};

代码（递归调用栈）：

class Solution {

public:

bool isMatch(string s, string p) {

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

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

else 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);

}

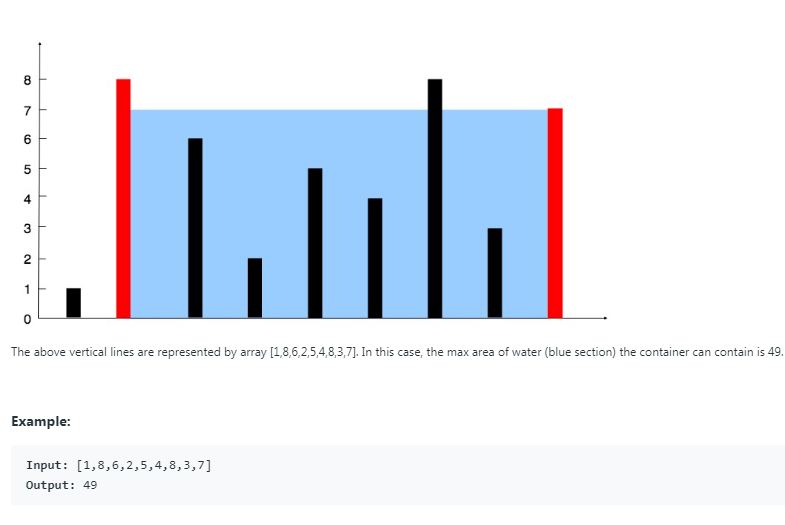
return isMatch(s,p.substr(2));

}

};

第十一题：

给定一个数组，每个元素代表此位置木板高度，找出最多能装多少水。



两个指针从两头开始，每走一步记录数值与max比较，每一次两个木板中短的一方往中间走一步。即使短的木板要作为装水的短边，那当时的底部l也比现在短，所以以后不会用得上这个边，所以去掉短的边是可行的。要检查整个数组的所有元素，所以时间复杂度为O（n）。

代码：

class Solution {

public:

int maxArea(vector<int>& height) {

int st=0,en=height.size()-1;

int max=(en-st)\*min(height[st],height[en]);

while((en-st)>=1)

{

if(height[st]>=height[en])

en--;

else

st++;

int temp=(en-st)\*min(height[st],height[en]);

max=max>temp?max:temp;

}

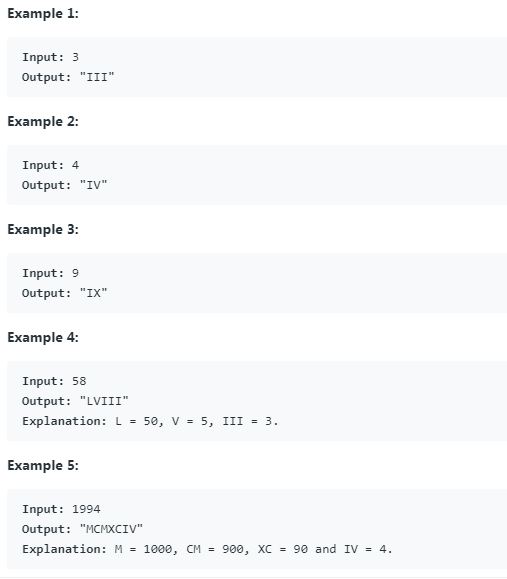
return max;

}

};

第十二题：

阿拉伯数字转换为罗马数字。



思路是模拟。可以知道，罗马数字从大到小排列，比如1994，最前面是1000，然后是900，以此类推。所以可以先把所有罗马数字表示出来，然后不断减小x的同时添加罗马字符。最长的时候输入8要输出VIII四个字符，所以时间复杂度为O（logn）。

代码：

class Solution {

public:

string intToRoman(int num) {

int values[] = {1000, 900, 500, 400, 100, 90, 50, 40, 10, 9, 5, 4, 1};

string reps[] = {"M", "CM", "D", "CD", "C", "XC", "L", "XL", "X", "IX", "V", "IV", "I"};

string res;

for (int i = 0; i < 13; i ++ )

while(num >= values[i])

{

num -= values[i];

res += reps[i];

}

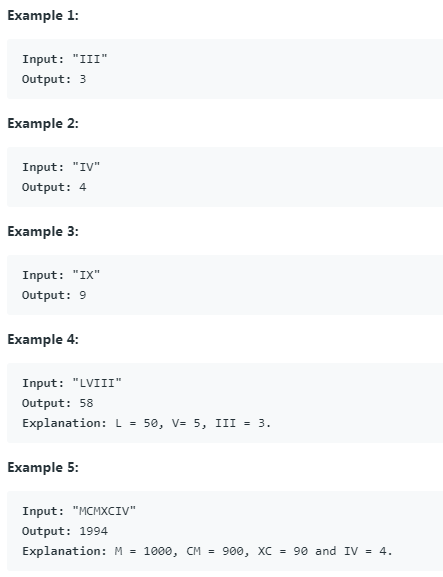
return res;

}

};

第十三题：

将罗马数字转换成阿拉伯数字。



思路跟刚才类似。首先使用map记录每个符号的数值，然后遍历整个s，当未到尾部的时候检查当前i和i+1位代表字符的大小，如果i+1大，说明要相减，同时i++；如果i大，说明就是单独一个数字加进去就行了。遍历整个s一遍，所以时间复杂度为O（n）。

代码：

class Solution {

public:

int romanToInt(string s) {

int result=0;

map<char,int> transfer={{'I',1},{'V',5},{'X',10},{'L',50},{'C',100},{'D',500},{'M',1000}};

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

if(i!=s.length()-1&&transfer[s[i]]<transfer[s[i+1]]){

result+=transfer[s[i+1]]-transfer[s[i]];

++i;

}

else

result+=transfer[s[i]];

}

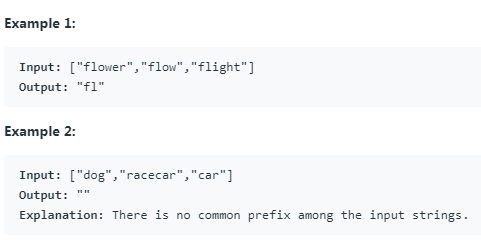
return result;

}

};

第十四题：

给定一个字符串数组，找出字符串数组的最大相同前缀。



思路很简单。首先初始化temp=strs[0]，然后从1开始查找temp和strs[1]的最长公共子串，然后用这个子串再去寻找和strs[2]的最长公共子串，以此类推。最好的做法是找到最短的那个strs[i]，然后以其为temp来进行查找，将其余的元素遍历一遍，所以时间复杂度为O（n\*最短子串的长度）。

代码：

class Solution {

public:

string longestCommonPrefix(vector<string>& strs) {

if(strs.size()==0)return "";

string temp=strs[0];

for(int i=1;i<strs.size()&&temp.size()>0;++i)

{

if(strs[i].find(temp)==0)

continue;

else

{

int j=0;

while(strs[i][j]==temp[j]&&j<strs[i].size()&&j<temp.size())

j++;

temp=temp.erase(j);

}

}

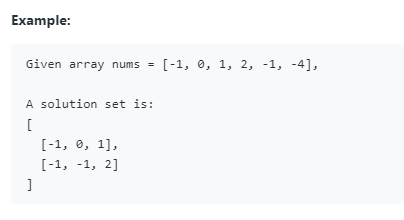
return temp;

}

};

第十五题：

给定一个数组nums，在数组中找出三个数之和为0，输出所有情况，答案不得有重复。



思路是首先将数组排序，固定第一个数i为0到size-2，然后在j=i+1到k=size-1中寻找其余两个数，定义temp=0-nums[i]，当nums[j]+nums[k]!=temp时不断缩小范围。当找到答案时保存答案，并去掉j后面和k前面重复的数字，继续循环。遍历完成后注意去掉i前面重复的数字，然后继续遍历。i遍历整个数组，每一次遍历i到数组尾之间的元素，所以时间复杂度为O（n^2）。

代码：

class Solution {

public:

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

vector<vector<int>> result;

if(nums.size()<3)return result;

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

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

{

int front=i+1,rear=nums.size()-1,temp=-nums[i];

while(front<rear)

{

if(nums[front]+nums[rear]>temp)

rear--;

else if(nums[front]+nums[rear]<temp)

front++;

else

{

result.push\_back({nums[i],nums[front],nums[rear]});

while(front<rear&&nums[front]==result.back()[1])

front++;

while(front<rear&&nums[rear]==result.back()[2])

rear--;

}

}

while(i<nums.size()-2&&nums[i]==nums[i+1])

i++;

}

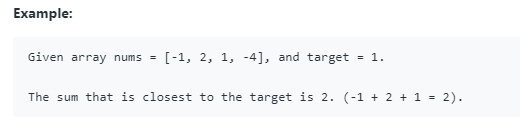
return result;

}

};

第十六题：

给定一个数组nums和一个整数target，返回nums中三数之和最接近target的那个和



找到三个数字与所给target最接近，输出三个数字之和。与十五题类似，首先设置三数之和与target差的绝对值为gap=0x3f3f3f3f，然后固定第一个数字i为0到size-2，遍历i+1到size-1，每次比较gap与当前gap，取较小值，不断缩小范围。每次固定i都要遍历剩余的数组元素，所以总的时间复杂度为O（n^2）。

代码：

class Solution {

public:

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

if(nums.size()<=3)

{

int res=0;

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

res+=nums[i];

return res;

}

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

int gap=0x3f3f3f3f,result;

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

{

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

while(k-j>0)

{

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

return target;

if(abs(nums[i]+nums[j]+nums[k]-target)<gap)

{

gap=abs(nums[i]+nums[j]+nums[k]-target);

result=nums[i]+nums[j]+nums[k];

}

if(nums[i]+nums[j]+nums[k]>target)

k--;

else

j++;

}

}

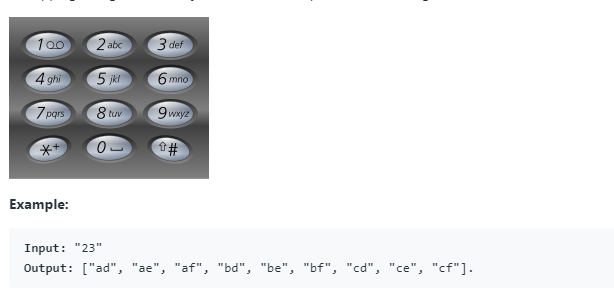
return result;

}

};

第十七题：

给定一个只含有2-9的字符串digits，按照九宫格手机的字母表，返回这个按键顺序能组成的所有字符序列。



思路就是全排列。首先初始化digit记录所有数字对应的字符，然后按顺序遍历digits，每次指定一个位置的字符，当长度符合要求后记录答案，以此类推。数字7和9对应四个字符，其他都是三个，所以最坏情况下一个数字要遍历四次，那么给定的digits长度为l，那么时间复杂度为O（4^l）。

代码：

class Solution {

public:

vector<vector<string>> digit={

{},

{},

{"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"}

};

vector<string> res;

void solve(string digits, int d, string cur) {

if (d == digits.length()) {

res.push\_back(cur);

return;

}

int cur\_num = digits[d] - '0';

for (int i = 0; i < digit[cur\_num].size(); i++)

solve(digits, d + 1, cur + digit[cur\_num][i]);

}

vector<string> letterCombinations(string digits) {

if (digits == "")

return res;

solve(digits, 0, "");

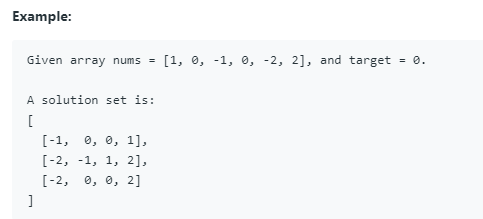
return res;

}

};

第十八题：

给定一个数组nums和一个整数target，找到四个数字之和为0，返回所有这样的四个元素，要求不得有重复。



查找数列中是否有4个数之和为target，返回所有不重复结果。与十五题类似，两层循环，先进行排序。i 从0到size-4，j从i+1到size-3，然后令temp=target-nums[i]-nums[j],令k=j+1，l=size-1，当k<l时，如果num[k]+num[l]==temp,则为结果，记录后去掉j和l的重复数字，否则当num[k]+num[l]>temp时l--,当num[k]+num[l]<temp，k++。在移动两个固定数字的时候应该先去掉重复的元素，保证固定的两个数字会不同。观看别人答案发现，进入循环前可以先判断固定的数字和最小的两个数如果大于target或者固定的数字与最大的两个数字之和小于target直接continue。因为固定的两个数字都要遍历一次数组，然后固定完了还要遍历一次，所以总的时间复杂度为O（n^3）。

代码：

class Solution {

public:

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

vector<vector<int>> res;

if(nums.size()<4)return res;

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

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

{

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

{

int k=j+1,l=nums.size()-1,temp=target-nums[i]-nums[j];

if(nums[i]+nums[j]+nums[k]+nums[k+1]>target||nums[i]+nums[j]+nums[l-1]+nums[l]<target)

continue;

while(k<l)

{

if(nums[k]+nums[l]==temp)

{

vector<int> ans;

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

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

ans.push\_back(nums[k]);

ans.push\_back(nums[l]);

res.push\_back(ans);

while(nums[++k]==ans[2]&&k<l)

continue;

while(nums[--l]==ans[3]&&k<l)

continue;

}

else if(nums[k]+nums[l]<temp)

k++;

else

l--;

}

while(nums[j]==nums[j+1]&&j<nums.size()-2)

j++;

}

while(nums[i]==nums[i+1]&&i<nums.size()-3)

i++;

}

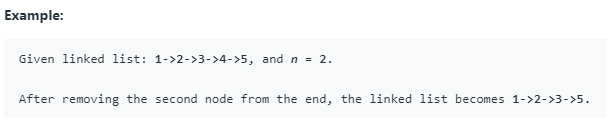
return res;

}

};

第十九题：

给定一个链表和一个正整数n，删掉倒数第n个元素然后返回链表头。



去掉链表中的倒数第n个节点。注意边界！！！！！！！思路比较简单，首先将一个指针q前进n位，然后与另一个指向头结点的指针p同时前进，当q->next==NULL时，p指向倒数第n+1个节点，p->next=p->next->next即可。注意，第一步时，如果p==NULL，说明去掉头结点，返回head->next。指针q遍历一次链表，所以时间复杂度为O（n）。

代码：

/\*\*

\* Definition for singly-linked list.

\* struct ListNode {

\* int val;

\* ListNode \*next;

\* ListNode(int x) : val(x), next(NULL) {}

\* };

\*/

class Solution {

public:

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

ListNode \*p=head,\*q=head;

while(n--)

q=q->next;

if(q==NULL)

return head->next;

while(q->next!=NULL)

{

q=q->next;

p=p->next;

}

p->next=p->next->next;

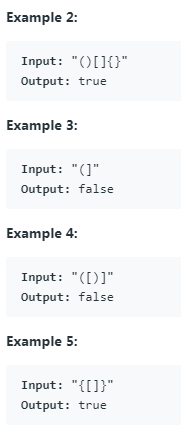
return head;

}

};

第二十题：

给定一个字符串s包含[](){}，判断这些括号是否全部匹配并且可以消除。



思路很简单。利用一个栈把左右的左括号压进栈中，如果遇到一个右括号，则检查如果栈为空或者栈顶不匹配则返回false，如果匹配则pop栈顶。到最后检查栈为空true否则false。遍历一次给定的字符串，时间复杂度为O（n）。

代码：

class Solution {

public:

bool isValid(string s) {

if(s.size()==0)

return true;

else if(s.size()%2==1)

return false;

stack<char> S;

map<char,char> cp={{')','('},{']','['},{'}','{'}};

for(char c:s)

{

if(c=='('||c=='['||c=='{')

S.push(c);

else if(c==')'){

if(S.empty()||S.top()!='(')

return false;

else

S.pop();

}

else if(c==']'){

if(S.empty()||S.top()!='[')

return false;

else

S.pop();

}else if(c=='}'){

if(S.empty()||S.top()!='{')

return false;

else

S.pop();

}

}

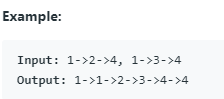
return S.empty()? true:false;

}

};

第二十一题：

给定两个有序链表，将其合并成一个链表且应保持有序。



思路很简单。首先new一个dummy作为合并后的链表头，然后当l1和l2均不为空时，不断比较l1和l2，并且把小的连接在dummy链中，以此类推。最后检查如果p非空则连接p，如果q非空则连接q。然后返回dummy->next。遍历l1和l2一次，所以时间复杂度为O（m+n）。

代码：

/\*\*

\* Definition for singly-linked list.

\* struct ListNode {

\* int val;

\* ListNode \*next;

\* ListNode() : val(0), next(nullptr) {}

\* ListNode(int x) : val(x), next(nullptr) {}

\* ListNode(int x, ListNode \*next) : val(x), next(next) {}

\* };

\*/

class Solution {

public:

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

ListNode \*dummy = new ListNode(-1),\*step=dummy;

while(l1||l2){

ListNode \*small=nullptr;

if((l1&&l2&&l1->val<l2->val)||!l2){

small=l1;

l1=l1->next;

}

else{

small=l2;

l2=l2->next;

}

step->next=small;

step=step->next;

}

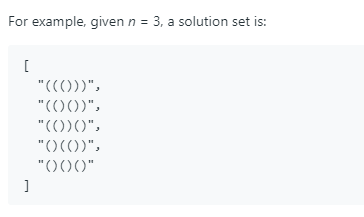
return dummy->next;

}

};

第二十二题：

给定数组n为括号的对数，输出所有合法的结果。



此题思路还是比较清晰的。此问题的解的数目是经典的卡特兰数，此处求所有的解，暂时用不上。首先假设l为”(”的个数，r为”)”的个数。递归检查l<n则l+1，然后检查r<l则r+1，跳出条件为l==r==n，注意两次检查都是if，而不是else if。卡特兰数的时间复杂度为O（C（n，2n））。

代码：

class Solution {

public:

vector<string> res;

void solve(int l, int r, int n, string cur) {

if (l == n && r == n) {

res.push\_back(cur);

return;

}

if (l < n)

solve(l + 1, r, n, cur + "(");

if (r < l)

solve(l, r + 1, n, cur + ")");

}

vector<string> generateParenthesis(int n) {

if (n == 0)

return res;

solve(0, 0, n, "");

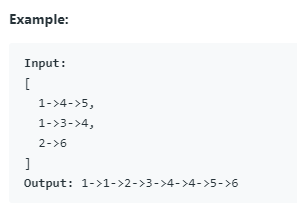
return res;

}

};

第二十三题：

给定n个有序链表，将其排列成一个有序链表。



思路稍微有点难想。使用存放链表节点的优先队列，在优先队列中n个链表结点按照升序排列，每次链接一个节点，然后pop掉，检查其后面是否还有节点，有则push。遍历所有的节点一次，所以时间复杂度是所有链表的长度之和O（l1+l2+...+ln）。

代码：

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

auto cmp=[](ListNode\* a,ListNode\* b){return a->val>b->val;};

priority\_queue<ListNode\*,vector<ListNode\*>,decltype(cmp)> Q(cmp);

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

Q.push(lists[i]);

ListNode\* res=new ListNode(0);

ListNode\* work=res;

while(!Q.empty())

{

work->next=Q.top();

Q.pop();

work=work->next;

if(work->next!=NULL)

Q.push(work->next);

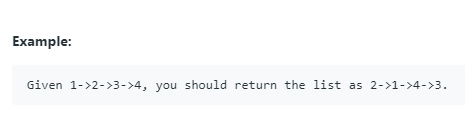
}

return res->next;

}

第二十四题：

给定一个链表，将其所有相邻的节点交换位置，然后返回。



思路很简单。首先定义一个空节点dummy连接到head，然后p指向head->next，然后检查p为空则break；q指向p->next，如果为空则break。然后以一定顺序交换pq，然后指针cur指向当前的最后一个节点，然后循环。遍历整个链表一次，时间复杂度为O（n）。上次考了个递归没写出来，这次写出来了，代码②自己看。

代码：

/\*\*

\* Definition for singly-linked list.

\* struct ListNode {

\* int val;

\* ListNode \*next;

\* ListNode(int x) : val(x), next(NULL) {}

\* };

\*/

class Solution {

public:

ListNode\* swapPairs(ListNode\* head) {

ListNode\* dummy = new ListNode(0);

dummy -> next = head;

ListNode\* cur = dummy;

while (cur != NULL) {

ListNode\* first = cur -> next;

if (first == NULL)

break;

ListNode\* second = first -> next;

if (second == NULL)

break;

// 按照一定的次序，交换相邻的两个结点。

cur -> next = second;

first -> next = second -> next;

second -> next = first;

cur = first;

}

return dummy -> next;

}

};

代码②：

/\*\*

\* Definition for singly-linked list.

\* struct ListNode {

\* int val;

\* ListNode \*next;

\* ListNode() : val(0), next(nullptr) {}

\* ListNode(int x) : val(x), next(nullptr) {}

\* ListNode(int x, ListNode \*next) : val(x), next(next) {}

\* };

\*/

class Solution {

public:

ListNode\* swapPairs(ListNode\* head) {

if(!head||!head->next)

return head;

ListNode \*temp=head->next;

head->next=swapPairs(temp->next);

temp->next=head;

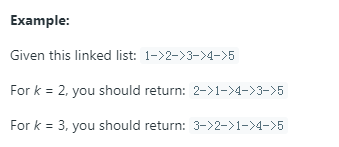
return temp;

}

};

第二十五题：

给定一个链表和一个正整数k，将每k个链表节点翻转，不足k个则不变，24题的加强版。



思路挺好想的。利用两个指针f和r，r一直前进直到到达f后第k个节点，如果不足k个节点则直接break。找到之后就是在f和r之间将其颠倒，需要借助另外一个指针。以此类推就能完成。遍历一遍数组，然后每次交换的时候又遍历了一次，所以一共2n次，所以时间复杂度为O（n）。

代码：

/\*\*

\* Definition for singly-linked list.

\* struct ListNode {

\* int val;

\* ListNode \*next;

\* ListNode(int x) : val(x), next(NULL) {}

\* };

\*/

class Solution {

public:

ListNode\* reverseKGroup(ListNode\* head, int k) {

ListNode\* dummy = new ListNode(0);

dummy -> next = head;

ListNode\* cur = dummy;

while (cur != NULL) {

ListNode \*first = cur -> next;

ListNode \*end = cur;

for (int i = 0; i < k && end != NULL; i++)

end = end -> next;

if (end == NULL)

break;

ListNode \*p1 = first;

ListNode \*p2 = first -> next;

while (p1 != end) {

ListNode \*new\_p2 = p2 -> next;

p2 -> next = p1;

p1 = p2;

p2 = new\_p2;

}

first -> next = p2;

cur -> next = end;

cur = first;

}

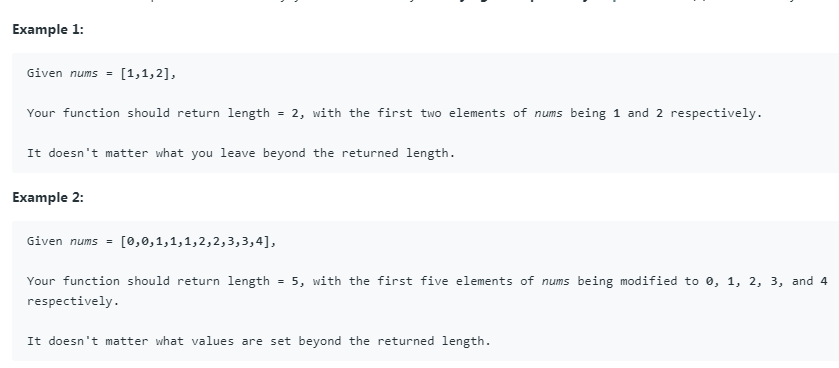
return dummy -> next;

}

};

第二十六题：

给定一个数组，原地“删除”所有重复元素，返回“删除”后的长度。



思路很简单。有点像unique函数做的事情，就是把不重复的元素挪到前面来，然后返回不重复元素的个数。两个指针k和i，k=0，i=1且不断前进，直到找nums[i]！=nums[k]的时候nums[++k]=nums[i]，直到数组末尾，最后返回k+1。遍历一遍数组，所以时间复杂度为O（n）。

代码：

class Solution {

public:

int removeDuplicates(vector<int>& nums) {

if (nums.size() == 0)

return 0;

int k = 0;

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

if (nums[i] != nums[k])

nums[++k] = nums[i];

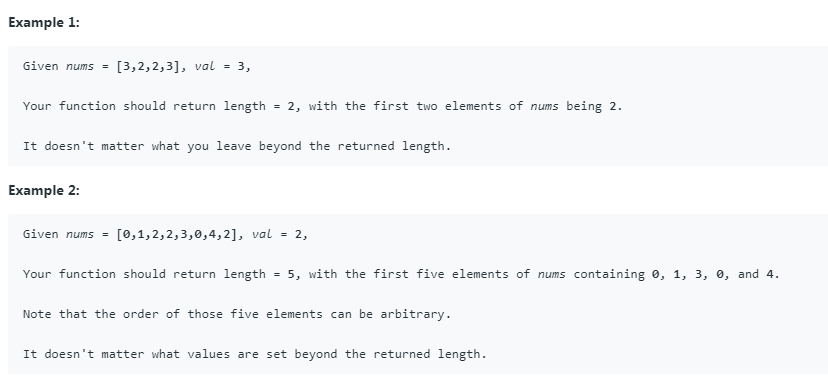
return k + 1;

}

};

第二十七题：

给定一个数组nums和一个val，“删除”数组里所有val的元素。

思路很简单。两个指针f和r，f从0开始找到一个val，r从最后面开始往前找到第一个不是val的数，然后交换f和r，直到f==r。检查f是否是val，是则返回f，否则返回f+1。遍历一遍数组，所以时间复杂度为O（n）。

代码：

class Solution {

public:

int removeElement(vector<int>& nums, int val) {

if(nums.size()==0)return 0;

int f=0,r=nums.size()-1;

while(f<r)

{

while(f<r&&nums[f]!=val)

++f;

while(f<r&&nums[r]==val)

r--;

swap(nums[f],nums[r]);

}

if(nums[f]==val)

return f;

else

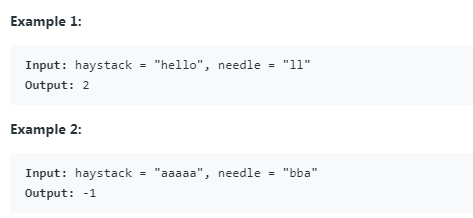
return f+1;

}

};

第二十八题：

给定一个字符串haystack和一个字符串needle，如果needle在haystack中则返回下标，否则返回-1.



思路很简单。就是实现一个KMP算法，KMP算法中主串是不会回溯的，所以时间复杂度为O（n+m）。

代码：

class Solution {

public:

int strStr(string haystack, string needle) {

int len1=haystack.size(),len2=needle.size();

if(len2==0)

return 0;

vector<int> next(len2,-1);

for(int i=0;i<len2-1;++i){

int j=next[i];

while(j!=-1&&needle[i]!=needle[j])

j=next[j];

next[i+1]=j+1;

}

int p1=0,p2=0;

while(p1<len1&&p2<len2){

if(p2!=-1&&haystack[p1]!=needle[p2])

p2=next[p2];

else

p1++,p2++;

}

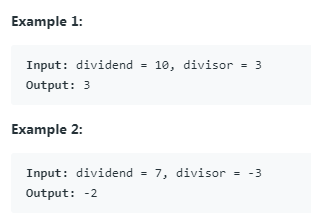
return p2==len2? p1-p2:-1;

}

};

第二十九题：

不能使用乘、除、mod运算来计算除法。



思路是通过位运算。首先用异或表示结果的符号sign，然后两个数取绝对值。用一个减法来模拟除法，首先将除数不断乘以2，并用一个m记录当前是几倍，然后ans+=m；然后被除数减去当前的除数，然后循环直到被除数小于除数。其实相当于在模拟一个二进制数，从最高位计算到最低位，最终得到结果。每一次计算一个二进制数位，最坏情况下是除以一，就是计算被除数的二进制表示了，所以时间复杂度为O（log被除数）。

代码：

class Solution {

public:

int divide(int dividend, int divisor) {

if (dividend == INT\_MIN && divisor == -1) {

return INT\_MAX;

}

long dvd = labs(dividend), dvs = labs(divisor), ans = 0;

int sign = dividend > 0 ^ divisor > 0 ? -1 : 1;

while (dvd >= dvs) {

long temp = dvs, m = 1;

while (temp << 1 <= dvd) {

temp <<= 1;

m <<= 1;

}

dvd -= temp;

ans += m;

}

return sign \* ans;

}

};

第三十题：

给定一个字符串s，和一个元素长度相等的字符串集合words，要求找出s中所有只包含集合中字符串不重复组合的子串的下标。



思路很难想。这题我认为是双指针的好题，暴力枚举会超时，所以必须另辟蹊径。注意到一个细节，words中所有元素的长度len都是相等的，所以s可以分成len种情况。上例中，可分为：“bar foo the foo bar man”、“b arf oot hef oob arm an”和“ba rfo oth efo oba rma n”三种情况，其中长度太小的会舍去，所以题目的复杂度可以得到改善。由于对时间要求比较高，我们使用unordered\_map<string，int> sign进行计数，首先将words中的元素进行计数，然后用correct记录sign的长度。上面三种情况可以用st=0到2来表示，i表示符合要求的第一个单词的下标，j表示准备检查的单词的下标，只有当j后面的字符足够组成一个单词才继续检查。用一个哈希表now表示i到j中单词的情况，每次检查j这个单词temp是否存在于sign中，如果不存在，则将now清空，cnt归零；否则now[temp]++。然后检查now[temp]==sign[temp]，如果是，则cnt++；然后检查是否如now[temp]>sign[temp]，说明这个单词多了，所以将其去掉，i前进，now[temp]--，注意，如果now[temp]==sign[temp]-1，说明已经不符合要求了，所以cnt--。最后检查cnt==correct，如果是说明此时i到j的单词符合要求，将i入队，然后i这个单词去掉，并且i前进，同时now[temp]--。每次分割之后都要遍历一次数组，总共分割了words[0].size（）次，最坏情况下每次都需要遍历两次s，所以总的时间复杂度为O（words[0].size（）\*s.size（））。

代码：

vector<int> findSubstring(string s, vector<string>& words) {

vector<int> res;

if(s.size()!=0&&words.size()!=0)

{

unordered\_map<string,int> sign;

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

sign[words[i]]++;

int correct=sign.size();

for(int st=0;st<words[0].size();++st)

{

int cnt=0;

unordered\_map<string,int> now;

for(int i=st,j=st;j<s.size()-words[0].size()+1;j+=words[0].size())

{

string temp=s.substr(j,words[0].size());

if(sign.find(temp)==sign.end())

{

i=j+words[0].size();

now.clear();

cnt=0;

}

else

{

now[temp]++;

if(now[temp]==sign[temp])

cnt++;

while(now[temp]>sign[temp])

{

string pity=s.substr(i,words[0].size());

now[pity]--;

i+=words[0].size();

if(now[pity]==sign[pity]-1)

cnt--;

}

if(cnt==correct)

{

res.push\_back(i);

now[s.substr(i,words[0].size())]--;

i+=words[0].size();

cnt--;

}

}

}

}

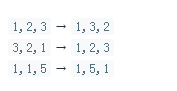
}

return res;

}

第三十一题：

给定一个数字nums，不能改变含有的数字，返回比这个数刚好大一点的数，如果已经是最大数则返回最小数（所有元素升序）。



思路主要还是要在纸上模拟。仔细思考发现，如果要比原数字大一点，那么必须是低位的数字增大一点，且必须交换之后数字是增大的，所以必须从最低位开始查找，并且这个交换是较高位的小数与较低位的大数交换。所以从后往前查找一个升序对，此时该较小数字i后面必然是降序排列，所以从最后开始找到第一个大于i的数字交换，然后将i后面的数字升序排列即可，如果没有找到，将把整个数字升序排列。遍历一遍这个数字的所有位，所以时间复杂度为O（n）。

代码：

class Solution {

public:

void nextPermutation(vector<int>& nums) {

int n = nums.size();

int j = -1;

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

if (nums[i] < nums[i + 1]) {

j = i;

break;

}

if (j == -1)

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

else {

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

if (nums[i] > nums[j]) {

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

break;

}

reverse(nums.begin() + j + 1, nums.end());

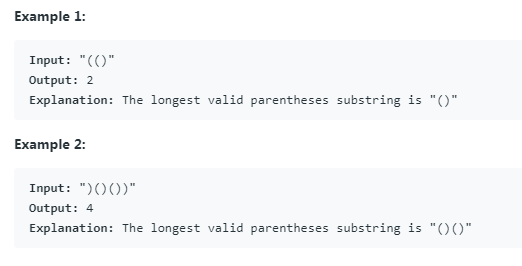
}

}

};

第三十二题：

给定一个字符串s，只包含“（”和“）”，找到最长的合法的括号序列，返回长度。



思路很简单。括号匹配问题的最好做法就是用栈，首先初始化一个start作为当前记录的开始下标。遇到左括号就将下标入栈，遇到右括号就检查栈是否为空，如果为空，说明不合法，start=i+1；如果不为空，则pop。如果pop后栈为空，则说明此时括号完全匹配，记录i-start+1；如果pop后栈不为空，则说明部分匹配，所以记录i-st.top。遍历了一遍整个字符串，所以时间复杂度为O（n）。

代码：

int longestValidParentheses(string s) {

int n = s.length();

stack<int> st;

int start = 0, ans = 0;

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

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

st.push(i);

else {

if (!st.empty()) {

st.pop();

if (st.empty())

ans = max(ans, i - start + 1);

else

ans = max(ans, i - st.top());

} else {

start = i + 1;

}

}

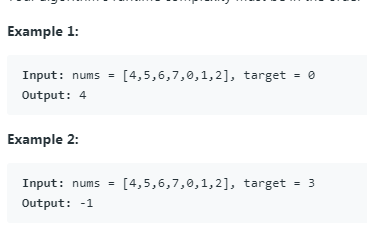
}

return ans;

}

第三十三题：

在一个旋转数组中查找target，有则返回下标，没有返回-1。



思路就是直接二分，但是判断稍微要思考一下。在这个数组翻转之后，我们通过判断nums[0]来判断当前的mid在哪一块，如果mid比0要大，说明mid当前在较大的这一块升序列表中。如果此时target比mid小且比0大，那target就在mid的左边，其他情况都在右边。以此类推，最终只剩一个元素的时候判断是否是target就行了。二分查找的时间复杂度是O（logn）。

代码：

class Solution {

public:

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

int n = nums.size();

if (n == 0)

return -1;

int l = 0, r = n - 1;

while (l < r) {

int mid = (l + r) >> 1;

if (nums[mid] >= nums[0]) {

if (target > nums[mid]||target < nums[0])

l = mid + 1;

if (target <= nums[mid] && target >= nums[0])

r = mid;

}

else {

if (target >= nums[0]||target <= nums[mid])

r = mid;

if (target > nums[mid] && target < nums[0])

l = mid + 1;

}

}

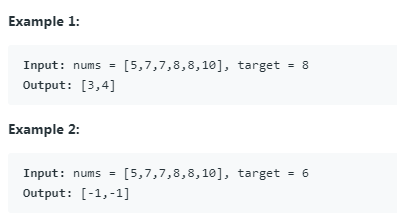
return nums[l] == target ? l : -1;

}

};

第三十四题：

给定一个升序数组，找到target在这个数组中的最小下标和最大下标，如果没有返回两个-1.



思路很简单，就是两次二分查找，一次靠左一次靠右。首先是靠左查找，找到第一个target所在的下标，这里注意跳出循环之后如果nums[f]！=target那说明target不存在，直接返回。如果是target，只需要将r复位，因为f肯定小于等于target的最右端，然后靠右找到target的下标，返回结果。两次二分查找，所以总的时间复杂度为O（logn）。

代码：

class Solution {

public:

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

vector<int> res={-1,-1};

if(nums.size()==0||nums[0]>target||nums[nums.size()-1]<target)

return res;

int f=0,r=nums.size();

while(f<r){

int mid=(f+r)>>1;

if(nums[mid]<target)

f=mid+1;

else

r=mid;

}

if(nums[f]==target){

res[0]=f,r=nums.size()-1;

while(f<r){

int mid=(f+r+1)>>1;

if(nums[mid]>target)

r=mid-1;

else

f=mid;

}

res[1]=f;

}

return res;

}

};

第三十五题：

给定一个升序数组，找到target这个数，如果没找到就输出它应该插入的下标。



思路很简单，就是靠左的二分。因为靠左的二分最后f会落在一个大于等于target的数字上，所以如果nums[f]大于等于target，直接返回下标即可；当target大于nums[f]，说明要插在最后，此时才返回nums.size。二分查找的时间复杂度为O（logn）。

代码：

class Solution {

public:

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

int n = nums.size();

if (n == 0)

return 0;

int l = 0, r = n - 1;

while (l < r) {

int mid = (l + r) >> 1;

if (nums[mid] < target)

l = mid + 1;

else

r = mid;

}

if (nums[l] >= target)

return l;

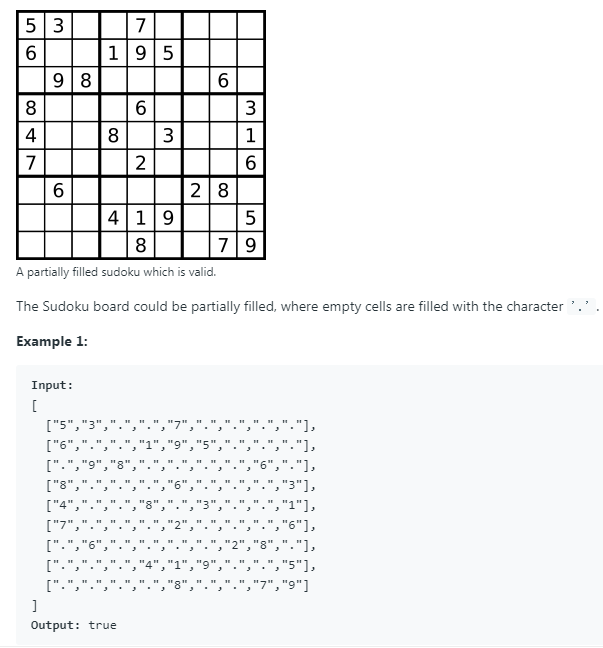
return n;

}

};

第三十六题：

判断这个九宫格里面已经填好的数字是否合法，合法返回true，否则返回false。



思路很简单，就是查找和记录。创建一个3x9x9的数组，第一层记录第i行出现的数字，第二层记录第j列出现的数字，第三次记录第k=3\*(i/3)+(j/3)个九宫格出现的数字。注意这里的数字是1-9，所以应该int[3][10][10]。

代码：

class Solution {

public:

int jd[3][10][10];

bool check(int i,int j,vector<vector<char>>& b)

{

int temp=int(b[i][j]-'0');

if(jd[0][i][temp]==1)

return true;

else

jd[0][i][temp]=1;

if(jd[1][j][temp]==1)

return true;

else

jd[1][j][temp]=1;

int bl=3\*(i/3)+(j/3);

if(jd[2][bl][temp]==1)

return true;

else

jd[2][bl][temp]=1;

return false;

}

bool isValidSudoku(vector<vector<char>>& board) {

for(int i=0;i<9;++i)

for(int j=0;j<9;++j)

if(board[i][j]!='.'&&check(i,j,board))

return false;

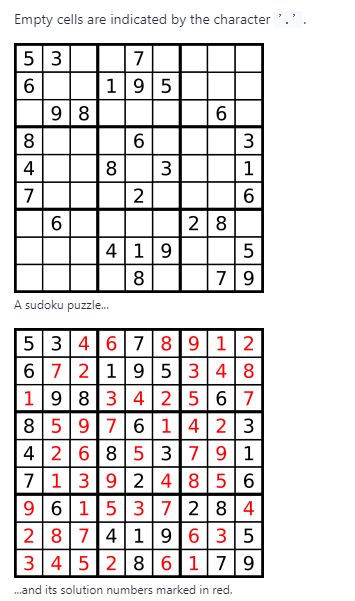
return true;

}

};

第三十七题：

解数独。



思路直接使用搜索算法，用数组signal分别记录第i行、第j列，第k块中数字1-9的使用情况。i为0-8，j为0-8，k为1-9，k的计算方法为3\*（i/3）+j/3+1，不能约分，因为需要求出i/3的整数值。首先将已有数字记录下来表示初始化，然后从左上角开始枚举，遇到“.”且数字i在此时signal中合法时才进行枚举。注意一点，枚举时要随时修改signal以及board中的值，当返回时要让signal归0且board归“.”。跳出条件为x==9，表示枚举完成。最坏情况下每个格子需要枚举1-9一共9个数字，一共有9\*9个格子，所以最坏情况下时间复杂度为O（9^3）。

代码：

class Solution {

public:

int signal[3][10][10],flag=0;

void dfs(vector<vector<char>>& b,int x,int y)

{

if(x==9)

{

flag=1;

return;

}

if(b[x][y]!='.')

{

if(y==8)

dfs(b,x+1,0);

else

dfs(b,x,y+1);

}

else

{

for(int i=1;i<10;++i)

if(signal[0][x][i]==0&&signal[1][y][i]==0&&signal[2][3\*(x/3)+y/3+1][i]==0)

{

b[x][y]=i+'0';

signal[0][x][i]=1;

signal[1][y][i]=1;

signal[2][3\*(x/3)+y/3+1][i]=1;

if(y==8)

dfs(b,x+1,0);

else

dfs(b,x,y+1);

if(flag)

return;

signal[0][x][i]=0;

signal[1][y][i]=0;

signal[2][3\*(x/3)+y/3+1][i]=0;

b[x][y]='.';

}

}

return;

}

void solveSudoku(vector<vector<char>>& board) {

memset(signal,0,sizeof(signal));

for(int i=0;i<9;++i)

for(int j=0;j<9;++j)

if(board[i][j]!='.')

{

int num=board[i][j]-'0';

signal[0][i][num]=1;

signal[1][j][num]=1;

signal[2][3\*(i/3)+j/3+1][num]=1;

}

dfs(board,0,0);

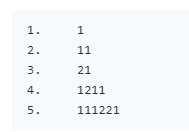
return;

}

};

第三十八题：

报数，比如“1211”读作一个一、一个二、二个一，所以下一个是“111221”，以此类推。



思路是模拟。当i<n的时候不断模拟，首先用一个下标j来记录当前数字last[j]，记录这个数字连续出现了count次，然后将count和这个数字一起加入字符串，作为这一次报数的结果。以此类推，直到i==n的时候返回。字符串长度是逐步递增的，所以很难计算每次枚举的字符串长度到底是多少，但是可以直到一共枚举了n次，所以时间复杂度为O（n）。

代码：

class Solution {

public:

string countAndSay(int n) {

string last = "1";

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

string gen = "";

int count = 0;

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

count++;

if (j == last.length() - 1 || last[j] != last[j + 1]) {

gen += to\_string(count) + last[j];

count = 0;

}

}

last = gen;

}

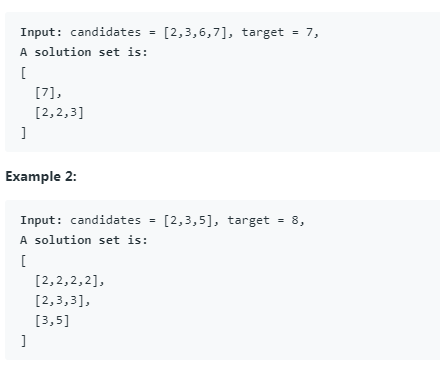
return last;

}

};

第三十九题：

给定一个数组candidates，数字是无重复的，找出里面的组合sum为target，数字可以重复使用。



思路是直接深度优先搜索算法，但是直接搜索复杂度肯定很大，所以一定要剪枝。最容易想到的剪枝策略是枚举范围超过candidates长度，其次就是枚举的大小超过target。开始前先对candidates排序，然后开始就行。这道题可以学习到一个技巧，如果要重复使用元素可以先push当前下标i，然后继续枚举当前下标i，然后pop掉之后再枚举i+1。因为每一次递归都会产生两个递归入口，且这两个递归入口的长度大致一样，所以大概理解为T（n）=2\*T（n-1），所以最终的时间复杂度为O（2^n）。

代码：

class Solution {

public:

void solve(int i, vector<int>& candidates, int sum,

vector<int> &ch, int target, vector<vector<int>>& ans) {

// 注意这里的 ch 是引用。

if (sum == target) { // 找到目标值，添加答案。

ans.push\_back(ch);

return;

}

if (i == candidates.size() || sum > target) // 超出范围回溯。

return;

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

solve(i, candidates, sum + candidates[i], ch, target, ans);

ch.pop\_back();

solve(i + 1, candidates, sum, ch, target, ans);

}

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

vector<vector<int>> ans;

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

vector<int> ch; // ch 记录已选择的数字。

solve(0, candidates, 0, ch, target, ans);

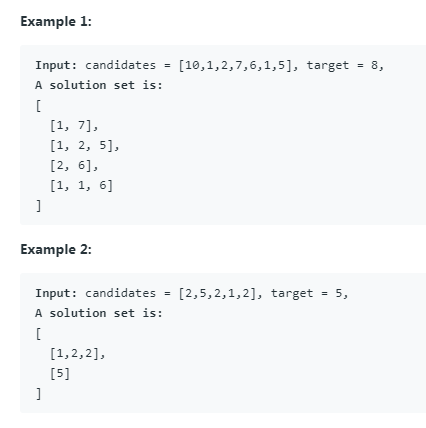
return ans;

}

};

第四十题：

给定一个数组candidates，数字有可能重复，找出里面的组合sum为target，数字不可重复使用。



思路是DFS。用一个for循环，检查当前数字是否与前一个相同，是的话就跳过。不用担心会漏掉，因为这只代表这一层，比如[1,1,2,3]，第一层只有一个1，第二层的时候由于i==order所以不会continue，会变成[1,1]，然后顺利完成。由于每次递归都有两个递归入口，所以总的时间复杂度应该是O（2^n）。

代码：

class Solution {

public:

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

{

vector<vector<int>> res;

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

vector<int> local;

findCombination(res, 0, target, local, num);

return res;

}

void findCombination(vector<vector<int>>& res, const int order, const int target, vector<int>& local, const vector<int>& num)

{

if(target==0)

{

res.push\_back(local);

return;

}

else

{

for(int i = order;i<num.size();i++) // iterative component

{

if(num[i]>target) return;

if(i&&num[i]==num[i-1]&&i>order) continue; // check duplicate combination

local.push\_back(num[i]),

findCombination(res,i+1,target-num[i],local,num); // recursive componenet

local.pop\_back();

}

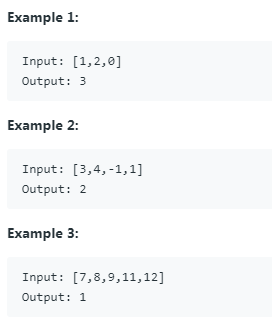
}

}

};

第四十一题：

给定一个数组，找到缺失的最小正整数，数组里面可能有重复、负数。



思路是桶排序，即将元素i放在nums[i-1]那里，最后检查nums[i]！=i+1就是要找的正整数。这里排序的判断条件是这个nums[i]在0到nums.size的范围内，如果超出范围说明没有他对应的位置，也说明了这个元素对于现在来说是多余的，因为此时必然前面缺失了一个正整数。如果遇到重复的元素也不担心，因为这个算法会检查该元素应该在的位置是否已经是对应的数字，所以多余的数字会想unique一样放在最后。遍历一次数组，所以时间复杂度为O（n）。

代码：

class Solution{

public:

int firstMissingPositive(vector<int>& nums)

{

int n = nums.size();

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

while(nums[i] > 0 && nums[i] <= n&& nums[nums[i] - 1] != nums[i])

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

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

if(nums[i] != i + 1)

return i + 1;

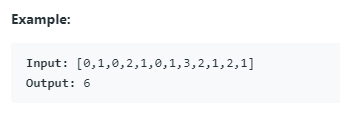
return n + 1;

}

};

四十二题：

给定一个数组，记录了每一个位置的柱子高度，记录一下下雨最多能存储多少水。



思路还是很独特的。储水量等于每一个柱子上方储水量的总和，而每个柱子上方的储水量等于它左边最高的和右边最高的较小的一个减去它自身的高度。那么有一个搜索三次的思路：维护一个数组记录每一个柱子左边最高的柱子，left[i]=max（left[i-1]，height[i]），右边同理，最后每一个柱子上方的储水量为min（left[i]，right[i]）-height[i]。第二个思路比较牛皮：每一个柱子上方的储水量为这个柱子的左边第一高的柱子和右边第一高的柱子围起来的面积，当然还有可能有更高的，所以这个思路的计算是一层层进行的，所以不会冲突，而且不会重复。每次遇到右边的柱子比当前高的时候就计算，如果当前与前一个柱子一样高，那个结果为0，也就是说，只有遇到左边比当前高的时候才会计算，所以这个时候不会重复。例如有一个数组[3,2,1,0,1,2,3]，一开始res=1，就是那个0的那个地方；然后是下标2~4，res=1+3=4，这里就是第二层；最后是下标1~5，res=4+5=9。

代码：

class Solution {

public:

int trap(vector<int>& height) {

int n=height.size(),res=0;

if(n==0)

return res;

stack<int> st;

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

{

while(!st.empty()&&height[st.top()]<height[i])

{

int top=st.top();

st.pop();

if(st.empty())

break;

res+=(i-st.top()-1)\*(min(height[st.top()],height[i])-height[top]);

}

st.push(i);

}

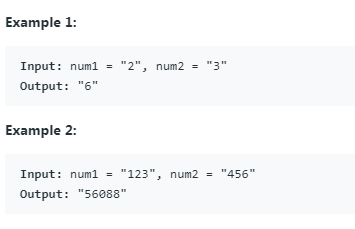
return res;

}

};

第四十三题：

大数相乘，给定两个很长的只包含0~9的字符串，以字符串的形式返回乘积。



思路就是模拟大数的相乘过程。一个数的长度为m，一个为n，那么他们的乘积最长为m+n-1到m+n，所以声明一个数组c长度为m+n，每个位置代表结果对应位置的数。因为我们从最低位开始计算，所以先把两个字符串翻转一下。最后不断计算结果并注意进位就行了。因为第一个数要与第二个数的每个数字相乘，所以时间复杂度为O（m\*n）。

代码：

class Solution {

public:

string multiply(string num1, string num2) {

int n = num1.length(), m = num2.length();

vector<int> a(n), b(m), c(n + m);

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

a[n - i - 1] = num1[i] - '0';

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

b[m - i - 1] = num2[i] - '0';

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

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

c[i + j] += a[i] \* b[j];

c[i + j + 1] += c[i + j] / 10;

c[i + j] %= 10;

}

int l = n + m;

while (l > 1 && c[l - 1] == 0) l--;

string ans = "";

for (int i = l - 1; i >= 0; i--)

ans += c[i] + '0';

return ans;

}

};

第四十四题：

**这题重要。**匹配符匹配，给定字符串s和字符串p，其中s只包含a-z，p包含a-z、\*、？三种，？对应任意一个字符，\*对应任意长度的字符串，可以对应0个，输出s和p是否匹配。



思路是动态规划。维护一个数组dp[i][j]表示s的前i个字符串和p的前j个字符串是否匹配，匹配为true，否则为false。字符串匹配问题用动态规划可以说非常经典了，将s整个字符串与p整个字符串匹配划分为s的长度为0~s.size（）的匹配。三种情况：①当s[i]==p[j]时，dp[i][j]=dp[i-1][j-1]，就是说s[i]与p[j]匹配，所以当s[i-1]和p[j-1]也匹配时，dp[i][j]为true；②当p[j]==‘\*’，dp[i][j]=dp[i][j]|=dp[i][j-1]，就是说把\*当做对应0个字符的时候，s[i]和p[j-1]匹配的时候为true；③当p[j]==‘\*’且i>0时，dp[i][j]=dp[i][j]|=dp[i-1][j]，就是说把\*当做对应n个字符的时候，s[i-1]和p[j]匹配的时候为true；比如当s=“abcde”，p=“a\*e”，在i=3，j=2的时候，s=“abc”，p=“a\*”，首先当\*对应0个字符串，“abc”与“a”不匹配，然后将\*对应一个字符串，问题变成“ab”和“a\*”是否匹配；这个时候就套娃了，“ab”与“a\*”首先把\*当做对应0个字符，“ab”和“a”不匹配，然后将\*对应1个字符，“a”和“a\*”是否匹配；把\*当做对应0个字符，“a”和“a”匹配，所以以上所说的全部匹配，都为true。

代码：

class Solution {

public:

bool isMatch(string s, string p) {

int s1=s.size(),s2=p.size();

vector<vector<bool>> dp(s1+1,vector<bool>(s2+1,false));

dp[0][0]=true;

for(int i=0;i<=s1;++i)

for(int j=1;j<=s2;++j)

{

char y=p[j-1];

if(i>0)

{

char x=s[i-1];

if(x==y||y=='?')

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

}

if(y=='\*')

{

dp[i][j]=dp[i][j]|dp[i][j-1];

if(i>0)

dp[i][j]=dp[i][j]|dp[i-1][j];

}

}

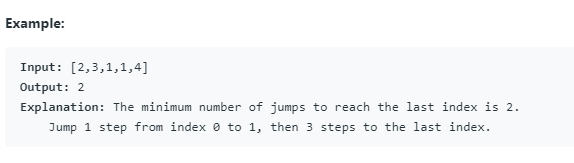
return dp[s1][s2];

}

};

第四十五题：

给定一个数组，每一个元素代表此时可以跳跃的最远距离，计算到达最后一个元素的最少跳跃次数。

思路是动态规划中加入贪心策略。维护一个数组f[n]表示到达n所需要的的最小步数，在当前位置last，可以到达的最远距离是last+nums[last]，所以用i遍历数组的时候，i<last+nums[last]的f[i]都是f[last]+1。当i超出last+nums[last]时，将last++，直到遍历整个数组。遍历一个整个数组，所以时间复杂度为O（n）。

代码：

class Solution {

public:

int jump(vector<int>& nums) {

int n = nums.size();

vector<int> f(n);

f[0] = 0;

int last = 0;

for (int i = 1; i < n; i++) { // 依次求f[i]的值。

while (i > last + nums[last]) // 根据i来更新last。

last++;

f[i] = f[last] + 1; // 根据f[last]更新f[i]。

}

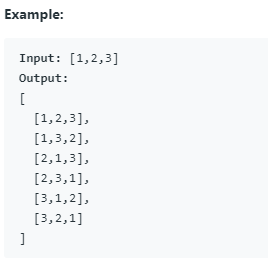
return f[n - 1];

}

};

第四十六题：

给定一个数组nums，里面的数字不会有重复，返回其所有的排列可能。



思路是全排列，最低的时间复杂度也要n！。所以通过枚举每个数字所在的位置效率更高：同样，使用num来标记当前递归对应的元素下标，当num==len时表示得到一个解。否则每次递归使用一个for循环枚举num到len的每个位置，每次开始是i=num，确保初始状态存在。因为每次枚举都确定了num对应下标的元素，所以不会重复并且不会有冗余，比直接枚举要好很多。第一个数有n个位置，第二个数有n-1个位置，以此类推，在最后push答案的时候复制n个元素，所以每次都要O（n），因此时间复杂度为O（n\*n！）。

代码：

class Solution {

public:

void dfs(int num,int len,vector<int>& nums,vector<vector<int>>& res)

{

if(num==len)

{

res.push\_back(nums);

return;

}

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

{

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

dfs(num+1,len,nums,res);

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

}

return;

}

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

vector<vector<int>> res;

vector<int> temp;

int len=nums.size();

if(len==0)

{

res.push\_back(temp);

return res;

}

dfs(0,len,nums,res);

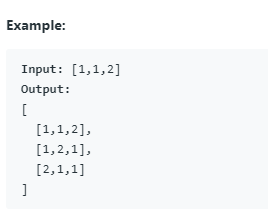
return res;

}

};

第四十七题：

给定一个数组nums，会有重复数字，要求找出所有不同的排列组合。



思路是全排列，与前一题不同的是，这里的数字会有重复，所以简单地交换会有重复。为了不重复，对于相同的数字必须保持其相对位置不变，比如2个数字“1”，第一个数字“1”必须一直在第二个数字“1”的前面。首先对数组进行排序，signal[i]表示位置i是否已经被占用，tp是一个解，step表示枚举nums[num]摆放位置的下标。因为要保持相对位置不变，所以如果num+1元素==num元素，那么num+1这个元素最小要放在num的后面，所以num+1的step是i+1；如果num+1的元素不同num，那么随便你放哪里，所以step为0。每次决定一个元素的摆放位置，所以总的情况是n！种，每次复制答案需要n，所以总的时间复杂度为O（n\*n！）。

代码：

class Solution {

public:

void dfs(int num,int step,int len,vector<int>& nums,vector<int>& sg,vector<int>& tp,vector<vector<int>>& res)

{

if(num==len)

{

res.push\_back(tp);

return;

}

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

if(sg[i]==0)

{

sg[i]=1;

tp[i]=nums[num];

if(num+1<len&&nums[num+1]!=nums[num])

dfs(num+1,0,len,nums,sg,tp,res);

else

dfs(num+1,i+1,len,nums,sg,tp,res);

sg[i]=0;

}

return;

}

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

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

int len=nums.size();

vector<vector<int>> res;

vector<int> signal(len,0),temp(len,0);

if(len==0)

{

res.push\_back({});

return res;

}

dfs(0,0,len,nums,signal,temp,res);

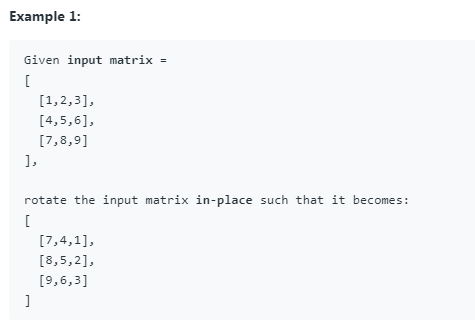
return res;

}

};

第四十八题：

给定一个n\*n的矩阵，要求将矩阵顺时针旋转90度，不得另外创建矩阵，必须在这个矩阵内实现。



观察规律可得，旋转即将[x，y]的数放到[y，len-1-x]中，其中len为边长n；每四个元素为一组进行互换，即[0,0]和[0，n]交换，[0，n]和[n，n]交换，[n，n]和[n，0]交换，最后[n，0]和[0,0]交换，注意，因为每次交换之后，下次交换的元素都在[0,0]处，所以每次都跟[0,0]交换就行。这里的m和n表示这一次参与交换的最小下标和最大下标，每次for循环会完成最外面一个正方形的交换，所以最小下标会变成m+1，最大下标会变成n-1，直到m==n的时候说明只有最后最中心的元素了，此时跳出循环。因为除了最中心的元素以外，所有的元素都换了一个位置，所以总的时间复杂度为O（n^2）。

代码：

class Solution {

public:

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

int size = matrix.size();

int m = 0, n = size - 1;

while (m < n) {

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

swap(matrix[m][m + i], matrix[m + i][n]);

swap(matrix[m][m + i], matrix[n][n - i]);

swap(matrix[m][m + i], matrix[n - i][m]);

}

++m; --n;

}

}

};

第四十九题：

给定一个含有很多字符串的数组strs，将字符串包含的所有字符相同的字符串归为一类，然后返回。



这题应该是考察STL的熟练程度，思路挺简单的，先将strs中每个元素str的字符进行排序，得到唯一表示这些字符的字符串key。比如“ate”和“eat”都表示为“aet”，然后利用一个unordered\_map，刚才得到的key作为键，然后元素str归入到value，以此类推就能得到一个归类完成的dict。然后将每一个key表示一个元素，将这个dict转换为vector就是答案。这里的move是右值引用，可以加快运行速度。遍历了一次每个元素，然后每个元素的长度为L，排序需要LlogL，所以总的时间复杂度为O（NLlogL）。

代码：

class Solution {

public:

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

vector<vector<string>> res;

unordered\_map<string,vector<string>> dict;

for(auto& str:strs)

{

string key=str;

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

dict[key].push\_back(move(str));

}

for(auto i=dict.begin();i!=dict.end();++i)

res.push\_back(move(i->second));//注意这里的i是一个指针，所以要用->

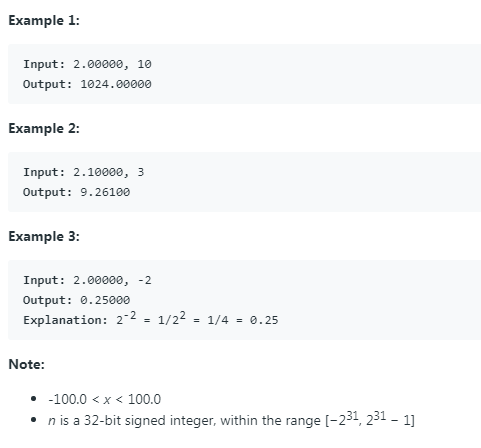
return res;

}

};

第五十题：

给定一个双精度浮点数x和一个整数n，计算x的n次方，其中n可以是负数。



如果使用n为偶数则计算x的n/2次方乘以x的n/2次方的方法，复杂度巨大，因为类似简单递归计算斐波那契数列，每次计算n都要重新拆分，但是如果添加备忘录，那么空间复杂度太高。因此选用二进制的方法，因为n为int类型，所以最高为32位，因此计算不会超过32次。因为10的二进制表示为1010，得x^10=x^8\*x^2，因此可以通过此方法计算时间复杂度和空间复杂度都很低。如果n为负数，那么最后返回的时候取倒数即可。计算的复杂度取决于pow的值n的二进制表示有多少位，所以总的时间复杂度为O（logn）。

在剑指offer上学到一招，好像速度也还行，这里我放在代码②里面吧，复杂度应该是O（logn）。

代码：

class Solution {

public:

double myPow(double x, int n) {

double res=1,p=x;

long long times=abs((long long)(n));//注意如果n为INT\_MIN直接abs会溢出，所以要先强制转换为longlong然后在赋值

for(;times;times>>=1)

{

if(times&1)

res\*=p;

p\*=p;

}

return n>0?res:1/res;

}

};

代码②：

class Solution {

public:

int purePow(double x, long n) {

if (n == 0)

return 1;

if (n == 1)

return x;

double ans = purePow(x, n >> 1);

ans \*= ans;

if (n & 1)

ans \*= x;

return ans;

}

double myPow(double x, int n) {

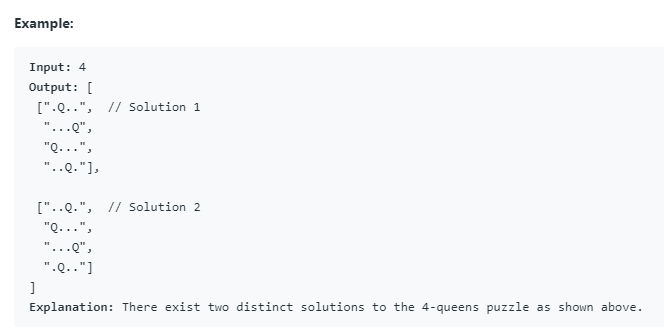
return n < 0 ? 1.0 / purePow(x, abs(n)) : purePow(x, n);

}

};

第五十一题：

N-皇后难题，给定一个n，要求在nxn的棋盘上摆放n个皇后，并且n个皇后无法互相攻击，已知皇后攻击范围为横、竖、对角线任意距离。



一道搜索题，在本题中学到了一个创建n个相同字符串的方法：vector<string> temp（n，string（n，‘x’））。横、竖很简单，只要维护一个二维数组就可以了。对角线有两个办法：如果存在两个点（x1，y1）和（x2，y2）是在同一个对角线上的，那么这两个点的x方向的距离和y方向的距离相等。所以只有abs（x1-x2）==abs（y1-y2）则说明两个点在对角线上。代码中dp[x]=y表示皇后x的坐标为（x，y）。因为每次都会确定一个皇后的列，所以第一个皇后可以有n个位置选，第二个皇后有n-1个位置，所以至少时间复杂度为O（n！）。

代码：

class Solution {

public:

vector<vector<string>> solveNQueens(int n) {

vector<vector<string> > res;

vector<string> temp(n, string(n,'.'));

int dp[n];//记录这n个皇后的纵坐标

helper(res, temp, dp, 0, n);

return res;

}

private:

void helper(vector<vector<string> > &res, vector<string> &temp, int dp[], int row, int n){

if(row==n){

res.push\_back(temp);

return;

}

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

if(valid(dp, row, col, n)){

dp[row]=col;

temp[row][col]='Q';

helper(res, temp, dp, row+1,n);

temp[row][col]='.';

}

}

}

bool valid(int dp[], int row, int col, int n){

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

if(dp[i]==col || abs(row-i)==abs(dp[i]-col)){

return false;

}

}

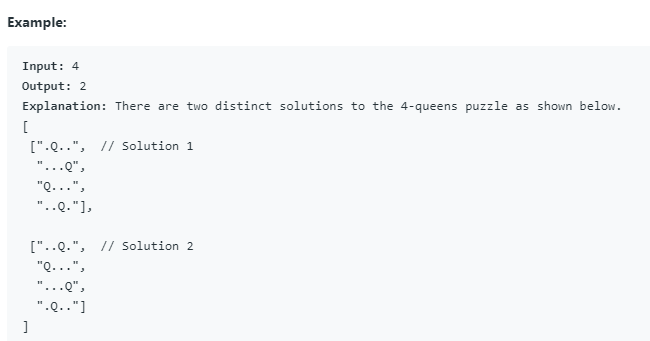
return true;

}

};

第五十二题：

给定一个正整数n，要求返回n皇后难题解得个数。



这一题的解题思路与上一题一样，但是不用temp来枚举了，只需要一个wh[i]=j来表示第i个皇后的坐标为（i，j）。

代码：

class Solution {

public:

bool vaild(int num,int col,vector<int>& wh){

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

if(col==wh[i]||abs(num-i)==abs(col-wh[i]))

return false;

return true;

}

void dfs(int num,int n,vector<int> wh,int& cnt){

if(num==n){

cnt++;

return;

}

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

if(vaild(num,i,wh)){

wh[num]=i;

dfs(num+1,n,wh,cnt);

}

return;

}

int totalNQueens(int n) {

vector<int> wh(n,0);

int cnt=0;

dfs(0,n,wh,cnt);

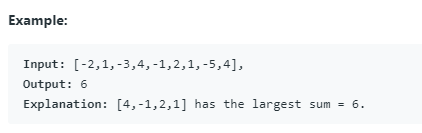
return cnt;

}

};

第五十三题：

给定一个数组nums，找出其最大连续子串和。



两个方法，我稍微觉得第二个方法比较微妙：①使用一个初始为0的数sum来记录当前连续子串的和，每次加一个元素，并更新最优解ans，同时检查sum=max（sum，0），如果sum小于0说明前面的连续子串和小于0，只需要重新找到一个正数重新开始计算即可；②这个方法不用检查sum是否小于0，只需要每次更新f[i]=max（f[i-1]+nums[i]，nums[i]），其中f[i]表示前i项子串和的最优解，由关系式可知，可以保证后面计算的子串连续性，然后每次更新最优解为ans=（ans，f[i]）。两种方法都是遍历一次整个数组，所以时间复杂度为O（n）。

代码①：

class Solution {

public:

int maxSubArray(vector<int>& A) {

int ans=A[0],i,sum=0;

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

sum+=A[i];

ans=max(sum,ans);

sum=max(sum,0);

}

return ans;

}

};

代码②：

class Solution {

public:

int maxSubArray(vector<int>& nums) {

int n = nums.size(), ans;

vector<int> f(n);

f[0] = nums[0];

ans = f[0];

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

f[i] = max(f[i - 1] + nums[i], nums[i]);

ans = max(ans, f[i]);

}

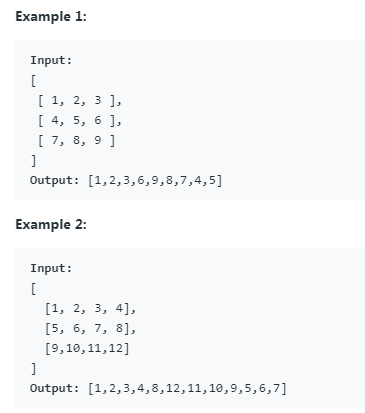
return ans;

}

};

第五十四题：

给定一个m\*n的数组，按螺旋顺序返回所有元素。



思路是模拟。初始化c=1，表明总共需要进入res的元素总数，并初始化l=0，r=n-1，t=0，d=m-1分别表示左右上下。然后初始化一个d表示现在的操作，d%4==0表示t这一行；1表示r这一列；2表示d这一行；3表示l这一列。每次完成后要将对应的字母+1表示这一行或者这一列的元素已经遍历。遍历一次整个数组，所以时间复杂度为O（n^2）。

最近新增了一种递归思路，在②里面：

代码：

class Solution {

public:

vector<int> spiralOrder(vector<vector<int>> arr) {

if(arr.size()==0)

return {};

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

vector<int> v;

int c=1;

int d=0;

int l=0,r=n-1,t=0,b=m-1;

while(c<=m\*n){

if(d%4==0){

for(int i=l;i<=r;i++){

v.push\_back(arr[t][i]);

}

c+=r-l+1;

t++;

}else if(d%4==1){

for(int i=t;i<=b;i++){

v.push\_back(arr[i][r]);

}

r--;

c+=b-t+1;

}else if(d%4==2){

for(int i=r;i>=l;i--){

v.push\_back(arr[b][i]);

}

c+=r-l+1;

b--;

}else{

for(int i=b;i>=t;i--){

v.push\_back(arr[i][l]);

}

c+=b-t+1;

l++;

}

d++;

}

return v;

}

};

代码②：

class Solution {

public:

void printCircle(int rows,int cols,int circle,vector<vector<int>> &matrix,vector<int> &ans){

//仔细观察这里面的每一个边界

        int xlimit=cols-circle-1,ylimit=rows-circle-1;

        for(int i=circle;i<=xlimit;++i)

            ans.push\_back(matrix[circle][i]);

        if(circle<ylimit)

            for(int i=circle+1;i<=ylimit;++i)

                ans.push\_back(matrix[i][xlimit]);

        if(circle<ylimit&&circle<xlimit)

            for(int i=xlimit-1;i>=circle;--i)

                ans.push\_back(matrix[ylimit][i]);

        if(circle<xlimit&&circle<ylimit-1)

            for(int i=ylimit-1;i>=circle+1;--i)

                ans.push\_back(matrix[i][circle]);

        return;

    }

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

        vector<int> ans;

        if(matrix.size()==0||matrix[0].size()==0)

            return ans;

        int circle=0,rows=matrix.size(),cols=matrix[0].size();

        while(rows>2\*circle&&cols>2\*circle)//边长为5或者6的时候，都要打印2圈；边长为7的时候就要3次了，式子成立；

            printCircle(rows,cols,circle++,matrix,ans);

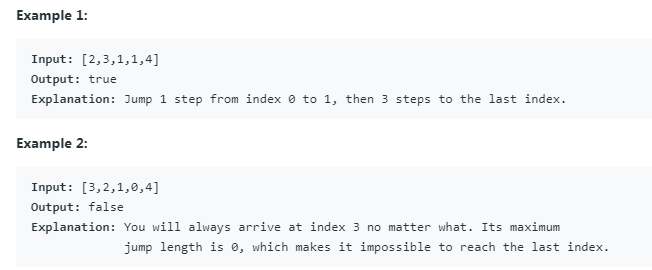
        return ans;

    }

};

第五十五题：

给定一个数组，每一个元素表示可以从这一个元素跳跃的距离，返回是否可以到达数组的最后一个元素。



有一个思路很清晰的写法：用一个循环上限不断更新的for循环，其上限为当前这一步能到达的最远距离m，然后遍历这个数组，并不断更新能到达的最远距离m。如果上限m大于等于nums.size()说明可以到达末尾，否则不能到达。遍历了一次整个数组，所以时间复杂度为O（n）。

代码：

class Solution {

public:

bool canJump(vector<int>& nums)

{

int m = 0;

for (int i = 0; i < nums.size() && i <= m && m < nums.size() - 1; ++i)

m = max(m, i + nums[i]);

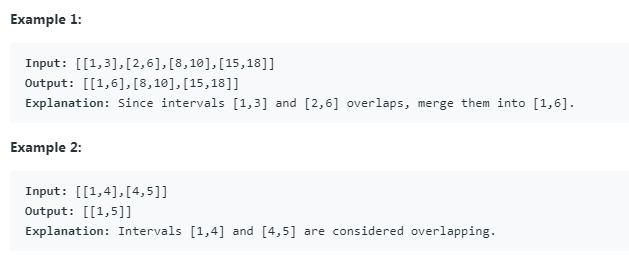
return m >= nums.size() - 1;

}

};

第五十六题：

给定一个二维数组，每个元素表示一个区间，将重叠的区间合并，返回合并后的数组。



首先将所有区间按照区间左边界从小到大排列，然后检查每一个区间，如果下一个区间的左边界小于等于现在区间的边界，则合并；当下一个区间的左边界比之前累加的区间的右边界大的时候，说明没有重叠了，继续检查后面的区间。首先排序需要nlogn的时间，然后检查每一个区间，需要n的时间，所以总的时间复杂度为O（nlogn）。

代码：

vector<vector<int>> merge(vector<vector<int>>& intervals) {

int n = intervals.size();

if(n==0) return intervals;

sort(begin(intervals), end(intervals),

[](const vector<int>& a, const vector<int>& b){ return a[0] < b[0]; });

//这里用到了一个匿名函数，学习一下

vector<vector<int>> res;

res.push\_back(intervals[0]);

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

if(intervals[i][0] <= res.back()[1]) {

// 重叠，更新右边界

res.back()[1] = intervals[i][1];

} else {

// 没有重叠，检查下一个区间

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

}

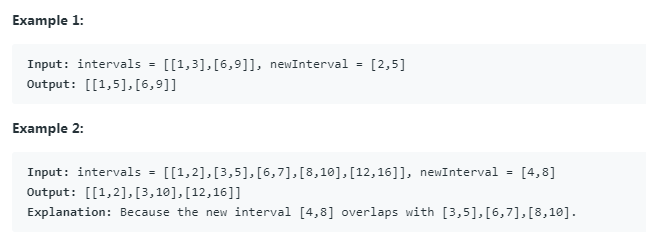
}

return res;

}

第五十七题：

给定一个非负的区间数组intervals和一个新区间newinterval，将这个新区间插入数组，并且合并重叠的区间。



用两个指针l，r来表示intervals中与newInterval区间重叠的元素范围：

1. l<len&&intervals[l][1]<newInterval[0] 表示l元素与newInterval不重叠，l++；

2. r<len&&intervals[r][0]<=newInterval[1] 表示r元素与newInterval重叠，r++；

3. 综上所述，重叠的元素范围为[l,r）;

4. 当l==r时，说明没有元素重叠，此时直接插入；

5. 当l！=r时，先去掉重叠的元素，再插入合并之后的元素；

遍历了一遍区间数组，所以时间复杂度为O（n）。

代码：

class Solution {

public:

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

int l=0,r=0,len=intervals.size();

while(l<len&&intervals[l][1]<newInterval[0])l++;

while(r<len&&intervals[r][0]<=newInterval[1])r++;

if(l<r){

vector<int> temp={min(intervals[l][0],newInterval[0]),max(intervals[r-1][1],newInterval[1])};

intervals.erase(intervals.begin()+l,intervals.begin()+r);

intervals.insert(intervals.begin()+l,temp);

}

else

intervals.insert(intervals.begin()+l,newInterval);

return intervals;

}

};

第五十八题：

给定一个字符串，返回最后一个词的长度。



思路是先去掉字符串后面的空格，然后用一个cnt记录当前单词的长度，直到遇到一个空格，返回cnt。最坏情况下遍历一遍整个s，所以时间复杂度为O（n）。

代码：

class Solution {

public:

int lengthOfLastWord(string s) {

int len=s.size()-1,cnt=0;

while(len>=0&&s[len]==' ')

len--;

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

len--,cnt++;

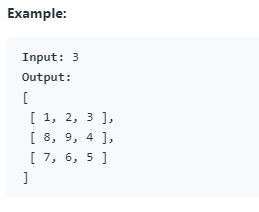
return cnt;

}

};

第五十九题：

给定一个正整数n，以螺旋顺序从1到n\*n填满n\*n的矩阵并返回。



以54题类似，初始化l=t=0，r=d=n，cnt=1，dec=0。其中dec表示现在填充的是行或者列，cnt表示现在填充的数字，当cnt>n\*n时填充完毕，填充顺序按照cnt递增的顺序填充。

代码：

class Solution {

public:

vector<vector<int>> generateMatrix(int n) {

vector<vector<int>> res(n,vector<int>(n,0));

int t=0,l=0,d=n,r=n,cnt=1,dec=0;

while(cnt<=n\*n){

if(dec%4==0){

for(int i=l;i<r;++i)

res[t][i]=cnt++;

t++;

}

else if(dec%4==1){

for(int i=t;i<d;++i)

res[i][r-1]=cnt++;

r--;

}

else if(dec%4==2){

for(int i=r-1;i>=l;--i)

res[d-1][i]=cnt++;

d--;

}

else if(dec%4==3){

for(int i=d-1;i>=t;--i)

res[i][l]=cnt++;

l++;

}

dec++;

}

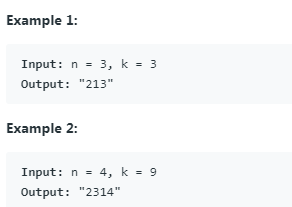
return res;

}

};

第六十题：

给定两个正整数n和k，n表示n位数字全排列，k表示字典序的第k个，返回全排列字典序的第k个数。



思路是从最高位开始判断每一位的数字是多少，当n=3时，n阶乘=6，所以1到3开头的数字分别有n！/n=2个，因此字典序的第三个应该是2开头，用k=3减去每一组的数字个数2，为1，说明是2开头这一组数字的第一个数字，所以是21，最后一个数字显然是3，答案为213。用ans来记录每一位是当时剩余数字中的第几个数字，wh记录剩余可用的数字。生成wh需要n的时间，计算ans需要n的时间，生成答案ans需要n的时间，所以总的时间复杂度为O（n）。

代码：

class Solution {

public:

string getPermutation(int n, int k) {

int divide=1,len=n;

vector<int> ans;

string res,wh;

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

divide\*=i;

wh+=to\_string(i);

}

while(len){

int temp=(k-1)/(divide/len);

ans.push\_back(temp);

k-=temp\*divide/len;

divide/=len;

len--;

}

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

res+=wh[ans[i]];

wh.erase(ans[i],1);

}

return res;

}

};

第六十一题：

给定一个链表，和正整数k，要求从倒数第k个位置开始接在链表最前方，返回链表头。



首先，这个k可以很大，所以最终取倒数第n=k%len（链表长度）个接在链表前就行，思路就是先得到链表长度len，然后求出n，用第一个指针f向前n步，然后第二个指针r指向head，两个指针同时向前移动直到到达链表末尾，然后将链表首尾连接，用head指向r->next，然后断开r和head之间的链接，返回head。所以要遍历链表两次，时间复杂度为O（n）。

代码：

/\*\*

\* Definition for singly-linked list.

\* struct ListNode {

\* int val;

\* ListNode \*next;

\* ListNode(int x) : val(x), next(NULL) {}

\* };

\*/

class Solution {

public:

ListNode\* rotateRight(ListNode\* head, int k) {

if(head==NULL) return NULL;

int len=1;

ListNode \*curr=head;

while(curr->next){

len++;

curr=curr->next;

}

//连接首尾

curr->next=head;

int step=len-k%len;

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

curr=curr->next;

}

ListNode \*newhead=curr->next;

curr->next=NULL;

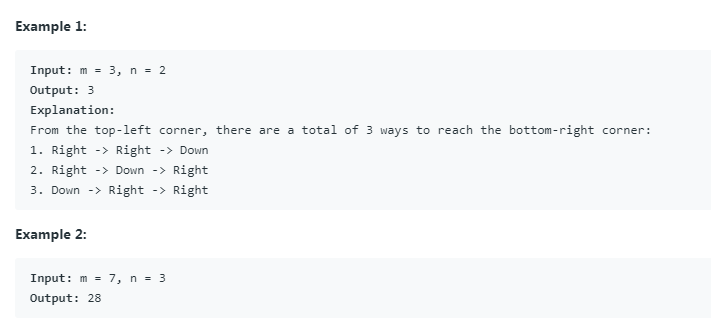
return newhead;

}

};

第六十二题：

给定一个m\*n的矩阵，从[0,0]出发到达[m-1，n-1]，只能往右或者往下，返回有多少种走法。



思路是动态规划。如果使用模拟的方法因为每个地方会有两种走法，所以时间复杂度为2^(m\*n)，肯定会超时。所以使用动态规划，如果从右下角出发，因为只能往上或者往左，所以m-1这一行和n-1这一列必然全部都是只有一种走法。注意到[i，j]的走法为[i，j+1]和[i+1，j]走法的和，易得当i<m-1且j<n-1时动态转移方程dp[i，j]=dp[i+1，j]+[i，j+1]，其他情况dp[i，j]=1，最后返回dp[0][0]，此时时间复杂度为O(mn)。

代码：

class Solution {

public:

int uniquePaths(int m, int n) {

vector<vector<int>> category(m,vector<int>(n,1));

for(int i=m-2;i>=0;--i)

for(int j=n-2;j>=0;--j)

category[i][j]=category[i+1][j]+category[i][j+1];

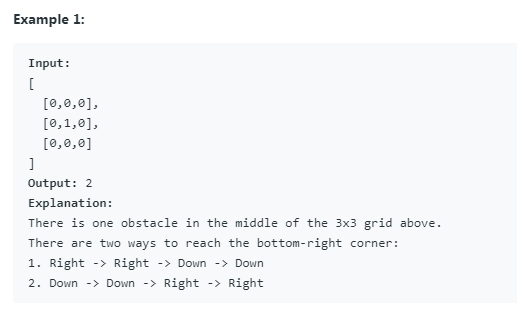
return category[0][0];

}

};

第六十三题：

与上一题类似，但是在矩阵中加入了障碍，这次直接给定m\*n的矩阵，当元素==1时为障碍。



思路还是动态规划，不过假设终点和起点换位，初始化m\*n二维数组全部元素为0，当obstacleGrid[i][j]==1时，跳过；当i>0且j>0时动态转移方程dp[i，j]=dp[i-1，j]+[i，j-1]；其他情况dp[i，j]=1，最后返回dp[m-1][n-1]。这里的dp[i，j]可以分为两步，先加dp[i-1，j]再加dp[i，j-1]，这样就可以将最后一行和最后一列统一起来了。填了一个m\*n的表格，所以时间复杂度为O（mn）。

代码：

class Solution {

public:

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

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

vector<vector<long long > > dp(n, vector<long long >(m));

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

for(int j = 0; j < m; j++)

if(nums[i][j]) continue;

else{

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

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

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

}

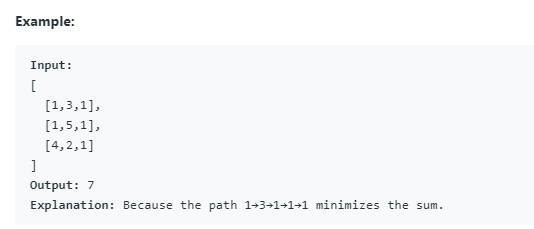
return dp[n - 1][m - 1];

}

};

第六十四题：

给定一个矩阵，每个元素代表这个格子的值，返回从左上角到右下角的路径的最小值。



经典的动态规划，首先，从左上角到右下角跟从右下角到左上角一样，所以计算的时候可以计算从（0,0）到（m-1，n-1）。主要的转移方程为dp[i][j]=grid[i][j]+min(dp[i-1][j],dp[i][j-1])，如果在最左边或者最上边会越界，这里有一个技巧，初始化的矩阵大小为（m+1）\*（n+1），grid中的（0,0）对应矩阵中的（1,1），然后将（0,1）=1就可以全部统一处理了，时间复杂度为O（mn）。

代码：

class Solution

{

public:

int minPathSum(std::vector<std::vector<int>> &grid)

{

std::vector<std::vector<unsigned long long>> paths(grid.size() + 1, std::vector<unsigned long long>(grid[0].size() + 1, INT\_MAX));

paths[0][1] = 0;

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

{

for (int j = 1; j < paths[0].size(); j++)

{

paths[i][j] = grid[i - 1][j - 1] + std::min(paths[i - 1][j], paths[i][j - 1]);

}

}

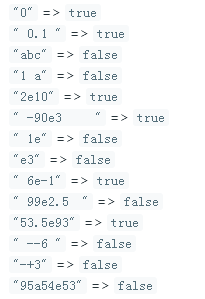
return paths[paths.size() - 1][paths[0].size() - 1];

}

};

第六十五题：

给定一个字符串，判断是否可以转换为十进制数，注意“.2”和“2.”都是合法的，最复杂的合法数字组合是“ -1.2e+3 ”，中间有空格或者其他字符都是错的。



思路为模拟，初始化一个记录下标的ord=0，步骤：①去除前面的所有空格；②判断后面的数字是否是一个合法有符号十进制数；③如果存在小数点，判断小数点后是否是一个合法的无符号十进制数；④如果存在“e”，判断“e”后是否是一个合法的有符号十进制数；⑤去除后面的所有空格；如果此时ord==s.size()，且过程都满足，则合法，扫描整个s一边，所有时间复杂度为O（s.size()）。这里有一个需要注意，表达式a=a||c，当a为true时，就不会判断c，如果要执行c，应该写作a=c||a。遍历一次整个字符串，时间复杂度为O（n）。

代码：

class Solution {

public:

bool checknum(string &s,int &ord){

if(s[ord]=='+'||s[ord]=='-')

ord++;

return checkunnum(s,ord);

}

bool checkunnum(string &s,int &ord){

int pos=ord;

while(ord<s.size()&&isdigit(s[ord]))

ord++;

return ord>pos;

}

bool isNumber(string s) {

int ord=0;

bool flag;

while(ord<s.size()&&s[ord]==' ')

ord++;

flag=checknum(s,ord);

if(s[ord]=='.')

flag=checkunnum(s,++ord)||flag;

if(tolower(s[ord])=='e')

flag=flag&&checknum(s,++ord);

while(ord<s.size()&&s[ord]==' ')

ord++;

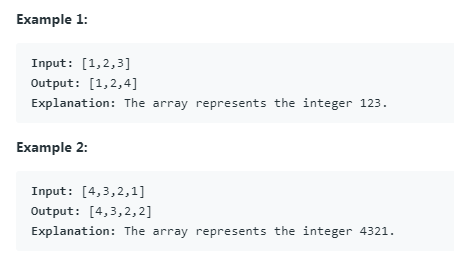
return flag&&ord==s.size();

}

};

第六十五题：

给定一个非空的数组，每个元素都是个位数，数字前面不会是0，除非本身为0，将其加一然后返回。



思路比较简单，从ord=digits.size()-1开始检查，如果为9，置0然后ord--。最后如果ord==-1，说明全是9，那么在前面插入1然后返回，否则digits[ord]++，返回。遍历一次整个数组，时间复杂度为O（n）。

代码：

class Solution {

public:

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

int ord=digits.size()-1;

while(ord>=0&&digits[ord]==9){

digits[ord]=0;

ord--;

}

if(ord<0){

digits.insert(digits.begin(),1);

return digits;

}

else{

digits[ord]++;

return digits;

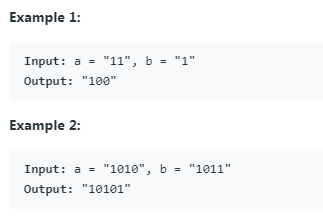
}

}

};

第六十七题：

给定两个string，两个都是二进制数，返回他们的和，用string表示。



思路为初始化一个进位carry=0，计算相同位数和temp，每次添加一个数字（carry+temp）%2，进位carry=（carry+temp）/2。**这里有一个技巧**，如果要计算a+b，可以分为先+a再+b，这样如果其中一个元素越界了，相当于令其为0。遍历一次两个字符串，所以总的时间复杂度为O（n）。

代码：

class Solution {

public:

string addBinary(string a, string b) {

string res;

int i = a.size()-1, j = b.size()-1, carry = 0;

while (i >= 0 || j >= 0) {

int val = carry;

if (i >= 0) val += a[i--] - '0';

if (j >= 0) val += b[j--] - '0';

carry = val / 2;

res.insert(res.begin(), (val % 2 + '0'));

}

if (carry) res.insert(res.begin(), '1');

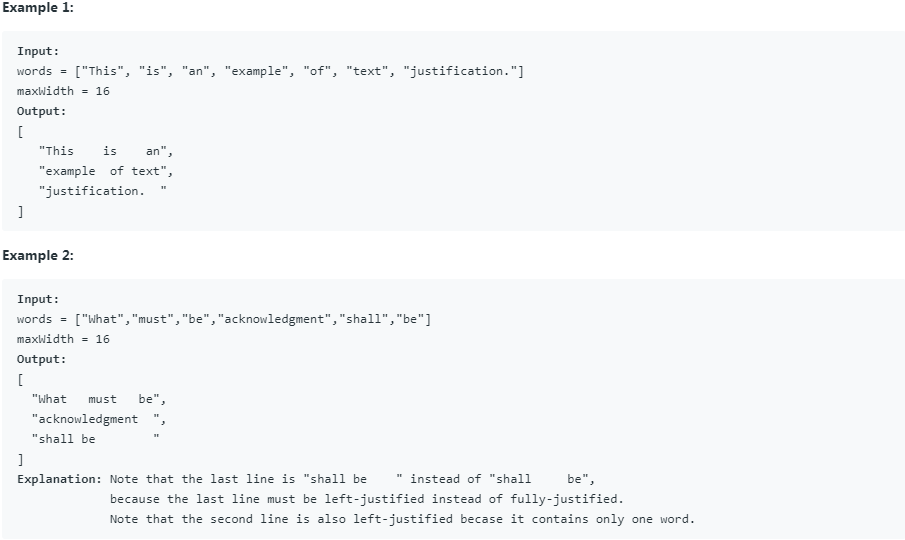
return res;

}

};

第六十八题：

给定一个vector<string>和一个正整数maxWidth，要求将每一行弄成maxWidth这么长，最后一行或者一个单词独占一行的所有单词往左对齐即可。例子如下：



思路是模拟，由于每一行至少有一个单词，所以我们可以初始化measure=words[f].size(),其中f还未分组的第一个单词，r表示下一次分组的第一个单词。①由于每个单词至少间隔一个space，所以比较时要计算measure+words[r]+1,1表示空格；②跳出while后，total表示当前需要填充空格的间隙，同时将计算spaces表示还有多少空格可以用；③这里是重点，分情况讨论：如果当前是最后一行，那么每个单词间隔一个space，其余空格放最后；如果当前只有一个单词，那么所有空格放最后；如果当前不止一个单词，检查spaces%total，如果为0，表示空格可以均分，剩余的所有单词间隔都一致；如果不为0，表示当前空格不能均分，所以每个间隙都多放一个空格，然后更新剩余的spaces和total。因为每个单词都扫描一次，所以时间复杂度为O（n）。

代码：

class Solution {

public:

vector<string> fullJustify(vector<string>& words, int maxWidth) {

vector<string> res;

int f=0;

while(f<words.size()){

int measure=words[f].size(),r=f+1,total;

while(r<words.size()&&measure+words[r].size()+1<=maxWidth)

measure+=words[r++].size()+1;

total=r-f-1,measure-=r-f-1;

string temp=words[f];

if(r==words.size()){

for(int i=f+1;i<r;++i)

temp+=' '+words[i];

temp+=string(maxWidth-temp.size(),' ');

}

else if(total==0)

temp+=string(maxWidth-temp.size(),' ');

else{

int spaces=maxWidth-measure;

for(int i=f+1;i<r;++i){

if(spaces%total==0)

temp+=string(spaces/total,' ')+words[i];

else{

temp+=string(spaces/total+1,' ')+words[i];

spaces-=spaces/total+1;

total--;

}

}

}

res.push\_back(temp);

f=r;

}

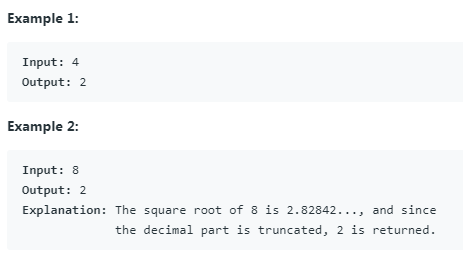
return res;

}

};

第六十九题：

实现根号x方法，要求x向下取整。



思路是二分法。由于向下取整，所以有可能最后结果的平方小于x，因此应该让结果偏左边，这样不会缩小范围的时候去掉正确答案。初始化f=0，r=x，然后计算mid（f+r+1）/2的平方是否小于等于x，如果是，说明太小了或者刚刚好，所以f=mid，否则r=mid-1。由于最后的答案比正确答案稍微要小一点，所以答案往左靠近，因此r需要-1。由于利用二分查找，范围为0到x，所以时间复杂度为O(logn)。

代码：

class Solution {

public:

int mySqrt(int x) {

long long f=0,r=x;

while(r-f>0){

long long mid=(f+r+1)/2;

if(mid\*mid<=x)

f=mid;

else

r=mid-1;

}

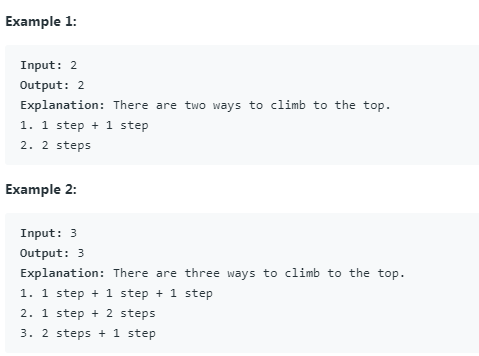
return f;

}

};

第七十题:

给定一个正整数n，你每一次可以走1步或者2步，返回走到n这个位置有多少种走法。



思路很简单，这是一个斐波那契数列，对于三个连续点a，a+1，a+2，到达a+2的不同走法为a的走法加上a+1的走法，即f（a+2）=f（a+1）+f（a）。由于从1计算到n，所以时间复杂度为O(n)。

代码：

class Solution {

public:

int climbStairs(int n) {

int one=1,two=2;

if(n<3)

return n;

else

n-=2;

while(n--){

int temp=one+two;

one=two;

two=temp;

}

return two;

}

};

第七十一题：

**这题重要。**给定一个路径，其中“.”代表当前目录，“..”代表上一层目录，返回简化后的路径。

①思路很重要，首先进入跟退出文件夹就是一个栈！所以要使用一个栈来表示当前的目录，其次顺序很重要，比如a/../b/c与a/b/../c不同，必须按顺序操作，所以应该用一个vector来存储每一个“/”间隔中的目录。②如果对于c++熟悉的话可以用getline（ss，s，“/”）表示从ss中以“/”为间隔读取一个字符串为s，默认是空格。整个过程扫描一遍字符串，然后模拟操作一次stack，最后连接答案，最大为3n，所以时间复杂度为O(n)。

代码①：

class Solution {

public:

string simplifyPath(string path) {

vector<string> wh;

stack<string> st;

int step=0;

while(step<path.size()){

while(step<path.size()&&path[step]=='/')

step++;

string temp;

while(step<path.size()&&path[step]!='/')

temp+=path[step++];

wh.push\_back(temp);

}

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

if(wh[i]=="."||wh[i]=="")

continue;

else if(wh[i]==".."){

if(!st.empty())

st.pop();

}

else

st.push(wh[i]);

}

string res;

while(!st.empty()){

res="/"+st.top()+res;

st.pop();

}

return res==""? "/":res;

}

};

代码②：

class Solution {

public:

string simplifyPath(string path) {

string res, s;

stack<string> stk;

stringstream ss(path);

while (getline(ss, s, '/')) {

if (s == "" || s == ".") {

continue;

}

if (s == ".." && !stk.empty()) {

stk.pop();

}

else if (s != "..") stk.push(s);

}

while (!stk.empty()) {

res = "/" + stk.top() + res;

stk.pop();

}

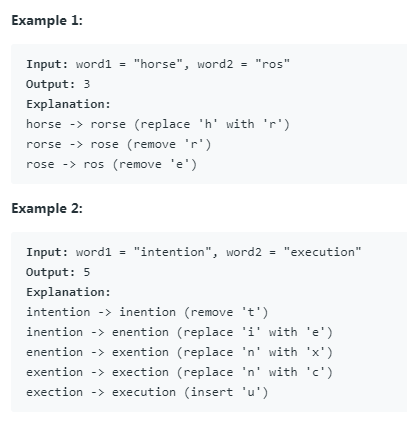
return res.empty() ? "/" : res;

}

};

第七十二题：

**这题重要。**给定两个字符串，可以对任意一个单词进行替换、删除、插入，将两个字符串转换成一样最少需要多少步。



经典的动态规划问题->**编辑距离**，初始化一个大小为（w1.size()+1,w2.size()+2）的二维数组dp，dp[i][j]表示w1[0，i)和w2[0，j)相匹配需要的步数，其中dp[0][j]=j且dp[i][0]=i，因为这时候只能插入。对于任意的dp[i][j]=dp[i-1][j-1]+（w1[i-1]！=w2[j-1]），注意这里比较的下标是i-1和j-1。如果w1[i-1]！=w2[j-1]，说明这个位置需要进行替换操作，所以操作数+1。然后重点来了,如果插入一个字符，对于dp[i][j]而言，给i插入一个字符与j匹配，所以只要[0,i)和[0,j-1)匹配上就行，因此dp[i][j]=dp[i][j-1]+1;如果删除一个字符，那么i字符就不管了，只要[0,i-1)[0,j)匹配上就行，因此dp[i][j]=dp[i-1][j]+1。动态规划需要填充一个二维数组，所以时间复杂度为O(n^2)。

代码：

class Solution {

public:

int minDistance(string word1, string word2) {

int len1=word1.size(),len2=word2.size();

if(len1\*len2==0)

return len1+len2;

vector<vector<int>> dp(len1+1,vector<int>(len2+1,0));

for(int i=1;i<=len1;++i)

dp[i][0]=i;

for(int j=1;j<=len2;++j)

dp[0][j]=j;

for(int i=1;i<=len1;++i)

for(int j=1;j<=len2;++j){

dp[i][j]=dp[i-1][j-1]+(word1[i-1]!=word2[j-1]);

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

}

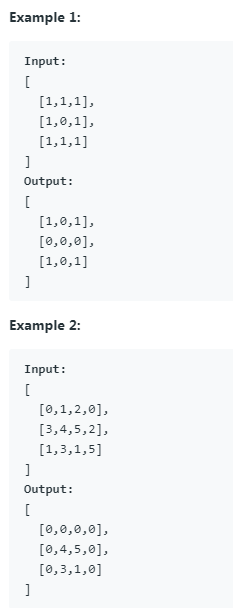
return dp[len1][len2];

}

};

第七十三题：

给定一个n\*m矩阵，如果matrix[i][j]==0，令i行j列全部为0，只能使用常数空间，而且in-place。



思路很简单，难点在于需要常数的额外空间。初始化两个变量，row0=1和col0=1，表示第0行和第0列是否有0，有的话置0。然后从（1,1）开始检查到（n-1，m-1），如果matrix[i][j]==0，令matrix[i][0]=0和matrix[0][j]=0，表示i行j列要全部置0。遍历完了以后检查0行0列，将（i，0）和（0，j）是0的行和列置0，最后检查row0和col0，如果为0就将对应的行或列全部置0。遍历数组三遍，第一遍检查，第二遍行置零，第三遍列置零，总共是3mn，所以时间复杂度为O（mn）。

代码：

class Solution {

public:

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

if (matrix.empty()) return;

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

int col0 = 1, row0 = 1;

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

if (!matrix[i][0]) col0 = 0;

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

if (!matrix[0][i]) row0 = 0;

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

for (int j = 1; j < m; j ++ )

if (!matrix[i][j])

{

matrix[i][0] = 0;

matrix[0][j] = 0;

}

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

if (!matrix[i][0])

for (int j = 1; j < m; j ++ )

matrix[i][j] = 0;

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

if (!matrix[0][i])

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

matrix[j][i] = 0;

if (!col0)

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

matrix[i][0] = 0;

if (!row0)

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

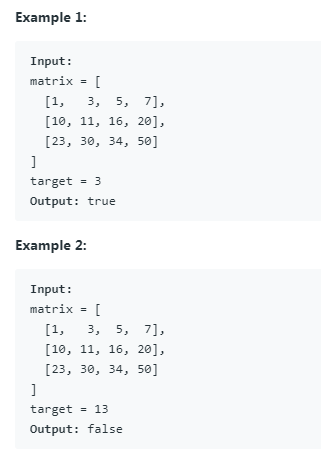
matrix[0][i] = 0;

}

};

第七十四题：

给定一个m\*n的数组，每一行都有序，且每一行第一个元素大于前面所有行的元素。



其实将所有行的元素连接起来就是一个有序的一维数组，元素个数为m\*n。所以最简单的思路就是将其想象成一个一维数组，下标范围为0~（m\*n-1），然后直接二分，注意此时在二维数组中的下标即可。也可以两次二分，第一次找到其所在的行，第二次找到其所在的列就行。方法一就是logmn，方法二是logm+logn=logmn，所以时间复杂度为O（logmn）。

代码：

class Solution {

public:

bool searchMatrix(vector<vector<int>>& matrix, int target) {

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

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

int l = 0, r = n \* m - 1;

while (l < r)

{

int mid = (l + r) / 2;

if (matrix[mid / m][mid % m] >= target) r = mid;

else l = mid + 1;

}

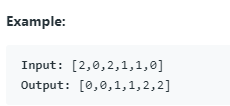
return matrix[r / m][r % m] == target;

}

};

第七十五题：

给定一个数组nums，只包含0,1,2三种元素，将其按照从小到大的顺序排好，要求只遍历一次，并且只使用常数额外空间。



很经典的指针题，设置三个指针，l=0，表示（0，l）所有元素为0；r=nums.size（）-1，表示（r，nums.size（）-1）所有元素为2；step=0表示检查指针。这里需要注意两点：①循环条件为step<=r，由于更新r之后nums[r]并没有检查，所以最后还是需要检查一下；②更新l后step同步更新，更新r后step不能更新。因为[0，l）均为0，[l，step）均为1，所以可以保证交换后nums[step]==1；但是更新r后不知道nums[step]是多少，0,1,2均有可能，所以要再次检查nums[step]。遍历整个数组一次，所以时间复杂度为O（n）。

代码：

class Solution {

public:

void sortColors(vector<int>& nums) {

int l=0,step=0,r=nums.size()-1;

while(step<=r){

if(nums[step]==0)

swap(nums[step++],nums[l++]);

else if(nums[step]==2)

swap(nums[step],nums[r--]);

else

++step;

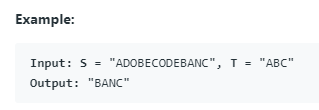
}

}

};

第七十六题：

**这题重要。**给定两个字符串s和t，返回s的一个最短子串，包含了t中所有的字母，要求时间复杂度为O（n）。



这一题的思路很巧妙，首先初始化一个cnt=0，表示t中字母的种类；初始化一个hash表<char，int> wh，表示每一个字母在t中的出现次数。记录完毕之后进入一个for循环，遍历s，同时初始化前指针f，后指针r，此时满足t中字母要求的种类n。每次让wh[s[r]]--，表示f和r之间的子串包含了一个这个字母，通过减小wh[i]的值来标记字符i是否已经满足，当wh[i]==0，说明该字符已经满足要求，此时n++。当n==cnt时，循环检查wh[s[f]]<0，如果满足，说明这个字母多了，可以舍去，令其++并且f++，直到wh[s[f]]==0。因为这里不会重回1，所以不会出现之前满足但是f++之后不满足的情况。最后如果n==cnt，检查r-f+1是否比res.size（）更短，是则更新，否则跳过。因为检查s和t中所有字符一次，所以时间复杂度为O（s.size()+t.size()），即O(n)。

代码：

class Solution {

public:

string minWindow(string s, string t) {

unordered\_map<char,int> wh;

int cnt=0;

string res;

for(auto c:t){

if(!wh[c])

cnt++;

wh[c]++;

}

for(int f=0,r=0,n=0;r<s.size();++r){

if((wh[s[r]]--)==1)

n++;

while(n==cnt&&wh[s[f]]<0)

wh[s[f++]]++;

if(n==cnt&&(res.empty()||res.size()>r-f+1))

res=s.substr(f,r-f+1);

}

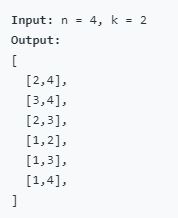
return res;

}

};

第七十七题：

给定两个正整数n和k，返回1~n中k个数所有可能的组合。



这道题目思路很简单，就是搜索。用一个数num记录好现在已经记录的数字的个数，当k==num的时候push就行。有两个点需要注意，第一个是每一次选择的数字要大于前一个数字，避免重复；第二个是遍历的上限是n减去还没有选的数字个数，所以是n-k+num+1。在这n个数里面选择k个数，所以时间复杂度应该是O（C（k n）），即O（n^k）。

代码：

class Solution {

public:

void dfs(int num,int step,int n,int k,vector<vector<int>> &res,vector<int> &temp){

if(num==k){

res.push\_back(temp);

return;

}

for(int i=step;i<=n-k+num+1;++i){

temp.push\_back(i);

dfs(num+1,i+1,n,k,res,temp);

temp.pop\_back();

}

return;

}

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

vector<vector<int>> res;

vector<int> temp;

dfs(0,1,n,k,res,temp);

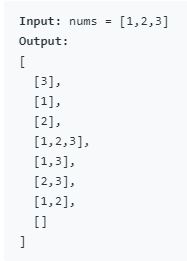
return res;

}

};

第七十八题：

给定一个不重复的数组nums，返回其所有的子集。



思路也是直接搜索，但是这里的搜索不是简单地重复所有长度的搜索，而是一边插入一边变长。可以注意到，由于for循环的起点总是大于前一个进入循环的点，所以最大长度的子集是唯一的，并且所有的中间结果都不会重复，所以利用这个特性解题。全排列的时间复杂度为O（n！）。

代码：

class Solution {

public:

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

vector<vector<int>> output;

vector<int> cur;

backtrack(nums, output, cur, 0);

return output;

}

private :

void backtrack(vector<int>& nums, vector<vector<int>> & output, vector<int>& cur, int begin)

{

output.push\_back(cur);

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

{

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

backtrack(nums, output, cur, i+1);

cur.pop\_back();

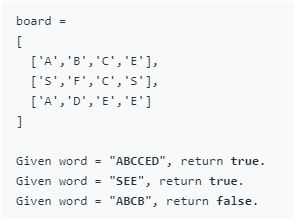
}

}

};

第七十九题：

给定一个二维数组board和一个字符串word，检查word是否存在于board中，注意字符必须水平或者垂直相邻。



思路很简单，就是直接搜索，不过需要初始化一个数组wh，标记已经走过的点防止重复走同一个点。注意边界依次检查上下左右四个方向即可。由于board数组中每个元素检查一次，然后每次除了走过来的方向还有3个方向可以走，字符串长度为k，总体的时间复杂度为O（n^2\*3^k）。

代码：

class Solution {

public:

bool checkit(int num,int i,int j,int len,string &word,vector<vector<char>>& board,vector<vector<int>>& wh){

if(num==len)

return true;

if(i>0&&wh[i-1][j]==0&&board[i-1][j]==word[num]){

wh[i-1][j]=1;

if(checkit(num+1,i-1,j,len,word,board,wh))

return true;

wh[i-1][j]=0;

}

if(j>0&&wh[i][j-1]==0&&board[i][j-1]==word[num]){

wh[i][j-1]=1;

if(checkit(num+1,i,j-1,len,word,board,wh))

return true;

wh[i][j-1]=0;

}

if(i<board.size()-1&&wh[i+1][j]==0&&board[i+1][j]==word[num]){

wh[i+1][j]=1;

if(checkit(num+1,i+1,j,len,word,board,wh))

return true;

wh[i+1][j]=0;

}

if(j<board[0].size()-1&&wh[i][j+1]==0&&board[i][j+1]==word[num]){

wh[i][j+1]=1;

if(checkit(num+1,i,j+1,len,word,board,wh))

return true;

wh[i][j+1]=0;

}

return false;

}

bool exist(vector<vector<char>>& board, string word) {

int len=word.size(),m=board.size(),n;

if(m)

n=board[0].size();

if(m==0||len==0||n==0)

return false;

vector<vector<int>> wh(m,vector<int>(n,0));

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

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

if(board[i][j]==word[0]){

wh[i][j]=1;

if(checkit(1,i,j,len,word,board,wh))

return true;

wh[i][j]=0;

}

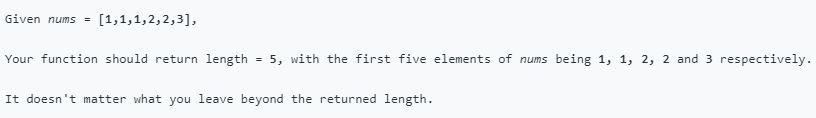
return false;

}

};

第八十题：

给定一个有序数组nums，令相同元素最多为2个，返回调整后的长度，要求in-place，只能使用o（1）额外空间。



这题的思路挺简单的，但是要一次想到确实不容易。解法为定义一个指针k，表示调整后的数组（包含k指向的这个元素）的末尾。所以工作指针i遍历原数组，检查nums[i]是否等于nums[k-2]和nums[k-1]，如果都相等，说明这个元素在新数组中已经有两个了，所以舍去。这里有个小技巧，因为数组有序，所以如果nums[i]！=nums[k-1]，那么一定存在nums[i]!=nums[k-2]，因为nums[k-1]>=nums[k-2]。因此只需要判断nums[i]！=nums[k-2]，则认为i这个元素新数组少于2个，可以插入新数组中。遍历一遍原数组，所以时间复杂度为O（n）。

代码：

class Solution {

public:

int removeDuplicates(vector<int>& nums) {

if (nums.size() < 3)

return nums.size();

int k = 2;

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

if (nums[i] != nums[k - 2])

nums[ k++] = nums[i];

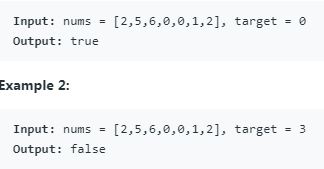
return k;

}

};

第八十一题：

**这题重要。**给定一个不知道旋转轴的旋转数组nums和一个数target，返回target是否在nums中。



这题的思路明显是二分法，但是有技巧。首先二分法找到一个mid，之后必须只取mid左边或者右边，所以必须明确选择方向。首先用mid跟nums[r]比较，判断mid在数值较大的一块还是数值较小的一块，其中如果mid==nums[r]，则r--。因为mid在区间偏左，所以mid跟r不可能指向同一元素，所以mid指向的元素会被保留，不用担心因此错过target。当mid在数值较小的一块时，如果target在mid左边，那么target与mid的大小不好判断，但是如果target在mid右边，那么必然存在target>nums[mid]&&targt<=nums[r]，此时l=mid+1。如果mid在数值较大一侧同理，如果target在mid左边，必然存在target<=nums[mid]&&target>=nums[l]，此时r=mid。当全部元素一样，二分法最坏情况为遍历整个数组，所以时间复杂度为O（n）。

代码：

class Solution {

public:

bool search(vector<int>& nums, int target) {

if(nums.size()==0)

return false;

int l=0,r=nums.size()-1;

while(l<r){

int mid=(l+r)/2;

if(nums[mid]<nums[r]){

if(target>nums[mid]&&target<=nums[r])

l=mid+1;

else

r=mid;

}

else if(nums[mid]>nums[r]){

if(target<=nums[mid]&&target>=nums[l])

r=mid;

else

l=mid+1;

}

else

r--;

}

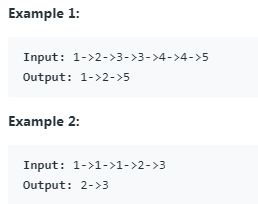
return target==nums[l];

}

};

第八十二题：

给定一个有序链表，如果某元素有重复，则删除所有这个元素，返回链表头。



这题的思路挺简单的，但是需要注意有一个技巧，因为链表头开始就可能需要删除，所以定义一个不会冲突的临时链表头dummy指向head，最后返回dummy->next会简单很多。

初始化一个工作指针p=dummy，一个遍历指针q=head。利用q遍历，使得q->next是下一个不重复的元素。如果此时q==p->next(是这个节点，不是val)，说明q的这个元素可以保留，q和p同时前进一步；否则q及p到q之间是重复元素，令p->next=q->next；重复直到q为空。遍历整个链表一次，所以时间复杂度为O（n）。

代码：

class Solution {

public:

ListNode\* deleteDuplicates(ListNode\* head) {

ListNode \*dummy = new ListNode(0);

dummy->next = head;

ListNode \*p = dummy, \*q = head;

while(q){

if(q->next && q->val == q->next->val){

q = q->next;

}

else{

if(p->next!=q){

q = q->next;

p->next = q;

}

else{

p = p->next;

q = q->next;

}

}

}

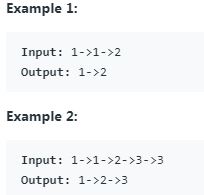
return dummy->next;

}

};

第八十三题：

给定一个有重复的有序链表，要求每个元素最多出现一次，删除多余元素后返回链表头。



这题思路很简单了，初始化一个f=head，r=head。首先用temp记录r的数值，然后让r指向下一个不同的数值，如果此时r->val==temp，说明到达链表末端了，f指向r->next，否则指向r，每次完毕之后f=f->next。

代码：

class Solution {

public:

ListNode\* deleteDuplicates(ListNode\* head) {

if(head==NULL||head->next==NULL)

return head;

ListNode \*f=head,\*r=head;

while(r->next){

int temp=r->val;

while(r->next&&r->val==temp)

r=r->next;

if(r->val==temp)

f->next=r->next;

else

f->next=r;

f=f->next;

}

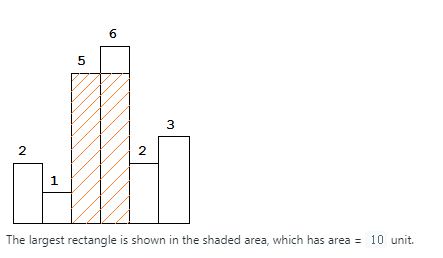
return head;

}

};

第八十四题：

**这题重要。**给定一个非负数组heights，返回其中组成的面积最大的长方形的面积。



这题思路非常难想，但是提到这种题的一种思路是单调栈（栈中元素是单调的，即栈顶是整个栈的最大值或者最小值）。首先为了方便计算，在最后插入一个-1。然后将元素入栈，如果下一个元素heights[i]>heights[st.top()],说明面积还有可能继续增大，所以不断入栈。直到遇到一个heights[i]<heights[st.top()]时停止，然后利用temp记录heights[st.top()]，并pop；此时temp这个数是（st.top（），i）这个区间中的最小值，其长度为i-st.top()-1，所以更新答案res=max（res，heights[temp]\*(i-st.top()-1)）；如果此时栈为空，说明此时是i之前所有元素的最小值，st.top()应该是-1，所以res=max（res，heights[temp]\*i）；然后不断循环直到heights[st.top()]<=height[i]或者栈空了，则入栈。比如弹出5的时候，i=4,height[i]=2,height[st.top()]=1,且5是（st.top()，i）即（1,4）区间内的最小值，所以此时计算的面积为heights[temp]\*(i-st.top()-1)=5\*（4-1-1）=10。最后遇到-1的时候会全部出栈并计算此时的最大面积。所有元素都入栈并出栈一次，所以总体时间复杂度为O（n）。

还有另外一种思路：对于当前的柱子i，初始化l=r=i，然后检查heights[i]<=heights[l-1]和heights[i]<=heights[r+1]，当l和r不越界并且高度比i大的时候扩张，最后计算面积area=heights[i]\*(r-l+1)，然后更新。这种方法的当全部高度一样的时候，每一次都要计算整个数组的面积，所以时间复杂度为O（n^2）。

代码：

class Solution {

public:

int largestRectangleArea(vector<int>& heights) {

int len=heights.size(),res=0;

heights.push\_back(-1);

stack<int> st;

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

while(!st.empty()&&heights[i]<heights[st.top()]){

int temp=st.top();

st.pop();

if(st.empty())

res=max(res,heights[temp]\*i);

else

res=max(res,heights[temp]\*(i-st.top()-1));

}

st.push(i);

}

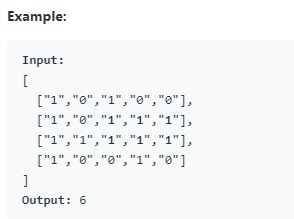
return res;

}

};

第八十五题：

给定一个只含有0和1的矩阵matrix，返回其中由1组成的最大矩形的面积。



思路其实跟上一题一样，以每一行作为底部来计算此时的最大面积。如果当前在第i行，如果matrix[i][j]==0，则高度清零，如果为1则高度为i-1行+1，以此类推，最终返回最大面积。整个矩阵的元素都要进栈一次，所以总体的时间复杂度为O（m\*n）。

代码：

class Solution {

public:

int maximalRectangle(vector<vector<char>>& matrix) {

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

if(n==0)

return 0;

m=matrix[0].size();

vector<int> wh(m+1,0);

wh[m]=-1;

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

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

if(matrix[i][j]=='0')

wh[j]=0;

else

wh[j]++;

}

stack<int> st;

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

while(!st.empty()&&wh[j]<wh[st.top()]){

int temp=wh[st.top()];

st.pop();

if(st.empty())

res=max(res,temp\*j);

else

res=max(res,temp\*(j-st.top()-1));

}

st.push(j);

}

}

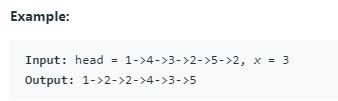
return res;

}

};

第八十六题：

给定一个链表和一个数x，要求将比x小的数全部放在大于等于x的数前面，其余数的相对位置不能改变。



思路很简单，初始化两个dummy1和dummy2，用f和r分别指向这个两个链表头，dummy1为比x小的链表，dummy2为大于等于x的链表。循环结束后dummy1的末端f指向dummy2的第一个元素即可，不过要注意dummy2的末端r必须指向NULL，否则链表会出现闭环。遍历所有元素一遍，时间复杂度为O（n）。

代码：

class Solution {

public:

ListNode\* partition(ListNode\* head, int x) {

ListNode \*dummy1=new ListNode(-1),\*dummy2=new ListNode(-1);

ListNode \*f=dummy1,\*r=dummy2,\*step=head;

while(step){

if(step->val<x)

f->next=step,f=f->next;

else

r->next=step,r=r->next;

step=step->next;

}

r->next=NULL;

f->next=dummy2->next;

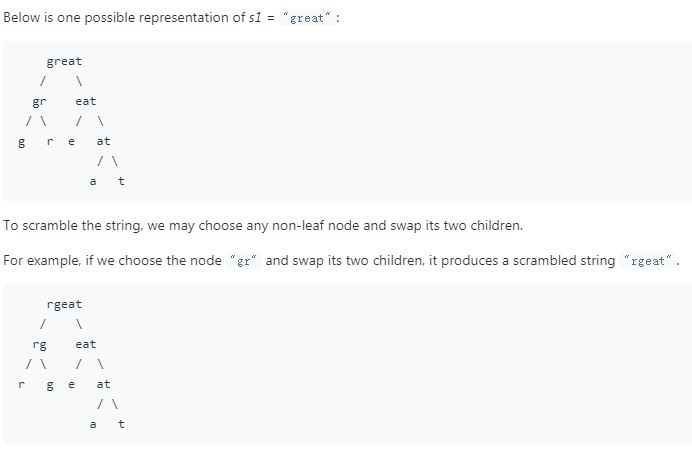
return dummy1->next;

}

};

第八十七题：

给定两个字符串s1和s2，通过以下介绍的方法可以将字符串进行打乱，大概意思就是，任意非叶子节点可以交换两个子节点，然后拼凑成新的字符串，返回s1和s2是否可以相互转化。



这题一开始没什么思路，其实这里面有一个隐藏的特性非常有用：初始字符串和通过任何交换打乱之后的字符串中，其任意长度的子串的所有字母仍然是连在一起的。比如，great不管怎么打乱，eat三个字母还是连在一起的，如grate、graet等，而不会出现eagrt这种情况。所以思路就是：如果没有进行交换，比较s1的前i位与s2的前i位和s1的后面和s2的后面是否同时由相同字母组成；如果进行了交换，比较s1的前i位和s2的倒数i位和s1的后面和s2的前面是否同时由相同字母组成。只要有一种情况成立就说明s1和s2能相互转化。这一题的复杂度比较难计算，首先对于长度为n的两个字符串，要比较四次，两次长度为i，两次长度为n-i，并且要对i=1到n-1进行求和。然后通过an-a（n-1）得到递归通项，所以最终的时间复杂度为O（5^n）。





代码：

class Solution {

public:

bool isScramble(string s1, string s2) {

if(s1==s2)

return true;

string subs1=s1,subs2=s2;

sort(subs1.begin(),subs1.end());

sort(subs2.begin(),subs2.end());

if(subs1!=subs2)

return false;

for(int i=1;i<s1.size();++i){

if(isScramble(s1.substr(0,i),s2.substr(0,i))&&isScramble(s1.substr(i),s2.substr(i)))

return true;

if(isScramble(s1.substr(0,i),s2.substr(s2.size()-i))&&isScramble(s1.substr(i),s2.substr(0,s2.size()-i)))

return true;

}

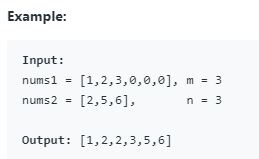
return false;

}

};

第八十八题：

给定两个有序数组及其长度，要求返回两个数组合并之后的有序数组，其中nums1的空间肯定足够大。



这一题的思路很简单，从后面开始插入，即比较nums1和nums2最后的元素，按从大到小排列，依次从m+n这个位置放到0，这种时间复杂度为（m+n），即O（n）。

代码：

class Solution {

public:

void merge(vector<int>& nums1, int m, vector<int>& nums2, int n) {

--m;

--n;

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

if (nums1[m] >= nums2[n]) {

nums1[m + n + 1] = nums1[m];

--m;

} else {

nums1[m + n + 1] = nums2[n];

--n;

}

}

while (n >= 0) {

nums1[n] = nums2[n];

--n;

}

}

};

第八十九题：

格雷码是一种相邻两个数字的二进制表示只有一位不同的序列，给定格雷码的长度n，返回格雷码序列。



这一题可以说就是有固定解法的吧，后面找到一个规律：比如n=2的时候是0132，那么n=3的时候是01326754，前4位是一样的，后四位是0132的镜像即2310加上4。然后n=4的时候就是8加上n=3的时候的镜像。答案的做法是i^(i>>1)，即i与i右移1位做异或操作，i的范围是0到pow（2，n）-1。因为最后序列的长度是2^n,所以时间复杂度就是O（2^n）。

代码：

class Solution {

public:

vector<int> grayCode(int n) {

vector<int> res{0};

int judge=1;

while(n--){

for(int i=res.size()-1;i>=0;--i)

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

judge\*=2;

}

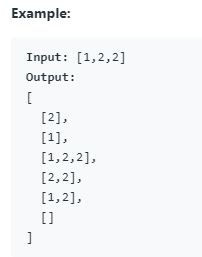
return res;

}

};

第九十题：

给定一个有重复未排序的数组nums，返回它的所有子集，不能重复。



思路跟之前的那道题差不多，就是遍历的同时push\_back就行。但是这个数组是有重复元素的，比如nums[1,4,4]，那么在长度为2的子集中，有两个[1,4]会造成重复。其实可以想到，只要nums[i]==nums[i-1]的时候跳过就行了，但是这么简单跳过的话[1,4,4]就不能入队了。可以留意到，重复是在相同长度的子集中比较的，当长度为2的时候，一开始的[1,4]可以入队，但是后面的4组成的[1,4]不能入队，所以当指针i！=step的情况下，nums[i]==nums[i-1]跳过就行了。遍历了所有子集一遍，所以时间复杂度是O（2^n）。

代码：

class Solution {

public:

void dfs(int step,vector<int> &nums,vector<int> &temp,vector<vector<int>> &res){

if(find(res.begin(),res.end(),temp)==res.end())

res.push\_back(temp);

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

if(i!=step&&nums[i]==nums[i-1])

continue;

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

dfs(i+1,nums,temp,res);

temp.pop\_back();

}

return;

}

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

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

vector<int> temp;

vector<vector<int>> res;

dfs(0,nums,temp,res);

return res;

}

};

第九十一题：

给定一个非空字符串s，规定A=1....Z=26，返回给定的s能翻译成多少种不同的序列。



这一题是一道变种斐波那契数列，首先规定当前检测的位置i，i-2的位置种类为f，i-1位置为r，里面的数字有以下几种情况：①开头为0，或者i和i-1两个位置组成的数字大于26且i位置为0，直接返回0；②组成的数字大于0小于10，因为08这种不成立，所以此时的种类为r；③组成的数字大于等于10且小于等于26，但是i位置为0，因为20这种不能拆开，所以种类为f；④组成的数字大于26，但是i位置不为0，此时必须分开，所以种类为r；⑤组成的数字大于等于10且小于等于26且i位置不为0，可以拆可以不拆，种类为f+r。所有数字都检查一次，所以时间复杂度为O（n）。

代码：

class Solution {

public:

int numDecodings(string s) {

if(s.size()==0||s[0]=='0')

return 0;

if(s.size()==1)

return 1;

int f=1,r=1,w=1;

while(w<s.size()){

int temp;

if((stoi(s.substr(w-1,2))>26&&s[w]=='0')||stoi(s.substr(w-1,2))==0)

return 0;

else if(stoi(s.substr(w-1,2))>26||stoi(s.substr(w-1,2))<10)

temp=r;

else if(stoi(s.substr(w-1,2))<=26&&s[w]=='0')

temp=f;

else

temp=f+r;

f=r,r=temp,w++;

}

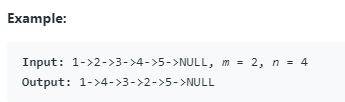
return r;

}

};

第九十二题：

给定一个链表和两个整数m和n，将m到n的元素翻转，然后返回链表头。



思路就是找到m这个节点，然后先把m到n的节点翻转，然后调整m前面和m这两个节点就可以了。由于遍历了一遍0到n的所有节点，最坏情况下所有节点都要翻转，所以时间复杂度为O（n）。

代码：

/\*\*

\* Definition for singly-linked list.

\* struct ListNode {

\* int val;

\* ListNode \*next;

\* ListNode(int x) : val(x), next(NULL) {}

\* };

\*/

class Solution {

public:

ListNode\* reverseBetween(ListNode\* head, int m, int n) {

if(m==n)

return head;

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

dummy->next=head;

ListNode \*r,\*f,\*mid,\*h=dummy,\*t=head;

n-=m;

while(--m)

t=t->next,h=h->next;

r=t,f=r->next,mid=r->next;

while(n--){

f=f->next;

mid->next=r;

r=mid,mid=f;

}

t->next=f,h->next=r;

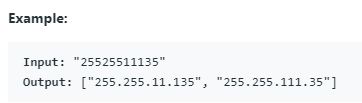
return dummy->next;

}

};

第九十三题：

给定一个字符串s，返回s能组成的所有合法的IP地址。



思路就是搜索，在所有字符的间隔中找到三个间隔插入“.”组成IP地址，每个部分范围为[0,255]。设定每一部分长度为1到3，然后记录已经使用的数字，当分配了四个部分之后再入队。情况有很多，重要的是剪枝部分：①由于入队的时候才去掉最后一个点，只有全部数字已经使用并且插入了4个点的序列才是有效的；②如果地址长度大于1，但是首数字为0那是无效的；③如果剩余的数字除以剩余要分配的部分大于3，那也是无效的；④如果分配的这部分数字大于255，那也是无效的。因为在所有n-1个间隔中找三个间隔插入点，所以最坏情况下为C（3，n-1），时间复杂度为O（n^3）。

代码：

class Solution {

public:

void dfs(int remain,int len,string &s,string &temp,vector<string> &res){

if(remain==0){

if(temp.size()==(s.size()+4))

res.push\_back(temp.substr(0,temp.size()-1));

return;

}

for(int i=s.size()-len,j=1;j<=3&&i+j-1<s.size();++j){

if(j>1&&s[i]=='0'||len/remain>3)

return;

if(stoi(s.substr(i,j))<=255){

temp+=s.substr(i,j)+".";

dfs(remain-1,len-j,s,temp,res);

temp.erase(i+4-remain);

}

}

return;

}

vector<string> restoreIpAddresses(string s) {

vector<string> res;

string temp;

dfs(4,s.size(),s,temp,res);

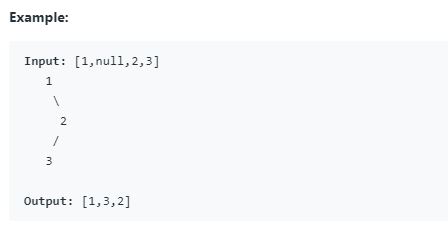
return res;

}

};

第九十四题：

给定一个二叉树，返回其中序遍历的结果，要求使用迭代的方法，不能使用递归。



思路就是压栈，递归用调用栈，迭代使用一个树节点的栈。这是浙大数据结构里的写法，自己的做法是利用pair<TreeNode\*，int>来标记，当int==0表示左节点未入栈，当int==1的时候不再遍历左节点，防止多次遍历左节点。浙大的方法是只有读取的时候才指向栈顶，然后立刻pop，并指向右节点，所以不会多次检查左节点。自己的方法元素在入栈的时候就是中序遍历，浙大的方法不是。由于每个元素入栈出栈一次，所以时间复杂度是O（n）。

代码：

/\*\*

\* Definition for a binary tree node.

\* struct TreeNode {

\* int val;

\* TreeNode \*left;

\* TreeNode \*right;

\* TreeNode(int x) : val(x), left(NULL), right(NULL) {}

\* };

\*/

class Solution {

public:

vector<int> inorderTraversal(TreeNode\* root) {

vector<int> v;

stack<TreeNode\*> s;

TreeNode\* now = root;

while (now || !s.empty()) {

while (now) { // 一直向左并将沿途结点压入堆栈

s.push(now);

now = now->left;

}

if (!s.empty()) { // 每次取出栈顶，访问它，再访问其右子树

now = s.top();

s.pop();

v.push\_back(now->val);

now = now->right;

}

}

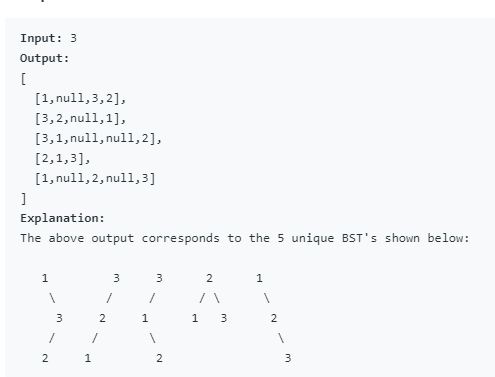
return v;

}

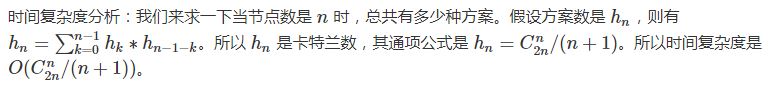
};

第九十五题：

**这题重要。**给定一个整数n，返回1-n可以组成的所有BST（搜索二叉树，中序为有序）。



思路就是分治，以i为现在的根节点，1~（i-1）为左子树，（i+1）~n为右子树。1~（i-1）以同样的方法分治，到最后层层拼接，最后返回根节点的集合。思路不难，关键是实现比较难。与数组不同，不能返回void，因为涉及到左右子树的生成需要返回左右子树的根节点的集合，所以返回类型为vector<TreeNode\*>，最后将左右子树的所有排列方法组合一遍就行，因为左子树的值小于i，右子树的值大于i，且左右子树集合不会重复，所以最后的结果不会重复。假设n个节点的方案数为hn，当以i为根节点，hn（i为根节点）=h（i-1）\*h（n-i）把i=1-n求和就是结果，比如2为根节点n=3，左右都只有一个节点，所以h3（i=2）=h1\*h1。因此时间复杂度如下：



代码：

/\*\*

\* Definition for a binary tree node.

\* struct TreeNode {

\* int val;

\* TreeNode \*left;

\* TreeNode \*right;

\* TreeNode(int x) : val(x), left(NULL), right(NULL) {}

\* };

\*/

class Solution {

public:

vector<TreeNode\*> dfs(int l,int r){

vector<TreeNode\*> res;

if(l>r){

res.push\_back(NULL);

return res;

}

for(int i=l;i<=r;++i){

vector<TreeNode\*> left=dfs(l,i-1),right=dfs(i+1,r);

for(auto &ln : left)

for(auto &rn : right){

TreeNode\* temp=new TreeNode(i);

temp->left=ln;

temp->right=rn;

res.push\_back(temp);

}

}

return res;

}

vector<TreeNode\*> generateTrees(int n) {

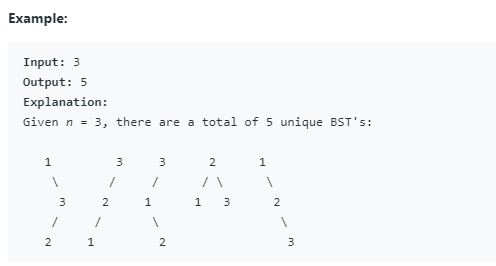
return n==0? vector<TreeNode\*>():dfs(1,n);

}

};

第九十六题：

与上一题类似，给定正整数n，返回一共能生成多少种不同的BST。



如果是利用大数阶乘可以直接计算卡特兰数，如果一步步计算就利用卡特兰数。假设res[i]表示n=i的时候的BST个数。例如n=3，其结果为以i=1,2,3为根节点的BST个数的和，根据BST的特性，当i=1or3的时候，一半没有节点一半两个节点，即res[0]\*res[2]，i=1的时候是res[1]\*res[1]，所以求和即可。因为左子树m种情况与右子树n种情况加起来一共m\*n种情况，所以是乘积。从res[1]计算到res[n]，一共计算n步，所以时间复杂度为O（n）。此外，利用大数相乘也是计算n步，不过涉及到进位处理，所以应该更慢。

代码：

class Solution {

public:

int numTrees(int n) {

vector<int> res(n+1,0);

res[0]=1;

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

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

res[i]+=res[j]\*res[i-j-1];

}

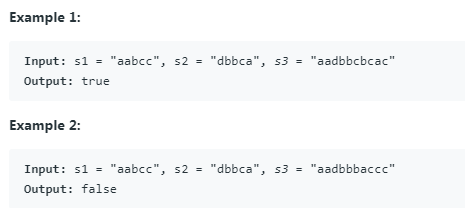
return res[n];

}

};

第九十七题：

**这题重要。**给定三个字符串s1，s2，s3，验证s3是否由s1，s2交织而成。



有三种解法：①自己想的，模拟的办法，初始化一个二维矩阵dp[s1.size][s2.size]。当s1[i]==s3[i+j]，说明s1匹配了，所以dp[i][j]=1，然后i+1；如果s2匹配上则dp[i][j]=2，然后j+1；如果没匹配上，说明到达这个点也是错的，所以令dp[i][j]=-1，然后检查左边和上边哪个不为0就往那个方向走。如果dp[i][j]==1，说明可以检查s2，如果s2也不行就置-1往回。如果dp[i][j]==2，说明s1s2都匹配不上，直接置-1往回。当dp[0][0]==-1的时候就失败了，到达dp[s1.size][s2.size]说明匹配成功。②动态规划。dp[i][j]表示s1前i个字符+s2前j个字符匹配了s3前i+j个字符，最后返回dp[s1.size][s2.size]。③递归，递归的思路也挺好想的，不断递归检查当前s1的i字符，s2的j字符，s3的k字符，哪个匹配上了，就和k一起加一。直到i或者j检查完了，就将另外一个字符串剩余字符与s3剩余字符比较。代码①太长就不放出了，动态规划填一个二维的矩阵，所以时间复杂度为O（n^2）。

代码②：

class Solution {

public:

bool isInterleave(string s1, string s2, string s3) {

int n = s1.size(), m = s2.size(), k = s3.size();

if (k != n + m) return false;

vector<vector<int>> f =

vector<vector<int>>(n + 1, vector<int>(m + 1));

f[0][0] = 1;

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

f[i][0] = f[i - 1][0] && s1[i - 1] == s3[i - 1];

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

f[0][i] = f[0][i - 1] && s2[i - 1] == s3[i - 1];

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

for (int j = 1; j <= m; j ++ )

{

f[i][j] = 0;

if (s1[i - 1] == s3[i + j - 1])

f[i][j] |= f[i - 1][j];

if (s2[j - 1] == s3[i + j - 1])

f[i][j] |= f[i][j - 1];

}

return f[n][m];

}

};

代码③：

class Solution {

public:

bool isInterleave(string s1, string s2, string s3)

{

int len1 = s1.size(), len2 = s2.size();

vector<vector<char>> memo(len1, vector<char>(len2, -1));

return helper(s1, 0, s2, 0, s3, 0, memo);

}

bool helper(string &s1, int i, string &s2, int j, string &s3, int k, vector<vector<char>> &memo)

{

int len1 = s1.size(), len2 = s2.size();

if (i == len1) return s2.substr(j) == s3.substr(k);

if (j == len2) return s1.substr(i) == s3.substr(k);

if (memo[i][j] != -1) return memo[i][j];

memo[i][j] = (s1[i] == s3[k] && helper(s1, i + 1, s2, j, s3, k + 1, memo)) ||

(s2[j] == s3[k] && helper(s1, i, s2, j + 1, s3, k + 1, memo));

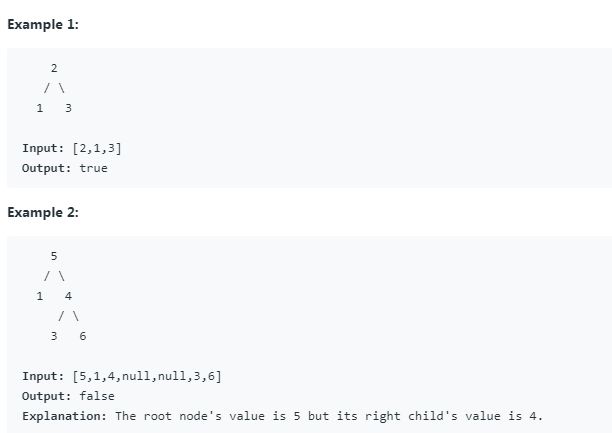
return memo[i][j];

}

};

第九十八题：

给定一个二叉树，判断是不是BST。



思路很简单，就是中序遍历，得到的序列是单调上升序列就行。如果人为设置最小值遇到第一个节点是最小值的时候就会误判，所以应该另外初始化一个空指针，当第一次遇到节点的时候就指向那个节点，等找到第二个节点的时候才开始比较。这里需要注意，当now取得一个节点的时候要先pop，否则judge指向now之后再pop的话容易出现指向空内存错误。遍历所有节点一次，时间复杂度为O（n）。

代码：

/\*\*

\* Definition for a binary tree node.

\* struct TreeNode {

\* int val;

\* TreeNode \*left;

\* TreeNode \*right;

\* TreeNode(int x) : val(x), left(NULL), right(NULL) {}

\* };

\*/

class Solution {

public:

bool isValidBST(TreeNode\* root) {

TreeNode \*now=root,\*judge=NULL;

stack<TreeNode\*> st;

while(now||!st.empty()){

while(now){

st.push(now);

now=now->left;

}

if(!st.empty()){

now=st.top();

st.pop();

if(judge&&now->val<=judge->val)

return false;

judge=now;

now=now->right;

}

}

return true;

}

};

第九十九题：

**这题重要。**给定一棵BST，其中有两个节点交换了，找到并将其恢复，要求使用O（1）的额外空间。



思路是一个叫Morris-traversal 的算法，这个算法可以实现中序遍历。对于一个节点，先判断有没有左节点：①存在左节点，说明其前面还有节点，找到其前驱节点（其左子树的最右下角那个节点）。如果前驱节点的右子树为空（如果未访问过左子树必然为空），则将其right连在cur上，然后进入左子树；如果前驱结点右子树不为空（此时肯定已经访问过左子树了），将其right恢复NULL，读取当前节点，然后进入右子树。②不存在左节点，读取当前节点，然后进入右节点。因为每次进入左节点之前都会找到其前驱节点并连线，所以肯定可以进入右节点，如果本身无右节点则回到其后驱节点，如果本身有右节点则进入右节点。这里需要注意的是，cur遍历的顺序就是一个中序遍历，所以可以读取的时候顺便标记并比较；由于所有的前驱节点与后驱节点都相连了，所以当cur右节点为空的时候就是最后一个节点了。中序遍历一次所有节点，对所有节点找两次前驱（第一次是找到并连线，第二次是找到并发现已经连线），找前驱的时候每条边总共需要遍历2次，所以总共是n+2\*（n-1），时间复杂度为O（n）。

代码：

/\*\*

\* Definition for a binary tree node.

\* struct TreeNode {

\* int val;

\* TreeNode \*left;

\* TreeNode \*right;

\* TreeNode(int x) : val(x), left(NULL), right(NULL) {}

\* };

\*/

class Solution {

public:

void recoverTree(TreeNode\* root) {

TreeNode \*first = NULL, \*second, \*prep = NULL;

while (root)

{

if (!root->left)

{

if (prep && prep->val > root->val)

{

if (!first) first = prep, second = root;

else second = root;

}

prep = root;

root = root->right;//右子树不可能为空的，如果为空那必然连在后驱

}

else

{

TreeNode \*p = root->left;

while (p->right && p->right != root) p = p->right;

if (!p->right)

{

p->right = root;

root = root->left;

}

else

{

p->right = NULL;

if (prep && prep->val > root->val)

{

if (!first) first = prep, second = root;

else second = root;

}

prep = root;

root = root->right;

}

}

}

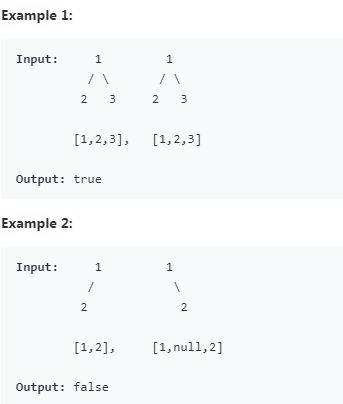
swap(first->val, second->val);

}

};

第一百题：

给定两棵二叉树，判断两棵树是否相同。



思路很简单，同时遍历两棵树，出现不同的时候就返回false，否则返回true。①递归做法，判断当前节点是否同时为空，是则true；只有一个为空就返回false；都不为空则判断val是否相等并同时判断左右子节点。②同层次遍历的方法调用两个栈，当遇到某个节点val不同或者左右子节点一个有一个没有则false，否则入栈，左右节点做法一样。遍历两棵树的所有节点，总共遍历了2n个节点，所以时间复杂度为O（n）。

代码①：

class Solution {

public:

bool isSameTree(TreeNode\* p, TreeNode\* q) {

if (!p && !q) return true;

if (!p && q || !q && p) return false;

return p->val == q->val && isSameTree(p->left,q->left) && isSameTree(p->right,q->right);

}

};

代码②：

/\*\*

\* Definition for a binary tree node.

\* struct TreeNode {

\* int val;

\* TreeNode \*left;

\* TreeNode \*right;

\* TreeNode(int x) : val(x), left(NULL), right(NULL) {}

\* };

\*/

class Solution {

public:

bool isSameTree(TreeNode\* p, TreeNode\* q) {

TreeNode \*t1=NULL,\*t2=NULL;

stack<TreeNode\*> st1,st2;

if(p==NULL&&q==NULL)

return true;

else if((p==NULL&&q)||(q==NULL&&p))

return false;

st1.push(p),st2.push(q);

while(!st1.empty()&&!st2.empty()){

t1=st1.top(),t2=st2.top();

st1.pop(),st2.pop();

if(t1->val!=t2->val)

return false;

if(t1->left&&t2->left)

st1.push(t1->left),st2.push(t2->left);

else if((t1->left&&t2->left==NULL)||(t1->left==NULL&&t2->left))

return false;

if(t1->right&&t2->right)

st1.push(t1->right),st2.push(t2->right);

else if((t1->right&&t2->right==NULL)||(t1->right==NULL&&t2->right))

return false;

}

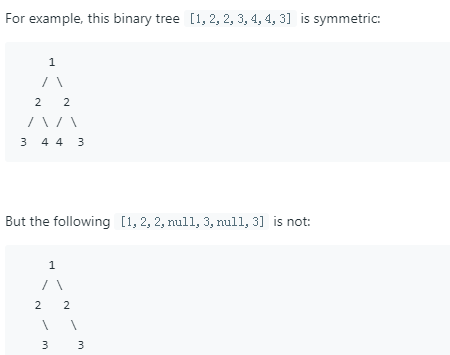
return st1.empty()&&st2.empty();

}

};

第一百零一题：

给定一个二叉树，判断其是否是对称的。



思路是递归检查。首先检查根节点是否为空，空的话直接返回true。如果这课二叉树是对称的，那么我们可以递归地对称检查左子树右节点和右子树左节点&&左子树左节点和右子树右节点，以此类推。遍历一遍整个二叉树，所以时间复杂度为O（n）。

代码：

/\*\*

\* Definition for a binary tree node.

\* struct TreeNode {

\* int val;

\* TreeNode \*left;

\* TreeNode \*right;

\* TreeNode(int x) : val(x), left(NULL), right(NULL) {}

\* };

\*/

class Solution {

public:

bool checkit(TreeNode \*l,TreeNode \*r){

if(l==NULL&&r==NULL)

return true;

else if(l==NULL||r==NULL||l->val!=r->val)

return false;

else

return checkit(l->left,r->right)&&checkit(l->right,r->left);

}

bool isSymmetric(TreeNode\* root) {

if(root==NULL)

return true;

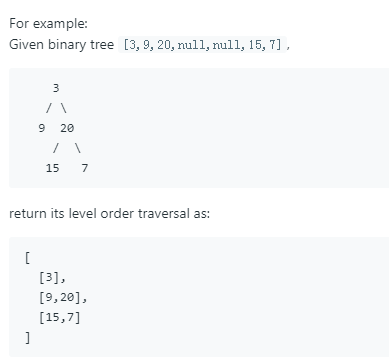
return checkit(root->left,root->right);

}

};

第一百零二题：

给定一棵二叉树，按层次遍历的顺序每一层为一个vector，返回所有的值。



思路是一层一层记录元素。首先将根节点入队，并且设cnt1=1表示当前这一层有多少个节点，cnt2=0表示下一层有多少个节点。每一层声明一个向量temp，当cnt1！=0，从队列的头部取节点进入temp，然后检查左右节点，如果有则cnt2++，并且入队。然后将cnt1=cnt2表示下一层的节点个数，并将cnt2置零。由于每个节点都要进队且出队一次，所以时间复杂度为O（n）。

代码：

/\*\*

\* Definition for a binary tree node.

\* struct TreeNode {

\* int val;

\* TreeNode \*left;

\* TreeNode \*right;

\* TreeNode(int x) : val(x), left(NULL), right(NULL) {}

\* };

\*/

class Solution {

public:

vector<vector<int>> levelOrder(TreeNode\* root) {

vector<vector<int>> res;

int cnt1=1,cnt2=0;

queue<TreeNode\*> qu;

TreeNode \*p=root;

if(root)

qu.push(p);

else

return res;

while(!qu.empty()){

vector<int> temp;

while(cnt1--){

p=qu.front();

qu.pop();

temp.push\_back(p->val);

if(p->left)

cnt2++,qu.push(p->left);

if(p->right)

cnt2++,qu.push(p->right);

}

res.push\_back(temp);

cnt1=cnt2,cnt2=0;

}

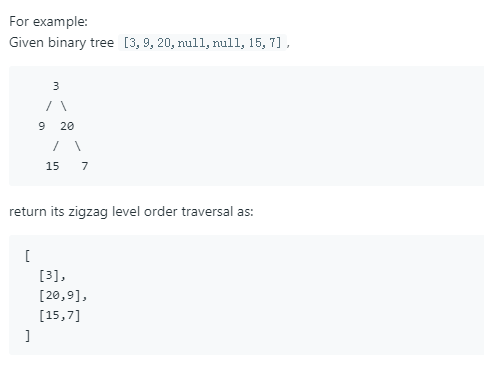
return res;

}

};

第一百零三题：

给定一个二叉树，按照Z字形返回层次遍历的结果。



思路很简单。维护两个向量，line保存当前需要入队的元素，newline保存下厨一次需要入队的元素，同时维护一个jud，用于判断当前是从前往后还是从后往前。以此类推直到line为空。由于每个节点入队出队一次，所以时间复杂度为O（n）。

代码：

/\*\*

\* Definition for a binary tree node.

\* struct TreeNode {

\* int val;

\* TreeNode \*left;

\* TreeNode \*right;

\* TreeNode(int x) : val(x), left(NULL), right(NULL) {}

\* };

\*/

class Solution {

public:

vector<vector<int>> zigzagLevelOrder(TreeNode\* root) {

vector<vector<int>> res;

vector<TreeNode\*> line;

int jud=0;

if(root)

line.push\_back(root);

else

return res;

while(line.size()){

vector<TreeNode\*> newline;

vector<int> temp;

for(auto p:line){

if(jud%2==0)

temp.push\_back(p->val);

else

temp.insert(temp.begin(),p->val);

if(p->left)

newline.push\_back(p->left);

if(p->right)

newline.push\_back(p->right);

}

res.push\_back(temp);

line=newline;

jud++;

}

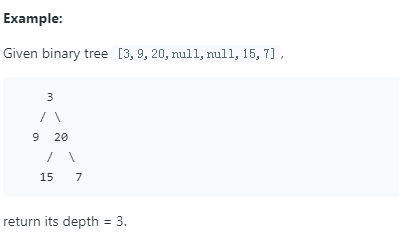
return res;

}

};

第一百零四题：

给定一棵二叉树，返回其最大深度。



思路很简单，就是递归遍历。从根节点开始，如果该节点不为空，则返回其左右子树的最大深度+1，否则返回0。遍历一遍整个二叉树，所以时间复杂度为O（n）。

代码：

/\*\*

\* Definition for a binary tree node.

\* struct TreeNode {

\* int val;

\* TreeNode \*left;

\* TreeNode \*right;

\* TreeNode(int x) : val(x),

left(NULL), right(NULL) {}

\* };

\*/

class Solution {

public:

int maxDepth(TreeNode\* root) {

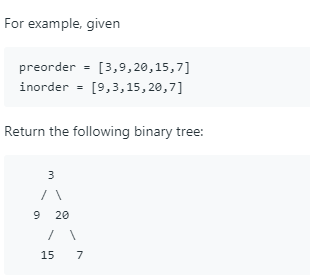
return root ? max(maxDepth(root->left), maxDepth(root->right)) + 1 : 0;

}

};

第一百零五题：

给定一棵二叉树的前序和中序遍历，生成这课二叉树。



思路是递归。因为前序遍历的第一个节点是根节点，只要找到中序遍历中根节点的下标，那么该下标之前的就是根节点的左子树，该下标之后的就是根节点的右子树，同时前序遍历中除了第一个元素之外的元素也自然就分为左右子树。由于要一直查找某节点在中序遍历中的坐标，可以先用unordered\_map将中序遍历的值与下标存储起来，方便查找。每次递归先new一个节点作为当前的根节点，然后其左右子树分别递归，最终返回根节点。现将中序遍历转化为哈希表，然后每个节点都要new一次并连接两次，所以总的时间复杂度为O（n）。

代码：

/\*\*

\* Definition for a binary tree node.

\* struct TreeNode {

\* int val;

\* TreeNode \*left;

\* TreeNode \*right;

\* TreeNode(int x) : val(x), left(NULL), right(NULL) {}

\* };

\*/

class Solution {

public:

unordered\_map<int,int> pos;

TreeNode\* buildTree(vector<int>& preorder, vector<int>& inorder) {

int n = preorder.size();

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

pos[inorder[i]] = i;

return dfs(preorder, inorder, 0, n - 1, 0, n - 1);

}

TreeNode\* dfs(vector<int>&pre, vector<int>&in, int pl, int pr, int il, int ir)

{

if (pl > pr) return NULL;

int k = pos[pre[pl]] - il;

TreeNode\* root = new TreeNode(pre[pl]);

root->left = dfs(pre, in, pl + 1, pl + k, il, il + k - 1);

root->right = dfs(pre, in, pl + k + 1, pr, il + k + 1, ir);

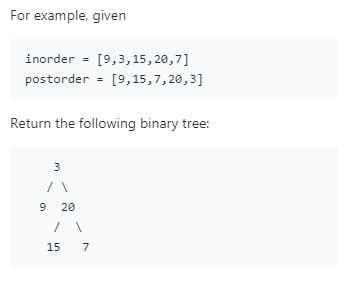
return root;

}

};

第一百零六题：

给定一棵二叉树的中序遍历和后序遍历，生成这课二叉树。



思路很简单，跟上一题一样，后序遍历的最后已给节点就是根节点，然后在中序遍历中找到这个根节点。用这个根节点分成左右子树，后序遍历的前面是左子树，除此之外就是右子树。

时间复杂度为O（n）。

代码：

/\*\*

\* Definition for a binary tree node.

\* struct TreeNode {

\* int val;

\* TreeNode \*left;

\* TreeNode \*right;

\* TreeNode(int x) : val(x), left(NULL), right(NULL) {}

\* };

\*/

class Solution {

public:

unordered\_map<int,int> pos;

TreeNode\* dividebuild(vector<int>& inorder, vector<int>& postorder,int il,int ir,int pl,int pr){

if(il==ir||pl==pr)

return NULL;

TreeNode \*p=new TreeNode(postorder[pr-1]);

int cut=pos[p->val];

p->left=dividebuild(inorder,postorder,il,cut,pl,pl+cut-il);

p->right=dividebuild(inorder,postorder,cut+1,ir,pl+cut-il,pr-1);

return p;

}

TreeNode\* buildTree(vector<int>& inorder, vector<int>& postorder) {

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

pos[inorder[i]]=i;

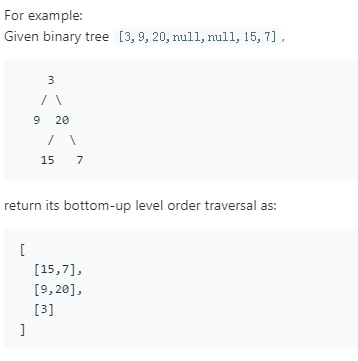
return inorder.size()==0? NULL:dividebuild(inorder,postorder,0,inorder.size(),0,postorder.size());

}

};

第一百零七题：

给定一棵二叉树，返回其自底向上的层次遍历结果。



思路很简单。就是层次遍历入队的时候，在res的最前方不断插入这一层的结果。每一个元素都要入队出队一次，所以时间复杂度为O（n）。

代码：

/\*\*

\* Definition for a binary tree node.

\* struct TreeNode {

\* int val;

\* TreeNode \*left;

\* TreeNode \*right;

\* TreeNode(int x) : val(x), left(NULL), right(NULL) {}

\* };

\*/

class Solution {

public:

vector<vector<int>> levelOrderBottom(TreeNode\* root) {

vector<vector<int>> res;

vector<TreeNode\*> line;

if(root==NULL)

return res;

else

line.push\_back(root);

while(line.size()){

vector<TreeNode\*> newline;

vector<int> temp;

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

if(line[i]->left)

newline.push\_back(line[i]->left);

if(line[i]->right)

newline.push\_back(line[i]->right);

temp.push\_back(line[i]->val);

}

line=newline;

res.insert(res.begin(),temp);

}

return res;

}

};

第一百零八题：

给定一个有序数组，用其生成一棵平衡二叉树。



思路是递归建立二叉树。其实二叉树的问题很多都可以用递归解决，因为二叉树可以看做是很多小二叉树的集合，每一个子节点都可以作为其左右子树的父节点来建立。因为这个数组有序且要建立平衡二叉树，所以中间的节点作为根节点就会自然分成左右子树了，然后不断递归建立。递归的下标范围是（0，nums.size），这里区间的右边是一个不可用的下标，所以我们左子树的范围是l到mid，因为mid这个下标已经用了，不可用；右子树的范围是mid+1到r，因为mid不可用，必须mid+1。数组的每一个元素都建立一个子节点，所以总体时间复杂度为O（n）。

代码：

/\*\*

\* Definition for a binary tree node.

\* struct TreeNode {

\* int val;

\* TreeNode \*left;

\* TreeNode \*right;

\* TreeNode(int x) : val(x), left(NULL), right(NULL) {}

\* };

\*/

class Solution {

public:

TreeNode\* buildtree(vector<int> &nums,int l,int r){

int mid=(l+r)/2;

TreeNode \*root=new TreeNode(nums[mid]);

if(l<mid)

root->left=buildtree(nums,l,mid);

if(mid<r-1)

root->right=buildtree(nums,mid+1,r);

return root;

}

TreeNode\* sortedArrayToBST(vector<int>& nums) {

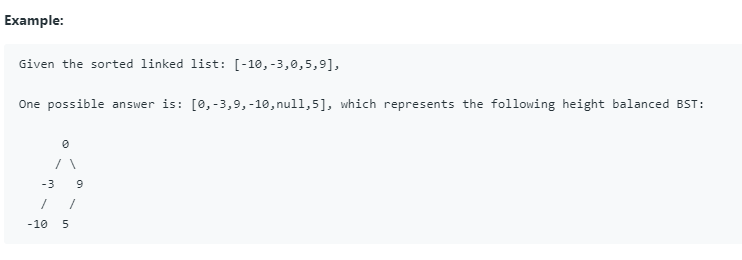
return nums.size()==0? NULL:buildtree(nums,0,nums.size());

}

};

第一百零九题：

**这题重要。**给定一个有序链表，建立一棵平衡二叉树。



思路是递归。首先找到链表的中间节点作为根节点，然后左右链表分别就是左子树和右子树。然后注意了，一开始的给定参数是链表的头结点，所以我们递归的时候也是要给定头结点，所以：左边头结点就是head，但是要在根节点的地方断开；右边头结点是根节点的next，因为根节点已经建立了所以不可用。这里找中间节点可以用快慢指针优化！！！找中间节点时间为1.5n，断开操作0.5n，所以一个节点需要2n；第二层虽然总时间减半为n，但是有两个节点，所以还是2n；总共有logn层，以此类推，时间复杂度为O（nlogn）。

代码：

/\*\*

\* Definition for singly-linked list.

\* struct ListNode {

\* int val;

\* ListNode \*next;

\* ListNode(int x) : val(x), next(NULL) {}

\* };

\*/

/\*\*

\* Definition for a binary tree node.

\* struct TreeNode {

\* int val;

\* TreeNode \*left;

\* TreeNode \*right;

\* TreeNode(int x) : val(x), left(NULL), right(NULL) {}

\* };

\*/

class Solution {

public:

TreeNode\* sortedListToBST(ListNode\* head) {

if (!head) return 0;

int l = 0;

for (auto i = head; i; i = i->next) l ++ ;

l /= 2;

ListNode\*p = head;

for (int i = 0; i < l; i ++ ) p = p->next;

TreeNode \*root = new TreeNode(p->val);

root->right = sortedListToBST(p->next);

if (l)

{

p = head;

for (int i = 0; i < l - 1; i ++ ) p = p->next;

p->next = 0;

root->left = sortedListToBST(head);

}

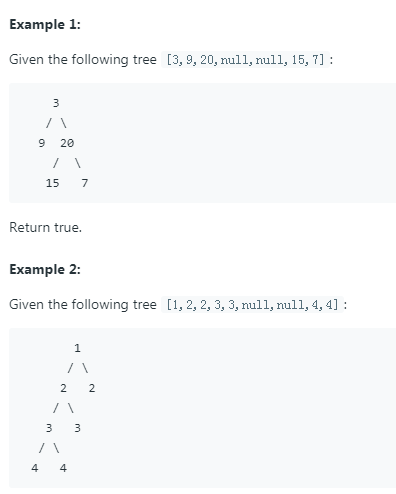
return root;

}

};

第一百一十题：

判断一棵二叉树是否是平衡二叉树。



思路是根据定义递归检查。每次都检查当前节点如果是空的返回true，然后检查左右子树高度差大于1返回false，最后检查当前节点的左右子树。由于每次都要检查左右子树的高度，会重复计算，时间为2logn，有n个节点，所以时间复杂度为O（nlogn）。

代码：

/\*\*

\* Definition for a binary tree node.

\* struct TreeNode {

\* int val;

\* TreeNode \*left;

\* TreeNode \*right;

\* TreeNode(int x) : val(x), left(NULL), right(NULL) {}

\* };

\*/

class Solution {

public:

int caldep(TreeNode\* root){

return root==NULL? 0:max(caldep(root->left),caldep(root->right))+1;

}

bool isBalanced(TreeNode\* root) {

if(root==NULL)

return true;

if(abs(caldep(root->left)-caldep(root->right))>=2)

return false;

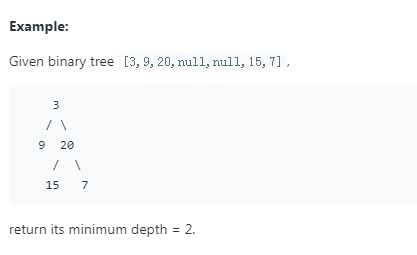
return isBalanced(root->left)&&isBalanced(root->right);

}

};

第一百一十一题：

给定一个二叉树，返回从根节点到叶子节点的最小深度。



思路是递归。先初始化一个res=INT\_MAX，表示这个节点的左右子树中较浅的那一棵的深度，如果这个节点是叶子节点，则返回1。一直递归检查整棵二叉树所有节点，所以时间复杂度为O（n）。

代码：

/\*\*

\* Definition for a binary tree node.

\* struct TreeNode {

\* int val;

\* TreeNode \*left;

\* TreeNode \*right;

\* TreeNode(int x) : val(x), left(NULL), right(NULL) {}

\* };

\*/

class Solution {

public:

int minDepth(TreeNode\* root) {

if (!root) return 0;

int res = INT\_MAX;

if (root->left) res = min(res, minDepth(root->left) + 1);

if (root->right) res = min(res, minDepth(root->right) + 1);

if (res == INT\_MAX) res = 1;

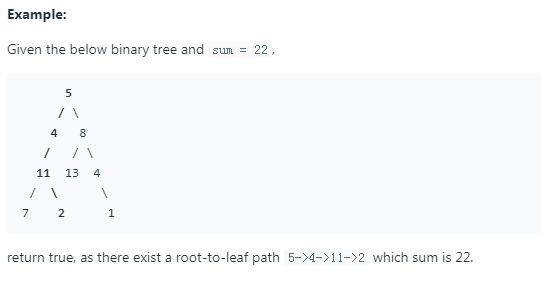
return res;

}

};

第一百一十二题：

给定一棵二叉树和一个整数（正负都有可能），返回是否有一条从根节点到叶子节点的路径之和等于这个整数。



思路是递归查找。如果当前节点不是叶子节点，则将sum减去当前节点的值，然后传到左子树和右子树。当到达一个叶子节点的时候，检查是否sum等于该叶子节点的值，是返回true否返回false。遍历整个二叉树的所有节点，所以时间复杂度为O（n）。

代码：

/\*\*

\* Definition for a binary tree node.

\* struct TreeNode {

\* int val;

\* TreeNode \*left;

\* TreeNode \*right;

\* TreeNode(int x) : val(x), left(NULL), right(NULL) {}

\* };

\*/

class Solution {

public:

bool hasPathSum(TreeNode\* root, int sum) {

if (!root) return false;

if (!root->left && !root->right) return root->val == sum;

if (root->left && hasPathSum(root->left, sum - root->val)) return true;

if (root->right && hasPathSum(root->right, sum - root->val)) return true;

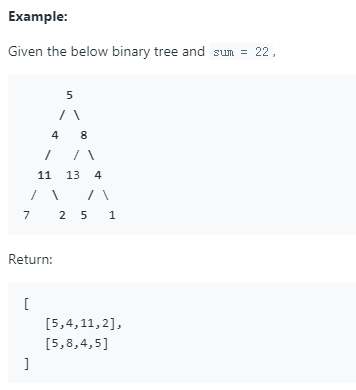
return false;

}

};

第一百一十三题：

给定一棵二叉树和一个正整数sum，找到根节点到叶子节点路径上所有值的和为sum，返回所有这样的路径。



思路就是递归。二叉树很多问题都可以递归解决，这题也是递归，首先将根节点入队，然后sum减去根节点的值，然后检查如果没有左右子节点说明是叶子节点，检查sum==0，是则找到一个答案，否则返回；如果当前不是根节点，如果有左子节点就进入左子节点，由于每次进入都入队一个元素，所以返回的时候记得pop掉。由于每个节点都入队出队一次，这部分的时间复杂度为O（n）；假设一种情况，一共有n个节点，n/2个节点连成一条链，剩余的n/2个节点组成一棵满二叉树，那么路径长度为n/2+log（n/2+1），叶子节点数为（n/2+1）/2，所以push\_back的复杂度为O（n^2）。

代码：

/\*\*

\* Definition for a binary tree node.

\* struct TreeNode {

\* int val;

\* TreeNode \*left;

\* TreeNode \*right;

\* TreeNode(int x) : val(x), left(NULL), right(NULL) {}

\* };

\*/

class Solution {

public:

void dfs(TreeNode \*root,int sum,vector<vector<int>> &res,vector<int> &temp){

temp.push\_back(root->val);

sum-=root->val;

if(root->left==NULL&&root->right==NULL&&sum==0){

res.push\_back(temp);

return;

}

if(root->left){

dfs(root->left,sum,res,temp);

temp.pop\_back();

}

if(root->right){

dfs(root->right,sum,res,temp);

temp.pop\_back();

}

return;

}

vector<vector<int>> pathSum(TreeNode\* root, int sum) {

vector<vector<int>> res;

if(root==NULL)

return res;

vector<int> temp;

dfs(root,sum,res,temp);

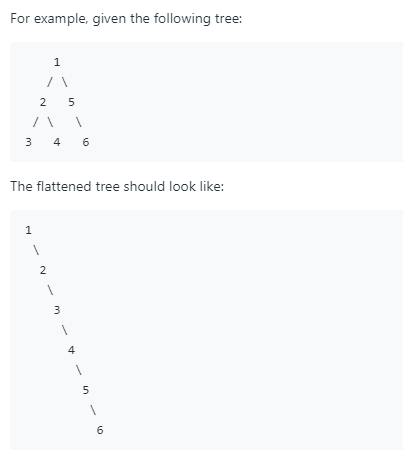
return res;

}

};

第一百一十四题：

给定一个二叉树，按照前序遍历的顺序将其连接成一条右子节点相连的链表。



思路也是递归。因为是按前序遍历排序的，所以当前的根节点肯定是当前的链表头，然后其右子节点的前驱是其左子节点的最右下角的那个节点，所以找到这个前驱结点，然后将其连接在右子节点上，然后根节点的左子树变成右子树。因为根节点已经确定了位置，所以往右走一步，以此类推。首先work会把所有节点遍历一遍；然后front会把左节点及左节点的所有右节点遍历一遍，下次就不会再遍历front刚才遍历过的这些节点；所以所有节点最多被遍历两次，时间复杂度为O（n）。

代码：

/\*\*

\* Definition for a binary tree node.

\* struct TreeNode {

\* int val;

\* TreeNode \*left;

\* TreeNode \*right;

\* TreeNode(int x) : val(x), left(NULL), right(NULL) {}

\* };

\*/

class Solution {

public:

void flatten(TreeNode\* root) {

if(!root)

return;

TreeNode \*work=root;

while(work){

while(work!=NULL&&!work->left)

work=work->right;

if(work){

TreeNode \*front=work->left;

while(front->right!=NULL)

front=front->right;

front->right=work->right;

work->right=work->left;

work->left=NULL;

work=work->right;

}

}

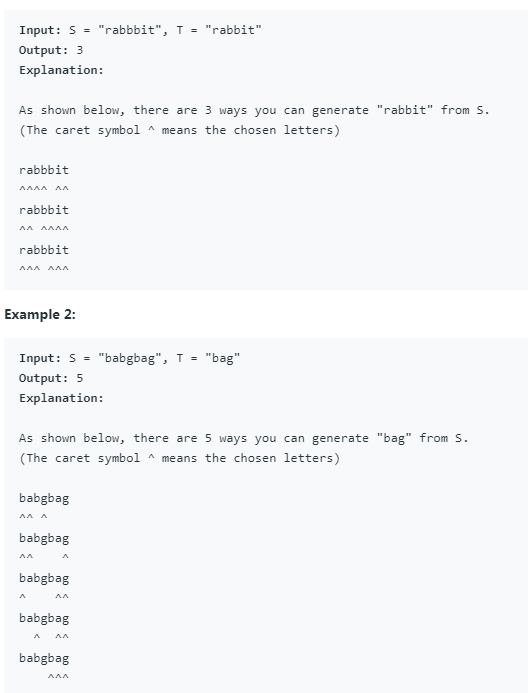
return;

}

};

第一百一十五题：

**这题重要。**给定一个字符串S和字符串T，可以随意删除S中的字符，但是不能改变字符的次序，问有多少种不同的方法使得S转换为T。



思路是动态规划，再次申明，字符串匹配的问题首选动态规划！这题其实一开始不好想，要使得s的子序列与t匹配，可以用dp[i][j]表示s的前i个字符与t的前j个字符匹配的方案数。有一个先决条件：字符串s可以通过去除字符跟t匹配，所以dp[i][0]都是1；有两种情况：①s[i-1]与t[j-1]不匹配，那么s[i-1]这个字符要不要都没什么影响，所以dp[i][j]=dp[i-1][j]；②s[i-1]与t[j-1]匹配，那么s[i-1]这个字符可以和t[j-1]抵消，那么此时的方案数为dp[i][j]=dp[i-1][j-1]，但是也可以不要s[i-1]这个字符，这个时候dp[i][j]=dp[i-1][j]，因此dp[i][j]=dp[i-1][j]+dp[i-1][j-1]；维护了一个大小为mn的二维数组，所以时间复杂度为O（mn）。

代码：

class Solution {

public:

int numDistinct(string s, string t) {

int n = s.size(), m = t.size();

vector<vector<long long>> f(n + 1, vector<long long>(m + 1));

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

f[i][0] = 1;

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

for (int j = 1; j <= m; j ++ )

{

f[i][j] = f[i - 1][j];

if (s[i - 1] == t[j - 1])

f[i][j] += f[i - 1][j - 1];

}

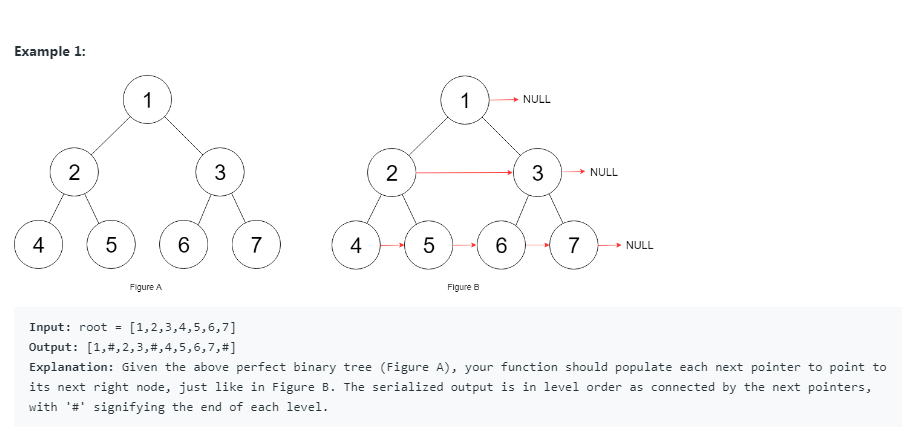
return f[n][m];

}

};

第一百一十六题：

**这题重要。**给定一个满二叉树，在树节点上面定义一个next节点，这个next指向自己的兄弟节点。



思路是迭代。将这棵二叉树分层处理，根节点那一层不用处理；第二层将两个节点连接在一起；第三层首先将4,5连接，然后检查2->next，如果不为空则将5,6连接，并向next移动，然后连接6,7，因为3->next为空，所以这一层结束；下面以此类推。除了最右边的节点以外，所有节点都连接操作了一次，然后每个节点检查了一次，所以总的时间复杂度为O（n）。

代码：

/\*

// Definition for a Node.

class Node {

public:

int val;

Node\* left;

Node\* right;

Node\* next;

Node() : val(0), left(NULL), right(NULL), next(NULL) {}

Node(int \_val) : val(\_val), left(NULL), right(NULL), next(NULL) {}

Node(int \_val, Node\* \_left, Node\* \_right, Node\* \_next)

: val(\_val), left(\_left), right(\_right), next(\_next) {}

};

\*/

class Solution {

public:

Node\* connect(Node\* root) {

if(!root)

return NULL;

Node \*last=root;

while(last->left){

for(Node \*p=last;p;p=p->next){

p->left->next=p->right;

if(p->next)

p->right->next=p->next->left;

}

last=last->left;

}

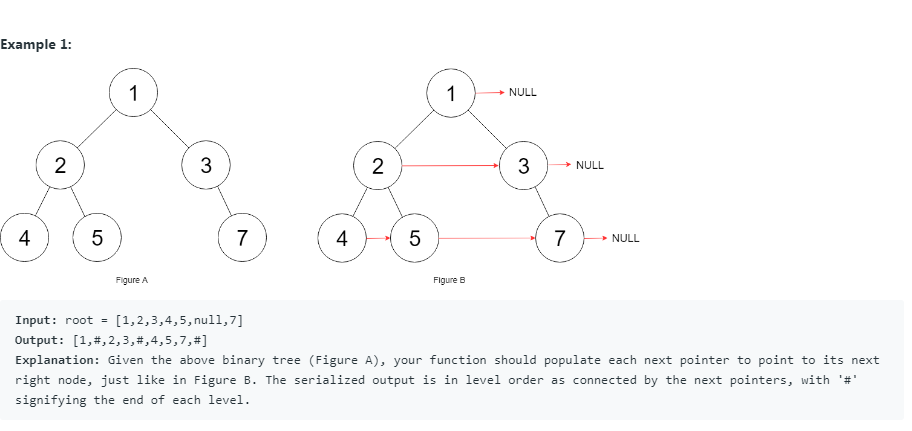
return root;

}

};

第一百一十七题：

给定一棵二叉树，定义一个新的节点next，指向其兄弟节点，连接完成后返回。



思路是迭代，跟上一题差不多。因为这个二叉树不一定是满二叉树，所以会存在缺失某些节点的情况，但是整体做法还是一层一层地连接。首先找到每一层的第一个节点，将其作为这题层链表的头结点，然后不断检查有没有兄弟节点，然后不断连接，直到连接完成。这里有个技巧，因为我们需要定位每一层的第一个节点，为了统一操作，可以声明一个dummy作为每一层的头结点。每个节点都检查连接一次并且遍历一次，所以总的时间复杂度为O（n）。

代码：

/\*\*

\* Definition for binary tree with next pointer.

\* struct TreeLinkNode {

\* int val;

\* TreeLinkNode \*left, \*right, \*next;

\* TreeLinkNode(int x) : val(x), left(NULL), right(NULL), next(NULL) {}

\* };

\*/

class Solution {

public:

void connect(TreeLinkNode \*root) {

while (root)

{

TreeLinkNode \*dummy = new TreeLinkNode(0);

TreeLinkNode \*tail = dummy;

while (root)

{

if (root->left)

{

tail->next = root->left;

tail = tail->next;

}

if (root->right)

{

tail->next = root->right;

tail = tail->next;

}

root = root->next;

}

tail->next = 0;

root = dummy->next;

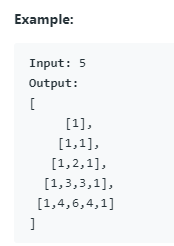
}

}

};

第一百一十八题：

给定一个非负数n，建立n层杨辉三角（帕斯卡三角）并返回。



思路是迭代生成。杨辉三角中val为其左上方和右上方元素之和（没有元素当成0）。可以注意到每一行都是对称的，所以在赋值的时候可以对称赋值；然后每一行的长度是固定的，第n行长度为n，每次声明的长度为n个1，然后每次赋值的次数为（该行长度-1）/2。该行第i个val为上一行的i和（i-1）的和，记得需要对称输入。每个元素都要赋值一遍，所以整体的时间复杂度为n（n+1）/2，为O（n^2）。

代码：

class Solution {

public:

vector<vector<int>> generate(int numRows) {

vector<vector<int>> res;

vector<int> jd;

while(numRows--){

vector<int> temp(jd.size()+1,1);

for(int i=1;i<((temp.size()+1)/2);++i){

temp[i]=jd[i-1]+jd[i];

temp[temp.size()-1-i]=temp[i];

}

res.push\_back(temp);

jd=temp;

}

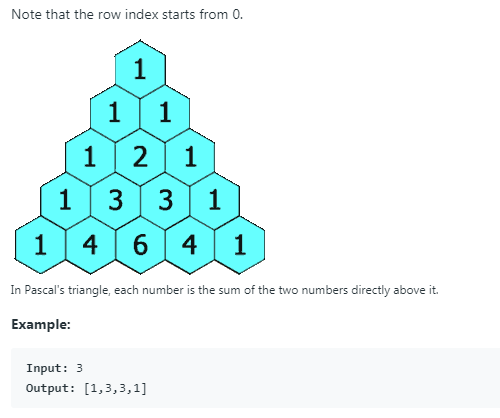
return res;

}

};

第一百一十九题：

假定杨辉三角第一层为第0层，给定一个正整数，返回杨辉三角里这一层的数组。



思路是迭代。因为第n层只与第n-1层有关，所以只要有两个数组就可以无限迭代得到答案。因为第n层要填充n个元素，所以直到答案那一层，一共需要填充n（n+1）/2个元素，因此时间复杂度为O（n^2）。

代码：

class Solution {

public:

vector<int> getRow(int rowIndex) {

vector<int> res(rowIndex+1,1),work(rowIndex+1,1);

if(rowIndex<2)

return res;

for(int i=2;i<=rowIndex;++i){

for(int j=1;j<=(i+1)/2;++j){

work[j]=res[j-1]+res[j];

work[i-j]=work[j];

}

res=work;

}

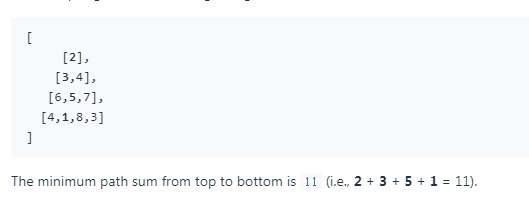
return res;

}

};

第一百二十题：

**这题重要。**给定一个三角形矩阵，每次只能从邻近的元素往下走，返回从山顶到山脚的最小路径和



思路是迭代进行搜索。首先需要重申一点，如果问题不好解决，可以考虑反方向解决问题，比如这题就可以从山脚走到山顶。因为从山顶开始搜索，由于搜索范围越来越大，所以贪心算法不太适合；如果从山脚开始搜索，搜索到最后只剩下一个元素，就是最终的答案。从倒数第二行开始搜索，每一个元素都表示从山脚走到这个格子时，路径长度的最小值。假设总的层数为n，除了三角形两边的元素遍历一次以外，其他元素均遍历两次，所以总共遍历n（n+1）-2n+1，所以总时间复杂度为O（n^2）。

代码：

class Solution {

public:

int minimumTotal(vector<vector<int>>& triangle) {

for (int i = triangle.size()-2 ; i>=0 ; --i){

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

triangle[i][j] += min(triangle[i+1][j], triangle[i+1][j+1]) ;

}

}

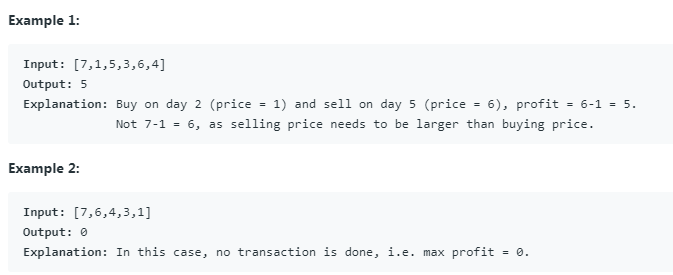
return triangle[0][0];

}

};

第一百二十一题：

给定一个数组，里面表示每天股票的价格，只能买一次卖一次，必须先买然后才能卖，返回利润最大值。



思路就是检查一遍整个数组，顺便更新答案。其实很容易想到要用两个指针来操作，但是怎样操作才是最方便的呢？首先有一个是工作指针i，遍历整个数组，然后一个min标记到目前为止遇到的股票价格最低的那一天，这个min必须小于i。所以就可以每次移动i，先更新profit，然后更新min。遍历一遍整个数组，所以时间复杂度为O（n）。

代码：

class Solution {

public:

int maxProfit(vector<int>& prices) {

if(prices.size()<2) return 0;

int profit = 0, low = prices[0]; //当前最低价格的初始值不是0

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

profit = max(profit, prices[i]-low);

low = min(low, prices[i]);

}

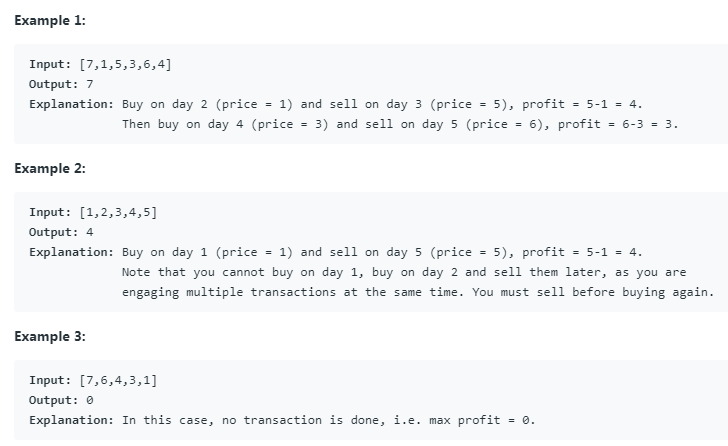
return profit;

}

};

第一百二十二题：

给定股票每天的价格，不限制买卖次数，问最大获益是多少。



思路很简单，就是将所有的升序对的差加起来，就相当于每次涨幅我都买了并且卖了。遍历一次整个数组，时间复杂度为O（n）。

代码：

class Solution {

public:

int maxProfit(vector<int>& prices) {

int res=0,front=0,rear=1;

while(rear<prices.size()){

if(prices[front]<prices[rear])

res+=prices[rear]-prices[front];

front++,rear++;

}

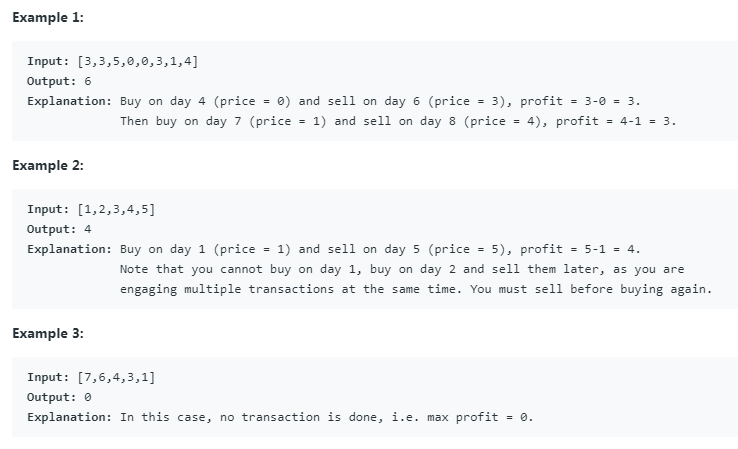
return res;

}

};

第一百二十二题：

**这题重要。**给定一个数组表示一个股票每天的价格变化，买卖两次，且必须两次买卖时间范围没有重叠，返回最大收益。



思路很独特，就是要找到一个分界线，这个分界线前后各有一次买卖，这样受益最大。首先维护一个数组表示买卖一次的时候，在第i天获得的最大受益，从前往后计算，具体方法是找到股票价格最低的那一天，然后在其后面找到一天，这两天的价格差最大；然后从后往前找到买卖一次的最大受益，具体方法是找到股票价格最高的那一天，然后在其前面找到一天，这两天的价格差最大。**重点**在于：我们从后往前求的时候要加上f[i-1]（由于买卖不能有冲突，所以第i天是第二次买股票的日子，所以不能是第一次卖的日子），表示前i-1天内买卖一次获得的最大受益，所以就在第i天把这两次买卖隔开了，并且求得了这两次买卖的最大受益。从前往后再从后往前遍历这个数组一次，所以总的时间复杂度为O（n）。

代码：

class Solution {

public:

int maxProfit(vector<int>& prices) {

int n = prices.size();

if (!n) return 0;

vector<int> f(n, 0);

int minv = prices[0];

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

{

f[i] = max(f[i - 1], prices[i] - minv);

minv = min(minv, prices[i]);

}

int res = f[n - 1];

int maxv = prices[n-1];

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

{

res = max(res, maxv - prices[i] + f[i - 1]);

maxv = max(maxv, prices[i]);

}

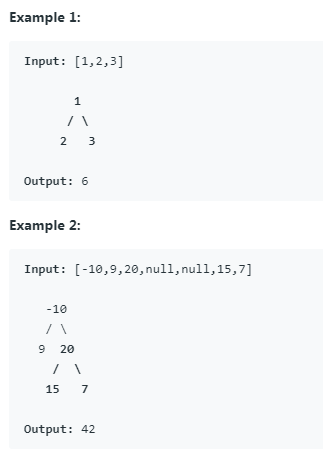
return res;

}

};

第一百二十四题：

给定一个非空二叉树，随便找到一条相连的路径，这条路径加起来的值最大，至少需要一个节点，返回这个最大值。



思路是递归遍历。这题告诉了我一个道理，不能因为是叶子节点就区别对待，而应该一视同仁，否则可能容易出错。也就是说，以后**判断左右节点为空则返回应该改为当前节点为空才返回**，同时进入左右节点不同判断是否为空，这样有利于所有节点都进行了同样的操作。这题的思路是维护一个变量ans记录当前获得的最大值。从根节点开始，首先获取左子树的最大值，跟0作比较，因为如果小于0可以直接舍弃，右子树同理；然后更新ans=max（ans，l+r+root->val），注意了，如果这里的root是叶子节点，也可以更新了，所以不会漏掉只有一个节点的情况。由于父节点只能连接当前root的左子树或者右子树，所以要返回max（l+root->val,r+root->val）。所有节点都遍历了一遍，所以时间复杂度为O（n）。

代码：

/\*\*

\* Definition for a binary tree node.

\* struct TreeNode {

\* int val;

\* TreeNode \*left;

\* TreeNode \*right;

\* TreeNode(int x) : val(x), left(NULL), right(NULL) {}

\* };

\*/

class Solution {

public:

int ans;

int maxPathSum(TreeNode\* root) {

ans = INT\_MIN;

dfs(root);

return ans;

}

int dfs(TreeNode\* root)

{

if (!root) return 0;

int left = max(0, dfs(root->left));

int right = max(0, dfs(root->right));

ans = max(ans, left + root->val + right);

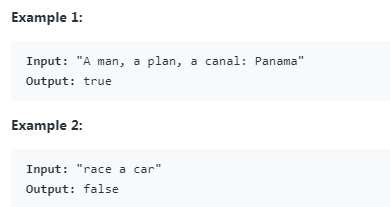
return root->val + max(left, right);

}

};

第一百二十五题：

给定一个字符串，忽略除了数字与英语字母外所有的字符，并忽略大小写，判断是否为回文字符串。



思路很简单，就是利用双指针迭代比较。一个指针f从前面开始，一个指针r从后面开始，一直比较知道f>=r的时候就返回true，如果中间有不符合要求的就返回false。

代码：

class Solution {

public:

bool isPalindrome(string s) {

int f=0,r=s.size()-1;

while(f<r){

while(f<r&&(!isdigit(s[f])&&!isalpha(s[f])))

f++;

if(isalpha(s[f]))

s[f]=tolower(s[f]);

while(f<r&&(!isdigit(s[r])&&!isalpha(s[r])))

r--;

if(isalpha(s[r]))

s[r]=tolower(s[r]);

if(s[f]!=s[r])

return false;

f++,r--;

}

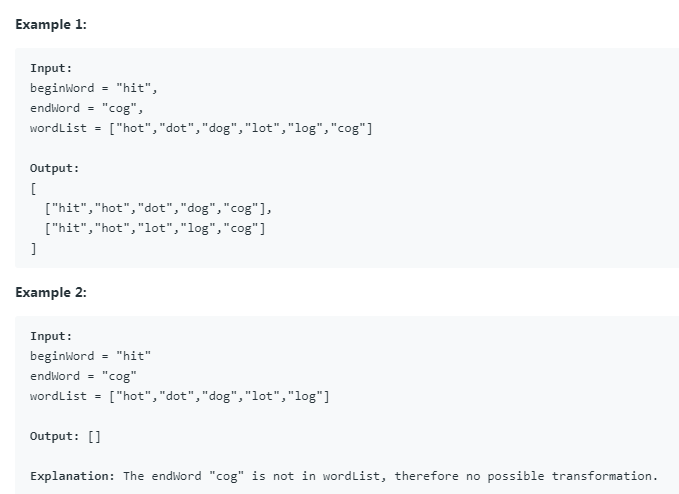
return true;

}

};

第一百二十六题：

**这题重要。**给定一个字符串beginword和一个字符串endword还有一个字符串数组wordlist，从beginword开始，每次变换只能改变一个字母并且变换后的字符串必须在wordlist内，返回所有的从beginword到endword的变换路径。



思路想出来不算太难，但是很难实现。因为从起点开始到终点跟终点开始到起点的变换路径是相同的，只不过顺序相反，但是终点变换到起点可以保证不会有冗余的变换，所以咱们从终点开始变换。首先建立一张图，每个单词都是其中的一个点，如果两个单词之间变换一次就可以互达，那么他们相连。用dist[i]表示i和起点的距离，只有距离为1的有一条边连接，其他情况忽略掉，BFS递归建立这个图。然后我们从终点开始，终点为dist[E]，如果dist[i]+1==dist[E]，说明i可以到end，然后递归搜索到达i的节点。建立图有两重循环，时间复杂度为O（n^2）；求最短路径使用BFS，所用时间为O（n）；求解最终答案用的是DFS，是指数级别的，每次记录答案需要O（n），所以这部分用时O（2^n\*n）；综上，时间复杂度为O（2^n\*n）。

代码解读：source记录包含beginword+wordlist的所有单词；mapw记录每个单词在source中的下标；循环①：g[i]表示与sorce[i]距离为1的点的集合，dist[i]暂时设置为无穷大；循环②：BFS来计算dist[i]，表示起点到这个点的距离；path表示从终点开始的路径，end是终点那个单词在source中的下标，这个path跟答案要求的路径是相反的；dfs中，传入的参数是当前节点在source中的下标，由于起点下标为0，所以dfs（0）就是记录答案的意思，循环只能从与自己的距离为1的节点里选择。

代码：

class Solution {

public:

bool check(string a, string b)

{

int res = 0;

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

res += a[i] != b[i];

return res == 1;

}

vector<string> source;

vector<vector<string>> ans;

vector<vector<int>> g;

vector<int> dist, path;

int n, end;

vector<vector<string>> findLadders(string beginWord, string endWord, vector<string>& wordList) {

unordered\_map<string, int> mapw;

n = 0;

mapw[beginWord] = n ++ ;

source.push\_back(beginWord);

for (auto &word : wordList)

if (mapw.count(word) == 0)

{

mapw[word] = n ++ ;

source.push\_back(word);

}

if (mapw.count(endWord) == 0) return ans;

end = mapw[endWord];

for (int i = 0; i < n; i ++ )//循环①

{

vector<int> temp;

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

if (check(source[i], source[j]))

temp.push\_back(j);

g.push\_back(temp);

dist.push\_back(100000000);

}

dist[0] = 0;

queue<int> que;

que.push(0);

while (!que.empty())//循环②

{

int t = que.front();

que.pop();

for (int i = 0; i < g[t].size(); i ++ )

{

int j = g[t][i];

if (dist[j] > dist[t] + 1)

{

dist[j] = dist[t] + 1;

que.push(j);

}

}

}

path.push\_back(end);

dfs(end);

return ans;

}

void dfs(int start)

{

if (!start)

{

vector<string> temp;

for (int i = path.size() - 1; i >= 0; i -- )

temp.push\_back(source[path[i]]);

ans.push\_back(temp);

return ;

}

for (int i = 0; i < g[start].size(); i ++ )

{

int j = g[start][i];

if (dist[j] + 1 == dist[start])

{

path.push\_back(j);

dfs(j);

path.pop\_back();

}

}

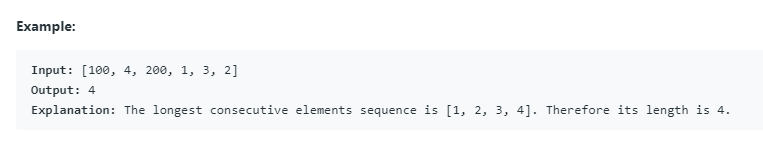
}

};

第一百二十七题：

第一百二十八题：

给定一个无序数组，返回最长连续序列长度，要求复杂度为O（n）。



思路是哈希表。首先最简单的当然是先排序然后再计算，但是这样的话复杂度不符合要求，因此另辟蹊径；可以知道的是，肯定得保存每个数字的状态，并且我们需要随时查看，所以使用哈希表来保存状态；我们使用unordered\_map<int,int> judge[i]表示数字i目前的连续长度，但是有一个问题，如果新增了一个数字4，是否要把judge[1],judge[2]的长度都更新呢？答案是不用的，我们只需要每次都去同一个地方取数值就行；使用judge[i]表示数字i目前的连续长度；例如judge[4]=3表示nums里面有2,3,4或者4,5,6；只有遇到新出现的数字才去进行计算；首先将judge[i]=1，然后获取其左边和右边的连续序列长度leftlen=judge[i-1]和rightlen=judge[i+1]，然后更新judge[i-leftlen]=judge[i+rightlen]=leftlen+rightlen+1；同时更新ans=max（ans，leftlen+rightlen+1）；**这里为什么只需要更新i-leftlen和i+rightlen呢？**因为只有边界值连续时，ans才有可能变长；因此我们每次仅需要更新边界点就行，因为**合并必然发生在边界点处。**遍历整个nums一遍，因此时间复杂度为O（n）。

代码：

class Solution {

public:

int longestConsecutive(vector<int>& nums) {

int ans=0;

if(nums.size()==0)

return ans;

unordered\_map<int,int> judge;

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

if(judge[nums[i]]==0){

judge[nums[i]]++;

int leftlen=judge[nums[i]-1],rightlen=judge[nums[i]+1];

judge[nums[i]-leftlen]=leftlen+rightlen+1;

judge[nums[i]+rightlen]=judge[nums[i]-leftlen];

ans=max(ans,leftlen+rightlen+1);

}

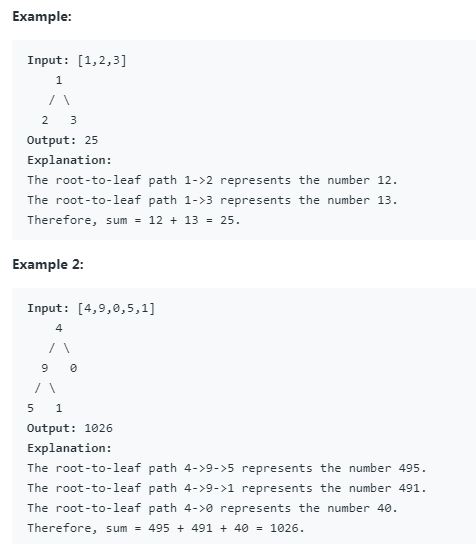
return ans;

}

};

第一百二十九题：

给定一棵二叉树，返回其所有根节点到叶子节点组成的元素的和。



思路很简单，就是遍历整棵二叉树顺便计算结果。关键是使用什么方法遍历，因为这里要保留经过的节点的值作为中间值，比如4-9-5和4-9-1，在到达5和1的节点的时候，保留着490这个中间值，所以这里选择后序遍历。每次检查是否是叶子节点，如果是则更新结果；如果不是则继续递归查找，记得要更新中间值。这里的一个问题是，必须保证每个节点的操作是一致的，否则会出现不可预知的错误。遍历整棵二叉树一遍，所以时间复杂度为O（n）。

代码：

/\*\*

\* Definition for a binary tree node.

\* struct TreeNode {

\* int val;

\* TreeNode \*left;

\* TreeNode \*right;

\* TreeNode(int x) : val(x), left(NULL), right(NULL) {}

\* };

\*/

class Solution {

public:

int res=0;

int sumNumbers(TreeNode\* root) {

if(root)

findout(root,0);

return res;

}

void findout(TreeNode\* root,int ans){

if(root->left==NULL&&root->right==NULL){

res+=ans\*10+root->val;

return;

}

if(root->left)

findout(root->left,ans\*10+root->val);

if(root->right)

findout(root->right,ans\*10+root->val);

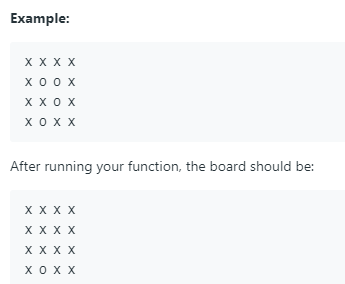
return;

}

};

第一百三十题：

给定一个2D模板，上面有X和O，如果O被X包围，那就将O变成X。tips：如果O与O之间是水平或者垂直相邻的，那说明这两个O是连在一起的；此外，如果这个连在一起的O有一个在边界上，那么连在一起的O都不会变成X。



思路很简单。首先检查最外围一圈的元素，如果是O，则递归将这一连串的O都变成Y。第二次检查整个2D模板，遇到X跳过；遇到Y则变回O；遇到O则变成X；最坏情况下递归检查整个模板，最后调整模板也是遍历整个模板，所以总的时间复杂度为O（n^2）。

代码：

class Solution {

public:

void labelit(vector<vector<char>> &b,int x,int y,int row,int col){

b[x][y]='Y';

if(x>0&&b[x-1][y]=='O')

labelit(b,x-1,y,row,col);

if(x<row-1&&b[x+1][y]=='O')

labelit(b,x+1,y,row,col);

if(y>0&&b[x][y-1]=='O')

labelit(b,x,y-1,row,col);

if(y<col-1&&b[x][y+1]=='O')

labelit(b,x,y+1,row,col);

return;

}

void solve(vector<vector<char>>& board) {

int row=board.size(),col=0;

if(row)

col=board[0].size();

if(row\*col==0)

return;

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

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

if((i==0||i==row-1)&&board[i][j]=='O')

labelit(board,i,j,row,col);

else if(i==0||i==row-1)

continue;

else{

if(board[i][j]=='O')

labelit(board,i,j,row,col);

j+=col-2;

}

}

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

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

if(board[i][j]=='O')

board[i][j]='X';

else if(board[i][j]=='Y')

board[i][j]='O';

}

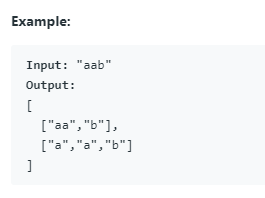
return;

}

};

第一百三十一题：

给定一个字符串s，将其分割为若干个回文串，返回所有分割结果。



思路是深度优先遍历。要将一个字符串分割成若干个回文串，就从左到右遍历整个字符串，如果当前的子串now不是一个回文串，就向里面添加一个字符，继续遍历；如果当前的子串now是一个回文串，就用path记录下来，然后清空now继续遍历；即使当前子串now是回文串，依然可以继续添加字符，继续遍历；当num==s.size（）说明整个字符串s枚举完毕，检查当前的now，如果是回文串，说明这一次分割成功，将path加入ans，否则return。把整个字符串s想象成一个一个字符，分割就是在各个字符中间插入板子，每一个间隔可以选择插入或者不插入，因此有2^(n-1)中情况，然后每一次入队都要O（n），所以总的时间复杂度为O（n\*2^n）。

代码：

class Solution {

public:

bool check(string &s){

if(s.size()==0)

return false;

for(int i=0,j=s.size()-1;i<j;i++,j--)

if(s[i]!=s[j])

return false;

return true;

}

void dfs(string now,int num,string &s,vector<vector<string>> &ans,vector<string> &path){

if(num==s.size()){

if(check(now)){

path.push\_back(now);

ans.push\_back(path);

path.pop\_back();

}

return;

}

if(check(now)){

path.push\_back(now);

dfs("",num,s,ans,path);

path.pop\_back();

}

dfs(now+s[num],num+1,s,ans,path);

return;

}

vector<vector<string>> partition(string s) {

vector<vector<string>> ans;

vector<string> path;

dfs("",0,s,ans,path);

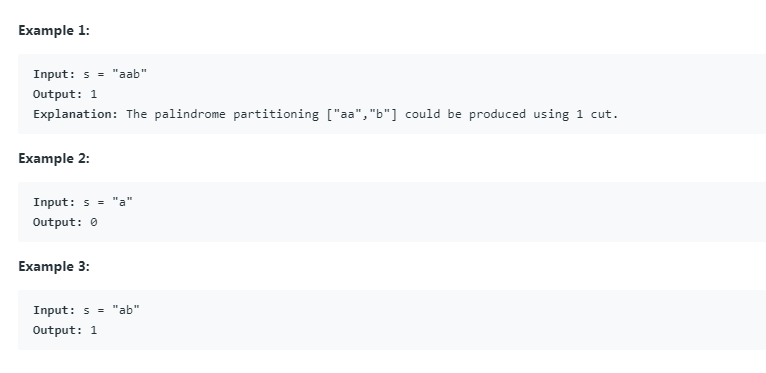
return ans;

}

};

第一百三十二题：

给定一个字符串s，分割为回文串数组，最少割几刀？



思路是动态规划。首先要使用judge[j][i]表示字符串s[j...i]是不是回文字符串，这里的遍历方式有点神奇，可以留意一下；然后我们使用一个dp[i]表示s[0...i-1]最少需要割多少刀；首先dp[0]=0，然后检查s[j...i-1]是不是回文字符串，如果是那说明此时应该把s[j...i-1]作为一个回文串，s[0...j-1]作为一整个回文串，所以dp[i]=min(dp[i],dp[j]+1)，其中dp[j]就是s[0...j-1]需要分割的最少次数；两个动态规划都是进行了n^2的检查，所以总的时间复杂度为O(n^2)。

代码：

class Solution {

public:

int minCut(string s) {

int len=s.size();

vector<int> dp(len+1,INT\_MAX);

vector<vector<bool>> judge(len,vector<bool>(len,false));

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

for(int j=i;j>=0;--j){

if(i-j<=1)

judge[j][i]=s[j]==s[i];

else

judge[j][i]=s[j]==s[i]&&judge[j+1][i-1];

}

dp[0]=0;

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

dp[i]=i;

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

if(judge[j][i-1])

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

}

}

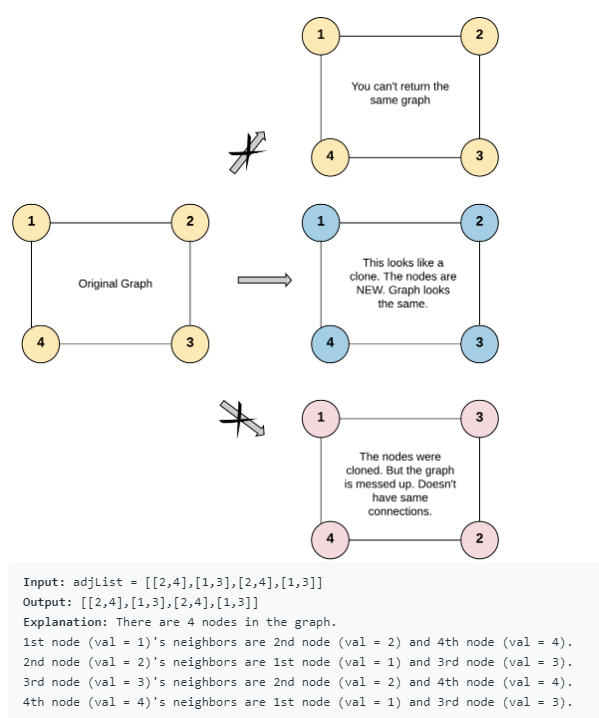
return dp[len]<0?0:dp[len]-1;

}

};

第一百三十三题：

给定一个无向图，克隆一个一模一样的图。其中每个节点带有一个val和一个邻接节点vector。



思路就是检查并克隆。首先检查如果node本身就是空的，说明图是空的，直接返回NULL；由于题目中的val唯一，所以用一个哈希表来的key记录这个val是否已经创建过节点，如果没有就new一个节点，然后放在哈希表的value中。然后遍历node的邻接点，如果哈希表中没有这个节点，递归建立；否则直接从哈希表中取出这个指针加入。每一个节点都要建立一次，然后遍历的时候最坏情况下需要n-1次，所以总的时间复杂度为O（n^2）。

代码：

/\*

// Definition for a Node.

class Node {

public:

int val;

vector<Node\*> neighbors;

Node() {

val = 0;

neighbors = vector<Node\*>();

}

Node(int \_val) {

val = \_val;

neighbors = vector<Node\*>();

}

Node(int \_val, vector<Node\*> \_neighbors) {

val = \_val;

neighbors = \_neighbors;

}

};

\*/

class Solution {

public:

unordered\_map<int,Node\*> wh;

Node\* cloneGraph(Node\* node) {

if(node==NULL)

return node;

Node \*p=new Node(node->val);

wh[p->val]=p;

if(!node->neighbors.empty())

for(auto x:node->neighbors){

if(wh[x->val]==NULL)

cloneGraph(x);

p->neighbors.push\_back(wh[x->val]);

}

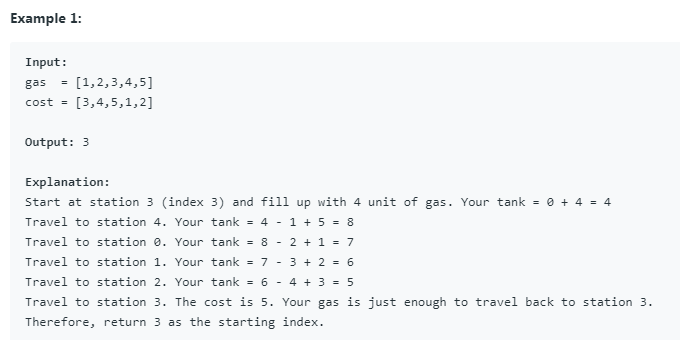
return p;

}

};

第一百三十四题：

给定一个数组gas表示每个油站的储油量，一个数组cost表示到达下一个油站需要的油量，所有油站环状排列，按照顺时针方向前进，是否有一个油站开始可以绕一圈回到此油站？有则返回该油站下标，否则返回-1.



1. 首先用每个位置的 gas 减去 cost 求出当前位置的真正花费 sum，然后将 sum 数组扩展为 2n，使得 sum[i] == sum[i+n]。

2. 定义两个指针 start 和 end，分别表示当前假设的起点，和在这个起点下能走到的终点，tot 为当前油量。

3. 如果发现 tot < 0，即不能走到 end 时，需要不断往后移动 start，使得 tot 能满足要求。注意到，向后移动 start 并不会使得 [start, end] 之间出现油量为负的情况。

4. 如果 end - start + 1 == n，即找到了一个环形路线。

5. 最多遍历整个数组，时间复杂度为O（n）；空间复杂度为O（n）；

代码：

class Solution {

public:

int canCompleteCircuit(vector<int>& gas, vector<int>& cost) {

int n = gas.size();

vector<int> sum = vector<int>(n \* 2, 0);

for (int i = 0; i < n \* 2; i++)

sum[i] = gas[i % n] - cost[i % n];

int start = 0, end = 0, tot = 0;

while (start < n && end <= 2 \* n) {

tot += sum[end];

while (tot < 0) {

tot -= sum[start];

start++;

}

if (end - start + 1 == n)

return start;

end++;

}

return -1;

}

};

作者：wzc1995

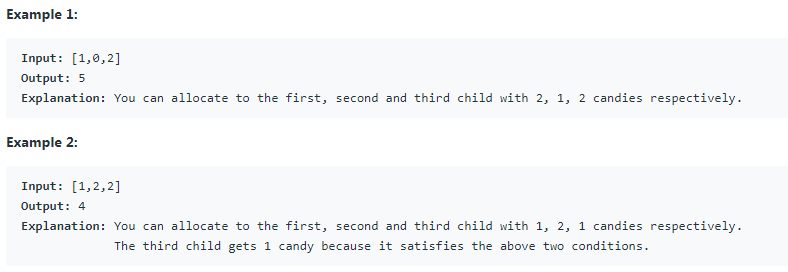
链接：https://www.acwing.com/solution/content/229/

来源：AcWing

著作权归作者所有。商业转载请联系作者获得授权，非商业转载请注明出处。

第一百三十五题：

这题重要。给定一个数组，表示每个小屁孩的能力值，能力值比相邻小屁孩大的，分到的糖果必须必相邻小屁孩多，每个小屁孩至少会分到一个糖果，问至少需要多少个糖果。



思路是扫描两次。首先给每个小屁孩保底一个糖果，然后我们从左往右扫描，如果第i个小屁孩比i-1的小屁孩能力值大，那这个小屁孩至少要拿到i-1小屁孩糖果+1；同理，我们从右往左扫描的时候，如果第i个小屁孩能力值比i+1能力值大，那至少要拿到i+1小屁孩糖果+1；最后统计一共需要多少个糖果即可。扫描三次数组，所以总的时间复杂度为O（n）。

代码：

class Solution {

public:

int candy(vector<int>& ratings) {

int len=ratings.size(),ans=0;

vector<int> wh(len,1);

for(int i=1;i<len;++i)

if(ratings[i]>ratings[i-1]&&wh[i]<=wh[i-1])

wh[i]=wh[i-1]+1;

for(int i=len-2;i>=0;--i)

if(ratings[i]>ratings[i+1]&&wh[i]<=wh[i+1])

wh[i]=wh[i+1]+1;

for(auto x:wh)

ans+=x;

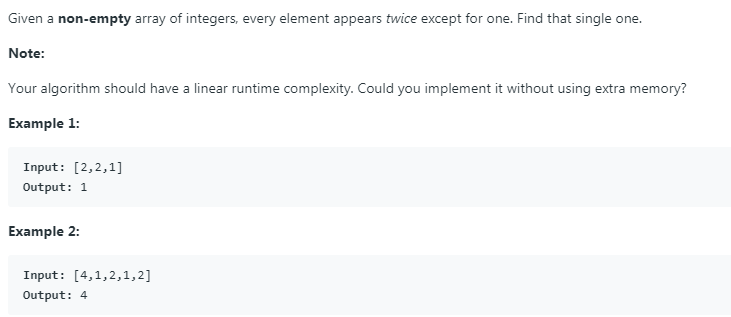
return ans;

}

};

第一百三十六题：

**这题重要。**给定一个数组，里面的数字都会出现2次，但是仅有一个数字出现一次，找到这个出现一次的数字。要求使用线性复杂度，常数额外空间。



思路是进行位运算。其实对于需要常数额外空间的问题，要不就是在原数组内进行操作，要不就是真的使用常数空间，而位运算是一种比较容易遗忘，但是又比较重要的一种。对于同一个数字，比如2，其二进制表示为10；如果2与2进行异或，就会归零。利用这个性质，将nums[0]设为初始值，然后不断与后面的元素进行异或操作，到最后剩下的就是出现次数为1的数字。遍历数组一遍，时间复杂度为O（n），使用常数额外空间。（**注意：**这一题可以用于利用位运算交换两个数：

a = a^b;//用a来存储a^b的结果，之后如果^a就会得到b；如果^b就会得到a；

b = a^b;//首先^b，就会得到a，这里相当于b = a^b^b

a = a^b;//然后^a，就会得到b，这里因为b已经变成了a，且a=a^b，相当于a = a^b^a

**但是**并不会加快交换速度！！！！传统的交换只需要交换三次即可，如果使用位运算，要异或三次，还需要读取ab的值，交换就是直接交换不用读取。因此一定不要将其用于实际项目中。）

代码：

class Solution {

public:

int singleNumber(vector<int>& nums) {

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

nums[0] ^= nums[i];

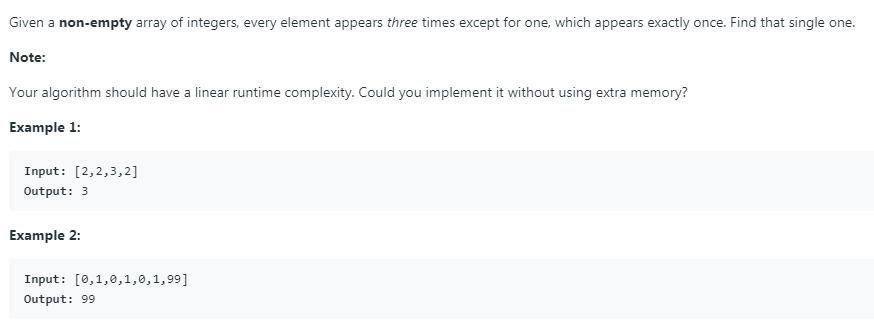
return nums[0];

}

};

第一百三十七题：

给定一个数组，里面的元素都出现3次，但是有一个元素只出现1次，找到这个元素。要求使用线性时间，常数额外空间



思路也是位运算。如果从位来考虑，每一位上如果1出现的次数为3n次，那说明出现1次的那个元素这一位肯定是0；如果这一位1出现的次数为3n+1次，那说明那个元素的这一位是1，由此可以得到答案。对0不计数，因为全是0的数只有一个，其他的数字必然含有1，所以不会混淆。因为这里的元素都是32位的，所以就统计32个位就行了。这里注意右移i位就可以获取第i位的数字，与上1就可以知道第i位是不是1；左移是将是为1的那一位放回它应该在的位置。遍历整个数组32次，时间复杂度为O（n），使用了两个变量，常数的额外空间。

代码：

class Solution {

public:

int singleNumber(vector<int>& nums) {

int ans = 0;

for(int i=0;i<32;++i){

int cnt = 0;

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

cnt += (nums[j] >> i) & 1;

ans += (cnt % 3) << i;

}

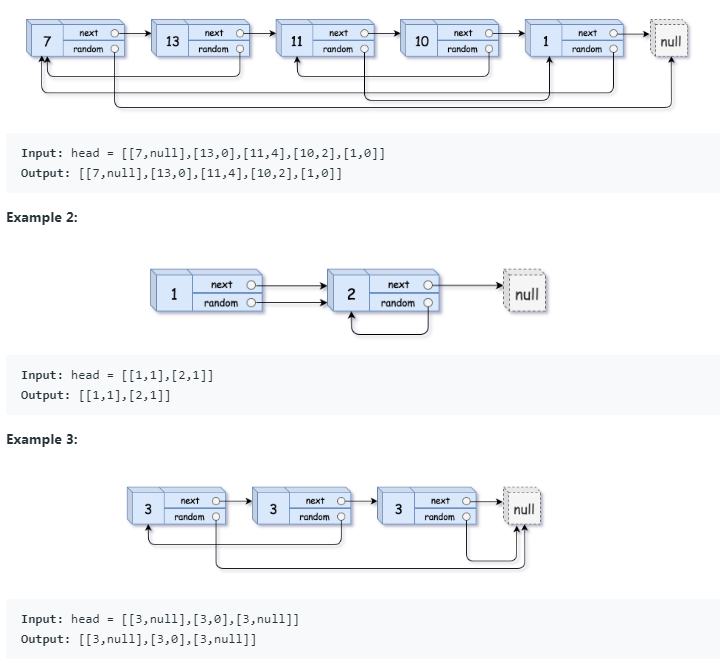
return ans;

}

};

第一百三十八题：

**这题重要。**给定一个链表，这个链表有三个成员变量：val一个值，next一个指针，random一个指针。val可以重复，random指向任意一个结点，可以是null，对其进行深拷贝。



思路特别巧妙。首先遍历一遍整个链表，每一个结点都复制一遍，然后立刻插在当前节点和下一个节点中间，然后就能得到一个（旧节点1->新节点1->旧节点2->新节点2->...）这样的链表，这样的好处是，我们到达旧节点的同时，就会知道对应的新节点就在其后边。然后遍历旧节点，如果其random不为空，则将对应的新节点连到其random相连的那个旧节点对应的新节点那里，旧节点对应的新节点就是旧节点的next。最后再将这个链表拆成两个就行了。遍历链表三遍，所以时间复杂度为O（n）。

代码：

/\*

// Definition for a Node.

class Node {

public:

int val;

Node\* next;

Node\* random;

Node(int \_val) {

val = \_val;

next = NULL;

random = NULL;

}

};

\*/

class Solution {

public:

Node\* copyRandomList(Node\* head) {

if(head==NULL)

return head;

Node \*cur = head;

while(cur!=NULL){

Node \*temp = new Node(cur->val);

temp->next=cur->next;

cur->next=temp;

cur=cur->next->next;

}

cur=head;

while(cur!=NULL){

if(cur->random!=NULL){

Node \*temp = cur->random;

cur->next->random=temp->next;

}

cur=cur->next->next;

}

cur=head;

Node \*newhead = cur->next;

while(cur!=NULL){

Node \*temp=cur->next;

cur->next=temp->next;

if(temp->next!=NULL)

temp->next=temp->next->next;

cur=cur->next;

}

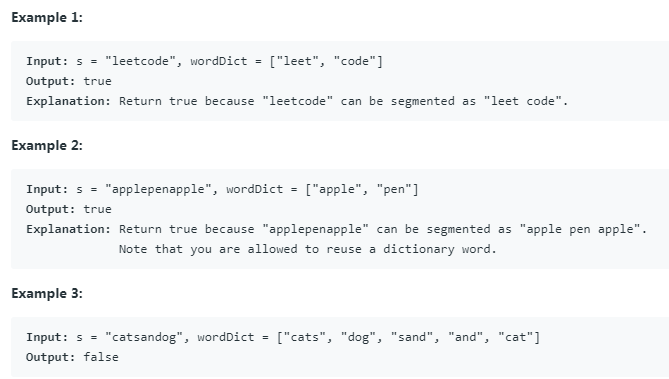
return newhead;

}

};

第一百三十九题：

**这题重要。**给定一个字符串s，和一个字典wordDict，判断s是否可以由字典里面的单词组成，字典里面无重复单词。



思路很巧妙，是一道动态规划题。首先这道题其实用搜索可以解，但是没有有效剪枝肯定会超时。所以有了动态规划，首先规定jd[i]表示s[0：i)这一个子串可以由字典里面的单词组成，所以最后只要返回jd[s.size()]即可。那么如何判断jd[i]是否为true呢？我们不断截取不同长度的s的子串，然后先判断这一个子串是否在字典中；如果存在，就判断这个子串前面的部分是否可以被字典组成，如果可以，那么加上这段子串也可以。例如，leetcode的子串code==s[4:8)存在于字典，那么判断jd[4]即（s[0:4)==leet）是否可以由字典组成，如果可以，则jd[8]=true。遍历一遍字典转化为unordered\_map，然后两层循环为n^2，因此时间复杂度为O（n^2）。

代码：

class Solution {

public:

bool wordBreak(string s, vector<string>& wordDict) {

unordered\_map<string,int> wh;

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

wh[wordDict[i]]++;

vector<bool> jd(s.size(),false);

jd[0]=true;

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

for(int j=i-1;j>=0;--j)

if(wh.count(s.substr(j,i-j))==1&&jd[j])

jd[i]=true;

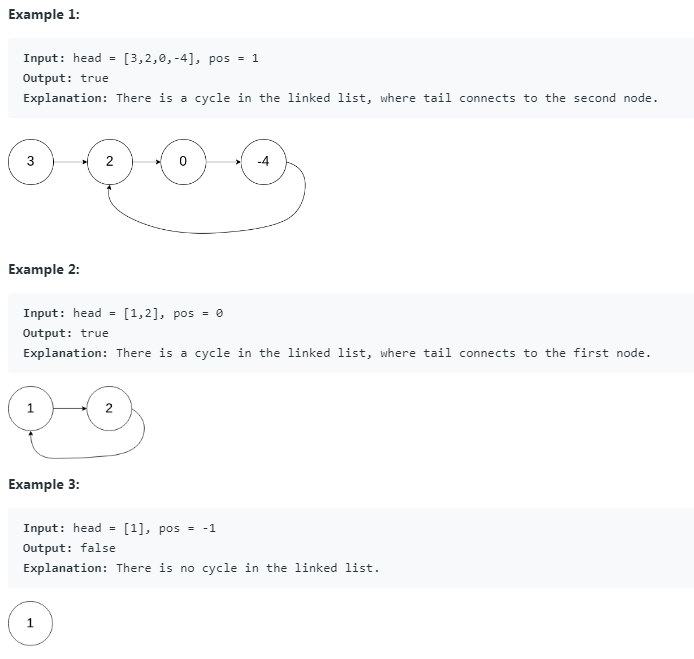
return jd[s.size()];

}

};

第一百四十题：

给定一个链表，判断其中是否存在环。通过给定一个pos表示最后一个节点连接第几个节点，-1表示没有环，这里的pos不传入函数，只作我们人为判断的标志。



思路是快慢指针。快指针一次两步，慢指针一次一步，那么每次两个指针之间的距离会加一。由于环的入口我们不能确定，所以慢指针必须一直往前走，快指针会比慢指针更快进入环，然后虽然与慢指针的距离每次+1，但是因为存在环，所以肯定会有一个时刻会追上慢指针，此时快慢指针指向同一个节点，表示有环；如果快指针指向NULL，说明没环。最坏情况下整个链表就是一个环，慢指针回到第一个节点的时候刚好被追上，所以时间复杂度为O（n）。

代码：

/\*\*

\* Definition for singly-linked list.

\* struct ListNode {

\* int val;

\* ListNode \*next;

\* ListNode(int x) : val(x), next(NULL) {}

\* };

\*/

class Solution {

public:

bool hasCycle(ListNode \*head) {

ListNode \*f=head,\*r=head;

while(r!=NULL&&r->next!=NULL){

f=f->next;

r=r->next->next;

if(f==r)

return true;

}

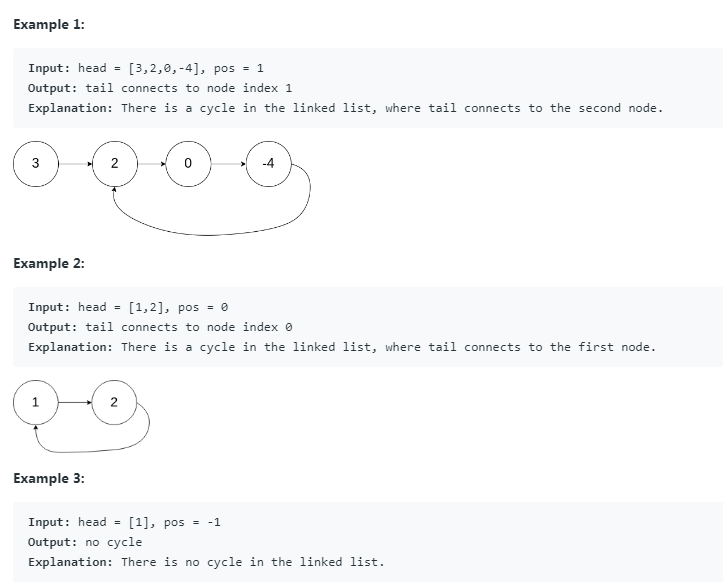
return false;

}

};

第一百四十二题：

给定一个链表，判断是否有环，如果有，则返回这个环开始的节点；如果没有环，就返回NULL



思路很独特。首先老方法，一个快指针一个慢指针，先判断是否有环，没有直接返回NULL；如果有环，则此时慢指针已经走了k步了，快指针2k步。由于快指针是追上了慢指针，也就是说，刚好比慢指针多走了n次环，也就是说，多出来的k步肯定是n倍的环包含的节点数，即k=n\*r，r是环包含的节点数；假设从head到环开始的节点为s，环开始的节点到相遇的节点为m，显然s=k-m==>s=nr-m==>s=（n-1）r+（r-m）；也就是说，从head到环开始的节点等于走n-1圈环然后再走r-m个节点；那我们可以用一个指针p从head出发，一个指针q从相遇点出发，p走s步就会遇到q，因此返回相遇的节点即可。最坏情况下环开始的节点在倒数第一个节点，此时一共走了2n步，因此时间复杂度为O（n）。

代码：

/\*\*

\* Definition for singly-linked list.

\* struct ListNode {

\* int val;

\* ListNode \*next;

\* ListNode(int x) : val(x), next(NULL) {}

\* };

\*/

class Solution {

public:

ListNode \*detectCycle(ListNode \*head) {

ListNode \*f=head,\*r=head;

while(r!=NULL&&r->next!=NULL){

f=f->next;

r=r->next->next;

if(f==r)

break;

}

if(f!=r||r==NULL||r->next==NULL)

return NULL;

f=head;

while(f!=r){

f=f->next;

r=r->next;

}

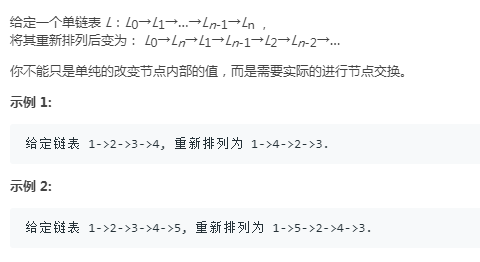
return f;

}

};

第一百四十三题：

给定一个链表，按照以下规则重新连接。就是将后面一半的链表先翻转，然后逐个连接。



思路是先分开两个链表，然后再逐个连接。**注意！！！！！指向NULL的指针跟野指针不同，野指针无法判定\*p==NULL！！！！**首先，找到中间的节点，这里我让快指针先走了一步，因为要从中间断开，断开之后就彻底分成两个链表了；然后就是后面的链表的翻转，得到了newhead和head；最后用work连接所有的节点，然后返回就行。首先二分链表遍历一次，翻转遍历了n/2，最后的连接时O（n），所以总的时间复杂度为O（n）。

代码：

/\*\*

\* Definition for singly-linked list.

\* struct ListNode {

\* int val;

\* ListNode \*next;

\* ListNode() : val(0), next(nullptr) {}

\* ListNode(int x) : val(x), next(nullptr) {}

\* ListNode(int x, ListNode \*next) : val(x), next(next) {}

\* };

\*/

class Solution {

public:

void reorderList(ListNode\* head) {

if(head==NULL||head->next==NULL)

return;

ListNode \*quick=head->next,\*low=head;

while(quick!=NULL&&quick->next!=NULL){

low=low->next;

quick=quick->next->next;

}

ListNode \*signal=low->next,\*newhead=NULL,\*temp;

low->next=NULL;

low=signal;

while(low!=NULL){//这里面是链表反转

temp=low->next;

low->next=newhead;

newhead=low;

low=temp;

}

ListNode \*work=head;

while(newhead!=NULL){

signal=work->next;

work->next=newhead;

newhead=newhead->next;

work->next->next=signal;

work=signal;

}

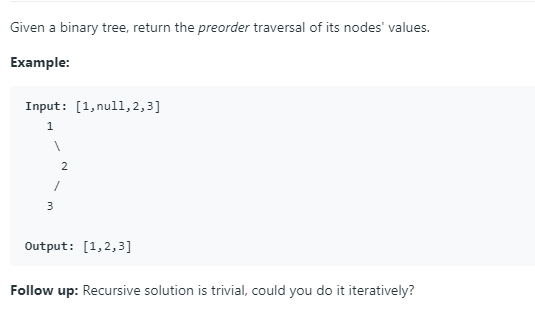
return;

}

};

第一百四十四题：

返回树的先序遍历，要求使用非递归方法。



思路是使用栈。根据中序遍历的思想，只需要在每次到达新节点的时候先读取val即可，从栈中取出的节点都是遍历过的，所以不需要理会。上次吃了不会递归的亏，这次补上递归方法吧，虽然很简单。遍历整棵树一遍，时间复杂度为O（n）。

递归：

/\*\*

\* Definition for a binary tree node.

\* struct TreeNode {

\* int val;

\* TreeNode \*left;

\* TreeNode \*right;

\* TreeNode() : val(0), left(nullptr), right(nullptr) {}

\* TreeNode(int x) : val(x), left(nullptr), right(nullptr) {}

\* TreeNode(int x, TreeNode \*left, TreeNode \*right) : val(x), left(left), right(right) {}

\* };

\*/

class Solution {

public:

void preorder(TreeNode \*root,vector<int> &ans){

if(root==NULL)

return;

ans.push\_back(root->val);

preorder(root->left,ans);

preorder(root->right,ans);

return;

}

vector<int> preorderTraversal(TreeNode\* root) {

vector<int> ans;

preorder(root,ans);

return ans;

}

};

非递归：

/\*\*

\* Definition for a binary tree node.

\* struct TreeNode {

\* int val;

\* TreeNode \*left;

\* TreeNode \*right;

\* TreeNode() : val(0), left(nullptr), right(nullptr) {}

\* TreeNode(int x) : val(x), left(nullptr), right(nullptr) {}

\* TreeNode(int x, TreeNode \*left, TreeNode \*right) : val(x), left(left), right(right) {}

\* };

\*/

class Solution {

public:

vector<int> preorderTraversal(TreeNode\* root) {

vector<int> ans;

TreeNode \*now=root;

stack<TreeNode\*> st;

while(now!=NULL||!st.empty()){

while(now!=NULL){

ans.push\_back(now->val);

st.push(now);

now=now->left;

}

if(!st.empty()){

now=st.top();

st.pop();

now=now->right;

}

}

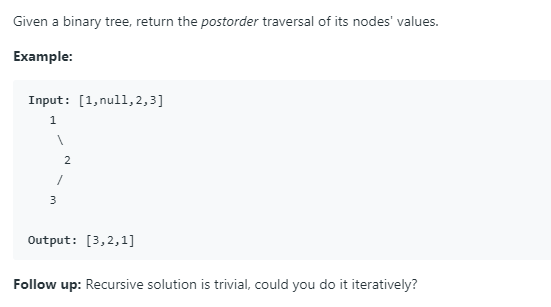
return ans;

}

};

第一百四十五题：

给定一棵二叉树，返回其后序遍历的结果。



思路也是使用栈，后序遍历是最难实现的了，因为需要两个指针。其中一个是now，另外一个是last指向上一次遍历的那个节点，如果我们now的right为空，即这个节点可以读取了；如果我们的now的right为last，这个节点也可以读取了；读取完一个节点之后要用last标记，并且将now悬空，这样才可以正确取得下一个节点；注意，没有读取之前不要pop。时间复杂度为O（n）。

递归：

/\*\*

\* Definition for a binary tree node.

\* struct TreeNode {

\* int val;

\* TreeNode \*left;

\* TreeNode \*right;

\* TreeNode() : val(0), left(nullptr), right(nullptr) {}

\* TreeNode(int x) : val(x), left(nullptr), right(nullptr) {}

\* TreeNode(int x, TreeNode \*left, TreeNode \*right) : val(x), left(left), right(right) {}

\* };

\*/

class Solution {

public:

void postorder(TreeNode \*root,vector<int> &ans){

if(root==NULL)

return;

postorder(root->left,ans);

postorder(root->right,ans);

ans.push\_back(root->val);

return;

}

vector<int> postorderTraversal(TreeNode\* root) {

vector<int> ans;

postorder(root,ans);

return ans;

}

};

非递归：

/\*\*

\* Definition for a binary tree node.

\* struct TreeNode {

\* int val;

\* TreeNode \*left;

\* TreeNode \*right;

\* TreeNode() : val(0), left(nullptr), right(nullptr) {}

\* TreeNode(int x) : val(x), left(nullptr), right(nullptr) {}

\* TreeNode(int x, TreeNode \*left, TreeNode \*right) : val(x), left(left), right(right) {}

\* };

\*/

class Solution {

public:

vector<int> postorderTraversal(TreeNode\* root) {

vector<int> ans;

stack<TreeNode\*> st;

TreeNode \*now=root,\*last;

while(now!=NULL||!st.empty()){

while(now!=NULL){

st.push(now);

now=now->left;

}

if(!st.empty()){

now=st.top();

if(now->right==NULL||now->right==last){

ans.push\_back(now->val);

st.pop();

last=now;

now=NULL;

}

else

now=now->right;

}

}

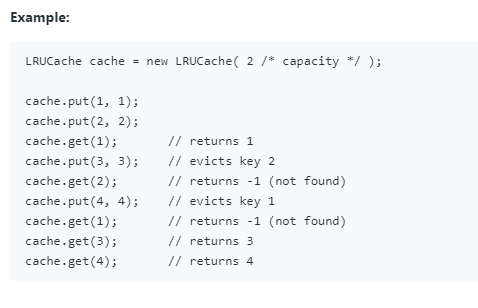
return ans;

}

};

第一百四十六题：

**这题重要。**实现LRU算法。首先会定义一个类，给定容量capacity，然后使用get和put两个方法使用这个类。get（key）返回key对应的value，不存在则返回-1；put（key，value）插入键值对key->value，如果已经满了就去掉最近最久没有访问的那个key->value，然后再插入。要求时间复杂度为O（1）。



思路是这样的，因为查询需要O（1），所以需要能够随机访问的，而且因为有key->value，基本可以锁定哈希表或者红黑树，即map或者unordered\_map；因为涉及到插入删除的操作，如果插入删除都需要O（1）那就需要链表了，因此整体应该是map（或者哈希）+链表。这里有一个小trick，因为删除需要用到key，但是我们put的时候当前应该删除的节点的key，所以要用pair.first来记录这个key；首先创建一个链表head和tail，当每次查询（查询可以调用put实现节点的前移）以及插入的时候，就插入在tail的前面，每次需要删除的时候就移除head的后面那个节点；注意，有可能是插入一个已有的节点，此时也需要移动tail的前面，然后更新值；插入删除已经查找的时间都是O（1），时间复杂度为O（1）。

代码：

class LRUCache {

private:

struct Node{

int val;

Node \*front,\*next;

Node(int v){

val=v;

front=next=nullptr;

}

};

int capacity\_;

int used\_;

unordered\_map<int,pair<int,Node\*>> wh;

Node \*head,\*tail;

public:

LRUCache(int capacity) {

capacity\_=capacity;

used\_=0;

head=new Node(-1);

tail=new Node(-1);

head->next=tail;

tail->front=head;

}

int get(int key) {

if(wh.find(key)==wh.end())

return -1;

Node\* temp=wh[key].second;//这行是不是没用?

put(key,wh[key].first);

return wh[key].first;

}

void put(int key, int value) {

if(wh.find(key)!=wh.end()){

wh[key].first=value;

Node\* temp=wh[key].second;

temp->front->next=temp->next;

temp->next->front=temp->front;

temp->next=tail;

tail->front->next=temp;

temp->front=tail->front;

tail->front=temp;

}

else{

if(used\_==capacity\_){

Node \*temp=head->next;

wh.erase(temp->val);

head->next=temp->next;

temp->next->front=head;

delete temp;

used\_--;

}

Node \*newnode=new Node(key);

wh[key]=make\_pair(value,newnode);

newnode->next=tail;

newnode->front=tail->front;

tail->front->next=newnode;

tail->front=newnode;

used\_++;

}

}

};

/\*\*

\* Your LRUCache object will be instantiated and called as such:

\* LRUCache\* obj = new LRUCache(capacity);

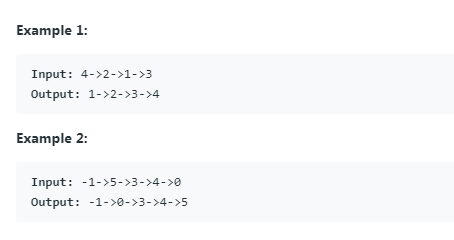
\* int param\_1 = obj->get(key);

\* obj->put(key,value);

\*/

第一百四十七题：

给定一个链表，按照直接插入排序的方法将其进行排序。



思路就是使用指针来执行直接插入排序。首先新建一个dummy表示连接后的头结点，然后从head开始一个接着一个执行直接插入排序；遍历dummy开头的链表，找到当前head插入位置的前一个节点，因为插入可能在两个节点之间，并且head需要先走到下一个需要排序的节点处，所以使用了一个ord；ord指向插入位置的后一个位置，这里有一个好处就是有可能ord为NULL，刚好断开连接；然后将front的next指向head，然后head可以往前移动了，然后front的next的next指向ord；至此就完成了一个节点。遍历整个数字一遍，每次插入还需要遍历一遍别的数组，所以总的时间复杂度为O（n^2）。

代码：

/\*\*

\* Definition for singly-linked list.

\* struct ListNode {

\* int val;

\* ListNode \*next;

\* ListNode() : val(0), next(nullptr) {}

\* ListNode(int x) : val(x), next(nullptr) {}

\* ListNode(int x, ListNode \*next) : val(x), next(next) {}

\* };

\*/

class Solution {

public:

ListNode\* insertionSortList(ListNode\* head) {

ListNode \*dummy = new ListNode(-1),\*front,\*ord;

while(head!=NULL){

front=dummy;

while(front->next&&front->next!=head&&front->next->val<head->val)

front=front->next;

ord=front->next;

front->next=head;

head=head->next;

front->next->next=ord;

}

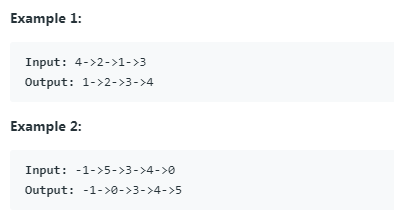
return dummy->next;

}

};

第一百四十八题：

给定一个链表，使用常数空间，时间复杂度为O（nlogn）进行排序。



思路是归并排序，因为快排需要额外的logn的空间。归并的主要思想是，先两个两个排序，然后四个四个，然后不断增加2倍。因为是链表，我们只能从头开始两个两个排序，然后从头四个四个排序。例如四个四个排序，其实是两个一组，将这两组合成有序的四个一组元素。所以我们需要用一个办法标记当前排序的范围。首先，统计有多少个节点cnt，为了限制我们分组的上限；然后使用dummy作为新的头结点；然后用i表示分组的大小，注意，i=1表示当前分组是1个元素一组，但是要将两组合并，所以是2\*i个元素进行排序；然后从头开始，用j表示此次排序的起点的下标，虽然节点没有下标，但是我们根据cnt可以直到是否越界；然后用first指向第一组的第一个，second指向第二组的第一个；因为每一组都有i个元素，所以我们用f和s分别记录这两组多少个元素已经排好序了；排序的过程为从dummy开始一次连接first和second中较小的一个；注意有可能出现其中一组排序完毕，但是另一组还有没排序的元素，需要再次处理；最后second指向了下一组的第一个，我们将已排序的最后一个元素begin指向second，然后j会增加2\*i，继续进行排序。每一次排序都需要O（n），然后最多进行logn次，所以时间复杂度为O（nlogn）。

代码：

/\*\*

\* Definition for singly-linked list.

\* struct ListNode {

\* int val;

\* ListNode \*next;

\* ListNode() : val(0), next(nullptr) {}

\* ListNode(int x) : val(x), next(nullptr) {}

\* ListNode(int x, ListNode \*next) : val(x), next(next) {}

\* };

\*/

class Solution {

public:

ListNode\* sortList(ListNode\* head) {

int cnt=0;

for(ListNode \*p=head;p!=NULL;p=p->next)

cnt++;

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

dummy->next=head;

for(int i=1;i<cnt;i\*=2){

ListNode \*begin=dummy;

for(int j=0;j+i<cnt;j+=i\*2){

ListNode \*first=begin->next,\*second=first;

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

second=second->next;

int f=0,s=0;

while(f<i&&s<i&&second){

if(first->val>second->val){

begin->next=second;

begin=begin->next;

second=second->next;

s++;

}

else{

begin->next=first;

begin=begin->next;

first=first->next;

f++;

}

}

while(f<i){

begin->next=first;

begin=begin->next;

first=first->next;

f++;

}

while(s<i&&second){

begin->next=second;

begin=begin->next;

second=second->next;

s++;

}

begin->next=second;

}

}

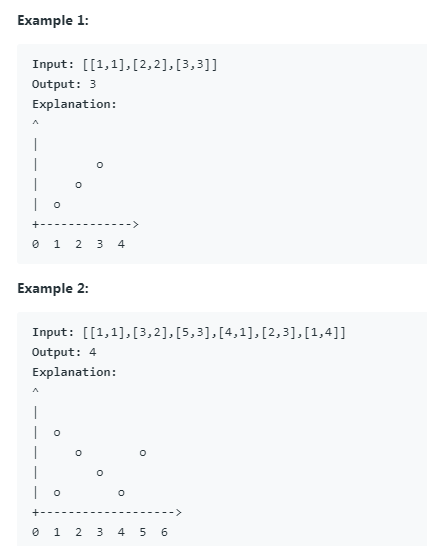
return dummy->next;

}

};

第一百四十九题：

给定很多2维的点，判断最多有多少个点在同一条直线上。



思路是通过斜率来判断两个点是否在同一个直线上。首先声明两个特殊情况，斜率不存在的点需要单独计算，否则会除0；此外，重复的点也要单独计算；前面两种特殊情况的判断方法为两个点的x值相等；如果x值不等，才判断斜率；使用一个哈希表来记录当前斜率有多少个点，注意第一次枚举斜率的时候需要令map[slop]=2，因为包含了我们枚举的这个点此后只需要++；注意这个map应该是每次换了当前点的时候重新声明，否则会有重复，比如前一次枚举[1,1]，与[2,2]共线；然后枚举[2,3]的时候，与[3,4]共线，斜率是一样的，虽然这四个点并不共线，但是会引起错误；此外还需要注意，计算完毕之后要找到wh里面最大的那个值，共线分为maxslop+duplicate和vertical，要取得这两者中较大的一个跟ans比较。遍历整个数组一次，每一次会遍历当前元素后面的点，因此总的时间复杂度为O（n^2）。

代码：

class Solution {

public:

int maxPoints(vector<vector<int>>& points) {

if(points.size()==0)

return 0;

int ans=1;

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

long double slop=0;

int vertival=1,duplicate=0,maxslop=0;

unordered\_map<long double,int> wh;

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

if(points[j][0]==points[i][0]){

vertival++;

if(points[i][1]==points[j][1])

duplicate++;

}

else{

slop=(long double)(points[j][1]-points[i][1])/(points[j][0]-points[i][0]);

if(wh[slop]==0)

wh[slop]=2;

else

wh[slop]++;

}

}

for(auto iter=wh.begin();iter!=wh.end();++iter)

maxslop=max(maxslop,iter->second);

ans=max(vertival,max(ans,maxslop+duplicate));

}

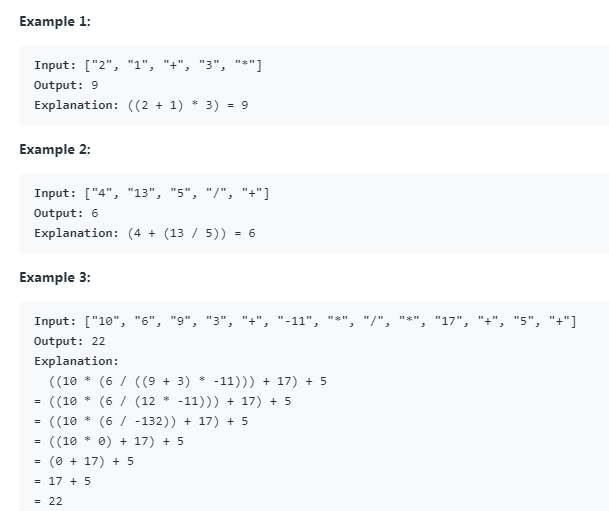
return ans;

}

};

第一百五十题：

给定一个数组包含数字和运算符，组成一个逆波兰表达式，计算这个表达式并且返回结果



思路是使用一个栈装载所有的数字，如果遇到运算符就取出头部的两个数字进行相应的运算，然后将结果压入栈中。最后栈中剩余的数字就是运算的结果。遍历一遍整个数组，运算次数也不会超过数组，所以时间复杂度为O（n）。这里我犯了一个错，判定是否是运算符，可以直接比较string，例如tokens[i]==“-”如果为true就是减号，而不是负数！此外，我可以先取出数字，然后在switch选择运算符；

代码：

class Solution {

public:

int evalRPN(vector<string>& tokens) {

stack<int> sta;

for (auto &t : tokens)

if (t == "+" || t == "-" || t == "\*" || t == "/")

{

int a = sta.top();

sta.pop();

int b = sta.top();

sta.pop();

if (t == "+") sta.push(a + b);

else if (t == "-") sta.push(b - a);

else if (t == "\*") sta.push(a \* b);

else sta.push(b / a);

}

else sta.push(atoi(t.c\_str()));

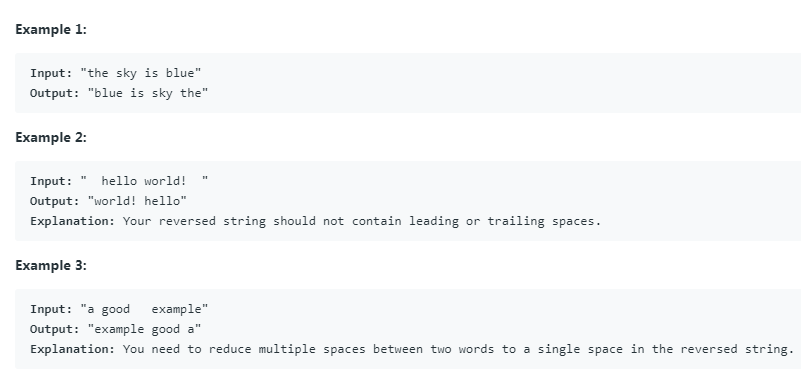
return sta.top();

}

};

第一百五十一题:

将一个字符串以空格为分割，所有单词逆序输出，单词本身不要逆序。前后的空格不要。



思路是使用一个栈，将所有的单词入栈，最后再连接起来。使用一个start和end来标记这一个单词的起始和结束下标，这里需要注意如果遇到空格，这时候会有一个空的temp，不要让空的temp入栈；结束的时候记得不要漏掉最后一个单词（这里好像可以在原s后面加一个空格避免代码重复？）。然后就会用ans拼接每一个栈中单词，最后如果ans不为空则去掉最后一个空格。遍历整个字符串一次，然后每个单词入栈出栈一次，所以时间复杂度为O（n）。（如果要求常数空间，可以考虑直接整个字符串逆序，然后以空格为分割每个单词分别逆序。）

代码：

class Solution {

public:

string reverseWords(string s) {

stack<string> st;

int start=0,end=0;

string ans,temp;

for(;end<s.size();++end)

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

temp=s.substr(start,end-start);

if(temp.size()>0)

st.push(temp);

start=end+1;

}

if(end>start)

st.push(s.substr(start,end-start));

while(!st.empty()){

temp=st.top();

ans+=temp+' ';

st.pop();

}

if(ans.size()&&ans[ans.size()-1]==' ')

ans.erase(ans.size()-1);

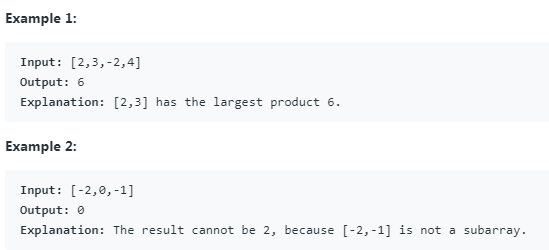
return ans;

}

};

第一百五十二题：

给定一个数组，返回其乘积最大的子集的乘积。



思路是维护一个滑动窗口。难点在于什么时候窗口可以往前移动，负数的情况怎么处理。首先我们知道0是天然的分割点，因此可以将0插在最后统一操作，但是要注意这个0是不存在的。我们用start指示当前乘积区间的开始下标，首先如果i不是最后那个插入的0，我们令ans=max（ans，nums[i]），因为有可能乘积最大的是一个负数或者一个0；然后检查这个元素是不是0，如果不是就令temp\*=nums[i]；如果是，先检查temp是不是>0并且start<i，是的话就直接更新ans和start然后开始下一个循环；如果小于0并且start<i，就将当前的temp从头尾两端开始不断除以nums[i]，直到为正数，更新ans；这么做是因为乘积最大肯定越长越好，去掉头或者尾的第一个负数肯定就是当时区间的最优解。遍历整个数组一遍，去掉头尾最坏情况下也要遍历一遍数组，所以总体的时间复杂度为O（n）。

代码：

class Solution {

public:

int maxProduct(vector<int>& nums) {

if(nums.size()==0)

return 0;

int front=0,ans=INT\_MIN,temp=1;

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

ans=max(ans,nums[i]);

if(nums[i]!=0){

temp\*=nums[i];

ans=max(ans,temp);

}

if(nums[i]==0||i==nums.size()-1){

if(temp<0){

int step=front;

while(step<i-1&&temp<0)

temp/=nums[step++];

ans=max(ans,temp);

}

temp=1;

front=i+1;

}

}

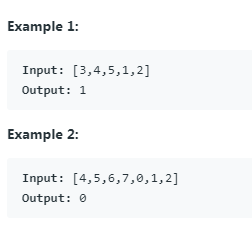
return ans;

}

};

第一百五十三题：

给定一个不知道旋转轴的不含重复数字的数组，找到其中最小的一个值。



思路是二分法。首先如果最后一个值大于第一个值，说明没有翻转，直接返回nums[0]。如果有翻转，就直接二分，有两种情况：①mid的值大于等于nums[0]，说明此时最小值在mid的右边，并且mid的值不会是最小值，所以直接left=mid+1；②如果mid的值小于nums[0]，说明最小的值要不就是mid要不就在左边的一半，所以直接right=mid；二分查找的复杂度为O（logn）。

代码：

class Solution {

public:

int findMin(vector<int>& nums) {

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

if(nums[left]<nums[right])

return nums[left];

while(left<right){

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

if(nums[mid]>=nums[0])

left=mid+1;

else

right=mid;

}

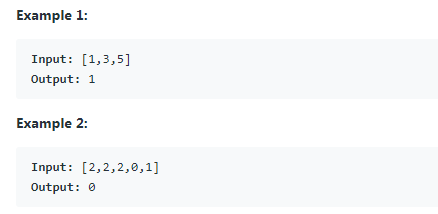
return nums[left];

}

};

第一百五十四题：

给定一个不知旋转轴的数组，元素有可能重复，返回其中最小的一个元素。



思路跟上一题比较类似，但是有一些区别。由于重复元素的出现，数组头尾可能元素的值是一样的，这就导致无法判断二分的区间选择；第一种方法，先去掉尾部与nums[0]相等的元素，然后再跟上一题一样二分；第二种方法比较巧妙，我们使用nums[right]作为judge，当nums[mid]比nums[right]大的时候，这个mid必然不是解，那就left=mid+1；如果nums[mid]比nums[right]小的时候，这个mid需要保留，那就right=mid；当nums[mid]跟nums[right]相等，我们其实不知道mid是在哪一部分，保险起见，将right--；最坏情况下所有元素都是一样的，每一次只会减少一个元素，因此时间复杂度为O（n）。

代码：

class Solution {

public:

int findMin(vector<int>& nums) {

if(nums.size()==0)

return 0;

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

while(left<right){

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

if(nums[right]<nums[mid])

left=mid+1;

else if(nums[right]>nums[mid])

right=mid;

else

right--;

}

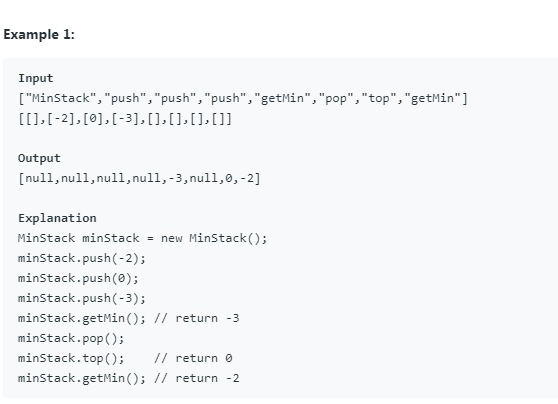
return nums[left];

}

};

第一百五十五题：

实现一个栈，有push、pop、top、getmin函数，其中getmin返回当前栈中最小的元素。



思路比较特别。当我们push的时候，先看看这个元素是否小于等于我们的最小值，如果是，则先将当前最小值入栈，然后将最小值设置为当前值，最后将当前值入栈；如果不是，正常入栈；出栈的时候，首先检查出栈元素是否等于最小值，是的话再次pop并且将最小值设置为这个元素。

代码：

class MinStack {

public:

/\*\* initialize your data structure here. \*/

int minnum\_;

stack<int> st\_;

MinStack() {

minnum\_=INT\_MAX;

}

void push(int x) {

if(x<=minnum\_){

st\_.push(minnum\_);

minnum\_=x;

}

st\_.push(x);

}

void pop() {

int temp=st\_.top();

st\_.pop();

if(temp==minnum\_){

minnum\_=st\_.top();

st\_.pop();

}

}

int top() {

return st\_.top();

}

int getMin() {

return minnum\_;

}

};

/\*\*

\* Your MinStack object will be instantiated and called as such:

\* MinStack\* obj = new MinStack();

\* obj->push(x);

\* obj->pop();

\* int param\_3 = obj->top();

\* int param\_4 = obj->getMin();

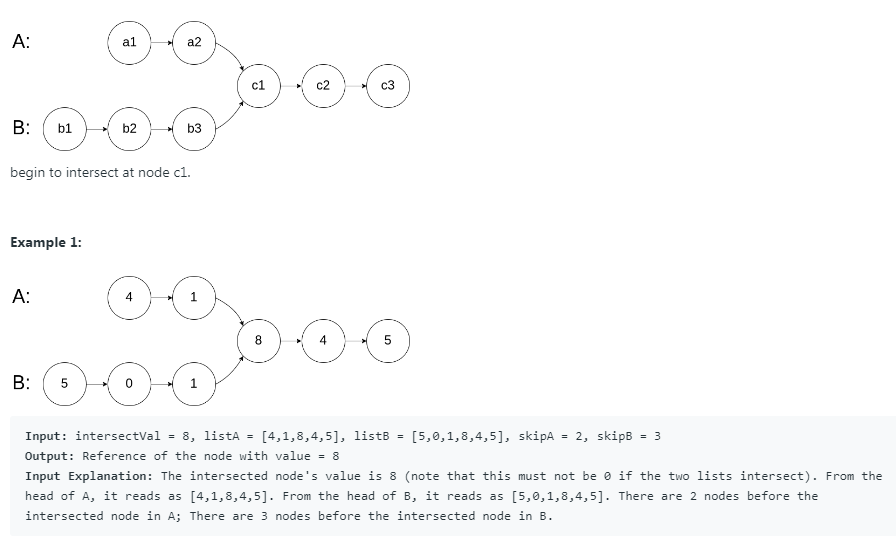
\*/

第一百五十六题~第一百五十九题：

有锁

第一百六十题：

**这题重要。**给定两个链表，如果两个链表有交汇点，就返回这个交汇点；如果没有，返回NULL。



思路很奇特。首先两个链表同时每次往前移动一步，如果某个到达了NULL，就从对方的头结点接着开始走，直到两个指针相等为止。最后这个指针如果不为空，就是交汇点，如果为空，就是没有交汇点。为什么呢？例如，交汇的那段路程为c，A链和B链独立的那段路程分别为a，b；那么两个指针都走了a+b+2c这么长的距离，在第二次到达交汇点的时候，两个指针都走了a+b+c这么长的距离，所以肯定会相等，否则就是没有交汇点了。遍历两个链表两遍，时间复杂度为O（n）。

代码：

/\*\*

\* Definition for singly-linked list.

\* struct ListNode {

\* int val;

\* ListNode \*next;

\* ListNode(int x) : val(x), next(NULL) {}

\* };

\*/

class Solution {

public:

ListNode \*getIntersectionNode(ListNode \*headA, ListNode \*headB) {

ListNode \*tempA=headA,\*tempB=headB;

while(tempA!=tempB){

if(tempA)

tempA=tempA->next;

else

tempA=headB;

if(tempB)

tempB=tempB->next;

else

tempB=headA;

}

return tempA;

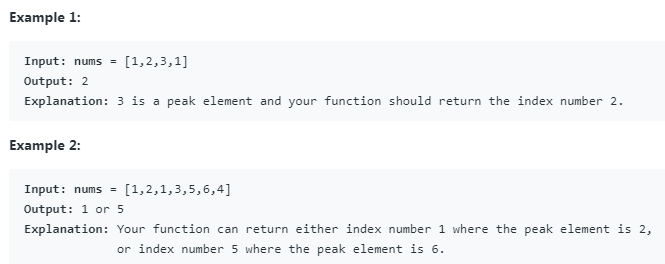
}

};

第一百六十一题：锁

第一百六十二题：

给定一个数组，其中nums[i]！=nums[i+1]，假设nums[-1]==nums[n]==-∞，要求找到随便一个元素，其大于左右两边的元素，返回其下标。要求时间复杂度为logn。



思路很清晰，就是二分。当我们二分之后，先看看mid+1是否大于mid，如果是，说明mid右边出现符合条件的值的概率更大，此时f=mid+1；否则就让r=mid；思考可能越界情况，因为涉及到mid+1，所以mid绝对不能为n-1，首先r初始地址就是n-1，如果mid==r，说明f与r重叠了，此时不会继续循环，因此不存在越界；时间复杂度为O（logn）。

代码：

class Solution {

public:

int findPeakElement(vector<int>& nums) {

int f=0,r=nums.size()-1;

while(f<r){

int mid=f+(r-f)/2;

if(nums[mid]<nums[mid+1])

f=mid+1;

else

r=mid;

}

return f;

}

};

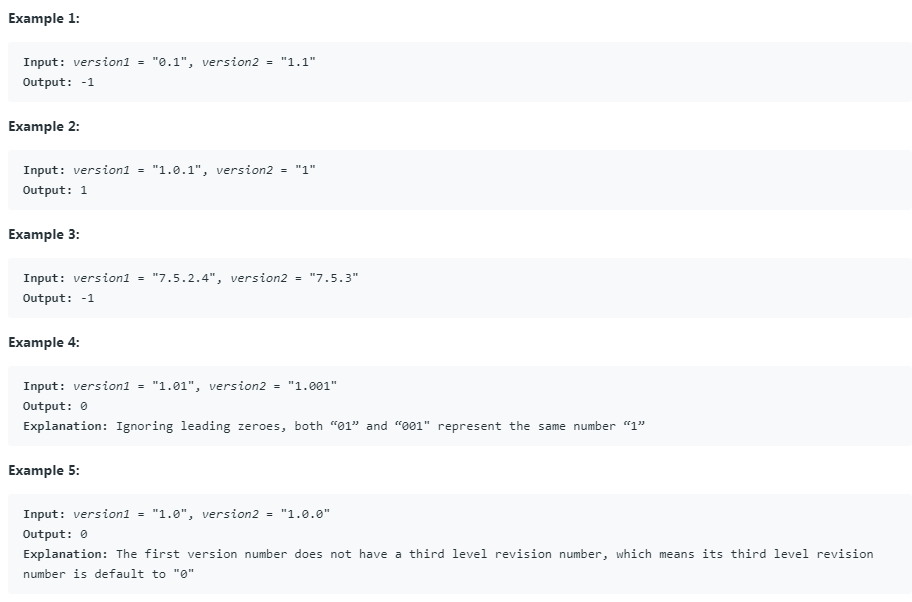
第一百六十三题：锁

第一百六十四题：

暂时不管

第一百六十五题：

给定两个字符串版本号，以‘.’分割，如果version1>version2就返回1，如果version1<version2就返回-1，相等返回0；如果其中有一个缺失，就默认为0；



思路是使用一个队列存储数据，逐个比较。因为是从前往后比较，所以使用队列可以更好组织数据，首先以‘.’为分隔符将两个字符串的元素保存起来，注意不要漏了最后一个元素；然后设置默认值为0，如果队列不为空就更新其默认值val1或者val2，逐个比较就ok了。遍历一遍两个字符串，然后比较字符串一次，所说时间复杂度为O（n）。

代码：

class Solution {

public:

int compareVersion(string version1, string version2) {

queue<int> q1,q2;

int start=0,i=0;

for(i=0;i<version1.size();++i)

if(version1[i]=='.'&&i>start){

string temp=version1.substr(start,i-start);

q1.push(stoi(temp));

start=i+1;

}

if(start<i)

q1.push(stoi(version1.substr(start,i-start)));

for(i=0,start=0;i<version2.size();++i)

if(version2[i]=='.'&&i>start){

string temp=version2.substr(start,i-start);

q2.push(stoi(temp));

start=i+1;

}

if(start<i)

q2.push(stoi(version2.substr(start,i-start)));

while(!q1.empty()||!q2.empty()){

int val1=0,val2=0;

if(!q1.empty()){

val1=q1.front();

q1.pop();

}

if(!q2.empty()){

val2=q2.front();

q2.pop();

}

if(val1>val2)

return 1;

else if(val1<val2)

return -1;

}

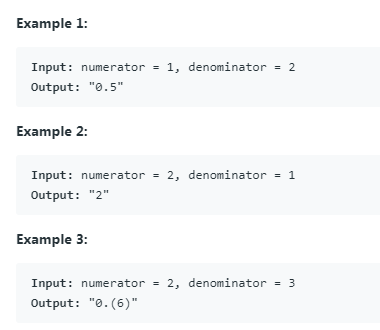
return 0;

}

};

第一百六十六题：

给定一个分子和分母，计算其小数表示，如果有循环部分用括号包裹。



思路是精确的除法计算。首先将其转换为longlong，因为有可能为负数，INT\_MIN不能转换为int，所以要直接转换为longlong之后再判断结果的正负；我们知道，当余数重复出现的时候，由于分母是不变的，所以余数再次出现，说明余数第一次出现的地方就是循环开始的地方，所以用一个哈希表记录每一个余数出现的下标；如果找到了，说明这个余数第二次出现了，所以循环部分就是ans中该余数第一次出现到末尾的部分，剩余部分可以保留，然后要break；如果余数为0了，就可以跳出来，最后再检查是整数还是负数；时间复杂度为O（n）。

代码：

class Solution {

public:

string fractionToDecimal(int \_numerator, int \_denominator) {

string ans;

bool minus=false;

unordered\_map<int,int> wh;

long long numerator=\_numerator,denominator=\_denominator;

if(numerator<0){

minus=!minus;

numerator=-numerator;

}

if(denominator<0){

minus=!minus;

denominator=-denominator;

}

ans=to\_string(numerator/denominator);

long long last=numerator%denominator;

if(last==0)

return stoll(ans,NULL,10)>0&&minus?'-'+ans:ans;

ans+='.';

while(last){

if(wh[last]){

ans=ans.substr(0,wh[last])+'('+ans.substr(wh[last])+')';

return minus?'-'+ans:ans;

}

wh[last]=ans.size();

ans+=to\_string(last\*10/denominator);

last=last\*10%denominator;

}

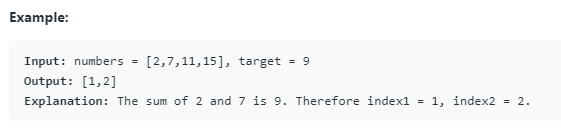
return minus?'-'+ans:ans;

}

};

第一百六十七题：

给定一个升序数组和一个target，找到两个元素加起来等于target，返回这两个元素的下标（下标需要+1）。



思路是从两边开始往里面缩小。由于说明了必定有解，所以我没检查边界。首先初始化l=0，r=numbers.size()-1，然后用temp将l和r指定的两个数加起来，如果等于target就找到了，否则大了就r--，小了就l++；最坏情况下遍历整个数组，所以时间复杂度为O（n）。

代码：

class Solution {

public:

vector<int> twoSum(vector<int>& numbers, int target) {

int l=0,r=numbers.size()-1;

while(l<r){

int temp=numbers[l]+numbers[r];

if(temp==target)

break;

else if(temp>target)

r--;

else

l++;

}

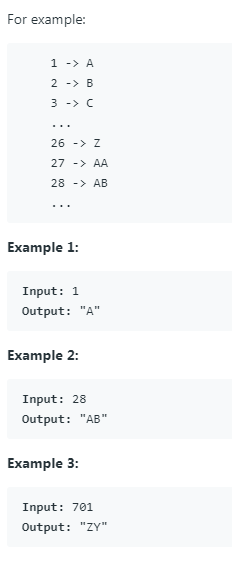
return vector<int>{l+1,r+1};

}

};

第一百六十八题：

给定一个正整数n，转换为26进制的A-Z。



思路就是十进制转26进制，这里的1-26其实也就对应0-25，就是一个26进制转换了。所以肯定是对26取模，但是有一个地方需要进行调整，要对正整数n减1处理。因为如果n=26，我们对26取模会得到26进制表示1 0，但是答案是需要0 25，所以就是n-1处理。这里不断对n进行26取模，所以时间复杂度为O（logn）。

代码：

class Solution {

public:

string convertToTitle(int n) {

string res;

while (n) {

res = char ((n - 1) % 26 + 'A') + res;

n = (n - 1) / 26; //注意corner case: n=26

}

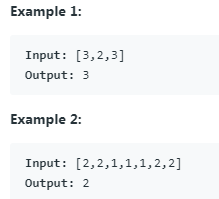
return res;

}

};

第一百六十九题：

给定一个数组，有一个元素个数占了整个数组的一半以上，找到这个元素。



思路是不断进行抵消。首先初始化一个cnt=0，一个candidate；如果cnt==0，candidate=nums[i]；如果candidate==nums[i]，就cnt++；否则cnt--；这样一来，只要遇到不相同的元素，就会相互抵消；即使所有元素相间，占了一半以上的元素总会最后剩下一个；别的情况就更加显而易见了。遍历一遍整个数组，所以时间复杂度为O（n）。

代码：

class Solution {

public:

int majorityElement(vector<int>& nums) {

int candidate,cnt=0;

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

if(cnt==0)

candidate=nums[i];

if(nums[i]==candidate)

cnt++;

else

cnt--;

}

return candidate;

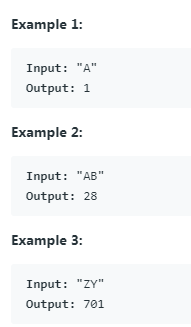
}

};

第一百七十题：

第一百七十一题：

将26进制转换为10进制，是168题的逆向。



思路是直接转换。有一个小点需要注意，A不是0而是1，所以计算的时候需要对结果+1；可以正向也可以反向；正向就是从高位开始，每一次都\*26再加上这一位的数；反向就是设定一个此位对应的值，最低位是26^0，然后是26^1，以此类推。遍历一次整个字符串，所以时间复杂度为O（n）。

代码：

class Solution {

public:

int titleToNumber(string s) {

int num = 0;

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

num = 26 \* num + (s[i] - 'A' + 1); //注意'A'不是0而是1

}

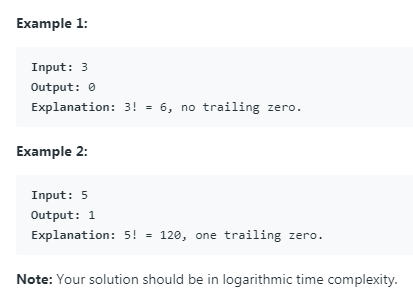
return num;

}

};

第一百七十二题：

给定一个正整数n，返回其阶乘n！的末尾有几个0。



这是一道数学题。首先枚举一下就知道，末尾的零是因为有2和5相乘得到的，因为能整除2的数字比能整除5的数字多很多，而2要配合一个5才能得到一个0，所以我们统计给定的数字n里面能提供多少个质因子5就行了。直接n/5就可以得到其中一部分，除此之外，像25这样的数字能提供2个5，其中一个刚才n/5已经计算过了，所以这里直接n/25又可以得到一部分；以此类推，125、625、...、5^n都要分别计算。最后相加就是最后结果。每次对给定n/5，所以时间复杂度为O（logn）。

代码：

class Solution {

public:

int trailingZeroes(int n) {

int cnt=0;

while(n){

cnt+=n/5;

n/=5;

}

return cnt;

}

};

第一百七十三题：

给定一个类，补全三个成员函数。该类存储一棵树，并且实现了中序遍历。



思路是使用栈进行中序遍历。初始化时就需要找到中序遍历第一个节点，就是不断的push进stack就行了，这里使用栈进行中序遍历，只有调用next的时候才进行读取，然后找到下一个中序遍历的节点，以此类推。中序遍历的时间复杂度为O（n），这里每次调用next的时间复杂度为O（1）。

代码：

/\*\*

\* Definition for a binary tree node.

\* struct TreeNode {

\* int val;

\* TreeNode \*left;

\* TreeNode \*right;

\* TreeNode() : val(0), left(nullptr), right(nullptr) {}

\* TreeNode(int x) : val(x), left(nullptr), right(nullptr) {}

\* TreeNode(int x, TreeNode \*left, TreeNode \*right) : val(x), left(left), right(right) {}

\* };

\*/

class BSTIterator {

public:

TreeNode \*nextmin;

stack<TreeNode\*> st;

BSTIterator(TreeNode\* root) {

nextmin=root;

while(nextmin){

st.push(nextmin);

nextmin=nextmin->left;

}

}

/\*\* @return the next smallest number \*/

int next() {

TreeNode \*temp=st.top();

int ans=temp->val;

st.pop();

temp=temp->right;

while(temp){

st.push(temp);

temp=temp->left;

}

return ans;

}

/\*\* @return whether we have a next smallest number \*/

bool hasNext() {

return !st.empty();

}

};

/\*\*

\* Your BSTIterator object will be instantiated and called as such:

\* BSTIterator\* obj = new BSTIterator(root);

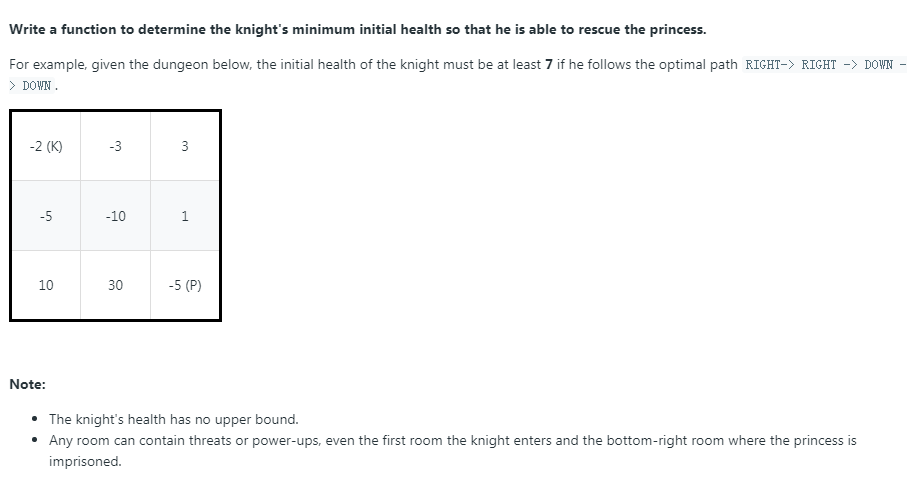
\* int param\_1 = obj->next();

\* bool param\_2 = obj->hasNext();

\*/

第一百七十四题：

**这题重要。**给定一个2D图，从左上角出发，每经过一格都会减少或者增加相应的血量，返回到达右下角所需要的的最小初始血量。

思路很难想。简单来说，就是初始血量为多少的时候，从左上角到右下角的过程中，最少会剩余1点血。由于从起点开始可选的路径太多，我们很难去分辨哪一条最优，但是**从终点出发，我们可以假设到达终点的最低血量为1，然后往回走找到一条路径中途需要的血量最少。**所以我们假设wh（i，j）表示从（i，j）这个点到达终点所需要的的最少的血量。wh（i，j）所需要的最少血量为min（wh（i+1，j），wh（i，j+1））-dungeon（i，j），因为wh（i+1，j）和wh（i，j+1）都是已知的到达终点所需要的最低血量，那么wh（i，j）所需的最低血量自然就是扣除dungeon（i，j）之后到达min（wh（i+1，j），wh（i，j+1））。所以转移方程为wh（i，j）=min（wh（i+1，j），wh（i，j+1））-dungeon（i，j），注意，如果wh（i，j）小于等于0，说明到这里的时候可能血量会剩余很多，但是为了计算准确，任意wh（i，j）至少为1，也就是每一个点最少需要1点血。需要填写一个2D数组，所以时间复杂度为O（n^2）。

代码：

class Solution {

public:

int calculateMinimumHP(vector<vector<int>>& dungeon) {

int row=dungeon.size()-1,col=dungeon[0].size()-1;

vector<vector<int>> wh(row+1,vector<int>(col+1,0x3f3f));

wh[row][col]=1-dungeon[row][col];

for(int i=row;i>=0;--i)

for(int j=col;j>=0;--j){

if(i<row)

wh[i][j]=wh[i+1][j]-dungeon[i][j];

if(j<col)

wh[i][j]=min(wh[i][j],wh[i][j+1]-dungeon[i][j]);

wh[i][j]=wh[i][j]<=0?1:wh[i][j];

}

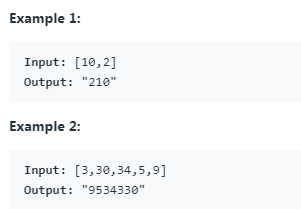
return wh[0][0];

}

};

第一百七十九题：

**这题重要。**给定一个数组nums，返回其能组成的最大的数



思路是先排序后拼接。首先我们要知道如何确定两个数字的顺序，其实就是从高位到逐位比较，直到有一个数较大就会在前，例如34应该在33或者3前面；这其实就是我们的strcmp函数的功能，C++中直接比较string的可以实现；而且，如果A在B前面，B在C前面，那么A肯定在C前面，因此我们知道了两个数字的顺序，就可以利用这个关系进行排序；排序之后拼接就是最后结果；注意一个情况，前导的0需要去掉，这里我使用的是计数的方法，其实还有很多方法；排序的时间复杂度为O（nlogn），拼接以及去除0都是O（n），因此总的时间复杂度为O（nlogn）。

代码：

class Solution {

public:

string largestNumber(vector<int>& nums) {

sort(nums.begin(),nums.end(),[](int x,int y){

string stra = to\_string(x);

string strb = to\_string(y);

return stra+strb > strb+stra;

});

string ans;

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

ans+=to\_string(nums[i]);

int count=0;

while(count<ans.size()){

if(ans[count]!='0')

break;

count++;

}

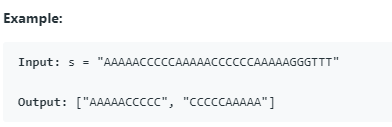
return count<ans.size()?ans.substr(count):"0";

}

};

第一百八十七题：

给定一个只包含ACGT的字符串，以10个字符的子串为单位，返回出现次数超过1次的这样的子串。



思路是使用一个哈希表将我们已经见过的子串保存起来，如果再次遇到就可以入队了。注意一点，因为有可能10个字符的子串重复出现多次，所以哈希表只有出现第二次的时候记录，第三次第四次都不作记录。遍历一遍整个字符串，所以时间复杂度为O（n），每个子串都需要存储，总共需要10n，所以空间复杂度为O（n）。

代码：

class Solution {

public:

vector<string> findRepeatedDnaSequences(string s) {

if(s.size()<10)

return {};

int head=s.size()-10;

vector<string> ans;

unordered\_map<string,int> wh;

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

{

string temp=s.substr(i,10);

if(wh[temp]==1)

ans.push\_back(temp);

wh[temp]++;

}

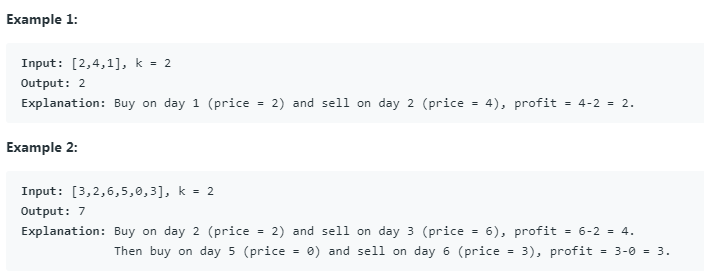
return ans;

}

};

第一百八十八题：

给定一个k，和每天股票价格，返回交易k次股票获取的最大利润。



思路是动态规划。用withoutstock[i][j]表示第i天交易j次股票，并且没有持有股票时的利润；withstock[i][j]表示第i天交易j次股票，并且持有股票时利润；转移方程

①withoutstock[i][j]=max（withoutstock[i-1][j]，withstock[i-1][j-1]+prices[i]），第i天交易j次的利润为第i-1天交易j次的利润或者第i-1天交易j-1次，然后卖掉的利润；

②withstock[i][j]=max（withstock[i-1][j]，withoutstock[i-1][j]-prices[i]），第i天交易j次持有股票的利润为第i-1天交易j次持有股票的利润或者第i-1天交易j次，然后买进的利润；这里i&1如果i为偶数为0，如果i为奇数就为1；因为每一天的利润只依赖于前一天的利润，所以我们只需要两行数据就好了；每一天都要计算k次利润，总共有n天，所以时间复杂度为O（nk）。因为不能在同一天卖出再买入，所以n天只能买卖n/2次，所以k>=n/2的话，就相当于每次都买入卖出。

代码：

class Solution {

public:

int maxProfitEasy(int k, vector<int>& prices){

int start=0,ans=0;

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

if(prices[i]<prices[i-1]){

if(start+1<i)

ans+=prices[i-1]-prices[start];

start=i;

}

if(i==prices.size()-1&&prices[start]<prices[i])

ans+=prices[i]-prices[start];

}

return ans;

}

int maxProfit(int k, vector<int>& prices) {

int len=prices.size(),ans=0,judge=-0x3f3f;

if(k>=len/2)

return maxProfitEasy(k,prices);

vector<vector<int>> withoutstock(2,vector<int>(k+1,judge)),withstock(2,vector<int>(k+1,judge));

withoutstock[0][0]=0;

for(int i=1;i<=len;++i)

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

withoutstock[i&1][j]=withoutstock[i-1&1][j];

withstock[i&1][j]=max(withstock[i-1&1][j],withoutstock[i-1&1][j]-prices[i-1]);

if(j>0)

withoutstock[i&1][j]=max(withoutstock[i&1][j],withstock[i-1&1][j-1]+prices[i-1]);

}

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

ans=max(ans,withoutstock[len&1][i]);

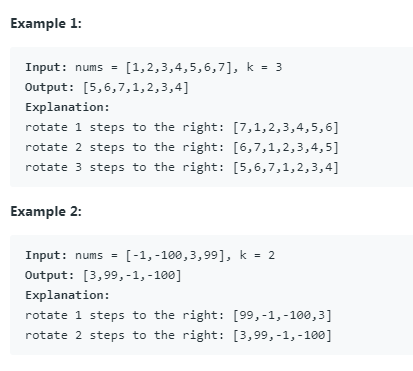
return ans;

}

};

第一百八十九题：

给定一个数组，和一个正整数k，将倒数k个数字移到前面去，相当于一个环。



思路是多次翻转。首先，我们将整个数组翻转，此时末尾的数字就到前面了，然后我们将前k个数翻转，剩余的数字也翻转就完成了。这里的k可能很大，所以需要先对nums.size()取模。所有数字都翻转了两次，所以时间复杂度为O（n）。

代码：

class Solution {

public:

void rotate(vector<int>& nums, int k) {

k=k%nums.size();

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

reverse(nums.begin(),nums.begin()+k);

reverse(nums.begin()+k,nums.end());

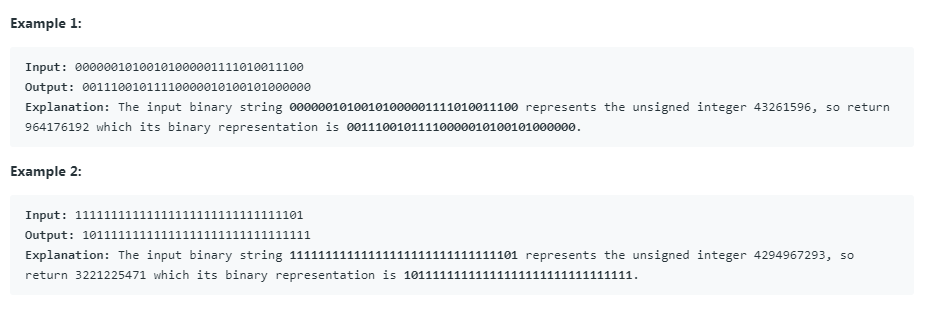
return;

}

};

第一百九十题：

给定一个32位无符号整数，将其翻转然后返回。



思路是位运算。因为都是二进制，所以检查某一位是否为1可以直接i&1，如果要在末尾加一个1可以直接ans=ans<<1+1；因此可以直接一步到位。最多运算32次。时间复杂度为O（1）。

代码：

class Solution {

public:

uint32\_t reverseBits(uint32\_t n) {

uint32\_t ans=0;

for(int i=0;i<32;++i)

ans=(ans<<1)+(n>>i&1);

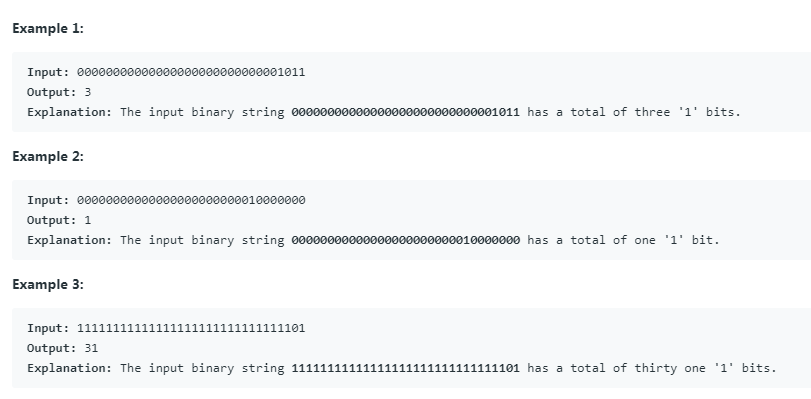
return ans;

}

};

第一百九十一题：

统计一个32位无符号整数有多少个1.



思路很简单了，就是位运算。一直&1，直到n为0，期间统计1的个数就行。最多运算32次，所以时间复杂度为O（1）。

代码：

class Solution {

public:

int hammingWeight(uint32\_t n) {

int count=0;

while(n){

count+=(n&1);

n>>=1;

}

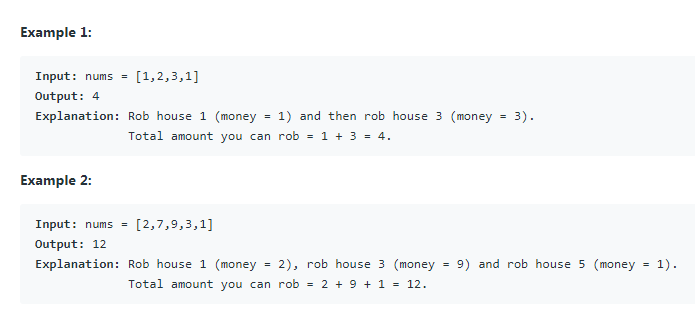
return count;

}

};

第一百九十八题：

给定一个数组，不能取彼此相邻的数字，问最终取得的最大值为多少。



思路是动态规划。对于每一数，要么取，要么不取，为了取得最大值，我们需要知道当前数字取和不取的最大值。设rob为取当前数的最大收益，初始值为nums[0]；notrob为不取当前值的最大收益，初始值为0；更新时，rob=前一个元素不取+nums[i]，notrob=max（前一个元素取，前一次不取的最大收益）；**这里的notrob更新需要注意，因为允许连续两次不取，因此不一定取了前一个元素收益更大，前一个元素不取的收益可能更大**；因此最终答案为max（rob，notrob）。遍历一遍整个数组，时间复杂度为O（n）。

代码：

class Solution {

public:

int rob(vector<int>& nums) {

if(nums.size()==0)

return 0;

int rob=nums[0],notrob=0;

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

int yesterdayRob=rob,yesterdayNotRob=notrob;

rob=yesterdayNotRob+nums[i];

notrob=max(yesterdayRob,yesterdayNotRob);

}

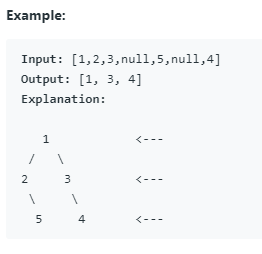
return max(rob,notrob);

}

};

第一百九十九题：

给定一棵二叉树，返回从右边看到的节点的值。



思路还是挺好想的，我的方法是用两个队列进行层次遍历，每一次都将末尾的元素入队。看到一个更简洁的做法：因为每一层只有一个数是答案，所以可以通过一个level标记当前的floor，跟ans.size()比较就知道这一层的元素是否已经确定。本来应该用level来确定是否需要入ans，这里使用了ans.size作为判断依据，值的学习。然后使用递归检查，每次都优先进入右子节点，确保答案正确。每个节点遍历一次，时间复杂度为O（n）。

代码：

/\*\*

\* Definition for a binary tree node.

\* struct TreeNode {

\* int val;

\* TreeNode \*left;

\* TreeNode \*right;

\* TreeNode() : val(0), left(nullptr), right(nullptr) {}

\* TreeNode(int x) : val(x), left(nullptr), right(nullptr) {}

\* TreeNode(int x, TreeNode \*left, TreeNode \*right) : val(x), left(left), right(right) {}

\* };

\*/

class Solution {

public:

void solveit(TreeNode \*root,int level,vector<int> &ans){

if(root==NULL)

return;

if(level==ans.size())

ans.push\_back(root->val);

solveit(root->right,level+1,ans);

solveit(root->left,level+1,ans);

return;

}

vector<int> rightSideView(TreeNode\* root) {

vector<int> ans;

solveit(root,0,ans);

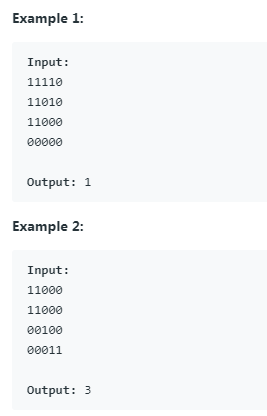
return ans;

}

};

第二百题：

给定一个二维数组，1表示陆地，0表示海洋，水平或者垂直相连的1表示同一块陆地，返回这里有多少块陆地。



思路是深度优先遍历。真的是老题目了，很经典的深度优先遍历题目。因为同一块陆地都是水平或者垂直相连的，我们可以将同一块陆地全部从1改成0，然后继续遍历，每一次遇到1就表示是一块新的陆地。最后返回count。遍历一遍整个数组，同时需要进行深度优先遍历，最多最多深度优先遍历一次整个地图，检查遍历一次，所以时间复杂度为O（n^2）。

代码：

class Solution {

public:

void labelisland(int i,int j, vector<vector<char>> &grid){

grid[i][j]='0';

if(i>0&&grid[i-1][j]=='1')

labelisland(i-1,j,grid);

if(j>0&&grid[i][j-1]=='1')

labelisland(i,j-1,grid);

if(i<grid.size()-1&&grid[i+1][j]=='1')

labelisland(i+1,j,grid);

if(j<grid[0].size()-1&&grid[i][j+1]=='1')

labelisland(i,j+1,grid);

return;

}

int numIslands(vector<vector<char>>& grid) {

if(grid.size()==0)

return 0;

int count=0;

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

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

if(grid[i][j]=='1'){

labelisland(i,j,grid);

count++;

}

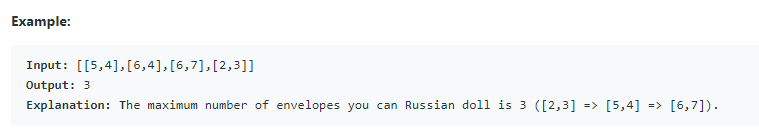
return count;

}

};

第三百五十四：

给定一个二维数组，从中选择若干个，要求选出来的元素在两个维度上均是单调递增的，不允许有重复的数字；



这题是动态规划。**首先按照第一维排序，然后可以从第二维中找到一个单调递增的子序列，同时在第一维中不允许有重复**；这题的难点在于如何保证第一维不重复的情况下，正确找到单调递增子序列。定义dp[i]表示envelopes[i]这个元素在单调递增子序列里面排第几；judge[i]表示长度为i的单调递增子序列的末尾元素；我们用i进行遍历，用j进行检查；只有当第一维不重复的时候，才可能让单调递增子序列边长，如果第一维相同，最多只能占据同一个位置；例如[6,4]可以占据第3个位置，①当下一个元素为[6,7]的时候，由于第一维相同，所以还是只能占据第3个位置；②当下一个元素为[7,5]的时候，由于第一维不同，因此可以占据第4个位置；③如果检查[7,5]的时候，其前面有一个[6,3]，那只会将judge第3个位置更新为3；排序所需的时间复杂度为O（nlogn），然后进行dp遍历一遍数组需要O（n），每一次都需要进行二分查找，时间复杂度为O（logn），因此总的时间复杂度为O（nlogn）。

代码：

class Solution {

public:

int maxEnvelopes(vector<vector<int>>& envelopes) {

if(envelopes.size()<2)

return envelopes.size();

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

int ans=0;

vector<int> dp(envelopes.size()+1,0),judge{0};

for(int i=0,j=0;i<envelopes.size();++i){

if(envelopes[i][0]!=envelopes[j][0])

while(j<i){

if(dp[j]==judge.size())

judge.push\_back(envelopes[j][1]);

else

judge[dp[j]]=min(judge[dp[j]],envelopes[j][1]);

j++;

}

int l=0,r=judge.size();

while(l<r){

int mid=l+(r-l)/2;

if(judge[mid]<envelopes[i][1])

l=mid+1;

else

r=mid;

}

dp[i]=l;

ans=max(ans,dp[i]);

}

return ans;

}

};