简单：

1. 给定一个长度为 n 的整数数组，你的任务是判断在最多改变 1 个元素的情况下，该数组能否变成一个非递减数列。

我们是这样定义一个非递减数列的： 对于数组中所有的 i (1 <= i < n)，满足 array[i] <= array[i + 1]。

来源：力扣（LeetCode）

链接：https://leetcode-cn.com/problems/non-decreasing-array

著作权归领扣网络所有。商业转载请联系官方授权，非商业转载请注明出处。

我的基本思路是：

设定一个记录调整次数的值，记录调整次数flag，然后循环依次调整数组将它调整为满足题目要求（ i (1 <= i < n)，array[i] <= array[i + 1]）的数组，最后判断调整次数是否大于一次 如果大于一次那么就返回false，否则就返回true。

具体怎么调整：

当数组元素小于三个的时候，无论怎么给都直接返回true。

当元素大于三个假设第i个元素大于第i+1(i>=1) 那么此时有两种调整方法 将i调整到小于i+1 或者将i+1调整到大于i，调整那个才的对的呢？就看i旁边的两个元素，当i+1 > i-1的时候此时只需要将i调整到i-1的值 就满足要求，当i+1 < i-1 的时候 此时只能将i+1调整为等于i的值 因为此时若调整i那么就只能让i<i+1,但是i+1 <i-1 所有调整i会让之前的序列破坏所有只能调整i+1，还有一种情况是i两边的元素是相等的，此时其实满足第一种情况，只需要调整i本身就行。

因为上述方法是i从第二个元素开始的，并且必要条件是之前的序列是满足要求的所以要提前处理前两个元素的序列，当第一个数大于第二个数的时候，将第一个数调整为第二个数的值，此时需要将调整次数flag++ 否则什么都不做，然后i从第二个元素开始按上述方法依次进行每一次调整都将flag++，直到i=n-1的时候结束，n为数组大小。

代码：

bool checkPossibility(vector<int>& nums) {

if(nums.size() < 3){

return true;

}

int flag = 0;

if(nums[0]>nums[1]){

nums[0] = nums[1];

flag++;

}

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

if(nums[i] > nums[i+1] && nums[i-1] < nums[i+1]){

nums[i] = nums[i-1];

flag++;

}

if(nums[i] > nums[i+1] && nums[i-1] > nums[i+1]){

nums[i+1] = nums[i];

flag++;

}

if(nums[i] > nums[i+1] && nums[i-1] == nums[i+1]){

nums[i] = nums[i-1];

flag++;

}

}

if(flag > 1){

return false;

}else{

return true;

}

}

这个算法执行325个用例使用了24ms 但是最快的方法是 8ms所以这个算法并不是最优的解法。

为什么会浪费这么多时间呢？

本题目的要求其实是判断，是否可以修改一个元素就让其成为非递减数组，所以，只需要判断能否成为就行，不需要对数组本身调整；而浪费的时间就是调整数组的时候浪费的。

下面是8ms的代码：

**bool** **checkPossibility**(vector<**int**>& nums) {

**int** k = 0;

**if**(nums.size() < 3){

**return** true;

}

**for**(**int** i = 0; i < nums.size() - 1; i++){

**if**(nums[i] > nums[i + 1]){

k++;

**if**(k > 1){

**return** false;

} **if**(i > 0 && nums[i + 1] < nums[i - 1] && i + 2 < nums.size() && nums[i + 2] < nums[i]){

**return** false;

}

}

}

**return** true;

}

上面代码就是只判断是否可以调整，但是没有真正的调整数组， 第一个if是判断只要有一个i>i+1就需要进行一次调整，然后如果k已经大于1直接返回false

但是在这个情况中有一种特殊情况是即使k=1 也不能进行一次调整就可以让数列非递减

如图：

依次为i-1,I,i+1,i+2; 如上图 不管是调整i还是i+1都会产生第二个不符合的序列所以排除这种特殊情况就可以判断是否可以调整为非递归数组。

学到的东西：

**只要求判断可能性的不必要去对数组做真正的改变**

**因为某些问题特殊形式很少**，如上题目中只有这一种特殊情况那么就将特殊情况排除掉单独处理然后其它的整体处理。

2. 给定两个由小写字母构成的字符串 A 和 B ，只要我们可以通过交换 A 中的两个字母得到与 B 相等的结果，就返回 true ；否则返回 false 。

我的思路：

如果A是空字符串，或者是一个字符那么直接返回false；

如果A字符串的长度不等于B字符串的长度，直接返回false

当A.length = B.length 时 用A中每一个字符的ASCLL码减去B中对应的ASCLL码，结果存入一个等长的数组中，那么数组中的结果有这几种情况：

1. 全为0 那么说明AB相同下标的字符是相同字符，此时只有A中存在相同的两个元素时才可以返回True否则返回False
2. 有两个不为0,，且两个不为0的值为相反数而且对应的下标有 A[i] = B[j] B[i] = A[j] 那么就返回true 否则返回False
3. 大于两个不为0的，返回False
4. 只有一个不为0的，返回False

下面是按我的思路写出的算法：

bool buddyStrings(string A, string B) {

if(A == "" || A.length()==1){

return false;

}

int C[A.length()];

int nozero=0;

int sum = 0;

int index1,index2,count=0;

if(A.length() == B.length()){

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

C[i] = A[i] - B[i];

}

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

sum = sum + C[j];

if(C[j] != 0){

nozero++;

if(nozero == 1){

index1 = j;

}

if(nozero == 2){

index2 = j;

}

if(nozero > 2){

return false;

}

}

}

if(nozero == 2 && sum == 0 && A[index1] == B[index2] && A[index2] == B[index1]){

return true;

}

if(nozero == 0){

for(int i = 0;i<A.length()-1;i++){

for(int j=A.length()-1;j>= i+1;j--){

if(A[i] == A[j]){

return true;

}

}

}

return false;

}

return false;

}

return false;

}

算法缺点：空间复杂度为O(n);时间复杂度为O(n^2)需要双层循环来判断A中是否有相同的元素

下面是最优解法：

**bool** **buddyStrings**(string A, string B) {

**if** (A.size() != B.size() || A.size() < 2) **return** false;

**int** help[26] = {0};

**bool** same = false;

**int** index = 0;

vector<**int**> diff;

**while** (index < A.size()) {

**if** (A[index] != B[index]) {

diff.push\_back(index);

**if** (diff.size() > 2) **return** false;

}

**if** (!same && ++help[A[index]-'a'] > 1) same = true;

++index;

}

**if** (diff.size() == 0 && same == true) **return** true;

**if** (diff.size() == 2 && A[diff[0]] == B[diff[1]] && A[diff[1]] == B[diff[0]]) **return** true;

**return** false;

}

思路分析：用一个动态数组diff来存不相同字符的下标，并且把下标存进去

用help数组来存取二十六个字母在A中出现的次数same标记A中是否有重复元素初始值为false

只要help中出现一个大于2的值说明某个字母已经出现了两次 所以设置为true

如果diff中的下标数量大于2 返回false

如果diff中的下标数正好是2 而且 diff[0]] == B[diff[1]] && A[diff[1]] == B[diff[0] 那么返回true

如果diff中下标数为0 而且same的值为true 那么返回true 否则返回false

优点：可以不用双层循环判断A中是否有重复的元素

可以不用算ASCLL值，值用下标就可以实现

学到的东西：

**对应知道元素范围的数组，判断其中有没有重复的元素时，只需要一个和范围相同大的辅助数组如上题目中的help（二十六个字母），初始值都是0，扫描数组，每出现一个值，将辅助数组中对应的元素的数量加1 与标记量！same 想与 如果大于1 则将same标记为true说明有重复元素，这个算法时间复杂度为O(n) 时间复杂度为O(1)而双层循环的时间复杂度为O(n^2);对以后的数组判断相同元素的都可以用此方法**

**3.** **冬季已经来临。 你的任务是设计一个有固定加热半径的供暖器向所有房屋供暖。**

**现在，给出位于一条水平线上的房屋和供暖器的位置，找到可以覆盖所有房屋的最小加热半径。**

**所以，你的输入将会是房屋和供暖器的位置。你将输出供暖器的最小加热半径。**

**说明:**

**给出的房屋和供暖器的数目是非负数且不会超过 25000。**

**给出的房屋和供暖器的位置均是非负数且不会超过10^9。**

**只要房屋位于供暖器的半径内(包括在边缘上)，它就可以得到供暖。**

**所有供暖器都遵循你的半径标准，加热的半径也一样。**

**来源：力扣（LeetCode）**

**链接：https://leetcode-cn.com/problems/heaters**

**著作权归领扣网络所有。商业转载请联系官方授权，非商业转载请注明出处。**

**我的想法：将供暖气数组从小到大排序，然后遍历房屋数组，每一个房屋数组都对其用二分查找找到他相邻的两个供暖气的位置，如果位置在最左边供暖气的左边，那么这个房屋的供暖半径就是最左边供暖气的值与房屋位置的差，如果在供暖气数组最右边的右边，那么这个房屋供暖气的供暖半径就是房屋位置与最右边供暖气位置的差值，除此之外，落在中间的通过二分查找都可以找到其左右相邻的最近供暖气，分别与房屋算差值，取最小的一个值作为本房屋的供暖半径，并且存入一个数组中，当循环结束后从存储供暖半径的数组中选取最大值就是要求的最小供暖半径**

**代码：**

void merge(vector<int> &R,int low,int mid,int high){

int n1 = mid-low+1;

int n2 = high-mid;

int L1[n1];

int L2[n2];

int k,i,j;

for( i=0;i<n1;i++){

L1[i] = R[low+i];

}

for( j=0;j<n2;j++){

L2[j] = R[mid+1+j];

}

i = j = 0;

k = low;

while(i<n1 && j <n2){

if(L1[i] < L2[j]){

R[k++] = L1[i++];

}else{

R[k++] = L2[j++];

}

}

while(i < n1){

R[k++] = L1[i++];

}

while(j < n2){

R[k++] = L2[j++];

}

}

void mergeSort(vector<int> &A,int low,int high){

int mid;

if(low < high){

mid = (low+high)/2;

mergeSort(A,low,mid);

mergeSort(A,mid+1,high);

merge(A,low,mid,high);

}

}

int findRadius(vector<int>& houses, vector<int>& heaters) {

mergeSort(heaters,0,heaters.size()-1);

vector<int> road;

int low,high;

int r,r1,r2,max,mid;

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

low = 0;

high = heaters.size()-1;

while(low <= high){

mid = (low+high)/2;

if(houses[i] > heaters[mid]){

low = mid + 1;

}else{

high = mid -1;

}

}

if(high+1 == 0){

r = heaters[0] - houses[i];

road.push\_back(r);

}

if(high+1 > heaters.size()-1){

r = houses[i] - heaters[heaters.size()-1];

road.push\_back(r);

}

if(high+1>0 && high+1 <=heaters.size()-1){

r1 = heaters[high+1] - houses[i];

r2 = houses[i] - heaters[high];

if(r1 < r2){

road.push\_back(r1);

}else{

road.push\_back(r2);

}

}

}

max = road[0];

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

if(max < road[i]){

max = road[i];

}

}

return max;

}

执行速度80ms

算法缺点：需要用一个额外的数组去存得到的距离，在找相邻的两个供暖气位置的时候全部的用二分查找法比较繁琐。

优点不用对house数组排序

最优算法代码：

**int** **findRadius**(vector<**int**>& houses, vector<**int**>& heaters) {

**const** **int** h1 = houses.size(), h2 = heaters.size();

**if**(!is\_sorted(houses.begin(),houses.end()))sort(houses.begin(),houses.end());

**if**(!is\_sorted(heaters.begin(),heaters.end()))sort(heaters.begin(),heaters.end());

heaters.push\_back(2000000000);

**int** r = 0,i = 0;

**if**(houses[0]<heaters[0]){

r = heaters[0] - houses[0];

**if**(houses[h1-1]<=heaters[0])**return** r;

**while**(houses[++i]<heaters[0]);

}

**for**(**int R** j=0;i<h1 && j<h2; ++i){

**while**(houses[i]<heaters[j]||houses[i]>heaters[j+1])++j;

r = max(r,min(houses[i]-heaters[j],heaters[j+1]-houses[i]));

}

**return** r;

算法分析：先将houses数组和heaters数组排序，然后将heaters后面加入一个很大的值，先判断houses数组和heaters数组的第一个元素的大小，如果house数组的第一个元素小于heat数组的第一个元素那么说明供暖气是从第一个房屋位置往后的某个位置开始的，此时计算第一个房屋位置与第一个供暖气的距离，当房屋位置的最后一个位置比供暖气的第一个值还小的话 说明整个房屋位置都在第一个供暖气的左边说明最小距离就是第一个房屋位置和第一个房屋位置的距离 否则的话，依次向后寻找第一个供暖气在数组house中出现的位置，用i来记录下标 直到在house数组中找到一个比第一个供暖气大的值，此时

I之前的所有房屋最小距离就是第一个房屋位置与第一个供暖气的距离，i后面的房屋可能存在于某两个供暖气之间，此时需要计算到两个供暖气之间的距离并且取最小的，

如何得到两个相邻的供暖气呢？ 当i的值大于heat[j] 小于heat[j+1]的时候说明是落在j和j+1的中间，那么如果house[i]小于heat[j]或者大于heat[j+1]此时都是不合适的j 让j++直到出现合适的j，此时计算i到j 和i到j+1的值取最小的与之前的第一个第一个房屋到第一个供暖气之间的距离r一起取最大的值得到的值更新r，然后让i++即计算下一个房屋，直到i结束后此时的r就是需要的最小供暖半径然后返回r。

优点：可以用一个r就可以存储最后的结果

在找相邻的两个供暖气的时候只需要顺序的扫描下去就行，不用二分查找，坏处是必须对house也进行一次排序

学到的东西：在一个有序的序列中，要找到某个值相邻的两个值的时候可以顺序的扫描这个数组，判断是否有一个下标满足，i>j且i<j+1 不满足就让j++直到找到为止。

4. 假设你有一个很长的花坛，一部分地块种植了花，另一部分却没有。可是，花卉不能种植在相邻的地块上，它们会争夺水源，两者都会死去。

给定一个花坛（表示为一个数组包含0和1，其中0表示没种植花，1表示种植了花），和一个数 n 。能否在不打破种植规则的情况下种入 n 朵花？能则返回True，不能则返回False。

来源：力扣（LeetCode）

链接：https://leetcode-cn.com/problems/can-place-flowers

著作权归领扣网络所有。商业转载请联系官方授权，非商业转载请注明出处。

我的想法：遍历数组，如果遇到的第i个数是0 那么判断他的下一个位置i+1是否是0 如果是0那么在i位置就是可以使用的位置 让sum++,如果i+1是1说明i+1 i+2 位置都是不可用的那么直接让I = i+3开始继续判断，如果i个数是1那么说明I 和i+1 都是不可用的所以直接让I = i+2 开始继续判断。如果i恰好在最后一个位置上等于0那么此时也让sum++ 循环结束后判断sum>= n则返回true否则返回flase;

代码：

bool canPlaceFlowers(vector<int>& flowerbed, int n) {

int h = flowerbed.size();

int sum=0;

int i=0;

while(i<=h-1){

if(flowerbed[i] == 0){

if(i+1 <= h-1 && flowerbed[i+1] == 0){

sum++;

i=i+2;

continue;

}

if(i+1<=h-1 && flowerbed[i+1] == 1){

i = i+3;

continue;

}

if(i+1 >h-1){

sum++;

i++;

continue;

}

}

if(flowerbed[i]== 1){

i = i+2;

}

}

if(sum >=n){

return true;

}else{

return false;

}

}

缺点：每次扫描一个元素要进行至少两次，至多三次的判断，优点是没有改变原来数组的值直接判断出结果时间复杂度为O(n);

最优解的解法：

假设有000三个0那么能放几个位置呢？可以是两个也可以是两个，要看他们左右的值，如果是10001那么就只能放一个如果是00000则可以放两个，那么我们可以把所有的序列都处理成10001这种形式这样用（k-1）/2 的规律就可以计算出可用的位置 k为两个1之间连续的0的个数。

怎么调整呢？我们可以这样认为---在两个边界上都是0 比如 [0,1,0,0,0,1,0,0],我们可以看成[1,0,0,1,0,0,0,1,0,0,0,1]这样来处理，所有的0都是在两个1之间就可以用上述公式进行计算

代码：bool canPlaceFlowers(vector<int>& flowerbed, int n) {

int k = 1;//开头的左边算作一个0所有这里不是0而是1

int sum = 0;

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

if(flowerbed[i] == 0)k++;

else{

sum += (k-1)/2;

k = 0;

}

}

K++ //循环结束以后将结尾后面看做一个0所以这里要加1；

Sum += (k-1)/2;

If(sum >=n){return true;}else{return false;}

}

算法优点：找出问题的一种规律找出数学公式直接算出需要的结果，最快的找到结果

学到的东西：当遇到一些规律性很强的问题的时候可以尝试找出他的数学规律然后用公式直接去计算结果可以大大提高算法的效率！！！

5. 机器人在一个无限大小的网格上行走，从点 (0, 0) 处开始出发，面向北方。该机器人可以接收以下三种类型的命令：

-2：向左转 90 度

-1：向右转 90 度

1 <= x <= 9：向前移动 x 个单位长度

在网格上有一些格子被视为障碍物。

第 i 个障碍物位于网格点 (obstacles[i][0], obstacles[i][1])

如果机器人试图走到障碍物上方，那么它将停留在障碍物的前一个网格方块上，但仍然可以继续该路线的其余部分。

返回从原点到机器人的最大欧式距离的平方。

来源：力扣（LeetCode）

链接：https://leetcode-cn.com/problems/walking-robot-simulation

著作权归领扣网络所有。商业转载请联系官方授权，非商业转载请注明出处。

笨逼代码（第一次写出来的代码，运行了1000+ms）：

int robotSim(vector<int>& commands, vector<vector<int>>& obstacles) {

int v[2]; //x轴，y轴，正向还是反向,每次接到一个命令的时候需要看当前V的状态来改变 result中x,y中的值

int result[2]; //机器人当前坐标;

int j;

int max = 0;

result[0]=0;//起始x的值为0

result[1]=0;//起始的y为0

v[0] = 0; //等于0表示机器人面向是y轴

v[1] = 1; //等于1表示机器人向Y轴正方向走

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

if(commands[i] == -1){

if(v[0] == 0 && v[1] == 1){

v[0] = 1;

v[1] = 1;

}else if(v[0] == 0 && v[1] == -1){

v[0] = 1;

v[1] = -1;

}else if(v[0] == 1 && v[1] == 1){

v[0] = 0;

v[1] = -1;

}else if(v[0] == 1 && v[1] == -1){

v[0] = 0;

v[1] = 1;

}

continue;

}

if(commands[i] == -2){

if(v[0] == 0 && v[1] == 1){

v[0] = 1;

v[1] = -1;

}else if(v[0] == 0 && v[1] == -1){

v[0] = 1;

v[1] = 1;

}else if(v[0] == 1 && v[1] == 1){

v[0] = 0;

v[1] = 1;

}else if(v[0] == 1 && v[1] == -1){

v[0] = 0;

v[1] = -1;

}

continue;

}

if(commands[i] >=1 && commands[i]<=9){

if(v[0] == 0 && v[1] == 1){

for( j=0;j<obstacles.size();j++){

if(obstacles[j][0] == result[0] && obstacles[j][1]>result[1] && obstacles[j][1]<= result[1]+v[1]\*commands[i]){

break;

}

}

if(j < obstacles.size()){

result[1] = obstacles[j][1]-1;

}else{

result[1] = result[1]+v[1]\*commands[i];

}

}else if(v[0] == 0 && v[1] == -1){

for( j=0;j<obstacles.size();j++){

if(obstacles[j][0] == result[0] && obstacles[j][1]<result[1] && obstacles[j][1]>= result[1]+v[1]\*commands[i]){

break;

}

}

if(j < obstacles.size()){

result[1] = obstacles[j][1]+1;

}else{

result[1] = result[1]+v[1]\*commands[i];

}

}else if(v[0] == 1 && v[1] == 1){

for( j=0;j<obstacles.size();j++){

if(obstacles[j][1] == result[1] && obstacles[j][0]>result[0] && obstacles[j][0]<= result[0]+v[1]\*commands[i]){

break;

}

}

if(j < obstacles.size()){

result[0] = obstacles[j][0]-1;

}else{

result[0] = result[0]+v[1]\*commands[i];

}

}else if(v[0] == 1 && v[1] == -1){

for( j=0;j<obstacles.size();j++){

if(obstacles[j][1] == result[1] && obstacles[j][0]<result[0] && obstacles[j][0]>= result[0]+v[1]\*commands[i]){

break;

}

}

if(j < obstacles.size()){

result[0] = obstacles[j][0]+1;

}else{

result[0] = result[0]+v[1]\*commands[i];

}

}

if(result[1]\*result[1]+result[0]\*result[0] > max){

max = result[1]\*result[1]+result[0]\*result[0];

}

}

}

return max;

}

稍加优化的代码：

int robotSim(vector<int>& commands, vector<vector<int>>& obstacles) {

int dir = 0;

int dx[4] = {0,1,0,-1};

int dy[4] = {1,0,-1,0};

int x = 0;

int y = 0;

int ans = 0;

set<pair<int,int>> obs; //set用来检测是否被阻挡，pair是将两个数打包成一个变量存入set中

for(vector<int> i:obstacles){

if(i.size()>0){

obs.insert(pair(i[0],i[1]));

}

}

for(int cmd:commands){

if(cmd == -1){

dir = (dir+1)%4;

}else if(cmd == -2){

dir = (dir+3)%4;

}else{

for(int k=0;k<cmd;k++){

int nx = x + dx[dir];

int ny = y + dy[dir];

if(obs.find(pair(nx,ny)) == obs.end()){

x = nx;

y = ny;

ans = max(ans,(x\*x+y\*y));

}

}

}

}

return ans;

}

大佬无敌的代码：

struct Int2 { int x, y; };

inline bool operator==(Int2 a, Int2 b) { return a.x == b.x && a.y == b.y; }

inline Int2 operator+(Int2 a, Int2 b) { return {a.x + b.x, a.y + b.y}; }

inline bool isFree(Int2 p, const vector<Int2>& obstacles) {

for (Int2 o : obstacles) {

if (p == o) return false;

} return true;

} inline Int2 turnLeft(Int2 h) {

return {-h.y, h.x};

} inline Int2 turnRight(Int2 h) {

return {h.y, -h.x};

}

class Solution { public: int robotSim(vector<int>& commands, vector<vector<int>>& obstacles) { const int division = 50;

vector<vector<Int2>> obs(60001 / division);

for (const vector<int>& o : obstacles) obs[(o[0] + 30000) / division].push\_back(Int2{o[0], o[1]});

Int2 p = {0, 0};

Int2 h = {0, 1};

int maxDistance = 0;

for (int command : commands) { if (command == -2) h = turnLeft(h);

else if (command == -1)

h = turnRight(h);

else {

for (int step = 1; step <= command; ++step) {

const Int2 n = p + h;

if (isFree(n, obs[(n.x + 30000) / division])) {

p = n;

maxDistance = max(maxDistance, p.x \* p.x + p.y \* p.y);

} else break;

}

}

}

return maxDistance;

}

代码分析：

动态规划尝试：

例子：斐波那契数列

1 1 2 3 5 7 12………

如何求出第N个斐波那契数？

要求出第n个斐波那契数就要求出第n-1 和n-2 个斐波那契数，而每一次求斐波那契数都是同样的方法，我们最暴力的想法就是从f1 f2开始一直求到我们需要的第N个位置的数为止

递归：

int fb(int n){

if(n == 1 || n==2){

return 1;

}

If(n<= 0){

Return 0;

}

Return fb(n-1) +fb(n-2);

}

这个算法的时间复杂度为O(2^n)递归树的时间复杂度

这个方法中算了很多重复的值，接下来是带记忆的递归树

int fb(int n,memor[]){

if(n == 1 || n== 2){

memor[n] = 1;

}

If(memor[n] > 0){

Return memor[n];

}

Mermor[n] = fb(n-1) +fb(n-2);

Return memor[n];

}

这样的话 时间复杂度就变成了O(n)每个位置只需要算一次不需要重复计算

动态规划：

我们要计算fn 则只需要计算fn-1 fn-2即可

就有fn = fn-1 +fn-2;

有了这个式子就可以将递归转化为普通的循环就可以解决

比如 我们知道了 f2=1 f1 =1;

那么f3 = f1 + f2;

Int fb(int n){

Dp[n+1];

Dp[1] = 1;

Dp[2] = 1;

For(int I = 3;i<=n;i++){

Dp[i] = dp[i-1]+dp[i-2];

}

Return dp[n];

}

这样又可以避免使用递归栈 减少了运行时间时间复杂度为O(n)动态规划与带记忆的递归其实是差不多的只是一个是从上到下 一个是从下到上

2.给定一个整数数组 nums ，找到一个具有最大和的连续子数组（子数组最少包含一个元素），返回其最大和。

暴力解法遍历每一个连续子序列然后将最大值记录最后返回最大值

如果找到所有的子序列？

如 [a,b,c,d]子序列可以有

a a,b a,b,c a,b,c,d

b b,c b,c,d

c c,d

d

可以发现规律就是都是以某个元素为开头 然后与后面的元素组成连续子序列

根据这一规律可以有暴力解法的代码：

int maxSubArray(vector<int>& nums){

int max=0;

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

int sum=0;

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

sum += nums[j];

if(sum > max){

max = sum;

}

}

}

return max;

}

还有一种遍历方法就是：

[a,b,c,d]有

a

a,b b

a,b,c b,c c

a,b,c,d b,c,d c,d d

发现规律就是以某一个字母为结尾 形成连续的子序列，这种方法有利于找出递推关系，使用动态规划就需要使用这种遍历的思想

如果要算出以d结尾的连续子序列的最大和 就要算出 a,b,c,d b,c,d c,d d这几个子序列的和然后作比较算出最大的和，而 a,b,c,d 可以由a,b,c的和加d来算出 b,c,d的和可以由b,c的和加d算出，c,d的和可以由c+d来算出，那么d的和就是d本身，那么abcd bcd cd d中怎样的值就是最大的值呢？ 如果abc是最大的那么abc加d就比bc+d和c+d都大其余的同理 所有要得到d结尾的连续子序列的最大值 只需要知道以c结尾的连续子序列的最大值，然后加d得到的值再与d取最大值就是以d结尾的子序列的最大值，为什么要与d取最大值呢max{max[c]+d,d}如果以c为结尾的最大值是一个负数，那么负数加一个数只会将一个数变小，那么这个时候，连续子序列中的最大值就只能是d本身啦

这样我们很容易得到一个递推关系

如果有一个大小为n的数组，我们知道第i个位置为止的最大连续子序列的和要求第i+1个位置的连续子序列的和就可以得到 max[i+1] = {max[i]+n[i+1],n[i+1]} 这样一个递推关系

在动态规划中这就是重要的状态转移方程，有了它就可以递推的自下而上的推断出以每一个元素结尾的子序列中的最大值，我们知道的是max[1] = n[0] 就是第一个元素结尾的子序列最大值显然就是它本身，那么以第二个元素结尾子序列中和的最大值就是 max[2] = max{n[0]+n[1],n[1]}

而上述的题目中要求的是这个数组的所有连续子序列的最大和，那么我们就可以每求出一个以某一元素结尾的所有子序列的最大和与之前求出的最大值作比较 如果大于则更新这个最大值，如果不大于则继续计算下一个，最终以最后一个元素结尾的子序列最大和之后，那么max就被更新为最大值 此时返回最大值即可

代码：

int maxSubArray(vector<int>& nums) {

int ans;

int dp[nums.size()+1];

dp[1] = nums[0];

ans= dp[1];

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

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

if(dp[i+1] > ans){

ans = dp[i+1];

}

}

return ans;

}

上述代码就是时间复杂度为O(n) 空间复杂度为O(n)的动态规划法，相比于暴力法的O(n2)已经大大的减少了运算次数

有没有更好的方法呢？

以上面abcd的例子为例，假如sum现在是以c为结尾的子序列中，最大子序列的和，当sum>0的时候，说明d+sum一定是变大的 我们理解为对d结尾的子序列有增强作用反之当sum<0的时候 sum+d一定是小于d的 此时我们直接将以d结尾的子序列的最大值更新为d就行用这个想法来写代码

int maxSubArray(vector<int>& nums){

int ans;

int sum

ans = nums[0];

sum = nums[0];

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

if(sum > 0){

sum += nums[i];

}else{

Sum = nums[i];

}

ans = max(sum,ans);

}

Return ans;

}

这个代码的执行速度其实和动态规划是差不多的，只不过好处是现在空间复杂度变成了O(1)时间复杂度不变

3. 给定一个包含非负整数的 m x n 网格，请找出一条从左上角到右下角的路径，使得路径上的数字总和为最小。

**说明：**每次只能向下或者向右移动一步。

想法：设定一个和grid一样大小的二维数组dp[m][n];如果想得到从左上角到右下角dp[m-1][n-1]的最短路径，那么首先得到从左上角到dp[m-2][n-1]和 dp[m-1][n-2]的最短路径然后选取其中最小的那个加上grid[m-1][n-1]的值就是要求的值，得到了递推关系

dp[i][j] = grid[i][j] + min{dp[i-1][j],dp[i][j-1]};

求出所有的dp最终输出dp[m-1][n-1]就可以了

但是我们发现，这种修改可以直接在原数组上修改，最后输出原数组的最后一个元素就是我们要求的值

代码：

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

int m = grid.size();

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

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

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

if(i==0 && j != 0){

grid[i][j] += grid[i][j-1];

}else if(j== 0 && i != 0){

grid[i][j] += grid[i-1][j];

}else if(i != 0 && j != 0){

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

}

}

}

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

}

代码优点：节省了空间，时间复杂度为O(mn)空间复杂度为O(1)

大佬的代码：



4.

一个机器人位于一个 m x n 网格的左上角 （起始点在下图中标记为“Start” ）。

机器人每次只能向下或者向右移动一步。机器人试图达到网格的右下角（在下图中标记为“Finish”）。

问总共有多少条不同的路径

来源：力扣（LeetCode）

链接：https://leetcode-cn.com/problems/unique-paths

著作权归领扣网络所有。商业转载请联系官方授权，非商业转载请注明出处。

我的想法：由于机器人只能向下走或者向右走，所以不存在绕回去的路径，那么要得到某个位置

Road(i,j)(i和j都不为0) = road(i-1,j)+road(i,j-1) 由于不能有绕回去的路，当i等于0或者j等于0的时候，那么这个地方只有唯一的一条路线于是根据上面的递推关系我们设定一个二维数组dp有如下代码：

int uniquePaths(int m, int n) {

int dp[m][n];

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

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

if(i==0 || j==0){

dp[i][j] = 1;

}

if(i != 0 && j != 0){

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

}

}

}

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

}

此题和上一题有类似的解法，属于经典的动态规划思想时间复杂度和空间复杂度都为O(mn)

当然有一种更好的方法

由于机器人要到底部，右下角所以最终一定是向下的方向走m-1次向右的次数走n-1次那么就是一个排列组合问题

m-1

C

m+n-2 直接用公式就可以算出来

5. 一个机器人位于一个 m x n 网格的左上角 （起始点在下图中标记为“Start” ）。

机器人每次只能向下或者向右移动一步。机器人试图达到网格的右下角（在下图中标记为“Finish”）。

现在考虑网格中有障碍物。那么从左上角到右下角将会有多少条不同的路径？

来源：力扣（LeetCode）

链接：https://leetcode-cn.com/problems/unique-paths-ii

著作权归领扣网络所有。商业转载请联系官方授权，非商业转载请注明出处。

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

int m = obstacleGrid.size();

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

double dp[m][n];

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

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

dp[i][j] = obstacleGrid[i][j];

}

}

int k;

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

if(dp[0][i] != 1){

dp[0][i] = -1;

}else{

while(i<n){

dp[0][i] = 1;

i++;

}

}

}

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

if(dp[i][0] != 1){

dp[i][0] = -1;

}else{

while(i<m){

dp[i][0] = 1;

i++;

}

}

}

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

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

if(dp[i][j] == 0){

if(dp[i-1][j] != 1 || dp[i][j-1] != 1 ){

if(dp[i-1][j] != 1){

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

}

if(dp[i][j-1] != 1){

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

}

}else{

dp[i][j] = 1;

}

}

}

}

k = 0 - dp[m-1][n-1];

if(k<0){

return 0;

}else{

return k;

}

}

整体思路和没有障碍物一样，动态规划，只不过中间要考虑一些判断，比如当i或者j等于0的时候如果遇到一个等于1的值那么它后面的所有的位置都不能去，直接赋值为1如果一个位置它的上方和前方都是1那么这个位置也是一个无法到达的点，直接赋值为1排除上述的几个其余的值都按正常的计算遇到1不加遇到不为1的值累加，知道循环完成，数组最后一个位置的值就是总的路径数量（在我的代码中使用了-1相加最终的值是负数 所以最后返回要用相反数，而且在用int时中间值会超出int的范围，所以用了double）如果原数组可以改为double那么空间复杂度可以为O(1)时间复杂度不变为O(mn)

这个题的动态转移方程为：

Dp(i,j) = Dp(i-1,j) + Dp(I,j-1);

但是有条件： Dp(i-1,j)和Dp(I,j-1)是不为1的值，且Dp(I,j)也不能是1的值

6.

给定一个字符串 s，找到 s 中最长的回文子串。你可以假设 s 的最大长度为 1000。

我的想法只能想到暴力法就是遍历每一个子串判断是不是回文串并且将最长的回文串保存然后输出时间复杂度为O(n3)很高通不过测试用例

看了解答后用dp优化，优化的思想如下

设有 p(I,j) = true 则s[I,j]为回文串

P(I,j) = false 则s[l,j]不是回文串

假设我们要算p(I,j)是否是回文的

那么 p(I,j) = p(i+1,j-1) && (s[i] == s[j]) 意思就是 如果 aba是回文的 那么 xabax x=x的时候就是回文的，这样要求出长度为n的回文串，首先就要求出n-1长度的回文串，然后再判断两端是否相等，如果相等那么就将长度为n的子串的p(I,j)赋值为true这样依次下去，直到遍历完所有长度以后，记录过程中最大的那个回文子串的长度，输出就行，但是长度为1和长度为2的回文串不能用上述公式 因为这样就会出现i<j的情况是不可能出现的 所以长度为1和长度为2的回文串的p(I,j)需要初始化

代码：

/\*p(i,j) = true s[i,j] 为回文子串

p(i,j) = false s[i,j] 不是回文子串

p(i,j) = (p(i+1,j-1) && s[i]==s[j]) \*/

string longestPalindrome(string s) {

int lenght = s.size();

if(lenght == 0){

return "";

}

int maxlen = 0;

int i;

bool P[lenght][lenght];

for(int len=1;len<=lenght;len++){

for(int start=0;start<lenght;start++){

int end = start+len-1;

if(end>=lenght){

break;

}

P[start][end] = (len == 1 || len == 2 || P[start+1][end-1]) && s[start] == s[end];

if(P[start][end] && len > maxlen){

i = start;

maxlen = len;

}

}

}

return s.substr(i,maxlen);

}

此时的空间复杂度和时间复杂度都是O(n2)

对于回文串，由于它是两边互相对称的，所有在判断的时候可以选取一个中点然后向两边扩散，记录下来长度，假如串的长度为n那么中心点就有2n-1个 因为当回文串的长度是奇数的时候中点是落在字符上的，但是当回文串是偶数的时候，如aabbaa那么中点就会落在bb中间的点，所以要遍历所有的回文串就必须考虑所有中点 那么字符有n个字符的间隙有n-1个一共就是n+n-1 = 2n-1个中点这样时间复杂度也是O(n2)

最长回文子串还有一个最经典的方法就是马拉车方法，这是一个O(n)的方法求出一个串中最长的回文子串 思想如下

第一步：在串的首尾分别添加上“^”和”$”这样的两个不可能在串中出现的字符，在从每一个字符之间插入一个”#”这个字符，经过这样的变化以后，整个串的长度一定是奇数

如： ^#a#b#a#a#b#a#$

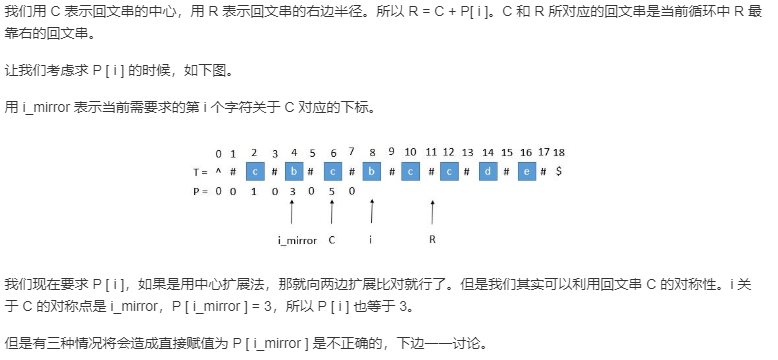
假如原来串的长度为n那么需要插入n-1+4个字符，此时的字符串长度为n+n-1+4= 2n+3一定是一个奇数

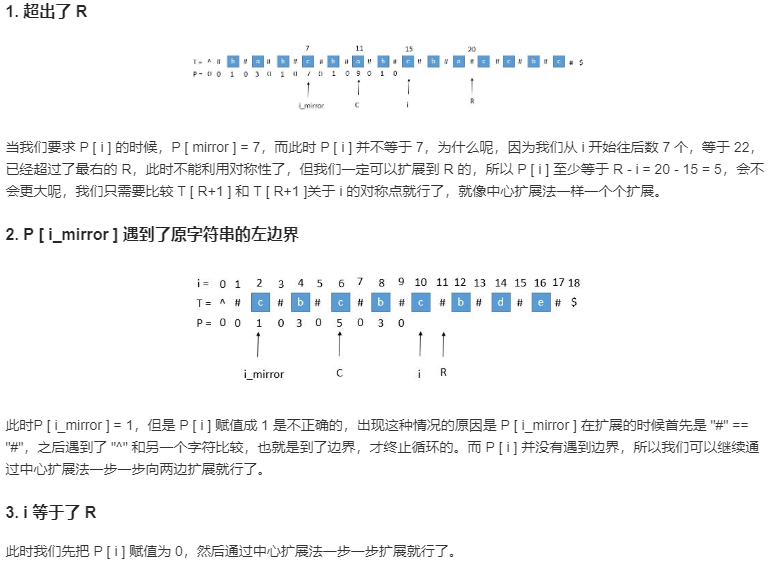
第二步：设置一个数组p大小为变化后的字符串的长度，p用来储存每一个字符为中心的时候他的最大扩展数，如步骤一中的字符串中第1个b的下标是4，以它为中心扩散的最大长度为3那么p[4]=3; 而且有一个关系，p[4]正好是去掉插入的字符后，以b为中心的回文串的长度 aba

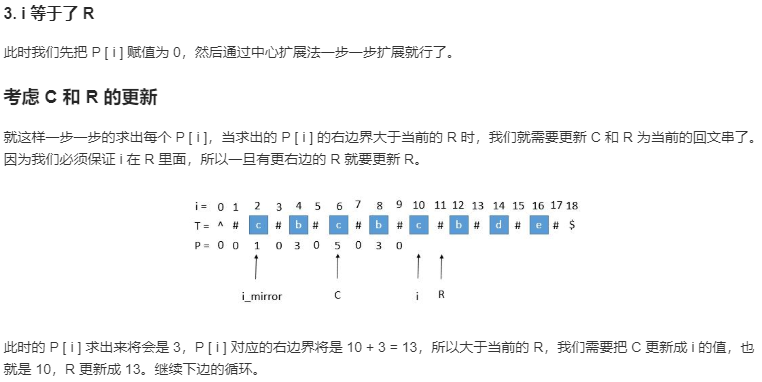
如何求出原来数组中的start就是回文串开始的值呢？

用变化后的数组下标i-p[i]然后再除以2就是原数组的start位置，输出的时候只需要输出s.substr(start,p[i])就可以了

第三步：求出所有的p[i]







我自己按照这个想法写了以后发现执行速度基本从120+到了100以内最快的一次是60ms然而我看到有人用这个方法写最高执行时间是8ms 所以我不知道为啥。。于是直接上大佬代码吧：string longestPalindrome(string s) {

int len=s.length();

int i;

if(len<1)

return "";

string s1;

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

s1+="#";

s1+=s[i];

}

s1+="#";

len=s1.length();

int Maxright=0;

int pos=0;

int MaxRL=0;

int Maxpos=0;

int \*RL=new int[len];

memset(RL, 0, len \* sizeof(int));//将数组全部赋值为0

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

if(i<Maxright){

RL[i]=min(RL[2\*pos-i],Maxright-i);

}

else

RL[i]=1;

while(i+RL[i]<len && i-RL[i]>=0 && s1[i+RL[i]]==s1[i-RL[i]]){

RL[i]=RL[i]+1;

}

if (RL[i] + i - 1 > Maxright)

{

Maxright = RL[i] + i - 1;

pos = i;

}

if (MaxRL <= RL[i])

{

MaxRL = RL[i];

Maxpos = i;

}

}

return s.substr((Maxpos - MaxRL + 1) / 2, MaxRL - 1);

}

7. 给定一个整数 n，求以 1 ... n 为节点组成的二叉搜索树有多少种？

动态规划

设G(n) 为长度为n的序列组成的二叉搜索树的个数

F(I,n)为当以第i个元素为根节点的时候不同的二叉搜索树的个数可以得到

F(i) = G(i-1)\*G(n-i);

G(n) = F(1)+F(2)+……F(n);

由上面的两个式子可以推出

G(n) = G(0)\*G(n-1)+G(1)\*G(n-2)+G(2)\*G(n-2)……G(n-1)\*G(0);

可以观察到G(n)与G(n-1) G(n-2)……G(0)有关

那么要求出G(n)就要求出G(n-1);

符合动态规划的思想，上面G(n)就是状态转移方程

代码：

int numTrees(int n) {

int G[n+1]; //这里如果用普通的数组就需要在下面加之前先将G[i]变为0发现速度会变慢，用vector会更快

//vector<int>G(n+1,0);

//memset(G,0,sizeof(int)\*(n+1));

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

G[i] = 0;

}

G[0] = 1;

G[1] = 1;

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

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

G[i] += G[j-1]\*G[i-j];

}

}

return G[n];

}

初始化是长度为0和长度为1的序列它的搜索树都是1，空树和只有一个根节点

第一层循环表示从小到大遍历每一个长度的序列，第二层循环表示要求出G(n)就要求出F(1)……F(n)所有j的范围是从1到i（表示当前序列的长度）最终求出G(n)就是我们要的答案

一定要注意G[i]的初始化，上面的三种初始化的方法都可以，但是如果从第一层循环中加入一个G[i]=0那么经过测试，执行时间会变成原来的二倍！！！！所以初始化数组最好是提前进行这样可以提高速度

8. 给定一个整数 n，生成所有由 1 ... n 为节点所组成的**二叉搜索树**。

思路：递归

我们要求出1到n的所有可能的搜索树，就求出以每一个节点为根的搜索树，比如要求出以i为根节点的所有二叉搜索树，那么它的左边的数都是它的左子树，右边的数都是它的右子树，而以i为根节点的搜索树就是左子树的所有情况乘以右子树的所有情况，而左右子树有可以用上述方法去计算，把每一种情况保存下来然后将左边的子树连接到i的左子树上，右边的子树连接到i的右子树上然后将i存进ans依次计算所有1到n的每个节点，最后输出ans即可

代码：

vector<TreeNode\*>getAns(int start,int end){

vector<TreeNode\*> ans;

if(start > end){

ans.push\_back(nullptr);

return ans;

}

for(int i=start;i<=end;i++){

vector<TreeNode\*> L = getAns(start,i-1);//从start ---- i-1中以每一个元素为根节点，并且求出以这个为根的所有可能的搜索树

vector<TreeNode\*> R = getAns(i+1,end);//从i+1 到end每一个元素为根节点，求出所有可能的搜索树

for(auto l : L){

for(auto r : R){

TreeNode\* node = new TreeNode(i); //以i为根节点

node->left = l;//i的左子树中的某一个

node->right = r;//所有右子树中的某一个

ans.push\_back(node);//将这种可能加入到数组中去 直到遍历所有的可能后ans中就存了以i为根的所有可能的搜索树 然后循环下一i就是最外层循环等最外层循环结束以后ans就存了所有的答案，然后将这个ans返回就行

}

}

}

return ans;

}

vector<TreeNode\*> generateTrees(int n) {

if(n == 0){

return {};

}

return getAns(1,n);

}

思考如何将递归的方法转换为动态规划方法

长度为0的搜索树个数为1个 dp[0] = 1 即[] 的值为1

长度为1的搜索树个数为1个 dp[1] = 1;[1] 的值为1

长度为2的搜索树个数即[1,2] 以1为根节点的个数是1将序列分成了长度为0和长度为1的两个字序列，长度为0的作为1的左子树，长度为1的作为右子树，左右子树的可能性相互组合就是1为根的所有搜索树的情况，以2为根的原理一样dp[n] = dp[0]\*dp[n-1]+dp[1]\*dp[n-2]…这样我们用长度为1的推出长度为2的然后用长度为2的算出所有长度为3的直到n最后输出所有的dp[n]中的节点就是要求的答案。具体代码还没完全明白。。。

9. 给你一个字符串 s 和一个字符规律 p，请你来实现一个支持 '.' 和 '\*' 的正则表达式匹配。

'.' 匹配任意单个字符

'\*' 匹配零个或多个前面的那一个元素

递归：

想法：

由于每次都只比较一个字符，如果相等就继续向下比较如果不相等直接返回false，那么每次递归只处理一个字符即可，当处理时遇到p字符串为空的时候，如果s串也为空那么说明匹配已经完成了直接返回true 如果遇到s的第一个字符与P的第一个字符相等，那么说明本次匹配的结果是true，或者s的第一个字符遇到P中的“.”也相当于相等也是匹配成功，将本次递归的结果存入first\_match中，然后去掉当前比较完的字符其余的进行下一次递归，由于p中有\*存在，\*意味着可以重复0次或者n次它前面的字符，所以每次比较的时候还需要判断本次的字符后面有没有\*，因为假如说本次匹配失败，first\_match应该是false，但是如果后面有\*存在那么说明可以将本次的p中做比较的字符去掉即\*重复0次，所以是否匹配就要看下一次递归的first\_match与本次无关，如果说本次匹配成功的后面也是有\*那么需要将\*保留在本次递归中相当于重复了一次，再进行下次比较，上述的两种情况都可以视为s与p在这次递归中是成功的就有如下代码

代码：

bool isMatch(string s,string p){

if(p == ""){

return s == "";

}

bool first\_match = (s[0] == p[0] || p[0] == '.'); // 本次匹配的结果

if(p.size() >= 2 && p[1] == '\*'){ //遇到了\*

if(s.empty()){// 遇到空字符串s

return isMatch(s,p.substr(2)); //如果遇到了p中的\*那么只能将p中的前两个直接重复0次就是取消掉然后再与s进行下一次递归

}else{

return (isMatch(s.substr(1),p) && first\_match) || isMatch(s,p.substr(2)); //如果不是空字符串那么遇到了\*有两种情况，一种是与\*前的元素匹配，那么本次匹配成功，匹配结果是first\_match且进行下一次递归的时候p不变相当于保留了\*，或者匹配不成功那么直接取消掉p的前两个字符，然后进行下一次递归，只要两种情况成功一种那么就视为p与s是匹配成功的

}

}

if(s == ""){

return false; //如果p是长度小于1或者后面不是\*的那么s是空的时候p一定不是空那么说明不可能匹配

}else{

return first\_match && isMatch(s.substr(1),p.substr(1));//如果不是空那么也没遇到\*就返回本次比较结果与后面的比较结果的和。

}

}

动态规划：没想明白。

想法

代码

字符串类型的题：

1. 编写一个函数来查找字符串数组中的最长公共前缀。

如果不存在公共前缀，返回空字符串 ""

通过横向比较，找出前缀的长度返回字符串

代码：

string longestCommonPrefix(vector<string>& strs) {

if(strs.size() < 1){

return "";

}

/\*

int n = strs.size();

int len = strs[0].size();

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

int j = 0;

while(strs[0][j] == strs[i][j] && j < strs[i].size() && j<len){

j++;

}

if(j<len){

len = j;

}

}

\*/

int i;

int j;

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

char ch = strs[0][j];

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

if(strs[i][j] != ch){

break;

}

}

if(i<strs.size()){

return strs[0].substr(0,j);

}

}

return strs[0];

}

中间注释部分运行时长是横向比较的三倍

2. 给定一个只包括 '('，')'，'{'，'}'，'['，']' 的字符串，判断字符串是否有效。

有效字符串需满足：

左括号必须用相同类型的右括号闭合。

左括号必须以正确的顺序闭合。

注意空字符串可被认为是有效字符串。

来源：力扣（LeetCode）

链接：https://leetcode-cn.com/problems/valid-parentheses

著作权归领扣网络所有。商业转载请联系官方授权，非商业转载请注明出处。

经典的题目，用栈解决

如果是左括号直接入栈，如果遇到右括号，判断栈顶是否是与右括号匹配的左括号，如果匹配则左括号出栈，如果不匹配直接返回false，注意，栈空的时候遇到右括号也是不匹配的情况直接返回false

代码：

bool isValid(string s) {

if(s ==""){

return true;

}

char st[10000];

int top = -1;

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

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

st[++top] = s[i];

}else{

if(top < 0){

return false;

}else{

if(s[i] == ')' && st[top] == '('){

top--;

}else if(s[i] == ']' && st[top] == '['){

top--;

}else if(s[i] == '}' && st[top] =='{'){

top--;

}else{

return false;

}

}

}

}

if(top >= 0){

return false;

}else{

return true;

}

}

3. 给定一个仅包含数字 2-9 的字符串，返回所有它能表示的字母组合。

给出数字到字母的映射如下（与电话按键相同）。注意 1 不对应任何字母。



想法：

动态规划----要求出长度为n的所有字符串的组合，假如已经知道长度为n-1长度的所有字符串组合，那么长度为n的字符的组合就是下一个字符与n-1长度的所有字符组合即

状态转移方程为 dp[i] = dp[i-1](所有字符串) + str[i]

代码：

vector<string> letterCombinations(string digits) {

if(digits.size()<1){

return{};

}

vector<vector<string> > dp;

map<char,string>strmap;

strmap['2'] = "abc";

strmap['3'] = "def";

strmap['4'] = "igh";

strmap['5'] = "jkl";

strmap['6'] = "mno";

strmap['7'] = "pqrs";

strmap['8'] = "tuv";

strmap['9'] = "wxyz";

vector<string>dp1;

for(auto str:strmap[digits[0]]){

string x = "";

x = x+str;

dp1.push\_back(x);

}

dp.push\_back(dp1);

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

vector<string> dp2;

for(auto str1 : dp[i-1]){

for(auto str2 : strmap[digits[i]]){

string str3 ="";

str3 = str1+str2;

dp2.push\_back(str3);

}

}

dp.push\_back(dp2);

}

return dp[digits.size()-1];

}

大佬代码：

vector<vector<string>> words = {

{"!", "@", "#"},

{"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> letterCombinations(string digits) {

vector<string> result;

int len = digits.size();

if (len <= 1) {

if (len == 1) {

int number = digits.front() - '0';

result = words[number - 1];

}

return result;

}

int number = digits.front() - '0';

vector<string> temp = words[number - 1];

vector<string> subTemp = letterCombinations(digits.substr(1)); //这里递归

for (int j = 0; j < temp.size(); j++) {

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

result.push\_back(temp[j] + subTemp[i]);

}

}

return result;

}

4. 罗马数字包含以下七种字符: I， V， X， L，C，D 和 M。

字符 数值

I 1

V 5

X 10

L 50

C 100

D 500

M 1000

例如， 罗马数字 2 写做 II ，即为两个并列的 1。12 写做 XII ，即为 X + II 。 27 写做 XXVII, 即为 XX + V + II 。

通常情况下，罗马数字中小的数字在大的数字的右边。但也存在特例，例如 4 不写做 IIII，而是 IV。数字 1 在数字 5 的左边，所表示的数等于大数 5 减小数 1 得到的数值 4 。同样地，数字 9 表示为 IX。这个特殊的规则只适用于以下六种情况：

I 可以放在 V (5) 和 X (10) 的左边，来表示 4 和 9。

X 可以放在 L (50) 和 C (100) 的左边，来表示 40 和 90。

C 可以放在 D (500) 和 M (1000) 的左边，来表示 400 和 900。

给定一个罗马数字，将其转换成整数。输入确保在 1 到 3999 的范围内。

来源：力扣（LeetCode）

链接：https://leetcode-cn.com/problems/roman-to-integer

著作权归领扣网络所有。商业转载请联系官方授权，非商业转载请注明出处。

每次比较两个字符

如果后面的字符比较大那么结果加后面字符代表的数值减去前面字符所代表的数值，如果后面字符比前面字符小或者等于前面的字符，直接加前面的字符所代表的数字，第一种情况i后移两位，第二种情况后移一位

代码：

int getVal(char ch){

switch(ch){

case'I': return 1;

case'V': return 5;

case'X': return 10;

case'L': return 50;

case'C': return 100;

case'D': return 500;

case'M': return 1000;

default : return 0;

}

}

int romanToInt(string s) {

/\*每次判断两个，如果后面是比自己大的值，那么结果是+一个后面值减去前面值的值，如果后面的值是比自己小的值，那么直接加当前的值扫描字符串最终得到答案\*/

if(s.size() < 1){

return 0;

}

/\*

map<char,int>Nmap;

Nmap['I'] = 1;

Nmap['V'] = 5;

Nmap['X'] = 10;

Nmap['L'] = 50;

Nmap['C'] = 100;

Nmap['D'] = 500;

Nmap['M'] = 1000;\*/

int i = 0;

int ans = 0;

while(s[i] != '\0'){

if(getVal(s[i]) < getVal(s[i+1])){

ans += getVal(s[i+1]) - getVal(s[i]);

i= i+2;

}else{

ans += getVal(s[i]);

i++;

}

}

return ans;

}

代码说明：刚开始用的map发现执行效率是28ms平均然后用了switch发现是8ms最快的时候，不知道原因，按理来说map的效率应该高于switch的啊

补充 测试了0ms的代码用的switch最慢还是会到24ms没跑的0ms应该和测试用例有关平均是12ms

1. 和上一题一致不过是给出数字求罗马数字

我的想法就是存入每一位上所有的可能然后将给出的数字分别拆分成千位+百位+十位+各位，对应的字符串相加得到最后的组合

好一些的想法：贪心算法，每一次都从这个数中减去与它最接近的数，并且把对应的字符串加进答案中两种方法的代码：

string intToRoman(int num) {

/\*

if(num == 0){

return"";

}

map<int,string>strMap;

strMap[1] = "I";

strMap[2] = "II";

strMap[3] = "III";

strMap[4] = "IV";

strMap[5] = "V";

strMap[6] = "VI";

strMap[7] = "VII";

strMap[8] = "VIII";

strMap[9] = "IX";

strMap[10] = "X";

strMap[20] = "XX";

strMap[30] = "XXX";

strMap[40] = "XL";

strMap[50] = "L";

strMap[60] = "LX";

strMap[70] = "LXX";

strMap[80] = "LXXX";

strMap[90] = "XC";

strMap[100] = "C";

strMap[200] = "CC";

strMap[300] = "CCC";

strMap[400] = "CD";

strMap[500] = "D";

strMap[600] = "DC";

strMap[700] = "DCC";

strMap[800] = "DCCC";

strMap[900] = "CM";

strMap[1000] = "M";

strMap[2000] = "MM";

strMap[3000] = "MMM";

strMap[0] = "";

int a,b,c,d;

a = num/1000\*1000;

b = (num-a)/100\*100;

c = (num-a-b)/10\*10;

d = num-a-b-c;

return strMap[a]+strMap[b]+strMap[c]+strMap[d];

\*/

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

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

string ans;

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

while(num>=v[i]){

num -= v[i];

ans += re[i];

}

}

return ans;

}

6. 请你来实现一个 atoi 函数，使其能将字符串转换成整数。

首先，该函数会根据需要丢弃无用的开头空格字符，直到寻找到第一个非空格的字符为止。

当我们寻找到的第一个非空字符为正或者负号时，则将该符号与之后面尽可能多的连续数字组合起来，作为该整数的正负号；假如第一个非空字符是数字，则直接将其与之后连续的数字字符组合起来，形成整数。

该字符串除了有效的整数部分之后也可能会存在多余的字符，这些字符可以被忽略，它们对于函数不应该造成影响。

注意：假如该字符串中的第一个非空格字符不是一个有效整数字符、字符串为空或字符串仅包含空白字符时，则你的函数不需要进行转换。

在任何情况下，若函数不能进行有效的转换时，请返回 0。

说明：

假设我们的环境只能存储 32 位大小的有符号整数，那么其数值范围为 [−231, 231 − 1]。如果数值超过这个范围，请返回 INT\_MAX (231 − 1) 或 INT\_MIN (−231)

来源：力扣（LeetCode）

链接：https://leetcode-cn.com/problems/string-to-integer-atoi

著作权归领扣网络所有。商业转载请联系官方授权，非商业转载请注明出处。

思路：扫描字符串，“ ”字符向后继续扫描 遇到+ - 号修改符号变量并且后移一位，判断是否是数字，如果是数字一直后移每后移一次结果\*10+当前字符表示的数字，当遇到不是数字的字符时跳出循环用结果\*符号变量判断是否溢出最后返回结果

代码

int getValue(char ch){

switch(ch){

case'1':return 1;

case'2':return 2;

case'3':return 3;

case'4':return 4;

case'5':return 5;

case'6':return 6;

case'7':return 7;

case'8':return 8;

case'9':return 9;

case'0':return 0;

default: return -1;

}

}

/\*'0' 48 \*/

int myAtoi(string str) {

int symbol = 1;

long int ans = 0;

int i = 0;

while((int)str[i] == 32 && i <str.size()){

i++;

}

if((int)str[i] == 45 && i<str.size()){

symbol = -1;

i++;

}else if((int)str[i] == 43 && i<str.size()){

i++;

}

int k = 0;

while((int)str[i]>=48 && (int)str[i]<=57 && i<str.size()){

k++;

ans = ans\*10 + getValue(str[i]);

if(ans\*symbol > 2147483647){

return 2147483647;

}else if(ans\*symbol < -2147483648){

return -2147483648;

}

i++;

}

if(k == 0){

return 0;

}else{

return ans\*symbol;

}

}

思路2：正则表达式过滤字符串

数组类型：

1.

给定一个整数数组 nums 和一个目标值 target，请你在该数组中找出和为目标值的那 两个 整数，并返回他们的数组下标。

你可以假设每种输入只会对应一个答案。但是，你不能重复利用这个数组中同样的元素。

来源：力扣（LeetCode）

链接：https://leetcode-cn.com/problems/two-sum

著作权归领扣网络所有。商业转载请联系官方授权，非商业转载请注明出处。

思路用一个map将下标和对应的值对应起来，然后用target-nums[i] 得到的这个值如果可以在map中找到而且下标不等于当前的i那么说明找到了结果，返回I和这个值就是答案

代码

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

map<int,int> nummap;

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

nummap[nums[i]] = i;

}

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

int ans = target - nums[i];

if(nummap.find(ans) != nummap.end() && nummap[ans] != i){

vector<int> a;

a.push\_back(i);

a.push\_back(nummap[ans]);

return a;

}

}

return{};

}

# 2020/11/4刷题打卡

给你两个有序整数数组 nums1 和 nums2，请你将 nums2 合并到 nums1 中，使 nums1 成为一个有序数组。

说明：

初始化 nums1 和 nums2 的元素数量分别为 m 和 n 。

你可以假设 nums1 有足够的空间（空间大小大于或等于 m + n）来保存 nums2 中的元素。

示例：

输入：

nums1 = [1,2,3,0,0,0], m = 3

nums2 = [2,5,6], n = 3

输出：[1,2,2,3,5,6]

提示：

-10^9 <= nums1[i], nums2[i] <= 10^9

nums1.length == m + n

nums2.length == n

来源：力扣（LeetCode）

链接：https://leetcode-cn.com/problems/merge-sorted-array

著作权归领扣网络所有。商业转载请联系官方授权，非商业转载请注明出处。

方法1：合并两个数组，然后使用Arrays工具类对数组从小到大排序

代码：class Solution {

public void merge(int[] nums1, int m, int[] nums2, int n) {

System.arraycopy(nums2, 0, nums1, m, n);

Arrays.sort(nums1);

}

}

* 时间复杂度 : O((n + m)\log(n + m))*O*((*n*+*m*)log(*n*+*m*))。
* 空间复杂度 : O(1)*O*(1)。

作者：LeetCode

方法二：单独列出来一个数组nums3，nums3是nums1的前m个元素，然后对nums3和nums2进行归并，即 双指针归并的方法

代码：

class Solution {

    public void merge(int[] nums1, int m, int[] nums2, int n) {

        int[] nums3 = new int[m];

        for(int i=0;i<m;i++){

            nums3[i] = nums1[i];

        }

        int p1 = 0;

        int p2 = 0;

        int k = 0;

        while(p1<m && p2<n){

            if(nums3[p1] < nums2[p2]){

                nums1[k++] = nums3[p1++];

            }else{

                nums1[k++] = nums2[p2++];

            }

        }

        while(p1<m){

            nums1[k++] = nums3[p1++];

        }

        while(p2<n){

            nums1[k++] = nums2[p2++];

        }

    }

}

***方法2是一个固定的两个数组归并所用的方法 可以当模板记住***

***时间复杂度O(m+n)***

***空间复杂度O(m)***

方法3：对于本题可以减少空间复杂度 因为nums1已经有足够的空间，那么可以不用使用额外的空间，从nums1的最高位 也是按双指针方法，从大到小的归并，那么不用申请新的nums3 本方法只适用于本题目，方法二更普遍。

代码：

class Solution {

    public void merge(int[] nums1, int m, int[] nums2, int n) {

        int index1 = m-1;

int index2 = n-1;

int cur = m+n-1;

        while(index1>=0 && index2 >= 0){

            if(nums1[index1]>nums2[index2]){

nums1[cur--] = nums1[index1--];

}else{

nums1[cur--] = nums2[index2--];

}

        }

        while(index1 >= 0){

            nums1[cur--] = nums1[index1--];

        }

        while(index2 >= 0){

             nums1[cur--] = nums2[index2--];

        }

    }

}

***时间复杂度O(m+n)***

***空间复杂度O(1)***

# 2020/11/5刷题打卡

和谐数组是指一个数组里元素的最大值和最小值之间的差别正好是1。

现在，给定一个整数数组，你需要在所有可能的子序列中找到最长的和谐子序列的长度。

示例 1:

输入: [1,3,2,2,5,2,3,7]

输出: 5

原因: 最长的和谐数组是：[3,2,2,2,3].

说明: 输入的数组长度最大不超过20,000.

来源：力扣（LeetCode）

链接：https://leetcode-cn.com/problems/longest-harmonious-subsequence

著作权归领扣网络所有。商业转载请联系官方授权，非商业转载请注明出处。

**方法一**，找出所有的 x,x+1 然后再次扫描数组，计算x,x+1的总数，最后取最大值

**方法二**，先扫描数组 然后将不同的数存如map中，key是数，value是key在数组中出现的次数，然后扫描map 查找x,x+1的总数的和，最后取最大值

**方法三**，暂时没理解

# 2020/11/7刷题打卡

给你一个由大小写英文字母组成的字符串 s 。

一个整理好的字符串中，两个相邻字符 s[i] 和 s[i+1]，其中 0<= i <= s.length-2 ，要满足如下条件:

若 s[i] 是小写字符，则 s[i+1] 不可以是相同的大写字符。

若 s[i] 是大写字符，则 s[i+1] 不可以是相同的小写字符。

请你将字符串整理好，每次你都可以从字符串中选出满足上述条件的 两个相邻 字符并删除，直到字符串整理好为止。

请返回整理好的 字符串 。题目保证在给出的约束条件下，测试样例对应的答案是唯一的。

注意：空字符串也属于整理好的字符串，尽管其中没有任何字符。

解题思路： 准备一个空字符串ret, 扫描字符串s 如果ret中的最后一个字符与s中扫描到的字符不是互为大小写的字母，那么就将s中的这个字符加入到ret中，如果是互为大小写的字符那么ret中需要将最后一个字符弹出直到s中的字符被扫描完，返回ret字符串

代码：

class Solution {

    public String makeGood(String s) {

       StringBuffer ret = new StringBuffer();

        int retIndex = -1;

        int length = s.length();

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

            char ch = s.charAt(i);

            if (ret.length() > 0 && Character.toLowerCase(ret.charAt(retIndex)) == Character.toLowerCase(ch) && ret.charAt(retIndex) != ch) {

                ret.deleteCharAt(retIndex);

                retIndex--;

            } else {

                ret.append(ch);

                retIndex++;

            }

        }

        return ret.toString();

    }

}

知识点 StringBuffer对象的删除对应下标的字符 deleteCharAt(index),字符转大小写 Character.toLowerCase(ch) 转大写 Character.toUpperCase(ch); StringBuffer转字符串类型使用toString()方法。

# 2020/11/8刷题打卡

给定一个非负索引 k，其中 k ≤ 33，返回杨辉三角的第 k 行。

在杨辉三角中，每个数是它左上方和右上方的数的和。

示例:

输入: 3

输出: [1,3,3,1]

进阶：

你可以优化你的算法到 O(k) 空间复杂度吗？

用了一个双重循环 两个数组 每次算下一行的时候带上上一次计算的数组 一层层的算下去 直到想要的层

第二种方法是数学上数列的问题，可用根据公式直接求解，就是O(k)的算法 k是层数。

来源：力扣（LeetCode）

链接：https://leetcode-cn.com/problems/pascals-triangle-ii

著作权归领扣网络所有。商业转载请联系官方授权，非商业转载请注明出处。

# 2020/11/10刷题打卡

矩形以列表 [x1, y1, x2, y2] 的形式表示，其中 (x1, y1) 为左下角的坐标，(x2, y2) 是右上角的坐标。矩形的上下边平行于 x 轴，左右边平行于 y 轴。

如果相交的面积为 正 ，则称两矩形重叠。需要明确的是，只在角或边接触的两个矩形不构成重叠。

给出两个矩形 rec1 和 rec2 。如果它们重叠，返回 true；否则，返回 false 。

示例 1：

输入：rec1 = [0,0,2,2], rec2 = [1,1,3,3]

输出：true

示例 2：

输入：rec1 = [0,0,1,1], rec2 = [1,0,2,1]

输出：false

示例 3：

输入：rec1 = [0,0,1,1], rec2 = [2,2,3,3]

输出：false

提示：

rect1.length == 4

rect2.length == 4

-109 <= rec1[i], rec2[i] <= 109

rec1[0] <= rec1[2] 且 rec1[1] <= rec1[3]

rec2[0] <= rec2[2] 且 rec2[1] <= rec2[3]

通过次数32,247提交次数64,763

来源：力扣（LeetCode）

链接：https://leetcode-cn.com/problems/rectangle-overlap

著作权归领扣网络所有。商业转载请联系官方授权，非商业转载请注明出处。

思路： 题目要求是 矩形的长与x平行 宽与y平行 所以 考虑 所有的不重合情况就行，之前试过写重合的情况 发现比较麻烦

不重合情况其实就4种 一个矩形的 上面部分 只要第二个矩形最下面的边高于第一个矩形最上面的边 即 y1>y2,依次类推有四种不重合的区域 然后 加上 x相等 y相等的情况 即一条线的情况是永远不重合的 这样就得出了if的条件然后判断重合。

class Solution {

    public boolean isRectangleOverlap(int[] rec1, int[] rec2) {

        int x1 = rec1[0];

        int y1 = rec1[1];

        int x2 = rec1[2];

        int y2 = rec1[3];

        int x3 = rec2[0];

        int y3 = rec2[1];

        int x4 = rec2[2];

        int y4 = rec2[3];

        if(y1 >= y4 || y2 <= y3 || x1 >= x4 || x2 <= x3 || x1 == x2 || y1 == y2 || x3 == x4 || y3 == y4){

            return false;

        }else{

            return true;

        }

    }

}

# 2020/11/6补签

统计一个数字在排序数组中出现的次数。

示例 1:

输入: nums = [5,7,7,8,8,10], target = 8

输出: 2

示例 2:

输入: nums = [5,7,7,8,8,10], target = 6

输出: 0

限制：

0 <= 数组长度 <= 50000

思路1：

遍历数组 遇到与target一致的数让ans++最后返回ans

class Solution {

    public int search(int[] nums, int target) {

        int ans = 0;

        int n = nums.length;

        for(int i = 0;i<n;i++){

            if(nums[i] == target){

                ans++;

            }

        }

        return ans;

    }

}

## 数组的长度是 nums.length 是取的属性！

## 字符串的长度是 str.length() 使用的是方法！

来源：力扣（LeetCode）

链接：https://leetcode-cn.com/problems/zai-pai-xu-shu-zu-zhong-cha-zhao-shu-zi-lcof

著作权归领扣网络所有。商业转载请联系官方授权，非商业转载请注明出处。

# 2020/11/16

对于字符串 S 和 T，只有在 S = T + ... + T（T 与自身连接 1 次或多次）时，我们才认定 “T 能除尽 S”。

返回最长字符串 X，要求满足 X 能除尽 str1 且 X 能除尽 str2。

示例 1：

输入：str1 = "ABCABC", str2 = "ABC"

输出："ABC"

示例 2：

输入：str1 = "ABABAB", str2 = "ABAB"

输出："AB"

示例 3：

输入：str1 = "LEET", str2 = "CODE"

输出：""

提示：

1 <= str1.length <= 1000

1 <= str2.length <= 1000

str1[i] 和 str2[i] 为大写英文字母

通过次数30,340提交次数51,795

来源：力扣（LeetCode）

链接：https://leetcode-cn.com/problems/greatest-common-divisor-of-strings

著作权归领扣网络所有。商业转载请联系官方授权，非商业转载请注明出处。

解题思路一：枚举所有前缀 判断是否可以组成两个字符串 如果是 返回字符串 否则 继续

解题思路二：对于字符串 如果有共同的除数 S1=XXX,S2 = XX 那么 一定有 s1+s2 = s2+s1 反之亦成立 那么 代码如下：

辗转相除法的递归模式 以及字符串的一个知识点

class Solution {

    public String gcdOfStrings(String str1, String str2) {

          // 解决两个问题   1.枚举所有可能的前缀  2.判断前缀是否可以拼接成两个字符串  如果可以 返回此前缀

          // 数学方法  如果有 str1 和 str2 有相同从除数  如  str1 = XXX  str2 = XX 那么  str1+str2 = str2+str1;

          // 假设str1是N个x，str2是M个x，那么str1+str2肯定是等于str2+str1的。

        if (!(str1 + str2).equals(str2 + str1)) {

            return "";

        }

        // 辗转相除法求gcd。

        return str1.substring(0, gcd(str1.length(), str2.length()));

    }

    private int gcd(int a, int b) {

        return b == 0? a: gcd(b, a % b);

    }

}

# 2020/11/17

实现一种算法，找出单向链表中倒数第 k 个节点。返回该节点的值。

注意：本题相对原题稍作改动

示例：

输入： 1->2->3->4->5 和 k = 2

输出： 4

说明：

给定的 k 保证是有效的。

来源：力扣（LeetCode）

链接：https://leetcode-cn.com/problems/kth-node-from-end-of-list-lcci

著作权归领扣网络所有。商业转载请联系官方授权，非商业转载请注明出处。

/\*\*

 \* Definition for singly-linked list.

 \* public class ListNode {

 \*     int val;

 \*     ListNode next;

 \*     ListNode(int x) { val = x; }

 \* }

 \*/

class Solution {

    public int kthToLast(ListNode head, int k) {

        // 1. 数出来节点的个数x 2. 计算倒数第k个是正数第几个 x-k+1 5-2+1=4

        int x = 1;

        ListNode p;

        p = head;

        while(p.next != null){

            p = p.next;

            x++;

        }

        p = head;

        for(int i=1;i<x-k+1;i++){

            p = p.next;

        }

        return p.val;

    }

}

# 2020/11/19

Your friend is typing his name into a keyboard.  Sometimes, when typing a character c, the key might get long pressed, and the character will be typed 1 or more times.

You examine the typed characters of the keyboard.  Return True if it is possible that it was your friends name, with some characters (possibly none) being long pressed.

Example 1:

Input: name = "alex", typed = "aaleex"

Output: true

Explanation: 'a' and 'e' in 'alex' were long pressed.

Example 2:

Input: name = "saeed", typed = "ssaaedd"

Output: false

Explanation: 'e' must have been pressed twice, but it wasn't in the typed output.

Example 3:

Input: name = "leelee", typed = "lleeelee"

Output: true

Example 4:

Input: name = "laiden", typed = "laiden"

Output: true

Explanation: It's not necessary to long press any character.

来源：力扣（LeetCode）

链接：https://leetcode-cn.com/problems/long-pressed-name

著作权归领扣网络所有。商业转载请联系官方授权，非商业转载请注明出处。

我的代码：

class Solution {

    public boolean isLongPressedName(String name, String typed) {

         // 双指针 指向两个字符串  i,j   如果 i == j 指针向后移动

         //如果i != j 那么可能是j重复  也可能是不同的字符， 若此时的j = i-1的值 j向后移动 直到j != i-1  比较此时的j 与 i

        if(name.charAt(0) != typed.charAt(0)){

            return false;

        }

         int m = name.length();

         int n = typed.length();

         int i = 0;

         int j = 0;

        while(i<m && j<n){

         if(name.charAt(i) == typed.charAt(j)){ //如果字符是相同的 之间向后移动

             i++;

             j++;

         }else{

            if(name.charAt(i) != typed.charAt(j) && i>0 && typed.charAt(j) == name.charAt(i-1) ){ //如果不同

                j++;

            }else if(name.charAt(i) != typed.charAt(j) && i>0 && typed.charAt(j) != name.charAt(i-1)){

                return false;

            }

         }

        }

        while(j<n && typed.charAt(j) == name.charAt(i-1)){

            j++;

        }

        if(i==m && j==n){

            return true;

        }else{

            return false;

        }

    }

}

别人的代码：

class Solution {

public boolean isLongPressedName(String name, String typed) {

int i = 0, j = 0;

while (j < typed.length()) {

if (i < name.length() && name.charAt(i) == typed.charAt(j)) {

i++;

j++;

} else if (j > 0 && typed.charAt(j) == typed.charAt(j - 1)) {

j++;

} else {

return false;

}

}

return i == name.length();

}

}

作者：LeetCode-Solution

链接：https://leetcode-cn.com/problems/long-pressed-name/solution/chang-an-jian-ru-by-leetcode-solution/

来源：力扣（LeetCode）

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。