**一维DP, 直接推公式，注意空间优化，一般可以到O(1)**

70. Climbing Stairs

You are climbing a stair case. It takes n steps to reach to the top.

Each time you can either climb 1 or 2 steps. In how many distinct ways can you climb to the top?

Note: Given n will be a positive integer.

**S1 DP**

用dp[i] 代表number of distinct ways to climb to i, = dp[i-1] + dp[i-2]

注意初始化，初始条件：dp[1]=1, dp[2]=2

space O(n)

public int climbStairs2(int n) {

if (n <= 2) return n;

int[] dp = new int[n + 1];

dp[1] = 1;

dp[2] = 2;

for (int i = 3;i <= n;i++) {

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

}

return dp[n];

}

**S2 空间优化：space O(1) 只和前面两个数字有关，只保存前面两个即可**

public int climbStairs(int n) {

if (n <= 2) return n;

int prev2step = 1, prev1step = 2;

int tmp = 0;

for (int i = 3;i <= n;i++) {

tmp = prev2step;

prev2step = prev1step;

prev1step += tmp;

}

return prev1step;

}

也可以用滚动数组，非常清晰：

public int climbStairs(int n) {

if (n <= 2) return n;

int[] dp = new int[3];

dp[1] = 1;

dp[2] = 2;

for (int i = 3;i <= n;i++) {

dp[i % 3] = dp[(i - 1) % 3] + dp[(i - 2) % 3];

}

return dp[n % 3];

}

121. Best Time to Buy and Sell Stock

Say you have an array for which the ith element is the price of a given stock on day i.

If you were only permitted to complete at most one transaction (ie, buy one and sell one share of the stock), design an algorithm to find the maximum profit.

Input: [7, 1, 5, 3, 6, 4]

Output: 5 max. difference = 6-1 = 5 (not 7-1 = 6, as selling price needs to be larger than buying price)

**题解：**

用反过来的minEndHere即可，

每个时刻i：此时的minEndHere，是之前的最小值，

用(prices[i] – minEndHere)获得的是在当前时刻卖出所获的最大利润。

*// minEndHere[i]: min number of prices[0, i]*

public int maxProfit(int[] prices) {

if (prices == null || prices.length < 2) return 0;

int minEndHere = prices[0], max = 0;

for (int i = 1;i < prices.length;i++) {

max = Math.max(max, prices[i] - minEndHere);

minEndHere = Math.min(minEndHere, prices[i]);

}

return max;

}

198. House Robber

一个int array表示每家有的钱数，相邻两家不能同时抢，问最多抢多少钱

**S1. DP**

dp[i]: rob前[0, i]栋房子，能拿到的最多的钱

time O(n), space O(n)

*// dp[i]: max money can get for robbing [0,i], and i is robbed for sure*

public int rob(int[] nums) {

if (nums == null || nums.length == 0) return 0;

if (nums.length < 2) return nums[0];

int n = nums.length;

int[] dp = new int[n];

dp[0] = nums[0];

dp[1] = Math.max(nums[0], nums[1]);

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

dp[i] = Math.max(nums[i] + dp[i-2], dp[i-1]); // 两种情况：抢当前的房子，和不抢

}

return Math.max(dp[n-2], dp[n-1]);

}

空间优化：可以优化到O(1)

其实每个点只和前两个点有关，

public int rob(int[] nums) {

if (nums == null || nums.length == 0) return 0;

if (nums.length < 2) return nums[0];

int n = nums.length;

int a = nums[0], b = Math.max(nums[0], nums[1]);

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

if (i % 2 == 1) b = Math.max(a, b + nums[i]);

else a = Math.max(b, a + nums[i]);

}

return Math.max(a, b);

}

另一种空间优化的方法 – 滚动数组，也很简洁：

public int rob(int[] nums) {

if (nums == null || nums.length == 0) return 0;

if (nums.length < 2) return nums[0];

int n = nums.length;

int[] dp = new int[2];

dp[0] = nums[0];

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

dp[i % 2] = Math.max(dp[(i-1) % 2], dp[i % 2] + nums[i]);

}

return Math.max(dp[0], dp[1]);

}

213. House Robber II

和上面相似，只是房子不是排成一行了，而是排成一个圈。首尾两个点现在相连了。

**S1 DP**

这样考虑会比较容易：

假设输入是 [1,3,4,2,7,5], 那么需要考虑两种情况，都算出来，并且取最大值：

[1,(3,4,2,7,5)] 相当于不偷第一家；

[(1,3,4,2,7), 5] 相当于不偷最后一家。

所以用helper function来写，并且用a表示: 上一个奇数位一定取的max profit

b表示：上一个偶数位一定取的max profit.

这样两个倒换着来，space 只需要是O(1)

*// {1, 3, 9, 5, 2, 4} 有两种是max的可能性：*

*// {(1, 3, 9, 5, 2), 4} OR {1, (3, 9, 5, 2, 4)}*

public int rob(int[] nums) {

if (nums == null || nums.length == 0) return 0;

int len = nums.length;

int left = helper(nums, 0, len - 1);

int right = helper(nums, 1, len);

return Math.max(left, right);

}

// the max profit can get for [start, end]

// 这里的helper function也可以用别的方式来写，和上面的题目一样

public int helper(int[] nums, int start, int end) {

if (nums == null || nums.length == 0 || start >= nums.length) return 0;

int a = nums[start], b = 0;

for (int i = start + 1; i < end;i++) {

if ((i - start) % 2 == 1) {

b = Math.max(a, b + nums[i]); // 保证一定不相连

} else {

a = Math.max(b, a + nums[i]);

}

}

return Math.max(a, b);

}

337. House Robber III

The thief has found himself a new place for his thievery again. There is only one entrance to this area, called the "root." Besides the root, each house has one and only one parent house. After a tour, the smart thief realized that "all houses in this place forms a binary tree". It will automatically contact the police if two directly-linked houses were broken into on the same night.

Determine the maximum amount of money the thief can rob tonight without alerting the police.

Maximum amount of money the thief can rob = 3 + 3 + 1 = 7.



**S1. Naive Recursion**

这其实是一个level order的问题，如果选了某个level, 就不能选相邻的两个level.

可以直接用recursion来做

public int rob(TreeNode root) {

if (root == null) return 0;

int val = root.val; *// max that can get with root*

if (root.left != null) {

val += rob(root.left.left) + rob(root.left.right);

}

if (root.right != null) {

val += rob(root.right.left) + rob(root.right.right);

}

return Math.max(val, rob(root.left) + rob(root.right));

}

**S2. Solve Overlapping Problem**

上一个做法有一个显著的问题，有很多overlapping problems，走到每一层时，都会把下面的所有层又都算一遍。

事实上对于每个TreeNode而言，都有两种可能性，add / not add，

所以每次返回int[] 来表示，res[0]表示不包含root.val, res[1]表示包含root.val

public int rob(TreeNode root) {

int[] res = helper(root);

return Math.max(res[0], res[1]);

}

*// res[0]: without root.val, next level*

*// res[1]: with root.val*

private int[] helper(TreeNode root) {

if (root == null) return new int[]{0, 0};

int[] left = helper(root.left);

int[] right = helper(root.right);

int[] res = new int[2];

res[0] = Math.max(left[0], left[1]) + Math.max(right[0], right[1]);

res[1] = root.val + left[0] + right[0];

return res;

}

276. Paint Fence

There is a fence with n posts, each post can be painted with one of the k colors.

You have to paint all the posts such that **no more than two adjacent fence** posts have the same color.

Return the total number of ways you can paint the fence.

**S1. DP**

dp[i]: # of ways to point the first i posts, 有两种可能性，

1.和dp[i-2]不一样 = (k-1)\*dp[i-2]

2.和dp[i-1]不一样 = (k-1)\*dp[i-1]

init dp[0] = k, dp[1] = k\*k， return dp[n-1]

time: O(n), space: O(n)

public int numWays(int n, int k) {

if (n == 0 || k == 0) return 0;

if (n == 1) return k;

int[] dp = new int[n];

dp[0] = k;

dp[1] = k\*k;

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

dp[i] = (k - 1)\*(dp[i - 1] + dp[i - 2]);

}

return dp[n - 1];

}

**空间优化：**

可以到 O(1)，只需要存之前两步的状态，用滚动数组就可以，很简单

public int numWays(int n, int k) {

if (n == 0 || k == 0) return 0;

if (n == 1) return k;

int[] dp = new int[3];

dp[0] = k;

dp[1] = k\*k;

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

dp[i % 3] = (k - 1)\*(dp[(i - 1) % 3] + dp[(i - 2) % 3]);

}

return dp[(n - 1) % 3];

}

256. Paint House

There are a row of n houses, each house can be painted with one of the three colors: red, blue or green. The cost of painting each house with a certain color is different. You have to paint all the houses such that no two adjacent houses have the same color.

The cost of painting each house with a certain color is represented by a n x 3 cost matrix. For example, costs[0][0] is the cost of painting house 0 with color red; costs[1][2] is the cost of painting house 1 with color green, and so on... Find the minimum cost to paint all houses.

是上道题的加强版，不仅需要考虑color，还需要考虑cost

**S1 DP**

time O(n), space O(1)

其实和213. Home Robber II 有些相像，都需要更新多个值，并且是根据其他数值来更新多个值。

用minRed, minBlue, minGreen来表示处理到某个cost时，当前选择red, blue, green所需要的min cost.

因为有三个需要设置，如果直接写 minRed = cost[0] + Math.min(minGreen, minBlue);

那么minRed在此时就会发生变化，后面两个就不能直接用，所以我们用curRed来暂存。

public int minCost(int[][] costs) {

int minRed = 0, minBlue = 0, minGreen = 0;

int curRed = 0, curBlue = 0, curGreen = 0;

for (int[] cost:costs) { *// cost[i][0-2] indicates cost of RBG*

curRed = cost[0] + Math.min(minGreen, minBlue);

curBlue = cost[1] + Math.min(minGreen, minRed);

curGreen = cost[2] + Math.min(minBlue, minRed);

minRed = curRed;

minBlue = curBlue;

minGreen = curGreen;

}

return Math.min(minRed, Math.min(minGreen, minBlue));

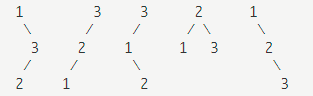
}

96. Unique Binary Search Trees

Given n, how many structurally unique BST's (binary search trees) that store values 1...n?

For example,

Given n = 3, there are a total of 5 unique BST's.



**S1. DP**

dp[i]: # of ways when node i is root.

内循环：j指的是# of nodes in the left-subtree, 可以是0到i-1,

那么与此对应，# of nodes in the right-subtree就会是i-j-1. 相当于总数-左边-root

public int numTrees(int n) {

if (n <= 0) return 0;

int[] dp = new int[n + 1];

dp[0] = 1;

for (int i = 1;i <= n;i++) { // node as root

for (int j = 0;j < i;j++) { // j: nodes on left-subtree.

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

}

}

return dp[n];

}

91. Decode Ways

A message containing letters from A-Z is being encoded to numbers using the following mapping:



Given an encoded message containing digits, determine the total number of ways to decode it.

For example, Given encoded message "12", it could be decoded as "AB" (1 2) or "L" (12).

The number of ways decoding "12" is 2.

**S1. 从前往后走**

划分为oneDigit 和twoDigit分别处理，

容易出问题的地方就是遇到”0”

public int numDecodings(String s) {

if (s == null || s.length() == 0 || s.charAt(0) == '0') return 0;

int len = s.length();

int[] dp = new int[len + 1];

dp[0] = 1;

dp[1] = 1; *// s, [0, 1)*

for (int i = 2;i <= len;i++) { // i是没有取到的位置

int oneDigit = Integer.valueOf(s.substring(i - 1, i));

int twoDigits = Integer.valueOf(s.substring(i - 2, i));

if (oneDigit >= 1 && oneDigit <= 9)

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

if (twoDigits >= 10 && twoDigits <= 26)

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

}

return dp[len];

}

**S2. 从后往前走**

如果遇到0就跳过

public int numDecodings(String s) {

if (s == null) return 0;

int len = s.length();

if (len == 0) return 0;

int[] nums = new int[len + 1];

nums[len] = 1;

nums[len - 1] = s.charAt(len - 1) != '0' ? 1 : 0;

for (int i = len - 2;i >= 0;i--) {

if (s.charAt(i) == '0') continue;

else if (Integer.parseInt(s.substring(i, i + 2)) <= 26) {

nums[i] = nums[i + 1] + nums[i + 2];

} else {

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

}

}

return nums[0];

}

DP 二维循环

两个for循环，内外循环，推公式 – 最经典的DP

**300. Longest Increasing Subsequence**

Given an unsorted array of integers, find the length of longest increasing subsequence.

For example, Given [10, 9, 2, 5, 3, 7, 101, 18],

The longest increasing subsequence is [2, 3, 7, 101], therefore the length is 4.

Your algorithm should run in **O(n^2**) complexity.

Follow up: Could you **improve it to O(n log n) time complexity?**

最长递增数组，数字间不需要相邻

**S1. DP**

**maxEndHere, maxSoFar推公式**

time O(n^2), 遍历，maxEndHere[i]含义是：在包含nums[i]的情况下的最大值，

针对每个i, 需要检查maxEndHere[0, i-1]的每个值是否满足匹配，并更新maxSoFar.

public int lengthOfLIS(int[] nums) {

if (nums == null || nums.length == 0) return 0;

int max = 0， n = nums.length;

int[] dp = new int[n]; // 相当于是maxEndHere

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

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

if (nums[j] < nums[i]) dp[i] = Math.max(dp[i], dp[j] + 1);

}

max = Math.max(max, dp[i]);

}

return max + 1; // 加上自己

}

**S2. Binary Search**

time O(n logn), List中保存的是递增的数组，这个方法有三个关键点，

1. 什么时候往list里面加数字？仅仅当num>list中最后一个数字时，才会append
2. 不满足append数字的条件时，也就代表num比上一个数字小，则list中对应位置的数字会被num替代。但这只是list中某个数字更换，它的大小不变。
3. only size of list matters. 并且size在遍历中只可能增长。

public int lengthOfLIS(int[] nums) {

if (nums == null || nums.length == 0) return 0;

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

for (int num:nums) {

if (res.size() == 0 || res.get(res.size() - 1) < num) {

res.add(num);

} else {

int index = Collections.binarySearch(res, num);

if (index < 0) index = -(index + 1);

res.set(index, num);

}

}

return res.size();

}

651. 4 Keys Keyboard

Imagine you have a special keyboard with the following keys:

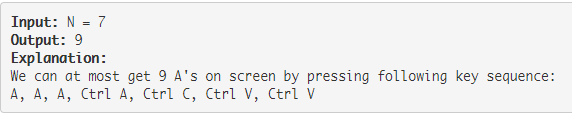
Key 1: (A): Print one 'A' on screen.

Key 2: (Ctrl-A): Select the whole screen.

Key 3: (Ctrl-C): Copy selection to buffer.

Key 4: (Ctrl-V): Print buffer on screen appending it after what has already been printed.

Now, you can only press the keyboard for N times (with the above four keys), find out the maximum numbers of 'A' you can print on screen.



**S1.DP O(n^2), space O(n)**

非常标准的DP，dp[i]的初始值是i，只print；

dp[i]的最大值可能来自于只print，或者dp[j] \* (i-j-1)，for j in [0, i-1]

相当于复制dp[j]数量的‘A’，粘贴(i-j-1)次。

public int maxA(int N) {

if (N < 1) return 0;

int[] dp = new int[N + 1]; // press keyboard i times得到的最多的’A’的数目

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

dp[i] = i; *// init: copy once and paste all the time*

for (int j = 1;j <= i-3;j++) {

*// dp[j]: max # of A for [0, j], max # of A could be copied*

*// [j+1, i]: # of times being pasted after copy, (i-j-1)*

dp[i] = Math.max(dp[i], dp[j]\*(i – j - 1));

}

}

return dp[N];

}

646. Maximum Length of Pair Chain

You are given n pairs of numbers. In every pair, the first number is always smaller than the second number.

Now, we define a pair (c, d) can follow another pair (a, b) if and **only if b < c.** Chain of pairs can be formed in this fashion.

Given a set of pairs, find the length longest chain which can be formed. You needn't use up all the given pairs. You can select pairs in any order.

Input: [[1,2], [2,3], [3,4]] Output: 2 Explanation: The longest chain is [1,2] -> [3,4]

**S1 Sort + DP**

每个pair是[start, end]的格式，就先按照starting point排序，然后只需要考虑end point。

确定maxEndHere[i] 的值需要考虑上一个节点是maxEndHere[j] when j in [0, i-1],

**Time O(n^2)**

public int findLongestChain(int[][] pairs) {

if (pairs == null || pairs.length == 0) return 0;

*// 1. sort based on starting point*

Arrays.sort(pairs, (a, b)->(a[0] - b[0]));

*// 2. dp[i]=longest chain for pairs for [0,i]*

*// to get dp[i], need to check dp[j] for j in [0, i-1], and find max*

int[] maxEndHere = new int[pairs.length];

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

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

*// when previous pair is pairs[j]:*

int cur = pairs[i][0] > pairs[j][1] ? maxEndHere[j] + 1 : maxEndHere[j];

maxEndHere[i] = Math.max(maxEndHere[i], cur);

}

}

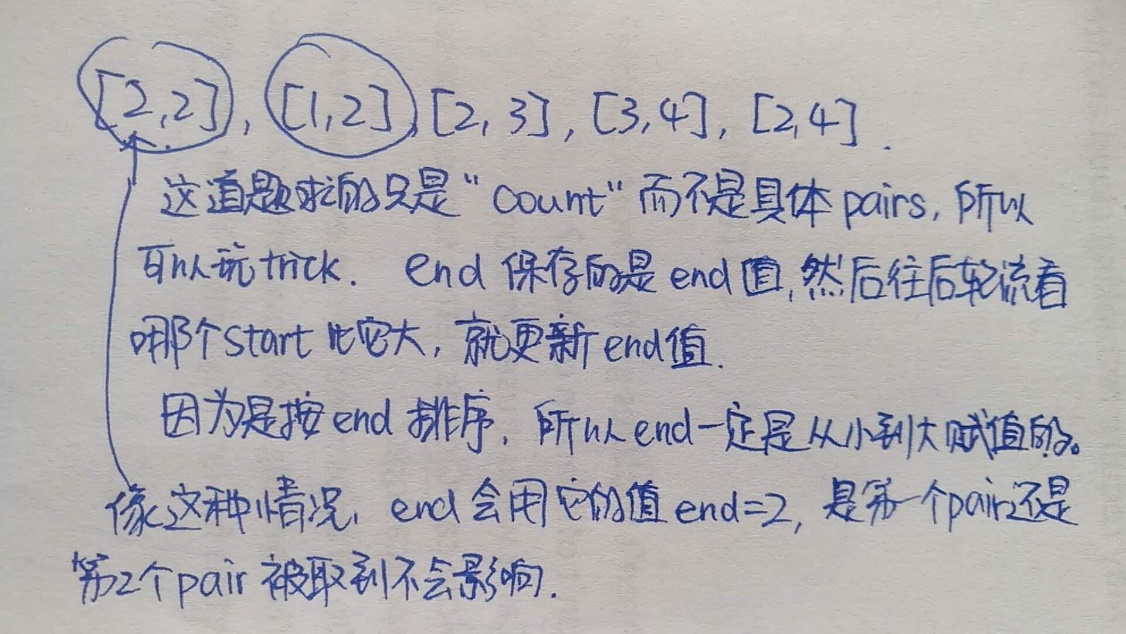
return maxEndHere[pairs.length - 1] + 1;

}

**S2 Pointer time O(n logn)， space O(1)**

保证end有序，从前往后遍历，只记录上一个满足chain条件的end，这个end一定是所有满足条件中的最小值。

同理可以用按照start排序来做，不过这样需要从后往前遍历数组。



public int findLongestChain(int[][] pairs) {

if (pairs == null || pairs.length == 0) return 0;

Arrays.sort(pairs, (a, b)->(a[1] - b[1]));

int max = 0, end = Integer.MIN\_VALUE;

for (int[] pair:pairs) {

if (pair[0] > end) {

max++;

end = pair[1];

}

}

return max;

}

按照start排序：

public int findLongestChain2(int[][] pairs) {

if (pairs == null || pairs.length == 0) return 0;

Arrays.sort(pairs, (a, b)->(a[0] - b[0]));

int max = 0, start = Integer.MAX\_VALUE;

for (int i = pairs.length - 1;i >= 0;i--) {

if (pairs[i][1] < start) {

max++;

start = pairs[i][0];

}

}

return max;

}

Whitepages/Hiya - Minimum number of jumps to reach end

Given an array of integers where each element represents the max number of steps that can be made forward from that element. Write a function to return the minimum number of jumps to reach the end of the array (starting from the first element). If an element is 0, then cannot move through that element.

**S1 DP**

public static int minJumpNum(int[] nums) {

if (nums == null) return -1;

int n = nums.length;

int[] dp = new int[n + 1];

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

dp[i] = i + 1; // set init value – invalid

*// current node: nums[i-1]*

*// min step:1, max step: nums[i-1]*

for (int j = 0;j < i;j++) { // 查看之前的各个点，能不能到达i的位置

if (j + nums[j] >= i) { // 如果可以，更新dp[i]值

dp[i] = Math.min(dp[i], dp[j] + 1);

}

}

}

return dp[n] == n + 1 ? -1:dp[n];

}

72. Edit Distance

Given two words word1 and word2, find the minimum number of steps required to convert word1 to word2. (each operation is counted as 1 step.)

You have the following 3 operations permitted on a word:

a) Insert a character

b) Delete a character

c) Replace a character

S1. DP time O(mn)

非常经典的计算2个string之间distance的题目，可以直接按照公式推

public int minDistance(String word1, String word2) {

if (word1 == null && word2 == null) return -1;

if (word1 == null || word1.length() == 0) return word2.length();

if (word2 == null || word2.length() == 0) return word1.length();

int m = word1.length(), n = word2.length();

int[][] dp = new int[m + 1][n + 1];

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

for (int j = 1;j <= m;j++) dp[j][0] = j;

*// 要特别注意，哪些地方要用i-1, j-1, 哪些地方用i, j*

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

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

if (word1.charAt(i - 1) == word2.charAt(j - 1)) {

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

} else {

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

}

}

}

return dp[m][n];

}

161. One Edit Distance

Given two strings S and T, determine if they are both one edit distance apart.

**S1. DP + 早回头 time O(min(m, n)), space O(1)**

和上道题的区别是，只需要判断是否distance = 1，

换句话说，用两个指针指向S,T的头，并排往后走，直到走到第一个char不等的位置，然后判断之后的string是否相等即可。

public boolean isOneEditDistance(String s, String t) {

int m = s.length(), n = t.length();

if (Math.abs(m - n) > 1) return false;

int min = Math.min(m, n);

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

if (s.charAt(i) == t.charAt(i)) conitnue;

if (m == n)

return s.substring(i+1).equals(t.substring(i+1));

else if (m - n == 1)

return s.substring(i+1).equals(t.substring(i));

else

return s.substring(i).equals(t.substring(i+1));

}

*// distance could be 0(same length) or 1*

return Math.abs(m - n) == 1;

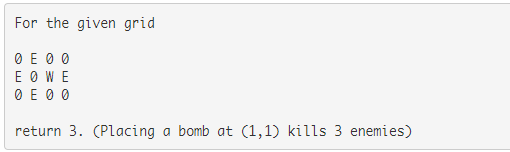
}

361. Bomb Enemy

Given a 2D grid, each cell is either a wall 'W', an enemy 'E' or empty '0' (the number zero), return the maximum enemies you can kill using one bomb.

The bomb kills all the enemies in the same row and column from the planted point until it hits the wall since the wall is too strong to be destroyed.

Note that you can only put the bomb at an empty cell.



**S1. DP time O(mn\*k), space O(m+n), m, n, are size of grid, k is # of walls**

用两个数组rows, cols分别表示在grid[i][j]放置炸弹，在这一行能炸死的人数，和这一列能炸死的人数。

对于rows来说，在i=0 || grid[i][j-1]=W时需要计算一次；

对于cols来说， 在j=0 || grid[i-1][j]=W时需要计算一次。

由于是按行遍历，所以rows可以进一步优化为row，space可以优化到O(m).

public int maxKilledEnemies2(char[][] grid) {

if (grid == null || grid.length == 0) return 0;

int m = grid.length, n = grid[0].length;

int row = 0;

int[] col = new int[n];

int max = 0;

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

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

if (grid[i][j] == 'W') continue;

if (j == 0 || grid[i][j - 1] == 'W') {

*// there is a wall, # before can't be used again,*

*// need to re-calculate, from j-th col to next wall/border*

row = killedEnemyInRow(grid, i, j);

}

if (i == 0 || grid[i - 1][j] == 'W') {

col[j] = killedEnemyInCol(grid, i, j);

}

if (grid[i][j] == '0') {

max = Math.max(max, row + col[j]);

}

}

}

return max;

}

*//calculate killed enemies for row i from column j*

private int killedEnemyInRow(char[][] grid, int i, int j){

int num = 0;

while(j <= grid[0].length-1 && grid[i][j] != 'W'){

if(grid[i][j] == 'E') num++;

j++;

}

return num;

}

*//calculate killed enemies for column j from row i*

private int killedEnemyInCol(char[][] grid, int i, int j){

int num = 0;

while(i <= grid.length -1 && grid[i][j] != 'W'){

if(grid[i][j] == 'E') num++;

i++;

}

return num;

}

二维DP 以及 MemorizedSearch

**这两种其实都可以归为解决overlap subproblems**

**dp:** 从下向上，保存过程中所有的值，便于下一步使用，最后求到最终结果。

如果dp[i][j]的值已经求过，可以用dp直接调用；

如果还没有求过，可以通过公式直接求得一步一步规规矩矩解决。

**memorizedSearch:** 从上向下，求解过程中求的都是需要的值，不需要的没有求，用递归实现

如果dp[i][j]的值已经求过，直接返回；

如果没有求过，需要调用递归求解。

记忆化搜索memorizedSearch的基本套路：

int memorizedSearch(Problem P) {

if (invalid condition) return 0/null/...;

if (a has been solved before) return Previous\_Result;

else {

divide P into several subproblems {P1, P2, P3...},

get {memorizedSearch(P1), memorizedSearch(P2)..} and get the Optimal\_Result of them;

write Optimal\_Result to DP\_Results;

return Optimal\_Result;

}

}

Lintcode - Longest Common Subsequence

Given two strings, find the longest common subsequence (LCS). Return the length of LCS.

LCS for input Sequences ABCDGH and AEDFHR is ADH of length 3.

LCS for input Sequences AGGTAB and GXTXAYB is GTAB of length 4.

<http://www.lintcode.com/en/problem/longest-common-subsequence/>

**Follow up:** 返回LCS – 不是长度，而是string

**S1 DP:**

**dp[i][j]: LCS for A[0, i], B[0, j],** 一开始想到dp = new int[m][n]即可，但是问题是会有：

dp[i][j] = dp[i-1][j-1]这样类似的操作，需要考虑i=0, j=0时的边界问题，所以可以一开始

建立new int[m+1][n+1]来解决。

**S1.1**

public int longestCommonSubsequence(String A, String B) {

if (A == null || B == null || A.length() == 0 || B.length() == 0) return 0;

int m = A.length(), n = B.length();

int[][] dp = new int[m][n];

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

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

if (A.charAt(i) == B.charAt(j)) {

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

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

} else {

if (i == 0 || j == 0) continue;

dp[i][j] = Math.max(dp[i - 1][j], dp[i][j - 1]);

}

}

}

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

}

**S1.2 简化版本**

public int longestCommonSubsequence(String A, String B) {

int m = A.length(), n = B.length();

int[][] dp = new int[m + 1][n + 1];

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

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

if (A.charAt(i - 1) == B.charAt(j - 1)) {

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

} else {

dp[i][j] = Math.max(dp[i - 1][j], dp[i][j - 1]);

}

}

}

return dp[m][n];

}

**MemorizedSearch:**

**helper function的基本套路： 主要目的：解决subproblem**

ResultObject memorizedSearch(Problem P) {

if (invalid condition) return 0/null/...;

if (a has been solved before) return Previous\_Result;

else {

divide P into several subproblems {P1, P2, P3...},

get {memorizedSearch(P1), memorizedSearch(P2)..} and get the Optimal\_Result of them;

write Optimal\_Result to DP\_Results;

return Optimal\_Result;

}

}

public int longestCommonSubsequence(String A, String B) {

int m = A.length(), n = B.length();

// dp[i][j]: len of LCS for A[0,m-1], B[0,n-1]

return memorizedSearch(A, B, m, n, new int[m][n]);

}

*// get length of LCS for A[0, m-1], B[0, n-1]*

int memorizedSearch(String A, String B, int m, int n, int[][] dp) {

if (m == 0 || n == 0) return 0;

if (dp[m-1][n-1] != 0) return dp[m-1][n-1];

if (A.charAt(m-1) == B.charAt(n-1)) {

dp[m-1][n-1] = 1 + memorizedSearch(A, B, m - 1, n - 1, dp);

} else {

dp[m-1][n-1] = Math.max(memorizedSearch(A, B, m, n - 1, dp),

memorizedSearch(A, B, m - 1, n, dp));

}

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

}

**Follow up:**

用prev和cur这两个pointer，来判断是不是一个新的LCS, 如果是新的的话，就把旧的丢掉，直接create，

如果是在原来的基础上，就append

public static String longestCommonSubstring(String s1, String s2) {

if (s1 == null || s2 == null || s1.length() == 0 || s2.length() == 0) {

return "";

}

int m = s1.length(), n = s2.length();

String res = "";

int[][] dp = new int[m + 1][n + 1];

int maxLen = 0;

int prev = -1; *// the index of previous match location.*

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

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

if (s1.charAt(i-1) == s2.charAt(j-1)) {

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

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

if (dp[i][j] == maxLen && prev != cur) {

res = res.substring(0, res.length()-1) + s2.charAt(j-1);

}

if (dp[i][j] > maxLen) {

maxLen = dp[i][j];

if (prev == -1 || prev == cur) {

res = res + s2.charAt(j-1);

} else {

prev = cur;

res = "" + s2.charAt(j-1);

}

}

} else {

dp[i][j] = Math.max(dp[i - 1][j], dp[i][j - 1]);

}

}

}

return res;

}

115. Distinct Subsequences

Given a string S and a string T, count the number of distinct subsequences of S which equals T.

A subsequence of a string is a new string which is formed from the original string by deleting some (can be none) of the characters without disturbing the relative positions of the remaining characters

(ie, "ACE" is a subsequence of "ABAACDCE" while "AEC" is not).

S = "rabbbit", T = "rabbit"，Return 3.

**S1. DP**

这个题的重点在于转移公式的求解，T的index是外循环，S的index是内循环，

根据charAt(i), charAt(j)是否相等进行分类

dp[i][j]: # of distinct subsequences of T.substring[0, i], S.substring[0, j]

其实另一种表述会更好：对于T[0, i]，S[0, j]有多少种不同的subsequence可以match上

所以在对应char不相等时：对于T[0, i]，S[0, j]能match的数量相比s[0, j-1]而言并没有变化，所以取之前的值；

在对应char相等时：对于T[0, i], 有两种可能：

1. 用刚刚对应的char来match，这样的distinct subsequence有多少个 = dp[i-1][j-1]
2. the number we had before, 也就是原有的，不用刚刚对应上的char来match的值, dp[i][j-1]

数值就是这两者相加。

注意初始化：j = 0, 也就是T长度为0时，是任意string的substring，数量为1.

public int numDistinct(String s, String t) {

if (s == null || t == null || s.length() == 0 || t.length() == 0) return 0;

int m = t.length();

int n = s.length();

int[][] dp = new int[m+1][n+1];

Arrays.fill(dp[0], 1); *// "" is a subsequence of any string but only 1 time.*

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

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

if (t.charAt(i-1) == s.charAt(j-1)) {

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

} else {

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

}

}

}

return dp[m][n];

}

329. Longest Increasing Path in a Matrix

Given an integer matrix, find the length of the longest increasing path.

From each cell, you can either move to four directions: left, right, up or down. You may NOT move diagonally or move outside of the boundary (i.e. wrap-around is not allowed).



Return 4, The longest increasing path is [1, 2, 6, 9].

**S1. DP 注意overlapping问题**

避免重复计算：通过dfs中的第一句实现，如果已经计算过，则直接返回。

public int longestIncreasingPath(int[][] matrix) {

if (matrix == null || matrix.length == 0) return 0;

int maxLen = 1;

int m = matrix.length, n = matrix[0].length;

int[][] dp = new int[m][n]; // dp[i][j]: longest increasing path start in matrix[i][j]

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

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

maxLen = Math.max(maxLen, dfs(matrix, i, j, dp));

}

}

return maxLen;

}

private int dfs(int[][] matrix, int x, int y, int[][] dp) {

if (dp[x][y] != 0) return dp[x][y];

int[] xbias = new int[]{0,0,1,-1};

int[] ybias = new int[]{1,-1,0,0};

int max = 1;

int m = matrix.length, n = matrix[0].length;

for (int i = 0;i < 4;i++) {

int curx = x + xbias[i], cury = y + ybias[i];

if (curx < 0 || cury < 0 || curx >= m || cury >= n

|| matrix[curx][cury] <= matrix[x][y]) continue;

int cur = 1 + dfs(matrix, curx, cury, dp);

max = Math.max(max, cur);

}

dp[x][y] = max; // 及时update

return max;

}

S2. 还可以用memorized Search来做

Form a palindrome

<http://practice.geeksforgeeks.org/problems/form-a-palindrome/0>

Given a string, find the minimum number of characters to be inserted to convert it to palindrome.

For Example: ab: Number of insertions required is 1. bab or aba

**思路：**

dp[i][j]: min # of characters to make substring[i, j] a palindrome

i loop from 0 to n-1,

j loop from i to 0,(重点是，通过这种遍历方法，使两者间interval从0加大来计算)

check if s.charAt(i) == s.charAt(j) or not,

如果相等, dp[i][j] = dp[i+1][j-1]，如果i,j相邻，dp[i][j]=dp[j][i]=0

如果不等, 有两种情况，dp[i][j-1], dp[i+1][j]，找出这两者中的小的+1

int minCharsConvertToPalindrome(String s) {

if (s == null || s.length() == 0) return 0;

int len = s.length();

int[][] dp = new int[n][n];

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

for (int j = i;j >= 0;j--) {

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

else {

if (s.charAt(i) == s.charAt(j)) {

dp[i][j] = dp[j][i] = (j+1 == i) ? 0 : dp[j+1][i-1];

} else {

dp[i][j] = dp[j][i] = (j+1 == i) ? 1:1+Math.min(dp[j+1][i], dp[j][i-1])

}

}

}

}

return dp[0][n-1];

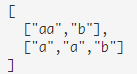
}

131. Palindrome Partitioning

Given a string s, partition s such that every substring of the partition is a palindrome.

Return all possible palindrome partitioning of s.

For example, given s = "aab",Return



**S1. DP + DFS**

首先用DP，dp[i][j]代表s.substring[i, j]是不是palindrome, 便于后续直接查找，

全部查清楚之后，再用dfs，其实也就是backtrack的第二种情况，找到所有解。思路清晰，打败93%

public List<List<String>> partition(String s) {

List<List<String>> result = new ArrayList<>();

int n = s.length();

boolean[][] dp = new boolean[n][n];

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

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

if (s.charAt(i) == s.charAt(j)

&& (i - j <= 2 || dp[j+1][i-1]))

dp[i][j] = dp[j][i] = true;

}

}

dfs(result, new ArrayList<>(), s, dp, 0);

return result;

}

private void dfs(List<List<String>> res, List<String> list,

String s, boolean[][] dp, int idx) {

if (idx == s.length()) {

res.add(new ArrayList<>(list));

return;

}

int n = s.length();

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

if (!dp[i][idx]) continue;

list.add(s.substring(idx, i + 1));

dfs(res, list, s, dp, i + 1);

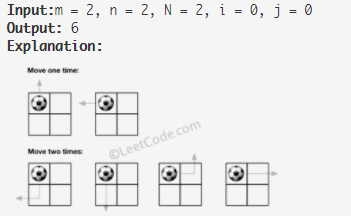
list.remove(list.size() - 1);

}

}

576. Out of Boundary Paths

There is an m by n grid with a ball. Given the start coordinate (i,j) of the ball, you can move the ball to adjacent cell or cross the grid boundary in four directions (up, down, left, right). However, you can at most move N times. Find out the number of paths to move the ball out of grid boundary. The answer may be very large, return it after mod 109 + 7.



求的其实就是，把ball从某个点移除grid有多少种方式。

**S1. DP**

DP[i][j][k] stands for how many possible ways to walk into cell j,k in step i, DP[i][j][k] only depends on DP[i - 1][j][k], so we can compress 3 dimensional dp array to 2 dimensional.

借鉴的是这个帖子的做法：<https://discuss.leetcode.com/topic/88570/java-solution-dp-with-space-compression>

初始化：一开始walk into cell dp[i][j]需要设成1，因为一开始初始的位置就在那个点。

public int findPaths(int m, int n, int N, int i, int j) {

if (i < 0 || j < 0 || i >= m || j >= n || N <= 0) return 0;

final int MOD = 1000000007;

int[][] dp = new int[m][n];

dp[i][j] = 1;

int res = 0;

int[][] dirs = {{-1, 0}, {1, 0}, {0, -1}, {0, 1}};

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

int[][] temp = new int[m][n];

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

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

for (int[] dir:dirs) {

int curRow = row + dir[0], curCol = col + dir[1];

if (curRow < 0 || curRow >= m || curCol < 0 || curCol >= n) {

res = (res + dp[row][col]) % MOD;

} else {

temp[curRow][curCol] = (temp[curRow][curCol] + dp[row][col]) % MOD;

}

}

}

}

dp = temp;

}

return res;

}

DP 三维矩阵

Probability of Knight to remain in the chessboard

Given an NxN chessboard and a Knight at position (x,y). The Knight has to take exactly K steps, where at each step it chooses any of the 8 directions uniformly at random.

What is the probability that the Knight remains in the chessboard after taking K steps?

it can’t enter the board again once it leaves it.

**S1 DP**

建立矩阵dp[n][n][k], 其中的点dp[i][j][m]用来表示在坐标（i, j）处，走m步之后，Knight仍然在棋盘上的概率。

int[] xbias = new int[]{-2,-1,1,2,2,1,-1,-2};

int[] ybias = new int[]{1,2,2,1,-1,-2,-2,-1};

public double findProbability(int x, int y, int steps, int n) {

int[][] prob = new int[n][n];

for (int i = 0;i < n;i++) Arrays.fill(prob[i], 1);

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

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

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

int inside = 0;

for (int m = 0;m < xbias.length;m++) {

if (isInside(i + xbias[m], j + ybias[m], n)) {

inside++;

}

}

prob[i][j] \*= (double)inside / 8;

}

}

}

return prob[x][y];

}

private boolean isInside(int i, int j, int n) {

if (i < 0 || j < 0 || i >= n || j >= n) return false;

return true;

}

**划分类DP**

给定原数组，将其划分为k个子数组，使其sum/product最大/最小。

optimal sub-structrure：

对于给定区间的globalMax，最优解一定由所属区间内的localMax区间组成，可能不连续。

用“必须以当前结尾”, maxEndHere, 来保证loca子问题之间的连续性，

用global来记录最优解可能的不连续性。

与区间类DP的主要区别：

只取其中k段，中间可以不连续；区间类子问题之间是连贯的。

常用方法：

Kadane’s algorithm，相比prefix sum的特点是，必须要以nums[0]做初始化，从index=1开始搜。

prefix sum

53. Maximum Subarray

Find the contiguous subarray within an array (containing at least one number) which has the largest sum.

For example, given the array [−2,1,−3,4,−1,2,1,−5,4],the contiguous subarray [4,−1,2,1] has the largest sum = 6.

**S1 区间和 Prefix sum**

从m到n的子数组的和为：sum(n) – sum(m-1), sum(n)为固定值，则达到最大子数组需要最小化sum(m-1)，所以需要维护min(sum(m-1)) 和sum of nums[0, i]。

time O(n), space O(1)

public int maxSubArray(int[] nums) {

if (nums == null || nums.length == 0) return 0;

int sumEndHere = 0; *// sum of nums[0, i]*

int minSum = Integer.MAX\_VALUE, maxSum = Integer.MIN\_VALUE;

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

sumEndHere += nums[i];

maxSum = Math.max(maxSum, sumEndHere - minSum);

minSum = Math.min(minSum, sumEndHere);

}

return maxSum;

}

**S2 Kadane’s Algorithm**

通过局部变量localMax: maxEndHere

来递推全局变量：maxSoFar.

必须要以nums[0] 做初始化，从index=1开始搜索。

public int maxSubArray(int[] nums) {

if (nums == null || nums.length == 0) return 0;

int maxSoFar = nums[0], maxEndHere = nums[0];

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

maxEndHere = nums[i] + Math.max(0, maxEndHere);

maxSoFar = Math.max(maxSoFar, maxEndHere);

}

return maxSoFar;

}

Lintcode、152 - Maximum Product Subarray

**S1. Kadane’s Algorithm**

需要考虑遇到负数和0的情况，min, max可能会反转，所以需要保留max, min两者。

public int maxProduct(int[] nums) {

if (nums == null || nums.length == 0) return 0;

int maxSoFar = nums[0];

int maxEndHere = nums[0], minEndHere = nums[0];

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

if (nums[i] > 0) {

maxEndHere = Math.max(nums[i], maxEndHere\*nums[i]);

minEndHere = Math.min(nums[i], minEndHere\*nums[i]);

} else {

int oldMax = maxEndHere;

maxEndHere = Math.max(nums[i], minEndHere\*nums[i]);

minEndHere = Math.min(nums[i], oldMax\*nums[i]);

}

maxSoFar = Math.max(maxSoFar, maxEndHere);

}

return maxSoFar;

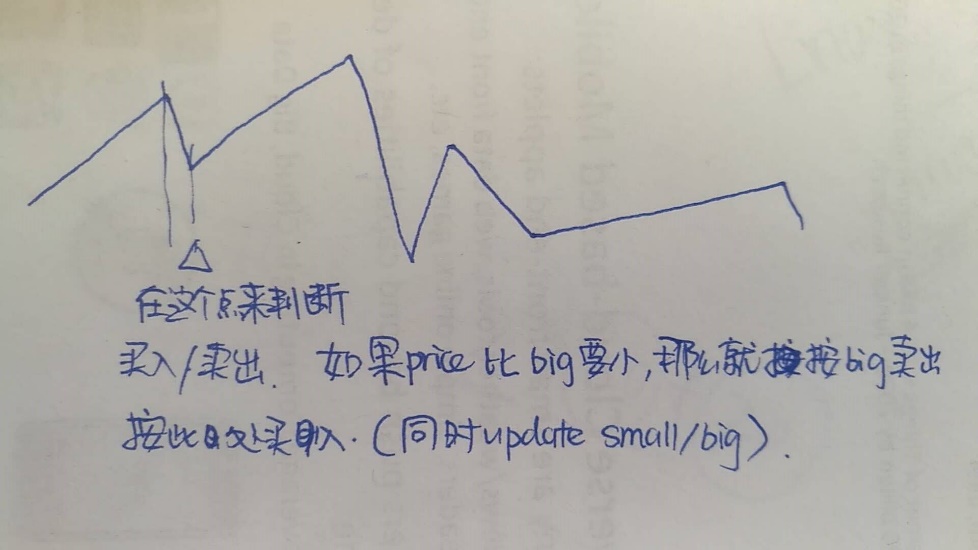
}

122. Best Time to Buy and Sell Stock II

Say you have an array for which the ith element is the price of a given stock on day i.

Design an algorithm to find the maximum profit. You may complete as many transactions as you like (ie, buy one and sell one share of the stock multiple times). However, you may not engage in multiple transactions at the same time (ie, you must sell the stock before you buy again).

和上一题的区别就是，想买多少次都可以，所以策略就是，在从高到底的拐点来判断和更新



public int maxProfit(int[] prices) {

if (prices == null || prices.length < 2) return 0;

int earn = 0;

int small = prices[0];

int big = prices[0];

for (int i = 1;i < prices.length;i++) {

if (prices[i] > big) {

big = prices[i];

} else if (prices[i] < big) {

earn += big - small;

small = big = prices[i];

}

}

return earn + big - small;

}

123. Best Time to Buy and Sell Stock III

Say you have an array for which the ith element is the price of a given stock on day i.

Design an algorithm to find the maximum profit. You may complete at most two transactions.

You may not engage in multiple transactions at the same time (ie, you must sell the stock before you buy again).

题解：time O(n), space O(1)

这道题目最重要的其实是思路，这里的思路可以推到其他数目的transactions上面：

The maximum, if we've just:

sold 2nd stock so far, - release2

buy 2nd stock so far, - hold2

sold 1st stock so far, - release1

buy 1st stock so far. - hold1

并且需要倒着算，不然hold1会影响release1算出的数值

release2 = Math.max(release2, hold2 + price)

hold2 = Math.max(hold2, release1 - price)

release1 = Math.max(release1, hold1 + price)

hold1 = Math.max(hold1, -price);

public int maxProfit(int[] prices) {

int hold1 = Integer.MIN\_VALUE;

int hold2 = Integer.MIN\_VALUE;

int release1 = 0, release2 = 0;

for (int price:prices) {

release2 = Math.max(release2, hold2 + price);

hold2 = Math.max(hold2, release1 - price);

release1 = Math.max(release1, hold1 + price);

hold1 = Math.max(hold1, -price);

}

return release2;

}

188. Best Time to Buy and Sell Stock IV

You may complete at most k transactions.

S1. 用上一道题的方法，用数组来存，timeO(nk), space O(2k)

S2. DP

121. Best Time to Buy and Sell Stock

Say you have an array for which the ith element is the price of a given stock on day i.

If you were only permitted to complete at most one transaction (ie, buy one and sell one share of the stock), design an algorithm to find the maximum profit.

Input: [7, 1, 5, 3, 6, 4] Output: 5

**S1.DP**

Time O(n), Space O(n), minEndHere[i]: min number in prices[0, i]

public int maxProfit1(int[] prices) {

if (prices == null || prices.length < 2) return 0;

int[] minEndHere = new int[prices.length];

minEndHere[0] = prices[0];

int max = 0;

for (int i = 1;i < prices.length;i++) {

max = Math.max(max, prices[i] - minEndHere[i-1]);

minEndHere[i] = Math.min(minEndHere[i-1], prices[i]);

}

return max;

}

**Optimize Space: O(1)**

minEndHere[i]只与minEndHere[i-1]有关，只要取好初始值，就不需要用数组。

public int maxProfit(int[] prices) {

if (prices == null || prices.length < 2) return 0;

int minEndHere = prices[0], max = 0;

for (int i = 1;i < prices.length;i++) {

max = Math.max(max, prices[i] - minEndHere);

minEndHere = Math.min(minEndHere, prices[i]);

}

return max;

}

309. Best Time to Buy and Sell Stock with Cooldown

Say you have an array for which the ith element is the price of a given stock on day i.

Design an algorithm to find the maximum profit. You may complete as many transactions as you like (ie, buy one

and sell one share of the stock multiple times) with the following restrictions:

You may not engage in multiple transactions at the same time (ie, you must sell the stock before you buy again).

After you sell your stock, you cannot buy stock on next day. (ie, cooldown 1 day)

Example:

prices = [1, 2, 3, 0, 2] maxProfit = 3

transactions = [buy, sell, cooldown, buy, sell]

**S1，Kadane’s Algorithm**

local[i] 存储第i天卖出股票的最大利益，global[i] 存储到第i天的全局最大利益。

local[i] 取值有两种可能性：

i-3或者之前进行上一次卖出，i-2 cooldown, i-1到i完成一次交易；

和local[i-1]的交易连续，local[i-1] + prices[i] – prices[i-1]相当于在之前某天买入，到第i天再卖出。

public int maxProfit(int[] prices) {

if (prices == null || prices.length == 0) return 0;

*// local[i]: max profit get when selling stock at day i*

int[] local = new int[prices.length];

int[] global = new int[prices.length];

*// local[0] = 0;*

for (int i = 1;i < prices.length;i++) {

*// i-3 sell, i-2 cooldown, i-1 buy, i sell*

int cooldown = i >= 3 ? global[i-3]:0 + prices[i] - prices[i-1];

*// some time buy, then sell at i.(sell and buy at i == no action at i)*

int continuous = local[i-1] + prices[i] - prices[i-1];

local[i] = Math.max(cooldown, continuous);

global[i] = Math.max(global[i-1], local[i]);

}

return global[prices.length - 1];

}

413. Arithmetic Slices

A sequence of number is called arithmetic if it consists of at least three elements and if the difference between any two consecutive elements is the same.

For example, these are arithmetic sequence:



A zero-indexed array A consisting of N numbers is given. A slice of that array is any pair of integers (P, Q) such that 0 <= P < Q < N.

A slice (P, Q) of array A is called arithmetic if the sequence:

A[P], A[p + 1], ..., A[Q - 1], A[Q] is arithmetic. In particular, this means that P + 1 < Q.

The function should return the number of arithmetic slices in the array A.



其实算作”arithmetic slice”有两点要满足：

1. 长度>= 3，并且需要相连
2. 满足等差数列

**S1. DP**

cur[i]: # of Arithmetic Slices for first i elements

return sum of cur[i] for i = 2 to n-1

public int numberOfArithmeticSlices(int[] A) {

if (A == null || A.length < 3) return 0;

int n = A.length;

int sum = 0;

int[] cur = new int[n];

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

if (A[i] - A[i-1] == A[i-1] - A[i-2]) {

cur[i] = cur[i-1] + 1;

} else {

cur[i] = 0;

}

sum += cur[i];

}

return sum;

}

**空间优化：**

现在是O(n)，但是可以优化成O(1)

public int numberOfArithmeticSlices(int[] A) {

if (A == null || A.length < 3) return 0;

int n = A.length;

int sum = 0;

int cur = 0;

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

if (A[i] - A[i-1] == A[i-1] - A[i-2]) {

cur++;

} else {

cur = 0;

}

sum += cur;

}

return sum;

}

516. Longest Palindromic Subsequence

Given a string s, find the longest palindromic subsequence's length in s. You may assume that the maximum length of s is 1000.

Input: "cbbd"，Output: 2

One possible longest palindromic subsequence is "bb".

**S1. DP time O(mn), space O(mn)**

**dp[i][j]: LPS of s.substring[i, j]**

初始条件：dp[i][i] = 1 for all i

转移公式：if s.charAt(i)==s.charAt(j), dp[i][j] = dp[i+1][j-1]+2

相当于每个位置都和它左下角的数值相关，所以行应该从下到上，列应该从左到右，且j>i

else 为dp[i+1][j], dp[i][j-1]取较大的

返回：dp[0][n-1]

public int longestPalindromeSubseq2(String s) {

if (s == null) return 0;

int n = s.length();

int[][] dp = new int[n][n];

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

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

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

if (s.charAt(i) == s.charAt(j)) {

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

} else {

dp[i][j] = Math.max(dp[i+1][j], dp[i][j-1]);

}

}

}

return dp[0][n-1];

}

Optimize：space 可以到O(n)

public int longestPalindromeSubseq(String s) {

if (s == null) return 0;

int n = s.length();

int[] dp = new int[n];

int prev = 0, current = 0;

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

dp[i] = 1;

prev = 0;

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

current = dp[j];

if (s.charAt(i) == s.charAt(j)) {

dp[j] = prev + 2;

} else {

dp[j] = Math.max(dp[j], dp[j-1]);

}

prev = current;

}

}

return dp[n-1];

}

**S2. MemorizedSearch**

和之前的思路一致，构建subproblem, 以及subproblem的返回条件。

public int longestPalindromeSubseq(String s) {

if (s == null) return 0;

int n = s.length();

return memorizedSearch(s, 0, n-1, new Integer[n][n]);

}

public int memorizedSearch(String s, int i, int j, Integer[][] dp) {

if (dp[i][j] != null) {

return dp[i][j];

}

if (i > j) return 0;

if (i == j) return 1;

if (s.charAt(i) == s.charAt(j)) {

dp[i][j] = memorizedSearch(s, i+1, j-1, dp) + 2;

} else {

int left = memorizedSearch(s, i+1, j, dp);

int right = memorizedSearch(s, i, j-1, dp);

dp[i][j] = Math.max(left, right);

}

return dp[i][j];

}

644. Maximum Average Subarray II

Given an array consisting of n integers, find the contiguous subarray whose length is greater than or equal to k that has the maximum average value. And you need to output the maximum average value.

Given an array consisting of n integers, find the contiguous subarray whose length is greater than or equal to k that has the maximum average value. And you need to output the maximum average value.

Input: [1,12,-5,-6,50,3], k = 4，Output: 12.75

Explanation:

when length is 4, maximum average value is 12.75,

when length is 5, maximum average value is 10.8,

when length is 6, maximum average value is 9.16667. Thus return 12.75.

523. Continuous Subarray Sum

Given a list of non-negative numbers and a target integer k, write a function to check if the array has a continuous subarray of size at least 2 that sums up to the multiple of k, that is, sums up to n\*k where n is also an integer.





**S1. Map**

需要考虑到k的取值，有可能是正数，负数或者0，本来考虑用一个int[], size为k，来保存上一次sum % k取到该值时的index，但是由于k可能为负或者0，所以不行

还是要用map，<mod后的值，上一次取到它的index>，然后每次取到同样的mod后的值时再判断，如果 >= 2，说明成功，返回true

public boolean checkSubarraySum(int[] nums, int k) {

Map<Integer, Integer> map = new HashMap<Integer, Integer>();

map.put(0, -1);

int sum = 0;

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

sum += nums[i];

if (k != 0) sum %= k;

Integer prev = map.get(sum);

if (prev != null) {

if (i - prev > 1) return true;

} else {

map.put(sum, i);

}

}

return false;

}