关于DP的分类和学习可以主要看以下的文章：

<http://blog.csdn.net/tobewhatyouwanttobe/article/details/42805225>

<http://blog.csdn.net/cc_again/article/details/25866971>

DP 博弈类

MinMax

LintCode - Coins in a Line

n coins in a line. Two players take turns to take 1 or 2 coins from right side until no more coins left.

The player who take the last coin wins. Given n, can you decide the first player will win/lose?

**S1.Recursion**

会有重复计算

**S2.DP**

初始化dp[0] = false, dp[1] = dp[2] = true

转移方程：dp[i] = !dp[i-1] || !dp[i-2]

public boolean firstWillWin(int n) {

if (n <= 0) return false;

if (n == 1) return true;

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

dp[1] = true;

dp[2] = true;

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

dp[i] = !dp[i - 1] || !dp[i - 2];

}

return dp[n];

}

LintCode - Coins in a Line II

There are n coins with different value in a line. Two players take turns to take 1 or 2 coins from left until no more coins left.

The player who take the coins with the most value wins. The first player will win/lose?

**S1 DP MinMax**

当前回合我们会选择max profit的走法，下个回合对手会选择min profit的走法。

DP 2D矩阵，滚动数组 – path问题和optimal面积问题

62. Unique Paths

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

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

**How many possible unique paths** are there?

**S1. DP**

dp[i][j]: # of ways to get (i, j), time O(mn), space O(mn), 可以优化到O(m)

public int uniquePaths2(int m, int n) {

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

Arrays.fill(dp[0], 1);

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

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

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

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

}

}

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

}

public int uniquePaths(int m, int n) {

int[] dp = new int[n];

Arrays.fill(dp, 1);

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

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

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

}

}

return dp[n-1];

}

S2. Maths

见C++版本

63. Unique Path II

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

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

For example, there is one obstacle in the middle of a 3x3 grid as illustrated below.

[[0,0,0],

[0,1,0],

[0,0,0]]

The total number of unique paths is 2.

Note: m and n will be at most 100.

**S1. DP**

可以直接用公式推出来。(i, j)的状态只与左侧结点和上一行有关，所以space O(n)

public int uniquePathsWithObstacles(int[][] obstacleGrid) {

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

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

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

dp[1] = 1;

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

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

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

*// cur = up(上一行) + left*

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

}

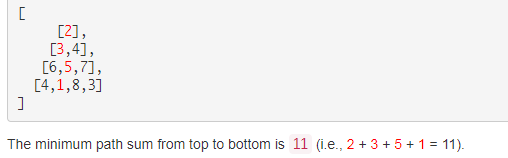
}

return dp[n];

}

120. Triangle

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



**S1 DP**

这个题目最简单的做法是从底向上，

初始值：最后一行的值。为了便于计算，一开始的size设置为最后一行的size + 1.

往上走，倒数第二行，第一位的数字要选从最后一行前两个数字中的一个加上来，选小的那个，

所以是dp[i] = min(dp[j], dp[j+1]) + triangle.get(i).get(j)，这里i是行数，j是这行中的第j个。

往上推，到最顶层，dp[0]即为所求。

public int minimumTotal(List<List<Integer>> triangle) {

int size = triangle.size();

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

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

int min = triangle.get(i).get(j)

+ Math.min(triangle.get(i + 1).get(j), triangle.get(i + 1).get(j + 1));

triangle.get(i).set(j, min);

}

}

return triangle.get(0).get(0);

}

public int minimumTotal(List<List<Integer>> triangle) {

int[] dp = new int[triangle.size() + 1];

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

for (int j = 0;j < triangle.get(i).size();j++) {

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

}

}

return dp[triangle.size()];

}

64. Minimum Path Sum

Given a m x n grid filled with non-negative numbers, find a **path from top left to bottom right** which **minimizes the sum of all numbers along its path**.

Note: You can **only move either down or right** at any point in time.

**S1. DP**

dp[i][j] sum of paths from [0, 0] to [i, j]，直接推公式，

可以在grid上面进行运算。

public int minPathSum(int[][] grid) {

int rows = grid.length;

int cols = grid[0].length;

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

grid[0][i] += grid[0][i - 1];

}

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

grid[i][0] += grid[i - 1][0];

}

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

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

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

}

}

return grid[rows - 1][cols - 1];

}

**区间类DP – 中间连续不留空**

基本思想：

简单来说：把区间分成左右两部分，求出左右区间再合并。

把求解的问题划分成多个子问题，然后按顺序求解各子问题，前一子问题的求解为后一子问题的求解提供了重要的信息，后一子问题根据某种决策来选取前一子问题的解以便解出自身的问题，从这些子问题的解得到原问题的解

区间型DP特征： dp[i][j] = dp[i][k] + dp[k+1][j] + costFunction(i, j)

dp[i][j]表示区间i到j的最优解。

先计算小区间，然后通过小区间迭代，得到大区间的值。

例题：

**Stone Game**

有n堆石子排成一列，每堆石子有一个重量w[i], 每次合并可以合并相邻的两堆石子，一次合并的代价为两堆石子的重量和w[i]+w[i+1]。问安排怎样的合并顺序，能够使得总合并代价达到最小。

题解：

dp[i][j]表示第i堆到第j堆石子合并后的最小代价，

状态转移方程：dp[i][j] = dp[i][k] + dp[k+1][j] + sum(i,j)

可以用memorizedSearch来做

public int stoneGame(int[] stones) {

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

int n = stones.length;

int[][] dp = new int[n][n]; *// min cost to merge stones[i,j]*

int[] sum = new int[n]; *// sum[i]: sum of stones in [0, i]*

sum[0] = stones[0];

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

sum[i] = stones[i] + sum[i-1];

}

return memorizedSearch(stones, dp, sum, 0, n-1);

}

private int memorizedSearch(int[] stones, int[][] dp, int[] sum, int start, int end) {

if (start >= end) return 0; // check corner case

if (start + 1 == end) return stones[start] + stones[end];

if (dp[start][end] != 0) return dp[start][end];

int min = Integer.MAX\_VALUE;

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

int left = memorizedSearch(stones, dp, sum, start, i-1);

int right = memorizedSearch(stones, dp, sum, i, end);

// after merged L/R stones, merge left and right together.

int cur = start == 0 ? sum[end] : sum[end] - sum[start - 1];

min = Math.min(min, left + right + cur);

}

dp[start][end] = min;

return min;

}

**312. Burst Balloons**

Given n balloons, indexed from 0 to n-1. Each balloon is painted with a number on it represented by array nums. You are asked to burst all the balloons. If the you burst balloon i you will get nums[left] \* nums[i] \* nums[right] coins. Here left and right are adjacent indices of i. After the burst, the left and right then becomes adjacent.

Find the **maximum coins you can collect by bursting the balloons wisely**.

Note:

(1) You may imagine nums[-1] = nums[n] = 1. They are not real therefore you can not burst them.

(2) 0 ≤ n ≤ 500, 0 ≤ nums[i] ≤ 100

**题解：**

这道题可以用memorizedSearch做，也可以用DP来做。

关于边界问题，可以新建一个长度+2的数组，在第一个和最后一个放1.

在memorizedSearch内部：left, right是左右区间的max，左右区间取得结果后，coins已经计算过，相当于两边的气球都已经没有了，最后融合那一步的cost是nums[start - 1] \* nums[i] \* nums[end + 1]

public int maxCoins(int[] nums) {

if (nums == null) return 0;

int n = nums.length;

int[] arr = new int[n + 2];

arr[0] = arr[n + 1] = 1;

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

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

}

int[][] dp = new int[n + 2][n + 2];

return memorizedSearch(arr, dp, 1, n);

}

// dp[i][j]: max # of coins that can be get, 通过消除[i, j]范围的气球

// 通过消除[start, end]范围内的气球，能得到的最多的coins number

private int memorizedSearch(int[] nums, int[][] dp, int start, int end) {

if (dp[start][end] != 0) return dp[start][end];

int max = 0;

// 尝试所有的组合：

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

int left = memorizedSearch(nums, dp, start, i-1);

int right = memorizedSearch(nums, dp, i+1, end);

// after processed left and right-left,right对应的气球事实上已经消除：

int cur = nums[start - 1]\*nums[i]\*nums[end + 1];

max = Math.max(max, cur + left + right);

}

dp[start][end] = max;

return max;

}

Coins in a line III

括号匹配，找出最长满足匹配的子序列

**343. Integer Break**

Given a positive integer n, break it into the **sum of at least two positive integers** and maximize the product of those integers. Return the maximum product you can get.

For example, given n = 2, return 1 (2 = 1 + 1); given n = 10, return 36 (10 = 3 + 3 + 4).

Note: You may assume that n is not less than 2 and not larger than 58.

题解：

这道题也是区间类DP，和burst ballons, 石子归并有点像。

用max[i]来存max product of i, 后面的数字可以通过前面的max product得到，

比如求10的max product, 2\*8, 3\*7... 还要看这里面2,8,3,7这些数字的max product是多少，需不需要继续拆。

这里要格外注意一点：at least 2 positive integers.

所以在初始化的时候就需要注意，对于max[n], 只能初始化为n-1, 因为是1\*(n-1), 但是对于其他值i,不需要初始化为i-1,可以初始化为i，因为它可以被直接用于i\*(n-i)来算，只用于内部计算，不会被返回。

里面的循环只要half就够了，避免重复计算。

public int integerBreak(int n) {

int[] max = new int[n + 1]; *// max product of n.*

max[1] = 1;

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

if (i != n) max[i] = i; *// directly i.*

else max[i] = i - 1; *// (i-1) \* 1*

int half = i/2;

for (int j = 2;j <= half;j++) {

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

}

}

return max[n];

}

**132. Palindrome Partitioning II**

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

Return the **minimum cuts needed for a palindrome partitioning** of s.

For example, given s = "aab",

Return 1 since the palindrome partitioning ["aa","b"] could be produced using 1 cut

题解：

求的是最短的划分次数，**global的最短划分次数一定由substring的最短划分次数决定**，而且subtring一定相连，是区间类DP问题。

转移方程：用boolean[][] palindrome表示s[i, j]是不是palindrome，

用int[] dp, dp[i]来表明min # of cut for s[0, i].

初始化：dp[i] = i.

public int minCut(String s) {

if (s == null) return 0;

int n = s.length();

boolean[][] palindrome = new boolean[n][n];

int[] dp = new int[n]; *// dp[i]:min # of cut for s[0,i]*

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

int min = i; // min的初始值，每个char都切分开

// 另一个顺序也可以，j从0加到i.

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

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

((i-j <= 2) || (palindrome[j+1][i-1]))) {

palindrome[j][i] = true;

min = j == 0 ? 0 : Math.min(min, dp[j - 1] + 1);

}

}

dp[i] = min;

}

return dp[n-1];

}

**5. Longest Palindromic Substring**

Given a string s, find **the longest palindromic substring** in s.

You may assume that the maximum length of s is 1000.

题解：

也是从中间向两边延展，属于区间类问题。

**S1**

直接针对每个位置向两侧延伸，每个位置都求一次：击败90%

public String longestPalindrome(String s) {

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

int n = s.length();

int[] index = new int[]{0, 0}; *// [start, end] both included*

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

palindromeHelper(s, i, i, index);

palindromeHelper(s, i, i + 1, index);

}

return s.substring(index[0], index[1] + 1);

}

private void palindromeHelper(String s, int i, int j, int[] index) {

while (i >= 0 && j < s.length() && s.charAt(i) == s.charAt(j)) {

i--;

j++;

}

if (j - i - 1 > index[1] - index[0] + 1) {

index[0] = i + 1;

index[1] = j - 1;

}

}

**214. Shortest Palindrome**

Given a string S, you are allowed to **convert it to a palindrome by adding characters in front of it**. Find and return the shortest palindrome you can find by performing this transformation.

For example:

Given "aacecaaa", return "aaacecaaa".

Given "abcd", return "dcbabcd".

题解：

其实就是求s从0往后的longest palindrome, 起点一定要是0，看看往后最靠后能到多少，然后如果没有到末尾，就把后面剩下的reverse加到前面。

public String shortestPalindrome(String s) {

if (s.length() <= 1) return s;

int i = 0;

int j = s.length() - 1;

int end = j;

while (i < j) {

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

i = 0;

j = --end; // end 每次往前走一步

} else {

i++;

j--;

}

}

StringBuilder sb = new StringBuilder(s.substring(end + 1)).reverse();

return sb.toString() + s;

}

32. Longest Valid Parentheses

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

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

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

题解：

看下面的例子：走到5的时候，长度需要加上3,4两点的长度，到7的时候需要加上0,1两点的长度。

所以global optimal 依赖于sub-optimal, 是区间DP。

主要的trick就在于对未match的左括号的统计，和每次match之后，查看当前匹配之前的index可不可以进行匹配，从而更新count值。

*// 0 1 2 3 4 5 6 7*

*// ( ) ( ( ) ( ) )*

public int longestValidParentheses(String s) {

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

int n = s.length(), max = 0;

int[] count = new int[n]; // count[i]: longest valid parenthesis of s[0,i].

int left = 0;

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

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

left++; // 此处的count[i] == 0.

} else if (left > 0) {

left--;

count[i] = count[i - 1] + 2; *// another ( ) pair match.*

if (i - count[i] > 0) { *// i-count[i]: index before current valid parenthesis*

count[i] += count[i - count[i]];

}

}

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

}

return max;

}

139. Word Break

Given a non-empty string s and a dictionary wordDict containing a list of non-empty words, determine if s can be segmented into a space-separated sequence of one or more dictionary words. You may assume the dictionary does not contain duplicate words.

For example, given s = "leetcode", dict = ["leet", "code"].

Return true because "leetcode" can be segmented as "leet code".

**S1. DP**

非常经典的dp类问题，use a canBreak[] to check if s.substring[j, i] is in dict or not.

Latter condition is based on previous conditions.

**And need to set canBreak[0] = true.**

public boolean wordBreak(String s, List<String> wordDict) {

int n = s.length();

*// contains [0, i] or not.*

boolean[] canBreak = new boolean[n + 1];

canBreak[0] = true;

for (int i = 1;i <= n;i++) { // 其实是substring的后缀，excluded.

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

if (!canBreak[j]) continue;

if (wordDict.contains(s.substring(j, i))) {

canBreak[i] = true;

}

}

}

return canBreak[n];

}

140. Word Break II

Follow up: Return all such possible sentences.

For example, given s = "catsanddog", dict = ["cat", "cats", "and", "sand", "dog"].

A solution is ["cats and dog", "cat sand dog"].

**S1.DFS, with memorization – memorized Search**

主要部分是DFS, 通过s.length() 是否为0 判断有没有处理完，并且通过startswith 进行后续的处理；

用map来存已经process过的string，用于剪枝，use a map, <string, List of string> to store the string s and its corresponding construct components.

public List<String> wordBreak(String s, List<String> wordDict) {

return DFS(s, wordDict, new HashMap<String, LinkedList<String>>());

}

*// s: string not processed/included in map yet.*

List<String> DFS(String s, List<String> wordDict, HashMap<String, LinkedList<String>> map) {

if (map.containsKey(s)) return map.get(s); *// check if processed before.*

LinkedList<String> res = new LinkedList<String>();

if (s.length() == 0) { *// check if end of processing.*

res.add(""); // 后续会通过res的size()来判断是不是只有 “”

return res;

}

for (String word:wordDict) {

if (s.startsWith(word)) {

List<String> sublist = DFS(s.substring(word.length()), wordDict, map);

for (String sub:sublist) { // 把当前的解和recursion的解合并

res.add(word + (sub.isEmpty() ? "" : " ") + sub);

}

}

}

map.put(s, res);

return res;

}

**背包问题及延伸 –**

背包这么经典的问题，不搞定的话早晚会栽在上面。

非常棒的参考资料：背包问题九讲2.0版本

**第一类 01背包问题**

01的意思是，每个物品只能选择放或者不放，也就是只能放一次。最经典的题目。

题目：**Backpack II**

Given n items with size Ai and value Vi, and a backpack with size m.

What's the maximum value can you put into the backpack?

题解：

初始化：也就是每行row为0的时候，初始化为0即可。

转移方程：考虑两种情况，物品i放入背包，或者不放；

* 如果不放,dp[i][j] = dp[i-1][j],
* 如果放，需要腾出A[j-1]的空间，并且value也可以加上，所以是V[i-1] + dp[i-1][j-A[i-1]

public int backPackII(int m, int[] A, int V[]) {

*// dp[i][j] within first i elements with size j, max Value.*

int n = A.length;

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

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

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

*// current node is A[i-1], V[i-1]*

if (j-A[i-1] >= 0) {

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

} else {

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

}

}

}

return dp[n][m];

}

**空间优化：**

每一行其实都只和上一行有关。

这里的关键点是，需要保证dp[i-1][j – A[i-1]]有效。相当于在处理同一行的dp时，会依赖前面的row的值。

所以必须从右往左处理，这样保证用到的数值是上一行的，而不是这一行刚刚处理完的。

public int backPackII(int m, int[] A, int V[]) {

*// dp[i][j] within first i elements with size j, max Value.*

int n = A.length;

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

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

for (int j = m;j >= 1;j--) {

*// current node is A[i-1], V[i-1]*

if (j - A[i-1] >= 0) {

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

}

}

}

return dp[m];

}

**第二类 完全背包**

完全背包和01背包的区别是：每个物品可以选无限次。求最大价值。

**例题：322. Coin Change**

You are given coins of different denominations and a total amount of money amount. Write a function to compute the fewest number of coins that you need to make up that amount. If that amount of money cannot be made up by any combination of the coins, return -1.

Example 1:

coins = [1, 2, 5], amount = 11 return 3 (11 = 5 + 5 + 1)

Example 2:

coins = [2], amount = 3 return -1

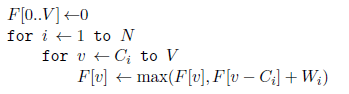
题解：

这道题的题目性质：

需要make up an amount,；**要找出满足条件的最少的数字；每个数字可以用无限次**。

所以是个完全背包问题。有一点小区别就是，这里**不需要求最大价值，求的是最小次数**。

只是把公式中加value的部分换成+1而已。

使用一维数组的算法：

这个算法和01背包的算法的区别就在于，内循环是从左到右进行。

01背包中之所以采用递减的方式，就是为了要保证每个物品只选一次，保证在考虑“选入第i件物品”时，依据的是之前绝对没有选入第i件物品的子结果。

现在可以选无限次，所以正好倒过来。

初始化：只有dp[0]=0, 其他位都必须初始化为某个无效值。而这里要compare min，但是Integer.MAX\_VALUE又是不行的，因为中间会有+1的操作，可能造成overflow.

所以选取了另一个无效值amount+1进行初始化，因为组成amount的数字个数不可能> amount.

public int coinChange(int[] coins, int amount) {

int m = coins.length;

int[] dp = new int[amount + 1]; // dp[i]: min # of coins that could make amount i.

Arrays.fill(dp, amount + 1);

dp[0] = 0;

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

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

*// cur node: coins[j-1].*

if (i - coins[j-1] >= 0) {

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

}

}

}

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

}

两层for循环的顺序可以变化 – 利用这一点进一步优化：

内层的for循环直接设定条件，也不需要if判断

public int coinChange(int[] coins, int amount) {

int m = coins.length;

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

Arrays.fill(dp, amount + 1);

dp[0] = 0;

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

for (int j = coins[i-1];j <= amount;j++) {

*// cur node: coins[j-1].*

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

}

}

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

}

**518. Coin Change 2**

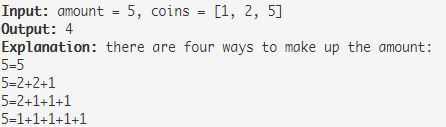
You are given coins of different denominations and a total amount of money. Write a function to **compute the number of combinations that make up that amount**. You may assume that you have infinite number of each kind of coin.

Note: You can assume that

0 <= amount <= 5000

1 <= coin <= 5000

the number of coins is less than 500

the answer is guaranteed to fit into signed 32-bit integer

**题解：**

每个coin可以选无限次，完全背包。

这道题可以算是背包的扩展题，**求的是# of combinations, 有多少种可能性**。

**借着这道题说一下需要特别注意的一点：**

**两个for循环的顺序** – 在这里，2+2+1 和 2+1+2 和 1+2+2 这三项是看作相同的。

这样针对amount的循环在里面。

如果把针对amount的循环放在外面，就会把以上三种情况当做三种情况来考虑，example里面的output就会是9种。

需要针对具体题设来判断for循环的内外设定。

*// dp[i]: # of ways to make up i.*

public static int change(int amount, int[] coins) {

if (coins == null) return 0;

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

int n = coins.length;

dp[0] = 1;

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

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

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

}

}

return dp[amount];

}

**279. Perfect Squares**

Given a positive integer n, find the least number of perfect square numbers (for example, 1, 4, 9, 16, ...) which sum to n.

For example, given n = 12, return 3 because 12 = 4 + 4 + 4; given n = 13, return 2 because 13 = 4 + 9.

题解：

这道题的性质：sum需要是n；可选的元素是特定的perfect square；每个元素可以无限选。

所以是完全背包。

public int numSquares(int n) {

int[] dp = new int[n + 1]; // dp[i]: min # of squares needed for sum i.

Arrays.fill(dp, n + 1);

dp[0] = 0;

int sqrt = (int)Math.sqrt(n);

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

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

int num = j\*j;

if (i-num >= 0) dp[i] = Math.min(dp[i], 1 + dp[i-num]);

}

}

return dp[n];

}

同样，内外循环也可以换，来进行优化 – 超过94%：

public int numSquares(int n) {

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

Arrays.fill(dp, n + 1);

dp[0] = 0;

int sqrt = (int)Math.sqrt(n);

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

int num = i\*i;

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

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

}

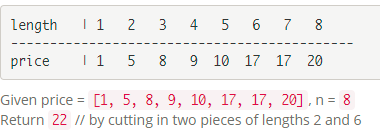
}

return dp[n];

}

例题：**Lintcode - Cutting a Rod**

Given a rod of length n inches and an array of prices that contains prices of all pieces of size smaller than n. Determine the maximum value obtainable by cutting up the rod and selling the pieces.

For example, if length of the rod is 8 and the values of different pieces are given as following, then the maximum obtainable value is 22 (by cutting in two pieces of lengths 2 and 6)

题解：

这里的每种长度可以用无限次。

dp[i]: max # of prices could get when whole length is i.

初始化：dp[0] = 0;

public int cutting(int[] prices, int n) {

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

int m = prices.length;

*// init: dp[0] = 0;*

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

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

*// whole length is i, j is length of current rod.*

if (i-j >= 0) dp[i] = Math.max(dp[i], dp[i-j] + prices[j-1]);

}

}

return dp[n];

}

可以优化，调整内循环的j范围，并且不需要if的判断

public int cutting(int[] prices, int n) {

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

int m = prices.length;

*// init: dp[0] = 0;*

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

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

*// whole length is i, j is length of current rod.*

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

}

}

return dp[n];

}

**第三类 多重背包**

和01背包的区别是，每种物品有一定的可用次数，通过数组输入；

有N种物品，背包容量为V，第i种物品最多有Mi件可用，耗费空间为Ci，价值是Wi，

求背包能装下的最大价值。

**第九类 问法的变化**

**9.0 求解最多可以装满多少背包的空间**

不再用int[][] dp来放价值，而改用boolean[][] 来放“前i个物品能不能刚好装满j的空间”。

例题：**Lintcode - backpack**

Given n items with size Ai, an integer m denotes the size of a backpack. **How full you can fill this backpack?**

题解：

有两点需要注意：初始化+空间优化。

初始化：dp的大小设成dp[n+1][m+1], 主要是为了初始化方便。dp[i][0] 均为true。

空间优化：可以优化到O(m), 要dp[j – A[i-1]]成立，内循环需要从右往左来扫。这样**保证用到的数值是上一行的，而不是这一行刚刚处理完的。**

public int backPack(int m, int[] A) {

*// dp[i][j]: for the first i elements, can we make sum of j*

int n = A.length;

boolean[] dp = new boolean[m + 1];

dp[0] = true;

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

for (int j = m;j >= 1;j--) {

if (j >= A[i - 1]) {

// 注意考虑到dp和A的index之差，current node是A[i-1]

dp[j] = dp[j] || dp[j-A[i-1]];// not choosing current one || choose current node

} else {

dp[j] = dp[j];

}

}

}

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

if (dp[j]) return j;

}

return -1;

}

**416. Partition Equal Subset Sum**

Given a non-empty array containing only positive integers, find if the array can be partitioned into two subsets such that the sum of elements in both subsets is equal.

Input: [1, 5, 11, 5] Output: true

Explanation: The array can be partitioned as [1, 5, 5] and [11].

题解：

这道题和上一道其实很像，只不过target value是数组中所有数的sum的一半。

所以其实还是01背包，是否能找到数字sum up to a target value, 每个数字最多用一次。

*// dp[i][j]: if first i nums could sum to j.*

public boolean canPartition(int[] nums) {

int sum = 0, n = nums.length;

for (int num:nums) sum += num;

if (sum % 2 != 0) return false;

int half = sum / 2;

boolean[][] dp = new boolean[n + 1][half + 1];

dp[0][0] = true;

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

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

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

if (j - nums[i-1] >= 0) {

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

} else {

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

}

}

}

return dp[n][half];

}

空间优化：可以直接优化到O(half)

public boolean canPartition(int[] nums) {

int sum = 0, n = nums.length;

for (int num:nums) sum += num;

if (sum % 2 != 0) return false;

int half = sum / 2;

boolean[] dp = new boolean[half + 1];

dp[0] = true;

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

for (int j = half;j >= 1;j--) {

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

if (j - nums[i-1] >= 0) {

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

}

}

}

return dp[half];

}

在这个基础上，还可以有时间优化，优化的不多 – 去掉if的判断条件

public boolean canPartition(int[] nums) {

int sum = 0, n = nums.length;

for (int num:nums) sum += num;

if (sum % 2 != 0) return false;

int half = sum / 2;

boolean[] dp = new boolean[half + 1];

dp[0] = true;

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

for (int j = half;j >= nums[i-1];j--) {

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

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

}

}

return dp[half];

}

**9.2 求解具体的装背包的方案** – 记录每个状态的最优解，以及是从哪一项推出来的。

**9.3 求方案总数 – 求解有多少种装背包的方法**

只需要把状态转移方程中的求max/min, 改成求sum即可。

初始化：dp[0] = 1. 取得sum = 0的方法有一个，就是什么都不装。而这里dp[i]是方法数量，所以一定要记得设为1.



例题：**377. Combination Sum IV**

Given an integer array with all positive numbers and no duplicates, find the number of possible combinations that add up to a positive integer target.

题解：

**求的是array中的数字的和相加成target的方法有多少种；每个数字可以用无限次。**

完全背包 – 求方案总数。

初始化：dp[0] = 1;

*// dp[i]: # of ways sum up to target.*

public int combinationSum4(int[] nums, int target) {

int[] dp = new int[target + 1]; // # of ways sum to i.

int n = nums.length;

dp[0] = 1;

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

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

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

}

}

return dp[target];

}

**Lintcode - k Sum**

Given n distinct positive integers, integer k (k <= n) and a number target.

Find k numbers where sum is target. Calculate how many solutions there are?

这道题其实就是Combination Sum 4再加上一个条件：

原来是不管多少个数字加起来都行，现在是必须要刚好k个数字相加。

题解：

比Combination Sum4多一个条件，也就多了一维。

dp[i][j][m]: # of ways from first i elements, pick j numbers, get sum m

public int kSum(int A[], int k, int target) {

int n = A.length;

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

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

dp[i][0][0] = 1;

}

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

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

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

if (m - A[i - 1] >= 0) {

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

} else {

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

}

}

}

}

return dp[n][k][target];

}

空间优化：

这里是三维空间，不能直接减掉一维，但是可以保留两个，采用滚动数组的方式。

public int kSum(int A[], int k, int target) {

int n = A.length;

int[][][] dp = new int[2][k + 1][target + 1];

dp[0][0][0] = 1;

dp[1][0][0] = 1;

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

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

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

if (m - A[i - 1] >= 0) {

dp[i%2][j][m] = dp[(i-1)%2][j][m] + dp[(i-1)%2][j-1][m-A[i-1]];

} else {

dp[i%2][j][m] = dp[(i-1)%2][j][m];

}

}

}

}

return dp[n % 2][k][target];

}

**9.4 最优方案总数**

最优方案指的是总价值最大的方案。

背包变形 – 还是DP的做法，但有了改动或者需要预处理

650. 2 Keys Keyboard

Initially on a notepad only one character 'A' is present. You can perform two operations on this notepad for each step:

Copy All: You can copy all the characters present on the notepad (partial copy is not allowed).

Paste: You can paste the characters which are copied last time.

Given a number n. You have to **get exactly n 'A' on the notepad by performing the minimum number of steps** permitted. Output the **minimum number of steps to get n 'A'.**

**S1 DP**

这道题和普通背包有一点不一样，也是可以利用的点：不需要完整的两遍遍历。

内循环从右往左来扫，第一个遇到的值一定比左边的更小，所以其他的可以不用看了，直接可以break。

*// dp[i]: min steps needed to get i 'A',*

*// init: i*

*// transformation:*

public int minSteps(int n) {

if (n < 2) return 0;

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

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

dp[i] = i;

for (int j = i/2;j > 1;j--) {

if (i % j == 0) {

dp[i] = dp[j] + i/j;

break;

}

}

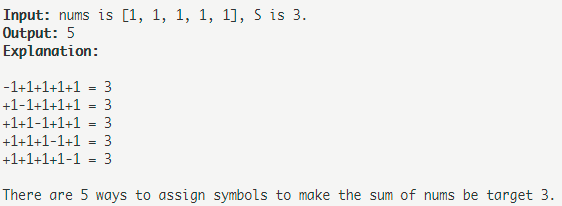
}

return dp[n];

}

494. Target Sum

You are given a list of non-negative integers, a1, a2, ..., an, and a target, S. Now you have 2 symbols + and -. For each integer, you should choose one from + and - as its new symbol.

Find out how many ways to assign symbols to make sum of integers equal to target S

思路：

每个数字有两种选择：正数，负数

相当于把所有的数字分成两组：Positive, Negative. 并且满足P+N = target

而且外面已经知道P+(-N) = sum of all numbers.

所以根据这两个公式可以得到2\*P = target + sum(all). 也就是说，求的是P的可能性。

So the original problem has been converted to a subset sum problem as follows:

Find a subset P of nums such that sum(P) = (target + sum(nums)) / 2

public int findTargetSumWays1(int[] nums, int S) {

int sum = 0;

for (int num:nums) sum += num;

if (sum < S || -sum > S || (sum + S) % 2 != 0) return 0;

return getSumWays((sum + S)/2, nums);

}

*// # of ways to sum up to target*

*// dp[i]: # of ways to sum up to i*

private int getSumWays(int target, int[] nums) {

int n = nums.length;

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

dp[0] = 1;

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

for (int j = target;j >= nums[i];j--) {

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

}

}

return dp[target];

}

**DP 杂题 – 不确定分类**

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.

1 3 3 2 1

\ / / / \ \

3 2 1 1 3 2

/ / \ \

2 1 2 3

**S1. DP**

用dp[i]记录数据从1 ~ i对应的BST个数，其中i可以在1~n中取值，

dp[i]的计算需要考虑root为1~i的情况，比如root为4，则左子树的取值可能性一共有dp[4-1]种，右子树的取值可能性一共有dp[n-4]种。

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++) {

for (int j = 1;j <= i;j++) { *// root is j.*

*// # of possible combinations of left sub-tree\*right sub-tree*

*// contains (j-1) nodes, and (i-j) nodes respectively.*

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

}

}

return dp[n];

}

95. Unique Binary Search Trees II

Follow up: return all the unique trees.

**S1.Recursion**

先找出left subtree, right subtree的所有可能性，再依次取值。

public List<TreeNode> generateTrees(int n) {

return n > 0 ? build(1, n) : new ArrayList<>();

}

private List<TreeNode> build(int start, int end) {

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

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

List<TreeNode> left = build(start, i - 1);

List<TreeNode> right = build(i + 1, end);

for (int m = 0;m < left.size();m++) {

for (int n = 0;n < right.size();n++) {

TreeNode root = new TreeNode(i);

root.left = left.get(m);

root.right = right.get(n);

res.add(root);

}

}

}

if (res.size() == 0) res.add(null);

return res;

}

467. Unique Substrings in Wraparound String

Consider the string s to be the infinite wraparound string of "abcdefghijklmnopqrstuvwxyz", so s will look like this: "...zabcdefghijklmnopqrstuvwxyzabcdefghijklmnopqrstuvwxyzabcd....".

Now we have another string p. Your job is to find out how many unique non-empty substrings of p are present in s. In particular, your input is the string p and you need to output the number of different non-empty substrings of p in the string s.

Input: "zab"，Output: 6

Explanation: There are six substrings "z", "a", "b", "za", "ab", "zab" of string "zab" in the string s.

376. Wiggle Subsequence

A sequence of numbers is called a wiggle sequence if the differences between successive numbers strictly alternate between positive and negative. The first difference (if one exists) may be either positive or negative. A sequence with fewer than two elements is trivially a wiggle sequence.

For example, [1,7,4,9,2,5] is a wiggle sequence because the differences (6,-3,5,-7,3) are alternately positive and negative. In contrast, [1,4,7,2,5] and [1,7,4,5,5] are not wiggle sequences, the first because its first two differences are positive and the second because its last difference is zero.

Given a sequence of integers, return the length of the longest subsequence that is a wiggle sequence. A subsequence is obtained by deleting some number of elements (eventually, also zero) from the original sequence, leaving the remaining elements in their original order.

需要返回的是最长的wiggle subsequence的长度

Input: [1,17,5,10,13,15,10,5,16,8]，Output: 7

There are several subsequences that achieve this length. One is [1,17,10,13,10,16,8].

**题解：**

这道题和常见的DP的区别在于，需要考虑两种情况，用两个DP矩阵来表示。

用up, dn分别表示从i-1到i是上升/下降的情况下，max length of wiggle.

并且dependent of each other,

public int wiggleMaxLength(int[] nums) {

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

int n = nums.length;

int[] up = new int[n]; *// up[i]: max length of wiggle in s[0,i]*

int[] dn = new int[n];

up[0] = dn[0] = 1;

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

if (nums[i] > nums[i-1]) { *// if this is up*

up[i] = dn[i-1] + 1;

dn[i] = dn[i-1];

} else if (nums[i] < nums[i-1]) {

up[i] = up[i-1];

dn[i] = up[i-1] + 1;

} else { *// equal*

up[i] = up[i-1];

dn[i] = dn[i-1];

}

}

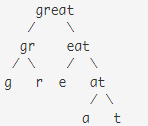
return Math.max(up[n-1], dn[n-1]);

}

87. Scramble String

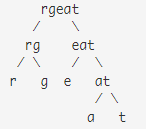
Given a string s1, we may represent it as a binary tree by partitioning it to two non-empty substrings recursively.

Below is one possible representation of s1 = "great":



To scramble the string, we may choose any non-leaf node and swap its two children.

For example, if we choose the node "gr" and swap its two children, it produces a scrambled string "rgeat".

We say that "rgeat" is a scrambled string of "great".

Given two strings s1 and s2 of the same length, determine if s2 is a scrambled string of s1.

题解

这道题其实很难分类，其实就是用recursion来做，

s1，s2是scrambled string，首先需要s1, s2的字母组成相同，否则false

如果满足的话，再用recursion把所有的split方法全部查看一遍

public boolean isScramble(String s1, String s2) {

if ((s1 == null && s2 == null) || s1.equals(s2)) return true;

if (s1 == null || s2 == null

|| s1.length() != s2.length()) return false;

int n = s1.length();

int[] letters = new int[26];

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

letters[s1.charAt(i)-'a']++;

letters[s2.charAt(i)-'a']--;

}

for (int i = 0; i < 26; i++) {

if (letters[i]!=0) return false;

}

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

if (isScramble(s1.substring(0,i), s2.substring(0,i))

&& isScramble(s1.substring(i), s2.substring(i)))

return true;

if (isScramble(s1.substring(0,i), s2.substring(s2.length()-i))

&& isScramble(s1.substring(i), s2.substring(0,s2.length()-i)))

return true;

}

return false;

}

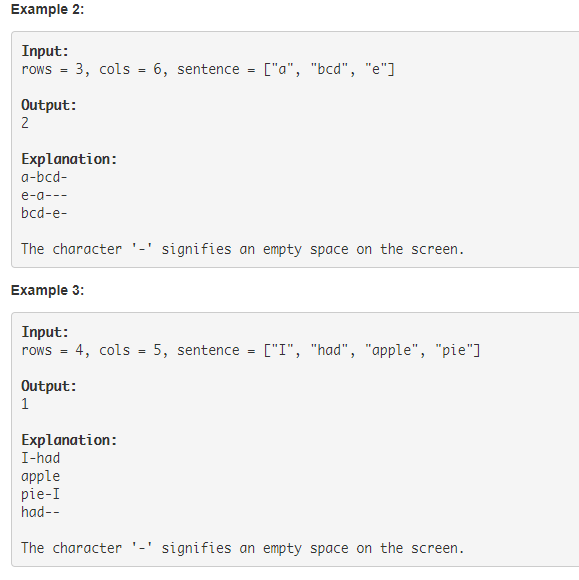
418. Sentence Screen Fitting

Given a rows x cols screen and a sentence represented by a list of non-empty words, find how many times the given sentence can be fitted on the screen.

1.A word cannot be split into two lines.

2.The order of words in the sentence must remain unchanged.

3.Two consecutive words in a line must be separated by a single space.



**S1.DP**

1. 1. String s = String.join(" ", sentence) + " " ;. This line gives us a formatted sentence to be put to our screen.
2. start is the counter for how many valid characters from s have been put to our screen.
3. if (s.charAt(start % l) == ' ') is the situation that we don't need an extra space for current row. The current row could be successfully fitted. So that we need to increase our counter by using start++.
4. The else is the situation, which the next word can't fit to current row. So that we need to remove extra characters from next word.
5. start / s.length() is (# of valid characters) / our formatted sentence.

start 指向的是作为新的一行开始的点，每次+col,

public int wordsTyping(String[] sentence, int rows, int cols) {

String s = String.join(" ", sentence) + " ";

int len = s.length(), start = 0;

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

start += cols;

// 完全fit，这一行的最后一个char刚好用上，并且不是空格

if (s.charAt(start % len) == ' ') start++;

else {

while (start > 0 && s.charAt((start-1) % len) != ' ') {

start--;

}

}

}

return start / len;

}

这个解法需要在每次+col之后，判断start指向的点是否为空格，可以在之前算好：

**优化：**

用map, map[i]记录每个点如果要作为新一行的开始，前面需要多加几个空格。

真的很难想到

public int wordsTyping(String[] sentence, int rows, int cols) {

String s = String.join(" ", sentence) + " ";

int len = s.length(), start = 0;

int[] map = new int[len];

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

map[i] = s.charAt(i) == ' ' ? 1:map[i-1] - 1;

}

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

start += cols;

start += map[start % len];

}

return start / len;

}

403. Frog Jump

A frog is crossing a river. The river is divided into x units and at each unit there may or may not exist a stone. The frog can jump on a stone, but it must not jump into the water.

Given a list of stones' positions (in units) in sorted ascending order, determine if the frog is able to cross the river by landing on the last stone. Initially, the frog is on the first stone and assume the first jump must be 1 unit.

**If the frog's last jump was k units, then its next jump must be either k - 1, k, or k + 1 units.** Note that the frog can only jump in the forward direction.

[0,1,3,5,6,8,12,17]

There are a total of 8 stones.

The first stone at the 0th unit, second stone at the 1st unit,

third stone at the 3rd unit, and so on...

The last stone at the 17th unit.

Return true. The frog can jump to the last stone by jumping

1 unit to the 2nd stone, then 2 units to the 3rd stone, then

2 units to the 4th stone, then 3 units to the 6th stone,

4 units to the 7th stone, and 5 units to the 8th stone.

**S1 DP**

重点是在每个stone上，可以走的step是上一个step-1,step,step+1

用map来存数据，<stone position, set<steps it can take from this stone>>

比如上面的例子：<0, <1>>, <1, <0, 1, 2>> ....

public boolean canCross(int[] stones) {

*// <stone position, set<steps it can take from this stone>>*

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

int n = stones.length;

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

map.put(stones[i], new HashSet<>());

}

map.get(0).add(1);

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

for (int step:map.get(stones[i])) {

*// the index of stone it can reach, after this step*

int reach = stones[i] + step;

if (reach == stones[n-1]) return true;

*// the steps frog can take at this point is:*

*// step-1, step, step+1*

*// not a stone in this position*

if (!map.containsKey(reach)) continue;

map.get(reach).add(step);

if (step - 1 > 0) map.get(reach).add(step-1);

map.get(reach).add(step+1);

}

}

return false;

}