题型一：爬楼梯

70. 爬楼梯

leetcode习题链接

思路分析：

假如当前我站在第8层楼梯，由于每次我都是只能趴1层或2层，因此我

上一个时刻一定要么在第7层，要么在第6层，所以假设爬到第8层的方法

有dp[8]种，那一定是爬到第6层的方法数加上爬上第7层的方法数。

注意，爬到第六层后，是直接抵达8，不能走7，否则重复。

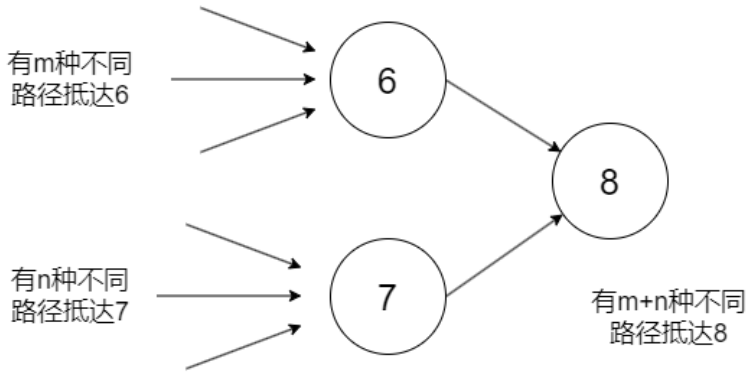
你可能会想，有n种方法抵达7，这其中不是就有途径6的和前面m种路径

重复的吗，是的，确实重复，比如 1356 和 13567 两条路前面完全重复，

不过不碍事，后面就不一样了，一个是13568，一个是135678，这是抵达8的两条不同路径。

所以关键看8之前到底有哪几种状态，哪几种情形。这就是动态规划的第一种思路，分析上一个

状态到底有多少情况。




```
class Solution {
public:
    int climbStairs(int n) {
        vector<int> dp(n+10);
        dp[0] = 0;
        dp[1] = 1;
        dp[2] = 2;
        for (int i = 3; i <= n; i++){
            dp[i] = dp[i-1]+dp[i-2];
        }
        return dp[n];

    }
};
```

746. 使用最小花费爬楼梯

leetcode习题链接

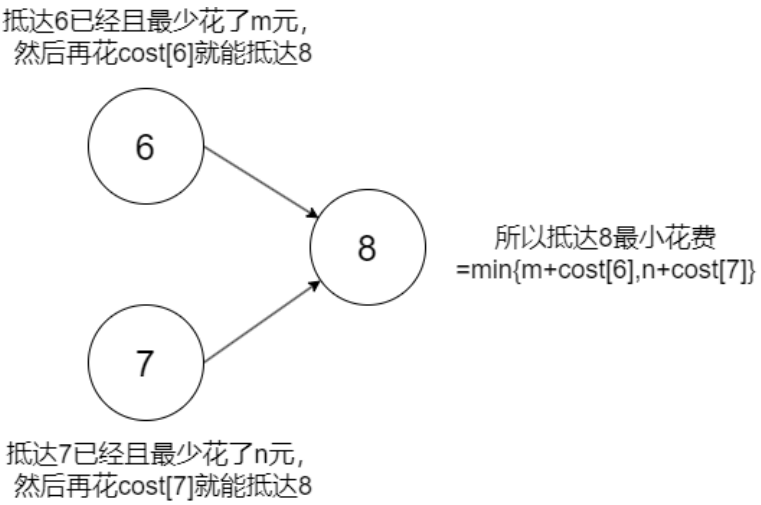
思路分析：

还是属于爬楼梯类型，分析前面有哪几种情况，然后前面的情况一定可以先求出来，就是多米诺骨牌，知道

初始条件，和递推条件，后面一步一步都能求出来，这里分析的就是中间的一个递推过程。

需要注意的是，题目里cost[0]已经是第一层台阶了，cost最后一个数值并不是最后一个台阶，比如cost里面

包含了9个元素，那么其实楼梯顶是第10层，你需要爬到顶。




```
class Solution {
public:
    int minCostClimbingStairs(vector<int>& cost) {
        int n = cost.size();
        vector<int> dp(n+10);
        dp[0]=0;   // 从0出发，所以抵达0不花钱
        dp[1]=0;   // 从1出发，所以抵达1不花钱
        for(int i = 2; i <= n; i++){
            dp[i] = min(dp[i-1]+cost[i-1],dp[i-2]+cost[i-2]);
        }
        return dp[n];
    }
};
```

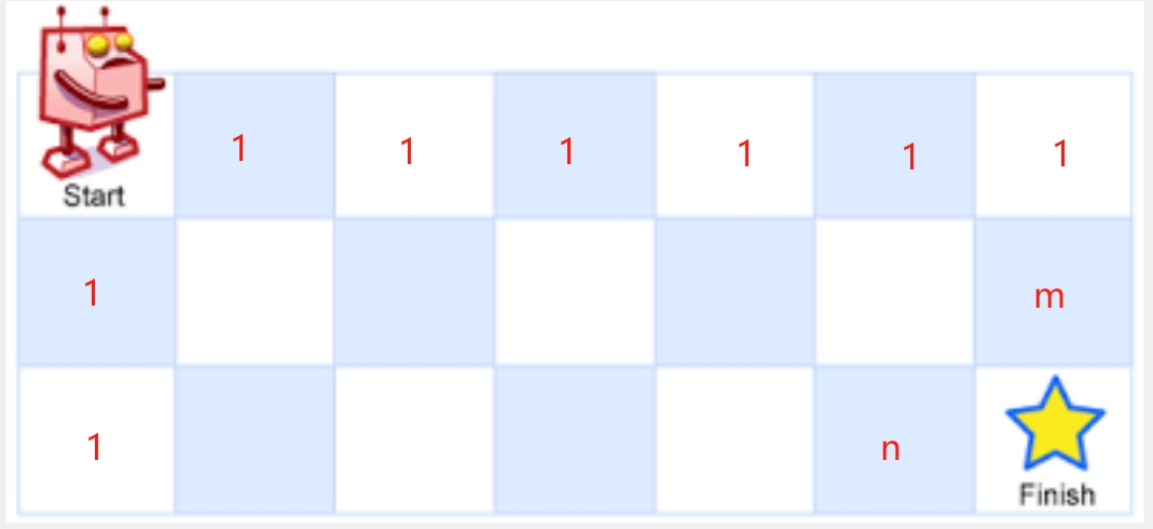
62. 不同路径

leetcode习题链接

思路分析：

其实还是爬楼梯，只不过这是二维的楼梯，每次只能向右爬和向下爬，然后还是分析到底有多少种爬法。

这里抵达最后那个位置之前，还是只有两个可能，这两种情况的爬法分别为m和n，那么到达终点就有m+n种方法。 然后初始第一行每个位置和第一列每个位置的爬法都是1，因为只能向右和向下，除非改规则，改规则的话，边界需要按照新规则重新计算，然后按照多米诺骨牌递推就ok了。




```
class Solution {
public:
    int uniquePaths(int m, int n) {
        vector<vector<int>> dp(m,vector(n,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];
    }
};
```

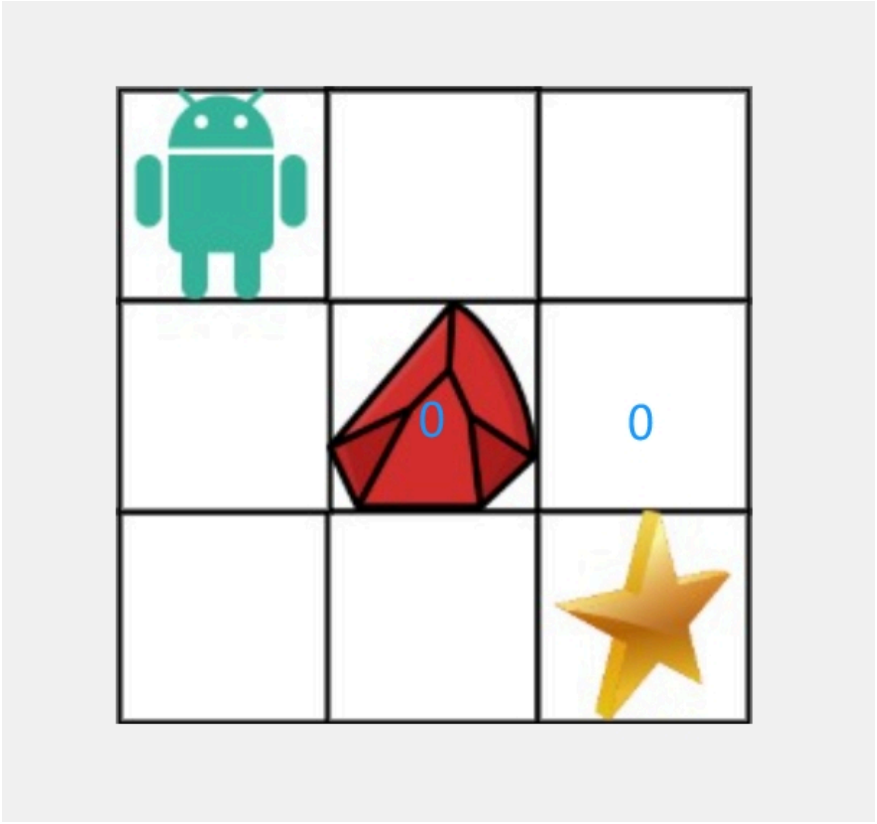
63. 不同路径II

leetcode习题链接

思路分析：

增加了障碍物，所以障碍物的地方都不能抵达。

对于第一行和第一列，障碍物出现后，后面的地方都不能到达。




```
class Solution {
public:
    int uniquePathsWithObstacles(vector<vector<int>>& obstacleGrid) {
        int m = obstacleGrid.size();
        int n = obstacleGrid[0].size();
        vector<vector<int>> dp(m,vector<int>(n,0)); // 方便后面遇到障碍物直接跳过
        //  初始化第一行和第一列
        for (int i = 0; i < m; i++) {
            if (obstacleGrid[i][0]==0) dp[i][0] = 1;
            else break;
        }
        for (int i = 0; i < n; i++) {
            if (obstacleGrid[0][i]==0) dp[0][i] = 1;
            else break;
        }
        // 递推求其他
        for (int i = 1; i < m; i++)
            for (int j = 1; j < n; j++)
                if (obstacleGrid[i][j]==0)
                    dp[i][j] = dp[i-1][j] + dp[i][j-1];
                else
                    dp[i][j] = 0;
        // 这里顺便写了打印dp数组的方法，方便调试
        for (int i = 0; i < m; i++){
            for (int j = 0; j < n; j++)
                cout << dp[i][j] << " ";
            cout << endl;
        }
        return dp[m-1][n-1];
        // 实际上在初始化第一行第一列之前就可以先判断终点是否是障碍物，直接返回0
        // 这里为了代码简洁，就不写了
    }
};
```

343. 整数拆分

leetcode习题链接

这里为什么还可以类比成一个爬楼梯问题呢，前面好歹有个楼梯或者格子，这里啥也没有，怎么办?

首先解决这个问题的关键是能够罗列出所有的拆分方法，才能知道到底怎么拆才能使得乘积最大。

实际上还是爬楼梯，比如爬到8，怎么爬，按条件分类只有两种情况，要么从6爬到8，要么从7爬到8，

然后怎么爬到6和爬到7又变成原始问题了。

而这里，比如拆分整数8，到底怎么拆呢，也可以分两种情况：

1.如果拆成2个数，可以拆成1+7，2+6，3+5，4+4

2.如果拆成3个或以上，可以对上面和式的任意一个数字进行拆分，也就是这些情况

拆加号左边的情况：拆2+6，拆3+5

拆加号右边的情况：1+拆7，2+拆6，3+拆5，4+拆4

比如拆2+6，其实就是1+1+6这一种拆法，

拆3+5，包括1+2+5，1+1+1+5两种拆法

但是这两种情况都可以被拆右边的情况覆盖，都可以由1+拆7得到，你可以试试。

所以拆成3个或者以上，只有这些情形：1+拆7，2+拆6，3+拆5，4+拆4

注：如果重复罗列了，也不要怕，只不过多计算一倍时间而已，算法复杂度没变，想不通你就

按照重复的计算，不会错。

如果设拆整数8的所有方法中，最大乘积是dp[拆8],设拆整数6的所有方法中，最大乘积是dp[拆6] ...

那么dp[拆8]就是这么求的：

1.求拆成两个数时，最大乘积是多少，max{1*7, 2*6, 3*5, 4*4} = max1;

2.求拆成3个或以上，最大乘积是多少，max 1*dp[拆7]，2*dp[拆6]，3*dp[拆5]，4*dp[拆4]} = max2;

然后求上面两种情况的最大值，也就是考虑了所有拆分方法后的最大乘积

dp[拆7]，dp[拆6], dp[拆5]，dp[拆4]这些都在多米诺骨牌dp[拆8]前面，全都已经求出来了。

```
class Solution {
public:
    int integerBreak(int n) {
        vector<int> dp(n+10);
        dp[0]=0;dp[1]=0;dp[2]=1;dp[3]=2;
        // 依次求dp[4],dp[5],..,dp[i],...,dp[n]
        for (int i = 4; i <= n; i++){
            // 罗列所有拆分情况，并不停更新最大乘积,注意没有初始化的dp[i]初始其实为0
            int m = i / 2;
            for (int j = 1; j <= m; j++){
                dp[i] = max(dp[i],max(j*(i-j),j*dp[i-j]));
            }
        }
        return dp[n];

    }
};
```

题型一小结

就是做一件事情，如果步骤太多，比如爬楼梯，前面爬那么多一长串，整数拆分，拆成2个，3个，4个，这么多拆，似乎一次性枚举所有情况不大现实，因此可以先假设做这件事前面一堆小事（事情也是同样的事情）都做完了，先把小事"折叠"起来，然后从小事的结果可以做出这件事。

虽然话是这么讲，但实际问题千变万化，大事和小事并不好定义，所以这里总结题型1只能这么总结：

1.爬楼梯，走迷宫，注意上一个状态有哪些，设上一个状态为已知即可得到递推式

2.整数拆分，就分情况拆

题型二： 背包问题

原始的背包问题简单来说就是去超市抢劫，只有一个有限容量的背包，怎么抢才能使得拿到的东西价值最高。

假设劫匪胆子小，向先拿个普通路人试试，这个路人只携带了一个小米手机（重1，价值15），一个华为平板（重3，价值20），和一个苹果电脑（重4，价值30），劫匪带着容量为4的背包。那么他最多能抢多少价值的东西?

物品	重量	价值
小米手机	1	15
华为平板	3	20
苹果电脑	4	30

背包容量：4

i \ j		0	1	2	3	4
		背包容量0	背包容量1	背包容量2	背包容量3	背包容量4
0	小米	0	15	15	15	15
1	小米+华为	0				?
2	小米+华为+苹果	0				原问题解

解这个问题的步骤其实就是劫匪不断抢劫涨经验的过程，他抢了很多次劫，

第一周，周一，他没拿背包，遇到行人带着小米手机，结果没法装，只好作罢；

第一周，周二，他带着容量为1的背包，又遇到这个只带一个小米的行人，对比重量，刚好装进去，收获价值15；

第一周，周三，他带着容量为2的背包，然后这个行人又只带一个小米手机，装进去，收获价值15；

第一周，周四和周五，他分别带着容量为3，4去抢劫，那个人又又又只带一个小米，还是只能收获价值15；

ok，第一周劫匪得到了满满的抢劫经验，他都记录在上面的表格里面了，也就是第一行，如果你问他，任意

1-4容量的背包装小米，能装的价值为多少，他能秒回，这就是他的经验

第二周，任意一天，比如周五，他带着容量4的背包，碰到了一个行人带着小米+华为，这时候劫匪的大脑开始运转了，华为是新东西，这个东西到底抢还是不抢呢：

选项A：不抢，盗亦有道，华为留给行人，剩下的物品（小米），剩余的背包容量（4），能装的最大价值，就是上面表格问号的上面记录的价值15，于是不抢华为的情况下，收获价值为15。

选项B：抢，恶人做到底，华为先塞进背包，剩下的物品（小米），剩余的背包容量（4-3=1），能装的最大价值，又能查表格第一行，为15，于是抢华为的情况下，收获价值为20+15=35。

然后劫匪作出抉择，抢才能获得最大收益，然后他抢完后，把35填在了上面问号的地方，作为下周抢劫的经验。

唯一需要注意的是，如果抢的时候，背包容量不够，那么选项B相当于无法执行，认为B选项获得的价值为0，因为这是执行不了的方案嘛。

如果设上面的表格为dp，三个物品编号分别为0，1，2。那么问号所在位置为[1,4],那么问号的数值设为dp[1] [4]，每个物品的价值设为value[0],value[1],value[2],两个选项就是这么计算的：

选项A：不抢，获得价值dp[0] [4];

选项B：抢，获得价值value[1]+ dp[0] [4-3]

dp[1] [4] = max{dp[0] [4], value[1]+ dp[0] [1]}

也就是抢最新看到的物品和不抢最新看到的物品两种方案取最大，抢的方案需要去掉重量，然后查剩余重量剩余物品能容纳多少价值，就是查上一行。不抢就是直接查上一格。递推公式如下：

dp[i] [j] = max{不抢 , 抢 }

不抢 = dp[i-1] [j]

抢 = value[j]+ dp[i-1] [背包容量-weight[j]] (当背包容量≥weight[j])

= 0 (当背包容量≤weight[j])

小结如下

1.这里示范的抢劫不是一下子就能算出来的，背包必须从0到当前容量一个一个试，抢劫的行人所拥有的物品也一定需要一个一个增加，这样才能根据上一次的抢劫经验快速获得这次的结果。

2.但是实际问题并不是这种让你求最大价值的问题，实际问题经常是，有这么多物品在，重量已知，给你一个容量为K的背包，能不能拿出一些物品刚好装满这个背包。这个问题其实可以转化为上面，就是把让每个物品的价值设置为他的重量，于是装多少重量就是装了多少价值，那么最多装多少价值，也就是最多装多少重量。

最多装多少价值就是前面的问题，我们会解，这个数值就是最多装的重量，假如刚好填满背包，那这个问题的答案就是true。如果装不满背包，那么答案就是false。

3.实际上上面二维数组可以压缩为一维，比如求第二行所有元素的时候，先求解第二行最后一个元素，dp[1] [4],

他只需要知道头上的dp[0] [4]和dp[0] [4]左边的某个值。然后求解第二行倒数第二个元素dp[1] [3]的时候，只需要知道dp[1] [3]和dp[1] [3]左边的某个值，也就是说这个时候dp[1] [4]头上的dp[0] [4]派不上用了，所以dp[1] [4]可以直接覆盖在它的头上。同理dp[1] [3]算出来后也可以覆盖在它的头上，dp[1] [2]算出来后也可以覆盖在它的头上，dp[1] [1]算出来后也可以覆盖在它的头上，dp[1] [0]算出来后也可以覆盖在它的头上。也就是第二行这个空间完全用不上，算完的结果覆盖在第一行就行了。计算第三行的时候，也是同理，计算从右向左，计算完直接覆盖即可，所以我们只需要开辟一维数组。

46. 携带研究材料

卡码网习题链接

由于不是leedcode上面的题，懒得注册账号，就不写了。读懂上面后，随便找个网上的题解，轻松读懂。

416. 分割等和子集

思路分析：

题目就是给你一堆数，然后问你这里面拿出一些数求和，能不能等于一个事先给定的数K（这题K=所有数和的一半）。

不就是给你一堆物品，问你能不能拿出一些物品，刚好装满容量为K的背包吗。

由于上面的题目没写，这里就按照前面的二维数组和一维数组的思路分别写一下题解

使用二维数组dp

```
// nums[i]既是物品重量，也是物品价值
// 背包容量为所有物品重量的一半，只能为整数，否则一定false

class Solution {
public:
    bool canPartition(vector<int>& nums) {
        // 先求背包重量
        int sum = 0;
        for (int i = 0; i < nums.size(); i++)
            sum+=nums[i];
        if (sum%2 != 0 ) return false;
        int K = sum/2;

        // 初始化二维数组dp
        int m = nums.size();
        int n = K + 1;
        vector<vector<int>> dp(m,vector<int>(n,0));  // 注意所有元素为0，包括第一列
        // 初始化第0行
        for (int j = 0; j < n; j++){
            if (nums[0] <= j) dp[0][j] = nums[0];
            else dp[0][j] = 0;
        }

        // 分别依次求第1行，第2行，到第m-1行
        for (int i = 1; i < m; i++){
            int planA,planB;
            // 求当前行，每个背包容量j最多放多少价值(重量)
            for (int j = n-1; j > 0; j--){
                planA = dp[i-1][j];
                if (j >= nums[i]) planB = nums[i] + dp[i-1][j-nums[i]];
                else planB = 0;
                dp[i][j] = max(planA,planB);
                // 这里加条判断，一旦装成功就返回
                if (dp[i][j]==K) return true;
            }
        }
        // 下面是打印dp数组，方便调试排查错误
        // cout << K << endl;
        // for (int i = 0; i < m; i++){
        //     for (int j = 0; j < n; j++)
        //         cout << dp[i][j] <<" ";
        //     cout << endl;
        // }
        return false;
```

```
    }
};
```

使用一维数组dp

```
class Solution {
public:
    bool canPartition(vector<int>& nums) {
        // 先求背包重量
        int sum = 0;
        for (int i = 0; i < nums.size(); i++)
            sum+=nums[i];
        if (sum%2 != 0 ) return false;
        int K = sum/2;

        // 初始化一维数组dp
        int m = nums.size();
        int n = K + 1;
        vector<int> dp(n,0);
        // 初始化第0行
        for (int j = 0; j < n; j++){
            if (nums[0] <= j) dp[j] = nums[0];
        }

        int planA,planB;
        // 分别依次求第1行，第2行，到第m-1行,但是每行用一维数组dp保存
        for (int i = 1; i < m; i++){
            // 求当前行，每个背包容量j最多放多少价值(重量)
            for (int j = K; j > 0; j--){
            planA = dp[j];
            if (j >= nums[i]) planB = nums[i] + dp[j-nums[i]];
            else planB = 0;
            dp[j] = max(planA,planB);
            // 这里加条判断，一旦装成功就返回
            if (dp[j]==K) return true;
        }
        }
        return false;
    }
};
```

继续改进

就是只在j>=nums[i]的地方更新，因为容量不够的地方结果和之前一样，这样planB就少一条判断

1049. 最后一块石头的重量II

思路分析：

假如石头重量分别是stones = [2,7,4,1,8,1]

想象你有一个天平，左边随便放一个2，右边随便放一个7，这样右边就重了5，于是你左边放个8，结果左边重了3，于是右边放个4，结果右边又重了1，于是左边放个1。刚好平衡，但是还剩下一块石头1。那就随便放在其中一边了，这时候最终天平两侧相差最小，为1。也就是说我的选择最终会导致天平两边差不多重，不如一开始直接计算这堆石头的总重量，2+7+4+1+8+1=23，一半就是11，12 。假如能从石头堆里面找一些石头重量和为11，那么剩下一堆就是12，反过来也一样。那么现在的问题就转为，到底从这堆石头中能不能找到一些石头重量之和为11.

假如能找到，那么最后天平两边重量差就是12-11=1，如果找不到呢？也别担心，找不到重量之和为11，那就尽可能装满，使得容量为11的袋子里装的石头越多越好。

然后这个问题是不是转化为，往袋子里装尽可能多“价值”的石头了对吗？问题迎刃而解。

是的，这类问题要么是让我从一堆数里面找一些数，让其和满足多少，要么让我从一堆数里找一些数，让其和在一定范围内尽可能大。背包问题无非就是让我干这两件事。

```
class Solution {
public:
    int lastStoneWeightII(vector<int>& stones) {
        // 先求天平左边预期容纳重量
        int sum = 0;
        for (int i = 0; i < stones.size(); i++)
            sum+=stones[i];
        int K = sum/2;

        // 初始化一维数组dp
        int m = stones.size();
        int n = K + 1;
```

```
        vector<int> dp(n,0);
        // 初始化第0行
        for (int j = 0; j < n; j++){
            if (j >= stones[0]) dp[j] = stones[0];
        }

        // 然后i从1开始遍历，求每一行
        for (int i = 1; i < m; i++){
            for (int j = K; j >= stones[i]; j--)
                dp[j] = max(dp[j],stones[i]+dp[j-stones[i]]);
        }
        int leftWeight = dp[K];
        int rightWeight = sum - dp[K];
        return rightWeight-leftWeight;

    }
};
```

494. 目标和

思路分析：

这题一上来发现没法直接套用前面的思路，就是问题并不是让我从给定的一些数里找一堆数让其和为target，而是

给这些数添加加减符号，使其表达式等于target，所以需要绕个弯子，把问题转为我能解决的问题形式。

假如手上还是有一个天平，左边放添加了+号的数，右边放添加了-号的数，目的就是让左右两边相差target。

那么左边应该放多少，右边应该放多少呢？

设左边放x，右边放y，则有x+y=sum，x-y=target，于是求得x=(sum+target)/2 右边就不用求了，用不上。

ok，问题通过上面的转换变成能不能从这堆数里找一些数和为x，但是这还没完，原始问题不是问我行不行，

而是问我有多少种方案，假如不行就是0种方案，假如行，有多少方案呢？

这样一看，似乎既不是问我这个容量背包里最多多少价值，也不是问题这个容量背包能不能刚好装这么多东西，

而是问我能有多少不同的装法。这样一来，二维dp[i] [j]里面存放的最大物品价值似乎没用了，对这题一点作用起不了，然而，真的如此吗？（注意我分析思路的时候，默认用二维数组分析）

对，真的如此，如果把dp[i] [j]当成最大价值来求确实没鸟用，但是当成最多方案数量来求，其递推求解思路基本完全一样。

先看看原来把它当成价值是怎么求dp[i] [j]的

```
dp[i] [j] = max{不抢 , 抢 }

不抢 = dp[i-1] [j]

抢 = value[j]+ dp[i-1] [背包容量-weight[j]]  (背包容量≥weight[j])
```

也就是在做决策时，考虑了两种情况，抢还是不抢，两种情况各自对应一个剩余背包装东西的子问题。

这里不妨把dp数组中的元素就当成从i个数字中，凑出和为x的方法数量，然后同样分为两种情况，拿第i个数和不拿第i个数，如果拿第i个数，那么这种情况对应的方法数量就是从剩下的i-1个数凑出和为x-nums[i]的方法数量，也就是dp[i-1] [x-nums[i]] . 如果不拿第i个数，那么就是从i-1个数中凑出和为x的方法数量，也就是dp[i-1] [x] 。

通过上面分析，问题一目了然，反正就是手上一堆东西，凑出和为某个数，考虑递推，就是对最后一件物品分两种情况考虑，这样就能分离出一模一样的子问题，问题迎刃而解。

实际上很多这种找规律的问题，比如上面爬楼梯，还有后面要讲的子序列问题，基本都是多米诺骨牌，关键看你会不会分析问题，把大问题化成具有相同结构的小问题，能化成，全都利用递推轻松解决。

但是有时候问题迷惑性很大，比如这题让你求一共多少种方法，或者最多多少价值，或者能不能装满，你肯定不知不觉绞尽脑汁去真的去数多少方法，多少价值去了，其实直接想着求最大方法，最多多少价值，可能偏向暴力搜索了，更应该关注如何凑出x，能不能把凑出x的情况罗列出来并化成子问题，每种情况可能不是最优解，但是最优解一定是从这些情况中找出来的，所以重点是把定义域划分，这里的定义域就是凑x，划分为拿nums[i]和不拿nums[i]再凑两种情况，值域设为f或者dp，然后求f（情况1），f（情况2），然后从值域中挑选最优解。

抽象为：

做一件事，求这件事获得的某种最大值（方法最大值，价值最大值）。

做这件事设为定义域，要求的值设为值域。把做这件事给分解，分情况，然后套上f求最优就是递推公式。

第一步一定不要关心要求什么，而是求这个东西需要做什么事，怎么做，怎么分情况做。

再比如上面爬楼梯，假如爬每个楼梯不是给你设个体力值，给你设个别的东西，比如奖励某些东西，

给个奖品数组=【小米平板，小米su7，小米手环，华为pad，辣条】

这就是爬每个台阶给的奖励，问你爬到顶最多获得多少价值的东西？

重点是爬楼梯怎么爬，而不是一上来就算爬的过程中能拿多少钱！一开始一定要无视他让你计算的东西。

等你分析出爬楼梯怎么爬，分哪几种情况爬之后，再去计算能拿多少钱。

当然，最后再来说一句，这题给你弄符号，搞表达式，问题转化过去这步很难想到，因为题目让你加正负号很迷惑人，类似这种需要见多识广了，第一次撞见，即便你背包爬楼梯再熟，也不是很容易想到这都能挂钩。

```
// left + right = sum;  left - right = target

class Solution {
public:
    int findTargetSumWays(vector<int>& nums, int target) {
        // 先求和
        int sum = 0;
        for (int i = 0; i < nums.size(); i++)
            sum += nums[i];

        // 特殊情况1：全部求和也达不到target或者-target
        if (target > sum || target < -sum) return 0;

        // 特殊情况2：left不为整数
        if ((sum+target)%2!=0) return 0;

         // 特殊情况3：去掉数字0之后，只剩1个数字，无法分成left-right形式
        int counts = 0;
        for (int i = 0; i < nums.size(); i++)
            if (nums[i] == 0) counts++;
        if (nums.size()-counts == 1){
            for (int i = 0; i < nums.size(); i++){
                if (nums[i] == target || nums[i] == -target) return pow(2,counts);
            }
            return 0;
        }

        // 正常情况
        // 先把0搬走
        if (counts != 0){
            for (int slow = 0, fast = 0; fast < nums.size();){
            if (nums[fast]!=0) nums[slow++] = nums[fast++];
            else fast++;
        }
        }

        int left = (sum + target)/2;
        // 初始化dp数组及第0行
        int m = nums.size() - counts;
        int n = left + 1;
        vector<int> dp(n,0);
        for (int j = 0; j < n; j++){
            // nums[0] 能凑出 数值j 的话，就是一种方法
            if (j == nums[0]) dp[j] = 1;
        }

        // 递推求解其他行
        for (int i = 1; i < m; i++){
            int planA,planB;
            for (int j = left; j >= nums[i]; j--){
                planA = dp[j];  // 不使用nums[i]凑j，那么就用剩下的凑j
                // 使用nums[i]凑j
                if (j == nums[i]) planB = 1; // 刚好这一个数nums[i]就凑成了j
                else planB = dp[j-nums[i]];  // 剩下的凑j-nums[i]的方法数量
                dp[j] = planA+planB;
            }

        }
        cout << dp[left] << endl;
        cout << counts << endl;
        return dp[left]*(pow(2,counts));

    }
};
```

注意上面的代码看上去比较繁琐，是因为有一些特例必须单独处理。

正常处理的模式中，不要碰到0，因为0既不是正数也不是负数，情况特殊。

然后就是一些提前返回的情况，这些情况都凑不成，直接失败。

说实话，这题处理特殊情况浪费了不少时间，还是挺费事的。

518. 零钱兑换II

leetcode习题链接

思路分析：

这题就是把上题稍微改了一下，上题是拿一些数凑left，这题是拿一些硬币凑amount，区别主要有两个，

一个是硬币不存在0面值的，不需要再考虑0了。另一个是这里面的每种硬币都有无限个，随便拿。

另外，这道题的方法其实也能用来解决前面那个整数拆分问题，把那题改成用1~n-1凑n有多少种方法，就不要

考虑拆的几种情况了。言归正传，再回到本题，本题的递推公式和上题基本一样，就是考虑硬币i和不考虑硬币i两种情况的方法之和。他们的区别就是初始条件不一样，也就是多米诺骨牌的前几张牌不一样。

```
class Solution {
public:

    void printlog(const vector<int> & dp){
        for (int i = 0; i < dp.size(); i++)
            cout << dp[i] << " ";
        cout << endl;
    }

    int change(int amount, vector<int>& coins) {
        // 初始化dp数组
        int m = coins.size();
        int n = amount + 1;
        vector<int> dp(n,0);

        // 初始化第0行,也就是用第0个硬币凑各种数值
        // 由于只有一个硬币,因此能凑成的一定是它的倍数,并且只有一种方法
        for (int j = coins[0]; j < n; j++){
            if (j%coins[0]==0) dp[j] = 1;
        }
        // 第0列，也就是amount = 0的情况
        // 如果所凑金额为0，就不要硬币，就这一种方法
        dp[0] = 1;

        // printlog(dp);

        // 开始从第1行开始递推
        for (int i = 1; i < m; i++){
            for (int j = amount; j >= 0; j--){
                // 不使用硬币i
                int planA = dp[j];
                // 考虑使用硬币i,那么就考虑用几枚
                int planB = 0;  // 默认硬币i超过j,无法使用
                if (j >= coins[i]){
                    int L = j / coins[i];
                    for (int k = 1; k <= L; k++ ){
                        planB+=dp[j-k*coins[i]];   // 分别用1枚,2枚,...,L枚硬币i
                }
                }
                dp[j] = planA + planB;
            }
            // printlog(dp);
        }

        return dp[amount];
    }
};
```

实际上上面第i行可以理解为分别使用0枚，1枚，2枚，...，L枚硬币i 的情形

322. 零钱兑换 I

leetcode习题链接

思路分析：

这题就是求第i行，凑amount=j，需要最少多少枚硬币，思路和前面还是一样，就是分别使用0枚，1枚，2枚，...，L枚硬币i 的情形，然后剩余部分怎么凑，需要最少多少，查看上一行就能得到，最后求最小枚数即可

第0行和第0列的初始化尤为重要，初始化错了，后面全错。

第0行就是用coins[0]凑成各种amount，如果凑的成，看看凑了多少枚，如果凑不成，使用INT_MAX，表示失败，方便后面的情况查表的时候知道INT_MAX是凑不成，求最小值只在凑的成的情形里面求。

第0列就是凑成0元需要多少枚硬币，那就是0枚呗。

```
 int coinChange(vector<int>& coins, int amount) {
        // 初始化dp数组
        int m = coins.size();
        int n = amount + 1;
        vector<int> dp(n,0);

        // 初始化第0行
        for (int j = 0; j < n; j++){
            if (j%coins[0]==0) dp[j] = j/coins[0];
            else dp[j] = INT_MAX;
        }
        // 第0列
        dp[0] = 0;

        // 开始从第1行开始递推
        for (int i = 1; i < m; i++){
            // 只有j >= coins[i]是,硬币i才能用得上,用不上的话,上一行就是最少使用硬币数
            for (int j = amount; j >= coins[i]; j--){
                // 不使用硬币i
                int planA = dp[j];
                // 考虑使用硬币i,那么就考虑用几枚
                if (j >= coins[i]){
                    int L = j / coins[i];
                     // 分别用1枚,2枚,...,L枚硬币i
                    for (int k = 1; k <= L; k++ ){
                        // 能够凑成才更新最少硬币数量
                        if (dp[j-k*coins[i]]!=INT_MAX)
                            planA=min(planA,k+dp[j-k*coins[i]]);
                    }
                }
                dp[j] = planA;
            }
        }
        if (dp[amount]==INT_MAX) return -1;
        return dp[amount];
    }
};
```

题型二小结

就是给一些数，让你装进指定大小的背包里：

1.最多装多少钱

2.刚好装满，有多少种装法

3.刚好装满，最少需要多少个

dp【j】 = 考虑{ dp【j不减】，dp【j 减 1枚nums[i]】，dp【j 减 2枚nums[i]】，dp【j 减 L枚nums[i]】}这些情况，并且这些情况都可以从上一行获取

题型三： 子序列问题

首先借鉴上个题型的经验，对于本题型，给定一个长度L的数组，找出满足某某条件的子数组或者子序列，然后这个某某条件完全不用管，先把在长度为i的数组中，找子数组或子序列这件事的过程想明白，到底怎么从长度i的数组中找出所有子数组或者子序列。然后再去想想题目让我做什么，以此来初始化边界条件，也就是多米诺骨牌的前几张牌。

这里我们把dp数组的长度当成背包容量，从0开始慢慢扩成最终的长度的。然后子序列的头或者尾巴的下标当成前面的物品i。

上面讲对于熟悉的人来说，一点就通，对于还是模模糊糊的人来说，听起来有些抽象，下面结合具体的题目讲解。

另外，子序列可以不连续，子数组必须连续，比如[1,2,3,4,5,6]中[1,3,5]就是子序列，[2，3，4]就是子数组

300. 最长递增子序列

leetcode习题链接

5	2	1	3

原数组长度为4，从中找一个最长递增子序列

	以5结尾的子序列	以2结尾的子序列	以1结尾的子序列	以3结尾的子序列
[5, 2, 1, 3]	a	b	c	

思路分析：

原问题是从数组[5, 2, 1, 3]找最长递增子序列，先别管最长递增这个条件，先找出所有子序列，然后里面求最长不就行了。ok，接下来就是找出所有子序列，那么这个数组总共包含如下的子序列：

以3结尾的子序列，以1结尾的子序列，以2结尾的子序列，以5结尾的子序列

ok，现在再来求解最长递增的那个子序列是谁，现在假设：

以1结尾的最长递增子序列长度为c，以2结尾的最长递增子序列长度为b，以1结尾的最长递增子序列长度为a，

然后怎么获得以5结尾的最长子序列长度？

其实就是分别拿前面长度为a，b，c的子序列尾巴添加一个3结尾，看看能不能组成更长的递增子序列，也就是拿3和前面三个子序列尾巴那个最大的数比较，如果大于，就构成了更长的序列，如果小于，就pass。假如求完为d。

然后这4种情况的子序列长度都知道了，a，b，c，d，然后再求最长，是不是就求出了所有子序列中最长递增的那个。

递推式如下：

dp[3] = max {以nums[3]结尾最长子序列，以nums[2]结尾最长子序列，以nums[1]结尾最长子序列，以nums[0]结尾最长子序列}

```
class Solution {
public:
    int lengthOfLIS(vector<int>& nums) {
        int n = nums.size();
        if (n==0) return 0;
        vector<int> dp(n,1);

        for (int i = 1; i < n; i++){
            for (int j = 0; j < i; j++){
                if (nums[i] > nums[j]) dp[i] = max(dp[i],dp[j]+1);
            }
        }

        int maxLen = dp[0];
        for (int i = 1; i < n; i++)
            maxLen = max(maxLen,dp[i]);
        return maxLen;
    }
};
```

674. 最长连续递增子序列

就是最长递增子数组，这样以nums[i]结尾的子序列要么只有它自己，要么和前面nums[i-1]结尾的子数组拼在一起，不能与前面拼在一起，不然中间不连续。

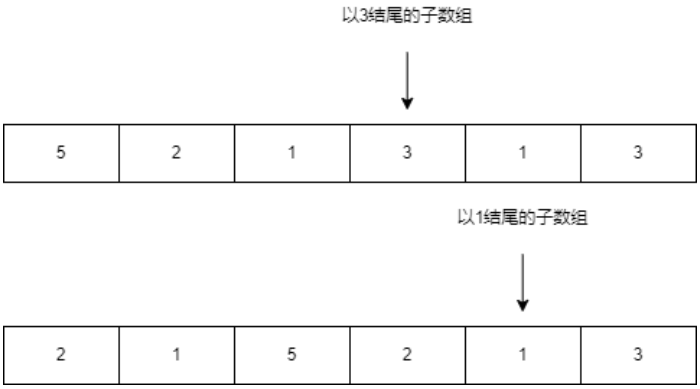
```
class Solution {
public:
    int findLengthOfLCIS(vector<int>& nums) {
        int n = nums.size();
        if (n==0) return 0;
        vector<int> dp(n,1);

        for (int i = 1; i < n; i++){
            if (nums[i] > nums[i-1]) dp[i] =dp[i-1]+1;
        }

        int maxLen = dp[0];
        for (int i = 1; i < n; i++)
            maxLen = max(maxLen,dp[i]);
        return maxLen;
    }
};
```

718. 最长公共子数组

leetcode习题链接



这是两个子序列问题，对于第一个数组，用i遍历以nums1[i]结尾的子数组，假如移动到i头上，固定下来，

接着用j遍历第二个数组，假如j到了j头上，此时如何计算i结尾的数组和j结尾的数组最长重复长度是多少呢?

由于数组是连续的，如果nums1[i]==nums2[j]，看看他们前面那个元素结尾的子数组最长重复长度是多少，

然后+1即可。如果不相等，为0，不重复。

后面还有题是求最长公共子序列，那么i和j代表子序列结尾的话，情况就变得极其复杂，如果i和j代表数组结尾，

那么就容易分析了。包括后面编辑距离，都是以数组长度为i，j，而不是以子序列结尾。

大多数情况，连续子序列用结尾，不连续情况用长度。

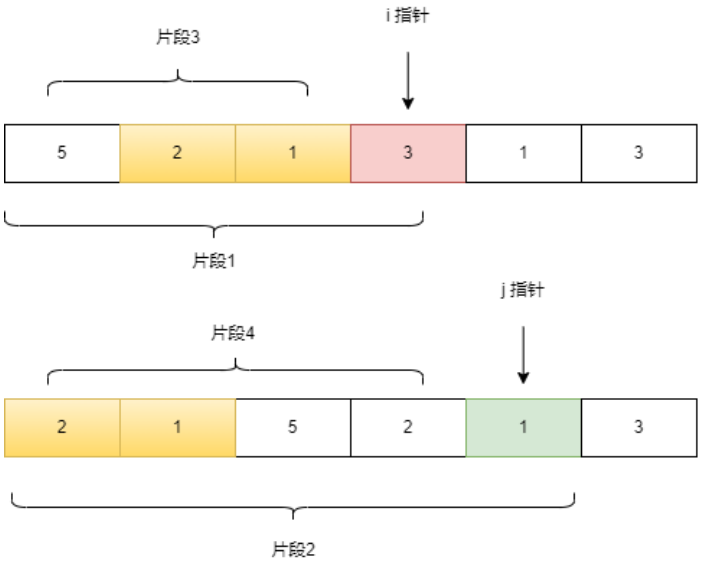
```
class Solution {
public:
    int findLength(vector<int>& nums1, vector<int>& nums2) {
        int m = nums1.size();
        int n = nums2.size();
        if(m==0 || n==0) return 0;
        vector< vector<int> > dp(m,vector(n,0));

        // 先求第0行,i指向0,j挨个遍历
        for (int j = 0; j < n; j++){
            if (nums1[0]==nums2[j]) dp[0][j]=1;
        }
        // 再求第0列,j指向0,i挨个遍历
        for (int i = 0; i < m; i++){
            if (nums1[i]==nums2[0]) dp[i][0]=1;
        }

        // 然后i分别指向1,2,...,m-1
        for (int i = 1; i < m; i++){
            for (int j = 1; j < n; j++){
                if (nums1[i]==nums2[j]) dp[i][j] = dp[i-1][j-1] + 1;
                else dp[i][j] = 0;  // 这句可以省略,因为已经初始化为0了
            }
        }
        int maxLen = 0;
        for (int i = 0; i < m; i++)
            for (int j = 0; j < n; j++)
                maxLen = max(maxLen,dp[i][j]);
        return maxLen;

    }
};
```

1143. 最长公共子序列



思路分析：

定义两个指针i和j，i指向text1的一段长度数组的最后一个元素，j指向text2的一段长度数组的最后一个元素，

i和j分别指定了两个数组（片段1和片段2），然后dp[i] [j] 就是片段1和片段2的最长公共子序列长度，dp[i-1] [j] 就是片段3和片段2的最长公共子序列长度，dp[i] [j-1] 就是片段1和片段4的最长公共子序列长度，dp[i-1] [j-1] 就是片段3和片段4的最长公共子序列长度。

如果这两个片段的最后一个元素相等，比如上图中j指针下面不是1，是3，那么这两个片段最后一个元素是重合的，我们再查查前面重合了多少，也就是片段3和片段4重合了多少，一查看，重合了2个元素，因此总共片段1和片段2重合了3个元素，也就是dp[i] [j] = dp[i-1] [j-1] + 1

如果两段数组最后一个元素不相等，那么我们就比较片段1和片段4的重合子序列长度，以及片段3和片段2的重合子序列长度，为什么比较这两对就行了呢，因为片段1和片段2最后一个元素对不上，只能删掉一个元素再比对重合的子序列。另外就是，片段3和片段4的重合子序列（[2, 1]）一定包含在上面两种情况中，所以这两个片段不需要再看了。

```
class Solution {
public:
    void printlog(const  vector< vector<int> > & dp){
        for (int i = 0; i < dp.size(); i++){
            for (int j = 0; j < dp[0].size(); j++){
                cout << dp[i][j] << " ";
            }
            cout << endl;
        }
    }
    int longestCommonSubsequence(string text1, string text2) {
        int m = text1.size();
        int n = text2.size();
        if(m==0 || n==0) return 0;
        vector< vector<int> > dp(m, vector<int>(n, 0));

        // 先求第0行,i指向0,j挨个遍历
        for (int j = 0; j < n; j++){
            if (text1[0]==text2[j]) {
                for (int k = j; k < n; k++)
                    dp[0][k]=1;
                break;
            }
        }
        // 再求第0列,j指向0,i挨个遍历
        for (int i = 0; i < m; i++){
            if (text1[i]==text2[0]) {
                for (int k = i; k < m; k++)
                    dp[k][0]=1;
                break;
            }
        }
        // 然后i分别指向1,2,...,m-1
        for (int i = 1; i < m; i++){
            for (int j = 1; j < n; j++){
                if (text1[i]==text2[j]) {
                    dp[i][j] = dp[i-1][j-1] + 1; // 两个指针同时向前移动一格
                }
                else {
                    dp[i][j] = max(dp[i-1][j], dp[i][j-1]); // 只移动一个指针
```

```
                }
            }
        }

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

53. 最大子数组和

leetcode习题链接

思路分析：

每个数字结尾的最大子数组用下标j遍历，假如j-1前面的每个元素结尾的最大子数组都已知了，如何求以下标j为结尾的最大子数组和，既然以j结尾，要么只有nums[j]一个元素，要么和前面j-1结尾的最大和子数组组合在一起，

然后这两种情况取最大就是以j结尾的最大子数组了。因为是连续的，所以只要考虑前面一个元素即可。

这个是单个序列的问题，因而需要遍历的就是满足条件的所有子序列而已，遍历一次即可，使用一维dp数组。

```
class Solution {
public:
    int maxSubArray(vector<int>& nums) {
        int n = nums.size();
        vector<int> dp(n,0);
        dp[0] = nums[0];
        for (int j = 1; j < n; j++){
            dp[j] = max(dp[j-1]+nums[j],nums[j]);
        }
        int maxSum = INT_MIN;
        for (int j = 0; j < n; j++){
            maxSum = max(maxSum,dp[j]);
        }
        return maxSum;
    }
};
```

72. 编辑距离

求将ros变成horse所需要最少的操作

	null	h	o	r	s	e
null	0	1	2	3	4	5
r	1		[null，r] -> [null，h，o]	[null，r] -> [null，h，o，r]		
o	2		[null，r，o] -> [null，h，o]	?		
s	3					

第0行 将null变成null，不需要操作，操作数为0

第0行 将null变成[null，h]，插入一个h，操作数为1

第0行 将null变成[null，h，o]，插入一个h，插入一个o，操作数为2

第0行 后面不难类推，分别得到将null变成各个字符串的操作数

第0列 将null变成null，不需要操作，操作数为0

第0列 将[null，r]变成null，删除r，操作数为1

第0列 后面类推

以上面将字符串“ros”变成字符串“horse”所需要的最少操作数为例，这里还是采用两个序列问题的动态规划思路，这两个长的不好处理，用i和j截取短的片段，把片段的问题先解决。

比如上面第0行需要解决的问题，就是把null（空串）分别变成null，[null，h]，[null，h，o]，...，[null，h，o，r，s，e]所需要的最少操作数，然后第1行需要解决的问题就是把[null，r]分别变成上面这些字符串所需的最少操作数。

既然明白了这个表格的含义，不难求出第0行和第0列的dp值。第0行和第0列的其中一个字符串为空串，因此比较好求，接下来看看[null，r，o]变成[null，h，o，r]所需要的操作数，也就是图中？所在位置需要填的数。

我们考虑如下2种修改方案，取3种方案操作数的最小值即可。

方案1：[null，r，o] --> [null，h，o]-->[null，h，o，r]

由于 [null，r，o] --> [null，h，o] 这个就是？左边干的事情，这个需要的最少操作数已知，因此只需要知道

[null，h，o]-->[null，h，o，r] 所需的操作数即可，这里执行了插入r的操作，因此把？左边操作数+1就是这个方案所需的操作数。

方案2：[null，r，o] --> [null，r]-->[null，h，o，r] 这个就是将[null，r，o] 执行了删除操作，变成了 [null，r]，

然后[null，r]-->[null，h，o，r] 就是问号上方的操作数，因此这种方案就是把上方的操作数+1 。

最终取两个方案的最小值即可。

方案3：[null，r，o] -->[null，h，o，o] -->[null，h，o，r]，先不看[null，r，o]和[null，h，o，r]的最后一个字符o和r，先把他们前面的字符变成一样，这步需要的操作数就是问号左上方的操作数，然后再把最后一个字符使用替换操作。因此这个方案的操作数就是左上方操作数+1。

最终取三个方案的最小值即可。

然后我们再看一个例子，如何求 [null，r，o] --> [null，h，o]，由于这两个字符串最后一个字母相同，因此只需要考虑 [null，r] --> [null，h]所需要的操作数，最后一个字符不需要替换。另外也不需要考虑方案1和方案2中的插入和删除操作，都是不必要的，因为最后一个字符已经一样了，不需要任何插入删除替换操作。

因此，表格中任意一格，都可以由它的上方，左方，左上方计算得到，这就是递推公式。

```
class Solution {
public:
    void printlog(const vector<vector<int>> & dp){
        for (int i = 0; i < dp.size(); i++){
            for (int j = 0; j < dp[0].size(); j++)
                cout << dp[i][j] << " ";
            cout << endl;
        }
    }
    int minDistance(string word1, string word2) {
        int m = word1.size()+1;
        int n = word2.size()+1;
        vector<vector<int>> dp(m,vector<int>(n,0));
        // 初始化第0行
        for (int j = 0; j < n; j++)
            dp[0][j] = j;
        // 初始化第0列
        for (int i = 0; i < m; i++)
            dp[i][0] = i;

        // 递推计算dp
        for (int i = 1; i < m; i++){
            for (int j = 1; j < n; j++){
                if (word1[i-1] == word2[j-1]) dp[i][j] = dp[i-1][j-1];
                else {
                    dp[i][j] = min(1+dp[i-1][j],1+dp[i][j-1]);
                    dp[i][j] = min(dp[i][j],1+dp[i-1][j-1]);
                }
            }
        }
        printlog(dp);
        return dp[m-1][n-1];
    }
};
```

需要注意的是，由于添加了null，所以判断式是if (word1[i-1] == word2[j-1])，而不是if (word1[i] == word2[j])

115. 不同的子序列

leetcode习题链接

思路分析：

这题就是编辑距离，问s里面包含多少种t，也就是t可以使用多少删除方法变成s。

递推式是什么样的：

一、假如i和j位置的两个字符相等，那么把s[0:i]变成t[0:j]有两种方案：

方案1：考虑使用s[0:i]的最后一个字符，这时候把s[0:i]变成t[0:j]其实就是把s[0:i-1]变成t[0:j-1]

方案2：不使用s[0:i]的最后一个字符，这时候就是把s[0:i-1]变成t[0:j] 。

上面两种其实就是分别考虑s[0:i]中（必须包含最后一个字符）包含多少种t[0:j]，以及s[0:i-1]中包含多少种t[0:j]。

也就是表格中当前元素的上方和左上方。

二、假如i和j位置的两个字符不相等，那么只需要考虑把s[0:i-1]变成t[0:j-1]

初始条件是什么样的：

第0行，就是问你null里面包含多少种[null]，[null，a]，....

很简单，就是1，0，0，0，null里面无论如何也不会包含更长的字符串了

第0列，就是问你[null]，[null，h]，...里面分别包含多少种[null]，很显然，都只包含一种null，也就是null仅出现一次

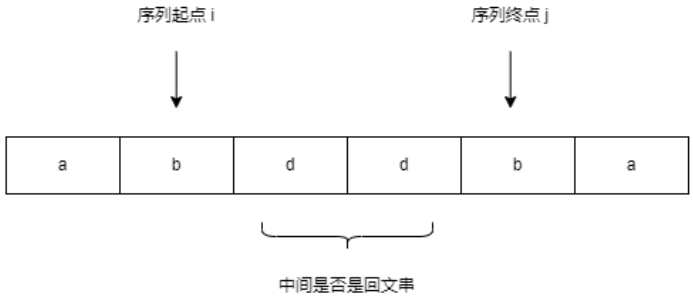
```
// t在s里面，也就是s经过删除变成t,或者说s里面包含多少种t
class Solution {
public:
    int numDistinct(string s, string t) {
        int m = s.size()+1;
        int n = t.size()+1;
        vector<vector<int>> dp(m,vector<int>(n,0));
        // 初始化第0行,上面已经默认初始化了
        // 初始化第0列
        for (int i = 0; i < m; i++)
            dp[i][0] = 1;

        // 递推计算dp
        for (int i = 1; i < m; i++){
            for (int j = 1; j < n; j++){
                if (s[i-1] == t[j-1]) dp[i][j] = dp[i-1][j-1] + dp[i-1][j];
                else {
                    dp[i][j] = dp[i-1][j];
                }
            }
        }
        return dp[m-1][n-1];
    }
};
```

上面的代码在leetcode中会出现数值溢出，按照题目要求取模

```
// t在s里面，也就是s经过删除变成t,或者说s里面包含多少种t
class Solution {
public:
    int numDistinct(string s, string t) {
        int m = s.size() + 1;
        int n = t.size() + 1;
        vector<vector<long long>> dp(m, vector<long long>(n, 0));
        // 初始化第0行,上面已经默认初始化了
        // 初始化第0列
        for (int i = 0; i < m; i++)
            dp[i][0] = 1;
        long long M = pow(10,9)+7;
        // 递推计算dp
        for (int i = 1; i < m; i++) {
            for (int j = 1; j < n; j++) {
                if (s[i - 1] == t[j - 1])
                    dp[i][j] = (dp[i - 1][j - 1] + dp[i - 1][j])%M;
                else {
                    dp[i][j] = dp[i - 1][j]%M;
                }
            }
        }
        return dp[m - 1][n - 1];
    }
};
```

647. 回文子串



	a	b	d	d	b	a
a	1	0				
b		1	0			
d			1	1		
d				1	0	
b					1	0
a						1

思路分析：

回文串就是正着读和倒着读完全一样的对称字符串，从一个字符串中找到所有回文串，实际就是遍历所有子串，然后一个一个判断是不是回文串，我们用i表示回文串的开头，j表示回文串的结尾，（j 一定 大于等于 i），这样就能用两层for循环遍历所有的子字符串，但是判断这个子字符串是不是回文串，我们不需要再用for循环来判断了。那么如何判断？我们把所有子串的状态信息用上面的dp表格记录，主要分为下面两种情况：

情形一：s[i] == s[j] ，这时候只要判断中间那串是不是回文串就知道这个串是不是回文串了，假设dp[i] [j]表示的就是s[i:j]是不是回文串，那么这种情况，只要看s[i-1 : j-1]是不是回文串，也就是dp[i+1] [j-1]的值。

情形二：s[i] ！ = s[j] ，那s[i:j]一定不是回文串，令dp[i] [j] = 0即可。

也就是求dp [i] [j] 需要依赖 dp[i+1] [j-1] ，也就是左下方的值，我们首先把子序列长度为1(i和j相等)以及长度为2（j比i大1）的情形求出来，长度为1的一定是回文串，长度为2的，两个字符相等就是回文串，把这两条对角线初始化之后，我们从下往上，从左往右遍历即可。

这题题目要我们求有多少回文串，我们把dp数组中所有的1求和即可，也可以弄个全局变量在上面情形一里面进行累加。

如果题目要我们求最长的回文串，我们在情形一里求j - i的值即可，然后弄个全局变量maxLen进行更新。

但是注意，下题要我们求的最长回文子序列，不是子串，所以用的不是这张记录子串的表格。

```
class Solution {
public:
    int countSubstrings(string s) {
        int n = s.size();
        vector<vector<int>> dp(n,vector<int>(n,0));

        int result = 0;

        // 初始化两个对角线
        for (int i = 0; i < n; i++) {
            dp[i][i] = 1;
            result++;
            if (s[i] == s[i+1]) {
                dp[i][i+1] = 1;
                result++;
            }
        }

        // 开始判断其他子序列是否是回文串
        for (int i = n-1; i >= 0; i--){
            for (int j = i+2; j < n; j++){
                if (s[i]==s[j] && dp[i+1][j-1]) {
                    dp[i][j] = 1;
                    result++;
```

```
                }
            }
        }
        return result;
    }
};
```

516. 最长回文子序列

leetcode习题链接

既然这里是回文子序列，那么我们的i和j设置为一个子数组的首尾，我们要找出s[0 : n-1]中的最长回文子序列，就需要从数组s[i : j] 中找到最长回文子序列，然后递推过去。如何求解s[i : j]中的最长回文子序列呢？分两种情况：

情况一：s[i] == s[j] , 那么只要看s[i+1 : j-1]的最长回文子序列长度，然后+2就是它的最长回文子序列长度

情况二：s[i] != s[j] , 那么在s[i : j-1]和s[i+1 : j]两个数组中找最长回文子序列，取最长即可

```
class Solution {
public:
    int longestPalindromeSubseq(string s) {
        int n = s.size();
        vector<vector<int>> dp(n,vector<int>(n,1));

        // 初始化两个对角线,第一条对角线上面已初始化为1
        // 对于不相等的情况,也至少有长度为1的回文子序列
        for (int i = 0; i < n; i++) {
            if (s[i] == s[i+1]) {
                dp[i][i+1] = 2;
            }
        }

        // 开始计算其他子数组中的最长回文子序列
        for (int i = n-1; i >= 0; i--){
            for (int j = i+2; j < n; j++){
                if (s[i]==s[j]) dp[i][j] = 2 + dp[i+1][j-1];
                else dp[i][j] = max(dp[i+1][j],dp[i][j-1]);
            }
        }
        return dp[0][n-1]; // i指向首,j指向尾
    }
};
```