动态规划的陨落篇

动态规划的题目相对较难而且耗时，所以面试的时候遇到的几率并不大，但也不是没有。所以该篇将分别介绍动态规划是什么，动态规划的主要组成要素，什么时候使用动态规划以及动态规划题目的分类，并将针对不同类型使用不同的例题进行讲解。

动态规划的通常思路：

动态规划是一种算法思路（注意这里不要和递归混淆，事实上递归和迭代只是两种不同的实现方法，并不是算法），用一句话来总结就是，动态规划是利用存储历史信息使得未来需要历史信息时不需要重新计算，从而达到降低时间复杂度，用空间复杂度换取时间复杂度目的方法。我个人喜欢把动态规划分为以下几步：

1）确定递推量。这一步需要确定递推过程中要保留的历史信息数量和具体含义，同时也会定下动态规划的维度；

2）推导递推式。根据确定的递推量，得到如何利用存储的历史信息在有效时间（通常是常量或者线性时间）内得到当前的信息结果；

3）计算初始条件。有了递推式之后，我们只需要计算初始条件，就可以根据递推式得到我们想要的结果了。通常初始条件都是比较简单的情况，一般来说直接赋值即可；

4）考虑存储历史信息的空间维度（可选）。这一步是基于对算法优化的考虑，一般来说几维动态规划我们就用几维的存储空间是肯定可以实现的。但是有时我们对于历史信息的要求不高，比如这一步只需要用到上一步的历史信息，而不需要更早的了，那么我们可以只存储每一步的历史信息，每步覆盖上一步的信息，这样便可以少一维的存储空间，从而优化算法的空间复杂度。

动态规划的时间复杂度是 O（（维度）×（每步获取当前值所用的时间复杂度））。基本上按照上面的思路，动态规划的题目都可以解决，不过最难的一般是在确定递推量，一个好的递推量可以使得动态规划的时间复杂度尽可能的低。

什么时候考虑使用动态规划：

1. 当题目让我们找 Maximum 或者 Minimum 的时候。
2. 当题目让我们判断 Yes 或者 No 的时候。
3. 当题目让我们 count all possible solutions 的时候。

动态规划最常见题目的分类：

1. Matrix Dynamic Programming
2. Sequence Dynamic Programming
3. Two Sequences Dynamic Programming
4. Back Pack

Part 1 – Matrix Dynamic Programming

这类题目是动态规划当中相对简单的，当我们明确了4个主要元素，问题就会迎刃而解。

1）确定递推量。res[x][y]一般表示从起点走到x, y时的状态；

2）推导递推式。一般可以研究最后一步应该如何走；

3）计算初始条件。起点/终点；

4）考虑存储历史信息的空间维度（可选）。二维动态规划使用二维空间肯定可以解决。但这类题往往对于历史信息的要求不高，这一层只需要用到上一层的历史信息，所以我们可以使用一维空间来解决即可，优化空间复杂度使其降低一个纬度。

在LeetCode中关于Matrix DP的题目有：

1. Unique Paths
2. Unique Paths II
3. Minimum Path Sum
4. Dungeon Game

1. Unique Paths, res[x][y]代表起点到该点共有的路径数。二维空间稳定，一维空间最优。

```
// 2D-Dynamic Programming-Stable
public int uniquePaths(int m, int n) {
    if(m <= 0 || n <= 0){
        return 0;
    }
    int[][] res = new int[m][n];
    for(int i = 0; i < m; i++){
        res[i][0] = 1;
    }
    for(int j = 0; j < n; j++){
        res[0][j] = 1;
```

```
        }
        for(int i = 1; i < m; i++){
            for(int j = 1; j < n; j++){
                res[i][j] = res[i - 1][j] + res[i][j - 1];
            }
        }
        return res[m - 1][n - 1];
    }

    // 1D-Dynamic Programming-Optimal
    public int uniquePaths(int m, int n) {
        if(m <= 0 || n <= 0){
            return 0;
        }
        int[] res = new int[n];
        res[0] = 1;
        for(int i = 0; i < m; i++){
            for(int j = 1; j < n; j++){
                res[j] += res[j - 1];
            }
        }
        return res[n - 1];
    }

    // Combination Formula-Special
    public int uniquePaths(int m, int n) {
        if(m <= 0 || n <= 0){
            return 0;
        }
        int min = m < n ? m - 1 : n - 1;
        int max = m > n ? m - 1 : n - 1;
        double numerator = 1.0;
        double denominator = 1.0;
        for(int i = 1; i <= min; i++){
            numerator *= i + max;
            denominator *= i;
        }
        return (int)(numerator / denominator);
    }
```

2. Unique Paths II，res[x][y]代表起点到该点共有的路径数。二维空间稳定，一维空间最优。注意对i == 0这行，j == 0这列的处理即可。

```
    // 2D-Dynamic Programming-Stable
    public int uniquePathsWithObstacles(int[][] obstacleGrid) {
        if(obstacleGrid == null || obstacleGrid.length == 0 || obstacleGrid[0].length ==
0){
            return 0;
        }
        int m = obstacleGrid.length;
        int n = obstacleGrid[0].length;
        int[][] res = new int[m][n];
        for(int i = 0; i < m; i++){
            if(obstacleGrid[i][0] == 1){
                break;
            }
            res[i][0] = 1;
        }
        for(int j = 0; j < n; j++){
            if(obstacleGrid[0][j] == 1){
                break;
            }
            res[0][j] = 1;
        }
        for(int i = 1; i < m; i++){
            for(int j = 1; j < n; j++){
                if(obstacleGrid[i][j] == 1){
                    res[i][j] = 0;
                } else {
                    res[i][j] = res[i - 1][j] + res[i][j - 1];
                }
            }
        }
        return res[m - 1][n - 1];
    }

    // 1D-Dynamic Programming-Optimal
    public int uniquePathsWithObstacles(int[][] obstacleGrid) {
        if(obstacleGrid == null || obstacleGrid.length == 0 || obstacleGrid[0].length ==
0){
```

```
            return 0;
        }
        int[] res = new int[obstacleGrid[0].length];
        res[0] = 1;
        for(int i = 0; i < obstacleGrid.length; i++){
            for(int j = 0; j < obstacleGrid[0].length; j++){
                if(obstacleGrid[i][j] == 1){
                    res[j] = 0;
                } else if(j > 0){
                    res[j] += res[j - 1];
                }
            }
        }
        return res[obstacleGrid[0].length - 1];
    }
```

3. Minimum Path Sum，Q1相当于每个格子权重为1，Q2相当于1的格子权重无限大。

```
    // 2D-Dynamic Programming-Stable
    public int minPathSum(int[][] grid) {
        if(grid == null || grid.length == 0 || grid[0].length == 0){
            return 0;
        }
        int m = grid.length;
        int n = grid[0].length;
        int[][] res = new int[m][n];
        res[0][0] = grid[0][0];
        for(int i = 1; i < m; i++){
            res[i][0] = res[i - 1][0] + grid[i][0];
        }
        for(int j = 1; j < n; j++){
            res[0][j] = res[0][j - 1] + grid[0][j];
        }
        for(int i = 1; i < m; i++){
            for(int j = 1; j < n; j++){
                res[i][j] = Math.min(res[i - 1][j], res[i][j - 1]) + grid[i][j];
            }
        }
        return res[m - 1][n - 1];
    }
```

```
// 1D-Dynamic Programming-Optimal
public int minPathSum(int[][] grid) {
    if(grid == null || grid.length == 0 || grid[0].length == 0){
        return 0;
    }
    int m = grid.length;
    int n = grid[0].length;
    int[] res = new int[n];
    for(int i = 0; i < m; i++){
        for(int j = 0; j < n; j++){
            if(j == 0){
                res[j] += grid[i][j];
            } else if(i == 0){
                res[j] = res[j - 1] + grid[i][j];
            } else {
                res[j] = Math.min(res[j - 1], res[j]) + grid[i][j];
            }
        }
    }
    return res[n - 1];
}
```

4. Dungeon Game，从终点逆推回起点的案例。可以使用矩阵自身来降低空间复杂度。

```
public int calculateMinimumHP(int[][] dungeon) {
    if(dungeon == null || dungeon.length == 0 || dungeon[0].length == 0){
        return 0;
    }
    int row = dungeon.length;
    int col = dungeon[0].length;
    for(int i = row - 1; i >= 0; i--){
        for(int j = col - 1; j >= 0; j--){
            if(i == row - 1 && j == col - 1){
                dungeon[i][j] = Math.max(1, 1 - dungeon[i][j]);
            } else if(i == row - 1){
                dungeon[i][j] = Math.max(1, dungeon[i][j + 1] - dungeon[i][j]);
            } else if(j == col - 1){
                dungeon[i][j] = Math.max(1, dungeon[i + 1][j] - dungeon[i][j]);
            } else {
```

```
                    dungeon[i][j] = Math.max(1, Math.min(dungeon[i][j + 1],
dungeon[i + 1][j]) - dungeon[i][j]);
                }
            }
        }
        return dungeon[0][0];
    }
```

Part 2 – Sequence Dynamic Programming

这类题目是动态规划当中中等难度的，递推量相对固定，但递推式需要根据题目的要求做出相应的变化。同样当我们明确了4个主要元素，问题就会迎刃而解。

1）确定递推量。res[i]一般表示第i个位置/数字/字母的状态；

2）推导递推式。一般可以研究res[i]之前的res[j]对res[i]的影响；

3）计算初始条件。res[0]/res[n - 1]；

4）考虑存储历史信息的空间维度（可选）。一般就是一维空间，不可优化。

在 LeetCode 和 LintCode 中关于 Sequence DP 的题目有：

1. Climbing Stairs
2. Jump Game
3. Jump Game II
4. Palindrome Partitioning II
5. Word Break
6. Longest Increasing Subsequence

1. Climbing Stairs，res[i]表示前i个位置跳到第i个位置的方案总数，递推式res[i] = res[i - 1] + res[i - 2]，典型的Fabonacci数列。初始化res[0] = 1, res[1] = 1，递推得到结果。

```
    // Dynamic Programming
    public int climbStairs(int n) {
        int[] map = new int[n + 1];
        return helper(n, map);
    }

    private int helper(int n, int[] map){
        if(n < 0){
```

```
            return 0;
        }
        if(n == 0){
            return 1;
        }
        if(map[n] != 0){
            return map[n];
        }
        map[n] = helper(n - 1, map) + helper(n - 2, map);
        return map[n];
    }

    // Fabonacci
    public int climbStairs(int n) {
        if(n < 0){
            return 0;
        }
        int f1 = 1;
        int f2 = 1;
        if(n == 0){
            return f1;
        }
        if(n == 1){
            return f2;
        }
        for(int i = 2; i <= n; i++){
            int f3 = f1 + f2;
            f1 = f2;
            f2 = f3;
        }
        return f2;
    }
```

2. Jump Game，其实这道题使用贪心算法比较好，所以第一种我们提供一个贪心算法的解法。虽然LeetCode加入了Test Case:[25000, 24999, 24998...]之后动态规划会超时，但动态规划的思想还是很值得借鉴。所以第二种是原版模板动态规划解法，第三种是针对该题优化版的动态规划解法。优化点1：既然某点可达，那么它前面的点肯定全部可达。优化点2：如果某点不可达了，直接返回false即可，不需要再往后计算了。

```
// Greedy Algorithm
public boolean canJump(int[] A) {
    if(A == null || A.length == 0){
        return true;
    }
    int reach = 0;
    for(int i = 0; i <= reach && i < A.length; i++){
        reach = Math.max(reach, A[i] + i);
        if(reach >= A.length - 1){
            return true;
        }
    }
    return false;
}

// DP-Original-TLE
public boolean canJump(int[] A) {
    if(A == null || A.length == 0){
        return true;
    }
    boolean[] reachable = new boolean[A.length];
    reachable[0] = true;
    for(int i = 1; i < A.length; i++){
        for(int j = 0; j < i; j++){
            if(reachable[j] && A[j] + j >= i){
                reachable[i] = true;
                break;
            }
        }
    }
    return reachable[A.length - 1];
}

// DP-Optimized-TLE
public boolean canJump(int[] A) {
    if(A == null || A.length == 0){
        return true;
    }
    for(int i = 1; i < A.length; i++){
        boolean reachable = false;
```

```
            for(int j = 0; j < i; j++){
                if(A[j] + j >= i){
                    reachable = true;
                    break;
                }
            }
            if(!reachable){
                return false;
            }
        }
        return true;
    }
```

3. Jump Game II，同样解法一使用贪心算法；解法二使用模板动态规划，但会超时。

```
    // Greedy Algorithm
    public int jump(int[] A) {
        if(A == null || A.length == 0){
            return 0;
        }
        int step = 0;
        int reach = 0;
        int lastReach = 0;
        for(int i = 0; i <= reach && i < A.length; i++){
            if(i > lastReach){
                step++;
                lastReach = reach;
            }
            reach = Math.max(reach, A[i] + i);
            if(lastReach >= A.length - 1){
                return step;
            }
        }
        return 0;
    }

    // DP-Original-TLE
    public int jump(int[] A) {
        if(A == null || A.length == 0){
            return 0;
```

```
        }
        int[] steps = new int[A.length];
        steps[0] = 0;
        for(int i = 1; i < A.length; i++){
            steps[i] = i;
            for(int j = 0; j < i; j++){
                if(steps[j] < i && A[j] + j >= i){
                    steps[i] = steps[j] + 1;
                    break;
                }
            }
        }
        return steps[A.length - 1];
    }
```

4. Palindrome Partitioning II, 相当于计算最少步数的经典模板, 不过 cut 数是子串数 - 1。

```
    public int minCut(String s) {
        if(s == null || s.length() == 0){
            return 0;
        }
        int[] res = new int[s.length() + 1];
        boolean[][] dict = getDict(s);
        for(int i = 0; i < s.length(); i++){
            res[i + 1] = i + 1;
            for(int j = 0; j <= i; j++){
                if(dict[j][i] && res[j] + 1 < res[i + 1]){
                    res[i + 1] = res[j] + 1;
                }
            }
        }
        return res[s.length()] - 1;
    }

    private boolean[][] getDict(String s){
        boolean[][] dict = new boolean[s.length()][s.length()];
        for(int i = s.length() - 1; i >= 0; i--){
            for(int j = i; j < s.length(); j++){
                if(s.charAt(i) == s.charAt(j) && (j - i <= 2 || dict[i + 1][j - 1])){
                    dict[i][j] = true;
```

```
                }
            }
        }
        return dict;
    }
```

5. Word Break，依然模板。巧妙的每次删除第一个字符，相当于dict[j][i]的作用。

```
    public boolean wordBreak(String s, Set<String> dict) {
        if(s == null || s.length() == 0){
            return true;
        }
        boolean[] res = new boolean[s.length() + 1];
        res[0] = true;
        for(int i = 0; i < s.length(); i++){
            StringBuilder piece = new StringBuilder(s.substring(0, i + 1));
            for(int j = 0; j <= i; j++){
                if(res[j] && dict.contains(piece.toString())){
                    res[i + 1] = true;
                    break;
                }
                piece.deleteCharAt(0);
            }
        }
        return res[s.length()];
    }
```

6. Longest Increasing Subsequence，res[i]表示前i个数字中以第i个数字结尾的LIS长度。

```
    public int longestIncreasingSubsequence(int[] nums) {
        if(nums == null || nums.length == 0){
            return 0;
        }
        int max = 0;
        int[] res = new int[nums.length];
        for(int i = 0; i < nums.length; i++){
            res[i] = 1;
            for(int j = 0; j < i; j++){
                if(nums[j] <= nums[i] && res[j] + 1 > res[i]){
                    res[i] = res[j] + 1;
                }
```

```
            }
            max = Math.max(max, res[i]);
        }
        return max;
    }
```

Part 3 – Two Sequences Dynamic Programming

这类题目是动态规划乃至全LeetCode题目当中最难的题目，4个主要元素完全不固定，递推量和递推式都要因题而异而且不好确定。但做多了还是有些许规律可循的，大部分是针对字符串处理的动态规划题目，理清s字符串第i个字符和p字符串第j个字符之间的关系即可。

1）确定递推量。res[i][j]表示第一个字符串前i个字符配上第二个字符串前j个字符的状态；

2）推导递推式。res[i][j]一般研究第i个字符和第j个字符的匹配关系；

3）计算初始条件。res[i][0]/res[0][i]；

4）考虑存储历史信息的空间维度（可选）。一般就是一维空间，不可优化。

在 LeetCode 和 LintCode 中关于 Two Sequences DP 的题目有：

1. Longest Common Subsequence
2. Longest Common Substring
3. Edit Distance
4. Distinct Subsequence
5. Interleaving String

1. Longest Common Subsequence，res[i][j]表示前i个字符配上前j个字符的LCS长度。

```
    public int longestCommonSubsequence(String A, String B) {
        if(A == null || A.length() == 0 || B == null || B.length() == 0){
            return 0;
        }
        int[][] res = new int[A.length() + 1][B.length() + 1];
        for(int i = 0; i < A.length(); i++){
            for(int j = 0; j < B.length(); j++){
                if(A.charAt(i) == B.charAt(j)){
                    res[i + 1][j + 1] = res[i][j] + 1;
                } else {
                    res[i + 1][j + 1] = Math.max(res[i + 1][j], res[i][j + 1]);
```

```
                }
            }
        }
        return res[A.length()][B.length()];
    }
```

2. Longest Common Substring，res[i][j]表示前i个字符配上前j个字符的LCS'的长度（一定以第i个和第j个字符结尾的LCS'），结果max也需要一直更新。

```
    public int longestCommonSubstring(String A, String B) {
        if(A == null || A.length() == 0 || B == null || B.length() == 0){
            return 0;
        }
        int max = 0;
        int[][] res = new int[A.length() + 1][B.length() + 1];
        for(int i = 0; i < A.length(); i++){
            for(int j = 0; j < B.length(); j++){
                if(A.charAt(i) == B.charAt(j)){
                    res[i + 1][j + 1] = res[i][j] + 1;
                } else {
                    res[i + 1][j + 1] = 0;
                }
                max = Math.max(max, res[i + 1][j + 1]);
            }
        }
        return max;
    }
```

3. Edit Distance，res[i][j]表示word1的前i个字符配上word2的前j个字符最少要用几次编辑使得他们相等。优先使用稳定版，有能力再使用优化版。

```
    // 2D-Dynamic Programming-Stable
    public int minDistance(String word1, String word2) {
        if(word1 == null || word1.length() == 0){
            return word2.length();
        }
        if(word2 == null || word2.length() == 0){
            return word1.length();
        }
        int[][] res = new int[word1.length() + 1][word2.length() + 1];
```

```
        for(int i = 0; i <= word1.length(); i++){
            res[i][0] = i;
        }
        for(int j = 0; j <= word2.length(); j++){
            res[0][j] = j;
        }
        for(int i = 0; i < word1.length(); i++){
            for(int j = 0; j < word2.length(); j++){
                if(word1.charAt(i) == word2.charAt(j)){
                    res[i + 1][j + 1] = res[i][j];
                } else {
                    res[i + 1][j + 1] = Math.min(Math.min(res[i][j + 1], res[i + 1][j]),
res[i][j]) + 1;
                }
            }
        }
        return res[word1.length()][word2.length()];
    }

    // 1D-Dynamic Programming-Optimal
    public int minDistance(String word1, String word2) {
        if(word1.length() == 0){
            return word2.length();
        }
        if(word2.length() == 0){
            return word1.length();
        }
        String minWord = word1.length() > word2.length() ? word2 : word1;
        String maxWord = word1.length() > word2.length() ? word1 : word2;
        int[] res = new int[minWord.length() + 1];
        for(int i = 0; i <= minWord.length(); i++){
            res[i] = i;
        }
        for(int i = 0; i < maxWord.length(); i++){
            int[] newRes = new int[minWord.length() + 1];
            newRes[0] = i + 1;
            for(int j = 0; j < minWord.length(); j++){
                if(maxWord.charAt(i) == minWord.charAt(j)){
                    newRes[j + 1] = res[j];
                } else {
```

```
                    newRes[j + 1] = Math.min(Math.min(res[j], res[j + 1]), newRes[j])
+ 1;
                }
            }
            res = newRes;
        }
        return res[minWord.length()];
    }
```

4. Distinct Subsequence，res[i][j]表示S的前i个字符和T的前j个字符有多少个可行的序列。假设S的第i个字符和T的第j个字符不相同，那么就意味着res[i][j]的值和res[i - 1][j]的值相同，前面该是多少还是多少，而第i个字符的加入也不会多出来任何可行结果。如果S的第i个字符和T的第j个字符相同，那么所有res[i - 1][j - 1]中满足的结果都会成为新的满足的序列，当然res[i - 1][j]的也仍是可行的结果。算法进行两层循环，时间复杂度为O(m * n)，空间复杂度O(m)。优先使用稳定版，有能力再使用优化版。

```
    // 2D-Dynamic Programming-Stable
    public int numDistinct(String S, String T) {
        if(S == null || S.length() == 0 || T == null || T.length() == 0){
            return 0;
        }
        int[][] res = new int[S.length() + 1][T.length() + 1];
        for(int i = 0; i <= S.length(); i++){
            res[i][0] = 1;
        }
        for(int i = 0; i < S.length(); i++){
            for(int j = 0; j < T.length(); j++){
                if(S.charAt(i) == T.charAt(j)){
                    res[i + 1][j + 1] = res[i][j] + res[i][j + 1];
                } else {
                    res[i + 1][j + 1] = res[i][j + 1];
                }
            }
        }
        return res[S.length()][T.length()];
    }
```

```
// 1D-Dynamic Programming-Optimal
public int numDistinct(String S, String T) {
    if(S == null || S.length() == 0 || T == null || T.length() == 0){
        return 0;
    }
    int[] res = new int[T.length() + 1];
    res[0] = 1;
    for(int i = 0; i < S.length(); i++){
        for(int j = T.length() - 1; j >= 0; j--){
            res[j + 1] += (S.charAt(i) == T.charAt(j)) ? res[j] : 0;
        }
    }
    return res[T.length()];
}
```

5. Interleaving String，res[i][j]表示用s1的前i个字符和s2的前j个字符能不能按照规则表示出前s3的前i+j个字符。优先使用稳定版，有能力再使用优化版。

```
// 2D-Dynamic Programming-Stable
public boolean isInterleave(String s1, String s2, String s3) {
    if(s1.length() + s2.length() != s3.length()){
        return false;
    }
    boolean[][] res = new boolean[s1.length() + 1][s2.length() + 1];
    res[0][0] = true;
    for(int i = 0; i < s1.length(); i++){
        res[i + 1][0] = res[i][0] && s1.charAt(i) == s3.charAt(i);
    }
    for(int j = 0; j < s2.length(); j++){
        res[0][j + 1] = res[0][j] && s2.charAt(j) == s3.charAt(j);
    }
    for(int i = 0; i < s1.length(); i++){
        for(int j = 0; j < s2.length(); j++){
            res[i + 1][j + 1] = res[i + 1][j] && s2.charAt(j) == s3.charAt(i + j + 1)
                              || res[i][j + 1] && s1.charAt(i) == s3.charAt(i + j + 1);
        }
    }
    return res[s1.length()][s2.length()];
}
```

```
// 1D-Dynamic Programming-Optimal
public boolean isInterleave(String s1, String s2, String s3) {
    if(s1.length() + s2.length() != s3.length()){
        return false;
    }
    String minWord = s1.length() < s2.length() ? s1 : s2;
    String maxWord = s1.length() < s2.length() ? s2 : s1;
    boolean[] res = new boolean[minWord.length() + 1];
    res[0] = true;
    for(int i = 0; i < minWord.length(); i++){
        res[i + 1] = res[i] && minWord.charAt(i) == s3.charAt(i);
    }
    for(int i = 0; i < maxWord.length(); i++){
        res[0] = res[0] && maxWord.charAt(i) == s3.charAt(i);
        for(int j = 0; j < minWord.length(); j++){
            res[j + 1] = res[j + 1] && maxWord.charAt(i) == s3.charAt(i + j + 1)
                    || res[j] && minWord.charAt(j) == s3.charAt(i + j + 1);
        }
    }
    return res[minWord.length()];
}
```

Part 4 – Back Pack

问题：给 n 个正整数，一个数 target，问能否从 n 个数中取出若干个数，使他们的和为 target。

1）递推量。res[i][S]表示“前i”个数字，取出一些能否组成和为S；

2）递推式。res[i][S] = res[i - 1][S – a[i]] or res[i - 1][S]；

3）初始化。res[0][0] = true; res[0][1...Sum] = false；

4）结果集。res[n][target]。

K Sum 问题：给 n 个数，从中取 k 个数，组成和为 target。

1）递推量。res[i][j][t]表示前i个数取j个数出来能否和为t；

2）递推式。res[i][j][t] = res[i - 1][j - 1][t – a[i]] or res[i - 1][j][t]；

K Sum 题有如下 3 种问法：

当问是否可行或方案总数时，使用 DP。

当问所有方案时，使用递归/搜索。

最小调整代价：n 个数，可以对每个数字进行调整，使得相邻的两个数的差都<=target，调整费用为 Sigma(|A[i] – B[i]|)，A[i]为原来的序列，B[i]为调整后的序列，让代价最小。

1）递推量。res[i][v]表示“前i”个数字，第i个数调整为v，满足相邻两数 <= target，所需要的最小代价；

2）递推式。res[i][v] = min(res[i - 1][v'] + |A[i] - v|，|v – v'| <= target)；

3）结果集。res[n][...]。

Part 5 – 综合题

这部分为动态规划中没有明确分类的综合题，有难有易，后面字符串处理题目较难，掌握思想即可。这里为保证动态规划题目的完整性，在此稍作解析。Maximum Subarray 和 Best Time to Buy and Sell Stock 这类题目在高频题部分还将详细讲解。

在 LeetCode 中动态规划的综合题目有以下几道：

1. Triangle
2. Unique Binary Search Trees
3. Unique Binary Search Trees II
4. Word Break II
5. Maximum Subarray
6. Maximum Product Subarray
7. Best Time to Buy and Sell Stock
8. Best Time to Buy and Sell Stock III
9. Longest Valid Parentheses
10. Decode Ways
11. Maximal Rectangle
12. Scramble String
13. Wildcard Matching
14. Regular Expression Matching

1. Triangle，注意行数，元素下标和循环起止点的关系即可。从下向上遍历的好处是，最后剩下的只有三角尖的一个元素，即为我们要的结果。

```
public int minimumTotal(List<List<Integer>> triangle) {
    if(triangle == null || triangle.size() == 0){
        return 0;
    }
    List<Integer> preList = triangle.get(triangle.size() - 1);
    for(int i = triangle.size() - 2; i >= 0; i--){
        List<Integer> curList = triangle.get(i);
        for(int j = i; j >= 0; j--){
            curList.set(j, curList.get(j) + Math.min(preList.get(j), preList.get(j + 1)));
        }
        preList = curList;
    }
    return triangle.get(0).get(0);
}
```

2. Unique Binary Search Trees，在树的陨落篇已经介绍，卡特兰数，直接计算即可。

```
public int numTrees(int n) {
    if(n <= 0){
        return 0;
    }
    int[] res = new int[n + 1];
    res[0] = 1;
    for(int i = 1; i <= n; i++){
        for(int j = 0; j < i; j++){
            res[i] += res[j] * res[i - 1 - j];
        }
    }
    return res[n];
}
```

3. Unique Binary Search Trees II，在树的陨落篇已经介绍，动态规划用在已知两边子树的根结点的时候，如果排列组合构造高一层的树。

```
public ArrayList<TreeNode> generateTrees(int n) {
    return helper(1, n);
}

private ArrayList<TreeNode> helper(int left, int right){
```

```
        ArrayList<TreeNode> res = new ArrayList<TreeNode>();
        if(left > right){
            res.add(null);
            return res;
        }
        for(int k = left; k <= right; k++){
            ArrayList<TreeNode> leftList = helper(left, k - 1);
            ArrayList<TreeNode> rightList = helper(k + 1, right);
            for(int i = 0; i < leftList.size(); i++){
                for(int j = 0; j < rightList.size(); j++){
                    TreeNode root = new TreeNode(k);
                    root.left = leftList.get(i);
                    root.right = rightList.get(j);
                    res.add(root);
                }
            }
        }
        return res;
    }
```

4. Word Break II，回溯篇已经介绍过，动态规划用在剪枝部分，注意possible数组的使用。

```
    public ArrayList<String> wordBreak(String s, Set<String> dict) {
        ArrayList<String> res = new ArrayList<String>();
        if(s == null || s.length() == 0 || dict == null || dict.size() == 0){
            return res;
        }
        boolean[] possible = new boolean[s.length() + 1];
        Arrays.fill(possible, true);
        StringBuilder item = new StringBuilder();
        helper(s, dict, 0, possible, item, res);
        return res;
    }

    private void helper(String s, Set<String> dict, int start, boolean[] possible,
StringBuilder item, ArrayList<String> res){
        if(start == s.length()){
            res.add(item.toString().substring(0, item.length() - 1));
            return;
        }
```

```
        for(int i = start; i < s.length(); i++){
            String piece = s.substring(start, i + 1);
            if(possible[i + 1] && dict.contains(piece)){
                item.append(piece).append(" ");
                int curResSize = res.size();
                helper(s, dict, i + 1, possible, item, res);
                if(res.size() == curResSize){
                    possible[i + 1] = false;
                }
                item.delete(item.length() - piece.length() - 1, item.length());
            }
        }
    }
```

5. Maximum Subarray，提供2种解法。第一种，动态规划“局部最优和全局最优”解法；第二种，使用Divide and Conquer技术的解法。

```
    // Solution 1 - Dynamic Programming
    public int maxSubArray(int[] A) {
        if(A == null || A.length == 0){
            return 0;
        }
        int local = A[0];
        int global = A[0];
        for(int i = 1; i < A.length; i++){
            local = Math.max(A[i], A[i] + local);
            global = Math.max(local, global);
        }
        return global;
    }

    // Solution 2 - Divide and Conquer
    public int maxSubArray(int[] A) {
        if(A == null || A.length == 0){
            return 0;
        }
        return helper(A, 0, A.length - 1);
    }

    private int helper(int[] A, int left, int right){
```

```
        if(left == right){
            return A[left];
        }
        int mid = left + (right - left) / 2;
        int leftSub = helper(A, left, mid);
        int rightSub = helper(A, mid + 1, right);
        int leftMax = A[mid];
        int temp = 0;
        for(int i = mid; i >= left; i--){
            temp += A[i];
            leftMax = Math.max(temp, leftMax);
        }
        int rightMax = A[mid + 1];
        temp = 0;
        for(int j = mid + 1; j <= right; j++){
            temp += A[j];
            rightMax = Math.max(temp, rightMax);
        }
        return Math.max(Math.max(leftSub, rightSub), leftMax + rightMax);
    }
```

6. Maximum Product Subarray，使用动态规划“局部最优和全局最优”解法。

```
    public int maxProduct(int[] A) {
        if(A == null || A.length == 0){
            return 0;
        }
        int localMax = A[0];
        int localMin = A[0];
        int global = A[0];
        for(int i = 1; i < A.length; i++){
            int tempLocalMax = localMax;
            localMax = Math.max(A[i], Math.max(A[i] * localMin, A[i] * localMax));
            localMin = Math.min(A[i], Math.min(A[i] * localMin, A[i] * tempLocalMax));
            global = Math.max(global, localMax);
        }
        return global;
    }
```

7. Best Time to Buy and Sell Stock，使用动态规划“局部最优和全局最优”解法。

```
public int maxProfit(int[] prices) {
    if(prices == null || prices.length == 0){
        return 0;
    }
    int local = 0;
    int global = 0;
    for(int i = 0; i < prices.length - 1; i++){
        local = Math.max(local, 0) + prices[i + 1] - prices[i];
        global = Math.max(local, global);
    }
    return global;
}
```

8. Best Time to Buy and Sell Stock III，提供2种解法。第一种，使用2次动态规划“局部最优和全局最优”来解；第二种，使用通用的K次解法，但较难理解。

```
public int maxProfit(int[] prices) {
    if(prices == null || prices.length <= 1){
        return 0;
    }
    int[] left = new int[prices.length];
    int local = 0;
    int global = 0;
    for(int i = 0; i < prices.length - 1; i++){
        local = Math.max(local, 0) + prices[i + 1] - prices[i];
        global = Math.max(local, global);
        left[i] = global;
    }
    int[] right = new int[prices.length];
    local = 0;
    global = 0;
    for(int j = prices.length - 2; j >= 0; j--){
        local = Math.max(local, 0) + prices[j + 1] - prices[j];
        global = Math.max(local, global);
        right[j] = global;
    }
    int max = 0;
    for(int i = 0; i < prices.length - 1; i++){
        max = Math.max(max, left[i] + right[i + 1]);
```

```
        }
        return max;
    }

    public int maxProfit(int[] prices) {
        if(prices == null || prices.length == 0){
            return 0;
        }
        int[] local = new int[3];
        int[] global = new int[3];
        for(int i = 0; i < prices.length - 1; i++){
            int diff = prices[i + 1] - prices[i];
            for(int j = 2; j >= 1; j--){
                local[j] = Math.max(global[j - 1] + Math.max(diff, 0), local[j] +
diff);
                global[j] = Math.max(local[j], global[j]);
            }
        }
        return global[2];
    }
```

9. Longest Valid Parentheses，使用stack和一个start指针来记录开始合法的位置。

```
    public int longestValidParentheses(String s) {
        if(s == null || s.length() == 0){
            return 0;
        }
        int max = 0;
        int start = 0;
        LinkedList<Integer> stack = new LinkedList<Integer>();
        for(int i = 0; i < s.length(); i++){
            if(s.charAt(i) == '('){
                stack.push(i);
            } else {
                if(stack.isEmpty()){
                   start = i + 1;
                } else {
                    stack.pop();
                    max = Math.max(max, stack.isEmpty() ? i - start + 1 : i -
stack.peek());
```

```
                }
            }
        }
        return max;
    }
```

10. Decode Ways，将2位数合理的分为4种，对不同种分别进行相应的处理即可。

```
    public int numDecodings(String s) {
        if(s == null || s.length() == 0 || s.charAt(0) == '0'){
            return 0;
        }
        int num1 = 1;
        int num2 = 1;
        int num3 = 1;
        for(int i = 1; i < s.length(); i++){
            if(s.charAt(i) == '0'){
                if(s.charAt(i - 1) == '1' || s.charAt(i - 1) == '2'){
                    num3 = num1;
                } else {
                    return 0;
                }
            } else {
                if(s.charAt(i - 1) == '0' || s.charAt(i - 1) >= '3'){
                    num3 = num2;
                } else {
                    if(s.charAt(i - 1) == '2' && s.charAt(i) >= '7' && s.charAt(i)
<= '9'){
                        num3 = num2;
                    } else {
                        num3 = num1 + num2;
                    }
                }
            }
            num1 = num2;
            num2 = num3;
        }
        return num2;
    }
```

11. Maximal Rectangle，巧妙的使用直方图最大面积那道题的解法作为subroutine，并使用动态规划来更新每层的height数组，可以把时间复杂度将到O(m * n)。

```
public int maximalRectangle(char[][] matrix) {
    if(matrix == null || matrix.length == 0 || matrix[0].length == 0){
        return 0;
    }
    int maxArea = 0;
    int[] height = new int[matrix[0].length];
    for(int i = 0; i < matrix.length; i++){
        for(int j = 0; j < matrix[0].length; j++){
            if(matrix[i][j] == '0'){
                height[j] = 0;
            } else {
                height[j]++;
            }
        }
        maxArea = Math.max(maxArea, findMaxArea(height));
    }
    return maxArea;
}

private int findMaxArea(int[] h){
    int maxArea = 0;
    LinkedList<Integer> stack = new LinkedList<Integer>();
    for(int i = 0; i <= h.length; i++){
        int value = (i == h.length ? -1 : h[i]);
        while(!stack.isEmpty() && h[stack.peek()] > value){
            int index = stack.pop();
            int height = h[index];
            int width = stack.isEmpty() ? i : i - stack.peek() - 1;
            maxArea = Math.max(maxArea, height * width);
        }
        stack.push(i);
    }
    return maxArea;
}
```

12. Scramble String，三维动态规划，第三纬度代表字符串的长度。

```
public boolean isScramble(String s1, String s2) {
    if(s1 == null || s2 == null || s1.length() != s2.length()){
        return false;
    }
    if(s1.length() == 0){
        return true;
    }
    boolean[][][] res = new boolean[s1.length()][s2.length()][s1.length() +
1];
    for(int i = 0; i < s1.length(); i++){
        for(int j = 0; j < s2.length(); j++){
            res[i][j][1] = s1.charAt(i) == s2.charAt(j);
        }
    }
    for(int len = 2; len <= s1.length(); len++){
        for(int i = 0; i < s1.length() - len + 1; i++){
            for(int j = 0; j < s2.length() - len + 1; j++){
                for(int k = 1; k < len; k++){
                    res[i][j][len] |= res[i][j + len - k][k] && res[i + k][j][len - k]
                            || res[i][j][k] && res[i + k][j + k][len - k];
                }
            }
        }
    }
    return res[0][0][s1.length()];
}
```

13. Wildcard Matching，提供2种解法。第一种，巧妙的双指针法；第二种，动态规划。

```
// Solution 1 - Two Points
public boolean isMatch(String s, String p) {
    int idxS = 0;
    int idxP = 0;
    int idxStar = -1;
    int match = 0;
    while(idxS < s.length()){
        if(idxP < p.length() && (s.charAt(idxS) == p.charAt(idxP) ||
p.charAt(idxP) == '?')){
            idxS++;
```

```
                idxP++;
            } else if(idxP < p.length() && p.charAt(idxP) == '*'){
                idxStar = idxP;
                match = idxS;
                idxP++;
            } else if(idxStar != -1){
                idxP = idxStar + 1;
                match++;
                idxS = match;
            } else {
                return false;
            }
        }
        while(idxP < p.length() && p.charAt(idxP) == '*'){
            idxP++;
        }
        return idxP == p.length();
    }

    // Solution 2 - Dynamic Programming
    public boolean isMatch(String s, String p) {
        if(p.length() == 0){
            return s.length() == 0;
        }
        if(s.length() > 300){
            return false;
        }
        boolean[] res = new boolean[s.length() + 1];
        res[0] = true;
        for(int j = 0; j < p.length(); j++){
            if(p.charAt(j) != '*'){
                for(int i = s.length() - 1; i >= 0; i--){
                    res[i + 1] = res[i] && (s.charAt(i) == p.charAt(j) || p.charAt(j)
== '?');
                }
            } else {
                int i = 0;
                while(i < s.length() && !res[i]){
                    i++;
                }
```

```
                for(; i < s.length(); i++){
                    res[i + 1] = true;
                }
            }
            res[0] = res[0] && p.charAt(j) == '*';
        }
        return res[s.length()];
    }
```

14. Regular Expression Matching，提供2种方法。第一种，递归；第二种，动态规划。

```
    // Solution 1 - Brute Force
    public boolean isMatch(String s, String p) {
        return helper(s, p, 0, 0);
    }

    private boolean helper(String s, String p, int i, int j){
        if(j == p.length()){
            return i == s.length();
        }
        if(j == p.length() - 1 || p.charAt(j + 1) != '*'){
            if(i == s.length() || s.charAt(i) != p.charAt(j) && p.charAt(j) != '.'){
                return false;
            } else {
                return helper(s, p, i + 1, j + 1);
            }
        }
        while(i < s.length() && (s.charAt(i) == p.charAt(j) || p.charAt(j) == '.')){
            if(helper(s, p, i, j + 2)){
                return true;
            }
            i++;
        }
        return helper(s, p, i, j + 2);
    }

    // Solution 2 - Dynamic Programming
    public boolean isMatch(String s, String p) {
        if(p.length() == 0){
            return s.length() == 0;
```

```
        }
        boolean[][] res = new boolean[s.length() + 1][p.length() + 1];
        res[0][0] = true;
        for(int j = 0; j < p.length(); j++){
            if(p.charAt(j) == '*'){
                if(j == 0){
                    continue;
                }
                if(res[0][j - 1]){
                    res[0][j + 1] = true;
                }
                if(p.charAt(j - 1) != '.'){
                    for(int i = 0; i < s.length(); i++){
                        res[i + 1][j + 1] = res[i + 1][j - 1] || res[i + 1][j] || i > 0 &&
res[i][j + 1] && s.charAt(i) == s.charAt(i - 1) && s.charAt(i - 1) == p.charAt(j - 1);
                    }
                } else {
                    int i = 0;
                    while(i < s.length() && !res[i + 1][j - 1] && !res[i + 1][j]){
                        i++;
                    }
                    for(; i < s.length(); i++){
                        res[i + 1][j + 1] = true;
                    }
                }
            } else {
                for(int i = 0; i < s.length(); i++){
                    res[i + 1][j + 1] = res[i][j] && (s.charAt(i) == p.charAt(j) ||
p.charAt(j) == '.');
                }
            }
        }
        return res[s.length()][p.length()];
    }
```