动态规划常用递推公式

单词拆分

```
for (int i = 1; i <= len; i++) {
    for (int j = 0; j < i; j++) {
// dp[j]表示字符串j位置结尾的子串在dict中是否存在
        if (dp[j] == true && dictSet.contains(s.substring(j, i))) {
            dp[i] = true;
// 如果i位置之前的元素可以被拆分，那么直接可以终止内层循环，继续移动i的位置
            break;
```

1-n区间数字可以组成BST的个数

```
// dp[i] ： 1到i为节点组成的二叉搜索树的个数为dp[i]。
int[] dp = new int[n + 1];
for (int i = 2; i <= n; i++) {
   for (int j = 1; j <= i; j++) {
      // dp[j - 1]表示总共有i个节点的情况下，以j为头结点，左边的二叉搜索树的可能情况的个数
      // dp[i - j]表示总共有i个节点的情况下，以j为头结点，右边的二叉搜索树的可能情况的个数
      dp[i] += dp[j - 1] * dp[i - j];
```

使用最少的完全平方数凑成n

01背包：dp[j] = min(dp[j - i * i] + 1, dp[j])

等分数组

典型的01背包问题 dp[i][j] = Math.max(dp[i - 1][j], dp[i - 1][j - w] + w);

1和0

```
dp[j][k]表示有j个0，k个1的子串中字符串个数最多的情况
    for (int i = 1; i <= len; i++) {
        int[] ints = getOneZeroCnt(strs[i - 1]);
        int zeroCnt = ints[0];
        int oneCnt = ints[1];
        for (int j = m; j >= zeroCnt; j--) {
            for (int k = n; k >= oneCnt; k--) {
                dp[j][k] = Math.max(dp[j][k], dp[j - zeroCnt][k - oneCnt] + 1);
    return dp[m][n];
```

矩阵路径和最小

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

和最大的子数组

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

最大正方形

```
// dp[i][j]表示以(i, j)为右下角顶点的最大正方形的边长。
dp[i][j] = Math.min(Math.min(dp[i - 1][j], dp[i][j - 1]), dp[i - 1][j - 1]) + 1;
```

最长回文子串

```
dp[i][j]表示字符串s的[i,j]区间是否是回文串
    char[] chars = s.toCharArray();  // 转化成数组可以加快程序运行效率
    int start = 0, max = 1;
    for (int j = 1; j < len; j++) {
        for (int i = 0; i <= j; i++) {
            if (i==j){
                dp[i][j]=true;
                continue;
            }
            if (chars[i] != chars[j]) {
                dp[i][j] = false;
            } else {
                if (j - i < 3) {
                    dp[i][j] = true;
                } else {
                    dp[i][j] = dp[i + 1][j - 1];
                }
            }

            if (dp[i][j] && j - i + 1 > max) {
                max = j - i + 1;
                start = i;
            }
        }
    }
    return s.substring(start, start + max);
```

硬币兑换

```
// 初始金额从1开始，因为严格意义上来说没有0元这个说法
for (int i = 1; i <= amount; i++)
    for (int j = 0; j < length; j++)
        if (i - coins[j] >= 0 && dp[i - coins[j]] != -1)
            // 下面的dp[i] > dp[i - coins[j]] + 1体现出了求最小的dp[i]的思想
            if (dp[i] == -1 || dp[i] > dp[i - coins[j]] + 1)
                dp[i] = dp[i - coins[j]] + 1;
```

硬币兑换2（求的是组合）

```
for (int i = 0; i < coins.length; i++) {
    for (int j = coins[i]; j <= amount; j++) {
        // dp[j]等于所有的dp[j - coins[i]]相加
        dp[j] += dp[j - coins[i]];
```

组合的和4（求的是排列）

```
// 先遍历背包
for (int j = 1; j <= target; j++) {
    // 再遍历物品
    for (int i = 0; i < nums.length; i++) {
        if (j >= nums[i]) {
            dp[j] += dp[j - nums[i]];
```

给定一个整数n，表示数字的位数。求所有的区间位数内的数字的个数，除开包含重复数字的数

```
// dp[i]表示小于等于i位数字，相加的和（除开重复数字）
int[] dp = new int[n + 1];
for (int i = 2; i <= n; i++) {
    dp[i] = dp[i - 1] + (dp[i - 1] - dp[i - 2]) * (10 - (i - 1));
```

单词编辑距离

```
dp[i][j]表示：word1以i-1位置结尾的子串，word2以j-1结尾的子串，两个子串变成一样，最少需要的操作次数
 dp[i][j] = Math.min(dp[i - 1][j - 1] + 1, Math.min(dp[i][j - 1] + 1, dp[i - 1][j] + 1));
```

整数拆分

```
// dp[i]表示，将整数i进行拆分，可以得到的最大乘积
        int[] dp = new int[n + 1];
dp[i] = Math.max(dp[i], Math.max(j * (i - j), j * dp[i - j]));
```

最长等差子数组

dp[diff][j] = dp[diff][i] + 1; // diff表示等差值

最长公共子序列

```
if (text1.charAt(i - 1) == text2.charAt(j - 1)) {
    dp[i][j] = dp[i - 1][j - 1] + 1;
} else {
    dp[i][j] = Math.max(dp[i - 1][j], dp[i][j - 1]);
```

最长上升子序列

```
// dp[i]表示以元素i结尾的最长上升子序列
int[] dp = new int[nums.length];
if (nums[i] > nums[i - 1]) {
        dp[i] = dp[i - 1] + 1;
```

最长回文序列

```
使用动态规划，将s反序，得到s1，求s和s1的LCS，结果就是本题求解的结果
        if (s.charAt(i - 1) == s1.charAt(j - 1)) {
            dp[i][j] = dp[i - 1][j - 1] + 1;
        } else {
            dp[i][j] = Math.max(dp[i][j - 1], dp[i - 1][j]);
```