动态规划算法：

基本思想：

动态规划算法通常用于求解具有某种最优性质的问题。在这类问题中，可能会有许多可行解。每一个解都对应于一个值，我们希望找到具有最优值的解。动态规划算法与分治法类似，其基本思想也是将待求解问题分解成若干个子问题，先求解子问题，然后从这些子问题的解得到原问题的解。与分治法不同的是，适合于用动态规划求解的问题，经分解得到子问题往往不是互相独立的。若用分治法来解这类问题，则分解得到的子问题数目太多，有些子问题被重复计算了很多次。如果我们能够保存已解决的子问题的答案，而在需要时再找出已求得的答案，这样就可以避免大量的重复计算，节省时间。我们可以用一个表来记录所有已解的子问题的答案。不管该子问题以后是否被用到，只要它被计算过，就将其结果填入表中。这就是动态规划法的基本思路。具体的动态规划算法多种多样，但它们具有相同的填表格式。

动态规划算法与分治法最大的差别是：适合于用动态规划法求解的问题，经分解后得到的子问题往往不是互相独立的（即下一个子阶段的求解是建立在上一个子阶段的解的基础上，进行进一步的求解）

应用场景：

适用动态规划的问题必须满足最优化原理、无后效性和重叠性。

1.最优化原理（最优子结构性质） 最优化原理可这样阐述：一个最优化策略具有这样的性质，不论过去状态和决策如何，对前面的决策所形成的状态而言，余下的诸决策必须构成最优策略。简而言之，一个最优化策略的子策略总是最优的。一个问题满足最优化原理又称其具有最优子结构性质。

2.无后效性　将各阶段按照一定的次序排列好之后，对于某个给定的阶段状态，它以前各阶段的状态无法直接影响它未来的决策，而只能通过当前的这个状态。换句话说，每个状态都是过去历史的一个完整总结。这就是无后向性，又称为无后效性。

3.子问题的重叠性　动态规划将原来具有指数级时间复杂度的搜索算法改进成了具有多项式时间复杂度的算法。其中的关键在于解决冗余，这是动态规划算法的根本目的。动态规划实质上是一种以空间换时间的技术，它在实现的过程中，不得不存储产生过程中的各种状态，所以它的空间复杂度要大于其它的算法。

动态规划算法经典案例：

案例一：

有 n 级台阶，一个人每次上一级或者两级，问有多少种走完 n 级台阶的方法。

分析：动态规划的实现的关键在于能不能准确合理的用动态规划表来抽象出 实际问题。在这个问题上，我们让 f(n)表示走上 n 级台阶的方法数。那么当 n 为 1 时，f(n) = 1,n 为 2 时，f(n) =2,就是说当台阶只有一级的时候，方法数是一种，台阶有两级的时候，方法数为 2。那么当我们要走上 n 级台阶，必然是从 n-1 级台阶迈一步或者是从 n-2 级台阶迈两步，所以到达 n 级台阶的方法数必然是到达 n-1 级台阶的方法数加上到达 n-2 级台阶的方法数之和。即 f(n) = f(n-1)+f(n-2)，我们用 dp[n]来表示动态规划表，dp[i],i>0,i<=n,表示到达 i 级台阶的方法数。

```
public class CalculationSteps {
    // 动态规划表，用来记录到达 i 级台阶的方法数
    public static int[] steps = new int[11];

    public static void main(String[] args) {
        steps[10] = calStep(10);
        for (int i = 0; i < steps.length; i++) {
```

```
            System.out.print(steps[i] + " ");
        }
        System.out.println();
        System.out.println(steps[10]);
    }

    // 计算到达 i 级台阶的方法数
    public static int calStep(int n) {
        // 如果为第一级台阶或者第二级台阶 则直接返回 n
        if (n == 1 || n == 2) {
            return n;
        }
        // 计算到达 n-1 级台阶的方法数
        if (steps[n - 1] == 0) {
            steps[n - 1] = calStep(n - 1);
        }
        // 计算到达 n-2 级台阶的方法数
        if (steps[n - 2] == 0) {
            steps[n - 2] = calStep(n - 2);
        }
        // 到达第 n 级台阶=到达 n-1 级台阶+到达 n-2 级台阶
        return steps[n - 1] + steps[n - 2];
    }
}
```

案例 2：

给定一个矩阵 m，从左上角开始每次只能向右走或者向下走，最后达到右下角的位置，路径中所有数字累加起来就是路径和，返回所有路径的最小路径和，如果给定的 m 如下，那么路径 1,3,1,0,6,1,0 就是最小路径和，返回 12.

```
1 3 5 9

8 1 3 4

5 0 6 1

8 8 4 0
```

分析：对于这个题目，假设 m 是 m 行 n 列的矩阵，那么我们用 dp[m][n]来抽象这个问题，dp[i][j]表示的是从原点到 i,j 位置的最短路径和。我们首先计算第一行和第一列，直接累加即可，那么对于其他位置，要么是从它左边的位置达到，要么是从上边的位置达到，我们取左边和上边的较小值，然后加上当前的路径值，就是达到当前点的最短路径。然后从左到右，从上到下依次计算即可。

```
/**
 * 给定一个矩阵 m，从左上角开始每次只能向右走或者向下走 最后达到右下角的位置，路径中所有数字累加起来就是路径和， 返回所有路径的最小路径和
```

```java
 */
public class MinSteps {

    public static int[][] steps = new int[4][4];

    public static void main(String[] args) {
        int[][] arr = { { 4, 1, 5, 3 }, { 3, 2, 7, 7 }, { 6, 5, 2, 8 }, { 8, 9, 4, 5 } };
        steps[3][3] = minSteps(arr, 3, 3);
        print(steps);
    }

    public static int minSteps(int[][] arr, int row, int col) {
        // 如果为起始位置，则直接返回
        if (row == 0 && col == 0) {
            steps[row][col] = arr[row][col];
            return steps[row][col];
        }

        // 计算到 arr[row][col]的左面位置的值
        if (col >= 1 && steps[row][col - 1] == 0) {
            steps[row][col - 1] = minSteps(arr, row, col - 1);
        }
        // 计算到 arr[row][col]的上面位置的值
        if (row >= 1 && steps[row - 1][col] == 0) {
            steps[row - 1][col] = minSteps(arr, row - 1, col);
        }
        // 如果为第一行，则直接加左面位置上的值
        if (row == 0 && col != 0) {
            steps[row][col] = arr[row][col] + steps[row][col - 1];
        } else if (col == 0 && row != 0) {
            // 如果为第一列，则直接加上上面位置上的值
            steps[row][col] = arr[row][col] + steps[row - 1][col];
        } else {
            // 比较到达左面位置和到达上面位置的值的大小，加上两者的最大值
            steps[row][col] = arr[row][col] + min(steps[row][col - 1], steps[row - 1][col]);
        }
        return steps[row][col];
    }

    private static int min(int minSteps, int minSteps2) {
        return minSteps > minSteps2 ? minSteps : minSteps2;
    }

    static void print(int[][] arr) {
```

```
        for (int i = 0; i < arr.length; i++) {
            for (int j = 0; j < arr[i].length; j++) {
                System.out.println("到达 arr[" + i + "][" + j + "]的最大路径：" + arr[i][j]);
            }
        }
    }
}
```

案例 3：最长公共子序列问题

最长公共子序列问题是要找到两个字符串间的最长公共子序列。假设有两个字符串 sudjxidjs 和 xidjxidpolkj，其中 djxidj 就是他们的最长公共子序列。许多问题都可以看成是公共子序列的变形。例如语音识别问题就可以看成最长公共子序列问题。

假设两个字符串分别为 A=a1a2..am,B=b1b2..bn,则 m 为 A 的长度，n 为 B 的长度。那么他们的最长公共子序列分为两种情况。

1、am=bn,这时他们的公共子序列一定为的长度 F(m,n)=F(m-1,n-1)+am；

2、am≠bn,这时他们的公共子序列一定为的长度 F(m,n)=Max(F(m-1,n),F(m,n-1))；

```
/**
 * 求两个字符串之间的最长子序列
 */
public class MaxCommonStr {
    // 数组用来存储两个字符串的最长公共子序列
    public static String[][] result = new String[10][15];

    public static void main(String[] args) {
        String strA = "sudjxidjs";
        String strB = "xidjxidpolkj";
        System.out.println(maxCommonStr(strA, strB));
        // System.out.println(strA.charAt(strA.length()-1));
    }

    /**
     * 获取两个字符串的最大公共子序列
     *
     * @param strA
     * @param strB
     * @return
     */
    public static String maxCommonStr(String strA, String strB) {
        // 分别获取两个字符串的长度
        int lenA = strA.length();
        int lenB = strB.length();

        // 如果字符串 strA 的长度为 1，那么如果 strB 包含字符串 strA,则公共子序列为 strA,否则为 null
```

```
        if (lenA == 1) {
            if (strB.contains(strA)) {
                result[lenA - 1][lenA - 1] = strA;
            } else {
                result[lenA - 1][lenA - 1] = "";
            }
            return result[lenA - 1][lenA - 1];
        }

        // 如果字符串 strB 的长度为 1，那么如果 strA 包含字符串 strB,则公共子序列为 strB,否则为 null
        if (lenB == 1) {
            if (strA.contains(strB)) {
                result[lenA - 1][lenA - 1] = strB;
            } else {
                result[lenA - 1][lenA - 1] = "";
            }
            return result[lenA - 1][lenA - 1];
        }

        // 如果字符串 strA 的最后一位和 strB 的最后一位相同的话，
        if (strA.charAt(lenA - 1) == strB.charAt(lenB - 1)) {
            //先判断数组 result[lenA - 2][lenB - 2] == null,这样可以减少一些重复运算
            if (result[lenA - 2][lenB - 2] == null) {
                //求 strA 和 strB 都去除最后一位剩余字符串的最大公共子序列 f
                result[lenA - 2][lenB - 2] = maxCommonStr(strLenSub(strA), strLenSub(strB));
            }
            //strA 和 strB 的最大公共子序列就是他们各去除最后一位剩余字符串的最大公共子序列+strA 或者 strB 的最后一位
            result[lenA - 1][lenB - 1] = result[lenA - 2][lenB - 2] + strA.charAt(lenA - 1);
        } else {
            //否则
            if (result[lenA - 2][lenB - 1] == null) {
                //计算 strA 去除最后一位后和 strB 的最大子序列
                result[lenA - 2][lenB - 1] = maxCommonStr(strLenSub(strA), strB);
            }
            if (result[lenA - 1][lenB - 2] == null) {
                //计算 strB 去除最后一位后和 strA 的最大子序列
                result[lenA - 1][lenB - 2] = maxCommonStr(strA, strLenSub(strB));
            }
            //等于 result[lenA - 2][lenB-1]和 result[lenA-1][lenB - 2]中的最大数
            result[lenA - 1][lenB - 1] = max(result[lenA - 2][lenB - 1], result[lenA - 1][lenB - 2]);
        }
        return result[lenA - 1][lenB - 1];
    }
```

```
    /**
     * 使字符串去除最后一位，返回该新的字符串
     *
     * @param str
     * @return
     */
    public static String strLenSub(String str) {
        return str.substring(0, str.length() - 1);
    }

    /**
     * 比较两个字符串长度，返回最长字符串 当两个字符串长度相等时，返回任意字符串
     *
     * @param strA
     * @param strB
     * @return
     */
    public static String max(String strA, String strB) {
        if (strA == null && strB == null) {
            return "";
        } else if (strA == null) {
            return strB;
        } else if (strB == null) {
            return strA;
        }
        if (strA.length() > strB.length()) {
            return strA;
        } else {
            return strB;
        }
    }
}
```