Dynamic Programming

DP定义：

动态规划是分治思想的延伸，通俗一点来说就是大事化小，小事化无的艺术。

在将大问题化解为小问题的分治过程中，保存对这些小问题已经处理好的结果，并供后面处理更大规模的问题时直接使用这些结果。

动态规划具备了以下三个特点

1. 把原来的问题分解成了几个**相似的子问题。**
2. 所有的子问题都**只需要解决一次。**
3. **储存**子问题的解。

动态规划的本质，是对问题**状态的定义**和**状态转移方程的定义**(状态以及状态之间的递推关系)

动态规划问题一般从以下四个角度考虑：

1. 状态定义
2. 状态间的转移方程定义
3. 状态的初始化
4. 返回结果

状态定义的要求：**定义的状态一定要形成递推关系。**

一句话概括：三特点四要素两本质

适用场景：最大值/最小值, 可不可行, 是不是，方案个数

第1题 Fibonacci

- 难度：Easy
- 备注：斐波那契数列，出自《剑指offer》
- 题目描述

> 大家都知道斐波那契数列，现在要求输入一个整数n，请你输出斐波那契数列的第n项（从0开始，第0项为0）。
>
> n<=39
>
> int Fibonacci(int n)
>
> 来源：牛客-剑指offer

答案-C/C++

```
/*
by 周荣
```

```
斐波那契数列定义：F(n)=F(n-1)+F(n-2)（n>=2，n∈N*），其中F(1)=1，F(2)=1

方法一：递归
*/
class Solution{
public:
    int Fibonacci(int n){
        // 初始值
        if (n <= 0){
            return 0;
        }
        if (n == 1 || n == 2) {
            return 1;
        }
        // F(n)=F(n-1)+F(n-2)
        return Fibonacci(n - 2) + Fibonacci(n - 1);
    }
};
/*
递归的方法时间复杂度为O(2^n)，随着n的增大呈现指数增长，效率低下
当输入比较大时，可能导致栈溢出
在递归过程中有大量的重复计算
*/
```

```
/*
方法二：动态规划
    状态：F(n)
    状态递推：F(n)=F(n-1)+F(n-2)
    初始值：F(1)=F(2)=1
    返回结果：F(N)
*/
class Solution2{
public:
    int Fibonacci(int n){
        // 初始值
        if (n <= 0){
            return 0;
        }
        if (n == 1 || n == 2) {
            return 1;
        }
        // 申请一个数组，保存子问题的解，题目要求从第0项开始
        int* record = new int[n + 1];
        record[0] = 0;
        record[1] = 1;
        for (int i = 2; i <= n; i++){
            // F(n)=F(n-1)+F(n-2)
            record[i] = record[i - 1] + record[i - 2];
        }
        return record[n];
        delete[] record;
    }
```

```
};
/*
上述解法的空间复杂度为O(n)
其实F(n)只与它相邻的前两项有关，所以没有必要保存所有子问题的解
只需要保存两个子问题的解就可以
下面方法的空间复杂度将为O(1)
*/

class Solution3{
public:
    int Fibonacci(int n){
        // 初始值
        if (n <= 0){
            return 0;
        }
        if (n == 1 || n == 2) {
            return 1;
        }
        int fn1 = 1;
        int fn2 = 1;
        int result = 0;
        for (int i = 3; i <= n; i++){
            // F(n)=F(n-1)+F(n-2)
            result = fn2 + fn1;
            // 更新值
            fn1 = fn2;
            fn2 = result;
        }
        return result;
    }
};

/*java*/
/*
方法一：递归
*/
public class Solution {
    public int Fibonacci(int n) {
        // 初始值
        if(n <= 0)
            return 0;
        if(n == 1 || n == 2)
            return 1;
        // F(n)=F(n-1)+F(n-2)
        return Fibonacci(n - 1) + Fibonacci(n - 2);
    }
}


/*
方法二：动态规划
*/
```

```java
public class Solution {
    public int Fibonacci(int n) {
        // 初始值
        if(n <= 0)
            return 0;
        // 申请一个数组，保存子问题的解，题目要求从第0项开始
        int[] array = new int[n + 1];
        array[0] = 0;
        array[1] = 1;
        for(int i = 2; i <= n; ++i)
        {
            // F(n)=F(n-1)+F(n-2)
            array[i] = array[i - 1] + array[i - 2];
        }
        return array[n];
    }
}
```

```java
public class Solution {
    public int Fibonacci(int n) {
        // 初始值
        if(n <= 0)
            return 0;
        if(n == 1 || n == 2)
            return 1;
        int ret = 0;
        int fn1 = 1, fn2 = 1;

        for(int i = 3; i <= n; ++i)
        {
            // F(n)=F(n-1)+F(n-2)
            ret = fn1 + fn2;
            // 更新值
            fn2 = fn1;
            fn1 = ret;
        }
        return ret;
    }
}
```


第2题 字符串分割(Word Break)

- 难度：Medium
- 备注：出自leetcode
- 题目描述

> Given a string *s* and a dictionary of words *dict*, determine if *s* can be segmented into a space-separated sequence of one or more dictionary words.
>
> For example, given *s* ="leetcode", *dict* =["leet", "code"].
>
> Return true because"leetcode"can be segmented as"leet code".

> bool wordBreak(string s, unordered_set &dict)
>
> 来源：牛客-leetcode

答案-C/C++

```
/*
by 周荣

题目描述
  给定一个字符串和一个词典dict，确定s是否可以根据词典中的词分成
  一个或多个单词。
  比如，给定
  s = "leetcode"
  dict = ["leet", "code"]
  返回true，因为"leetcode"可以被分成"leet code"

方法：动态规划
状态：
  子状态：前1，2，3，...,n个字符能否根据词典中的词被成功分词
  F(i)：前i个字符能否根据词典中的词被成功分词
状态递推：
  F(i)：true{j <i && F(j) && substr[j+1,i]能在词典中找到} OR false
  在j小于i中，只要能找到一个F(j)为true，并且从j+1到i之间的字符能在词典
  中找到，则F(i)为true
初始值：
  对于初始值无法确定的，可以引入一个不代表实际意义的空状态，作为状态的起始
  空状态的值需要保证状态递推可以正确且顺利的进行，到底取什么值可以通过简单
  的例子进行验证
  F(0) = true
返回结果：F(n)
*/

class Solution{
public:
    bool wordBreak(string s, unordered_set<string> &dict){
        if (s.empty()){
            return false;
        }
        if (dict.empty()){
            return false;
        }

        vector<bool> can_break(s.size() + 1, false);
        // 初始化F(0) = true
        can_break[0] = true;

        for (int i = 1; i <= s.size(); i++){
            for (int j = i - 1; j >= 0; j--){
                // F(i): true{j <i && F(j) && substr[j+1,i]能在词典中找到} OR false
                // 第j+1个字符的索引为j
                if (can_break[j] && dict.find(s.substr(j, i - j)) != dict.end()){
                    can_break[i] = true;
```

```
                    break;
                }
            }
        }
        return can_break[s.size()];

    }
};

/*java*/
import java.util.Set;
public class Solution {
    public boolean wordBreak(String s, Set<String> dict) {
        boolean[] canBreak = new boolean[s.length() + 1];
        // 初始化F(0) = true
        canBreak[0] = true;
        for(int i = 1; i <= s.length(); ++i){
            for(int j = i - 1; j >= 0; --j){
                // F(i): true{j <i && F(j) && substr[j+1,i]能在词典中找到} OR false
                // 第j+1个字符的索引为j
                if(canBreak[j] && dict.contains(s.substring(j,i))){
                    canBreak[i] = true;
                    break;
                }
            }
        }
        return canBreak[s.length()];
    }
}
```

第3题 三角矩阵(Triangle)

- 难度：Medium
- 备注：出自leetcode
- 题目描述

> Given a triangle, find the minimum path sum from top to bottom. Each step you may move to adjacent numbers on the row below.
>
> For example, given the following triangle
>
> ```
> [
>   [2],
>  [3,4],
> [6,5,7],
> [4,1,8,3]
> ]
> ```
>
> The minimum path sum from top to bottom is11(i.e., 2 + 3 + 5 + 1 = 11).
>
> int minimumTotal(vector<vector> &triangle)
>
> 来源：牛客-leetcode

答案-C/C++

```
/*
by 周荣
题目描述:
  给定一个三角矩阵，找出自顶向下的最短路径和，每一步可以移动到下一行的相邻数字。
  比如给定下面一个三角矩阵，自顶向下的最短路径和为11。

方法：动态规划
状态:
  子状态：从(0,0)到(1,0),(1,1),(2,0),...(n,n)的最短路径和
  F(i,j)：从(0,0)到(i,j)的最短路径和
状态递推:
  F(i,j) = min( F(i-1, j-1), F(i-1, j)) + triangle[i][j]
初始值:
  F(0,0) = triangle[0][0]
返回结果:
  min(F(n-1, i))
*/

class Solution {
public:
    int minimumTotal(vector<vector<int>> &triangle) {
        if (triangle.empty()){
            return 0;
        }
        // F[i][j], F[0][0]初始化
        vector<vector<int>> min_sum(triangle);
        int line = triangle.size();
        for (int i = 1; i < line; i++){
            for (int j = 0; j <= i; j++){
                // 处理左边界和右边界
                if (j == 0){
                    min_sum[i][j] = min_sum[i - 1][j];
                }
                else if (j == i){
                    min_sum[i][j] = min_sum[i - 1][j - 1];
                }
                else{
                    min_sum[i][j] = min(min_sum[i - 1][j], min_sum[i - 1][j - 1]);
                }
                // F(i,j) = min( F(i-1, j-1), F(i-1, j)) + triangle[i][j]
                min_sum[i][j] = min_sum[i][j] + triangle[i][j];
            }
        }

        int result = min_sum[line - 1][0];
        // min(F(n-1, i))
        for (int i = 1; i < line; i++){
            result = min(min_sum[line - 1][i], result);
        }
        return result;
    }
```

```
};

/*
方法二：动态规划（反向思维）
状态：
  子状态：从(n,n),(n,n-1),...(1,0),(1,1),(0,0)到最后一行的最短路径和
  F(i,j)：从(i,j)到最后一行的最短路径和
状态递推：
  F(i,j) = min( F(i+1, j), F(i+1, j+1)) + triangle[i][j]
初始值：
  F(n-1,0) = triangle[n-1][0], F(n-1,1) = triangle[n-1][1],..., F(n-1,n-1) = triangle[n-1]
[n-1]
返回结果：
  F(0, 0)

这种逆向思维不需要考虑边界，也不需要最后寻找最小值，直接返回F(0,0)即可
*/

class Solution2 {
public:
    int minimumTotal(vector<vector<int>> &triangle) {
        if (triangle.empty()){
            return 0;
        }
        // F[n-1][n-1],...F[n-1][0]初始化
        vector<vector<int>> min_sum(triangle);
        int line = triangle.size();
        // 从倒数第二行开始
        for (int i = line - 2; i >= 0; i--){
            for (int j = 0; j <= i; j++){
                // F(i,j) = min( F(i+1, j), F(i+1, j+1)) + triangle[i][j]
                min_sum[i][j] = min(min_sum[i + 1][j], min_sum[i + 1][j + 1]) + triangle[i]
[j];
            }
        }

        return min_sum[0][0];
    }
};

/*java*/
public class Solution {
    public int minimumTotal(ArrayList<ArrayList<Integer>> triangle) {
        if(triangle.isEmpty())
            return 0;
        List<List<Integer>> minPathSum = new ArrayList<>();
        for(int i = 0; i < triangle.size(); ++i) {
            minPathSum.add(new ArrayList<>());
        }
        // F[0][0]初始化
        minPathSum.get(0).add(triangle.get(0).get(0));
        for(int i = 1; i < triangle.size(); ++i) {
            int curSum = 0;
```

```
            for(int j = 0; j <= i; ++j) {
                // 处理左边界和右边界
                if(j == 0) {
                    curSum = minPathSum.get(i - 1).get(0);
                }
                else if(j == i){
                    curSum = minPathSum.get(i - 1).get(j - 1);
                }
                else{
                    curSum = Math.min(minPathSum.get(i - 1).get(j),
                            minPathSum.get(i - 1).get(j - 1));
                }
                // F(i,j) = min( F(i-1, j-1), F(i-1, j)) + triangle[i][j]
                minPathSum.get(i).add(triangle.get(i).get(j) + curSum);
            }
        }
        int size = triangle.size();
        // min(F(n-1, i))
        int allMin = minPathSum.get(size - 1).get(0);
        for(int i = 1; i < size; ++i)
        {
            allMin = Math.min(allMin,minPathSum.get(size - 1).get(i));
        }
        return allMin;
    }
}

/*
方法二：动态规划（反向思维）
状态：
  子状态：从(n,n),(n,n-1),...(1,0),(1,1),(0,0)到最后一行的最短路径和
  F(i,j)：从(i,j)到最后一行的最短路径和
状态递推：
  F(i,j) = min( F(i+1, j), F(i+1, j+1)) + triangle[i][j]
初始值：
  F(n-1,0) = triangle[n-1][0], F(n-1,1) = triangle[n-1][1],..., F(n-1,n-1) = triangle[n-1]
[n-1]
返回结果：
  F(0, 0)

这种逆向思维不需要考虑边界，也不需要最后寻找最小值，直接返回F(0,0)即可
*/
import java.util.List;
import java.util.ArrayList;
public class Solution {
    public int minimumTotal(ArrayList<ArrayList<Integer>> triangle) {
        if(triangle.isEmpty())
            return 0;
        // F[n-1][n-1],...F[n-1][0]初始化
        ArrayList<ArrayList<Integer>> minPathSum = new ArrayList<>(triangle);
        int row = minPathSum.size();
        // 从倒数第二行开始
        for(int i = row - 2; i >= 0; --i){
```

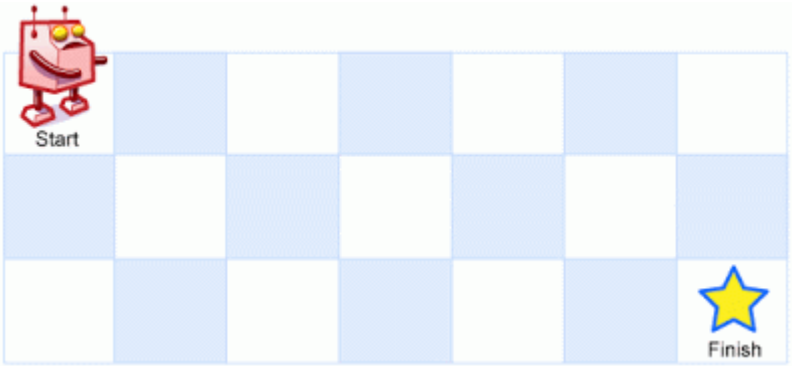
```
            for(int j = 0; j <= i; ++j){
                // F(i,j) = min( F(i+1, j), F(i+1, j+1)) + triangle[i][j]
                int curSum = Math.min(triangle.get(i + 1).get(j),
                        triangle.get(i + 1).get(j + 1))
                + triangle.get(i).get(j);
                minPathSum.get(i).set(j, curSum);
            }
        }
        return minPathSum.get(0).get(0);
    }
}

/*
注:
  易错点:只保留每一步的最小值,忽略其他路径,造成最终结果错误
          局部最小不等于全局最小

总结:
  遇到关于矩阵,网格,字符串间的比较,匹配的问题,
  单序列(一维)动规解决不了的情况下,
  就需要考虑双序列(二维)动规
*/
```

第4题 路径总数(Unique Paths)

- 难度:Easy
- 备注:出自leetcode
- 题目描述

> 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?
>
> 
>
>
> Above is a 3 x 7 grid. How many possible unique paths are there?
>
> **Note:** *m* and *n* will be at most 100.
>
> int uniquePaths(int m, int n)

> 来源：牛客-leetcode

答案-C/C++

```
/*
by 周荣
题目描述:
  在一个m*n的网格的左上角有一个机器人，机器人在任何时候只能向下或者向右移动，
  机器人试图到达网格的右下角，有多少可能的路径。

方法：动态规划
状态:
  子状态：从(0,0)到达(1,0),(1,1),(2,1),...(m-1,n-1)的路径数
  F(i,j)：从(0,0)到达F(i,j)的路径数
状态递推:
  F(i,j) = F(i-1,j) + F(i,j-1)
初始化:
  特殊情况：第0行和第0列
  F(0,i) = 1
  F(i,0) = 1
返回结果:
  F(m-1,n-1)
*/

class Solution {
public:

    int uniquePaths(int m, int n) {
        if (m < 1 || n < 1) {
            return 0;
        }

        // 申请F(i,j)空间，初始化
        vector<vector<int> > ret(m, vector<int>(n, 1));

        for (int i = 1; i < m; ++i) {
            for (int j = 1; j < n; ++j) {
                // F(i,j) = F(i-1,j) + F(i,j-1)
                ret[i][j] = ret[i - 1][j] + ret[i][j - 1];
            }
        }

        return ret[m - 1][n - 1];
    }
};

/*java*/
public class Solution {
    public int uniquePaths(int m, int n) {
        List<List<Integer>> pathNum = new ArrayList<>();
        // 申请F(i,j)空间，初始化
        for(int i = 0; i < m; ++i){
            pathNum.add(new ArrayList<>());
```

```
            pathNum.get(i).add(1);
        }
        for(int i = 1; i < n; ++i){
            pathNum.get(0).add(1);
        }
        for(int i = 1; i < m; ++i){
            for(int j = 1; j < n; ++j){
                // F(i,j) = F(i-1,j) + F(i,j-1)
                pathNum.get(i).add(pathNum.get(i).get(j - 1)
                + pathNum.get(i - 1).get(j));
            }
        }
        return pathNum.get(m - 1).get(n - 1);

    }
}
```

第5题 最小路径和(Minimum Path Sum)

- 难度：Medium
- 备注：出自leetcode
- 题目描述

> 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.
>
> int minPathSum(vector<vector > &grid)
>
> 来源：牛客-leetcode

答案-C/C++

```
/*
by 周荣
题目描述：
  给定一个m*n的网格，网格用非负数填充，找到一条从左上角到右下角的最短路径。
  注：每次只能向下或者向右移动。
方法：动态规划
状态：
  子状态：从(0,0)到达(1,0),(1,1),(2,1),...(m-1,n-1)的最短路径
  F(i,j)：从(0,0)到达F(i,j)的最短路径
状态递推：
  F(i,j) = min{F(i-1,j) , F(i,j-1)} + (i,j)
初始化：
  F(0,0) = (0,0)
  特殊情况：第0行和第0列
  F(0,i) = F(0,i-1) + (0,i)
  F(i,0) = F(i-1,0) + (i,0)
返回结果：
  F(m-1,n-1)
*/
```

```
class Solution {
public:

    int minPathSum(vector<vector<int> > &grid) {
        // 如果为空或者只有一行，返回0
        if (grid.empty() || grid[0].empty()) {
            return 0;
        }
        // 获取行和列大小
        const int M = grid.size();
        const int N = grid[0].size();
        // F(i,j)
        vector<vector<int> > ret(M, vector<int>(N, 0));

        // F(0,0), F(0,i), F(i,0)初始化
        ret[0][0] = grid[0][0];
        for (int i = 1; i != M; ++i) {
            ret[i][0] = grid[i][0] + ret[i - 1][0];
        }
        for (int i = 1; i != N; ++i) {
            ret[0][i] = grid[0][i] + ret[0][i - 1];
        }

        // F(i,j) = min{F(i-1,j) , F(i,j-1)} + (i,j)
        for (int i = 1; i < M; ++i) {
            for (int j = 1; j < N; ++j) {
                ret[i][j] = grid[i][j] + min(ret[i - 1][j], ret[i][j - 1]);
            }
        }

        return ret[M - 1][N - 1];
    }
};

/*java*/
public class Solution {
    public int minPathSum(int[][] grid) {

        int row = grid.length;
        int col = grid[0].length;
        //// 如果为空或者只有一行，返回0
        if(row == 0 || col == 0) {
            return 0;
        }
        // F(0,0), F(0,i), F(i,0)初始化
        for(int i = 1;i < row;i++) {
            grid[i][0] = grid[i - 1][0] + grid[i][0];
        }
        for(int i = 1;i < col;i++) {
            grid[0][i] = grid[0][i - 1] + grid[0][i];
        }
        // F(i,j) = min{F(i-1,j) , F(i,j-1)} + (i,j)
```

```
        for(int i = 1;i < row;i++) {
            for(int j = 1;j < col;j++) {
                grid[i][j] = Math.min(grid[i - 1][j],grid[i][j - 1]) + grid[i][j];
            }
        }
        return grid[row - 1][col - 1];
    }
}
```

第6题 背包问题

- 题目描述

> 有 `n` 个物品和一个大小为 `m` 的背包. 给定数组 `A` 表示每个物品的大小和数组 `V` 表示每个物品的价值.
>
> 问最多能装入背包的总价值是多大?

> 背包问题

```
/*
状态:
    F(i, j): 前i个物品放入大小为j的背包中所获得的最大价值
状态递推: 对于第i个商品，有一种例外，装不下，两种选择，放或者不放
    如果装不下: 此时的价值与前i-1个的价值是一样的
    F(i,j) = F(i-1,j)
    如果可以装入: 需要在两种选择中找最大的
    F(i, j) = max{F(i-1,j), F(i-1, j - A[i]) + V[i]}
    F(i-1,j): 表示不把第i个物品放入背包中， 所以它的价值就是前i-1个物品放入大小为j的背包的最大价值
    F(i-1, j - A[i]) + V[i]: 表示把第i个物品放入背包中，价值增加V[i],但是需要腾出j - A[i]的大小放第i
个商品
初始化: 第0行和第0列都为0，表示没有装物品时的价值都为0
    F(0,j) = F(i,0) = 0
返回值: F(n,m)
*/

class Solution {
public:
    /**
    * @param m: An integer m denotes the size of a backpack
    * @param A: Given n items with size A[i]
    * @param V: Given n items with value V[i]
    * @return: The maximum value
    */
    int backPackII(int m, vector<int> A, vector<int> V) {
        if (A.empty() || V.empty() || m < 1) {
            return 0;
        }
        //多加一行一列，用于设置初始条件
        const int N = A.size() + 1;
        const int M = m + 1;
        vector<vector<int> > result;
        result.resize(N);
        //初始化所有位置为0，第一行和第一列都为0，初始条件
```

```
        for (int i = 0; i != N; ++i) {
            result[i].resize(M, 0);
        }

        for (int i = 1; i < N; ++i) {
            for (int j = 1; j != M; ++j) {
                //第i个商品在A中对应的索引为i-1: i从1开始
                //如果第i个商品大于j,说明放不下， 所以(i,j)的最大价值和(i-1,j)相同
                if (A[i - 1] > j) {
                    result[i][j] = result[i - 1][j];
                }
                //如果可以装下，分两种情况，装或者不装
                //如果不装，则即为(i-1, j)
                //如果装，需要腾出放第i个物品大小的空间： j - A[i-1],装入之后的最大价值即为(i - 1, j
- A[i-1]) + 第i个商品的价值V[i - 1]
                //最后在装与不装中选出最大的价值
                else {
                    int newValue = result[i - 1][j - A[i - 1]] + V[i - 1];
                    result[i][j] = max(newValue, result[i - 1][j]);
                }
            }
        }
        //返回装入前N个商品，物品大小为m的最大价值
        return result[N - 1][m];
    }
};

/*
优化算法:
上面的算法在计算第i行元素时，只用到第i-1行的元素，所以二维的空间可以优化为一维空间
但是如果是一维向量，需要从后向前计算，因为后面的元素更新需要依靠前面的元素未更新（模拟二维矩阵的上一行的值）
的值
*/

class Solution {
public:
    /**
    * @param m: An integer m denotes the size of a backpack
    * @param A: Given n items with size A[i]
    * @param V: Given n items with value V[i]
    * @return: The maximum value
    */
    int backPackII(int m, vector<int> A, vector<int> V) {
        if (A.empty() || V.empty() || m < 1) {
            return 0;
        }

        const int N = A.size();
        //多加一列，用于设置初始条件，因为第一件商品要用到前面的初始值
        const int M = m + 1;
        vector<int> result;
        //初始化所有位置为0，第一行都为0，初始条件
        result.resize(M, 0);
```

```
        //这里商品的索引位置不需要偏移，要和未优化的方法区分开
        //这里的i-1理解为上一行，或者未更新的一维数组值
        for (int i = 0; i != N; ++i) {
            for (int j = M - 1; j > 0; --j) {
                //如果第i个商品大于j,说明放不下， 所以(i,j)的最大价值和(i-1,j)相同
                if (A[i] > j) {
                    result[j] = result[j];
                }
                //如果可以装下，分两种情况，装或者不装
                //如果不装，则即为(i-1, j)
                //如果装，需要腾出放第i个物品大小的空间： j - A[i],装入之后的最大价值即为(i - 1, j - A[i-1]) + 第i个商品的价值V[i]
                //最后在装与不装中选出最大的价值
                else {
                    int newValue = result[j - A[i]] + V[i];
                    result[j] = max(newValue, result[j]);
                }
            }
        }
        //返回装入前N个商品，物品大小为m的最大价值
        return result[m];
    }
};

/*java*/
public class Solution {
    /**
     * @param m: An integer m denotes the size of a backpack
     * @param A: Given n items with size A[i]
     * @param V: Given n items with value V[i]
     * @return: The maximum value
     */
    public int backPackII(int m, int[] A, int[] V) {
        // write your code here
        int num = A.length;
        if(m == 0 || num == 0)
            return 0;
        //多加一行一列，用于设置初始条件
        int[][] maxValue = new int[num + 1][m + 1];
        //初始化所有位置为0，第一行和第一列都为0，初始条件
        for(int i = 0; i <= num; ++i){
            maxValue[i][0] = 0;
        }
        for(int i = 1; i <= m; ++i){
            maxValue[0][i] = 0;
        }
        for(int i = 1; i <= num; ++i){
            for(int j = 1; j <= m; ++j){
                //第i个商品在A中对应的索引为i-1: i从1开始
                //如果第i个商品大于j,说明放不下， 所以(i,j)的最大价值和(i-1,j)相同
                if(A[i - 1] > j){
                    maxValue[i][j] = maxValue[i - 1][j];
                }
```

```
                else{
                    //如果可以装下，分两种情况，装或者不装
                    //如果不装，则即为(i-1, j)
                    //如果装，需要腾出放第i个物品大小的空间： j - A[i-1],装入之后的最大价值即为(i -
1, j - A[i-1]) + 第i个商品的价值V[i - 1]
                    //最后在装与不装中选出最大的价值
                    int newValue = maxValue[i - 1][j - A[i - 1]]
                    + V[i - 1];
                    maxValue[i][j] = Math.max(newValue
                    , maxValue[i - 1][j]);
                }
            }
        }
        //返回装入前N个商品，物品大小为m的最大价值
        return maxValue[num][m];
    }
}

//优化算法
public class Solution {
    /**
     * @param m: An integer m denotes the size of a backpack
     * @param A: Given n items with size A[i]
     * @param V: Given n items with value V[i]
     * @return: The maximum value
     */
    public int backPackII(int m, int[] A, int[] V) {
        // write your code here
        int num = A.length;
        if(m == 0 || num == 0)
            return 0;
        //多加一列，用于设置初始条件，因为第一件商品要用到前面的初始值
        int[] maxValue = new int[m + 1];
        //初始化所有位置为0，第一行都为0，初始条件
        for(int i = 0; i <= m; ++i){
            maxValue[i] = 0;
        }
        for(int i = 1; i <= num; ++i){
            for(int j = m; j > 0; --j){
                //如果第i个商品大于j,说明放不下， 所以(i,j)的最大价值和(i-1,j)相同
                //如果可以装下，分两种情况，装或者不装
                //如果不装，则即为(i-1, j)
                //如果装，需要腾出放第i个物品大小的空间： j - A[i],装入之后的最大价值即为(i - 1, j -
A[i-1]) + 第i个商品的价值V[i]
                //最后在装与不装中选出最大的价值
                if(A[i - 1] <= j){
                    int newValue = maxValue[j - A[i - 1]] + V[i - 1];
                    maxValue[j] = Math.max(newValue, maxValue[j]);
                }
            }
        }
        //返回装入前N个商品，物品大小为m的最大价值
        return maxValue[m];
```

```
    }
}
```

第7题 回文串分割(Palindrome Partitioning)

- 难度：Hard
- 备注：回文串知识，出自leetcode
- 题目描述

> 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", Return1since the palindrome partitioning["aa","b"]could be produced using 1 cut.
>
> int minCut(string s)
>
> 来源：牛客-leetcode

答案-C/C++

```
/*
by 周荣
回文串：正读和反读都一样的字符串，比如noon，level，字符串左右对称
题目描述：
  给定一个字符串 s,把 s 分割成一系列的子串，分割的每一个子串都为回文串
  返回最小的分割次数
  比如，给定 s = "aab",
  返回1，因为一次cut就可以产生回文分割["aa","b"]

方法：动态规划
状态：
  子状态：到第1,2,3,...,n个字符需要的最小分割数
  F(i)：到第i个字符需要的最小分割数
状态递推：
  F(i) = min{F(i), 1 + F(j)}, where j<i && j+1到i是回文串
  上式表示如果从j+1到i判断为回文字符串，且已经知道从第1个字符
  到第j个字符的最小切割数，那么只需要再切一次，就可以保证
  1-->j, j+1-->i都为回文串。
初始化：
  F(i) = i - 1
  上式表示到第i个字符需要的最大分割数
  比如单个字符只需要切0次，因为单子符都为回文串
  2个字符最大需要1次，3个2次......
返回结果：
  F(n)
遗留问题：如何判断一段字符串为回文串
  循环判断首尾元素是否相同，如果全部相同，则是回文串
*/
```

```
class Solution {
public:
    int minCut(string s) {
        if (s.empty()) return 0;

        int len = s.size();
        vector<int> cut;
        // F(i)初始化
        // F(0) = -1，必要项，如果没有这一项，对于重叠字符串“aaaaa”会产生错误的结果
        for (int i = 0; i < 1 + len; ++i) {
            cut.push_back(i - 1);
        }

        for (int i = 1; i < 1 + len; ++i) {
            for (int j = 0; j < i; ++j) {
                  // F(i) = min{F(i), 1 + F(j)}, where j<i && j+1到i是回文串
                // 从最长串判断，如果从第j+1到i为回文字符串
                // 则再加一次分割，从1到j，j+1到i的字符就全部分成了回文字符串
                if (isPalindrome(s, j, i - 1)) {
                    cut[i] = min(cut[i], 1 + cut[j]);
                }
            }
        }

        return cut[len];
    }

    //判断是否回文串
    bool isPalindrome(string s, int i, int j){
        while (i<j) {
            if (s[i] != s[j]){
                return false;
            }
            i++;
            j--;
        }
        return true;
    }
};

/*
上述方法两次循环时间复杂度是O(n^2)，
判断回文串时间复杂度是O(n)，
所以总时间复杂度为O(n^3)
对于过长的字符串，在OJ的时候会出现TLE(Time Limit Exceeded)

判断回文串的方法可以继续优化，使总体时间复杂度将为O(n^2)
判断回文串，这是一个“是不是”的问题，所以也可以用动态规划来实现
判断回文串：动态规划
状态：
  子状态：从第一个字符到第二个字符是不是回文串，第1-3，第2-5，...
  F(i,j)：字符区间 [i,j] 是否为回文串
状态递推：
```

```
  F(i,j): true->{s[i]==s[j] && F(i+1,j-1)} OR false
  上式表示如果字符区间首尾字符相同且在去掉区间首尾字符后字符区间仍为回文串，
  则原字符区间为回文串
  从递推公式中可以看到第i处需要用到第i+1处的信息，所以i应该从字符串末尾遍历
初始化：
  F(i,j) = false
返回结果：
  矩阵F(n,n)，只更新一半值（i <= j），n^2 / 2
*/

class Solution2 {
public:
    int minCut(string s) {
        if (s.empty()) return 0;

        int len = s.size();
        vector<int> cut;
        // F(i)初始化
        // F(0) = -1，必要项，如果没有这一项，对于重叠字符串“aaaaa”会产生错误的结果

        for (int i = 0; i < 1 + len; ++i) {
            cut.push_back(i - 1);
        }
        vector<vector<bool> > mat = getMat(s);

        for (int i = 1; i < 1 + len; ++i) {
            for (int j = 0; j < i; ++j) {
                // F(i) = min{F(i), 1 + F(j)}，where j<i && j+1到i是回文串
                // 从最长串判断，如果从第j+1到i为回文字符串
                // 则再加一次分割，从1到j，j+1到i的字符就全部分成了回文字符串
                if (mat[j][i - 1]) {
                    cut[i] = min(cut[i], 1 + cut[j]);
                }
            }
        }

        return cut[len];
    }

    vector<vector<bool> > getMat(string s) {
        int len = s.size();
        vector<vector<bool> > mat = vector<vector<bool> >(len, vector<bool>(len, false));
        for (int i = len - 1; i >= 0; --i) {
            for (int j = i; j < len; ++j) {
                if (j == i) {
                    // 单字符为回文字符串
                    mat[i][j] = true;
                }
                else if (j == i + 1) {
                    // 相邻字符如果相同，则为回文字符串
                    mat[i][j] = (s[i] == s[j]);
                }
                else {
```

```
                    // F(i,j) = {s[i]==s[j] && F(i+1,j-1)
                    // j > i+1
                    mat[i][j] = ((s[i] == s[j]) && mat[i + 1][j - 1]);
                }
            }
        }

        return mat;
    }

};

/*java*/
public class Solution {
    //判断是否回文串
    public boolean isPal(String s, int start, int end){
        while(start < end){
            if(s.charAt(start) != s.charAt(end))
                return false;
            ++start;
            --end;
        }
        return true;
    }
    public int minCut(String s) {
        int len = s.length();
        if(len == 0)
            return 0;
        int[] minCut = new int[len + 1];
        // F(i)初始化
        // F(0) = -1，必要项，如果没有这一项，对于重叠字符串“aaaaa”会产生错误的结果
        for(int i = 0; i <= len; ++i){
            minCut[i] = i - 1;
        }
        for(int i = 1; i <= len; ++i){
            for(int j = 0; j < i; ++j){
                // F(i) = min{F(i), 1 + F(j)}, where j<i && j+1到i是回文串
                // 从最长串判断，如果从第j+1到i为回文字符串
                // 则再加一次分割，从1到j，j+1到i的字符就全部分成了回文字符串
                if(isPal(s, j, i - 1)){
                    minCut[i] = Math.min(minCut[i], minCut[j] + 1);
                }
            }
        }
        return minCut[len];
    }
}

/*
上述方法两次循环时间复杂度是O(n^2)，
判断回文串时间复杂度是O(n)，
所以总时间复杂度为O(n^3)
对于过长的字符串，在OJ的时候会出现TLE(Time Limit Exceeded)
```

判断回文串的方法可以继续优化，使总体时间复杂度将为O(n^2)
判断回文串，这是一个“是不是”的问题，所以也可以用动态规划来实现
判断回文串：动态规划
状态：
  子状态：从第一个字符到第二个字符是不是回文串，第1-3，第2-5，...
  F(i,j)：字符区间 [i,j] 是否为回文串
状态递推：
  F(i,j)：true->{s[i]==s[j] && F(i+1,j-1)} OR false
  上式表示如果字符区间首尾字符相同且在去掉区间首尾字符后字符区间仍为回文串，
  则原字符区间为回文串
  从递推公式中可以看到第i处需要用到第i+1处的信息，所以i应该从字符串末尾遍历
初始化：
  F(i,j) = false
返回结果：
  矩阵F(n,n)，只更新一半值（i <= j），n^2 / 2
*/
```
public class Solution {
    public boolean[][] getMat(String s){
        int len = s.length();
        boolean[][] Mat = new boolean[len][len];
        for(int i = len - 1; i >= 0; --i){
            for(int j = i; j < len; ++j){
                if(j == i)
                    // 单字符为回文字符串
                    Mat[i][j] = true;
                else if(j == i + 1){
                    // 相邻字符如果相同，则为回文字符串
                    if(s.charAt(i) == s.charAt(j))
                        Mat[i][j] = true;
                    else
                        Mat[i][j] = false;
                }
                else{
                    // F(i,j) = {s[i]==s[j] && F(i+1,j-1)
                    // j > i+1
                    Mat[i][j] = (s.charAt(i) == s.charAt(j)) && Mat[i + 1][j - 1];
                }
            }
        }
        return Mat;
    }
    public int minCut(String s) {
        int len = s.length();
        if(len == 0)
            return 0;
        boolean[][] Mat = getMat(s);
        int[] minCut = new int[len + 1];
        / F(i)初始化
        // F(0) = -1，必要项，如果没有这一项，对于重叠字符串“aaaaa”会产生错误的结果
        for(int i = 0; i <= len; ++i){
            minCut[i] = i - 1;
        }
```

```
        for(int i = 1; i <= len; ++i){
            for(int j = 0; j < i; ++j){
                // F(i) = min{F(i), 1 + F(j)}, where j<i && j+1到i是回文串
                // 从最长串判断，如果从第j+1到i为回文字符串
                // 则再加一次分割，从1到j，j+1到i的字符就全部分成了回文字符串
                if(Mat[j][i - 1]){
                    minCut[i] = Math.min(minCut[i], minCut[j] + 1);
                }
            }
        }
        return minCut[len];
    }
}

/*
上述方法判断回文串时间复杂度O(n^2)
主方法两次循环时间复杂度O(n^2)
总体时间复杂度O(n^2) ~ O(2 * n^2) = O(n^2) + O(n^2)

总结:
   简单的动态规划问题，状态，状态递推和状态初始化都比较直观
   对于复杂的动态规划问题，状态，状态递推和状态初始化都比较隐含，需要仔细推敲
   尤其是状态递推可能需要额外的辅助判断条件才能达成。

*/
```

第8题 编辑距离(Edit Distance)

- 难度：Hard
- 备注：出自leetcode
- 题目描述

> Given two words *word1* and *word2*, find the minimum number of steps required to convert *word1* to *word2*. (each operation is counted as 1 step.)
>
> You have the following 3 operations permitted on a word:
>
> a) Insert a character b) Delete a character c) Replace a character
>
> int minDistance(string word1, string word2)
>
> 来源：牛客-leetcode

答案-C/C++

```
/*
by 周荣
题目描述:
   给定两个单词word1和word2，找到最小的修改步数，把word1转换成word2
   每一个操作记为一步
   允许在一个word上进行如下3种操作:
   a）插入一个字符
   b）删除一个字符
```

c）替换一个字符

编辑距离（Edit Distance）：
是指两个字串之间，由一个转成另一个所需的最少编辑操作次数。

方法：动态规划
状态：
子状态：word1的前1，2，3，...m个字符转换成word2的前1，2，3，...n个字符需要的编辑距离

```
/*
  F(i,j):word1的前i个字符于word2的前j个字符的编辑距离
状态递推：
  F(i,j) = min { F(i-1,j) +1, F(i,j-1) +1, F(i-1,j-1) +(w1[i]==w2[j]?0:1) }
  上式表示从删除，增加和替换操作中选择一个最小操作数
  F(i-1,j): w1[1,...,i-1]于w2[1,...,j]的编辑距离，删除w1[i]的字符--->F(i,j)
  F(i,j-1): w1[1,...,i]于w2[1,...,j-1]的编辑距离，增加一个字符--->F(i,j)
  F(i-1,j-1): w1[1,...,i-1]于w2[1,...,j-1]的编辑距离，如果w1[i]与w2[j]相同，
  不做任何操作，编辑距离不变，如果w1[i]与w2[j]不同，替换w1[i]的字符为w2[j]--->F(i,j)
初始化：
  初始化一定要是确定的值，如果这里不加入空串，初始值无法确定
  F(i,0) = i :word与空串的编辑距离，删除操作
  F(0,i) = i :空串与word的编辑距离，增加操作
返回结果：F(m,n)
*/

class Solution {
public:

    int minDistance(string word1, string word2) {
        // word与空串之间的编辑距离为word的长度
        if (word1.empty() || word2.empty()) {
            return max(word1.size(), word2.size());
        }

        int len1 = word1.size();
        int len2 = word2.size();
        // F(i,j)初始化
        vector<vector<int> >  f(1 + len1, vector<int>(1 + len2, 0));
        for (int i = 0; i <= len1; ++i) {
            f[i][0] = i;
        }
        for (int i = 0; i <= len2; ++i) {
            f[0][i] = i;
        }

        for (int i = 1; i <= len1; ++i) {
            for (int j = 1; j <= len2; ++j) {
                // F(i,j) = min { F(i-1,j) +1, F(i,j-1) +1, F(i-1,j-1) +(w1[i]==w2[j]?0:1) }
                // 判断word1的第i个字符是否与word2的第j个字符相等
                if (word1[i - 1] == word2[j - 1]) {
                    f[i][j] = 1 + min(f[i][j - 1], f[i - 1][j]);
                    // 字符相等，F(i-1,j-1)编辑距离不变
                    f[i][j] = min(f[i][j], f[i - 1][j - 1]);
```

```
                }
                else {
                    f[i][j] = 1 + min(f[i][j - 1], f[i - 1][j]);
                    // 字符不相等，F(i-1,j-1)编辑距离 + 1
                    f[i][j] = min(f[i][j], 1 + f[i - 1][j - 1]);
                }
            }
        }

        return f[len1][len2];
    }
};

/*java*/
public class Solution {
    public int minDistance(String word1, String word2) {
        // word与空串之间的编辑距离为word的长度
        if(word1.isEmpty() || word2.isEmpty())
            return Math.max(word1.length(), word2.length());
        int len1 = word1.length();
        int len2 = word2.length();
        int[][] minDis = new int[len1 + 1][len2 + 1];
        // F(i,j)初始化
        for(int i = 0; i <= len1; ++i){
            minDis[i][0] = i;
        }
        for(int i = 0; i <= len2; ++i){
            minDis[0][i] = i;
        }
        for(int i = 1; i <= len1; ++i){
            for(int j = 1; j <= len2; ++j){
                // F(i,j) = min { F(i-1,j) +1, F(i,j-1) +1, F(i-1,j-1) +(w1[i]==w2[j]?0:1) }

                minDis[i][j] = 1 + Math.min(minDis[i - 1][j]
                        , minDis[i][j - 1]);
                // 判断word1的第i个字符是否与word2的第j个字符相等
                if(word1.charAt(i - 1) == word2.charAt(j - 1)){
                    // 字符相等，F(i-1,j-1)编辑距离不变
                    minDis[i][j] = Math.min(minDis[i][j]
                    ,minDis[i - 1][j - 1]);
                }
                else{
                    // 字符不相等，F(i-1,j-1)编辑距离 + 1
                    minDis[i][j] = Math.min(minDis[i][j]
                            ,minDis[i - 1][j - 1] + 1);
                }
            }
        }
        return minDis[len1][len2];
    }
}

/*
```

```
注：字符串类的动态规划，可引入空串进行初始化
*/
```

第9题 不同子序列(Distinct Subsequences)

- 难度：Hard
- 备注：出自leetcode
- 题目描述

> Given a string **S** and a string **T**, count the number of distinct subsequences of **T** in **S**.
>
> A subsequence of a string is a new string which is formed from the original string by deleting some (can be none) of the characters without disturbing the relative positions of the remaining characters. (ie,"ACE"is a subsequence of"ABCDE"while"AEC"is not).
>
> Here is an example: **S** ="rabbbit", **T** ="rabbit"
>
> Return3.
>
> int numDistinct(string S, string T)
>
> 来源：牛客-leetcode

答案-C/C++

```
/*
by 周荣
题目描述：
  给定两个字符串S和T，求S有多少个不同的子串与T相同。
  S的子串定义为在S中任意去掉0个或者多个字符形成的串。
  子串可以不连续，但是相对位置不能变。
  比如“ACE”是“ABCDE”的子串，但是“AEC”不是。
问题翻译：S有多少个不同的子串与T相同
  S[1:m]中的子串与T[1:n]相同的个数
  由S的前m个字符组成的子串与T的前n个字符相同的个数
状态：
  子状态：由S的前1,2,...,m个字符组成的子串与T的前1,2,...,n个字符相同的个数
  F(i,j): S[1:i]中的子串与T[1:j]相同的个数
状态递推：
  在F(i,j)处需要考虑S[i] = T[j] 和 S[i] != T[j]两种情况
  当S[i] = T[j]:
    1>: 让S[i]匹配T[j]，则
        F(i,j) = F(i-1,j-1)
    2>: 让S[i]不匹配T[j],则问题就变为S[1:i-1]中的子串与T[1:j]相同的个数，则
        F(i,j) = F(i-1,j)
  故，S[i] = T[j]时，F(i,j) = F(i-1,j-1) + F(i-1,j)

  当S[i] != T[j]:
    问题退化为S[1:i-1]中的子串与T[1:j]相同的个数
  故，S[i] != T[j]时，F(i,j) = F(i-1,j)
初始化：引入空串进行初始化
  F(i,0) = 1 ---> S的子串与空串相同的个数，只有空串与空串相同
返回结果：
```

```
  F(m,n)
*/

class Solution {
public:

    int numDistinct(string S, string T) {
        int s_size = S.size();
        int t_size = T.size();
        // S的长度小于T长度，不可能含有与T相同的子串
        if (S.size() < T.size()) return 0;
        // T为空串，只有空串与空串相同，S至少有一个子串，它为空串
        if (T.empty()) return 1;

        // F(i,j)，初始化所有的值为0
        vector<vector<int> > f(s_size + 1, vector<int>(t_size + 1, 0));
        // 空串与空串相同的个数为1
        f[0][0] = 1;
        for (int i = 1; i <= s_size; ++i) {
            // F(i,0)初始化
            f[i][0] = 1;
            for (int j = 1; j <= t_size; ++j) {
                // S的第i个字符与T的第j个字符相同
                if (S[i-1] == T[j-1]) {
                    f[i][j] = f[i-1][j] + f[i-1][j-1];
                }
                else {
                    // S的第i个字符与T的第j个字符不相同
                    // 从S的前i-1个字符中找子串，使子串与T的前j个字符相同
                    f[i][j] = f[i-1][j];
                }
            }
        }

        return f[s_size][t_size];
    }
};

/*
此题也可优化空间复杂度为O(n)
f[i][j] 只和 f[i - 1][j], f[i - 1][j - 1]有关
类似于背包问题，可以用一维数组保存上一行的结果，每次从最后一列更新元素值
*/

int numDistinct(string S, string T) {
    if (S.empty())
        return 0;
    if (T.empty())
        return 1;

    int len1 = S.size();
    int len2 = T.size();
    vector<int> numDis(len2 + 1, 0);
```

```
    numDis[0] = 1;

    for (int i = 1; i <= len1; ++i)
    {
        for (int j = len2; j > 0; --j)
        {
            if (S[i - 1] == T[j - 1])
                numDis[j] = numDis[j - 1] + numDis[j];
            else
                numDis[j] = numDis[j];
        }
    }
    return numDis[len2];
}

/*java*/
public class Solution {
    public int numDistinct(String S, String T) {
        int sLen = S.length();
        int tLen = T.length();
        int[][] numDis = new int[sLen + 1][tLen + 1];
        numDis[0][0] = 1;
        // F(i,j)，初始化第一行剩余列的所有值为0
        for(int i = 1; i <= tLen; ++i){
            numDis[0][i] = 0;
        }
        //F(i, 0) = 1
        for(int i = 1; i <= sLen; ++i){
            numDis[i][0] = 1;
        }
        for(int i = 1; i <= sLen; ++i){
            for(int j = 1; j <= tLen; ++j){
                // S的第i个字符与T的第j个字符相同
                if(S.charAt(i - 1) == T.charAt(j - 1)){
                    numDis[i][j] = numDis[i - 1][j] + numDis[i - 1][j - 1];
                }
                else{
                    // S的第i个字符与T的第j个字符不相同
                    // 从S的前i-1个字符中找子串，使子串与T的前j个字符相同
                    numDis[i][j] = numDis[i - 1][j];
                }

            }
        }
        return numDis[sLen][tLen];
    }
}

/*
此题也可优化空间复杂度为O(n)
f[i][j] 只和 f[i - 1][j], f[i - 1][j - 1]有关
类似于背包问题，可以用一维数组保存上一行的结果，每次从最后一列更新元素值
*/
```

```
public class Solution {
    public int numDistinct(String S, String T) {
        int sLen = S.length();
        int tLen = T.length();
        int[] numDis = new int[tLen + 1];
        numDis[0]= 1;
        for(int i = 1; i <= tLen; ++i){
            numDis[i] = 0;
        }
        for(int i = 1; i <= sLen; ++i){
            for(int j = tLen; j > 0; --j){
                if(S.charAt(i - 1) == T.charAt(j - 1)){
                    numDis[j] = numDis[j] + numDis[j - 1];
                }
            }
        }
        return numDis[tLen];
    }
}
```