12 区间型动规

笔记本： DP Note
创建时间： 10/13/2019 3:15 PM **更新时间：** 12/27/2019 5:11 PM
作者： tanziqi1756@outlook.com

最长回文子序列，注意这不是最长回文子串，中心扩展法没有办法用了
Leetcode 516. Longest Palindromic Subsequence

516. Longest Palindromic Subsequence

难度 中等 129 收藏 分享 切换为中文 关注

题目描述 评论 (88) 题解(28) 提交记录

Given a string s, find the longest palindromic subsequence's length in s. You may assume that the maximum length of s is 1000.

Example 1:
Input:

```
"bbbab"
```

Output:

```
4
```

One possible longest palindromic subsequence is "bbbb".

Example 2:
Input:

```
"cbbd"
```

Output:

```
2
```

One possible longest palindromic subsequence is "bb".

动规：

最长回文子序列 subsequence leetcode 516

Dec 26, 2019 at 4:22 PM

Input: String s

N = s.length

1. 确定状态

f[i][j] 表示在区间 [i...j] 的最长回文子序列

2. 状态转移

i

x x b x ... x j b x x

f[i][j] = Math.max(

f[i+1][j-1] + 2 | s[i] == s[j],

f[i+1][j],

f[i][j-1])

3. 初始状态

当 i=j 时，单个字符组成回文串

当 j-i=1 时，单个或两个字符组成回文串

状态初始化后

i \	0	1	2	3	4
0	1	1/2	0	0	0
1	α	1	1/2	0	0
2	α	α	1	1/2	0
3	α	α	α	1	1/2
4	α	α	α	α	1

```java
class Solution {
    public int longestPalindromeSubseq(String ss) {
        char[] s = ss.toCharArray();
        int N = ss.length();
        // dynamic programming

        // state
        // dp[i][j] means
        // the length of the longenst palidromic subsequence of substring[i..j]
        int[][] f = new int[N][N];

        // initialize
        // length = 1
        for( int i = 0; i < N; i++ ) {
            f[i][i] = 1;  // one character forms a palindrome.
        }
        // i <= j
        // length = 2
        // when j -i = 1, two characters form a palindrome.
        for( int i = 0; i < N - 1; i++ ) {
            f[i][i+1] = (s[i] == s[i+1]) ? 2 : 1;
        }

        // state transfer
        // f[i][j] = max(f[i+1][j-1] + 2 when s[i] = s[j], f[i+1][j], f[i][j-1]);
        // length >= 3
        for( int len = 3; len <= N; len++ ) {
            for( int i = 0; i + len <= N; i++ ) {
                int j = i + len - 1;
                if( s[i] == s[j] ) {
                    f[i][j] = Math.max(f[i][j], f[i+1][j-1] + 2);
                }
                else {
                    f[i][j] = Math.max(f[i+1][j], f[i][j-1]);
                }
            }
        }
        return f[0][N-1];
    }
}
```

但是这样算还不够快，我们看记忆化搜索（自上而下 Top-down），
而前面的递推是用的是自下而上（Bottom-up)。

任何一种动规，都能用这两种方法写。

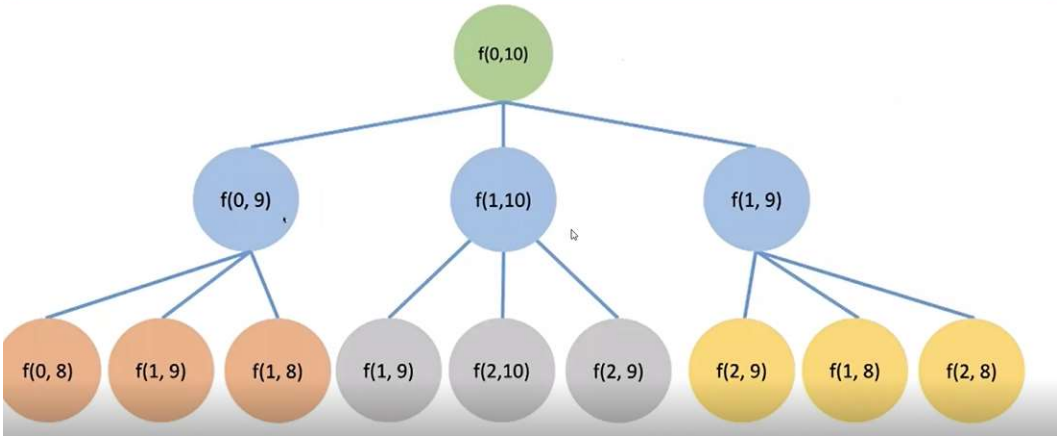
记忆化搜索方法

九章算法

- 动态规划编程的另一个选择
- f[i][j] = max{f[i+1][j], f[i][j-1], f[i+1][j-1] + 2|S[i]=S[j]}
- 计算f(0, N-1)
- **递归**计算f(1, N-1), f(0, N-2), f(1, N-2)
- **记忆化:** 计算f(i, j)结束后，将结果保存在数组f[i][j]里，下次如果需要再次计算f(i, j)，直接返回f[i][j]

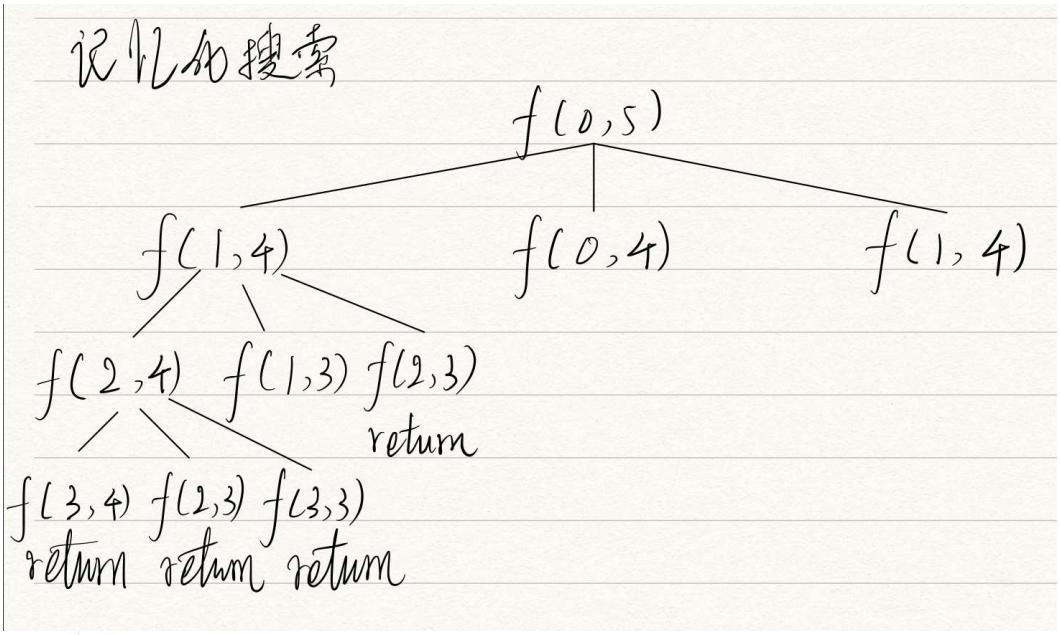
记忆化搜索方法

九章算法



存在大量重复计算，比如f(1,9)

- 递推方法自下而上：f[0], f[1], …, f[N]
- 记忆化方法自上而下：f(N), f(N-1), …
- 记忆化搜索编写一般比较简单
- 递推方法在某些条件下可以做空间优化，记忆化搜索则必须存储所有f值




```java
class Solution {

    int[][] memo;
    char[] s;

    private void topDownSearch(int i, int j) {
        if( memo[i][j] != 0 ) {
            return ;
        }
        if( i == j ) {
            memo[i][j] = 1;
            return ;
        }
        if( j - i == 1 ) {
            memo[i][j] = (s[i] == s[j]) ? 2 : 1;
            return ;
        }

        topDownSearch(i + 1, j);
        topDownSearch(i, j - 1);
        topDownSearch(i + 1, j - 1);

        memo[i][j] = Math.max(memo[i+1][j], memo[i][j-1]);
        if( s[i] == s[j] ) {
            memo[i][j] = Math.max(memo[i][j], memo[i+1][j-1] + 2);
        }
    }

    public int longestPalindromeSubseq(String ss) {
        int N = ss.length();
        memo = new int[N][N];
        s = ss.toCharArray();
        // clear memory
        /*for( int i = 0; i < N; i++ ) {
            for( int j = 0; j < N; j++ ) {
                memo[i][j] = -1;
            }
        }*/
        topDownSearch(0, N-1);
        return memo[0][N-1];
    }
}
```

516. Longest Palindromic Subsequence

难度 中等 129 收藏 分享 切换为中文 关注

题目描述 评论 (88) 题解(28) 提交记录

执行结果： 通过 显示详情 >

执行用时： 81 ms , 在所有 java 提交中击败了 5.30% 的用户

内存消耗： 47.9 MB , 在所有 java 提交中击败了 90.67% 的用户

必做

多选题 **一般来说, 对于同一个问题, 使用相同的动态规划思路(状态定义和状态转移方程), 不做任何优化的情况下, 递推(循环)与记忆化搜索(递归)的写法会有什么不同?**

A 时间复杂度不同

B 空间复杂度不同

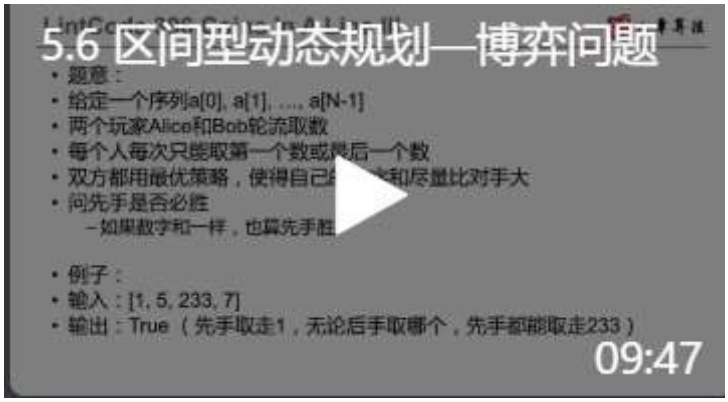
C 子问题被计算出来的顺序不同

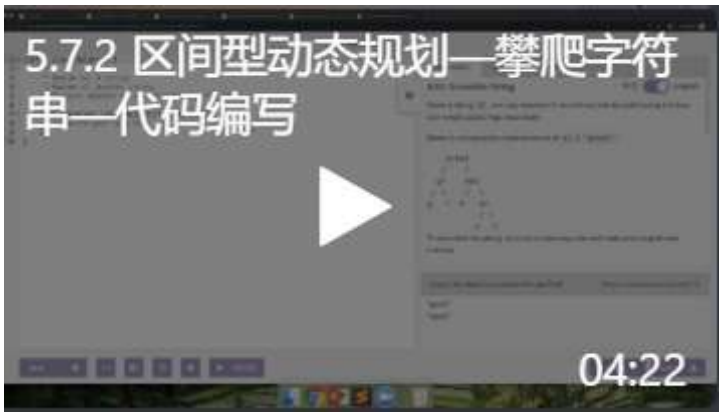
D 子问题初次被使用的顺序不同

我不会

正确答案是 D ， 有13%的同学做对了这道题目哦，继续努力！

解析： 一般来说, 动态规划的时间复杂度等于 状态的数量 * 状态转移的时间. 这一点很好理解, 就相当于我们需要计算的状态的数量乘以计算每个状态的时间. 无论是递推还是记忆化搜索, 使用相同的状态定义和状态转移方程, 所以时空复杂度相同. C属于迷惑选项, 即使是使用递归, 子问题被计算出来的顺序也是从小的状态到大的状态, 遵从状态转移方程. 另外, 递推的方法有时候可以做空间上的优化.





必做

单选题 **给你这样一个数组 [4, 1, 5, 10], 每次你可以拿走其中一个元素, 然后得到这个元素两边的元素以及它本身的乘积的分数. 那么最多能获得多少分呢?假定两端都有一个 1, 也就是说, 如果你第一个拿走 4, 你会得到 1 * 4 * 1 = 4 分.**

A 140

B 264

C 270

D 69

我不会

正确答案是 C ，有38%的同学答对了，要加油了。

解析：最优的策略就是先拿走 1, 得到 20 分, 剩下 [4, 5, 10]; 然后拿走 5, 得到 200 分, 剩下 [4, 10]; 然后拿走 4, 得到 40 分, 剩下 [10]; 最终拿走 10 得到 10 分. 共计 20 + 200 + 40 + 10 = 270分.

必做

多选题 **上面的问题是求解最优的拿走元素的顺序, 可以获得最多的分数. 这个问题可以用什么样的算法/思想解决呢?**

A 回溯法，枚举出所有顺序

B 动态规划

C 贪心

D 排序

我不会

正确答案是 A B ，有45%的同学答对了，加油赶上他们!

解析：枚举出所有的情况固然是可以的, 只不过时间复杂度变成了阶乘级别. 而动态规划可以在多项式时间复杂度内解决这个问题.

必做

多选题 **重述问题: 给定一个非负整数数组 arr, 每次可以拿走其中一个元素, 并且获得 左边的元素 * 该元素 * 右边的元素 的分数. 最多可以得到多少分数? 可以认为, 最左/右边的元素的左/右边是 1. 如果使用动态规划解决它, 它是什么类型的动态规划, 应该定义怎么样的状态?**

A 划分型动态规划

B 区间型动态规划

C f[i] 表示 arr 的前 i 个元素组成的子数组可以得到的最高分数

D f[i][j] 表示 arr 的第 i 到第 j 个元素之间的元素组成的子数组可以得到的最高分数

我不会

正确答案是 B D ，有55%的同学超过了你，但是千万不要气馁。

解析：如果你的答案不是 AC 或 BD, 那么你可能需要复习一下划分型和区间型动态规划的区别. 不过我们也可以不关注是什么型, 只要能设定出合理的状态, 列出状态转移方程, 最后能够得到正确的解就足够啦. "管它黑猫白猫, 能抓耗子的猫就是好猫."

必做

单选题 **我们设定的状态是 f[i][j] 表示 arr 的第 i 个到第 j 个元素之间的元素组成的子数组被一个一个拿走时可以得到的最高分数. 假如 arr 的大小为 n 并且 f[i][j] 包含 arr[i] 和 arr[j], 那么此时的状态转移方程是?**

A f[i][j] = max(f[i][k] + f[k][j] + arr[k]*arr[k-1]*arr[k+1]) i ≤ k ≤ j

B f[i][j] = max(f[i][k] + f[k][j] + arr[k]*arr[i]*arr[j]) i ≤ k ≤ j

C f[i][j] = max(f[i][k-1] + f[k+1][j] + arr[k]*arr[i-1]*arr[j+1]) i ≤ k ≤ j

提交　我不会

正确答案是 C ，有34%的同学做对了这道题目哦，继续努力!

解析：还记得区间型动态规划的一个要点吗? "倒着". 要解决状态 f[i][j], 我们的决策其实是倒着来的, 我们要枚举的是这个区间的元素最后拿走的是哪个, 拿走这个获得的分数就是这个元素的值乘这个区间两边的元素, 也就是 arr[i-1] 和 arr[j+1], 特殊地, 如果 arr[i - 1] 或 arr[j + 1] 越界, 那应该使用 1, 如果 f[i][j] 中 i > j, 应该得到 0.

必做

单选题 **最后一个要确定的问题，边界！对应上面定义的状态，这个问题的初始态边界应该是什么呢？**

A f[i][i] = arr[i]

B f[i][0] = arr[i]

C f[i][i] = arr[i - 1] * arr[i] * arr[i + 1]

提交　　我不会

正确答案是C，有28%的同学答对了，要加油了。

解析：按照上面的思路，我们的初始化，就应该是拿走第i个元素的得分，就是说f[i][i]的得分，那么根据题目描述拿走一个元素的得分是左边的元素 * 该元素 * 右边的元素，所以f[i][i] = arr[i - 1] * arr[i] * arr[i + 1]。

312. Burst Balloons

难度 困难　167　收藏　分享　切换为中文　关注

题目描述　评论 (39)　题解(25)　提交记录

Given `n` balloons, indexed from `0` to `n-1`. Each balloon is painted with a number on it represented by array `nums`. You are asked to burst all the balloons. If the you burst balloon `i` you will get `nums[left] * nums[i] * nums[right]` coins. Here `left` and `right` are adjacent indices of `i`. After the burst, the `left` and `right` then becomes adjacent.

Find the maximum coins you can collect by bursting the balloons wisely.

Note:

- You may imagine `nums[-1] = nums[n] = 1`. They are not real therefore you can not burst them.
- `0 ≤ n ≤ 500, 0 ≤ nums[i] ≤ 100`

Example:

```
Input: [3,1,5,8]
Output: 167
Explanation: nums = [3,1,5,8] --> [3,5,8] -->   [3,8]   -->  [8]  --> []
             coins =  3*1*5      +  3*5*8    +  1*3*8      + 1*8*1   = 167
```

```java
class Solution {
    public int maxCoins(int[] nums) {

        int[] num = new int[nums.length + 2];
        int N = num.length;
        num[0] = 1;
        num[N-1] = 1;
        for( int i = 1, j = 0; i < N - 1; i++, j++ ) {
            num[i] = nums[j];
        }

        int[][] dp = new int[N][N];

        for( int len = 2; len < N; len++ ) {
            for( int i = 0; i < N - len; i++ ) {
                int j = i + len;
                for( int k = i + 1; k < j; k++ ) {
                    dp[i][j] = Math.max(dp[i][j], dp[i][k] + dp[k][j] + num[i]*num[k]*num[j]);
                }
            }
        }
        return dp[0][N-1];
    }
}
```

312. Burst Balloons

难度 困难 167 收藏 分享 切换为中文 关注

题目描述 评论 (39) 题解(25) 提交记录

执行结果: 通过 显示详情 >

执行用时 : 6 ms , 在所有 java 提交中击败了 97.51% 的用户

内存消耗 : 34.8 MB , 在所有 java 提交中击败了 41.38% 的用户