OneStepOffer 算法讲义
第八讲

DP Part4

字符串编辑 - 72 Edit Distance (Hard)

题目描述
给定两个字符串，已知你可以删除、替换和插入任意字符串的任意字符，求最少编辑几步可
以将两个字符串变成相同。

输入是两个字符串，输出是一个整数，表示最少的步骤。

Input: word1 = "horse", word2 = "ros"
Output: 3

在这个样例中，一种最优编辑方法是（1）horse -> rorse （2）rorse -> rose（3）rose -> ros。

题目分析

我们使用一个二维数组 dp[i][j]，表示将第一个字符串到位置 i 为止，和第二个字符串到位置 j 为止，**最少需要几步编辑**。

当第 i 位和第 j 位对应的字符相同时，dp[i][j] 等于 dp[i-1][j-1]；当二者对应的字符不同时，修改的消耗是 dp[i-1][j-1]+1，插入 i 位置/删除 j 位置的消耗是 dp[i][j-1] + 1，插入 j 位置/删除 i 位置的消耗是 dp[i-1][j] + 1。

dp[i-1][j-1] + ((word1[i-1] == word2[j-1])? 0: 1 -- 修改

dp[i][j-1] + 1 -- 插入第一个字符串i位置 / 删除第二个字符串j位置

C++ 代码实现

```
int minDistance(string word1, string word2) {
    int m = word1.length(), n = word2.length();
    vector<vector<int>> dp(m + 1, vector<int>(n + 1, 0));
    for (int i = 0; i <= m; ++i) {
        for (int j = 0; j <= n; ++j) {
            if (i == 0) {
                dp[i][j] = j;
            } else if (j == 0) {
                dp[i][j] = i;
            } else {
                dp[i][j] = min(dp[i-1][j-1] + ((word1[i-1] == word2[j-1])? 0: 1),min(dp[i-1][j] + 1, dp[i][j-1] + 1));
            }
        }
    }
    return dp[m][n];
}
```

Java 代码实现

```java
public int minDistance(String word1, String word2) {
        int m = word1.length(); int n = word2.length();
        int[][] dp = new int[m+1][n+1];
        for (int i = 0; i <= m; i++) {
            for (int j = 0; j <=n; j++) {
                if (i == 0) dp[i][j] = j;
                else if (j == 0) dp[i][j] = i;
                else dp[i][j] = Math.min(
                    dp[i-1][j-1] +  (word1.charAt(i-1) == word2.charAt(j-1)? 0: 1),
                    Math.min(dp[i-1][j], dp[i][j-1])+1);
            }
        }
        return dp[m][n];
    }
```

Python 代码实现

```
def minDistance(self, word1: str, word2: str) -> int:
    m = len(word1)
    n = len(word2)

    dp = [[0] * (n+1) for _ in range (m+1)]

    for i in range (m+1):
        dp[i][0] = i

    for j in range (n+1):
        dp[0][j] = j

    for i in range (1, m+1):
        for j in range(1, n+1):
            left = dp[i-1][j] + 1
            down = dp[i][j-1] + 1
            left_down = dp[i-1][j-1]
            if word1[i-1] != word2[j-1]:
                left_down += 1
            dp[i][j] = min(left, down, left_down)

    return dp[m][n]
```

复杂度分析

时间复杂度 -- O(m*n) 其中m是word1长度，n是word2长度

空间复杂度 -- O(m*n)

650 Keys Keyboard (Medium)

给定一个字母 A，已知你可以每次选择复制全部字符，或者粘贴之前复制的字符，求最少需要几次操作可以把字符串延展到指定长度。

输入是一个正整数，代表指定长度；输出是一个整数，表示最少操作次数。

Input: 3 Output: 3

在这个样例中，一种最优的操作方法是先复制一次，再粘贴两次。

Input: 9 Output: 6

在这个样例中，CopyAll (A), Paste(AA), Patse(AAA), CopyAll(AAA), Paste(AAAAAA), PASTE(AAAAAAAAA)

题目解答

不同于以往通过加减实现的动态规划，这里需要乘除法来计算位置，因为粘贴操作是倍数增加的。

我们使用一个一维数组 dp，其中位置 i 表示延展到长度 i 的最少操作次数。

对于每个位置j，如果 j 可以被 i 整除，那么长度 i 就可以由长度 j 操作得到，其操作次数等价于把一个长度为 1 的 A 延展到长度为 i/j。

因此我们可以得到递推公式 dp[i] = dp[j] + dp[i/j]。

C++ 解答

```
int minSteps(int n) {
    vector<int> dp(n + 1);
    int h = sqrt(n);
    for (int i = 2; i <= n; ++i) {
        dp[i] = i;
        for (int j = 2; j <= h; ++j) {
            if (i % j == 0) {
                dp[i] = dp[j] + dp[i/j];
                Break;
            }
        }
    }
    return dp[n];
}
```

Java 解法

```java
public int minSteps(int n) {
        int[] dp = new int [n + 1];
        int h = (int) Math.sqrt(n);
        for (int i = 2; i <= n; ++i) {
            dp[i] = i;
            for (int j = 2; j <= h; ++j) {
                if (i % j == 0) {
                    dp[i] = dp[j] + dp[i/j];
                    break;
                }
            }
        }
        return dp[n];
    }
```

Python 算法

```
def minSteps(self, n):
        ans = 0
        d = 2
        while n > 1:
            while n % d == 0:
                ans += d
                n /= d
            d += 1
        return ans
```

复杂度分析

时间复杂度 - O(n^3/2)

空间复杂度 - O(n)

10. Regular Expression Matching (Hard)

题目描述：给定一个字符串和一个正 则表达式（regular expression, regex），求该字符串是否可以被匹配。

“*” 出现在字母后，代表字母可以出 现任意次 “.” 代表可以替代任意的字母

输入是一个待匹配字符串和一个用字符串表示的正 则表达式，输出是一个布尔值，表示是否
可以匹配成功。

Input: s = "aab", p = "c*a*b"
Output: true

在这个样例中，我们可以重复 c 零次，重复 a 两次。

题目分析

我们可以使用一个二维数组 dp，其中 dp[i][j] 表示以 i 截止的字符串是否可以被以j 截止的正则表达式匹配。根据正则表达式的不同情况，即字符、星号，点号，我们可以分情况讨论来更新 dp 数组

C++ 代码解读

```cpp
bool isMatch(string s, string p) {
    int m = s.size(), n = p.size();
    vector<vector<bool>> dp(m + 1, vector<bool>(n + 1, false));
    dp[0][0] = true;
    for (int i = 1; i < n + 1; ++i) {
        if (p[i-1] == '*') {
            dp[0][i] = dp[0][i-2];
        }
    }

    for (int i = 1; i < m + 1; ++i) {
        for (int j = 1; j < n + 1; ++j) {
            if (p[j-1] == '.') {
                // 如果p[j-1]是点 可以代替任意字母所以取决于前面的结果
                dp[i][j] = dp[i-1][j-1];
            } else if (p[j-1] != '*') {
                // p[j-1]既不是星 又不是点 判断两个字母相等与否
                dp[i][j] = dp[i-1][j-1] && p[j-1] == s[i-1];
            } else if (p[j-2] != s[i-1] && p[j-2] != '.') {
                // p[j-1]是星 但是p[j-2]不是点 且和s[i-1] 不相等，考虑前面p[j-2]即可
                dp[i][j] = dp[i][j-2];
            } else {
                // .* 的情况 取决于dp[i][j-1], dp[i-1][j]等共同结果
                dp[i][j] = dp[i][j-1] || dp[i-1][j] || dp[i][j-2];
            }
        }
```

Java 代码解读

```java
public boolean isMatch(String s, String p) {
    int m = s.length(), n = p.length();
    Boolean[][] dp = new Boolean[m+1][n+1];

    dp[0][0] = true;
    for (int i = 1; i < n + 1; ++i) {
        if (p.charAt(i-1) == '*') {
            dp[0][i] = dp[0][i-2];
        } else {
            dp[0][i] = false;
        }
    }

    for (int i = 1; i < m+1; ++i) {
        dp[i][0] = false;
    }
    for (int i = 1; i < m + 1; ++i) {
        for (int j = 1; j < n + 1; ++j) {
            if (p.charAt(j-1) == '.') {
                dp[i][j] = dp[i-1][j-1];
            } else if (p.charAt(j-1) != '*') {
                dp[i][j] = dp[i-1][j-1] && p.charAt(j-1) ==
s.charAt(i-1);
            } else if (p.charAt(j-2) != s.charAt(i-1) &&
p.charAt(j-2) != '.') {
                dp[i][j] = dp[i][j-2];
            } else {
                dp[i][j] = dp[i][j-1] || dp[i-1][j] || dp[i][j-2];
            }
        }
    }
    return dp[m][n];
  }
}
```

复杂度分析

时间复杂度 - O(m*n) m = length of s, n = length of p

空间复杂得 - O(m*n)