Assignment Two- Programming

1951976 李林飞

2021 年 5 月 8 日

1 三维 dp

1.1 算法思想

由于这个问题是求最值，并且问题具有重叠子问题和最优子结构两个重要特征，所以很容易想到用动态规划算法。其难点就是如何写出正确的状态转移方程。

为了得到正确的状态转移方程，首先要明确 dp 数组的含义，然后考虑最简单的情况，确定 dp 的初值，再由数学归纳思想得到状态转移方程。

由于题目要求可以连续的旋转 1-3 位数字，所以很容易得到这样一个规则：假设第 i 位、第 i+1 位、第 i+2 位上的数字变为目标数字分别需要旋转 x、y、z 次，则必有 $x \geq y \geq z$ 成立。基于此，我们可以先确定第 i 位数字变为目标数字所需的步数，然后分别向上和向下枚举得到三位数字变成目标数字的最少移动次数。枚举每一位数字，就可以得到问题的最优解。

假设字符串为 s1 和 s2, 其长度为 len。使用 DP table 自底向上求解，先明确 dp 数组的定义：定义 dp 数组为 $dp[pos][i][j]$ 表示第 pos 位是合法的，i 表示第 pos+1 位上的数字变成目标数字需要旋转 i 次,j 表示第 pos+2 位上的数字变成目标数字需要旋转 j 次。在此定义上，初始值 $dp[0][0][0] = 0$，然后依次计算第 pos 位上向上和向下旋转目标数字的次数，接着根据上述规则进行向上和向下枚举遍历，遍历所有数字后 $dp[len][0][0]$ 即是问题的解。注意，在状态转移期间为保证循环旋转，需要对 i 和 j 进行取模处理。

1.2 核心代码

```
dp[0][0][0] = 0;              // base case
for(int pos = 0; pos < len; pos++)
{
  for(int i = 0; i < 10; i++)
  {
    for(int j = 0; j < 10; j++)
    {
      // 第pos位数字向上旋转到目标数字的步数
      int up = (s2[pos] - s1[pos] - i + 20)%10;

      // 第pos位数字向下旋转到目标数字的步数
      int down = 10 - up;

      // 枚举向上旋转 —— 状态转移方程
      for(int p = 0; p < up  + 1; p++)
      for(int q = 0; q < p + 1; q++)
      dp[pos+1][(j+p)%10][q] =
          min(dp[pos+1][(j+p)%10][q], dp[pos][i][j] + up);

      // 枚举向下旋转 —— 状态转移方程
      for(int p = 0; p < down + 1; p++)
      for(int q = 0; q < p+1; q++)
      dp[pos+1][(j-p)%10][(10-q)%10] =
           min(dp[pos+1][(j-p+10)%10][(10-q)%10], dp[pos][i][j] + down);
    }
  }
}

```

1.3 效率分析

根据上述核心代码可知，问题中的 base operation 是状态转移方程，其执行次数取决于循环的次数。设输出规模——字符串的长度为 n，则最外层循环次数 (pos) 为 n; 为遍历得到第 pos+1 位和第 pos+2 位上数字变成目标数字的次数需要循环嵌套各 10 次 (i,j)。在计算得出第 pos 位数字需要向上或向下旋转次数后，在最坏情况下，向上或向下需要 10 次，平均为 5 次，进而根据状态转移方程向上和向下枚举遍历是并列的，其循环次数最坏情况下是 100 次。综上所述，在最坏情况下，base operation 执行次数约为 $10*10*10*10*n = 10000n$。因此当 n 趋于无穷大时，时间复杂度是线性的，即 $\Theta(n)$。而空间复杂度主要是 dp 数组，容量为 $10*10*n$, 即 $\Theta(n)$。在

本题中，由于限定 $n \leq 1000$, 因此实际考虑复杂度时不能忽略系数的影响。

1.4 运行结果

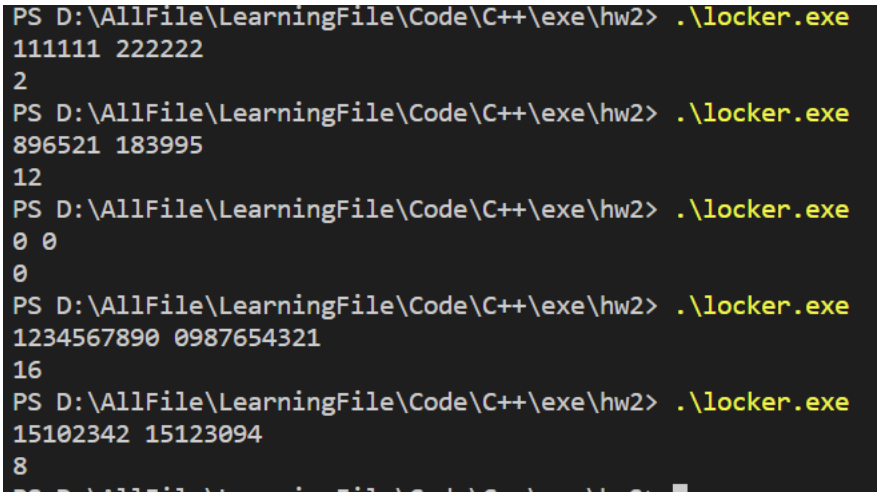

图 1: 运行结果截图

2 状态压缩——二维 dp

2.1 算法思想

在动态规划算法中，如果状态转移方程中的状态 dp 需要的都是 dp 相邻的状态，则一般可以使用状态压缩技巧，从而降低空间复杂度。在 locker 这个问题中，三个数字的旋转状态是相互关联的，即有：假设第 i 位、第 i+1 位、第 i+2 位上的数字变为目标数字分别需要旋转 x、y、z 次，则必有 $x \geq y \geq z$ 成立。而在上述算法中，使用的是三维的 dp 数组，实际上，最先的想法是四维 dp 数组，即 $dp[pos][i][j][k]$, 其中 pos 表示第 pos 位，i、j、k 分别表示第 pos、pos+1、pos+2 位上的数字。改变 dp 数组的定义，可以用三维的 dp 数组来表示，如上述算法中：$dp[pos][i][j]$ 表示第 pos 位，i 表示第 pos+1 位上的数字变成目标数字需要旋转 i 次,j 表示第 pos+2 位上的数字变成目标数字需要旋转 j 次。进一步，我们可以将每三个数字作为一维，则到二维的 dp 数组，即 $dp[pos][current]$，其中 pos 表示第 pos 位，current 表示第 pos、pos+1、pos+2 位上数字组成的三位数。经此处理，可得到以下算法。该算法的核心思想与上述算法一样，得到三位数后又拆成三位数字，先计算 pos 位上旋转到目标数字的步数，然后通过向上和向下枚举计算第 pos+1、pos+2 位上数字旋转到目标数字的步数，遍历全部位数，即可得到问题的解。遗憾的是，这个过程并不是真正的状态压缩，只是类似状态压缩的处理方法。

2.2 核心代码

```
dp[1][start] = 0;       // base case
for(int pos = 1; pos <= len - 3 + 1; pos ++)
{
  for(int current = 0; current <= 999; current ++)
  {
    // 锁定三位数字
    if(dp[pos][current] == INF) continue;

    // 获取三位数j的个、十、百位上的数字
    int hun = current / 100;
    int ten = (current - hun * 100) / 10;
    int num = current - hun * 100 - ten * 10;

    // 获取下一个三位数的末尾数字
    int next_end = s1[pos + 3] - '0';
```

```
16
17      // 第pos位数字向上旋转到目标状态的步数
18      int up = (s2 [pos] -'0'+ 10 - hun)% 10;
19      // 第pos位数字向下旋转到目标状态的步数
20      int down = 10 - up;
21
22      // 枚举向上旋转 —— 状态转移方程
23      for (int p = 0; p < up + 1; p ++) // may change the second
24      for (int q = 0; q < p + 1; q ++) // third change
25      {
26        // 获取pos+1位置上对应的三位数
27        int next = 100 * (ten + p > 9 ? ten + p - 10:ten + p) + 10 * (num + q
        > 9 ? num + q - 10:num + q) + next_end;
28        // 状态转移方程
29        dp[pos + 1][next] = min(dp[pos + 1][next],dp[pos][current] + up);
30      }
31
32      // 枚举向下旋转 —— 状态转移方程
33      for(int p = 0;p < down + 1;p ++)
34      for(int q = 0;q < p + 1;q ++)
35      {
36        // 获取pos+1位置上对应的三位数
37        int next = 100 * (ten - p < 0 ? ten - p + 10:ten - p) + 10 * (num - q
        < 0 ? num - q + 10:num - q) + next_end;
38        // 状态转移方程
39        dp[pos + 1][next] = min(dp[pos + 1][next],dp[pos][current] + down);
40      }
41    }
42  }
```

2.3 效率分析

由上述核心代码中可知，在枚举过程中，外层循环是输入字符串的长度 n，内层循环是将数字字符串拆成三位数的个数，为 n。在枚举遍历下两位数字旋转的次数最坏情况是 100 次。因此，在最坏情况下，base operation 执行次数为 $100 * n^2$。在 n 趋于无穷大时，时间复杂度为 $\Theta(n^2)$。虽然这种算法将 dp 数组改为二维，但时间复杂度却增大，所以并不是好的压缩方法。

2.4 运行结果

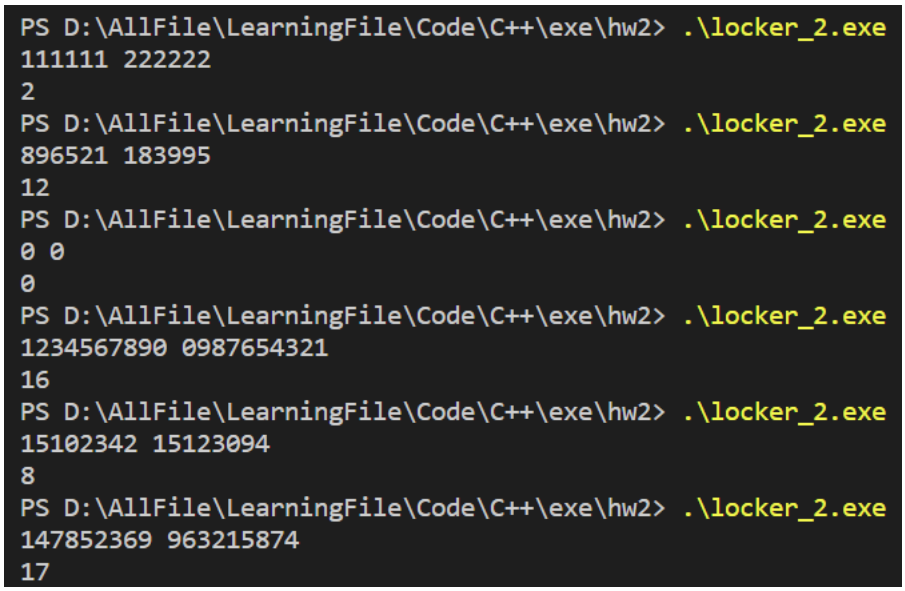
PS D:\AllFile\LearningFile\Code\C++\exe\hw2> .\locker_2.exe
111111 222222
2
PS D:\AllFile\LearningFile\Code\C++\exe\hw2> .\locker_2.exe
896521 183995
12
PS D:\AllFile\LearningFile\Code\C++\exe\hw2> .\locker_2.exe
0 0
0
PS D:\AllFile\LearningFile\Code\C++\exe\hw2> .\locker_2.exe
1234567890 0987654321
16
PS D:\AllFile\LearningFile\Code\C++\exe\hw2> .\locker_2.exe
15102342 15123094
8
PS D:\AllFile\LearningFile\Code\C++\exe\hw2> .\locker_2.exe
147852369 963215874
17

图 2: 运行结果截图