作业2-编程

# 问题重述

一个有N位（N≤1000）的密码锁，其中每一位都可以被旋转到0~9。一次可以同时向上或者向下旋转连续的1~3位。

现已知密码锁初始状态和正确密码，求出最少旋转多少次可以将其解锁。

# 算法设计

为了解决本问题，采用的算法主要是动态规划（Dynamic Programming，DP）算法。

我们注意到，第i位被旋转到正确位置需要的旋转次数依赖于第i-1位被旋转到正确位置需要的旋转次数。此外，该密码锁一次最多可以旋转3位，因此在考虑第i位被旋转到正确位置需要的旋转次数时同时还需要考虑第i+1位和第i+2位的状态。

基于这一原则，我们设置状态**dp[i][x][y]**，表示前i位被旋转到正确的位置，第i+1位旋转到数字x，第i+2旋转到数字y所需要的最小旋转次数。记a[i]为第i位初始数字，b[i]为目标状态的数字。

针对初始条件：

即在不做任何操作时，第1位初始为a[1]，第2位初始为a[2]。

针对状态转移方程：我们注意到，如果前i-1位已旋转到正确位置，此时第i位是x，则第i位被旋转到正确位置所需要的向上旋转次数*upperDis*=(*b*[*i*]+10−*x*)%10次，需要的向下旋转次数*lowerDis=10-upperDis*次。同时，在旋转第i位时，可以只旋转一位、也可以同时旋转两位或者三位。即在第i位旋转upperDis的状态下，第i+1旋转的次数j（0≤j≤upperDis），第i+2位旋转的次数k(0≤k≤j)。因此，向上旋转的状态转移方程为：

向下旋转同理。

为了避免在边界处理时的繁琐，我们设置a[n+1]=a[n+2]=b[n+1]=b[n+2]=0（可理解为n+1和n+2位不需要特殊处理），则最终需要的答案即dp[n][0][0]。

# 算法代码

**string** start, target;

cin **>>** start **>>** target;

int n = start.**size**();

*//存储每一位的数字，为简化处理设置大小为n+1*

**vector**<int> **a**(n + 3, 0);

**vector**<int> **b**(n + 3, 0);

*//初始化*

for (int i = 1; i <= n; ++i) {

    a**[**i**]** = start**[**i - 1**]** - '0';

    b**[**i**]** = target**[**i - 1**]** - '0';

}

*//dp[i][x][y]:前i位已归位，第i+1位为x，第i+2位为y最小移动次数*

**vector**<**vector**<**vector**<int> > > **dp**(n + 1,

**vector**<**vector**<int> >(10, **vector**<int>(10, 10\*n)));

*//边界条件:初始状态*

dp**[**0**][**a**[**1**]][**a**[**2**]]** = 0;

for (int i = 1; i <= n; ++i) {

    for (int x = 0; x <= 9; ++x) {

        for (int y = 0; y <= 9; ++y) {

*//将第i位从x移动到目标b[i]需要的向上移动次数*

            int  upperDis = (b**[**i**]** - x + 10) % 10;

            for (int j = 0; j <= upperDis; ++j) {

                for (int k = 0; k <= j; ++k) {

                    dp**[**i**][**(y + j) % 10**][**(a**[**i + 2**]** + k) % 10**]** = **min**(dp**[**i**][**(y + j) % 10**][**(a**[**i + 2**]** + k) % 10**]**, dp**[**i - 1**][**x**][**y**]** + upperDis);

                }

            }

*//将第i位从x移动到目标b[i]需要的向下移动次数*

            int lowerDis = 10 - upperDis;

            for (int j = 0; j <= lowerDis; ++j) {

                for (int k = 0; k <= j; ++k) {

                    dp**[**i**][**(y - j + 10) % 10**][**(a**[**i + 2**]** - k + 10) % 10**]** = **min**(dp**[**i**][**(y - j + 10) % 10**][**(a**[**i + 2**]** - k + 10) % 10**]**, dp**[**i - 1**][**x**][**y**]** + lowerDis);

                }

            }

        }

    }

}

*//前n位归位即为结果*

cout **<<** dp**[**n**][**0**][**0**]**;

# 算法分析

通过主循环分析，循环次数为n\*10\*10\*10\*10，即时间复杂度为O(n)。在动态规划过程中，建立了n\*10\*10的数组，因此空间复杂度为O(n)。

# 运行截图

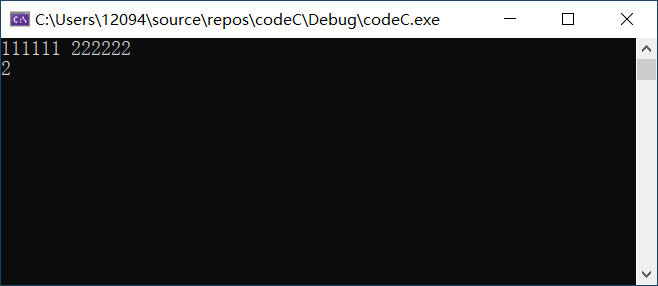
## 函数测试

**测试用例1**

输入：111111 222222

预期输出：2

实际输出：

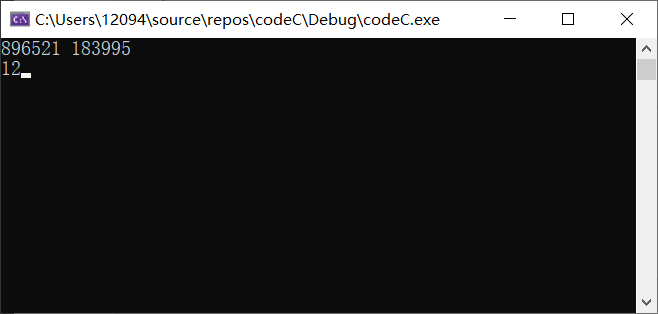


**测试用例2**

输入： 896521 183995

预期输出： 12

实际输出：



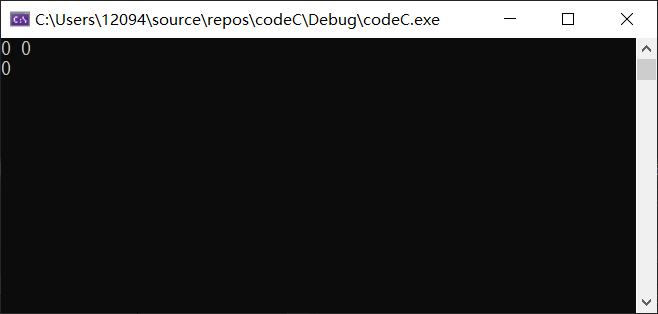
## 边界测试

**测试用例1**

输入：0 0

预期输出： 0

实际输出：

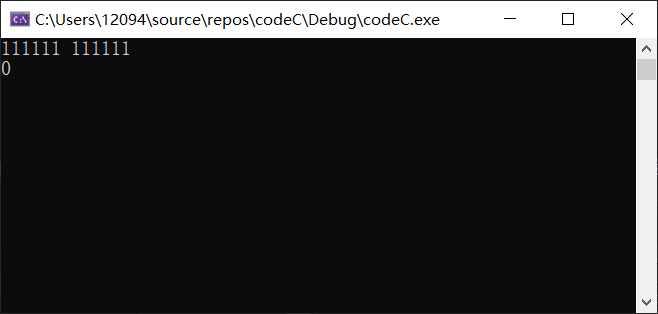


**测试用例2**

输入： 111111 111111

预期输出：0

实际输出：

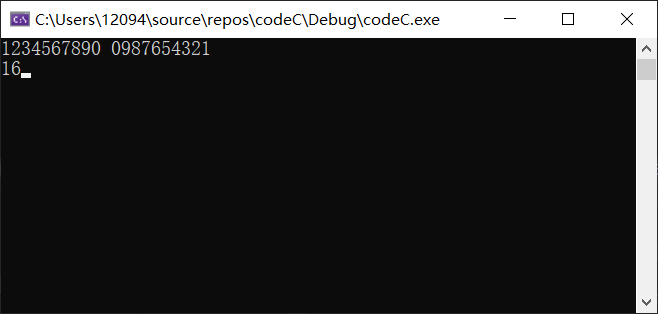


## 性能测试

**测试用例1**

输入： 1234567890 0987654321

实际输出：16



**测试用例2**

输入： 15102342 15123094

实际输出：8

