Homework 2

注意事项：

1. 书面作业需提交手写版，可以用 iPad 等设备书写
2. 请在作业首页注明姓名、学号
3. 作答时只需要写清题号，不需要抄写题目
4. 作业通过教学网提交，2025 年 3 月 23 日 23：59 截止

一、判断题（如果判断为错误，需要写出理由）

1. 字符串是特殊的线性表，特殊性体现在其内容限制为字符。✓
2. 字符串匹配的朴素算法时间复杂度为 O(m*n)，KMP 算法的时间复杂度为 O(m+n)，其中 m，n 分别为模式串、目标串长度。✓
3. 栈是抽象数据类型，是操作受限的线性表，可以用顺序存储或链式存储的方式实现。✓
4. 若让元素 1，2，3 依次进栈，并允许在栈非空的任意时刻进行出栈操作，则在 3 个元素可能形成的 6 种全排列中，仅有一种序列不可能成为合法的出栈序列，即 3，1，2。✓
5. 递归算法必须包含终止部分和递归部分。✓

二、简答题

1. 考虑字符串模式匹配问题，T=’abaabaabaabcc’，P=’abaabc’。给出模式串 P 的改进前的和改进后的 next 数组，并分别给出，使用改进前的 KMP 算法、改进后的 KMP 算法、以及朴素模式匹配算法，各需要进行多少趟的匹配。

改进前 KMP：3趟

改进后 KMP：3趟

朴素模式：7趟

改进前：next = [-1, 0, 0, 1, 1, 2]

改进后：improved_next = [-1, 0, -1, 1, 0, 2]

2.（动态规划）字符串的编辑距离（又称 Levenshtein 距离）定义如下：若 S, T 是两个字符串，对 S 进行如下三种操作：修改一个字符、在任意位置增加一个字符、删除任意一个字符，以使 S 转换为 T；则称所需要操作的最少数量为 S 与 T 之间的编辑距离。

(1) 计算如下两字符串的编辑距离：a=’kitten’，b=’sitting’。

(2) 尝试使用动态规划方法计算字符串的编辑距离。在这一问题中，涉及的子问题为计算 S[:i]与 T[:j]的编辑距离，记为 L(i, j)，0≤i≤m+1，0≤j≤n+1。分析原问题直接依赖于哪些子问题，并给出状态转移方程。(提示：考虑 S 串中最终与 T[-1]匹配的字符的几种情况)

(3) 设计一个动态规划 (Dynamic Programming) 算法，在 O(m*n)的时间内计算 S, T 的编辑距离，其中 m, n 分别为 S, T 的长度。

(1) kitten
sitting
编辑距离为3

(2) 初始条件：①若 S 为空，则向 S 中添加字符直至 S=T
②若 T 为空，则删除 S 中字符直至 S 为空

状态转移方程：

① 如果 S[i-1] = T[j-1]，无需额外操作，L(i,j) = L(i-1, j-1)

② 如果 S[i-1] != T[j-1]，有三种可选操作：

(1) 改变S字符. 则 L(i,j) = L(i-1, j-1) + 1

(2) 向S中添加字符 L(i,j) = L(i, j-1) + 1

(3) 删除S中字符. L(i,j) = L(i-1, j) + 1

实际操作中取3种方案中最小即可

(3) 代码如下：

物院 > 大一春季学期 > 数据结构与算法 > 每日选做 > 字符串编辑距离.py > ...

```python
S, T = input().split()
n, m = len(S), len(T)
dp = [[0] * (m + 1) for i in range(n + 1)]
for i in range(n + 1):
    dp[i][0] = i
for j in range(m + 1):
    dp[0][j] = j
for i in range(1, n + 1):
    for j in range(1, m + 1):
        if S[i - 1] == T[j - 1]:
            dp[i][j] = dp[i - 1][j - 1]
        else:
            dp[i][j] = min(dp[i - 1][j - 1], dp[i - 1][j], dp[i][j - 1]) + 1
print(dp[n][m])
```

3.（动态规划）假设你在一栋 M 层高的楼中工作并拥有 N 个完全相同的鸡蛋。我们可以把鸡蛋的硬度定义如下：如果鸡蛋从 L 楼被扔下后能保持完好，而从 L+1 楼被扔下时未能保持完好，则其硬度就是 L。如果鸡蛋足够硬，以至于从第 M 楼扔下也无法摔碎它，我们也把它的硬度定义为 M。

你希望确定这批鸡蛋的硬度，具体方式就是不断在不同楼层向楼下扔鸡蛋。当然，如果测试发生在过高的楼层，鸡蛋就会破碎，可用的鸡蛋也会因此减少。你需要设计一个测试方案，使得对于任意硬度的鸡蛋，该方案都能保证只损失不超过 N 个鸡蛋，就能确定鸡蛋的硬度。在满足这一条件的前提下，方案在最坏情况下需要的测试次数应该尽可能少。

(1) 假设 N=2，并且你选择在 k 楼进行第一次测试（$1 \le k \le H$），结果发现鸡蛋破碎了。指出此后唯一可行的测试方法，并指出最坏情况下总共需要进行多少次测试。

(2) 记原问题中，在最坏情况下需要的最少测试次数为 S(M, N)。即，存在一个测试方案，对于任意硬度的鸡蛋，都保证能够在 S(M, N)次尝试内确认鸡蛋的硬度。给出 S(M, 1)的值。

(3) 考虑使用动态规划方法来求解 S(M, N)。假定第一次测试发生在 k 楼，根据鸡蛋是否会破碎，问题将演变为两个子问题。即，如果鸡蛋破碎了，则接下来只需要在 1~k-1 楼中使用 N-1 个鸡蛋测试硬度，对应子问题 S(k-1, N-1)；如果鸡蛋未破碎，则接下来只需要在 k+1~M 楼中使用 N-1 个鸡蛋测试硬度，对应子问题 S(M-k, N-1)。指出 S(M, N)直接依赖的所有子问题，并给出状态转移方程。

(4) 依据(3)中的状态转移方程，设计动态规划算法求解 S(M, N)，并分析算法的时间复杂度。

(5) (选做) 设计一个时间复杂度为 O(tN)的动态规划算法解决上述问题，其中 t 为 S(M, N)的大小。

(1) 比如从1楼开始尝试，直到鸡蛋在 L+1 < K 层摔碎，或在 K-1 层未被摔碎。

最坏要进行 K-1 次测试

(2) $S(M,1) = M$

(3) 边界 ① $N=1$，$S(M,1)=0$　　三个子问题

② $N=0$，$S(M,N)=\infty$

③ $M=0$，$S(M,N)=0$

状态转移方程：

若在第一次在 K 层开始测试，两种情况：

① 鸡蛋碎了，在 1～K-1 层对 N-1 个鸡蛋测试，

$S(M,N) = S(K-1, N-1) + 1$

② 鸡蛋没碎，在 K+1～M 层对 N 个鸡蛋测试

$S(M,N) = S(M-K, N) + 1$

最坏情况，$S(M,N) = \max\{S(K-1,N-1), S(M-K,N)\} + 1$

对 K 取从 1，M，我们期望取其中最小的方案，即

$$S(M,N)=\min_{1\le k\le M}\left\{\max\left\{S(k-1,N-1),S(M-k,N)\right\}\right\}+1$$

(4) 代码1：动态规划

```
M, N = map(int, input().split())
dp = [[float('inf')] * (N + 1) for i in range(M + 1)]
for i in range(1, M + 1):
    dp[i][0] = float('inf')
    dp[i][1] = i
for j in range(1, N + 1):
    dp[0][j] = 0
for i in range(1, M + 1):
    for j in range(2, N + 1):
        for k in range(1, i + 1):
            dp[i][j] = min(dp[i][j], max(dp[k - 1][j - 1], dp[i - k][j]) + 1)
print(dp[M][N])
```

时间复杂度 $O(NM^2)$

代码2：记忆化搜索

```
from functools import lru_cache
@lru_cache(maxsize=None)
def S(M, N):
    if N == 1:
        return M
    if M == 0:
        return 0
    if N == 0:
        return float('inf')
    else:
        s = float('inf')
        for i in range(1, M + 1):
            s1 = max(S(i - 1, N - 1), S(M - i, N)) + 1
            s = min(s, s1)
        return s
M, N = map(int, input().split())
print(S(M,N))
```

(5) 考虑反着来，即有 n 个鸡蛋，t 次测试，最大可测楼层范围。

初始化二维数组 dp，$dp[t][n]$ 表示 t 次测试，n 个鸡蛋所能测得最高楼层。

初始条件：$\begin{cases} dp[0][n]=0 \\ dp[t][1]=t \end{cases}$

状态转移方程 $dp[t][n]=dp[t-1][n-1]+dp[t-1][n]+1$

t 从 0 开始逐步搜索，直到 $dp[t][n]\ge M$ 终止。

```python
def min_egg_drops(M, N):
    dp = [[0] * (N + 1) for _ in range(M + 1)]
    for t in range(1, M + 1):
        dp[t][1] = t
        dp[0][N] = 0
    t = 0
    while dp[t][N] < M:
        t += 1
        for n in range(2, N + 1):
            dp[t][n] = dp[t-1][n-1] + dp[t-1][n] + 1

    return t
M, N = map(int, input().split())
print(min_egg_drops(M, N))
```