一、实验题目与要求

1. Matrix-chain product.
2. Longest Common Subsequence.
3. Lognest Common Substring.
4. Max Sum.
5. Shortest path in multistage graphs. Find the shortest path from 0 to 15 for the given graph.

二、算法思想

1. Matrix-chain product

使用动态规划优化链式矩阵乘法的乘法次数时，需要构造一个 $n\times n$ 的二维矩阵，该矩阵为对称矩阵，索引为 (i, j) 的元素表示从第 i 个矩阵到第 j 个矩阵的最少乘法次数；还需要构造一个 $(n-1)\times(n-1)$ 的二维对称矩阵，该矩阵中索引为 (i, j) 的元素表示第 i 个矩阵到第 j 个矩阵的链式乘法中，需要断开的位置，以得到最少的乘法次数。

根据上述描述，矩阵 m 的元素值应为

$$m[i,j]=\begin{cases}0, & i=j,\\ \min\{m[i,k]+m[k+1,j]+p_{i-1}p_kp_j\}, & i<j.\end{cases}$$

按照上述规则计算得到两个矩阵后，能直接读出该链式乘法的最小乘法开销。为了得到该链式矩阵乘法的具体结合方式，需要使用第二个矩阵递归逆推：读取 $m[i, j]$ 的值为 k，则 $[i, k]$ 和 $[k+1, j]$ 各自分开，随后递归求解。

2. Longest Common Subsequence

给定两个字符串 s_1 和 s_2，为了计算它们的公共最长子序列，需要创建一个 $(m+1)\times(n+1)$ 的矩阵，其中 m 和 n 分别为 s_1 和 s_2 的长度。之所以需要加 1，是为了在计算首行首列元素的最长公共子序列时不会导致数组越界。

对于矩阵 (i, j) 的元素，值的确定方式为

$$c[i,j]=\begin{cases}c[i-1,j-1]+1, & x_i=y_j,\\ \max(c[i,j-1],c[i-1,j]), & x_i\neq y_j.\end{cases}$$

二维矩阵的所有元素都计算完成后，需要从整个矩阵的右下角（最后一个元素开始）逆向推回：若 $x_i=y_j$，则 x_i 和 y_j 是最长公共子序列的一部分；否则则在 $(i, j-1)$ 和 $(i-1, j)$ 中选择较大的一方继续执行算法。

3. Lognest Common Substring

相比于最长公共子序列，最长公共子串需要修改动态规划矩阵的计算方式：若元素比较不 相等，则从 0 开始计算：

$$c[i,j] = \begin{cases} c[i-1,j-1]+1, & x_i = y_j, \\ 0, & x_i \neq y_j. \end{cases}$$

此外，最长公共子序列只需要读取动态规划数组的最后一个元素即可获得该子序列的长 度，而最长公共子串则需要在生成动态规划数组时时刻记录出现过的最长子串的长度以及范 围，以便于得到最长公共子串。

4. Max Sum

对于动态规划数组 dp，令 $dp[i]$ 表示以索引为 i 的数为序列结尾时最长的子段和，则 $dp[i] = \max(dp[i-1] + a[i], a[i])$。该方程的含义为，若包含前一个元素的最大子段和（$dp[i-1]$）再加上当前元素，能构成更大的子段和，则这样做；否则忽略前面所有元素，仅包含当前元素即可。

按照 dp 数组的含义，该数组的最后一项仍然不是结果，因此需要在计算过程中时刻统计最大子段和以及该子段的范围。

5. Shortest path in multistage graphs

若要使用动态规划算法计算给定图中的最短路径，则同样需要设定一维动态数组 dp，其中 $dp[i]$ 表示从起始节点到索引为 i 的节点的最短路径长，该数组的计算需要从后向前进行： 从节点 i 起，对于它的每个直接后继结点 j，计算 i 到 j 的距离以及 j 到目的位置的最短距离的和，从中选择最短的路径作为 $dp[i]$ 的结果。

为了同时能获得最短路径具体经过的节点，还需要在上述迭代过程中保存最短路径对应的 下一个节点。

三、算法步骤与核心代码

1. Matrix-chain product

该算法实现于 矩阵链乘法.py中。

首先用动态规划进行乘法优先级的选择：

```python
def matrix_chain_order(p):
    n = len(p) - 1
    m = [[0 for i in range(n)] for i in range(n)]
    s = [[0 for i in range(n)] for i in range(n)]
    for l in range(2, n+1):
        for i in range(1, n-l+2):
            j = i + l -1
            m[i-1][j-1] = float('inf')
            for k in range(i, j):
                q = m[i-1][k-1] + m[k][j-1] + p[i-1] * p[k] * p[j]
                if q < m[i-1][j-1]:
                    m[i-1][j-1] = q
                    s[i-1][j-1] = k
    return m, s
```

然后以此为顺序实现矩阵括号及矩阵的打印（以A_i来表示）：

```python
def print_optimal_parens(s, i, j):
    if i == j:
        print('A', i, sep='',end='')
    else:
        print('(', end='')
        print_optimal_parens(s, i, int(s[i-1][j-1]))
        print_optimal_parens(s, int(s[i-1][j-1])+1, j)
        print(')', end='')
```

2. Longest Common Subsequence

该算法实现于最长公共子序列.py中。

该算法的动态规划转移方程在上文中已经给出，对应于该转移方程的代码实现为

```
    if s1[x] == s2[y]:
        dp[x][y] = 1 + best(x + 1, y + 1)
    else:
        dp[x][y] = max(best(x + 1, y), best(x, y + 1))
```

加上特殊、边界情况处理后的完整递归函数为：

```
def best(x, y):
    if dp[x][y] != -1: return dp[x][y]
    if x == l1 or y == l2:
        return 0
    if s1[x] == s2[y]:
        dp[x][y] = 1 + best(x + 1, y + 1)
    else:
        dp[x][y] = max(best(x + 1, y), best(x, y + 1))
    return dp[x][y]
```

3. Lognest Common Substring

该算法实现于最长公共子串.py中。我认为这和第二题是一样的，所以解答见上即可，不再赘述。

4. Max Sum

该算法实现于最大子数组的和.py中。

计算最大字段和时，根据上文给出的 _dp 数组计算方式，由于 _dp[i] 只依赖于 _dp[i-1]，因此可以不比存放所有元素，只存放上一个元素即可。同最长公共子串相同，该问题中动态规划数组的最后一个元素仍然不是问题的解，因此需要在遍历生成数组的每一个元素时，记录当前位置前的最大子段和和子段位置：

```
array=list(map(int,input("请依次输入序列值：").split()))
dp = [i for i in array]
ans=array[0]
for i in range(1, len(array)):
    dp[i] = max(dp[i - 1] + array[i], array[i])
print("最大和为：{}".format(max(dp)))
```

5. Shortest path in multistage graphs

该算法实现于首尾最短路.py中。

实际上此题就是多源最短路径的特殊情况（即只需要计算首尾节点的最短距离即可），用的也还是Floyd算法：

```
for k in range(n):
    for i in range(n):
        for j in range(n):
            graph[i][j] = min(graph[i][j], graph[i][k] + graph[k][j])
```

四、总结

本次实现的算法很多都没有标准库中对应的原型，因此我自己设计了接口的格式，如矩阵 链乘算法以语法树的形式返回、最短路径以图的形式返回。