PS11-231830106

Problem 1

(a):

0.

	1	2	3	4	5	6
1	0	∞	∞	∞	-1	∞
2	1	0	∞	2	∞	∞
3	∞	2	0	∞	∞	-6
4	-2	∞	∞	0	3	∞
5	∞	7	∞	∞	0	∞
6	∞	5	12	∞	∞	0

1.

	1	2	3	4	5	6
1	0	∞	∞	∞	-1	∞
2	1	0	∞	2	0	∞
3	∞	2	0	∞	∞	-6
4	-2	∞	∞	0	-3	∞
5	∞	7	∞	∞	0	∞
6	∞	5	12	∞	∞	0

2.

	1	2	3	4	5	6
1	0	∞	∞	∞	-1	∞
2	1	0	∞	2	0	∞
3	∞	2	0	4	2	-6
4	-2	∞	∞	0	-3	∞
5	∞	7	∞	9	0	∞
6	∞	5	12	7	5	0

3.

	1	2	3	4	5	6
1	0	∞	∞	∞	-1	∞
2	1	0	∞	2	0	∞
3	∞	2	0	4	2	-6
4	-2	∞	∞	0	-3	∞
5	∞	7	∞	9	0	∞
6	∞	5	12	7	5	0

4.

	1	2	3	4	5	6
1	0	∞	∞	∞	-1	∞
2	0	0	∞	2	0	∞
3	2	2	0	4	1	-6
4	-2	∞	∞	0	-3	∞
5	7	7	∞	9	0	∞
6	5	5	12	7	4	0

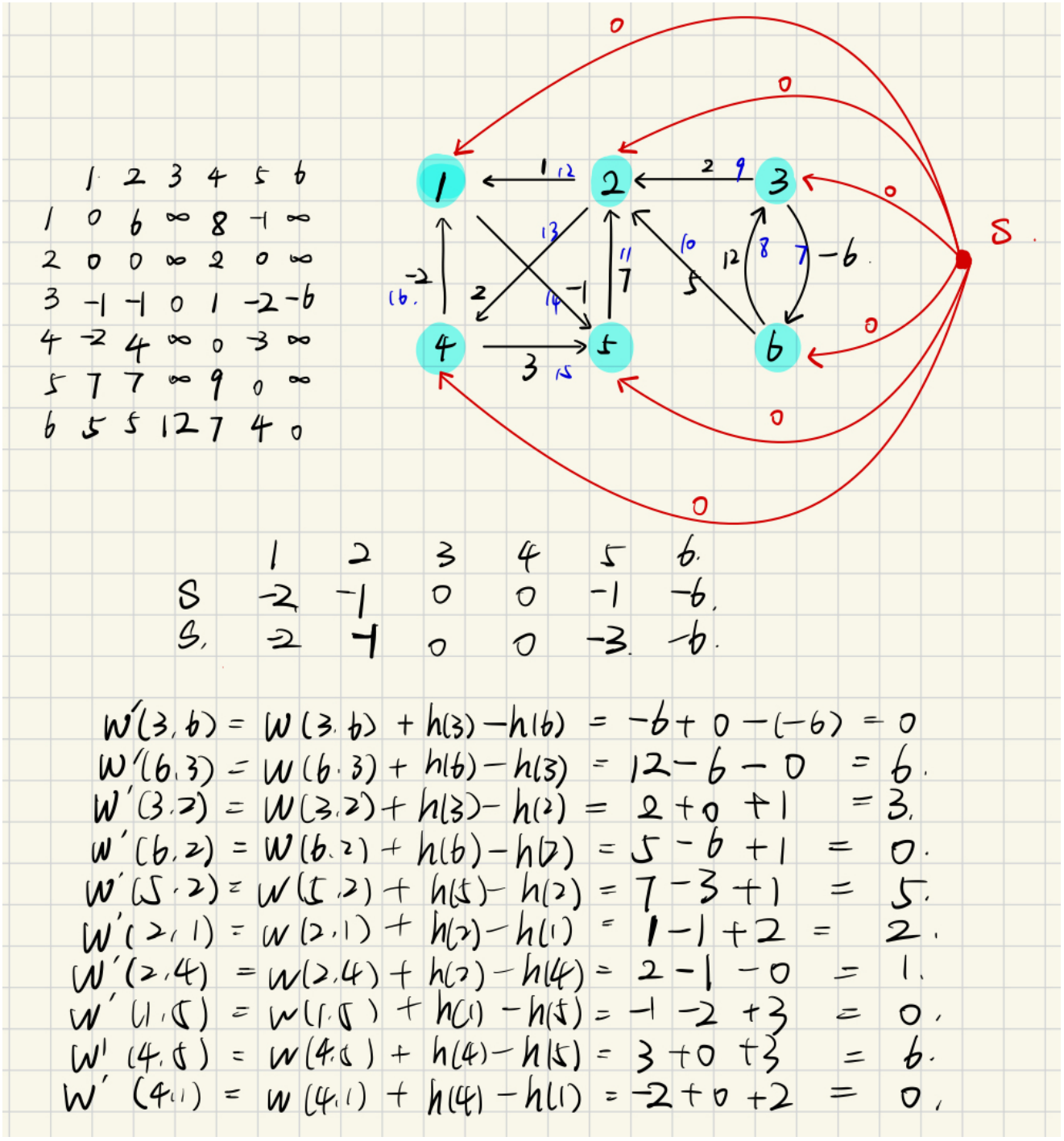
5.

	1	2	3	4	5	6
1	0	6	∞	8	-1	∞
2	0	0	∞	2	0	∞
3	2	2	0	4	1	-6
4	-2	4	∞	0	-3	∞
5	7	7	∞	9	0	∞
6	5	5	12	7	4	0

6.

	1	2	3	4	5	6
1	0	6	∞	8	-1	∞
2	0	0	∞	2	0	∞
3	-1	-1	0	1	-2	-6
4	-2	4	∞	0	-3	∞
5	7	7	∞	9	0	∞
6	5	5	12	7	4	0

(b):



Problem 2

(a):

1. 思路：显然本题可类比于一个图，c_i为顶点，$r_{i,j}$为边，本题即为找到s到t的最大乘积路径。设$w_{i,j} = -\log(r_{i,j})$，则直接等价于求最短路径，由于图中可能含有负边，因此使用Bellman-Ford算法即可

```
Input: n (货币数量), r (汇率矩阵), s (起点货币), t (终点货币)
Output: 最短路径的权重 distance[t], 以及对应的路径

1. 将汇率矩阵 r 转化为图的权重矩阵 w:
   for i = 1 to n:
```

```
        for j = 1 to n:
            if i ≠ j:
                w[i][j] = -log(r[i][j])
            else:
                w[i][j] = 0   # 自环权重为0
```

2. 初始化最短路径数组 distance 和前驱数组 predecessor：

```
distance = [∞, ∞, ..., ∞]   (大小为 n)
predecessor = [-1, -1, ..., -1]   (大小为 n)
distance[s] = 0                  (起点的距离为 0)
```

3. 进行 Bellman-Ford 松弛操作：

```
for k = 1 to n-1:                   (最多 n-1 次迭代)
    for i = 1 to n:                 (遍历每条边)
        for j = 1 to n:
            if distance[i] + w[i][j] < distance[j]:
                distance[j] = distance[i] + w[i][j]
                predecessor[j] = i
```

4. 检测负权环（用于验证是否存在套利环）：

```
for i = 1 to n:
    for j = 1 to n:
        if distance[i] + w[i][j] < distance[j]:
            print("存在负权环（套利环）")
            return
```

5. 构造从 s 到 t 的路径：

```
path = []
current = t
while current ≠ -1:
    path.append(current)
    current = predecessor[current]
path.reverse()
```

6. 输出结果：

```
print("最优路径权重:", distance[t])
print("最优路径:", path)
```

3. 时间复杂度：和Bellman-Ford算法一致，都为$O(|V|+|E|)$

(b):

1. 思路，监测负权环，使用Bellman-Ford算法即可
2. 伪代码：为上一问的一个部分
3. 时间复杂度：和上一问一致

Problem 3

(a):

1. 思路：设传递闭包关系图为一个二维数组，每次加入一个元素，就检测当前元素的行列的交点中有没有可改变的点即可。
2. 伪代码：

```
function Element_insert((u,v),T):
        if (T[u][v]==1)
                return 0

        else
                T[u][v]=1
                for i in range(n)
                        for j in range(n)
                                if T[i][u] == 1 and T[v][j]==1
                                        T[i][j]==1

        return 0
```

3. 时间复杂度：
   显然两层循环的时间复杂度为$O(|V|^2)$。

(b)

1. 思路：设G有两个强连通分量A、B，且$|A|=|B|=\frac{|V|}{2}, a\in A, b\in B$。
   `Element_insert((a,b),T)`，即可。
2. 时间复杂度证明：对$\forall x\in A, y\in B$有$(x,y)=0$，在`Element_insert((a,b),T)`后，$\forall x\in A, y\in B$有$(x,y)=1$，修改的边数量为$|V|^2/4$，显然为$O(|V|^2)$

(c):

1. 思路：运行x次(a)中的算法
2. 伪代码

```
function x_insert(S,T):
        for (u,v) in S:
                Element_insert((u,v),T)
```

```
        return 0
```

3. 时间复杂度：
   执行x次insert，时间复杂度为$O(|x|\cdot|V|^2)$，在最坏情况下x为V^2。但是从摊还分析的视角来看，插入的(u,v)有可能已经是1了，此时本次insert的时间复杂度为O(1)，而矩阵一共需要$O(|V|^2)$次从0-1的修改，而0-1后不可能变回0，因此本算法的时间复杂度和warshall算法的更新传递闭包一致都为$O(|V|^3)$。

Problem 4

(a)：

对一个子问题$A_p\cdots A_q$，且$q-p=n-1$，需要依赖其中所有的$i,j\in[p,q]$

1. 点数量显然为所有可能的(i,j)的数量和各个矩阵本身，$n+\frac{n(n-1)}{2}=\frac{n(n+1)}{2}$。
2. 边的数量，对所有$k\in[p,q]$，有p-q条边与(p,k)和(k+1,j)连接，因此最后答案为$\sum_{i=1}^{n}\sum_{j=1}^{n}((j-i)=O(n^3)$。

(b)：

子问题只有规模相同，而每个子问题中需要处理的数具体来说是不同的，不能通过数组或其其他方式进行记忆化和复用。

Problem 5

(a):

1. 思路：只需要在递归表达式中增加减去c这一项即可，最后的表达式为
   定义R(n)为长度为n时的最大收益，R(0)

$$R(n)=\max_{1\le i\le n}(p_i+p_{n-i}-c+R(i)+R(n-i))$$

2. 伪代码

```
# 输入：钢条各长度段价格列表 p，钢条总长度 n，每次切割的固定成本 c
# 输出：长度为 n 的钢条能获得的最大收益
def rod_cutting_with_cost(p, n, c):
    R = [0] * (n + 1)
    for i in range(1, n + 1):
        max_value = float('-inf')
        for j in range(1, i + 1):
            value = p[j - 1] + R[i - j] - c
            max_value = max(max_value, value)
```

```
        dp[i] = max_value
    return R[n]
```

3. 时间复杂度:
   两层循环，时间复杂度为$O(n^2)$。

(b):

1. 思路
   1. 定义子问题：设 $dp[k]$ 表示长度为 k的杆所能获得的最大收益。
   2. 状态转移: 对于每种长度 j，考虑每种可能的切割数量 m从 1 到l_j:
      - 我们可以做出选择使用 m 个长度为 j 的杆段，切割后的总长度为 $j \times m$。
      - 对每种选择，更新 $dp[k]$。
   3. 递推关系：
      $dp[k] = \max\left(dp[k], dp[k - j \times m] + p[j] \times m\right)$　对于所有 m 使得 $j \times m \leq k$
2. 伪代码

```
function rod_cutting_withlimits(n, p, l):
    dp = array of size (n+1) initialized to 0

    for j from 1 to n: // 遍历每种长度
        for k from 0 to n: // 遍历总长度
            for m from 1 to l[j]: // 遍历每种长度 j
                if j * m <= k: // 确保不超出当前长度
                    dp[k] = max(dp[k], dp[k - j * m] + p[j] * m)
                else:
                    break // 如果长度超过，不用继续检查

    return dp[n]
```

3. 时间复杂度：

- 外层循环 j 和中层循环 k 的时间复杂度为 $O(n^2)$。
- 内层循环 m 的最大次数为 $O(l_j)$，整个结构的时间复杂度为：

$$O(n^2 \cdot L) \quad (\text{其中 } L = \max_{1 \leq j \leq n} l_j)$$

Problem 6

(a)

1. 状态转移方程：
   if s[i]=s[j]: dp[i][j]=dp[i-1][j-1]+2
   else dp[i][j]=max(dp[i][j-1],dp[i+1][j])
2. 伪代码：

```
def longest_palindrome_subsequence(s):
    n = len(s)
    dp = [[0] * n for _ in range(n)]

    # 初始化边界
    for i in range(n):
        dp[i][i] = 1  # 单个字符的回文长度为 1

    # 填充 dp 表，从短区间到长区间
    for length in range(2, n + 1):  # 子串长度从 2 开始
        for i in range(n - length + 1):  # 起始位置
            j = i + length - 1  # 结束位置
            if s[i] == s[j]:
                dp[i][j] = dp[i + 1][j - 1] + 2
            else:
                dp[i][j] = max(dp[i + 1][j], dp[i][j - 1])

    return dp[0][n - 1]
```

3. 时间复杂度
   两层循环，时间复杂度明显为$O(n^2)$

(b)

1. 状态转移方程
   if s[i]=s[j]: dp[i][j]=dp[i-1][j-1]
   else dp[i][j]=min(dp[i][j-1],dp[i+1][j])+1
2. 伪代码

```
def shortest_palindrome_supersequence(s):
    n = len(s)
    dp = [[0] * n for _ in range(n)]

    # 初始化边界
    for i in range(n):
        dp[i][i] = 0  # 单个字符本身是回文

    # 填充 dp 表，从短区间到长区间
```

```
    for length in range(2, n + 1):  # 子串长度从 2 开始
        for i in range(n - length + 1):  # 起始位置
            j = i + length - 1  # 结束位置
            if s[i] == s[j]:
                dp[i][j] = dp[i + 1][j - 1]
            else:
                dp[i][j] = min(dp[i + 1][j], dp[i][j - 1]) + 1

    # 最短回文超序列的长度
    return n + dp[0][n - 1]
```

3. 时间复杂度：
   两层循环，时间复杂度明显为$O(n^2)$