PS12-231830106

Problem 1

(a):

[1、2、10、5]
假设玩家二绝对理智
若采用贪心策略，玩家一至多获得7分
不采用贪心，则最多获得11分

(b):

1. 思路
   设$DP[i][j]$表示从s_i到s_j玩家最佳策略获得的分数
   有
   $DP[i][j] = dp[i][j] = max(vi + min(DP[i+2][j], DP[i+1][j-1]), vj + min(DP[i+1][j-1],$
   初始化:
   $DP[i][i] = v_i$
   $DP[i][i+1] = max(v_i, v_{i+1})$
   然后通过自底向上的迭代就可以更新DP
2. 伪代码

```
function OptimalStrategy(v[]):
    n = length(v)
    dp = new Array(n) of Array(n) filled with 0

    // 初始化
    for i from 0 to n-1:
        dp[i][i] = v[i]
        if i < n-1:
            dp[i][i+1] = max(v[i], v[i+1])

    // 动态规划计算
    for length from 3 to n:
        for i from 0 to n-length:
            j = i + length - 1
            dp[i][j] = max(v[i] + min(dp[i+2][j], dp[i+1][j-1]),
                           v[j] + min(dp[i+1][j-1], dp[i][j-2]))

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

```
v = 输入卡片序列
maxValue = OptimalStrategy(v)
print("第一个玩家能获得的最大总价值为:", maxValue)
```

3. 时间复杂度：
   初始化部分的时间复杂度为$O(n)$，动态规划部分是两层循环，因此时间复杂度为$O(n^2)$，总时间复杂度为$O(n^2)$。

Problem 2

1. 思路：
   设$DP[i][0]$表示点i不在顶点覆盖集合中时。以 u 为根的子树的最小顶点覆盖大小，
   $DP[i][0]$则表示在顶点覆盖集合中时，以 u 为根的子树的最小顶点覆盖大小。
   $if(u,v)then$
   $DP[u][0] = DP[v][1]$
   $DP[u][1] = min(DP[v][1], DP[v][0])$
   从任意一个点出发通过dfs的拓扑排序进行逐个边点计算
2. 伪代码

```
function min_vertex_cover(tree):
    1. Choose an arbitrary root node for the tree.
    2. Define dp[u][0] and dp[u][1] for all nodes u.
    3. Perform a DFS from the root node:
        a. For each node u:
            i. dp[u][0] = sum(dp[v][1] for all children v of u)
            ii. dp[u][1] = 1 + sum(min(dp[v][0], dp[v][1]) for all
children v of u)
    4. Return min(dp[root][0], dp[root][1]).
```

3. 时间复杂度
   遍历图的点和边一次，因此时间复杂度为$O(|V|+|E|)$。

Problem 3

1. 思路：
   对数组中的元素A[k]，有三种可能性使得A[k]被包含在最大乘积中：
   1. A[k]作为最大乘积的开头元素
   2. A[k]>0，乘以之前的最大乘积
   3. A[k]<0，乘以之前的最小乘积
   因此有递推公式
   $cur\_max = max(A[k], A[k] \times pre\_max, A[k] \times pre\_min)$
   $cur\_min = min(A[k], A[k] \times pre\_max, A[k] \times pre\_min)$.

$result = max(result, cur\_max)$
$pre\_max = cur\_max, pre\_min = cur\_min$

2. 伪代码

```
Function MaxProductSubarray(A):
    Input: Array A of length n (can be positive, negative, or zero)
    Output: Maximum product of a contiguous subarray in A

    # Initialize variables
    prev_max ← 1          # Maximum product ending at the previous step
    prev_min ← 1          # Minimum product ending at the previous step
    result   ← 1          # Global maximum result (accounts for empty
interval)

    For each element num in A:
        If num == 0:      # Handle zeros in the array
            prev_max ← 1  # Reset to 1 (subarray restarts)
            prev_min ← 1  # Reset to 1
            Continue      # Skip further calculations for this step

        # Calculate the current max and min product
        curr_max ← max(num, num * prev_max, num * prev_min)
        curr_min ← min(num, num * prev_max, num * prev_min)

        # Update the global result
        result ← max(result, curr_max)

        # Update previous max and min for the next iteration
        prev_max ← curr_max
        prev_min ← curr_min

    Return result
```

3. 时间复杂度:
   只遍历一遍数组因此时间复杂度为$O(n)$。
4. 空间复杂度:
   没有使用额外数组，因此空间复杂度$O(1)$。

Probelm 4

(a):

伪代码

```
Function Knapsack(n, W, weights, values):
    Input:
        n: Number of items
        W: Maximum weight capacity of the knapsack
        weights: Array of weights of the items (1 to n)
        values: Array of values of the items (1 to n)
    Output:
        Maximum value achievable with weight limit W

    # Step 1: Initialize DP table
    Let dp[0...n][0...W] = 0  # A 2D array initialized to 0

    # Step 2: Populate DP table
    For i from 1 to n:        # Iterate over all items
        For w from 0 to W:    # Iterate over all capacities
            If weights[i-1] <= w:
                # Item i can be included
                dp[i][w] = max(dp[i-1][w], dp[i-1][w - weights[i-1]] + 
values[i-1])
            Else:
                # Item i cannot be included
                dp[i][w] = dp[i-1][w]

    # Step 3: Return the result
    Return dp[n][W]  # Maximum value for n items and weight limit W
```

时间复杂度；
两层循环，一层为$0--W$，一层为$0--N$，因此总时间复杂度为$O(Wn)$。

(b):

否，输入规模为$k = \log W$，因此设时间复杂度为$T(n, w)$
有$T(n, w) = O(n \cdot 2^{\log w}) = O(n \cdot 2^k)$
因此不是多项式时间

Optional Problem 1

(a) 证明 GRAPH-ISOMORPHISM ∈ NP

1. 图同构的语言描述：
GRAPH-ISOMORPHISM 问题要求判断两个图 G_1 和 G_2 是否是同构的，即是否存在一个双射 $f : V_1 \to V_2$，使得：$$
(u, v) \in E_1 \iff (f(u), f(v)) \in E_2

2.证明$GRAPH-ISOMORPHISM \in NP$： -1： 如果\$$G_1$\$和\$$G_2$\$是同构的，则有\$$f: V_1 -$

(b) 证明 TAUTOLOGY ∈ coNP

1. 重言式的语言描述：
   TAUTOLOGY 问题要求判断布尔公式 ϕ 是否在所有变量赋值下的结果均为 1。即证：

$$\phi \in \text{TAUTOLOGY} \iff \forall x \in \{0,1\}^k, \phi(x) = 1$$

2. $coNP$ **的定义：**
一个语言 $L \in coNP$ 当且仅当其补集 L^C 属于 NP。
L^C 是指所有不属于 L 的实例集合。
对于 TAUTOLOGY：$$
\phi \notin \text{TAUTOLOGY} \iff \exists x \in {0, 1}^k, \phi(x) = 0

$$3.证明1.$$

\text{TAUTOLOGY}^C = {\phi : \phi \text{ 在某个变量赋值下结果为 } 0}
\$如果\$$\phi \notin \text{TAUTOLOGY}$，那么存在一个具体的布尔变量赋值 x^* 使得 $\phi(x^*) = 0$。
2.
   1. 给定 ϕ 和赋值 x^*，计算 $\phi(x^*)$ 的值。
   2. 代入计算公式，值是否为 0：
      代入公式计算可以在多项式时间内完成，因为公式的大小决定了计算复杂度，变量数量 k 是输入规模的一部分。
   3. - 对于任何不属于 TAUTOLOGY 的公式 ϕ，存在一个多项式规模的“证明” x^*，且验证 $\phi(x^*) = 0$ 的过程可以在多项式时间内完成。
      - 因此，$\text{TAUTOLOGY}^C \in NP$，从而 $\text{TAUTOLOGY} \in coNP$。

Optional Problem 2

原题即证：

1. 对于 P 类中除了 $\emptyset$ 和 $\{0,1\}^*$ 以外的任意语言 L，L 都是 P 完备的。
2. 空集 $\emptyset$ 和全语言 $\{0,1\}^*$ 不是 P 完备的语言。

1. P 类中除了 $\emptyset$ 和 $\{0,1\}^*$ 以外的语言是 P 完备的

设 $L \in P$，且 $L \neq \emptyset$ 且 $L \neq \{0,1\}^*$。
由于 L 是非平凡语言，存在至少一个字符串 $a \in L$ 和一个字符串 $b \notin L$。
构造一个多项式时间归约函数 f，使得对于任意 $L' \in P$，都有 $L' \leq_P L$。

$$f(x) = \begin{cases} a & \text{如果 } x \in L' \\ b & \text{如果 } x \notin L' \end{cases}$$

- 多项式时间可计算性： 由于 $L' \in P$，存在一个多项式时间的判定算法来判断 $x \in L'$。因此，函数 f 可以在多项式时间内决定要输出 a 还是 b。

- 正确性：对于任意 x，有：$$
x \in L' \iff f(x) = a \in L \
x \notin L' \iff f(x) = b \notin L

  因此，$x \in L'$当且仅当$f(x) \in L$，满足$L' \leq_P L$。由于对于任意$L' \in P$都存在这样的归

2. $\emptyset$ 和 $\{0,1\}^*$ 不是 P 完备的语言

- 空集 $\emptyset$:
  假设存在某个非空语言 $L' \in P$，且 $L' \leq_P \emptyset$。即存在一个多项式时间可计算的函数 f，使得对于所有 x，有：

$$x \in L' \iff f(x) \in \emptyset$$

  由于 $\emptyset$ 中没有任何元素，无论 $f(x)$ 如何，总有 $f(x) \notin \emptyset$。因此：

$$x \in L' \implies f(x) \notin \emptyset$$

  即 L' 必须是空集，否则无法满足归约关系。因此，只有 $L' = \emptyset$ 时，归约才成立。对于非空语言，无法将其归约到 $\emptyset$。
- 全语言 $\{0,1\}^*$:
  假设存在某个非全语言 $L' \in P$，且 $L' \leq_P \{0,1\}^*$。根据归约的定义，需要存在一个多项式时间可计算的函数 f，使得对于所有 x，有：

$$x \in L' \iff f(x) \in \{0,1\}^*$$

  由于 $\{0,1\}^*$ 包含所有可能的字符串，无论 $f(x)$ 输出什么，都有 $f(x) \in \{0,1\}^*$。则：

$$x \in L' \iff \text{True}$$

  即 L' 必须是全语言 $\{0,1\}^*$，否则无法满足归约关系。对于非全语言，无法将其归约到 $\{0,1\}^*$。
  因此，$\emptyset$ 和 $\{0,1\}^*$ 不是 P 完备的语言。

即证：
在 P 类中，只有空集 $\emptyset$ 和全语言 $\{0,1\}^*$ 不是 P 完备的语言。其余所有非平凡语言都是 P 完备的。

Optional Problem 3

(a)

一. 思路：
采用逐步确定变量赋值的方法：

1. 对于布尔公式 ϕ 中的每一个变量x_i，尝试将其赋值为 `True`，并检查剩余公式在此赋值下是否仍然可满足。如果在 $x_i = \text{True}$ 的情况下公式可满足，则将 x_i 设为 `True`；否则，将其设为 `False`。

2. 重复上述过程，直到所有变量均被赋值。
3. 通过上述过程得到的赋值即为一个可满足的赋值。
   二. 算法：
   设布尔公式 ϕ 包含变量 $x_1, x_2, \ldots, x_n$。
4. 初始化一个空赋值 $S = \{\}$。
5. 对于每个变量 x_i（$i = 1$ 到 n）执行以下步骤：
   - 设置 $x_i = \text{True}$，得到新的公式 $\phi' = \phi$ 且 $x_i = \text{True}$。
   - 使用算法 A 判断 ϕ' 是否可满足。
     - 如果可满足，将 $x_i = \text{True}$ 记录在赋值 S 中。
     - 否则，将 $x_i = \text{False}$ 记录在赋值 S 中。
   - 更新公式 ϕ 为根据当前赋值 S 简化后的公式。
6. 完成所有变量的赋值后，赋值 S 即为一个满足 ϕ 的赋值。

三. 正确性：

- 通过逐一确定每个变量的赋值，确保在每一步都选择一个不会使公式不可满足的赋值。
  最终得到的赋值使得整个公式 ϕ 可满足。
  四. 时间复杂度：
  - 对于每个变量，需要调用一次算法 A 来判断可满足性。假设有 n 个变量，且每次调用 A 的时间为 $O(p(n))$，其中 $p(n)$ 是多项式时间复杂度。因此总时间复杂度为 $O(n \cdot p(n))$，仍为多项式时间。

(b)

一、思路：
2-SAT 问题可以通过构建一个蕴含图（Implication Graph并检测其强连通性来在多项式时间内解决。具体步骤如下：

1. 转换为蕴含图：对于每个子句 $x \lor y$，将其转换为两个蕴含式：
   - $\neg x \to y$
   - $\neg y \to x$
     这样，每个子句对应图中的两条有向边。
2. 构建有向图： 每个变量 x 和其否定 $\neg x$ 都被视为图中的节点。根据上述转换，将相应的有向边添加到图中。
3. 检测双变量的可满足性：使用强连通分量算法（如 Kosaraju 算法或 Tarjan 算法）来检测有向图中的强连通分量。如果一个变量和它的否定属于同一个强连通分量，则公式不可满足。否则，公式可满足。
4. 构造赋值：如果公式可满足，通过拓扑排序确定每个变量的赋值。
   二、算法：
5. 输入： 2-CNF 布尔公式 ϕ，其中每个子句有两个文字。
6. 构建蕴含图：
   - 对于每个子句 $x \lor y$，添加两条有向边：
     - $\neg x \to y$

- $\neg y \to x$

7. 检测强连通分量：
    - 对构建的有向图进行强连通分量分析。
    - 检查是否存在变量 x 使得 x 和 $\neg x$ 在同一个强连通分量中。
    - 如果存在，公式不可满足，返回“不满足”。
    - 否则，公式可满足，继续。
    三、正确性：

- 正确性：
    - 通过构建蕴含图并检测强连通分量，可以有效地发现公式中的矛盾，即变量和其否定同时为真。这确保了算法正确地判断 2-SAT 公式的可满足性。构造赋值步骤确保了找到一个满足公式的具体赋值。
    四、时间复杂度
    - 构建蕴含图的时间复杂度为 $O(m)$，其中 m 是公式中的子句数量。强连通分量的检测（使用 Kosaraju 或 Tarjan 算法）具有时间复杂度 $O(n+m)$，其中 n 是变量的数量。因此总时间复杂度为 $O(n+m)$，是多项式时间。