习题实验 5

提交文件格式为 PDF。

姓名: xxx
学号: xxxxxxxx

题目 **5-1.** 简答题

(a) 简述回溯算法求解问题的基本步骤。

回溯算法是一种通过试探和回退来寻找问题所有解的方法。其基本步骤如下：

1. 定义问题的解空间：确定问题的状态空间，包括初始状态和目标状态。
2. 确定约束条件：定义问题的限制条件，用于判断当前状态是否合法。
3. 设计递归结构：
   - 递归终止条件：找到问题的一个解或者无法继续搜索时终止。
   - 搜索过程：在当前状态下，尝试进行各种可能的选择。
   - 约束函数：判断当前选择是否满足约束条件。
   - 状态更新：如果选择有效，更新状态并继续搜索。
   - 回溯操作：如果当前选择不能得到有效解，撤销选择，返回上一状态，继续尝试其他选择。

(b) 简述求解 N 皇后问题的回溯算法步骤，分析该算法的时间复杂度并给出一个具体的算例。

N 皇后问题是在 N×N 的棋盘上放置 N 个皇后，使得它们互不攻击（不在同一行、同一列或同一对角线上）。回溯算法步骤如下：

1. 创建一个 N×N 的棋盘，初始为空。

2. 从第一行开始，尝试在每一列放置皇后。
3. 检查当前位置是否可行（不与之前放置的皇后冲突）。
4. 如果可行，就在该位置放置一个皇后，然后递归处理下一行。
5. 如果当前行的所有列都不可行或递归返回失败，则回溯到上一行，尝试其他列。
6. 当成功放置 N 个皇后时，记录一个解。

时间复杂度分析：在最坏情况下，需要尝试所有可能的放置方式。第一行有 N 个可能的位置，第二行最多有 N-1 个可能位置，以此类推。所以最坏情况下的时间复杂度是 O(N!)。但实际上由于剪枝，实际复杂度通常远小于 O(N!)。

算例（4 皇后问题）：

我们通过代码解决了 4 皇后问题，得到了 2 个解决方案：

```
解决方案 1:
.Q..
...Q
Q...
..Q.

解决方案 2:
..Q.
Q...
...Q
.Q..
```

(c) 简述装载问题回溯算法的求解步骤，分析该算法的时间复杂度并给出一个具体的算例。

装载问题是指有一批物品和一艘船，每个物品有一定重量，船有一定的载重量，如何选择物品使得船上的物品总重量最大且不超过船的载重量。其回溯算法步骤如下：

1. 定义状态空间：对每个物品，选择装载或不装载。
2. 对每个物品递归处理：

- 尝试装载当前物品，更新当前总重量，递归处理下一个物品。
- 尝试不装载当前物品，递归处理下一个物品。
- 在过程中记录满足载重限制的最大重量方案。

3. 当所有物品都考虑完毕时，返回最优解。

时间复杂度分析：对于 n 个物品，每个物品有装或不装两种选择，所以最坏情况下的时间复杂度是 $O(2^n)$。但通过剪枝（如当前重量已超过载重量时立即回溯），可以减少实际计算量。

算例：我们用代码解决了一个装载问题：

```
物品重量: [10, 40, 30, 50, 35, 25]
船的最大载重: 100
能够装载的最大重量: 100
选择的物品索引: [0, 1, 3]
选择的物品重量: [10, 40, 50]
```

(d) 简述图的 m 着色问题回溯算法的求解步骤，分析该算法的时间复杂度并给出一个具体的算例。

图的 m 着色问题是指给定一个无向图和 m 种颜色，如何为图的每个顶点分配一种颜色，使得相邻顶点的颜色不同。其回溯算法步骤如下：

1. 从第一个顶点开始，尝试为每个顶点分配一种颜色。
2. 检查当前颜色分配是否有效（相邻顶点颜色不同）。
3. 如果有效，递归处理下一个顶点。
4. 如果当前顶点的所有颜色都不可行或递归返回失败，则回溯到上一个顶点，尝试其他颜色。
5. 当所有顶点都成功分配颜色时，记录一个解。

时间复杂度分析：对于 n 个顶点和 m 种颜色，每个顶点有 m 种可能的颜色选择，所以最坏情况下的时间复杂度是 $O(m^n)$。但实际上由于剪枝（检查相邻顶点），复杂度通常小于这个上界。

算例：我们通过代码解决了一个 4 顶点 3 着色问题：

```
图的邻接矩阵表示:
```

```
0 1 1 1
1 0 1 0
1 1 0 1
1 0 1 0

使用3种颜色的着色方案:
顶点 0: 颜色 1
顶点 1: 颜色 2
顶点 2: 颜色 3
顶点 3: 颜色 2
```

题目 **5-2.** 给定 n 个不同的字符，打印出由这 n 个字符组成的全排列。

(a) 按照回溯算法该问题，分析其时间复杂度。

使用回溯算法解决全排列问题的基本思路是：从第一个位置开始，依次尝试将每个未使用的字符放在当前位置，然后递归处理下一个位置。算法过程如下：

1. 定义状态：当前已经排列好的字符和剩余未排列的字符。
2. 递归终止条件：所有字符都已被排列，此时记录一个解。
3. 对于当前位置，尝试放置每个未使用的字符：
   - 选择一个未使用的字符放在当前位置。
   - 递归处理下一个位置。
   - 回溯，撤销选择，尝试其他字符。

时间复杂度分析：对于 n 个字符，第一个位置有 n 种选择，第二个位置有 n-1 种选择，以此类推。所以总的时间复杂度是 O(n!)。此外，需要 O(n) 的空间来存储一个排列，总共有 n! 个排列，所以空间复杂度为 O(n • n!)。但是如果只考虑递归调用栈的空间，则空间复杂度为 O(n)。

代码实现中，我们通过交换字符的方式来生成全排列，避免了额外的存储空间，提高了效率。

(b) 如果输入的字符分别是"XYR"，请列出代码的输出。

通过运行我们编写的代码，输入字符"XYR" 的全排列结果如下：

```
XYR
XRY
YXR
YRX
RYX
RXY
```

题目 **5-3.** 有一只老鼠为了寻找食物需要穿过迷宫，假设从迷宫左上角进入右下角出来，请设计算法实现判断老鼠是否能穿过迷宫，若能则对行进路径进行展示。为了描述方便，特作如下设定：

- 在此用二维数组表示迷宫，围墙用 1 表示；通路用 0 表示。
- 此处设为四向迷宫，即在上下左右皆可。
- 迷宫可能是死的，即没有通路，此时输出提示。

(a) 请按照回溯算法给出具体实现。[10 分]

使用回溯算法解决老鼠走迷宫问题的实现思路如下：

1. 从迷宫的左上角开始，向四个方向（右、下、左、上）尝试移动。
2. 对于每个可能的移动，检查该位置是否有效（在迷宫范围内且是通路）。
3. 如果有效，标记当前位置为已访问，并递归尝试从该位置继续移动。
4. 如果从当前位置无法到达终点，则回溯并尝试其他方向。
5. 如果到达右下角，则找到一条有效路径。

代码实现的关键点：

- 使用一个二维数组记录解决方案路径，1 表示路径，0 表示非路径。
- 使用 visited 数组避免重复访问同一位置，防止无限递归。
- 定义四个方向的移动偏移量，方便遍历所有可能的移动方向。
- 通过 is_safe 函数检查位置的有效性。

```
def solve_maze(maze):
    # 迷宫尺寸
    n = len(maze)
```

```
    m = len(maze[0])

    # 创建一个二维数组来存储路径
    solution = [[0 for _ in range(m)] for _ in range(n)]

    # 定义移动方向: 右、下、左、上
    move_x = [0, 1, 0, -1]
    move_y = [1, 0, -1, 0]

    # 创建访问标记数组, 避免重复访问
    visited = [[False for _ in range(m)] for _ in range(n)]

    if not solve_maze_util(maze, 0, 0, solution, move_x, move_y, n, m, visited):
        print("没有找到通往出口的路径")
        return None

    return solution

def solve_maze_util(maze, x, y, solution, move_x, move_y, n, m, visited):
    # 达到右下角, 找到解
    if x == n - 1 and y == m - 1 and maze[x][y] == 0:
        solution[x][y] = 1
        return True

    # 检查当前单元格是否有效且未访问
    if is_safe(maze, x, y, n, m) and not visited[x][y]:
        # 标记当前单元格为已访问
        visited[x][y] = True

        # 标记当前单元格为解决方案路径的一部分
        solution[x][y] = 1
```

```
            # 尝试四个方向移动
            for i in range(4):
                next_x = x + move_x[i]
                next_y = y + move_y[i]

                if solve_maze_util(maze, next_x, next_y, solution, move_x, move_y, n, m
                    return True

            # 如果没有方向可行，回溯
            solution[x][y] = 0
            return False

        return False

    def is_safe(maze, x, y, n, m):
        # 检查是否在迷宫内且是通路
        return 0 <= x < n and 0 <= y < m and maze[x][y] == 0

    def print_solution(solution):
        if solution is None:
            return

        n = len(solution)
        m = len(solution[0])

        print("路径解决方案（1表示路径）: ")
        for i in range(n):
            for j in range(m):
                print(solution[i][j], end=" ")
            print()
```

```
# 测试用例
if __name__ == "__main__":
    maze = [
        [0, 1, 0],
        [0, 0, 0],
        [1, 0, 0]
    ]

    print("迷宫（0表示通路, 1表示墙）: ")
    for row in maze:
        print(" ".join(map(str, row)))
    print()

    solution = solve_maze(maze)

    if solution:
        print_solution(solution)
    else:
        print("未找到路径")
```

(b) 当给出的迷宫如下所示，给出代码输出结果。[5 分]

$$maze = \begin{bmatrix} 0 & 1 & 0 \\ 0 & 1 & 0 \\ 1 & 0 & 0 \end{bmatrix}$$

对于给定的迷宫，我们首先尝试运行代码，发现没有可行的路径：

```
迷宫（0表示通路, 1表示墙）:
0 1 0
0 1 0
1 0 0
```

没有找到通往出口的路径

这是因为中间的墙 (1) 阻断了所有可能的路径。如果我们修改迷宫，将中间的墙改为通路：

$$maze = \begin{bmatrix} 0 & 1 & 0 \\ 0 & 0 & 0 \\ 1 & 0 & 0 \end{bmatrix}$$

然后运行代码，得到一条可行的路径：

迷宫（0表示通路，1表示墙）：

```
0 1 0
0 0 0
1 0 0
```

路径解决方案（1表示路径）：

```
1 0 0
1 1 1
0 0 1
```

这条路径表示老鼠从左上角 (0,0) 出发，向下移动到 (1,0)，然后向右移动到 (1,1) 和 (1,2)，最后向下移动到右下角 (2,2)，成功找到了出口。