【回溯】-加密算法

题目描述与示例

题目描述

有一种特殊的加密算法，明文为一段数字串，经过密码本查找转换，生成另一段密文数字串。规则如下

1. 明文为一段数字串由 `0-9` 组成
2. 密码本为数字 `0-9` 组成的二维数组
3. 需要按明文串的数字顺序在密码本里找到同样的数字串，密码本里的数字串是由相邻的单元格数字组成，上下和左右是相邻的，注意:对角线不相邻，同一个单元格的数字不能重复使用。
4. 每一位明文对应密文即为密码本中找到的单元格所在的行和列序号(序号从 `0` 开始)组成的两个数字。如明文第 `i` 位 `Data[i]` 对应密码本单元格为 `Book[X][Y]` ，则明文第 `i` 位对应的密文为 `X Y` ， `X` 和 `Y` 之间用空格隔开。如果有多条密文，返回字符序最小的密文。如果密码本无法匹配，返回 `"error"` .

请你设计这个加密程序。

示例 1:

密码本:

```
{0,0,2},
{1,3,4},
{6,6,4}
```

明文 `"3"` ，密文 `"1 1"`

示例 2:

密码本:

```
{0,0,2},
{1,3,4},
{6,6,4}
```

明文 `"0 3"` ,密文 `"0 1 1 1"`

示例 3:

密码本:

```
{0,0,2,4}
{1,3,4,6}
{3,4,1,5}
{6,6,6,5}
```

明文 `"0 0 2 4"` ，密文 `"0 0 0 1 0 2 0 3"` 和 `"0 0 0 1 0 2 1 2"` ，返回字典序小的 `"0 0 0 1 0 2 0 3"`

输入描述

第一行输入 `1` 个正整数 `N` ，代表明文的长度 `(1 <= N <= 9)`

第二行输入 `N` 个明文数字组成的序列 `Data[i]` (整数， `0 <= Data[i] <= 9`)

第三行输入 `1` 个正整数 `M` ，(`1 <= M <= 9`)

接下来输入一个 `M*M` 的矩阵代表密码本 `Book[i][i]` ，(整数， `0 <= Book[i][i] <= 9`)

输出描述

如明文 第 `i` 位 `Data[i]` 对应密码本单元格为 `Book[i][j]` ，则明文第 `i` 位对应的密文为 `X Y` ， `X` 和 `Y` 之间用空格隔开。如果有多条密文，**返回字符序最小的密文**。如果密码本无法匹配，返回 `"error"` 。

示例一

输入

```
2
```

```
2 0 3
3 3
4 0 0 2
5 1 3 4
6 6 6 4
```

输出

```
1 0 1 1 1
```

示例二

输入

```
1 4
2 0 0 2 4
3 4
4 0 0 2 4
5 1 3 4 6
6 3 4 1 5
7 6 6 6 5
```

输出

```
1 0 0 0 1 0 2 0 3
```

解题思路

> 注意，本题和LeetCode79. 单词搜索、【回溯】2023C-找到它非常类似。唯一的区别是，题目不保证答案是唯一的，当存在多个合适的密文的时候，需要返回字典序最小的那个。

本题基本思路和【回溯】2023C-找到它一致。本题需要着重考虑**最小字典序**的问题。

这个时候，搜索的方向数组 `DIRECTIONS` 里的顺序就非常重要了。假设当前点为 `(x, y)`，那么其近邻点为

- 上方：`(x-1, y)`
- 左方：`(x, y-1)`
- 右方：`(x, y+1)`
- 下方：`(x+1, y)`

显然，如果存在多个近邻点同时满足下一个字符的时候，按照**上、左、右、下**这个顺序来搜索的话，一定能够得到最小的字典序，因为**坐标为更小字典序的近邻点被优先搜索了**。

这也是极少数的，我们需要特别注意方向数组 `DIRECTIONS` 的顺序的题目。即

```
DIRECTIONS = [(-1, 0), (0, -1), (0, 1), (1, 0)]
```

代码

Python

```python
# 题目：2023C-加密算法
# 分值：200
# 作者：许老师-闭着眼睛学数理化
# 算法：回溯
# 代码看不懂的地方，请直接在群上提问


# 全局的方向数组，表示上下左右移动四个方向
# 【特别注意】：此处的顺序是非常重要的，必须按照【上、左、右、下】来排布
# 这样才能使得计算得到的密文一定是字典序最小
DIRECTIONS = [(-1, 0), (0, -1), (0, 1), (1, 0)]


# 构建回溯函数，各个参数的含义为
# grid:          原二维矩阵
```

```python
# M:                原二维矩阵的大小M
# check_list:       大小和grid一样的检查列表，用于判断某个点是否已经检查过
# x,y:              当前在grid中的点的坐标
# s:                待搜索的明文
# s_idx:            待搜索的明文此时遍历到的索引位置
# path:             当前路径
def backtracking(grid, M, check_list, x, y, s, s_idx, path):
    # 声明全局变量isFind
    global isFind
    # 如果在之前的回溯中已经找到了最小字典序的密文，直接返回
    if isFind:
        return
    # 若此时s_idx等于s的长度-1，即len(s)-1
    # 说明s中的所有数字都在grid中找到了
    # 修改isFind为True，输出答案，同时终止递归
    if s_idx == len(s) - 1:
        isFind = True
        print(" ".join(f"{item[0]} {item[1]}" for item in path))
        return
    # 遍历四个方向，获得点(x,y)的近邻点(nx,ny)
    for dx, dy in DIRECTIONS:
        nx, ny = x+dx, y+dy
        # (nx,ny)必须满足以下三个条件，才可以继续进行回溯函数的递归调用
        # 1. 不越界；2. 尚未检查过；
        # 3.在grid中的值grid[nx][ny]为s的下一个字符s[s_idx+1]
        if 0 <= nx < M and 0 <= ny < M and check_list[nx][ny] == False and grid[nx][ny] == s[s_idx+1]:
            # 状态更新：将点(nx,ny)在check_list中的状态更新为True，更新path末尾
            check_list[nx][ny] = True
            path.append((nx, ny))
            # 回溯：将点(nx,ny)传入回溯函数中，注意此时s_idx需要+1
            backtracking(grid, M, check_list, nx, ny, s, s_idx+1, path)
            # 回滚：将点(nx,ny)在check_list中的状态重新修改回False，将path末尾的函数弹出
            check_list[nx][ny] = False
            path.pop()

# 输入明文长度N
N = int(input())
# 输入待查找的明文
s = input().split()
# 输入密码本的行数列数M
M = int(input())
# 构建密码本二维网格
grid = list()
for _ in range(M):
    grid.append(input().split())
```

```python

# 构建全局变量isFind，初始化为False
isFind = False

# 构建大小和grid一样的检查数组check_list
# 用于避免出现重复检查的情况
check_list = [[False] * M for _ in range(M)]
# 双重遍历整个二维网格grid
for i in range(M):
    for j in range(M):
        # 找到点(i,j)等于s的第一个数字
        # 则点(i,j)可以作为递归的起始位置
        if grid[i][j] == s[0]:
            # 将点(i,j)在check_list中设置为已检查过
            check_list[i][j] = True
            # 回溯函数递归入口，path初始为储存点(i,j)
            backtracking(grid, M, check_list, i, j, s, 0, [(i, j)])
            # 将点(i,j)在check_list中重置为未检查过，因为本次回溯不一定找到答案
            check_list[i][j] = False
            # 如果在回溯中，全局变量isFind被改为True，说明找到了明文，直接退出循环
            if isFind:
                break
    # 关于i的循环同理，找到明文之后直接退出循环
    if isFind:
        break

# 如果最终没找到明文，输出error
if not isFind:
    print("error")
```

Java

```java
import java.util.ArrayList;
import java.util.List;
import java.util.Scanner;

public class Main {
    // 全局的方向数组，表示上下左右移动四个方向
    // 【特别注意】：此处的顺序是非常重要的，必须按照【上、左、右、下】来排布
    // 这样才能使得计算得到的密文一定是字典序最小
    static final int[][] DIRECTIONS = {{-1, 0}, {0, -1}, {0, 1}, {1, 0}};
    static boolean isFind = false;

    // 构建回溯函数
```

```
    static void backtracking(String[][] grid, int M, boolean[][] checkList, int x, int y, String[] s, int sIdx, List<List<Integer>> path) {
        // 如果在之前的回溯中已经找到了最小字典序的密文，直接返回
        if (isFind) return;
        // 若此时sIdx等于s的长度-1，即N-1
        // 说明s中的所有数字都在grid中找到了
        // 修改isFind为True，输出答案，同时终止递归
        if (sIdx == s.length - 1) {
            isFind = true;
            StringBuilder sb = new StringBuilder();
            for (List<Integer> point : path) {
                sb.append(point.get(0)).append(" ").append(point.get(1)).append(" ");
            }
            System.out.println(sb);
            return;
        }
        // 遍历四个方向，获得点(x,y)的近邻点(nx,ny)
        for (int[] dir : DIRECTIONS) {
            int nx = x + dir[0];
            int ny = y + dir[1];
            // (nx,ny)必须满足以下三个条件，才可以继续进行回溯函数的递归调用
            // 1. 不越界；2. 尚未检查过；
            // 3.在grid中的值grid[nx][ny]为s的下一个字符s[sIdx+1]
            if (nx >= 0 && nx < M && ny >= 0 && ny < M && !checkList[nx][ny] && grid[nx][ny].equals(s[sIdx + 1])) {
                // 状态更新：将点(nx,ny)在checkList中的状态更新为True，更新path末尾
                checkList[nx][ny] = true;
                List<Integer> point = new ArrayList<>();
                point.add(nx);
                point.add(ny);
                path.add(point);
                // 回溯：将点(nx,ny)传入回溯函数中，注意此时sIdx需要+1
                backtracking(grid, M, checkList, nx, ny, s, sIdx + 1, path);
                // 回滚：将点(nx,ny)在checkList中的状态重新修改回False，将path末尾的点弹出
                checkList[nx][ny] = false;
                path.remove(path.size() - 1);
            }
        }
    }

    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        int N = scanner.nextInt();
        String[] s = new String[N];
        for (int i = 0; i < N; i++) {
```

```
            s[i] = scanner.next();
        }
        int M = scanner.nextInt();
        String[][] grid = new String[M][M];
        for (int i = 0; i < M; i++) {
            for (int j = 0; j < M; j++) {
                grid[i][j] = scanner.next();
            }
        }
        // 构建大小和grid一样的检查数组checkList
        // 用于避免出现重复检查的情况
        boolean[][] checkList = new boolean[M][M];
        // 双重遍历整个二维网格grid
        for (int i = 0; i < M; i++) {
            for (int j = 0; j < M; j++) {
                // 找到点(i,j)等于s的第一个数字
                // 则点(i,j)可以作为递归的起始位置
                if (grid[i][j].equals(s[0])) {
                    // 将点(i,j)在checkList中设置为已检查过
                    checkList[i][j] = true;
                    List<List<Integer>> path = new ArrayList<>();
                    List<Integer> point = new ArrayList<>();
                    point.add(i);
                    point.add(j);
                    path.add(point);
                    // 回溯函数递归入口，path初始为储存点(i,j)
                    backtracking(grid, M, checkList, i, j, s, 0, path);
                    // 将点(i,j)在checkList中重置为未检查过，因为本次回溯不一定找到答
案
                    checkList[i][j] = false;
                    // 如果在回溯中，全局变量isFind被改为True，说明找到了明文，直接退
出循环
                    if (isFind) {
                        break;
                    }
                }
            }
            // 关于i的循环同理，找到明文之后直接退出循环
            if (isFind) {
                break;
            }
        }
        // 如果最终没找到明文，输出error
        if (!isFind) {
            System.out.println("error");
        }
    }
```

```
}

```

C++

```cpp
#include <iostream>
#include <vector>
#include <string>

using namespace std;

// 全局的方向数组，表示上下左右移动四个方向
// 【特别注意】：此处的顺序是非常重要的，必须按照【上、左、右、下】来排布
// 这样才能使得计算得到的密文一定是字典序最小
const vector<vector<int>> DIRECTIONS = {{-1, 0}, {0, -1}, {0, 1}, {1, 0}};
bool isFind = false;

// 构建回溯函数
void backtracking(vector<vector<string>>& grid, int M, vector<vector<bool>>& checkList, int x, int y, vector<string>& s, int sIdx, vector<vector<int>>& path) {
    // 如果在之前的回溯中已经找到了最小字典序的密文，直接返回
    if (isFind) return;
    // 若此时sIdx等于s的长度-1，即N-1
    // 说明s中的所有数字都在grid中找到了
    // 修改isFind为True，输出答案，同时终止递归
    if (sIdx == s.size() - 1) {
        isFind = true;
        for (int i = 0; i < path.size(); ++i) {
            cout << path[i][0] << " " << path[i][1] << " ";
        }
        cout << endl;
        return;
    }
    // 遍历四个方向，获得点(x,y)的近邻点(nx,ny)
    for (auto& dir : DIRECTIONS) {
        int nx = x + dir[0];
        int ny = y + dir[1];
        // (nx,ny)必须满足以下三个条件，才可以继续进行回溯函数的递归调用
        // 1. 不越界；2. 尚未检查过；
        // 3.在grid中的值grid[nx][ny]为s的下一个字符s[sIdx+1]
        if (nx >= 0 && nx < M && ny >= 0 && ny < M && !checkList[nx][ny] && grid[nx][ny] == s[sIdx + 1]) {
            // 状态更新：将点(nx,ny)在checkList中的状态更新为True，更新path末尾
            checkList[nx][ny] = true;
```

```
38              path.push_back({nx, ny});
39              // 回溯：将点(nx,ny)传入回溯函数中，注意此时sIdx需要+1
40              backtracking(grid, M, checkList, nx, ny, s, sIdx + 1, path);
41              // 回滚：将点(nx,ny)在checkList中的状态重新修改回False，将path末尾的点弹
    出
42              checkList[nx][ny] = false;
43              path.pop_back();
44          }
45      }
46  }
47
48  int main() {
49      int N;
50      cin >> N;
51      vector<string> s(N);
52      for (int i = 0; i < N; ++i) {
53          cin >> s[i];
54      }
55      int M;
56      cin >> M;
57      vector<vector<string>> grid(M, vector<string>(M));
58      for (int i = 0; i < M; ++i) {
59          for (int j = 0; j < M; ++j) {
60              cin >> grid[i][j];
61          }
62      }
63      // 构建大小和grid一样的检查数组checkList
64      // 用于避免出现重复检查的情况
65      vector<vector<bool>> checkList(M, vector<bool>(M, false));
66      // 双重遍历整个二维网格grid
67      for (int i = 0; i < M; ++i) {
68          for (int j = 0; j < M; ++j) {
69              // 找到点(i,j)等于s的第一个数字
70              // 则点(i,j)可以作为递归的起始位置
71              if (grid[i][j] == s[0]) {
72                  // 将点(i,j)在checkList中设置为已检查过
73                  checkList[i][j] = true;
74                  vector<vector<int>> path = {{i, j}};
75                  // 回溯函数递归入口，path初始为储存点(i,j)
76                  backtracking(grid, M, checkList, i, j, s, 0, path);
77                  // 将点(i,j)在checkList中重置为未检查过，因为本次回溯不一定找到答案
78                  checkList[i][j] = false;
79                  // 如果在回溯中，全局变量isFind被改为True，说明找到了明文，直接退出循环
80                  if (isFind) {
81                      break;
82                  }
83              }
```

```
        }
        // 关于i的循环同理，找到明文之后直接退出循环
        if (isFind) {
            break;
        }
    }
    // 如果最终没找到明文，输出error
    if (!isFind) {
        cout << "error" << endl;
    }
    return 0;
}

```

时空复杂度

时间复杂度：`O(M^2*3^N)`。其中`N`为密文`s`的长度，这是一个比较宽松的上界，回溯过程中每一个点都最多有三个分支可以进入。

空间复杂度：`O(M^2)`。`check_list`所占空间。