【华为OD机考 统一考试机试C卷】找单词（C++ Java JavaScript Python）

华为OD机考:统一考试 C卷 + D卷 + B卷 +A卷

2023年11月份，华为官方已经将 华为OD机考：OD统一考试（A卷 / B卷）切换到 OD统一考试（C卷）和 OD统一考试（D卷） 。根据考友反馈：目前抽到的试卷为B卷或C卷/D卷，其中C卷居多 ，按照之前的经验C卷D卷部分考题会复用A卷/B卷题，博主正积极从考过的同学收集C卷和D卷真题，可以查看下面的真题目录。

真题目录：华为OD机考机试 真题目录（C卷 + D卷 + B卷 + A卷） + 考点说明

专栏：2023华为OD机试(B卷+C卷+D卷)（C++JavaJSPy）

华为OD面试真题精选：华为OD面试真题精选

在线OJ：点击立即刷题，模拟真实机考环境 华为OD机考B卷C卷华为OD机考华为OD机考B卷华为OD机试B卷华为OD机试C卷华为OD机考C卷华为OD机考D卷题目华为OD机考C卷/D卷答案华为OD机考C卷/D卷解析华为OD机考C卷和D卷真题华为OD机考C卷和D卷题解

题目描述

给一个字符串和一个二维字符数组，如果该字符串存在于该数组中，则按字符串的字符顺序输出字符串每个字符所在单元格的位置下标字符串，如果找不到返回字符串“N”。

1.需要按照字符串的字符组成顺序搜索，且搜索到的位置必须是相邻单元格，其中“相邻单元格”是指那些水平相邻或垂直相邻的单元格。

2.同一个单元格内的字母不允许被重复使用。

3.假定在数组中最多只存在一个可能的匹配。

输入描述

第1行为一个数字N指示二维数组在后续输入所占的行数。

第2行到第N+1行输入为一个二维大写字符数组，每行字符用半角,分割。

第N+2行为待查找的字符串，由大写字符组成。

二维数组的大小为N*N，0<N<=100。

单词长度K，0<K<1000。

输出描述

输出一个位置下标字符串，拼接格式为：第1个字符行下标+","+第1个字符列下标+","+第2个字符行下标+","+第2个字符列下标… +","+第N个字符行下标+","+第N个字符列下标。

用例

输入	4 A,C,C,F C,D,E,D B,E,S,S F,E,C,A ACCESS
输出	0,0,0,1,0,2,1,2,2,2,2,3
说明	ACCESS分别对应二维数组的[0,0] [0,1] [0,2] [1,2] [2,2] [2,3]下标位置。

C++

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

using namespace std;

int n;  // 二维数组的大小
vector<vector<string>> matrix;  // 二维数组
string tar;  // 待查找的字符串
vector<vector<bool>> visited;  // 记录每个单元格是否已被访问

string findString();  // 查找字符串的函数声明
bool dfs(int i, int j, int k, vector<vector<int>>& path);  // 深度优先搜索的函数声明

int main() {
    cin >> n;  // 读取二维数组的大小
    cin.ignore();  // 忽略换行符
    matrix.resize(n, vector<string>(n));  // 初始化二维数组
    for (int i = 0; i < n; i++) {  // 读取二维数组的每一行
        string line;
        getline(cin, line);
        int pos = 0;
        for (int j = 0; j < n; j++) {  // 使用逗号分割每一行，得到每个单元格的字符
```

```
            int comma = line.find(',', pos);
            matrix[i][j] = line.substr(pos, comma - pos);
            pos = comma + 1;
        }
    }
    getline(cin, tar);  // 读取待查找的字符串

    visited.resize(n, vector<bool>(n));  // 初始化访问记录数组
    string result = findString();  // 查找字符串
    cout << result << endl;  // 输出结果

    return 0;
}

string findString() {
    vector<vector<int>> path;  // 存储路径的列表
    for (int i = 0; i < n; i++) {  // 遍历二维数组的每个单元格
        for (int j = 0; j < n; j++) {
            if (matrix[i][j] == tar.substr(0, 1)) {  // 如果当前单元格的字符与待查找字符串的第一个字符相同
                bool found = dfs(i, j, 0, path);  // 使用深度优先搜索查找字符串
                if (found) {  // 如果找到了字符串
                    string result;  // 初始化结果字符串
                    for (vector<int>& pos : path) {  // 将路径中的每个单元格的位置添加到结果字符串中
                        result += to_string(pos[0]) + "," + to_string(pos[1]) + ",";
                    }
                    result.pop_back();  // 删除最后一个逗号
                    return result;  // 返回结果字符串
                }
            }
        }
    }
    return "N";  // 如果没有找到字符串，返回"N"
}

bool dfs(int i, int j, int k, vector<vector<int>>& path) {
    // 如果当前位置越界，或已被访问，或当前位置的字符与待查找字符串的当前字符不相同
    if (i < 0 || i >= n || j < 0 || j >= n || visited[i][j] || tar.substr(k, 1) != matrix[i][j]) {
        return false;  // 返回false
    }
    path.push_back({i, j});  // 将当前位置添加到路径中

```

```
    visited[i][j] = true;  // 标记当前位置已被访问
    if (k == tar.length() - 1) {  // 如果已经找到了所有的字符
        return true;  // 返回true
    }
    vector<vector<int>> directions = {{0, 1}, {0, -1}, {1, 0}, {-1, 0}};  // 定义四个方向
    for (vector<int>& direction : directions) {  // 对四个方向进行深度优先搜索
        int ni = i + direction[0];
        int nj = j + direction[1];
        bool res = dfs(ni, nj, k + 1, path);
        if (res) {  // 如果在某个方向上找到了字符串
            return true;  // 返回true
        }
    }
    visited[i][j] = false;  // 撤销对当前位置的访问标记
    path.pop_back();  // 从路径中移除当前位置
    return false;  // 返回false
}
```

JavaScript

```
const readline = require('readline');

const rl = readline.createInterface({
  input: process.stdin,  // 输入流为标准输入
  output: process.stdout  // 输出流为标准输出
});

// 存储输入的每一行数据
const lines = [];
// 存储二维数组的大小
let n;

// 监听命令行输入的每一行数据
rl.on("line", (line) => {
  // 将每一行数据存入lines数组
  lines.push(line);
  // 如果输入的是二维数组的大小
  if (lines.length === 1) {
    n = parseInt(lines[0]);
  }
```

```
  // 如果输入的是二维数组和待查找的字符串
  if (n && lines.length === n + 2) {
    // 删除lines数组中的第一个元素，即二维数组的大小
    lines.shift();
    // 将lines数组中的最后一个元素，即待查找的字符串，存入变量str
    const str = lines.pop();
    // 将lines数组中的其余元素，即二维数组的每一行，按逗号分隔存入二维数组grid
    const grid = lines.map((line) => line.split(","));
    // 调用findString函数，输出结果
    console.log(findString(grid, n, str));
    // 清空lines数组
    lines.length = 0;
  }
});

// 查找字符串的函数
function findString(grid, n, tar) {
  // 创建一个n*n的二维数组，所有元素初始化为false，表示没有被访问过
  const visited = Array.from(Array(n), () => Array(n).fill(false));
  // 存储路径的数组
  const path = [];
  // 遍历二维数组的每个单元格
  for (let i = 0; i < n; i++) {
    for (let j = 0; j < n; j++) {
      // 如果当前单元格的字符与待查找字符串的第一个字符相同
      if (grid[i][j] === tar[0]) {
        // 使用深度优先搜索查找字符串
        const found = dfs(i, j, 0, path);
        // 如果找到了字符串
        if (found) {
          // 初始化结果字符串
          let result = "";
          // 将路径中的每个单元格的位置添加到结果字符串中
          for (const pos of path) {
            result += pos[0] + "," + pos[1] + ",";
          }
          // 删除最后一个逗号
          result = result.slice(0, -1);
          // 返回结果字符串
          return result;
```

```
      }
    }
  }
}
// 如果没有找到字符串，返回"N"
return "N";

// 深度优先搜索的函数
function dfs(i, j, k, path) {
  // 如果当前位置越界，或已被访问，或当前位置的字符与待查找字符串的当前字符不相同
  if (
    i < 0 ||
    i >= n ||
    j < 0 ||
    j >= n ||
    visited[i][j] ||
    tar[k] !== grid[i][j]
  ) {
    // 返回false
    return false;
  }
  // 将当前位置添加到路径中
  path.push([i, j]);
  // 标记当前位置已被访问
  visited[i][j] = true;
  // 如果已经找到了所有的字符
  if (k === tar.length - 1) {
    // 返回true
    return true;
  }
  // 定义四个方向
  const directions = [[0, 1], [0, -1], [1, 0], [-1, 0]];
  // 对四个方向进行深度优先搜索
  for (const direction of directions) {
    const ni = i + direction[0];
    const nj = j + direction[1];
    const res = dfs(ni, nj, k + 1, path);
    // 如果在某个方向上找到了字符串
    if (res) {
      // 返回true

```

```
            return true;
        }
    }
    // 撤销对当前位置的访问标记
    visited[i][j] = false;
    // 从路径中移除当前位置
    path.pop();
    // 返回false
    return false;
  }
}
```

Java

```
import java.util.LinkedList;
import java.util.Scanner;

public class Main {
    // 定义全局变量
    private static int n;  // 二维数组的大小
    private static String[][] matrix;  // 二维数组
    private static String tar;  // 待查找的字符串
    private static boolean[][] visited;  // 记录每个单元格是否已被访问

    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        n = scanner.nextInt();  // 读取二维数组的大小
        scanner.nextLine();  // 读取并忽略换行符
        matrix = new String[n][n];  // 初始化二维数组
        // 读取二维数组的每一行
        for (int i = 0; i < n; i++) {
            String line = scanner.nextLine();
            matrix[i] = line.split(",");  // 使用逗号分割每一行，得到每个单元格的字符
        }
        tar = scanner.nextLine();  // 读取待查找的字符串
        scanner.close();  // 关闭扫描器

        visited = new boolean[n][n];  // 初始化访问记录数组
        String result = findString();  // 查找字符串
        System.out.println(result);  // 输出结果
```

```
    }

    // 查找字符串的函数
    public static String findString() {
        LinkedList<Integer[]> path = new LinkedList<>();  // 存储路径的链表
        // 遍历二维数组的每个单元格
        for (int i = 0; i < n; i++) {
            for (int j = 0; j < n; j++) {
                // 如果当前单元格的字符与待查找字符串的第一个字符相同
                if (matrix[i][j].equals(tar.substring(0, 1))) {
                    // 使用深度优先搜索查找字符串
                    boolean found = dfs(i, j, 0, path);
                    // 如果找到了字符串
                    if (found) {
                        StringBuilder sb = new StringBuilder();
                        // 将路径中的每个单元格的位置添加到结果字符串中
                        for (Integer[] pos : path) {
                            sb.append(pos[0]).append(",").append(pos[1]).append(",");
                        }
                        sb.deleteCharAt(sb.length() - 1);  // 删除最后一个逗号
                        return sb.toString();  // 返回结果字符串
                    }
                }
            }
        }
        return "N";  // 如果没有找到字符串，返回"N"
    }

    // 深度优先搜索的函数
    public static boolean dfs(int i, int j, int k, LinkedList<Integer[]> path) {
        // 如果当前位置越界，或已被访问，或当前位置的字符与待查找字符串的当前字符不相同
        if (i < 0 || i >= n || j < 0 || j >= n || visited[i][j] || !tar.substring(k, k + 1).equals(matrix[i][j])) {
            return false;  // 返回false
        }
        path.add(new Integer[] {i, j});  // 将当前位置添加到路径中
        visited[i][j] = true;  // 标记当前位置已被访问
        // 如果已经找到了所有的字符
        if (k == tar.length() - 1) {
            return true;  // 返回true
        }

```

```
        // 定义四个方向
        int[][] directions = {{0, 1}, {0, -1}, {1, 0}, {-1, 0}};
        // 对四个方向进行深度优先搜索
        for (int[] direction : directions) {
            int ni = i + direction[0];
            int nj = j + direction[1];
            boolean res = dfs(ni, nj, k + 1, path);
            // 如果在某个方向上找到了字符串
            if (res) {
                return true;  // 返回true
            }
        }
        visited[i][j] = false;  // 撤销对当前位置的访问标记
        path.removeLast();  // 从路径中移除当前位置
        return false;  // 返回false
    }
}
```

Python

```
n = int(input())  # 读取二维数组的大小
matrix = []  # 初始化二维数组
for _ in range(n):  # 读取二维数组的每一行
    line = input()
    matrix.append(line.split(","))  # 使用逗号分割每一行，得到每个单元格的字符
tar = input()  # 读取待查找的字符串

visited = [[False] * n for _ in range(n)]  # 初始化访问记录数组

def findString():
    path = []  # 存储路径的列表
    for i in range(n):  # 遍历二维数组的每个单元格
        for j in range(n):
            if matrix[i][j] == tar[0]:  # 如果当前单元格的字符与待查找字符串的第一个字符相同
                found = dfs(i, j, 0, path)  # 使用深度优先搜索查找字符串
                if found:  # 如果找到了字符串
                    result = ""  # 初始化结果字符串
                    for pos in path:  # 将路径中的每个单元格的位置添加到结果字符串中
                        result += str(pos[0]) + "," + str(pos[1]) + ","
                    result = result[:-1]  # 删除最后一个逗号
```

```
                    return result  # 返回结果字符串
    return "N"  # 如果没有找到字符串，返回"N"

def dfs(i, j, k, path):
    # 如果当前位置越界，或已被访问，或当前位置的字符与待查找字符串的当前字符不相同
    if i < 0 or i >= n or j < 0 or j >= n or visited[i][j] or tar[k] != matrix[i][j]:
        return False  # 返回false
    path.append([i, j])  # 将当前位置添加到路径中
    visited[i][j] = True  # 标记当前位置已被访问
    if k == len(tar) - 1:  # 如果已经找到了所有的字符
        return True  # 返回true
    directions = [[0, 1], [0, -1], [1, 0], [-1, 0]]  # 定义四个方向
    for direction in directions:  # 对四个方向进行深度优先搜索
        ni = i + direction[0]
        nj = j + direction[1]
        res = dfs(ni, nj, k + 1, path)
        if res:  # 如果在某个方向上找到了字符串
            return True  # 返回true
    visited[i][j] = False  # 撤销对当前位置的访问标记
    path.pop()  # 从路径中移除当前位置
    return False  # 返回false

result = findString()  # 查找字符串
print(result)  # 输出结果
```

文章目录


机考真题
华为OD
No No No
CSDN @算法大师