【华为OD机考 统一考试机试C卷】考古学家考古问题（C++ Java JavaScript Python C语言）

华为OD机考:统一考试 C卷 + D卷 + B卷 +A卷

目前在考C卷，经过两个月的收集整理，C卷真题已基本整理完毕
抽到原题的概率为2/3到3/3，也就是最少抽到两道原题。请注意：大家刷完C卷真题，最好要把B卷的真题刷一下，因为C卷的部分真题来自B卷。
另外订阅专栏还可以联系笔者开通在线 OJ 进行刷题，提高刷题效率。
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

题目描述

有一个考古学家发现一个石碑，但是很可惜，发现时其已经断成多段，原地发现n个断口整齐的石碑碎片。为了破解石碑内容，考古学家希望有程序能帮忙计算复原后的石碑文字组合数，你能帮忙吗?

输入描述

第一行输入n，n表示石碑碎片的个数。

第二行依次输入石碑碎片上的文字内容s，共有n组。

输出描述

输出石碑文字的组合（按照升序排列），行末无多余空格。

用例

输入	3 a b c
输出	abc acb bac

	bca cab cba
说明	无

输入	3 a b a
输出	aab aba baa
说明	无

题目解析

全排列问题！！！

原题参考：47. 全排列 II - 力扣（LeetCode）

解体思路

解决这个问题的方法是使用深度优先搜索（DFS）遍历所有可能的组合。以下是详细的思路：

1. 首先，读取输入的石碑碎片个数 `n` 和石碑碎片上的文字内容 `s` 。
2. 将输入的石碑碎片内容存入一个列表 `charArray` ，并对其进行排序。排序的目的是为了在遍历过程中方便地跳过重复的组合。
3. 定义一个深度优先搜索函数 `dfs` ，其中包含以下参数：
    - `charArray` ：存储石碑碎片内容的列表。
    - `depth` ：当前搜索的深度。
    - `path` ：存储已经使用过的碎片。
    - `used` ：记录每个碎片是否被使用过。
    - `result` ：存储所有可能的组合。
4. 在 `dfs` 函数中，首先检查当前搜索的深度是否等于石碑碎片的个数。如果是，则将当前组合加入结果列表 `result` 。
5. 遍历 `charArray` 中的每个碎片。对于每个碎片，检查它是否已经被使用过，以及它是否与前一个碎片相同且前一个碎片未被使用。如果满足这些条件，则跳过当前碎片。

6. 将当前碎片添加到 `path` 中，并标记它为已使用。然后递归地搜索下一个碎片。
7. 在递归返回后，执行回溯操作：将当前碎片从 `path` 中移除，并标记它为未使用。
8. 调用 `dfs` 函数，开始搜索所有可能的组合。
9. 最后，输出所有找到的组合。

这种方法可以有效地遍历所有可能的石碑文字组合，并通过跳过重复的组合来减少搜索空间。

C++

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

using namespace std;

// 深度优先搜索函数
void dfs(vector<string>& charArray, int depth, deque<string>& path, vector<bool>& used, vector<vector<string>>& result);

int main() {
    // 处理输入
    int n;
    cin >> n;
    cin.ignore(); // 忽略第一行剩余的换行符
    string inputLine;
    getline(cin, inputLine);
    vector<string> charArray;
    string tempStr = "";
    for(int i = 0; i < inputLine.size(); i++) {
        if(inputLine[i] == ' ') {
            charArray.push_back(tempStr); // 将输入的碎片存入 charArray 中
            tempStr = "";
        }
        else {
            tempStr += inputLine[i];
        }
    }
```

```
    charArray.push_back(tempStr); // 存入最后一个碎片
    vector<vector<string>> result;

    // 先对碎片进行排序
    sort(charArray.begin(), charArray.end());
    // path 中存储已经使用过的碎片
    deque<string> path;
    // 记录每个碎片是否被使用过
    vector<bool> used(n, false);
    dfs(charArray, 0, path, used, result);

    // 输出所有组合
    for (int i = 0; i < result.size(); i++) {
        string outputStr = "";
        for(int j = 0; j < result[i].size(); j++) {
            outputStr += result[i][j];
        }
        cout << outputStr << endl;
    }

    return 0;
}

// 深度优先搜索函数
void dfs(vector<string>& charArray, int depth, deque<string>& path, vector<bool>& used, vector<vector<string>>& result) {
    // 如果碎片都已经被使用过，将当前组合加入结果中
    if (depth == charArray.size()) {
        result.push_back(vector<string>(path.begin(), path.end()));
        return;
    }
    for (int i = 0; i < charArray.size(); i++) {
        // 如果碎片已经被使用过，则跳过
        if (used[i]) {
            continue;
        }
        // 如果当前碎片和前一个碎片相同，并且前一个碎片还没有被使用，则跳过
        if (i > 0 && charArray[i] == charArray[i - 1] && !used[i - 1]) {
            continue;
        }
        path.push_back(charArray[i]); // 将当前碎片存入 path 中
```

```
        used[i] = true; // 标记当前碎片已被使用
        dfs(charArray, depth + 1, path, used, result); // 递归搜索下一个碎片
        path.pop_back(); // 回溯，将当前碎片从 path 中移除
        used[i] = false; // 标记当前碎片未被使用
    }
}
```

JavaScript

```
const readline = require('readline');
const rl = readline.createInterface({
    input: process.stdin,
    output: process.stdout
});

// 深度优先搜索函数
function dfs(charArray, depth, path, used, result) {
    // 如果碎片都已经被使用过，将当前组合加入结果中
    if (depth === charArray.length) {
        result.push([...path]);
        return;
    }
    for (let i = 0; i < charArray.length; i++) {
        // 如果碎片已经被使用过，则跳过
        if (used[i]) {
            continue;
        }
        // 如果当前碎片和前一个碎片相同，并且前一个碎片还没有被使用，则跳过
        if (i > 0 && charArray[i] === charArray[i - 1] && !used[i - 1]) {
            continue;
        }
        path.push(charArray[i]); // 将当前碎片存入 path 中
        used[i] = true; // 标记当前碎片已被使用
        dfs(charArray, depth + 1, path, used, result); // 递归搜索下一个碎片
        path.pop(); // 回溯，将当前碎片从 path 中移除
        used[i] = false; // 标记当前碎片未被使用
    }
}

rl.on('line', function (input) {
```

```
    let n = parseInt(input);
    rl.on('line', function (input) {
        let charArray = input.split(' ');
        let result = [];

        // 先对碎片进行排序
        charArray.sort();
        // path 中存储已经使用过的碎片
        let path = [];
        // 记录每个碎片是否被使用过
        let used = new Array(n).fill(false);
        dfs(charArray, 0, path, used, result);

        // 输出所有组合
        for (let i = 0; i < result.length; i++) {
            console.log(result[i].join(''));
        }
    });
});
```

Java

```
import java.util.*;

public class Main {
    // 深度优先搜索函数
    public static void dfs(String[] charArray, int depth, StringBuilder path, boolean[] used, List<String> result) {
        // 如果碎片都已经被使用过，将当前组合加入结果中
        if (depth == charArray.length) {
            result.add(path.toString());
            return;
        }
        for (int i = 0; i < charArray.length; i++) {
            // 如果碎片已经被使用过，则跳过
            if (used[i]) {
                continue;
            }
            // 如果当前碎片和前一个碎片相同，并且前一个碎片还没有被使用，则跳过
            if (i > 0 && charArray[i].equals(charArray[i - 1]) && !used[i - 1]) {
                continue;
```

```
            }
            path.append(charArray[i]); // 将当前碎片存入 path 中
            used[i] = true; // 标记当前碎片已被使用
            dfs(charArray, depth + 1, path, used, result); // 递归搜索下一个碎片
            path.setLength(path.length() - 1); // 回溯，将当前碎片从 path 中移除
            used[i] = false; // 标记当前碎片未被使用
        }
    }

    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        // 处理输入
        int n = scanner.nextInt();
        scanner.nextLine(); // 忽略第一行剩余的换行符
        String inputLine = scanner.nextLine();
        String[] charArray = inputLine.split(" ");
        List<String> result = new ArrayList<>();

        // 先对碎片进行排序
        Arrays.sort(charArray);
        // path 中存储已经使用过的碎片
        StringBuilder path = new StringBuilder();
        // 记录每个碎片是否被使用过
        boolean[] used = new boolean[n];
        dfs(charArray, 0, path, used, result);

        // 输出所有组合
        for (String s : result) {
            System.out.println(s);
        }
    }
}
```

Python

```
import sys

# 深度优先搜索函数
def dfs(charArray, depth, path, used, result):
```

```
    # 如果碎片都已经被使用过，将当前组合加入结果中
    if depth == len(charArray):
        result.append(list(path))
        return
    for i in range(len(charArray)):
        # 如果碎片已经被使用过，则跳过
        if used[i]:
            continue
        # 如果当前碎片和前一个碎片相同，并且前一个碎片还没有被使用，则跳过
        if i > 0 and charArray[i] == charArray[i - 1] and not used[i - 1]:
            continue
        path.append(charArray[i]) # 将当前碎片存入 path 中
        used[i] = True # 标记当前碎片已被使用
        dfs(charArray, depth + 1, path, used, result) # 递归搜索下一个碎片
        path.pop() # 回溯，将当前碎片从 path 中移除
        used[i] = False # 标记当前碎片未被使用

if __name__ == '__main__':
    # 处理输入
    n = int(input())
    inputLine = input()
    charArray = inputLine.split()
    result = []

    # 先对碎片进行排序
    charArray.sort()
    # path 中存储已经使用过的碎片
    path = []
    # 记录每个碎片是否被使用过
    used = [False] * n
    dfs(charArray, 0, path, used, result)

    # 输出所有组合
    for i in range(len(result)):
        outputStr = ''.join(result[i])
        print(outputStr)
```

C语言

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

// 定义石碑碎片的最大个数
#define MAX_N 100

// 比较函数，用于qsort排序
int compare(const void *a, const void *b) {
    return strcmp(*(const char**)a, *(const char**)b);
}

// 深度优先搜索函数
void dfs(char *charArray[], int n, int depth, char *path, int *pathIndex, int *used, char result[][MAX_N], int *resultIndex) {
    // 如果深度等于n，将当前path存入result
    if (depth == n) {
        path[*pathIndex] = '\0';
        strcpy(result[*resultIndex], path);
        (*resultIndex)++;
        return;
    }

    for (int i = 0; i < n; i++) {
        // 跳过已使用的碎片
        if (used[i]) {
            continue;
        }
        // 跳过重复的碎片
        if (i > 0 && strcmp(charArray[i], charArray[i - 1]) == 0 && !used[i - 1]) {
            continue;
        }
        // 添加当前碎片到path
        path[*pathIndex] = *charArray[i];
        (*pathIndex)++;
        used[i] = 1; // 标记为已使用
        dfs(charArray, n, depth + 1, path, pathIndex, used, result, resultIndex); // 递归
        (*pathIndex)--; // 回溯
        used[i] = 0; // 标记为未使用
    }
}
```

```c

int main() {
    int n;
    scanf("%d", &n); // 读取碎片个数
    char *charArray[MAX_N];
    char buffer[MAX_N];

    // 读取碎片
    for (int i = 0; i < n; i++) {
        charArray[i] = (char*)malloc(MAX_N * sizeof(char));
        scanf("%s", charArray[i]);
    }

    // 对碎片排序
    qsort(charArray, n, sizeof(char*), compare);

    // 结果数组，path数组，和辅助数组
    char result[MAX_N][MAX_N];
    char path[MAX_N];
    int pathIndex = 0;
    int used[MAX_N] = {0};
    int resultIndex = 0;

    // 执行深度优先搜索
    dfs(charArray, n, 0, path, &pathIndex, used, result, &resultIndex);

    // 输出所有组合
    for (int i = 0; i < resultIndex; i++) {
        printf("%s\n", result[i]);
    }

    // 释放内存
    for (int i = 0; i < n; i++) {
        free(charArray[i]);
    }

    return 0;
}
```

文章目录