【华为OD机考 统一考试机试C卷】最多几个直角三角形（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N条线段，长度分别为a[1]-a[n]。

现要求你计算这N条线段最多可以组合成几个直角三角形。

每条线段只能使用一次，每个三角形包含三条线段。

输入描述

第一行输入一个正整数T（1<=T<=100），表示有T组测试数据.

对于每组测试数据，接下来有T行，

每行第一个正整数N，表示线段个数（3<=N<=20），接着是N个正整数，表示每条线段长度，（0<a[i]<100）。

输出描述

对于每组测试数据输出一行，每行包括一个整数，表示最多能组合的直角三角形个数

用例

输入

```
1
7 3 4 5 6 5 12 13
```

输出

```
2
```

说明

> 可以组成2个直角三角形（3，4，5）、（5，12，13）

C++ 递归

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

// dfs函数用于查找最多可以组成的直角三角形数量
int dfs(const std::vector<int>& arr, std::vector<bool>& used, int index, int count) {
    // 初始化答案为当前已找到的直角三角形数量
    int ans = count;

    // 遍历线段数组，从index开始
    for (int i = index; i < arr.size(); i++) {
        // 如果当前线段已被使用，跳过
        if (used[i]) continue;
        // 遍历后续线段，从i+1开始
        for (int j = i + 1; j < arr.size(); j++) {
            // 如果当前线段已被使用，跳过
            if (used[j]) continue;
            // 遍历后续线段，从j+1开始
            for (int k = j + 1; k < arr.size(); k++) {
                // 如果当前线段已被使用，跳过
                if (used[k]) continue;

                // 检查当前三条线段是否满足勾股定理
                if (arr[i] * arr[i] + arr[j] * arr[j] == arr[k] * arr[k]) {

```

```
                    // 标记当前线段为已使用
                    used[i] = true;
                    used[j] = true;
                    used[k] = true;

                    // 递归调用dfs函数，并更新答案
                    ans = std::max(ans, dfs(arr, used, i + 1, count + 1));

                    // 回溯，将当前线段标记为未使用
                    used[i] = false;
                    used[j] = false;
                    used[k] = false;
                }
            }
        }
    }

    // 返回答案
    return ans;
}

int main() {
    // 读取测试数据组数
    int t;
    std::cin >> t;
    // 创建一个二维数组来存储每组测试数据
    std::vector<std::vector<int>> cases(t);

    // 读取每组测试数据
    for (int i = 0; i < t; i++) {
        // 读取线段数量
        int n;
        std::cin >> n;
        // 创建一个数组来存储线段长度
        std::vector<int> arr(n);
        // 读取每条线段的长度
        for (int j = 0; j < n; j++) {
            std::cin >> arr[j];
        }
        // 将线段长度数组存入cases数组

```

```
        cases[i] = arr;
    }

    // 遍历每组测试数据
    for (const auto& arr : cases) {
        // 对线段长度进行升序排序
        std::vector<int> sorted_arr = arr;
        std::sort(sorted_arr.begin(), sorted_arr.end());
        // 创建一个布尔数组表示线段是否已使用
        std::vector<bool> used(sorted_arr.size(), false);
        // 调用dfs函数并输出结果
        std::cout << dfs(sorted_arr, used, 0, 0) << std::endl;
    }

    return 0;
}
```

C++ - 栈

```
#include <iostream>
#include <vector>
#include <algorithm>
#include <stack>

// State结构体用于存储每个状态的信息，包括当前索引、已找到的直角三角形数量和已使用的线段
struct State {
    int index;
    int count;
    std::vector<bool> used;

    State(int index, int count, const std::vector<bool>& used)
        : index(index), count(count), used(used) {}
};

int non_recursive_dfs(const std::vector<int>& lengths);

int main() {
    // 读取测试数据组数
```

```
    int test_cases;
    std::cin >> test_cases;

    // 创建一个二维数组来存储每组测试数据
    std::vector<std::vector<int>> input_data(test_cases);

    // 读取每组测试数据
    for (int i = 0; i < test_cases; ++i) {
        // 读取线段数量
        int segments;
        std::cin >> segments;

        // 创建一个数组来存储线段长度
        std::vector<int> lengths(segments);

        // 读取每条线段的长度
        for (int j = 0; j < segments; ++j) {
            std::cin >> lengths[j];
        }

        // 将线段长度数组存入input_data数组
        input_data[i] = lengths;
    }

    for (auto& lengths : input_data) {
        // 对线段长度进行升序排序
        std::sort(lengths.begin(), lengths.end());

        // 调用non_recursive_dfs函数并输出结果
        std::cout << non_recursive_dfs(lengths) << std::endl;
    }

    return 0;
}

int non_recursive_dfs(const std::vector<int>& lengths) {
    int max_triangles = 0;
    std::vector<bool> used_segments(lengths.size(), false);
    std::stack<State> stack;
    stack.push(State(0, 0, used_segments));
```

```
    // 遍历状态栈
    while (!stack.empty()) {
        State current_state = stack.top();
        stack.pop();
        int current_index = current_state.index;
        int current_count = current_state.count;
        used_segments = current_state.used;

        max_triangles = std::max(max_triangles, current_count);

        // 遍历线段数组，从current_index开始
        for (int i = current_index; i < lengths.size(); ++i) {
            if (used_segments[i]) continue;
            for (int j = i + 1; j < lengths.size(); ++j) {
                if (used_segments[j]) continue;
                for (int k = j + 1; k < lengths.size(); ++k) {
                    if (used_segments[k]) continue;

                    // 检查当前三条线段是否满足勾股定理
                    if (lengths[i] * lengths[i] + lengths[j] * lengths[j] == lengths[k] * lengths[k]) {
                        std::vector<bool> new_used_segments = used_segments;
                        new_used_segments[i] = true;
                        new_used_segments[j] = true;
                        new_used_segments[k] = true;
                        stack.push(State(i + 1, current_count + 1, new_used_segments));
                    }
                }
            }
        }
    }

    return max_triangles;
}
```

javascript 递归

```
const readline = require('readline');
const rl = readline.createInterface({
```

```
    input: process.stdin,
    output: process.stdout
});
// 查找最多可以组成的直角三角形数量
function dfs(arr, used, index, count) {
    let ans = count;

    for (let i = index; i < arr.length; i++) {
        if (used[i]) {
            continue;
        }
        for (let j = i + 1; j < arr.length; j++) {
            if (used[j]) {
                continue;
            }
            for (let k = j + 1; k < arr.length; k++) {
                if (used[k]) {
                    continue;
                }

                if (arr[i] ** 2 + arr[j] ** 2 === arr[k] ** 2) {
                    used[i] = true;
                    used[j] = true;
                    used[k] = true;

                    ans = Math.max(ans, dfs(arr, used, i + 1, count + 1));

                    used[i] = false;
                    used[j] = false;
                    used[k] = false;
                }
            }
        }
    }

    return ans;
}
// 读取测试数据组数
rl.on('line', (test_cases) => {
    test_cases = parseInt(test_cases);
```

```
    const input_data = [];
    let read_count = 0;

    // 读取每组测试数据
    rl.on('line', (line) => {
        input_data.push(line.split(' ').slice(1).map(Number));
        read_count++;

        if (read_count === test_cases) {
            rl.close();

            for (const testCase of input_data) {
                // 对线段长度进行升序排序
                testCase.sort((a, b) => a - b);

                // 调用non_recursive_dfs函数并输出结果
                console.log(dfs(testCase, Array(testCase.length).fill(false), 0, 0));
            }
        }
    });
});
```

JavaScript - 栈

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

// State类用于存储每个状态的信息，包括当前索引、已找到的直角三角形数量和已使用的线段
class State {
    constructor(index, count, used) {
        this.index = index;
        this.count = count;
        this.used = used;
    }
}

```

```
function combinations(arr, k) {
    if (k > arr.length || k <= 0) {
        return [];
    }

    if (k === arr.length) {
        return [arr];
    }

    if (k === 1) {
        return arr.map((value) => [value]);
    }

    const result = [];
    for (let i = 0; i < arr.length - k + 1; i++) {
        const head = arr.slice(i, i + 1);
        const tailcombs = combinations(arr.slice(i + 1), k - 1);
        for (let j = 0; j < tailcombs.length; j++) {
            result.push(head.concat(tailcombs[j]));
        }
    }
    return result;
}

function non_recursive_dfs(lengths) {
    let max_triangles = 0;
    const used_segments = Array(lengths.length).fill(false);
    const stack = [new State(0, 0, used_segments)];

    // 遍历状态栈
    while (stack.length > 0) {
        const current_state = stack.pop();
        const current_index = current_state.index;
        const current_count = current_state.count;
        const used_segments = current_state.used;

        max_triangles = Math.max(max_triangles, current_count);

        // 遍历线段数组，从current_index开始
        for (const [i, j, k] of combinations([...Array(lengths.length - current_index).keys()].map(x => x + current_index), 3)) {
```

```
            if (used_segments[i] || used_segments[j] || used_segments[k]) {
                continue;
            }

            // 检查当前三条线段是否满足勾股定理
            if (lengths[i] ** 2 + lengths[j] ** 2 === lengths[k] ** 2) {
                const new_used_segments = used_segments.slice();
                new_used_segments[i] = true;
                new_used_segments[j] = true;
                new_used_segments[k] = true;
                stack.push(new State(i + 1, current_count + 1, new_used_segments));
            }
        }
    }

    return max_triangles;
}

// 读取测试数据组数
rl.on('line', (test_cases) => {
    test_cases = parseInt(test_cases);
    const input_data = [];
    let read_count = 0;

    // 读取每组测试数据
    rl.on('line', (line) => {
        input_data.push(line.split(' ').slice(1).map(Number));
        read_count++;

        if (read_count === test_cases) {
            rl.close();

            for (const testCase of input_data) {
                // 对线段长度进行升序排序
                testCase.sort((a, b) => a - b);

                // 调用non_recursive_dfs函数并输出结果
                console.log(non_recursive_dfs(testCase));
            }
        }

```

```
    });
});
```

java-栈解法

```java
import java.util.Arrays;
import java.util.Scanner;
import java.util.Stack;

public class Main {
    public static void main(String[] args) {
        // 创建Scanner对象以读取用户输入
        Scanner scanner = new Scanner(System.in);

        // 读取测试数据组数
        int testCases = scanner.nextInt();
        // 创建一个二维数组来存储每组测试数据
        int[][] inputData = new int[testCases][];

        // 读取每组测试数据
        for (int i = 0; i < testCases; i++) {
            // 读取线段数量
            int segments = scanner.nextInt();
            // 创建一个数组来存储线段长度
            int[] lengths = new int[segments];
            // 读取每条线段的长度
            for (int j = 0; j < segments; j++) {
                lengths[j] = scanner.nextInt();
            }
            // 将线段长度数组存入inputData数组
            inputData[i] = lengths;
        }

        // 遍历每组测试数据
        for (int[] lengths : inputData) {
            // 对线段长度进行升序排序
            Arrays.sort(lengths);
            // 调用nonRecursiveDFS函数并输出结果
```

```
        System.out.println(nonRecursiveDFS(lengths));
    }
}

public static int nonRecursiveDFS(int[] lengths) {
    int maxTriangles = 0;
    boolean[] usedSegments = new boolean[lengths.length];
    Stack<State> stack = new Stack<>();
    stack.push(new State(0, 0, usedSegments));

    // 遍历状态栈
    while (!stack.isEmpty()) {
        State currentState = stack.pop();
        int currentIndex = currentState.index;
        int currentCount = currentState.count;
        usedSegments = currentState.used.clone();

        maxTriangles = Math.max(maxTriangles, currentCount);

        // 遍历线段数组，从currentIndex开始
        for (int i = currentIndex; i < lengths.length; i++) {
            if (usedSegments[i]) continue;
            for (int j = i + 1; j < lengths.length; j++) {
                if (usedSegments[j]) continue;
                for (int k = j + 1; k < lengths.length; k++) {
                    if (usedSegments[k]) continue;

                    // 检查当前三条线段是否满足勾股定理
                    if (lengths[i] * lengths[i] + lengths[j] * lengths[j] == lengths[k] * lengths[k]) {
                        boolean[] newUsedSegments = usedSegments.clone();
                        newUsedSegments[i] = true;
                        newUsedSegments[j] = true;
                        newUsedSegments[k] = true;
                        stack.push(new State(i + 1, currentCount + 1, newUsedSegments));
                    }
                }
            }
        }
    }
```

```
        return maxTriangles;
    }

    // State类用于存储每个状态的信息，包括当前索引、已找到的直角三角形数量和已使用的线段
    static class State {
        int index;
        int count;
        boolean[] used;

        State(int index, int count, boolean[] used) {
            this.index = index;
            this.count = count;
            this.used = used;
        }
    }
}
```

Java - 递归解法

```
import java.util.Arrays;
import java.util.Scanner;

public class Main {
    public static void main(String[] args) {
        // 创建Scanner对象以读取用户输入
        Scanner sc = new Scanner(System.in);

        // 读取测试数据组数
        int t = sc.nextInt();
        // 创建一个二维数组来存储每组测试数据
        int[][] cases = new int[t][];

        // 读取每组测试数据
        for (int i = 0; i < t; i++) {
            // 读取线段数量
            int n = sc.nextInt();
            // 创建一个数组来存储线段长度
            int[] arr = new int[n];
            // 读取每条线段的长度
```

```
            for (int j = 0; j < n; j++) {
                arr[j] = sc.nextInt();
            }
            // 将线段长度数组存入cases数组
            cases[i] = arr;
        }

        // 遍历每组测试数据
        for (int[] arr : cases) {
            // 对线段长度进行升序排序
            Arrays.sort(arr);
            // 调用dfs函数并输出结果
            System.out.println(dfs(arr, new boolean[arr.length], 0, 0));
        }
    }

    // dfs函数用于查找最多可以组成的直角三角形数量
    public static int dfs(int[] arr, boolean[] used, int index, int count) {
        // 初始化答案为当前已找到的直角三角形数量
        int ans = count;

        // 遍历线段数组，从index开始
        for (int i = index; i < arr.length; i++) {
            // 如果当前线段已被使用，跳过
            if (used[i]) continue;
            // 遍历后续线段，从i+1开始
            for (int j = i + 1; j < arr.length; j++) {
                // 如果当前线段已被使用，跳过
                if (used[j]) continue;
                // 遍历后续线段，从j+1开始
                for (int k = j + 1; k < arr.length; k++) {
                    // 如果当前线段已被使用，跳过
                    if (used[k]) continue;

                    // 检查当前三条线段是否满足勾股定理
                    if (arr[i] * arr[i] + arr[j] * arr[j] == arr[k] * arr[k]) {
                        // 标记当前线段为已使用
                        used[i] = true;
                        used[j] = true;
                        used[k] = true;
```

```
                        // 递归调用dfs函数，并更新答案
                        ans = Math.max(ans, dfs(arr, used, i + 1, count + 1));

                        // 回溯，将当前线段标记为未使用
                        used[i] = false;
                        used[j] = false;
                        used[k] = false;
                    }
                }
            }
        }

        // 返回答案
        return ans;
    }
}
```

Python - 栈

```
from itertools import combinations

# State类用于存储每个状态的信息，包括当前索引、已找到的直角三角形数量和已使用的线段
class State:
    def __init__(self, index, count, used):
        self.index = index
        self.count = count
        self.used = used

def non_recursive_dfs(lengths):
    max_triangles = 0
    used_segments = [False] * len(lengths)
    stack = [State(0, 0, used_segments)]

    # 遍历状态栈
    while stack:
        current_state = stack.pop()
        current_index = current_state.index
        current_count = current_state.count
```

```
        used_segments = current_state.used

        max_triangles = max(max_triangles, current_count)

        # 遍历线段数组，从current_index开始
        for i, j, k in combinations(range(current_index, len(lengths)), 3):
            if used_segments[i] or used_segments[j] or used_segments[k]:
                continue

            # 检查当前三条线段是否满足勾股定理
            if lengths[i] ** 2 + lengths[j] ** 2 == lengths[k] ** 2:
                new_used_segments = used_segments.copy()
                new_used_segments[i] = True
                new_used_segments[j] = True
                new_used_segments[k] = True
                stack.append(State(i + 1, current_count + 1, new_used_segments))

    return max_triangles

# 读取测试数据组数
test_cases = int(input())

# 创建一个二维数组来存储每组测试数据
input_data = []

# 读取每组测试数据
for _ in range(test_cases):

     input_data =  [list(map(int, input().split()))[1:] for i in range(test_cases)]

for testCase in input_data:
    # 对线段长度进行升序排序
    testCase.sort()
    # 调用non_recursive_dfs函数并输出结果
    print(non_recursive_dfs(testCase))
```

python 递归

```python

import sys
from typing import List, Tuple

# 查找最多可以组成的直角三角形数量
def dfs(arr: List[int], used: List[bool], index: int, count: int) -> int:
    ans = count

    for i in range(index, len(arr)):
        if used[i]:
            continue
        for j in range(i + 1, len(arr)):
            if used[j]:
                continue
            for k in range(j + 1, len(arr)):
                if used[k]:
                    continue

                if arr[i] ** 2 + arr[j] ** 2 == arr[k] ** 2:
                    used[i] = True
                    used[j] = True
                    used[k] = True

                    ans = max(ans, dfs(arr, used, i + 1, count + 1))

                    used[i] = False
                    used[j] = False
                    used[k] = False

    return ans
# 读取测试数据组数
test_cases = int(input())

# 创建一个二维数组来存储每组测试数据
input_data = []

# 读取每组测试数据
for _ in range(test_cases):

     input_data =  [list(map(int, input().split()))[1:] for i in range(test_cases)]
```

```
for testCase in input_data:
    # 对线段长度进行升序排序
    testCase.sort()
    # 调用non_recursive_dfs函数并输出结果
    print(dfs(testCase,  [False] * len(testCase), 0, 0))
```

C语言 - 递归

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

#define MAX_T 100
#define MAX_N 20

// dfs函数用于查找最多可以组成的直角三角形数量
int dfs(int arr[], bool used[], int size, int index, int count) {
    int ans = count; // 初始化答案为当前已找到的直角三角形数量

    // 遍历线段数组，从index开始
    for (int i = index; i < size; i++) {
        // 如果当前线段已被使用，跳过
        if (used[i]) continue;
        // 遍历后续线段，从i+1开始
        for (int j = i + 1; j < size; j++) {
            // 如果当前线段已被使用，跳过
            if (used[j]) continue;
            // 遍历后续线段，从j+1开始
            for (int k = j + 1; k < size; k++) {
                // 如果当前线段已被使用，跳过
                if (used[k]) continue;

                // 检查当前三条线段是否满足勾股定理
                if (arr[i] * arr[i] + arr[j] * arr[j] == arr[k] * arr[k] ||
                    arr[i] * arr[i] + arr[k] * arr[k] == arr[j] * arr[j] ||
                    arr[j] * arr[j] + arr[k] * arr[k] == arr[i] * arr[i]) {
                    // 标记当前线段为已使用
                    used[i] = true;
```

```
                    used[j] = true;
                    used[k] = true;

                    // 递归调用dfs函数，并更新答案
                    int newCount = dfs(arr, used, size, i + 1, count + 1);
                    if (newCount > ans) {
                        ans = newCount;
                    }

                    // 回溯，将当前线段标记为未使用
                    used[i] = false;
                    used[j] = false;
                    used[k] = false;
                }
            }
        }
    }

    // 返回答案
    return ans;
}

// 用于比较两个整数的函数，用于qsort
int compare(const void *a, const void *b) {
    return (*(int *)a - *(int *)b);
}

int main() {
    int t;
    scanf("%d", &t); // 读取测试数据组数

    while (t--) {
        int n;
        scanf("%d", &n); // 读取线段数量
        int arr[MAX_N];
        bool used[MAX_N] = {false}; // 创建一个布尔数组表示线段是否已使用

        for (int i = 0; i < n; i++) {
            scanf("%d", &arr[i]); // 读取每条线段的长度
        }
```

```
        // 对线段长度进行升序排序
        qsort(arr, n, sizeof(int), compare);

        // 调用dfs函数并输出结果
        printf("%d\n", dfs(arr, used, n, 0, 0));
    }

    return 0;
}
```

C语言 - 栈

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

#define MAX_N 20

// State结构体用于存储每个状态的信息，包括当前索引、已找到的直角三角形数量和已使用的线段
typedef struct State {
    int index;
    int count;
    bool used[MAX_N];
} State;

// 用于比较两个整数的函数，用于qsort
int compare(const void *a, const void *b) {
    return (*(int *)a - *(int *)b);
}

// non_recursive_dfs函数用于非递归地进行深度优先搜索
int non_recursive_dfs(int lengths[], int size) {
    int max_triangles = 0;
    bool used_segments[MAX_N] = {false};
    State stack[MAX_N * MAX_N * MAX_N]; // 堆栈大小取决于可能的状态数
    int top = -1; // 栈顶初始化为-1，表示栈为空

    // 初始状态入栈
    stack[++top] = (State){0, 0, {false}};
```

```
    // 遍历状态栈
    while (top != -1) {
        State current_state = stack[top--]; // 弹出栈顶元素
        int current_index = current_state.index;
        int current_count = current_state.count;

        for (int i = 0; i < size; ++i) {
            used_segments[i] = current_state.used[i];
        }

        if (current_count > max_triangles) {
            max_triangles = current_count;
        }

        // 遍历线段数组，从current_index开始
        for (int i = current_index; i < size; ++i) {
            if (used_segments[i]) continue;
            for (int j = i + 1; j < size; ++j) {
                if (used_segments[j]) continue;
                for (int k = j + 1; k < size; ++k) {
                    if (used_segments[k]) continue;

                    // 检查当前三条线段是否满足勾股定理
                    if (lengths[i] * lengths[i] + lengths[j] * lengths[j] == lengths[k] * lengths[k] ||
                        lengths[i] * lengths[i] + lengths[k] * lengths[k] == lengths[j] * lengths[j] ||
                        lengths[j] * lengths[j] + lengths[k] * lengths[k] == lengths[i] * lengths[i]) {
                        used_segments[i] = true;
                        used_segments[j] = true;
                        used_segments[k] = true;
                        stack[++top] = (State){i + 1, current_count + 1, {0}};
                        for (int l = 0; l < size; ++l) {
                            stack[top].used[l] = used_segments[l];
                        }
                        used_segments[i] = false;
                        used_segments[j] = false;
                        used_segments[k] = false;
                    }
                }
            }
```

```
        }
    }

    return max_triangles;
}

int main() {
    int test_cases;
    scanf("%d", &test_cases); // 读取测试数据组数

    while (test_cases--) {
        int segments;
        scanf("%d", &segments); // 读取线段数量
        int lengths[MAX_N];

        for (int i = 0; i < segments; ++i) {
            scanf("%d", &lengths[i]); // 读取每条线段的长度
        }

        // 对线段长度进行升序排序
        qsort(lengths, segments, sizeof(int), compare);

        // 调用non_recursive_dfs函数并输出结果
        printf("%d\n", non_recursive_dfs(lengths, segments));
    }

    return 0;
}
```

文章目录

javascript 递归
JavaScript - 栈
java-栈解法
Java - 递归解法
Python - 栈
python 递归
C语言 - 递归
C语言 - 栈