

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

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

输入 m 和 n 两个数, m 和 n 表示一个 $m*n$ 的棋盘。输入棋盘内的数据。棋盘中存在数字和 "." 两种字符, 如果是数字表示该位置是一匹马, 如果是 "." 表示该位置为空的, 棋盘内的数字表示为该马能走的最大步数。

例如棋盘内某个位置一个数字为 k , 表示该马只能移动 $1\sim k$ 步的距离。

棋盘内的马移动类似于中国象棋中的马移动, 先在水平或者垂直方向上移动一格, 然后再将其移动到对角线位置。

棋盘内的马可以移动到同一个位置, 同一个位置可以有多匹马。

请问能否将棋盘上所有的马移动到同一个位置, 若可以请输入移动的最小步数。若不可以输出 0。

输入描述

输入 m 和 n 两个数, m 和 n 表示一个 $m*n$ 的棋盘。输入棋盘内的数据。

输出描述

能否将棋盘上所有的马移动到同一个位置, 若可以请输入移动的最小步数。若不可以输出 0。

用例1

输入

1	3 2
2	. .
3	2 .
4	. .

输出

1	0
---	---

用例二

输入

1	3 5
2	4 7 . 4 8
3	4 7 4 4 .
4	7

输出

1	17
---	----

给定的用例是一个3行5列的棋盘，其中一些位置有数字，代表马的位置和它们可以走的最大步数。我们将逐步模拟广度优先搜索（BFS）的过程来找到所有马都能到达的位置，并计算出最小步数。

棋盘布局：

1	4 7 . 4 8
2	4 7 4 4 .
3	7

模拟计算

步骤：

1. 初始化:

- 马的位置和最大步数分别为: (0,0,4), (0,1,7), (0,3,4), (0,4,8), (1,0,4), (1,1,7), (1,2,4), (1,3,4), (2,0,7)。

2. 对棋盘上的每个位置进行BFS:

- 我们需要检查棋盘上的每个位置, 看看是否所有马都能到达那里。例如, 我们检查位置(0,2)。

3. 对每个马进行BFS:

- 从马(0,0,4)开始, 它可以在4步内到达的位置有限。我们将这些位置和步数记录下来, 并检查是否包括(0,2)。
- 接下来, 我们对马(0,1,7)执行同样的操作, 记录它可以到达的位置和步数。
- 我们重复这个过程, 直到考虑了所有的马。

4. 累加步数:

- 如果所有马都可以在它们的最大步数内到达位置(0,2), 我们将这些步数累加起来。

5. 更新最小步数:

- 如果我们发现所有马都能到达位置(0,2), 我们将这个累加的步数与当前的最小步数进行比较, 并更新最小步数。
- 如果有任何一个马不能到达位置(0,2), 我们将忽略这个位置, 并继续检查下一个位置。

6. 重复以上步骤:

- 我们重复以上步骤, 对棋盘上的每个位置进行检查。

7. 得出结果:

- 在检查完所有位置后, 我们得到所有马都能到达的位置的最小步数。
- 如果没有这样的位置, 则返回-1。

解题思路

1. 初始化数据结构:

- 读取棋盘的行数和列数。

- 创建一个二维数组来表示棋盘。
- 创建一个列表来存储每个马的位置和它们可以走的最大步数。

2. 广度优先搜索 (BFS) :

- 定义一个函数来执行BFS, 该函数将遍历棋盘上的每个位置, 尝试找到所有马都能到达的位置, 并计算出最小步数。
- 在BFS中, 定义马可以走的八个方向。
- 对于棋盘上的每个位置, 初始化步数为0, 并设置一个标志来判断是否所有马都能到达该位置。

3. 遍历棋盘上的每个位置:

- 对于棋盘上的每个位置, 遍历每个马, 使用BFS来判断马是否能到达该位置。
- 对于每个马, 使用一个队列来存储它可以到达的位置和对应的步数。
- 使用一个集合来记录已经访问过的位置, 避免重复访问。

4. 遍历每个马:

- 从马的当前位置开始, 将其加入队列, 并将该位置标记为已访问。
- 当队列不为空时, 取出队列的头部元素, 这是当前马的位置和步数。
- 如果该位置是目标位置, 累加步数并标记找到目标位置。
- 否则, 遍历马可以走的八个方向, 对于每个方向, 计算新的位置并检查是否有效且未访问过。
- 如果新位置有效, 将其加入队列并标记为已访问。

5. 更新步数和可能性:

- 如果找到目标位置, 累加步数。
- 如果没有找到目标位置, 标记为不可能到达。

6. 计算最小步数:

- 如果所有马都能到达当前位置, 更新最小步数。
- 如果没有任何位置能被所有马到达, 返回-1。
- 否则, 返回找到的最小步数。

C++

```
1  #include <iostream>
2  #include <vector>
3  #include <queue>
4  #include <set>
5  #include <string>
6  #include <climits>
7  #include <tuple>
8
9  using namespace std;
10
11 // 定义棋盘的行数和列数
12 int m, n;
13 // 定义棋盘
14 vector<vector<char>> board;
15 // 定义马的位置和步数的列表
16 vector<tuple<int, int, int>> horses;
17
18 // 定义广度优先搜索方法
19 int bfs() {
20     // 定义马能走的八个方向
21     vector<pair<int, int>> directions = {{-1, -2}, {-2, -1}, {-2, 1}, {-1, 2}, {1, 2}, {2, 1}, {2, -1}, {1, -2}};
22     // 初始化最小步数为最大值
23     int minSteps = INT_MAX;
24
25     // 遍历棋盘上的每个位置
26     for (int i = 0; i < m; ++i) {
27         for (int j = 0; j < n; ++j) {
28             // 初始化当前位置的步数为0
29             int steps = 0;
30             // 标记是否所有马都能到达当前位置
31             bool possible = true;
32
33             // 遍历每个马
34             for (auto& horse : horses) {
35                 // 使用队列进行BFS
36                 queue<tuple<int, int, int>> queue;
37                 // 使用集合记录已访问的位置
38                 set<string> visited;
```

```
// 获取马的位置和最大步数
int x, y, maxSteps;
tie(x, y, maxSteps) = horse;
// 将当前马的位置和步数加入队列
queue.push(make_tuple(x, y, 0));
// 将当前马的位置添加到已访问集合中
visited.insert(to_string(x) + "," + to_string(y));
// 标记是否找到当前位置
bool found = false;

// 当队列不为空且可能到达当前位置时
while (!queue.empty() && possible) {
    // 取出队列头部元素
    tuple<int, int, int> current = queue.front();
    queue.pop();
    int cx, cy, cs;
    tie(cx, cy, cs) = current; // Unpack the tuple
    // 如果当前元素位置等于目标位置
    if (cx == i && cy == j) {
        // 累加步数
        steps += cs;
        // 标记为找到
        found = true;
        break;
    }

    // 遍历马能走的八个方向
    for (auto& dir : directions) {
        // 计算新的位置
        int nx = cx + dir.first;
        int ny = cy + dir.second;
        // 如果新位置有效且未访问过, 则加入队列
        if (nx >= 0 && nx < m && ny >= 0 && ny < n && cs < maxSteps && !visited.count(to_string(nx) + "," + to_string(ny))) {
            queue.push(make_tuple(nx, ny, cs + 1));
            visited.insert(to_string(nx) + "," + to_string(ny));
        }
    }
}

// 如果没有找到目标位置, 则标记为不可能到达
```

```
81         if (!found) {
82             possible = false;
83         }
84     }
85
86     // 如果所有马都能到达当前位置, 则更新最小步数
87     if (possible) {
88         minSteps = min(minSteps, steps);
89     }
90 }
91
92 // 如果最小步数为最大值, 则返回-1, 否则返回最小步数
93 return minSteps == INT_MAX ? -1 : minSteps;
94 }
95
96 // 主函数
97 int main() {
98     // 读取棋盘的行数和列数
99     cin >> m >> n;
100     // 初始化棋盘
101     board.resize(m, vector<char>(n));
102     // 读取棋盘上每个位置的输入
103     for (int i = 0; i < m; ++i) {
104         for (int j = 0; j < n; ++j) {
105             cin >> board[i][j];
106             // 如果当前位置不是空点, 则将马的位置和步数添加到列表中
107             if (board[i][j] != '.') {
108                 horses.emplace_back(i, j, board[i][j] - '0');
109             }
110         }
111     }
112
113     // 调用bfs方法并打印结果
114     cout << bfs() << endl;
115     return 0;
116 }
```

Java


```
1 import java.io.BufferedReader;
2 import java.io.IOException;
3 import java.io.InputStreamReader;
4 import java.util.LinkedList;
5 import java.util.Queue;
6 import java.util.HashSet;
7 import java.util.Set;
8
9 public class Main {
10     // 定义棋盘的行数和列数
11     private static int m, n;
12     // 定义棋盘
13     private static int[][] board;
14     // 定义马的位置和步数的列表
15     private static LinkedList<int[]> horses = new LinkedList<>();
16
17     public static void main(String[] args) throws IOException {
18         // 使用BufferedReader读取输入
19         BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
20         // 读取第一行输入, 获取棋盘的行数和列数
21         String[] firstLine = br.readLine().split(" ");
22         m = Integer.parseInt(firstLine[0]);
23         n = Integer.parseInt(firstLine[1]);
24         // 初始化棋盘
25         board = new int[m][n];
26
27         // 读取棋盘上每个位置的输入
28         for (int i = 0; i < m; i++) {
29             String[] line = br.readLine().split("");
30             for (int j = 0; j < n; j++) {
31                 // 如果当前位置不是空点, 则将马的位置和步数添加到列表中
32                 if (!line[j].equals(".")) {
33                     horses.add(new int[]{i, j, Integer.parseInt(line[j])});
34                 }
35             }
36         }
37
38         // 调用bfs方法并打印结果
39         System.out.println(bfs());
40     }
41 }
```

```
42 // 定义广度优先搜索方法
```

```
43 private static int bfs() {
```

```
44     // 定义马能走的八个方向
```

```
45     int[][] directions = {{-1, -2}, {-2, -1}, {-2, 1}, {-1, 2}, {1, 2}, {2, 1}, {2, -1}, {1, -2}};
```

```
46     // 初始化最小步数为最大值
```

```
47     int minSteps = Integer.MAX_VALUE;
```

```
49     // 遍历棋盘上的每个位置
```

```
50     for (int i = 0; i < m; i++) {
```

```
51         for (int j = 0; j < n; j++) {
```

```
52             // 初始化当前位置的步数为0
```

```
53             int steps = 0;
```

```
54             // 标记是否所有马都能到达当前位置
```

```
55             boolean possible = true;
```

```
57             // 遍历每个马
```

```
58             for (int[] horse : horses) {
```

```
59                 // 使用队列进行BFS
```

```
60                 Queue<int[]> queue = new LinkedList<>();
```

```
61                 // 使用集合记录已访问的位置
```

```
62                 Set<String> visited = new HashSet<>();
```

```
63                 // 将当前马的位置和步数0加入队列
```

```
64                 queue.offer(new int[]{horse[0], horse[1], 0});
```

```
65                 // 将当前马的位置添加到已访问集合中
```

```
66                 visited.add(horse[0] + "," + horse[1]);
```

```
67                 // 标记是否找到当前位置
```

```
68                 boolean found = false;
```

```
70                 // 当队列不为空且可能到达当前位置时
```

```
71                 while (!queue.isEmpty() && possible) {
```

```
72                     // 取出队列头部元素
```

```
73                     int[] current = queue.poll();
```

```
74                     // 如果当前元素位置等于目标位置
```

```
75                     if (current[0] == i && current[1] == j) {
```

```
76                         // 累加步数
```

```
77                         steps += current[2];
```

```
78                         // 标记为找到
```

```
79                         found = true;
```

```
80                         break;
```

```

82         }
83
84         // 遍历马能走的八个方向
85         for (int[] dir : directions) {
86             // 计算新的位置
87             int nx = current[0] + dir[0];
88             int ny = current[1] + dir[1];
89             // 如果新位置有效且未访问过, 则加入队列
90             if (nx >= 0 && nx < m && ny >= 0 && ny < n && current[2] < horse[2] && !visited.contains(nx + "," + ny)) {
91                 queue.offer(new int[]{nx, ny, current[2] + 1});
92                 visited.add(nx + "," + ny);
93             }
94         }
95     }
96 }
97
98 // 如果没有找到目标位置, 则标记为不可能到达
99 if (!found) {
100     possible = false;
101 }
102
103 // 如果所有马都能到达当前位置, 则更新最小步数
104 if (possible) {
105     minSteps = Math.min(minSteps, steps);
106 }
107 }
108 }
109
110 // 如果最小步数为最大值, 则返回-1, 否则返回最小步数
111 return minSteps == Integer.MAX_VALUE ? -1 : minSteps;
112 }
}

```

JavaScript

```

1  const readline = require('readline');
2
3  // 创建 readline 接口实例
4  const rl = readline.createInterface({
5      input: process.stdin,
6  });

```

```
6   output: process.stdout
7   });
8
9   const lines = []; // 用于存储输入的所有行
10
11   rl.on('line', (input) => {
12     lines.push(input); // 将每行输入存储到 lines 数组中
13   }).on('close', () => { // 输入结束时触发
14     const [m, n] = lines[0].split(' ').map(Number);
15     const board = [];
16     const horses = [];
17
18     // 从第二行开始读取棋盘数据
19     for (let i = 1; i <= m; i++) {
20       const line = lines[i];
21       board.push(line.trim().split(''));
22       // 如果当前位置不是空点, 则将马的位置和步数添加到列表中
23       line.trim().split('').forEach((cell, j) => {
24         if (cell !== '.') {
25           horses.push({ x: i - 1, y: j, steps: parseInt(cell, 10) });
26         }
27       });
28     }
29
30     // 调用 bfs 方法并打印结果
31     console.log(bfs(m, n, horses));
32   });
33
34
35   // 注意: 不要复制这行注释, bfs 函数的实现应该在这里
36
37   // 定义广度优先搜索函数
38   function bfs(m, n, horses) {
39     // 定义马能走的八个方向
40     const directions = [[-1, -2], [-2, -1], [-2, 1], [-1, 2], [1, 2], [2, 1], [2, -1], [1, -2]];
41     // 初始化最小步数为无穷大
42     let minSteps = Infinity;
43
44     // 遍历棋盘上的每个位置
45     for (let i = 0; i < m; i++) {
46
```

```
47 for (let j = 0; j < n; j++) {
48     // 初始化当前位置的步数为0
49     let steps = 0;
50     // 标记是否所有马都能到达当前位置
51     let possible = true;
52
53     // 遍历每个马
54     for (const horse of horses) {
55         // 使用队列进行 BFS
56         const queue = [{ x: horse.x, y: horse.y, step: 0 }];
57         // 使用集合记录已访问的位置
58         const visited = new Set([`${horse.x},${horse.y}`]);
59         // 标记是否找到当前位置
60         let found = false;
61
62         // 当队列不为空且可能到达当前位置时
63         while (queue.length > 0 && possible) {
64             const { x, y, step } = queue.shift();
65             // 如果当前元素位置等于目标位置
66             if (x === i && y === j) {
67                 // 累加步数
68                 steps += step;
69                 // 标记为找到
70                 found = true;
71                 break;
72             }
73
74             // 遍历马能走的八个方向
75             for (const [dx, dy] of directions) {
76                 // 计算新的位置
77                 const nx = x + dx;
78                 const ny = y + dy;
79                 // 如果新位置有效且未访问过, 则加入队列
80                 if (nx >= 0 && nx < m && ny >= 0 && ny < n && step < horse.steps && !visited.has(`${nx},${ny}`)) {
81                     queue.push({ x: nx, y: ny, step: step + 1 });
82                     visited.add(`${nx},${ny}`);
83                 }
84             }
85         }
86     }
87 }
```

```
88     // 如果没有找到目标位置, 则标记为不可能到达
89     if (!found) {
90         possible = false;
91     }
92 }
93
94 // 如果所有马都能到达当前位置, 则更新最小步数
95 if (possible) {
96     minSteps = Math.min(minSteps, steps);
97 }
98 }
99 }
100
101 // 如果最小步数为无穷大, 则返回-1, 否则返回最小步数
    return minSteps === Infinity ? -1 : minSteps;
}
```

Python

```
1 from collections import deque
2
3 # 定义广度优先搜索函数
4 def bfs(m, n, horses):
5     # 定义马能走的八个方向
6     directions = [(-1, -2), (-2, -1), (-2, 1), (-1, 2), (1, 2), (2, 1), (2, -1), (1, -2)]
7     # 初始化最小步数为无穷大
8     min_steps = float('inf')
9
10    # 遍历棋盘上的每个位置
11    for i in range(m):
12        for j in range(n):
13            # 初始化当前位置的步数为0
14            steps = 0
15            # 标记是否所有马都能到达当前位置
16            possible = True
17
18            # 遍历每个马
19            for horse in horses:
20                # 使用队列进行BFS
21                queue = deque([(horse[0], horse[1], 0)])
```

```
22 # 使用集合记录已访问的位置
23 visited = {(horse[0], horse[1])}
24 # 标记是否找到当前位置
25 found = False
26
27 # 当队列不为空且可能到达当前位置时
28 while queue and possible:
29     x, y, step = queue.popleft()
30     # 如果当前元素位置等于目标位置
31     if x == i and y == j:
32         # 累加步数
33         steps += step
34         # 标记为找到
35         found = True
36         break
37
38 # 遍历马能走的八个方向
39 for dx, dy in directions:
40     # 计算新的位置
41     nx, ny = x + dx, y + dy
42     # 如果新位置有效且未访问过, 则加入队列
43     if 0 <= nx < m and 0 <= ny < n and step < horse[2] and (nx, ny) not in visited:
44         queue.append((nx, ny, step + 1))
45         visited.add((nx, ny))
46
47 # 如果没有找到目标位置, 则标记为不可能到达
48 if not found:
49     possible = False
50
51 # 如果所有马都能到达当前位置, 则更新最小步数
52 if possible:
53     min_steps = min(min_steps, steps)
54
55 # 如果最小步数为无穷大, 则返回-1, 否则返回最小步数
56 return -1 if min_steps == float('inf') else min_steps
57
58 # 主函数
59 def main():
60     # 读取棋盘的行数和列数
61     m, n = map(int, input().split())
62     --
```

```
63 # 初始化棋盘
64 board = []
65 # 初始化马的位置和步数的列表
66 horses = []
67
68 # 读取棋盘上每个位置的输入
69 for i in range(m):
70     row = input().strip()
71     board.append(list(row))
72     for j, cell in enumerate(row):
73         # 如果当前位置不是空点, 则将马的位置和步数添加到列表中
74         if cell != '.':
75             horses.append((i, j, int(cell)))
76
77 # 调用bfs方法并打印结果
78 print(bfs(m, n, horses))
79
80 if __name__ == '__main__':
81     main()
```

C语言

```
1 #include <stdio.h>
2 #include <limits.h>
3 #include <string.h>
4
5 // 定义棋盘的最大行数和列数
6 #define MAX_M 100
7 #define MAX_N 100
8
9 // 定义棋盘的行数和列数
10 int m, n;
11 // 定义棋盘
12 char board[MAX_M][MAX_N];
13 // 定义马的位置和步数的结构体
14 typedef struct {
15     int x, y, steps;
16 } Horse;
17 // 定义马的数组
18 Horse horses[MAX_M * MAX_N];
19
```



```
19 // 定义马的数量
20 int horse_count = 0;
21
22 // 定义队列中的元素结构体
23 typedef struct {
24     int x, y, step;
25 } QueueItem;
26
27 // 定义队列
28 QueueItem queue[MAX_M * MAX_N * 8];
29 // 队列的头和尾
30 int queue_head = 0, queue_tail = 0;
31
32 // 队列操作函数
33 void queue_push(QueueItem item) {
34     queue[queue_tail++] = item;
35 }
36
37 QueueItem queue_pop() {
38     return queue[queue_head++];
39 }
40
41 int queue_empty() {
42     return queue_head == queue_tail;
43 }
44
45 // 定义广度优先搜索方法
46 int bfs() {
47     // 定义马能走的八个方向
48     int directions[8][2] = {{-1, -2}, {-2, -1}, {-2, 1}, {-1, 2}, {1, 2}, {2, 1}, {2, -1}, {1, -2}};
49     // 初始化最小步数为最大值
50     int minSteps = INT_MAX;
51
52     // 遍历棋盘上的每个位置
53     for (int i = 0; i < m; ++i) {
54         for (int j = 0; j < n; ++j) {
55             // 初始化当前位置的步数为0
56             int steps = 0;
57             // 标记是否所有马都能到达当前位置
58             int possible = 1;
59         }
60     }
```

```
60
61 // 遍历每个马
62 for (int h = 0; h < horse_count; ++h) {
63     // 使用队列进行BFS
64     queue_head = queue_tail = 0;
65     // 使用二维数组记录已访问的位置
66     int visited[MAX_M][MAX_N];
67     memset(visited, 0, sizeof(visited));
68
69     // 获取马的位置和最大步数
70     Horse horse = horses[h];
71     // 将当前马的位置和步数0加入队列
72     queue_push((QueueItem){horse.x, horse.y, 0});
73     // 将当前马的位置添加到已访问数组中
74     visited[horse.x][horse.y] = 1;
75     // 标记是否找到当前位置
76     int found = 0;
77
78     // 当队列不为空且可能到达当前位置时
79     while (!queue_empty() && possible) {
80         // 取出队列头部元素
81         QueueItem current = queue_pop();
82         // 如果当前元素位置等于目标位置
83         if (current.x == i && current.y == j) {
84             // 累加步数
85             steps += current.step;
86             // 标记为找到
87             found = 1;
88             break;
89         }
90
91         // 遍历马能走的八个方向
92         for (int d = 0; d < 8; ++d) {
93             // 计算新的位置
94             int nx = current.x + directions[d][0];
95             int ny = current.y + directions[d][1];
96             // 如果新位置有效且未访问过, 则加入队列
97             if (nx >= 0 && nx < m && ny >= 0 && ny < n && current.step < horse.steps && !visited[nx][ny]) {
98                 queue_push((QueueItem){nx, ny, current.step + 1});
99                 visited[nx][ny] = 1;
100
```

```
101         }
102     }
103 }
104
105     // 如果没有找到目标位置, 则标记为不可能到达
106     if (!found) {
107         possible = 0;
108     }
109 }
110
111     // 如果所有马都能到达当前位置, 则更新最小步数
112     if (possible) {
113         minSteps = minSteps < steps ? minSteps : steps;
114     }
115 }
116 }
117
118 // 如果最小步数为最大值, 则返回-1, 否则返回最小步数
119 return minSteps == INT_MAX ? -1 : minSteps;
120 }
121
122 // 主函数
123 int main() {
124     // 读取棋盘的行数和列数
125     scanf("%d %d", &m, &n);
126     // 读取棋盘上每个位置的输入
127     for (int i = 0; i < m; ++i) {
128         for (int j = 0; j < n; ++j) {
129             scanf(" %c", &board[i][j]); // 注意%c前的空格, 用于跳过空白字符
130             // 如果当前位置不是空点, 则将马的位置和步数添加到列表中
131             if (board[i][j] != '.') {
132                 horses[horse_count++] = (Horse){i, j, board[i][j] - '0'};
133             }
134         }
135     }
136
137     // 调用bfs方法并打印结果
138     printf("%d\n", bfs());
139     return 0;
140 }
```

文章目录

华为OD机考:统一考试 C卷 + D卷 + B卷 +A卷

题目描述

输入描述

输出描述

用例1

用例二

模拟计算

解题思路

C++

Java

JavaScript

Python

C语言

机考真题 华为OD

