【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~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

```
3 2
. .
2 .
. .
```

输出

```
0
```

用例二

输入

```
3 5
4 7 . 4 8
4 7 4 4 .
7 . . . .
```

输出

```
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:

```
4 7 . 4 8
4 7 4 4 .
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++

```cpp
#include <iostream>
#include <vector>
#include <queue>
#include <set>
#include <string>
#include <climits>
#include <tuple>

using namespace std;

// 定义棋盘的行数和列数
int m, n;
// 定义棋盘
vector<vector<char>> board;
// 定义马的位置和步数的列表
vector<tuple<int, int, int>> horses;

// 定义广度优先搜索方法
int bfs() {
    // 定义马能走的八个方向
    vector<pair<int, int>> directions = {{-1, -2}, {-2, -1}, {-2, 1}, {-1, 2}, {1, 2}, {2, 1}, {2, -1}, {1, -2}};
    // 初始化最小步数为最大值
    int minSteps = INT_MAX;

    // 遍历棋盘上的每个位置
    for (int i = 0; i < m; ++i) {
        for (int j = 0; j < n; ++j) {
            // 初始化当前位置的步数为0
            int steps = 0;
            // 标记是否所有马都能到达当前位置
            bool possible = true;

            // 遍历每个马
            for (auto& horse : horses) {
                // 使用队列进行BFS
                queue<tuple<int, int, int>> queue;
                // 使用集合记录已访问的位置
                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0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
```

```
                if (!found) {
                    possible = false;
                }
            }

            // 如果所有马都能到达当前位置，则更新最小步数
            if (possible) {
                minSteps = min(minSteps, steps);
            }
        }
    }

    // 如果最小步数为最大值，则返回-1，否则返回最小步数
    return minSteps == INT_MAX ? -1 : minSteps;
}

// 主函数
int main() {
    // 读取棋盘的行数和列数
    cin >> m >> n;
    // 初始化棋盘
    board.resize(m, vector<char>(n));
    // 读取棋盘上每个位置的输入
    for (int i = 0; i < m; ++i) {
        for (int j = 0; j < n; ++j) {
            cin >> board[i][j];
            // 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
            if (board[i][j] != '.') {
                horses.emplace_back(i, j, board[i][j] - '0');
            }
        }
    }

    // 调用bfs方法并打印结果
    cout << bfs() << endl;
    return 0;
}
```

Java

```java
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.util.LinkedList;
import java.util.Queue;
import java.util.HashSet;
import java.util.Set;

public class Main {
    // 定义棋盘的行数和列数
    private static int m, n;
    // 定义棋盘
    private static int[][] board;
    // 定义马的位置和步数的列表
    private static LinkedList<int[]> horses = new LinkedList<>();

    public static void main(String[] args) throws IOException {
        // 使用BufferedReader读取输入
        BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
        // 读取第一行输入，获取棋盘的行数和列数
        String[] firstLine = br.readLine().split(" ");
        m = Integer.parseInt(firstLine[0]);
        n = Integer.parseInt(firstLine[1]);
        // 初始化棋盘
        board = new int[m][n];

        // 读取棋盘上每个位置的输入
        for (int i = 0; i < m; i++) {
            String[] line = br.readLine().split("");
            for (int j = 0; j < n; j++) {
                // 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
                if (!line[j].equals(".")) {
                    horses.add(new int[]{i, j, Integer.parseInt(line[j])});
                }
            }
        }

        // 调用bfs方法并打印结果
        System.out.println(bfs());
    }
```

```
    // 定义广度优先搜索方法
    private static int bfs() {
        // 定义马能走的八个方向
        int[][] directions = {{-1, -2}, {-2, -1}, {-2, 1}, {-1, 2}, {1, 2}, {2, 1}, {2, -1}, {1, -2}};
        // 初始化最小步数为最大值
        int minSteps = Integer.MAX_VALUE;

        // 遍历棋盘上的每个位置
        for (int i = 0; i < m; i++) {
            for (int j = 0; j < n; j++) {
                // 初始化当前位置的步数为0
                int steps = 0;
                // 标记是否所有马都能到达当前位置
                boolean possible = true;

                // 遍历每个马
                for (int[] horse : horses) {
                    // 使用队列进行BFS
                    Queue<int[]> queue = new LinkedList<>();
                    // 使用集合记录已访问的位置
                    Set<String> visited = new HashSet<>();
                    // 将当前马的位置和步数0加入队列
                    queue.offer(new int[]{horse[0], horse[1], 0});
                    // 将当前马的位置添加到已访问集合中
                    visited.add(horse[0] + "," + horse[1]);
                    // 标记是否找到当前位置
                    boolean found = false;

                    // 当队列不为空且可能到达当前位置时
                    while (!queue.isEmpty() && possible) {
                        // 取出队列头部元素
                        int[] current = queue.poll();
                        // 如果当前元素位置等于目标位置
                        if (current[0] == i && current[1] == j) {
                            // 累加步数
                            steps += current[2];
                            // 标记为找到
                            found = true;
                            break;
```

```
                    }

                    // 遍历马能走的八个方向
                    for (int[] dir : directions) {
                        // 计算新的位置
                        int nx = current[0] + dir[0];
                        int ny = current[1] + dir[1];
                        // 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 && current[2] < horse[2] && !visited.contains(nx + "," + ny)) {
                            queue.offer(new int[]{nx, ny, current[2] + 1});
                            visited.add(nx + "," + ny);
                        }
                    }
                }

                // 如果没有找到目标位置，则标记为不可能到达
                if (!found) {
                    possible = false;
                }
            }

            // 如果所有马都能到达当前位置，则更新最小步数
            if (possible) {
                minSteps = Math.min(minSteps, steps);
            }
        }
    }

    // 如果最小步数为最大值，则返回-1，否则返回最小步数
    return minSteps == Integer.MAX_VALUE ? -1 : minSteps;
  }
}
```

javaScript

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

// 创建 readline 接口实例
const rl = readline.createInterface({
  input: process.stdin,
```

```
  output: process.stdout
});

const lines = []; // 用于存储输入的所有行

rl.on('line', (input) => {
  lines.push(input); // 将每行输入存储到 lines 数组中
}).on('close', () => { // 输入结束时触发
  const [m, n] = lines[0].split(' ').map(Number);
  const board = [];
  const horses = [];

  // 从第二行开始读取棋盘数据
  for (let i = 1; i <= m; i++) {
    const line = lines[i];
    board.push(line.trim().split(''));
    // 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
    line.trim().split('').forEach((cell, j) => {
      if (cell !== '.') {
        horses.push({ x: i - 1, y: j, steps: parseInt(cell, 10) });
      }
    });
  }

  // 调用 bfs 方法并打印结果
  console.log(bfs(m, n, horses));
});


// 注意：不要复制这行注释，bfs 函数的实现应该在这里

// 定义广度优先搜索函数
function bfs(m, n, horses) {
  // 定义马能走的八个方向
  const directions = [[-1, -2], [-2, -1], [-2, 1], [-1, 2], [1, 2], [2, 1], [2, -1], [1, -2]];
  // 初始化最小步数为无穷大
  let minSteps = Infinity;

  // 遍历棋盘上的每个位置
  for (let i = 0; i < m; i++) {
```

```
    for (let j = 0; j < n; j++) {
      // 初始化当前位置的步数为0
      let steps = 0;
      // 标记是否所有马都能到达当前位置
      let possible = true;

      // 遍历每个马
      for (const horse of horses) {
        // 使用队列进行 BFS
        const queue = [{ x: horse.x, y: horse.y, step: 0 }];
        // 使用集合记录已访问的位置
        const visited = new Set([`${horse.x},${horse.y}`]);
        // 标记是否找到当前位置
        let found = false;

        // 当队列不为空且可能到达当前位置时
        while (queue.length > 0 && possible) {
          const { x, y, step } = queue.shift();
          // 如果当前元素位置等于目标位置
          if (x === i && y === j) {
            // 累加步数
            steps += step;
            // 标记为找到
            found = true;
            break;
          }

          // 遍历马能走的八个方向
          for (const [dx, dy] of directions) {
            // 计算新的位置
            const nx = x + dx;
            const ny = y + dy;
            // 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 && step < horse.steps && !visited.has(`${nx},${ny}`)) {
              queue.push({ x: nx, y: ny, step: step + 1 });
              visited.add(`${nx},${ny}`);
            }
          }
        }
```

```
        // 如果没有找到目标位置，则标记为不可能到达
        if (!found) {
          possible = false;
        }
      }

      // 如果所有马都能到达当前位置，则更新最小步数
      if (possible) {
        minSteps = Math.min(minSteps, steps);
      }
    }
  }

  // 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

```
from collections import deque

# 定义广度优先搜索函数
def bfs(m, n, horses):
    # 定义马能走的八个方向
    directions = [(-1, -2), (-2, -1), (-2, 1), (-1, 2), (1, 2), (2, 1), (2, -1), (1, -2)]
    # 初始化最小步数为无穷大
    min_steps = float('inf')

    # 遍历棋盘上的每个位置
    for i in range(m):
        for j in range(n):
            # 初始化当前位置的步数为0
            steps = 0
            # 标记是否所有马都能到达当前位置
            possible = True

            # 遍历每个马
            for horse in horses:
                # 使用队列进行BFS
                queue = deque([(horse[0], horse[1], 0)])
```

```
                # 使用集合记录已访问的位置
                visited = {(horse[0], horse[1])}
                # 标记是否找到当前位置
                found = False

                # 当队列不为空且可能到达当前位置时
                while queue and possible:
                    x, y, step = queue.popleft()
                    # 如果当前元素位置等于目标位置
                    if x == i and y == j:
                        # 累加步数
                        steps += step
                        # 标记为找到
                        found = True
                        break

                    # 遍历马能走的八个方向
                    for dx, dy in directions:
                        # 计算新的位置
                        nx, ny = x + dx, y + dy
                        # 如果新位置有效且未访问过，则加入队列
                        if 0 <= nx < m and 0 <= ny < n and step < horse[2] and (nx, ny) not in visited:
                            queue.append((nx, ny, step + 1))
                            visited.add((nx, ny))

                # 如果没有找到目标位置，则标记为不可能到达
                if not found:
                    possible = False

            # 如果所有马都能到达当前位置，则更新最小步数
            if possible:
                min_steps = min(min_steps, steps)

    # 如果最小步数为无穷大，则返回-1，否则返回最小步数
    return -1 if min_steps == float('inf') else min_steps

# 主函数
def main():
    # 读取棋盘的行数和列数
    m, n = map(int, input().split())
```

```
    # 初始化棋盘
    board = []
    # 初始化马的位置和步数的列表
    horses = []

    # 读取棋盘上每个位置的输入
    for i in range(m):
        row = input().strip()
        board.append(list(row))
        for j, cell in enumerate(row):
            # 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
            if cell != '.':
                horses.append((i, j, int(cell)))

    # 调用bfs方法并打印结果
    print(bfs(m, n, horses))

if __name__ == '__main__':
    main()
```

C语言

```
#include <stdio.h>
#include <limits.h>
#include <string.h>

// 定义棋盘的最大行数和列数
#define MAX_M 100
#define MAX_N 100

// 定义棋盘的行数和列数
int m, n;
// 定义棋盘
char board[MAX_M][MAX_N];
// 定义马的位置和步数的结构体
typedef struct {
    int x, y, steps;
} Horse;
// 定义马的数组
Horse horses[MAX_M * MAX_N];
```

```
// 定义马的数量
int horse_count = 0;

// 定义队列中的元素结构体
typedef struct {
    int x, y, step;
} QueueItem;

// 定义队列
QueueItem queue[MAX_M * MAX_N * 8];
// 队列的头和尾
int queue_head = 0, queue_tail = 0;

// 队列操作函数
void queue_push(QueueItem item) {
    queue[queue_tail++] = item;
}

QueueItem queue_pop() {
    return queue[queue_head++];
}

int queue_empty() {
    return queue_head == queue_tail;
}

// 定义广度优先搜索方法
int bfs() {
    // 定义马能走的八个方向
    int directions[8][2] = {{-1, -2}, {-2, -1}, {-2, 1}, {-1, 2}, {1, 2}, {2, 1}, {2, -1}, {1, -2}};
    // 初始化最小步数为最大值
    int minSteps = INT_MAX;

    // 遍历棋盘上的每个位置
    for (int i = 0; i < m; ++i) {
        for (int j = 0; j < n; ++j) {
            // 初始化当前位置的步数为0
            int steps = 0;
            // 标记是否所有马都能到达当前位置
            int possible = 1;
```

```
        // 遍历每个马
        for (int h = 0; h < horse_count; ++h) {
            // 使用队列进行BFS
            queue_head = queue_tail = 0;
            // 使用二维数组记录已访问的位置
            int visited[MAX_M][MAX_N];
            memset(visited, 0, sizeof(visited));

            // 获取马的位置和最大步数
            Horse horse = horses[h];
            // 将当前马的位置和步数0加入队列
            queue_push((QueueItem){horse.x, horse.y, 0});
            // 将当前马的位置添加到已访问数组中
            visited[horse.x][horse.y] = 1;
            // 标记是否找到当前位置
            int found = 0;

            // 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
                QueueItem current = queue_pop();
                // 如果当前元素位置等于目标位置
                if (current.x == i && current.y == j) {
                    // 累加步数
                    steps += current.step;
                    // 标记为找到
                    found = 1;
                    break;
                }

                // 遍历马能走的八个方向
                for (int d = 0; d < 8; ++d) {
                    // 计算新的位置
                    int nx = current.x + directions[d][0];
                    int ny = current.y + directions[d][1];
                    // 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 && current.step < horse.steps && !visited[nx][ny]) {
                        queue_push((QueueItem){nx, ny, current.step + 1});
                        visited[nx][ny] = 1;
```

```
                    }
                }
            }

            // 如果没有找到目标位置，则标记为不可能到达
            if (!found) {
                possible = 0;
            }
        }

        // 如果所有马都能到达当前位置，则更新最小步数
        if (possible) {
            minSteps = minSteps < steps ? minSteps : steps;
        }
    }
}

// 如果最小步数为最大值，则返回-1，否则返回最小步数
return minSteps == INT_MAX ? -1 : minSteps;
}

// 主函数
int main() {
    // 读取棋盘的行数和列数
    scanf("%d %d", &m, &n);
    // 读取棋盘上每个位置的输入
    for (int i = 0; i < m; ++i) {
        for (int j = 0; j < n; ++j) {
            scanf(" %c", &board[i][j]); // 注意%c前的空格，用于跳过空白字符
            // 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
            if (board[i][j] != '.') {
                horses[horse_count++] = (Horse){i, j, board[i][j] - '0'};
            }
        }
    }

    // 调用bfs方法并打印结果
    printf("%d\n", bfs());
    return 0;
}
```

文章目录