【华为OD机考 统一考试机试C卷】亲子游戏（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

宝宝和妈妈参加亲子游戏，在一个二维矩阵（N*N）的格子地图上，宝宝和妈妈抽签决定各自的位置，地图上每个格子有不同的糖果数量，部分格子有障碍物。

游戏规则是妈妈必须在最短的时间（每个单位时间只能走一步）到达宝宝的位置，路上的所有糖果都可以拿走，不能走障碍物的格子，只能上下左右走。

请问妈妈在最短到达宝宝位置的时间内最多拿到多少糖果（优先考虑最短时间到达的情况下尽可能多拿糖果）。

输入描述

第一行输入为 N，N 表示二维矩阵的大小

之后 N 行，每行有 N 个值，表格矩阵每个位置的值，其中：

- -3：妈妈
- -2：宝宝
- -1：障碍
- ≥0：糖果数（0表示没有糖果，但是可以走）

输出描述

输出妈妈在最短到达宝宝位置的时间内最多拿到多少糖果，行末无多余空格

用例

输入

```
4
3 2 1 -3
1 -1 1 1
1 1 -1 2
-2 1 2 3
```

输出

```
9
```

说明

此地图有两条最短路径可到达宝宝位置，绿色线和黄色线都是最短路径6步，但黄色拿到的糖果更多，9个。

输入

```
4
3 2 1 -3
-1 -1 1 1
1 1 -1 2
-2 1 -1 3
```

输出

```
-1
```

说明

此地图妈妈无法到达宝宝位置

解题思路

1. **初始化**：
   - 初始化 `grid` 矩阵存储输入的网格数据。
   - 初始化 `visited` 三维数组来记录每个位置的最短步数和糖果数量，初始值为 `-1`。
2. **读取网格信息**：
   - 通过双层循环读取网格中的每个值。
   - 如果遇到起点（值为 `-3`），则创建起点 `Node` 并更新 `visited` 数组中起点的步数和糖果数为 `0`。
3. **广度优先搜索（BFS）**：
   - 将起点 `Node` 加入队列 `queue`。
   - 初始化最大糖果数 `maxCandies` 为 `0`，用 `flag` 标记是否到达终点。
   - 当队列不为空时，循环执行以下步骤：
     - 从队列中取出一个节点 `current`。
     - 如果当前节点是终点（值为 `-2`），则更新 `maxCandies` 并继续循环。
     - 遍历四个可能的移动方向。
     - 对于每个方向，计算新位置的坐标 `nx` 和 `ny`。
     - 检查新位置是否在网格内、不是障碍物（值不为 `-1`）。
     - 计算到达新位置的糖果数 `newCandies` 和步数 `newSteps`。
     - 如果新位置未访问过，或者可以以更少的步数到达，或者步数相同但糖果数更多，则更新 `visited` 数组并将新位置的 `Node` 加入队列。
4. **输出结果**：
   - 如果 `flag` 为 `0`，说明没有到达终点，将 `maxCandies` 设置为 `-1`。
   - 输出最大糖果数 `maxCandies`。如果没有到达终点，则输出 `-1`。
5. **辅助类 `Node`**：
   - 定义了一个 `Node` 类来表示 BFS 中的每个状态，包含位置坐标 `(x, y)`、糖果数 `candies` 和步数 `steps`。

C++

```cpp
#include <iostream>
#include <vector>
#include <queue>
#include <algorithm>
using namespace std;

// 定义四个方向移动的坐标变化（上、右、下、左）
const int dx[4] = {-1, 0, 1, 0};
const int dy[4] = {0, 1, 0, -1};

// 节点类，用于表示 BFS 中的每个状态
struct Node {
    int x, y, candies, steps;
    // 构造函数，初始化节点位置、糖果数和步数
    Node(int _x, int _y, int _candies, int _steps) : x(_x), y(_y), candies(_candies), steps(_steps) {}
};

int main() {
    int N; // 矩阵的大小
    cin >> N; // 输入矩阵的大小
    vector<vector<int>> grid(N, vector<int>(N)); // 创建一个二维矩阵存储输入的网格
    // 创建一个三维数组来标记每个位置的访问状态，包括步数和糖果数
    vector<vector<vector<int>>> visited(N, vector<vector<int>>(N, vector<int>(2, -1)));
    queue<Node> queue; // 创建一个队列用于 BFS
    Node start(0, 0, 0, 0); // 初始化起点
    int maxCandies = 0; // 最大糖果数初始化为 0
    int flag = 0;
    // 读取网格数据，并找到起点
    for (int i = 0; i < N; ++i) {
        for (int j = 0; j < N; ++j) {
            cin >> grid[i][j]; // 输入网格中的每个值
            if (grid[i][j] == -3) { // 如果是起点
                start = Node(i, j, 0, 0); // 更新起点信息
                visited[i][j][0] = 0; // 标记起点的步数为 0
                visited[i][j][1] = 0; // 标记起点的糖果数为 0
            }
        }
    }

    queue.push(start); // 将起点加入队列
```

```
    // BFS 搜索
    while (!queue.empty()) {
        Node current = queue.front(); // 取出队列前端的节点
        queue.pop(); // 弹出队列前端的节点
        if (grid[current.x][current.y] == -2) { // 如果到达终点
            flag = 1;
            maxCandies = max(maxCandies, current.candies); // 更新最大糖果数
            continue; // 继续搜索其他路径
        }

        // 遍历四个方向
        for (int i = 0; i < 4; ++i) {
            int nx = current.x + dx[i]; // 计算新的 x 坐标
            int ny = current.y + dy[i]; // 计算新的 y 坐标

            // 检查新坐标是否在网格内以及是否可走
            if (nx >= 0 && nx < N && ny >= 0 && ny < N && grid[nx][ny] != -1) {
                int newCandies = current.candies + max(grid[nx][ny], 0); // 计算新的糖果数
                int newSteps = current.steps + 1; // 计算新的步数
                // 如果新坐标未访问过或者有更优的路径（步数更少或糖果数更多）
                if (visited[nx][ny][0] == -1 || visited[nx][ny][0] > newSteps ||
                    (visited[nx][ny][0] == newSteps && visited[nx][ny][1] < newCandies)) {
                    queue.push(Node(nx, ny, newCandies, newSteps)); // 将新节点加入队列
                    visited[nx][ny][0] = newSteps; // 更新访问状态的步数
                    visited[nx][ny][1] = newCandies; // 更新访问状态的糖果数
                }
            }
        }
    }
    if(flag == 0){
        maxCandies = -1;
    }

    // 输出最大糖果数，如果没有到达终点则输出 -1
    cout << (maxCandies >= 0 ? maxCandies : -1) << endl;
    return 0;
}
```

Java

```java
import java.util.*;

// 主类
public class Main {
    // 定义四个方向移动的坐标变化（上、右、下、左）
    private static final int[] dx = {-1, 0, 1, 0};
    private static final int[] dy = {0, 1, 0, -1};

    // 主函数
    public static void main(String[] args) {
        // 使用 Scanner 读取输入数据
        Scanner scanner = new Scanner(System.in);
        // 读取矩阵的大小
        int N = scanner.nextInt();
        // 初始化矩阵
        int[][] grid = new int[N][N];
        // 初始化访问数组，记录到达每个位置的最短步数和糖果数量
        int[][][] visited = new int[N][N][2]; // [x][y][0] 代表步数，[x][y][1] 代表糖果数
        // 将访问数组初始化为 -1
        for (int[][] layer : visited) {
            for (int[] cell : layer) {
                Arrays.fill(cell, -1);
            }
        }
        // 初始化队列，用于 BFS 搜索
        Queue<Node> queue = new LinkedList<>();
        // 初始化起点
        Node start = null;

        // 读取矩阵信息，并找到起点位置
        for (int i = 0; i < N; i++) {
            for (int j = 0; j < N; j++) {
                grid[i][j] = scanner.nextInt();
                if (grid[i][j] == -3) { // 如果是起点
                    start = new Node(i, j, 0, 0); // 创建起点节点
                    visited[i][j][0] = 0; // 起点的步数为 0
                    visited[i][j][1] = 0; // 起点的糖果数为 0
                }
            }
        }
```

```
        // 关闭 Scanner
        scanner.close();

        // 将起点加入队列
        queue.add(start);
        // 初始化最大糖果数
        int maxCandies = 0;
        int flag = 0;
        // BFS 搜索
        while (!queue.isEmpty()) {
            Node current = queue.poll();
            // 如果到达终点，更新最大糖果数
            if (grid[current.x][current.y] == -2) {
                flag = 1;
                maxCandies = Math.max(maxCandies, current.candies);
                continue;
            }

            // 遍历四个方向
            for (int i = 0; i < 4; i++) {
                int nx = current.x + dx[i];
                int ny = current.y + dy[i];

                // 检查新位置是否有效
                if (nx >= 0 && nx < N && ny >= 0 && ny < N && grid[nx][ny] != -1) {
                    // 计算新位置的糖果数和步数
                    int newCandies = current.candies + Math.max(grid[nx][ny], 0);
                    int newSteps = current.steps + 1;
                    // 如果新位置未访问过，或者可以以更少的步数到达，或者步数相同但糖果数更多，则更新信息并加入队列
                    if (visited[nx][ny][0] == -1 || visited[nx][ny][0] > newSteps ||
                        (visited[nx][ny][0] == newSteps && visited[nx][ny][1] < newCandies)) {
                        queue.add(new Node(nx, ny, newCandies, newSteps));
                        visited[nx][ny][0] = newSteps;
                        visited[nx][ny][1] = newCandies;
                    }
                }
            }
        }
        if(flag == 0){
            maxCandies = -1;
```

```
        }
        // 输出最大糖果数，如果没有到达终点则输出 -1
        System.out.println(maxCandies >= 0 ? maxCandies : -1);
    }

    // 节点类，用于表示 BFS 中的每个状态
    static class Node {
        int x, y, candies, steps;

        // 节点构造函数
        public Node(int x, int y, int candies, int steps) {
            this.x = x;
            this.y = y;
            this.candies = candies;
            this.steps = steps;
        }
    }
}
```

javaScript

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

// 定义四个方向移动的坐标变化（上、右、下、左）
const dx = [-1, 0, 1, 0];
const dy = [0, 1, 0, -1];

// 节点类，用于表示 BFS 中的每个状态
class Node {
    constructor(x, y, candies, steps) {
        this.x = x; // 当前节点的 x 坐标
        this.y = y; // 当前节点的 y 坐标
        this.candies = candies; // 从起点到当前节点收集到的糖果数
        this.steps = steps; // 从起点到当前节点的步数
    }
}
```

```
// 当读取到一行输入时触发
rl.on('line', (line) => {
    const parts = line.split(' ').map(x => parseInt(x)); // 将输入的行分割成数组并转换为整数
    if (parts.length === 1) {
        // 如果只有一个数字，则表示是矩阵的大小
        const N = parts[0]; // 矩阵的大小
        const grid = []; // 存储矩阵的二维数组
        // 创建一个三维数组来标记每个位置的访问状态，包括步数和糖果数
        const visited = Array.from({ length: N }, () => Array.from({ length: N }, () => [-1, -1]));
        let linesRead = 0; // 已读取的行数
        let start = null; // 起点
        let maxCandies = 0; // 最大糖果数

        // 当读取到 N 行后，开始处理矩阵
        rl.on('line', (line) => {
            grid.push(line.split(' ').map(x => parseInt(x))); // 将每行的数据添加到矩阵中
            linesRead++; // 增加已读取的行数
            if (linesRead === N) {
                // 如果已读取完矩阵的所有行
                // 初始化起点和队列
                const queue = [];
                for (let i = 0; i < N; i++) {
                    for (let j = 0; j < N; j++) {
                        if (grid[i][j] === -3) {
                            start = new Node(i, j, 0, 0); // 找到起点并创建起点节点
                            visited[i][j] = [0, 0]; // 标记起点为已访问
                            break;
                        }
                    }
                }
                queue.push(start); // 将起点加入队列
                let flag = 0;
                // BFS 搜索
                while (queue.length > 0) {
                    const current = queue.shift(); // 取出队列的第一个节点
                    if (grid[current.x][current.y] === -2) {
                        // 如果当前节点是终点
                        flag = 1;
                        maxCandies = Math.max(maxCandies, current.candies); // 更新最大糖果数
```

```
                    continue;
                }

                // 遍历四个方向
                for (let i = 0; i < 4; i++) {
                    const nx = current.x + dx[i]; // 计算新的 x 坐标
                    const ny = current.y + dy[i]; // 计算新的 y 坐标

                    // 检查新坐标是否在矩阵内以及是否可走
                    if (nx >= 0 && nx < N && ny >= 0 && ny < N && grid[nx][ny] !== -1) {
                        const newCandies = current.candies + Math.max(grid[nx][ny], 0); // 计算新的糖果数
                        const newSteps = current.steps + 1; // 计算新的步数
                        // 如果新坐标未访问过或者有更优的路径（步数更少或糖果数更多）
                        if (visited[nx][ny][0] === -1 || visited[nx][ny][0] > newSteps ||
                            (visited[nx][ny][0] === newSteps && visited[nx][ny][1] < newCandies)) {
                            queue.push(new Node(nx, ny, newCandies, newSteps)); // 将新节点加入队列
                            visited[nx][ny] = [newSteps, newCandies]; // 更新访问状态
                        }
                    }
                }
            }
            if(flag === 0){
                maxCandies = -1;
            }
            // 输出最大糖果数，如果没有到达终点则输出 -1
            console.log(maxCandies >= 0 ? maxCandies : -1);
            rl.close(); // 关闭 readline 接口
        }
    });
}
});
```

Python

```
from collections import deque

# 定义四个方向移动的坐标变化（上、右、下、左）
dx = [-1, 0, 1, 0]
dy = [0, 1, 0, -1]

```

```python
# 节点类，用于表示 BFS 中的每个状态
class Node:
    def __init__(self, x, y, candies, steps):
        self.x = x  # 当前节点的 x 坐标
        self.y = y  # 当前节点的 y 坐标
        self.candies = candies  # 从起点到当前节点收集到的糖果数
        self.steps = steps  # 从起点到当前节点的步数

# 主函数
if __name__ == "__main__":
    N = int(input())  # 读取矩阵的大小
    grid = []  # 存储矩阵的二维数组
    # 创建一个三维数组来标记每个位置的访问状态，包括步数和糖果数
    visited = [[[-1, -1] for _ in range(N)] for _ in range(N)]
    max_candies = 0  # 最大糖果数

    # 读取矩阵数据并找到起点
    for i in range(N):
        grid.append(list(map(int, input().split())))  # 将每行的数据添加到矩阵中
        for j in range(N):
            if grid[i][j] == -3:  # 如果当前位置是起点
                start = Node(i, j, 0, 0)  # 创建起点节点
                visited[i][j] = [0, 0]  # 标记起点为已访问

    # 使用双端队列来进行 BFS 搜索
    queue = deque([start])
    flag = 0
    # BFS 搜索
    while queue:
        current = queue.popleft()  # 取出队列的第一个节点
        if grid[current.x][current.y] == -2:  # 如果当前节点是终点
            flag = 1
            max_candies = max(max_candies, current.candies)  # 更新最大糖果数
            continue

        # 遍历四个方向
        for i in range(4):
            nx = current.x + dx[i]  # 计算新的 x 坐标
            ny = current.y + dy[i]  # 计算新的 y 坐标
```

```
            # 检查新坐标是否在矩阵内以及是否可走
            if 0 <= nx < N and 0 <= ny < N and grid[nx][ny] != -1:
                new_candies = current.candies + max(grid[nx][ny], 0)  # 计算新的糖果数
                new_steps = current.steps + 1  # 计算新的步数
                # 如果新坐标未访问过或者有更优的路径（步数更少或糖果数更多）
                if visited[nx][ny][0] == -1 or visited[nx][ny][0] > new_steps or \
                   (visited[nx][ny][0] == new_steps and visited[nx][ny][1] < new_candies):
                    queue.append(Node(nx, ny, new_candies, new_steps))  # 将新节点加入队列
                    visited[nx][ny] = [new_steps, new_candies]  # 更新访问状态
    if flag == 0:
        max_candies = -1
    # 输出最大糖果数，如果没有到达终点则输出 -1
    print(max_candies if max_candies >= 0 else -1)
```

C语言

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

#define MAX_N 50

// 定义四个方向移动的坐标变化（上、右、下、左）
const int dx[4] = {-1, 0, 1, 0};
const int dy[4] = {0, 1, 0, -1};

// 节点结构体，用于表示 BFS 中的每个状态
typedef struct {
    int x, y, candies, steps;
} Node;

// 队列结构体，用于 BFS 搜索
typedef struct {
    Node nodes[MAX_N * MAX_N];
    int front, rear;
} Queue;

// 初始化队列
void initQueue(Queue *q) {
    q->front = q->rear = 0;
```

```
}

// 队列是否为空
int isEmpty(Queue *q) {
    return q->front == q->rear;
}

// 入队操作
void push(Queue *q, Node node) {
    q->nodes[q->rear++] = node;
}

// 出队操作
Node pop(Queue *q) {
    return q->nodes[q->front++];
}

// 主函数
int main() {
    int N; // 矩阵的大小
    scanf("%d", &N); // 输入矩阵的大小
    int grid[MAX_N][MAX_N]; // 创建一个二维矩阵存储输入的网格
    int visited[MAX_N][MAX_N][2]; // 创建一个三维数组来标记每个位置的访问状态，包括步数和糖果数
    memset(visited, -1, sizeof(visited)); // 初始化访问状态数组为 -1
    Queue queue; // 创建一个队列用于 BFS
    initQueue(&queue); // 初始化队列
    Node start = {0, 0, 0, 0}; // 初始化起点
    int maxCandies = 0; // 最大糖果数初始化为 0
    int flag = 0; // 标记是否到达终点

    // 读取网格数据，并找到起点
    for (int i = 0; i < N; ++i) {
        for (int j = 0; j < N; ++j) {
            scanf("%d", &grid[i][j]); // 输入网格中的每个值
            if (grid[i][j] == -3) { // 如果是起点
                start.x = i;
                start.y = j;
                visited[i][j][0] = 0; // 标记起点的步数为 0
                visited[i][j][1] = 0; // 标记起点的糖果数为 0
            }

```

```
    }
}

push(&queue, start); // 将起点加入队列

// BFS 搜索
while (!isEmpty(&queue)) {
    Node current = pop(&queue); // 取出队列前端的节点
    if (grid[current.x][current.y] == -2) { // 如果到达终点
        flag = 1;
        if (current.candies > maxCandies) {
            maxCandies = current.candies; // 更新最大糖果数
        }
        continue; // 继续搜索其他路径
    }

    // 遍历四个方向
    for (int i = 0; i < 4; ++i) {
        int nx = current.x + dx[i]; // 计算新的 x 坐标
        int ny = current.y + dy[i]; // 计算新的 y 坐标

        // 检查新坐标是否在网格内以及是否可走
        if (nx >= 0 && nx < N && ny >= 0 && ny < N && grid[nx][ny] != -1) {
            int newCandies = current.candies + (grid[nx][ny] > 0 ? grid[nx][ny] : 0); // 计算新的糖果数
            int newSteps = current.steps + 1; // 计算新的步数
            // 如果新坐标未访问过或者有更优的路径（步数更少或糖果数更多）
            if (visited[nx][ny][0] == -1 || visited[nx][ny][0] > newSteps ||
                (visited[nx][ny][0] == newSteps && visited[nx][ny][1] < newCandies)) {
                Node newNode = {nx, ny, newCandies, newSteps};
                push(&queue, newNode); // 将新节点加入队列
                visited[nx][ny][0] = newSteps; // 更新访问状态的步数
                visited[nx][ny][1] = newCandies; // 更新访问状态的糖果数
            }
        }
    }
}
if (flag == 0) {
    maxCandies = -1; // 如果没有到达终点，则最大糖果数为 -1
}
```

```
    // 输出最大糖果数，如果没有到达终点则输出 -1
    printf("%d\n", maxCandies);
    return 0;
}
```

文章目录