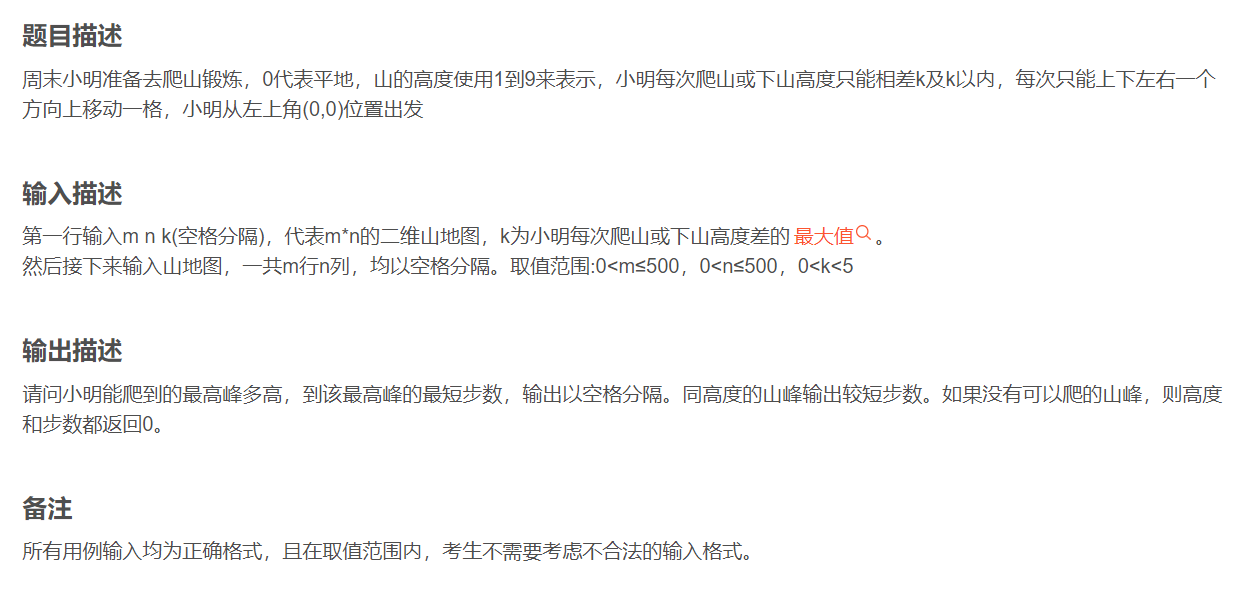
# **E卷-小明周末爬山[200分]（ Java | Python3 | C++ | C语言 | JsNode | Go ）**





5 4 1

0 1 2 0

1 0 0 0

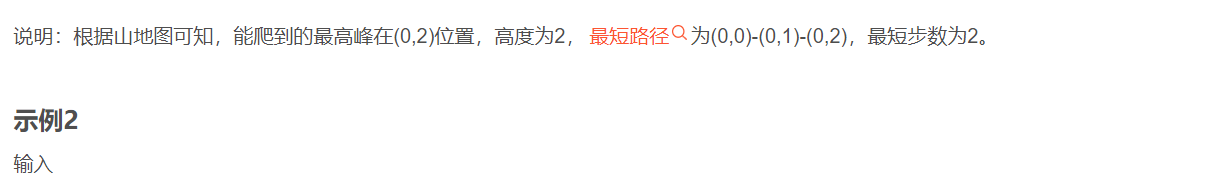
1 0 1 2

1 3 1 0

0 0 0 9



2 2



5 4 3

0 0 0 0

0 0 0 0

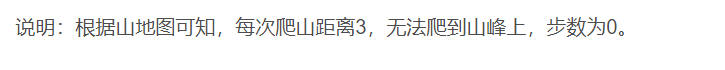
0 9 0 0

0 0 0 0

0 0 0 9



0 0





import java.util.\*;

public class Main {

// 判断从当前高度移动到下一个高度是否满足高度差要求

private static boolean canClimb(int current, int next, int k) {

return Math.abs(current - next) <= k;

}

// 使用广度优先搜索（BFS）来遍历地图

private static int[] bfs(int m, int n, int k, int[][] mountain) {

int[] directions = {-1, 0, 1, 0, 0, -1, 0, 1}; // 上下左右四个方向

boolean[][] visited = new boolean[m][n]; // 记录已访问的位置

Queue<int[]> queue = new LinkedList<>(); // 存储 (x, y, steps)

queue.offer(new int[]{0, 0, 0}); // 从左上角开始

visited[0][0] = true;

int highestPeak = 0;

int shortestSteps = 0;

while (!queue.isEmpty()) {

int[] current = queue.poll();

int x = current[0], y = current[1], steps = current[2];

int currentHeight = mountain[x][y];

// 如果到达更高的峰或到达同样高度的峰但步数更小

if (currentHeight > highestPeak || (currentHeight == highestPeak && steps < shortestSteps)) {

highestPeak = currentHeight;

shortestSteps = steps;

}

for (int i = 0; i < directions.length; i += 2) {

int nx = x + directions[i], ny = y + directions[i + 1];

if (nx >= 0 && nx < m && ny >= 0 && ny < n && !visited[nx][ny]

&& canClimb(currentHeight, mountain[nx][ny], k)) { // 如果可以爬到

visited[nx][ny] = true;

queue.offer(new int[]{nx, ny, steps + 1});

}

}

}

return new int[]{highestPeak, shortestSteps};

}

public static void main(String[] args) {

Scanner scanner = new Scanner(System.in);

// 读取输入

int m = scanner.nextInt();

int n = scanner.nextInt();

int k = scanner.nextInt();

int[][] mountain = new int[m][n];

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

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

mountain[i][j] = scanner.nextInt();

}

}

// 获取结果

int[] result = bfs(m, n, k, mountain);

System.out.println(result[0] + " " + result[1]);

}

}



from collections import deque

def can\_climb(current, next, k):

# 判断从current高度到next高度是否满足要求

return abs(current - next) <= k

def bfs(m, n, k, mountain):

directions = [(-1, 0), (1, 0), (0, -1), (0, 1)]

visited = [[False] \* n for \_ in range(m)]

queue = deque([(0, 0, 0)]) # 存储 (x, y, steps)

visited[0][0] = True

highest\_peak = 0

shortest\_steps = 0

while queue:

x, y, steps = queue.popleft()

current\_height = mountain[x][y]

# 如果到达更高的峰或到达同样高度的峰但步数更小

if current\_height > highest\_peak or (

current\_height == highest\_peak and steps < shortest\_steps

):

highest\_peak = current\_height

shortest\_steps = steps

for direction in directions:

nx, ny = x + direction[0], y + direction[1]

if 0 <= nx < m and 0 <= ny < n and not visited[nx][ny]:

# 如果可以爬到

if can\_climb(current\_height, mountain[nx][ny], k):

visited[nx][ny] = True

queue.append((nx, ny, steps + 1))

return highest\_peak, shortest\_steps

# 读取输入

m, n, k = map(int, input().strip().split())

mountain = []

for \_ in range(m):

mountain.append(list(map(int, input().strip().split())))

# 获取结果

result = bfs(m, n, k, mountain)

print(result[0], result[1])



#include <iostream>

#include <vector>

#include <queue>

#include <cmath>

using namespace std;

// 判断从当前高度移动到下一个高度是否满足高度差要求

bool canClimb(int current, int next, int k) {

return abs(current - next) <= k;

}

// 使用广度优先搜索（BFS）来遍历地图

pair<int, int> bfs(int m, int n, int k, vector<vector<int>>& mountain) {

vector<int> directions = {-1, 0, 1, 0, 0, -1, 0, 1}; // 上下左右四个方向

vector<vector<bool>> visited(m, vector<bool>(n, false)); // 记录已访问的位置

queue<tuple<int, int, int>> q; // 存储 (x, y, steps)

q.push({0, 0, 0}); // 从左上角开始

visited[0][0] = true;

int highestPeak = 0;

int shortestSteps = 0;

while (!q.empty()) {

auto [x, y, steps] = q.front();

q.pop();

int currentHeight = mountain[x][y];

// 如果到达更高的峰或到达同样高度的峰但步数更小

if (currentHeight > highestPeak || (currentHeight == highestPeak && steps < shortestSteps)) {

highestPeak = currentHeight;

shortestSteps = steps;

}

for (int i = 0; i < directions.size(); i += 2) {

int nx = x + directions[i], ny = y + directions[i + 1];

if (nx >= 0 && nx < m && ny >= 0 && ny < n && !visited[nx][ny]

&& canClimb(currentHeight, mountain[nx][ny], k)) { // 如果可以爬到

visited[nx][ny] = true;

q.push({nx, ny, steps + 1});

}

}

}

return {highestPeak, shortestSteps};

}

int main() {

int m, n, k;

cin >> m >> n >> k;

vector<vector<int>> mountain(m, vector<int>(n));

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

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

cin >> mountain[i][j];

}

}

auto result = bfs(m, n, k, mountain);

cout << result.first << " " << result.second << endl;

return 0;

}



#include <stdio.h>

#include <stdlib.h>

#include <stdbool.h>

#include <limits.h>

// 判断从当前高度到达下一个高度是否满足要求

bool canClimb(int current, int next, int k) {

return abs(current - next) <= k;

}

// 定义队列节点结构体

typedef struct Node {

int x, y, steps;

struct Node\* next;

} Node;

// 定义队列结构体

typedef struct Queue {

Node\* front, \*rear;

} Queue;

// 初始化队列

Queue\* createQueue() {

Queue\* q = (Queue\*)malloc(sizeof(Queue));

q->front = q->rear = NULL;

return q;

}

// 向队列添加节点

void enqueue(Queue\* q, int x, int y, int steps) {

Node\* node = (Node\*)malloc(sizeof(Node));

node->x = x;

node->y = y;

node->steps = steps;

node->next = NULL;

if (q->rear == NULL) {

q->front = q->rear = node;

return;

}

q->rear->next = node;

q->rear = node;

}

// 从队列获取节点

Node\* dequeue(Queue\* q) {

if (q->front == NULL) return NULL;

Node\* temp = q->front;

q->front = q->front->next;

if (q->front == NULL) q->rear = NULL;

return temp;

}

// 使用广度优先搜索（BFS）来遍历地图

void bfs(int m, int n, int k, int mountain[m][n], int\* highestPeak,

int\* shortestSteps) {

int directions[8] = {-1, 0, 1, 0, 0, -1, 0, 1}; // 上下左右四个方向

bool visited[m][n]; // 记录已访问的位置

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

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

visited[i][j] = false;

}

}

Queue\* queue = createQueue();

enqueue(queue, 0, 0, 0); // 从左上角开始

visited[0][0] = true;

\*highestPeak = 0;

\*shortestSteps = 0;

while (queue->front != NULL) {

Node\* current = dequeue(queue);

int x = current->x, y = current->y, steps = current->steps;

int currentHeight = mountain[x][y];

// 如果到达更高的峰或到达同样高度的峰但步数更小

if (currentHeight > \*highestPeak || (currentHeight == \*highestPeak &&

steps < \*shortestSteps)) {

\*highestPeak = currentHeight;

\*shortestSteps = steps;

}

for (int i = 0; i < 8; i += 2) {

int nx = x + directions[i], ny = y + directions[i + 1];

if (nx >= 0 && nx < m && ny >= 0 && ny < n && !visited[nx][ny]

&& canClimb(currentHeight, mountain[nx][ny], k)) { // 如果可以爬到

visited[nx][ny] = true;

enqueue(queue, nx, ny, steps + 1);

}

}

free(current);

}

free(queue);

}

int main() {

int m, n, k;

scanf("%d %d %d", &m, &n, &k);

int mountain[m][n];

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

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

scanf("%d", &mountain[i][j]);

}

}

int highestPeak, shortestSteps;

bfs(m, n, k, mountain, &highestPeak, &shortestSteps);

printf("%d %d\n", highestPeak, shortestSteps);

return 0;

}



const readline = require("readline");

// 创建接口用于读取标准输入

const rl = readline.createInterface({

input: process.stdin,

output: process.stdout,

});

let inputLines = []; // 存储输入的每一行

rl.on("line", function (line) {

inputLines.push(line.trim()); // 读取输入行

});

rl.on("close", function () {

const [m, n, k] = inputLines[0].split(" ").map(Number); // 解析第一行的 m, n, k

let mountain = [];

for (let i = 1; i <= m; i++) {

mountain.push(inputLines[i].split(" ").map(Number)); // 解析山地图

}

// 判断从当前高度移动到下一个高度是否满足高度差要求

function canClimb(current, next, k) {

return Math.abs(current - next) <= k;

}

// 使用广度优先搜索（BFS）来遍历地图

function bfs(m, n, k, mountain) {

const directions = [-1, 0, 1, 0, 0, -1, 0, 1]; // 上下左右四个方向

const visited = Array.from({ length: m }, () => Array(n).fill(false)); // 记录已访问的位置

let queue = [{ x: 0, y: 0, steps: 0 }]; // 从左上角开始

visited[0][0] = true;

let highestPeak = 0;

let shortestSteps = 0;

while (queue.length) {

const { x, y, steps } = queue.shift();

const currentHeight = mountain[x][y];

// 如果到达更高的峰或到达同样高度的峰但步数更小

if (

currentHeight > highestPeak ||

(currentHeight == highestPeak && steps < shortestSteps)

) {

highestPeak = currentHeight;

shortestSteps = steps;

}

for (let i = 0; i < directions.length; i += 2) {

const nx = x + directions[i],

ny = y + directions[i + 1];

if (

nx >= 0 &&

nx < m &&

ny >= 0 &&

ny < n &&

!visited[nx][ny] &&

canClimb(currentHeight, mountain[nx][ny], k)

) {

// 如果可以爬到

visited[nx][ny] = true;

queue.push({ x: nx, y: ny, steps: steps + 1 });

}

}

}

return [highestPeak, shortestSteps];

}

const result = bfs(m, n, k, mountain);

console.log(result.join(" ")); // 输出结果

});



package main

import (

"bufio"

"fmt"

"math"

"os"

"strconv"

"strings"

)

// 判断从当前高度移动到下一个高度是否满足高度差要求

func canClimb(current, next, k int) bool {

return math.Abs(float64(current-next)) <= float64(k)

}

// 使用广度优先搜索（BFS）来遍历地图

func bfs(m, n, k int, mountain [][]int) (int, int) {

directions := []int{-1, 0, 1, 0, 0, -1, 0, 1} // 上下左右四个方向

visited := make([][]bool, m) // 记录已访问的位置

for i := range visited {

visited[i] = make([]bool, n)

}

queue := [][3]int{{0, 0, 0}} // 队列存储 (x, y, steps)

visited[0][0] = true

highestPeak, shortestSteps := 0, 0

for len(queue) > 0 {

current := queue[0]

queue = queue[1:]

x, y, steps := current[0], current[1], current[2]

currentHeight := mountain[x][y]

// 如果到达更高的峰或到达同样高度的峰但步数更小

if currentHeight > highestPeak || (currentHeight == highestPeak && steps < shortestSteps) {

highestPeak = currentHeight

shortestSteps = steps

}

for i := 0; i < len(directions); i += 2 {

nx, ny := x+directions[i], y+directions[i+1]

if nx >= 0 && nx < m && ny >= 0 && ny < n && !visited[nx][ny] && canClimb(currentHeight, mountain[nx][ny], k) { // 如果可以爬到

visited[nx][ny] = true

queue = append(queue, [3]int{nx, ny, steps + 1})

}

}

}

return highestPeak, shortestSteps

}

func main() {

reader := bufio.NewReader(os.Stdin)

input, \_ := reader.ReadString('\n')

input = strings.TrimSpace(input)

params := strings.Split(input, " ")

m, \_ := strconv.Atoi(params[0])

n, \_ := strconv.Atoi(params[1])

k, \_ := strconv.Atoi(params[2])

mountain := make([][]int, m)

for i := 0; i < m; i++ {

line, \_ := reader.ReadString('\n')

line = strings.TrimSpace(line)

strs := strings.Split(line, " ")

mountain[i] = make([]int, n)

for j := 0; j < n; j++ {

mountain[i][j], \_ = strconv.Atoi(strs[j])

}

}

highestPeak, shortestSteps := bfs(m, n, k, mountain)

fmt.Printf("%d %d\n", highestPeak, shortestSteps)

}