天津大学


# 程序设计综合实践课程报告

# 搜索算法实验

学生姓名 陈昊昆

学院名称 智算学部

专　　业 软件工程

学　　号 **3021001196**

# 1. 正方形

## 1.1 题目分析

**在搜索过程中，需要记录已经搜索过的木棍，以避免重复搜索**

**如果当前木棍能够用于构成当前正方形边长，则进行搜索**

**如果当前边长等于正方形边长，则表示已经找到了一条边**

**需要重置长度为 0，然后继续搜索下一条边**

**最后，如果找到了 4 条边，则表示可以用这些木棍构成正方形**

## 1.2 题目代码（带注释）

```
#include <bits/stdc++.h>
using namespace std;

const int MAXN = 20;
double nums[MAXN];
double sum, side;
int cnt;
bool used[MAXN];

// 判断能否构成正方形
bool dfs(int cur, double len) {
    if (len == side) {
        if (++cnt == 4) return true;
        return dfs(0, 0); // 继续搜索下一边
    }
    for (int i = cur; i < sum; i++) {
        if (!used[i] && len + nums[i] <= side) { // 能用于构成边
            used[i] = true;
            if (dfs(i + 1, len + nums[i])) return true; //加入后 构造成功
            used[i] = false;
        }
    }
    return false;
}
```

```
int main() {
    int t;
    cin >> t;
    while (t--) {
        int n;
        cin >> n;
        sum = 0;
        for (int i = 0; i < n; i++) {
            cin >> nums[i];
            sum += nums[i];
            used[i] = false;
        }
        if (sum < 4) {
            cout << "no" << endl;
            continue;
        }
        side = sum / 4.0; // 计算正方形边长
        if (side != int(side)) {
            cout << "no" << endl;
            continue;
        }
        cnt = 0;
        if (dfs(0, 0)) cout << "yes" << endl;
        else cout << "no" << endl;
    }
    return 0;
}
```

# 2. prime circle

## 2.1 题目分析

**第一个位置开始搜索**

**每次只需要枚举下一个位置的数，判断是否满足相邻之和为素数**

**如果当前枚举到的位置是最后一个位置，则判断它和第一个位置的和是否为素数，如果是则输出结果，否则继续搜索**

## 2.2 题目代码（带注释）

```
#include <bits/stdc++.h>
using namespace std;

const int N = 20;
int n;
bool used[N];  // 标记数字是否已经使用过
int path[N];  // 存储搜索结果

// 判断一个数是否是素数
bool is_prime(int x)
{
    if (x < 2) return false;
    for (int i = 2; i <= x / i; i++)
        if (x % i == 0) return false;
    return true;
}

void dfs(int u)
{
    // 如果已经搜索到最后一个位置，则判断是否满足条件，输出结果并返回
    if (u == n)
    {
        if (is_prime(path[0] + path[n - 1]))
        {
            for (int i = 0; i < n; i++) cout << path[i] << " ";
            cout << endl;
        }
```

```
        return;
    }
    // 枚举下一个位置可以使用的数字
    for (int i = 2; i <= n; i++)
        if (!used[i] && is_prime(i + path[u - 1]))
        {
            used[i] = true;  // 标记该数字已经使用过
            path[u] = i;  // 将该数字存入搜索结果中
            dfs(u + 1);  // 继续搜索下一个位置
            used[i] = false;  // 恢复现场
        }
}

int main()
{

    int T = 0;
    while (cin >> n)
    {
        if (T++) cout << endl;
        for (int i = 0; i < n; i++){
            used[i] = false;
        }
        path[0] = 1;  // 第一个位置必须是 1
        cout << "Case " << T << ":" << endl;
        dfs(1);
    }

    return 0;
}
```

# 3. 棋盘问题

## 3.1 题目分析

**通过深度优先搜索，枚举每一个棋子放在哪个位置**

**并检查是否满足任意两个棋子不在同一行或同一列的要求**

**最终计算所有可行的方案数**

## 3.2 题目代码（带注释）

```
#include <bits/stdc++.h>
using namespace std;

const int MAXN = 10;
int n, k, tot;  // n: 棋盘大小, k: 棋子数量, tot: 棋盘中的 # 数量
int row[MAXN], a[MAXN][MAXN];
long long ans;  // 方案数
char c;

void dfs(int x, int now) {
    if (x == k + 1) {
        ans++;
        return;
    }
    // 从上一个棋子所在的列号之后开始枚举可能的列号
    for (int i = now + 1; i <= tot; i++) {
        int flag = 1;
        for (int j = 1; j <= n; ++j) {
            if (a[row[j]][i]) {
                flag = 0;
                break;
            }
        }
        // 如果在第 i 列可以放置棋子
        if (flag) {
            row[x] = i;
            dfs(x + 1, i);
            row[x] = 0;
```

```
        }
    }
}

int main() {
    while (cin >> n >> k, n != -1 || k != -1) {
        tot = 0;
        ans = 0;
        for (int i = 0; i < n; i++){
            row[i] = 0;
            for(int j = 0; j < n; j++){
                a[i][j] = 0;
            }
        }
        for (int i = 1; i <= n; ++i) {
            for (int j = 1; j <= n; j++) {
                cin >> c;
                if (c == '#') {
                    a[i][++tot] = 1;  // 表示可以放棋子
                }
            }
        }
        dfs(1, 0);
        cout << ans << endl;
    }
    return 0;
}
```

# 4. 迷宫问题

## 4.1 题目分析

**定义了一个结构体 N 来表示每个格子的位置和它的前一个位置**

**一个二维数组 maze 来表示迷宫的矩阵**

**一个二维数组 visited 来表示每个格子是否被访问过**

**一个队列 q 来保存每个被访问的格子**

**以及一个数组 kDirections 来表示上下左右四个方向**

**程序从起点开始遍历，每次访问周围的四个方向，如果这个方向的格子是空的且没有被访问过，则把它加入队列。当遍历到终点时，程序就找到了最短路径，最后使用输出路径**

## 4.2 题目代码（带注释）

```
#include <bits/stdc++.h>
using namespace std;

const int MAX_N = 5;

int maze[MAX_N][MAX_N];  // 地图
bool vis[MAX_N][MAX_N];  // 记录节点是否已访问

struct Node {
  int x, y, pre;  // 当前节点坐标、前驱节点编号
} nodes[50];  // 存储 BFS 遍历时的节点信息

const int kDirections[4][2] = {{-1, 0}, {1, 0}, {0, -1}, {0, 1}};  // 上下左右四个方向

int front = 0, rear = 0;  // BFS 遍历队列的头尾指针

void bfs(int sx, int sy, int ex, int ey) {
    nodes[0].x = sx;
    nodes[0].y = sy;
    nodes[0].pre = -1;
```

```
    rear++;
    vis[sx][sy] = true;
    while (front < rear) {
    for (int i = 0; i < 4; i++) {
        int nx = nodes[front].x + kDirections[i][0];
        int ny = nodes[front].y + kDirections[i][1];
        if (nx < 0 || nx >= MAX_N || ny < 0 || ny >= MAX_N) {
            continue;  // 越界，继续检查下一个方向
        }
        if (maze[nx][ny] == 1) {
            continue;  // 障碍物，无法通过，继续检查下一个方向
        }
        if (vis[nx][ny]) {
        continue;  // 已经访问过，不需要再次访问，继续检查下一个方向
        }
        nodes[rear].x = nx;
        nodes[rear].y = ny;
        nodes[rear].pre = front;
        rear++;
        vis[nx][ny] = true;
        if (nx == ex && ny == ey) {
            return;  // 已经找到终点，结束搜索
        }
    }
    front++;
    }
}

void print_path(Node now) {
    if (now.pre == -1) {
        printf("(%d, %d)\n", now.x, now.y);
        return;
    }
    print_path(nodes[now.pre]);
    printf("(%d, %d)\n", now.x, now.y);
}

int main() {
    // 读入地图
    for (int i = 0; i < MAX_N; i++) {
        for (int j = 0; j < MAX_N; j++) {
            cin >> maze[i][j];
            vis[i][j] = false;
        }
```

```
    }

  // BFS 搜索
    bfs(0, 0, 4, 4);

  // 输出路径
    print_path(nodes[rear - 1]);

    return 0;
}
```

# 5. Find a way

## 5.1 题目分析

**题目中给出了一个地图，包含了若干条路径、若干个位置以及一些不能经过的障碍物，需要求出两个人到达某个位置的最小时间和**

**在 BFS 中，需要使用队列来存储待搜索的节点，每次从队列中取出一个节点进行扩展，直到找到目标节点为止**

**每次扩展一个节点需要记录该节点所在位置、已经走过的时间、以及另一个人的位置和已经走过的时间，每次扩展时需要判断是否到达了终点或者不能扩展了**

## 5.2 题目代码（带注释）

```
#include <bits/stdc++.h>
using namespace std;

const int INF = 100000;

int n, m;
char mp[205][205];  // 存储地图信息
int dis[2][205][205];  // 存储起点到各个点的最短路径
int sx[2], sy[2];  // 起点的坐标
bool vis[205][205];  // 标记是否访问过

struct Node {
    int x, y;
};

// 方向数组，用于在地图上移动
int dx[] = {0, 1, 0, -1};
int dy[] = {1, 0, -1, 0};

// 判断当前点是否可以走
bool is_valid(int x, int y) {
    return x >= 1 && x <= n && y >= 1 && y <= m && mp[x][y] != '#';
}
```

```
// bfs 遍历求出起点到各个点的最短路径
void bfs(int id) {
    memset(vis, false, sizeof(vis));
    queue<Node> q;
    q.push({sx[id], sy[id]});
    dis[id][sx[id]][sy[id]] = 0;
    vis[sx[id]][sy[id]] = true;
    while (!q.empty()) {
        auto node = q.front();
        q.pop();
        int x = node.x, y = node.y;
        for (int i = 0; i < 4; i++) {
            int nx = x + dx[i], ny = y + dy[i];
            if (is_valid(nx, ny) && !vis[nx][ny]) {
                vis[nx][ny] = true;
                dis[id][nx][ny] = dis[id][x][y] + 11;
                q.push({nx, ny});
            }
        }
    }
}

int main() {
    while (cin >> n >> m) {
        memset(mp, 0, sizeof(mp));
        memset(dis, INF, sizeof(dis));
        for (int i = 1; i <= n; i++) {
            for (int j = 1; j <= m; j++) {
                cin >> mp[i][j];
                if (mp[i][j] == 'Y') {
                    sx[0] = i;
                    sy[0] = j;
                } else if (mp[i][j] == 'M') {
                    sx[1] = i;
                    sy[1] = j;
                }
            }
        }

        bfs(0);  // 以 Y 点为起点进行 bfs
        bfs(1);  // 以 M 点为起点进行 bfs

        int ans = INF;
        for (int i = 1; i <= n; i++) {
```

```
            for (int j = 1; j <= m; j++) {
                if (mp[i][j] == '@') {
                    ans = min(ans, dis[0][i][j] + dis[1][i][j]);  // 计
算最小总时间
                }
            }
        }
        cout << ans << endl;
    }
    return 0;
}
```

# 6. 马的遍历

## 6.1 题目分析

**对于每个格子，使用 BFS（广度优先搜索）求出从马当前位置到此位置最少的步数**

**如果走到了当前点但还没有达到目标位置，应该继续搜索下去**

## 6.2 题目代码（带注释）

```
#include<bits/stdc++.h>
using namespace std;

int n, m, sx, sy; //n, m 为棋盘的长和宽，sx, sy 为马的坐标
int dx[8] = {1, 2, 2, 1, -1, -2, -2, -1};
int dy[8] = {2, 1, -1, -2, -2, -1, 1, 2};
int d[405][405]; //d[i][j]表示从马的坐标到(i, j)的最少步数
void bfs()
{
    memset(d, -1, sizeof(d)); //先将所有的距离初始化为-1，表示无法到达
    queue<pair<int, int> >q;
    q.push(make_pair(sx, sy));
    d[sx][sy] = 0; //马到达自己的坐标，距离为 0
    while(!q.empty())//队列非空，进行 BFS 搜索
    {
        pair<int, int> p = q.front();
        q.pop();
        for(int i = 0; i<8; i++)//尝试八个方向的移动
        {
            int nx = p.first+dx[i]; //横坐标移动的距离
            int ny = p.second+dy[i]; //纵坐标移动的距离
            if(nx >= 1 && nx<= n && ny>= 1 && ny<= m && d[nx][ny] == -
1)//如果能移动到该坐标且该坐标还没有被访问过
            {
                d[nx][ny] = d[p.first][p.second]+1; //更新到该坐标的最短
距离
                q.push(make_pair(nx, ny));
            }
```

```
        }
    }
}
int main()
{
    cin >> n >> m >> sx >> sy;
    bfs(); //进行 BFS 搜索
    for(int i = 1; i <= n; i++)//输出结果
    {
        for(int j = 1; j <= m; j++)
        {
            cout << d[i][j] << " "; //输出(i, j)的最少步数
        }
        cout << endl;
    }
    return 0;
}
```

# 7. 求细胞数量

## 7.1 题目分析

**从每个起点开始，利用 DFS 找到与其相邻的所有细胞，并将它们标记为已访问计算未被标记为已访问的细胞的数量，即为细胞个数。**

**用一个二维数组来记录矩阵中各个位置是否被访问过**

## 7.2 题目代码（带注释）

```
#include<bits/stdc++.h>
using namespace std;

const int MAXN = 400;
int n, m;
char a[MAXN][MAXN];
int vis[MAXN][MAXN]; // 记录该位置是否被访问过
int ans; // 细胞个数

int dx[4] = {1, -1, 0, 0}; // 上下左右四个方向
int dy[4] = {0, 0, -1, 1};

void dfs(int x, int y) // 搜索函数
{
    vis[x][y] = 1; // 标记该位置已经被访问过
    for (int i = 0; i < 4; i++) // 枚举四个方向
    {
        int nx = x + dx[i];
        int ny = y + dy[i];
        if (nx >= 1 && nx <= n && ny >= 1 && ny <= m && a[nx][ny] != '0'
&& !vis[nx][ny]) // 判断是否可以往该位置走
            dfs(nx, ny);
    }
}

int main()
{
    cin >> n >> m;
```

```
    for (int i = 1; i <= n; i++)
        for (int j = 1; j <= m; j++)
            cin >> a[i][j];
    for (int i = 1; i <= n; i++)
        for (int j = 1; j <= m; j++)
            if (a[i][j] != '0' && !vis[i][j]) // 如果该位置有细胞且还没被
访问过
            {
                ans++;
                dfs(i, j); // 搜索该位置的连通块
            }
    cout << ans << endl; // 输出答案
    return 0;
}
```

# 8. 01 迷宫

## 8.1 题目分析

**利用深度优先搜索，遍历迷宫的每一个位置，并搜索与当前位置相邻且值不同的位置，**

**用 vis 数组标记已经遍历过的位置，以避免重复遍历**

**当询问某一位置时，如果该位置已经被标记，则说明该位置在同一个连通块中，否则说明该位置不在同一个连通块中**

**在搜索某一位置时，求解该位置所在的连通块大小**

## 8.2 题目代码（带注释）

```
#include<bits/stdc++.h>
using namespace std;

const int N = 410;

int n, m;
char g[N][N];
bool vis[N][N];
int dx[4] = {-1, 0, 1, 0}, dy[4] = {0, 1, 0, -1};

// 深度优先搜索，求解连通块大小
int dfs(int x, int y)
{
    // 标记当前位置已经遍历过
    vis[x][y] = true;
    // 连通块大小初始化为 1（包括自身）
    int size = 1;
    // 枚举相邻四个位置
    for (int i = 0; i < 4; i ++)
    {
        int a = x + dx[i], b = y + dy[i];
        // 判断相邻位置是否越界，是否为连通块的一部分，是否已经遍历过
        if (a < 0 || a >= n || b < 0 || b >= n || g[x][y] == g[a][b] ||
vis[a][b])
```

```
            continue;
        // 如果符合条件，则以相邻位置为起点继续遍历
        size += dfs(a, b);
    }
    // 返回连通块大小
    return size;
}

int main()
{
    cin >> n >> m;
    for (int i = 0; i < n; i ++)
        cin >> g[i];

    while (m -- )
    {
        int x, y;
        cin >> x >> y;
        x --, y --;
        // 搜索从当前位置开始的连通块大小
        int size = dfs(x, y);
        // 输出连通块大小
        cout << size << endl;
        // 将已经遍历过的位置重新标记为未遍历过
        memset(vis, false, sizeof vis);
    }

    return 0;
}
```