福州大学
FUZHOU UNIVERSITY

搜 索 算 法

福州大学 FUZHOU UNIVERSITY

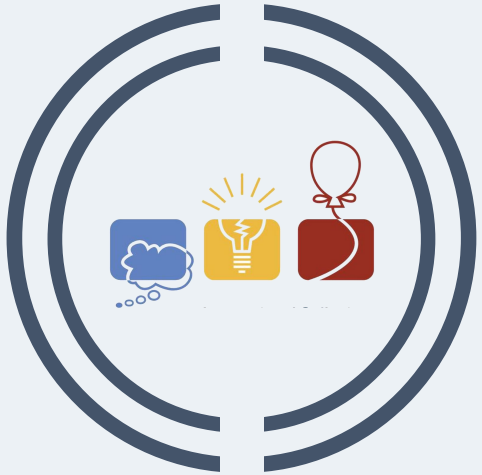
目　　录

深度优先搜索

全排列问题

走迷宫问题

前置知识：递归



广度优先搜索

走迷宫（改）问题

水坑计数问题

前置知识：队列

福州大学
FUZHOU UNIVERSITY

深 度 优 先 搜 索

福州大学 FUZHOU UNIVERSITY

全 排 列 问 题

求自然数 1 到 n 所有的排列，即 n 的全排列。

$1 \le n \le 9$

Source: Luogu P1706

全 排 列 问 题

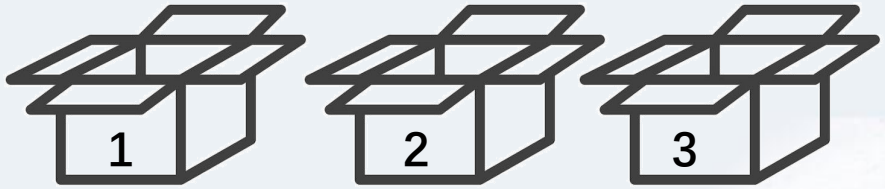
由于数据范围小，显然可以直接用循环语句枚举。

但是，有更简单的方法吗?

我们不妨从 n = 3 的小规模情况入手。

假设我们手上有三张卡片，分别写着 1 2 3。

然后有三个标号为1 2 3的盒子。



把卡片放入盒子里，有哪些放法?

全排列问题（n=3）

简单手玩后，我们可以得到这样一段伪代码。

```
for(检查所有卡片){
    if(这张卡没用过){
        放到盒子里面
        去搞下一个盒子
        回收卡片
    }
}
```

全排列问题（n=3）

考虑把伪代码进行转化。

检查所有卡片可以用for循环从 1 到 3 实现。

确定一张卡片有没有用过可以开一个 bool 数组 used。

used[i] = true 表示这张卡片已经被放过了

直接判断 used[i] 的值即可知道这张卡片有没有被用过。

```
for(int i=1;i<=3;i++){
    if(used[i]!=true){
        放到盒子里面
        去搞下一个盒子
        回收卡片
    }
}
```

福州大学 FUZHOU UNIVERSITY

全排列问题（n=3）

如何实现把卡片放到盒子里面的效果?

开个 int 数组 box。

box[i] 里面存放进第 i 个盒子里面的卡片数字。

放卡的时候记得要更新对应卡片的 used。

```
for(int i=1;i<=3;i++){
    if(used[i]!=true){
        box[j]=i;  //假设当前放到了第 j 个盒子
        used[i]=true;
        去搞下一个盒子
        回收卡片
    }
}
```

全排列问题（n=3）

如何回收卡片?

改 used 和 box 的值?

仔细一想会发现下次放卡片的时候，

新的卡片数值会覆盖掉原来的卡片。

因此不用改 box 的值，只要改 used 的值即可。

```
for(int i=1;i<=3;i++){
    if(used[i]!=true){
        box[j]=i;  //假设当前放到了第 j 个盒子
        used[i]=true;
        去搞下一个盒子
        used[i]=false;
    }
}
```

全排列问题（n=3）

如何做到去找下一个盒子?

用循环语句显然不合适。

我们注意到每次处理盒子的过程是相似的。

不妨考虑使用递归。

可以发现，递归很好的满足了我们的要求。

```
void put_card_into_box(int j){

    for(int i=1;i<=3;i++){
        if(used[i]!=true){
            box[j]=i;  //假设当前放到了第 j 个盒子
            used[i]=true;
            put_card_into_box(j+1); //去处理第 j+1 个盒子
            used[i]=false;
        }
    }

}
```

福州大学 FUZHOU UNIVERSITY

全排列问题（n=3）

但是，我们不能让程序一直递归下去。

边界条件是什么？

考虑 n = 3 时，那第 4 个盒子就是不存在的。

这时候就要结束递归。

使用 if 语句在 for 循环之前判断一下即可。

```
bool used[mxn];
int box[mxn];

void put_card_into_box(int j){

    if(j==4){  //第四个盒子不存在，结束递归
        for(int i=1;i<=3;i++)
            printf("%d ",box[i]);
        putchar('\n');  //输出每个盒子里的卡片编号
        return;
    }

    for(int i=1;i<=3;i++){
        if(used[i]!=true){
            box[j]=i;  //假设当前放到了第 j 个盒子
            used[i]=true;
            put_card_into_box(j+1); //去处理第 j+1 个盒子
            used[i]=false;
        }
    }

}

int main(){
    put_card_into_box(1);  //从第一个盒子开始放
    return 0;
}
```

全 排 列 问 题

推广 n = 3 情况下的代码：

- ✓ 从第 1 个盒子处理到第 n 个盒子；
- ✓ 卡片编号从 1 到 n；
- ✓ 第 n + 1 个盒子不存在。

在原来的代码上简单修改即可。

```
bool used[mxn];
int box[mxn],n;

void put_card_into_box(int j){

    if(j==n+1){  //第n+1个盒子不存在，结束递归
        for(int i=1;i<=n;i++)
            printf("%d ",box[i]);
        putchar('\n');  //输出每个盒子里的卡片编号
        return;
    }

    for(int i=1;i<=n;i++){
        if(used[i]!=true){
            box[j]=i;  //假设当前放到了第 j 个盒子
            used[i]=true;
            put_card_into_box(j+1); //去处理第 j+1 个盒子
            used[i]=false;
        }
    }

}

int main(){
    scanf("%d",&n);
    put_card_into_box(1);  //从第一个盒子开始放
    return 0;
}
```

深 度 优 先 搜 索

恭喜你解决了全排列问题！

复盘一下这个过程？有两个关键点：

1. 做出某一步的决策，然后进入下一步，直到碰到边界；

2. 回头，还原现场，尝试其它决策的可能。

这个过程就是深度优先搜索（Depth First Search，简称DFS）。

其中，走回头路还原现场的部分有个别称：回溯。

```
void dfs(int step){  //当前进行第 step 步决策
    if(/*到达边界*/){
        return; //回头
    }
    //考虑第一种决策可能
    dfs(step+1);
    //还原现场
    //考虑第二种决策可能
    dfs(step+1);
    //还原现场
    //...
}
```

走 迷 宫 问 题

给定一个 N * M 方格的迷宫，迷宫里有 T 处障碍，障碍处不可通过。

在迷宫中移动有上下左右四种方式，每次只能移动一个方格。数据保证起点上没有障碍。

给定起点坐标和终点坐标，每个方格最多经过一次，问有多少种从起点坐标到终点坐标的方案。

1 <= N,M <= 5, 1 <= T <= 10

Source: Luogu P1605

走 迷 宫 问 题

同样的，我们可以先自己在纸上画一个小的迷宫玩一玩。根据我们上一问的模型，可以得到：

- 状态：在迷宫中的某个位置；
- 下一步的决策：上下左右四个方向选一个没有障碍没有出迷宫边界的方向走；
- 边界条件：走到迷宫终点。

于是我们可以得到这样一段代码。

```
void dfs(int now_x,int now_y){
    if(now_x==end_x&&now_y==end_y){
        count_of_ans++;
        return;
    }
    if(now_x+1>=0&&now_y>=0&&now_x+1<n&&now_y<m&&Map[now_x+1][now_y]!='#'){
        //判断是否超出了迷宫地图边界并且下一步走的位置不是墙
        dfs(now_x+1,now_y);
    }
    if(now_x-1>=0&&now_y>=0&&now_x-1<n&&now_y<m&&Map[now_x-1][now_y]!='#'){
        dfs(now_x-1,now_y);
    }
    if(now_x>=0&&now_y+1>=0&&now_x<n&&now_y+1<m&&Map[now_x][now_y+1]!='#'){
        dfs(now_x,now_y+1);
    }
    if(now_x>=0&&now_y-1>=0&&now_x<n&&now_y-1<m&&Map[now_x][now_y-1]!='#'){
        dfs(now_x,now_y-1);
    }
}
```

走 迷 宫 问 题

然后我们发现，这段代码死循环了。

仔细发现，因为每次决策都是向四个方向走，从第二步开始就可能会走回头路，最后导致了无限的递归。

解决问题的关键是在每一步决策时，知道自己那些位置已经走过了。我们可以开一个二维的 bool 数组 vis，vis[i][j]=true代表（i，j）这个格子已经被访问过了。每次决策只去走 vis[x][y]=false 的格子，把 vis[x][y] 先修改为 true，回溯的时候改回 false，就可以避免走回头路的问题。

我们把这种记下已经搜过的东西来避免重复搜索节省时间的方法，称为记忆化，这种搜索方式也叫记忆化搜索。

走 迷 宫 问 题

加入上述修改后，核心代码如下：

```
void dfs(int now_x,int now_y){
    if(now_x==end_x&&now_y==end_y){  //走到终点，方案数+1
        count_of_ans++;
        return;
    }
    if(now_x+1>=0&&now_y>=0&&now_x+1<n&&now_y<m&&Map[now_x+1][now_y]!='#'&&!vis[now_x+1][now_y]){
        //判断是否超出了迷宫地图边界并且下一步走的位置不是墙
        vis[now_x+1][now_y]=true;
        dfs(now_x+1,now_y);
        vis[now_x+1][now_y]=false;
    }
    if(now_x-1>=0&&now_y>=0&&now_x-1<n&&now_y<m&&Map[now_x-1][now_y]!='#'&&!vis[now_x-1][now_y]){
        vis[now_x-1][now_y]=true;
        dfs(now_x-1,now_y);
        vis[now_x-1][now_y]=false;
    }
    if(now_x>=0&&now_y+1>=0&&now_x<n&&now_y+1<m&&Map[now_x][now_y+1]!='#'&&!vis[now_x][now_y+1]){
        vis[now_x][now_y+1]=true;
        dfs(now_x,now_y+1);
        vis[now_x][now_y+1]=false;
    }
    if(now_x>=0&&now_y-1>=0&&now_x<n&&now_y-1<m&&Map[now_x][now_y-1]!='#'&&!vis[now_x][now_y-1]){
        vis[now_x][now_y-1]=true;
        dfs(now_x,now_y-1);
        vis[now_x][now_y+1]=false;
    }
}
```

然而，这个代码重复的部分是不是感觉有点多。

更具体的，我们要对四个方向每个方向都要写一大段的 if 语句进行操作，而这四个部分都大同小异。

这只是四个方向，如果是八个方向（象棋中的马）怎么办？

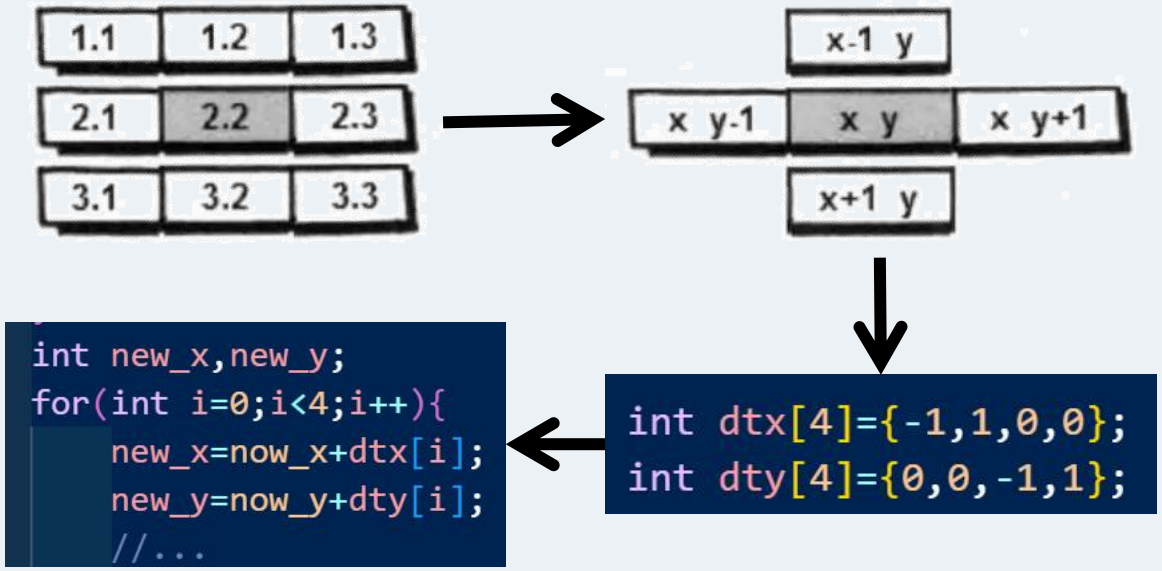
需要简化！

走 迷 宫 问 题

仔细观察会发现，每一个方向只有 x，y 坐标的偏移量不一样。

可以使用一个数组把每一步的偏移量存起来。

如此一来，无论多少个方向，只需要一个 for 循环和一个 if 就可以做完了。




```
int dtx[4]={-1,1,0,0};
int dty[4]={0,0,-1,1};

void dfs(int now_x,int now_y){
    if(now_x==end_x&&now_y==end_y){  //走到终点，方案数+1
        count_of_ans++;
        return;
    }
    int new_x,new_y;
    for(int i=0;i<4;i++){
        new_x=now_x+dtx[i];
        new_y=now_y+dty[i];
        if(new_x>=0&&new_y>=0&&new_x<n&&new_y<m&&Map[new_x][new_y]!='#'&&!vis[new_x][new_y]){
            vis[new_x][new_y]=true;
            dfs(new_x,new_y);
            vis[new_x][new_y]=false;
        }
    }
}
```

福州大学
FUZHOU UNIVERSITY

广 度 优 先 搜 索

走 迷 宫 问 题 （ 改 ）

输入一个 n*m 的迷宫和它起点与终点的坐标，输出走出这个迷宫的最短路径长度。

假设迷宫地图由’.’和’#’组成，其中’.’代表空地，‘#’代表墙。

1 <= n,m <= 1000

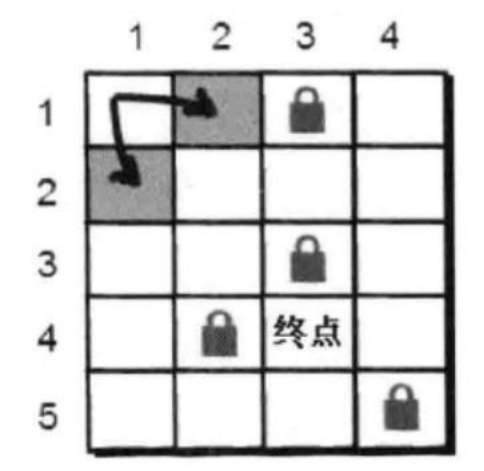
Source: 编的

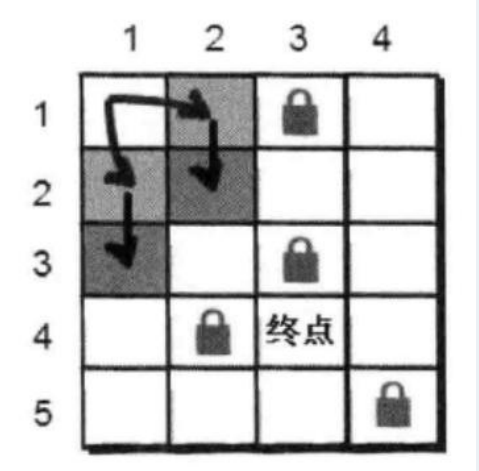
走 迷 宫 问 题 （ 改 ）

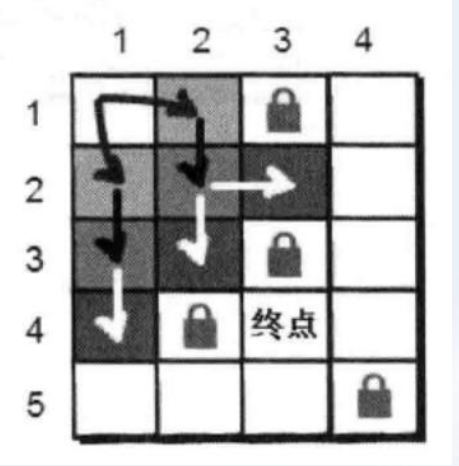
如果还是用 DFS，由于 n, m 的范围很大，会导致超时。

注意到我们只是要找最短路径，那么回溯的操作可能就不需要了。

更具体的，我们可以尝试同时搜索多条路径，最先搜到迷宫终点的就是最短路径。







走 迷 宫 问 题 （ 改 ）

我们仔细分析这个过程：

1. 对于当前的一个点，同时拓展出它能走到的几个点；
2. 把它能拓展出的这几个点存起来；
3. 在存的这些点中再选一个点，重复这些步骤，直到拓展出终点。

选择一个点后拓展的操作跟深度优先搜索中的差不多一样，就不多赘述了。

难点在于选择什么数据结构把待拓展的点存起来。

我们进一步分析发现，先存进来的点要先被拓展，是要一个先进先出的数据结构。

因此我们选用队列来进行维护。

走 迷 宫 问 题 （ 改 ）

分析完成，现在考虑代码实现。

队列中要存的东西：

- 当前格子的坐标(x, y)
- 当前状态已经走了几步 step

开始搜索时把起点入队。

搜索过程中，从队首取出状态进行拓展，产生新的状态，检测新状态的合法性后加入队尾。

最后，从队首取出的第一个坐标是终点的状态用的步数就是最短路径长度。

```
head=tail=0;  //队列置空

tail++;  //起点入队
que[tail].x=start_x;
que[tail].y=start_y;
que[tail].step=0;
vis[start_x][start_y]=true;
//vis 与之前提到的 vis 作用相同：记忆化，防止走回头路

while(head!=tail){  //队列不为空，就不断循环
    head++;  //移动队首指针,指向一个未拓展的新状态
    if(que[head].x==end_x&&que[head].y==end_y){
        printf("%d\n",que[head].step);
        return 0;
    }
    for(int i=0;i<4;i++){
        new_x=que[head].x+dtx[i];
        new_y=que[head].y+dty[i];  //产生一个新坐标
        if(new_x>=0&&new_y>=0&&new_x<n&&new_y<m&&Map[new_x][new_y]!='#'&&!vis[new_x][new_y]){
            //新坐标合理且没拓展过，入队
            tail++;
            que[tail].x=new_x;
            que[tail].y=new_y;
            que[tail].step=que[head].step+1;
            vis[new_x][new_y]=true;
        }
    }
}
```

走 迷 宫 问 题 （ 改 ）

回顾刚刚的过程：

- 从队列队首取出一个状态，拓展
- 从队列队尾存入合理的新状态
- 重复上述两步直到边界

这个过程就是广度优先搜索

（Breadth First Search，简称BFS）

```
head=tail=0;    //队列置空

//初始状态入队

while(head!=tail){  //队列不为空则一直循环
    head++;     //移动队首指针，取出一个状态
    if(/*新状态到达边界*/){
        //输出或者记录答案
        break;
    }
    //拓展队首的状态，产生出一些新的状态
    if(/*这个新状态合理*/){
        tail++;  //移动队尾指针
        //从队尾存入新状态
    }
}
```

水　坑　计　数　问　题

给出一个 n*m 的字符矩阵，表示农田积水的情况。其中‘.‘代表田地，’W’代表水坑。每个格子与周围八个方向相连，相连一起水坑的算作一个水坑。试统计农田中水坑的个数。

2 <= N,M <= 100

Source: Luogu P1596

输入输出样例

输入 #1 复制

```
10 12
W........WW.
.WWW.....WWW
....WW...WW.
.........WW.
.........W..
..W......W..
.W.W.....WW.
W.W.W.....W.
.W.W......W.
..W.......W.
```

输出 #1 复制

```
3
```

福州大学 FUZHOU UNIVERSITY

水　坑　计　数　问　题

对于矩阵中的每个点，如果它是水坑，就以它为起点开始 BFS。通过 BFS，把这个水坑和它相连的所以水坑都标记访问过，记为一整个大水坑。搜完这个大水坑后，再去找新的水坑，进行同样的操作。

最后统计有多少个大水坑即可。

```
int main(){
    scanf("%d%d",&n,&m);
    for(int i = 0 ; i < n ; i++)
        for(int j = 0 ; j < m ; j++)
            scanf(" %c", &mp[i][j]);
    for(int i = 0 ; i < n ; i++){
        for(int j = 0 ; j < m ; j++){
            if(mp[i][j] == 'W' && !vis[i][j]){
                cnt++;
                vis[i][j] = true;
                bfs(i,j);
            }
        }
    }
    printf("%d",cnt);
    return 0;
}

```

福州大学
FUZHOU UNIVERSITY

THANKS