《初级算法与数据结构》

数据结构-搜索算法



深度优先搜索与回溯

深度优先搜索

书本P186页

在之前的章节介绍了暴力枚举策略，将**所有可能的情况**都枚举一遍以**获得最优解**，但是枚举全部元素的效率如同愚公移山，无法应付**数据范围稍大**的情形。

本章将在暴力枚举的基础上介绍搜索算法，包括**深度优先搜索和广度优先搜索**，**从起点开始，逐渐扩大寻找范围**，直到找到需要的答案为止。

严格来说，搜索算法也算是一种暴力枚举策略，但是其算法特性决定了效率比直接的枚举所有答案要高，因为搜索可以**跳过一些无效状态**，降低问题规模。

题目背景【深基14.例1】四阶数独OJ:B0201

数独是一种经典的逻辑游戏，玩家需要根据已有数字填充空白格子，满足特定的排列规则。本题研究的是四阶数独的合法解的数量问题。

要求：给定一个完全空白的 4×4 网格，要求填写数字 1 到 4，使得以下条件同时满足：

1. 每一行必须包含 1，2，3，4 各一次（无重复）。
2. 每一列必须包含 1，2，3，4 各一次（无重复）。
3. 每个 2×2 子区域（共4个）必须包含 1，2，3，4 各一次（无重复）。

（子区域划分：将4×4网格均分为左上、右上、左下、右下四个2×2的小方块）

求所有可能的合法填写方法的总数。

解法一：面对一个 4×4（共 16 个格子）的空白数独，最简单的解法是什么？

就是写 16 层嵌套的 `for` 循环！

- 第 1 个格子：试 1、2、3、4
- 第 2 个格子：试 1、2、3、4
- ……
- 第 16 个格子：试 1、2、3、4

全填满之后，再去检查行、列、小方块有没有重复。

计算代价： 每个格子 4 种选择，16 个格子总共是 4^{16} 种情况。 4^{16} 大约等于 **42 亿**！

计算机为了算这一个迷你数独，要足足判断 42 亿次。这种方式非常低效，不可行。

解法二：既然每一行都必须是由 1、2、3、4 组成，且不能重复，那我们干嘛要让第一行出现 1，1，1，1 这种明显违规的排列呢？

我们可以把“按格子填”改成“**按行填**”。 1 到 4 的全排列（比如 1234，1243，1324...）一共有 $4! = 24$ 种。 既然有 4 行，我们只需要写 4 层 for 循环，去组合这 4 行的排列：

- 第 1，2，3，4 行：**24 种选择**

计算代价： 总情况数降到了 $24 \times 24 \times 24 \times 24 = 331{,}776$ 种。 33 万次计算，计算机大概零点几秒就能跑完。

这种解法完美了吗？ 如果你把问题放大到 9 阶数独，每行是 9 的全排列（36 万种），9 行组合起来就是 $362{,}880^9$，依然是个天文数字！**解法 2 依然存在巨大的资源浪费。**

解法二的愚蠢之处

模拟解法二的运行： 假设程序正在运行解法二，此时：

- 第 1 行填了：`[1, 2, 3, 4]`
- 第 2 行准备填：`[1, 3, 4, 2]`

看一眼这两行，发现了什么问题？ **第一列出现了两个 1**，已经违规了！

但是，解法 2 并不会发现违规了，它依然会傻乎乎地去把第 3 行的 24 种可能、第 4 行的 24 种可能全部生成出来，拼成完整的 4×4 矩阵，然后才去检查。 也就是说，仅仅因为前两行这个明显的错误，程序在后面做了 $24 \times 24 = 576$ 次**毫无意义的拼凑和检查**。

思考： 聪明人会怎么做？

一旦发现某一步已经错了，立刻停手，绝不继续往下走，赶紧换一种填法！

而在算法中，这种提前发现错误并中断后续操作的思路，就叫做**剪枝**。

我们现在有了要随时检查，及时止损的策略。但问题来了：`for` 循环非常死板，写了 16 层就是 16 层，很难做到进退自如。我们需要一种更灵活的框架。

深度优先搜索

递归枚举（DFS）的逻辑思想：

我们不用循环，而是写一个递归函数 `void dfs(当前填第几个空)`。流程可以想象成走迷宫：

1. **走一步看一步：**站在第 1 个空格，从 1~4 逐个尝试填。
2. **立刻检查：**看看当前的行、列、小方块里有没有刚刚填写的数字。
3. **合法就深入（深度优先）：**如果没有冲突，说明这个填的这个数字暂时没问题！我把它填进去（保存现场），然后进入下一个空格 `dfs(2)`。
4. **死胡同就回头（回溯）：**假如我在第 5 个空格试了 1、2、3、4 发现**全都冲突**了，怎么办？这说明我在**前面几步走错路**了！此时，我不能强行往下走，而是要退回到上一个空格，把之前填的数字擦掉（恢复现场），**换一个数字重新试。**

通过这种“**填数字 → 检查 → 不行就擦掉重填**”的递归机制，我们就避开了所有的无效状态。这种一般化的枚举框架，就是**深度优先搜索与回溯法。**

深度优先搜索

既然逻辑已经理顺了，现在就把这些想法变成一行行能跑的代码。

首先要解决一个最基本的问题：**为什么非得用函数，而且是自己调自己的递归函数?**

知识回顾：栈不仅仅是一个数据结构。它其实是一种**暂时挂起、等待时机的逻辑模型**。它让我们在面对复杂、嵌套、或者需要回头看的任务时，能够不慌不忙，一次只专注处理眼前**最紧急**的那件事。

系统栈： 当函数调用另一个函数时，计算机会把当前的进度和相关信息**压栈**，等结束了再**弹栈**回来继续干活。

递归函数 void dfs(..)是如何对应道栈的暂时挂起、等待时机的逻辑模型的?

在普通的 for 循环里，代码是一口气执行到底的，没有停留。

但在递归中，当 dfs(k) 执行到 dfs(k + 1) 这一行时，**dfs(k) 会暂时停工。**

- **挂起：** 假设你现在是 dfs(1)，你刚刚把数字 1 填入第一个格子。接下来你调用了 dfs(2)。此时，dfs(1) 的这个函数并没有结束，它的 for 循环停在了 num = 1 这一步，不再往下走了。
- **本质：** 计算机把 dfs(1) 当前所有的数据（当前在第几格、填了什么数字、接下来还要试哪些数字）打包放进了栈里。任务干了一半，要准备填第二个格子了，那我就先搁置在这里等着你的结果。

递归函数 `void dfs(..)`是如何对应道栈的暂时挂起、等待时机的逻辑模型的?

什么是等待时机?

被挂起的 `dfs(1)` 待在栈里干什么呢？它在**等待**。

- 它在等什么？ 它在等 `dfs(2)` 给它一个明确的结局。
- 因为 `dfs(1)` 填入数字 `1` 这个动作，到底是通向胜利的大道，还是通向死胡同？`dfs(1)` 自己是不知道的。它必须等待后续的格子尝试后返回到自己这里，才能得到答案。
- 这种“**我先做一个初步决定，然后静静等待后续发展**”的模式，就是**等待时机**。

为什么非得用函数，而且是自己调自己的递归函数?

回答：因为递归，以及它背后的**函数调用栈**，天然就是为这种“保存现场、等待时机”的逻辑模型而生的!

- **流程模拟：** `dfs(1)` 开始执行：
  - 它的目标是填充第1个空格。
  - 它会进入一个循环，尝试数字1到4。
  - 尝试数字 `1`： 将 `1` 填入第1格。检查发现合法。
  - 下一步是什么？ 既然第1格暂时填好了，程序现在面临的新问题是：“在第1格是1的前提下，如何解决剩下的15个格子？”
  - 这个问题，本质上就是从第2个格子开始的、一个规模稍小但结构完全相同的数独子问题。
  - 因此，就是让 `dfs(1)` 调用 `dfs(2)` 去解决这个新问题。这就是递归调用。

- `dfs(2)` 开始执行：
  - 它的目标是填充第2个空格。
  - 它不关心 `dfs(1)` 内部的循环状态，它只知道当前的棋盘（第1格是1）是合法的。
  - 它也进入一个循环，尝试数字1到4。
  - 尝试数字 `1`： 检查发现与第1格冲突，不合法。
  - 尝试数字 `2`： 检查发现合法。
  - 下一步？ 同样，它调用 `dfs(3)` 去解决后续的问题。

这个 `dfs(k)` 调用 `dfs(k+1)` 的链条会一直持续下去

现在，假设程序执行到了 `dfs(k)`，它正在为第 `k` 个空格寻找合适的数字。

- `dfs(k)` 的循环开始：
  - 尝试数字`1` -> 冲突；尝试数字`2` -> 冲突。尝试数字`3` -> 冲突；尝试数字`4` -> 冲突。

此时，`dfs(k)` 的循环执行完毕，但没有为第 `k` 格找到任何一个合法的数字。这意味着，**在当前棋盘状态下，尝试了所有的路径都失败**，故第 `k` 格无解。

- `dfs(k)` 必须返回： `dfs(k)` 的任务结束了，它会执行 `return`，并将执行控制权交还给它的调用者——`dfs(k-1)`。

dfs(k-1) 的状态：

当 dfs(k-1) 调用 dfs(k) 时，dfs(k-1) 的执行是暂停的。它暂停在它的 for 循环内部，例如，它当时可能刚刚为第 k-1 格填入了数字 X，然后就调用了 dfs(k)。

return 之后的效果：

- 现在 dfs(k) 返回了，dfs(k-1) 从暂停处恢复执行。
- dfs(k-1) 接收到 dfs(k) 的返回。这说明它当初为 k-1 格填入 X 这个决策，需要重新更换填入的数字，因为它导致了后续的格子无解，需要回退到上一个决策点。
- dfs(k-1) 的代码继续执行，在循环的下一轮开始前，它必须清除自己刚才做的决策，即把第 k-1 格的 X 擦掉，以便程序更换其他填入的数字。
- 然后，dfs(k-1) 的 for 循环继续，尝试为第 k-1 格填入下一个数字 X+1。

总结：

- 递归调用 `dfs(k+1)`：代表基于当前成功的决策，向问题的下一个阶段前进。
- 函数返回 `return`：代表当前阶段的**尝试失败/找到了一个解**，需要回退到上一个决策点。
- 递归函数调用栈：是这个机制的底层实现。它自动保存了每个暂停的函数（如 `dfs(k-1)`）的状态，包括它的局部变量和它在代码中执行到的位置（比如 `for` 循环的哪一轮）。当被调用的函数（`dfs(k)`）返回时，调用栈能确保 `dfs(k-1)` 可以从正确的位置继续执行。

因此，递归将 **前进探索 和 失败回退** 这两个核心操作，映射到了 **函数调用 和 函数返回** 这两个编程构造上，使得复杂的搜索与回溯逻辑变得结构清晰且易于实现。

我们就可以根据思路，手动来模拟一下执行的流程

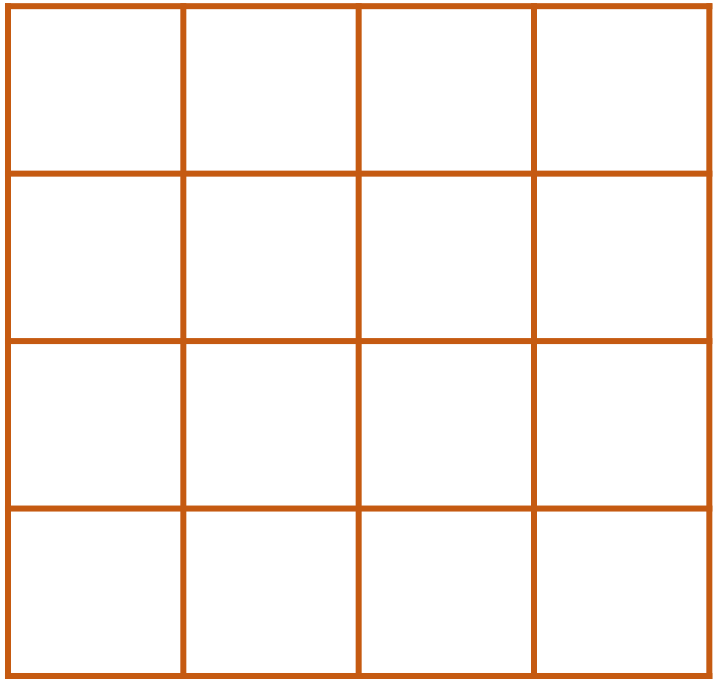
（书本P189页 图14-4）

1．**走一步看一步**：站在第 1 个空格，从 1 开始填。

2．**立刻检查**：看看当前的行、列、小方块里有没有刚刚填写的数字。

3．**合法就深入（深度优先）**：如果没有冲突，把它填进去（保存现场），然后进入下一个空格。

4．**死胡同就回头（回溯）**：退回到上一个空格，把之前填的数字擦掉（恢复现场），**换一个数字重新试**。



解决了为什么要用递归函数的问题，现在就实际来书写代码。

准备工作：定义棋盘和递归函数框架

首先，我们需要一个 4x4 的棋盘、辅助变量和一个 dfs 函数的基本框架。

```
vector<vector<int>> grid(4, vector<int>(4, 0)); // 的4x4棋盘，0代表空格
int count = 0;  // 用来统计解的总数

// 检查函数，判断在(x, y)位置填num是否合法（这部分假设已写好）
bool check(int x, int y, int num);

// dfs函数，参数k代表当前要填的是第k个空格（从0到15）
void dfs(int k) {

}
```

递归的结束条件与坐标转化

```
// dfs函数，参数k代表“当前要填的是第k个空格” （从0到15）
void dfs(int k) {
    // 递归的出口：如果k等于16，说明所有格子都填完了
    if (k == 16) {
        count++; // 找到了一个完整的合法解
        return;  // 返回，结束这条探索路径
    }
    // 将第几个空格 k 转换为二维的(x, y)坐标
    int x = k / 4;
    int y = k % 4;
```

	0	1	2	3
0	0	1	2	3
1	4	5	6	7
2	8	9	10	11
3	12	13	14	15

步骤一：走一步看一步

逻辑思想：站在第 1 个空格，我试着填 1。

这个思想的本质是**遍历所有可能性**。对于当前这个空格，我们需要尝试填入数字1，然后尝试2，然后3，然后4。

对应代码：一个 for 循环就能实现这个逐一尝试的过程。

```
void dfs(int k) {
    // ... 前面的代码 ...
    // 步骤 1: 走一步看一步 —— 遍历所有可能的数字
    for (int num = 1; num <= 4; ++num) {
        // 在这里，我们依次试着填入num

    }
}
```

步骤二：立即检查

逻辑思想： 看看当前的行、列、小方块里有没有 刚刚我选择的数字。

在决定是否要将数字固定下来并前进之前，必须先检查这一步是否合法。

对应代码： 一个 `if` 语句，调用我们预先写好的 `check` 函数。

```
void dfs(int k) {
    // ...
    for (int num = 1; num <= 4; ++num) {
        // 步骤 2: 立刻检查 — 判断填入num是否合法
        if (check(x, y, num)) {
            // 只有当check函数返回true时，才说明这一步是合法的
        }
    }
}
```

步骤三：合法就深入

逻辑思想: 如果没有冲突，说明可以该数字暂时没问题！把它填进去，然后进入下一个空 `dfs(2)`。

一旦当前步骤确认合法，我们就要基于这个临时的决定，去解决下一个子问题。

对应代码: 这包含两个动作：

1. 修改棋盘状态 (`grid[x][y] = num;`)，这对应“保存现场”。
2. 递归调用 (`dfs(k + 1);`)，这对应“进入下一个空格”。

```
if (check(x, y, num)) {
    // 把数字实际填入棋盘
    grid[x][y] = num;
    // 深入，去处理下一个空格 (k+1)
    dfs(k + 1);
```

步骤四：死胡同就回头（回溯）

逻辑思想: 退回到上一个空格，把之前填的数字擦掉（恢复现场），换一个数字重新试。

当`dfs(k + 1)`的探索全部结束后（无论是找到了解，还是走进了死胡同），程序会返回到`dfs(k)`暂停的地方。此时，为了不影响`dfs(k)`对其他数字的尝试，我们必须**撤销之前的决定**。

```
if (check(x, y, num)) {
    grid[x][y] = num; // 保存现场
    dfs(k + 1);       // 深入

    // 当dfs(k+1)返回，意味着从那一步开始的所有路径都探索完了。
    // 我们必须恢复现场，把格子变回空格，以便for循环下一次尝试新的num。
    grid[x][y] = 0;
}
```

代码回顾

```
void dfs(int k) {
    // ... 递归的出口省略
    for (int num = 1; num <= 4; ++num) {
        if (check(x, y, num)) {
            grid[x][y] = num;  // 保存现场

            dfs(k + 1); // 深入

            // 当dfs(k+1)返回，意味着从那一步开始的所有路径都探索完了。
            // 恢复现场，把格子变回空格，以便for循环下一次尝试新的num。
            grid[x][y] = 0;
        }
    }
    // 当for循环结束，说明为第k格尝试了所有数字都无法最终构成解
    // dfs(k)函数自然结束，自动“回溯”到dfs(k-1)。
}
```

最后还剩下，如何完成 check 函数

检查行

逻辑： num 不能与 (x, y) 所在行的任何其他数字相同。

这意味着我们要遍历第 x 行的所有格子，看看 num 是否已经存在。

- 行号 x 是固定的。我们需要循环改变的是列号。
- 如何遍历？ 使用一个 for 循环，让一个变量（比如 i）从 0 遍历到 3，代表每一列。

```
// 检查函数，判断在(x, y)位置填num是否合法
bool check(int x, int y, int num) {
    // 1. 循环检查行
    for (int i = 0; i < 4; ++i)
        if (grid[x][i] == num) return false;
```

检查列

逻辑： `num` 不能与 `(x, y)` 所在列的任何其他数字相同。

这意味着我们要遍历第 `x` 行的所有格子，看看 `num` 是否已经存在。

- 列号 `y` 是固定的。我们需要循环改变的是行号。
- 同样使用一个 `for` 循环，让一个变量从 `0` 遍历到 `3`，代表每一行。

```
// ...
// 2. 循环检查列
for (int i = 0; i < 4; ++i)
    if (grid[i][y] == num) return false;
```

检查2x2子区域

逻辑： 不能与当前坐标所在的2x2小方块内的任何其他数字相同。

我们如何根据 `(x, y)` 快速确定它属于哪个小方块，并只遍历那个方块内的4个格子?

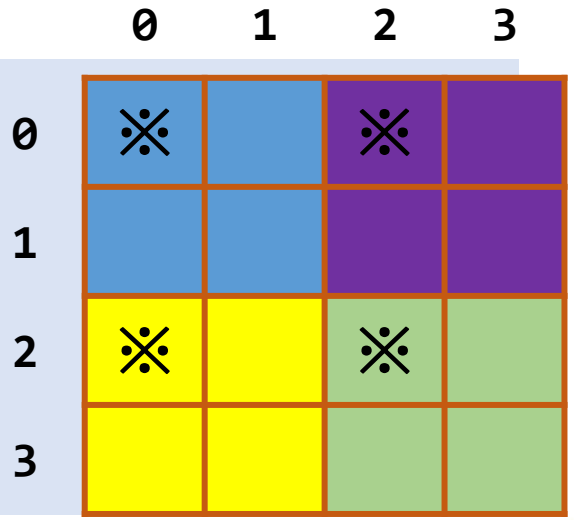
解决方法： 找到该方块的**左上角顶点**坐标 `(startX, startY)`。

如何计算 `startX` 和 `startY`?

我们观察到：

- 对于第0、1行，左上角的行号总是0。
- 对于第2、3行，左上角的行号总是2。

这个规律可以用一个整数除法公式来表示：`int startX = (x / 2) * 2;`



是怎么得来：（x / 2）* 2 的呢

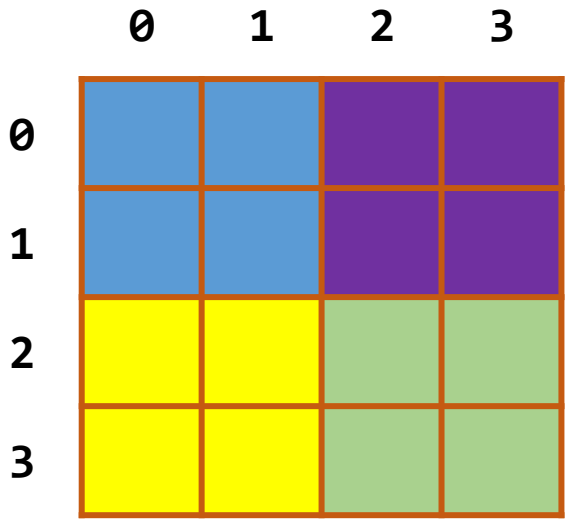
在 4x4 网格中需要切割为 2x2 的小方块，我们可以把行和列都看成是两两一组的的：

行分组：

- 第 0 行和第 1 行属于第 0 组；
- 第 2 行和第 3 行属于第 1 组。

列分组：

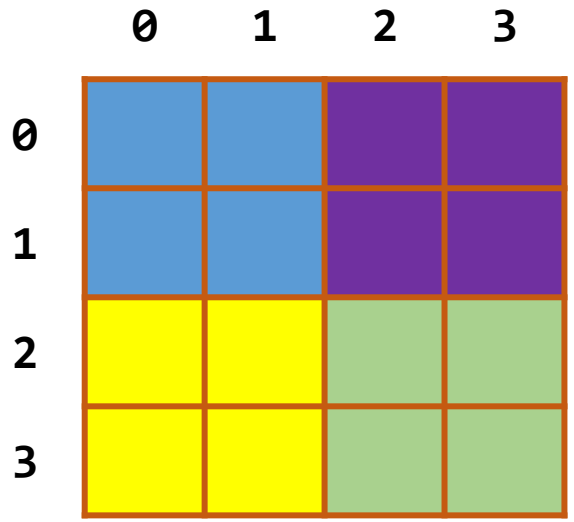
- 第 0 列和第 1 列属于第 0 组；
- 第 2 列和第 3 列属于第 1 组。



整数除法有一个特性就是可以用来分组。

- 当 row 是 0 时：0 / 2 = 0（第 0 组）
- 当 row 是 1 时：1 / 2 = 0（第 0 组）
- 当 row 是 2 时：2 / 2 = 1（第 1 组）
- 当 row 是 3 时：3 / 2 = 1（第 1 组）

row / 2 的结果告诉了我们当前行属于哪个组（0 或 1）。

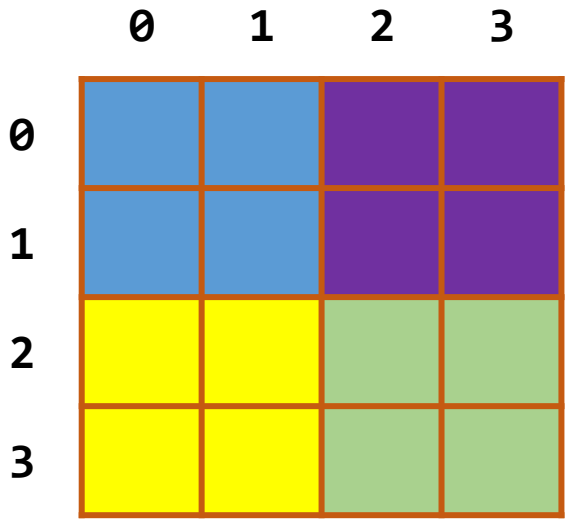


现在需要把这个组编号转换回实际的起始行号，因为一组代表着两行，所以这个关系是一个乘法：起始行号 = 组编号 * 2。

同理，`int startY = (y / 2) * 2;` 也能计算出对应左上角的列号。

```
bool check(int x, int y, int num) {
    // 检查行...检查列...
    // 检查所在的2x2子区域
    int startX = (x / 2) * 2;
    int startY = (y / 2) * 2;

    for (int i = startX; i < startX + 2; ++i) {
        for (int j = startY; j < startY + 2; ++j) {
            if (grid[i][j] == num) {
                return false; // 说明num已经存在于这个方格
            }
        }
    }
    return true; // 所有检查都通过了
}
```



我们还可以引入三个桶数组的方法，用来实时记录每个区域的数字占用情况来实现check函数，效率会稍高一些，但是会导致代码更加复杂。

1. `b1` (行状态): `bool b1[4][4];`
   - `b1[i][num]` 表示：在第 `i` 行，数字 `num` 是否已经被占用了。
   - 例如，`b1[0][3] = true` 的意思是：第0行已经有数字3了。
2. `b2` (列状态): `bool b2[4][4];`
   - `b2[j][num]` 表示：在第 `j` 列，数字 `num` 是否已经被占用了。
   - 例如，`b2[2][1] = true` 的意思是：第2列已经有数字1了。
3. `b3` (子区域状态): `bool b3[4][4];`
   - `b3[k][num]` 表示：在第 `k` 个2x2子区域，数字 `num` 是否已经被占用了。
   - 例如，`b3[3][4] = true` 的意思是：右下角的那个子区域已经有数字4了。

如果使用这种方法，那么dfs函数内部也需要发生改变！

```
// “保存现场”：不仅要填数字，还要更新三个桶数组！
b1[x][num] = true;
b2[y][num] = true;
b3[block_idx][num] = true;
grid[x][y] = num;
// 外层提前已经判断合法，深入
dfs(k + 1);

// “恢复现场”（回溯）：把所有状态都恢复回去！
b1[x][num] = false;
b2[y][num] = false;
b3[block_idx][num] = false;
grid[x][y] = 0;
```

至此，我们已经将所有逻辑思想转换为了代码，它们组合在一起，就构成了一个完整的、带有回溯功能的深度优先搜索。

我们可以把这道题目的精髓总结为一套框架。

一、从暴力枚举到启发式剪枝

- 问题抽象：
  - 在 4×4 矩阵中填充数字，寻找满足行、列、块约束的所有合法解。
  - 本质是在有限的状态空间中寻找目标状态。
- 算法复杂度演变：
  - **盲目搜索：**每个格子独立尝试，时间复杂度高达 $O(4^{16})$。
  - **排列组合搜索：**按行生成全排列，时间复杂度降至 $O(24^4)$，但存在大量无意义的后须匹配。
  - **深度优先搜索配合剪枝：**引入“**及时止损**”策略，一旦当前前缀状态不合法，立即截断后续搜索分支，极大收敛了搜索空间。

二、深度优先搜索（DFS）流程可概括为四个步骤：

1. **状态扩展** ：确定当前决策节点，枚举所有可能的候选动作（如尝试数字 1-4）。
2. **约束满足性检查**：在执行动作前，验证动作的合法性。
3. **状态转移与深入**：

   保存现场：将当前合法的动作更新至全局状态（修改棋盘、更新标记）。

   递归调用：将问题规模缩小，调用进入下一层计算。
4. **状态回退**：

   恢复现场：在子问题计算结束后，必须撤销当前动作对全局状态的修改(如决策填写了 3，而决策为 3 的子问题计算结束了，那就需要将其撤销)，以保证下一次同级枚举的初始状态纯净。

三、递归的本质

为什么回溯法必须依赖递归函数？其底层逻辑是对**系统调用栈** 的利用。

挂起 ：

- 当函数 $f(k)$ 调用 $f(k+1)$ 时，$f(k)$ 并未销毁，而是将其局部变量上下文（当前枚举的进度、节点状态）压入系统栈。
- 将全局的调度问题分解转化为局部的问题，一次只处理当前层级的决策。

恢复 ：

- 当 $f(k+1)$ 执行完毕（弹栈），系统将控制权及当时的上下文交还给 $f(k)$。
- 实现了逻辑上的时光倒流，使得算法能够完美回到做出决策的那个分叉口，进行状态清理（恢复现场）并尝试下一条分支。

深度优先搜索

DFS 回溯代码模板在实现上，会呈现出极其严格的**对称性**：

递归调用的上方是对状态的修改，下方必定是对状态的逆向撤销。

```
void dfs(int step) {
    if (到达目标状态) { 记录解; return; }
    for (遍历所有候选选项) {
        if (当前选项满足约束条件) {
            // 1. 压栈前的准备：保存现场
            update_state(选项);
            // 2. 状态转移：进入子问题
            dfs(step + 1);
            // 3. 弹栈后的清理：恢复现场
            update_state(选项);
        }
    }
}
```

深度优先搜索

八皇后问题（书本P190页 例14-2）

【题目描述】（OJ:B0202）

在一个 $N \times N$ 的国际象棋棋盘上，我们要放置 N 个皇后。 皇后的攻击力非常高，她的攻击范围呈**米**字型：**同一行、同一列、以及两条斜线（左上到右下、右上到左下）**上的所有格子。

【任务要求】

1．放置 N 个皇后，使得她们互不攻击（即任何两个皇后都不能处于同一行、同一列或同一斜线上）。

2．给定棋盘大小 N（$N \leq 13$），求出一共有多少种合法的放置方案。

3．按字典序输出前 3 种具体的放置方法（用每一行皇后所在的列号表示）。

解题思路分析：

很多人一看到 $N \times N$ 的棋盘，第一反应是：建一个二维数组 `board[13][13]`，然后写两层 `for` 循环，每个格子去试放不放皇后。

但是规则里有**同一行只能放一个皇后。** 既然如此，我们根本不需要去遍历棋盘上的每一个格子！我们只需要给每一行**分配**一个皇后就行了。

于是我们的任务从**在棋盘上找空位放置皇后**，变成了**决定第 1 行的皇后坐在第几列、第 2 行的皇后坐在第几列……**

而我们只需要一个一维桶数组 `a[row] = col`，它代表着 row 行的皇后坐在col列，就足以记录整个棋盘的状态！比如 `a[1] = 3`，就代表第 1 行的皇后坐在了第 3 列。

这就把一个复杂的二维平面搜索，变成了在一维上**逐行推进搜索**。

我们的递归函数 `dfs(row)` 的含义也呼之欲出：

当前正在为第 `row` 行的皇后挑选列号。

在确定了逐行推进（`dfs(row)`）后，我们准备在第 `col` 列放下皇后。放下之前，必须**立刻检查**（回溯四步法之二）。

- 检查同列冲突：简单！搞一个布尔数组 `b[col]`，它的含义是如果 `b[col] = true`，那么代表着 `col` 这一列已经有皇后了，反之则没有皇后。
- 检查斜线冲突：难道要写 `for` 循环，左上右下地去扫描棋盘吗？太慢了！

我们不扫描，我们可以通过坐标的数学规律来实现 $O(1)$ 的瞬间判断！

右上 → 左下的斜线： 你观察一下这条线上的坐标：

发现了什么？它们的 **行号 + 列号** 永远等于一个常数！

解决方案：开一个数组 `c[row + col]`。只要在这个斜线上放了皇后，就把 `c[5] = true`。

下次谁想放在坐标和为 5 的地方，就先查看一下 `c[5]` 发现结果等于 `true` ，也就是已经被占用了！

	0	1	2	3
0	0,0	0,1	0,2	0,3
1	1,0	1,1	1,2	1,3
2	2,0	2,1	2,2	2,3
3	3,0	3,1	3,2	3,3

我们不扫描，我们可以通过坐标的数学规律来实现 $O(1)$ 的瞬间判断！

左上 → 右下的斜线： 你再观察这条线上的坐标

规律是：它们的 **行号 - 列号** 永远等于一个常数！

但是，行号减列号可能是负数（比如 $1-2=-1$），而数组下标不能是负数。怎么办？

解决方案：那我们统一加上一个常数 N！不让他变为负数不就可以了，让下标变成 `row - col + N`。

我们就开一个数组 `d[row - col + N]`。只要这条线上有皇后，对应的`d[row - col + N]` 就是 `true`。

	0	1	2	3
0	0,0	0,1	0,2	0,3
1	1,0	1,1	1,2	1,3
2	2,0	2,1	2,2	2,3
3	3,0	3,1	3,2	3,3

书写 dfs() 函数前的准备工作：

```
int n;              // 棋盘大小，最多为 13 x 13
int ans = 0;        // 记录总方案数
int a[20];          // 记录答案：a[行号] = 列号

// 三个桶数组，分别是列，副对角线，主对角线
bool b[20];         // 监控列：b[列号]
bool c[40];         // 监控副对角线：c[行 + 列]
bool d[40];         // 监控主对角线：d[行 - 列 + n]
```

递归函数出口与答案输出(本次代码 row, col 从 1 开始):

```
// 递归函数：为第 row 行的皇后选列的座位
void dfs(int row) {
    // 递归出口：如果 1 到 n 行都安排满了，说明棋盘已经放完所有的皇后了！
    if (row > n) {
        ans++; // 找到一种合法方案，总数 +1
        if (ans <= 3) {  // 题目要求：打印前 3 种方案
            for (int i = 1; i <= n; i++)
                cout << a[i] << " "; // a[i] 代表着 i 行的皇后在a[i]列
            cout << endl;
        }
        return; // 功成身退，弹出回溯，尝试其他可能
    }
```

从第一列开始，面对 1 到 n 列，挨个尝试放置皇后

面对第 i 列尝试放置时，需要检查此位置是否合法

```
// 走一步看一步：站在第 row 行，面对 1 到 n 列，挨个尝试
for (int col = 1; col <= n; col++) {
    // 立刻检查：看三个标记数组是不是全是 false
    if (!b[col] && !c[row + col] && !d[row - col + n]) {
```

递归与回溯：

```
// 位置合法，那就深入
a[row] = col; // 记录当前行的决定

// 放了皇后，对应的标记数组标记为 true
b[col] = c[row + col] = d[row - col + n] = true;

// 挂起当前状态，带着现有的布局，给下一行安排皇后
dfs(row + 1);

// 当 dfs(row + 1) 返回时，说明后面的路走完了（要么成功，要么死胡同）
// 我们必须撤销刚才的决定，准备 for 循环尝试在换个位置在下一列放皇后
b[col] = c[row + col] = d[row - col + n] = false;
```

最后只需要在主函数中调用 dfs 即可

```
int main() {
    cin >> n;
    dfs(1); // 游戏开始：从第 1 行的皇后开始安排
    cout << ans << endl; // 最后输出总方案数
    return 0;
}
```

总结：

我们在填入皇后时标上 `true`（**保存现场**），然后让`dfs(row + 1)` 走向未来去帮我们在当前选择下，去搜索所有可能的解。

当未来的搜索结束，回到这一行时，我们做的第一件事就是把它们改成 `false`（**恢复现场**）。

如果不写这句 `false` 会怎样?

哪怕你已经退回到了上一行准备换个策略，标记数组依然以为那个位置有皇后。这就好比你搬出了酒店，却没有退房，后面的人永远住不进来。

总结：

八皇后问题和四阶数独，本质上是在用同一个思想。

数独是在空格子里填数字，八皇后是在行里填列号。 它们的骨架全都是**试探** → **占座** → **递归** → **退座**。

这告诉我们：只要能找出问题中需要尝试填写什么，并设计出检查函数（怎么判断冲突），然后在当前决策下，尝试递归去搜索解，无论是什么样的复杂的迷宫，DFS 都能用同一套逻辑帮解决！

例14-3 考前临时抱佛脚(书本P191页，OJ:B0203)

这次期末考试，kkksc03 需要考 4 科。因此要开始刷习题集，每科都有一个习题集，分别有 s_1, s_2, s_3, s_4 道题目，完成每道题目需要一些时间，可能不等（$A_1, A_2, \ldots, A_{s_1}$，$B_1, B_2, \ldots, B_{s_2}$，$C_1, C_2, \ldots, C_{s_3}$，$D_1, D_2, \ldots, D_{s_4}$）。

kkksc03 有一个能力，他的左右两个大脑可以同时计算 2 道不同的题目，但是仅限于同一科。因此，kkksc03 必须一科一科的复习。

他希望尽快把事情做完，所以他希望知道能够完成复习的最短时间。

每科都有一个习题集，分别有 s_1, s_2, s_3, s_4 道题目，完成每道题目需要一些时间，可能不等。他的左右两个大脑可以同时计算 2 道不同的题目，但是仅限于同一科。必须一科一科的复习。希望知道能够完成复习的最短时间。

本题目的题意可以总结为：

给定 4 个独立的数字集合。对于每个集合，要求将其分为两个子集，使得这两个子集的“**元素之和的最大值**”尽可能小。求这 4 个集合的该最小值之和。

这道题的算法可以拆解为三个关键点：

每科独立相加（独立互不干扰）

规定“必须一科一科复习”，意味着 4 个科目是完全独立的。

公式：总最短时间 = 科目1最短时间 + 科目2最短时间 + 科目3最短时间 + 科目4最短时间。

我们只需要专心写好“单科的最优解”函数，调用 4 次即可。

取最大值

因为规定可以“左右脑同时计算同一科的不同题目”，意味着面对单科的一堆题目，我们要把它们装进“左脑”和“右脑”两个桶里。

公式：单科耗时 = max(左脑做题总耗时，右脑做题总耗时)。

均分逼近（让两边尽可能相等）

为了让 max(左脑，右脑) 尽可能小，最好的策略就是让左右脑分配到的任务重量尽可能接近（即贴近该科总耗时的一半），谁也不能闲着。

现在再回看题意总结，就会比较清晰了：

给定 4 个独立的数字集合。对于每个集合，要求将其分为两个子集，使得这两个子集的“**元素之和的最大值**”尽可能小。求这 4 个集合的该最小值之和。

思路分析：

科目与科目之间是**隔离**的。所以，这道题虽然给了 4 科的数据，但我们只需要写一个函数，专门用来计算“某一科的最短复习时间”。然后用一个循环调用它 4 次，把这 4 次的结果加起来，就是最终答案！

现在，我们把目光盯在**第一科**上。左脑和右脑可以同时工作：

- 如果把题分配给左脑，左脑用时增加。反之右脑用时增加。
- 而总时间取决于谁更慢！ 也就是 max(左脑总用时，右脑总用时)。

目标： 把这些题分成两拨，尽量让左脑和右脑的任务**一样重**，这样总耗时最低。

套用 DFS 框架

我们怎么把题目分给左右脑呢？这不就是我们在八皇后和数独里的“走一步看一步”吗！

对于每一道题，我们只有 **2 种选择**：

- 选择 A： 交给左脑（压栈，挂起，去处理下一题）。
- 选择 B： 交给右脑（压栈，挂起，去处理下一题）。

我们可以构思一个递归函数 dfs，并且在参数里传递：当前正准备分配第 i 题的时候，左右脑的占用情况，就能更方便的来计算了。

```
// index: 现在正准备分配第几题
// sum_L: 左脑目前累计被分配了多少时间
// sum_R: 右脑目前累计被分配了多少时间
void dfs(int index, int sum_L, int sum_R) { ... }
```

dfs 逻辑：

递归出口：

如果 index 超过了这一科的总题数，说明所有题都分完了！ 此时，这一种分配方案耗时是 max(sum_L, sum_R)。我们把它和我们记录的“历史最短时间”比一比，如果更短，就更新记录。然后 return 弹栈。

```
void dfs(int index, int sum_L, int sum_R) {
 // 所有题目都已分配就退出，min_time为全局变量，记录用时最小的方案
    if (index == s) {
        min_time = min(min_time, max(sum_L, sum_R));
        return;
    }
```

dfs 逻辑：

走一步做出决策与深入(分配给左脑)：

- 我决定把第 index 题给左脑！ 怎么深入？直接调用 dfs(index + 1, sum_L + 这道题的时间, sum_R);

```
void dfs(int index, int sum_L, int sum_R) {
if (index == s) {
        min_time = min(min_time, max(sum_L, sum_R));
        return;
    }

    // 将第 index 道题分配给左脑，problem是全局数组，用于储存题目所花的时间
    dfs(index + 1, sum_L + problem[index], sum_R);
```

dfs 逻辑：

回溯与换路（分配给右脑）：

- 等上面的函数执行完弹栈回来，说明给左脑引发的所有可能都已经探索完了。 此时，我们换一条路，把这道题给右脑试试！

```
void dfs(int index, int sum_L, int sum_R) {
if (index == s) {
        min_time = min(min_time, max(sum_L, sum_R)); return;
    }
    dfs(index + 1, sum_L + problem[index], sum_R);
    // 将第 index 道题分配给右脑
    dfs(index + 1, sum_L, sum_R + problem[index]);
}
```

最终只需要在主函数中分别处理四门科目，每次调用dfs函数即可解决本问题

目前代码已经足够通过oj了，但是本题目还可以进行一些剪枝优化：

1.最优性剪枝：

如果在搜索过程中，sum_L 或 sum_R 已经超过了当前已经找到的 min_time，那么无论后续题目如何分配，最终的 max(sum_L, sum_R) 肯定会比 min_time 更大。因此，这条路不可能通向更优的解，可以直接放弃。

2.优化搜索顺序：

为了让剪枝条件尽早触发，我们可以改变搜索的顺序。如果先处理时间较长的题目，sum_L 和 sum_R 会增长得更快，从而更快地达到剪枝的阈值。

因此，一个非常有效的优化是在 DFS 开始前，将该科的所有题目按时间从大到小排序。

可以自行尝试剪枝前与剪枝后在OJ中运行时间的差异

书本上的做法：

如果我们把任务分成两堆，必然是一堆 $\leq sum/2$，另一堆 $\geq sum/2$。

既然左右脑是对称的，我们完全可以**规定**：左脑永远是拿得较少（或相等）的那一半。也就是说，左脑的任务上限就是 `sum / 2`！

目标变成了：**在不超过 `sum / 2` 的前提下，让左脑拿到的题目总耗时尽可能大（越接近 `sum / 2` 越好）。**

当我们准备把某道题给左脑时，如果 当前左脑时间 + 这道题的时间 > `sum / 2`，这就意味着打破了我们的规定，直接判定为**不合法分支，立刻剪枝！**

使用该做法就不再需要在参数里同时记录左脑和右脑的时间了。

我们只需要关注左脑！因为对于每一道题，我们的选择变成了：

- 装进左脑（前提是装得下，不超过 `sum / 2`）。
- 不装进左脑（那就意味着丢给右脑了，不管它）。

跑完这个 `dfs` 后，左脑尽了最大努力拿到了 `max_left` 的时间。

那么这科的总耗时到底是多少呢？

因为左脑拿的是较少的一半，所以右脑肯定拿了剩下较多的一半。

最终这科的耗时就是右脑的时间：`sum - max_left`。

函数原型为：

```
// index: 当前正准备选第几题; current_time: 左脑目前已累积的时间
void dfs(int index, int current_time)
```

关键代码：

```
    // 选择 1：把这道题装进，只有当加入这道题后不超过 half，才允许深入！
    if (current_time + A[index] <= half) {
        // 深入：把题装进去了，时间增加
        dfs(index + 1, current_time + A[index]);
    }

    // 选择 2：不把这道题装进左脑（相当于留给右脑）
    // 挂起当前状态，题跳过了，时间不变
    dfs(index + 1, current_time);
```

深度优先搜索

总结：DFS 回溯代码模板在实现上，会呈现出极其严格的**对称性**：

递归调用的上方是对状态的修改，下方必定是对状态的逆向撤销。

```
void dfs(int step) {
    if (到达目标状态) { 记录解; return; }
    for (遍历所有候选选项) {
        if (当前选项满足约束条件) {
            // 1. 压栈前的准备：保存现场
            update_state(选项);
            // 2. 状态转移：进入子问题
            dfs(step + 1);
            // 3. 弹栈后的清理：恢复现场
            update_state(选项);
        }
    }
}
```

广度优先搜索

广度优先搜索

深度优先搜索会优先考虑搜索的深度。形象点说，就是不找到一个答案不回头。当答案在整棵解答树中比较稀疏时，深度优先搜索可能会先陷入过深的情况，一时半会儿找不到解。有时候需要解决连通性、最短路问题时，可以考虑使用广度优先搜索。

书本P193页 例14-4 马的遍历(OJ: B0204)

有一个 $n \times m$ 的棋盘，在某个点 (x, y) 上有一个马，要求你计算出马到达棋盘上任意一个点最少要走几步。

棋盘大小最大是 400×400（共 16 万个格子），我们的目标是求最少走几步（最短路径）。

- **能用DFS吗？如果用 DFS 会发生什么？**

假设马从起点出发，认准一个方向一直走（深搜本性），绕了棋盘一大圈，花了 1000 步终于走到了终点旁边的一个格子。

- 怎么知道这 1000 步是不是最少的？
- 答案是：它不知道！DFS 为了证明这是最短的，必须把全图无数种绕路的走法全部试一遍，然后再比较出一个最小值。
- 结论： 面对求最短路径问题，DFS 极度低效，在这个数据规模下会直接超时到天荒地老。

广度优先搜索

问题： 假设你快迟到了，但你的钥匙丢在了家里，你要怎么找?

DFS 型选手：

他会一条路走到底，走进卧室，打开衣柜，翻开衣柜里的第一个抽屉，打开抽屉里的一个首饰盒，把首饰盒里的东西全倒出来……如果没找到，再退回衣柜，去找其他盒子。

但是很有可能钥匙就明晃晃地摆在客厅的茶几上，但他却在卧室抽屉里浪费了很久！

问题： 假设你快迟到了，但你的钥匙丢在了家里，你要怎么找？

BFS 型选手（广度逐层扩散）

站在客厅中央。先环顾四周，扫视所有的桌面、沙发面（**第 1 层搜索**）。如果桌面没有，再拉开所有房间最外层的抽屉（**第 2 层搜索**）。如果还没有，再去翻更深处的箱子（**第 3 层搜索**）。

广度优先搜索

有一个 $n \times m$ 的棋盘，在某个点 (x, y) 上有一个马，要求你计算出马到达棋盘上任意一个点最少要走几步。

对于这个马的遍历问题，我们也可以按照一层层去推进搜索。

在起点 (x, y) 放上一匹马。我们在它脚下的格子上写下 0。这代表到达这里需要 0 步。

- 第一次扩散： 按照“马走日”的规则，向四周最多 8 个合法位置跳跃。我们可以想象为克隆了八匹马，克隆的马跳过去之后，我们在这 8 个格子上全部写下 1。我们叫这些马为 1步马， 代表该马是跳了一次的。

- 第二次扩展： 现在棋盘上有 8 匹写着 1 的马。每个马都会再向外跳出 8 步变成2步马。

此时会出现一个问题：

如果一匹 2 步马 跳到了之前起点那个 0 步 的格子上，或者跳到了旁边刚刚踩过的 1 步格子上，怎么办？

解决办法：不走回头路！ 只要这个格子之前被人写过数字了，说明早就有人比你更快到达这里，这匹 2 步马就不应该跳在这里！只有跳到新格子上，我们才应该写下 2。

第三次扩展（以此类推）： 所有活下来的 2 步马 继续向外克隆跳跃，占领新格子，写下 3……直到整个棋盘都被填满，再也没有新格子可以跳为止。

因为我们是**严格按照 1 步、2 步、3 步的顺序，一批一批地齐头并进**的。所以，任何一个格子，只要它被一匹马**第一次**踩中，这匹马身上的步数，就 **100% 是起点到这里的最短步数**！因为本思路的路径是逐层扩散来的，不可能会有更短的路径。

而且后续哪怕有绕了远路的马再跳过来，我们会判断这个格子已经被其他马踩过了，这匹马就会直接销毁掉。

广度优先搜索

BFS可以想象成**往平静的湖面（棋盘）上扔了一块石头（起点）**。

- 第 0 层： 石头落水的中心点（马的起点），步数为 0
- 第 1 层： 水波扩散出第一圈。马跳 1 步能到达的所有格子，步数全是 1
- 第 2 层： 从第一圈的格子继续向外扩散，生成第二圈。步数全为 2

被水波第一次碰到的格子，当下这个波纹的层数，就是起点到它的最短距离！后续哪怕有其他波纹绕了一大圈再荡到这里，我们都不理它了（因为肯定更远）。这就是 BFS 解决最短路径问题的底层逻辑！

广度优先搜索

但是我们在写循环的时候计算机它一次只能处理一匹马的跳跃。它要怎么保证“第 1 批”的 8 匹马全都跳完了，才准“第 2 批”的马跳呢?

我们可以想象在排队一样，每一匹马都是逐个进行跳跃的，每一匹马跳跃后会多出 8 匹新的马，因为要保证第 1 批的 8 匹马全都跳完，所以新生的马就应该放在队伍的最后进行等待。

这种结构不就是**队列**吗：

- 入口在队尾： 所有新克隆出来的马，只能从通道的最后面排队进入。
- 出口在队首： 每次只让排在第一位的马出来执行跳跃克隆任务。

因为队列“**先进先出**”的规律，**计算机绝对不可能在处理完所有 1步马 之前，去提前处理任何一匹 2步马!**

思路总结：

1. 将起点马的坐标排进队列 Q。并在记录表上，把起点的步数改为 0。
2. **队首出列：只要队列里还有马在排队，**就获取队首并让它离开队伍。我们称它为当前马。
3. **获取坐标：** 获取当前马向 8 个方向跳跃的每一个坐标。
4. **检查：** 对于每一个跳跃过去的新坐标：
   - 检查A（越界）： 新坐标有没有跳出棋盘外？如果是，不合法，尝试下一个坐标。
   - 检查B（回头路）： 看看记录表上，这个新坐标是否已经被踩过了？如果被之前的马踩过了，不合法，尝试下一个坐标！
5. **记录与入队：** 如果新坐标通过了检查：
   - 记录步数： 在记录表上写下它的步数 → 新马的步数 = 当前马的步数 + 1。
   - 去后面排队： 让这匹合法的新马，走到队列的最末尾排队，等待未来被叫号。

我们可以根据思路来手动模拟马的跳跃，-1代表着还没有马到达该格子

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

队首

队列：

广度优先搜索

书写代码的准备工作：

1.坐标与步数的绑定

因为排进队伍的不能只是一个坐标，还得带着“当前是第几步”，所以我们需要打包： `struct Node { int x, y, step; }`

2.记录表

```
vector<vector<int>> ans(405, vector<int>(405, -1));
```

ans[x][y]：代表着走到x，y 的格子最少需要的步数

马走日字，在棋盘中间最多有 8 个方向可以跳。 如果我们按照直觉来写代码，当一匹马准备向四周跳时，代码会变成这样：

```
// 8 个 if 分支：
if (左上跳不越界且没踩过) { 登记，入队； }
if (右上跳不越界且没踩过) { 登记，入队； }
// ... (还要再复制粘贴 6 遍)
```

这不仅写起来手酸，而且只要你复制粘贴时少改了一个正负号，整个程序就会进入死循环或者得出错误答案，而且查错的时候眼睛都看不过来！

3．方向数组

我们把这 8 个方向的移动规则，提前提取出来，做成两个映射表。 在二维网格里，任何一次移动都可以拆解为 X **轴（行）的变化量和** Y **轴（列）的变化**量。

可以通过方向数组来简化实现：

```
int dx[8] = {-2, -2, -1, -1, 1, 1, 2, 2};
int dy[8] = {-1, 1, -2, 2, -2, 2, -1, 1};
```

这两个数组是**上下严格一一对应**的

```
// 马跳 8 个方向的坐标偏移量：
int dx[8] = {-2, -2, -1, -1, 1, 1, 2, 2};
int dy[8] = {-1, 1, -2, 2, -2, 2, -1, 1};
```

比如 `dx[0]` 是 `-2`，`dy[0]` 是 `-1`。组合在一起 `(-2, -1)` 代表什么？

代表**向上走 2 格，向左走 1 格**！

有了这个数组，一会儿在写循环时，我们只需要一个简单的 `for (int i = 0; i < 8; i++)` 循环。 我们让马的坐标分别加上 `dx[i]` 和 `dy[i]`，它就能像自动地把 8 个方向全部算出来遍历一遍。

广度优先搜索

我们书写一个bfs函数进行搜索：

```
void bfs(int start_x, int start_y)
```

其中参数 start_x 与 start_y 代表着马的起点坐标：

```
void bfs(int start_x, int start_y) {
    queue<Node> q; // 建立队列

    // 一开始只有起点的马，而起点的马步数是0步，排进队伍中
    q.push({start_x, start_y, 0});

    // 在记录表上登记：起点到起点的最短距离是 0
    ans[start_x][start_y] = 0;
```

广度优先搜索

开启循环，只要队列里还有元素，就要获取排在最前面的马，准备进行跳跃

```
// 只要通道里还有马在排队，就要一直循环下去
while (!q.empty()) {
    // 获取最前面的那匹马的信息，获取后将其踢出
    Node current = q.front(); q.pop();
```

如果只看不踢出，这匹马就会永远卡在队伍最前面无限分裂

广度优先搜索

当前马开始发力，尝试向 8 个方向跳跃(克隆)，并计算跳跃后落地的新坐标

```
// 马跳 8 个方向的坐标偏移量:
int dx[8] = {-2, -2, -1, -1, 1, 1, 2, 2};
int dy[8] = {-1, 1, -2, 2, -2, 2, -1, 1};
```

```
// 尝试向 8 个方向跳跃
for (int i = 0; i < 8; i++) {
    // 计算跳跃后落地的新坐标 (nx, ny)
    int nx = current.x + dx[i];
    int ny = current.y + dy[i];
```

进行边界检查，防止跳到不合法的位置

```
// 跳出棋盘边界的马，放弃
if (nx < 1 || nx > n || ny < 1 || ny > m) {
    continue; // 放弃这个方向，赶紧试下一个方向
}

// 如果 ans 表上不是 -1，说明早就被前人踩过了，放弃!
if (ans[nx][ny] != -1) {
    continue;
}
```

continue 的意思是“这个方向不合法，继续看下一个方向跳跃”。而BFS 能求最短路径的核心就是第二个 if ：**只认第一次到的马，后来的马全部无视，不走回头路!**

边界检查通过，是第一次踩到这个格子上的马，步数为它生出来的那匹老马的步数 + 1，记录最短步数后放入队列中。

```
// 为第一次踩到这个格子的马，步数为老马的步数 + 1
int next_step = current.step + 1;

// 记录下最短步数
ans[nx][ny] = next_step;

// 新马走到队伍最后面去排队，等待未来到达队首再进行跳跃
q.push({nx, ny, next_step});
```

广度优先搜索

通过队列的不断循环，马所能跳的格子也在一步步扩展，直到队列为空循环结束。

最后只需要输出 ans 数组，就能完成本题目

```
for (int i = 0; i < n; ++i) {
    for (int j = 0; j < m; ++j) {
        cout << ans[i][j] << " ";
    }
    cout << endl;
}
```

代码讲完了，可以闭上眼睛仔细想一想整个画面：

队伍最前面的马不断出来，往八个方向变出新马；新马通过检查后，跑到队伍最后面排队。 因为队列是**先进先出**的，所以步数为 1 的马绝对会排在步数为 2 的马前面！

通过这样的思路就把从四面八方同时扩散的动作，变成了一条一维队列！

以后不管题目怎么变，只要是求**最短步数、最少操作**次数的**无权图**问题，那基本就是这一套**结构体 + 方向数组 + 队列调度**的模板代码

我们就通过看到书本**P194页的代码模板与分析**回顾一遍BFS的流程，将本题目讲解之后再回看书本上的分析内容应该就会非常容易理解了。

书本P196页 例14-5 奇怪的电梯(OJ: B0205)

大楼的每一层楼都可以停电梯，而且第 i 层楼（$1 \leq i \leq N$）上有一个数字 K_i（$0 \leq K_i \leq N$）。电梯只有二个按钮：上，下。上下的层数等于当前楼层上的那个数字。当然，如果不能满足要求，相应的按钮就会失灵。例如： $3,3,1,2,5$ 代表了 K_i（$K_1 = 3$，$K_2 = 3$，……），从 1 楼开始。在 1 楼，按“上”可以到 4 楼，按“下”是不起作用的，因为没有 -2 楼。那么，从 A 楼到 B 楼至少要按几次按钮呢?

广度优先搜索

这道题不要多了个电梯的外衣就不认识了，它和马的跳跃在逻辑上是一模一样的！我们来对比一下这两道题目

核心	马的遍历 (二维)	奇怪的电梯 (一维)	核心本质
状态	`Node {x, y, step}`	`Node {floor, step}`	记录当前在哪，花了多少步
扩展	马朝 8 个方向移动	**按上/下键，朝两个方向移动**	状态转移
边界	$1 \le x \le n, 1 \le y \le m$	$1 \le floor \le N$	不能越界（不能遁地或穿顶）
判重	`ans[405][405] != -1`	`ans[205] != -1`	不走回头路
目标	填满整个最短步数的矩阵	**找到前往 B 楼的最短步数**	搜索最短路径

广度优先搜索

本题状态扩展非常简单。假设当前在第 i 层，电梯上的数字是 K_i：

- 动作 1（向上）：　新楼层 = $i + K_i$
- 动作 2（向下）：　新楼层 = $i - K_i$

推演：输入样例 5 1 5，K = [3，3，1，2，5]

*目标：从 **1 楼**出发，去 **5 楼**。*

- **起点：** 站在 1 楼。步数：0。
- **第 1 层**（按 1 次按钮能到的地方）：　1 楼的 $K_1 = 3$。
  - 选择向上：$1 + 3 = 4$ 楼。（合法，存入队列：{4楼，1步}）
  - 选择向下：$1 - 3 = -2$ 楼。（越界，直接销毁！）

第 2 层（按 2 次按钮能到的地方）： 队首是 4 楼。4 楼的 $K_4 = 2$。

- 选择向上：$4 + 2 = 6$ 楼。（越界）
- 选择向下：$4 - 2 = 2$ 楼。（合法，存入队列：{2楼，2步}）

第 3 层（按 3 次按钮能到的地方）：队首是 2 楼。2 楼的 $K_2 = 3$。

- 选择向上：$2 + 3 = 5$ 楼。（**命中终点**！返回步数 3，结束！）
- 选择向下：$2 - 3 = -1$ 楼。（越界）

通过 BFS 把所有 **按 X 次按钮能到达的楼层按批次扫描一遍**。只要第一次踩到目标楼层 B，返回的绝对是最少按键次数！

广度优先搜索

回忆我们上一节课写的 BFS 模板。结构体、队列、初始化、while 循环拿队首踢队首、状态扩展、越界判断、判重入队……我们把这些套路原封不动地搬过来，就能将本题解决掉。

准备工作

```
int N, A, B;
vector<int> K(205);         // 记录每层楼的按键数字
vector<int> ans(205, -1); // 记忆数组：记录到达某层的最少按键次数，兼职判重

// 状态结构体
struct Node {
    int floor;     // 当前楼层
    int step;      // 按键次数
};
```

看清本质后，就会发现看似不同的问题，其实底层都是同一套逻辑代码

```
// 特判：如果起点就是终点，直接不用按电梯
if (A == B) return 0;

queue<Node> q;

// 起点入列并将步数登记在册
q.push({A, 0});
ans[A] = 0;
```

建立队列 q。把起点楼层 A 和初始步数 0 打包成 Node 塞进队列，并在 ans 表上写下：起点到起点的按键次数是 0。

广度优先搜索

```
while (!q.empty()) {
    Node current = q.front();
    q.pop();

    int curr_f = current.floor; // 当前所在楼层
    int curr_s = current.step;  // 走到这层花了多少步
```

- 经典的 `while (!q.empty())` 是我们 BFS 的主循环。只要队列里还有“等候处理的楼层”，搜索就不会停止。
- 拿队首，踢队首： `front()` 读取数据，`pop()` 踢出队列。随后，我们把当前楼层 `curr_f` 和当前步数 `curr_s` 提取出来，准备以此为跳板进行扩散分裂。

```
// 状态扩展（因为只有上和下两个方向，可以直接手写）
int next_floors[2] = {curr_f + K[curr_f], curr_f - K[curr_f]};
```

- 上节课马走日有 8 个方向，我们用了 dx 和 dy 数组。但在电梯里，你在第 i 层只能按“上”或“下”。
- 所以目标楼层：要么加上这层的数字（curr_f + K[curr_f]），要么减去这层的数字（curr_f - K[curr_f]）。我们直接把这两个结果装进一个小数组 next_floors 里，方便下一步循环判断。

广度优先搜索

```cpp
for (int i = 0; i < 2; i++) {
    int nf = next_floors[i]; // 取出一个即将到达的新楼层 nf
    // 检查是否越界（遁地或穿顶）
    if (nf < 1 || nf > N) continue;
    // 检查是否已经来过
    if (ans[nf] != -1) continue;

    // 到达了目标楼层，得到最小的步数结果，直接返回
    if (nf == B) return curr_s + 1;
```

- 物理限制： 楼层不能小于 1，也不能大于最高层 N。遇到越界的，直接 continue。
- 不走回头路： 如果 ans[nf] 不是 -1，说明之前已经有更快的路径到达过这个楼层了。

继续走只会绕远路，直接 continue。

```
// 如果不是目标，登记最小步数，派去队伍后排等候下一步扩散
ans[nf] = curr_s + 1;
q.push({nf, curr_s + 1});
```

入队等待： 如果这个新楼层合法，且不是终点。我们就把它的最小步数记录在案，并把它打包成新的 Node 排入队尾，等待下一次的扩散。

```
// 如果队列全部清空（能到的楼层全走遍了）
// 还没找到目标 B，说明物理上就到不了 B
return -1;
```

如果 `while` 循环一直运行到队列 `q` 空了，都没触发前面的 `return curr_s + 1`，说明什么？

说明穷尽了所有合法的按键组合，也无法到达目标楼层 B。这时候程序跳出循环，执行最后的 `return -1`，契合题意无法到达输出 `-1`。

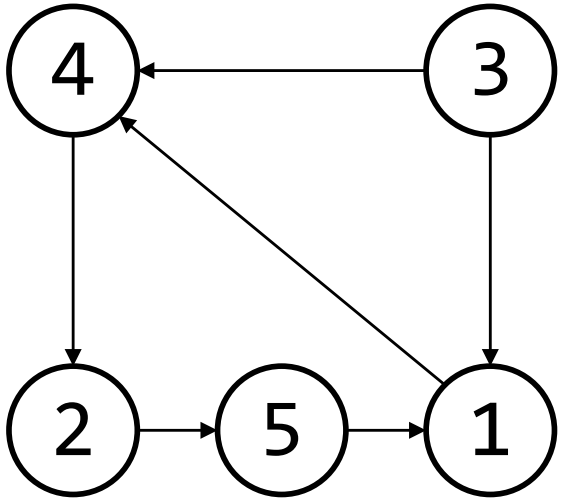
广度优先搜索

这道题的本质是在求解一个**有向无权图的最短路径**问题。

- **顶点集 (V)：** 大楼里的每一层楼（1 到 N）构成图的节点。
- **边集 (E)：** 在第 i 层按“上”或“下”按钮，等价于从节点 i 向节点 $i + K_i$ 和 $i - K_i$ 连出两条有向边。因为按一次按钮的代价固定，所以所有边的权重均为 1。
- 目标： 求解从起点 A 到终点 B 的**最短路径长度**。

```
K = [3, 3, 1, 2, 5]
```

目标：从 1 楼出发，去 5 楼。



广度优先搜索

对于无权图的最短路问题，**广度优先搜索（BFS）**是理论上的最优解。

- **为什么不用 DFS?**

深度优先搜索容易会顺着一条并非最优的路径一直走到黑，无法保证求出的是“最少按键次数”。

- **BFS 的正确性基于单调性：**

BFS 借助 FIFO（先进先出）队列，强制算法按照距离起点 0 步、1 步、2 步的拓扑层序推进。 这种层序遍历的物理特性保证了：**任何一个状态（楼层）第一次入队时，所携带的步数必定是全局最短步数。**

练习：

给定一个 $N \times M$ 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，问从起点坐标到终点坐标最少的移动步数是多少。

广度优先搜索

书本P197页 例14-6 Meteor Shower S 流星雨(OJ: B0206)

根据预报，一共有 M 颗流星会坠落在农场上，其中第 i 颗流星会在时刻 T_i砸在坐标为 (X,Y) 的格子里。流星的力量会将它所在的格子，以及周围 4 个相邻的格子都化为焦土，贝茜无法再在这些格子上行走。

贝茜在时刻 0 开始行动，她只能在会在横纵坐标 $X,Y \geq 0$ 的区域中，平行于坐标轴行动，每 1 个时刻中，她能移动到相邻的格子中的任意一个，目标格子没有被烧焦才行。如果一个格子在时刻 t 被流星撞击或烧焦，那么贝茜只能在 t 之前的时刻在这个格子里出现。贝茜一开始在 $(0,0)$。

计算贝茜最少需要多少时间才能到达一个安全的格子。如果不可能到达输出 -1。

广度优先搜索

过去我们做迷宫题，墙永远是墙，一条路永远是路。但在本题里，地图是随着时间出现变换的！有些格子现在还能走，下一秒就变成了焦土不能走。

BFS 解决时间动态障碍问题的方法就是：**预处理障碍时间表。**

我们可以提前储存好**每个格子的死亡时间。**

- 安全区： 死亡时间是一个非常大的数（永远炸不到）。
- 倒计时区： 死亡时间是 `T`。只要我们到达这个格子的时间 `step < T`，

我们就能活着踩上去，并在下一秒离开。

广度优先搜索

将每个格子的死亡(变成焦土)时间填入二维表格 `death_time` 中

`death_time[x][y] = t` 就代表着(x,y)这个格子在t时刻变成了焦土

- **初始化：** 把整张表填满一个非常大的数字（`INT_MAX`），代表一开始都是安全的。
- **记录：** 每读入一颗流星 `(X, Y, T)`，就把 `(X, Y)` 以及它的上下左右 4 个邻居格子的死亡时间更新。
- **取最小值：** 如果同一个格子被多颗流星砸中怎么办？当然是谁先砸下来，就听谁的！ 所以要用 `min(原来的死亡时间, 当前流星的时间)`。

广度优先搜索

```
// 记录每个格子的最早被炸毁时间
vector<vector<int>> death_time(305, vector<int>(305, INT_MAX));
// 流星波及的 5 个位置：自己加上下左右
int mx[5] = {0, -1, 1, 0, 0};  int my[5] = {0, 0, 0, -1, 1};

for (int i = 1; i <= M; i++) {          // 循环每一颗流星
    int x, y, t;  cin >> x >> y >> t;  // 输入流星的数据
    for (int k = 0; k < 5; k++) {
        int nx = x + mx[k];  int ny = y + my[k];
        if (nx >= 0 && ny >= 0)
            death_time[nx][ny] = min(death_time[nx][ny], t); // 记录最早时间
    }
}
```

提前储存好每个格子的死亡时间

广度优先搜索

在前面的迷宫或电梯题里，我们都有一个明确的终点。但在这道题里，**终点的安全区到底应该如何判断已经到达？**

我们不知道具体的安全坐标，但我们知道安全区的**属性**！

- 只要贝茜踏上了一个格子，我们发现这个格子的毁灭时间是 `INT_MAX` 说明这里永远安全！
- BFS 队列中，第一个踩到毁灭时间为 `INT_MAX` 的格子的分身，它所花的时间就是我们的最终答案。

广度优先搜索

传统的 BFS 只需要判断“**越不越界**”和“**有没有走过**”这两个边界条件。

但是对于本题目我们还要加上**时间线**。 假设贝茜当前走到新格子需要花费的时间是 `next_time`（也就是上一步的时间 + 1）。

- **存活条件：** `next_time` < 毁灭时间。注意是严格小于， 题目明确规定：如果格子在时刻 t 被烧焦，贝茜只能在 t 之前出现。等于 t 也会被烧死！
- **越界：** 流星落点的坐标最大是 300。但为了躲避流星，贝茜可以跑到 301、302 甚至更远的地方！所以，地图的上方和右方没有边界限制，只需要保证横纵坐标 ≥ 0 即可。

整个程序的逻辑已经非常清晰了，可以总结出四个步骤：

1. 初始化一张足够大的二维数组（比如 305 × 305），把所有的格子填上INT_MAX。

2. **写时间：** 读入所有流星数据，把波及到的格子更新为最早的毁灭时间。

3. 特判： 看看原点（0，0）的毁灭时间是不是 0。如果是，则无法出发，直接输出 -1。

4. BFS： 带着步数（时间）的参数开始向四个方向扩展。**只要新坐标 ≥ 0、没走过、且到达时间 < 毁灭时间**，就入队。一旦发现新格子的毁灭时间是 INT_MAX ，说明已经找到了一个解。

只讲解部分代码，初始化与读入流星数据部分省略

```
// 0 时刻在起点 (0,0) 入队
q.push({0, 0, 0});
time[0][0] = 0; // time[x][y]表示走到(x,y)点需要的时间
while (!q.empty()) {
    Node curr = q.front(); q.pop(); // 获取队首节点
    // 只要踩到一个安全的格子，根据 BFS 单调性，此时的时间绝对是最短的
    if (death_time[curr.x][curr.y] == INT_MAX) {
        cout << curr.time << endl; return 0;
    }
```

广度优先搜索

带着步数（时间）的参数开始向四个方向扩展。只要新坐标 ≥ 0、没走过、且到达时间 $<$ 毁灭时间，就入队。

```
// Bessie 开始向 4 个方向行走
for (int i = 0; i < 4; i++) {
    int nx = curr.x + dx[i]; int ny = curr.y + dy[i]; int next_time = curr.time + 1;

// 新坐标是否越界
if (nx < 0 || ny < 0 || nx > 305 || ny > 305) continue;
    // 不走回头路
    if (time[nx][ny] != -1) continue;

    // 走到这个格子的时间，必须小于它的毁灭时间
    if (next_time >= death_time[nx][ny]) continue;

    // 本格子可以走！记录时间并入队，等待下一次尝试扩展
    time[nx][ny] = next_time;
    q.push({nx, ny, next_time});
}
```

广度优先搜索

如果队列全部清空（能跑的地方全跑遍了），还没遇到 INT_MAX 的安全区，说明必死无疑，输出 -1 即可

```
while (!q.empty()) {
    Node curr = q.front();  q.pop();
    // 踩到安全的格子，此时的时间是最短的
    if (death_time[curr.x][curr.y] == INF) {
        cout << curr.time << endl;
        return 0;
    }
    // ...省略
}

// 如果队列全部清空，还没遇到安全区，说明必死无疑
cout << -1 << endl;
```

广度优先搜索

本题本质上是一个**带有动态时间约束的无权图最短路径问题。**

算法通过预处理，将所有障碍物的时间信息映射为一个方格毁灭时间数组（重叠区域取时间最小值）

随后根据BFS的层序扩展特性，在常规的空间边界判定与判重(走回头路)之外，引入“到达时间必须小于网格毁灭时间”的约束，从而在二维坐标系中，高效搜索出通往安全区的最少步数。