C++算法讲义：搜索算法 - 深度优先搜索 (DFS) 与回溯

1. 引言

- **问题引入：四阶数独（OJ:B0201）**
  数独是一种经典的逻辑游戏。我们这里要解决的是一个简化版的 4x4 数独。
  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）
     任务： 求所有可能的合法填写方法的总数。
- **价值阐述**
  暴力枚举的策略，会将所有可能的情况都枚举一遍。但其效率如同愚公移山，无法应付数据范围稍大的情形。
  搜索算法（包括 DFS 和 BFS）是在暴力枚举的基础上，引入了更聪明的策略，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，但是它的**算法特性决定了效率比直接的枚举所有答案要高**，因为它可以通过**剪枝**跳过一些无效状态，降低问题规模。

  DFS 及其背后的**回溯思想**，是解决很多复杂问题（如全排列、组合、路径寻找、NP 问题）的基石。掌握后我们可以：
  - **应对复杂问题**：解决那些需要穷举所有可能性，但又可以通过“提前发现错误”来优化的问题。
  - **理解递归深层逻辑**：深入理解函数调用栈在算法中的关键作用。
  - **培养抽象思维**：将现实问题抽象为搜索路径，并设计剪枝策略。

---

2. 核心思想

2.1 从暴力到智能：数独问题的思考

我们以 4x4 数独为例，看看为什么我们需要 DFS。

- **解法一：纯暴力枚举**
  - **思路**：4×4 共 16 个格子。第 1 个格子试 1、2、3、4；第 2 个格子试 1、2、3、4.....直到第 16 个格子。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：按行生成全排列**
  - **思路**：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。
  - **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?
  如果我们把问题放大到 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 4x4 矩阵，然后才去检查。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。

2.2 深度优先搜索 (DFS) 的核心逻辑

DFS 的核心思想就是：**走一步看一步，不合法就回头，换条路再走。**

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**：站在当前需要填写的空格，从所有可能的数字（1~4）中逐个尝试。
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(下一个空格)`。
4. **死胡同就回头（回溯）**：假如我在某一个空格试了 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 $\rightarrow$ 检查 $\rightarrow$ 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。

2.3 递归与回溯的底层机制：系统栈

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 (push)**，等结束了再**弹栈 (pop)** 回来继续干活。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**系统栈**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。

- **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。
- **流程模拟**：
  - `dfs(k)` 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。`dfs(k)` 的状态被压栈。
  - `dfs(k+1)` 返回 `return`：
    - 如果 `dfs(k+1)` 找到了解：`return` 后，`dfs(k)` 所在的那条路径可能是一条完整解的一部分。
    - 如果 `dfs(k+1)` 发现死胡同：`dfs(k+1)` 尝试完所有可能性后，没有为第 `k+1` 格找到任何一个合法的数字，它会执行 `return`，并将执行控制权交还给它的调用者——`dfs(k)`。
      此时，`dfs(k)` 从暂停处恢复执行。它收到 `dfs(k+1)` 的返回，意识到当初为 `k` 格填入的数字，导致了后续格子无解。它必须清除自己刚才做的决策，即把第 `k` 格的数字擦掉（**恢复现场**），以便 `for` 循环尝试下一个数字。
      然后，`dfs(k)` 的 `for` 循环继续，尝试为第 `k` 格填入下一个数字。

总结：
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。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的状态，确保函数可以从正确的位置继续执行。

---

3. 实例

3.1 数独问题：完整框架搭建 (例15-3)

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)); // 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（下面会详细实现）
bool check(int x, int y, int num) {
    // 检查行
    for (int j = 0; j < 4; ++j) {
        if (grid[x][j] == num) return false;
    }
    // 检查列
    for (int i = 0; i < 4; ++i) {
        if (grid[i][y] == num) return false;
    }
    // 检查2x2子区域
    int startX = (x / 2) * 2; // 计算2x2子区域的左上角行坐标
    int startY = (y / 2) * 2; // 计算2x2子区域的左上角列坐标
    for (int i = startX; i < startX + 2; ++i) {
        for (int j = startY; j < startY + 2; ++j) {
            if (grid[i][j] == num) return false;
```

```
        }
    }
    return true; // 所有检查都通过，合法
}

// 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k个空格（0-15）转换为二维的(x, y)坐标
    int x = k / 4; // 行号
    int y = k % 4; // 列号

    // 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) {
        // 步骤 2: 立刻检查 -- 判断填入num是否合法
        if (check(x, y, num)) {
            // 步骤 3: 合法就深入 -- 保存现场，递归调用
            grid[x][y] = num; // 把数字实际填入棋盘
            dfs(k + 1);       // 深入，去处理下一个空格 (k+1)

            // 步骤 4: 死胡同就回头（回溯）-- 恢复现场
            // 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;
        }
    }
}

int main() {
    dfs(0); // 游戏开始：从第0个格子开始填写
    cout << count << endl;
    return 0;
}
```

`check` **函数的实现细节：**

`bool check(int x, int y, int num)`

- **检查行**：`num` 不能与 `(x, y)` 所在行的任何其他数字相同。
  - 行号 `x` 是固定的。我们需要循环改变的是列号 `j`。
  - `for (int j = 0; j < 4; ++j) { if (grid[x][j] == num) return false; }`
- **检查列**：`num` 不能与 `(x, y)` 所在列的任何其他数字相同。
  - 列号 `y` 是固定的。我们需要循环改变的是行号 `i`。
  - `for (int i = 0; i < 4; ++i) { if (grid[i][y] == num) return false; }`
- **检查 2x2 子区域**：不能与当前坐标所在的 2x2 小方块内的任何其他数字相同。
  - **如何确定子区域的左上角 `(startX, 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;`
- 同理，`int startY = (y / 2) * 2;`

- 然后遍历 `(startX, startY)` 到 `(startX+1, startY+1)` 这四个格子。

优化 `check` 函数：使用桶数组

为了提高 `check` 函数的效率，可以引入三个**桶数组**（布尔数组）来实时记录每个区域的数字占用情况。这样，`check` 函数就不需要遍历，而是 O(1) 地查询。

```
vector<vector<int>> grid(4, vector<int>(4, 0)); // 4x4棋盘
int count = 0;

// 三个桶数组，用来快速检查冲突
bool row_used[4][5];    // row_used[i][num] 表示第i行num是否已被使用 (数字1-4)
bool col_used[4][5];    // col_used[j][num] 表示第j列num是否已被使用
bool block_used[4][5];  // block_used[k][num] 表示第k个2x2子区域num是否已被使用

// check函数变得非常简单，只是查询桶数组
bool check(int x, int y, int num) {
    int block_idx = (x / 2) * 2 + (y / 2); // 计算当前格子所在的2x2子区域索引 (0-3)
    return !row_used[x][num] && !col_used[y][num] && !block_used[block_idx][num];
}

void dfs(int k) {
    if (k == 16) {
        count++;
        return;
    }

    int x = k / 4;
    int y = k % 4;
    int block_idx = (x / 2) * 2 + (y / 2);

    for (int num = 1; num <= 4; ++num) {
        if (check(x, y, num)) {
            // 保存现场：更新棋盘和桶数组
            grid[x][y] = num;
            row_used[x][num] = true;
            col_used[y][num] = true;
            block_used[block_idx][num] = true;

            dfs(k + 1); // 深入

            // 恢复现场：撤销棋盘和桶数组的更改
            grid[x][y] = 0;
            row_used[x][num] = false;
            col_used[y][num] = false;
            block_used[block_idx][num] = false;
        }
    }
}

int main() {
    // 初始化桶数组
```

```
    for (int i = 0; i < 4; ++i) {
        for (int j = 0; j < 5; ++j) { // 数字1-4，所以大小为5
            row_used[i][j] = false;
            col_used[i][j] = false;
            block_used[i][j] = false;
        }
    }

    dfs(0);
    cout << count << endl;
    return 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总结为一套框架。

3.2 DFS 回溯框架总结

从暴力枚举到启发式剪枝：

- **问题抽象**：在 N×N 矩阵中填充数字/放置棋子，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续匹配。
  - **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。

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（剪枝）。
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。

DFS 回溯代码模板在实现上，会呈现其严格的对称性：
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。以下是给出**DFS的模板代码**：

```
void dfs(int step) {
    // 递归出口：如果到达目标状态/所有决策都已完成
    if (到达目标状态) {
        记录解;
        return;
    }

    // 状态扩展：遍历所有候选选项
    for (遍历所有候选选项) {
```

```
            // 约束满足性检查：剪枝
            if (当前选项满足约束条件) {
                // 1. 保存现场（对全局状态进行修改）
                update_state(选项);

                // 2. 状态转移：进入子问题，递归调用
                dfs(step + 1);

                // 3. 恢复现场（撤销对全局状态的修改）
                undo_state(选项);
            }
        }
    }
```

3.3 经典案例：八皇后问题 (P190页 例14-2, OJ:B0202)

题目描述：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：放置 N 个皇后，使得她们互不攻击。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，并按字典序输出前 3 种具体的放置方法。

核心转化：
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……
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。

如何 `check` 冲突?
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）。

1. **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。
2. **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！
    - **右上 $\rightarrow$ 左下的斜线 (副对角线)**：你观察一下这条线上的坐标：
      `(0,0), (1,1), (2,2)`
      `(0,1), (1,2), (2,3)`
      `(1,0), (2,1), (3,2)`
      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[sum] = true`。下次谁想放在坐标和为 `sum` 的地方，就先查看一下 `c[sum]`，发现结果等于 `true`，也就是已经被占用了！
    - **左上 $\rightarrow$ 右下的斜线 (主对角线)**：你再观察这条线上的坐标：
      `(0,2), (1,1), (2,0)`
      `(0,0)`
      `(1,2), (2,3)`
      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[offset]` 就是 `true`。

```
int N;                  // 棋盘大小，最多为 13 x 13
int ans_count = 0;      // 记录总方案数
int a[20];              // 记录答案：a[行号] = 列号 (a[1]表示第一行皇后的列号)

// 三个桶数组，分别是列，副对角线，主对角线
bool col_used[20];      // 监控列：col_used[列号]
bool diag_sum_used[40];     // 监控副对角线：diag_sum_used[行 + 列] (取值最大为
12+12=24)
bool diag_diff_used[40];    // 监控主对角线：diag_diff_used[行 - 列 + N] (取值最大为
12 - 1 + 12 = 23)

// 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_count++; // 找到一种合法方案，总数 +1
        if (ans_count <= 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
    for (int col = 1; col <= N; col++) {
        // 检查：看三个标记数组是不是全是 false (即当前位置是否合法)
        // (row + col) 作为副对角线索引
        // (row - col + N) 作为主对角线索引 (加上N保证非负)
        if (!col_used[col] && !diag_sum_used[row + col] && !diag_diff_used[row -
col + N]) {
            // 位置合法，那就深入
            a[row] = col; // 记录当前行的决定：第row行的皇后放在col列

            // 保存现场：放了皇后，对应的标记数组标记为 true
            col_used[col] = true;
            diag_sum_used[row + col] = true;
            diag_diff_used[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);

            // 恢复现场：当 dfs(row + 1) 返回时，说明后面的路走完了
            // 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
            col_used[col] = false;
            diag_sum_used[row + col] = false;
            diag_diff_used[row - col + N] = false;
        }
    }
}
```

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

注意：如果不写 `b[col] = c[row + col] = d[row - col + N] = false;` 这句会怎样？
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。**恢复现场**是回溯算法一定要做的事情！

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你解决！

3.4 剪枝优化：考前临时抱佛脚 (P191页 例14-3, OJ:B0203)

题目描述：kkksc03 需要考 4 科。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。kkksc03 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**一科一科地复习。**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。**

题意总结：给定 4 个独立的数字集合（每科的题目耗时）。对于每个集合，要求将其分为两个子集（左右脑），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。

算法拆解：

1. **每科独立计算**：
    - “必须一科一科复习”意味着 4 个科目是完全独立的。
    - 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。
2. **单科耗时取最大值**：
    - “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。
    - 单科耗时 = `max(左脑做题总耗时, 右脑做题总耗时)`。
3. **均分逼近目标**：
    - 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。

DFS 框架应用：
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。
- **选择 B**：交给右脑。

我们可以构思一个递归函数 `dfs(index, sum_L, sum_R)`，参数传递：当前正准备分配第 `index` 题，以及此时左右脑的占用情况 `sum_L` 和 `sum_R`。

```
int s;              // 当前科目题目总数
vector<int> dfs_times; // 当前科目每道题所需时间
```

```cpp
int min_total_time; // 记录当前科目的最短完成时间

// dfs函数：为当前科目的第index道题分配给左脑或右脑
// index: 当前处理的题目索引
// sum_L: 左脑当前累计的完成时间
// sum_R: 右脑当前累计的完成时间
void dfs(int index, int sum_L, int sum_R) {
    // 递归出口：如果所有题目都已分配完毕
    if (index == s) {
        min_total_time = min(min_total_time, max(sum_L, sum_R));
        return;
    }

    // 剪枝1：最优性剪枝
    // 如果当前的 sum_L 或 sum_R 已经比目前找到的最优解还要大，
    // 那么这条路径无论如何都无法得到更优的解，可以直接放弃。
    if (max(sum_L, sum_R) >= min_total_time) {
        return;
    }

    // 决策分支1：将第 index 道题分配给左脑
    // 深入：带着新的左脑时间进入下一层递归
    dfs(index + 1, sum_L + dfs_times[index], sum_R);

    // 决策分支2：将第 index 道题分配给右脑
    // 深入：带着新的右脑时间进入下一层递归
    dfs(index + 1, sum_L, sum_R + dfs_times[index]);
}

int main() {
    int total_min_time = 0; // 四科的总最短时间

    for (int k = 0; k < 4; ++k) { // 处理4门科目
        cin >> s; // 读取当前科目的题目数量
        dfs_times.resize(s); // 调整 vector 大小
        for (int i = 0; i < s; ++i) {
            cin >> dfs_times[i]; // 读取每道题的时间
        }

        // 剪枝2：优化搜索顺序 (可选)
        // 将题目按时间从大到小排序，可以更早触发最优性剪枝。
        sort(dfs_times.rbegin(), dfs_times.rend());

        min_total_time = 1e9; // 初始化当前科目的最短时间为一个极大值
        dfs(0, 0, 0); // 从第0题开始分配，左右脑初始时间为0
        total_min_time += min_total_time; // 累加当前科目的最短时间
    }

    cout << total_min_time << endl;
    return 0;
}
```

剪枝优化：

1. **最优性剪枝**：

- 在 `dfs` 函数开头添加判断：`if (max(sum_L, sum_R) >= min_total_time) return;`
- 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_total_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_total_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。

书本上的做法：进一步优化
如果我们把任务分成两堆，必然是一堆 $\leq \text{sum}/2$，另一堆 $\geq \text{sum}/2$。
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 `total_sum / 2`！
目标变成了：在不超过 `total_sum / 2` 的前提下，让左脑拿到的题目总耗时**尽可能大**（越接近 `total_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 > total_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 `total_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_time`。
  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：`total_sum - max_left_time`。

```
int s;              // 当前科目题目总数
vector<int> times; // 当前科目每道题所需时间
int half_sum;         // 当前科目总时间的一半
int max_left; // 左脑能累积到的最大时间（不超过half_sum）

// 优化后的dfs函数：只关注左脑的累积时间
// index: 当前处理的题目索引
// current_L_time: 左脑当前累计的完成时间
void dfs(int index, int current_L_time) {
    // 递归出口：如果所有题目都已分配完毕
    if (index == s) {
        max_left = max(max_left, current_L_time);
        return;
    }

    // 将第 index 道题尝试分配给左脑
    // 条件：只有当加入这道题后不超过 half_sum，才可以分配给左脑！
    if (current_L_time + times[index] <= half_sum) {
        // 深入：把题装进去了，时间增加
        dfs(index + 1, current_L_time + times[index]);
    }

    // 如果不分配给左脑的话，将第 index 道题尝试分配给右脑
    dfs(index + 1, current_L_time);
}
```

```cpp
int main() {
    int ans = 0; // 四科的总最短时间

    for (int k = 0; k < 4; ++k) { // 处理4门科目
        cin >> s; // 读取当前科目的题目数量
        times.resize(s); // 调整 vector 大小
        int current_sum = 0; // 当前科目的总时间
        for (int i = 0; i < s; ++i) {
            cin >> times[i]; // 读取每道题的时间
            current_sum += times[i];
        }

        // 剪枝：优化搜索顺序 (推荐)
        sort(times.rbegin(), times.rend());

        half_sum = current_sum / 2; // 左脑的理论上限
        max_left = 0;             // 初始化左脑能累积到的最大时间

        dfs(0, 0); // 从第0题开始分配，左脑初始时间为0

        // 当前科目的最短时间 = max(左脑时间, 右脑时间)
        // 因为左脑尽量接近 half_sum，所以右脑的时间 = current_sum - max_left
        // 最终最短时间 = max(max_left, current_sum - max_left)
        ans += max(max_left, current_sum - max_left);
    }

    cout << ans << endl;
    return 0;
}
```

---

4. 总结与练习

- **要点回顾**
    1. **DFS 核心思想**：通过递归试探性地向前探索，遇到死胡同或不合法状态则回溯，撤销当前决策，尝试其他路径。
    2. **回溯本质**：利用系统栈保存函数上下文，实现状态的“前进探索”和“失败回退”。
    3. **DFS 基本框架**：
        - **递归函数** `dfs(step)`
        - **递归出口**：达到目标或所有决策完成。
        - **决策分支** (`for` 循环)：遍历当前步骤的所有可能性。
        - **剪枝** (`if` 条件)：提前判断当前决策是否合法，不合法则跳过，避免无效搜索。
        - **保存现场**：在递归调用前，修改全局状态。
        - **恢复现场**：在递归调用返回后，撤销对全局状态的修改，确保回溯到决策点时状态纯净。
    4. **剪枝策略**：
        - **可行性剪枝**：检查当前决策是否违反规则。
        - **最优性剪枝**：如果当前路径已不可能达到更优解，则剪掉。
        - **搜索顺序优化**：合理排序决策选项，有助于更早触发剪枝。