DFS 回溯与剪枝优化

搜索树状态空间、回溯恢复现场与五大剪枝技巧

HGNU-ACM 实验室算法组

配套：搜索树与指数枚举 + 全排列回溯机制 + 五大剪枝 + 3 道真题精讲 + 6 题练习





本章目录（1/2）

搜索树状态空间与选/不选框架

DFS 回溯框架与“恢复现场”机制

五大剪枝优化技巧：半路及时刹车

前三节承接《分治算法》的递归思路，从零建立搜索树概念、回溯“做选择 → 递归 → 撤销现场”机制，并逐个拆解五大剪枝技巧。

本章目录（2/2）

三大 DFS 回溯与剪枝真题精讲

常见错误、检查清单与练习安排

后两节逐题推导《组合的输出》《八皇后》《马走日》，附上高频错误对照、六必查清单与课后练习安排。

从《分治算法》出发：什么时候必须“把每条路都试一遍”？

先对比两件事

上一章《分治算法》里，我们把大问题（如快速幂 b^p）对半拆成子问题，沿公式算完就能合并；但如果题目问“列出 $\{1,2,3\}$ 的全部合法选法”，没有现成公式，只能把所有选择逐一试完。

- 分治：大块拆小块，子问题各算一次再拼起来。
- 搜索（穷举）：面对一连串岔路口，系统地把每条可能路径走一遍，不遗漏也不重复。
- 写搜索前先答三问：当前走到哪一步、面前有几个选择、走到头怎么算答案。

本页要点

当问题要求“列出所有方案”或“在复杂限制下找合法解”时，就需要把选择连成搜索树！

先想一个小例子：3 件物品装书包，有几种选法？

生活例子：桌上有苹果、面包、水杯 3 件物品，每件都可以“带”或“不带”

手算把所有装法列出来，看看一共有多少种：

- 第 1 件（苹果）：选（✓）或不选（×），共 2 种。
- 第 2 件（面包）：在前面每种基础上，又有选/不选 2 种，变成 $2 \times 2 = 4$ 种。
- 第 3 件（水杯）：再乘 2，得到 $2 \times 2 \times 2 = 8$ 种（包括一件都不带的空书包）。

发现了什么规律？

每一次做决定，方案就分叉成两支；n 件物品连续做 n 次决定，所有选择正好连成一棵不断分叉的树！

本页要点

先用手算小例子数清每一步有几个分支，再考虑怎么让电脑替你走这棵树。

新术语大白话：状态、状态空间与搜索树

第一次见到“搜索树”和“状态空间”，不要被名词吓到，用大白话翻译如下：

- 状态：程序走到某一步时，手里已经做出的决定和当前位置。比如“前 2 件已决定选苹果、不选面包，准备看第 3 件”。
- 状态空间：这道题所有可能出现的状态合在一起的“大地图”。3 件物品选或不选，共有 $2^3 = 8$ 个终点状态。
- 搜索树：把“从起点出发，每做一次选择就长出一条树枝”的过程画成的树形图。
- 叶子节点（递归出口）：所有物品都问完了，不再往下分叉，直接输出或统计当前的方案。

本页要点

状态 = 当前局面；状态空间 = 所有局面集合；搜索树 = 把所有选择分叉连起来的图。

为什么不用多层 for 循环硬写？

如果想用之前学过的 for 循环枚举 n 件物品的“选与不选”：

- $n = 3$ 时，要嵌套写 3 层 `for (int i = 0; i <= 1; ++i)`，每层取 0 或 1。
- 如果题目输入 n 最多到 20，你总不能手写 20 层嵌套 for 循环，而且 n 是输入时才确定的！
- 更严重的是算一下操作次数：$n = 20$ 有 $2^{20} \approx 1.05 \times 10^6$ 个叶子，还勉强能跑；若 $n = 30$，则有 $2^{30} \approx 1.07 \times 10^9$ 个叶子，远超 1 秒约 10^8 次运算的上限，直接 **TLE**！

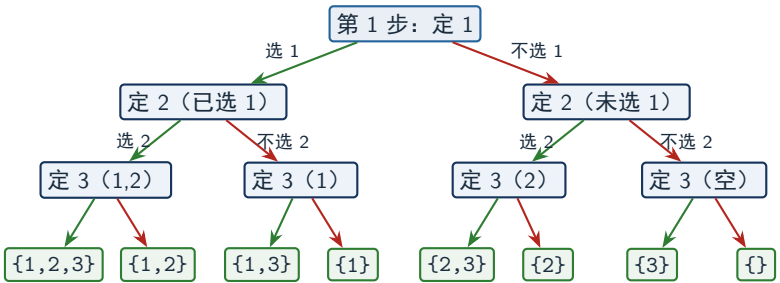
解决方法

用递归函数 `dfs(step)` 代替固定层数的 for 循环：参数 step 自动记录当前正在决定第几件物品！

本页要点

递归的深度代替嵌套循环的层数，既能适应任意输入的 n，又便于中途提前刹车剪枝。

TikZ 图解：3 件物品“选与不选”搜索树



本页要点

树上共有 8 个绿框叶子，对应 $2^3 = 8$ 种选法；DFS 会一直顺着左边“选”走到叶子，再退回换右枝。

手算跟踪表：按“先选后不选”走完 $n = 2$ 搜索树

以 $n = 2$ 为例，手算每一步调用的层数 step、记录状态 pick[1..2] 与系统调用栈（包含最后条件退回步）：

步序	当前调用	pick[1,2]	step > 2?	动作与下一步
1	dfs(1)	[0, 0]	假	令 pick[1]=1，调用 dfs(2)
2	dfs(2)	[1, 0]	假	令 pick[2]=1，调用 dfs(3)
3	dfs(3)	[1, 1]	真（终止）	输出 {1, 2} 并返回 dfs(2)
4	回到 dfs(2)	[1, 1]	假	改令 pick[2]=0，调用 dfs(3)
5	dfs(3)	[1, 0]	真（终止）	输出 {1}，dfs(2) 两条路走完退回 dfs(1)
6	回到 dfs(1)	[1, 0]	假	改令 pick[1]=0，继续搜 {2} 和 {} 后结束

本页要点

第 3、5 步 step > 2 为真时立刻返回，不再递归；每层两支都试完才退回上一层。

模板代码：选与不选（指数型枚举）

```
int n, pick[25];                                    // pick[i]=1 表示选 i，0 表示不选
void dfs(int step) {
    if (step > n) {                                 // 1. 递归出口：1~n 全部决定完
        for (int i = 1; i <= n; ++i)
            if (pick[i]) cout << i << ' ';
        cout << '\n';
        return;                                     // 一定要 return，防止继续往下走
    }
    pick[step] = 1; dfs(step + 1);                  // 2. 分支一：选当前数 step
    pick[step] = 0; dfs(step + 1);                  // 3. 分支二：不选（顺便覆盖掉 pick
    )
}
```

本页要点

先写 `if (step > n)` 出口；因 `step` 单向递增不用判重，`pick[step]=0` 顺手擦掉选过的痕迹。

为什么这样写是对的？搜索树遍历不变量

用三个阶段检查这段递归为什么不重不漏：

阶段	要验证的断言
初始化	刚调用 `dfs(1)` 时，前 0 个数字已经决定完，等待处理第 1 个数。
保持	进入 `dfs(step)` 时，`pick[1..step-1]` 已定好；程序把 `pick[step]` 分别设为 1 和 0 各递归一次，因此对前 `step-1` 步的每种组合，都完整生出了第 `step` 步的两种延展。
终止	当且仅当 `step == n + 1` 时停止分叉，此时 `pick[1..n]` 唯一对应一条从根到叶子的路径，恰好覆盖全部 2^n 种方案。

本页要点

从根到叶子的每一条路径互不相同，因此不重复；每个节点两个分支全走，因此不遗漏。

上机验证与反事实追问：调换顺序会怎样？

```
#include <iostream>
using namespace std;
int n, pick[25];
void dfs(int step) {
    if (step > n) {
        for (int i = 1; i <= n; ++i) if (pick[i]) cout << i << ' ';
        cout << '\n'; return;
    }
    pick[step] = 1; dfs(step + 1);
    pick[step] = 0; dfs(step + 1);
}
int main() { if (cin >> n) dfs(1); }
```

- 三组验证：$0 \to 1$ 个空行；$1 \to$ 两行 1 和空行；$2 \to$ 1 2、1、2、空行共 4 行。
- 反事实追问：把第 9、10 行调换成“先不选再选”，总行数仍是 2^n，输出顺序变为从空行到 1 2 ... n。

易错对照：选与不选模板的三大失误

错误类型	错误写法	正确做法与原因
少算最后一位（**WA**）	边界写成 `if (step == n)`	`step == n` 时第 n 件还没选，应写 `step > n`
漏写 `return`（**RE**）	出口里输出完忘记写 `return;`	会继续往下调用 `dfs(n+2)` 无限递归爆栈
多组未重置（**WA**）	若改用 `vector` 存路径，打印后没 `pop_back()`	上一枝的数字残留在容器里污染下一枝

本页要点

递归出口一定要配 `return;`；境界条件 `step > n` 必须比最后一个编号 n 多 1。

生活引入：把 $1,2,3$ 排成一排，不能重复用同一个数字

场景：面前有 3 个空盒子，桌上放着写有 $1,2,3$ 的 3 张卡片

每次只能拿桌上还剩下的卡片放进当前空盒：

- 先把卡片 1 放进第 1 个盒，桌上只剩 $\{2,3\}$；再把 2 放进第 2 个盒，最后把 3 放进第 3 个盒，得到 1 2 3。
- 接下来想在第 2 个盒改放 3，必须先把第 **2** 个盒里的 2 拿出来放回桌上，才能拿起 3 放进去！
- 如果不把用完的卡片放回桌上，桌上很快没卡片了，后面的排列全都拼不出来。

本页要点

卡片“拿起占用”和“放回桌上”，就是算法里最核心的标记与恢复现场！

暴力枚举为何 TLE：多重循环 vs 带 vis 回溯

如果用朴素多重循环枚举 $1 \sim n$ 的全排列再过滤重复：

- $n = 3$ 时，写 3 层 `for`（`1..3`）并检查互不相等，共执行 $3^3 = 27$ 次循环，其中只有 $3! = 6$ 次合法。
- 若 $n = 10$，朴素 10 层循环要走 10^{10} 次操作，按每秒 10^8 次计算大约需要 100 秒，必定 **TLE**！
- 而用布尔数组 `vis[v]` 标记数字 v 是否已在手上，每层只挑没用过的数字，总叶子数降为 $10! = 3{,}628{,}800$，仅几十毫秒就能跑完！

本质差距

朴素循环连 $\{1, 1, 1, \dots\}$ 这种明显非法的分支都照走不误；回溯用 `vis` 直接把重复数字分支提前挡在门外。

本页要点

过滤越早，白走的子树越少；n^n 与 $n!$ 在 $n = 10$ 时差了将近 3000 倍！

新术语大白话：DFS、回溯与“恢复现场”

- 深度优先搜索（**DFS**）：遇到岔路口时，先选第一条路一直往深处走，不撞南墙不回头；走到尽头或死路再退回上一岔路口换下一条。
- 回溯（**Backtracking**）：沿着原路退回上一层继续尝试别的分支的过程。
- 恢复现场：退回上一层时，把刚才这一步修改过的全局状态（如 `vis[v]`）改回原样，好比离开房间时随手把灯关掉、把借来的道具归还。

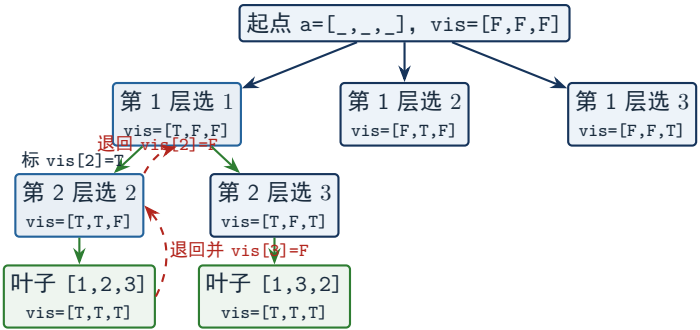
一句话记住回溯三步曲

做选择（占坑标记） $\longrightarrow$ 递归到下一层 $\longrightarrow$ 撤销选择（恢复现场）。

本页要点

DFS 负责“一路走到底”，恢复现场保证“退回来时一切如初”，合起来就是回溯搜索。

TikZ 图解：全排列“做选择 → 递归 → 撤销现场”



本页要点

绿色实线表示“占坑往下走”，红色虚线表示“返回时把 `vis` 擦回 `false`”。

手算跟踪表：$1,2,3$ 全排列前 6 步（含终止与退回）

跟踪从开始到输出第 1 组 1 2 3、并退回改选 1 3 2 的关键变量变化：

步序	当前 step	尝试数字 v	vis[1,2,3]	动作与状态说明
1	1	选 1	[T, F, F]	置 vis[1]=T, a[1]=1，进入 dfs(2)
2	2	跳过 1，选 2	[T, T, F]	置 vis[2]=T, a[2]=2，进入 dfs(3)
3	3	跳过 1,2，选 3	[T, T, T]	置 vis[3]=T, a[3]=3，进入 dfs(4)
4	4	触发 step > 3	[T, T, T]	打印 1 2 3 并返回；随后 dfs(3) 擦掉 vis[3]=F
5	退回 2	撤销 2，改选 3	[T, F, T]	先置 vis[2]=F，循环继续试 $v=3$ 并置 vis[3]=T
6	3	选 2	[T, T, T]	第 3 层拿走剩下的 2，下一层打出 1 3 2

本页要点

第 5 步如果漏掉 vis[2]=F，第 6 步就没有数字可拿，1 3 2 永远出不来！

模板代码：标准全排列回溯框架

```
int n, a[15]; bool vis[15];                          // vis[v] 记录数字 v 是否已被占用
void dfs(int step) {
    if (step > n) {                                  // 1. 递归出口：填满了 1~n 共 n 个位
    置
        for (int i = 1; i <= n; ++i) cout << a[i] << ' ';
        cout << '\n'; return;
    }
    for (int v = 1; v <= n; ++v) {
        if (vis[v]) continue;                        // 跳过已经被占用的数字
        vis[v] = true; a[step] = v;                  // 2. 做选择（占坑并写答案）
        dfs(step + 1);                               // 3. 递归填下一个位置
        vis[v] = false;                              // 4. 恢复现场（归还数字 v）
    }
}
```

要点：“成对出现”原则：进递归前写 `vis[v] = true;`，出递归后立刻写 `vis[v] = false;`！

深度拆解：为什么 vis[v] = false 漏写就全盘皆错？

正确情况（有恢复现场）

- 搜完 [1，2，3] 返回时，依次把 vis[3]、vis[2] 改回 false。
- 回到第 2 层循环试 $v = 3$ 时，vis 是 [T，F，F]，于是 3 能进第 2 位，2 还能进第 3 位，顺利输出 1 3 2。

漏写 vis[v] = false（灾难现场）

- 输出完第一个排列 1 2 3 后，vis[1]，vis[2]，vis[3] 全部永远卡在 true！
- 退回第 2 层、第 1 层时，循环里所有 if (vis[v]) continue; 全部触发，后面 5 个排列一个都找不出来！

本页要点

全排列只输出了第一行就结束，99% 是因为你忘了在 dfs(step + 1) 后面写 vis[v] = false！

为什么对？回溯状态不变量（进入与离开对称）

我们用“进入前、执行中、离开时”三段检查这个回溯框架为什么正确：

阶段	要验证的不变量
初始化	调用 `dfs(1)` 前，`vis[1..n]` 全为 `false`，已占用的数字集合为空。
保持	进入 `dfs(step)` 时，恰有 `step-1` 个 `vis` 为 `true`（就是 `a[1..step-1]`）；每次尝试新数字 `v` 时先置 `true`，等 `dfs(step+1)` 返回后再置回 `false`，保证本层循环试下一个 `v` 时现场与进入本层时完全相同。
终止	`step == n + 1` 时不再修改 `vis`，此时 `a[1..n]` 互不相同且按字典序生成。

本页要点

只要“进入一层前的状态”和“子递归返回后的状态”完全一致，分支之间就绝不会互相串台！

上机验证与反事实追问：为什么 a[step] 不用恢复？

```cpp
#include <iostream>
using namespace std;
int n, a[15]; bool vis[15];
void dfs(int step) {
    if (step > n) {
        for (int i = 1; i <= n; ++i) cout << a[i] << (i == n ? '\n' : ' ');
        return;
    }
    for (int v = 1; v <= n; ++v) if (!vis[v]) {
        vis[v] = true; a[step] = v; dfs(step + 1); vis[v] = false;
    }
}
int main() { if (cin >> n) dfs(1); }
```

- 三组验证：$1 \to 1$；$2 \to 1\ 2$、$2\ 1$；$3 \to$ 按字典序输出 6 行全排列。
- 反事实追问：为什么没写 a[step] = 0？因为下次写 a[step] 会直接覆盖旧值，而 vis[v] 会被 if (!vis[v]) 读取判断！

对比拓展：从“全排列”切换到“组合搜索”

很多初学者容易把排列和组合搞混，其实只差一个起点参数 `start`：

问题类型	关心顺序吗？	核心写法差别
全排列	关心：1 2 和 2 1 是两种	每层 `v` 都从 1 扫到 `n`，需要 `vis[v]` 判重
组合	不关心：1 2 和 2 1 算同一种	规定只选递增序列：传入参数 `start`，每层 `v` 从 `start` 扫到 `n`，下一层传 `v + 1`，不需要 `vis`！

记住这句转换口诀

需要“顺序不同算不同”→ 传 `step` 配 `vis`；只要“挑出来的集合不重复”→ 多传一个 `start` 强制递增！

本页要点

强制按从小到大的顺序去挑组合元素，就能用“顺序唯一性”天然去掉所有重复！

易错对照：DFS 回溯四大高频 Bug 速查

症状与评测结果	常见错误原因	修复方法
只输出第一组解（**WA**）	递归后忘记写 `vis[v] = false`	检查所有“做选择”语句，是否都有对称恢复
运行崩溃爆栈（**RE**）	写错递归参数，如 `dfs(step)` 没加 1	确保每次递归参数严格向出口推进
多组测试答案多（**WA**）	上一组留下的 `vis` 或 `ans` 没清空	每组 `while (T--)` 开头用 `memset` 或清零
输出少一格或越界（**WA/RE**）	`step` 从 0 开始却写了 `if (step > n)`	统一约定：`step` 从 1 开始则边界 `step > n`；从 0 开始则边界 `step == n`

本页要点

动手写回溯前先统一约定：`step` 从 0 还是 1 计，边界写 `== n` 还是 `> n`。

引入：不剪枝到底浪费了多少步？

小例子：从 $1 \sim 20$ 中挑 10 个数输出所有组合，第一步选了 15，还往下搜吗？

稍微想一下就知道：根本不用继续！因为比 15 大的数只剩下 $16, 17, 18, 19, 20$ 这 5 个，连同 15 一起全选也只有 6 个数，永远凑不够 10 个！

- 如果不拦住，程序还是会老老实实把 15 后面的 $2^5 = 32$ 种选法全走一遍；从 12 到 20 开头的全都白搜了！
- 新术语：剪枝（**Pruning**）——在搜索过程中一旦发现当前这根树枝绝对不可能走出合法解（或更好解），不等走到叶子就直接 `return`，像园丁剪掉枯枝一样砍掉整棵子树！

本页要点

剪枝第一铁律：“宁可少剪，绝不错剪”——只砍掉 100% 确定出不了合法解或更优解的分支！

五大剪枝全景图：各自解决什么问题？

1. 可行性剪枝
剩余名额不够 / 死胡同
奇偶不匹配直接判死

2. 最优性剪枝
当前代价 $\geq$ 已知最优
后续非负不可能反超

3. 上下界剪枝
估出剩余最小/最大可能
落在目标区间外就砍

4. 记忆化与排除等效冗余
固定递增序 / 跳过同层重复值
已算过的失败状态查表直接退

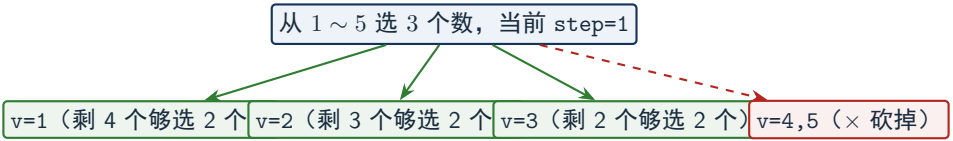
5. 优化搜索顺序
先选分支最少 / 数值大的
更早撞死胡同或拿到小 best

本页要点

实际比赛里往往要把 $2 \sim 3$ 种剪枝组合在一起用，搜索量能从几亿次骤降到几万次！

剪枝 1 | 可行性剪枝：看到必死分支立刻掉头

- 大白话：还没走到终点，就已经确定这条路不可能满足题目硬性要求，立刻放弃。
- 常见三种信号：
  1. 名额不够：还缺 k 个数，后面剩下的候选数字却不足 k 个。
  2. 奇偶/染色矛盾：比如棋盘黑白染色后步数奇偶性永远对不上目标。
  3. 死胡同孤岛：走过去之后把某个必须经过的格子堵死了（后面例 3《马走日》会用到）。



本页要点

判断公式：还需选 `r - step` 个数时，当前数 `v` 后面最多还有 `n - v` 个数，必须满足 `n - v >= r - step`！

剪枝 1 | 可行性剪枝代码与手算验证

```
void dfs(int step, int start) {
    if (step > r) { print_ans(); return; }      // 选满 r 个，打印并退出
    int limit = n - (r - step);                 // 后面还要选 r-step 个，最大只能到
    limit
    for (int v = start; v <= limit; ++v) {
        a[step] = v;                            // 选定第 step 个数为 v
        dfs(step + 1, v + 1);                   // 下一层从 v+1 开始选
    }
}
```

- 手算 $n = 5, r = 3$：第 1 层还需选 2 个，`limit = 5 - 2 = 3`，只试 1，2，3；试 v=4 的分支被整棵砍掉。
- 效果估算：$n = 25, r = 20$ 时，加上 `limit` 能直接砍掉绝大部分凑不够的死枝。

本页要点

把“剩余元素够不够”直接移项算成循环上限 `limit = n - (r - step)`，零开销！

剪枝 2 | 最优性剪枝：已经输了，后面全免费也赢不了

生活例子：从家去学校，已经找到一条只花 10 分钟的路线

现在你正在试另一条路，刚走到半路手表已经走了 11 分钟，而剩下的路不可能倒退时间（时间 ≥ 0）。你还用接着往学校走完再看时间吗？不用！立刻掉头！

- 核心写法：用全局变量 `best` 记住目前找到的最小总代价（初值设为极大值 `INF`）。
- 每当走到某个状态时已经花费 `cur`，只要 `cur >= best`（注意：后面每步代价必须 ≥ 0），立刻 `return`！
- 为什么写 `>=` 而不是 `>`？因为如果只求最小代价数值，等于 `best` 也无法更小，一起剪掉更省时间！

本页要点

前提条件：后续所有步骤的代价必须非负（≥ 0），才能写 `if (cur >= best) return;`！

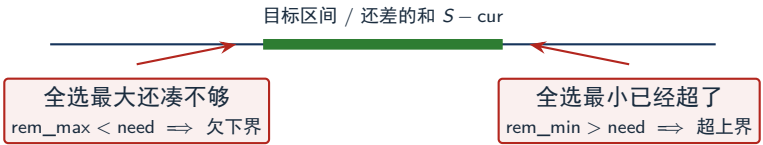
剪枝 2 | 代码框架、正确性论证与反事实追问

```
int best = 1e9;                                    // 已知最小代价，初值设为无穷大
void dfs(int step, int cur) {
    if (cur >= best) return;                       // 最优性剪枝：已不比 best 更优，直接退
    if (step > n) { best = cur; return; }          // 能到这里说明 cur < best，安全刷新最优值
    for (int nxt : choices(step)) {
        dfs(step + 1, cur + cost(step, nxt));      // 前提：cost >= 0 才不会越走越小
    }
}
```

- 为什么对：因为后续所有 `cost >= 0`，所以从当前节点出发到达任何叶子的总代价 $\geq$ cur $\geq$ best，绝对不可能产生更小的答案。
- 反事实追问：如果某步允许出现负代价（比如走这格能捡回 -5 元金币），还能写 `if (cur >= best) return;` 吗？
- 回答：绝对不能！现在 `cur=11` 看似输给了 `best=10`，但后面如果减掉 5 就变成 6 反超了，硬剪会直接 **WA**！

剪枝 3 | 上下界剪枝：用“最好情况”和“最坏情况”卡死范围

有时当前代价还没超标，但算一算“剩下的步数最多/最少能凑出多少”，照样能提前刹车！



- 小例子：用 4 个 $1 \sim 9$ 的整数凑和等于 25。第 1 个数选 1，还差 24；剩下 3 个数就算全选最大的 9 也才 $27 \geq 24$（行）；若第 2 个数也选 1，还差 23，剩 2 个数最大才 $9 + 9 = 18 < 23$，连上限都够不着，立刻砍掉！

本页要点

上下界剪枝比单纯看当前值更狠：把“剩余最好情况”和“最坏情况”都算进去一起判断！

剪枝 3 | 上下界剪枝手算跟踪表（凑和问题）

题设：选 $k=3$ 个互不相同的 $1\sim5$ 的数（按升序选），使总和恰好等于 $S=12$。跟踪前 5 步：

步序	已选序列	还需要和 need	剩余最大可能和	判断结果
1	[1]	$12-1=11$	后 2 个最大 $4+5=9$	$9<11$，不够下界，直接剪掉！
2	[2]	$12-2=10$	后 2 个最大 $4+5=9$	$9<10$，不够下界，直接剪掉！
3	[3]	$12-3=9$	后 2 个最大 $4+5=9$	$9\geq9$，放行继续往下试
4	[3，4]	$12-7=5$	后 1 个最大 5	$5\geq5$，且最小 $5\leq5$，放行
5	[3，4，5]	0	终止步 step > 3	命中合法解 3+4+5=12！

本页要点

原本要试 $1,2$ 开头的大量组合，靠一句“剩余最大可能和 $<$ need”一步到位全部跳过！

剪枝 3 | 上下界剪枝代码示范

```
// 用后缀和 suf_max[i] 表示从位置 i 到结尾全部取最大值能拿到的最大总和
void dfs(int idx, int picked, int cur_sum) {
    if (cur_sum > target) return;                  // 上界剪枝：已超目标和（非负数）
    if (cur_sum + suf_max[idx] < target) return;// 下界剪枝：全拿也不够 target
    if (picked == k) { if (cur_sum == target) ++ans; return; }
    if (idx > n) return;
    dfs(idx + 1, picked + 1, cur_sum + a[idx]); // 分支一：选 a[idx]
    dfs(idx + 1, picked, cur_sum);              // 分支二：不选 a[idx]
}
```

本页要点

预处理后缀和 `suf_max[idx]` 只需 $O(n)$，就能在每层 $O(1)$ 瞬间砍掉凑不够的枝！

剪枝 4 | 排除等效冗余与记忆化搜索

什么是“等效冗余”？大白话：两条路长得一样，只走一条就行！

- 情景 **A**（顺序冗余）：拼凑总和时，先拿 1 再拿 2 与先拿 2 再拿 1 结果一样。规定按编号递增顺序拿（传入 start），就能把 $k!$ 种顺序压成 1 种！
- 情景 **B**（同层相同值冗余）：数组 [1，1，2] 选数，同一层选第 1 个 1 和第 2 个 1 效果一样，只试第一个即可！

什么是“记忆化搜索”？大白话：踩过的坑记在小本本上

不同路径走到完全相同的状态（如同一个位置 idx 且剩余容量 rem 相同），之前算过答案是 X 就记入 memo[idx][rem]，下次撞见直接查表返回！

剪枝 4 | 写法 A：排序后同层跳过重复元素

```
sort(a + 1, a + n + 1);                              // 先把相同元素排在一起
void dfs(int step, int start, int sum) {
    for (int i = start; i <= n; ++i) {
        if (i > start && a[i] == a[i - 1]) continue; // 同层已试过相同值，跳过！
        dfs(step + 1, i + 1, sum + a[i]);
    }
}
```

- 为什么判断 `i > start` 而不是 `i > 1`？因为不同层可以分别选出两个 1（构成 `1 + 1`），只有同一层连续尝试两个一样的数才算等效冗余！

本页要点

“先排序，再在循环里加 `i > start && a[i] == a[i - 1]`”，是组合去重标准写法！

剪枝 4 | 写法 B：记忆化搜索查表返回

```
int solve(int idx, int rem) {
    if (idx > n) return rem == 0 ? 1 : 0;
    if (memo[idx][rem] != -1) return memo[idx][rem]; // 查小本本，算过直接给结果！
    int res = solve(idx + 1, rem);                   // 不选 idx
    if (rem >= w[idx]) res += solve(idx + 1, rem - w[idx]); // 选 idx
    return memo[idx][rem] = res;                     // 离开前把结果写进小本本
}
```

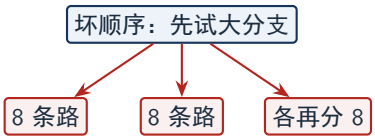
- 注意初始化：memo 必须初始化为不可能出现的值（如 -1），不要设成 0（因为方案数本身可能恰好等于 0，会导致重复白算）。

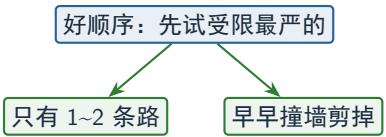
本页要点

记忆化要求结果只依赖当前参数（idx，rem），不能依赖外部走过的路径！

剪枝 5 | 搜索顺序优化：先挑分支少的走！

同样一棵搜索树，选先搜哪边，对剪枝效率影响巨大：





- 典型做法 **1**（数独/填格子）：优先填当前可填数字最少的格子（分支少，一错立马回溯）。
- 典型做法 **2**（木棍拼接/凑和）：把木棍从大到小排序（先放长的，很快就能触发超重或凑不出，不用在短木棍里打转）。

本页要点

优先决定“选择余地最小”或“体量最大”的分支，能把后面的可行性/最优性剪枝威力放大十倍！

上机验证与反事实追问：对比剪枝前后的调用次数

```cpp
#include <iostream>
using namespace std;
int n, r; long long cnt = 0;
void dfs(int step, int start, bool prune) {
    ++cnt; if (step > r) return;
    int lim = prune ? n - (r - step) : n;
    for (int v = start; v <= lim; ++v) dfs(step + 1, v + 1, prune);
}
int main() {
    cin >> n >> r;
    cnt = 0; dfs(1, 1, false); cout << "no_prune=" << cnt << ' ';
    cnt = 0; dfs(1, 1, true);  cout << "prune=" << cnt << '\n';
}
```

- 三组验证：5 3 → 26 vs 21；10 10 → 1024 vs 11；20 18 → 差出近万倍！
- 反事实追问：把 cur >= best 改成 cur > best，求最小代价不错但更慢；但若要求所有并列最优方案数，必须改用 cur > best 才不会漏解！

易错对照：五大剪枝最容易犯的“错剪”事故

剪枝类别	常见错误写法	后果与正确写法
可行性上界算错（**WA**）	组合写 `v <= n - r + step - 1`	少剪不致命，写小了直接漏答案！记住公式：剩 `r-step` 个位置，上限 `n-(r-step)`
负权误用最优剪枝（**WA**）	有减分项仍写 `if (cur >= best) return;`	后面可能拉回分数，错剪掉真正最优解
同层去重漏判起点（**WA**）	写成 `if (a[i] == a[i-1]) continue;`	忘了加 `i > start`，导致合法的重复数字全被跳过
等效顺序排序写反（**TLE**）	木棍拼凑题从小到大排序	短木棍组合无数，改成 `greater<int>()` 从大到小排

本页要点

每次加一条新剪枝，先在脑海里问一遍：“被我砍掉的那一侧，有没有哪怕 0.01% 可能藏着正确答案?”

例 1 | 组合的输出（一本通 1317）[入门]：先读懂要求

大白话题意

给你两个整数 n, r（$2 \le n \le 20,\ 1 \le r \le n$），要求用递归方法从 $1, 2, \ldots, n$ 中挑选 r 个数，按从小到大排列且全部组合按字典序输出。

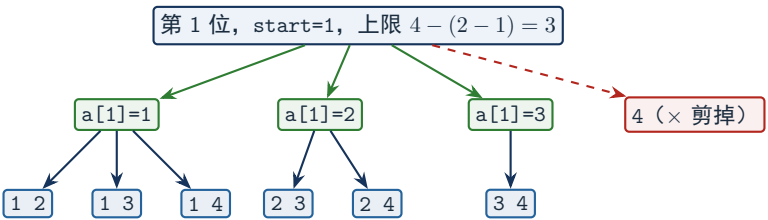
- 输入：一行两个自然数 n, r（如 `5 3`）。
- 处理：穷举所有递增的 r 元组合；由于组合不看顺序，固定后面选的数必须严格大于前面的数。
- 输出：每个组合占一行，每个数固定占 **3** 个字符宽、右对齐（例如 `5 3` 首行为 `" 1 2 3"`，共 10 行到 `" 3 4 5"`）。

本页要点

读题抓两个关键：一是“组合递增不重复”，二是“每个数字占 3 格右对齐”的格式要求！

例 1 | 怎么想到的？从暴力到可行性剪枝建模

如果不做任何优化，直接对 $1 \sim n$ 做“选与不选”需要搜到最底层再数是不是选了 r 个，还要调顺序。换个思路画搜索树（以 $n = 4, r = 2$ 为例）：



本页要点

每层只选比上一位大的数（从 `start` 开始），并把循环上限收紧到 `n - (r - step)`！

例 1 | 样例手推表（以 $n=4, r=2$ 完整走一遍）

步序	调用参数	上限 4-(2-step)	a[1..2]	动作说明
1	dfs(1, 1)	3	[_, _]	试 v=1，写 a[1]=1，调用 dfs(2, 2)
2	dfs(2, 2)	4	[1, _]	试 v=2，写 a[2]=2，调用 dfs(3, 3)
3	dfs(3, 3)	边界 step>2	[1, 2]	输出 1 2，返回后循环依次输出 1 3、1 4
4	回到第 1 层	3	[1, 4]	循环试 v=2，写 a[1]=2，进 dfs(2, 3) 输出 2 3、2 4
5	回到第 1 层	3	[2, 4]	循环试 v=3，写 a[1]=3，进 dfs(2, 4) 输出 3 4
6	回到第 1 层	3	[3, 4]	v=4 > 3 循环条件为假，终止结束

本页要点

第 6 步 v=4 直接被上限 3 挡在循环外，不用进递归就能结束！

例 1 | 核心步骤与相对模板改动

按顺序计算	纸上得到	核对问题
第 step 层确定循环上限	`n - (r - step)`	选完当前数后还剩几个更大数可选？
从 start 到上限试每个 v	`a[step] = v`	下一层起始参数是否传了 `v + 1`？
当 step > r 打印结果	每个数 `setw(3) << a[i]`	数字之间有没有多敲额外空格？

相对全排列模板改了哪里？

1. 删掉 vis 数组，改为给 dfs 增加参数 start，下一层传 `v + 1` 保证严格递增。
2. 递归出口从 `step > n` 改为 `step > r`；循环上限从 n 缩小为 `n - (r - step)`。

例 1 | 完整可编译代码（1/2）

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

// 一本通 1317 组合的输出：按递增顺序搜索 + 剩余名额可行性剪枝
int n, r, a[25];

void dfs(int step, int start) {
    if (step > r) {
        for (int i = 1; i <= r; ++i) cout << setw(3) << a[i]; // 每个数字固定占 3
    格右对齐
        cout << '\n';
        return;
    }
```

要点：`setw(3)` 只对紧跟在后的输出项生效，必须写进循环里！

例 1 | 完整可编译代码（2/2）

```
    for (int v = start; v <= n - (r - step); ++v) { // 剪枝：后面还需 r-step 个数，v
     太大必凑不够
        a[step] = v;
        dfs(step + 1, v + 1);
    }
}

int main() {
    ios::sync_with_stdio(false);
    cin.tie(nullptr);
    if (cin >> n >> r) dfs(1, 1);
    return 0;
}
```

要点：与 code/comb_output.cpp 完全一致，不需 vis 也不用手动还原 a[step]。

例 1 | 易错点与 3 组自测边界

- 易错 **1**（**PE** 格式错）：写成 `cout << setw(3) << a[i] << ' ';`，导致数字间出现 4 格宽度。正确做法是紧挨输出 `setw(3) << a[i]`。
- 易错 **2**（**WA** 漏解）：把上限误写成 `n - r + step - 1`，当 $n=5, r=3, \text{step}=1$ 时上限变成 2，会漏掉 `3 4 5`！正确公式是 `n - (r - step)`。

3 组必测边界数据

1. 最小边界：`2 1` → 两行：`" 1"` 和 `" 2"`。
2. 全选边界：`4 4` → 只有一行：`" 1 2 3 4"`。
3. 原题样例：`5 3` → 共 10 行，首行 `" 1 2 3"`，末行 `" 3 4 5"`。

例 2 | 八皇后（一本通 1214）[基础]：先读懂要求

大白话题意

在 8×8 的国际象棋棋盘上放 8 个皇后，要求任意两个皇后不能在同一行、同一列、同一条斜线上。把第 $1 \sim 8$ 行皇后所在的列号连成 8 位数串 $b_1 b_2 \ldots b_8$，已知共有 92 组解。给出 n 次询问，每次给序号 b，输出按数字从小到大排第 b 个皇后串。

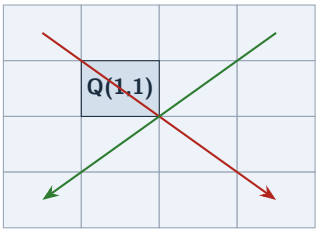
- 输入：第 1 行 n（$1 \le n \le 10^5$），后接 n 行正整数 b（$1 \le b \le 92$）。
- 处理：先用回溯搜出全部 92 个合法皇后串存起来，再按询问下标直接输出。
- 输出：n 行，每行一个 8 位数串（样例 `2 / 1 / 92` $\rightarrow$ `15863724` 和 `84136275`）。

本页要点

询问次数 n 高达 10^5，绝对不能每次询问重跑一次搜索，必须预处理打表一次搜完！

例 2 | 怎么想到的？按行搜索 + 对角线 $O(1)$ 编号法

为什么不从 64 个格子里选 8 格（$\binom{64}{8} \approx 4.4 \times 10^9$ 必超时）？因为每行恰好只能放 1 个皇后！所以第 `row` 层递归只管“第 `row` 行放在哪一列 `c`”。



列冲突：`used_col[c]` 标记这一列

副对角线（右上 ↔ 左下）：`row + c` 相等！

主对角线（左上 ↔ 右下）：`row - c` 相等！

为防下标变负，加偏移变成 `row - c + 7`（$0 \sim 14$）

本页要点

用 `used_col[c]`、`d1[row+c]`、`d2[row-c+7]` 三个数组，判断冲突只需 $O(1)$！

例 2 | 样例手推表（以 4×4 皇后求第 1 组解为例）

用 4×4 小棋盘（row，c 从 $0 \sim 3$，主对角线偏移 $+3$）手推找到第一组解的过程：

row	试列 c	占用的 col	冲突检查	结果与动作
0	0	{0}	均空	落子 $(0,0)$，进 row=1
1	$0,1,2$	{0,2}	0 同列，1 同主对角	$(1,2)$ 合法！落子进 row=2
2	0..3	无合法列	全被前面斜线/列挡住	死胡同！退回 row=1 撤销 $(1,2)$
1	3	{0,3}	合法	改放 $(1,3)$，进 row=2 放 $(2,1)$
3	0..3	退回重试	$(3,2)$ 冲突，退到 row=0	$(0,0)$ 全路不通，撤子改放 $(0,1)$
0..3	顺推	{1,3,0,2}	全部不冲突	搜到第 1 组解 **2413**（列号加 1）

本页要点

列 c 按 $0 \to 7$ 从小到大尝试，先搜出来的解数字天然从小到大排，不需要再排序！

例 2 | 核心步骤与相对模板改动

按顺序计算	纸上得到	核对问题
进入第 `row` 行扫列 `c=0..7`	判三数组是否全为 `false`	`d2` 下标有没有加上 `+ 7`?
落子占领三条线并递归	记下 `col[row] = c + 1`	题目列号从 1 开始还是从 0 开始?
`row == 8` 拼接 8 位串	存入 `sols` 并返回	退回时有没有把三数组全设回 `false`?

相对全排列模板改了哪里?

1. 判重由一维的 `vis[v]` 升级为“列 + 主对角线 + 副对角线”三个布尔数组联动占位与撤回。
2. 找到解后不直接打印，而是压入全局 `vector<string> sols`，实现 $O(1)$ 查表应答。

例 2 | 完整可编译代码（1/3）：状态数组与出口

```cpp
#include <bits/stdc++.h>
using namespace std;
// 一本通 1214 八皇后：列与两条对角线 O(1) 状态数组标记 + 回溯撤销现场
vector<string> sols;
int col[8];
bool used_col[8], d1[15], d2[15];
void dfs(int row) {
    if (row == 8) {
        string s;
        for (int i = 0; i < 8; ++i) s.push_back(char('0' + col[i]));
        sols.push_back(s);
        return;
    }
```

要点：与 code/eight_queens.cpp 一致；row+c 与 row-c+7 最大均为 14，开 15 够用。

例 2 | 完整可编译代码（2/3）：占领三线与恢复现场

接上一页 code/eight_queens.cpp 的 dfs 主体循环：

```
    for (int c = 0; c < 8; ++c) {
        if (used_col[c] || d1[row + c] || d2[row - c + 7]) continue; // 主/副对角线
    冲突
        used_col[c] = d1[row + c] = d2[row - c + 7] = true;          // 落子占领
        col[row] = c + 1;
        dfs(row + 1);
        used_col[c] = d1[row + c] = d2[row - c + 7] = false;         // 回溯必须恢
    复现场！
    }
}
```

本页要点

注意第 4 行的 c + 1：数组索引从 0 算起，但题目输出的皇后串列号从 1 ∼ 8 计！

例 2 | 完整可编译代码（3/3）：打表与 $O(1)$ 查询

接上一页 code/eight_queens.cpp 的 main 函数：

```
int main() {
    ios::sync_with_stdio(false);
    cin.tie(nullptr);
    dfs(0);
    int n, b;
    if (cin >> n) while (n-- && cin >> b) cout << sols[b - 1] << '\n';
    return 0;
}
```

本页要点

第 4 行先跑一遍 dfs(0) 生成全部 92 个解，后续哪怕有 10^5 次查询也能瞬间输出 sols[b-1]！

例 2 | 易错点与 3 组自测边界

- 易错 **1**（**RE** 数组负下标）：忘记加偏移直接写 `d2[row - c]`，当 `row=0`，`c=7` 时访问 `d2[-7]` 越界崩溃！
- 易错 **2**（**WA** 下标错位）：`sols` 下标从 0 开始而输入 b 从 1 开始，输出必须写 `sols[b - 1]`。
- 易错 **3**（**TLE** 重复搜索）：在 `while (n--)` 里面反复调用 `dfs(0)`，且忘了清空 `sols`（既超时又内存爆掉）。

3 组必测边界数据

1. 最小下标：`1 / 1` → `15863724`。
2. 最大下标：`1 / 92` → `84136275`。
3. 多组混合：`3 / 1 / 92 / 1` → 依次输出 `15863724`、`84136275`、`15863724`（验证多次查询无副作用）。

例 3 | 马走日（一本通 1219）[提高]：先读懂要求

大白话题意

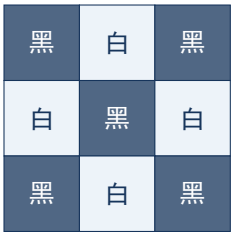
中国象棋里的马按“日”字走（横 1 竖 2 或横 2 竖 1，共 8 个方向）。现在给你 $n \times m$ 的棋盘（$1 \le n, m < 10$）和起点 (x, y)（坐标从 0 开始），要求每个格子恰好踩 **1** 次并走遍全部 $n \times m$ 个格子，求共有多少种不同的路线？

- 输入：第 1 行 T（$1 \le T < 10$）；每组一行四个整数 n, m, x, y。
- 处理：带回溯的网格 DFS 统计哈密顿路径总数，并用奇偶 + 孤岛剪枝防止超时。
- 输出：每组输出一个整数，不能走完输出 0（样例 `1 / 5 4 0 0` $\rightarrow$ 32）。

本页要点

9×9 棋盘若硬搜可达 8^{80} 次分支！这道题是学习“剪枝如何救命”的最佳战场。

例 3 | 怎么想到的？黑白奇偶染色 + 孤岛死点剪枝



3×3 棋盘：5 黑 4 白

奇偶染色剪枝：马走“日”字时横纵坐标之和改变 $\pm 1 \pm 2$（必定变奇偶！），即黑 $\leftrightarrow$ 白交替走。若 $n \cdot m$ 为奇数（如 3×3 有 5 黑 4 白），从少色格（白格，(sx+sy)%2 == 1）出发永远踩不完 5 个黑格，直接判 0！

孤岛死点剪枝：记录每个未走格子还剩几个未走邻居 deg[ax][ay]。刚跳进新格时更新周围邻居度数；若某个邻居度数降成 0 而全盘还剩 > 2 步没走，说明造出了永远进不去的“孤岛”，立刻撤回！

本页要点

一前一后两层剪枝：入搜前奇偶秒杀一半无解数据，搜途中孤岛剪枝砍掉大量必败子树！

例 3 | 样例手推表（3×3 奇偶剪枝与孤岛度数演示）

棋盘与起点	n*m%2	(sx+sy)%2	手推结论与度数变化
$3 \times 3,\ (0,1)$	1（奇）	1（白格）	白格只有 4 个，黑格有 5 个，交替走最多走 8 格，直接判 0！
$3 \times 3,\ (0,0)$	1（奇）	0（黑格）	从 $(0,0)$ 出发跳 $(1,2)$（step=2，remain=7）：中心 $(1,1)$ 无邻居可走不算，但若令某格 deg 降为 0 则立即剪枝
$5 \times 4,\ (0,0)$	0（偶）	0	黑白各 10 格，允许继续深搜；最终搜出 32 条完整路径

为什么孤岛判定写 `deg[ax][ay] == 0 && remain > 2`？

这里 `remain = n*m - step` 是在走完 (nx, ny) 之前剩余的步数：若 `remain == 2`，只剩最后一步就能踩进那个“没出路”的终点格（死胡同作为最后一格是合法的！），所以必须写 `remain > 2` 才剪！

例 3 | 核心步骤与相对模板改动

按顺序计算	纸上得到	核对问题
进入前做黑白奇偶判断	`n*m%2==0 || (sx+sy)%2==0`	(0,0) 的 sx+sy 奇偶性是 0 吗？
跳进 (nx,ny) 后减邻接度数	`--deg[ax][ay]`	发现 `deg==0 && remain>2` 是否设 `dead=true`？
递归返回后对称加回度数	`++deg[ax][ay]`	`vis[nx][ny]=false` 有没有写在恢复 deg 后？

相对全排列模板改了哪里？

1. 决策从一维循环 $1 \sim n$ 变成二维方向数组 `dx[8]`，`dy[8]`，状态标记升级为二维 `vis[x][y]`。
2. 除了恢复 `vis[nx][ny]`，还必须对刚修改过的邻居度数 `deg[ax][ay]` 对称执行 `++deg` 还原现场！

例 3 | 完整可编译代码（1/5）：定义与入口

```
#include <bits/stdc++.h>
using namespace std;
// 一本通 1219 马走日：奇偶染色剪枝 + 度数可行性剪枝（预防死胡同孤岛）
int n, m, sx, sy; long long ans;
bool vis[10][10]; int deg[10][10];
const int dx[8] = {1, 1, -1, -1, 2, 2, -2, -2};
const int dy[8] = {2, -2, 2, -2, 1, -1, 1, -1};
void dfs(int x, int y, int step) {
    if (step == n * m) { ++ans; return; }
    int remain = n * m - step;
```

本页要点

用 long long ans 存总方案数，避免较大棋盘上方案数突破 int 上限！（源码 code/knight_tour.cpp）

例 3 | 完整可编译代码（2/5）：尝试跳跃与孤岛检测

```
    for (int d = 0; d < 8; ++d) {
        int nx = x + dx[d], ny = y + dy[d];
        if (nx < 0 || nx >= n || ny < 0 || ny >= m || vis[nx][ny]) continue;
        vis[nx][ny] = true;
        bool dead = false;
        for (int e = 0; e < 8; ++e) {
            int ax = nx + dx[e], ay = ny + dy[e];
            if (ax < 0 || ax >= n || ay < 0 || ay >= m || vis[ax][ay]) continue;
            if (--deg[ax][ay] == 0 && remain > 2) dead = true; // 剪枝：死点
        }
        if (!dead) dfs(nx, ny, step + 1);
```

要点：即使 dead=true 也不能提前 continue，必须把刚才减掉的 deg 加回来！

例 3 | 完整可编译代码（3/5）：对称恢复度数与 main

```
        for (int e = 0; e < 8; ++e) {
            int ax = nx + dx[e], ay = ny + dy[e];
            if (ax < 0 || ax >= n || ay < 0 || ay >= m || vis[ax][ay]) continue;
            ++deg[ax][ay];
        }
        vis[nx][ny] = false;                          // 回溯清理当前步状态
    }
}
int main() {
    ios::sync_with_stdio(false); cin.tie(nullptr);
    int T; if (!(cin >> T)) return 0;
```

要点：恢复顺序与修改相反：先还原邻居 ++deg，再把 vis[nx][ny] 改回 false！

例 3 | 完整可编译代码（4/5）：奇偶剪枝与预处理 deg

```
while (T--) {
    cin >> n >> m >> sx >> sy;
    ans = 0;
    if (n * m % 2 == 0 || (sx + sy) % 2 == 0) { // 黑白染色奇偶剪枝
        memset(vis, 0, sizeof(vis));
        for (int i = 0; i < n; ++i)
            for (int j = 0; j < m; ++j) {
                deg[i][j] = 0;
                for (int d = 0; d < 8; ++d) {
                    int x = i + dx[d], y = j + dy[d];
                    if (x >= 0 && x < n && y >= 0 && y < m) ++deg[i][j];
                }
            }
```

要点：每组数据开头清空 ans 与 vis，重新统计每个格子的初始邻居度数 deg[i][j]。

例 3 | 完整可编译代码（5/5）：标记起点并输出结果

接上一页 code/knight_tour.cpp 结尾：

```
            vis[sx][sy] = true;
            for (int d = 0; d < 8; ++d) {
                int x = sx + dx[d], y = sy + dy[d];
                if (x >= 0 && x < n && y >= 0 && y < m) --deg[x][y];
            }
            dfs(sx, sy, 1);
        }
        cout << ans << '\n';
    }
    return 0;
}
```

本页要点

在调用 dfs(sx, sy, 1) 前，先把起点 vis[sx][sy] 标好并把起点邻居的 deg 减 1！

例 3 | 易错点与 3 组自测边界

- 易错 **1**（**WA** 漏还度数）：发现 `dead = true` 后立刻写 `vis[nx][ny] = false; continue;`，忘了把刚才循环里减掉的 `deg[ax][ay]` 加回去！
- 易错 **2**（**WA** 多组脏数据）：上一组留下的 `vis` 没 `memset` 清零，第二组直接变 0。
- 易错 **3**（**WA** 孤岛误判终点）：把 `remain > 2` 写成 `remain >= 2`，把合法的倒数第二步当成死路剪掉。

3 组必测边界数据

1. 最小 1×1 棋盘：`1 / 1 1 0 0` $\rightarrow$ `1`（初始 `step == n*m` 直接计 1）。
2. 奇偶无解：`1 / 3 3 0 1` $\rightarrow$ `0`（白格起点秒判无解）。
3. 原题样例：`1 / 5 4 0 0` $\rightarrow$ `32`。

DFS 回溯与剪枝高频错误对照表

题目 / 场景	常见错误写法	正确做法与原因
忘记恢复现场	`dfs` 返回后没置 `vis[x]=false`	搜完一条路径后别的分支无法再用该点，答案偏少 **WA**
多组数据忘清空	全局 `vis`，`ans` 沿用上一组脏数据	每组 `while(T--)` 前务必 `memset` 或清零
对角线下标为负	直接写 `d2[row - c]`	$row - c$ 可负至 -7 越界 **RE**，必须加偏移 `+ 7`
右对齐格式错误	《组合的输出》里自己额外加空格	直接用 `cout << setw(3) << a[i]`
求最短步硬套 DFS	用纯 DFS 不加限制找迷宫最少步数	DFS 首次到达不保证最短，搜到底必 **TLE**，请用下一章 BFS！

本页要点

“占坑与恢复必须成对出现”，对角线减法下标记得加偏移量保证非负！

常见错误：看懂评测结果再修改

提交代码看到 OJ 返回结果时，按下面顺序快速定位：

- 编译错误（**CE**）：检查头文件（如 `setw` 需要 `<iomanip>` 或万能头）、中文标点、分号与花括号成对。
- 答案错误（**WA**）：先手算最小样例；检查是否漏写“恢复现场”、多组数据是否清零、剪枝条件有没有把合法解误删（错剪）。
- 格式错误（**PE**）：检查行末是否多了空格或空行、`setw(3)` 后有没有重复加空格。
- 运行错误（**RE**）：检查数组是否越界（特别是 `row - c` 变负、`nx/ny` 未先判界就访问数组）、是否漏写递归出口导致爆栈。
- 运行超时（**TLE**）：检查是否漏了可行性/奇偶剪枝、或者把“求最少步数”的最短路题错用成了 DFS。

本页要点

定位口诀：**RE** 查越界与死递归，**TLE** 查剪枝与搜索方向，**WA** 查现场恢复与错剪！

提交前检查清单（六必查）

1. 递归出口（如 `if (step > target)`）是否放在函数最开头，且一定能触发 `return`？
2. 凡是在进入 `dfs` 前修改的全局状态（`vis`，`used_col`，`deg`），退出时有没有按相反顺序还原？
3. 网格与对角线下标计算（`nx`，`ny`，`row-c+n-1`）是否先判断了不越界、非负再读写数组？
4. 多组测试数据时，全局变量 `ans`，`vis`，`deg` 有没有在每组开始重置干净？
5. 加上的每一条剪枝，是否都 100% 确认被砍掉的分支里不可能有合法解（绝不错剪）？
6. 统计方案总数时（如《马走日》或大棋盘计数），结果变量是否已用 `long long`？

最后一步

造一组极小数据（如 $n = 1$）和一组连续多组输入样例跑一遍，确认第二组结果没受第一组残留污染再提交！

本章全部练习与完成标准

1. [入门] 组合的输出（一本通 1317，`code/comb_output.cpp`）：练递增传参 `start`、可行性上界与 `setw(3)`。
2. [入门] 自然数的拆分（一本通 1318）：练“不减拆分顺序去重”与输出格式控制。
3. [入门] 红与黑（一本通 1216）：练二维网格四连通 DFS 计数与多组输入清零。
4. [基础] 迷宫（一本通 1215）：练网格可达性判定、起点终点特判（遇 `'#'` 直接 `NO`）。
5. [基础] 八皇后（一本通 1214，`code/eight_queens.cpp`）：练按行搜索与主副对角线 $O(1)$ 编号。
6. [提高] 马走日（一本通 1219，`code/knight_tour.cpp`）：练黑白奇偶染色与度数孤岛剪枝。

完成标准

能不看模板默写“选与不选”和“全排列回溯”两套框架；能说清五大剪枝的适用条件与错剪禁忌；独立 AC 上述 6 题。

课后安排与下一章预告

- 复习：合上课件在草稿纸上画出 3 个数字的全排列搜索树，用红笔圈出每一步“恢复现场”的动作。
- 练习：独立复写 code/ 下 3 份源码；试着把《马走日》里的奇偶剪枝和孤岛剪枝分别删掉，实测运行时间差距。
- 下一章预告：第 2 章《BFS 最短路与双向 A*》——当题目问的不再是“有多少种走法”而是“最少走几步到达终点”时，为什么换用队列一层一层向外扩散就能一次精准命中！

本章一句话

一条路走到底、撞墙就撤回脚印改道，再用可行性与上下界提前刹车，就是搜索与剪枝的精髓。

谢谢！