第七章

分治算法

分开来求解，再合起来

HGNU-ACM 实验室算法组

第七章　截图配套：4 道例题，14 道练习题





本章目录 | 知识

分治算法入门

二分查找

二分答案

归并排序与逆序对

快速幂

本章目录 | 例题与练习

例 7.4：循环比赛日程表

例 7.5：取余运算（mod）

例 7.6：黑白棋子的移动

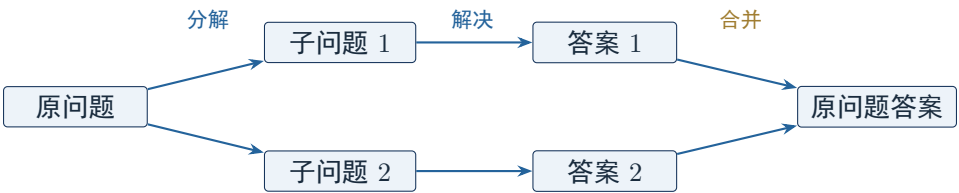
例 7.7：光荣的梦想

易错点与模板

全部练习索引

分治：把大问题拆成同样的小问题

- 分治（divide and conquer）：把大问题拆成若干更小的同类子问题。
- 分别解决这些子问题，再把子问题的答案合并成原问题的答案。
- 三个关键词：分解、解决、合并。



一句话理解

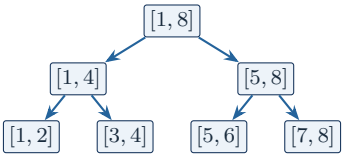
把“一口吃不下”的问题，切成几口能吃完的小块；吃完后再拼回原来的形状。

分解：每次分成规模接近的两半

- 对数组区间 $[l, r]$，取中点 $mid = \lfloor (l + r)/2 \rfloor$。
- 左子区间 $[l, mid]$，右子区间 $[mid + 1, r]$，两边长度最多差 1。
- 规模 $n = 8$ 时：$8 \to 4 + 4 \to 2 + 2 + 2 + 2 \to 1 + \cdots + 1$，只需 3 层。

原数组（区间 $[1, 8]$）

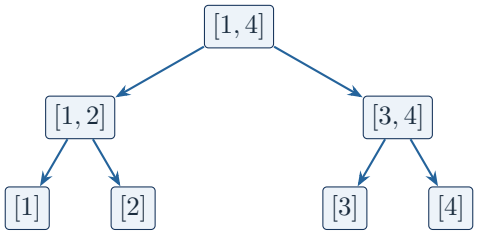




- 若分成 1 与 $n - 1$，递归树会退化成一条链，复杂度升到 $O(n^2)$。

解决：递归到足够小的子问题

- 边界：区间只剩一个元素时，答案就是它自己，直接返回。
- 递归：否则继续对左右两半调用同一个函数。
- 递归调用必须让区间长度严格变小，否则永远停不下来。



叶子：长度 1，直接求解

- 写递归函数时第一件事就是写边界；边界写错，后面的分解和合并都无从谈起。

合并：分治最关键的一步

- 子问题解决后，必须把两个有序段拼成一个有序段。
- 合并的代价决定总复杂度：归并的合并是 $O(n)$，总时间 $O(n \log n)$。
- 合并要满足三点：不重不漏、保留必要信息、代价可控。



分治执行全过程：求区间最大值

- 以数组 $a = [5, 2, 8, 1, 9, 3, 7, 4]$ 为例，求 $[1, 8]$ 的最大值。
- 先一路分解到单元素，再在返回时逐层合并。

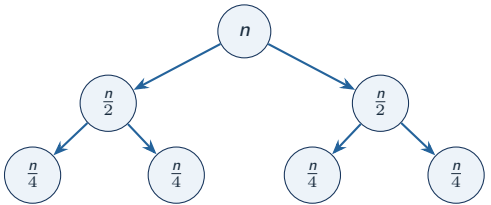
当前区间	分解结果	子问题答案	合并结果
$[1, 8]$	$[1, 4]$、$[5, 8]$	8、9	9
$[1, 4]$	$[1, 2]$、$[3, 4]$	5、8	8
$[1, 2]$	$[1, 1]$、$[2, 2]$	5、2	5
$[5, 8]$	$[5, 6]$、$[7, 8]$	9、7	9
$[5, 6]$	$[5, 5]$、$[6, 6]$	9、3	9

执行顺序

分解是 “往下走”，合并是 “往回走”；每个区间只在合并时计算一次。

递归树：每一层的工作量

- 规模为 n 的问题分成两个 $n/2$ 的问题，合并需要 $O(n)$。
- 递推式：$T(n) = 2T(n/2) + O(n)$。
- 递归树有 $\log_2 n + 1$ 层，每一层的总工作量都是 n。



本层共 n

本层共 n

本层共 n

共 $\log_2 n + 1$ 层

$$T(n) = n + n + \cdots + n = O(n \log n)$$

递归树：常见递推式对照

- 看递归树时抓两点：树有多高、每一层加起来多大。
- 两者相乘就是总工作量；若每层代价不同，取最大的一层做上界。

递推式	递归树形态	复杂度
$T(n) = T(n/2) + O(1)$	一条链，每层常数	$O(\log n)$
$T(n) = 2T(n/2) + O(1)$	满二叉树，共 n 个结点	$O(n)$
$T(n) = 2T(n/2) + O(n)$	每层 n，共 $\log n$ 层	$O(n \log n)$
$T(n) = T(n-1) + O(1)$	一条链，共 n 层	$O(n)$
$T(n) = 2T(n-1) + O(1)$	每层翻倍	$O(2^n)$

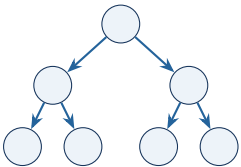
本章对应

二分查找是第一条，归并排序是第三条，快速幂是第一条（指数每层减半）。

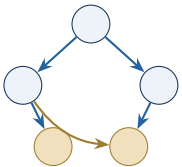
分治的适用条件

1. 可分解：原问题能拆成若干个同类型的子问题。
2. 子问题独立：左右两半互不影响，不会重复求解同一个子问题。
3. 可合并：两个子问题的答案能较快拼成原问题的答案。
4. 规模缩小：每次至少缩小一个常数比例，否则递归树会退化成链。

分治：子问题不重叠



动态规划：子问题重叠



同一个子问题被算两次

分治与动态规划的分界线

- 分治适合子问题互不重叠的场景；动态规划适合子问题大量重叠的场景。
- 判断方法：画出递归树，看同一个子问题是否出现多次。
- 若重复出现，就把子问题的答案存起来，避免重复计算，这就是记忆化搜索或动态规划。

	分治	动态规划
子问题	独立、不重叠	大量重叠
计算方向	自顶向下递归	常自底向上填表
中间结果	不保存	必须保存
典型例子	归并排序、快速幂	斐波那契、背包

一句话

分治是“各算各的，最后合并”；动态规划是“算过的不再算”。

分治的代码骨架

```
int solve(int l, int r) {                  // 当前区间 [l, r]
    if (l == r) {                          // 边界：一个元素
        return a[l];
    }
    int mid = l + (r - l) / 2;             // 分解
    int left = solve(l, mid);              // 解决左半
    int right = solve(mid + 1, r);         // 解决右半
    return max(left, right);               // 合并
}
```

- 先写边界：什么时候不用再分。
- 再写分：确定中点，保证两边规模接近。
- 最后写合：把两个子问题的答案拼起来。

分治的常见错误

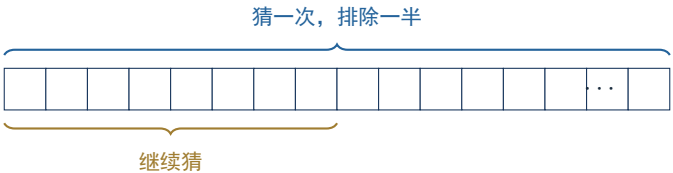
1. 忘记边界：递归一直进行，最后栈溢出。
2. 区间写错：左半写成 $[l, mid-1]$ 或右半写成 $[mid, r]$，导致漏元素或死循环。
3. 中点溢出：`mid = (l + r) / 2` 在 $l+r$ 超过 `int` 时出错。
4. 合并顺序错：先覆盖了原数组，再拿被覆盖的值去合并。
5. 子问题重叠：明明适合动态规划，却硬用分治，复杂度指数级。

自测方法

用长度 $0, 1, 2$ 的数组各跑一遍；再打印每次进入的 $[l, r]$，检查是否每个元素恰好出现一次。

二分查找：有序才能折半

- 前提：数据有序（或具有单调性），并且能随机访问中间元素。
- 直觉：猜数字 $1 \sim 100$，每次猜中间值，一次排除一半。
- 区间长度从 n 变成 $n/2, n/4, \ldots, 1$，最多 $\lceil \log_2 n \rceil$ 轮。
- 无序数组不能直接二分；链表不能随机访问，也不适合。

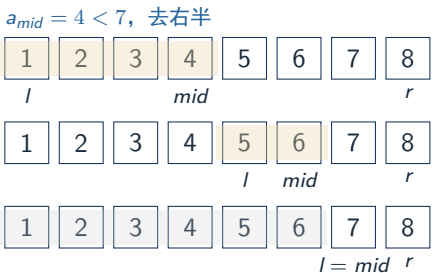


轮数

$n = 10^5$ 时 $\lceil \log_2 10^5 \rceil = 17$；$n = 10^9$ 时也只要 30 轮。

区间收缩：l、mid、r 怎么移动

- 始终维护一个答案一定在其中的区间 $[l, r]$。
- 看中点 a_{mid}：小于 x 去右半，大于 x 去左半，等于 x 直接返回。
- 每轮必须让区间严格变小，否则会死循环。



手推：在 8 个数中查找 7

- 数组 $a=[1,2,3,4,5,6,7,8]$，目标 $x=7$，下标从 1 开始。
- 闭区间模板，循环条件 $l \le r$。

轮次	l	r	mid	a_{mid}	下一步
1	1	8	4	4	$4<7$，令 $l=5$
2	5	8	6	6	$6<7$，令 $l=7$
3	7	8	7	7	相等，返回 7

对比

顺序查找要比较 7 次，二分只比较 3 次；数据越大，差距越明显。

闭区间模板：$[l, r]$

```cpp
int find(const vector<int>& a, int x) {
    int l = 0, r = (int)a.size() - 1;   // 闭区间 [l, r]
    while (l <= r) {
        int mid = l + (r - l) / 2;      // 防溢出
        if (a[mid] == x) return mid;    // 找到
        if (a[mid] < x) {
            l = mid + 1;                // 去右半
        } else {
            r = mid - 1;                // 去左半
        }
    }
    return -1;                          // 没找到
}
```

- 循环结束条件 $l > r$，此时区间为空。
- 返回 -1 表示不存在；若题目保证存在，可以省略。

闭区间的边界测试

- 二分最容易错在边界：长度为 0、1、2 时各手动跑一遍。
- 目标在第一个、最后一个、不存在，是三类必测情况。

数组	目标	正确结果	说明
$[]$	3	-1	$l=0, r=-1$，直接不进循环
$[5]$	5	0	第一轮命中
$[5]$	4	-1	第一轮后 $r=-1$
$[1,3]$	1	0	目标在第一个
$[1,3]$	3	1	目标在最后一个
$[1,3]$	2	-1	最后 $l>r$
$[2,2,2]$	2	任意一个	返回哪个都可以

检查口诀

先看空不空，再看两端，最后看不存在。

左闭右开模板：$[l, r)$

```
int lowerBound(const vector<int>& a, int x) {
    int l = 0, r = (int)a.size();       // 答案在 [l, r)
    while (l < r) {
        int mid = l + (r - l) / 2;
        if (a[mid] < x) {
            l = mid + 1;                // 丢弃左半
        } else {
            r = mid;                    // mid 可能是答案
        }
    }
    return l;                           // 第一个 >= x 的位置
}
```

- 初始 $r = n$，答案可能是 n，所以数组末尾要留一个“虚拟位置”。
- 循环结束时 $l = r$，这个位置就是第一个大于等于 x 的下标。

两种模板怎么选

- 两种写法都对，关键是全程统一，不要一半闭一半开。
- 求 “第一个满足条件的位置”，左闭右开更自然。
- 只求 “存在不存在”，闭区间更直观。

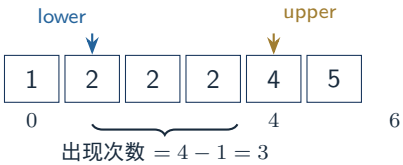
	闭区间	左闭右开
区间表示	$[l, r]$	$[l, r)$
初始 r	$n-1$	n
循环条件	$l \le r$	$l < r$
空区间	$l > r$	$l = r$
循环结束	可能 $l = r+1$	必有 $l = r$
返回	命中的下标或 -1	第一个可行位置

建议

初学阶段固定用一种模板，把边界测试表背下来，比记两种写法更可靠。

lower_bound 与 upper_bound

- `lower_bound(x)`：第一个 $\geq x$ 的位置。
- `upper_bound(x)`：第一个 $> x$ 的位置。
- 两者相减，就是 x 在有序数组中出现的次数。



写法

```
int p = lower_bound(a.begin(), a.end(), x) - a.begin();
```

upper_bound 的写法

```
int upperBound(const vector<int>& a, int x) {
    int l = 0, r = (int)a.size();
    while (l < r) {
        int mid = l + (r - l) / 2;
        if (a[mid] <= x) {
            l = mid + 1;                // 只丢弃 <= x
        } else {
            r = mid;                    // 第一个 > x
        }
    }
    return l;
}
```

- 与 lower_bound 只差一个等号：< 改成 <=。
- 重复元素较多时，二分仍然只要 $O(\log n)$。

边界与死循环

- `mid = (l + r) / 2` 向下取整；当区间只有两个元素时，`mid == l`。
- 若此时执行 `l = mid`，区间没有缩小，就会永远循环。
- 要么写 `l = mid + 1`，要么把中点改成向上取整。

场景	错误写法	正确写法
找左边界	`l = mid`	`r = mid` 或 `l = mid + 1`
找右边界	`r = mid`	`l = mid + 1`
向上取整	`(l + r) / 2`	`(l + r + 1) / 2`
防溢出	`(l + r) / 2`	`l + (r - l) / 2`

两元素测试

用 $[1,3]$ 查 3、查 1、查 2 各跑一遍；只要有一组死循环，就说明更新写错。

二分查找的四个常见错误

1. 对无序数组二分：先排序，或确认题目给出的是非降序列。
2. 死循环：更新区间时没有让 l 增大或 r 减小。
3. 漏掉端点：该用闭区间时写成开区间，最后一个元素永远查不到。
4. 整数溢出：`mid=(l+r)/2` 在 $l+r$ 超过 `int` 时出错。

检查方法

拿长度为 1、2 的数组，以及目标在第一个、最后一个、不存在这三种情况手动跑一遍。

- 若数组有重复元素，先想清楚要返回哪一个，再决定用 `lower_bound` 还是 `upper_bound`。

例：查找最接近的元素

```
int closest(const vector<int>& a, int x) {
    int p = lower_bound(a.begin(), a.end(), x)
            - a.begin();                    // 第一个 >= x
    if (p == 0) return a[0];                // 只能取第一个
    if (p == (int)a.size()) {
        return a.back();                    // 只能取最后一个
    }
    if (x - a[p - 1] <= a[p] - x) {
        return a[p - 1];                    // 距离相同取较小
    }
    return a[p];
}
```

- 候选只有两个：a_{p-1} 与 a_p。
- 距离相同取较小值，所以左边用 “$\leq$” 比较。
- 单次询问 $O(\log n)$，m 次询问共 $O(m \log n)$。

把求最值变成判可行

- 很多题要求“最大值最小”或“最小值最大”，直接构造很难。
- 若有一个判定函数 `check(x)`，它关于 x 单调，就可以二分 x。
- 这样把最优化问题变成了判定问题。
- 三步：确定答案范围、写 `check`、二分答案。

例子

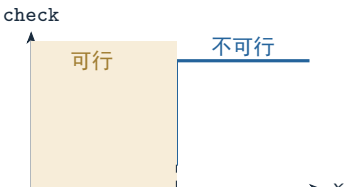
网线越长，能切出的段数越少；段数随长度单调不增，所以可以二分长度。

- 二分答案不是“二分数组”，二分的对象是答案本身。
- 答案范围通常由题目的上下界给出，例如 $[1, \max a_i]$。

单调性：两种方向

- 设 $x_1 < x_2$。若 check 满足 $\texttt{check}(x_1) \geq \texttt{check}(x_2)$，就是“真变假”。
- 反过来 $\texttt{check}(x_1) \leq \texttt{check}(x_2)$，就是“假变真”。
- 无论哪种，可行性只允许变化一次；这是二分答案成立的前提。





判定函数 check 怎么写

- `check(x)` 只能依赖 x 和题目输入，返回 `bool`。
- 常见形式：计数（能切几段）、贪心（要分几组）、模拟（能否跳过去）。
- 复杂度通常要 $O(n)$ 或 $O(n \log n)$，否则二分次数乘上去会超时。

写 check 的顺序

先问 “如果答案是 x，怎样验证最快”；再问 “x 变大时，验证结果会不会只变一次”。

- 反例：`check` 里又写了一个二分，复杂度变成 $O(n \log^2 n)$，多数题还能过，但要估算。
- 反例：`check` 依赖随机数或全局状态，结果不单调，不能二分。

整数二分答案模板（最小可行）

```
int minFeasible(int low, int high) {
    int l = low, r = high, ans = -1;
    while (l <= r) {
        int mid = l + (r - l) / 2;
        if (check(mid)) {                // mid 可行
            ans = mid;                   // 记录答案
            r = mid - 1;                 // 尝试更小
        } else {
            l = mid + 1;                 // 只能更大
        }
    }
    return ans;                          // 无解时为 -1
}
```

- ans 必须初始化；若 check 在整个区间都假，返回 -1。
- 循环条件与闭区间二分一致。

最小可行与最大可行

- 求“最小可行”：可行后记录答案，然后往小试，即 `r = mid - 1`。
- 求“最大可行”：可行后记录答案，然后往大试，即 `l = mid + 1`。
- 两者的 `check` 形状相反，更新方向也相反。

目标	可行时	不可行时
最小可行	`ans=mid; r=mid-1;`	`l=mid+1;`
最大可行	`ans=mid; l=mid+1;`	`r=mid-1;`

最大可行模板

```
if (check(mid)) { ans = mid; l = mid + 1; }
else { r = mid - 1; }
```

- 先判断题目要的是“最小”还是“最大”，再选模板；写反了会得到错误答案。

答案区间的上下界

- 下界 low 不一定要不可行；它只是答案的最小可能值。
- 上界 high 必须覆盖所有可能答案，否则会漏解。
- 对 “最小可行”，若 check(high) 为假，说明无解。



区间长度

二分轮数约为 $\log_2(\texttt{high} - \texttt{low} + 1)$；范围开大一点通常只多几轮。

网线主管：题意与判定

- 有 N 根网线，要切出至少 K 段等长网线，求每段的最大长度。
- 输入长度是两位小数，统一乘 100 换成整数厘米，避免浮点误差。
- 二分长度 x（厘米），判定：所有网线按 x 切，能否得到至少 K 段。
- 段数 $\sum_i \lfloor L_i/x \rfloor$ 随 x 增大而减少，满足单调性。

范围

$1 \le N, K \le 10000$，单根最长 100 千米 $= 10^7$ 厘米。二分约 $\log_2 10^7 \approx 24$ 轮，每轮 $O(N)$，完全够用。

- 若 1 厘米都切不出 K 段，输出 `0.00`。

网线主管：代码（读入与换算）

```cpp
#include <bits/stdc++.h>
using namespace std;
int main() {
    int n, k;
    cin >> n >> k;
    vector<long long> len(n);
    long long hi = 0;
    for (int i = 0; i < n; ++i) {
        double x;
        cin >> x;
        len[i] = llround(x * 100.0);    // 米转厘米
        hi = max(hi, len[i]);
    }
```

- 用 `llround` 四舍五入到整数厘米，不用 `(long long)(x*100)` 直接截断。
- `hi` 记录最长的一根，作为二分上界。

网线主管：代码（判定函数）

```
auto check = [&](long long x) {
    long long pieces = 0;
    for (long long L : len) {
        pieces += L / x;            // 每根能切几段
    }
    return pieces >= k;
};
```

- check(x) 统计所有网线在长度 x 下能切出的总段数。
- 段数随 x 增大而减少，所以 check 是“真变假”的单调函数。

网线主管：代码（二分与输出）

```
    long long lo = 1, ans = 0;
    while (lo <= hi) {
        long long mid = lo + (hi - lo) / 2;
        if (check(mid)) {
            ans = mid;                  // 长度可行
            lo = mid + 1;               // 试更长
        } else {
            hi = mid - 1;
        }
    }
    printf("%lld.%02lld\n", ans / 100, ans % 100);
    return 0;
}
```

- 这是“最大可行”模板：可行就记录，并往更长试。
- ans 初始为 0，切不出任何一段时输出 0.00。

河中跳房子：题意与贪心判定

- 要从起点 0 跳到终点 L，可以移走至多 M 块岩石。
- 二分最短跳跃距离 d，判定能否通过移走不超过 M 块岩石，使每步都 $\geq d$。
- 判定用贪心：从左往右扫描，若当前岩石与上一块保留岩石的距离 $< d$，就移走当前岩石。
- 为什么贪心正确：当前岩石离上一块太近，留着它只会让后面的选择更差。

端点

终点不能移走；若最后一段仍不足 d，只能继续移走前面的岩石。

- 二分下界 1，上界 L；答案至少为 1 也可以为 0，按题目要求处理。

河中跳房子：代码（读入坐标）

```cpp
#include <bits/stdc++.h>
using namespace std;
int main() {
    long long L;
    int n, m;
    cin >> L >> n >> m;
    vector<long long> p(n + 2);
    p[0] = 0;                              // 起点
    for (int i = 1; i <= n; ++i) {
        cin >> p[i];
    }
    p[n + 1] = L;                          // 终点
```

- 把起点和终点也放进数组，判定时不用特殊处理两端。
- p 的下标从 0 到 $n+1$，共 $n+2$ 个位置。

河中跳房子：代码（判定函数）

```
auto ok = [&](long long d) {
    int removed = 0, last = 0;
    for (int i = 1; i <= n; ++i) {
        if (p[i] - p[last] < d) {
            ++removed;                  // 移走当前岩石
        } else {
            last = i;                   // 保留，作为落脚点
        }
    }
    if (L - p[last] < d) {
        ++removed;                      // 最后一段也不够
    }
    return removed <= m;
};
```

- last 是上一块保留的岩石，不是上一块被移走的岩石。
- 最后一段不足 d 时，只能再移走一块前面的岩石，所以 removed 再加一。

河中跳房子：代码（二分答案）

```
    long long lo = 1, hi = L, ans = 0;
    while (lo <= hi) {
        long long mid = lo + (hi - lo) / 2;
        if (ok(mid)) {
            ans = mid;                    // 可行，试更大
            lo = mid + 1;
        } else {
            hi = mid - 1;
        }
    }
    cout << ans << '\n';
    return 0;
}
```

- 这是“最大可行”模板：可行就记录并往大试。
- 若题目允许答案为 0，把 `lo` 改成 0，并把 `ans` 初始化为 0。

月度开销：最小化最大值

```
bool check(long long x) {
    long long sum = 0;
    int cnt = 1;                          // 至少分一段
    for (int i = 0; i < n; ++i) {
        if (sum + a[i] <= x) {
            sum += a[i];                  // 并入当前段
        } else {
            ++cnt;                        // 新开一段
            sum = a[i];
        }
    }
    return cnt <= m;
}
```

- 把 N 天按顺序分成 M 段，使各段和的最大值最小。
- 二分这个最大值 x；下界是单天最大开销，上界是全部开销之和。
- 每天必须完整归入某一段，所以 x 不能小于某一天的开销。

浮点二分：固定循环次数

```
double solve(double low, double high) {
    for (int it = 0; it < 100; ++it) {
        double mid = (low + high) / 2.0;
        if (check(mid)) {
            high = mid;                  // 可行，往小试
        } else {
            low = mid;                   // 不可行，往大试
        }
    }
    return low;
}
```

- 实数答案不能靠 $l < r$ 停止，通常固定循环 100 次。
- 区间长度每轮减半，100 轮后缩小到 2^{-100}，远超题目精度。
- 最后输出 `low` 或 `high` 都可以，两者已经非常接近。

浮点二分的三个典型题

- 一元三次方程：在 $[-100, 100]$ 的每个单位区间检查端点符号，变号就二分。
- 膨胀的木棍：设圆心角一半为 θ，由 $\theta / \sin\theta = L'/L$ 二分 θ，再算中心偏移。
- 函数的零点：已知 $f(1.5) > 0$、$f(2.4) < 0$，直接二分 $[1.5, 2.4]$。

共同点

先找一个包含答案且端点异号的区间，再固定次数二分；最后按题目要求保留小数位。

- 端点异号是二分的前提；若区间内有两个零点，端点可能同号，需要先分区间。

二分答案的常见错误

1. 单调性判断错：check 随 x 不是只变一次，二分结果毫无意义。
2. 上下界漏解：答案可能等于上界，但上界取小了。
3. 更新方向写反：求最小可行却用了最大可行的模板。
4. check 太慢：每轮 $O(n^2)$，总复杂度过高。
5. 整数溢出：`mid=(l+r)/2`、`sum+a[i]` 都可能超过 `int`。
6. 浮点精度：循环次数太少，或输出时四舍五入方式不对。

自测

把答案调到边界值、无解值、刚好可行值各测一次，再和暴力程序对拍小数据。

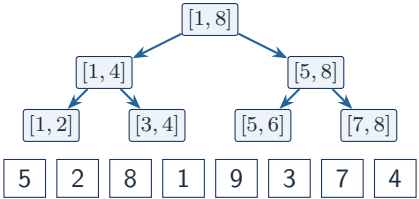
归并排序：先分开，再合并

- 把数组从中间分成两半，分别排序；
- 再把两个有序段合并成一个有序段。
- 合并时只用两个指针从头扫描，$O(n)$ 完成。



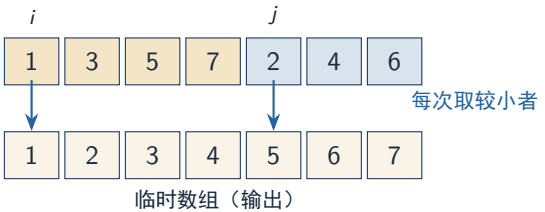
分割：递归把区间对半

- 分割只负责 “把区间变小”，不改变元素顺序。
- 递归到区间长度 1 时天然有序，开始往回合并。
- 分割树是一棵满二叉树，叶子有 n 个，树高约 $\log_2 n$。



合并：两个有序段怎么并

- 两个指针 i, j 分别指向左右段的首元素，k 指向临时数组。
- 每次取较小的那个放进临时数组；相等时先取左边，保证稳定。
- 某一段取完后，把另一段剩下的元素直接接上。



关键

左、右段本身已经有序，所以只需要看两个队首，不需要回头比较。

合并两个有序段：merge 函数（1/2）

```
void mergeArray(int l, int mid, int r) {
    int i = l, j = mid + 1, k = l;
    while (i <= mid && j <= r) {
        if (a[i] <= a[j]) {
            tmp[k++] = a[i++];          // 取左边，保持稳定
        } else {
            tmp[k++] = a[j++];
        }
    }
```

- 两个指针从各自段首出发，比较两个队首。
- 相等时取左边，保证排序稳定。

合并两个有序段：merge 函数（2/2）

```
    while (i <= mid) {
        tmp[k++] = a[i++];              // 左段剩余
    }
    while (j <= r) {
        tmp[k++] = a[j++];              // 右段剩余
    }
    for (int t = l; t <= r; ++t) {
        a[t] = tmp[t];                  // 拷回原数组
    }
}
```

- 某段先空后，把另一段剩余元素直接接上。
- 最后拷回原数组，供上一层继续合并。

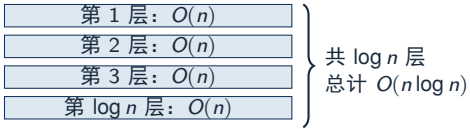
归并排序：递归与主函数

```
void mergeSort(int l, int r) {
    if (l >= r) return;                    // 不足两个元素
    int mid = l + (r - l) / 2;
    mergeSort(l, mid);                     // 排左半
    mergeSort(mid + 1, r);                 // 排右半
    mergeArray(l, mid, r);                 // 合并
}
int main() {
    int n;
    cin >> n;
    a.resize(n);
    tmp.resize(n);
    for (int &x : a) cin >> x;             // 读入
    mergeSort(0, n - 1);
    for (int x : a) cout << x << ' ';      // 输出
    return 0;
}
```

- 递归式 $T(n) = 2T(n/2) + O(n)$，解得 $O(n \log n)$。

归并排序：复杂度、稳定性与适用

- 递归深度 $O(\log n)$，每层合并总量 $O(n)$，总时间 $O(n \log n)$。
- 需要长度为 n 的临时数组，额外空间 $O(n)$。
- 合并时相等元素先取左边，所以归并排序是稳定的。
- 适合：求逆序对、外部排序、需要稳定排序的场合。



与快速排序对比

归并排序最坏也是 $O(n \log n)$，但多花 $O(n)$ 空间；快速排序平均快，最坏会退化。

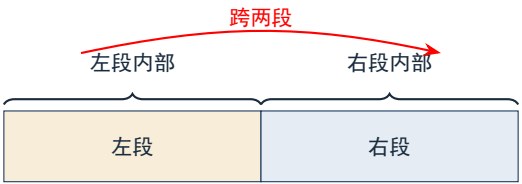
逆序对：定义与朴素统计

```cpp
long long countNaive(vector<int>& a) {
    long long cnt = 0;
    int n = (int)a.size();
    for (int i = 0; i < n; ++i) {
        for (int j = i + 1; j < n; ++j) {
            if (a[i] > a[j]) {
                ++cnt;                  // 发现一个逆序对
            }
        }
    }
    return cnt;
}
```

- 逆序对：$i < j$ 且 $a_i > a_j$。$[2, 1, 4, 3]$ 有两个逆序对。
- 朴素做法枚举所有数对，时间 $O(n^2)$；$n = 10^5$ 时约 5×10^9 次，太慢。

分治统计三类逆序对

- 把数组分成左右两段后，逆序对恰好分成三类：左段内部、右段内部、跨两段。
- 前两类在递归中已经统计；当前层只需要数跨段的。
- 这样每个逆序对只在“它跨越的那一层”被数一次，不重不漏。

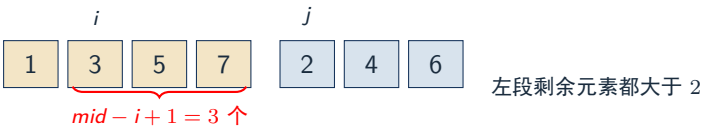


递归顺序

先递归左右两段，把内部逆序对数完；再在当前层统计跨段，最后合并。

跨段逆序对：一次统计一批

- 合并时若 $a_i > a_j$，因为左段已经有序，左段从 i 到 mid 的所有元素都大于 a_j。
- 所以一次产生 $mid - i + 1$ 个逆序对，直接累加，不用逐个枚举。
- 若 $a_i \le a_j$，不产生逆序对，正常取左边。



记忆

`ans += mid - i + 1;` 只在取右段元素时执行；取左段元素时不计。

归并求逆序对：代码（递归与统计）

```
long long ans = 0;
vector<int> a, tmp;
void mergeSort(int l, int r) {
    if (l >= r) return;                    // 边界
    int mid = l + (r - l) / 2;
    mergeSort(l, mid);                     // 左段内部
    mergeSort(mid + 1, r);                 // 右段内部
    int i = l, j = mid + 1, k = l;
    while (i <= mid && j <= r) {
        if (a[i] <= a[j]) {
            tmp[k++] = a[i++];
        } else {
            ans += mid - i + 1;            // 跨段逆序对
            tmp[k++] = a[j++];
        }
    }
```

归并求逆序对：代码（收尾）

```
    while (i <= mid) {
        tmp[k++] = a[i++];
    }
    while (j <= r) {
        tmp[k++] = a[j++];
    }
    for (int t = l; t <= r; ++t) {
        a[t] = tmp[t];                    // 拷回
    }
}
```

- 收尾阶段不再产生逆序对，因为其中一段已经为空。
- 拷贝回原数组，保证上一层拿到的是有序段。

归并求逆序对：代码（主函数）

```
int main() {
    int n;
    cin >> n;
    a.resize(n);
    tmp.resize(n);
    for (int &x : a) cin >> x;
    mergeSort(0, n - 1);
    cout << ans << '\n';
    return 0;
}
```

- 答案用 `long long`：$n = 10^5$ 时逆序对最多约 5×10^9。
- 数组下标采用闭区间 $[0, n-1]$，与 `mergeSort(0,n-1)` 对应。

逆序对：复杂度与易错点

- 时间 $O(n \log n)$，空间 $O(n)$。
- 相等元素不算逆序：合并时用 `a[i] <= a[j]` 先取左边。
- 统计式 `ans += mid - i + 1`：左段剩余元素个数，不能写成 `mid-i`。
- 递归区间是闭区间时，`mid+1` 才是右段起点；写成左闭右开时对应 `mid`。

检查

升序数组答案为 0；降序数组答案为 $n(n-1)/2$；所有元素相等时答案为 0。

朴素幂太慢

- 计算 b^p 的朴素做法要乘 p 次，$p = 10^9$ 时无法接受。
- 把指数写成二进制：$p = (1010)_2$，则 $b^{10} = b^8 \cdot b^2$。
- 只需不断平方：$b^1, b^2, b^4, b^8, \ldots$，再按二进制位相乘。
- 乘法次数从 $O(p)$ 降到 $O(\log p)$。

例子

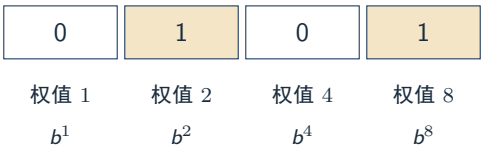
$2^{10} = 2^8 \cdot 2^2 = 256 \cdot 4 = 1024$，只做了 4 次平方和 2 次乘法。

- 指数每右移一位，底数就平方一次；二进制位为 1 时把当前底数乘进答案。

指数拆成二进制

- $p = 10 = (1010)_2$，从低位到高位是 $0, 1, 0, 1$。
- 对应权值 $1, 2, 4, 8$；只有权值 2 和 8 的位为 1。
- 所以 $b^{10} = b^8 \cdot b^2$。

二进制位（从高位到低位）



金色表示该位为 1，对应幂乘入答案

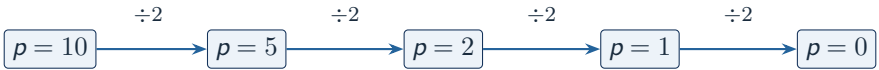
结果

$b^{10} = b^8 \cdot b^2$；没有被选中的 b^1, b^4 只用于平方，不乘进答案。

折半递推：把指数除以 2

- 设 $f(b,p) = b^p \bmod k$。
- 若 p 为偶数：$b^p = (b^{p/2})^2$，只需求一次 $b^{p/2}$。
- 若 p 为奇数：$b^p = (b^{\lfloor p/2 \rfloor})^2 \cdot b$。
- 边界：$p = 0$ 时 $b^0 = 1$，答案为 $1 \bmod k$。

$$f(b,p) = \begin{cases} 1 \bmod k, & p = 0, \\ f(b,p/2)^2 \bmod k, & p \text{ 为偶数}, \\ f(b,p/2)^2 \cdot b \bmod k, & p \text{ 为奇数}. \end{cases}$$



递归深度 $O(\log p)$

递归快速幂

```
long long power(long long b, long long p, long long mod) {
    if (p == 0) {
        return 1 % mod;                  // 边界
    }
    long long t = power(b, p / 2, mod);
    t = t * t % mod;                     // 平方
    if (p & 1) {
        t = t * b % mod;                 // 奇数补乘 b
    }
    return t;
}
```

- 递归式与 “分治三步” 完全对应：先求一半，再平方，最后补上奇数次幂。
- 以 $2^{10} \bmod 9$ 为例，递归链为 $10 \to 5 \to 2 \to 1 \to 0$。

迭代快速幂

```
long long power(long long b, long long p, long long mod) {
    long long ans = 1 % mod;
    b %= mod;
    while (p > 0) {
        if (p & 1LL) {
            ans = ans * b % mod;          // 这一位是 1
        }
        b = b * b % mod;                  // 底数平方
        p >>= 1;                          // 指数右移
    }
    return ans;
}
```

- 循环不变式：ans 乘上 b 的 p 次方，始终等于最初的 b^{p_0}（模意义下）。
- p 为偶数时只平方；p 为奇数时先乘一次再平方。

快速幂的手推过程

- 计算 $2^{10} \bmod 9$，初始 `ans=1`、`b=2`、`p=10`。
- 每轮看 `p` 的最低位，为 1 就乘进答案，然后平方底数、右移指数。

轮次	p	$p \bmod 2$	`ans`	`b`	动作
0	10	0	1	2	只平方
1	5	1	2	4	乘入，再平方
2	2	0	2	7	只平方
3	1	1	7	4	乘入，再平方
4	0	–	7	7	结束

验证

$2^{10} = 1024$，$1024 \bmod 9 = 7$，与表中结果一致。

取模、溢出与负底数

- `ans = 1 % mod`：保证 `mod=1` 时答案为 0。
- `b %= mod`：先缩小底数，避免平方时过大。
- $b \cdot b$ 可能超过 `long long`，用 `__int128` 中转。

安全乘法

```
long long mul(long long a, long long b,
  long long m) {
  return (long long)((__int128)a * b % m);
}
```

- 底数为负时，先做 `b %= mod`，若仍为负再加 `mod`。
- 指数是 `long long` 时，循环条件写 `p > 0`，不要写 `p != 0` 再右移负数。

超大指数：按十进制读入

```
int main() {
    long long b, k;
    string s;
    cin >> b >> s >> k;                    // p 以字符串读入
    long long ans = 1 % k;
    for (char c : s) {
        ans = power(ans, 10, k);           // 前缀整体乘 10 次方
        ans = ans * power(b, c - '0', k) % k;
    }
    cout << ans << '\n';
    return 0;
}
```

- p 可能有 200 位，不能存进 `long long`。
- 递推：$b^{10x+d} = (b^x)^{10} \cdot b^d$，其中 d 是当前十进制位。
- 这就是例 7.5 与练习 “2011” 的共同思路。

快速幂的常见错误

1. `p=0`：应返回 $1 \bmod k$；`k=1` 时返回 0。
2. `k=1`：任何数模 1 都是 0，初始化 `ans=1%k` 正好处理。
3. 乘法溢出：`b*b` 可能超过 `long long`，用 `__int128`。
4. 底数为负：先取模再调整到 $[0, k)$。
5. 指数读入：超过 `long long` 时要用字符串。
6. 输出格式：例 7.5 要求 `b^p mod k=ans`，空格和等号位置必须一致。

例 7.4 | 题意与样例

题意

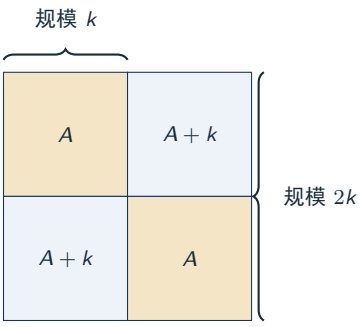
有 $N = 2^M$ 名选手进行循环赛。每名选手每天比赛一次，共进行 $N-1$ 天，每天没有选手轮空。输出 $N \times N$ 的比赛安排表，第 i 行第 j 个数表示第 i 名选手在第 j 天的对手。

- 输入 M，样例 $M = 3$，即 $N = 8$；下表只画左上角 4×4。
- 样例第一行：$1, 2, \ldots, 8$，表示第 1 名选手依次对阵 $2 \sim 8$ 号。
- 表格关于对角线对称；每行是 $1 \sim N$ 的一个排列。

行/列	1	2	3	4
1	1	2	3	4
2	2	1	4	3
3	3	4	1	2
4	4	3	2	1

例 7.4 | 从 2×2 拼到 $2^M \times 2^M$

- 已知规模 k 的日程表 A，要构造规模 $2k$ 的表。
- 右下角复制 A；右上角、左下角都填 $A + k$。
- 为什么正确：前 k 名与后 k 名在新增的 k 天里互相交手，编号整体偏移 k。
- 从 1×1 的 $A = [1]$ 开始，不断翻倍，共 $\log_2 N = M$ 轮。



例 7.4 | 构造步骤与正确性

- 当前已知左上角 $k \times k$ 的日程表，其余三块按规则填。
- 左上、右下两块相同，表示两组选手各自内部的比赛安排一致。
- 右上、左下两块为 $A + k$，表示跨组比赛，对手编号加上 k。
- 每填一轮，规模翻倍；填完 M 轮得到 $N \times N$ 表。

为什么每行是排列

每一轮新增的 k 天里，前 k 名选手分别对阵后 k 名选手，编号互不相同；组内比赛又由归纳假设保证。

- 下标从 0 开始：$a[0][0] = 1$ 对应第 1 名选手。

例 7.4 | 代码（构造）

```cpp
#include <bits/stdc++.h>
using namespace std;
int main() {
    int m;
    cin >> m;
    int n = 1 << m;
    vector<vector<int>> a(n, vector<int>(n, 0));
    a[0][0] = 1;                          // 规模 1 的表
    for (int k = 1; k < n; k <<= 1) {
        for (int i = 0; i < k; ++i) {
            for (int j = 0; j < k; ++j) {
                a[i + k][j] = a[i][j] + k;      // 左下
                a[i][j + k] = a[i][j] + k;      // 右上
                a[i + k][j + k] = a[i][j];      // 右下
            }
        }
    }
```

- 三个赋值顺序不能乱：右上、左下用旧值加 k，右下复制旧值。

例 7.4 | 代码（输出）

```
    for (int i = 0; i < n; ++i) {
        for (int j = 0; j < n; ++j) {
            if (j > 0) {
                cout << ' ';
            }
            cout << a[i][j];
        }
        cout << '\n';
    }
    return 0;
}
```

- 同一行数字之间一个空格，行末不能多输出空格。
- 输出本身是 $O(N^2)$，与构造同阶。

例 7.4 | 复杂度与易错点

- 构造需要 $O(N^2)$ 次赋值，输出本身也是 $O(N^2)$，总复杂度 $O(N^2)$。
- 空间 $O(N^2)$；N 较大时输出文件本身就会很大。
- 下标从 0 开始，$a[0][0] = 1$ 对应第 1 名选手。
- 三个赋值块的顺序不能乱：右上、左下用旧值加 k，右下复制旧值。
- 输出格式：同一行数字之间一个空格，行末不能多输出空格。

检查

用 $M = 1, 2, 3$ 手算，确认每行是 $1 \sim N$ 的排列，且 $a[i][j] = a[j][i]$。

例 7.5 | 题意与样例手推

题意

输入 b, p, k，求 $b^p \bmod k$。b, p, k 均为长整型数。输出格式为 `b^p mod k=ans`。

- 样例：$b = 2, p = 10, k = 9$，输出 `2^10 mod 9=7`。
- 手推：$2^1 = 2$，$2^2 = 4$，$2^3 = 8$，$2^4 = 16 \equiv 7$，$2^5 \equiv 5$，$2^{10} \equiv 5^2 = 25 \equiv 7$。
- 直接乘 10 次可以，但 p 可能很大，必须用快速幂。

输出细节

等号左边是原始的 b, p, k，不是取模后的值；$k = 1$ 时答案为 0。

例 7.5 | 分治折半

- 设 $f(b,p) = b^p \bmod k$。
- 若 p 为偶数：$b^p = (b^{p/2})^2$，只需求一次 $b^{p/2}$。
- 若 p 为奇数：$b^p = (b^{\lfloor p/2 \rfloor})^2 \cdot b$。
- 边界：$p = 0$ 时 $b^0 = 1$，答案为 $1 \bmod k$。

$$f(b,p) = \begin{cases} 1 \bmod k, & p = 0, \\ f(b,p/2)^2 \bmod k, & p \text{ 为偶数}, \\ f(b,p/2)^2 \cdot b \bmod k, & p \text{ 为奇数}. \end{cases}$$

复杂度

递归深度 $O(\log p)$，每层常数次乘法，总时间 $O(\log p)$。

- 递归版和迭代版结果相同；迭代版不需要递归栈，更适合 p 很大的情况。

例 7.5 | 完整代码（乘法与初始化）

```cpp
#include <bits/stdc++.h>
using namespace std;
using i128 = __int128_t;
long long mul(long long a, long long b, long long m) {
    return (long long)((i128)a * b % m);
}
int main() {
    long long b, p, k;
    cin >> b >> p >> k;
    long long base = b % k;
    long long ans = 1 % k;
    long long e = p;
```

- mul 用 __int128 中转，防止 $a \cdot b$ 溢出。
- base、ans 先取模；e 是不断右移的指数。

例 7.5 | 完整代码（迭代与输出）

```
    while (e > 0) {
        if (e & 1LL) {
            ans = mul(ans, base, k);
        }
        base = mul(base, base, k);
        e >>= 1;
    }
    cout << b << '^' << p << " mod " << k << '=' << ans << '\n';
    return 0;
}
```

- 指数最低位为 1 时，把当前底数乘进答案；随后底数平方、指数右移。
- 输出等号左边是原始的 b, p, k，不是取模后的值。

例 7.5 | 易错点

1. $p = 0$：$b^0 = 1$，答案应为 $1 \bmod k$；$k = 1$ 时为 0。
2. $k = 1$：任何数模 1 都是 0，初始化 `ans=1%k` 正好处理。
3. 底数为负：先做 $b\%k$，若为负再加 k，保证余数非负。
4. 乘法溢出：$b * b$ 可能超过 `long long`，用 `__int128` 或先取模。
5. 输出格式：`b^p mod k=ans`，空格和等号位置必须与样例一致。

自测

$p = 0$、$k = 1$、$b = 0$、p 接近 10^{18} 各测一组。

例 7.6 | 题意与目标

- $2n$ 个棋子（$n \geq 4$）：左边 n 个白子，右边 n 个黑子，末尾两个空位。
- 每次把相邻两个棋子整体移到空位，不能交换左右顺序，必须跳过若干棋子。
- 目标：变成黑白相间的一行。



$n = 5$ 初始



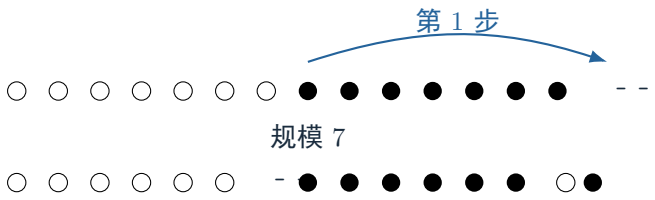
目标：黑白相间

关键

每次移动后，空位的位置会改变；要把规模 n 的局面化成规模 $n-1$ 的局面。

例 7.6 | 把规模 n 降到 $n-1$

- 第 1 步：把交界处的 “白黑” 一对移到末尾空位，空位来到交界处。
- 第 2 步：把最右边的一对棋子移到刚才的空位。
- 此时前 $2(n-1)$ 个位置恢复成 “$n-1$ 个白子 + $n-1$ 个黑子”，末尾留出两个空位。
- 后面只需递归解决规模 $n-1$；规模 4 时用固定的 5 步收尾。



第 2 步后：前 12 位是规模 6 的形状

例 7.6 | 代码（输出与移动）

```
#include <bits/stdc++.h>
using namespace std;
int n, stepNo;
string s;
void printStep() {
    printf("step%2d:", stepNo);
    for (int i = 1; i <= 2 * n + 2; ++i) {
        putchar(s[i]);
    }
    putchar('\n');
}
void movePair(int from, int to) {
    s[to] = s[from];
    s[to + 1] = s[from + 1];
    s[from] = s[from + 1] = '-';
    ++stepNo;
    printStep();
}
```

例 7.6 | 代码（递归主体）

```
void solve(int k) {
    if (k == 4) {
        movePair(4, 9);
        movePair(8, 4);
        movePair(2, 8);
        movePair(7, 2);
        movePair(1, 7);
        return;
    }
    movePair(k, 2 * k + 1);
    movePair(2 * k - 1, k);
    solve(k - 1);
}
```

- $k = 4$ 是递归出口，用固定的 5 步完成。
- 另外两步把规模 k 化成规模 $k - 1$，然后递归。

例 7.6 | 代码（主函数）

```
int main() {
    cin >> n;
    s.assign(2 * n + 3, '-');
    for (int i = 1; i <= n; ++i) {
        s[i] = 'o';
    }
    for (int i = n + 1; i <= 2 * n; ++i) {
        s[i] = '*';
    }
    stepNo = 0;
    printStep();
    solve(n);
    return 0;
}
```

- 字符串下标从 1 开始，末尾多留两个空位。
- 初始局面也要输出，所以先调用一次 `printStep()`。

例 7.6 | 复杂度与易错点

- 规模每降一层用 2 步，规模 4 用 5 步；总步数约 $2(n-4)+5=2n-3$ 步。
- 每步输出 $2n+2$ 个字符，总输出 $O(n^2)$，这是题目要求的输出量。
- 递归深度 $O(n)$；n 很大时要注意栈深度，可改成循环。
- 样例 $n=7$ 共 12 行（`step 0` 到 `step11`），最后一行为 `--o*o*o*o*o*o*o*`。
- `step%2d`：第 $0\sim9$ 步输出 `step 0:`，第 10 步起输出 `step10:`。
- 下标从 1 开始，移动前确认两个位置都是棋子、目标两个位置都是空位。

自测

$n=4$ 应输出 5 步，最后一行是 `- - o * o * o * o *`。

例 7.7 | 题意：相邻交换最少

- 只能交换相邻两个数，把数列变成升序，求最少交换次数。
- 关键观察：一次相邻交换恰好使逆序对总数增加或减少 1。
- 因此把逆序对全部消掉，最少交换次数就等于逆序对个数。

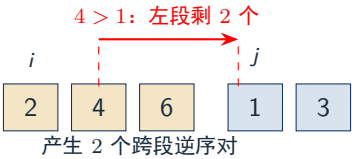
样例

$2, 1, 4, 3$ 的逆序对有 $(2, 1)$、$(4, 3)$，共 2 个，答案是 2。

- $n \leq 10000$，$O(n^2)$ 暴力数逆序对约 10^8 次，偏慢；用归并排序做到 $O(n \log n)$。
- 交换相邻元素一次，逆序对数只会改变 1，所以下界就是逆序对数，且一定能达到。

例 7.7 | 归并统计跨段逆序

- 与上一节归并求逆序对完全相同：递归左右段，合并时统计跨段。
- 合并时若 `a[i] > a[j]`，左段剩余 `mid-i+1` 个元素都与 `a[j]` 构成逆序对。
- 答案用 `long long`，最大约 $n(n-1)/2$。



与归并排序的关系

把归并排序的合并过程加上一行 `ans += mid-i+1;`，排序和计数一次完成。

例 7.7 | 代码（递归与统计）

```
#include <bits/stdc++.h>
using namespace std;
long long ans = 0;
vector<long long> a, tmp;
void msort(int l, int r) {
    if (l >= r) return;                    // 边界
    int mid = l + (r - l) / 2;
    msort(l, mid);
    msort(mid + 1, r);
    int i = l, j = mid + 1, k = l;
    while (i <= mid && j <= r) {
        if (a[i] <= a[j]) {
            tmp[k++] = a[i++];
        } else {
            ans += mid - i + 1;            // 跨段逆序对
            tmp[k++] = a[j++];
        }
    }
```

例 7.7 | 代码（收尾）

```
    while (i <= mid) {
        tmp[k++] = a[i++];
    }
    while (j <= r) {
        tmp[k++] = a[j++];
    }
    for (int t = l; t <= r; ++t) {
        a[t] = tmp[t];
    }
}
```

- 收尾阶段不再产生逆序对，因为其中一段已经为空。
- 拷贝回原数组，保证上一层拿到的是有序段。

例 7.7 | 代码（主函数）

```
int main() {
    int n;
    cin >> n;
    a.resize(n);
    tmp.resize(n);
    for (long long &x : a) cin >> x;
    msort(0, n - 1);
    cout << ans << '\n';
    return 0;
}
```

- 答案用 `long long`：最大逆序对约为 $n(n-1)/2$。
- 递归区间是闭区间 $[0, n-1]$，与 `msort` 的参数一致。

例 7.7 | 复杂度与易错点

- 时间 $O(n \log n)$，空间 $O(n)$。
- 答案最大约为 $\frac{n(n-1)}{2}$；$n = 10000$ 时约 5×10^7，虽然能放进 `int`，但用 `long long` 更稳妥。
- 相等元素不算逆序：合并时用 `a[i] <= a[j]` 先取左边。
- 统计式 `ans += mid - i + 1`：左段剩余元素个数，不要写成 `mid-i`。

检查

升序数组答案为 0，降序数组答案为 $n(n-1)/2$，所有元素相等时答案为 0。

二分答案：先写 check，再写二分

1. 答案的单调性是否成立？用一句话说明 “x 变大时，可行性如何变化”。
2. `check(x)` 是否真的只依赖 x，且能在 $O(n)$ 或 $O(n \log n)$ 内完成？
3. 二分的下界、上界是否覆盖所有可能答案？区间是闭还是开？
4. 求 “最小可行” 还是 “最大可行”？更新方向不能写反。
5. 整数二分的循环条件是 $l \le r$ 还是 $l < r$？结束后答案取哪个变量？

模板

最小可行：`if (ok(mid)) { ans=mid; r=mid-1; } else l=mid+1;`
最大可行：`if (ok(mid)) { ans=mid; l=mid+1; } else r=mid-1;`

二分查找：提交前检查清单

1. 数组是否有序？题目给的是非降序还是严格递增？
2. 区间是闭区间还是左闭右开？初始 r 写对了吗？
3. 循环条件是 $l \le r$ 还是 $l < r$？
4. `mid` 是否用 `l+(r-l)/2` 防溢出？
5. 目标不存在时返回什么？会不会越界访问？
6. 有重复元素时，要返回第一个、最后一个，还是任意一个？

必测数据

空数组、长度 1、长度 2、目标在首、目标在尾、目标不存在。

复杂度速查表

算法	时间复杂度	额外空间
二分查找	$O(\log n)$	$O(1)$
二分答案（一次 check $O(n)$）	$O(n \log V)$	$O(1)$
归并排序	$O(n \log n)$	$O(n)$
归并求逆序对	$O(n \log n)$	$O(n)$
快速幂	$O(\log p)$	$O(1)$
循环比赛日程表	$O(N^2)$	$O(N^2)$
黑白棋子移动	$O(n^2)$（输出量）	$O(n)$

- V 是答案的值域大小；二分次数约为 $\log_2 V$。
- 输出量本身很大的题，复杂度不能低于输出规模。

提交前自查

1. 样例是否逐字符一致？包括空格、空行、`step 0:` 这种格式。
2. 边界是否测过：$n = 1$、空输入、0、重复元素、最大规模。
3. 是否用了 `long long`？逆序数、区间和、快速幂的中间乘积都要检查。
4. 二分的中点是否写成 `l+(r-l)/2`？区间更新是否让范围严格缩小？
5. 浮点二分是否固定了足够多的循环次数？输出是否按题目要求保留位数？
6. 数组下标从 0 还是 1 开始？递归区间是开还是闭？全程保持一致。

练习安排：从二分到分治

- 先练二分查找与二分答案：2011、输出前 k 大的数、查找最接近的元素、和为给定数。
- 再练二分答案的判定设计：网线主管、月度开销、河中跳房子。
- 接着练实数二分：一元三次方程、二分法求零点、膨胀的木棍。
- 最后练归并与分治：区间合并、求排列的逆序数、统计数字、不重复地输出数。

做题顺序

每题先写暴力或手算小样例，再写 `check` 或分治合并，最后用边界数据自测。

本章练习题索引 | 1–7 / 14

1. P4981　2011（一本通 1234）
2. P5244 | 输出前 k 大的数（一本通 1235）
3. P5245 | 区间合并（一本通 1236）
4. P5246 | 求排列的逆序数（一本通 1237）
5. P5247 | 一元三次方程求解（一本通 1238）
6. P5248 | 统计数字（一本通 1239）
7. P5249 | 查找最接近的元素（一本通 1240）

点击蓝色题名进入学校 OJ；没有 OJ 题号的题目标为 “题包”。

本章练习题索引 | 8–14 / 14

8. P5250 | 二分法求函数的零点（一本通 1241）
9. P5251 | 网线主管（一本通 1242）
10. P5252 | 月度开销（一本通 1243）
11. P5253 | 和为给定数（一本通 1244）
12. P5254 | 不重复地输出数（一本通 1245）
13. P5255 | 膨胀的木棍（一本通 1246）
14. P5256 | 河中跳房子（一本通 1247）

本章 14 道练习全部列在此处，一条不落。