数据排序

先规定顺序，再选择方法

HGNU-ACM 实验室算法组

截图配套：2 道例题，12 道练习题





本章目录 | 基础篇

排序的目标与规则

关键字与稳定性

比较器与严格弱序

sort 与 stable_sort

冒泡、选择、插入

归并排序与逆序对

本章目录 | 例题与应用

计数排序与桶排序

多关键字排序

例 2.2：车厢重组

例 2.5：求逆序对

排序的应用与练习

什么是排序？

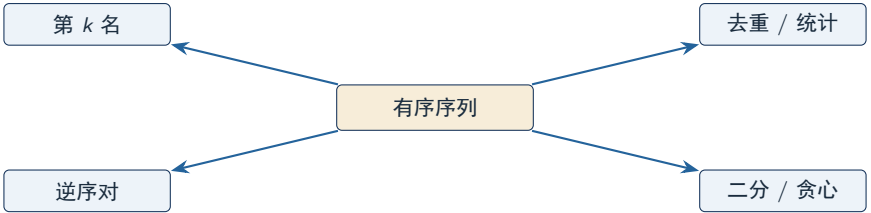
- 排序：把一组记录按某个关键字重新排列，使关键字满足指定的先后次序（升序或降序）。
- 输入是 n 条记录，输出是同一个记录集合的一个排列；元素不增不减，只改变位置。
- 相同关键字的记录谁在前、谁在后，需要由稳定性或题目规则决定。



本页要点

排序 = 重排位置 + 遵守顺序规则；先明确“按什么排、怎么排”。

排序能解决什么问题？

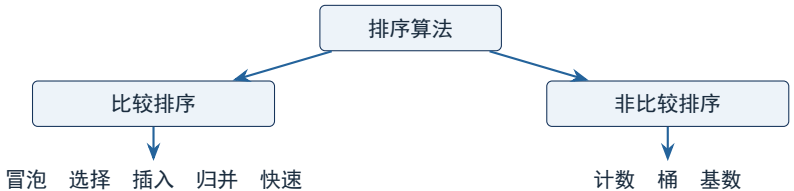


- 第 k 名：升序排序后直接取下标 $k-1$。
- 去重 / 统计：排序后相同值相邻，一次扫描即可。
- 逆序对：归并排序合并时顺带计数，不必两两枚举。
- 二分 / 贪心：很多算法的前置条件就是“数据有序”。

本页要点

排序常常不是最终答案，而是把问题变成“有序后好处理”的形态。

排序算法的两张地图



- 比较排序：只通过两两比较确定次序，最坏至少 $\Omega(n \log n)$ 次比较。
- 非比较排序：利用关键字的数值或分布直接定位，可以做到线性时间，但附加条件更严格。
- 另一条分类线是稳定 / 不稳定：相同关键字是否保持原先后。

本页要点

没有“万能最快”的排序；先看数据条件，再选算法。

评价排序算法看五个维度

算法	最好	平均	最坏	额外空间	稳定	主要特点
冒泡	$O(n)$	$O(n^2)$	$O(n^2)$	$O(1)$	是	相邻交换，可提前结束
选择	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(1)$	否	交换次数少
插入	$O(n)$	$O(n^2)$	$O(n^2)$	$O(1)$	是	近乎有序时很快
归并	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	$O(n)$	是	稳定，需辅助数组
快速	$O(n \log n)$	$O(n \log n)$	$O(n^2)$	$O(\log n)$	否	常数小，最坏退化
计数	$O(n+k)$	$O(n+k)$	$O(n+k)$	$O(n+k)$	可稳定	值域小的整数
桶	$O(n)$	$O(n)$	$O(n \log n)$	$O(n+k)$	视实现	数据需均匀分布

- 先问：n 多大？关键字是什么类型？值域多大？是否需要稳定？
- 再问：最坏情况能不能接受？额外空间够不够？

本页要点

竞赛里通常直接用 `sort`；理解这些差异，才能判断“什么时候不能只用 sort”。

先手算一遍：把 $[5,2,4,1,3]$ 排成升序

轮次	操作	序列
初始	原序列	$5,2,4,1,3$
第 1 轮	把最小值 1 换到最前	$1,2,4,5,3$
第 2 轮	最小值 2 已在原位	$1,2,4,5,3$
第 3 轮	把最小值 3 换到第 3 位	$1,2,3,5,4$
第 4 轮	把最小值 4 换到第 4 位	$1,2,3,4,5$

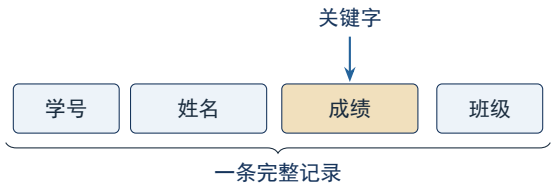
- 每一轮“确定一个位置”，已经排好的前缀不再参与比较。
- 这只是一种排法；冒泡、插入、归并的过程不同，结果相同。
- 手推时把每一步的序列写下来，是检查程序是否正确的好方法。

本页要点

排序过程可以记录成“状态序列”；会手推，才会调试。

记录、关键字与卫星数据

- 记录：要排序的一条完整信息，例如一名学生的姓名、成绩、学号。
- 关键字：排序时真正用来比较的字段，例如成绩。
- 卫星数据：跟着记录一起移动、但不参与比较的字段，例如姓名、学号。



本页要点

比较器只看关键字；交换记录时，卫星数据要跟着一起走。

关键字不一定是一个字段

- 关键字可以是一个表达式：按绝对值、按个位数字、按字符串长度。
- 也可以是组合关键字：先总分、再语文、最后学号。
- 写比较器之前，先把关键字写成一句话：“先按什么升/降序，相同再按什么”。

例子

整数按绝对值升序；若绝对值相同，按原值升序。比较器先比 $|a|$ 与 $|b|$，相等时再比 a 与 b。

本页要点

关键字是“排序依据”的抽象，不必等于结构体里的某一个成员。

稳定性：相同关键字保持原有先后

定义

若排序前记录 a 在 b 前面，且 $key(a) = key(b)$，排序后 a 仍在 b 前面，则称这个排序算法稳定。

排序前	90(A)	85(B)	90(C)	85(D)	90(E)
排序后	85(B)	85(D)	90(A)	90(C)	90(E)

相同关键字内部：B 仍在 D 前，A、C、E 次序不变

本页要点

稳定性只约束“相同关键字”的记录；不同关键字之间由大小决定。

稳定性为什么有用：多关键字排序

- 先按次关键字稳定排序，再按主关键字稳定排序。
- 第二次排序不会打乱第一次的结果：主关键字相同的记录，仍保持次关键字的顺序。
- 例：先按姓名升序，再按班级稳定排序，得到“每个班内姓名有序”。



本页要点

“从次要键到主要键”的多趟稳定排序，就是基数排序的思想。

哪些排序算法稳定？

算法	稳定性	原因
冒泡	稳定	只有严格逆序（>）才交换
插入	稳定	只有严格大于才右移
归并	稳定	合并时相等先取左半
选择	不稳定	远距离交换会跨过相等元素
快速	不稳定	分区交换会打乱相等元素
堆	不稳定	堆顶与末尾交换
计数	可稳定	从后往前放置即可稳定
桶	视实现	桶内稳定且按桶序拼接才稳定

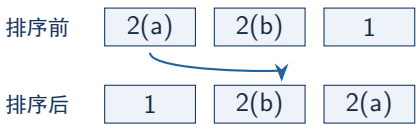
- 稳定的算法在“相同关键字”上不会制造新的先后差异。
- 不稳定的算法也可能碰巧给出稳定结果，但不能依赖。

本页要点

需要稳定时显式使用 `stable_sort`；不要假设 `sort` 稳定。

选择排序为什么不稳定？

- 反例：$[2_a, 2_b, 1]$。第一轮把 1 与第一个 2 交换，得到 $[1, 2_b, 2_a]$，两个 2 的相对次序反了。
- 原因是选择排序做远距离交换，相等元素可能被跨过。
- 冒泡、插入只做相邻移动，所以能保持稳定。



本页要点

选择排序用“远距离交换”换来了“交换次数少”，代价是失去稳定性。

比较器：只回答“谁在前”

- 比较器 `comp(a,b)` 返回 `true`，表示 a 必须排在 b 前面。
- 升序：`return a < b;`；降序：`return a > b;`。
- 关键字相等时必须返回 `false`：既不认为 a 在 b 前，也不认为 b 在 a 前。
- 比较器不是“返回差值”，而是一个布尔判断。

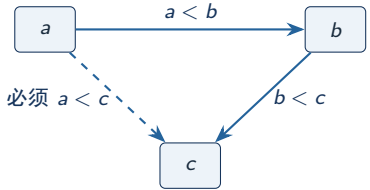
一句话检查

把 `a` 和 `b` 互换，两个方向不能同时返回 `true`。

本页要点

比较器描述的是“严格先后关系”，不是“大小减法”。

严格弱序：sort 对比较器的三条要求



- 非自反：comp(x,x) 必须为 false。
- 非对称：comp(a,b) 为真，则 comp(b,a) 必须为假。
- 传递：comp(a,b) 且 comp(b,c)，则 comp(a,c)。
- 等价关系也要传递：a 与 b 等价、b 与 c 等价，则 a 与 c 等价。

本页要点

违反任何一条，sort 可能越界、死循环或给出错误顺序。

最常见的错误：把 < 写成 <=

```
bool cmp_bad(int a, int b) {
    return a <= b; // 错误：cmp(x, x) 为 true
}

bool cmp_good(int a, int b) {
    return a < b; // 正确：严格小于
}
```

- <= 让元素“小于自己”，破坏非自反性。
- 多关键字比较器，最后一层也必须是严格比较。
- 浮点数不要用 == 判等；按题意转成整数（如乘 100 取整），或留出误差。

本页要点

比较器里只应出现严格不等式，除非你在比较“组合键”的某一段。

lambda 语法：把函数写在表达式里



- `[]` 不捕获外部变量；`[&]` 引用捕获；`[=]` 值捕获。
- 返回类型通常省略；只有一条 `return` 时自动推导。
- 例：`auto add = [](int a, int b) { return a + b; };`

本页要点

lambda 是匿名函数对象，可以像变量一样传给 `sort`。

lambda 写比较器：先看懂再使用

```cpp
sort(a.begin(), a.end(), [](int x, int y) {
    return x > y; // 降序
});

sort(v.begin(), v.end(), [](const Student &x, const Student &y) {
    return x.score > y.score; // 按分数降序
});
```

- `[]` 表示不捕获外部变量；比较器通常不需要捕获。
- 参数用 `const T &`：避免拷贝，也能接收临时对象。
- 比较器会被调用很多次，内部不要修改元素，也不要输出调试信息。

本页要点

`sort` 把 lambda 当成一个“可调用的比较规则”，而不是普通函数名。

比较器的三种写法

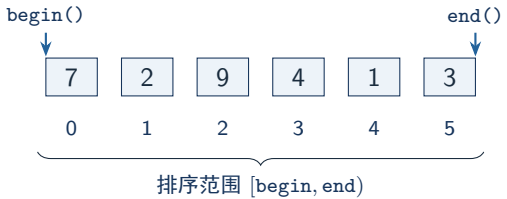
写法	形式	适用
普通函数	`bool cmp(const T&, const T&)`	规则简单、多处复用
函数对象	`struct Cmp { bool operator()(...) };`	需要携带状态
lambda	`[](const T &a, const T &b) { ... }`	就地书写、最短

```
struct Cmp {
    bool operator()(int a, int b) const {
        return a > b; // 降序
    }
};
sort(a.begin(), a.end(), Cmp());
```

本页要点

三种写法本质相同：给 sort 一个“能回答谁在前”的可调用对象。

sort 的区间是左闭右开



- sort(first, last) 排序 [first, last)，不包含 last 指向的元素。
- v.begin() 指向下标 0；v.end() 指向最后一个元素之后。
- 子数组：sort(a + l, a + r) 排序下标 $l, l+1, \ldots, r-1$。

本页要点

最常见的漏排，就是把 end 写成了最后一个元素而不是“末尾之后”。

sort 的三种常见调用

```cpp
#include <algorithm>
#include <functional>
#include <vector>
using namespace std;

int main() {
    vector<int> a = {4, 1, 3, 2};
    sort(a.begin(), a.end()); // 升序：1 2 3 4
    sort(a.begin(), a.end(), greater<int>()); // 降序：4 3 2 1
    sort(a.begin(), a.end(), [](int x, int y) {
        return x < y; // 自定义规则
    });
    return 0;
}
```

sort 三种调用的区别

写法	顺序	需要的头文件
`sort(begin, end)`	升序	`<algorithm>`
`sort(begin, end, greater<int>())`	降序	`<functional>`
`sort(begin, end, lambda)`	自定义	`<algorithm>`

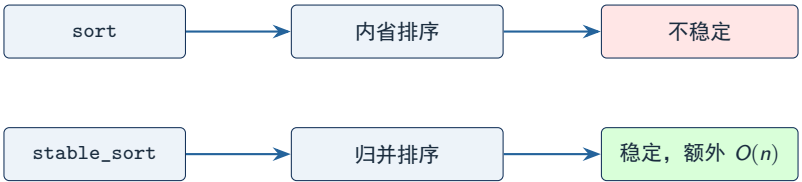
- 默认比较器是 `less<int>()`，即升序。
- `greater<int>()` 是标准库提供的“大于”比较器。
- lambda 最灵活，适合结构体和多关键字；写法见后面的比较器一页。

本页要点

先确定顺序，再选择“默认 / `greater` / lambda”三种写法之一。

sort 与 stable_sort 的复杂度

函数	比较次数	额外空间	稳定性
sort	平均/最坏 $O(n \log n)$	$O(\log n)$	不稳定
stable_sort	有足够内存 $O(n \log n)$	$O(n)$	稳定
	内存不足 $O(n \log^2 n)$	较少	稳定



本页要点

sort 用内省排序（快排 + 堆排 + 插入），stable_sort 通常用归并。

sort 的四个常见坑

```
sort(a.begin(), a.end() - 1); // 错误：最后一个元素没排
sort(a.begin(), a.end()); // 正确

sort(a.begin(), a.end(), [](int x, int y) {
    return x <= y; // 错误：不是严格弱序
});
```

- 区间少一个元素：检查 end 是否指向“末尾之后”。
- 比较器用了 <=：改成严格比较。
- 排序后原下标丢失：把关键字和原下标一起存进 pair 或结构体。
- 需要稳定却用了 sort：改用 stable_sort 或多关键字比较器。

本页要点

排序出错时，先检查区间，再检查比较器。

完整例子：学生成绩排序（上）

```cpp
#include <algorithm>
#include <iostream>
#include <string>
#include <vector>
using namespace std;

struct Student {
    string name;
    int total; // 总分：主关键字
    int chinese; // 语文：次关键字
    int id; // 学号：最后的区分键
};
```

- 结构体把一条记录的所有字段放在一起。
- 排序时关键字是 `total`、`chinese`、`id` 的组合。

完整例子：学生成绩排序（中）

```
int main() {
    int n;
    cin >> n;
    vector<Student> a(n);
    for (int i = 0; i < n; ++i) {
        cin >> a[i].name >> a[i].total >> a[i].chinese;
        a[i].id = i + 1; // 保留输入顺序
    }
```

- 读入 n 条记录，并给每条记录补上原始下标。
- 原始下标是最后的“平局键”，保证结果可复现。

完整例子：学生成绩排序（下）

```
    sort(a.begin(), a.end(), [](const Student &x, const Student &y) {
        if (x.total != y.total) {
            return x.total > y.total; // 总分高的在前
        }
        if (x.chinese != y.chinese) {
            return x.chinese > y.chinese; // 语文高的在前
        }
        return x.id < y.id; // 学号小的在前
    });
    for (const Student &s : a) {
        cout << s.name << ' ' << s.total << '\n';
    }
    return 0;
}
```

冒泡排序：相邻比较，大的往后走

- 每一趟从左到右比较相邻两个元素；若左边大于右边，就交换。
- 一趟结束后，未排序部分的最大值一定到达最右边。
- 最多做 $n-1$ 趟；某一趟没有发生交换，说明已经有序，可以提前结束。

初始	5	1	4	2	8
$5>1$ 交换	1	5	4	2	8
$5>4$ 交换	1	4	5	2	8
$5>2$ 交换	1	4	2	5	8
$5<8$ 不换	1	4	2	5	8

本页要点

第 1 趟结束，最大值 8 已经“冒”到末尾。

冒泡排序全过程与性质

趟数	本趟结果	比较次数	交换次数
初始	$5,1,4,2,8$	–	–
第 1 趟	$1,4,2,5,8$	4	3
第 2 趟	$1,2,4,5,8$	3	1
第 3 趟	$1,2,4,5,8$	2	0（提前结束）

- 最坏比较次数 $n(n-1)/2$，时间 $O(n^2)$；额外空间 $O(1)$。
- 交换次数恰好等于初始逆序对数：每次交换只消去一对逆序。
- 只有 $a[i] > a[i+1]$ 才交换，相等不换，所以稳定。
- 已经有序时只扫一趟，最好 $O(n)$。

本页要点

冒泡排序是理解“相邻交换与逆序对”的最好模型。

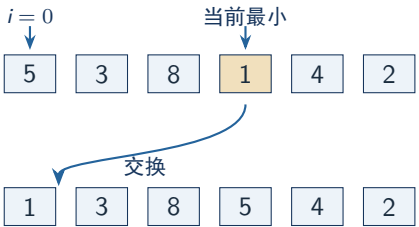
冒泡排序代码（理解用）

```cpp
void bubble_sort(vector<int> &a) {
    int n = a.size();
    for (int pass = 0; pass < n - 1; ++pass) {
        bool changed = false; // 本趟是否发生交换
        for (int i = 0; i < n - 1 - pass; ++i) {
            if (a[i] > a[i + 1]) { // 严格大于才交换
                swap(a[i], a[i + 1]);
                changed = true;
            }
        }
        if (!changed) { // 已经有序，提前结束
            break;
        }
    }
}
```

时间 $O(n^2)$，空间 $O(1)$，稳定；末尾 pass 个元素已经到位。

选择排序：每轮把最小值换到前面

- 已排序区是 $a[0..i-1]$，未排序区是 $a[i..n-1]$。
- 在未排序区找到最小值的下标 `min_pos`。
- 把 `a[min_pos]` 与 `a[i]` 交换，已排序区扩大一格。
- 做 $n-1$ 轮后全部有序。



本页要点

选择排序的“找最小”是线性扫描，所以总比较次数固定为 $n(n-1)/2$。

选择排序全过程

轮次	操作	序列
初始	原序列	$5,3,8,1,4,2$
第 1 轮	最小值 1 与 5 交换	$1,3,8,5,4,2$
第 2 轮	最小值 2 与 3 交换	$1,2,8,5,4,3$
第 3 轮	最小值 3 与 8 交换	$1,2,3,5,4,8$
第 4 轮	最小值 4 与 5 交换	$1,2,3,4,5,8$
第 5 轮	只剩一个元素，结束	$1,2,3,4,5,8$

- 第 i 轮比较 $n-i-1$ 次，总比较次数 $n(n-1)/2$。
- 交换最多 $n-1$ 次，适合“交换代价大、比较便宜”的场景。
- 时间 $O(n^2)$，空间 $O(1)$。

本页要点

选择排序的交换次数少，但比较次数无法提前减少。

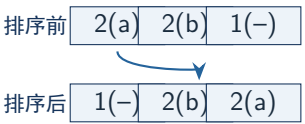
选择排序代码（理解用）

```
void selection_sort(vector<int> &a) {
    int n = a.size();
    for (int i = 0; i < n - 1; ++i) {
        int min_pos = i; // 先假设 i 是最小值
        for (int j = i + 1; j < n; ++j) {
            if (a[j] < a[min_pos]) {
                min_pos = j; // 记录更小的位置
            }
        }
        if (min_pos != i) {
            swap(a[i], a[min_pos]); // 换到已排序区末尾
        }
    }
}
```

内层只负责“找最小”，每轮最多交换一次；时间 $O(n^2)$，空间 $O(1)$。

选择排序的性质：交换少，但不稳定

- 不稳定反例：$[2_a, 2_b, 1]$。第一轮把 1 与第一个 2 交换，得到 $[1, 2_b, 2_a]$，两个 2 的相对次序反了。
- 时间复杂度最好、平均、最坏都是 $O(n^2)$，与初始顺序无关。
- 空间 $O(1)$；交换次数最多 $n-1$，这是它相对冒泡的优点。

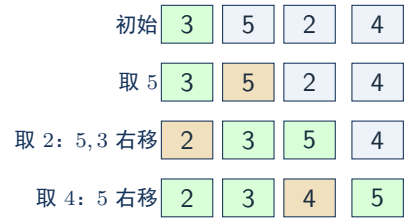


本页要点

选择排序用“远距离交换”换来了“交换次数少”，代价是失去稳定性。

插入排序：像整理扑克牌

- 已排序前缀是 $a[0..i-1]$；把 $a[i]$ 取出来作为 key。
- 从右往左找位置：比 key 大的元素依次右移一格。
- 找到空位后，把 key 放进去。
- 对 $i = 1, 2, \ldots, n-1$ 重复，前缀不断扩大。



本页要点

绿色是已排序前缀，金色是本轮取出的 key。

插入排序全过程与性质

轮次	操作	序列
初始	前缀只有 3	$3 \mid 5,2,4$
$i=1$	key$=5$，不动	$3,5 \mid 2,4$
$i=2$	key$=2$，$5,3$ 右移	$2,3,5 \mid 4$
$i=3$	key$=4$，5 右移	$2,3,4,5$

- 最好情况（已经升序）：每轮只比较一次，时间 $O(n)$。
- 最坏情况（完全降序）：比较与移动都是 $O(n^2)$。
- 平均 $O(n^2)$，额外空间 $O(1)$。
- 只有严格大于才右移，相等元素不跨过，所以稳定。

本页要点

插入排序对“近乎有序”的数据非常快，常被用作小数组的收尾排序。

插入排序代码（理解用）

```
void insertion_sort(vector<int> &a) {
    int n = a.size();
    for (int i = 1; i < n; ++i) {
        int key = a[i]; // 待插入的牌
        int j = i - 1;
        while (j >= 0 && a[j] > key) { // 严格大于才右移
            a[j + 1] = a[j];
            --j;
        }
        a[j + 1] = key; // 放进空位
    }
}
```

- 内层 `while` 同时完成“找位置”和“右移”。
- 条件 `j >= 0` 必须写在前面，避免越界。
- 时间 $O(n^2)$，空间 $O(1)$，稳定。

三种基础排序对比

算法	最好	平均	最坏	额外空间	稳定
冒泡	$O(n)$	$O(n^2)$	$O(n^2)$	$O(1)$	是
选择	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(1)$	否
插入	$O(n)$	$O(n^2)$	$O(n^2)$	$O(1)$	是

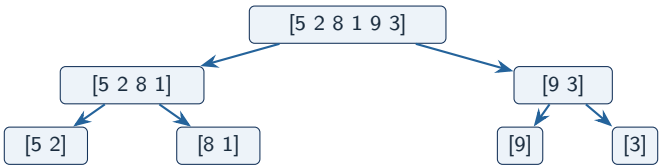
- 冒泡：相邻交换，能顺带理解逆序对；有提前结束优化。
- 选择：交换次数最少，但比较次数固定，且不稳定。
- 插入：近乎有序时接近 $O(n)$，稳定，适合小数组。
- 竞赛中通常直接 `sort`；这三种算法主要用于理解过程和性质。

本页要点

“比较—交换—插入”是排序的三种基本动作，后面的高级算法都由此组合而来。

归并排序：分治三步

- 分：把区间 $[l, r)$ 从 `mid` 切成左右两半。
- 治：递归排序左半 $[l, \texttt{mid})$ 与右半 $[\texttt{mid}, r)$。
- 合：把两个已经有序的半区合并成一个有序区间。
- 递归基：区间长度 0 或 1 时天然有序，直接返回。



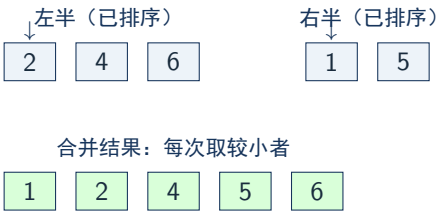
继续分到长度 1，再两两合并回去

本页要点

归并排序的核心在“合”：两个有序数组能在 $O(n)$ 时间内合并。

合并两个有序数组：双指针

- i 指向左半当前元素，j 指向右半当前元素。
- 比较 $a[i]$ 与 $a[j]$，把较小者放进临时数组 `tmp`。
- 相等时先取左边，这样相同关键字保持原顺序，归并才稳定。
- 一侧取完后，把另一侧剩余元素全部复制过去。



本页要点

合并一次只扫描左右两半各一遍，时间 $O(r-l)$。

归并排序代码（上）：递归与合并

```
void merge_sort(vector<int> &a, vector<int> &tmp, int l, int r) {
    if (r - l <= 1) {
        return; // 0 或 1 个元素
    }
    int mid = l + (r - l) / 2; // 防止 l + r 溢出
    merge_sort(a, tmp, l, mid);
    merge_sort(a, tmp, mid, r);
    int i = l, j = mid, k = l;
    while (i < mid && j < r) {
        if (a[i] <= a[j]) { // 相等取左边，保持稳定
            tmp[k++] = a[i++];
        } else {
            tmp[k++] = a[j++];
        }
    }
```

归并排序代码（下）：收尾与拷回

```
    while (i < mid) {
        tmp[k++] = a[i++]; // 左半剩余
    }
    while (j < r) {
        tmp[k++] = a[j++]; // 右半剩余
    }
    for (int p = l; p < r; ++p) {
        a[p] = tmp[p]; // 拷回原数组
    }
}
```

- 函数排序的是左闭右开区间 $[l, r)$，调用时写 `merge_sort(a, tmp, 0, n)`。
- 两个 `while` 最多执行一个；它们把剩余元素直接搬过去。

归并排序的主程序（上）

```
#include <iostream>
#include <vector>
using namespace std;

int main() {
    int n;
    cin >> n;
    vector<int> a(n), tmp(n);
    for (int &x : a) {
        cin >> x;
    }
```

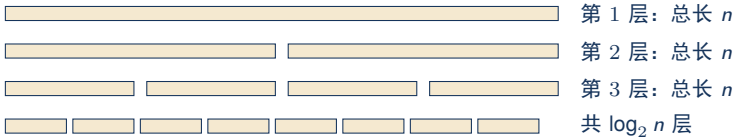
读入 n 个元素，并准备同大小的辅助数组 `tmp`。

归并排序的主程序（下）

```
    merge_sort(a, tmp, 0, n); // 区间 [0, n)
    for (int x : a) {
        cout << x << ' ';
    }
    cout << '\n';
    return 0;
}
```

调用时区间是 $[0, n)$；cin 会自动跳过空白。

归并排序的复杂度与稳定性



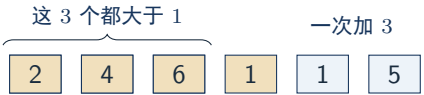
- 每一层合并扫描的总长度都是 n，共有 $\lceil \log_2 n \rceil$ 层，所以时间 $O(n \log n)$。
- 辅助数组 `tmp` 需要 $O(n)$ 额外空间。
- 合并时相等取左，保证稳定。
- 归并排序的最好、平均、最坏都是 $O(n \log n)$，没有退化情况。

本页要点

归并排序用 $O(n)$ 的空间换来了稳定与最坏 $O(n \log n)$。

归并排序求逆序对：跨区间计数

- 左半、右半内部的逆序，已经在递归中分别统计完毕。
- 合并时只需统计“左半取一个、右半取一个”的跨区间逆序。
- 若 $a[i] > a[j]$，那么左半从 i 到 $\texttt{mid}-1$ 的 所有元素都大于 $a[j]$，一次加上 `mid - i`。
- 若 $a[i] \le a[j]$，先取左边，不产生逆序；相等不算逆序。



取 1 时：`answer += mid - i`

本页要点

每个跨区间逆序恰好被统计一次：左半元素出队时，右半已经出队的都比它小。

计数排序：关键字是小数范围内的整数

适用条件

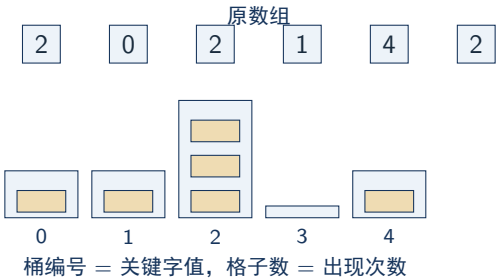
关键字必须是整数（或能一一映射到整数），且值域 $[0, k]$ 不太大；k 与 n 同阶或内存可以接受。

- 不比较元素大小，直接统计每个值出现了多少次。
- 用前缀和确定每个值在结果中的结束位置。
- 从后往前扫描原数组并放置，可以保证稳定。
- 时间 $O(n + k)$，空间 $O(n + k)$。

本页要点

计数排序用“值域换时间”；$k = 10^9$ 时不能直接开数组。

计数排序的桶与计数



本页要点

值 2 出现 3 次，所以编号 2 的桶最高。

计数排序的三步

1. 计数：cnt[x] 统计值 x 出现的次数。
2. 前缀和：cnt[v] += cnt[v-1] 后，cnt[v] 表示“值 $\leq v$ 的元素个数”，也就是值 v 的结束位置。
3. 从后往前放：把元素放到 cnt[a[i]]-1，再让 cnt[a[i]] 减 1，相同值的相对次序不变。

为什么必须从后往前？

从前往后放会把相同值按相反次序塞进结果，破坏稳定性；从后往前放，最后一个相同值占据最后一个位置，前面的依次往前排。

本页要点

先计数，再前缀和，最后从后往前放；三步都不能少。

计数排序代码

```
void counting_sort(vector<int> &a, int max_value) {
    vector<int> cnt(max_value + 1, 0);
    for (int x : a) {
        ++cnt[x]; // 统计每个值出现次数
    }
    for (int v = 1; v <= max_value; ++v) {
        cnt[v] += cnt[v - 1]; // 前缀和：结束位置
    }
    vector<int> out(a.size());
    for (int i = (int)a.size() - 1; i >= 0; --i) {
        out[cnt[a[i]] - 1] = a[i]; // 从后往前放，保证稳定
        --cnt[a[i]];
    }
    a = out;
}
```

`max_value` 是最大值；负数先加偏移；只能排整数。

计数排序的边界与易错点

- 值域过大：$k = 10^9$ 时数组开不下；改用 map、离散化或比较排序。
- 有负数：先求最小值 min，把 x 映射到 $x - \min$，输出时再还原。
- 值从 1 开始：数组要开到 $k + 1$，或者统一减 1。
- 稳定性：必须从后往前放置；从前往后会让相同值逆序。
- 输出位置：前缀和是“结束位置”，所以下标要减 1。

检查清单

值域多大？有没有负数？需不需要稳定？这三个问题决定计数排序能不能用、怎么写。

本页要点

计数排序的常数很小，但条件最苛刻：整数、值域小、内存够。

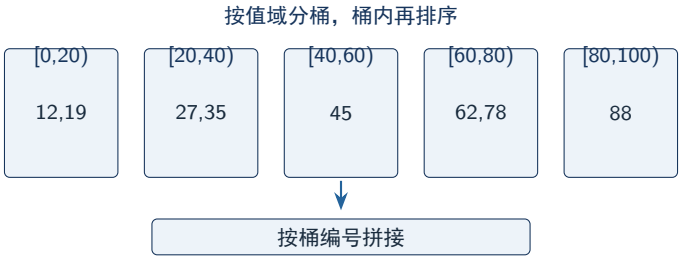
桶排序：适用条件与步骤

- 适用条件：数据均匀分布在一个区间内，例如 $[0,1)$ 上的浮点数。
- 把值域分成 m 个桶，元素按值落进对应桶。
- 每个桶内部再排序（常用插入排序或 `sort`）。
- 按桶编号从小到大拼接，整体就有序了。
- 平均时间 $O(n+m+n^2/m)$；取 $m \approx n$ 且分布均匀时接近 $O(n)$。

本页要点

桶排序快的前提是“分布均匀”；若桶内用插入排序，所有元素挤进一个桶会退化成 $O(n^2)$。

桶排序示意



- 编号小的桶里的值整体更小，所以桶间不需要再比较。
- 图中桶内已排好；实现时要对每个桶单独排序。

本页要点

先分桶，再桶内排序，最后按桶编号拼接。

桶排序代码与条件

```cpp
void bucket_sort(vector<double> &a) {
    int n = a.size();
    vector<vector<double>> bucket(n); // n 个桶
    for (double x : a) {
        int id = min(n - 1, (int)(x * n)); // 假设 0 <= x < 1
        bucket[id].push_back(x);
    }
    int pos = 0;
    for (int i = 0; i < n; ++i) {
        sort(bucket[i].begin(), bucket[i].end());
        for (double x : bucket[i]) {
            a[pos++] = x; // 按桶编号拼回
        }
    }
}
```

一般数据先归一化到 $[0,1)$；桶内排序不保证稳定。

计数、桶、比较排序怎么选？

数据特征	推荐算法	原因
n 较小，或近乎有序	插入排序	常数小，近乎 $O(n)$
一般整数、结构体、多关键字	`sort`	通用、常数小
必须保持相同关键字原序	`stable_sort`	稳定
整数且值域 k 很小	计数排序	$O(n+k)$
浮点数且分布均匀	桶排序	平均 $O(n)$
n 很大、值域也很大	归并 / 快速	$O(n \log n)$

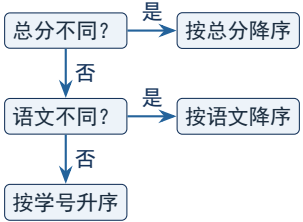
- 先看数据范围：$n \le 10^5$ 通常 $O(n \log n)$；$n \le 10^3$ 可以 $O(n^2)$。
- 再看关键字：整数小值域可以计数；浮点均匀可以桶排。
- 最后看稳定性与内存：需要稳定选归并 / `stable_sort`。

本页要点

算法选择不是背结论，而是按“数据条件”逐条排除。

多关键字排序：字典序规则

- 先比较主关键字；主关键字不同，立即决定先后。
- 主关键字相同，才比较次关键字；仍相同再比较第三关键字。
- 所有平局规则必须写全；否则相同键的记录顺序由实现决定，可能不稳定。
- 例：总分降序 → 语文降序 → 学号升序。



本页要点

多关键字比较就是“字典序”：前面的键相同，才轮到后面的键。

多关键字比较器的写法

```cpp
struct Student {
    int id;
    int total;
    int chinese;
};

sort(a.begin(), a.end(), [](const Student &x, const Student &y) {
    if (x.total != y.total) {
        return x.total > y.total; // 主键：总分降序
    }
    if (x.chinese != y.chinese) {
        return x.chinese > y.chinese; // 次键：语文降序
    }
    return x.id < y.id; // 第三键：学号升序
});
```

每一层都必须严格比较；需要保留原序时，把原下标作为最后一个键。

用稳定排序实现多关键字

- 先按最次要的键稳定排序，再按更高优先级的键稳定排序，依次进行。
- 每一步稳定排序都不会打乱上一步的结果，最终等价于按完整字典序排序。
- 例：先按姓名，再按班级，再按年级；最后每个年级内班级有序、班内姓名有序。
- 这就是基数排序（LSD）的思想：从低位到高位，多趟稳定排序。



每一步都必须稳定，否则前一步的结果会被打乱

本页要点

多趟稳定排序把“复杂比较器”拆成“多个简单比较器”。

病人排队：老年人优先，同龄按年龄降序

```cpp
struct Patient {
    int age;
    int order; // 登记顺序
};

stable_sort(a.begin(), a.end(), [](const Patient &x, const Patient &y) {
    bool old_x = x.age >= 60;
    bool old_y = y.age >= 60;
    if (old_x != old_y) {
        return old_x; // 老年人整体在前
    }
    if (old_x && x.age != y.age) {
        return x.age > y.age; // 老年组按年龄降序
    }
    return x.order < y.order; // 其余保持登记顺序
});
```

先分组，再在组内按规则比较；最后用登记顺序兜底。

多关键字排序的两种实现

方法	优点	注意
一个比较器一次排序	只需一趟 $O(n \log n)$，代码集中	规则多时容易漏层、写错严格性
多趟稳定排序	每趟规则简单，易验证	需要稳定排序；k 个键要排 k 趟，$O(kn \log n)$

- 键的优先级固定、规则不多：直接写一个多关键字比较器。
- 键很多（如整数按每一位）：多趟稳定排序更自然。
- 两种方法都必须保证“相同完整关键字”的记录顺序符合题意。

本页要点

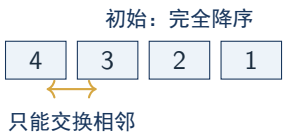
先写清楚优先级表，再决定用哪种实现。

例 2.2 | 车厢重组：题目与建模

题目

$N \le 10000$，N 个互不相同的数表示车厢顺序。桥一次只能交换 相邻两节车厢，求排成升序的最少旋转次数。

- 一次旋转 $=$ 一次相邻交换；目标是把序列变成升序。
- 样例：$4,3,2,1$ 的每一对位置都逆序，共 $\binom{4}{2} = 6$ 对，输出 6。
- 建模：最少相邻交换次数 $=$ 初始序列的逆序对数。



本页要点

把“桥的旋转”抽象成“相邻交换”，问题就变成了逆序对计数。

例 2.2 | 为什么答案正好是逆序对数？

- 下界：每一对逆序车厢最终都必须交换相对次序；一次相邻交换只改变一对相邻元素的相对次序，因此至少需要“逆序对数”次。
- 上界：只要序列还没排好，就存在相邻的逆序对；交换它，逆序对数恰好减 1。反复操作，逆序对数降到 0 时序列有序。
- 两边合起来：最少次数 $=$ 初始逆序对数。

注意

这里说的是相邻交换。如果允许任意两个位置互换，最少次数会不同。

本页要点

下界与上界相等，是“最少操作”类问题最常用的证明方式。

例 2.2 | $O(n^2)$ 统计程序（上）

```
#include <iostream>
#include <vector>
using namespace std;

int main() {
    int n;
    cin >> n;
    vector<int> a(n);
    for (int &x : a) {
        cin >> x;
    }
```

- 读入 n 与 n 个车厢编号。
- 下一步用两层循环枚举所有位置对。

例 2.2 | $O(n^2)$ 统计程序（下）

```
    long long answer = 0;
    for (int i = 0; i < n; ++i) {
        for (int j = i + 1; j < n; ++j) {
            if (a[i] > a[j]) {
                ++answer; // 每一对逆序计一次
            }
        }
    }
    cout << answer << '\n';
    return 0;
}
```

- 外层固定左端 i，内层枚举右端 $j > i$。
- $N = 10000$ 时约有 5×10^7 对，$O(n^2)$ 可以过。
- 答案用 `long long`；本题 N 较小，`int` 也够，但统一更稳。

例 2.2 | 手算与边界

- $[4,3,2,1]$：4 右边有 3 个更小数，3 右边有 2 个，2 右边有 1 个，总数 $3+2+1=6$。
- 已经升序：没有逆序对，答案 0。
- 完全降序：答案 $n(n-1)/2$；$n=10000$ 时是 $49\,995\,000$。
- 题目保证车厢编号互不相同；若允许相等，条件要写成严格 $>$，相等不算逆序。

进阶

N 增大到 10^5 时，$O(n^2)$ 太慢；下一节用归并排序把计数降到 $O(n \log n)$。

本页要点

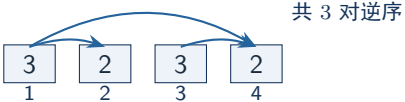
先写对 $O(n^2)$，再考虑优化；小数据对拍是验证优化程序的好方法。

例 2.5 | 题目与样例

题目

给定 $a_1, \ldots, a_n$，统计满足 $i < j$ 且 $a_i > a_j$ 的位置对。$n \le 10^5$，$a_i \le 10^5$。

- 样例 $[3, 2, 3, 2]$：逆序对是 $(1, 2)$、$(1, 4)$、$(3, 4)$，输出 3。
- 注意相等不算逆序：条件是严格 $>$，不是 $\ge$。
- $O(n^2)$ 要检查约 5×10^9 对位置，必须改用 $O(n \log n)$。

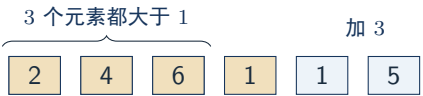


本页要点

逆序对只看“位置在前、值反而更大”的两个元素。

例 2.5 | 归并计数：合并时一次加 mid - i

- 左右半区已经各自升序；内部逆序已在递归中统计。
- 合并时若 $a[i] \le a[j]$，先取左，不产生新的跨区间逆序。
- 若 $a[i] > a[j]$，则左半 $i, i+1, \ldots, \texttt{mid}-1$ 共 $\texttt{mid}-i$ 个元素都大于 $a[j]$，一次全部计入。
- 这个“一次加一段”的动作，是把 $O(n^2)$ 降到 $O(n \log n)$ 的关键。



本页要点

统计跨区间逆序时，右半元素出队一次，就结算它与左半剩余元素的全部逆序。

例 2.5 | merge_count（上）

```
long long merge_count(vector<int> &a, vector<int> &tmp, int l, int r) {
    if (r - l <= 1) {
        return 0; // 0 或 1 个元素
    }
    int mid = l + (r - l) / 2;
    long long left_answer = merge_count(a, tmp, l, mid);
    long long right_answer = merge_count(a, tmp, mid, r);
    long long answer = left_answer + right_answer;
    int i = l, j = mid, k = l;
    while (i < mid && j < r) {
        if (a[i] <= a[j]) {
            tmp[k++] = a[i++]; // 相等不算逆序
        } else {
```

例 2.5 | merge_count（下）

```
            answer += mid - i; // 左半剩余全部大于 a[j]
            tmp[k++] = a[j++];
        }
    }
    while (i < mid) {
        tmp[k++] = a[i++];
    }
    while (j < r) {
        tmp[k++] = a[j++];
    }
    for (int p = l; p < r; ++p) {
        a[p] = tmp[p];
    }
    return answer;
}
```

返回类型和答案变量都必须是 `long long`。

例 2.5 | 主程序

```
#include <iostream>
#include <vector>
using namespace std;

int main() {
    int n;
    cin >> n;
    vector<int> a(n), tmp(n);
    for (int &x : a) {
        cin >> x;
    }
    cout << merge_count(a, tmp, 0, n) << '\n';
    return 0;
}
```

区间是 $[0, n)$；cin 会自动跳过空白。

例 2.5 | 样例手推与边界

- 样例 $[3,2,3,2]$：$[3]$ 与 $[2]$ 合并得 1 对；$[3]$ 与 $[2]$ 合并得 1 对；最后合并 $[2,3]$ 与 $[2,3]$ 时，$3>2$ 再得 1 对，共 3 对。
- 全相等 $[2,2,2]$：过程中一直取左边，答案 0。
- 已升序：答案 0；完全降序：答案 $n(n-1)/2$。
- 单个元素：递归立即返回 0。
- $n=10^5$ 完全降序时答案约为 $4\,999\,950\,000$，超过 `int` 上限。

本页要点

合并时“相等取左边”既保证稳定，也保证相等元素不被误算成逆序。

排序后去重：先排再比较相邻项

- 排序后相同的值一定相邻；扫描时只保留“与前一个保留值不同”的元素。
- C++ 也可用 sort 后接 unique，但它只整理区间，不会自动缩短容器，必须再 erase。

```
sort(values.begin(), values.end());
values.erase(unique(values.begin(), values.end()), values.end()); // 去重
for (int x : values) {
    cout << x << ' ';
}
cout << '\n';
```

$[20, 40, 20, 15] \to [15, 20, 20, 40] \to [15, 20, 40]$。

本页要点

unique 返回新的逻辑末尾；不 erase 的话，容器长度不变。

从有序结果中取答案

- 第 k 名：升序排序后取下标 $k-1$；注意下标从 0 开始。
- 分数线：先确定要保留的人数 m，排序后第 m 名的分数就是线；与线同分的人全部入选，所以实际人数可能超过 m。
- 出现次数超过一半：排序后若存在这样的数，它一定占据中间位置；仍要再扫描一次核对出现次数。
- 离散化：排序 $+$ 去重后，用 `lower_bound` 把大值映射成小下标。

先检查题意

升序还是降序？并列怎么处理？输出数量、分数，还是完整记录？

本页要点

排序后的下标、边界、并列规则，是取答案时最容易错的三处。

离散化：把大值域映射成小下标

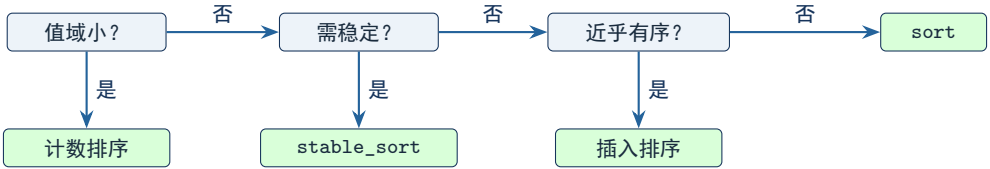
```
vector<int> v = a; // 复制一份用于离散化
sort(v.begin(), v.end());
v.erase(unique(v.begin(), v.end()), v.end());
for (int &x : a) {
    // 在有序去重数组中找到 x 的排名，从 1 开始
    x = lower_bound(v.begin(), v.end(), x) - v.begin() + 1;
}
```

- 排序去重后，v 中的每个值对应一个排名。
- lower_bound 返回迭代器，减去 begin() 得到下标；加 1 让排名从 1 开始。
- 离散化后值域变成 $O(n)$，可以开数组做计数、树状数组等。

本页要点

离散化 = 排序 + 去重 + 二分查找，是“值域太大”时的常用技巧。

选择排序算法的决策顺序



- 先排除“值域小”的计数排序；再检查稳定性；最后才是一般排序。
- 数据规模、内存限制、最坏情况也要一起考虑。

本页要点

先问条件，再选算法；不要一上来就写 `sort`。

练习建议：从单键到多键

1. 先做奇数序列、整数奇偶排序，熟悉筛选与升降序。
2. 再做成绩排序、奖学金、病人排队，练多关键字比较器与稳定性。
3. 最后做去重、分数线与统计，练排序后的扫描。

每题都要问

排序前后的哪些信息还需要保留？是否要保留原下标？相同元素怎么处理？

本页要点

练习顺序：单键 $\rightarrow$ 多键 $\rightarrow$ 稳定 $\rightarrow$ 扫描取答案。

本章练习题索引 | 1–6 / 12

1. P5170 | 谁考了第 k 名（一本通 1176）
2. P5171 | 奇数单增序列（一本通 1177）
3. P5172 | 成绩排序（一本通 1178）
4. P5173 | 奖学金（一本通 1179）
5. P5174 | 分数线划定（一本通 1180）
6. P5175 | 整数奇偶排序（一本通 1181）

点击蓝色题名进入学校 OJ；本章截图中的练习全部列在索引中。

本章练习题索引 | 7–12 / 12

7. P5176 | 合影效果（一本通 1182）
8. P5177 | 病人排队（一本通 1183）
9. P5178 | 明明的随机数（一本通 1184）
10. P5179 | 单词排序（一本通 1185）
11. P5180 | 出现次数超过一半的数（一本通 1186）
12. P5181 | 统计字符数（一本通 1187）

点击蓝色题名进入学校 OJ；本章截图中的练习全部列在索引中。