第 3 章 数组与基础排序

> 本文是《XCPCer 从零开始的 C/C++ 入门指南（算法竞赛向）》系列第 3 章，系列训练大纲见本文。本章是重点章，配套第三次训练的题单。

3.0 导语：从“一个变量”到“一批数据”

前两章里，我们处理的数据大多是 `a`、`b`、`n` 这样的单个变量。但算法竞赛很少只给一个数：题目可能给出 100 个成绩、一个棋盘，或者要求把一批数从小到大排列。

如果为每个数据单独声明一个变量，写 100 个 `cin` 显然不现实。**数组**把一批同类型的数据放在一起，用一个名字加一个下标访问其中的每个元素。

上一章的 2.8.4 节我们练习了“边读边处理”——每读入一个数就立刻更新答案，处理完就丢掉。这样做很轻量，但有两个局限：数据没法留到后面再用（比如“先存下来，最后倒着输出”），也没办法回头比较（比如“某个数是不是这批数里的最大值”）。**数组解决的就是“存下来、反复用”这件事。**

本次训练结束后，你应该能够：

- 声明、初始化和访问一维数组，记住合法下标范围 `[0, n)`；
- 用一个循环完成数组的输入、输出、求和、最值、计数和查找；
- 用计数数组统计每个值出现的次数，并正确处理多组测试数据的数组重置；
- 统计连续段、枚举连续子数组，并用前缀和 O(1) 回答区间和询问；
- 声明二维数组，完成按行读入、按行输出等基础遍历；
- 理解冒泡排序的过程，并在实际题目中优先使用 `std::sort` 排整数。

> **数组的核心只有三件事：有多少个元素、每个元素在哪里、循环访问哪些下标。**

每个完整程序，我们都按同一条路线走：**先讲清楚概念，看最小示例，给出输入输出，再拿小数据把执行过程手算一遍，最后才是完整程序和常见错误**。手算不丢人——赛场上最快的 debug 手段，往往就是拿纸笔走一遍。

3.1 一维数组：连续的一排格子

3.1.1 声明与访问

声明数组的基本形式是：

```
类型 数组名[元素个数];
```

例如：

```
int score[5];
```

可以把它想象成五个连续的格子：

下标	0	1	2	3	4
元素	score[0]	score[1]	score[2]	score[3]	score[4]

数组下标从 **0** 开始，因此 score[0] 是第一个元素，score[4] 是第五个元素。访问或修改元素的方式与普通变量很像：

```
score[0] = 90;
score[1] = score[0] + 5;
```

数组中的元素类型必须相同。int score[5] 的五个格子都是 int，不能在其中某个格子里放字符串或 double。

3.1.2 [0, n)：下标范围与 a、a + n 的图解

长度为 n 的数组，合法下标是：

```
0, 1, 2, ..., n - 1
```

数学上把这个范围记作 [0, n)，叫**半开区间**：左端包含 0，右端不包含 n。数一数就知道里面恰好有 n 个整数。这套写法会贯穿后面的字符串、vector、排序和二分。

下面把数组、下标和两个记号画在一起。设已经读入了 n = 5 个数：

```
下标       0       1       2       3       4
         +-------+-------+-------+-------+-------+
  元素   |   7   |  -2   |   7   |  -5   |   0   |  （这一格不存在）
         +-------+-------+-------+-------+-------+
           ↑                               ↑
           a                             a + 5
```

- a 是数组**开头位置**的记号，从这个位置开始数，一格就是一个元素；
- a + n（这里是 a + 5）指向**最后一个元素再往后一格**的位置，它不属于数组；
- 区间写法 [a, a + n) 表示：从 a 开始，到 a + n 为止，但**不包含 a + n 那一格**——恰好覆盖 a[0] 到 a[n - 1] 共 n 个元素。

为什么要专门记住 a + n？因为 C++ 标准库的操作几乎都用它来表示“处理范围”，本章会反复见到这样的写法：

```
sort(a, a + n);        // 给 [0, n) 这 n 个元素排序
fill(a, a + n, 0);     // 把 [0, n) 这 n 个元素填成 0
```

这两行你现在只需要“认得”：它们都表示“对从 `a` 开始的连续 `n` 个元素操作”。本章的完整程序里会反复用到，用多了自然记住。

记住一个恒等式：**`[x, y)` 里的元素个数 = 右端 − 左端**。`[0, n)` 有 `n` 个，`[a, a + n)` 也是 `n` 个；`[1, n)` 有 `n - 1` 个，正好跳过 `a[0]`。

对应到循环上，三件事是一回事：

```
n 个有效元素   ⇔   下标 [0, n)   ⇔   for (int i = 0; i < n; i++)
```

3.1.3 数组越界：RE 的原因之一

原生数组不会自动检查下标。对于 `int score[5]`：

- `score[0]` 到 `score[4]` 合法；
- `score[5]` 越过了数组末尾；
- `score[-1]` 越过了数组开头。

越界访问属于未定义行为，可能表现为 RE，也可能暂时输出“看起来正确”的结果，甚至悄悄覆盖其他变量。不能因为本地运行没有崩溃，就认为它是安全的。

最常见的错误是多跑一次：

```
for (int i = 0; i <= n; i++)   // 危险：最后一轮访问 a[n]
```

若数组中有 `n` 个有效元素，遍历应写成：

```
for (int i = 0; i < n; i++)
```

建议把“数组长度为 `n` → 下标范围是 `[0, n)`”当成一个整体记忆，写循环条件时不再临场犹豫。

3.1.4 容量 `MAXN` 与有效长度 `n`

竞赛题通常会给出 `n` 的上限。例如题目保证 `n <= 1000`，可以预留一个固定容量：

```
#define MAXN 1000
int a[MAXN];
```

这里要区分两个概念：

- `MAXN` 是数组真正申请的**容量**；
- `n` 是本组数据实际使用的**有效长度**。

后续所有处理只访问 `a[0]` 到 `a[n - 1]`。不能因为数组开到了 `MAXN`，就把 `MAXN` 个位置都当成输入数据——后面没有读入的格子装着不确定的值。

也不要把下面这种写法当成标准 C++：

```
int n;
cin >> n;
int a[n];       // 非标准：标准 C++ 不允许运行时才确定长度的原生数组
```

有些编译器接受了这种扩展，但不保证评测机也接受。固定上限用 `int a[MAXN]`；长度运行时才确定的场合，C++ 的替代是 `vector<int> a(n)`，本系列后面会专门学，现在先知道方向即可。

容量也不是越大越好。一个 `int` 通常占 4 字节，`int a[1000000]` 约占 4 MB；二维数组还要再乘行数。看到 `n`、`m` 的范围后，先估算内存，再决定开多大。

3.2 初始化与读入

3.2.1 初始化：四种写法

```
int a[5] = {1, 2, 3, 4, 5}; // 完全初始化
int b[5] = {1, 2};          // 只给前两个，剩余自动补 0
int c[] = {3, 1, 4};        // 不写长度，编译器按初始值个数推断，c 有 3 个元素
int d[5] = {};              // 一个都不给，全部为 0
```

下面的程序里，`a` 只初始化前两个元素，`b` 全部清零，`c` 不写长度、由编译器按初始值个数推断。程序把 `a` 和 `b` 的 5 个元素逐个打印出来，并用 `sizeof(c) / sizeof(c[0])` 数出 `c` 的长度：

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

int main()
{
    int a[5] = {1, 2};
    int b[5] = {};
    int c[] = {3, 1, 4};

    for (int i = 0; i < 5; i++)
    {
        cout << a[i] << (i + 1 == 5 ? '\n' : ' ');
    }
    for (int i = 0; i < 5; i++)
    {
        cout << b[i] << (i + 1 == 5 ? '\n' : ' ');
    }
    cout << sizeof(c) / sizeof(c[0]) << '\n';
```

```
    return 0;
}
```

运行结果：

```
1 2 0 0 0
0 0 0 0 0
3
```

初始化列表里的元素不能多于数组容量（写 `int a[2] = {1, 2, 3};` 会直接编译错误），但允许少于容量。

`sizeof(c)` 是整个数组占用的字节数，`sizeof(c[0])` 是一个元素占用的字节数，两者相除就是元素个数。注意这个写法**只在定义数组的地方可靠**：以后把数组传给函数时，请把 `n` 一起传过去，不要指望在函数内部再用 `sizeof` 算出原数组长度（原因在学完函数传参后会讲清楚）。现阶段最稳妥的习惯只有一个——**始终显式维护 `n`**。

3.2.2 局部数组与全局数组

定义在函数内部的**局部数组**如果不初始化，元素值是不确定的，不能直接拿来计算。定义在所有函数外面的**全局数组**会自动初始化为 0——这是竞赛里计数数组、标记数组常放全局的原因：省心，且不占函数内的栈空间。

不过"放全局"不是万能药：大小仍要按数据范围合理估算，太大一样超内存；`3.4` 节还会看到，多组数据时全局数组的"旧值"反而会带来麻烦，需要主动重置。

3.2.3 读入 `n` 个数并求和

数组最常见的用法是"读入一批数据，再遍历这批数据"。下面的程序读入 `n` 个整数并计算总和：

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

#define MAXN 1000

int main()
{
    int n;
    cin >> n;

    int a[MAXN];
    for (int i = 0; i < n; i++)
    {
        cin >> a[i];
    }

    long long sum = 0;
    for (int i = 0; i < n; i++)
    {
        sum += a[i];
```

```
    }

    cout << sum << '\n';
    return 0;
}
```

输入：

```
5
7 -2 7 -5 3
```

输出：

```
10
```

拿小数据手算一遍，确认自己真的能“看见”循环的执行过程：

```
进入求和循环前：sum = 0

i    a[i]    执行 sum += a[i] 之后
0     7      sum = 0 + 7  = 7
1    -2      sum = 7 - 2  = 5
2     7      sum = 5 + 7  = 12
3    -5      sum = 12 - 5 = 7
4     3      sum = 7 + 3  = 10

i = 5 时条件 5 < 5 不成立，循环结束，输出 10
```

这里有三个值得固定下来的习惯：

1. 数组下标由循环变量 `i` 控制，循环条件写成 `i < n`；
2. 总和使用 `long long`——和可能超过 `int`，类型按**答案范围**选择（第一章的内容）；
3. `n` 是有效数据长度，后续循环都围绕 `n`，而不是 `MAXN`。

竞赛题会保证输入格式与题面一致（这里的 `n` 一定在 `1 ~ MAXN` 之间），所以程序直接用 `cin >> n`，不再额外检查。

3.3 遍历数组的六个固定套路

数组题看起来千变万化，入门阶段大多可以归结为“遍历每个元素时维护一个变量”。先把这张表过一遍，后面逐个展开：

目标	辅助变量的初始化	遍历时的操作
求和	`sum = 0`	`sum += a[i]`
计数	`count = 0`	满足条件时 `count++`
最大值	`maximum = a[0]`	若 `a[i] > maximum` 就更新
最小值	`minimum = a[0]`	若 `a[i] < minimum` 就更新
查找	`position = -1`	找到目标后记录下标，必要时 `break`
逆序	`left = 0`、`right = n - 1`	交换 `a[left]` 与 `a[right]`，两个下标向中间收拢

3.3.1 求和、计数、最值：一次遍历做完

上一章的 2.8.4 已经用“边读边处理”做过这件事：读一个数，立刻更新 `sum`、`count`、`maximum`，处理完就丢掉。现在数据先存进数组，遍历套路一模一样，只是把“读入”和“处理”拆成了两个循环。

唯一要特别注意的是**最大值的初始化**：如果数据可能全是负数，下面这种初始化就是错的：

```
int maximum = 0;   // 错误示范：0 可能不是数组里的任何一个数
```

输入 `-8 -3 -10` 时，程序会错误地输出 0。只要 `n > 0`，应直接使用数组的第一个元素：

```
int maximum = a[0];
int minimum = a[0];
```

然后从 `i = 1` 开始处理剩余元素。

另一个容易混的点是**下标和元素值**。题目问“最大值出现在哪个位置”时，要保存 `i`；题目问“最大值是多少”时，要保存 `a[i]`。遍历过程中可以同时维护：

```
if (a[i] > maximum)
{
    maximum = a[i];
    position = i;
}
```

题目若使用“第几个”这样的自然语言，通常要输出 `position + 1`；程序内部仍建议使用从 0 开始的下标，输出时再做转换。

3.3.2 综合例题：一次遍历统计一批数据

下面的程序同时完成四件事：找最大值、记录最大值第一次出现的位置、找最小值、统计负数个数。

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

#define MAXN 1000

int main()
{
    int n;
    cin >> n;

    int a[MAXN];
    for (int i = 0; i < n; i++)
    {
        cin >> a[i];
    }

    int maximum = a[0];
    int minimum = a[0];
    int maximumIndex = 0;
    int negativeCount = (a[0] < 0) ? 1 : 0;

    for (int i = 1; i < n; i++)
    {
        if (a[i] > maximum)
        {
            maximum = a[i];
            maximumIndex = i;
        }
        if (a[i] < minimum)
        {
            minimum = a[i];
        }
        if (a[i] < 0)
        {
            negativeCount++;
        }
    }

    cout << "maximum = " << maximum << '\n';
    cout << "maximum index = " << maximumIndex << '\n';
    cout << "minimum = " << minimum << '\n';
    cout << "negative count = " << negativeCount << '\n';
    return 0;
}
```

输入：

```
6
7 -2 7 -5 0 3
```

输出：

```
maximum = 7
maximum index = 0
minimum = -5
negative count = 2
```

手算验证一遍。初始状态取自 `a[0] = 7`：`maximum = 7`，`minimum = 7`，`maximumIndex = 0`，`negativeCount = 0`（7 不是负数）。随后逐个处理 `a[1]` 到 `a[5]`：

```
i    a[i]    maximum   maximumIndex   minimum   negativeCount
1    -2       7             0            -2           1
2     7       7             0            -2           1       （7 > 7 不成立，
maximum 与 maximumIndex 都不更新）
3    -5       7             0            -5           2
4     0       7             0            -5           2
5     3       7             0            -5           2
```

两个细节值得记住：

1. 更新条件用的是 `>` 而不是 `>=`。遇到重复最大值（本例的 `a[2] = 7`）时不会更新，最终保留**第一次**出现的位置；如果题目要求最后一次出现的位置，把条件改成 `a[i] >= maximum` 即可。
2. `(a[0] < 0) ? 1 : 0` 是第二章的条件运算符：成立取 1，不成立取 0。负数个数从第一个元素就开始统计，循环里只管剩下的。

整个程序只遍历数组一次，共 `n` 轮。求最值和计数**没有必要先排序**——排序是 `O(n log n)`，而一趟遍历是 `O(n)`。

3.3.3 查找：找到就 `break`

查找目标值第一次出现的位置：从左到右遍历，命中就记录下标并用 `break` 结束循环；若循环走完都没命中，`position` 保持 `-1`。`break` 在这里不只是省时间，还保证了“第一次”——不 `break` 而每次都覆盖 `position`，得到的会是最后一次出现的位置。

3.3.4 逆序：两个下标向中间靠拢

把数组原地倒过来，用左右两个下标：`left` 从 0 向右走，`right` 从 `n - 1` 向左走，每轮交换 `a[left]` 和 `a[right]`，直到两者相遇。这里第一次用到 `swap(x, y)`——C++ 标准库提供的“交换两个变量”的函数，来自头文件 `<utility>`：

```
#include <utility>
// ...
swap(a[left], a[right]);
```

把查找和逆序放进一个完整程序：先查找目标值第一次出现的位置，再把数组原地逆序后输出。

```cpp
#include <iostream>
#include <utility>
using namespace std;

#define MAXN 1000

int main()
{
    int n;
    cin >> n;

    int a[MAXN];
    for (int i = 0; i < n; i++)
    {
        cin >> a[i];
    }

    int target;
    cin >> target;

    int position = -1;
    for (int i = 0; i < n; i++)
    {
        if (a[i] == target)
        {
            position = i;
            break;
        }
    }

    for (int left = 0, right = n - 1; left < right; left++, right--)
    {
        swap(a[left], a[right]);
    }

    cout << "first index = " << position << '\n';
    for (int i = 0; i < n; i++)
    {
        cout << a[i] << (i + 1 == n ? '\n' : ' ');
    }
    return 0;
}
```

输入：

```
6
1 2 3 2 5 2
2
```

输出：

```
first index = 1
2 5 2 3 2 1
```

手算逆序过程。数组为 `1 2 3 2 5 2`，`left` 与 `right` 从两端出发：

```
轮次   left   right   交换前            交换后
初始    0      5      1 2 3 2 5 2
 1      0      5      交换 a[0] 和 a[5]  2 2 3 2 5 1
 2      1      4      交换 a[1] 和 a[4]  2 5 3 2 2 1
 3      2      3      交换 a[2] 和 a[3]  2 5 2 3 2 1
结束    3      2      left < right 不成立，停止
```

循环条件是 `left < right`，不是 `left <= right`。当数组长度为奇数时，正中间元素留在原地，天然正确；当长度为 1 时循环一次也不执行，数组仍然正确。交换 `a[i]` 和 `a[n - 1 - i]` 的“镜像下标”关系，值得对照上面的表看熟。

另外注意 `2` 出现了三次，输出 `first index = 1` 而不是 3 或 5——这正是 3.3.3 里 `break` 保证的“第一次”。

3.4 用数组解决这一批基础题

第三次训练的必做题里，有一类题反复出现同一种模式：**遍历数组时，用一个额外的数组把“统计信息”记下来**。本节按题目形态逐一拆解，所有完整程序都附小数据手算。

3.4.1 计数数组：用值当下标

计数数组的思路只有一句话：用“值本身”当下标，出现一次就加一。比如统计 0～100 的分数，就开一个 `int cnt[101]`，读到 75 就执行 `cnt[75]++`。这样一来，“每个数出现几次”自动按分数排好了序——遍历 `cnt[0]` 到 `cnt[100]` 输出即可。

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

#define MAXN 1000

int cnt[101];   // 全局数组，自动清零

int main()
{
    int n;
    cin >> n;

    int a[MAXN];
    for (int i = 0; i < n; i++)
    {
```

```
        cin >> a[i];
        cnt[a[i]]++;    // a[i] 这个值又出现了一次
    }

    for (int value = 0; value <= 100; value++)
    {
        if (cnt[value] > 0)
        {
            cout << value << ": " << cnt[value] << '\n';
        }
    }
    return 0;
}
```

输入：

```
8
75 90 75 60 90 100 60 75
```

输出：

```
60: 2
75: 3
90: 2
100: 1
```

手算：依次读入时，`cnt` 的变化是 `cnt[75]` 三次、`cnt[90]` 两次、`cnt[60]` 两次、`cnt[100]` 一次。输出阶段从下标 0 扫到 100，只打印计数大于 0 的项，天然按分数从小到大排列——不需要排序。

三个要点：

1. **计数数组的大小按“值域”开**，不是按 `n` 开。值在 0～100 就开到 101；
2. 计数前必须全为 0。这里用了全局数组自动清零的特性；
3. 值域太大时（比如值达到 `10^9`）开不下数组，这种题目要换别的办法，本章不涉及。

3.4.2 多组数据与数组重置

第二章 2.7.1 和 2.7.2 规定了多组输入的两套模板：给出组数 `t` 就用 `while (t--)`；读到 EOF 就用 `while (cin >> n)`。数组题再加一条新规则：

> **每一组数据都必须“从零开始”——上一组留下的统计信息，一组都不能带进来。**

看一个典型的错法：把计数数组放在 `main` 外面（全局），多组数据共用。第一组数完 `cnt[1] = 5`，第二组开始时 `cnt[1]` 还是 5，答案自然连环出错。修复办法是在每组开始时重置：

```
fill(cnt, cnt + 10, 0);   // 把 [0, 10) 全部填成 0
```

`fill` 来自头文件 `<algorithm>`，三个参数依次是**起点位置、终点位置（取不到）、要填入的值**，同样使用 `[a, a + n)` 记号。它是 C++ 里“按元素赋值”的工具；C 里常见的 `memset` 按字节填充，只能可靠地清零，`memset(a, 1, sizeof a)` 得到的并不是“每个元素都是 1”，初学阶段统一用 `fill` 就好。

完整程序：有 `testCount` 组数据，每组读 `n` 个 0～9 的整数，输出 0～9 各自出现的次数（一行 10 个数，空格隔开）。

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

#define MAXN 1000

int cnt[10];   // 值域是 0~9，开 10 格就够

int main()
{
    int testCount;
    cin >> testCount;

    while (testCount--)
    {
        int n;
        cin >> n;

        int a[MAXN];
        for (int i = 0; i < n; i++)
        {
            cin >> a[i];
        }

        fill(cnt, cnt + 10, 0);   // 关键：每组开始前清零
        for (int i = 0; i < n; i++)
        {
            cnt[a[i]]++;
        }

        for (int value = 0; value < 10; value++)
        {
            cout << cnt[value] << (value + 1 == 10 ? '\n' : ' ');
        }
    }
    return 0;
}
```

输入：

```
3
5
1 2 1 0 0
```

```
4
9 9 9 9
3
0 1 2
```

输出：

```
2 2 1 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 4
1 1 1 0 0 0 0 0 0 0
```

手算第一组：`0` 两次、`1` 两次、`2` 一次，其余为 0，所以输出 `2 2 1 0 0 0 0 0 0 0`。第二组全是 9，只有 `cnt[9] = 4`——如果忘记 `fill` 清零，第二组输出里 `0`、`1`、`2` 的位置会带着第一组的 `2 2 1`，直接 WA。第三组同理。

分清两种“不用重置”和“必须重置”：

- 数据数组 `a`：每组只访问 `[0, n)`，新一组会重新读入覆盖，**不需要**清零；
- 计数、标记这类“累加型”数组：每组开始**必须**清零（`fill`，或重新声明并 `= {}` 初始化）。

3.4.3 连续段统计

有些题目要数“连续的一段”。例如气温记录里连续多少天高于 0 度、队伍里连续几个人没有缺勤。**段（run）**的定义是：满足某条件且位置相邻的最大连续元素块。统计段数有一个非常固定的写法：

> 遍历到元素 `a[i]` 时，如果它**自己是段的起点**，段数加一。`a[i]` 是起点 ⇔ 它满足条件，并且（它是第一个元素，或者前一个元素 `a[i-1]` 不满足条件）。

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

#define MAXN 1000

int main()
{
    int n;
    cin >> n;

    int a[MAXN];
    for (int i = 0; i < n; i++)
    {
        cin >> a[i];
    }

    int segmentCount = 0;
    for (int i = 0; i < n; i++)
    {
        if (a[i] != 0 && (i == 0 || a[i - 1] == 0))
        {
```

```
            segmentCount++;
        }
    }

    cout << segmentCount << '\n';
    return 0;
}
```

输入（以 0 为"间隔"，统计连续非零段）：

```
10
1 0 0 3 4 0 5 0 0 2
```

输出：

```
4
```

手算：逐个元素判断"是不是段的起点"。

```
i    a[i]    是不是起点？                      segmentCount
0     1      a[0] != 0，且 i == 0            → 1
1     0      不满足条件                       1
2     0      不满足条件                       1
3     3      a[3] != 0，且 a[2] == 0         → 2
4     4      a[4] != 0，但 a[3] != 0          2
5     0      不满足条件                       2
6     5      a[6] != 0，且 a[5] == 0         → 3
7     0      不满足条件                       3
8     0      不满足条件                       3
9     2      a[9] != 0，且 a[8] == 0         → 4
```

四段分别是 `[1]`、`[3 4]`、`[5]`、`[2]`。判断条件里的 `i == 0 || a[i - 1] == 0` 用到了短路求值：当 `i == 0` 时不会访问 `a[-1]`，这正是第二章短路规则的保护作用。如果把条件写成 `a[i - 1] == 0 || i == 0`，`i = 0` 时会先访问 `a[-1]`——越界。

3.4.4 枚举所有连续子数组

子数组指原数组中"连续的一段"。注意它和"子序列"不同：子数组要求元素在原数组中紧挨着，比如 `a = [1, 2, 3]` 的子数组有 `[1]`、`[1 2]`、`[1 2 3]`、`[2]`、`[2 3]`、`[3]` 共 6 个，`[1 3]` 不是。

枚举所有连续子数组只需要两层循环：外层定左端点 `left`，内层让右端点 `right` 从 `left` 一路走到 `n - 1`，每走一步就多纳入一个元素。经典应用：**最大子段和**——找和最大的连续子数组（至少选一个元素）。

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

```cpp
#define MAXN 200

int main()
{
    int n;
    cin >> n;

    int a[MAXN];
    for (int i = 0; i < n; i++)
    {
        cin >> a[i];
    }

    long long best = a[0];   // 至少选一个元素，初值就是 a[0]
    for (int left = 0; left < n; left++)
    {
        long long currentSum = 0;
        for (int right = left; right < n; right++)
        {
            currentSum += a[right];        // [left, right] 的和
            if (currentSum > best)
            {
                best = currentSum;
            }
        }
    }

    cout << best << '\n';
    return 0;
}
```

输入：

```
5
-2 11 -4 13 -5
```

输出：

```
20
```

手算：`n = 5` 时共有 `5 + 4 + 3 + 2 + 1 = 15` 个连续子数组。内层循环复用 `currentSum`，每个 `right` 只做一次加法，所以整体是 n^2 量级。关键几步：

```
left = 0: 依次得到 [-2] = -2, [-2,11] = 9, [-2,11,-4] = 5,
                   [-2,11,-4,13] = 18, [-2,11,-4,13,-5] = 13
left = 1: 依次得到 [11] = 11, [11,-4] = 7, [11,-4,13] = 20 ← 最大值
```

```
left = 2: [-4] = -4, [-4,13] = 9, [-4,13,-5] = 4
left = 3: [13] = 13, [13,-5] = 8
left = 4: [-5] = -5
```

`best` 初始为 `a[0]` 而不是 0，这保证了"全负数"输入也正确：输入 `4` 个数 `-1 -2 -3 -4` 时，答案是 `-1`（必须选一个），程序输出 `-1`。如果随手把 `best` 初始成 0，这种输入就会错误地输出 0。

复杂度方面，`n = 200` 时两层循环约两万次，非常宽裕；但 `n = 10^5` 时约 `10^10` 次，必然超时（第二章 2.13.4 的量级直觉）。最大子段和的 `O(n)` 做法属于进阶内容，本题数据范围小，枚举够用。

3.4.5 前缀和：区间和的 O(1) 查询

如果题目**反复**询问"第 `l` 个到第 `r` 个数的和"（比如询问 `10^5` 次），每次重新累加一遍就是 `O(nq)`，太慢。**前缀和**把"从头累加的结果"提前存下来，让每个区间和变成一次减法。

定义：`pre[i]` 表示 `a[0]` 到 `a[i - 1]` 这前 `i` 个数的和（前 0 个数的和是 0）。递推式只有一行：

```
pre[0] = 0
pre[i + 1] = pre[i] + a[i]      （i 从 0 到 n - 1）
```

于是闭区间 `[l, r]` 的和 = `pre[r + 1] - pre[l]`：从开头加到 `r`，减去开头加到 `l - 1` 的部分，剩下的正好是 `[l, r]`。

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

#define MAXN 1000

int a[MAXN];
long long pre[MAXN + 1];   // 长度是 n + 1，下标 0 到 n

int main()
{
    int n;
    cin >> n;

    for (int i = 0; i < n; i++)
    {
        cin >> a[i];
    }

    pre[0] = 0;
    for (int i = 0; i < n; i++)
    {
        pre[i + 1] = pre[i] + a[i];
    }

    int queryCount;
    cin >> queryCount;
```

```
    while (queryCount--)
    {
        int left, right;
        cin >> left >> right;              // 0 开始的闭区间 [left, right]
        cout << pre[right + 1] - pre[left] << '\n';
    }
    return 0;
}
```

输入：

```
6
1 2 3 4 5 6
3
0 2
1 4
3 5
```

输出：

```
6
14
15
```

先手算 `pre`（注意它比 `a` 长一格，整体右移了一位）：

```
i        0    1    2    3    4    5    6
pre[i]   0    1    3    6   10   15   21
         ↑    └── pre[1] = a[0] = 1 ──┘
              └── pre[2] = a[0] + a[1] = 3 ──┘   …… 依此类推
```

三次询问：

```
[0, 2] → pre[3] - pre[0] = 6 - 0  = 6
[1, 4] → pre[5] - pre[1] = 15 - 1 = 14
[3, 5] → pre[6] - pre[3] = 21 - 6 = 15
```

前缀和是本章最容易写错下标的地方：`pre` 比 `a` 长一格，询问答案是 `pre[r + 1] - pre[l]`。背公式不如理解——`pre[k]` 永远是“前 `k` 个数的和”，`[l, r]` 有 `r - l + 1` 个数，用“前 `r + 1` 个”减掉“前 `l` 个”正好。写完后拿上面的手算表对一遍，比死记可靠。

3.5 二维数组：把一维数组排成表格

3.5.1 声明、下标与读入输出

二维数组是一维数组的推广：一整排格子摆成一张表。声明形式：

```
int matrix[3][4];   // 3 行 4 列
```

合法下标为 `matrix[0][0]` 到 `matrix[2][3]`：第一个下标是**行**，范围 `[0, 行数)`；第二个下标是**列**，范围 `[0, 列数)`。访问它要两层循环——外层控制行，内层控制这一行里的列：

```
for (int row = 0; row < n; row++)
{
    for (int column = 0; column < m; column++)
    {
        cin >> matrix[row][column];
    }
}
```

`row` 和 `column` 写反通常不会编译报错，却会得到错误答案；若因此越过边界，还可能 RE。读入时按"一行一行、每行从左到右"的顺序，正好对应题目输入的格式。

完整程序：读入 `n` 行 `m` 列的矩阵，原样输出，再输出每一行的和。

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

#define MAXN 100

int main()
{
    int n, m;
    cin >> n >> m;

    int matrix[MAXN][MAXN];
    long long rowSum[MAXN] = {};

    for (int row = 0; row < n; row++)
    {
        for (int column = 0; column < m; column++)
        {
            cin >> matrix[row][column];
            rowSum[row] += matrix[row][column];   // 读入时顺手累加
        }
    }

    for (int row = 0; row < n; row++)
    {
        for (int column = 0; column < m; column++)
        {
```

```
            cout << matrix[row][column] << (column + 1 == m ? '\n' : ' ');
        }
    }

    for (int row = 0; row < n; row++)
    {
        cout << rowSum[row] << (row + 1 == n ? '\n' : ' ');
    }
    return 0;
}
```

输入：

```
2 3
1 2 3
4 5 6
```

输出：

```
1 2 3
4 5 6
6 15
```

执行顺序就是上面两层循环的老规矩：外层 `row = 0` 时内层走完 `column = 0, 1, 2`，读完第一行；然后 `row = 1`，读完第二行。行和在读入时顺便累加，不需要再遍历一次；但如果后续还要用到矩阵里的单个元素，矩阵本身得留着，不能省。

3.5.2 遍历顺序

默认按“行优先”遍历：外层行、内层列，访问顺序是 `(0,0) (0,1) … (0,m-1) (1,0) …`。如果题目要求按列处理（比如统计每一列的和），把两层循环的角色互换即可——外层列、内层行，访问顺序变成 `(0,0) (1,0) … (n-1,0) (0,1) …`。

二维数组的访问量是 `n * m` 个位置，复杂度写作 `O(nm)`。转置、旋转、对角线这类更有意思的二维操作，统一留到第 4 章再展开——本章先确保“读得进、走得遍、输得出”。

3.6 排序

3.6.1 排序在做什么

排序就是把一批数据按大小关系重新排列。默认的升序要求：

```
a[0] <= a[1] <= ... <= a[n - 1]
```

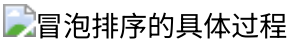
排序之后，最大值、最小值、中位数（排序后正中间的那个数）、相邻元素之间的关系都会变得一目了然。排序不改变元素的个数，只改变顺序。

本章主线只要求两件事：理解**冒泡排序**的原理（它把“交换”和“循环边界”讲得非常清楚），以及在正式题目中用 `sort` 对**整数**排序。自定义比较规则、结构体排序放在本章后面的【拓展】小节里（3.6.4、3.6.5）。

3.6.2 冒泡排序：逐轮把最大值推到最后

冒泡排序每次比较相邻的一对元素：如果左边比右边大，就交换。一轮从左扫到右之后，当前范围内**最大的元素会被一路换到最右边**，就像气泡升到水面——这就是名字的由来。下一轮不再理会已经归位的末尾元素，范围缩短一格，继续冒泡。

下面的动画展示完整过程：橙色元素是当前正在比较的一对，绿色元素表示已经在本轮或前几轮中确定位置的元素。


冒泡排序的具体过程

拿 `3 5 1 4 2` 逐轮手算，把“每轮谁归位”看清楚：

```
初始：  3 5 1 4 2

第 1 轮（比较到倒数第 2 个）：
  (3,5) 不交换；  (5,1) 交换 → 3 1 5 4 2；  (5,4) 交换 → 3 1 4 5 2；
  (5,2) 交换 → 3 1 4 2 5
  本轮最大值 5 归位：3 1 4 2 | 5

第 2 轮（不用再碰 5）：
  (3,1) 交换 → 1 3 4 2 5；  (3,4) 不交换；  (4,2) 交换 → 1 3 2 4 5
  本轮最大值 4 归位：1 3 2 | 4 5

第 3 轮（不用再碰 4、5）：
  (1,3) 不交换；  (3,2) 交换 → 1 2 3 4 5
  3 归位：1 2 | 3 4 5

第 4 轮：一轮比较下来没有发生任何交换，说明已经有序，提前结束。
```

完整程序（含“本轮无交换就提前结束”的优化）：

```cpp
#include <iostream>
#include <utility>
using namespace std;

#define MAXN 1000

int main()
{
    int n;
    cin >> n;

    int a[MAXN];
```

```
    for (int i = 0; i < n; i++)
    {
        cin >> a[i];
    }

    for (int pass = 0; pass < n - 1; pass++)
    {
        bool swapped = false;
        for (int j = 0; j < n - pass - 1; j++)
        {
            if (a[j] > a[j + 1])
            {
                swap(a[j], a[j + 1]);
                swapped = true;
            }
        }

        if (!swapped)
        {
            break;
        }
    }

    for (int i = 0; i < n; i++)
    {
        cout << a[i] << (i + 1 == n ? '\n' : ' ');
    }
    return 0;
}
```

输入：

```
5
3 5 1 4 2
```

输出：

```
1 2 3 4 5
```

内层上界 `n - pass - 1` 不是随手写的，有两个理由：

1. 每轮比较的是 `a[j]` 与 `a[j + 1]`，`j + 1` 不能超过最后一个下标 `n - 1`，所以 `j` 最大是 `n - 2`；
2. 每完成一轮，末尾已经有一个元素归位（第 1 轮后 `a[n-1]` 就位，第 2 轮后 `a[n-2]` 就位……），下一轮的内层上界跟着收缩，就是 `n - pass - 1`。

`n - 1` 个元素归位后，剩下的那个自动就位，所以外层只需 `n - 1` 轮。最坏情况下冒泡排序要比较约 `n² / 2` 次（`O(n²)`）：`n = 100000` 时约 `5 * 10^9` 次，完全不可行。它适合帮我们把交换和边界想明白，也适合 `n` 只有几百的手写练习；正式题目请交给下面的 `sort`。

3.6.3 `std::sort`：整数排序的默认选择

C++ 在 `<algorithm>` 中提供了 `sort`，默认把整数从小到大排：

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

#define MAXN 1000

int main()
{
    int n;
    cin >> n;

    int a[MAXN];
    for (int i = 0; i < n; i++)
    {
        cin >> a[i];
    }

    sort(a, a + n);                 // 从小到大
    for (int i = 0; i < n; i++)
    {
        cout << a[i] << (i + 1 == n ? '\n' : ' ');
    }

    sort(a, a + n, greater<int>()); // 从大到小
    for (int i = 0; i < n; i++)
    {
        cout << a[i] << (i + 1 == n ? '\n' : ' ');
    }
    return 0;
}
```

输入：

```
6
3 1 4 1 5 9
```

输出：

```
1 1 3 4 5 9
9 5 4 3 1 1
```

四个要点：

1. 头文件是 `<algorithm>`；降序的 `greater<int>()` 需要 `<functional>`；
2. 两个位置参数是 `a` 和 `a + n`，处理的正是 3.1.2 图解过的半开区间 `[a, a + n)`——右端取不到，所以不会访问 `a[n]`；
3. 只排序有效区间。如果实际数据只有前 `n` 个，误写成 `sort(a, a + MAXN)` 会把没读入的格子一起参与排序，结果全错；
4. `sort` 平均 `O(n log n)`，`n = 10^5` 轻松通过。重复元素不影响正确性（`1` 出现了两次，照样排好）。

从 C 迁移过来的同学可能还会见到 `qsort`：它需要手动传元素大小和比较函数，类型信息不直观，竞赛代码统一用 `sort`。

【拓展】3.6.4 自定义比较：sort 的第三个参数

如果排序规则不是“数值大小”，比如“绝对值小的在前、绝对值相同时数值小的在前”，就要给 `sort` 传第三个参数——一个比较规则。本系列到后续才系统学函数，这里先照抄结构用一次：

```
#include <algorithm>
#include <cstdlib>
#include <iostream>
using namespace std;

#define MAXN 1000

bool cmp(int x, int y)
{
    if (abs(x) != abs(y))
    {
        return abs(x) < abs(y);   // 绝对值小的排前面
    }
    return x < y;                 // 绝对值相同，数值小的排前面
}

int main()
{
    int n;
    cin >> n;

    int a[MAXN];
    for (int i = 0; i < n; i++)
    {
        cin >> a[i];
    }

    sort(a, a + n, cmp);
    for (int i = 0; i < n; i++)
    {
        cout << a[i] << (i + 1 == n ? '\n' : ' ');
    }
    return 0;
}
```

输入：

```
6
-3 1 -1 2 -2 3
```

输出：

```
-1 1 -2 2 -3 3
```

`cmp(x, y)` 的约定是：**返回 `true` 表示 `x` 应该排在 `y` 前面**。上面六个数的绝对值是 `3 1 1 2 2 3`，先按绝对值分组排序；绝对值同为 1 的 `-1` 和 `1`，再由第二条规则把 `-1` 排在前面。`abs` 取整数的绝对值，来自 `<cstdlib>`。

再进一步的"结构体排序"（比如按成绩排序、成绩相同按姓名排序）需要 `struct` 知识，后续学完结构体后回来补，练习里先留标记。

【拓展】3.6.5 MEX 与滑动窗口：两个拓展方向

题单的拓展部分会出现两个新概念，先在这里混个脸熟，能做就做，做不动就跳过，不影响必做题。

MEX 是"集合中没出现过的最小非负整数"（minimum excludant）。例如 `{0, 1, 2, 4, 5}` 里有 0、1、2，缺 3，MEX 就是 3。题目保证每个数都满足 `0 <= a[i] <= 1000`，恰好可以用计数数组解决：把出现过的数标记进 `cnt`，从下标 0 开始向上扫，第一个 `cnt` 为 0 的位置就是答案。

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

#define MAXN 1000

int cnt[MAXN + 1];   // 值域 0~1000，下标 0~1000 都要放得下

int main()
{
    int n;
    cin >> n;

    int a[MAXN];
    for (int i = 0; i < n; i++)
    {
        cin >> a[i];
        cnt[a[i]] = 1;   // 标记"出现过"
    }

    int mex = 0;
    while (cnt[mex] == 1)
    {
        mex++;
    }
```

```
    cout << mex << '\n';
    return 0;
}
```

输入：

```
5
0 1 2 4 5
```

输出：

```
3
```

手算：`cnt[0] = cnt[1] = cnt[2] = 1`，`cnt[3] = 0`，循环在 `mex = 3` 处停下。若数组恰好包含 `0 ~ n - 1` 各一个，MEX 会是 `n`（最大 1000），所以计数数组要开到 `MAXN + 1` 格，把下标 0～1000 全部覆盖。

滑动窗口是处理"连续子数组"问题的进阶思路：维护一个左端点 `left`、右端点 `right` 都只向右移动的区间（像一扇只会右滑的窗），配合一个区间和 `currentSum`，把 3.4.4 里 `O(n²)` 的枚举压成 `O(n)`。以"和至少为 `S` 的最短连续子数组长度"为例（假设所有数都是正数）：

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

#define MAXN 1000

int main()
{
    int n;
    long long targetSum;
    cin >> n >> targetSum;

    int a[MAXN];
    for (int i = 0; i < n; i++)
    {
        cin >> a[i];
    }

    int bestLength = n + 1;   // “还没找到”的标记
    long long currentSum = 0;
    int left = 0;

    for (int right = 0; right < n; right++)
    {
        currentSum += a[right];              // 右端纳入新元素
        while (currentSum >= targetSum)      // 条件满足就尝试收缩左端
        {
```

```
            if (right - left + 1 < bestLength)
            {
                bestLength = right - left + 1;
            }
            currentSum -= a[left];
            left++;
        }
    }

    if (bestLength == n + 1)
    {
        cout << -1 << '\n';
    }
    else
    {
        cout << bestLength << '\n';
    }
    return 0;
}
```

输入：

```
6 15
1 2 3 4 5 6
```

输出：

```
3
```

手算：`right` 走到下标 4 时 `currentSum = 1+2+3+4+5 = 15`，记录长度 5，随后左端收缩：减去 `a[0] = 1` 变成 14，小于 15 停止。`right = 5` 纳入 6 后 `currentSum = 20`，收缩过程依次经过窗口 `[2,5] = 3+4+5+6 = 18`（长度 4）、`[3,5] = 4+5+6 = 15`（长度 3，记录后再次收缩变成 11，停止）。最短是 3。整个过程中 `left`、`right` 都只向右移动，总计 `O(n)`——这正是它比 3.4.4 的枚举快的原因。

3.7 提交前，用四种小数据手算一遍

数组题的错误经常只在边界输入上出现。写完后，至少用纸笔走下面四组数据（含义在前面各节都讲过，这里只做清单）：

1. `n = 1`：最值都是 `a[0]`；逆序循环一次不执行；排序无需交换。程序里从 `i = 1` 开始的循环要能自然跳过。
2. **全是负数**：最大值初始化必须是 `a[0]`（3.3.1）；最大子段和的 `best` 同理（3.4.4）。
3. **有重复元素**：确认题目要"第一次"还是"最后一次"出现的位置（3.3.2）；排序不受重复影响（3.6.3）。
4. **`n = 0` 或空区间**：若题目允许空数组，不能访问 `a[0]`；`[0, 0)` 表示空区间，循环 `0` 次，`sort(a, a)` 合法。本次题单一般都保证 `n >= 1`。

多组数据再补第 5 组：**两组都含相同的小值**，检查上一组的计数有没有漏进这一组（3.4.2 的 `fill`）。

3.8 常见错误清单

1. **越界（RE / 莫名其妙 WA）**：循环边界写成 `i <= n`，访问了 `a[n]`。统一用 `[0, n)`，写完检查最后一次访问的下标是不是 `n - 1`。
2. **最值初始化错（WA）**：最大值随手初始化为 0，全负数组输出 0。非空数组从 `a[0]` 开始；空数组先按题意特判。
3. **排序范围错（WA）**：实际数据只有前 `n` 个却写 `sort(a, a + MAXN)`；或者输入从 `a[1]` 开始、排序却写 `[a, a + n)`。排序前先回答两个问题：数据从哪开始、一共有几个。
4. **多组数据没重置（WA）**：计数、标记数组带着上一组的值。每组开始时 `fill(cnt, cnt + 大小, 0)`；普通数据数组不需要清，重新读入即覆盖。
5. **前缀和下标错（WA）**：`pre` 长度是 `n + 1`，区间 `[l, r]` 的答案是 `pre[r + 1] - pre[l]`。写完对照 3.4.5 的手算表检查。
6. **`memset` 误用（WA）**：`memset` 按字节填充，按值初始化请统一用 `fill` 或 `= {}`，原因见 3.4.2。
7. **二维数组行列写反（WA / RE）**：`n` 行 `m` 列就写 `row < n`、`column < m`，以题目为准，不要凭变量名猜。
8. **连续段判断越界（RE）**：`a[i - 1]` 必须放在 `i == 0 ||` 的右侧，利用短路保护（3.4.3）。
9. **变长原生数组（CE 风险）**：`int a[n]` 不是标准 C++。固定上限用 `int a[MAXN]`；运行时才定长等学完用 `vector`。

3.9 本章小结

```
数组 = 同类型元素的连续存储
    ↓
下标从 0 开始，合法范围是 [0, n)，最后一个下标是 n - 1
    ↓
一次遍历完成求和、计数、最值、查找；两个下标向中间收拢完成逆序
    ↓
计数数组 cnt[a[i]]++；多组数据记得 fill 重置
    ↓
连续段看“是不是起点”；子数组用两层循环枚举
    ↓
前缀和 pre[r + 1] - pre[l] 把区间和变成一次减法
    ↓
二维数组两层循环，按行读入、按行输出
    ↓
理解冒泡排序，正式排序交给 std::sort(a, a + n)
```

再固定两条容易丢分的习惯：数组容量 `MAXN` 按数据范围估算，不要随手开大；非空数组求最值从 `a[0]` 初始化，`sum`、`best` 等答案变量按答案范围选 `long long`。

从 C 迁移到 C++，本章用到的替换关系：

C 里的习惯	C++ 中的写法	说明
`int a[n]`（变长数组）	`int a[MAXN]` / 后续学 `vector<int> a(n)`	固定容量用常量；运行时才定长等 `vector`

C 里的习惯	C++ 中的写法	说明
命名常量	`#define MAXN 1000`	简单竞赛常量先用 `#define`，需要类型与作用域时再了解 `constexpr`
`memset(a, 1, sizeof a)`	`fill(a, a + n, 1)`	按元素赋值，语义直白
`qsort(...)`	`sort(a, a + n)`	类型安全，默认升序；降序加 `greater<int>()`
手写交换 `t = x; x = y; y = t;`	`swap(x, y)`	标准库直接提供
手写完整遍历	范围 `for`（如 `for (int v : c)`）	只在“遍历全部元素、不需要下标”时用；`[0, n)` 场景仍用下标循环

3.10 配套练习

第三次训练题单已发布在 SYUCTOJ：2026 级 SYUCT-ACM 新生第三次完整训练（**比赛仍在进行中，题目、顺序与分值以 OJ 页面为准**）。下面按“基础 / 拓展”分级，做完基础部分再碰拓展。

基础部分（对应本章 3.1～3.6.3 主线，必做；3.6.4、3.6.5 是选做拓展，见“拓展部分”）

每道题都先写出“输入 → 手算 → 输出”，再写程序；提交前过一遍 3.7 的四种小数据。

1. 读 `n` 个整数，输出和、最大值、最小值（3.2.3、3.3.2）。对应题单 A《数组练习1》。
2. 查找目标值第一次出现的下标，找不到输出 `-1`；再把数组逆序输出（3.3.3、3.3.4）。
3. 读入 `n` 个 0～100 的整数，输出每个值出现的次数（3.4.1）。对应题单 C《梦中的统计》、D《小明不会数》、E《不要质数》这一类“计数”题。
4. 多组数据，每组统计 0～9 各出现几次，注意每组重置（3.4.2）。
5. 统计连续非零段的个数；变形：统计最长的一段有多长（3.4.3）。对应题单 F《统计天数》这类“连续段”题。
6. `n <= 200`，求最大子段和（3.4.4）。
7. 多次询问区间 `[l, r]` 的和，用前缀和回答（3.4.5）。
8. 读入 `n * m` 矩阵并原样输出，输出每行之和、每列之和（3.5）。
9. 手写冒泡排序完成升序（3.6.2），再用 `sort` 完成升序与降序（3.6.3）。对应题单 Y～AD 的排序系列与《求和+排序》。

拓展部分（对应【拓展】小节，选做）

1. 按绝对值排序，绝对值相同按数值排序（3.6.4）。
2. 求 MEX（3.6.5）。对应题单 V《MEX（easy version）》、W《Max Mex》。
3. 所有数为正数时，求和至少为 `S` 的最短连续子数组长度（3.6.5 的滑动窗口写法）；想想为什么全正数这个前提重要。
4. 结构体排序（按成绩排序、成绩相同按名字排序）：等第 5 章学完 `struct` 再回来做，本题单可先跳过。
5. X《MEX（hard version）》、O～Q《树的质量》系列：需要本章概念加更多技巧，建议完成基础部分后再挑战。

> 练习中出现的新词都在这里解释一下：**中位数**是排序后正中间位置的数（`n` 为奇数时唯一确定，`n` 为偶数时题目会说明取哪一个）；**最大子段和**是“和最大的连续子数组”的和；**MEX** 见 3.6.5。**子数组**要求连

> 续，**子序列**不要求——读题时认准这两个字。