基础算法 · 第零章

复杂度与常用技巧

先估计代价，再选择方法

HGNU-ACM 实验室算法组

配套：4 道例题，13 道练习与知识点题包

复杂度 · 二分 · 前缀和 · 差分 · 双指针

本章目录 | 前半

复杂度：估算程序的代价

空间复杂度：内存也要预算

二分：每次扔掉一半

前缀和：一次累加，随时查询

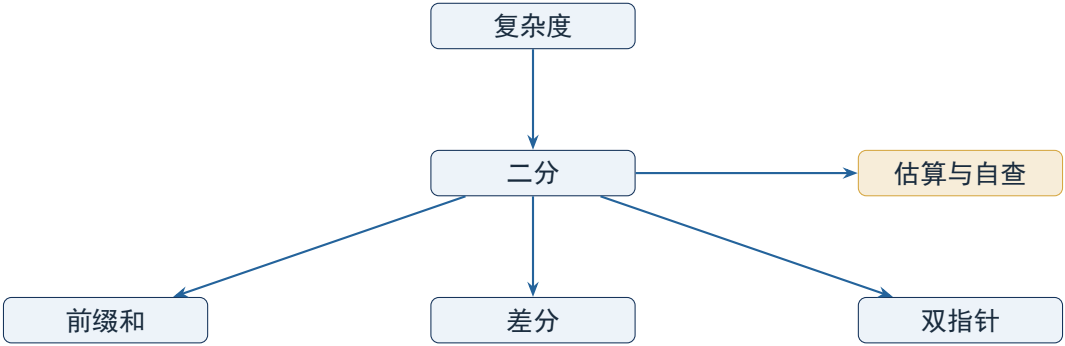
本章目录｜后半

差分：区间修改只动两个点

双指针：让两个下标一起走

练习与题包索引

本章地图：先会估算，再学四件工具



- 复杂度回答“这个做法能不能在时限内跑完”。
- 二分、前缀和、差分、双指针都是用单调性或预处理减少重复计算。
- 每节先讲适用条件，再用图推演，最后逐题写代码。

为什么需要复杂度

- 同一道题，写法不同，运行时间可能相差成千上万倍。
- 评测机每秒大约完成 10^8 次简单运算；超过时限就会 TLE。
- 复杂度不是精确秒数，而是操作次数随输入规模 n 增长的趋势。
- 看到数据范围，先反推“允许的复杂度”，再选算法。

本课的目标

给你一段程序或一个思路，能在读题阶段估计它大约要做多少次运算、会不会超时。

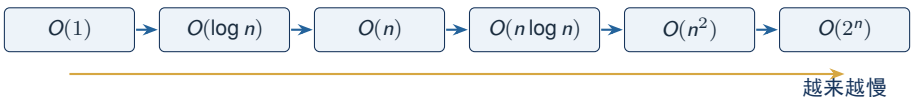
大 O 记号：忽略常数，只看增长

- 若存在正常数 C 和 N，当 $n > N$ 时 $f(n) \le C \cdot g(n)$，记作 $f(n) = O(g(n))$。
- 只保留增长最快的一项，并去掉系数：$3n + 5 = O(n)$，$\frac{1}{2}n^2 + 100n = O(n^2)$，$2n \log n + 7n = O(n \log n)$。
- 常数操作、输入输出、赋值都记作 $O(1)$。
- 大 O 描述的是上界；竞赛里常用来表示“最坏情况”的量级。

写法

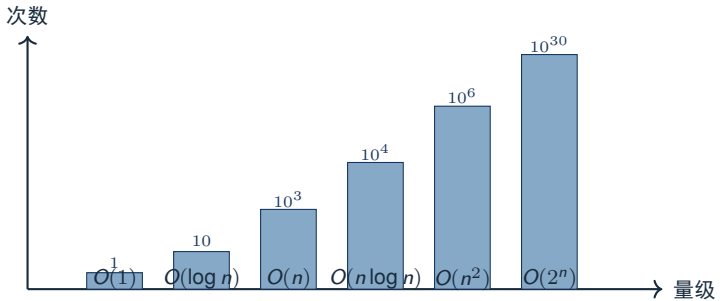
把 n 放在函数里：$O(1)$、$O(\log n)$、$O(n)$、$O(n \log n)$、$O(n^2)$、$O(2^n)$。

常见量级：从快到慢



- $O(1)$：与 n 无关，例如访问 `a[i]`。
- $O(\log n)$：每次规模减半，例如二分；$n = 10^9$ 时约 30 次。
- $O(n)$：扫一遍数组；$O(n \log n)$：排序、分治。
- $O(n^2)$：双重循环；$O(2^n)$：枚举子集、朴素递归。

柱状图：$n = 1000$ 时的相对操作次数



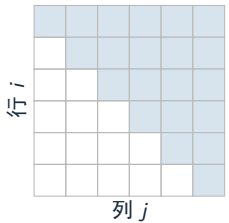
柱高只为示意，不按比例；$O(2^n)$ 在 $n = 1000$ 时已是天文数字。

直觉

n 每增大一倍：$O(n)$ 的工作量翻倍，$O(n^2)$ 翻四倍，$O(2^n)$ 直接平方级爆炸。

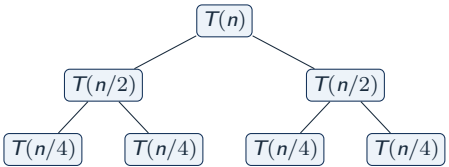
数循环：看最内层语句执行多少次

```
for (int i = 1; i <= n; ++i) {
    for (int j = i; j <= n; ++j) {
        ++ans; // 内层语句
    }
}
```



- 内层执行次数为 $n + (n-1) + \cdots + 1 = \frac{n(n+1)}{2}$。
- 只保留最高次项：$O(n^2)$；不要写成 $O(n(n+1)/2)$。
- 并列的两个循环相加：$O(n) + O(n) = O(n)$；嵌套循环相乘。

数递归：把调用画成一棵树



- 每层所有节点的问题规模之和都是 n。
- 共有 $\log_2 n + 1$ 层。
- 总代价 $T(n) = 2T(n/2) + n = O(n \log n)$。

另一个极端

朴素递归 $f(n) = f(n-1) + f(n-2)$ 会反复算同一子问题，调用次数约 2^n，记作 $O(2^n)$。

10^8 次操作的直觉

复杂度	1 秒大致可承受	典型做法
$O(n)$	$n \le 10^8$	一次扫描
$O(n \log n)$	$n \le 2 \times 10^6$	排序、分治
$O(n^2)$	$n \le 10^4$	双重循环
$O(n^3)$	$n \le 500$	三重循环
$O(2^n)$	$n \le 25$	子集枚举

- 时限 1 秒约 10^8 次；2 秒约 2×10^8 次，但常数大的程序要打折。
- $n = 10^5$ 时，$O(n^2)$ 是 10^{10}，必然超时；$O(n \log n)$ 只有约 1.7×10^6，很安全。
- 先看数据范围，再决定是否需要优化。

常见误判（一）：常数与 break

- $O(2n)$、$O(100n)$ 都记作 $O(n)$；常数只影响实际运行快慢，不改变量级。
- 但常数不能无限大：100 层嵌套的单次操作，实际也会很慢。
- break 只跳出当前一层循环：

```
for (int i = 1; i <= n; ++i) {
    for (int j = 1; j <= n; ++j) {
        if (a[i] + a[j] == x) {
            break; // 只结束 j 这一层
        }
    }
}
```

结论

最坏情况下仍是 $O(n^2)$；只有确定每层平均只走常数次时，才能降复杂度。

常见误判（二）：递归展开与均摊

- 递归式要展开看总工作量，不能只看递归深度。
- $T(n) = 2T(n/2) + n$：深度 $\log n$，但每层合并 n，总计 $O(n \log n)$。
- 朴素斐波那契递归深度只有 n，调用次数却约 2^n。
- 循环内调用 `reverse`、`sort` 或字符串拼接，要把这些操作的代价乘进去。

例子

在长度 n 的循环里每次拼接一个长度为 n 的字符串，总代价可能是 $O(n^2)$，不是 $O(n)$。

例 0.1 | 数清这段程序

```
int ans = 0;
for (int i = 1; i <= n; ++i) {
    for (int j = i; j <= n; j += i) {
        ++ans; // j = i, 2i, 3i, ...
    }
}
```

- 固定 i 时，内层执行 $\lfloor n/i \rfloor$ 次。
- 总次数 $n/1 + n/2 + \cdots + n/n = n\left(1 + \frac{1}{2} + \cdots + \frac{1}{n}\right)$。
- 调和级数约为 $\ln n$，所以总复杂度是 $O(n \log n)$。
- 不能因为“有两层循环”就写成 $O(n^2)$。

复杂度自查表

1. 输入规模 n 最大是多少？时限几秒？
2. 最内层语句执行了多少次？用求和式写出来。
3. 递归：每层代价多少？共有多少层？是否重复计算？
4. 循环里的 `sort`、`reverse`、字符串操作是否计入了？
5. 实际常数大不大？是否需要更快的输入输出？

习惯

写完程序先估复杂度，再测最大数据；不要等 TLE 才回头分析。

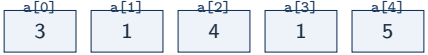
空间复杂度：程序用了多少内存

- 空间复杂度也记作大 O，描述额外内存随 n 的增长。
- 常见单位：`int` 约 4 字节，`long long` 约 8 字节，`char` 1 字节。
- 256 MB $\approx 2.68 \times 10^8$ 字节，约能放 6.7×10^7 个 `int`。
- 总空间 $=$ 全局/静态数组 $+$ 局部数组 $+$ 递归栈 $+$ 容器额外开销。

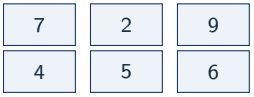
提醒

评测机的内存限制通常按“进程峰值”计算；开多大的数组要心里有数。

数组与二维数组



$5 \times 4 = 20$ 字节

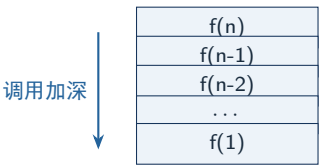


二维数组：每行 3 个，共 4 字节/个

1000×1000 的 `int` 矩阵 $= 4$ MB
5000×5000 的 `int` 矩阵 $= 100$ MB

- 一维数组 10^5 个 `int` 只有 0.4 MB，通常不用担心。
- 二维数组按行连续存放；开 5000×5000 已接近 256 MB 限制的一半。
- `long long` 是 `int` 的两倍，前缀和、答案数组要按需选择。

递归栈也要算



- 每层递归都要保存参数、局部变量和返回地址。
- 递归深度 10^5 时，栈可能达到几 MB，容易栈溢出。
- 递归求斐波那契还有大量重复调用，时间也会爆炸。
- 能用循环递推时，优先用循环，省时间也省栈。

经验值

一般评测机栈空间约 8 MB；递归深度超过 10^5 就要警惕，必要时改递推或手动开栈。

空间换时间：预处理的常见形式

- 前缀和数组：多花 $O(n)$ 空间，把区间查询从 $O(n)$ 降到 $O(1)$。
- 差分数组：多花 $O(n)$ 空间，把区间修改从 $O(n)$ 降到 $O(1)$。
- 标记数组/桶：用值域做下标，$O(1)$ 判断是否出现。
- 记忆化：把算过的答案存起来，避免重复递归。

取舍

先看内存限制，再决定开多大的辅助数组；不要为了省一点空间，让时间复杂度升一个量级。

例 0.2 | 估一估这个程序的内存

```
int a[5000][5000]; // 5000*5000*4 = 100 MB
long long s[1000005]; // 1000005*8 = 8 MB
int depth = 100000; // 递归深度 1e5，栈空间另算
```

- 二维数组 a 已占约 100 MB；若内存限制 128 MB，只剩很少余量。
- 前缀和用 `long long` 是有必要的：10^5 个 10^9 相加会超过 `int`。
- 如果同时开两个 5000×5000 的 `int` 矩阵，就会超过 256 MB。
- 把不必要的大数组改成 `short`、`vector` 按需分配，或滚掉一维。

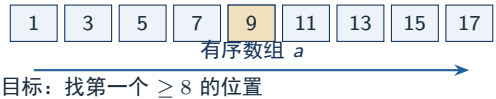
空间自查表

1. 最大的数组有多少个元素？每个元素几字节？
2. 是否开了多个同样大的二维数组？
3. 递归深度多大？每层有多少局部变量？
4. 容器（`vector`、`map`）的额外开销算了吗？
5. 能否用滚动数组、复用数组或按需分配？

单位换算

$1\ \text{MB} = 1024\ \text{KB} = 1024^2$ 字节；估算时直接用“元素个数 $\times$ 每元素字节数”。

二分的前提：单调



- 数组有序，才能用“中间值与目标比较”的结果排除一整半。
- 更一般地，二分答案要求判定函数具有单调性：可行状态和不可行状态分居两侧。
- 无序数组不能直接二分；先排序会改变下标，题目要求原下标时要注意。

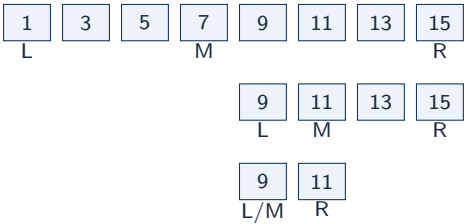
整数二分：先想清楚区间的含义

- 常见区间写法：闭区间 $[l, r]$、左闭右开 $[l, r)$、答案范围 $[1, n+1]$。
- 每次取中点 `mid`，根据判定结果保留一半。
- 关键是：答案一定在保留下来的区间里，并且区间每次严格变小。
- `mid` 用 `l + (r - l) / 2` 计算，避免 `l + r` 溢出。

两种常用模板

闭区间 `while (l <= r)`：适合“找到一个位置就停”。
左闭右开 `while (l < r)`：适合“收敛到唯一答案”。

手推：查找第一个不小于 x 的位置



$x = 8$：$a_4 = 7 < 8$，舍去左半

$a_6 = 11 \geq 8$，答案不晚于 6

$a_5 = 9 \geq 8$，收敛到 5

答案

第一个不小于 8 的位置是 5，值为 9；若所有数都小于 x，答案是 $n + 1$。

模板 A：闭区间 while (l <= r)

```
int l = 1, r = n, ans = n + 1; // ans 先设为“不存在”
while (l <= r) {
    int mid = l + (r - l) / 2;
    if (a[mid] >= x) {
        ans = mid; // 记录可行位置
        r = mid - 1; // 继续往左找更早的
    } else {
        l = mid + 1; // 中点太小，答案在右边
    }
}
// 循环结束后 ans 即第一个 >= x 的位置
```

- 区间始终是闭区间 $[l, r]$，循环条件带等号。
- 找到可行位置后不要立刻退出，因为左边可能还有更早的。

模板 B：左闭右开 while (l < r)

```
int l = 1, r = n + 1; // 答案可能是 n+1
while (l < r) {
    int mid = l + (r - l) / 2;
    if (a[mid] >= x) {
        r = mid; // mid 可行，答案在 [l, mid]
    } else {
        l = mid + 1; // mid 不可行，答案在 [mid+1, r)
    }
}
// 此时 l == r，就是第一个 >= x 的位置
```

- 区间是 $[l, r)$，循环条件不带等号。
- mid 永远小于 r，不会访问 a[n+1]；最后 l 就是答案。

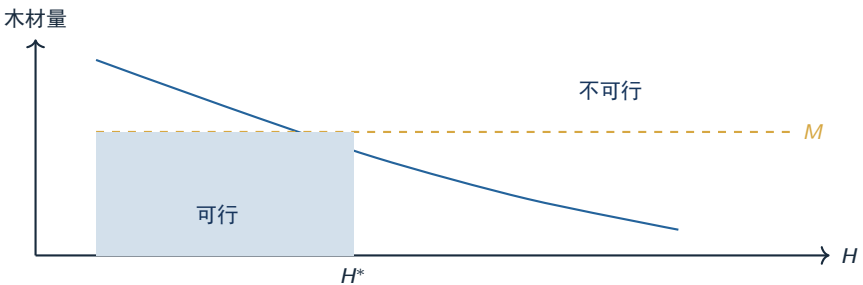
两种写法的边界对照

	闭区间写法	左闭右开写法
初始区间	$[1, n]$	$[1, n+1]$
循环条件	`l <= r`	`l < r`
可行时	`r = mid - 1`	`r = mid`
不可行时	`l = mid + 1`	`l = mid + 1`
结束状态	`l > r`	`l == r`
答案位置	记录的 `ans`	`l`

防死循环

若更新写成 `l = mid`，必须保证 `mid` 取上整，否则 `l` 可能不动。最稳妥的做法是：先写清区间含义，再逐行检查“区间是否真的变小”。

二分答案：把“求最值”变成“判可行”



- 判定函数：给定锯口高度 H，计算能得到的木材量 `wood(H)`。
- H 越小，木材越多，所以 `wood(H)` $\geq$ `M` 在左侧连续成立。
- 要求“最大的可行 H”：二分这个分界点，每次用 $O(n)$ 判定。

例 0.3 | 砍树：题意与样例

题意

有 n 棵树，第 i 棵高 h_i。选择一个整数锯口高度 H，把所有高于 H 的部分锯下，得到木材 $\sum_{h_i>H}(h_i-H)$。求使木材量至少为 M 的最大 H。

- 输入：n 和 M，第二行 n 个高度。
- 样例：$n=4$，$M=7$，高度 $20,15,10,17$。
- 手算：$H=15$ 时得到 $(20-15)+(17-15)=7$，正好够；$H=16$ 时只有 $4+1=5$，不够。
- 所以答案 $H=15$。注意 H 可以取 0，此时全部锯下。

砍树｜判定函数与单调性

- 对固定 H，遍历每棵树：

$$\texttt{wood}(H) = \sum_{i=1}^{n} \max(0, h_i - H).$$

- 若 $H_1 < H_2$，则每棵树的贡献都不增，所以 `wood` 单调不增。
- 答案范围：$0 \le H \le \max h_i$；超过最大高度只能得到 0。
- 判定：若 `wood(mid)` $\ge$ `M`，说明 `mid` 可行，答案在右半；否则在左半。

二分方向

求“最大的可行值”，所以可行时令 `l = mid`，不可行时令 `r = mid - 1`。
此时 `mid` 必须取上整：`mid = (l + r + 1) / 2`。

砍树 | 完整代码（上）

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

int main() {
    int n;
    long long m;
    cin >> n >> m;
    vector<long long> h(n);
    long long max_h = 0;
    for (int i = 0; i < n; ++i) {
        cin >> h[i];
        max_h = max(max_h, h[i]); // 二分上界
    }
```

砍树 | 完整代码（中）

```
long long l = 0, r = max_h;
while (l < r) {
    long long mid = (l + r + 1) / 2; // 上取整
    long long wood = 0;
    for (int i = 0; i < n; ++i) {
        if (h[i] > mid) {
            wood += h[i] - mid; // 高于锯口的部分
        }
    }
```

砍树 | 完整代码（下）

```
        if (wood >= m) {
            l = mid; // mid 可行，尝试更高
        } else {
            r = mid - 1; // mid 不可行，降低锯口
        }
    }
    cout << l << '\n';
    return 0;
}
```

砍树｜复杂度与易错点

- 时间复杂度：二分 $O(\log \max h)$ 次，每次判定 $O(n)$，总计 $O(n \log \max h)$。
- 空间复杂度：存树高 $O(n)$，其余 $O(1)$。
- 易错 1：`mid` 写成下取整，`l = mid` 时可能死循环。
- 易错 2：木材量用 `int`，10^6 棵树、每棵 10^9 会溢出，必须 `long long`。
- 易错 3：上界只取到 $\max h_i$，不要凭感觉写 10^9（虽然后者也能过，但多做了无用二分）。

自查

二分结束后，`l` 一定是可行高度；若 $M = 0$，答案应为 $\max h_i$。

浮点二分：控制误差，而不是下标



- 不比较下标，而是比较函数值；用固定次数或区间长度控制精度。
- 固定 100 轮：区间长度变为原来的 2^{-100}，通常远小于题目要求。
- 或用 `while (r - l > 1e-7);` 注意浮点误差，输出前不要再做多余运算。

```
double l = -1e4, r = 1e4;
for (int it = 0; it < 100; ++it) {
    double mid = (l + r) / 2;
    if (f(mid) < 0) l = mid;
    else r = mid;
}
printf("%.6f\n", l);
```

二分易错清单

1. 数组无序却直接二分；或排序后下标含义变了。
2. `mid` 用 `(l + r) / 2` 导致 `int` 溢出。
3. 该用上取整时用了下取整，`l = mid` 死循环。
4. 可行时更新错方向，把答案排除出区间。
5. 二分答案的判定函数不单调，或边界（0、$n+1$）没考虑。
6. 浮点二分轮数太少，或把 `<` 与 `<=` 混用导致答案偏移。

检查方法

拿最小规模、最大规模、无解、恰好等于阈值四类数据各手推一遍。

多次区间查询：暴力为什么慢

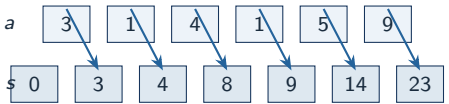


查询 $[2,6]$：逐项相加

每个询问 $O(n)$，q 个询问就是 $O(nq)$

- $n, q = 10^5$ 时，$nq = 10^{10}$，必然超时。
- 区间查询的特点是：数组不变，只是反复问不同的区间。
- 能不能先算好一些“部分和”，让每个询问 $O(1)$ 回答？

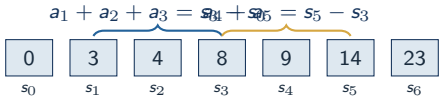
一维前缀和：一次累加，随时查询



$s_i = s_{i-1} + a_i$

- 定义 $s_0 = 0$，$s_i = a_1 + a_2 + \cdots + a_i$。
- 从左到右递推，$O(n)$ 时间预处理，$O(n)$ 空间。
- 之后任意区间和都能用两个前缀相减得到。

区间和：$s_r - s_{l-1}$



公式

$$\sum_{i=l}^{r} a_i = s_r - s_{l-1}.$$

下标从 1 开始，$s_0 = 0$；若从 0 开始存，则区间为 $[l, r]$ 时用 $s_{r+1} - s_l$。

- 预处理 $O(n)$，每个询问 $O(1)$；总复杂度 $O(n + q)$。
- 值可能为负，也可能很大；前缀和数组用 `long long`。

例 0.4 | 区间和：题意与思路

题意

给出长度为 n 的整数序列 a，回答 q 个询问，每个询问给出 l, r，输出 $a_l + a_{l+1} + \cdots + a_r$。

- 数据范围：$1 \le n, q \le 2 \times 10^5$，$|a_i| \le 10^9$。
- 直接对每个询问循环求和是 $O(nq)$，会超时。
- 先预处理前缀和 s，之后每个询问 $O(1)$ 回答。
- 样例：$a = 3, 1, 4, 1, 5$，询问 $[2, 4]$，答案 $1 + 4 + 1 = 6$。

区间和 | 完整代码（上）

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

int main() {
    ios::sync_with_stdio(false);
    cin.tie(nullptr);
    int n, q;
    cin >> n >> q;
    vector<long long> s(n + 1, 0);
    for (int i = 1; i <= n; ++i) {
        long long x;
        cin >> x;
        s[i] = s[i - 1] + x; // 前缀和递推
    }
```

区间和 | 完整代码（下）

```
    while (q--) {
        int l, r;
        cin >> l >> r;
        cout << s[r] - s[l - 1] << '\n'; // 区间和
    }
    return 0;
}
```

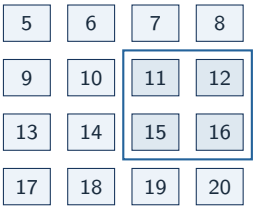
区间和｜复杂度与易错点

- 时间复杂度：预处理 $O(n)$，回答 $O(q)$，总计 $O(n+q)$。
- 空间复杂度：前缀和数组 $O(n)$。
- 易错 1：忘记 $s_0 = 0$，询问 $[1, r]$ 时越界或漏加第一项。
- 易错 2：用 `int` 存前缀和；2×10^5 个 10^9 相加会溢出。
- 易错 3：下标从 0 开始时仍套用 $s_r - s_{l-1}$，导致差一位。

记忆

前缀和数组比原数组长一位，空出 s_0，所有区间公式都统一成 $s_r - s_{l-1}$。

二维前缀和：容斥原理



$s_{i,j}$：左上角 $(1,1)$ 到 (i,j) 的和

$$s_{i,j} = s_{i-1,j} + s_{i,j-1} - s_{i-1,j-1} + a_{i,j}$$

- 把二维前缀和看成“矩形区域的总和”。
- 上方矩形和左方矩形相加，左上角被加了两次，要减掉一次。
- 预处理 $O(nm)$，之后每个子矩阵查询 $O(1)$。

二维区间查询：四个角



(x_1, y_1) 到 (x_2, y_2)

$$= s_{x_2,y_2} - s_{x_1-1,y_2} - s_{x_2,y_1-1} + s_{x_1-1,y_1-1}$$

容斥记忆

大矩形，减去上方，减去左方，再把左上角加回来；四个角各用一次。

- 查询 $O(1)$，适合 $n, m \leq 1000$、$q \leq 2 \times 10^5$ 的静态矩阵。
- 二维前缀和数组同样可能很大，注意内存：1000×1000 的 `long long` 是 8 MB。

什么时候用前缀和

- 数组不变，多次询问区间和、子矩阵和：首选前缀和。
- 若还有区间修改，单用前缀和每次修改要 $O(n)$，应改用差分或树状数组。
- 前缀和也常用于计数：把“区间内满足条件的个数”转成两个前缀相减。
- 与差分配合：差分负责“改”，前缀和负责“还原”和“查询”。

判断口诀

多次查询、不改数组 $\rightarrow$ 前缀和；
多次修改、最后再问 $\rightarrow$ 差分；
又改又问 $\rightarrow$ 树状数组/线段树（后续章节）。

前缀和易错点

1. 数组下标从 0 还是从 1 开始，公式要统一。
2. 前缀和数组长度是 $n+1$，最后一项是 s_n。
3. 负数、大数：用 `long long`，不要用无符号类型。
4. 二维前缀和的四个角：减两个、加一个，顺序不能错。
5. 查询范围越界：$l \geq 1$、$r \leq n$，输入不保证时要先检查。

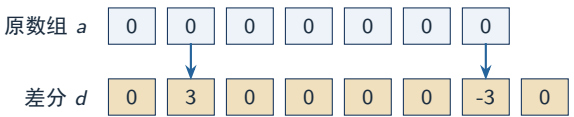
多次区间修改：暴力为什么慢



每次修改扫一遍区间：$O(n)$；m 次修改就是 $O(nm)$

- $n, m = 10^5$ 时，$nm = 10^{10}$，超时。
- 能否只记录“从哪里开始加、到哪里停止加”？
- 差分数组记录相邻元素的差，区间修改只影响两个位置。

一维差分：只改两个位置



$d_l \mathrel{+}= c$
$d_{r+1} \mathrel{-}= c$

还原：$a_i = a_{i-1} + d_i$

- 定义 $d_1 = a_1$，$d_i = a_i - a_{i-1}$（$i \geq 2$）。
- 区间 $[l, r]$ 加 c：d_l 加 c，d_{r+1} 减 c。
- 最后对 d 做一次前缀和，就得到修改后的数组。

例 0.5 | 区间加：题意与手推

题意

长度为 n 的数组初始全为 0。执行 m 次操作，每次把区间 $[l, r]$ 内的数都加上 c。输出全部操作后的数组。

- 数据范围：$1 \le n, m \le 2 \times 10^5$，$|c| \le 10^9$。
- 样例：$n = 5$，操作 $(1, 3, 2)$、$(2, 4, 3)$、$(5, 5, -1)$。
- 差分：$d_1 += 2, d_4 -= 2$；$d_2 += 3, d_5 -= 3$；$d_5 += -1$。
- 还原：$2, 5, 5, 3, -1$。

区间加 | 完整代码（上）

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

int main() {
    int n, m;
    cin >> n >> m;
    vector<long long> diff(n + 2, 0);
    for (int i = 0; i < m; ++i) {
        int l, r;
        long long c;
        cin >> l >> r >> c;
        diff[l] += c; // 区间起点开始加
        diff[r + 1] -= c; // 区间终点之后撤销
    }
```

区间加 | 完整代码（下）

```
    long long cur = 0;
    for (int i = 1; i <= n; ++i) {
        cur += diff[i]; // 差分前缀和即当前值
        cout << cur;
        if (i == n) cout << '\n';
        else cout << ' ';
    }
    return 0;
}
```

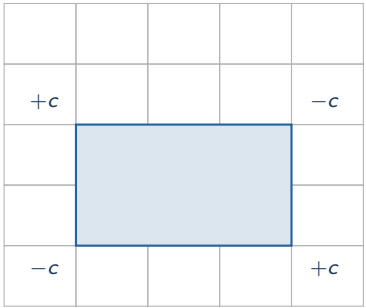
区间加｜复杂度与易错点

- 时间复杂度：每次修改 $O(1)$，还原 $O(n)$，总计 $O(n+m)$。
- 空间复杂度：差分数组 $O(n)$。
- 易错 1：`diff` 要开到 $n+2$，因为要访问 $r+1$。
- 易错 2：c 可能为负，累加值用 `long long`。
- 易错 3：先做完全部修改再还原；不要每改一次就还原一遍。

初始值不为零时

先把原数组的差分建好：$d_1 = a_1$，$d_i = a_i - a_{i-1}$，再叠加修改。

二维差分：四个角



(x_1, y_1) 加 c
$(x_2 + 1, y_1)$ 减 c
$(x_1, y_2 + 1)$ 减 c
$(x_2 + 1, y_2 + 1)$ 加 c

- 二维差分记录“矩形修改”的四个角，最后做二维前缀和还原。
- 预处理/修改 $O(1)$，还原 $O(nm)$；适合多次子矩阵加、最后统一输出。

二维差分还原：二维前缀和

```
// 对差分数组 d 做二维前缀和，得到修改后的矩阵
for (int i = 1; i <= n; ++i) {
    for (int j = 1; j <= m; ++j) {
        d[i][j] += d[i - 1][j] + d[i][j - 1]
                 - d[i - 1][j - 1];
    }
}
// 此时 d[i][j] 就是 (i,j) 位置最终的值
```

- 修改时四个角：$(x_1, y_1) + c$、$(x_2 + 1, y_1) - c$、$(x_1, y_2 + 1) - c$、$(x_2 + 1, y_2 + 1) + c$。
- 数组要开到 $(n + 1) \times (m + 1)$，防止越界。
- 若还要多次子矩阵查询，再对还原后的矩阵做一次二维前缀和。

差分与前缀和互为逆运算

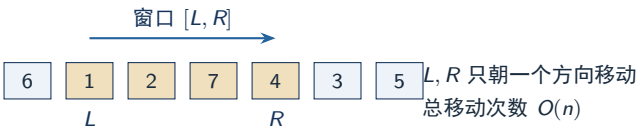


- 差分适合“多次区间修改、最后统一还原”。
- 前缀和适合“数组不变、多次区间查询”。
- 两者都基于“相邻元素的关系”，是同一对逆运算。

差分易错点

1. 差分数组开小：区间右端为 n 时要访问 d_{n+1}。
2. 忘记做前缀和还原，直接输出差分数组。
3. 二维差分的四个角符号写反。
4. 多次修改后重复还原，导致结果叠加错误。
5. 初始数组非零时，没有先把初始差分建好。
6. 负数和 10^9 级别的累加仍用 `int`。

双指针：用单调性省掉重复枚举



- 暴力枚举所有区间是 $O(n^2)$；双指针让 L, R 各自最多走 n 步。
- 前提：当 R 向右移动时，窗口的某种性质单调变化。
- 常见三种：同向窗口、相向两端、快慢指针。

同向双指针：窗口只进不退



- R 负责“扩大窗口”，L 负责“缩小窗口”；两者都不回退。
- 每个元素最多进窗口一次、出窗口一次，总操作 $O(n)$。

相向双指针：两端向中间



- 只适用于有序数组；有序才能确定“移动哪一端”。
- 每次移动至少排除一个端点，L, R 相遇时结束，总操作 $O(n)$。
- 典型题：两数之和、判断回文、三数之和（需先排序）。

滑动窗口的适用条件

- 数组元素非负时，“窗口越长，和越大”成立，L 才能单调右移。
- 若含负数，扩大窗口不一定让和变大，窗口收缩条件失效，不能直接用。
- 要求“最短/最长满足某性质的区间”时，先验证这个性质是否随窗口扩大而单调。
- 计数类问题：固定右端点，统计以它为右端点的合法左端点数量，也常用双指针。

反例

数组 $1, -5, 3$，目标和为 3：右端扩到第三个数时和从 -1 变 3，但中间曾小于 0，简单滑动窗口会漏解或错判。

例 0.6 | 最短区间：题意与手推

题意

给定长度为 n 的正整数数组和整数 K，求元素和至少为 K 的连续区间的最短长度；不存在输出 -1。

- 数据范围：$1 \le n \le 2 \times 10^5$，$1 \le a_i \le 10^9$，$1 \le K \le 10^{18}$。
- 样例：$n = 6$，$K = 9$，数组 $2, 1, 3, 2, 4, 1$。
- 手推：$[3, 5]$ 的和 $3 + 2 + 4 = 9$，长度 3；$[4, 6]$ 的和 $2 + 4 + 1 = 7$，不够。
- 答案 3；若所有数之和都小于 K，输出 -1。

最短区间｜完整代码（上）

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

int main() {
    ios::sync_with_stdio(false);
    cin.tie(nullptr);
    int n;
    long long k;
    cin >> n >> k;
    vector<long long> a(n);
    for (int i = 0; i < n; ++i) cin >> a[i];
    int left = 0, best = n + 1;
    long long sum = 0;
```

最短区间｜完整代码（下）

```
    for (int right = 0; right < n; ++right) {
        sum += a[right]; // 扩大窗口
        while (sum >= k) { // 窗口已满足
            best = min(best, right - left + 1);
            sum -= a[left]; // 尝试缩小
            ++left;
        }
    }
    if (best == n + 1) cout << -1 << '\n';
    else cout << best << '\n';
    return 0;
}
```

最短区间｜复杂度与易错点

- 时间复杂度：`right` 走 n 步，`left` 总共走不超过 n 步，$O(n)$。
- 空间复杂度：存数组 $O(n)$。
- 易错 1：`sum` 用 `int`；2×10^5 个 10^9 相加会溢出。
- 易错 2：把 `while` 写成 `if`，一个右端点可能对应多次收缩。
- 易错 3：先更新答案再减 `a[left]`，顺序不能反。
- 易错 4：无解时输出 $n+1$，要单独判断成 -1。

窗口模板

右端进、左端出；进一次、出一次，总代价 $O(n)$。写完后用“全正数但无解”“单元素”“正好等于 K”三类数据检查。

双指针易错清单

1. 数组含负数却套用“和越大窗口越长”的滑动窗口。
2. 左指针回退，导致复杂度退化成 $O(n^2)$。
3. 窗口收缩用 `if` 而不是 `while`。
4. 答案初值取 0，与“无解”混淆。
5. 相向双指针用在无序数组上。
6. 移动指针后忘记更新窗口内的和维护量。

练习安排：按知识点分组

1. 复杂度：先做“入门/Hello,World!”热身，再回到本节自查表估算手写程序。
2. 二分：先做“查找第一个不小于”，再练“砍树”，最后做“方程的根”。
3. 前缀和：先做“区间和”，再做“子矩阵求和”，挑战“和为 K 的倍数”。
4. 差分：先做“区间加”，再做“子矩阵加”，最后做“搭建积木”。
5. 双指针：先做“最短区间”，再做“两数之和”，最后做“最长无重复子串”。

做题顺序

每题先写朴素做法并估复杂度，再改成目标算法；最后用边界数据自测。

题目索引 | 1–6 / 17

1. Hello,World!（入门热身，P4987）
2. 能买到饮料的商店数（练习，P4986）
3. 查找第一个不小于（练习，P4984）
4. 砍树（例题，P4985）
5. 方程的根（练习，P4983）
6. 区间和（例题，P4989）

标“题包”的题目没有学校 OJ 题号，使用本课程题包评测。

题目索引 | 7–12 / 17

7. 两种区间求和（练习，P4988）
8. 子矩阵求和（练习，P4991）
9. 和为 K 的倍数（练习，P4990）
10. 区间加（例题，P4996）
11. 推荐温度统计（练习，P4998）
12. 子矩阵加（练习，P4997）

前缀和与差分经常一起出现：先差分修改，再前缀和还原。

题目索引 | 13–17 / 17

13. 搭建积木（练习，P4999）
14. 最短区间（例题，P4994）
15. 和至少为 K 的连续区间（练习，P4993）
16. 两数之和（练习，P4992）
17. 最长无重复子串（练习，P4995）

以上 17 道题均有完整题包：题面、标程、数据与 PDF。

结课自查

1. 看到 $n \le 10^5$，能立刻排除 $O(n^2)$ 吗？
2. 能写出两种整数二分模板，并说清各自的区间含义吗？
3. 二分答案时，判定函数为什么必须单调？
4. 前缀和、差分分别适合“多次查询”还是“多次修改”？
5. 滑动窗口为什么要求元素非负（或性质单调）？
6. 每道题都测过最小、最大、无解、恰好边界四类数据吗？

下一步

带着这四件工具进入后续章节；遇到“又修改又查询”的问题，再学树状数组与线段树。