并查集与 ST 表

近乎 $O(1)$ 集合合并判环 + 静态 RMQ 区间最值秒答

HGNU-ACM 实验室算法组

配套：路径压缩与按大小合并 + 带权并查集 + ST 表倍增覆盖 + 3 道真题精讲





本章目录

并查集（DSU）与路径压缩

按大小合并、sz[] 与带权并查集

ST 表原理：静态 RMQ 与重叠覆盖

第 1–3 节：承接第 3 章树形父子结构，掌握并查集路径压缩/按大小合并/带权扩展，以及 ST 表解决静态 RMQ 的幂等原理。

本章目录（续）

ST 表 $O(n \log n)$ 预处理与 $O(1)$ 查询

三大并查集与 ST 表真题精讲

常见错误、检查清单与练习安排

第 4–6 节：彻底掌握 ST 表 $O(n \log n)$ 倍增填表与 $O(1)$ 查询、3 道真题逐行手推与避坑收尾。

承接第 3 章：只记“谁是我上级”，不记下属

- 第 3 章我们用 `lch/rch` 从树根自顶向下找孩子；现在把方向倒过来！
- 现实问题（认亲 / 朋友圈）：已知 $n = 2 \times 10^4$ 个人和 $m = 10^6$ 条“x 和 y 是亲戚”关系，再问 $q = 10^6$ 次“a 和 b 是不是亲戚”。
- 我们根本不关心“x 有哪些儿子”，只关心“x 所在家族的最高老祖宗是不是和 y 一样”！

写程序前先回答三个问题

1. 每个人的直接上级用什么数组记录？怎样表示自己就是老祖宗？
2. 如果家族排成一条几千人的单链，怎么快速找老祖宗不超时？
3. 如何把两个原本不相干的家族合并成一个大集合？

本页要点

用一维数组 `fa[x]` 记每个点只认一位“直接上级”，就能用一片树林管理全部集合！

暴力合并与暴力搜图为什么会 TLE？

当 $n = 2 \times 10^4, m = 10^6, q = 10^6$ 时两种暴力方案都会超时

- 暴力一（组号染色数组）：用 `belong[i]` 记每个人所在组号，查询只需 $O(1)$；但每次合并两组时要全表扫描把旧组号全改成新组号，单次 $O(n)$，m 次总操作 $O(nm) \approx 2 \times 10^{10}$ 次，必 **TLE**！
- 暴力二（建邻接表 **BFS**）：建图只要 $O(1)$ 加边，但每次问“a 和 b 是否连通”都要 BFS 跑一遍连通块，总操作 $O(q \cdot n) \approx 2 \times 10^{10}$ 次，同样 **TLE**！
- 并查集：用树形父亲指针 + 两条优化，把合并和查询同时压到近乎 $O(1)$！

本页要点

并查集把“改组号”变成“改树根指针”，一次合并只改一个位置，彻底躲开 $O(n)$ 扫表！

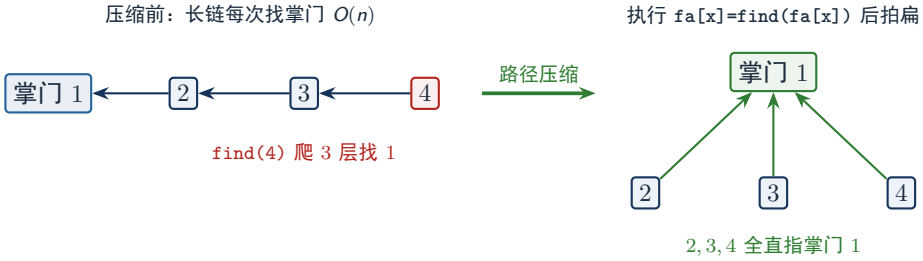
新术语大白话：代表元、find、unite 与路径压缩

- 代表元（总掌门）：每个集合挑出唯一一个根节点代表整个集合；满足 `fa[x] == x`（自己指向自己）的节点就是总掌门。
- 查询 **find(x)**：顺着 `fa[x]` 一直往上找，直到撞见 `fa[x] == x` 的总掌门。判断两人是否同组，就看 `find(x) == find(y)`！
- 合并 **unite(x, y)**：分别找出两人的总掌门 `rx`，`ry`，若不是同一人，就让 `fa[ry] = rx`（让其中一位掌门向另一位低头认老大）。
- 路径压缩：既然我们只关心“总掌门是谁”，在 `find(x)` 回溯时顺手写 `fa[x] = find(fa[x])`，把沿途经过的所有点直接挂到总掌门下面！

本页要点

路径压缩后，原本走 1000 步的长链被一刀拍扁成深度 1 的“大扇子”，下次查询 1 步直达！

图解：退化长链 vs 路径压缩“一步直连掌门”



本页要点

压缩过程是顺路完成的：只要调用过一次 `find(4)`，沿途的 $4, 3, 2$ 的 `fa` 就全部直接指向 1！

手算跟踪表：5 个点合并与路径压缩的 fa[] 变化

初始 5 个人各自为掌门 `fa = [1, 2, 3, 4, 5]`（下标 $1 \sim 5$）：

操作步骤	寻找过程与判定	改动的 fa[]	操作后 fa[1..5]
初始	每人指向自己 `fa[i] = i`	—	[1, 2, 3, 4, 5]
`unite(1, 2)`	find(1)=1, find(2)=2, 不同根	`fa[2] = 1`	[1, 1, 3, 4, 5]
`unite(3, 4)`	find(3)=3, find(4)=4, 不同根	`fa[4] = 3`	[1, 1, 3, 3, 5]
`unite(2, 3)`	find(2)=1, find(3)=3, 不同根	`fa[3] = 1`	[1, 1, 1, 3, 5]
`find(4)` 递归	4->3->1，命中 fa[1]==1 返回 1	归程改 `fa[4]=1`	[1, 1, 1, 1, 5]
终止校验	`find(4)==find(2)`（均为 1）	同掌门，返回真	$1, 2, 3, 4$ 同组，5 独立

本页要点

注意第 4 步：合并 (2, 3) 后 `fa[4]` 还暂时指着 3，直到第 5 步调 `find(4)` 才把它压直连到 1！

为什么路径压缩不会弄乱集合归属？（代表元不变量）

阶段	要验证的内容
初始化	令 `fa[i] = i`，每个元素独自成派且自己是根，各自集合互不相交
保持（unite）	只把一个根 `ry` 挂到另一个根 `rx` 下面，两棵树所有节点的“总根”统一成 `rx`
保持（压缩）	`find(x)` 沿着边往上一定到达当前唯一总根 `r`；把 `fa[x]` 改成 `r` 只缩短步数，并未离开 `r` 这棵树
终止	任意时刻，两个点在同一集合当且仅当它们沿着 `fa` 找到的最终根相等

本页要点

路径压缩只改变树的“内部连线形状”，绝不改变“谁和谁在同一棵树里”！

模板代码：路径压缩 find 与基础 unite

```
int fa[MAXN];

void init(int n) {
    for (int i = 1; i <= n; ++i) fa[i] = i;          // 初值每人都是自己的掌门
}
int find(int x) {
    if (fa[x] == x) return x;                        // 找到总掌门，递归终止！
    return fa[x] = find(fa[x]);                      // 回溯时把路径压缩直连掌门
}
void unite(int x, int y) {
    int rx = find(x), ry = find(y);                  // 先各自找到最高总掌门
    if (rx != ry) fa[ry] = rx;                       // 合并掌门（不是写 fa[y] = x）
}
```

本页要点

千万不要在 unite 里直接写 fa[y] = x！必须先用 find 找各自的根 rx，ry 再连根！

上机验证与反事实追问：路径压缩基础测试

3 组本地验证数据与预期输出

- 传递认亲：n=4，合并 (1,2)，(2,3) 后查 (1,3) 和 (1,4) ⇒ 输出 Yes 与 No。
- 自连查询：n=1，未合并直接查 find(1) == find(1) ⇒ 输出 Yes。
- 长链压扁：n=5，依次合并 (4,5)，(3,4)，(2,3)，(1,2)，再查 find(5) ⇒ 查完后 fa[5],fa[4],fa[3],fa[2] 全等于 1。

反事实追问

如果判断两人是否同组时，懒得写 find(x) == find(y) 而直接写了 fa[x] == fa[y]，在上面的“跟踪表第 4 步刚合并完 unite(2, 3) 时”查 (2, 4) 会怎样？此时 fa[2] == 1 而 fa[4] == 3，明明已经是亲戚，却会错误输出 No！

本页要点

查询是否连通永远写 find(x) == find(y)，绝不能偷懒写 fa[x] == fa[y]！

并查集基础高频易错对照（WA / RE / TLE）

错误表现与后果	错误写法	正确写法与原因
忘记初始化 fa（**WA**）	开完全局数组直接用，全为 0	所有人一开始就全连到 0 号点，务必先写 fa[i] = i
不调 find 直接比（**WA**）	写成 if (fa[x] == fa[y])	中间点可能还没被路径压缩过，必须比 find(x) == find(y)
合并没找根直接连（**WA**）	在 unite 中写 fa[y] = x	会把 y 从原家族硬扯出来丢掉旧祖宗，必须 fa[ry] = rx
下标越界（**RE**）	题目 1-based 却只开 vector<int> fa(n)	1-based 必须开大小为 n + 1

本页要点

记牢三件事：开 n + 1 初始化 fa[i]=i、合必找根 rx/ry、问必调 find！

为什么光有路径压缩还不够？（合并方向的讲究）

- 假设我们把一棵有 10000 个点的树 A，直接挂到只有 1 个点的树 B 下面（`fa[rootA] = rootB`），会让这 10000 个点的深度全都立刻 $+1$！
- 相反，如果反过来把只有 1 个点的小树 B 挂到大树 A 下面（`fa[rootB] = rootA`），那 10000 个点的深度纹丝不动！
- 按大小合并（启发式合并）：用数组 `sz[r]` 记录以 `r` 为掌门的帮派共有多少人，合并时永远让人数少的小树挂到人数多的大树下面！
- 复杂度：路径压缩 $+$ 按大小合并配合使用，单次操作均摊仅 $O(\alpha(n))$（阿克曼反函数，对宇宙内所有整数 $\alpha(n) \le 4$，视作 $O(1)$）！

本页要点

“小树挂大树”既让树高永远不超过 $\log_2 n$，还顺便白送了实时统计每个连通块人数 `sz[]` 的功能！

新术语大白话：按大小合并、按秩合并与带权并查集

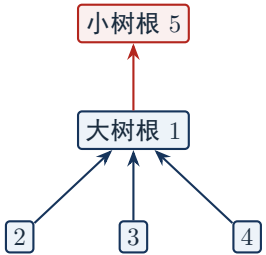
- 按大小合并 **sz[x]**：维护每个连通块节点个数 `sz[root]`，合并时小块并入大块，`sz[big] += sz[small]`。
- 按秩合并 **rk[x]**：用 `rk[root]` 记录树高的上界（秩），矮树挂高树下，只有两棵树一样高时新树高才 $+1$。
- 带权并查集 **d[x]**：不仅记“谁是我上级”，还顺便用 `d[x]` 记“我和直接上级之间相差多少”（比如体重差、食物链敌友关系、距离）。

本页要点

题目只问是否连通时用 `sz` 或 `rk` 都可以；若还要回答“这组一共有几个人”，直接用 `sz[find(x)]`！

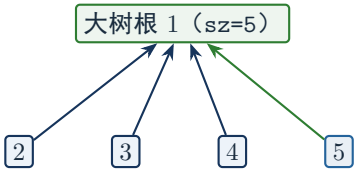
图解：大树挂小树（树高 $+1$）vs 小树挂大树（树高不变）

反面：大树（4 点）挂到小树（1 点）



全树高度从 2 变成 3！

正确：小树（1 点）挂到大树（4 点）



树高仍为 2，纹丝不动！

本页要点

为什么树高最多 $\log_2 n$？因为每次只有被挂上去的那棵小树深度会 $+1$，而此时它所在新树规模至少翻倍！

手算跟踪表：sz[] 维护连通块大小与同根跳过

初始 5 个点：fa = [1,2,3,4,5]，sz = [1,1,1,1,1]：

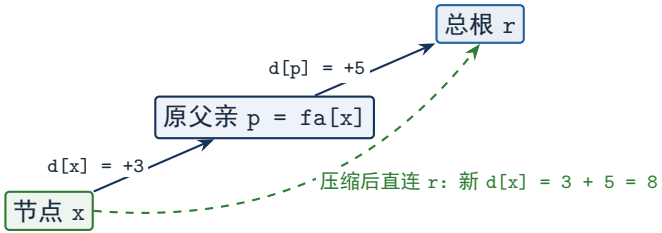
合并操作	找到根与各自 sz	谁挂谁	fa[1..5]	sz[1..5]
unite(1,2)	1(sz=1)，2(sz=1)	2 挂 1	[1, 1, 3, 4, 5]	[2, 1, 1, 1, 1]
unite(3,4)	3(sz=1)，4(sz=1)	4 挂 3	[1, 1, 3, 3, 5]	[2, 1, 2, 1, 1]
unite(4,5)	3(sz=2)，5(sz=1)	小 5 挂大 3	[1, 1, 3, 3, 3]	[2, 1, 3, 1, 1]
unite(1,3)	1(sz=2)，3(sz=3)	小 1 挂大 3	[3, 1, 3, 3, 3]	[2, 1, 5, 1, 1]
unite(2,5)	根均为 3（rx == ry）	同根直接跳过	[3, 1, 3, 3, 3]	[2, 1, 5, 1, 1]

本页要点

看第 5 步：2 和 5 早已同根，务必先判 if（rx == ry）return false;；否则 sz[3] 会被自己加自己错误翻倍！

带权并查集图解：路径压缩时如何同步累加权值？

设 d[x] 表示“x 到它直接父亲 fa[x] 的偏移量”（例如分数差 $score[x] - score[fa[x]]$）：



- 压缩前 x 到 p 差 3，p 到根 r 差 5；压缩后 x 直接认 r 当父亲，新偏移量就是两段相加 $3+5=8$！

本页要点

记住顺序：必须先把原父亲 p = fa[x] 的 find(p) 算完（让 d[p] 变成到根的距离），再令 d[x] += d[p]！

带权并查集为什么更新顺序不能反？（正确性论证）

把下面两段代码对比看一眼就永远不会写反：

- 正确写法（先保存旧父亲并递归，再累加旧父亲的权）：
  1. 先记 `int p = fa[x];`
  2. 递归压缩父亲：`fa[x] = find(p);`（此时 `p` 已被压缩直连总根，`d[p]` 正好是 `p` 到总根的距离！）
  3. 最后更新 `d[x] += d[p];` 并返回 `fa[x]`。
- 致命错法：先写了 `fa[x] = find(fa[x]);` 再写 `d[x] += d[fa[x]];`！这时 `fa[x]` 已经被改成了总根 `r`，而总根的 `d[r] == 0`，直接把中间父亲 `p` 的权值丢光！

本页要点

先递归让“旧父亲”拿到到总根的权值，再把旧父亲的权加到自己身上，最后才改自己指向总根！

模板代码（1/2）：标准 DSU（含 sz[] 按大小合并）

```cpp
struct DSU {
    vector<int> p, sz;
    explicit DSU(int n) : p(n), sz(n, 1) { iota(p.begin(), p.end(), 0); }
    int find(int x) { return p[x] == x ? x : (p[x] = find(p[x])); }
    bool unite(int x, int y) {
        x = find(x); y = find(y);
        if (x == y) return false;                    // 同组跳过，防止 sz 重复翻倍！
        if (sz[x] < sz[y]) swap(x, y);               // 小树挂到大树下面
        p[y] = x; sz[x] += sz[y]; return true;
    }
};
```

本页要点

需要查询 x 所在连通块人数时，只需调用 `dsu.sz[dsu.find(x)]`！

模板代码（2/2）：带权并查集 find 与 unite 核心

```
int find_w(int x) {
    if (fa[x] == x) return x;
    int p = fa[x];                                  // 先保存旧父亲 p
    fa[x] = find_w(p);                              // 递归把 p 压缩到总根并更新 d[p]
    d[x] += d[p];                                   // 再累加旧父亲到总根的偏移量
    return fa[x];
}
bool unite_w(int x, int y, long long w) {           // 已知 val[x] - val[y] = w
    int rx = find_w(x), ry = find_w(y);
    if (rx == ry) return (d[x] - d[y] == w);        // 已同组，检查新条件是否矛盾！
    fa[rx] = ry; d[rx] = d[y] + w - d[x];           // 画矢量环：d[x]+d[rx] == d[y]+w
    return true;
}
```

本页要点

合并推导：d[x] + d[rx] = d[y] + w，移项立刻得 d[rx] = d[y] + w - d[x]！

上机验证与反事实追问：sz[] 与带权并查集

3 组本地测试数据与预期输出

- 连通块人数：n=5，合并 (1,2),(3,4),(1,3) 后查 sz[find(2)] 和 sz[find(5)] ⇒ 4 与 1。
- 重复合并同组：接上再执行 unite(2,4) ⇒ 返回 false，且 sz[find(2)] 保持 4 不变。
- 带权矛盾判定：已知 val[1]-val[2]=3、val[2]-val[3]=2，再传入 val[1]-val[3]=4 ⇒ 发现 d[1]-d[3]=5 != 4，判定矛盾！

反事实追问

若 DSU::unite 漏写 if (x == y) return false;，再执行 unite(2, 4) 会将 p[r] = r 且执行 sz[r] += sz[r] 使整组人数翻倍，全部 sz 统计崩盘（**WA**）！

本页要点

查人数必须在根上读：dsu.sz[dsu.find(x)]，不能读非根节点过时的 sz[x]！

并查集扩展易错与适用边界

场景 / 易错点	错误做法	正确做法与适用边界
读取非根节点 `sz[x]`（**WA**）	直接打印 `dsu.sz[x]`	小树并入大树后只更新根，必须写 `dsu.sz[dsu.find(x)]`
同组重复相加（**WA**）	合并前没判 `if (x == y) return false;`	同根直接返回 `false`，既防 `sz` 翻倍又能顺便判环
带权先改 `fa[x]` 后加（**WA**）	先写 `fa[x]=find(fa[x])` 再加 `d[fa[x]]`	先存旧父亲 `p = fa[x]`，递归完再加 `d[p]`
需要“删边/拆集合”	强行在并查集里把 `fa[x]` 改回 `x`	标准并查集不支持直接拆散！遇到删边题常改为离线倒序加边

本页要点

并查集三大适用场景：动态合并判连通、连边前判是否成环、统计连通块大小与带权关系。

生活引入：站好队不动，连问 100 万次“这段谁最高”

体育课上 $n = 10^5$ 名同学按学号排成一列，每个人身高固定下来不再变化（静态序列）。老师要连问 $q = 10^6$ 次：“从第 l 位到第 r 位同学里，最高的是多高？”

- 这类问题在算法竞赛里有个专门英文缩写：**RMQ**（Range Minimum/Maximum Query，区间最值查询）。

写程序前先回答三个问题

1. 如果每次从 l 扫到 r 找最大值，100 万次询问要算多少步？
2. 第 2 篇学过的“前缀和”能 $O(1)$ 求区间和，为什么求不了区间最大值？
3. 能不能提前算好一些“特定长度”的段，查询时用两段拼出任意 $[l, r]$？

本页要点

当询问高达 10^6 次且序列不修改时，每次回答必须是 $O(1)$ 才能稳稳不超时！

暴力扫描为何 TLE？前缀和为何失灵？

- 暴力循环扫描（**TLE**）：每次询问跑一遍 `for (int i = l; i <= r; ++i)`，最坏要走 $n = 10^5$ 步。乘以 $q = 10^6$ 次询问，总计算量高达 10^{11} 次（需要几百秒），必 **TLE**！
- 前缀和为什么救不了最大值？：区间和满足“可减性”——$sum(l, r) = pre[r] - pre[l-1]$；但最大值没有逆运算！
  - 比如前 5 个人最高 180，前 2 个人最高也是 180；光看这两个数，你根本猜不出第 $3 \sim 5$ 个人最高是 180 还是 165！
- 出路：把所有“长度为 $2^0, 2^1, 2^2, \ldots$ 的区间最值”提前算出来存成稀疏表（**Sparse Table**，简称 **ST** 表）！

本页要点

因为 max 不能做减法，我们改用“两段拼凑”；而且这两段就算中间重叠了一块也不怕！

核心难点：为什么 ST 表不能算区间和，只能算 max/min/gcd？

假设我们要算区间 $[1,5]$ 的结果，手头只预先存了 $[1,4]$ 和 $[2,5]$ 两段（中间 $[2,4]$ 被重复算了两次）：

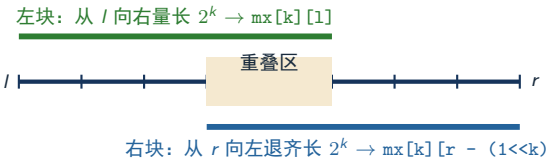
运算类型	重复算一次会变吗？	用 $[1,4]$ 与 $[2,5]$ 合并对吗？	能不能用 ST 表？
最大值 max	$\max(x,x)=x$（不变）	重复部分比较两次还是最大	可以！
最小值 min	$\min(x,x)=x$（不变）	重复部分比较两次还是最小	可以！
最大公约数 gcd	$\gcd(x,x)=x$（不变）	重复部分取两次公约数不变	可以！
区间和 $+$	$x+x=2x$（翻倍了）	中间 $[2,4]$ 被加了两次，全错	绝对不行！（用前缀和）

- 新术语大白话——可重复贡献（幂等律）：一个数跟它自己做运算结果还是它自己（$f(x,x)=x$），那两段区间重叠着算就完全不影响结果！

本页要点

记住 ST 表铁律：只做静态、且满足“重复算不算错”的运算（max, min, gcd、按位与/或）；求和用前缀和，带修改换第 5 章线段树！

核心图解：任意区间 $[l, r]$ 如何用两块长 2^k 的块盖住？



- 设查询长度 $len = r - l + 1$，挑出不超过 len 的最大 2 的幂次 2^k（即 $k = \lfloor \log_2 len \rfloor$）。
- 左块覆盖 $[l,\ l + 2^k - 1]$，右块覆盖 $[r - 2^k + 1,\ r]$，两块合起来刚好拼成 $[l, r]$！

本页要点

右块起点必须写 `r - (1 << k) + 1`！比如查 $[2, 5]$ 取 $k = 2$（长 4），右块起点是 $5 - 4 + 1 = 2$！

为什么两块长度 2^k 一定盖严 $[l, r]$ 不会漏掉中间？

设区间长度 $len = r - l + 1$，取 $k = \lfloor \log_2 len \rfloor$，这是不超过 len 的最大 2 的幂，因此必定满足：

$$2^k \le len < 2^{k+1} = 2 \cdot 2^k$$

- 两块长度之和为 $2^k + 2^k = 2 \cdot 2^k > len$！
- 左块顶着最左端 l、右块顶着最右端 r，两块总长度严格大于整个区间长度 len，中间要么正好严丝合缝相接，要么有一段重叠，绝对不可能出现任何空隙漏掉元素！
- 当 len 本身正好就是 2 的幂（如 $len = 4, k = 2$）时，左块和右块完全重合为同一块 $[l, r]$，同样正确！

本页要点

这就是 ST 表能 $O(1)$ 查询的数学底气：每次只查表两次取一次 max / min，完全不用循环！

手算查询跟踪表：在 a = [1，7，3，4，2，5] 上拆解 4 次最大值查询

数组 $n = 6$（1-based 下标 $1 \sim 6$），观察不同长度区间的拆解结果：

查询 [l，r]	长度	k=floor(log2 len)	左块 [l，l+2^k-1]	右块 [r-2^k+1，r]	结果
[1，5]	5	$k = 2$ $(2^k = 4)$	[1，4]（最大 7）	[2，5]（最大 7）	$\max(7,7) = 7$
[4，6]	3	$k = 1$ $(2^k = 2)$	[4，5]（最大 4）	[5，6]（最大 5）	$\max(4,5) = 5$
[2，2]	1	$k = 0$ $(2^k = 1)$	[2，2]（最大 7）	[2，2]（最大 7）	$\max(7,7) = 7$
[1，6]	6	$k = 2$ $(2^k = 4)$	[1，4]（最大 7）	[3，6]（最大 5）	$\max(7,5) = 7$

本页要点

注意单点查询 [2，2]：长度 1 算出 $k = 0, 2^0 = 1$，左右两块都是 [2，2]，无需写任何特判！

横向对比：前缀和 vs ST 表 vs 第 5 章线段树

什么时候该用 ST 表，什么时候该换别的算法？

数据结构	预处理 / 建树	单次查询	适用条件与下一章预告
前缀和（第 2 篇）	$O(n)$	$O(1)$	静态区间和、异或和（需满足可减性）
ST 稀疏表（本章）	$O(n \log n)$	$O(1)$	静态区间 max / min / gcd（可重复贡献），不支持修改
树状数组/线段树（第 5 章）	$O(n)$	$O(\log n)$	既要查区间又要随时修改数组元素（动态维护）

本页要点

遇到静态最值且询问上百万次，选 ST 表最稳最快；只要题目出现“把第 x 个数改成 v”，立刻换第 5 章数据结构！

倍增状态 f[j][i]（或 mx[j][i]）的精确含义

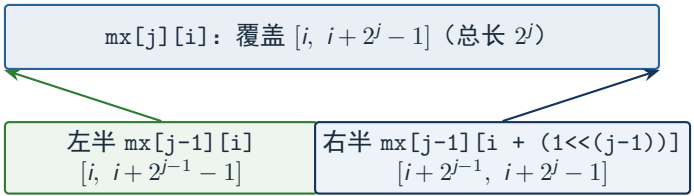
为了覆盖所有长度 2^j 的区间，我们把状态定义得清清楚楚：

- 状态定义：mx[j][i] 表示从位置 i 开始、向右连续 2^j 个元素 $[i,\ i+2^j-1]$ 的最大值。
- 第 **0** 层初值（$j=0$）：长度 $2^0=1$，区间 $[i,i]$ 只有自己，所以 mx[0][i] = a[i]！
- 为什么建议把幂次 j 放第一维？：内层循环 i 连续递增时，访问 mx[j][i] 在内存里是连续相邻的，缓存命中更快，而且开 $K\approx 20$ 行就能直观代表倍增的“层数”。
- 最大幂次 K 开多大？：只要 $2^{K-1}\le n$ 即可，例如 $n\le 10^5$ 时 $\lfloor \log_2 10^5 \rfloor = 16$，所以设 K = __lg(n) + 1（或定长开 20 层）绝对够用！

本页要点

把每个状态理解成“起点 i + 步长 2^j”，所有下标加减都靠“起点 + 一半步长”推出来！

图解：长 2^j 的区间如何由两个长 2^{j-1} 拼出来



- 状态转移方程：`mx[j][i] = max(mx[j - 1][i], mx[j - 1][i + (1 << (j - 1))])`。
- 右端点不能越界：区间最后一个下标 $i+2^j-1 \le n$，因此内层循环条件是 `i + (1 << j) - 1 <= n`！

本页要点

左半长度 2^{j-1}、右半紧接着从 $i+2^{j-1}$ 开始也占 2^{j-1}，两半合起来不重不漏正好 2^j！

手算预处理跟踪表：对 [3，2，4，5，6，8] 填完整张 mx 表

序列长度 $n = 6$（下标 $1 \sim 6$），依次计算第 $j = 0, 1, 2$ 层（$2^3 = 8 > 6$ 停止）：

j（长 2^j）	$i=1$	$i=2$	$i=3$	$i=4$	$i=5$	$i=6$	停止边界
$j=0$（1）	3	2	4	5	6	8	到 $i=6$ 止
$j=1$（2）	$\max(3,2)=3$	$\max(2,4)=4$	$\max(4,5)=5$	$\max(5,6)=6$	$\max(6,8)=8$	越界	$6+2-1>6$
$j=2$（4）	$\max(3,5)=5$	$\max(4,6)=6$	$\max(5,8)=8$	越界	越界	越界	$4+4-1>6$

- 验证 `mx[2][2]`（即 $[2,5]$ 最大值）：由上一层的 `mx[1][2] = 4` 和 `mx[1][4] = 6` 合并得到 $\max(4,6) = 6$！

本页要点

当 `i + (1 << j) - 1 > n` 时说明这段已经伸出数组末尾，立即停止当前层循环！

为什么外层循环必须枚举 j，内层枚举 i？（依赖论证）

- 看转移公式：`mx[j][i]` 依赖第 $j-1$ 层的两个格子——正上方的 `mx[j-1][i]` 和右下方的 `mx[j-1][i + 2^(j-1)]`。
- 外层枚举 `j`（正确）：算第 j 层任何格子时，第 $j-1$ 层的全部格子早已在上一轮算完，随时可以读取！
- 若错写成外层枚举 `i`、内层枚举 `j`（**WA**）：当 $i=1, j=2$ 时，需要读取右边的 `mx[1][3]`，但外层才刚刚走到 $i=1$，下标 $i=3$ 这一整列全还是初始值 0！

记忆口诀

ST 表永远按区间长度从小到大一层层往上叠：先铺满长度 1，再铺长度 2、长度 4、长度 $8\ldots$

本页要点

“外 `j` 内 `i`”是所有倍增、区间 DP 的铁律：小区间全算好，才能拼大区间！

模板代码（1/2）：ST 表 $O(n \log n)$ 倍增预处理

```
int K = __lg(n) + 1;                                    // 所需最大倍增层数
vector<vector<int>> mx(K, vector<int>(n + 1));
for (int i = 1; i <= n; ++i)
    mx[0][i] = a[i];                                    // 第 0 层：长 2^0 = 1 就是自己

for (int j = 1; j < K; ++j) {                           // 务必外层枚举幂次 j
    for (int i = 1; i + (1 << j) - 1 <= n; ++i) {       // 右端 i + 2^j - 1 不能超出 n
        mx[j][i] = max(mx[j - 1][i],
                       mx[j - 1][i + (1 << (j - 1))]);
    }
}
```

本页要点

写位移优先级务必加括号：`1 << j` 放到加减法里一定要写成 `i + (1 << j) - 1`，因为 `+` 优先级高于 `<<`！

模板代码（2/2）：$O(1)$ 查询与 lg2[] 预处理对比

```
// 方式 A：GCC 内置 __lg(x)（O(1) 硬件指令，比赛可直接用）
int query(int l, int r) {
    int k = __lg(r - l + 1);
    return max(mx[k][l], mx[k][r - (1 << k) + 1]);
}
// 方式 B：自己线性递推 lg2[]（所有编译器通用，零浮点误差）
vector<int> lg2(n + 1, 0);
for (int i = 2; i <= n; ++i) lg2[i] = lg2[i / 2] + 1;
```

- 不要在 10^6 次查询里调用 <cmath> 的浮点函数 (int)log2(r - l + 1)，不仅比 __lg / lg2[] 慢，还可能有浮点精度风险！

本页要点

用 __lg(x) 或递推数组 lg2[i] = lg2[i >> 1] + 1，才是真正无误差的 $O(1)$ 查询！

上机验证与反事实追问：ST 表边界测试

3 组本地验证数据（序列 a = [3, 2, 4, 5, 6, 8]）

- 一般询问 [1, 5]：$k = 2$，查 mx[2][1]=5 与 mx[2][2]=6 $\Rightarrow$ 输出 6。
- 单点询问 [3, 3]：$k = 0$，查 mx[0][3]=4 与 mx[0][3]=4 $\Rightarrow$ 输出 4。
- 全长询问 [1, 6]：$k = 2$，查 mx[2][1]=5 与 mx[2][3]=8 $\Rightarrow$ 输出 8。

反事实追问

如果查询右段少写了 + 1，写成 mx[k][r - (1 << k)]，在查询单点 [1, 1]（$k = 0$）时会访问谁？它会去读 mx[0][1 - 1] = mx[0][0]（0 号未初始化单元）！若数组是负数，立刻输出错误的 0（**WA**）！

本页要点

每次写完 ST 表查询，立刻在心里拿 $[1, 1]$ 代进 r - (1 << k) + 1 算一遍是不是等于 1！

ST 表高频易错对照（WA / RE / TLE）

错误表现与后果	错误写法	正确写法与原因
右段起点少 +1（**WA**）	写成 `mx[k][r - (1 << k)]`	少包含最右端 r 或错读左边一格，务必 `r - (1 << k) + 1`
j，i 循环反了（**WA**）	外层枚举 i、内层枚举 j	右边短区间还没算就先算长区间，必须外 j 内 i
位运算漏括号（**WA/RE**）	写成 `i + 1 << j - 1`	加减优先级高于位移！必须写 `i + (1 << j) - 1`
幂次第一维开小（**RE**）	$n = 10^5$ 只开 `mx[16][N]`	$\lfloor \log_2 10^5 \rfloor = 16$，下标 16 需要开到 17（建议 `__lg(n)+1`）

本页要点

记住四条避坑线：位移加括号、外 j 内 i、右块补 +1、层数 `__lg(n)+1`！

例 1 | 亲戚（一本通 1346）[基础]：先读懂要求

大白话题意

一共有 n 个人（编号 $1 \sim n$），给你 m 条“a_i 和 b_i 是亲戚”的信息（亲戚关系可传递）。随后有 q 次询问，每次问 c_i 和 d_i 是不是亲戚，是就输出 `Yes`，否则输出 `No`。

- 输入：第 1 行 n, m（$n \le 20000,\ m \le 10^6$）；下接 m 行每行 a_i, b_i；然后整数 q（$q \le 10^6$）及 q 行询问 c_i, d_i。
- 处理：每读一条关系合并两人所在家族；每次询问看两人总掌门是否相同。
- 输出：共 q 行，每行 `Yes` 或 `No`（注意首字母大写）。

原题样例

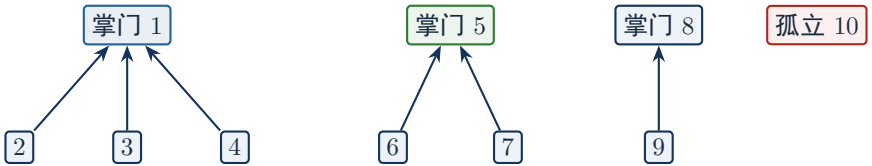
$n = 10, m = 7$，关系 `(2,4)(5,7)(1,3)(8,9)(1,2)(5,6)(2,3)`；$q = 3$ 问 `(3,4),(7,10),(8,9)` $\Rightarrow$ `Yes / No / Yes`

本页要点

数据量 m, q 高达 10^6，不仅算法要用 $O(\alpha(n))$ 并查集，还必须关闭同步流防读入输出超时！

例 1 | 亲戚 | 思路与树林建模图解

把每个“亲戚圈子”建成一棵并查集树，样例 7 条关系合并完后，10 个人自动分成 4 棵互不相干的小树：



- 问 (3, 4): `find(3)=1 == find(4)=1` ⇒ Yes;
- 问 (7, 10): `find(7)=5 != find(10)=10` ⇒ No;
- 问 (8, 9): `find(8)=8 == find(9)=8` ⇒ Yes。

本页要点

注意原题输入有 $q = 3$ 放在第 $m + 2$ 行，读完 m 条边后再读 q!

例 1 | 亲戚 | 样例手推跟踪表

初始 p[1..10] = [1,2,3,4,5,6,7,8,9,10]，用“小树挂大树（同大时前者为根）”手推 7 条关系：

步骤	输入边 / 询问	执行动作与找根结果	关键连通块变化
1	unite(2, 4)	根 2，4 规模均 1，4 挂 2	{2,4}(根 2，sz=2)
2	unite(5, 7)	根 5，7 规模均 1，7 挂 5	{5,7}(根 5，sz=2)
3	unite(1, 3)	根 1，3 规模均 1，3 挂 1	{1,3}(根 1，sz=2)
4	unite(8, 9)	根 8，9 规模均 1，9 挂 8	{8,9}(根 8，sz=2)
5	unite(1, 2)	根 1(sz=2)，2(sz=2)，2 挂 1	{1,2,3,4}(根 1，sz=4)
6	unite(5, 6)	根 5(sz=2)，6(sz=1)，小 6 挂大 5	{5,6,7}(根 5，sz=3)
7	unite(2, 3)	find(2)=1，find(3)=1，已同根跳过	集合不变（顺便压扁 p[2]=1）

本页要点

看第 7 步：(2，3) 本来就在同一块里，unite 自动识别 x == y 并跳过，毫无副作用！

例 1 | 亲戚 | 核心步骤与相对模板改动

按顺序计算	纸上得到	核对问题
1. 初始化 `DSU dsu(n + 1)`	`p[i]=i, sz[i]=1`	1-based 是否开到了 `n + 1`?
2. 读入 m 条边调用 `unite(x, y)`	形成 4 棵树林	有没有关闭 `cin` 同步流?
3. 读入 q 次询问，比 `find(x)==find(y)`	`Yes / No / Yes`	是否调用了 `find` 而不是比 `p`?

相对模板改了哪里？

直接复用第 2 节标准 `DSU` 结构体！只把最后查询结果按题目严格要求的 `"Yes"` / `"No"` 换行打印，并加 `ios::sync_with_stdio(false); cin.tie(nullptr);` 加速百万级 IO！

本页要点

$m, q = 10^6$ 时用 `endl` 会刷新缓冲区百万次导致 **TLE**，输出换行务必用 `'\n'`！

例 1 | 亲戚 | 核心代码（1/2）：DSU 结构体

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

struct DSU {
    vector<int> p, sz;
    explicit DSU(int n) : p(n), sz(n, 1) { iota(p.begin(), p.end(), 0); }
    int find(int x) { return p[x] == x ? x : (p[x] = find(p[x])); } // 路径压缩
    bool unite(int x, int y) {
        x = find(x); y = find(y);
        if (x == y) return false;
        if (sz[x] < sz[y]) swap(x, y);              // 小树挂大树下面防退化
        p[y] = x; sz[x] += sz[y];
        return true;
    }
};
```

本页要点

这段 DSU 结构体只有 12 行，通用性强，后续做并查集题目可以直接背熟手写！

例 1 ｜亲戚｜核心代码（2/2）：主函数读入与查询

```cpp
int main() {
    ios::sync_with_stdio(false); cin.tie(nullptr);
    int n, m; if (!(cin >> n >> m)) return 0;
    DSU dsu(n + 1);
    for (int i = 0, x, y; i < m; ++i) {
        cin >> x >> y;
        dsu.unite(x, y);
    }
    int q; cin >> q;
    while (q--) {
        int x, y; cin >> x >> y;
        cout << (dsu.find(x) == dsu.find(y) ? "Yes" : "No") << '\n';
    }
    return 0;
}
```

本页要点

完整源码见 code/dsu_relations.cpp，已通过 C++17 编译与原题样例验证！

例 1 | 亲戚 | 易错与 3 组自测边界

高频易错（对比排雷）

- 输出大小写写错（**WA**）：错写成 `YES/NO` 或 `yes/no`，题目严格要求 `Yes` 与 `No`。
- 写 `p[x] == p[y]`（**WA**）：合并后某些叶子还没触发压缩，必须写 `dsu.find(x) == dsu.find(y)`。
- 未关同步流或用 `endl`（**TLE**）：$m, q \le 10^6$ 量级非常吃 IO 速度，必用 `cin.tie(nullptr)` + `'\n'`。

3 组自测边界与预期输出

1. 单人边界 $n = 1$：`1 0 / 1 / 1 1` ⇒ 自己和自己是亲戚，输出 `Yes`。
2. 全无关系：`3 0 / 2 / 1 2 / 2 2` ⇒ `No` 与 `Yes`。
3. 原题样例：输出 `Yes / No / Yes`。

本页要点

哪怕 $m = 0$ 没有任何关系，一个人问自己（`x`，`x`）也一定是 `Yes`！

例 2 | 格子游戏（一本通 1347）[基础]：先读懂要求

大白话题意

在一个 $n \times n$ 的正方形点阵上，两人轮流在相邻格点之间画线：给定起点 (r, c) 和方向（`'D'` 向下连一格，`'R'` 向右连一格）。求第几步第一次围成一个封闭的圈？如果 m 步画完都没有封闭圈，输出 `draw`。

- 输入：第一行 n, m（$n \le 200$）；下接 m 行每行两个坐标 r, c（$1 \sim n$）和字符 `dir`（`'D'` 或 `'R'`）。
- 处理：每画一条线相当于在两点间加一条边；一旦连边前发现两点早已连通，这第 *step* 条边必封成环！
- 输出：一行，成环的最早步数；未成环则输出 `draw`。

原题样例

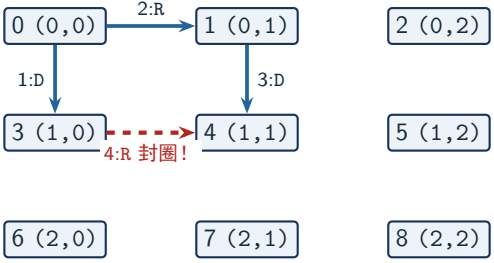
`3 5 / 1 1 D / 1 1 R / 1 2 D / 2 1 R / 2 2 D` ⇒ 输出：`4`

本页要点

题目说了面积不必为 1，也就是说只要任意多个点连出封闭环路都算赢，正合“并查集判环”！

例 2 ｜格子游戏｜思路与二维转一维建模图解

把 3×3 坐标 (r, c)（先减 1 变成 0-based $0 \sim n-1$）拉平成一维编号 `id = r * n + c`：



- 方向 `'D'` 连向 `(r+1, c)` → 编号 `(r+1)*n + c`；方向 `'R'` 连向 `(r, c+1)` → 编号 `r*n + c + 1`。

本页要点

并查集判环黄金法则：连边 (u, v) 之前，如果 `find(u) == find(v)`，说明早有另一条路把 u 和 v 连着，这条边落笔立刻成环！

例 2 | 格子游戏 | 样例手推跟踪表

3×3 网格共 9 个点（0-based 编号 $0 \sim 8$），初始 `p[i] = i`：

步数	读入 (r, c, dir)	转 0-based 边 (u, v)	ru=find(u), rv=find(v)	动作与判定
1	1 1 D	(0,0)->(1,0): (0, 3)	ru=0, rv=3（不等）	合并 p[3]=0
2	1 1 R	(0,0)->(0,1): (0, 1)	ru=0, rv=1（不等）	合并 p[1]=0
3	1 2 D	(0,1)->(1,1): (1, 4)	ru=0, rv=4（不等）	合并 p[4]=0
4	2 1 R	(1,0)->(1,1): (3, 4)	ru=0, rv=0（相等!）	第 **4** 步成环，输出 4 退出
5	2 2 D	（已提前退出不处理）	—	—

本页要点

第 4 步结束时立即 `return 0;`，既不会错把第 5 步再输出一遍，也不会在结尾多打印 `draw!`

例 2 | 格子游戏 | 核心步骤与相对模板改动

按顺序计算	纸上得到	核对问题
1. 二维点映射一维 `r*n + c`	$(1,1) \to 0,\ (2,1) \to 3$	读入坐标有没有先 `--r; --c;`?
2. 按 `'D'/'R'` 算终点 `v`	向下 `+n`，向右 `+1`	有没有搞反向下加 `n` 和向右加 `1`?
3. 连边前判 `find(u) == find(v)`	第 4 步均得根 `0`	成环时是否立刻输出 `step` 并退出?
4. 走完 m 步均未触发	打印 `draw`	字符串大小写是不是全小写 `draw`?

相对模板改了哪里？

把二维格点映射成 $0 \sim n^2 - 1$ 的一维编号； 利用 `ru == rv` 抓出第一条成环边提前结束程序！

本页要点

用 `r * n + c` 映射到一维数组后，并查集根本不用写二维结构体，几十行代码轻松搞定！

例 2 | 格子游戏 | 核心代码（1/2）：映射开数组与路径压缩

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

int main() {
    ios::sync_with_stdio(false); cin.tie(nullptr);
    int n, m; if (!(cin >> n >> m)) return 0;
    vector<int> p(n * n);
    iota(p.begin(), p.end(), 0);                    // 二维 (r,c) -> 0-based 一维 r
    *n + c
    auto find = [&](auto &&self, int x) -> int {
        return p[x] == x ? x : p[x] = self(self, p[x]);
    };
```

本页要点

也可以把第 1 题的 `struct DSU` 直接搬过来写 `DSU dsu(n * n);`，可读性更强且自带按大小合并！

例 2 | 格子游戏 | 核心代码（2/2）：逐边判环主循环

```
    for (int step = 1; step <= m; ++step) {
        int r, c; char dir; cin >> r >> c >> dir;
        --r; --c;                                          // 转成 0-based 坐标
        int u = r * n + c;
        int v = (dir == 'D' ? (r + 1) * n + c : r * n + c + 1);
        int ru = find(find, u), rv = find(find, v);
        if (ru == rv) {                                    // 连成封闭环，立刻输出步数结束！
            cout << step << '\n';
            return 0;
        }
        p[rv] = ru;
    }
    cout << "draw\n";
    return 0;
}
```

本页要点

完整源码见 code/grid_cycle.cpp，已通过 C++17 编译与原题样例实测！

例 2 | 格子游戏 | 易错与 3 组自测边界

高频易错（对比排雷）

- 忘记先做 `--r; --c;`：题目坐标 $1 \sim n$，若直接算 `r * n + c`，点 (n, n) 会变成 $n^2 + n$ 越界触发 **RE**！
- 成环后继续循环没退出：后面又绕出第二个环时多打印好几个数字（WA），成环一定要立刻 `return 0;`！

3 组自测边界与预期输出

1. 无步数 $m = 0$：`2 0` $\Rightarrow$ `draw`。
2. 画成一长条不成环：`3 4 / 1 1 R / 1 2 R / 1 3 D / 2 3 D` $\Rightarrow$ `draw`。
3. 原题样例：`3 5 ...` $\Rightarrow$ `4`。

本页要点

画完 4 条边不一定成环（比如 L 形拐弯），只要没围出闭合区域就必须坚持到最后输出 `draw`！

例 3 | Balanced Lineup（一本通 1545）[提高]：先读懂要求

大白话题意

农场里有 N 头牛排成固定一排，第 i 头牛身高为 h_i。有 Q 次询问，每次给一段区间 $[A, B]$，求这段牛里最高身高与最矮身高的差值（极差）。

- 输入：第 1 行 N, Q（$1 \le N \le 5 \times 10^4,\ 1 \le Q \le 1.8 \times 10^5$）；下接 N 行身高 h_i（$\le 10^6$）；再下接 Q 行区间 A, B。
- 处理：序列完全静态且 Q 高达 1.8×10^5，同时建两张 ST 表 `mx`（最大）与 `mn`（最小），每问 $O(1)$ 输出 max − min。
- 输出：共 Q 行，每行一个整数表示该区间身高极差。

原题样例

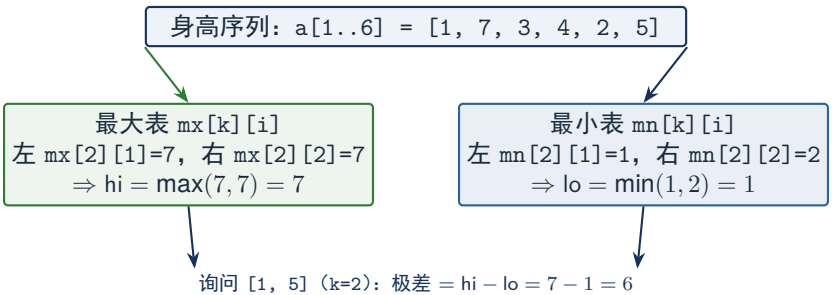
$N = 6, Q = 3$，身高 `[1, 7, 3, 4, 2, 5]`；询问 `[1,5]`, `[4,6]`, `[2,2]` ⇒ 输出 `6 / 3 / 0`

本页要点

牛排好队不动、只查不改、询问近 20 万次——这正是 ST 表“预处理一次、$O(1)$ 秒答”的主场！

例 3 | Balanced Lineup | 思路与双 ST 表建模图解

用一张最大表 `mx[j][i]` 和一张最小表 `mn[j][i]` 平行倍增，把区间极差拆成两次 $O(1)$ 查表：



本页要点

最大值和最小值都满足“幂等律”，可以在同一个双层循环里并排预处理完成！

例 3 | Balanced Lineup | 样例手推跟踪表

身高 `[1, 7, 3, 4, 2, 5]` 的双 ST 表前 3 层及 3 个样例询问手推结果：

层数（长度）	$i=1$	$i=2$	$i=3$	$i=4$	$i=5$	$i=6$
$j=0$ `mx / mn`	$1/1$	$7/7$	$3/3$	$4/4$	$2/2$	$5/5$
$j=1$ `mx / mn`	$7/1$	$7/3$	$4/3$	$4/2$	$5/2$	—
$j=2$ `mx / mn`	$7/1$	$7/2$	$5/2$	—	—	—

- 问 `[1, 5]`（$k=2$）：$\max(\texttt{mx[2][1]}, \texttt{mx[2][2]}) - \min(\texttt{mn[2][1]}, \texttt{mn[2][2]}) = 7 - 1 = \mathbf{6}$。
- 问 `[4, 6]`（$k=1$）：
  $\max(\texttt{mx[1][4]}, \texttt{mx[1][5]}) - \min(\texttt{mn[1][4]}, \texttt{mn[1][5]}) = \max(4,5) - \min(2,2) = 5 - 2 = \mathbf{3}$。
- 问 `[2, 2]`（$k=0$）：$\max(7,7) - \min(7,7) = \mathbf{0}$。

本页要点

每步手算结果与原题样例输出 `6 / 3 / 0` 完全吻合！

例 3 | Balanced Lineup | 核心步骤与相对模板改动

按顺序计算	纸上得到	核对问题
1. 初始化 $j=0$ 两张表	`mx[0][i] = mn[0][i] = h[i]`	1-based 是否开到了 `n + 1`?
2. 外 `j` 内 `i` 填满双表	`j=1, 2, ..., K-1`	`i + (1 << j) - 1 <= n` 括号写了没?
3. 对每个 $[l,r]$ 算 $k=__\lg(r-l+1)$	$k=2,1,0$	长度 1 时 `__lg(1) == 0` 对不对?
4. 算 `hi - lo` 换行输出	`6, 3, 0`	是否用 `'\n'` 防 1.8×10^5 次刷新超时?

相对模板改了哪里?

从单张 `mx` 表拓展为并行的 `mx + mn` 双表，用同一份下标公式同时求区间最大和最小再相减!

本页要点

用 `vector<vector<int>> mx(K, vector<int>(n + 1))` 时，若 $n=50000, K=16$ 仅占约 6 MB 内存，完全不超!

例 3 | Balanced Lineup | 核心代码（1/2）：双 ST 表预处理

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

int main() {
    ios::sync_with_stdio(false); cin.tie(nullptr);
    int n, q; if (!(cin >> n >> q)) return 0;
    int K = __lg(n) + 1;
    vector<vector<int>> mx(K, vector<int>(n + 1)), mn(K, vector<int>(n + 1));
    for (int i = 1; i <= n; ++i) { cin >> mx[0][i]; mn[0][i] = mx[0][i]; }
    for (int j = 1; j < K; ++j)
        for (int i = 1; i + (1 << j) - 1 <= n; ++i) {
            mx[j][i] = max(mx[j - 1][i], mx[j - 1][i + (1 << (j - 1))]);
            mn[j][i] = min(mn[j - 1][i], mn[j - 1][i + (1 << (j - 1))]);
        }
```

本页要点

预处理 $O(N\log N)$ 只跑一次，当 $N = 5 \times 10^4$ 时循环体最多执行不到 80 万次！

例 3 | Balanced Lineup | 核心代码（2/2）：$O(1)$ 查询区间极差

```
    while (q--) {
        int l, r; cin >> l >> r;
        int k = __lg(r - l + 1);                    // 取不超过区间长度的最大 2^k
        int hi = max(mx[k][l], mx[k][r - (1 << k) + 1]);
        int lo = min(mn[k][l], mn[k][r - (1 << k) + 1]);
        cout << hi - lo << '\n';
    }
    return 0;
}
```

本页要点

完整源码见 `code/st_rmq.cpp`，已通过 C++17 编译与原题样例实测！

例 3 | Balanced Lineup | 易错与 3 组自测边界

高频易错（对比排雷）

- 初始化漏赋 `mn[0][i]`：只读进了 `mx[0][i]`，忘了写 `mn[0][i] = mx[0][i]`，导致所有最小值全算成 0（WA）。
- 查询少写 `+1` 或漏关同步流：少 `+1` 会偏左一格；$Q = 1.8 \times 10^5$ 忘了关同步流会拖慢 IO。

3 组自测边界与预期输出

1. 单点边界 $A = B$：`1 1 / 42 / 1 1` $\Rightarrow$ `0`（极差为 0）。
2. 全相等序列：`4 1 / 5 5 5 5 / 1 4` $\Rightarrow$ `0`。
3. 原题样例：`6 3 / 1 7 3 4 2 5 / 1 5 / 4 6 / 2 2` $\Rightarrow$ `6 / 3 / 0`。

本页要点

交题前一定同时测一组 $N = 1$ 单点查询和一组全相等序列，确保 `mn[0][i]` 没漏初始化！

并查集与 ST 表高频错误对照与定位指南

题目 / 场景	常见错误写法	正确做法与原因
未调 find 直接比父亲	写 `if (p[x] == p[y])`	可能没来得及路径压缩到同一根，必须写 `find(x) == find(y)`
并查集忘记初始化	开完数组忘记 `p[i] = i`	全数组是 0 导致所有人一开始就连通在一起
二维编号乘错系数	二维坐标 (r, c) 转成 `r * m + c` 或未减 1	$n \times n$ 网格先 `--r; --c;` 再算 `r * n + c`，数组开足 `n * n`
ST 表循环顺序颠倒	外层枚举起点 `i`、内层枚举幂次 `j`	长区间依赖已经算好的短区间，必须外层枚举 `j`！
ST 表查询右半起点	写 `mx[k][r - (1 << k)]` 少 `+ 1`	长度为 2^k 且以 r 结尾的起点是 `r - (1 << k) + 1`！

- 定位口诀：**WA** 先查 `r - (1<<k) + 1` 少没少 +1、外层是不是枚举 `j`、判断连通是不是调了 `find`；**RE** 查 1-based 数组是否开到 `n+1` 与 ST 表层数 `K`；**TLE** 查百万读入是否关同步流、换行有没有用 `'\n'`。

本页要点

并查集问必调 `find`，ST 表“外 `j` 内 `i`”填表、“`r - (1<<k) + 1`”对齐右端！

提交前检查清单（六必查）

1. 并查集开数组大小是否覆盖了 0-based 或 1-based 全部节点，且执行了 `p[i] = i` 初始化？
2. 判断两点连通或判环时，是否先对两端各自调用 `find(u)` 与 `find(v)`？
3. 使用 `sz[]` 统计人数时，是否判了 `if (x == y) return false;` 防重复相加，且从根 `sz[find(x)]` 读取？
4. ST 表第一维 K 是否开够 `__lg(n) + 1`，且最外层循环是 `j`？
5. ST 表内层构建边界 `i + (1 << j) - 1 <= n` 和查询右起 `r - (1 << k) + 1` 没少 `+1/-1` 与位运算括号？
6. 若查询单点区间 $l == r$，公式算得 `__lg(1) = 0`，是否能正确返回单个元素极值（极差 0）？

最后一步

用单点查询 $[i, i]$ 和全长查询 $[1, n]$ 验证 ST 表两端无越界；用自环或重复边测试并查集再交！

本章全部练习与完成标准

1. 学校 OJ：P1346 | 亲戚（[基础]，一本通 1346，code/dsu_relations.cpp）：路径压缩 + 按大小合并维护动态连通性。
2. 学校 OJ：P1347 | 格子游戏（[基础]，一本通 1347，code/grid_cycle.cpp）：二维坐标转一维编号，连边前判同根抓第一个环。
3. 学校 OJ：P1545 | Balanced Lineup（[提高]，一本通 1545，code/st_rmq.cpp）：双 ST 表 $O(n \log n)$ 预处理、$O(1)$ 求静态区间极差。
4. 学校 OJ：P1541 | 数列区间最大值（[基础]，一本通 1541）：$N \le 10^5, M \le 10^6$ 静态区间最大值（注意关闭同步流防 TLE！）。

完成标准

能闭眼写出路径压缩 + 按大小合并的 DSU 结构体；能说清为什么 ST 表不能算区间和且外层必须枚举 j；独立 AC 上述 4 道习题。

课后安排与下一章预告

- 复习：合上代码在纸上验算一遍 $n = 6, l = 1, r = 5$ 时 ST 表 `k=2` 拼出来的左段 $[1..4]$ 与右段 $[2..5]$；手画一遍带权并查集压缩时 `d[x] += d[p]` 的顺序。
- 练习：把 `code/` 下 3 份源码自己独立打一遍，重点测单点查询 $[1, 1]$ 与《数列区间最大值》的百万级查询。
- 下一章预告：第 5 章《树状数组与线段树》——当数组既有高频查询、又要随时单点甚至区间修改时（ST 表失效了！），如何用 `lowbit` 和懒标记把动态维护做到 $O(\log n)$！

本章一句话

并查集靠扁平化掌门实现近乎 $O(1)$ 连通判环，ST 表靠可重叠倍增块做到 $O(1)$ 静态最值秒答。

谢谢！