数据结构漫谈

南京外国语学校　许昊然

数据结构题是 OI 竞赛中经常出现的题目，本文漫谈了与序列、字符串、树有关的一些数据结构题，希望能起到抛砖引玉的作用。因为作者水平非常有限，文中错误疏漏还望读者指出。

目录

1 数据结构在序列中的应用——树形结构与分块

与序列有关的数据结构题在 OI 竞赛中出现频率很高。这类题目的一般模式是，给定一个序列，然后要求支持修改/插入序列元素或者对这个序列的某个子序列进行查询。

1.1 几个约定

- 为方便叙述，我们约定序列长度为 N，下标从 1 开始，所有序列元素都是正整数，且小于 2^{31}。
- 约定 $a[i]$ 表示序列的第 i 个元素。
- 约定区间 $[L,R]$ 表示序列的第 L 个数到第 R 个元素组成的子序列。

1.2 从一个经典问题引入—— RMQ

给定一个序列和一些询问，每个询问要求回答区间 $[L,R]$ 中最大的数。

这个问题实在是非常古老的问题，做法也非常多。我们这里只介绍一种线段树的做法。这个做法的关键是利用线段树，任何一个区间都可以分解成 $O(\log N)$ 个线段树上的小区间，而这些小区间的答案都是可以提前预处理出的，合并起来就得到了这个查询的答案。于是我们做到了 $O(\log N)$ 每次查询。注意到，这个做法的几个关键词有：任务分解，预处理，合并答案。本文这一部分都将围绕这三个关键词展开。

1.3 两种任务分解方法——树形结构与分块结构

1.3.1 树形结构

树形结构，主要是指线段树和 Splay。线段树是一棵深度是 $O(\log N)$ 的二叉树，每个结点对应原序列的一个区间。假设非叶结点结点 P 对应的区间是 $[L,R](L<R)$，那么 P 的左孩子 $Left[P]$ 对应区间就是 $[L,(L+R)/2]$，P 的右孩子 $Right[P]$ 对应区间就是 $[(L+R)/2+1,R]$。显然任意一个询问或修改 $[L,R]$ 都可以用 $O(\log N)$ 个线段树结点对应的区间“拼接”而成。Splay 也类似，但因为其特殊性可以支持插入元素的操作。树形结构的核心就是合并答案，无论是树的建立或是修改或是查询，都要求把两个子问题的结果拼接起来，拼出父问题的答案。于是，我们发现，可以用树形结构维护的题目只需满足以下要求：

- 支持快速合并两个序列的结果
- 如果有懒标记的话，还应当支持对一个序列快速应用懒标记和快速合并两个懒标记

用树形结构维护序列的数据结构题包括经典的 SPOJ GSS 系列和 NOI2005 的维护数列。

【例 1】SPOJ GSS1

给定一个序列和一些询问，要求支持查询区间最大部分和操作。也就是对每个询问 $[L,R]$, 要找到 $L\le i\le j\le R$，使得 $a[i]+a[i+1]+\ldots+a[j]$ 的值最大，返回这个最大值。

这题非常经典，考虑一个询问 $[L,R]$ 的答案，我们取 $L\le M\le R$，那么 $[L,R]$ 的答案必然是区间 $[L,M]$ 的答案，区间 $[M+1,R]$ 的答案，和跨越 M 的区间答案这三者的最大值。

而跨越 M 的区间最大值必然是 $[L,M]$ 的最大后缀和加上 $[M+1,R]$ 的最大前缀和。于是我们对线段树节点 $P[L,R]$ 维护 4 个域，分别是前缀最大和 $L_{\max}$，后缀最大和 $R_{\max}$，最大部分和 $M_{\max}$ 和整段区间的和 S。

合并答案时，我们显然有：

$$\begin{aligned}
L_{\max}(P) &= \max\{L_{\max}(\text{Left}[P]),\ S(\text{Left}[P])+L_{\max}(\text{Right}[P])\}\\
R_{\max}(P) &= \max\{R_{\max}(\text{Right}[P]),\ S(\text{Right}[P])+R_{\max}(\text{Left}[P])\}\\
M_{\max}(P) &= \max\{M_{\max}(\text{Left}[P]),\ M_{\max}(\text{Right}[P]),\ R_{\max}(\text{Left}[P])+L_{\max}(\text{Right}[P])\}\\
S(P) &= S(\text{Left}[P])+S(\text{Right}[P])
\end{aligned}$$

于是我们解决了合并答案的问题，这道题在 $O(\log N)$ 每次询问的时间内得到了解决。

【例 2】SPOJ GSS3

给定一个序列，要求在支持查询区间最大部分和操作的基础上，还要支持区间整体赋值操作。

我们发现所谓整体赋值，显然就是一个懒标记操作。对一个序列应用一个懒标记并更新结果，显然非常简单。合并懒标记也很简单，后到的懒标记直接覆盖掉原先的懒标记就行了。于是这题也可以用线段树解决。

【例 3】NOI2005 维护数列

给定一个序列，要求支持：在某个位置插入一个序列，删除一段序列，对一段序列进行整体赋值，对一段序列进行翻转，查询一段序列的最大部分和。

插入序列和删除序列就是 Splay 的工作。具体维护的域和懒标记只需要支持整体赋值、翻转和最大部分和操作。最大部分和要维护的域和整体赋值操作的处理前文已经讨论过，我们只要支持翻转操作就行了。显然翻转操作也是一个懒标记，实际上，对一段序列应用翻转懒标记可以表示为，交换这个序列对应的 Splay 结点的左子树和右子树，并对其左子树和右子树也应用翻转标记。两个翻转懒标记的合并显然会导致标记被清除。而翻转懒标记与整体赋值懒标记的合并也很显然，因为先翻转后整体赋值和先整体赋值后翻转完全没有区别，所以我们根本不需要关心这两个标记的先后次序，这两个标记完全可以共存。于是，这道题也在 $O(\log N)$ 每次操作的复杂度内解决了。

1.3.2 分块结构

但是并不是所有题目要维护的东西都可以做到支持快速合并两个子序列的结果。经典的例子有：区间众数，带区间加减操作的区间最大前缀和，区间最大部分 XOR 和等。

分块结构的优势这时候就体现出来了。所谓分块，其实是一种平衡的思想，一般而言，就是把序平均分成 S 块（S 值可视情况而定），显然任意一个区间询问或修改都可以由 $O(S)$ 个完整的块和至多 2 个不完整的块覆盖。往往完整的块可以预先维护好，不完整的块因为最

多只有 2 个，每个块只有 $O(\frac{N}{S})$ 个元素，往往可以暴力解决。我们发现，分块做法较树形结构的优势在于：

- 分块结构在建立和修改时不需要合并两个子序列的结果（尽管有时查询依然要求快速合并答案）
- 在大多数情况下，分块结构可以不支持快速合并两个懒标记。

这两个优势使得分块做法能解决很多树形结构不能解决的问题。

我们可以根据“在查询时，是否要求快速合并答案”把用分块结构解决的数据结构题分为两类。

我们首先介绍查询时不要求快速合并答案的分块结构。

【例 4】经典问题

给定一个序列，要求在线支持区间加减（区间里每个数都加上或减去某个数）和查询区间里第 K 大的数。

如果我们用树形结构维护，那么对于每个节点 $P[L,R]$，我们都要维护区间 $[L,R]$ 排序后的序列。显然合并两个子结点的排序后的序列得到父结点的排序后的序列，即使用归并排序也是 $O(N)$ 的，无法快速支持区间加减操作。于是线段树不能解决这个问题。

我们发现问题矛盾在于不能快速合并答案。而分块结构恰好在建立和修改时不需要快速合并答案。我们考虑用分块结构维护。

我们考虑把序列平均分成 S 块，对每一块，维护其对应的区间排好序后的结果，并使用懒标记记录这个块被全体区间加减的值。显然全体区间加减不会改变排序后元素的相对位置，于是我们支持了快速应用懒标记。对于修改时没有被完整覆盖的块，因为最多只有 2 个，我们直接暴力修改并重新排序，时间复杂度 $O(\frac{N}{S}\log\frac{N}{S})$。查询时，我们虽然不能快速合并出整个查询区间排序后的序列，但通过二分答案，查询可以转化为在每个区间里求有多少个数比某个小，这样就避免了合并两个序列的答案。于是查询的时间复杂度是 $O(S\log S\log N)$。为了让查询的修改的时间差不多，我们令

$$\begin{aligned} O\left(\frac{N}{S}\log\frac{N}{S}\right) &= O(S\log S\log N) \\ O(S\log S) &= O\left(\frac{N}{S}\right) \\ S &= O(\sqrt{N}) \end{aligned}$$

取 $S=\sqrt{N}$，我们得到了一个 $O(\sqrt{N}\log^2 N)$ 每次操作的做法。

【例 5】SPOJ UNTITLE1

给定一个序列，要求在线支持区间加减操作和查询区间最大前缀和。

首先，我们发现直接用例 1 的方法维护部分和序列是不行的，因为不能支持对一段序列快速应用懒标记。我们不妨将一个序列的前 i 项的和 $s[i]$ 表示为点 $(i,s[i])$ 画在平面上。这样，对整个序列应用区间加减操作就等价于把点 $(i,s[i])$ 变换到点 $(i,s[i]+X*i)$（假设区间

加减的值是 X)。而询问操作，就是求最大的 Y 坐标。进一步观察，我们发现对整个序列进行区间加减很类似于旋转了一下坐标系。因此我们猜测，最大前缀和对应的点必然在这些点组成的上凸壳上。事实上，这是可以用数学方法证明的。如 j 不在上凸壳上，那么必然存在 $i < j < k$, 使得

$$\text{slope}(P[i], P[j]) < \text{slope}(P[j], P[k]) \quad \text{其中 slope}(X, Y)\text{表示向量}\overrightarrow{XY}\text{的斜率}$$

$$\frac{s[j] - s[i]}{j - i} < \frac{s[k] - s[j]}{k - j}$$

假如 j 在区间加减 X 的情况下成为了最优解，那么 j 必然同时优于 i 和 k。

于是有

$$s[j] + j * X > s[i] + i * X$$

$$s[j] + j * X > s[k] + k * X$$

即

$$\frac{s[k] - s[j]}{k - j} < -X < \frac{s[j] - s[i]}{j - i}$$

$$\frac{s[j] - s[i]}{j - i} > \frac{s[k] - s[j]}{k - j}$$

与前面的条件矛盾！于是我们证明了，最优解一定位于上凸壳上。

假如我们用线段树维护，$P[L, R]$ 维护区间 $[L, R]$ 组成的上凸壳，同样我们发现，合并两个上凸壳只能做到 $O(N)$，不能支持快速修改，于是分块结构再次大显身手。考虑把序列平均分成 S 块，对于每一块维护凸壳，那么显然懒标记的应用不会影响凸壳，直接记录一下就行了，对非整块的修改可以暴力重建凸壳；而查询时，对于被完整覆盖的块，显然凸壳的 Y 坐标具有单峰性，可以三分找最大值。对于非完整覆盖的块，暴力即可。同样取 $S = \sqrt{N}$，我们做到了每次操作 $O(\sqrt{N} \log N)$。

【例 6】Codeforces 86D Power Array

给定一个序列，对于一个区间 $[L, R]$，定义 $K(L, R, x)$ 表示数 x 在区间 $[L, R]$ 中出现次数。定义 $w(L, R, x) = K(L, R, x) * K(L, R, x) * x$。定义 $F[L, R] =$ 所有 $w[L, R, x]$ 的和。每次询问一个 $F[L, R]$ 的值。

我们发现这道题的询问很难下手，但发现这题没有要求在线算法，于是我们想到使用离线做法。我们把所有的询问按 R 从小到大排序，然后依次处理每个询问。假设当前处理到了 R_0，我们要回答所有形如 $[L, R_0]$ 的询问。但这个问题依然不是很好处理。换一个做法，我们令 $D[i]$ 表示数 $a[i]$ 在区间 $[i, R_0]$ 贡献的代价，于是如果 $a[i]$ 出现了 X 次，那么 $D[i] = (2X - 1) * a[i]$。那么查询 $F[L, R_0]$ 就可以转化为查询区间 $[L, R_0]$ 里所有 $D[i]$ 的和，而从 R_0 转移到 $R_0 + 1$ 就相当与往右端加入一个新数，就相当于对所有的满足 $a[i] = a[R_0 + 1]$ 的 i 值，把 $D[i]$ 加上 $2 * a[i]$。如果用树形结构维护 D，我们会发现我们不能支持合并两个懒标记。于是我们只能把所有懒标记都记录下来，这样下传标记的复杂度就变成了 $O(N)$，不

可接受。于是我们想到了块状结构。如果将序列平均分成 S 块，对每一块维护对应区间里所有 $D[i]$ 的和，那么显然懒标记的应用是 $O(1)$ 的，修改时就是打标记。查询时，对于对于完整的块答案就是所有 $D[i]$ 的和，对于不被完整覆盖的块的查询直接暴力，因为 $D[i]$ 的意义就是 $a[i]$ 在 $[i, R_0]$ 的贡献，所以直接逐个计算 $D[i]$ 就行了。总时间复杂度 $O((N+Q)*\sqrt{N})$.

但是对于某些题目，我们不仅不能快速合并两个序列的答案，更加糟糕的是，查询时我们不能将一个查询直接等价的用很多小区间上的独立的查询来表示，必须合并出整个查询区间才能得到答案。经典的例子有区间众数和区间最大部分 xor 和。但这类看似非常棘手的问题实际上也有很优美的处理技巧和算法。

接下来，我们将介绍查询时要求快速合并答案的问题的一些处理方法。

【例 7】BZOJ 2724 蒲公英

给定一个序列，要求在线回答区间 $[L, R]$ 的众数，如果有多个，输出最小的一个。

如果用树形结构来维护这个问题，会发现几乎无从下手。

两个小区间的众数和它们拼出的大区间的众数几乎没有任何关系，我们必须维护每个数在区间里出现的次数，但这样势必导致合并两个区间结果的复杂度是 $O(N)$。

更加麻烦的是，就算用块状结构维护，那么尽管建立的时候不需要合并两个区间的结果，但是查询时还是必须把整个大区间合出来才能得到答案，而不能像前两题一样，把一个询问转化为只与小区间有关的一系列独立的询问，这样就与朴素无异了。

我们不妨先考虑一个弱化版的题目，假如可以离线处理，怎么做？这个问题在张昆玮的论文《统计的力量——线段树详细教程》中提到过，文中给出的解法大致是这样：我们用线段树维护一段区间的每个数出现次数，那么假设当前的线段树维护的区间是 $[L, R]$，我们下一个询问在 $[L', R']$, 那么我们就应该把 $[L', L')$,$(R', R]$ 里的数都从线段树里删掉（或补足），这个操作次数不会超过 $|L-L'|+|R-R'|$，假如把询问 $[L, R]$ 表示为平面上的点 (L, R)，我们就发现，从一个询问转移到另一个询问的代价是这两点的曼哈顿距离。于是，用平面图曼哈顿最小生成树 $O(N \log N)$ 获得一个较优的总转移次数，可以证明这个总转移次数是 $O((N+Q)\sqrt{N})$ 级别的。于是总时间复杂度就是 $O((N+Q)\sqrt{N}\log N)$.

这个做法非常之复杂，因为要写一个平面图曼哈顿最小生成树。这里给出一个更加简单的离线算法。我们依然把询问 $[L, R]$ 表示为点 (L, R), 估价函数也不变，但我们不求其最小生成树，而是构造一个较小的生成树。具体就是，把横轴平均分成 $O(\sqrt{N})$ 段，每段里直接按纵轴上升次序处理，显然横轴转移代价不超过 $N\sqrt{N}$, 纵轴转移代价不超过 $Q\sqrt{N}$，所以时间复杂度还是 $O((N+Q)\sqrt{N}\log N)$，但是没有了平面图曼哈顿最小生成树，实现简单了很多。

离线做法给我们带来了一些启示，因为不能快速合并答案，我们只能以朴素方法合并出区间 $[L, R]$ 的答案，我们要做的就是通过预处理和其他技巧使得这个代价尽量小。

我们考虑把这个序列分成 S 块，然后把所有的连续的一段块里，每个数出现了多少次记录下来，这一步显然是 $O(N*S^2)$ 的。然后考虑对一个询问 $[L, R]$，必然能找到其能覆盖的最大的一段已经预处理出的区间，而没有被覆盖的元素的个数显然是 $O(\frac{N}{S})$ 级别的。

显然这个询问的答案要么是最大覆盖块的答案，要么是没有被覆盖的元素。把这些元素暴力插入到最大覆盖块中，记录出现次数最多的，二者较大值就是答案。显然时间复杂度是 $O(N*S^2+Q*\frac{N}{S})$ 的，假设 N 和 Q 同阶的话，取 $S=N^{\frac{1}{3}}$ 就能做到在线 $O(N^{\frac{5}{3}})-O(N^{\frac{2}{3}})$。

这个复杂度虽然不错，但并不是很让人满意。但从上面的做法中，我们可以得到一些启发。我们发现，如果区间 $[L,R]$ 能覆盖区间 $[L',R']$，那么区间 $[L,R]$ 的众数或者就是区间 $[L',R']$ 的众数，或者是位于 $[L',R']$ 之外但位于 $[L,R]$ 内的某个数。如果我们能预处理一些 $[L',R']$ 的答案，使得任何一个询问 $[L,R]$ 都能找到一个已经预处理过的区间 $[L',R']$，使得 $[L,R]$ 包含 $[L',R']$ 且没有被 $[L',R']$ 包含的元素不多，那么我们只需枚举这些没被包含的元素并更新答案就行了。于是，我们依然考虑把序列分成 S 块，然后记录下所有“左端点是某个块起点，右端点任意”的区间的答案，因为不用记录每个数出现次数，这一步是 $O(N*S)$ 的。然后对于每个询问，找到它能覆盖的最大的块，显然没有被覆盖的元素必然是左端连续的一些元素，不超过 N/S 个。然后问题转化成了询问一个数在一段区间内出现了多少次。这个可以把每个数出现的位置放在 vector 或 set 里，然后转化成求某个数是第几大，于是每次回答 $O(\log N)$，每个查询要查询 $\frac{N}{S}$ 次，查询的复杂度是 $O(\frac{N}{S}\log N)$。取 $S=\sqrt{N}$, 时间复杂度为 $O(N\sqrt{N})-O(\sqrt{N}\log N)$。

还有没有可改进的空间呢？答案是有的。我们可以想办法去掉那个 $\log N$。想像一下查询一个数在一段区间内出现了多少次的暴力方法，显然可以用 $F[i,j]$ 表示 j 在区间 $[1,i]$ 出现了多少次，于是有

$$F[i,j]=F[i-1,j] \qquad 当 j\neq a[i]$$
$$F[i,j]=F[i-1,j]+1 \qquad 当 j=a[i]$$

但这样是 $O(n^2)$ 的。我们考虑 $F[i][\ \ ]$ 到 $F[i+1][\ \ ]$ 的转移实际就是修改了一个数，其他的数都是一样的。于是问题进一步转化为：维护一个序列，要求支持：修改一个数，查询第 X 次修改完成时某个数的值（第 X 次修改完成后就的序列就是 $F[X]$）。这种要求能“撤销”做下去的操作的题目一般有两种通用方法。一种方法是离线回答，在这里当然就会导致多出一个 $\log N$ 从而变回成了前一种方法了，另一种方法就是函数式数据结构。（函数式数据结构可以参考范浩强在 WC2012 的讲稿。）

考虑用函数式数据结构维护这个序列。显然修改后 $N-1$ 个元素都是可以重用的，真正变化的只有 1 个元素。我们发现如果以序号为关键字建函数式完全二叉树维护的话，查询和修改依然是 $O(\log n)$，不行，因为我们必须做到查询 $O(1)$。于是想到一个更“暴力”的方法，直接分块然后函数式块状结构维护！假设分成 T 块，那么修改的时候 $T-1$ 块都可以重用，包含修改元素的那一块必须重新建，于是时间复杂度 $O(\frac{N}{T}+T)$，于是取 $T=\sqrt{N}$ 就做到了 $O(\sqrt{N})$ 每次修改，查询时候因为可以直接找到第 X 次修改前的那个块状结构，于是做到了每次 $O(1)$，而每次查询要查询 $O(\frac{N}{S})$ 次。于是取 $S=\sqrt{N}$，总时间复杂度 $O(N\sqrt{N})-O(\sqrt{N})$。

有没有更好的方法呢？答案是有的！我们将问题转化为“查询一个数在一个区间内出现了多少次”时，舍弃了这个问题的一些特殊性！这个特殊性就是，询问区间的两端点都必然是我们分块的端点！但这个特殊性质能怎么被利用呢？答案是预处理询问！我们预处理所有

左端点是 1，右端点是分块的某个端点，查询值是 X 的询问，显然这些询问只有 $O(N\sqrt{N})$ 种，可以在 $O(N\sqrt{N})$ 内处理出来。然后，每个询问都可以用部分和的方式由两个预处理过得询问相减而得到答案！于是查询就做到了 $O(1)$ 每次，时间复杂度依然是 $O(N\sqrt{N})-O(\sqrt{N})$，但是优势在于它没有用到任何高级数据结构，而且常数比前一个方法小很多，且十分易于实现。

这个看似很棘手的问题被我们以 $O(N\sqrt{N})-O(\sqrt{N})$ 的复杂度非常漂亮的解决了。

【例 8】BZOJ 2741 L

给定一个序列，要求在线支持查询区间最大部分 xor 和操作。也就是对每个询问 $[L,R]$, 要找到 $L\le i\le j\le R$，使得 $a[i]$ xor $a[i+1]$ xor ... xor $a[j]$ 的值最大，返回这个最大值。.

首先我们考虑这道题的朴素算法。我们令 $s[i]=a[1]$ xor $a[2]$ xor ... xor $a[i]$, 每个询问 $[L,R]$ 就相当于找到 $L-1\le i<j\le R$ 使得 $s[i]$ xor $s[j]$ 最大。这个问题有一个很经典的 Trie 树做法，可以做到 $O(32N)$，这里不再赘述。

我们需要找到一个时间复杂度更优秀的做法。借鉴上一题的经验，我们发现，假如对询问 $[L,R]$ （注意：这里的询问意思已经变了，这里的询问 $[L,R]$ 表示要找 $L\le i<j\le R$ 使得 $s[i]$ xor $s[j]$ 最大）我们能找到一个已经被预处理过的区间 $[L',R'](L\le L'\le R'\le R)$，那么最优答案有 2 种情况：$i$,$j$ 都在 $[L',R']$ 中；i,j 至少有一个不在 $[L',R']$ 中。前一种情况已经被预处理出来，我们只需解决后一种情况。像上一题一样，我们将序列分成 S 块，然后记录下所有“左端点是某个块起点，右端点任意”的区间的答案，这一步利用 Trie 可以做到 $O(32*N*S)$。这样对于每个询问，找到它能覆盖的最大的块，没有被覆盖的元素必然是左端连续的一些元素，不超过 $\frac{N}{S}$ 个。于是问题转化为，每次给定一个数 X，求在区间 $[L,R]$ 里选另一个数 $s[i]$，使得 X xor $s[i]$ 最大。再次借鉴上一题思路，我们发现这又是一个经典的“可撤销”数据结构题，可以通过函数式 Trie 解决。利用函数式 Trie，我们可以直接找到区间 $[1,R]$ 对应的 Trie 树，而 L 的限制很好解决，只需额外记录某个函数式 Trie 节点对应的前缀最晚在哪里出现即可。于是一次询问的复杂度 $O(32)$，每次查询需要询问 $O(\frac{N}{S})$ 次，每次查询复杂度 $O(32*\frac{N}{S})$. 取 $S=\sqrt{N}$, 我们得到了一个 $O(32N\sqrt{N})-O(32\sqrt{N})$ 的做法。

例 7 和例 8 给出了很多种算法，这些做法的可扩展性都非常的强，其中蕴含的思想和技巧对很多棘手的序列上的数据结构题都非常实用。我写这篇文章的目的并不是具体的介绍某几题数据结构题，而是尝试着给出一些具有启发性的通用解决思想。

2 数据结构在字符串中的应用——指纹算法及其他

其实字符串本质就是一个序列。区别之处在于，两个字符串可以比较大小。也就是，定义了两个序列的比较函数。因此，如果我们能用很低的复杂度实现序列的比较函数，我们就能把很多字符串题目剥去外壳，转化为序列上的题目。

2.1 几个约定

- 为方便叙述，下文出现的“序列”均指“字符串”。

- 约定序列 a 的长度为 $Len(a)$，$a[i]$ 表示序列的第 i 个元素，下标从 1 开始。
- 约定区间 $a[L,R]$ 表示序列 a 的第 L 个元素到第 R 个元素组成的子序列。

2.2 从一个经典问题引入——两个序列的比较函数

给定两个序列 a,b。我们如下定义两个序列的比较函数。

- 首先我们找到最大的 k 使得对于任意 $1 \le i \le k$ 均有 $a[i] = b[i]$。
- 此时，如果 $k = Len(a) = Len(b)$，则 $a = b$
- 否则如果 $k = Len(a)$ 则 $a < b$，如果 $k = Len(b)$ 则 $a > b$
- 否则如果 $a[k+1] < b[k+1]$ 则 $a < b$，否则 $a > b$

我们定义 $LCP(a,b)$ 为第一步中找到的最大 k 值. 显然只要能快速求出 $LCP(a,b)$，剩下的步骤就是 $O(1)$ 的了。我们发现 k 值满足二分性质。于是我们二分 k，问题转化成了快速判定两个序列是否全等。这可以通过字典序 hash（指纹算法）在 $O(N)$ 预处理后 $O(1)$ 回答。

指纹算法是一个强力的工具，利用它我们做到了快速的序列比较 $O(N) - O(\log N)$，因此能把很多字符串上的问题转化为序列上的问题。本部分的大部分内容都将围绕它展开，接下来我们将看到它更多的应用。

2.3 指纹算法的应用

指纹算法的应用范围十分广大，我们接下来将详细介绍。

2.3.1 字符串匹配

【例 9】经典问题

给定母串 S 和待匹配串 T，要求找到所有 k，使得 $S[k, k + Len(T) - 1] = T$。

因为我们在 $O(N)$ 预处理后可以 $O(1)$ 比较两个序列是否相等，因此我们直接枚举 k 后 $O(1)$ 判断即可。时间复杂度 $O(N)$。

2.3.2 LCP 查询

【例 10】JSOI2008 火星人

给定序列 a，要求支持：插入或修改一个元素，查询 $LCP(a[X, Len(a)], a[Y, Len(a)])$。

显然原版指纹算法修改元素和插入元素只能 $O(N)$ 重头计算，不可接受。不过没关系，用 Splay 维护 hash 值就好了。可以做到 $O(\log^2 N)$ 每次操作。

【例 11】BZOJ 增强型 LCP

给定序列 a，要求支持：插入或修改一个元素，查询 $LCP(a[X, Len(a)], a[Y, Len(a)])$。其中绝大多数操作 (98%) 都是查询。

这道题如果用 Splay 维护，会因为比较巨大的常数而超时。但我们发现这题特殊性在于绝大多数操作都是查询，因此我们考虑通过提高修改的代价来降低查询的代价，从而提高速

度。于是我们想到用块状链表维护 hash 值。显然通过降低分块数目 S，我们可以在降低修改的速度的代价下提高查询速度。选择合适的 S 就可以 AC 此题。

2.3.3 与回文串有关的题目

【例 12】经典问题

定义 $Rev(S)$ 表示序列 S 的逆序。定义序列 S 是“回文的”当且仅当 $S = Rev(S)$ 成立。

给定序列 S，要求找到尽量长的一段子序列 $S[L, R]$ 使得 $S[L, R]$ 是回文的。

回文串的长度可能是奇数也可能是偶数，但这两种回文串本质是一样的，因此我们这里只考虑长度为奇数的回文串。我们枚举回文串的中心。考虑对于给定的中心位置 X，以它为中心的最长回文子串的长度就是 $LCP(Rev(S[1, X\ 1]), S[X + 1, Len(s)])$。于是我们枚举这个中心位置，就可以在 $O(\log N)$ 时间内算出以它为中心的最长回文子串。于是时间复杂度 $O(N \log N)$。为后文方便起见，我们定义 $Extend(S, X) = LCP(Rev(S[1, X\ 1]), S[X + 1, Len(s)])$, 也就是以 X 为中心最长能向两边延伸的回文子串长度。

【例 13】Codeforces 30E - 改

为突出主题，略去了本题一些比较简单但琐碎的部分，只突出关键部分。给定序列 S，每次查询区间 $S[L, R]$ 内最长的长度为奇数的回文子串的长度 X。

首先我们预处理出 $Extend(S)$。但是，序列 $Extend(S)$ 的区间 $[L, R]$ 的最大值显然不一定是答案，因为有可能回文子串有的部分已经超出了区间 $[L, R]$。于是我们考虑二分答案长度的一半 $Y = \left[\frac{X}{2}\right]$，进而转化问题为判定序列 $Extend(S)$ 在区间 $[L + Y, R\ Y]$ 的最大值是否大于等于 Y。我们用线段树维护 $Extend(S)$ 的 RMQ, 于是可以做到 $O(\log N)$ 每次判定，于是时间复杂度 $O(\log^2 N)$ 每次查询。

【例 14】SHOI 双回文子串

定义序列 S 是“双回文”的，当且仅当 $Len(S) = 2X$ (X 为整数), 且 $S[1, X] = S[X + 1, 2X]$，且 $S[1, X]$ 是回文的。给定序列 S，求其尽量长的一段子序列 $S[L, R]$ 使得 $S[L, R]$ 是双回文的。

根据定义，容易发现一个双回文串本身也是一个长度为偶数的回文串。于是我们考虑枚举双回文串的中心 X，并求出以其为中心的最长回文子串，假设其长度为 L，那么我们只需在 $[X, X + L - 1]$ 里找到最长的左端点在 X 的回文子串。于是我们只需求最大的 R 属于 $[X, X + L - 1]$，使得 $S[X, R]$ 是回文子串。为简单起见，我们只考虑长度为奇数的回文子串。容易发现，$S[X, R]$ 的中心显然不能超过区间 $[X, X + L - 1]$ 的一半。于是假如中心 Y 不超过上述区间的一半，那么如果 Y 是可行的，就只需满足 $Extend(S, Y) \geq Y - X$，即 $Y - Extend(S, Y) \leq X$。定义 $D[i] = i - Extend(S, i)$, 那么我们需要维护序列 D，使得支持查询一个区间内最大的 i 使得 $D[i] \leq X$。这个问题可以通过用线段树维护 $D[i]$ 的 RMQ，查询时在线段树上二分来做到 $O(\log N)$. 于是总时间复杂度 $O(N \log N)$。

2.3.4 构建后缀数组

由后缀数组定义可知，只需对后缀进行排序，就得到了后缀数组。而 $Height$ 数组就是排名相邻的两个后缀的最长公共前缀。由前文，因为我们已经实现了 $O(\log N)$ 的字符串比较函数，我们直接使用任何一个时间复杂度是 $O(N \log N)$ 的排序算法直接对后缀排序，就可以在 $O(N \log^2 N)$ 时间内构建后缀数组。$Height$ 数组直接使用我们已经实现的 LCP 函数就可以在 $O(N \log N)$ 内构建出来。快速排序在 Pascal/C++ 内都提供有现成的代码可以直接使用，于是我们只需手动实现 LCP 函数就可直接构建后缀数组与 $Height$ 数组，代码极其简洁方便。利用指纹算法，我们用一个非常简单直观的方法构建了后缀数组。

这个方法对大多数后缀数组题目都足够使用了，虽然我们可以做到更快。我们可以用倍增算法 + 基数排序做到 $O(N \log N)$，甚至存在 $O(N)$ 构建后缀数组的 DC3 算法，而 $Height$ 数组也可以利用特殊性质在 $O(N)$ 内计算出来。但这些算法显然没有上文所述算法易于实现，而且因为常数原因其速度优势并不是很大，限于篇幅这里不再介绍，有兴趣者可以参考罗穗骞《后缀数组——处理字符串的有力工具》。

2.3.5 扩展后缀数组

因为指纹算法强大的可扩展性，结合了指纹算法的后缀数组还能支持一些普通后缀数组不能支持的操作。

【例 15】FZU 1916 - 改

为突出主题，去掉了题目中一些简单但琐碎的部分，只保留关键部分，并对题目做了微小改动。

给定一个序列 S，要求在线支持：往序列前端插入一个字符，查询序列 S 有多少个不同的子串。

首先，可以证明，一个序列相同的子串数目就是其后缀数组的 $Height$ 数组所有元素之和。于是我们需要支持在前端插入元素，并动态维护后缀数组的 $Height$ 之和。

显然，倍增算法或 DC3 构建后缀数组不能支持快速插入新字符操作。我们考虑回到定义，所谓后缀数组，其实就是排序后的后缀！而在前端插入新元素显然不会改变原本存在的后缀的顺序，而只会新增加一个后缀。于是我们考虑用平衡树维护排序后的后缀，利用快速比较做到 $O(\log N)$ 每次比较。那么在前端插入新字符就等价于往平衡树中插入一个元素。于是时间复杂度 $O(\log^2 N)$ 每次操作，而插入一个元素后只会修改 2 个 $Height$ 值，直接更新即可。我们再次见识到了指纹算法的各种强大的应用。

2.4 其他字符串后缀结构

与字符串后缀有关的数据结构很多，除了后缀数组，还有后缀自动机、后缀树等。但这不是本文的重点，故不做具体介绍。更多关于后缀自动机的内容建议参考陈立杰在 WC2012 的讲稿，后缀树建议参考 3xian 的文章。这里只介绍一道很有趣的题目。

【例 16】CTSC2010 珠宝商

给定一棵 N 个结点的树，树的每个结点上都有一个字符。定义 $Path(X,Y)$ 等于从结点 X 到结点 Y 的最短路径所经过的结点上的字符顺次连接起来形成的字符串。给定长度为 M 的母串 S，定义 $Occur(X,Y)$ 为字符串 $Path(X,Y)$ 在母串 S 中出现的次数。求 $sigma\{Occur(X,Y)\}$ 其中$1 \le X,Y \le N$。

这道题咋一看令人无从下手。我们不妨先分析几种朴素算法。

我们需要解决一个字符串 S 在某个长母串里出现了多少次这个问题。这个问题可以通过对母串建后缀自动机后走一遍，在 $O(Len(S))$ 时间内回答。于是暴力枚举根节点，DFS 整棵树，因为每次往下走都只会加一个字符，只需 $O(1)$，于是可以做到 $O(N^2)$ 其中 N 为要计算的树的结点数目。

我们考虑另一种朴素算法。显然任意两个 X,Y 都必然有一个 lca。我们考虑所有 lca 是某个结点 Z 的 (X,Y) 的 $Occur$ 次数之和。这必然可以表示为两条从 Z 结点走下去的路径（要求 lca 必须是结点 Z）。于是对母串正序反序建两颗后缀树，我们可以做到 $O(M)$ 统计。时间复杂度 $O(M)$ 每个 Z 结点。

我们对这颗树进行点分治。找重心作为根结点。于是我们有两种选择，第一种是使用朴素算法 1，在 $O(Size^2)$ 时间内算出整棵子树的所有询问之和。另一种是使用朴素算法 2，用 $O(M)$ 时间算出所有过根结点路径的和，然后对每棵子树递归处理。

朴素算法 2 的意义在于用 $O(M)$ 时间把一个大任务分解为若干小任务，而朴素算法 2 在结点数很少的情况下显然不如时间复杂度与子树大小有关的朴素算法 1 划算。于是我们产生了一个大致方向：化整为零，逐个击破。

我们产生了一个想法，如果树结点数比较多，就用第二种方法 $O(M)$ 算出过根结点的答案，然后分治每个子树，化整为零。否则直接第一种方法 $O(N^2)$ 算出整棵树的答案，逐个击破。

因为我们选择重心作为根结点，所以每次使用朴素算法 2 必然至少能把当前任务分解成 2 个规模不超过当前任务一半的子任务，而且显然分解出的子任务越多，对逐个击破越有利，所以最坏情况就是分解出 2 个规模为当前任务一半的子任务。假设我们已经得到 K 个大小为 $\frac{N}{K}$ 的子任务，我们对 K 个子树都进行一次化整为零，就可以得到 $2K$ 个大小为 $\frac{N}{2K}$ 的子任务。于是利用循环不变式可以证明，分解出 S 个大小为 $\frac{N}{S}$ 的子任务最多只需执行 $O(S)$ 次化整为零。而对这 $\frac{N}{S}$ 个子任务逐个击破时间复杂度为 $O\left(\left(\frac{N}{S}\right)^2\right) * O(S) = O(\frac{N^2}{S})$。于是取 $S=\sqrt{N}$，化整为零的复杂度是 $O(M\sqrt{N})$, 逐个击破的复杂度是 $O(N\sqrt{N})$, 总时间复杂度是 $O((N+M)\sqrt{N})$。

这个做法的时间复杂度比官方题解的做法要好（官方题解做法是 $O((N+M)\sqrt{N}\log N)$），而且常数比官方题解的常数要小很多，且易于思考和实现。这个做法的关键思想在于充分发挥两个朴素算法各自的优点，通过分块均衡，最终得到了一个复杂度非常优秀的做法。

3 数据结构在树形结构中的应用—— DFS 序与树链剖分

我们这里只讨论无根树。所谓无根树，就是一个无向无环连通图。

与往常一样，我们希望把与树有关的数据结构题归约转化到序列上去。常见的转化方法有 DFS 序和树链剖分。

3.1 几个约定

- 为方便叙述，我们约定无根树的结点标号为 $1 \sim N$。
- 约定无根树的根为 1。
- 约定 $parent(X)$ 表示结点 X 的父结点。
- 约定 $depth(X)$ 表示结点 X 的深度（约定根结点深度为 1）。
- 约定 $lca(X,Y)$ 表示结点 X 和结点 Y 的最近公共祖先。

3.2 DFS 序的性质与应用

所谓 DFS 序，就是 DFS 整棵树依次访问到的结点组成的序列。

DFS 序有一个很强的性质：一棵子树的所有结点在 DFS 序内是连续一段。利用这个性质我们可以解决很多问题。

【例 17】7 个经典问题

给定一棵树，点上有权值。

1. 要求支持：对某个点 X 权值加上一个数 W，查询某个子树 X 里所有点权值和。

解：因为子树 X 里所有结点在 DFS 序中是连续一段，所以我们只需维护一个序列，支持：修改一个数，查询一段数的和。显然树状数组可以完成。

2. 要求支持：对 X 到 Y 最短路上所有点权值加上一个数 W，查询某个点的权值

解：首先明确，X 到 Y 的最短路是由 X 到 $lca(X,Y)$ 的路径加上 Y 到 $lca(X,Y)$ 的路径组成的。于是修改操作可以等价的转化为：X 到 1 的路径所有点权加 X，Y 到 1 的路径所有点权加 X，$lca(X,Y)$ 到 1 的路径所有点权减 X，$parent(lca(X,Y))$ 到 1 的路径所有点权减 X。于是我们只需要支持修改从某个点到根的路径上所有点的权值和查询一个点的权值。考虑 X 某个修改对查询 Y 的贡献。显然只有当 X 在 Y 的子树里时候才会产生贡献，且贡献为 W。于是进一步转化问题：我们需要修改一个点的权值，查询某个子树里所有点权值之和。于是通过计算贡献的思想，这个问题被转化为问题 1。

3. 要求支持：对 X 到 Y 最短路上所有点权值加上一个数 W，查询子树 X 内所有点的权值之和。

解：与上一题类似，首先把修改等价转化为修改 X 到 1 的路径上所有点的权值。然后同样考虑修改 X 对查询 Y 的贡献。显然当 X 在 Y 的子树内时候才会产生贡献，且贡献为 $W(X)*(depth[X]-depth[Y]+1)$，分离变量得 $W(X)*(depth[X]+1)-W(X)*depth[Y]$。前一项与 Y 无关，于是转化为问题 2，可以用一个树状数组维护，后一项里 $W(X)$ 与 Y 无关，也转化为问题 2，用一个树状数组维护，有了 $W(X)*(depth[X]+1)$ 和 $W(X)$ 我们就可以计算出询问的答案。

4. 要求支持：对某个点 X 权值加上一个数 W，查询 X 到 Y 路径上所有点权值和。

解：这题有一个经典做法，首先把询问等价转化为求 X 到根的点权之和。然后修改时假设点 X 对应子树是 $[L,R]$，那么 $D[L]$ 加 W, $D[R+1]$ 减 W，X 到 1 路径上点的权值和就是 $D[1]$ 到 $D[L]$ 的和。于是用树状数组维护。

5. 要求支持：对子树 X 里所有节点加上一个权值 W，查询某个点的权值。

解：同样，考虑某个修改 X 对查询 Y 的贡献。显然，当 Y 在 X 的子树里时 X 对 Y 才有贡献，且贡献就是 W。于是转化为修改一个点权，查询某个点到根的路径的权值，于是转化为问题 4。还有一种做法是直接使用线段树维护 DFS 序，变成区间加减和单点查询，显然是经典问题。

6. 要求支持：对子树 X 里所有结点加上一个权值 W，查询某个子树 X 里所有结点权值之和。

解：直接使用线段树维护 DFS 序，转化为区间加减和区间求和，显然是经典问题。

7. 要求支持：对子树 X 里所有结点加上一个权值 W，查询 X 到 Y 最短路上所有结点的权值之和。

解：照例把最短路转化为 X 到根结点路径上所有权值之和。考虑修改 X 对查询 Y 的贡献，显然当 Y 在 X 的子树里时才有贡献，贡献为 $(X)*(depth[X]-depth[Y]+1)$. 分离变量得 $W(X)*(depth[X]+1)-W(X)*depth[Y]$。于是照例分成两部分处理，每一部分都相当于修改一个点权，查询某个点到根路径上权值和。于是转化为问题 4.

【例 18】High-Level Ancients

给定一棵树，点上有权值，定义 $Edit(X,W)$ 表示对每个 X 子树内的结点 Y，假设 Y 与 X 距离为 K，那么 Y 的权值要加上 $W+K$。定义 $Query(X)$ 表示查询 X 子树内所有点权值和。要求支持这两个操作。

这个任务看似相当复杂。我们考虑逐步分解任务。首先把权值 W 移除。$Edit(X,W)$ 可以分解为 X 子树内每个结点权值加 $W-1$ 然后执行 $Edit(X,1)$。然后我们发现 $Edit(X,1)$ 具有类似递归的性质。我们考虑把它分解为对每个 X 子树内的点 Y，$D[Y]$ 加上 $Size[Y]$，其中 $Size[Y]$ 表示子树 Y 的大小。这样如果修改 X 在查询 Y 的子树内，那么所有 $D[Z]$ 其中 Z 属于 X 的子树的和就是正确的 X 对 Y 的贡献了。这个显然可以用线段树维护。但还有一种情况就是 X 是 Y 的祖先，这时候对 Y 除了上述的 $D[Z]$ 的贡献，还有额外的 $(depth[X]-depth[Y])*Size[Y]$ 的贡献。对这个式子分离变量，我们发现转化为了例 17 的问题 4。于是这个复杂的任务被我们一步步分解，直到被解决。

3.3 树链剖分的应用

所谓树链剖分，又称轻重边路径剖分。定义某个结点的连往其 $Size$ 最大的孩子（如有多个任选其一）的边为重边，其余边为轻边。首尾相连的重边组成的尽量长的链称为重链。于是，一棵树被我们剖分成了若干重链和若干轻边。可以证明，任意两点之间最短路最多经过 $O(\log N)$ 条重链和 $O(\log N)$ 条轻边。于是我们只需快速处理经过重链的一部分的情况，而这就将树上的问题转化为了链上的问题。

【例 19】SPOJ QTREE

给定一棵树，边上有权值。要求支持：修改一条边的权值，查询结点 X 到结点 Y 最短路径上最大的边权。

对这颗树进行树链剖分之后我们只需支持查询一条重链的一个区间的最大值和修改一条重链的一个元素。于是用线段树维护重链的 RMQ 即可做到 $O(\log N)$。因为查询需要至多经过 $O(\log N)$ 条重链，而修改只会修改至多 1 条重链，于是总时间复杂度 $O(\log^2 N)$ 每次查询，$O(\log N)$ 每次修改。

【例 20】SPOJ GSS7

给定一棵树，点上有权值。要求支持：将结点 X 到结点 Y 最短路径上所有点权值设置为 W，查询结点 X 到结点 Y 最短路径上结点依次组成的序列的最大部分和。

对这颗树进行树链剖分之后我们只需支持查询一条重链的一个区间的最大部分和和将一条重链的区间整体赋值操作。于是我们把问题转化为例 2，可以通过线段树维护。于是时间复杂度 $O(\log^2 N)$ 每次操作。

【例 21】SPOJ QTREE4

给定一棵树，边上有权值，结点有黑白两色之分。初始时所有节点都是白色的。要求支持：对一个结点反色，查询距离最远的两个白点的距离。

首先对这颗树进行树链剖分。我们假设最远的两个白点分别是 X 和 Y，令 $Z = lca(X, Y)$，则 Z 必然属于某个重链 L。于是我们考虑，如果对每个重链，我们都能维护最高点在这个重链上的所有路径的最大值，那么我们就解决了此题。我们发现这个“最高点在重链上”的路径必然可以拆分成 3 段：重链上一段，重链上那一段的左端点沿某条轻边往下延伸到某个白点最长距离，重链上那一段的右端点沿某条轻边往下延伸到某个白点的最长距离。特殊的，如果“重链上的一段”只有一个结点，那么只能是沿某两条轻边往下延伸到某两个白点距离之和，也就是最大值和次大值。我们发现“重链上的一段”这个东西和部分最大和很相似，事实上我们也确实可以用线段树维护部分和序列的方法来维护的。而“往下沿轻边延伸的最长和次长距离”怎么维护呢？因为一个点连出的轻边是 $O(N)$ 数目的，我们不能一条一条暴力检索轻边。所以我们对重链上每个结点用一个堆来维护其所有连出的轻边到达的顶点往下延伸的最大值和次大值，而这恰好就是一条更深处的重链所维护的东西。于是我们成功地维护了这些域。查询就可以 $O(1)$ 实现了，而修改时，因为每次修改最多只会修改 $O(\log N)$ 条重链和 $O(\log N)$ 个堆，所以是 $O(\log^2 N)$ 的。

4 总结与感谢

感谢贾志鹏学长给我提供的大量帮助。

感谢通过网络或现实与我讨论、给我提供了大量帮助的诸多同学们，没有你们，也就没有我的这篇文章。谢谢你们!