数据结构漫谈

南京外国语学校 许昊然

数据结构在序列中的应用

RMQ

给定一个序列
查询一个区间里最大的数

RMQ

线段树
每个结点维护一个区间
预处理出这个区间的最大值
查询时把查询分成很多小区间的并
利用预处理的结果，合并出查询区间的答案

任务分解

把查询区间分解成一些小区间的并
分解出的小区间的数量决定每次查询复杂度

预处理

预处理每个小区间的答案
预处理的小区间的数量决定预处理复杂度

合并答案

最终合并出所需的结果
能合并出答案的树形结构，才是有意义的数据结构

两种任务分解方法

树形结构？
分块结构？

树形结构

线段树
Splay

树形结构为什么能保证时间复杂度

每个询问被分解为 O(log N) 个已经被预处理的小区间
这些小区间能被快速的合并起来，得到最终的查询区间
有修改时两个子区间能快速合并起来，更新父区间的答案
懒标记能快速应用到一个区间
两个懒标记能快速合并

回忆一下树形结构的大致框架？

一个序列能用树形结构维护，需要满足什么？

使用树形结构的要求

1. 支持快速合并两个序列的结果
2. 如果有懒标记，支持对一个序列快速应用懒标记和快速合并两个懒标记

例題

GSS1

给定一个序列
要求支持区间最大部分和

GSS1 solution

考虑一个询问 [L,R] 的答案
我们取 L<=Mid<=R
那么 [L,R] 的答案必然是
区间 [L,Mid] 的答案
区间 [Mid+1,R] 的答案
和跨越 Mid 的区间答案这三者的最大值
而跨越 Mid 的区间最大值必然是
[L,Mid] 的最大后缀和加上 [Mid+1,R] 的最大前缀和

GSS1 solution

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 Lmax, 后缀最大和 Rmax, 最大部分和和 Mmax 和整段区间的和 S.

GSS1 solution

关键：快速合并两个区间的答案

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

GSS3

给定一个序列
要求支持区间整体赋值
和区间最大部分和

GSS3 solution

整体赋值？
懒标记！

GSS3 solution

树形结构里，懒标记要求支持什么？
快速应用懒标记和快速合并两个懒标记！

GSS3 solution

快速应用懒标记：
可以直接根据整体赋值的值计算要维护的 4 个域
快速合并懒标记：
后到的区间赋值直接覆盖原来的区间赋值

NOI2005 维护数列

给定一个序列 要求支持
插入一个序列
删除一段序列
区间整体赋值
区间翻转
查询区间最大部分和。

维护数列 solution

插入删除？ Splay ！
区间整体赋值：懒标记前文已经讨论过
区间最大部分和：维护的域也已经讨论过

维护数列 solution

区间翻转？ 翻转标记！

维护数列 solution

对一个序列应用翻转标记：交换 Lmax 和 Rmax，交换其左右子树并对左右子树应用翻转标记！
翻转标记的合并：两个翻转等于没转

维护数列 solution

标记的共存？
翻转标记与赋值标记的先来后到关系不会影响最终结果！
可以共存！

啥？ 不能快速合并答案怎么办？

给你两个排过序的区间，
你能快速合并出大区间排序后的结果吗？？

啥？ 懒标记不能合并怎么办？

给区间里所有属性为 X 的元素加 1，
不同属性种类一大堆，
你准备用多少种懒标记来记录？？

分块结构！

分块结构

什么是分块结构？
把一个序列等分成 S 块
对每一块维护东西

分块结构

一般而言
每一个询问
分解成 O(S) 个完整的块和 O(N/S) 个零散元素

分块结构的优势

不是树形结构，没有父子关系！
因为没有父子关系
初始化和修改时没有合并两个区间答案的操作！
因为没有父子关系
懒标记也不必下放
所以大多数情况下也不需要支持合并两个懒标记！

diff 分块结构 树形结构

共同点：
要求支持快速应用懒标记
不同点：
分块结构在初始化和修改时不需要合并区间答案
分块结构一般不需要支持快速合并懒标记

啥？ 查询时还是要合并答案？

—— 有时候，真的不需要的哦 ~

例題

经典问题 区间加减 + 区间 K 大

给定一个序列 要求支持
区间加减
区间 K 大

区间加减 + 区间 K 大

—— 啥？ 你想用树形结构？怎么支持 K 大？
—— 每个结点维护其对应区间排好序后的样子！
查询时二分答案，然后线段树每个区间里二分查找！
—— 恩。不错。区间加减呢？
—— 懒标记！ 不影响元素相对次序！
—— 恩。不错。然后修改了子结点，父节点要更新，
怎么合并答案呢？
——…… 囧

区间加减 + 区间 K 大 solution

块状结构！ 维护每个块排序后的样子！
懒标记不影响元素相对位置
没有父子关系，建立和修改都不需要合并答案！
查询时二分答案，然后在每个块里查找其 rank，
也不需要合并答案！
问题解决。

区间加减 + 区间 K 大 solution

时间复杂度？ 假设分成 S 块

修改：$O(\frac{N}{S}\log\frac{N}{S})$

查询：$O(S\log S\log N)$

区间加减 + 区间 K 大 solution

为了让修改和查询耗费的时间差不多，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。

SPOJ UNTITLE1

给定一个序列 要求在线支持
区间加减
区间最大前缀和

SPOJ UNTITLE1

—— 啥？ 你还想用树形结构？
—— 恩。
—— 说说怎么维护？
—— 还是那 4 个域！
—— 然后修改呢？
—— 像例 2 那么改？
—— 错的！
区间加减后很多本来不应该选的负数都变成正数了
这时候新的 Lmax 和原来的 Lmax 无任何关系
不能快速应用懒标记！

线段树死于不能快速应用懒标记

UNTITLE1 solution

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。

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 $(i, s[i] + X * i)$

而询问操作，就是求最大的 Y 坐标。

区间加减操作，很类似于对坐标系的旋转！

猜测：最优解一定在点集的上凸壳上！

UNTITLE1 solution

严格的证明：

如 j 不在上凸壳上，那么必然存在 i<j<k, 使得

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

UNTITLE1 solution

—— 啥？ 你又想到新的做法了？
—— 恩！ 用线段树维护每个区间的凸壳！
—— 恩。然后修改呢？
—— 修改操作不改变凸壳，直接懒标记！
—— 恩，然后子结点修改了，要更新父结点，
—— 怎么快速合并答案呢？
——..... 又犯 2 了囧

线段树再次死于不能快速合并答案

UNTITLE1 solution

再次想到块状结构
对每一块维护其上凸壳
修改：整体修改不改变凸壳，直接懒标记
部分修改直接暴力重建凸壳
查询：对零散的元素暴力
对完整的块在凸壳上三分答案找最优值
时间复杂度 O(N^0.5*log N)

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。

Power Array Solution

——…… 无从下手……

—— 离线算法？

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 R0，我们要回答所有形如 [L,R0] 的询问。

Power Array Solution

—— 还是无从下手啊……
—— 进一步提示：计算贡献？

Power Array Solution

令 D[i] 表示数 a[i] 在区间 [i,R0] 贡献的代价
如果 a[i] 出现了 X 次，那么 D[i]=(2X-1)*a[i].
查询 F[L,R0] 等价查询区间 [L,R0] 里所有 D[i] 的和
而从 R0 转移到 R0+1 等价往右端加入一个新数
等价对所有的满足 a[i]=a[R0+1] 的 i 值
把 D[i] 加上 2*a[i].

Power Array Solution

这样看是不是可做很多了？
不过注意这题修改比较特殊，
如果用懒标记的话，有多达 O(N) 种不同的懒标记
我们不能合并两个不同种类的懒标记
于是如果用树形结构维护，下传标记时间复杂度将达 O(N)
不可接受

线段树死于不能快速合并两个懒标记

Power Array Solution

懒标记的应用可以通过维护前一个数出现在哪里
和这一块中某个数出现了多少次做到。
块状结构没有父子关系，不需要支持合并懒标记！
于是对块的整体修改被支持了
对于零散的元素，可以直接绕开懒标记，
利用定义进行计算
于是查询也被支持了

你这是投机取巧！

—— 你到头来还是不能快速合并答案！
　　所以你查询时候都用各种技巧规避开了合并答案！
—— 废话你拿着小刀对手拿着 B51 你难道冲上去硬拼？

而且，其实查询时不得不合并答案的题目，也不一定不可做哦 ~

你拿着小刀，对手拿着 B51 ，你还得去拆 C4 ，怎么办？

例子

BZOJ 蒲公英

给定一个序列 要求在线回答
区间众数，如有多个输出最小的那个

蒲公英

—— 这货是什么玩意…… 完全没有办法维护啊……
—— 恩。其实你可以先给出一个离线算法。

蒲公英 离线算法

—— 离线我会做！ 在张昆玮的论文《统计的力量——线段树详细教程 》中提到过！
—— 恩。确实。

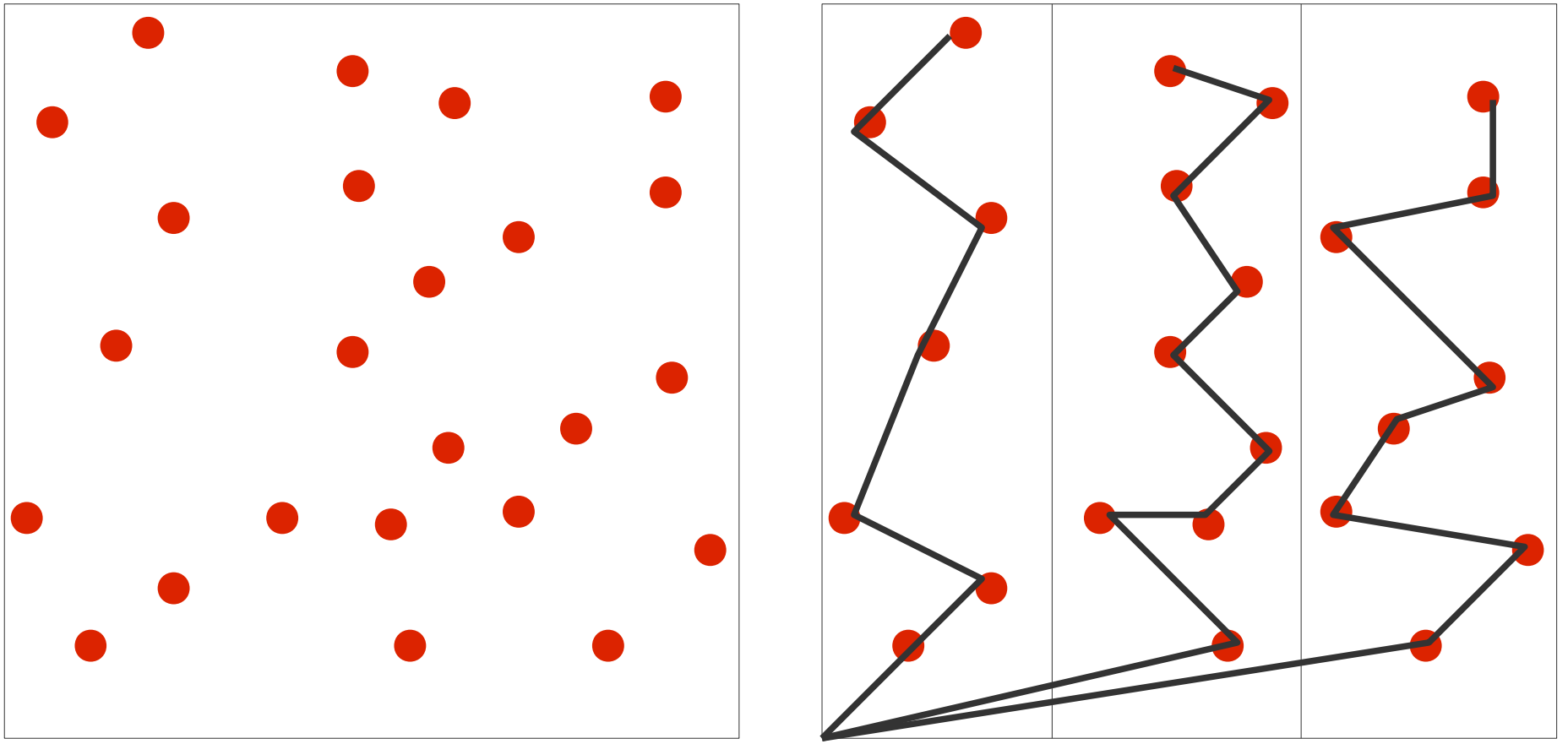
我们用线段树维护一段区间的每个数出现次数，那么假设当前的线段树维护的区间是 [L,R] ，我们下一个询问在 [L',R'], 那么我们就应该把 [L',L'),(R',R] 里的数都从线段树里删掉（或补足），这个操作次数不会超过 abs(L-L')+abs(R-R') ，假如把询问 [L,R] 表示为平面上的点 (L,R) ，我们就发现，从一个询问转移到另一个询问的代价是这两点的曼哈顿距离。于是，用平面图曼哈顿最小生成树 O(NlgN) 获得一个较优的总转移次数，可以证明这个总转移次数是 O(N^1.5+N^0.5*Q) 级别的。于是总时间复杂度就是 O(N^1.5*lgN+N^0.5*Q*lgN).

蒲公英 离线算法

—— 平面图曼哈顿最小生成树你会写吗？
—— 不会……
—— 其实，我们可以不用平面图曼哈顿最小生成树的。

蒲公英 离线算法

我们的任务，只是用可以接受的代价把所有点连接起来！
如果我们能构造出一种通用方法，
那么为什么还要写平面图曼哈顿最小生成树呢？



构造一棵生成树

具体构造方法

把横轴平均分成 O(n^0.5) 段，
每段里直接按纵轴上升次序处理

时间复杂度

横轴转移代价不超过 N^1.5,
纵轴转移代价不超过 Q*N^0.5 ,
总转移次数 (N+Q)*N^0.5
与曼哈顿最小生成树计算出的转移次数的复杂度相同

但是…… 问题要求在线算法

从离线算法中得到启示

一方面，我们要得到答案，必须获得查询区间的答案
另一方面，合并两个查询区间复杂度很高
怎么办？
那就是，预处理一些区间，
然后把查询区间往预处理的这些区间中的某个上面靠，
代价取决于两个区间的相似性
不重叠的部分越多代价越大

预处理多少区间？
预处理哪些区间？

蒲公英 solution1

将序列等分成 S 块
处理所有连续的块里每个数出现了多少次，
出现最多的数出现了多少次
O(S^2*N)

对于每个询问找到其能覆盖的最大的块
未能覆盖的元素 (区间不重叠的部分) O(N/S) 个
暴力修改取最大值 O(N/S)

蒲公英 solution1

假设 N 与 Q 同阶
那么我们要让
O(S^2*N)=O(N*N/S)
所以 S=O(N^(1/3))
时间复杂度 O(N^(5/3))-O(N^(2/3))

蒲公英 solution2

有没有更好的方法？
更科学的预处理方式！

蒲公英 solution2

从 solution1 得到启发
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。

蒲公英 solution2

我们依然考虑把序列分成 S 块，
记录“左端点是某个块起点，右端点任意”的区间的答案
不用记录每个数出现次数
O(N*S)
然后对于每个询问
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 N/S 个
答案要么是覆盖块的答案，要么是没被覆盖的元素
转化为询问一个数在一段区间内出现了多少次。

蒲公英 solution2

问题被转化后是不是可做了很多呢？
给定一个序列
询问一个数在一段区间内出现了多少次

蒲公英 solution2

把每个数出现的位置放在 vector 或 set 里
转化成求某个数是第几大
每次回答 O(lgN)
每个查询要查询 N/S 次
查询的复杂度是 O(N/S*lgN)
取 S=N^0.5, 时间复杂度为 O(N^1.5) – O(N^0.5*lgN) 。

蒲公英 solution3

进一步改进

考虑查询一个数在一段区间内出现了多少次的暴力方法

用 F[i,j] 表示 j 在区间 [1,i] 出现了多少次

$$F[i,j] = F[i-1,j] \quad \text{当} j \neq a[i]$$

$$F[i,j] = F[i-1,j] + 1 \quad \text{当} j = a[i]$$

但这样是 O(n^2) 的

蒲公英 solution3

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] ）。

能“撤销”做下去的操作！

函数式数据结构！

函数式二叉树？ O(log N) 不行
函数式块状链表！

蒲公英 solution3

假设分成 T 块
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(N/T+T)
取 T=N^0.5 做到 O(N^0.5) 每次修改
查询时可以直接找到第 X 次修改前的那个块状结构
每次 O(1)
每次查询众数要在块链中查询 O(N/S) 次
取 S=N^0.5 ，总时间复杂度 O(N^1.5) – O(N^0.5) 。

蒲公英 solution4

还能更好吗？ 能！

蒲公英 solution4

我们舍弃了一些特殊性！
所有的询问端点，都是分块的端点！
这个特殊性能怎么利用呢？

蒲公英 solution4

对于区间内某个数出现了多少次这个问题
我们可以借鉴最初的离线算法！
预处理一些区间的答案，
然后把询问区间往某个已经被预处理的区间上靠！

蒲公英 solution4

每个询问都能表示成部分和形式！
Query(L,R,X)=Query(1,R,X)-Query(1,L-1,X) ！
假设原分块第 i 块右端点是 R[i]
预处理所有 Query(1,R[i],X) 的询问的答案！
O(N^1.5)

蒲公英 solution4

每个询问可以利用部分和， O(1) 求出
因为每个询问端点都在分块端点上
故两个预处理区间相减就直接得到了我们要求的区间！

蒲公英 solution4

时间复杂度依然是 O(N^1.5) – O(N^0.5)
但优势在于它没有用到任何高级数据结构
常数比前一个方法小很多
十分易于实现

这个看似很棘手的问题被我们以 O(N^1.5) – O(N^0.5) 的复杂度非常漂亮的解决了。

前面的例子给出了很多种算法，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，望读者仔细理解。

更多例子请见我的论文

数据结构在字符串中的应用

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。

两个序列的比较函数

给定两个序列 a,b. 首先我们找到最大的 k 使得对于任意
1<=i<=k 均有 a[i]=b[i].
此时，如果 k=Len(a)=Len(b)，则 a=b
例： 'aa'='aa'
否则如果 k=Len(a) 则 a<b，如果 k=Len(b) 则 a>b
例： 'aab'>'aa', 'aa'<'aab'
否则如果 a[k+1]<b[k+1] 则 a<b，否则 a>b
例： 'aac'>'aabb', 'aabb'<'aac'

两个序列的比较函数

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) 了
发现 k 值满足二分性质
二分 k，问题转化成了快速判定两个序列是否全等
字典序 hash（指纹算法）
在 O(N) 预处理后 O(1) 回答。

两个序列的比较函数

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(lgN)
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。

指纹算法的直接应用

指纹算法的直接应用包括字符串匹配、各种 LCP 查询、各种与回文串有关的题目等。

字符串匹配

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.

字符串匹配 solution

在 O(N) 预处理后利用指纹算法
可以 O(1) 比较两个序列是否相等
因此直接枚举 k 后 O(1) 判断即可
时间复杂度 O(N).

LCP 查询

JSOI2008 火星人

给定一个序列 S，要求支持
插入一个元素
修改一个元素
查询 LCP(S[X,Len(S)],S[Y,Len(S)]).

火星人 solution

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
不过用 splay 维护 hash 值就可以 O(log N) 修改或插入了
总时间复杂度 O(log^2 N) 每次操作

更多例子请见我的论文

与回文串有关的题目

最长回文子串

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)
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

最长回文子串 solution

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是

$$LCP(Rev(S[1, X\ 1]), S[X + 1, Len(s)])$$

最长回文子串 solution

于是我们枚举这个中心位置
就可以在 O(lgN) 时间内算出以它为中心的最长回文子串
时间复杂度 O(NlgN)

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

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
每次查询某区间内最长的长度为奇数的回文子串的长度

30E Solution

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]
我们考虑二分答案长度的一半 Y
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
可以做到 O(lgN) 每次判定
时间复杂度 O(lg^2N) 每次查询。

更多例子请见我的论文

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

因为我们已经实现了 O(lgN) 的字符串比较函数
使用时间复杂度是 O(NlgN) 的排序算法直接对后缀排序
就可以在 O(Nlg^2N) 时间内构建后缀数组
Height 数组用 LCP 函数也可以在 O(NlgN) 内构建出来

构建后缀数组

快速排序在 Pas/C++ 内都提供有现成的代码可直接使用
只需手动实现 LCP 函数就可构建后缀数组
代码极简洁方便，且十分直观

虽然我们可以做到更快

扩展后缀数组

FZU1916 - 改

给定序列 S
在线支持：
在前端插入一个字符
查询当前序列有多少个不同的子串

Solution

一个序列相同子串个数是其后缀数组的 Height 数组元素和
因此我们被要求支持：
在序列前端插入元素
动态维护其后缀数组

传统后缀数组不能支持插入新元素！

Solution

后缀数组是什么？
排序后的后缀！
在前端插入新元素的影响？
不会影响原来后缀的次序！
只是插入了一个新后缀！
对 Height 数组的影响？
改变了新后缀插入位置相邻的两个 Height ！

用以后缀大小的关键字，快速比较为比较函数的
Splay 树维护后缀数组！ O(lg^2N)

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。

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) } 1<=X,Y<=N.

珠宝商 solution

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。

珠宝商 朴素算法 1

我们需要解决一个长度 S 的字符串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(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)
N 为要计算的树的结点数目。

珠宝商 朴素算法 2

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 。
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)
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。

珠宝商 solution

我们对这颗树进行点分治
找重心作为根结点
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。第二种是使用朴素算法 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。

珠宝商 solution

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。

珠宝商 solution

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 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 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 N/S 的子任务最多只需执行 O(S) 次化整为零。

而对这 N/S 个子任务逐个击破时间复杂度为 O((N/S)^2)*S = O(N^2/S).

于是取 S=N^0.5，化整为零的复杂度是 O(N^0.5*M), 逐个击破的复杂度是 O(N^1.5), 总时间复杂度是 O((N+M)*N^0.5).

珠宝商 solution

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)*N^0.5*lgN) ），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。

树上的数据结构

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。

DFS 序的性质与应用

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

给定一棵树，点上有权值。

单点加减 / 子树整体加减 / 路径整体加减
单点查询 / 子树和查询 / 路径和查询

组合出 3*3=9 个不同问题
除了路径整体赋值 + 路径和查询
其余问题我们将使用 DFS 序逐一解决

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。

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

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，这个问题被转化为前一个问题。

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。

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。

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，于是转化为上一问。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。

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。

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.

路径整体赋值 + 路径和查询还没有解决！

没关系，我们马上解决它。

树链剖分及其应用

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(lgN) 条重链和 O(lgN) 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。

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。

QTREE solution

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。

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。

QTREE4 solution

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。

更多例子请参见我的论文

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！

Thank you for listening.

Questions are welcomed.