NOIP2015 模拟赛 题解

By nodgd

1. 无聊的计算

1.1. 题目大意

根据题目给的规则生成了两个整数序列$a_1, a_2, \dots, a_n, b_1, b_2, \dots, b_m$，这些数的绝对值可能很大。求满足$0 \le a_i^{b_j} \bmod p \le q$的$(i,j)$有多少对。

1.2. 算法一

$n, m \le 6$，也就是说两个序列都不超过 6 项，而与生成序列有关系的数都不超过 1000，所以这个序列的所有数不会超出带符号 64 位整数类型的范围，可以直接计算出来。接下来就逐一验证每一对(i,j)就行了。这么做可以得到20分。

1.3. 算法二

$n, m \le 1000$时，我们显然不该直接把序列的每一项都算出来。这时，我们应该把目光放到$a_i^{b_j} \bmod p$这个运算身上。

我们知道，在$\bmod p$的意义下，a_i没有必要用真是的值，用$a_i' = a_i \bmod p$来代替a_i就可以得到完全一样的效果。这样一来，真正算出来的a_i'的大小都在0到$p-1$之间，就很方便了。

由费马小定理我们知道，当a不是质数p的倍数时，$a^{p-1} \equiv 1 (\bmod p)$。所以用$b_j' = b_j \bmod (p-1)$来代替$b_j$也可以得到完全一样的效果。于是，$b_j'$的大小也控制在了0到$p-2$之间。

然后，再枚举所有(i,j)，用快速幂计算出$a_i^{b_j} \bmod p$的值，就能统计出答案了。这个做法时间复杂度$O(nm \log p)$，可以得到40分。

1.4. 算法三

当$n, m \le 5000$时，我们期望得到一个算法，相比算法二，时间复杂度里少一个$\log p$，也就是去掉算法二中的快速幂这一步。

我们发现，对于任意的一个a_i，假如我们先不考虑b_j，而是先用看似暴力的普通求幂的方法来计算所有的a_i^k。也就是说，一开始得到了$a_i^0 = 1$，每次乘一个a_i。因为$b_j \le p-2$，所以至多进行$p-2$次就可以算出所有的a_i^k了。然后这时再枚举b_j，就能$O(1)$地获取$a_i^{b_j}$的值。

这个算法的时间复杂度是$O\big(n(m+p)\big)$，可以得到60分。

1.5. 算法四

容易发现，a_i和b_j的值并没有什么用处，有用的只是“有多少个$a_i = c$”、“有

多少个$b_j = d$”，不妨记为$cnta[c], cntb[d]$。显然$0 \le c \le p-1, 0 \le d \le p-2$，所以只需要用$O(p)$的时间枚举$c$，像算法三一样算出$c^0, c^1, \dots, c^{p-2}$，然后再枚举所有的$d$，就能统计出答案。

这个算法的时间复杂度是$O(n+m+p^2)$，可以得到100分。

2. 奇怪的队列

2.1. 题目大意

有n个人排队，身高为a_i的人要么左边由b_i个比她高的人，要么右边有b_i个比她高的人。找出字典序最小的满足条件的身高序列。$1 \le n \le 100000$。

2.2. 算法一

什么都不思考，直接输出“`impossible`”，可以得到10分。

2.3. 算法二

强行枚举所有排列，排列共有$n!$个，每个排列理论上说需要$O(n^2)$的时间来验证，但实际上会很快，因为发现任意一个数左右小于自己的数都不等于b_i就可以结束验证。可以证明，枚举完所有排列后，这一步验证的复杂度的平均值是$O(n)$的。所以总的时间复杂度是$O(n \times n!)$，可以得到40分。

2.4. 算法三

先判断无解的情况。对每个a_i找出有多少个a_j比他大记为c_i，如存在某个i的$b_i > c_i$就无解。这一步可以用排序来解决，把所有人按身高排序，然后扫一遍就能求出c_i并判定无解。

接下来，我们尝试把这个身高序列构造出来。

一开始我们有一个空序列。按身高从矮到高加入每个人。所有人的身高都不相等，显然身高为a_i的人要么加入到当前的从左到右的第$b_i + 1$个空位置里，要么加入到当前第$c_i - b_i + 1$个空位置里。注意到，比a_i更小的数值已经全部填入了身高序列中，所以a_i是剩下的数中最小的数，要想字典序最小，只需要让a_i的位置尽量靠左就行。于是，将a_i填入了第$\min\{b_i + 1, c_i - b_i + 1\}$个空位置里。

每加入一个数需要$O(n)$的时间来找出填入的位置，所以总的时间复杂度是$O(n^2)$，可以得到60分。

2.5. 算法四

算法二的瓶颈在于每次去找出从左到右的第$\min\{b_i + 1, c_i - b_i + 1\}$个空位置。我们针对这一步进行优化。

我们发现，针对空位个数的问题只有两个，一个是删除一个空位，一个是求出第k个空位在哪里。显然，我们可以用一个线段树来解决这个问题。首先，我们对整个空序列建一个线段树，线段树的每个节点记录下对应区间内的剩余空位的个数，在建树的时候也就是区间长度。然后，删除一个空位就是在线段树上进行简单的修改操作。求第k个空位置可以从线段树的根节点开始向下移动，如果左子节点对应区间的空位置个数大于等于k，则递归到左子节点；否则用k减去左子节点对应区间的空位置个数，再递归到右子节点；递归到叶子节点时，就找出了第k个空位置。

我们知道，这两个线段树上的操作的复杂度都是$O(\log n)$，所以最终的时间复杂度就是$O(n \log n)$，可以得到100分。

3. 仔细的检查

3.1. 题目大意

给了两棵树，判断他们的形状是否相同。如果相同，找出节点编号的对应法则，使第一棵树的每条边经过对应之后在第二棵树种都出现。两棵树的节点数相同，$n \leq 100000$，时间限制1s，内存限制256MB。测试数据中只有5分的数据答案是不相同，其他都是相同。

3.2. 算法一

什么都不思考，直接输出“NO”，可以得到5分。

3.3. 算法二

对应法则有$n!$种，强行枚举，在按照题目给出的判定规则判定是否正确。根据算法实现的常数大小可以得到5~10分，再结合算法一可以得到10~15分。

3.4. 算法三

假设现在要判定的是两棵有根树是否相同，我们可以同时 DFS 两棵树。每到一对节点$\langle u_1, u_2\rangle$时（u_1在第一棵树中，u_2在第二棵树中），我们需要判断它们管辖的子树是否相同。枚举u_1的每个子节点，当枚举到v_1时，再枚举u_2的每个还没有配对的子节点v_2，递归判断$\langle v_1, v_2\rangle$这一对节点，如果相同，则配对，否则枚举下一个v_2。如果所有的v_2都不能与v_1配对，则说明u_1与u_2两个子树不同，否则相同。

我们来计算一下这个算法的时间复杂度。写出时间复杂度的递推式

$$T(\langle u_1, u_2\rangle) = \sum_{v_1}\sum_{v_2} T(\langle v_1, v_2\rangle)$$

$$T\big(\text{叶子}, \text{叶子}\big) = O(1)$$

$$T\big(\text{叶子}, \text{非叶子}\big) = O(1)$$

计算的过程技巧性较强，在此略去，计算的结果表明，这个算法在最坏情况下复杂度是$O(n^2)$的。

针对这个算法，我们还可以加入很多优化，抓出一些有用的性质来加快判定的速度，例如比较子树的节点个数，或者比较子节点的个数，或者比较子树中与根节点距离最远节点的深度。这样可以大大加快判定速度。

上述算法只针对于有根树，也就是说，我们已经知道了第一个数的某个节点一定和第二颗树的某个节点是可以配对的。那么，无根树怎么处理呢？一个很自然的想法就是，在第一棵树中随便找一个节点，在第二棵树中枚举一个节点，然后运行上述算法。这样做的总时间复杂度是$O(n^3)$，加入一些优化之后可以让判定速度变得非常快，可以得到30~100分。

3.5. 算法四

我们发现，算法三的核心部分虽然是$O(n^2)$的，但是实际运行速度其实非常

快，几乎可以达到$O(n)$，主要是枚举根节点的操作把整个算法的速度拖慢了。这促使我们思考，如何避开枚举根的过程。

我们的算法是基于已知一对节点配对的，要想避开枚举根的过程，就必须快速找出一对一定配对的节点。于是我们马上联想树上的特殊点，比如树的重心，比如树直径的中点。树的重心至多有两个，所以可以用两次枚举来代替原来的n次枚举，从而大大的优化了算法；树的直径中点可能在一条边的中间，所以可以把原来的每条边替换成两条边和一个点，使直径中点一定是一个节点，然后直接运行这个算法就行了。这个算法时间复杂度虽是$O(n^2)$，但是可以得到50~100分。

3.6. 算法五

无论是算法三还是算法四，虽然实际运行效率非常高，但是理论上仍然是可以被卡掉的。于是，我们需要找一个理论上更加可靠的算法。注意到，这个算法理论上分析起来会很慢的原因是它不停的在比较两棵子树，比较两棵子树的时候有不得不多次递归调用。如果我们把树化为其他的方便进行比较的形态，说不定就能获得更优的理论复杂度。

我们知道，一棵有根树是可以写成括号表达式的，一个节点就是一对括号，括号内部的东西就是子树内的东西，括号外部的东西就是子树外的东西。当一个节点的子节点不止一个的时候，括号表达式就有多种写法了。但是，如果我们规定了括号表达式的顺序，然后把括号表达式按照一定的关系进行排序，那么括号表达式就是唯一的了。这就是我们所说的最小表示法。括号表达式的本质是字符串，可以进行简单的比大小操作。所以我们在求u子树的括号表达式的时候，先按照字典序由小到大对每个子节点的子树的括号表达式进行排序，然后再拼接起来得到u子树的括号表达式。由于是字符串排序，我们可以用基数排序之类的特殊排序方法，最后总的时间复杂度是$O(n^2)$，可以得到50~100分。

3.7. 算法六

我们发现，算法五虽然把树转换为了括号表达式，使信息线性化，但是并没有减小信息的量，即括号表达式的长度仍然和子树节点数是统一级别的。我们知道，要像压缩信息，就一定会有信息的损失，对于本题来说就是判断上的错误。

我们可以设定一个 Hash 函数，将一棵树换算为一个整数，然后只需要进行数字的比较就能判定两棵树是否相同。由于信息会有损失，所以我们的每次判定都是可能出错的。Hash 函数的设置比较自由，只要能够让两种不同形态的树对应的值不同即可。只要恰当的选择 Hash 函数，出错的概率就可以忽略不计。

这样做的时间复杂度是$O(n\log n)$或者$O(n)$，可以得到100分。

3.8. 算法七

Hash 虽好，但是毕竟有出错的概率。或者说，只要出题人仔细阅读了你的程序，就能构造出一组数据让你的 Hash 出错。那么，有没有能够确保万无一失的做法呢？显然是有的。

在算法五中，瓶颈在于对每个子节点的子树的括号序列进行排序。如果我们稍微改变一下排序的规则，就能降低最终的复杂度。对于两个括号序列，最体现特征的是序列的长度，而长度正好就是子树节点个数的两倍。所以我们可以先按照子树节点的个数进行排序，当结点个数相同的时候再考虑括号序列的字典序大小。

当我们要对u节点的子节点$v_1, v_2, \ldots, v_m$的括号序列进行排序时，$size[v_i]$都是相等的。使用基数排序，时间复杂度是$O(m \times size[v_i])$。另外，我们可以得到一个不等式关系

$$size[u] \geq m \times size[v_i]$$

注意到$m \geq 2$，我们可以发现一个很有趣的东西：对任意一个节点u，在u的某个祖先g处时如果u所在子树p与别的子树进行了字典序的比较，那么一定有

$$size[g] \geq 2 \times size[p]$$

显然，对于一个节点u的所有祖先，节点u所在子树用来基数排序的次数不可能超过$\log_2 n$。我们把每次基数排序的时间复杂度$O(m \times size[v_i])$均摊到这$m \times size[v_i]$个节点身上，则每个节点被均摊的时间复杂度是$O(\log n)$的，即基数排序的总的时间复杂度是$O(n \log n)$。

于是，我们就能够在$O(n \log n)$的时间复杂度里求出一个树的括号序列的类似于最小表示法的表示法，这样就能确保准确无误的判定两棵树是否相同了。用这个算法可以得到100分。