Hash 在信息学竞赛中的一类应用

【正文】

Hash 表作为一种高效的数据结构，有着广泛的应用。如果 Hash 函数设计合理，理想情况下每次查询的时间花费仅仅为 O(h/r)，即和 Hash 表容量与剩余容量的比值成正比。只要 Hash 表容量达到实际使用量的大约 1.5 倍以上，查询花费的时间基本就可以认为恒为 O(1)。

对于一个 Hash 表，一个好的 Hash 函数是尤其重要的，因为它能使 Hash 表保证效率。一个好的 Hash 函数最显而易见的特征是，能使不相同的东西经过 Hash 之后只有很小的几率相同。这样能避免过多冲突的产生。

Hash 表离不开 Hash 函数，但是反过来呢？有的时候，Hash 函数却是可以离开 Hash 表的。一个常见的例子就是著名的 MD5 算法，它是一个 Hash 函数，但是它的用途往往是对信息进行加密，以验证信息的正确性。换句话说，我们事实上是通过直接比较 MD5 算出的结果是否相同以推断原文内容是否一致。除了 MD5，常用的 CRC32 校验码和 SHA-1 算法也是基于类似的想法而产生的。

那么，信息学竞赛中，这样的算法有没有用武之地呢？

本文要讨论的，就是这一类以判重或判等价为目标的 Hash 函数。让我们来看看例题 1。

例题 1 多维匹配

题目大意

在一个串中求另一个串第一次出现的位置，很简单，*KMP* 即可。扩展到二维情况，就是求在一个矩阵中求另一个矩阵第一次出现的位置。而如果扩展到 k 维的情况，又该怎么做呢？待匹配数组 X 各维的尺寸为 N_1，N_2，...，N_k，模式数组 Y 各维的尺寸为 M_1，M_2，...，M_k。记 $N=N_1N_2\ldots N_k$，$M=M_1M_2\ldots M_k$。保证 $k\leq 10$，$N_i\geq M_i$，而 N 和 M 都不超过 500000。

算法分析

本题常见的算法是多维情况的 *KMP*，先算 1 维时的匹配情况，然后处理 2 维，3 维.....直到 N 维时的情况。时间复杂度为 O(k*(N+M))。但是它难以理解和记忆，也不容易在比赛中的短短几个小时完成和写对。

朴素的解法相对易于实现，但是使朴素算法很容易想到，而且针对数据又很容易制作，因此只有在完全没有思路的时候才值得一试。

有没有第三种选择呢？答案是肯定的。朴素的解法相当于枚举了每个起点（起点数量显然不会超过 N）并加以判断，然而判断两个子数组是否相同的时间复杂度高达 O(M)，这就是导致朴素算法在遇到针对数据的情况下很慢的原因。能不能在比较两个子数组之前先快速地排除一些明显不可能的情况呢？由于很容易构造出仅有一个字符不同的数据，不管采用什么顺序比较都会消耗大量的时间。

Hash 函数在这里派上了用场。我们可以对每个尺寸与模式数组一样的子数组计算一个 Hash 值。显然，只有当某个子数组的 Hash 值与模式数组的 Hash 值一样，它们才值得比较。

多维的 *KMP* 要从 1 维的情况开始考虑，并推广至高维。那么这种计算 Hash 函数的匹配算法我们也可以先考虑一维的情况——就是 Rabin-Karp 算法。

对于一个字符串 S，可以使用这个函数求出其 Hash 值：

$$f(S)=\sum_{i=1}^{len(S)} S[i]p^{len(S)-i} \mod q$$

那么，模式串 Y 的 Hash 值就可以轻松地求出。记待匹配串 X 从第 i 个字符开始的长度为 M 的子串为 S_i，则不难发现 $f(Si_{+1})$和 $f(S_i)$的关系：

$$f(S_i)=\sum_{j=i}^{i+M-1}(X[j]p^{i+M-1-j}) \mod q$$

$$\begin{aligned} f(S_{i+1}) &= \sum_{j=i+1}^{i+M}(X[j]p^{i+M-j}) \mod q \\ &= \left[p\cdot\sum_{j=i}^{i+M-1}(X[j]p^{i+M-1-j})+X[i+M]-p^M X[i] \right] \mod q \\ &= \left[pf(S_i)+X[i+M]-p^M X[i] \right] \mod q \end{aligned}$$

换个角度，如果不考虑 mod q，这个函数就是把字符串看作一个 p 进制数求出的值，这样，X_{i+1} 就是 X_i “左移”一位，然后去掉最前面一位，再加上右面新进来的一位得到的。因此上面的递推公式也是显然的。

有了这个递推公式，不难在线性时间内求出 X 的所有长度为 M 的子串的 Hash 值。

现在把问题扩展到高维的情况。不难发现要计算的 Hash 值的个数仍然不超过 N 个，可见，只要有合适的递推方法，仍然可以在线性时间内求出所有子矩阵的 Hash 值。事实上，一个 M_1 行，M_2 列的子矩阵可以被看作是 M_2 个“竖条”组成的一个串。我们可以先把每个长度为 M_1 的“竖条”计算出一个 Hash 值，然后再计算二维情况的 Hash 值。

需要注意的一点是，在计算一维的 Hash 值（“竖条”的 Hash 值）和计算二维的 Hash 值时使用的 p 值不能一样，不然不难想到下面反例：

a	b
a	a

a	a
b	a

它们并不相同，但是 Hash 的结果肯定一样。这就违背了使用 Hash 的初衷。因此，在第一维的计算和第二维的计算中要使用不同的 p 值。

二维情况时，求出所有长度为 M_1 的“竖条”所需要花费的时间是 O(N)的，然后把这些竖条的 Hash 值看作“字母”计算横向的 Hash 值（即各个子矩阵的 Hash 值），所花费的时间也是 O(N)的。总时间复杂度仍然是 O(N)的。

二维情况的 Hash 函数为（*len*1(*S*)和 *len*2(*S*)分别表示 *S* 在两个维度上的大小）：

$$f_2(S)=\sum_{i_1=1}^{len1(S)}\sum_{i_2=1}^{len2(S)}\left(p_1^{len1(S)-i_1}p_2^{len1(S)-i_2}S[i_1,i_2]\right)\ \bmod\ q$$

类似地，记待匹配矩阵 *X* 从第 i_1 行，第 i_2 列为左上角的 M_1 行，M_2 列的子矩阵为 X_{i_1,i_2}，

我们有递推式：

$$f_2\left(X_{i_1,i_2}\right)=\sum_{j_1=i_1}^{i_1+M_1-1}\sum_{j_2=i_2}^{i_2+M_2-1}\left(p_1^{i_1+M_1-1-j_1}p_2^{i_2+M_2-1-j_2}X[j_1,j_2]\right)\ \bmod\ q$$

$$f_2\left(X_{i_1,i_2+1}\right)=\sum_{j_1=i_1}^{i_1+M_1-1}\sum_{j_2=i_2+1}^{i_2+M_2}\left(p_1^{i_1+M_1-1-j_1}p_2^{i_2+M_2-j_2}X[j_1,j_2]\right)\ \bmod\ q$$

$$=p_2\cdot\sum_{j_1=i_1}^{i_1+M_1-1}\sum_{j_2=i_2}^{i_2+M_2-1}\left(p_1^{i_1+M_1-1-j_1}p_2^{i_2+M_2-1-j_2}X[j_1,j_2]\right)+\sum_{j_1=i_1}^{i_1+M_1-1}\left(p_1^{i_1+M_1-1-j_1}X[j_1,i_2+M\right.$$

$$\left.-\ p_2^{M_2}\sum_{j_1=i_1}^{i_1+M_1-1}\left(p_1^{i_1+M_1-1-j_1}X[j_1,i_2]\right)\right)\ \bmod\ q$$

$$=p_2\cdot f_2\left(X_{i_1,i_2}\right)+\sum_{j_1=i_1}^{i_1+M_1-1}\left(p_1^{i_1+M_1-1-j_1}X[j_1,i_2+M_2]\right)-\ p_2^{M_2}\sum_{j_1=i_1}^{i_1+M_1-1}\left(p_1^{i_1+M_1-1-j_1}X[j_1,i_2]\right)\ \bmod$$

式中 $\sum_{j_1=i_1}^{i_1+M_1-1}\left(p_1^{i_1+M_1-1-j_1}X[j_1,i_2+M_2]\right)$ 和 $\sum_{j_1=i_1}^{i_1+M_1-1}\left(p_1^{i_1+M_1-1-j_1}X[j_1,i_2]\right)$ 都是一维情况下的 Hash 值，即预先计算出的“竖条”的 Hash 值。

扩展到 *k* 维情况，则不难想到，先算出所有尺寸为 $M_1\times M_2\times\ldots\times M_{k-1}$ 的子数组的 Hash 值，再使用类似上面的办法把这些尺寸为 $M_1\times M_2\times\ldots\times M_{k-1}$ 的子数组的 Hash 值看作一个个字符而使用 Rabin-Karp 的 Hash 函数计算出各个尺寸为 $M_1\times M_2\times\ldots\times M_k$ 的子数组的 Hash 值。可见，需要 *k* 轮计算就可以算出所有尺寸为 $M_1\times M_2\times\ldots\times M_k$ 的子数组的 Hash 值，时间复杂度为 O(*kN*+*M*)。

k 维时的 Hash 函数：

$$f_k\left(X_{i_1,i_2,\cdots,i_k}\right)=\sum_{j_1=i_1}^{i_1+M_1-1}\sum_{j_2=i_2}^{i_2+M_2-1}\cdots\sum_{j_k=i_k}^{i_k+M_k-1}\left(p_1^{i_1+M_1-1-j_1}p_2^{i_2+M_2-1-j_2}\cdots p_k^{i_k+M_k-1-j_k}X[j_1,j_2,\cdots,j_k]\right)\ m$$

类似地可以推出递推式

$$f_k\left(X_{i_1,i_2,\cdots,i_k+1}\right)=\left[p_2\cdot f_k\left(X_{i_1,i_2,\cdots,i_k}\right)\right.$$

$$+\sum_{j_1=i_1}^{i_1+M_1-1}\sum_{j_2=i_2}^{i_2+M_2-1}\cdots\sum_{j_{k-1}=i_{k-1}}^{i_{k-1}+M_{k-1}-1}\left(p_1^{i_1+M_1-1-j_1}p_2^{i_2+M_2-1-j_2}\cdots p_{k-1}^{i_{k-1}+M_{k-1}-1-j_k}X[j_1,j_2,\cdots\right.$$

$$\left.-\ \mathrm{p_k^{M_k}}\sum_{j_1=i_1}^{i_1+M_1-1}\sum_{j_2=i_2}^{i_2+M_2-1}\cdots\sum_{j_{k-1}=i_{k-1}}^{i_{k-1}+M_{k-1}-1}\left(p_1^{i_1+M_1-1-j_1}p_2^{i_2+M_2-1-j_2}\cdots p_{k-1}^{i_{k-1}+M_{k-1}-1-j_k}X[j_1,j\right.\right.$$

如果没有最后的 mod *q* 操作，那么这个 Hash 函数可以理解成一个进制转换，当每次的 *p* 都取得比当时的字符集还要大的时候，只要 Hash 值不同就一定能确定内容不同。但是事实上计算出来的 Hash 值就会大到使这个算法没有意义的程度（比较这个“Hash 值”的速度不会比直接比较更快），因此取余是必须的。

一个理想情况下的 Hash 函数，如果能产生 *S* 种不同的值，那么对两个不一样的子数组算出一样的值的可能性就只有 1/*S*。我们的 Hash 函数当然未必能达到理想情况，但是由“1/*S*”这个式子不难想到，加大 *S* 就能有效地减少判错的可能性。并且还能得出另一个结论：如果一个 Hash 函数的精确程度不够，只需要再计算一个与之不同的 Hash 函数，就可以有效地提高正确率！当然，在本题中，时间复杂度已经近似于（因为毕竟这个 Hash 函数不是理想的 Hash 函数）O(*kN*+*N**(1/*S*)**M*)了。*S* 只要大小不比 *N*/*k* 小，后一项就不是时间复杂度的瓶颈了，也没有必要在这方面下太大功夫。反而，如果计算多个 Hash 函数，会显著增加算法的常数。

关于 p 和 q 的取值，由于可以理解成一个 p 进制的转换，再对 q 取余，应当在范围允许的情况下尽量避免冲突，建议：

1、 p 不宜过小，不然很容易出错。

2、 q 可以取一个质数，然后 p 选取一个能使对于所有 1 至 p-2 的 i，$p^i \bmod q \neq 1$。

这个 Hash 函数不但可以这样滚动计算，也可以采用分段，分块，线段树等办法计算和维护，具体的一些办法可以参见后面的一些例题。

例题 2 Equal squares (Ural 1486)

题目大意

在一个 $N \times M$ 的字符矩阵中找到两个相同的子正方形矩阵（可以相交），并使找到的两个子正方形矩阵的边长尽量大。

算法分析

有了例题 1 的经验，在面对本题的时候不难想到，二分查找正方形的边长，然后使用一个 Hash 表来判断该边长是否可行。Hash 函数与例题 1 的二维情况一致。时间复杂度是 $O(NM\log(N*M))$的。

但是不幸的是，本题时间限制很严，这样做的程序依然会超时。最后在使用一个看上去有点冒险的改动之后，终于通过了本题：直接检查是否产生了相同的 Hash 函数，如果存在相同的 Hash 函数，则认为该边长可行，否则即认为它不可行。

这道题目就这样通过了，但是这样的改动是不是太冒险了一点呢？事实上，两个子正方形的内容不同而 Hash 值相同的可能性很小。所以，一般情况下我们可以认为，如果 Hash 值相同，则原内容相同。

这里还有一个问题：Hash 表存放了什么？如果存放的是 Bool 而不使用位压缩，则似乎浪费了不少空间，而且 Hash 表不可能开到很大，因此两个不同的内容 Hash 到同一个位置还是有可能的，简单地比较"这个位置是否被 Hash 到"仍然不算保险，毕竟，一次查询有 100 万分之一的可能性出错的话，100 万次查询出错概率就很可观了。这里介绍的一个小技巧是计算两个 Hash 值，一个用于确定 Hash 表中的位置，另一个则用于比较。这样，只有如果在 Hash 表的某个地址开始查找找到了一个 Hash 值与自己的 Hash 值一样的元素，才认为自己在 Hash 表中出现过。

例题 3 不稳定匹配

题目大意

有两个串 A 和 B，每次可以进行的操作有：

INSERT i j：在 A 中第 i 个字符前插入 j。

DELETE i：删除 A 中第 i 个字符。

REVERSE i j：把 A 中 i 到 j 之间的内容反转。

QUERY i j：求 A 从第 i 个字符开始，B 从第 j 个字符开始能匹配的最大长度，即询问 A 从第 i 个字符开始的后缀与 B 从第 j 个字符开始的后缀的 LCP 长度。

算法分析

如果两个串不会变化，求 LCP 只需要求出 $A+B$ 的后缀数组即可。但是本题的 A 串是不断变化的，而且由变化的方式可以看出，每次操作都会导致后缀数组发生很大的变化，因此我们应该另辟蹊径。

对于一个 k，不难判断两个串的 LCP 是否有至少 k 个字符：计算这两个串的前 k 个字符的 Hash 值，并且比较它们是否相等。如果相等，就几乎可以肯定地认为这两个串的 LCP 至少有 k 个字符，否则，它们的 LCP 长度肯定不到 k。这样，就可以通过二分查找来计算 LCP。

现在问题又转化成了怎么求一个子串的 Hash。不妨仍然采用 Rabin-Karp 的 Hash 函数，如果已知了 S_1 和 S_2 的 Hash 值，不难求出 S_1+S_2 的 Hash 值：

$$
\begin{aligned}
&f(S_1+S_2)\\
&=\left[\sum_{i=1}^{len(S_1)+len(S_2)} p^{len(S_1)+len(S_2)-i}(S_1+S_2)[i]\right] \quad \bmod q\\
&=\left[\sum_{i=1}^{len(S_1)} p^{len(S_1)+len(S_2)-i}S_1[i]+\sum_{i=1}^{len(S_2)} p^{len(S_2)-i}S_2[i]\right] \quad \bmod q\\
&=\left[p^{len(S_2)}\sum_{i=1}^{len(S_1)} p^{len(S_1)-i}S_1[i]+\sum_{i=1}^{len(S_2)} p^{len(S_2)-i}S_2[i]\right] \quad \bmod q\\
&=\left[p^{len(S_2)}f(S_1)+f(S_2)\right] \quad \bmod q
\end{aligned}
$$

而 $p^{len(S_2)}$ 的 Hash 值显然又是可以在 O(n)时间 内预处理的得到的。因此，可以在 O(1)时间内通过两个字符串的 Hash 值得到它们连接后得到的串的 Hash 值。因此，可以使用块状链表维护计算 A 中子串的 Hash 值，方法于维护计算部分和类似，不同之处在于一个字符串正向和反向的 Hash 值是不同的，为了能在 O($n^{0.5}$)时间内完成 reverse 操作，应当要能在 O(1)时间内把一个块“反转”，这就要求我们为一个块维护两个 Hash 值：一个是正向的，一

个是倒向的。除此之外的操作于维护部分和或者维护最大值类似。这样，插入，删除，反转操作是 $O(n^{0.5})$的，而查询操作是 $O(n^{0.5}\log n)$的。

类似地，本题的另一种解法是在一棵 *splay* 树上维护 Hash 值。每次一个节点被旋转或以它为根的子树被修改时，则计算它的正向 Hash 值和反向 Hash 值，这样，就可以在 O(1)时间你 reverse 一棵子树，通过 *split* 可以不难地把一棵子树在均摊 $O(\log n)$时间内反转。插入，删除操作显然也是均摊 $O(\log n)$的，而查询操作的均摊时间复杂度为 $O(\log^2 n)$。

例题 4 一类同构判定的问题

问题 1：比较两棵树是否相同。

不难想到的算法是使用两个字符串分别表示两棵树，但是如果使用 Hash 的话应该怎么做呢？

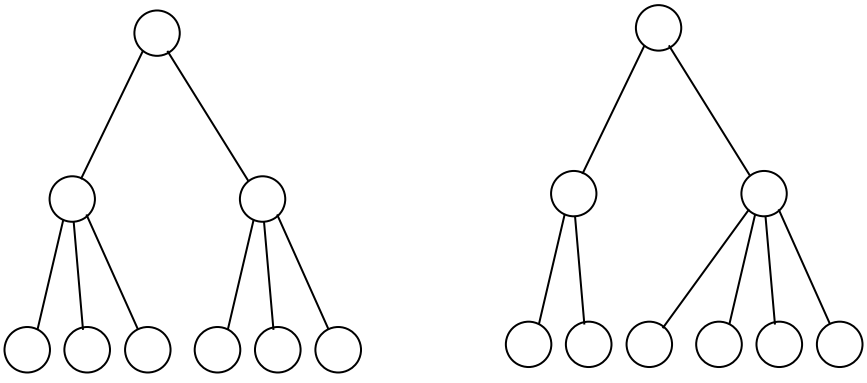
可以使用一种类似树状递推的方法来计算 Hash 值：

对于一个节点 v，先求出它所有儿子节点的 Hash 值，并从小到大排序，记作 H_1，H_2，...，H_D。那么 v 的 Hash 值就可以计算为：

$$Hash(v) = ((\cdots(((((a \cdot p \ \ xor\ H_1) \bmod q) \cdot p\ xor\ H_2) \bmod q) \cdot p\ xor \cdots H_{D-1}) \bmod q) \cdot p\ xor\ H_D) \cdot b$$

换句话说，就是从某个常数开始，每次乘以 p，和一个元素异或，再除以 q 取余，再乘以 p，和下一个元素异或，除以 q 取余.....一直进行到最后一个元素为止。最后把所得到的结果乘以 b，再对 q 取余。

之所以事先对儿子的 Hash 值进行排序，是因为仅仅儿子的顺序不同并不会导致树的不同，而后面如何 Hash 就比较随意了，即使不用这个 Hash 函数，只要使用一个效果足够好的 Hash 函数也是可以的。但是要注意，诸如 Rabin-Karp 的那个 Hash 函数就不适合在这里应用，因为不难找到反例：



考虑右边的树的儿子的顺序如图所示的情况（可以认为 Hash 函数的大小是随机分布的，因此有一半的可能性出现这种情况），由于叶子节点的 Hash 值必然相同（因为都是等价的），不妨记作 l，然后递推关系是：

$$Hash(v) = b \cdot \sum_{i=1}^{D} p^{D-i} v_i \bmod q$$

则左边的树的 Hash 值为：

$$lb^2(p^2 + p + 1)(p + 1) \bmod q$$

右边为：

$$lb^2[(p + 1)p + (p^3 + p^2 + p + 1)] \bmod q$$

不难看出他们是相等的，而两棵树是不相等的。

类似地，前面的计算方法如果最后不乘以 b，也是错的（可以分析树退化成线形的情况）。

那么，既然直接比较 Hash 值肯定是有概率出错的，为什么还要指出哪些 Hash 函数适用，哪些不适用呢？有的错误是“偶然误差”，不改变 Hash 的计算方法，仅仅改变 p 等常数即可消除，而有的则是“系统误差”，是这个 Hash 函数本身的不合理导致的，后一种情况应尽量避免出现。当然，在比赛的短短几个小时中未必能够判断这个 Hash 函数究竟合不合理，但是尽量选择在不确定 p,q 等可以改变的常数的情况下难以构造反例的 Hash 函数会是比较好的办法。

现在，只需要比较两棵树的 Hash 值，即可分辨它们是否相同。时间复杂度为 O($n$log$n$)。

还有另一种 Hash 函数也能解决这个问题，而且对于解决下一个问题很有帮助：考虑把这棵树使用一个串表示出来（类似最小表示），然后计算那个串的 Hash 值。当然，如果真的像最小表示法那样把字符串弄出来，就有点得不偿失了：可以仅仅根据儿子的 Hash 值对儿子进行排序，由于 Rabin-Karp 的 Hash 函数是可以处理两个串连接后的 Hash 函数值的，在确定了顺序并且知道了各个儿子的节点数（即子串的长度），不难确定当前节点的 Hash 值。

换句话说，如果用 $c_1,c_2,\ldots,c_D$ 表示当前节点 v 的儿子节点，$s(x)$表示根节点为 x 的子树所转化成的子串，则

$s(v) = g(v)s(c_1)s(c_2)\cdots s(c_D)$，这里 $g(v)$是某个关于 v 的有一定识别能力的函数，比如取节点 v 的儿子数，或者某个关于节点 v 的 Hash 函数均可。

而 Rabin-Karp 算法的 Hash 函数为

$$f(S) = \sum_{i=1}^{len(S)} S[i] p^{len(S)-i} \bmod q$$

因此有

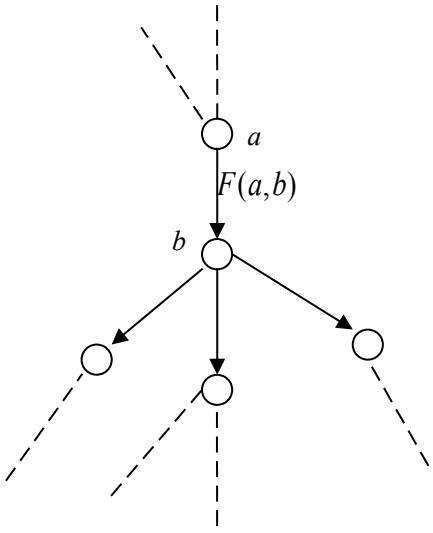
$$\begin{aligned} f(s(v)) &= f(g(v)s(c_1)s(c_2)\cdots s(c_D)) \\ &= \left\{ p^{len(s(v))-1} \cdot g(v) + \sum_{i=1}^{D} \left[f(c_i) p^{\sum_{j=i+1}^{D} len(C_D)} \right] \right\} \bmod q \end{aligned}$$

可见，进行一次在树上的递推，则 Hash 值不难算出。

问题 2：比较两棵无根树是否相同。

此时再使用最小表示的话，时间复杂度就很高了。但是，如果合理地使用 Hash，时间复杂度依然只有 O($n$log$n$)。

具体办法如下：对于每条边 $(a,b)\in E$，计算一个 Hash 值 $F(a,b)$，使之与所有满足 $i\neq a$ 且 $(b,i)\in E$ 的 $F(b,i)$ 有关，这里 $F(a,b)$ 和 $F(b,a)$ 是两个概念，如图：

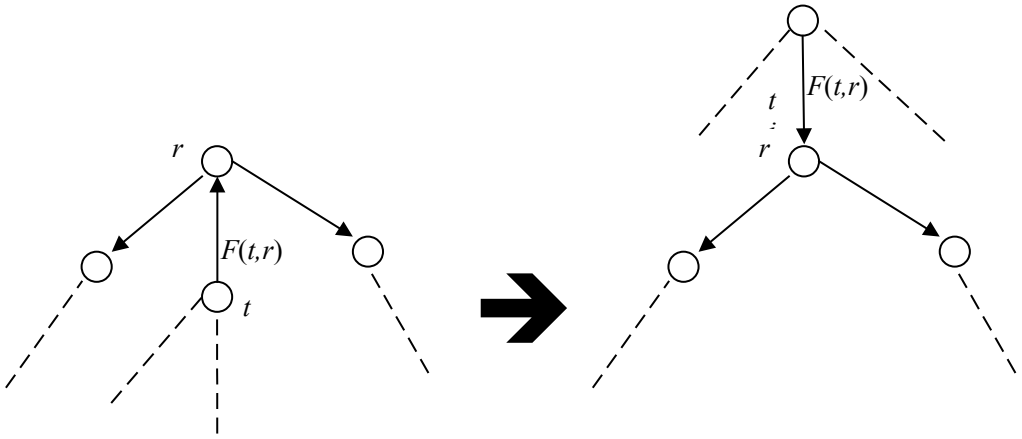


可见，这样的递推计算不会引起循环，一定可以依照某个顺序进行。

$F(a,b)$ 就用来表示"以 a 作为 b 的父亲节点时，b 为根的子树的 Hash 值。"

首先，任意选取一个节点 r 作为根节点，对于一个有序点对 (a,b)，满足 $(a,b)\in E$，如果 a 比 b 离根节点近，则称 (a,b) 是向下的，否则是向上的。显然，可以使用刚才问题 1 中给出的第二个方法在 O($n$log$n$)时间内求出所有向下的 (a,b) 对应的 $F(a,b)$。

现在考虑这个根节点 r。显然所有 (r,i) 都是向下的，因此所有 $F(r,i)$ 都是已知的。现在就是要计算各个 $F(i,r)$。对于 r 的某一个儿子节点 t，如图所示：



那么 $F(t,r)$ 可以通过 r 的除了 t 以外其他的儿子节点的 Hash 值计算出来。但是如果直接这样计算，时间复杂度在最坏情况下就是 O(n^2)的了。

这里要用到树状递推的一个小技巧：由于所有 $F(r,i)$ 都是已知的了，不妨事先对它们进行排序，然后考虑。由于这个 Hash 函数本质上还是对串的 Hash，不妨把 r 所连接的各个点按 $F(r,i)$ 排序，分别记作 $c_1,c_2,\ldots,c_D$，记 $S(a,b)$ 为 $F(a,b)$ 所对应的那个串。

可以在 $c_1,c_2,\ldots,c_D$ 中二分查找到 t 的位置 k，则 $S(t,r)$ 可以表示成：

$$s(t,r)=g(r)s(r,c_1)s(r,c_2)\cdots s(r,c_{k-1})s(r,c_{k+1})\cdots s(r,c_D)$$

$g(r)$ 略去不谈（同例一），$s(r,c_1)s(r,c_2)\cdots s(r,c_{k-1})$ 和 $s(r,c_{k+1})\cdots s(r,c_D)$ 在排完序后都不难在 O(n)时间预处理算出。因此，可以在 O(log n)时间内算出 $F(t,r)$，因为用到了二分查找。

当一个节点 v 满足 $F(v,father(v))$ 已经被计算出之后，就相当于对于所有与之相邻的点 i，$F(v,i)$ 都是已经计算出的了。这个情况就和刚才关于根节点 r 的讨论相同了，在此不再赘述。

这样，第一遍 *dfs* 可以确定所有向下的 $F(a,b)$，而第二遍 *dfs* 又可以确定所有向上的 $F(a,b)$，这样，就计算出了。时间复杂度如何呢？第一遍复杂度显然是 O($n$log$n$)，而第二遍的时间复杂度：对于一个有 D 个儿子节点的节点，时间复杂度为 O(Dlog$D$) < O(Dlog$n$)，而 ΣD=N，因此时间复杂度仍然是 O($n$log$n$)。

一个顶点的 Hash 值可以定义为，以这个节点为根时整棵树的 Hash 值。在所有的 $F(a,b)$ 计算出之后，顶点的 Hash 值不难计算。

最后，如果两个无根树的顶点的 Hash 值可以一一对应（即排序以后完全对应，或者使用一个 Hash 表判定），就可以认为它们是相等的，否则认为它们是不相等的。

由于顶点的 Hash 值就相当于把这个顶点作为根节点时得到一棵有根树，再 Hash 得到的结果。可以直接用顶点的 Hash 值是否相同比较两个顶点是否在树里处于等价的位置上，或者可以通过顶点的一一对应得到两棵结构一样但是顶点顺序不同的无根树的顶点的对应关系。由于变更一个顶点会导致整个树上所有节点的 Hash 值发生变化，因此 Hash 值相同也能反映出“这两个节点所属的树很可能是同一棵，或是完全一样的两棵”，这种 Hash 甚至可以用来判断一个森林里哪些点是处在等价的位置上的。

问题 3：比较两个有向图是否相同。

由于无向图的情况可以转化为有向图的情况考虑，这里只讨论有向图的情况。我目前还没有找到什么非常有效的 Hash 的办法，但是有一个实践中没有遇到过问题的算法：

先枚举一个起始点 i，所有结点的 Hash 值置为某个常数（不妨取 1）。然后进行迭代，每次一个节点新的 Hash 值计算为：

$$f_{j+1}(v)=\left[a\cdot f_j(v)+b\cdot\sum_{w\in V,v\to w\in E}f_j(w)+c\cdot\sum_{w\in V,w\to v\in E}f_j(w)+g(v)\right]\mod p$$

这里 $g(v)=\begin{cases}d & v=i\\ 0 & v\neq i\end{cases}$

然后迭代 k 次，计算出的 $f_k(i)$ 就是 i 点的 Hash 值。

其实关键就是在于 $f_j(i)$ 的计算和其他值不同，即可避免很多特殊情况。

有关题目：The labyrinth of wells(POI 1999/2000 Stage 2)

题目大意：有若干个房间，最上面一个是起点，最下面一个是终点，除了最下面的房间外每个房间都是 3 种颜色中的一种，且有 3 根不同的管道通往它下面的房间。所有的房间都可达，且所有的管道最后通往的都是最底下的房间。但是有些节点其实本质是一样的（采取一样的走法就能经过同样颜色），问把本质一样的节点合并之后，最后最少能剩下多少点。两个点能合并当且仅当它们后面的路的颜色完全一样，可以据此设计一个 Hash 函数，即 Hash(*x*)=*f*(Hash(*x*->*c*1),Hash(*x*->*c*2),Hash(*x*->*c*3),*x*->*color*)，*f* 是某个 Hash 函数。在我的程序中，Hash 函数为：

f(*a*,*b*,*c*,*d*)=((129(129*a xor b*) *xor c*) *xor* 987654321) + *d*。

顺便介绍一下，为了提高程序的运行效率，可以使用一些位操作的技巧，如 *a* * 129 = (*a shl* 7) + *a*。

例题 5 等价表达式(NOIP2005)

题目大意

给定一个没有化简的整式，然后 *n* 个整式，求出这 *n* 个整式里哪些与一开始给定的多项式相等。

限制条件：*n* <= 26，表达式长度不超过 50，整式中只出现数字,*a*,+,-,*,^（乘方运算），左右括号和空格。数字都小于 10000，乘方运算中幂次都不超过 10，+-*^()都只能作运算符，且不会出现省略乘号的现象。

算法分析

常规的解法是把各个整式都展开并合并同类项，然后进行比较。这个算法当然是可以通过比赛时的测试数据的。

但是这样的数据呢（显然这组数据是符合题目要求的）？

(*a*+9999)^9^9^9^9^9^9^9^9^9^9^9^9^9^9^9^9^9^9^9^9

不难看出，这组数据展开以后得到的多项式根本无法存储下来，也难以表示它后面各项的系数。面对这样的庞然大物，怎么比较两个都如此巨大的整式是否相同？显然，即便把这个式子计算出来都是极为困难的，更不要说去比较了。

事实上，解决办法很简单：给 *a* 带一个或几个值，根据带值计算出的结果对某个大数字取余的余数（不取余的话上面那组"巨型"数据还是解决不了)是否相等，直接判定这两个整式是否相等。而代入一个值去求值，也可以看作是在本题应用了一个特殊的 Hash 函数对两个整式计算了 Hash 值，并通过比较 Hash 值来比较原式是否相等。当然，如果 *a* 取了若干个合适，可以证明出这个算法就变成一定正确的了。

例 题 6 Guess the Number(OIBH Reminiscence Programming Contest 改编)

题目大意

给定一个 S(最多 500000 位)，求 N，使得 $N^N=S$。但可能给定的 S 可能有一些位上的数字错了(不会出现增加或减少位数的情况)，此时应输出-1。

算法分析

首先，对于比较小的 S 可以直接判断处理，而对于超过 10^{10} 的 S，我们可以确定一个可能的 N（因为 $\frac{(N+1)^{N+1}}{N^N}$ 在 $N\geq 10$ 时显然大于 10，不可能位数相同），剩下的工作就是判断这个 N 满不满足 $N^N=S$。显然，算对数是不可取的，因为实数类型没有这么高的精度。同理，高精度计算也是难以实现的（即使是 *FFT* 也面临着时间复杂度的挑战）联系到前面的例题，可以想到，找到一个 Hash 函数 $f(x)$，通过比较 $f(N^N)$和 $f(S)$是否相同即可确定 N^N 是否和 S 相等。同时，这个 $f(x)$还要满足，如果 $x=ab$，则可以方便地通过 $f(a)$和 $f(b)$来计算得到 $f(x)$，以便可以不把 N^N 计算出来就直接比较。

一个很简单但是很有效的 Hash 函数就是取余！即，$f(x)=x \bmod p$。如果 $x=ab$，则 $f(x)=f(a)\,f(b) \bmod p$。

对于原题（S 和 N^N 最多相差一位），只要 p 不能被 10 整除，即可保证判断的准确性。

而改编之后的题目呢（S 可以修改任意位）？取若干个 p，由于 S 的长度限制（只有 500000）位，不超过 130000 以内质数的乘积。换句话说，在 500000 内随便取一个 p，恰好能被 $|S-N^N|$ 整除的可能性只有大约 1/4，如果多取一些，或者再加大 p 的范围就可以几乎肯定判断是正确的了。

【总结】

Hash 函数除了做为 Hash 表的辅助工具，在单独使用的时候可以实现几乎肯定地判断两个数据是否相同或等价。Hash 函数的本质是对含有较大信息量的信息加以“概括”。因此在遇到需要频繁比较两个数据是否相同的操作的问题时，不妨考虑使用 Hash 函数作为解题的工具。

有时使用 Hash 函数会牺牲一定的正确性，而且使用 Hash 函数的算法看上去也不够优美。但是灵活使用 Hash 函数，可以化难为易，解决一些原来难以解决的问题。也往往可以

用较短的代码获得较高的效率。

尽管有所舍弃，却也有所收获；正因为有所舍弃，才会有所收获。在选择算法的时候，总会遇到这样的矛盾，有失必有得，有得也必有失，选择恰当的适合题目也适合自己思维习惯的算法才是最好的。

【参考文献】

刘汝佳 黄亮《算法艺术与信息学竞赛》 清华大学出版社

陈启峰《NEW LCP》2006 年国家集训队作业

《数据结构》清华大学出版社