上回说到，小Hi和小Ho接受到了河蟹先生伟大而光荣的任务：**河蟹先生将要给与他们一篇从互联网上收集来的文章，和一本厚厚的河蟹词典，而他们要做的是判断这篇文章中是否存在那些属于河蟹词典中的词语。**

当时，小Hi和小Ho的水平还是十分有限，他们只能够想到：**“枚举每一个单词，然后枚举文章中可能的起始位置，然后进行匹配，看能否成功。”**这样非常朴素的想法，但是这样的算法时间复杂度是相当高的，**如果说词典的词语数量为N，每个词语长度为L，文章的长度为M，那么需要进行的计算次数是在N\*M\*L这个级别的**，而这个数据在河蟹先生看来是不能够接受的。

于是河蟹先生决定先给他们个机会学习一下，于是给出了一个条件N=1，也就是说词典里面事实上只有一个词语，但是希望他们能够统计这个词语在文章中出现的次数，这便是我们常说的**模式匹配**问题。而小Hi和小Ho呢，通过这一周的努力，学习钻研了KMP算法，并在互相帮助之下，已经成功的解决掉了这个问题！

这便是[Hiho一下第三周](http://hihocoder.com/contest/hiho3)发生的事情，而现在第四周到了，小Hi和小Ho也要踏上解决真正难题的旅程了呢！

任务回顾

小Hi和小Ho是一对好朋友，出生在信息化社会的他们对编程产生了莫大的兴趣，他们约定好互相帮助，在编程的学习道路上一同前进。

这一天，他们……咳咳，说远了，且说小Ho好不容易写完了第三周程序，却发现自己错过了HihoCoder上的提交日期，于是找小Hi哭诉，小Hi虽然身为管理员，但是也不好破这个例，于是把小Ho赶去[题库](http://hihocoder.com/problemset)交了代码，总算是哄好了小Ho。

小Ho交完程序然后屁颠屁颠的跑回了小Hi这边，问道：“小Hi，你说我们是不是可以去完成河蟹大大的任务了呢？”

小Hi思索半天，道：“老夫夜观星象……啊不，我这两天查阅了很多资料，发现这个问题其实也是很经典的问题，早在06年就有信息学奥林匹克竞赛国家集训队的论文中详详细细的分析了这一问题，而他们使用的就是Trie图这样一种数据结构！”

“Trie图？是不是和我们在第二周遇到的那个Trie树有些相似呀？”小Ho问道。

“没错！Trie图就是在Trie树的基础上发展成的一种数据结构。**如果要想用一本词典构成Trie图的话，那么就首先要用这本词典构成一棵Trie树，然后在Trie树的基础上添加一些边，就能够变成Trie图了！**”小Hi又作老师状。

“哦！但是你说了这么多，我都不知道Trie图是什么样的呢！”小Ho无奈道。

“也是！那我们还是从头开始，先讲讲怎么用Trie树来解决这个问题，然后在Trie树的基础上，讨论下一步应该如何。”小Hi想了想说道。

#### 提示一：如何用Trie树进行“河蟹”

“还记得我们在第二周时，是如何使用Trie树解决字符串自动补全问题的么？”小Hi如是问道。

“还记得，就是对于每一个询问，根据其每个位置上的字符，在Trie树上走出对应的边！”小Ho的记忆力还是挺不错的，很快便答了上来。

小Hi满意的点了点头，继续问道：“那你想想怎么用Trie树来解决河蟹先生交代的任务？”

“好的！”小Ho满口答应，随即分析道：“现在的这个问题和第二周遇到的问题的不同之处在于，**第二周时一定是从询问的第一个字符开始匹配，然后找出所有可能的匹配，而我们现在遇到的问题是可以从询问的任意一个位置开始匹配，看是否会在Trie树上走到一个标记结点（标记结点对应路径为一个属于词典的单词）。**”

“没错，那你准备怎么做呢？”

“我准备对于螃蟹先生给我的文章，还是像之前我们相出的朴素算法那样，枚举一个起始位置，然后我们的问题就变成了：**是否从这个起始位置开始的一段字符（也就是从这个起始位置开始的字符串的一个前缀字符串），它存在于“河蟹”词典里面 ？**而这个问题，就和第二周的问题几乎一样了，唯一不同的是，我是要一直在Trie树中走下去直到无边可走，或者走到一个标记结点的时候才能够停下来，前者代表没有任何需要河蟹的单词，后者则说明我们找到了。”小Ho井井有条的分析道。

“也就是说，**第二周我们成功解决了计算前缀匹配的数量这样一个问题，而这一周的任务却是可以在任意位置匹配，所以我们就枚举一个起始点，将这个问题转化成前缀匹配这样一个我们已知的问题来做，这样的思路么？”**小Hi总结道。

“嗯！我就是这么想的~”小Ho道。

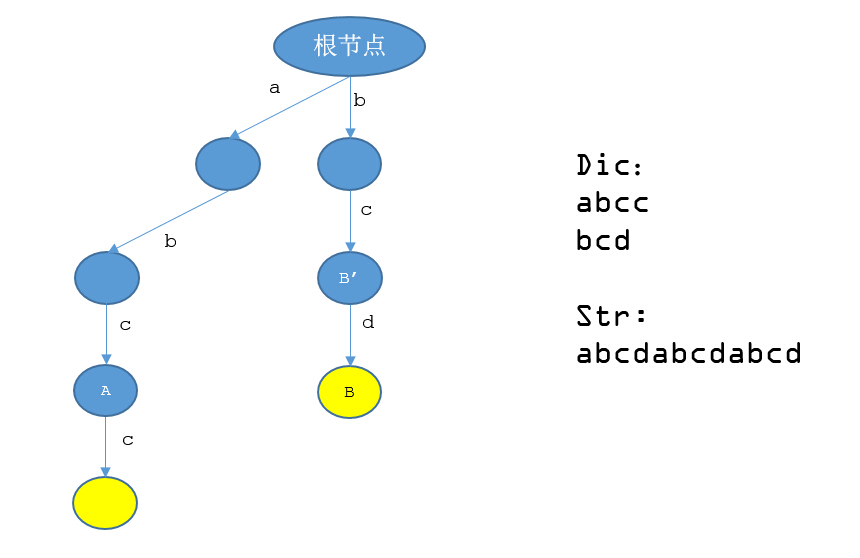
“嗯，这个方法听起来挺有意思的，而且仔细分析一下，这样做所需要的计算次数会在M\*L这个数量级上，比我们之前的朴素算法已经好了很多呢~”小Hi夸奖了一番。

“嘿嘿，但是你之前说的Trie图是怎么回事，它又能将计算次数缩减到怎样的数量级呢？”小Ho的好奇心也是燃烧了起来。

“且听我说~”

“现在我们有了一个时间复杂度在O(ML)级别的方法，但是我们的征途在星辰大海，啊不，我们不能满足于这样一个60分的方法。所以呢，我们还是要贯彻我们一贯的做法，**寻找在这个算法中那些冗余的计算**！“小Hi道：”那么我们现在来看看Trie树进行计算的时候都发生了些什么。”

“你看这组输入——文章str、词典dic还有我们构建的Trie树tree，我们在算法过程中，**先枚举第一个字符作为起始位置，并最多匹配到第k个字符，因为str[1..k]这一段在tree中对应的结点A结点没有str[k+1]这一条边。这时候我们便要枚举第二个字符作为起始位置，并最多匹配到第k2个字符，这同样是因为str[2..k2]这一段在tree中对应的结点B结点没有str[k2+1]这一条边。也就是说我们在最开始的计算中，要先从tree的0号结点走到A结点，然后回到0号结点，再走到B结点。**”小Hi在黑板上画了一些奇奇怪怪的符号，对小Ho如是解说道。



“是的！等等，我怎么觉得这里似曾相识呢？”小Ho奇道。

“问得好~那么你觉不觉得这个过程和上一周的KMP算法很相似，都是**枚举原串（文章、str）的起始位置，然后在模式串（Trie树）中依次进行匹配？**”小Hi说道。

“是的！不同之处就在于**模式串就是在一个数组里一个个匹配下来，而Trie树则是在一个树结构中一个个顺着边走~这无非就是单个词语和多个词语的差别了是么？**”小Ho也是一点就透。

“没错！那我们再回想一下我们当时是怎么优化KMP的——**我们既然已经从str的当前起点i开始匹配了l个长度，那么在枚举str的下一个起点i+1的时候，就意味着最开始的l-1个字符都已经在之前的计算中匹配过了，如果我们能够利用好这个信息的话，就能够大大的减少时间复杂度。**”

“换句话说，**如果我们从str的当前起点开始，匹配了l个长度走到了A结点，如果我们把A结点对应的字符串（即从tree的0号走到A结点的路径）去掉第一个字符，形成一个新的字符串，那么这个字符串肯定是和从str的下一个起点开始，长度为l-1的子串是一样的，而如果我们能够预先找到这个字符串在tree中对应的结点B'，我们就不用像之前所说的那样从0号节点走到A结点然后回到0号结点再走到B结点，而是可以直接从0号结点走到A结点然后直接跳转到B’结点然后再根据从str[i+l..k1]这一段走到B结点！**”小Hi一口气说道，顿时感觉口干舌燥，于是拿起了一旁的杯子，猛灌了一口凉开水。

”哦！那么如果用之前的这个例子的话，从str的第一个位置开始，匹配了3个字符走到了A结点，对应的字符串是abc，如果第一个字符a去掉变成bc，这个字符和从str的第二个位置开始长度为2的字串bc的确是一样的，此时bc在tree中对应的结点是B'结点，所以我们用之前的算法的话就是从0号结点走到A结点，然后再从0号结点走到B结点，现在可以直接从A结点走到B‘结点，然后根据str的第4（i+l=1+3）个字符走到B结点！”小Ho趁着小Hi休息的功夫，也是拿起了之前小Hi给出的例子推演道。”

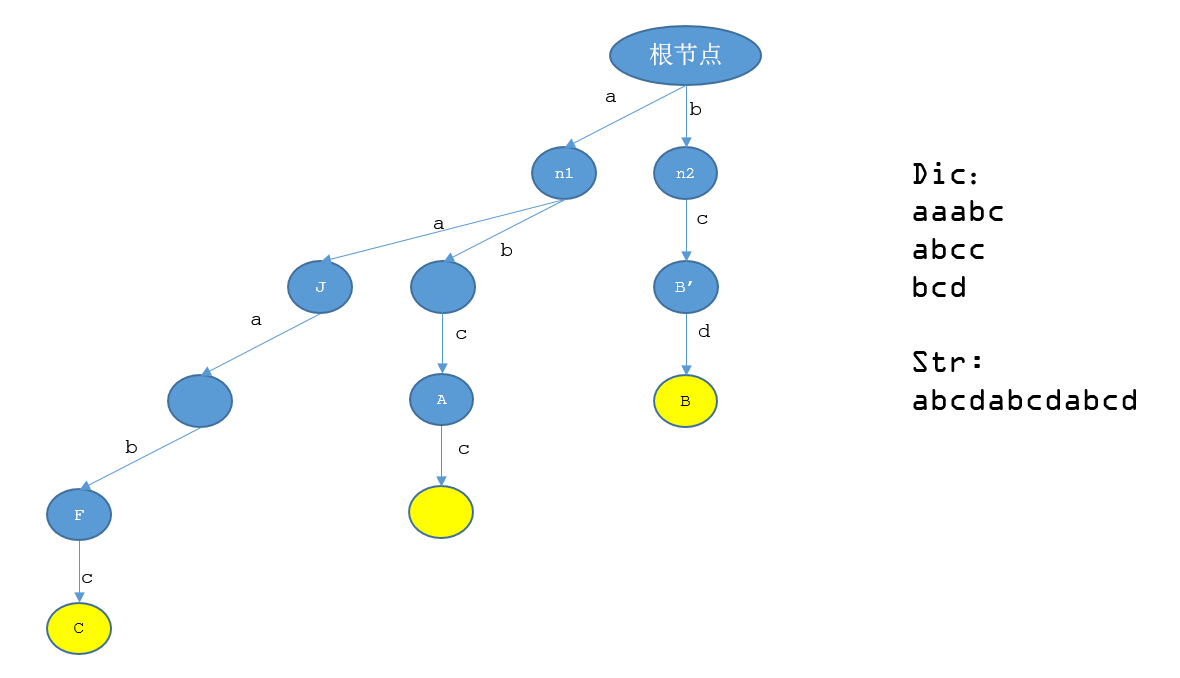
“没错！所以我们的问题规约成了：**如何对于一棵给定的Trie树，找到其中每一个结点对应的后缀结点——这个结点在Trie中对应路径去掉第一个字符之后在Trie中对应的结点。**“小Hi擦了把汗，感觉舒爽许多，于是继续说道。

“我大致懂了！这个后缀结点就和我们在KMP算法中求解的NEXT数组是一个意思！”小Ho开心道。

“你真聪明~”小Hi夸奖道。

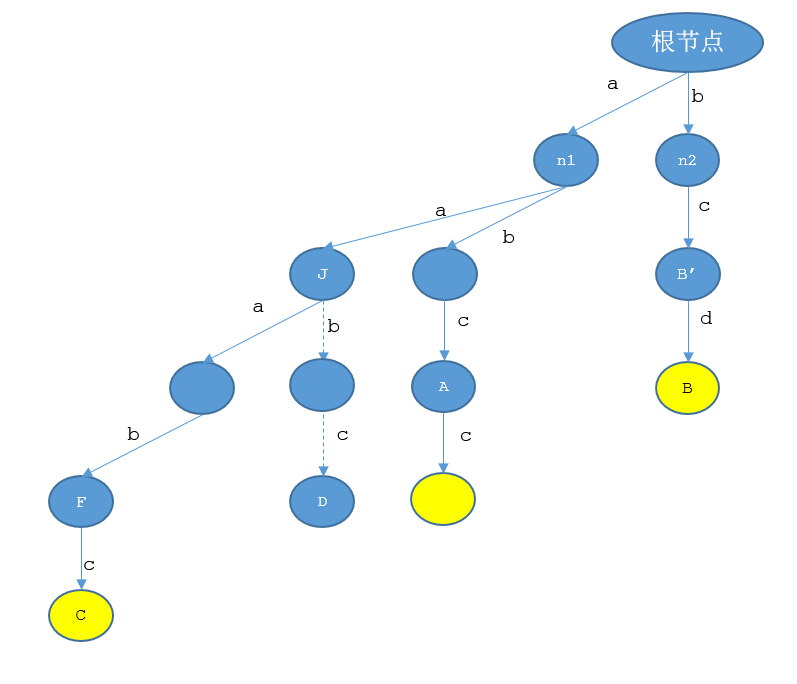
“那么现在……”小Hi刚要开口，就被小Ho无情打断。

“可是小Hi老师~你看在这种情况下，结点C找不到对应的后缀结点，它对应的路径是aaabc，而aabc在Trie里面是走不出来的！”小Ho手中挥舞着一张纸，问道。



“你个瓜娃子，老是拆老子台做啥子！……阿不，小Ho你别担心，我这就要讲解如何求后缀结点呢~”小Hi笑容满面的说道。

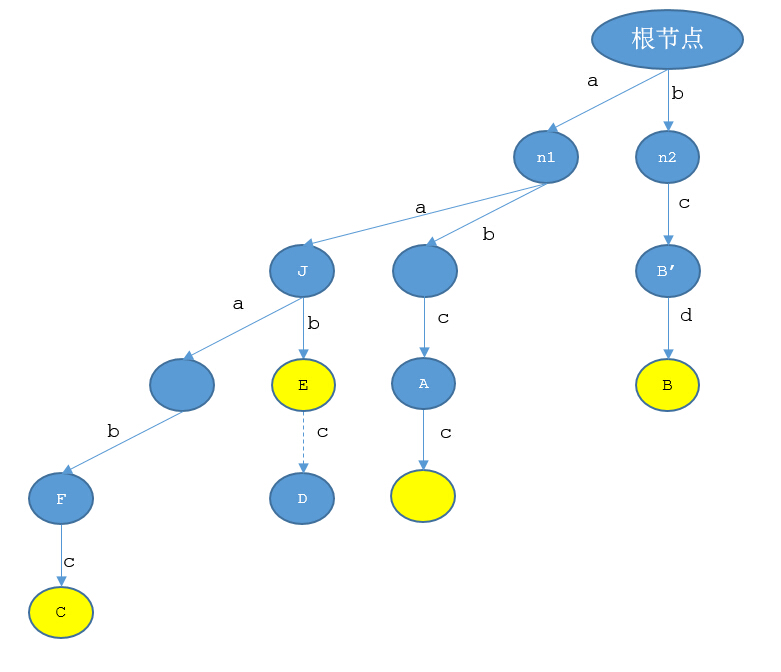
“先看之前你说的那个例子，如果tree中存在一个结点D，其对应的路径是aabc，那么这个结点的后缀结点是哪一个？”小Hi问道。



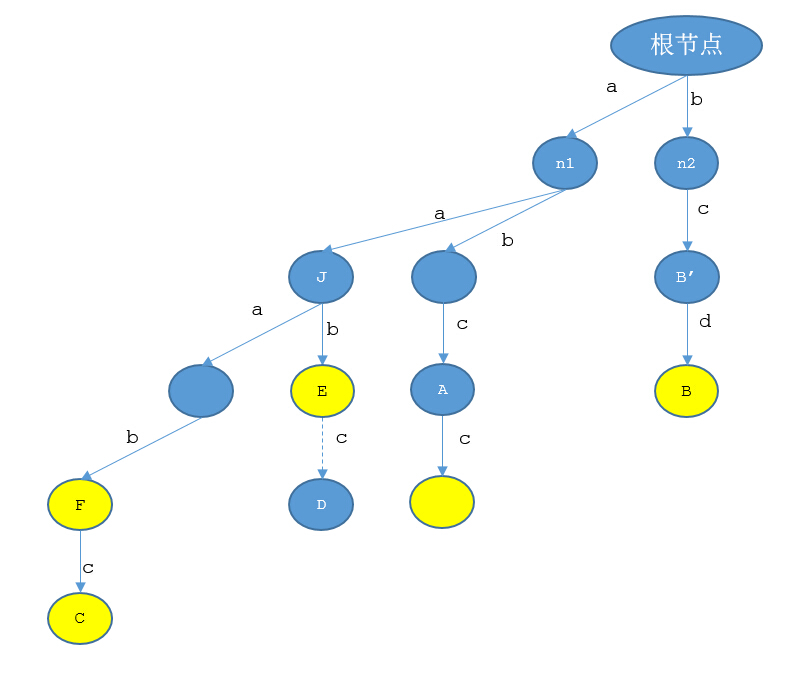
“aabc……去掉第一个字符就是abc，对应的是A结点，所以D结点的后缀结点是A！”小Ho很快便做出了回答。

“那么问题不就简单了么，既然结点D是不存在的，那么不就意味着这个开始结点的枚举，是肯定在中途就要找不到实际上是没有意义的么，直接从C结点跳转到A结点就可以了！所以只需要令C结点的后缀结点是A结点，像D结点这种不存在的结点当然要视为冗余计算，扔掉就行了！”小Hi老师斩钉截铁道。

“D结点好可怜……但是，如果从tree的根节点到D结点的路径中有标记结点怎么办？这样的跳过会不会导致标记结点被忽略掉了？”小Ho问道。



“如果不注意的话是会的呢！这就要引进一个新的概念，**后缀结点为标记结点的结点也需要被标记**，比如像对应路径为aab的E结点就是标记点对么？而aaab对应的F结点的后缀结点便是E结点，所以需要对F结点进行标记，这样在走到F结点的时候，就知道已经匹配出了一个河蟹词语了呢。”小Hi耐心答道。



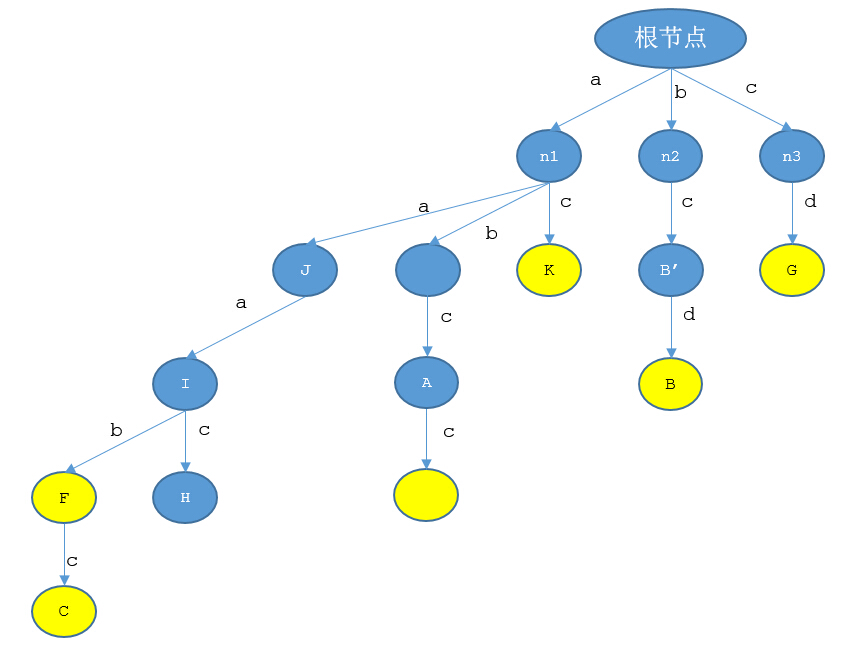
“那么接下来就开始说怎么快速有效的求后缀结点！小Ho，你先回答我：**树结构最大的特点是什么？**”小Hi问道。

“是递归结构！”小Ho想也没想就回答道。

“真聪明！虽然是导演安排好的台词，但是回答速度真是一流呢！”小Hi点了点头，继续说道：“所以我们**想要求Trie树种每个结点的后缀结点，最直观的方法也就是像当初我们求解KMP的NEXT数组时那种从左到右的拓扑顺序一样，从根节点开始，以宽度优先遍历的顺序，依次求解每一个结点的后缀结点。**”

“嗯！这样可以保证**每个结点对应的后缀结点，由于其对应字符串长度一定至少少1，所以一定会在它之前得到计算**？但是这样有什么用呢？诶，我想到一个，这样就可以知道它的后缀结点是不是标记结点了，从而决定自己是不是要被标记是么？”小Ho决定打破砂锅问到底。

“别急！听我慢慢说来。”小Hi不知从哪摸出一把羽扇，扇了两下，问道：“你看这棵Trie树，根节点的后缀结点是哪个？”



”根节点对应的字符串是空，去掉第一个字符……还是空，所以就是根节点自己了是吧？”小Ho想了想，说道。

“是的，那你看从根节点连出去的这三个点n1,n2,n3他们的后缀结点是哪个？”小Hi继续问道。

“他们对应的字符串都只有一个字符，所以去掉一个字符就变成空了，于是他们的后缀结点也都是根节点。”小Ho也继续答道。

“那么现在，**假设所有深度小于B结点的结点的后缀结点都已经算出来了，我想要算B结点的后缀结点，有没有什么好的方法呢？**”小Hi随手填了几个结点的后缀结点，向小Ho问道。

“如果考虑递归的思路的话，B结点的父亲结点是对应字符串为bc的B'结点，B'结点的后缀结点是n3结点，所以从B'结点出发经'd'这样一条边到达的结点B的后缀结点自然应该就是从B'结点的后缀结点n3出发经'd'这样一条边到达的结点——G结点了！”小Ho仔细研究了下，答道。”这么说来，是不是所有结点都可以这么求呀，如果它父亲结点是通过编号为char的一条边走向它的，那么只要找到它父亲的后缀结点，并且走出编号为char的一条边，就能够找到它的后缀结点了？”

“差不多就是这个思路呢！但是你有没有想过**如果它父亲结点的后缀结点并没有编号为char的一条边，你该怎么办**？”小Hi也是不厌其烦，继续问道。

“我想想，比如说结点G的父亲结点I的后缀结点J没有'c'这样一条边，但是结点J的后缀结点n1却有'c'这样一条边，由于后缀结点每次都是去掉前几个字符，所以后缀结点的后缀结点也相当于是“弱”一点的后缀结点，在没有更好的选择的情况下（因为这是第一次找到的有'c'这样一条边的后缀结点），G的后缀结点就应该是结点K了吧！”小Ho仔细想想，答道。

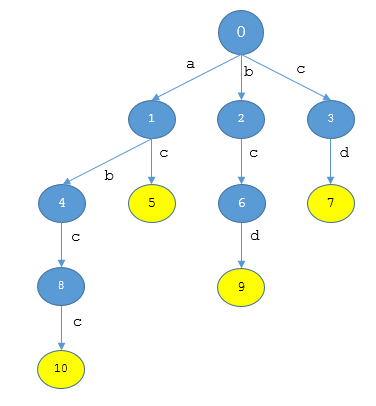
“你这样的话，会不会觉得，每次都要往回不停的找后缀结点，挺浪费时间的呢？”这下子换到小Hi打破砂锅问到底了。

“那该怎么办？”小Ho也是没辙了。

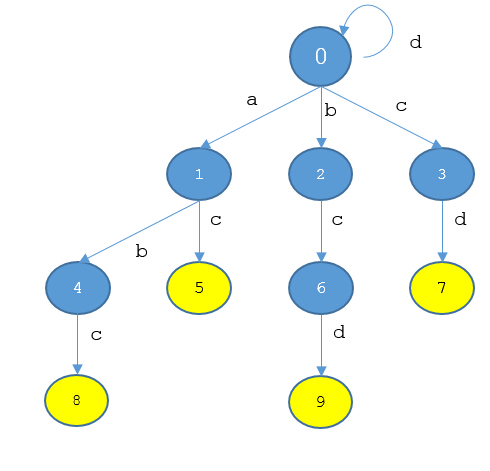
“你看看这么做怎么样，我还是按照宽度优先搜索的顺序遍历整棵树，对于每一个结点，我不仅仅要求出它的后缀结点，我还要求出到达这个点后，经由每一个char（比如'a'..'d'）会走到的结点。由于到达这个结点之后，所有深度比它小的结点的这些值都算出来了，于是我可以直接通过父亲节点的后缀结点经由“父亲节点走到当前结点经过的边”走到的结点来计算我的后缀结点，同时这个后缀结点所要计算的值也都计算出来了，所以我可以通过这个后缀结点经由每一个char（比如'a'..'d'）会走到的结点来计算我经由每一个char（比如'a'..'d'）会走到的结点。”小Hi大致的说了一下思路。

“小Hi老师，我听晕了！”小Ho报告说。

“这个简单，我就拿这个例子给你依次算一算。”



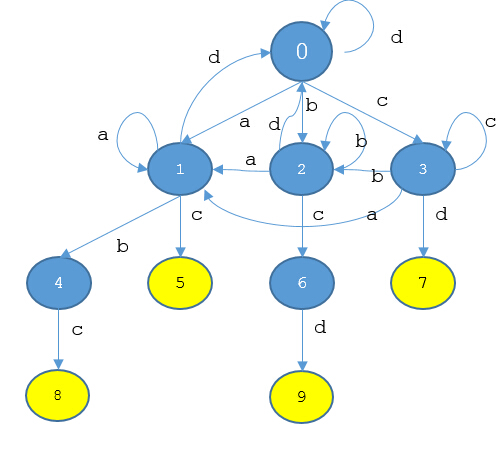
“如果用trie(X)表示X的根节点，next(X)('a')表示从X出发标号为'a'的边指向的结点，我们可以知道trie(0)=0, next(0)('a')=1, next(0)('b')=2, next(0)('c')=3, next(0)('d')=0。”



“由于trie(1)=0, 我们可以补上从1出发的'a','d'这两条边：next(1)('a')=next(0)('a')=1, next(1)('d')=next(0)('d')=0”

“由于trie(2)=0, 我们可以补上从2出发的'a','b','d'这三条边：next(2)('a')=next(0)('a')=1, next(2)('b')=next(0)('b')=2, next(2)('d')=next(0)('d')=0”

“由于trie(2)=0, 我们可以补上从3出发的'a','b','c'这三条边：next(3)('a')=next(0)('a')=1, next(3)('b')=next(0)('b')=2, next(2)('c')=next(0)('c')=3”



“由于trie(4)=next(trie(1))('b')=2, 我们可以补上从4出发的'a','b','d'这三条边：next(4)('a')=next(2)('a')=1, next(4)('b')=next(2)('b')=2, next(4)('d')=next(2)('d')=0”

“由于trie(5)=next(trie(1))('c')=3, 我们可以补上从5出发的'a','b','c','d'这四条边：next(5)('a')=next(3)('a')=1, next(5)('b')=next(3)('b')=2, next(5)('c')=next(3)('c')=3, next(5)('d')=next(3)('d')=7”

“由于trie(6)=next(trie(2))('c')=3, 我们可以补上从6出发的'a','b','c'这三条边：next(6)('a')=next(3)('a')=1, next(6)('b')=next(3)('b')=2, next(6)('c')=next(3)('c')=3”

“由于trie(7)=next(trie(3))('d')=0, 我们可以补上从7出发的'a','b','c','d'这四条边：next(7)('a')=next(0)('a')=1, next(7)('b')=next(0)('b')=2, next(7)('c')=next(0)('c')=3, next(7)('d')=next(0)('d')=0”

“由于trie(8)=next(trie(4))('c')=6, 我们可以补上从8出发的'a','b','c','d'这四条边：next(8)('a')=next(6)('a')=1, next(8)('b')=next(6)('b')=2, next(8)('c')=next(6)('c')=3, next(8)('d')=next(6)('d')=9”

“由于trie(9)=next(trie(6))('d')=7, 我们可以补上从9出发的'a','b','c','d'这四条边：next(9)('a')=next(7)('a')=1, next(9)('b')=next(7)('b')=2, next(9)('c')=next(7)('c')=3, next(9)('d')=next(7)('d')=0”

“此时这个图已经变得过于复杂了，我就不画出来了，但是我想你已经可以从我上面所说的知道每个节点的后缀结点了呢！”小Hi道。

“原来如此！这样我就知道了每一个结点的后缀结点了，接下来我就可以很轻松的解决河蟹先生交给我的问题了呢！”小Ho高兴的说道：“但是，说好的Trie图在哪里呢？”

小Hi不由笑道：“你这叫买椟还珠你知道么？还记得我们再计算后缀结点的时候计算出的从每个点出发，经由每一个char（比如'a'..'d'）会走到的结点么？把这些边添加到Trie树上，就是Trie图了！”

“原来是这样，但是这些边感觉除了计算后缀结点之外，没有什么用处呀？”小Ho又开始问问题了。

“这就是Trie图的巧妙之处了，你想想你什么时候需要知道一个结点的后缀结点？”小Hi实在不忍看自己的兄弟这般呆萌，只能耐着性子解释。

小Ho顿时恍然大悟，“**在这个结点不能够继续和文章str继续匹配了的时候，也就是这个结点没有“文章的下一个字符”对应的那条边，哦！我知道了，在Trie图中，每个结点都补全了所有的边，所以原来需要先找到后缀结点再根据“str的下一个字符”这样一条边找到下一个结点，现在可以直接通过当前结点的“str的下一个字符”这样一条边就可以接着往下匹配了，如果本来是有这条边的，那不用多说，而如果这条边是根据后缀结点补全的，那便是我们想要的结果！**”

“所以呢！完成这个任务的方法总的来说就是这样，先根据字典构建一棵Trie树，然后根据我们之前所说的构建出对应的Trie图，然后从Trie图的根节点开始，沿着文章str的每一个字符，走出对应的边，直到遇到一个标记结点或者整个str都已经匹配完成了~”小Hi适时的总结道。

“而这样的时间复杂度则在O(NL+M)级别的呢！想来是足以完成河蟹先生的要求了呢~”小Ho搬了搬手指，说道。

“是的！但是河蟹先生要求的可不是想法哦，他可是希望我们写出程序给它呢！”

输入

每个输入文件有且仅有一组测试数据。

每个测试数据的第一行为一个整数N，表示河蟹词典的大小。

接下来的N行，每一行为一个由小写英文字母组成的河蟹词语。

接下来的一行，为一篇长度不超过M，由小写英文字母组成的文章。

对于60%的数据，所有河蟹词语的长度总和小于10, M<=10

对于80%的数据，所有河蟹词语的长度总和小于10^3, M<=10^3

对于100%的数据，所有河蟹词语的长度总和小于10^6, M<=10^6, N<=1000

输出

对于每组测试数据，输出一行"YES"或者"NO"，表示文章中是否含有河蟹词语。

**样例输入**

6

aaabc

aaac

abcc

ac

bcd

cd

aaaaaaaaaaabaaadaaac

**样例输出**

YES