KMP

一、算法介绍

KMP 算法是拿来处理字符串匹配的。也就是给你两个字符串，你需要回答，B 串是否是 A 串的子串（A 串是否包含 B 串）。比如，字符串 A="I'm matrix67"，字符串 B="matrix"，我们就说 B 是 A 的子串。我们称等待匹配的 A 串为主串（母串），用来匹配的 B 串为模式串。

解决这类问题，通常我们的方法是枚举从 A 串的什么位置起开始与 B 匹配，然后验证是否匹配。假如 A 串长度为 n，B 串长度为 m，那么这种方法的复杂度是 $O(nm)$ 的。虽然很多时候复杂度达不到（验证时只看头一两个字母就发现不匹配了），但我们有许多“最坏情况”，比如，$A = aaaaaaaaaaaaaaaab$，$B = aaaaaaaaaaab$.我们将介绍的是一种最坏情况下 $O(n)$ 的算法（这里假设 $m \le n$），即 KMP 算法。

二、算法流程

假如，$A = abababaababacb$，$B = ababacb$.我们来看看 KMP 是怎么工作的。我们用两个指针 i 和 j 分别表示，$A[i-j+1\ldots i]$ 与 $B[1\ldots j]$ 完全相等。也就是说，i 是不断增加的，随着 i 的增加 j 相应地变化，且 j 满足以 $A[i]$ 结尾的长度为 j 的字符串正好匹配 B 串的前 j 个字符（j 越大越好，后面会说明这样的正确性），现在需要检验 $A[i+1]$ 和 $B[j+1]$ 的关系。当 $A[i+1]=B[j+1]$ 时，i 和 j 各加一；当 $j=m$ 了，我们就说 B 是 A 的子串（B 串已经完整匹配了），并且可以根据这时的 i 值算出匹配的位置。当 $A[i+1]\neq B[j+1]$，KMP 的策略是调整 j 的位置（减小 j 值）使得 $A[i-j+1\ldots i]$ 与 $B[1\ldots j]$ 保持匹配且尝试匹配新的 $A[i+1]$ 与 $B[1\ldots j]$（j 的值这里仍然尽量大）。我们看一看当 $i=j=5$ 时的情况。

```
i = 1 2 3 4 5 6 7 8 9 ……
A = a b a b a b a a b a b …
B = a b a b a c b
j = 1 2 3 4 5 6 7
```

此时，$A[6]\neq B[6]$。这表明，此时 j 不能等于 5 了，我们要把 j 改成比它小的值 j'。j'

可能是多少呢？仔细想一下，我们发现，j'必须要使得$B[1\ldots j]$中的头j'个字母和末j'个字母完全相等（这样j变成了j'后才能继续保持i和j的性质）。这个j'当然要越大越好。在这里，$B[1\ldots5]=ababa$，头3个字母和末3个字母都是aba。而当新的j为3时，$A[6]$恰好和$B[4]$相等。于是，i变成了6，而j则变成了4：

```
i = 1 2 3 4 5 6 7 8 9 ……
A = a b a b a b a a b a b …
B =     a b a b a c b
j =     1 2 3 4 5 6 7
```

从上面的这个例子我们可以看到，新的j可以取多少与i无关，只与B串有关。我们完全可以预处理出这样一个数组$P[j]$，表示当匹配到B数组的第j个字母而第$j+1$个字母不能匹配了时，新的j最大是多少。$P[j]$应该是所有满足$B[1\ldots k]=B[j-k+1\ldots j]$的$k(k<j)$的最大值。

再后来，$A[7]=B[5]$，i和j又各增加1。这时，又出现了$A[i+1]\neq B[j+1]$的情况：

```
i = 1 2 3 4 5 6 7 8 9 ……
A = a b a b a b a a b a b …
B =     a b a b a c b
j =     1 2 3 4 5 6 7
```

由于$P[5]=3$，因此新的$j=3$：

```
i = 1 2 3 4 5 6 7 8 9 ……
A = a b a b a b a a b a b …
B =       a b a b a c b
j =       1 2 3 4 5 6 7
```

这时，新的$j=3$仍然不能满足$A[i+1]=B[j+1]$，此时我们再次减小j值，将j再次更新为$P[3]$：

```
i = 1 2 3 4 5 6 7 8 9 ……
A = a b a b a b a a b a b …
B =             a b a b a c b
j =             1 2 3 4 5 6 7
```

现在，i 还是 7，j 已经变成 1 了。而此时 $A[i+1]$ 仍然不等于 $B[j+1]$。这样，j 必须减小到 $P[1]$，即 0：

```
i = 1 2 3 4 5 6 7 8 9 ……
A = a b a b a b a a b a b …
B =               a b a b a c b
j =             0 1 2 3 4 5 6 7
```

终于，$A[8]=B[1]$，i 变为 8，j 为 1。事实上，有可能 j 到了 0 仍然不能满足 $A[i+1]=B[j+1]$(比如 $A[8]=d$ 时)。因此，准确的说法是，当 $j=0$ 时，我们增加 i 值但忽略 j 直到出现 $A[i]=B[1]$ 为止。

这个过程的代码很短，我们在这里给出：

```
j = 0;
for (int i = 1; i <= n; ++i) {
  while (j > 0 && B[j + 1] != A[i]) j = P[j];
  if (B[j + 1] == A[i]) ++j;
  if (j == m) {
    printf("%d\n", i - m + 1);
    j = p[j];
  }
}
```

最后的 $j=P[j]$ 是为了让程序继续做下去，因为我们有可能找到多处匹配。

这个程序或许比想像中的要简单，因为对于 i 值的不断增加，代码用的是 for 循环。因此，这个代码可以这样形象地理解：扫描字符串 A，并更新可以匹配到 B 的什么位置。

现在，我们还遗留了两个重要的问题：一，为什么这个程序是线性的；二，如何快速预处理 P 数组。实际上 P 数组也就是我们常说的 next 数组。

为什么这个程序是$O(n)$的？其实，主要的争议在于，while 循环使得执行次数出现了不确定因素。我们将用到时间复杂度的摊还分析中的主要策略，简单地说就是通过观察某一个变量或函数值的变化来对零散的、杂乱的、不规则的执行次数进行累计。KMP 的时间复杂度分析可谓摊还分析的典型。我们从上述程序的 j 值入手。每一次执行 while 循环都会使 j 减小（但不能减成负的），而另外的改变 j 值的地方只有一处。每次执行了这一处，j 都只能加 1；因此，整个过程中 j 最多加了 n 个 1。于是，j 最多只有 n 次减小的机会（j 值减小的次数当然不能超过 n，因为 j 永远是非负整数）。这告诉我们，while 循环总共最多执行了 n 次。按照摊还分析的说法，平摊到每次 for 循环中后，一次 for 循环的复杂度为 $O(1)$。整个过程显然是 $O(n)$的。这样的分析对于后面 P 数组预处理的过程同样有效，同样可以得到预处理过程的复杂度为 $O(m)$.

预处理不需要按照 P 的定义写成 $O(m^2)$ 甚至 $O(m^3)$ 的。我们可以通过 $P[1], P[2], \cdots, P[j-1]$的值来获得 $P[j]$的值。对于刚才的 $B = ababacb$，假如我们已经求出了 $P[1], P[2], P[3], P[4]$，看看我们应该怎么求出 $P[5], P[6]$。$P[4]=2$，那么 $P[5]$显然等于 $P[4]+1$，因为由 $P[4]$可以知道，$B[1\ldots2]$已经和 $B[3\ldots4]$相等了，现在又有 $B[3]=B[5]$，所以 $P[5]$可以由 $P[4]$后面加一个字符得到。$P[6]$ 也等于 $P[5]+1$ 吗？显然不是，因为 $B[P[5]+1]\neq B[6]$。那么，我们要考虑“退一步”了。我们考虑 $P[6]$是否有可能由 $P[5]$的情况所包含的子串得到，即是否 $P[6]=P[P[5]]+1$。这里想不通的话可以仔细看一下：

```
    1 2 3 4 5 6 7
B = a b a b a c b
P = 0 0 1 2 3 ?
```

$P[5]=3$是因为 $B[1\ldots3]$和 $B[3\ldots5]$都是 aba；而 $P[3]=1$则告诉我们，$B[1], B[3], B[5]$都是 a。既然 $P[6]$不能由 $P[5]$得到，或许可以由 $P[3]$得到（如果 $B[2]$恰好和 $B[6]$相等的话，$P[6]$就等于 $P[3]+1$了）。显然，$P[6]$也不能通过 $P[3]$得到，因为 $B[2]\neq B[6]$。事实上，这样一直推到 $P[1]$也不行，最后，我们得到 $P[6]=0$.

怎么这个预处理过程跟前面的 KMP 主程序这么像呢？其实，KMP 的预处理本身就是一个 B 串“自我匹配”的过程。它的代码和上面的代码神似：

```
P[1] = 0;
j = 0;
for (int i = 2; i <= m; ++i) {
  while (j > 0 && B[j + 1] != B[i]) j = P[j];
  if (B[j + 1] == B[i]) ++j;
  P[i] = j;
}
```

最后补充一点：由于 KMP 算法只预处理 B 串，因此这种算法很适合这样的问题：给定一个 B 串和一群不同的 A 串，问 B 是哪些 A 串的子串。

三、例题

Oulipo(POJ3461)

题目大意：

给定两个串 S,T，求 S 在 T 中的出现次数。

|S| <= 10^4 , |T| <= 10^6

分析：

KMP 模板题，找出 S 在 T 中出现的所有位置，自然也就能统计次数了。

代码：

```
#include <cstdio>
#include <cstring>

const int N = 1e4 + 5;
const int M = 1e6 + 5;
int T, n, m, nxt[N];
char s[N], t[M];

int main() {
  scanf("%d", &T);
  while (T--) {
    scanf("%s%s", s + 1, t + 1);
    n = strlen(s + 1);
    m = strlen(t + 1);
    for (int i = 2, j = 0; i <= n; ++i) {
      while (j > 0 && s[j + 1] != s[i]) j = nxt[j];
      if (s[j + 1] == s[i]) ++j;
```

```
      nxt[i] = j;
    }

    int ans = 0;
    for (int i = 1, j = 0; i <= m; ++i) {
      while (j > 0 && s[j + 1] != t[i]) j = nxt[j];
      if (s[j + 1] == t[i]) ++j;
      if (j == n) ++ans, j = nxt[j];
    }
    printf("%d\n", ans);
  }
  return 0;
}
```

练习题

POJ2406 ; CF 526D

上述内容参考：

1.http://www.matrix67.com/blog/archives/115

字典树

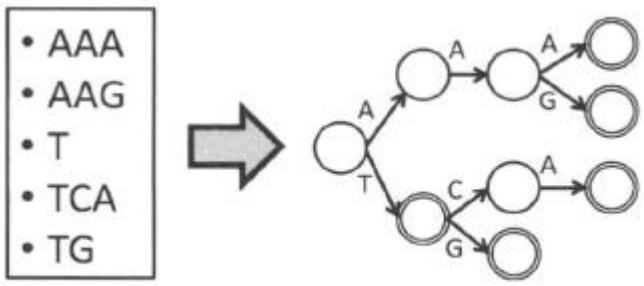
一、简介

字典树，也称 Trie、字母树，指的是某个字符串集合对应的形如下图的有根树。树的每条边上对应有恰好一个字符，每个顶点代表从根到该节点的路径所对应的字符串（将所有经过的边上的字符按顺序连接起来）。有时我们也称 Trie 上的边为转移，顶点为状态。

顶点上还能存储额外的信息，例如下图中双圈圈表示顶点所代表的字符串是实际字符串集合中的元素。而实际字符串中的元素都会对应 Trie 中某个顶点代表的串。实际上，任意一个节点所代表的字符串，都是实际字符串集合中某些串的前缀，特别地，根节点表示空串。

此外，对于任意一个节点，它到它儿子节点边上的字符都互不相同。可以发现，Trie 很好的利用了串的公共前缀，节约了存储空间。

若将字符集看做是小写英文字母，则 Trie 也可以看做是一个 26 叉树，在插入询问新字符串时与树一样，找到对应的边往下走。



Trie

二、操作实现

字典树主要有两种操作，插入和查询，这两种操作都非常简单，用一个一重循环均可解决，即第 i 次循环时找到前 i 个字符所对应的节点，然后进行相应的操作。

Trie 的节点可以使用一个结构体进行存储，如下代码中，$trans[i]$ 表示这个节点边上字符为 i 的转移边所到达的儿子节点编号，若为 0 则表示没有这个儿子。

```
struct node {
  int trans[Z];  // Z 为字符集大小
  bool bo;       // 若 bo=true 则表示这个顶点代表的字符串是实际字符串集合中的元素
} tr[N];
```

现在要对一个字符集为小写英文字母的 Trie 插入一个字符串 S：

```
void insert(char *s) {
  int len = strlen(s);
  int u = 1;                              // 1 为根节点
  for (int i = 0; i < len; ++i) {
    if (!tr[u].trans[s[i] - 'a'])      // 若不存在这条边则要新建一个节点与转移边
      tr[u].trans[s[i] - 'a'] = ++tot; // tot 为总点数
    u = tr[u].trans[s[i] - 'a'];
  }
  tr[u].bo = true;  // 在串的结尾处将 bo 赋值，表示它代表一个实际字符串集合中的元素
}
```

查询一个字符串 S 是否是给定字符串集合中某个串的前缀：

```
bool insert(char *s) {
  int len = strlen(s);
  int u = 1;
  for (int i = 0; i < len; ++i) {
    if (!tr[u].trans[s[i] - 'a']) return false;
    u = tr[u].trans[s[i] - 'a'];
  }
  return true;
}
```

将一个字符串集合构成一棵 Trie 的复杂度为 $O(\sum L)$，$\sum L$ 为所有字符串的总长。查询一个串 S 的时间复杂度则为 $O(|S|)$.

三、例题

Phone List (POJ3630)

题目大意：

给定 n 个长度不超过 10 的数字串，问其中是否存在两个数字串 S,T，使得 S 是 T 的前缀。多组数据，数据组数不超过 40.

n <= 10^4

分析：

考虑将所有的字符串构成一棵 Trie，在构建过程中可以顺便判断答案。若当前串插入后没有新建任何节点，则当前串肯定是之前插入的某个串的前缀；若插入过程中，有某个经过

的节点带有串结尾标记，则之前插入的某个串是当前串的前缀。依据上面两种情况判断答案。

代码：

```
#include <cstdio>
#include <cstring>

const int N = 1e5 + 5;
const int Z = 10;

int T, n, tot;
struct node {
  int trans[Z];
  bool bo;
  void clear() {
    memset(trans, 0, sizeof(trans));
    bo = false;
  }
} tr[N];
char s[20];

bool insert(char *s) {
  int len = strlen(s);
  int u = 1;
  bool flag = false;
  for (int i = 0; i < len; ++i) {
    if (!tr[u].trans[s[i] - '0'])
      tr[tr[u].trans[s[i] - '0'] = ++tot].clear();
    else if (i == len - 1)
      flag = true;
    u = tr[u].trans[s[i] - '0'];
    if (tr[u].bo) flag = true;
  }
  tr[u].bo = true;
  return flag;
}

int main() {
  scanf("%d", &T);
  while (T--) {
    scanf("%d", &n);
    tr[tot = 1].clear();
    bool ans = false;
    for (int i = 1; i <= n; ++i) {
      scanf("%s", s);
```

```
42.       if (insert(s)) ans = true;
43.     }
44.     if (!ans) puts("YES");
45.     else puts("NO");
46.   }
47.   return 0;
48. }
```

练习题

POJ 2945 ; POJ 1816 ; CF 633C

上述内容参考：

1.挑战程序设计竞赛（第二版）

AC 自动机

一、简介

与 KMP 类似，AC 自动机也是用来处理字符串匹配的问题。与 KMP 不同的是，KMP 用来处理单模式串问题，即问模式串 T 是否是其他主串 S_i 的子串，而 AC 自动机则能处理多模式串的问题。AC 自动机处理的常见问题如：给出 n 个单词 T_i，再给出一段包含 m 个字符的文章 S，问有多少个单词在文章里出现了。

构建一个 AC 自动机并用于匹配需要三个步骤：将所有的模式串构建成一棵 Trie，对 Trie 上所有的节点构造失败指针(fail)，利用失败指针对主串进行模式匹配。实际上这个失败指针与 KMP 算法中的 next 数组非常相似，因而 AC 自动机可以看做是 Trie 与 KMP 算法的结合（Trie 上的 KMP 算法）。

我们将模式串所构出的 Trie 的节点称为 AC 自动机的节点（状态），Trie 中的边称为 AC 自动机中的边（转移），将失配指针所对应的节点称为失配转移。

二、算法流程

若我们没有构建出失败指针，则 AC 自动机就是一个普通的 Trie，而用 Trie 完成上面的单词查询问题，我们需要对于文章的每一个位置 i 开始，将 $S[i\dots m]$ 视作一个字符串，在 Trie 中进行查询，将所有所到达的节点进行标记，最后统计那些代表模式串的节点的标记总个数。

这与 KMP 中的暴力算法非常相似，我们回忆 KMP 是通过寻找模式串 B 中最大的 j，使得在当前位置 i 下 $B[1\dots j]=B[i-j+1\dots i]$，使下次尝试匹配的位置变为 j，减少明显无用的尝试来优化复杂度的。AC 自动机也可以借鉴这个思想。

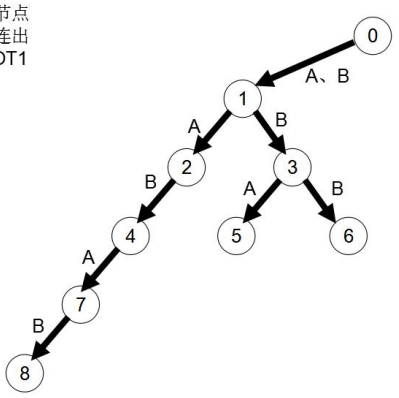
现在匹配是在 Trie 上进行的，因此当我们匹配到 Trie 中某个节点 u 时，它代表的是模式串中某个（些）串的前缀 $T[1\dots j]$，同时也是主串的一段子串 $S[i-j+1\dots i]$，而 j 其实就是 u 在 Trie 中的深度。当在 u 匹配失败（即失配）时，我们需要找到另一个串（也可能是自己本身，在 KMP 中由于只有一个模式串，所以每次它只能找自己），使得它有尽量长的前缀与 u 所代表的串的后缀相等，即 $T'[1\dots k]=T[j-k+1\dots j]$，且 $k<j$，而 $T'[1\dots k]$ 也肯定对应着 Trie 中的某个节点 v，因此我们将 u 的失败指针指向 v，就想 KMP 中的 next 数组

那样，当失配后（Trie 中对应字符的边不存在），下次尝试匹配的位置就换为 v 节点。

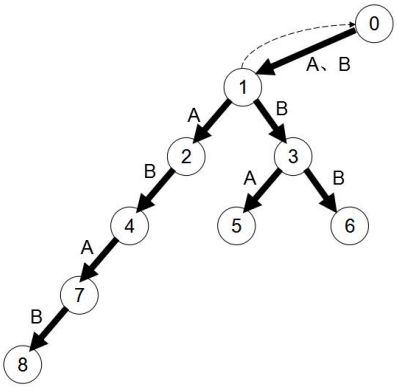
显然这个 v 节点的深度是小于 u 节点的，因此我们可以通过按节点的深度大小，也就是 BFS 的顺序来构建失配指针，构建过程与 KMP 类似，KMP 中 i 的 next 是沿着 $i-1$ 的 next 不停往前跳来求，而 AC 自动机中 u 的 fail 则通过父节点的 fail 不停往上跳，直到找到一个节点它拥有对应字符的转移边为止来求得。

下面对一张 Trie 图求 fail 指针（即下述的前缀指针）：

定义虚拟节点0号节点，0号节点的所有连出的字边都连向ROOT1号节点



ROOT1号节点的前缀指针指向0号节点

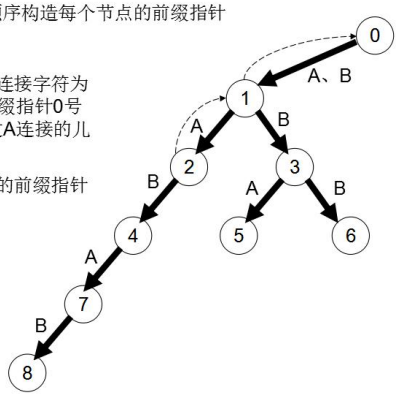


接下来按照BFS顺序构造每个节点的前缀指针

2号节点：

父亲是1号节点，连接字符为A，查找父亲的前缀指针0号节点，是否有通过A连接的儿子。

有！于是2号节点的前缀指针指向1号节点

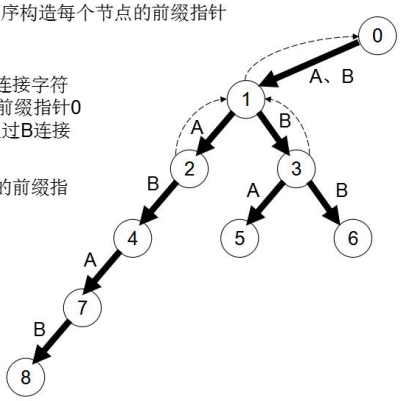


接下来按照BFS顺序构造每个节点的前缀指针

3号节点：

父亲是1号节点，连接字符为B，查找父亲的前缀指针0号节点，是否有通过B连接的儿子。

有！于是3号节点的前缀指针指向1号节点

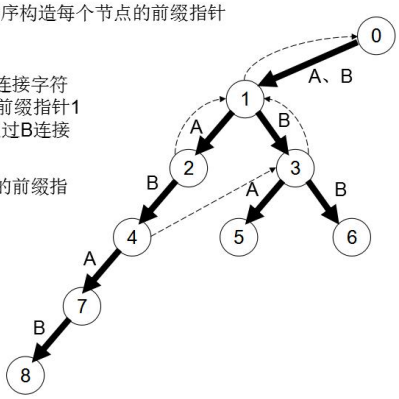


接下来按照BFS顺序构造每个节点的前缀指针

4号节点：

父亲是2号节点，连接字符为B，查找父亲的前缀指针1号节点，是否有通过B连接的儿子。

有！于是4号节点的前缀指针指向3号节点

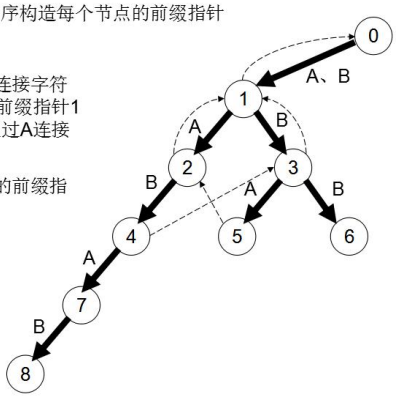


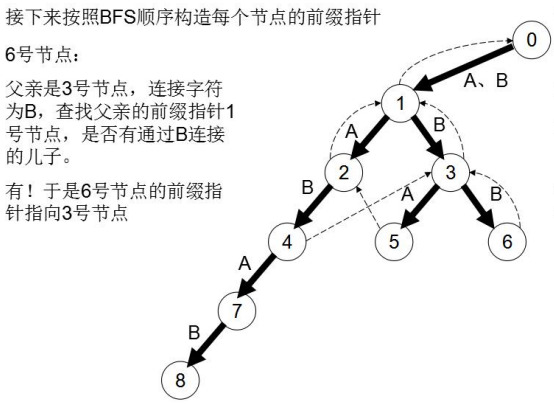
接下来按照BFS顺序构造每个节点的前缀指针

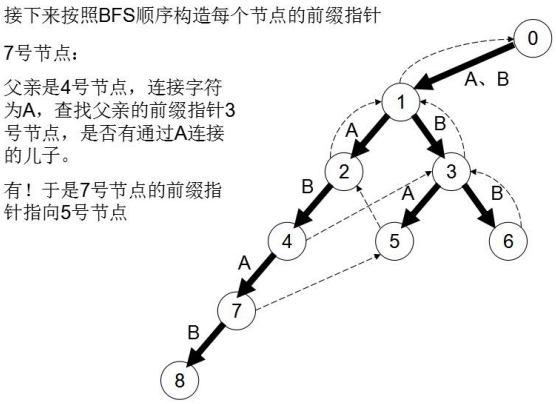
5号节点：

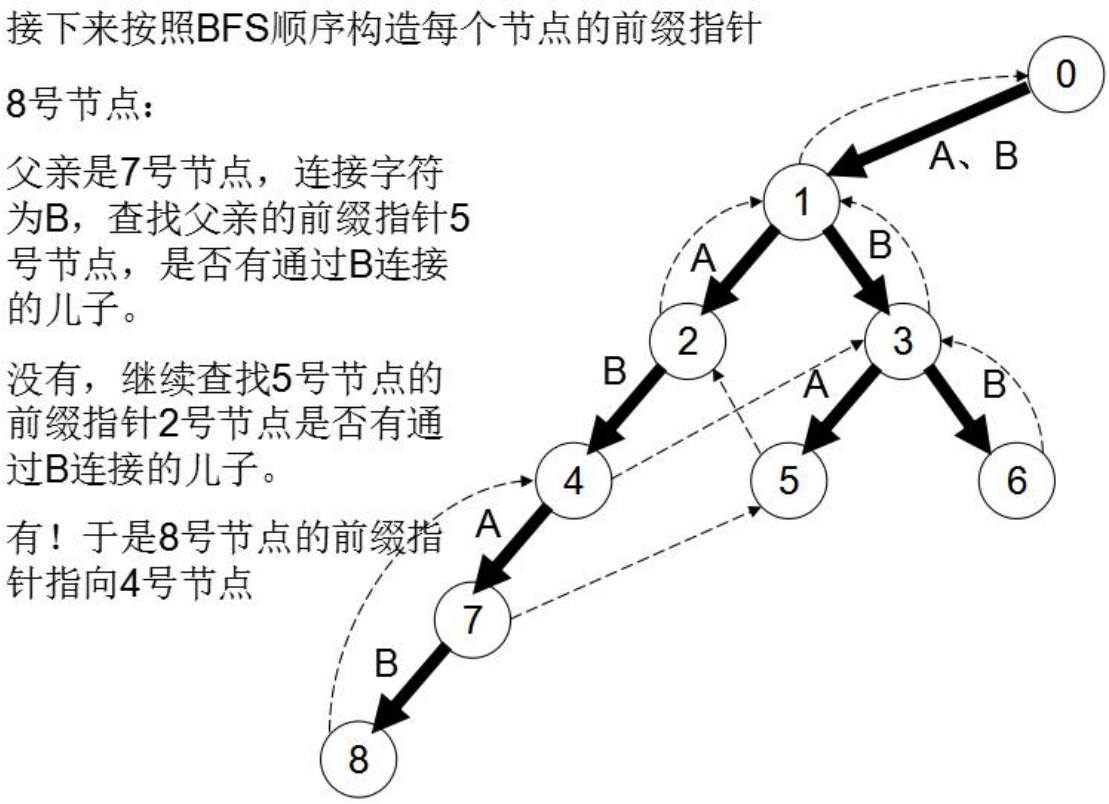
父亲是3号节点，连接字符为A，查找父亲的前缀指针1号节点，是否有通过A连接的儿子。

有！于是5号节点的前缀指针指向2号节点









三、操作实现

首先是 AC 自动机结构体：

```
struct node {
  int cnt, fail;        // cnt 表示有多少个字符串以这个节点为结尾
  int trans[26];
  void init() {
    cnt = fail = 0;
    memset(trans, 0, sizeof(trans));
  }
} tr[N * L];    // N 是串个数，L 是单串长度
```

构建 AC 自动机的 Trie 与普通 Trie 无异：

```
void insert(char *s) {
  int len = strlen(s);
  int u = 1;
  for (int i = 0; i < len; ++i) {
    if (!tr[u].trans[s[i] - 'a'])
      tr[tr[u].trans[s[i] - 'a'] = ++tot].init();
    u = tr[u].trans[s[i] - 'a'];
  }
  ++tr[u].cnt;
}
```

构建 fail 指针：

```
void buildFail() {
  for (int i = 0; i < 26; ++i) tr[0].trans[i] = 1;
  // 为了方便将 0 的所有边都设为根节点 1，且 1 的 fail 为 0
  static int qn, que[N * L];
  que[qn = 1] = 1;
  for (int ql = 1; ql <= qn; ++ql) {
    int u = que[ql], v, w;
    for (int i = 0; i < 26; ++i) {
      v = tr[u].fail;
      while (!tr[v].trans[i]) v = tr[v].fail;
      v = tr[v].trans[i], w = tr[u].trans[i];
      if (w) tr[w].fail = v, que[++qn] = w;    // 若有这个儿子则将其加入队列中
      else tr[u].trans[i] = v;
      // 小优化，若没有对应儿子则直接将转移边指向 v，这样下次要匹配 i 这条边时能直接用
      trans 进行转移，而不需要再不停地跳 fail 指针进行尝试匹配
    }
  }
}
```

匹配：

```
scanf("%s", s);
int len = strlen(s);
int now = 1;
for (int i = 0; i < len; ++i) now = tr[now].trans[s[i] - 'a'];
```

四、例题

Keywords Search (HDU2222)

题目大意：

给定 n 个长度不超过 50 的由小写英文字母组成的单词，以及一篇长为 m 的文章，问有多少个单词在文章中出现了，多组数据。

n <= 10^4 , m <= 10^6

分析：

AC 自动机模板题，注意统计答案时，每个节点只能统计一次不要重复统计。

代码：

```
#include <queue>
#include <cstdio>
#include <cstring>

const int N = 10000 + 3;
const int M = 1e6 + 3;
const int L = 50 + 3;

int T, tot, n, vst[N * L];
struct node {
  int cnt, fail;
  int trans[26];
  void init() {
    cnt = fail = 0;
    memset(trans, 0, sizeof(trans));
  }
} tr[N * L];
char s[M];

void insert(char *s) {
  int len = strlen(s);
  int u = 1;
  for (int i = 0; i < len; ++i) {
    if (!tr[u].trans[s[i] - 'a'])
      tr[tr[u].trans[s[i] - 'a'] = ++tot].init();
    u = tr[u].trans[s[i] - 'a'];
  }
  ++tr[u].cnt;
}

void buildFail() {
  static int qn, que[N * L];
  que[qn = 1] = 1;
  for (int ql = 1; ql <= qn; ++ql) {
```

```
    int u = que[ql], v, w;
    for (int i = 0; i < 26; ++i) {
      v = tr[u].fail;
      while (!tr[v].trans[i]) v = tr[v].fail;
      v = tr[v].trans[i], w = tr[u].trans[i];
      if (w) tr[w].fail = v, que[++qn] = w;
      else tr[u].trans[i] = v;
    }
  }
}

int main() {
  for (int i = 0; i < 26; ++i) tr[0].trans[i] = 1;

  scanf("%d", &T);
  for (int tt = 1; tt <= T; ++tt) {
    tr[tot = 1].init();
    scanf("%d", &n);
    for (int i = 1; i <= n; ++i) {
      scanf("%s", s);
      insert(s);
    }
    buildFail();

    scanf("%s", s);
    int len = strlen(s);
    int now = 1, tmp, ans = 0;
    for (int i = 0; i < len; ++i) {
      now = tr[now].trans[s[i] - 'a'];
      tmp = now;
      while (tmp && vst[tmp] != tt) {
        vst[tmp] = tt;
        ans += tr[tmp].cnt;
        tmp = tr[tmp].fail;
      }
    }
    printf("%d\n", ans);
  }
  return 0;
}
```

练习题

POJ 1204 ; POJ 4052(题目在 POJ 4044 上下载) ; HDU 3065

Manacher

一、基本概念

字符的非空有限集，称为**字母表**(alphabet).

字母表中字符的有限序列，称为**字符串**(string).

字符串中字符的个数，称为该字符串的**长度**(length).

字符串中连续的一段，称为该字符串的**子串**(substring).

字符串中反转后与反转前相同的子串，称为该字符串的**回文子串**(palindromic substring).

字符串中长度最大的回文子串，称为该字符串的**最长回文子串**(longest palindromic substring).

以字符串第 i 位为中心的回文子串的最大长度的一半，称为该字符串第 i 位的**回文半径**(radius of palindrome).

在处理回文子串问题时，一般需要求出字符串每一位的回文半径。

为了避免奇偶讨论和边界问题，从而降低代码复杂度，我们在字符串每一位两侧都添加同一个特殊字符，在字符串的首位前添加一个不同的特殊字符，在字符串的末位后再添加一个不同的特殊字符，如将 abbabcba 变为$#a#b#b#a#b#c#b#a#@.由此我们得到了一个长度为 n 的新字符串 S[1...n]，而要求出该字符串每一位的回文半径 R[1...n].

Table 1: S 和 R 的对应关系

S	#	a	#	b	#	b	#	a	#	b	#	c	#	b	#	a	#
R	1	2	1	2	5	2	1	4	1	2	1	6	1	2	1	2	1

二、Manacher 算法流程

在 Manacher 算法中，我们添加两个辅助变量 mx 和 p，分别表示已有回文半径覆盖到的最右边界和对应中心的位置。

计算 R[i]时，我们先给它一个下界，令 i 关于 p 的对称点 j = 2p - i，分以下三种情况讨论(示意图中，上方线段描述了字符串，中间线段描述了 R[p]，左下方线段描述了 R[j]，右下方线段描述了 R[i]的下界):

(1) mx < i 时，只有 R[i] >= 1

(2) mx − i > R[j] 时，以第 j 位为中心的回文子串包含于以第 p 位为中心的回文子串，由于 i 和 j 关于 p 对称，以第 i 位为中心的回文子串必然也包含于以第 p 位为中心的回文子串，故有 R[i] = R[j]

(3) mx − i <= R[j] 时，以第 j 位为中心的回文子串不一定包含于以第 p 位为中心的回文子串，但由于 i 和 j 关于 p 对称，以第 i 位为中心的回文子串向右至少会扩展到 mx 的位置，故有 R[i] >= mx − i

Figure 1: $mx < i$的情况

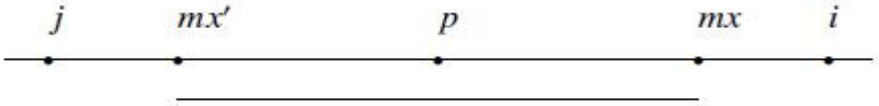


Figure 2: $mx - i > R[j]$的情况

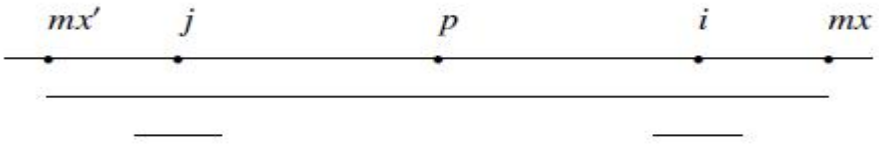
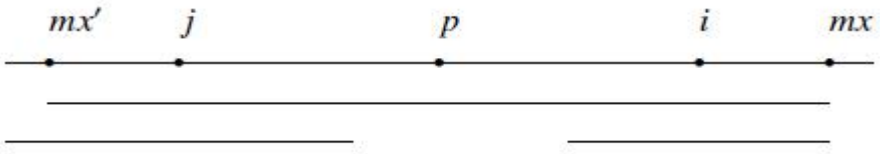


Figure 3: $mx - i \leq R[j]$的情况



对于超过下界的部分，我们只要逐位匹配就可以了。

由于逐位匹配成功必然导致 mx 右移，而 mx 右移不超过 n 次，故 Manacher 算法的时间复杂度为 O(n).

伪代码：

```
MANACHER'S-ALGORITHM(S)
 1  p ← 0
 2  mx ← 0
 3  for i ← 1 to n
 4       do if mx > i
 5            then R[i] ← MIN(R[2p − i], mx − i)
 6            else  R[i] ← 1
 7          while S[i + R[i]] = S[i − R[i]]
 8             do R[i] ← R[i] + 1
 9          if i + R[i] > mx
10            then p ← i
11                 mx ← i + R[i]
12  return R
```

三、例题

Palindrome (POJ 3974)

题目大意：

给定一个长为 n 的字符串，求它的最长回文子串。数据组数不超过 30.

n <= 10^6

分析：

manacher 模板题

代码：

```cpp
#include <cstdio>
#include <cstring>
#include <algorithm>

const int N = 1e6 + 5;
int n, m, T, p[N * 2];
char s[N * 2], t[N];

void manacher() {
  int id = 0, pos = 0, x = 0;
  for (int i = 1; i <= n; ++i) {
```

```
    if (pos > i) x = std::min(p[id * 2 - i], pos - i);
    else x = 1;
    while (s[i - x] == s[i + x]) ++x;
    if (i + x > pos) pos = i + x, id = i;
    p[i] = x;
  }
}

int main() {
  while (scanf("%s", t + 1), t[1] != 'E') {
    m = strlen(t + 1);
    s[n = 0] = '!';
    for (int i = 1; i <= m; ++i) {
      s[++n] = '#';
      s[++n] = t[i];
    }
    s[++n] = '#';
    s[n + 1] = '?';
    manacher();

    int ans = 0;
    for (int i = 1; i <= n; ++i)
      if (p[i] > ans) ans = p[i];
    printf("Case %d: %d\n", ++T, ans - 1);
  }
  return 0;
}
```

练习题

BZOJ 2565 ; CF 17E

上述内容参考

1.2014 年中国国家集训队论文集，浅谈回文子串问题，徐毅。