

# AC 自动机

## 概述

AC (Aho-Corasick) 自动机是 **以 Trie 的结构为基础**，结合 **KMP 的思想** 建立的自动机，用于解决多模式匹配等任务。

AC 自动机本质上是 Trie 上的自动机。

在阅读本文之前，请先阅读 [KMP](#) 和 [Trie](#)。

## 解释

简单来说，建立一个 AC 自动机有两个步骤：

1. 基础的 Trie 结构：将所有的模式串构成一棵 Trie；
2. KMP 的思想：对 Trie 树上所有的结点构造失配指针。

建立完毕后，就可以利用它进行多模式匹配。

## 字典树构建

AC 自动机在初始时会将若干个模式串插入到一个 Trie 里，然后在 Trie 上建立 AC 自动机。这个 Trie 就是普通的 Trie，按照 Trie 原本的建树方法建树即可。

需要注意的是，Trie 中的结点表示的是某个模式串的前缀。我们在后文也将其称作状态。一个结点表示一个状态，Trie 的边就是状态的转移。

形式化地说，对于若干个模式串  $s_1, s_2 \dots s_n$ ，将它们构建一棵字典树后的所有状态的集合记作  $Q$ 。

## 失配指针

AC 自动机利用一个 fail 指针来辅助多模式串的匹配。

状态  $u$  的 fail 指针指向另一个状态  $v$ ，其中  $v \in Q$ ，且  $v$  是  $u$  的最长后缀（即在若干个后缀状态中取最长的一个作为 fail 指针）。

fail 指针与 [KMP](#) 中的 next 指针相比：

1. 共同点：两者同样是在失配的时候用于跳转的指针。
2. 不同点：next 指针求的是最长 Border（即最长的相同前后缀），而 fail 指针指向所有模式串的前缀中匹配当前状态的最长后缀。

因为 KMP 只对一个模式串做匹配，而 AC 自动机要对多个模式串做匹配。有可能 fail 指针指向的结点对应着另一个模式串，两者前缀不同。

总结下来，AC 自动机的失配指针指向当前状态的最长后缀状态。

注意：AC 自动机在做匹配时，同一位上可匹配多个模式串。

## 构建指针

下面介绍构建 fail 指针的 **基础思想**：

构建 fail 指针，可以参考 KMP 中构造 next 指针的思想。

考虑字典树中当前的结点  $u$ ， $u$  的父结点是  $p$ ， $p$  通过字符  $c$  的边指向  $u$ ，即  $\text{trie}(p, c) = u$ 。假设深度小于  $u$  的所有结点的 fail 指针都已求得。

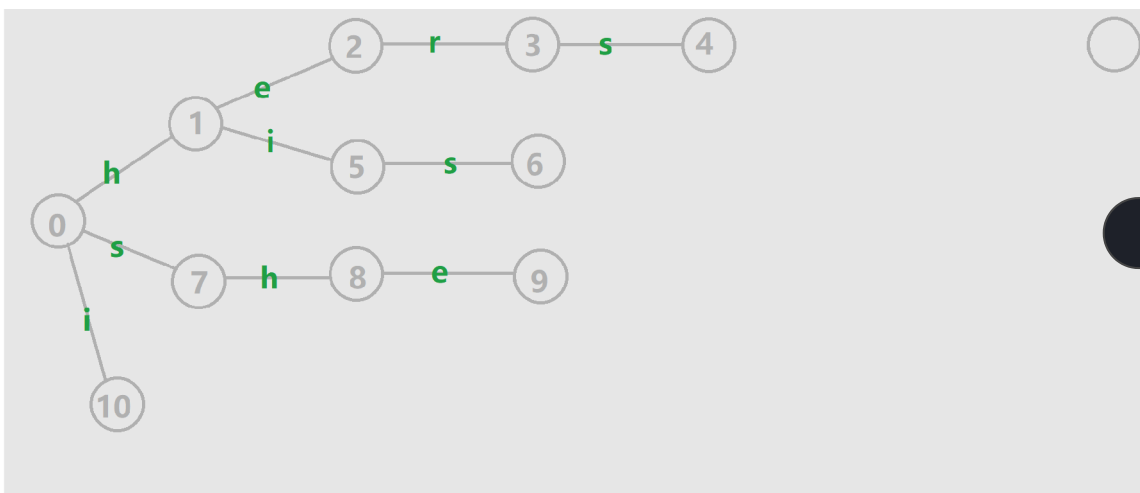
1. 如果  $\text{trie}(\text{fail}(p), c)$  存在：则让  $u$  的 fail 指针指向  $\text{trie}(\text{fail}(p), c)$ 。相当于在  $p$  和  $\text{fail}(p)$  后面加一个字符  $c$ ，分别对应  $u$  和  $\text{fail}(u)$ ；
2. 如果  $\text{trie}(\text{fail}(p), c)$  不存在：那么我们继续找到  $\text{trie}(\text{fail}(\text{fail}(p)), c)$ 。重复判断过程，一直跳 fail 指针直到根结点；
3. 如果依然不存在，就让 fail 指针指向根结点。

如此即完成了  $\text{fail}(u)$  的构建。

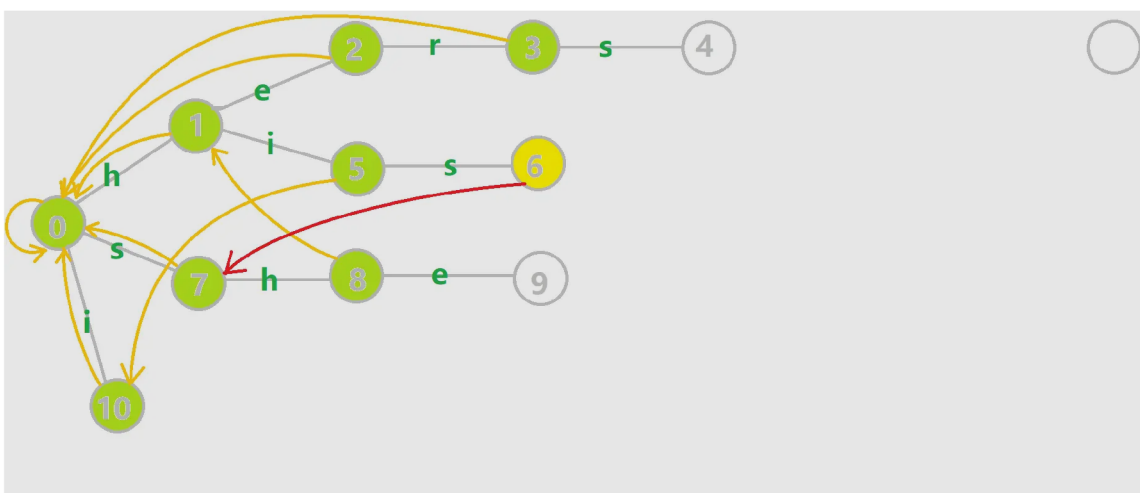
## 例子

下面将使用若干张 GIF 动图来演示对字符串 i、he、his、she、hers 组成的字典树构建 fail 指针的过程：

1. 黄色结点：当前的结点  $u$ 。
2. 绿色结点：表示已经 BFS 遍历完毕的结点。
3. 橙色的边：fail 指针。
4. 红色的边：当前求出的 fail 指针。

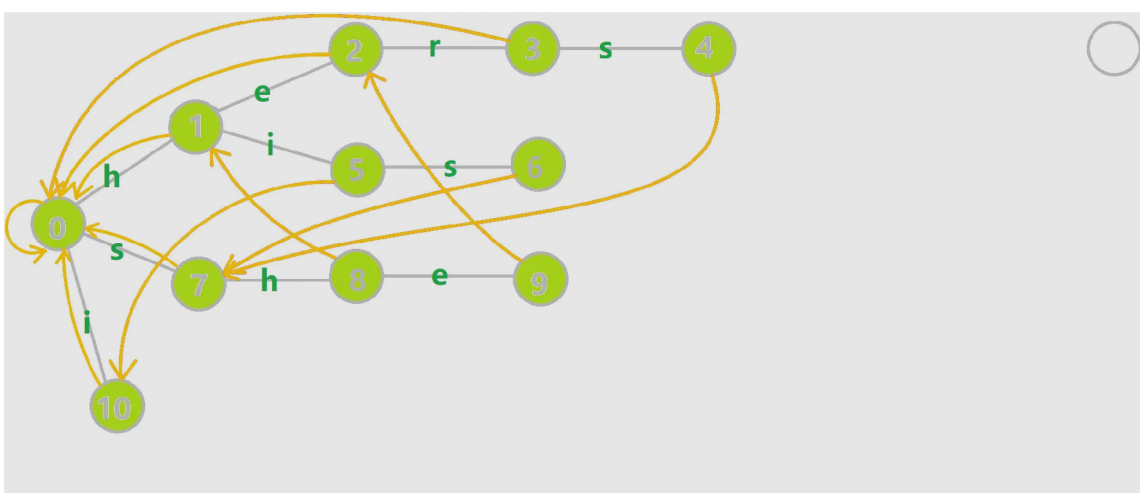


我们重点分析结点 6 的 fail 指针构建：



找到 6 的父结点 5， $\text{fail}(5) = 10$ 。然而结点 10 没有字母 s 连出的边；继续跳到 10 的 fail 指针， $\text{fail}(10) = 0$ 。发现 0 结点有字母 s 连出的边，指向 7 结点；所以  $\text{fail}(6) = 7$ 。

下图展示了构建完毕的状态：



## 字典树与字典图

关注构造函数 `build`，该函数的目标有两个，一个是构建 `fail` 指针，一个是构建自动机。相关变量定义如下：

1. `tr[u].son[c]`：有两种理解方式。我们可以简单理解为字典树上的一条边，即 `trie(u, c)` 也可以理解为从状态（结点）`u` 后加一个字符 `c` 到达的状态（结点），即一个状态转移函数 `trans(u, c)`。为了方便，下文中我们将用第二种理解方式。
2. 队列 `q`：用于 BFS 遍历字典树。
3. `tr[u].fail`：结点 `u` 的 `fail` 指针。

### 实现

#### C++

```
1 void build() {
2     queue<int> q;
3     for (int i = 0; i < 26; i++)
4         if (tr[0].son[i]) q.push(tr[0].son[i]);
5     while (!q.empty()) {
6         int u = q.front();
7         q.pop();
8         for (int i = 0; i < 26; i++) {
9             if (tr[u].son[i]) {
10                 tr[tr[u].son[i]].fail = tr[tr[u].fail].son[i];
11                 q.push(tr[u].son[i]);
12             } else
13                 tr[u].son[i] = tr[tr[u].fail].son[i];
14         }
15     }
16 }
```

#### Python

```
1 def build():
2     for i in range(0, 26):
3         if tr[0][i] != 0:
4             q.append(tr[0][i])
5     while q:
6         u = q.pop(0)
7         for i in range(0, 26):
8             if tr[u][i] != 0:
9                 fail[tr[u][i]] = tr[fail[u]][i]
10                q.append(tr[u][i])
11            else:
12                tr[u][i] = tr[fail[u]][i]
```

## 解释

`build` 函数将结点按 BFS 顺序入队，依次求 `fail` 指针。这里的字典树根结点为 0，我们将根结点的子结点一一入队。若将根结点入队，则在第一次 BFS 的时候，会将根结点儿子的 `fail` 指针标记为本身。因此我们将根结点的儿子一一入队，而不是将根结点入队。

然后开始 BFS：每次取出队首的结点  $u$  (`fail(u)` 在之前的 BFS 过程中已求得)，然后遍历字符集（这里是  $0 \sim 25$ ，对应  $a \sim z$ ，即  $u$  的各个子结点）：

1. 如果 `trans(u, c)` 存在，我们就将 `trans(u, c)` 的 `fail` 指针赋值为 `trans(fail(u), c)`。根据之前的描述，我们应该用 `while` 循环，不停地跳 `fail` 指针，判断是否存在字符  $c$  对应的结点，然后赋值，但此处通过特殊处理简化了这些代码，将在下文说明；
2. 否则，令 `trans(u, c)` 指向 `trans(fail(u), c)` 的状态。

这里的处理是，通过 `else` 语句的代码修改字典树的结构，将不存在的字典树的状态链接到了失配指针的对应状态。在原字典树中，每一个结点代表一个字符串  $S$ ，是某个模式串的前缀。而在修改字典树结构后，尽管增加了许多转移关系，但结点（状态）所代表的字符串是不变的。

而 `trans(S, c)` 相当于是在  $S$  后添加一个字符  $c$  变成另一个状态  $S'$ 。如果  $S'$  存在，说明存在一个模式串的前缀是  $S'$ ，否则我们让 `trans(S, c)` 指向 `trans(fail(S), c)`。由于 `fail(S)` 对应的字符串是  $S$  的后缀，因此 `trans(fail(S), c)` 对应的字符串也是  $S'$  的后缀。

换言之在 Trie 上跳转的时候，我们只会从  $S$  跳转到  $S'$ ，相当于匹配了一个  $S'$ ；但在 AC 自动机上跳转的时候，我们会从  $S$  跳转到  $S'$  的后缀，也就是说我们匹配一个字符  $c$ ，然后舍弃  $S$  的部分前缀。舍弃前缀显然是能匹配的。同时如果文本串能匹配  $S$ ，显然它也能匹配  $S$  的后缀，所以 `fail` 指针同样在舍弃前缀。所谓的 `fail` 指针其实就是  $S$  的一个后缀集合。

Trie 的结点的孩子数组 `son` 还有另一种比较简单的理解方式：如果在位置  $u$  失配，我们会跳转到 `fail(u)` 的位置。注意这会导致我们可能沿着 `fail` 数组跳转多次才能来到下一个能匹配的位置。所以我们可以用 `son` 直接记录记录下一个能匹配的位置，这样保证了程序的时间复杂度。

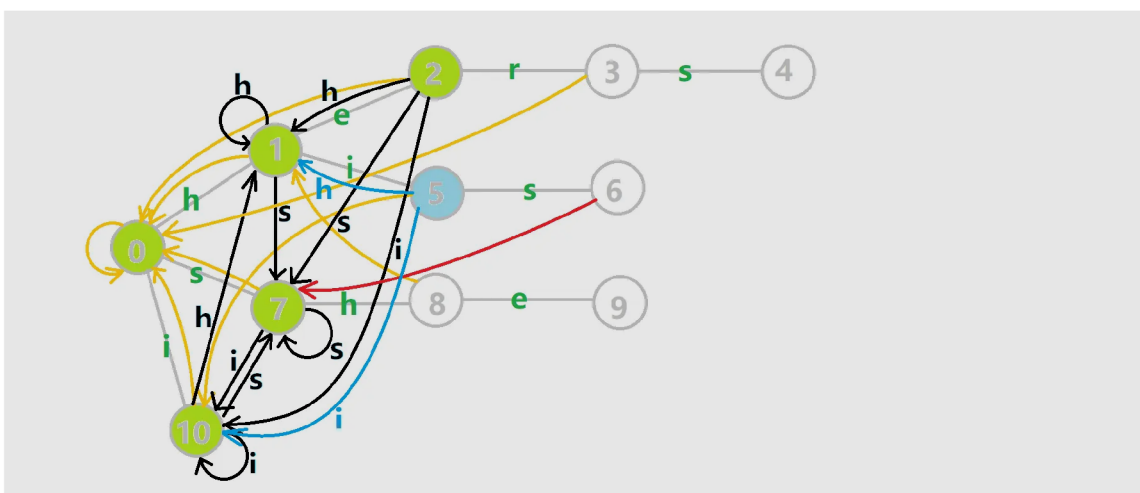
此处对字典树结构的修改，可以使得匹配转移更加完善。同时它将 `fail` 指针跳转的路径做了压缩，使得本来需要跳很多次 `fail` 指针变成跳一次。

## 过程

这里依然用若干张 GIF 动图展示构建过程：



- 可以发现，众多交错的黑色边将字典树变成了 **字典图**。图中省略了连向根结点的黑边（否则会  
更乱）。我们重点分析一下结点 5 遍历时的情况。我们求  $\text{trans}(5, \text{s}) = 6$  的 fail 指针：



这就是 `build` 完成的两件事：构建 fail 指针和建立字典图。这个字典图也会在查询的时候起到关键作用。

## 多模式匹配

接下来分析匹配函数 `query`：

### 实现

#### C++

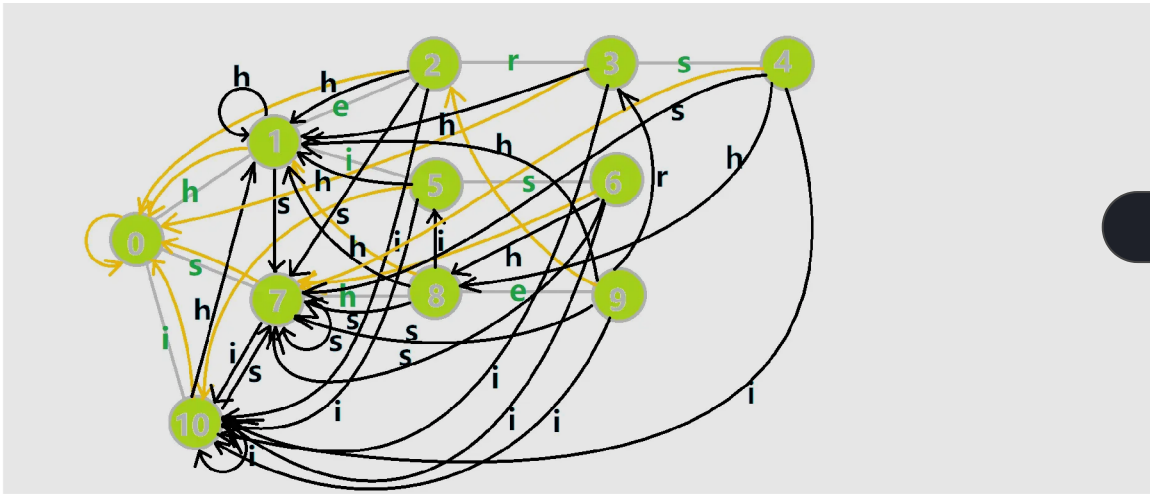
```
1  int query(const char t[]) {
2      int u = 0, res = 0;
3      for (int i = 1; t[i]; i++) {
4          u = tr[u].son[t[i] - 'a'];
5          for (int j = u; j && tr[j].cnt != -1; j = tr[j].fail) {
6              res += tr[j].cnt, tr[j].cnt = -1;
7          }
8      }
9      return res;
10 }
```

#### Python

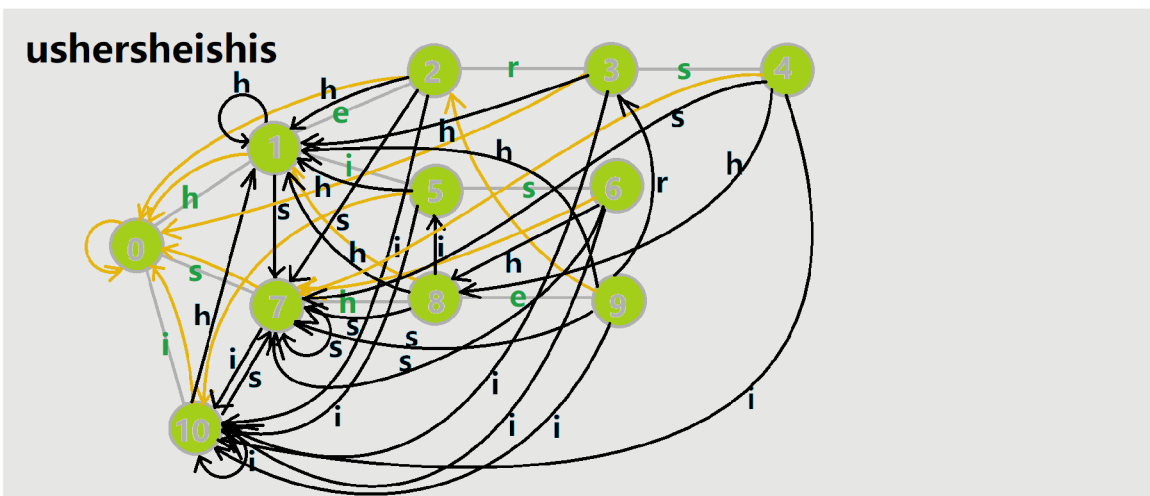
```
1  def query(t: str) -> int:
2      u, res = 0, 0
3      for c in t:
4          u = tr[u][c - ord("a")]
5          j = u
6          while j and e[j] != -1:
7              res += e[j]
8              e[j] = -1
9              j = fail[j]
10     return res
```

### 解释

这里  $u$  作为字典树上当前匹配到的结点，`res` 即返回的答案。循环遍历匹配串， $u$  在字典树上跟踪当前字符。利用 `fail` 指针找出所有匹配的模式串，并累加到答案中。然后将匹配到的串的出现次数清零，这样就不会重复统计同一个串。在上文中我们分析过，字典树的结构其实就是一个 `trans` 函数，而构建好这个函数后，在匹配字符串的过程中，我们会舍弃部分前缀达到最低限度的匹配。`fail` 指针则指向了更多的匹配状态。最后上一份图。对于刚才的自动机：



我们从根结点开始尝试匹配 ushersheishis，那么  $p$  的变化将是：



1. 红色结点： $p$  结点。
2. 粉色箭头： $p$  在自动机上的跳转。
3. 蓝色的边：成功匹配的模式串。
4. 蓝色结点：示跳 fail 指针时的结点（状态）。

## 效率优化

题目请参考洛谷 [P5357](#) 【模板】AC 自动机。

因为我们的 AC 自动机中，每次匹配，会一直向 fail 边跳来找到所有的匹配，但是这样的效率较低，在某些题目中会超时。

那么需要如何优化呢？首先需要了解到 fail 指针的一个性质：一个 AC 自动机中，如果只保留 fail 边，那么剩余的图一定是一棵树。

这是显然的，因为 fail 不会成环，且深度一定比现在低，所以得证。



这样 AC 自动机的匹配就可以转化为在 fail 树上的链求和问题，只需要优化一下该部分就可以了。

这里提供两种思路。

## 拓扑排序优化

观察到时间主要浪费在在每次都要跳 fail。如果我们预先记录，最后一并求和，那么效率就会优化。

于是我们按照 fail 树，做一次内向树上的拓扑排序，就能一次性求出所有模式串的出现次数。

`build` 函数在原先的基础上，增加了入度统计一部分，为拓扑排序做准备。

### 构建

```
1 void build() {
2     queue<int> q;
3     for (int i = 0; i < 26; i++)
4         if (tr[0].son[i]) q.push(tr[0].son[i]);
5     while (!q.empty()) {
6         int u = q.front();
7         q.pop();
8         for (int i = 0; i < 26; i++) {
9             if (tr[u].son[i]) {
10                 tr[tr[u].son[i]].fail = tr[tr[u].fail].son[i];
11                 tr[tr[tr[u].fail].son[i]].du++; // 入度计数
12                 q.push(tr[u].son[i]);
13             } else
14                 tr[u].son[i] = tr[tr[u].fail].son[i];
15         }
16     }
17 }
```

然后我们在查询的时候就可以只为找到结点的 `ans` 打上标记，在最后再用拓扑排序求出答案。

### 查询

```
1 void query(const char t[]) {
2     int u = 0;
3     for (int i = 1; t[i]; i++) {
4         u = tr[u].son[t[i] - 'a'];
5         tr[u].ans++;
6     }
7 }
8
9 void topu() {
10     queue<int> q;
11     for (int i = 0; i <= tot; i++)
12         if (tr[i].du == 0) q.push(i);
13     while (!q.empty()) {
14         int u = q.front();
15         q.pop();
16         ans[tr[u].idx] = tr[u].ans;
17         int v = tr[u].fail;
18         tr[v].ans += tr[u].ans;
19         if (--tr[v].du) q.push(v);
20     }
21 }
```

最后是主函数：

### 主函数

```
1 int main() {
2     // do_something();
3     AC::build();
4     scanf("%s", s + 1);
5     AC::query(s);
6     AC::topu();
7     for (int i = 1; i <= n; i++) printf("%d\n", AC::ans[idx[i]]);
8     // do_another_thing();
9 }
```

## Luogu P5357 【模板】AC 自动机

```
1  #include <cstdio>
2  #include <cstring>
3  #include <queue>
4  using namespace std;
5
6  constexpr int N = 2e5 + 6;
7  constexpr int LEN = 2e6 + 6;
8  constexpr int SIZE = 2e5 + 6;
9
10 int n;
11
12 namespace AC {
13     struct Node {
14         int son[26]; // 子结点
15         int ans;     // 匹配计数
16         int fail;    // fail 指针
17         int du;      // 入度
18         int idx;
19
20         void init() { // 结点初始化
21             memset(son, 0, sizeof(son));
22             ans = fail = idx = 0;
23         }
24     } tr[SIZE];
25
26     int tot; // 结点总数
27     int ans[N], pid;
28
29     void init() {
30         tot = pid = 0;
31         tr[0].init();
32     }
33
34     void insert(char s[], int &idx) {
35         int u = 0;
36         for (int i = 1; s[i]; i++) {
37             int &son = tr[u].son[s[i] - 'a']; // 下一个子结点的引用
38             if (!son) son = ++tot, tr[son].init(); // 如果没有则插入
39             // 新结点，并初始化
40             u = son; // 从下一个结点继
41             // 续
42         }
43         // 由于有可能出现相同的模式串，需要将相同的映射到同一个编号
44         if (!tr[u].idx) tr[u].idx = ++pid; // 第一次出现，新增编号
45         idx = tr[u].idx; // 这个模式串的编号对应这个结点的编号
46     }
47 }
```

```

48 void build() {
49     queue<int> q;
50     for (int i = 0; i < 26; i++)
51         if (tr[0].son[i]) q.push(tr[0].son[i]);
52     while (!q.empty()) {
53         int u = q.front();
54         q.pop();
55         for (int i = 0; i < 26; i++) {
56             if (tr[u].son[i]) { // 存
57                 在对应子结点
58                 tr[tr[u].son[i]].fail = tr[tr[u].fail].son[i]; // 只
59                 用跳一次 fail 指针
60                 tr[tr[tr[u].fail].son[i]].du++; // 入
61                 度计数
62                 q.push(tr[u].son[i]); // 并
63                 加入队列
64             } else
65                 tr[u].son[i] =
66                 tr[tr[u].fail]
67                 .son[i]; // 将不存在的字典树的状态链接到了失配
68                 指针的对应状态
69             }
70         }
71     }
72
73 void query(char t[]) {
74     int u = 0;
75     for (int i = 1; t[i]; i++) {
76         u = tr[u].son[t[i] - 'a']; // 转移
77         tr[u].ans++;
78     }
79 }
80
81 void topu() {
82     queue<int> q;
83     for (int i = 0; i <= tot; i++)
84         if (tr[i].du == 0) q.push(i);
85     while (!q.empty()) {
86         int u = q.front();
87         q.pop();
88         ans[tr[u].idx] = tr[u].ans;
89         int v = tr[u].fail;
90         tr[v].ans += tr[u].ans;
91         if (!--tr[v].du) q.push(v);
92     }
93 }
94 } // namespace AC
95
96 char s[LEN];
97 int idx[N];
98
99 int main() {

```

```

100     AC::init();
101     scanf("%d", &n);
102     for (int i = 1; i <= n; i++) {
103         scanf("%s", s + 1);
104         AC::insert(s, idx[i]);
105         AC::ans[i] = 0;
106     }
107     AC::build();
108     scanf("%s", s + 1);
    AC::query(s);
    AC::topu();
    for (int i = 1; i <= n; i++) {
        printf("%d\n", AC::ans[idx[i]]);
    }
    return 0;
}

```

## DFS 优化

和拓扑排序的思路接近，不过我们使用 DFS 来代替拓扑排序。其实这两种方法本质上是相同的，都是将 fail 树的子树求和。

完整代码请见总结模板 3。

## AC 自动机上 DP

这部分将以 [P2292 \[HNOI2004\] L 语言](#) 为例题讲解。

不难想到一个朴素的思路：建立 AC 自动机，在 AC 自动机上对于所有 fail 指针的子串转移，最后取最大值得到答案。

主要代码如下。若不熟悉代码中的类型定义，可以先看末尾的完整代码：



#### 查询部分主要代码



```
1  int query(const char t[]) {
2      int u = 0, len = strlen(t + 1);
3      for (int i = 1; i <= len; i++) dp[i] = 0;
4      for (int i = 1; i <= len; i++) {
5          u = tr[u].son[t[i] - 'a'];
6          for (int j = u; j; j = tr[j].fail) {
7              if (tr[j].idx && (dp[i - tr[j].depth] || i - tr[j].depth ==
8  0)) {
9                  dp[i] = dp[i - tr[j].depth] + tr[j].depth;
10             }
11         }
12     }
13     int ans = 0;
14     for (int i = 1; i <= len; i++) ans = std::max(ans, dp[i]);
15     return ans;
16 }
```

但是这样的思路复杂度不是线性（因为要跳每个结点的 fail），会在第二个子任务中超时，所以我们需要进行优化。

我们再看看题目的特殊性质，我们发现所有单词的长度只有 20，所以可以想到状态压缩优化。

我们发现，目前的时间瓶颈主要在跳 fail 这一步，如果我们可以将这一步优化到  $O(1)$ ，就可以保证整个问题在严格线性的时间内被解出。

我们可以将前 20 位字母中，可能的子串长度存下来，并压缩到状态中，存在每个子结点中。

那么我们在 build 的时候就可以这么写：

### 构建 fail 指针

```
1 void build() {
2     queue<int> q;
3     for (int i = 0; i < 26; i++)
4         if (tr[0].son[i]) {
5             q.push(tr[0].son[i]);
6             tr[tr[0].son[i]].depth = 1;
7         }
8     while (!q.empty()) {
9         int u = q.front();
10        q.pop();
11        int v = tr[u].fail;
12        // 对状态的更新在这里
13        tr[u].stat = tr[v].stat;
14        if (tr[u].idx) tr[u].stat |= 1 << tr[u].depth;
15        for (int i = 0; i < 26; i++) {
16            if (tr[u].son[i]) {
17                tr[tr[u].son[i]].fail = tr[tr[u].fail].son[i];
18                tr[tr[u].son[i]].depth = tr[u].depth + 1; // 记录深度
19                q.push(tr[u].son[i]);
20            } else
21                tr[u].son[i] = tr[tr[u].fail].son[i];
22        }
23    }
24 }
```

然后查询时就可以去掉跳 fail 的循环，将代码简化如下：

### 查询

```
1 int query(const char t[]) {
2     int u = 0, mx = 0;
3     unsigned st = 1;
4     for (int i = 1; t[i]; i++) {
5         u = tr[u].son[t[i] - 'a'];
6         st <<= 1; // 往下跳了一位每一位的长度都+1
7         if (tr[u].stat & st) st |= 1, mx = i;
8     }
9     return mx;
10 }
```

我们的 `tr[u].stat` 维护的是从结点 `u` 开始，整条 fail 链上的长度集（因为长度集小于 32 所以不影响），而 `st` 则维护的是查询字符串走到现在，前 32 位（因为状态压缩自然溢出）的长度集。

`&` 运算后结果不为 0，则代表两个长度集的交集非空，我们此时就找到了一个匹配。

P2292 [HNOI2004] L 语言

```
1  #include <cstdio>
2  #include <cstring>
3  #include <queue>
4  using namespace std;
5
6  constexpr int N = 20 + 6, M = 50 + 6;
7  constexpr int LEN = 2e6 + 6;
8  constexpr int SIZE = 450 + 6;
9
10 int n, m;
11
12 namespace AC {
13     struct Node {
14         int son[26];
15         int fail;
16         int idx;
17         int depth;
18         unsigned stat;
19
20         void init() {
21             memset(son, 0, sizeof(son));
22             fail = idx = depth = 0;
23         }
24     } tr[SIZE];
25
26     int tot;
27
28     void init() {
29         tot = 0;
30         tr[0].init();
31     }
32
33     void insert(char s[], int idx) {
34         int u = 0;
35         for (int i = 1; s[i]; i++) {
36             int &son = tr[u].son[s[i] - 'a'];
37             if (!son) son = ++tot, tr[son].init();
38             u = son;
39         }
40         tr[u].idx = idx;
41     }
42
43     void build() {
44         queue<int> q;
45         for (int i = 0; i < 26; i++)
46             if (tr[0].son[i])
47                 q.push(tr[0].son[i]);
```



```

48     tr[tr[0].son[i]].depth = 1;
49 }
50 while (!q.empty()) {
51     int u = q.front();
52     q.pop();
53     int v = tr[u].fail;
54     // 对状态的更新在这里
55     tr[u].stat = tr[v].stat;
56     if (tr[u].idx) tr[u].stat |= 1 << tr[u].depth;
57     for (int i = 0; i < 26; i++) {
58         if (tr[u].son[i]) {
59             tr[tr[u].son[i]].fail = tr[tr[u].fail].son[i];
60             tr[tr[u].son[i]].depth = tr[u].depth + 1; // 记录深度
61             q.push(tr[u].son[i]);
62         } else
63             tr[u].son[i] = tr[tr[u].fail].son[i];
64     }
65 }
66 }
67
68 int query(char t[]) {
69     int u = 0, mx = 0;
70     unsigned st = 1;
71     for (int i = 1; t[i]; i++) {
72         u = tr[u].son[t[i] - 'a'];
73         st <= 1;
74         if (tr[u].stat & st) st |= 1, mx = i;
75     }
76     return mx;
77 }
78 } // namespace AC
79
80 char s[LEN];
81
82 int main() {
83     AC::init();
84     scanf("%d%d", &n, &m);
85     for (int i = 1; i <= n; i++) {
86         scanf("%s", s + 1);
87         AC::insert(s, i);
88     }
89     AC::build();
90     for (int i = 1; i <= m; i++) {
91         scanf("%s", s + 1);
92         printf("%d\n", AC::query(s));
93     }
94     return 0;
95 }

```

总结

时间复杂度：定义  $|s_i|$  是模板串的长度， $|S|$  是文本串的长度， $|\Sigma|$  是字符集的大小（常数，一般为 26）。如果连了 trie 图，时间复杂度就是  $O(\sum |s_i| + n|\Sigma| + |S|)$ ，其中  $n$  是 AC 自动机中结点的数目，并且最大可以达到  $O(\sum |s_i|)$ 。如果不连 trie 图，并且在构建 fail 指针的时候避免遍历到空儿子，时间复杂度就是  $O(\sum |s_i| + |S|)$ 。

## Luogu P3808 AC 自动机 (简单版)

```
1  #include <cstdio>
2  #include <cstring>
3  #include <queue>
4  using namespace std;
5
6  constexpr int N = 1e6 + 6;
7  constexpr int LEN = 1e6 + 6;
8  constexpr int SIZE = 1e6 + 6;
9
10 int n;
11
12 namespace AC {
13     struct Node {
14         int son[26]; // 子结点
15         int cnt;     // 尾为该结点的串的个数
16         int fail;    // fail 指针
17
18         void init() { // 结点初始化
19             memset(son, 0, sizeof(son));
20             cnt = fail = 0;
21         }
22     } tr[SIZE];
23
24     int tot; // 结点总数
25
26     void init() {
27         tot = 0;
28         tr[0].init();
29     }
30
31     void insert(char s[]) {
32         int u = 0;
33         for (int i = 1; s[i]; i++) {
34             int &son = tr[u].son[s[i] - 'a']; // 下一个子结点的引用
35             if (!son) son = ++tot, tr[son].init(); // 如果没有则插入新
36             结点, 并初始化
37             u = son; // 从下一个结点继续
38         }
39         tr[u].cnt++;
40     }
41
42     void build() {
43         queue<int> q;
44         for (int i = 0; i < 26; i++)
45             if (tr[0].son[i]) q.push(tr[0].son[i]);
46         while (!q.empty()) {
47             int u = q.front();
```

```

48     q.pop();
49     for (int i = 0; i < 26; i++) {
50         if (tr[u].son[i]) { // 存
51             在对应子结点
52             tr[tr[u].son[i]].fail = tr[tr[u].fail].son[i]; // 只
53             用跳一次 fail 指针
54             q.push(tr[u].son[i]); // 并
55             加入队列
56         } else
57             tr[u].son[i] =
58             tr[tr[u].fail]
59             .son[i]; // 将不存在的字典树的状态链接到了失配指
60             针的对应状态
61         }
62     }
63 }
64
65 int query(char t[]) {
66     int u = 0, res = 0;
67     for (int i = 1; t[i]; i++) {
68         u = tr[u].son[t[i] - 'a']; // 转移
69         for (int j = u; j && tr[j].cnt != -1; j = tr[j].fail) {
70             res += tr[j].cnt, tr[j].cnt = -1;
71         }
72     }
73     return res;
74 }
75 } // namespace AC
76
77 char s[LEN];
78
79 int main() {
80     AC::init();
81     scanf("%d", &n);
82     for (int i = 1; i <= n; i++) {
83         scanf("%s", s + 1);
84         AC::insert(s);
85     }
86     AC::build();
87     scanf("%s", s + 1);
88     printf("%d", AC::query(s));
89     return 0;
90 }

```

## Luogu P3796 AC 自动机 (简单版 II)

```
1  #include <cstdio>
2  #include <cstring>
3  #include <queue>
4  using namespace std;
5
6  constexpr int N = 150 + 6;
7  constexpr int LEN = 1e6 + 6;
8  constexpr int SIZE = N * 70 + 6;
9
10 int n;
11
12 namespace AC {
13     struct Node {
14         int son[26];
15         int fail;
16         int idx;
17
18         void init() {
19             memset(son, 0, sizeof(son));
20             idx = fail = 0;
21         }
22     } tr[SIZE];
23
24     int tot;
25
26     void init() {
27         tot = 0;
28         tr[0].init();
29     }
30
31     void insert(char s[], int idx) { // 将第 idx 个字符串 s 插入
32         int u = 0;
33         for (int i = 1; s[i]; i++) {
34             int &son = tr[u].son[s[i] - 'a'];
35             if (!son) son = ++tot, tr[son].init();
36             u = son;
37         }
38         tr[u].idx = idx;
39     }
40
41     void build() {
42         queue<int> q;
43         for (int i = 0; i < 26; i++)
44             if (tr[0].son[i]) q.push(tr[0].son[i]);
45         while (!q.empty()) {
46             int u = q.front();
47             q.pop();
```

```

48     for (int i = 0; i < 26; i++) {
49         if (tr[u].son[i]) {
50             tr[tr[u].son[i]].fail = tr[tr[u].fail].son[i];
51             q.push(tr[u].son[i]);
52         } else
53             tr[u].son[i] = tr[tr[u].fail].son[i];
54     }
55 }
56 }
57
58 int query(char t[], int cnt[]) {
59     int u = 0, res = 0;
60     for (int i = 1; t[i]; i++) {
61         u = tr[u].son[t[i] - 'a'];
62         for (int j = u; j; j = tr[j].fail)
63             ++cnt[tr[j].idx]; // 统计每个字符串出现的次数
64     }
65     for (int i = 0; i <= tot; ++i)
66         if (tr[i].idx) res = max(res, cnt[tr[i].idx]);
67     return res;
68 }
69 } // namespace AC
70
71 char s[N][75], t[LEN];
72 int cnt[N]; // 每一个字符串出现的次数
73
74 int main() {
75     while (scanf("%d", &n) != EOF && n != 0) {
76         AC::init();
77         for (int i = 1; i <= n; i++) {
78             scanf("%s", s[i] + 1);
79             AC::insert(s[i], i);
80             cnt[i] = 0;
81         }
82         AC::build();
83         scanf("%s", t + 1);
84         int x = AC::query(t, cnt);
85         printf("%d\n", x);
86         for (int i = 1; i <= n; i++)
87             if (cnt[i] == x) printf("%s\n", s[i] + 1);
88     }
89     return 0;
90 }

```

## Luogu P5357 【模板】AC 自动机

```
1  #include <cstdio>
2  #include <cstring>
3  #include <queue>
4  #include <vector>
5  using namespace std;
6
7  constexpr int N = 2e5 + 6;
8  constexpr int LEN = 2e6 + 6;
9  constexpr int SIZE = 2e5 + 6;
10
11 int n;
12
13 namespace AC {
14 struct Node {
15     int son[26];
16     int ans;
17     int fail;
18     int idx;
19
20     void init() {
21         memset(son, 0, sizeof(son));
22         ans = idx = 0;
23     }
24 } tr[SIZE];
25
26 int tot;
27 int ans[N], pidx;
28
29 vector<int> g[SIZE]; // fail 树
30
31 void init() {
32     tot = pidx = 0;
33     tr[0].init();
34 }
35
36 void insert(char s[], int &idx) {
37     int u = 0;
38     for (int i = 1; s[i]; i++) {
39         int &son = tr[u].son[s[i] - 'a'];
40         if (!son) son = ++tot, tr[son].init();
41         u = son;
42     }
43     // 由于有可能出现相同的模式串，需要将相同的映射到同一个编号
44     if (!tr[u].idx) tr[u].idx = ++pidx; // 第一次出现，新增编号
45     idx = tr[u].idx; // 这个模式串的编号对应这个结点的编号
46 }
47
```

```

48 void build() {
49     queue<int> q;
50     for (int i = 0; i < 26; i++)
51         if (tr[0].son[i]) {
52             q.push(tr[0].son[i]);
53             g[0].push_back(tr[0].son[i]); // 不要忘记这里的 fail
54         }
55     while (!q.empty()) {
56         int u = q.front();
57         q.pop();
58         for (int i = 0; i < 26; i++) {
59             if (tr[u].son[i]) {
60                 tr[tr[u].son[i]].fail = tr[tr[u].fail].son[i];
61                 g[tr[tr[u].fail].son[i]].push_back(tr[u].son[i]); //
62 记录 fail 树
63                 q.push(tr[u].son[i]);
64             } else
65                 tr[u].son[i] = tr[tr[u].fail].son[i];
66         }
67     }
68 }
69
70 void query(char t[]) {
71     int u = 0;
72     for (int i = 1; t[i]; i++) {
73         u = tr[u].son[t[i] - 'a'];
74         tr[u].ans++;
75     }
76 }
77
78 void dfs(int u) {
79     for (int v : g[u]) {
80         dfs(v);
81         tr[u].ans += tr[v].ans;
82     }
83     ans[tr[u].idx] = tr[u].ans;
84 }
85 } // namespace AC
86
87 char s[LEN];
88 int idx[N];
89
90 int main() {
91     AC::init();
92     scanf("%d", &n);
93     for (int i = 1; i <= n; i++) {
94         scanf("%s", s + 1);
95         AC::insert(s, idx[i]);
96         AC::ans[i] = 0;
97     }
98     AC::build();
99     scanf("%s", s + 1);

```



```
100     AC::query(s);
101     AC::dfs(0);
102     for (int i = 1; i <= n; i++) {
103         printf("%d\n", AC::ans[idx[i]]);
104     }
105     return 0;
}
```

🔧 本页面最近更新：2025/8/4 22:47:10，[更新历史](#)

✎ 发现错误？想一起完善？ [在 GitHub 上编辑此页！](#)

👤 本页面贡献者：Ir1d, Tiphereth-A, sshwy, ksyx, Marcythm, orzAtalod, Xeonacid, Entertainer, GavinZhengOI, Henry-ZHR, iamtwz, 383494, abc1763613206, aofall, Chrogeek, CoelacanthusHex, Dafenghh, DanJoshua, Gesruea, kenlig, lyccrius, Menci, opsiff, ouuan, partychicken, Persdre, Ruakker, shuzhouliu, StudyingFather, szdytom, XuYueming520, ZXyaang, alphagocc, c-forrest, Early0v0, GoodCoder666, HeRaNO, liangbob2023, qq2964, r-value, rickyxrc, Rickyxrc, shawllew, Unnamed2964, zica87, ZnPdCo

© 本页面的全部内容在 [CC BY-SA 4.0](#) 和 [SATA](#) 协议之条款下提供，附加条款亦可能应用