CSP-S 提高组

字符串算法

ACAM - AC自动机 (Aho-Corasick Automaton)

AC 自动机概述

基本概念

AC 自动机是多模式匹配算法，用于在文本串中同时查找多个模式串

核心思想：

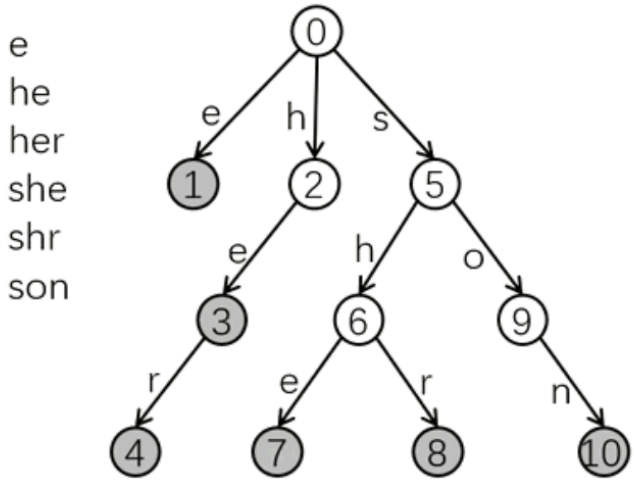
- 结合 KMP 算法的失配指针思想
- 基于 $Trie$ 树结构构建
- 实现高效的多模式匹配

时间复杂度：

- 构建：$O(\sum |P_i|)$，其中 P_i 是模式串
- 匹配：$O(|T| + \text{匹配次数})$

AC自动机 是多模式匹配算法。给定 n个模式串和一个主串，查找有多少个模式串在主串中出现过。

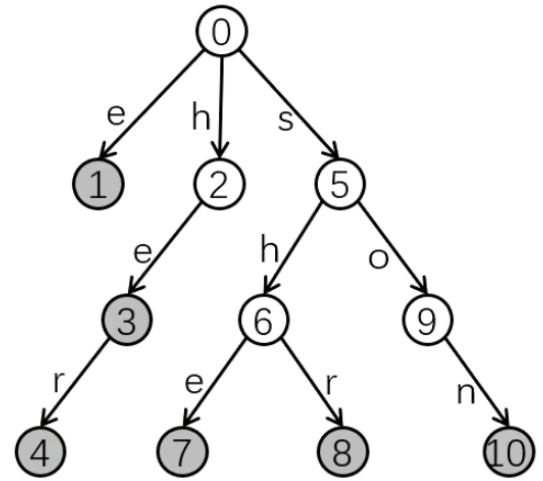
1. **构造 Trie 树**
   我们先用n个模式串构造一颗 Trie 。
   Trie 中的一个**节点**表示一个从根到当前节点的**字符串**。
   根节点表示空串，节点⑤表示“s”，节点⑥表示“sh”，节点⑦表示“she”。
   如果节点是个模式串，则打个标记。例如，cnt[7]=1。
2. **构造 AC自动机**
   在 Trie 上构建两类边：**回跳边**和**转移边**。
3. **扫描主串匹配**



AC自动机 是多模式匹配算法。给定 n个模式串和一个主串，查找有多少个模式串在主串中出现过。

1. **构造 Trie 树**
   我们先用n个模式串构造一颗 Trie 。
   Trie 中的一个**节点**表示一个从根到当前节点的**字符串**。
   根节点表示空串，节点⑤表示“s”，节点⑥表示“sh”，节点⑦表示“she”。
   如果节点是个模式串，则打个标记。例如，cnt[7]=1。
2. **构造 AC自动机**
   在 Trie 上构建两类边：**回跳边**和**转移边**。
3. **扫描主串匹配**

e
he
her
she
shr
son



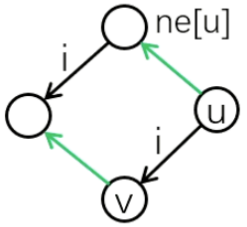

```
int ch[N][26],cnt[N],idx;
int ne[N];

void insert(char *s){//建树
  int p=0;
  for(int i=0;s[i];i++){
    int j=s[i]-'a';
    if(!ch[p][j])ch[p][j]=++idx;
    p=ch[p][j];
  }
  cnt[p]++;
}
```

ne[v] 存节点 v 的**回跳边**的终点。**ne[7]=3**
回跳边指向**父节点的回跳边所指节点的儿子**。
四个点（v, u, ne[u], ch[][]）构成四边形。
回跳边所指节点一定是当前节点的**最长后缀**。

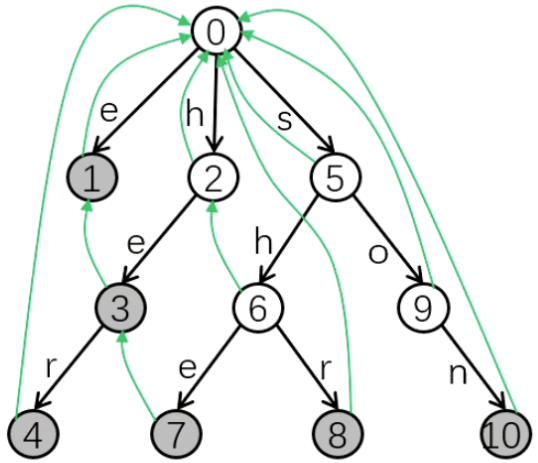
ch[u][i] 存节点 u 的**树边**的终点。**ch[6][e]=7**
ch[u][i] 存节点 u 的**转移边**的终点。**ch[7][r]=4**
转移边指向**当前节点的回跳边所指节点的儿子**。
三个点（u, ne[u], ch[][]）构成三角形。
转移边所指节点一定是当前节点的**最短路**。

```
void build(){//建AC自动机
  queue<int> q;
  for(int i=0;i<26;i++)
    if(ch[0][i])q.push(ch[0][i]);
  while(q.size()){
    int u=q.front();q.pop();
    for(int i=0;i<26;i++){
      int v=ch[u][i];
      if(v)ne[v]=ch[ne[u]][i],q.push(v);
      else ch[u][i]=ch[ne[u]][i];
    }
  }
}
```

用 BFS 构造 AC自动机
初始化，把根节点的儿子们入队。
只要队不空，节点 u 出队，
枚举 u 的 26 个儿子，
1. 若儿子存在，则爹帮儿子建**回跳边**，并把儿子入队。
2. 若儿子不存在，则爹自建**转移边**。



e
he
her
she
shr
son



{1,2,5} 入队
1出队：
ch[1][e]=1
ch[1][h]=2
ch[1][s]=5
ch[1][…]=0
2出队：
ne[3]=1, {3}
ch[2][h]=2
ch[2][s]=5

5出队：
ne[6]=2, {6}
ne[9]=0, {9}
ch[5][e]=1
ch[5][s]=5
3出队：
ne[4]=0, {4}
ch[3][e]=1
ch[3][h]=2
ch[3][s]=5

6出队：
ne[7]=3, {7}
ne[8]=0, {8}
ch[6][h]=2
ch[6][s]=5
9出队：
ne[10]=0, {10}
ch[9][e]=1
ch[9][h]=2
ch[9][s]=5

4出队：
ch[4][e]=1
ch[4][h]=2
ch[4][s]=5
7出队：
ch[7][e]=1
ch[7][h]=2
ch[7][r]=4
ch[7][s]=5

8出队：
ch[8][e]=1
ch[8][h]=2
ch[8][s]=5
10出队：
ch[8][e]=1
ch[8][h]=2
ch[8][s]=5

查找单词出现次数

扫描主串，依次取出字符 s[k]，

1. i 指针走主串对应的节点，沿着**树边**或**转移边**走，保证不回退。
2. j 指针沿着**回跳边**搜索模式串，每次从当前节点走到根节点，把当前节点中的所有**后缀模式串**一网打尽，保证不漏解。
3. 扫描完主串，返回答案。

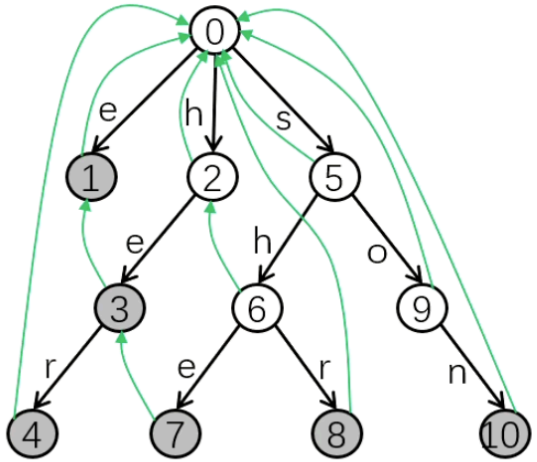
算法一边走串，一边把当前串的所有后缀串搜出来，实在是强。

```
int query(char *s){
  int ans=0;
  for(int k=0,i=0;s[k];k++){
    i=ch[i][s[k]-'a'];
    for(int j=i;j&&~cnt[j];j=ne[j])
      ans+=cnt[j], cnt[j]=-1;
  }
  return ans;
}
```

```
y  i=0
a  i=0
s  i=5
   ans=0 j=0
h  i=6
   ans=0 j=2
   ans=0 j=0
e  i=7
   ans=1 j=3
   ans=2 j=1
   ans=3 j=0
r  i=4
   ans=4 j=0
h  i=2
s  i=5
h  i=6
e  i=7
```

e
he
her
she
shr
son

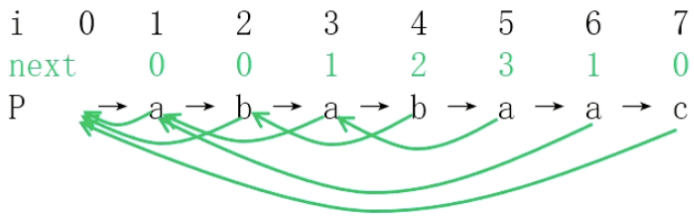
主串：yasherhshe

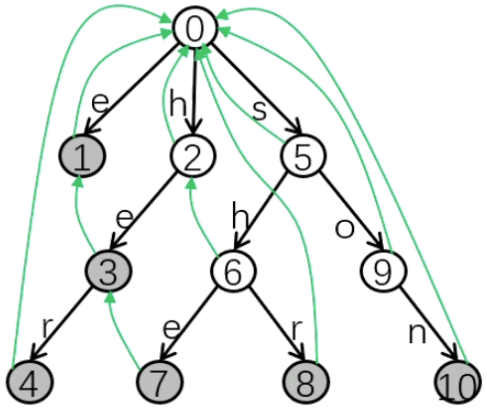


	KMP	AC自动机
匹配	单模式匹配算法	多模式匹配算法
形态	链上构造自动机	Trie 树上构造自动机
数据维护	回跳边 ne[]	回跳边 ne[] 树边，转移边 ch[][]
算法流程	双指针建边，双指针匹配	BFS 建边，双指针匹配
时间复杂度	$O(n+m)$	$O(26n+m)$

若KMP跑多模匹配 $O(n+km)$

例，k=1000, n=10000, m=10^7





```
void build(){//建AC自动机
  queue<int> q;
  for(int i=0;i<26;i++)
    if(ch[0][i])q.push(ch[0][i]);
  while(q.size()){
    int u=q.front();q.pop();
    for(int i=0;i<26;i++){
      int v=ch[u][i];
      if(v)ne[v]=ch[ne[u]][i],q.push(v);
      else ch[u][i]=ch[ne[u]][i];
    }
  }
}
```

```
int query(char *s){
  int ans=0;
  for(int k=0,i=0;s[k];k++){
    i=ch[i][s[k]-'a'];
    for(int j=i;j&&~cnt[j];j=ne[j])
      ans+=cnt[j], cnt[j]=-1;
  }
  return ans;
}
```

◆ 重点

- **ne[v]** 存节点 v 的**回跳边**的终点
  ch[u][i] 存节点 u 的**树边**或**转移边**的终点
- **回跳边**指向**父节点的回跳边所指节点的儿子**
  转移边指向**当前节点的回跳边所指节点的儿子**
- 时间复杂度：O(26n + m)
- Luogu P3808

P3796 AC 自动机（简单版 II）

题意

给定 N 个模式串和一个文本串 T，需要找出在文本串中出现次数最多的模式串。如果多个模式串出现次数相同，按输入顺序输出。

数据范围：最多 50 组数据，$1 \leq N \leq 150$，模式串长度 ≤ 70，文本串长度 $\leq 10^6$。

> 样例解释：第一组数据中，模式串 "aba" 在文本串中出现 4 次，是最多的；第二组数据中，"alpha" 和 "haha" 都出现 2 次，是最多的。

分析

核心思路：使用 AC 自动机高效解决多模式匹配问题。

1. **Trie 树构建**：将所有模式串插入到 Trie 树中，每个叶子节点记录对应的模式串编号。
2. **Fail 指针构建**：通过 BFS 为每个节点构建失败指针，用于在匹配失败时快速跳转。
3. **文本匹配**：在 AC 自动机上遍历文本串，对于每个位置，沿着 fail 链统计所有匹配的模式串。
4. **结果统计**：记录每个模式串的出现次数，按出现次数降序和输入顺序升序排序输出。

时间复杂度：$O(\sum |P_i| + |T|)$，其中 $\sum |P_i|$ 是所有模式串总长度，$|T|$ 是文本串长度。

参考代码

```cpp
const int N = 1e6 + 10;
struct Node {
    int fail;          // 失败指针
    int son[26];       // 子节点
    int end;           // 模式串编号（0表示不是结尾）
} trie[N];

int tot;                // 节点计数器
string patterns[200];   // 存储模式串
int ans[200];           // 每个模式串的出现次数

// 初始化节点
void initNode(int x) {
    memset(trie[x].son, 0, sizeof(trie[x].son));
    trie[x].fail = 0;
    trie[x].end = 0;
}

// 插入模式串到Trie树
void insert(const string& s, int id) {
    int p = 0;
    for (char c : s) {
        int idx = c - 'a';
        if (!trie[p].son[idx]) {
            trie[p].son[idx] = ++tot;
            initNode(tot);
        }
        p = trie[p].son[idx];
    }
    trie[p].end = id;  // 记录模式串编号
}
```

```cpp
// 构建AC自动机的fail指针
void build() {
    queue<int> q;
    // 第一层节点的fail指针指向根节点
    for (int i = 0; i < 26; i++) {
        if (trie[0].son[i]) {
            trie[trie[0].son[i]].fail = 0;
            q.push(trie[0].son[i]);
        }
    }

    // BFS构建fail指针
    while (!q.empty()) {
        int u = q.front();
        q.pop();

        for (int i = 0; i < 26; i++) {
            if (trie[u].son[i]) {
                // 子节点的fail指针指向父节点fail指针的对应子节点
                trie[trie[u].son[i]].fail = trie[trie[u].fail].son[i];
                q.push(trie[u].son[i]);
            } else {
                // 优化：直接记录转移边
                trie[u].son[i] = trie[trie[u].fail].son[i];
            }
        }
    }
}

// 在文本串中查询所有模式串
void query(const string& s) {
    int p = 0;
    for (char c : s) {
        p = trie[p].son[c - 'a'];
        // 沿着fail链统计所有匹配的模式串
        for (int j = p; j; j = trie[j].fail) {
            if (trie[j].end) {
                ans[trie[j].end]++;
            }
        }
    }
}
```

```cpp
int main() {
    int n;
    while (cin >> n && n) {
        // 初始化
        tot = 0,initNode(0),memset(ans, 0, sizeof(ans));

        // 读取模式串并构建Trie
        for (int i = 1; i <= n; i++)
            cin >> patterns[i],insert(patterns[i], i);

        // 构建AC自动机
        build();

        // 读取文本串并查询
        string text;
        cin >> text;
        query(text);

        // 找出最大出现次数
        int maxCount = 0;
        for (int i = 1; i <= n; i++)
            maxCount = max(maxCount, ans[i]);

        // 输出结果
        cout << maxCount << "\n";
        for (int i = 1; i <= n; i++)
            if (ans[i] == maxCount)
                cout << patterns[i] << "\n";
    }

    return 0;
}
```

小结：

1. **数据结构**：
   - `Node` 结构体表示 Trie 树节点，包含失败指针、子节点数组和模式串编号
   - `patterns` 数组存储所有模式串
   - `ans` 数组记录每个模式串的出现次数
2. **核心函数**：
   - `insert()`：将模式串插入 Trie 树，在结尾节点记录编号
   - `build()`：使用 BFS 构建 fail 指针，优化转移边
   - `query()`：在文本串中匹配所有模式串，沿 fail 链统计出现次数

1. **思路**：
    - 初始化 AC 自动机
    - 构建 Trie 树和 fail 指针
    - 在文本串中匹配并统计
    - 输出出现次数最多的模式串
2. **优化技巧**：
    - 转移边优化：避免匹配时频繁跳转 fail 指针
    - 按输入顺序输出：天然满足题目要求

AC 自动机应用：统计模式串出现次数

```
const int N = 1000010;
const int M = 26;

struct Node {
    int son[M], fail, count;
} ac[N];

int idx;
char pattern[N], text[N];

void init() {
    memset(ac, 0, sizeof(ac));
    idx = 0;
}

void insert(const char* s) {
    int p = 0;
    for (int i = 0; s[i]; i++) {
        int c = s[i] - 'a';
        if (!ac[p].son[c]) ac[p].son[c] = ++idx;
        p = ac[p].son[c];
    }
    ac[p].count++;
}
```

```
void buildFail() {
    queue<int> q;
    for (int i = 0; i < M; i++)
        if (ac[0].son[i])
            ac[ac[0].son[i]].fail = 0, q.push(ac[0].son[i]);

    while (!q.empty()) {
        int u = q.front(); q.pop();
        for (int i = 0; i < M; i++) {
            int v = ac[u].son[i];
            if (v)
                ac[v].fail = ac[ac[u].fail].son[i],q.push(v);
             else
                ac[u].son[i] = ac[ac[u].fail].son[i];
        }
    }
}
```

```
int query(const char* s) {
    int p = 0, res = 0;
    for (int i = 0; s[i]; i++) {
        int c = s[i] - 'a';
        p = ac[p].son[c];
        for (int j = p; j && ac[j].count != -1; j = ac[j].fail) {
            res += ac[j].count;
            ac[j].count = -1;
        }
    }
    return res;
}
```

AC 自动机优化：记录所有匹配位置

```
// 记录每个模式串的出现位置
vector<int> positions[N];

void queryWithPositions(const char* s) {
    int p = 0;
    for (int i = 0; s[i]; i++) {
        int c = s[i] - 'a';
        p = ac[p].son[c];

        for (int j = p; j; j = ac[j].fail)
            if (ac[j].end)
                // 记录模式串ac[j].end在位置i出现
                positions[ac[j].end].push_back(i);
    }
}
```

Fail 树的性质

重要概念

Fail 树：将 $fail$ 指针反向构成的树

性质：

- 节点 u 的子树中的所有节点，其 $fail$ 指针路径都会经过 u
- 用于统计每个模式串被匹配的次数

```
// 构建fail树
vector<int> fail_tree[N];

void buildFailTree() {
    for (int i = 1; i <= idx; i++)
        fail_tree[ac[i].fail].push_back(i);
}
```

AC 自动机与动态规划

应用场景

在包含禁止模式的文本中计数合法字符串

```
// dp[i][j]: 长度为i，在AC自动机节点j的合法方案数
const int MOD = 1000000007;
int dp[1010][N];

int countValidStrings(int n) {
    dp[0][0] = 1;

    for (int i = 0; i < n; i++) {
        for (int j = 0; j <= idx; j++) {
            if (dp[i][j] == 0) continue;

            for (int k = 0; k < M; k++) {
                int next = ac[j].son[k];
                // 如果next节点不是模式串结尾
                if (ac[next].count == 0)
                    dp[i + 1][next] = (dp[i + 1][next] + dp[i][j]) % MOD;
            }
        }
    }

    int ans = 0;
    for (int j = 0; j <= idx; j++)
        ans = (ans + dp[n][j]) % MOD;
    return ans;
}
```

算法复杂度分析

操作	时间复杂度	空间复杂度
构建 Trie 树	$O(\sum P_i)$	$O(\sum P_i \times \Sigma)$
构建 Fail 指针	$O(\sum P_i \times \Sigma)$	$O(\sum P_i)$
文本匹配	$O(T + m)$	$O(1)$

其中：

- P_i：第 i 个模式串
- T：文本串
- m：匹配次数
- Σ：字符集大小