# 字符串

## 哈希

### 哈希表

#include <unordered\_map>

其余略

### 字符串哈希

和oi-wiki上不同，这里介绍另一种哈希，这样做方便进行单点修改。

选定的进制为，并且要对取模（一个大质数）。

* 求算哈希值

如字符串的哈希值可以表示为，我们可以把字符串中下表较大的位看成高位（反过来也可以，但利用前缀求子串不同）

* 代码

先给出几个可以用于双模哈希的模数。

915650123  
915707077  
945519319  
1000000123  
1000000321  
1000008899  
1000001213  
1000018333  
1000019333  
1000222337  
1000223377

下面是一个值域为的双模哈希，一般认为可以处理级别的不同字符串。但是**实际应用的时候从来不会这么写**，因为效率太低。

下面给出前缀哈希数组的求解函数，这份代码中求解了正反两个方向的双模哈希（共4组）用以判断回文子串。

int P[2] = {915707077, 1000223377}, B[2] = {77, 99};  
  
void get\_hash(char \*s) {  
 int len = strlen(s + 1);  
 for (int t = 0; t < 2; ++t) { // 双模hash  
 ll res = 0, b = 1;  
 for (int i = 1; i <= len; ++i) { // 正向hash  
 res = (res + b \* (s[i] - 'a')) % P[t];  
 b = (b \* B[t]) % P[t];  
 // tr[t].add(i, res);  
 // tr[t].add(i, res, P[t]);  
 tr[t].a[i] = res;  
 }  
 tr[t].build(1, 1, len, P[t]);  
  
 res = 0;  
 b = 1;  
 for (int i = len; i >= 1; --i) { // 反向hash  
 res = (res + b \* (s[i] - 'a')) % P[t];  
 b = (b \* B[t]) % P[t];  
 re[t].a[i] = res;  
 }  
 re[t].build(1, 1, len, P[t]);  
 }  
}

特别注意**所有涉及到乘法的运算都有更可能溢出**，因此要用long long维护

* 前缀哈希

可以预处理出一个数组表示的前缀的哈希值。

观察两个式子：

那么子串的哈希值可以表示为：

若要比较两个子串是否相同，只需要注意将式子左边的指数对其即可。

下面给出用线段树维护前缀hash数组，并通过求子串正反向哈希值并判断回文的例子：

int P[2] = {915707077, 1000223377}, B[2] = {30, 33};  
int BINV[2] = {fpow(B[0], P[0] - 2, P[0]), fpow(B[1], P[1] - 2, P[1])};  
// 费马小定理求大质数取模乘法的逆元  
//...  
int l, r;  
cin >> l >> r;  
for (int t = 0; t < 2; ++t) {  
 int h1, h2, exp1 = l - 1 + 1, exp2 = n - r + 1;  
 // 这里少计算了第一位的哈希，所以指数为做相应调整，不过无伤大雅  
  
 h1 = mod(tr[t].query(1, 1, n, r, r, P[t]) -   
 tr[t].query(1, 1, n, l, l, P[t]), P[t]);  
 h2 = mod(re[t].query(1, 1, n, l, l, P[t]) -   
 re[t].query(1, 1, n, r, r, P[t]), P[t]);  
  
 // 利用费马小定理求算乘法逆元，并将多余的指数除掉  
 h1 = mod((ll)h1 \* fpow(BINV[t], exp1, P[t]), P[t]);   
 h2 = mod((ll)h2 \* fpow(BINV[t], exp2, P[t]), P[t]);  
  
 if (h1 != h2 || s[l] != s[r]) {  
 ans = false;  
 break;  
 }  
}

显然，若不用支持修改，那线段树也没有开的必要。

单点修改也是同理，若要修改位置，显然只影响了的区间，并且在区间内的每一个前缀，的贡献是相同的。若要单点修改，只要转化成修改的后缀区间前缀哈希值即可。

以下是单点修改转换为区间修改哈希值的例子：

int x;  
char c;  
cin >> x >> c;  
int offset = c - s[x];  
s[x] = c; // 修改字符串  
for (int t = 0; t < 2; ++t) {  
 tr[t].add(1, 1, n, x, n, mod((ll)offset \*   
 fpow(B[t], x - 1, P[t]), P[t]), P[t]);  
 re[t].add(1, 1, n, 1, x, mod((ll)offset \*   
 fpow(B[t], n - x, P[t]), P[t]), P[t]);  
}

* **如何用哈希判断字符串是否相等？**

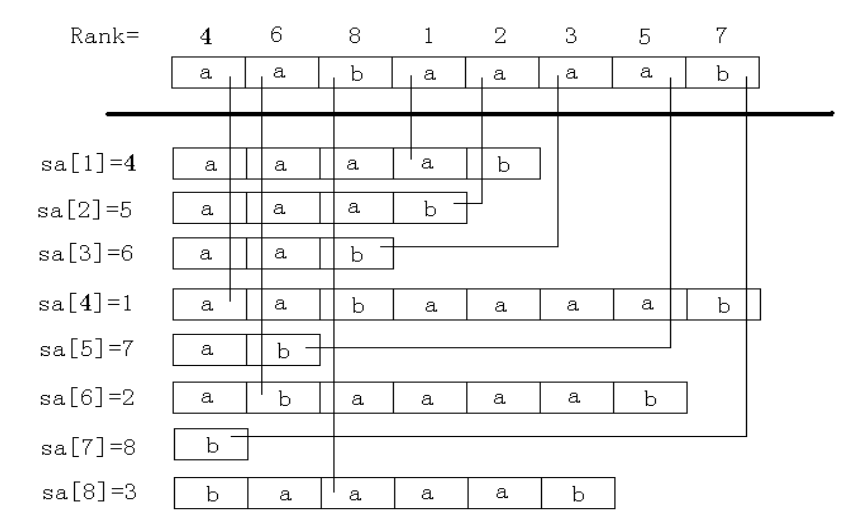
基本思路是，二分查找两个串最长的相同的前缀，然后看的位置的大小关系

inline int halffind() {  
 int l = 1, r = (rig - lef + 1) >> 1;  
 int mid, res = 1;  
 while (l <= r) {  
 mid = (l + r) >> 1;  
 if (check(mid)) { // 判断两段的hash值是否相等  
 res = mid;  
 l = mid + 1;  
 } else {  
 r = mid - 1;  
 }  
 }  
 return res;  
}

## 后缀数组（SA）

后缀数组（Suffix Array）主要关系到两个数组：和。

其中，表示将所有后缀排序第小的后缀数组编号，也就是所说的后缀数组；表示后缀的排名。这两个数组满足性质：



SA

求解sa数组的基本思想涉及到倍增和基数排序：

* 倍增

若已知长度为的子串的排名，那只需将前半部分作为第一关键字，后半部分作为第二关键字，即可求得长度为的子串的排名。

* 基数排序

对于上文中的双关键字排序还可进一步使用基数排序优化到级别。

当然，还需要一些**常数优化**，于是我们就得到了OI Wiki上的板子：

#include <algorithm>  
#include <cstdio>  
#include <cstring>  
#include <iostream>  
  
using namespace std;  
  
const int N = 1000010;  
  
char s[N];  
int n, sa[N], rk[N << 1], oldrk[N << 1], id[N], cnt[N];  
  
int main() {  
 int i, m, p, w;  
  
 scanf("%s", s + 1);  
 n = strlen(s + 1);  
 m = 127;  
 for (i = 1; i <= n; ++i) ++cnt[rk[i] = s[i]];  
 for (i = 1; i <= m; ++i) cnt[i] += cnt[i - 1];  
 for (i = n; i >= 1; --i) sa[cnt[rk[i]]--] = i;  
 memcpy(oldrk + 1, rk + 1, n \* sizeof(int));  
 for (p = 0, i = 1; i <= n; ++i) {  
 if (oldrk[sa[i]] == oldrk[sa[i - 1]]) {  
 rk[sa[i]] = p;  
 } else {  
 rk[sa[i]] = ++p;  
 }  
 }  
  
 for (w = 1; w < n; w <<= 1, m = n) {  
 // 对第二关键字：id[i] + w进行计数排序  
 memset(cnt, 0, sizeof(cnt));  
 memcpy(id + 1, sa + 1,  
 n \* sizeof(int)); // id保存一份儿sa的拷贝，实质上就相当于oldsa  
 for (i = 1; i <= n; ++i) ++cnt[rk[id[i] + w]];  
 for (i = 1; i <= m; ++i) cnt[i] += cnt[i - 1];  
 for (i = n; i >= 1; --i) sa[cnt[rk[id[i] + w]]--] = id[i];  
  
 // 对第一关键字：id[i]进行计数排序  
 memset(cnt, 0, sizeof(cnt));  
 memcpy(id + 1, sa + 1, n \* sizeof(int));  
 for (i = 1; i <= n; ++i) ++cnt[rk[id[i]]];  
 for (i = 1; i <= m; ++i) cnt[i] += cnt[i - 1];  
 for (i = n; i >= 1; --i) sa[cnt[rk[id[i]]]--] = id[i];  
  
 memcpy(oldrk + 1, rk + 1, n \* sizeof(int));  
 for (p = 0, i = 1; i <= n; ++i) {  
 if (oldrk[sa[i]] == oldrk[sa[i - 1]] &&  
 oldrk[sa[i] + w] == oldrk[sa[i - 1] + w]) {  
 rk[sa[i]] = p;  
 } else {  
 rk[sa[i]] = ++p;  
 }  
 }  
 }  
  
 for (i = 1; i <= n; ++i) printf("%d ", sa[i]);  
  
 return 0;  
}

* 应用

给你一个字符串，每次从首或尾取一个字符组成字符串，问所有能够组成的字符串中字典序最小的一个。

显然首尾不同的时候就贪心地选最小的那个，若首尾相同则需要比较原字符串和反字符串的后缀的字典序，取较小的更优。

一种做法是上面提到的哈希，然后二分比较字符串大小。

使用后缀数组需要将反串复制到原串的末尾，中间插入一个比合法字符都 小的字符以免影响，然后求相应的后缀数组。

比如这样：

// 复制反字符串  
int nn;  
scanf("%d", &nn);  
n = nn \* 2 + 1;  
for (int i = 1; i <= nn; ++i) {  
 int c = getchar();  
 while (c > 'Z' || c < 'A') {  
 c = getchar();  
 }  
 s[i] = c;  
 s[n - i + 1] = c;  
}  
s[nn + 1] = 'A' - 1; // 设定最小值  
  
// 利用后缀数组求解  
int l = 1, r = nn, tot = 0;  
while (l <= r) {  
 putchar(rk[l] < rk[n - r + 1] ? s[l++] : s[r--]);  
 ++tot;  
 if (tot == 80) {  
 putchar('\n');  
 tot = 0;  
 }  
}

于是乎，我们似乎能得到后缀数组的规律——大部分用后缀数组解决的问题都需要**按一定方式将原字符串展开**，比如直接复制两份，或者复制反串等。

### height 数组

定义：，即第名后缀与第名后缀的最长公共前缀的长度。认为。

for (i = 1, k = 0; i <= n; ++i) {  
 if (rk[i] == 0) continue;  
 if (k) --k;  
 while (s[i + k] == s[sa[rk[i] - 1] + k]) ++k;  
 height[rk[i]] = k;  
}

有了 height 数组，我们可以更好理解“子串”这一概念——**后缀的前缀**。

## 自动机

而非。

自动机是一个对信号序列进行判定的**数学模型**，并非算法或数据结构，实现其的方法可能有许多种。

「信号序列」是指一连串有顺序的信号，例如字符串从前到后的每一个字符、数组从 1 到 n 的每一个数、数从高到低的每一位等。「判定」是指针对某一个命题给出或真或假的回答。

自动机的结构是一张有向图，节点是单纯的状态。OI中的自动机一般指确定有限状态自动机（DFA）

### 子序列自动机

有次询问，判断是否为的子串。

pos[i]用于维护字符i在串中出现的所有位置，也就是说对于当前需要匹配的字符，只需要在对应的pos数组中查找即可。

下面的代码中，需要特别注意last\_p的初始值是-1，数组的下标从0开始。

for (int i = 0; i < n; ++i) {  
 a[i] = get\_int();  
 pos[a[i]].push\_back(i);  
}  
while (q--) {  
 bool flg = true;  
 int l = get\_int(), last\_p = -1; // 开始在没有字符的地方  
 for (int i = 0; i < l; ++i) {  
 int b = get\_int();  
 auto it = upper\_bound(pos[b].begin(), pos[b].end(), last\_p);  
 if (it == pos[b].end()) {  
 for (int j = i + 1; j < l; ++j) b = get\_int();  
 flg = false;  
 break;  
 }  
 last\_p = \*it;  
 }  
 printf("%s\n", flg? "Yes" : "No");  
}

### KMP

~~看毛片~~

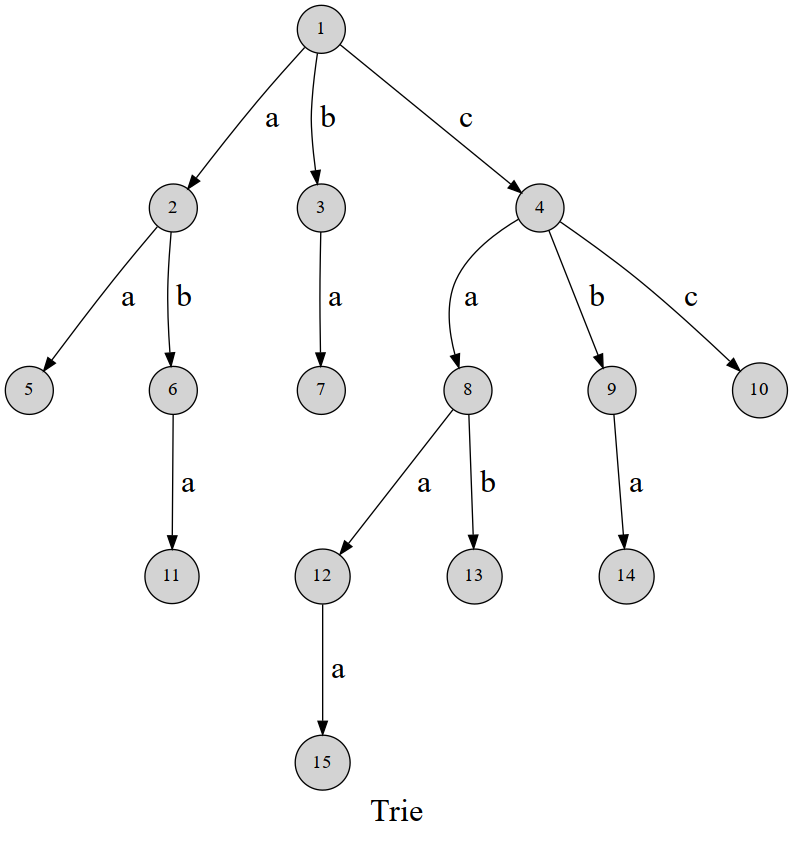
KMP可以视作自动机。

注释中已经标注了各个变量与数组的作用。

kmp[0] = 0; // 表示前i个字符的前缀中，真前缀与真后缀匹配的最大长度（并非本身）  
int prefix\_len = 0; // 当前前后缀匹配长度  
for (int i = 1; i < pattern.length(); ++i) {  
 while (prefix\_len != 0 && pattern[i] != pattern[prefix\_len]) {  
 prefix\_len = kmp[prefix\_len - 1];  
 }  
 if (pattern[i] == pattern[prefix\_len]) {  
 ++prefix\_len;  
 }  
 kmp[i] = prefix\_len;  
}  
prefix\_len = 0;  
for (int i = 0; i < text.length(); ++i) {  
 while (prefix\_len != 0 && text[i] != pattern[prefix\_len]) {  
 prefix\_len = kmp[prefix\_len - 1];  
 }  
 if (text[i] == pattern[prefix\_len]) {  
 ++prefix\_len;  
 }  
 if (prefix\_len == pattern.length()) {  
 cout << i - pattern.length() + 2 << '\n';  
 prefix\_len = kmp[prefix\_len - 1];  
 }  
}

### 字典树(Trie)

字典树也可以视作是自动机，接受且只接受指定字符串集合中的元素。



Trie

从OI Wiki上扒下来的

struct Trie {  
 int nex[100000][26], cnt;  
 bool exist[100000]; // 该结点结尾的字符串是否存在  
 void insert(char \*s, int l) { // 插入字符串  
 int p = 0;  
 for (int i = 0; i < l; i++) {  
 int c = s[i] - 'a';  
 if (!nex[p][c]) nex[p][c] = ++cnt; // 如果没有，就添加结点  
 p = nex[p][c];  
 }  
 exist[p] = 1;  
 }  
 bool find(char \*s, int l) { // 查找字符串  
 int p = 0;  
 for (int i = 0; i < l; i++) {  
 int c = s[i] - 'a';  
 if (!nex[p][c]) return 0;  
 p = nex[p][c];  
 }  
 return exist[p];  
 }  
};

### AC自动机

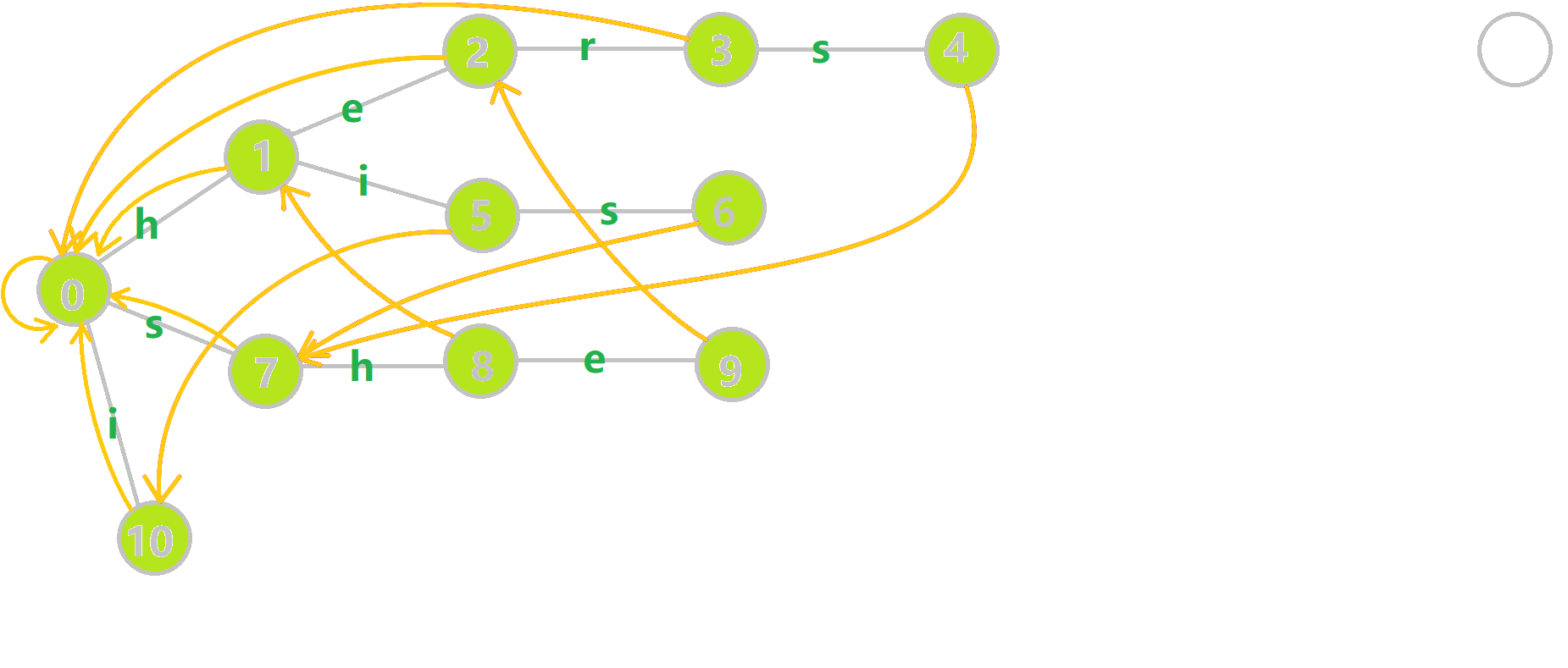
而非，用于实现多模式串匹配。

前置算法：KMP和Trie；实现方法：~~在树上看毛片~~。

两个概念：Trie和fail指针

1. 先将所有模式串丢到一个Trie里，这里用word[i]代表Trie上从根节点走到节点i所代表的字串。
2. 然后构建fail指针，其中fail[i] = j表明word[j]是word[i]的最长后缀，若指向根就表明找不到后缀；当前串失配之后跳转到fail指针，这样的话失配之后的前缀信息不会浪费，而是会利用已有的前缀信息继续匹配。

假设模式串分别是i、she、he、hers、his，Trie和fail指针就该是下面的样子



fail指针

~~fail指针告诉我们，不要当舔狗，此处不留爷，必有留爷处~~

构建AC自动机，需要使用BFS，先将第一层节点（0号的所有儿子）压入队列

if(tr[u][i])fail[tr[u][i]]=tr[fail[u]][i],q.push(tr[u][i]);  
else tr[u][i]=tr[fail[u]][i];

这里相当于做了路径压缩，可以地进行状态转移，重新构建了字典树，这些新加的边使字典树变成了**字典图**

假设模式串分别为r、er、her，文本串为herr，当前匹配到了文本串第4位（第二个r）。如果没有路径压缩就要按虚线（蓝色）跳转，若有路径压缩就可以按粗线（红线）跳转，这就是上面代码的else的作用；由此，也不需要用while来实现对fail的求解，只需要简单赋值即可

![字典图](data:application/octet-stream;base64,)

字典图

结合下面的query()能更好地理解，可以直接从字典图的边进行状态转移，而统计答案的时候需要逐步fail

* **简单版**（P3808）：求有多少个不同的模式串在文本串中出现过

简单之处在于，所有节点的val只需统计一遍即可，因为只要求得“出现过”而非次数

在主函数里先insert()模式串，再build()，最后query()文本串

int tr[MAXN][26], val[MAXN], fail[MAXN], cnt;  
// 字典树，节点模式串数量，fail指针，节点总数  
void insert(const char \*s) { // 构建Trie  
 size\_t len = strlen(s);  
 int now = 0;  
 for (int i = 0; i < len; ++i) {  
 int c = s[i] - 'a';  
 if (tr[now][c] == 0) {  
 tr[now][c] = ++cnt;  
 }  
 now = tr[now][c];  
 }  
 ++val[now]; // 记录当前的点权(该点有多少模式串)  
}  
void build() { // 也可以叫get\_fail  
 for (int i = 0; i < 26; ++i) { // 第一层节点压入队列  
 if (tr[0][i] != 0) {  
 fail[tr[0][i]] = 0;  
 que.push(tr[0][i]);  
 }  
 }  
 while(que.empty() == false) {  
 int u = que.front();  
 que.pop();  
 for (int i = 0; i < 26; ++i) {  
 if (tr[u][i]) {  
 fail[tr[u][i]] = tr[fail[u]][i];  
 que.push(tr[u][i]);  
 } else {  
 tr[u][i] = tr[fail[u]][i];  
 }  
 }  
 }  
}  
int query(const char \*s) { // 查询  
 size\_t len = strlen(s);  
 int now = 0, ans = 0;  
 for (int i = 0; i < len; ++i) {  
 now = tr[now][s[i] - 'a'];  
 for (int j = now; j != 0 && val[j] != -1; j = fail[j]) {  
 ans += val[j];  
 val[j] = -1;  
 }  
 }  
 return ans;  
}

* **加强版**（P3796）：求哪些模式串（）出现的次数最多（）

对于上面的insert()和query()稍加修改即可，val[]不再保存出现的次数，而是保存哪个字符串以此结尾；修改相应的查询语句，并去掉val[j] = -1;即可

* **二次加强版**（P5357）：求模式串出现的次数（）

只保留所有的fail，可以构成一个由儿子指向父亲的树，那么每次不用暴力fail，而只统计当前节点，最后query之后再根据fail构成的图求前缀和即可。

并不用真的把图建出来，只用统计入度就行。

#include <iostream>  
#include <queue>  
#include <string>  
#include <vector>  
using namespace std;  
#define MAXN 2000007  
  
queue<int> que;  
vector<int> flag[MAXN];  
  
// Trie，节点存储的模式串编号，fail，答案，前缀和，入度  
int tr[MAXN][26], fail[MAXN]  
 , prefix\_sum[MAXN], ans[MAXN], in[MAXN], cnt;  
  
string pattern[MAXN]; // 模式串（编号从1开始，因为val为0表示没有）  
  
void insert(const string &s, int idx) { // 构建Trie  
 size\_t len = s.size();  
 int now = 0;  
 for (size\_t i = 0; i < len; ++i) {  
 int c = s[i] - 'a';  
 if (tr[now][c] == 0) {  
 tr[now][c] = ++cnt;  
 }  
 now = tr[now][c];  
 }  
 flag[now].push\_back(idx); // 可能会有重复的模式串  
}  
  
void build() { // 也可以叫get\_fail  
 for (int i = 0; i < 26; ++i) { // 第一层节点压入队列  
 if (tr[0][i] != 0) {  
 fail[tr[0][i]] = 0;  
 que.push(tr[0][i]);  
 }  
 }  
 while(que.empty() == false) {  
 int u = que.front();  
 que.pop();  
 for (int i = 0; i < 26; ++i) {  
 if (tr[u][i]) {  
 fail[tr[u][i]] = tr[fail[u]][i];  
 ++in[fail[tr[u][i]]];  
 que.push(tr[u][i]);  
 } else {  
 tr[u][i] = tr[fail[u]][i];  
 }  
 }  
 }  
}  
  
void query(const string &s) { // 查询  
 size\_t len = s.size();  
 int now = 0;  
 for (size\_t i = 0; i < len; ++i) {  
 now = tr[now][s[i] - 'a'];  
 ++prefix\_sum[now]; // 这里改成前缀和  
 }  
}  
  
void topo() {  
 for (int i = 1; i <= cnt; ++i) {  
 if (in[i] == 0) {  
 que.push(i);  
 }  
 }  
 while (que.empty() == false) {  
 int now = que.front();   
 que.pop();  
 --in[fail[now]]; // 更新fail所指节点的入度  
 prefix\_sum[fail[now]] += prefix\_sum[now]; // 更新fail所指节点的前缀和  
 // ans[val[now]] = prefix\_sum[now];  
 for (size\_t i = 0; i < flag[now].size(); ++i) { // 统计答案  
 ans[flag[now][i]] = prefix\_sum[now];  
 }  
 if (in[fail[now]] == 0) {  
 que.push(fail[now]);  
 }  
 }  
}  
  
int n;  
string p;  
  
int main() {  
 #ifdef LOCAL  
 freopen("in.txt", "r", stdin);  
 freopen("out.txt", "w", stdout);  
 #endif // LOCAL  
 cin >> n;  
 for (int i = 1; i <= n; ++i) {  
 cin >> pattern[i];  
 insert(pattern[i], i);  
 }  
 build();  
 cin >> p;  
 query(p);  
 topo();  
 for (int i = 1; i <= n; ++i) {  
 cout << ans[i] << '\n';  
 }  
 cin >> p;  
 return 0;  
}

这一版代码没有封装到结构体里，好像没大用……

* **更进一步的优化**（CSP202009-5 密信与计数）

如果有很多次查询，每次只需要查询当前状态有无模式串，可以不必每次跳转fail，只需要build的时候做一些小改动，把fail指向的状态继承过来即可

if (tr[u][i]) {  
 fail[tr[u][i]] = tr[fail[u]][i];  
 val[tr[u][i]] |= val[fail[tr[u][i]]];  
 que.push(tr[u][i]);  
}

那query就被改成了这个样子

int query(const string &s, int now) { // 查若存在返回-1  
 size\_t len = s.size();  
 for (size\_t i = 0; i < len; ++i) {  
 now = tr[now][s[i] - 'a'];  
 if (val[now] != 0) {  
 return -1; // 若存在返回-1  
 }  
 }  
 return now; // 若不存在返回结束时状态  
}

但是以上写法在这道题里会被卡常，**不能用string，有时候它很慢**；这道题只能边生成对应密文，边转移状态

### 后缀自动机（SAM）

直观上，字符串的SAM可以理解为给定字符串的所有子串的压缩形式，构造的时空复杂度均为，可以线性复杂度解决：

* 在另一个字符串中搜索一个字符串的所有出现位置
* 计算给定的字符串中有多少个不同的子串

（学不动了）