字符串

STL中的string类

构造函数

```
string(n,c) // 使用n个字符c进行初始化
```

输出

如果想要用 $printf$ 输出的话，要调用 $c\_str()$ 函数进行输出，否则编译错误。

```
    printf("%s", s.c_str());
```

$find$函数

`find(str,pos)` 函数可以用来查找字符串中一个字符/字符串在 `pos`（包含）之后第一次出现的位置（若不传参给 `pos` 则默认为 `0`）。如果没有出现，则返回 `string::npos`（被定义为 `-1`，但类型仍为 `size_t`/`unsigned long`）。

```
    string str="abc";
    if(str.find('d') == str.npos){
        cout << 1 << endl;
    }
```

$find$的重载版本：

```
int find(const char *s,int pos,int n) const; // 从pos开始查找s的前n个字符出现的第一次位置
```

$rfind$的重载版本

```
int rfind(const string &str,int pos=npos) const; // 从pos开始查找str的最后一次出现的位置
int rfind(const char *s,int pos,int n) const; // 从pos开始查找s的前n个字符最后一次出现的位置
int rfind(const char c,int pos=0) const; // 查找字符c最后一次出现位置
```

$replace$函数

`replace(pos,count,str)` 和 `replace(first,last,str)` 是比较常见的替换函数。它们分别表示将从 `pos` 位置开始 `count` 个字符的子串替换为 `str` 以及将以 `first` 开始（含）、`last` 结束（不含）的子串替换为 `str`，其中 `first` 和 `last` 均为迭代器（必须是迭代器否则会造成进入重载的版本）

```
    string s = "OI Wiki";
    s.replace(2, 5, "");
    printf("将字符串 s 的第 3~7 位替换为空串后得到的字符串是 %s\n", s.c_str());
    s.replace(s.begin(), s.begin() + 2, "NOI");
    printf("将字符串 s 的前两位替换为 NOI 后得到的字符串是 %s", s.c_str());
/*
输出：

将字符串 s 的第 3~7 位替换为空串后得到的字符串是 OI
将字符串 s 的前两位替换为 NOI 后得到的字符串是 NOI
*/
```

比较

和 `std::vector` 类似，`string` 重载了比较运算符，同样是按字典序比较的，所以我们可以直接调用 `std::sort` 对若干字符串进行排序。

插入与删除

`insert(index,count,ch)` 和 `insert(index,str)` 是比较常见的插入函数。它们分别表示在 `index` 处连续插入 `count` 次字符串 `ch` 和插入字符串 `str`。

`erase(index,count)` 函数将字符串 `index` 位置开始（含）的 `count` 个字符删除（若不传参给 `count` 则表示删去 `count` 位置及以后的所有字符）。

```
    string s = "OI Wiki", t = " Wiki";
    char u = '!';
    s.erase(2);
    printf("从字符串 s 的第三位开始删去所有字符后得到的字符串是 %s\n", s.c_str());
    s.insert(2, t);
    printf("在字符串 s 的第三位处插入字符串 t 后得到的字符串是 %s\n", s.c_str());
    s.insert(7, 3, u);
    printf("在字符串 s 的第八位处连续插入 3 次字符串 u 后得到的字符串是 %s", s.c_str());
/*
输出：

从字符串 s 的第三位开始删去所有字符后得到的字符串是 OI
在字符串 s 的第三位处插入字符串 t 后得到的字符串是 OI Wiki
在字符串 s 的第八位处连续插入 3 次字符串 u 后得到的字符串是 OI Wiki!!!
*/
```

*substr*函数

`substr(pos, len)` 函数的参数返回从 `pos` 位置开始截取最多 `len` 个字符组成的字符串（如果从 `pos` 开始的后缀长度不足 `len` 则截取这个后缀）。

```
    string s = "OI Wiki", t = "OI";
    printf("从字符串 s 的第四位开始的最多三个字符构成的子串是 %s\n",s.substr(3,
3).c_str());
    printf("从字符串 t 的第二位开始的最多三个字符构成的子串是 %s",t.substr(1,
3).c_str());
/*
输出：

从字符串 s 的第二位开始的最多三个字符构成的子串是 Wik
从字符串 t 的第二位开始的最多三个字符构成的子串是 I
*/
```

stringstream类

需要包含头文件$< sstream >$

$stringstream$ ss 赋值有两种方式， $1.ss(s); 2.ss << s$;使用ss可以先接收字符串，再将字符串作为 输入流 >> 转化成整型（会忽略字符串里 的空格、回车、tab）。 缓冲区中，$getline(cin, s)$ 遇到回车会结束接收，并且删掉回车。所以，在$while$（$m--$）前面会加一个$getline()$;

应用：分割空格

```
#include<iostream>
#include<sstream>        //istringstream 必须包含这个头文件
#include<string>
using namespace std;
int main(){
    string str="i am a boy";
    istringstream is(str);
    string s;
    while(is>>s)  {
        cout<<s<<endl;
    }
}
```

kmp

应用：

kmp 是用来处理字符串匹配的相关问题的，即给定一个子串，问其在父串里面出现了几次，并给出每次出现的下标，字符串的下标均从 1 开始。

思路：

创建了一个 $ne[i]$ 数组，其所表示的含义是模式串（子串）中从 1 到 i 中最长的的相等前后缀（要求为真即下标不能完全相同，例如$ne[1]$ = 0 , 虽然 $a[1]$ == $a[1]$,但是不能处理，会造成死循环，因此 $ne[1]$ 必然为 0 , $ne[i]$ = j,它所表示的含义就是$s[1, j]$ == $s[i - l + 1, i]$, 是以i结尾的后缀与前缀相等的最大长度，那么每次 $(j + 1)$ 匹配不成功时，就可以跳回 $ne[j]$,继续匹配 $j + 1$, 来进行所谓的预判

```
const int N = 1e6+10;

int ne[N];
char p[N],s[N];

int main()
{
```

```
    int n,m;
    cin >> n >> p+1 >> m >> s+1;
    ne[1] = 0;

    for(int i=2,j=0;i<=n;i++){
        while(j && p[i] != p[j+1])
            j = ne[j];
        if(p[i] == p[j+1])
            j++;
        ne[i] = j;
    }

    for(int i=1,j=0;i<=m;i++){
        while(j && s[i] != p[j+1])
            j = ne[j];
        if(s[i] == p[j+1])
            j++;
        if(j == n)
            cout << i-n << " ";
    }

    return 0;
}
```

字符串哈希

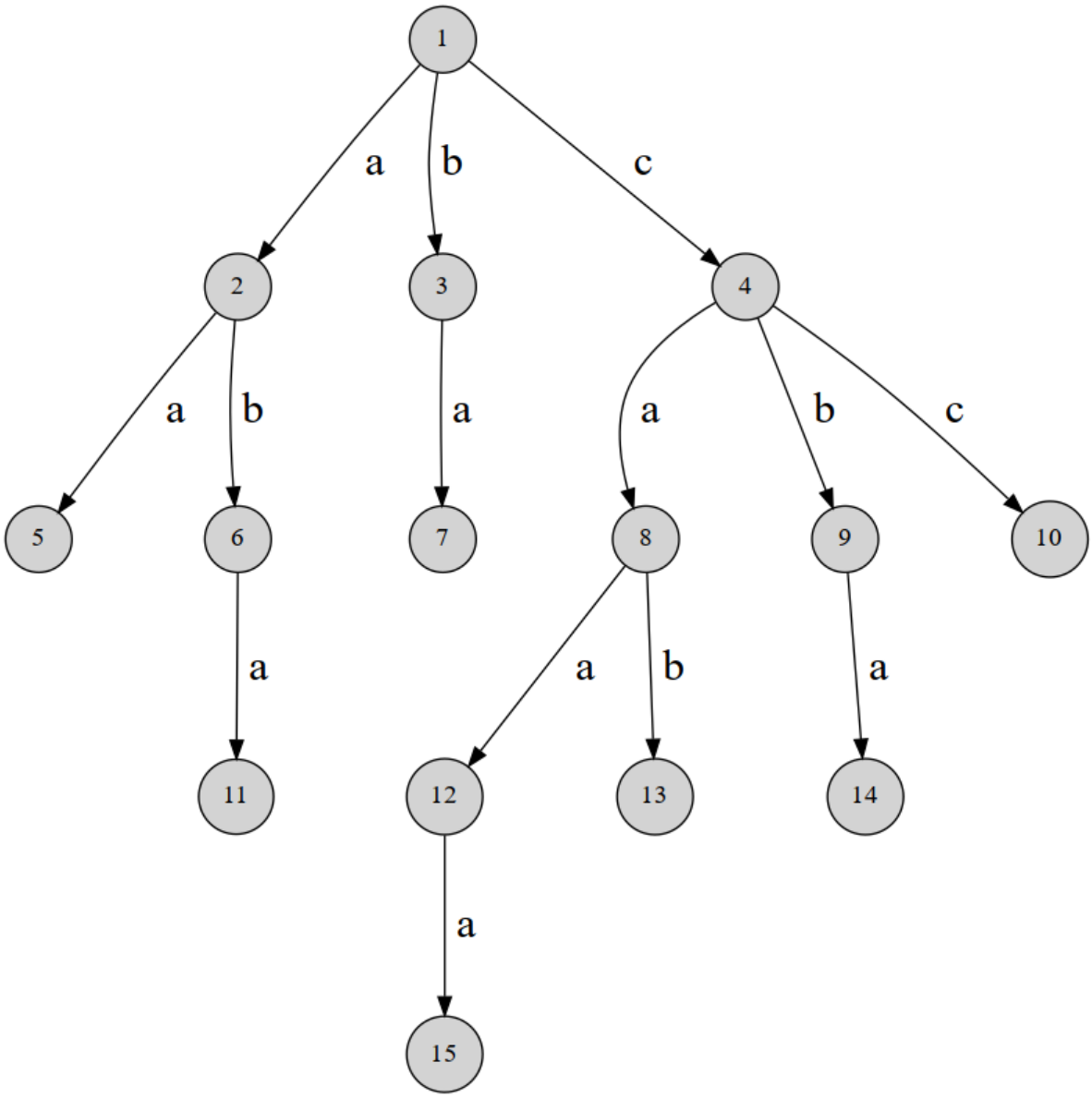
模板代码：

```
const int N = 1e6+10;
const int hash_cnt = 2; // 几重哈希
int hashBase[hash_cnt] = {131,13331};
int hashMod[hash_cnt] = {998244353,(int)1e9+9};

struct StringWithHash {
    char s[N];
    int len;
    int hsh[hash_cnt][N];
    int pwMod[hash_cnt][N];

    void init(){
        len = 0;
        for(int i=0;i<hash_cnt;i++){
            hsh[i][0] = 0;
            pwMod[i][0] = 1;
        }
    }

    StringWithHash(){ init(); }

    void extend(char c){
        s[++len] = c;
        for(int i=0;i<hash_cnt;++i){
            pwMod[i][len] = 1ll*pwMod[i][len-1]*hashBase[i]%hashMod[i];
            hsh[i][len] = (1ll*hsh[i][len-1]*hashBase[i]+c)%hashMod[i];
        }
    }
```

```
    vector<int> getHash(int l, int r){
        vector<int> res(hash_cnt,0);
        for (int i=0;i<hash_cnt;++i) {
          int t = (hsh[i][r]-1ll*hsh[i][l - 1]*pwMod[i][r - l + 1])%hashMod[i];
          t = (t+hashMod[i])%hashMod[i];
          res[i] = t;
        }
        return res;
    }
};

bool equal(const vector<int> &h1, const vector<int> &h2) {
    assert(h1.size() == h2.size());
    for (unsigned i=0;i<h1.size();i++){
        if (h1[i] != h2[i]) return false;
    }
    return true;
}
```

字典树(Tire树)

普通Tire树



Trie

可以发现，这棵字典树用边来代表字母，而从根结点到树上某一结点的路径就代表了一个字符串。举个例子，$1->4->8->12$ 表示的就是字符串 `caa`。

$trie$ 的结构非常好懂，我们用 $tr(u,c)$表示 u 结点的 c 字符指向的下一个结点，或着说是结点 u 代表的字符串后面添加一个字符 c形成的字符串的结点。(c 的取值范围和字符集大小有关，不一定是 0~26)

有时需要标记插入进 $trie$ 的是哪些字符串，每次插入完成时在这个字符串所代表的节点处打上标记即可。

字符串的插入与删除操作，其中 cnt 数组就是我们打上的标记，表示该字符串出现了几次

```
const int N = 1e6 + 10;

int cnt[N];
int son[N][26],idx;

void insert(string str) // 插入操作
{
    int p=0;
    for(int i=0;str[i];i++){
        int u=str[i]-'a';
```

```
        if(!son[p][u]) // 如果该节点没有u的子节点 就创建出来
            son[p][u] = ++idx;
        p = son[p][u]; // 进入子节点部分
    }
    cnt[p]++;
}

int query(string str)
{
    int p=0;
    for(int i=0;str[i];i++){
        int u=str[i]-'a';
        if(!son[p][u])
            return 0;
        p = son[p][u];
    }

    return cnt[p];
}
```

经典例题之最大异或和

思路：具体是对每个数都建立了一个 01 $tire$ 树，在每次从高位向低位贪心时，如果当前位置存在的话就必须转移，否则的话才能走相对的路。

技巧：在处理异或的时候可以不考虑顺序，而处理同或时需要考虑顺序，最简单的思路是边遍历边插入，先查询再插入可以避免自己的值带来影响。

代码：

```
const int bit = 32;
struct trie{
    int ch[N*(bit+1)][2],cnt[N*(bit+1)],val[N*(bit+1)];
    int idx=0;

    void insert(int x){
        int p=0;
        for(int i=bit-1;i>=0;i--){
            int s=x>>i&1;
            if(!ch[p][s])
                ch[p][s] = ++idx;
            p = ch[p][s];
            cnt[p]++;
        }
        val[p] = x;
    }

    void del(int x){
        int p=0;
        for(int i=bit-1;i>=0;i--){
            int s=x>>i&1;
            p = ch[p][s];
            cnt[p]--;
        }
    }

    int query_max(int x){
        int p=0;
```

```
        for(int i=bit-1;i>=0;i--){
            int s=x>>i&1;
            if(ch[p][s^1] && cnt[ch[p][s^1]])
                p = ch[p][s^1];
            else
                p = ch[p][s];
        }
        return x^val[p];
    }
};
```

可持久化 $Tire$ 树

经典例题：最大异或和

可持久化字典树（Trie）

Luogu P4735 最大异或和

给定一个非负整数序列 {a}，初始长度为 n。
有 m 个操作，有以下两种操作类型：
A x：添加操作，表示在序列末尾添加一个数 x，序列的长度 n+1。
Q l r x：询问操作，你需要找到一个位置 p，满足 $l \le p \le r$，使得：$a[p] \oplus a[p+1] \oplus \cdots \oplus a[n] \oplus x$ 最大，输出最大值。

输入格式
第一行包含两个整数 N,M，含义如问题描述所示。
第二行包含 N 个非负整数，表示初始的序列A。
接下来 M 行，每行描述一个操作，格式如题面所述。

输出格式
假设询问操作有 T 个，则输出应该有 T 行，每行一个整数表示询问的答案。

输入	输出
5 5 2 6 4 3 6 A 1 Q 3 5 4 A 4 Q 5 7 0	4 5

说明/提示 N,M≤3×10^5。0≤a[i]≤10^7。

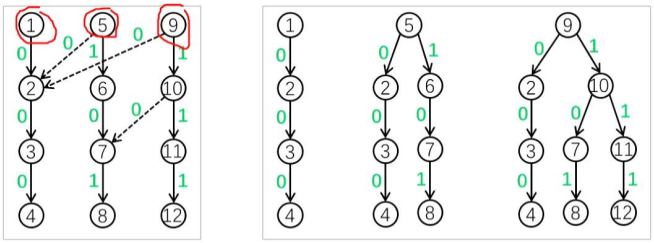
区间和的问题通常转化为**前缀和**。

令 $s[i] = a[1] \oplus a[2] \oplus \cdots \oplus a[i]$，
$a[p] \oplus a[p+1] \oplus \cdots \oplus a[n] \oplus x$
$= (a[1] \oplus \cdots \oplus a[p-1]) \oplus (a[1] \oplus \cdots \oplus a[p-1] \oplus a[p] \ldots \oplus a[n]) \oplus x$
$= s[p-1] \oplus s[n] \oplus x$
查询时，$s[n] \oplus x$ 是定值，令 $v = s[n] \oplus x$，
题目转化为：在区间 $[l-1, r-1]$中 与 v 异或的最大值。

我们用 **可持久化 01Trie** 维护 s[i] 即可。

可持久化字典树（Trie）

可持久化 Trie 和**可持久化线段树**的构造方式是相似的，利用**动态开点**，在前一个版本的基础上连边，每次只添加新节点，而保留没有被改动的节点。最终从每个历史版本的树根遍历，所分离出的 Trie 都是完整且包含前面全部信息的。如图，依次插入 0,5,7，构造**可持久化 01Trie** 。



先考虑查询 $[1,r]$ 的区间，即查询 $[0,r-1]$ 的区间。我们只需拿出 $r-1$ 版本的 Trie，查询的时候，尽量往当前位的相反位的地方跳，即可得到 $[0,r-1]$ 区间与定值v的异或最大值。

再考虑查询 $[l,r]$ 的区间，即查询 $[l-1,r-1]$ 的区间。我们仍然拿出 $r-1$ 版本的 Trie，查询的时候，尽量往当前位的相反位的地方跳，注意不能超过左侧边界 $l-1$，即可得到 $[l-1,r-1]$ 区间与定值的异或最大值。

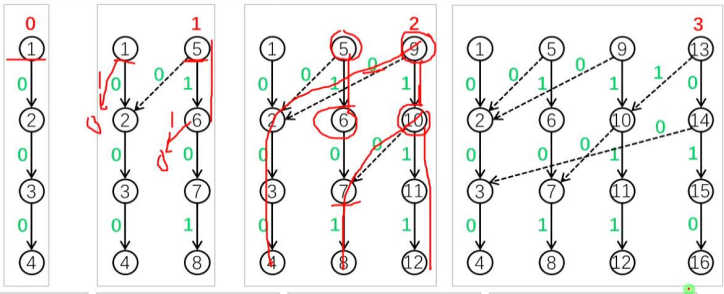
可持久化字典树（Trie）

```
const int N=600010, len=23;
int n, m, s[N];
int ch[N*25][2], ver[N*25];
int root[N], idx;
```

儿子：ch[1][0]=2, ch[1][1]=0
版本：ver[1,2,3,4]=0, ver[5,6,7,8]=1
根：root[0]=1, root[1]=5
因为 ai=10^7，位长度 len=23

insert(): x 为当前版本的根，y 为前一版本的根。
从高位到低位枚举，取出 s[i] 的第 k 位数字 c;
不是新节点就指向旧版本，是新节点就添加;
然后 x 走到新儿子，y 同步走到对应儿子。

```
void insert(int x,int y,int i){
  ver[x] = i;
  for(int k=len; k>=0; k--){
    int c = s[i]>>k&1;
    ch[x][!c]=ch[y][!c];
    ch[x][c] = ++idx;
    x=ch[x][c]; y=ch[y][c];
    ver[x] = i;
  }
}
```




```
ver[0] = -1; // 空节点0 的版本为-1
root[0] = ++idx;
insert(root[0],0,0);
for(int i=1; i <= n; i++){
  scanf("%d", &x);
  s[i] = s[i-1]^x;
  root[i] = ++idx;
  insert(root[i],root[i-1],i);
}
```

x: 5 2 5
s: 5 7 2

s[0]=0 (000)
root[0]=1
ch[1][1]=0
ch[1][0]=2
ch[2][1]=0
ch[2][0]=3
ch[2][1]=0
ch[3][0]=4

s[1]=5 (101)
root[1]=5
ch[5][0]=ch[1][0]=2
ch[5][1]=6
ch[6][1]=0
ch[6][0]=7
ch[7][0]=0
ch[7][1]=8

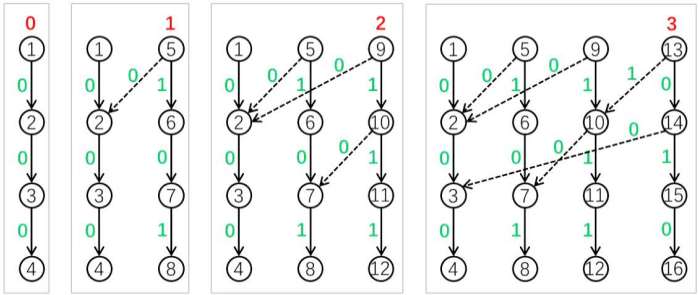
s[2]=7 (111)
root[2]=9
ch[9][0]=ch[5][0]=2
ch[9][1]=10
ch[10][0]=ch[6][0]=7
ch[10][1]=11
ch[11][0]=0
ch[11][1]=12

s[3]=2 (010)
root[3]=13
ch[13][1]=ch[9][1]=10
ch[13][0]=14
ch[14][0]=ch[2][0]=3
ch[14][1]=15
ch[15][1]=0
ch[15][0]=16

可持久化字典树（Trie）

```
int query(int x,int L,int v){
  int res=0;
  for(int k=len; k>=0; k--){
    int c = v>>k&1;
    if(ver[ch[x][!c]] >= L)
      x=ch[x][!c], res+=1<<k;
    else x=ch[x][c];
  }
  return res;
}
```

query(): x 为 r-1 版本的根，L 为搜索区的左边界，v 为定值。
从高位到低位枚举，取出 v 的第 k 位数字 c;
尽量走当前位的相反位，若不越界，则走过去，且加上该位的贡献;
若越界（或空节点），则走当前位的儿子，此位没贡献。




```
ver[0] = -1; // 空节点0 的版本为-1
```

```
scanf("%d%d%d",&l,&r,&x);
ans=query(root[r-1],l-1,s[n]^x);
printf("%d\n", ans);
```

3 2
5 2 5
(5 7 2)
Q 2 3 4
Q 1 3 4

Q 2 3 4
root[3-1]=9, l-1=1, s[3]^4=010^100=110
query(x=9, L=1, v=110)
k=2: c=1, (ch[9][0]=2, ver[2]=0, 0<L), x=10
k=1: c=1, (ch[10][0]=7, ver[7]=1, 1==L), x=7, res=2
k=0: c=0, (ch[7][1]=8, ver[8]=1, 1==L), x=8, res=3
return res=3

Q 1 3 4
root[3-1]=9, l-1=0, s[3]^4=010^100=110
query(x=9, L=0, v=110)
k=2: c=1, (ch[9][0]=2, ver[2]=0, 0==L), x=2, res=4
k=1: c=1, (ch[2][0]=3, ver[3]=0, 0==L), x=3, res=6
k=0: c=0, (ch[3][1]=0, ver[0]=-1, -1<L), x=4
return res=6

可持久化字典树（Trie）

```
void insert(int x,int y,int i){
  ver[x] = i;
  for(int k=len; k>=0; k--){
    int c = s[i]>>k&1;
    ch[x][!c]=ch[y][!c];
    ch[x][c] = ++idx;
    x=ch[x][c]; y=ch[y][c];
    ver[x] = i; // 节点 x 的版本为 i
  }
```

```
ver[0] = -1; // 空节点0 的版本为-1
root[0] = ++idx;
insert(root[0],0,0);
for(int i=1; i <= n; i++){
  scanf("%d", &x);
  s[i] = s[i-1]^x;
  root[i] = ++idx;
  insert(root[i],root[i-1],i);
}
```

```
int query(int x,int L,int v){
  int res=0;
  for(int k=len; k>=0; k--){
    int c = v>>k&1;
    if(ver[ch[x][!c]] >= L)
      x=ch[x][!c], res+=1<<k;
    else x=ch[x][c];
  }
  return res;
}
```

- ◆ 重点
- ➢ **区间和**的问题通常转化为**前缀和**。
- ➢ **异或最大值**：尽量走当前位的相反位
- ➢ **可持久化 Trie** ≡ n颗 **Trie** 的叠合
- ➢ 为什么插入空 Trie？为什么 ver[0]=-1？
- ➢ 时间复杂度：$O(n*23)$
- ➢ Luogu P4735 最大异或和
- ➢ https://www.cnblogs.com/dx123/
- ➢ QQ: 1253269950
- ➢ Tel: 15215363576

代码：

```
const int N = 600010, M = N * 25;

int n, m;
int s[N];
int tr[M][2], max_id[M];
int root[N], idx;

void insert(int i,int k,int p,int q)
{
    if(k<0){
        max_id[q] = i;
        return ;
    }
    int v=s[i]>>k&1;
    if(p)
        tr[q][v^1] = tr[p][v^1];
    tr[q][v] = ++idx;
    insert(i,k-1,tr[p][v],tr[q][v]);
    max_id[q] = max(max_id[tr[q][0]],max_id[tr[q][1]]);
}

int query(int root,int C,int L)
{
    int p=root;
    for(int i=23;i>=0;i--){
        int v=C>>i&1;
        if(max_id[tr[p][v^1]] >= L)
            p = tr[p][v^1];
        else
            p = tr[p][v];
    }
    return C^s[max_id[p]];
}

int main()
{
    IOS;
    cin >> n >> m;
    max_id[0] = -1;
    root[0] = ++idx;
    insert(0,23,0,root[0]);

    for(int i=1;i<=n;i++){
        int x;
        cin >> x;
        s[i] = s[i-1]^x;
        root[i] = ++idx;
        insert(i,23,root[i-1],root[i]);
    }

    while(m--){
        string op;
        cin >> op;
        if(op == "A"){
            int x;
```

```
            cin >> x;
            ++n;
            s[n] = s[n-1]^x;
            root[n] = ++idx;
            insert(n,23,root[n-1],root[n]);
        }
        else if(op == "Q"){
            int l,r,x;
            cin >> l >> r >> x;
            cout << query(root[r-1],s[n]^x,l-1) << endl;
        }
    }

    return 0;
}
```

AC自动机

原理：

模板：给定n个字符串，和一个文本，问文本里面出现了多少次不同的这n个字符串，每个字符串只算一次。

```
const int N = 1e6+10;

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

void insert(string str)
{
    int p=0;
    for(int i=0;i<str.size();i++){
        int u = str[i]-'a';
        if(!ch[p][u])
            ch[p][u] = ++idx;
        p = ch[p][u];
    }
    cnt[p]++;
}

void build_ac()
{
    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(string s)
{
    int ans=0;
    for(int k=0,i=0;k<s.size();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;
}

int main()
{
    IOS;
    int n;
    cin >> n;
    string str;
    rep(i,1,n){
        cin >> str;
        insert(str);
    }

    build_ac();
    cin >> str;
    cout << query(str) << endl;

    return 0;
}
```

简单应用：一个单词在原文中出现很多次，现在给你n个单词和 一个原文，问每个单词在原文中出现了多少次。这里直接考ac自动机中ne数组的含义，ne数组指的是与该点匹配的最长后缀。

```
const int N = 1e6+10;

string str[N];
int ch[N][26],idx;
int cnt[N],ne[N];
int q[N],dp[N];
int id[210];

void insert(string str,int x)
{
    int p=0;
    for(int i=0;i<str.size();i++){
        int u = str[i]-'a';
        if(!ch[p][u])
            ch[p][u] = ++idx;
```

```
        p = ch[p][u];
        dp[p]++;
    }
    cnt[p]++;
    id[x] = p;
}

void build_ac()
{
    int hh=0,tt=-1;
    for(int i=0;i<26;i++){
        if(ch[0][i])
            q[++tt] = ch[0][i];
    }

    while(hh<=tt){
        int u=q[hh++];
        for(int i=0;i<26;i++){
            int v=ch[u][i];
            if(v){
                ne[v] = ch[ne[u]][i];
                q[++tt] = v;
            }
            else
                ch[u][i] = ch[ne[u]][i];
        }
    }
}

int main()
{
    IOS;
    int n;
    cin >> n;
    rep(i,1,n){
        cin >> str[i];
        insert(str[i],i);
    }

    build_ac();
    for(int i=idx-1;i>=0;i--)
        dp[ne[q[i]]] += dp[q[i]];
    rep(i,1,n)
        cout << dp[id[i]] << endl;
    return 0;
}
```

后缀数组

后缀自动机

回文自动机

扩展kmp

相关概念：

lcp :最长公共前缀

Z函数：对应一个长度为n的字符串。$z[i]$表示S与其后缀$S[i,n]$的最长公共前缀的长度。

法一：暴力计算$O(n^2)$

法二：采用之前的状态来加速计算新的状态，即由$z[1],\dots,z[i-1]$快速计算$z[i]$

对于i，我们称区间$[i,i+z[i]-1]$是i的匹配段，也可以叫做$Z-box$,算法过程中维护右端点最靠右的匹配段，为了方便记作$[l,r]$，$S[l,r]$一定是S的前缀。计算过程中保证$l<=i$

算法流程：

计算完前$i-1$个z函数，维护盒子$[l，r]$，则$s[l,r]=s[1,r-l+1]$,就是说$[l,r]$区间与前缀是相等的，那如果该元素在$[l,r]$盒内，他肯定有相对应的位置在前缀，如果他对应的位置的z函数仍在$s[1,r-l+1]$内，那么就完全可以转移过来，如果已经超过盒子了，则需要从$r+1$暴力枚举。

1.如果$i<=r$(在盒内)，则有$s[i,r]=s[i-l+1,r-l+1]$(本质上是$[l,r]$与$[1,r-l+1]$相对应,i在$[1,r-l+1]$对应的位置是$i-l+1$)

(1)若$z[i-l+1]<r-i+1$则$z[i]=z[i-l+1]$

(2)若$z[i-l+1]>=r-i+1$则令$z[i]=r-i+1$,从$r+1$往后暴力枚举。

2.如果$i>r$则从i开始暴力枚举

3.求出$z[i]$后，如果$i+z[i]-1>r$,则更新盒子$l=i,r=i+z[i]-1$

时间复杂度$O(n)$

代码：

```
const int N = 1e5+10;
char str[N];
int z[N];

void get_z(int n)
{
    z[1] = n;
    for(int i=2,l,r=0;i<=n;i++){
        if(i<=r)
            z[i]=min(z[i-l+1],r-i+1);
        while(str[1+z[i]] == str[i+z[i]])
            z[i]++;
        if(i+z[i]-1>r)
            l=i,r=i+z[i]-1;
    }
}
```

模板：给定两个字符串 a,b，你要求出两个数组：

- b 的 z 函数数组 z，即 b 与 b 的每一个后缀的 LCP 长度。
- b 与 a 的每一个后缀的 LCP 长度数组 p。

  代码：

```
const int N = 2e7+10;

int z[N],p[N];
string a,b;
```

```
void get_z(string str)
{
    int n = str.size();
    str = " " + str;
    z[1] = n;
    for(int i=2,l,r=0;i<=n;i++){
        if(i<=r)
            z[i]=min(z[i-l+1],r-i+1);
        while(str[1+z[i]] == str[i+z[i]])
            z[i]++;
        if(i+z[i]-1>r)
            l=i,r=i+z[i]-1;
    }
}

void get_p(string a,string b)
{
    int n=a.size();
    int m=b.size();
    a = " " + a;
    b = " " + b;
    for(int i=1,l,r=0;i<=m;i++){
        if(i<=r)
            p[i] = min(z[i-l+1],r-i+1);
        while(1+p[i]<=n && 1+p[i]<=m && a[1+p[i]] == b[i+p[i]])
            p[i]++;
        if(i+p[i]-1>r)
            l = i,r = i+p[i]-1;
    }
}

int main()
{
    IOS;
    cin >> a >> b;
    get_z(b);
    get_p(b,a);

    ll ans1=0;
    ll ans2=0;
    for(int i=1;i<=b.size();i++)
        ans2 ^= 1ll*i*(z[i]+1);
    for(int i=1;i<=a.size();i++)
        ans1 ^= 1ll*i*(p[i]+1);
    cout << ans2 << endl << ans1 << endl;


    return 0;
}
```

$manacher$算法

理解：$d[i]$ 表示以 i 为中心的回文半径，在统计答案的时候最长的回文串长度就是 $max(d[i])-1$

```
const int N =1e7+1e6+10;
```

```
char a[N],s[N*2];
int d[N*2];

int main()
{
    scanf("%s",a+1);
    int n = strlen(a+1),k = 0;
    s[0] = '$',s[++k] = '#';
    for(int i=1;i<=n;i++){
        s[++k] = a[i];
        s[++k] = '#';
    }
    s[++k] = '^';
    n = k;

    d[1] = 1;
    for(int i=2,l,r=1;i<=n-1;i++){
        if(i<=r)
            d[i] = min(d[r-i+l],r-i+1);
        while(s[i-d[i]] == s[i+d[i]])
            d[i]++;
        if((i+d[i]-1)>r){
            r = i + d[i] - 1;
            l = i - d[i] + 1;
        }
    }

    int mx = 1;
    for(int i=1;i<n;i++){
        if(d[i] > mx)
            mx = d[i];
    }
    printf("%d\n",mx-1);

    return 0;
}
```

最小表示法

定义：最小表示法是用于解决字符串最小表示问题的方法。

循环同构：当字符串中可以选定一个位置$S[i\sim n]+S[1\sim i-1]=T$ 则T是S的循环同构串。设$S=$"$bcad$",其循环同构串有"$bcad$","$cadb$","$adbc$","$dbca$".

最小表示：字符串的最小表示为与循环同构的所有字符串中字典序最小的字符串,上面例子中最小表示为"$adbc$"。

算法过程：

1.把字符串复制一倍

2.初始化指针$i=1,j=2$，匹配长度$k=0$

3.比较$s[i+k]$和$s[j+k]$是否相等

(1) $s[i+k]==s[j+k]$,则$k++$

(2) $s[i+k]>s[j+k]$,则$i=i+k+1$

(3) $s[i+k] < s[j+k]$,则$j = j+k+1$

若跳转后两个指针相同，则$j++$,以确保比较的两个字符串不同。

4.重复上述过程，直到比较结束

5.答案为$min(i,j)$

代码：

```
/*
样例解释：
S=acacaba
acacabaacacaba
acacabaacacaba
当i=1,j=3,k=3时
第一个为acac
第二个为acab s[i+k]>s[j+k],则i跳转到i+k+1,从i~i+k的所有开始的都比第二个中的部分小，例如cac就没cab开始的小 ac就没有ab开始的小
*/
string get_m(string str)
{
    int n=str.size();
    rep(i,1,n){
        s[i] = str[i-1];
        s[n+i] = s[i];
    }

    int i=1,j=2,k=0;
    int ans = 0;
    while(i<=n&&j<=n){
        for(k=0;k<n&&s[i+k]==s[j+k];k++);
        s[i+k]>s[j+k]?i=i+k+1:j=j+k+1;
        if(i==j)
            j++;
    }
    ans = min(i,j);

    string res;
    for(int i=ans;i<=ans+n-1;i++)
        res += s[i];
    return res;
}
```

$Lyndon$分解

后缀树