0.1 字符串哈希

```
struct Hash
{
    ll mod,base,h[maxn],p[maxn],n;
    void make_hash()
    {
        p[0]=1;
        for(int i=1;i<=n;i++)
        {
            h[i]=((h[i-1]%mod*base%mod)%mod+s[i])%mod;
            p[i]=(p[i-1]*base)%mod;
        }
    }
    ll get_hash(ll l,ll r)
    {
        return (h[r]-(h[l-1]%mod*p[r-l+1]%mod)%mod+mod)%mod;
    }
}h;
```

允许k次失配的字符串匹配

模式串t与主串s进行匹配时，若不同的位置不大于k个，则认为匹配成功。

哈希+二分。枚举所有可能匹配的子串，假设现在枚举的子串为s'，通过哈希+二分可以快速找到s'与t第一个不同的位置。之后将s'与t在这个失配位置及之前的部分删除掉，继续查找下一个失配位置。这样的过程最多发生k次。时间复杂度$O(m+kn\log m)$。

最长回文子串

记R_i表示以i作为结尾的最长回文的长度，那么答案就是$\max_{i=1}^{n} R_i$。考虑到$R_{i-1}\le R_i+2$，因此我们只需要暴力从$R_{i-1}+2$开始递减，直到找到第一个回文即可。记变量z表示当前枚举的R_i，初始时为0，则z在每次i增大的时候都会增大2，之后每次暴力循环都会减少1，故暴力循环最多发生$2n$次，总的时间复杂度为$O(n)$。

最长公共子字符串

给定m个总长不超过n的非空字符串，查找所有字符串的最长公共子字符串。

二分最长公共子字符串的长度。假设现在的长度为k，check(k) 的逻辑为我们将所有所有字符串的长度为k的子串分别进行哈希，将哈希值放入n个哈希表中存储。之后求交集即可。时间复杂度$O(n \log \frac{n}{m})$。

0.2 字典树

```
void insert(char s[])
{
    int p=0,l=strlen(s);
    for(int i=0;i<l;i++)
    {
        int u=s[i]-'a';
        if(!trie[p][u]) trie[p][u]=++cnt;
        p=trie[p][u];
    }
    son[p]++;
}
int find(char s[])
{
    int p=0,l=strlen(s);
    for(int i=0;i<l;i++)
    {
        int u=s[i]-'a';
        if(!trie[p][u]) return -1;
        p=trie[p][u];
    }
    return son[p];
}
```

维护异或和

使用Trie维护多个数字的异或和可以实现：“插入”，“删除”，“全局加一”的功能。与使用Trie维护异或极值不同的是，如果要维护异或和，需要按值从低位到高位建立Trie。

插入&删除

如果要维护异或和，我们只需要知道某一位上0和1个数的奇偶性即可，也就是对于数字1来说，当且仅当这一位上数字1的个数为奇数时，这一位上的数字才是1。

num[u]是指字典树上到达u点的数量。xorv[u]指以u为根的子树维护的异或和，xorv[root]即维护的异或和。

```
void maintain(int p)
{
    num[p]=xorv[p]=0;
    if(trie[p][0])
        num[p]+=num[trie[p][0]],xorv[p]^=xorv[trie[p][0]];
    if(trie[p][1])
    {
        num[p]+=num[trie[p][1]],xorv[p]^=xorv[trie[p][1]];
        \\这一位为1且出现次数为奇数，则异或和为1
        if(num[trie[p][1]]&1) xorv[p]|=(1<<dep[p]);
    }
}
void insert(int p,int x)
{
    if(dep[p]>20) {num[p]++;return;}
    if(!trie[p][x&1]) trie[p][x&1]=++idx;
    dep[trie[p][x&1]]=dep[p]+1;
    insert(trie[p][x&1],x>>1);
    maintain(p);
}
void erase(int p,int x)
{
    if(dep[p]>20) {num[p]--;return;}
    erase(trie[p][x&1],x>>1);
    maintain(p);
}
```

全局加一

思考一下二进制意义下+1是如何操作的。我们只需要从低位到高位开始找第一个出现的0，把它变成1，然后这个位置后面的1都变成0即可。

```
void addall(int p) //传入字典树根节点，字典树维护全局加一
{
    swap(trie[p][0],trie[p][1]);
```

```
    if(trie[p][0]) addall(trie[p][0]);
    maintain(p);
}
```

字典树合并

字典树合并即将两个字典树的信息整合，合并为一个字典树。可以理解为将两个字符串上的串取出来，全部新加入到一个字符串。

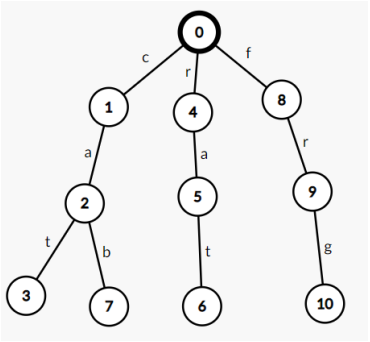
```
int merge(int a,int b)
{
    if(!a) return b; //无a选b
    if(!b) return a; //无b选a
    //a,b都有，将b的信息整合到a上
    num[a]+=num[b];xorv[a]^=xorv[b];
    trie[a][0]=merge(trie[a][0],trie[b][0]);
    trie[a][1]=merge(trie[a][1],trie[b][1]);
    return a;
}
```

可持久化字典树

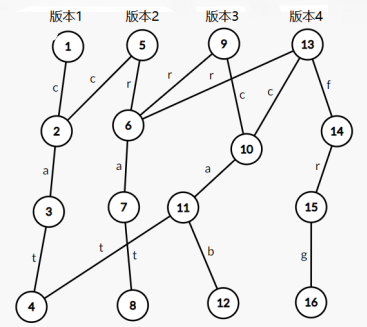
可持久化Trie就是记录了所有历史版本的字典树。

维护方式：将每次insert理解为额外开出一个全新的根节点，每一个根节点下的子树对应一个版本的字典树对于该根节点，将本次插入的数据完全新建，再将上一个版本的其他数据全部复制过来。

对cat、rat、cab、frg按顺序建普通Trie：



建立可持久化Trie：




```
void insert(int q,int p,int x)//将x插入版本p的根中，版本p的上一个版本为版本q
{
    max_id[q]=x; //max_id纪录了该位置下可以找到的最新版本编号
    for(int i=24;i>=0;i--)
    {
        int u=(sum[x]>>i)&1;
        if(q) trie[p][!u]=trie[q][!u];
        trie[p][u]=++idx;max_id[idx]=x;
        q=trie[q][u],p=idx;
    }
}
void query(int l,int r,int x)//查询第l个到第r个数字中异或x得到的最大值
{
    int p=root[r],ans=0;
    for(int i=24;i>=0;i--)
    {
        int u=(x>>i)&1;
        if(trie[p][!u] && max_id[trie[p][!u]]>=l)
        {
            ans+=(1<<i);p=trie[p][!u];
        }
        else p=trie[p][u];
    }
    printf("%d\n",ans);
}
```

0.3 KMP

```
void getnext(char S[])
```

```
{
    int l=strlen(S+1);
    Next[1]=0;
    for(int i=2,k=0;i<=l;i++)
    {
        while(k && S[i]!=S[k+1]) k=Next[k];
        if(S[i]==S[k+1]) k++;
        Next[i]=k;
    }
}
int kmp(char P[],char S[])
{
    int lp=strlen(P+1),ls=strlen(S+1);
    for(int i=1,j=1;i<=lp;i++) //i为主串指针，j为模式串指针
    {
        while(j>1 && P[i]!=S[j]) j=Next[j-1]+1;
        if(P[i]==S[j]) j++;
        if(j==ls+1) return i-ls+1;
    }
    return -1;
}
```

Border树

1.每个前缀Prefix[i]的所有Border：节点i到根的链。
2.哪些前缀有长度为x的Border：x的子树。
3.求两个前缀的公共Border等价于求LCA。

字符串的周期

p是S的周期=|S|−p是S的Border,S的Border的Border也是S的Border。

统计每个前缀的出现次数

统计每个前缀Prefix[i]在同一个字符串的出现次数。

```
for(int i=1;i<=n;i++) ans[nex[i]]++;
for(int i=n;i>=1;i--) ans[nex[i]]+=ans[i];
for(int i=1;i<=n;i++) ans[i]++;
```

考虑第二个问题，字符串S的每个前缀在字符串T中的出现次数是多少。构造字符串S+#+T，对新字符串长度≥ n的部分进行上面的操作即可。

0.4 拓展KMP

```
// exkmp(s,n,s,n,nxt,nxt) 求字符串s的Zbox
// exkmp(t,m,s,n,ext,nxt) 求字符串t的后缀与s的LCP
void exkmp(char *s,int lens,char *t,int lent,int *ext,int *
    nxt)
{
    ext[0]=0;
    for(int i=1,l=0,r=0;i<=lens;i++)
    {
        ext[i]=i<=r?min(nxt[i-l+1],r-i+1):0;
        while(i+ext[i]<=lens && ext[i]<lent && s[i+ext[i]]==
            t[ext[i]+1]) ext[i]++;
        if(i+ext[i]-1>=r && i!=1) l=i,r=i+ext[i]-1;
    }
}
```

0.5 AC自动机

KMP:一个模式串在一个主串上匹配。AC自动机：多个模式串在主串上匹配。

1.Trie树构建的复杂度是$O(m \times len)$，其中m为模式串数量，len为模式串平均长度。

2.构建fail指针时，时间也是线性的。

3.在匹配时，最耗时的是fail指针的跳动，每次最多前跳len次，若主串长度为n，那么总匹配时间复杂度为$O(n * len)$。

```
void insert(char s[])
{
    int p=0,l=strlen(s);
    for(int i=0;i<l;i++)
    {
        int u=s[i]-'a';
        if(!trie[p][u]) trie[p][u]=++cnt;
        p=trie[p][u];
```

```
    }
    son[p]++;
}
void make_fail()
{
    queue<int>q;
    for(int i=0;i<26;i++)
        if(trie[0][i]) q.push(trie[0][i]);
    while(!q.empty())
    {
        int t=q.front(); q.pop();
        for(int i=0;i<26;i++)
        {
            int p=trie[t][i];
            if(!p) trie[t][i]=trie[fail[t]][i];
            else
            {
                fail[p]=trie[fail[t]][i];
                q.push(p);
            }
        }
    }
}
int query(char s[]) //查询多少模式串在字符串s中出现过
{
    int nn=0;
    int num=0,l=strlen(s);
    for(int i=0,j=0;i<l;i++)
    {
        int t=s[i]-'a';j=trie[j][t];
        int p=j;
        while(p && ~son[p])
        {
            nn++;num+=son[p];
            son[p]=-1;p=fail[p];
        }
    }
    return num;
}
```

AC自动机上拓扑排序

假设当前的位置为p，这意味着我们当前匹配到了p以及p的fail链上的所有位置。所以对fail树进行拓扑排序可较快求解。

```
for(int i=1;i<=cnt;i++) d[fail[i]]++;
for(int i=1;i<=cnt;i++) if(d[i]==0) q.push(i);
while(!q.empty())
{
    int u=q.front();q.pop();
    d[fail[u]]--;ans[fail[u]]+=ans[u];
    if(d[fail[u]]==0) q.push(fail[u]);
}
```

last指针优化

在自动机上跳fail时，有时不得不遍历一些无用的fail结点，我们将fail链压缩，压缩为只包含关键结点的last指针。时间复杂度为$O(\sqrt{n})$。

AC自动机结合DP

通常先对son数组进行预处理，常见的有在BFS过程中：son[t]|=son[fail[t]]或son[t]+ =son[fail[t]]。

```
for(int i=1;i<=len;i++)
{
    for(int j=0;j<=cnt;j++)
    {
        for(int u=0;u<26;u++)
        {
            f[i][trie[j][u]]=max(f[i][trie[j][u]],f[i-1][j]+
                son[trie[j][u]]);
            ans=max(ans,f[i][trie[j][u]]);
        }
    }
}
```

fail树上DFS序+数据结构操作

由于AC自动机属于离线算法，有时候我们会遇到一些需要在线维护更新的题目。将AC自动机上的fail指针全部反过来，就得到了fail树。这种题目一般是利用fail树上的性质，配合数据结构（差分，树状数组，离线，树

剖）来变成树论中的数据结构问题。

遇到这种问题，首先要想清楚如何在fail树上处理单组询问，不断尝试用数据结构优化即可。

常见数据结构技巧：

1.询问某结点子树下的数字之和。使用DFS序将结点映射为序列，u结点代表的子树序列为$L[u], R[u]$（入栈时间和出栈时间）。

2.将某结点子树下的权值统一加一。可使用上述DFS序用树状数组区间修改。或者使用树状数组维护一个差分数组，u结点子树都加一，等于$L[x]$加1，$R[x]+1$减1。

3.将链上的值统一加一。使用树剖。或者用树状数组维护树上差分，差分后某点/边的值等于子树之和，用1求解。

0.6 Manacher

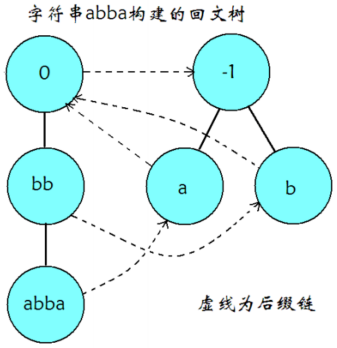
```
int manacher(char S[],int P[],char New[])
{
    int len=0,k=1,l=strlen(S+1);
    New[k]='#';
    for(int i=1;i<=l;i++)
    {
        New[++k]=S[i];New[++k]='#';
    }
    int mx=0,mid;
    for(int i=1;i<=k;i++)
    {
        if(i<mx) P[i]=min(mx-i,P[2*mid-i]);
        else P[i]=1;
        while(i-P[i]>=1 && i+P[i]<=k && New[i-P[i]]==New[i+P
            [i]]) P[i]++;
        if(i+P[i]>mx)
        {
            mx=i+P[i];mid=i;len=max(len,P[i]);
        }
    }
    return len-1; //找到的最长回文长度
}
```

0.7 回文自动机

对于一个字符串s，它的本质不同回文子串个数最多只有$|s|$个。因此回文树状态数是$|s|$的。对于每一个状态，它实际只代表一个本质不同的回文子串，即转移到该节点的状态唯一，因此总转移数也是$|s|$的。

和其它的自动机一样，一个节点的fail指针指向的是这个节点所代表的**回文串的最长回文后缀**所对应的节点，但是转移边并非代表在原节点代表的回文串后加一个字符，而是表示在原节点代表的回文串前后各加一个相同的字符。

回文树有两个初始状态，分别代表长度为$-1, 0$的回文串。我们可以称它们为奇根，偶根。它们不表示任何实际的字符串，仅作为初始状态存在。考虑构造完前$p-1$个字符的回文树后，向自动机中添加在原串里位置为p的字符。我们从以上一个字符结尾的最长回文子串对应的节点开始，不断沿着fail指针走，直到找到一个节点p满足s[i]=s[i-len[p]-1]，即满足此节点所对应回文子串的上一个字符与待添加字符相同。构造时间复杂度$O(|s|)$




```
struct PAM
{
    int len[maxn],ch[maxn][26],fail[maxn],cnt;
    char s[maxn];
    PAM()
    {
        len[1]=-1;len[0]=0;fail[1]=0;fail[0]=1;cnt=1;
    }
    int get_fail(int x,int i) //看看x的fail链上哪个可以接上s[i]
    {
        while(s[i-len[x]-1]!=s[i]) x=fail[x];
        return x;
    }
    void insert()
```

```
    {
        int l=strlen(s+1),p=0;
        for(int i=1;i<=l;i++)
        {
            int u=s[i]-'a',pos=get_fail(p,i);
            if(!ch[pos][u])
            {
                fail[++cnt]=ch[get_fail(fail[pos],i)][u];
                ch[pos][u]=cnt;len[cnt]=len[pos]+2;
            }
            p=ch[pos][s[i]-'a'];
        }
    }
}pam;
```

本质不同回文子串个数

一个串的本质不同回文子串个数等于回文树的状态数（排除奇根和偶根两个状态）。

回文子串出现次数

建出回文树，使用类似后缀自动机统计出现次数的方法。由于回文树的构造过程中，节点本身就是按照拓扑序插入，因此只需要逆序枚举所有状态，将当前状态的出现次数加到其fail指针对应状态的出现次数上即可。

```
void build()
{
    for(int i=cnt;i>=0;i--) sz[fail[i]]+=sz[i];
}
```

也可以在字符插入的过程维护nump[u]=nump[fail[u]]+1，即求出字符串当前位置结尾的字符串数量，统计插入过程的所有nump[u]之和即可。

双向插入PAM

由于回文串的特殊性，PAM在建造时可以从字符串一个位置出发，向左右两边用时插入字符，这需要维护两个结束结点pre和last，当插入字符后发现整个字符串就是一个回文串时，要同时更新pre和last。

```
void insert_front(int x,int c)
{
```

```
    while(s[x]!=s[x+len[pre]+1]) pre=fail[pre];
    if (!ch[pre][c])
    {
        len[++cnt]=len[pre]+2;
        int j=fail[pre];while(s[x+len[j]+1]!=s[x]) j=fail[j
            ];
        fail[cnt]=ch[j][c];ch[pre][c]=cnt;
        nump[cnt]=nump[fail[cnt]]+1;
    }
    pre=ch[pre][c];
    if(len[pre]==r-l+1) last=pre; //若本身为回文串更新last
    sum=sum+nump[pre]; //所有回文串总数
}
void insert_back(int x,int c)
{
    while(s[x]!=s[x-len[last]-1]) last=fail[last];
    if (!ch[last][c])
    {
        len[++cnt]=len[last]+2;
        int j=fail[last];while(s[x-len[j]-1]!=s[x]) j=fail[j
            ];
        fail[cnt]=ch[j][c];ch[last][c]=cnt;
        nump[cnt]=nump[fail[cnt]]+1;
    }
    last=ch[last][c];
    if(len[last]==r-l+1) pre=last; //若本身为回文串更新pre
    sum=sum+nump[last]; //所有回文串总数
}
```

0.8 后缀数组

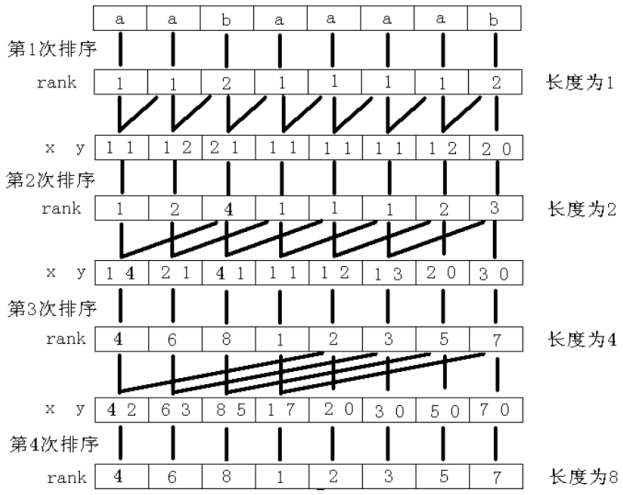
将字符串的所有后缀进行字典序排序，令$sa[i]$代表排名第i的后缀字符串的起始位置的下标，$rk[i]$代表从第i位开始的后缀字符串的排名，这两个数组满足性质：$sa[rk[i]] = rk[sa[i]] = i$。

0.8.1 倍增

使用倍增的思想，先对每个长度为1的子串（即每个字符）进行排序。$S[i,k]$表示从i开始，长度为2^k的字符串，则$S[i,k+1] = S[i,k] + S[i+$

$2^k, k]$。

用上一轮排好的顺序，两两组合，进行双关键字排序（$pair$排序），即可用长度为k的排名推出长度为$k+1$的排名。



如果使用$sort$进行排序，共需排序$\log n$次，时间复杂度$O(n \log^2 n)$，使用基数排序将排序时间优化至$O(n)$，于是总体时间复杂度被优化到了$O(n \log n)$。

```
struct Suffix_Array
{
    char s[maxn];
    int n,m=122,x[maxn],y[maxn];
    int rk[maxn],sa[maxn],height[maxn],c[maxn];
    void get_SA() //记得对n初始化
    {
        memset(height,0,sizeof(height));
        memset(rk,0,sizeof(rk));
        memset(y,0,sizeof(y));
        for(int i=1;i<=m;i++) c[i]=0;
        for(int i=1;i<=n;i++) c[x[i]=s[i]]++;
        for(int i=1;i<=m;i++) c[i]+=c[i-1];
        for(int i=n;i>=1;i--) sa[c[x[i]]--]=i; //基数排序先计算出长度为1子串的顺序
        for(int k=1;k<=n;k<<=1)
        {
            int num=0;
            //y[i]表示第二关键字排名为i的数，第一关键字的位置
            //第n-k+1到第n位是没有第二关键字的,所以排名在最前面
            for(int i=n-k+1;i<=n;i++) y[++num]=i;
            for(int i=1;i<=n;i++) if(sa[i]>k) y[++num]=sa[i
```

```
                ]-k;
22              for(int i=1;i<=m;i++) c[i]=0;
23              for(int i=1;i<=n;i++) c[x[i]]++;
24              for(int i=1;i<=m;i++) c[i]+=c[i-1];
25              //因为y的顺序是按照第二关键字的顺序来排的
26              //第二关键字靠后的，在同一个第一关键字桶中排名越靠后
27              for(int i=n;i>=1;i--) sa[c[x[y[i]]]--]=y[i],y[i
                    ]=0;
28              for(int i=1;i<=n;i++) swap(x[i],y[i]);
29              x[sa[1]]=1;num=1;
30              //合并并列排名
31              for(int i=2;i<=n;i++)
32             x[sa[i]]=(y[sa[i]]==y[sa[i-1]]&&y[sa[i]+k]==y[sa[
                   i-1]+k])?num:++num;
33              if(num==n) break; m=num;
34          }
35          int k=0;height[1]=0;
36          for(int i=1;i<=n;i++) rk[sa[i]]=i;
37          for(int i=1;i<=n;i++)
38          {
39              if(rk[i]==1) continue;
40              if(k) k--;
41              int j=sa[rk[i]-1];
42              while(i+k<=n&&j+k<=n&&s[i+k]==s[j+k]) k++;
43              height[rk[i]]=k;
44          }
45      }
46  }s1,s2;
```

0.8.2 SA-IS

```
1   struct SAIS
2   {
3       int sa[maxn],rk[maxn],s[maxn*2],op[maxn*2],pos[maxn*2];
4       int c1[maxn],c[maxn],ht[maxn];
5       char str[maxn];
6       #define L(x) sa[c[s[x]]--]=x
7       #define R(x) sa[c[s[x]]++]=x
8       inline void sa_sort(int *S,int n,int m,int *s,int *op,
            int tn)
9       {
10          for(int i=1;i<=n;i++) sa[i]=0;
```

```
11          for(int i=1;i<=m;i++) c1[i]=0;
12          for(int i=1;i<=n;i++) c1[s[i]]++;
13          for(int i=2;i<=m;i++) c1[i]+=c1[i-1];
14          for(int i=1;i<=m;i++) c[i]=c1[i];
15          for(int i=tn;i;i--) L(S[i]);
16          for(int i=1;i<=m+1;i++) c[i]=c1[i-1]+1;
17          for(int i=1;i<=n;i++)
18          if(sa[i]>1 && op[sa[i]-1]) R(sa[i]-1);
19          for(int i=1;i<=m;i++) c[i]=c1[i];
20          for(int i=n;i;i--)
21          if(sa[i]>1 && !op[sa[i]-1]) L(sa[i]-1);
22      }
23      void SA_IS(int n,int m,int *s,int *op,int *pos)//m代表字符
            的范围
24      {
25          int tot=0,cnt=0;int *S=s+n;
26          op[n]=0;
27          for(int i=n-1;i;i--) op[i]=(s[i]!=s[i+1])?s[i]>s[i
                +1]:op[i+1];
28          rk[1]=0;
29          for(int i=2;i<=n;i++)
30          if(op[i-1]==1 && op[i]==0) pos[++tot]=i,rk[i]=tot;
31          else rk[i]=0;
32          sa_sort(pos,n,m,s,op,tot);
33          int u=0,p=0;
34          for(int i=1;i<=n;i++)
35          if(rk[sa[i]])
36          {
37              u=rk[sa[i]];
38              if(cnt<=1 || pos[u+1]-pos[u]!=pos[p+1]-pos[p])
                    ++cnt;
39              else
40              {
41                  for(int j=0;j<=pos[u+1]-pos[u];j++)
42                  if(s[pos[u]+j]!=s[pos[p]+j]||op[pos[u]+j]!=
                        op[pos[p]+j])
43                  {++cnt;break;}
44              }
45              S[u]=cnt;
46              p=u;
47          }
48          if(tot!=cnt) SA_IS(tot,cnt,S,op+n,pos+n);
49          else for(int i=1;i<=tot;i++) sa[S[i]]=i;
```

```
50          for(int i=1;i<=tot;i++) S[i]=pos[sa[i]];
51          sa_sort(S,n,m,s,op,tot);
52      }
53      void get_ht(int n) //ht[1]=1
54      {
55          for(int i=1;i<=n;i++) rk[sa[i]=sa[i+1]]=i;
56          for(int i=1,p=0;i<=n;ht[rk[i]]=p,i++)
57          if(rk[i]!=1)
58          for(p=p-!!p;sa[rk[i]-1]+p<=n && i+p<=n&&s[i+p]==s[sa
                [rk[i]-1]+p];p++);
59      }
60      void Get_SA(int n)
61      {
62          for(int i=1;i<=n;i++) s[i]=str[i];
63          s[++n]=1;SA_IS(n--,122,s,op,pos);//122为字符串的ASCII码
                范围
64          get_ht(n);
65      }
66  }sa;
```

寻找最小的循环移动位置

将字符串S复制一份变成SS就转化成了后缀排序问题。

从字符串首尾取字符最小化字典序

需要在原串后缀与反串后缀构成的集合内比较大小，可以将反串拼接在原串后，并在中间加上一个没出现过的字符（如#，代码中可以直接使用空字符），求后缀数组，即可$O(1)$完成这一判断。

LCP（最长公共前缀）

定义$LCP(i,j)$为$suff(i)$和$suff(j)$的最长公共前缀。那么$height[i] = LCP(sa[i], sa[i-1])$，即第$i$名的后缀与它前一名的后缀的最长公共前缀，$height[1] = 0$。

$height[rk[i]] \geq height[rk[i-1]] - 1$。

两子串最长公共前缀

$LCP(sa[i], sa[j]) = \min\{height[i+1..j]\}$，于是就转换为了RMQ问题。

```
1   void get_ST()
```

```
{
    for(int i=1;i<=n;i++) st[i][0]=height[i];
    for(int j=1;j<16;j++)
    {
        for(int i=1;i+(1<<(j-1))<=n;i++)
        {
            st[i][j]=min(st[i][j-1],st[i+(1<<(j-1))][j-1]);
        }
    }
}
int LCP(int l,int r)
{
    l=rk[l],r=rk[r];
    if(l>r) swap(l,r); l++;
    int k=log2(r-l+1);
    return min(st[l][k],st[r-(1<<k)+1][k]);
}
```

比较一个字符串的两个子串的大小关系

假设需要比较的是$A[a..b]$和$B[c..d]$的大小关系。

若$LCP(a,c) \geq \min(|A,|B|)$，$A < B \iff |A| < |B|$。否则，$A < B \iff rk[a] < rk[c]$。

不同子串的数目

子串就是后缀的前缀，所以可以枚举每个后缀，计算前缀总数，再减掉重复。“前缀总数”其实就是子串个数，为$n(n+1)/2$。

如果按后缀排序的顺序枚举后缀，每次新增的子串就是除了与上一个后缀的LCP剩下的前缀。这些前缀一定是新增的，否则会破坏$LCP(sa[i],sa[j])$的性质。只有这些前缀是新增的，因为LCP部分在枚举上一个前缀时计算过了。

所以答案为$n(n+1)/2-\sum_{i=2}^{n} height[i]$。

连续的若干个相同子串

我们可以枚举连续串的长度$|S|$，按照$|S|$对整个串进行分块，对相邻两块的块首进行LCP与LCS查询。

字符串多少子串可写成AABB形式

```
for(int len=1;len<=n/2;len++)
{
    for(int i=1;i<=n;i+=len)
    {
        int l=i,r=i+len;int lcp=s1.LCP(l,r);lcp=min(lcp,len)
            ;
        int L=n-r+2,R=n-l+2;int lcs=s2.LCP(L,R);lcs=min(lcs,
            len-1);
        if(lcs+lcp>=len)
        {
            int t=lcs+lcp-len;
            b[i-lcs]++;b[i-lcs+t+1]--;
            a[r+lcp-1-t]++;a[r+lcp]--;
        }
    }
}
for(int i=1;i<=n;i++) a[i]+=a[i-1],b[i]+=b[i-1];
ll ans=0; for(int i=1;i<n;i++) ans+=a[i]*b[i+1];
```

结合并查集

某些题目求解时要求你将后缀数组划分成若干个连续LCP长度大于等于某一值的段，亦即将h数组划分成若干个连续最小值大于等于某一值的段并统计每一段的答案。如果有多次询问，我们可以将询问离线。观察到当给定值单调递减的时候，满足条件的区间个数总是越来越少，而新区间都是两个或多个原区间相连所得，且新区间中不包含在原区间内的部分的h值都为减少到的这个值。我们只需要维护一个并查集，每次合并相邻的两个区间，并维护统计信息即可。

结合单调栈

有些时候我们需要计算$\sum LCP$，使用单调栈，找到每个$height$向左向右可以覆盖的有效范围，直接计算即可。

```
vector<int>st;
for(int i=2;i<=SA.n;i++)
{
    while(!st.empty() && SA.height[st.back()]>SA.height[i])
        st.pop_back();
    if(st.empty()) l[i]=1;
    else l[i]=st.back();
    st.pb(i);
}
st.clear();
for(int i=SA.n;i>=2;i--)
{
    while(!st.empty() && SA.height[st.back()]>=SA.height[i])
        st.pop_back();
    if(st.empty()) r[i]=SA.n+1;
    else r[i]=st.back();
    st.pb(i);
}
for(int i=2;i<=SA.n;i++) ans+=(r[i]-i)*(i-l[i])*SA.height[i
    ];
```

0.9 后缀自动机

在SAM中状态数最多有$2*n-1$个，转移数最多$3n-4$。

- s的子串可以根据它们结束的位置endpos被划分为多个等价类。
- SAM由初始状态t_0和与每一个endpos等价类对应的每个状态组成。
- 对于每一个状态v，一个或多个子串与之匹配。我们记longest(v)为其中最长的一个字符串，记len(v)为它的长度。类似地，记shortest(v)为最短的子串，它的长度为minlen(v)。那么对应这个状态的所有字符串都是字符串longest(v)的不同的后缀，且所有字符串的长度恰好覆盖区间[minlen(v), len(v)]中的每一个整数。
- 对于任意不是t_0的状态v，定义后缀链接为连接到对应字符串longest(v)的长度为minlen$(v)-1$的后缀的一条边。从根节点t_0出发的后缀链接可以形成一棵树。这棵树也表示endpos集合间的包含关系。
- 对于t_0以外的状态v，可用后缀链接link(v)表达minlen(v)：minlen$(v)=$ len(link(v)) $+1$。

- 如果我们从任意状态v_0开始顺着后缀链接遍历，总会到达初始状态t_0。这种情况下我们可以得到一个互不相交的区间$[\text{minlen}(v_i), \text{len}(v_i)]$的序列，且它们的并集形成了连续的区间$[0, \text{len}(v_0)]$。

```
struct SAM
{
    int idx,last,len[maxn*2],link[maxn*2],nex[maxn*2][26],sz
        [maxn*2];
    SAM()
    {
        idx=0;last=0;len[0]=0;link[0]=-1;
    }
    void insert(int c)
    {
        int cur=++idx;
        len[cur]=len[last]+1;sz[cur]=1;
        int p=last;
        while(p!=-1 && !nex[p][c]) nex[p][c]=cur,p=link[p];
        if(p==-1) link[cur]=0;
        else
        {
            int q=nex[p][c];
            if(len[p]+1==len[q]) link[cur]=q;
            else
            {
                int clone=++idx;
                len[clone]=len[p]+1;link[clone]=link[q];sz[
                    clone]=0;
                for(int i=0;i<26;i++) nex[clone][i]=nex[q][i
                    ];
                while(p!=-1 && nex[p][c]==q) nex[p][c]=clone
                    ,p=link[p];
                link[q]=link[cur]=clone;
            }
        }
        last=cur;
    }
}sam;
```

结点代表的子串出现的次数即字符串上的结尾位置

```
int c[2*maxn],a[2*maxn];
void build(int n)
{
    int now=1;
    for(int i=1;i<=n;i++) endpos[now=nex[now][s[i]-'a']]=i;
    for(int i=1;i<=idx;i++) c[len[i]]++;
    for(int i=1;i<=n;i++) c[i]+=c[i-1];
    for(int i=idx;i>=1;i--) a[c[len[i]]--]=i;
    for(int i=idx;i>=1;i--)
    {
        int pos=a[i];sz[link[pos]]+=sz[pos];
        if(!endpos[link[pos]]) endpos[link[pos]]=endpos[pos
            ];
    }
}
```

不同子串个数

每个节点对应的子串数量是$\text{len}(i)-\text{len}(\text{link}(i))$，对自动机所有节点求和即可。

所有不同子串的总长度

每个节点对应的所有后缀长度是$\frac{\text{len}(i)\times(\text{len}(i)+1)}{2}$，减去其link节点的对应值就是该节点的净贡献，对自动机所有节点求和即可。

字典序第k大子串

字典序第k大的子串对应于SAM中字典序第k大的路径，因此在计算每个状态的路径数后，我们可以很容易地从SAM的根开始找到第k大的路径。预处理的时间复杂度为$O(|S)$，单次查询的复杂度为$O(|ans|\cdot|\Sigma|)$（其中$|ans|$是查询的答案，$|\Sigma|$为字符集的大小）。

```
void dfs(int pos,int num)
{
    if(num<=sz[pos]) return;
    num-=sz[pos];
    for(int i=0;i<26;i++)
    {
        int p=nex[pos][i];
        if(!p) continue;
        if(sum[p]<num) num-=sum[p];
```

```
        else
        {
            printf("%c",'a'+i);
            dfs(p,num);return;
        }
    }
}
void build(int n)
{
    for(int i=1;i<=idx;i++) c[len[i]]++;
    for(int i=1;i<=n;i++) c[i]+=c[i-1];
    for(int i=idx;i>=1;i--) a[c[len[i]]--]=i;
    for(int i=idx;i>=1;i--)
    {
        int pos=a[i];sz[link[pos]]+=sz[pos];
        if(t==0) sz[link[pos]]=1;//t为0则表示不同位置的相同子串算作一个
    }
    sz[0]=0;
    for(int i=0;i<=idx;i++) sum[i]=sz[i];
    for(int i=idx;i>=0;i--)
    {
        for(int j=0;j<26;j++)
        {
            int pos=nex[a[i]][j];
            if(!pos) continue;
            sum[a[i]]+=sum[pos];
        }
    }
    if(sum[0]<k){puts("-1");return;}
    dfs(0,k);
}
```

最小循环移位

容易发现字符串$S+S$包含字符串S的所有循环移位作为子串。贪心地访问最小的字符即可。总的时间复杂度为$O(|S|)$。

两个字符串的最长公共子串

以其中一个字符串建立SAM，另一个串在SAM上转移。

```
void solve()
```

```
{
    scanf("%s",s+1);
    int n=strlen(s+1),p=0,l=0,ans=0;
    for(int i=1;i<=n;i++)
    {
        while(p&&!nex[p][s[i]-'a']) p=link[p],l=len[p];
        if(nex[p][s[i]-'a']) p=nex[p][s[i]-'a'],l++,ans=max(
            ans,l);
    }
    printf("%d\n",ans);
}
```

多个字符串间的最长公共子串

以其中一个字符串建立SAM，其余串在SAM上转移。

```
void build()
{
    for(int i=1;i<=idx;i++) c[len[i]]++;
    for(int i=1;i<=idx;i++) c[i]+=c[i-1];
    for(int i=idx;i>=1;i--) a[c[len[i]]--]=i;
}
void solve()
{
    memset(mn,0x3f,sizeof(mn));build();
    while(scanf("%s",s+1)!=EOF)
    {
        int n=strlen(s+1),pos=0,l=0;
        for(int i=1;i<=n;i++)
        {
            while(pos&&!nex[pos][s[i]-'a']) pos=link[pos],l=
                len[pos];
            if(nex[pos][s[i]-'a']) pos=nex[pos][s[i]-'a'],l
                ++;
            mx[pos]=max(mx[pos],l);
        }
        for(int i=idx;i>=1;i--)
        {
            int now=a[i],fa=link[now];
            mx[fa]=max(mx[fa],min(mx[now],len[fa]));
            mn[now]=min(mn[now],mx[now]);mx[now]=0;
        }
```

```
    }
    int ans=0;
    for(int i=1;i<=idx;i++) ans=max(ans,mn[i]);
    printf("%d\n",ans);
}
```

区间本质不同字符串个数

```
#include<bits/stdc++.h>
using namespace std;
typedef long long ll;
const int maxn = 2e5 + 100;
/* 维护最后出现位置在（左端点）的本质不同串数量i */
struct SegmentTree_Sum{
    ll Sum[maxn * 8],Lazy[maxn*8];
    void down(int x,int l,int mid,int r){
        Sum[x<<1] += Lazy[x] * (mid - l + 1);
        Sum[x<<1|1] += Lazy[x] * (r - mid);
        Lazy[x<<1] += Lazy[x];
        Lazy[x<<1|1] += Lazy[x];
        Lazy[x] = 0;
    }
    void up(int x){Sum[x] = Sum[x<<1] + Sum[x<<1|1];}
    void update(int x,int l,int r,int L,int R,int val){
        if (l > R or L > r)return;
        if (L <= l and r <= R){
            Sum[x] += 1ll * val * (r - l + 1);
            Lazy[x] += val;
            return;
        }
        int mid = l + r >> 1;down(x,l,mid,r);
        update(x<<1,l,mid,L,R,val);update(x<<1|1,mid+1,r,L,R
            ,val);
        up(x);
    }
    ll query(int x,int l,int r,int L,int R){
        if (l > R or L > r)return 0;
        if (L <= l and r <= R)return Sum[x];
        int mid = l + r >> 1;down(x,l,mid,r);
        return query(x<<1,l,mid,L,R) + query(x<<1|1,mid+1,r,
            L,R);
    }
```

```
33 }segtree;
34 struct SegmentTree_Max{
35     int Max[maxn*8];
36     void update(int x,int l,int r,int pos,int val){
37         Max[x] = max(Max[x],val);
38         if (l == r)return;
39         int mid = l + r >> 1;
40         if (pos <= mid)update(x<<1,l,mid,pos,val);
41         else update(x<<1|1,mid+1,r,pos,val);
42     }
43     int query(int x,int l,int r,int L,int R){
44         if (l > R or L > r)return -1;
45         if (L <= l and r <= R)return Max[x];
46         int mid = l + r >> 1;
47         return max(query(x<<1,l,mid,L,R),query(x<<1|1,mid+1,
               r,L,R));
48     }
49 }dfstree;
50 int n,q;
51 char s[maxn];
52 ll ans[maxn];
53 typedef pair<pair<int,int>,int> Query;
54 vector<Query> query;
55 struct Suffix_Automaton{
56     int nxt[maxn*2][26],fa[maxn*2],l[maxn*2];
57     int last,cnt;
58     /* 每个最上边一个点color */
59     int up_to[maxn];
60     /* 是否被染过色 */
61     bool used[maxn*2];
62     Suffix_Automaton(){ clear(); }
63     void clear(){
64         last =cnt=1;fa[1]=l[1]=0;
65         memset(nxt[1],0,sizeof nxt[1]);
66     }
67     void init(char *s){
68         while (*s){add(*s-'a');s++;}
69     }
70     void add(int c){
71         int p = last;
72         int np = ++cnt;
73         memset(nxt[cnt],0,sizeof nxt[cnt]);
```

```
74          l[np] = l[p]+1;last = np;
75          while (p&&!nxt[p][c])nxt[p][c] = np,p = fa[p];
76          if (!p)fa[np]=1;
77          else{
78              int q = nxt[p][c];
79              if (l[q]==l[p]+1)fa[np] =q;
80              else{
81                  int nq = ++ cnt;
82                  l[nq] = l[p]+1;
83                  memcpy(nxt[nq],nxt[q],sizeof (nxt[q]));
84                  fa[nq] =fa[q];fa[np] = fa[q] =nq;
85                  while (nxt[p][c]==q)nxt[p][c] =nq,p = fa[p];
86              }
87          }
88      }
89      vector<int> E[maxn * 2];
90      int in[maxn*2],out[maxn*2],dfn;
91      void dfs(int u){
92          in[u] = ++dfn;
93          for (int v:E[u])dfs(v);
94          out[u] = dfn;
95      }
96      void gao(){
97          for (int i=2;i<=cnt;i++)E[fa[i]].push_back(i);
98          dfs(1);
99          for (int i=1,now = 1;i<=n;i++){
100             now = nxt[now][s[i] - 'a'];
101             assert(l[now] == i);
102             segtree.update(1,1,n,1,i,1);
103             int u = now;
104             while (u != 1 and !used[u]){
105                 used[u] = true;
106                 u = fa[u];
107             }
108             while (u != 1){
109                 int cur = dfstree.query(1,1,cnt,in[u],out[u
                        ]);
110                 segtree.update(1,1,n,cur - l[u]+1,cur - l[
                        up_to[cur]],-1);
111                 swap(up_to[cur],u);
112             }
113             dfstree.update(1,1,cnt,in[now],i);
```

```
114             up_to[i] = 1;
115             while (!query.empty() and query.back().first.
                    second == i){
116                 int l = query.back().first.first;
117                 int id = query.back().second;
118                 ans[id] = segtree.query(1,1,n,l,i);
119                 query.pop_back();
120             }
121         }
122     }
123 }sam;
124 int main(){
125     scanf("%s%d",s+1,&q);n=strlen(s+1);
126     sam.init(s+1);
127     for(int i=1;i<=q;i++){
128         int l,r;scanf("%d%d",&l,&r);
129         query.push_back({{l,r},i});
130     }
131     sort(query.begin(),query.end(),[](Query x,Query y){
132         return x.first.second > y.first.second;
133     });
134     sam.gao();
135     for(int i=1;i<=q;i++) cout<<ans[i]<<endl;
136 }
```

0.10 广义后缀自动机

后缀自动机是用于处理单个字符串的子串问题的强力工具。而广义后缀自动机则是将后缀自动机整合到字典树中来解决对于多个字符串的子串问题。

0.10.1 伪广义后缀自动机

1. 通过用特殊符号将多个串直接连接后，再建立SAM。
2. 对每个串，重复在同一个SAM上进行建立，每次建立前，将last指针置零。

实现方式简单，而且在面对题目时通常可以达到和广义后缀自动机一样的正确性。但是时间复杂度较为危险。

0.10.2 构造广义后缀自动机

1. 将所有字符串插入到字典树中。
2. 从字典树的根节点开始进行BFS，记录下顺序以及每个节点的父亲节点。
3. 将得到的BFS序列按照顺序，对每个节点在原字典树上进行构建，注意不能将len小于当前len的数据进行操作。

使用广义后缀自动机解决问题时，建议不要在建立自动机时打标记，最好等自动机建好后遍历字符串打标机，或者直接建立$link$树DFS。

```
struct GSA
{
    int nex[maxn][26],idx,len[maxn*2],link[maxn*2];
    GSA()
    {
        link[0]=-1;
    }
    void insert_trie(char *s)
    {
        int p=0,l=strlen(s+1);
        for(int i=1;i<=l;i++)
        {
            int u=s[i]-'a';
            if(!nex[p][u]) nex[p][u]=++idx;
            p=nex[p][u];
        }
    }
    int insert_sam(int last,int u)
    {
        int cur=nex[last][u];
        if(len[cur]) return cur;
        len[cur]=len[last]+1;
        int p=link[last];
        while(p!=-1 && !nex[p][u]) nex[p][u]=cur,p=link[p];
        if(p==-1) link[cur]=0;
        else
        {
            int q=nex[p][u];
            if(len[p]+1==len[q]) link[cur]=q;
            else
            {
```

```
32                  int clone=++idx;
33                  len[clone]=len[p]+1;
34                  link[clone]=link[q];
35                  for(int i=0;i<26;i++)
36                      nex[clone][i]=len[nex[q][i]]!=0?nex[q][i
                            ]:0;
37                  while(p!=-1 && nex[p][u]==q) nex[p][u]=clone
                        ,p=link[p];
38                  link[q]=link[cur]=clone;
39              }
40          }
41          return cur;
42      }
43      void build()
44      {
45          queue<pair<int,int>>q;
46          for(int i=0;i<26;i++) if(nex[0][i]) q.push({0,i});
47          while(!q.empty())
48          {
49              auto item=q.front();q.pop();
50              int last=insert_sam(item.first,item.second);
51              for(int i=0;i<26;i++) if(nex[last][i]) q.push({
                    last,i});
52          }
53      }
54  }sam;
```

0.11 最小表示法

循环同构

当字符串S中可以选定一个位置i满足$S[i\cdots n]+S[1\cdots i-1]=T$则称$S$与$T$循环同构。

最小表示

字符串S的最小表示为与S循环同构的所有字符串中字典序最小的字符串。

算法核心

考虑对于一对字符串在原字符串S中的起始位置分别为i, j，且它们的前k个字符均相同，即$S[i \cdots i+k-1] = S[j \cdots j+k-1]$。假设$S[i+k] > S[j+j]$，我们发现起始位置下标$l$满足$i \le l \le i+k$的字符串均不能成为答案。因为对于任意一个字符串$S_{i+p}$一定存在字符串$S_{j+p}$比它更优。所以我们比较时可以跳过下标$l \in [i, i+k]$, 直接比较$S_{i+k+1}$。时间复杂度$O(n)$。

```
int minshow()//下标从开始，方便取模0
{
    int k=0,i=0,j=1;
    while(k<n&&i<n&&j<n)
    {
        if(s[(i+k)%n]==s[(j+k)%n]) k++;
        else
        {
            s[(i+k)%n]>s[(j+k)%n]?i=i+k+1:j=j+k+1;
            if(i==j) i++; k=0;
        }
    }
    return min(i,j);
}
```

0.12 Lyndon分解

对于字符串S，如果S的字典序严格小于S的所有后缀的字典序，我们称S为Lyndon串。

串S的Lyndon分解记为$S = w_1 w_2 \cdots w_k$，其中所有w_i为Lyndon串，并且他们的字典序按照非严格单减排序，即$w_i \ge w_{i+1}$。可以发现，这样的分解存在且唯一。

Duval算法

如果一个字符串t能够分解为$t = ww \cdots \overline{w}$的形式，其中$w$是一个Lyndon串，而$\overline{w}$是$w$的前缀（可能是空串），那么称$\overline{w}$是近似Lyndon串。一个Lyndon串也是近似Lyndon串。

Duval算法运用了贪心的思想。算法过程中我们把串S分成三个部分$S = s_1 s_2 s_3$，其中s_1是一个Lyndon串，它的Lyndon分解已经记录；s_2是一个近似Lyndon串；s_3是未处理的部分。

定义一个指针i指向s_2的首字符，则i从1遍历到n（字符串长度）。在循

环的过程中我们定义另一个指针j指向s_3的首字符，指针k指向s_2中我们当前考虑的字符（意义是j在s_2的上一个循环节中对应的字符）。我们的目标是将$s[j]$添加到s_2的末尾，这就需要将$s[j]$与$s[k]$做比较：

1. 如果$s[j] = s[k]$，则将$s[j]$添加到s_2末尾不会影响它的近似简单性。于是我们只需要让指针j, k自增（移向下一位）即可。

2. 如果$s[j] > s[k]$，那么$s_2s[j]$就变成了一个Lyndon串，于是我们将指针j自增，而让k指向s_2的首字符，这样s_2就变成了一个循环次数为1的新Lyndon串了。

3. 如果$s[j] < s[k]$，则$s_2s[j]$就不是一个近似简单串了，那么我们就要把s_2分解出它的一个Lyndon子串，这个Lyndon子串的长度将是$j - k$，即它的一个循环节。然后把s_2变成分解完以后剩下的部分，继续循环下去（注意，这个情况下我们没有改变指针j, k），直到循环节被截完。对于剩余部分，我们只需要将进度「回退」到剩余部分的开头即可。

时间复杂度$O(n)$。

```
vector<string> duval(string s)
{
    int n=s.size(),i=0;
    vector<string> factorization;
    while(i<n)
    {
        int j=i+1,k=i;
        while(j<n&&s[k]<=s[j])
        {
            if(s[k]<s[j]) k=i;
            else k++;
            j++;
        }
        while(i<=k)
        {
            factorization.push_back(s.substr(i,j-k));
            i+=j-k;
        }
    }
    return factorization;
}
```