1. 1.**后缀数组**

Str ：需要处理的字符串(长度为Len)

Suffix[i] ：Str下标为i ~ Len的连续子串(即后缀)

Rank[i] : Suffix[i]在所有后缀中的排名

SA[i] : 满足Suffix[SA[1]] < Suffix[SA[2]] …… < Suffix[SA[Len]],即排名为i的后缀为Suffix[SA[i]] (与Rank是互逆运算)

**后缀数组**指的就是这个**SA[i]**,有了它，我们就可以实现一些很强大的功能(如不相同子串个数、连续重复子串等)。如何快速的到它，便成为了这个算法的**关键**。而**SA**和**Rank**是互逆的，只要求出任意一个，另一个就可以O(Len)得到

对于一个后缀**Suffix[i]**,如果想直接得到**Rank**比较困难，但是我们可以对每个字符开始的长度为2k2𝑘的字符串求出排名，k从0开始每次递增1(每递增1就成为**一轮**)，当2k2𝑘大于Len时，**所得到的序列就是Rank，而SA也就知道了**

1. **后缀自动机**

定义：字符串 S𝑆 的 SAM 是一个可以接受 S𝑆 的所有后缀的**最小** DFA （确定性有限状态自动机）。

SAM 最常用的功能是储存一个文本串 S𝑆 的**每一个子串**，这也是我们在题目主要需要利用的性质。

值得注意的是，对于长度为 O(n)𝑂(𝑛) 的字符串 S𝑆， SAM 可以在在 O(n)𝑂(𝑛) 的空间内储存 S𝑆 所有子串的信息，另外，构造 SAM 的算法的时间复杂度仅为 O(n)𝑂(𝑛) 。

构造：设当前字母为c，从上一个点las转移过来，当前新开一个点np，endpos多出一个{n}来先考虑转移，为了最大程度的利用节点，只对las的祖先节点中，计当前祖先节点为p，若不存在字母c的转移的点，对其进行转移到np的操作，并令p回跳到p的父亲，最终p停留在了某个节点

代码：

#include<iostream>

#include<cstring>

#include<algorithm>

using namespace std;

struct SAM{

static const int N=1e5+10;

struct NODE{

int ch[26]; // 每个节点添加一个字符后到达的节点

int len; // 每个节点代表的状态中的最大串长

int fa; // 每个节点在parent树上的父亲节点, 即该节点link边指向的节点

int sz; // 每个节点对应的endpos集合的大小（即串的出现次数），等于所有parent树上儿子的大小

NODE(){memset(ch,0,sizeof(ch));len=0;}

}dian[N<<1]; // 节点数开串长的两倍

int n; // 串长

char s[N]; // 串

int las=1; // las: 上一个用到的节点编号

int tot=1; // tot: 当前用到的节点编号

// 向SAM中插入一个字符c

void add(int c){

int p=las; // 上一个状态的节点

int np=las=++tot; // 要加入的状态的节点

dian[np].sz=1; // 叶子节点endpos大小为1

dian[np].len=dian[p].len+1; // 新状态比上一个装填多一个首字符

for(;p&&!dian[p].ch[c];p=dian[p].fa)dian[p].ch[c]=np; // 指针p沿link边回跳,直至找到一个节点包含字符c的出边，无字符c的出边则将出边指向新状态的节点

if(!p)dian[np].fa=1;// 以上为case 1，指针p到SAM的起点的路径上的节点都无字符c的出边，将新节点作为SAM的起点的一个儿子节点

else{ // 节点p包含字符c的出边

int q=dian[p].ch[c]; // 节点p的字符c的出边指向的节点

if(dian[q].len==dian[p].len+1)dian[np].fa=q;// 以上为case 2，节点p和q代表的状态的最大串长相差1

else{ // 节点p和q代表的状态的最大串长相差>1

int nq=++tot; // 新建一个节点nq

dian[nq]=dian[q]; // 节点nq克隆节点q的信息

dian[nq].sz=0; // nq产生时，是一个分支节点，需要从后续儿子节点里更新获取sz

dian[nq].len=dian[p].len+1; // 保证节点p和nq代表的状态的最大串长相差1

dian[q].fa=dian[np].fa=nq;

for(;p&&dian[p].ch[c]==q;p=dian[p].fa)dian[p].ch[c]=nq;// 以上为case 3，将节点p到SAM的起点的路径上的所有节点的字符c的出边指向的节点替换为nq

}

}

}

void init(){

scanf("%s",s);

n=strlen(s);

for(int i=0;i<n;i++)add(s[i]-'a');

}

int b[N<<1],a[N<<1]; // b: 用于基数排序 a: 用于记录点号

// 按长度基数排序，短的在前长的在后

// 另一种方法是用vector直接建出parent树，对parent树直接dfs

void base\_sort(){

for(int i=1;i<=tot;++i)b[dian[i].len]++;

for(int i=1;i<=tot;++i)b[i]+=b[i-1];

for(int i=1;i<=tot;++i)a[b[dian[i].len]--]=i;

}

// 用于逆拓扑序遍历求出sz

void get\_sz(){

for(int i=tot;i>=1;--i){

int u=a[i],fa=dian[u].fa;

dian[fa].sz+=dian[u].sz;

}

}

void solve(){

init();

base\_sort();

get\_sz();

}

}sam;

int main(){

sam.solve();

return 0;

}

1. AC自动机

## 字典树（trie）

构造与操作：

创建一个公共的父节点结构体（一般来说），包含接下来可能出现的所有字母对应的子节点指针数组，以及bool类型的end标记一个完整单词的结尾。

接下来就是插入，通过遍历目标单词的每一个字母，向下深入，若没有子节点则直接创立子节点再深入。直到该单词被遍历完打上end标记。

查找等操作此处略，已无难点。

字典树数组构造方法、

    void insert(char \*s)

    {

        long root = 0, lens = strlen(s);

        for (long i = 0; i < lens; i++)

        {

            long temp = s[i] - 'a';

            if (!trie[root][temp])

                trie[root][temp] = ++index;

            root = trie[root][temp];

        }

        end[root]++;//构造字典树

    }

指针构造方法、

#include<bits/stdc++.h>

using namespace std;

typedef struct trienode{

    struct trienode\*childr[27];

    int end;

}Trie;

Trie\* createTrie(void)

{

    Trie \*node=(Trie \*)malloc(sizeof(Trie));

    for(int i=0;i<26;i++)

    node->childr[i]=NULL;

    node->end=0;

    return node;

};

void insert(Trie \*header,const char \*p1)

{

    Trie \*cur=header;

    for(int i=0;i<strlen(p1);i++)

    {

        int temp=p1[i]-'a';

        if(!cur->childr[temp])

        cur->childr[temp]=createTrie();

        cur=cur->childr[temp];

    }

    cur->end=1;

}

int search(Trie \*header,const char \*p1)

{

    Trie \*cur=header;

    for(int i=0;i<strlen(p1);i++)

    {

        int temp=p1[i]-'a';

        if(!cur->childr[temp])

        return 0;

        cur=cur->childr[temp];

    }

    if(!cur||!cur->end)

        return 0;

    else

        return 1;

}

int main()

{

    Trie\* head=createTrie();

    insert(head,"i");

    insert(head,"in");

    insert(head,"intern");

    insert(head,"inward");

    printf("%d\n",search(head,"ink"));

    printf("%d\n",search(head,"intern"));

    printf("%d\n",search(head,"inward"));

    system("pause");

}

## KMP

1. 运用有限状态自动机的原理，使用dp数组进行记录每个状态下遇见每一个字母该转移到什么状态。构造这样的数组只需要用到模式串本身。
2. 上述为一个方向，经典的做法是先求出部分匹配表（PMT），再进行匹配，有趣的是，匹配算法和求出PMT的算法虽然是不相同的，但都利用了求一个字符串的最长公共前后缀，有dp的思想。

PMT数组（也叫失配指针数组）每一项的定义就是从开始到当前的位置这一串字符串最长的前缀和后缀交集的元素长度

KMP代码如下：#include <bits/stdc++.h>

using namespace std;

void nexCreate(const char \*s1, long \*nex)

{

    nex[0] = 0;

    long long i = 0, j = 1; // i指向模式串，j指向目标串，代表的是当前模式长度对应的nex值

    long long temlen = strlen(s1);

    while (j <= temlen)

    {

        if (s1[i] == s1[j])

        {

            nex[j] = i + 1;

            i++;

            j++;

        }

        else if (i != 0)

            i = nex[i - 1];//跳转至有i,j对应字母之前有相同前缀的位置

        else

        {

            nex[j] = 0;

            j++;

        }

    }

}

int search(char \*txt, char \*pat)//nex数组记录的不仅是最长公共前后缀的长度，也是减少回溯的定位指针，可谓精妙

{

    long kmpnex[1000050] = {0};

    nexCreate(pat, kmpnex);

    long long j = 0, i = 0, patlen = strlen(pat), txtlen = strlen(txt);

    while (i < txtlen)

    {

        if (txt[i] == pat[j])

        {

            i++;

            j++;

        }

        else if (j)

            j = kmpnex[j - 1];

        else

            i++;

    }

    if (j == strlen(pat))

        return 1;

    else

        return 0;

}

int main()

{

    char str[2000], substr[2000];

    scanf("%s", str);

    scanf("%s", substr);

    printf("%s", search(str, substr) ? "GOTIT" : "NULL");

    system("pause");

}

## AC自动机

首先创建所有模式串的trie树，其次构造失配指针，最后进行模式匹配。

AC自动机代码如下：#include <bits/stdc++.h>

#include <queue>

using namespace std;

#define maxn 1000005

struct Trie

{

    long trie[maxn][26], fail[maxn], end[maxn];//那么该字符串的所有后缀必然也能与该文本串匹配成功。因此，当一个字符串匹配成功，即可通过fail指针在子树之间跳跃，快速查找所有同样能够匹配成功的后缀，从而避免回溯，节省大量时间

    long index = 0;

    void insert(char \*s)

    {

        long root = 0, lens = strlen(s);

        for (long i = 0; i < lens; i++)

        {

            long temp = s[i] - 'a';

            if (!trie[root][temp])

                trie[root][temp] = ++index;

            root = trie[root][temp];

        }

        end[root]++;//构造字典树

    }

    void failbuild(void) // 把trie[now][]当作子节点的指针，now为当前节点的指针

    {

        queue<long> q;

        for (long i = 0; i < 26; i++)

            if (trie[0][i]) // 若根节点有对应的子结点指针

            {

                fail[trie[0][i]] = 0;

                q.push(trie[0][i]);

            }

        while (!q.empty())//bfs一层层的遍历

        {

            long now = q.front();

            q.pop();

            for (long i = 0; i < 26; i++)

                if (trie[now][i])//若子节点存在

                {

                    fail[trie[now][i]] = trie[fail[now]][i];//其实trie[fail[temp]][i]要么就是最长后缀节点要么就是根节点

                    q.push(trie[now][i]);//压入子节点

                }

                else//若子节点不存在

                    trie[now][i] = trie[fail[now]][i]; // 在后缀最长情况下尽量延长最长后缀长度，当前节点的子节点接到了

        }

    }

    void query(char \*s)

    {

        long now=0,ans=0;

        long lens=strlen(s);

        for(long i=0;i<lens;i++)

        {

            now=trie[now][s[i]-'a'];//状态转移得到当前正在识别的前缀，43行为此准备，能够进行随时跳转

            for(long j=now;j&&end[j]!=-1;)//对于当前得到的字符串，得到其相同后缀的单词个数

            {

                ans+=end[j];

                end[j]=-1;//得做记忆化

                j=fail[j];

            }

        }

        printf("%ld",ans);

    }

};

struct Trie ac;

int main()

{

    long n;

    char pat[maxn],text[maxn];

    scanf("%ld",&n);

    while(n--)

    {

        scanf("%s",pat);

        ac.insert(pat);

    }

    ac.failbuild();

    scanf("%s",text);

    ac.query(text);

    system("pause");

}

1. 主席树

#### 权值线段树

##### 定义：线段树的特定形式，维护一段区间的数出现的次数，有点像桶排序建立的数据结构

##### 应用：一般用于求区间内第K小的数

#### 动态开点

##### 操作：1.对节点访问不再有数学关系，而是根据需要的点为每个节点赋予逐渐增大的编号，所以需要结构体记录左右子节点的编号；2.一般只会是单点更新

##### 每更新一个节点就相当于创造一条链

##### update函数使用传引用方法可简化代码（&用于传引用）

# 主席树

1. 先建立一个所有权值为零的权值线段树（也就是把数组初始为零）
2. 从左到右依次遍历每一个无序列表中的值，执行插入操作
3. 插入操作：也就是动态开点，但要为每一次插入操作分配一个根节点，且为该根节点分配上一根节点的左右节点，代表历史线段树，然后递归创建新线段树。所以插入操作不仅是动态开点，还要继承前一个历史线段树的一部分。每次操作最多分配logn+1个节点（也就是线段树的一条链），前提是离散化。
4. 查询操作：访问[l,r]的第K小个数，首先将[1,l-1]的历史记录从[1,r]中删除，然后进行类似于桶排序的查询，就结束了。

代码（静态区间第K小）：

#include<bits/stdc++.h>

using namespace std;

#define ll long long

#define maxn 2001

#define INF 0x3f3f3f3f

struct node {

    ll val;

    ll order;

}a[maxn];

ll ls[maxn<<5],rs[maxn<<5],sum[maxn<<5],b[maxn],rts[maxn],at[maxn];

ll M,N;

ll tot=0;

int cmp(struct node p1,struct node p2)

{

    return p1.val<p2.val;

}

int build(int l,int r)

{

    int root=++tot;

    if(l==r)

    {

        return root;

    }

    int mid=(l+r)>>1;

    ls[root]=build(l,mid);

    rs[root]=build(mid+1,r);

    return root;

}

int update(int rt,int l,int r,int x)

{

    int root=++tot;

    ls[root]=ls[rt];

    rs[root]=rs[rt];

    sum[root]=sum[rt]+1;

    if(l==r)

    {

        return root;

    }

    int mid=(l+r)>>1;

    if(x<=mid)

    ls[root]=update(ls[rt],l,mid,x);

    else

    rs[root]=update(rs[rt],mid+1,r,x);

    return root;

}

ll query(int u,int v,int l,int r,int k)

{

    int temp=sum[ls[u]]-sum[ls[v]];

    int mid=(l+r)>>1;

    if(l==r)

    return l;

    if(k<=temp)

    return query(ls[u],ls[v],l,mid,k);

    else

    return query(rs[u],rs[v],mid+1,r,k-temp);

}

ll init(void)

{

    sort(a+1,a+N+1,cmp);

    sort(at+1,at+N+1);

    ll cnt=1;

    b[a[1].order]=1;

    for(ll i=2;i<=N;i++)

    if(a[i].val==a[i-1].val)

    b[a[i].order]=cnt;

    else

    b[a[i].order]=++cnt;

    unique(at+1,at+N+1);

    unique(b+1,b+N+1);

    return cnt;

}

int main()

{

    ll le,ri,ki,num;

    scanf("%lld%lld",&N,&M);

    for(ll i=1;i<=N;i++)

    {

        scanf("%lld",&a[i].val);

        a[i].order=i;

        at[i]=a[i].val;

    }

    num=init();

    rts[0]=build(1,num);

    for(ll i=1;i<=N;i++)

    rts[i]=update(rts[i-1],1,num+1,b[i]);

    for(ll i=0;i<M;i++)

    {

        scanf("%lld%lld%lld",&le,&ri,&ki);

        ll tem=query(rts[ri],rts[le-1],1,num+1,ki);

        printf("%lld\n",at[tem]);

    }

    system("pause");

}