ACM Template

Zeng Xiaocan

March 26, 2019

Contents

1 字符串

1.1 KMP

```
/*
 * 初始化 nex 数组
 * nex[i]：表示前 i 个字符的最长相同前后缀长度
 */
void getNext(char s[],int n){
    int i=0,j=-1;
    nex[0]=-1;
    while(i<n){
        if(j==-1 || s[i]==s[j]){
            nex[++i]=++j;
        }else{
            j=nex[j];
        }
    }
}
/*
 * KMP 匹配 (位置/个数 (可重叠/不可重叠))
 * 循环节:
 *     前 i 个字符的最小循环节长度: i-nex[i]
 *     循环节个数: i/(i-nex[i])
 */
int kmp(char s[],int n,char p[],int m){
    int i=0,j=0;
    // int cnt=0;
    getNext(p,m);
    while(i<n && j<m){
        if(j==-1 || s[i]==p[j]){
            i++;
            j++;
        }else{
            j=nex[j];
        }
        if(j==m){
            //匹配位置
            return i-j+1;
            //匹配个数
            //cnt++;
            //不可重叠
            //j=0;
            //可重叠
            //j=nex[j];
        }
    }
    //return cnt;
}
```

1.2 Trie 树/AC 自动机

```
/*
 * 字典树/tr 自动机
 */
struct node{
    int son[26];
    //根据需要维护一些奇奇怪怪的东西
    //被查询次数
    int cnt;
    //是否完整单词
    bool flag;
    //作为前缀次数
    int sum;
    //=======AC 自动机 =======
    //模式串下标
    int end;
    //失配指针，表示以当前失配的字符串的最长后缀字符串
    //比如匹配到 abcd，d 失配，那么 c 的 fail 指针指向的就是后缀为 abc(或者 bc 或者 c) 的最长字符
    int fail;
}tr[N];
int tot;
void init(){
    tot=0;
    memset(tr,0,sizeof(tr));
}
/*
 * 插入字符串构建 Trie 树
 */
void insert(char *s,int num){
    int len=strlen(s);
    int rt=0;
    for(int i=0;i<len;i++){
        int id=s[i]-'a';
        if(!tr[rt].son[id]){
            tr[rt].son[id]=++tot;
        }
        rt=tr[rt].son[id];
        tr[rt].sum++;
    }
    //标记完整单词
    tr[rt].flag=true;
    tr[rt].end=num;
}
/*
 * 字典树简单查询：以所给串作为前缀个模式串个数
 */
int search(char *s){
    int len=strlen(s);
    int rt=0;
```

```
    int ans=0;
    for(int i=0;i<len;i++){
        int id=s[i]-'a';
        if(!tr[rt].son[id]){
            //not found
            return 0;
        }
        rt=tr[rt].son[id];
    }
    return tr[rt].sum;
}
/*
 * 构建 tr 自动机 fail 指针
 */
void getFail(){
    queue<int> q;
    //先处理第一层，都指向根
    for(int i=0;i<26;i++){
        if(tr[0].son[i]!=0){
            tr[tr[0].son[i]].fail=0;
            q.push(tr[0].son[i]);
        }
    }
    //bfs 处理
    while(!q.empty()){
        int u=q.front();
        q.pop();
        for(int i=0;i<26;i++){
            if(tr[u].son[i]!=0){
                //子节点的 fail 指向父节点 fail 的对应子节点
                tr[tr[u].son[i]].fail=tr[tr[u].fail].son[i];
                //记得入队
                q.push(tr[u].son[i]);
            }else{
                //不存在这个子节点，指向 fail 指针的对应子节点
                tr[u].son[i]=tr[tr[u].fail].son[i];
            }
        }
    }
}
/*
 * tr 自动机简单查询：每个模式串在所给串中分别出现的次数
 * ans[]：结构体数组 {pos,num} 表示模式串的下标和出现次数
 */
void query(char s[]){
    int len=strlen(s);
    int rt=0;
    for(int i=0;i<len;i++){
        int id=s[i]-'a';
```

```
        rt=tr[rt].son[id];
        //沿着失配路径到根
        //比如两个模式串 abcd 和 bcd, 所给字符串 abcdbcd
        //一开始直接匹配到 abcd 出现一次, 然后这时候沿着 fail 找到 bcd,
        //所以 bcd 也出现了一次, 同理直到根
        //就能把出现的子串都记录一遍
        for(int j=rt;j!=0;j=tr[j].fail){
            ans[tr[j].end].num++;
        }
    }
}
```

1.3 Manacher

```
/*
 * 求字符串 s 的最大回文子串
 * ma[]: 新字符串 (ma,mp 都注意要开两倍空间!)
 * mp[i]: 表示以 i 为中心的回文子串的半径 (包括特殊字符)
 * mx: 能延伸到最右端的位置
 * id: 能延伸到最右端的回文串中心位置
 */
int manacher(char s[],int len){
    //构造新字符串 两个字符之间插入一个其他字符
    //第 0 个字符忽略 (即加入另一种字符)
    int l=0;
    ma[l++]='$';
    ma[l++]='#';
    for(int i=0;i<len;i++){
        ma[l++]=s[i];
        ma[l++]='#';
    }
    ma[l]='\0';
    int mx=0,id=0;
    for(int i=1;i<l;i++){
        //递推部分 比较 mx 和 i 的位置
        //右边:2*id-i 表示 i 以 id 的对称点 因为不能超出 mx 所以要和 mx-i 取 min
        //左边: mp[i]=1
        mp[i]=mx>i?min(mp[2*id-i],mx-i):1;
        //更新部分
        //往两边更新
        while(ma[i+mp[i]]==ma[i-mp[i]]){
            mp[i]++;
        }
        //更新全局 mx 和 id
        if(i+mp[i]>mx){
            mx=i+mp[i];
            id=i;
        }
    }
    return *max_element(mp,mp+l)-1;
```

```
}
```

1.4 后缀数组

```
//"banana" 后缀为 [banana anana nana ana na a $]
//sa[i]: 排名第 i(从 0 开始) 小的后缀的首字符下标
//比如 [6 5 3 1 0 4 2]==>[$ a$ ana$ anana$ banana$ na$ nana$]
//rk[i]: 下标 i 开始的后缀 (不含 $) 的排名 (按字典序从小到大, 相当于 sa 的逆)
//[4 3 6 2 5 1]
//h[i]: 排名为 i 的后缀和排名为 i-1 的后缀的最长公共前缀
//[1(ana$-a$) 3(anana$-ana$) 0 0 2]
//一些应用:
//*** 求两个后缀的最长公共前缀
// 两个后缀的首字符编号为 a,b, LCP(a,b)
//*** 不同字串个数, 由于子串一定是某个后缀的前缀, 所以 h 数组就是 lcp, 也是是相同前缀个数
//所以 ans=n*(n+1)/2-\sum(h)
//*** 出现次数大于等于 k 次的最大子串长度
//二分子串长度, 调用 check(mid,k)
//*** 出现次数大于等于 l 小于等于 r 的子串个数
//RMQ+ 容斥  RMQ 维护 h[] 数组的区间最小值 (省略)  调用 solve(l,r)
//*** 最长不重叠重复出现 (2 次以上) 子串
//二分子串长度, 调用 check(mid)
char s[N];
int sa[N],rk[N],h[N];
//辅助数组
int t[N],t2[N],c[N];
void build_sa(int n, int m){
    //n 为字符串的长度, 字符集的值为 0~m-1
    //相当于在后面加一个 $
    n++;
    int *x=t, *y=t2;
    //基数排序
    for(int i=0;i<m;i++){
        c[i]=0;
    }
    for(int i=0;i<n;i++){
        c[x[i]=s[i]]++;
    }
    for(int i=1;i<m;i++){
        c[i]+=c[i-1];
    }
    //或者 ~i 表示 i!=-1
    for(int i=n-1;i>=0;i--){
        sa[--c[x[i]]]=i;
    }
    for(int k=1; k<=n; k<<=1){
        int p=0;
        for(int i=n-k;i<n;i++){
            y[p++]=i;
        }
```

```
        for(int i=0;i<n;i++){
            if(sa[i]>=k){
                y[p++]=sa[i]-k;
            }
        }
        //类似上面，只是把 i 换成 y[i]
        for(int i=0;i<m;i++){
            c[i]=0;
        }
        for(int i=0;i<n;i++){
            c[x[y[i]]]++;
        }
        for(int i=1;i<m;i++){
            c[i]+=c[i-1];
        }
        for(int i=n-1;i>=0;i--){
            sa[--c[x[y[i]]]]=y[i];
        }
        swap(x, y);
        p=1;
        x[sa[0]]=0;
        for(int i=1;i<n;i++){
            x[sa[i]]=y[sa[i-1]]==y[sa[i]] && y[sa[i-1]+k]==y[sa[i]+k]? p-1 : p++;
        }
        if (p>=n){
            break;
        }
        m = p;
    }
    //去掉 $
    n--;
    for(int i = 0; i <= n; i++){
        rk[sa[i]] = i;
    }
    //计算 h
    int k=0;
    for(int i = 0; i < n; i++){
        if(k){
            k--;
        }
        int j = sa[rk[i] - 1];
        while(s[i + k] == s[j + k]){
            k++;
        }
        h[rk[i]] = k;
    }
}
int LCP(int a,int b){
    a=rk[a];
```

```
    b=rk[b];
    if(a>b){
        swap(a,b);
    }
    return ask(a,b);
}
//判断是否存在长度为 mid 的子串出现次数大于等于 k
bool check(int mid,int k){
    int cnt=1;
    for(int i=1;i<=n;i++){
        if(h[i]>=mid){
            cnt++;
        }else{
            cnt=1;
        }
        if(cnt>=k){
            return true;
        }
    }
    return false;
}
//判断是否存在长度为 mid 的不重叠重复子串
bool check2(int mid){
    int low=INF,hight=-INF;
    bool flag=false;
    for(int i=2;i<=n;i++){
        if(h[i]<mid){
            //不连续 断开，相当于把所有后缀分组
            //例如 aabaaaab
            //最后后缀和排名和 h 数组是
            /*
              $
              aaaab
            3 aaab
            2 aab
            3 aabaaaab
            ----------------
            1 ab
            2 abaaaab
            ----------------
            0 b
            ----------------
            1 baaaab
            */
            //当 k=2 时分组如上所示，分组就是把有最大相同前缀的后缀放在一起，然后比较这个公共的前
            //如果大于等于 mid，说明存在，如果小于，说明这一组已经结束了
            //重置 low hight 判断下一组 (h 数组的连续性)
            low=INF;
            hight=-INF;
```

```
        }else{
            //更新相同 LCP 的一组后缀中的最左和最右的 sa 值
            low=min(low,min(sa[i],sa[i-1]));
            hight=max(hight,max(sa[i],sa[i-1]));
            if(hight-low>=mid){
                flag=true;
                break;
            }
        }
    }
    return flag;
}
//查询排名区间在 [l,r] 的后缀的最长公共前缀
//查询排名区间在 [l,r] 的后缀含有的出现次数大于等于 r-l+1 次的子串个数
int ask(int l,int r){
    //因为 h[i] 就是表示排名为 i 和排名为 i-1 的后缀的最长公共前缀
    //RMQ 得到的就是 [l,r] 这一段所有排名后缀的最长公共前缀
    if(l==r){
        //特判 l==r，即求的是排名为 r 的后缀的长度，即 n-sa[r]
        return n-sa[r];
    }else{
        return RMQ(l+1,r);
    }
}
//查询出现次数在 [l,r] 区间内的子串个数
//设 f(k) 表示出现次数大于等于 k 的子串个数，即 f(k)=ask(区间长度为 k)
//答案为 f(l)-f(r+1)
//因此我们枚举长度为 l 的区间，再从两端构造出 r+1 的区间，利用容斥定理解决
int solve(int l,int r){
    initRMQ(n);
    int ans=0;
    //枚举区间
    for(int i=1;i+l-1<=n;i++){
        //查询排名 [i,i+l-1] 这一段的后缀出现次数大于等于 l 的子串个数
        ans+=ask(i,i+l-1);
        //这里固定减 1
        if(i-1>0){
            //同理，减去出现次数至少为 l+1 的
            ans-=ask(i-1,i+r-1);
        }
        if(i+r<=n){
            //减去出现次数至少为 r+1 的
            ans-=ask(i,i+r);
        }
        if(i-1>0 && i+r<=n){
            //容斥，加上出现次数至少为 r+2 的，因为被上面两个 if 重复减去
            ans+=ask(i-1,i+r);
        }
    }
```

```
    return ans;
}
```

1.5 后缀数组经典应用

```
#include <bits/stdc++.h>
using namespace std;
const int N=1005;
const int INF=0x3f3f3f3f;
int n;
//"banana" 后缀为 [banana anana nana ana na a $]
//sa[i]: 排名第 i(从 0 开始) 小的后缀的首字符下标
//比如 [6 5 3 1 0 4 2]==>[$ a$ ana$ anana$ banana$ na$ nana$]
//rk[i]: 下标 i 开始的后缀 (不含 $) 的排名 (按字典序从小到大, 相当于 sa 的逆)
//[4 3 6 2 5 1]
//h[i]: 排名为 i 的后缀和排名为 i-1 的后缀的最长公共前缀
//[1(ana$-a$) 3(anana$-ana$) 0 0 2]
//一些应用:
//-- 求两个后缀的最长公共前缀
// 两个后缀的首字符编号为 a,b, LCP(a,b)
//-- 不同字串个数, 由于子串一定是某个后缀的前缀, 所以 h 数组就是 lcp, 也是是相同前缀个数
//所以 ans=n*(n+1)/2-\sum(h)
//-- 出现次数大于等于 k 次的最大子串长度
//二分子串长度, 调用 check(mid,k)
//-- 出现次数大于等于 l 小于等于 r 的子串个数
//RMQ+ 容斥 RMQ 维护 h[] 数组的区间最小值 (省略) 调用 solve(l,r)
//-- 最长不重叠重复出现 (2 次以上) 子串
//未完待续
char s[N];
int sa[N],rk[N],h[N];
//辅助数组
int t[N],t2[N],c[N];
int dp[N][30];
void build_sa(int n, int m){
    //n 为字符串的长度, 字符集的值为 0~m-1
    //相当于在后面加一个 $
    n++;
    int *x=t, *y=t2;
    //基数排序
    for(int i=0;i<m;i++){
        c[i]=0;
    }
    for(int i=0;i<n;i++){
        c[x[i]=s[i]]++;
    }
    for(int i=1;i<m;i++){
        c[i]+=c[i-1];
    }
    //或者 ~i 表示 i!=-1
    for(int i=n-1;i>=0;i--){
```

```
    sa[--c[x[i]]]=i;
}
for(int k=1; k<=n; k<<=1){
    int p=0;
    for(int i=n-k;i<n;i++){
        y[p++]=i;
    }
    for(int i=0;i<n;i++){
        if(sa[i]>=k){
            y[p++]=sa[i]-k;
        }
    }
    for(int i=0;i<m;i++){
        c[i]=0;
    }
    for(int i=0;i<n;i++){
        c[x[y[i]]]++;
    }
    for(int i=1;i<m;i++){
        c[i]+=c[i-1];
    }
    for(int i=n-1;i>=0;i--){
        sa[--c[x[y[i]]]]=y[i];
    }
    swap(x, y);
    p=1;
    x[sa[0]]=0;
    for(int i=1;i<n;i++){
        x[sa[i]]=y[sa[i-1]]==y[sa[i]] && y[sa[i-1]+k]==y[sa[i]+k]? p-1 : p++;
    }
    if (p>=n){
        break;
    }
    m = p;
}
//去掉 $
n--;
for(int i = 0; i <= n; i++){
    rk[sa[i]] = i;
}
//计算 h
int k=0;
for(int i = 0; i < n; i++){
    if(k){
        k--;
    }
    int j = sa[rk[i] - 1];
    while(s[i + k] == s[j + k]){
        k++;
```

```
        }
        h[rk[i]] = k;
    }
}
//判断是否存在长度为 mid 的子串出现次数大于等于 k
bool check(int mid,int k){
    int cnt=1;
    for(int i=1;i<=n;i++){
        if(h[i]>=mid){
            cnt++;
        }else{
            cnt=1;
        }
        if(cnt>=k){
            return true;
        }
    }
    return false;
}
bool check2(int mid){
    int low=INF,hight=-INF;
    bool flag=false;
    for(int i=2;i<=n;i++){
        if(h[i]<mid){
            low=INF;
            hight=-INF;
        }else{
            low=min(low,min(sa[i],sa[i-1]));
            hight=max(hight,max(sa[i],sa[i-1]));
            if(hight-low>=mid){
                flag=true;
                break;
            }
        }
    }
    return flag;
}
/*
 * 预处理 O(nlogn)
 * dp[i][j]: 从 a[i] 开始 2^j 个数的最小值
 */
void RMQ_init(int n){
    for(int i=0;i<n;i++){
        dp[i][0]=h[i];
    }
    for(int j=1;(1<<j)<=n;j++){
            for(int i=0;i+(1<<j)-1<n;i++){
                //两段重叠部分小区间
                dp[i][j]=min(dp[i][j-1],dp[i+(1<<(j-1))][j-1]);
```

```
            }
        }
    }
/*
 * 查询 [l,r] 最小值，最大值同理
 */
int RMQ(int l,int r){
    int k=0;
    //保证刚好 [l,l+2^k] 和 [r-2^k,r] 重叠
    while((1<<(k+1))<=r-l+1){
        k++;
    }
    return min(dp[l][k],dp[r-(1<<k)+1][k]);
}

//============ 后缀数组应用总结 ============
//==== 任意两个后缀的最长公共前缀长度 ========
//转化为求两个后缀对应排名区间的最小 h 值，即 RMQ 问题
/*
 * 要先初始化 RMQ(n+1)，求回文串时 RMQ(2*n+2)
 * 分清查询的是两个后缀首字符还是两个后缀排名
 * l,r 为两个后缀的排名
 */
int solve0(int l,int r){
    if(l==r){
        return n-sa[l];
    }
    if(l>r){
        swap(l,r);
    }
    //这里可灵活处理，有时候不需要 +1
    return RMQ(l+1,r);
}
/*
 * i,j 为两个后缀的首字符下标
 */
int solve1(int i,int j){
    if(i==j){
        return n-i;
    }
    int ri=rk[i];
    int rj=rk[j];
    if(ri>rj){
        swap(ri,rj);
    }
    //注意 h 数组是类似差分的表示，区间 n 其实只有 n-1 的 h 数组
    return RMQ(ri+1,rj);
}
//==== 可重叠最长重复子串长度 ===============
```

```
//重复子串=最长公共前缀=区间内 min(h)
//最长重复子串=所有区间 max(min(h))=max(h)
int solve2(){
    int ans=0;
    //注意从 1 开始, 排名为 0 的是无效后缀'$'
    for(int i=1;i<=n;i++){
        ans=max(ans,h[i]);
    }
    return ans;
}
//==== 不可重叠最长重复子串 ================
//先二分答案 (子串长度) 将问题转化为判定性问题
//即判断字符串里是否存在长度为 mid 的重复子串
//将后缀按排名顺序进行分组, 每组后缀间 h 值 >=mid
//否则单独一个组
//同一组中判断 max(sa[i]) 和 min(sa[i])
//即最左后缀和最右后缀的差值是否 >=mid(即不重叠)
int solve3(){
    int ans=0;
    int l=0,r=n/2;
    //分组和判断同时进行
    //mx mn 分别维护组内最右和最左后端
    int mx=-INF,mn=INF;
    while(l<=r){
        int mid=(l+r)/2;
        bool flag=false;
        //可以直接从 2 开始, 因为 1 和 0 的 h 值在这里无意义
        for(int i=2;i<=n;i++){
            if(h[i]<mid){
                //新的分组
                mx=-INF;
                mn=INF;
            }else{
                //h[i] 表示的就是 sa[i] 和 sa[i-1] 这两个后缀的 LCP
                //所以这两个位置都有可能
                mx=max(mx,max(sa[i],sa[i-1]));
                mn=min(mn,min(sa[i],sa[i-1]));
                //poj1743 求的是差分数组所以这里是 >
                if(mx-mn>=mid){
                    flag=true;
                    break;
                }
            }
        }
        if(flag){
            ans=mid;
            l=mid+1;
        }else{
            r=mid-1;
```

```
        }
    }
    return ans;
}
//==== 可重叠出现至少 k 次的最长重复子串 ========
//同样是二分答案，后缀分组再进行判断
//不过这里判断的是组内后缀个数是否 >=k
int solve4(int k){
    int ans=0;
    int l=0,r=n/2;
    while(l<=r){
        int mid=(l+r)/2;
        int cnt=1;
        bool flag=false;
        for(int i=2;i<=n;i++){
            if(h[i]<mid){
                cnt=1;
            }else{
                cnt++;
                if(cnt>=k){
                    flag=true;
                    break;
                }
            }
        }
        if(flag){
            ans=mid;
            l=mid+1;
        }else{
            r=mid-1;
        }
    }
    return ans;
}
//==== 不同子串个数 ==================
//按排名枚举后缀，对于排名为 i 的后缀来说，贡献为 n-sa[i]+h[i]
//即后缀长度（从后缀第一个字符到后缀第 i 个字符算一个子串）减去重复部分
int solve5(){
    int ans=0;
    for(int i=1;i<=n;i++){
        ans+=(n-sa[i]-h[i]);
    }
    return ans;
}
//==== 最长回文子串 ==================
//将字符串反转接在原串后面（中间加入特殊分隔符），将问题转化为求 LCP
//枚举每一个位置分别求出奇数回文串和偶数回文串长度
int solve6(){
    //要先把字符串反转拼接再求 sa
```

```
    int ans=0;
    for(int i=0;i<n;i++){
        //奇数回文串
        ans=max(ans,2*solve1(i,2*n-i)-1);
        //偶数回文串
        ans=max(ans,2*solve1(i,2*n-i+1));
    }
    return ans;
}
//======= 出现次数 [a,b]/k 次 (即 [k,k]) 的子串个数 =====
//设 cal(k) 为出现次数大于等于 k 的子串个数
//问题转化为求 cal(a)-cal(b+1)
int cal(int k){
    //特判 k
    if(k==1){
        //即所有不同子串个数
        return solve5();
    }
    int ans=0;
    //不断枚举每一段 k-1 的 h 区间 (其实就是 k 个后缀)
    //0 没有 h 值, 1 的 h 值恒为 0
    int l=2,r=k;
    int pre=0;
    while(r<=n){
        //这里直接使用 RMQ
        int now=RMQ(l,r);
        if(now>=pre){
            ans+=now-pre;
        }
        pre=now;
        l++;
        r++;
    }
    return ans;
}
int solve7(int a,int b){
    return cal(a)-cal(b+1);
}
int main(void){
    scanf("%s",s);
    n=strlen(s);
    s[n]='$';
    for(int i=0;i<n;i++){
        s[i+n+1]=s[n-i-1];
    }
    build_sa(n*2+1,250);
    RMQ_init(n*2+2);
    printf("%s\n",s);
    for(int i=0;i<=n*2+1;i++){
```

```
        printf("%d ",sa[i]);
    }
    printf("\n");
    for(int i=0;i<=n*2+1;i++){
        printf("%d ",rk[i]);
    }
    printf("\n");
    for(int i=0;i<=n*2+1;i++){
        printf("%d ",h[i]);
    }
    printf("\n");
    // printf("%d\n",check(3,2));
    // printf("%d\n",check2(3));
    // int i,j;
    // RMQ_init(n+1);
    // while(~scanf("%d%d",&i,&j)){
    //     printf("%d\n",solve1(i,j));
    // }
    // int k;
    // while(~scanf("%d",&k)){
    //     printf("%d\n",solve4(k));
    // }
    printf("%d\n",solve6());
    return 0;
}
```

1.6 字符串哈希

```
typedef unsigned long long ull;
const ull seed1=146527;
const ull seed2=19260817;
const ull MOD1=1000000009;
const ull MOD2=998244353;
/*
 * 字符串哈希 (双哈希)
 * 记得初始化 (init())
 */
void init(){
    p1[0]=p2[0]=1;
    for(int i=1;i<N;i++){
        p1[i]=p1[i-1]*seed1%MOD1;
        p2[i]=p2[i-1]*seed2%MOD2;
    }
}
void Hash(char s[],ull *h1,ull *h2){
    int n=strlen(s);
    h1[0]=h2[0]=0;
    for(int i=1;i<=n;i++){
        h1[i]=(h1[i-1]*seed1%MOD1+s[i-1])%MOD1;
        h2[i]=(h2[i-1]*seed2%MOD2+s[i-1])%MOD2;
```

```
    }
}
pair<ull,ull> substr(int l,int len,ull *h1,ull *h2){
    int r=l+len;
    pair<ull,ull> res;
    res.first=(h1[r]-h1[l]*p1[r-l]%MOD1+MOD1)%MOD1;
    res.second=(h2[r]-h2[l]*p2[r-l]%MOD2+MOD2)%MOD2;
}
```

1.7 其他

1.7.1 最大/小表示法

```
/*
 * 求循环字符串 s 的最小/最大表示
 * i,j: 当前比较两个字符串的起始位置
 * k: 这两个字符串已比较的长度
 */
int getMin(char s[]){
    int n=strlen(s);
    int i=0,j=1,k=0;
    while(i<n && j<n && k<n){
        int t=s[(i+k)%n]-s[(j+k)%n];
        if(!t){
            k++;
        }else{
            if(t>0){
                //如果是求最大表示则为 j+=k+1
                i+=k+1;
            }else{
                 //同理则为 i+=k+1
                j+=k+1;
            }
            if(i==j){
                j++;
            }
            k=0;
        }
    }
    return min(i,j);
}
```

2 树图

2.1 拓扑排序

```
/*
 * 拓扑排序 (bfs)
 * to[i]: 拓扑排序第 i 个节点
 * flag: 是否存在环
```

```
 */
void topo(){
    queue<int> q;
    int k=0;
    for(int i=1;i<=n;i++){
        if(ind[i]==0){
            q.push(i);
            topo[k++]=u;
        }
    }
    int cnt=0;
    while(!q.empty()){
        int u=q.front();
        q.pop();
        cnt++;
        for(int i=head[u];i!=-1;i=edge[i].next){
            int v=edge[i].v;
            ind[v]--;
            if(!ind[v]){
                q.push(v);
                topo[k++]=v;
            }
        }
    }
    if(cnt!=n){
        //有环
        flag=true;
    }
}
```

2.2 并查集

2.2.1 普通并查集

```
/*
 * 基础并查集 (注意初始化 p[i]=i)
 * p[i]: i 的根
 */
int find(int x){
    //递归写法
    return x==p[x]?p[x]:p[x]==find(p[x]);
}
//=========================//
//有时候需要写成这样
int find(int x){
    if(x!=p[x]){
        int fa=p[x];
        p[x]=find(p[x]);
        //这里就可以对临时保存的 fa 进行操作，特别在带权并查集中
        //也可以直接 p[x]=find(p[x]), 然后对修改后的 p[x] 进行操作
```

```
        v[x]+=v[fa];
    }
    return p[x];
}
int find(int x){
    //有时需要维护多个信息（带权并查集），需要非递归写法
    int fa=x;
    //找到 x 的根
    while(fa!=p[fa]){
        fa=p[fa];
    }
    //从 x 重新一步一步往上，沿途更新
    while(x!=p[x]){
        x=p[x];
        p[x]=fa;
    }
    return p[x];
}
int unit(int a,int b){
    //最基础的合并
    int fa=find(a);
    int fb=find(b);
    if(fa!=fb){
        p[fa]=fb;
    }
}
```

2.2.2 带权并查集

```
/*
 * 带权并查集
 * 所谓带权并查集就是在并查集的基础上除了维护根节点再维护一些其他的值
 * 例如集合大小，元素移动次数，元素到根节点的距离等
 * 状态的转移在 find 和 unit 都要考虑，而且要注意顺序
 * eg. poj1182——食物链 设 r[i] 表示 i 和父节点（根）的关系
 * r[i]=0 表示和父节点同类，1 表示被父节点吃，2 表示吃父节点
 */
int find(int x){
    if(x!=p[x]){
        //暂存当前根节点，用于状态转移更新信息
        int t=p[x];
        //递归压缩路径，此时 p[x] 信息已更新
        p[x]=find(p[x]);
        //元素移动次数/元素到根距离
        cnt[x]+=cnt[t];
        //假设 r[x]=1，即 x 被 t 吃，如果 r[t]=2，即 t 吃 p[x]
        //那说明 x 和 p[x] 同类，r[x]=(1+2)%3=0，其他同理
        r[x]=(r[x]+r[t])%3;
    }
    return p[x];
```

```
}
void unit(int a,int b){
    int fa=find(a);
    int fb=find(b);
    if(fa!=fb){
        p[fa]=fb;
        //hdu2818 将一堆放到另一堆上 维护某一元素下方元素个数，即到根的距离
        cnt[fa]=siz[fb];
        //集合大小
        siz[fb]+=siz[fa];
    }
}
bool unit(int a,int b,int q){
    q--;
    //q 是操作，0 表示同类，1 表示 a 吃 b
    int fa=find(a);
    int fb=find(b);
    if(fa!=fb){
        p[fb]=fa;
        //可以画图去看 r[fb] 即求 fb 到 fa 的关系
        //必须通过 fb->b(-r[b]) b->a(q) a->fa(r[a]) 三个向量相加即可
        r[fb]=(-r[b]+r[a]+q+3)%3;
        return false;
    }else{
        //已在同一集合
        //同样通过画图，此时 a,b 有相同根 f
        //r[a]+q 即 b->a(q) a->f(r[a])==r[b]，若不符合，即矛盾
        if((r[a]+q)%3!=r[b]){
            //矛盾
            return true;
        }else{

        }
    }
}
```

2.3 最小生成树

2.3.1 Prim

```
/*
 * 最小生成树 (MST)Prim 算法，使用邻接矩阵
 * 顶点 0~n-1
 * dis[i]: 已选点集到 i 的最短边
 */
int Prim(){
    for(int i=0;i<n;i++){
        dis[i]=cost[0][i];
        vis[i]=false;
    }
```

```
    vis[0]=true;
    int ans=0;
    for(int i=1;i<n;i++){
        int k=-1;
        int mc=INF;
        for(int j=0;j<n;j++){
            if(!vis[j] && dis[j]<mc){
                mc=dis[j];
                k=j;
            }
        }
        if(k==-1){
            break;
        }
        vis[k]=true;
        ans+=mc;
        for(int j=0;j<n;j++){
            if(!vis[j] && dis[j]>cost[k][j]){
                dis[j]=cost[k][j];
            }
        }
    }
    return ans;
}
```

2.4 次小生成树

```
/*
 * 最小生成树 Prim 算法
 * used[i][j]: i-j 这条边是否在 MST 中
 * path[i][j]: <i...j> 的路径上的最长边
 * pre[i]: MST 中 i 的前驱节点，即上一步能够更新 low[i] 的点，即将 i 加入 MST 的点
 */
int Prim(){
    for(int i=0;i<n;i++){
        vis[i]=false;
        low[i]=cost[0][i];
        pre[i]=0;
        for(int j=0;j<n;j++){
            used[i][j]=false;
            path[i][j]=false;
        }
    }
    vis[0]=true;
    pre[0]=-1;
    low[0]=0;
    int ans=0;
    for(int i=1;i<n;i++){
        int k=-1;
        int Min=INF;
```

```
        for(int j=0;j<n;j++){
            if(!vis[j] && low[j]<Min){
                Min=low[j];
                k=j;
            }
        }
        if(k==-1){
            break;
        }
        ans+=Min;
        vis[k]=true;
        used[k][pre[k]]=used[pre[k]][k]=true;
        for(int j=0;j<n;j++){
            //因为 j 访问过，也就是在 MST 中，即此时可以看成是 j--pre[k]--k
            //j 是 MST 点集的一点，而 k 就刚好被选中要加入这个点集
            //pre[k] 是 k 的前驱，也就是点集的边缘，也就是把 k 带进 MST 的点
            //所以 j 到 k 的最大边就是由两部分更新而来，path[j][pre[k]] 和 low[k]
            if(vis[j] && j!=k){
                path[j][k]=path[k][j]=max(path[j][pre[k]],low[k]);
            }else if(!vis[j] && low[j]>cost[k][j]){
                low[j]=cost[k][j];
                pre[j]=k;
            }
        }
    }
    return ans;
}
/*
 * 先求出 MST，再枚举不在 MST 中的边求出 SST
 */
int solve(){
    int ans=Prim();
    for(int i=0;i<n;i++){
        for(int j=i+1;j<n;j++){
            if(!used[i][j]){
                ans=min(ans,ans-path[i][j]+cost[i][j]);
            }
        }
    }
    return ans;
}
```

2.5 tarjan

2.5.1 强连通分量

```
/*
 * 求有向图的强连通分量
 * low[u]:u 能回溯到的最早祖先
 * dfn[u]:dfs 序
```

```
 * scc[u]:u 所属 scc 编号
 * inSta[u]:u 是否在栈中
 * s: 暂存节点
 */
void dfs(int u){
    //初始化 dfs 序
    low[u]=dfn[u]=++idx;
    s.push(u);
    inSta[u]=true;
    int siz=g[u].size();
    for(int i=0;i<siz;i++){
        int v=g[u][i];
        if(dfn[v]==-1){
            //未访问
            dfs(v);
            //此时已经递归到底，可以用 low[v] 来更新 low[u]
            low[u]=min(low[u],low[v]);
        }else if(inSta[v]){
            //访问过，在栈中 (low[v] 还没完全更新)
            low[u]=min(low[u],dfn[v]);
        }
    }
    if(low[u]==dfn[u]){
        //能回到的最早节点就是本身，说明得到一个强连通分量
        cnt++;
        while(!s.empty()){
            int t=s.top();
            scc[t]=cnt;
            s.pop();
            inSta[t]=false;
            if(t==u){
                break;
            }
        }
    }
}
void tarjan(int n){
    memset(dfn,-1,sizeof(dfn));
    memset(scc,-1,sizeof(scc));
    while(!s.empty()){
        s.pop();
    }
    idx=cnt=0;
    for(int i=1;i<=n;i++){
        //未访问
        if(dfn[i]==-1){
            dfs(i);
        }
    }
```

```
}
```

2.5.2 边-双连通分量

```
/*
 * tarjan 算法求桥/边-双连通分量
 * 双连通分量即删除桥后的各个连通分量，可以说 BCC=SCC+ 桥的判定
 * 小结论：一个有桥的连通图要变为边双连通图至少需要加几条边？
 *     双连通分量缩点建树，求叶子节点数量，答案为 (leaf+1)/2
 */
void dfs(int u,int f){
    low[u]=dfn[u]=++idx;
    s.push(u);
    inStack[u]=true;
    for(int i=head[u];i!=-1;i=edge[i].next){
        int v=edge[i].v;
        //无向图
        if(v==f){
            continue;
        }
        if(dfn[v]==-1){
            dfs(v,u);
            low[u]=min(low[u],low[v]);
            //桥的判定
            if(low[v]>dfn[u]){
                bridegs.push_back(make_pair(u,v));
            }
        }else if(inStack[v]){
            low[u]=min(low[u],dfn[v]);
        }
    }
    //删掉桥剩下的就是边-双联通分类
    if(low[u]==dfn[u]){
        num++;
        while(!s.empty()){
            int t=s.top();
            s.pop();
            inStack[t]=false;
            bcc[t]=num;
            if(t==u){
                break;
            }
       }
    }
}
void solve(int n){
    num=0;
    memset(dfn, -1, sizeof(dfn));
    memset(low, -1, sizeof(low));
    for(int i=1;i<=n;i++){
```

```
        if(dfn[i]==-1){
            dfs(i,0);
        }
    }
}
```

2.5.3 点-双连通分量

```
/*
 * tarjan 算法求割点/点-双连通分量
 * 边/点双连通分量：不存在桥/割点的极大子图
 * 一个割点属于多个点双连通分量，一个桥不属于任何边双连通分量
 * bcc[i]: 表示编号为 i 的连通分量的节点 (vector 数组)
 * rt: 根，而不是父节点
 */
void dfs(int u,int rt){
    low[u]=dfn[u]=++idx;
    //记录儿子节点个数，用于根的特判
    int son=0;
    s.push(u);
    inStack[u]=true;
    for(int i=head[u];i!=-1;i=edge[i].next){
        int v=edge[i].v;
        if(dfn[v]==-1){
            dfs(v,rt);
            low[u]=min(low[u],low[v]);
            //割点的判定
            if(low[v]>=dfn[u]){
                num++;
                while(!s.empty()){
                    int t=s.top();
                    s.pop();
                    inStack[t]=false;
                    bcc[num].push_back(t);
                    if(t==v){
                        break;
                    }
                }
                //点双连通分量点可重复
                bcc[num].push_back(u);
                //根的特判
                if(u==rt){
                    son++;
                }else{
                    cut[u]=true;
                }
            }
        }else if(inStack[v]){
            low[u]=min(low[u],dfn[v]);
        }
```

```
    }
    //根节点特判，只要多于两个儿子即为割点
    if(son>=2 && u==rt){
        cut[u]=true;
    }
}
void solve(){
    memset(dfn,-1,sizeof(dfn));
    memset(low,-1,sizeof(low));
    for(int i=1;i<=n;i++){
        if(dfn[i]==-1){
            //注意调用方式
                dfs(i,i);
        }
    }
}
```

2.6 LCA

2.6.1 ST 表在线

```
/*
 * ver[i]: 第 i 个访问节点的编号
 * fir[i]: 编号为 i 的节点第一次出现的时间
 * dep[i]: 第 i 个访问节点的深度
 * ver,dep 注意开两倍空间
 */
void dfs(int u,int f,int d){
    tot++;
    ver[tot]=u;
    fir[u]=tot;
    dep[tot]=d;
    for(int i=head[u];i!=-1;i=edge[i].next){
        int v=edge[i].v;
        if(v==f){
            continue;
        }
        dfs(v,u,d+1);
        //回溯再记录一次
        tot++;
        ver[tot]=u;
        dep[tot]=d;
    }
}
/*
 * RMQ 初始化
 * dp[i][j]: 从 dep[i] 起 2^j 个数的编号最小值 (ver 和 dep 的编号即访问时间)
 * dp 数组注意开两倍空间
 */
```

```
void RMQ_init(int n){
    for(int i=0;i<n;i++){
        dp[i][0]=i;
    }
    for(int j=1;(1<<j)<=n;j++){
        for(int i=0;i+(1<<j)-1<n;i++){
            //和普通的 rmq 不同，这里比较的是深度，但 dp 保存的是位置
            int a=dp[i][j-1];
            int b=dp[i+(1<<(j-1))][j-1];
            dp[i][j]=(dep[a]<dep[b]?a:b);
        }
    }
}
int RMQ(int l,int r){
    int k=0;
    while(1<<(k+1)<=r-l+1){
        k++;
    }
    int a=dp[l][k];
    int b=dp[r-(1<<k)+1][k];
    return (dep[a]<dep[b]?a:b);
}
/*
 * 找出访问 u 和 v 之间 (fir[u]...fir[v]) 深度最小的点的编号 idx，由 ver 映射成节点编号
 */
int LCA(int u,int v){
    int x=fir[u];
    int y=fir[v];
    if(x>y){
        swap(x,y);
    }
    int idx=RMQ(x,y);
    return ver[idx];
}
```

2.6.2 Tarjan 离线

```
/*
 * Tarjan 算法 + 并查集求 LCA(离线)
 * vis[u]: 访问标记
 * p[u]:u 的根/父节点
 * que[u]: 跟 u 有联系的查询链表
 */
void LCA(int u,int f){
    vis[u]=true;
    int siz=g[u].size();
    for(int i=0;i<siz;i++){
        int v=g[u][i].v;
        if(v==f){
            continue;
```

```
        }
        LCA(v,u);
        //回溯之后更新父节点
        p[v]=u;
    }
    int qsiz=que[u].size();
    for(int i=0;i<qsiz;i++){
        // 遍历跟 u 有查询关系的点
        int v=que[u][i].q;
        int id=que[u][i].idx;
        if(vis[v]){
            //如果已经访问过则 lca 为并查集中 v 的根
            lca[id]=Find(v);
        }
    }
}
```

2.7 最短路

2.8 网络流

2.8.1 Dinic

```
/*
 * s,t: 源点, 汇点
 * dep[i]: i 点的深度
 * cur[i]: i 的邻接链表当前访问到的边
 */
//=====================
/*
 * bfs 构建分层图
 */
bool bfs(){
    queue<int> q;
    memset(dep,0,sizeof(dep));
    dep[s]=1;
    q.push(s);
    while(!q.empty()){
        int u=q.front();
        q.pop();
        for(int i=head[u];i!=-1;i=edge[i].next){
            int v=edge[i].v;
            int w=edge[i].w;
            //可用流量大于 0 && 未标记深度
            if(w>0 && dep[v]==0){
                dep[v]=dep[u]+1;
                q.push(v);
            }
        }
    }
    return dep[t]!=0;
```

```
}
/*
 * dfs 增广
 */
int dfs(int u,int flow){
    if(u==t){
        return flow;
    }
    //当前弧优化
    for(int &i=cur[u];i!=-1;i=edge[i].next){
        int v=edge[i].v;
        int w=edge[i].w;
        if(dep[v]==dep[u]+1 && w!=0){
            //dfs 向下增广
            int dis=dfs(v,min(w,flow));
            if(dis>0){
                //正向边和反向边更新可用流量
                edge[i].w-=dis;
                edge[i^1].w+=dis;
                return dis;
            }
        }
    }
    return 0;
}
/*
 * dinic 算法
 */
int dinic(int n){
    int ans=0;
    while(bfs()){
        //不断构建分层图再增广，加上当前弧优化
        for(int i=1;i<=n;i++){
            cur[i]=head[i];
        }
        while(int d=dfs(s,INF)){
            ans+=d;
        }
    }
    return ans;
}
```

2.8.2 上下界网络流

```
/*
上下界网络流主要是建图问题
0. 假设流量上下界为 [l,r]
1. 无源汇可行流（循环流）
    添加超级源汇 ss,tt，对于每一条弧 (u,v)，拆成 (u,v,(r-l)),(u,tt,l),(ss,v,l)
    跑一遍 ss 到 tt 的最大流，若附加边 (u,tt),(ss,v) 都满流，说明存在可行流
```

```
2. 有源汇可行流
    建立弧 (t,s,(INF)), 转化为无源汇可行流, 按无源汇可行流的方法建图跑一遍最大流即可,
    而且此时 (t,s) 的流量就是原图的总流量
3. 有源汇最大流
    按照有源汇可行流的方法建图, 如果存在可行流, 不用清空流量, 再跑一遍从 s 到 t 的最大流
4. 有源汇最小流
    按照有源汇可行流建图, 但不要建立弧 (t,s), 跑一遍 ss 到 tt 的最大流, 再建立弧 (t,s,(INF)),
    再跑一遍 ss 到 tt 的最大流
*/
```

3 动态规划

3.1 子序列/子串

3.1.1 最大连续子序列和

```
/*
 * 求最大连续子序列和
 * 定义 dp[i] 表示前 i 个值的最大连续和, dp[i]=max(dp[i-1]+a[i],a[i])
 */
void maxSum(int a[],int n){
    //根据题目具体要求赋初值
    int ans=-2147483647;
    int cur=0;
    for(int i=0;i<n;i++){
        cur=max(cur+a[i],a[i]);
        ans=max(ans,cur);
    }
    return ans;
}
```

3.1.2 最大上升/下降子序列

```
/*
 * O(n^2) 求最大上升/下降子序列 (不连续)
 * d[i]: 到 i 位置的最长上升子序列长度
 */
void solve(){
    for(int i=0;i<n;i++){
        for(int j=0;j<i;j++){
            if(a[j]<a[i]){
                d[i]=max(d[i],d[j]+1);
            }
        }
    }
}
/*
 * O(nlogn) 算法
 * 也许不是 dp...
 */
```

```
void solve2(){
    for(int i=1;i<=n;i++){
        g[i]=INF;
    }
    //求下降子序列将原数组取反即可
    //g[i] 表示 LIS 为 i 的子序列的最后一个数的最小值
    for(int i=0;i<n;i++){
        //找到 g[k] 刚好大于 a[i]
        //改成 upper_bound() 可以求最大不下降序列
        int k=lower_bound(g+1,g+1+n,a[i])-g;
        d[i]=k;
        //更新 g[k]
        g[k]=a[i];
    }
}
```

3.2 背包

3.3 数位 dp

```
/*
 * 数位 dp，核心就是定义状态，并从高位枚举，从前一位 (pre**) 的状态转移而来
 * a[i]: 第 i 个数位
 * dp[i][j][k][l]:
 *     从最高位到第 i 位，前一个数字是 j，前面数字模为 k，前面是/否已经出现 13 的所有数的个数
 */
//pos 当前位 mod 前面数字模 13 余数 pre 前一位数字 sta 前面是否有 13
ll dfs(int pos,int mod,int pre,int sta,bool limit){
    //递归边界，枚举完所有位数
    if(pos<0){
        return (sta==1 && mod==0);
    }
    //记忆化搜索，必须是没有上限的情况下
    if(!limit && dp[pos][pre][mod][sta]!=-1){
        return dp[pos][pre][mod][sta];
    }
    //上限
    int up=limit?a[pos]:9;
    ll ans=0;
    for(int i=0;i<=up;i++){
        //枚举当前位数字
        ans+=dfs(pos-1,(mod*10+i)%13,i,sta||(pre==1 && i==3),limit&&(i==up));
    }
    //dp 数组定义，必须是所有数
    if(!limit){
        dp[pos][pre][mod][sta]=ans;
    }
    return ans;
}
ll solve(ll x){
```

```
    //如果有多次询问，memset 一次 dp 即可
    int k=0;
    while(x){
        a[k++]=x%10;
        x/=10;
    }
    return dfs(k-1,0,0,false,true);
}
```

3.4 区间 dp

3.4.1 石头合并问题

```
/*
 * 经典石头归并问题 (2 合 1)
 * dp[i][j]: 区间 [i..j] 合并成 1 堆的最小代价
 */
int solve(){
    //初始化
    memset(dp,INF,sizeof(dp));
    for(int i=1;i<=n;i++){
        dp[i][i]=0;
    }
    //最外层一定是枚举区间长度，从小区间更新到大区间
    for(int len=2;len<=n;len++){
        //枚举左端点，计算右端点
        for(int l=1;l+len-1<=n;l++){
            int r=l+len-1;
            //枚举区间分割点
            for(int m=l;m<r;m++){
                //状态转移
                dp[l][r]=min(dp[l][r],dp[l][m]+dp[m+1][r]+pre[r]-pre[l-1]);
            }
        }
    }
    return dp[1][n];
}
/*
 * 拓展到 k 合 1 的情况 (Leetcode1000)
 * dp2[i][j][k]: 区间 [i..j] 合并成 k 堆的最小代价
 */
int ksolve(int K){
    memset(dp2,INF,sizeof(dp2));
    for(int i=1;i<=n;i++){
        dp2[i][i][1]=0;
    }
    for(int len=2;len<=n;len++){
        for(int l=1;l+len-1<=n;l++){
            int r=l+len-1;
            for(int m=l;m<r;m++){
```

```
                for(int k=2;k<=len;k++){
                    dp2[l][r][k]=min(dp2[l][r][k],dp2[l][m][k-1]+dp2[m+1][r][1]);
                }
            }
            dp2[l][r][1]=dp2[l][r][K]+pre[r]-pre[l-1];
        }
    }
    return dp2[1][n][1];
}
```

3.5 其他

3.5.1 编辑距离

```
/*
 * 编辑距离/莱温斯坦距离
 * 两个字符串，三种操作 (替换/插入/删除一个字符)，求使两个字符串相同的最小操作数
 * dp[i][j]: 表示 str1 前 i 个字符和 str2 前 j 个字符的最小编辑距离
 * dp[i][0]=dp[0][i]=i;
 * dp[i][j]=dp[i-1][j-1]                                    (str1[i-1]==str2[j-1])
 *         =min(dp[i-1][j],dp[i][j-1],dp[i-1][j-1])+1       (otherwise)
 */
void LD(char str1[],char str2[]){
    int n=strlen(str1);
    int m=strlen(str2);
    for(int i=0;i<=n;i++){
        dp[i][0]=i;
    }
    for(int i=0;i<=m;i++){
        dp[0][i]=i;
    }
    for(int i=1;i<=n;i++){
        for(int j=1;j<=m;j++){
            if(str1[i-1]==str2[j-1]){
                dp[i][j]=dp[i-1][j-1];
            }else{
                dp[i][j]=min(min(dp[i-1][j],dp[i][j-1]),dp[i-1][j-1])+1;
            }
        }
    }
}
```

4 基础数论

4.1 快速幂取模

```
/*
 * 快速幂取模：求 a^n%mod
 */
ll PowMod(ll a,ll n,ll mod){
```

```
    ll ans=1;
    a%=mod;
    while(n){
        if(n%2){
            ans=(ans*a)%mod;
        }
        a=a*a%mod;
        n/=2;
    }
    return ans;
}
```

4.2 欧拉函数

```
/*
 * 线性筛同时筛出素数和欧拉函数
 * check[i]: i 是否是合数 (被筛)
 * p[i]: 第 i 个素数, p[0] 为个数
 * phi[i]: i 的欧拉函数值
 */
void init(){
    check[1]=true;
    phi[1]=1;
    for(int i=2;i<=N;i++){
        if(!check[i]){
            p[++p[0]]=i;
            phi[i]=i-1;
        }
        for(int j=1;j<=p[0];j++){
            int t=i*p[j];
            if(t>N){
                break;
            }
            check[t]=true;
            //t 拥有多个相同质因子 (p[j] 至少就 2 次)
            if(i%p[j]==0){
                //i 是 p[j] 的倍数, 那 t 和 i 的质因子相同, 由欧拉函数计算式可得两者只差一个系数
                phi[t]=phi[i]*p[j];
            }else{
                //欧拉函数是积性函数 phi[t]=phi[i]*phi[p[j]]
                phi[t]=phi[i]*(p[j]-1);
            }
        }
    }
}
/*
 * 求单个欧拉函数 O(sqrt(n))
 */
int getPhi(int x){
    int res=x;
```

```
    for(int i=2;i*i<=x;i++){
        if(x%i==0){
            res=res-res/i;
            while(x%i==0){
                x/=i;
            }
        }
    }
    if(x!=1){
        res=res-res/x;
    }
    return res;
}
```

4.3 欧拉降幂

```
/*
 * 欧拉降幂：解决 A^B%C(B 特别大)
 * 1. 欧拉降幂：要求 A 和 C 互质，A^B%C=A^(B%phi(C))
 * 2. 广义欧拉降幂：A^B%C=A^(B%phi(C)+phi(C))
 */
ll solve(ll a,char *s,ll c){
    int n=strlen(s);
    ll b=0;
    ll mod=getPhi(c);
    for(int i=0;i<n;i++){
        b=(b+s[i]-'0')%mod;
    }
    b=(b+mod)%mod;
    return PowMod(a,b,c);
}
```

4.4 拓展欧几里得 (EXGCD)

```
#define Mod(a,b) ((a)%(b)+(b))%(b)
/*
 * exgcd: 给定 ax+by=gcd(a,b) 已知 a,b 求 x,y 且 |x|+|y| 最小
 */
ll exgcd(ll a,ll b,ll &x,ll &y){
    if(b==0){
        x=1;
        y=0;
        return a;
    }
    ll d=exgcd(b,a%b,x,y);
    ll t=x;
    x=y;
    y=t-a/b*y;
    return d;
}
```

```
/*
 * 解 ax+by=c: 求非负整数解
 * x,y: 最小非负整数解
 * dx,dy: X=x+k*dx Y=y-k*dy 为方程通解
 * 无解返回 false
 */
bool solve(ll a,ll b,ll c,ll &x,ll &y,ll &dx,ll &dy){
    //无非负整数解
    if(a==0 && b==0){
        return false;
    }
    ll x0,y0;
    ll d=exgcd(a,b,x0,y0);
    //c 不整除 gcd(a,b), 无解
    if(c%d!=0){
        return false;
    }
    //注意 a 和 b 要反过来
    dx=b/d;
    dy=a/d;
    //取最小正数解
    x=Mod(x0*c/d,dx);
    y=(c-a*x)/b;
    return true;
}
```

4.5 中国剩余定理 (CRT)

4.6 逆元

```
/*
 * exgcd 求逆元 当且仅当 gcd(a,m)=1 时才存在逆元
 * 原理: ax=1(mod m) => ax+km=1 (mod m)
 */
ll inv(ll a,ll m){
    ll x,y;
    ll d=exgcd(a,m,x,y);
    return d==1?(x+m)%m:-1;
}
/*
 * 费马小定理求逆元 当 a<p 且 p 为素数
 */
ll inv(ll a,ll p){
    return PowMod(a,p-2,p);
}
```

4.7 小技巧

4.7.1 求 n! 位数

```
/*
 * 使用斯特林公式求解 n 阶乘的位数
 */
int count(ll n){
    if(n==1){
        return 1;
    }
    return (int)ceil(0.5*log10(2*M_PI*n)+n*log10(n)-n*log10(M_E));
}
```

5 博弈

5.1 SG 函数

```
/*
 * SG 函数打表找规律
 * mex(S): 表示不属于集合 S 的最小非负整数
 * sg[x]=mex(s);    sg[x]=0 当且仅当 x 为必败态
 * f[i]: 表示每一步可走的情况 (步数/可取的石子数)
 */
void getSG(int n){
    memset(sg,0,sizeof(sg));
    for(int i=1;i<=n;i++){
        memset(s,0,sizeof(s));
        for(int j=1;j<N && f[j]<=i;j++){
            //打标记
            s[sg[i-f[j]]]=1;
        }
        for(int j=0;j<=n;j++){
            if(s[j]==0){
                sg[i]=j;
                break;
            }
        }
    }
}
/*
 * 初始化 f[]
 * 示例: Bash 博弈
 */
void init(){
    for(int i=1;i<N;i++){
        if(i<=m){
            f[i]=i;
        }else{
            f[i]=m;
        }
```

```
    }
}
```

5.2 Bash Game

```
/*
 * 有一堆 n 个石子，两人轮流取，每次取 1-m 个
 */
bool solve(int n,int m){
    return n%(m+1);
}
```

5.3 Wythoff Game

```
/*
 * 有两堆石子 (a,b)，两人轮流从一堆或者两堆中取出相同个数，每次最少取一个
 */
bool solve(int a,int b){
    if(a>b){
        swap(a,b);
    }
    int c=int((b-a)*(sqrt(5.0)+1)/2);
    //根据 (0,0) 先手必败态来判断
    return c!=a;
}
/*
 * SG 函数打表
 */
void getSG(int n){
    //枚举两堆分别的数量和取的数量
    for(int i=0;i<=n;i++){
        for(int j=0;j<=n;j++){
            //记录后继状态形式
            int win=0,lose=0;
            //同时从两堆中取 k 个
            for(int k=1;k<=min(i,j);k++){
                if(!sg[i-k][j-k]){
                    win++;
                }else{
                    lose++;
                }
            }
            for(int k=1;k<=i;k++){
                if(!sg[i-k][j]){
                    win++;
                }else{
                    lose++;
                }
            }
            for(int k=1;k<=j;k++){
```

```
                if(!sg[i][j-k]){
                    win++;
                }else{
                    lose++;
                }
            }
            //后继都是必败态，则当前状态是必胜态
            if(lose==0){
                sg[i][j]=1;
            }else{
                sg[i][j]=0;
            }
        }
    }
}
```

5.4 Nim Game

```
/*
 * 有三堆石子 (a,b,c)，两人轮流取，每次从一堆中取至少一个
 */
bool solve(int a,int b,int c){
    //可拓展到 n 堆
    //根据必败态 (0,0,0) 判断
    return a^b^c;
}
/*
 * SG 函数打表
 */
void getSG(int n){
    for(int i=0;i<=n;i++){
        for(int j=0;j<=n;j++){
            for(int g=0;g<=n;g++){
                int win=0,lose=0;
                for(int k=1;k<=i;k++){
                    if(!sg[i-k][j][g]){
                        win++;
                    }else{
                        lose++;
                    }
                }
                for(int k=1;k<=j;k++){
                    if(!sg[i][j-k][g]){
                        win++;
                    }else{
                        lose++;
                    }
                }
                for(int k=1;k<=g;k++){
                    if(!sg[i][j][g-k]){
```

```
                    win++;
                }else{
                    lose++;
                }
            }
            if(lose==0){
                sg[i][j][k]=1;
            }else{
                sg[i][j][k]=0;
            }
        }
    }
}
```

5.5 Fibonaci Game

```
/*
 * 有一堆 n 个石子，两人轮流取，一次可以取任意多个但不能取完，也不能超过上一个人取的两倍
 * 预处理出斐波那契数列，先手必败态当且仅当 n 为斐波那契数
 */
bool solve(int n){
    bool flag=false;
    for(int i=0;i<100;i++){
        if(n==fib[i]){
            return true;
        }
    }
    return false;
}
```

6 计算几何

7 区间问题

7.1 线段树

7.2 RMQ

```
/*
 * 预处理 O(nlogn)
 * dp[i][j]: 从 a[i] 开始 2^j 个数的最小值
 */
void RMQ_init(int n){
    for(int i=0;i<n;i++){
        dp[i][0]=a[i];
    }
    for(int j=1;(1<<j)<=n;j++){
        for(int i=0;i+(1<<j)-1<n;i++){
            //两段重叠部分小区间
```

```
            dp[i][j]=min(dp[i][j-1],dp[i+(1<<(j-1))][j-1]);
        }
    }
}
/*
 * 查询 [l,r] 最小值，最大值同理
 */
int RMQ(int l,int r){
    int k=0;
    //保证刚好 [l,l+2^k] 和 [r-2^k,r] 重叠
    while((1<<(k+1))<=r-l+1){
        k++;
    }
    return min(dp[l][k],dp[r-(1<<k)+1][k]);
}
```

7.3 树状数组

7.3.1 单点更新区间求和

```
/*
 * 树状数组 (普通版): 单点更新，区间求和 (sum(r)-sum(l-1))
 * c[i]: 树状数组，c[i]=a[i-lowbit(i)+1]+a[i-lowbit(i)+2]+...+a[i]
 */
int lowbit(int x){
    return x&(-x);
}
int add(int i,int x){
    while(i<=n){
        c[i]+=x;
        i+=lowbit(i);
    }
}
int sum(int i){
    int ans=0;
    while(i>0){
        //这里也可以维护最值 ans=max(ans,c[i]);
        ans+=c[i];
        i-=lowbit(i);
    }
    return ans;
}
```

7.3.2 求逆序数

```
/*
 * 离散化 + 树状数组求逆序数
 */
int solve(){
    for(int i=1;i<=n;i++){
        int k=lower_bound(b+1,b+n+1,a[i])-b;
```

```
        add(k,1);
        ans+=i-sum(k);
    }
    return ans;
}
```

7.3.3 区间更新单点查询

```
/*
 * 树状数组 (加强版 1): 区间更新, 单点查询 (sum(i))
 * 设 d[i]=a[i]-a[i-1] 所以 a[i]=d[1]+d[2]+...+d[i]
 * 树状数组 c 维护 d 的前缀和, 也相当于变相维护了 a[i], 所以单点查询是查询 sum
 */
void update(int l,int r){
    add(l,1);
    add(r+1,-1);
}
```

7.3.4 区间更新区间求和

```
/*
 * 树状数组 (加强版 2): 区间更新, 区间求和
 * 维护两个树状数组 c1 保存 d[i] 的前缀和 c2 保存 d[i]*i 的前缀和
 * 求和 ans=\sum[((k+1)*c1[i]-c2[i])]
 */
void add(int i,int x){
    int k=i;
    while(i<=n){
        c1[i]+=x;
        c2[i]+=k*x;
        i+=lowbit(i);
    }
}
void add_range(int l,int r,int x){
    add(l,x);
    add(r+1,-x);
}
int sum(int i){
    int ans=0;
    int k=i;
    while(i>0){
        ans+=((k+1)*c1[i]-c2[i]);
        i-=lowbit(i);
    }
    return ans;
}
int ask_range(int l,int r){
    return sum(r)-sum(l-1);
}
```

7.3.5 二维树状数组–单点更新区间求和

```
/*
 * 二维数组数组 (I), 单点更新, 区间求和 (sum(x2,y2)-sum(x1-1,y2)-sum(x2,y1-1)+sum(x1-1,y1-1))
 */
void add(int x,int y,int z){
    for(int i=x;i<=n;i+=lowbit(i)){
        for(int j=y;j<=m;j+=lowbit(j)){
            c[i][j]+=z;
        }
    }
}
int sum(int x,int y){
    int ans=0;
    for(int i=x;i>=1;i-=lowbit(i)){
        for(int j=y;j>=1;j-=lowbit(j)){
            ans+=c[i][j];
        }
    }
    return ans;
}
```

7.3.6 二维树状数组–区间更新单点查询

```
/*
 * 二维树状数组 (II): 区间更新, 单点查询 (sum(x,y))
 * 和一维的一样也是用树状数组维护一个差分数组
 */
void add_range(int x1,int y1,int x2,int y2,int w){
    add(x1,y1,w);
    add(x2+1,y2+1,w);
    add(x2+1,y1,-w);
    add(x1,y2+1,-w);
}
```

8 其他

8.1 双指针/尺取法

8.1.1 一维

```
/*
 * 封装成一个滑窗结构体, 维护区间内不同字母个数 (根据题目需要)
 */
struct window{
    int siz;
    int cnt[26];
    window(){
        siz=0;
        memset(cnt,0,sizeof(cnt));
    }
```

```
    void add(int x){
        if(!cnt[x]){
            siz++;
        }
        cnt[x]++;
    }
    void remove(int x){
        cnt[x]--;
        if(!cnt[x]){
            siz--;
        }
    }
};
/*
 * 尺取法
 */
void solve(char s[],int n,int k){
        window w;
    int n=strlen(s);
    int l=0,r=0;
    ll ans=0;
    while(l<n){
        //右指针速度大于左指针
        while(w.siz<k && r<n){
            w.add(s[r++]-'a');
        }
        if(w.siz<k){
            break;
        }
        //同时更新答案
        ans+=(n-r+1);
        w.remove(s[l++]-'a');
    }
}
```

8.1.2 二维

```
/*
 * 二维尺取法 (枚举一维): 求满足 01 矩阵中 1 个数大于等于 k 的最小矩形大小
 * pre[i][j]: 从 (1,1) 到 (i,j) 的 1 个数 (二维前缀和)
 */
void solve(){
    int ans=INF;
    //枚举列, 确定矩形左右边界
    for(int i=1;i<=m;i++){
        for(int j=1;j<=m;j++){
            //尺取
            int l=1,r=1;
            while(r<=n){
                while(pre[r][j]-pre[r][i-1]-pre[l-1][j]+pre[l-1][i-1]<k && r<=n){
```

```
                r++;
            }
            if(pre[r][j]-pre[r][i-1]-pre[l-1][j]+pre[l-1][i-1]<k){
                break;
            }
            ans=min(ans,(j-i+1)*(r-l+1));
            l++;
        }
    }
}
}
```

8.2 单调队列/单调栈

8.2.1 最大 m 子段和

```
/*
 * 一般来说单调栈就是半个单调队列
 * 子段长度小于等于 m 的最大和
 * pre[]: 前缀和数组
 * q: 单调队列 (双端), 维护下标, 对应的前缀和从小到大
 */
int solve(int n,int m){
    deque<int> q;
    q.push_back(1);
    int ans=0;
    int i=2;
    while(i<=n){
        //维护单调性
        while(!q.empty() && pre[q.back()]>pre[i]){
            q.pop_back();
        }
        //后来的下标总是放在队尾
        q.push_back(i);
        //把老的下标 (超过 i-m 范围) 出队
        while(!q.empty() && q.front()<i-m){
            q.pop_front();
        }
        ans=max(ans,pre[i]-pre[q.front()]);
        i++;
    }
    return ans;
}
```

8.2.2 m 区间最小值

```
/*
 * m 区间的最小值
 * 有时候要根据题目需要手写单调队列
 * 这题也可以直接用 deque 维护下标即可
 * 维护 pair(idx,val), val 从小到大
```

```
 */
struct Queue{
    pair<int,int> a[N];
    int l,r;
    Queue(){
        memset(a,0,sizeof(a));
        l=0;
        r=0;
    }
    void push(int i,int x){
        if(isEmpty()){
            a[r++]=make_pair(i,x);
            return;
        }
        //维护单调递增性
        while(l<r && a[r-1].second>x){
            r--;
        }
        //去除过老的元素
        while(l<r && a[l].first+m<=i){
            l++;
        }
        a[r++]=make_pair(i,x);
    }

    //直接取出堆头（最小值）
    int getMin(){
        return a[l].second;
    }

    bool isEmpty(){
        return l==r;
    }
}q;
```

8.3 矩阵快速幂

```
/*
 * 矩阵快速幂
 * Mat: 矩阵类型，包含一个二维数组
 */
Mat operator * (Mat a, Mat b) {
    Mat ans;
    memset(ans.mat, 0, sizeof(ans.mat));
    for(int i=0; i<n; i++) {
        for(int k=0; k<n; k++) {
            for(int j=0; j<n; j++) {
                ans.mat[i][j] += (a.mat[i][k] % mod) * (b.mat[k][j] % mod) % mod;
                ans.mat[i][j] %= mod;
            }
```

```
        }
    }
    return c;
}
Mat operator ^ (Mat a, ll n)  {
    Mat ans;
    for(int i=0; i<6; i++){
        for(int j=0; j<6; j++){
             ans.mat[i][j]=(i==j);
        }
    }
    while(n) {
        if(n%2){
            ans=ans*a;
        }
        a=a*a;
        n/=2;
    }
    return ans;
}
```

8.4 判断重边

```
map<int,int> mp;
bool isHash(int u,int v){
    if(mp[u*N+v] || mp[v*N+u]){
        return true;
    }
    mp[u*N+v]=mp[v*N+u]=1;
    return false;
}
```

8.5 BM 递推

```
/*
 * 万一真的要用到照抄就是了。注意别抄错
 */
#include<bits/stdc++.h>
using namespace std;
#define rep(i,a,n) for (int i=a;i<n;i++)
#define per(i,a,n) for (int i=n-1;i>=a;i--)
#define pb push_back
#define mp make_pair
#define all(x) (x).begin(),(x).end()
#define fi first
#define se second
#define SZ(x) ((int)(x).size())
typedef vector<int> VI;
typedef long long ll;
typedef pair<int,int> PII;
```

```
const ll mod=1000000007;
ll powmod(ll a,ll b){
    ll res=1;a%=mod; assert(b>=0);
    for(;b;b>>=1){
        if(b&1)res=res*a%mod;a=a*a%mod;
    }
    return res;
}
// head
ll n;
namespace linear_seq {
    const int N=10010;
    ll res[N],base[N],_c[N],_md[N];

    vector<int> Md;
    void mul(ll *a,ll *b,int k) {
        rep(i,0,k+k) _c[i]=0;
        rep(i,0,k) if (a[i]) rep(j,0,k) _c[i+j]=(_c[i+j]+a[i]*b[j])%mod;
        for (int i=k+k-1;i>=k;i--) if (_c[i])
            rep(j,0,SZ(Md)) _c[i-k+Md[j]]=(_c[i-k+Md[j]]-_c[i]*_md[Md[j]])%mod;
        rep(i,0,k) a[i]=_c[i];
    }
    int solve(ll n,VI a,VI b) { // a 系数 b 初值 b[n+1]=a[0]*b[n]+...
        ll ans=0,pnt=0;
        int k=SZ(a);
        assert(SZ(a)==SZ(b));
        rep(i,0,k) _md[k-1-i]=-a[i];_md[k]=1;
        Md.clear();
        rep(i,0,k) if (_md[i]!=0) Md.push_back(i);
        rep(i,0,k) res[i]=base[i]=0;
        res[0]=1;
        while ((1ll<<pnt)<=n) pnt++;
        for (int p=pnt;p>=0;p--) {
            mul(res,res,k);
            if ((n>>p)&1) {
                for (int i=k-1;i>=0;i--) res[i+1]=res[i];res[0]=0;
                rep(j,0,SZ(Md)) res[Md[j]]=(res[Md[j]]-res[k]*_md[Md[j]])%mod;
            }
        }
        rep(i,0,k) ans=(ans+res[i]*b[i])%mod;
        if (ans<0) ans+=mod;
        return ans;
    }
    VI BM(VI s) {
        VI C(1,1),B(1,1);
        int L=0,m=1,b=1;
        rep(n,0,SZ(s)) {
            ll d=0;
            rep(i,0,L+1) d=(d+(ll)C[i]*s[n-i])%mod;
```

```
            if (d==0) ++m;
            else if (2*L<=n) {
                VI T=C;
                ll c=mod-d*powmod(b,mod-2)%mod;
                while (SZ(C)<SZ(B)+m) C.pb(0);
                rep(i,0,SZ(B)) C[i+m]=(C[i+m]+c*B[i])%mod;
                L=n+1-L; B=T; b=d; m=1;
            } else {
                ll c=mod-d*powmod(b,mod-2)%mod;
                while (SZ(C)<SZ(B)+m) C.pb(0);
                rep(i,0,SZ(B)) C[i+m]=(C[i+m]+c*B[i])%mod;
                ++m;
            }
        }
        return C;
    }
    int gao(VI a,ll n) {
        VI c=BM(a);
        c.erase(c.begin());
        rep(i,0,SZ(c)) c[i]=(mod-c[i])%mod;
        return solve(n,c,VI(a.begin(),a.begin()+SZ(c)));
    }
};

int main() {
    /*push_back 进去前 8~10 项左右、最后调用 gao 得第 n 项 */
    //2018 焦作网络赛 L 题
    vector<int>v;
    v.push_back(1);
    v.push_back(4);
    v.push_back(17);
    v.push_back(49);
    v.push_back(129);
    v.push_back(321);
    v.push_back(769);
    v.push_back(1793);
    v.push_back(4097);
    v.push_back(9217);
    int nCase;
    scanf("%d", &nCase);
    while(nCase--){
        scanf("%lld", &n);
        printf("%lld\n",1LL * linear_seq::gao(v,n-1) % mod);
    }
}
```

8.6 大数平方数判断

```
import java.math.BigInteger;
import java.util.Scanner;
```

```
public class IsSquare {
    public static boolean check(BigInteger now){
        if(now.compareTo(BigInteger.ZERO)==0||now.compareTo(BigInteger.ONE)==0){
            return true;
        }
        if(now.mod(BigInteger.valueOf(3)).compareTo(BigInteger.valueOf(2))==0){
            return false;
        }
        String s = now.toString();
        if(s.length()%2==0){
            s = s.substring(0,s.length()/2+1);
        }else{
            s = s.substring(0,(1+s.length())/2);
        }
        BigInteger res = BigInteger.ZERO;
        BigInteger m = new BigInteger(s);
        BigInteger two = new BigInteger("2");
        if(s == "1"){
            res = BigInteger.ONE;
        }else{
            while(now.compareTo(m.multiply(m)) < 0){
                m = (m.add(now.divide(m))).divide(two);
            }
            res = m;
        }
        if (res.multiply(res).compareTo(now) == 0){
            return true;
        }
        return false;
    }
}
```

8.7 技巧

8.7.1 快读

8.7.2 离散化

```
/*
 * 离散化
 */
void solve(){
    int x[]={0,2,10000,1,3,500,19999999,212222222,13};
    int n=8;
    sort(x+1,x+1+n);
    for(int i=1;i<=n;i++){
        //根据具体题目调整
        int k=lowerbound(x+1,x+1+n,x[i])-x;
    }
}
```

8.8 一些比较有意思的题目

8.8.1 hdu6468——求 1-n 字典序第 m 个数

```
#include <bits/stdc++.h>
using namespace std;
/*
 * 求 1-n 字典序排列的第 m 个数
 */
int solve(int n,int m){
    //考虑一颗完全 10 叉树，树的所有节点就是 1-n，要求的就是前序遍历的第 m 个节点
    //m 是可以走的步数
    int i=1;
    m--;
    while(m!=0){
        //计算 i 到 i+1 的字典序中间相隔的个数
        int s=i,e=i+1;
        int num=0;
        //防止越界
        while(s<=n){
            //计算每一层相差的个数
            //n+1: 比如 20-29 其实是 10 个，而 e 就不用 +1，因为 e 在这里表示 30(40/50...)
            num+=min(n+1,e)-s;
            s*=10;
            e*=10;
        }
        if(m<num){
            //向下
            i*=10;
            //走一步
            m--;
        }else{
            //向右
            i++;
            //对前序遍历来说，走了 num 步
            m-=num;
        }
    }
    return i;
}
int main(){
    int n,m;
    scanf("%d%d",&n,&m);
    printf("%d\n",solve(n,m));
    return 0;

}
```