ACM Template

Zeng Xiaocan

September 16, 2019

Contents

1 字符串

1.1 string 类操作

```
/**
reverse(s.begin(), s.end());
transform(s.begin(), s.end(), s.begin(), ::toupper);  (::tolower)
string->number:
    int a;
    stringstream(s) >> a;
number->char*
    char s[100];
    sprint(s,"%d",a);
//返回 pos 开始的长度为 len 的字符串
substr(pos,len);
//在 pos 位置插入字符串 s
insert(int pos,string s)
//从索引 pos 开始往后全删除
erase(pos);
//从索引 pos 开始往后删 num 个
erase(pos,num);
//删除迭代器 it 指向的字符，返回删除后迭代器的位置
erase(it);
//删除迭代器 [first, last) 之间的所有字符，返回删除后迭代器的位置
erase(first,last);
//从 pos 开始查找字符 c/字符串 s 在当前字符串的位置
int find(c/s,pos);
**/
```

1.2 KMP

1.3 Trie 树/AC 自动机

```
/*
 * AC 自动机
 * tr[u][i]: 表示节点 u 的子节点 ('a'-'z'=>0-25) 编号
 * cnt: 总节点数/节点编号
 * fail[u]: 节点 u 的 fail 指针指向节点编号
 * ==> fail 指针: 指向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) 的最长字符串
 * val[u]: 以节点 u 所表示字符串作为结尾的单词数
 * sum[u]: 以节点 u 所表示字符串作为前缀的单词数
 */
//插入单词，构建字典树
void insert(char *s){
    int len=strlen(s);
    int now=0;
    for(int i=0;i<len;i++){
        int id=s[i]-'a';
        if(!tr[now][id]){
            tr[now][id]=++cnt;
        }
        now=tr[now][id];
        sum[now]++;
    }
    val[now]++;
}
```

```
//使用 bfs 构造 fail 指针
void build(){
    queue<int> q;
    //第一层节点均指向根
    for(int i=0;i<26;i++){
        if(tr[0][i]){
            fail[tr[0][i]]=0;
            q.push(tr[0][i]);
        }
    }
    while(!q.empty()){
        int u=q.front();
        q.pop();
        //处理出 fail 指针或者该节点走一步 (fail 跳转不算) 能到达的节点
        for(int i=0;i<26;i++){
            if(tr[u][i]){
                //已经路径压缩，子节点 fail 直接指向父节点的 fail 的对应子节点
                //(无论是否存在，若不存在会自动指向上一级 fail)
                fail[tr[u][i]]=tr[fail[u]][i];
                q.push(tr[u][i]);
            }else{
                //预处理该节点走一步 (fail 跳转不算) 能到达的节点
                //相当于路径压缩，这样下面的节点就不用 while(fail[fail[...]]) 这样跳
                tr[u][i]=tr[fail[u]][i];
            }
        }
    }
}
int query(char *s){
    int len=strlen(s);
    int now=0;
    int ans=0;
    for(int i=0;i<len;i++){
        int id=s[i]-'a';
        now=tr[now][id];
        //从当前节点一直往上跳转直到根或者 val[t] 为-1
        for(int t=now;t&&~val[t];t=fail[t]){
            ans+=val[t];
            //避免重复匹配
            val[t]=-1;
        }
    }
    return ans;
}
/*
 * 01 异或字典树
 */
void insert(ll x){
    int rt=0;
    for(int i=32;i>=0;i--){
        //从高位，分解为二进制数位
        int id=(x>>i)&1;
        if(!tr[rt][id]){
            tr[rt][id]=++cnt;
        }
        rt=tr[rt][id];
```

```
    }
    //末尾标记原数字
    val[rt]=x;
}
ll query(ll x){
    int rt=0;
    for(int i=32;i>=0;i--){
        int id=(x>>i)&1;
        //不管这一位 (id) 是 1 还是 0, 优先找与这一位不同的节点, 这样异或值就会尽量大
        if(tr[rt][id^1]){
            rt=tr[rt][id^1];
        }
        else{
            rt=tr[rt][id];
        }
    }
    return val[rt];
}
/*
 * AC 自动机上 dp
 * bzoj1030: 求长度为 m 且不含有所给 n 个单词的字符串的个数
 * dp[i][j]: 长度为 i(从 trie 树根节点走 i 步) 在节点 j 的满足条件字符串个数
 * ans=sum(dp[m][i]) i(0-cnt)
 * val[i]: 标记 i 节点是否可以访问 (即对应字符串是否含有所给单词)
 *         在构建 fail 指针的时候, 也要更新 val[u]|=val[fail[u]]
 */
int solve(){
    dp[0][0]=1;
    //枚举步数
    for(int i=1;i<=m;i++){
        //枚举所有节点
        for(int j=0;j<=cnt;j++){
            //标记的单词 (或是 fail 指针有标记) 都不能经过
            if(val[j]){
                continue;
            }
            for(int k=0;k<26;k++){
                dp[i][tr[j][k]]=(dp[i][tr[j][k]]+dp[i-1][j])%MOD;
            }
        }
    }
    int sum=1;
    for(int i=1;i<=m;i++){
        sum=(sum*26)%MOD;
    }
    int ans=0;
    for(int i=0;i<=cnt;i++){
        if(!val[i]){
            ans+=dp[m][i];
        }
    }
    return ((sum-ans)%MOD+MOD)%MOD;
}
/*
 * AC 自动机上数位 dp
 * bzoj3530: 给定整数 n 和其他 m 个整数, 求 0-N 中, 不含有这 m 个整数子串的数的个数
```

```
 * 先将模式串插入 trie 树中, 跑数位 dp 模板 (不用分解数位了), dp 数组多加一维标记节点位置
 * 判断下一个节点有效性 (即是否含有模式串, 注意题目是含有任意一个还是含有全部 (状态压缩))
 * zero: 是否有前导零限制, 即 zero 为 true 时, 该位不能选 0
 * flag: 表示是否含有 trie 树中的串
 * dp[i][j][k]: 表示前 i 位数在 u 节点上前面是否含有无效串的数的个数
 */
int dfs(int len,int u,int flag,int limit,int zero){
    //递归边界, 即枚举完一个可能的数, 如果没有任何限制就返回 1, 即一个满足要求的数
        if(len<0){
        return (!flag && !zero) ? 1 : 0;
    }
    //在没有上限限制和没有前导零限制的情况下直接记忆化搜索
        if(dp[len][u][flag]!=-1 && !limit && !zero){
        return dp[len][u][flag];
    }
        int up = limit ? n[len]-'0' : 9;
    int ans = 0;
        for(int i=0;i<=up;i++){
        //有前导零限制且当前枚举位为 0, 相当于这一状态下面递归的所有状态都是无效的
        //所以直接舍弃掉这一位从下一位开始并从 trie 树根节点开始
        //比如 1234, 枚举最高位的时候 0 是不可以的, 有前导零限制
        //所以 0000,0001,0002 等这些数都是无效的
        //直接从高二位开始算, 也就相当于是 234, 同理这一位枚举到也不行
        //枚举到 1 的时候 100, 101 这些就是有效状态
            if(i==0 && zero){
            ans=(ans+dfs(len-1,0,flag,limit && i==up, 1))%MOD;
        }else{
            //当前节点含有 trie 树的串或者是下一个节点含有
            ans=(ans+dfs(len-1,tr[u][i],flag|val[tr[u][i]],limit && i==up,zero && i==0))%MOD;
        }
    }
    if(!limit && !zero){
        dp[len][u][flag] = ans;
    }
        return ans;
}
int solve(){
    len=strlen(n);
    //反转, 从高位枚举
    reverse(n,n+len);
    memset(dp,-1,sizeof(dp));
    return dfs(len-1,0,0,1,1);
}
/*
 * 矩阵快速幂加速 dp(递推)
 * 有时候推出 ac 自动机加 dp, 结果 n 的范围是 1e9, 就需要用到矩阵快速幂来优化 dp
 * bzoj1009: 求有多少个 n 位数不含有所给 m 位数的子串
 * 和上面 ac 自动机上 dp 是一样的, 枚举长度, 枚举节点, 枚举子节点 (0-9), 状态转移
 * 由于 n 比较大, 而 m 比较小, 我们可以把问题转化为求出从 trie 树根节点走 n 步的有效路径数
 * 所以通过构建 fail 树后, 可以计算出 trie 树初始邻接矩阵 (1 步能走到的), 注意无效状态
 * 然后使用矩阵快速幂加速, 计算邻接矩阵的 n 次方, 即走 n 步的路径数矩阵
 * 枚举 i, 计算 sum(m[0][i]) 即为答案 (从根节点开始到其他各个节点刚好走 n 步的路径数)
 */
int solve(){
    Mat tmp,a;
    memset(tmp.m,0,sizeof(tmp.m));
```

```
    //dp 初始状态
    tmp.m[0][0]=1;
    for(int i=0;i<=tot;i++){
        for(int j=0;j<10;j++){
            if(!val[tr[i][j]]){
                //从 i 节点到 j 节点存在可行路径（包括计算 fail 指针时指向祖先节点的关系）
                a.m[i][tr[i][j]]+=1;
            }
        }
    }
    //矩阵快速幂求长度为 n 的路径条数
    a=tmp*(a^n);
    int ans=0;
    for(int i=0;i<=tot;i++){
        //所有从根节点到任意节点 n 步的路径数
        ans+=a.m[0][i];
        ans%=k;
    }
    return ans;
}
```

1.4 Manacher

1.5 后缀数组

1.5.1 倍增

1.5.2 DC3

1.5.3 后缀数组经典应用

```
/*
 * 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++){
            //i 起点 1<<j 长度
            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 ========================
//1. 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);
}
//2. 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;
}
//3. 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;
}
//4. 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;
}
//5.(长度大于等于 k) 的不同子串个数
/*
 * 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](k 为 1 的情况)
 * 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++){
        if(h[i]){
            h[i]=max(0,h[i]-(k-1));
        }
        ans+=max(0,(n-sa[i]-(k-1)-h[i]));
        //k 为 1 的情况
        //ans+=(n-sa[i]-h[i]);
    }
    return ans;
}
//6. 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;
}
//7. 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);
}
//8. 字符串 S 由字符串 T 重复 Q 次得到, 求最大 Q
/*
 * 使用 kmp 更方便, 如果非要用后缀数组请用 dc3 模板
 * 枚举 T 串长度 k, 先判断 len(S)%len(T)==0?
 * 再判断 lcp(suffix(0),suffix(k))==n-k?
 * suffix(i) 指首字符位置为 i 的后缀
 * 因为 suffix(0) 固定, 无需预处理出整个 RMQ
 * 只需 O(n) 求出所有 lcp(suffix(0),x) 即可
 */
int solve8(){
    lcp[rk[0]]=n;
    for(int i=rk[0]-1;i>=0;i--){
        lcp[i]=min(lcp[i+1],h[i+1]);
    }
    for(int i=rk[0]+1;i<=n;i++){
        lcp[i]=min(lcp[i-1],h[i-1]);
    }
    for(int k=1;k<=n;k++){
        if(n%k==0 && lcp[rk[k]]==n-k){
            return n/k;
        }
    }
}
//9. 重复次数最多的连续重复子串
/*
 * 暂时无法理解, 贴个题解板子
 */
void solve9(){
    int ans=0,pos=0,lenn;
    //保证有重复, 所以是 len/2
    for(int i=1;i<=len/2;i++){
        for(int j=0;j<len-i;j+=i){
            if(str[j]!=str[j+i]){
                continue;
            }
            //通过下标查询
```

```
            int k=solve1(j,j+i);
            int tol=k/i+1;
            int r=i-k%i;
            int p=j;
            int cnt=0;
            for(int m=j-1;m>j-i&&str[m]==str[m+i]&&m>=0;m--){
                cnt++;
                if(cnt==r){
                    tol++;
                    p=m;
                }
                else if(rk[p]>rk[m]){
                    p=m;
                }
            }
            if(ans<tol){
                ans=tol;
                pos=p;
                lenn=tol*i;
            }else if(ans==tol && rk[pos]>rk[p]){
                pos=p;
                lenn=tol*i;
            }
        }
    }
    //这里，如果字符总长度小于 2，那么就在原串中找出一个最小的字符就好
    if(ans<2){
        char ch='z';
        for(int i=0;i<len;i++){
            if(str[i]<ch){
                ch=str[i];
            }
        }
        printf("%c\n",ch);
        return;
    }
    for(int i=pos;i<pos+lenn;i++){
        printf("%c",str[i]);
    }
    printf("\n");
}
//10. 最长公共子串
/*
 * 求 A 和 B 的最长公共子串即转化为求 A 和 B 后缀的最长公共前缀
 * 将 A 和 B 拼接求出 h 数组，当 suffix(sa[i]) 和 suffix(sa[i-1]) 不在同个字符串时
 * h[i] 才有效，求出最大值即可
 */
int solve10(){
    int ans=0;
    for(int i=1;i<=n;i++){
        //al 为第一个字符串长度
        if(sa[i]<al && sa[i-1]<al || sa[i]>=al && sa[i-1]>=al){
            continue;
        }
        ans=max(ans,h[i]);
    }
```

```
    return ans;
}
//11. 求长度大于等于 k 的公共子串个数
/*
 * 同样是转化为 A 和 B 所有后缀的最长公共前缀大于等于 k 的贡献累加起来
 * 用常用的处理方法，将两个字符串拼接，中间用分隔符隔开，求出 h 数组
 * 然后对 h 数组分组 (把 h 值大于等于 k 的分在一组)
 * 然后在这道题里要使用一个单调栈来维护前面的后缀对后面的后缀的贡献
 * 单调栈维护一个 h 值和一个该 h 值在前面充当 lcp 的
 * sta[i][0]: 维护 h 值 sta[i][1]: 维护这个 h 值代表前面多少个后缀的 lcp 贡献 (最小 h 值)
 * 维护的类似一种前缀和的东西，具体看代码详细注释
 */
int solve11(int k){
    int top=0;
    int tot=0, ans=0;
    //对 h 分组
    for(int i=1;i<=n;i++){
        if(h[i]<k){
            //单独一个后缀的分组，无需考虑
            top=0;
            tot=0;
        }else{
            //cnt 表示 h[i] 作为 lcp 的后缀个数
            int cnt=0;
            //前一个后缀属于 A，计算贡献
            //即能产生多少个长度大于等于 k 的公共子串
            if(sa[i-1]<al){
                //前面 (i-1) 必须是属于 A 的后缀，才能计算这个贡献
                //所以后面把 B 的后缀入栈是没毛病的，这里维护的实际上是一个类似前缀和的东西
                //即栈顶元素已经是累加了栈底所有元素的贡献
                //但是这个单调栈是动态变化的，所以不能单纯保留栈顶元素
                cnt++;
                //比如 h[i]=4，例如 abcd，那么大于等于 k 的子串就可以是
                //ab abc abcd 这三个，所以是 h[i]-k+1
                tot+=h[i]-k+1;
            }
            //维护单调栈 (栈顶到栈底递减)
            //这里不用考虑 h[i] 是 A 的还是 B 的
            while(top && h[i]<=sta[top-1][0]){
                top--;
                //更新贡献
                //这里实际上应该是 tot(贡献) 减去 sta[top][0]*sta[top][1]
                //再加上 h[i]*sta[top][1]
                //相当于就是栈顶被替换了，所以栈顶这个 h 值之前充当了多少后缀的 lcp
                //都要改成这个新的栈顶 h[i] 的贡献了
                tot+=(h[i]-sta[top][0])*sta[top][1];
                //既然 h[i] 比 sta[top][0] 小，那么 sta[top][0] 能作为 lcp(h 最小值) 的 h[i]
                //肯定也可以，所以累加上即可
                cnt+=sta[top][1];
            }
            //此时栈里的所有 h 值都小于 h[i]
            //入栈，维持单调性
            sta[top][0]=h[i];
            sta[top][1]=cnt;
            top++;
            //属于 B 的后缀，累加 A 的贡献
```

```
            if(sa[i]>al){
                ans+=tot;
            }
        }
    }
    //同理对 B 串的后缀维护单调栈, 累计 B 的贡献
    top=tot=0;
    for(int i=1;i<=n;i++){
        if(h[i]<k){
            top=0;
            tot=0;
        }else{
            int cnt=0;
            if(sa[i-1]>al){
                cnt++;
                tot+=h[i]-k+1;
            }
            while(top && h[i]<=sta[top-1][0]){
                top--;
                tot+=(h[i]-sta[top][0])*sta[top][1];
                cnt+=sta[top][1];
            }
            sta[top][0]=h[i];
            sta[top][1]=cnt;
            top++;
            if(sa[i]<al){
                ans+=tot;
            }
        }
    }
    return ans;
}
//12. 给 n 个字符串, 出现在不少于 k 个字符串的最长子串
/*
 * 同样是拼接字符串对 h 数组分组, 二分答案进行判断
 */
bool check12(int mid){
    bool flag=false;
    for(int i=1;i<len;i++){
        //不满足要求的组直接不用管
        if(h[i]<mid){
            continue;
        }
        //组内不同字符串个数
        int cnt=0;
        //标记某个字符串是否出现过
        memset(vis,false,sizeof(vis));
        while(h[i]>=mid && i<len){
            if(!vis[idx[sa[i-1]]]){
                vis[idx[sa[i-1]]]=true;
                cnt++;
            }
            //找到 1 组
            i++;
        }
        //注意最后一个的判断
```

```
        if(!vis[idx[sa[i-1]]]){
            vis[idx[sa[i-1]]]=1;
            cnt++;
        }
        //出现在 k 个不同字符串中
        if(cnt>k){
            if(!flag){
                flag=true;
                //清空之前的答案
                //ans 存储出现在至少 k 个不同字符串的后缀 sa 值
                ans.clear();
                ans.push_back(sa[i-1]);
            }
            else{
                //只需加入最后一个满足条件后缀的 sa
                ans.push_back(sa[i-1]);
            }
        }
    }
    //是否有满足要求的答案
    return flag;
}
void solve12(){
    ans.clear();
    int l=1,r=len;
    while(l<=r){
        int mid=(l+r)>>1;
        if(check(mid)){
            l=mid+1;
        }
        else{
            r=mid-1;
        }
    }
    //输出多个最长公共子串 (至少在 k 个字符串中出现过)
    for(int i=0;i<ans.size();i++){
        for(int j=ans[i];j<ans[i]+r;j++)
            printf("%c",s[j]-1+'a');
        printf("\n");
    }
}
//13. 给 n 个字符串, 在每个字符串中至少出现两次且不重叠的最长子串
/*
 * 把每个字符串拼接起来, 中间用分隔符分开, 求 h 数组
 * 二分子串长度, 对 h 数组分组, 维护每一个字符串在当前组的最小和最大 sa 值
 * 然后判断不重叠即可 (也同时判断了出现 2 次)
 */
bool check13(int mid){
    //后缀起始位置的最大最小值
    for(int i=0;i<n;i++){
        mx[i]=0;
        mn[i]=INF;
    }
    for(int i=1;i<=len;i++){
        if(h[i]<mid){
            //重置分组
```

```
            for(int i=0;i<n;i++){
                mx[i]=0;
                mn[i]=INF;
            }
        }else{
            //一个 h 值表示两个后缀
            mx[idx[sa[i]]]=max(mx[idx[sa[i]]],sa[i]);
            mn[idx[sa[i]]]=min(mn[idx[sa[i]]],sa[i]);
            mx[idx[sa[i-1]]]=max(mx[idx[sa[i-1]]],sa[i-1]);
            mn[idx[sa[i-1]]]=min(mn[idx[sa[i-1]]],sa[i-1]);
            bool flag=true;
            //需要判断每个字符串 (都要出现两次以上，且不重叠)
            for(int j=0;j<n;j++){
                if(mx[j]-mn[j]<mid){
                    flag=false;
                    break;
                }
            }
            if(flag){
                return true;
            }
        }
    }
    return false;
}
int solve13(){
    //len 为拼接后的字符串长度
    int l=0,r=len+1;
    int ans=0;
    while(l<=r){
        int mid=(l+r)>>1;
        if(check13(mid)){
            ans=mid;
            l=mid+1;
        }else{
            r=mid-1;
        }
    }
    return ans;
}
//14. 给 n 个字符串，正序或逆序出现在每个字符串的最长子串
/*
 * 把每个字符串正序和逆序拼接起来，中间用分隔符分开，求 h 数组
 * 二分子串长度，对 h 数组分组，判断是否存在一组 h 值里面所属 n 个不同字符串
 * 所以在拼接字符串的时候要把每个字符串 (正序和逆序) 的每个字符标记是第几个字符串
 * for(int j=0;j<siz;j++){
        idx[len]=i;
        //加 2*n+1 保证不会跟分隔符相同
        s[len++]=ss[i][j]-'A'+2*n+1;
    }
    idx[len]=i;
    s[len++]=sep++;
 */
bool check14(int mid){
    //这里用 set 来进行组内所属字符串的去重
    set<int> se;
```

```
    for(int i=1;i<len;i++){
        if(h[i]>=mid){
            se.insert(idx[sa[i]]);
            se.insert(idx[sa[i-1]]);
        }else{
            if(se.size()==n){
                return true;
            }
            se.clear();
        }
    }
    //注意最后一个组
    return se.size()==n;
}
int solve14(){
    if(n==1){
        return strlen(ss[0]);
    }
    int l=1,r=100;
    //变量记得初始化!!!
    int ans=0;
    while(l<=r){
        int mid=(l+r)>>1;
        if(check14(mid)){
            ans=mid;
            l=mid+1;
        }else{
            r=mid-1;
        }
    }
    return ans;
}
//15. 求两个字符串的公共子串，且只能在母串中出现一次，求满足的最短子串
/*
 * 两字符串拼接，按求最长公共子串的方法，不过要判断是否重复出现，重复出现的话，
 * 其对应后缀一定是相邻的，所以要判断相邻的是否有可能重复出现，要枚举从 h[i] 到 1
 */
int solve(int n){
    int ans=0x3f3f3f3f;
    for(int i=1;i<=n;i++){
        if(h[i]==0 || sa[i]<len1 && sa[i-1]<len1 || sa[i]>=len1 && sa[i-1]>=len1){
            continue;
        }
        for(int j=h[i];j>=1;j--){
            if(j>h[i-1] && j>h[i+1]){
                ans=min(ans,j);
            }
        }
    }
    if(ans==0x3f3f3f3f){
        ans=-1;
    }
    return ans;
}
//几个注意点
//字符串数组一般可以用一个 int 数组来表示，特别是需要拼接字符串的题目
```

```
//拼接字符串先设定一个比较大的分隔符，比如 300，然后每次分割一次都要 +1
//注意最后一位补 0 s[len]=0
/*
    len=0;
    int sep=30;
    for(int i=0;i<n;i++){
        scanf("%s",str[i]);
        siz[i]=strlen(str[i]);
        for(int j=0;j<siz[i];j++){
             s[len++]=str[i][j]-'a'+1;
             idx[len-1]=i;
        }
        s[len++]=sep++;
    }
    s[len]=0;
    build_sa(len,250);
 */
```

1.6 字符串哈希

1.7 其他

1.7.1 最大/小表示法

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
        return (r[a]+q)%3!=r[b];
    }
}
```

2.3 最短路

2.4 最小生成树

2.4.1 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;
}
```

2.5 次小生成树

```
/*
 * 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,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;
}
```

2.6 tarjan

2.6.1 强连通分量

```
/*
 * 求有向图的强连通分量
 * 强连通分量: 分量内任意两个点都能互相到达
 * 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);
        }
    }
}
```

2.6.2 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);
        }
    }
}
```

2.6.3 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);
        }
    }
}
```

2.7 LCA

2.7.1 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 = 31 - __builtin_clz(r - l + 1);
    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];
}
```

2.7.2 Tarjan 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);
        }
    }
}
```

2.8 最小支配集/最小点覆盖/最大独立集

```
/*
 * 贪心求解树上最小支配集/最小点覆盖/最大独立集
 * p[i]: i 的父节点
 * dft[i]: dfs 序第 i 个节点
 * s[]: 求解的集合
 * vis[i]: i 是否被集合覆盖
 */
void dfs(int u,int f){
    //dfs 序
    dft[k++]=u;
    for(int i=head[u];i!=-1;i=edge[i].next){
        int v=edge[i].v;
        if(v==f){
            continue;
        }
        p[v]=u;
        dfs(v,u);
    }
}
bool s[N],vis[N];
//最小支配集 (覆盖所有点)
int greedy1(){
    int ans=0;
    //逆序进行贪心
    for(int i=n-1;i>=0;i--){
        int t=dft[i];
        //当前点未被覆盖
        if(!vis[t]){
```

```
            //父节点不属于支配集
            if(!s[p[t]]){
                //将父节点加入支配集
                s[p[t]]=true;
                ans++;
            }
            //父节点加入支配集, 即该节点和父节点的父节点都被覆盖
            vis[t]=true;
            vis[p[t]]=true;
            vis[p[p[t]]]=true;
        }
    }
    return ans;
}
//最小点覆盖集 (覆盖所有边)
int greedy2(){
    int ans=0;
    //该贪心策略要排除根节点 dft[0]
    for(int i=n-1;i>=1;i--){
        int t=dft[i];
        //当前节点和父节点都不属于覆盖集
        if(!vis[t] && !vis[p[t]]){
            //父节点加入覆盖集
            s[p[t]]=true;
            ans++;
            //当前点和父节点的父节点被覆盖
            vis[t]=true;
            vis[p[t]]=true;
        }
    }
    return ans;
}
//最大点独立集
int greedy3(){
    int ans=0;
    for(int i=n-1;i>=0;i--){
        int t=dft[i];
        //当前节点没有被覆盖
        if(!vis[t]){
            //加入独立集
            s[t]=true;
            ans++;
            //当前节点和父节点被覆盖
            vis[t]=true;
            vis[p[t]]=true;
        }
    }
    return ans;
}
```

3 网络流

3.1 最大流 (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，dfs 增广，时间复杂度：O(n^2 m)
 */
bool bfs(){
    memset(dep,0,sizeof(dep));
    dep[s]=1;
    queue<int> q;
    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;
            //可用流量大于 0 && 未标记深度
            if(edge[i].w>0 && dep[v]==0){
                dep[v]=dep[u]+1;
                q.push(v);
            }
        }
    }
    return dep[t]!=0;
}
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;
}
int dinic(){
    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;
```

```
}
```

3.2 最小费用最大流

```
/*
 * 最小费用最大流 (mcmf)
 * inq[i]: i 节点是否在队列中
 * d[i]: bellmanford 算法的距离
 * p[i]: 记录流向 i 的上一条边编号
 * a[i]: i 的可改进量 (通过量)
 */
bool BellmanFord(int &flow,int &cost){
    for(int i=0;i<N;i++){
        d[i]=INF;
        inq[i]=false;
    }
    d[s]=0;
    p[s]=0;
    a[s]=INF;
    queue<int> q;
    q.push(s);
    inq[s]=true;
    while(!q.empty()){
        int u=q.front();
        q.pop();
        inq[u]=false;
        for(int i=head[u];i!=-1;i=edge[i].next){
            int v=edge[i].v;
            int w=edge[i].w;
            int c=edge[i].c;
            if(w>0 && d[v]>d[u]+c){
                d[v]=d[u]+c;
                p[v]=i;
                a[v]=min(a[u],w);
                if(!inq[v]){
                    q.push(v);
                    inq[v]=true;
                }
            }
        }
    }
    if(d[t]==INF){
        return false;
    }
    flow+=a[t];
    cost+=d[t]*a[t];
    for(int u=t;u!=s;u=edge[p[u]].u){
        edge[p[u]].w-=a[t];
        edge[p[u]^1].w+=a[t];
    }
    return true;
}
void mcmf(int &flow,int &cost){
    flow=0,cost=0;
    while(BellmanFord(flow,cost));
}
```

3.3 上下界网络流

```
/*
 * 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
 * */
```

3.4 常见模型

```
/*
 * 0. 最小割 = 最大流
 * 1. 最大权闭合子图 = 正权点和-最小割
 *    闭合子图就是给定一个有向图, 从中选择一些点组成一个点集 V。对其中任意一个点, 其后续节点都仍然在 V 中
 *    建立源点 s 和汇点 t, 将源点 s 与所有权值为正的点相连, 容量为权值
 *    将所有权值为负的点与汇点 t 相连, 容量为权值的绝对值
 *    权值为 0 的点不做处理, 同时将原来的边容量设置为无穷大
 * 2. 最小路径覆盖 (见二分图)
 * 3. 有组合/套餐的最大收益问题转化为最小割问题 (luoguP1361)
 * 4. 最小割点, 拆点 (包括 s 和 t, 最后跑 s+n 到 t 的最小割) 转化为最小割 (边)
 * 5. 牛吃饭问题, 一头牛有喜欢的饮料列表和食物列表, 求最大能满足的牛数
 *    牛拆点, 一边连饮料, 一边连食物, 跑最大流即可
 * 6. 火星探险/深海机器人问题, 在一个二维矩阵上走, 某些点第一次走到能获得收益, 可重复走, 求最大收益
 *    如果有多个起点和终点分别连上源汇点, 中间拆点, x 和 x'之间连两条边, <x,x',1,-1> 和 <x,x',INF,0>
 * 7. 跑多次网络流的情况, 如果无需重新建图, 也要记住将之前跑的流量重置, 可以在 Edge 结构体里设置一个 cap
 *    加边时 cap 和 w 相同, 重置时枚举所有边使 w=cap
 * 8. 输出方案考虑枚举边判断流量, 以及在 dfs 时记录前驱
 * 9. 注意 N 的大小, 注意顶点数 (特别是拆点后), 注意 s 和 t, 注意网络流中是有向边, 反边不等于无向边
 */
```

4 二分图

4.1 判断二分图

```
/*
 * 染色法判断二分图 (奇圈): 任何无回路的图都是二分图
 * col[i]: 点 i 的颜色 (1,-1 代表不同颜色, 0 代表未染色)
 */
bool dfs(int u,int c){
    //将 u 涂成 c 颜色
    col[u]=c;
    for(int i=head[u];i!=-1;i=edge[i].next){
        int v=edge[i].v;
        //相邻顶点不能同色
        if(col[v]==c){
            return false;
        }
```

```
        //如果未涂色则尝试递归涂色，如果失败返回 false，则该判断也 false
        if(col[v]==0 && !dfs(v,-c)){
            return false;
        }
    }
    return true;
}
bool solve(){
    for(int i=0;i<n;i++){
        if(col[i]==0 && !dfs(i,1)){
            return false;
        }
    }
    return true;
}
```

4.2 最大匹配

```
/*
 * 可以加个源点汇点跑最大流，但有时候比较毒瘤可能会 T...
 * 匈牙利算法
 * g[][]: 图的邻接矩阵表示
 * vis[i]: 本轮是否已访问
 * mt[i]: i 所匹配的点
 */
bool dfs(int u, int n){
    for(int i=1;i<=n;++i){
        if(!vis[i] && g[u][i]){
            vis[i] = true;
            if(mt[i]==-1 || dfs(mt[i], n)){
                mt[i] = u;
                return true;
            }
        }
    }
    return false;
}
int solve(){
    int ans=0;
    memset(mt,-1,sizeof(mt));
    for(int i=1;i<=n;++i){
        memset(vis,0,sizeof(vis));
        ans += dfs(i, n);
    }
    return ans;
}
```

4.3 完美匹配

```
/*
 * 跑一遍 Hungary 算法，判断是否每个点都有匹配
 * mat[i]: 左点集 i 对应的匹配点
 */
```

4.4 最优匹配 (KM 算法)

```
/* * KM 算法计算二分图带权最优匹配
 * n 个男生和 n 个女生，两两之间有好感度 (边权)，匹配使得总好感度最大
 * love[i][j]: 女生 i 和男生 j 的好感度
 * exg[i]/exb[i]: 女生期望值 (初始值为最高好感度)，男生期望值 (初始值为 0)
 * visg[i]/visb[i]: 记录每一轮匹配中是否匹配过
 * match[i]: 男生 i 匹配到的女生，初始值为-1
 * slack[i]: 男生 i 还需要多少期望值才能匹配到女生
 */
bool dfs(int u){
    visg[u]=true;
    for(int i=0;i<n;i++){
        if(visb[i]){
            continue;
        }
        //计算期望度和好感度差值
        int gap=exg[u]+exb[i]-love[u][i];
        //这里改成 gap!=0 可以求出最小完备匹配
        if(gap==0){
            //符合要求
            visb[i]=true;
            //男生没有匹配，或者男生对应女生可以找另一个男生
            if(match[i]==-1 || dfs(match[i])){
                match[i]=u;
                return true;
            }
        }else{
            //无法找到匹配的，必须修改期望度 (对当前女生 u，也就是 KM 算法里枚举的 i)
            slack[i]=min(slack[i],gap);
        }
    }
    return false;
}
int km(){
    memset(match,-1,sizeof(match));
    memset(exb,0,sizeof(exb));
    for(int i=0;i<n;i++){
        exg[i]=love[i][0];
        for(int j=1;j<n;j++){
            exg[i]=max(exg[i],love[i][j]);
        }
    }
    for(int i=0;i<n;i++){
        memset(slack,INF,sizeof(slack));
        while(true){
            memset(visg,false,sizeof(visg));
            memset(visb,false,sizeof(visb));
            //找到
            if(dfs(i)){
                break;
            }
            //找不到
            int d=INF;
            //枚举男生
            for(int j=0;j<n;j++){
```

```
                if(!visb[j]){
                    //降到某个 slack[j]
                    d=min(d,slack[j]);
                }
            }
            //这一轮问过的女生降低期望值，男生增加期望值
            for(int j=0;j<n;j++){
                if(visg[j]){
                    exg[j]-=d;
                }
                if(visb[j]){
                    exb[j]+=d;
                }else{
                    //没访问过的男生，因为女生期望值降低，所以距离女生期望值又近一步
                    slack[j]-=d;
                }
            }
        }
    }
    int ans=0;
    for(int i=0;i<n;i++){
        ans+=love[match[i]][i];
    }
    return ans;
}
```

4.5 最小支配集/最小点覆盖/最大独立集/最大团/最小路径覆盖

```
/*
    1. 二分图一些性质
        最小点覆盖数 == 最大匹配数    ||   最小边覆盖数 == 最大独立数 == 顶点数-最大匹配数
        最大独立集 == 补图的最大团
        (最大独立集: 一个最大的点的集合, 该集合内的任意两点没有边相连)
        (最大团: 一个最大的点的集合, 该集合内的任意两点都有边相连)
    2. 有向图的最小路径覆盖问题 ==> 二分图解决
        路径覆盖: 通过几条路径 (多条边) 覆盖有向图所有顶点, 一个点也可成为路径覆盖, 长度为 0
        2.1 最小不相交路径覆盖：每一条路径经过的顶点各不相同。例如 1->3>4, 2, 5
            将有向图的每个点拆成入点和出点, 对每一条有向边 (u,v), 都在二分图中将 u 的入点连向 v 的出点
            结论: 最小路径覆盖数 =|V(G)|-二分图最大匹配数
        2.2 打印出每条路径:
            在 dinic 算法的 dfs 增广中, 记录流向
            if(dis>0){
                ......
                to[u]=v;
                return dis;
            }
            枚举所有入点 (1-n), 当 to[i] 不为零时, 沿着 i=to[i] 路径输出即可
            for(int i=1;i<=n;i++){
                if(to[i]){
                    int t=i;
                    do{
                        if(t>n){
                            t-=n;
                        }
                        printf("%d ",t);
                        int x=to[t];
```

```
                    to[t]=0;
                    t=x;
                }while(t);
            printf("\n");
        }
    2.3 最小可相交路径覆盖：每一条路径经过的顶点可以相同。例如 1->3->4, 2->3>5
        先用 floyd 求出传递闭包，然后对当前的图求最大匹配 (可以直接用邻接矩阵跑匈牙利算法)
        for(int k=1;k<=n;k++){
            for(int i=1;i<=n;i++){
                for(int j=1;j<=n;j++){
                    if(a[i][k]&&a[k][j]){
                        a[i][j]=true;
                    }
                }
            }
        }
 */
```

4.6 多重匹配

```
/*
    二分图多重匹配即二分图中一个点可以和多条边相关联，Li 表示点 i 最多可以和多少条边相关联。
    二分图多重匹配可分为最大匹配与最优匹配两种，分别可以用最大流与最大费用最大流解决。
    1. 二分图多重最大匹配：
        建立源点 S 和汇点 T，S 向 X 连边，Y 向 T 连边，容量为 Li，原二分图中各边保留，容量为 1
        (若该边可以使用多次则容量大于 1)，答案即该网络的最大流
    2. 二分图多重最优匹配：
        建立源点 S 和汇点 T，S 向 X 连边，Y 向 T 连边，容量为 Li，费用为 0，原来二分图中各边保留，容量为 1
        (若该边可以使用多次则容量大于 1)，费用为该边的权值，答案即该网络的最大费用最大流
*/
```

5 动态规划

5.1 子序列/子串

5.1.1 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;
}
```

5.1.2 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];
    }
}
```

5.2 背包

5.3 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);
}
```

5.4 区间 dp

5.4.1 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];
}
```

5.5 其他

5.5.1 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;
            }
        }
    }
}
```

6 基础数论

6.1 素数

6.1.1 O(logn) 素数判定

```
/* O(logN) 判定单个素数
 * 最好都开 long long
 * for 循环里 i*i 可能会爆 int 导致 TLE(天梯赛 L1-028)
 */
bool check(ll n){
    if(n==1){
        return false;
    }
```

```
    for(ll i=2;i*i<=n;i++){
        if(n%i==0){
            return false;
        }
    }
    return true;
}
```

6.2 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;
}
```

6.3 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;
}
```

6.4 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);
}
```

6.5 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;
}
```

6.6 中国剩余定理 (CRT)

6.7 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);
}
```

6.8 BSGS

```
/*
 * 大步小步算法用于解决：已知 A, B, C, 求 x 使得 A^x=B(mod C) 成立
 * 令 x = im - j | m = ceil(sqrt(C)),  i = [1, m],  j = [0, m]
 * 原式变成  A^(im) = A^j * B
 * 枚举 j, 把 A^j*B 加入哈希表, 再枚举 i, 在表中查找 A^(i*m), 复杂度为 O(n^0.5)
 */
#include <bits/stdc++.h>
```

```
typedef long long ll;
using namespace std;
ll p, a, b, X1, t, T;
ll pow(ll a, ll b, ll p) {
    ll ans = 1;
    while(b) {
        if(b & 1) ans = ans * a % p;
        b >>= 1;
        a = a * a % p;
    }
    return ans;
}
ll inv(ll a, ll p) {
    return pow(a, p-2, p);
}
map<ll, ll> mp;
ll BSGS(ll A, ll B, ll C) {
    mp.clear();
    if(A % C == 0) return -2;
    ll m = ceil(sqrt(C));
    ll ans;
    for(int i = 0; i <= m; i++) {
        if(i == 0) {
            ans = B % C;
            mp[ans] = i;
            continue;
        }
        ans = (ans * A) % C;
        mp[ans] = i;
    }
    ll t = pow(A, m, C);
    ans = t;
    for(int i = 1; i <= m; i++) {
        if(i != 1)ans = ans * t % C;
        if(mp.count(ans)) {
            int ret = i * m % C - mp[ans] % C;
            return (ret % C + C)%C;
        }
    }
    return -2;
}
int main() {
    scanf("%lld", &T);
    while(T--) {
        scanf("%lld %lld %lld %lld %lld", &p, &a, &b, &X1, &t);
        if(X1 == t) {
            printf("%d\n", 1);
            continue;
        }
        if(a == 0) {
            if(t == b) {
                printf("%d\n", 2);
            }
            else printf("%d\n", -1);
            continue;
        }
```

```
        if(a == 1) {
            if(b == 0) {
                printf("%d\n", -1);
                continue;
            }
            ll ans = (((t-X1)%p + p)%p * inv(b, p)) % p;
            printf("%lld\n", ans+1);
            continue;
        }
        X1 %= p, a %= p, b %= p, t%= p;
        ll tmp = (b%p * inv(a-1, p))%p;
        ll B = ((t+tmp)%p * inv((X1+tmp) % p, p)) % p;
        ll A = a;
        ll ans = BSGS(A, B, p);
        printf("%lld\n", ans+1);
    }
    return 0;
}
```

6.9 小技巧

6.9.1 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));
}
```

7 博弈

7.1 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;
        }
    }
}
```

7.2 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);
}
```

7.3 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;
            }
        }
    }
}
```

7.4 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;
                }
            }
        }
    }
}
```

7.5 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;
}
```

8 计算几何

8.1 注意点

```
/*
* 1. 注意舍入方式 (0.5 的舍入方向); 防止输出-0.
* 2. 几何题注意多测试不对称数据.
* 3. 整数几何注意 xmult 和 dmult 是否会出界;
*    符点几何注意 eps 的使用.
* 4. 避免使用斜率; 注意除数是否会为 0.
* 5. 公式一定要化简后再代入.
* 6. 判断同一个 2*PI 域内两角度差应该是
*    abs(a1-a2)<beta||abs(a1-a2)>pi+pi-beta;
*    相等应该是
*    abs(a1-a2)<eps||abs(a1-a2)>pi+pi-eps;
* 7. 需要的话尽量使用 atan2, 注意:atan2(0,0)=0,
*    atan2(1,0)=pi/2,atan2(-1,0)=-pi/2,atan2(0,1)=0,atan2(0,-1)=pi.
* 8. cross product = |u|*|v|*sin(a)
*    dot product = |u|*|v|*cos(a)
* 9. (P1-P0)x(P2-P0) 结果的意义:
*    正: <P0,P1> 在 <P0,P2> 顺时针 (0,pi) 内
*    负: <P0,P1> 在 <P0,P2> 逆时针 (0,pi) 内
*    0 : <P0,P1>,<P0,P2> 共线, 夹角为 0 或 pi
```

```
* 10. 误差限缺省使用 1e-8!
*/
```

8.2 公式

```
/*
* 三角形:
* 1. 半周长 P=(a+b+c)/2
* 2. 面积 S=aHa/2=absin(C)/2=sqrt(P(P-a)(P-b)(P-c))
* 3. 中线 Ma=sqrt(2(b^2+c^2)-a^2)/2=sqrt(b^2+c^2+2bccos(A))/2
* 4. 角平分线 Ta=sqrt(bc((b+c)^2-a^2))/(b+c)=2bccos(A/2)/(b+c)
* 5. 高线 Ha=bsin(C)=csin(B)=sqrt(b^2-((a^2+b^2-c^2)/(2a))^2)
* 6. 内切圆半径 r=S/P=asin(B/2)sin(C/2)/sin((B+C)/2)
*                  =4Rsin(A/2)sin(B/2)sin(C/2)=sqrt((P-a)(P-b)(P-c)/P)
*                  =Ptan(A/2)tan(B/2)tan(C/2)
* 7. 外接圆半径 R=abc/(4S)=a/(2sin(A))=b/(2sin(B))=c/(2sin(C))
*
* 四边形:
* D1,D2 为对角线,M 对角线中点连线,A 为对角线夹角
* 1. a^2+b^2+c^2+d^2=D1^2+D2^2+4M^2
* 2. S=D1D2sin(A)/2
* (以下对圆的内接四边形)
* 3. ac+bd=D1D2
* 4. S=sqrt((P-a)(P-b)(P-c)(P-d)),P 为半周长
*
* 正 n 边形:
* R 为外接圆半径,r 为内切圆半径
* 1. 中心角 A=2PI/n
* 2. 内角 C=(n-2)PI/n
* 3. 边长 a=2sqrt(R^2-r^2)=2Rsin(A/2)=2rtan(A/2)
* 4. 面积 S=nar/2=nr^2tan(A/2)=nR^2sin(A)/2=na^2/(4tan(A/2))
*
* 圆:
* 1. 弧长 l=rA
* 2. 弦长 a=2sqrt(2hr-h^2)=2rsin(A/2)
* 3. 弓形高 h=r-sqrt(r^2-a^2/4)=r(1-cos(A/2))=atan(A/4)/2
* 4. 扇形面积 S1=rl/2=r^2A/2
* 5. 弓形面积 S2=(rl-a(r-h))/2=r^2(A-sin(A))/2
*
* 棱柱:
* 1. 体积 V=Ah,A 为底面积,h 为高
* 2. 侧面积 S=lp,l 为棱长,p 为直截面周长
* 3. 全面积 T=S+2A
*
* 棱锥:
* 1. 体积 V=Ah/3,A 为底面积,h 为高
* (以下对正棱锥)
* 2. 侧面积 S=lp/2,l 为斜高,p 为底面周长
* 3. 全面积 T=S+A
*
* 棱台:
* 1. 体积 V=(A1+A2+sqrt(A1A2))h/3,A1.A2 为上下底面积,h 为高
* (以下为正棱台)
* 2. 侧面积 S=(p1+p2)l/2,p1.p2 为上下底面周长,l 为斜高
* 3. 全面积 T=S+A1+A2
*
```

```
* 圆柱:
* 1. 侧面积 S=2PIrh
* 2. 全面积 T=2PIr(h+r)
* 3. 体积 V=PIr^2h
*
* 圆锥:
* 1. 母线 l=sqrt(h^2+r^2)
* 2. 侧面积 S=PIrl
* 3. 全面积 T=PIr(l+r)
* 4. 体积 V=PIr^2h/3
*
* 圆台:
* 1. 母线 l=sqrt(h^2+(r1-r2)^2)
* 2. 侧面积 S=PI(r1+r2)l
* 3. 全面积 T=PIr1(l+r1)+PIr2(l+r2)
* 4. 体积 V=PI(r1^2+r2^2+r1r2)h/3
*
* 球:
* 1. 全面积 T=4PIr^2
* 2. 体积 V=4PIr^3/3
*
* 球台:
* 1. 侧面积 S=2PIrh
* 2. 全面积 T=PI(2rh+r1^2+r2^2)
* 3. 体积 V=PIh(3(r1^2+r2^2)+h^2)/6
*
* 球扇形:
* 1. 全面积 T=PIr(2h+r0),h 为球冠高,r0 为球冠底面半径
* 2. 体积 V=2PIr^2h/3
*/
```

8.3 模板 1

```
#include <bits/stdc++.h>
using namespace std;
typedef long long int LL ;
const int N = 100000+7;
#define abs(x)  (((x)>0)?(x):-(x))
/*****************************************/
const double PI  = acos(-1.0);
const double eps = 1e-8;
const double INF = 1e18;
#define pb push_back
#define mp make_pair
///************* 基础 ***********/
double torad(double deg) { return deg / 180 * PI; }
inline int dcmp(double x){
    if(fabs(x) < eps) return 0;
    else return x < 0 ? -1 : 1;
}
struct Point{
    double x, y;
    Point(double x=0, double y=0):x(x),y(y) { }
    inline void read(){
        scanf("%lf%lf", &x, &y);
    }
```

```
};
typedef vector<Point> Polygon;
typedef Point Vector;
inline Vector operator+ (Vector A, Vector B) { return Vector(A.x + B.x, A.y + B.y); }
inline Vector operator- (Point A, Point B) { return Vector(A.x - B.x, A.y - B.y); }
inline Vector operator* (Vector A, double p) { return Vector(A.x * p, A.y * p); }
inline Vector operator/ (Vector A, double p) { return Vector(A.x / p, A.y / p); }
inline bool operator < (Point a, Point b) { return a.x < b.x || (a.x == b.x && a.y < b.y); }
inline bool operator == (Point a, Point b) { return dcmp(a.x - b.x) == 0 && dcmp(a.y - b.y) == 0; }
inline double Dot(Vector A, Vector B){ return A.x * B.x + A.y * B.y;}
inline double Length(Vector A) { return sqrt(Dot(A, A)); }
inline double Angle(Vector A, Vector B) { return acos(Dot(A, B) / Length(A) / Length(B)); }
inline double angle(Vector v) { return atan2(v.y, v.x); }
inline double Cross(Vector A, Vector B) { return A.x * B.y - A.y * B.x; }
inline Vector Unit(Vector x) { return x / Length(x); }  //单位向量
inline Vector Normal(Vector x) { return Point(-x.y, x.x) / Length(x); } //垂直法向量
inline double Area2(Point A, Point B, Point C) { return Cross(B - A, C - A); }
inline Vector Rotate(Vector A, double rad){
    return Vector(A.x * cos(rad) - A.y*sin(rad), A.x*sin(rad) + A.y * cos(rad));
}
/**************** 直线与线段 **************/
//求直线 p+tv 和 q+tw 的交点 Cross(v, w) == 0 无交点
Point GetLineIntersection(Point p, Vector v, Point q, Vector w){
    Vector u = p - q;
    double t = Cross(w, u) / Cross(v, w);
    return p + v * t;
}
//点 p 在直线 ab 的投影
inline Point GetLineProjection(Point P, Point A, Point B){
    Vector v = B - A;
    return A + v * (Dot(v, P - A) / Dot(v, v));
}
//点到直线距离
inline double DistanceToLine(Point P, Point A, Point B){
    Vector v1 = B - A, v2 = P - A;
    return fabs(Cross(v1, v2)) / Length(v1); // 如果不取绝对值，得到的是有向距离
}
//点在 p 线段上 (包括端点)
inline bool OnSegment(Point p, Point a1, Point a2){
    return dcmp(Cross(a1-p, a2-p)) == 0 && dcmp(Dot(a1-p, a2-p)) <= 0;
}
// 过两点 p1, p2 的直线一般方程 ax+by+c=0
// (x2-x1)(y-y1) = (y2-y1)(x-x1)
inline void getLineGeneralEquation(Point p1, Point p2, double& a, double& b, double &c){
    a = p2.y - p1.y;
    b = p1.x - p2.x;
    c = -a * p1.x - b * p1.y;
}
//点到线段距离
double DistanceToSegment(Point p, Point a, Point b){
    if(a == b) return Length(p - a);
    Vector v1 = b - a, v2 = p - a, v3 = p - b;
    if(dcmp(Dot(v1, v2)) < 0) return Length(v2);
    else if(dcmp(Dot(v1, v3)) > 0) return Length(v3);
    else return fabs(Cross(v1, v2)) / Length(v1);
}
```

```
//两线段最近距离
inline double dis_pair_seg(Point p1, Point p2, Point p3, Point p4){
    return min(min(DistanceToSegment(p1, p3, p4), DistanceToSegment(p2, p3, p4)),
           min(DistanceToSegment(p3, p1, p2), DistanceToSegment(p4, p1, p2)));
}
//线段相交判定
inline bool SegmentProperIntersection(Point a1, Point a2, Point b1, Point b2){
    double c1 = Cross(a2 - a1, b1 - a1), c2 = Cross(a2 - a1, b2 - a1),
           c3 = Cross(b2 - b1, a1 - b1), c4 = Cross(b2 - b1, a2 - b1);
    return dcmp(c1) * dcmp(c2) < 0 && dcmp(c3) * dcmp(c4) < 0;
}
// 有向直线。它的左边就是对应的半平面
struct Line{
    Point p, q;    // 直线上任意一点,p 作为起点
    Vector v;   // 方向向量
    double ang; // 极角, 即从 x 正半轴旋转到向量 v 所需要的角 (弧度)
    Line() {}
//    Line(Point P, Vector v):p(P),v(v)
//    {
//        ang = atan2(v.y, v.x);
//    }
    Line(Point P, Point Q):p(P), q(Q){
        v = q - p;
        ang = atan2(v.y, v.x);
    }
    inline bool operator < (const Line& L) const{
        return ang < L.ang;
    }
    inline Point point(double t){
        return p + v * t;
    }
    inline Line move(double d){
        return Line(p + Normal(v) * d, v);
    }
    inline void read(){
        Point q;
        p.read(), q.read();
        v = q - p;
        ang = atan2(v.y, v.x);
    }
};
//两直线交点
inline Point GetLineIntersection(Line a, Line b){
    return GetLineIntersection(a.p, a.v, b.p, b.v);
}
// 点 p 在有向直线 L 的左边 (线上不算)
inline bool OnLeft(const Line& L, const Point& p){
    return Cross(L.v, p - L.p) > 0;
}
//// 二直线交点, 假定交点惟一存在
//Point GetLineIntersection(const Line& a, const Line& b) {
//  Vector u = a.P-b.P;
//  double t = Cross(b.v, u) / Cross(a.v, b.v);
//  return a.P+a.v*t;
//}
// 半平面交主过程
```

```
vector<Point> HalfplaneIntersection(vector<Line> L){
    int n = L.size();
    sort(L.begin(), L.end()); // 按极角排序
    int first, last;         // 双端队列的第一个元素和最后一个元素的下标
    vector<Point> p(n);      // p[i] 为 q[i] 和 q[i+1] 的交点
    vector<Line> q(n);       // 双端队列
    vector<Point> ans;       // 结果
    q[first=last=0] = L[0];  // 双端队列初始化为只有一个半平面 L[0]
    for(int i = 1; i < n; i++){
        while(first < last && !OnLeft(L[i], p[last-1])) last--;
        while(first < last && !OnLeft(L[i], p[first])) first++;
        q[++last] = L[i];
        if(fabs(Cross(q[last].v, q[last-1].v)) < eps) {// 两向量平行且同向，取内侧的一个
            last--;
            if(OnLeft(q[last], L[i].p)) q[last] = L[i];
        }
        if(first < last) p[last-1] = GetLineIntersection(q[last-1], q[last]);
    }
    while(first < last && !OnLeft(q[first], p[last-1])) last--; // 删除无用平面
    if(last - first <= 1) return ans; // 空集
    p[last] = GetLineIntersection(q[last], q[first]); // 计算首尾两个半平面的交点
    // 从 deque 复制到输出中
    for(int i = first; i <= last; i++) ans.push_back(p[i]);
    return ans;
}
/*********** 多边形 **************/
double PolygonArea(vector<Point> p) {
  int n = p.size();
  double area = 0;
  for(int i = 1; i < n - 1; i++)
    area += Cross(p[i]-p[0], p[i+1]-p[0]);
  return area / 2;
}
//判断点是否在多边形内
int isPointInPolygon(Point p, Polygon poly){
    int wn = 0;
    int n = poly.size();
    for (int i = 0; i < n; i++){
        if (OnSegment(p, poly[i], poly[(i + 1) % n])) return -1;    //边界
        int k = dcmp(Cross(poly[(i + 1) % n] - poly[i], p - poly[i]));
        int d1 = dcmp(poly[i].y - p.y);
        int d2 = dcmp(poly[(i + 1) % n].y - p.y);
        if (k > 0 && d1 <= 0 && d2 > 0) wn++;
        if (k < 0 && d2 <= 0 && d1 > 0) wn--;
    }
    if (wn != 0) return 1;  //内部
    return 0;   //外部
}
//多边形重心 点集逆时针给出
Point PolyGravity(Point *p, int n) {
    Point tmp, g = Point(0, 0);
    double sumArea = 0, area;
    for (int i=2; i<n; ++i) {
        area = Cross(p[i-1]-p[0], p[i]-p[0]);
        sumArea += area;
        tmp.x = p[0].x + p[i-1].x + p[i].x;
```

```
        tmp.y = p[0].y + p[i-1].y + p[i].y;
        g.x += tmp.x * area;
        g.y += tmp.y * area;
    }
    g.x /= (sumArea * 3.0); g.y /= (sumArea * 3.0);
    return g;
}
//多边形重心计算模板
Point bcenter(vector<Point> pnt){
    int n = pnt.size();
    Point p, s;
    double tp, area = 0, tpx = 0, tpy = 0;
    p.x = pnt[0].x;
    p.y = pnt[0].y;
    //FE(i, 1, n)
    for(int i=1;i<=n;i++){
        s.x = pnt[(i == n) ? 0 : i].x;
        s.y = pnt[(i == n) ? 0 : i].y;
        tp = (p.x * s.y - s.x * p.y);
        area += tp / 2;
        tpx += (p.x + s.x) * tp;
        tpy += (p.y + s.y) * tp;
        p.x = s.x;
        p.y = s.y;
    }
    s.x = tpx / (6 * area);
    s.y = tpy / (6 * area);
    return s;
}
// 点集凸包
// 如果希望在凸包的边上有输入点，把两个 <= 改成 <
// 注意：输入点集会被修改
vector<Point> ConvexHull(vector<Point>& p){
    // 预处理，删除重复点
    sort(p.begin(), p.end());
    p.erase(unique(p.begin(), p.end()), p.end());
    int n = p.size();
    int m = 0;
    vector<Point> ch(n+1);
    for(int i = 0; i < n; i++){
        while(m > 1 && Cross(ch[m-1]-ch[m-2], p[i]-ch[m-2]) <= 0) m--;
        ch[m++] = p[i];
    }
    int k = m;
    for(int i = n-2; i >= 0; i--){
        while(m > k && Cross(ch[m-1]-ch[m-2], p[i]-ch[m-2]) <= 0) m--;
        ch[m++] = p[i];
    }
    if(n > 1) m--;
    ch.resize(m);
    return ch;
}
inline double Dist2(Point a, Point b){
    return sqrt(a.x - b.x) + sqrt(a.y - b.y);
}
// 返回点集直径的平方
```

```
double diameter2(vector<Point>& points){
    vector<Point> p = ConvexHull(points);
    int n = p.size();
    if(n == 1) return 0;
    if(n == 2) return Dist2(p[0], p[1]);
    p.push_back(p[0]); // 免得取模
    double ans = 0;
    for(int u = 0, v = 1; u < n; u++){
        // 一条直线贴住边 p[u]-p[u+1]
        for(;;){
            // 当 Area(p[u], p[u+1], p[v+1]) <= Area(p[u], p[u+1], p[v]) 时停止旋转
            // 即 Cross(p[u+1]-p[u], p[v+1]-p[u]) - Cross(p[u+1]-p[u], p[v]-p[u]) <= 0
            // 根据 Cross(A,B) - Cross(A,C) = Cross(A,B-C)
            // 化简得 Cross(p[u+1]-p[u], p[v+1]-p[v]) <= 0
            int diff = Cross(p[u+1]-p[u], p[v+1]-p[v]);
            if(diff <= 0){
                ans = max(ans, Dist2(p[u], p[v])); // u 和 v 是对踵点
                if(diff == 0)
                    ans = max(ans, Dist2(p[u], p[v+1])); // diff == 0 时 u 和 v+1 也是对踵点
                break;
            }
            v = (v + 1) % n;
        }
    }
    return ans;
}
//两凸包最近距离
double RC_Distance(Point *ch1, Point *ch2, int n, int m){
    int q=0, p=0;
    for(int i=1;i<=n;i++)
    if(ch1[i].y-ch1[p].y < -eps) p=i;
    for(int i=1;i<=m;i++)
    if(ch2[i].y-ch2[q].y > eps) q=i;
    ch1[n]=ch1[0];
    ch2[m]=ch2[0];
    double tmp, ans=1e100;
    for(int i=1;i<=n;i++){
        while((tmp=Cross(ch1[p+1]-ch1[p],ch2[q+1]-ch1[p])-Cross(ch1[p+1]-ch1[p],ch2[q]-ch1[p]))>eps)
            q=(q+1)%m;
        if(tmp < -eps) ans = min(ans,DistanceToSegment(ch2[q],ch1[p],ch1[p+1]));
        else ans = min(ans,dis_pair_seg(ch1[p],ch1[p+1],ch2[q],ch2[q+1]));
        p=(p+1)%n;
    }
    return ans;
}
//两凸包最近距离
//使用 vector
double RC_Distance(vector<Point> ch1, vector<Point> ch2){
    int q = 0, p = 0, n = ch1.size(), m = ch2.size();
    for(int i=1;i<=n;i++)
    if(ch1[i].y-ch1[p].y < -eps) p=i;
    for(int i=1;i<=m;i++)
    if(ch2[i].y-ch2[q].y > eps) q=i;
    ch1.push_back(ch1[0]), ch2.push_back(ch2[0]);
    double tmp, ans=1e100;
    for(int i=1;i<=n;i++){
```

```
        while((tmp=Cross(ch1[p+1]-ch1[p],ch2[q+1]-ch1[p])-Cross(ch1[p+1]-ch1[p],ch2[q]-ch1[p]))>eps)
            q=(q+1)%m;
        if(tmp < -eps) ans = min(ans,DistanceToSegment(ch2[q],ch1[p],ch1[p+1]));
        else ans = min(ans,dis_pair_seg(ch1[p],ch1[p+1],ch2[q],ch2[q+1]));
        p=(p+1)%n;
    }
    return ans;
}
//凸包最大内接三角形
double RC_Triangle(Point* res,int n){
    if(n < 3)    return 0;
    double ans = 0, tmp;
    res[n] = res[0];
    int j, k;
    //REP(i, n)
    for(int i=1;i<=n;i++){
        j = (i+1)%n;
        k = (j+1)%n;
        while((j != k) && (k != i)){
            while(Cross(res[j] - res[i], res[k+1] - res[i]) > Cross(res[j] - res[i], res[k] - res[i])){
                k= (k+1)%n;
            }
            tmp = Cross(res[j] - res[i], res[k] - res[i]);
            if(tmp > ans) ans = tmp;
            j = (j+1)%n;
        }
    }
    return ans;
}
//凸包最大内接三角形
double RC_Triangle(vector<Point> res, Point& a, Point& b, Point& c)
{
    int n = res.size();
    if(n < 3)    return 0;
    double ans=0, tmp;
    res.push_back(res[0]);
    int j, k;
    //REP(i, n)
    for(int i=1;i<=n;i++){
        j = (i+1)%n;
        k = (j+1)%n;
        while((j != k) && (k != i)){
            while(Cross(res[j] - res[i], res[k+1] - res[i]) > Cross(res[j] - res[i], res[k] - res[i])){
                k= (k+1)%n;
            }
            tmp = Cross(res[j] - res[i], res[k] - res[i]);
            if(tmp > ans){
                a = res[i], b = res[j], c = res[k];
                ans = tmp;
            }
            j = (j+1)%n;
        }
    }
    return ans;
}
//判断两凸包是否有交点
```

```
bool ConvexPolygonDisjoint(const vector<Point> ch1, const vector<Point> ch2){
    int c1 = ch1.size();
    int c2 = ch2.size();
    for(int i = 0; i < c1; i++)
        if(isPointInPolygon(ch1[i], ch2) != 0)
            return false; // 内部或边界上
    for(int i = 0; i < c2; i++)
        if(isPointInPolygon(ch2[i], ch1) != 0)
            return false; // 内部或边界上
    for(int i = 0; i < c1; i++)
        for(int j = 0; j < c2; j++)
            if(SegmentProperIntersection(ch1[i], ch1[(i+1)%c1], ch2[j], ch2[(j+1)%c2]))
                return false;
    return true;
}
inline double dist(Point a, Point b){
    return Length(a - b);
}
////模拟退火求费马点 保存在 ptres 中
//double fermat_point(Point *pt, int n, Point& ptres)
//{
//    Point u, v;
//    double step = 0.0, curlen, explen, minlen;
//    int i, j, k;
//    bool flag;
//    u.x = u.y = v.x = v.y = 0.0;
//    //REP(i, n)
//    for(int i=1;i<=n;i++)
//    {
//        step += fabs(pt[i].x) + fabs(pt[i].y);
//        u.x += pt[i].x;
//        u.y += pt[i].y;
//    }
//    u.x /= n;
//    u.y /= n;
//    flag = 0;
//    while(step > eps)
//    {
//        for(k = 0; k < 10; step /= 2, ++k)
//            for(i = -1; i <= 1; ++i)
//                for(j = -1; j <= 1; ++j)
//                {
//                    v.x = u.x + step*i;
//                    v.y = u.y + step*j;
//                    curlen = explen = 0.0;
//                    //REP(i, n)
//                    for(int i=1;i<=n;i++)
//                    {
//                        curlen += dist(u, pt[idx]);
//                        explen += dist(v, pt[idx]);
//                    }
//                    if(curlen > explen)
//                    {
//                        u = v;
//                        minlen = explen;
//                        flag = 1;
```

```
//                      }
//                  }
//    }
//    ptres = u;
//    return flag ? minlen : curlen;
//}
//多边形费马点
//到所有顶点的距离和最小
Point Fermat(int np, Point* p){
    double nowx = 0, nowy = 0;
    double nextx = 0, nexty = 0;
    //REP(i, np)
    for(int i=1;i<=np;i++)
    {
        nowx += p[i].x;
        nowy += p[i].y;
    }
    for (nowx /= np, nowy /= np;; nowx = nextx, nowy = nexty)
    {
        double topx = 0, topy = 0, bot = 0;
        //REP(i, np)
        for(int i=1;i<=np;i++)
        {
            double d = sqrt(sqrt(nowx - p[i].x) + sqrt(nowy - p[i].y));
            topx += p[i].x / d;
            topy += p[i].y / d;
            bot += 1 / d;
        }
        nextx = topx / bot;
        nexty = topy / bot;
        if (dcmp(nextx - nowx) == 0 && dcmp(nexty - nowy) == 0)
            break;
    }
    Point fp;
    fp.x = nowx;
    fp.y = nowy;
    return fp;
}
//最近点对
//使用前先对输入的 point 进行排序, 使用 cmpxy 函数
Point point[N];
int tmpt[N];
inline double dist(int x, int y){
    Point& a = point[x];
    Point& b = point[y];
    return sqrt(sqrt(a.x - b.x) + sqrt(a.y - b.y));
}
inline bool cmpxy(Point a, Point b){
    if(a.x != b.x)
        return a.x < b.x;
    return a.y < b.y;
}
inline bool cmpy(int a, int b){
    return point[a].y < point[b].y;
}
double Closest_Pair(int left, int right){
```

```
    double d = INF;
    if(left==right)
        return d;
    if(left + 1 == right)
        return dist(left, right);
    int mid = (left+right)>>1;
    double d1 = Closest_Pair(left,mid);
    double d2 = Closest_Pair(mid+1,right);
    d = min(d1,d2);
    int k=0;
    //分离出宽度为 d 的区间
    //FE(i, left, right)
    for(int i=left;i<=right;i++){
        if(fabs(point[mid].x-point[i].x) <= d)
            tmpt[k++] = i;
    }
    sort(tmpt,tmpt+k,cmpy);
    //线性扫描
    //REP(i, k)
    for(int i=0;i<k;i++){
        for(int j = i+1; j < k && point[tmpt[j]].y-point[tmpt[i]].y<d; j++){
            double d3 = dist(tmpt[i],tmpt[j]);
            if(d > d3)
                d = d3;
        }
    }
    return d;
}
/************ 圆 ************/
struct Circle{
    Point c;
    double r;
    Circle() {}
    Circle(Point c, double r):c(c), r(r) {}
    inline Point point(double a) {//根据圆心角求点坐标
        return Point(c.x+cos(a)*r, c.y+sin(a)*r);
    }
    inline void read(){
        scanf("%lf%lf%lf", &c.x, &c.y, &r);
    }
};

//求 a 点到 b 点 (逆时针) 在的圆上的圆弧长度
double DisOnCircle(Point a, Point b, Circle C){
    double ang1 = angle(a - C.c);
    double ang2 = angle(b - C.c);
    if (ang2 < ang1) ang2 += 2 * PI;
    return C.r * (ang2 - ang1);
}

//直线与圆交点 返回个数
int getLineCircleIntersection(Line L, Circle C, double& t1, double& t2, vector<Point>& sol){
    double a = L.v.x, b = L.p.x - C.c.x, c = L.v.y, d = L.p.y - C.c.y;
    double e = a*a + c*c, f = 2*(a*b + c*d), g = b*b + d*d - C.r*C.r;
    double delta = f*f - 4*e*g; // 判别式
    if(dcmp(delta) < 0) return 0; // 相离
```

```
    if(dcmp(delta) == 0){
        // 相切
        t1 = t2 = -f / (2 * e);
        sol.push_back(L.point(t1));
        return 1;
    }
    // 相交
    t1 = (-f - sqrt(delta)) / (2 * e);
    sol.push_back(L.point(t1));
    t2 = (-f + sqrt(delta)) / (2 * e);
    sol.push_back(L.point(t2));
    return 2;
}
//两圆交点 返回个数
int getCircleCircleIntersection(Circle C1, Circle C2, vector<Point>& sol){
    double d = Length(C1.c - C2.c);
    if(dcmp(d) == 0){
        if(dcmp(C1.r - C2.r) == 0) return -1; // 重合, 无穷多交点
        return 0;
    }
    if(dcmp(C1.r + C2.r - d) < 0) return 0;
    if(dcmp(fabs(C1.r-C2.r) - d) > 0) return 0;
    double a = angle(C2.c - C1.c);
    double da = acos((C1.r*C1.r + d*d - C2.r*C2.r) / (2*C1.r*d));
    Point p1 = C1.point(a-da), p2 = C1.point(a+da);
    sol.push_back(p1);
    if(p1 == p2) return 1;
    sol.push_back(p2);
    return 2;
}
// 过点 p 到圆 C 的切线。v[i] 是第 i 条切线的向量。返回切线条数
int getTangents(Point p, Circle C, Vector* v){
    Vector u = C.c - p;
    double dist = Length(u);
    if(dist < C.r) return 0;
    else if(dcmp(dist - C.r) == 0){ // p 在圆上, 只有一条切线
        v[0] = Rotate(u, PI/2);
        return 1;
    }else{
        double ang = asin(C.r / dist);
        v[0] = Rotate(u, -ang);
        v[1] = Rotate(u, +ang);
        return 2;
    }
}
//两圆的公切线, -1 表示无穷条切线
//返回切线的条数, -1 表示无穷条切线
//a[i] 和 b[i] 分别是第 i 条切线在圆 A 和圆 B 上的切点
int getTangents(Circle A, Circle B, Point* a, Point* b){
    int cnt = 0;
    if (A.r < B.r) swap(A, B), swap(a, b);
    int d2 = (A.c.x - B.c.x) * (A.c.x - B.c.x) + (A.c.y - B.c.y) * (A.c.y - B.c.y);
    int rdiff = A.r - B.r;
    int rsum = A.r + B.r;
    if (d2 < rdiff * rdiff) return 0;   //内含
    double base = atan2(B.c.y - A.c.y, B.c.x - A.c.x);
```

```
    if (d2 == 0 && A.r == B.r) return -1;    //无线多条切线
    if (d2 == rdiff * rdiff) {//内切, 1 条切线
        a[cnt] = A.point(base); b[cnt] = B.point(base); cnt++;
        return 1;
    }
    //有外公切线
    double ang = acos((A.r - B.r) / sqrt(d2 * 1.0));
    a[cnt] = A.point(base + ang); b[cnt] = B.point(base + ang); cnt++;
    a[cnt] = A.point(base - ang); b[cnt] = B.point(base - ang); cnt++;
    if (d2 == rsum * rsum) {//一条内公切线
        a[cnt] = A.point(base); b[cnt] = B.point(PI + base); cnt++;
    }else if (d2 > rsum * rsum) {//两条内公切线
        double ang = acos((A.r + B.r) / sqrt(d2 * 1.0));
        a[cnt] = A.point(base + ang); b[cnt] = B.point(PI + base + ang); cnt++;
        a[cnt] = A.point(base - ang); b[cnt] = B.point(PI + base - ang); cnt++;
    }
    return cnt;
}
// 过点 p 到圆 C 的切点
int getTangentPoints(Point p, Circle C, vector<Point>& v){
    Vector u = C.c - p;
    double dist = Length(u);
    if(dist < C.r) return 0;
    else if(dcmp(dist - C.r) == 0){// p 在圆上, 只有一条切线
        v.push_back(p);
        return 1;
    }else{
        double ang = asin(C.r / dist);
        double d = sqrt(dist * dist - C.r * C.r);
        v.push_back(p + Unit(Rotate(u, -ang)) * d);
        v.push_back(p + Unit(Rotate(u, +ang)) * d);
        return 2;
    }
}
//圆 A 与圆 B 的切点
void getTangentPoints(Circle A, Circle B, vector<Point>& a){
    if (A.r < B.r) swap(A, B);
    int d2 = sqrt(A.c.x - B.c.x) + sqrt(A.c.y - B.c.y);
    int rdiff = A.r - B.r, rsum = A.r + B.r;
    if (d2 < rdiff * rdiff) return;   //内含
    double base = atan2(B.c.y - A.c.y, B.c.x - A.c.x);
    if (d2 == 0 && A.r == B.r) return;    //无线多条切线
    if (d2 == rdiff * rdiff) {//内切, 1 条切线
        a.push_back(A.point(base));
        a.push_back(B.point(base));
        return;
    }
    //有外公切线
    double ang = acos((A.r - B.r) / sqrt(d2 * 1.0));
    a.push_back(A.point(base + ang)); a.push_back(B.point(base + ang));
    a.push_back(A.point(base - ang)); a.push_back(B.point(base - ang));
    if (d2 == rsum * rsum) {//一条内公切线
        a.push_back(A.point(base));
        a.push_back(B.point(PI + base));
    }else if (d2 > rsum * rsum) { //两条内公切线
        double ang = acos((A.r + B.r) / sqrt(d2 * 1.0));
```

```
        a.push_back(A.point(base + ang));
        a.push_back(B.point(PI + base + ang));
        a.push_back(A.point(base - ang));
        a.push_back(B.point(PI + base - ang));
    }
}
//三角形外接圆
Circle CircumscribedCircle(Point p1, Point p2, Point p3){
    double Bx = p2.x-p1.x, By = p2.y-p1.y;
    double Cx = p3.x-p1.x, Cy = p3.y-p1.y;
    double D = 2*(Bx*Cy-By*Cx);
    double cx = (Cy*(Bx*Bx+By*By) - By*(Cx*Cx+Cy*Cy))/D + p1.x;
    double cy = (Bx*(Cx*Cx+Cy*Cy) - Cx*(Bx*Bx+By*By))/D + p1.y;
    Point p = Point(cx, cy);
    return Circle(p, Length(p1-p));
}
//三角形内切圆
Circle InscribedCircle(Point p1, Point p2, Point p3){
    double a = Length(p2-p3);
    double b = Length(p3-p1);
    double c = Length(p1-p2);
    Point p = (p1*a+p2*b+p3*c)/(a+b+c);
    return Circle(p, DistanceToLine(p, p1, p2));
}
//所有经过点 p 半径为 r 且与直线 L 相切的圆心
vector<Point> CircleThroughPointTangentToLineGivenRadius(Point p, Line L, double r){
    vector<Point> ans;
    double t1, t2;
    getLineCircleIntersection(L.move(-r), Circle(p, r), t1, t2, ans);
    getLineCircleIntersection(L.move(r), Circle(p, r), t1, t2, ans);
    return ans;
}
//半径为 r 与 a b 两直线相切的圆心
vector<Point> CircleTangentToLinesGivenRadius(Line a, Line b, double r){
    vector<Point> ans;
    Line L1 = a.move(-r), L2 = a.move(r);
    Line L3 = b.move(-r), L4 = b.move(r);
    ans.push_back(GetLineIntersection(L1, L3));
    ans.push_back(GetLineIntersection(L1, L4));
    ans.push_back(GetLineIntersection(L2, L3));
    ans.push_back(GetLineIntersection(L2, L4));
    return ans;
}
//与两圆相切 半径为 r 的所有圆心
vector<Point> CircleTangentToTwoDisjointCirclesWithRadius(Circle c1, Circle c2, double r){
    vector<Point> ans;
    Vector v = c2.c - c1.c;
    double dist = Length(v);
    int d = dcmp(dist - c1.r -c2.r - r*2);
    if(d > 0) return ans;
    getCircleCircleIntersection(Circle(c1.c, c1.r+r), Circle(c2.c, c2.r+r), ans);
    return ans;
}
//多边形与圆相交面积
Point GetIntersection(Line a, Line b) {//线段交点
    Vector u = a.p-b.p;
```

```
    double t = Cross(b.v, u) / Cross(a.v, b.v);
    return a.p + a.v*t;
}
inline bool InCircle(Point x, Circle c){
    return dcmp(c.r - Length(c.c - x)) >= 0;
}
inline bool OnCircle(Point x, Circle c){
    return dcmp(c.r - Length(c.c - x)) == 0;
}
//线段与圆的交点
int getSegCircleIntersection(Line L, Circle C, Point* sol){
    Vector nor = Normal(L.v);
    Line pl = Line(C.c, nor);
    Point ip = GetIntersection(pl, L);
    double dis = Length(ip - C.c);
    if (dcmp(dis - C.r) > 0) return 0;
    Point dxy = Unit(L.v) * sqrt(sqrt(C.r) - sqrt(dis));
    int ret = 0;
    sol[ret] = ip + dxy;
    if (OnSegment(sol[ret], L.p, L.point(1))) ret++;
    sol[ret] = ip - dxy;
    if (OnSegment(sol[ret], L.p, L.point(1))) ret++;
    return ret;
}
//线段切割圆
double SegCircleArea(Circle C, Point a, Point b){
    double a1 = angle(a - C.c);
    double a2 = angle(b - C.c);
    double da = fabs(a1 - a2);
    if (da > PI) da = PI * 2.0 - da;
    return dcmp(Cross(b - C.c, a - C.c)) * da * sqrt(C.r) / 2.0;
}
//多边形与圆相交面积
double PolyCiclrArea(Circle C, Point *p, int n){
    double ret = 0.0;
    Point sol[2];
    p[n] = p[0];
    for(int i=1;i<=n;i++){
        int cnt = getSegCircleIntersection(Line(p[i], p[i+1]-p[i]), C, sol);
        if (cnt == 0){
            if (!InCircle(p[i], C) || !InCircle(p[i+1], C)) ret += SegCircleArea(C, p[i], p[i+1]);
            else ret += Cross(p[i+1] - C.c, p[i] - C.c) / 2.0;
        }
        if (cnt == 1){
            if (InCircle(p[i], C) && !InCircle(p[i+1], C)){
                ret+=Cross(sol[0]-C.c,p[i]-C.c)/2.0, ret+=SegCircleArea(C,sol[0],p[i+1]);
            }else{
                ret+=SegCircleArea(C,p[i],sol[0]), ret+=Cross(p[i+1]-C.c,sol[0]-C.c)/2.0;
            }
        }
        if (cnt == 2){
            if ((p[i] < p[i + 1]) ^ (sol[0] < sol[1])) swap(sol[0], sol[1]);
            ret += SegCircleArea(C, p[i], sol[0]);
            ret += Cross(sol[1] - C.c, sol[0] - C.c) / 2.0;
            ret += SegCircleArea(C, sol[1], p[i+1]);
        }
```

```
    }
    return fabs(ret);
}
double area(vector<Point>p) {//计算凸包的面积
    double ans = 0;
    int sz = p.size();
    for (int i = 1; i < sz - 1; i++)
        ans += Cross(p[i] - p[0], p[i + 1] - p[0]);
    return ans / 2.0;
}

double seg(Point o, Point a, Point b){
    if (dcmp(b.x - a.x) == 0) return (o.y - a.y) / (b.y - a.y);
    return (o.x - a.x) / (b.x - a.x);
}
vector<Point> pp[110];
pair<double, int> s[2000200];
double polyunion(vector<Point>*p, int n){//求 n 个多凸包的面积交
    double ret = 0;
    for (int i = 0; i < n; i++){
        int sz = p[i].size();
        for (int j = 0; j < sz; j++){
            int m = 0;
            s[m++] = mp(0, 0);
            s[m++] = mp(1, 0);
            Point a = p[i][j], b = p[i][(j + 1) % sz];
            for (int k = 0; k < n; k++){
                if (i != k){
                    int siz = p[k].size();
                    for (int ii = 0; ii < siz; ii++){
                        Point c = p[k][ii], d = p[k][(ii + 1) % siz];
                        int c1 = dcmp(Cross(b - a, c - a));
                        int c2 = dcmp(Cross(b - a, d - a));
                        if (c1 == 0 && c2 == 0){
                            if (dcmp(Dot(b - a, d - c)) > 0 && i > k){
                                s[m++] = mp(seg(c, a, b), 1);
                                s[m++] = mp(seg(d, a, b), -1);
                            }
                        }else{
                            double s1 = Cross(d - c, a - c);
                            double s2 = Cross(d - c, b - c);
                            if (c1 >= 0 && c2 < 0) s[m++] = mp(s1 / (s1 - s2), 1);
                            else if (c1 < 0 && c2 >= 0) s[m++] = mp(s1 / (s1 - s2), -1);
                        }
                    }
                }
            }
            sort(s, s + m);
            double pre = min(max(s[0].first, 0.0), 1.0), now;
            double sum = 0;
            int cov = s[0].second;
            for (int j = 1; j < m; j++){
                now = min(max(s[j].first, 0.0), 1.0);
                if (!cov) sum += now - pre;
                cov += s[j].second;
                pre = now;
```

```
            }
            ret += Cross(a, b)*sum;
        }
    }
    return ret / 2;
}
int Main(){
    int n, m, kcase = 1;
    double x1,x2,x3,x4,y1,y2,y3,y4;
    while (~scanf("%lf%lf%lf%lf",&x1,&y1,&x2,&y2)){
        scanf("%lf%lf%lf%lf",&x3,&y3,&x4,&y4);

        pp[0].clear();
        pp[1].clear();

        pp[0].pb(Point(x1,y1));
        pp[0].pb(Point(x1,y2));
        pp[0].pb(Point(x2,y1));
        pp[0]=ConvexHull(pp[0]);
//        for(int i=0;i<pp[0].size();i++){
//            printf("%lf  %lf\n",pp[0][i].x,pp[0][i].y);
//        }puts("-----");
        pp[1].pb(Point(x3,y3));
        pp[1].pb(Point(x3,y4));
        pp[1].pb(Point(x4,y3));
        pp[1].pb(Point(x4,y4));
        pp[1]=ConvexHull(pp[1]);
//        for(int i=0;i<pp[1].size();i++){
//            printf("%lf  %lf\n",pp[1][i].x,pp[1][i].y);
//        }puts("-----");

        double t1 = area(pp[0]) + area(pp[1]);
        double t2 = polyunion(pp, 2);
        printf("%.10lf\n",t1-t2);
    }
    return 0;
}

int main(){
#ifndef ONLINE_JUDGE
    freopen("asdf.in" ,"r",stdin );
    freopen("asdf.out","w",stdout);
    double _time_tabris=clock();
#endif // ONLINE_JUDGE
    Main();
#ifndef ONLINE_JUDGE
        printf("time: %lf\n",clock()-_time_tabris);
#endif //ONLINE_JUDGE
    return 0;
}
```

9 区间问题

9.1 线段树

9.2 RMQ

```
/*
 * 预处理 O(nlogn)
 * dp[i][j]: 从 a[i] 开始 2^j 个数的最小值
 * 数组下标从 1 开始, 注意-1 +1 <= 等细节
 */
void init(){
    for(int i=1;i<=n;i++){
        dp[i][0]=a[i];
    }
    for(int j=1;(1<<j)<=n;j++){
        for(int i=1;i+(1<<(j-1))-1<=n;i++){
            dp[i][j]=max(dp[i][j-1],dp[i+(1<<(j-1))][j-1]);
        }
    }
}
/*
 * 查询 [l,r] 最大值
 */
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 max(dp[l][k],dp[r-(1<<k)+1][k]);
}
```

9.3 树状数组

9.3.1 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;
}
```

9.3.2 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;
}
```

9.3.3 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);
}
```

9.3.4 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);
}
```

9.3.5 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;
}
```

9.3.6 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);
}
```

10 其他

10.1 双指针/尺取法

10.1.1 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--;
        }
    }
};
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');
    }
}
```

10.1.2 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++;
            }
        }
    }
}
```

10.2 单调队列/单调栈

10.2.1 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;
}
```

10.2.2 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;
```

10.2.3 作为最大/最小值能延伸的区间

```
#include <bits/stdc++.h>
using namespace std;
const int N=1e6+50;
int h[N],n;
int le[N],ri[N];
int main(void){
    scanf("%d",&n);
    for(int i=1;i<=n;i++){
        scanf("%lld",&h[i]);
    }
    //求出每个数作为最大值/最小值能延伸到的区间
    stack<int> ss;
    for(int i=1;i<=n;i++){
        while(ss.size()>0 && h[i]<=h[ss.top()]){
            ss.pop();
        }
        if(ss.size()>0){
            le[i]=ss.top()+1;
        }else{
            le[i]=1;
        }
        ss.push(i);
    }
    while(!ss.empty()){
        ss.pop();
    }
    for(int i=n;i>=1;i--){
        while(ss.size()>0 && h[i]<=h[ss.top()]){
            ss.pop();
        }
        if(ss.size()>0){
            ri[i]=ss.top()-1;
        }else{
            ri[i]=n;
        }
        ss.push(i);
    }
```

```
    return 0;
}
```

10.3 笛卡尔树

```
/*
 * 笛卡尔树，一种特殊的二叉树
 * key 满足二叉搜索树的特性（下标小的在左边，下标大的在右边）
 * value 满足堆的性质（小根堆：父节点比子节点小，大根堆相反）
 */
int build(){
    for(int i=1;i<=n;i++){
        le[i]=ri[i]=fa[i]=0;
    }
    stack<int> st;
    for(int i=1;i<=n;i++){
        bool flag=false;
        int tmp;
        //维护一个递增的单调栈，构造大根堆
        while(st.size()>0 && a[st.top()]<a[i]){
            tmp=st.top();
            st.pop();
            flag=true;
        }
        //栈中还有元素，那么这个即将入栈的元素 i 就是右子树
        if(st.size()>0){
            ri[st.top()]=i;
            fa[i]=st.top();
        }
        //维护单调性过程中最后一个出栈的就是左子树
        if(flag){
            le[i]=tmp;
            fa[tmp]=i;
        }
        st.push(i);
    }
    for(int i=1;i<=n;i++){
        if(!fa[i]){
            return i;
        }
    }
}
```

10.4 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 j=0; j<n; j++) {
            for(int k=0; k<n; k++) {
                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;
}
```

10.5 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;
}
```

10.6 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);
    }
}
```

10.7 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;
    }
}
```

10.8 技巧

10.8.1 Bitset

10.8.2 快读

10.8.3 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;
    }
}
```

10.9 一些比较有意思的题目

10.9.1 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;

}
```

10.9.2 cf1144E——求字符串中位数 (26 进制模拟)

```
#include <bits/stdc++.h>
using namespace std;
const int N=2e5+50;
char s[N],t[N];
int n,a[N],b[N],c[N];
int main(void){
        scanf("%d",&n);
        scanf("%s",s);
        scanf("%s",t);
        for(int i=0;i<n;i++){
                //后移一位, 可能有进位
                a[i+1]=s[i]-'a';
                b[i+1]=t[i]-'a';
        }
        //加法
        for(int i=n;i>=1;i--){
                c[i]+=(a[i]+b[i]);
                if(c[i]>=26){
                        c[i]-=26;
                        c[i-1]++;
                }
        }
        //除法
        for(int i=0;i<=n;i++){
                int t=c[i]%2;
                if(t){
                        c[i+1]+=26;
```

```
                }
                c[i]=c[i]/2;
        }
        for(int i=1;i<=n;i++){
                printf("%c",(char)('a'+c[i]));
        }
        printf("\n");
        return 0;
}
```

10.9.3 牛客 548B——除法模拟

```
#include <bits/stdc++.h>
using namespace std;
typedef long long ll;
int t;
ll a,b,l,r,c;
ll Pow(ll a,ll n){
    ll ans=1;
    while(n){
        if(n%2){
            ans=ans*a%b;
        }
        a=a*a%b;
        n/=2;
    }
    return ans;
}
int main(){
    scanf("%d",&t);
    while(t--){
        scanf("%lld%lld%lld%lld",&a,&b,&l,&r);
        int i=l;
        c=a*Pow(10,l-1);
        while(i++<=r){
            c%=b;
            c*=10;
            printf("%lld",c/b);
        }
        printf("\n");
    }
}
```

10.10 hdu4507——数位 dp 求平方和

```
#include <bits/stdc++.h>
using namespace std;
typedef long long ll;
const ll MOD=1e9+7;
struct node{
    ll cnt;
    ll sum;
    ll powsum;
}dp[30][10][10];
int a[30];
ll pw[30];
void init(){
```

```
    pw[0]=1;
    for(int i=1;i<30;i++){
        pw[i]=(pw[i-1]*10)%MOD;
    }
    for(int i=0;i<30;i++){
        for(int j=0;j<10;j++){
            for(int k=0;k<10;k++){
                dp[i][j][k]=node{-1,0,0};
            }
        }
    }
}
node dfs(int pos,int summod,int valmod,bool limit){
    if(pos<0){
        node tmp;
        tmp.cnt=(summod!=0 && valmod!=0);
        //只有计数才能在这里返回，其他两个值要在回溯的时候计算
        //换句话说，递归边界只是为了判断是否存在这个合法的数
        tmp.sum=tmp.powsum=0;
        return tmp;
    }
    if(!limit && dp[pos][summod][valmod].cnt!=-1){
        return dp[pos][summod][valmod];
    }
    int up=limit?a[pos]:9;
    node ans;
    ans.cnt=ans.sum=ans.powsum=0;
    for(int i=0;i<=up;i++){
        if(i==7){
            continue;
        }
        node tmp=dfs(pos-1,(summod+i)%7,(valmod*10+i)%7,limit&&(i==up));

        //普通数位 dp 计数
        ans.cnt=(ans.cnt+tmp.cnt)%MOD;
        //sum 计算
        //比如递归到最后一位是 3 4 两个符合的数位，当前位是第 1 位，数位为 2
        //回溯计算就是 3+4=7 -->3+4+23+24=54
        //也就是加上 pw[1]=10 乘以 i=2 乘以递归边界计数 tmp.cnt=2
        ans.sum=(ans.sum+tmp.sum+((pw[pos]*i)%MOD*tmp.cnt)%MOD)%MOD;
        //powsum 计算
        //同上 3^2+4^2=25 --> 3^2+4^2+23^2+24^2
        //也就是加上 (20+3)^2+(20+4)^2=20^2+20^2+3^2+4^2+2*10*(3+4)
        ans.powsum=((ans.powsum+tmp.powsum)%MOD+((((tmp.cnt*pw[pos])
        %MOD*pw[pos])%MOD*i*i)%MOD)+(((tmp.sum*pw[pos])%MOD*2*i)%MOD))%MOD;
    }
    if(!limit){
        dp[pos][summod][valmod]=ans;
    }
    return ans;
}
ll solve(ll x){
    int k=0;
    while(x){
        a[k++]=x%10;
        x/=10;
```

```
    }
    return dfs(k-1,0,0,true).powsum;
}
int t;
ll l,r;
int main(void){
    init();
    scanf("%d",&t);
    while(t--){
        scanf("%lld%lld",&l,&r);
        ll ansr=solve(r)%MOD;
        ll ansl=solve(l-1)%MOD;
        printf("%lld\n",(ansr-ansl+MOD)%MOD);
    }
    return 0;
}
```