ACM 模板

黄佳瑞，钱智煊，林乐逍

2025 年 9 月 4 日

目录

1 做题指导 **3**
- 1.1 上机前你应该注意什么 . . . 3
- 1.2 机上你应该注意什么 . . . 3
- 1.3 交题前你应该注意什么 . . . 3
- 1.4 如果你的代码挂了 . . . 3
- 1.5 另外一些注意事项 . . . 3

2 图论 **4**
- 2.1 Tarjan 边双、点双（圆方树） . . . 4
- 2.2 O(1) LCA . . . 4
- 2.3 某些路径问题单 log 做法 . . . 4
- 2.4 重链剖分 . . . 4
- 2.5 2-SAT . . . 5
- 2.6 Dinic 网络流、费用流 . . . 5
  - 2.6.1 无源汇有上下界可行流 . . . 5
  - 2.6.2 有源汇有上下界最大流 . . . 6
  - 2.6.3 网络流总结 . . . 7
- 2.7 差分约束 . . . 7
- 2.8 欧拉路径 . . . 7
- 2.9 同余最短路 . . . 8
- 2.10 点分治 . . . 8

3 数据结构 **9**
- 3.1 平衡树 . . . 9
  - 3.1.1 FHQ Treap . . . 9
  - 3.1.2 平衡树合并 . . . 9
  - 3.1.3 Splay . . . 10
- 3.2 LCT 动态树 . . . 10
- 3.3 ODT 珂朵莉树 . . . 10
- 3.4 李超线段树 . . . 11
- 3.5 二维树状数组 . . . 11
- 3.6 虚树 . . . 11
- 3.7 左偏树 . . . 11
- 3.8 吉司机线段树 . . . 11
- 3.9 树分治 . . . 12

4 字符串 **13**
- 4.1 Hash 类 . . . 13
- 4.2 后缀数组（llx） . . . 13
- 4.3 后缀数组与后缀树（旧） . . . 13
- 4.4 AC 自动机 . . . 13
- 4.5 回文自动机 . . . 14
- 4.6 Manacher 算法 . . . 14
- 4.7 KMP 算法与 border 理论 . . . 14
- 4.8 Z 函数 . . . 14
- 4.9 后缀自动机 . . . 14
- 4.10 后缀自动机 (map 版) . . . 14
- 4.11 最小表示法 . . . 15

5 线性代数 **15**
- 5.1 高斯消元 . . . 15
- 5.2 线性基 . . . 15
- 5.3 行列式 . . . 15
- 5.4 矩阵树定理 . . . 15
- 5.5 单纯形法 . . . 16
- 5.6 全幺模矩阵 . . . 16
- 5.7 对偶原理 . . . 16

6 多项式 **17**
- 6.1 FFT . . . 17
- 6.2 NTT . . . 17
- 6.3 集合幂级数 . . . 17
  - 6.3.1 并卷积、交卷积与子集卷积 . . . 17
  - 6.3.2 对称差卷积 . . . 17
- 6.4 多项式全家桶 . . . 17

7 数论 **19**
- 7.1 中国剩余定理 . . . 19
- 7.2 扩展中国剩余定理 . . . 19
- 7.3 BSGS . . . 19
- 7.4 扩展 BSGS . . . 19
- 7.5 Lucas 定理 . . . 19
- 7.6 扩展 Lucas 定理 . . . 19
- 7.7 杜教筛 . . . 20
- 7.8 Min-25 筛 . . . 20

8 计算几何 **22**
- 8.1 声明与宏 . . . 22
- 8.2 点与向量 . . . 22
- 8.3 线 . . . 22
- 8.4 圆 . . . 22
- 8.5 凸包 . . . 23
- 8.6 三角形 . . . 24
- 8.7 多边形 . . . 24
- 8.8 半平面交 . . . 24

9 杂项 **26**
- 9.1 生成树计数 . . . 26
- 9.2 类欧几里得 . . . 26
- 9.3 没有精度问题的整除 . . . 26

10 其它工具 **27**
- 10.1 编译命令 . . . 27
- 10.2 快读 . . . 27
- 10.3 Python Hints . . . 27
- 10.4 对拍器 . . . 27
- 10.5 常数表 . . . 27
- 10.6 试机赛 . . . 27
- 10.7 阴间错误集锦 . . . 27

1 做题指导

1.1 上机前你应该注意什么

1. 预估你需要多少机时，以及写出来以后预计要调试多久，并在纸上记下。
2. 先想好再上机，不要一边写一边想。
3. 如果有好写的题，务必先写好写的。
4. 把题目交给擅长的人来写，而不是空闲的人。

1.2 机上你应该注意什么

1. 如果你遇到了问题（做法假了、需要分讨等），先下机并通知队友。不要占着机时想。
2. 建议使用整块时间写题，尽量不要断断续续的写。

1.3 交题前你应该注意什么

1. `long long` 开了没有。
2. 数组开够没有。
3. 多测清了没有。
4. 边界数据 (corner case) 考虑了没有
5. 调试输出删了没有

编译命令：

```
g++ X.cpp -Wall -O2 -fsanitize=undefined -fsanitize=address X
# -fsanitize=undefined: 检测未定义行为
# -fsanitize=address: 检测内存溢出
```

1.4 如果你的代码挂了

按照优先级列出：

1. 先 P 再下机。(P 还没有送来就分屏调。)
2. 看 `long long` 开了没有，数组开够没有，多测清了没有。
3. 检查 typo ，你有没有打错一些难蚌的地方。
4. 看看你题读错没有。
5. 检查你的代码逻辑，即你的代码实现是否与做法一致。同时让另一个人重新读题。
6. 怀疑做法假了。拉一个人一起看代码。

1.5 另外一些注意事项

- 千万注意节奏，不要让任何一个人在一道题上卡得太久。必要的时候可以换题。
- 如果你想要写题，请想好再上机。诸如分类讨论一类的题目，更要想好再上。
- 后期的时候可以讨论，没必要一个人挂题。

1.6 阴间错误集锦

- 多测不清空。
- 该开 `long long` 的地方不开 `long long` 。一般情况下建议直接 `#define int long long` 。
- 注意变量名打错。例如 u 打成 v 或 a 打成 t 。建议读代码的时候专门检查此类错误。

2 图论

2.1 Tarjan 边双、点双（圆方树）

记录上一个访问的边时要记录边的编号，不能记录上一个过来的节点（因为会有重边）!!!

（如果选择在加边的时候特判，注意编号问题：用输入顺序来对应数组中位置的时候，重边跳过，但是需要 tot+=2。）

圆方树示意图：

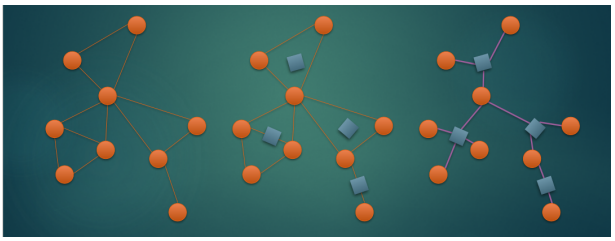

图 1: 圆方树示意图

```
/*** 缩点 ***/
// 找出强联通分量后，对每一条边查看是否在同一个 scc 中，如果不在就加边
void tarjan(int x)
{
   dfn[x]=low[x]=++Time,sta[++tp]=x,ins[x]=true;
   for(int i=hea[x];i;i=nex[i])
   {
      if(!dfn[ver[i]])
        tarjan(ver[i]),low[x]=min(low[x],low[ver[i]]);
      else if(ins[ver[i]]) low[x]=min(low[x],dfn[ver[i]]);
   }
   if(dfn[x]==low[x])
   {
        scc++;
      do { x=sta[tp],tp--,ins[x]=false,bel[x]=scc; }
        while (dfn[x]!=low[x]);
   }
}
/*** 割点 ***/
void tarjan(int x,int Last)
// Last 是边的编号，tot 初始值为 1，i 与 i^1 互为反边
{
    dfn[x]=low[x]=++Time;
    for(int i=hea[x];i;i=nex[i])
    {
        if(!dfn[ver[i]])
        {
            tarjan(ver[i],i),low[x]=min(low[x],low[ver[i]]);
            if(low[ver[i]]>dfn[x]) edg[i]=edg[i^1]=true;
        }
        else if(dfn[ver[i]]<dfn[x] && i!=(Last^1))
          low[x]=min(low[x],dfn[ver[i]]);
    }
}
/*** 圆方树、点双 ***/
// G 表示原图，T 是新建的圆方树
// 一条边也是点双
// 圆方树只存在圆 - 方边!!!
int Time,dfn[N],low[N],sta[N],siz[N];
vector<int> G[N],T[N<<1]; // 不要忘记给数组开两倍
void tarjan(int u)
{
    dfn[u]=low[u]=++Time,sta[++tp]=u;
    for(int v:G[u])
    {
        if(!dfn[v])
        {
            tarjan(v),low[u]=min(low[u],low[v]);
            // 这里对 low[v]>=dfn[u] 进行计数，根节点需要 2 个，其他节点需要 1 个，那么就是割点。
            if(low[v]==dfn[u])
            {
                int hav=0; ++All;
                for(int x=0;x!=v;tp--)
                  x=sta[tp],T[x].pb(All),T[All].pb(x),hav++;
                T[u].pb(All),T[All].pb(u);
                siz[All]=++hav;
            }
        }
        else low[u]=min(low[u],dfn[v]);
    }
}
```

2.2 O(1) LCA

按照顺序遍历子树，并查集更新。

```
int qhea[Maxq],qver[Maxq],qnex[Maxq],qid[Maxq];
inline void addq(int x,int y,int d)
    { qver[++qtot]=y, qnex[qtot]=qhea[x], qhea[x]=qtot, qid[qtot]=d; }
void dfs(int x,int Last_node)
{
    vis[x]=true;
    for(int i=hea[x];i;i=nex[i])
        if(ver[i]!=Last_node)
            dfs(ver[i],x),fa[ver[i]]=x;
    for(int i=qhea[x];i;i=qnex[i])
        if(vis[qver[i]])
            ans[qid[i]]=Find(qver[i]);
}

for(int i=1;i<=n;i++) fa[i]=i;
for(int i=1,x,y;i<n;i++) x=rd(),y=rd(),add(x,y),add(y,x);
for(int i=1,x,y;i<=m;i++) x=rd(),y=rd(),addq(x,y,i),addq(y,x,i);
dfs(rt,rt);
for(int i=1;i<=m;i++) printf("%d\n",ans[i]);
```

2.3 某些路径问题单 log 做法

路径加/单点求和 & 子树求和：设 $a[u]$ 为子树内 $d[u]$ 之和，则

1. 路径加：$d[u]+w$，$d[v]+w$，$d[lca(u,v)]-wd[fa[lca(u,v)]]-w$
2. 单点求和：$a[u]=sumd(u)$
3. 子树求和：$suma=\sum_v d[v]*(dep[v]-dep[u]+1)$，树状数组维护 2 系数即可

单点加 & 子树加/路径求和：设 $g[u]$ 表示 1 到 u 权值之和，则

1. 单点加：子树 d 加
2. 子树加：u 每个子树内 v 有 $g[v]+=w*(dep[v]-dep[u]+1)$ 则维护系数加即可
3. 路径和：$sum(u,v)=g[u]+g[v]-g[lca(u,v)]-g[fa[lca(u,v)]]$

2.4 重链剖分

```
vector<int> G[N];
int dep[N],fa[N],hson[N],sz[N],top[N],dfn[N],d;
void dfs1(int u,int f){
    dep[u]=dep[f]+1;
    fa[u]=f;sz[u]=1;
    for(int v:G[u]){
        if(v==f)continue;
        dfs1(v,u);sz[u]+=sz[v];
        if(sz[v]>sz[hson[u]])hson[u]=v;
    }
}
void dfs2(int u,int t){
    dfn[u]=++d;top[u]=t;
    if(hson[u])dfs2(hson[u],t);
    for(int v:G[u]){
        if(v==fa[u]||v==hson[u])continue;
        dfs2(v,v);
    }
}
void operate_path(int u,int v){ // 维护路径
    while(top[u]!=top[v]){
        if(dep[top[u]] < dep[top[v]]) swap(u,v);
        OPERATE(dfn[top[u]],dfn[u]); // 对这一区间执行任意操作
        u = fa[top[u]];
    }
    if(dep[u]>dep[v])swap(u,v);
    OPERATE(dfn[u],dfn[v]); // 跳到同一条重链上了
}
void operate_tree(int u){ // 维护子树
    OPERATE(dfn[u],dfn[u]+sz[u]-1);
}
int lca(int u,int v){ // 求最近公共祖先
    while(top[u]!=top[v]){
        if(dep[top[u]] < dep[top[v]]) swap(u,v);
        u = fa[top[u]];
    }
    if(dep[u]>dep[v])swap(u,v);
    return u;
}
```

2.5 2-SAT

2-sat 至多支持对**两个变量**之间关系进行限制。

n 个点中**恰好**只有一个点为 'true' 怎么办? 这是 $K-sat!!$ 至多有 1 个是可做的（前缀优化建图），但 "恰好有 1 个" 是不行的。

构造方案：Tarjan 对强联通分量的拓扑排序时逆序的，序号小的不可能到达序号大的。如果 $bel_i < bel_{i+n}$，则 i 所在的强联通分量不会推出 $i+n$，因此不产生矛盾，选择 i；反之，则选择 $i+n$。

DFS 求解 2-sat 有些题目要求某些条件优先满足，比如第 i 位优先选择 0/1 之类，可以采用 DFS 的方式求解。相当于二分图染色，如果染过了就不染，如果自己的反面染过了就返回失败。这样最坏复杂度是 $O(nm)$ 的，但是能够解决优先的问题。

2.6 Dinic 网络流、费用流

```cpp
/*** Dinic 最大流 ***/
struct Dinic
{
    #define Maxn 点数
    #define Maxm 边数
    int tot=1;
    int hea[Maxn],nex[Maxm<<1],ver[Maxm<<1];
    int tmphea[Maxn],dep[Maxn];
    ll edg[Maxm<<1],sum;
    inline void init()
        { tot=1,memset(hea,0,sizeof(hea)); }
    inline void add_edge(int x,int y,ll d)
    {
        ver[++tot]=y,nex[tot]=hea[x],hea[x]=tot,edg[tot]=d;
        ver[++tot]=x,nex[tot]=hea[y],hea[y]=tot,edg[tot]=0;
    }
    inline bool bfs(int s,int t)
    {
        memset(dep,0,sizeof(dep)),dep[s]=1;
        memcpy(tmphea,hea,sizeof(hea));
        queue<int> q; q.push(s);
        while(!q.empty())
        {
            int cur=q.front(); q.pop();
            if(cur==t) return true;
            for(int i=hea[cur];i;i=nex[i]) if(edg[i]>0 &&
              ↪ !dep[ver[i]])
                dep[ver[i]]=dep[cur]+1,q.push(ver[i]);
        }
        return false;
    }
    ll dfs(int x,ll flow,int t)
    {
        if(x==t || !flow) return flow;
        ll rest=flow,tmp;
        for(int i=tmphea[x];i && rest;i=nex[i])
        {
            tmphea[x]=i;
            if(dep[ver[i]]==dep[x]+1 && edg[i]>0)
            {
                if(!(tmp = dfs(ver[i],min(edg[i],rest),t)))
                  ↪ dep[ver[i]]=0;
                edg[i]-=tmp,edg[i^1]+=tmp,rest-=tmp;
            }
        }
        return flow-rest;
    }
    inline ll solve(int s,int t)
    {
        sum=0;
        while(bfs(s,t)) sum+=dfs(s,inf,t);
        return sum;
    }
    #undef Maxn
    #undef Maxm
}G;
```

```cpp
/*** Dinic 最小费最大流 ***/
struct Dinic_cost
{
    #define Maxn 点数
    #define Maxm 边数
    int tot=1;
    int tmphea[Maxn],hea[Maxn],nex[Maxm<<1],ver[Maxm<<1];
    ll sumflow,sumcost,edg[Maxm<<1],Cost[Maxm<<1];
    bool inq[Maxn];
    ll dis[Maxn];
    inline void init() { tot=1,memset(hea,0,sizeof(hea)); }
    inline void add_edge(int x,int y,ll d,ll c)
    {
        ver[++tot]=y, nex[tot]=hea[x], hea[x]=tot, edg[tot]=d,
          ↪ Cost[tot]=c;
        ver[++tot]=x, nex[tot]=hea[y], hea[y]=tot, edg[tot]=0,
          ↪ Cost[tot]=-c;
    }
    inline bool spfa(int s,int t)
    {
        memset(dis,0x3f,sizeof(dis)),dis[s]=0;
        memcpy(tmphea,hea,sizeof(hea));
        queue<int> q; q.push(s);
        while(!q.empty())
        {
            int cur=q.front(); q.pop(),inq[cur]=false;
            for(int i=hea[cur];i;i=nex[i])
                if(edg[i] && dis[ver[i]]>dis[cur]+Cost[i])
                {
                    dis[ver[i]]=dis[cur]+Cost[i];
                    if(!inq[ver[i]])
                      ↪ q.push(ver[i]),inq[ver[i]]=true;
                }
        }
        return dis[t]!=infll;
    }
    ll dfs(int x,ll flow,int t)
    {
        if(x==t || !flow) return flow;
        ll rest=flow,tmp;
        inq[x]=true;
        for(int i=tmphea[x];i && rest;i=nex[i])
        {
            tmphea[x]=i;
            if(!inq[ver[i]] && edg[i] &&
              ↪ dis[ver[i]]==dis[x]+Cost[i])
            {
                if(!(tmp = dfs(ver[i],min(edg[i],rest),t)))
                    dis[ver[i]]=infll;
                sumcost+=Cost[i]*tmp, edg[i]-=tmp,
                  ↪ edg[i^1]+=tmp, rest-=tmp;
            }
        }
        inq[x]=false;
        return flow-rest;
    }
    inline pll solve(int s,int t)
    {
        sumflow=sumcost=0;
        while(spfa(s,t)) sumflow+=dfs(s,infll,t);
        return pll(sumflow,sumcost);
    }
    #undef Maxn
    #undef Maxm
}G;
```

2.6.1 无源汇有上下界可行流

给定无源汇流量网络 G，询问是否存在一种标定每条边流量的方式，使得每条边流量满足上下界同时每一个点流量平衡。

```cpp
#include <bits/stdc++.h>
using namespace std;
#define infll 0x3f3f3f3f3f3f3f3f
#define inf 0x3f3f3f3f
#define pb push_back
#define eb emplace_back
#define pa pair<int, int>
#define fi first
#define se second
typedef long long ll;
inline int rd() {
    int x = 0;
    char ch, t = 0;
    while (!isdigit(ch = getchar())) t |= ch == '-';
    while (isdigit(ch)) x = x * 10 + (ch ^ 48), ch = getchar();
    return t ? -x : x;
}
struct Dini {
#define Maxn 505
#define Maxm 20005
    int tot = 1;
    int hea[Maxn], tmphea[Maxn], dep[Maxn];
    int nex[Maxm << 1], ver[Maxm << 1], num[Maxm << 1];
    ll edg[Maxm << 1];
    inline void addedge(int x, int y, int d, int _n) {
        ver[++tot] = y, nex[tot] = hea[x], hea[x] = tot,
          ↪ edg[tot] = d, num[tot] = -1;
        ver[++tot] = x, nex[tot] = hea[y], hea[y] = tot,
          ↪ edg[tot] = 0, num[tot] = _n;
    }
    inline bool bfs(int s, int t) {
```

```
        memcpy(tmphea, hea, sizeof(hea));
        memset(dep, 0, sizeof(dep)), dep[s] = 1;
        queue<int> q;
        q.push(s);
        while (!q.empty()) {
            int cur = q.front();
            q.pop();
            if (cur == t) return true;
            for (int i = hea[cur]; i; i = nex[i])
                if (edg[i] > 0 && !dep[ver[i]])
                    dep[ver[i]] = dep[cur] + 1, q.push(ver[i]);
        }
        return false;
    }
    ll dfs(int x, int t, ll flow) {
        if (!flow || x == t) return flow;
        ll rest = flow, tmp;
        for (int i = tmphea[x]; i && rest; i = nex[i]) {
            tmphea[x] = i;
            if (dep[ver[i]] == dep[x] + 1 && edg[i] > 0) {
                if (!(tmp = dfs(ver[i], t, min(rest, edg[i]))))
                  ↪ dep[ver[i]] = 0;
                edg[i] -= tmp, edg[i ^ 1] += tmp, rest -= tmp;
            }
        }
        return flow - rest;
    }
    inline ll solve(int s, int t) {
        ll sum = 0;
        while (bfs(s, t))
            sum += dfs(s, t, infll);
        return sum;
    }
#undef Maxn
#undef Maxm
} G;
#define Maxn 205
#define Maxm 10205
int n, m, ss, tt;
int ans[Maxm];
ll needin, needout;
ll Ind[Maxn], Outd[Maxn];
int main() {
    n = rd(), m = rd(), ss = n + 1, tt = n + 2;
    for (int i = 1, x, y, Inf, Sup; i <= m; i++) {
        x = rd(), y = rd(), Inf = rd(), Sup = rd();
        G.addedge(x, y, Sup - Inf, i);
        Outd[x] += Inf;
        Ind[y] += Inf;
        ans[i] = Inf;
    }
    for (int i = 1; i <= n; i++) {
        if (Ind[i] > Outd[i])
            G.addedge(ss, i, Ind[i] - Outd[i], -1), needin +=
              ↪ Ind[i] - Outd[i];
        if (Ind[i] < Outd[i])
            G.addedge(i, tt, Outd[i] - Ind[i], -1), needout +=
              ↪ Outd[i] - Ind[i];
    }
    ll tmp = G.solve(ss, tt);
    if (needin != needout || needin != tmp) printf("NO\n");
    else {
        for (int i = 2; i <= G.tot; i++)
            if (G.num[i] != -1) ans[G.num[i]] += G.edg[i];
        printf("YES\n");
        for (int i = 1; i <= m; i++) printf("%d\n", ans[i]);
    }
    return 0;
}
```

2.6.2 有源汇有上下界最大流

给定有源汇流量网络 G，询问是否存在一种标定每条边流量的方式，使得每条边流量满足上下界同时除了源点和汇点每一个点流量平衡。

假设源点为 S，汇点为 T，则我们可以加入一条 T 到 S 的上界为 ∞，下界为 0 的边转化为无源汇上下界可行流问题。

若有解，则 S 到 T 的可行流流量等于 T 到 S 的附加边的流量。

```
#include <bits/stdc++.h>
using namespace std;
#define infll 0x3f3f3f3f3f3f3f3f
#define inf 0x3f3f3f3f
#define pb push_back
#define eb emplace_back
#define pa pair<int, int>
#define fi first
#define se second
typedef long long ll;
inline int rd() {
    int x = 0;
    char ch, t = 0;
    while (!isdigit(ch = getchar())) t |= ch == '-';
    while (isdigit(ch)) x = x * 10 + (ch ^ 48), ch = getchar();
    return t ? -x : x;
}
struct Dinic {
#define Maxn 505
#define Maxm 20005
    int tot = 1;
    int hea[Maxn], tmphea[Maxn], dep[Maxn];
    int nex[Maxm << 1], ver[Maxm << 1], num[Maxm << 1];
    ll edg[Maxm << 1];
    inline int addedge(int x, int y, ll d, int _n) {
        ver[++tot] = y, nex[tot] = hea[x], hea[x] = tot,
          ↪ edg[tot] = d, num[tot] = -1;
        ver[++tot] = x, nex[tot] = hea[y], hea[y] = tot,
          ↪ edg[tot] = 0, num[tot] = _n;
        return tot;
    }
    inline bool bfs(int s, int t) {
        memcpy(tmphea, hea, sizeof(hea));
        memset(dep, 0, sizeof(dep)), dep[s] = 1;
        queue<int> q;
        q.push(s);
        while (!q.empty()) {
            int cur = q.front();
            q.pop();
            if (cur == t) return true;
            for (int i = hea[cur]; i; i = nex[i])
                if (edg[i] > 0 && !dep[ver[i]])
                    dep[ver[i]] = dep[cur] + 1, q.push(ver[i]);
        }
        return false;
    }
    ll dfs(int x, int t, ll flow) {
        if (!flow || x == t) return flow;
        ll rest = flow, tmp;
        for (int i = tmphea[x]; i && rest; i = nex[i]) {
            tmphea[x] = i;
            if (dep[ver[i]] == dep[x] + 1 && edg[i] > 0) {
                if (!(tmp = dfs(ver[i], t, min(rest, edg[i]))))
                    dep[ver[i]] = 0;
                edg[i] -= tmp, edg[i ^ 1] += tmp, rest -= tmp;
            }
        }
        return flow - rest;
    }
    inline ll solve(int s, int t) {
        ll sum = 0;
        while (bfs(s, t))
            sum += dfs(s, t, infll);
        return sum;
    }
#undef Maxn
#undef Maxm
} G;
#define Maxn 505
#define Maxm 20005
int n, m, ss, tt, s, t;
int ans[Maxm];
ll needin, needout;
ll Ind[Maxn], Outd[Maxn];
int main() {
    n = rd(), m = rd(), s = rd(), t = rd(), ss = n + 1, tt = n +
      ↪ 2;
    for (int i = 1, x, y, Inf, Sup; i <= m; i++) {
        x = rd(), y = rd(), Inf = rd(), Sup = rd();
        G.addedge(x, y, Sup - Inf, i);
        Outd[x] += Inf;
        Ind[y] += Inf;
        ans[i] = Inf;
    }
    for (int i = 1; i <= n; i++) {
        if (Ind[i] > Outd[i])
            G.addedge(ss, i, Ind[i] - Outd[i], -1), needin +=
              ↪ Ind[i] - Outd[i];
        if (Ind[i] < Outd[i])
            G.addedge(i, tt, Outd[i] - Ind[i], -1), needout +=
              ↪ Outd[i] - Ind[i];
    }
    G.addedge(t, s, infll, -1);
    ll tmp = G.solve(ss, tt);
    if (needin != needout || needin != tmp) printf("please go
      ↪ home to sleep\n");
    else {
        tmp = G.edg[G.tot];
        G.edg[G.tot] = G.edg[G.tot - 1] = 0;
        tmp += G.solve(s, t);
        printf("%lld\n", tmp);
```

```
    }
    return 0;
}
```

2.6.3 网络流总结

最小割集，最小割必须边以及可行边

最小割集 从 S 出发，在残余网络中 BFS 所有权值非 0 的边（包括反向边），得到点集 $\{S\}$，另一集为 $\{V\}-\{S\}$.

最小割集必须点 残余网络中 S 直接连向的点必在 S 的割集中，直接连向 T 的点必在 T 的割集中；若这些点的并集为全集，则最小割方案唯一.

最小割可行边 在残余网络中求强联通分量，将强联通分量缩点后，剩余的边即为最小割可行边，同时这些边也必然满流.

最小割必须边 在残余网络中求强联通分量，若 S 出发可到 u，T 出发可到 v，等价于 $\text{scc}_S=\text{scc}_u$ 且 $\text{scc}_T=\text{scc}_v$，则该边为必须边.

常见问题

最大权闭合子图 点权，限制条件形如：选择 A 则必须选择 B，选择 B 则必须选择 C，D. 建图方式：B 向 A 连边，CD 向 B 连边. 求解：S 向正权点连边，负权点向 T 连边，其余边容量 ∞，求最小割，答案为 S 所在最小割集.

二次布尔型（文理分科） n 个点分为两类，i 号点有 l_i 或 r_i 的代价，i,j 同属一侧分别获得 l_{ij} 或 r_{ij} 的代价，问最小代价. $L\to i:(l_i+1/2\sum_j l_{ij})$，$i\to R:(r_i+1/2\sum_j r_{ij})$，$i\leftrightarrow j:1/2(l_{ij}+r_{ij})$. 实现时边权乘 2 为整数，求解后答案除 2 为整数. 图拆点可以看作二分图.
如果是二元限制是分类不同时有一个代价 d_{ij}，建图可以简化为 $L\to i:l_i$，$i\to R:r_i$，$i\leftrightarrow j:d_{ij}$. 经典例子：`xor` 最小值，按位拆开建图.

混合图欧拉回路 把无向边随便定向，计算每个点的入度和出度，如果有某个点出入度之差 $\deg_i=\text{in}_i-\text{out}_i$ 为奇数，肯定不存在欧拉回路. 对于 $\deg_i>0$ 的点，连接边 $(i,T,\deg_i/2)$；对于 $\deg_i<0$ 的点，连接边 $(S,i,-\deg_i/2)$. 最后检查是否满流即可.

二物流 水源 S_1，水汇 T_1，油源 S_2，油汇 T_2，每根管道流量共用. 求流量和最大. 建超级源 SS_1 汇 TT_1，连边 $SS_1\to S_1$,$SS_1\to S_2$,$T_1\to TT_1$,$T_2\to TT_1$，设最大流为 x_1. 建超级源 SS_2 汇 TT_2，连边 $SS_2\to S_1$，$SS_2\to T_2$,$T_1\to TT_2$，$S_2\to TT_2$，设最大流为 x_2. 则最大流中水流量 $\frac{x_1+x_2}{2}$，油流量 $\frac{x_1-x_2}{2}$.

无源汇有上下界可行流 每条边 (u,v) 有一个上界容量 $C_{u,v}$ 和下界容量 $B_{u,v}$，我们让下界变为 0，上界变为 $C_{u,v}-B_{u,v}$，但这样做流量不守恒. 建立超级源点 SS 和超级汇点 TT，用 du_i 来记录每个节点的流量情况，$du_i=\sum B_{j,i}-\sum B_{i,j}$，添加一些附加弧. 当 $du_i>0$ 时，连边 (SS,i,du_i)；当 $du_i<0$ 时，连边 $(i,TT,-du_i)$. 最后对 (SS,TT) 求一次最大流即可，当所有附加边全部满流时（即 maxflow == $du_i>0$ ）时有可行解.

有源汇有上下界最大可行流 建立超级源点 SS 和超级汇点 TT，首先判断是否存在可行流，用无源汇有上下界可行流的方法判断. 增设一条从 T 到 S 没有下界容量为无穷的边，那么原图就变成了一个无源汇有上下界可行流问题. 同样地建图后，对 (SS,TT) 进行一次最大流，判断是否有可行解. 如果有可行解，删除超级源点 SS 和超级汇点 TT，并删去 T 到 S 的这条边，再对 (S,T) 进行一次最大流，此时得到的 maxflow 即为有源汇有上下界最大可行流.

有源汇有上下界最小可行流 建立超级源点 SS 和超级汇点 TT，和无源汇有上下界可行流一样新增一些边，然后从 SS 到 TT 跑最大流. 接着加上边 (T,S,∞)，再从 SS 到 TT 跑一遍最大流. 如果所有新增边都是满的，则存在可行流，此时 T 到 S 这条边的流量即为最小可行流.

有上下界费用流 如果求无源汇有上下界最小费用可行流或有源汇有上下界最小费用最大可行流，用 1.6.3.1/1.6.3.2 的构图方法，给边加上费用即可. 求有源汇有上下界最小费用最小可行流，要先用 1.6.3.3 的方法建图，先求出一个保证必要边满流情况下的最小费用. 如果费用全部非负，那么这时的费用就是答案. 如果费用有负数，那么流多了可能更好，继续做从 S 到 T 的流量任意的最小费用流，加上原来的费用就是答案.

费用流消负环 新建超级源 SS 汇 TT，对于所有流量非空的负权边 e，先流满（`ans+=e.f*e.c, e.rev.f+=e.f, e.f=0`），再连边 SS→e.to，e.from→TT，流量均为 e.f(>0)，费用均为 0. 再连边 T→S 流量 ∞ 费用 0. 此时没有负环了. 做一遍 SS 到 TT 的最小费用最大流，将费用累加 ans，拆掉 T→S 的那条边（此边的流量为残量网络中 S→T 的流量）. 此时负环已消，再继续跑最小费用最大流.

整数线性规划转费用流

首先将约束关系转化为所有变量下界为 0，上界没有要求，并满足一些等式，每个变量在均在等式左边且出现恰好两次，系数为 +1 和 −1，优化目标为 $\max\sum v_ix_i$ 的形式. 将等式看做点，等式 i 右边的值 b_i 若为正，则 S 向 i 连边 $(b_i,0)$，否则 i 向 T 连边 $(-b_i,0)$. 将变量看做边，记变量 x_i 的上界为 m_i(无上界则 $m_i=inf$)，将 x_i 系数为 +1 的那个等式 u 向系数为 −1 的等式 v 连边 (m_i,v_i).

2.7 差分约束

差分约束系统是一种特殊的 n 元一次不等式组。

差分约束系统中的每个约束条件 $x_i-x_j\le c_k$ 都可以变形成 $x_i\le x_j+c_k$ 与 $x_j\ge x_i-c_k$ ，这与单源最短路中的三角形不等式非常相似。因此，我们可以把每个变量 x_i 看做图中的一个结点，对于每个约束条件连边。

需要注意的是，有些题目看能会对解的上、下界进行约束，因此我们需要对这些条件处理（这里只考虑对于这 n 个元素 **只约束了上界**或 **只约束了下界**）：

- 只约束下界：有 0 号点向每一个点连一条长为 Lim_i 的边，表示第 i 号元素的 **下界**为 Lim_i ，如图所示建边：

题意	转化	连边
$x_a-x_b\ge c$	$x_a\ge x_b+c$	`add(b,a,c)`
$x_a-x_b\le c$	$x_b\ge x_a-c$	`add(a,b,-c)`
$x_a=x_b$	$x_a\ge x_b$ ，$x_b\le x_a$	`add(a,b,0),add(b,a,0)`

之后对整张图跑 **最长路**。

- 只约束上界：有 0 号点向每一个点连一条长为 Lim_i 的边，表示第 i 号元素的 **上界**为 Lim_i ，如图所示建边：

题意	转化	连边
$x_a-x_b\ge c$	$x_b\le x_a-c$	`add(a,b,-c)`
$x_a-x_b\le c$	$x_a\le x_b+c$	`add(b,a,c)`
$x_a=x_b$	$x_a\le x_b$ ，$x_b\le x_a$	`add(a,b,0),add(b,a,0)`

之后对整张图跑 **最短路**。

设 $dist[0]=0$ ，若存在负环 / 正环，则不等式无解，否则 $x_i=dist[i]$ 是该差分约束系统的一组解。

最坏情况下（存在负环 / 正环）复杂度为 $O(nm)$ 。

注意：整个图不一定是联通的！

2.8 欧拉路径

2.8.1 有向图

```
vector<int> G[N];
int cur[N];
void dfs(int u){
    for(int& i=cur[u];i<G[u].size();){
        i++; // 提前加 i
        dfs(G[u][i-1]);
        // 在这里往栈中加入边 (u -> G[u][i-1])
    }
    // 在这里往栈中加入点 u
    ans.push(u);
}
```

2.8.2 无向图

```cpp
struct Edge{
    int to,rev; // 终点，反向边编号
    bool exist;
};
vector<Edge> G[N];
int cur[N];
stack<int> ans;
void add_edge(int u,int v){
    G[u].push_back(Edge{v,(int)G[v].size(),true});
    if(u!=v)G[v].push_back(Edge{u,(int)G[u].size()-1,true});
}
void dfs(int u){
    for(int& i=cur[u];i<G[u].size();){
        i++;
        if(!G[u][i-1].exist)continue;
        G[u][i-1].exist = 0;
        G[G[u][i-1].to][G[u][i-1].rev].exist=0;
        dfs(G[u][i-1].to);
        // 在这里加入边 (u->G[u][i-1].to)
    }
    // 在这里加入节点 u
}
```

2.9 同余最短路

形如：

- 设问 1：给定 n 个整数，求这 n 个整数在 $h(h \le 2^{63}-1)$ 范围内 **能拼凑出多少的其他整数（整数可以重复取）**。
- 设问 2：给定 n 个整数，求这 n 个整数 **不能拼凑出的最小（最大）的整数**。

设 x 为 n 个数中最小的一个，令 $ds[i]$ 为只通过增加其他 $n-1$ 种数能够达到的最低楼层 p，并且满足 $p \equiv i \pmod{x}$。
对于 $n-1$ 个数与 x 个 $ds[i]$，可以如下连边：

```cpp
for(int i=0;i<x;i++) for(int j=2;j<=n;j++)
  ↪ add(i,(i+a[j])%x,a[j]);
```

之后进行最短路，对于：

- 问在 h 范围内能够到达的**点的数量**：答案为（加一因为 i 本身也要计算）

$$\sum_{i=0}^{x-1} [d[i] \le h] \times \frac{h-d[i]}{x} + 1$$

- 问不能达到的**最小的数**：答案为：（i 一定时最小表示的数为 $d[i] = s \times x + i$，则 $(s-1) \times x + i$ **一定不能**被表示出来）

$$\min_{i=1}^{x-1} \{d[i] - x\}$$

注意：ds 与 h 范围相同，一般也要开 long long！

3 数据结构

3.1 平衡树

3.1.1 FHQ Treap

```
struct FHQ_number
{
   #define Maxn 点数
   #define ls tree[p].pl
   #define rs tree[p].pr
   private:
      int All=0,root=0;
      struct NODE { int pl,pr,siz,cnt,rnd; ll val; };
      NODE tree[Maxn];
      inline int Dot() { return ++All; }
      inline int New(ll Val)
      {
         int p=Dot();
         tree[p].rnd=rand(),tree[p].val=Val;
         tree[p].siz=tree[p].cnt=1;
         tree[p].pl=tree[p].pr=0;
         return p;
      }
      inline void pushdown(int p) { p--; }
      inline void pushup(int p)
         { tree[p].siz=tree[ls].siz+tree[rs].siz+tree[p].cnt; }
      void split(int p,int k,int &x,int &y)
      {
         if(!p) { x=y=0; return; }
         pushdown(p);
         if(tree[p].val<=k) x=p,split(rs,k,rs,y);
         else y=p,split(ls,k,x,ls);
         pushup(p);
      }
      int merge(int x,int y)
      {
         if(!x || !y) return x+y;
         if(tree[x].rnd<tree[y].rnd)
         {
            pushdown(x),tree[x].pr=merge(tree[x].pr,y),pushup(x);
            return x;
         }
         else
         {
            pushdown(y),tree[y].pl=merge(x,tree[y].pl),pushup(y);
            return y;
         }
      }
      inline int kth(int p,int Rank)
      {
         while(p)
         {
            if(tree[ls].siz>=Rank) p=ls;
            else if(tree[ls].siz+tree[p].cnt>=Rank) return p;
            else Rank-=tree[ls].siz+tree[p].cnt,p=rs;
         }
         return p;
      }
      int x,y,z;
   public:
      inline void Insert(ll Val)
         { split(root,Val,x,y),root=merge(merge(x,New(Val)),y);
           ↪ }
      inline void Delete_one(int Val)
      {
         split(root,Val,x,z),split(x,Val-1,x,y);
         y=merge(tree[y].pl,tree[y].pr);
         root=merge(merge(x,y),z);
      }
      inline ll Rank_to_Value(int Rank)
         { return tree[kth(root,Rank)].val; }
      inline int Value_to_Rank(ll Value)
      {
         split(root,Value-1,x,y);
         int ret=tree[x].siz+1;
         root=merge(x,y);
         return ret;
      }
      inline ll Findpre(ll Value)
      {
         split(root,Value-1,x,y);
         ll ret=tree[kth(x,tree[x].siz)].val;
         root=merge(x,y);
         return ret;
      }
      inline ll Findnex(ll Value)
      {
         split(root,Value,x,y);
         ll ret=tree[kth(y,1)].val;
         root=merge(x,y);
         return ret;
      }
}T;
struct FHQ_sequence
{
   #define Maxn 点数
   #define ls tree[p].pl
   #define rs tree[p].pr
   int All=0,root=0;
   struct NODE { int pl,pr,siz,cnt,rnd,val; bool tag; };
   NODE tree[Maxn];
   inline int Dot() { return ++All; }
   inline int New(int Val)
   {
      int p=Dot();
      tree[p].rnd=rand(),tree[p].val=Val;
      tree[p].cnt=tree[p].siz=1;
      tree[p].pl=tree[p].pr=0;
      return p;
   }
   inline void pushdown(int p)
   {
      if(!tree[p].tag) return;
      swap(tree[ls].pl,tree[ls].pr);
      swap(tree[rs].pl,tree[rs].pr);
      tree[ls].tag^=1,tree[rs].tag^=1;
      tree[p].tag=false;
   }
   inline void pushup(int p)
      { tree[p].siz=tree[ls].siz+tree[p].cnt+tree[rs].siz; }
   void split(int p,int k,int &x,int &y)
   {
      if(!p) { x=y=0; return; }
      pushdown(p);
      if(tree[ls].siz<k) x=p,split(rs,k-tree[ls].siz-1,rs,y);
      else y=p,split(ls,k,x,ls);
      pushup(p);
   }
   int merge(int x,int y)
   {
      if(!x || !y) return x+y;
      if(tree[x].rnd<tree[y].rnd)
      {
         pushdown(x),tree[x].pr=merge(tree[x].pr,y),pushup(x);
         return x;
      }
      else
      {
         pushdown(y),tree[y].pl=merge(x,tree[y].pl),pushup(y);
         return y;
      }
   }
   int kth(int p,int Rank)
   {
      while(p)
      {
         if(tree[ls].siz>=Rank) p=ls;
         else if(tree[ls].siz+tree[p].cnt>=Rank) return p;
         else Rank-=tree[ls].siz+tree[p].cnt,p=ls;
      }
      return p;
   }
   void Insert(int Val) // 插到末尾
      { root=merge(root,New(Val)); }
   int x,y,z;
   inline void Reverse(int l,int r)
   {
      split(root,r,x,z),split(x,l-1,x,y);
      swap(tree[y].pl,tree[y].pr),tree[y].tag^=1;
      root=merge(merge(x,y),z);
   }
   void print(int p)
   {
      pushdown(p);
      if(ls) print(ls);
      printf("%d ",tree[p].val);
      if(rs) print(rs);
   }
}T;
```

3.1.2 平衡树合并

如果需要合并两个有交集的 Treap 时该怎么做?我们可以每次将较小的数合并到较大的树中去，这样每个点最多只会合并 $\log n$ 次，每次合并复杂度 $O(n \log n)$，总时间复杂度 $O(n \log n \log V)$。

代码其实非常暴力，就是直接对更小的那棵树直接一个个插入进去：

```
// ls,rs 为 tree[p].pl 和 tree[p].pr
```

```
void mergetree(int p,int &rt) // rt = (rt+p)
{
    if(!p) return;
    mergetree(ls,rt);
    mergetree(rs,rt);
    int x,y;
    split(rt,tree[p].val,x,y);
    ls=rs=0,rt=merge(merge(x,p),y);
}
```

可以证明，若只支持合并与分裂操作，则时间复杂度为 $O(n \log n)$ 。

3.1.3 Splay

```
// 洛谷 P3391
// 理论上 splay 的每次操作以后都应该立刻做 splay 操作以保证均摊时间复杂度，但是某些地方又不能立刻做，例如下面的 kth 。
int rt;
struct Splay {
    struct node {
        int val, siz, fa, ch[2], rv;
        #define ls(x) nod[x].ch[0]
        #define rs(x) nod[x].ch[1]
        #define val(x) nod[x].val
        #define siz(x) nod[x].siz
        #define fa(x) nod[x].fa
        #define rv(x) nod[x].rv
    } nod[N];
    int cnt;
    bool chk(int x) { return x == rs(fa(x)); }
    void reverse(int x) { rv(x) ^= 1, swap(ls(x), rs(x)); }
    void pushup(int x) { siz(x) = siz(ls(x)) + 1 + siz(rs(x)); }
    void pushdown(int x) { if (rv(x)) reverse(ls(x)), reverse(rs(x)), rv(x) = 0; }
    void connect(int x, int fa, int son) { fa(x) = fa, nod[fa].ch[son] = x; }
    void rotate(int x) {
        int y = fa(x), z = fa(y), ys = chk(x), zs = chk(y), u = nod[x].ch[!ys];
        connect(u, y, ys), connect(y, x, !ys), connect(x, z, zs), pushup(y), pushup(x);
    }
    void pushall(int x) { if (fa(x)) pushall(fa(x)); pushdown(x); }
    void splay(int x, int to) {
        pushall(x);
        while (fa(x) != to) {
            int y = fa(x);
            if (fa(y) != to) rotate(chk(x) == chk(y) ? y : x);
            rotate(x);
        }
        if (!to) rt = x;
    } // 将 x 伸展为 to 的儿子。
    void append(int val) {
        if (!rt) nod[rt = ++cnt] = {val, 1};
        else {
            int x = rt;
            while (rs(x)) pushdown(x), x = rs(x);
            splay(x, 0), nod[rs(x) = ++cnt] = {val, 1, x}, pushup(x);
        }
    }
    int kth(int k) {
        int x = rt;
        while (x) {
            pushdown(x);
            if (siz(ls(x)) + 1 == k) return x;
            else if (k <= siz(ls(x))) x = ls(x);
            else k -= siz(ls(x)) + 1, x = rs(x);
        }
        return -1;
    } // kth 做完以后不能立刻 splay ，因为需要提取区间。
    void reverse(int l, int r) {
        splay(kth(r + 2), 0), splay(kth(l), rt);
        reverse(rs(ls(rt))), pushup(ls(rt)), pushup(rt);
    } // 这里添加了前后两个哨兵，以避免额外的分类讨论。
} spl;
```

3.2 LCT 动态树

```
// 洛谷 P3690
struct LCT {
    struct node {
        int rv, ch[2], fa, sm, val;
        #define ls(x) nod[x].ch[0]
        #define rs(x) nod[x].ch[1]
        #define fa(x) nod[x].fa
        #define sm(x) nod[x].sm
        #define rv(x) nod[x].rv
        #define val(x) nod[x].val
    } nod[N];
    // 根节点的父亲：链顶节点的树上父亲。
    // 其余节点的父亲：splay 中的父亲。

    bool chk(int x) { return rs(fa(x)) == x; }
    bool isroot(int x) { return nod[fa(x)].ch[chk(x)] != x; }
    void pushup(int x) { sm(x) = sm(ls(x)) ^ val(x) ^ sm(rs(x)); }
    void reverse(int x) { rv(x) ^= 1, swap(ls(x), rs(x)); }
    void pushdown(int x) {
        if (rv(x)) reverse(ls(x)), reverse(rs(x)), rv(x) = 0;
    }
    void connect(int x, int fa, int son) { fa(x) = fa, nod[fa].ch[son] = x; }
    void rotate(int x) {
        int y = fa(x), z = fa(y), ys = chk(x), zs = chk(y), u = nod[x].ch[!ys];
        if (isroot(y)) fa(x) = z;
        else connect(x, z, zs);
        connect(u, y, ys), connect(y, x, !ys), pushup(y), pushup(x);
    }
    void pushall(int x) { if (!isroot(x)) pushall(fa(x)); pushdown(x); }
    void splay(int x) {
        pushall(x);
        while (!isroot(x)) {
            if (!isroot(fa(x))) rotate(chk(x) == chk(fa(x)) ? fa(x) : x);
            rotate(x);
        }
    }
    void access(int x) { for (int y = 0; x; y = x, x = fa(x)) splay(x), rs(x) = y, pushup(x); }
    void makeroot(int x) { access(x), splay(x), reverse(x); }
    int findroot(int x) { access(x), splay(x); while (ls(x)) pushdown(x), x = ls(x); return splay(x), x; }
    void link(int x, int y) { makeroot(y); if (findroot(x) != y) fa(y) = x;}
    void split(int x, int y) { makeroot(y), access(x), splay(x); }
    void cut(int x, int y) { split(x, y); if (ls(x) == y) ls(x) = fa(y) = 0, pushup(x); }
    void modify(int x, int val) { splay(x), val(x) = val, pushup(x); }
    int sum(int x, int y) { split(x, y); return sm(x); }
    // 任何操作过后都应该立即 splay 以保证均摊复杂度。
} lct;
```

3.3 ODT 珂朵莉树

```
// 珂朵莉树的本质是颜色段均摊。若保证数据随机，可以证明其期望时间复杂度为 O(nlogn) 。
struct ODT {
    struct node {
        int l, r;
        mutable int val;
        node(int L, int R, int V) { l = L, r = R, val = V; }
        bool operator < (const node &rhs) const { return l < rhs.l; }
    };
    set<node> s;
    auto split(int x) {
        auto it = s.lower_bound(node(x, 0, 0));
        if (it != s.end() && it->l == x) return it;
        it--;
        int l = it->l, r = it->r, val = it->val;
        s.erase(it), s.insert(node(l, x - 1, val));
        return s.insert(node(x, r, val)).first;
    }
    void assign(int l, int r, int val) {
        // 此处须先 split(r + 1) 。因为若先 split(l) ，则后来的 split(r + 1) 可能致使 itl 失效。
        auto itr = split(r + 1), itl = split(l);
        s.erase(itl, itr), s.insert(node(l, r, val));
    }
    void perform (int l, int r) {
        auto itr = split(r + 1), itl = split(l);
        while (itl != itr) {
            // do something...
            itl++;
        }
    }
} odt;
```

3.4 李超线段树

```
struct Line { double b,k; }a[Maxn];
inline double calc(int p,int x){ return a[p].b+a[p].k*1.0*x; }
inline int tomax(int p1,int p2,int x)
    { return (calc(p1,x)>calc(p2,x)+eps)?p1:p2; }
struct Segment_Tree
{
    int tree[Maxn<<2];
    void add(int p,int nl,int nr,int l,int r,int x)
    {
        int mid=(nl+nr)>>1;
        if(nl>=l && nr<=r)
        {
            if(calc(x,mid)>calc(tree[p],mid)+eps)
              ↪ swap(tree[p],x);
            if(calc(x,nl)>calc(tree[p],nl)+eps)
              ↪ add(p<<1,nl,mid,l,r,x);
            if(calc(x,nr)>calc(tree[p],nr)+eps)
              ↪ add(p<<1|1,mid+1,nr,l,r,x);
            return;
        } // 插入 O(log^2n) : 定位到 O(logn) 个区间，每个区间
          ↪ O(logn) 递归到叶子。
        if(mid>=l) add(p<<1,nl,mid,l,r,x);
        if(mid<r) add(p<<1|1,mid+1,nr,l,r,x);
    }
    int query(int p,int nl,int nr,int x)
    {
        if(nl==nr) return tree[p];
        int mid=(nl+nr)>>1;
        if(mid>=x) return tomax(tree[p],query(p<<1,nl,mid,x),x);
        else return tomax(tree[p],query(p<<1|1,mid+1,nr,x),x);
    }
}T;
```

李超线段树的合并本质上和线段树没什么区别，只是在 merge 完两棵子树后的 update 中改为将 y 的最优线段再放到 x 中进行一次 add 即可。

```
int merge(int x,int y,int nl,int nr)
{
    if(!x || !y) return x+y;
    int mid=(nl+nr)>>1;
    tree[x].pl=merge(tree[x].pl,tree[y].pl,nl,mid);
    tree[x].pr=merge(tree[x].pr,tree[y].pr,mid+1,nr);
    add(x,nl,nr,tree[y].num);
    return x;
}
```

3.5 二维树状数组

```
struct BIT {
    int sm[N][N][4]; // sm 是 sum 的缩写。
    int lowbit(int x) { return x & -x; }
    void add(int x, int y, int k) {
       int a = k, b = k * x, c = k * y, d = k * x * y;
       for (int i = x; i <= n; i += lowbit(i)) {
            for (int j = y; j <= m; j += lowbit(j)) {
               sm[i][j][0] += a;
               sm[i][j][1] += b;
               sm[i][j][2] += c;
               sm[i][j][3] += d;
          }
       }
    }
    int query(int x,int y) {
       int ret = 0, a = x * y + x + y + 1, b = y + 1, c = x + 1,
         ↪ d = 1;
       for (int i = x; i; i -= lowbit(i)) {
            for(int j = y; j; j -= lowbit(j)) {
               ret += sm[i][j][0] * a;
               ret -= sm[i][j][1] * b;
               ret -= sm[i][j][2] * c;
               ret += sm[i][j][3] * d;
          }
       }
       return ret;
    }
} bit;

// [a, c] * [b, d] + x : add(a, b, x), add(a, d + 1, -x), add(c
  ↪ + 1, b, -x), add(c + 1, d + 1, x);
// sum of [a, c] * [b, d] : query(c, d) - query(a - 1, d) -
  ↪ query(c, b - 1) + query(a - 1, b - 1);
```

3.6 虚树

★attention：由于整个图需要用到的边与点很少，所以在每次新建虚树的时候不能全局清空，而是在把一个新的点加入栈中的时候清空这个点连过的边。
★attention：一定要在将这个点加入栈的时候清空这个点的连边情况!!!! 不能早也不能晚。

```
inline void build(int s)
{
    sort(dot+1,dot+m+1,cmp),tot=0;
    sta[tp=1]=1,hea[1]=0;
    for(int i=1,l;i<=m;i++)
    {
        if(dot[i]==1) continue; // 别忘了 1!
        l=Lca(sta[tp],dot[i]);
        if(l!=sta[tp])
        {
            while(dfn[l]<dfn[sta[tp-1]])
                add(sta[tp-1],sta[tp],dist(sta[tp]-sta[tp-1])),
                  ↪ tp--;
            if(sta[tp-1]==l)
                add(l,sta[tp],dist(sta[tp]-l)),tp--;
            else
                hea[l]=0, add(l,sta[tp],dist(sta[tp]-l)),
                  ↪ sta[tp]=l;
        }
        hea[dot[i]]=0,sta[++tp]=dot[i];
    }
    for(int i=1;i<tp;i++)
      ↪ add(sta[i],sta[i+1],dist(sta[i+1]-sta[i]));
}
```

3.7 左偏树

```
struct heap {
    struct node {
        int val, dis, ch[2];
        #define val(x) nod[x].val
        #define ls(x) nod[x].ch[0]
        #define rs(x) nod[x].ch[1]
        #define dis(x) nod[x].dis
    } nod[N]; // dis : 节点到叶子的最短距离。
    int merge(int x, int y) {
        if (!x || !y) return x | y;
        if (val(x) > val(y)) swap(x, y);
        rs(x) = merge(rs(x), y);
        if (dis(ls(x)) < dis(rs(x))) swap(ls(x), rs(x));
        dis(x) = dis(rs(x)) + 1;
    } // 若根的 dis 为 d ，则左偏树至少包含 O(2^d) 个节点，因此
      ↪ d=O(logn) 。
} hp;
```

特别的，若要求给定节点所在左偏树的根，须使用并查集。对于每个节点维护 rt[] 值，查找根时使用函数：

```
int find(int x) { return rt[x] == x ? x : rt[x] = find(rt[x]); }
```

在合并节点时，加入：

```
rt[x] = rt[y] = merge(x, y);
```

在弹出最小值时加入：

```
rt[ls(x)] = rt[rs(x)] = rt[x] = merge(ls(x), rs(x));
```

另外，删除过的点是不能复用的，因为这些点可能作为并查集的中转节点。

3.8 吉司机线段树

- 区间取 min 操作：通过维护区间次小值实现，即将区间取 min 转化为对区间最大值的加法，当要取 min 的值 v 大于次小值时停止递归。时间复杂度通过标记回收证明，即将区间最值视作标记，这样每次多余的递归等价于标记回收，总时间复杂度为 $O(m \log n)$。
- 区间历史最大值：通过维护加法标记的历史最大值实现。应该可以通过 max+ 矩乘维护。
- 区间历史版本和：使用矩阵乘法维护。由于矩乘具备结合律，历史版本和可以简单实现。

总而言之，主要的手段是标记回收和矩乘合并标记。

```cpp
void Max(auto &x, auto y) { x = max(x, y); }
void Min(auto &x, auto y) { x = min(x, y); }
struct SMT {
    #define ls (k << 1)
    #define rs (k << 1 | 1)
    #define mid ((l + r) >> 1)
    struct node {
        int sm, mx, mx2, c, hmx, ad, ad2, had, had2;
        // 区间和，最大值，次大值，最大值个数，历史最大值，最大值加法标记，其余值加法标记，最大值的历史最大加法标记，其余值的历史最大加法标记。
        node operator + (const node &x) const {
            node t = *this;
            t.sm += x.sm, Max(t.hmx, x.hmx);
            if (t.mx > x.mx) Max(t.mx2, x.mx);
            else if (t.mx < x.mx) t.mx2 = max(t.mx, x.mx2), t.mx = x.mx, t.c = x.c;
            else Max(t.mx2, x.mx2), t.c += x.c;
            t.ad = t.ad2 = t.had = t.had2 = 0;
            return t;
        }
    } nod[N << 2];
    #define sm(x) nod[x].sm
    #define mx(x) nod[x].mx
    #define mx2(x) nod[x].mx2
    #define c(x) nod[x].c
    #define hmx(x) nod[x].hmx
    #define ad(x) nod[x].ad
    #define ad2(x) nod[x].ad2
    #define had(x) nod[x].had
    #define had2(x) nod[x].had2
    void pushup(int k) { nod[k] = nod[ls] + nod[rs]; }
    void add(int k, int l, int r, int ad, int ad2, int had, int had2) {
        Max(had(k), ad(k) + had), Max(had2(k), ad2(k) + had2), Max(hmx(k), mx(k) + had);
        sm(k) += c(k) * ad + (r - l + 1 - c(k)) * ad2, mx(k) += ad, ad(k) += ad, ad2(k) += ad2, mx2(k) += ad2;
    }
    void pushdown(int k, int l, int r) {
        int mx = max(mx(ls), mx(rs));
        if (mx(ls) == mx) add(ls, l, mid, ad(k), ad2(k), had(k), had2(k));
        else add(ls, l, mid, ad2(k), ad2(k), had2(k), had2(k));
        if (mx(rs) == mx) add(rs, mid + 1, r, ad(k), ad2(k), had(k), had2(k));
        else add(rs, mid + 1, r, ad2(k), ad2(k), had2(k), had2(k));
        ad(k) = ad2(k) = had(k) = had2(k) = 0;
    }
    void build(int k, int l, int r) {
        if (l == r) return nod[k] = {a[l], a[l], -inf, 1, a[l]}, void();
        build(ls, l, mid), build(rs, mid + 1, r);
        pushup(k);
    }
    void add(int k, int l, int r, int x, int y, int v) {
        if (x <= l && r <= y) return add(k, l, r, v, v, v, v);
        pushdown(k, l, r);
        if (x <= mid) add(ls, l, mid, x, y, v);
        if (y > mid) add(rs, mid + 1, r, x, y, v);
        pushup(k);
    }
    void Min(int k, int l, int r, int x, int y, int v) {
        if (v >= mx(k)) return;
        if (x <= l && r <= y && v > mx2(k)) return add(k, l, r, v - mx(k), 0, v - mx(k), 0), void();
        pushdown(k, l, r);
        if (x <= mid) Min(ls, l, mid, x, y, v);
        if (y > mid) Min(rs, mid + 1, r, x, y, v);
        pushup(k);
    }
    node query(int k, int l, int r, int x, int y) {
        if (x <= l && r <= y) return nod[k];
        pushdown(k, l, r);
        if (y <= mid) return query(ls, l, mid, x, y);
        else if (x > mid) return query(rs, mid + 1, r, x, y);
        else return query(ls, l, mid, x, y) + query(rs, mid + 1, r, x, y);
    }
} smt;
```

3.9 树分治（点分治）

```cpp
/*** （静态）点分治 ***/
void Find1(int x,int fa)
{
    siz[x]=1,subsiz++;
    for(int i=hea[x];i;i=nex[i]) if(!used[ver[i]] && ver[i]!=fa)
        Find1(ver[i],x),siz[x]+=siz[ver[i]];
}
void Find2(int x,int fa)
{
    bool isrt=true;
    for(int i=hea[x];i;i=nex[i]) if(!used[ver[i]] && ver[i]!=fa)
    {
        Find2(ver[i],x);
        if((siz[ver[i]]<<1)>subsiz) isrt=false;
    }
    if(((subsiz-siz[x])<<1)>subsiz) isrt=false;
    if(isrt) rt=x;
}
void solve(int x)
{
    subsiz=0,Find1(x,0),Find2(x,0);
    // 处理和 rt 有关的答案
    used[rt]=true;
    for(int i=hea[rt];i;i=nex[i]) if(!used[ver[i]])
        solve(ver[i]);
}
```

熟知序列分治的过程是选取恰当的分治点并考虑所有跨过分治点的区间。而树分治的过程也是类似的，以点分治为例，每一次选择当前联通块的重心作为分治点，然后考虑所有跨越分治点的路径，并对分割出的联通块递归。

若要处理树上邻域问题，可以考虑建出点分树。处理点 x 的询问时，只需考虑 x 在点分树上到根的路径，每一次加上除开 x 所在子树的答案即可。

```cpp
int siz[N], rt, tot, dfa[N], mxp[N], vis[N], mxd;
// rt: 当前重心，tot: 联通块大小，dfa: 点分树父亲，mxp: 最大子树大小，mxd: 最大深度。
void find(int x, int fa) {
    siz[x] = 1, mxp[x] = 0;
    for (int i = head[x]; i; i = e[i].next)
    {
        if (e[i].b != fa && !vis[e[i].b])
        {
            find(e[i].b, x);
            siz[x] += siz[e[i].b];
            mxp[x] = max(mxp[x], siz[e[i].b]);
        }
    }
    mxp[x] = max(mxp[x], tot - siz[x]);
    rt = mxp[rt] > mxp[x] ? x : rt;
}
void get_dis(int x, int fa, int dep) {
    mxd = max(mxd, dep);
    for (int i = head[x]; i; i = e[i].next)
        if (e[i].b != fa && !vis[e[i].b])
            get_dis(e[i].b, x, dep + 1);
}
// bit[x][0]: 存储了以 x 为重心时 x 的子树内的信息。
// bit[x][1]: 存储了以 dfa[x] 为重心时 x 的子树内的信息。
void divide(int x, int lim) {
    vis[x] = 1, mxd = 0, get_dis(x, 0, 0);
    bit[x][0].build(mxd);
    if (dfa[x]) bit[x][1].build(lim);
    for (int i = head[x]; i; i = e[i].next) {
        if (!vis[e[i].b]) {
            rt = 0, tot = siz[e[i].b] < siz[x] ? siz[e[i].b] : lim - siz[x];
            find(e[i].b, x);
            dfa[rt] = x, divide(rt, tot);
        }
    }
}
void modify(int ct, int v) {
    int x = ct;
    while (x) {
        bit[x][0].add(dis(x, ct), v); // dis(x, y): 返回 x 到 y 的距离。
        if (dfa[x]) bit[x][1].add(dis(dfa[x], ct), v);
        x = dfa[x];
    }
} // 将点 ct 的值加上 v 。
int query(int ct, int k) {
    int x = ct, res = 0, lst = 0;
    while (x) {
        int d = dis(x, ct);
        if (d <= k) {
            res += bit[x][0].query(k - d);
            if (lst) res -= bit[lst][1].query(k - d);
        }
        lst = x, x = dfa[x];
    }
    return res;
}
```

```
57 void init() {
58     // dis 的预处理。
59     mxp[rt = 0] = tot = n, find(1, 0), divide(rt, tot);
60 }
```

3.10 树的计数

3.10.1 树的计数 Prufer 序列

prufer 编码长度为 $n-2$, 且度数为 d_i 的点在 prufer 编码中出现 d_i-1 次.
由树得到序列: 总共需要 $n-2$ 步, 第 i 步在当前的树中寻找具有最小标号的叶子节点, 将与其相连的点的标号设为 Prufer 序列的第 i 个元素 p_i, 并将此叶子节点从树中删除, 直到最后得到一个长度为 $n-2$ 的 Prufer 序列和一个只有两个节点的树.
由序列得到树: 先将所有点的度赋初值为 1, 然后加上它的编号在 Prufer 序列中出现的次数, 得到每个点的度; 执行 $n-2$ 步, 第 i 步选取具有最小标号的度为 1 的点 u 与 $v=p_i$ 相连, 得到树中的一条边, 并将 u 和 v 的度减一. 最后再把剩下的两个度为 1 的点连边, 加入到树中.
相关结论: n 个点完全图, 每个点度数依次为 d_1,d_2,...,d_n, 这样生成树的棵树为: $\frac{(n-2)!}{(d_1-1)!(d_2-1)!\ldots(d_n-1)!}$.
左边有 n_1 个点, 右边有 n_2 个点的完全二分图的生成树棵树为 $n_1^{n_2-1}\times n_2^{n_1-1}$.
m 个连通块, 每个连通块有 c_i 个点, 把他们全部连通的生成树方案数: $(\sum c_i)^{m-2}\prod c_i$

3.10.2 有根树计数 1,1,2,4,9,20,48,115,286,719,1842,4766

无标号 $a_{n+1}=1/n\sum_{k=1}^{n}(\sum_{d|k}d\cdot a(d))\cdot a(n-k+1)$

3.10.3 无根树计数

n 是奇数时, 有 $a_n-\sum_i^{n/2}a_ia_{n-i}$ 种不同的无根树.
n 时偶数时, 有 $a_n-\sum_i^{n/2}a_ia_{n-i}+\frac{1}{2}a_{n/2}(a_{n/2}+1)$ 种不同的无根树.

3.10.4 生成树计数 Kirchhoff's Matrix-Tree Theorem

Kirchhoff Matrix $T=Deg-A$, Deg 是度数对角阵, A 是邻接矩阵. 无向图度数矩阵是每个点度数; 有向图度数矩阵是每个点入度.
邻接矩阵 $A[u][v]$ 表示 $u\to v$ 边个数, 重边按照边数计算, 自环不计入度数.
无向图生成树计数: $c=|K$ 的任意 1 个 $n-1$ 阶主子式 $|$
有向图外向树计数: $c=|$ 去掉根所在的那阶得到的主子式 $|$

3.10.5 有向图欧拉回路计数 BEST Thoerem

$$\mathrm{ec}(G)=t_w(G)\prod_{v\in V}(\deg(v)-1)!$$

其中 deg 为入度 (欧拉图中等于出度), $t_w(G)$ 为以 w 为根的外向树的个数. 相关计算参考生成树计数.
欧拉连通图中任意两点外向树个数相同: $\mathrm{t_v}(G)=\mathrm{t_w}(G)$.

4 字符串

4.1 Hash 类

```cpp
const ll P = (ll)1e18+9;
static constexpr u128 inv = [](){
    u128 ret = P;
    for(int i=0;i<6;i++)ret*=2-ret*P;
    return ret;
}();
constexpr u128 chk = u128(-1) / P;
bool check(i128 a,i128 b){
    if(a<b)swap(a,b);
    return (a-b)*inv<=chk;
}
```

4.2 后缀数组（llx）

```cpp
// 0-下标, h[i] 表示后缀 sa[i] 和 sa[i+1] 的 LCP 长度
struct SuffixArray {
    int n;
    vector<int> sa, rk, h;
    SuffixArray(const string &s) {
        n = s.length();
        sa.resize(n);
        h.resize(n - 1);
        rk.resize(n);
        iota(sa.begin(), sa.end(), 0);
        sort(sa.begin(), sa.end(), [&](int a, int b) {return s[a] < s[b];});
        rk[sa[0]] = 0;
        for (int i=1;i<n;++i)
            rk[sa[i]]=rk[sa[i-1]]+(s[sa[i]]!=s[sa[i-1]]);
        int k = 1;
        vector<int> tmp, cnt(n);
        tmp.reserve(n);
        while (rk[sa[n - 1]] < n - 1) {
            tmp.clear();
            for (int i=0;i<k;i++) tmp.push_back(n-k+i);
            for (auto i:sa)if(i>=k) tmp.push_back(i-k);
            fill(cnt.begin(), cnt.end(), 0);
            for (int i=0;i<n;i++) cnt[rk[i]]++;
            for (int i=1;i<n;i++) cnt[i]+=cnt[i-1];
            for (int i=n-1;i>=0;i--)
                sa[--cnt[rk[tmp[i]]]]=tmp[i];
            swap(rk, tmp);
            rk[sa[0]] = 0;
            for (int i=1;i<n;i++){
                rk[sa[i]]=rk[sa[i-1]];
                if(tmp[sa[i-1]]<tmp[sa[i]] || sa[i-1]+k==n || tmp[sa[i-1]+k]<tmp[sa[i]+k]) rk[sa[i]]++;
            }
            k *= 2;
        }
        for(int i=0,j=0;i<n;i++){
            if(rk[i]==0) j=0;
            else{
                if(j) j--;
                while(i+j<n && sa[rk[i]-1]+j<n && s[i+j]==s[sa[rk[i]-1]+j])++j;
                h[rk[i]-1] = j;
            }
        }
    }
};

// 用 ST 表 O(1) 求 LCP
vector<vector<int>> st(n-1, vector<int>(20));
for(int i=0;i<n-1;i++)st[i][0]=t.h[i];
for(int j=1;j<20;j++){
    for(int i=0;i+(1<<(j))-1<n-1;i++){
        st[i][j] = min(st[i][j-1], st[i+(1<<(j-1))][j-1]);
    }
}
auto query = [&](int l,int r){
    if(l==r)return n-r+1;
    if(l>r)swap(l,r);
    --r;
    int o = __lg(r-l+1);
    return min(st[l][o], st[r-(1<<o)+1][o]);
};
auto lcp = [&](int l1,int r1,int l2,int r2)->int {
    int x = query(t.rk[l1], t.rk[l2]);
    return min(x, min(r1-l1+1, r2-l2+1));
};
```

4.3 后缀数组与后缀树（旧）

```cpp
const int N = 1e6 + 5;

char s[N];
int sa[N], rk[N], n, h[N];
// 后缀数组。h[i] = lcp(sa[i], sa[i - 1])
int rt, ls[N], rs[N], fa[N], val[N];
// 后缀树。实际上就是 height 数组的笛卡尔树。
// val[x] : x 与 fa[x] 对应的子串等价类的大小之差，也就是 x 贡献的本质不同子串数。

struct suffix {
    int k1[N], k2[N << 1], cnt[N], mx, stk[N], top;
    void radix_sort() {
        for (int i = 1; i <= mx; i++) cnt[i] = 0;
        for (int i = 1; i <= n; i++) cnt[k1[i]]++;
        for (int i = 1; i <= mx; i++) cnt[i] += cnt[i - 1];
        for (int i = n; i >= 1; i--) sa[cnt[k1[k2[i]]]--] = k2[i];
    } // 基数排序
    void sort() {
        mx = 'z';
        for (int i = 1; i <= n; i++) k1[i] = s[i], k2[i] = i;
        radix_sort();
        for (int j = 1; j <= n; j <<= 1) {
            int num = 0;
            for (int i = n - j + 1; i <= n; i++) k2[++num] = i;
            for (int i = 1; i <= n; i++) if (sa[i] > j) k2[++num] = sa[i] - j;
            radix_sort();
            for (int i = 1; i <= n; i++) k2[i] = k1[i];
            k1[sa[1]] = mx = 1;
            for (int i = 2; i <= n; i++) k1[sa[i]] = k2[sa[i]] == k2[sa[i - 1]] && k2[sa[i] + j] == k2[sa[i - 1] + j] ? mx : ++mx;
        }
    } // 后缀排序
    void height() {
        for (int i = 1; i <= n; i++) rk[sa[i]] = i;
        int k = 0;
        for (int i = 1; i <= n; i++) {
            if (k) k--;
            if (rk[i] == 1) continue;
            int j = sa[rk[i] - 1];
            while (i + k <= n && j + k <= n && s[i + k] == s[j + k]) k++;
            h[rk[i]] = k;
        }
    } // 计算 height 数组
    void build() {
        if (n == 1) return rt = 1, void();
        ls[2] = n + 1, rs[2] = n + 2, fa[ls[2]] = fa[rs[2]] = rt = stk[++top] = 2;
        for (int i = 3; i <= n; i++) {
            while (top && h[stk[top]] > h[i]) top--;
            int p = stk[top];
            if (top) ls[i] = rs[p], fa[rs[p]] = i, rs[p] = i, fa[i] = p;
            else ls[i] = rt, fa[rt] = i, rt = i;
            rs[i] = n + i, fa[rs[i]] = i, stk[++top] = i;
        }
        for (int i = 2; i <= n + n; i++) val[i] = (i > n ? n - sa[i - n] + 1 : h[i]) - h[fa[i]];
    } // 构建后缀树
} SA;
```

4.4 AC 自动机

```cpp
struct ACAM {
    int ch[N][26], cnt, fail[N], vis[N];
    queue <int> q;
    void insert(char *s, int n, int id) {
        int x = 0;
        for (int i = 1; i <= n; i++) {
            int nw = s[i] - 'a';
            if (!ch[x][nw]) ch[x][nw] = ++cnt;
            x = ch[x][nw];
        }
        vis[x]++;
    }
    void build() {
        for (int i = 0; i < 26; i++) {
            if (ch[0][i]) q.push(ch[0][i]);
        }
        while (!q.empty()) {
            int x = q.front(); q.pop();
            for (int i = 0; i < 26; i++) {
                if (ch[x][i]) {
                    fail[ch[x][i]] = ch[fail[x]][i];
```

```
                    q.push(ch[x][i]);
                } else ch[x][i] = ch[fail[x]][i];
            }
        }
    }
    void match(char *s, int n) {
        int x = 0;
        for (int i = 1; i <= n; i++) {
            int nw = s[i] - 'a';
            x = ch[x][nw];
        }
    }
} AC;
```

4.5 回文自动机

```
struct PAM {
   int fail[N], ch[N][26], len[N], s[N], tot, cnt, lst;
    // fail : 当前节点的最长回文后缀。
    // ch : 在当前节点的前后添加字符，得到的回文串。
   PAM() {
      len[0] = 0, len[1] = -1, fail[0] = 1;
      tot = lst = 0, cnt = 1, s[0] = -1;
   }
   int get_fail(int x) {
      while (s[tot - 1 - len[x]] != s[tot]) x = fail[x];
      return x;
   }
   void insert(char c) {
      s[++tot] = c - 'a';
      int p = get_fail(lst);
      if(!ch[p][s[tot]]) {
         len[++cnt] = len[p] + 2;
         int t = get_fail(fail[p]);
         fail[cnt] = ch[t][s[tot]];
         ch[p][s[tot]] = cnt;
      }
      lst=ch[p][s[tot]];
   }
} pam;
```

4.6 Manacher 算法

```
char s[N], t[N];
// s[] : 原串，t[] : 加入分割字符的串，这样就只需考虑奇回文串了。
int mxp, cen, r[N], n;
// mxp : 最右回文串的右端点的右侧，cen : 最右回文串的中心，r[i] : 以位置 i 为中心的回文串半径，即回文串的长度一半向上取整。

void manacher() {
    t[0] = '~', t[1] = '#';
    // 在 t[0] 填入特殊字符，防止越界。
    int m = 1;
    for (int i = 1; i <= n; i++) {
        t[++m] = s[i], t[++m] = '#';
    }
    for (int i = 1; i <= m; i++) {
        r[i] = mxp > i ? min(r[2 * cen - i], mxp - i) : 1;
        // 若 i (cen, mxp)，则由对称性 r[i] 至少取 min(r[2 * cen - i], mxp - i)。否则直接暴力扩展。
        while (t[i + r[i]] == t[i - r[i]]) r[i]++;
        if (i + r[i] > mxp) mxp = i + r[i], cen = i;
    }
}
```

4.7 KMP 算法与 border 理论

```
char s[N], t[N];
int nex[N], n, m;

void kmp() {
    int j = 0;
    for (int i = 2; i <= n; i++) {
        while (j && s[j + 1] != s[i]) j = nex[j];
        if (s[j + 1] == s[i]) j++;
        nex[i] = j;
    }
}

void match() {
    int j = 0;
    for (int i = 1; i <= m; i++) {
        while (j && s[j + 1] != t[i]) j = nex[j];
        if (s[j + 1] == t[i]) j++;
        if (j == n) {
            //match.
            j = nex[j];
        }
    }
}
```

字符串的 border 理论：以下记字符串 S 的长度为 n 。

- 若串 S 具备长度为 m 的 border ，则其必然具备长度为 $n-m$ 的周期，反之亦然。
- 弱周期性引理：若串 S 存在周期 p 、q ，且 $p+q \le n$ ，则 S 必然存在周期 $\gcd(p,q)$ 。
- 引理 1：若串 S 存在长度为 m 的 border T，且 T 具备周期 p ，满足 $2m-n \ge p$ ，则 S 同样具备周期 p 。
- 周期性引理：若串 S 存在周期 p 、q ，满足 $p+q-\gcd(p,q) \le n$ ，则串 S 必然存在周期 $\gcd(p,q)$ 。
- 引理 2：串 S 的所有 border 的长度构成了 $O(\log n)$ 个不交的等差数列。更具体的，记串 S 的最小周期为 p ，则其所有长度包含于区间 $[n \bmod p+p, n)$ 的 border 构成了一个等差数列。
- 引理 3：若存在串 S 、T ，使得 $2|T| \ge n$ ，则 T 在 S 中的所有匹配位置构成了一个等差数列。
- 引理 4：PAM 的失配链可以被划分为 $O(\log n)$ 个等差数列。

4.8 Z 函数

Z 函数用于求解字符串的每一个后缀与其本身的 lcp 。其思路和 manacher 算法基本一致，都是维护一个扩展过的最右端点和对应的起点，而当前点要么暴力扩展使最右端点右移，要么处在记录的起点和终点间，从而可以利用已有的信息快速转移。

```
char s[N];
int z[N];

void zfunc() {
    z[1] = n;
    int j = 0;
    for (int i = 2; i <= n; i++) {
        if (j && j + z[j] - 1 >= i) z[i] = min(z[i - j + 1], j + z[j] - i);
        while (i + z[i] <= n && s[i + z[i]] == s[1 + z[i]]) z[i]++;
        if (i + z[i] > j + z[j]) j = i;
    }
}
```

4.9 后缀自动机

```
struct SAM{ // 注意修改字符集！字符集是小写字母吗？
    int last = 1, tot = 1;
    int ch[N<<1][26], len[N<<1], f[N<<1];
    void ins(char c){
        c -= 'a';
        int p = last, cur = last = ++tot;
        len[cur]=len[p]+1;
        for(;p&&!ch[p][c];p=f[p])ch[p][c]=cur;
        if(!p){f[cur]=1;return;}
        int q=ch[p][c];
        if(len[q]==len[p]+1){f[cur]=q;return;}
        int clone=++tot;
        for(int i=0;i<26;i++)ch[clone][i]=ch[q][i];
        f[clone]=f[q],len[clone]=len[p]+1;
        f[q]=f[cur]=clone;
        for(;p&&ch[p][c]==q;p=f[p])ch[p][c]=clone;
    }
}sam; // 注意任何跟 SAM 有关的数组都要开两倍
```

4.10 后缀自动机 (map 版)

```
struct SAM{ // 注意修改字符集！字符集是小写字母吗？
    int last = 1, tot = 1;
    map<int,int> ch[N<<1];
    int len[N<<1], f[N<<1];
    vector<int> G[N<<1];
    void ins(char c){
        c -= 'a';
        int p = last, cur = last = ++tot;
        len[cur]=len[p]+1;
        for(;p&&!ch[p].count(c);p=f[p])ch[p][c]=cur;
        if(!p){f[cur]=1;return;}
```

```
        int q=ch[p][c];
        if(len[q]==len[p]+1){f[cur]=q;return;}
        int clone=++tot;
        ch[clone]=ch[q];
        f[clone]=f[q],len[clone]=len[p]+1;
        f[q]=f[cur]=clone;
        for(;p&&ch[p].count(c)&&ch[p][c]==q;p=f[p])
          ↪ ch[p][c]=clone;
    }
}sam; // 注意任何跟 SAM 有关的数组都要开两倍
```

4.11 最小表示法

```
// 0-下标的!
int minpos(const string& s){
    int n = s.size();
    int k = 0, i = 0, j = 1;
    while (k < n && i < n && j < n) {
        if (s[(i+k)%n]==s[(j+k)%n])++k;
        else{
            if(s[(i + k) % n]>s[(j + k) % n]) i+=k+1;
            else j+=k+1;
            k = 0;
            if(i==j)i++;
        }
    }
    return min(i,j);
}
```

5 线性代数

5.1 高斯消元

```
scanf("%d", &n);
for (int i = 1; i <= n; i++)
    for (int j = 1; j <= n + 1; j++)
        scanf("%lf", &a[i][j]);
for (int i = 1, Max = 1; i <= n; Max = ++i) {
    for (int s = i + 1; s <= n; s++)
        if (fabs(a[s][i]) > fabs(a[Max][i]))
            Max = s;  // 找出绝对值最大的
    for (int j = 1; j <= n + 1; j++)
        swap(a[i][j], a[Max][j]);
    if (a[i][i] < 10e-8 && a[i][i] > -10e-8) {
        p = false;
        break;
    }  // 记得 double 的精度问题
    for (int s = 1; s <= n; s++)
        if (s != i)  // 这样省去了第二步处理的麻烦
        {
            double tmp = 0 - (a[s][i] / a[i][i]);
            a[s][i] = 0;
            for (int j = i + 1; j <= n + 1; j++)
                a[s][j] += tmp * a[i][j];
        }
}
if (p)
    for (int i = 1; i <= n; i++)
        printf("%.2lf\n", a[i][n + 1] / a[i][i]);
else
    printf("No Solution\n");
```

5.2 线性基

```
void Insert(ll x) // 加入一个数
{
    for (int i = 62; i >= 0; i--)
        if ((x >> i) & 1) {
            if (!p[i]) {
                p[i] = x;
                break;
            }
            x ^= p[i]; // 更新 [0,i-1] 位的更优答案
        }
}
bool exist(ll x) // 查询一个元素是否可以被异或出来
{
    for (int i = 62; i >= 0; i--)
        if ((x >> i) & 1) x ^= p[i];
    return x == 0;
}
ll query_max() // 查询异或最大值
{
    ll ret = 0;
    for (int i = 62; i >= 0; i--)
        if ((ans ^ p[i]) > ans) ans ^= p[i];
    return ans;
}
ll query_min() // 查询异或最小值
{
    for (int i = 0; i <= 62; i++)
        if (p[i]) return p[i];
    return 0;
}
ll kth(ll k) // 查询异或第 k 小
{
    // 重建 d 数组，求出哪些位可以被异或为 1
    // d[i] 中任意两个数在任意一个二进制位上不可能同时为 1
    for (int i = 62; i >= 1; i--) // 从高到低防止后效性
        for (int j = i - 1; j >= 0; j--)
            if ((p[i] >> j) & 1) p[i] ^= p[j];
    for (int i = 0; i <= 62; i++)
        if (p[i]) d[cnt++] = p[i];

    if (!k) return 0ll;                    // 特判 0
    if (k >= (1ll << (ll)cnt)) return -1ll; // k 大于可以表示出的
      ↪ 数的个数
    ll ret = 0;
    for (int i = 62; i >= 0; i--)
        if ((k >> i) & 1) ret ^= d[i];
    return ret;
}
ll Rank(ll k) {
    ll Now = 1, ret = 0;
    for (int i = 0; i <= 62; i++)
        if (p[i]) // 记得保证 k 可以被异或出来
        {
            if ((k >> i) & 1) ret += Now;
            Now <<= 1;
        }
    return ret;
}
```

5.3 行列式

容易证明转置后行列式相等，积的行列式等于行列式的积。但是这样的性质并不对和成立，而每次只能拆一行或一列。

以下是任意模数求行列式的算法：

```
// 解法 1：类似辗转相除
n = rd(), mod = rd();
for (int i = 1; i <= n; i++)
    for (int j = 1; j <= n; j++)
        a[i][j] = rd() % mod;
for (int i = 1; i <= n; i++) {
    for (int j = i + 1; j <= n; j++) {
        while (a[j][i]) {
            ll tmp = a[i][i] / a[j][i];
            for (int k = i; k <= n; k++)
                a[i][k] = (a[i][k] - tmp * a[j][k] % mod + mod)
                  ↪ % mod;
            swap(a[i], a[j]), w = -w; // ?
        }
    }
}
for (int i = 1; i <= n; i++)
    ans = ans * a[i][i] % mod;
printf("%lld\n", (mod + w * ans) % mod);
```

5.4 矩阵树定理

以下叙述允许重边，不允许自环。

对于无向图 G，定义度数矩阵 D 为：

$$D_{ij} = \deg(i)[i = j]$$

设 $\#e(i,j)$ 为连接点 i 和 j 的边数，定义邻接矩阵 A 为：

$$A_{ij} = \#e(i,j)$$

显然 $A_{ii} = 0$。定义 Laplace 矩阵 L 为 $D - A$，记 G 的生成树个数为 $t(G)$，则其恰为 L 的任意一个 $n-1$ 阶主子式的值。

对于有向图 G，分别定义出度矩阵 D^{out} 和入度矩阵 D^{in} 为：

$$D^{out}_{ij} = \deg^{out}(i)[i = j]$$
$$D^{in}_{ij} = \deg^{in}(i)[i = j]$$

设 $\#e(i,j)$ 为从点 i 到 j 的边数，定义邻接矩阵 A 为：

$$A_{ij} = \#e(i,j)$$

显然 $A_{ii} = 0$。再分别定义出度 Laplace 矩阵 L^{out} 和入度 Laplace 矩阵 L^{in} 为：

$$L^{out} = D^{out} - A$$
$$L^{in} = D^{in} - A$$

分别记 G 的以 k 为根的根向树形图个数为 $t^{root}(k)$，以及以 k 为根的叶向树形图个数为 $t^{leaf}(k)$。则 $t^{root}(k)$ 恰为 L^{out} 的删去 k 行 k 列的 $n-1$ 阶主子式的值；$t^{leaf}(k)$ 恰为 L^{in} 的删去 k 行 k 列的 $n-1$ 阶主子式的值。

5.5 单纯形法

线性规划的标准型：

$$\begin{aligned} \text{maximize}: \quad & c^{\mathrm{T}}x \\ \text{constraints}: \quad & Ax \le b \\ & x \ge 0 \end{aligned}$$

在标准型的基础上得到松弛型：

$$\begin{aligned} \text{maximize}: \quad & c^{\mathrm{T}}x \\ \text{constraints}: \quad & \alpha = b - Ax \\ & \alpha, x \geq 0 \end{aligned}$$

单纯性法以松弛型为基础。具体的，松弛型隐含了一个基本解，即 $x = 0$ ，$\alpha = b$ （这里要求 $b \geq 0$ ）。我们称 α 中的变量为基变量，其余为非基变量。单纯性的主过程被称作 pivot 操作。一次 pivot 操作的本质就是进行基变量与非基变量之间的变换以使得带入基本解的目标函数更大。具体的，我们每一次选定一个在目标函数中系数为正的变量为换入变量，再选择对这个换入变量约束最紧的的线性约束所对应的基变量，称其为换出变量。然后，我们将换入变量和换出变量分别换为基变量和非基变量，并对其余的式子做出对应的代换以使得定义满足即可。另外单纯型法的时间复杂度虽然是指数级别的，但是跑起来效果还是很好的，期望迭代次数貌似可以大致看作约束个数的平方级别。

```cpp
typedef double db;
const db eps = 1e-8, inf = 1e9;
int n, m, ans;
db b[N], a[N][M], c[M]; // n: 约束个数。m: 变量个数。
bool dcmp(db x) { return fabs(x) > eps; }

db pivot(int out, int in) {
    b[out] /= a[out][in];
    for (int i = 1; i <= m; i++) if (i != in) a[out][i] /= a[out][in];
    a[out][in] = 1; // 理论上是 1 / a[out][in] ，但这个系数可以任取，但也不要随便取。
    for (int i = 1; i <= n; i++) {
        if (i != out && dcmp(a[i][in])) {
            b[i] -= a[i][in] * b[out];
            for (int j = 1; j <= m; j++)
                if (j != in) a[i][j] -= a[i][in] * a[out][j];
            a[i][in] *= -a[out][in];
        }
    }
    db res = c[in] * b[out];
    for (int i = 1; i <= m; i++) if (i != in) c[i] -= a[out][i] * c[in];
    c[in] *= -a[out][in];
    return res;
}
void simplex() {
    db res = 0;
    while (1) {
        int in = 0;
        for (int i = 1; i <= m; i++) {
            if (c[i] > eps) {
                in = i;
                break;
            }
        }
        if (!in) break; // simplex 完成，找到最优解。
        int out = 0;
        db mn = inf;
        for (int i = 1; i <= n; i++)
            if (a[i][in] > eps && b[i] / a[i][in] < mn) mn = b[i] / a[i][in], out = i;
        if (!out) {
            res = inf;
            break;
        } // 解为无穷大。
        res += pivot(out, in);
    }
    ans = round(res);
}
```

5.6 全幺模矩阵

当一个矩阵的任意一个子方阵的行列式都为 $\pm 1, 0$ 时，我们称这个矩阵是全幺模的。

如果单纯形矩阵是全幺模的，那么单纯形就具有整数解。

5.7 对偶原理

线性规划的对偶原理：原线性规划与对偶线性规划的最优解相等。即：

$$\begin{aligned} \text{minimize}: \quad & c^{\mathrm{T}}x & & \text{maximize}: \quad & b^{\mathrm{T}}y \\ \text{constraints}: \quad & Ax \geq b & \text{dual} \quad & \text{constraints}: \quad & A^{\mathrm{T}}y \leq c \\ & x \geq 0 & & & y \geq 0 \end{aligned}$$

直观上看，对于一个最小化的线性规划，我们尝试构造一个最大化的线性规划，使得它们目标函数的最优解相同。具体的，为每个约束设置一个非负的新变量，代表其系数。对于每个原变量，其对应了一个新约束，要求原约束的线性组合的对应系数不大于原目标函数的系数，从而得到原目标函数的下界。而新目标函数则要使得原约束的组合最大化，从而得到最紧的下界。而线性规划对偶性则指出，原线性规划的最优解必然与对偶线性规划的最优解相等。

对偶线性规划具备互补松弛性。即，设 x 和 y 分别为原问题与对偶问题的可行解，则 x 和 y 均为最优解，当且仅当以下两个命题同时成立：

$$\forall j \in [1, m], x_j = 0 \vee \sum_{i=1}^{n} a_{ij} y_i = c_j$$

$$\forall i \in [1, n], y_i = 0 \vee \sum_{j=1}^{m} a_{ij} x_j \leq b_i$$

对偶松弛性的意义是，其指出若最优解中的变量不取 0 ，则对应约束在最优解中一定取等。

6 多项式

6.1 FFT

```cpp
struct comp {
    double x, y;
    comp(double X = 0, double Y = 0) { x = X, y = Y; }
    comp operator + (const comp &rhs) const { return comp(x + rhs.x, y + rhs.y); }
    comp operator - (const comp &rhs) const { return comp(x - rhs.x, y - rhs.y); }
    comp operator * (const comp &rhs) const { return comp(x * rhs.x - y * rhs.y, x * rhs.y + y * rhs.x); }
    comp operator / (const comp &rhs) const {
        double t = rhs.x * rhs.x + rhs.y * rhs.y;
        return {(x * rhs.x + y * rhs.y) / t, (y * rhs.x - x * rhs.y) / t};
    }
};
const double pi = acos(-1);
int lim, tr[N];
void adjust(int n) { // n: 多项式的次数。
    lim = 1;
    while (lim <= n) lim <<= 1;
    for (int i = 0; i < lim; i++) tr[i] = (tr[i >> 1] >> 1) | ((i & 1) ? lim >> 1 : 0);
} // 准备蝶形变换。
void fft(comp *f, int op) { // op: 1 为 dft , -1 为 idft 。
    for (int i = 0; i < lim; i++) if (tr[i] < i) swap(f[tr[i]], f[i]);
    for (int l = 1; l < lim; l <<= 1) {
        comp w1(cos(2 * pi / (l << 1)), sin(2 * pi / (l << 1)) * op);
        for (int i = 0; i < lim; i += l << 1) {
            comp w(1, 0);
            for (int j = i; j < i + l; j++, w = w * w1) {
                comp x = f[j], y = f[j + l] * w;
                f[j] = x + y, f[j + l] = x - y;
            }
        }
    }
    if (op == -1) {
        for (int i = 0; i < lim; i++) f[i].x /= lim;
    }
}
```

6.2 NTT

```cpp
const int mod = 998244353, G = 3, iG = 332748118;
int tr[N], lim;
void adjust(int n) { // n: 多项式的次数。
    lim = 1;
    while (lim <= n) lim <<= 1;
    for (int i = 0; i < lim; i++) tr[i] = (tr[i >> 1] >> 1) | ((i & 1) ? lim >> 1 : 0);
} // 准备蝶形变换。
void ntt(int *f, int op) { // op: 1 为 dft , -1 为 idft 。
    for (int i = 0; i < lim; i++) if (tr[i] < i) swap(f[tr[i]], f[i]);
    for (int l = 1; l < lim; l <<= 1) {
        int w1 = qpow(op == 1 ? G : iG, (mod - 1) / (l << 1)); // qpow: 快速幂。
        for (int i = 0; i < lim; i += l << 1) {
            for (int j = i, w = 1; j < i + l; j++, (w *= w1) %= mod) {
                int x = f[j], y = f[j + l] * w % mod;
                f[j] = (x + y) % mod, f[j + l] = (x - y) % mod;
            }
        }
    }
    if (op == -1) {
        int iv = qpow(lim); // 算逆元。
        for (int i = 0; i < lim; i++) (f[i] *= iv) %= mod;
    }
}
```

6.3 集合幂级数

6.3.1 并卷积、交卷积与子集卷积

集合并等价于二进制按位或，因此并卷积的计算实际上就是做高维前缀和以及差分，也被称作莫比乌斯变换。

```cpp
void fmt(int *f, int op) { // op: 1 为 dft , -1 为 idft 。
    for (int l = 1; l < lim; l <<= 1) {
        for (int i = 0; i < lim; i += l << 1) {
            for (int j = i; j < i + l; j++) {
                f[j + l] += f[j] * op;
            }
        }
    }
}
```

而集合交卷积则对应后缀和。

```cpp
void fmt(int *f, int op) { // op: 1 为 dft , -1 为 idft 。
    for (int l = 1; l < lim; l <<= 1) {
        for (int i = 0; i < lim; i += l << 1) {
            for (int j = i; j < i + l; j++) {
                f[j] += f[j + l] * op;
            }
        }
    }
}
```

子集卷积则较为特殊，为了使得产生贡献的集合没有交集，考虑引入代表集合大小的占位符。这样只需做 n 次 FMT ，再枚举长度做 n^2 次卷积。因为 FMT 具备线性性，所以最后只需做 n 次 iFMT 即可。

```cpp
void multi(int *f, int *g, int *h) { // 对 f 和 g 做子集卷积，答案为 h 。
    static int a[n][lim], b[n][lim], c[n][lim]; // lim = 1 << n
    memset(a, 0, sizeof a);
    memset(b, 0, sizeof b);
    memset(c, 0, sizeof c);
    for (int i = 0; i < lim; i++) a[pcnt[i]][i] = f[i]; // pcnt[i] = popcount(i)
    for (int i = 0; i < lim; i++) b[pcnt[i]][i] = g[i];
    for (int i = 0; i <= n; i++) fmt(a[i], 1), fmt(b[i], 1);
    for (int i = 0; i <= n; i++)
        for (int j = 0; j <= i; j++)
            for (int k = 0; k < lim; k++)
                c[i][k] += a[j][k] * b[i - j][k];
    for (int i = 0; i <= n; i++) fmt(c[i], -1);
    for (int i = 0; i < lim; i++) h[i] = c[pcnt[i]][i];
}
```

特别的，子集卷积等价于 n 元保留到一次项的线性卷积。

6.3.2 对称差卷积

集合对称差等价于按位异或，而异或卷积则等价于 n 元模 2 的循环卷积，因此，FWT 实质上和 n 元 FFT 没有什么区别。

```cpp
void fwt(int *f, int op) { // op: 1 为 dft , -1 为 idft 。
    for (int l = 1; l < lim; l <<= 1) {
        for (int i = 0; i < lim; i += l << 1) {
            for (int j = i; j < i + l; j++) {
                int x = f[j], y = f[j + l];
                f[j] = x + y, f[j + l] = x - y;
                if (op == -1) f[j] /= 2, f[j + l] /= 2;
                // 模意义下改成乘逆元。
            }
        }
    }
}
```

6.4 多项式全家桶

```cpp
const int N = 5000005;
const long long mod = 998244353;
#define int long long
namespace poly {
    const long long G = 3, iG = 332748118;
    int tr[N], lim;
    int qpow(int a, int b = mod - 2) {
        int res = 1;
        while (b) {
            if (b & 1) (res *= a) %= mod;
            (a *= a) %= mod, b >>= 1;
        }
        return res;
    }
    void adjust(int n) {
        lim = 1;
        while (lim <= n) lim <<= 1;
        for (int i = 0; i < lim; i++) tr[i] = (tr[i >> 1] >> 1) | ((i & 1) ? lim >> 1 : 0);
    } // 准备蝶形变换
```

```
    void copy(int *f, int *g, int n) { for (int i = 0; i <= n; i++) f[i] = g[i]; }
    void clear(int *f, int n) { for (int i = 0; i <= n; i++) f[i] = 0; }
    void erase(int *f, int l, int r) { for (int i = l; i <= r; i++) f[i] = 0; }
    void reverse(int *f, int n) { for (int i = 0; i <= n / 2; i++) swap(f[i], f[n - i]); }
    void integral(int *f, int n) {
        for (int i = n + 1; i >= 1; i--) f[i] = f[i - 1] * qpow(i) % mod;
        f[0] = 0;
    } // 对 n 次多项式 f 求积分，默认常数项为 0 。
    void dif(int *f, int n) {
        for (int i = 0; i < n; i++) f[i] = f[i + 1] * (i + 1) % mod;
        f[n] = 0;
    } // 对 n 次多项式 f 求导。
    void ntt(int *f, int op) { // op: 1 为 dft , -1 为 idft 。
        for (int i = 0; i < lim; i++) if (tr[i] < i) swap(f[tr[i]], f[i]);
        for (int l = 1; l < lim; l <<= 1) {
            int w1 = qpow(op == 1 ? G : iG, (mod - 1) / (l << 1)); // qpow: 快速幂。
            for (int i = 0; i < lim; i += l << 1) {
                for (int j = i, w = 1; j < i + l; j++, (w *= w1) %= mod) {
                    int x = f[j], y = f[j + l] * w % mod;
                    f[j] = (x + y) % mod, f[j + l] = (x - y) % mod;
                }
            }
        }
        if (op == -1) {
            int iv = qpow(lim); // 算逆元。
            for (int i = 0; i < lim; i++) (f[i] *= iv) %= mod;
        }
    }
    void multiply(int *h, int *f, int n, int *g, int m) {
        static int a[N], b[N];
        copy(a, f, n), copy(b, g, m);
        adjust(n + m);
        ntt(a, 1), ntt(b, 1);
        for (int i = 0; i < lim; i++) h[i] = a[i] * b[i] % mod;
        ntt(h, -1);
        clear(a, lim - 1), clear(b, lim - 1);
    } // 计算 f 与 g 的积，存放在 h 中，f 与 g 不变。
    void inverse(int *f, int *g, int n) {
        static int t[N];
        if (!n) return f[0] = qpow(g[0]), void();
        inverse(f, g, n >> 1);
        adjust(2 * n), copy(t, g, n);
        ntt(t, 1), ntt(f, 1);
        for (int i = 0; i < lim; i++) f[i] = f[i] * (2 - f[i] * t[i] % mod) % mod;
        ntt(f, -1), erase(f, n + 1, lim - 1), clear(t, lim - 1);
    } // 计算 g 的 n 次逆，存放在 f 中，g 不变。不要让 f 和 g 为同一个数组。
    void ln(int *f, int *g, int n) {
        static int t[N];
        copy(t, g, n), inverse(f, g, n), dif(t, n);
        adjust(n * 2), ntt(t, 1), ntt(f, 1);
        for (int i = 0; i < lim; i++) (f[i] *= t[i]) %= mod;
        ntt(f, -1), integral(f, n);
        erase(f, n + 1, lim - 1), clear(t, lim - 1);
    } // 要求 g[0] = 1 。
    void exp(int *f, int *g, int n) {
        static int t[N];
        if (!n) return f[0] = 1, void();
        exp(f, g, n >> 1), ln(t, f, n);
        for (int i = 0; i <= n; i++) t[i] = (g[i] - t[i]) % mod;
        t[0]++;
        adjust(n * 2), ntt(f, 1), ntt(t, 1);
        for (int i = 0; i < lim; i++) (f[i] *= t[i]) %= mod;
        ntt(f, -1), clear(t, lim - 1), erase(f, n + 1, lim - 1);
    } // 要求 g[0] = 0 。
    void pow(int *f, int *g, int n, int k) {
        static int t[N];
        ln(t, g, n);
        for (int i = 0; i <= n; i++) (t[i] *= k) % mod;
        exp(f, t, n);
    } // 要求 g[0] = 1 。
    void divide(int *q, int *r, int *f, int n, int *g, int m) {
        static int a[N], b[N], c[N];
        copy(a, f, n), copy(b, g, m);
        reverse(a, n), reverse(b, m);
        inverse(c, b, n - m);
        multiply(q, a, n - m, c, n - m);
        reverse(q, n - m);
        multiply(a, g, m, q, n - m);
        for (int i = 0; i < m; i++) r[i] = (f[i] - a[i] + mod) % mod;
    } // 多项式带余除法，其中 q 为商，r 为余数。
}
using namespace poly;
```

7 数论

7.1 中国剩余定理

$$\begin{cases} x \equiv a_1 \pmod{m_1} \\ x \equiv a_2 \pmod{m_2} \\ \cdots \\ x \equiv a_n \pmod{m_n} \end{cases}$$

求解 x ，其中 $m_1, m_2, \ldots m_n$ 互素。

$$x \equiv \sum_{i=1}^{n} a_i \prod_{j \neq i}^{n} m_j \times \left(\left(\prod_{j \neq i}^{n} m_j \right)^{-1} \pmod{m_i} \right) \pmod{\prod_{i=1}^{n} m_i}$$

7.2 扩展中国剩余定理

用于求解同余方程组的模数并不互素的情况。我们考虑如何合并两个同余式：

$$\begin{cases} x \equiv a_1 \pmod{m_1} \\ x \equiv a_2 \pmod{m_2} \end{cases}$$

显然其等价于：

$$\begin{cases} x = a_1 + k_1 \times m_1 \\ x = a_2 + k_2 \times m_2 \end{cases}$$

联立可得：

$$k_1 m_1 - k_2 m_2 = a_2 - a_1$$

我们解这个方程即可得出当前的解 x_0 。且注意到我们若给 x_0 加上若干个 $\operatorname{lcm}(m_1, m_2)$ ，上式仍然成立，即当前的解是在 $\bmod \operatorname{lcm}(m_1, m_2)$ 意义下的。这样我们得出新的同余式：

$$x \equiv x_0 \pmod{\operatorname{lcm}(m_1, m_2)}$$

与其它式子继续合并即可。注意在数据范围比较大的时候需要龟速加。

7.3 BSGS

在 $\sqrt{p}$ 的时间内求解 $a^x \equiv b \pmod{p}$，要求 a 与 p 互质。

```
inline int BSGS(int a, int mod, int b) {
    int sq = ceil(sqrt(mod));
    mp.clear();
    int powa = 1;
    for (int B = 0, x = 1; B <= sq; B++)
        x = 1ll * b * powa % mod, mp[x] = B,
        powa = 1ll * powa * ((B == sq) ? 1 : a) % mod;
    for (int A = 0, B, x = 1; A <= sq; A++) {
        if (mp.find(x) != mp.end()) {
            B = mp[x];
            if (A * sq - B >= 0)
                return A * sq - B;
        }
        x = 1ll * x * powa % mod;
    }
    return -1;  // 无解
}
```

7.4 扩展 BSGS

不要求 a, p 互质。

```
inline int inv(int a, int mod) {
    int x, y;
    exgcd(a, mod, x, y);
    return (x % mod + mod) % mod;
}
inline int exBSGS(int a, int mod, int b) {
    b %= mod;
    if (b == 1 || mod == 1)
        return 0;
    int k = 0, D = 1, U = 1, tmod = mod, tb = b;
    for (int x; (x = gcd(tmod, a)) > 1;) {
        if (tb % x)
            return -1;
        tmod /= x, tb /= x, D *= x, k++;
        U = 1ll * U * a / x % tmod;
        if (U == tb)
            return k;
    }
    tb = 1ll * tb * inv(U, tmod) % tmod;
    int ret = BSGS(a, tmod, tb);
    if (ret == -1)
        return -1;
    return ret + k;
}
```

7.5 Lucas 定理

```
ll C(ll x, ll y) {
    if (x < y)
        return 0;
    if (x < mod && y < mod)
        return mi[x] * invmi[y] % mod * invmi[x - y] % mod;
    return C(x / mod, y / mod) * C(x % mod, y % mod) % mod;
}  // mod 为一素数。
```

7.6 扩展 Lucas 定理

```
// 代码是题解里拉的
#include <bits/stdc++.h>
#define ll long long
using namespace std;
#ifndef Fading
inline char gc() {
    static char now[1 << 16], *S, *T;
    if (T == S) {
        T = (S = now) + fread(now, 1, 1 << 16, stdin);
        if (T == S)
            return EOF;
    }
    return *S++;
}
#endif
#ifdef Fading
#define gc getchar
#endif
void exgcd(ll a, ll b, ll& x, ll& y) {
    if (!b)
        return (void)(x = 1, y = 0);
    exgcd(b, a % b, x, y);
    ll tmp = x;
    x = y;
    y = tmp - a / b * y;
}
ll gcd(ll a, ll b) {
    if (b == 0)
        return a;
    return gcd(b, a % b);
}
inline ll INV(ll a, ll p) {
    ll x, y;
    exgcd(a, p, x, y);
    return (x + p) % p;
}
inline ll lcm(ll a, ll b) {
    return a / gcd(a, b) * b;
}
inline ll mabs(ll x) {
    return (x > 0 ? x : -x);
}
inline ll fast_mul(ll a, ll b, ll p) {
    ll t = 0;
    a %= p;
    b %= p;
    while (b) {
        if (b & 1LL)
            t = (t + a) % p;
        b >>= 1LL;
        a = (a + a) % p;
    }
    return t;
}
inline ll fast_pow(ll a, ll b, ll p) {
    ll t = 1;
    a %= p;
    while (b) {
        if (b & 1LL)
            t = (t * a) % p;
        b >>= 1LL;
```

```
        a = (a * a) % p;
    }
    return t;
}
inline ll read() {
    ll x = 0, f = 1;
    char ch = gc();
    while (!isdigit(ch)) {
        if (ch == '-')
            f = -1;
        ch = gc();
    }
    while (isdigit(ch))
        x = x * 10 + ch - '0', ch = gc();
    return x * f;
}
inline ll F(ll n, ll P, ll PK) {
    if (n == 0)
        return 1;
    ll rou = 1;  // 循环节
    ll rem = 1;  // 余项
    for (ll i = 1; i <= PK; i++) {
        if (i % P)
            rou = rou * i % PK;
    }
    rou = fast_pow(rou, n / PK, PK);
    for (ll i = PK * (n / PK); i <= n; i++) {
        if (i % P)
            rem = rem * (i % PK) % PK;
    }
    return F(n / P, P, PK) * rou % PK * rem % PK;
}
inline ll G(ll n, ll P) {
    if (n < P)
        return 0;
    return G(n / P, P) + (n / P);
}
inline ll C_PK(ll n, ll m, ll P, ll PK) {
    ll fz = F(n, P, PK), fm1 = INV(F(m, P, PK), PK),
       fm2 = INV(F(n - m, P, PK), PK);
    ll mi = fast_pow(P, G(n, P) - G(m, P) - G(n - m, P), PK);
    return fz * fm1 % PK * fm2 % PK * mi % PK;
}
ll A[1001], B[1001];
// x=B(mod A)
inline ll exLucas(ll n, ll m, ll P) {
    ll ljc = P, tot = 0;
    for (ll tmp = 2; tmp * tmp <= P; tmp++) {
        if (!(ljc % tmp)) {
            ll PK = 1;
            while (!(ljc % tmp)) {
                PK *= tmp;
                ljc /= tmp;
            }
            A[++tot] = PK;
            B[tot] = C_PK(n, m, tmp, PK);
        }
    }
    if (ljc != 1) {
        A[++tot] = ljc;
        B[tot] = C_PK(n, m, ljc, ljc);
    }
    ll ans = 0;
    for (ll i = 1; i <= tot; i++) {
        ll M = P / A[i], T = INV(M, A[i]);
        ans = (ans + B[i] * M % P * T % P) % P;
    }
    return ans;
}
signed main() {
    ll n = read(), m = read(), P = read();
    printf("%lld\n", exLucas(n, m, P));
    return 0;
}
```

7.7 杜教筛

实际上是利用迪利克雷卷积来构造递推式，从而对一些积性函数快速求和的方法。

我们现在考虑求取积性函数 f 的前缀和 F 。设存在函数 g ，使得 $f*g$ 的前缀和可以被快速计算，那么：

$$\begin{aligned}\sum_{k=1}^{n}(f*g)(k)&=\sum_{k=1}^{n}\sum_{d|k}f\left(\frac{k}{d}\right)\times g(d)\\&=\sum_{d=1}^{n}\sum_{k=1}^{\lfloor n/d\rfloor}f(k)\times g(d)\\&=\sum_{d=1}^{n}g(d)\times F\left(\left\lfloor\frac{n}{d}\right\rfloor\right)\\&=\sum_{d=2}^{n}g(d)\times F\left(\left\lfloor\frac{n}{d}\right\rfloor\right)+g(1)\times F(n)\end{aligned}$$

则：

$$F(n)=\left(\sum_{k=1}^{n}(f*g)(k)-\sum_{d=2}^{n}g(d)\times F\left(\left\lfloor\frac{n}{d}\right\rfloor\right)\right)\Big/g(1)$$

若 $f*g$ 的前缀和可以被快速计算，我们就可以使用整除分块，从而把 $F(n)$ 划分为若干个子问题。使用时使用线性筛来预处理 F 的前 $n^{\frac{2}{3}}$ 项，这样杜教筛的时间复杂度为 $O(n^{\frac{2}{3}})$ 。

7.8 Min-25 筛

Min-25 筛本质上是对埃氏筛进行了扩展，用于求解积性函数的前缀和，要求其在质数与质数的次幂处的取值可以被快速计算。

下面对 Min-25 筛的运行过程做一个简要的推导：

记 $\mathbb{P}$ 中的数为 p ，p_k 为 $\mathbb{P}$ 中第 k 小的数，$\mathrm{lpf}(n)$ (lowest prime factor) 为 n 的最小素因子，$F(n)=\sum_{p\le n}f(p)$ ，$F_k(n)=\sum_{i=2}^{n}[p_k\le\mathrm{lpf}(i)]f(i)$ ，不难发现答案即为 $F_1(n)+1$ 。

考虑在 F_k 与 F 之间建立递推关系，从素因子的角度出发，并应用积性函数的性质，我们有：

$$\begin{aligned}F_k(n)&=\sum_{i=1}^{n}[\mathrm{lpf}(i)\ge k]f(i)\\&=\sum_{i\ge k,p_i^2\le n}\ \sum_{c\ge1,p_i^c\le n}f(p_i^c)\times([c>1]+F_{k+1}(n/p_i^c))+F(n)-F(p_{k-1})\end{aligned}$$

现在的问题在于如何快速求取 F 。首先可以注意到只有 $O(\sqrt{n})$ 处的 F_k 和 F 的取值对我们来说是有用的，这一点保障了我们的计算复杂度。现在我们只关注 f 在质数处的取值，在具体的问题中，这一部分往往可以被表示为一个低次多项式，因此我们可以考虑分别计算每一项的贡献，最后再把它们加起来。也就是说现在我们只需要求 $g(n)=n^s$ 在只考虑素数处的取值的情况下的前缀和。注意到 g 具有非常优美的性质，其是一个完全积性函数，因此我们考虑构造 $G_k(n)=\sum_{i=2}^{n}[\mathrm{lpf}(i)>p_k\vee i\in\mathbb{P}]g(i)$ ，即埃氏筛第 k 轮以后剩下的数处的 g 的取值之和。从埃氏筛的过程入手，我们考虑如何递推求解 $G_k(n)$ ，即：1. 对于 $p_k^2>n$ ，显然，所有满足的条件的数都会是素数，因此 $G_k(n)=G_{k-1}(n)$ 。2. 否则我们希望去除掉所有 lpf 为 p_k 的合数处的取值，即减去 $g(p_k)G_{k-1}(n/p_k)$ 。3. 在第 2 步中会多减一部分，这部分均仅含一个小于 p_k 的素因子，因此我们加上 $g(p_k)G_{k-1}(p_{k-1})$ 。

概括一下，我们有：

$$G_k(n)=G_{k-1}(n)-[p_k^2\le n]g(p_k)(G_{k-1}(n/p_k)-G_{k-1}(p_{k-1}))$$

以下是洛谷 P5325 求积性函数 $f(p^k)=p^k(p^k-1),p\in\mathbb{P}$ 的前缀和的代码：

```
#include <bits/stdc++.h>
#define rep(i, a, b) for (R int i = a; i <= b; i++)
#define R register
#define endl putchar('\n')
#define puts putchar(' ')
const int maxn = 5000005;
const long long mod = 1e9 + 7, inv3 = 333333336;
#define int long long
using namespace std;

int n, sqr, vis[maxn], pri[maxn], cnt;
int sp1[maxn], sp2[maxn], w[maxn], tot;
int id1[maxn], id2[maxn], g1[maxn], g2[maxn];

void get_prime(int n) {
    vis[1] = 1;
    rep(i, 2, n) {
```

```
        if (!vis[i]) {
            pri[++cnt] = i;
            sp1[cnt] = (sp1[cnt - 1] + i) % mod;
            sp2[cnt] = (sp2[cnt - 1] + 1ll * i * i) % mod;
        }
        for (R int j = 1; j <= cnt && i * pri[j] <= n; j++) {
            vis[i * pri[j]] = 1;
            if (!(i % pri[j])) break;
        }
    }
}

void get_g() {
    for (int i = 1; i <= n;) {
        int j = n / (n / i);
        w[++tot] = n / i;
        g1[tot] = w[tot] % mod;
        g2[tot] = g1[tot] * (g1[tot] + 1) / 2 % mod * (2 *
          ↪ g1[tot] + 1) % mod * inv3 % mod - 1;
        g1[tot] = g1[tot] * (g1[tot] + 1) / 2 % mod - 1;
        if (n / i <= sqr)
            id1[n / i] = tot;
        else
            id2[n / (n / i)] = tot;
        i = j + 1;
    }
    rep(i, 1, cnt) {
        for (int j = 1; j <= tot && pri[i] * pri[i] <= w[j];
          ↪ j++) {
            int k = w[j] / pri[i] <= sqr ? id1[w[j] / pri[i]] :
              ↪ id2[n / (w[j] / pri[i])];
            g1[j] = (g1[j] - pri[i] * (g1[k] - sp1[i - 1] +
              ↪ mod)) % mod;
            g2[j] = (g2[j] - pri[i] * pri[i] * (g2[k] - sp2[i -
              ↪ 1] + mod)) % mod;
        }
    }
}

int S(int x, int y) {
    if (pri[y] >= x) return 0;
    int k = x <= sqr ? id1[x] : id2[n / x];
    int res = (g2[k] - g1[k] - (sp2[y] - sp1[y]) + mod) % mod;
    for (R int i = y + 1; i <= cnt && pri[i] * pri[i] <= x; i++)
      ↪ {
        int now = pri[i];
        for (R int e = 1; now <= x; e++, now = now * pri[i]) {
            res = (res + now % mod * ((now - 1) % mod) % mod *
              ↪ (S(x / now, i) + (e > 1))) % mod;
        }
    }
    return res;
}

signed main() {
    scanf("%lld", &n);
    sqr = sqrt(n);
    get_prime(sqr);
    get_g();
    printf("%lld", (S(n, 0) + 1) % mod);
    return 0;
}
```

8 计算几何

8.1 声明与宏

```cpp
#define cp const P&
#define cl const L&
#define cc const C&
#define vp vector<P>
#define cvp const vector<P>&
#define D double
#define LD long double
std::mt19937 rnd(time(0));
const LD eps = 1e-12;
const LD pi = std::numbers::pi;
const LD INF = 1e9;
int sgn(LD x) {
    return x > eps ? 1 : (x < -eps ? -1 : 0);
}
```

8.2 点与向量

```cpp
struct P {
    LD x, y;

    P(const LD &x = 0, const LD &y = 0) : x(x), y(y) {}
    P(cp a) : x(a.x), y(a.y) {}

    P operator=(cp a) {
        x = a.x, y = a.y;
        return *this;
    }
    P rot(LD t) const {
        LD c = cos(t), s = sin(t);
        return P(x * c - y * s, x * s + y * c);
    } // counterclockwise
    P rot90() const { return P(-y, x); } // counterclockwise
    P _rot90() const { return P(y, -x); } // clockwise
    LD len() const { return sqrtl(x * x + y * y); }
    LD len2() const { return x * x + y * y; }
    P unit() const {
        LD d = len();
        return P(x / d, y / d);
    }
    void read() { scanf("%Lf%Lf", &x, &y); }
    void print() const { printf("(%.9Lf,%.9Lf)", x, y); }
};

bool operator<(cp a, cp b) { return a.x == b.x ? a.y < b.y : a.x < b.x; }
bool operator>(cp a, cp b) { return a.x == b.x ? a.y > b.y : a.x > b.x; }
bool operator==(cp a, cp b) { return !sgn(a.x - b.x) && !sgn(a.y - b.y); }
P operator+(cp a, cp b) { return P(a.x + b.x, a.y + b.y); }
P operator-(cp a, cp b) { return P(a.x - b.x, a.y - b.y); }
P operator*(const LD &a, cp b) { return P(a * b.x, a * b.y); }
P operator*(cp b, const LD &a) { return P(a * b.x, a * b.y); }
P operator/(cp b, const LD &a) { return P(b.x / a, b.y / a); }
LD operator*(cp a, cp b) { return a.x * b.x + a.y * b.y; } // 点积
LD operator^(cp a, cp b) { return a.x * b.y - a.y * b.x; } // 叉积
LD rad(cp a, cp b) { return acosl((a.x == 0 && a.y == 0 || b.x == 0 && b.y == 0) ? 0 : (a * b / a.len() / b.len())); } // 夹角
bool left(cp a, cp b) { return sgn(a ^ b) > 0; } // 没什么用
```

8.3 线

```cpp
struct L {
    P s, t;

    L(cp a, cp b) : s(a), t(b) {}
    L(cl l) : s(l.s), t(l.t) {}
    void read() { s.read(), t.read(); }
};

bool point_on_line(cp a, cl l) { return sgn((a - l.s) ^ (l.t - l.s)) == 0; }
bool point_on_segment(cp a, cl l) { return sgn((a - l.s) ^ (l.t - l.s)) == 0 && sgn((a - l.s) * (a - l.t)) <= 0; }
bool two_side(cp a, cp b, cl l) { return sgn((a - l.s) ^ (a - l.t)) * sgn((b - l.s) ^ (b - l.t)) < 0; }
bool intersection_judge_strict(cl a, cl b) { return two_side(a.s, a.t, b) && two_side(b.s, b.t, a); }
bool intersection_judge(cl a, cl b) { return point_on_segment(a.s, b) || point_on_segment(a.t, b) || point_on_segment(b.s, a) || point_on_segment(b.t, a) || intersection_judge_strict(a, b); }
P ll_intersection(cl a, cl b) {
    LD s1 = (a.t - a.s) ^ (b.s - a.s), s2 = (a.t - a.s) ^ (b.t - a.s);
    return (s2 * b.s - s1 * b.t) / (s2 - s1);
}
bool point_on_ray(cp a, cl b) { return sgn((a - b.s) ^ (b.t - b.s)) == 0 && sgn((a - b.s) * (b.t - b.s)) >= 0; }
bool ray_intersection_judge(cl a, cl b) {
    P p(ll_intersection(a, b));
    return sgn((p - a.s) * (a.t - a.s)) >= 0 && sgn((p - b.s) * (b.t - b.s)) >= 0;
} // 似乎有更快的做法，但是看不懂
LD point_to_line(cp a, cl b) { return fabs((b.t - b.s) ^ (a - b.s)) / (b.t - b.s).len(); } // 距离
P project_to_line(cp a, cl b) { return b.s + (a - b.s) * (b.t - b.s) / (b.t - b.s).len2() * (b.t - b.s); } // 垂足
LD point_to_segment(cp a, cl b) {
    if (b.s == b.t) return (a - b.s).len();
    if (sgn((a - b.s) * (b.t - b.s)) * sgn((a - b.t) * (b.t - b.s)) <= 0)
        return fabs((b.t - b.s) ^ (a - b.s)) / (b.t - b.s).len();
    return min((a - b.s).len(), (a - b.t).len());
}
```

8.4 圆

```cpp
struct C {
    P c;
    LD r;

    C(cp c, const LD &r = 0) : c(c), r(r) {}
    C(cc a) : c(a.c), r(a.r) {};
    C(cp a, cp b) {
        c = (a + b) / 2;
        r = (a - c).len();
    }
    C(cp x, cp y, cp z) {
        P p(y - x), q(z - x);
        P s(p * p / 2, q * q / 2);
        LD d = p ^ q;
        p = P(s ^ P(p.y, q.y), P(p.x, q.x) ^ s) / d;
        c = x + p, r = p.len();
    }
};

bool in_circle(cp a, cc b) { return sgn((b.c - a).len() - b.r) <= 0; }
C min_circle(vp p) // O(n)
{
    shuffle(p.begin(), p.end(), rnd);
    int n = p.size(), i, j, k;
    C ret(p[0], 0);
    for (i = 1; i < n; ++i)
        if (!in_circle(p[i], ret))
            for (ret = C(p[i], p[0]), j = 1; j < i; ++j)
                if (!in_circle(p[j], ret))
                    for (ret = C(p[i], p[j]), k = 0; k < j; ++k)
                        if (!in_circle(p[k], ret))
                            ret = C(p[i], p[j], p[k]);
    return ret;
}

vp lc_intersection(cl l, cc c) {
    LD x = point_to_line(c.c, l);
    if (sgn(x - c.r) > 0) return vp();
    x = sqrtl(c.r * c.r - x * x);
    P p(project_to_line(c.c, l));
    if (sgn(x) == 0) return vp({p});
    return vp({p + x * (l.t - l.s).unit(), p - x * (l.t - l.s).unit()});
}

LD cc_intersection_area(cc a, cc b) {
    LD d = (a.c - b.c).len();
    if (sgn(d - (a.r + b.r)) >= 0) return 0;
    if (sgn(d - fabs(a.r - b.r)) <= 0) {
        LD r = min(a.r, b.r);
        return pi * r * r;
    }
    LD x = (d * d + a.r * a.r - b.r * b.r) / (2 * d), t1 = acosl(min((LD)1, max((LD)-1, x / a.r))), t2 = acosl(min((LD)1, max((LD)-1, (d - x) / b.r)));
    return a.r * a.r * t1 + b.r * b.r * t2 - d * a.r * sinl(t1);
}
```

```
vp cc_intersection(cc a, cc b) {
    LD d = (a.c - b.c).len();
    if (a.c == b.c || sgn(d - a.r - b.r) > 0 || sgn(d - fabs(a.r - b.r)) < 0) return vp();
    P r = (b.c - a.c).unit();
    LD x = (d * d + a.r * a.r - b.r * b.r) / (d * 2);
    LD h = sqrtl(a.r * a.r - x * x);
    if (sgn(h) == 0) return vp({a.c + r * x});
    return vp({a.c + r * x + r.rot90() * h, a.c + r * x - r.rot90() * h});
}

vp tangent(cp a, cc b) { return cc_intersection(b, C(a, b.c)); } // 点到圆的切点

vector<L> tangent(cc a, cc b, int f) // f = 1 extangent;f = -1 intangent
{
    D d = (b.c - a.c).len();
    int sg = sgn(fabs((b.r - f * a.r) / d) - 1);
    if (sg == 1)
        return {};
    else if (sg == 0) {
        P p = a.c - sgn(b.r - f * a.r) * f * a.r * (b.c - a.c).unit();
        return {{p, p + (b.c - a.c).rot90()}};
    } // 内切/外切
    D theta = asin(min((D)1, max((D)-1, f * (b.r - f * a.r) / d)));
    P du = (b.c - a.c).unit(), du1 = du.rot(theta + pi / 2), du2 = du.rot(-theta - pi / 2);
    return {{a.c + a.r * du1, b.c + f * b.r * du1}, {a.c + a.r * du2, b.c + f * b.r * du2}};
}
```

8.5 凸包

```
vp convex(vp a) {
    if (a.size() < 2) return a;
    sort(a.begin(), a.end());
    int n = a.size(), cnt = 0;
    vp con({p[0]});
    for (int i = 0; i < n; ++i) {
        while (cnt > 0 && sgn(((p[i] - con[cnt - 1]) ^ (con[cnt] - con[cnt - 1]))) > 0) //>=
            --cnt, con.pop_back();
        ++cnt, con.push_back(p[i]);
    }
    int fixed = cnt;
    for (int i = n - 2; ~i; --i) {
        while (cnt > fixed && sgn(((p[i] - con[cnt - 1]) ^ (con[cnt] - con[cnt - 1]))) > 0) //>=
            --cnt, con.pop_back();
        ++cnt, con.push_back(p[i]);
    }
    con.pop_back();
    return con;
}

vp minkowski(vp p1, vp p2) {
    if (p1.empty() || p2.empty()) return vp();
    int n1 = p1.size(), n2 = p2.size();
    vp ret;
    if (n1 == 1) {
        for (int i = 0; i < n2; ++i)
            ret.push_back(p1[0] + p2[i]);
        return ret;
    }
    if (n2 == 1) {
        for (int i = 0; i < n1; ++i)
            ret.push_back(p1[i] + p2[0]);
        return ret;
    }
    p1.push_back(p1[0]), p1.push_back(p1[1]), p2.push_back(p2[0]), p2.push_back(p2[1]);
    ret.push_back(p1[0] + p2[0]);
    int i1 = 0, i2 = 0;
    while (i1 < n1 || i2 < n2) {
        if (((p1[i1 + 1] - p1[i1]) ^ (p2[i2 + 1] - p2[i2])) > 0)
            ret.push_back(p1[++i1] + p2[i2]);
        else
            ret.push_back(p2[++i2] + p1[i1]);
    } // p1.pop_back(),p1.pop_back(),p2.pop_back(),p2.pop_back();
    ret.pop_back();
    return ret;
}

struct Con {
    int n;
    vp a, upper, lower;

    Con(cvp _a) : a(_a) {
        n = a.size();
        int k = 0;
        for (int i = 1; i < n; ++i)
            if (a[i] > a[k]) k = i;
        for (int i = 0; i <= k; ++i)
            lower.push_back(a[i]);
        for (int i = k; i < n; ++i)
            upper.push_back(a[i]);
        upper.push_back(a[0]);
    }

}

// Below is from Nemesis

// Convex
int n;
vector<point> a; // 可以封装成一个 struct
bool inside(cp u) { // 点在凸包内
    int l = 1, r = n - 2;
    while (l < r) {
        int mid = (l + r + 1) / 2;
        if (turn(a[0], a[mid], u) >= 0)
            l = mid;
        else
            r = mid - 1;
    }
    return turn(a[0], a[l], u) >= 0 && turn(a[l], a[l + 1], u) >= 0 && turn(a[l + 1], a[0], u) >= 0;
}
int search(auto f) { // 凸包求极值，需要 C++17
    int l = 0, r = n - 1;
    int d = f(a[r], a[l]) ? (swap(l, r), -1) : 1;
    while (d * (r - l) > 1) {
        int mid = (l + r) / 2;
        if (f(a[mid], a[l]) && f(a[mid], a[mid - d]))
            l = mid;
        else
            r = mid;
    }
    return l;
}
pair<int, int> get_tan(cp u) { // 求切线
    return // 严格在凸包外；需要边界上时，特判 a[n-1] -> a[0]
        {search([&](cp x, cp y) { return turn(u, y, x) > 0; }),
         search([&](cp x, cp y) { return turn(u, x, y) > 0; })};
}
point at(int i) { return a[i % n]; }
int inter(cp u, cp v, int l, int r) {
    int sl = turn(u, v, at(l));
    while (l + 1 < r) {
        int m = (l + r) / 2;
        if (sl == turn(u, v, at(m)))
            l = m;
        else
            r = m;
    }
    return l % n;
}
bool get_inter(cp u, cp v, int &i, int &j) { // 求直线交点
    int p0 = search([&](cp x, cp y) { return det(v - u, x - u) < det(v - u, y - u); }),
        p1 = search([&](cp x, cp y) { return det(v - u, x - u) > det(v - u, y - u); });
    if (turn(u, v, a[p0]) * turn(u, v, a[p1]) < 0) {
        if (p0 > p1) swap(p0, p1);
        i = inter(u, v, p0, p1);
        j = inter(u, v, p1, p0 + n);
        return true;
    } else
        return false;
}
LD near(cp u, int l, int r) {
    if (l > r) r += n;
    int sl = sgn(dot(u - at(l), at(l + 1) - at(l)));
    LD ret = p2s(u, {at(l), at(l + 1)});
    while (l + 1 < r) {
        int m = (l + r) / 2;
        if (sl == sgn(dot(u - at(m), at(m + 1) - at(m))))
            l = m;
        else
            r = m;
    }
    return min(ret, p2s(u, {at(l), at(l + 1)}));
}
LD get_near(cp u) { // 求凸包外点到凸包最近点
    if (inside(u)) return 0;
    auto [x, y] = get_tan(u);
```

```
    return min(near(u, x, y), near(u, y, x));
}

// Dynamic convex hull
struct hull { // upper hull, left to right
    set<point> a;
    LL tot;
    hull() { tot = 0; }
    LL calc(auto it) {
        auto u = it == a.begin() ? a.end() : prev(it);
        auto v = next(it);
        LL ret = 0;
        if (u != a.end()) ret += det(*u, *it);
        if (v != a.end()) ret += det(*it, *v);
        if (u != a.end() && v != a.end()) ret -= det(*u, *v);
        return ret;
    }
    void insert(point p) {
        if (!a.size()) {
            a.insert(p);
            return;
        }
        auto it = a.lower_bound(p);
        bool out;
        if (it == a.begin())
            out = (p < *it); // special case
        else if (it == a.end())
            out = true;
        else
            out = turn(*prev(it), *it, p) > 0;
        if (!out) return;
        while (it != a.begin()) {
            auto o = prev(it);
            if (o == a.begin() || turn(*prev(o), *o, p) < 0)
                break;
            else
                erase(o);
        }
        while (it != a.end()) {
            auto o = next(it);
            if (o == a.end() || turn(p, *it, *o) < 0)
                break;
            else
                erase(it), it = o;
        }
        tot += calc(a.insert(p).first);
    }
    void erase(auto it) {
        tot -= calc(it);
        a.erase(it);
    }
};
```

8.6 三角形

```
// From Nemesis
point incenter(cp a, cp b, cp c) { // 内心
    double p = dis(a, b) + dis(b, c) + dis(c, a);
    return (a * dis(b, c) + b * dis(c, a) + c * dis(a, b)) / p;
}
point circumcenter(cp a, cp b, cp c) { // 外心
    point p = b - a, q = c - a, s(dot(p, p) / 2, dot(q, q) / 2);
    double d = det(p, q);
    return a + point(det(s, {p.y, q.y}), det({p.x, q.x}, s)) /
      ↪ d;
}
point orthocenter(cp a, cp b, cp c) { // 重心
    return a + b + c - circumcenter(a, b, c) * 2.0;
}
point fermat_point(cp a, cp b, cp c) { // 费马点
    if (a == b) return a;
    if (b == c) return b;
    if (c == a) return c;
    double ab = dis(a, b), bc = dis(b, c), ca = dis(c, a);
    double cosa = dot(b - a, c - a) / ab / ca;
    double cosb = dot(a - b, c - b) / ab / bc;
    double cosc = dot(b - c, a - c) / ca / bc;
    double sq3 = PI / 3.0;
    point mid;
    if (sgn(cosa + 0.5) < 0)
        mid = a;
    else if (sgn(cosb + 0.5) < 0)
        mid = b;
    else if (sgn(cosc + 0.5) < 0)
        mid = c;
    else if (sgn(det(b - a, c - a)) < 0)
        mid = line_inter({a, b + (c - b).rot(sq3)}, {b, c + (a -
          ↪ c).rot(sq3)});
    else
        mid = line_inter({a, c + (b - c).rot(sq3)}, {c, b + (a -
          ↪ b).rot(sq3)});
    return mid;
} // minimize(|A-x|+|B-x|+|C-x|)
```

8.7 多边形

```
vp Poly_cut(vp p, cl l) // 直线切多边形，返回一侧的图形端点，O(n)
{
    if (p.empty()) return vp();
    vp ret;
    int n = p.size();
    p = push_back(p[0]);
    for (int i = 0; i < n; ++i) {
        if (((p[i] - l.s) ^ (l.t - l.s)) <= 0)
            ret.push_back(p[i]);
        if (two_size(p[i], p[i + 1], l))
            ret.push_back(ll_intersection(l, L(p[i], p[i +
              ↪ 1])));
    } // p.pop_back();
    return ret;
}

LD Poly_area(vp p) {
    LD ar = 0;
    int n = p.size();
    for (int i = 0, j = n - 1; i < n; j = i++)
        ar += p[i] ^ p[j];
    return fabs(ar) / 2;
}

// Below is from Nemesis

// 多边形与圆交
LD angle(cp u, cp v) {
    return 2 * asin(dis(u.unit(), v.unit()) / 2);
}
LD area(cp s, cp t, LD r) { // 2 * area
    LD theta = angle(s, t);
    LD dis = p2s({0, 0}, {s, t});
    if (sgn(dis - r) >= 0) return theta * r * r;
    auto [u, v] = line_circle_inter({s, t}, {{0, 0}, r});
    point lo = sgn(det(s, u)) >= 0 ? u : s;
    point hi = sgn(det(v, t)) >= 0 ? v : t;
    return det(lo, hi) + (theta - angle(lo, hi)) * r * r;
}
LD solve(vector<point> &p, cc c) {
    LD ret = 0;
    for (int i = 0; i < (int)p.size(); ++i) {
        auto u = p[i] - c.c;
        auto v = p[(i + 1) % p.size()] - c.c;
        int s = sgn(det(u, v));
        if (s > 0)
            ret += area(u, v, c.r);
        else if (s < 0)
            ret -= area(v, u, c.r);
    }
    return abs(ret) / 2;
} // ret 在 p 逆时针时为正
```

8.8 半平面交

```
// From Nemesis
int half(cp a) { return a.y > 0 || (a.y == 0 && a.x > 0) ? 1 :
  ↪ 0; }
bool turn_left(cl a, cl b, cl c) {
    return turn(a.s, a.t, line_inter(b, c)) > 0;
}
bool is_para(cl a, cl b) { return !sgn(det(a.t - a.s, b.t -
  ↪ b.s)); }
bool cmp(cl a, cl b) {
    int sign = half(a.t - a.s) - half(b.t - b.s);
    int dir = sgn(det(a.t - a.s, b.t - b.s));
    if (!dir && !sign)
        return turn(a.s, a.t, b.t) < 0;
    else
        return sign ? sign > 0 : dir > 0;
}
vector<point> hpi(vector<line> h) { // 半平面交
    sort(h.begin(), h.end(), cmp);
    vector<line> q(h.size());
    int l = 0, r = -1;
    for (auto &i : h) {
        while (l < r && !turn_left(i, q[r - 1], q[r]))
            --r;
        while (l < r && !turn_left(i, q[l], q[l + 1]))
            ++l;
        if (l <= r && is_para(i, q[r])) continue;
```

```
25          q[++r] = i;
26      }
27      while (r - l > 1 && !turn_left(q[l], q[r - 1], q[r]))
28          --r;
29      while (r - l > 1 && !turn_left(q[r], q[l], q[l + 1]))
30          ++l;
31      if (r - l < 2) return {};
32      vector<point> ret(r - l + 1);
33      for (int i = l; i <= r; i++)
34          ret[i - l] = line_inter(q[i], q[i == r ? l : i + 1]);
35      return ret;
36  }
37  // 空集会在队列里留下一个开区域；开区域会被判定为空集。
38  // 为了保证正确性，一定要加足够大的框，尽可能避免零面积区域。
39  // 实在需要零面积区域边缘，需要仔细考虑 turn_left 的实现。
```

9 杂项

9.1 生成树计数

【根据度数求方案】对于给定每个点度数为 d_i 的无根树，方案数为：

$$\frac{(n-2)!}{\prod_{i=1}^{n}(d_i-1)!}$$

【根据连通块数量与大小求方案】一个 n 个点 m 条边的带标号无向图有 k 个连通块，每个连通块大小为 s_i，需要增加 $k-1$ 条边使得整个图联通，方案数为：（但是当 $k=1$ 时需要特判）

$$n^{k-2}\cdot\prod_{i=1}^{k}s_i$$

证明只需考虑 prufer 序列即可。

9.2 类欧几里得

$ax+by=n$ 的几何意义可以想象为一条直线，那么 $[0,n]$ 中可以被表示出来的整数就是 $(0,0),\left(\frac{n}{a},0\right),\left(0,\frac{n}{b}\right)$ 为顶点的三角形在第一象限内含有的整点个数。

显然的结论就是，在 $[0,n]$ 可以表示出的整数数量为：

$$\sum_{x=0}^{\lfloor\frac{n}{a}\rfloor}\left\lfloor\frac{n-ax}{b}\right\rfloor$$

类欧几里得可以在 $\mathcal{O}(\log\max(\dashv,\lfloor\rfloor))$ 的时间内解决此类问题。

求 $\sum_{i=0}^{n}\lfloor\frac{ai+b}{c}\rfloor$：

```
ll solve(ll a, ll b, ll c, ll n) {
    if (a == 0) return (n + 1) * (b / c) % mod;
    if (a >= c || b >= c)
        return (n * (n + 1) % mod * inv2 % mod * (a / c) % mod +
          ↪ (n + 1) * (b / c) % mod + solve(a % c, b % c, c, n))
          ↪ % mod;
    ll m = (a * n + b) / c;
    return (n * (m % mod) % mod - solve(c, c - b - 1, a, m - 1)
      ↪ + mod) % mod;
}
```

求 $\sum_{i=0}^{n}\lfloor\frac{ai+b}{c}\rfloor^2$ 和 $\sum_{i=0}^{n}i\lfloor\frac{ai+b}{c}\rfloor$，分别对应以下的 g 和 h：

```
struct Euclid {
    ll f, g, h;
};
Euclid solve(ll a, ll b, ll c, ll n) {
    Euclid ans, tmp;
    if (a == 0) {
        ans.f = (n + 1) * (b / c) % mod;
        ans.g = n * (n + 1) % mod * (b / c) % mod * inv2 % mod;
        ans.h = (n + 1) * (b / c) % mod * (b / c) % mod;
        return ans;
    }
    if (a >= c || b >= c) {
        tmp = solve(a % c, b % c, c, n);
        ans.f = (n * (n + 1) % mod * inv2 % mod * (a / c) % mod
          ↪ + (n + 1) * (b / c) % mod + tmp.f) % mod;
        ans.g = (n * (n + 1) % mod * (2 * n + 1) % mod * (a / c)
          ↪ % mod * inv6 % mod + n * (n + 1) % mod * (b / c) %
          ↪ mod * inv2 % mod + tmp.g) % mod;
        ans.h = ((a / c) * (a / c) % mod * n % mod * (n + 1) %
          ↪ mod * (n * 2 + 1) % mod * inv6 % mod +
                 (b / c) * (b / c) % mod * (n + 1) % mod + (a /
                   ↪ c) * (b / c) % mod * n % mod * (n + 1) %
                   ↪ mod +
                 2 * (b / c) % mod * tmp.f % mod + 2 * (a / c) %
                   ↪ mod * tmp.g % mod + tmp.h) %
                mod;
        return ans;
    }
    ll m = (a * n + b) / c;
    tmp = solve(c, c - b - 1, a, m - 1);
    ans.f = (n * (m % mod) % mod + mod - tmp.f) % mod;
    ans.g = (n * (m % mod) % mod * (n + 1) % mod - tmp.h -
      ↪ tmp.f) % mod * inv2 % mod;
    ans.h = (n * (m % mod) % mod * (m + 1) % mod - 2 * tmp.g - 2
      ↪ * tmp.f - ans.f) % mod;
    return ans;
}

ans.f = (ans.f % mod + mod) % mod;
ans.g = (ans.g % mod + mod) % mod;
ans.h = (ans.h % mod + mod) % mod;
```

9.3 没有精度问题的整除

```
int ceil(int x, int y) {
  return (x>0 ? (x+y-1)/y : x/y);
}
int floor(int x, int y) {
  return (x>0 ? x/y : (x-y+1)/y);
}
```

10 其它工具

10.1 编译命令

```
g++ X.cpp -Wall -O2 -fsanitize=undefined -fsanitize=address X
# -fsanitize=undefined: 检测未定义行为
# -fsanitize=address: 检测内存溢出
```

10.2 快读

```
char *p1, *p2, buf[100000];
#define gc() (p1 == p2 && (p2 = (p1 = buf) + fread(buf, 1, 100000, stdin), p1 == p2) ? EOF : *p1++)
int read() {
    int x = 0, f = 1;
    char ch = gc();
    while (!isdigit(ch)) { if (ch == '-') f = -1; ch = gc(); }
    while (isdigit(ch)) x = x * 10 + ch - '0', ch = gc();
    return x * f;
}
```

10.3 Python Hints

itertools 库

```
from itertools import *
# 笛卡尔积
product('ABCD', 'xy') # Ax Ay Bx By Cx Cy Dx Dy
product(range(2), repeat=3) # 000 001 010 011 100 101 110 111
# 排列
permutations('ABCD', 2) # AB AC AD BA BC BD CA CB CD DA DB DC
# 组合
combinations('ABCD', 2) # AB AC AD BC BD CD
# 有重复的组合
combinations_with_replacement('ABC', 2) # AA AB AC BB BC CC
```

random 库

```
from random import *
randint(1, r) # 在 [1, r] 内的随机整数
choice([1, 2, 3, 5, 8]) # 随机选择序列中一个元素
sample([1, 2, 3, 4, 5], k=2) # 随机抽样两个元素
shuffle(x) # 原地打乱序列 x
l,r = sorted(choices(range(1, N+1), k=2)) # 生成随机区间 [l,r]
binomialvariate(n, p) # 返回服从 B(n,p) 的一个变量
normalvariate(mu, sigma) # 返回服从 N(mu,sigma) 的一个变量
```

列表操作

```
# 列表操作
l = sample(range(100000), 10)
l.sort() # 原地排序
l.sort(key=lambda x:x%10) # 按末尾排序
from functools import cmp_to_key
l.sort(key=cmp_to_key(lambda x,y:y-x)) # 比较函数，小于返回负数
sorted(l) # 非原地排序
l.reverse() reversed(l)
```

字典操作

```
from collections import defaultdict
# 提供一个函数返回缺省值
d = defaultdict(list)
d["a"].append(2)
d["a"].append(3)
d["b"].append(4)
print(d) # {'a': [2, 3], 'b': [4]}
# 用 lambda 可以快速构造出需要的默认值
d = defaultdict(lambda: 2)
# 遍历键值对
for k,v in d.items():
    print(k, v)
```

复数

```
a = 1+2j
print(a.real, a.imag, abs(a), a.conjugate())
```

高精度小数

```
from decimal import Decimal, getcontext, FloatOperation, ROUND_HALF_EVEN
getcontext().prec = 100 # 设置有效位数
getcontext().rounding = getattr(ROUND_HALF_EVEN) # 四舍六入五成双
getcontext().traps[FloatOperation] = True # 禁止 float 混合运算
a = Decimal("114514.1919810")
print(a, f"{a:.2f}")
a.ln() a.log10() a.sqrt() a**2
```

记忆化搜索

```
from functools import cache
# 记忆化搜索，还可以记忆化元组，只要参数满足 Hashable 即可
@cache
def fib(n):
    if n<=2:
        return 1
    return fib(n-1)+fib(n-2)
```

10.4 对拍器

```
import os

while True:
    os.system("python3 data.py > in")
    os.system("/usr/bin/time -f 'test Time=%es' ./test < in > out")
    os.system("/usr/bin/time -f 'std Time=%es' ./std < in > ans")
    if os.system("diff out ans >/dev/null"):
        print("WA")
        exit(1)
    print("AC")
```

10.5 常数表

n	$\log_{10} n$	$n!$	$C(n, n/2)$	$\text{lcm}(1...n)$
2	0.301030	2	2	2
3	0.477121	6	3	6
4	0.602060	24	6	12
5	0.698970	120	10	60
6	0.778151	720	20	60
7	0.845098	5040	35	420
8	0.903090	40320	70	840
9	0.954243	362880	126	2520
10	1.000000	3628800	252	2520
11	1.041393	39916800	462	27720
12	1.079181	479001600	924	27720
15	1.176091	1.31e12	6435	360360
20	1.301030	2.43e18	184756	232792560
25	1.397940	1.55e25	5200300	2.68e10
30	1.477121	2.65e32	155117520	2.33e12

$n \le$	10^1	10^2	10^3	10^4	10^5	10^6
$\max\{\omega(n)\}$	2	3	4	5	26	7
$\max\{d(n)\}$	4	12	32	64	128	240
$\pi(n)$	4	25	168	1229	9592	78498
$n \le$	10^7	10^8	10^9	10^{10}	10^{11}	10^{12}
$\max\{\omega(n)\}$	8	8	9	10	10	11
$\max\{d(n)\}$	448	768	1344	2304	4032	6720
$\pi(n)$	664579	5761455	5.08e7	4.55e8	4.12e9	3.7e10
$n \le$	10^{13}	10^{14}	10^{15}	10^{16}	10^{17}	10^{18}
$\max\{\omega(n)\}$	12	12	13	13	14	15
$\max\{d(n)\}$	10752	17280	26880	41472	64512	103680
$\pi(n)$	$\pi(x) \sim x/\ln(x)$					

10.6 试机赛

- 测试编译器版本。

- `#include<bits/stdc++.h>`
- `pb_ds`
- C++20: `cin>>(s+1);`
- C++17: `auto [x,y]=pair1,"abc";`
- C++11: `auto x=1;`

- 测试 ___int128, ___float128, long double
- 测试 pragma 是否 CE 。
- 测试 -fsanitize=address,undefined
- 测试本地性能