ACM/ICPC 模板大全

Made By GAUSS_CLB

ChenLiangBo

第一章 图论

1.1 搜索

1.1.1 DFS 序

```
struct edge{
    int to,next;
}p[Maxn];
int head[Maxn],st[Maxn],l[Maxn],r[Maxn];
int tot,top,t;
void addedge(int a,int b){
    p[tot].to=b;
    p[tot].next=head[a];
    head[a]=tot++;
}
void dfs(int u){  //dfs 序
    st[++top]=u;
    fa[u]=-1;
    while(top){
        int v=st[top];
        if(!l[v]){
            l[v]=++t;
            for(int i=head[v];i!=-1;i=p[i].next)
                if(p[i].to!=fa[v]){
                    st[++top]=p[i].to;
                    fa[p[i].to]=v;
                }
        }
        else{
            r[v]=++t;
            top--;
        }
    }
}
```

1.1.2 树的重心

```
struct edge{
```

```
    int to,next;
}p[Maxn];
void addedge(int a,int b){
    p[tot].to=b;
    p[tot].next=head[a];
    head[a]=tot++;
    p[tot].to=a;
    p[tot].next=head[b];
    head[b]=tot++;
}

//非递归版 DFS
int ans,id,n;
void dfs(int u){
    st[++top]=u;
    while(top){
        int v=st[top],sum=0,bal=0;
        if(!vis[v]){  //第一次访问
            vis[v]=1;
            for(int i=head[v];i!=-1;i=p[i].next)
                if(!vis[p[i].to]){
                    st[++top]=p[i].to;
                }
        }
        else{  //第二次访问，孩子节点已处理完毕
            for(int i=head[v];i!=-1;i=p[i].next){
                int r=p[i].to;
                if(vis[r]==2){
                    sum+=son[r];
                    bal=max(bal,son[r]);
                }
            }
            son[v]=sum+1;
            bal=max(bal,n-son[v]);
            vis[v]++; //以便统计孩子节点
            top--;
            if(bal<ans||bal==ans&&v<id){
                ans=bal,id=v;
            }
        }
    }
}

//递归版 DFS
```

```
void dfs(int u,int fa){
    int sum=0,bal=0;
    for(int i=head[u];i!=-1;i=p[i].next){
        int v=p[i].to;
        if(v!=fa){
            dfs(v,u);
            sum+=son[v];
            bal=max(bal,son[v]);
        }
    }
    son[u]=sum+1;
    bal=max(bal,n-son[u]);
    if(bal<res){
        res=bal;
        num=0;
        ans[num++]=u;
    }
    else if(bal==res) ans[num++]=u;
}

//BFS 版
st[++top]=1;vis[1]=1;
for(int i=1;i<=n;i++){
    for(int j=head[st[i]];j!=-1;j=p[j].next)
        if(!vis[p[j].to]){
            st[++top]=p[j].to;
            vis[p[j].to]=1;
        }
}
int ans=1<<30,id;
for(int i=top;i>0;i--){
    int sum=0,bal=0;
    for(int j=head[st[i]];j!=-1;j=p[j].next){
        int v=p[j].to;
        if(!vis[v]){
            sum+=son[v];
            bal=max(bal,son[v]);
        }
    }
    vis[st[i]]=0;
    son[st[i]]=sum+1;
    bal=max(bal,n-1-sum);
    if(bal<ans||bal==ans&&st[i]<id){
        ans=bal,id=st[i];
```

```
    }
}
```

1.1.3 树的直径

```
//树形 dp
struct edge{
    int to,v,next;
}p[Maxn];
int max1[Maxn],max2[Maxn],head[Maxn];
int tot,res;
void addedge(int a,int b,int c){
    p[tot].to=b;
    p[tot].v=c;
    p[tot].next=head[a];
    head[a]=tot++;
}
void dfs(int u,int fa){ //过 u 的最长链
    for(int i=head[u];i!=-1;i=p[i].next){
        int v=p[i].to;
        if(v!=fa){
            dfs(v,u);
            if(max1[v]+p[i].v>max1[u]){ //更新链的最长部分
                max2[u]=max1[u];
                max1[u]=max1[v]+p[i].v;
            }
            //更新链的次长部分
            else max2[u]=max(max2[u],max1[v]+p[i].v);
        }
    }
    res=max(res,max1[u]+max2[u]);
}

//两次 BFS
void bfs(int u){
    memset(dist,-1,sizeof dist);
    int s=0,e=-1;
    q[++e]=u,dist[u]=0;
    while(s<=e){
        int v=q[s++];
        for(int i=head[v];i!=-1;i=p[i].next){
            int r=p[i].to;
            if(dist[r]==-1){
```

```
                q[++e]=r;
                //直径边权为 0,为了计算侧链的最大长度
                if(mark[r]) dist[r]=dist[v];
                else dist[r]=dist[v]+p[i].v;
                from[r]=v;
            }
        }
    }
}

//da,db 表示直径两端,st[]存放直径序列
//dist[]记录直径上各点到 da 的路径长度
int da,db;
bfs(1);
da=1;
for(int i=1;i<=n;i++)
    if(dist[i]>dist[da]) da=i;
bfs(da);
db=1;
for(int i=1;i<=n;i++)
    if(dist[i]>dist[db]) db=i;
top=0;
while(db!=da){st[top++]=db;db=from[db];}
st[top++]=da;
//将 st 反转过来
for(int i=0;i<(top-1)/2;i++) swap(st[i],st[top-1-i]);

//搜侧链
int l=0,r=dist[st[top-1]]; //r 直径的长度
memset(mark,0,sizeof mark);
for(int i=0;i<top;i++) mark[st[i]]=1;
bfs(da);
//l 侧链的最大长度
for(int i=1;i<=n;i++) l=max(l,dist[i]);
```

1.1.4 树的点分治

```
const int inf=0x3f3f3f3f;
int sum; //点分的子树总结点树,初值为 n
int rt; //重心,初始为 0,bal[0]=inf
int ans; //最后的结果
int sz[Maxn]; //树的节点数
int bal[Maxn]; //节点最多的子树的 sz
```

```
int vis[Maxn];
int d[Maxn];
int st[Maxn],top;
int k; //统计树上距离小于 k 的点对
void get(int u,int fa){ //得到重心
    sz[u]=1,bal[u]=0;
    for(int i=head[u];i!=-1;i=p[i].next){
        int v=p[i].to;
        if(v==fa||vis[v]) continue;
        get(v,u);
        sz[u]+=sz[v];
        bal[u]=max(bal[u],sz[v]);
    }
    bal[u]=max(bal[u],sum-sz[u]);
    if(bal[u]<bal[rt]) rt=u;
}
void dfs(int u,int fa){ //计算以 u 为根的子树节点到 u 的距离
    st[top++]=d[u]; //存入数组,方便统计
    for(int i=head[u];i!=-1;i=p[i].next){
        int v=p[i].to;
        if(v==fa||vis[v]) continue;
        d[v]=d[u]+p[i].w;
        dfs(v,u);
    }
}
//题目维护的信息必经过 u 的情况
//x 表示 u 和 fa[u]之间的信息
int cal(int u,int x){
    d[u]=x; //初始化为 x
    top=0;
    dfs(u,-1);
    sort(st,st+top); //排序
    int res=0;
    for(int l=0,r=top-1;l<r;){ //枚举 l,r 必单调减
        if(st[l]+st[r]<=k) res+=r-l,l++;
        else r--;
    }
    return res;
}
void work(int u){
    vis[u]=1;
    ans+=cal(u,0); //这里包含了同一颗子树
    for(int i=head[u];i!=-1;i=p[i].next){
        int v=p[i].to;
```

```
        if(vis[v]) continue;
        //u 算了 p[i].w,两个分支在同一颗子树也要算
        ans-=cal(v,p[i].w); //减去同一颗子树
        sum=sz[v]; //找子树 v 的重心前的初始化
        bal[rt=0]=inf;
        get(v,-1); //rt 存了重心
        work(rt);
    }
}
void solve(int n){ //调用前先建树
    memset(vis,0,sizeof vis);
    sum=n; //求重心前的初始化工作
    bal[rt=0]=inf;
    get(1,-1);
    ans=0;
    work(rt);
}
```

1.1.5 LCA

```
/******在线 RMQ*****/
/***O(nlogn)-O(1)***/
struct edge{
    int to,w,next;
}p[Maxn<<1];
int head[Maxn];
int tot;
void addedge(int a,int b,int c){
    p[tot].to=b;
    p[tot].w=c;
    p[tot].next=head[a];
    head[a]=tot++;
}
int pw[Maxn<<1]; //2^p[i]<=i
int dp[Maxn<<1][30]; //dp[i][j]表示[i,i+(1<<j)-1]深度值最小对应的下标
int es[Maxn<<1]; //欧拉序列
int d[Maxn<<1]; //深度数组
int fs[Maxn]; //fs[i]表示第一次在欧拉序列中出现位置
int num; //用于计数欧拉序列
int mymin(int x,int y){
    return d[x]<d[y]?x:y;
}
void rmq_init(int n){
    pw[0]=-1;
```

```
    for(int i=1;i<=n;i++)
        pw[i]=i&i-1?pw[i-1]:pw[i-1]+1;
    for(int i=1;i<=n;i++) dp[i][0]=i;
    for(int j=1;j<=pw[n];j++)
        for(int i=1;i+(1<<j)-1<=n;i++)
            dp[i][j]=mymin(dp[i][j-1],dp[i+(1<<j-1)][j-1]);
}
int rmq_ask(int l,int r){ //查询 l 和 r 的 LCA
    l=fs[l],r=fs[r]; //求得第一次出现的区间
    if(l>r) swap(l,r);
    int k=pw[r-l+1];
    return es[mymin(dp[l][k],dp[r-(1<<k)+1][k])];
}
void dfs(int u,int fa,int sp){ //sp 表示深度
    es[++num]=u; //第一次访问
    d[num]=sp;
    fs[u]=num; //维护第一次出现的位置
    for(int i=head[u];i!=-1;i=p[i].next){
        int v=p[i].to;
        if(v!=fa){
            dfs(v,u,sp+1);
            es[++num]=u; //子树返回
            d[num]=sp;
        }
    }
}
void init(){
    tot=num=0;
    memset(head,-1,sizeof head);
}
void LCA(int n){ //求得第一次出现的区间
    dfs(1,-1,0);
    rmq_init(2*n-1);
}

/*****离线 Tarjan*****/
/***复杂度 O(n+q)***/
struct edge{
    int fr,to,w,lca,next;
}p[Maxn<<1],ask[Maxm<<1];
int head[Maxn],ah[Maxn];
int tot,tot1;
void addedge(int a,int b,int c){
    p[tot].to=b;
```

```
    p[tot].w=c;
    p[tot].next=head[a];
    head[a]=tot++;
}
void addedge1(int a,int b){
    ask[tot1].fr=a;
    ask[tot1].to=b;
    ask[tot1].next=ah[a];
    ah[a]=tot1++;
}
int fa[Maxn];
int findset(int x){
    return fa[x]==x?x:(fa[x]=findset(fa[x]));
}
int vis[Maxn];
void LCA(int u,int f){
    fa[u]=u; //自己指向自己,不这样的话,访问 u 和 u 的 LCA 就可能出错
    for(int i=head[u];i!=-1;i=p[i].next){
        int v=p[i].to;
        if(f!=v){
            LCA(v,u);
            fa[v]=u;
        }
    }
    vis[u]=1; //设置已访问
    for(int i=ah[u];i!=-1;i=ask[i].next){ //处理与 u 关联的边
        int v=ask[i].to;
        if(vis[v]) //若 v 已访问,则说明 u,v 的 lca 是 v 所在集合的指向
            ask[i].lca=ask[i^1].lca=findset(v);
    }
}
void init(){
    tot=tot1=0;
    memset(head,-1,sizeof head);
    memset(ah,-1,sizeof ah);
    memset(vis,0,sizeof vis);
}

/*****非递归 Tarjan*****/
int cur[Maxn];
int pre[Maxn];
int st[Maxn];
void LCA(int x,int n){ //n 为节点数
    memcpy(cur,head,sizeof head);
```

```
    for(int i=1;i<=n;i++) fa[i]=i;
    int top;
    st[top=1]=x;
    pre[x]=-1;
    while(top){
        int u=st[top];
        bool flag=false;
        while(cur[u]!=-1){
            int i=cur[u],v=p[i].to;
            if(v==pre[u]){ //略过指向父亲的边
                cur[u]=p[i].next;
                continue;
            }
            st[++top]=v; //入栈
            pre[v]=u; //记录父亲
            cur[u]=p[i].next; //修改 cur
            flag=true;
            break;
        }
        if(flag) continue;
        //u 的所有孩子都已搜索完
        vis[u]=1; //设置 u 的访问标志
        for(int i=ah[u];i!=-1;i=ask[i].next){ //处理与 u 关联的查询
            int v=ask[i].to;
            if(vis[v])
                ask[i].lca=ask[i^1].lca=findset(v);
        }
        top--;
        if(top) fa[u]=st[top]; //u 子树搜索完,指向其父亲
    }
}

/*****LCA(倍增)*****/
/*****O(nlogn)-O(logn)*****/
int f[Maxn][20]; //f[i][j]表示 i 的 2^j 个父亲
int dep[Maxn]; //深度
void dfs(int u,int fa){
    f[u][0]=fa;
    dep[u]=dep[fa]+1; //保证 dep[0]=0
    for(int i=head[u];i!=-1;i=p[i].next){
        int v=p[i].to;
        if(v!=fa) dfs(v,u);
    }
}
```

```
void init(int n){ //预处理
    //dep[0]=0; //全局变量已为 0
    dfs(1,0); //父亲为 0
    for(int j=0;j<18;j++) //更新至 2^18=262144
        for(int i=1;i<=n;i++)
            if(!f[i][j]) f[i][j+1]=0;
            else f[i][j+1]=f[f[i][j]][j];
}
int LCA(int u,int v){
    if(dep[v]>dep[u]) //保证 v 的深度小
        swap(u,v);
    int df=dep[u]-dep[v],t=0;
    while(df){ //二进制拆分
        if(df&1) u=f[u][t];
        t++; //倍增
        df>>=1;
    }
    if(u==v) return u;
    for(int i=18;i>=0;i--) //二分逼近 LCA
        if(f[u][i]!=f[v][i]){
            u=f[u][i];
            v=f[v][i];
        }
    return f[u][0]; //最后逼近到 LCA 的儿子
}
```

1.1.6 树链剖分

```
struct edge{
    int to,next;
}p[Maxn<<1];
int head[Maxn];
int tot;
void addedge(int a,int b){
    p[tot].to=b;
    p[tot].next=head[a];
    head[a]=tot++;
}

int n;
int num[Maxn];
int c[Maxn];
void update(int x,int d){
```

```
    while(x<=n){
        c[x]+=d;
        x+=x&-x;
    }
}
void U(int l,int r,int d){
    update(l,d);
    update(r+1,-d);
}
int sum(int x){
    int res=0;
    while(x){
        res+=c[x];
        x-=x&-x;
    }
    return res;
}

int fa[Maxn]; //父亲
int son[Maxn]; //重儿子
int sz[Maxn]; //子树节点个数
int dep[Maxn]; //深度
//防止下标越界,请调用 dfs1(1,0)
void dfs1(int u,int pre){ //u->[1,n]
    sz[u]=1;
    fa[u]=pre;
    son[u]=0; //初始化为 0
    dep[u]=dep[pre]+1; //必须保证 dep[0]=0
    for(int i=head[u];i!=-1;i=p[i].next){
        int v=p[i].to;
        if(v==pre) continue;
        dfs1(v,u);
        sz[u]+=sz[v]; //更新 sz[u]
        if(sz[v]>sz[son[u]]) //必须保证 sz[0]=0,这样第一步才能正常更新
            son[u]=v; //更新 son[u]
    }
    //叶子节点不做 for,因此 son[u]=0
}
int tmpdfn;
int top[Maxn]; //重链的靠近树根端,单个点也视为重链
int dfn[Maxn]; //时间戳
void dfs2(int u,int pre){
    top[u]=pre;
    dfn[u]=++tmpdfn;
```

```
    if(son[u]) //存在重儿子
        dfs2(son[u],pre); //优先搜索重儿子
    for(int i=head[u];i!=-1;i=p[i].next){
        int v=p[i].to;
        if(v!=fa[u]&&v!=son[u]) //搜索轻边
            dfs2(v,v); //可能成为重链的 top 端,父亲是自己
    }
}
void init(){
    tmpdfn=0;
    //sz[0]=dep[0]=0; //全部变量已为 0
    dfs1(1,0);
    dfs2(1,1);
}
void solve(int a,int b,int w){ //a,b 为原先的 id
    while(top[a]!=top[b]){ //a,b 在不同的重链上
        if(dep[top[a]]<dep[top[b]])
            swap(a,b); //保证 a 重链顶端节点深度大
        U(dfn[top[a]],dfn[a],w); //维护区间[top[a],a]
        a=fa[top[a]]; //沿轻边往上走
    }
    //a,b 在同一条重链上
    if(dep[a]>dep[b]) swap(a,b); //保证 a 的深度小(时间戳小)
    U(dfn[a],dfn[b],w);
}

/*******非递归*******/
int q[Maxn],Top;
void bfs(int x){ //维护好 dep,sz,son,fa
    q[0]=x,Top=1;
    dep[x]=sz[x]=1;
    son[x]=fa[x]=0;
    for(int i=0;i<Top;i++){ //维护好 dep 和 fa
        int u=q[i];
        for(int j=head[u];j!=-1;j=p[j].next){
            int v=p[j].to;
            if(v==fa[u]) continue;
            q[Top++]=v;
            fa[v]=u;
            dep[v]=dep[u]+1;
            sz[v]=1;
            son[v]=0;
        }
    }
```

```
    for(int i=Top-1;i>=0;i--){ //维护好 sz 和 son
        int u=q[i];
        for(int j=head[u];j!=-1;j=p[j].next){
            int v=p[j].to;
            if(v==fa[u]) continue;
            sz[u]+=sz[v];
            if(sz[v]>sz[son[u]])
            son[u]=v;
        }
    }
}
int cur[Maxn],vis[Maxn];
void dfs(int x){ //维护好 top 和 dfn
    q[0]=x,Top=1;
    dfn[x]=++tmpdfn;
    memset(vis,0,sizeof vis);
    memcpy(cur,head,sizeof head);
    vis[x]=1;
    top[x]=x;
    while(Top){
        int u=q[Top-1];
        if(son[u]&&!vis[son[u]]){ //访问重儿子
            q[Top++]=son[u];
            dfn[son[u]]=++tmpdfn;
            top[son[u]]=top[u];
            vis[son[u]]=1;
            continue;
        }
        else{ //访问其他儿子
            if(cur[u]==-1){ //所有儿子访问完
                Top--;
                continue;
            }
            for(int i=cur[u];i!=-1;i=p[i].next){
                int v=p[i].to;
                cur[u]=p[i].next;
                if(!vis[v]){
                    q[Top++]=v;
                    vis[v]=1;
                    dfn[v]=++tmpdfn;
                    top[v]=v;
                    break;
                }
            }
```

```
        }
    }
}
void init(int n){ //树根可以选(n+1)/2
    tmpdfn=0;
    bfs((n+1)/2);
    dfs((n+1)/2);
}
```

1.2 拓扑排序

```
/************拓扑排序************/
/**复杂度 O(n+m),顶点数+边数*******/
int in[Maxn]; //入度
int q[Maxn],qt;
bool topu(int n){ //点的编号 1~n,调用前统计好 in
    qt=0;
    for(int i=1;i<=n;i++) //入度为零点入队
        if(!in[i]) q[qt++]=i;
    for(int i=0;i<qt;i++){
        int u=q[i];
        for(int j=head[u];~j;j=p[j].next){
            int v=p[j].to;
            if(--in[v]==0) q[qt++]=v; //新入度为零的点
        }
    }
    return qt==n; //是否存在拓扑关系
}
```

1.3 最短路

1.3.1 Dijkstra

```
/******Dijkstra(不存在负权的边)******/
/*******复杂度 O(n^2)*******/
int adj[Maxn][Maxn]; //记录边权
int vis[Maxn],dist[Maxn];
const int inf=0x3f3f3f3f;
//主函数中初始化
memset(adj,0x3f,sizeof adj);
//源点到每个点都存在最短路
```

```
void dijkstra(int u,int n){ //u 为源点,n 为顶点个数
    for(int i=1;i<=n;i++)  //初始化
        dist[i]=adj[u][i],vis[i]=0;
    vis[u]=1,dist[u]=0;
    for(int i=1;i<n;i++){ //n-1 趟添加点
        int minn=inf,v=0;
        for(int j=1;j<=n;j++)
            if(!vis[j]&&dist[j]<minn)
                minn=dist[j],v=j;
        vis[v]=1;
        if(v==0) break; //如果 v=0,则说明源点到有些点不存在最短路
        for(int j=1;j<=n;j++) //更新当前距离的最新情况
            if(!vis[j]&&dist[v]+adj[v][j]<dist[j])
                //dist[v]+adj[v][j]不爆 int
                dist[j]=dist[v]+adj[v][j];
    }
}
```

1.3.2 Bellman_ford

```
struct line{
    int u,v;
    double w;
    line(){}
    line(int uu,int vv,double ww):u(uu),v(vv),w(ww){}
}p[Maxn*Maxn];
/****bellman_ford(不存在负环)****/
/********复杂度 O(n*m)********/
//顶点编号从 0 开始，边的编号从 0 开始（邻接表版）
void bellman(int u,int n,int m){  //u 表示源点,n 表示顶点数,m 表示边数
    for(int i=0;i<n;i++)
        dist[i]=inf;
    dist[u]=0;
    bool flag=true;  //标记如果没有修改,说明已全部是最短路
    for(int i=1;i<n&&flag;i++){ //n-1 趟更新
        flag=false;
        for(int j=0;j<m;j++) //对每条边进行松弛
            if(dist[p[j].u]+p[j].w<dist[p[j].v]){
                //dist[p[j].u]+p[j].w 不爆 int
                dist[p[j].v]=dist[p[j].u]+p[j].w;
                flag=true; //有改变,修改标志
            }
    }
}
```

```
//顶点从 1 开始(邻接矩阵版)
/********复杂度 O(n^3)********/
void bellman(int u,int n){
    for(int i=1;i<=n;i++)
        dist[i]=inf;
    dist[u]=0,pre[u]=0; //源点
    for(int i=1;i<n;i++)
        for(int j=1;j<=n;j++)
            for(int k=1;k<=n;k++)
                if(adj[k][j]!=-1&&dist[k]+adj[k][j]<dist[j])
                    dist[j]=dist[k]+adj[k][j],pre[j]=k;
}

//构造字典序最小的解
//反向构图，在起点和终点插入点，直到插入点等于终点算法终止
int tot=0,fr=s,to=e,k;
shortest[tot++]=fr;
while(fr!=to){
   for(k=1;k<=n;k++)
       if(k!=fr&&adj[k][fr]!=-1&&adj[k][fr]+dist[k]==dist[fr]) break;
   shortest[tot++]=k;
   fr=k;
}
```

1.3.3 Spfa

```
/*****spfa(循环队列版)*****/
/*****复杂度 O(kn),k 平均为 2******/
const int inf=0x3f3f3f3f;
struct edge{
    int to,next,w;
    edge(){}
    edge(int t,int n,int ww):to(t),next(n),w(ww){}
}p[Maxn*Maxn];
int head[Maxn];
int dist[Maxn];
int vis[Maxn];
int cnt[Maxn]; //入队次数
int q[Maxn];
int tot;
void addedge(int a,int b,int c){
    p[tot].to=b;
    p[tot].w=c;
```

```
    p[tot].next=head[a];
    head[a]=tot++;
}
bool spfa(int u,int n){  //n 为顶点总数
    memset(vis,0,sizeof vis);
    memset(cnt,0,sizeof cnt);
    memset(dist,0x3f,sizeof dist);
    dist[u]=0,cnt[u]++;
    int s=0,e;
    q[e=1]=u;
    while(s!=e){
        int v=q[s=(s+1)%Maxn];
        vis[v]=0;
        for(int i=head[v];i!=-1;i=p[i].next){
            int r=p[i].to,w=p[i].w;
            if(dist[v]+w<dist[r]){
                dist[r]=dist[v]+w;
                if(++cnt[r]>=n) return false;//松弛超过 n-1 次,存在负圈
                if(!vis[r]){
                    vis[r]=1;
                    q[e=(e+1)%Maxn]=r;
                }
            }
        }
    }
    return true;
}
void init(){
    tot=0;
    memset(head,-1,sizeof head);
}
```

1.3.4 Floyd

```
/*********floyd*********/
/*****复杂度 O(n^3)******/
//当 i==j 时,adj[i][j]=inf,最短路
for(int k=1;k<=n;k++)
    for(int i=1;i<=n;i++)
        for(int j=1;j<=n;j++)
            adj[i][j]=min(adj[i][j],adj[i][k]+adj[k][j]);

//当 i==j 时,adj[i][j]=inf,最小值最大化
```

```
for(int k=0;k<n;k++)
    for(int i=0;i<n;i++)
        for(int j=0;j<n;j++)
            adj[i][j]=max(adj[i][j],min(adj[i][k],adj[k][j]));

//当 i==j 时,adj[i][j]=0,集合运算
for(int k=1;k<=n;k++)
    for(int i=1;i<=n;i++)
        for(int j=1;j<=n;j++)
            adj[i][j]|=adj[i][k]&adj[k][j];

//当 i==j 时,adj[i][j]=inf,最大值最小化
for(int k=0;k<n;k++)
    for(int i=0;i<n;i++)
        for(int j=0;j<n;j++)
            adj[i][j]=min(adj[i][j],max(adj[i][k],adj[k][j]));
```

1.4 最小生成树

1.4.1 Prim

```
/*********prime*********/
/*****复杂度 O(n^2)******/
for(int i=0;i<n;i++) adj[i][i]=-1;
int adj[Maxn][Maxn],lowcost[Maxn];
//加入生成树的点 dist=-1
void prim(int u,int n){
    int ans=0;
    for(int i=0;i<n;i++)
        lowcost[i]=adj[u][i];
    for(int i=1;i<n;i++){
        int minn=1<<30,v;
        for(int j=0;j<n;j++)
            if(lowcost[j]!=-1&&lowcost[j]<minn)
                minn=lowcost[j],v=j;
        ans+=minn;
        for(int j=0;j<n;j++)
            lowcost[j]=min(lowcost[j],adj[v][j]);
    }
    printf("%d\n",ans);
}
```

1.4.2 Kruskal

```
/******kruskal******/
/*****复杂度 O(eloge)*****/
struct line{
    int u,v,w;
    line(){}
    line(int uu,int vv,int ww):u(uu),v(vv),w(ww){}
    bool operator<(const line &a)const{
        return w<a.w;
    }
}edge[Maxn<<1];
int fa[Maxn];
void init(int n){   //顶点编号从 1 开始
    for(int i=1;i<=n;i++)
        fa[i]=i;
}
int findset(int x){
    return fa[x]==x?x:(fa[x]=findset(fa[x]));
}
void unionset(int a,int b){
    fa[a]=b;
}
//存在最小生成树
//不要忘了 init()和 sort()
int kruskal(int n,int tot){  //返回 MST 的权值之和
    int cnt=0,ans=0;
    for(int i=0;i<tot;i++){
        int a=findset(edge[i].u),b=findset(edge[i].v);
        if(a!=b){
            ans+=edge[i].w;
            unionset(a,b);
            if(++cnt==n-1) break;
        }
    }
    return ans;
}
```

1.4.3 次小生成树

```
/****次小生成树****/
/****复杂度 O(m*n^2)****/
#include<cstdio>
```

```
#include<iostream>
#include<cstring>
#include<algorithm>
#define Maxn 110
using namespace std;

struct line{
    int u,v,w;
    bool operator<(const line &a)const{
        return w<a.w;
    }
}p[Maxn*Maxn];
int fa[Maxn],head[Maxn],ed[Maxn],nxt[Maxn],use[Maxn*Maxn],
maxcost[Maxn][Maxn];
void init(int n){
    for(int i=1;i<=n;i++)
        fa[i]=head[i]=ed[i]=i,nxt[i]=-1;
}
int findset(int x){
    return fa[x]==x?x:(fa[x]=findset(fa[x]));
}
void unionset(int a,int b){
    fa[b]=a;
    nxt[ed[a]]=head[b];
    ed[a]=ed[b];
}
int kruskal(int n,int m){
    init(n);
    int ans=0,cnt=0;
    for(int i=0;i<m;i++){
        int a=findset(p[i].u),b=findset(p[i].v);
        if(a!=b){
            ans+=p[i].w;
            //合并时求瓶颈数组
            for(int j=head[a];j!=-1;j=nxt[j])
                for(int k=head[b];k!=-1;k=nxt[k])
                    maxcost[j][k]=maxcost[k][j]=p[i].w;
            unionset(a,b);
            use[i]=1;
            if(++cnt==n-1) break;
        }
    }
    return ans;
}
```

```
int main()
{
    int t,n,m;
    scanf("%d",&t);
    while(t--){
        scanf("%d%d",&n,&m);
        for(int i=0;i<m;i++)
            scanf("%d%d%d",&p[i].u,&p[i].v,&p[i].w);
        sort(p,p+n);
        memset(use,0,sizeof use);
        int ans=kruskal(n,m);
        bool flag=false;
        //枚举未在 MST 中的边,替换瓶颈边
        for(int i=0;i<m;i++)
            if(!use[i]&&p[i].w==maxcost[p[i].u][p[i].v])
               {flag=true;break;}
        if(flag) puts("Not Unique!");
        else printf("%d\n",ans);
    }
    return 0;
}

//大白书上给出了复杂度为 O(n^2)的解法
//先求 MST,然后树形 dp 求瓶颈数组
```

1.4.4 生成树计数

Kirchhoff 矩阵:

当 i==j 时,A[i][j]表示 i 的度数;

当 i!=j 时,若存在边(i,j),则 A[i][j]=A[j][i]=-1;否则

A[i][j]=A[j][i]=0;

Kirchhoff 矩阵的任意一个 n-1 阶主子式的行列式的绝对值为生成树个数,一般可以取去掉 Kirchhoff 矩阵的最后一行和最后一列的 n-1 阶主子式

1.5 图的连通性

1.5.1 图的深度优先遍历详解

```
/*
首先注意一点:每条边均会被遍历两次，正向一次(反向一次)。
正向边分两种:
①树边(dfs 树中的边)
②对反向边的反向遍历(先遍历反向边[返回祖先],之后再重复遍历一次)
反向边分两种:
①返回祖先(不包括父亲),先于正向边②的遍历
②返回父亲,晚于正向边①的遍历

通过 vis,dfn 数组和 fa 可以区别 4 种边
(假设当前正在访问节点为 u,孩子节点为 v):
①vis[v]=0,(u,v)为树边,正向边的第①种
②vis[v]=1,dfn[v]>dfn[u],(u,v)为正向边的第②种
③vis[v]=1,dfn[v]<dfn[u],v!=fa,(u,v)为反向边的第①种,返回祖先(不包括父亲)
④ vis[v]=1,dfn[v]<dfn[u],v==fa,(u,v)为反向边的第②种,返回父亲

并且有等式:
正向边的第①种=反向边的第②种
正向边的第②种=反向边的第①种
正向边的第①种+正向边的第②种=总边数
反向边的第①种+反向边的第②种=总边数

遍历所有边一遍:
可以选择:正向边的第①种(树边)+反向边的第①种(返回祖先)
 */
```

1.5.2 割点(含连通分量的统计)

```
/*
树边:dfs 树中的边
回边(反向边):原图中存在的但不在 dfs 树中的边
交叉边(横叉边):原图中不存在的边,在 dfs 树中横跨两个子树
*/

/*
low[u]表示从 u 或者 u 的子孙出发通过回边可以到达的最小深度优先数
```

```
low[u]=min(
    dfn[u],
    min(low[v]|v 为 u 的孩子节点),
    min(dfn[v]|(u,v)为回边)
)
割点的充要条件:
①根节点,有大于 1 个孩子节点
②非根节点,至少存在一个孩子节点,其 low 值>=父亲节点的 dfn 值
*/

/****割点****/
/****复杂度 O(n+m)****/
int dfn[Maxn];
int sub[Maxn]; //去掉该点的连通分量数
int tim; //时间戳
int dfs(int u){ //返回 low(u)
    int lowu;
    dfn[u]=lowu=++tim;
    for(int i=head[u];~i;i=p[i].next){
        int v=p[i].to;
        if(!dfn[v]){ //u->v 为树边
            int lowv=dfs(v);
            lowu=min(lowu,lowv);
            if(lowv>=dfn[u]) sub[u]++;
        }
        else //u->v 为反向边
            lowu=min(lowu,dfn[v]);
    }
    return lowu;
}
void init(int n){ //n 为节点数
    tim=0;
    memset(dfn,0,sizeof dfn);
    memset(sub,0,sizeof sub);
    dfs(1);
    //除根节点外,其余节点还有父亲方向的连通分量
    for(int i=2;i<=n;i++) sub[i]++;
}
```

1.5.3 点双连通分量

```
/***点双连通分量***/
/***复杂度 O(n+m)***/
typedef pair<int,int> pii;
```

```
int dfn[Maxn];
int cut[Maxn]; //是否是割点
int id[Maxn]; //属于点双连通分量的编号
vector<int> bcc[Maxn]; //存每个点双连通分量的点
stack<pii> st;
int tim; //时间戳
int cnt; //点双连通分量个数,1,2,..,cnt
/*** 使用 dfs 需保证图的点数大于 1 ***/
int dfs(int u,int fa){ //返回 low(u)
    int lowu;
    dfn[u]=lowu=++tim;
    int son=0;
    for(int i=head[u];~i;i=p[i].next){
        int v=p[i].to;
        pii cur=make_pair(u,v),tmp;
        if(!dfn[v]){ //u->v 为树边
            son++;
            st.push(cur);
            int lowv=dfs(v,u);
            lowu=min(lowu,lowv);
            if(lowv>=dfn[u]){ //u 为割点
                cut[u]=1;
                ++cnt;
                bcc[cnt].clear(); //清空上组数据
                do{ //得到编号为 cnt 的点双连通分量
                    tmp=st.top();
                    st.pop();
                    int x=tmp.first,y=tmp.second;
                    if(id[x]!=cnt){
                        bcc[cnt].push_back(x);
                        id[x]=cnt;
                    }
                    if(id[y]!=cnt){
                        bcc[cnt].push_back(y);
                        id[y]=cnt;
                    }
                }while(tmp!=cur);
            }
        }
        else if(dfn[v]<dfn[u]&&v!=fa){ //不指向父亲的反向边
            st.push(cur);
            lowu=min(lowu,dfn[v]);
        }
    }
```

```
    if(son==1&&!fa) cut[u]=0; //根节点为割点,孩子数大于1
    return lowu;
}
void init(int n){ //n为节点数
    tim=cnt=0;
    memset(dfn,0,sizeof dfn);
    memset(id,0,sizeof id);
    memset(cut,0,sizeof cut);
    for(int i=1;i<=n;i++)
        if(!dfn[i]) dfs(i,0);
}
```

1.5.4 割边(含多重边)

```
/****前向星,根节点为1号点****/
/****复杂度O(n+m)****/
struct line{
    int to,next,tag,id;
}p[Maxn*20];
int tot,tmpdfn,num;
int head[Maxn],vis[Maxn],dfn[Maxn],low[Maxn],st[Maxn*10];
void addedge(int a,int b,int id){
    for(int i=head[a];i!=-1;i=p[i].next)
        if(p[i].to==b) {p[i].tag++;return;}
    p[tot].next=head[a];
    p[tot].to=b;
    p[tot].tag=1;
    p[tot].id=id;
    head[a]=tot++;
}
void dfs(int u,int fa){
    dfn[u]=low[u]=tmpdfn;
    vis[u]=1;
    for(int i=head[u];i!=-1;i=p[i].next){
        int v=p[i].to;
        if(!vis[v]){
            tmpdfn++;
            dfs(v,u);
            low[u]=min(low[u],low[v]);
            //割边条件:low>dfn并且不能是重边
            if(low[v]>dfn[u]&&p[i].tag==1)
                st[num++]=p[i].id;
        }
        else if(v!=fa) //求割边这条回边不能算
```

```
            low[u]=min(low[u],dfn[v]);
        }
    }
    void init(){
        tot=tmpdfn=num=0;
        memset(head,-1,sizeof head);
        memset(vis,0,sizeof vis);
    }
```

1.5.5 点连通度

```
/*
独立轨:设 A,B 为无向图的两个顶点,从 A 到 B 的 2 条没有公共内部顶点的路径,
互称为独立轨,A 到 B 独立轨的最大条数,记做 P(A,B)

Menger 定理:无向图 G 的顶点连通度 K(G)和顶点间最大独立轨的存在以下关系,
当 G 为完全图时, K(G)=|V(G)|-1;当 G 为非完全图时, K(G)=min{P(A,B)}{(AB)∉E}
*/
```

1.5.6 强连通分量

```
/*****Tarjan(前向星版)*****/
/*****复杂度 O(n+m)*****/
int dfn[Maxn]; //时间戳
int low[Maxn]; //u 和 u 的子孙能返回的最低深度
int belong[Maxn]; //属于哪个连通分量
int in[Maxn]; //是否在栈中
int st[Maxn]; //栈
int tmpdfn; //时间戳,从 1 开始
int top;
int scc;
void tarjan(int u){
    dfn[u]=low[u]=++tmpdfn;
    st[++top]=u;
    in[u]=1;
    for(int i=head[u];i!=-1;i=p[i].next){
        int v=p[i].to;
        if(!dfn[v]){ //未访问
            tarjan(v);
            low[u]=min(low[u],low[v]);
        }
        else if(in[v]) //反向边
```

```
            low[u]=min(low[u],dfn[v]);
    }
    if(dfn[u]==low[u]){ //找到强连通分量
        scc++;
        do{
            in[st[top]]=0;
            belong[st[top]]=scc;
        }while(st[top--]!=u);
    }
}
int scc_cnt(int n){
    memset(dfn,0,sizeof dfn);
    memset(in,0,sizeof in);
    scc=0;
    for(int i=1;i<=n;i++){ //保证每个点都被搜索
        tmpdfn=top=0;
        if(!dfn[i]) tarjan(i);
    }
    return scc;
}

/*****非递归版本*****/
stack<int> ss;
int nx[Maxn]; //下一个儿子
int lx[Maxn]; //上一个儿子
void tarjan(int u){
    while(!ss.empty()) ss.pop();
    ss.push(u);
    for(int i=1;i<=n;i++) nx[i]=head[i],lx[i]=-1; //n为节点总数
    while(!ss.empty()){
        int v=ss.top();
        if(!dfn[v]){ //第一次访问
            dfn[v]=low[v]=++tmpdfn;
            st[++top]=v;
            in[v]=1;
        }
        if(lx[v]!=-1) low[v]=min(low[v],low[lx[v]]); //访问完儿子
        if(nx[v]!=-1){ //有儿子
            while(nx[v]!=-1){ //寻找下一个还未访问的儿子
                if(!dfn[p[nx[v]].to]){ //树边
                    lx[v]=p[nx[v]].to;
                    nx[v]=p[nx[v]].next;
                    ss.push(lx[v]);
                    break;
```

```
                }
                else if(in[p[nx[v]].to]) //回边
                    low[v]=min(low[v],dfn[p[nx[v]].to]);
                nx[v]=p[nx[v]].next;
            }
        }
        else{ //全部儿子访问完毕
            if(low[v]==dfn[v]){
                scc++;
                do{
                    belong[st[top]]=scc;
                    in[st[top]]=0;
                }while(st[top--]!=v);
            }
            ss.pop();
        }
    }
}

/*****Kosaraju*****/
/*****复杂度 O(n+m)*****/
struct edge{
    int t1,t2,nx1,nx2;
}p[Maxn*Maxn<<1];
int hd1[Maxn]; //原图
int hd2[Maxn]; //逆图
int tot;
void addedge(int a,int b){
    p[tot].t1=b;
    p[tot].nx1=hd1[a];
    hd1[a]=tot++;
    p[tot].t2=a;
    p[tot].nx2=hd2[b];
    hd2[b]=tot++;
}
int st[Maxn];
int belong[Maxn];  //属于哪个连通分量
int vis[Maxn];
int top; //栈顶
int scc; //连通分量个数
void dfs1(int u){ //求原图 dfs 后序遍历序列
    vis[u]=1;
    for(int i=hd1[u];i!=-1;i=p[i].nx1){
        int v=p[i].t1;
```

```
        if(!vis[v]) dfs1(v);
    }
    st[++top]=u;
}
void dfs2(int u){ //对逆图按 dfs 后序递减搜索
    vis[u]=1;
    belong[u]=scc;
    for(int i=hd2[u];i!=-1;i=p[i].nx2){
        int v=p[i].t2;
        if(!vis[v]) dfs2(v);
    }
}
int kosaraju(int n){
    top=scc=0;
    memset(vis,0,sizeof vis);
    for(int i=1;i<=n;i++)
        if(!vis[i]) dfs1(i);
    memset(vis,0,sizeof vis);
    for(int i=top;i>0;i--)
        if(!vis[st[i]]){
            scc++;
            dfs2(st[i]);
        }
    return scc;
}
```

1.5.7 2-SAT

1.5.8 欧拉回路

```
/***无向图欧拉路***/
/***Hierholzer 算法***/
/***复杂度 O(n+m)***/
int tmp[Maxm]; //临时栈,栈中可能点数可能达到边数
int ans[Maxm]; //最终栈,栈中可能点数可能达到边数
int cur[Maxn]; //记录下一条要搜索的边
//这里用 i 来存正向边,i^1 来存反向边
int vis[Maxm<<1]; //记录边是否被访问
int tp,ap;
void Hierholzer(int start){
    tp=ap=0;
    memcpy(cur,head,sizeof head);
```

```
    memset(vis,0,sizeof vis);
    tmp[tp++]=start; //选择起点
    while(tp){
        bool flag=false;
        int u=tmp[tp-1];
        for(int v=cur[u];v!=-1;v=p[v].next){
            if(vis[v]) continue;
            vis[v]=vis[v^1]=1; //记录已访问边
            tmp[tp++]=p[v].to;
            cur[u]=p[v].next; //下一条需要访问的边
            flag=true;
            break;
        }
        if(!flag){ //u 已经搜索完
            ans[ap++]=u;
            tp--;
        }
    }
}
```

1.6 网络流

1.6.1 基本概念

```
/*
弧的容量:每条弧的最大承载力(权值),用 c(u,v)表示
弧的流量:每条弧的实际传输量,用 f(u,v)表示
容量网络:弧的容量组成的网络
网络流:弧上流量的集合
规定:链的正方向为源点到汇点的方向
前向弧:方向与链的正方向一致的弧
反向弧:方向与链的正方向相反的弧
增广路(可改进路):所有前向弧都是非饱和弧,所有后向弧都是非零弧
残留网络:原图中弧(c(u,v),f(u,v))对应残留网络中与其同向的流量为 c(u,v)-f(u,v)的弧和与其反向的 f(v,u)的弧
割(s-t 割):原图被分成两个子集 T,S(S 含有源点 Vs,T 含有汇点 Vt),[S,T]代表一个边集合{⟨u,v⟩|⟨u,v⟩ ∈ E,u ∈ S,v ∈ T}
割的前向弧:< u,v >,u ∈ Vs,v ∈ Vt
割的反向弧:< u,v >,u ∈ Vt,v ∈ Vs
割的容量:所有前向弧的容量总和
割的净流量:所有弧的流量总和
顶点的层次:在残留网络中,从源点到该顶点的最短路径长度(边的数目,不是指
```

权值)
层次网络:对残留网络分层后,删去比汇点层次更高或者同层的顶点(保留汇点),并删去与这些顶点关联的弧,再删去从某层顶点指同层顶点和底层顶点的弧,其余不变
阻塞流:当可行流的层次网络不存在增广路时, 该可行流为层次网络的阻塞流
允许弧:在反向标号的残留网络中,边<u,v>满足 d[u]=d[v]+1; 在正向标号的层次网络中,边<u,v>满足 d[u]+1=d[v]
赢余:流入的总流量减去流出的总流量
活跃顶点:赢余大于 0, 并且不是源点和汇点
预流:除源点和汇点外, 其余点赢余都大于等于 0 的网络流
饱和推进:推进的流量等于弧上的残留容量

最大流最小割定理:网络流流量等于任意割净流量,割的净流量小于等于割的容量,因此网络流流量小于等于任意割的容量,等号成立当且仅当网络流流量达到最大,割的容量达到最小,也就是最大流等于最小割。
最大流:原网络不存在增广路,残留网络不存在从源点到汇点的路径

限制条件:
①0<=f(u,v)<=c(u,v)
②源点流出的总流量等于汇点流入的总流量
③除源点、汇点外, 其他点流量守恒(流入等于流出)
*/

1.6.2 Edmonds_Karp

```
/*****Edmonds_Karp*****/
/*****复杂度 O(n*m^2)*****/
const int inf=0x3f3f3f3f;
int cap[Maxn][Maxn]; //边容量
int flow[Maxn][Maxn]; //边实际流量
int pre[Maxn]; //增广路节点前驱
int alpha[Maxn]; //改进量
int q[Maxn]; //队列
int m,n; //边,节点数
void init(){
    memset(cap,0,sizeof cap);
    memset(flow,0,sizeof flow);
}
int Edmonds_Karp(int src,int t){ //源点,汇点
    int maxflow=0;
    while(1){
        memset(alpha,0,sizeof alpha);
        alpha[src]=inf; //源点初始化无穷大
        pre[src]=-1;
```

```
        int s=0,e=-1;
        q[++e]=src;
        while(s<=e&&!alpha[t]){ //bfs 找增广路
            int u=q[s++];
            for(int i=1;i<=n;i++){ //节点编号[1,n]须根据具体题目修改
                if(!alpha[i]&&flow[u][i]<cap[u][i]){
                    pre[i]=u;
                    alpha[i]=min(alpha[u],cap[u][i]-flow[u][i]);
                    q[++e]=i;
                }
            }
        }
        if(!alpha[t]) break; //找不到增广路
        int k=t;
        while(pre[k]!=-1){
            flow[pre[k]][k]+=alpha[t];
            flow[k][pre[k]]-=alpha[t];
            k=pre[k];
        }
        maxflow+=alpha[t];
    }
    return maxflow;
}
```

1.6.3 Dinic(递归+多路增广)

```
/*****Dinic*****/
/*****复杂度 O(n^2*m)*****/
const int inf=0x3f3f3f3f;
struct line{
    int to,next,cap;
}p[Maxn*Maxn*2]; //完全图(反向边)乘 2 倍
int head[Maxn]; //初始化-1
int q[Maxn]; //BFS 队列
int d[Maxn]; //层次
int tot; //初始化 0
int src,t; //源点,汇点
int n,m; //边,节点数
void addedge(int a,int b,int c){
    p[tot].to=b;
    p[tot].next=head[a];
    p[tot].cap=c;
    head[a]=tot++;
}
```

```
void insert(int a,int b,int c){
    addedge(a,b,c);
    addedge(b,a,0);
}
bool bfs(){
    memset(d,-1,sizeof d);
    int s=0,e=-1;
    q[++e]=src;
    d[src]=0;
    while(s<=e){
        int u=q[s++];
        for(int i=head[u];i!=-1;i=p[i].next){
            int v=p[i].to;
            if(d[v]==-1&&p[i].cap){
                d[v]=d[u]+1;
                q[++e]=v;
            }
        }
    }
    return d[t]!=-1;
}
int dfs(int u,int alpha){ //dfs 返回值小于等于 alpha
    if(u==t) return alpha;
    int w,used=0; //used 不可能大于 alpha
    for(int i=head[u];i!=-1&&used<alpha;i=p[i].next){
    //当 used=alpha,直接退出
        int v=p[i].to;
        if(p[i].cap&&d[v]==d[u]+1){
            w=dfs(v,min(alpha-used,p[i].cap)); //w<=alpha-used
            used+=w; //used<=alpha
            p[i].cap-=w;
            p[i^1].cap+=w;
        }
    }
    if(!used) d[u]=-1;
    return used;
}
//源点和汇点必须不同,否则死循环
int dinic(){ //不要忘记初始化 tot 和 head 数组
    int ans=0;
    src=1,t=n; //初始化源点和汇点
    while(bfs())
        ans+=dfs(src,inf);
    return ans;
```

```
}

/*****Dinic(当前弧优化)*****/
/*****复杂度 O(n^2*m)*****/
const int inf=0x3f3f3f3f;
struct line{
    int to,next,cap;
}p[Maxn*Maxn*2]; //完全图(反向边)乘 2 倍
int head[Maxn]; //初始化-1
int q[Maxn]; //BFS 队列
int d[Maxn]; //层次
int cur[Maxn]; //当前弧
int tot; //初始化 0
int src,t; //源点,汇点
int n,m; //边,节点数
void addedge(int a,int b,int c){
    p[tot].to=b;
    p[tot].next=head[a];
    p[tot].cap=c;
    head[a]=tot++;
}
void insert(int a,int b,int c){
    addedge(a,b,c);
    addedge(b,a,0);
}
bool bfs(){
    memset(d,-1,sizeof d);
    int s=0,e=-1;
    q[++e]=src;
    d[src]=0;
    while(s<=e){
        int u=q[s++];
        for(int i=head[u];i!=-1;i=p[i].next){
            int v=p[i].to;
            if(d[v]==-1&&p[i].cap){
                d[v]=d[u]+1;
                q[++e]=v;
            }
        }
    }
    return d[t]!=-1;
}
int dfs(int u,int alpha){ //dfs 返回值小于等于 alpha
    if(u==t) return alpha;
```

```
    int w,used=0; //used 不可能大于 alpha
    for(int i=cur[u];i!=-1&&used<alpha;i=p[i].next){
    //当 used=alpha,直接退出
        int v=p[i].to;
        if(p[i].cap&&d[v]==d[u]+1){
            w=dfs(v,min(alpha-used,p[i].cap)); //w<=alpha-used
            used+=w; //used<=alpha
            p[i].cap-=w;
            p[i^1].cap+=w;
            cur[u]=i; //修改当前弧
        }
    }
    if(!used) d[u]=-1;
    return used;
}
//源点和汇点必须不同,否则死循环
int dinic(){ //不要忘记初始化 tot 和 head 数组
    int ans=0;
    src=1,t=n; //初始化源点和汇点
    while(bfs()){
        for(int i=src;i<=t;i++) cur[i]=head[i]; //当前弧初始化
        ans+=dfs(src,inf);
    }
    return ans;
}
```

1.6.4 Isap(递归+多路增广)

```
/*****Isap(递归+gap 优化+无 BFS)*****/
/*****复杂度 O(n^2*m)*****/
const int inf=0x3f3f3f3f;
struct line{
    int to,next,cap;
}p[Maxn*Maxn*2];
int head[Maxn]; //初始化为-1
int d[Maxn]; //层次
int gap[Maxn]; //gap[x]为 d[]=x 出现的个数
int tot; //初始化为 0
int src,t; //源点,汇点
int n,m; //边,节点数
void addedge(int a,int b,int c){
    p[tot].to=b;
    p[tot].next=head[a];
```

```
    p[tot].cap=c;
    head[a]=tot++;
}
void insert(int a,int b,int c){
    addedge(a,b,c);
    addedge(b,a,0);
}
int dfs(int u,int alpha){ //dfs 返回值小于等于 alpha
    if(u==t) return alpha;
    int w,used=0; //used 不可能大于 alpha
    int mind=n-1; //最低层次,使用 n
    for(int i=head[u];i!=-1&&used<alpha&&d[src]<n;i=p[i].next){
        //当 used=alpha,直接退出,使用 n
        int v=p[i].to;
        if(p[i].cap){
            if(d[v]+1==d[u]){
                w=dfs(v,min(alpha-used,p[i].cap)); //w<=alpha-used
                used+=w; //used<=alpha
                p[i].cap-=w;
                p[i^1].cap+=w;
            }
            mind=min(mind,d[v]); //不存在允许弧起作用
        }
    }
    if(!used){ //没有允许弧
        if(--gap[d[u]]==0) d[src]=n; //出现断层,使用 n
        gap[d[u]=mind+1]++;
    }
    return used;
}
int isap(){
    int res=0;
    src=n,t=n+1; //源点,汇点需修改
    memset(d,0,sizeof d);
    memset(gap,0,sizeof gap);
    n+=2; //必须修改为点的个数
    gap[0]=n; //使用 n
    while(d[src]<n) //使用 n
        res+=dfs(src,inf);
    return res;
}

/*****Isap(递归+gap 优化+cur 优化+BFS 初始化)*****/
/*****复杂度 O(n^2*m)*****/
```

```
const int inf=0x3f3f3f3f;
struct line{
    int to,next,cap;
}p[Maxn*Maxn*2];
int head[Maxn]; //初始化为-1
int d[Maxn]; //层次
int q[Maxn]; //队列
int cur[Maxn]; //当前弧
int gap[Maxn]; //gap[x]为 d[]=x 出现的个数
int tot; //初始化为 0
int src,t; //源点,汇点
int n,m; //边,节点数
void addedge(int a,int b,int c){
    p[tot].to=b;
    p[tot].next=head[a];
    p[tot].cap=c;
    head[a]=tot++;
}
void insert(int a,int b,int c){
    addedge(a,b,c);
    addedge(b,a,0);
}
void bfs(){
    memset(d,-1,sizeof d);
    memset(gap,0,sizeof gap);
    int s=0,e=-1;
    q[++e]=t;
    gap[d[t]=0]=1; //汇点为 0 层
    while(s<=e){
        int u=q[s++];
        for(int i=head[u];i!=-1;i=p[i].next){
            int v=p[i].to;
            if(d[v]==-1&&!p[i].cap){ //利用反向边
                gap[d[v]=d[u]+1]++;
                q[++e]=v;
            }
        }
    }
}
int dfs(int u,int alpha){ //dfs 返回值小于等于 alpha
    if(u==t) return alpha;
    int w,used=0; //used 不可能大于 alpha
    for(int i=cur[u];i!=-1&&used<alpha&&d[src]<n;i=p[i].next){
        //当 used=alpha,直接退出,使用 n
```

```
            int v=p[i].to;
            if(p[i].cap&&d[v]+1==d[u]){
                w=dfs(v,min(alpha-used,p[i].cap)); //w<=alpha-used
                used+=w; //used<=alpha
                p[i].cap-=w;
                p[i^1].cap+=w;
                if(used==w) cur[u]=i;
            }
        }
        if(!used){ //没有允许弧
            int mind=n-1; //最低层次,使用 n
            for(int i=head[u];i!=-1;i=p[i].next)
                if(p[i].cap&&d[p[i].to]<mind){
                    mind=d[p[i].to];
                    cur[u]=i;
                }
            if(--gap[d[u]]==0) d[src]=n; //出现断层,使用 n
            gap[d[u]=mind+1]++;
        }
        return used;
}
int isap(){
        int res=0;
        src=n,t=n+1; //源点,汇点需修改
        bfs();
        n+=2; //必须修改为点的个数
        memcpy(cur,head,sizeof head);
        while(d[src]<n) //使用 n
            res+=dfs(src,inf);
        return res;
}
```

1.6.5 Push_relabel

```
/*****push_relabel(邻接矩阵版)*****/
/*****复杂度 O(n^2*m)*****/
const int inf=0x3f3f3f3f;
int cap[Maxn][Maxn]; //残留容量
int ef[Maxn]; //顶点余流
int d[Maxn]; //顶点高度
int n; //点的个数
void init(int x){
    n=x; //须修改
    memset(cap,0,sizeof cap);
```

```
    memset(d,0,sizeof d);
    memset(ef,0,sizeof ef);
}
int push_relabel(int src,int t){
    int maxflow=0;
    d[src]=n; //源点高度为 n
    ef[src]=inf; //源点初始余流
    queue<int> q;
    q.push(src);
    while(!q.empty()){
        int u=q.front();
        q.pop();
        int mind=inf; //最低高度
        for(int v=0;v<n;v++){ //下标[0...n)
            int alpha=min(cap[u][v],ef[u]);
            if(alpha>0){ //cap[u][v]>0
                if(u==src||d[u]==d[v]+1){
                    cap[u][v]-=alpha;
                    cap[v][u]+=alpha;
                    //如果到达汇点,就将该流值加入最大流
                    if(v==t) maxflow+=alpha;
                    ef[u]-=alpha;
                    ef[v]+=alpha;
                    if(v!=src&&v!=t) q.push(v);
                }
                else mind=min(mind,d[v]); //不可推进点
            }
        }
        //如果不是源点/汇点且仍有余流,则重标记高度再进队
        if(u!=src&&u!=t&&ef[u]>0){
            d[u]=mind+1;
            q.push(u);
        }
    }
    return maxflow;
}
```

1.6.6 HLPP

```
/*****HLPP(邻接矩阵版)*****/
/*****复杂度 O(n^2*sqrt(m))*****/
struct List{
    int u,d;
```

```
    List(int uu,int dd):u(uu),d(dd){}
    bool operator<(const List &x)const{
        return d<x.d;
    }
};
const int inf=0x3f3f3f3f;
int cap[Maxn][Maxn]; //残留容量
int ef[Maxn]; //顶点余流
int d[Maxn]; //顶点高度
int n; //点的个数
void init(int x){
    n=x; //须修改
    memset(cap,0,sizeof cap);
    memset(d,0,sizeof d);
    memset(ef,0,sizeof ef);
}
int HLPP(int src,int t){
    int maxflow=0;
    d[src]=n; //源点高度为n
    ef[src]=inf; //源点初始余流
    //queue<int> q;
    priority_queue<List> q; //优先队列
    q.push(List(src,d[src]));
    while(!q.empty()){
        int u=q.top().u;
        q.pop();
        int mind=inf; //最低高度
        for(int v=0;v<n;v++){ //下标[0...n)
            int alpha=min(cap[u][v],ef[u]);
            if(alpha>0){ //cap[u][v]>0
                if(u==src||d[u]==d[v]+1){
                    cap[u][v]-=alpha;
                    cap[v][u]+=alpha;
                    //如果到达汇点,就将该流值加入最大流
                    if(v==t) maxflow+=alpha;
                    ef[u]-=alpha;
                    ef[v]+=alpha;
                    if(v!=src&&v!=t) q.push(List(v,d[v]));
                }
                else mind=min(mind,d[v]); //不可推进点
            }
        }
        //如果不是源点/汇点且仍有余流,则重标记高度再进队
        if(u!=src&&u!=t&&ef[u]>0){
```

```
            d[u]=mind+1;
            q.push(List(u,d[u]));
        }
    }
    return maxflow;
}
```

1.6.7 最小割

```
//vis[]=1 为 S 集合的元素,vis[]=0 为 T 集合的元素
int vis[Maxn];
void DFS(int u){ //搜索割[S,T]
    vis[u]=1;
    for(int i=head[u];i!=-1;i=p[i].next){
        int v=p[i].to;
        if(vis[v]||!p[i].cap) continue;
        DFS(v);
    }
}
```

有向图闭合图子图：原图的一个点集 V，对原图中任何一条出边(u,v)

$$u \in V \Rightarrow v \in V$$

简单割：割中的每条边都不属于原图，即一个端点或者是源点或者是汇点
闭合图与简单割一一对应
最大权闭合图：原图中正权点连向源点，负权点连向汇点，边权为点权，原图中的每条边保留，但边权改为 inf，则最小割+闭合图最大权=所有正权点权值之和
具体方案：割[S,T]的 S 包含的点集(除源点外)

无向图图的密度：边数/点数
最大密度子图：选择(E,V)

$$\max\left\{\frac{\sum_{e\in E} x_e}{\sum_{v\in V} x_v}\right\}$$

01 规划：$h(g)=\max_x\left\{\sum_{e\in E} x_e - g\sum_{v\in V} x_v\right\}$

定理：任何一个图的两个不同的子图，密度至少相差 $\frac{1}{n^2}$，n 为顶点数

因此二分的 eps 可设为 $\frac{1}{n^2}$

建图：设原图的顶点数为 n，边数为 m，原图中的无向边变成两条容量为 1 的有向边，源点到每个点连边，容量为 m，每个点向汇点连边，对于点 v，容量为 $m+2g-d_v,d_v$ 为点 v 的度数，g 为二分值，那么

$$h(g)=\frac{mn-c[S,T]}{2}$$

最大化 h(g)就是求最小割，然后根据 h(g)的单调递减的性质，调整二分上下限
具体方案：割[S,T]的 S 包含的点集(除源点外)

推广 1：带正边权的图的密度 $\frac{\sum w}{|V|}$ ，设原图的顶点数为 n，边数为 m，原图中的无向边变成两条容量为边权 w 的有向边，源点到每个点连边，容量为 $\sum w$，每个点向汇点连边，对于点 v，容量为 $\sum w+2g-d_v,d_v$ 为 $\sum_{(u,v)\in e} w_e$，g 为二分值，那么

$$h(g)=\frac{n\sum w-c[S,T]}{2}$$

二分精度需要自己设置

推广 2：带正边权和点权(可负)的图的密度 $\frac{\sum p+\sum w}{|V|}$ ，设原图的顶点数为 n，边数为 m，原图中的无向边变成两条容量为边权 w 的有向边，源点到每个点连边，容量为 $U=2\sum|p|+\sum w$，每个点向汇点连边，对于点 v，容量为 $U+2g-d_v-2p_v,d_v$ 为 $\sum_{(u,v)\in e} w_e$，p_v 为点 v 的权值，g 为二分值，那么

$$h(g)=\frac{nU-c[S,T]}{2}$$

二分精度需要自己设置

二分图最小点权覆盖：源点向 X 部的点连边，边权为 X 部的点权，Y 部向汇点连边，边权为 Y 部的点权，X 部向 Y 部连边，边权为 inf，最小割就等于最小点权覆盖

二分图最大点权独立：转化为最小点权覆盖，最大点权独立+最小点权覆盖=全部点的权值和，即两者互为补集关系

1.6.8 上下界网络流

1.无源汇上下界可行流

由于无源汇，因此可行流中任一点都满足流量平衡,设边(u,v)的容量下界是 B(u,v),容量上界是 C(u,v),我们考虑将(u,v)的容量拆成两部分,其中一部分是必须的 B(u,v),另一部分是自由的 g(u,v),对于 u 这点,满足流量平衡,可以得到

$$\sum_{(i,u)\in E}[B(i,u)+g(i,u)]=\sum_{(u,j)\in E}[B(u,j)+g(u,j)]$$

$$\Leftrightarrow \sum_{(i,u)\in E}B(i,u)-\sum_{(u,j)\in E}B(u,j)=\sum_{(u,j)\in E}g(u,j)-\sum_{(i,u)\in E}g(i,u)$$

设 $M[u]=\sum_{(i,u)\in E}B(i,u)-\sum_{(u,j)\in E}B(u,j)$,

考虑在自由流组成的网络中求得可行流,然后累加上每条边的下界来得到可行流,但这样无法保证每个点的流量平衡。于是需要加入附加源点和附加汇点。

若 M[u]>0,说明流入点 u 的下界流多,那么为了达到平衡,自由流必须流出去的多,所以由附加源向 u 连一条容量为 M[u]的边,保证点 u 有多流出 M[u]的流量

若 M[u]<0,说明流出点 u 的下界流多,那么为了达到平衡,自由流必须流入的多,所以由点 u 向附加汇连一条容量为 -M[u] 的边,保证点 u 多流入 M[u]能够流出到附加汇

然后保留原图的边,容量改为自由流,这样在有附加源汇的新图能够达到满流,则对应在原图得到一个可行流,否则,原图不存在可行流

注意: 边(u,v)的实际流量是 B(u,v)+g(u,v)

2.有源汇上下界可行流

增加一条由汇点连向源点的下界为 0,上界为 inf 的边,这样不改变原图的可行流同时使得源汇也满足流量平衡的条件,于是采用无源汇上下界可行流的方法即可

3.有源汇上下界最大流

方法一: 按有源汇上下界可行流方法判断可行流是否存在,若存在则对原图的源点和汇点做一遍最大流,扩大之前的可行流,每条边的实际流量应加上下界。注意: 最大流不是直接加上源点出去边的反向边的流量,而是要多减去一个 inf,因为第一遍判断可行流时,加了一条汇点连向源点的下界为 0,上界为 inf 的边,这条边在第一次被增广后,而后第二遍求最大流时,第一次被增广的流量又回到汇点,因此源点通过这条边流向汇点的流量为 0,即反向边流量为 inf,所以这条边被加了一次,因此要减掉 inf,在计算最大流时是否要考虑加上源点连出去边的下界,取决于是否删除附加源和附加汇,若删除了则需加上下界,否则因为源点连出的下界和已记录在源点连向附加汇的边上,并且在第一遍判可行流时已被增广,所以无需再加(是否在做第二遍最大流前删除附加源和附加汇以及它们的所有连边并不影响第二遍的增广,因为不存在从源点经过这些边到汇点的增广路,只是如果不删除的话,源点和汇点(包括那些下界出入不平衡的中间点)的相邻边会增加,也就是上面讨论的上下界是否要累加的原因),同样的在做第二遍最大流前不删除汇点到源点的这条边,也并没有大的影响,因为这条边必然会被增广完,

也就是最后统计最大流时多了一个 inf

方法二：二分汇点连向源点这条边的下界,然后判断无源汇上下界网络流是否存在可行流,因为若二分下界小于等于最大流等价于存在可行流

4.有源汇的上下界最小流

方法一：与最大流类似,先判断可行流是否存在。若存在,则以原图的汇点作为源点,以原图的源点作为汇点,边不改变,做一遍反向最大流,实际对原图来说是在不断地减少可行流,当反向达到最大,则正向刚好是最小流。通过累加原图的源点连出去的边的反向边的流量得到最小流,同样注意如果删掉附加源和附加汇以及它们的所有连边,那么最小流还需要加上源点连出去边的下界,并且由于是方向增广,因此汇点连向源点的边增广完全,因此对源点来说这条边的反向边刚好为 0,所以不需要像最大流一样减去 inf,而每条边的实际流量还是和上面的计算方法一样,均为反向边的流量加上该边的下界。

方法二：二分汇点连向源点这条边的上界,然后判断无源汇上下界网络流是否存在可行流,因为若二分上界大于等于最小流等价于存在可行流

方法三：和判断有源汇上下界可行流一样建图,但是需要分两个阶段,第一阶段是不需要加汇点连向源点的边(下界为 0,上界为 inf),然后跑一遍附加源到附加汇的最大流;第二阶段是加上汇点连向源点的边,然后再跑一遍附加源到附加汇的最大流,如果满流,那么最小流刚好为汇点连向源点的这条边的流量,否则不存在可行流。可以这么理解,只

要最后补齐的图一样,分阶段增广与一步增广最后达到的效果是一样的,因此判断是否存在可行流依然成立。我们可以考虑下为什么要加汇点到源点的这条边,本质是因为源汇不满足流量平衡的条件,因此加入附加源和附加汇建立的图除源汇点外,在增广自由流后得到的伪流和下界叠加后刚好得到可行流,但这样的前提是强制考虑自由流可以流动,实际因为汇点并不平衡,因此这些被强制考虑流动的自由流应囤积在汇点,然而真正在增广的图的汇点是附加汇,因此不会有流囤积在原图的汇点,也就是自由流并不会流动,那么我们强制在汇点到源点连一条下界为 0,上界为 inf 的边,强制让所谓的囤积在汇点的自由流全部流向源点,并且这样做刚好使得源汇也流量平衡,成为无源汇网络。那么要得到最小流,我们的想法是尽量使得囤积在汇点的流量减少,因此,第一阶段不加这条边,是想借用网络本身的结构让附加源的流量尽量流向附加汇,这种情况会出现是原图中存在回路(也就是电路当中的短路情况),这样便能尽量增广一部分,而剩余的一部分只能通过第二阶段来增广了,于是如果原图中不存在短路情况,那么可以省去第一阶段,直接进行第二阶段就可以得到最小流了

1.6.9 费用流

```
/***最大费用最大流***/
const int inf=0x3f3f3f3f;
struct edge{
    int from,to,cap,w,next;
}p[Maxn*Maxn*2];
int head[Maxn];
int tot;
int src,t; //源汇
void addedge(int u,int v,int c,int w){
```

```
    p[tot].from=u;
    p[tot].to=v;
    p[tot].cap=c;
    p[tot].w=w;
    p[tot].next=head[u];
    head[u]=tot++;
}
void insert(int u,int v,int c,int w){
    addedge(u,v,c,w);
    addedge(v,u,0,-w); //反向边费用为负
}
int dist[Maxn],q[Maxn],vis[Maxn],from[Maxn];
bool spfa(){
    memset(dist,-1,sizeof dist);
    memset(vis,0,sizeof vis);
    int s=0,e;
    q[e=1]=src;
    dist[src]=0;
    from[src]=-1;
    while(s!=e){
        int u=q[s=(s+1)%Maxn];
        vis[u]=0;
        for(int i=head[u];i!=-1;i=p[i].next){
            int v=p[i].to,w=p[i].w;
            if(p[i].cap>0&&dist[u]+w>dist[v]){
                dist[v]=dist[u]+w;
                from[v]=i;
                if(!vis[v]){
                    vis[v]=1;
                    q[e=(e+1)%Maxn]=v;
                }
            }
        }
    }
    if(dist[t]==-1) return false;
    return true;
}
int augment(){
    int alpha=inf,ans=0;
    for(int i=from[t];i!=-1;i=from[p[i].from])
        alpha=min(alpha,p[i].cap); //可改进量
    for(int i=from[t];i!=-1;i=from[p[i].from]){ //增广
        p[i].cap-=alpha;
        p[i^1].cap+=alpha;
```

```
        ans+=alpha*p[i].w;
    }
    return ans;
}
int maxcf(){ //最大费用流
    int ans=0;
    while(spfa())
        ans+=augment();
    return ans;
}
```

1.7 匹配

1.7.1 基本概念

*/**
*支配与支配集:设无向图 G(V,E),V*为 V 的子集,对于∀v ∈ (V − V*),∃u ∈ V*,使得(u,v) ∈ E,称 u 支配 v, 并称V*为 G 的一个点支配集(简称支配集)*
*极小支配集:若支配集V*的任何真子集都不是支配集,称V*为极小支配集*
最小支配集:顶点数最少的支配集
点支配数(γ_0):最小支配集中的顶点数
*性质:连通图中V*为极小支配集,V − V* 也为一个支配集*

*点覆盖集:设无向图 G(V,E),V*为 V 的子集,若对于∀e ∈ E,∃v ∈ V*,使得 v 与 e 相关联,则称 v 覆盖 e,并称V*为 G 的一个点覆盖集(简称点覆盖)*
*极小点覆盖:若点覆盖V*的任何真子集都不是点覆盖,称V*为极小点覆盖*
最小点覆盖:顶点数最少的点覆盖
点覆盖数(α_0):最小点覆盖中的顶点数

*点独立集:设无向图 G(V,E),V*为 V 的子集,若V*中任何两个顶点均不相邻,则称V*为 G 的点独立集(简称独立集)*
*极大点独立集:若在V*中加入任何顶点都不再是独立集,则称V*为极大点独立集*
最大点独立集:顶点数最多的点独立集
点独立数(β_0):最大点独立集的顶点数

*团:设无向图 G(V,E),V*为 V 的子集,且V*是完全图*
*极大团:若团N*的任何真子集都不是团,称V*为极大团*
最大团:顶点数最多的团
团数(v_0):最大团的顶点数

*覆盖与边覆盖集:设无向图 G(V,E),E*为 E 的子集, 若对于∀v ∈ V,∃e ∈ E*,使得 v 与 e 相关联,则称 e 覆盖 v,并称E*为边覆盖集(简称边覆盖)*

极小边覆盖:若边覆盖E^*的任何真子集都不是边覆盖,则称E^*是极小边覆盖
最小边覆盖:边数最少的边覆盖集
边覆盖数(α_1):最小边覆盖所含的边数

边独立集(匹配):设无向图 G(V,E),E^*为 E 的子集,若E^*中任何两条边均不相邻,则称E^*为 G 的边独立集(匹配)【任何两条边不相邻就是任何两条边没有公共顶点】
极大匹配:若在E^*中加入任意一条边所得到的集合都不是匹配,则称E^*为极大匹配
最大匹配:边数最多的匹配
边独立数(匹配数)(β_1):最大匹配的边数

盖点与未盖点:设 v 是图 G 的一个顶点,如果 v 与匹配 M 中的某条边关联,则称 v 是 M 的盖点(M 饱和点);如果 v 不与任意一条属于匹配 M 的边相关联,则称 v 是匹配 M 的未盖点(非 M 饱和点)

定理 1:在无向连通图中,极大点独立集一定是极小点支配集;反之不成立.
证明:首先该极大点独立集是一个点支配集,假设另外一个集合中存在一个顶点不受该极大点独立集支配,那么该点可以加入该极大点独立集,与极大点独立集的定义矛盾;
极小性证明:假设不是极小的,根据定义在该极大点独立集存在一个真子集仍是点支配集,也就是除真子集外的顶点与这些真子集中的某些顶点相关联,这与独立集的定义相矛盾;
而可以举出反例,极小点支配集中可以有顶点相邻,即不是点独立集,更不用说极大点独立集了

定理 2:如果一个独立集是极大独立集,当且仅当它是一个支配集. $\rightarrow\{\gamma_0 \le \beta_0\}$
这里已经假设是一个独立集,因此可以反推.也就是既是独立集又是支配集,可以推出既是极大独立集又是极小支配集.

定理 3:在无向连通图中,点覆盖集是点支配集,反之不成立;$\rightarrow\{\gamma_0 \le \alpha_0\}$
扩展:在任意无向图中,点覆盖集(包含所有孤立点)是点支配集

定理 4:在无向连通图中,V^*是点覆盖,$V-V^*$是独立集
推论: 在无向连通图中,V^*是极小(最小)点覆盖,$V-V^*$是极大(最大)独立集
$\rightarrow\{\alpha_0+\beta_0=n, n$ 为顶点数$\}$

定理 5:在无向图 G 中, V^*是 G 的团,当且仅当V^*是 G 的补图的独立集
推论: 在无向图 G 中, V^*是 G 的极大(最大)团,当且仅当V^*是 G 的补图的极大(最大)独立集$\rightarrow\{v_0=\beta_0\}$

定理 6:设无向图 G 的顶点个数为 n,且 G 中无孤立点
(1) 设 M 为 G 的一个最大匹配,对于 G 中 M 的每个未盖点 v,选取一条与 v 关联的边所组成的集合为 N,则 W=M∪N 为 G 中的最小边覆盖
(2) 设 W 为 G 的最小边覆盖,若 G 中存在相邻的边就移去其中的一条,设移去的边

集为 N, 则 M=W-N 为 G 中一个最大匹配
(3) G 中边覆盖数α_1与匹配数β_1, 满足$\alpha_1 + \beta_1 = n$

其他性质:
支配集要包含所有的孤立点
极大独立集包含所有的孤立点
极小点覆盖不包含任何孤立点

公式:
$\alpha_0 + \beta_0 = n$ *无孤立点*
$\gamma_0 \leq \beta_0$
$\gamma_0 \leq \alpha_0$ *无孤立点*
$v_0 = \beta_0$
$\alpha_1 + \beta_1 = n$ *无孤立点*

二分图特殊性质:
最小点覆盖=最大匹配
最小点覆盖+最大独立集=|V|
最小边覆盖+最大匹配=|V|
由上面三式推出:
最小边覆盖=最大独立集=|V|-最大匹配

构造算法:首先使用匈牙利算法求得一个最大匹配, 以 X 部的每个未盖点作为起点, 试图寻求增广路(显然无法找到增广路), 将寻找过程的所有点都做上标记, 所有 X 部未做标记的点和 Y 部做好标记的点组成了最小点覆盖集

有向无环图中的最小路径覆盖:用最少的不相交的路径覆盖所有的点
将所有的点拆成两个点, V 拆成 Vx 和 Vy, 原图中有边 A->B, 新图中添加边 Ax->By, 这样就得到一个二分图, 并有
最小路径覆盖数=原图的节点数-新图的最大匹配

证明:一开始每个点都是一条独立的路径, 共有|V|条路径, 二分图中加一条边, 原图合并两条路径. 并且由于是匹配, X 部的每个点最多连出去一条边到 Y 部, 对应原图这些点的出度至多为 1, 同理 Y 部的每个点最多来自 X 部一条边, 对应原图每个点至多入度为 1. 但是如果原图是有向图的环, 那么终点连向起点, 可使起点也有入度, 终点也有出度, 对应新图中匹配数会增加, 导致结果出错, 因此该定理只适用有向无环图
**/*

1.7.2 最大团

```
/*****最大团*****/
int adj[Maxn][Maxn];
```

```
int cq[Maxn]; //当前最大团序列,下标从 0 开始
int clique[Maxn]; //最终最大团序列,下标从 0 开始
int cnt[Maxn]; //cnt[i]表示点[i,n]组成的子图的最大团
int ans; //最大团数
int n; //顶点数,下标范围[1,n]
bool dfs(int u,int tot){
    if(tot>ans){ //每次增加 1 个点,最大团数最多增加 1
        ans=tot;
        //需要序列
        for(int i=0;i<ans;i++)
            clique[i]=cq[i];
        return true;
    }
    for(int v=u+1;v<=n;v++){
        //就算取[v,n]所有的点也不可能超过最优解
        if(tot+cnt[v]<=ans) return false;
        if(adj[u][v]){ //取点 v
            bool flag=true;
            for(int w=0;w<tot;w++)
                if(!adj[cq[w]][v]){ //点 v 与已有最大团不相容
                    flag=false;
                    break;
                }
            if(flag){
                cq[tot]=v;
                if(dfs(v,tot+1)) return true;
            }
        }
    }
    return false;
}
void solve(){
    ans=0;
    for(int i=n;i>0;i--){
        cq[0]=i; //选择点 i
        dfs(i,1);
        cnt[i]=ans; //更新 cnt
    }
}
```

1.7.3 二分图(匈牙利 DFS 版)

```
/*****匈牙利算法(DFS)*****/
```

```
/*****复杂度 O(n^3)*****/
int adj[Maxn][Maxn]; //邻接矩阵
int match[Maxn]; //match[i]表示 y 组 i 与 x 组 match[i]匹配
int vis[Maxn];
int x,y; //两部分点的总数
int deep; //优化 vis 数组
bool dfs(int u){
    for(int v=1;v<=y;v++){ //扫描 y 组,下标[1,y]
        if(adj[u][v]&&vis[v]!=deep){ //相邻且没有访问
            vis[v]=deep;
            if(match[v]==-1||dfs(match[v])){ //可以找到增广路
                match[v]=u;
                return true;
            }
        }
    }
    return false;
}
int hungary(){ //匈牙利算法
    memset(match,-1,sizeof match);
    memset(vis,-1,sizeof vis);
    int ans=0;
    for(int i=1;i<=x;i++){ //扫描 x 组,[1,x]
        deep=i;
        if(dfs(i)) ans++;
    }
    return ans;
}
```

1.7.4 二分图(匈牙利 BFS 版)

```
/*****匈牙利算法(BFS)*****/
/*****复杂度 O(n^3)*****/
int adj[Maxn][Maxn]; //邻接矩阵
int mx[Maxn]; //X 匹配 Y
int my[Maxn]; //Y 匹配 X
int vis[Maxn];
int q[Maxn]; //BFS 队列
int pre[Maxn]; //记录 Y 指向的 X
int x,y; //两部分点的总数
bool bfs(int deep){
    int s=0,e=-1;
    q[++e]=deep;
    while(s<=e){
```

```
            int u=q[s++];
            for(int v=1;v<=y;v++){
                if(adj[u][v]&&vis[v]!=deep){
                    vis[v]=deep;
                    if(my[v]==-1){ //找到增广路
                        while(1){  //回溯修改
                            int t=mx[u]; //临时记录
                            mx[u]=v;
                            my[v]=u;
                            if(t==-1) break;
                            else u=pre[v=t];
                        }
                        return true;
                    }
                    else{
                        q[++e]=my[v];
                        pre[v]=u;
                    }
                }
            }
        }
        return false;
}
int hungary(){ //匈牙利算法
    memset(mx,-1,sizeof mx);
    memset(my,-1,sizeof my);
    memset(vis,-1,sizeof vis);
    int ans=0;
    for(int i=1;i<=x;i++) //扫描 x 组,[1,x]
        if(bfs(i)) ans++;
    return ans;
}
```

1.7.5 二分图(Hopcroft_Karp)

```
/*****Hopcroft_Karp*****/
/*****复杂度 O(n^2.5)*****/
const int inf=0x3f3f3f3f;
int adj[Maxn][Maxn]; //邻接矩阵
int dx[Maxn],dy[Maxn]; //X/Y 的层次
int mx[Maxn],my[Maxn]; //X/Y 匹配 Y/X
int vis[Maxn];
int q[Maxn]; //BFS 队列
int x,y; //两部分点的总数
```

```
int deep; //优化 vis 数组
bool bfs(){ //是否存在增广路
    int len=inf; //最短增广路长度
    memset(dx,-1,sizeof dx);
    memset(dy,-1,sizeof dy);
    int s=0,e=-1;
    for(int i=1;i<=x;i++) //扫描 X 组
        if(mx[i]==-1){ //所有未盖点入队
            q[++e]=i;
            dx[i]=0; //距离初始化为 0
        }
    while(s<=e){
        int u=q[s++];
        if(dx[u]==len) break; //这以后不可能是最短增广路
        for(int v=1;v<=y;v++){ //扫描 Y 组
            if(adj[u][v]&&dy[v]==-1){ //还未更新距离的邻接点
                dy[v]=dx[u]+1; //更新距离
                if(my[v]==-1) //找到最短增广路
                    len=dy[v]; //更新最短长度
                else{ //从 my[v]继续往下增广
                    dx[my[v]]=dy[v]+1;
                    q[++e]=my[v];
                }
            }
        }
    }
    return len!=inf;
}
bool dfs(int u){
    for(int v=1;v<=y;v++){ //扫描 Y 组
        if(adj[u][v]&&vis[v]!=deep&&dy[v]==dx[u]+1){
            vis[v]=deep;
            if(my[v]==-1||dfs(my[v])){
                my[v]=u,mx[u]=v;
                return true;
            }
        }
    }
    return false;
}
int Hopcroft_Karp(){
    int ans=0;
    memset(mx,-1,sizeof mx);
    memset(my,-1,sizeof my);
```

```
    memset(vis,0,sizeof vis);
    deep=0;
    while(bfs()){
        for(int i=1;i<=x;i++)
            if(mx[i]==-1){ //未盖点
                deep++;
                if(dfs(i)) ans++;
            }
    }
    return ans;
}
```

1.7.6 完全二分图的最大匹配(KM)

```
/***Kuhn-Munkres***/
/***复杂度 O(n^3)***/
const int INF=-1000000; //保证比最小边小
const int inf=0x3f3f3f3f;
int adj[Maxn][Maxn]; //邻接矩阵
int match[Maxn]; //match[i]表示 X 组 i 与 Y 组 match[i]匹配
int lx[Maxn],ly[Maxn]; //X 和 Y 部的标号,辅助作用
int vx[Maxn],vy[Maxn]; //X 和 Y 部的点是否被访问
int mn[Maxn]; //mn[i]表示 y 组 i 与交错路上的 X 组点边权最小值
int x,y; //X 和 Y 部的点数
int dep; //优化 vx 和 vy 数组
bool dfs(int u){
    vx[u]=dep;
    for(int v=1;v<=y;v++){ //扫描 y 组,下标[1,y]
        if(lx[u]+ly[v]>adj[u][v]) //不在相等子图的边
            mn[v]=min(mn[v],lx[u]+ly[v]-adj[u][v]);
        else if(vy[v]!=dep){ //可行边且没有访问
            vy[v]=dep;
            if(match[v]==-1||dfs(match[v])){ //可以找到增广路
                match[v]=u;
                return true;
            }
        }
    }
    return false;
}
int KM(){ //保证 x<=y
    dep=0;
    memset(match,-1,sizeof match);
```

```
    memset(vx,0,sizeof vx);
    memset(vy,0,sizeof vy);
    for(int i=1;i<=x;i++){
        lx[i]=-inf;
        for(int j=1;j<=y;j++)
            lx[i]=max(lx[i],adj[i][j]);
    }
    for(int i=1;i<=y;i++) ly[i]=0;
    for(int i=1;i<=x;i++){ //扫描 x 组,[1,x]
        for(int j=1;j<=y;j++) mn[j]=inf;
        while(true){ //不存在完备匹配会死循环
            dep++;
            //for(int j=1;j<=x;j++) mn[j]=inf;
            if(dfs(i)) break; //成功找到增广路
            int d=inf;
            for(int j=1;j<=y;j++)
                if(vy[j]!=dep) d=min(d,mn[j]);
            //更新在交错路上的点的标号
            for(int j=1;j<=x;j++)
                if(vx[j]==dep) lx[j]-=d;
            for(int j=1;j<=y;j++)
                if(vy[j]==dep) ly[j]+=d;
                else mn[j]-=d;
        }
    }
    int ans=0;
    //x=y 且是完全二分图的简单写法
    //for(int i=1;i<=x;i++) ans+=lx[i];
    //for(int i=1;i<=y;i++) ans+=ly[i];
    for(int i=1;i<=y;i++){
        if(match[i]!=-1){
            if(adj[match[i]][i]!=INF)
                ans+=adj[match[i]][i];
            else //不存在完备匹配
                return ans=-inf;
        }
    }
    return ans;
}
void init(int n,int m){
    //adj 的值回赋值给 lx,可能在后面会减小
    //INF 不能太小,以防爆 int
    x=n,y=m; //不要忘记给 x,y 赋值,x<=y
    for(int i=1;i<=x;i++)
```

```
        for(int j=1;j<=y;j++)
            adj[i][j]=INF;
}
```

1.8 平面图

1.8.1 基本概念

```
/*
平面图:可以画成所有边只相交于顶点的无向图,画出这样的图叫做平面嵌入
K4(4 阶完全图),K1,n(完全二部图[1,n]), K2,n(完全二部图[2,n])是平面图
K5(5 阶完全图), K3,3(完全二部图[3,3])是非平面图
区域(面):平面图(已经是平面嵌入),由一条或多条边界所界定的范围内不含顶点和边的范围
外部区域:无界的区域
内部区域:有界的区域
边界:顶点和边都与某个区域关联的子图,边界是回路(但不一定是圈,例如外部区域的边界)
区域的度数:区域边界的边的个数
极大平面图:若在平面图的任意不相邻顶点之间加一条边都变成了非平面图
极小非平面图:若在非平面图中任意删除一条边都变成了平面图
对偶图:在平面图(已经是平面嵌入)的每个区域内画对应的一个点,若原平面图区域 i 与区域 j 有公共边,则连接这两个区域对应的点,若原平面图有桥(可看成外部区域的公共边),则过外部区域对应的点画一个圈包围桥的一端(叶子)

性质:
平面图的对偶图是平面图(在构造时就已经是平面嵌入)
平面图的对偶图是连通图
原图为环,对偶图为桥;原图为桥,对偶图为环
多数情况,对偶图含有较多平行边

定理 1:平面图的任何子图都是平面图
定理 2:平面图加平行边和自环后还是平面图
定理 3:平面图所有区域的度数之和等于边数的 2 倍(每条边作为两个区域的边界,桥作为外部区域的边界也来回计算两次)
定理 4:设 G 为 n(n>=3)阶简单连通的平面图,G 为极大平面图当且仅当 G 的每个区域度数为 3
定理 5:设G* 是连通平面图 G 的对偶图, n*,m*,r* 和 n,m,r 分别为G* 和 G 的顶点数、边数和面数,则有:
(1) m* = m (原图每条边与对偶图对应一条边相交)
(2) r* = n (一个面对应一个点)
(3) n* = r (结合上面两个式子以及欧拉公式推得)
```

(4) 设顶点v_i^*位于 G 的区域R_i, 则$\deg(v_i^*) = deg(R_i)$, 即对偶图中顶点v_i^*的度数等于平面图中区域R_i的度数

定理 6:如果 G 是一个阶 n>=3, 边数为 m 的平面图, 则 m<=3n-6

推论 1:如果 G 是一个阶 n>=3, 边数为 m>3n-6 的图, 则图 G 是非平面图

推论 2:每个平面图都含有一个小于等于 5 的顶点

推论 3:5 阶完全图是非平面图

2 度顶点内同构:给定两个图 G1 和 G2, 如果它们是同构的, 或者可以通过反复插入和(或)去掉度数为 2 的顶点后, 使得 G1 和 G2 同构, 则称 G1 和 G2 是在 2 度顶点内同构的

库拉托夫斯基(Kuratowski)定理:一个图是平面图, 当且仅当它不含有与 $K_{3,3}$ 和 K_5 在 2 度结点内同构的子图

收缩:设 e 是图 G 的一条边, 从 G 中删去 e 并将 e 的两个顶点合并, 删除由此得到的环边和平行边, 这个过程称为 e 的收缩. 若图 G1 课通过一系列边的收缩得到与图 G2 同构的图, 则称 G1 可以收缩到 G2. 注意, 收缩边 e 并不要求 e 的两个顶点度数为 2

Wagner 定理:图 G 是平面图, 当且仅当 G 中既没有可收缩到 K_5 的子图, 也没有可以收缩到 $K_{3,3}$ 的子图

欧拉公式:如果 G 是一个顶点数为 n, 边数为 m, 面数为 r 的连通平面图, 则有恒等式:n-m+r=2

推广:对于有 k(k>=2)个连通分支的平面图, 有恒等式:n-m+r=k+1

*/

1.9 图的着色问题

1.9.1 基本概念

/*

顶点着色:给图 G(不存在自环)的每个顶点指定一种颜色, 使得任何两个相邻的顶点颜色均不同

色数(X(G)):如果能用 k 种颜色对图 G 进行顶点着色, 就称对图 G 进行了 k 着色, 也称 G 是 k-可着色的. 若 G 是 k-可着色的, 但不是(k-1)-可着色的, 称 G 是 k 色的图, 并称这样的 k 为图 G 的色数

定理 1:X(G)=1 当且仅当 G 是零图(没有边的图)

定理 2:n 阶完全图的色数为 n

定理 3:奇圈的色数为 3

定理 4:图 G 的色数为 2 当且仅当 G 是一个非空的二部图

定理 5:对任意的图 G(不存在自环), 均有 X(G)<=Δ(G)+1, Δ(G)为顶点的最大度

布鲁斯(Brooks)定理:设连通图 G 不是完全图 K_n, 也不是奇圈, 则 X(G)<=Δ(G)

边着色：给图 G 的每条边指定一种颜色，使得任何两条相邻的边颜色均不同
边色数($X_1(G)$)：如果能用 k 种颜色对图 G 进行边着色，就称对图 G 进行了 k 边着色，也称 G 是 k-边可着色的. 若 G 是 k-边可着色的，但不是(k-1)-边可着色的，称 G 是 k 边色的图，并称这样的 k 为图 G 的边色数

维津(Vizing)定理：对任何一个非空简单图 G，都有
$X_1(G)=\Delta(G)$ 或 $X_1(G)=\Delta(G)+1$
定理 1：设图 G 为长度大于或等于 2 的偶圈，则 $X_1(G)=\Delta(G)=2$；设图 G 为长度大于或等于 3 的奇圈，则 $X_1(G)=\Delta(G)+1=3$
定理 2：对二部图，有 $X_1(G)=\Delta(G)$

平面图着色：对平面图 G 来说，它将平面分成若干区域(不包含外部区域)，对每个区域着色，使得有公共边的区域颜色均不同
面色数($X^*(G)$)：如果能用 k 种颜色给平面图 G 进行面着色，则称 G 是 k-面可着色的，在进行面着色时，所用最少颜色数称为平面图的面色数

定理 1：平面图 G 是 k-面可着色的，当且仅当它的对偶图 G^* 是 k 色的图
五色定理：连通简单图 G 的色数不超过 5(4 色猜想的证明不被大部分数学家接受)
**/*

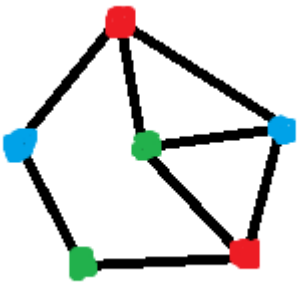
1.9.2 顺序染色(不一定是最优解)

```
//注意:不一定是最优解
/*****连通图染色(邻接矩阵)*****/
/*****复杂度 O(n^2)*****/
int adj[Maxn][Maxn];
int vis[Maxn]; //每趟染色判断颜色是否使用过
int color[Maxn]; //color[i]表示点 i 染的颜色
int dye(int n){ //染色范围[0,n-1]
    int ans=-1; //需要染色最大标号,从 0 开始
    memset(color,-1,sizeof color);
    for(int i=0;i<n;i++){
        memset(vis,0,sizeof vis);
        for(int j=0;j<n;j++) //标记点 i 的邻接点染过的颜色
            if(adj[i][j]&&color[j]!=-1)
                vis[color[j]]=1;
        for(int j=0;j<n;j++)
            if(!vis[j]){ //取[0,n-1]中最小的没有染过的颜色
                color[i]=j; //给点 i 染色
                ans=max(ans,j); //更新答案
                break;
            }
    }
    return ans+1;
```

```
}

/*****平面图染色(邻接矩阵)*****/
/*****复杂度 O(n^2)*****/
//说明:平面图的对偶图是平面图,平面图 G 是 k-面可着色的,当且仅当它的对偶图 G* 是 k 色的图,因此求平面图的点着色可转换成面着色,而面着色有四色猜想(不叫定理是因为还有一部分数学家不承认这种证明方法)作保证,因此可利用这四色猜想剪枝
int adj[Maxn][Maxn];
int vis[Maxn]; //每趟染色判断颜色是否使用过
int color[Maxn]; //color[i]表示点 i 染的颜色
int dye(int n){ //染色范围[0,3]
    int ans=-1; //需要染色最大标号,从 0 开始
    memset(color,-1,sizeof color);
    for(int i=0;i<n;i++){
        memset(vis,0,sizeof vis);
        for(int j=0;j<n;j++) //标记点 i 的邻接点染过的颜色
            if(adj[i][j]&&color[j]!=-1)
                vis[color[j]]=1;
        for(int j=0;j<4;j++)
            if(!vis[j]){ //取[0,3]中最小的没有染过的颜色
                color[i]=j; //给点 i 染色
                if(j==3) return 4; //四色定理剪枝
                ans=max(ans,j); //更新答案
                break;
            }
    }
    return ans+1;
}

//染色的贪心思想是错误的,得到的不一定是最优解
```



```
//上图若用贪心思想,得到的最少颜色数将会是 4,而正确的应该是 3
```

1.9.3 平面图暴力染色(最优解)

```
/*****平面图染色(邻接矩阵)*****/
/*****复杂度 O(3^n)*****/
/***利用四色定理暴力染色***/
int adj[Maxn][Maxn];
int color[Maxn]; //color[i]表示点 i 染的颜色
int n; //顶点数
int ans; //最少颜色数
bool check(int u,int c){ //判断 u 是否可以染成颜色 c
    for(int v=0;v<u;v++)
        if(adj[u][v]&&color[v]==c) //冲突
            return false;
    return true;
}
void dfs(int d){ //最多只染 3 种颜色
    if(d==n){ //搜到最底层,找到方案
        //必定存在一种连续的 1,2..的染色方案
        //当 color 中的最大元素最小时,染色连续
        ans=min(ans,*max_element(color,color+n));
        return;
    }
    for(int c=1;c<4;c++)
        if(check(d,c)){
            color[d]=c;
            dfs(d+1);
        }
}
//在主函数里加 memset(adj,0,sizeof adj)
//调用 solve 前构造好邻接矩阵
void solve(){
    color[0]=1; //点 0 染成颜色 1
    ans=4; //必须这么初始化
    dfs(1); //从点 1 开始染色
}
```

1.9.4 二分图判定

```
/*****连通图的二分图判定*****/
/*****复杂度 O(E)*****/
struct edge{
    int to,next,w;
}p[Maxn*Maxn];
```

```
int tot;
int head[Maxn];
int c[Maxn]; //染色 1,2
void addedge(int a,int b,int c){
    p[tot].w=c;
    p[tot].to=b;
    p[tot].next=head[a];
    head[a]=tot++;
}
memset(c,0,sizeof c); //不要忘记初始化
c[1]=1; //染色第一个点要在 dfs 外初始化
bool dfs(int u){ //dfs 黑白染色
    for(int i=head[u];i!=-1;i=p[i].next){
        int v=p[i].to;
        if(c[v]==c[u]) return false;
        if(!c[v]){
            c[v]=3-c[u];
            if(!dfs(v)) return false;
        }
    }
    return true;
}
```

第二章 数据结构

2.1 分块

2.1.1 块状数组

```
/*****块状数组*****/
/*****单次操作复杂度 O(√n)*****/
int a[Maxn]; //存元素
int b[Maxn]; //b[x]表示 a[x]...block(a[x])末端的最大值
int sz=10; //每块大小
int n; //元素总数[0,n-1]
void maintain(int x){ //维护 x 所在的块
    int lb=x/sz*sz,rb=min(n,lb+sz)-1;  //[lb,rb]
    if(x==rb) {b[x]=a[x];x--;} //防止 b[x+1]下标溢出
    while(x>=lb) {b[x]=max(a[x],b[x+1]);x--;}
```

```
}
void init(){  //初始化
    for(int i=sz-1;i<n;i+=sz)
        maintain(i);
}
void modify(int x,int y){  //修改 a[x]=y
    a[x]=y;
    maintain(x);
}
int query(int x,int y){  //查询区间[x,y]的最大值
    int x_sz=x/sz,y_sz=y/sz,ret;
    if(x_sz==y_sz){  //同一块
        ret=a[x];
        for(int i=x+1;i<=y;i++)
            ret=max(ret,a[i]);
    }
    else{  //不同块
        ret=b[x];
        for(int i=x_sz+1;i<y_sz;i++)  //按块计算
            ret=max(ret,b[i*sz]);
        for(int i=y_sz*sz;i<=y;i++)  //按个数计算
            ret=max(ret,a[i]);
    }
    return ret;
}

/***分块区间更新、查询***/
/***复杂度 sqrt(n)***/
//块 x 对应的区间[x*sz,(x+1)*sz-1]
//块内处理需要及时维护块内性质
int a[Maxn]; //原数组
int b[Maxn]; //记录信息的数组
int tag[Maxn]; //标记数组
int pos[Maxn]; //pos[i]表示 i 所在的块号
int sz; //块大小
int n; //元素总数,[0...n-1]
int m; //总块数,[0...m-1]
void maintain(int x){ //维护块 x 的性质
    int l=x*sz,r=min(n,l+sz); //[l,r)
    for(int i=l;i<r;i++) b[i]=a[i];
    sort(b+l,b+r);
}
void init(int n){
    sz=(int)sqrt(n); //块大小为 sqrt(n)
```

```
    m=(n+sz-1)/sz; //n/sz 向上取整
    for(int i=0;i<n;i++) pos[i]=i/sz;
    //a,b,tag 数组初始化
    for(int i=0;i<m;i++) tag[i]=0;
    for(int i=0;i<m;i++) maintain(i);
}
void update(int x,int y,int v){ //区间[x,y]加上 v
    if(pos[x]==pos[y]){ //同一块,暴力
        for(int i=x;i<=y;i++) a[i]+=v;
        maintain(pos[x]);
    }
    else{
        //x,y 所在块暴力
        for(int i=x;i<(pos[x]+1)*sz;i++) a[i]+=v;
        maintain(pos[x]);
        for(int i=pos[y]*sz;i<=y;i++) a[i]+=v;
        maintain(pos[y]);
        //[pos[x]+1,pos[y]-1]这些块按块处理
        for(int i=pos[x]+1;i<pos[y];i++) tag[i]+=v;
    }
}
//区间[x,y]大于等于 v 的个数统计
int query(int x,int y,int v){
    int ans=0;
    if(pos[x]==pos[y]){ //同一块,暴力统计
        for(int i=x;i<=y;i++)
            if(a[i]>=v-tag[pos[x]]) ans++;
    }
    else{
        //x,y 所在块内暴力统计
        for(int i=x;i<(pos[x]+1)*sz;i++)
            if(a[i]>=v-tag[pos[x]]) ans++;
        for(int i=pos[y]*sz;i<=y;i++)
            if(a[i]>=v-tag[pos[y]]) ans++;
        //[pos[x]+1,pos[y]-1]这些块按块统计,二分
        for(int i=pos[x]+1;i<pos[y];i++)
            ans+=b+(i+1)*sz-lower_bound(b+i*sz,b+(i+1)*sz,v-tag[i]);
    }
    return ans;
}
```

2.1.2 块状链表

```
/*****块状链表*****/
/*****单次操作复杂度 O(√n)*****/
#include<cstdio>
#include<iostream>
#include<cstring>
#include<cstdlib>
#define max_size 10
using namespace std;

struct blocklist{
    int num[max_size]; //max_size 为每块元素个数上限,一般取 sqrt(n)
    int sz; //该块元素个数
    int sum; //信息
    blocklist* next;
    blocklist():sz(0),next(0){}
    void refresh(){ //对该块 sum 更新
        sum=0;
        for(int i=0;i<sz;i++)
            sum+=num[i];
    }
}*head=NULL;
//x 必须传引用,使 x 变成该块的个数
blocklist* get_pos(int &x){  //获得第 x 个元素在 ret 块的位置
    blocklist* ret=head;
    while(ret&&x>ret->sz){
        x-=ret->sz;
        ret=ret->next;
    }
    return ret; //返回块地址
}
bool merge(blocklist* pos){  //合并 pos 和 pos->next
    if(!pos||!pos->next) return false;
    if(pos->sz+pos->next->sz<=max_size){

memmove(pos->num+pos->sz,pos->next->num,sizeof(int)*pos->next->sz);
        pos->sz+=pos->next->sz;
        pos->refresh();
        blocklist* u=pos->next;
        pos->next=u->next;
        delete u;
```

```
            return true;
        }
        return false;
    }
    void maintain(){  //维护块状链表数目在 O(sqrt(n))
        blocklist* pos=head;
        while(pos){
            if(!merge(pos))
                pos=pos->next;
        }
    }
    //一般先调用 get_pos,使 x 变成该块的个数
    void split(blocklist* pos,int x){ //分裂成 x-1 和 sz-x+1
        blocklist* tmp=pos->next;
        pos->next=new blocklist();
        pos->next->next=tmp;
        memmove(pos->next->num,pos->num+x-1,sizeof(int)*(pos->sz-x+1));
        //先更新 sz,再 refresh
        pos->next->sz=pos->sz-x+1;pos->next->refresh();
        pos->sz=x-1;pos->refresh();
    }
    //x>=1,若 x 大于块表元素总数,插在表尾
    void insert(int x,int n){  //将 n 个元素插入到 x 所在的位置,x 后挪
        blocklist* pos=get_pos(x);
        if(pos) split(pos,x);
        else{
            pos=new blocklist();
            if(!head) head=pos; //空表插入
            else{ //x 大于块表元素总个数
                blocklist* it; //表尾插入
                for(it=head;it->next;it=it->next);
                it->next=pos;
            }
        }
        for(int i=0;i<n;i++){
            if(pos->sz==max_size){
                blocklist* tmp=pos->next;
                pos->next=new blocklist();
                pos->next->next=tmp;
                pos->refresh();
                pos=pos->next;
            }
            scanf("%d",&pos->num[pos->sz++]); //读入方式
        }
```

```
    pos->refresh();
    maintain(); //维护链表
}
//删除元素须存在,否则出错
void remove(int x,int n){ //删除 x 后(包括 x)的 n 个元素,删除区间[x-1,y-1]
    //删除元素须不存在,删至最后一个
    /*int sum=0,y=x+n-1;
    for(blocklist *it=head;it;it=it->next)
        sum+=it->sz;
    if(x>sum) return;
    y=min(y,sum);
    n=y-x+1;*/
    int y=x+n-1;
    blocklist *x_pos=get_pos(x),*y_pos=get_pos(y);
    if(x_pos==y_pos){ //同一块

memmove(x_pos->num+x-1,y_pos->num+y,sizeof(int)*(y_pos->sz-y));
        x_pos->sz-=n;
        x_pos->refresh();
    }
    else{
        split(x_pos,x);
        split(y_pos,y+1);
        for(blocklist *it=x_pos->next;it!=y_pos;it=it->next)
            delete it;
        x_pos->next=y_pos->next;
        delete y_pos;
    }
    maintain();
}
int get_sum(int x,int n){ //获得 x 后(包括 x) n 个数的和
    int y=x+n-1,ret=0;
    blocklist *x_pos=get_pos(x),*y_pos=get_pos(y);
    if(x_pos==y_pos){ //同一块,计算[x-1,y-1]
        for(int i=x-1;i<y;i++)
            ret+=x_pos->num[i];
    }
    else{ //不同块
        for(int i=x-1;i<x_pos->sz;i++) //x 所在块
            ret+=x_pos->num[i];
        for(blocklist *it=x_pos->next;it!=y_pos;it=it->next)
            ret+=it->sum; //按块计算
        for(int i=0;i<y;i++) //y 所在块
            ret+=y_pos->num[i];
```

```
    }
    return ret;
}
void remove_blocklist(blocklist* head){
    blocklist* it;
    while(head){
        it=head;
        head=head->next;
        delete it;
    }
}
void print(blocklist* head){
    if(!head) printf("Empty!\n");
    else{
        int cnt=0;
        for(blocklist* it=head;it;it=it->next){
            printf("block %d: ",++cnt);
            for(int i=0;i<it->sz;i++)
                printf("%d ",it->num[i]);
            printf("   sum: %d\n",it->sum);
        }
    }
}
```

2.2 位图

```
//bitmap[i]的第 j 位表示 32*i+j
//i=x>>5, j=x&31
int bitmap[Maxn];
void set(int x){
    bitmap[x>>5]|=1<<(x&31);
}
bool get(int x){
    return bitmap[x>>5]&(1<<(x&31));
}
void clear(int x){
    bitmap[x>>5]&=~(1<<(x&31));
}
```

2.3 并查集

```
/*****并查集*****/
/*****复杂度 O(n*α(n))*****/
void makeset(){
    for(int i=0;i<n;i++)
        fa[i]=i,rank[i]=0;
}
int findset(int x){ //递归版
    return x==fa[x]?x:(fa[x]=findset(fa[x]));
}
int findset(int x){ //非递归版
    int i,r=x;
    while(r!=fa[r]) r=fa[r];
    while(x!=r){i=fa[x];fa[x]=r;x=i;}
    return r;
}
void unionset(int x,int y){ //按秩合并
    x=findset(x),y=findset(y);
    if(rank[x]>rank[y]) fa[y]=x;
    else{
        fa[x]=y;
        if(rank[x]==rank[y]&&x!=y)
        rank[y]++;
    }
}

/*****删点并查集*****/
int tmpdfn;
int dfn[Maxn]; //时间戳
int fa[Maxn];
void init(int n){
    for(int i=1;i<=n;i++){
        dfn[i]=fa[i]=i;
    }
    tmpdfn=n;
}
int findset(int x){ //传入时间戳
    return fa[x]==x?x:(fa[x]=findset(fa[x]));
}
void unionset(int x,int y){ //传入时间戳
    fa[findset(x)]=findset(y);
}
```

```
void del(int x){ //删除 x 就是更改 x 的时间戳
    dfn[x]=++tmpdfn;
    fa[tmpdfn]=tmpdfn;
}

#define ls tr[k].l,l,mid
#define rs tr[k].r,mid+1,r
/***可持久化并查集***/
struct PUFD{
    int l,r; //左右儿子下标
    int fa; //父亲
    int dep; //深度
    PUFD(int _fa=0){
        l=r=dep=0,fa=_fa;
    }
}tr[Maxn*50];
int rt[Maxn]; //根
int n; //区间范围[1,n]
int sz; //节点总数
void build(int &k,int l,int r){
    k=++sz;
    if(l==r){
        tr[k]=PUFD(l);
        return;
    }
    int mid=l+r>>1;
    build(ls);
    build(rs);
}
//返回以数组下标 k 为根的子树中 pos 的下标
int query(int k,int l,int r,int pos){
    if(l==r) return k;
    int mid=l+r>>1;
    if(pos<=mid) return query(ls,pos);
    return query(rs,pos);
}
//只需复制一份从根(数组下标为 k)到叶子 u 的一条路径
//修改 tr[t].fa=v
void update(int k,int l,int r,int &t,int u,int v){
    t=++sz; //这里复制节点
    if(l==r){ //找到 u,对应新的节点 t
        tr[t]=PUFD(v); //修改父亲指向 v
        //tr[t].dep=tr[k].dep; //继承深度,但并没快多少
        return;
```

```
    }
    tr[t].l=tr[k].l,tr[t].r=tr[k].r; //连接好左右儿子
    int mid=l+r>>1;
    if(u<=mid) update(ls,tr[t].l,u,v);
    else update(rs,tr[t].r,u,v);

}
//返回以数组下标 k 为根的子树中 pos 的父亲下标
int findset(int k,int pos){
    int p=query(k,1,n,pos);
    if(tr[p].fa==pos) return p;
    int id=findset(k,tr[p].fa);
    //update(k,1,n,k,tr[p].fa,id); //路径压缩,更慢
    return id;
}
//修改以数组下标 k 为根的子树中 pos 对应的深度
void add(int k,int l,int r,int pos){
    if(l==r){ //找到 pos 对应节点,深度+1
        tr[k].dep++;
        return;
    }
    int mid=l+r>>1;
    if(pos<=mid) add(ls,pos);
    else add(rs,pos);
}
void init(int n){ //初始化第 0 次操作,fa[i]=i
    sz=0;
    build(rt[0],1,n);
}
void unionset(int i,int u,int v){ //第 i 次操作
    rt[i]=rt[i-1];
    int p=findset(rt[i],u),q=findset(rt[i],v);
    if(tr[p].fa==tr[q].fa) return;
    if(tr[p].dep>tr[q].dep) swap(p,q); //p 深度小
    update(rt[i-1],1,n,rt[i],tr[p].fa,tr[q].fa);
    if(tr[p].dep==tr[q].dep) add(rt[i],1,n,tr[q].fa);
}
void print(int k){ //中序遍历
    if(!k) return;
    printf("%d %d %d %d %d\n",k,tr[k].l,tr[k].r,tr[k].fa,tr[k].dep);
    print(tr[k].l);
    print(tr[k].r);
}
```

2.4 Dancing Links X 算法

```
/*
    精确覆盖:每一列被选中行覆盖当且仅当一次,
    选中 k 行,若 k 行有元素 i,j,则覆盖 i,j 的其他行均
    不应该被选入,对应 dance<删除覆盖行 row[i]元素的列>
    ps:删除一列时,会断开覆盖这列元素的所有行的其他元素
    在纵方向上的连接,但恢复时必须是断开时的上下相邻元素,由于
    删除时是从上到下,后面可能会删除待恢复元素的上下相邻元素,
    因此要倒着恢复,同时<删除覆盖行 row[i]元素的列>操作是从左到右
    删除列,后面可能会删除待恢复元素的上下相邻元素,因此也要倒着恢复
*/
#define Maxn 345970 //行*列+列(辅助元素)
#define Maxr 970 //行数
#define Maxc 370 //列数
/***Dancing Links X 算法***/
struct DLX{
    int sz; //节点个数
    int st[Maxr],top; //答案存在[0,top-1]
    int row[Maxn],col[Maxn]; //各点行列编号
    int L[Maxn],R[Maxn],U[Maxn],D[Maxn]; //十字链表
    int s[Maxc]; //每列的元素个数
    int fst[Maxr]; //fst[i]=-1 表示该行为空
    void init(int m){ //m 为列数
        memset(fst,-1,sizeof fst);
        memset(s,0,sizeof s);
        for(int i=0;i<=m;i++) //辅助元素双向循环列表
            L[i]=i-1,R[i]=i+1,U[i]=D[i]=i;
        L[0]=m,R[m]=0;
        sz=m;
    }
    void link(int r,int c){ //插入 r 行 c 列为 1 的元素
        ++sz;
        row[sz]=r,col[sz]=c,s[c]++;
        D[sz]=D[c],U[sz]=c; //纵方向首部插入
        U[D[sz]]=D[U[sz]]=sz;
        if(fst[r]==-1) //第一个元素指向自己
            fst[r]=L[sz]=R[sz]=sz;
        else{ //横方向首部插入
            R[sz]=R[fst[r]],L[sz]=fst[r];
            L[R[sz]]=R[L[sz]]=sz;
        }
    }
```

```
/*
    并未删除 c 列,只是在辅助元素将 c 删除,
    同时将覆盖 c 列元素的行元素在纵方向删除
    (横方向并未删除),这样效果等价于删除同时方便恢复
*/
void del(int c){ //删除 c 列
    R[L[c]]=R[c],L[R[c]]=L[c];
    for(int i=D[c];i!=c;i=D[i])
        for(int j=R[i];j!=i;j=R[j])
            D[U[j]]=D[j],U[D[j]]=U[j],s[col[j]]--;
}
/*
    由于要利用被删除元素在纵方向上原来的指针,
    因此需要倒着恢复,这样使
*/
void rec(int c){ //恢复 c 列
    R[L[c]]=L[R[c]]=c;
    for(int i=U[c];i!=c;i=U[i]) //这里需要倒着恢复
        for(int j=R[i];j!=i;j=R[j]) //这里并不需要倒着恢复
            D[U[j]]=U[D[j]]=j,s[col[j]]++;
}
/*
    调用 dance(0),返回 true,表示存在解
    由于双向循环链表插入无序,因此得到的解的行号可能无序,
    因此需要排序才能得到一组行号递增的解
*/
bool dance(int dep){
    if(!R[0]){
        top=dep;
        return true;
    }
    int c=R[0];
    for(int i=R[0];i;i=R[i])
        if(s[i]<s[c]) c=i;
    del(c);
    for(int i=D[c];i!=c;i=D[i]){ //枚举选中行 row[i]
        st[dep]=row[i]; //记录
        for(int j=R[i];j!=i;j=R[j]) //删除覆盖行 row[i]元素的列
            del(col[j]);
        if(dance(dep+1)) return true;
        for(int j=L[i];j!=i;j=L[j]) //这里需要倒着恢复
            rec(col[j]);
    }
    rec(c);
```

```
        return false;
    }
}dlx;
```

2.5 堆

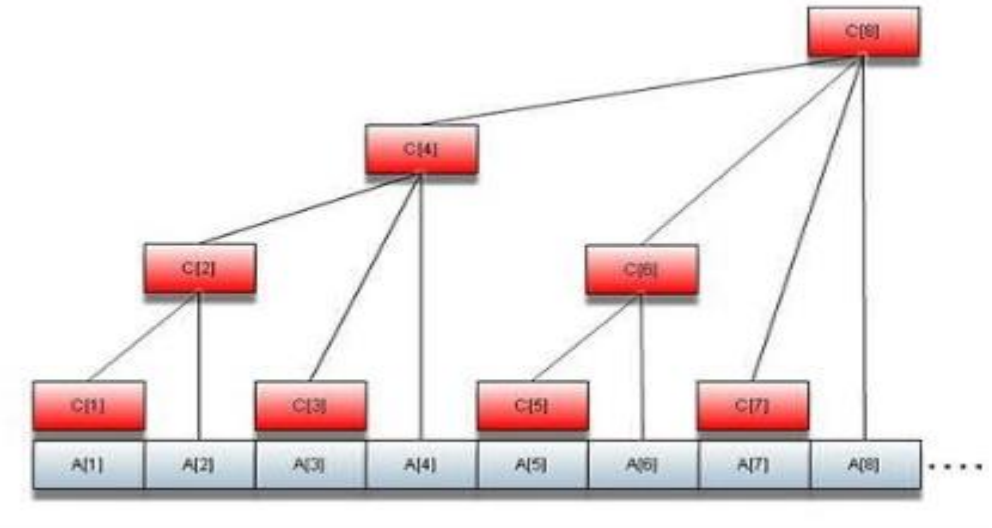
```
/*****堆*****/
/***建堆 O(n),单次访问 O(logn)***/
void swim(int p){  //向上调整
    int q=p>>1,a=heap[p];
    while(q&&a<heap[q]){
        heap[p]=heap[q];p=q;q=p>>1;
    }
    heap[p]=a;
}
void sink(int p){ //向下调整
    int q=p<<1,a=heap[p];
    while(q<=hs){
        if(q<hs&&heap[q]>heap[q+1]) q++;
        if(heap[q]>=a) break;
        heap[p]=heap[q];p=q;q=p<<1;
    }
    heap[p]=a;
}
void insert(int a){ //插入
    heap[++hs]=a;swim(hs);
}
int getmin(){ //取堆顶
    int r=heap[1];heap[1]=heap[hs--];sink(1);
    return r;
}
int decreaseKey(int p,int a){ //a<heap[p]
    heap[p]=a;swim(p);
}
void bulid(){ //数组建堆
    for(int i=hs/2;i>0;i--)
        sink(i);
}

/*****支持删除的堆*****/
struct heap{
    priority_queue<int> h,d; //h 存所有元素,d 存删除的元素
    void add(int x){ //插入 x
```

```
        h.push(x);
    };
    void del(int x){ //删除 x
        d.push(x);
    };
    void fix(){ //将已删除的元素弹出 h
        while(d.size()&&h.top()==d.top())
            h.pop(),d.pop();
    }
    int first(){ //堆顶
        fix();
        return h.top();
    };
    int second(){ //次堆顶
        int x=first();
        h.pop();
        int y=first();
        h.push(x);
        return y;
    }
    void pop(){ //弹出堆顶
        fix();
        h.pop();
    }
    int size(){ //堆的大小
        return h.size()-d.size();
    }
};
```

2.6 树形结构

2.6.1 树状数组




```
/*****树状数组*****/
/*****单次操作 O(log(n)))*****/
//x 从 1 开始
//c[i]表示 a[i-lowbit[i]+1]+...+a[i]
//即 i(包括 i)往前前 lowbit(i)项之和
int n;
int c[Maxn];
int lowbit(int x){
    return x&-x;
}
//例如 x=11010
//x=[11010,11001]+[11000,10001]+[10000+00001]
int sum(int x){
    int res=0;
    while(x>0){
        res+=c[x];
        x-=lowbit(x);
    }
    return res;
}
//x=101000(可用数学归纳法证明)
//后三位(除全 0 外)无论取什么得 x'
//c[x']不会包含 a[x]这一项
//而取 x'=x+lowbit(x),可证 x'包含 a[x]这一项
void update(int x,int d){
```

```
    while(x<=n){
        c[x]+=d;
        x+=lowbit(x);
    }
}

int getx(int x){ //获得a[x]的值
    int res=c[x];
    for(int i=1;i<(x&-x);i<<=1)
        res-=c[x-i];
    return res;
}
int kth(int k){ //第k小数
    int ans=0,cnt=0;
    for(int i=20;i>=0;i--){
        ans+=1<<i;
        if(ans>=n||cnt+c[ans]>=k) ans-=1<<i;
        else cnt+=c[ans];
    }
    return ans+1;
}

/*****二维树状数组*****/
/*****单次操作O(log(n)*log(n))*****/
//(x,y)从(1,1)开始
/*c[x][y]=
a[x][y-lowbit(y)+1]+...+a[x][y]+
a[x-1][y-lowbit(y)+1]+...+a[x-1][y]+
...+
a[x-lowbit[x]+1][y-lowbit(y)+1]+...+a[x-lowbit[x]+1][y]
*/
int n;
int c[Maxn][Maxn];
int lowbit(int x){
    return x&-x;
}
void update(int x,int y,int d){
    for(int i=x;i<=n;i+=lowbit(i))
        for(int j=y;j<=n;j+=lowbit(j))
            c[i][j]+=d;
}
int ask(int x,int y){
    int res=0;
    for(int i=x;i>0;i-=lowbit(i))
```

```
        for(int j=y;j>0;j-=lowbit(j))
            res+=c[i][j];
    return res;
}
```

支持区间修改,单点查询
差分：C[i]=A[i]-A[i-1](i>1),C[1]=A[1],则

$$A[i]=\sum_{j=1}^{i}C[j]$$

将区间[L,R]都加上 x,则执行 update(L,x)和 update(R+1,-x)

支持区间修改和区间查询

$$\sum_{i=1}^{n}A[i]=\sum_{i=1}^{n}\sum_{j=1}^{i}C[j]=\sum_{1\le j\le i\le n}C[j]=\sum_{j=1}^{n}\sum_{i=j}^{n}C[j]=\sum_{j=1}^{n}C[j]*(n+1-j)$$

$$=(n+1)\sum_{j=1}^{n}C[j]-\sum_{j=1}^{n}j*C[j]$$

分别维护 C[i]和 i*C[i],用 C1 和 C2 记录 C[i]和 i*C[i]
将区间[L,R]都加上 x,则执行 update(C1,L,x)和 update(C1,R+1,-x)和 update(C2,L,L*x)和 update(C2,R+1,-(R+1)*x)
查询区间[1,x]的 A[i]的和：

```
ll get(int x){return (x+1)*sum(C1,x)-sum(C2,x);}
```

二维差分：C[i][j]=a[i][j]-a[i][j-1]-a[i-1][j]+a[i-1][j-1],则

$$A[i][j]=\sum_{k=1}^{i}\sum_{l=1}^{j}C[k][l]$$

$$\sum_{i=1}^{n}\sum_{j=1}^{m}A[i][j]=\sum_{i=1}^{n}\sum_{j=1}^{m}\sum_{k=1}^{i}\sum_{l=1}^{j}C[k][l]$$

$$=\sum_{1\le k\le i\le n}\ \sum_{1\le l\le j\le m}C[k][l]$$

$$=\sum_{k=1}^{n}\sum_{l=1}^{m}\sum_{i=k}^{n}\sum_{j=l}^{m}C[k][l]$$

$$=\sum_{k=1}^{n}\sum_{l=1}^{m}C[k][l]*(n+1-k)*(m+1-l)$$

$$=(n+1)(m+1)\sum_{k=1}^{n}\sum_{l=1}^{m}C[k][l]-(m+1)\sum_{k=1}^{n}\sum_{l=1}^{m}kC[k][l]$$

$$-(n+1)\sum_{k=1}^{n}\sum_{l=1}^{m}lC[k][l]+\sum_{k=1}^{n}\sum_{l=1}^{m}klC[k][l]$$

维护四个数状数组 C[i][j],i*C[i][j],j*C[i][j],i*j*C[i][j]
子矩阵(x1,y1)->(x2,y2)增加 d: update(x1,y1,d), update(x1,y2+1,-d), update(x2+1,y1,-d), update(x2+1,y2+1,d)
d 根据维护的数状数组更改
询问子矩阵(x1,y1)->(x2,y2)的
和,sum(x2,y2)-sum(x1-1,y2)-sum(x2,y1-1)+sum(x1-1,y1-1)

```
/*****二维数状数组*****/
/*****单次修改 O((logn)^2)*****/
int n,m;
/*
0->C[i][j]
1->i*C[i][j]
2->j*C[i][j]
3->i*j*C[i][j]
*/
ll c[4][Maxn][Maxn];
void U(int id,int x,int y,int d){
    for(int i=x;i<=n;i+=i&-i)
        for(int j=y;j<=m;j+=j&-j)
            c[id][i][j]+=d;
}
ll sum(int id,int x,int y){
    ll res=0;
    for(int i=x;i>0;i-=i&-i)
```

```
        for(int j=y;j>0;j-=j&-j)
            res+=c[id][i][j];
    return res;
}
//每个数状数组修改 4 个点,总共修改 16 个点
void update(int x1,int y1,int x2,int y2,int d){
    U(0,x1,y1,d);U(0,x1,y2+1,-d); //c[i][j]
    U(0,x2+1,y1,-d);U(0,x2+1,y2+1,d);
    U(1,x1,y1,x1*d);U(1,x1,y2+1,-x1*d); //i*C[i][j]
    U(1,x2+1,y1,-(x2+1)*d);U(1,x2+1,y2+1,(x2+1)*d);
    U(2,x1,y1,y1*d);U(2,x1,y2+1,-(y2+1)*d); //j*C[i][j]
    U(2,x2+1,y1,-y1*d);U(2,x2+1,y2+1,(y2+1)*d);
    U(3,x1,y1,x1*y1*d);U(3,x1,y2+1,-x1*(y2+1)*d); //i*j*C[i][j]
    U(3,x2+1,y1,-(x2+1)*y1*d);U(3,x2+1,y2+1,(x2+1)*(y2+1)*d);
}
ll get(int x,int y){ //(1,1)->(x,y)的子矩阵和
    return (x+1)*(y+1)*sum(0,x,y)-(y+1)*sum(1,x,y)-(x+1)*sum(2,x,y)
    +sum(3,x,y);
}
ll solve(int x1,int y1,int x2,int y2){ //(x1,y1)->(x2,y2)的子矩阵和
    return get(x2,y2)-get(x1-1,y2)-get(x2,y1-1)+get(x1-1,y1-1);
}

/***数状数组区间最值/单点修改***/
/***单次查询/修改 O((logn)^2)***/
/*
c[i]表示[i-lowbit(i)+1,i]的信息,
拆成两部分[i-lowbit(i)+1,i-1]和[i,i]
[i-lowbit(i)+1,i-1]:由所有儿子组成
[i,i]:a[i]
找 i 的儿子:for(int j=1;j<lowbit(i);j<<=1) i-j 为儿子
找 i 的父亲:for(int j=i;j<=n;j+=lowbit(j)) j 为父亲(包括 i)
*/
int n;
int c[Maxn],a[Maxn];
int lowbit(int x){
    return x&-x;
}
void init(){
    for(int i=1;i<=n;i++){ //c[i]是由自身和所有孩子的值更新而来
        c[i]=a[i]; //自身
        for(int j=1;j<lowbit(i);j<<=1)
            c[i]=max(c[i],c[i-j]); //孩子
    }
```

```
}
int query(int l,int r){
    int ans=a[r];
    while(r>=l){
        if(r-lowbit(r)>=l-1){ //往前跳 lowbit
            ans=max(ans,c[r]);
            r-=lowbit(r);
        }
        else
            ans=max(ans,a[r--]); //往前跳 1
    }
    return ans;
}
void update(int x,int d){ //将 a[x]修改为 d
    a[x]=d; //原数组的值修改
    for(int i=x;i<=n;i+=lowbit(i)){ //父亲需要修改
        if(d>=c[i]) {c[i]=d;continue;} //剪枝
        c[i]=a[i]; //自身
        for(int j=1;j<lowbit(i);j<<=1)
            c[i]=max(c[i],c[i-j]); //i-j 为儿子
    }
}
```

2.6.2 线段树

```
/*****线段树*****/
/***单次查询 O(logn)***/
#define ls id<<1,l,mid
#define rs id<<1|1,mid+1,r
#define Maxn 400010
int tr[Maxn];
void pushdown(int id){
    if(tr[id]){ //标记下放给左右孩子
        tr[id<<1]=tr[id<<1|1]=1;
        tr[id]=0;
    }
}
void pushup(int id){ //左右孩子标记向上合并
    tr[id]=tr[id<<1]&&tr[id<<1|1];
}
void build(int id,int l,int r){
    //对整棵树操作
    if(l==r){
```

```
        //到叶子节点操作
        return;
    }
    int mid=l+r>>1;
    build(ls); //左子树
    build(rs); //右子树
}
//对区间[a,b]操作
void update(int id,int l,int r,int a,int b){
    if(a<=l&&r<=b){ //全部包含在[a,b]中
        tr[id]=1; //整体处理
        return;
    }
    pushdown(id); //标记下放
    int mid=l+r>>1;
    if(a<=mid) update(ls,a,b); //左子树有部分区间在[a,b]中
    else; //这里是不包含在[a,b]区间的操作,左孩子
    if(b>mid) update(rs,a,b); //右子树有部分区间在[a,b]中
    else; //这里是不包含在[a,b]区间的操作,右孩子
    pushup(id); //标记合并
}
void query(int id,int l,int r,int a,int b){
    if(a<=l&&r<=b){ //全部包含在[a,b]中
        //查询结果
        return;
    }
    pushdown(id); //标记下放
    int mid=l+r>>1;
    if(a<=mid) query(ls,a,b); //左子树有部分区间在[a,b]中
    else; //这里是不包含在[a,b]区间的操作,左孩子
    if(b>mid) query(rs,a,b); //右子树有部分区间在[a,b]中
    else; //这里是不包含在[a,b]区间的操作,右孩子
}

/***可持久化线段树***/
/***空间复杂度 O((n+m)logn)***/
#define ls tr[k].l,l,mid
#define rs tr[k].r,mid+1,r
struct node{
    int l,r;
    int s,tag;
    node(int _s=0){
        l=r=tag=0,s=_s;
    }
```

```
    void ouput(int id){
        printf("id: %d ,l: %d ,r: %d ,s: %d ,tag: %d\n",id,l,r,s,tag);
    }
}tr[Maxn*50];
int sz; //总结点数
int rt[Maxn]; //线段树根
void pushdown(int t,int m){
    if(tr[t].tag){
        tr[++sz]=tr[tr[t].l],tr[t].l=sz; //复制左儿子
        tr[++sz]=tr[tr[t].r],tr[t].r=sz; //复制右儿子
        //正常修改
        tr[tr[t].l].tag+=tr[t].tag;
        tr[tr[t].r].tag+=tr[t].tag;
        tr[tr[t].l].s+=(m-(m>>1))*tr[t].tag;
        tr[tr[t].r].s+=(m>>1)*tr[t].tag;
        tr[t].tag=0;
    }
}
void pushup(int t){
    tr[t].s=tr[tr[t].l].s+tr[tr[t].r].s;
}
void build(int &k,int l,int r){ //建树
    k=++sz;
    tr[k]=node(0); //tag 清零,防止前一组数据影响
    if(l==r){
        int x;
        scanf("%d",&x);
        tr[k]=node(x);
        return;
    }
    int mid=l+r>>1;
    build(ls);
    build(rs);
    pushup(k);
}
void init(int n){
    sz=rt[1]=0;
    build(rt[1],1,n);
}
//单点更新
//pos 位置增加 v,注意传入 t 的引用,这样 t 的孩子才会改变
void update(int k,int l,int r,int &t,int pos,int v){
    t=++sz;
    tr[t]=node(0); //tag 清零,防止前一组数据影响
```

```
    if(l==r){
        tr[t]=node(tr[k].s+v);
        return;
    }
    tr[t].l=tr[k].l,tr[t].r=tr[k].r;
    int mid=l+r>>1;
    if(pos<=mid) update(ls,tr[t].l,pos,v);
    else update(rs,tr[t].r,pos,v);
    pushup(t);
}
//区间更新
//调用前保证 k=t,否则不会复制节点
void update(int k,int l,int r,int &t,int a,int b,int c){
    if(k==t){ //不相等可能是由于 pushdown
        t=++sz;
        tr[t]=tr[k]; //复制过来
    }
    if(a<=l&&r<=b){
        tr[t].tag+=c;
        tr[t].s+=c*(r-l+1);
        return;
    }
    int mid=l+r>>1;
    pushdown(t,r-l+1);
    if(a<=mid) update(ls,tr[t].l,a,b,c);
    if(b>mid) update(rs,tr[t].r,a,b,c);
    pushup(t);
}
//没有标记的区间查询
int query(int k,int l,int r,int a,int b){
    if(a<=l&&r<=b)
        return tr[k].s;
    int mid=l+r>>1;
    int ans=0;
    if(a<=mid) ans+=query(ls,a,b);
    if(b>mid) ans+=query(rs,a,b);
    return ans;
}
//区间查询
//调用前保证 k=t,否则不会复制节点
int query(int k,int l,int r,int &t,int a,int b){
    if(k==t){ //不相等可能是由于 pushdown
        t=++sz;
        tr[t]=tr[k]; //复制过来
```

```
    }
    if(a<=l&&r<=b)
        return tr[t].s;
    pushdown(t,r-l+1);
    int mid=l+r>>1;
    int ans=0;
    if(a<=mid) ans+=query(ls,tr[t].l,a,b);
    if(b>mid) ans+=query(rs,tr[t].r,a,b);
    return ans;
}
//有标记的单点查询
//调用前保证 k=t,否则不会复制节点
int query(int k,int l,int r,int &t,int pos){
    if(k==t){
        t=++sz;
        tr[t]=tr[k];
    }
    if(l==r)
        return tr[t].s;
    int mid=l+r>>1;
    pushdown(t,r-l+1);
    if(pos<=mid) return query(ls,tr[t].l,pos);
    return query(rs,tr[t].r,pos);
}
void print(int t){ //中序遍历
    if(!t) return;
    tr[t].ouput(t);
    print(tr[t].l);
    print(tr[t].r);
}
```

2.6.3 二维线段树

```
#define Maxn 1010
#define Maxm 1010
#define ls id<<1,l,mid
#define rs id<<1|1,mid+1,r
const int inf=0x3f3f3f3f;
struct seg{
    int mx,mn;
    seg(){}
    seg(int _mx,int _mn){mx=_mx,mn=_mn;}
    seg operator+(seg a){
```

```
        return seg(max(mx,a.mx),min(mn,a.mn));
    }
}tr[Maxn<<2][Maxm<<2];
int n,m; //x:[1,n],y:[1,m]
int lx,ly,rx,ry; //用于区间更新的全局变量
int x,y; //用于单点更新的全局变量
void pushup(int idx,int idy){
    tr[idx][idy]=tr[idx][idy<<1]+tr[idx][idy<<1|1];
}
//tag!=0,pushup 维护外层信息
void buildy(int idx,int id,int l,int r,int tag=0){
    if(l==r){
        if(!tag){
            scanf("%d",&x);
            tr[idx][id]=seg(x,x);
        }
        else{ //y 同构,idx 用左右儿子来维护
            tr[idx][id]=tr[idx<<1][id]+tr[idx<<1|1][id];
        }
        return;
    }
    int mid=l+r>>1;
    buildy(idx,ls,tag);
    buildy(idx,rs,tag);
    pushup(idx,id);
}
void buildx(int id,int l,int r){
    if(l==r){
        buildy(id,1,1,m);
        return;
    }
    int mid=l+r>>1;
    buildx(ls);
    buildx(rs);
    buildy(id,1,1,m,1); //类似 pushup
}
//单点更新(x,y)=v
void updatey(int idx,int id,int l,int r,int v,int tag=0){ //y
    if(l==r){
        if(!tag) tr[idx][id]=seg(v,v);
        else{ //维护外层
            tr[idx][id]=tr[idx<<1][id]+tr[idx<<1|1][id];
        }
        return;
```

```
    }
    int mid=l+r>>1;
    if(y<=mid) updatey(idx,ls,v,tag);
    else updatey(idx,rs,v,tag);
    pushup(idx,id);
}
void updatex(int id,int l,int r,int v){ //x
    if(l==r){
        updatey(id,1,1,m,v);
        return;
    }
    int mid=l+r>>1;
    if(x<=mid) updatex(ls,v);
    else updatex(rs,v);
    updatey(id,1,1,m,v,1);
}
//区间查询(lx,rx)->(ly,ry)
seg queryy(int idx,int id,int l,int r){ //[ly,ry]
    if(ly<=l&&r<=ry)
        return tr[idx][id];
    int mid=l+r>>1;
    seg ans(-inf,inf);
    if(ly<=mid) ans=ans+queryy(idx,ls);
    if(ry>mid) ans=ans+queryy(idx,rs);
    return ans;
}
seg queryx(int id,int l,int r){ //[lx,rx]
    if(lx<=l&&r<=rx)
        return queryy(id,1,1,m);
    int mid=l+r>>1;
    seg ans(-inf,inf);
    if(lx<=mid) ans=ans+queryx(ls);
    if(rx>mid) ans=ans+queryx(rs);
    return ans;
}
```

2.6.4 ZKW 线段树

```
/*****ZKW 线段树*****/
/***单次查询 O(logn)***/
#define Maxn 262144
/*
 *特点:
```

```
 *ZKW 线段树是一颗完全二叉树
 *根节点编号为 1,
 *如果有 M 层,最大节点编号 2^M-1
 *节点编号二进制表示最后一位
 *0 表示左儿子,1 表示右儿子
 *兄弟节点编号 XOR 为 1
*/
struct ZKW{
    int tr[Maxn]; //值
    int tag[Maxn]; //标记
    int l[Maxn],r[Maxn]; //区间端点
    int st[30],top; //栈,栈顶指针
    int n; //最后一层节点个数
    void mark(int x,int c){
        tag[x]+=c; //修改标记
        tr[x]+=c; //修改值
    }
    void pushup(int id){
        tr[id]=tr[id<<1]+tr[id<<1|1];
    }
    void pushdown(int id){
        if(tag[id]&&id<n){ //有标记并且不是叶节点
            mark(id<<1,tag[id]);
            mark(id<<1|1,tag[id]);
            tag[id]=0; //标记清 0
        }
    }
    void trace(int x){
        top=0;
        for(;x;x>>=1) st[++top]=x; //记录到根的路径
        while(top--) pushdown(st[top]); //由上向下传标记
    }
    void build(int sz){ //区间[1,sz]
        for(n=1;n<(sz+2);n<<=1); //最后一层节点至少 sz+2
        for(int i=1;i<(n<<1);i++) tag[i]=0; //初始化标记
        for(int i=0;i<n;i++) l[n+i]=r[n+i]=i;
        for(int i=0;i<n;i++){
            if(1<=i&&i<=sz) tr[n+i]=0; //初始化值
            else tr[n+i]=0; //这些值必须对维护信息没有影响
        }
        for(int i=n-1;i>=1;i--){
            l[i]=l[i<<1],r[i]=r[i<<1|1];
            pushup(i);
        }
```

```
    }
    void update(int a,int b,int c){ //区间[a,b]加标记 c
        int fa=0,fb=0;
        //左端点-1,右端点+1,i^j^1 表示 i 和 j 互为兄弟节点
        for(int i=a+n-1,j=b+n+1;i^j^1;i>>=1,j>>=1){
            if(~i&1){ //i 为左儿子
                if(!fa) //保证标记值下放一次
                    trace(fa=i^1); //标记下放
                mark(i^1,c);
            }
            if(j&1){ //j 为右儿子
                if(!fb) //保证标记值下放一次
                    trace(fb=j^1); //标记下放
                mark(j^1,c);
            }
        }
        //修改完沿 fa,fb 到根 pushup 一遍
        for(int i=fa>>1;i;i>>=1) pushup(i);
        for(int i=fb>>1;i;i>>=1) pushup(i);
    }
    int query(int a,int b){
        int fa=0,fb=0,ls=0,rs=0;
        for(int i=a+n-1,j=b+n+1;i^j^1;i>>=1,j>>=1){
            if(~i&1){ //i 为左儿子
                if(!fa)
                    trace(i^1);
                ls+=tr[i^1];
            }
            if(j&1){ //j 为右儿子
                if(!fb)
                    trace(j^1);
                rs+=tr[j^1];
            }
        }
        return ls+rs;
    }
}zkw;
```

2.6.5 Treap

```
/*****Treap(最简版)*****/
/*****单次操作 O(logn)*****/
struct treap{
```

```
    int l,r,v,d;
}tr[Maxn];
int rt,sz,ans; //每组数据 rt=sz=0
void rturn(int &k){
    int t=tr[k].l;
    tr[k].l=tr[t].r;
    tr[t].r=k;
    k=t;
}
void lturn(int &k){
    int t=tr[k].r;
    tr[k].r=tr[t].l;
    tr[t].l=k;
    k=t;
}
void insert(int &k,int x){
    if(!k){ //找到空节点,即插入位置
        k=++sz;
        tr[k].v=x;
        tr[k].l=tr[k].r=0;
        tr[k].d=rand();
        return;
    }
    if(x<tr[k].v){
        insert(tr[k].l,x);
        if(tr[tr[k].l].d<tr[k].d) rturn(k); //插入完调整树
    }
    else{
        insert(tr[k].r,x);
        if(tr[tr[k].r].d<tr[k].d) lturn(k); //插入完调整树
    }
}
void del(int &k,int x){
    if(!k) return;
    if(tr[k].v==x){ //找到删除节点
        if(tr[k].l*tr[k].r==0) k=tr[k].l+tr[k].r; //至少有一个孩子为空
        else if(tr[tr[k].l].d<tr[tr[k].r].d) rturn(k),del(tr[k].r,x);
        else lturn(k),del(tr[k].l,x);
    }
    else if(tr[k].v<x) del(tr[k].r,x);
    else del(tr[k].l,x);
}
void suc(int k,int x){ //后继
    if(!k) return;
```

```
    if(tr[k].v>=x) ans=tr[k].v,suc(tr[k].l,x); //找到后继,但可能有更小
    else suc(tr[k].r,x);
}

/*****Treap*****/
/*****单次操作 O(logn)*****/
struct treap{
    int l,r; //左右孩子
    int v; //值
    int s; //子树大小
    int d; //优先数(小顶堆)
    int w; //每个数出现的次数
}tr[Maxn];
int rt,sz,ans; //每组数据 rt=sz=0
void update(int &k){
    tr[k].s=tr[tr[k].l].s+tr[tr[k].r].s+tr[k].w;
}
void rturn(int &k){ //右旋
    int t=tr[k].l;
    tr[k].l=tr[t].r;
    tr[t].r=k;
    tr[t].s=tr[k].s;
    update(k);
    k=t;
}
void lturn(int &k){ //左旋
    int t=tr[k].r;
    tr[k].r=tr[t].l;
    tr[t].l=k;
    tr[t].s=tr[k].s;
    update(k);
    k=t;
}
void insert(int &k,int x){
    if(!k){
        k=++sz;
        tr[k].s=tr[k].w=1;
        tr[k].v=x;
        tr[k].l=tr[k].r=0;
        tr[k].d=rand();
        return;
    }
    tr[k].s++;
    if(tr[k].v==x) tr[k].w++;
```

```
    else if(tr[k].v<x){
        insert(tr[k].r,x);
        if(tr[tr[k].r].d<tr[k].d) lturn(k);
    }
    else{
        insert(tr[k].l,x);
        if(tr[tr[k].l].d<tr[k].d) rturn(k);
    }
}
void del(int &k,int x){
    if(!k) return;
    if(tr[k].v==x){
        if(tr[k].w>1){
            tr[k].w--;
            tr[k].s--;
            return;
        }
        if(tr[k].l*tr[k].r==0){
            k=tr[k].l+tr[k].r;
            return;
        }
        if(tr[tr[k].l].d<tr[tr[k].r].d){
            rturn(k);
            del(tr[k].r,x);
        }
        else{
            lturn(k);
            del(tr[k].l,x);
        }
    }
    else if(x>tr[k].v) del(tr[k].r,x);
    else del(tr[k].l,x);
    update(k);
}
//x 的排名,多个相同的 x,取排名最小的一个
int query_rank(int k,int x){
    if(!k) return 0;
    if(tr[k].v==x) return tr[tr[k].l].s+1;
    if(x<tr[k].v) return query_rank(tr[k].l,x);
    return tr[tr[k].l].s+tr[k].w+query_rank(tr[k].r,x);
}
//排名为 x 的数
int query_val(int k,int x){
    if(!k) return k;
```

```
    if(x<=tr[tr[k].l].s) return query_val(tr[k].l,x);
    if(x<=tr[tr[k].l].s+tr[k].w) return tr[k].v;
    return query_val(tr[k].r,x-tr[tr[k].l].s-tr[k].w);
}
//求严格前驱(ans<x)
void pre(int k,int x){
    if(!k) return;
    if(tr[k].v<x){
        ans=tr[k].v;pre(tr[k].r,x);
    }
    else pre(tr[k].l,x);
}
//求严格后继(ans>x)
void suc(int k,int x){
    if(!k) return;
    if(tr[k].v>x){
        ans=tr[k].v;suc(tr[k].l,x);
    }
    else suc(tr[k].r,x);
}
void print(int k){
    if(!k) return;
    print(tr[k].l);
    printf("%d ",tr[k].v);
    print(tr[k].r);
}
```

2.6.6 Splay

```
/*****Splay*****/
/*****单次操作 O(logn)*****/
const int inf=0x3f3f3f3f;
struct Splay{
    int ch[2],fa;
    int s,v,mx;
    int tg,rev; //标志
    Splay(){}
    Splay(int x){
        v=mx=x;s=1;tg=rev=ch[0]=ch[1]=0;
    }
}tr[Maxn];
int rt,sz;
int val[Maxn]; //val[1...n]为真实数组
```

```
void push_up(int x){
    int l=tr[x].ch[0],r=tr[x].ch[1];
    tr[x].s=tr[l].s+tr[r].s+1;
    tr[x].mx=max(tr[x].v,max(tr[l].mx,tr[r].mx));
}
void push_down(int x){ //不要更新节点 0 的值
    if(!x) return;
    int l=tr[x].ch[0],r=tr[x].ch[1],t=tr[x].tg;
    if(t){
        if(l) tr[l].tg+=t,tr[l].v+=t,tr[l].mx+=t;
        if(r) tr[r].tg+=t,tr[r].v+=t,tr[r].mx+=t;
        tr[x].tg=0;
    }
    if(tr[x].rev){
        tr[l].rev^=1,tr[r].rev^=1;
        swap(tr[x].ch[0],tr[x].ch[1]);
        tr[x].rev=0;
    }
}
//单旋,供内部使用
void rotate(int x,int &k){
    int y=tr[x].fa,z=tr[y].fa;
    int l,r;
    if(tr[y].ch[0]==x) l=0; //按左儿子考虑
    else l=1;
    r=l^1;
    //维护 x 与 z 的关系,特殊考虑 y=k,k 刚好是根
    if(y==k) k=x; //k 可能是根,也可能是某个节点的儿子
    else if(tr[z].ch[0]==y) tr[z].ch[0]=x;
    else tr[z].ch[1]=x;
    tr[x].fa=z; //k 为根时,这个维护本质上没有意义,但是不影响结果
    //维护 x 的右子树和 y 的关系
    tr[tr[x].ch[r]].fa=y;tr[y].ch[l]=tr[x].ch[r];
    //维护 x 和 y 的关系
    tr[y].fa=x;tr[x].ch[r]=y;
    push_up(y);push_up(x);
}
//旋转编号为 x 的节点到编号为 k 的节点
void splay(int x,int &k){
    while(x!=k){
        int y=tr[x].fa,z=tr[y].fa;
        if(y!=k){ //三点的情况
            if(tr[y].ch[0]==x^tr[z].ch[0]==y)
                rotate(x,k); //三点不共线,旋转 x
```

```
            else //三点共线,旋转 y
                rotate(y,k);
        }
        //这里包含了三种情况
        //y=k,直接旋转 x
        //y!=k,三点不共线,第二次旋转 x
        //y!=k,三点共线,旋转 y 后,旋转 x
        rotate(x,k);
    }
}
//在以 x 为根的子树中查找第 k 大元素
int find(int x,int k){
    if(tr[x].tg||tr[x].rev) push_down(x);
    int l=tr[x].ch[0],r=tr[x].ch[1];
    if(tr[l].s+1==k) return x;
    if(tr[l].s>=k) return find(l,k);
    return find(r,k-tr[l].s-1);
}
//返回原数组[l,r]对应子树的根,实际取[l+1,r+1]
int extract(int &rt,int l,int r){ //子树与原树不分开
    int x=find(rt,l),y=find(rt,r+2);
    splay(x,rt);
    splay(y,tr[x].ch[1]);
    return tr[y].ch[0];
}
int split(int &rt,int l,int r){ //子树与原树分开
    int z=extract(rt,l,r);
    int y=tr[z].fa,x=tr[y].fa;
    tr[y].ch[0]=tr[z].fa=0; //断开
    push_up(y);push_up(x);
    return z;
}
//将以 r 为根的子树合并到 rt 第 l+1 小的位置
int merge(int &rt,int l,int r){ //l=0,即插入最小元素
    int x=find(rt,l+1),y=find(rt,l+2);
    splay(x,rt);splay(y,tr[x].ch[1]);
    tr[y].ch[0]=r; //接上,r 节点为第 l+1 小
    tr[r].fa=y;
    push_up(y);push_up(x);
    return rt;
}
//返回根节点
int build(int l,int r){
    if(l>r) return 0;
```

```
    if(l==r){
        tr[++sz]=Splay(val[l]);
        return sz;
    }
    int mid=l+r>>1;
    int ls=build(l,mid-1),rs=build(mid+1,r);
    tr[++sz]=Splay(val[mid]);
    tr[sz].ch[0]=ls,tr[sz].ch[1]=rs;
    tr[ls].fa=sz,tr[rs].fa=sz;
    push_up(sz);
    return sz;
}
//val[0]和 val[n+1]为保护节点,初值任意
//因为 split 返回的区间永远不包含它们
//但必须保证 tr[].s=1,tr[].ch[]=0
void init(int n){ //n 个元素
    sz=0;
    //初始化 0 号节点,必须不能影响含有 0 号孩子的节点
    tr[0].s=0,tr[0].mx=-inf;
    rt=build(0,n+1); //区间向右平移 1 个单位
    tr[rt].fa=0;
}
void update(int l,int r,int v){
    int z=extract(rt,l,r);
    tr[z].tg+=v,tr[z].v+=v,tr[z].mx+=v;
}
void reverse(int &rt,int l,int r){
    int z=extract(rt,l,r);
    tr[z].rev^=1;
}

/**
 * 其他操作
 * 删除第 k 小,split(rt,k,k)
 * 插入第 k+1 小,值为 v,tr[++sz]=Splay(v),merge(rt,k,sz);
 *
*/
//输出以 x 为根的子树,包括保护节点
void print(int x){
    if(!x) return;
    if(tr[x].tg||tr[x].rev) push_down(x);
    print(tr[x].ch[0]);
    printf("%d ",tr[x].v);
    print(tr[x].ch[1]);
```

```
}
//按值插入
void insert(int &k,int x,int f){
    if(!k){
        k=++sz;
        tr[k]=Splay(x);
        tr[k].fa=f;
        splay(k,rt);
        return;
    }
    if(tr[x].tg||tr[x].rev) push_down(x);
    if(x<tr[k].v) insert(tr[k].ch[0],x,k);
    else insert(tr[k].ch[1],x,k);
}
//第 k 小的值修改为 v
void replace(int &rt,int k,int v){
    int x=find(rt,k+1); //考虑保护节点
    splay(x,rt);
    tr[rt].v=v;
    push_up(rt);
}
```

2.6.7 Link-Cut-Tree

```
/*****Link-Cut-Tree*****/
/*****单次操作均摊 O(logn)*****/
struct Splay{
    int ch[2],fa; //splay 的 fa 和 path-parent 共用 fa
    int s;
    int rev; //标志
    int val,sum,mx;
    Splay(int _x=0){ //注意 0 号节 s=sum=0,mx=-inf
        s=1;rev=fa=ch[0]=ch[1]=0;
        val=sum=mx=_x;
    }
}tr[Maxn];
int st[Maxn]; //用于 pushdown
void push_up(int x){ //必须保证 tr[0]的信息正确
    int l=tr[x].ch[0],r=tr[x].ch[1];
    //tr[0].s=tr[0].sum=0,tr[0].mx=-inf
    tr[x].s=1+tr[l].s+tr[r].s;
    tr[x].sum=tr[x].val+tr[l].sum+tr[r].sum;
    tr[x].mx=max(tr[x].val,max(tr[l].mx,tr[r].mx));
```

```
}
void push_down(int x){ //不要影响 tr[0]的信息
    if(!x) return;
    int l=tr[x].ch[0],r=tr[x].ch[1];
    if(tr[x].rev){
        tr[l].rev^=1,tr[r].rev^=1;
        swap(tr[x].ch[0],tr[x].ch[1]);
        tr[x].rev=0;
    }
}
//splay 的根的父亲的左右儿子都不是根,但 splay 的非根节点的父亲的左右儿
//子必然有一个是该节点
inline bool isroot(int x){ //判断 x 是否是 splay 的根
    return tr[tr[x].fa].ch[0]!=x&&tr[tr[x].fa].ch[1]!=x;
}
void rotate(int x){
    int y=tr[x].fa,z=tr[y].fa;
    int l,r;
    if(tr[y].ch[0]==x) l=0; //按左儿子考虑
    else l=1;
    r=l^1;
    //维护 x 与 z 的关系
    if(!isroot(y)){ //非根是需要设置 z 的孩子
        if(tr[z].ch[0]==y) tr[z].ch[0]=x;
        else tr[z].ch[1]=x;
    }
    tr[x].fa=z;
    //维护 x 的右子树和 y 的关系
    tr[tr[x].ch[r]].fa=y;tr[y].ch[l]=tr[x].ch[r];
    //维护 x 和 y 的关系
    tr[y].fa=x;tr[x].ch[r]=y;
    push_up(y);push_up(x);
}
void splay(int x){
    int top=0;st[++top]=x;
    for(int i=x;!isroot(i);i=tr[i].fa) st[++top]=tr[i].fa;
    for(int i=top;i;i--) push_down(st[i]);
    while(!isroot(x)){
        int y=tr[x].fa,z=tr[y].fa;
        if(!isroot(y)){ //三点的情况
            if(tr[y].ch[0]==x^tr[z].ch[0]==y)
                rotate(x); //三点不共线,旋转 x
            else //三点共线,旋转 y
                rotate(y);
```

```
        }
        //这里包含了三种情况
        //y=k,直接旋转 x
        //y!=k,三点不共线,第二次旋转 x
        //y!=k,三点共线,旋转 y 后,旋转 x
        rotate(x);
    }
}
void access(int x){ //access 之后 x 不一定是根,需要 splay(x)
    int y=0;
    while(x){ //保证原树的根的 fa 为 0,循环才会结束
        splay(x);
        tr[x].ch[1]=y; //断开右子树并连接
        push_up(x);
        //原右子树 fa 不变,新右子树的 path-parent 刚好指向 x
        y=x,x=tr[x].fa;
    }
}
void mkroot(int x){ //将 x 换成原树的根
    access(x);
    splay(x); //将 x 转成根,为了加标记
    tr[x].rev^=1; //反转 splay
}
void link(int x,int y){ //将以 x 为根的子树与 y 合并
    mkroot(x); //将 x 变成根
    tr[x].fa=y; //设置 path-parent 指向 y
}
void cut(int x,int y){ //断开 x 与 y,显然原树中 x 与 y 相邻
    mkroot(x); //将 x 变成根,x 深度最小
    access(y); //x 与 y 连成一条 prefer-path
    splay(y); //将 y 转成根,x 深度小,所以为 y 的左子树
    //断开(x,y)等价于断开 y 的左儿子
    tr[y].ch[0]=tr[x].fa=0;
    push_up(y);
}
void split(int x,int y){ //连通(x,y)这条路径
    mkroot(x); //x 变根
    access(y); //y 与 x 连通
    splay(y); //y 旋转到根
}
bool sameTree(int x,int y){ //x,y 是否在同一棵树
    split(x,y);
    while(tr[y].ch[0]) y=tr[y].ch[0];
    return x==y;
```

```
}
```

2.6.8 划分树

```
#include<algorithm>
#define ls id<<1,l,mid
#define rs id<<1|1,mid+1,r
#define Maxn 100010
/*****划分树(基于快排)*****/
/*****单次查询第 k 大,O(logn)*****/
int pw[Maxn<<2];
void init(int n){
    pw[0]=-1;
    for(int i=1;i<=n;i++)
        pw[i]=i&i-1?pw[i-1]:pw[i-1]+1;
}
int a[Maxn]; //从小到大排序后的原数组
int seq[20][Maxn]; //线段树
int s[20][Maxn]; //s[i][j]表示线段树第 i 层,[1,j]区间进入左子树的个数
void build(int id,int l,int r){
    if(l==r) return;
    int mid=l+r>>1,t=pw[id],same=0;
    for(int i=mid;i>=l;i--) //计算与 a[mid]等值的进入左子树的个数
        if(a[i]==a[mid]) same++;
        else break;
    int j=l,k=mid+1;
    for(int i=l;i<=r;i++){
        if(i==l) s[t][i]=0; //特殊处理 1
        else s[t][i]=s[t][i-1];
        if(seq[t][i]<a[mid]){ //小于中位数直接进入左子树
            seq[t+1][j++]=seq[t][i];
            s[t][i]++;
        }
        //等于中位数并且 same!=0,进入左子树
        else if(seq[t][i]==a[mid]&&same){
            seq[t+1][j++]=seq[t][i];
            s[t][i]++;
            same--;
        }
        else //进入右子树
            seq[t+1][k++]=seq[t][i];
    }
    build(ls);
```

```
    build(rs);
}
void print(int id,int l,int r){
    printf("%d %d %d:\n",id,l,r);
    for(int i=l;i<=r;i++)
        printf("%d ",seq[pw[id]][i]);
    puts("");
    if(l==r) return;
    int mid=l+r>>1;
    print(ls);
    print(rs);
}
int query(int id,int l,int r,int x,int y,int k){
    if(l==r) return a[l];
    int mid=l+r>>1,t=pw[id];
    int L=x==l?0:s[t][x-1],R=s[t][y]; //特殊处理 x-1
    if(R-L>=k) //[x,y]之间进入左子树的个数大于等于 k
        //除去[l,x-1]和[y+1,r]进入左子树的个数
        return query(ls,l+L,l+R-1,k);
    //[l,x-1]有 x-1-L 个数进入右子树,[l,y]有 y-l+1-R 个数进入右子树
    return query(rs,mid+1+x-l-L,mid+1+y-l-R,k-R+L);
}
//初始化 init(400010);
//for(int i=1;i<=n;i++) seq[0][i]=a[i];
//sort(a+1,a+n+1);
//build(1,1,n);
```

2.6.9 归并树

```
#define ls id<<1,l,mid
#define rs id<<1|1,mid+1,r
#define Maxn 100010
/*****归并树(基于归并排序)*****/
/*****单次查询第 k 大,复杂度 O((logn)^3)*****/
int pw[Maxn<<2];
int seq[20][Maxn];
int n; //元素个数
int ans; //记录排名
void init(int n){
    pw[0]=-1;
    for(int i=1;i<=n;i++)
        pw[i]=i&i-1?pw[i-1]:pw[i-1]+1;
}
void build(int id,int l,int r){
```

```
    int t=pw[id];
    if(l==r){
        scanf("%d",&seq[t][l]);
        return;
    }
    int mid=l+r>>1;
    build(ls);
    build(rs);
    int j=l,k=mid+1;
    for(int i=l;i<=r;i++) //回溯时归并
        if(j>mid||k<=r&&seq[t+1][k]<=seq[t+1][j])
           seq[t][i]=seq[t+1][k++];
        else if(k>r||seq[t+1][j]<seq[t+1][k]) seq[t][i]=seq[t+1][j++];
}
void print(int id,int l,int r){
    printf("%d %d %d:\n",id,l,r);
    for(int i=l;i<=r;i++)
        printf("%d ",seq[pw[id]][i]);
    puts("");
    if(l==r) return;
    int mid=l+r>>1;
    print(ls);
    print(rs);
}
//得到区间[l,r]小于 v 的个数
void get_rank(int id,int l,int r,int x,int y,int v){
    int t=pw[id];
    if(x<=l&&r<=y){
        if(seq[t][r]<v) ans+=r-l+1; //[l,r]都小于 v
        else //二分查找[l,r]小于 v 的个数
            ans+=lower_bound(seq[t]+l,seq[t]+r+1,v)-seq[t]-l;
        return;
    }
    int mid=l+r>>1;
    if(x<=mid) get_rank(ls,x,y,v);
    if(y>mid) get_rank(rs,x,y,v);
}
int get_val(int x,int y,int k){
    int l=1,r=n,res;
    while(l<=r){ //二分下标,线段树第 0 层存的是排序好的数组
        int mid=l+r>>1;
        ans=1;get_rank(1,1,n,x,y,seq[0][mid]);
        if(ans<=k) {res=mid;l=mid+1;} //可能更大
        else r=mid-1;
```

```
    }
    return seq[0][res];
}
//init(400010);
//build(1,1,n);
```

2.6.10 主席树

```
/*****主席树*****/
/*****单次查询第 k 大,复杂度 O(logn)*****/
/*****空间复杂度 O(nlogn)*****/
#define L l,mid,ls[x],ls[y]
#define R mid+1,r,rs[x],rs[y]
#define Maxn 2000010
//Maxn 一般开 20n
int a[100010],sa[100010]; //原数组和离散化后的数组
int root[100010]; //root[i]表示区间[1,i]线段树的根
int s[Maxn]; //s[i]表示以 i 为根的子树的 size
int ls[Maxn]; //左儿子
int rs[Maxn]; //右儿子
int sz; //节点总数,O(nlogn)
int tot; //离散化后的数量,线段树的区间[1,tot]
//在以 x 为根的线段树中插入 v,变成以 y 为根的线段树
void update(int l,int r,int x,int &y,int v){
    y=++sz;
    s[y]=s[x]+1;
    if(l==r){
        ls[y]=rs[y]=0; //防止前一组数据影响
        return;
    }
    ls[y]=ls[x],rs[y]=rs[x];
    int mid=l+r>>1;
    if(v<=mid) update(L,v);
    else update(R,v);
}
//[x,y]第 k 大,x 传入 root[x-1],y 传入 root[y]
//返回的是 sa[]的下标
int query(int l,int r,int x,int y,int k){
    if(l==r) return l;
    int mid=l+r>>1;
    if(s[ls[y]]-s[ls[x]]>=k) return query(L,k);
    return query(R,k-s[ls[y]]+s[ls[x]]);
}
```

```
void init(){
    root[0]=ls[0]=rs[0]=s[0]=sz=0; //空树
    sort(sa+1,sa+n+1);
    tot=unique(sa+1,sa+n+1)-sa-1; //离散化
}
```

2.6.11 树套树(线段树+Treap)

```
/*****线段树套平衡树*****/
/*****单次操作 O(n(logn)^2)*****/
/***get_val 操作 O(n(logn)^3)***/
/***空间复杂度(nlogn+mlogn),n 和节点数,m 为单点更新总次数***/
const int inf=0x3f3f3f3f;
struct treap{
    int l,r;
    int v,s,w,d; //值,子树大小,重复元素个数,优先值
}tr[Maxn*40]; //基本空间 O(nlogn+mlogn)
int root[Maxn<<2]; //4 倍节点数
int sz; //初始化 0,表示空间的占用量
int ans; //rank,pre,suc 需要使用
int n; //元素个数
int a[Maxn]; //原数组,下标从 1 开始
void update(int k){
    tr[k].s=tr[tr[k].l].s+tr[k].w+tr[tr[k].r].s;
}
void rturn(int &k){
    int t=tr[k].l;
    tr[k].l=tr[t].r;
    tr[t].r=k;
    update(k);
    update(t);
    k=t;
}
void lturn(int &k){
    int t=tr[k].r;
    tr[k].r=tr[t].l;
    tr[t].l=k;
    update(k);
    update(t);
    k=t;
}
void insert(int &k,int x){
    if(!k){
```

```
        k=++sz;
        tr[k].v=x;
        tr[k].w=tr[k].s=1;
        tr[k].d=rand();
        return;
    }
    tr[k].s++;
    if(x==tr[k].v) tr[k].w++;
    else if(x<tr[k].v){
        insert(tr[k].l,x);
        if(tr[tr[k].l].d<tr[k].d) rturn(k);
    }
    else{
        insert(tr[k].r,x);
        if(tr[tr[k].r].d<tr[k].d) lturn(k);
    }
}
void del(int &k,int x){
    if(!k) return;
    if(x==tr[k].v){
        if(tr[k].w>1){
            tr[k].w--;
            tr[k].s--;
            return;
        }
        if(tr[k].l*tr[k].r==0){
            k=tr[k].l+tr[k].r;
            return;
        }
        if(tr[tr[k].l].d<tr[tr[k].r].d){
            rturn(k);
            del(tr[k].r,x);
        }
        else{
            lturn(k);
            del(tr[k].l,x);
        }
    }
    else if(x<tr[k].v) del(tr[k].l,x);
    else del(tr[k].r,x);
    update(k);
}
//平衡树中比 x 小的个数,排名还需+1
void ask_rank(int k,int x){ //ans 初始化为 1
```

```
    if(!k) return;
    if(x==tr[k].v){
        ans+=tr[tr[k].l].s;
        return;
    }
    else if(x<tr[k].v) ask_rank(tr[k].l,x);
    else{
        ans+=tr[tr[k].l].s+tr[k].w;
        ask_rank(tr[k].r,x);
    }
}
void pre(int k,int x){ //严格前驱
    if(!k) return;
    if(x>tr[k].v){
        ans=max(ans,tr[k].v);
        pre(tr[k].r,x);
    }
    else pre(tr[k].l,x);
}
void suc(int k,int x){ //严格后继
    if(!k) return;
    if(x<tr[k].v){
        ans=min(ans,tr[k].v);
        suc(tr[k].l,x);
    }
    else suc(tr[k].r,x);
}
void print(int k){
    if(!k) return;
    print(tr[k].l);
    printf("%d ",tr[k].v);
    print(tr[k].r);
}
//插入下标为 x 的元素 a[x]
void build(int id,int l,int r,int x){
    insert(root[id],a[x]);
    if(l==r) return;
    int mid=l+r>>1;
    if(x<=mid) build(ls,x);
    else build(rs,x);
}
//查询区间[x,y],元素 v 的排名,调用前 ans 初始化为 1
void get_rank(int id,int l,int r,int x,int y,int v){
    if(x<=l&&r<=y){
```

```
        ask_rank(root[id],v);
        return;
    }
    int mid=l+r>>1;
    if(x<=mid) get_rank(ls,x,y,v);
    if(y>mid) get_rank(rs,x,y,v);
}
//查询区间[x,y]排名为k的元素
int get_val(int x,int y,int k){
    int l=0,r=inf,res;
    while(l<=r){
        int mid=l+r>>1;
        ans=1;get_rank(1,1,n,x,y,mid);
        if(ans<=k) {res=mid;l=mid+1;}
        else r=mid-1;
    }
    return res;
}
//将a[pos]修改为x,实际数组a[pos]也须修改为x
void update(int id,int l,int r,int pos,int x){
    del(root[id],a[pos]);
    insert(root[id],x);
    if(l==r) return;
    int mid=l+r>>1;
    if(pos<=mid) update(ls,pos,x);
    else update(rs,pos,x);
}
//询问区间[l,r]的严格前驱,ans初始化为无穷小
void get_pre(int id,int l,int r,int x,int y,int v){
    if(x<=l&&r<=y){
        pre(root[id],v);
        return;
    }
    int mid=l+r>>1;
    if(x<=mid) get_pre(ls,x,y,v);
    if(y>mid) get_pre(rs,x,y,v);
}
//询问区间[l,r]的严格后继,ans初始化为无穷大
void get_suc(int id,int l,int r,int x,int y,int v){
    if(x<=l&r<=y){
        suc(root[id],v);
        return;
    }
    int mid=l+r>>1;
```

```
    if(x<=mid) get_suc(ls,x,y,v);
    if(y>mid) get_suc(rs,x,y,v);
}
```

2.6.12 字典树

```
/***Trie***/
struct Trie{
    int ch[Maxn][26]; //0 表示没有孩子
    int val[Maxn]; //附加属性
    int sz; //节点总数
    void init(){
        sz=val[0]=0;
        memset(ch[0],0,sizeof(ch[0]));
    }
    void insert(char *s){ //不统计 val[0]了
        int u=0;
        int len=strlen(s);
        for(int i=0;i<len;i++){
            int v=s[i]-'a';
            if(!ch[u][v]){
                ch[u][v]=++sz;
                memset(ch[sz],0,sizeof ch[sz]);
            }
            u=ch[u][v];
            val[u]++; //每个节点被单词覆盖次数
        }
        //此处 u 为单词结束的节点
    }
    void query(char *s){
        int u=0;
        int len=strlen(s);
        for(int i=0;i<len;i++){
            int v=s[i]-'a';
            u=ch[u][v];
            printf("%c",s[i]);
            if(val[u]==1) break;
        }
    }
}tr;
```

2.6.13AC 自动机

```
/*****AC 自动机*****/
/***复杂度:所有模式串+匹配串总长度***/
const int NSZ=26;
struct AC_Automaton{
    int ch[Maxn][NSZ];
    int val[Maxn]; //字符串编号,大于 0
    int pre[Maxn]; //前缀指针
    int last[Maxn]; //存上一个单词终结点的数组下标
    int sz; //节点总数
    void init(){
        sz=val[0]=pre[0]=last[0]=0;
        memset(ch[0],0,sizeof(ch[0]));
    }
    int idx(char c){
        return c-'a';
    }
    void insert(char *s,int id){ //id 表示字符串编号
        int u=0;
        for(int i=0;s[i];i++){
            int c=idx(s[i]);
            if(!ch[u][c]){
                ch[u][c]=++sz;
                val[sz]=0;
                memset(ch[sz],0,sizeof(ch[0]));
            }
            u=ch[u][c];
        }
        val[u]=id;
    }
    void getfail(){
        queue<int> q;
        for(int c=0;c<NSZ;c++){
            int u=ch[0][c];
            if(u){
                pre[u]=last[u]=0;
                q.push(u);
            }
        }
        while(!q.empty()){
            int r=q.front();
            q.pop();
```

```
            for(int c=0;c<NSZ;c++){
                int u=ch[r][c];
                if(!u){ //没有对应儿子的补边
                    ch[r][c]=ch[pre[r]][c];
                    continue;
                }
                q.push(u);
                pre[u]=ch[pre[r]][c];
                //递推求 last 数组
                last[u]=val[pre[u]]?pre[u]:last[pre[u]];
            }
        }
    }
    void find(char *s){
        int u=0;
        for(int i=0;s[i];i++){
            int c=idx(s[i]);
            u=ch[u][c];
            //找 last 链的 val 值非 0 进行递归统计
            print(val[u]?u:last[u]);
        }
    }
    void print(int u){
        if(u){ //只要不为 0,说明是单词终结点
            //使用 val[u]统计信息
            print(last[u]);
        }
    }
}ac;
```

2.6.14 后缀树

```
/***SuffixTree***/
/***复杂度 O(n)***/
/***空间需要 2n,因此 Maxn=2n***/
const int NSZ=256;
const int inf=0x3f3f3f3f;
struct SuffixTree{
    int an; //活动节点
    int ae; //活动边
    int al; //活动长度
    int pos; //字符串插入位置
    int sz; //节点个数
```

```
int rm; //剩余后缀数
int sl; //是否需要后缀链接
int l[Maxn],r[Maxn]; //区间[l,r),左闭右开
int link[Maxn]; //后缀链接
int ch[Maxn][NSZ]; //孩子
char text[Maxn]; //建后缀树的字符串
int R(int x){ //r[x]=inf,表示 r[x]=pos+1
    return min(r[x],pos+1);
}
int length(int x){ //返回区间长度
    return R(x)-l[x];
}
int idx(char c){
    //if(c=='$') return NSZ-1;
    return c-'\0';
}
int node(int s,int e=inf){ //新建一个节点
    ++sz;
    l[sz]=s,r[sz]=e,link[sz]=0;
    memset(ch[sz],0,sizeof ch[sz]);
    return sz;
}
void init(){ //初始化根节点,0 号节点
    an=ae=al=sl=rm=0,sz=pos=-1;
    node(-1); //0 号节点
}
void add(int split){ //维护后缀链接
    if(sl) link[sl]=split; //第一次 sl 为 0,不用链接
    sl=split; //之后记录上一个 split
}
bool walk(int node){
    int len=length(node);
    //这种情况说明之前已有边分裂,因此沿分裂过的边往下走
    if(al>=len){ //设置新的活动点
        ae+=len;
        al-=len;
        an=node;
        return true;
    }
    return false;
}
void extend(char c){ //往后缀树上加一个字符
    text[++pos]=c;
    sl=0;
```

```
        rm++;
        while(rm){
            if(!al) ae=pos;
            //活动节点后面没有插入字符对应的孩子,因此直接新建节点
            int nc=idx(text[ae]);
            int nxt=ch[an][nc];
            if(!nxt){
                ch[an][nc]=node(pos);
                add(an); //单个字符的后缀链接指向空串
            }
            else{ //活动节点后面已经有插入字符对应的孩子
                if(walk(nxt)) continue;
                if(text[l[nxt]+al]==c){ //al 对应的字符是插入字符
                    al++;
                    add(an);
                    break;
                }
                /*
                 *active_len 对应的字符不是插入字符,则需分裂
                 *nxt:[start,#]分裂成 split:[start,start+active_len)
                 *和 nxt:[start+active_len,#]
                 *active_node->split->nxt
                 *生成插入字符对应的叶子节点 leaf:[pos,#],split->leaf
                */
                int split=node(l[nxt],l[nxt]+al);
                ch[an][nc]=split; //an->split
                ch[split][idx(c)]=node(pos); //split->leaf
                l[nxt]+=al;
                ch[split][idx(text[l[nxt]])]=nxt; //split->nxt
                add(split); //维护后缀链
            }
            rm--; //新建或分裂节点,剩余后缀数-1
            if(!an&&al) al--,ae=pos-rm+1;
            else an=link[an]; //沿后缀链走
        }
    }
    int dfs(int u){ //返回 u 的叶子节点个数
        int ans=0;
        bool flag=false;
        for(int i=0;i<NSZ;i++){
            int v=ch[u][i];
            if(v){
                ans+=dfs(v);
                flag=true;
```

```
            }
        }
        if(!flag) return 1;
        return ans;
    }
    int find(char *s){ //判断 s 是否是 text 的子串
        int u=0;
        int len=strlen(s);
        for(int i=0;i<len;){
            int v=ch[u][idx(s[i])];
            if(!v) return 0;
            for(int j=l[v];j<R(v);j++){
                if(i==len) break; //找到
                if(s[i]!=text[j]) return 0;
                i++;
            }
            u=v;
        }
        //return 1;
        return dfs(u); //查 s 出现次数
    }
}tr;

/***另一版本***/
//rule1:叶节点,直接加在边上
//rule2:内部节点且没有 s[i]儿子,加叶子;边上对应位置字符不是 c 须分裂
//rule3:内部节点且有 s[i]儿子,什么都不做
/*
 *实现细节:
 *根节点为 1 号节点,其中 0 号节点是为了方便
 *叶节点的 r 用 inf 表示
 *只维护内部节点的 depth,叶子节点需用 fa 的 depth+自己的 len
 *技巧 1:
 *0 号节点的 l=-2,这样 next()返回 1,于是 go()
 *可以保证 st+1,也就是从 s[st+1,i]继续匹配
 *出现上面的情况是后缀链接指向根节点,当前匹配
 *s[st,i]实际已完成,而后缀链接实际应为空
*/
const int NSZ=26;
const int inf=0x3f3f3f3f;
int pos; //当前处理字符串编号
int idx(char c){
    return c-'a';
}
```

```
struct Node{
    int ch[NSZ];
    int link,l,r,fa,depth;
    int next(char c){
        if(l==-2) return 1;
        if(c=='\0') return 0; //防越界
        return ch[idx(c)];
    }
    void node(int _l,int _r,int _fa){
        l=_l,r=_r,link=depth=0,fa=_fa;
        memset(ch,0,sizeof ch);
    }
    int len(){
        return min(r,pos+1)-l;
    }
    void dep(int x){
        depth=x+len();
    }
};
//char s[Maxn];
struct SiffixTree{
    int sz,st,ed; //ed 为当前扩展节点,st 为 ed 之后 s[st]为当前扩展字符
    char s[Maxn];
    Node node[Maxn<<1];
    queue<int> leaf; //存放叶子节点
    void init(){
        node[0].node(-2,-1,0);
        node[1].node(-1,0,0);
        sz=1,ed=1,st=0;
        while(!leaf.empty()) leaf.pop();
    }
    void go(int t){ //沿后缀链到达节点后继续往下走
        //注意 go(i+1)则 st 可能越界,因此保证 s 的结束符为'\0'
        while(st<=t){
            int p=node[ed].next(s[st]);
            if(node[p].len()>t-st) break;
            st+=node[p].len();
            ed=p;
        }
    }
    int add(int i){ //第 i 阶段扩展
        pos=i; //维护 pos
        char c=s[i]; //扩展字符
        int last=1,cur;
```

```
        while(st<=i){ //s[st,i]正在加入后缀树
            cur=ed; //当前节点
            if(st<i){
                char nc=s[st];
                int p=node[ed].next(nc); //必定存在 p
                int k=node[p].l+i-st;
                if(s[k]==c) break; //rule3
                //split,rule2
                cur=++sz; //下面需要链接一个叶子节点
                node[sz].node(node[p].l,k,ed); //[node[p].l,k)
                node[sz].dep(node[ed].depth); //维护内部节点的深度
                node[sz].ch[idx(s[k])]=p; //split->p
                node[ed].ch[idx(nc)]=sz; //ed->split
                node[p].fa=sz;
                node[p].l=k; //[k,node[p].r)
            }
            else if(node[ed].next(c)) break;
            //rule2,共用代码
            ++sz;
            node[sz].node(i,inf,cur);
            node[cur].ch[idx(c)]=sz;
            leaf.push(sz);
            //cur 必定是内部节点
            //定理保证扩展 s[st-1,i]到 s[st,i]内部节点必产生后缀链接
            if(last!=1) node[last].link=cur;
            last=cur;
            ed=node[ed].link; //沿后缀链接走
            go(i); //接下来维护 s[st+1,i]
        }
        //这里的情况是沿后缀链接到内部节点,
        //而后面字符恰好有 c,而 break
        if(last!=1) node[last].link=cur;
        go(i+1);
        return leaf.size(); //返回不同子串增加数
    }
    void merge(int u,int son){ //node[u].fa->son
        int fa=node[u].fa;
        node[son].l-=node[u].len();
        node[son].fa=fa;
        node[fa].ch[idx(s[node[u].l])]=son;
        if(ed==u){ //修改 ed 和 st
            ed=fa;
            st-=node[u].len();
        }
```

```
    }
    int del(int i){ //返回减少的不同子串数,i=pos
        int ans=0;
        int u=leaf.front();leaf.pop();
        int fa=node[u].fa;
        int active=ed;
        if(st<=i) active=node[ed].next(s[st]);
        //st>i,剩余后缀数为0,活动边为'',所以不用移到孩子
        if(active!=u){ //可删除 u 的情况
            node[fa].ch[idx(s[node[u].l])]=0;
            int cnt=0,son;
            for(int i=0;i<NSZ;i++)
                if(node[fa].ch[i]){
                    cnt++;
                    son=node[fa].ch[i];
                }
            ans+=node[u].len();
            if(cnt==1&&fa!=1) //只有一个孩子且不是根节点
                merge(fa,son); //合并边
        }
        else{ //ed->u
            ans+=st-node[u].l; //不同子串减少数
            node[u].l=st; //缩短边长
            ed=node[ed].link; //沿后缀链走
            go(i+1);
            leaf.push(u);
        }
        return ans;
    }
}tr;
```

2.6.15 后缀自动机

```
/***后缀自动机***/
/***复杂度 O(n)***/
/***空间开销为 2n,因此 Maxn=2n***/
const int NSZ=26;
struct SuffixAutomaton{ //根节点为 1 号节点
    int ch[Maxn][NSZ];
    int pre[Maxn];
    int mx[Maxn];
    int pos[Maxn]; //原串中最大结束位置
    int val[Maxn]; //标志
```

```
int sz; //节点总数
int cur; //当前要处理的节点
int idx(char c){
    return c-'a';
}
int node(int p,int maxpos){
    ++sz;
    memset(ch[sz],0,sizeof(ch[sz]));
    mx[sz]=mx[p]+1;
    val[sz]=0;
    pos[sz]=maxpos;
    return sz;
}
void init(){ //根为 1 号节点
    mx[1]=pre[1]=val[1]=pos[1]=0;
    sz=cur=1;
    memset(ch[1],0,sizeof(ch[1]));
}
void add(int tag){ //在每个后缀结束节点加标志
    int p=cur;
    while(p!=1){
        val[p]=tag;
        p=pre[p];
    }
}
void extend(char s,int maxpos){ //s 在字符串中的下标,从 1 开始
    int p=cur;
    int np=node(p,maxpos);
    int c=idx(s);
    while(p&&!ch[p][c]){ //沿 pre 指针向前
        ch[p][c]=np;
        p=pre[p];
    }
    if(!p) pre[np]=1; //p 为 0 号节点,np 的 pre 指向根
    else{ //p 有 c 孩子
        int q=ch[p][c]; //q 为 p 的 c 孩子
        if(mx[p]+1==mx[q]) //距离差 1,q 作为终态
            pre[np]=q; //np 的 pre 指向 q
        else{
            int nq=node(p,maxpos); //新建 q 的一个副本
            memcpy(ch[nq],ch[q],sizeof(ch[q])); //q 的儿子赋值给 nq
            pre[nq]=pre[q]; //q 的 pre 指针赋值给 nq
            pre[q]=pre[np]=nq; //调整 q 和 np 的 pre 指针指向 nq
            while(p&&ch[p][c]==q){ //沿 p 的 pre 指针往前走
```

```
                    ch[p][c]=nq; //将原先 c 孩子为 q 的调整为 nq
                    p=pre[p];
                }
            }
        }
        cur=np; //重置插入节点编号
    }
    int query(char *s){ //返回 s 的最长前缀且是自动机的后缀
        int u=1,ans=0,dp=0;
        for(int i=0;s[i];i++){
            int v=ch[u][idx(s[i])];
            if(!v) return ans;
            dp++;
            if(val[v]) ans=max(ans,dp);
            u=v;
        }
        return ans;
    }
}sam;

//基数排序,之后按 rk 从大到小拓扑
int sum[Maxn],rk[Maxn];
void radix_sort(){
    memset(sum,0,sizeof sum);
    for(int i=1;i<=sz;i++) sum[mx[i]]++;
    for(int i=1;i<=Maxn/2;i++) sum[i]+=sum[i-1];
    for(int i=1;i<=sz;i++) rk[sum[mx[i]]--]=i;
}

/*支持删除后缀,增加字符,查询长度为 L 的子串或者长度小于 L 的后缀*/
/*删除只修改标记,并且借助指针的思想,删除后缀的自动机并不一定是最简自动机*/
char s[Maxn];
const int NSZ=26;
struct SuffixAutomaton{ //根节点为 1 号节点
    int ch[Maxn][NSZ];
    int pre[Maxn];
    int mx[Maxn];
    int val[Maxn]; //标志
    int pos[Maxn]; //pos[i]表示第 i 个字符在后缀自动机上的节点编号
    int ed[Maxn]; //right 集合中最小位置
    int DEL[Maxn]; //修改 DEL 可以同步修改节点副本
    int del[Maxn]; //当前节点是否被删除,存 DEL 下标
    int dt; //DEL 的范围[0,dt-1]
```

```
int vt; //val[]=vt 表示后缀
int tot; //当前字符串长度
int sz; //节点总数
int cur; //当前要处理的节点
int L; //长度等于 L 的子串或者小于等于 L 的后缀
int idx(char c){
    return c-'a';
}
int node(int p){
    ++sz;
    memset(ch[sz],0,sizeof(ch[sz]));
    ed[sz]=mx[sz]=mx[p]+1;
    val[sz]=0;
    return sz;
}
void init(){ //根为 1 号节点
    mx[1]=pre[1]=val[1]=dt=vt=tot=0;
    sz=cur=pos[0]=1;
    memset(ch[1],0,sizeof(ch[1]));
}
void add(int tag){ //在每个后缀结束节点加标志
    int p=cur;
    while(p!=1){
        val[p]=tag;
        p=pre[p];
    }
}
void extend(char s){
    int p=cur;
    int np=node(p);
    pos[++tot]=np; //主链上的节点
    DEL[dt]=0; //初始化未被删除
    del[np]=dt++; //类似指针
    int c=idx(s);
    while(p&&(!ch[p][c]||DEL[del[ch[p][c]]])){ //沿 pre 指针向前
        ch[p][c]=np;
        p=pre[p];
    }
    if(!p) pre[np]=1; //p 为 0 号节点,np 的 pre 指向根
    else{ //p 有 c 孩子
        int q=ch[p][c]; //q 为 p 的 c 孩子
        if(mx[p]+1==mx[q]) //距离差 1,q 作为终态
            pre[np]=q; //np 的 pre 指向 q
        else{
```

```
                int nq=node(p); //新建 q 的一个副本
                memcpy(ch[nq],ch[q],sizeof(ch[q])); //q 的儿子赋值给 nq
                pre[nq]=pre[q]; //q 的 pre 指针赋值给 nq
                pre[q]=pre[np]=nq; //调整 q 和 np 的 pre 指针指向 nq
                ed[nq]=ed[q]; //复制
                del[nq]=del[q]; //复制
                while(p&&ch[p][c]==q){ //沿 p 的 pre 指针往前走
                    ch[p][c]=nq; //将原先 c 孩子为 q 的调整为 nq
                    p=pre[p];
                }
            }
        }
        cur=np; //重置插入节点编号
    }
    void Delete(int len){ //删除长度为 len 的后缀
        for(int i=tot+1-len;i<=tot;i++)
            DEL[del[pos[i]]]=1;
        tot-=len;
        cur=pos[tot]; //tot=0,cur=1
    }
    int dfs(int u,int dep){ //返回最小下标
        if(dep==L) return ed[u]+1-dep; //长度为 L 的子串
        if(val[u]==vt) return tot+1-dep; //长度小于 L 的后缀
        for(int i=0;i<NSZ;i++){
            int v=ch[u][i];
            if(!v||DEL[del[v]]) continue;
            return dfs(v,dep+1);
        }
    }
    int solve(int len){
        L=len;
        add(++vt);
        return dfs(1,0);
    }
}sam;
```

2.6.16 回文树

```
/***PalindromeTree***/
/***时间 O(n*字符集大小),空间最多 n+2 个节点***/
/*
 *原理:
 *last 记录以 s[i-1]结尾的最长回文串对应节点编号
```

```
 *当考虑加 s[i]时,沿 last 的后缀链走找到左端字符为 s[i]
 *得到的就是以 s[i]结尾的最长回文串,同时沿 last.link
 *的后缀链重复上述过程,维护当前节点的 link
 *由于 xAx,那么找到 xBx,B 是 A 的回文后缀,根据 A 是回文串,则可以对应
 *在 A 的前缀找到 Bx,加上 A 之前的 x,就得到了 xBx,保证存在后缀链指向的节点
*/
const int NSZ=26;
int idx(char c){
    return c-'a';
}
struct Node{
    int len; //回文长度
    int link; //后缀链接
    int cnt; //节点对应回文串个数
    int ed; //以当前加入字符为结尾的回文串个数
    int ch[NSZ];
    void node(int l){
        len=l,link=1,cnt=ed=0;
        memset(ch,0,sizeof ch);
    }
};
//注意 s[0]=-1,表示未出现过的字符
//真正使用从下标 1 开始
char s[Maxn];
struct PalindromeTree{
    int sz;
    int last; //最长回文后缀
    Node node[Maxn];
    void init(){
        node[0].node(0); //偶数长度回文中心
        node[1].node(-1); //奇数长度回文中心
        sz=1,last=0; //last 初始为 0,表示空串
        s[0]=-1; //未出现字符
    }
    int go(int x,int i){ //沿后缀链接走,找到 xAx 形式的回文
        while(s[i-node[x].len-1]!=s[i])
            x=node[x].link;
        return x;
    }
    //加第 i 个字符,倒着加字符串必须反转
    int add(int i){ //返回以 s[i]结尾的回文串个数
        int c=idx(s[i]);
        last=go(last,i); //沿上次最长回文后缀走
```

```
        if(!node[last].ch[c]){ //节点不存在
            ++sz; //创建新节点
            node[sz].node(node[last].len+2); //父亲节点长度+2
            //找 last 的后缀链接满足左端是 c 字符
            int cur=go(node[last].link,i);
            //先赋值给 link,再赋值 sz,这样 node[1]的孩子可保证 link=0
            node[sz].link=node[cur].ch[c];
            node[last].ch[c]=sz;
            //维护以 s[i]结尾的回文串个数
            node[sz].ed=node[node[sz].link].ed+1;
        }
        last=node[last].ch[c]; //维护好当前最长回文后缀
        node[last].cnt++; //节点对应回文串+1
        //此时沿 link 向上必然都是符合的回文后缀,
        //这里不做都最后 count 再统计
        return node[last].ed;
    }
    int count(){ //每个节点对应回文串数
        int ans=0; //注意爆 int
        for(int i=sz;i>=2;i--){
            node[node[i].link].cnt+=node[i].cnt;
            ans=ans+node[i].cnt;
        }
        return ans;
    }
}t;
```

2.6.17RMQ

```
/*****RMQ*****/
/*****复杂度 O(nlogn)*****/
//最大跨度 1<<20
int p[Maxn],d[Maxn][20],height[Maxn]; //需要查找的数组
void rmq_init(int n){
    p[0]=-1;
    for(int i=1;i<=n;i++) //2^p[i]<=i,i>=1
        p[i]=i&i-1?p[i-1]:p[i-1]+1; //100...0 二进制形式 p[i]=p[i-1]+1
    for(int i=1;i<=n;i++) d[i][0]=height[i];
    for(int j=1;j<=p[n];j++)
        for(int i=1;i+(1<<j)-1<=n;i++)
            d[i][j]=min(d[i][j-1],d[i+(1<<j-1)][j-1]);
}
int rmq_ask(int l,int r){ //返回[l,r]的最小值
```

```
    int k=p[r-l+1];
    return min(d[l][k],d[r-(1<<k)+1][k]);
}
```

2.7 哈希

2.7.1 常用字符串哈希函数

```
#define ui unsigned int
ui BKDRHash(char *str){
    ui seed=131,h=0;
    while(*str)
        h=h*seed+(*str++);
    return h&0x7FFFFFFF;
}
ui APHash(char *str){
    ui h=0;
    for(int i=0;*str;i++){
        if(i&1) h^=~((h<<11)^(h>>5)^(*str++));
        else h^=(h<<7)^(h>>3)^(*str++);
    }
    return h&0x7FFFFFFF;
}
ui ELFHash(char *str){
    ui h=0,x;
    while(*str){
        h=(h<<4)+(*str++);
        x=h&0xF0000000;
        if(x) h^=x>>24,h&=~x;
    }
    return h&0x7FFFFFFF;
}
ui SDBMHash(char *str){
    ui h=0;
    while(*str){ //n=65599*h+(*str++)
        h=(h<<6)+(h<<16)-h+(*str++);
    }
    return h&0x7FFFFFFF;
}
ui RSHash(char *str){
    ui b=378551,a=63689,h=0;
    while(*str){
        h=h*a+(*str++);
```

```
        a*=b;
    }
    return h&0x7FFFFFFF;
}
ui JSHash(char *str){
    ui h=1315423911;
    while(*str){
        h^=(h<<5)+(h>>2)+(*str++);
    }
    return h&0x7FFFFFFF;
}
ui DJBHash(char *str){
    ui h=5381;
    while(*str){
        h=(h<<5)+(*str++);
    }
    return h&0x7FFFFFFF;
}
```

2.7.2 LCPhash

```
/*****Prefix-Hash*****/
/*****访问单个子串的复杂度 O(1)*****/
const ll base=131; //随机数,尽量取质数
const ll mod=1000000007;
ll pw[Maxn];
void init(){
    pw[0]=1;
    for(int i=1;i<Maxn;i++)
        pw[i]=pw[i-1]*base%mod;
}
void get(char *s,ll *h){ //得到 s 的前缀哈希数组
    h[0]=s[0]; //小写字母可以-'a'
    for(int i=1;s[i];i++)
        h[i]=(h[i-1]*base+s[i])%mod;
}
ll subStr(ll *h,int s,int t){ //返回子串[s,t]的哈希值
    if(!s) return h[t];
    ll ans=(h[t]-h[s-1]*pw[t+1-s])%mod;
    if(ans<0) ans+=mod;
    return ans;
}
```

2.7.3 New_LCP

```
/*****New_LCP(类似 RMQ),有概率出错*****/
/*****预处理 O(nlogn),单次查询 O(logn)*****/
#include<cstdlib>
#include<ctime>
#define ull unsigned long long
ull ra[30]; //给 ASCII 码赋一个随机值
ull dp[Maxn][20]; //dp[i][j]为 s[i...i+2^j-1]的哈希值
int pw[Maxn]; //2^pw[i]<=i
char s[Maxn]; //所有字符串拼接成一个字符串
int id1[Maxn]; //标识每个字符串的起始位置
int id2[Maxn]; //标识每个字符串的结束位置
ull bigrand(){ //随机产生一个 ull 类型的数字
    ull rd=rand();
    for(int i=0;i<3;i++)
        rd=(rd<<16)+rand();
    return rd;
}
//s[]的下标从 1 开始,对 s 进行一个映射,此处只考虑小写字母
void init(int n){
    pw[0]=-1;
    for(int i=1;i<=n;i++)
        pw[i]=i&i-1?pw[i-1]:pw[i-1]+1;
    for(int i=0;i<30;i++)
        ra[i]=bigrand();
    for(int i=1;i<=n;i++) dp[i][0]=ra[s[i]-'a'];
    for(int j=1;j<=pw[n];j++)
        for(int i=1;i+(1<<j)-1<=n;i++)
            dp[i][j]=dp[i][j-1]*ra[s[i]-'a']^dp[i+(1<<j-1)][j-1];
}
int getLCP(int a,int b){ //第 a 个字符串和第 b 个字符串的 LCP
    int c=id1[a],d=id1[b];
    for(int i=18;i>=0;i--){ //二分逼近
        if(c+(1<<i)-1<=id2[a]&&d+(1<<i)-1<=id2[b]
        &&dp[c][i]==dp[d][i]){
            c+=1<<i;d+=1<<i;
        }
    }
    return c-id1[a];
}
```

2.7.4 线性探测

```
const int mod=4000037;
struct ha{
    int num; //元素个数
    unsigned v; //值
}hash[4000037];
int HASH(unsigned x){ //线性探测
    int t=x%mod;
    //直到找到一个空桶或者桶的值为 x
    while(hash[t].num>0&&hash[t].v!=x)
        t=(t+1)%mod;
    return t; //返回桶号
}
```

2.7.5 开散列判冲突

```
//<键类型,值类型,哈希表大小,哈希取的模值>
template<class ktype,class vtype,int hsz,int mod=10007>
struct HASH_MAP{
    int head[mod],next[hsz];
    ktype kval[hsz]; //键
    vtype val[hsz]; //值
    int sz;
    void init(){
        sz=0;
        memset(head,-1,sizeof head);
    }
    int hash_fun(ktype key){
        return key%mod;
    }
    int find(ktype key){ //返回存 key 的数组下标
        int k=hash_fun(key);
        for(int i=head[k];i!=-1;i=next[i])
            if(key==kval[i]) return i;
        return -1; //不存在键值 key
    }
    void insert(ktype key,vtype value){
        int k=find(key);
        if(k==-1){ //新建 key 的哈希值存储空间
            val[sz]=value;
            kval[sz]=key;
```

```
            next[sz]=head[key%mod];
            head[key%mod]=sz++;
        }
        else //已经存在做的操作(累加、覆盖等)
            val[k]+=value;
    }
};
```

2.7.6 Rabin_Karp

```
//未加冲突判断
#define ll long long
const int mod=1e9+7;
bool Rabin_Karp(char *ob,char *pat){
    int m=strlen(ob),n=strlen(pat);
    ll base=131,exp=1;
    for(int i=0;i<n-1;i++) //exp=base^(n-1)
        exp=exp*base%mod;
    ll p=0,o=0;
    for(int i=0;i<n;i++){
        p=(p*base+pat[i])%mod;
        o=(o*base+ob[i])%mod;
    }
    if(o==p) return true;
    for(int i=n;i<m;i++){ //高位移出低位移入
        o=((o-ob[i-n]*exp)*base+ob[i])%mod;
        if(o<0) o+=mod; //小心负数
        if(o==p) return true;
    }
    return false;
}
```

2.8 字符串

2.8.1 最大最小表示

```
/***字符串的最小(大)表示***/
/***复杂度O(n)***/
char s[Maxn];
//flag=1,最小表示
```

```
//flag=0,最大表示
int getmin_max(int flag){ //返回最小(大)表示的起始下标
    int len=strlen(s);
    int i=0,j=1,k=0; //j 始终在 i 后面
    while(j<len&&k<len){
        int ni=(i+k)%len,nj=(j+k)%len;
        if(flag){
            if(s[ni]>s[nj]) i=max(i+k+1,j),j=i+1,k=0;
            else if(s[ni]<s[nj]) j+=k+1,k=0;
            else k++;
        }
        else{
            if(s[ni]<s[nj]) i=max(i+k+1,j),j=i+1,k=0;
            else if(s[ni]>s[nj]) j+=k+1,k=0;
            else k++;
        }
    }
    return i;
}
```

2.8.2 Manacher

```
/*****manacher 算法*****/
/*****复杂度 O(n)*****/
/*
 *说明:
 *'#','*'为原串均没有出现过的字符
 *新串,用#分隔原串,并且 b[0]='*',防止越界,以'\0'结束
 *原串 i 对应新串 2*i,原串 i 与 i+1 之间对应新串 2*i+1
 *p[i]表示新串以 i 为中心包括 i 的最大回文半径
 *p[i]-1 表示原串以 i 为中心的最长回文长度
*/
char a[Maxn]; //原串,下标以 1 开始
char b[Maxn<<1]; //新串,用#分隔原串,并且 b[0]='*'
int p[Maxn<<1]; //最长回文半径
int md; //当前回文最大可到达的下标+1
int id; //md 对应的回文中心下标
void init(){
    int i=1;
    for(;a[i];i++){ //a[0]不存值
        b[i<<1]=a[i];
        b[i<<1|1]='#';
    }
```

```
    b[0]='*',b[1]='#',b[i<<1]='\0';
    md=id=0;
}
int manacher(){ //返回最大回文长度
    int len=0;
    for(int i=1;b[i];i++){
        /*
         *包含
         *考虑以 2*id-i 和 id 为中心的回文最左能到位置的关系
         *1.小于,应取 p[2*id-i],不可扩展
         *2.等于,应取 p[2*id-i],可扩展
         *3.大于,应取 maxid-i,不可扩展
        */
        if(md>i) p[i]=min(p[(id<<1)-i],md-i);
        else p[i]=1; //不包含
        while(b[i+p[i]]==b[i-p[i]]) p[i]++; //扩展
        if(i+p[i]>md){ //更新 md 和 id
            md=i+p[i];
            id=i;
        }
        len=max(len,p[i]);
    }
    return len-1;
}
```

2.8.3 KMP

```
/*****KMP*****/
/*****复杂度 O(n+m)*****/
//普通 KMP
char s[Maxn],o[Maxn];
int pre[Maxn];
//s 为模板串
void getFail(char *s){
    pre[0]=pre[1]=0;
    for(int i=1;s[i];i++){
        int j=pre[i];
        while(j&&s[j]!=s[i]) j=pre[j];
        pre[i+1]=s[i]==s[j]?j+1:0;
    }
}
//o 为目标串
void kmp(char *o,char *s){
```

```
    int m=strlen(s);
    int j=0;
    for(int i=0;o[i];i++){
        while(j&&s[j]!=o[i]) j=pre[j];
        if(s[j]==o[i]) j++;
        if(j==m){ //所有匹配位置
            printf("%d ",i+1-m);
            j=pre[j];
            continue;
        }
    }
}
```

2.8.4 后缀数组与 LCP

```
/******后缀数组(倍增算法)*****/
/******复杂度 O(nlogn)*****/
//注意数组前补 0,元素个数 n+1,数组后补 0,保证其余元素均大于 0
int r[Maxn],sa[Maxn],rk[Maxn],height[Maxn];
int wa[Maxn],wb[Maxn],rs[Maxn],wv[Maxn];
int cmp(int *r,int a,int b,int l){
    return r[a]==r[b]&&r[a+l]==r[b+l];
}
//r 为原数组,sa 为后缀数组,n 为元素个数+1,m 为数组最大值+1
//前面添 0,保证 sa 和 rank 从 1 开始,后面添 0 防止数组越界,并且比较不会相等
void da(int n,int m){
    int i,j,p,*x=wa,*y=wb;
    y[n]=0;  //防溢出
    //长度为 1 的基数排序
    for(i=0;i<m;i++) rs[i]=0;
    for(i=0;i<n;i++) rs[x[i]=r[i]]++;
    for(i=1;i<m;i++) rs[i]+=rs[i-1];
    for(i=n-1;i>=0;i--) sa[--rs[x[i]]]=i;
    //j 为后缀长度，合并为 2*j 的后缀,p=n 说明名次均不一样
    //m 为当前名次的最大值
    for(j=1,p=1;p<n;j<<=1,m=p){
        //第二关键字排序
        for(p=0,i=n-j;i<n;i++) y[p++]=i;
        for(i=0;i<n;i++) if(sa[i]>=j) y[p++]=sa[i]-j;
        //第一关键字排序
        for(i=0;i<m;i++) rs[i]=0;
        for(i=0;i<n;i++) rs[wv[i]=x[y[i]]]++;
```

```
        for(i=1;i<m;i++) rs[i]+=rs[i-1];
        for(i=n-1;i>=0;i--) sa[--rs[wv[i]]]=y[i];
        //合并，相同的情况,名次算一样的
        swap(x,y);
        for(p=1,x[sa[0]]=0,i=1;i<n;i++)
            x[sa[i]]=cmp(y,sa[i-1],sa[i],j)?p-1:p++;
    }
}
//height[i]=LCP(i-1,i)
void calheight(int n){
    int i,j,k=0;
    for(int i=1;i<n;i++) rk[sa[i]]=i;
    for(int i=1;i<n;height[rk[i++]]=k){
        //rank[i]=1,借助前补 0,j=0,r[i]!=r[0],height[1]=0
        //k=0,从首字母枚举,k>=1,从第 k 个字母开始枚举（k-1）
        //k=k==0||rank[i]==1?0:k-1;
        if(k) k--;
        for(j=sa[rk[i]-1];r[i+k]==r[j+k];k++);
    }
}

//m 范围极大时，采用归并排序
int r[Maxn],sa[Maxn],rk[Maxn],height[Maxn];
int wa[Maxn],wb[Maxn],rs[Maxn],wv[Maxn];

struct cmp_1{
    bool operator()(const int &a,const int &b)const{
        if(r[a]==r[b]) return a<b;
        return r[a]<r[b];
    }
};
struct cmp_2{
    bool operator()(const int &a,const int &b)const{
        if(wv[a]==wv[b]) return a<b;
        return wv[a]<wv[b];
    }
};
int cmp(int *r,int a,int b,int l){
    return r[a]==r[b]&&r[a+l]==r[b+l];
}
void da(int n,int m){
    int i,j,p,*x=wa,*y=wb;
    for(int i=0;i<n;i++) rs[i]=i,x[i]=r[i];
    sort(rs,rs+n,cmp_1());
```

```
    for(int i=1;i<n;i++) sa[i]=rs[i];
    for(j=1,p=1;p<n;j<<=1,m=p){
        for(p=0,i=n-j;i<n;i++) y[p++]=i;
        for(i=0;i<n;i++) if(sa[i]>=j) y[p++]=sa[i]-j;
        if(m>20000){
            for(int i=0;i<n;i++) rs[i]=i,wv[i]=x[y[i]];
            sort(rs,rs+n,cmp_2());
            for(int i=1;i<n;i++) sa[i]=y[rs[i]];
        }
        else{
            for(i=0;i<m;i++) rs[i]=0;
            for(i=0;i<n;i++) rs[wv[i]=x[y[i]]]++;
            for(i=1;i<m;i++) rs[i]+=rs[i-1];
            for(i=n-1;i>=0;i--) sa[--rs[wv[i]]]=y[i];
        }
        swap(x,y);
        for(p=1,x[sa[0]]=0,i=1;i<n;i++)
            x[sa[i]]=cmp(y,sa[i-1],sa[i],j)?p-1:p++;
    }
}
//最大跨度 1<<20
int p[Maxn],d[Maxn][20];
void rmq_init(int n){
    p[0]=-1;
    for(int i=1;i<=n;i++)
        p[i]=i&i-1?p[i-1]:p[i-1]+1;
    for(int i=1;i<=n;i++) d[i][0]=height[i];
    for(int j=1;j<=p[n];j++)
        for(int i=1;i+(1<<j)-1<=n;i++)
            d[i][j]=min(d[i][j-1],d[i+(1<<j-1)][j-1]);
}
int rmq_ask(int l,int r){
    int k=p[r-l+1];
    return min(d[l][k],d[r-(1<<k)+1][k]);
}
//lcp(suffix(a),suffix(b))=LCP(rank[a],rank[b])
int lcp(int a,int b){
    a=rk[a],b=rk[b];
    if(a<b) swap(a,b);
    return rmq_ask(b+1,a);
}
```

2.9 莫队

2.9.1 分块替代算法

```
/*****离线区间询问(不修改)*****/
/*****复杂度 O(n^1.5)*****/
/*
一、i 与 i+1 在同一块内，r 单调递增，所以 r 是 O(n)的。由于有 n^0.5 块,所以这一部分时间复杂度是 n^1.5。
二、i 与 i+1 跨越一块，r 最多变化 n，由于有 n^0.5 块，所以这一部分时间复杂度是 n^1.5
三、i 与 i+1 在同一块内时 l 变化不超过 n^0.5,跨越一块 l 也不会超过 2*n^0.5,不妨看作是 n^0.5。由于有 n 个数，所以时间复杂度是 n^1.5
*/

/***莫队算法***/
/***复杂度 O((n+Q)logn)***/
//询问和数组 a 的下标均从 0 开始,范围[0,n-1]
int pos[Maxn];
struct query{ //从 0 开始
    int l,r,id;
    //同一块内按 r 递增
    bool operator<(const query &a)const{
        return pos[l]<pos[a.l]||
        pos[l]==pos[a.l]&&r<a.r;
    }
}q[Maxn];
int qt; //询问个数
int n,m,sz;
ll ans;
int a[Maxn]; //a[i]表示第 i 个位置对应的颜色
int s[Maxn]; //每种颜色数量
void init(){
    sz=(int)sqrt(n); //块大小
    m=(n+sz-1)/sz; //块数
    for(int i=0;i<n;i++) pos[i]=i/sz;
    //读入原数组 a[]
    //读入询问
    sort(q,q+qt); //对询问排序
}
void cal(int x,int v){ //维护平方和
    ans-=(ll)s[a[x]]*s[a[x]]; //减去旧值
    s[a[x]]+=v; //修改
```

```
    ans+=(ll)s[a[x]]*s[a[x]]; //加上新值
}
void solve(){
    int l=0,r=-1; //区间[l,r]
    //l 往左,r 往右,先自增/减
    //l 往右,r 往左,后自增/减
    ans=0;
    for(int i=0;i<qt;i++){ //处理每个询问,[l,r]=>[l',r']
        //先处理右端点再处理左端点
        while(r<q[i].r) cal(++r,1);
        while(r>q[i].r) cal(r--,-1);
        while(l<q[i].l) cal(l++,-1);
        while(l>q[i].l) cal(--l,1);
        //其他处理
    }
}
/*****注意询问个数是 qt,不是 m*****/
```

第三章 计算几何

3.1 基本定义

```
#define type double/int //可选择
const double eps=1e-8;

int sgn(double x){ //double 形式在某些函数里必须使用
    if(fabs(x)<eps) return 0;
    if(x<0) return -1;
    return 1;
}
//在某些函数里这个函数充当符号,必须返回正负 1
int sgn(int x){
    if(!x) return 0;
    return x>0?1:-1;
}
struct vec{ //向量
    type x,y;
    vec(type xx=0,type yy=0):x(xx),y(yy){}
    bool operator==(vec a){
        return !sgn(x-a.x)&&!sgn(y-a.y);
    }
```

```
    vec operator*(type k){ //向量乘常数
        return vec(k*x,k*y);
    }
};
struct point{ //点
    type x,y;
    point(type xx=0,type yy=0):x(xx),y(yy){}
    vec operator-(point a){
        return vec(x-a.x,y-a.y);
    }
    point operator+(vec a){
        return point(x+a.x,y+a.y);
    }
    bool operator==(point a){
        return !sgn(x-a.x)&&!sgn(y-a.y);
    }
    bool operator<(const point &a)const{
        return sgn(x-a.x)<0||sgn(x-a.x)==0&&sgn(y-a.y)<0;
    }
};
```

3.1.1 读入

```
//读入一个点
point read_point(){
    point a;
    scanf("%lf%lf",&a.x,&a.y);
    return a;
}
```

3.1.2 基本运算

```
double len(vec a){ //矢量a的模长
    return sqrt(a.x*a.x+a.y*a.y);
}

type dotp(vec a,vec b){ //点积a·b
    return a.x*b.x+a.y*b.y;
}

type crossp(vec a,vec b){ //叉积a×b
    return a.x*b.y-b.x*a.y;
```

```
}
```

3.1.3 极角排序

```
bool cmp(vec &a,vec &b){
    if((ll)a.y*b.y<=0){
        if(a.y>0||b.y>0) return a.y<b.y;
        if(!a.y&&!b.y) return a.x<b.x;
    }
    return crossp(a,b)>0;
}
```

3.1.4 向量夹角

```
//两个向量的夹角[0,PI]
double angle(vec a,vec b){
    return acos(rectify(dotp(a,b)/len(a)/len(b)));
}

//矢量 a,b 的有向夹角(-PI,PI),(0,PI)是顺时针方向夹角,(-PI,0)是逆时针方向夹角
double ins_angle(vec a,vec b){ //acos 返回值是[0,PI]
    return sgn(crossp(a,b))*acos(rectify(dotp(a,b)/(len(a)*len(b))));
}

//矫正 acos 和 asin 参数,防止出现 nan
double rectify(double x){
    if(x>1) return 1;
    if(x<-1) return -1;
    return x;
}
```

3.1.5 矢量旋转

```
//将向量 a 逆时针旋转 r 弧度
vec rotate(vec a,double r){
    return vec(a.x*cos(r)-a.y*sin(r),a.x*sin(r)+a.y*cos(r));
}
```

3.2 点、线段、多边形关系

3.2.1 点在矩形内

```
//判断 c 是否在以线段 ab 为对角线的矩形内,包括矩形边界
bool inrect(point &a,point &b,point &c){
    return sgn(min(a.x,b.x)-c.x)<=0
        &&sgn(min(a.y,b.y)-c.y)<=0
        &&sgn(c.x-max(a.x,b.x))<=0
        &&sgn(c.y-max(a.y,b.y))<=0;
}
```

3.2.2 点在线段上

```
//判断 c 是否在线段 ab 上,包括 a,b 两点,矩形版
bool onseg(point a,point b,point c){
    return !sgn(crossp(b-a,c-a))&&inrect(a,b,c);
}

//判断 c 是否在线段 ab 上,包括 a,b 两点,点积版
bool onseg(point a,point b,point c){
    return !sgn(crossp(b-a,c-a))&&sgn(dotp(a-c,b-c))<=0;
}
//判断 c 是否在线段 ab 上,不包括 a,b 两点,点积版
bool onseg(point a,point b,point c){
    return !sgn(crossp(b-a,c-a))&&sgn(dotp(a-c,b-c))<0;
}
```

3.2.3 点在任意多边形内

```
//判断 p 在 n 边形(凹凸均适用)内,包括边界上
//poly 按逆时针存多边形的顶点
//角度和法
point poly[Maxn],p;
bool inpolygon(point &p,int n){
    poly[n]=poly[0];
    for(int i=0;i<n;i++) //待测点在边界上
        if(onseg(poly[i],poly[i+1],p)) return true;
    double sum=0;
    for(int i=1;i<=n;i++)
```

```
        sum+=ins_angle(poly[i-1]-p,poly[i]-p);
    return !sgn(fabs(sum)-2*PI);
    //return !sgn(sum-2*PI); //逆时针
    //return !sgn(sum+2*PI) //顺时针
}

//判断 p 在 n 边形(凹凸均适用)内,三种情况
//poly 按逆（顺）时针存多边形的顶点
//内角和的改进
int inpolygon(point &p,int n,point *poly){
    int wn=0; //绕数
    poly[n]=poly[0];
    for(int i=0;i<n;i++){
        if(onseg(poly[i],poly[i+1],p)) return -1; //在边界上
        int k=sgn(crossp(poly[i+1]-poly[i],p-poly[i]));
        int d1=sgn(poly[i].y-p.y);
        int d2=sgn(poly[i+1].y-p.y);
        if(k>0&&d1<=0&&d2>0) wn++; //从下往上穿过一次
        if(k<0&&d2<=0&&d1>0) wn--; //从上往下穿过一次
    }
    if(wn) return 1; //内部
    return 0; //外部
}

//判断 p 在 n 边形(凹凸均适用)内,包括边界上
//poly 按逆（顺）时针存多边形的顶点
//射线法
const double inf=1LL<<60; //足够大
bool inpolygon(point &p,int n){
    int cnt=0;
    point far=point(inf,p.y); //取水平无穷远点
    poly[n]=poly[0];
    for(int i=1;i<=n;i++){
        if(onseg(poly[i-1],poly[i],p)) //待测点在边界上
            return true;
        if(!sgn(poly[i-1].y-poly[i].y)) //平行
            continue;
        if(seg_ins(poly[i-1],poly[i],p,far)){ //线段相交
            point high; //高点
            if(poly[i-1].y>poly[i].y) high=poly[i-1];
            else high=poly[i];
            if(onseg(p,far,high)); //高点交不算
            else cnt++;
        }
```

```
    }
    return cnt&1;
}
```

3.2.4 线段与线段相交

```
//判断线段 ab,cd 是否不规范相交
bool seg_ins(point a,point b,point c,point d){ //使用 int 小心溢出
    return sgn(crossp(a-c,d-c)*crossp(b-c,d-c))<=0
        &&sgn(crossp(c-a,b-a)*crossp(d-a,b-a))<=0
        &&sgn(min(a.x,b.x)-max(c.x,d.x))<=0
        &&sgn(min(a.y,b.y)-max(c.y,d.y))<=0
        &&sgn(min(c.x,d.x)-max(a.x,b.x))<=0
        &&sgn(min(c.y,d.y)-max(a.y,b.y))<=0;
}

//判断线段 ab,cd 是否规范相交(两条线段恰有唯一一个不是端点的公共点)
bool seg_ins(point a,point b,point c,point d){
    return sgn(crossp(a-c,d-c)*crossp(b-c,d-c))<0
        &&sgn(crossp(c-a,b-a)*crossp(d-a,b-a))<0;
}
```

3.2.5 直线与直线的交点

```
//直线 ab 和直线 cd 求交点
//调用前保证 ab 和 cd 有交点,即 sgn(crossp(v,w))!=0
point line_ins(point a,point b,point c,point d){
    vec u=a-c,v=b-a,w=d-c;
    double t=crossp(w,u)/crossp(v,w);
    return a+v*t;
}

//直线 a+vt 与直线 b+wt 的交点
//调用前保证 ab 和 cd 有交点,即 sgn(crossp(v,w))!=0
point line_ins(point a,vec v,point c,vec w){
    vec u=a-c;
    double t=crossp(w,u)/crossp(v,w);
    return a+v*t;
}
```

3.2.6 多边形面积

```
//多边形(凹凸都适用)面积
//使用前请确保顶点已经按逆时针方向排好序
point p[Maxn]; //下标[0,n-1]
double polygonarea(int n){ //以 p[0]为参考
    double ans=0;
    for(int i=1;i<n-1;i++)
        ans+=crossp(p[i]-p[0],p[i+1]-p[0]);
    //return ans/2; //顺时针为负,需要取绝对值
    return fabs(ans/2);
}
```

3.3 凸包

```
/***ConvexHull***/
/***复杂度 O(nlogn)***/
point p[Maxn],ch[Maxn];
//原数组 p,下标[0,n-1]
//返回点存在 ch 中
//注意: n<3 返回 n,处理要小心
int Andrew(int n){ //返回凸包点数
    sort(p,p+n); //先 x 后 y
    int m=0;
    for(int i=0;i<n;i++){ //求下凸包
        //在右下
        while(m>1&&crossp(ch[m-2]-ch[m-1],ch[m-2]-p[i])<=0) m--;
        ch[m++]=p[i];
    }
    int k=m;
    for(int i=n-2;i>=0;i--){ //求上凸包
        //在右上
        while(m>k&&crossp(ch[m-2]-ch[m-1],ch[m-2]-p[i])<=0) m--;
        ch[m++]=p[i];
    }
    if(n>1) m--;
    return m;
}

/***Graham***/
/***复杂度 O(nlogn)***/
//按最左下角点 p[0]以逆时针方向极角序排列
```

```
//注意: n<3 返回 n,处理要小心
bool cmp(point a,point b){ //p[0]必须是最最左下角的点
    return sgn(crossp(a-p[0],b-p[0]))>0||
    sgn(crossp(a-p[0],b-p[0]))==0&&len(a-p[0])<len(b-p[0]);
}
int Graham(int n){ //返回凸包点数
    for(int i=1;i<n;i++) //得到最左下角的点
        if(p[i]<p[0]) swap(p[0],p[i]);
    sort(p+1,p+n,cmp);
    int m=0;
    for(int i=0;i<n;i++){
        //在右下方
        while(m>1&&crossp(ch[m-2]-ch[m-1],ch[m-2]-p[i])<=0) m--;
        ch[m++]=p[i];
    }
    return m;
}
```

3.3.1 旋转卡壳

```
/***旋转卡壳***/
/***复杂度 O(n)***/
int dis2(point a,point b){ //距离的平方
    return (a.x-b.x)*(a.x-b.x)+(a.y-b.y)*(a.y-b.y);
}
int rotate_caliper(point *ch,int n){ //旋转卡壳求最远点对
/*
 *当 Area(ch[i],ch[i+1],ch[j+1])<=Area(ch[i],ch[i+1],ch[j])时停止旋转
 *即 crossp(p[i+1]-p[i],p[j+1]-p[i])-crossp(p[i+1]-p[i],p[j]-p[i])<=0
 *根据 crossp(A,B)-crossp(A,C)=crossp(A,B-C)
 *化简得 crossp(p[i+1]-p[i],p[j+1]-p[j])<=0
*/
    ch[n]=ch[0];
    int ans=0,d;
    //枚举边 i->i+1,对踵点与该边所围三角形面积最大
    for(int i=0,j=1;i<n;i++){
        //面积增大一直旋转,直到面积减小,找到 u 的对踵点 v
        while((d=crossp(ch[i+1]-ch[i],ch[j+1]-ch[j]))>0) j=(j+1)%n;
        ans=max(ans,dis2(ch[i],ch[j])); //对踵点(i,j)
        //平行时,多了对踵点(i,j+1)
        if(!d) ans=max(ans,dis2(ch[i],ch[j+1]));
    }
    return ans;
```

```
}
```

第四章 数论

4.1 整数的因子分解

4.1.1 Miller_Rabin(大素数测试)

```
/*****Miller_Rabin*****/
/***复杂度 O(次数*(logn)^2)***/
int s=30; //测试次数
ll mul_mod(ll a,ll b,ll mod){ //a*b%mod
    a%=mod,b%=mod;
    ll res=0;
    while(b){
        if(b&1) res=(res+a)%mod;
        a=(a<<1)%mod; //a*2 不爆 ll
        b>>=1;
    }
    return res;
}
ll quick_pow(ll a,ll b,ll mod){ //a^b%mod
    ll res=1;
    a%=mod,b%=mod;
    while(b){
        if(b&1) res=mul_mod(res,a,mod);
        a=mul_mod(a,a,mod);
        b>>=1;
    }
    return res;
}
//以 a 为基,a^(n-1)=1 (mod n)
//n-1=2^t*u,u 为奇数,t>=1,n>=3
//返回 true,则 n 一定为合数
//返回 false 也可能是合数
bool check(ll a,ll n,ll u,ll t){
    ll res=quick_pow(a,u,n); //a^(n-1)=(a^u)^(2^t)
    ll last=res;
    for(int i=1;i<=t;i++){ //^(2^t)相等于 t 次平方
        res=mul_mod(res,res,n);
```

```
        //只有 n 为合数时,last^2=1 (mod n)
        //才可能存在非平凡平方根
        if(res==1&&last!=1&&last!=n-1)
            return true; //必然不是素数
        last=res; //设置新的 res
    }
    //费马小定理:当 n 为素数时,res 必然等于 1
    if(res!=1) return true; //必然不是素数
    return false;
}
//返回 false 必定是合数
//返回 true 是素数,但也有概率出错
//一般出错的概率为 2^(-s),s 为测试次数
bool Miller_Rabin(ll n){ //判断 n 是否为素数
    if(n<2) return false;
    if(n==2) return true;
    if((n&1)==0) return false; //偶数
    ll u=n-1,t=0;
    while((u&1)==0) //n-1=2^t*u
        u>>=1,t++;
    for(int i=0;i<s;i++){
        ll a=rand()%(n-1)+1; //随机[1,n-1]
        if(check(a,n,u,t)) //必然是合数
            return false;
    }
    return true;
}
```

4.1.2 Prollard_Rho(质因数分解)

```
/*****Prollard_Rho*****/
/*****复杂度 O(n^0.25*logn)*****/
ll gcd(ll a,ll b){
    return b==0?a:gcd(b,a%b);
}
//返回的可能是 p 的倍数,p|n
//切记 n=1,会死循环
ll Prollard_Rho(ll n,ll c){ //找出 n 的因子
    //cout<<n<<endl;
    ll i=1,k=2,x,y,p,d;
    y=x=rand()%n;
    while(true){
        i++;
```

```
        //c 不要取 0 和-2
        x=(mul_mod(x,x,n)+c)%n; //迭代函数 f(x)=x*x+c
        if(x==y) return n; //迭代出现循环
        p=y>x?y-x:x-y;
        d=gcd(p,n);
        if(d!=1&&d!=n) //找到 n 的非平凡因子
            return d;
        if(i==k){ //k 下标序列 1,2,4,...
            y=x; //y 存下标为 2 的幂的 x
            k<<=1;
        }
    }
}
ll f[110]; //这里存的是 n 的所有质因数(无序)
int ft; //初始化为 0
void factorize(ll n){ //将 n 分解因式
    if(Miller_Rabin(n)){ //素数
        f[ft++]=n;
        return;
    }
    ll p=n;
    while(p>=n) //p 必然是合数,多次分解直到找到 n 的非平凡因子
        p=Prollard_Rho(p,rand()%(n-1)+1); //c 必须随机,否则死循环
    //找到因子 p,递归分解
    factorize(p);
    factorize(n/p);
}
```

4.1.3 扩展欧几里得

```
/****Extend Euclid****/
/****复杂度 O(log a)****/
/****计算 ax+by=gcd (x,y)****/
#define ll long long

//a,b 可为负数
//递归算法, |q|=gcd(|a|,|b|);
void extend_gcd(ll a,ll b,ll &x,ll &y,ll &q){
    if(b==0){x=1;y=0;q=a;}
    else{
        extend_gcd(b,a%b,x,y,q);
        ll tmp=x;x=y;y=tmp-a/b*y;
    }
```

```
}
```

说明：$\gcd(a,b)=\gcd(b,a\%b)$，则$ax_1+by_1=bx_2+\left(a-\frac{a}{b}*b\right)y_2$

$$=ay_2+b\left(x_2-\frac{a}{b}*y_2\right)$$

有恒等关系：$x_1=y_2, y_1=x_2-\frac{a}{b}*y_2$

```
//非递归算法
ll extend_gcd(ll a,ll b,ll &x,ll &y){
    ll x1,y1,x0,y0,r,q;
    x0=y1=1,y0=x1=0;
    r=a%b; //余数
    q=(a-r)/b; //商
    x=0,y=1; //r=0,gcd(a,b)=0*a+1*b;
    while(r){
        x=x0-q*x1,y=y0-q*y1;
        x0=x1,y0=y1;
        x1=x,y1=y;
        a=b,b=r,r=a%b;
        q=(a-r)/b;
    }
    return b; //|b|=gcd(|a|,|b|);
}
```

说明：$r_0=a-q_0b, r_1=b-q_1r_0=-q_1a+(1+q_0q_1)b$

$r_i=r_{i-2}-q_ir_{i-1}=(x_{i-2}a+y_{i-2}b)-q_i(x_{i-1}a+y_{i-1}b)=$

$(x_{i-2}-q_ix_{i-1})a+(y_{i-2}-q_iy_{i-1})\mathrm{b}$

有递推式：$x_0=1, y_1=-q_0, x_2=-q_1, y_1=1+q_0q_1$

$$x_i=x_{i-2}-q_ix_{i-1}, y_i=y_{i-2}-q_iy_{i-1}$$

4.2 乘法逆元

```
/*
a*x≡1(mod p),称 x 是 a 关于 p 的乘法逆元
条件:a 和 p 要互质,因为 a*x≡1(mod p)等价于 ax+py=1,而方程 ax+py=1 有解的充要条件是 gcd(a,p)=1
性质 1:求 b/a (mod p),前提条件是 a|b,否则分数模上一个数没有意义,其次 gcd(a,p)=1,否则 a 关于 p 不存在乘法逆元,满足上述两个条件,即有 a*x≡1(mod p),设 b/a≡t(mod p)两式相乘得:(b/a)*(ax)≡t*1(mod p),推出 b*x≡1(mod p)
```

```
由此可见,乘法逆元可以将除法转乘法(必须满足上述两个条件)
性质 2:inv(a*b)=inv(a)*inv(b) //完全奇性函数
性质 3:a 关于 p 的逆元等于 a%p 关于 p 的逆元
*/
```

4.2.1 费马小定理(阶乘逆元)

$$a^{p-1} \equiv 1 \pmod{p}, \gcd(a,p) = 1 \text{ 且 p 为质数}$$

```
const int n=1000000; //最大上限
char s[Maxn];
const int mod=1000000007;
ll fac[Maxn],inv[Maxn];
ll mypow(ll a,int b){ //快速幂
    ll ans=1;
    while(b){
        if(b&1) ans=ans*a%mod;
        a=a*a%mod;
        b>>=1;
    }
    return ans;
}
fac[0]=1; //0!为 1
for(int i=1;i<=n;i++) //i!
    fac[i]=fac[i-1]*i%mod;
/*
inv(n!)≡inv((n-1)!)*inv(n) (mod p)
inv(n!)*n≡inv((n-1)!)*inv(n)*n (mod p)
inv(n)*n≡1 (mod p)
inv((n-1)!)≡inv(n!)*n (mod p)
*/
inv[n]=mypow(fac[n],mod-2); //利用费马小定理求 n!的逆元
for(int i=n;i>0;i--) //i!的逆元
    inv[i-1]=inv[i]*i%mod;
```

4.2.2 Lucas 定理

```
/***Lucas 定理***/
/***复杂度 O(log(mod))***/
ll Lucas(ll a,ll b){ //必须保证 a>=b
    if(b==0) return 1;
```

```
    ll ans=1,c=a%mod,d=b%mod;
    if(c<d) return 0;
    ll tmp=fac[c]*mypow(fac[d]*fac[c-d]%mod,mod-2)%mod;
    return (Lucas(a/mod,b/mod)*tmp)%mod;
}
```

4.2.3 扩展欧几里得

```
const int n=1000000; //最大上限
const int mod=1000000007;
ll fac[Maxn],inv[Maxn];
void extend_gcd(ll a,ll b,ll &x,ll &y,ll &q){
    if(b==0){x=1;y=0;q=a;}
    else{
        extend_gcd(b,a%b,x,y,q);
        ll tmp=x;x=y;y=tmp-a/b*y;
    }
}
fac[0]=1; //0!为1
for(int i=1;i<=n;i++) //n!
    fac[i]=fac[i-1]*i%mod;
ll x,y,q;
//对a求关于p的逆元,有ax+py=1,请调用extend_gcd(a,b,x,y,q)
//x为a关于p的逆元,x=(x+mod)%mod确保x>0
extend_gcd(fac[n],mod,inv[n],y,q); //n!的逆元
inv[n]=(inv[n]+mod)%mod; //确保inv[n]>0
for(int i=n;i>0;i--) //i!的逆元
    inv[i-1]=inv[i]*i%mod;
```

4.3 高斯消元

4.3.1 行列式的值

```
/*****高斯消元*****/
/*****复杂度O(n^3)*****/
#define ld long double
#define eps 1e-8
ld a[Maxn][Maxn];
int sign(ld x){
    if(fabs(x)<eps) return 0;
```

```
    return x<0?-1:1;
}
//执行完 Det,数组 a 会改变,但不是上三角行列式
ld Det(int n){ //n=0,返回 1
    int i,j,k,r=1;
    ld res=1;
    for(i=1;i<=n;i++){
        if(!sign(a[i][i])){
            for(j=i+1;j<=n;j++)
                if(sign(a[j][i])) break;
            if(j>n) return 0;
            for(k=i;k<=n;k++) //交换 i,j 行
                swap(a[i][k],a[j][k]);
            r=-r; //交换奇数次行列式值变负
        }
        res*=a[i][i];
        for(j=i+1;j<=n;j++) //每行消元的系数
            a[j][i]/=a[i][i]; //a[j][i]非 0,实际应消为 0,
                              但不影响求行列式的值
        for(j=i+1;j<=n;j++) //(j,k)-(i,k)*系数[存在(k,i)]
            for(k=i+1;k<=n;k++)
                a[j][k]-=a[i][k]*a[j][i];
    }
    return r*res;
}

/*****高斯消元对合数取模*****/
/*****复杂度 O(n^3)*****/
ll Det(int n){ //n=0,返回 1
    int i,j,k,r=1;
    ll res=1;
    for(i=1;i<=n;i++) //保证 0<=a[i][j]<mod
        for(j=1;j<=n;j++){
            a[i][j]%=mod;
            if(a[i][j]<0) a[i][j]+=mod;
        }
    for(i=1;i<=n;i++){
        if(!a[i][i]){
            for(j=i+1;j<=n;j++)
                if(a[j][i]) break;
            if(j>n) return 0;
            for(k=i;k<=n;k++) //交换 i,j 行
                swap(a[i][k],a[j][k]);
            r=-r; //交换奇数次行列式值变负
```

```
        }
        for(j=i+1;j<=n;j++){
            while(a[j][i]){ //类似辗转相除法
                for(k=i;k<=n;k++)
                    swap(a[i][k],a[j][k]);
                r=-r;
                ll tmp=a[j][i]/a[i][i]; //第 j 行-第 i 行*tmp
                for(k=i;k<=n;k++){
                    a[j][k]=(a[j][k]-a[i][k]*tmp%mod)%mod;
                    if(a[j][k]<0) a[j][k]+=mod;
                }
            }
        }
        res=(res*a[i][i])%mod;
    }
    if(r<0) res=mod-res;
    return res;
}
```

4.3.2 求解线性方程组

```
#define eps 1e-6
double a[Maxn][Maxn];
int sign(double x){
    if(fabs(x)<eps) return 0;
    return x>0?1:-1;
}
//将系数矩阵消元至行最简矩阵
bool gauss(int n){ //增广矩阵为 n 行 n+1 列
    int cur=1,i,j,k;
    for(int i=1;i<=n;i++){ //枚举列
        if(!sign(a[cur][i])){
            for(j=cur+1;j<=n;j++)
                if(a[j][i]) break;
            if(j>n) continue; //继续下一列
            for(k=i;k<=n+1;k++)
                swap(a[cur][k],a[j][k]);
        }
        double t=a[cur][i];
        for(j=1;j<=n+1;j++)
            a[cur][j]/=t;
        for(j=1;j<=n;j++){
            if(cur==j) continue;
```

```
            t=a[j][i];
            for(k=1;k<=n+1;k++)
                a[j][k]-=t*a[cur][k];
        }
        //注意 BUG:若列数-行数>1 的情况,则 cur 会大于行数,
        //此时不 break,则可能使用前一组数据的下一行,导致出错
        //if(++cur>n) break;
        cur++; //这里列数-行数=1,因此当 cur=n+1 时,i=n,则 i++后跳出循环
    }
    for(i=cur;i<=n;i++) //最底下系数全部消为 0
        if(fabs(a[i][n+1])>eps) return false;
    return true; //有解(包括有自由变元)
}

/***模 2 线性方程组***/
int a[Maxn][Maxn];
//模 2 线性方程组<==>异或方程组
void gauss(int n){ //增广矩阵为 n 行 n+1 列
    int i,j,k;
    for(i=1;i<=n;i++){ //枚举主元(i,i)
        if(!a[i][i]){
            for(j=i+1;j<=n;j++)
                if(a[j][i]) break;
            if(j>n) continue;
            for(k=1;k<=n+1;k++)
                swap(a[i][k],a[j][k]);
        }
        for(j=i+1;j<=n;j++){
            if(!a[j][i]) continue;
            for(k=i;k<=n+1;k++) //异或消元
                a[j][k]^=a[i][k];
        }
    }
}
int ans[Maxn]; //存一组解
int res; //初始化为 inf,所有解的 1 的个数最小值
int tot; //初始化为 0,当前解的 1 的个数
bool flag; //初始化为 false
void dfs(int dep,int n){ //调用 dfs(n,n),倒着求解
    if(tot>=res) return; //剪枝
    if(!dep){
        flag=true; //有解
        res=min(res,tot);
        return;
```

```
    }
    int x=a[dep][n+1];
    for(int i=dep+1;i<=n;i++)
        x^=a[dep][i]&&ans[i];
    if(a[dep][dep]){
        ans[dep]=x;
        tot+=x;
        dfs(dep-1,n);
        tot-=x;
    }
    else if(!x){
        ans[dep]=1;
        tot++;
        dfs(dep-1,n);
        tot--;
        ans[dep]=0;
        dfs(dep-1,n);
    }
}

/***高斯消元求线性基***/
int gauss(int n){ //返回矩阵的秩,即线性基的个数
    int cur=1,i,j;
    for(i=60;i>=0;i--){
        if(!(a[cur]>>i&1)){
            for(j=cur+1;j<=n;j++)
                if(a[j]>>i&1) break;
            if(j>n) continue;
            swap(a[cur],a[j]);
        }
        for(j=1;j<=n;j++){
            if(j==cur||!(a[j]>>i&1)) continue;
            a[j]^=a[cur];
        }
        if(++cur>n) break;
    }
    return cur-1; //a[1...cur-1]存了线性基
}
```

4.4 同余式

4.4.1 解模线性方程组

```
ll m[Maxn],r[Maxn]; //x≡r(mod m)
ll abs(ll x){
    return x>0?x:-x;
}
void extend_gcd(ll a,ll b,ll &x,ll &y,ll &q){
    if(b==0){x=1;y=0;q=a;}
    else{
        extend_gcd(b,a%b,x,y,q);
        ll tmp=x;x=y;y=tmp-a/b*y;
    }
}
//解两个方程
ll solve_modequation(ll m1,ll r1,ll m2,ll r2,ll &q){
    ll x,y;
    extend_gcd(m1,-m2,x,y,q);
    if((r2-r1)%q) return -1; //无解
    ll ans=(r2-r1)/q*x%(m2/q); //这里 q 还为 gcd，防止爆 ll
    ans=m1*ans+r1;
    q=abs(m1/q*m2); //修正为 lcm
    return (ans%q+q)%q; //返回最小非负解
}
//通解 x=x'(返回值)+k*q,即 x≡x'(mod q)

//解两个以上方程
ll solve_equation(int n){
    m[0]=1,r[0]=0;
    for(int i=1;i<=n;i++){
        r[0]=solve_modequation(m[0],r[0],m[i],r[i],m[0]);
        if(r[0]==-1) return -1; //无解
    }
    //此时 m[0]=lcm(m1,m2,...,mn)
    return r[0]; //返回最小非负解
}
```

4.5 素数

4.5.1 埃拉托斯特尼筛法

```
/***埃拉托斯特尼筛法***/
/*****复杂度 O（nloglogn）*****/
#define Maxn 1000010
#define ll long long
int c[Maxn],prime[Maxn];
int tot=0;
void Eratosthenes(int n){ //筛出[1,n]之间的素数
    memset(c,0,sizeof c);
    for(ll i=2;i<=n;i++){
        if(!c[i]){
            prime[tot++]=i;
            for(ll j=i*i;j<=n;j+=i)
                c[j]=1;
        }
    }
}

/***质因数分解***/
vector<int> f[Maxn];
void cal(int x,int y){
    int tmp=x;
    while(tmp%y==0){
        f[x].push_back(y);
        tmp/=y;
    }
}
void Eratosthenes(int n){
    for(int i=1;i<=n;i++) f[i].clear();
    for(int i=2;i<=n;i++){
        if(f[i].size()) continue;
        f[i].push_back(i);
        for(int j=i+i;j<=n;j+=i)
            f[j].push_back(i); //每个素因子只出现一次
            //cal(j,i); //素因子有次数
    }
}
```

4.5.2 欧拉筛法

```
/***欧拉筛法***/
/*****复杂度 O(n)*****/
#define Maxn 1000010
#define ll long long
int c[Maxn],prime[Maxn];
int tot;
void Euler(int n){ //筛出[1,n]之间的素数
    memset(c,0,sizeof c);
    tot=0;
    for(int i=2;i<=n;i++){
        if(!c[i]) prime[tot++]=i;
        for(int j=0;j<tot;j++){
            if(i*prime[j]>n) break;
            c[i*prime[j]]=1;
            if(i%prime[j]==0) break; //保证被最小质因数筛除
        }
    }
}
```

说明:设合数 n 的最小质因数为 p,它的另一大于 p 的质因数为p',令 n = pm = p′m′;考虑p′ > p，那么m′ < m;由于 gcd(p,p') = 1,则 p|m';当 i 循环到m'时,j 循环到 p 就会跳出循环,因此p′无法筛除 n

```
/***质因数分解***/
int prime[Maxn];
int c[Maxn];
int tot;
vector<int> f[Maxn];
int cnt[Maxn]; //质因子个数
void euler(int n){
    memset(c,0,sizeof c);
    for(int i=1;i<=n;i++) f[i].clear();
    tot=0;
    for(int i=2;i<=n;i++){
        if(!c[i]){
            prime[tot++]=i;
            cnt[i]=1;
            f[i].push_back(i);
        }
```

```
        for(int j=0;j<tot;j++){
            if(i*prime[j]>n) break;
            c[i*prime[j]]=1;
            /******质因子有重复******/
            cnt[i*prime[j]]=cnt[i]+1;
            f[i*prime[j]]=f[i];
            f[i*prime[j]].push_back(prime[j]);
            if(i%prime[j]==0) break;
            /******质因子有重复******/

            /******质因子无重复******/
            /* 质因子无重复
            if(i%prime[j]==0){
                f[i*prime[j]]=f[i];
                cnt[i*prime[j]]=cnt[i];
                break;
            }
            cnt[i*prime[j]]=cnt[i]+1;
            f[i*prime[j]]=f[i];
            f[i*prime[j]].push_back(prime[j]);
            */
            /******质因子无重复******/
        }
    }
}
```

4.6 一些重要积性函数

$f(x)$为积性函数，且 $m = m_1 \cdot m_2$，且$m_1 \perp m_2$，

则$f(m) = f(m_1) \cdot f(m_2)$

性质：$f(1) = 1$;

积性函数的前缀和也是积性函数;

4.6.1 莫比乌斯函数

```
/***欧拉筛筛 mobius 函数***/
/***复杂度 O(n)***/
int c[Maxn],prime[Maxn];
```

```
int mu[Maxn];
int tot;
void mobius(int n){
    memset(c,0,sizeof c);
    mu[1]=1;
    tot=0;
    for(int i=2;i<=n;i++){ //i 循环结束,mu[i]必然已赋值
        if(!c[i]){ //质数
            prime[tot++]=i;
            mu[i]=-1;
        }
        for(int j=0;j<tot;j++){
            if(i*prime[j]>n) break;
            c[i*prime[j]]=1;
            if(i%prime[j]==0){
                mu[i*prime[j]]=0; //gcd(i,prime[j])!=1
                break;
            }
            //i%prime[j]!=0,即 gcd(i,prime[j])=1
            mu[i*prime[j]]=-mu[i];
        }
    }
}

/***分块处理 f(i)*(n/i)*(m/i)***/
ll cal(int n,int m){
    if(n>m) swap(n,m);
    int nxt;
    ll res=0;
    for(int i=1;i<=n;i=nxt+1){
        nxt=min(n/(n/i),m/(m/i));
        res+=(ll)(n/i)*(m/i)*(s[nxt]-s[i-1]);
    }
    return res;
}
```

4.6.2 欧拉函数

```
/***phi(n)***/
/***复杂度 O(sqrt(n))***/
int phi(int n){
    int ans=n;
    for(int i=2;i*i<=n;i++){
```

```
        if(n%i==0){
            ans=ans/i*(i-1);
            while(n%i==0) n/=i;
        }
    }
    if(n>1) ans=ans/n*(n-1);
    return ans;
}
/***欧拉筛筛 phi 函数***/
/***复杂度 O(n)***/
int c[Maxn],prime[Maxn];
int phi[Maxn];
int tot;
void Phi(int n){
    memset(c,0,sizeof c);
    phi[1]=1;
        tot=0;
    for(int i=2;i<=n;i++){ //i 循环结束,phi[i]必然已赋值
        if(!c[i]){ //质数
            prime[tot++]=i;
            phi[i]=i-1;
        }
        for(int j=0;j<tot;j++){
            if(i*prime[j]>n) break;
            c[i*prime[j]]=1;
            if(i%prime[j]==0){
                //gcd(i,prime[j])!=1
                phi[i*prime[j]]=phi[i]*prime[j];
                break;
            }
            //i%prime[j]!=0,即 gcd(i,prime[j])=1
            phi[i*prime[j]]=phi[i]*(prime[j]-1);
        }
    }
}
```

4.6.3 逆元函数

$$\text{设大质数} p = \mathrm{nt} + \mathrm{k}, \text{则 } \mathrm{nt} \equiv -\mathrm{k} \pmod{p}$$

$$\Longrightarrow ntf(k) \equiv -1 \pmod{p}$$

$$\Longrightarrow n(-t)f(k) \equiv 1 \pmod{p}$$

$$\Longrightarrow n(p-t)f(k) \equiv 1 \pmod{p}$$

$$\Longrightarrow n^{-1} \equiv (p-t)f(k) \pmod{p}$$

```
/***递推求 inverse 函数***/
/***复杂度 O(n)***/
const int mod=1e9+7;
ll inv[Maxn];
void inverse(int n){
    inv[1]=1;
    for(int i=2;i<=n;i++)
        inv[i]=(mod-mod/i)*inv[mod%i]%mod;
}
```

4.6.4 约数函数(约数和/约数个数)

```
/***欧拉筛筛约数函数***/
/***复杂度 O(n)***/
int f[Maxn]; //f[i]表示 i 的所有约数之和,注意爆 int
int g[Maxn]; //g[i]表示 i 的约数个数
int div[Maxn]; //div[i]表示 i 的素数分解中最小质数的乘积
int c[Maxn],prime[Maxn];
int tot;
void divisor(int n){
    memset(c,0,sizeof c);
    f[1]=g[1]=1;
    tot=0;
    for(int i=2;i<=n;i++){
        if(!c[i]){ //质数只有 1 和 i 因子
            prime[tot++]=i;
            f[i]=i+1,g[i]=2;
            div[i]=i;
        }
        for(int j=0;j<tot;j++){
            int t=i*prime[j];
            if(t>n) break;
            c[t]=1;
            if(i%prime[j]==0){ //gcd(i,prime[j])!=1
                div[t]=div[i]*prime[j];
                if(t/div[t]==1){
                    //t=prime[j]^k,约数为 1,prime[j],...,prime[j]^k
                    int num=floor(log(t)/log(prime[j])+0.5);
```

```
                f[t]=(t*prime[j]-1)/(prime[j]-1);
                g[t]=num+1;
            }
            else{
                f[t]=f[t/div[t]]*f[div[t]];
                g[t]=g[t/div[t]]*g[div[t]];
            }
            break;
        }
        f[t]=f[i]*f[prime[j]];
        g[t]=g[i]*g[prime[j]];
        div[t]=prime[j];
    }
  }
}
/***枚举因子更新***/
/***复杂度 O(nlogn)***/
for(int i=1;i<=n;i++)
    for(int j=i;j<=n;j+=i)
        f[j]+=i,g[j]++;
```

4.7 离散对数

4.7.1 Baby step gaint step

```
/***baby step gaint step***/
/***O(sqrt(p))***/
ll pw(ll a,int b,int p){
    ll ans=1;
    while(b){
        if(b&1) ans=ans*a%p;
        a=a*a%p;
        b>>=1;
    }
    return ans;
}
ll gcd(ll a,ll b){
    return b==0?a:gcd(b,a%b);
}
//求 a^x=b (mod p)的最非负整数解,无解返回-1
int bsgs(ll a,ll b,int p){
    a%=p,b%=p;
    int c=0;
```

```
    ll t=1;
    //a^x=b (mod p) <=> (a/g)*a^(x-1)=b/g (mod p/g)
    for(ll g=gcd(a,p);g!=1;g=gcd(a,p)){
        if(b==t) return c; //只有一个解的情况
        /*
            b*a^x=t (mod p),b=t&&(b,p)=1 ==>a^x=1 (mod p)
            g=(a,p)!=1 ==> x=0
        */
        if(b%g) return -1;
        p/=g,b/=g,t=t*a/g%p;
        c++; //每次解少1,最后须累加这个值
    }
    //以下是(a,p)=1的情况,存在多解
    if(b==t) return c; //(a,p)=1有多解
    map<ll,ll> ha; //map慢使用hash表
    int m=int(sqrt(p*1.0)+1); //设x=km-r,(k>0)
    ll base=b;
    for(int i=0;i<=m;i++){ //b*a^r,r={0,1,...,m-1}
        ha[base]=i; //相同base值,hash值要取大
        base=base*a%p;
    }
    base=pw(a,m,p); //a^m%p
    ll cur=t;
    //t*a^(km)=b*a^r,r={0,1,...,m-1} (mod p)
    //x=km-r=>km=x+r<=p+m=>k<=p/m+1
    for(int k=1;k<=m+1;k++){ //取上界m+1
        cur=cur*base%p;
        if(ha.count(cur)) //(a,p)=1有多解
            return k*m-ha[cur]+c;
    }
    return -1;
}
```

4.8 其他

4.8.1 快速求前缀和

$$令F_n=\sum_{i=1}^{n} f_i, G_n=\sum_{i|n} f_i$$

$$\sum_{i=1}^{n} G_i = \sum_{i=1}^{n} f_i \cdot \left\lfloor \frac{n}{i} \right\rfloor = \sum_{i=1}^{n} F_{\left\lfloor \frac{n}{i} \right\rfloor}$$

$[1,n]$中 i 的倍数恰好为 $\left\lfloor \frac{n}{i} \right\rfloor$ 个,所以f_i出现 $\left\lfloor \frac{n}{i} \right\rfloor$ 次,第一个等号成立

$\left\lfloor \frac{n}{i} \right\rfloor$ 是满足 $i \le \left\lfloor \frac{n}{t} \right\rfloor$ 且 $i > \left\lfloor \frac{n}{t+1} \right\rfloor$ 的最大整数 t, 第二个等号成立

$$F_n = \sum_{i=1}^{n} G_i - \sum_{i=2}^{n} F_{\left\lfloor \frac{n}{i} \right\rfloor}$$

如果可以快速求出 $\sum_{i=1}^{n} G_i$,比如莫比乌斯函数或者欧拉函数,

那么对于

$k \le n^{\frac{2}{3}}$,暴力计算F_k(考虑容斥或者莫比乌斯反演),

否则递归计算,需记忆化,复杂度 $O\left(n^{\frac{2}{3}}\right)$

```
int f[Maxn];
int F[Maxn]; //f[]的前缀和
void init(int n){ //n<=1000000 直接算
    for(int i=1;i<=n;i++){ //f[i]存除了 i 之外因子的贡献值
        f[i]=(g(i)-f[i])%mod; //获得真正的 f[i],g(i)=sigma[d|i](f[d])
        if(f[i]<0) f[i]+=mod;
        for(int j=i+i;j<=n;j+=i) //对除了自身的倍数做贡献
            f[j]=(f[j]+f[i])%mod;
        F[i]=(F[i-1]+f[i])%mod;
    }
}
map<int,ll> mp;
ll cal(int n){
    if(n<=1000000) return F[n];
    if(mp.find(n)!=mp.end()) return mp[n];
    ll tmp=G(n); //G(n)=sigma[1…n]g(i)
    for(int L=2,R;L<=n;L=R+1){ //x 属于[L,R],n/x 均相同
        R=min(n,n/(n/L));
        tmp=(tmp-(R-L+1)*cal(n/L)%mod)%mod;
    }
    if(tmp<0) tmp+=mod;
```

```
    return mp[n]=tmp;
}
```

第五章 组合数学

5.1 全排列

```
/*****全排列*****/
/*****复杂度 O(n!)*****/
//递归
int a[Maxn];
for(int i=1;i<=n;i++) a[i]=i;
void perm(int k,int n){
    if(k==n){   //产生的全排列
        for(int i=1;i<=n;i++)
            printf("%d ",a[i]);
        puts("");
    }
    else{
        for(int i=k;i<=n;i++){
            swap(a[k],a[i]);
            perm(k+1,n);
            swap(a[i],a[k]);
        }
    }
}
//非递归
int a[Maxn],x[Maxn]; //模拟栈
void perm(int n){
    int k=1;
    x[1]=0;
    while(k>0){
        x[k]++; //模拟递归 for 循环
        if(x[k]>n){
            k--; //深一层退栈
            swap(a[k],a[x[k]]); //模拟 for 里的第二个 swap
        }
        else{
            swap(a[k],a[x[k]]); //模拟 for 里的第一个 swap
            if(k==n){ //得到全排列,做相应处理
```

```
                for(int i=1;i<=n;i++)
                    printf("%d ",a[i]);
                puts("");
            }
            else{
                k++; //深一层
                x[k]=k-1; //初始值
            }
        }
    }
}
```

5.2 组合数

```
//计算 C(m,n)%mod,fac 和 inv 见费马小定理求乘法逆元
ll C(int m,int n){ //n>=m>=0,C(0,n)=1
    return fac[n]*inv[m]%mod*inv[n-m]%mod;
}

//递推求 c[i][j],0<=j<=i<=n
int c[Maxn][Maxn];
const int mod=1000007;
void init(int n){
    for(int i=0;i<=n;i++){
        c[i][0]=c[i][i]=1;
        for(int j=1;j<i;j++)
            c[i][j]=(c[i-1][j]+c[i-1][j-1])%mod;
    }
}
```

5.3 置换

```
/*
性质 1:对于长度为 n 的置换 T,T^k 的循环个数为 gcd(n,k)

因为当 k|n 时,T 可以分解成 k 个循环,每个循环都是 k 的同余类
当 gcd(n,k)=1 时,T 不能分解,因为 k*s%n=k*t%n 等价于 n|k(s-t),由此可以肯定
n|(s-t),在 s!=t 的情况下,s=n*h+t>n,也就是要形成循环必须跳 n 步以上,也就
是循环的长度至少为 n,而最大也只能为 n,因此不能分解
其他情况 k=gcd(n,k)*(k/gcd(n,k)),等价计算(T^gcd(n,k))^(k/gcd(n,k)),由
于(k/gcd(n,k))|n,所以 T^gcd(n,k)分解成 gcd(n,k)个循环,并且每个循环的长
度为 n/gcd(n,k),计算(T^gcd(n,k))^(k/gcd(n,k))时可以独立计算 T^gcd(n,k)
```

的每个循环的k/gcd(n,k)次幂,由于gcd(k/gcd(n,k), n/gcd(n,k))=1,那么每个循环均不能分解,最终就是gcd(n,k)个循环

性质2: 对于长度为n的置换T,T^k为恒等置换的最小正整数k为n
因为T错位为1,T^2错位为2,……,T^(n-1)错位为n-1,T^n错位为0
推论: 对于含有若干循环的置换T, T^k为恒等置换的最小正整数k为所有循环的长度的最小公倍数

*开k次方:设有长度为L的循环,寻找gcd(m*L,k)=m的最小正整数,这里m必然是gcd(k,l)的倍数,如果存在这样的m,那么将m个长度为L的循环合并*
/

5.3.1 置换乘法和快速幂

```
int n; //置换长度
struct Permutation{ //[0,n-1]
    int perm[Maxn];
    Permutation(){
        for(int i=0;i<n;i++)
            perm[i]=i;
    }
    Permutation operator*(Permutation p){
        Permutation ans;
        for(int i=0;i<n;i++)
            ans.perm[i]=p.perm[perm[i]];
        return ans;
    }
}p;
Permutation quick_pow(Permutation p,int k){ //p^k,注意p被改变
    Permutation ans;
    while(k){
        if(k&1) ans=ans*p;
        p=p*p;
        k>>=1;
    }
    return ans;
}
```

5.3.2 单循环开方(互质)

```
int init[Maxn],perm[Maxn]; //[1,n]
perm[1]=1;
int k=1,id=1;
```

```
while(init[k]!=1){ //将 init 转换成循环形式 perm
    k=init[k];
    perm[++id]=k;
}
k=1;
for(int i=1;i<=n;i++){ //开 t 次方,结果存在 init 里
    init[k]=perm[i];
    k=(k+t-1)%n+1;  //向前 t 步,范围[1,n]
}
for(int i=1;i<n;i++) //将循环形式转成置换,结果存在 perm 里
    perm[init[i]]=init[i+1];
perm[init[n]]=init[1];
```

5.3.3 Polya 定理

```
int vis[Maxn];
int perm[Maxn]; //置换
int polya(int n){ //返回循环节数
    int ans=0;
    memset(vis,0,sizeof vis);
    for(int i=0;i<n;i++){
        if(!vis[i]){
            int j=i;
            while(!vis[j]){
                vis[j]=1;
                j=perm[j];
            }
            ans++;
        }
    }
    return ans;
}
```

5.4 母函数

```
/*****母函数*****/
/***复杂度 O(n*m*logm)***/
//Maxm 表示最大价值,Maxn 表示物品种类数
int ans[Maxm]; //答案
int tmp[Maxm]; //临时答案
int a[Maxn]; //物品的价值
```

```
int b[Maxn]; //物品的个数
void gene_fun(int n,int m){ //n 种物品,最大价值 m
    for(int i=0;i<=m;i++) ans[i]=tmp[i]=0;
    for(int i=0;i<=b[1]&&i*a[1]<=m;i++)
        ans[i*a[1]]=1;
    //枚举第 k 类,做乘法(x^i)*(x^(j*a[k]))
    for(int k=2;k<=n;k++){
        for(int i=0;i<=m;i++){
            if(!ans[i]) continue;
            for(int j=0;j<=b[k]&&i+j*a[k]<=m;j++)
                tmp[i+j*a[k]]+=ans[i];
        }
        for(int i=0;i<=m;i++)
            ans[i]=tmp[i],tmp[i]=0;
    }
}
```

第六章 博弈论

6.1 尼姆博弈

有若干堆物品，两人依次从中拿取，每次只能从一堆中拿取若干个，可以全部拿走，但不可不取，最后取完者获胜。

奇异局势：全部堆物品的个数异或值为 0

结论：先来者，奇异局势必败，非奇异局势必胜。

定理 1：必定存在一种取法使得非奇异局势变成奇异局势。

设$S = A_1 \oplus A_2 \oplus \cdots \oplus A_n \neq 0$,设 S 的二进制表示最高位为 1 是第 k 位,那么必然存在A_t的第 k 位为 1,并且有$A_t \oplus S \leq A_t$,所以可取走$A_t - (A_t \oplus S)$个物品,使得$A_1 \oplus A_2 \oplus \cdots \oplus (A_t \oplus S) \oplus \cdots \oplus A_n = 0$

定理 2：对于任意一种取法，奇异局势必定变为非奇异局势。

有若干堆物品，两人依次从中拿取，每次只能从一堆中拿取

若干个，可以全部拿走，但不可不取，最后取完者失败。

定义：个数为 1 的堆为孤单堆，否则为充裕堆。

用 S 表示非奇异局势，T 为奇异局势，S0 为有 0 个充裕堆的非奇异局势，S1 为有 1 个充裕堆的非奇异局势，S2 为充裕堆个数大于等于 2 的非奇异局势，T0 为有 0 个充裕堆的奇异局势，T2 为充裕堆个数大于等于 2 的奇异局势，注：没有 T1，因为仅含一个充裕堆的异或值高位必然有 1，故不会是奇异局势。

结论：先来者，S0，T2 必败，S1，S2，T0 必胜。

定理 1：S0(奇数个孤单堆)必败，T(偶数个孤单堆)必胜。

定理 2：S1 可以转成 S0 或者 T0，根据 S1 的孤单堆的奇偶性，选择取走充裕堆的全部物品或者剩余 1 个。

定理 3：S2 可以转成 T2(S2 可以转成 T，但不能转成 T0)，T2 可以转成 S2 或者 S1(T2 可以转成 S，但不能转成 S0)。

6.2 威佐夫博弈

有两堆物品，两人轮流从某一堆或同时从两堆中取同样多的物品，每次至少取一个，多者不限，最后取完者获胜。

前几个奇异局势是(0,0),(1,2),(3,5),(4,7),(6,10),(8,13),(9,15),(11,18)

性质一:任何一个自然数包含在有且仅有一个奇异局势中。

性质二:任意两个奇异局势的差值不同。

性质三:差值为 n 的奇异局势为$\left(\frac{n\cdot(1+\sqrt{5})}{2},\frac{n\cdot(1+\sqrt{5})}{2}+n\right)$

Beatty 定理: 正无理数 α , β 满足$\frac{1}{\alpha}+\frac{1}{\beta}=1$, 则数列$a_n=\lfloor n\cdot\alpha\rfloor$,

$b_n = \lfloor n \cdot \beta \rfloor$严格递增,且集合$\{a_n : n \in Z^+\}$和$\{b_n : n \in Z^+\}$构成$Z^+$的一个划分

上述除(0,0)以外的奇异局势,恰好是 Beatty 序列,$a_n = \lfloor n \cdot \alpha \rfloor, b_n = a_n + n = \lfloor n \cdot (\alpha + 1) \rfloor$,则 $\beta = \alpha + 1$,又$\frac{1}{\alpha} + \frac{1}{\beta} = 1$,得到 $\alpha = \frac{1+\sqrt{5}}{2}$

6.3 斐波那契博弈

有一堆个数大于 1 的物品，两人轮流从中取物品，最后取完者获胜，符合以下两条规则:

1. 先手不能在第一次把所有物品取完，至少取 1 个
2. 之后每次可取的物品至少为 1，至多为上一轮对手取的两倍

可以观察得到奇异局势为 2,3,5,8,13,…是个斐波那契数列

Zeckendorf 定理: 任何整数可以表示为若干个不连续的 Fibonacci 数之和(连续的 Fib 数可以合并成更大的 Fib 数)

对于 Fib[i],可以拆成 Fib[i-1]+Fib[i-2],由于 Fib[i-1]<2Fib[i-1],因此先手第一次不能取 Fib[i-2],否则后手直接取剩余的 Fib[i-1],于是后手取完 Fib[i-2]中最后一个物品,并且可证下一次先手不能一次取完 Fib[i-1],于是后手又取完 Fib[i-1]中最后一个物品,因此先手必败

对于非 Fib 数,则可拆成若干个不连续的数,因此拆分方案中次小的 Fib 数大于最小的 Fib 数的 2 倍,于是先手可在第一次取完最小的 Fib 数,而后手不能取完次小的 Fib 数,于是转化成后手先来的若干个 Fib 数,这样后手必败,也就是先手必胜

6.4 SG 函数

定义状态 X 的所有后继状态集合$\{Y_1, Y_2, ..., Y_n\}$,G(X)为后继状态组成的 SG 集合$\{SG(Y_1), SG(Y_2), ..., SG(Y_n)\}$的补集，则 SG(X)=min G(X)

满足两条性质:

1. SG(X)不属于后继状态组成的 SG 集合$\{SG(Y_1), SG(Y_2), ..., SG(Y_n)\}$
2. 后继状态组成的 SG 集合$\{SG(Y_1), SG(Y_2), ..., SG(Y_n)\}$中包含了子集$\{0, 1, ..., SG(X)-1\}$

第七章 各种规划

7.1 分数规划

求$\lambda = f(x) = a(x)/b(x)$最大(小)值,$(x \in S), \forall x \in S, b(x) > 0$

其中,解向量 x 在解空间 S 内,$a(x)$与 $b(x)$都是连续的实值函数

$\lambda^* = f(x^*)$为最优解,构造函数 $g(\lambda) = \min_{x \in S}\{a(x) - \lambda b(x)\}$

性质(单调性):$g(\lambda)$是一个严格递减函数,即对于$\lambda_1 < \lambda_2$,

一定有 $g(\lambda_1) > g(\lambda_2)$

Dinkelbach 定理: 设λ^*为原规划的最优解,则 $g(\lambda) = 0$

当且仅当$\lambda = \lambda^*$

推论:$\begin{cases} g(\lambda) = 0, & \lambda = \lambda^* \\ g(\lambda) < 0, & \lambda > \lambda^* \\ g(\lambda) > 0, & \lambda < \lambda^* \end{cases}$

7.1.1 0-1 分数规划

$$求L^* = \sum_{i=1}^{n} CiXi \Big/ \sum_{i=1}^{n} DiXi \text{ 的最大(小)值，Xi}\epsilon\{0,1\}，\ \sum_{i=1}^{n} DiXi$$

$$\text{为正整数,Ci,Di 为整数,} L^* \in [-nC, nC], C = \max_{1\le i\le n}\{|Ci|, 1\}$$

$$\mathrm{Q(L)} = \sum_{i=1}^{n} CiXi - L\sum_{i=1}^{n} DiXi\,, Z(L)\text{为 Q(L)的最大(小)值}$$

$$\mathrm{Z(L)} = \begin{cases} > 0 & \mathrm{L} < L^* \\ = 0 & L = L^* \\ < 0 & L > L^* \end{cases}$$

7.2 数位 dp

```
/***数位 dp(dfs 版)***/
/***允许前导 0***/
int dp[20][Maxn]; //初始化-1
int num[20]; //num[n...0]表示原数字的十进制
int target; //终态
int newState; //新状态
/*
 *cur 表示当前考虑位,同时也是可变位数[00..0,99..9]
 *state 表示当前位之前的位构成的状态
 *flag:true 是上界,后面位数有限制;false 不是上界,后面位数可变
*/
int dfs(int cur,int state,bool flag){
    if(cur==-1) return state==target; //是否符合终态
    if(!flag&&dp[cur][state]>=0) //不是上界且搜索过
        return dp[cur][state];
    int ans=0;
    int up=flag?num[cur]:9; //计算上界
    for(int i=0;i<=up;i++) //枚举当前位
        //上界条件:前面是上界且此位也为上界
        ans+=dfs(cur-1,newState,flag&&i==up);
    return flag?ans:dp[cur][state]=ans; //不是上界才可记忆化
}

/***数位 dp(dfs 版)***/
```

```
/***不允许前导 0***/
int dp[20][Maxn]; //初始化-1
int num[20]; //num[n...0]表示原数字的十进制
int target; //终态
int newState; //新状态
/*
 *cur 表示当前考虑位,同时也是可变位数[00..0,99..9]
 *state 当前位之前的位构成的状态
 *flag:true 是上界,后面位数有限制;false 不是上界,后面位数可变
 *first:true 表示前面没有出现过非 0 数字;false 表示前面出现过非 0 数字
*/
int dfs(int cur,int state,bool flag,bool first){
    if(cur==-1) return state==target; //是否符合终态
    if(!flag&&!first&&dp[cur][state]>=0) //不是上界且非前导 0 且搜索过
        return dp[cur][state];
    int ans=0;
    int up=flag?num[cur]:9; //计算上界
    for(int i=0;i<=up;i++){ //枚举当前位
        if(first&&!i&&cur>0) //有前导 0 并且不是个位数字 0,状态不转移
            ans+=dfs(cur-1,state,flag&&i==up,true);
        else //上界条件:前面是上界且此位也为上界
            ans+=dfs(cur-1,newState,flag&&i==up,false);
    }
    //不是上界并且前面有非 0 数才可记忆化
    if(!flag&&!first) return dp[cur][state]=ans;
    return ans;
}
```

7.3 线性规划

标准形式：

$$\max \sum_{i=1}^{n} c_i x_i$$

$$\sum_{j=1}^{n} a_{ij} x_j \le b_i,\ i = 1,2,...,m$$

$$x_i \ge 0,\ i = 1,2,...,n$$

对偶形式：

$$\min \sum_{i=1}^{m} b_i y_i$$

$$\sum_{j=1}^{m} a_{ji} y_j \le b_i,\ i = 1,2,...,n$$

$$y_i \ge 0,\ i = 1,2,...,m$$

对偶就是把 n 和 m 交换，b 和 c 交换，A 转置
求具体解：取非基变量为全 0,解出基变量

第八章 常用模板

8.1 输入挂

```
inline int read(){ //输入挂
    int x=0;
    char ch=getchar();
    while(ch<'0'||ch>'9') ch=getchar();
    while(ch>='0'&&ch<='9'){
        x=x*10+ch-'0';
        ch=getchar();
    }
    return x;
}

inline int read(){ //支持负数
    int x=0,f=1;
    char ch=getchar();
    while(ch<'0'||ch>'9'){
        if(ch=='-') f=-1;
        ch=getchar();
    }
    while(ch>='0'&&ch<='9'){
        x=x*10+ch-'0';
        ch=getchar();
    }
    return x*f;
}
```

8.2 各种快速变换

8.2.1 FFT

```
/*****快速傅里叶变换*****/
/*****复杂度 O(nlogn)*****/
//可取 Maxn 的值,2^18=262144,2^19=524288
const double PI=acos(-1.0);
struct complex{
    double r,i; //实部,虚部
    complex(){}
    complex(double _r,double _i):r(_r),i(_i){}
    complex operator+(const complex a){
        return complex(r+a.r,i+a.i);
    }
    complex operator-(const complex a){
        return complex(r-a.r,i-a.i);
    }
    complex operator*(const complex a){
        return complex(r*a.r-i*a.i,r*a.i+i*a.r);
    }
}w[Maxn],ya[Maxn],yb[Maxn];
int a[Maxn],b[Maxn],c[Maxn];
void init(int n,int sign){ //初始化 x^n=1 的 n 个单位复根,DFT 要使用
    w[0]=complex(1,0);
    w[1]=complex(cos(2*PI/n),sign*sin(2*PI/n));
    for(int i=2;i<n;i++)
        w[i]=w[i-1]*w[1];
}
//将长 bit 位的 x 的二进制表示反转
int rev(int x,int bit){ //x 的范围[0,bit-1]
    int res=0;
    for(int i=0;i<bit;i++){
        res=res<<1|x&1;
        x>>=1;
    }
    return res;
}
//将系数表示转换成点值表示
//传入时 y 是系数向量,虚部为 0,函数返回时,y 是点值法纵坐标(复数)
void DFT(complex *y,int bit){
    int n=1<<bit;
    for(int i=0;i<n;i++){ //最后一层满足 y[i]=a[rev(i)],虚部为 0
```

```
        int j=rev(i,bit);
        if(i<j) swap(y[i],y[j]);
    }
    for(int i=2;i<=n;i<<=1){
        int m=i>>1; //合并的区间长度
        for(int j=0;j<n;j+=i){ //枚举起点
            //合并[j,j+m-1]和[j+m,j+2*m-1]
            for(int k=0;k<m;k++){
                complex t=y[j+k+m]*w[n/i*k]; //(wi)^k=(wn)^(n/i*k)
                y[j+k+m]=y[j+k]-t;
                y[j+k]=y[j+k]+t;
            }
        }
    }
}
//将点值表示转换成系数表示
//对点值法纵坐标 y 做 DFT
void IDFT(int *c,complex *y,int bit){
    DFT(y,bit);
    int n=1<<bit;
    for(int i=0;i<n;i++)
        c[i]=floor(y[i].r/n+0.5); //原 bug,负数四舍五入用(int)会出错
}
//系数向量 a 和 b,长度分别为 la 和 lb,最高次为 la-1 和 lb-1
//多项式为 a[0]+a[1]*x+...,b[0]+b[1]*x+...
//c=a*b
void FFT(int *a,int *b,int *c,complex *ya,complex *yb,int la,int lb){
    int len=la+lb-1; //结果的系数向量长度
    int n=1,bit=0;
    while(n<len) n<<=1,bit++;
    for(int i=la;i<n;i++) a[i]=0; //高位补 0
    for(int i=lb;i<n;i++) b[i]=0; //高位补 0
    for(int i=0;i<n;i++)
        ya[i]=complex(a[i],0),yb[i]=complex(b[i],0);
    init(n,1);
    DFT(ya,bit);
    DFT(yb,bit);
    for(int i=0;i<n;i++) ya[i]=ya[i]*yb[i]; //点值法做乘积
    init(n,-1);
    memset(c,0,sizeof c); //考虑最高位进位,防止前几组数据影响
    IDFT(c,ya,bit);
}
```

8.2.2 FFT 支持模数

```
/*****快速傅里叶变换*****/
/*****复杂度 O(nlogn)*****/
//可取 Maxn 的值,2^18=262144,2^19=524288
const int mod=1e9+7;
const int m=sqrt(mod);
const long double PI=acos(-1.0);
struct complex{
    long double r,i; //实部,虚部
    complex(){}
    complex(long double _r,long double _i):r(_r),i(_i){}
    complex operator+(const complex a){
        return complex(r+a.r,i+a.i);
    }
    complex operator-(const complex a){
        return complex(r-a.r,i-a.i);
    }
    complex operator*(const complex a){
        return complex(r*a.r-i*a.i,r*a.i+i*a.r);
    }
}w[Maxn],ya[Maxn],yb[Maxn],yc[Maxn],yd[Maxn],ye[Maxn];
int a[Maxn],b[Maxn],c[Maxn]; //a[i]=A[i]*m+B[i],b[i]=C[i]*m+D[i]
void init(int n,int sign){ //初始化 x^n=1 的 n 个单位复根,DFT 要使用
    w[0]=complex(1,0);
    w[1]=complex(cos(2*PI/n),sign*sin(2*PI/n));
    for(int i=2;i<n;i++)
        w[i]=w[i-1]*w[1];
}
//将长 bit 位的 x 的二进制表示反转
int rev(int x,int bit){ //x 的范围[0,bit-1]
    int res=0;
    for(int i=0;i<bit;i++){
        res=res<<1|x&1;
        x>>=1;
    }
    return res;
}
//将系数表示转换成点值表示
//传入时 y 是系数向量,虚部为 0,函数返回时,y 是点值法纵坐标(复数)
void DFT(complex *y,int bit){
    int n=1<<bit;
    for(int i=0;i<n;i++){ //最后一层满足 y[i]=a[rev(i)],虚部为 0
```

```
        int j=rev(i,bit);
        if(i<j) swap(y[i],y[j]);
    }
    for(int i=2;i<=n;i<<=1){
        int m=i>>1; //合并的区间长度
        for(int j=0;j<n;j+=i){ //枚举起点
            //合并[j,j+m-1]和[j+m,j+2*m-1]
            for(int k=0;k<m;k++){
                complex t=y[j+k+m]*w[n/i*k]; //(wi)^k=(wn)^(n/i*k)
                y[j+k+m]=y[j+k]-t;
                y[j+k]=y[j+k]+t;
            }
        }
    }
}
//系数向量 a 和 b,长度分别为 la 和 lb,最高次为 la-1 和 lb-1
//多项式为 a[0]+a[1]*x+...,b[0]+b[1]*x+...
//a=a*b
void FFT(int *a,int *b,int la,int lb){
    int len=la+lb-1; //结果的系数向量长度
    int n=1,bit=0;
    while(n<len) n<<=1,bit++;
    for(int i=la;i<n;i++) a[i]=0; //高位补 0
    for(int i=lb;i<n;i++) b[i]=0; //高位补 0
    for(int i=0;i<n;i++){
        ya[i]=complex(a[i]/m,0),yb[i]=complex(a[i]%m,0);
        yc[i]=complex(b[i]/m,0),yd[i]=complex(b[i]%m,0);
    }
    init(n,1);
    DFT(ya,bit),DFT(yb,bit);
    DFT(yc,bit),DFT(yd,bit);
    for(int i=0;i<n;i++){ //点值法做乘积
        ye[i]=ya[i]*yc[i];
        ya[i]=ya[i]*yd[i]+yb[i]*yc[i];
        yb[i]=yb[i]*yd[i];
    }
    init(n,-1); //做 IDFT 前初始化
    DFT(ya,bit),DFT(yb,bit),DFT(ye,bit);
    for(int i=0;i<n;i++){
        a[i]=(ll)floor(ye[i].r/n+0.5)%mod*m*m%mod;
        a[i]=(a[i]+(ll)floor(ya[i].r/n+0.5)%mod*m)%mod;
        a[i]=(a[i]+(ll)floor(yb[i].r/n+0.5))%mod;
    }
}
```

8.2.3 NTT

```
/*****快速数论变换*****/
/*****复杂度 O(nlogn)*****/
//可取 Maxn 的值,2^18=262144,2^19=524288
//mod=3*2^18+1=786433,g=10
//mod=479*2^21+1=1004535809,g=3
const int mod=786433; //费马素数
const int g=10; //mod 的原根
ll pw(ll a,int b){
    ll ans=1;
    while(b){
        if(b&1) ans=ans*a%mod;
        a=a*a%mod;
        b>>=1;
    }
    return ans;
}
ll w[Maxn];
ll a[Maxn],b[Maxn];
void init(int n,int sign){ //初始化 x^n=1 的 n 个单位根,FNT 要使用
    w[0]=1;
    w[1]=pw(g,(mod-1)/n); //n 阶单位根
    if(sign==-1) w[1]=pw(w[1],mod-2); //逆元
    for(int i=2;i<n;i++)
        w[i]=w[i-1]*w[1]%mod;
}
//将长 bit 位的 x 的二进制表示反转
int rev(int x,int bit){ //x 的范围[0,bit-1]
    int res=0;
    for(int i=0;i<bit;i++){
        res=res<<1|x&1;
        x>>=1;
    }
    return res;
}
//将系数表示转换成点值表示
//传入时 y 是系数向量,函数返回时,y 是点值法纵坐标
void FNT(ll *y,int bit){
    int n=1<<bit;
    for(int i=0;i<n;i++){ //最后一层满足 y[i]=a[rev(i)]
        int j=rev(i,bit);
```

```
        if(i<j) swap(y[i],y[j]);
    }
    for(int i=2;i<=n;i<<=1){
        int m=i>>1; //合并的区间长度
        for(int j=0;j<n;j+=i){ //枚举起点
            //合并[j,j+m-1]和[j+m,j+2*m-1]
            for(int k=0;k<m;k++){
                ll t=y[j+k+m]*w[n/i*k]%mod; //(wi)^k=(wn)^(n/i*k)
                y[j+k+m]=(y[j+k]-t+mod)%mod;
                y[j+k]=(y[j+k]+t)%mod;
            }
        }
    }
}
//将点值表示转换成系数表示
//对点值法纵坐标 y 做 FNT
void IFNT(ll *y,int bit){
    FNT(y,bit);
    int n=1<<bit;
    ll inv=pw(n,mod-2);
    for(int i=0;i<n;i++)
        y[i]=y[i]*inv%mod;
}
//系数向量 a 和 b,长度分别为 la 和 lb,最高次为 la-1 和 lb-1
//多项式为 a[0]+a[1]*x+...,b[0]+b[1]*x+...
//a=a*b%mod,a 和 b 会改变,结果存在 a 中
void NTT(ll *a,ll *b,int la,int lb){
    int len=la+lb-1; //结果的系数向量长度
    int n=1,bit=0;
    while(n<len) n<<=1,bit++;
    for(int i=la;i<n;i++) a[i]=0; //高位补 0
    for(int i=lb;i<n;i++) b[i]=0; //高位补 0
    init(n,1);
    FNT(a,bit);
    FNT(b,bit);
    for(int i=0;i<n;i++) a[i]=a[i]*b[i]%mod; //点值法做乘积
    init(n,-1);
    IFNT(a,bit);
}
//注意:对于大数乘法(进位),避免上一组测试数据的影响,应清空 a,b 数组
```

8.3 高精度

```
/*****高精度*****/
```

```
struct bign{
    int len;
    int s[Maxn];
    bign(){ //构造函数
        memset(s,0,sizeof(s));
        len=1;
    }
    void operator=(const int num){
        int n=num;
        len=0;
        do{
            s[len++]=n%10;
            n/=10;
        }while(n);
    }
    bign(int num){  //拷贝构造
        *this=num;
    }
    void operator=(const char *num){ //去掉前导 0,num 非空
        int i=-1,j;
        while(num[++i]=='0');
        len=strlen(num+i);
        if(len==0){ //单独处理 0
            memset(s,0,sizeof(s));
            len=1;
            return;
        }
        for(j=i+len-1;j>=i;j--)
            s[len-1+i-j]=num[j]-'0';
    }
    bign operator+(const int b)const{
        bign t(b);
        return t+*this;
    }
    bign& operator+=(const int b){
        *this=*this+b;
        return *this;
    }
    bign operator+(const bign &b)const{
        bign c;
        c.len=0;
        int m=len>b.len?len:b.len;
        for(int i=0,g=0;g||i<m;i++){ //g 为进位
            if(i<len) g+=s[i];
```

```
        if(i<b.len) g+=b.s[i];
        c.s[c.len++]=g%10;
        g/=10;
    }
    return c;
}
bign& operator+=(const bign &b){
    *this=*this+b;
    return *this;
}
bign operator*(const int b)const{ //大数乘(int)
    bign c;
    c.len=0;
    long long g=0;
    for(int i=0;i<len;i++){
        g+=s[i]*b;
        c.s[c.len++]=g%10;
        g/=10;
    }
    while(g){
        c.s[c.len++]=g%10;
        g/=10;
    }
    return c;
}
bign& operator*=(const int b){
    *this=*this*b;
    return *this;
}
bign operator*(const bign &b)const{ //大数乘大数
    bign c;
    c.len=len+b.len;
    for(int i=0;i<len;i++)
        for(int j=0;j<b.len;j++)
            c.s[i+j]+=s[i]*b.s[j];
    for(int i=0;i<c.len;i++){
        c.s[i+1]+=c.s[i]/10;
        c.s[i]%=10;
    }
    c.clean();
    return c;
}
bign& operator*=(const bign &b){
    *this=*this*b;
```

```
        return *this;
    }
    bign operator-(const int b)const{
        bign t(b);
        return *this-t;
    }
    bign& operator--(const int b){
        *this=*this-b;
        return *this;
    }
    bign operator-(const bign &b)const{ //大数减小数
        bign c;
        c.len=0;
        for(int i=0,g=0;i<len;i++){ //g为借位
            int x=s[i]-g;
            if(i<b.len) x-=b.s[i];
            if(x>=0) g=0;
            else {g=1;x+=10;}
            c.s[c.len++]=x;
        }
        c.clean();
        return c;
    }
    bign& operator-=(const bign &b){
        *this=*this-b;
        return *this;
    }
    bign operator/(const int b)const{ //大数除以(int),整除
        bign c;
        long long g=0;
        for(int i=len-1;i>=0;i--){
            g=g*10+s[i];
            c.s[i]=g/b;
            g%=b;
        }
        c.len=len;
        c.clean();
        return c;
    }
    bign& operator/=(const int b){
        *this=*this/b;
        return *this;
    }
    bign operator/(const bign &b)const{ //大数除以大数,整除
```

```
        bign c,g;
        for(int i=len-1;i>=0;i--){
            g*=10;
            g.s[0]=s[i];
            while(g>=b){
                c.s[i]++;
                g-=b;
            }
        }
        c.len=len;
        c.clean();
        return c;
    }
    bign& operator/=(const bign &b){
        *this=*this/b;
        return *this;
    }
    bign operator%(const int b)const{
        bign r=*this/b;
        r=*this-r*b;
        return r;
    }
    bign& operator%=(const int b){
        *this=*this%b;
        return *this;
    }
    bign operator%(const bign &b)const{
        bign r=*this/b;
        r=*this-r*b;
        return r;
    }
    bign& operator%=(const bign &b){
        *this=*this%b;
        return *this;
    }
    bign operator^(int n)const{ //大数的n次方
        bign res(1),tmp(*this);
        while(n){
            if(n&1)
                res*=tmp;
            tmp*=tmp;
            n>>=1;
        }
        return res;
```

```
}
bign& operator^=(int n){
    *this=*this^n;
    return *this;
}
bign& operator++(){
    *this+=1;
    return *this;
}
bign operator++(int){
    bign tmp(*this);
    ++*this;
    return tmp;
}
bign& operator--(){
    *this-=1;
    return *this;
}
bign operator--(int){
    bign tmp(*this);
    --*this;
    return tmp;
}
bool operator<(const bign &b)const{
    if(len!=b.len) return len<b.len;
    for(int i=len-1;i>=0;i--)
        if(s[i]!=b.s[i]) return s[i]<b.s[i];
    return false;
}
bool operator>(const bign &b)const{
    if(len!=b.len) return len>b.len;
    for(int i=len-1;i>=0;i--)
        if(s[i]!=b.s[i]) return s[i]>b.s[i];
    return false;
}
bool operator==(const bign &b)const{
    if(len!=b.len) return false;
    for(int i=len-1;i>=0;i--)
        if(s[i]!=b.s[i]) return false;
    return true;
}
bool operator!=(const bign &b)const{
    return !(*this==b);
}
```

```
    bool operator<=(const bign &b)const{
        return *this<b||*this==b;
    }
    bool operator>=(const bign &b)const{
        return *this>b||*this==b;
    }
    void output()const{ //输出
        for(int i=len-1;i>=0;i--)
            printf("%d",s[i]);
    }
    /*void output(int n)const{  //n 位小数，去后导 0;0.XX 显示为.XX
        int i,j;
        for(i=len-1;i>=n;i--)
            printf("%d",s[i]);
        for(i=0;i<n;i++){
            if(s[i]) break;
        }
        if(i<n){
            printf(".");
            for(j=n-1;j>=i;j--)
                printf("%d",s[j]);
        }
        puts("");
    }*/
    void clean(){ //去前导 0
        while(len>1&&!s[len-1]) len--;
    }
};
istream& operator>>(istream &in,bign &x){
    char str[Maxn];
    //if(in>>str)
        //printf("%s",str);
    in>>str;
    x=str;
    return in;
}
ostream& operator<<(ostream &out,const bign &x){
    x.output();
    return out;
}
```

8.4 矩阵

8.4.1 快速幂

```
/*****矩阵快速幂*****/
/*****复杂度 O(d^3*logk)*****/
const int mod=10000007; //模
int d; //维数
struct matrix{
    long long mat[Maxn][Maxn];
    matrix(){memset(mat,0,sizeof mat);}
    matrix operator*(matrix &x){ //矩阵乘法,值返回
        matrix ans;
        for(int i=0;i<d;i++)
            for(int j=0;j<d;j++)
                for(int k=0;k<d;k++){
                    ans.mat[i][j]+=mat[i][k]*x.mat[k][j];
                    ans.mat[i][j]%=mod; //加模运算
                }
        return ans;
    }
};
//值传递，值返回
matrix quick_pow(matrix x,int k){ //x^k,矩阵快速幂
    matrix ans;
    for(int i=0;i<d;i++) //单位元
        ans.mat[i][i]=1;
    while(k){ //二进制思想
        if(k&1) ans=ans*x; //该位有 1
        x=x*x; //倍增
        k>>=1;
    }
    return ans;
}
```

8.5 排序

8.5.1 归并排序

```
/*****归并排序*****/
/*****复杂度 O(nlogn)*****/
```

```
//递归版
void merge(int l,int m,int r){
    int n1=m-l+1,n2=r-m,i,j;
    for(i=1;i<=n1;i++) L[i]=a[l+i-1];
    L[n1+1]=inf;
    for(i=1;i<=n2;i++) R[i]=a[m+i];
    R[n2+1]=inf;
    i=j=1;
    for(int k=l;k<=r;k++)
        if(L[i]<=R[j]) {a[k]=L[i];i++;}
        else {a[k]=R[j];j++;}
}
void merge_sort(int l,int r){
    if(l<r){
        int mid=l+r>>1;
        merge_sort(l,mid);
        merge_sort(mid+1,r);
        merge(l,mid,r);
    }
}

//非递归的归并排序，下标[0,n-1]
void merge(int *a,int *b,int l,int m,int r){ //a 合并到 b
    int ls=l,rs=m+1,k=l;
    while(ls<=m&&rs<=r){
        if(a[ls]<a[rs]) b[k++]=a[ls++];
        else b[k++]=a[rs++];
    }
    if(ls<=m)
        for(int i=ls;i<=m;i++) b[k++]=a[i];
    else
        for(int i=rs;i<=r;i++) b[k++]=a[i];
}
void merge_pass(int *a,int *b,int s,int n){ //a 合并到 b
    int i=0;
    while(i+2*s<=n){ //合并大小为 s 的相邻两段子数组
        merge(a,b,i,i+s-1,i+2*s-1);
        i=i+2*s;
    }
    //剩余数字
    if(i+s<n) merge(a,b,i,i+s-1,n-1);
    else
        for(int j=i;j<n;j++) b[j]=a[j];
}
```

```
void merge_sort(int *a,int *b,int n){
    int s=1;
    while(s<n){
        merge_pass(a,b,s,n);
        s+=s;
        merge_pass(b,a,s,n);
        s+=s;
    }
}
```

8.6 128 位模拟 64 位乘 64 位

```
struct num_128{
    ull ah,al; //二进制高 64 位和低 64 位
    num_128(){}
    num_128(ull _ah,ull _al):ah(_ah),al(_al){}
};
num_128 mul_64(ull a,ull b){
    num_128 x,y,res;
    x.ah=a>>32,x.al=a&0xffffffff;
    y.ah=b>>32,y.al=b&0xffffffff;
    ull c;
    res.al=x.al*y.al; //低 32 位相乘
    c=x.al*y.ah; //低 32 和高 32,乘积前 32 位对应[64,96)
    res.ah=c>>32; //提取前 32 位放在 rh
    c<<=32; //第 32 位左移 32 位
    res.al+=c; //累加在 rl
    if(res.al<c) res.ah++; //rl 说明进位为 1
    c=x.ah*y.al; //同上
    res.ah+=(c>>32);
    c<<=32;
    res.al+=c;
    if(res.al<c) res.ah++;
    res.ah+=x.ah*y.ah; //高 32 位相乘
    return res;
}
ull mod_128(ull a,ull b,ull mod){
    num_128 res=mul_64(a,b);
    ull ans=(res.ah%mod+res.al%mod)%mod;
    ull t=-1ULL; //res.ah*(2^64-1)%mod
    ull tmp=res.ah;
    for(int i=1;i<=64;i++){
        ans=(ans+tmp)%mod;
```

```
        tmp=(tmp<<1)%mod;
    }
    return ans;
}
```

8.7 分治

8.7.1 CDQ 分治

```
/*****cdq 分治*****/
/***复杂度 O(n(logn)^2)***/
/*
 *query 里重要变量
 *op:操作类型,修改和询问
 *id:询问对应的下标(第 id 个询问)
 *dfn:操作的时间序
*/
struct query{
    int x,y,v,op,id,dfn;
    query(){}
    query(int _x,int _y,int _v,int _op,int _id,int _dfn){
        x=_x,y=_y,v=_v,op=_op,id=_id,dfn=_dfn;
    }
    bool operator<(const query &a)const{
        if(x==a.x&&y==a.y) return op<a.op;
        if(x==a.x) return y<a.y;
        return x<a.x;
    }
}q[200010],tmp[200010];
int ans[10010];
/*
 *只有在询问 i 之前的修改才能对询问 i 做贡献
 *cdq(l,r)中计算[l,mid]的修改对[mid+1,r]的询问的贡献
 *同时递归左右区间,因为[mid+1,r]中的修改也可以对
 *[mid+1,r]中在修改之后的询问做贡献,递归保证了任何
 *一个询问都受到所有它之前的修改的贡献
 *递归时注意恢复一些数据和保证数组的序
*/
void cdq(int l,int r){
    if(l==r) return;
    int mid=l+r>>1;
    for(int i=l;i<=r;i++){
```

```
        //计算[l,mid]的修改对[mid+1,r]的询问的影响
        if(q[i].dfn<=mid&&q[i].op==1) add(q[i].y,q[i].v);
        if(q[i].dfn>mid&&q[i].op==2)
            ans[q[i].id]+=q[i].v*sum(q[i].y);
    }
    for(int i=l;i<=r;i++) //恢复树状数组
        if(q[i].dfn<=mid&&q[i].op==1) add(q[i].y,-q[i].v);
    int L=l,R=mid+1;
    for(int i=l;i<=r;i++) //按时间序分左右两部分
        if(q[i].dfn<=mid) tmp[L++]=q[i];
        else tmp[R++]=q[i];
    for(int i=l;i<=r;i++) q[i]=tmp[i];
    cdq(l,mid);
    cdq(mid+1,r);
}
```

第九章 数值分析

第十章 STL/Java

10.1 Set

成员函数	详细说明
set<int,less<int> >	数据类型，函数对象
less<int>	函数对象，不以降方式排序，即<=，非严格递增
greater<int>	函数对象，不以升方式排序,即>=，非严格递减
	对任意排在 y 前面的 x，key_comp(y,x)返回 false，

key_comp()	拟序<，即 x<y，key_comp(x,y)返回 true 严格来说：key_comp()返回的是 key_compare 函数对象，该函数对象里定义了成员函数，返回值 bool
set<int>::iterator	双向迭代器
begin()	返回一个双向迭代器，指向被控序列的第一个元素，如果序列为空，指向紧接着序列末端的下一个位置
end()	返回一个双向迭代器，指向紧接着序列末端的下一个位置
set<int>::reverse_iterator	反转型双向迭代器
rbegin()	返回一个反转型双向迭代器，指向序列的最后一个元素
rend()	返回一个反转型双向迭代器，指向序列的第一个元素的前一个位置
empty()	序列为空，返回 true
lower_bound()	返回一个指向常量的迭代器，指向被控序列第一个让 key_comp(x,key)为 false 的元素 x,若不存在这样的元素，返回 end() (less<int>:x>=key;greater<int>:x<=key)
upper_bound()	返回一个指向常量的迭代器，指向被控序列第一个让 key_comp(key,x)为 true 的元

	素 x,若不存在这样的元素，返回 end() (less<int>:x>key;greater<int>:x<key)
size()	返回被控序列的长度，size_type
max_size()	返回被控序列的最大长度，size_type
pair<iterator,int> insert(const value_type& x)	这个函数检测被控序列中是否存在其键与 x 有相等次序的元素 y；如果没有的话，会创建一个这样的元素 y，并把 x 作为它的初始值；如果有插入发生，函数将返回 pair<it,true>,否则返回 pair<it,false>
iterator insert(iterator it,const value_type & x)	从 it 位置开始查找插入 x，如果 it 靠近插入位置，时间分摊为常数；插入成功，返回指向插入位置的迭代器；否则，容器保持不变，向外抛出异常
void insert(Init first,Init last)	插入区间[first，last)中的元素，Init 可以是迭代器，也可以是数组
void erase(iterator it)	删除由 it 所指向的元素；删除成功迭代器失效，因此要取得下一个元素的位置，务必使用后置自增技术
void erase(iterator first,iterator last)	删除区间[first,last)中的所有元素
size_type erase(const int &key)	删除[lower_bound(key), upper_bound(key))中具有给定排序键的所

	有元素，返回删除元素的个数
find()	返回一个常量迭代器，指向被控序列中排序键与 key 相等的元素，不存在这样的元素函数将返回 end()　注：复杂度 log(n)
equal_range()	pair<const_iterator,const_iterator> equal_range(const int &key) const; 返回一个迭代器对 x，其中 x.first==lower_bound(key), x.second==upper_bound(key)
count()	size_type count(const int &key) const 返回[lower_bound(key), upper_bound(key))中的 key 的个数
clear()	调用 erase(begin(),end())

10.2 Bitset

成员函数	详细说明
时间复杂度	对于整个 bitset 操作(32 位一起操作),比普通数组快 32 倍，对于单个位操作(需要解析),比普通数组慢 8 倍左右
bitset<size_ t N>()	默认构造函数

bitset<size_ t N>(int)	用 int 型构造
bitset<size_ t N>(string)	用 string 型构造
[]	取位,下标从 0 开始
&, |, ^, <<, >>, ~	与,或,异或,左移,右移,按位取反
&=, |=, ^=, <<=, >>=, ~=	赋值与,赋值或,赋值异或,赋值左移,赋值右移,赋值按位取反
bitset& set()	全部置 1
bitset& reset()	全部置 0
bitset& flip()	全部位翻转(0<->1)
bitset& reset(size_t pos)	第 pos 位置 0
bitset& flip(size_t pos)	第 pos 位翻转
bitset& set(size_t pos,bool value=true)	第 pos 位置 value,默认为 true
size()	返回被控序列的长度
count()	返回被控序列的 1 的位数
bool test(size_t pos) const	第 pos 位为 1,返回 true,否则 false
bool any() const	全 0 返回 false,否则 true
bool none() const	全 0 返回 true,否则 false
bool all() const (c++11)	全 1 返回 true,否则 false

10.3 BigInteger

类方法	详细说明
ZERO,ONE,TEN	静态常量
BigInteger(byte[])	构造方法,大端字节序
BigInteger(int sign,byte[])	-1 将符号位置 1,1 将符号位置 0,0 则要求 byte[]每个字节为 0
BigInteger(String)	默认 10 进制,前面可加'-',表示负数
BigInteger(String,int radix)	将字符串转成大数,进制范围[2,36],{0,1,...,9,A(a),B(b),...,Z(z)}
BigInteger(int bound,new Random())	随机产生[0,2^bound-1]的一个数
BigInteger(int bound,int certainty,new Random())	以大于 1-0.5^certainty 的概率随机产生[2,2^bound-1]的一个素数
valueOf(long)	返回 BigInteger
probablePrime(int bound, new Random())	返回一个[2,2^bound-1]的素数,有 2^-100 的概率返回合数,注意位数和时间/空间成比例
nextProbablePrime()	返回大于 this 的第一个素数,有 2^-100 的概率返回合数,但绝不会跳过一个素数

add(BigInter)	加
subtract(BigInter)	减
multiply(BigInter)	乘
divide(BigInter)	除
remainder(BigInter)	取余,符合和被除数相同
divideAndRemainder(BigInter)	返回 BigInteger[],商和余数
pow(int)	乘幂,参数为大于 0
abs()	取绝对值
negate()	返回相反数
signum()	正数返回 1,负数返回-1,0 返回 0
gcd(BigInter)	两个数绝对值的最大公约数,gcd(0,0)=0
mod(BigInter)	模数为正,返回值非负
modInverse(BigInter)	乘法逆元,模数为正,保证逆元存在
modPow(BigInter, BigInter)	乘幂取模,模数为正,指数可负(按照逆元的正次幂算)
shiftLeft(int)	左移,参数为负则右移
shiftRight(int)	右移,参数为负则左移
and(BigInter)	与
or(BigInter)	或
xor(BigInter)	异或

not()	按位取反
andNot(BigInter)	与参数的相反数
boolean testBit(int)	右数参数位是否为 1,从 0 开始
setBit(int)	右数参数位置 1,从 0 开始
clearBit(int)	右数参数位置 0,从 0 开始
flipBit(int)	右数参数位取反,从 0 开始
getLowestSetBit()	返回 lowbit 的索引 this==0?-1:log2(this&-this)
bitLength()	最小的二进制补码表示形式的位数,不包括符号位, ceil(log2(this<0?-this:this+1))
bitCount()	返回二进制补码与符号位不同的位数
isProbablePrime(int certainty)	素数返回 true,是素数的概率超出 1-0.5^certainty
compareTo(BigInter)	相等返回 0,大于返回 1,小于返回-1
boolean equals(Object)	返回 true 当且仅当传入数值相等的 BigInteger
max(BigInter)	取最大值
min(BigInter)	取最小值
int hashCode()	返回哈希值
toString(int radix)	返回 radix[2,36]进制形式的字符串,不在这个范围默认 10 进制

toString()	返回 10 进制形式的字符串
byte[] toByteArray()	返回 byte[]的长度为 ceil((this.bitLength() + 1)/8)
intValue()	截取后 32 位
longValue()	截取后 64 位
floatValue()	转成 float
doubleValue()	转成 double

第十一章 辅助数学公式

11.1 常数

11.1.1 圆周率

$$\pi = 3.14159265358979324$$

11.1.2 自然对数的底数

$$e = 2.7182818284590452$$

11.2 求和

$$1^2 + 2^2 + 3^2 + \cdots + n^2 = \frac{n(n+1)(2n+1)}{6}$$

$$1^3 + 2^3 + 3^3 + \cdots + n^3 = \left[\frac{n(n+1)}{2}\right]^2$$

$$1\cdot n+2\cdot(n-1)+\cdots+n\cdot 1=\frac{n(n+1)(n+2)}{6}$$

11.3 分解因式

$$x^n-1=(x-1)(x^{n-1}+x^{n-2}+\cdots+x+1),n\in Z^+$$

$$\mathrm{x}^n+1=(\mathrm{x}+1)(x^{n-1}-x^{n-2}+x^{n-3}-x^{n-4}+\cdots$$

$$+(-1)^{k-1}x^{n-k}+\cdots+1),\mathrm{n}\text{ 为奇数}$$

$$a^n-b^n=(a-b)(a^{n-1}+a^{n-2}b+\cdots+a^{n-k}b^{k-1}+\cdots$$

$$+b^{n-1}),n\in Z^+$$

11.4 重要不等式

$$\left(\sum_{k=1}^{\mathrm{n}}a_k\right)\left(\sum_{k=1}^{n}b_k\right)=n\sum_{k=1}^{n}a_k\,b_k-\sum_{1\le j<k\le n}(a_k-a_j)(b_k-b_j)$$

$$\left(\sum_{k=1}^{\mathrm{n}}a_k\right)\left(\sum_{k=1}^{n}b_k\right)\le n\sum_{k=1}^{n}a_k\,b_k,a_1\le\cdots\le a_n,b_1\le\cdots\le b_n$$

$$\left(\sum_{k=1}^{\mathrm{n}}a_k\right)\left(\sum_{k=1}^{n}b_k\right)\ge n\sum_{k=1}^{n}a_k\,b_k,a_1\le\cdots\le a_n,b_1\ge\cdots\ge b_n$$

Chcbyshcv（切比雪夫）不等式：顺序和 ≥ 乱序和 ≥ 倒序和

11.5 范式运算律

11.5.1 交换律

$$\mathrm{X}\vee\mathrm{Y}=\mathrm{Y}\vee\mathrm{X},\mathrm{X}\wedge\mathrm{Y}=\mathrm{Y}\wedge\mathrm{X}$$

11.5.2结合律

$$(X \vee Y) \vee Z = X \vee (Y \vee Z), (X \wedge Y) \wedge Z = X \wedge (Y \wedge Z)$$

11.5.3分配律

$$\mathrm{X} \vee (Y \wedge Z) = (X \vee Y) \wedge (X \vee Z),$$

$$\mathrm{X} \wedge (Y \vee Z) = (X \wedge Y) \vee (X \wedge Z)$$

11.5.4吸收律

$$\mathrm{X} \vee \mathrm{X} = \mathrm{X}, \mathrm{X} \wedge \mathrm{X} = \mathrm{X}, \mathrm{X} \vee (X \wedge Y) = X, X \wedge (X \vee Y) = X$$

11.6 旋转

(x, y)绕坐标原点逆时针旋转 β 到(s, t): $\begin{pmatrix} s \\ t \end{pmatrix} =$

$$\begin{pmatrix} \cos\beta & -\sin\beta \\ \sin\beta & \cos\beta \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix}$$

三维叉积: $\vec{\mathrm{a}} * \vec{b} = \begin{vmatrix} \vec{i} & \vec{j} & \vec{k} \\ a_x & a_y & a_z \\ b_x & b_y & b_z \end{vmatrix}$

$$= (a_y b_z - a_z b_y)\vec{i} + (a_z b_x - a_x b_z)\vec{j} + (a_x b_y - a_y b_x)\vec{k}$$

11.7 求导

$$(\mathrm{a}^x)' = a^x \ln a$$

$$(\log_a x)' = \frac{1}{x \ln a}$$

11.8 欧拉公式

$$e^{ix} = \cos x + i \sin x$$

$$\text{n(顶点)} - \text{m(边)} + \text{r(面)} = 2$$

11.9 组合数公式

$$C_n^m = C_{n-1}^m + C_{n-1}^{m-1}$$

$$C_n^k = C_n^{k-1} \cdot \frac{n-k+1}{k}$$

$$C_{n-1}^k + C_{n-2}^k + \cdots + C_{k+1}^k + C_k^k = C_n^{k+1}$$

11.9.1 第一类 Stirling 数

$$x^{n\uparrow} = x(x+1)\cdots(x+n-1) = \sum_{k=0}^{n} S_u(n,k) \cdot x^k$$

$$x^{n\downarrow} = x(x-1)\cdots(x-n+1) = \sum_{k=0}^{n} S_s(n,k) \cdot x^k$$

无符号: $S_u(n+1,m) = S_u(n,m-1) + n \cdot S_u(n,m)$

有符号: $S_s(n+1,m) = S_s(n,m-1) - n \cdot S_s(n,m)$

释义:$S_u(n,m)$是将n个不同元素拆分成m个圆排列的方案数

性质:$S_1(0,0) = S_1(n,n) = 1, S_1(n,0) = 0,$

$$S_S(n,m) = (-1)^{n+m} S_u(n,m)$$

$$S_1(n,m) = 0, \begin{cases} m > n \\ n > 0, m \le 0 \end{cases}$$

11.9.2第二类 Stirling 数

$$S_2(n+1,m) = S_2(n,m-1) + m \cdot S_2(n,m)$$

释义:$S_2(n,m)$是将n个不同元素拆分成m个集合的方案数

性质:$S_2(0,0) = S_2(n,n) = 1, S_2(n,0) = 0,$

$$S_2(n,m) = 0, \begin{cases} m > n \\ n > 0, m \le 0 \end{cases}$$

11.10莫比乌斯反演

$$\sum_{d|n} u(d) = (n == 1)$$

$$F(n) = \sum_{d|n} f(d) \leftrightarrow f(n) = \sum_{d|n} u(d) \cdot F\left(\frac{n}{d}\right)$$

$$F(n) = \sum_{n|d} f(d) \Leftrightarrow f(n) = \sum_{n|d} u\left(\frac{d}{n}\right) \cdot F(d)$$

11.11几何公式

11.11.1 两点式的一般式

已知直线方程 $Ax + By + C = 0$ 过两个不同的点$(x_1, y_1), (x_2, y_2)$,

则 A, B, C 可取 $A = y_2 - y_1, B = x_1 - x_2, C = x_2 y_1 - x_1 y_2$

11.11.2 点到直线的公式

已知直线方程 $Ax + By + C = 0$ 和点(x_0, y_0),

则距离 $\mathrm{d}=\frac{|Ax_0+By_0+C|}{\sqrt{A^2+B^2}}$

11.11.3 皮克定理

对于一个顶点全部是整点的简单多边形(边不在除了线段端点外的地方相交),

简称格点多边形,设内部格点有 A 个,边界上的格点有 B 个,面积为 S,

则有 $\mathrm{S}=\mathrm{A}+\frac{B}{2}-1$

已知两点$(\mathrm{x}_1,y_1),(\mathrm{x}_2,y_2)$,则经过该线段(不包括线段两端点)

的格点数为

$$\gcd(|x_1-x_2|,|y_1-y_2|)-1$$

11.12数论定理

11.12.1 威尔逊定理

$$(\mathrm{p}-1)!\equiv\begin{cases}-1,\mathrm{p}\text{ 为素数}\\ 0,\mathrm{p}\text{ 为合数}(\mathrm{p}\neq 4)\end{cases}\quad(\mathrm{mod}\ \mathrm{p})$$

11.12.2 欧拉定理

$$\mathrm{a}^{\varphi(p)}\equiv 1\ (mod\ \ p)\quad a\perp p$$

11.12.3 Lucas 定理

$$C(\mathrm{n},\mathrm{m})\equiv C\left(\frac{n}{p},\frac{m}{p}\right)\cdot C(n\%p,m\%p)\quad(mod\ \ p)\ \ p\text{为素数}$$

11.12.4 Kummer 定理

引理：n! 中 p 的次幂为 $\sum_{i=1}^{k}\left\lfloor \frac{n}{p^i} \right\rfloor$，若 n! 的 p 进制表示为

$n_k n_{k-1} \dots n_0$，

则次幂也可表示为 $\frac{n-(n_k+n_{k-1}+\cdots+n_0)}{p-1}$

Kummer 定理：C_n^m 中 p 的次幂为 m 和 $(n-m)$ 的 p 进制表示相加产生的进位次数

加强结论：$\mathrm{LCM}(C_n^0, C_n^1, \dots, C_n^n) = \frac{\mathrm{LCM}(1,2,\dots,n+1)}{n+1}$

11.12.5 其他

$$\gcd\left(a^n-b^n, a^m-b^m\right) = a^{\gcd(n,m)}-b^{\gcd(n,m)},\ a>b$$

第十二章 补充

12.1 Hdu 手动扩栈

```
#pragma comment(linker, "/STACK:1024000000,1024000000")
```

12.2 C++取消同步

```
ios::sync_with_stdio(false);
```

12.3 C++运算符优先级

优先	操作	描述	例子	结合

级	符			性

1	()
	[]
	->
	.
	::
	++
	--
		调节优先级的括号操作符
		数组下标访问操作符
		通过指向对象的指针访问成员的操作符
		通过对象本身访问成员的操作符作用域操作符
		后置自增操作符
		后置自减操作符
			(a + b) / 4;
			array[4] = 2;
			ptr->age = 34;
			obj.age = 34;
			Class::age = 2;
			for(i = 0; i < 10; i++) ...
			for(i = 10; i > 0; i--) ...
				从左到右

2	!
	~
	++
	--
	-
	+
	*
	&
	(type)
	sizeof
		逻辑取反操作符
		按位取反(按位取补)
		前置自增操作符
		前置自减操作符
		一元取负操作符
		一元取正操作符
		解引用操作符
		取地址操作符
		类型转换操作符
		返回对象占用的字节数操作符
			if(!done) ...
			flags = ~flags;
			for(i = 0; i < 10; ++i) ...
			for(i = 10; i > 0; --i) ...
			int i = -1;
			int i = +1;
			data = *ptr;
			address = &obj;
			int i = (int) floatNum;
			int size = sizeof(floatNum);
				从右到左

3	->* .*	在指针上通过指向成员的指针访问成员的操作符 在对象上通过指向成员的指针访问成员的操作符	ptr->*var = 24; obj.*var = 24;	从左到右

4	*
	/
	%
		乘法操作符
		除法操作符
		取余数操作符
			int i = 2 * 4;
			float f = 10 / 3;
			int rem = 4 % 3;
				从左到右

5	+ -	加法操作符 减法操作符	int i = 2 + 3; int i = 5 - 1;	从左到右
6	<< >>	按位左移操作符 按位右移操作符	int flags = 33 << 1; int flags = 33 >> 1;	从左到右

7	<
	<=
	>
	>=
		小于比较操作符
		小于或等于比较操作符
		大于比较操作符
		大于或等于比较操作符
			if(i < 42) ...
			if(i <= 42) ...
			if(i > 42) ...
			if(i >= 42) ...
				从 左到右

8	== !=	等于比较操作符 不等于比较操作符	if(i == 42) ... if(i != 42) ...	从 左到右
9	&	按位与操作符	flags = flags & 42;	从 左到右
10	^	按位异或操作符	flags = flags ^ 42;	从 左到右
11	|	按位或操作符	flags = flags | 42;	从 左到右
12	&&	逻辑与操作符	if(conditionA && conditionB) ...	从 左到右
13	||	逻辑或操作符	if(conditionA || conditionB) ...	从 左到右
14	? :	三元条件操作符	int i = (a > b) ? a : b;	从 右到左

15	=
	+=
	-=
	*=
	/=
	%=
	&=
	^=
	|=
	<<=
	>>=
		赋值操作符
		复合赋值操作符(加法)
		复合赋值操作符(减法)
		复合赋值操作符(乘法)
		复合赋值操作符(除法)
		复合赋值操作符(取余)
		复合赋值操作符(按位与)
		复合赋值操作符(按位异或)
		复合赋值操作符(按位或)
		复合赋值操作符(按位左移)
		复合赋值操作符(按位右移)
			int a = b;
			a += 3;
			b -= 4;
			a *= 5;
			a /= 2;
			a %= 3;
			flags &= new_flags;
			flags ^= new_flags;
			flags |= new_flags;
			flags <<= 2;
			flags >>= 2;
				从 右到左

16	,	逗号操作符	for(i = 0, j = 0; i < 10; i++, j++) ...	从 左到右

12.4 待测试代码

```
//点 a 到直线 bc 的距离
type distoline(point a,point b,point c){
    vec u=a-b,v=c-b;
    return fabs(crossp(u,v))/len(v); //面积除以底
}
//点 a 到线段 bc 的距离
type distoseg(point a,point b,point c){
    vec u=a-b,v=c-b,w=c-a;
    if(sgn(dotp(u,v))<0) return len(u);
    if(sgn(dotp(v,w))<0) return len(w);
    return fabs(crossp(u,v))/len(v);
}
//点 a 在直线 bc 上的投影点
point linepro(point a,point b,point c){
    vec v=c-b;
    return b+v*(dotp(v,a-b)/dotp(v,v));
}
```