读入优化

μ's赛高 (只是各类常用函数的重命名,个人喜好)

无符号

```
void maki(int &s)
{
    s=0;char c=getchar();while(c>'9' || c<'0')c=getchar();
    while(c>='0' && c<='9')s*=10,s+=c-'0',c=getchar();return;
}
```

有符号

```
void maki(int &s)
{
    int f = 1;s=0;char c=getchar();
    while(c>'9' || c<'0'){if(c=='-')f=-1;c=getchar();}
    while(c<='9' && c>='0')s*=10,s+=c-'0',c=getchar();
    s*= f;return;
}
```

离散化

```
int getid(int x)
{
    return lower_bound(v.begin(),v.end(),x)-v.begin()+1;
}
int main()
{
    ...
    sort(v.begin(),v.end());
    v.erase(unique(v.begin(),v.end()),v.end());
    ...
}
```

其中,

```
lower_bound(start,end,x); //返回第一个大于等于x的元素的位置
unique(st,ed); //将所有（相邻的重复元素中的一个元素）移至末尾，返回删除后的末尾地址
v.erase(unique(v.begin(),v.end()),v.end()); //删除所有（相邻元素中的一个）
```

并查姬

初始化

```
for(int i = 1;i <= m;i++)
fa[i] = i;
```

给每个点的fa初始化成本身.

合并

```
void uni(int x,int y)
{
    if( (x=find(x)) != (y=find(y)) )
    fa[y]=x;
    return;
}
```

先判断两个点是否在一个集合里再合并

查询

```
int findx(int x)
{
    if(x!=fa[x])fa[x]=find(fa[x]);
    return fa[x];
}
```

求连通块

求连通块就是去查找维护好的并查姬中,fa[x]==x的结点

```
inline int getkuai()
{
  int cnt=0;
    for(int i=1;i<=n;i++)
    if(fa[i]==i)
    cnt++;
    return cnt;
}
```

完整程序

```
#include <iostream>
using namespace std;
int fa[10005];
int findx(int x)
{
    if(x!=fa[x])fa[x]=find(fa[x]);
    return fa[x];
}
void uni(int x,int y)
{
    if( (x=find(x)) != (y=find(y)) )
    fa[y]=x;
    return;
}
int main()
{
    int m,n;
    cin >> m >> n;
```

```
    for(int i = 1;i <= m;i++)
    fa[i]=i;
    for(int i = 0;i < n;i++)
    {
        int ta,tb,tc;
        cin >> tc >> ta >> tb;
        if(tc == 1)//合并ta和tb
        {
            uni(x,y);
        }else if(tc == 2)//查询ta tb
        {
            ta=findx(ta);
            tb=findx(tb);
            if(ta == tb)
            {
                cout << "Y" << endl;
            }else{
                cout << "N" << endl;
            }
        }
    }
}

```

快速幂

```
lovelive ksm(lovelive a,lovelive k){
    if(k==1) return a%mod;
    if(k==0) return 1%mod;
    lovelive ha = ksm(a,k/2)%mod;
    if(k%2 == 1)return ha*ha*a%mod;
    return ha*ha%mod;
}
```

线性筛素数

主程序

```
for(int i = 2;i <= N;i++)
{
    if(prime[i])
    {
        primes[numprime++] = i;
    }
    for(int j = 0;j < numprime && i * primes[j] <= N;j++)
    {
        prime[i * primes[j]] = false;
        if(i % primes[j] == 0)//与普通筛的区别
        {
            break;
        }
    }
}
```

枚举每一个数i,如果已经被认为是质数,加入素数表中.

并枚举素数表中每一个数p,将i*p筛去,直到i*p > N或**(i%n==0)**

0,1为特例

完整程序

```
#include <iostream>
#include <cstdio>
#define MAXN 1000005
using namespace std;
bool prime[MAXN];
int primes[MAXN];
int numprime = 0;
int main()
{
    for(int i = 0;i < MAXN;i++)
    {
        prime[i] = 1;
    }
    prime[0] = prime[1] = 0;
    int N;
    scanf("%d",&N);
    for(int i = 2;i <= N;i++)
    {
        if(prime[i])
        {
            primes[numprime++] = i;
        }
        for(int j = 0;j < numprime && i * primes[j] <= N;j++)
        {
            prime[i * primes[j]] = false;
            if(i % primes[j] == 0)
            {
                break;
            }
        }
    }
    int M;
    scanf("%d",&M);
    for(int i = 0;i < M;i++)
    {
        int t;
        scanf("%d",&t);
        printf("%s\n",prime[t]?"Yes":"No");
    }
}
```

高精度

clean() 清除前导0

初始值为0 所以构造函数里len=1;

在char* -> BigNumber和BigNumber -> string里

倒序的代码一致:

```
for(int i = 0;i < len;i++)
{
    res[i] = d[len-1-i] +/- '0';
}
```

只有加法可能会出现结果的位数与两个操作数均不一致的情况,所以在加法里需要加一句更新位数:

如果第一位不是0并且位数不对,就更新.

```
if(c.d[i] != 0 && c.len != i+1)c.len=i+1;
```

其他的只要在之前加一句:

```
c.len = max(len,b.len);
```

即可.

完整程序

```
#include <iostream>
#include <cstring>
#include <cstdio>
#include <string>
#define MAXN 1005
using namespace std;
struct BigNumber{
    int d[MAXN],len;
    BigNumber(){memset(d,0,sizeof d);len=1;}
    void clean()
    {
        while(len > 1 && d[len] == 0)len--;
    }
    BigNumber operator = (const char *num)
    {
        memset(d,0,sizeof d);
        len = strlen(num);
        for(int i = 0;i < len;i++)
        {
            d[i] = num[len-i-1] -'0';
        }
        clean();
        return *this;
    }
    BigNumber operator = (const int num)
    {
        char t[20];
        sprintf(t,"%d",num);
        *this = t;
        return *this;
    }
    BigNumber operator + (const BigNumber &b)
    {
        BigNumber c = *this;int i;
```

```
        for(i = 0;i < b.len;i++)
        {
            c.d[i] += b.d[i];
            if(c.d[i] >= 10)c.d[i] %= 10,c.d[i+1]++;
        }
        while(c.d[i] >= 10)c.d[i]%=10,c.d[i+1]++,i++;
        c.len = max(len,b.len);
        if(c.d[i] != 0 && c.len != i+1)c.len=i+1;
        c.clean();
        return c;
    }
    BigNumber operator - (const BigNumber &b)
    {
        BigNumber c = *this;int i;
        for(i = 0;i < b.len;i++)
        {
            c.d[i] -= b.d[i];
            if(c.d[i] < 0)c.d[i] += 10,c.d[i+1]--;
        }
        while(c.d[i] < 0)c.d[i]+=10,c.d[i+1]--,i++;
        c.len = max(len,b.len);
        c.clean();
        return c;
    }
    BigNumber operator * (const BigNumber &b)
    {
        BigNumber c;
        c.len = b.len + len;
        for(int i = 0;i < b.len;i++)
        {
            for(int j = 0;j < len;j++)
            {
                c.d[i+j] = d[j] * b.d[i];
            }
        }
        for(int i = 0;i < c.len;i++)
        {
            c.d[i+1] += c.d[i]/10;
            c.d[i] %= 10;
        }
        c.clean();
        return c;
    }
    string str()
    {
        char res[MAXN] = {};
        for(int i = 0;i < len;i++)
        {
            res[i] = d[len-1-i] + '0';
        }
        return string(res);
    }
};
int main()
{
    BigNumber a;
    BigNumber b;
    a=1,b=2;
```

```
    for(int i = 0;i < 100;i++)
    {
        a = a+b;
        cout << a.str() << endl;
    }
}
```

二分模板

```
int l,r;
while(l + 1 <r)
{
    int mid = ((l+r)>>1);
    if(judge(mid))
    {
        l = mid;
    }else{
        r = mid;
    }
}
ans = l;
```

最长不上升子序列-O(logn)

主要思想是用二分优化"寻找a[j]>=a[i]中f[j]最大的j.

在原有的基础上新开一个数组temp[],temp[i]表示f[j] = i的最大的a[j],翻译成人话就是f值等于i的最大的数是多少.

因为是最长不上升子序列,所以在f值相同时,这个数越大越有价值.

可以发现这个temp是有单调性的.

因为要找到一个最大的i,即f值,所以二分是要注意下一个小细节.

假如按照temp[1]到temp[N]这样从左到右写出来,我们要做的就是找出满足temp[k] >= a[nown]的最右边的值

所以在二分时,如果当前二分的中点值等于a[nown],应当把左区间端点的值设置为mid,而不是mid+1,防止丢失解.

如果二分中点小于a[nown],可以大胆的将右端点设为mid-1.

部分程序

```
int l,r;
while(l + 1 <= r)
{
    int mid = ((l+r)>>1);
    if(temp[l] >= a[nown])
    {
        l = mid;
    }else{
        r = mid-1;
    }
}
if(l < r && temp[r] >= a[nown])l = r;
temp[l+1] = maxx(temp[l+1],a[nown]);
```

STL-字符串匹配

返回一字符串a在字符串s中第一次出现的位置

```
 position = s.find(a);
if (position != s.npos)  //如果没找到，返回npos.
{
    cout << "position is : " << position << endl;
}
else
{
    cout << "Not found the flag"<<endl;
}
```

返回一字符串a中任意字符在字符串s中第一次出现位置

```
 position = s.find_first_of(a);
 cout << "s.find_first_of(a) is : " << position << endl;
```

返回一字符串a在字符串s中任意位置之后的第一次出现位置

```
position=s.find(a,5);
cout<<"s.find(a,5) is : "<<position<<endl;
```

返回一字符串a在字符串s中出现的所有位置(KMP)

```
string s = "aboaboab";
string flag="aboab";
int position=0;
int i=1;
while((position=s.find(flag,position))!=string::npos)
{
    cout<<"position  "<<i<<" : "<<position<<endl;
    position++;
    i++;
}
```

Kruskal求最小生成树

主程序

```
...
sort(e+1,e+M+1,cmp);
...
int kruskal()
{
    int ecnt = 0;
    int val = 0;
    for(int i=1;i<=M;i++)
    {
        if(ecnt>=N-1)break;
        if(findx(e[i].op)==findx(e[i].ed))continue;
        ecnt++;
        uni(e[i].op,e[i].ed);
        val+=e[i].w;
    }
    return val;
}
```

按边权从小到大排序后,枚举每一条边.

利用并查姬维护连通性,发现一条边连接了两个不连通的点,就加入这条边.

直到加入了N-1条边(最小生成树)

或者枚举完了所有边(原图不连通)

完整程序

```
#include<iostream>
#include<algorithm>
#include<cstdio>
#include<cstring>
#define MAXN 5010
#define MAXM 200010
using namespace std;
int N,M;
int fa[MAXN];
void init()
{
    for(int i=1;i<=N;i++)
        fa[i]=i;
    return;
}
int findx(int x)
{
    if(x!=fa[x])fa[x]=find(fa[x]);
    return fa[x];
}
void uni(int x,int y)
{
    if((x=find(x))!=(y=find(y)))
        fa[y]=x;
```

```
    return;
}
struct Edge{
    int op,ed,w;
}e[MAXM];
bool cmp(Edge a,Edge b)
{
    return a.w<b.w;
}
int kruskal()
{
    int ecnt = 0;
    int val = 0;
    for(int i=1;i<=M;i++)
    {
        if(ecnt>=N-1)break;
        if(findx(e[i].op)==findx(e[i].ed))continue;
        ecnt++;
        uni(e[i].op,e[i].ed);
        val+=e[i].w;
    }
    return val;
}
int main()
{
    cin>>N>>M;
    init();
    for(int i=1;i<=M;i++)
    {
      cin>>e[i].op>>e[i].ed>>e[i].w;
    }
    sort(e+1,e+M+1,cmp);
    int flag=kruskal();
    cout<<flag<<endl;
    return 0;
}

```

Dijkstra求单源最短路

初始化

dis全部初始化为INF,距离全部未锁定

主程序

```
void dijkstra()
{
    memset(vis,0,sizeof(vis));
    for(int i = 1;i <= N;i++)
    {
        dis[i] = INF;
    }
    q.push((Node){0,S});
    dis[S] = 0;
    while(!q.empty())
```

```
    {
        Node nown = q.top();
        q.pop();
        if(vis[nown.n])continue;
        for(int i = head[nown.n];i;i = e[i].ne)
        {
            if(dis[e[i].to] > dis[nown.n] + e[i].w)
            {
                dis[e[i].to] = dis[nown.n] + e[i].w;
                q.push((Node){dis[e[i].to],e[i].to});
            }
        }
        vis[nown.n] = true;
    }
}
```

每次取出队列中d值最小的结点,直到队列为空

如果取出的结点距离已锁定,continue;

否则,松弛该节点所有相邻节点的dis值,如果松弛成功,加入队列.

将该节点距离锁定.

完整程序

```
#include <iostream>
#include <cstdio>
#include <cstring>
#include <queue>
#define MAXN 10005
#define MAXM 500005
#define INF 2147483647
using namespace std;
int N,M,S,dis[MAXN];
struct Node{
    int d,n;
};
bool operator < (Node a,Node b)
{
    return a.d > b.d;
}
priority_queue<Node>q;
bool vis[MAXN];
struct Edge{
    int to,w,ne;
}e[MAXM];
int head[MAXN];
int ecnt = 0;
void nico(int u,int v,int w)
{
    e[++ecnt].to = v;
    e[ecnt].w = w;
    e[ecnt].ne = head[u];
    head[u] = ecnt;
    return;
}
void maki(int &s)
```

```
{
    s=0;char c=getchar();while(c>'9' || c<'0')c=getchar();
    while(c>='0' && c<='9')s*=10,s+=c-'0',c=getchar();return;
}
void dijkstra()
{
    memset(vis,0,sizeof(vis));
    for(int i = 1;i <= N;i++)
    dis[i] = INF;
    q.push((Node){0,S});
    dis[S] = 0;
    while(!q.empty())
    {
        Node nown = q.top();
        q.pop();
        if(vis[nown.n])continue;
        for(int i = head[nown.n];i;i = e[i].ne)
        {
            if(dis[e[i].to] > dis[nown.n] + e[i].w)
            {
                dis[e[i].to] = dis[nown.n] + e[i].w;
                q.push((Node){dis[e[i].to],e[i].to});
            }
        }
        vis[nown.n] = 1;
    }
}
int main()
{
    maki(N);maki(M);maki(S);
    for(int i = 1;i <= M;i++)
    {
        int a,b,c;
        maki(a);maki(b);maki(c);
        nico(a,b,c);
    }
    dijkstra();
    for(int i = 1;i <= N;i++)
    {
        cout << dis[i] << " ";
    }
}

```

SPFA求单源最短路

它死了

初始化

dis全部初始化为INF,全部不在队列

主程序

```
void spfa()
{
```

```
    memset(inq,0,sizeof(inq));
    for(int i=1;i<=N;i++)dis[i]=INF;
    q.push(S);
    inq[S] = 1;
    dis[S] = 0;
    while(!q.empty())
    {
        int nown = q.front();
        inq[nown] = 0;
        q.pop();
        for(int i = head[nown];i;i=e[i].ne)
        {
            if(dis[e[i].to] > dis[nown] + e[i].w)
            {
                dis[e[i].to] = dis[nown] + e[i].w;
                if(!inq[e[i].to])
                {
                    q.push(e[i].to);
                    inq[e[i].to] = 1;
                }
            }
        }
    }
}
```

取出队首元素,将其标记为不在队列,直到队列为空.

松弛该节点所有相邻节点的dis值,如果松弛成功并且不在队列,加入队列并标记为在队列.

判负环

在SPFA的时候加一个入队次数的数组,判断当前节点是否入队大于等于n次,如果是那就存在负环

```
#include<iostream>
#include<cstring>
#include<cstdio>
#include<queue>
#define maxn 20100
#define maxm 30100
#define INF 2147483647
using namespace std;
int n,m,dis[maxn],inq[maxn],head[maxn],cnt=0;
int tims[maxn],vis[maxn];
queue<int>q;
struct edge{
  int to,w,ne;
}e[maxm<<1];
inline void nico(int u,int v,int w)
{
  e[++cnt].to=v;
  e[cnt].w=w;
  e[cnt].ne=head[u];
  head[u]=cnt;
  return;
}
inline int SPFA(int s)
{
```

```
  for(register int i=1;i<=n;i++)
  dis[i]=INF;

  q.push(s);
  inq[s]=1;
  dis[s]=0;
  tims[s]++;
  while(!q.empty())
  {
    int cur=q.front();
    q.pop();
    inq[cur]=0;
    vis[cur]=1;
    if(tims[cur]>=n)return 1;
    for(register int i=head[cur];i;i=e[i].ne)
    {
      if(dis[e[i].to]>dis[cur]+e[i].w)
      {
        dis[e[i].to]=dis[cur]+e[i].w;
        if(!inq[e[i].to])
        {
          q.push(e[i].to);
          inq[e[i].to]=1;
          tims[e[i].to]++;
          if(tims[e[i].to]>=n)
          return 1;
        }
      }
    }
  }
  return 0;
}
int main()
{
  int t;
  scanf("%d",&t);
  while(t--)
  {
    scanf("%d%d",&n,&m);
    memset(e,0,sizeof(e));
    memset(head,0,sizeof(head));
    memset(vis,0,sizeof(vis));
    memset(inq,0,sizeof(inq));
    memset(tims,0,sizeof(tims));
    while(!q.empty())q.pop();
    cnt=0;
    int flag=0;
    for(register int i=1;i<=m;i++)
    {
      int u,v,w;
      scanf("%d%d%d",&u,&v,&w);
      if(w<0)
      nico(u,v,w);
      else
      nico(u,v,w),nico(v,u,w);
    }
    if(SPFA(1))
    printf("YE5\n");
```

```
    else
    printf("NO\n");
  }
  return 0;
}

```

完整程序(正常SPFA)

```
#include <iostream>
#include <cstdio>
#include <cstring>
#include <queue>
#define MAXN 10005
#define MAXM 500005
#define INF 2147483647
using namespace std;
int N,M,S,dis[MAXN];
queue<int> q;
bool inq[MAXN],cnt[MAXN];
struct Edge{
    int to,w,ne;
}e[MAXM];
int head[MAXN];
int ecnt = 0;
void nico(int u,int v,int w)
{
    e[++ecnt].to = v;
    e[ecnt].w = w;
    e[ecnt].ne = head[u];
    head[u] = ecnt;
    return;
}
void maki(int &s)
{
    s=0;char c=getchar();while(c>'9' || c<'0')c=getchar();
    while(c>='0' && c<='9')s*=10,s+=c-'0',c=getchar();return;
}
bool spfa()
{
    memset(inq,0,sizeof(inq));
    for(int i=1;i<=N;i++)dis[i]=INF;
    q.push(S);
    inq[S] = 1;
    dis[S] = 0;
    cnt[S]++;
    while(!q.empty())
    {
        int nown = q.front();
        inq[nown] = 0;
        q.pop();
        for(int i = head[nown];i;i=e[i].ne)
        {
            if(dis[e[i].to] > dis[nown] + e[i].w)
            {

```

```
                dis[e[i].to] = dis[nown] + e[i].w;
                if(!inq[e[i].to])
                {
                    inq[e[i].to]=1;
                    cnt[e[i].to]++;
                    if(cnt[e[i].to]>n)
                    return 1;
                    q.push(e[i].to);
                }
            }
        }
    }
    return 0;
}
int main()
{
    maki(N);maki(M);maki(S);
    for(int i = 1;i <= M;i++)
    {
        int a,b,c;
        maki(a);maki(b);maki(c);
        nico(a,b,c);
    }
    if(spfa())
    {
        printf("lzy AK NOIP 2018\n");
      printf("sx AK NOIP 2018\n");
        return 0;
    }
    for(int i = 1;i <= N;i++)
    {
        cout << dis[i] << " ";
    }
}

```

树状数组

and线段树

LowBit

```
int lowb(int x)
{
    return x & (-x);
}
```

区间查询

树状数组只能实现从0开始的区间查询,从当前结点往前爬

```
int sum(int x)
{
    int ans=0;
    while(x!=0)
    {
        ans+=c[x];
        x-=lowb(x);
    }
    return ans;
}
```

单点修改

把每个包含该结点的区间都改一下,往前爬

```
void update(int x,int k)
{
    while(x<=N)
    {
        c[x]+=k;
        x+=lowb(x);
    }
    return;
}
```

区间修改

可以用树状数组维护差分实现区间修改

差分之后,对于区间[L,R],只需单点修改差分数组的b[L]=b[L]+Δ和b[r+1]=b[r+1]-Δ

```
inline void update(int x,int y)
{
  while(x<=n)
  {
    b[x]+=y;
    x+=lowbit(x);
  }
}
```

单点查询

只需要往回求一遍差分数组的前缀和就行了

```
inline int query(int i)
{
  int s=0;
  while(i>0)
  {
    s+=b[i];
    i-=lowbit(i);
  }
  return s;
}
```

完整程序(单点修改和区间查询)

```cpp
#include<iostream>
#include<cstring>
#include<cstdio>
#define maxn 500010
using namespace std;
int n,m;
long long int a[maxn],c[maxn];
inline int lowbit(int x)
{return x&-x;}
inline void add(int x,int y)
{
  while(x<=n)
  {
    c[x]+=y;
    x+=lowbit(x);
  }
  return;
}
inline long long int sum(int x)
{
  long long int ans=0;
  while(x>0)
  {
    ans+=c[x];
    x-=lowbit(x);
  }
  return ans;
}
int main()
{
  scanf("%d %d",&n,&m);
  for(int i=1;i<=n;i++)
  scanf("%lld",&a[i]),add(i,a[i]);
  while(m--)
  {
    int flag;
    scanf("%d",&flag);
    if(flag==1)
    {
      int x,k;
      scanf("%d %d",&x,&k);
      add(x,k);
    }
    else if(flag==2)
    {
      int x,y;
      scanf("%d %d",&x,&y);
      printf("%lld\n",sum(y)-sum(x-1));
    }
  }
  return 0;
}
```

线段树

and树状数组

结构体定义

```
#define lson (o<<1)
#define rson (o<<1|1)
#define mid ((l+r)>>1)

struct SegTree{
    lovelive val,lazy;
}tree[maxn<<2];
```

需要开四倍结点数.

PushDown向下传递Lazy标记

```
void pushdown(int l,int r,int o)
{
    tree[lson].val+=(mid-l+1)*tree[o].lazy;
    tree[rson].val+=(r-mid)*tree[o].lazy;
    tree[lson].lazy+=tree[o].lazy;
    tree[rson].lazy+=tree[o].lazy;
    tree[o].lazy=0;
    return;
}
```

更新一个结点的Lazy值的同时,就要更新该结点的val值.

建树

```
void buildtree(int l,int r,int o)
{
    if(l==r)
    {
        tree[o].val=a[l];
        return;
    }
    buildtree(l,mid,lson);
    buildtree(mid+1,r,rson);
    tree[o].val=tree[lson].val+tree[rson].val;
    return;
}
```

递归建树后应向上更新val值

叶子节点要return

单点/区间查询

```
lovelive query(int ql,int qr,int l,int r,int o)
{
    if(ql>r || qr<l)return 0;
    if(ql<=l && qr>=r)return tree[o].val;
    pushdown(l,r,o);
    return query(ql,qr,l,mid,lson)+query(ql,qr,mid+1,r,rson);
}
```

单点/区间修改

```
void update(int ql,int qr,int l,int r,int o,int addval)
{
    if(ql>r || qr<l)return;
    if(ql<=l && r<=qr)
    {
        tree[o].val+=(r-l+1)*addval;
        tree[o].lazy+=addval;
        return;
    }
    pushdown(l,r,o);
    update(ql,qr,l,mid,lson,addval);
    update(ql,qr,mid+1,r,rson,addval);
    tree[o].val=tree[lson].val+tree[rson].val;
    return;
}
```

完整程序

```
#include<iostream>
#include<cstdio>
#include<cstring>
#define maxn 100010
#define lson (o<<1)
#define rson (o<<1|1)
#define mid ((l+r)>>1)
#define lovelive long long int
using namespace std;
struct segtree{
  lovelive val,lazy;
}tree[maxn<<2];
int n,m;
lovelive a[maxn];
inline void pushdown(int l,int r,int o)
{
  if(tree[o].lazy==0)
  return;
  tree[lson].val+=(mid-l+1)*tree[o].lazy;
  tree[rson].val+=(r-mid)*tree[o].lazy;
  tree[lson].lazy+=tree[o].lazy;
  tree[rson].lazy+=tree[o].lazy;
  tree[o].lazy=0;
  return;
}
inline void buildtree(int l,int r,int o)
{
```

```
  if(l==r)
  {
    tree[o].val=a[l];
    return;
  }
  buildtree(l,mid,lson);
  buildtree(mid+1,r,rson);
  tree[o].val=tree[lson].val+tree[rson].val;
  return;
}
inline void update(int ql,int qr,int l,int r,int o,int add)
{
  if(l>=ql && r<=qr)
  {
    tree[o].val+=(r-l+1)*add;
    tree[o].lazy+=add;
    return;
  }
  pushdown(l,r,o);
  if(ql<=mid) update(ql,qr,l,mid,lson,add);
  if(qr>mid) update(ql,qr,mid+1,r,rson,add);
  tree[o].val=tree[lson].val+tree[rson].val;
  return;
}
inline lovelive query(int ql,int qr,int l,int r,int o)
{
  if(l>=ql && r<=qr) return tree[o].val;
  pushdown(l,r,o);
  lovelive ans=0;
  if(ql<=mid) ans+=query(ql,qr,l,mid,lson);
  if(qr>mid) ans+=query(ql,qr,mid+1,r,rson);
  return ans;
}
int main()
{
  scanf("%d%d",&n,&m);
  for(int i=1;i<=n;i++)
  scanf("%lld",&a[i]);
  buildtree(1,n,1);
  while(m--)
  {
    int flag;
    scanf("%d",&flag);
    if(flag==1)
    {
      int l,r,add;
      scanf("%d%d%d",&l,&r,&add);
      update(l,r,1,n,1,add);
    }
    if(flag==2)
    {
      int l,r;
      scanf("%d%d",&l,&r);
      printf("%lld\n",query(l,r,1,n,1));
    }
  }
  return 0;
}
```

ST表RMQ(求区间极值)

考场一遍过(祭奠NOIP2018)

生成对数数组

```
lg[1] = 0;
for(int i=2;i<=N;i++)
{
    lg[i]=lg[i>>1]+1;
}
```

初始化

```
for(int i=1;i<=n;i++)
    f[0][i]=a[i];
for(int j=1;(1<<j)<=n;j++)
{
    for(int i=1;i+(1<<j)-1<=n;i++)
    f[j][i]=honoka(f[j-1][i],f[j-1][i+(1<<(j-1))]);
}
```

查询区间最大值

```
int que(int x,int y)
{
  int k=lg[y-x+1];
    return honoka(f[k][x],f[k][y-(1<<k)+1]);
}
```

完整程序

```
#include <iostream>
#include <cstdio>
#define MAXN 100005
using namespace std;
int lg[MAXN],f[maxw][MAXN];
int N,M;
inline int honoka(int a,int b)
{
    return a>b?a:b;
}
void init(int *a,int n)
{
    lg[1]=0;
    for(int i=2;i<=n;i++)
    lg[i]=lg[i>>1]+1;

    for(int i=1;i<=n;i++)
    f[0][i]=a[i];
    for(int j=1;(1<<j)<=n;j++)
    {
```

```
        for(int i=1;i+(1<<j)-1<=n;i++)
        f[j][i]=honoka(f[j-1][i],f[j-1][i+(1<<(j-1))]);
    }
    return;
}
int que(int x,int y)
{
    int k=lg[y-x+1];
    return honoka(f[k][x],f[k][y-(1<<k)+1]);
}
int main()
{
    scanf("%d%d",&N,&M);
    init();
    for(int i=0;i<M;i++)
    {
        int x,y;
        scanf("%d%d",&x,&y);
        printf("%d\n",que(x,y));
    }
}
```

树上倍增求LCA(最近公共祖先)

[一整个二晚的付出]

初始化

求出每个点的深度,以及每个点往上跳2^i步能跳到的点

```
void dfs(int u,int father)
{
  depth[u]=depth[father]+1;
  fa[u][0]=father;

  for(int i=1;(1<<i)<=depth[u];i++)
  fa[u][i]=fa[fa[u][i-1]][i-1];

  for(int i=head[u];i;i=e[i].ne)
  {
    if(e[i].to!=father)
    dfs(e[i].to,u);
  }
  return;
}
```

跳表查询

```
int LCA(int u,int v)
{
  if(depth[u]<depth[v])
  umi(u,v);//如果深度不同就交换,使u的深度始终大于v

  while(depth[u]>depth[v])
  u=fa[u][lg[depth[u]-depth[v]]];//跳到同一高度
```

```
  if(u==v)
  return v;//如果此时相等,说明v使最近公共祖先

  for(int k=lg[depth[u]];k>=0;k--)
  {
    if(fa[u][k]!=fa[v][k])//如果往上跳2^k步到达的点不同那就往上跳
    u=fa[u][k],v=fa[v][k];
  }
  return fa[u][0];
}
```

完整程序

```
#include<iostream>
#include<cstdio>
#include<cstring>
#define maxn 500050
#define maxm 500050
#define maxd 20
using namespace std;
struct edge{
  int to,ne;
}e[maxn<<1];
int lg[maxn],head[maxn],cnt=0,depth[maxn],fa[maxn][maxd];
inline void maki(int &s)
{
  s=0;char c=getchar();while(c>'9' || c<'0') c=getchar();
  while(c<='9' && c>='0')s*=10,s+=c-'0',c=getchar();return;
}
inline void nico(int u,int v)
{
  e[++cnt].to=v;
  e[cnt].ne=head[u];
  head[u]=cnt;
  return;
}
inline void umi(int &a,int &b)
{int t=a;a=b;b=t;return;}
inline void log(int n)
{
  lg[1]=0;
  for(int i=2;i<=n;i++)
  lg[i]=lg[i>>1]+1;
  return;
}
void dfs(int u,int father)
{
  depth[u]=depth[father]+1;
  fa[u][0]=father;

  for(int i=1;(1<<i)<=depth[u];i++)
  fa[u][i]=fa[fa[u][i-1]][i-1];

  for(int i=head[u];i;i=e[i].ne)
  {
    if(e[i].to!=father)
```

```
    dfs(e[i].to,u);
  }
  return;
}
int LCA(int u,int v)
{
  if(depth[u]<depth[v])
  umi(u,v);

  while(depth[u]>depth[v])
  u=fa[u][lg[depth[u]-depth[v]]];

  if(u==v)
  return v;

  for(int k=lg[depth[u]];k>=0;k--)
  {
    if(fa[u][k]!=fa[v][k])
    u=fa[u][k],v=fa[v][k];
  }
  return fa[u][0];
}
int main()
{
  int n,m,s;
  maki(n),maki(m),maki(s);
  for(int i=1;i<n;i++)
  {
    int u,v;
    maki(u),maki(v);
    nico(u,v);
    nico(v,u);
  }
  log(n);

  memset(depth,0,sizeof(depth));
  dfs(s,0);

  while(m--)
  {
    int u,v;
    maki(u),maki(v);
    int f=LCA(u,v);
    printf("%d\n",f);
  }
  return 0;
}
```

树链剖分

用两个dfs函数,第一个处理出每个结点的父亲,深度,重儿子和子树大小

第二个处理按照第一个dfs留下的各类信息给树上结点重新编号,以方便建线段树

按照重儿子的顺序走能走出一条连续的链,对于每个轻儿子它又会是一条新链的顶端

所以dfs2时先把整棵树最大的重儿子链跑出来,再分别处理轻儿子即可

```cpp
inline void dfs1(int x,int f,int deep)//当前结点,父亲,深度
{
  depth[x]=deep;
  fa[x]=f;
  scale[x]=1;
  //记录父亲,深度,子树
  int maxson=-1;//记录子树中重儿子的规模

  for(int i=head[x];i;i=e[i].ne)
  {
    if(e[i].to==f) continue;//不往回走
    dfs1(e[i].to,x,deep+1);//处理儿子
    scale[x]+=scale[e[i].to];//合并子树大小
    if(scale[e[i].to]>maxson)//更新重儿子
    hson[x]=y,maxson=scale[y];
  }
  return;
}
```

```cpp
inline void dfs2(int x,int topp)//当前结点,当前结点所在链的顶点
{
  id[x]=++dfn;
  w[dfn]=weight[x];
  top_p[x]=topp;
  //记录dfs序,把点权赋值到线段树上,记录链顶
  if(!hson[x]) return;//没有重儿子(即没有儿子)就返回
  dfs2(hson[x],topp);//处理重儿子
  for(int i=head[x];i;i=e[i].ne)
  {
    if(e[i].to==fa[x] || e[i].to==hson[x])continue;//往回走或者重儿子就continue
    dfs2(e[i].to,e[i].to);//处理轻儿子
  }
}
```

然后考虑如何修改这个线段树

首先是较简单的子树修改,考虑到每个子树内的所有节点编号一定连续,所以直接在线段树上动即可

每棵子树对应的区间是$[id[x],id[x]+scale[x]-1]$,前面统计子树大小的信息就有用了

```cpp
inline int query_sontree(int x)
{return query(id[x],id[x]+scale[x]-1,1,n,1)%lzy;}
inline void update_sontree(int x,int add)
{update(id[x],id[x]+scale[x]-1,1,n,1,add);return;}
```

其次是难一点的路径修改

既然已经把树分成链了,那就应该通过链来修改树

当两个点在一条链里面的时候,直接维护即可

如果不在一条链里面,那我们就让他们跳到一条链里

首先保持x一直是链顶结点深度较深的点,那每次让x跳到它链顶的结点的父亲上,然后维护这条链的信息即可

其次在一条链上的时候,直接维护就行

```
inline int query_road(int x,int y)
{
  int ans=0;
  while(top_p[x]<top_p[y])
  {
    if(depth[top_p[x]]<depth[top_p[y]])
    umi(x,y);
    ans+=query(id[top_p[x]],id[x],1,n,1);
    ans%=lzy;
    x=fa[top_p[x]];
  }

  if(depth[x]>depth[y])
  umi(x,y);
  ans+=query(id[x],id[y],1,n,1);
  return ans%lzy;
}
inline void update_road(int x,int y,int add)
{
  while(top_p[x]!=top_[y])
  {
    if(depth[top_p[x]]<depth[top_p[y]])
    umi(x,y);
    update(id[top_p[x]],id[x],1,n,1,add);
    x=fa[top_p[x]];
  }

  if(depth[x]>depth[y])
  umi(x,y);
  update(id[x],id[y],1,n,1,add);
  return;
}
```

完整代码:

```
#include<iostream>
#include<cstring>
#include<cstdio>
#define maxn 100010
#define lson (o<<1)
#define rson (o<<1|1)
#define mid ((l+r)>>1)
using namespace std;
struct segtree{
  int val,lazy;
}tree[maxn<<2];
struct edge{
  int from,to,ne;
}e[maxn<<1];
int n,m,r,lzy;
int head[maxn],cnt=0;
int weight[maxn],w[maxn];
int hson[maxn],fa[maxn],depth[maxn],dfn=0,scale[maxn];
int id[maxn],top_p[maxn];
inline void maki(int &s)
{
  s=0;char c=getchar();while(c<'0' || c>'9') c=getchar();
```

```
  while(c>='0'&& c<='9') s*=10,s+=c-'0',c=getchar();return;
}
inline void nico(int u,int v)
{
  e[++cnt].from=u;
  e[cnt].to=v;
  e[cnt].ne=head[u];
  head[u]=cnt;
  return;
}
inline void umi(int &a,int &b)
{int t=a;a=b;b=t;return;}
inline void pushdown(int l,int r,int o)
{
  tree[lson].val+=(mid-l+1)*tree[o].lazy;
  tree[rson].val+=(r-mid)*tree[o].lazy;
  tree[lson].lazy+=tree[o].lazy;
  tree[rson].lazy+=tree[o].lazy;
  tree[o].lazy=0;
  return;
}
inline void buildtree(int l,int r,int o)
{
  tree[o].lazy=0;
  if(l==r)
  {
    tree[o].val=w[l]%lzy;
    return;
  }
  buildtree(l,mid,lson);
  buildtree(mid+1,r,rson);
  tree[o].val=(tree[lson].val+tree[rson].val)%lzy;
  return;
}
inline void update(int ql,int qr,int l,int r,int o,int add)
{
  if(ql>r || qr<l) return;
  if(l>=ql && r<=qr)
  {
    tree[o].val=(tree[o].val+(r-l+1)*add)%lzy;
    tree[o].lazy=(tree[o].lazy+add)%lzy;
    return;
  }
  pushdown(l,r,o);
  update(ql,qr,l,mid,lson,add);
  update(ql,qr,mid+1,r,rson,add);
  tree[o].val=(tree[lson].val+tree[rson].val)%lzy;
  return;
}
inline int query(int ql,int qr,int l,int r,int o)
{
  if(ql>r || qr<l) return 0;
  if(l>=ql && r<=qr) return tree[o].val;
  pushdown(l,r,o);
  return (query(ql,qr,l,mid,lson)+query(ql,qr,mid+1,r,rson))%lzy;
}
inline void dfs1(int x,int f,int deep)
{
```

```
  depth[x]=deep;
  fa[x]=f;
  scale[x]=1;
  int maxson=-1;
  for(int i=head[x];i;i=e[i].ne)
  {
    if(e[i].to==f) continue;
    dfs1(e[i].to,x,deep+1);
    scale[x]+=scale[e[i].to];
    if(scale[e[i].to]>maxson)
    hson[x]=e[i].to,maxson=scale[e[i].to];
  }
  return;
}
inline void dfs2(int x,int topp)
{
  id[x]=++dfn;
  w[dfn]=weight[x];
  top_p[x]=topp;
  if(!hson[x]) return;
  dfs2(hson[x],topp);
  for(int i=head[x];i;i=e[i].ne)
  {
    if(e[i].to==fa[x] || e[i].to==hson[x])continue;
    dfs2(e[i].to,e[i].to);
  }
  return;
}
inline int query_road(int x,int y)
{
  int ans=0;
  while(top_p[x]!=top_p[y])
  {
    if(depth[top_p[x]]<depth[top_p[y]])
    umi(x,y);
    ans+=query(id[top_p[x]],id[x],1,n,1);
    ans%=lzy;
    x=fa[top_p[x]];
  }
  if(depth[x]>depth[y])
  umi(x,y);
  ans+=query(id[x],id[y],1,n,1);
  return ans%lzy;
}
inline void update_road(int x,int y,int add)
{
  while(top_p[x]!=top_p[y])
  {
    if(depth[top_p[x]]<depth[top_p[y]])
    umi(x,y);
    update(id[top_p[x]],id[x],1,n,1,add);
    x=fa[top_p[x]];
  }
  if(depth[x]>depth[y])
  umi(x,y);
  update(id[x],id[y],1,n,1,add);
  return;
}
```

```
inline int query_sontree(int x)
{return query(id[x],id[x]+scale[x]-1,1,n,1)%lzy;}
inline void update_sontree(int x,int add)
{update(id[x],id[x]+scale[x]-1,1,n,1,add);return;}
int main()
{
  maki(n),maki(m),maki(r),maki(lzy);
  for(int i=1;i<=n;i++)
  maki(weight[i]);
  for(int i=1;i<n;i++)
  {
    int u,v;
    maki(u),maki(v);
    nico(u,v);
    nico(v,u);
  }
  dfs1(r,0,1);
  dfs2(r,r);
  buildtree(1,n,1);
  while(m--)
  {
    int flag;
    maki(flag);
    if(flag==1)
    {
      int x,y,z;
      scanf("%d%d%d",&x,&y,&z);
      update_road(x,y,z%lzy);
    }
    else if(flag==2)
    {
      int x,y;
      scanf("%d%d",&x,&y);
      printf("%d\n",query_road(x,y));
    }
    else if(flag==3)
    {
      int x,z;
      scanf("%d%d",&x,&z);
      update_sontree(x,z%lzy);
    }
    else if(flag==4)
    {
      int x;
      scanf("%d",&x);
      printf("%d\n",query_sontree(x));
    }
  }
  return 0;
}
```

树链剖分求LCA(最近公共祖先)

此方法常数极小

在进行询问链时,最后一步中较浅的点即为LCA.

完整程序

```cpp
#include <iostream>
#include <vector>
#include <algorithm>
#include <cstring>
#include <cstdio>
#define MAXN 500005
#define lovelive long long int
using namespace std;
int N,M,R;
int son[MAXN],l[MAXN],siz[MAXN];
int top[MAXN],fa[MAXN],dep[MAXN],treew[MAXN];
int head[MAXN];
struct Edge{
    int to,ne;
}e[2*MAXN];
int ecnt = 0;
int addedge(int x,int y)
{
    e[++ecnt].to = y;
    e[ecnt].ne = head[x];
    head[x] = ecnt;
}
int dfs1(int x)
{
    int tsize = 0;
    for(int i = head[x];i;i=e[i].ne)
    {
        int to = e[i].to;
        if(to == fa[x])continue;
        fa[to] = x;
        dep[to] = dep[x]+1;
        dfs1(to);
        tsize += siz[to];
        if(son[x] == -1)
        {
            son[x] = to;
        }else if(siz[son[x]] < siz[to]){
            son[x] = to;
        }
    }
    siz[x] = tsize+1;
    return 0;
}
int xu = 0;
int dfs2(int x,int ttop)
{
    l[x] = ++xu;
    top[x] = ttop;
    if(son[x] != -1)
    {
        dfs2(son[x],ttop);
    }

    for(int i = head[x];i;i=e[i].ne)
    {
```

```
        int to = e[i].to;
        if(to == fa[x] || to == son[x])continue;
        dfs2(to,to);
    }
    return 0;
}
int poque(int x,int y)
{
    int ans = 0;

    while(top[x] != top[y])
    {
        if(dep[top[x]] < dep[top[y]])
        {
            swap(x,y);
        }
        x = fa[top[x]];
    }
    if(dep[x] < dep[y])
    {
        swap(x,y);
    }
    return y;
}
void maki(int &s){
    s=0;char ch=getchar();
    while(ch<'0'||ch>'9')ch=getchar();
    while(ch>='0'&&ch<='9'){s=s*10+ch-'0';ch=getchar();}
    return;
}
int main()
{
    memset(l,0,sizeof(l));
    memset(son,-1,sizeof(son));
    memset(dep,0,sizeof(dep));
    memset(siz,0,sizeof(siz));
    memset(top,0,sizeof(top));
    memset(fa,0,sizeof(fa));
    maki(N);maki(M);maki(R);
    for(int i = 1;i <= N-1;i++)
    {
        int op,ed;
        maki(op);
        maki(ed);
        addedge(op,ed);
        addedge(ed,op);
    }
    dfs1(R);
    dfs2(R,R);
    for(int i = 0;i < M;i++)
    {
        int a,b;
        maki(a);
        maki(b);
        printf("%d\n",poque(a,b));
    }
}
```

Tarjan求强连通分量(+DAG)

Tarjan+DAG

对于一些存在依赖关系的模型，若其建图是一个DAG，则可以直接通过拓扑排序解决，但若其中有环则需要特殊处理

对于有环的问题，会出现一些互相依赖的关系，这些关系组成了一个强连通分量，根据题目要求的性质，对于这个强连通分量可以将其缩为一个点

将所有强连通分量缩成点后即可在DAG上求解

建模方式和DAG很相似，建出图不是DAG就先跑一边SCC缩点即可

在一个有向有环图上DFS，找出每一个强连通分量

考虑每一个强连通分量高度最低（离根近）的那个点

这些点将DFS树分割成了许多个子树，每个子树中的点组成了一个强连通分量

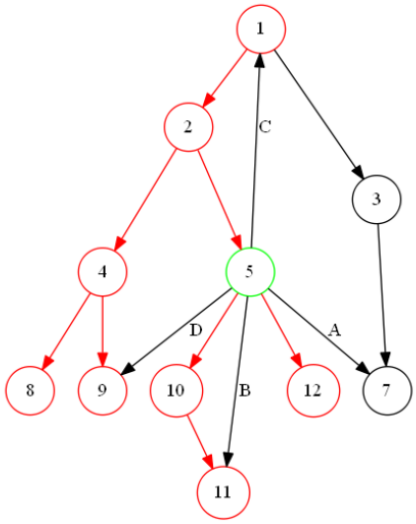
分割的方法是在dfs同时另外维护一个栈存放节点，离开分割点时把分割点往下的部分全部取出来就是一个强连通分量。

现在需要找到这些分割点

维护一个数组low，low[u]代表点u所能到达的，深度最小的点的dfs序编号

初始low[u]=dfn[u] 对于边(u,v) 若为树枝边，则用low[v]更新 若为后向边，则用dfn[v]更新 若为前向边，因为指向的点的信息已经通过树枝边传递过来，所以无需更新 若为横叉边，则指向另一个强连通分量，无需更新 当low[u] == dfn[u]时，就是一个分割点

判定方式



- 红色标记为DFS树
- 记录dfs序
- A边终点未访问过，为树枝边
- B边终点已被访问过，且dfn[v]>dfn[u]，说明在子树中，说明为前向边
- C边终点已被访问过且不在子树中，终点在栈中则为后向边
- D边终点已被访问过且不在子树中且已经出栈，为横叉边

默写流程

1.变量定义

(1).第一组图

(2).缩点及Tarjan

(3).第二组图

(4).全局和答案

3.加边，最大最小，快读

4.主函数

5.Tarjan

注意变量名别打错，此处不使用e数组，使用名字较长的ed数组

6.主函数的建图

此时根据遍历原图数组，由d数组记录的关系建出新图e

7.topo_sort

注意在所有路径里选取一条最长的

8.调试

完整程序

加入了缩点和DAG上的dp.

```
#include<iostream>
#include<cstring>
#include<cstdio>
#include<stack>
#include<queue>
#define maxn 10010
#define maxm 100010
using namespace std;
struct edge{
  int from,to,ne;
};
edge ed[maxm];
int e1cnt=0,head[maxn];
int d[maxn],weight[maxn];
edge e[maxm];
int ecnt=0,h[maxn];
stack<int>s;
int DFN[maxn],LOW[maxn],SCC[maxn],scccnt=0,cnt=0;
int n,m,ans=0,dis[maxn],indo[maxn];
inline void nico(int u,int v)
{
  ed[++e1cnt].to=v;
  ed[e1cnt].from=u;
  ed[e1cnt].ne=head[u];
  head[u]=e1cnt;
  return;
}
inline void maki(int &ss)
{
  ss=0;char c=getchar(); while(c<'0' || c>'9') c=getchar();
  while(c<='9' && c>='0')ss*=10,ss+=c-'0',c=getchar();return;
}
inline int kotori(int a,int b)
{return a<b?a:b;}
inline int honoka(int a,int b)
{return a>b?a:b;}
void tarjan(int x)
{
```

```
  DFN[x]=LOW[x]= ++cnt;
  s.push(x);
  for(int i=head[x];i;i=ed[i].ne)
  {
    if(!DFN[ed[i].to])
    {
      tarjan(ed[i].to);
      LOW[x]=kotori(LOW[x],LOW[ed[i].to]);
    }
    else if(!SCC[ed[i].to])
    LOW[x]=kotori(LOW[x],DFN[ed[i].to]);
  }

  if(LOW[x]==DFN[x])
  {
    scccnt++;
    int y;
    while(y=s.top())
    {
      s.pop();
      SCC[y]=scccnt;
      d[y]=x;
      if(x==y) break;
      weight[x]+=weight[y];
    }
  }
  return;
}
void topo()
{
  queue<int>q;
  for(int i=1;i<=n;i++)
  {
    if(d[i]==i && !indo[i])
    {
      q.push(i);
      dis[i]=weight[i];
    }
  }
  while(!q.empty())
  {
    int cur=q.front();
    q.pop();
    for(int i=h[cur];i;i=e[i].ne)
    {
      dis[e[i].to]=honoka(dis[e[i].to],dis[cur]+weight[e[i].to]);
      indo[e[i].to]--;
      if(indo[e[i].to]==0)
      q.push(e[i].to);
    }
  }
  for(int i=1;i<=n;i++)
  ans=honoka(ans,dis[i]);
}
int main()
{
  maki(n),maki(m);
  for(int i=1;i<=n;i++)
```

```
  maki(weight[i]);
  for(int i=1;i<=m;i++)
  {
    int u,v;
    maki(u),maki(v);
    nico(u,v);
  }
  memset(DFN,0,sizeof(DFN));
  memset(LOW,0,sizeof(LOW));
  for(int i=1;i<=n;i++)
  {
    if(!DFN[i])
    tarjan(i);
  }
  for(int i=1;i<=m;i++)
  {
    int x=d[ed[i].from],y=d[ed[i].to];
    {
      if(x!=y)
      {
        e[++ecnt].to=y;
        e[ecnt].from=x;
        e[ecnt].ne=h[x];
        h[x]=ecnt;
        indo[y]++;
      }
    }
  }
  topo();
  printf("%d\n",ans);
  return 0;
}
```

矩阵快速幂

利用矩阵快速幂可以快速转移斐波那契数列的递推公式

$$\left(\begin{bmatrix}1 & 1\\ 1 & 0\end{bmatrix}\text{^(n-1))*}\begin{bmatrix}f2\\ f1\end{bmatrix}\right.$$

主程序

```
struct mat{
    lovelive ma[5][5];
    int h,l;
    mat(){memset(ma,0,sizeof ma);h=0;l=0;}
    mat operator * (mat b)
    {
        mat c;
        c.l = l;
        c.h = b.h;
        for(int i = 1;i <= l;i++)
        {
            for(int j = 1;j <= b.h;j++)
            {
                for(int k = 1;k <= h;k++)
```

```
                {
                    c.ma[i][j] += ma[i][k] * b.ma[k][j];
                    c.ma[i][j] %= mod;
                }
            }
        }
        return c;
    }
};
mat ksm(mat d,lovelive n)
{
    if(n == 1)return d;
    mat c = ksm(d,n/2);
    c.h = c.l = d.h;
    if(n%2==0)
    {
        return c*c;
    }else{
        return c*c*d;
    }
}
```

完整程序

```
#include <iostream>
#include <cstring>
#define lovelive long long int
using namespace std;
const lovelive mod = 1e9+7;
struct mat{
    lovelive ma[5][5];
    int h,l;
    mat(){memset(ma,0,sizeof ma);h=0;l=0;}
    mat operator * (mat b)
    {
        mat c;
        c.l = l;
        c.h = b.h;
        for(int i = 1;i <= l;i++)
        {
            for(int j = 1;j <= b.h;j++)
            {
                for(int k = 1;k <= h;k++)
                {
                    c.ma[i][j] += ma[i][k] * b.ma[k][j];
                    c.ma[i][j] %= mod;
                }
            }
        }
        return c;
    }
};
mat ksm(mat d,lovelive n)
{
    if(n == 1)return d;
    mat c = ksm(d,n/2);
    c.h = c.l = d.h;
```

```
    if(n%2==0)
    {
        return c*c;
    }else{
        return c*c*d;
    }
}
int main()
{
    lovelive N;
    cin >> N;
    if(N == 1)
    {
        cout << 1;
        return 0;
    }
    mat c;
    c.h = c.l = 2;
    c.ma[1][1] = 1;
    c.ma[1][2] = 1;
    c.ma[2][1] = 1;
    c.ma[2][2] = 0;
    mat d;
    d.h = 2,d.l = 1;
    d.ma[1][1] = 1;
    d.ma[2][1] = 1;
    c = ksm(c,N-1);
    d = d * c;
    cout << d.ma[1][1] << endl;
    return 0;
}
```

平衡树(treap)

结构体定义

参数分别为子树大小

当前节点的值

当前节点所代表的数有几个

优先级w

左右儿子

```
#define ls t[x].ch[0]
#define rs t[x].ch[1]

struct treap{
  int size,val,cnt,w,son[2];
}t[maxn];
```

向上更新结点大小

```
inline void pushup(int x)
{t[x].size=t[ls].size+t[rs].size+t[x].cnt;return;}
```

左右旋

d=0表示左旋，首先保存根的右儿子，然后用右儿子的左儿子更新根的右儿子，再把根挂到右儿子的左儿子上

d=1表示右旋，首先保存根的左儿子，然后用左儿子的右儿子更新根的左儿子，再把根挂到左儿子的右儿子上

随后pushup

```
inline void maware(int &x,int d)
{
  int son=t[x].son[d^1];//0左旋,先保存根的右儿子
  t[x].son[d^1]=t[son].son[d];//用右儿子的左儿子更新根的右儿子
  t[son].son[d]=x;//然后让根变成右儿子的左儿子
  pushup(x);
  pushup(x=son);
  return;
}
```

插入结点

如果是空节点，那就新建

如果插入的值和当前节点值相等那就记录

然后寻找子树并且维持treap的性质

如果去了右子树，并且更新之后根的优先级更小，那就左旋调整，把右儿子换上来

```
inline void insert(int &x,int val)
{
  if(!x)
  {
    x=++cnt;
    t[x].size=t[x].cnt=1;
    t[x].val=val;
    t[x].w=rand();
    return;
  }
  if(t[x].val==val)
  {
    t[x].size++;
    t[x].cnt++;
    return;
  }
  int d=(val>t[x].val);//大于的话d为1，去右边了
  insert(t[x].son[d],val);
  if(t[x].w<t[t[x].son[d]].w) maware(x,d^1);//如果小于对应儿子的优先级，那就旋转，左旋是把右儿子替换上来
  pushup(x);
  return;
}
```

删除结点

空节点返回，大于去右边，小于去左边

相等如果是叶子节点就直接删

有一个子节点就把节点转上来然后去子树解决

如果有两个子节点，就把优先级大的转上来，然后去另一边解决，因为左旋之后根去了左子树，右旋之后根去了右子树

```
inline void del(int &x,int val)
{
  if(!x) return;
  if(val<t[x].val) del(ls,val);
  else if(val>t[x].val) del(rs,val);
  else
  {
    if(!ls && !rs)
    {
      t[x].cnt--;t[x].size--;
      if(t[x].cnt==0) x=0;
    }
    else if(ls && !rs)
    {
      maware(x,1);
      del(rs,val);
    }
    else if(!ls && rs)
    {
      maware(x,0);
      del(ls,val);
    }
    else if(ls && rs)
    {
      int d=t[ls].w>t[rs].w;//两个儿子都存在，哪边优先级大就向另一边旋转然后去删除对应儿子
      maware(x,d);
      del(t[x].son[d],val);
    }
  }
  pushup(x);
  return;
}
```

查询数的最小Rank

空节点返回，相等左子树大小+1

如果根比val小那就得去右子树并且要加上左子树大小和根的cnt

如果根比val大那就去左子树

```
inline int rank(int x,int val)
{
  if(!x) return 0;
  if(t[x].val==val) return t[ls].size+1;
  if(t[x].val<val) return t[ls].size+t[x].cnt+rank(rs,val);
  if(t[x].val>val) return rank(ls,val);
}
```

查询某一Rank的数

空节点返回

左子树节点数大于等于rk那解在左子树

左子树加根比x小，解在右子树，并且新的rk为rk-t[x].cnt-t[ls].size

左子树加根节点大于等于rk，再加上刚刚排除了左子树大于等于rk的情况，说明当前节点就是要查询的值

```
inline int find(int x,int rk)
{
  if(!x) return 0;
  if(t[ls].size>=rk) return find(ls,rk);
  else if(t[ls].size+t[x].cnt<rk) return find(rs,rk-t[x].cnt-t[ls].size);
  else return t[x].val;
}
```

查询一数的前驱

前驱就是小于x的最大数

空节点无前驱

如果当前节点值比要查询的值大，那右边的值肯定都大于val，去左子树找答案

否则的话解在根或右子树，取个最大值就行

```
inline int pre(int x,int val)
{
  if(!x) return -inf;
  if(t[x].val>=val) return pre(ls,val);
  else return honoka(t[x].val,pre(rs,val));
}
```

查询一数的后继

后继就是大于x的最小数

空节点无后继

如果当前节点值比要查询的值小，那左边的值都更小，直接去右子树找答案

否则解在根或者左子树，取个较小值就行

```
inline int suc(int x,int val)
{
  if(!x) return inf;
  if(t[x].val<=val) return suc(rs,val);
  else return kotori(t[x].val,suc(ls,val));
}
```

完整程序

```
#include<bits/stdc++.h>
#define ls t[x].son[0]
#define rs t[x].son[1]
#define maxn 100010
#define inf 2147483645
using namespace std;
int cnt=0,root=0;
struct treap{
  int size,val,cnt,w,son[2];
}t[maxn];
inline int honoka(int a,int b)
{return a>b?a:b;}
inline int kotori(int a,int b)
{return a<b?a:b;}
inline void pushup(int x)
{t[x].size=t[ls].size+t[rs].size+t[x].cnt;return;}
inline void maware(int &x,int d)
{
  int son=t[x].son[d^1];//0左旋,先保存根的右儿子
  t[x].son[d^1]=t[son].son[d];//用右儿子的左儿子更新根的右儿子
  t[son].son[d]=x;//然后让根变成右儿子的左儿子
  pushup(x);
  pushup(x=son);
  return;
}
inline void insert(int &x,int val)
{
  if(!x)
  {
    x=++cnt;
    t[x].size=t[x].cnt=1;
    t[x].val=val;
    t[x].w=rand();
    return;
  }
  if(t[x].val==val)
  {
    t[x].size++;
    t[x].cnt++;
    return;
  }
  int d=(val>t[x].val);//大于的话d为1，去右边了
  insert(t[x].son[d],val);
  if(t[x].w<t[t[x].son[d]].w) maware(x,d^1);//如果小于对应儿子的优先级，那就旋转，左旋是把右儿子替换上来
  pushup(x);
  return;
```

```
}
inline void del(int &x,int val)
{
  if(!x) return;
  if(val<t[x].val) del(ls,val);
  else if(val>t[x].val) del(rs,val);
  else
  {
    if(!ls && !rs)
    {
      t[x].cnt--;t[x].size--;
      if(t[x].cnt==0) x=0;
    }
    else if(ls && !rs)
    {
      maware(x,1);
      del(rs,val);
    }
    else if(!ls && rs)
    {
      maware(x,0);
      del(ls,val);
    }
    else if(ls && rs)
    {
      int d=t[ls].w>t[rs].w;//两个儿子都存在，哪边优先级大就向另一边旋转然后去删除对应儿子
      maware(x,d);
      del(t[x].son[d],val);
    }
  }
  pushup(x);
  return;
}
inline int rank(int x,int val)
{
  if(!x) return 0;
  if(t[x].val==val) return t[ls].size+1;
  if(t[x].val<val) return t[ls].size+t[x].cnt+rank(rs,val);
  if(t[x].val>val) return rank(ls,val);
}
inline int find(int x,int rk)
{
  if(!x) return 0;
  if(t[ls].size>=rk) return find(ls,rk);
  else if(t[ls].size+t[x].cnt<rk) return find(rs,rk-t[x].cnt-t[ls].size);
  else return t[x].val;
}
inline int pre(int x,int val)
{
  if(!x) return -inf;
  if(t[x].val>=val) return pre(ls,val);
  else return honoka(t[x].val,pre(rs,val));
}
inline int suc(int x,int val)
{
  if(!x) return inf;
  if(t[x].val<=val) return suc(rs,val);
```

```
  else return kotori(t[x].val,suc(ls,val));
}
int main()
{
  ios::sync_with_stdio(0);
  cin.tie(0);
  int n;
  cin>>n;
  while(n--)
  {
    int flag,x;
    cin>>flag>>x;
    if(flag==1) insert(root,x);
    else if(flag==2) del(root,x);
    else if(flag==3) cout<<rank(root,x)<<endl;
    else if(flag==4) cout<<find(root,x)<<endl;
    else if(flag==5) cout<< pre(root,x)<<endl;
    else if(flag==6) cout<< suc(root,x)<<endl;
  }
  return 0;
}
```

Dinic求最大流(无当前弧优化)

```
#include <iostream>
#include <cstdio>
#include <queue>
#include <cstring>
#define MAXM 100005
#define MAXN 10005
#define INF 2147483647
using namespace std;
inline int mymin(int a,int  b)
{
    return a < b ? a : b;
}
struct Edge{
    int to,v,cap,flow,ne;
};
class Graph{
    private:
        Edge e[MAXM*2];
        int head[MAXN];
        int dep[MAXN];

        queue <int> q;
        int ecnt;
    public:
        int s,t;
        void init()
        {
            memset(e,0,sizeof e);
            ecnt = -1;
            memset(head,-1,sizeof head);
            memset(dep,0,sizeof dep);
        }
```

```
        void addedge(int u,int v,int cap,int flow)
        {
            e[++ecnt].to = v;
            e[ecnt].cap = cap;
            e[ecnt].flow = flow;
            e[ecnt].ne = head[u];
            head[u] = ecnt;
            e[++ecnt].to = u;
            e[ecnt].cap = 0;
            e[ecnt].flow = -flow;
            e[ecnt].ne = head[v];
            head[v] = ecnt;
            return;
        }
        int bfs()
        {
            while(!q.empty())q.pop();
            memset(dep,0,sizeof dep);
            q.push(s);
            dep[s] = 1;
            while(!q.empty())
            {
                int nown = q.front();
                q.pop();
                for(int i = head[nown];i!=-1;i = e[i].ne)
                {
                    if(dep[e[i].to] != 0 || e[i].cap <= e[i].flow)continue;
                    dep[e[i].to] = dep[nown]+1;
                    q.push(e[i].to);
                }
            }
            return dep[t];
        }
        int dfs(int nown,int fl)
        {
            if(nown == t)return fl;
            for(int i = head[nown];i!=-1;i = e[i].ne)
            {
                if(dep[e[i].to] == dep[nown]+1 && e[i].cap > e[i].flow)
                {
                    int f = dfs(e[i].to,mymin(fl,e[i].cap - e[i].flow));
                    if(f > 0)
                    {
                        e[i] .flow += f;
                        e[i^1].flow -= f;
                        return f;
                    }
                }
            }
            return 0;
        }
        int Dinic()
        {
            int ans = 0;
            while(bfs())
            {
                while(int d = dfs(s,INF))ans+=d;
            }
```

```
            return ans;
        }
};
Graph g;
int main()
{
    g.init();
    int n,m,s,t;
    cin >> n >> m >> s >> t;
    g.s = s,g.t = t;
    for(int i = 0;i < m;i++)
    {
        int op,ed,w;
        cin >> op >> ed >> t;
        g.addedge(op,ed,t,0);
    }
    cout << g.Dinic();
}
```

增广路求最小费用最大流

```
#include <iostream>
#include <cstdio>
#include <queue>
#include <cstring>
#define MAXM 100005
#define MAXN 10005
#define INF 2147483647
using namespace std;
inline int mymin(int a,int  b)
{
    return a < b ? a : b;
}
struct Edge{
    int to,v,cap,flow,ne,cost;
};
class Graph{
    private:
        Edge e[MAXM*2];
        int head[MAXN];
        int dep[MAXN];
        int pred[MAXN];
        int pree[MAXN];
        int dis[MAXN];
        bool inq[MAXN];
        queue <int> q;
        int ecnt;
    public:
      int flow,cost;
        int s,t;
        void init()
        {
          flow = cost = 0;
            memset(e,0,sizeof e);
            ecnt = -1;
            memset(head,-1,sizeof head);
```

```
            memset(dep,0,sizeof dep);
        }
        void addedge(int u,int v,int cap,int flow,int cost)
        {
            e[++ecnt].to = v;
            e[ecnt].cap = cap;
            e[ecnt].flow = flow;
            e[ecnt].cost = cost;
            e[ecnt].ne = head[u];
            head[u] = ecnt;
            e[++ecnt].to = u;
            e[ecnt].cap = 0;
            e[ecnt].flow = -flow;
            e[ecnt].ne = head[v];
            e[ecnt].cost = - cost;
            head[v] = ecnt;
            return;
        }
        int spfa()
        {
            memset(inq,false,sizeof inq);
            while(!q.empty())q.pop();
            for(int i = 0;i < MAXN;i++)dis[i] = INF;
            dis[s] = 0;
            q.push(s);
            inq[s] = true;
            while(!q.empty())
            {
                int nown = q.front();
                inq[nown]=false;
                q.pop();
                for(int i = head[nown];i!=-1;i=e[i].ne)
                {
                    if(e[i].flow < e[i].cap && dis[e[i].to] > dis[nown] +
e[i].cost)
                    {
                        dis[e[i].to] = dis[nown] + e[i].cost;
                        pred[e[i].to] = nown;
                        pree[e[i].to] = i;
                        if(!inq[e[i].to])
                        {
                            q.push(e[i].to);
                            inq[e[i].to] = true;
                        }
                    }
                }
            }
            return dis[t];
        }
        int MCMF()
        {
            while(true)
            {
                int d;
                memset(pree,-1,sizeof pree);
                memset(pred,0,sizeof pred);
                d = spfa();
                if(d ==INF)break;
```

```
                int tf = INF;
                for(int i = t;pree[i]!=-1;i = pred[i])
                {
                    tf = mymin(tf,e[pree[i]].cap - e[pree[i]].flow);
                }
                for(int i = t;pree[i]!=-1;i = pred[i])
                {
                    e[pree[i]].flow += tf;
                    e[pree[i]^1].flow -= tf;
                }
                flow += tf;
                cost += d*tf;
            }
            return flow;
        }
};
Graph g;
int main()
{
    g.init();
    int n,m,s,t;
    cin >> n >> m >> s >> t;
    g.s = s,g.t = t;
    for(int i = 0;i < m;i++)
    {
        int op,ed,w,c;
        cin >> op >> ed >> w >> c;
        g.addedge(op,ed,w,0,c);
    }
    g.MCMF();
    cout << g.flow << " " << g.cost << endl;
}
```

堆

非常好的模板题

头文件和堆的定义

```
#include<iostream>
#include<cstdio>
using namespace std;
int minheap[1000010],n=0;//n为当前元素个数
```

插入函数

插入子结点直接按顺序插入,然后依次向上交换直到该节点比起父节点大或者改节点为根

```
inline void push(int w)
{
  minheap[++n]=w;
  for(int i=n,j=i>>1;j;i=j,j=i>>1)
  if(minheap[i]<minheap[j])swap(minheap[i],minheap[j]);
}
```

删除函数

删除子节点的时候将最后加入的元素调到根,然后依次向下调整,如果子节点比左右两个父节点都小,那就选择更小的父节点调整

```
inline void pop()
{
  minheap[1]=minheap[n--];
  for(int i=1,j=i<<1;j<=n;i=j,j=i<<1)
  {
    if(j<n && minheap[j+1]<minheap[j])j=j+1;
    if(minheap[i]<minheap[j])
    break;
    swap(minheap[i],minheap[j]);
  }
}
```

乘法逆元

请登录luogu后查看

用于求出1-n中所有整数在模p意义下的逆元

```
#include<iostream>
#include<cstdio>
#define lovelive long long
#define N 3000010
using namespace std;
int inv[N],n,p;
int main()
{
  scanf("%d%d",&n,&p);
  inv[1]=1;
  printf("1\n");
  for(int i=2;i<=n;i++)
  {
    inv[i]=(lovelive)(p-(p/i))*inv[p%i]%p;
    printf("%d\n",inv[i]);
  }
  return 0;
}
```

单调队列

dddl!

维护一个区间内的单调性和最值(存于队首)

求每个固定长度区间最值

顾名思义,单调队列即是一个单调的队列.

这里只考虑求区间最小值.

建立一个单调递增的队列,队首是最小值.

开始队列为空,每次添元素的时候,从队尾向前比较,

直到找到最靠前的比自己小的数,将待添加元素添加到这个数后面,

舍弃后面的所以数.(这是因为新添加的数一定比后面的数更优)

每次取最小值时,因为区间左端点在变,

故对于队列中的每一个元素,记录它在原数列中的位置.

当当前所求区间中不含队首元素时,将队首舍弃.

直到队首在所求区间时,输出队首即可.

由于每个元素最多进出队一次,故复杂度为O(n).

```
#include<iostream>
#include<cstdio>
#include<cstring>
#define maxn 100010
using namespace std;
struct dddl{
  int q[maxn],t[maxn],head,tail,ji;
  void csh(){head=tail=0;}
  void push(int e,int tim)//待添加的数是e,其在原数组中的位置是tim
  {
    if(ji==0)
    while(head<tail && q[tail-1]>=e)
    tail--;//单增

    if(ji==1)
    while(head<tail && q[tail-1]<=e)
    tail--;//单减

    tail++;
    queue[tail-1]=e;
    t[tail-1]=tim;
  }
  int gettop(int tim)//tim为当前区间所含的 下标最小的数 的下标
  {
    while(t[head]<tim)head++;
    return q[head];
  }
}queue;

```

二分图判定(DFS)

luogu1330(请登录后查看)

简单来说就是黑白染色

从任意一节点开始,进行dfs遍历,如果走过,判断颜色是否相同,如果不同说明无法染色,return 0

```
bool dfs(int u,int nowclr)
{
  if(vis[u])
  {return nowclr==clr[u];}
  vis[u]=1;
```

```
  clr[u]=nowclr;
  bool ok=1;
  for(int i=head[u];i;i=e[i].next)
  {
    if(!dfs(e[i].v,ans,1-nowclr))
    {ok=0;break;}
  }
  return ok;
 }
```

STL二分模板

STL大法好!

1.lower_bound

在递增序列里查找第一个大于等于被查找值的值,返回一个迭代器

```
#include<iostream>
#include<algorithm>
using namespace std;
int main()
{
  int a[10]={1,3,5,6,9,10,12,15,19,20};
  int* pos=lower_bound(a,a+10,7);
  cout<<(*pos)<<" "<<(pos-a)<<endl;//9 4
  return 0;
}
```

2.upper_bound

在递增序列里查找第一个大于被查找值的值,返回一个迭代器

```
#include<iostream>
#include<algorithm>
using namespace std;
int main()
{
  int a[10]={1,3,5,7,7,9,12,15,19,20};
  int* pos=upper_bound(a,a+10,7);
  cout<<(*pos)<<" "<<(pos-a)<<endl;//9 5
  return 0;
}
```

3.lower_bound(greater())

在递减序列里查找第一个小于或等于num的数字,返回迭代器

4.lower_bound(greater())

在递减序列里查找第一个小于或等于num的数字,返回迭代器

欧拉函数

很单纯的模板

```
#include<iostream>
#define maxn 10000010
using namespace std;
int phi[maxn],prime[maxn];
int cnt=0;
void euler()
{
    phi[1]=1;
    for(int i=2;i<10000010;i++)
    {
        if(!phi[i])
        {
            phi[i]=i-1;
            prime[cnt++]=i;
        }
        for(int j=0;j<cnt && i*prime[j]<maxn;j++)
        {
            if(i%prime[j])
            phi[i*prime[j]]=phi[i]*(prime[j]-1);
            else
            {
                phi[i*prime[j]]=phi[i]*prime[j];
                break;
            }
        }
    }
}
```

动规方程总结

背包类

01背包

```
for(int i=1;i<=n;i++)
{
  for(int j=m;j>=v[i];j--)
  f[j]=max(f[j],f[j-v[i]]+w[i]);
}
```

完全背包

```
for(int i=1;i<=n;i++)
{
  for(int j=v[i];j<=m;j++)
  f[j]=max(f[j],f[j-v[i]]+w[i]);
}
```

有依附的背包问题

确定每件物品的父子关系,先选父亲,如果父亲可以选就去判断儿子,然后诶个转移或者多个挑选

```
for(int i=1;i<=n;i++)
{
  for(int j=m;j>=v[i];j--)
  {
    f[j]=max(f[j],f[j-v[i]]+w[i]);
    if(can_choose_i())
    f[j]=max(f[j],f[j-v[i]-v_son_i]+w[i]+w_son_i);
  }
}
```

01背包筛符合条件的数(举例为倍数)

考试原题

在a[1]~a[n]里,筛去某个数组元素的倍数,每个元素只能用一次

用f[i]表示一个数是否被筛去

那么这个数被筛去的条件就是他自身被筛去或者他减去一个数之后可以被筛去

```
f[0]=1;
for(int i=1;i<=n;i++)
{
  if(f[a[i]])
  {
    ans--;
    continue;
  }
  for(int j=a[n];j>=a[i];j--)
  f[j]=f[j]|f[j-a[i]];
}
```

完全背包筛数(NOIP2018D1T2)

状态定义同上

与01背包和完全背包的关系一样,把循环改成正序即可

因为此处数可以选无数次

```
for(int i=1;i<=n;i++)
{
    for(int j=a[i];j<=/*上界*/;j++)
    f[j]=f[j] | f[j-a[i]];
}
```

线性结构

LIS类

求各种符合条件的子序列

用f[i]表示以i结尾的满足条件的最长子序列长度

那么$f[i]=max(f[i],f[j]+1),1<=j<=i$

可是n方的复杂度很是吃不消

nln实现方式

首先我们需要一个数组**a**，存储从第1个到第n个导弹的高度

然后一个数组**d**(其实是个单调栈)，存储不上升序列

把a中的**每个元素**挨个加到d里面：

(a中第i个元素为a[i]，d长度为len，d中最后一个(也是最小的一个)为d[len])

如果a[i] <= d[len]，说明a[i]可以接在d后面(而整个d还是有序的)，那就简单粗暴地把a[i]丢进d：

```
d[ ++len ] = a[i]
```

如果a[i] > d[len]，说明a[i]接不上

但是我们发扬**瞎搞精神**：**接的上要接，接不上创造条件也要接！**

在d中找到**第一个小于**a[i]的数，把它**删去**，用a[i]**代替**它！(为什么正确在下面)

假设这个数是y，怎样踹掉它呢?

很明显，我们需要使用lower_bound和upper_bound来查找

考虑使用什么

我们知道，要求的是最大**不上升**子序列长度，也就是如果两个元素**相等也是可以**的

所以我们删元素就**不用删等于a[i]的**了

结合上面，应该使用**upper_bound**(终于想起来它了)并且**使用>作为比较器**(这是个下降序列)

```
int p = upper_bound(d + 1, d + 1 + len, a[i], greater <int> () ) - d;
d[p] = a[i];
```

成功把a[i]塞了进去

下面是最长不上升

```
memset(dp,0,sizeof(dp));
  int ans=0;
  for(int i=1;i<=n;i++)
  {
    if(i==1)dp[++ans]=a[i];
    else
    {
      if(a[i]<=dp[ans]) dp[++ans]=a[i];
      else
      {
        int pos=upper_bound(dp+1,dp+ans+1,a[i],greater<int>())-dp;
        dp[pos]=a[i];
```

```
      }
    }
  }
```

Dilworth定理

把一个数列划分成最少的最长不升子序列的数目就等于这个数列的最长上升子序列的长度

链的最少划分等于最大反链长度

一般的线性dp

luogu1280

f[i]为第i~n分钟的最大休息时间

采用逆推法

$f[i]=f[i+1]+1$本时刻无任务

$f[i]=max(f[i],f[i+task[j]])$本时刻有任务

有任务的话就从前面1~当前时间,挑休息时间最长的更新现在的答案

tim[i]表示当前时间点有几个任务,遍历他们挑一个休息时间最长的

```
for(i=n;i>=1;i--)
{
  if(tim[i]==0)
  f[i]=f[i+1]+1;
  else for(j=1;j<=tim[i];j++)
  {
    if(f[i+z[num].js]>f[i])
    f[i]=f[i+z[num].js];
    num++;//当前已扫过的任务数
  }
}
```

luogu1880

$f[i][j]$表示i到j合并之后的最大的分,d可用前缀和优化

那么$f[i][j]=max(f[i][k]+f[k+1][j]+d[i][j])$

因为整个东西是环形的,所以我们开两倍空间断环成链即可

此处是记忆化搜索,每次搜索完的时候注意答案更新是[i,i+n+1],同样前面输入石子的时候也要开双倍并且stone[n+i]=stone[i];

```
void dfs1(int l,int r)
{
  if(f[l][r]!=-1)
  return;
  if(l==r)
  {f[l][r]=0;return;}
  else
  {
    int ans=2147483640;
    for(int k=l;k<r;k++)
```

```
    {
      dfs1(l,k);
      dfs1(k+1,r);
      ans=min(ans,f[l][k]+f[k+1][r]+(sum[r]-sum[l-1]));
    }
    f[l][r]=ans;
  }
}
void ans(void)
{
  for(int i=1;i<=n;i++)
  mincost=min(mincost,f[i][i+n-1]);
}
```

点分治

一.点分治是什么

点分治是一种树上分治的算法,用于处理各类树上信息(目前碰到的也就只是树上路径统计)

通过分治降低复杂度

二.点分治和序列分治的不同之处

实际上序列分治就是随机找点把序列分成两段,但是这个点找的好的情况下(中点),可以有效降低复杂度

点分治就是在树上寻找到这样的点,去把整棵树分开来求解问题

三.树的重心

树的重心就是这样一个平衡点,可以很有效的把复杂度优化出一个对数

树的重心:

树的重心也叫树的质心。找到一个点,其所有的子树中最大的子树节点数最少,那么这个点就是这棵树的重心,删去重心后，生成的多棵树尽可能平衡

其他充要条件:

每个子树大小都不大于$\frac{n}{2}$

到所有点距离之和最短

通常情况下一棵树只有一个重心,只有当结点数为偶数,并且有一条边连接大小均为$\frac{n}{2}$时,树有两个重心

求解树的重心可以根据定义来求

挨个求解子树大小,然后统计这个结点为重心时,各个子树大小中的最大值

然后$siz-size[u]$就是父节点那部分子树的大小

最后更新的重心

```
void dfs_G(int u,int fa,int siz)
{
  size[u]=1;
  max_part[u]=0;
  for(int i=head[u];i;i=e[i].ne)
  {
```

```
    if(e[i].to==fa) continue;
    dfs(e[i].to,u);
    size[u]+=size[e[i].to];
    max_part[u]=honoka(max_part[u],size[v]);
  }
  max_part[u]=honoka(max_part[u],siz-size[u]);
  if(max_part[u]<max_part[G]) G=u;
}
```

也可以根据第一个充要条件来求

这种写法需要两遍dfs,第一遍求出子树大小,第二遍从根开始,往子树大小大于$\frac{n}{2}$的子树里面去寻找并交换重心

直到最后找到一个点的子树大小均不大于$\frac{n}{2}$

唯一不好的就是两遍dfs

复杂度虽说不会被卡

但是对于本着瞎搞精神解题的本蒟蒻来说能省一点时间是一点时间

```
inline void dfs_sz(int u,int f)
{
  size[u]=1;
  for(int i=head[u];i;i=e[i].ne)
  {
    if(e.to==f) continue;
    dfs_sz(e[i].to,u);
    size[u]+=size[e[i].to];
  }
  return;
}
inline int dfs_find(int u,int f,int siz)
{
  for(int i=head[u];i;i=e[i].ne)
  {
    if(e.to==f) continue;
    if(size[e[i].to]*2>siz) return dfs_find(e.to,u);
  }
  return now;
}
inline int find_G(int s)
{
  dfs_sz(s,0);
  return dfs_find(s,0);
}

```

四.分治

我们把树上的路径分为两种

经过重心,然后分别分布在两颗子树里面的

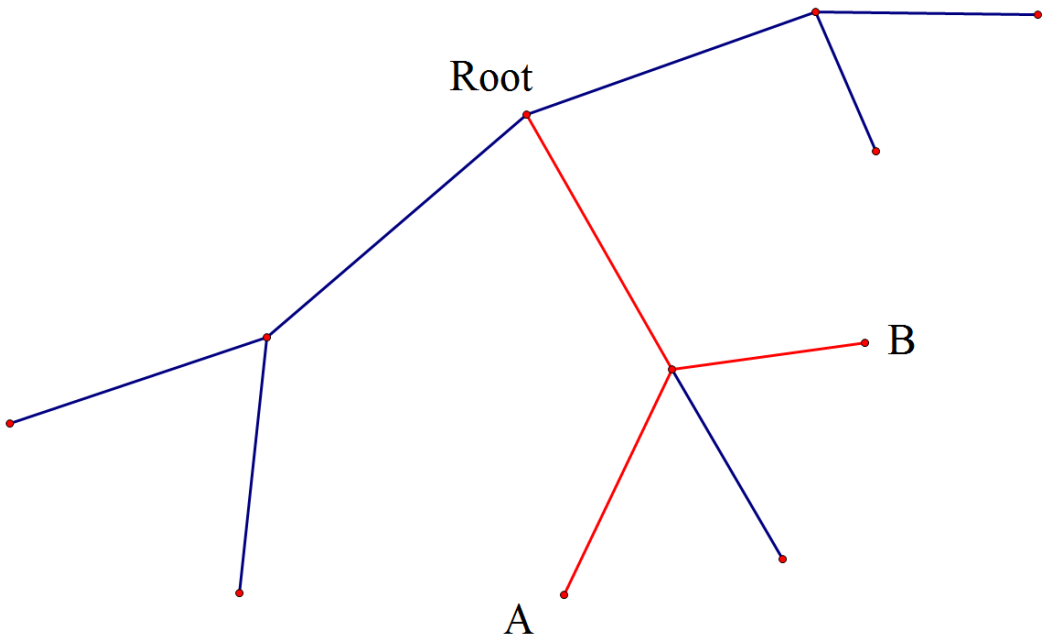
以及不经过重心单独分布在一颗子树里面的

第一种的话对当前重心的子树处理出一个$dis[maxn]$,记录子树内每个点到重心的距离然后配对就行了

第二种的话就把该点删掉,然后分别到它的几颗子树里面去寻找子树的重心,再次转化为一的情况去求解

```
inline void divide(int x)
{
  vis[x]=1;
  ans+=GET_ANS(x);//标记为走过并且统计该点答案
  for(int i=head[x];i;i=e[i].ne)//枚举每个子树
  {
    if(vis[e[i].to]) continue;
    siz=size[e[i].to],G=0
    dfs_G(e[i].to,x),divide(G);//寻找子树重心,并且分治子树
  }
  return;
}
```

但是这样会有重复,因为儿子结点可能是重心,重心到儿子,然后儿子到子树,中间重心到子树这一段会重复一次



处理方法是,先把答案加上重心到子结点的距离,然后每次统计出子树内的结点的距离的时候容斥一下给减掉即可

目前统计答案的模式是,我们对于树上路径长度为l的路径,用ans[l]记录长度为l的路径有几条,然后每发现一条给答案贡献1即可

get_ans函数实际上是modify_ans,由于不同题目要求不同,这个函数的内容也不一定一样,所以先不给出

五.好乱

突然感觉点分治的流程好乱

总结一下

我们高效处理树上问题

所以要使用点分治

使用点分治就得找到高效的分割点

树的重心就出来了

所以伪代码简单来说是这样

```
divide_tree()
{
  get_now_ans()

  for(every son_tree)
  {
    get_sontree_G()
    divide_sontree()
  }

  return;
}

int main()
{
  input()

  get_tree_G()

  divide_tree()

  ans_put()

  return 0;
}
```

流程:

1. 输入
2. 处理得到整棵树的重心
3. 开始分治
4. 对于每个重心暴力统计他的各个子树的答案
5. 分治每个重心的子树
6. 分治完毕输出答案

六.各个部分的函数以及参数定义

变量定义

```
int n,m,head[],e[],cnt//存图输入
int size[],maxp[]//统计重心
int dfn,id[],dis[],ans[]//统计答案
int vis[]//分治时标记点是否走过
```

存图输入没啥

统计重心也没啥

第三行dfn表示给一个重心的各个子树打上dfs序,方便枚举答案时统计

ans[k]表示长度为k的路径有几条

dis[i]表示i到重心(根)的距离,id[]则是的dfs序

vis[]保证不走重复点

主函数和输入

```
#include<iostream>
#include<cstring>
#include<cstdio>
#define maxn 10010
#define maxk 10000010
using namespace std;
struct edge{
  int to,w,ne;
}e[maxn<<1];
int n,m,head[maxn],cnt=0;
int size[maxn],maxp[maxn];
int dfn,id[maxn],dis[maxn],ans[maxk];
int vis[maxn];
int G=0;
inline void maki(int &s)
{
  s=0;char c=getchar();while(c<'0' || c>'9') c=getchar();
  while(c<='9' && c>='0')s*=10,s+=c-'0',c=getchar();return;
}
inline void nico(int u,int v,int w)
{
  e[++cnt].to=v;
  e[cnt].w=w;
  e[cnt].ne=head[u];
  head[u]=cnt;
  return;
}
inline int honoka(int a,int b)
{return a>b?a:b;}
int main()
{
  maki(n),maki(m);
  for(int i=1;i<n;i++)
  {
    int u,v,w;
    maki(u),maki(v),maki(w);
    nico(u,v,w);
    nico(v,u,w);
  }

  maxp[0]=n//千万别少这一句因为未初始化的重心是0

  dfs_G(1,0,n);

  divide(G);

  for(int i=1;i<=m;i++)
  {
    int k;
    maki(k);
    if(ans[k])
    printf("AYE\n");
    else
```

```
    printf("NAY\n");
  }
  return 0;
}

```

求重心

我们传给dfs一个目前结点,父节点,以及整颗子树的大小

vis是防止在分治子树的时候走回头路

用第一定义来求,全局变量G记录重心

```
inline void dfs_G(int u,int fa,int sum)
{
  size[u]=1;
  maxp[u]=0;
  for(int i=head[u];i;i=e[i].ne)
  {
    if(e[i].to==fa || vis[e[i].to])continue;
    dfs_G(e[i].to,u,sum);
    size[u]+=size[e[i].to];
    maxp[u]=honoka(maxp[u],size[e[i].to]);
  }
  maxp[u]=honoka(maxp[u],sum-size[u]);
  if(maxp[u]<maxp[G])
  G=u;
  return;
}
```

点分治

先标记该点,然后统计该点答案

遍历每个子树,并且去重

重复情况就是两个重心相连的情况,即整颗树的重心和下一个子树重心相连(如上图)

这种情况就得先在以原重心为根的地方先把答案加上

再到下一个结点上面减去这个答案

所以modify_ans函数里面先传个1,再传个-1

第一次路径长度为0,第二次加上两个重心之间的距离

```
inline void divide(int x)
{
  vis[x]=1;

  modify_ans(x,0,1);

  for(int i=head[u];i;i=e[i].ne)
  {
    if(vis[e[i].to])continue;
    modify_ans(e[i].to,e[i].w,-1);
    G=0;dfs_G(e[i].to,x,size[e[i].to]);
```

```
    divide(G);
  }
  return;
}
```

然后是统计答案

首先是初始化dfn标记

然后根节点距离就是我们传进来的len,w代表对答案的贡献值

先统计出各个结点到根的距离

再根据dfn序去枚举所有子结点统计答案

dis[]是否初始化无需考虑,因为每次只用到dfn而且是从头开始记的,并不影响其他的答案

```
inline void modify_ans(int x,int len,int w)
{
  dfn=0;
  dis[x]=len;
  get_dis(x,len,0);
  for(int i1=1;i1<=dfn;i1++)
  {
    for(int i2=1;i2<=dfn;i2++)
    {
      if(i1!=i2)
      ans[dis[i1]+dis[i2]]+=w;
    }
  }
  return;
}
inline void dfs_dis(int x,int d,int fa)
{
  dis[++dfn]=d;
  for(int i=head[x];i;i=e[i].ne)
  {
    if(vis[e[i].to] || e[i].to==fa) continue;
    dfs_dis(e[i].to,d+e[i].w,x);
  }
  return;
}
```

七.完整代码

```
#include<iostream>
#include<cstring>
#include<cstdio>
#define maxn 10010
#define maxk 10000010
using namespace std;
struct edge{
  int to,w,ne;
}e[maxn<<1];
int n,m,head[maxn],cnt=0;
int size[maxn],maxp[maxn];
int dfn,id[maxn],dis[maxn],ans[maxk];
```

```
int vis[maxn];
int G=0;
inline void maki(int &s)
{
  s=0;char c=getchar();while(c<'0' || c>'9') c=getchar();
  while(c<='9' && c>='0')s*=10,s+=c-'0',c=getchar();return;
}
inline void nico(int u,int v,int w)
{
  e[++cnt].to=v;
  e[cnt].w=w;
  e[cnt].ne=head[u];
  head[u]=cnt;
  return;
}
inline int honoka(int a,int b)
{return a>b?a:b;}
inline void dfs_G(int x,int fa,int sum)
{
  size[x]=1;
  maxp[x]=0;
  for(int i=head[x];i;i=e[i].ne)
  {
    if(e[i].to==fa || vis[e[i].to])continue;
    dfs_G(e[i].to,x,sum);
    size[x]+=size[e[i].to];
    maxp[x]=honoka(maxp[x],size[e[i].to]);
  }
  maxp[x]=honoka(maxp[x],sum-size[x]);
  if(maxp[x]<maxp[G])
  G=x;
  return;
}
inline void dfs_dis(int x,int d,int fa)
{
  dis[++dfn]=d;
  for(int i=head[x];i;i=e[i].ne)
  {
    if(vis[e[i].to] || e[i].to==fa) continue;
    dfs_dis(e[i].to,d+e[i].w,x);
  }
  return;
}
inline void modify_ans(int x,int len,int w)
{
  dfn=0;
  dis[x]=len;
  dfs_dis(x,len,0);
  for(int i1=1;i1<=dfn;i1++)
  {
    for(int i2=1;i2<=dfn;i2++)
    {
      if(i1!=i2)
      ans[dis[i1]+dis[i2]]+=w;
    }
  }
  return;
}
```

```cpp
inline void divide(int x)
{
  vis[x]=1;

  modify_ans(x,0,1);

  for(int i=head[x];i;i=e[i].ne)
  {
    if(vis[e[i].to])continue;
    modify_ans(e[i].to,e[i].w,-1);
    G=0;dfs_G(e[i].to,x,size[e[i].to]);
    divide(G);
  }
  return;
}
int main()
{
  maki(n),maki(m);
  for(int i=1;i<n;i++)
  {
    int u,v,w;
    maki(u),maki(v),maki(w);
    nico(u,v,w);
    nico(v,u,w);
  }
  maxp[0]=n;
  dfs_G(1,0,n);
  //cout<<G<<endl;
  divide(G);

  for(int i=1;i<=m;i++)
  {
    int k;
    maki(k);
    if(ans[k])
    printf("AYE\n");
    else
    printf("NAY\n");
  }
  return 0;
}

```

八.后记

首先主函数里面不要忘了maxp[0]=n

或者是一个很大的数也行

因为未初始化之前默认重心是0

第二这个算法复杂度并不是很能保证,因为统计答案时的枚举没法避免在同一棵子树里面的,容易超时而且要去重

至于染色标记子树内的点做到复杂度和答案不重复的方法,以后再学,现在这个已经够用了

第三

宽度十来天,总计想了5h+无数细碎的时间左右学会了点分治

剩下的背模板就好

二维线段树

二维线段树

扫描线

扫描线属于线段树的应用

一.实际需求

见luogu5490，最常见的情况就是求坐标平面内各个矩形的面积并

二.基本流程

(不找图了)

我们按照某一个顺序，比如从左往右，用一条垂直于y轴的直线依次扫过各个矩形

然后按照矩形的上下边去把这些相交的矩形分割成一个个不相交的小矩形，求出这些矩形的面积，自然就是答案

如果这样的话，也就是说我们要维护这条从下边移动到上边的线，也就是扫描线，对于这条线，我们要一直维护它交到了多少矩形面积，也就是这条直线上哪些区间有矩形

三.线段树和线段的定义

```
struct scanline{
  lovelive l,r,h;
  int mark;
}line[maxn<<1];

struct segtree{
  int l,r,sum;
  lovelive len;
}tree[maxn<<2];
```

这里的每条线段为一个四元组(对应于每条从下到上的矩形的边，包含左端点坐标，右端点坐标，这条边所在的高度，以及一个标记)

这个标记是这样的，每次碰到矩形的上边界，意思就是说我们要离开这个矩形了，就把这个flag设置成-1，下边界就是1，这样在线段树维护扫描线对应区间的时候我们要根据这个1把这段区间加进线段树，根据-1再减去它的贡献

然后线段树的定义就是这个结点维护的区间l,r,区间维护的线段长度，区间是否完全被覆盖

关于这个覆盖标记，比如有一条扫描线中间断了一部分，而我们要求的则是里面有线段覆盖的区间的长度(有效长度)，这个时候如果覆盖标记是1，那区间有效长度就是r-l，否则就是左右儿子加一起

四.建树和向上合并

正常情况下我的线段树是不写pushup只写pushdown的

但是扫描线里面不用往下传递，只需要直接修改，所以向上合并修改过后的信息就比往下传递要重要了

于是单写一个函数，pushup的规则如上面定义线段所述

```
inline void pushup(int o)
{
  int l=tree[o].l,r=tree[o].r;
  if(tree[o].sum) tree[o].len=X[r+1]-X[r];
  else{tree[o].len=tree[lson].len+tree[rson].len;}
  return;
}
inline void buildtree(int l,int r,int o)
{
  tree[o].l=l,tree[o].r=r;
  tree[o].len=0;
  tree[o].sum=0;
  if(l==r)
  return;
  buildtree(l,mid,lson);
  buildtree(mid+1,r,rson);
  return;
}
```

五.在写更新之前我们先看看如何处理区间

```
lovelive X[maxn<<1];
for(int i=1;i<=n;i++)
{
  int x1,x2,y1,y2;
  scanf("%d%d%d%d",&x1,&y1,&x2,&y2);
  X[(i<<1)-1]=x1,X[i<<1]=x2;
  line[(i<<1)-1]=(scanline){x1,x2,y1,1};
  line[i<<1]=(scanline){x1,x2,y2,-1};
}
n=n<<1;
sort(line+1,line+n+1,nozomi);
sort(X+1,X+n+1);
int tot=unique(X+1,X+n+1)-(X+1);
```

首先我们肯定不能把所有的xi全部制作成区间，这样肯定得MLE

随机我们想到了离散化，把每一个坐标按照相对大小存在X数组里面

然后排序，去重

然后最容易混乱的一步来了

线段树里面判断区间用的是X数组，我们用X数组的下标来寻找对应的线段的区间值

在刚开始存所有涉及到的xi的坐标的时候(从下往上扫)，我们把这些xi全部存入X数组以便查询，然后对于这个X数组左右动来判断线段树修改里面的区间交并

然后线段树维护的区间长度就用这些X[i]计算

这里的意义就和普通线段树的意义不一样了

总结一下，普通线段树直接维护坐标，并且在普通线段树里面这个区间可以直接开

但是在这题里面，或者其他坐标是实数而不是整数的题目里面就不能这么开

离散化之后线段树的根据就变成了离散化的数组，我们把只需要相对大小信息的原数组离散化为新的1~n数组，在这个数组上面去维护l,r

此时的l,r也和普通线段树的l,r不同，到此就很好理解了

六.区间修改

```
inline void update(lovelive ql, lovelive qr,int o,int add)
{
  int l=tree[o].l,r=tree[o].r;
  if(X[l]>=qr || X[r+1]<=ql) return;
  if(ql>=X[l] && qr<=X[r+1])
  {
    tree[o].sum+=c;
    pushup(o);
    return;
  }
  update(ql,qr,lson,add);
  update(ql,qr,rson,add);
  pushup(o);
  return;
}
```

七.关于y区间右偏一个的问题

略

八.完整代码(改成自己的码风)

```
#include<iostream>
#include<cstdio>
#include<algorithm>
#include<cstring>
#define MAKI main
#define lovelive long long
#define maxn 1000010
#define lson (o<<1)
#define rson (o<<1|1)
#define mid ((l+r)>>1)
using namespace std;
int n;
lovelive X[maxn<<1];
struct scanline{
  lovelive l,r,h;
  int flag;
}line[maxn<<1];
struct segtree{
  int sum;
  lovelive len;
}tree[maxn<<2];

bool nozomi(scanline l1,scanline l2)
{return l1.h<l2.h;}
inline void eli(int &s)
{
  s=0;char c=getchar();while(c<'0' || c>'9') c=getchar();
  while(c<='9' && c>='0')s*=10,s+=c-'0',c=getchar();return;
}
inline void pushup(int l,int r,int o)
```

```
{
  if(tree[o].sum) tree[o].len=X[r+1]-X[l];
  else{tree[o].len=tree[lson].len+tree[rson].len;}
  return;
}
inline void buildtree(int l,int r,int o)
{
  tree[o].len=0;
  tree[o].sum=0;
  if(l==r)
  return;
  buildtree(l,mid,lson);
  buildtree(mid+1,r,rson);
  return;
}
inline void update(lovelive ql,lovelive qr,int l,int r,int o,int add)
{
  if(X[l]>=qr || X[r+1]<=ql) return;
  if(X[l]>=ql && X[r+1]<=qr)
  {
    tree[o].sum+=add;
    pushup(l,r,o);
    return;
  }
  update(ql,qr,l,mid,lson,add);
  update(ql,qr,mid+1,r,rson,add);
  pushup(l,r,o);
  return;
}
int main()
{
  eli(n);
  for(int i=1;i<=n;i++)
  {
    int x1,x2,y1,y2;
    eli(x1),eli(y1),eli(x2),eli(y2);
    X[(i<<1)-1]=x1,X[i<<1]=x2;
    line[(i<<1)-1]=(scanline){x1,x2,y1,1};
    line[i<<1]=(scanline){x1,x2,y2,-1};
  }
  n<<=1;
  sort(line+1,line+n+1,nozomi);
  sort(X+1,X+n+1);
  int tot=unique(X+1,X+n+1)-X-1;
  tot--;
  buildtree(1,tot,1);
  lovelive ans=0;
  for(int i=1;i<n;i++)
  {
    update(line[i].l,line[i].r,1,tot,1,line[i].flag);
    ans+=tree[1].len*(line[i+1].h-line[i].h);
  }
  printf("%lld\n",ans);
  return 0;
}
```

Attention!

1. tot--,这里的tot才是线段树对应的区间
2. 数组开成1000010,否则会MLE+RE

解不定方程

裴蜀定理和拓展欧几里得

出处，二元一次不定方程的解法

$ax+by=c$，形如这样的方程叫做二元一次不定方程,现在我们要求解这个方程

定理1.原方程有解的充要条件是$gcd(a,b)|c$

定理2.若$gcd(a,b)=1$，求解出一个特解就能求出通解

通解表示为$\begin{cases}x=x_0-bt\\y=y_0+at\end{cases}$

拓展欧几里得

求$ax+by=gcd(a,b)=1$的整数解，其中a和b互质，并且这时候x就是a模b意义下的逆元

代码和过程推演

$$ax+by=gcd(a,b)=gcd(b,a\%b)$$
$$bx^{'}+(a\%b)y^{'}=gcd(b,a\%b)$$
$$\therefore ax+by=bx^{'}+(a\%b)y^{'}=bx^{'}+(a-a/b)y^{'}$$
$$\cdots$$

因为a,b互质，所以终止的时候是gcd(1,0),也就是说b=0，此时a是1，那么把x设置为1即可

那么$x^{'}$和$y^{'}$怎么办呢

$ax+by=bx^{'}+(a-a/b)y^{'}=ay^{'}+b(x'-a/b\ y')$

如果求出了上一层方程的解x',y',$\begin{cases}x=y'\\y=x'-a/b\ y'\end{cases}$便可完成递归

```
inline long long exgcd(long long a,long long  b,long long &x,long long &y)
{
  if(b==0)
  {
    x=1;
    y=0;
    return a;
  }
  long long d=exgcd(b,a%b,y,x);
  y=y-a/b*x;
  return d;
}
inline long long rev(long long a,long long n)
{
  long long x,y,d=exgcd(a,n,x,y);
  return x%n<=0?x%n+n:x%n;
}
```

裴蜀定理

若$gcd(a,b)=d$，那么$ax+by$一定都是d的倍数，换言之，$ax+by=d$一定存在整数解

接着刚刚的方程，然后我们考虑$gcd(a,b)=d$

方程$ax+by=1$，拓展欧几里得算法可解，这种情况是$gcd(a,b)=1$也就是ab互质的特殊情况

拓展欧几里得实际上求出的是方程$ax+by=gcd(a,b)$的一组特解

最后我们考虑$ax+by=c$，怎么变化这个方程得到同解方程，假设$gcd(a,b)|c, c/gcd(a,b)=d$

那我们先解$ax'+by'=gcd(a,b)$令解为x'，y'

也就是说，我们要求$ax+by=c$的解，我们先求解$ax'+by'=gcd(a,b)$,就能得到$ax+by=c$的一组特解$\begin{cases}x_0=x'\cdot\frac{c}{gcd(a,b)}\\y_0=y'\cdot\frac{c}{gcd(a,b)}\end{cases}$

然后利用方程的同解变换

$\begin{cases}ax+by=c\\ax_0+by_0=c\end{cases}$

做差可得$a(x-x_0)+b(y-y_0)=0$

同时除以$gcd(a,b)$得到

$\frac{a}{gcd(a,b)}(x-x_0)=-\frac{b}{gcd(a,b)}(y-y_0)$,由此可得$\frac{b}{gcd(a,b)}|(x-x_0)$,也就是$x=x_0+\frac{b}{gcd(a,b)}\times t$

那么通解的表达式就求出来了

$\begin{cases}x=x_0+\frac{b}{gcd(a,b)}\times t\\y=y_0+\frac{a}{gcd(a,b)}\times t\end{cases}$

总结

1.先考虑$ax+by=1, ax+by=gcd(a,b)$

2.接着对于$ax+by=c$,我们首先考虑$ax'+by'=gcd(a,b)$,并求出其特解x'_0, y'_0

3.同乘$\frac{c}{gcd(a,b)}$将x'_0, y'_0转化为$ax+by=c$的一组特解$\begin{cases}x_0=x'\cdot\frac{c}{gcd(a,b)}\\y_0=y'\cdot\frac{c}{gcd(a,b)}\end{cases}$

4.然后利用同解变换，找到通解公式$\begin{cases}x=x_0+\frac{b}{gcd(a,b)}\times t\\y=y_0+\frac{a}{gcd(a,b)}\times t\end{cases}$

```
int x00,y00;
exgcd()->x00,y00;
int x0,y0;
x0=x00*c/gcd(a,b),y0=y00*c/gcd(a,b);
int x,y;
x=x0+b/gcd(a,b)*t,y=y0+a/gcd(a,b)*t;
```

模板题洛谷1516青蛙的约会

```
#include<iostream>
```

```cpp
using namespace std;
inline long long int gcd(long long int a,long long int b)
{return b==0?a:gcd(b,a%b);}
inline long long int exgcd(long long int a,long long int b,long long int &x,long long int &y)
{
  if(b==0)
  {
    x=1,y=0;
    return a;
  }
  long long int tmp=exgcd(b,a%b,y,x);
  y=y-a/b*x;
  return tmp;
}
inline void umi(long long int &a,long long int &b)
{long long int t=a;a=b;b=t;return;}
int main()
{
  long long int x,y,m,n,l;
  cin>>x>>y>>m>>n>>l;
  if(m<n)
  {
    umi(x,y);
    umi(m,n);
  }
  long long int a=m-n;
  long long int g=gcd(a,l);
  if((y-x)%g!=0)
  {
    cout<<"Impossible"<<endl;
  }
  else
  {
    long long int t=(y-x)/g;
    long long int x1,y1;
    long long int i=exgcd(a,l,x1,y1);
    long long int ans=x1*t;
    l/=g;
    if(ans%l<=0)
    cout<<ans%l+l<<endl;
    else
    cout<<ans%l<<endl;
  }
  return 0;
}
```

解不定方程组

中国剩余定理

中国剩余定理用于求解形如

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

其中$m_1,m_2,\ldots,m_n$两两互质的方程组

解

我们构造$M=\prod_{i=1}^{n}m_i, M_i=\frac{M}{m_i}, M_it_i\equiv 1(mod\ m_i)$

那么就可以构造出一个解$x=\sum_{i=1}^{n}a_iM_it_i$

最小正整数解为$x\%M$

```
#include<iostream>
#include<cstring>
#include<cstdio>
using namespace std;
long long n,a[20],m[20],Mi[20],M=1,X;

inline void exgcd(long long a,long long b,long long &x,long long &y)
{
  if(b==0){x=1;y=0;return;}
  exgcd(b,a%b,y,x);
  y-=a/b*x;
}
int main()
{
  ios::sync_with_stdio(0);
  cin.tie(0);
  cin>>n;
  for(int i=1;i<=n;i++)
  {
    cin>>m[i]>>a[i];
    M*=m[i];
  }
  for(int i=1;i<=n;i++)
  {
    Mi[i]=M/m[i];
    long long x=0,y=0;
    exgcd(Mi[i],m[i],x,y);
    X+=a[i]*Mi[i]*(x<0?x+m[i]:x);
  }
  cout<<X%M<<endl;
  return 0;
}
```

解同余方程组(EXCRT)

中国剩余定理没法处理模数不互质的线性同余方程组,因为中国剩余定理里面解的构造原理就要求了模数两两互质

$$ans=\sum_{i=1}^{n}a_i\cdot M_i\cdot inv(M_i,m_i)(mod\ M)$$

如果M_i和m_i不互质的话,逆元就不存在,从而破坏整个解

那么我们怎么求解不互质的同余方程组呢

假设已经求出了前$k-1$个方程组成的同余方程组的一个解x

设$M=LCM_{i=1}^{k-1}m_i$

那么前$k-1$个方程组的通解为$x+t\cdot M(t\in Z)$

那么加入第k个方程之后

就是要求解一个同余方程满足$x+t\cdot M\equiv a_k(mod\ m_k)$

也就是要求一个t满足

$x+t\cdot M=m_k\cdot p+a_k$

化简一下就是$M\cdot t-m_k\cdot p=a_k-x$

对于这个方程,我们用扩展欧几里得和裴蜀定理就能求解

令$M=a,m_k=b,a_k-x=c,t=x,-p=y$,就是二元一次不定方程

$ax+by=c$

求解这个方程，更新答案即可

```
#include<iostream>
#include<cstring>
#include<cstdio>
#define maxn 100010
using namespace std;
long long n,bb[maxn],aa[maxn];
inline long long exgcd(long long a,long long b,long long &x,long long &y)
{
  if(b==0){x=1;y=0;return a;}
  long long gcd=exgcd(b,a%b,y,x);
  y-=a/b*x;
  return gcd;
}
inline long long multi(long long x,long long k,long long mod)
{
  long long res=0;
  while(k>0)
  {
    if(k&1) res=(res+x)%mod;
    x=(x+x)%mod;
    k>>=1;
  }
  return res;
}
long long excrt(void)
{
  long long x,y,k;
  long long M=aa[1],ans=bb[1];//第一个方程x和b[1]模a[1]同余，特解为b[1],M=a[1]
  for(int i=2;i<=n;i++)
  {
    long long a=M,b=aa[i],c=(bb[i]-ans%aa[i]+aa[i])%aa[i];//ax+by=c,Mt-
m_i*p=bi-x,ax \equiv c (mod b)
    long long gcd=exgcd(a,b,x,y);
    if(c%gcd!=0) return -1;

    //x=x*(c/gcd)%(b/gcd);
    x=multi(x,c/gcd,b/gcd);//乘法如果溢出，要手写
```

```
    ans+=x*M;
    M*=(b/gcd);
    ans=(ans%M+M)%M;
  }
  return (ans%M+M)%M;
}

int main()
{
  ios::sync_with_stdio(0);
  cin.tie(0);
  cin>>n;
  for(int i=1;i<=n;i++)
  {
    cin>>aa[i]>>bb[i];
  }
  cout<<excrt()<<endl;
  return 0;
}
```

AC自动机

节点定义

```
struct trie{
  int son[26],fail,flag;
}tree[maxn];
```

建立tie树

```
inline void insert(char *s)
{
  int u=1,l=strlen(s);
  for(int i=0;i<l;i++)
  {
    int v=s[i]-'a';
    if(!tree[u].son[v])
    tree[u].son[v]=++cnt;
    u=tree[u].son[v];
  }
  tree[u].flag++;
  return;
}
```

构建fail关系

```
inline void buildfail(void)
{
  for(int i=0;i<26;i++)
  tree[0].son[i]=1;
  q.push(1);tree[1].fail=0;
  while(!q.empty())
  {
    int u=q.front();q.pop();
```

```
    for(int i=0;i<26;i++)
    {
      if(tree[u].son[i]==0)
      {
        tree[u].son[i]=tree[tree[u].fail].son[i];
        continue;
      }
      tree[tree[u].son[i]].fail=tree[tree[u].fail].son[i];
      q.push(tree[u].son[i]);
    }
  }
  return;
}
```

匹配查询

```
inline int query(char *s)
{
  int u=1,l=strlen(s),ans=0;
  for(int i=0;i<l;i++)
  {
    int v=s[i]-'a';
    int k=tree[u].son[v];
    while(k>1 && tree[k].flag!=-1)
    {
      ans+=tree[k].flag;
      tree[k].flag=-1;
      k=tree[k].fail;
    }
    u=tree[u].son[v];
  }
  return ans;
}
```

前缀函数

```
inline int* kmp(char *s)
{
  int l=strlen(s);
  int pi[l+10];
  memset(pi,0,sizeof(pi));
  for(int i=1;i<l;i++)
  {
    int j=pi[i-1];
    while(j>0 && s[i]!=s[j]) j=pi[j-1];
    if(s[i]==s[j]) j++;
    pi[i]=j;
  }
  return pi;
}
```

前缀函数的应用

1.KMP

求字符串s在文本串t中出现的所有次数

设s'=s+#+t，对s'跑一遍前缀函数

其中如果$\pi[i]=len(s)$,则s在t的$i-(n-1)-(n+1)=i-2n$处出现

2.字符串的周期

周期的定义:对字符串s和$0<p\leq|s|$，若$s[i]=s[i+p]$对所有$i\in[0,|s|-1-p]$成立，则称p是s的周期。

border的定义:对字符串s和$0\leq r<|s|$,若s长度为r的前缀和长度为r的后缀相等，就称s长度为r的前缀是s的border。

根据前缀函数的定义,s的所有border长度为$\pi[n-1],\pi[\pi[n-1]-1],\cdots$

那么$\pi[n-1]$是s最长的border长度,所以s的最小周期为$n-\pi[n-1]$

Manacher

```
inline void manacher(char *s)
{
  memset(d1,0,sizeof(d1));
  memset(d2,0,sizeof(d2));
  int len=strlen(s);
  for(int i=0,l=0,r=-1;i<len;i++)
  {
    int k=(i>r)?1:kotori(d1[l+r-i],r-i+1);
    while(0<=i-k && i+k<len && s[i-k]==s[i+k])k++;
    d1[i]=k--;
    if(i+k>r)
    {
      l=i-k;
      r=i+k;
    }
  }
  for(int i=0,l=0,r=-1;i<len;i++)
  {
    int k=(i>r)?0;kotori(d2[l+r-i+1],r-i+1);
    while(0<=i-k-1 && i+k<len && s[i-k-1]==s[i+k])k++;
    d2[i]=k--;
    if(i+k>r)
    {
      l=i-k-1;
      r=i+k;
    }
  }
  int ans=0;
  for(int i=0;i<len;i++)
  {
    ans=honoka(ans,2*d1[i]-1);
    ans=honoka(ans,2*d2[i]);
  }
  cout<<ans<<endl;
```

```
}
```

字符串哈希

```
#define ll long long
#define pll pair<ll,ll>
namespace stringhash{
  static const int maxn=100;
  ll base,pow[2][maxn],mod[2];
  struct hash{
    ll hs[2][maxn];
    inline void init(char *s)
    {
      int n=strlen(s+1);
      hs[0][0]=hs[1][0]=0;
      for(int i=1;i<=n;i++)
      {
        hs[0][i]=(1ll*hs[0][i-1]*base%mod[0]+s[i])%mod[0];
        hs[1][i]=(1ll*hs[1][i-1]*base%mod[1]+s[i])%mod[1];
      }
      return;
    }
    inline pll gethash(int x,int y)
    {
      ll a=(hs[0][y]-hs[0][x-1]*pow[0][y-x+1]%mod[0]+mod[0])%mod[0];
      ll b=(hs[1][y]-hs[1][x-1]*pow[1][y-x+1]%mod[1]+mod[1])%mod[1];
      return make_pair(a,b);
    }
  };
  inline int isprime(ll x)
  {
    for(int i=2,swk=sqrt(x);i<=swk;i++)
    if(x%i==0) return 0;
    return 1;
  }
  inline void init(int n)
  {
    ll lim=5e7;
    srand(time(NULL));
    base=1ll*rand()%lim+lim;
    while(!isprime(base))++base;
    lim*=10;
    for(int i=0;i<=1;i++)
    {
      mod[i]=1ll*rand()%lim+lim;
      while(!isprime(mod[i])) ++mod[i];
      pow[i][0]=1;
      for(int j=1;j<=n;j++)
      pow[i][j]=pow[i][j-1]*base%mod[i];
    }
    return;
  }
}
```

康托展开

一.概念

康托展开是一个全排列到自然数的双射

也就是一一映射,康托展开的算法把每个全排列换算成一个独立的数,并且由这个数是可以推出原来的全排列,可以很方便地求n个数的第x个排列是什么

一句话，给出一个全排列，求它是第几个全排列，叫做康托展开。

另一句话，给出全排列长度和它是第几个全排列，求这个全排列，叫做逆康托展开

二.公式

$$X = a_n(n-1)! + a_{n-1}(n-2)! + \cdots + a_1 \cdot 0!$$

X是康托展开的数值

ai表示在给出的排列中,所有在第i位数后面,比i小的数的个数

例如: 3 4 1 5 2

A[1]=2,因为list[1]=3,从list[2]-list[5]中,有list[3]=1,list[5]=2比3小,所以A[1]=2

A[2]=2

A[3]=0

A[4]=1

A[5]=0

算出来X是61

那么怎么用61推出原排列3 4 1 5 2呢

首先$\frac{61}{(5-1)!} = 2 \cdot \cdot \cdot \cdot \cdot \cdot 13$

说明A[1]=2,那么在构成排列的五个数里面,首位后面有两个数比首位小,首位自然是3

继续$\frac{13}{(4-1)!} = 2 \cdot \cdot \cdot \cdot \cdot \cdot 1$

说明A[2]=2,3被挑过了就不考虑了,此时的第二位后面有两个数比它小,那第二位只能是4

继续$\frac{1}{(3-1)!} = 0 \cdot \cdot \cdot \cdot \cdot \cdot 1$

说明A[3]=0,那第三位只能是1

继续$\frac{1}{(2-1)!} = 1 \cdot \cdot \cdot \cdot \cdot \cdot 0$

说明A[4]=1,那第四位只能是5

然后第五位只能是还没被挑选过的2

这样就解出来了原排列3 4 5 1 2

三.代码实现

1.求康托展开值

```
int cantor(int *a,int n)
{
  int ans=0;
  for(int i=1;i<=n;i++)
  {
    int x=0;
    for(int j=i+1;j<=n;j++)
    if(a[j]<a[i])x++;
    ans+=x*jiecheng[len-i];
  }
  return ans+1;
}
```

2.逆康托展开

```
void decantor(int x, int n)
{
    vector<int> v;  // 存放当前可选数
    vector<int> a;  // 所求排列组合
    for(int i=1;i<=n;i++)
        v.push_back(i);
    for(int i=m;i>=1;i--)
    {
        int r = x % FAC[i-1];
        int t = x / FAC[i-1];
        x = r;
        sort(v.begin(),v.end());// 我从网上找的代码里面这里是要排序的，但是我不知道erase之后其他数据是不是保持原来的顺序，也就是说这个排序真的必要吗
        a.push_back(v[t]);      // 剩余数里第t+1个数为当前位
        v.erase(v.begin()+t);   // 移除选做当前位的数
    }
}
```

四.树状数组的优化

我们发现，第一个求解方法的复杂度是n平分的，这显然不是很友好，那么怎么优化呢

用树状数组维护逆序对!

首先先把n个数按照顺序加入树状数组

然后每次扫到一个数，那么比他小的，也就是说和他构成逆序对的数我们肯定还没扫到

直接求树状数组上sum(i)即可

好比树状数组上对于每个i我们都add(i,1)，这样sum(i)=i，去掉它本身就是比他小的数，如果在扫到它之前我们扫到比他小的数，直接在树状数组上另一个j位置add一个-1，就能消除这个数的影响，换言之它俩就没构成逆序对

有序数列1，2，3，4，...，n

add(2,1),add(3,1)

题目给定的目标数列:...，2，...，3，...，a[n]

我们提前扫到了2，那就add(2,-1)，就抵消了上面有序数列里面的add(2,1)，再扫到3的时候就不会产生影响

代码实现

```
int main()
{
  scanf("%d",&n);
  fac[0]=1;
  for(int i=1;i<=n;i++)
  {
    fac[i]=fac[i-1]*i%998244353;//预处理阶乘和树状数组
    add(i,1);
  }
  for(int i=1;i<=n;i++)
  {
    int x;
    scanf("%d",&x);
    ans+=(sum(x)-1)*fac[n-i];
    add(x,-1);
  }
  printf("%lld",ans+1);
  return 0;
}
```

主席树

主席树要使用动态开点

查询区间[l,r]对应的不仅是数列的[l,r],也还是时间轴的[l,r]

因为我们构建主席树的时候每个节点插入一次就把时间往后推一个点

用t[i]维护第i节点插入时的线段树的根节点,而第i个节点插入的时间点ti就对应了原序列的[1,ti]区间

t[r]-t[l-1]就是对应的[l,r]所插入的数的数量,也就是原数列区间[l,r]内的数,设这个差为x

现在询问第k小,那就去判断一下左子树内有多少数,如果比这个k大那就往左走,不然就往右边走

用root[i]维护一系列随时间(插入数)变化的线段树根节点,线段树维护值域区间[1,max(a[n])]

注意离散化

完整代码

```
#include<iostream>
#include<cstring>
#include<cstdio>
#include<algorithm>
#define maxn 200010
#define mid ((l+r)>>1)
using namespace std;
struct segtree{
  int lson,rson,sum;
}tree[maxn<<5];
int cnt=0;
int n,m,a[maxn],b[maxn];
int root[maxn];
```

```cpp
inline int buildtree(int l,int r,int o)
{
  o=++cnt;
  if(l==r) return o;
  tree[o].lson=buildtree(l,mid,tree[o].lson);
  tree[o].rson=buildtree(mid+1,r,tree[o].rson);
  return o;
}
inline int modify(int i,int l,int r,int o)
{
  int oo=++cnt;
  tree[oo].lson=tree[o].lson;
  tree[oo].rson=tree[o].rson;
  tree[oo].sum=tree[o].sum+1;
  if(l==r) return oo;
  if(i<=mid) tree[oo].lson=modify(i,l,mid,tree[oo].lson);
  else tree[oo].rson=modify(i,mid+1,r,tree[oo].rson);
  return oo;
}
inline int query(int ql,int qr,int l,int r,int k)
{
  int ans,x=tree[tree[qr].lson].sum-tree[tree[ql].lson].sum;
  if(l==r) return l;
  if(x>=k) ans=query(tree[ql].lson,tree[qr].lson,l,mid,k);
  else ans=query(tree[ql].rson,tree[qr].rson,mid+1,r,k-x);
  return ans;
}
int main()
{
  ios::sync_with_stdio(0);
  cin.tie(0);
  cin>>n>>m;
  for(int i=1;i<=n;i++)
  cin>>a[i],b[i]=a[i];
  sort(b+1,b+n+1);
  int nn=unique(b+1,b+n+1)-b-1;
  root[0]=buildtree(1,nn,root[0]);
  for(int i=1;i<=n;i++)
  {
    int ii=lower_bound(b+1,b+nn+1,a[i])-b;
    root[i]=modify(ii,1,nn,root[i-1]);
  }
  while(m--)
  {
    int l,r,k;
    cin>>l>>r>>k;
    cout<<b[query(root[l-1],root[r],1,nn,k)]<<endl;
  }
  return 0;
}
```

图的欧拉路，强连通分量，割点与割边，双连通分量，2SAT问题，**二分图匹配**，网络流，最大流最小割

增广路

我们用未覆盖点来表示不与任何匹配边邻接的点，其他点为匹配点，即恰好和一条匹配边邻接的点

从未覆盖点出发，依次经过非匹配边，匹配边...所得到的路称作交替路，如果交替路的终点是一个未覆盖点，那么这条路就是一条增广路，非匹配边比匹配边多一条

增广路可以改进匹配，假设我们已经找到一个匹配，如何判断该匹配是不是最大匹配

一个匹配是最大匹配的充要条件是不存在增广路，适用于任意图

根据增广路定理，最大匹配可以通过反复寻找增广路来求解，如何找到答案？根据定义首先找到一个未覆盖的点u作为起点，设这个u是X的结点。接下来需要选一个从u出发的非匹配边（u，v），达到Y结点v。如果v是未覆盖点。说明我们成功找到了一条增广路，如果v是匹配点，那我们下一不得走匹配边，因为一个匹配点恰好与一个匹配边邻接。设匹配点v邻接的匹配边的另一端点left[v],呢么可以理解从u直接走到了left[v],而这个left[v]也是一个X结点。如果始终没有找到覆盖点，最后会扩展出一颗匈牙利树。

欧拉回路

欧拉**路径**:指一个图里面存在一条路径,使得所有边恰好经过一次,这条路径称为图的欧拉路径

欧拉**回路**:如果这条路径的起点和终点相同,那么这条路称为欧拉回路

具有欧拉**回路**的图称为欧拉图,具有欧拉**路径**但是不具有回路的图称为半欧拉图

1.无向图存在欧拉回路的充要条件

一个无向图存在欧拉回路,当且仅当该图的所有顶点的度数都为偶数,且图联通

2.无向图存在欧拉路径的充分条件

这个图里面度数为奇数的点的个数为0或2,如果是2,那么必然一个是起点一个是终点

3.有向图存在欧拉回路的充要条件

一个有向图存在欧拉回路,所有顶点的入度等于出度且该图是连通图

4.有向图存在欧拉路径的充分条件

最多只有两个点的入度不等于出度,起点出度比入度大1,终点入度比出度大1

5.欧拉回路确实要求图联通,但是只要求所求的部分图联通,如果只对一个连通块求欧拉回路,那其他的连通块如何是无所谓的

2-SAT问题

给出一系列变量和方程,这些变量只能选择0或1两种值,求是否存在一种变量取值能满足给出的方程组

每一个方程都是对变量进行限制,**对于明确关系进行连边**

例如限制条件是 A&B=0

那么我们需要连边 $A_1->B_0$和$B_1->A_0$

虽然$B_0->A_1$满足关系式，但是当B=0的时候实际上推不出来A的明确取值，换言之$B_0->A_0$也能满足但是不连

对于一元关系的连边

```
add(a0,a1);//a=1
add(a1,a0);//a=0
```

如果这个变量一定要等于1那从a0走也只能走到a1

如果变量等于0,那就从a1走向a0

对于二元关系的连边

```
if(A & B ==1)//A=1,B=1
{
  add(A0,A1);
  add(B0,B1);
}
if(A & B ==0)
{
  add(A1,B0);
  add(B1,A0);
}
if(A | B ==0)//A=0,B=0
{
  add(A1,A0);
  add(B1,B0);
}
if(A | B ==1)
{
  add(A0,B1);
  add(B0,A1);
}
if(A ^ B ==0)
{
  add(A0,B0);
  add(B0,A0);
  add(A1,B1);
  add(B1,A1);
}
if(A ^ B ==1)
{
  add(A1,B0);
  add(A0,B1);
  add(B1,A0);
  add(B0,A1);
}
```

接下来我们只需要跑一遍tarjan缩点，如果A1和A0在一个强连通分量里面，那就不合法

至于取值，结论是如果x的拓扑序在!x之前那么x为真,由于tarjan是反向拓扑序,所以判断写成

```
if(scc[x]>scc[x+n]) x=1;
else x=0;
```

二分图匹配

1.二分图的判断

可以DFS黑白染色

2.二分图匹配

二分图中的匹配是一组边的选择方式,使得一个端点不会对应两条边

最大匹配是最大边数的匹配,在最大匹配中如果添加了任何边,则不再是匹配

```
inline bool dfs(int u)
{
  for(int i=head[u];i;i=e[i].ne)
  {
    if(!vis[e[i].to])
    {
      vis[e[i].to]=1;
      if(!match[e[i].to] || dfs(match[e[i].to]))
      {
        match[e[i].to]=u;
        return 1;
      }
    }
  }
  return 0;
}
int main()
{
  int n,m,e;
  maki(n),maki(m),maki(e);
  for(int i=1;i<=e;i++)
  {
    int x,y;
    maki(x),maki(y);
    nico(x,y);
  }
  int ans=0;
  for(int i=1;i<=n;i++)
  {
    memset(vis,0,sizeof(vis));
    if(dfs(i))
    ans++;
  }
  printf("%d\n",ans);
  return 0;
}
```

3.二分图最大权完美匹配

给每条边附一个权值，在做出匹配的同时最大化权值和

可行顶标:给每个节点分配一个权值$l(i)$，满足$\forall(u,v),w(u,v)\leq l(u)+l(v)$

相等子图:在一组可行顶标下，原图的生成子图，包含所有点但只包含$w(u,v)=l(u)+l(v)$的边

定理一：对于某组可行顶标，如果其相等子图存在完美匹配，那么，该匹配就是原二分图的最大权完美匹配

考虑原二分图任意一组完美匹配M，其边权和为

$val(M)=\sum_{(u,v)\in M}\leq\sum_{(u,v)\in M}l(u)+l(v)\leq\sum_{i=1}^{n}l(i)$

任意一组可行顶标的相等子图的完美匹配M'的边权和

$val(M')=\sum_{(u,v)\in M}l(u)+l(v)=\sum_{i=1}^{n}l(i)$

即任意一组完美匹配的边权和都不会大于$val(M')$，那个M'就是最大权匹配

有了这个定理，我们现在就要不断调整可行顶标，使得相等子图完美匹配

设节点数为n，$lx(i)$表示左边点的顶标，$ly(i)$表示右边点的顶标，$w(u,v)$表示左边第u个点和右边第v个点间的权重

那么初始化一组可行顶标

$lx(i)=max_{1\leq j\leq n}\{w(i,j)\},ly(i)=0$

然后选择一个为匹配点，求增广路

找不到增广路的时候我们就来调整顶标

调整到最后便可求出答案

```
#include<iostream>
#include<cstring>
#include<cstdio>
#include<queue>
#define maxn 510
#define maxm 250010
#define inf 214748364
using namespace std;
struct edge{
  int from,to,ne,w;
}e[maxm];
int n,m,tim=0;
long long lx[maxn],ly[maxn],slack[maxn];
int visx[maxn],visy[maxn];
int matchx[maxn],matchy[maxn];
int pre[maxn];
queue<int>q;
int cnt=0,head[maxn];
inline void nico(int u,int v,int w)
{
  e[++cnt].to=v;
  e[cnt].from=u;
  e[cnt].w=w;
  e[cnt].ne=head[u];
  head[u]=cnt;
  return;
}
inline long long kotori(long long a,long long b)
{return a<b?a:b;}
inline int honoka(int a,int b)
{return a>b?a:b;}
inline void modify(int u)
{
```

```
  for(int i=u,ne;i;i=ne)
  {
    ne=matchx[pre[i]];
    matchx[pre[i]]=i;
    matchy[i]=pre[i];
  }
  return;
}
inline void bfs(int u)
{
  for(int i=1;i<=n;i++)
  slack[i]=inf,pre[i]=0;
  while(!q.empty())
  q.pop();
  q.push(u);
  ++tim;
  while(1)
  {
    while(!q.empty())
    {
      int cur=q.front();
      q.pop();
      visx[cur]=tim;
      for(int i=head[cur];i;i=e[i].ne)
      {
        if(visy[e[i].to]==tim)continue;

        long long del=lx[cur]+ly[e[i].to]-e[i].w;

        if(del<slack[e[i].to])
        {
          slack[e[i].to]=del;
          pre[e[i].to]=cur;
          if(!del)
          {
            visy[e[i].to]=tim;
            if(!matchy[e[i].to])
            {
              modify(e[i].to);
              return;
            }
            q.push(matchy[e[i].to]);
          }
        }
      }
    }
    long long del=inf;
    for(int i=1;i<=n;i++)
    {
      if(visy[i]!=tim)
      del=kotori(del,slack[i]);
    }
    for(int i=1;i<=n;i++)
    {
      if(visx[i]==tim)
      lx[i]-=del;
      if(visy[i]==tim)
      ly[i]+=del;
```

```
      else
      slack[i]-=del;
    }
    for(int i=1;i<=n;i++)
    {
      if(visy[i]!=tim && !slack[i])
      {
        visy[i]=tim;
        if(!matchy[i])
        {
          modify(i);
          return;
        }
        q.push(matchy[i]);
      }
    }
  }
}
inline void KM()
{
  for(int i=1;i<=n;i++)
  bfs(i);
  long long ans=0;
  for(int i=1;i<=n;i++)
  ans+=lx[i]+ly[i];
  printf("%lld\n",ans);
}
int main()
{
  cin>>n>>m;
  for(int i=1;i<=n;i++)
  lx[i]=-inf,ly[i]=0;
  for(int i=1;i<=m;i++)
  {
    int u,v,w;
    cin>>u>>v>>w;
    nico(u,v,w);
    lx[u]=honoka(lx[u],w);
  }
  KM();
  return 0;
}
```

网络流

1.网络的概念

网络指一个有向图,每条边有一个权值c称为容量,具有源点和汇点

2.流量的定义

设$f(u,v)$定义在二元组$(u \in V, v \in V)$上的实数函数并且满足

1.容量限制:对于每条边,流经该边的流量不超过容量$f(u,v) \leq c(u,v)$

2.斜对称性:每条边的流量与其相反边的流量之和为0 ,$f(u,v) = -f(v,u)$

3.流守恒性:从源点流出的流量等于汇点流入的流量,即
$\forall x \in V-\{s,t\}, \sum_{(u,x)\in E} f(u,x)=\sum_{(x,v)\in E} f(x,v)$

那么称f为网络G的流函数,对于$(u,v)\in E, f(u,v)$称为边的流量,$c(u,v)-f(u,v)$称为边的剩余容量.整个网格的流量为$\sum_{(s,v)\in E} f(s,v)$,即从源点出发的所有流量之和

注:$f(u,v)=0,(u,v)\notin E,(v,u)\notin E$

3.常见问题

(1).最大流

有一张图,要求从源点流向汇点的最大流量

也就是说求出一个流函数

核心算法:寻找增广路

只要没有增广路，顺着残量网络到不了汇点，就存在一个无残量的割，其净流量等于容量

因为网络流的总流量同时等于图上任何一个割的净流量，也就同时不会超过图上任何一个割的容量

那么当总流量等于某割的容量时，没有比它更大的流，就求出了最大流

思路:从源点开始，顺着有流量的边，只要能找到一条到汇点的路，就会使得网络流量增加，在这个过程中建立反向边，方便以后回收流量

FF算法(暴力)

```
#include<iostream>
#include<cstring>
#include<cstdio>
#include<queue>
#define maxn 10010
#define maxm 200010
using namespace std;
struct edge{
  int to,ne;
  long long w;
}e[maxm];
int cnt=1,head[maxn];
int vis[maxn];
int n,m,s,t;
inline void nico(int u,int v,int w)
{
  e[++cnt].to=v;
  e[cnt].w=w;
  e[cnt].ne=head[u];
  head[u]=cnt;
  return;
}
inline long long kotori(long long a,long long b)
{return a<b?a:b;}
inline long long dfs(int u,long long flow)
{
  if(u==t) return flow;//走到汇点，返回流量
  vis[u]=1;
  for(int i=head[u];i;i=e[i].ne)
  {
```

```
    if(e[i].w==0 || vis[e[i].to]) continue;//没有残量了，直接继续
    long long res=dfs(e[i].to,kotori(flow,e[i].w));//顺着流量过去，要在当前流量和边的容量限制里面取个较小值
    if(res>0)
    {
      e[i].w-=res;//走过去了就残余容量减少,以后顺着反向边收回这些流量
      e[i^1].w+=res;
      return res;
    }
  }
  return 0;//不连通的情况下直接返回0
}
int main()
{
  cin>>n>>m>>s>>t;
  for(int i=1;i<=m;i++)
  {
    int u,v;
    long long w;
    cin>>u>>v>>w;
    nico(u,v,w);
    nico(v,u,0);
    //正向边直接连，反向边最开始容量为0
  }
  long long res=0,sum=0;
  while(1)
  {
    memset(vis,0,sizeof(vis));
    res=dfs(s,1e18);//源点无限流量
    if(res<=0) break;
    sum+=res;
  }
  printf("%lld\n",sum);
  return 0;
}
```

接下来我们考虑如何优化

每次多路增广：u 点通过一条边，向 v 输出流量以后，v 会尝试到达汇点（到达汇点才真正增广），然后 v 返回实际增广量。这时，**如果 u 还有没用完的供给，就继续尝试输出到其它边**

源点**顺着残量网络**想要到达其它点，需要经过一些边对吧？**按照经过的边数（即源点出发以后的距离）把图分层，即用 bfs 分层。** 每次尝试给予时，**只考虑给予自己下一层的点**，就可以防止混乱

综合上面两条。每回合也是从源点出发，**先按照当前残量网络分一次层**，随后多路增广，尽可能增加流量。增广过程中，会加入一些反向边，这些反向边逆着层次图，本回合并不会走。所以还需要进入下一回合。一直到 bfs 分层时搜不到汇点（即残量网络断了）为止。

这就是Dinic算法

Dinic

```
#include<iostream>
#include<cstring>
#include<cstdio>
#include<queue>
#define maxn 10010
#define maxm 200010
```

```cpp
using namespace std;
struct edge{
  int to,ne;
  long long lft;
}e[maxm];
int cnt=1,head[maxn];
int vis[maxn];
int depth[maxn];
int n,m,s,t;
queue<int>q;
inline void nico(int u,int v,int w)
{
  e[++cnt].to=v;
  e[cnt].lft=w;
  e[cnt].ne=head[u];
  head[u]=cnt;
  return;
}
inline long long kotori(long long a,long long b)
{return a<b?a:b;}
inline bool bfs()
{
  memset(depth,0,sizeof(depth));
  depth[s]=1;
  q.push(s);
  while(!q.empty())
  {
    int u=q.front();
    q.pop();
    for(int i=head[u];i;i=e[i].ne)
    {
      if(e[i].lft && !depth[e[i].to])
      {
        depth[e[i].to]=depth[u]+1;
        q.push(e[i].to);
      }
    }
  }
  return depth[t];
}
inline long long dfs(int u,long long in_flow)
{
  if(u==t) return in_flow;
  long long out_flow=0;
  for(int i=head[u];i;i=e[i].ne)
  {
    if(e[i].lft && depth[e[i].to]==depth[u]+1)
    {
      long long res=dfs(e[i].to,kotori(e[i].lft,in_flow));
      e[i].lft-=res;
      e[i^1].lft+=res;
      in_flow-=res;
      out_flow+=res;
    }
  }
  if(out_flow==0)
  depth[u]=0;
  return out_flow;
```

```
}
int main()
{
  cin>>n>>m>>s>>t;
  for(int i=1;i<=m;i++)
  {
    int u,v;
    long long w;
    cin>>u>>v>>w;
    nico(u,v,w);
    nico(v,u,0);
  }
  long long ans=0;
  while(bfs())
  ans+=dfs(s,1e18);
  printf("%lld\n",ans);
  return 0;
}
```

---

(2).最小费用最大流

每一条边都有一个费用,代表单位流量流过这条边的开销,要在求出最大流的同时,要求花费的费用最小

(3).最小割

删掉x条边使得源点汇点不互通,要求这x条边加起来的流量总和最小