

pragma GCC optimize(2) c++手动开启o2优化；

快读1：

```
inline int read()
{
    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;
}
```

然后main函数里读就这样写

```
d = read();
```

重载运算符

使用结构体但涉及比较或演算：要重载运算符；因为没有定义过node的运算。

要重载运算符，如下格式：

```
struct node{
    int x,y;
    node(int x=2,int y=2):x(x),y(y){
    }
    node operator + (const node b) const{
        node c;
        c.x = this->x + b.x;
        c.y = this->y + b.y;
        return c;
    }
    node operator - (const node b) const{
        node c;
        c.x = this->x - b.x;
        c.y = this->y - b.y;
        return c;
    }
    node operator * (const node b) const{
        node c;
        c.x = this->x * b.x;
        c.y = this->y * b.y;
        return c;
    }
    node operator / (const node b) const{
        node c;
        c.x = this->x / b.x;
        c.y = this->y / b.y;
        return c;
    }
};
```

比较运算符：注意以上内容加减比较法则均是自己定义，比如洛谷题解P1309可以先比较

```
if(t.s!=s) return s>t.s;//s是结构体中一元  
return idx<t.idx  
  
struct node{  
    int x;  
    bool operator <(const node b){  
        return this->x < b.x;  
    }  
    bool operator >(const node b){  
        return this->x > b.x;  
    }  
};  
int main(){  
    node a,b;  
    a.x=2,b.x=3;  
    printf("%d",b>a);  
}
```

merge

```
merge(v1.begin(), v1.end(), v2.begin(),v2.end(), v3.begin());
```

v:vector自己定义，可以是结构体，可以是栈，队……

勿忘重载运算符。

字典树trie

初始化根节点1，按多叉树形式与字典序方式排，各节点编号按出现顺序排列

```
int trie[size][26];// trie[i][j]指编号为i的（自然总数有size个）j子节点的编号
int tot = 1;//tot为目前节点数，初始化为一
//插入：
void insert(char*str)//插入str
{
    int len = strlen(str),p=1//p指下一步要操作的编号，初始化为1
    for(int k=0;k<=len-1;k++)
    {
        int ch = str[k]-'a';
        if(trie[p][ch]==0)//如果编号为p的节点无ch这一儿子
        {
            trie[p][ch]=++tot//tot节点数加一，新节点第++tot个出现编号为此
        }
        p=trie[p][ch];//以这个儿子的编号更新编号，继续操作
    }
    end[p] = true;//表示该节点结尾是个单词
}
//查询操作：
bool search(char*str)
{
    int len = strlen(str)-1;p=1//p=1只从根节点开始查
    for(int k =0;k<len;k++)
    {
        p=trie[p][str[k]-'a'];//更新节点
        if(p==0) return false//等于零说明没编号
    }
    return end[p] ; //到了最后一个，end[p]是true表示为一个单词
}
```

单调队列应用之 *slidingWindows*

description: An array of size $n \leq 106$ is given to you.

There is a sliding window of size k which is moving from the very

left of the array to the very right. You can only see the k numbers in the window.

Each time the sliding window moves rightwards by one position.

Your task is to determine the maximum and minimum values in the sliding window at each position.

```

#include<stdio.h>
#include<deque>
using std::deque;
const int N=1e6+3;
int a[N];
deque<int> q1,q2;
int main()
{
    int n,k;scanf("%d%d",&n,&k);
    for(int i=1;i<=n;i++)scanf("%d",&a[i]);
    for(int i=1;i<=n;i++)
    {
        //找最小，队头表示最小的index
        //当新的元素出现后是“更可能影响”更多windows的元素，先在末尾比较能不能更新
        while(!q1.empty()&&a[q1.back()]>a[i])q1.pop_back();
        q1.push_back(i);
        //由windows大小限制，从头更新
        if(!q1.empty()&&q1.front()<i-k+1)q1.pop_front();
        if(i>=k)printf("%d%c",a[q1.front()],i==n?'\\n':' ');
    }
    for(int i=1;i<=n;i++)
    {
        //找最大
        while(!q2.empty()&&a[q2.back()]<a[i])q2.pop_back();
        q2.push_back(i);
        if(!q2.empty()&&q2.front()<i-k+1)q2.pop_front();
        if(i>=k)printf("%d%c",a[q2.front()],i==n?'\\n':' ');
    }
}
//!维护一个单增队列，找寻最小值

```

二叉堆，优先队列

priority_queue<int, vector, greater >q;//这是一个小根堆q

注意某些编译器在定义一个小根堆的时候 `greater <int>` 和后面的`>`要隔一个空格，

不然会被编译器识别成位运算符号`>>`

小根堆显然也可以预处理每一个元素乘以`-1`然后建立大根堆来实现。

但优先队列不支持删除堆中任意元素

如果要删除非对顶元素则先标记，若要删除堆顶则删到没有标记为止。

优先队列类型为pair时，当pair的first值相同时，比较second。

Huffman tree

Huffman tree构造与Huffman 编码板子：

```

#include<cstdio>
#include<cstring>
#include<queue>
#include<algorithm>
#define ll long long
using namespace std;
struct node
{
    ll w,h;
    node(){w=0,h=0;}
    node(ll w,ll h):w(w),h(h){}
    bool operator <(const node &a) const{return a.w==w?h>a.h:w>a.w;}
};//排序原则：如若长度相等，高度小的优先；
ll ans;
priority_queue<node>q;
int main()
{
    ll n,k;ans=0;scanf("%lld%lld",&n,&k);
    for(int i=1;i<=n;i++)
    {
        ll w;scanf("%lld",&w);
        q.push(node(w,1));//一开始大家都是根节点
    }
    while((q.size()-1)%(k-1)!=0)q.push(node(0,1));//补零使得k-1|n-1
    while(q.size()>=k)
    {
        ll h=-1;ll w=0;
        for(int i=1;i<=k;++i)
        {
            node t=q.top();q.pop();
            h=max(h,t.h);
            w+=t.w;
        }
        ans+=w;
        q.push(node(w,h+1));//把节点权值之和加一起成一个新节点，height+1;
    }
    printf("%lld\n%lld\n",ans,q.top().h-1);
    return 0;
}

```

位运算，与状态压缩：

二进制状态压缩：将一个长度为m的bool数组用一个m位二进制整数表示储存：

用位运算查找：

第k位： $(n \gg k) \& 1$

取出n在二进制表示下第0到k-1位（后k位）： $n \& ((1 \ll k) - 1)$

第k位取反： $n \wedge (1 \ll k)$

对整数n二进制下第k位赋值为1： $n \mid (1 \ll k)$

对整数二进制下第k位赋值为0： $n \& (\sim(1 \ll k))$

`lowbit(n)`：定义为非负整数n在二进制表示下“最低位1及其后面所有的0”构成的数值

`lowbit(n)`实现：先将n取反，此时第k位变为0，第0-k-1位都是1。令 $n=n+1$ ，进位，第k位变为1，0-k-1位为0

所以 $\text{lowbit}(n) = n \& (\sim n + 1) = n \& (-n)$

用途：配和hash找出整数二进制下所有是一的位， $O(1)$ 数量级

只需不断 $n = \text{lowbit}(n)$ 直至 $n=0$ ，第几位自然可通过`lowbit(n)`得知；

111用法

```
#include <stdio.h>
#include <stdlib.h>

int main()
{
    int a = 1LL * 100000 * 100000 / 1000;
    printf("%d", a);
    system("pause");
    return 0;
}
//否则会溢出;
```

unique(,);

unique(a, a+t) : 对数组第零项到第t项去重

```
#include <iostream>
#include <algorithm>
int main(void){
    int a[8] = {2, 2, 2, 4, 4, 6, 7, 8};
    int c;
    std::sort(a, a + 8); //对于无序的数组需要先排序
    c = (std::unique(a, a + 8) - a); //c表示去重后有多少项;
    std::cout<< "c = " << c << std::endl;
    //打印去重后的数组成员
    for (int i = 0; i < c; i++)
        std::cout<< "a = [" << i << "] = " << a[i] << std::endl;
    return 0;
}
```

迭代器

```
vector<int> vect;
vector<int>::iterator it = vect.begin();
```

首先定义了一个int类型的向量；然后定义了一个具有int元素的迭代器类型。it的类型就是
cpp vector<int>::iterator

迭代器 iterator 是C++ STL的组件之一，作用是用来遍历容器，而且是通用的遍历容器元素的方式，无论容器是基于什么数据结构实现的，尽管不同的数据结构，遍历元素的方式不一样，但是用迭代器遍历不同容器的代码是完全一样的。

经典的迭代器遍历容器的代码如下：

```
vector::iterator it = vec.begin();
for (; it != vec.end(); ++it)
{
    cout << *it << " ";
```

```
}
```

```
cout << endl;
```

c++ map

可以将任何基本类型映射到任何基本类型。如 `int array[100]` 事实上就是定义了一个 `int` 型到 `int` 型的映射。

`map` 提供一对一的数据处理，`key-value` 键值对，其类型可以自己定义，第一个称为关键字，第二个为关键字的值

`map` 内部是自动排序的

```
#include<map>头文件
map<type1name,type2name> maps;//第一个是键的类型，第二个是值的类型

map<string,int> maps;
```

`map` 可以使用 `it->first` 来访问键，使用 `it->second` 访问值

```
#include<map>
#include<iostream>
using namespace std;
int main()
{
    map<char,int>maps;
    maps['d']=10;
    maps['e']=20;
    maps['a']=30;
    maps['b']=40;
    maps['c']=50;
    maps['r']=60;
    for(map<char,int>::iterator it=mp.begin();it!=mp.end();it++)
    {
        cout<<it->first<<" "<<it->second<<endl;
    }
    return 0;
}
maps.insert() 插入
// 定义一个map对象
map<int, string> m;

//用insert函数插入pair
m.insert(pair<int, string>(111, "kk"));

// 用insert函数插入value_type数据
m.insert(map<int, string>::value_type(222, "pp"));

// 用数组方式插入
m[123] = "dd";
m[456] = "ff";
maps.begin()返回指向map头部的迭代器
maps.end()返回指向map末尾的迭代器
maps.rbegin()返回指向map尾部的逆向迭代器
maps.rend()返回指向map头部的逆向迭代器
//反向迭代
map<string,int>::reverse_iterator it;
for(it = maps.rbegin(); it != maps.rend(); it++)
    cout<<it->first<<' '<<it->second<<endl;
maps.empty()判断其是否为空
maps.swap()交换两个map
maps:映射即可建立hash表
```

lower_bound

`lower_bound(begin, end, num)`: 从数组的begin位置到end-1位置二分查找第一个大于或等于num的数字，

找到返回该数字的地址，不存在则返回end。通过返回的地址减去起始地址begin, 得到找到数字在数组中的下标。

`upper_bound(begin, end, num)`: 从数组的begin位置到end-1位置二分查找第一个大于num的数字，找到返回该数字的地址，不存在则返回end。通过返回的地址减去起始地址begin, 得到找到数字在数组中的下标。

时间复杂度:二分的复杂度；

`lower_bound(begin, end, num, greater())`: 从数组的begin位置到end-1位置二分查找第一个小于或等于num的数字， 找到返回该数字的地址，不存在则返回end。通过返回的地址减去起始地址begin, 得到找到数字在数组中的下标。

`upper_bound(begin, end, num, greater())`: 从数组的begin位置到end-1位置二分查找第一个小于num的数字，

找到返回该数字的地址，不存在则返回end。通过返回的地址减去起始地址begin, 得到找到数字在数组中的下标。

memset 赋值最大最小值的各种情况集合

int

"较"的原则：加法不爆。

极大值: 0x7f

较大值: 0x3f

较小值: 0xc0

极小值: 0x80

long long

"较"的原则：加法不爆。

极大值：0x7f

较大值：0x3f

较小值：0xc0

极小值：0x80

float

"较"的原则：保证一定位精度。

7f以上一直到be都是-0（实际上是一个很小的>-1.0的负数）

极大值：0x7f

较大值：0x4f

较小值：0xce

极小值：0xfe

0xff是 -1.#QNAN0000…… (-∞?)

double

"较"的原则：保证一定位精度。

极大值：0x7f

较大值：0x43

较小值：0xc2

极小值：0xfe

线性dp拾遗：

!!t0提醒：dp成功的精髓是由最优子结构可以推到最终最优，若最终最优实际上不由最优子结构推出，则不可；

(详见凸优化理论，线性规划属于典型的凸优化问题，具有局部最优即为全局最优的特点)

1. 数组由二维降维到一维：

0/1背包：注意第i次状态只与第i减一次有关：

那么应该可以让前一个分量只需开0或1就可，每个阶段会不断更新此且有照顾到了相邻两个阶段在阶段数大时节省空间。

```
for(int i=1;i<=n;i++)
{
    for(int j=0;j<=m;j++)
        dp[i&1][j]=dp[(i-1)&1][j];
    for(int j=v[i];j<=m;j++)
        f[i&1][j]=max(f[i&1][j],f[(i-1)&1][j-v[i]]+w[i]);
}
int ans =0;
for(int j=0;j<=m;j++)
{
    ans = max(ans,f[n&1][j]);
}
```

!但实际上还进行了一次拷贝：所以应该还可以优化：

dp[j]:放入j的最大价值和：

```

int dp[];
memset(dp, 0xc0, sizeof(dp)); // -INF;
dp[0] = 0;
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]); // !!为什么是倒叙呢，因为是求最大值，倒叙前面大的部分才是i+1状态可能
// !!另外正序可能会同一个i内多次状态更新，状态相当于用了多次
int ans = 0;
for(int j=0; j<=m; j++)
{
    ans = max(ans, f[n+1][j]);
}

```

!但同时意味这正序正好是完全背包，因为完全背包本来就要在同一个i下更新

完全背包：

dp[j]: 放入j的最大价值和：

```

int dp[];
memset(dp, 0xc0, sizeof(dp)); // -INF;
dp[0] = 0;
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]);
int ans = 0;
for(int j=0; j<=m; j++)
{
    ans = max(ans, f[n+1][j]);
}

```

多重背包及其改进：

二进制拆分：背景每个物品，每个最多选C[i]次：

若看作c[i]个物品效率太低：

令 $c[i] >= 2^{0+……+2^p}$, $c[i] < 2^{0+……+2^{(p+1)}}$;

把 $c[i]*v[i]$ 看作

$2^0 v[i], 2^{1*v[i]}, \dots, 2^{p_v[i]}, r[i]*v[i],$

$r[i] = c[i] - 2^{(p+1)+1};$

example: luogu1833

分组背包：n组，每组 $c[i]$ 个物品， i 组 j 个 $v[i][j]$, 价值 $w[i][j]$ ；

```
memset(f, 0xcf, sizeof(f));
f[0]=0;
for(int i=1; i<=n; i++)
{
    for(int j=m; j>=0; j++)
    {
        for(int k=1; k<=c[i]; k++)
        {
            if(k>=v[i][k])
                f[j]=max(f[j], f[j-v[i][k]]+w[i][k]);
        }
    }
}
```

首先因为是0/1背包，所以 j 是倒序枚举，此外对 $c[i]$ 个物品的枚举在 j 内层

!lyd之提醒：分清阶段（此处第*i*个），状态（此处*j*重量），决策（第*k*个物品选不选），三者枚举由外到内；

example: luogu1064

区间dp: 区间长度作为阶段，区间左右两端描述维度，

example lydp294

dp环形结构减少枚举处理：

!任选一个位置断开，复制形成2倍长度的链。

树形dp: DP ON THE TREE

example : lydp300;

采用递归的形式，按结点y分类，`dp(y) return y下的最值`，

递推：自上而下。

个人以为就是记忆化搜索，即把之前`dfs return`的内容用`dp`数组储存即可了；

然后有个二次扫描合根法，`remember`；

状态压缩：

集合的状态区与数对应，可以采用进制，

数位dp：

特点：给一些限制条件，求第k小数或区间内有多少满足限制条件的数

dp初始化问题

初始化的细节问题

我们看到的求最优解的背包问题题目中，事实上有两种不太相同的问法。有的题目要求“恰好装满背包”时的最优解，有的题目则并没有要求必须把背包装满。

一种区别这两种问法的实现方法是在初始化的时候有所不同。

如果是第一种问法，要求恰好装满背包，那么在初始化时除了 $f[0]$ 为0其它 $f[1..V]$ 均设为 $-\infty$ ，这样就可以保证最终得到的 $f[N]$ 是一种恰好装满背包的最优解。

如果没有要求必须把背包装满，而是只希望价格尽量大，初始化时应该将 $f[0..V]$ 全部设为0。

为什么呢？可以这样理解：初始化的 f 数组事实上就是在没有任何物品可以放入背包时的合法状态。

如果要求背包恰好装满，那么此时只有容量为0的背包可能被价值为0的nothing“恰好装满”，其它容量的背包均没有合法的解，属于未定义的状态，它们的值就都应该是 $-\infty$ 了。

如果背包并非必须被装满，那么任何容量的背包都有一个合法解“什么都不装”，这个解的价值为0，所以初始时状态的值也就全部为0了。

这个小技巧完全可以推广到其它类型的背包问题，后面也就不再对进行状态转移之前的初始化进行讲解。

用dp找到背包问题第k优解：

```
for (int i=1;i<=n;i++) //阶段仍是按每个分类
    for (int j=v;j>=w[i];j--)
    {
        //0/1背包
        int t1=1,t2=1,t[60],len=0;
        //!第k解更新思想：f[m][i]表示v<=m第i好解，其值能由f[m-w[i]][s](s待定)+c[i]与前i-1的f[m]
        //!那么分别枚举i,s以完成状态更新
        while (t1+t2<=k+1)
        {//!保证len只看前k个（大一点也没毛病？）
            if (f[j][t1]>f[j-w[i]][t2]+c[i]) //!判断第i好的解需不需要更新
                t[++len]=f[j][t1++];//!不需要则保持原状向上更新
            else t[++len]=f[j-w[i]][t2++]+c[i];//!状态开始更新，t2是要更新时才递增
        }
        for (int K=1;K<=k;K++) f[j][K]=t[K];
    }
```

并查集：

储存结构：树，每个集合选择一个固定的元素作为整个集合的代表，树根作为代表

维护fa[x]：保存x的父节点。

操作：

- get: get to know x belongs to which group
- merge: merge group a and group b;

初始化：fa[x]=x；

get的路径压缩：每次查询让访问过的节点都直接指向树根

```

int get(int x)
{
    if(fa[x]==x) return x;
    return fa[x]=get(fa[x]);//向上递归并且让经过的点都与根直接相连
}
// merge:
void merge(int x,int y){
    fa[get(x)] = get(y);
}
//若边带权值，可以在路径压缩的同时将其更新为到根的权值之和:
int get(int x)
{
    if(x==fa[x]) return x;
    int root = get(fa[x]);
    d[x]+=d[fa[x]];//维护d数组---对边权求和
    return fa[x]=root;//路径压缩
}
//merge中维护边权: 即边权加上被并入的集合的大小,额外维护一个size数组
void merge(int x,int y)
{
    x=get(x),y=get(y);
    fa[x]=y;d[x]=size[y];//更新父节点与边权
    size[y]+=size[x];//更新size[]
}

}

```

!但如果只关心最大边最小的情形: 最小生成树

!按边权从小到大来建树(Kruskal算法)

并查集特别适用的地方: 可传递的关系中动态维护并查集。

树状数组

定义: c[x]用于保存序列a的区间 $[x-\text{lowbit}(x)+1, x]$ 中所有数的和;

- 支持操作1: 求前缀和和

```

int ask(int x)
{
    int ans=0;
    for(; x; x-=x&-x)//(即x-=lowbit(x))
    {
        ans+=c[x];
    }
    return x;
}

```

- 支持操作2：单点增加

序列中某个 $a[x]+y$ 只有 $c[x]$ 及其所有祖先保存的区间和含 $a[x]$ ，其祖先最多 $\log n$ 个

```

void add(int x,int y)
{
    for( ; x<=N;x += x&-x)c[x]+=y;
}
//!节点x的祖先: x+lowbit(x);

```

树状数组的初始化：

c 初始全空，然后对每个 x (x 从小到大， $a[x]$ 在每个要加的地方加一遍) 执行 $add(x, a[x])$, $N \log N$ 量级

对于二维也是一样，每一维实际独立，也是记录 $x-lowbit(x)+1$, y 与 $y-lowbit(y)+1$, 双重循环一下就ok

线段树

定义：

- 1每个节点代表一个区间
- 2有唯一根节点，代表整个统计范围，如 $[1, N]$
- 3每个叶节点代表长为1的元区间 $[x, x]$ ；
- 4每个内部节点 $[l, r]$ ，左节点是 $[l, mid]$ ，右节点是 $[mid, r]$, $mid=(l+r)/2$;
- 5. 编号规则：父子两倍：根节点1， x 左子节点为 $2x$, 右子节点 $2x+1$ ，最后一行编号可以不连续，保存线段树数组长度要不小于 $4N$.

建树

以区间dp中区间最大值为例：

计为dat(l, r), dat(l, r)=max(dat(l, mid), dat(mid+1, r));

```
struct SegmentTree
{
    int l,r;
    int dat;
}t[size*4];

void build (int p,int l,int r)
{
    t[p].l=l,t[p].r=r;
    if(l==r)
    {
        t[p].dat = a[l];
        return;//叶节点
    }
    int mid=(l+r)/2;
    build(p*2,l,mid);
    build(p*2,mid+1,r);
    t[p].dat = max(t[2*p].dat,t[2*p+1].dat)//先递归得到t[2*p],t[2*p+1],再得到t[p];
}
build(1,1,n);//调用入口
```

修改（单点直接修改）

单点修改：自下至上修改x为v，从[x, x]开始

```

void change(int p, int x, int v)
{
    if(t[p].l == t[p].r)
    {
        t[p].dat=v;
        return ;
    }
    int mid = (t[p].l+t[p].r)/2;
    if(x<=mid)change(p*2,x,v); //x在左边
    else change(p*2+1,x,v); //x在右边，从下到上改，从上到下找
    t[p].dat=max(t[2*p].dat, t[2*p+1].dat);
}
change(1,x,v); //调用入口

```

查询

区间查询：递归过程：

- 1. 若 $[l, r]$ 完全覆盖当前节点代表的区间，则立即回溯，并且该节点的dat值为候选答案。
- 2. 若左子节点与 $[l, r]$ 有重叠部分，则递归访问左子节点
- 3. 若右子节点与 $[l, r]$ 有重叠部分，则递归访问右子节点

```

int ask(int p,int l,int r)
{
    if(l<=t[p].l&&r>=t[p].r) return t[p].dat;
    //若完全包含，就直接作为候选答案之一，在后面val=max(val,ask());中更新所需答案
    int mid = (t[p].l+t[p].r)/2;
    int val = -(1<<30); //负无穷大
    if(l<=mid)val=max(val,ask(p*2,l,r)); //左有重叠，就找下去
    if(r>mid)val = max(val,ask(p*2+1,l,r)); //右有重叠，就找下去
    return val;
    //图示详见lyd213;
    //以上操作自己模拟一下时间复杂度就是logN
}
cout<<ask(1,1,r)<<endl;
//区间最值自此除了用st表，还可以用线段树。

```

!!ps:

此处以询问区间和为例。实际上线段树可以处理很多符合结合律的操作

根据需求可在线段树里动态储存许多信息，关键是抓住怎么由 $2p$ 与 $2p+1$ 状态递推到编号 p ；

区间修改：

lazy tag:

比如说修改 $[l, r]$, $[pr, lr]$ 属于 $[l, r]$ 但 $[pr, lr]$ 的父节点也包含于 $[l, r]$,

则无须修改 $[pr, lr]$, 只需修改更大的（才能作为备选）就行。

也就是说除了在修改指令中直接划分成的 $O(\log N)$ 个节点外，任意节点修改延迟到“后续操作中递归到进入父节点时”执行

时间复杂度： $O(N)$

对区间要改的区间 $[l, r]$ 中的 x , lazy tag所在位置 $[l1, r1]$: 树上有标号的最大的含有 x 且属于 $[l, r]$ 的区间

example: 把数列中第 $l-r$ 个数都加 d

```

struct segmenttree
{
    int l,r;
    long long sum,add;
#define l(x) tree[x].l
#define r(x) tree[x].r
#define sum(x) tree[x].sum
#define add(x) tree[x].add
    //lazy tag:表示该节点曾经被修改但其子节点尚未被更新
}tree[4*n];
int a[n],n,m;

//-----建树-----
void build(int p,int l ,int r)
{
    l(p)=l,r(p)=r;
    if(l==r)
    {
        sum(p)=a[l];
        return ;
    }
    int mid=(l+r)/2;
    build(p*2,l,mid);
    build(p*2+1,mid+1,r);
    sum(p) = sum(p*2)+sum(p*2+1);
}

//----下实现延迟标记向下传递-----
void spread(int p)
{
    if(add(p))//!节点p有标记
    {
        sum(p*2)+=add(p)*(r(p*2)-l(p*2)+1);
        /**
         * !更新左子节点信息,若父节点有标记,则没必要递归到子节点,让子节点全部加d就OK, 然后让标记下延
         */
        sum(p*2+1)+=add(p)*(r(p*2+1)-l(p*2+1)+1); // 更新右子节点信息
        add(p*2)+=add(p); //给左节点打标记
        add(p*2+1)+=add(p); //给右节点打标记
        add(p) = 0; //清除p标记(此时其子节点也完成修改),保证标记只在第一个包含地方打
    }
}

```

```

//-----进行区间修改-----//
void change(int p, int l, int r, int d)
{
    if(l<=l(p)&&r>=r(p))//见lazy tag笔记中对应的完全覆盖情况
    {
        sum(p)+=(long long)d * (r(p)-l(p)+1);
        add(p)+=d;
        return;
    }
    spread(p) //下传延时标记
    int mid = (l(p)+r(p))/2;
    if(l<=mid)change (p*2,l,r,d);
    if(r>mid)change(p*2+1,l,r,d);
    sum(p) = sum(p*2)+sum(p*2+1);
}

```

```

//-----进行区间查询-----//
long long ask(int p,int l,int r)
{
    if(l<=l(p)&&r>=r(p))return sum(p);
    spread(p); //下传延时标记
    //!ask里也必须要有spread不然无法对节点信息更新:
    //!!注意lazy tag修改的只是根节点的信息
    int mid = (l(p)+r(p))/2;
    long long val = 0;
    if(l<=mid) val+=ask(p*2,l,r);
    if(r>mid)val += ask(p*2+1,l,r);
    return val;
}

```

__int128的使用：

用途：long long不够但不想升高精度，范围是 -2^{127} 到 $2^{127}-1$ ，也就是10的39次方左右

__int128在gcc、codeblocks、vs2017都是不被支持的，不过__int128在Linux上可以编译并且能用。

我们提交到大部分OJ上都是可以编译且能用的。

输入输出

C/C++标准IO是不认识__int128这种数据类型的，cin和cout是无法输出__int128的，所以我们要自己实现输入输出，

其他的运算，与int没有什么不同。

可以对long long直接强转成__int128

__int128能直接做加减乘除赋值

输入输出按以下格式：

```

#include <bits/stdc++.h>
using namespace std;
inline __int128 read()
{
    __int128 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;
}
inline void print(__int128 x)
{
    if(x < 0){
        putchar('-');
        x = -x;
    }
    if(x > 9)
        print(x / 10);
    putchar(x % 10 + '0');
}
int main(void)
{
    __int128 a = read();
    __int128 b = read();
    print(a + b);
    cout << endl;
    return 0;
}

```

快速幂板子

求 $a^b \% c$;

```

long long mc(long long a, long long b, long long c)
{
    long long jc=1;
    while(b)
    {
        if(b&1) jc=(jc*a)%c;
        a=(a*a)%c;//a一直更新着a^(2^k)
        b>>=1;
    }
    return jc;
}

```

模p的逆元板子：用于分数取模，组合数等等

1. 利用费马小定理&欧拉定理+快速幂：

```

long long qkpow(long long a,long long p,long long mod)
{
    long long t=1,tt=a%mod;
    while(p)
    {
        if(p&1)t=t*tt%mod;
        tt=tt*tt%mod;
        p>>=1;
    }
    return t;
}//快速幂
long long getInv(long long a,long long mod)
{
    return qkpow(a,mod-2,mod);
}//简单欧拉定理

```

2. 实质即 $ax+by\equiv 1 \pmod{p}$ 的不定方程求解（取 $b=p$ 即可）

解不定方程用的方法叫exgcd(扩展欧几里德算法)

```

void Exgcd(ll a, ll b, ll &x, ll &y) {
    if (!b) x = 1, y = 0;
    else Exgcd(b, a % b, y, x), y -= a / b * x;
}

```

i mol p的逆：先定义好x,y,调用exgcd(i,p,x,y),输出x%p就可

3. 线性算法：在算一连串的逆比上两个快：

设 $p = k * i + r$ ；

$k * i + r \equiv 0 \pmod{p}$

$\text{multi } i^{-1}, r^{-1}$

$k * r^{-1+i} - 1 \equiv 0$ ；

$i^{-1} \equiv -k * r^{-1}$ ；

$i^{-1} \equiv -[p/i] * (p \bmod i)^{-1}$

由此知当然是可从 $i=1$ 一直往后递推得到一连串的

代码：

```
inv[1] = 1;
for(int i = 2; i < p; ++ i)
    inv[i] = (p - p / i) * inv[p % i] % p;
```

可持续化线段树

过程详解：见知乎收藏文章

代码（luoguP3919）：

```

#include <bits/stdc++.h>
using namespace std;
int read()
{
    int ans = 0, sgn = 1;
    char c = getchar();
    while (!isdigit(c))
    {
        if (c == '-')
            sgn *= -1;
        c = getchar();
    }
    while (isdigit(c))
    {
        ans = ans * 10 + c - '0';
        c = getchar();
    }
    return ans * sgn;
}
const int MAXV = 20000000, MAXN = 1000005;
#define ls(x) tree[x].ls
#define rs(x) tree[x].rs
#define val(x) tree[x].val
#define mark(x) tree[x].mark
struct node
{
    int val, ls, rs;
} tree[5*MAXV];//动态开点会好一点
int A[MAXN], roots[MAXN], n, m, cnt = 1; // roots记录每个历史版本的根节点
void build(int l = 1, int r = n, int p = 1)
{
    if (l == r)
        val(p) = A[l];
    else
    {
        ls(p) = ++cnt, rs(p) = ++cnt;
        int mid = (l + r) / 2;
        build(l, mid, ls(p));
        build(mid + 1, r, rs(p));
        val(p) = val(ls(p)) + val(rs(p));
    }
}
//!update更新新的节点同时保留过去版本：也即新建一个节点来储存新的内容。其余完全同线段树

```

```

void update(int x, int d, int p, int q, int cl = 1, int cr = n) // 单点修改
{
    if (cl == cr)
        val(q) = d;
    else
    {
        ls(q) = ls(p), rs(q) = rs(p);
        int mid = (cl + cr) / 2;
        if (x <= mid)
            ls(q) = ++cnt, update(x, d, ls(p), ls(q), cl, mid);
        else
            rs(q) = ++cnt, update(x, d, rs(p), rs(q), mid + 1, cr);
        val(q) = val(ls(q)) + val(rs(q));
    }
}

int query(int l, int r, int p, int cl = 1, int cr = n) // 区间查询
{
    if (cl > r || cr < l)
        return 0;
    else if (cl >= l && cr <= r)
        return val(p);
    else
    {
        int mid = (cl + cr) / 2;
        return query(l, r, ls(p), cl, mid) + query(l, r, rs(p), mid + 1, cr);
    }
}

int main()
{
    n = read(), m = read();
    for (int i = 1; i <= n; ++i)
        A[i] = read();
    build();
    roots[0] = 1;
    for (int t = 1; t <= m; ++t)
    {
        int v = read(), o = read();
        if (o == 1)
        {
            int x = read(), d = read();
            roots[t] = ++cnt; // 新建节点
            update(x, d, roots[v], roots[t]);
        }
    }
}

```

```

else
{
    int x = read();
    roots[t] = roots[v]; // 复用v号版本
    printf("%d\n", query(x, x, roots[v]));
}
}

return 0;
}

```

对于可持续化线段树的区间修改

若是普通线段树，会用lazy tag去修改sum数组的值，利用spread操作

!这里可持续化线段树中只用lazy tag记录当前区间应该怎么改

!!关键是lazy tag不下传

搜索时向下传lazy tag

```

void pushup(int id,int l,int r)
{
    //当前区间的值等于左右子节点的值的和加上当前区间的lazy标记
    sum[id]=sum[ln[id]]+sum[rn[id]]+(r-l+1)*lazy[id];
}

```

区间更新

l和r为当前区间左右端点，L和R为目标区间左右端点

```

void update_interval(int pre,int id,int L,int R,int val,int l,int r)
{
    ln[id]=ln[pre];rn[id]=rn[pre],sum[id]=sum[pre],lazy[id]=lazy[pre];
    if(l>=L&&r<=R)
    {
        lazy[id]+=val;
        sum[id]+=(r-l+1)*val;
        return ;
    }
    int mid=l+r>>1;
    if(L<=mid) ln[id]=++idx,update_interval(ln[pre],ln[id],L,R,val,l,mid);
    if(mid+1<=R) rn[id]=++idx,update_interval(rn[pre],rn[id],L,R,val,mid+1,r);
    pushup(id,l,r);
}

```

区间查询

l 和 r 为当前区间左右端点， L 和 R 为目标区间左右端点

询问时需要下传 lz ，因为没有spread操作，参数中要有 lz

```

ll query_interval(int id,int L,int R,ll lz,int l,int r)
{
    if(l>=L&&r<=R) return lz*(r-l+1)+sum[id];
    int mid=l+r>>1;
    ll ans=0;
    if(L<=mid) ans+=query_interval(ln[id],L,R,lz+lazy[id],l,mid);
    if(mid+1<=R) ans+=query_interval(rn[id],L,R,lz+lazy[id],mid+1,r);
    return ans;
}

```

- 应用一：利用可持久化线段树求静态区间第 k 小数的方法

luoguP3834

离散化：只关心偏序关系不关心实际大小

- 一. 按大小偏序关系不变：即先创立副本，排序，去重，查找每个元素在副本中的位置，排名作为离散化后的值

```

for (int i = 1; i <= n; ++i) // step 1
    tmp[i] = arr[i];
std::sort(tmp + 1, tmp + n + 1); // step 2
int len = std::unique(tmp + 1, tmp + n + 1) - (tmp + 1); // step 3
for (int i = 1; i <= n; ++i) // step 4
    arr[i] = std::lower_bound(tmp + 1, tmp + len + 1, arr[i]) - tmp;
//(查找即二分, lower_bound)

```

复杂度: $n \log n$

- 二. 根据输入顺序离散化为不同数据对数据建立结构体标明出现顺序与值将副本从小到大排序, 相同值按顺序从小到大排序
离散化后数组放回原数组

```

struct Data {
    int idx, val;

    bool operator<(const Data& o) const {
        if (val == o.val)
            return idx < o.idx; // 当值相同时, 先出现的元素离散化后的值更小
        return val < o.val;
    }
} tmp[maxn]; // 也可以使用 std::pair

for (int i = 1; i <= n; ++i) tmp[i] = (Data){i, arr[i]};
std::sort(tmp + 1, tmp + n + 1);
for (int i = 1; i <= n; ++i) arr[tmp[i].idx] = i;

```

莫队算法

莫队是处理区间问题的乱搞神器，

尤其是对于离线查询问题，因为会对查询做一些预处理

当然也可以做在线查询，比如带修莫队。

对查询先进行预处理，即排序

因为如果是在一个区间已经处理了的情况下，另一个区间只需这个区间首尾平移很近的距离

那么这个计算量就完全可以接受

挪动操作：

```
//ans1是上一次查询的左端点，ans2是上一次查询的右端点  
//lef是当前查询的左端点，rig是当前查询的右端点  
while(ans1>lef) ans1--,add(a[ans1]); //add是扩展函数  
while(ans2<rig) ans2++,add(a[ans2]);  
while(ans1<lef) del(a[ans1]),ans1++; //del是删除函数  
while(ans2>rig) del(a[ans2]),ans2--;
```

其中add和del表示挪动端点x过程中要改变什么，

排序操作：先分块成 \sqrt{n} 块，然后先将这些区域按左端点l所在的块从左到右排序

再把l所在相同区间按r从小到大排序

(说人话就是从内含（不一定完全包含，所以顺序可以规定先左后右）的小区间往外扩)

```
int block=sqrt(n);  
inline bool cmp(node x,node y){  
    if((x.l-1)/block==(y.l-1)/block) return x.r<y.r;  
    return x.l/block<y.l/block;  
}
```

vector

vector：实际啊可以理解为一个长度可变的数组、支持随机访问，删改一般在末尾

头文件：include

```
vector <int> a //相当于一个长度动态变化的数组
```

```
vector <int> b[233] //相当与第一维长233，第二维长度变化的int数组
```

迭代器：相当于stl容器的指针

set & multiset

分别代表有序集合与有序多重集

内部实现：实质是红黑树，与优先队列一样，储存元素要先定义小于号

`insert()` 插入操作：时间复杂度为 $\log n$

`set` 的 `erase` 操作：

`set` 的 `erase` 操作是基于元素值的，而不是基于元素在数组中的位置。因此，`s.erase(nums[i-k])` 可能会删除多个相同的元素，而不是仅仅删除滑动窗口中最早进入的那个元素。

`set` 的 `rbegin()` 方法：

`set` 的 `rbegin()` 方法返回的是 `set` 中最大的元素，但这并不一定是当前滑动窗口中的最大值。因为 `set` 中可能包含已经不在滑动窗口中的元素。

示例

考虑以下输入：

`nums = [1, 3, -1, -3, 5, 3, 6, 7], k = 3`

`CopyInsert`

在某些情况下，`set` 中可能会包含已经不在滑动窗口中的元素，导致 `rbegin()` 返回的值不正确。

map

建立一个key-value的映射，实质是一个以key为关键码的红黑树

key必须定义小于号：`map<key_type , value_type> name;`

`example`

`map <string ,int>hash` : 实质是一个哈希表

操作： $\log(n)$

`h[key]`: 就是 `value`, 若找不到 `key`, 则会新建一个 `(key, 0)`, 即 `h[key]=0`, 所以找之前最好先 `find`

这个特性反而可以用来建立表统计字符串出现个数

```

map <string,int> h;
char str[25];
for(int i=1;i<=n;i++){
    scanf("%s",str);h[str]++;
}
//下为m次询问:
for(int i=1;i<=m;i++){
{
    scanf("%s",str);
    if(h.find(str)==h.end())puts("0");
    else
}
}

```

bitset:

多位二进值数，相当于状态压缩的而二进制数组

位运算ok, s[k] 表示第几位, s[0] 最低为,

s. count() 返回多少个一

s. set(), 所有为变成1, reset() 变成0

s. set(k, v) 把s的第k位改成v, 即s[k]=v;

图的储存

邻接表

数据被分为多个链表，每个链表代表一个类，每个类有一个代表元素对应表头head，表头构成表头数组
建立邻接表：head与next数组保存的是ver（顶点）数组的下标，相当于指针，0表示指向空

```

void add(int x,int y,int z)//加入有向边(x,y),权值为z
{
    ver[++tot]=y;edge[tot]=z;
    next[tot]=head[x];head[x]=tot;//在表头x处插入

}

// 访问从x出发的所有条边:

for(int i=head[x];i;i=next[i])
<br>{
    int y = ver[i],z=edge[i];
    //找到了一条有向边(x,y),权值为z

}

```

空间复杂度 $O(n+m)$

无向图用两条有向边插入就好,适用于大量遍历邻近节点的算法

邻接矩阵

$a[i][j]=0: i==j, w[i][j], (i, j) \in E, INF, (i, j) \notin E$
 空间复杂度 $O(n*n)$

链式前向星

把边集数组中的每一条边按照起点从小到大排序,如果起点相同就按照终点从小到大

并且记录下以某个点为起点的所有边在数组中的起始位置和存储长度

```

const int maxn = 1005;//点数最大值
int n, m, cnt;//n个点, m条边
struct Edge
{
    int to, w, next;//终点, 边权, 同起点的上一条边的编号
}edge[maxn];//边集
int head[maxn];//head[i], 表示以i为起点的第一条边在边集数组的位置（编号）
void init()//初始化
{
    for (int i = 0; i <= n; i++) head[i] = -1;
    cnt = 0;
}

void add_edge(int u, int v, int w)//加边, u起点, v终点, w边权
{
    edge[cnt].to = v; //终点
    edge[cnt].w = w; //权值
    edge[cnt].next = head[u];//以u为起点上一条边的编号, 也就是与这个边起点相同的上一条边的编号
    head[u] = cnt++;//更新以u为起点上一条边的编号
}
for (int i = 1; i <= m; i++)//输入m条边
{
    cin >> u >> v >> w;
    add_edge(u, v, w);//加边
    //加双向边
    //add_edge(u, v, w);
    //add_edge(v, u, w);
}

```

单源最短路径：

DIJKSTRA算法

给定一张有向图G (V, E) , n个点m个边，节点【1, n】连续编号，(x, y, z) 表示一条从x开始到达y长度为z的有向边

题目：设一号店为起点，求长度为n的数组dist，其中dist【i】表示从节点1到节点i的最短路径长度

算法过程：贪心，见代码：

```
int a[3030][3030], d[3030], n, m;  
  
bool v[3030];//标记数组;  
  
void dijkstra()  
{  
  
    memset(d, 0x3f, sizeof(d));//先全部初始化为正无穷  
    memset(v, 0, sizeof(v));//节点标记初始化  
    d[1]=0;//标记起点  
    for(int i=1;i<n;i++){  
        int x=0;  
        //找到未标记节点中dist最小的  
        for(int j=1;j<=n;j++)  
            if(!v[j]&&(x==0||d[j]<d[x]))x=j;  
        v[x]=1;//找到未标记节点中dist最小的并标记，有贪心知这个最小肯定更新不了了，那就是最优  
        //下用全局最小值点x更新其他节点  
        for(int y=1;y<=n;y++)  
        {  
            d[y]=min(d[y], d[x]+a[x][y]);  
        }  
    }  
  
    // 适用于边权为正，为负贪心错误
```

但以上找最小明显可以用优先队列维护：

```
priority_queue <pair <int , int> > q;

void dijkstra() {

    memset(d,0x3f,sizeof(d)); //先全部初始化为正无穷
    memset(v,0,sizeof(v)); //节点标记初始化
    d[1]=0; //标记起点
    q.push(make_pair(0,1)); //第二维为编号
    while(q.size())
    {
        int x=q.top().second; q.pop();
        if(v[x]) continue;
        v[x]=1; //标记过的就没用了，直接pop
        for(int i=1;i<=n;i++) //用邻接表就是(for(int i=head[x];i,i=next[i])
        {
            if(d[i]>d[x]+z)d[y]=d[x]+z;
            q.push(make_pair(-d[y],y)); //更新，把新的d[y]扔进去
        }
        //这里其实可以看到邻接矩阵的劣势：其实根本不用枚举n个只要枚举相连边就ok
        //用邻接表效率更好
    }
}
```

Bellman-Ford算法： (SPFA)

最短路显然满足： $d[y] < d[x] + z$, 任意 (x, y) ；

操作是：1建立一个队列，起初只有起点1

2. 取出节点x, 扫描它的所有出边 (x, y, z) , 若 $dist[y] \gg dist[x] + z$, 则使用 $dist[x] + z$ 更新 $dist[y]$, 若y不在队列中则把y入队

3. 重复操作直到队列为空

注意到此处只是在迭代并未贪心取得最优（在非负前提贪心），这个迭代完全可以使用与边权为负数的情况

若用优先队列做一个局部优化：dj算法

代码把priority_queue换成queue就好

*/

/*

若是离线询问一条从1到N的路径中第k+1短路最小值：

法一：

可以二分答案：mid， 把大于mid的看作边长为一的边， 小于mid的看作长度为0的边， 然后求1-n最短路是否超过k

这个可以用双端队列深搜加广搜：若为1从队尾进入（广搜）更新边长，若为0则从队头进入（深搜模板），改进用A*算法，h值设为已知点之间最短值

（像并查集，树状数组的那个排序后在生成的思想）

!法二：分层图最短路：

可以考虑二元组 (x, p) 表示节点， x, p 到 (y, p) 有长度为z的边， (x, p) 到 $(y, p+1)$ 有长为0的边

那么假设k层n个点，把一个图的情况拷贝n次，一个点编号 $t, t+n, \dots, t+(k-1)*n$ 这种方式建图

然后dj算法

*/

任意两点的最短路径：

floyd算法：

设 $d(k, i, j)$ 表示“经过若干编号不超过 k 的节点”从 i 到 j 的最短路长度

dp: $d(k, i, j) = \min(d(k-1, i, j), d(k-1, i, k) + d(k-1, k, j))$

显然这里 k 才是“阶段”， i, j 是内层枚举：

dp如下：

```
int d[310][310], N, M;
```

```
int main()
{
```

```
    cin >> n >> m;
    memset(d, 0x3f, sizeof(d));
    for (int i = 1; i <= m; i++)
    {
        int x, y, z;
        scanf("%d%d%d", &x, &y, &z);
        d[x][y] = min(d[x][y], z);
    }
    for (int k = 1; k <= n; k++)
    {
        for (int i = 1; i <= n; i++)
        {
            for (int j = 1; j <= n; j++)
            {
                d[i][j] = min(d[i][j], d[i][k] + d[k][j]);
            }
        }
    }
    //output
```

```
}
```

```
*/
```

全源最短路之Johnson题解

任意两点间的最短路可以通过枚举起点，跑n次Bellman-Ford算法解决，时间复杂度是 n^2m

也可以直接floyd，复杂度为 n^3

若边权是都正的那用dj算法会很快， $nmlogm$ 量级

所以遇到有负边权的要进行预处理，但显然不能简单将边权平移某个大小，因为不能保证加入的边权权值占比会一样

Johnson算法如下：

1. 新建一个节点，设为0，从这个点引其他所有边边权为0的点
2. 用BELLMAN-Ford算法求出0号点到其他点单元最短路，即为 h_i
3. 若存在从 u 到 v 的点，该边权重新设置为 $w + h_u - h_v$

接下来以每个节点为起点，跑n轮dj算法，时间复杂度为 $nmlogm$

```

#include <cstring>
#include <iostream>
#include <queue>
#define INF 1e9
using namespace std;
struct edge {
    int v, w, next;
} e[10005];
struct node {
    int dis, id;
    bool operator<(const node& a) const { return dis > a.dis; }
    node(int d, int x) { dis = d, id = x; }
};
int head[5005], vis[5005], t[5005];
int cnt, n, m;
long long h[5005], dis[5005];
void addedge(int u, int v, int w) {
    e[++cnt].v = v;
    e[cnt].w = w;
    e[cnt].next = head[u];
    head[u] = cnt;
}
bool spfa(int s) {
    queue<int> q;
    memset(h, 63, sizeof(h));
    h[s] = 0, vis[s] = 1;
    q.push(s);
    while (!q.empty()) {
        int u = q.front();
        q.pop();
        vis[u] = 0;
        for (int i = head[u]; i; i = e[i].next) {
            int v = e[i].v;
            if (h[v] > h[u] + e[i].w) {
                h[v] = h[u] + e[i].w;
                if (!vis[v]) {
                    vis[v] = 1;
                    q.push(v);
                    t[v]++;
                    if (t[v] == n + 1) return false;
                }
            }
        }
    }
}

```

```

    }
    return true;
}
void dijkstra(int s) {
    priority_queue<node> q;
    for (int i = 1; i <= n; i++) dis[i] = INF;
    memset(vis, 0, sizeof(vis));
    dis[s] = 0;
    q.push(node(0, s));
    while (!q.empty()) {
        int u = q.top().id;
        q.pop();
        if (vis[u]) continue;
        vis[u] = 1;
        for (int i = head[u]; i; i = e[i].next) {
            int v = e[i].v;
            if (dis[v] > dis[u] + e[i].w) {
                dis[v] = dis[u] + e[i].w;
                if (!vis[v]) q.push(node(dis[v], v));
            }
        }
    }
    return;
}
int main() {
    ios::sync_with_stdio(false);
    cin >> n >> m;
    for (int i = 1; i <= m; i++) {
        int u, v, w;
        cin >> u >> v >> w;
        addedge(u, v, w);
    }
    for (int i = 1; i <= n; i++) addedge(0, i, 0);
    if (!spfa(0)) {
        cout << -1 << endl;
        return 0;
    }
    for (int u = 1; u <= n; u++)
        for (int i = head[u]; i; i = e[i].next) e[i].w += h[u] - h[e[i].v];
    for (int i = 1; i <= n; i++) {
        dijkstra(i);
        long long ans = 0;
        for (int j = 1; j <= n; j++) {

```

```

if (dis[j] == INF)
    ans += j * INF;
else
    ans += j * (dis[j] + h[j] - h[i]);
}
cout << ans << endl;
}
return 0;
}

```

最小生成树

定理：最小生成树一定含无向图权值最小的边

推论：给定一张无向图 $G = (v, e)$ ，从 e 中选出 $k < n-1$ 条边构成 G 的一个生成森林，若再从剩余 $m-k$ 条边中选出 $n-1-k$ 条边添加到森林里面

使 g 变成生成树。切边权和最小，则该生成树必含这 $m-k$ 条边中连接生成森林的两个不连通节点权值最小的边

Kruskal 算法：流程：

1. 建立并查集，每个点各自构成一个集合
2. 把所有边按权值从小到大排序，依次扫描每条边 (x, y, z)
3. 若 x, y 属于同一集合（连通），则忽略这条边，继续扫描下一条
4. 否则合并 x, y ，把 z 累加到答案里（过程可以理解为推论的反复使用）

时间复杂度 $O(m \log m)$

```

struct edge
{
    int x,y,z;
}edge[500010];
int fa[100010],n,m,ans;
bool operrator < (rec a, rec b)
{
    return a.z<b.z;
}
int get(int x)
{
    if(x==fa[x])
        return x;
    return fa[x]=get(fa[x]);
}
int main()
{
    cin>>n>>m;
    for(int i=1;i<=m;i++)
    {
        scanf("%d%d%d",&edge[i].x,&edge[i].y,&edge[i].z);
    }//按边权排序
    sort(edge+1,edge+m+1);
    for(int i=1;i<=n;i++)
    {
        fa[i]=i;
    }
    for(int i=1;i<=m;i++)
    {
        int x=get(edge[i].x);
        int y=get(edge[i].y);
        if(x==y)continue;
        fa[x]=y;//都不需要压缩路径直接连
        ans+=edge[i].z;
    }
    cout>>ans<<endl;
}

```

Prim算法：

(相当于上面的一个小优化)

任意时刻以确定的最小生成树集合为T，剩余集合为S。Prim算法找到 $\min(x \in S, y \in T) \{z\}$ 然后把x放到t中，z加到答案里

可以用数组标记是否属于t

适用范围：稠密图，用优先队列可优化到 $m\log n$

```
const int maxn=2e5+15;
```

```
const int mxn=5e3+15;
```

```
struct node  
{
```

```
    int t;int d;  
    bool operator < (const node &a) const  
{  
        return d>a.d;  
    }
```

```
} ;
```

```
int n,m;
```

```
int vis[mxn];  
vector e[mxn];
```

```
priority_queue q;
```

```
inline int read()  
{
```

```

char ch=getchar();
int s=0,f=1;
while (!(ch>='0'&&ch<='9')) {if (ch=='-') f=-1;ch=getchar();}
while (ch>='0'&&ch<='9') {s=(s<<3)+(s<<1)+ch-'0';ch=getchar();}
return s*f;

}

|| prim()
{

ll ans=0;
int cnt=0;
q.push((node){1,0});
while (!q.empty()&&cnt<=n)
{
    node k=q.top();q.pop();
    if (vis[k.t]) continue;
    vis[k.t]=1;
    ans+=k.d;
    cnt++;
    for (int i=0;i<e[k.t].size();i++)
        if (!vis[e[k.t][i].t]){
            q.push((node){e[k.t][i].t,e[k.t][i].d});
        }
}
return ans;

}

int main()
{

n=read();m=read();
for (int i=1;i<=m;i++)
{
    int x=read(),y=read(),z=read();
    e[x].push_back((node){y,z});e[y].push_back((node){x,z});
}
printf("%lld",prim());
return 0;
}

```

```
 }  
 }  
 */
```

找连通块：

用stack写个深搜就好

```
int k=0, cnt=0;  
  
stack sta;  
  
int v[maxn];  
  
void add(int x, int y, int z)//加入有向边 (x, y), 权值为z  
{  
  
    ver[++tot]=y; edge[tot]=z;  
    next[tot]=head[x]; head[x]=tot;//在表头x处插入  
  
}  
sta.push(1); k=1; cnt=1; v[1]=1;//cnt编号连通块;  
  
while(k<n)  
{
```

```

while(sta.size())
{
    int t=sta.top();
    sta.pop();
    for(int i=head[t];i;i=next[i])
    {
        if(!v[i])
        {
            v[i]=cnt;k++;
            sta.push(i);
        }
    }
    cnt++;
}

*/

```

树的直径

直径：最远两节点距离

法一：树形dp： $d[x]$ ：从节点x出发能到达节点最远距离

$$d[x] = \max_{1 \leq i \leq t} \{d[y_i] + \text{edge}(x, y_i)\}$$

$f[x]$ ：经过x的最长连长度：

$$f[x] = \max_{1 < j < i \leq t} \{d[y_i] + d[y_j] + \text{edge}(x, y_i) + \text{edge}(x, y_j)\}$$

注意到计算 $d[x]$ 枚举到 i 是已经是 $\max d[y_i] + \text{edge}(x, y_i)$

所以先用 $d[x] + d[y_i] + \text{edge}(x, y_i)$ 更新 $f[x]$ ，再用 $d[y_i] + \text{edge}(x, y_i)$ 更新 $d[x]$ 就好：

```

void dp(int x)
{

```

```

v[x]=1;
for(int i=head[x];i;i=next[i])
{
int y=ver[i];
if(v[y])continue;
dp(y);//树形dp一般采用递归
ans=max(ans,d[x]+d[y]+edge[i]);
d[x]=max(d[x],d[y]+edge[i]);
//上述两个更新
}
}

```

法二：两次搜索：从任意一个节点出发，遍历树，记录最远节点p

2. 从p出发，遍历树，求出距离p最远的节点q

p, q为直径

适用范围，边均为正

*/

最小公共祖先问题 (LCA)

树上倍增法：

记 $f[x, k]$ 为x的 2^k 次方祖先，若不存在即为0

$f[x, 0]$ 即为x的父节点

$f[x, k] = f(f[x, k-1], k-1)$ ； 预处理f数组

下为算法步骤：

$d[x]$ 标为x的深度

首先先两者调为同一高度

然后x, y同时向上调整 $k=\text{pow}(2, \log n)$ 2, 1步 (幂次从大到小试)

若未向会, 则 $x=f(x, k), y=f(y, k)$

复杂度: $O((n+m) \log n)$

```
const int size = 50010;

int f[size][20], d[size], dist[size];

int ver[2size], next[2size], edge[size*2], head[size];

int T, n, m, tot, t;

queue q;

void add(int x, int y, int z)
{
    ver[++tot]=y; edge[tot]=z; next[tot]=head[x]; head[x]=tot;
}

//邻接表

void bfs()
```

```

//预处理
q.push(1);d[1]=1;
while(q.size()){
    int x = q.front();q.pop();
    for(int i=head[x];i;i=next[i]){
        int y = ver[i];
        if(d[y])continue;
        d[y]=d[x]+1;
        dist[y]=dist[x]+edge[i];
        f[y][0]=x;
        for(int j=1;j<=t;j++)
            f[y][j]=f[f[y][j-1]][j-1];
        q.push(y);
    }
}
}

int lca(int x,int y)//回答一个询问
{
    if(d[x]>d[y])swap(x,y);
    for(int i=t;i>=0;i--)
        if(d[f[y][i]]>=d[x])y=f[y][i];//先调到同一高度
    if(x==y) return x;
    for(int i=t;i>=0;i--)
        if(f[x][i]!=f[y][i])x=f[x][i],y=f[y][i];
    return f[x][0];
}

int main()
{

```

```

cin>>T;
while(T--){
    cin>>n>>m;
    t=(int)(log(n)/log(2))+1;
    memset(head,0,sizeof(head));
    memset(d,0,sizeof(d));
    tot=0;
    //读树
    for(int i=1;i<=n;i++)
    {
        int x,y,z;
        scanf("%d%d",&x,&y);
        add(x,y,z);add(y,x,z);
    }
    bfs();
    for(int i=1;i<=m;i++)
    {
        int x, y;
        scanf("%d%d",&x,&y);
        printf("%d\n",dist[x]+dist[y]-2*dsit[lca(x,y)]);
    }
}
*/

```

lca的Tarjan算法：

离线算法，时间复杂度 $O(n+m)$ ，节点分为三类：

- 已经访问完毕并且回溯的节点，标记为2
- 已经开始递归，但尚未回溯的节点。这些节点就是当前正在访问的节点x及其祖先，标记为1
- 尚未访问的节点，这些节点没有标记。

若x正在访问，y已经访问完了，则lca (x, y) 就是从y向上走到根，第一个遇到的节点(拓扑序)

可以利用并查集优化，当一个节点标记为2的时候，把他所在集合合并到父节点里）（自底向上，合并时显然父节点标记为1）

code:

```

const int SIZE = 50010;
int ver[2*SIZE],next[2*SIZE],edge[2*SIZE],head[SIZE];
int fa[SIZE], d[SIZE], v[SIZE],lca[SIZE],ans[SIZE];
vector <int> query[SIZE],query_id[SIZE];
int T,n,m,tot,t;

void add(int x,int y,int z){
    ver[++tot]=y;edge[tot]=z;next[tot]=head[x];head[x]=tot;
}

void add_query (int x,int y ,int id){
    query[x].push_back(y),query_id[x].push_back(id);
    query[y].push_back(x),query_id[y].push_back(id);
}

int get(int x)
{
    if(x==fa[x]) return x;
    else
        return fa[x]=get(fa[x]);
}

//由上tarjan就是拓扑序加一个并查集优化
void tarjan(int x){
    v[x]=1;
    for(int i =head[x];i;i=next[x]){
        int y=ver[i];
        if(v[y]) continue;
        d[y] = d[x] +edge[i];
        tarjan(y);
        //递归为y的向上回溯
        fa[y]=x;
    }
    //x的子节点全部回溯完
    for(int i=0;i< query[x].size();i++)
    {
        int y=query[x][i],id=query[x][i];
        if (v[y]==2){
            int lca = get(y);//lca
            ans[id] = min(ans[id],d[x]+d[y]-2*d[lca]);//利用求的lca (x, y) 求得x到y的距离
        }
    }
    v[x]=2;
}

```

```

}

int main(){
    cin>>T;
    while(T--){
        cin >>n>>m;
        for(int i=1;i<=n;i++){
            head[i]=0;fa[i]=i;v[i]=0;
            query[i].clear(),query_id[i].clear();
        }
        tot = 0;
        for(int i=1;i<n;i++)
        {
            int x,y,z;
            sacnf("%d%d%d",&x,&y,&z);
            add(x,y,z),add(y,x,z);
        }
        for(int i=1;i<=m;i++)
        {
            int x,y;
            scanf("%d%d",&x,&y);
            if(x==y) ans[i] = 0;
            else{
                add_query(x,y,i);
                ans[i]=1<<30;
            }
        }
        tarjan(1);
        for(int i=1;i<=m;i++) printf("%d\n",ans[i]);
    }
}

```

二叉树前序，中序，后序遍历

https://blog.csdn.net/qq_61959780/article/details/127690872

前序遍历：code

/先序遍历/

```
void PreOrder(BiTree T)
{
    if (T != NULL)
    {
        visit(T);           // 访问结点
        PreOrder(T->lchild); // 遍历结点左子树
        PreOrder(T->rchild); // 遍历结点右子树
    }
}
```

/输出树结点/

```
void visit(BiTree T)
{
    printf("树结点的值: %c\n", T->data);
}
```

就是从左到右，到一个节点就输出一直左到底，

不行或结束了返回到右

先输出，再左树先序遍历，再右树先序遍历

/中序遍历/

```
void InOrder(BiTree T)
{
    if (T != NULL)
    {
        InOrder(T->lchild); // 遍历结点左子树
        visit(T);           // 访问结点
        InOrder(T->rchild); // 遍历结点右子树
    }
}
```

/输出树结点/

```
void visit(BiTree T)
{
    printf("树结点的值: %c\n", T->data);
}
```

先一直到左，不行了在输出此点，再到右一位，开始重复（详见递归）

先中序遍历，再输出，再有序遍历

/后序遍历/

```
void PostOrder(BiTree T)
{
    if (T != NULL)
    {
        PostOrder(T->lchild);      // 遍历结点左子树
        PostOrder(T->rchild);      // 遍历结点右子树
        visit(T);                  // 访问结点
    }
}
```

/输出树结点/

```
void visit(BiTree T)
{
    printf("树结点的值: %c\n", T->data);
}

*/
```

树的重心

定义：计算以无根树每个点为根节点时的最大子树大小，这个值最小的点称为无根树的重心。

性质1：某点是树的重心等价于它最大子树不大于整棵树大小的一半

那么根据这个结合深搜就可以找到重心

```
void dfs(int x){  
  
    v[x]=1;size[x]=1;//v记录是否访问过， size记录删除x后分成的最大子树的大小  
    int max_part = 0;  
    for(int i=head[x];i;i=next[i]){  
        int y = ver[i];  
        if(v[y])continue;  
        dfs(y);  
        size[x] += size[y];//从子节点向父节点递推  
        max_part = max(max_part,size[y]);  
    }  
    max_part = max(max_part,n-size[x]);  
    if(max_part<ans){  
        ans=max_part;//全局变量ans记录重心对应的max_part值  
        pos=x;//全局变量pos记录了重心  
    }  
}
```

性质2：树最多两个重心，若有两个则它们相邻

性质3：树中所有点到某个点的距离和中，到重心的距离和是最小的；如果有两个重心，那么到它们的距离和一样。反过来，距离和最小的点一定是重心。

性质4：往树上增加或减少一个叶子，如果原节点数是奇数，那么重心可能增加一个，原重心仍是重心；如果原节点数是偶数，重心可能减少一个，另一个重心仍是重心。

【模板】重链剖分/树链剖分

题目描述

如题，已知一棵包含 N 个结点的树（连通且无环），每个节点上包含一个数值，需要支持以下操作：

- $1 \ x \ y \ z$ ，表示将树从 x 到 y 结点最短路径上所有节点的值都加上 z 。
- $2 \ x \ y$ ，表示求树从 x 到 y 结点最短路径上所有节点的值之和。
- $3 \ x \ z$ ，表示将以 x 为根节点的子树内所有节点值都加上 z 。
- $4 \ x$ 表示求以 x 为根节点的子树内所有节点值之和

输入格式

第一行包含 4 个正整数 N, M, R, P ，分别表示树的结点个数、操作个数、根节点序号和取模数（即所有的输出结果均对此取模）。

接下来一行包含 N 个非负整数，分别依次表示各个节点上初始的数值。

接下来 $N - 1$ 行每行包含两个整数 x, y ，表示点 x 和点 y 之间连有一条边（保证无环且连通）。

接下来 M 行每行包含若干个正整数，每行表示一个操作。

输出格式

输出包含若干行，分别依次表示每个操作 2 或操作 4 所得的结果（对 P 取模）。

方法：树链剖分

树链剖分

树链剖分 就是对一棵树分成几条链，把树形变为线性，减少处理难度

概念

重儿子：对于每一个非叶子节点，它的儿子中 以那个儿子为根的子树节点数最大的儿子 为该节点的重儿子

轻儿子：对于每一个非叶子节点，它的儿子中 非重儿子 的剩下所有儿子即为轻儿子
叶子节点没有重儿子也没有轻儿子（因为它没有儿子。。）

重边：一个父亲连接他的重儿子的边称为重边 //原写法：连接任意两个重儿子的边叫做重边

轻边：剩下的即为轻边

重链：相邻重边连起来的 连接一条重儿子 的链叫重链

对于叶子节点，若其为轻儿子，则有一条以自己为起点的长度为1的链

每一条重链以轻儿子为起点

dfs1()

这个dfs要处理几件事情：

标记每个点的深度dep[]

标记每个点的父亲fa[]

标记每个非叶子节点的子树大小(含它自己)

标记每个非叶子节点的重儿子编号son[]

```
inline void dfs1(int x,int f,int deep){//x当前节点, f父亲, deep深度
    dep[x]=deep;//标记每个点的深度
    fa[x]=f;//标记每个点的父亲
    siz[x]=1;//标记每个非叶子节点的子树大小
    int maxson=-1;//记录重儿子的儿子数
    for(Rint i=begin[x];i;i=nex[i]){
        int y=to[i];
        if(y==f)continue;//若为父亲则continue
        dfs1(y,x,deep+1);//dfs其儿子
        siz[x]+=siz[y];//把它的儿子数加到它身上
        if(siz[y]>maxson)son[x]=y,maxson=siz[y];//标记每个非叶子节点的重儿子编号
    }
}
```

dfs2()

这个dfs2也要预处理几件事情

标记每个点的新编号

赋值每个点的初始值到新编号上

处理每个点所在链的顶端

处理每条链

顺序：先处理重儿子再处理轻儿子，理由后面说

```
inline void dfs2(int x,int topf){//x当前节点, topf当前链的最顶端的节点
    id[x]=++cnt;//标记每个点的新编号
    wt[cnt]=w[x];//把每个点的初始值赋到新编号上来
    top[x]=topf;//这个点所在链的顶端
    if(!son[x])return;//如果没有儿子则返回
    dfs2(son[x],topf);//按先处理重儿子, 再处理轻儿子的顺序递归处理
    for(Rint i=begin[x];i;i=nex[i]){
        int y=to[i];
        if(y==fa[x]||y==son[x])continue;
        dfs2(y,y);//对于每一个轻儿子都有一条从它自己开始的链
    }
}
```

第三步

因为顺序是先重再轻，所以每一条重链的新编号是连续的

因为是dfs，所以每一个子树的新编号也是连续的（以上两者达成剖分成链的条件）

现在回顾一下我们要处理的问题

处理任意两点间路径上的点权和

处理一点及其子树的点权和

修改任意两点间路径上的点权

修改一点及其子树的点权

1、当我们要处理任意两点间路径时：

设所在链顶端的深度更深的那个点为x点

ans加上x点到x所在链顶端 这一段区间的点权和

把x跳到x所在链顶端的那个点的上面一个点

不停执行这两个步骤，直到两个点处于一条链上，这时再加上此时两个点的区间和即可

这时我们注意到，我们所要处理的所有区间均为连续编号(新编号)，于是想到线段树，用线段树处理连续编号区间和

每次查询时间复杂度为 $O(\log n^2)$

```
inline int qRange(int x,int y){
    int ans=0;
    while(top[x]!=top[y]){//当两个点不在同一条链上
        if(dep[top[x]]<dep[top[y]])swap(x,y);//把x点改为所在链顶端的深度更深的那个点
        res=0;
        query(1,1,n,id[top[x]],id[x]);//ans加上x点到x所在链顶端 这一段区间的点权和
        ans+=res;
        ans%=mod;//按题意取模
        x=fa[top[x]];//把x跳到x所在链顶端的那个点的上面一个点
    }
    //直到两个点处于一条链上
    if(dep[x]>dep[y])swap(x,y);//把x点深度更深的那个点
    res=0;
    query(1,1,n,id[x],id[y]);//这时再加上此时两个点的区间和即可
    ans+=res;
    return ans%mod;
}//变量解释见最下面
```

2、处理一点及其子树的点权和：

想到记录了每个非叶子节点的子树大小(含它自己)，并且每个子树的新编号都是连续的

于是直接线段树区间查询即可

时间复杂度为 $O(\log n)$

```

inline int qSon(int x){
    res=0;
    query(1,1,n,id[x],id[x]+siz[x]-1);//子树区间右端点为id[x]+siz[x]-1
    return res;
}
//查询
inline void updRange(int x,int y,int k){
    k%=mod;
    while(top[x]!=top[y]){
        if(dep[top[x]]<dep[top[y]])swap(x,y);
        update(1,1,n,id[top[x]],id[x],k);
        x=fa[top[x]];
    }
    if(dep[x]>dep[y])swap(x,y);
    update(1,1,n,id[x],id[y],k);
}
inline void updSon(int x,int k){
    update(1,1,n,id[x],id[x]+siz[x]-1,k);
}//变量解释见最下面

```

ac代码：

```

#include<algorithm>
#include<iostream>
#include<cstdlib>
#include<cstring>
#include<cstdio>
#define Rint register int
#define mem(a,b) memset(a,(b),sizeof(a))
#define Temp template<typename T>
using namespace std;
typedef long long LL;
Temp inline void read(T &x){
    x=0;T w=1,ch=getchar();
    while(!isdigit(ch)&&ch!='-')ch=getchar();
    if(ch=='-')w=-1,ch=getchar();
    while(isdigit(ch))x=(x<<3)+(x<<1)+(ch^'0'),ch=getchar();
    x=x*w;
}
#define mid ((l+r)>>1)
#define lson rt<<1,l,mid
#define rson rt<<1|1,mid+1,r
#define len (r-l+1)

const int maxn=200000+10;
int n,m,r,mod;
//见题意
int e,beg[maxn],nex[maxn],to[maxn],w[maxn],wt[maxn];
//链式前向星数组， w[]、 wt[]初始点权数组
int a[maxn<<2],laz[maxn<<2];
//线段树数组、 lazy操作
int son[maxn],id[maxn],fa[maxn],cnt,dep[maxn],siz[maxn],top[maxn];
//son[]重儿子编号,id[]新编号,fa[]父亲节点,cnt dfs_clock/dfs序,dep[]深度,siz[]子树大小,top[]当前链尾
int res=0;
//查询答案

inline void add(int x,int y){//链式前向星加边
    to[+e]=y;
    nex[e]=beg[x];
    beg[x]=e;
}
//-----以下为线段树
inline void pushdown(int rt,int lenn){
    laz[rt<<1]+=laz[rt];
}

```

```

        laz[rt<<1|1]+=laz[rt];
        a[rt<<1]+=laz[rt]*(lenn-(lenn>>1));
        a[rt<<1|1]+=laz[rt]*(lenn>>1);
        a[rt<<1]%=mod;
        a[rt<<1|1]%=mod;
        laz[rt]=0;
    }

inline void build(int rt,int l,int r){
    if(l==r){
        a[rt]=wt[l];
        if(a[rt]>mod)a[rt]%=mod;
        return;
    }
    build(lson);
    build(rson);
    a[rt]=(a[rt<<1]+a[rt<<1|1])%mod;
}

inline void query(int rt,int l,int r,int L,int R){
    if(L<=l&&r<=R){res+=a[rt];res%=mod;return;}
    else{
        if(laz[rt])pushdown(rt,len);
        if(L<=mid)query(lson,L,R);
        if(R>mid)query(rson,L,R);
    }
}

inline void update(int rt,int l,int r,int L,int R,int k){
    if(L<=l&&r<=R){
        laz[rt]+=k;
        a[rt]+=k*len;
    }
    else{
        if(laz[rt])pushdown(rt,len);
        if(L<=mid)update(lson,L,R,k);
        if(R>mid)update(rson,L,R,k);
        a[rt]=(a[rt<<1]+a[rt<<1|1])%mod;
    }
}
//-----以上为线段树
inline int qRange(int x,int y){
    int ans=0;

```

```

while(top[x] != top[y]){//当两个点不在同一条链上
    if(dep[top[x]] < dep[top[y]]) swap(x,y);//把x点改为所在链顶端的深度更深的那个点
    res=0;
    query(1,1,n,id[top[x]],id[x]);//ans加上x点到x所在链顶端 这一段区间的点权和
    ans+=res;
    ans%=mod;//按题意取模
    x=fa[top[x]];//把x跳到x所在链顶端的那个点的上面一个点
}
//直到两个点处于一条链上
if(dep[x] > dep[y]) swap(x,y);//把x点深度更深的那个点
res=0;
query(1,1,n,id[x],id[y]);//这时再加上此时两个点的区间和即可
ans+=res;
return ans%mod;
}

inline void updRange(int x,int y,int k){//同上
k%=mod;
while(top[x] != top[y]){
    if(dep[top[x]] < dep[top[y]]) swap(x,y);
    update(1,1,n,id[top[x]],id[x],k);
    x=fa[top[x]];
}
if(dep[x] > dep[y]) swap(x,y);
update(1,1,n,id[x],id[y],k);
}

inline int qSon(int x){
res=0;
query(1,1,n,id[x],id[x]+siz[x]-1);//子树区间右端点为id[x]+siz[x]-1
return res;
}

inline void updSon(int x,int k){//同上
update(1,1,n,id[x],id[x]+siz[x]-1,k);
}

inline void dfs1(int x,int f,int deep){//x当前节点, f父亲, deep深度
dep[x]=deep;//标记每个点的深度
fa[x]=f;//标记每个点的父亲
siz[x]=1;//标记每个非叶子节点的子树大小
int maxson=-1;//记录重儿子的儿子数
for(Rint i=beg[x];i;i=nex[i]){
}
}

```

```

        int y=to[i];
        if(y==f)continue;//若为父亲则continue
        dfs1(y,x,deep+1);//dfs其儿子
        siz[x]+=siz[y];//把它的儿子数加到它身上
        if(siz[y]>maxson)son[x]=y,maxson=siz[y];//标记每个非叶子节点的重儿子编号
    }
}

inline void dfs2(int x,int topf){//x当前节点，topf当前链的最顶端的节点
    id[x]=++cnt;//标记每个点的新编号
    wt[cnt]=w[x];//把每个点的初始值赋到新编号上来
    top[x]=topf;//这个点所在链的顶端
    if(!son[x])return;//如果没有儿子则返回
    dfs2(son[x],topf);//按先处理重儿子，再处理轻儿子的顺序递归处理
    for(Rint i=begin[x];i;i=nex[i]){
        int y=to[i];
        if(y==fa[x]||y==son[x])continue;
        dfs2(y,y);//对于每一个轻儿子都有一条从它自己开始的链
    }
}

int main(){
    read(n);read(m);read(r);read(mod);
    for(Rint i=1;i<=n;i++)read(w[i]);
    for(Rint i=1;i<n;i++){
        int a,b;
        read(a);read(b);
        add(a,b);add(b,a);
    }
    dfs1(r,0,1);
    dfs2(r,r);
    build(1,1,n);
    while(m--){
        int k,x,y,z;
        read(k);
        if(k==1){
            read(x);read(y);read(z);
            updRange(x,y,z);
        }
        else if(k==2){
            read(x);read(y);
            printf("%d\n",qRange(x,y));
        }
    }
}

```

```

else if(k==3){
    read(x);read(y);
    updSon(x,y);
}
else{
    read(x);
    printf("%d\n",qSon(x));
}
}
}

```

树上差分

一，修改点权，要求u, v, 路径经过的点全部增加x

利用差分这样改， $\text{num}[u] += x, \text{num}[v] += x, \text{num}[\text{lca}(u, v)] -= x, \text{num}[\text{fa}[\text{lca}(u, v)]] -= x;$

查询点权：以每个节点为根的节点权值和！这个深搜就可以办到。

二，边差分，u, v的路径上的所有边加x

操作： $\text{num}[u] += x, \text{num}[v] += x, \text{num}[\text{lca}(x, y)] -= 2*x;$

查询边权：以每个节点为根的节点权值和，这个深搜就可以办到。