聪明的投资者

雪球

代码行向左或向右缩进: Ctrl+[、 Ctrl+]

复制或剪切当前行/当前选中内容: Ctrl+C 、 Ctrl+V

代码格式化: Shift+Alt+F

向上或向下移动一行: Alt+Up 或 Alt+Down

向上或向下复制一行: Shift+Alt+Up 或 Shift+Alt+Down

在当前行下方插入一行: Ctrl+Enter

在当前行上方插入一行: Ctrl+Shift+Enter

删除当前行 Ctrl+Shift+k // 反复做知识点，加快速度， 万变不离其宗

初始化操作

头文件

```
#include <iostream>
#include <string>
#include <cstdlib>
#include <algorithm>
#include <cmath>
#include <cstring>
#include <cstdio>
#include <vector>
#include <queue>
#include <assert.h>
#include <map>
#include <set>
#include <bitset>
#include <iomanip>
#include <stack>
#include <unordered_map>
#include <cmath>
using namespace std;
#pragma comment(linker, "/STACK:1024000000,1024000000")
#define INF 0x7f7f7f7f  //2139062143
#define INF1 0x3f3f3f3f //1061109567
#define INF2 2147483647
#define llINF 9223372036854775807
#define pi 3.141592653589793238462643383279502 54
#define pb push_back
#define ll long long
#define debug cout << "debug\n";
#define STDIN                      \
    freopen("in.txt", "r", stdin); \
    freopen("out.txt", "w", stdout);
#define IOS                        \
    ios::sync_with_stdio(false); \
```

```
    cin.tie(NULL);
#define scai(x) scanf("%d", &x)
#define sca2i(x, y) scanf("%d %d", &x, &y)
#define scaf(x) scanf("%lf", &x)
#define sca2f(x, y) scanf("%lf %lf", &x, &y)
#define For(m, n) for (int i = m; i < n; i++)
#define PII pair<int, int>
#define PLL pair<long, long>
#define ft first
#define sd second
#define pb push_back
#define lson o << 1, l, mid
#define rson o << 1 | 1, mid + 1, r
#define FOR(i, a, b) for (int i = (a); i <= (b); i++)
#define ROF(i, a, b) for (int i = (a); i >= (b); i--)
#define MEM(x, v) memset(x, v, sizeof(x))
#define rep(i, a, b) for (int i = a; i <= b; i++)
#define rrep(i, a, b) for (int i = a; i >= b; i--)
#define PIII pair<int, PII>

cout.precision(20); // 设置输出精度
```

整数初始化

```
const int INF = 0x7fffffff; // int的最大值
const int INF = 0x3f3f3f3f; // 一般用这个，和0x3fffffff一个数量级，但和其他数相加不会溢出
```

数组初始化

```
memset(a, 0, sizeof(a));
memset(a, -1, sizeof(a));
memset(a, 0x3f, sizeof(a)); // memset这个函数是按字节来赋值的，int有4个字节，所以把每个字节都赋值成0x3f以后就是0x3f3f3f3f
```

小技巧

1. 01区间取反： 每次取反区间差分+1，最后求和%2就可判断该数是0还是1; poj2155

基础算法

quick_sort:

```
// 1.找到分界点x，q[L]，q[(L + R)>>1]，q[R];
// 2.左边所有数Left <= x，右边所有数Right>=x
// 3.递归排序Left，递归排序Right

void quick_sort(int q[], int l, int r)
{
```

```
    if (l >= r) return;

    int i = l - 1, j = r + 1, x = q[l + r >> 1];
    while (i < j)
    {
        do i ++ ; while (q[i] < x);
        do j -- ; while (q[j] > x);
        if (i < j) swap(q[i], q[j]);
    }
    quick_sort(q, l, j), quick_sort(q, j + 1, r);
}
```

快速选择算法

```
int quick_sort(int l, int r, int k)
{
    if (l == r) return a[l];
    int x = a[(l+r)>>1], i = l-1, j = r+1;
    while(i<j)
    {
        while(a[++i]<x);
        while(a[--j]>x);
        if (i<j) swap(a[i],a[j]);
    }
    int sl = j-l+1;
    if (k<=sl) return quick_sort(l,j,k);
    else return quick_sort(j+1, r, k-sl);
}
```

归并排序

```
// 1.确定分界点 mid = (l+r)/2
// 2. 递归排序左边和右边
// 1. 归并，合二为一 难点 稳定

void merge_sort(int q[], int l, int r)
{
    if (l >= r) return;

    int mid = l + r >> 1;
    merge_sort(q, l, mid);
    merge_sort(q, mid + 1, r);

    int k = 0, i = l, j = mid + 1;
    while (i <= mid && j <= r)
        if (q[i] < q[j]) tmp[k ++ ] = q[i ++ ];
        else tmp[k ++ ] = q[j ++ ];

    while (i <= mid) tmp[k ++ ] = q[i ++ ];
    while (j <= r) tmp[k ++ ] = q[j ++ ];

    for (i = l, j = 0; i <= r; i ++, j ++ ) q[i] = tmp[j];
}
```

逆序对数

```
// 1.左半边内部逆序对数量 mergesort(l, mid)
// 2.右半边内部逆序对数量 mergesort(mid+1, r)
// 3.
ll merge_sort(int a[], int l, int r)
{
    if (l>=r) return 0ll;
    int mid = l + r >> 1;
    ll res = merge_sort(a, l, mid) + merge_sort(a, mid + 1, r);
    int k = 0, i = l, j = mid+1;
    while(i<=mid && j <= r)
    {
        if (a[i]<=a[j]) tmp[k++] = a[i++];
        else
        {
            res+=mid-i+1;
            tmp[k++] = a[j++];
        }
    }
    while(i<=mid) tmp[k++] = a[i++];
    while(j<=r) tmp[k++] = a[j++];
    for (int i = l, j = 0; j < k; j++, i++) a[i] = tmp[j];
    return res;
}
```

大数模拟

大数相加： 例题 https://ac.nowcoder.com/acm/contest/3005/E

大数取模

```
int mod(string a, int b) //高精度a除以单精度b
{
    int d = 0ll;
    for (int i = 0; i < a.size(); i++)  d = (d * 10 + (a[i] - '0')) % b;  //求出
余数
    return d;
}
```

二分

整数二分

```
// 1.整数二分
// 有单调性的一定可以二分， 没有单调性的可能可以二分
// 二分的本质不是单调性,是边界性质
bool check(int x) {/* ... */} // 检查x是否满足某种性质

// 区间[l, r]被划分成[l, mid]和[mid + 1, r]时使用:
int bsearch_1(int l, int r)
{
```

```
    while (l < r)
    {
        int mid = l + r >> 1;
        if (check(mid)) r = mid;    // check()判断mid是否满足性质
        else l = mid + 1;
    }
    return l;
}
// 区间[l, r]被划分成[l, mid - 1]和[mid, r]时使用：
int bsearch_2(int l, int r)
{
    while (l < r)
    {
        int mid = l + r + 1 >> 1;
        if (check(mid)) l = mid;
        else r = mid - 1;
    }
    return l;
}
```

浮点数二分

```
bool check(double x) {/* ... */} // 检查x是否满足某种性质

double bsearch_3(double l, double r)
{
    const double eps = 1e-6;   // eps 表示精度，取决于题目对精度的要求
    while (r - l > eps)
    {
        double mid = (l + r) / 2;
        if (check(mid)) r = mid;
        else l = mid;
    }
    return l;
}
```

前缀和

一维前缀和

```
S[i] = a[1] + a[2] + ... a[i]
a[l] + ... + a[r] = S[r] - S[l - 1]
```

二维前缀和

```
S[i, j] = 第i行j列格子左上部分所有元素的和
s[i,j] = s[i-1, j] + s[i,j-1] + a[i][j];
以(x1, y1)为左上角，(x2, y2)为右下角的子矩阵的和为:
S[x2, y2] - S[x1 - 1, y2] - S[x2, y1 - 1] + S[x1 - 1, y1 - 1]
```

差分

一维差分

```
给区间[l, r]中的每个数加上c：B[l] += c, B[r + 1] -= c
```

二维差分

```
给以(x1, y1)为左上角，(x2, y2)为右下角的子矩阵中的所有元素加上c：
S[x1, y1] += c, S[x2 + 1, y1] -= c, S[x1, y2 + 1] -= c, S[x2 + 1, y2 + 1] += c
```

双指针算法

1.先暴力 2.找找单调性

```
for (int i = 0, j = 0; i < n; i ++ )
{
    while (j < i && check(i, j)) j ++ ;

    // 具体问题的逻辑
}
常见问题分类：
    (1) 对于一个序列，用两个指针维护一段区间
    (2) 对于两个序列，维护某种次序，比如归并排序中合并两个有序序列的操作
```

位运算

```
求n的第k位数字: n >> k & 1
返回n的最后一位1：lowbit(n) = n & -n
```

离散化

```
vector<int> alls; // 存储所有待离散化的值
sort(alls.begin(), alls.end()); // 将所有值排序
alls.erase(unique(alls.begin(), alls.end()), alls.end());   // 去掉重复元素

// 二分求出x对应的离散化的值
int find(int x) // 找到第一个大于等于x的位置
{
    int l = 0, r = alls.size() - 1;
    while (l < r)
    {
        int mid = l + r >> 1;
        if (alls[mid] >= x) r = mid;
        else l = mid + 1;
    }
    return r + 1; // 映射到1, 2, ...n
}
```

区间合并

```
// 将所有存在交集的区间合并
void merge(vector<PII> &segs)
{
    vector<PII> res;

    sort(segs.begin(), segs.end());

    int st = -2e9, ed = -2e9;
    for (auto seg : segs)
        if (ed < seg.first)
        {
            if (st != -2e9) res.push_back({st, ed});
            st = seg.first, ed = seg.second;
        }
        else ed = max(ed, seg.second);

    if (st != -2e9) res.push_back({st, ed});

    segs = res;
}
```

数学知识

同余

两个整数a、b，若它们除以整数m所得的余数相等，则称a与b对于模m同余或a同余于b模m。

记作：a≡b (mod m)即(a-b)/m是一个整数

性质

1.反身性：$a \equiv a(modm)$；

2.对称性：若$a \equiv b(modm)$，则$b \equiv a(modm)$；

3.传递性：若$a \equiv b(modm)$，$b \equiv c(modm)$，则$a \equiv c(modm)$；

4.同余式相加：若$a \equiv b(modm)$，$c \equiv d(modm)$，则$a + c \equiv b + d(modm)$；

5.同余式相乘：若$a \equiv b(modm)$，$c \equiv d(modm)$，则$ac \equiv bd(modm)$。

费马小定理

费马小定理是数论中的一个定理：加入a是一个整数，p是一个质数，那么$a^p - a$是p的倍数，可以表示为

$a^p \equiv a(modp)$

如果a不是p的倍数，这个定理也可以写成

$a^{p-1} \equiv 1(modp)$这个书写方式更加常用

欧拉定理

在数论中，欧拉定理是一个关于同余的性质。欧拉定理表明，若n,a为正整数,且n,a互素，则

$a^{\phi(n)} \equiv 1 (mod n)$

容斥原理

质数

质数的判定-试除法

```
bool is_prime(int x)
{
    if (x < 2) return false;
    for (int i = 2; i <= x / i; i ++ )
        if (x % i == 0)
            return false;
    return true;
}
```

分解质因数-试除法

```
void divide(int x)
{
    for (int i = 2; i <= x / i; i ++ )
        if (x % i == 0)
        {
            int s = 0;
            while (x % i == 0) x /= i, s ++ ;
            cout << i << ' ' << s << endl;
        }
    if (x > 1) cout << x << ' ' << 1 << endl;
    cout << endl;
}
```

埃氏筛法

```
int primes[N], cnt;     // primes[]存储所有素数
bool st[N];         // st[x]存储x是否被筛掉

void get_primes(int n)
{
    for (int i = 2; i <= n; i ++ )
    {
        if (st[i]) continue;
        primes[cnt ++ ] = i;
        for (int j = i; j <= n; j += i)
            st[j] = true;
    }
}
```

线性筛法

```
//线性筛法-O(n), n = 1e7的时候基本就比埃式筛法快一倍了
//算法核心：x仅会被其最小质因子筛去
int primes[N], cnt;     // primes[]存储所有素数
bool st[N];         // st[x]存储x是否被筛掉

void get_primes(int n)
{
    for (int i = 2; i <= n; i ++ )
    {
        if (!st[i]) primes[cnt ++ ] = i;
        for (int j = 0; primes[j] <= n / i; j ++ )
        {
            st[primes[j] * i] = true;
            if (i % primes[j] == 0) break;
        }
    }
}
```

线性筛法求莫比乌斯函数mobius

```
int primes[N], cnt;
bool st[N];
int n;
int mu[N];
void getMu()
{
    mu[1] = 1;
    for (int i = 2; i <= n; i++)
    {
        if (!st[i]) primes[cnt++] = i, mu[i] = -1;
        for (int j = 0; primes[j] <= n / i; ++j)
        {
            st[primes[j]*i] = true;
            if (i % primes[j] == 0)
            {
                mu[primes[j]*i] = 0;
                break;
            }
            mu[primes[j]*i] = -mu[i];
        }
    }
}
```

mobius反演

已知$f(n) = \sum_{d|n} g(d)$

那么$g(n) = \sum_{d|n} u(d) * f(\frac{n}{d}))$

约数

试除法求约数

```
vector<int> get_divisors(int x)
{
    vector<int> res;
    for (int i = 1; i <= x / i; i ++ )
        if (x % i == 0)
        {
            res.push_back(i);
            if (i != x / i) res.push_back(x / i);
        }
    sort(res.begin(), res.end());
    return res;
}
```

约数个数和约数之和

如果 $N = p_1^{c_1} * p_2^{c_2} * \ldots * p_k^{c_k}$ 约数个数：$(c_1+1)*(c_2+1)*\ldots*(c_k+1)$ 约数之和：$(p_1^0+p_1^1+\ldots+p_1^{c_1})*\ldots*(p_k^0+p_k^1+\ldots+p_k^{c_k})$

```
// 求约数个数
const int mod = 1e9 + 7;
int main()
{
    int n; cin >> n;
    unordered_map<int, int> res;
    while (n--)
    {
        int k; cin >> k;
        for (int i = 2; i <= k/i; i++)
        {
            while(k%i == 0)
            {
                res[i]++;
                k/=i;
            }
        }
        if (k > 1) res[k]++;
    }
    long long ans = 1;
    for (auto i:res) ans = ans*(i.second+1)%mod;
    cout << ans << endl;
}
```

```
// 求约数之和
unordered_map<int, int> primes;
while (n--)
{
    int x; cin >> x;
    for (int i = 2; i <= x/i; i++)
    {
        while (x%i == 0)
        {
            x/=i;
            primes[i]++;
```

```
        }
    }
    if (x > 1) primes[x]++;
    LL res = 1;
    for (auto p:primes)
    {
        LL a = p.first, b = p.second;
        LL t = 1;
        whiel (b--) t = (t*a+1)%mod;
        res = res*t%mod;
    }
}
```

欧几里得算法

```
int gcd(int a, int b)
{
    return b?gcd(b, a%b):a;
}
// 在c++中可以直接调用
__gcd(a,b);
```

- gdc(a1, a2, a3, a4 ..... an) = gcd(a1, a2 - a1, a3 - a2, ... an - an-1);
- gcd(a, b, c) = gcd(gcd(a, b), c)

lcm=a/gcd*b

欧拉函数

1~N 中与N互质的数的个数被称为欧拉函数，记为$\phi(N)$

若在算数基本定理中，$N = p_1^{a_1} p_2^{a_2} \cdots p_m^{a_m}$,则：

$\phi(N) = N * \frac{p_1-1}{p_1} * \frac{p_2-1}{p2} * \cdots * \frac{p_m-1}{p_m}$

```

int res=x;
for(int i=2;i<=x/i;i++){
    if(x%i==0){
        res=res/i*(i-1);    //防止溢出
        while(x%i==0) x/=i;
    }
}
if(x>1) res=res/x*(x-1);
cout<<res<<endl;
```

筛法求欧拉函数

	质数i的欧拉函数即为`phi[i] = i - 1`：1 ~ i-1均与i互质，共i-1个。
	`phi[primes[j] * i]`分为两种情况：
	① `i % primes[j] == 0`时：`primes[j]`是`i`的最小质因子，也是`primes[j] * i`的最小质因子，因此`1 - 1 / primes[j]`这一项在`phi[i]`中计算过了，只需将基数N修正为`primes[j]`倍，最终结果为`phi[i] * primes[j]`。
	② `i % primes[j] != 0`：`primes[j]`不是`i`的质因子，只是`primes[j] * i`的最小质因子，因此不仅需要将基数N修正为`primes[j]`倍，还需要补上`1 - 1 / primes[j]`这一项，因此最终结果`phi[i] * (primes[j] - 1)`。

```
                                                  |int primes[N], cnt;     //
primes[]存储所有素数
int euler[N];           // 存储每个数的欧拉函数
bool st[N];         // st[x]存储x是否被筛掉


void get_eulers(int n)
{
    euler[1] = 1;
    for (int i = 2; i <= n; i ++ )
    {
        if (!st[i])
        {
            primes[cnt ++ ] = i;
            euler[i] = i - 1;
        }
        for (int j = 0; primes[j] <= n / i; j ++ )
        {
            int t = primes[j] * i;
            st[t] = true;
            if (i % primes[j] == 0)
            {
                euler[t] = euler[i] * primes[j];
                break;
            }
            euler[t] = euler[i] * (primes[j] - 1);
        }
    }
}
```

快速乘

```
ll mult_mod(ll a,ll b,ll mod){
    return (a*b-(ll)(a/(long double)mod*b+1e-3)*mod+mod)%mod;
}
```

快速幂

普通

求m^k%p，时间复杂度 O(log_k)。

```
int qmi(int m, int k, int p)
{
    int res = 1 % p, t = m;
    while (k)
    {
        if (k&1) res = res * t % p;
        t = t * t % p;
        k >>= 1;
    }
    return res;
}
```

快速乘快速幂

```
ll mult_mod(ll a,ll b,ll mod){
    return (a*b-(ll)(a/(long double)mod*b+1e-3)*mod+mod)%mod;
}
ll pow_mod(ll x, ll n, ll mod) { //x^n%c
    if(n == 1)return x % mod;
    x %= mod;
    ll tmp = x;
    ll ret = 1;
    while(n) {
        if(n & 1) ret = mult_mod(ret, tmp, mod);
        tmp = mult_mod(tmp, tmp, mod);
        n >>= 1;
    }
    return ret;
}
```

扩展欧几里得求解逆元

- 这个方法速度快于快速幂求逆元
- 如果逆元算出来为0则说明不存在逆元

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

    for (ll i = 1ll; i <= n; i++)
    {
        ll a, p; cin >> a >> p;
        ll x, y;
        Exgcd(a,p,x,y);
        x = (x % p + p) % p;
        if (x)
        cout << x << endl;
        else puts("impossible");
    }
```

```
}
```

快速幂求逆元

`ll x = qmi(a, p - 2, p); //x为a在mod p意义下的逆元` 当然如果x能够整除p的话，不存在逆元 https://ac.nowcoder.com/acm/contest/3005/C 这道题还可以用线段树来做

线性算法

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

- https://www.luogu.com.cn/problem/P3811

矩阵快速幂

```

typedef vector<int> vec;
typedef vector<vec> mat;
typedef long long ll;
const int M = 10000;

// 计算A*B
mat nul(mat &A, mat &B)
{
    mat C(A.size(), vec(B[0].size()));
    for (int i = 0)
}
```

裴蜀定理

$ax + by = \gcd(a, b)$

高斯消元求解线性方程组

求组合数

递归法求组合数

时间复杂度$o(n^2)$

```
// c[a][b] 表示从a个苹果中选b个的方案
for (int i = 0; i < N; i++)
    for (int j = 0; j <= i; j++)
    {
        if (!j) c[i][j] = 1;
        else c[i][j] = (c[i - 1][j] + c[i - 1][j - 1]) % mod;
    }
```

通过预处理逆元的方式求组合数

首先预处理出所有阶乘取模的余数`fact[N]`，以及所有阶乘取模的逆元`infact[N]`

如果取模的数是质数，可用费马小定理求逆元

数据范围：$1 \le a \le b \le 1e5$

```
int qmi(int a, int k, int p) // 快速幂模板
{
    int res = 1;
    while (k)
    {
        if (k & 1) res = (LL) res*a%p;
        a = (LL) a * a % p;
        k >>= 1;
    }
    return res;
}
// 预处理阶乘的余数和阶乘的逆元的余数
fact[0] = infact[0] = 1;
for (int i = 1; i < N; i++)
{
    fact[i] = (LL) fact[i-1] * i % mod;
    infact[i] = (LL)infact[i - 1] * qmi(i, mod - 2, mod) % mod;
}
```

Lucas（卢卡斯）定理

用来解决大组合数求模是很有用的 Lucas定理最大的数据处理能力是p在10^5左右，不能再大了

https://ac.nowcoder.com/acm/contest/4381/B

卡特兰数

容斥原理

抽屉原理

多项式

秦九韶定理

$p(x) = 2x^4 - x^3 + 3x^2 + x - 5$

$= x(2x^3 - x^2 + 3x + 1) - 5$

$= x(x(2x^2 - x + 3) + 1) - 5$

$= x(x(x(2x - 1) + 3) + 1) - 5$

本原多项式

1. 设$f(x) = a_0 + a_1x + a_2x^2 + ... + a_nx^n$是唯一分解整环D上的多项式，如果$gcd(a_0, a_1, ..., a_n) = 1$, 则称$f(x)$为D上的一个本原多项式
   - $f(x)$ 是既约的，即不能再分解因式；

    - $f(x)$可整除x^m+1, 这里的$m=2^n-1$;
    - $f(x)$不能整除x^q+1, 这里$q<m$.
2. 定理
    - 高斯引理：本原多项式的乘积还是本原多项式

函数求峰

函数单峰

三分法

就是比较定义域中的两个三等分点的映射值，若左边的三等分点比较大，则将右边界右移，对于左边界同理，最终不断逼近得到单峰的位置。

```
ll ef(ll l,ll r)//三分
{
    while(r - l > eps) //保证精度，最好是k+2位精度哦！
    {
        ll lmid=l+(r-l)/3,rmid=r-(r-l)/3;
        if(f(lmid)<=f(rmid)) l=lmid;
        else r=rmid;
    }
    return r;
}
```

```
 int l = 0, r = min(a/2, b/3);
        while (r-l>10)  // 这边调精度，将答案约束在这个范围内
        {
            int k = (r - l)/3;
            int x1 = l + k, x2 = r - k;
            if (f(x1) >= f(x2)) r = x2;
            else l = x1;
        }
        int ans = 0;
        for (int i = l; i <= r; i++) ans = max(ans, f(i));  // 最后在精度内枚举一遍就好啦
        cout << ans << endl;
```

多重集合排列组合问题

设多重集合 S = { n1 * a1, n2 * a2, ..., nk * ak },n = n1 + n2 + ... + nk,

即集合 S 中含有n1个元素a1， n2个元素a2，...，nk个元素ak，ni被称为元素ai的重数，k成为多重集合的类别数

在 S 中任选 r 个元素的排列称为S的r排列，当r = n时，有公式 P(n; n1*a1, n2*a2, ..., nk*ak) = n! / (n1! * n2! * ...* nk!)

在 S 中任选 r 个元素的组合称为S的r组合，当r<=任意ni时，有公式 C(n; n1*a1, n2*a2, ..., nk*ak) = C(k+r-1, r),

由公式可以看出多重集合的组合只与类别数k 和选取的元素r 有关，与总数无关！

数据结构

单链表

```
// head存储链表头，e[]存储节点的值，ne[]存储节点的next指针，idx表示当前用到了哪个节点
int head, e[N], ne[N], idx;
// 初始化
void init()
{
    head = -1;
    idx = 0;
}

// 将x插到头节点
void add_to_head(int x)
{
    e[idx] = x, ne[idx] = head, head = idx++;
}
// 将x插到下标是k的点后面
void add(int k, int x)
{
    e[idx] = x, ne[idx] = ne[k], ne[k] = idx++;
}
// 将下表是k的后面的点删掉
void remove(int k)
{
    ne[k] = ne[ne[k]];
}
// 将头节点删掉
void remove_head()
{
    head = ne[head];
}
```

双链表

```
// e[]表示节点的值，l[]表示节点的左指针，r[]表示节点的右指针，idx表示当前用到了哪个节点
int e[N], l[N], r[N], idx;
// 初始化
void init()
{
    // 0表示左端点，1表示右端点
    r[0] = 1, l[1] = 0;
    idx = 2;
}

// 在下标是k的点的右边，插入x
void add(int k, int x)
{
    e[idx] = x;
    r[idx] = r[k], l[idx] = k;
    l[r[idx]] = idx;
    r[k] = idx++;
}
```

```
// 删除第k个点
void remove(int k)
{
    l[r[k]] = l[k];
    r[l[k]] = r[k];
}
```

模拟栈

```
// tt表示栈顶
int stk[N], tt = 0;

// 向栈顶插入一个数
stk[ ++ tt] = x;

// 从栈顶弹出一个数
tt -- ;

// 栈顶的值
stk[tt];

// 判断栈是否为空
if (tt > 0)
{

}
```

模拟队列

```
// hh 表示队头，tt表示队尾
int q[N], hh = 0, tt = -1;

// 向队尾插入一个数
q[ ++ tt] = x;

// 从队头弹出一个数
hh ++ ;

// 队头的值
q[hh];

// 判断队列是否为空
if (hh <= tt)
{

}
```

单调栈

```
// 常见模型：找出每个数左边离它最近的比它大/小的数
int tt = 0;
for (int i = 1; i <= n; i ++ )
{
    while (tt && check(stk[tt], i)) tt -- ;
    stk[ ++ tt] = i;
}
```

Trie

Trie字符串统计

```
int son[N][26], cnt[N], idx;
char str[N];

void insert(char *str)
{
    int p = 0;
    for (int i = 0; str[i]; i ++ )
    {
        int u = str[i] - 'a';
        if (!son[p][u]) son[p][u] = ++ idx;
        p = son[p][u];
    }
    cnt[p] ++ ;
}

int query(char *str)
{
    int p = 0;
    for (int i = 0; str[i]; i ++ )
    {
        int u = str[i] - 'a';
        if (!son[p][u]) return 0;
        p = son[p][u];
    }
    return cnt[p];
}
```

堆

堆排序

```
const int N = 100010;

int n, m;
int h[N], cnt;

void down(int u)
{
    int t = u;
    if (u * 2 <= cnt && h[u * 2] < h[t]) t = u * 2;
```

```
    if (u * 2 + 1 <= cnt && h[u * 2 + 1] < h[t]) t = u * 2 + 1;
    if (u != t)
    {
        swap(h[u], h[t]);
        down(t);
    }
}

int main()
{
    scanf("%d%d", &n, &m);
    for (int i = 1; i <= n; i ++ ) scanf("%d", &h[i]);
    cnt = n;

    for (int i = n / 2; i; i -- ) down(i);

    while (m -- )
    {
        printf("%d ", h[1]);
        h[1] = h[cnt -- ];
        down(1);
    }

    puts("");

    return 0;
}
```

模拟堆

```
// h[N]存储堆中的值, h[1]是堆顶，x的左儿子是2x, 右儿子是2x + 1
// ph[k]存储第k个插入的点在堆中的位置
// hp[k]存储堆中下标是k的点是第几个插入的
int h[N], ph[N], hp[N], size;

// 交换两个点，及其映射关系
void heap_swap(int a, int b)
{
    swap(ph[hp[a]],ph[hp[b]]);
    swap(hp[a], hp[b]);
    swap(h[a], h[b]);
}

void down(int u)
{
    int t = u;
    if (u * 2 <= size && h[u * 2] < h[t]) t = u * 2;
    if (u * 2 + 1 <= size && h[u * 2 + 1] < h[t]) t = u * 2 + 1;
    if (u != t)
    {
        heap_swap(u, t);
        down(t);
    }
}

void up(int u)
{
```

```
    while (u / 2 && h[u] < h[u / 2])
    {
        heap_swap(u, u / 2);
        u >>= 1;
    }
}

// O(n)建堆
for (int i = n / 2; i; i -- ) down(i);
```

哈希

质数表

61, 83, 113, 151, 211, 281, 379, 509 683, 911 / 一千以下

1217, 1627, 2179, 2909, 3881, 6907, 9209, /一万以下

12281, 16381, 21841, 29123, 38833, 51787, 69061, 92083, /十万以下

122777, 163729, 218357, 291143, 388211, 517619, 690163, 999983, /百万以下

1226959, 1635947, 2181271, 2908361, 3877817, 5170427, 6893911, 9191891, /千万以下

12255871, 16341163, 21788233, 29050993, 38734667, 51646229,68861641, 91815541,/一亿以下

1e9+7 和 1e9+9 //十亿左右

122420729,163227661,217636919,290182597,386910137,515880193,687840301,917120411,/十亿以下

1222827239,1610612741, 3221225473ul, 4294967291ul

模拟散列表

```
(1) 拉链法
    int h[N], e[N], ne[N], idx;

    // 向哈希表中插入一个数
    void insert(int x)
    {
        int k = (x % N + N) % N;
        e[idx] = x;
        ne[idx] = h[k];
        h[k] = idx ++ ;
    }

    // 在哈希表中查询某个数是否存在
    bool find(int x)
    {
        int k = (x % N + N) % N;
        for (int i = h[k]; i != -1; i = ne[i])
            if (e[i] == x)
                return true;

        return false;
```

```
    }

(2) 开放寻址法
    int h[N];

    // 如果x在哈希表中，返回x的下标；如果x不在哈希表中，返回x应该插入的位置
    int find(int x)
    {
        int t = (x % N + N) % N;
        while (h[t] != null && h[  t] != x)
        {
            t ++ ;
            if (t == N) t = 0;
        }
        return t;
    }
```

字符串哈希

```
/*
核心思想：将字符串看成P进制数，P的经验值是131或13331，取这两个值的冲突概率低
小技巧：取模的数用2^64，这样直接用unsigned long long存储，溢出的结果就是取模的结果
*/

typedef unsigned long long ULL;
ULL h[N], p[N]; // h[k]存储字符串前k个字母的哈希值, p[k]存储 P^k mod 2^64

// 初始化
p[0] = 1;
for (int i = 1; i <= n; i ++ )
{
    h[i] = h[i - 1] * P + str[i];
    p[i] = p[i - 1] * P;
}

// 计算子串 str[l ~ r] 的哈希值
ULL get(int l, int r)
{
    return h[r] - h[l - 1] * p[r - l + 1];
}
```

可并堆

左偏树

```

```

树状数组

1. 快速求前缀和
2. 修改某一个数 3.注意树状数组从1开始，树状数组的范围根据具体情况而定
3. 如果是区间修改，则用差分来实现，区间（0，1）取反可以%2来实现

4. 对于值全是1的树状数组，tr[i] = lowbit(i);

树状数组求逆序对

```
#define lowbit(x) (x&-x)

int a[N];
int tr[N];
int n ;
void add(int x, int d)
{
    for (int i = x; i <= n; i += lowbit(i)) tr[i] += d;
}

int sum(int x)
{
    int res = 0;
    for (int i = x; i; i-= lowbit(i)) res += tr[i]; return res;
}

int Greater[N];
int main()
{

    while (~scanf("%d", &n) && n)
    {
        memset(tr, 0, sizeof tr);
        // 离散化
        vector<int> vect;
        for (int i = 1; i <= n; i++)
        {
            scai(a[i]);
            vect.push_back(a[i]);
        }
        sort(vect.begin(), vect.end());
        vect.erase(unique(vect.begin(), vect.end()), vect.end());
        for (int i = 1; i <= n; i++)
        {
            a[i] = lower_bound(vect.begin(), vect.end(), a[i]) - vect.begin() +
1;
        }
        for (int i = 1; i <= n; i++)
        {
            Greater[i] = sum(n) - sum(a[i]);
            add(a[i], 1);
        }
        ll res = 0;

        for (int i = 1; i <= n; i++)
        {
            res +=(ll) Greater[i];
        }
        printf("%lld\n", res);
    }
}
```

二维树状数组

```
int N;
int c[maxn][maxn];
inline int lowbit(int x)
{
    return x&(-x);
}
void update(int x,int y,int v)
{
    for (int i=x; i<=N; i+=lowbit(i))
        for (int j=y; j<=N; j+=lowbit(j))
            c[i][j]+=v;
}
int query(int x,int y)
{
    int s=0;
    for (int i=x; i>0; i-=lowbit(i))
        for (int j=y; j>0; j-=lowbit(j))
            s+=c[i][j];
    return s;
}
int sum(int x,int y,int xx,int yy)
{
    x--,y--;
    return query(xx,yy)-query(xx,y)-query(x,yy)+query(x,y);
}
```

三维树状数组

```
int N;
long long c[130][130][130]= {};
inline int lowbit(int t)
{
    return t&(-t);
}
void update(int x,int y,int z,long long v)
{
    for (int i=x; i<=N; i+=lowbit(i))
        for (int j=y; j<=N; j+=lowbit(j))
            for (int k=z; k<=N; k+=lowbit(k))
                c[i][j][k]+=v;
}
long long query(int x,int y,int z)
{
    long long s=0;
    for (int i=x; i>0; i-=lowbit(i))
        for (int j=y; j>0; j-=lowbit(j))
            for (int k=z; k>0; k-=lowbit(k))
                s+=c[i][j][k];
    return s;
}
long long sum(int x,int y,int z,int xx,int yy,int zz)
{
    x--,y--,z--;
```

```
    return query(xx,yy,zz)
    -query(x,yy,zz)-   query(xx,y,zz)-query(xx,yy,z)
    +query(x,y,zz)+query(xx,y,z)+query(x,yy,z)
    -query(x,y,z);
}
```

分块

例题poj3468

```
const int N = 1e5 + 10;

int a[N], sum[N], add[N];
int L[N], R[N]; // 每段左右端点
int pos[N];     //每个位置属于哪一段
int n, m, t;

void change(int l, int r, int d)
{
    int p = pos[l], q = pos[r];
    if (p == q)
    {
        rep(i, l, r) a[i] += d;
        sum[p] += d * (r - l + 1);
    }
    else
    {
        rep(i, p + 1, q - 1) add[i] += d;
        rep(i, l, R[p])
        {
            a[i] += d;
        }
        sum[p] += d * (R[p] - l + 1);
        rep(i, L[q], r)
        {
            a[i] += d;
        }
        sum[q] += d * (r - L[q] + 1);
    }
}

int ask(int l, int r)
{
    int p = pos[l], q = pos[r];
    int ans = 0;
    if (p == q)
    {
        rep(i, l, r) ans += a[i];
        ans += add[p] * (r - l + 1);
    }
    else
    {
        rep(i, p + 1, q - 1)
        {
            ans += sum[i] + add[i] * (R[i] - L[i] + 1);
        }
```

```
            rep(i, l, R[p]) ans += a[i];
            ans += add[p] * (R[p] - l + 1);
            rep(i, L[q], r) ans += a[i];
            ans += add[q] * (r - L[q] + 1);
    }
    return ans;
}
signed main()
{
    STDIN
    n = re, m = re;
    rep(i, 1, n) a[i] = re;

    // 分块
    t = sqrt(n);
    rep(i, 1, t)
    {
        L[i] = (i - 1) * sqrt(n) + 1;
        R[i] = i * sqrt(n);
    }
    if (R[t] < n)
        t++, L[t] = R[t - 1] + 1, R[t] = n;
    // 预处理
    rep(i, 1, t)
    {
        rep(j, L[i], R[i])
        {
            pos[j] = i;
            sum[i] += a[j];
        }
    }

    // 指令
    while (m--)
    {
        char op[3];
        int l, r, d;
        scanf("%s%lld%lld", op, &l, &r);
        if (op[0] == 'C')
        {
            scanf("%lld", &d);
            change(l, r, d);
        }
        else
            printf("%lld\n", ask(l, r));
    }
}
```

图论

树与图的存储

树是一种特殊的图，与图的存储方式相同。 对于无向图中的边ab，存储两条有向边a->b, b->a。 因此我们可以只考虑有向图的存储。

1. 邻接矩阵：g[a][b] 存储边a->b

2. 邻接表：

```
// 对于每个点k，开一个单链表，存储k所有可以走到的点。h[k]存储这个单链表的头结点
int h[N], e[N], ne[N], idx;

// 添加一条边a->b
void add(int a, int b)
{
    e[idx] = b, ne[idx] = h[a], h[a] = idx++;
}

// 初始化
idx = 0;
memset(h, -1, sizeof h);
```

树与图的遍历

时间复杂度 O(n+m)O(n+m), nn 表示点数，mm 表示边数

深度优先遍历

```
int dfs(int u)
{
    st[u] = true; // st[u] 表示点u已经被遍历过

    for (int i = h[u]; i != -1; i = ne[i])
    {
        int j = e[i];
        if (!st[j]) dfs(j);
    }
}
```

树的宽度优先遍历

```
queue<int> q;
st[1] = true; // 表示1号点已经被遍历过
q.push(1);

while (q.size())
{
    int t = q.front();
    q.pop();

    for (int i = h[t]; i != -1; i = ne[i])
    {
        int j = e[i];
        if (!s[j])
        {
            st[j] = true; // 表示点j已经被遍历过
            q.push(j);
        }
    }
}
```

宽度优先搜索打印路径

```
void bfs()
{
    memset(dist, 0x3f, sizeof dist);
    memset(hop, -1, sizeof hop);
    queue<int> q;
    q.push(st);
    dist[st] = 0;

    while (q.size())
    {

        auto t = q.front();
        q.pop();
        for (int i = 0; i < n; i++)
        {
            if (ti[t][i] + dist[t] < dist[i])
            {
                dist[i] = ti[t][i] + dist[t];
                hop[i] = t;
                q.push(i);
            }
        }
    }
}
```

树的重心

poj 1655

代码定义：树的重心也叫树的质心。对于一棵树n个节点的无根树，找到一个点，使得把树变成以该点为根的有根树时，最大子树的结点数最小。换句话说，删除这个点后最大连通块（一定是树）的结点数最小。

性质：

1. 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，如果有两个距离和，他们的距离和一样。
2. 把两棵树通过一条边相连，新的树的重心在原来两棵树重心的连线上。
3. 一棵树添加或者删除一个节点，树的重心最多只移动一条边的位置。
4. 一棵树最多有两个重心，且相邻。

```
const int N = 2e4 + 10;

int e[N<<1], ne[N<<1], h[N], idx;
int maxp[N], Size[N];
int rt; // 重心
int st[N];
int n;
void add(int a, int b)
{
    e[idx] = b;
    ne[idx] = h[a];
    h[a] = idx++;
}

void getrt(int u, int fa)
```

```
{
    Size[u] = 1; maxp[u] = 0;
    for (int i = h[u]; ~i; i = ne[i])
    {
        int j = e[i];
        if (j == fa || st[j]) continue; // 搜到父亲节点或者节点已经访问过，则跳过
        getrt(j, u);
        Size[u] += Size[j];
        maxp[u] = max(maxp[u], Size[j]);
    }
    maxp[u] = max(maxp[u], n - Size[u]);
    if (maxp[u] < maxp[rt]) rt = u;
}
signed main()
{
    STDIN
    case{
        rt = 0;
        memset(h, -1, sizeof h);
        idx = 0;
        n = re;
        rep(i, 1 , n - 1)
        {
            int a, b; a = re, b = re;
            add(a, b), add(b, a);
        }
        maxp[rt] = n;
        getrt(1, 0);
        cout << rt << " " << maxp[rt] << endl;
    }
}
```

树的直径

模板题

- https://ac.nowcoder.com/acm/contest/4462/B
- https://www.luogu.com.cn/problem/SP1437

方法

我使用的是比较常见的方法：两边dfs，第一遍从任意一个节点开始找出最远的节点x，第二遍从x开始做dfs找到最远节点的距离即为树的直径。

```
const int N = 3e5 + 10;
int e[N], h[N], ne[N],idx, v[N];
bool st[N];
int dis[N];
void add(int a, int b, int c)
{
    e[idx] = b;
    v[idx] = c;
    ne[idx] = h[a];
    h[a] = idx++;
}

void dfs(int u, int step)
```

```
{
    st[u] = true;
    dis[u] = step;
    for (int i = h[u]; i != -1; i = ne[i])
    {
        int j = e[i];
        if (!st[j])
        {
            dfs(j, step + v[i]);
        }
    }
}
int main()
{
    int n;
    cin >> n;
    memset(h, -1, sizeof h);
    idx = 0;
    for (int i = 1; i < n; i++)
        {
            int a, b;
            cin >> a >> b;
            add(a, b, 1), add(b, a, 1);
        }
    memset(st, false, sizeof st);
    dfs(1, 0);
    int maxn = -1;
    int j = -1;
    for (int i = 1; i <= n; i++)
    {
        if (dis[i] > maxn)
        {
            maxn = dis[i]; j = i;
        }
    }
    memset(st, false, sizeof st);
    dfs(j, 0);
    maxn = -1;
    j = -1;
    for (int i = 1; i <= n; i++)
    {
        if (dis[i] > maxn)
        {
            maxn = dis[i]; j = i;
        }
    }
    cout << maxn << endl;
}
```

动态规划法求

```
const int N = 10010, M = N * 2;

int n;
int h[N], e[M], w[M], ne[M], idx;
int ans;
```

```
void add(int a, int b, int c)
{
    e[idx] = b, w[idx] = c, ne[idx] = h[a], h[a] = idx ++ ;
}

int dfs(int u, int father)
{
    int dist = 0; // 表示从当前点往下走的最大长度
    int d1 = 0, d2 = 0;
    for (int i = h[u]; i != -1; i = ne[i])
    {
        int j = e[i];
        if (j == father) continue;
        int d = dfs(j, u) + w[i];
        dist = max(dist, d);

        if (d >= d1) d2 = d1, d1 = d;
        else if (d > d2) d2 = d;
    }
    ans = max(ans, d1 + d2);
    return dist;
}

int main()
{
    cin >> n;
    memset(h, -1, sizeof h);
    for (int i = 0; i < n - 1; i ++ )
    {
        int a, b, c;
        cin >> a >> b >> c;
        add(a, b, c), add(b, a, c);
    }
    dfs(1, -1);
    cout << ans << endl;
    return 0;
}
```

一些定理

- 树的所有直径拥有相同的中点

求单源最短路

友情提示:正权图请使用dijkstradijkstra算法,负权图请使用SPFASPFA算法

朴素dijkstra

$On^2 + m$

```
int g[N][N];  // 存储每条边
int dist[N];  // 存储1号点到每个点的最短距离
bool st[N];   // 存储每个点的最短路是否已经确定

// 求1号点到n号点的最短路，如果不存在则返回-1
```

```
int dijkstra()
{
    memset(dist, 0x3f, sizeof dist);
    dist[1] = 0;

    for (int i = 0; i < n - 1; i ++ )
    {
        int t = -1;     // 在还未确定最短路的点中，寻找距离最小的点
        for (int j = 1; j <= n; j ++ )
            if (!st[j] && (t == -1 || dist[t] > dist[j]))
                t = j;

        // 用t更新其他点的距离
        for (int j = 1; j <= n; j ++ )
            dist[j] = min(dist[j], dist[t] + g[t][j]);

        st[t] = true;
    }

    if (dist[n] == 0x3f3f3f3f) return -1;
    return dist[n];
}
```

堆优化dijkstra

```
typedef pair<int, int> PII;

int n;      // 点的数量
int h[N], w[N], e[N], ne[N], idx;       // 邻接表存储所有边
int dist[N];        // 存储所有点到1号点的距离
bool st[N];     // 存储每个点的最短距离是否已确定

// 求1号点到n号点的最短距离，如果不存在，则返回-1
int dijkstra()
{
    memset(dist, 0x3f, sizeof dist);
    dist[1] = 0;
    priority_queue<PII, vector<PII>, greater<PII>> heap;
    heap.push({0, 1});      // first存储距离，second存储节点编号

    while (heap.size())
    {
        auto t = heap.top();
        heap.pop();

        int ver = t.second, distance = t.first;

        if (st[ver]) continue;
        st[ver] = true;

        for (int i = h[ver]; i != -1; i = ne[i])
        {
            int j = e[i];
            if (dist[j] > distance + w[i])
            {
                dist[j] = distance + w[i];
                heap.push({dist[j], j});
```

```
            }
        }
    }

    if (dist[n] == 0x3f3f3f3f) return -1;
    return dist[n];
}
```

bellman_ford

一般用来求有边数限制的最短路 时间复杂度 O(nm)O(nm), nn 表示点数，mm 表示边数 注意在模板题中需要对下面的模板稍作修改，加上备份数组，详情见模板题。

```
int n, m;       // n表示点数，m表示边数
int dist[N];        // dist[x]存储1到x的最短路距离

struct Edge     // 边，a表示出点，b表示入点，w表示边的权重
{
    int a, b, w;
}edges[M];

// 求1到n的最短路距离，如果无法从1走到n，则返回-1。
int bellman_ford()
{
    memset(dist, 0x3f, sizeof dist);
    dist[1] = 0;

    // 如果第n次迭代仍然会松弛三角不等式，就说明存在一条长度是n+1的最短路径，由抽屉原理，路径中至少存在两个相同的点，说明图中存在负权回路。
    for (int i = 0; i < n; i ++ )
    {
        for (int j = 0; j < m; j ++ )
        {
            // 如果要避免发生串联，就备份数组。
            int a = edges[j].a, b = edges[j].b, w = edges[j].w;
            if (dist[b] > dist[a] + w)
                dist[b] = dist[a] + w;
        }
    }

    if (dist[n] > 0x3f3f3f3f / 2) return -1;
    return dist[n];
}
```

多源最短路问题

floyd

Floyd（弗洛伊德）算法是用来求解带权图（无论正负）中的多源最短路问题。算法的原理是动态规划。

```
// 初始化：
    for (int i = 1; i <= n; i ++ )
        for (int j = 1; j <= n; j ++ )
            if (i == j) d[i][j] = 0;
```

```
            else d[i][j] = INF;

// 算法结束后，d[a][b]表示a到b的最短距离
void floyd()
{
    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]);
}
```

求最小生成树

朴素prim算法

时间复杂度是 $O(n^2 + m)$, n 表示点数， m 表示边数

```
int n;      // n表示点数
int g[N][N];        // 邻接矩阵，存储所有边
int dist[N];        // 存储其他点到当前最小生成树的距离
bool st[N];     // 存储每个点是否已经在生成树中


// 如果图不连通，则返回INF(值是0x3f3f3f3f), 否则返回最小生成树的树边权重之和
int prim()
{
    memset(dist, 0x3f, sizeof dist);

    int res = 0;
    for (int i = 0; i < n; i ++ )
    {
        int t = -1;
        for (int j = 1; j <= n; j ++ )
            if (!st[j] && (t == -1 || dist[t] > dist[j]))
                t = j;

        if (i && dist[t] == INF) return INF;

        if (i) res += dist[t];
        st[t] = true;

        for (int j = 1; j <= n; j ++ ) dist[j] = min(dist[j], g[t][j]);
    }

    return res;
}
```

Kruskal算法

时间复杂度是 O(mlogm)O(mlogm), nn 表示点数，mm 表示边数

1. 若一个环能被二分，必定是奇数个点

```
int n;      // n表示点数
int h[N], e[M], ne[M], idx;     // 邻接表存储图
```

```
int color[N];       // 表示每个点的颜色，-1表示未染色，0表示白色，1表示黑色

// 参数：u表示当前节点，c表示当前点的颜色
bool dfs(int u, int c)
{
    color[u] = c;
    for (int i = h[u]; i != -1; i = ne[i])
    {
        int j = e[i];
        if (color[j] == -1)
        {
            if (!dfs(j, !c)) return false;
        }
        else if (color[j] == c) return false;
    }

    return true;
}

bool check()
{
    memset(color, -1, sizeof color);
    bool flag = true;
    for (int i = 1; i <= n; i ++ )
        if (color[i] == -1)
            if (!dfs(i, 0))
            {
                flag = false;
                break;
            }
    return flag;
}
```

二分图

染色法判别二分图

树一定能够二分染色

```
int n;      // n表示点数
int h[N], e[M], ne[M], idx;     // 邻接表存储图
int color[N];       // 表示每个点的颜色，-1表示未染色，0表示白色，1表示黑色

// 参数：u表示当前节点，c表示当前点的颜色
bool dfs(int u, int c)
{
    color[u] = c;
    for (int i = h[u]; i != -1; i = ne[i])
    {
        int j = e[i];
        if (color[j] == -1)
        {
            if (!dfs(j, !c)) return false;
        }
        else if (color[j] == c) return false;
    }
```

```
    return true;
}

bool check()
{
    memset(color, -1, sizeof color);
    bool flag = true;
    for (int i = 1; i <= n; i ++ )
        if (color[i] == -1)
            if (!dfs(i, 0))
            {
                flag = false;
                break;
            }
    return flag;
}
```

点分治

模板题 P3806

```
const int N = 2e4 + 10;
const int maxk = 2e7 + 10;
int e[N << 1], ne[N << 1], h[N], idx, w[N << 1];
int maxp[N], Size[N], dis[N], tmp[N], q[105];
bool vis[N], judge[maxk], ans[105];
int sum;
// judge[i]记录在之前子树中距离i是否存在
int rt; // 重心
int st[N];
int n, m;
int cnt; // 计数器
void add(int a, int b, int c)
{
    e[idx] = b;
    ne[idx] = h[a];
    w[idx] = c;
    h[a] = idx++;
}

void getrt(int u, int fa)
{
    Size[u] = 1;
    maxp[u] = 0;
    for (int i = h[u]; ~i; i = ne[i])
    {
        int j = e[i];
        if (j == fa || vis[j])
            continue; // 搜到父亲节点或者节点已经被删掉，则跳过
        getrt(j, u);
        Size[u] += Size[j];
        maxp[u] = max(maxp[u], Size[j]);
    }
    maxp[u] = max(maxp[u], sum - Size[u]);
```

```cpp
    if (maxp[u] < maxp[rt])
        rt = u;
}
void getdis(int u, int f)
{
    tmp[cnt++] = dis[u];
    for (int i = h[u]; ~i; i = ne[i])
    {
        int j = e[i];
        if (j == f || vis[j])
            continue;
        dis[j] = dis[u] + w[i];
        getdis(j, u);
    }
}
// 计算
// 计算经过根结点的路径
// solve 根据具体题目来写
void solve(int u)
{
    static std::queue<int> que;
    for (int i = h[u]; ~i; i = ne[i])
    {
        int j = e[i];
        if (vis[j])
            continue;
        cnt = 0; // 计数器置零
        dis[j] = w[i];
        getdis(j, u); // 把距离都处理出来
        for (int j = 0; j < cnt; j++)
            for (int k = 0; k < m; k++)
                if (q[k] >= tmp[j])
                    ans[k] |= judge[q[k] - tmp[j]];
        for (int j = 0; j < cnt; j++)
        {
            que.push(tmp[j]);
            judge[tmp[j]] = true;
        }
    }

    while (!que.empty())    // 清空judge数组， 不要用memset
    {
        judge[que.front()] = false;
        que.pop();
    }
}
// 分治
void divide(int u)
{
    vis[u] = judge[0] = true;
    solve(u);
    for (int i = h[u]; ~i; i = ne[i])
    {
        int j = e[i];
        if (vis[j])
            continue;
        maxp[rt = 0] = sum = Size[j]; // 把重心置为0， 并把maxp[0]置为最大值
        getrt(j, 0);
```

```
        getrt(rt, 0);
        divide(rt);
    }
}
signed main()
{
    STDIN
    rt = 0;
    memset(h, -1, sizeof h);
    idx = 0;
    n = re, m = re;
    rep(i, 1, n - 1)
    {
        int a, b, c;
        a = re, b = re, c = re;
        add(a, b, c), add(b, a, c);
    }
    rep(i, 0, m - 1) q[i] = re;
    maxp[rt] = sum = n;
    getrt(1, 0);
    getrt(rt, 0);
    divide(rt); // 分治

    for (int i = 0; i < m; i++)
    {
        if (ans[i]) puts("AYE");
        else puts("NAY");
    }
}
```

动态规划

背包问题

01背包

体积从大到小枚举

1. 最初版本

```
for (int i = 1; i <= N; i++)
    for (int j = 0; j <= V; j++)
    {
        dp[i][j] = dp[i-1][j];
        if (j > v[i])
        {
            dp[i][j] = max(dp[i][j], dp[i-1][j-v[i]] + w[i]);
        }
    }
```

2. 优化版本

```
 for (int i = 1; i <= N; i++)
     for (int j = V; j >= v[i]; j--)
         dp[j] = max(dp[j], dp[j-v[i]] + w[i]);
```

完全背包问题

1. 最初版本

```
for (int i = 1; i <= N; i++)
    for (int j = 0; j <= V; j++)
        for (int k = 0; k*v[i] <= j; k++)
        {
            dp[i][j] = max(dp[i][j], dp[i-1][j - k*v[i]] + k*w[i]);
        }
```

2. 优化版本

```
for (int i = 1; i <= N; i++)
    for (int j = v[i]; j <= V; j++)
        dp[j] = max(dp[j], dp[j - v[i]] + w[i]);
```

多重背包问题

1. 朴素版本

```
for (int i = 1; i <= N; i++)
    for (int j = 0; j <= V; j++)
    {
        for (int k = 0; k <= s[i] && k*v[i] <= j; k++)
        {
            dp[i][j] = max(dp[i][j], dp[i-1][j-k*v[i]] + k*w[i]);
        }
    }
```

2. 优化版本

```
int N, V; cin >> N >> V;
int a, b, s;
int cnt = 0;
for (int i = 1; i <= N; i++)
{
    cin >> a >> b >> s;
    int t = 1;
    while (t < s)
    {
        cnt++;
        v[cnt] = t*a; w[cnt] = t*b;
        s-=t;
        t <<= 1;
    }
    while (s)
    {
        cnt++;
        v[cnt] = s*a; w[cnt] = s*b;
```

```
    }
    for (int i = 1; i <= cnt; i++)
    {
        for (int j = V; j >= v[i]; j--)
        {
            dp[j] = max(dp[j], dp[j - v[i]]+w[i]);
        }
    }
}
```

线性dp

https://ac.nowcoder.com/acm/contest/3006/F *已解决*

动态规划求M字段和问题

【问题描述】----最大M子段和问题 给定由 n个整数（可能为负整数）组成的序列a1，a2，a3，……，an，以及一个正整数 m，要求确定序列 a1，a2，a3，……，an的 m个不相交子段， 使这m个子段的总和达到最大，求出最大和。

- 题目链接
- http://acm.hdu.edu.cn/showproblem.php?pid=1024

区间dp

1. dp[i, j]表示区间 i, j范围内的方案集合的某个状态值
2. 注意递推的方式要是前面的已经算过， 例如递推长度从小到大。

计数类dp

搜索

dfs 深度优先搜索

1. 画出搜索树加深理解
2. 递归终止条件
3. 还原现场（回溯）
4. 剪枝

bfs 广度优先搜索

1. 用队列来搜索
2. 搜索的状态是一个一个节点，若遇到复杂的状态，可将其转化为数字或者字符串

字符串

是否有想要的字串

```
// 令x_{i}x
令x表示字符串SS遍历到位置ii,与字符串X=匹配的最大长度。
for(int i=1;i<=n;i++)
    {
        if(s[i]==a[x]) x++;
    }
```

所有字串的个数

dp

```
for (int i = 0; i < s.length(); ++i)
    {
        dp[3] = (dp[3] + (s[i] == 'o')*dp[2]) %mod;
        dp[2] = (dp[2] + (s[i] == 'l')*dp[1]) %mod;
        dp[1] = (dp[1] + (s[i] == 'i')) %mod;
    }
```

其他

蔡勒公式

$w = (y + [\frac{y}{4}] + [\frac{c}{4}] - 2*c + [\frac{26*(m+1)}{10}] + d - 1)\%7$

or

$w = (y + [\frac{y}{4}] + [\frac{c}{4}] - 2*c + 2*m + [\frac{3*(m+1)}{5}] + d + 1)\%7$

公式中的符号含义如下:

- w：星期（计算所得的数值对应的星期：0-星期日；1-星期一；2-星期二；3-星期三；4-星期四；5-星期五；6-星期六
- c：年份前两位数
- y：年份后两位数
- m：月（m的取值范围为3至14，即在蔡勒公式中，某年的1、2月要看作上一年的13、14月来计算，比如2003年1月1日要看作2002年的13月1日来计算）
- d：日
- [ ]：称作高斯符号，代表向下取整，即，取不大于原数的最大整数。

若要计算的日期是在1582年10月4日或之前年代，公式则为:

$w = (y + [\frac{y}{4}] - c + [\frac{26*(m+1)}{10}] + d + 4)\%7$

c++STL

```
vector, 变长数组，倍增的思想
    size()  返回元素个数
    empty()  返回是否为空
    clear()  清空
    front()/back()
    push_back()/pop_back()
    begin()/end()
    []
    支持比较运算，按字典序

pair<int, int>
    first, 第一个元素
    second, 第二个元素
    支持比较运算，以first为第一关键字，以second为第二关键字（字典序）
```

```
string, 字符串
    size()/length()  返回字符串长度
    empty()
    clear()
    substr(起始下标, (子串长度))  返回子串
    c_str()  返回字符串所在字符数组的起始地址


    string &insert(int pos, const char *s);
    string &insert(int pos, const string &s);
    //前两个函数在pos位置插入字符串s
    string &insert(int pos, int n, char c);  //在pos位置 插入n个字符c

    string &erase(int pos=0, int n=npos);  //删除pos开始的n个字符, 返回修改后的字符串

queue, 队列
    size()
    empty()
    push()  向队尾插入一个元素
    front()  返回队头元素
    back()  返回队尾元素
    pop()  弹出队头元素

priority_queue, 优先队列, 默认是大根堆
    push()  插入一个元素
    top()  返回堆顶元素
    pop()  弹出堆顶元素
    定义成小根堆的方式: priority_queue<int, vector<int>, greater<int>> q;

    struct Time{
        int start, end;
        bool operator < (const Time& t)const{
            return start > t.start;
        }
    };

stack, 栈
    size()
    empty()
    push()  向栈顶插入一个元素
    top()  返回栈顶元素
    pop()  弹出栈顶元素

deque, 双端队列
    size()
    empty()
    clear()
    front()/back()
    push_back()/pop_back()
    push_front()/pop_front()
    begin()/end()
    []

set, map, multiset, multimap, 基于平衡二叉树(红黑树), 动态维护有序序列
    size()
    empty()
    clear()
    begin()/end()
```

```
    ++, -- 返回前驱和后继，时间复杂度 O(logn)

    set/multiset
        insert()  插入一个数
        find()  查找一个数
        count()  返回某一个数的个数
        erase()
            (1) 输入是一个数x，删除所有x   O(k + logn)
            (2) 输入一个迭代器，删除这个迭代器
        lower_bound()/upper_bound()
            lower_bound(x)  返回大于等于x的最小的数的迭代器
            upper_bound(x)  返回大于x的最小的数的迭代器
    map/multimap
        insert()  插入的数是一个pair
        erase()  输入的参数是pair或者迭代器
        find()
        []  注意multimap不支持此操作。 时间复杂度是 O(logn)
        lower_bound()/upper_bound()

unordered_set, unordered_map, unordered_multiset, unordered_multimap, 哈希表
    和上面类似，增删改查的时间复杂度是 O(1)
    不支持 lower_bound()/upper_bound()， 迭代器的++，--

### bitset, 压位
    bitset<10000> s;
    ~, &, |, ^
    >>, <<
    ==, !=
    []

    count()  返回有多少个1

    any()  判断是否至少有一个1
    none()  判断是否全为0

    set()  把所有位置成1
    set(k, v)  将第k位变成v
    reset()  把所有位变成0
    flip()  等价于~
    flip(k) 把第k位取反
```

```c++
    // 构造
    bitset<4> bitset1;    //无参构造，长度为４，默认每一位为０

    bitset<8> bitset2(12);    //长度为８，二进制保存，前面用０补充

    string s = "100101";
    bitset<10> bitset3(s);    //长度为10，前面用０补充

    char s2[] = "10101";
    bitset<13> bitset4(s2);    //长度为13，前面用０补充

    cout << bitset1 << endl;    //0000
    cout << bitset2 << endl;    //00001100
    cout << bitset3 << endl;    //0000100101
    cout << bitset4 << endl;    //0000000010101

    // 可用操作符
```

```
    bitset<4> foo (string("1001"));
    bitset<4> bar (string("0011"));

    cout << (foo^=bar) << endl;       // 1010 (foo对bar按位异或后赋值给foo)
    cout << (foo&=bar) << endl;       // 0010 (按位与后赋值给foo)
    cout << (foo|=bar) << endl;       // 0011 (按位或后赋值给foo)

    cout << (foo<<=2) << endl;        // 1100 (左移 2 位，低位补 0，有自身赋值)
    cout << (foo>>=1) << endl;        // 0110 (右移 1 位，高位补 0，有自身赋值)

    cout << (~bar) << endl;           // 1100 (按位取反)
    cout << (bar<<1) << endl;         // 0110 (左移，不赋值)
    cout << (bar>>1) << endl;         // 0001 (右移，不赋值)

    cout << (foo==bar) << endl;       // false (0110==0011为false)
    cout << (foo!=bar) << endl;       // true  (0110!=0011为true)

    cout << (foo&bar) << endl;        // 0010 (按位与，不赋值)
    cout << (foo|bar) << endl;        // 0111 (按位或，不赋值)
    cout << (foo^bar) << endl;        // 0101 (按位异或，不赋值)

    bitset<4> foo ("1011");
    cout << foo[0] << endl;    //1
    cout << foo[1] << endl;    //1
    cout << foo[2] << endl;    //0

    // 函数
    bitset<8> foo ("10011011");

    cout << foo.count() << endl;    //5    （count函数用来求bitset中1的位数，foo中共有 5 个 1
    cout << foo.size() << endl;     //8    （size函数用来求bitset的大小，一共有 8 位

    cout << foo.test(0) << endl;    //true    （test函数用来查下标处的元素是 0 还是 1，并返回false或true，此处foo[0]为 1，返回true
    cout << foo.test(2) << endl;    //false    （同理，foo[2]为 0，返回false

    cout << foo.any() << endl;    //true    （any函数检查bitset中是否有 1
    cout << foo.none() << endl;    //false    （none函数检查bitset中是否没有 1
    cout << foo.all() << endl;    //false    （all函数检查bitset中是全部为 1

    bitset<8> foo ("10011011");

    cout << foo.flip(2) << endl;    //10011111    （flip函数传参数时，用于将参数位取反，本行代码将foo下标 2 处"反转"，即 0 变 1， 1 变 0
    cout << foo.flip() << endl;     //01100000    （flip函数不指定参数时，将bitset每一位全部取反

    cout << foo.set() << endl;        //11111111    （set函数不指定参数时，将bitset的每一位全部置为 1
    cout << foo.set(3,0) << endl;    //11110111    （set函数指定两位参数时，将第一参数位的元素置为第二参数的值，本行对foo的操作相当于foo[3]=0
    cout << foo.set(3) << endl;      //11111111    （set函数只有一个参数时，将参数下标处置为 1

    cout << foo.reset(4) << endl;    //11101111    （reset函数传一个参数时将参数下标处置为 0
```

```
    cout << foo.reset() << endl;     //00000000    （reset函数不传参数时将bitset的每一位全部置为 0

    // 类型转换
    bitset<8> foo ("10011011");

    string s = foo.to_string();    //将bitset转换成string类型
    unsigned long a = foo.to_ulong();    //将bitset转换成unsigned long类型
    unsigned long long b = foo.to_ullong();    //将bitset转换成unsigned long long类型

    cout << s << endl;    //10011011
    cout << a << endl;    //155
    cout << b << endl;    //155
```

list lista{1,2,3} lista(n) //声明一个n个元素的列表，每个元素都是0 lista(n, m) //声明一个n个元素的列表，每个元素都是m lista(first, last) //声明一个列表，其元素的初始值来源于由区间所指定的序列中的元素，first和last是迭代器 push_back()和push_front() front()和back() 在编写程序时，最好先调用empty()函数判断list是否为空，再调用front()和back()函数。 使用pop_back()可以删掉尾部第一个元素，pop_front()可以删掉头部第一个元素。注意：list必须不为空，如果当list为空的时候调用pop_back()和pop_front()会使程序崩掉。

```
a.insert(a.begin(),100); //在a的开始位置（即头部）插入100
a.insert(a.begin(),2, 100); //在a的开始位置插入2个100
a.insert(a.begin(),b.begin(), b.end());//在a的开始位置插入b从开始到结束的所有位置的元素

a.erase(a.begin()); //将a的第一个元素删除
a.erase(a.begin(),a.end()); //将a的从begin()到end()之间的元素删除。

list<int>a{6,7,8,9,7,10};
a.remove(7);
```

```
## 计算几何



### 凸包
#### Andrw_algorithm
```c++
struct vec
{
    double x, y;
    bool operator < (const vec & W)const
    {
        if (x != W.x)
            return x < W.x;
        return y < W.y;
    }
}p[N];
double Cross(vec A, vec B)
{
    return A.x*B.y-A.y*B.x;
}
double Side(vec a, vec b, vec p)
{
    vec A = {b.x - a.x, b.y-a.y}; //  向量ab;
    vec B = {p.x-a.x, p.y-a.y}; //向量ap;
```
```

```
    return Cross(A, B);
}

double DistancePow(vec a, vec b)
{
    return (a.x-b.x)*(a.x-b.x)+(a.y-b.y)*(a.y-b.y);
}
int n;
vec st[N];
int Andrew(int top) // 返回栈顶
{
    sort(p+1, p+n+1);
    if (n<3)
    {
        printf("-1\n");return -1;
    }
    st[1] = p[1], st[2] = p[2];
    top = 2;
    for (int i = 3; i <= n; i++) // 从p1 开始的下凸包
    {
        while(top>=2&&Side(st[top-1], st[top],p[i]) <= 0) top--;
        st[++top] = p[i];
    }
    st[++top] = p[n-1];
    for (int i = n-2; i >= 1; i--) // 从pn 开始的上凸包 注意这边i一定要到1，到2是错的
    {
        while (top >= 2 &&Side(st[top-1], st[top],p[i]) <= 0) top--;
        st[++top] = p[i];
    }

    return top;
}
```

```
/*/////////////////////////////////////
***************Content***************
范数（模的平方）
向量的模(求线段距离/两点距离)
点乘
叉乘
判断向量垂直
判断a1-a2与b1-b2垂直
判断线段垂直
判断向量平行
判断a1-a2与b1-b2平行
判断线段平行
求点在线段上的垂足
求点关于直线的对称点
判断P0,P1,P2三点位置关系,Vector p0-p2 在p0-p1的相对位置
判断p1-p2与p3-p4是否相交
判断线段是否相交
a,b两点间的距离
点到直线的距离
点到线段的距离
两线段间的距离(相交为0)
求两线段的交点
求直线交点
中垂线
```

```
向量a,b的夹角，范围[0,180]
向量（点）极角，范围[-180,180]
角度排序(从x正半轴起逆时针一圈)范围为[0,180)
判断点在多边形内部
凸包(CCW/CW)
求向量A,向量B构成三角形的面积
(旋转卡壳)求平面中任意两点的最大距离
三点所成的外接圆
三点所成的内切圆
过点p求过该点的两条切线与圆的两个切点
极角(直线与x轴的角度)
点与圆的位置关系
已知点与切线求圆心
已知两直线和半径,求夹在两直线间的圆
求与给定两圆相切的圆
多边形(存储方式:点->线)
半平面交
求最近点对的距离(分治)
*////////////////////////////////////
#include<cstdio>
#include<algorithm>
#include<cmath>
#include<cstring>
#include<vector>
using namespace std;
#define EPS (1e-10)
#define equals(a, b) (fabs(a-b)<EPS)
const double PI = acos(-1);
const double INF = 1e20;

static const int CCW=1;//逆时针
static const int CW=-1;//顺时针
static const int BACK=-2;//后面
static const int FRONT=2;//前面
static const int ON=0;//线段上

struct Point
{
    double x, y;
    Point(double x=0, double y=0):x(x), y(y) {}
};//点
typedef Point Vector;//向量

struct Segment
{
    Point p1, p2;
    Segment(Point p1=Point(), Point p2=Point()):p1(p1), p2(p2) {}
    double angle;
};//线段
typedef Segment Line;//直线
typedef vector<Point> Polygon;//多边形(存点)
typedef vector&lt;Segment&gt; Pol;//多边形（存边）

class Circle
{
public:
    Point c;
    double r;
```

```
    Circle(Point c=Point(), double r=0.0):c(c), r(r) {}
    Point point(double a)
    {
        return Point(c.x + cos(a)*r, c.y + sin(a)*r);
    }
};//圆

Point operator + (Point a, Point b)
{
    return Point(a.x+b.x, a.y+b.y);
}
Point operator - (Point a, Point b)
{
    return Point(a.x-b.x, a.y-b.y);
}
Point operator * (Point a, double p)
{
    return  Point(a.x*p, a.y*p);
}
Point operator / (Point a, double p)
{
    return Point(a.x/p, a.y/p);
}
//排序左下到右上
bool operator < (const Point &a,const Point &b)
{
    return a.x<b.x||(a.x==b.x&&a.y<b.y);
}
bool operator == (Point a, Point b)
{
    return fabs(a.x-b.x)<EPS && fabs(a.y-b.y)<EPS;
}
//范数（模的平方）
double norm(Vector a)
{
    return a.x*a.x+a.y*a.y;
}
//向量的模(求线段距离/两点距离)
double abs(Vector a)
{
    return sqrt(norm(a));
}
//点乘
double dot(Vector a, Vector b)
{
    return a.x*b.x+a.y*b.y;
}
//叉乘
double cross(Vector a, Vector b)
{
    return a.x*b.y-a.y*b.x;
}
//判断向量垂直
bool isOrthgonal(Vector a, Vector b)
{
    return equals(dot(a, b), 0.0);
}
//判断a1-a2与b1-b2垂直
```

```
bool isOrthgonal(Point a1, Point a2, Point b1, Point b2)
{
    return isOrthgonal(a1-a2, b1-b2);
}
//判断线段垂直
bool isOrthgonal(Segment s1, Segment s2)
{
    return equals(dot(s1.p2-s1.p1, s2.p2-s2.p1), 0.0);
}
//判断向量平行
bool isParallel(Vector a, Vector b)
{
    return equals(cross(a, b), 0.0);
}
//判断a1-a2与b1-b2平行
bool isParallel(Point a1, Point a2, Point b1, Point b2)
{
    return isParallel(a1-a2, b1-b2);
}
//判断线段平行
bool isParallel(Segment s1, Segment s2)
{
    return equals(cross(s1.p2-s1.p1, s2.p2-s2.p1), 0.0);
}
//求点在线段上的垂足
Point project(Segment s, Point p)
{
    Vector base=s.p2-s.p1;
    double r=dot(p-s.p1, base)/norm(base);
    return s.p1+base*r;
}
//求点关于直线的对称点
Point reflect(Segment s, Point p)
{
    return p+(project(s, p)-p)*2.0;
}
//判断P0,P1,P2三点位置关系,Vector p0-p2 在p0-p1的相对位置
int ccw(Point p0, Point p1, Point p2)
{
    Vector a=p1-p0;
    Vector b=p2-p0;
    if( cross(a, b)>EPS ) return CCW;
    if( cross(a, b)<-EPS ) return CW;
    if( dot(a, b)<-EPS ) return BACK;
    if( norm(a)<norm(b) ) return FRONT;
    return ON;
}
//判断p1-p2与p3-p4是否相交
bool intersect(Point p1, Point p2, Point p3, Point p4)
{
    return (ccw(p1, p2, p3)*ccw(p1, p2, p4)<=0 &&ccw(p3, p4, p1)*ccw(p3, p4, p2)
<=0);
}
//判断线段是否相交
bool intersect(Segment s1, Segment s2)
{
    return intersect(s1.p1, s1.p2, s2.p1, s2.p2);
}
```

```
//a,b两点间的距离
double getDistance(Point a, Point b)
{
    return abs(a-b);
}
//点到直线的距离
double getDistanceLP(Line l, Point p)
{
    return abs(cross(l.p2-l.p1, p-l.p1)/abs(l.p2-l.p1));
}
//点到线段的距离
double getDistanceSP(Segment s, Point p)
{
    if(dot(s.p2-s.p1, p-s.p1)<0.0) return abs(p-s.p1);
    if(dot(s.p1-s.p2, p-s.p2)<0.0) return abs(p-s.p2);
    return getDistanceLP(s, p);
}
//两线段间的距离(相交为0)
double getDistanceSS(Segment s1, Segment s2)
{
    if(intersect(s1, s2)) return 0.0;
    return min( min(getDistanceSP(s1,s2.p1), getDistanceSP(s1,s2.p2)),
                min(getDistanceSP(s2,s1.p1), getDistanceSP(s2,s1.p2)) );
}
//求两线段的交点
Point getCrossPoint(Segment s1, Segment s2)
{
    Vector base=s2.p2-s2.p1;
    double d1=abs(cross(base, s1.p1-s2.p1));
    double d2=abs(cross(base, s1.p2-s2.p1));
    double t=d1/(d1+d2);
    return s1.p1+(s1.p2-s1.p1)*t;
}
//求直线交点
Point intersectL(Segment a, Segment b)
{
    double x1=a.p1.x,y1=a.p1.y,x2=a.p2.x,y2=a.p2.y;
    double x3=b.p1.x,y3=b.p1.y,x4=b.p2.x,y4=b.p2.y;
    double k1=(x4-x3)*(y2-y1),k2=(x2-x1)*(y4-y3);
    double ans_x=(k1*x1-k2*x3+(y3-y1)*(x2-x1)*(x4-x3))/(k1-k2);
    double ans_y=(k2*y1-k1*y3+(x3-x1)*(y2-y1)*(y4-y3))/(k2-k1);
    return Point(ans_x,ans_y);
}
//中垂线
Line mid_vert(Line l)
{
    double x1=l.p1.x,y1=l.p1.y;
    double x2=l.p2.x,y2=l.p2.y;
    double xm=(x1+x2)/2,ym=(y1+y2)/2;
    Line s;
    s.p1.x=xm+ym-y1;
    s.p1.y=ym-xm+x1;
    s.p2.x=xm-ym+y1;
    s.p2.y=ym+xm-x1;
    return s;
}
//向量a,b的夹角，范围[0,180]
double Angle(Vector a, Vector b)
```

```
{
    return acos(dot(a, b)/(abs(a)*abs(b)));
}
//向量（点）极角，范围[-180,180]
double angle(Vector v)
{
    return atan2(v.y, v.x);
}
//角度排序(从x正半轴起逆时针一圈)范围为[0,180)
double SortAngle(Vector a)
{
    Point p0(0.0, 0.0);
    Point p1(a.x, a.y);
    Point p2(-1.0, 0.0);
    Vector b=p2;
    if(ccw(p0, p1, p2)==CW) return acos(dot(a, b)/(abs(a)*abs(b)));
    if(ccw(p0, p1, p2)==CCW) return 2*PI-acos(dot(a, b)/(abs(a)*abs(b)));
    if(ccw(p0, p1, p2)==BACK) return PI;
    else return 0;
}
/*
判断点在多边形内部
IN 2
ON 1
OUT 0
*/
int contains(Polygon g, Point p)
{
    int n=g.size();
    bool x=false;
    for(int i=0; i<n; i++)
    {
        Point a=g[i]-p;
        Point b=g[(i+1)%n]-p;
        if(abs(cross(a, b))<EPS && dot(a, b)<EPS) return 1;
        if(a.y>b.y) swap(a,b);
        if(a.y<EPS && EPS<b.y && cross(a,b)>EPS) x=!x;
    }
    return (x? 2 : 0);
}
//凸包(CCW/CW)
Polygon andrewScan(Polygon s)
{
    Polygon u, l;
    if(s.size()<3) return s;
    sort(s.begin(), s.end());
    u.push_back(s[0]);
    u.push_back(s[1]);
    l.push_back(s[s.size()-1]);
    l.push_back(s[s.size()-2]);
    for(int i=2; i<s.size(); i++)
    {
        for(int n=u.size(); n>=2 && ccw(u[n-2], u[n-1], s[i])!=CW; n--)
            u.pop_back();
        u.push_back(s[i]);
    }
    for(int i=s.size()-3; i>=0; i--)
    {
```

```
        for(int n=l.size(); n>=2 && ccw(l[n-2], l[n-1], s[i])!=CW; n--)
            l.pop_back();
        l.push_back(s[i]);
    }
    reverse(l.begin(), l.end());
    for(int i=u.size()-2; i>=1; i--) l.push_back(u[i]);
    return l;
}
//求向量A,向量B构成三角形的面积
double TriArea(Vector a, Vector b)
{
    return 0.5*abs(cross(a,b));
}
//求平面中任意两点的最大距离(旋转卡壳)
double RotatingCalipers(const Polygon& s)
{
    Polygon l;
    double dis, maxn=0.0;
    int len, i, k;
    l=andrewScan(s);
    len=l.size();
    if(len>=3)
    {
        for(i=0, k=2; i<len; i++)
        {
            while(cross(l[(k+1)%len]-l[i], l[(k+1)%len]-
l[(i+1)%len])>=cross(l[k%len]-l[i], l[k%len]-l[(i+1)%len]))
                k++;
            dis=max(norm(l[k%len]-l[i]), norm(l[k%len]-l[(i+1)%len]));
            if(dis>maxn) maxn=dis;
        }
    }
    else maxn=norm(l[1]-l[0]);
    return maxn;
}
//三点所成的外接圆
Circle CircumscribedCircle(Point a, Point b, Point c)
{
    double x=0.5*(norm(b)*c.y+norm(c)*a.y+norm(a)*b.y-norm(b)*a.y-norm(c)*b.y-
norm(a)*c.y)
             /(b.x*c.y+c.x*a.y+a.x*b.y-b.x*a.y-c.x*b.y-a.x*c.y);
    double y=0.5*(norm(b)*a.x+norm(c)*b.x+norm(a)*c.x-norm(b)*c.x-norm(c)*a.x-
norm(a)*b.x)
             /(b.x*c.y+c.x*a.y+a.x*b.y-b.x*a.y-c.x*b.y-a.x*c.y);
    Point O(x, y);
    double r=abs(O-a);
    Circle m(O, r);
    return m;
}
//三点所成的内切圆
Circle InscribedCircle(Point a, Point b, Point c)
{
    double A=abs(b-c), B=abs(a-c), C=abs(a-b);
    double x=(A*a.x+B*b.x+C*c.x)/(A+B+C);
    double y=(A*a.y+B*b.y+C*c.y)/(A+B+C);
    Point O(x, y);
    Line l(a, b);
    double r=getDistanceLP(l, O);
```

```
    Circle m(o, r);
    return m;
}
//过点p求过该点的两条切线与圆的两个切点
Segment TangentLineThroughPoint(Circle m, Point p)
{
    Point c=m.c;
    double l=abs(c-p);
    double r=m.r;
    double k=(2*r*r-l*l+norm(p)-norm(c)-2*p.y*c.y+2*c.y*c.y)/(2*(p.y-c.y));
    double A=1+(p.x-c.x)*(p.x-c.x)/((p.y-c.y)*(p.y-c.y));
    double B=-(2*k*(p.x-c.x)/(p.y-c.y)+2*c.x);
    double C=c.x*c.x+k*k-r*r;
    double x1, x2, y1, y2;

    x1=(-B-sqrt(B*B-4*A*C))/(2*A);
    x2=(-B+sqrt(B*B-4*A*C))/(2*A);
    y1=(2*r*r-l*l+norm(p)-norm(c)-2*(p.x-c.x)*x1)/(2*(p.y-c.y));
    y2=(2*r*r-l*l+norm(p)-norm(c)-2*(p.x-c.x)*x2)/(2*(p.y-c.y));
    Point p1(x1, y1), p2(x2, y2);
    Segment L(p1, p2);
    return L;
}
//极角(直线与x轴的角度)
double PolarAngle(Vector a)
{
    Point p0(0.0, 0.0);
    Point p1(1.0, 0.0);
    Point p2(a.x, a.y);
    Vector b=p1;
    double ans=0;
    if(ccw(p0, p1, p2)==CW) ans=180-acos(dot(a,
b)/(abs(a)*abs(b)))*180/acos(-1);
    else if(ccw(p0, p1, p2)==CCW) ans=acos(dot(a,
b)/(abs(a)*abs(b)))*180/acos(-1);
    else ans=0;
    if(ans>=180) ans-=180;
    if(ans<0) ans+=180;
    return ans;
}
//点与圆的位置关系
int CircleContain(Circle m, Point p)
{
    double r=m.r;
    double l=abs(p-m.c);
    if(r>l) return 2;
    if(r==l) return 1;
    if(r<l) return 0;
}
//已知点与切线求圆心
void CircleThroughAPointAndTangentToALineWithRadius(Point p, Line l, double r)
{
    Point m=project(l, p);
    if(abs(p-m)>2*r)
    {
        printf("[]\n");
    }
    else if(abs(p-m)==2*r)
```

```
    {
        Circle c((p+m)/2, r);
        printf("[(%.6f,%.6f)]\n", c.c.x, c.c.y);
    }
    else if(abs(p-m)<EPS)
    {
        Point m0(m.x+10, m.y);
        if(abs(m0-project(l, m0))<EPS) m0.y+=20;
        Point m1=project(l, m0);
        Circle c1(m-(m0-m1)/abs(m0-m1)*r, r);
        Circle c2(m+(m0-m1)/abs(m0-m1)*r, r);
        if(c1.c.x>c2.c.x) swap(c1, c2);
        else if(c1.c.x==c2.c.x && c1.c.y>c2.c.y) swap(c1, c2);
        printf("[(%.6f,%.6f),(%.6f,%.6f)]\n", c1.c.x, c1.c.y, c2.c.x, c2.c.y);
    }
    else if(abs(p-m)<2*r)
    {
        double s=abs(p-m);
        double d=sqrt(r*r-(r-s)*(r-s));
        Point m1, m2;
        m1=(m+(l.p1-l.p2)/abs(l.p1-l.p2)*d);
        m2=(m-(l.p1-l.p2)/abs(l.p1-l.p2)*d);
        Circle c1(m1+(p-m)/abs(p-m)*r, r);
        Circle c2(m2+(p-m)/abs(p-m)*r, r);
        if(c1.c.x>c2.c.x) swap(c1, c2);
        else if(c1.c.x==c2.c.x && c1.c.y>c2.c.y) swap(c1, c2);
        printf("[(%.6f,%.6f),(%.6f,%.6f)]\n", c1.c.x, c1.c.y, c2.c.x, c2.c.y);
    }
    return ;
}

bool cmp_CircleTangentToTwoLinesWithRadius(Circle x, Circle y)
{
    if(x.c.x==y.c.x) return x.c.y<y.c.y;
    else return x.c.x<y.c.x;
}
//已知两直线和半径,求夹在两直线间的圆
void CircleTangentToTwoLinesWithRadius(Line l1, Line l2, double r)
{
    Point p=intersectL(l1, l2);
    Vector a, b;
    l1.p2=p+p-l1.p1;
    l2.p2=p+p-l2.p1;

    a=(l1.p1-p)/abs(l1.p1-p), b=(l2.p2-p)/abs(l2.p2-p);
    Circle c1(p+(a+b)/abs(a+b)*(r/sin(Angle(a,a+b))),r);

    a=(l1.p1-p)/abs(l1.p1-p), b=(l2.p1-p)/abs(l2.p1-p);
    Circle c2(p+(a+b)/abs(a+b)*r/sin(Angle(a,a+b)),r);

    a=(l1.p2-p)/abs(l1.p2-p), b=(l2.p1-p)/abs(l2.p1-p);
    Circle c3(p+(a+b)/abs(a+b)*(r/sin(Angle(a,a+b))),r);

    a=(l1.p2-p)/abs(l1.p2-p), b=(l2.p2-p)/abs(l2.p2-p);
    Circle c4(p+(a+b)/abs(a+b)*(r/sin(Angle(a,a+b))),r);

    vector<Circle> T;
    T.push_back(c1);
```

```
    T.push_back(c2);
    T.push_back(c3);
    T.push_back(c4);
    sort(T.begin(), T.end(), cmp_CircleTangentToTwoLinesWithRadius);
    printf("[(%.6f,%.6f),(%.6f,%.6f),(%.6f,%.6f),(%.6f,%.6f)]\n", T[0].c.x,
T[0].c.y, T[1].c.x, T[1].c.y, T[2].c.x, T[2].c.y, T[3].c.x, T[3].c.y);
}
//求与给定两圆相切的圆
void CircleTangentToTwoDisjointCirclesWithRadius(Circle C1, Circle C2)
{
    double d = abs(C1.c-C2.c);
    if(d<EPS) printf("[]\n");
    else if(fabs(C1.r+C2.r)<d) printf("[]\n");
    else if(fabs(C1.r-C2.r)>d) printf("[]\n");
    else
    {
        double sita = angle(C2.c - C1.c);
        double da = acos((C1.r*C1.r + d*d - C2.r*C2.r) / (2 * C1.r*d));
        Point p1 = C1.point(sita - da), p2 = C1.point(sita + da);
        if(p1.x>p2.x) swap(p1, p2);
        else if(p1.x==p2.x && p1.y>p2.y) swap(p1, p2);
        if (p1 == p2) printf("[(%.6f,%.6f)]\n", p1.x, p1.y);
        else printf("[(%.6f,%.6f),(%.6f,%.6f)]\n", p1.x, p1.y, p2.x, p2.y);
    }
}
//多边形(存储方式:点->线)
Pol Polygon_to_Pol(Polygon L)
{
    Pol S;
    Segment l;
    for(int i=0; i+1<L.size(); i++)
    {
        l.p1=L[i];
        l.p2=L[i+1];
        swap(l.p1, l.p2);//注意顺序
        l.angle=angle(l.p2-l.p1);
        S.push_back(l);
    }
    l.p1=L[L.size()-1];
    l.p2=L[0];
    swap(l.p1, l.p2);//注意顺序
    l.angle=angle(l.p2-l.p1);
    S.push_back(l);
    return S;
}
//半平面交
bool SortAnglecmp(Segment a, Segment b)
{
    if(fabs(a.angle-b.angle)>EPS)
        return a.angle>b.angle;
    return ccw(b.p1, b.p2, a.p1)!=CW;
}

int intersection_of_half_planes(Pol s)
{
    Segment deq[1505];
    Segment l[1505];
    Point p[1505];
```

```
    memset(deq, 0, sizeof(deq));
    memset(l, 0, sizeof(l));
    memset(p, 0, sizeof(p));

    sort(s.begin(), s.end(), SortAnglecmp);
    int cnt=0;
    for(int i=0; i<s.size(); i++)
        if(fabs(s[i].angle-l[cnt].angle)>EPS)
            l[++cnt]=s[i];
    int le=1,ri=1;
    for(int i=1; i<=cnt; i++)
    {
        while(ri>le+1 && ccw(l[i].p1, l[i].p2, intersectL(deq[ri-1],deq[ri-
2]))==CW) ri--;
        while(ri>le+1 && ccw(l[i].p1, l[i].p2,
intersectL(deq[le],deq[le+1]))==CW) le++;
        deq[ri++]=l[i];
    }
    while(ri>le+2 && ccw(deq[le].p1, deq[le].p2, intersectL(deq[ri-1],deq[ri-
2]))==CW) ri--;
    while(ri>le+2 && ccw(deq[ri-1].p1, deq[ri-1].p2,
intersectL(deq[le],deq[le+1]))==CW) le++;
    //***************getArea*****************
    /*
    if(ri<=le+2)
    {
        printf("0.00\n");
        return 0;
    }
    deq[ri]=deq[le];
    cnt=0;
    for(int i=le; i<ri; i++)
        p[++cnt]=intersectL(deq[i],deq[i+1]);
    double ans=0.0;
    for(int i=2; i<cnt; i++)
        ans+=fabs(TriArea(p[i]-p[1],p[i+1]-p[1]));
    printf("%.2f\n", ans);
    return 0;
    */
    //****************************************
    if(ri>le+2) return 1;
    return 0;
}
//求最近点对的距离(分治)
//**********************************************************************
bool cmpxy(Point a, Point b)
{
    if(a.x != b.x) return a.x < b.x;
    else return a.y < b.y;
}
bool cmpy(Point a, Point b)
{
    return a.y < b.y;
}
int n;
Point closest_p[100010];//将点都存入closest_p中!
Point closest_tmpt[100010];
double Closest_Pair(int left,int right)
```

```
{
    double d = INF;
    if(left == right) return d;
    if(left + 1 == right)
        return getDistance(closest_p[left],closest_p[right]);
    int mid = (left+right)/2;
    double d1 = Closest_Pair(left,mid);
    double d2 = Closest_Pair(mid+1,right);
    d = min(d1,d2);
    int k = 0;
    for(int i = left; i <= right; i++)
        if(fabs(closest_p[mid].x - closest_p[i].x) <= d)
            closest_tmpt[k++] = closest_p[i];
    sort(closest_tmpt,closest_tmpt+k,cmpy);
    for(int i = 0; i <k; i++)
        for(int j = i+1; j < k && closest_tmpt[j].y - closest_tmpt[i].y < d;
j++)
            d = min(d,getDistance(closest_tmpt[i],closest_tmpt[j]));
    return d;
}
double Closest()//直接调用此函数即可
{
    sort(closest_p,closest_p+n,cmpxy);
    return Closest_Pair(0,n-1)/2;
}
//***************************************************************************
int main()
{
    while(scanf("%d",&n)==1 && n)
    {
        for(int i = 0; i < n; i++)
            scanf("%lf%lf",&closest_p[i].x,&closest_p[i].y);
        printf("%.2lf\n",Closest());
    }
    return 0;
}
```

其他算法

粒子群优化算法

```
const int cnt=100;
int n;
double xs[15];//系数
double l,r;//x的范围
double f(double x) {//计算函数值
    double y = 0;
    for (int i=n+1; i>=1; i--) {
        y+=xs[n-i+2]*pow(x,i-1);
    }
    return y;
}
double Rand() {
    return (double)rand()/RAND_MAX;//返回一个[0,1]的随机实数
}
```

```
struct node {
    double xv,x,y,besty,bestx;
} b[105];
//xv是速度向量，x是位置，y是当前位置的函数值，besty是该粒子历史最优值，bestx是该粒子历史最优值时的x的值

double by=-1e233,bx;
//by是全局当前最优值，bbx是取到全局最优值时的自变量x

void update(int a) {
    //更新速度向量
    //速度向量   惯性                  全局最优  局部最优   当前位置
    b[a].xv=b[a].xv*0.5+Rand()*2*(bx+b[a].bestx-b[a].x*2);//更新公式

    //通过速度向量更新位置
    b[a].x+=b[a].xv;

    //位置出界处理            速度向量方向反转
    if (b[a].x<l) b[a].x=l,b[a].xv=b[a].xv*-1;
    if (b[a].x>r) b[a].x=r,b[a].xv=b[a].xv*-1;

    b[a].y=f(b[a].x);     //计算当前位置函数值
    if (b[a].y>b[a].besty) { //更新局部最优解
        b[a].bestx=b[a].x;
        b[a].besty=b[a].y;
    }
}

int main() {
    scanf("%d%lf%lf",&n,&l,&r);
    for (int i=1; i<=n+1; i++) {
        scanf("%lf",&xs[i]);//读入系数
    }
    srand(xs[1]+xs[n]);
    //生成粒子
    for (int i=1; i<=cnt; i++) {
        //xv是速度向量，x是位置，y是当前位置的函数值，besty是该粒子历史最优值，bestx是该粒子历史最优值时的x的值
        b[i].x=b[i].bestx=l+Rand()*(r-l);//初始x的值 为 l~r 的一个实数
        b[i].xv=0;      //速度向量初始化为0
        b[i].y=b[i].besty=f(b[i].x);    //计算当前函数值
        if (by<b[i].y) { //若当前函数值优于全局最优函数值则更新全局最优
            bx=b[i].bestx;
            by=b[i].besty;
        }
    }
    //开始迭代
    for (int k=1; k<=100; k++) {
        for (int i=1; i<=cnt; i++) {
            //对每个粒子速度和位置更新
            update(i);
            if (by<b[i].besty) {
                //更新全局最优解
                bx=b[i].bestx;
                by=b[i].besty;
            }
        }
    }
```

```
    printf("%.5lf\n",bx);//全局最优的x的值即为答案
    return 0;
}
```

线段树模板

区间修改加等差数列

```
#include <iostream>
#include <cstdio>
#include <cstring>
#include <string>
#include <vector>
#include <cmath>
#include <algorithm>
using namespace std;
 #define STDIN freopen("in.txt","r",stdin);
freopen("out.txt","w",stdout);//************************************

#define ll long long
const int maxn = 500500;
inline int read(){
    char ch=getchar();int x=0,f=0;
    while(ch<'0' || ch>'9') f|=ch=='-',ch=getchar();
    while(ch>='0' && ch<='9') x=x*10+ch-'0',ch=getchar();
    return f?-x:x;
}
//int primes[9] = { 0,3 ,5 ,7 ,11 ,13, 17 ,19 ,23 };

const ll mod = 111546435;
const ll INV2 = mod-mod/2;
ll a[maxn], b[maxn];
int n, m;

ll val[maxn];
ll sum[maxn];

struct Node
{
    int l, r;
    ll ax, bx; // 等差数列的首项和公差
    ll sum;
}N[maxn << 2];

inline int cal(int l,int r,int val,int d)
    {
        int cnt=r-l+1;
        int ret=(1LL*val+(1LL*val+1LL*(cnt-1)*d%mod)%mod)%mod;
        ret=1LL*ret*cnt%mod*INV2%mod;
        return ret;
    }
void pushDown(int i){
    ll av = N[i].ax% mod, bv = N[i].bx% mod;
    if (N[i].ax != 0 || N[i].bx != 0){
        int mid = N[i].r + N[i].l >> 1;
        N[i << 1].ax = (N[i << 1].ax + av)% mod;
```

```
        N[i << 1].bx = (N[i << 1].bx + bv)% mod;
        // int len = N[i << 1].r - N[i << 1].l + 1;
        N[i << 1 | 1].ax = (N[i << 1 | 1].ax + (av + bv*(mid-N[i].l+1)%mod) %
mod)% mod;
        N[i << 1 | 1].bx =( N[i << 1 | 1].bx + bv)% mod;
        N[i << 1].sum = (N[i << 1].sum + cal(N[i].l,mid,av, bv))% mod;
        N[i << 1 | 1].sum = (N[i << 1 | 1].sum + cal(mid+1,N[i].r, (av + bv*
(mid-N[i].l+1)%mod) % mod, bv))% mod;
        N[i].ax = N[i].bx = 0;
    }
}

void pushUp(int i){
    N[i].sum = (N[i << 1].sum + N[i << 1 | 1].sum)% mod;
}

void build(int i, int L, int R){
    if (L == R){
        N[i] = {L, R, 0,0,a[L]};
    }
    else
    {
        N[i] = {L,R,0,0,0};
        int M = (L + R) >> 1;
        build(i << 1, L, M);
        build(i << 1 | 1, M + 1, R);
        pushUp(i);
    }
}

void update(int i, int L, int R, ll val, ll d){
    if (N[i].l >= L&&N[i].r <= R){
        N[i].ax = (N[i].ax + val) % mod;
        N[i].bx = (N[i].bx + d) % mod;
        int len = R - L + 1;
        N[i].sum = (N[i].sum +cal(L,R, val,d))%mod;
        return;
    }
    pushDown(i);
    int M = (N[i].l + N[i].r) >> 1;
    if (R <= M){
        update(i << 1, L, R, val, d);
    }
    else if (L > M){
        update(i << 1 | 1, L, R, val, d);
    }
    else{
        int len = (M - L + 1);
        update(i << 1, L, M, val, d);
        update(i << 1 | 1, M + 1, R, (val + (M-L+1)*d %mod) % mod, d);
    }
    pushUp(i);
}

ll query(int i, int L, int R){
    if (N[i].l >= L&&N[i].r <= R){
        return N[i].sum% mod;
    }
```

```
    pushDown(i);
    int M = (N[i].l + N[i].r) >> 1;
    ll ret = 0ll;
    if (L <= M) ret = (query(i<<1, L, R) + ret) % mod;
    if (R > M) ret = (query(i<<1|1, L, R) + ret) % mod;
    return ret;
}

int main()
{
   // STDIN
    n = read();
    for (int i = 1; i <= n; i++) a[i] =  read();
    build(1, 1, n);
    // exit(0);
    int oper, l, r;
    m = read();
    int val, d;
    int tt;
    for (int i = 0; i < m; i++){
        // scanf("%d%d%d", &oper, &l, &r);
        oper = read();
        l = read(); r = read();
        if (oper == 1)
        {

            val = read(); d = read();
            // scanf("%d %d", &val, &d);
            update(1, l, r, val, d);
        }
        else {
            ll res = query(1, l, r);
            tt = read();
            printf("%lld\n", res%tt);
        }
    }
    return 0;
}
```

线段树经典例题

洛谷 P4145 https://www.luogu.com.cn/problem/P4145 1e12的数开方6次就变成了1，所以需要修改的次数实际上很少 可以看成是单点修改

```
ll n,m;
const int N = 1e5 +10;
ll a[N];
struct node
{
    int l, r;
    ll v;  // 区间最大值
    ll tsum; // 区间内数的和;
}tr[N<<2];

void pushup(node &u, node &l, node &r)
{
```

```
    u.v = max(l.v, r.v);
    u.tsum = l.tsum + r.tsum;
}
void pushup(int u)
{
    pushup(tr[u], tr[u<<1], tr[u<<1|1]);
}
void build(int u, int l, int r)
{
    tr[u] = {l, r};
    if (l == r)
    {
        tr[u] = {l, r, a[l], a[l]};
        return ;
    }
    int mid = l + r >> 1;
    build(u<<1, l, mid), build(u<<1|1, mid + 1, r);
    pushup(u); return ;
}

ll query(int u, int l, int r)
{
    if (tr[u].l >= l && tr[u].r <= r) return tr[u].tsum;
    int mid = tr[u].l + tr[u].r >> 1;
    ll v = 0;
    if (l <= mid) v = query(u << 1, l, r);
    if (r > mid) v = v + query(u<<1|1, l, r);
    return v;
}
void modify(int u, int l, int r)
{
    if (tr[u].v == 1ll) return;
    else if (tr[u].l == tr[u].r)
    {
        tr[u].v = (ll)sqrt(tr[u].v);
        tr[u].tsum = tr[u].v;
    }
    else
    {
        int mid = tr[u].l + tr[u].r >> 1;
        if (l <= mid) modify(u<<1, l, r);
        if (r > mid) modify(u<<1|1, l, r);
        pushup(u);
    }
    return ;
}
int main()
{
    STDIN
    n = read();
    for (int i = 1; i <= n; i++)
    {
        a[i] = read();
    }
    build(1, 1, n);
    int m; cin >> m;
    int op,l,r;
    while (m--)
```

```
    {
        scanf("%d%d%d", &op, &l, &r);
        if (l > r) swap(l, r);
        if (op == 1)
        {
            printf("%lld\n", query(1, l, r));
        }
        else
        {
            modify(1, l, r);
        }
    }
    return 0;
}
```