目录

1 动态规划

1.1 背包问题

1.1.1 0-1 背包

n 个物体，每个物体的费用为 $cost_i$, 价值为 val_i. 最大允许费用为 max_cost, 问能挑选出来的最大价值是什么。

```
for (int i = 1; i <= max_cost; i++)
  for (int l = max_cost - i; l >= 0; l--)
    f[l + i] = max(f[l] + cost[i], f[l + i]);
```

2 拓展欧几里得算法

欧几里得算法直接使用 g++ 中的 <algorithm> 库中 __gcd() 函数即可。

$(a,b)=(b,a \bmod b)$.

拓展欧几里得算法用于求出不定方程 $ax+by=(a,b)$ 的一个特解 x_0,y_0, 顺带求出 (a,b), **通解** $x=x_0+\frac{b}{(a,b)}t,\ y=y_0-\frac{a}{(a,b)}t\ (t\in Z)$.

2.1 解不定方程

不定方程 $ax+by=c$ 有解等价于 $(a,b)\mid c$. 据此判断是否有解，若有解，假设有一组特解 $x_0^{'},y_0^{'}$, 则它们的 $\frac{c}{(a,b)}$ 倍显然是原不定方程的一组特解。$x_0=\frac{c}{(a,b)}x_0^{'},y_0=\frac{c}{(a,b)}y_0^{'}$, 而通解依旧是 $x=x_0+\frac{b}{(a,b)}t,\ y=y_0-\frac{a}{(a,b)}t\ (t\in Z)$.

2.2 求解模线性方程（线性同余方程）

$ax\equiv c\pmod m \iff ax+my=c$.

2.3 求乘法逆元

$ab\equiv 1\pmod m$, 则 a 关于模 m 的乘法逆元是 b，b 关于模 m 的乘法逆元是 a。或者说 Z_m 群中 a 和 b 互为乘法逆元。

用乘法逆元有 $\frac{A}{b}\equiv A\times b^{-1}\pmod c$. 当左边的式子 A 是很大的数，而 b 是小规模数，且除出来的数一定是整数的时候，可以用右式边算边模。

求解 $ax\equiv 1\pmod m \iff ax+my=1$. 解出的 x 即为解，只是注意需要用通解公式将 x 调整到 Z_m 范围内。

2.4 线性同余方程组

方程组 $a_ix\equiv c_i\pmod{m_i}\quad(i=1,2,3,\ldots,n)$

对每一个现行同余方程，若无解，则方程组无解；否则，可以解得 $x=k_it_i+x_i,\ t_i\in Z$. 而 $x_i\in[0,k_i)\,,k_i=\frac{m_i}{(a_i,m_i)}\le m_i$. 不妨增加 $x_0=0,k_0=1$, 即增加式子 $x=t_0+0$.

现在考虑同时满足 $x=k_1t_1+x_1$ 与 $x=k_2t_2+x_2$ 两个约束的 x 能否合并成一个依旧如此形式的一个表达式，即 $x=k_0t+x_0$.

联立两个方程，易得 $k_1t_1-k_2t_2=x_2-x_1$, 将其视作关于 t_1,t_2 的不定方程。若这个方程无解，说明同时满足两个条件的 x 不存在；否则，每确定一个 t_1 可以代入 $x=k_1t_1+x_1$ 确定一个 x. 如果我们只是解出 t_1 的话，不妨换成 $k_1t_1+k_2t_2=x_2-x_1$，t_1 的每一个解是不变的。

假设 $k_1t_1+k_2t_2=x_2-x_1$ 有解，并且解为 $t_1=t_{1_0}+\frac{k_2}{(k_1,k_2)}t\quad t_{1_0}\in\left[0,\frac{k_2}{(k_1,k_2)}\right),\ t\in Z$.

代入 $x=k_1t_1+x_1$ 得 $x=k_1(t_{1_0}+\frac{k_2}{(k_1,k_2)}t)+x_1=\frac{k_1k_2}{(k_1,k_2)}t+(k_1t_{1_0}+x_1)$. 而 $k_1t_{1_0}+x_1<\frac{k_1k_2}{(k_1,k_2)}\iff t_{1_0}+\frac{x_1}{k_1}<\frac{k_2}{(k_1,k_2)}$. 而 $t_{1_0}<\frac{k_2}{(k_1,k_2)}$. 注意到 $t_{1_0},\frac{k_2}{(k_1,k_2)}$ 是整数，故 $t_{1_0}\le\frac{k_2}{(k_1,k_2)}-1$. 又

$\frac{x_1}{k_1} < 1$. 这两个不等式相加即可得到

$$t_{1_0} + \frac{x_1}{k_1} < \frac{k_2}{(k_1, k_2)}$$

。即符合前面定义的形式：

$$(k_1 t_{1_0} + x_1) \in \left[0, \frac{k_1 k_2}{(k_1, k_2)}\right)$$

.

合并为 $x = kt + x_0$. 的形式有：

$$k = \frac{k_1, k_2}{(k_1, k_2)} = [k_1, k_2]$$

$$x_0 = k_1 t_{1_0} + x_1$$

对于写程序，由于我们引入了 $x = x_0 = 0, k = k_0 = 1$. 可以每次 x_0, k_0 与 x_i, k_i 合并成 x_0, k_0. 故程序迭代是 $x+ = kt, k = [k, k_i]$, 其中 t_0 是 $k_0 t_0 + k_i t_i = x_i - x_0$ 的不定方程的最小非负整数解。

程序设计方面爆 long long 的问题及应对策略

由于 k 的迭代是不断求最小公倍数，而特解 x 始终是小于 k 的，因此 k 和 x 可能会增长的很快导致爆 long long. 尤其是 k 很容易爆掉。

如何解决爆炸的问题？或许可以用 __int128。如果 k 和 x 还是都爆掉了，那么没法子，只能设法自己实现 k 和 x 的存储，注意到解不定方程需要求 (k_0, k_i) 与 $\frac{k_i}{(k_0, k_i)}$, 所以要大数加减，大数乘除模普通数的实现。估计够呛。

2.5 code

复杂度，拓展欧几里得算法复杂度 $\ln val$, 不定方程、线性同余方程、逆元都是一次拓欧，其余部分是 $O(1)$. 线性同余方程组每一个方程需要解 2 个不定方程，其余操作单个都是 $O(1)$, 复杂度 $O(n \ln val)$.

```
// poj 2891 然而poj不支持__int128和C++11
#include <bits/stdc++.h>
typedef __int128 ll;

// 求解不定方程ax+by=(a,b)的一组特解并返回a,b最大公约数
// x,y存储返回的一组特解。易懂version
ll ex_gcd1(ll a, ll b, ll &x, ll &y) {
  if (b) {
    auto d = ex_gcd1(b, a%b, x,  y);
    auto x_bac = x;
    x = y; // x设为后
    y = x_bac - a/b * y; // y设为前-a/b*后
    return d;
  } else {
    x = 1; y = 0;
```

```
    return a;
  }
}

// 求解不定方程ax+by=(a,b)的一组特解并返回a,b最大公约数
// x,y存储返回的一组特解。
ll ex_gcd(ll a, ll b, ll &x, ll &y) {
  if (b) {
    auto d = ex_gcd(b, a%b, y, x); // 注意x和y位置互换了。
    // x是后， 无需赋值， y是 前-a/b*后 即 y -= a/b*x
    y -= a/b*x;
    return d;
  } else {
    x = 1; y = 0;
    return a;
  }
}

// 求解不定方程ax+by=c.
// 返回值表示是否有解
// d存储是(a,b)
// 当有解的情况下
// x,y存储一组特解,并且确保x是最小的非负整数。
// 通解是X=x+(b/d)*t,Y=y-(a/d)*t t是整数。
bool binary_linear_indefinite_equation(ll a, ll b, ll c, ll &x, ll &y, ll &d) {
  d = ex_gcd(a, b, x, y); // solve: ax+by=(a,b)
  if (c%d) return false;
  x *= c/d;
  y *= c/d;
  auto k = b/d;
  x = (x%k+k)%k; // 调为最小非负整数
  y = (d-a*x)/b;
  return true;
}

// 线性同余方程 Linear congruence equation
// ax = c (mod m) <===> ax+my=c
// x存储最小非负解， 通解X=x+kt t为整数
// 有解的情况下， 最小非负解x肯定在[0,m)范围内
bool linear_congruence_equation(ll a, ll c, ll m, ll &x, ll &k) {
  ll y, d;
  auto ans = binary_linear_indefinite_equation(a, m, c, x, y, d);
  k = m/d;
  return ans;
}
```

```
// 求a在Zm<+,*>中的乘法逆元x
// 返回逆元是否存在,x存储逆元
// ax = 1 (mod m)
bool multiplicative_inverse(ll a, ll m, ll &x) {
  ll k;
  return linear_congruence_equation(a, 1, m, x, k);
  // assert(k == m);
}

// 线性同余方程组 Linear congruence equations
// a_ix = c_i (mod m_i) 共n个
// 可能存在的问题，由于迭代过程中k一直在求最小公倍数，所以可能会爆long long，这
    个，最佳的方法是直接暴力把ll的定义改为__int128
// 但是要注意__int128的输入输出
// 如果还是爆，我没法子了
bool linear_congruence_equations(int n, ll a[], ll c[], ll m[], ll &x, ll &k) {
  ll x_i, k_i, t, t_i, d;
  x = 0; k = 1;
  for (int i = 0; i < n; ++i) {
    if (!linear_congruence_equation(a[i], c[i], m[i], x_i, k_i))
      return false;
    // kt+x
    // k_it_i+x_i
    if (!binary_linear_indefinite_equation(
      k, k_i, x_i-x, t, t_i, d))
      return false;
    x += k*t;
    k *= k_i/d;
  }
  return true;
}

inline ll read()
{
  ll x = 0;
  bool f = 0;
  char ch = getchar();
  while (ch < '0' || '9' < ch)
    f |= ch == '-', ch = getchar();
  while ('0' <= ch && ch <= '9')
    x = x * 10 + ch - '0', ch = getchar();
  return f ? -x : x;
}

void write(ll a)
{
```

```cpp
  if (a < 0)
  {
    putchar('-');
    a = -a;
  }
  if (a >= 10)
  {
    write(a / 10);
  }
  putchar(a % 10 + '0');
}

const int kMaxN = 10000;
ll a[kMaxN]; // ax=c (mod c)
ll m[kMaxN];
ll c[kMaxN];
void solve(int n) {
  for (int i = 0; i < n; ++i) {
    a[i] = 1;
    m[i] = read();
    c[i] = read();
  }
  ll x,k;
  auto ans = linear_congruence_equations(n,a,c,m,x,k);
  if (ans) {
    write(x);
    putchar('\n');
  } else
    puts("-1");
}

int main()
{
  int n;
  while (scanf("%d",&n) != EOF) {
    solve(n);
  }
}
```

3 快速乘幂及矩阵快速幂

3.1 快速模乘与快速模幂

时间复杂度：快速乘、普通快速幂 $O(\log_2 n)$，使用快速乘的快速幂 $O(\log_2 n \times \log_2 max\_val) = O(\log_2 n \times \log_2 mod)$

```
struct mod_sys{
    typedef long long ll;
    ll mod;
    // mod_sys类初始化设置模数
    inline void set_mod(ll mod0) {mod = mod0;}
    // 返回a在[0,mod)内标准等价的数，即数学意义上的a%mod
    inline ll to_std(ll a) {return (a%mod+mod)%mod;}
    // 计算数学意义上的a*n%mod
    ll mlt(ll a, ll n) {
        a = to_std(a); n = to_std(n);
        if (0 == a || 0 == n) return 0;
        // 始终维持要求的数可以表示为n(a)+t
        ll t = 0;
        while (n > 1) {
            if (n&1) t = (t+a)%mod;
            n >>= 1; a = (a<<1)%mod;
        }
        return (a+t)%mod; // now n = 1
    }
    // 计算数学意义上的a^n%mod 输入应当a,n>=0
    ll pow(ll a, ll n)
    {
        if (n == 0) return 1%mod;
        a = to_std(a);
        // 始终维持要求的数可以表示为(a)^n*t
        ll t = 1;
        while (n > 1)
        {
            if (n&1) t = t*a%mod;
            n >>= 1; a = a*a%mod;
        }
        return a*t%mod; // now n = 1
    }
    // 计算数学意义上的a^n%mod 输入应当a,n>=0
    // 此版本使用quick_mlt防止相乘爆ll
    ll pow_v2(ll a, ll n)
    {
        if (n == 0) return 1%mod;
        a = to_std(a);
```

```
        // 始终维持要求的数可以表示为(a)^n*t
        ll t = 1;
        while (n > 1)
        {
            if (n&1) t = mlt(t,a);
            n >>= 1; a = mlt(a,a);
        }
        return mlt(t,a); // now n = 1
    }
};
```

3.2 矩阵快速幂

矩阵乘法时间复杂度:$n \times m$ 与 $m \times r$ 的矩阵相乘，复杂度 $O(nmr)$。计算 A^n. 矩阵乘法的次数 $O(\log_2 n)$, 总复杂度 $|A|^3 \log_2 n$.

```
// 除非是设置单位矩阵, 否则必须调用set_size进行设置大小并清零 (或指定值)的初始化
// 所有函数都预设传入了正确的参数
// 矩阵乘法使用取模版, 加减数乘未取模(默认它们不爆) 因为一般题目也没这些操作
struct mtr{
    int r_sz, c_sz;
    typedef ll item_type;
    typedef vector<item_type> row_type;
    vector<row_type> data;
    mtr():r_sz(0),c_sz(0),data(){}
    // 设置大小, 并且全部元素设置为item_val值
    void set_size(int r_size, int c_size, int item_val = 0) {
        r_sz = r_size; c_sz = c_size;
        data.resize(r_sz);
        for (auto &row : data)
            row.resize(c_sz, item_val);
    }
    inline bool is_square() { return r_sz == c_sz; }
    // inline row_type& operator()(int r) { return data[r]; }
    // inline item_type& operator()(int r,int c) { return data[r][c];}

    // 会自动调用set_size,调用之前请勿调用set_size
    // 设置成n阶单位矩阵
    void set_identity(int n) {
        set_size(n, n, 0);
        for (int i = 0; i < n; ++i)
            data[i][i] = 1;
    }
    void in() {
        for (int i = 0; i < r_sz; ++i)
```

```
            for (int j = 0; j < c_sz; ++j)
                scanf("%lld", &data[i][j]);
    }
    // 矩阵输出，主要为了调试
    void out() {
        for (auto &row : data) {
            for (auto &cell : row)
                cout<<cell<<" ";
            cout<<"\n";
        }
    }
    // 矩阵加，假设传参合法
    mtr operator+(const mtr& obj) const {
        mtr ans;
        ans.set_size(r_sz, c_sz);
        for (int i = 0; i < r_sz; ++i)
            for (int j = 0; j < c_sz; ++j)
                ans.data[i][j] = data[i][j] + obj.data[i][j];
        return ans;
    }
    mtr operator-(const mtr& obj) const {/*只要把加法改成减法即可*/}
    // 矩阵数乘 数在右边
    // 数乘 数在左边必须在类外边用函数实现，模板不提供，容易改出来
    mtr operator*(item_type obj) const {/*只要把加法改成*obj即可*/}
    // 所有元素对mod取模(数学意义)
    void get_mod(ll mod) {/*只要把加法改成(data[i][j]%mod+mod)%mod再%mod即可*/}
    // 矩阵乘法 不用运算符乘号进行重载，便于增加mod参数修改成取模版
    // 默认元素乘法不爆long long，否则需要引入mod_sys模板
    // 默认待两个输入矩阵已经get_mod规约过了。
    mtr mlt(const mtr& obj, ll mod) const {
        mtr ans;
        ans.set_size(r_sz, obj.c_sz);
        for (int i = 0; i < r_sz; ++i)
            for (int j = 0; j < obj.c_sz; ++j) {
                item_type t = 0;
                for (int k = 0; k < c_sz; ++k)
                    t = (t+(data[i][k]*obj.data[k][j])%mod)%mod;
                ans.data[i][j] = t;
            }
        return ans;
    }
    // 预设n>=0
    mtr pow(ll n, ll mod) const {
        mtr a = *this;
        mtr t;
        t.set_identity(r_sz);
```

```
        // (a)^n*t
        if (n == 0) return t;
        while (n>1) {
            if (n&1) t = a.mlt(t, mod);
            n >>= 1; a = a.mlt(a, mod);
        }
        return a.mlt(t, mod);
    }
};
```

4 Miller-Rabin 素数检测算法

其基于以下两个定理。

1. Fermat 小定理若 n 是素数，则 $\forall a (a \not\equiv 0 \pmod{n})$，有 $a^{n-1} \equiv 1 \pmod{n}$.
2. 二次探测定理若 n 是素数，则 $x^2 \equiv 1 \pmod{n}$ 只有平凡根 $x = \pm 1$，即 $x = 1, x = n - 1$.

4.1 具体算法

假设 n 是奇数，令 $n = m \times 2^q (q \geq 1)$，其中 m 是奇数. 对于序列 $a^m \bmod n, a^{2m} \bmod n, a^{4m} \bmod n, \ldots, a^{2^q \times m} \bmod n$. 最后一项就是费马小定理中的 a^{n-1}，并且每一项都是前一项的平方。我们一项一项往后计算。

- 若当前项为 1，后面每一项显然都是 1。而根据二次探测定理，n 是素数必须前面一项是 1 或 n-1. 如果不符合，断言不是素数；符合，断言是素数。
- 若当前项不是 1，暂时不断言，接着往后算。除非当前是最后一项了，那么断言不是素数。

当然，如果第一项是 1，由于不存在二次探测的方程，所以不检验前面一项（或者认为前面一项符合条件）。

4.2 Code

使用了快速幂模和快速幂加模板 mod_sys。下面代码只是 miller-rabin 核心代码。

```
// 如果只是int范围内，可以将pow_v2改为pow，mlt改为普通乘法
bool miller_rabin(ll a, ll n, ll q, ll m, mod_sys& mod) {
    a = mod.pow_v2(a, m);
    bool is_ordinary = true;
    for (int i = 0; i < q; ++i) {
        if (a == 1) {
            return is_ordinary;
        } else {
```

```
            is_ordinary = (a == n-1);
            a = mod.mlt(a,a);
        }
    }
    return (a==1)&&(is_ordinary); // 最后一项
}

// 使用miller_rabin检测是否是素数
const int kCheckCnt = 8;
// 为了随机数
random_device rd;
mt19937_64 gen(rd());
bool miller_rabin(ll n) {
    if (n == 2) return true;
    if ((n <= 2) || (n&1^1)) return false;
    // 2^q×m表示原本输入的n-1
    ll m = n, q = 0;
    do { m >>= 1; ++q; } while(m&1^1);
    // 随机数生成, [1,n-1] 均匀分布
    uniform_int_distribution<> dis(1, n-1);
    mod_sys mod;
    mod.set_mod(n);
    for (int i = 0; i < kCheckCnt; ++i)
        if (!miller_rabin(dis(gen), n, q, m, mod))
            return false;
    return true;
}
```

5 lucas 定理

$$\binom{n}{m} \bmod p = \binom{\lfloor \frac{n}{p} \rfloor}{\lfloor \frac{m}{p} \rfloor}\binom{n \bmod p}{m \bmod p} \bmod p = \binom{n/p}{m/p}\binom{n\%p}{m\%p} \bmod p$$

先预先求出 $i!$ $(i \in [0,p))$. 并利用费马小定理和快速幂乘求出每一个 $i!$ 的逆元 $(i!)^{-1}$。求 $\binom{n}{m} \bmod p$，当 $m=0$ 直接就是 1. 若 n,m 都在 p 范围内，则直接转化为 $n! \times (m!)^{-1} \times [(n-m)!]^{-1}$. 否则就是 *lucas* 定理缩小规模。

对一个固定的 p，预处理求阶乘及快速模幂求其逆元，时间复杂度 $O(p\log_2 p)$。空间复杂度 $O(p)$。预处理之后，单次求 $\binom{n}{m} \bmod p$ 复杂度 $O(\log_p m)$

```
void prepare(ll p, vector<ll>&fac, vector<ll>&inv_fac) {
    fac.resize(p); inv_fac.resize(p);
    mod_sys mod;
    mod.set_mod(p);
    fac[0] = 1;
```

```
    inv_fac[0] = 1;
    for (int i = 1; i < p; ++i) {
        fac[i] = (fac[i-1]*i)%p;
        inv_fac[i] = mod.pow(fac[i], p-2); // 既然能枚举一遍，p*p不应该爆ll
    }
}

// 输入预设0=<n,m<p
inline ll combination(ll n, ll m, ll p, vector<ll>&fac, vector<ll>&inv_fac) {
    if (n < m) return 0;
    return fac[n]*inv_fac[m]%p*inv_fac[n-m]%p;
}

ll lucas(ll n, ll m, ll p, vector<ll>&fac, vector<ll>&inv_fac) {
    if (n < m) return 0;
    ll ans = 1;
    while(true) {
        if (m == 0) return ans;
        if (n < p && m < p) return ans*combination(n,m,p,fac,inv_fac)%p;
        ans = ans * combination(n%p,m%p,p,fac,inv_fac)%p;
        n/=p; m/=p;
    }
}
```

6 欧拉函数

$\phi(n)$ 表示 1 到 n 中与 n 互素的个数.

公式 $\phi(n) = n \prod_{p|n\&p\in P} 1 - \frac{1}{p}$

$\phi(1..n)$ 总体求解可用线性筛 $O(n)$ 求出.

单个 $\phi(n)$ 可用分解质因数法直接用公式 $O(\sqrt{n}$ 求出。

```
// 时间复杂度sqrt(n)求phi(n) n最大1e12-1e14的级别
// more beautiful version, but slower (just a little bit)
ll phi(ll n) {
  ll a = n;
  for (ll p = 2; p * p <= n; ++p)
    if (!(n % p)) {
      do
        n /= p;
      while (!(n % p));
      a = a / p * (p - 1);
    }
  if (n > 1) a = a / n * (n - 1);  // the rest n is a prime
  return a;
```

```cpp
}

// 计算1--n的所有phi(i) 线性时空复杂度, n应该最大是1e7级别的
void get_all_phi(int n, vector<int>& phi) {
  phi.resize(n + 1);
  vector<bool> is_prime(n + 1, true);
  vector<int> prime;
  is_prime[1] = is_prime[0] = false;
  phi[1] = 1;
  for (int i = 2; i <= n; ++i) {
    if (is_prime[i]) {
      prime.push_back(i);
      phi[i] = i - 1;
    }
    for (auto p : prime) {
      if (i * p > n) break;
      is_prime[i * p] = false;
      if (i % p) {
        // i不具有素因子p, i*p对于素因子p来讲次数=1。贡献是(p-1)
        phi[i * p] = phi[i] * (p - 1);
      } else {
        phi[i * p] = phi[i] * p;  // i具有素因子p,则i*p对于欧拉函数值来讲乘以p
        break;  // 保证每个数只被最小素因子访问到。
      }
    }
  }
}
```

7 线性筛

时空复杂度 $O(n)$.

```cpp
void linear_sieve(int n, vector<int>& f) {
  f.resize(n + 1);
  vector<bool> is_prime(n + 1, true);
  vector<int> prime;
  is_prime[1] = is_prime[0] = false;
  f[1] = 1;
  for (int i = 2; i <= n; ++i) {
    if (is_prime[i]) {
      prime.push_back(i);
      // code here for i 当i为素数
    }
    for (auto p : prime) {
      if (i * p > n) break;
```

```
      is_prime[i * p] = false;
      if (i % p) {
        // code here for (i * p), 当(i * p)关于素因子p的次数大于等于2
      } else {
        // code here for (i * p), 当(i * p)关于素因子p的次数仅仅为1
        break;  // 保证每个数只被最小素因子访问到。保证线性。
      }
    }
  }
}
```

8 min 25 筛

min 25 筛即**基于质因数分解的亚线性函数前缀和求法**，可以在 $O(\frac{n^{\frac{3}{4}}}{\log n})$ 的时间内求积性函数 $f(x)$ 的前缀和。要求 $f(p)$ 是一个关于 p 的简单多项式，$f(p^c)$ 可以快速计算。

9 莫比乌斯反演

F 是已知函数，f 是未知函数。$\mu(x)$ 是固定函数，即**莫比乌斯函数**。

9.1 莫比乌斯函数 $\mu(x)$

- $\mu(1) = 1$
- x 为不同的质数的乘积。若质数个数为奇数，则 $\mu(x) = -1$; 偶数个 $\mu(x) = 1$.
- 剩下的情况，即 x 的某个素因子的次数大于等于 2. $\mu(x) = 0$.

莫比乌斯函是积性函数，10^7 规模的数据，故可用**线性筛思想**解决，否则需要使用杜教筛。

```
// 计算所有的\mu(x) x in [1..n]. 线性复杂度。
void get_all_mu(int n, vector<int>&mu) {
  mu.resize(n+1);
  vector<bool>is_prime(n+1, true);
  vector<int>prime;
  is_prime[1] = is_prime[0] = false; mu[1] = 1;
  for (int i = 2; i <= n; ++i) {
    if (is_prime[i]) {
      prime.push_back(i);
      mu[i] = -1;
    }
    for (auto p : prime) {
      if (i*p > n) break;
      is_prime[i*p] = false;
```

```
      if (i%p) {
        mu[i*p] = -mu[i]; // i不具有素因子p, 具有积性
      } else {
        mu[i*p] = 0; // i具有素因子p,则i*p具有素因子的平方的因子
        break; // 保证每个数只被最小素因子访问到。
      }
    }
  }
}
```

9.2 整除分块

基本形式 $\sum_{i=1}^{n}\lfloor\frac{n}{i}\rfloor$. 假设 n 的因数从小到大保存在 d[1..k] 中。

$$\mathbf{d} = \left(\ d_1, d_2, d_3, \ldots, d_k \ \right) \tag{1}$$

显然 $i \in (d_j, d_{j+1}]$ 时 $\lfloor\frac{n}{i}\rfloor$ 结果相同, 假设结果为 k, 则 $d_{j+1} = \lfloor\frac{n}{k}\rfloor$。因此可以一段一段区间的跳跃求和，将复杂度降到 $\sqrt{n}$.

大多数时候不是基本形式，而是基本形式乘以一个函数求和。这个时候，可以对这个函数求前缀和，然后用整除分块的代码做一下变通即可。

```
for(int l=1,r;l<=n;l=r+1) {
  r=n/(n/l);
  ans+=(r-l+1)*(n/l)*(sum[r]-sum[l-1]); // sum是乘以函数的前缀和
}
```

9.3 莫比乌斯反演定理

9.3.1 因数形式

d 取遍 n 的所有因数。$F(n) = \sum_{d|n} f(d)$, 反演求出未知的 f 的表达式为 $f(n) = \sum_{d|n}\mu(d)F(\frac{n}{d}) = \sum_{d|n}F(d)\mu(\frac{n}{d})$

9.3.2 倍数形式

d 取遍 n 的所有倍数。$F(n) = \sum_{n|d} f(d)$, 反演求出未知的 f 的表达式为 $f(n) = \sum_{n|d}\mu(\frac{d}{n})F(d)$.

9.4 做题思路或者技巧

1. 构造能够直接写出表达式的已知函数 F，然后尝试因数或者倍数形式的小 f.
2. 利用反演定理求出 f 的表达式。

3. 将 ans 用小 f 的形式表达出来。可能还是一个关于 f 的 Σ 求和式。
4. 代入 f 的反演结果，根据数学尝试变换枚举变量的顺序。
5. 与 gcd 有关的莫比乌斯反演。一般我们都是套路的去设 $f(d)$为$gcd(i,j)=d$的个数，$F(n)$为$gcd(i,j)=d$的倍数 的个数。
6. 注意最后的式子是不是包含一个整除分块的部分。

10 线段树

采用左闭右开区间模式。数组元素、线段树节点下标通通从 0 开始记。$p = (ch-1)/2, l = 2p+1, r = 2p+2$.

元素个数为 $item\_sz$, 线段树节点数组大小 $seg\_nd\_sz = 2^{\lceil \log_2 item\_sz \rceil + 1} - 1 < 2^{\log_2 item\_sz + 1 + 1} = 4 \times item\_sz$. 放缩操作是向上取整的结果肯定小于直接加 1 的结果，并且还没有减去最后需要减去的 1. 所以，简单奢侈的方式是直接开 *4* 倍。

lazy 标记，就是记录区间中元素共有的操作。节点的最值、和等标记是不考虑节点自身的 lazy 标记的结果，但是这些标记由子节点求解的时候需要考虑子节点的 lazy 标记；但是对于最值、和等的询问，返回的值是考虑了 lazy 标记的结果。

即 *mx,mn,sum* 标记的值和 *get_min,get_mn,get_sum* 的返回的值不一定相同。

10.1 区间集体加 + 查询区间和最大最小值模板

```
#include <bits/stdc++.h>
using namespace std;
typedef long long ll;

namespace lly {
// 所有下标符合C++风格
// 奢侈做法：使用全局变量开4倍数组。
// 最小值，最大值，区间和，区间加相同数模板
// 考场抄代码所有item_type, sum_type 换成ll即可
struct segment_tree {
  // 最大的初始数组大小和响应的线段树节点数组最大大小。
  const static int kMaxItemSize = 100000;
  const static int kMaxSegTreeSize = kMaxItemSize << 2;

  int item_sz;
  int seg_sz;
  int &n = item_sz;
  int &stn = seg_sz;

  typedef ll item_type;
  typedef ll sum_type;

  struct nd {
    int l, r;
    item_type mx;  // max
    item_type mn;  // min
    sum_type sm;   // sum flag
    // lazy flags
    item_type all_add;  // lazy标记，表示区间内的数都要加上的数
    // other flags
```

```
  inline void add(item_type a) {
    all_add += a;
    mx += a;
    mn += a;
    sm += (sum_type)(r-l)*a;
  }
  inline int mid() { return l + (r - l) / 2; }
};

item_type a[kMaxItemSize];
nd nds[kMaxSegTreeSize];

void init(int cnt) {  // 屏幕输入数组a版本
  n = cnt;
  for (int i = 0; i < n; ++i) scanf("%lld", a+i);//cin >> a[i];
  seg_sz = (2 << (int)(ceil( log2(item_sz) ))) - 1;
}

void init(item_type src[], int cnt) {  // 内存数组输入数组a版本
  n = cnt;
  for (int i = 0; i < n; ++i) a[i] = src[i];
  seg_sz = (2 << (int)(ceil( log2(item_sz) ))) - 1;
}

inline int parent(int x) { return (x - 1) >> 1; }
inline int lchild(int x) { return (x << 1) | 1; }
inline int rchild(int x) { return (x << 1) + 2; }

inline void build() { build(0, n, 0); }

inline void set_flags(int root, int i) {  // nds[root]用a[i]设置各类标志
  auto &p = nds[root];
  p.l = i;
  p.r = i + 1;
  p.mx = a[i];
  p.mn = a[i];
  p.sm = a[i];
}

inline void merge_flags(int root) {
  auto &p = nds[root];
  auto &l = nds[lchild(root)];
  auto &r = nds[rchild(root)];
  p.l = l.l;
  p.r = r.r;
  p.mx = max(l.mx, r.mx);
```

```
  p.mn = min(l.mn, r.mn);
  p.sm = l.sm + r.sm + p.all_add * (sum_type)(p.r - p.l);
}

void build(int l, int r, int root) {
  nds[root].all_add = 0;
  if (l + 1 == r) {
    set_flags(root, l);
    return;
  }
  int m = l + (r - l) / 2;
  build(l, m, lchild(root));
  build(m, r, rchild(root));
  merge_flags(root);
}

// [l,r)区间的数都加上val
void add(int l, int r, item_type val, int root = 0) {
  if (l == nds[root].l && r == nds[root].r) {
    nds[root].add(val);
    return;
  }
  int m = nds[root].mid();
  if (r <= m) {  // only left part
    add(l, r, val, lchild(root));
  } else if (l >= m) {  // only right part
    add(l, r, val, rchild(root));
  } else {
    add(l, m, val, lchild(root));
    add(m, r, val, rchild(root));
  }
  merge_flags(root);
}

item_type get_max(int l, int r, int root = 0) {
  if (l == nds[root].l && r == nds[root].r) {
    return nds[root].mx;
  }
  // 勿忘加上all_add lazy标记
  int m = nds[root].mid();
  if (r <= m) {  // only left part
    return get_max(l, r, lchild(root)) + nds[root].all_add;
  } else if (l >= m) {  // only right part
    return get_max(l, r, rchild(root)) + nds[root].all_add;
  } else {
    return max(get_max(l, m, lchild(root)),  // left
```

```
                 get_max(m, r, rchild(root)))  // right
             + nds[root].all_add;
    }
  }

  item_type get_min(int l, int r, int root = 0) {
    if (l == nds[root].l && r == nds[root].r) {
      return nds[root].mn;
    }
    // 勿忘加上all_add lazy标记
    int m = nds[root].mid();
    if (r <= m) {  // only left part
      return get_min(l, r, lchild(root)) + nds[root].all_add;
    } else if (l >= m) {  // only right part
      return get_min(l, r, rchild(root)) + nds[root].all_add;
    } else {
      return min(get_min(l, m, lchild(root)),  // left
                 get_min(m, r, rchild(root)))  // right
             + nds[root].all_add;
    }
  }

  sum_type get_sum(int l, int r, int root = 0) {
    if (l == nds[root].l && r == nds[root].r) {
      return nds[root].sm;
    }
    // 勿忘加上all_add lazy标记
    int m = nds[root].mid();
    ll lazy = nds[root].all_add * (sum_type)(r - l);
    if (r <= m) {  // only left part
      return get_sum(l, r, lchild(root)) + lazy;
    } else if (l >= m) {  // only right part
      return get_sum(l, r, rchild(root)) + lazy;
    } else {
      return get_sum(l, m, lchild(root))    // left
             + get_sum(m, r, rchild(root))  // right
             + lazy;
    }
  }

  void out() {
    cout << "n = " << n << "\n";
    for (int i = 0; i < n; ++i) cout << a[i] << " ";
    cout << "\n";
  }
};
```

```
};  // namespace lly

lly::segment_tree tr;

int main() {
  // 洛谷 P3372 【模板】线段树 1

  int n, m;
  //cin>>n>>m;
  scanf("%d%d",&n,&m);
  tr.init(n);
  tr.build();

  int o,x,y;
  ll k;
  for (int i = 0; i < m; ++i) {
    //cin>>o>>x>>y;
    scanf("%d%d%d",&o,&x,&y);
    --x;
    if (o == 1) { // [x,y)内都加上k
      scanf("%lld",&k);
      tr.add(x,y,k);
    } else { // 询问区间和
      auto sum = tr.get_sum(x,y);
      printf("%lld\n",sum);
    }
  }
  return 0;
}
```

10.2 区间集体加与乘 + 查询区间和线段树模板

```
#include <bits/stdc++.h>
using namespace std;
typedef long long ll;

namespace lly {
// 所有下标符合C++风格
// 奢侈做法: 使用全局变量开4倍数组。
// 考场抄代码所有item_type, sum_type 换成ll即可
// 支持区间和查询(%mod) 区间集体乘k, 区间集体加k
// kMaxItemSize是最大的原始数据数组的大小
// 区间加 下传所有路径上的mlt标记
// 区间乘 下传所有路径上的mlt标记和add标记
// 注意考场上抄代码敲完一个函数, 别的函数可以复制然后粘贴替换, 别少替换了。
```

```
struct segment_tree {
  // 最大的初始数组大小和响应的线段树节点数组最大大小。
  const static int kMaxItemSize = 100000; // 可手动更换
  const static int kMaxSegTreeSize = kMaxItemSize << 2;

  ll mod;

  int item_sz;
  int seg_sz;
  int &n = item_sz;
  int &stn = seg_sz;

  typedef ll item_type;
  typedef ll sum_type;

  struct nd {
    int l, r;
    sum_type sm;  // sum flag
    // lazy flags
    item_type all_add;  // lazy标记，表示区间内的数都要加上的数
    sum_type all_mlt;   // lazy标记，表示区间内的数都要乘以的数
    // 运算顺序，先乘后加，即先乘以all_mlt再加上all_add；先儿子运算，后父亲运
        算。
    // 因此区间×k操作，需要把all_add标记乘以k

    // other flags
    inline int mid() { return l + (r - l) / 2; }
    inline void set_basic(int l, int r) {
      this->l = l;
      this->r = r;
      all_add = 0;
      all_mlt = 1;
    }
    inline void add(item_type a, ll mod) {
      a %= mod;
      all_add = (all_add + a) % mod;
      sm = (sm + (sum_type)(r - l) * a) % mod;
    }
    inline void mlt(item_type m, ll mod) {
      m %= mod;
      all_add = (all_add * (sum_type)m) % mod;
      all_mlt = (all_mlt * m) % mod;
      sm = (sm * m) % mod;
    }
  };
```

```
item_type a[kMaxItemSize];
nd nds[kMaxSegTreeSize];

void init() {  // 屏幕输入数组a版本，需要先确定n
  for (int i = 0; i < n; ++i) scanf("%lld", a + i);  // cin >> a[i];
  seg_sz = (2 << (int)(ceil(log2(item_sz)))) - 1;
}

void init(item_type src[], int cnt) {  // 内存数组输入数组a版本
  n = cnt;
  for (int i = 0; i < n; ++i) a[i] = src[i];
  seg_sz = (2 << (int)(ceil(log2(item_sz)))) - 1;
}

inline int parent(int x) { return (x - 1) >> 1; }
inline int lchild(int x) { return (x << 1) | 1; }
inline int rchild(int x) { return (x << 1) + 2; }

inline void build() { build(0, n, 0); }

inline void set_flags(int root, int i) {  // nds[root]用a[i]设置各类标志
  auto &p = nds[root];
  p.sm = a[i];
}

inline void merge_flags(int root) {
  auto &p = nds[root];
  auto &l = nds[lchild(root)];
  auto &r = nds[rchild(root)];
  p.sm = (l.sm + r.sm) % mod * (p.all_mlt) % mod +
         ((sum_type)(p.r - p.l) * p.all_add) % mod;
  p.sm %= mod;
}

inline void down_mlt_flag(int root) {
  auto &p = nds[root];
  auto &l = nds[lchild(root)];
  auto &r = nds[rchild(root)];
  l.mlt(p.all_mlt, mod);
  r.mlt(p.all_mlt, mod);
  p.all_mlt = 1;
}

inline void down_flags(int root) {  // mlt and add
  auto &p = nds[root];
  auto &l = nds[lchild(root)];
```

```
    auto &r = nds[rchild(root)];
    l.mlt(p.all_mlt, mod);
    r.mlt(p.all_mlt, mod);
    p.all_mlt = 1;
    l.add(p.all_add, mod);
    r.add(p.all_add, mod);
    p.all_add = 0;
  }

  void build(int l, int r, int root) {
    nds[root].set_basic(l, r);
    if (l + 1 == r) {
      set_flags(root, l);
      return;
    }
    int m = l + (r - l) / 2;
    build(l, m, lchild(root));
    build(m, r, rchild(root));
    merge_flags(root);
  }

  // [l,r)区间的数都加上val
  void add(int l, int r, item_type val, int root = 0) {
    if (l == nds[root].l && r == nds[root].r) {
      nds[root].add(val, mod);
      return;
    }
    down_mlt_flag(root);
    int m = nds[root].mid();
    if (r <= m) {  // only left part
      add(l, r, val, lchild(root));
    } else if (l >= m) {  // only right part
      add(l, r, val, rchild(root));
    } else {
      add(l, m, val, lchild(root));
      add(m, r, val, rchild(root));
    }
    merge_flags(root);
  }

  // [l,r)区间的数都乘以val
  void mlt(int l, int r, item_type val, int root = 0) {
    if (l == nds[root].l && r == nds[root].r) {
      nds[root].mlt(val, mod);
      return;
    }
```

```cpp
    down_flags(root);
    int m = nds[root].mid();
    if (r <= m) {  // only left part
      mlt(l, r, val, lchild(root));
    } else if (l >= m) {  // only right part
      mlt(l, r, val, rchild(root));
    } else {
      mlt(l, m, val, lchild(root));
      mlt(m, r, val, rchild(root));
    }
    merge_flags(root);
  }

  sum_type get_sum(int l, int r, int root = 0) {
    if (l == nds[root].l && r == nds[root].r) {
      return nds[root].sm;
    }
    // 勿忘加上all_add lazy标记
    int m = nds[root].mid();
    ll lazy = (nds[root].all_add * (sum_type)(r - l)) % mod;
    ll tmp;
    if (r <= m) {  // only left part
      tmp = get_sum(l, r, lchild(root));
    } else if (l >= m) {  // only right part
      tmp = get_sum(l, r, rchild(root));
    } else {
      tmp = get_sum(l, m, lchild(root))     // left
            + get_sum(m, r, rchild(root));  // right
    }
    tmp %= mod;
    return (tmp * nds[root].all_mlt % mod + lazy) % mod;
  }

  void out() {
    cout << "n = " << n << "\n";
    for (int i = 0; i < n; ++i) cout << a[i] << " ";
    cout << "\n";
  }
};
};  // namespace lly

lly::segment_tree tr;

int main() {
  // 洛谷 P3373 【模板】线段树 2
```

```
  int n, m;
  ll mod;
  // cin>>n>>m;
  scanf("%d%d%lld", &n, &m, &mod);
  tr.n = n;
  tr.mod = mod;
  tr.init();
  tr.build();

  int o, x, y;
  ll k;
  for (int i = 0; i < m; ++i) {
    // cin>>o>>x>>y;
    scanf("%d%d%d", &o, &x, &y);
    --x;
    if (o == 2) {  // [x,y)内都加上k
      scanf("%lld", &k);
      tr.add(x, y, k);
    } else if (o == 3) {  // 询问区间和
      auto sum = tr.get_sum(x, y);
      printf("%lld\n", sum);
    } else {
      scanf("%lld", &k);
      tr.mlt(x, y, k);
    }
  }
  return 0;
}
```