Min_25 筛

定义

从此种筛法的思想方法来说，其又被称为「Extended Eratosthenes Sieve」。

由于其由 Min_25 发明并最早开始使用，故称「Min_25 筛」。

性质

其可以在 $O\left(\frac{n^{\frac{3}{4}}}{\log n}\right)$ 或 $\Theta\left(n^{1-\epsilon}\right)$ 的时间复杂度下解决一类 **积性函数** 的前缀和问题。

要求：$f(p)$ 是一个关于 p 可以快速求值的完全积性函数之和（例如多项式）；$f(p^c)$ 可以快速求值。

记号

- **如无特别说明，本节中所有记为 p 的变量的取值集合均为全体质数。**
- $x/y := \left\lfloor \frac{x}{y} \right\rfloor$
- $\operatorname{isprime}(n) := [|\{d : d \mid n\}| = 2]$，即 n 为质数时其值为 1，否则为 0。
- p_k：全体质数中第 k 小的质数（如：$p_1 = 2, p_2 = 3$）。特别地，令 $p_0 = 1$。
- $\operatorname{lpf}(n) := [1 < n] \min\{p : p \mid n\} + [1 = n]$，即 n 的最小质因数。特别地，$n = 1$ 时，其值为 1。
- $F_{\mathrm{prime}}(n) := \sum_{2 \le p \le n} f(p)$
- $F_k(n) := \sum_{i=2}^{n} [p_k \le \operatorname{lpf}(i)] f(i)$

解释

观察 $F_k(n)$ 的定义，可以发现答案即为 $F_1(n) + f(1) = F_1(n) + 1$。

考虑如何求出 $F_k(n)$。通过枚举每个 i 的最小质因子及其次数可以得到递推式：

$$
\begin{aligned}
F_k(n) &= \sum_{i=2}^{n}[p_k \le \operatorname{lpf}(i)]f(i) \\
&= \sum_{\substack{k\le i\\ p_i^2\le n}} \sum_{\substack{c\ge 1\\ p_i^c\le n}} f\left(p_i^c\right)([c>1] + F_{i+1}\left(n/p_i^c\right)) + \sum_{\substack{k\le i\\ p_i\le n}} f(p_i) \\
&= \sum_{\substack{k\le i\\ p_i^2\le n}} \sum_{\substack{c\ge 1\\ p_i^c\le n}} f\left(p_i^c\right)([c>1] + F_{i+1}\left(n/p_i^c\right)) + F_{\text{prime}}(n) - F_{\text{prime}}(p_{k-1}) \\
&= \sum_{\substack{k\le i\\ p_i^2\le n}} \sum_{\substack{c\ge 1\\ p_i^{c+1}\le n}} \left(f\left(p_i^c\right)F_{i+1}\left(n/p_i^c\right) + f\left(p_i^{c+1}\right)\right) + F_{\text{prime}}(n) - F_{\text{prime}}(p_{k-1})
\end{aligned}
$$

最后一步推导基于这样一个事实：对于满足 $p_i^c \le n < p_i^{c+1}$ 的 c，有 $p_i^{c+1} > n \iff n/p_i^c < p_i < p_{i+1}$，故 $F_{i+1}\left(n/p_i^c\right) = 0$。
其边界值即为 $F_k(n) = 0 (p_k > n)$。

假设现在已经求出了所有的 $F_{\text{prime}}(n)$，那么有两种方式可以求出所有的 $F_k(n)$：

1. 直接按照递推式计算。
2. 从大到小枚举 p 转移，仅当 $p^2 < n$ 时转移增加值不为零，故按照递推式后缀和优化即可。

现在考虑如何计算 $F_{\text{prime}}(n)$。
观察求 $F_k(n)$ 的过程，容易发现 F_{prime} 有且仅有 $1, 2, \dots, \lfloor\sqrt{n}\rfloor, n/\sqrt{n}, \dots, n/2, n$ 这 $O(\sqrt{n})$ 处的点值是有用的。
一般情况下，$f(p)$ 是一个关于 p 的低次多项式，可以表示为 $f(p) = \sum a_i p^{c_i}$。
那么对于每个 p^{c_i}，其对 $F_{\text{prime}}(n)$ 的贡献即为 $a_i \sum_{2\le p\le n} p^{c_i}$。
分开考虑每个 p^{c_i} 的贡献，问题就转变为了：给定 $n, s, g(p) = p^s$，对所有的 $m = n/i$，求 $\sum_{p\le m} g(p)$。

> Notice：$g(p) = p^s$ 是完全积性函数！

于是设 $G_k(n) := \sum_{i=2}^{n} [p_k < \operatorname{lpf}(i) \lor \operatorname{isprime}(i)]g(i)$，即埃筛第 k 轮筛完后剩下的数的 g 值之和。
对于一个合数 $x \le n$，必定有 $\operatorname{lpf}(x) \le \sqrt{x} \le \sqrt{n}$。设 $p_{\ell(n)}$ 为不大于 $\sqrt{n}$ 的最大质数，则 $\sum_{2\le p\le n} g(p) = G_{\ell(n)}(n)$，即在埃筛进行 ℓ 轮之后剩下的均为质数。 考虑 G 的边界值，显然为 $G_0(n) = \sum_{i=2}^{n} g(i)$。（还记得吗？特别约定了 $p_0 = 1$）
对于转移，考虑埃筛的过程，分开讨论每部分的贡献，有：

1. 对于 $n < p_k^2$ 的部分，G 值不变，即 $G_k(n) = G_{k-1}(n)$。
2. 对于 $p_k^2 \le n$ 的部分，被筛掉的数必有质因子 p_k，即 $-g(p_k)G_{k-1}(n/p_k)$。
3. 对于第二部分，由于 $p_k^2 \le n \iff p_k \le n/p_k$，满足 $\operatorname{lpf}(i) < p_k$ 的 i 会被额外减去。这部分应当加回来，即 $g(p_k)G_{k-1}(p_{k-1})$。

则有：

$$
G_k(n) = G_{k-1}(n) - [p_k^2 \le n]g(p_k)(G_{k-1}(n/p_k) - G_{k-1}(p_{k-1}))
$$

复杂度分析

对于 $F_k(n)$ 的计算，其第一种方法的时间复杂度被证明为 $O\left(n^{1-\epsilon}\right)$（见 zzt 集训队论文 2.3）；
对于第二种方法，其本质即为洲阁筛的第二部分，在洲阁论文中也有提及（6.5.4），其时间复杂度被证明为 $O\left(\frac{n^{\frac{3}{4}}}{\log n}\right)$。

对于 $F_{\text{prime}}(n)$ 的计算，事实上，其实现与洲阁筛第一部分是相同的。
考虑对于每个 $m = n/i$，只有在枚举满足 $p_k^2 \le m$ 的 p_k 转移时会对时间复杂度产生贡献，则时间复杂度可估计为：

$$
\begin{aligned}
T(n) &= \sum_{i^2 \le n} O\left(\pi\left(\sqrt{i}\right)\right) + \sum_{i^2 \le n} O\left(\pi\left(\sqrt{\frac{n}{i}}\right)\right) \\
&= \sum_{i^2 \le n} O\left(\frac{\sqrt{i}}{\ln \sqrt{i}}\right) + \sum_{i^2 \le n} O\left(\frac{\sqrt{\frac{n}{i}}}{\ln \sqrt{\frac{n}{i}}}\right) \\
&= O\left(\int_1^{\sqrt{n}} \frac{\sqrt{\frac{n}{x}}}{\log \sqrt{\frac{n}{x}}} \mathrm{d}x\right) \\
&= O\left(\frac{n^{\frac{3}{4}}}{\log n}\right)
\end{aligned}
$$

对于空间复杂度，可以发现不论是 F_k 还是 F_{prime}，其均只在 n/i 处取有效点值，共 $O(\sqrt{n})$ 个，仅记录有效值即可将空间复杂度优化至 $O(\sqrt{n})$。

首先，通过一次数论分块可以得到所有的有效值，用一个大小为 $O(\sqrt{n})$ 的数组 lis 记录。对于有效值 v，记 $\text{id}(v)$ 为 v 在 lis 中的下标，易得：对于所有有效值 v，$\text{id}(v) \le \sqrt{n}$。

然后分开考虑小于等于 $\sqrt{n}$ 的有效值和大于 $\sqrt{n}$ 的有效值：对于小于等于 $\sqrt{n}$ 的有效值 v，用一个数组 le 记录其 $\text{id}(v)$，即 $\text{le}_v = \text{id}(v)$；对于大于 $\sqrt{n}$ 的有效值 v，用一个数组 ge 记录 $\text{id}(v)$，由于 v 过大所以借助 $v' = n/v < \sqrt{n}$ 记录 $\text{id}(v)$，即 $\text{ge}_{v'} = \text{id}(v)$。

这样，就可以使用两个大小为 $O(\sqrt{n})$ 的数组记录所有有效值的 id 并 $O(1)$ 查询。在计算 F_k 或 F_{prime} 时，使用有效值的 id 代替有效值作为下标，即可将空间复杂度优化至 $O(\sqrt{n})$。

过程

对于 $F_k(n)$ 的计算，我们实现时一般选择实现难度较低的第一种方法，其在数据规模较小时往往比第二种方法的表现要好；

对于 $F_{\text{prime}}(n)$ 的计算，直接按递推式实现即可。

对于 $p_k^2 \le n$，可以用线性筛预处理出 $s_k := F_{\text{prime}}(p_k)$ 来替代 F_k 递推式中的 $F_{\text{prime}}(p_{k-1})$。
相应地，G 递推式中的 $G_{k-1}(p_{k-1}) = \sum_{i=1}^{k-1} g(p_i)$ 也可以用此方法预处理。

用 Extended Eratosthenes Sieve 求 **积性函数** f 的前缀和时，应当明确以下几点：

- 如何快速（一般是线性时间复杂度）筛出前 $\sqrt{n}$ 个 f 值；

- $f(p)$ 的多项式表示；
- 如何快速求出 $f(p^c)$。

明确上述几点之后按顺序实现以下几部分即可：

1. 筛出 $[1, \sqrt{n}]$ 内的质数与前 $\sqrt{n}$ 个 f 值；
2. 对 $f(p)$ 多项式表示中的每一项筛出对应的 G，合并得到 F_{prime} 的所有 $O(\sqrt{n})$ 个有用点值；
3. 按照 F_k 的递推式实现递归，求出 $F_1(n)$。

例题

求莫比乌斯函数的前缀和

求 $\sum\limits_{i=1}^{n}\mu(i)$。

易知 $f(p) = -1$。则 $g(p) = -1, G_0(n) = \sum_{i=2}^{n} g(i) = -n + 1$。
直接筛即可得到 F_{prime} 的所有 $O(\sqrt{n})$ 个所需点值。

求欧拉函数的前缀和

求 $\sum\limits_{i=1}^{n}\varphi(i)$。

首先易知 $f(p) = p - 1$。
对于 $f(p)$ 的一次项 (p)，有 $g(p) = p, G_0(n) = \sum_{i=2}^{n} g(i) = \frac{(n+2)(n-1)}{2}$；
对于 $f(p)$ 的常数项 (-1)，有 $g(p) = -1, G_0(n) = \sum_{i=2}^{n} g(i) = -n + 1$。
筛两次加起来即可得到 F_{prime} 的所有 $O(\sqrt{n})$ 个所需点值。

「LOJ #6053」简单的函数

给定 $f(n)$：

$$f(n) = \begin{cases} 1 & n = 1 \\ p \operatorname{xor} c & n = p^c \\ f(a)f(b) & n = ab \land a \perp b \end{cases}$$

易知 $f(p) = p - 1 + 2[p = 2]$。则按照筛 φ 的方法筛，对 2 讨论一下即可。
此处给出一种 C++ 实现：

参考代码

```cpp
/* 「LOJ #6053」简单的函数 */
#include <cmath>
#include <iostream>

constexpr int MAXS = 200000;  // 2sqrt(n)
constexpr int mod = 1000000007;

template <typename x_t, typename y_t>
void inc(x_t &x, const y_t &y) {
  x += y;
  (mod <= x) && (x -= mod);
}

template <typename x_t, typename y_t>
void dec(x_t &x, const y_t &y) {
  x -= y;
  (x < 0) && (x += mod);
}

template <typename x_t, typename y_t>
int sum(const x_t &x, const y_t &y) {
  return x + y < mod ? x + y : (x + y - mod);
}

template <typename x_t, typename y_t>
int sub(const x_t &x, const y_t &y) {
  return x < y ? x - y + mod : (x - y);
}

template <typename _Tp>
int div2(const _Tp &x) {
  return ((x & 1) ? x + mod : x) >> 1;
}

// 以上目的均为防负数和取模
template <typename _Tp>
long long sqrll(const _Tp &x) {  // 平方函数
  return (long long)x * x;
}

int pri[MAXS / 7], lpf[MAXS + 1], spri[MAXS + 1], pcnt;

void sieve(const int &n) {
  for (int i = 2; i <= n; ++i) {
    if (lpf[i] == 0) {  // 记录质数
      lpf[i] = ++pcnt;
      pri[lpf[i]] = i;
      spri[pcnt] = sum(spri[pcnt - 1], i);  // 前缀和
    }
```

```
    for (int j = 1, v; j <= lpf[i] && (v = i * pri[j]) <= n;
++j) lpf[v] = j;
  }
}

long long global_n;
int lim;
int le[MAXS + 1],  // x <= \sqrt{n}
    ge[MAXS + 1];  // x > \sqrt{n}
#define idx(v) (v <= lim ? le[v] : ge[global_n / v])

int G[MAXS + 1][2], Fprime[MAXS + 1];
long long lis[MAXS + 1];
int cnt;

void init(const long long &n) {
  for (long long i = 1, j, v; i <= n; i = n / j + 1) {
    j = n / i;
    v = j % mod;
    lis[++cnt] = j;
    (j <= lim ? le[j] : ge[global_n / j]) = cnt;
    G[cnt][0] = sub(v, 1ll);
    G[cnt][1] = div2((long long)(v + 2ll) * (v - 1ll) % mod);
  }
}

void calcFprime() {
  for (int k = 1; k <= pcnt; ++k) {
    const int p = pri[k];
    const long long sqrp = sqrll(p);
    for (int i = 1; lis[i] >= sqrp; ++i) {
      const long long v = lis[i] / p;
      const int id = idx(v);
      dec(G[i][0], sub(G[id][0], k - 1));
      dec(G[i][1], (long long)p * sub(G[id][1], spri[k - 1]) %
mod);
    }
  }
  /* F_prime = G_1 - G_0 */
  for (int i = 1; i <= cnt; ++i) Fprime[i] = sub(G[i][1], G[i]
[0]);
}

int f_p(const int &p, const int &c) {
  /* f(p^{c}) = p xor c */
  return p ^ c;
}

int F(const int &k, const long long &n) {
  if (n < pri[k] || n <= 1) return 0;
  const int id = idx(n);
  long long ans = Fprime[id] - (spri[k - 1] - (k - 1));
```

```
  if (k == 1) ans += 2;
  for (int i = k; i <= pcnt && sqrll(pri[i]) <= n; ++i) {
    long long pw = pri[i], pw2 = sqrll(pw);
    for (int c = 1; pw2 <= n; ++c, pw = pw2, pw2 *= pri[i])
      ans +=
          ((long long)f_p(pri[i], c) * F(i + 1, n / pw) +
f_p(pri[i], c + 1)) %
          mod;
  }
  return ans % mod;
}

using std::cin;
using std::cout;

int main() {
  cin.tie(nullptr)->sync_with_stdio(false);
  cin >> global_n;
  lim = sqrt(global_n);  // 上限

  sieve(lim + 1000);  // 预处理
  init(global_n);
  calcFprime();
  cout << (F(1, global_n) + 1ll + mod) % mod << '\n';

  return 0;
}
```

本页面最近更新：2025/9/7 21:50:39，更新历史

发现错误？想一起完善？ 在 GitHub 上编辑此页！

本页面贡献者：Marcythm, Xeonacid, MegaOwIer, StudyingFather, Tiphereth-A, CSPNOIP, Enter-tainer, aofall, Backl1ght, CoelacanthusHex, Great-designer, Haohu Shen, iamtwz, Ir1d, kenlig, Konano, ksyx, Persdre, Revltalize, SamZhangQingChuan, shuzhouliu, ZnPdCo

© 本页面的全部内容在 **CC BY-SA 4.0 和 SATA** 协议之条款下提供，附加条款亦可能应用