筛法

素数筛法

引入

如果我们想要知道小于等于 n 有多少个素数呢?

一个自然的想法是对于小于等于 n 的每个数进行一次质数检验。这种暴力的做法显然不能达到最优复杂度。

埃拉托斯特尼筛法

过程

考虑这样一件事情：对于任意一个大于 1 的正整数 n，那么它的 x 倍就是合数（$x > 1$）。利用这个结论，我们可以避免很多次不必要的检测。

如果我们从小到大考虑每个数，然后同时把当前这个数的所有（比自己大的）倍数记为合数，那么运行结束的时候没有被标记的数就是素数了。

实现

C++

```cpp
vector<int> prime;
bool is_prime[N];

void Eratosthenes(int n) {
  is_prime[0] = is_prime[1] = false;
  for (int i = 2; i <= n; ++i) is_prime[i] = true;
  for (int i = 2; i <= n; ++i) {
    if (is_prime[i]) {
      prime.push_back(i);
      if ((long long)i * i > n) continue;
      for (int j = i * i; j <= n; j += i)
        // 因为从 2 到 i - 1 的倍数我们之前筛过了，这里直接从 i
        // 的倍数开始，提高了运行速度
        is_prime[j] = false;  // 是 i 的倍数的均不是素数
    }
  }
}
```

Python

```python
prime = []
is_prime = [False] * N


def Eratosthenes(n):
    is_prime[0] = is_prime[1] = False
    for i in range(2, n + 1):
        is_prime[i] = True
    for i in range(2, n + 1):
        if is_prime[i]:
            prime.append(i)
            if i * i > n:
                continue
            for j in range(i * i, n + 1, i):
                is_prime[j] = False
```

以上为 **Eratosthenes 筛法**（埃拉托斯特尼筛法，简称埃氏筛法），时间复杂度是 $O(n\log\log n)$。

证明

现在我们就来看看推导过程：

如果每一次对数组的操作花费 1 个单位时间，则时间复杂度为：

$$O\left(\sum_{k=1}^{\pi(n)}\frac{n}{p_k}\right)=O\left(n\sum_{k=1}^{\pi(n)}\frac{1}{p_k}\right)$$

其中 p_k 表示第 k 小的素数，$\pi(n)$ 表示 $\le n$ 的素数个数。$\sum_{k=1}^{\pi(n)}$ 表示第一层 for 循环，其中累加上界 $\pi(n)$ 为 `if (prime[i])` 进入 true 分支的次数；$\frac{n}{p_k}$ 表示第二层 for 循环的执行次数。

根据 Mertens 第二定理，存在常数 B_1 使得：

$$\sum_{k=1}^{\pi(n)}\frac{1}{p_k}=\log\log n+B_1+O\left(\frac{1}{\log n}\right)$$

所以 **Eratosthenes 筛法** 的时间复杂度为 $O(n\log\log n)$。接下来我们证明 Mertens 第二定理的弱化版本 $\sum_{k\le\pi(n)}1/p_k=O(\log\log n)$：

根据 $\pi(n)=\Theta(n/\log n)$，可知第 n 个素数的大小为 $\Theta(n\log n)$。于是就有

$$\begin{aligned}\sum_{k=1}^{\pi(n)}\frac{1}{p_k}&=O\left(\sum_{k=2}^{\pi(n)}\frac{1}{k\log k}\right)\\&=O\left(\int_2^{\pi(n)}\frac{\mathrm{d}x}{x\log x}\right)\\&=O(\log\log\pi(n))=O(\log\log n)\end{aligned}$$

当然，上面的做法效率仍然不够高效，应用下面几种方法可以稍微提高算法的执行效率。

筛至平方根

显然，要找到直到 n 为止的所有素数，仅对不超过 $\sqrt{n}$ 的素数进行筛选就足够了。

C++

```cpp
vector<int> prime;
bool is_prime[N];

void Eratosthenes(int n) {
  is_prime[0] = is_prime[1] = false;
  for (int i = 2; i <= n; ++i) is_prime[i] = true;
  // i * i <= n 说明 i <= sqrt(n)
  for (int i = 2; i * i <= n; ++i) {
    if (is_prime[i])
      for (int j = i * i; j <= n; j += i) is_prime[j] = false;
  }
  for (int i = 2; i <= n; ++i)
    if (is_prime[i]) prime.push_back(i);
}
```

Python

```python
prime = []
is_prime = [False] * N


def Eratosthenes(n):
    is_prime[0] = is_prime[1] = False
    for i in range(2, n + 1):
        is_prime[i] = True
    # 让 i 循环到 <= sqrt(n)
    for i in range(2, isqrt(n) + 1):  # `isqrt` 是 Python 3.8 新增的函数
        if is_prime[i]:
            for j in range(i * i, n + 1, i):
                is_prime[j] = False
    for i in range(2, n + 1):
        if is_prime[i]:
            prime.append(i)
```

这种优化不会影响渐进时间复杂度，实际上重复以上证明，我们将得到 $n \ln \ln \sqrt{n} + o(n)$，根据对数的性质，它们的渐进相同，但操作次数会明显减少。

只筛奇数

因为除 2 以外的偶数都是合数，所以我们可以直接跳过它们，只用关心奇数就好。

首先，这样做能让我们内存需求减半；其次，所需的操作大约也减半。

减少内存的占用

我们注意到筛选时只需要 `bool` 类型的数组。`bool` 数组的一个元素一般占用 1 字节（即 8 比特），但是存储一个布尔值只需要 1 个比特就足够了。

我们可以使用 位运算 的相关知识，将每个布尔值压到一个比特位中，这样我们仅需使用 n 比特（即 $\dfrac{n}{8}$ 字节）而非 n 字节，可以显著减少内存占用。这种方式被称为「位级压缩」。

值得一提的是，存在自动执行位级压缩的数据结构，如 C++ 中的 `vector<bool>` 和 `bitset<>`。

另外，`vector<bool>` 和 `bitset<>` 对程序有常数优化，时间复杂度 $O(n \log\log n)$ 的埃氏筛使用 `bitset<>` 或 `vector<bool>` 优化后，性能甚至超过时间复杂度 $O(n)$ 的欧拉筛。

参见 bitset: 与埃氏筛结合。

分块筛选

由优化「筛至平方根」可知，不需要一直保留整个 `is_prime[1...n]` 数组。为了进行筛选，只保留到 $\sqrt{n}$ 的素数就足够了，即 `prime[1...sqrt(n)]`。并将整个范围分成块，每个块分别进行筛选。这样，我们就不必同时在内存中保留多个块，而且 CPU 可以更好地处理缓存。

设 s 是一个常数，它决定了块的大小，那么我们就有了 $\lceil \frac{n}{s} \rceil$ 个块，而块 $k(k = 0 \dots \lfloor \frac{n}{s} \rfloor)$ 包含了区间 $[ks, ks + s - 1]$ 中的数字。我们可以依次处理块，也就是说，对于每个块 k，我们将遍历所有质数（从 1 到 $\sqrt{n}$）并使用它们进行筛选。

值得注意的是，我们在处理第一个数字时需要稍微修改一下策略：首先，应保留 $[1, \sqrt{n}]$ 中的所有的质数；第二，数字 0 和 1 应该标记为非素数。在处理最后一个块时，不应该忘记最后一个数字 n 并不一定位于块的末尾。

以下实现使用块筛选来计算小于等于 n 的质数数量。

实现

```cpp
int count_primes(int n) {
  constexpr static int S = 10000;
  vector<int> primes;
  int nsqrt = sqrt(n);
  vector<char> is_prime(nsqrt + 1, true);
  for (int i = 2; i <= nsqrt; i++) {
    if (is_prime[i]) {
      primes.push_back(i);
      for (int j = i * i; j <= nsqrt; j += i) is_prime[j] =
false;
    }
  }
  int result = 0;
  vector<char> block(S);
  for (int k = 0; k * S <= n; k++) {
    fill(block.begin(), block.end(), true);
    int start = k * S;
    for (int p : primes) {
      int start_idx = (start + p - 1) / p;
      int j = max(start_idx, p) * p - start;
      for (; j < S; j += p) block[j] = false;
    }
    if (k == 0) block[0] = block[1] = false;
    for (int i = 0; i < S && start + i <= n; i++) {
      if (block[i]) result++;
    }
  }
  return result;
}
```

分块筛法的渐进时间复杂度与埃氏筛法是一样的（除非块非常小），但是所需的内存将缩小为 $O(\sqrt{n}+S)$，并且有更好的缓存结果。另一方面，对于每一对块和区间 $[1,\sqrt{n}]$ 中的素数都要进行除法，而对于较小的块来说，这种情况要糟糕得多。因此，在选择常数 S 时要保持平衡。

块大小 S 取 10^4 到 10^5 之间，可以获得最佳的速度。

线性筛法

埃氏筛法仍有优化空间，它会将一个合数重复多次标记。有没有什么办法省掉无意义的步骤呢？答案是肯定的。

如果能让每个合数都只被标记一次，那么时间复杂度就可以降到 $O(n)$ 了。

实现

C++

```cpp
vector<int> pri;
bool not_prime[N];

void pre(int n) {
  for (int i = 2; i <= n; ++i) {
    if (!not_prime[i]) {
      pri.push_back(i);
    }
    for (int pri_j : pri) {
      if (i * pri_j > n) break;
      not_prime[i * pri_j] = true;
      if (i % pri_j == 0) {
        // i % pri_j == 0
        // 换言之，i 之前被 pri_j 筛过了
        // 由于 pri 里面质数是从小到大的，所以 i 乘上其他的质数的结果一定会被
        // pri_j 的倍数筛掉，就不需要在这里先筛一次，所以这里直接 break
        // 掉就好了
        break;
      }
    }
  }
}
```

Python

```python
pri = []
not_prime = [False] * N


def pre(n):
    for i in range(2, n + 1):
        if not not_prime[i]:
            pri.append(i)
        for pri_j in pri:
            if i * pri_j > n:
                break
            not_prime[i * pri_j] = True
            if i % pri_j == 0:
                """
                i % pri_j == 0
                换言之，i 之前被 pri_j 筛过了
                由于 pri 里面质数是从小到大的，所以 i 乘上其他的质数的结果一定会被
                pri_j 的倍数筛掉，就不需要在这里先筛一次，所以这里直接 break
```

```
            掉就好了
            """
            break
```

上面的这种 **线性筛法** 也称为 **Euler 筛法**（欧拉筛法）。

> **Note**
>
> 注意到筛法求素数的同时也得到了每个数的最小质因子。

筛法求欧拉函数

注意到在线性筛中，每一个合数都是被最小的质因子筛掉。比如设 p_1 是 n 的最小质因子，$n' = \frac{n}{p_1}$，那么线性筛的过程中 n 通过 $n' \times p_1$ 筛掉。

观察线性筛的过程，我们还需要处理两个部分，下面对 $n' \bmod p_1$ 分情况讨论。

如果 $n' \bmod p_1 = 0$，那么 n' 包含了 n 的所有质因子。

$$
\begin{aligned}
\varphi(n) &= n \times \prod_{i=1}^{s}\frac{p_i - 1}{p_i} \\
&= p_1 \times n' \times \prod_{i=1}^{s}\frac{p_i - 1}{p_i} \\
&= p_1 \times \varphi(n')
\end{aligned}
$$

那如果 $n' \bmod p_1 \neq 0$ 呢，这时 n' 和 p_1 是互质的，根据欧拉函数性质，我们有：

$$
\begin{aligned}
\varphi(n) &= \varphi(p_1) \times \varphi(n') \\
&= (p_1 - 1) \times \varphi(n')
\end{aligned}
$$

实现

C++

```cpp
vector<int> pri;
bool not_prime[N];
int phi[N];

void pre(int n) {
  phi[1] = 1;
  for (int i = 2; i <= n; i++) {
    if (!not_prime[i]) {
      pri.push_back(i);
      phi[i] = i - 1;
```

```
    }
    for (int pri_j : pri) {
      if (i * pri_j > n) break;
      not_prime[i * pri_j] = true;
      if (i % pri_j == 0) {
        phi[i * pri_j] = phi[i] * pri_j;
        break;
      }
      phi[i * pri_j] = phi[i] * phi[pri_j];
    }
  }
}
```

Python

```
pri = []
not_prime = [False] * N
phi = [0] * N


def pre(n):
    phi[1] = 1
    for i in range(2, n + 1):
        if not not_prime[i]:
            pri.append(i)
            phi[i] = i - 1
        for pri_j in pri:
            if i * pri_j > n:
                break
            not_prime[i * pri_j] = True
            if i % pri_j == 0:
                phi[i * pri_j] = phi[i] * pri_j
                break
            phi[i * pri_j] = phi[i] * phi[pri_j]
```

筛法求莫比乌斯函数

定义

根据莫比乌斯函数的定义，设 n 是一个合数，p_1 是 n 的最小质因子，$n' = \frac{n}{p_1}$，有：

$$\mu(n) = \begin{cases} 0 & n' \bmod p_1 = 0 \\ -\mu(n') & \text{otherwise} \end{cases}$$

若 n 是质数，有 $\mu(n) = -1$。

实现

C++

```cpp
vector<int> pri;
bool not_prime[N];
int mu[N];

void pre(int n) {
  mu[1] = 1;
  for (int i = 2; i <= n; ++i) {
    if (!not_prime[i]) {
      mu[i] = -1;
      pri.push_back(i);
    }
    for (int pri_j : pri) {
      if (i * pri_j > n) break;
      not_prime[i * pri_j] = true;
      if (i % pri_j == 0) {
        mu[i * pri_j] = 0;
        break;
      }
      mu[i * pri_j] = -mu[i];
    }
  }
}
```

Python

```python
pri = []
not_prime = [False] * N
mu = [0] * N


def pre(n):
    mu[1] = 1
    for i in range(2, n + 1):
        if not not_prime[i]:
            pri.append(i)
            mu[i] = -1
        for pri_j in pri:
            if i * pri_j > n:
                break
            not_prime[i * pri_j] = True
            if i % pri_j == 0:
                mu[i * pri_j] = 0
                break
            mu[i * pri_j] = -mu[i]
```

筛法求约数个数

用 d_i 表示 i 的约数个数，num_i 表示 i 的最小质因子出现次数。

约数个数定理

定理：若 $n=\prod_{i=1}^m p_i^{c_i}$ 则 $d_i=\prod_{i=1}^m (c_i+1)$。

证明：我们知道 $p_i^{c_i}$ 的约数有 $p_i^0, p_i^1, \ldots, p_i^{c_i}$ 共 $c_i + 1$ 个，根据乘法原理，n 的约数个数就是 $\prod_{i=1}^{m}(c_i + 1)$。

实现

因为 d_i 是积性函数，所以可以使用线性筛。

在这里简单介绍一下线性筛实现原理。

1. 当 i 为质数时，$num_i \leftarrow 1, d_i \leftarrow 2$，同时设 $q = \left\lfloor \frac{i}{p} \right\rfloor$，其中 p 为 i 的最小质因子。
2. 当 p 为 q 的质因子时，$num_i \leftarrow num_q + 1, d_i \leftarrow \frac{d_q}{num_i} \times (num_i + 1)$。
3. 当 p, q 互质时，$num_i \leftarrow 1, d_i \leftarrow d_q \times (num_i + 1)$。

C++

```cpp
vector<int> pri;
bool not_prime[N];
int d[N], num[N];

void pre(int n) {
  d[1] = 1;
  for (int i = 2; i <= n; ++i) {
    if (!not_prime[i]) {
      pri.push_back(i);
      d[i] = 2;
      num[i] = 1;
    }
    for (int pri_j : pri) {
      if (i * pri_j > n) break;
      not_prime[i * pri_j] = true;
      if (i % pri_j == 0) {
        num[i * pri_j] = num[i] + 1;
        d[i * pri_j] = d[i] / num[i * pri_j] * (num[i * pri_j] + 1);
        break;
      }
      num[i * pri_j] = 1;
      d[i * pri_j] = d[i] * 2;
    }
  }
}
```

Python

```python
pri = []
not_prime = [False] * N
d = [0] * N
num = [0] * N


def pre(n):
```

```python
    d[1] = 1
    for i in range(2, n + 1):
        if not not_prime[i]:
            pri.append(i)
            d[i] = 2
            num[i] = 1
        for pri_j in pri:
            if i * pri_j > n:
                break
            not_prime[i * pri_j] = True
            if i % pri_j == 0:
                num[i * pri_j] = num[i] + 1
                d[i * pri_j] = d[i] // num[i * pri_j] * (num[i * pri_j] + 1)
                break
            num[i * pri_j] = 1
            d[i * pri_j] = d[i] * 2
```

筛法求约数和

f_i 表示 i 的约数和，g_i 表示 i 的最小质因子的 $p^0 + p^1 + p^2 + \dots p^k$.

实现

C++

```cpp
vector<int> pri;
bool not_prime[N];
int g[N], f[N];

void pre(int n) {
  g[1] = f[1] = 1;
  for (int i = 2; i <= n; ++i) {
    if (!not_prime[i]) {
      pri.push_back(i);
      g[i] = i + 1;
      f[i] = i + 1;
    }
    for (int pri_j : pri) {
      if (i * pri_j > n) break;
      not_prime[i * pri_j] = true;
      if (i % pri_j == 0) {
        g[i * pri_j] = g[i] * pri_j + 1;
        f[i * pri_j] = f[i] / g[i] * g[i * pri_j];
        break;
      }
      f[i * pri_j] = f[i] * f[pri_j];
      g[i * pri_j] = 1 + pri_j;
    }
  }
}
```

Python

```python
pri = []
not_prime = [False] * N
f = [0] * N
g = [0] * N


def pre(n):
    g[1] = f[1] = 1
    for i in range(2, n + 1):
        if not not_prime[i]:
            pri.append(i)
            g[i] = i + 1
            f[i] = i + 1
        for pri_j in pri:
            if i * pri_j > n:
                break
            not_prime[i * pri_j] = True
            if i % pri_j == 0:
                g[i * pri_j] = g[i] * pri_j + 1
                f[i * pri_j] = f[i] // g[i] * g[i * pri_j]
                break
            f[i * pri_j] = f[i] * f[pri_j]
            g[i * pri_j] = 1 + pri_j
```

一般的积性函数

假如一个 积性函数 f 满足：对于任意质数 p 和正整数 k，可以在关于 k 的低次多项式时间内计算 $f(p^k)$，那么可以在 $O(n)$ 时间内筛出 $f(1), f(2), \dots, f(n)$ 的值。

设合数 n 的质因子分解是 $\prod_{i=1}^{k} p_i^{\alpha_i}$，其中 $p_1 < p_2 < \dots < p_k$ 为质数，我们在线性筛中记录 $g_n = p_1^{\alpha_1}$，假如 n 被 $x \cdot p$ 筛掉（p 是质数），那么 g 满足如下递推式：

$$g_n = \begin{cases} g_x \cdot p & x \bmod p = 0 \\ p & \text{otherwise} \end{cases}$$

假如 $n = g_n$，说明 n 就是某个质数的次幂，可以 $O(1)$ 计算 $f(n)$；否则，$f(n) = f(\frac{n}{g_n}) \cdot f(g_n)$。

本节部分内容译自博文 Решето Эратосфена 与其英文翻译版 Sieve of Eratosthenes。其中俄文版版权协议为 Public Domain + Leave a Link；英文版版权协议为 CC-BY-SA 4.0。

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：Ir1d, StudyingFather, Enter-tainer, Tiphereth-A, LJFYC007, Xeonacid, H-J-Granger, iamtwz, mgt, shuzhouliu, CCXXXI, countercurrent-time, NachtgeistW, Early0v0, HeRaNO, MegaOwler, Peanut-Tang, YOYO-UIAT, AngelKitty, cjsoft, diauweb, ezoixx130,

GekkaSaori, Great-designer, greyqz, Konano, LovelyBuggies, Makkiy, minghu6, Mr-Python-in-China, P-Y-Y, PotassiumWings, SamZhangQingChuan, sshwy, Suyun514, TravorLZH, weilycoder, weiyong1024, 1804040636, 383494, aofall, CoelacanthusHex, cubeheadsun, frank-xjh, GavinZhengOI, Gesrua, hqztrue, ImpleLee, inkydragon, ksyx, kxccc, luojiny1, Lutra-Fs, lychees, Marcythm, Menci, opsiff, partychicken, PerfectPan, Persdre, shawlleyw StableAgOH, Steaunk, SukkaW, sunruisjtu2020, TianKong-y, TrisolarisHD, untitledunrevis WAAutoMaton, WineChord, wkywkyQAQ, wood3, YanWQ-monad, Yisheng Gong, zhouyuyang2002, ZnPdCo, 代建杉

© 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