[toc]

素数筛法

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

考虑这样一件事情：如果 x 是合数，那么 x 的倍数也一定是合数。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。

```
int Eratosthenes(int n) {
  int p = 0;
  for (int i = 0; i <= n; ++i) is_prime[i] = 1;
  is_prime[0] = is_prime[1] = 0;
  for (int i = 2; i <= n; ++i) {
    if (is_prime[i]) {
      prime[p++] = i;  // prime[p]是i,后置自增运算代表当前素数数量
      if ((long long)i * i <= n)
        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] = 0;  // 是i的倍数的均不是素数
    }
  }
  return p;
}
```

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 $O(n\log\log n)$。

怎么证明这个复杂度呢？我们先列出复杂度的数学表达式。

发现数学表达式显然就是素数的倒数和乘上 n，即 $n\sum_p {\frac{1}{p}}$。

我们相当于要证明 $\sum_p {\frac{1}{p}}$ 是 $O(\log\log n)$ 的。我们考虑一个很巧妙的构造来证明这个式子是 $O(\log\log n)$ 的：

证明：

注意到调和级数 $\sum_n {\frac{1}{n}}=\ln n$。

而又由唯一分解定理可得：$\sum_n {\frac{1}{n}}=\prod_p {(1+\frac{1}{p}+\frac{1}{p^2}+\cdots)}=\prod_p {\frac{p}{p-1}}$。

我们两边同时取 $\ln$，得：

$$ \begin{aligned} \ln \sum_n {\frac{1}{n}}&=\ln \prod_p {\frac{p}{p-1}}\ \ln\ln n&=\sum_p {(\ln p-\ln {(p-1)})} \end{aligned} $$

又发现 $\int {\frac{1}{x}dx}=\ln x$，所以由微积分基本定理：

$$ \sum_p {(\ln p-\ln {(p-1)})}=\sum_p {\int_{p-1}^p {\frac{1}{x}dx}} $$

画图可以发现，$\int_{p-1}^p {\frac{1}{x}dx}>\frac{1}{p}$，所以：

$$ \ln\ln n=\sum_p {\int_{p-1}^p {\frac{1}{x}dx}}>\sum_p {\frac{1}{p}} $$

所以 $\sum_p {\frac{1}{p}}$ 是 $O(\log\log n)$ 的，所以 **Eratosthenes 筛法** 的复杂度是 $O(n\log\log n)$ 的。

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。

```
int n;
vector<char> is_prime(n + 1, true);
is_prime[0] = is_prime[1] = false;
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;
  }
}
```

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

我们注意到筛法只需要 n 比特的内存。因此我们可以通过将变量声明为布尔类型，只申请 n 比特而不是 n 字节的内存，来显著的减少内存占用。即仅占用 $\dfrac n 8$ 字节的内存。

但是，这种称为 **位级压缩** 的方法会使这些位的操作复杂化。任何位上的读写操作都需要多次算术运算，最终会使算法变慢。

因此，这种方法只有在 n 特别大，以至于我们不能再分配内存时才合理。在这种情况下，我们将牺牲效率，通过显著降低算法速度以节省内存（减小 8 倍）。

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<>`。

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($k = 0 ... \lfloor {\frac n s} \rfloor$) 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。

```
int count_primes(int n) {
  const 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;
}
```

分块筛分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。

```
void init() {
  phi[1] = 1;
  for (int i = 2; i < MAXN; ++i) {
    if (!vis[i]) {
      phi[i] = i - 1;
      pri[cnt++] = i;
    }
    for (int j = 0; j < cnt; ++j) {
      if (1ll * i * pri[j] >= MAXN) break;
      vis[i * pri[j]] = 1;
      if (i % pri[j]) {
        phi[i * pri[j]] = phi[i] * (pri[j] - 1);
      } else {
        // 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也是
        // pri[j] 的倍数 它们都被筛过了，就不需要再筛了，所以这里直接 break
        // 掉就好了
        phi[i * pri[j]] = phi[i] * pri[j];
        break;
      }
    }
  }
}
```

上面代码中的 phi 数组，会在下面提到。

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：

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

```
void phi_table(int n, int* phi) {
  for (int i = 2; i <= n; i++) phi[i] = 0;
  phi[1] = 1;
  for (int i = 2; i <= n; i++)
    if (!phi[i])
      for (int j = i; j <= n; j += i) {
        if (!phi[j]) phi[j] = j;
        phi[j] = phi[j] / i * (i - 1);
      }
}
```