多重オイラー関数の計算について

梶田光

2025/09/13

1. はじめに

オイラー関数の計算は前回の研究で, ルックアップテーブル(LUT)を生成することによって高速に探索できることを示した.

一方, メモリマップトファイルは大きな領域の中のランダムアクセスに弱いため, アクセス箇所が様々な場所に飛ぶ多重オイラー関数の計算には不向きである.

そこで, 今回はそのような問題を解決する, 高速に探索可能なアルゴリズムを考えた.

2. 問題の定式化

n について, $k+1$ 個の数からなる組 $\left(n, \varphi(n), ..., \varphi^k(n)\right)$ を φ^k-set とよぶ.

この組に対して, $f : \varphi^k\text{-set} \to \{0,1\}$ を計算する関数があるとする. この計算量は実用上 $\Theta(1)$ とする.

そして, $1 \leq n \leq N$ の範囲で, $f\left(n, \varphi(n), ..., \varphi^k(n)\right) = 1$ を満たす n とその素因数分解を列挙する問題を φ^k-problem とよぶ.

> **例 2.1**: $n - 2\varphi^2(n) = 1$ を満たす $1 \leq n \leq 10^6$ を計算する問題に対しては, $N = 10^6$,
> f を $n - 2\varphi^2(n) = 1$ なら 1, そうでなければ 0 を返す関数とすればよい.

以降, N は 10^{10} から 10^{18} ほどのオーダーであると仮定する.

このとき, 一般的なコンピュータでは長さ N の配列が RAM に収まらないという点は特筆すべきである.

また, $\varphi^i(n) = 1$ となる最小の i を $R(n)$ と書くと, S. S. Pillai [1] より $R(n) \leq 1 + \log_2 n$ である.

また, $\varphi(1) = 1$ より, $R(n)$ 以上のすべての i について $\varphi^i(n) = 1$ が成り立つので, それより大きい k の方程式について考えることに意味はない.

したがって, k は最大でも $1 + \log_2 N$ ほどであると考えてよい.

さて, この問題を考えるときに, メモリマップドファイルを使って計算を高速化することを考える.

つまり, 前もって $1 \leq n \leq N$ のすべての n に対し, n の素因数分解と $\varphi(n), ..., \varphi^k(n)$ を記録しておくことで, 個別の φ^k-problem はその参照のみで解くことができるようにする.

以降, このファイルに記録するプロセスをビルド, 個別の φ^k-problem をファイルを読み込むことで解くプロセスをロードと呼ぶことにする.

また, 異なる N で実験したい場合があるため, ファイルの中の区間を参照するだけでよいように作成したメモリマップドファイルの中で n は順番に並んでいることが望ましい.

そして, ロード時には $[1, N]$ を RAM に収まる程度の長さに分割し, それぞれの区間について順にファイルの対応する区間をメモリにマップすることになる.

本論文の中では N は常に固定されているものとする.

3. φ^1-problem について

これは単純に $n, \varphi(n)$ の組と素因数分解を列挙すればよい.

いま N が大きいことを考えて, 区間篩を利用する.

まず, $\varphi(n)$ の値だけを求めることを考えると, $\varphi(n) = n \prod_{p|n} \left(1 - \frac{1}{p}\right)$ の公式を利用して, 以下のようにオイラー関数を計算すればよい.

function *segmented-sieve*(start, end, smallprimes, f):

  totients ← [start, start+1, ..., end]

  prime-left ← [start, start+1, ..., end]

  for p **in** smallprimes:

    for all m, **s.t.** start $\leq m \leq$ end **and** $p \mid m$:

      $\text{totients}[m - \text{start}] \leftarrow \text{totients}[m - \text{start}] \cdot \left(1 - \frac{1}{p}\right)$

      while $p \mid \text{prime-left}[m - \text{start}]$: $\text{prime-left}[m - \text{start}] \leftarrow \dfrac{\text{prime-left}[m - \text{start}]}{p}$

  for n **in** start...end:

    $p \leftarrow \text{prime-left}[n - \text{start}]$

    if $p > 1$: $\text{totients}[n - \text{start}] \leftarrow \text{totients}[n - \text{start}] \cdot \left(1 - \frac{1}{p}\right)$

    Write $\varphi(n)$ as $\text{totients}[n - \text{start}]$

smallprimes ← $\left(\text{list of primes} \leq \sqrt{N}\right)$ // *Use the normal sieve of Eratosthenes*

for (start, end) **in** $(1, \text{chunk-size}), (\text{chunk-size} + 1, 2 \cdot \text{chunk-size}), ..., (..., N)$:

  segmented-sieve(start, end, smallprimes)

ここで, chunk-size はその長さを持つ配列が RAM に収まるように設定する ($10^8, 10^9$ など.)

smallprimes は $\sqrt{N}$ 以下の素数が含まれているから, prime-left は 1 もしくは $\sqrt{N}$ より大きい素数である.

よって各 n について, $\text{totients}[n - \text{start}] = n \prod_{p|n} \left(1 - \frac{1}{p}\right)$ が成り立つ.

以降, 議論が煩雑になるのを防ぐため, 区間篩内部でのアルゴリズムだけ示し, またそこで配列は常に start だけオフセットされているものとする.

さて, 次に素因数分解を取得することについて考える.

通常エラトステネスの篩では各 n ごとに最小素因数 spf[n] を記録し, 以下のアルゴリズムによって n のすべての素因数を取得する.

n_temp ← n

factors ← []

while n_temp $>$ 1:

$p \leftarrow$ spf[n_temp]

factors $\leftarrow$ [...factors, p]

n_temp $\leftarrow$ n_temp$/n$

一方, この方法はそのままでは今回のメモリマップドファイルに素因数分解を記録し使い回す目的に応用できない. 以下, 理由を説明する.

まず, spf 配列はビルドのときに簡単にわかる. 区間篩の中で, 各区間 $[\mathrm{start}, \mathrm{end}]$ 内の任意の n について, 最初に割り切れた素数 $p \leq \sqrt{N}$ を記録していく.

n が素数であったら, spf$[n]$ に n を代入しておけばよい.

問題はそこから n のすべての素因数をどのように取得するかということである.

まず, ビルド時に spf だけを記録しておき, ロードのときにそれを利用して n の素因数分解を取得する方法はうまくいかない.

これは, 各 n について, 上のアルゴリズムでの n_temp は非常に不規則に, 不連続的に動くからである.

上のアルゴリズムではそのように動く n_temp の spf を順に追っていかなければならず, とても大きい範囲のマップを作るか頻繁にマップする区間を切り替えなければならない.

したがって, たとえばある正整数 n の素因数分解を知りたい場合に, メモリマップドファイルでマッピングしている(n を含む)区間外に, n_temp が飛ぶ可能性があり, 結局 N ほどの長さの区間をまとめてマッピングするか, マップする区間を頻繁に切り替えなければならなくなってしまうからである.

もう少しよい方法としては, ビルド中に区間篩を走らせるとき, n のすべての素因数がわかるのだから, (辞書型を用いて $n = p_1^{e_1} p_2^{e_2} ... p_k^{e_k}$ に現れる (p_i, e_i) の組を保存するなどして) n の素因数分解をすべてメモリマップドファイルに記録する方法である.

しかしこの方法では, 各 n について素因数は複数個あることがあるので, 結果的にメモリマップドファイルが大きくなってしまう.

これを解決したのが前回証明した定理と, それに依存するアルゴリズムである.

定理 3.1: $n \leq N$ とし, n の素因数分解を $n = p_0 p_1 ... p_{m-1} (p_i : \mathrm{prime}, p_i \leq p_{i+1})$ と書く.
i_0 を $0 \leq i \leq m$ かつ $\prod_{0 \leq j < i} p_j \leq \sqrt{N}$ を満たす最大の i, i_1 を $i_0 \leq i \leq m$ かつ $\prod_{i_0 \leq j < i} p_j \leq \sqrt{N}$ を満たす最大の i として定義し, $f_0 = \prod_{0 \leq j < i_0} p_j, f_1 = \prod_{i_0 \leq j < i_1} p_j, f_2 = \frac{n}{f_0 f_1}$ とおく.

すると, f_2 は必ず 1 もしくは素数.

なお, n や N の値によっては上記の積は空積となることもあるが, 慣習にしたがってその場合は 1 とする.

下はこの定理の, 前回の証明を改良した証明である.

Proof: 背理法で示す; f_2 が合成数であると仮定する.

すると f_2 は二つの素数の積で割り切ることができる. したがって, $i_1 \leq m - 2, p_{m-2} p_{m-1} \mid f_2$.

さて, $f_0 f_1 p_{m-2} p_{m-1} \leq f_0 f_1 f_2 = n \leq N$ が成り立つ.

ところで, $i_1 \leq m - 2$ から $i_0 \leq i_1 \leq m - 2$. したがって, p_{i_0}, p_{i_1} が存在する.

そして, i_0, i_1 の定義より, $f_0 p_{i_0} > \sqrt{N}, f_1 p_{i_1} > \sqrt{N}$.

しかし, $p_{i_0} \leq p_{m-2}, p_{i_1} \leq p_{m-2} \leq p_{m-1}$ より, $f_0 p_{m-2} > \sqrt{N}, f_1 p_{m-1} > \sqrt{N}$.

両辺を掛けると $f_0 f_1 p_{m-2} p_{m-1} > N$ が得られるが, これは先程示した不等式に矛盾する. ■

さて, これをアルゴリズムに組み込むことを考える. 区間篩の部分は以下のようになる.

totients ← [start, start+1, ..., end]

$f_0 \leftarrow [1, 1, ..., 1]$ // Length : end − start + 1

$f_1 \leftarrow [1, 1, ..., 1]$ // Length : end − start + 1

for p **in** smallprimes:

  for all m, **s.t.** start $\leq m \leq$ end **and** $p \mid m$:

    totients$[m]$ ← totients$[m] \cdot \left(1 - \dfrac{1}{p}\right)$

    temp_m ← m

    while $p \mid$ temp_m:

      temp_m ← $\dfrac{\text{temp\_m}}{p}$

      if $f_0[m] \cdot p \leq \sqrt{N}$: $f_0[m] \leftarrow f_0[m] \cdot p$

      else if $f_1[m] \cdot p \leq \sqrt{N}$: $f_1[m] \leftarrow f_1[m] \cdot p$

for n **in** start...end:

  $p \leftarrow$ prime-left$[n]$

  if $p > 1$: totients$[n]$ ← totients$[n] \cdot \left(1 - \dfrac{1}{p}\right)$

  Write n, totients$[n]$, $f_0[n], f_1[n]$ to memory-mapped file

ロード時には, $\sqrt{N}$ までの素因数分解を計算しておき (これは前計算でもファイルを利用した読み込みでもよい, $\sqrt{N}$ は小さいのでボトルネックにはならない), 各区間中の n について $f_0[n], f_1[n]$ の素因数分解を取得し, 素因数分解が自明な f_2 と合成すればよい.

このテクニックを部分分解と呼ぶことにする.

4. φ^2-problem について

$n, \varphi(n), \varphi^2(n)$ と n の素因数分解を列挙する問題である.

ここで重要になるのは, $\varphi^2(n)$ が $\varphi(n)$ と本質的に異なる点がいくつかあることである.

例えば, 乗法性 $\gcd(n, m) = 1 \Longrightarrow \varphi(nm) = \varphi(n)\varphi(m)$ は φ^2 では成り立たないし, $\varphi^2(n) = n \left\{ \prod_{p|n} \left(1 - \frac{1}{p}\right) \right\} \left\{ \prod_{p|\varphi(n)} \left(1 - \frac{1}{p}\right) \right\}$ を考えたとき, $\varphi^2(n)$ と $\varphi^2(pn)$ を比較すると因数には $\varphi(n)$ の素因数分解が関わるため非常に複雑である.

一方, 計算するという目的の上であれば, 先ほどの f_0, f_1, f_2 に分解するというテクニックを利用することで $\varphi^2(n)$ を効率的に計算することが可能である. 以下, その方法について議論する.

まず, n, m が互いに素であったとしても, $\varphi(n), \varphi(m)$ が互いに素であるとは限らないので, 互いに素な自然数の積で書くことは難しいように見える.

したがって, まず一般に互いに素とは限らない自然数の積のオイラー関数について考える.

下はよく知られた性質である.

定理 4.1: 任意の n, m に対し, $d = \gcd(n, m)$ とおくと $\varphi(nm) = \varphi(n)\varphi(m)\dfrac{d}{\varphi(d)}$.

さて, 今 $d \mid m$ より, $\varphi(d) \mid \varphi(m)$.

さらに, $\varphi(m)\dfrac{d}{\varphi(d)} = m\left\{\prod_{p|m}\left(1-\dfrac{1}{p}\right)\right\}\left\{\prod_{p|d}\left(1-\dfrac{1}{p}\right)\right\}^{-1} = m\prod_{p|m, p\nmid d}\left(1-\dfrac{1}{p}\right) \leq m$.

つまり, $\varphi(nm)$ は, それぞれ n と m 以下の整数の積に書くことができる.

したがって, 以下のような関数を考えることができる.

function *totient-product*(totients, $[n_1, ..., n_k]$):

if $k = 1$:

return [totients$[n_1]$]

else:

$d \leftarrow \gcd(n_1 ... n_{k-1}, n_k)$

return $\left[\textit{...totient-product}(\text{totients}, [n_1, ..., n_{k-1}]), \varphi(n_k)\dfrac{d}{\varphi(d)}\right]$

ここで totients は $\sqrt{N}$ 以下の n についてオイラー関数の値が入るリストで, $n_1, ..., n_k \leq \sqrt{N}$ とする.

今, *totient-product*(totients, $[n_1, ..., n_k]$) $= [n'_1, ..., n'_k]$ と書く.

いくつか例を挙げると,

- $k = 1 \rightarrow n'_1 = \varphi(n_1)$
- $k = 2 \rightarrow n'_1 = \varphi(n_1), n'_2 = \varphi(n_2)\dfrac{\gcd(n_1, n_2)}{\varphi(\gcd(n_1, n_2))}$
- $k = 3 \rightarrow n'_1 = \varphi(n_1), n'_2 = \varphi(n_2)\dfrac{\gcd(n_1, n_2)}{\varphi(\gcd(n_1, n_2))}, n'_3 = \varphi(n_3)\dfrac{\gcd(n_1 n_2, n_3)}{\varphi(\gcd(n_1 n_2, n_3))}$

など.

ここで, 一般の k について $\varphi(n_1 ... n_k) = n'_1 ... n'_k$ が成り立つ.

さらに, $n'_1 \leq n_1, ..., n'_k \leq n_k$ が成り立つため, 同じ totients の配列を使いまわしながらこの関数を
繰り返し適用することで $\varphi^2(n_1 ... n_k)$ や $\varphi^3(n_1 ... n_k)$ も計算することができる.

これによって, すべての $n \leq N$ について, (f_0, f_1 は定義上 $\sqrt{N}$ 以下であるから) $f_2 \leq \sqrt{N}$ が成り立つならば少しの totients の前計算と *totient-product* を利用することによって $\varphi^2(n)$ を計算することができる.

したがって, 問題は $f_2 > \sqrt{N}$ の場合である. (このとき定理から f_2 は素数である.)

$f_0 f_1$ を α と書くと, $n = \alpha f_2$ で, $f_2 > \sqrt{N} \geq \sqrt{n}$ より, n は $\sqrt{n}$ より大きい素因数を 1 つしか持てないことから $\gcd(\alpha, f_2) = 1$.

したがって $\varphi(n) = \varphi(\alpha)(f_2 - 1)$.

さて, ここから $\varphi^2(n)$ を計算するには $f_2 - 1$ の部分分解が必要である.

しかし, これは案外簡単に解決できる.

まず, $\alpha = 1$ の場合について考える.

このとき, $n = f_2$ であるから, $f_2 - 1 = n - 1$ は n の隣にある.

したがって, $\sqrt{N} < \text{start}$ を満たす区間について, start が常に非素数となるように調整すれば (これは例えば start が奇数ならば 1 減らすことで解決できる), 任意の $\sqrt{N}$ より大きい素数 n について $n-1$ の部分分解は簡単に手に入る.

次に, $\alpha > 1$ の場合について考える.

このとき, $f_2 = \dfrac{n}{\alpha}$ ということは, $\dfrac{\text{start}}{\alpha} \leq f_2 \leq \dfrac{\text{end}}{\alpha}$ が成り立つ.

また, $\alpha f_2 \leq N$ より, $\alpha \leq \sqrt{N}$.

ここで, 任意の $\alpha > 1, f_2 > \sqrt{N}, \text{start} \leq n \leq \text{end}$ を満たす n について, f_2 は $\left[\dfrac{\text{start}}{2}, \dfrac{\text{end}}{2}\right], \left[\dfrac{\text{start}}{3}, \dfrac{\text{end}}{3}\right], \left[\dfrac{\text{start}}{4}, \dfrac{\text{end}}{4}\right], \ldots, \left[\dfrac{\text{start}}{\sqrt{N}}, \dfrac{\text{end}}{\sqrt{N}}\right]$ の区間たちのいずれかに含まれている.

そして, この区間の長さを合計すると $(\text{end} - \text{start}) \log N$ ほどのオーダーになり, これは小さい.

したがって, $\varphi^2(n)$ も列挙する区間篩内部のアルゴリズムは以下のように書くことができる.

ただし, $\sqrt{N}$ 以下の n について $\varphi(n)$ の値を持つ totients は, 区間篩の前計算で smallprimes とともに計算しておく.

また長さ N の primechain という配列を初期化しておく. (これはメモリマップして利用する.)

primechain のそれぞれの要素は 2 つの 32 ビット符号なし整数を書くサイズがあり, 0 で初期化されているものとする.

(具体的には, f_0 と f_1 のペアを書き込み, $f_0, f_1 \leq \sqrt{N}$ で N は実用上 $2^{64} \sim 10^{19}$ 未満としてよいので f_0, f_1 は 32 ビットに収まることになる.)

low_start $\leftarrow$ start

if start : odd and start > 1: low_start $\leftarrow$ start $- 1$

$f_0 \leftarrow [1, 1, \ldots, 1]$ // Length : end $-$ low_start $+ 1$

$f_1 \leftarrow [1, 1, \ldots, 1]$ // Length : end $-$ low_start $+ 1$

Memory-map interval $\left[\left\lfloor\dfrac{\text{start}}{2}\right\rfloor, \left\lfloor\dfrac{\text{end}}{2}\right\rfloor\right], \left[\left\lfloor\dfrac{\text{start}}{3}\right\rfloor, \left\lfloor\dfrac{\text{end}}{3}\right\rfloor\right], \left[\left\lfloor\dfrac{\text{start}}{4}\right\rfloor, \left\lfloor\dfrac{\text{end}}{4}\right\rfloor\right], \ldots, \left[\left\lfloor\dfrac{\text{start}}{\sqrt{N}}\right\rfloor, \left\lfloor\dfrac{\text{end}}{\sqrt{N}}\right\rfloor\right]$ of primechain

for p **in** smallprimes:

  for all m, **s.t.** low_start $\leq m \leq$ end **and** $p \mid m$:

    m_temp $\leftarrow m$

    while $p \mid$ m_temp:

      m_temp $\leftarrow \dfrac{\text{m\_temp}}{p}$

      if $f_0[m] \cdot p \leq \sqrt{N} : f_0[m] \leftarrow f_0[m] \cdot p$

      else if $f_1[m] \cdot p \leq \sqrt{N} : f_1[m] \leftarrow f_1[m] \cdot p$

for n **in** start…end:

$f_2 \leftarrow \dfrac{n}{f_0[n]f_1[n]}$

if $f_2 \le \sqrt{N}$:

$[f_0', f_1', f_2'] \leftarrow \textit{totient-product}(\text{totients}, [f_0[n], f_1[n], f_2])$

$[f_0'', f_1'', f_2''] \leftarrow \textit{totient-product}(\text{totients}, [f_0', f_1', f_2'])$

Write $\varphi(n) = f_0'f_1'f_2', \varphi^2(n) = f_0''f_1''f_2''$ to file

else if $f_0[n] = f_1[n] = 1$: // n is a prime $> \sqrt{N}$

$[f_0', f_1', f_2'] \leftarrow \textit{totient-product}\left(\text{totients}, f_0[n-1], f_1[n-1], \dfrac{n-1}{f_0[n-1]f_1[n-1]}\right)$

Write $\varphi(n) = n-1, \varphi^2(n) = f_0'f_1'f_2'$ to file

Write tuple $f_0[n-1], f_1[n-1]$ to primechain$[n]$

else:

$\alpha \leftarrow f_0[n]f_1[n]$

$[f_0', f_1'] \leftarrow \text{primechain}[f_2]$

$f_2' \leftarrow \dfrac{f_2 - 1}{f_0'f_1'}$

Write $\varphi(n) = \text{totient}[\alpha](f_2 - 1)$

if $f_2' \le \sqrt{N}$:

Write $\varphi^2(n) = \textit{totient-product}(\text{totients}, [\text{totient}[\alpha], f_0', f_1', f_2')$

else:

Write $\varphi^2(n) = \textit{totient-product}(\text{totients}, [\text{totient}[\alpha], f_0', f_1')(f_2' - 1)$

5. φ^k-problem について

適当な区間 $[\text{start}, \text{end}]$ 内の正整数 n の $\varphi(n), \varphi^2(n), ..., \varphi^k(n)$ までを計算するには, 区間 $[\text{start}, \text{end}]$ を見ている時点でどのような情報が得られればよいかを考える.

まず, n の分解 $n = f_0f_1f_2$ を考える.

もし $f_2 \le \sqrt{N}$ であれば, $[f_0, f_1, f_2]$ に *totient-product* を繰り返し適用し続ければよい.

それ以外の場合, $\alpha = f_0f_1$ とおくと $\alpha = \dfrac{n}{f_2} < \dfrac{N}{\sqrt{N}} = \sqrt{N}$ で, $\varphi(n) = \varphi(\alpha f_2) = \varphi(\alpha)(f_2 - 1)$ である.

$\varphi^2(n)$ を計算するためには, $f_2 - 1$ の分解 $f_2 - 1 = f_0'f_1'f_2'$ が必要である.

もし $f_2' \le \sqrt{N}$ であれば, $\varphi(n) = \varphi(\alpha)f_0'f_1'f_2'$ は $\sqrt{N}$ 以下の正整数の積で書けるから *totient-product* を繰り返し適用すればよい.

それ以外の場合, $\alpha' = \varphi(\alpha)f_0'f_1'$ とおくと $k' = \dfrac{\varphi(n)}{f_2'} < \dfrac{n}{\sqrt{N}} \le \dfrac{N}{\sqrt{N}} = \sqrt{N}$ で, $\varphi^2(n) = \varphi(\alpha')(f_2' - 1)$.

$\varphi^3(n)$ を計算するためには, $f_2' - 1$ の分解 $f_2' - 1 = f_0''f_1''f_2''$ が必要である.

もし $f_2'' \le \sqrt{N}$ であれば, $\varphi^2(n) = \varphi(\alpha')f_0''f_1''f_2''$ は $\sqrt{N}$ 以下の正整数の積で書けるから *totient-product* を繰り返し適用すればよい.

それ以外の場合, $\alpha'' = \varphi(\alpha')f_0''f_1''f_2''$ とおくと $\alpha'' = \dfrac{\varphi^2(n)}{f_2''} < \dfrac{n}{\sqrt{N}} \le \dfrac{N}{\sqrt{N}} = \sqrt{N}$ で, $\varphi^3(n) = \varphi(\alpha'')(f_2'' - 1)$.

f''' を $f^{(3)}$, f'''' を $f^{(4)}$ などして表記すると, $\varphi^k(n)$ を計算するためには $f_0^{(k-1)}, f_1^{(k-1)}, f_2^{(k-1)}$ までが必要である.

さて, アルゴリズムの議論のために 2 つ命題を証明しておく.

補題 5.1: n, i を正整数とし, $f_2, f_2', f_2'', ..., f_2^{(i)}$ がすべて $\sqrt{N}$ より大きいとする.

$\beta^{(j)} := f_0^{(j)}f_1^{(j)}, B_j := \prod_{m=1}^{j} \beta^{(j)}$ とおくと $n = \beta\left(f_2^{(i)}B_i + \sum_{m=0}^{i-1} B_m\right)$.

なお B_0 は 1 と定義する.

Proof: i についての帰納法で証明する.

$i = 1$ の場合は $n = f_0f_1f_2 = \beta(1 + f_0'f_1'f_2') = \beta(f_2'\beta' + B_0) = \beta(f_2'B_1 + B_0)$ よりよい.

i の場合について命題が成り立っていると仮定すると,

$n = \beta\left(f_2^{(i)}B_i + \sum_{m=0}^{i-1} B_m\right) = \beta\left(\left(1 + \beta^{(i+1)}f_2^{(i+1)}\right)B_i + \sum_{m=0}^{i-1} B_m\right) = \beta\left(f_2^{(i+1)}B_{i+1} + \sum_{m=0}^{i} B_m\right)$ より $i+1$ の場合も成り立つ. ■

定理 5.1: n, i を正整数とし, $f_2, f_2', f_2'', ..., f_2^{(i)}$ をすべて $\sqrt{N}$ より大きい奇素数とする.

先の命題と同様に $\beta^{(i)}, B_i$ を定義すると $f_2^{(i)} = \left\lfloor \dfrac{n}{\beta B_i} \right\rfloor$.

Proof: $f_2^{(i)} = \left\lfloor \dfrac{n}{\beta B_i} \right\rfloor$ は $f_2^{(i)} = \left\lfloor \dfrac{f_2}{B_i} \right\rfloor$ と同値なので, これを証明する.

まず, 先の命題から $f_2 = f_2^{(i)}B_i + \sum_{m=0}^{i-1} B_m$ が得られた.

両辺を B_i で割って $\dfrac{f_2}{B_i} = f_2^{(i)} + \sum_{m=0}^{i-1} \dfrac{B_m}{B_i}$ を得る.

さて, 任意の $0 \le m < i$ について, $B_{m+1} = \beta^{(m+1)}B_m$ であるが, $f_2^{(m)} - 1 = \beta^{(m+1)}f_2^{(m+1)}$ で $f_2^{(m)}, f_2^{(m+1)}$ は奇素数なので $\beta^{(m+1)}$ は偶数で, これは 2 以上.

したがって任意の $0 \le m < i$ について $B_{m+1} \ge 2B_m$ である.

ここから $\dfrac{B_m}{B_i} = \dfrac{B_m}{B_{m+1}} \cdot \dfrac{B_{m+1}}{B_{m+2}} \cdot ... \cdot \dfrac{B_{i-1}}{B_i} \le \left(\dfrac{1}{2}\right)^{i-m}$ が従うので, $0 \le \sum_{m=0}^{i-1} \dfrac{B_m}{B_i} \le \sum_{m=0}^{i-1} \left(\dfrac{1}{2}\right)^{i-m} < 1$.

よって, $f_2^{(i)} \le \dfrac{f_2}{B_i} < f_2^{(i)} + 1$ から, $f_2^{(i)} = \left\lfloor \dfrac{f_2}{B_i} \right\rfloor$ が示された. ■

さて, この定理の重要な点は, どんなに大きい i を取ってきても, $f_2^{(i)}$ が $\sqrt{N}$ より大きければ, $f_2^{(i)}$ は必ず harmonic map の範囲に含まれているということである (このとき $\beta B_i \le \sqrt{N}$ は明らかであろう.)

したがって, $\varphi^k(n)$ までを計算するために必要な情報は, すべて *totient-product* と primechain の harmonic map から得られることになる.

これは GPT-5 との議論から発展した.

もともと筆者は $f_2^{(i)}$ は primechain の範囲外であると思い込んでおり, そのため最初に考えていたアルゴリズムは各 primechain の要素には長さ $k-1$ の配列を持たせておき, f_2 が区間に含まれているときに f_2' の primechain をコピーするという方法を取っていた.

そのため, 空間計算量は $O\left(k\sqrt{N}\log N\right)$ になっていた.

しかし, GPT-5 にそのアルゴリズムの概要を入力し, 発展のアイデアを出すよう指示したところ, GPT-5 は $f_2^{(i)}$ が常に primechain の範囲**内**にあると思い込んでおり (証明はなかったし, おそらく内部でも証明はしていないであろう), primechain の f_2 番目には f_0', f_1' のみを書き, primechain 全体を単連結リストの森のように扱うことを考案した.

筆者はこの議論から先の補題と定理に気づき, 現在のアルゴリズムを構築した.

以上を踏まえ, φ^k-problem のためのファイルをビルドするときの区間篩のアルゴリズムを考えると以下のようになる.

なお, primechain は φ^2-problem のときと同様, 長さ N の配列とする.

```
low_start ← start
if start : odd and start > 1: low_start ← start − 1
f_0 ← [1, 1, ..., 1]   // Length : end − low_start + 1
f_1 ← [1, 1, ..., 1]   // Length : end − low_start + 1
Memory-map interval [⌊start/2⌋, ⌊end/2⌋], [⌊start/3⌋, ⌊end/3⌋], [⌊start/4⌋, ⌊end/4⌋], ..., [⌊start/√N⌋, ⌊end/√N⌋] of
primechain
for p in smallprimes:
    for all m, s.t. low_start ≤ m ≤ end and p | m:
        m_temp ← m
        while p | m_temp:
            m_temp ← m_temp / p
            if f_0[m] · p ≤ √N : f_0[m] ← f_0[m] · p
            else if f_1[m] · p ≤ √N : f_1[m] ← f_1[m] · p
for n in start...end:
    f_2 ← n / (f_0[n] f_1[n])
    if f_2 ≤ √N:
        seq ← [f_0[n], f_1[n], f_2]
        for i in 1..k:
            seq ← totient_product(totients, seq)
            Write φ^i(n) = seq[0] · seq[1] · seq[2]
        continue
    if f_0[n] = f_1[n] = 1:    // n is a prime > √N
        primechain[n] ← [f_0[n − 1], f_1[n − 1]]
    α^(i−1) ← f_0[n] f_1[n]
    f_2^(i−1) ← f_2
    for i in 1...k:
```

27 Write $\varphi^i(n) = \mathrm{totients}\left[\alpha^{(i-1)}\right]\left(f_2^{(i-1)} - 1\right)$

28 **if** $i = k$: **break**

29 $\left[f_0^{(i)}, f_1^{(i)}\right] \leftarrow \mathrm{primechain}\left[f_2^{(i-1)}\right]$

30 $f_2^{(i)} \leftarrow \dfrac{f_2^{(i-1)} - 1}{f_0^{(i)} f_1^{(i)}}$

31 **if** $f_2^{(i)} \leq \sqrt{N}$:

32 $\quad \mathrm{seq} \leftarrow \left[\mathrm{totients}\left[\alpha^{(i-1)}\right], f_0^{(i)}, f_1^{(i)}, f_2^{(i)}\right]$

33 $\quad$ **for** j in $i+1 ... k$:

34 $\quad\quad \mathrm{seq} \leftarrow \textit{totient-product}(\mathrm{totients}, \mathrm{seq})$

35 $\quad\quad$ Write $\varphi^j(n) = \mathrm{seq}[0] \cdot \mathrm{seq}[1] \cdot \mathrm{seq}[2] \cdot \mathrm{seq}[3]$

36 $\quad$ **break**

37 $\alpha^{(i-1)} \leftarrow \mathrm{totients}\left[\alpha^{(i-1)}\right] f_0^{(i)} f_1^{(i)}$

38 $f_2^{(i-1)} \leftarrow f_2^{(i)}$

$k = O(\log N)$ に注意すると, 一つの区間について, 空間計算量は $O((\mathrm{end} - \mathrm{start}) k \log N)$, 時間計算量は $O((\mathrm{end} - \mathrm{start}) \log N)$.

全体のビルドのアルゴリズムについては, 区間の長さを $\sqrt{N}$ 程度とすると空間計算量が $O\left(\sqrt{N} \log N\right)$, 時間計算量が $O(kN \log N)$ (gcd の計算がボトルネック), 必要なディスクの容量は $O(kN)$.

なお, ルックアップテーブルを作らず, 単一の φ^k-problem のみを計算するアルゴリズムも考えられる.

その場合, 時間・空間計算量は変わらないが, ディスクに置くのが長さ N の primechain のみでよくなり, $O(N)$ で済む.

6. 各種最適化

以下では, 前章の最後に示したアルゴリズムについて議論する.

6.1. 並列化

このアルゴリズムは並列化も可能である.

並列化が効率に貢献する主な箇所は 2 箇所ある.

一つは区間 $[\mathrm{low\_start}, \mathrm{end}]$ 内の各 n について f_0, f_1 を計算する 6-12 行目の箇所である.

データの競合を防ぐため, m に関するループを並列化することが限界と思われる.

二つ目は $\varphi(n), \varphi^2(n), ..., \varphi^k(n)$ を計算する 13-39 行目の n に関するループ全体である.

注意しなければならないのは, このループ (**for** n in start...end:) 内では基本的に n に関するポインタのみ書き込み/読み込みを行うが, primechain では離れた場所のポインタの読み込みを行うのでデータ競合が発生しうるということである.

つまり, 30 行目で primechain から整数の組を読み込んでいるが, これは f_2 での primechain への書き込み (22-24 行目) が行われた後でなければならず, 愚直な並列化ではこれが成り立たない可能性がある.

したがって, 13-39 行目のループ全体を並列化する際には, start と end を調整して, 読み込みと書き込みの順番の整合性が取れるようにする必要がある.

具体的には, $[\text{start}, \text{end}]$ 内のすべての整数 n について, n に対応する $f_2 = \dfrac{n}{f_0[n]f_1[n]}$ が $\sqrt{N}$ を超えるとき, $\text{primechain}[f_2]$ は $[\text{start}, \text{end}]$ の区間の処理の前に計算されていなければならない.

これを解決する単純な方法は常に $\text{start} * 2 > \text{end}$ とすることで,

これは, 以下のように start と end を決めることで解決できる:

$\text{start} \leftarrow 0$

$\text{end} \leftarrow \left\lfloor \sqrt{N} \right\rfloor$

loop:

$\text{start} \leftarrow \text{end} + 1$

if $\text{start} > N$: **break**

$\text{end} \leftarrow \min(\text{end} + \text{chunk-size}, 2 * \text{end} + 1, N)$

$\textit{segmented-sieve}(\text{start}, \text{end}, \text{smallprimes})$

この $\text{end} \leftarrow \min(\text{end} + \text{chunk-size}, 2 * \text{end} + 1, N)$ が重要である.

この処理をする直前, start は $\text{end} + 1$ であるから, end が $2 * \text{end} + 1$ を超えないようにすることで $\text{start} * 2 > \text{end}$ が成り立つようにできる.

6.2. primechain の長さの削減

先のアルゴリズムで, 配列 primechain の添字には素数しか現れない.

したがって, 単純な効率化のアイデアとしては, primechain の 2 以上の偶数の部分を省くということである.

このとき, $\text{compress}(n) := \left\lceil \frac{n}{2} \right\rceil$ のように定義した関数を用いて, primechain の長さを $\text{compress}(N) + 1$ に設定, 区間 $[\text{start}, \text{end}]$ を計算するときのメモリマップする範囲を $\left[\left\lfloor \frac{\text{start}}{2} \right\rfloor, \left\lfloor \frac{\text{end}}{2} \right\rfloor\right], \left[\left\lfloor \frac{\text{start}}{3} \right\rfloor, \left\lfloor \frac{\text{end}}{3} \right\rfloor\right], ..., \left[\left\lfloor \frac{\text{start}}{\sqrt{N}} \right\rfloor, \left\lfloor \frac{\text{end}}{\sqrt{N}} \right\rfloor\right]$ から $\left[\left\lfloor \frac{\text{compress}(\text{start})}{2} \right\rfloor, \left\lfloor \frac{\text{compress}(\text{end})}{2} \right\rfloor\right], ..., \left[\left\lfloor \frac{\text{compress}(\text{start})}{\sqrt{N}} \right\rfloor, \left\lfloor \frac{\text{compress}(\text{end})}{\sqrt{N}} \right\rfloor\right]$ に変更し, すべてのアクセス $\text{primechain}[p]$ を $\text{primechain}[\text{compress}(p)]$ に置き換えることができる.

すると, primechain に必要な長さや空間計算量はほぼ半分に削減できる.

同様に 3 の倍数, 5 の倍数なども除くように, 添字を圧縮する compress 関数を定義することができる.

これをより一般に考え, wheel sieve のような考え方を利用する.

今, 素数を小さい方から順に m 個とって $p_0, p_1, ..., p_{m-1}$ とする.

そして m は $P = p_0 p_1 ... p_{m-1} \leq \sqrt{N}$ を満たす最大の m とおく.

$S = \{p_0, p_1, ..., p_{m-1}\} \cup \{n \mid n > p_{m-1}, \gcd(n, P) = 1\}$ とおけば, S は素数全体の集合を含む.

そして, $\text{compress}(n)$ は $\sharp\{p \in S \mid p \leq n\} - 1$ とおけばよい.

したがって, $\text{compress}(n)$ を計算するアルゴリズムは以下のようになる:

$C_{\text{small}} = [1, 1, ..., 1, 1] \quad // \text{ Length} : P + 1$

$C_{\text{large}} = [1, 1, ..., 1, 1] \quad // \text{ Length} : P + 1$

$C_{\text{small}}[0] \leftarrow 0; C_{\text{small}}[1] \leftarrow 0; C_{\text{small}}[P] \leftarrow 0$

$C_{\text{large}}[0] \leftarrow 0; C_{\text{large}}[P] \leftarrow 0$

for p in $[p_0, p_1, ..., p_{m-1}]$:

  $C_{\text{large}}[p] \leftarrow 0$

  for all j **s.t.** $2 * p \leq j < P, p \mid j$:

    $C_{\text{small}}[j] \leftarrow 0; C_{\text{large}}[j] \leftarrow 0$

for n in $0..P - 1$:

  $C_{\text{small}}[n+1] \leftarrow C_{\text{small}}[n] + C_{\text{small}}[n+1]$

  $C_{\text{large}}[n+1] \leftarrow C_{\text{large}}[n] + C_{\text{large}}[n+1]$

function compress(n):

  if $n \leq P$: **return** $C_{\text{small}}[n] - 1$

  else: **return** $C_{\text{small}}[P] + C_{\text{large}}[P] * \left(\left\lfloor \frac{n}{P} \right\rfloor - 1\right) + C_{\text{large}}[n \bmod P] - 1$

これを利用すると, 全体の空間計算量を $O\left(\frac{\sqrt{N}\log N}{\log\log N}\right)$, 必要なディスクの容量を $O\left(\frac{kN}{\log\log N}\right)$ に抑えられる.

参考文献

[1] S. S. Pillai, "On a function connected with ϕ (n)," *Bulletin of the American Mathematical Society*, vol. 35, no. 6, pp. 837–841, 1929.