「素因数分解問題に関する研究調査」
素因数分解問題調査報告書

～SQUFOF 法の実装報告～

２００３年2月１７日

日本電信電話株式会社

青木　和麻呂
植田　　広樹
内山　　成憲

「素因数分解問題に関する研究調査」
素因数分解問題調査報告書

～SQUFOF 法の実装報告～

NTT 情報流通プラットフォーム研究所

2003 年 2 月 17 日

概 要

16 ビット以下の素因子がない 60 ビット程度の数の分解には Shanks による SQUFOF 法が有力であると考えられてきた。本稿では、その SQUFOF 法を Pentium III および Pentium 4 で実装し、最適化方法について考察した。SQUFOF の入力についての考察、平方数判定および除算の高速化により平均 2.03 ミリ秒 (Pentium 4 2.0GHz Willamette) で 60 ビット程度の合成数が分解できるようになった。

1 条件

1.1 問題設定

次の条件の**合成数** N の因子を出来るだけ高速生成する。

- 62 ビット以下
- 16 ビット以下の素因子が存在しない
- 24 ビット以下の素因子があまりない

まずは UBASIC で約 50 ミリ秒/合成数で分解できたものを、半分程度の時間で分解できるようにすることを目標とした。また、低い確率であれば分解できなくとも構わないとの条件も付加した。

実装評価には、200 桁の特殊数体ふるい法分解実験 [木田 01] で実際に用いられた 4 large prime の候補となる 943 個の 2 素数の積からなる合成数を用い、その合成数の平均分解時間により行なった。入力に用いられた合成数の分布は次の通りである。

ビット長	個数	累積	累積の割合 (%)
57	146	146	15
58	159	305	32
59	140	445	47
60	168	613	65
61	165	778	83
62	165	943	100

理由は不明であるが、ランダムに生成した 62 ビットの数と違い、57 ビットから 62 ビットまでほぼ一様に分布している。また、入力の合成数の素因子のうち、小さい方の素因子の分布は次の通りである。

ビット長	個数	累積	累積の割合 (%)
18	6	6	1
19	6	12	1
20	15	27	3
21	18	45	5
22	47	92	10
23	71	163	17
24	131	294	31
25	116	410	43
26	117	527	56
27	109	636	67
28	135	771	82
29	104	875	93
30	53	928	98
31	15	943	100

1.2 計算機環境

実装実験に当たり次の計算機環境を用いた。

CPU1	Pentium III 700MHz (Coppermine)
CPU2	Pentium 4 2GHz (Willamette)
OS	FreeBSD-4.5R
コンパイラ 1	gcc version 2.95.3 20010315 (release) [FreeBSD]
コンパイラ 2	gcc version pgcc-2.95.2.1 20001224 (release)

CPU について 現在および近い将来手に入る CPU として、費用対効果がよい CPU として x86 アーキテクチャの CPU が優れていること、数体篩法において「篩」部分はメモリアクセス効率が良いことから、最終的には Pentium 4 (Northwood) が採用されると考えられることから、Pentium 4 をメインの CPU として採用した。しかしながら、Northwood の実験をする環境を持っている人が少ないことから Pentium 4 については、Willamette で対応した。

また、Pentium III については、筆者が精通していることや、まだ広く分布していると考えられることから基本的に Pentium III 上で最適化を行い、Pentium 4 は最終的に最適化されたプログラムの計測のみに用いた。本稿中で断りなく、Pentium III、Pentium 4 と表記した場合は上記 CPU を表す。

しかし、Pentium III と、Pentium 4 はアーキテクチャが大きく異なることから今後は Pentium 4 の特性をさらに考慮し、Pentium 4 用に最適化することが必要と考えられる。なお、実装実験中はもちろん他の動作周波数の CPU を用いることも当然行ない、クロックサイクル数で評価した。また、Athlon でも若干テストしたが、あまり一般的でないので途中から使用を停止したので細かい情報は不明である。

OS について Linux[†1] と比べて今回の計算については約 3%ほど速いということから、FreeBSD を採用した。実験中は他の 4.x 系列のバージョンも使った。

コンパイラについて 基本的に

```
pgcc -Wall -fomit-frame-pointer -O6 -lm
```

で実験した。最終的に出来上がったコードについては、さまざまなコンパイラといくつかのオプションの組合せで試されたもののうち、最も有効なものを採用した。実験途中は、参考までに icc[†2] でも実験した。多くの場合、上記コンパイラより 1 割程度速い数値が達成された。また、gcc のバージョン 3 系列については、まだ不安定なので、今回は試さなかった。

利用言語について 基本的に C 言語 (ANSI-C や ISO-C に限らない) で書き、部分的にアセンブラも利用した。アセンブラでは MMX, SSE, SSE2 の利用も行なった場合もある。

2 SQUFOF 法について

今回実装した SQUFOF 法は [Rie93, pp.190–193] にある Pascal プログラムをもとに作成した。SQUFOF 法の詳細については [Rie93] を参照されたい。

SQUFOF 法では (半) 連分数展開を繰り返し、平方数を探索することから

- 連分数回数の削減
- 連分数展開の高速化 (除算の高速化)
- 平方数判定の高速化

が最適化の重点となる。以下で、それぞれについての高速化について考察する。また、これらの処理はすべて 32 ビットに納まっていることに注意されたい。

[†1] Linux version 2.2.14 (root@neo), gcc version 2.95.2 20000116 (Debian GNU/Linux)

[†2] Intel(R) C++ Compiler for 32-bit applications, Version 5.0.1

3 SQUFOF 法の改良

文献 [Coh93, p.430] に「周期が短い N を避けるため、2 つの判別式、例えば $N \equiv 1 \pmod 4$ のときは N と $5N$ の場合を並列に計算するとよい」とある。今回は、

- 一つの合成数を高速に分解したいのではなく、多くの合成数を分解しなければならないことから、並列計算をするぐらいなら他の数の分解を行ないたいこと
- 入力 N が最大 62 ビットであり、SQUFOF 法の主繰り返し処理の最大値が $2\sqrt{N}$ であることから、たとえかける数が 2 であったとしても計算機での取扱が極めて繁雑になり効率が落ちること

から、この方法は一見採用する価値に乏しいようであるが、

- 62 ビットより小さな数の分解もそれなり行なう必要があること
- 小さな数を、分解したい合成数にかけることにより周期が変化すること

から、小さな数をかけることによりどの程度周期が変化するのかを実験により確認した。

かける数	対象となる合成数の個数	first phase の平均連分数回数	とその標準偏差	成功率 (%)
1	943	50522	54807	100
2	778	52793	56111	100
3	678	25297	25821	64
4	613	23358	23206	65
5	568	24483	24829	69
6	521	27902	29085	67
7	483	26922	27665	68
8	445	18760	18467	61
9	420	19444	21991	63
10	401	25242	25073	66
11	376	23083	23712	66
12	358	17152	16919	55
13	343	24355	21081	66
14	333	29414	28821	64
15	323	15448	15682	60
16	305	14676	15380	57
17	290	27439	25791	65
18	279	21776	18419	64
19	271	23790	24799	68
20	254	14941	13096	57
21	240	15623	14131	66

表を観察すると大雑把にいって

- 成功率は約 $\frac{2}{3}$ に低下[†3]
- 平均連分数回数は約半分

となることが分かる。従って、小さい数 (3) をかけたものに対し SQUFOF を実行し、失敗したものについてもとの SQUFOF を実行すると全体としては、3 をかけられるものが全体の約 $\frac{2}{3}$ あることから

$$\frac{2}{3} \times (\frac{2}{3} \times \frac{1}{2} + (1 - \frac{2}{3}) \times (\frac{1}{2} + 1)) + (1 - \frac{2}{3}) \times 1 = \frac{8}{9}$$

に実行時間が削減されることが期待される。

以上の考察から図 1 のアルゴリズムを構築した。図 1 のアルゴリズム中の `oSQUFOF` はオリジナルの SQUFOF アルゴリズムであり、第一引数に分解したい数、第 2 引数に first phase での連分数展開数の最大値を渡す。連分数展開を最大値まで実行しても因子が見つからなかった場合は 0 を返す。図 1 を実行したところオリジ

Input: N (to be factored)

Output: factor of N

$\texttt{Cf}_{\max} \leftarrow \dfrac{2^{62}-1}{N}$
if $\texttt{Cf}_{\max} > 11$ then
    $\texttt{Cf}_{\max} \leftarrow 11$
for $\texttt{Cf} \leftarrow 3$ to $\texttt{Cf}_{\max}$ step by $+2$ do
    $p \leftarrow \texttt{oSQUFOF}(\texttt{Cf} \times N, \dfrac{2^{17}}{\texttt{Cf}_{\max} - 1})$
    if $p > \texttt{Cf}$ then
        return non-trivial factor of p
return $\texttt{oSQUFOF}(N, 2^{19})$

図 1: 改良 SQUFOF アルゴリズム

ナルの SQUFOF 法に比べて約 1 割の高速化が計られた。

4 SQUFOF 法実現要素技術の改良

[Rie93, pp.190–193] にある SQUFOF をそのまま実装した場合、最も重い処理が除算であり、次に重い処理が平方数判定である。Pentium III において

	一回当たりの平均処理時間 [clock cycle]
first phase	100
second phase	53

[†3]自明な因子 (かける小さな数) を発見すると失敗となる。

であり、[Fog00] によると、除算で用いている `div` および平方数判定で用いている `fsqrt` instruction の latency はそれぞれ 39[†4]、69[†5]であること、`fsqrt` は first phase の連分数展開の 2 回に一回であり、$\frac{1}{2}$ の確率で呼ばれていることを考慮すると、

	平均処理時間中	
	`fsqrt` の占める割合 (%)	`div` の占める割合 (%)
first phase	17	39
second phase	0	74

も占めていることが分かる。従って、除算および平方数判定の高速化は極めて重要である。

4.1 除算の高速化

商 q_i は

$$1 \leqq q_i = \left\lfloor \frac{\left\lfloor \sqrt{N} \right\rfloor + P_i}{Q_i} \right\rfloor < 2\sqrt{N}$$

を満たし、かつ分子分母はほぼランダムであろうことから商のほとんどは 1 であることが予想される。実際、計算機実験により

q_i	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
割合 (%)	42	17	9	6	4	3	2	2	1	1	1	1	1	1	1	1	0
累計 (%)	42	59	68	74	78	81	83	85	86	87	88	89	90	91	91	92	92

から、小さな商がほとんどであることが分かる。q_i の計算を

$\texttt{numerator} = \left\lfloor \sqrt{N} \right\rfloor + P_i$
if $\texttt{numerator} - Q_i < Q_i$ then
　　$q_i = 1$
else
　　$q_i = \left\lfloor \dfrac{\texttt{numerator}}{Q_i} \right\rfloor$

と書き換えることにより、商が 1 であるかどうかは引き算、比較、条件分岐で判定できるのでそれぞれ 1 clock cycle であると仮定し、条件分岐予測ミスの penalty を 26 clock cycles、条件分岐は商の分布表の割合が大きいものに予測されるもの、1 の代入も 1 clock cycle とすると上記コードの除算部分は

$$1 + 1 + 1 + 42\% \times (26 + 1) + (1 - 42\%) \times 39 = 37 < 39$$

から、若干高速化されることが分かる。商 q_i が 1 と分かっていることを用いると後続の乗算 2 つを省略できるので、さらに高速化可能である。実際には

[†4]但し `idiv` の経験からもう若干速い (31 以上) 可能性あり。
[†5]但し、float で使っているのでもう少し速い可能性あり。

- 乗算がもう少し速い
- 分岐予測の精度はもっとよい
- 分岐予測ミスの penalty が 26 clock cycle というのは最悪値である

ということからもう少し速いことが期待され、商が 10 までを特別扱いする場合が一番速く、そのときは

	一回当たりの平均繰り返し処理時間 [clock cycle]
first phase	79
second phase	33

であった。実際には乗算の高速化も含んでいるが、すべてが除算の高速化に結び付いたとすると、$100 - 79 \approx 53 - 33 = 20$ clock cycles の高速化であり、また、もとの除算が 39 clock cycles を要していたと仮定すると除算については約倍の高速化が計られたことになる。

4.2 高速平方数判定

平方根は first phase でしか出てこないので、この節では first phase についてのみ言及する。また、一回当たりの平均繰り返し処理時間はすでに除算高速化が行なわれたものとしている。

[Rie93, pp.190–193] でも一部実現されているように、SQUFOF 法の first phase では平方根の計算は一回で十分であり、残りの場合は平方数かどうかの判定のみ高速にできれば良い。[Rie93, pp.190–193] では

$$\forall R[R^2 \not\equiv 2, 3 \pmod 4]$$

であることを利用し

if $Q_{i+1} \wedge 3 > 1$ then
    go to the next continued fraction
else
    $R \leftarrow \left\lfloor \sqrt{Q_{i+1}} \right\rfloor$
    if $R^2 = Q_{i+1}$ then
        go to second phase
    go to the next continued fraction

となっており、**重い**平方根計算はなるべく計算しないで済むようにしていた。文献 [Coh93, pp.39–41] にも同様の手法、つまり 4 以外の法について、平方数ならとり得る値を表としてあらかじめ用意する平方数判定が示されているが、計算機では $\mathrm{mod} 2^k$ は計算し易いことはもちろん、$\mathrm{mod} 2^k \pm 1$ もそれなりに計算し易いこと、さらに、表の大きさ、$\mathrm{mod}\, 2^k(\pm 1)$ での非平方数の割合を考慮し次のアルゴリズムを構築した。

if not `isSquareMod256[`$Q_{i+1} \wedge$ `0xff]` then
    go to the next continued fraction   /* 83% discarded */
else   /* compute mod$2^8 - 1$ */
    `tmp` $\leftarrow (Q_{i+1} \wedge$ `0xff00ff`$) + ((\frac{Q_{i+1}}{2^8}) \wedge$ `0xff00ff`$)$
    `tmp` $\leftarrow$ `tmp` $+ \frac{\texttt{tmp}}{2^{16}}$
    `tmp` $\leftarrow$ `tmp` $+ \frac{\texttt{tmp}}{2^8}$
    `tmp` $\leftarrow$ `tmp` $\wedge$ `0xff`[†6]
    if not `isSquareMod255[tmp]` then
        go to the next continued fraction   /* rest of 79% discarded */
$R \leftarrow \left\lfloor \sqrt{Q_{i+1}} \right\rfloor$
if $R^2 = Q_{i+1}$ then
    go to second phase
go to the next continued fraction

この高速化により、`isSquareMod256[]` のみを用いた場合、一回当たりの平均繰り返し処理時間は 55 clock cycles であり、`isSquareMod255[]` も用いると 43 clock cycles となった。なお、事前に mod4 の判定を行なうと若干速度が低下した。

平方数判定のみが問題となることにさらに着目し、平方数判定したい数の上位 17 ビットが定まれば、残り 15 ビットで一意に平方数かどうかを判定できることをも発見された。つまり、上位 17 ビットを index とし、対応する平方数の下位 15 ビットを保存すれば、上位 17 ビットを切り出し、表を引き、下位 15 ビットが一致するかどうかを判定することにより判定できる。この 192KB[†7]の巨大な表を使ってよいのだとすると一回当たりの平均繰り返し時間は 39 clock cycles まで削減できる。なお、この場合には `isSquareMod255[]` は利用するとかえって遅くなるので使わない。first phase と second phase の違いは基本的に平方数判定のみであることを考えると実質、平方数判定は 2 連分数展開につき一回実行することを考えると

$$2 \times (39 - 33) = 12 \text{ [clock cycle]}$$

で実現できていることが分かる。また、この考えを用い、表を引くだけで、平方根かまたはその値の +1 を得ることが出来るので、2 乗して大小関係を確認することにより平方根も高速に求めることが出来るが、first phase における実際の平方根回数が 1 回に削減できたので最終的にはこの方法を採用しなかった。

5 結果

その他、loop unroll や命令スケジューリングなど一般的な最適化手法を使った結果、平均

[†6]好運なことに `0x101` の場合に必要な `isSquareMod255[0]` = `isSquareMod255[1]` が成立した。
[†7]先頭ビットが 0 ではじまる表は不要であるので 256KB の表を確保する必要はない。

CPU	実行時間 (標準偏差) [Mcycles/合成数]	実行時間 [ミリ秒]
Pentium III	2.45 (2.84)	1.87 (1.3GHz 換算)
Pentium 4	4.06 (4.69)	1.60 (2.53GHz 換算)

で分解できた。コンパイルオプションについては Pentium III は 1.2 節と同じで、Pentium 4 については、1.2 節のオプションに加え、gcc を用い、`-mcpu=i386` を加えた場合が`-mcpu=`オプションの引数を変化させたうちでは最も速かった。この分解においてほとんどは、小さい方の因子を発見したが、35 合成数 (3.7%) については大きい方の因子を発見した。なお、分解時間の分布などを図 2、3、4 に示す。

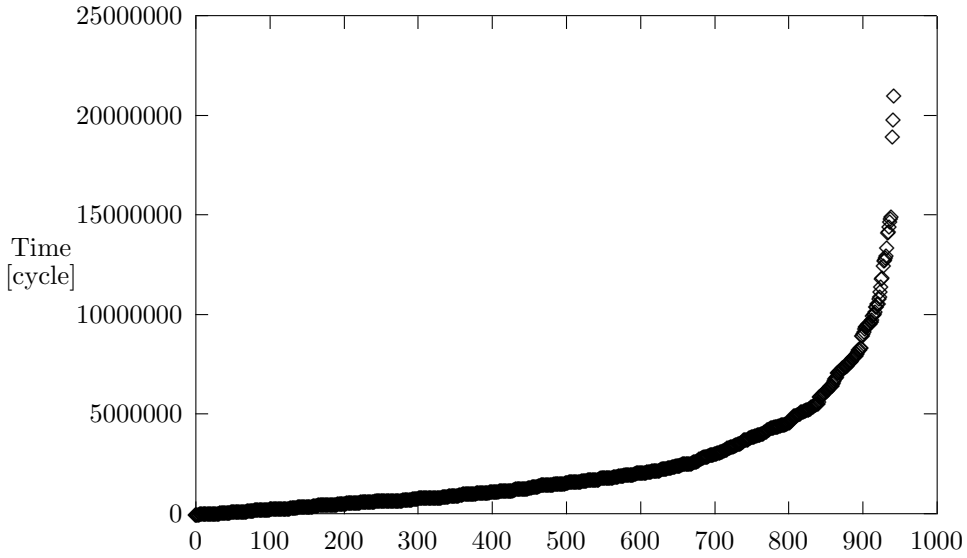

図 2: 分解時間の分布 [Pentium III]

6 今後の課題

今後の課題として次の問題が上げられる。

- 小さな数をかけたときの、挙動の理論づけ。とくに合成数をかけた場合に効率が良くなりそうな問題の解明。
- 篩の結果から洩れてくる large prime の候補に対して、素数判定からの実装最適化。
- $p-1$ 法、ρ 法などを前処理とする分解高速化。
- Pentium 4 での最適化。

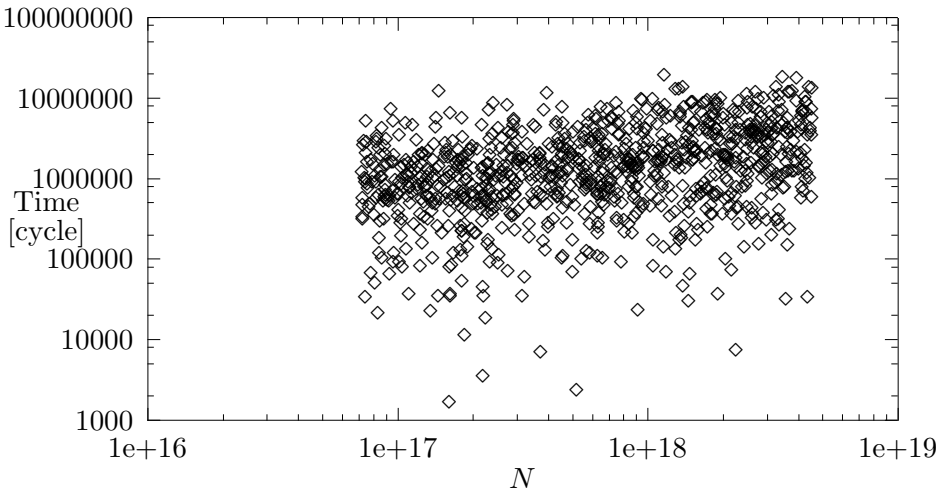

図 3: 分解時間と合成数 N の関係 [Pentium III]

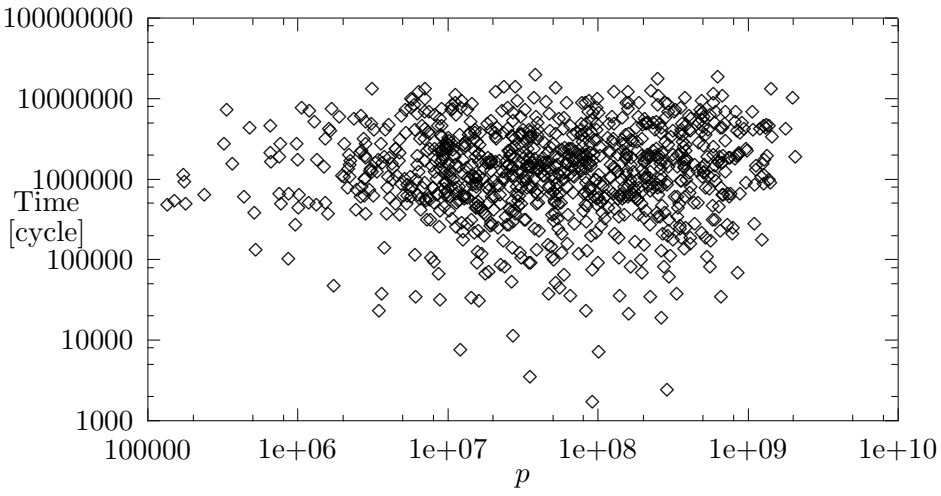

図 4: 分解時間と発見された因子 p の関係 [Pentium III]

参考文献

[Coh93] Henri Cohen. *A Course in Computational Algebraic Number Theory*, Vol. 138 of *Graduate Texts in Mathematics*. Springer-Verlag, Boston, Basel, Berlin, 1993. Second Corrected Printing 1995.

[Fog00] Agner Fog. *How to optimize for the Pentium microprocessors*, 2000. (`http://www.agner.org/assem/`).

[Rie93] Hans Riesel. *Prime Numbers and Computer Methods for Factorization*, Vol. 126 of *Progress in Mathematics*. Birkhäuser, Boston, Basel, Berlin, second edition, 1993.

[木田 01] 木田祐司. 特殊数体ふるい法による 200 桁の素因数分解, 2001. 日本応用数理学会「数論アルゴリズムとその応用」研究部会 (JANT) 第 5 回研究集会.