2000 年:多倍長整数計算研究記録

梅谷 武

作成：2000-04-01 更新：2005-04-20

2000 年:多倍長整数計算研究記録
IMS:20000401002; NDC:418; keywords:多倍長整数計算, 整数;

目　　次

1　C++ による代数的構造の表現 (2000.4.1)

最初に C++ でどのように数を表現したらいいかを考えてみました。ANSI C++ について勉強しているうちに、ブルバキ風の抽象的な表現がある程度可能であることに気が付きました。これについては次の文書にまとめてみました。

「C++ による代数的構造の表現」

2　エラトステネスのふるい (2000.4.15)

実際に整数型を実装してみたのですが、前節で考えたようにはいかないようです。数学上の抽象概念を C++ の抽象クラスとして記述するというのは難しいということがわかってきました。難しいというよりも C++ がそのようにできていないといった方がいいかもしれません。例えばある数クラスが順序環であるときに、順序

環という代数構造に対応する抽象クラスがあって、そのクラスを継承することによって、順序環であることを表現したいところです。そして例えば順序環に対して定義できる関数があったときに、その関数の定義を、順序環という抽象クラスを継承するという性質だけを使って記述するのが理想的です。ところが現在の C++ ではこういうことを直接的に記述することは残念ながらできないのです。

最初の理想は実現できないことがわかりましたが、それでもなお C++ には C や FORTRAN にはない魅力的な機能がいっぱいつまっています。いろいろ実験しながら C++ による数値計算の可能性をさぐっていきたいと思います。

ソースを以下に置いておきます。

- s0_base.h
- s0_math.h
- s0_math.cpp
- main.cpp

Visual C++ 6.0 + Windows 98 という環境で試しましたが、その他の環境でも少しの修正で動かすことができると思います。

実験の結果ですが、上のソースを Visual C++ で最適化無しで、インライン展開もしないで実行すると数クラスでやった方が組込型の約２０倍程度時間がかかるとことがわかりました。次回は数クラスをチューンナップして、どれだけ組込型に迫れるか実験してみたいと思います。

3 最適化とインライン化 (2000.4.19)

前節のソースを Visual C++ で実行速度優先最適化及びインライン展開を行なってコンパイルリンクすると約２０倍程度であった実行時間の差が約１．５倍程度に短縮されました。だいたい予想通りの効果があることがわかりました。なお、この実験はかなり遅いマシンでやらないとはっきりした差がでないと思います。さらに組込型は 32 ビット整数演算で、数クラスは Pentium 内臓 FPU の 64 ビット整数演算で行なっていますので、必ずしも条件が対等ではないことをお断りしておきます。次回は数クラスの機能を拡張して、より複雑な計算をしてみたいと思います。

4 ユークリッド互除法 (2000.4.22)

a, b を整数とし、$b \neq 0$ としますと、次のような整数 q, r が一意的に存在します。

$$a = bq + r, 0 \leqq r < |b|$$

このことによって、$q \equiv a/b, r \equiv a\%b$ という 2 項演算を定義することができます。このような除法定理は、整数だけでなく例えば多項式環においても成り立ちます。一般に整域 R の 0 でない各元に対して、その大きさと呼ばれる整数 $d(a) \geqq 0$ が定義され、0 でない任意の元 $a, b \in R$ について、次のような $q, r \in R$ が存在するとき、

$$a = bq + r, r = 0 \text{ または } d(r) < d(b)$$

R をユークリッド整域といいます。さらに $q, r \in R$ が一意的に定まるという条件のときに/と % が定義できるのですが、逆に/と % が定義できるユークリッド整域を代数構造としてどのように特徴付ければいいかは承知しておりません。

ユークリッド整域において/と % が定義できるとき、任意の 2 元 a, b について、ユークリッドの互除法に

よりその最大公約数 $gcd(a,b)$ を計算することができます。$gcd()$ をテンプレート関数として書いてみました。ソースを以下に置いておきます。

- s0_base.h
- s0_math.h
- s0_math.cpp
- main.cpp

$gcd()$ においてその引数クラスが/と % が定義されているユークリッド整域という性質をもっていることを検査したいところですが、実行効率という観点から型検査はしないことにします。

5 拡張ユークリッド互除法 (2000.4.24)

前節で最小公倍数を求める関数：$gcd()$ を作りましたが、最小公倍数を求める関数を $lcd(a,b)$ とすれば、

$$a * b = gcd(a,b) * lcd(a,b)$$

という性質から $lcd()$ は $gcd()$ を使って簡単に作ることができます。

/と % が定義できるユークリッド整域を簡単のために、仮に「一意ユークリッド整域」と呼ぶことにします。一意ユークリッド整域 R において、自明でない2元一次不定方程式：

$$a * X + b * Y = c,\ a,b,c \in R$$

は c が $gcd(a,b)$ の倍数であるときに限り解が存在し、その解は拡張ユークリッド互除法によって求めることができます。これを $euc()$ という関数にしてみました。

- s0_base.h
- s0_math.h
- s0_math.cpp
- main.cpp

6 暗黙の型変換 (2000.4.24)

文献 [C1] を読んでいて、整数クラスにおいて、

$$operator\ int()$$

を定義しておくことによって、基本型:int が必要とされる文脈でこの整数クラスが使われる場合に暗黙の型変換が行なわれるという便利な機能があることに気がつきました。そこでクラス定義を修正しました。またメンバ関数名を小文字にするという修正も施しています。

これでエラトステネスのふるいを書き直してみます。

- s0_base.h
- s0_math.h
- s0_math.cpp
- main.cpp

7 べき乗 (2000.4.27)

よく使われる整数べき乗を $ipow()$ という関数にしました。実数べき乗については標準関数の $pow()$ を多重定義することにします。べき乗は RSA 暗号における暗号化及び復号化で使われるためにその高速算法が現在でも盛んに研究されているようです。ここでは汎用性があって簡単な繰り返し 2 乗法を文献 [N3] を参考に実装しています。

- s0_base.h
- s0_math.h
- s0_math.cpp
- main.cpp

8 Mersenne 数 (2000.4.27)

そろそろ多倍長整数の実装に取り掛かりたいと思いますが、その評価用に Lucas による Mersenne 数の判定プログラムを書いておきます。現段階では演算のオーバーフローが検出できませんので、判定が本当に正しいかどうかを知るにはオーバーフローの検出機能を付け加える必要があります。

また整数クラスが 2 進表現のときに bit 毎の論理演算による高速算法を記述できるように組込型に対する bit 毎の論理演算と同じ演算子関数を実装しています。

- s0_base.h
- s0_math.h
- s0_math.cpp
- main.cpp

9 多倍長整数クラスの設計 (2000.5.9)

試行錯誤の末、多倍長整数クラスを次のようにするのがいいのではないかという結論になりました。まず基数 P を定めて、任意の正整数 a を

$$a = a_{s-1}P^{s-1} + \cdots + a_1P + a_0,\ a_{s-1} \neq 0,\ 0 \leqq a_i < P(i = 0, \cdots, s-1)$$

と P 進表現します。このときの s を a の次数と呼ぶことにします。多倍長整数クラスは P を定数として定め、配列の長さを可変とするテンプレートクラスとして表現します。

```
UINT32 status;
  UINT32 degree;
  UINT32 v[n];
```

$degree$ に次数を、$v[i]$ に a_i を格納します。$status$ は後に Pentium 用に手書きで最適化することも考えて、Pentium のフラグレジスタと同じ bit 配列にしました。現在はすべて C++ で記述しているので、使っているのは OF(オーバーフロー),ZF(ゼロ),SF(負) の 3 bit です。$a = 0$ のときには $degree = 0$ と定めることにします。

簡単な方法 (文献 [N5],[N6]) で演算を実装して、Mersenne 数の判定をしてみました。

- s0_base.h
- s0_uint32.h
- main.cpp

とりあえずはうまく動いているようですが、他の方法では検証しておりませんので、不具合があるかもしれません。Pentium-133MHz で p=1279 のときに２４秒もかかってしまいます。UBASIC では１秒もかかりませんので話になりません。

次回は高速乗算法を少し研究してみたいと思います。

10 高速乗算法 (2000.5.20)

２つの多倍長整数 a, b:

$$\begin{aligned} a &= a_{s-1}P^{s-1} + \cdots + a_1P + a_0,\ 0 \leqq a_i < P(i = 0, \cdots, s-1) \\ b &= b_{s-1}P^{s-1} + \cdots + b_1P + b_0,\ 0 \leqq b_i < P(i = 0, \cdots, s-1) \\ & \quad a_{s-1} \neq 0 \text{ または } b_{s-1} \neq 0 \end{aligned}$$

の積 ab を計算することについて考えます。まず基数 P を変数 X に置き換えることによって多項式 $a(X), b(X)$ を作ります。

$$\begin{aligned} a(X) &= a_{s-1}X^{s-1} + \cdots + a_1X + a_0,\ 0 \leqq a_i < P(i = 0, \cdots, s-1) \\ b(X) &= b_{s-1}X^{s-1} + \cdots + b_1X + b_0,\ 0 \leqq b_i < P(i = 0, \cdots, s-1) \\ & \quad a_{s-1} \neq 0 \text{ または } b_{s-1} \neq 0 \end{aligned}$$

この積 $c(X) \equiv a(X)b(X)$ は高々 $2s-2$ 次ですが、長さを $2s$ に揃えるために

$$\begin{aligned} c(X) &= c_{2s-1}X^{2s-1} + \cdots + c_1X + c_0 \\ c_j &= \sum_{k=0}^{s-1} a_{j-k}b_k,\ j = 0, \cdots, 2s-1,\ (k < 0, k \geqq s \to a_k \equiv 0) \end{aligned}$$

と書くことにします。このとき数列 $(c_0, \cdots, c_{2s-1})$ は、長さ $2s$ の数列 $(a_0, \cdots, a_{s-1}, 0, \cdots, 0),\ (b_0, \cdots, b_{s-1}, 0, \cdots, 0)$ の畳み込みになっていますが、次のことが成り立ちます。

定理 10.1 (多項式の高速乗算法) 長さ $2s$ の数列 $(a_0, \cdots, a_{s-1}, 0, \cdots, 0),\ (b_0, \cdots, b_{s-1}, 0, \cdots, 0)$ に対して、その *DFT(*離散フーリエ変換*)* を $(a'_0, \cdots, a'_{2s-1}),\ (b'_0, \cdots, b'_{2s-1})$ とすると、畳み込み $(c_0, \cdots, c_{2s-1})$ は長さ $2s$ の数列 $(a'_0b'_0, \cdots, a'_{2s-1}b'_{2s-1})$ の *IDFT(*逆離散フーリエ変換*)* に等しい。

この証明については、例えば文献 [S1] を参照してください。これを利用して多倍長整数の乗算を行うことにします。その手順を簡単にまとめておきます。

算法 10.2 (多倍長整数の高速乗算法)

1. 長さ $2s$ の数列 $(a_0, \cdots, a_{s-1}, 0, \cdots, 0),\ (b_0, \cdots, b_{s-1}, 0, \cdots, 0)$ に対して、その$DFT(a'_0, \cdots, a'_{2s-1}),\ (b'_0, \cdots, b'_{2s-1})$ を計算する。
2. $(a'_0b'_0, \cdots, a'_{2s-1}b'_{2s-1})$ の$IDFT$を計算し、畳み込み $(c_0, \cdots, c_{2s-1})$ を求める。
3. $c(X) = c_{2s-1}X^{2s-1} + \cdots + c_1X + c_0$に基数 P を代入し、桁上げ処理を行なう。

これを実装することが目標ですが、まずは DFT の高速算法である FFT(高速フーリエ変換) について実験してみようと思います。

11 The art of computer programming(2000.8.5)

多項式の高速乗算法を応用して多倍長整数の高速乗算法を実装しようという予定だったのですが、T.W. ケルナーの本に書いてある「このやり方自体かなり手間が省けるとはいえ、エレガントとはいえない。」という言葉が気になって、ストラッセン・ショーンハーゲ法に挑戦する気になってきました。これについては、[N5],[N6] にも大雑把な説明があるのですが、やはりクヌースの本 [S2] を熟読しないと独自に実装できるレベルまで理解することは難しいようです。

12 **夏休みの宿題** (2000.8.12)

クヌースの本を読んでわかったのですが、n 桁の乗算を計算量 $O(n \log n \log\log n)$ で行なうというストラッセン・ショーンハーゲ法は、理論上の計算量の上界を改善するためのものであって実装を目指したものではないということです。クヌース自身も、実際には別の方法がいいということを述べています。したがってクヌースの本においても具体的なストラッセン・ショーンハーゲ法の紹介はされていなくて、論文が示してあるだけでした。和田秀男さんの [N5],[N6] には、ストラッセン・ショーンハーゲ法の概要が説明されていて、実装するときはクヌースの本を参考にしなさいということが書いてあるのですが、どうもこの意味を勘違いしていたのかもしれません。

ストラッセン・ショーンハーゲ法では、計算する数の大きさによって語長が変わっていきます。ですからそのまま作譜しようとすると演算が多倍長演算になってしまいます。また $2^n + 1$ を法としているために効率の良い最適化ができません。このように不利な状況が多いので、実装しようかどうか迷っています。ただ、すべて整数演算のみで、フーリエ変換がシフト演算と加法のみでできるという点に非常に魅力を感じています。特に、整数の乗算のために浮動小数点演算を使うというのがどうも引っかかっています。整数の演算は整数演算命令だけでやってみたいということもあります。

このようなわけで夏休みに、ストラッセン・ショーンハーゲ法を効率よく実装する方法を考えることにしました。

13 **夏休みの成果** (2000.8.26)

ストラッセン・ショーンハーゲ法を実装するというのは、長手順の詰め将棋のような感じで、頭の中ではすべてを読みきったつもりでいるのですが、いざこれを文章で説明したり、作譜しようとするとかなりの情報量になってしまいそうです。そこで、論点を整理して見通しよくするために離散 Fourier 変換を抽象化することについては、別にひとつの文書としてまとめてみました。[F1]

さて「ストラッセン・ショーンハーゲ法を効率よく実装する方法」なのですが、効率がいいかどうかは別にして、少なくとも実装する方法についてはなんとかなりそうです。たばになったメモをめくりながらこれをまとめている途中です。完成予定は９月中というところでしょうか。多倍長整数クラスを UBASIC 並みの性能にするのは、何とか今世紀中に達成することを目標にしています。

参考文献

代数学

[A1] 松坂 和夫, “代数系入門”, 岩波書店, 1976

[A2] 上野 健爾, “代数入門”, 岩波書店, 2004

数論

[N1] 高木 貞治, “初等整数論講義 第 2 版”, 共立出版, 1971
[N2] 芹沢 正三, “*C* による初等整数論”, 森北出版, 1993
[N3] 木田 祐司, 牧野 潔夫, “*UBASIC* によるコンピュータ整数論”, 日本評論社, 1994
[N4] 山本 芳彦, “数論入門”, 岩波書店, 2003
[N5] 和田 秀男, “コンピュータと素因子分解 改訂版”, 遊星社, 1999
[N6] 和田 秀男, “高速乗算法と素数判定法”, 上智大学数学教室, 1983
[N7] P. Ribenboim(我郷 孝視訳), “素数の世界 -その探索と発見-”, 共立出版, 1995

Fourier 解析

[F1] 梅谷 武, 離散 *Fourier* 変換

C++

[C1] B. Stroustrup(長尾 高弘訳), “プログラミング言語 *C++* 第 3 版”, Addison-Wesley, 1998
[C2] P. and G. Anderson(長尾 高弘訳), “最新 *ANSI C++* オブジェクト指向プログラミング”, プレンティスホール出版, 1999

算法

[S1] 野下 浩平, 高岡 忠雄, 町田 元, “基本的算法”, 岩波書店, 1983
[S2] D. E. Knuth(中川 圭介訳), “準数値算法/算術演算”, サイエンス社, 1986

x86

[X1] 蒲池 輝尚, “はじめて読む *486*”, アスキー, 1994
[X2] インテル, “*Pentium* プロセッサ・ファミリー アーキテクチャとプログラミング”, インテル, 1993
[X3] 藤波 順久, アセンブラでの高速化
[X4] A. Fog(藤波 順久訳), *HOW TO OPTIMIZE FOR THE PENTIUM PROCESSOR*