そろばんにおける計算アルゴリズムの解析

2021446 工藤杏菜

令和 6 年度提出

小樽商科大学 商学部
社会情報学科

目次

第1章

はじめに

人類は紀元前から現在にいたるまで様々な場面で計算を行い、また計算の改良に取り組んでいる。日本ではそろばんという計算器具が中国から 16 世紀後半に伝来したと言われ、17 世紀前半には算術書に載るほどに普及した。それ以降そろばんは和算の発展を助け、生活に関する計算に用いられるようになったが、この独自に発展した計算アルゴリズムは現在において使われていないものと使われているものがある。今回はそろばんの計算アルゴリズムに注目し、情報科学の観点から評価を行っていく。

1.1 研究背景

そろばんは 17 世紀前半に塵劫記 [7] などの和算書で登場して以来、日本で広く使われ続けている計算器具である。そして、そろばんを用いた計算のことを珠算と言う。 しかしこの珠算の長い歴史に反して現在普及している計算アルゴリズム、つまり計算手順や処理手順は、一つの演算につき一つであることが多く、長く使用されている事実に対して少ないと感じる。唯一複数の計算アルゴリズムが用いられている計算は乗算だが、それも二種類であるため多いとは言い難い。ただし本研究では、そろばんにおいて法と呼ぶ乗数 (かける数) または除数 (割る数) を盤面に置く方法と置かない方法を、同一の手法であると見做して数えている。手法の数の少なさは、珠算の指導から明らかである。例えば、そろばんの授業の副教材とされているテキスト [5] では加算と減算の指導を中心として、乗算は暗算で行う両落とし法を紹介するに留めている。両落とし法は被乗数も置かない手法として珠算にも適用される。他に石戸珠算学園が監修したテキスト [1] では、同様に加算と減算の指導を行っている。そして、そろばんを使った乗算と除算も解説を行っている。乗算は乗数を盤面に置かない方落とし法のみを解説している。そして除算は商除法と呼ばれるそろばん以外でも普及している、かけ算九九を用いた除算の解説をしている。

何故上記の手法が指導され、普及しているのか。現在の珠算は、検定や競技のどちらにおいても時間内に正答した問題数で合否や順位が決まる。つまり、速く正確に問題を解くことがが求められる。そのため、

計算アルゴリズムの見直しが行われていないことに疑問を覚える。一方情報科学の分野では、新たな乗算アルゴリズムとしてKaratsuba法が1960年代に提案された。このKaratsuba法 [2] は、従来の手法と比較して加算の回数は増えるが、乗算の回数は減って、全体の計算量は小さくなる。つまり、加算を増やして乗算を減らすことで従来よりも速い計算を可能にした計算アルゴリズムである。このような高速化を図る提案が珠算において観測できないことは、非常に不自然だと思われる。

以上のことから、これまで提案されてきた手法の中で現在残っているものが珠算において最適な手法であるという仮説を立てられる。この仮説について、Karatsuba法も取り上げて新たな乗法として珠算への実装の是非を起点に検討していく。

1.2 関連研究

本研究とアプローチは異なるが、そろばんに関する研究をいくつか紹介する。まず最初に山崎ら [6] の研究ではそろばんの発明以前も含め、四則を時系列に沿ってそれぞれ確認している。その中でも特に注目したい点がいくつかある。

一つ目は書籍全体を通して多くの手法が提案されていたこと。特に乗算除算は普及したかどうかに関わらず多くの手法が提案されていたことが分かる。

二つ目は、乗法の歴史についてである。かけ算九九が乗法に用いられるようになってからは、現在普及している乗算アルゴリズムと大きく変化はしていないようである。乗数が2桁以上でかつ一番大きな位の数字が1である場合や、2桁以上の乗数を因数分解するなどの、特定の条件下における工夫は見られる。しかし、現在も用いられている方落とし法が主に使われ、両落とし法も時折和算書に取り上げられていたことから乗法において計算アルゴリズムは大きな変化がなかったと考えられる。

三つ目は、除法の歴史についてである。除法は乗法とは反対に、普及している計算アルゴリズムに大きな変化があったことが分かる。一度目は帰除法が広く普及した時である。帰除法とは九帰法と呼ばれる割り声 (割り算九九) を用いる除算アルゴリズムで、除数が1桁の計算を簡略化するために考案されたという。そして帰除法は次第に2桁以上にも対応するようになり、多くの指導書では帰除法ばかり取り上げられるほど普及していた。そして、帰除法から商除法に移行した時が二度目の変化である。帰除法への転換以降商除法が普及しなかった理由として山崎らは、刊本和算書全体の半数に達する塵劫記・改算記と名の付く和算書が帰除法を取り上げるあまりに商除法を駆逐したと述べている。そのような背景から商除法は明治時代から徐々に普及し、大正時代において商除法が完成されたようだ。

また、帰除法について詳しく述べられている曽我氏が執筆した研究を二つ紹介する。一つは帰除法の計算方法 [4] を丁寧に解説していて、もう一つは帰除法から商除法への転換 [3] について述べられている。こ

れによると、曽我氏は帰除法を用いることは不利である。もしくは商除法は有利であると考えていることが分かるが手法の比較については否定的で、明確な根拠は示されていない。

これらの関連研究から、 帰除法から商除法へと転換した理由は関連研究において時代背景や両方の手法を知る者による判断で、計算アルゴリズムの観点からは明らかになっていないと考える。そのため本研究では普及している手法としていない手法を、人間が行う時の複雑さと計算の速さという二つの観点から、手法の不変や遷移について検討する。

1.3 本研究の目的

本研究の目的は、関連研究で示された計算アルゴリズムやコンピュータで利用されている手法が何故現在の珠算で使われていないのかを明らかにすること。そのために、アルゴリズムをプログラムコードのようにした疑似コードを作成し、情報科学の観点から各計算アルゴリズムの解析を行う。

疑似コード作成後に検討することは、具体的に次の通りである。

- Karatsuba 法が新たな乗算アルゴリズムとして珠算で使用を期待できるのか
- 帰除法が再び除算アルゴリズムとして使用されるのか

本研究の目的はどれも関連研究では達成していないもので、これを達成することで期待できる貢献は二つある。一つは、本研究の目的が明らかにされることで、そろばんの新たな計算手法を提案できる可能性があること。もう一つは、そろばんの手法がコンピュータへ実装しやすくなるため教育の分野による活躍が見込めることだ。

1.4 本論文の構成

第 2 章使用するそろばんや研究環境について確認を行い、次に加法と減法の疑似コードを基に、加算の紹介と解析方法について具体的な説明と定義を行う。第 3 章では乗法である方落とし法・両落とし法・Karatsuba 法の三種類の疑似コードを作成する。これを基に各手法の説明を簡単に行った後解析を行い、結果を述べる。第 4 章では除法である商除法と帰除法の疑似コードを作成する。その後同様に各手法の説明と解析を行い、最後結果を述べる。第 5 章では得られた全ての結果をまとめ、考察を行うとともに、本研究の結論と今後の課題について述べる。また、別表として本研究内に載せていない疑似コードを 2 ページに渡って収録する。

第2章

研究方法

そろばんは手法の説明や指導では問題を用いて具体的にかつ全ての場合を別にして行うことが多く、手法の評価は計算者の主観に頼るものが多いが、ここでは情報科学におけるアルゴリズムの評価方法をそろばんに応用し、明確な基準を制定する。

2.1 本研究のそろばん

初めにそろばんの基本的な名称や操作について紹介し、次に用語の定義や本研究で設定する条件を確認する。本研究では図 2.1 の通り、梁から下には一珠が 4 個、梁より上には五珠が 1 個のそろばんを想定する。3 桁毎に打たれている定位点は今後「点」と呼び、適宜コンマや小数点の役割を持つ。このそろばんは、基数を 10 として 0 から 9 までの数字を使う 10 進法で表現される。任意の点を一の位に設定し、その左に 1 つシフトした位置を十の位、さらに 1 つ左にシフトした位置を百の位と呼ぶ。一の位から右にシフトすると、小数となる。1 珠を足す時、必要な一珠の個数を梁に向かって上げる。五珠を足す時は、梁に向かって下げる。引く場合は一珠、五珠ともに足す時と反対の操作となる。

また、本研究では必要な桁数を確保できるとする。つまり、入力する値が大きいなどの理由でそろばんの桁が足りないという事態は本研究では発生しないとする。

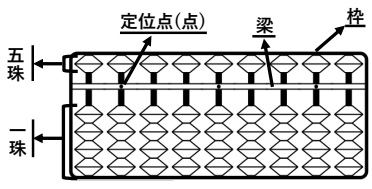

図 2.1: そろばん簡易図

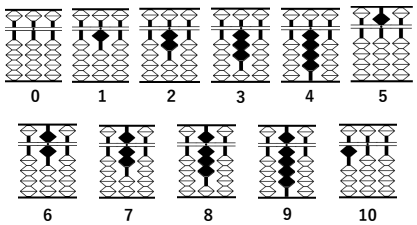

図 2.2: そろばんで表す数字

2.2 作成する疑似コード

疑似コードとはアルゴリズムをプログラムコードのようにしたもので、まず初めに本研究で最も簡単な疑似コードを紹介する。この合計 7 行で構成された手法 1 は、そろばん上で点に合うように操作の実行開始位置を決定する。1 行目より、疑似コードの名前は設定 1 で、x というそろばんへ入力する値の桁数を引数とする。そして 2 行目から条件によって処理方法が異なることが分かる。まず x を 3 で割った余りが 1 であるなら 3 行目の処理が、余りが 1 ではなく、2 であるなら 5 行目の処理が、余りが 1 でも 2 でもないなら 7 行目の処理が実行され、この手法 1 は終了する。

手法 1 実行開始位置の決定

```
1: procedure 設定 1(x)
2:     if x%3 == 1 //%は余りを求める演算子
3:         実行開始位置は一の位
4:     else if x%3 == 2
5:         実行開始位置は十の位
6:     else
7:         実行開始位置は百の位
```

このような疑似コードを作成する際に定めた条件は次の通りで、今後登場する疑似コードに適用する。

- 入力は k 桁の整数 A と m 桁の整数 B とする (首位はそれぞれ a_1, b_1、末位は a_k, b_m)
- 被乗数と被除数を実と、乗数と除数を法と呼び、それぞれ A と B が該当する
- A,B の首位である a_1, b_1 は 0 を取らない (1 以上 9 以下) とするため、k と m は自然数である
- 答えは小数切り捨てとして処理

小数を扱わない理由は、疑似コードが最も基本的な計算処理のみ扱うことで比較や説明に適した形になると判断したためだ。特に実行開始位置の決定方法や答えの処理が多岐に渡る。補数も同様の理由で取り上げないが、ほぼ同様にして小数を含む計算や答えが負の値になる計算も可能である。

2.3 加法と減法

次に基本となる加法と減法について説明を疑似コードを用いて行うが、減算のアルゴリズムは加算とほぼ同様であるため説明は加法のみ行う。加法と減法の違う点は加算の説明後に言及し、減算の疑似コードは別表に記載する。そして本研究における解析方法を説明した後、加算の疑似コードの解析を行う。

手法 2 は、k 桁の A をそろばん上に加える疑似コードである。首位である a_1 から末位である a_k に向かうため、i が k 回更新される。a_i を加えるために必要な珠の数 (図 2.2 を参照) が充分なら、そのまま足す。

手法 2 足し算の方法

```
1: procedure 加算 (A = a_1 a_2 ... a_k)
2:     設定 1(k) で実行開始位置を決定
3:     for i = 1 to k
4:         if a_i が足せる
5:             a_i を足す
6:             実行開始位置を 1 つ右にシフトする
7:             i = i + 1
8:         else if 1 ≤ a_i ≤ 4 and 五珠が足せる
9:             五珠を足す
10:            | 5 − a_i | を引く
11:            実行開始位置を 1 つ右にシフトする
12:            i = i + 1
13:        else
14:            | 10 − a_i | を引く
15:            10 を足す//十の位に一珠を足す
16:            実行開始位置を 1 つ右にシフトする
17:            i = i + 1
```

不足しているなら、a_i が 5 以下でかつ五珠が足せる場合と、それ以外の場合に処理方法が二通りある。減算の疑似コード (手法 9) では、全て「足す」が「引く」に、「引く」が「足す」になる。また、一部処理手順が加算と異なる。この加算と減算の疑似コードに関わらず、処理手順は実際に執筆者が計算する時のものを参考にしているため、計算者によっては処理手順が合致しない可能性がある。これは指導や計算者の好みによる違いだが、アルゴリズムに大きく影響を与えるものではない。

本研究では情報科学の評価方法を珠算に応用し、疑似コードの解析と評価を行う。まず手法評価の基準として、1 桁の加算もしくは減算の実行回数 (以降加減回数) と疑似コードの行数の 2 種類をコストとしてカウントする。これらを基準とする根拠として、加減回数は珠算において複数の処理をまとめて行うことがあることを基にした。22 – 17 のように繰り下がりの 10 とまとめて 20 を引くことがある。これは少しでも処理回数を減らすことを目的にしていることから、加算や減算の回数をカウントすることは理論でも実践でも速度の指標として有効であると考える。また 1 桁の加減回数とすることで、より正確な結果に繋がるだろう。そして疑似コードの行数は、条件分岐や処理方法が複数存在すると手法の修得そのものが困難であることを基にした。これらは、執筆者が珠算を修得した時と指導を行った両方で観測したため検討の価値がある基準だと判断する。よって加減回数は計算速度の指標として、疑似コードの行数は人が手法の修得や実行をするための難しさの指標とし、回数と行数は少ない方が速く簡単に計算できる手法であるとする。ただし、加算、減算共に a_i の処理方法による違いはなく、すべて同様に 1 回としてカウントする。

これらの設定を基に手法 2 で実際に解析を行う。手法 2 は、1 桁の a_i を k 回足す (加える) ため加減回数は k 回で、全 17 行で構成された疑似コードである。この解析の結果である加減回数は、情報科学で主要な最悪 (最大) の回数である。入力される値や、前述のようにまとめた処理などによって、算出した加減回数よりも少なくなることは考えられるが、これより大きくなることはない。

第3章

乗法

そろばんにおける乗法は実を盤面に置く方落とし法と置かない両落とし法が用いられる。計算の速さにおいて、コンピュータで用いられる Karatsuba 法が珠算へ提案できるのかについて検討を行う。

3.1 方落とし法

方落とし法は実をそろばん上に置き、実の末位から計算する手法である。実の他に法も置く手法と区別されるが、法を置く以外は全て同一のアルゴリズムであるため本研究では同一の手法と考える。また、全ての乗算アルゴリズムと商除法で 1 桁同士の積を求める際には九九 (後に区別する必要があるため、以降かけ算九九と呼ぶ) の使用を想定する。

手法 3 普及しているかけ算

```
1:  procedure 方落とし法 (A = a_1 a_2 ... a_k, B = b_1 b_2 ... b_m)
2:      加算 (A) を実行
3:      for i = k to 1
4:          実行開始位置は a_i から 1 つ右にシフトした位置
5:          for j = 1 to m
6:              x = a_i · b_j
7:              if x が 2 桁である
8:                  加算 (x) を実行
9:                  実行開始位置を 1 つ右にシフトする
10:             else
11:                 実行開始位置を 1 つ右にシフトする
12:                 加算 (x) を実行
13:             j = j + 1
14:         減算 (a_i) を実行
15:         i = i − 1
```

手法 3 は実である A をそろばんに置き、その末位である a_k から計算を始める。実行中の実である a_i と法である B との積を順にそろばんに足していく。法 B は首位から末位に向かって j を m 回更新すると、a_i を引く。この一連の操作を実の首位 a_1 を引くまで繰り返す。方落とし法は、実の一の位が点に一致するように置いて計算を始めるため、積の 1 の位を合わせるようには置いていない。よって、最初に実をシフ

トさせて置くことや珠を弾かない左手の手を一の位に当たる場所に置いて目印にする工夫が行われる。

この手法 3 は、2 行目で k 桁の加算が実行される。加減回数が最悪であるため、8 行目の 2 桁の加算は m 回繰り返される。この繰り返しが k 回行われるため、8 行目の加算は合計 $2km$ 回ある。そして 14 行目の 1 桁の減算が k 回ある。以上より、最悪の加減回数は $2km + 2k$ 回で、合計 15 行で構成される。

3.2 両落とし法

両落とし法は実も法もそろばん上に置かない手法で、暗算にも用いられる。

手法 4 暗算にも適用されるかけ算

```
1: procedure 両落とし法 (A = a_1 a_2 ... a_k, B = b_1 b_2 ... b_m)
2:     for j = 1 to m
3:         設定 2(k, m − j + 1) をもとに実行開始位置を決定
4:         for i = 1 to k
5:             x = a_i · b_j
6:             if x が 2 桁である
7:                 加算 (x) を実行
8:                 実行開始位置を 1 つ右にシフトする
9:             else
10:                実行開始位置を 1 つ右にシフトする
11:                加算 (x) を実行
12:            i = i + 1
13:        j = j + 1
```

手法 4 の 3 行目より、新たに設定 2 を用いて実行開始位置を決定する。別表に収録した手法 10 より、設定 2 は設定 1 のように計算終了後に一の位が点に合うようにするが、法と実両方の桁数を用いて実行開始位置を決定する。これは、実を置かないことにより方落とし法同じ方法で実行開始位置を決定することができないため、新たな決定方法を採用している。

さらに両落とし法は、実と法の順番が方落とし法と異なる。実 A は首位である a_1 から末位 a_k に向かって i を更新し、法 B も首位である b_1 から末位 b_m に向かって j を更新する。そして実行中の a_i と更新する b_j の積を足し続け、j の更新が終わると$_1$を進め、次の a_i で同様に実行する。

また、この手法 4 で行われる「加算」を「減算」に置き換え実行することを、「掛け引く」と言う。積を引きたい時に用いる手法で、本研究では次の Karatsuba 法で使用する。

次に手法 4 の解析を行う。最大の加減回数を求めるため、積である x はすべて 2 桁であるとする。よって 2 桁の加算が k 回繰り返す。この繰り返しが m 回あるため、最悪の加減回数は $2km$ で、合計 13 行で構成された疑似コードである。

3.3 Karatsuba 法

Karatsuba 法 [2] は情報科学において乗算の回数を減らし加算の回数を増やしたことで、従来の乗法よりも速い計算を可能にしたアルゴリズムで実際に使用されている。他にも速い乗法は提案されているが、Karatsuba 法が珠算への適用が可能なアルゴリズムであると判断したため本研究で採用する。しかしそれでも Karatsuba 法はそのままでは珠算に適した処理手順ではないと考えられる。そこで順序を一部変更し、無駄な操作にならないようにする Karatsuba 法 1 と、さらにそろばんで計算しやすいように改善を行う Karatsuba 法 2 を作成する。

手法 5 コンピュータで適用されるかけ算

1: **procedure** Karatsuba 法 1($A = a_1a_2 \dots a_k, B = b_1b_2 \dots b_m$)
2: $x = (k+m)/2$ //演算子/は、商を求める
3: $A = A_L \cdot 10^x + A_R$
4: $B = B_L \cdot 10^x + B_R$
5: $z_2 = A_L \cdot B_L$
6: $z_0 = A_R \cdot B_R$
7: $Z = z_2 \cdot 10^{2x} + z_1 \cdot 10^x + z_0$
8: 両落とし法 $(A_L \cdot 10^x, B_L \cdot 10^x)$ を実行、この答えと 10^{2x} との商を z_2 と呼ぶ
9: 両落とし法 (A_R, B_R) を実行、z_0 と呼ぶ
10: 加算 $(z_2 \cdot 10^x)$ を実行
11: 加算 $(z_0 \cdot 10^x)$ を実行
12: **if** $\text{sign}((A_L - A_R)) == \text{sign}((B_L - B_R))$
13: 両落とし法 $((A_L - A_R), (B_L - B_R))$ を実行、この時一の位は 10^x の位置
14: **else**
15: 掛け引く $((A_L - A_R), (B_L - B_R))$ を実行、この時一の位は 10^x の位置

まず Karatsuba 法の説明を手法 5 を用いて行う。Karatsuba 法は実と法を同じ場所で 2 つに分割する必要があるため、分割地点を x とした。本研究では、A と B の桁数である k と m の和を折半して得られた値を分割の桁数である x とする。分割した A と B をそれぞれ 10^x の一次方程式として考え、それぞれの首位から分割地点までを A_L, B_L、分割地点から末位までをそれぞれ A_R, B_R とする。この分割方法については本研究の解析に関係するため、解析時に詳しく述べる。

$A \cdot B$ を分配法則に従って式を展開し、整理したものを Z として各項を z_2, z_1, z_0 と定義する。次に z_2 と z_0 を求めるが、この時二度手間にならないように両落とし法で適切な場所に計算する。また、z_2 の計算を両落とし法 $(A_L \cdot 10^x, B_L \cdot 10^x)$ としているが、計算終了時に位を一致させるための調整であるため加減回数は両落とし法 (A_L, B_L) と同義である。以降の 10^x も同様の役割であり、すべてそろばん上で実行されないため加減回数のカウントは 10^x を含まない桁数で行う。そして、$z_1 = A_L \cdot B_R + A_R \cdot B_L = A_L \cdot B_L + A_R \cdot B_R - (A_L - A_R)(B_L - B_R)$ より、既に算出した z_2, z_0 を適した場所に加え、$(A_L - A_R)$ と $(B_L - B_R)$ の符号が同じなら両落とし法を実行し、同じでないなら掛け引くを実行して計算は完了する。この掛け引くは、両落とし法の「加算」が「減算」に変わったものである。

次に手法 5 の解析を行う。既に述べたように、10^x は実行開始位置を指定するものであるため加減回数

にはカウントしない。また、本研究では分割地点を $x = (k + m)/2$ とした理由は解析のしやすさから設定した。計算のしやすさを考慮すると、k と m が同値なら相加平均の折半として 4 で割り、ある程度の差までなら 3 で、20 桁と 2 桁のように k と m に大きな差がある場合は 2 で割るのが良いと思われる。しかし、解析において 2 以外の値で割ると 12 行目の結果である差の最大の桁数が場合によって異なる。しかし、2 で割ると必ず最大の差の桁数は A_L と B_L の桁数であり $(k+m)/2$ に決まる。この時 A_L と B_L の片方もしくは両方が 0 の可能性がある。そのため 8 行目の両落とし法の実行が必要ない場合もあるが、最悪の加減回数を求めるため全て含めて Karatsuba 法の加減回数とする。また、x を変更した解析を行う場合にも適用ができるように、x を用いた式で導き、結果は今回設定した x に基づいて整理した k と m の式とする。そして z_2, z_0 の桁数は起りえる最大の桁数である実と法の和として解析を行う。

まず 8 行目で $k - x$ 桁の A_L と $m - x$ 桁の B_L で実行される両落とし法より、$2(k - x)(m - x)$ 回。同様にして、9 行目で $2x^2$ 回。10 行目の加算では、z_2 の桁数は実と法の和であるため $(k - x) + (m - x)$ 回。同様にして 11 行目は、$x + x$ 回。12 行目では A_L の加算 A_R の減算、B_L の加算と B_R の減算を実行すると考えるため、それぞれ $k - x + x$ 回と、$m - x + x$ 回。そして 13 行目と 15 行目はどちらか一方が実行され、両落とし法もしくは掛け引くの実と法の桁数は今回の分割で必ず大きい桁数になる A_R と B_R に依存する。繰り下がりが起きない時が最大の桁数であるため、13 行目と 15 行目の加減回数は $2x^2$ 回。

得られた全ての加減回数の和を求めると手法 5 の最悪の加減回数は $2(km + k + m + 3x^2 - (k + m))x$ 回で、今回定義した x に従うと $3km + k + m + (k^2 + m^2)/2$ 回、合計 15 行で構成された疑似コードである。

次に、Karatsuba 法 1 をそろばんに合わせて改善した疑似コードを作成する。Karatsuba 法 1 で、すでにそろばんで計算しやすいように z_2, z_0 を始めから適切な位置で計算を行うなどの変更は加えたが、特に手法 5 の 12 行目の場合分けが珠算において不向きで、正負によるミスを誘発しやすいと考える。そこで、z_1 を求める式に一部変更を加える。具体的な変更内容と式変形について手法 6 を用いて説明を行う。

手法 6 Karatsuba 法をそろばん用に改良したもの

1: **procedure** Karatsuba 法 $2(A = a_1a_2 \dots a_k, B = b_1b_2 \dots b_m)$
2: $x = (k + m)/2$ //x は小数位以下切り上げとする
3: $A = A_L \cdot 10^x + A_R$
4: $B = B_L \cdot 10^x + B_R$
5: $z_2 = A_L \cdot B_L$
6: $z_0 - A_R \cdot B_R$
7: $Z = z_2 \cdot 10^{2x} + z_1 \cdot 10^x + z_0$
8: 両落とし法 $(A_L \cdot 10^x, B_L \cdot 10^x)$ を実行、この答えと 10^{2x} との商を z_2 と呼ぶ
9: 両落とし法 (A_R, B_R) を実行、z_0 と呼ぶ
10: 両落とし法 $((A_L + A_R)\ , (B_L + B_R))$ を実行、一の位は 10^x とする
11: 減算 $(z_2 \cdot 10^x)$ を実行
12: 減算 $(z_0 \cdot 10^x)$ を実行

手法 6 は 9 行目まで手法 5 と全く同じもので、10 行目以降が改善を施した箇所である。今回 z_1 を、$z_1 = A_L \cdot B_R + A_R \cdot B_L = (A_L + A_R)(B_L + B_R) - A_L \cdot B_L - A_R \cdot B_R$ として式を変形したため、先に両

落とし法を行い (13 行目)、次に z_2 と z_0 を減らすようになる (14,15 行目)。これによって、符号による場合分けの必要がなくなった。

手法 6 の加減回数もまた、手法 5 とほぼ変わらないが、10 行目で実行される両落とし法の桁数が一致しない可能性がある。本研究で設定した x より、A_R と B_R が必ず A_L と B_L より大きな桁数になり、A_R と B_R の桁数の x に和を求めた桁数が依存する。和を求める時、繰り上がりが発生したものが想定できる最大の桁数であるため 10 行目で実行される両落とし法の実と法の桁数をそれぞれ $x+1$ と考えると、加減回数は $2(x+1)^2$ となる。

よって、手法 6 の最悪の加減回数は $2(km+k+m+3x^2-(k+m-2)x+1)$ 回で、今回定義した x に従うと $3km+4k+4m+(k^2+m^2)/2+2$ 回、合計 12 行で構成された疑似コードである。

3.4 結果

各乗算アルゴリズムの解析結果は表 3.1 にまとめた通りである。加減回数は両落とし法が最も少なく、行数は Karatsuba 法 2 が最も少なくなった。そして最も加減回数が多い疑似コードは Karatsuba 法 2 で、行数が最も多い疑似コードは方落とし法と Karatsuba 法 1 となった。また、Karatsuba 法 1 と Karatsuba 法 2 の分割方法である x の定義によって z_2 を求める両落とし法省略される可能性があると説明をしたが、その z_2 を求める両落とし法の加減回数を引いたとしても Karatsuba 法 1 は $2km+k+m+k^2+m^2$ 回、Karatsuba 法 2 は $2km+4k+4m+k^2+m^2+2$ 回であるため、順位に変わりはない。

	方落とし法	両落とし法	Karatsuba 法 1	Karatsuba 法 2
行数	15	13	15	12
最悪の加減回数	$2km+2k$	$2km$	$3km+k+m+(k^2+m^2)/2$	$3km+4k+4m+(k^2+m^2)/2+2$

表 3.1: 各乗算アルゴリズムの解析結果

この解析から得られた結果から、Karatsuba 法がそろばんの新たな乗算アルゴリズムとして提案することができないことが分かる。理由として、Karatsuba 法の加減回数が現在普及している方落とし法と両落とし法の加減回数よりも必ず多くなってしまうことが挙げられる。本研究で作成した手法 5 と手法 6 では両落とし法を採用し、できるだけ無駄のない手順にした。よって、両落とし法を修得した上で、もう一度手法修得の労力を割いて方落とし法よりも加減回数が多い、つまり遅い計算しかできない Karatsuba 法を採用するメリットはないからである。以上より現在のコンピュータで用いられている乗法は、普及している珠算の乗法に代わる手法としての提案ができないとわかった。次章では、かつてそろばんで使用されていた除法が現在のそろばん競技や検定において改めて採用されるのかについて検討していく。

第4章

除法

そろばんにおける除法は現在商除法のみ使用されている。かつて主な除法として使われていた帰除法は法が 1 桁の計算を簡略化するための手法であったという [6]。そのため、法が 1 桁と 2 桁以上で分けた二通りの解析を行う。

4.1 商除法

商除法は A を B で割って得られる商である C を立て、C と B の積を A から減らしていく手法である。疑似コードを用いて商除法について確認した後、法の桁数に場合分けをして解析を行う。これは帰除法が法の桁数で処理方法が大きく異なる手法であり、手法全体を通した解析による比較では不十分だと考えられるためだ。

手法 7 より、商除法は実と法の大小によって処理方法を二通りに分けている。実行開始位置を決定する設定 3(別表手法 11) もまた、実と法の大小で場合を分けている。これらの場合分けは、実行中の実 A と法の首位 b_1 の商の桁数が 1 桁か 2 桁かを決定するためのものである。b_1 の方が小さい時は a_i との商を、大きい時は a_ia_{i+1} との商を求めることで新たに置く商 c_h は 1 桁になる。a_ia_{i+1} は a_i を十の位、a_{i+1} を一の位として見た 2 桁の値と考える。この、6 行目もしくは 17 行目で商を求める時はかけ算九九の使用を想定する。

法が 1 桁であれば、a_i もしくは a_ia_{i+1} と b_j(法の首位であるため b_1) との商を設定 3 によって決定した実行開始位置に置く。この操作を商として必要な桁数 (今回は p) を得られるまで繰り返す。実際にそろばんを行う時は、方落とし法のように商の一の位と点が一致するように置いたり指を使って一の位を把握したりするが、本研究ではそれを p とすることで同じように把握するものと考える。また、問題によって割り切れなかった時や小数が発生する時の処理方法が複数存在するが、本研究では整数のみ扱うため小数位以下を切り捨てとして c_p までを求めることで計算を完了したこととする。

手法 7 現在使用されている割り算

```
1: procedure 商除法 (A = a_1 a_2 ... a_k, B = b_1 b_2 ... b_m)
2:     加算 (A) を実行
3:     設定 3(a_1, b_1) より実行開始位置を決定
4:     for h = 1 to p  //p は商の桁数
5:         if a_i ≥ b_1
6:             c_h = a_i/b_j とし、求めた c_h は実行開始位置に加える (加算 (c_h) の実行)
7:             for j = 1 to m
8:                 oldi = i
9:                 減算 (b_j · c_h) を実行
10:                i = i + 1
11:                j = j + 1
12:            h = h + 1
13:            i = oldi + 1
14:            実行開始位置を右に 1 つシフトする
15:            j = 1
16:        else
17:            c_h = (a_i a_{i+1})/b_j とし、求めた c_h は実行開始位置に加える (加算 (c_h) の実行)
18:            for j = 1 to m
19:                oldi = i
20:                減算 (b_j · c_h) を実行
21:                i = i + 1
22:                j = j + 1
23:            h = h + 1
24:            i = oldi + 1
25:            実行開始位置を右に 1 つシフトする
26:            j = 1
```

法が 2 桁以上の場合は、法の 2 位から末位まで商との積を実から引く。法の末位まで引くことができたら、次の商を立てて同様に法と商の積を実から引くということを商が p 桁になるまで繰り返す。法の 2 位以降と商との積が実から減らせない時は、還元と呼ばれる操作を実行する。具体的な処理方法は手法 13 の通りで、立てた商を 1 減らしてその分引いてしまったの積を A に加えることを実から積を引けるようになるまで繰り返し、その後再び手法 7 に戻る。本研究でこの還元を商除法の疑似コード内に組み込まなかったことは今回小数や補数を扱わなかった理由と同じく、最も基本的な計算処理のみを表すためである。そのため今回の商除法では、6 行目もしくは 17 行目で求められる c_h はそのまま答えとなる真商であると考えて解析を行う。また、還元を使用せずに全て引く前の状態に戻してから商を小さく立てて引き直すといった方法もあるため、法の 2 位以降が引けない場合の処理は受けた指導や計算者の好みもあると考えられる。

次に手法 7 の解析は、最初に法が 1 桁の時、次に法が 2 桁以上の時の順に行う。また 2 行目で実行される k 桁の実 A の加算と、商 c が合計 p 回加えられることは共通であるため、以降の解析では説明を省略する。

法が 1 桁である時、つまり $m = 1$ であり 7 行目から 11 行目もしくは 18 行目から 22 行目の繰り返しが 1 回で済むため最大 2 桁の減算が p 回実行される。本研究において商は整数であるため、最大の p は $k - m + 1$ で、この値を採用する。また、法が 1 桁であることに伴って 7 行目から 15 行目と 18 行目から 26 行目のうち、9,12,14,20,23,25 行目以外が不要となる。よって、法が 1 桁である時の最悪の加減回数は $k + 3p$ 回で、前述の説明に従って m と p に代入すると、$4k$ 回、合計 14 行を使用する。

法が 2 桁以上の場合は 7 行目から 11 行目もしくは 18 行目から 22 行目の繰り返しで最大 2 桁の減算が m 回あるため、最悪の加減回数は $k+p+2mp = 2km+2k+m-2m^2+1$ で、合計 26 行を必要とする。

4.2 帰除法

次に紹介する帰除法は過去盛んに使われていたものの、現在の商除法に以降してから全く用いられなくなった除算アルゴリズムである。計算には、本研究では「割算九九 (別表手法 15、手法 16)」、「見一 (別表手法 14」と呼ぶ割声 (割算をするときに唱える九九) を使う。これは曾我氏 [4] の解説を基に作成したが、今回疑似コードに適用するために法が 1 の時の処理を含めるなどの変更を一部加えている。

手法 8 かつて使用されていた割り算

```
1:  procedure 帰除法 (A = a_1 a_2 ... a_k, B = b_1 b_2 ... b_m)
2:      加算 (A) を実行
3:      設定 4(a_1, b_1, m) より実行開始位置を決定
4:      if m == 1   //法が 1 桁の場合
5:          for i = 1 to p + 1//p は商の桁数
6:              while a_i > b_1
7:                  a_i = a_i − b_1
8:                  左に 1 つシフトした場所で加算 (1) を実行
9:              割算九九 (a_i, b_1, m) を実行
10:             i = i + 1
11:             実行開始位置を右に 1 つシフトする
12:     else
13:         for i = 1 to p + 1
14:             oldi = i
15:             while a_i ≥ b_1
16:                 a_i = a_i − b_1
17:                 左に 1 つシフトした場所で加算 (1) を実行
18:                 for j = 2 to m
19:                     a_i = a_i − b_j
20:                     i = i + 1
21:                     j = j + 1
22:             i = oldi
23:             if a_i == b_1
24:                 見一 (a_i) を実行
25:             else
26:                 割算九九 (a_i, b1, m) を実行
27:             i = i + 1
28:             for j = 2 to m
29:                 減算 (a_i · b_j) を実行
30:                 i = i + 1
31:                 j = j + 1
32:             i = oldi + 1
33:             実行開始位置を右に 1 つシフトする
```

帰除法は手法 8 の通りで、p は商除法と同じように答えに必要な桁数と考えとするが、帰除法は繰り上がりの発生が常にあるため a_{p+1} まで求める。さらに帰除法は、法の桁数によって処理方法が変化する。法が 1 桁の場合は、a_i が法の首位である b_1 以下になるまで b_1 を減らし、その回数分十の位に 1 を加える。この時の一の位は実行中の a_i である。ここまでの操作は本来割算九九に収録され、五進一十のように n 進一

十 (n には 2 から 9 までが当てはまる) と表現される処理方法である。そして a_i が法の首位未満になると、割算九九 (別表手法 15 手法 16、長いため分割) を使用し、商となる。この 5 行目から 11 行目の処理は、更新した i が $p+1$ を越えるまで実行し続ける。この割算九九使用によって a_{i+1} の値が変化する場合、変化後の値で同じように計算を繰り返す。

法が 2 桁以上の場合は実行中の実 a_i が法の首位 b_1 より大きい時、十の位に 1 を加え、法 2 位以降も引いていく。その後、a_i が法の首位 b_1 と等しい場合、本研究では「見一 (別表手法 14)」と呼ぶ割り声を、等しくない場合は割算九九を用いて a_i を商とする。そして商と法 2 位以降の積を商除法 (手法 7) と同様に引いていく。この 14 行目からの処理は、更新した i が $p+1$ を越えるまで実行し続ける。法が 2 桁以上も同様に、割算九九使用によって a_{i+1} の値が変化する場合、変化後の値で同じように計算を繰り返す。法が 2 桁以上の帰除法もまた、商と法の積が引けない場合は還元 (別表手法 13) などを用いて処理を行う。

手法 8 の解析は、最初に法が 1 桁の時、次に法が 2 桁以上の時の順に行う。

法が 1 桁の時、6 行目から始まる n 進一十の処理は法が最低の値である 2 を、a_i が最大の値の 9 を取った時が最大の繰り返し回数で、4 回実行される可能性がある。つまり、十の位に 1 を足すことと a_i から b_1 を引くことが 4 回ずつ、これを $p+1$ 繰り返すため $(4+4)(p+1) = 8p+8$ 回。次に割算九九が実行される時、最大 a_i と a_{i+1} の 2 箇所に操作が実行されるため、$2p+2$ 回。よって法が 1 桁である時の最悪の加減回数は、$10p+10$ 回である。商除法と同様に $m=1$ と $p=k-m+1$ を代入すると、$10k+10$ 回、合計 11 行を使用する。

法が 2 桁以上の時、法が 1 桁の時と異なることが二つある。一つは法の首位が 1 の可能性もあること。もう一つは法の首位だけでなく 2 位以降も引く必要があること。よって 15 行目から始まる n 進一十の繰り返しは最大で 8 回実行される可能性がある (a_i が最大 9 を取り、b_1 が最小の 1 である時、8 回 1 を減らすと繰り返しの条件から外れる)。これらの前提と、法が 1 桁の時のカウント方法を合わせると、$(8+8+8(m-1))(p+1) = 8mp+8m+8p+8$ 回。見一もしくは割算九九の使用は同様に 2 箇所で、法 2 位以降の処理は最大 2 桁の減算が行われるため $(2+2(m-1))(p+1) = 2mp+2m$ 回。よって法が 2 桁以上である時の最悪の加減回数は、$10mp+10m+8p+8$ 回で同様に $p=k-m+1$ を代入すると、$10km+8k+12m-10m^2+16$ 回、合計 26 行を使用する。

今回の解析においてそれぞれ n 進一十が発生する最大の操作回数を考えたが、この操作を商除法と同じように商を立てるようにしてまとめて処理を行うこともできる。つまり、解析では 4 回繰り返された処理を 1 回にまとめることもできる。この時の加減回数は、法が 1 桁では $4p+4 = 4k+4$ 回、法が 2 桁以上では $3mp+3m+p+1 = 3km+k+5m-3m^2+2$ 回となる。この結果は本研究の方針と一致しないため、参考程度に留める。

4.3 結果

各除算アルゴリズムの解析結果は表 4.1 にまとめた通りである。法が 1 桁である時、つまり $m = 1$ の時の加減回数は $4k$ 回である商除法の方が少なくなり、必要な疑似コードの行数は 11 行である帰除法の方が少なくなった。

そして法が 2 桁以上である時、つまり $m \geq 2$ の時の加減回数は帰除法の結果から商除法の結果を引いた差が 0 より大きくなるような不等式を作成すると、$8km + 6k + 11m - 8m^2 + 15 > 0$ となり、この不等式が成り立つのであれば帰除法の方が回数は多いということになる。これを k について解くと、$k > (8m^2 - 11m - 15)/2(4m + 3)$ となり、分子を分母で割ると、$k > m - 2 - (m + 3)/(8m + 6)$ という不等式を得られる。この不等式から、k が $m - 2$ 未満の時、商除法が帰除法よりも加減回数が多くなる可能性はある。しかし、その場合は商が整数にならないため、本研究が定めた条件の範囲において帰除法は商除法よりも法の桁数を問わず最悪の加減回数は多くなる。また、必要な疑似コードの行数は共に 26 行である。

これらの得られた結果から、現在の珠算検定や競技において帰除法の再採用を提案することは難しいと言える。何故なら、法の桁数に関係なく最悪の加減回数が商除法よりも多くなるからだ。帰除法が得意とする法が 1 桁の計算も数字によっては商除法の倍以上加減回数を要求されることは、大きく計算速度に影響するだろう。

また必要とする疑似コードの行数から、法が 1 桁であるならば帰除法の方が修得は容易であると考えられるが、法が 2 桁以上にも対応すると合計 33 行になる。よって除法の修得という面からも、合計で 26 行で構成される商除法の方が総合的に見て容易であるとも言えるからだ。

	商除法	帰除法
行数 (法が 1 桁)	14	11
行数 (法が 2 桁以上)	26	26
最悪の加減回数 ($m = 1$)	$4k$	$10k + 10$
最悪の加減回数 ($m \geq 2$)	$2km + 2k + m - 2m^2 + 1$	$10km + 8k + 12m - 10m^2 + 16$

表 4.1: 各除算アルゴリズムの解析結果

第5章

まとめ

これまで現在珠算で使われている乗算と除算の他に、新たにもしくは再び手法を珠算に提案することができるのかについて検討して来た。ここで得られた成果と結論を述べ、今後の課題について考える。

5.1　本研究の成果と課題

本研究は、関連研究で示された計算アルゴリズムやコンピュータで利用されている手法が何故現在の珠算で使われていないのかを明らかにすることを目的に取り組んで来た。この目的達成のために、計算アルゴリズムを疑似コードに書き起こすこと、そしてそれをもとに解析と比較を行うことはこれまでになかった挑戦であり大きな成果であると感じる。今回は一部扱う内容などを狭めたものであったが、今回使用した用途以外にも珠算の指導で新たに疑似コードを用いることや、入力される自由な値に対応した複数の手法を表示するようなそろばんの教育ツールの開発などにも繋がると考えられる。また、そろばんは本来計算器としての役割を持っていたが、現代においてそのように使用されることは非常に稀である。つまり、ただの計算器としてではなく、本研究で行ったように疑似コードを読み取り、その解析を行うことで情報科学に触れるツールとして教育分野への貢献が期待できるのではないかと思われる。

そして乗法と除法の解析を行った結果、速さと修得の 2 つの面からどちらも現在使用されている手法に代わるものは無かったという結果を今回得られた。これは、最初に立てたこれまで提案されてきた手法の中で現在残っているものが珠算において最適な手法であるという仮説を裏付けるものである。

しかし、では何故方落とし法が珠算教育で第一に採用されているのか。これについては指導のしやすさが挙げられるのではないかと推測するが、本研究の基準ではそれを導くことはできない。また、今回扱わなかった処理や小数、他の計算アルゴリズムを含める場合はどのようになるのか。現在普及していない割算九九の修得のコストも考慮すべきかなどが今後の研究において重要な課題となるだろう。

参考文献

[1] 石戸珠算学園. いしど式で簡単 大人のそろばんドリル. メイツ出版社, 2019.

[2] 紺谷拓弥, 野呂正行. 多項式乗算の様々なアルゴリズムの比較. 数理解析研究所講究録, 1085 巻, pp. 140–150, 1999.

[3] 曽我和三郎. 帰除法から商除法への転換期とその背景. 珠算春秋, 第 66 巻 104 号, pp. 58–70, 2020.

[4] 曽我和三郎. 帰除法による計算の全容. 珠算春秋, 第 66 巻 104 号, pp. 50–57, 2020.

[5] たのしいそろばん（小学校用副教材）制作チーム. たのしいそろばん. 全国珠算教育団体連合会, https://syuzan-rengo.jp/reference/tanoshisoroban.pdf, 2024 年 11 月閲覧.

[6] 山崎与右衛門, 戸谷清一, 鈴木久男. 珠算算法の歴史. 森北出版社, 1958.

[7] 吉田光由. 塵劫記. 1627.

(別表-1)

手法 9 引き算の方法

```
1: procedure 減算 (A = a_1 a_2 ... a_k)
2:     設定 1(k) で実行開始位置を決定
3:     for i = 1 to k
4:         if a_i が引ける
5:             a_i を引く
6:             実行開始位置を 1 つ右にシフトする
7:             i = i + 1
8:         else if 1 ≤ a_i ≤ 4 and 五珠が引ける
9:             | 5 − a_i | を足す
10:            五珠を引く
11:            実行開始位置を 1 つ右にシフトする
12:            i = i + 1
13:        else
14:            10 を引く //十の位の一珠を引く
15:            | 10 − a_i | を足す
16:            実行開始位置を 1 つ右にシフトする
17:            i = i + 1
```

手法 10 両落とし法における実行開始位置の決定

```
1: procedure 設定 2(x,y)
2:     if (x + y) % 3 == 1
3:         実行開始位置は一の位
4:     else if (x + y) % 3 == 2
5:         実行開始位置は十の位
6:     else
7:         実行開始位置は百の位
```

手法 11 商除法における実行開始位置の決定

```
1: procedure 設定 3(x, y)
2:     if x ≥ y
3:         実行開始位置は x から左に 2 つシフトした場所
4:     else
5:         実行開始位置は x から左に 1 つシフトした場所
```

手法 12 帰除法における実行開始位置の決定

```
1: procedure 設定 4(x, y, z)
2:     if (x == 1 and x ≥ y) or (z ≥ 2and x > y)
3:         実行開始位置は x から左に 1 つシフトした場所
4:     else
5:         実行開始位置は x が置いてある場所
```

手法 13 商が引けない時の処理方法

```
1: procedure 還元 (A, a, b, c, j)
2:     a = a − b · c
3:     while a < 0
4:         c = c − 1 //c == 1 のとき 10 として扱う
5:         A = A + b_{1...(j−1)} //実 A に戻す
```

手法 14 法の首位と a_i が同じ時使う割算九九

```
1: procedure 見一 (x)
2:     x = 9
3:     右に 1 つシフトした場所で加算 (x)
```

(別表-2)

手法 15 帰除法で使用する割算九九 1

```
1: procedure 割算九九 (x, y, z)
2:     if y == 1 and z == 1//自明のためそのまま
3:     else if y == 1
4:     else if y == 2
5:         x = 5 //2 未満の処理になるため一通り

6:     else if y == 3
7:         if x == 1
8:             x = 3
9:             右に 1 つシフトして加算 (1) を実行
10:        else
11:            x = 6
12:            右に 1 つシフトして加算 (2) を実行

13:    else if y == 4
14:        if x == 1
15:            x = 2
16:            右に 1 つシフトして加算 (2) を実行
17:        else if x == 2
18:            x = 5
19:        else
20:            x = 7
21:            右に 1 つシフトして加算 (2) を実行

22:    else if y == 5
23:        x = x + x
24:    else if y == 6
25:        if x == 1
26:            右に 1 つシフトして加算 (4) を実行
27:        else if x == 2
28:            x = 3
29:            右に 1 つシフトして加算 (2) を実行
30:        else if x == 3
31:            x = 5
32:        else if x == 4
33:            x = 6
34:            右に 1 つシフトして加算 (4) を実行
35:        else
36:            x = 8
37:            右に 1 つシフトして加算 (2) を実行
```

手法 16 帰除法で使用する割算九九 2

```
1: procedure 割算九九 (x, y, z)
2:     if y == 7
3:         if x == 1
4:             右に 1 つシフトして加算 (3) を実行
5:         else if x == 2
6:             右に 1 つシフトして加算 (6) を実行
7:         else if x == 3
8:             x = 4
9:             右に 1 つシフトして加算 (2) を実行
10:        else if x == 4
11:            x = 5
12:            右に 1 つシフトして加算 (5) を実行
13:        else if x == 5
14:            x = 7
15:            右に 1 つシフトして加算 (1) を実行
16:        else
17:            x = 8
18:            右に 1 つシフトして加算 (4) を実行

19:    else if y == 8
20:        if x == 1
21:            右に 1 つシフトして加算 (2) を実行
22:        else if x == 2
23:            右に 1 つシフトして加算 (4) を実行
24:        else if x == 3
25:            右に 1 つシフトして加算 (6) を実行
26:        else if x == 4
27:            x = 5
28:        else if x == 5
29:            x = 6
30:            右に 1 つシフトして加算 (2) を実行
31:        else if x == 6
32:            x = 7
33:            右に 1 つシフトして加算 (4) を実行
34:        else
35:            x = 8
36:            右に 1 つシフトして加算 (6) を実行

37:    else if y == 9
38:        右に 1 つシフトして加算 (x) を実行
```