素朴な方法と Rabin–Karp 法による可逆文字列照合アルゴリズム *

谷崎 海良 † 平工 真基 † 横山 哲郎 ‡

概要

可逆性を有効に活用した計算システムでは，アルゴリズムも可逆であることが望ましいことが少なくない．本稿では素朴な方法と Rabin と Karp による文字列照合アルゴリズムの可逆版をそれぞれ作成し，時間計算量，空間計算量，及びゴミの量の解析を行う．可逆アルゴリズムは全ステップが単射でなければならない．我々は Rabin–Karp アルゴリズムで用いられるハッシュ値を更新する写像が単射であることを示す．素朴な方法の可逆版は（非可逆である）素朴な方法と時間計算量が同じであるものを作成した．可逆な Rabin–Karp アルゴリズムは，前処理と後処理を除いた計算の時間計算量が非可逆なものと同じものを作成した．両者とも，空間計算量は対応する非可逆アルゴリズムと同じであり，ゴミは入力のみである．提案した可逆アルゴリズムは可逆プログラミング言語 Janus を用いて実装して動作を確認した．

1 はじめに

可逆計算は任意の状態において直前と直後の状態がそれぞれ高々一意に定まる計算のことである（例えば [14]）．つまり，可逆計算は単射部分写像を表す [4, 1]．Landauer の原理 [8] によると非可逆な計算を行うと必ず環境に正の熱が散逸する．しかし，可逆計算では理論的にはその熱量の下限はゼロである [3]．計算の可逆性は，情報の消去が問題をもつと見なされる量子的及び生物的な計算モデルで必要とされることがあり [7]，並列離散事象シミュレーションの高速化のために使われている [5]．可逆計算システムにおいて使用される可逆アルゴリズムの設計及び実装には，通常のアルゴリズムを対象とするときとは異なる考え方が必要である．

任意のアルゴリズムを可逆化する手法が知られている．例えば，埋込み法 [8]，call–copy–uncall 法 [3, 4]，Lange–McKenzie–Tapp 法 [9] やそれらの拡張が知られている．しかし，元のアルゴリズムに対して線形時間かつ線形空間の可逆シミュレーションを得る可逆化手法が存在するかは未解決問題 [4] であるし，これらのような現在知られている可逆化手法は時間計算量，空間計算量を悪化させたり大量のゴミを生じさせたりする問題がある．一方で，こうした事実は個別の解法の効率化の妨げとはならない．実際，これまで比較整列法など特定の非可逆なアルゴリズムを効率的な可逆シミュレーションに可逆化する方法が研究がされてきた [2]．

文字列照合アルゴリズムは，ファイルから特定の文字列を検索したり DNA 配列の中から特定のパターンを探し出したりするなど様々な場面で有用なアルゴリズムであり，様々なアルゴリズムの一部として使われる基本的なアルゴリズムである．本研究では文字列照合アルゴリズムの効率的な可逆シミュレーションを作成する．特に，基本的なアルゴリズムである素朴な方法と Rabin–Karp アルゴリズムを対象とする [6]．これらが様々な可逆アルゴリズムの一部として使われたり，これらの可逆シミュレーションの作成法の知見が他でも生

---

* Reversibilization of the Naive String-Match Algorithm and the Rabin–Karp Algorithm

† 南山大学理工学部ソフトウェア工学科

‡ 南山大学理工学部電子情報工学科

かされることが期待される．

可逆シミュレーションは引数渡し機構をもつ可逆プログラミング言語 Janus を用いて実装する [10, 12]．実装したプログラムを元に，これらの可逆シミュレーションの時間計算量，空間計算量，及びゴミの量を解析する．本稿のプログラムは `https://tetsuo.jp/ref/nanzan2022/` から手に入れることができ，オンラインインタプリタで動作を確かめることができる．

2 可逆文字列照合アルゴリズム

テキストからあるパターンの出現を全て求める問題を文字列照合問題といい，本稿では次のように定式化する．有限のアルファベット Σ を考える．Σ に属する文字のみを要素にもつ配列を文字列という．テキストを長さ n の文字列 $t[0..n-1]$，パターンを長さ m の文字列 $p[0..m-1]$ とする．任意の s $(0 \leq s \leq n-m)$ に対して，$t[s..s+m-1] = p[0..m-1]$ ならば，パターン p はテキスト t のシフト s に出現するという．p が t のシフト s に出現するならば，s を正当なシフトという．

通常，アルゴリズムは単射でないステップをもつ．全ステップが単射であるアルゴリズムを可逆という．アルゴリズム A が写像 f を計算し，任意の x に対して可逆アルゴリズム A' が出力に $f(x)$ を他の出力と区別できるように含むならば，A' を A の可逆シミュレーションといい，$f(x)$ 以外の出力をゴミという．与えられたアルゴリズム A から A の可逆シミュレーション A' を作成することを可逆化という．

文字列照合問題の入力はテキスト t とその長さ n 及びパターン p とその長さ m であり，出力は全ての正当なシフトである．これを表す写像[*1]は明らかに単射では無く，単射ステップのみからなる可逆アルゴリズムで実現することはできない．しかし，文字列照合アルゴリズムが使われる場合，入力がそのままメモリに保持され他でも使用されることは多い．また，量子回路も全ステップが単射であり，元の入力を出力に含むものが使われることは多い（量子加算回路の例 [11]）．以降では，通常の文字列照合問題の元の出力に加えて，元の入力（テキスト t とその長さ n 及びパターン p とその長さ m）を追加で出力する問題を考える．言い換えると，ゴミが元の入力のみである文字列照合アルゴリズムの可逆シミュレーションを考える．

埋込み法や call–copy–uncall 法では, 元の時間計算量 T と空間計算量 S に対して, 空間計算量 $\mathrm{O}(\max(T, S))$ となり，オンラインアルゴリズムではなくなる [3]．現在知られている call–copy–uncall 法の改良版でも時間計算量及び空間計算量の少なくともいずれか一方のオーバヘッドをもつ．Lange–McKenzie–Tapp 法では，時間計算量のオーバヘッドがある [9]．本稿ではこうした一般解法よりも効率の良い可逆文字列照合アルゴリズムを求める．

2.1 素朴な方法の可逆版

文字列照合問題では，p が t のシフト $s = 0, 1, \ldots, n-m$ にそれぞれ出現するか，すなわち $p[0..m-1] = t[s..s+m-1]$ であるかを確かめれば良い．素朴な方法では，素朴にそれぞれの s $(0 \leq s \leq n-m)$ について p と $t[s..s+m-1]$ の全要素が一致するかをそれぞれ照合することで正当なシフトを全て求める．

素朴な方法を可逆的に実現したプロシージャ `naivesearch` を示す．

```
1 procedure naivesearch(int t[], int n, int p[], int m, stack r)
2     local int s = 0
3     from s = 0 do
```

[*1] 簡潔にするため，以降では部分写像のことを単に写像と呼ぶ．

```
4         call naivematch(t, n, p, m, s, r)
5         s += 1
6     until s = n-m+1
7     delocal int s = n-m+1
```

naivesearch は長さ n のテキスト t と長さ m のパターン p から，初期状態では nil をもつスタック r に全ての正当なシフトをプッシュする．実行中に入力を保持する変数 t, n, p, m の値は更新しない．2 行目では，シフトを保持するための変数 s を確保する．3–6 行目のループ中は，0, 1, 2, ..., $n - m$ と変化させていったそれぞれの s について，t[s..s+m-1] と p が照合するかを 4 行目の naivematch の呼出しで確かめている．ループが終了すると 7 行目で s は n-m+1 であることが確認されて解放される．

プロシージャ naivematch では，t[s..s+m-1] と p が照合するか確かめる，すなわち s が正当なシフトである場合にスタック r に s をプッシュする．

```
1  procedure naivematch(int t[], int n, int p[], int m, int s, stack r)
2      local int i = 0
3      local int f = 1
4      call nm(t, n, p, m, s, i, f) // 1.call
5      if f = 1 then // 照合成功
6          local int tmp = s
7          push(tmp,r) // 2.copy
8          delocal int tmp = 0
9      fi f = 1
10     uncall nm(t, n, p, m, s, i, f) // 3.uncall
11     delocal int f = 1
12     delocal int i = 0
```

本体の先頭（2–3 行目）と末尾（11–12 行目）では，局所変数 i と f の確保と解放が行われる．その内側では，call–copy–uncall 法 [3] が用いられている．すなわち，次の 3 ステップが行われる．第 1 に，4 行目のプロシージャ nm の呼出し（call）ではゴミを i に保持しながら照合の結果を f に保持する．第 2 に，照合が成功していれば 5–9 行目の条件文の then 節内で結果をスタック r に保存する．具体的には，7 行目で正当なシフト s をスタック r にプッシュする．第 3 に，10 行目の逆呼出し（uncall）で i に保持していたゴミをクリアする．

ゴミを i に保持しながらパターン p と部分文字列 t[s..s+m-1] の照合の結果を f に保持するプロシージャ nm は，以下の通りである．

```
1  procedure nm(int t[], int n, int p[], int m, int s, int i, int f)
2      from i = 0 do
3          if t[s+i] != p[i] then
4              f ^= 1
5          fi f = 0
6          i += 1
7      until f = 0 || i = m
```

初期状態ではフラグ f は真を表す 1 を保持している．カウンタ変数 i を増分（+1）していき，パターンと部分文字列の対応する要素が一致しないとき，すなわち 3 行目で t[s+i] != p[i] が成り立ったとき 4 行目でフラグ f をフリップして偽を表す 0 にする．ループは照合の成否が決定したときに終了する．つまり，フラグ

f が偽を表す 0 であるか，パターンと部分文字列の末尾まで確認が終わったときに終了する．

2.2 可逆 Rabin–Karp アルゴリズム

Rabin–Karp アルゴリズムでは文字列を整数に変換するハッシュ関数 a を用いて文字列照合を行う．Rabin–Karp アルゴリズムでは，テキスト t とその長さ n, パターン p とその長さ m, アルファベットのサイズ d, 法 q が入力として与えられ，全ての正当なシフトを出力する．ハッシュ関数を

$$a(p) = h(p_0p_1p_2\cdots p_{m-1}) = (p_0d^{m-1} + p_1d^{m-2} + p_2d^{m-3} + \cdots + p_{m-1}) \bmod q \tag{1}$$

とする．ここで p_i は文字列 p の i 文字目を表す．t 中の長さ m の部分文字列も同様に整数 $a(t_it_{i+1}\cdots t_{i+m-1})$ に変換する $(0 \le i \le n-m)$．q で割った余りで考えるので 2 つの文字列 p，$t_it_{i+1}\cdots t_{i+m-1}$ は一致していないがハッシュ値が等しくなることがある．しかし，この 2 つの文字列が一致していているのにハッシュ値が異なることはない．したがって，2 つのハッシュ値が等しい場合のみ 2 つの文字列が一致しているかを素朴な方法を用いて確かめることで p が t のどこに存在しているかを調べることができる．

Rabin–Karp アルゴリズムでは以下のステップを行う．

1. 初期化：$h = d^{m-1}$, $h_p = a(p)$, $h_t = a(t_0t_1\cdots t_{m-1})$ の値を求める．$i = 0$ とする．
2. $h_p = h_t$ の場合，素朴な方法を用いて文字列を比較する．
3. $i < n - m$ の場合，h の値を用いて h_t の値を $a(t_{i+1}t_{i+2}\cdots t_{i+m})$ に更新して，i を増分して 2 に戻る．

更新前のハッシュ値を $h_t = a(t_it_{i+1}\cdots t_{i+m-1})$, 更新後のハッシュ値を $h'_t = a(t_{i+1}t_{i+2}\cdots t_{i+m})$ とする．このとき更新前後のハッシュ値の関係は

$$\begin{aligned} h'_t &= f(h_t, i) && (2)\\ &= (d(h_t - t_id^{m-1}) + t_{i+m}) \bmod q && (3)\end{aligned}$$

である．d と q が互いに素であるとき，f は第 1 引数について単射である（付録 A の補題 3 を参照）．以降では，d と q が互いに素であるときのみを考える．

f は第 1 引数について単射であるので次が成り立つ．

補題 1. d と q を互いに素な正整数とする．写像

$$\begin{aligned} f(h_t, i) &= (d(h_t - t_id^{m-1}) + t_{i+m}) \bmod q && (4)\\ g(h'_t, i) &= d^{q-2}(h'_t + t_id^m - t_{i+m}) \bmod q && (5)\end{aligned}$$

を考える．任意の整数 h_t $(0 \le h_t \le q-1)$, 非負整数 i, 正整数 m, 長さ n $(i + m \le n - 1)$ の文字列 t について $g(f(h'_t, i), i) = 1$ が成り立つ．

証明.

$$\begin{aligned} g(f(h_t,i),i) &= d^{q-2}(((d(h_t - t_id^{m-1}) + t_{i+m}) \bmod q) + t_id^m - t_{i+m}) \bmod q && \because \text{式 4 と式 5}\\ &= (d^{q-2}((d(h_t - t_id^{m-1}) + t_{i+m}) + t_id^m - t_{i+m}) \bmod q) \bmod q\\ &= (d^{q-1}h_t \bmod q) \bmod q\\ &= d^{q-1}h_t \bmod q\\ &= h_t \bmod q && \because \text{フェルマーの小定理}\\ &= h_t && \because 0 \le h_t \le q-1\end{aligned}$$

□

Rabin–Karp アルゴリズムの可逆版のプロシージャ rabinkarp を示す.

```
1  procedure rabinkarp(int t[], int n, int p[], int m, int d, int q, stack r)
2      local int h = 0
3      local int mod = 0
4      call pow(d, m-1, h) // h = d^{m-1}
5      call pow(d, q-2, mod) // mod = d^{q-2}
6      call rk(n, m, t, p, d, q, h, mod, r)
7      uncall pow(d, q-2, mod) // zero clear mod
8      uncall pow(d, m-1, h) // zero clear h
9      delocal int mod = 0
10     delocal int h = 0
```

ここで，入力はテキスト t とその長さ n, パターン p とその長さ m, アルファベットのサイズ d, 法 q であり，初期状態ではスタック r は空である．実行が終わると入力はそのままであり，スタック r は全ての正当なシフトをもつ．h と mod を局所変数として用いるために本体の先頭（2–3 行目）と末尾（9–10 行目）でそれぞれ確保と解放を行う．4–5 行目では，累乗を計算するプロシージャ pow を用いて，d の m-1 乗を h に，d の q-2 乗を mod にそれぞれ求める．この前処理には $\Theta(m)$ の実行時間がかかる．6 行目では，補助プロシージャ rk を呼び出す．7–8 行目では，h と mod をゼロクリアする．非可逆な Rabin–Karp アルゴリズムでは不要な後処理であるが，実行時間は前処理と同じ $\Theta(m)$ であり漸近的な複雑度を悪化させるものでは無い.

x の n 乗をゼロクリアされた res に求めるプロシージャ pow は以下の通りである.

```
1  procedure pow(int x, int n, int res) // res = x^n, n>=0
2      res ^= 1
3      local int i = 0
4      from i = 0 loop
5          local int tmp = res*x
6          res <=> tmp
7          delocal int tmp = res/x
8          i += 1
9      until i = n
10     delocal int i = n
```

5–7 行目には単射写像による更新のコードパターン

```
local int tmp = φ(w)
w <=> tmp
delocal int tmp = φ^{-1}(w)
```

が出現している．これは写像 ϕ が単射であるとき，変数 w を値 ϕ(w) に更新する．上記のコード中では，ϕ(res) = res*x であり逆写像 ϕ^{-1}(res) = res/x が存在し，このコードパターンで res を x 倍にする．この乗算は単射ではなく，実際 res が 0 のときやオーバーフローが起きたときに 7 行目の解放が失敗する．本稿の考えている範囲では，簡単のため，乗算によるオーバーフローは起こらないと仮定する.

補助プロシージャである rk を図 1 に示す．2–35 行目では前処理を行う．h_t を h(t[0..m-1]), h_p を

```
1  procedure rk(int n, int m, int t[], int
       p[], int d, int q, int h, int mod,
       stack r)
2    local int h_t = 0
3    local int h_p = 0
4    local int i = m-1
5    local int tmp = 1
6    from i = m-1 do
7      h_p += p[i]*tmp
8      h_t += t[i]*tmp
9    loop
10     local int tmps = tmp*d // tmp *= d
11     tmp <=> tmps
12     delocal int tmps = tmp/d
13     i -= 1
14   until i = 0
15   delocal int tmp = h
16   delocal int i = 0
17   local int tmpt = h_t%q
18   local int tmpp = h_p%q
19   tmpt <=> h_t
20   tmpp <=> h_p
21   local int i = m-1
22   local int tmp = 1
23   from i = m-1 do // clear tmpp & tmpt
24     tmpp -= p[i]*tmp
25     tmpt -= t[i]*tmp
26   loop
27     local int tmps = tmp*d
28     tmp <=> tmps
29     delocal int tmps = tmp/d
30     i -= 1
31   until i = 0
32   delocal int tmp = h
33   delocal int i = 0
34   delocal int tmpp = 0
35   delocal int tmpt = 0
36
37   local int i = 0
38   from i = 0 do
39     if h_p = h_t then // pseudo-match
40       call naivematch(t,n,p,m,i,r)
41     fi h_p = h_t
42   loop
43     local int tmp = (d*(h_t-t[i]*h)+t[i+m
         ]) % q // calculate f
44     tmp <=> h_t
45     delocal int tmp = (mod*(h_t+t[i]*h*d-
         t[i+m])) % q // calculate f^{-1}
46     i += 1
47   until i = n-m
48   delocal int i = n-m
49
50   local int tmpp = 0
51   local int tmpt = 0
52   local int i = n-1
53   local int j = m-1
54   local int tmp = 1
55   from j = m-1 do
56     tmpp += p[j]*tmp
57     tmpt += t[i]*tmp
58   loop
59     local int tmps = tmp*d
60     tmp <=> tmps
61     delocal int tmps = tmp/d
62     i -= 1
63     j -= 1
64   until j = 0
65   delocal int tmp = h
66   delocal int j = 0
67   delocal int i = n-m
68   h_p -= tmpp%q // clear h_p
69   h_t -= tmpt%q // clear h_t
70   local int i = n-1
71   local int j = m-1
72   local int tmp = 1
73   from j = m-1 do // clear tmpp & tmpt
74     tmpp -= p[j]*tmp
75     tmpt -= t[i]*tmp
76   loop
77     local int tmps = tmp*d
78     tmp <=> tmps
79     delocal int tmps = tmp/d
80     i -= 1
81     j -= 1
82   until j = 0
83   delocal int tmp = h
84   delocal int j = 0
85   delocal int i = n-m
86   delocal int tmpt = 0
87   delocal int tmpp = 0
88   delocal int h_p = 0
89   delocal int h_t = 0
```

図 1　可逆 Rabin–Karp アルゴリズムの補助プロシージャ

$h(\mathtt{p})$ にする．

37–48 行目で主な処理を行う．`i` を 0 から `n-m` まで増分していき，39 行目でテキストとパターンのハッシュ値が等しいかテストし，等しい場合（擬ヒットした場合）は 40 行目で素朴な方法で照合する．`i` が `n-m` 未満ならば 43–45 行目の単射写像 f による更新のコードパターンでハッシュ値 `h_t` を $f(\mathtt{h\_t}, \mathtt{i})$ に更新する．具体的には 43 行目の右辺で写像 f の値を計算し，45 行目の右辺で写像 $g(\mathtt{h\_t}, \mathtt{i})$ の値を計算して更新する．その後に，`i` を増分する．

50–89 行目では後処理を行う．具体的には，`h_t` と `h_p` をゼロクリアして解放する．

単射写像 $\phi(\mathtt{tmp}) = \mathtt{tmp*d}$ による更新のコードパターンが 10–12 行目，27–29 行目，59–61 行目，及び 77–79 行目で出現している．

3　まとめ

本研究は文字列照合法のうち素朴な方法と Rabin–Karp 法の可逆シミュレーションの作成・解析・実装を行った．非可逆なアルゴリズムと可逆化したアルゴリズムの時間計算量，空間計算量，及びゴミの量を解析し非可逆なアルゴリズムと可逆なアルゴリズムの比較を行った．可逆 Rabin–Karp アルゴリズムはハッシュ値を更新する写像が第 1 引数について単射であることを示した．本研究で行った可逆化は他のアルゴリズムの可逆化でも応用できることが期待できる．

提案する素朴な方法の可逆版と可逆 Rabin–Karp アルゴリズムでは，非可逆なものと同様に，照合にかかる実行時間が $\mathrm{O}((n-m+1)m)$, メモリ使用量が $\mathrm{O}(n+m)$ である．また，2 つの可逆アルゴリズムのゴミは入力のみとなった．プロシージャ `nm` は 2 パスであるが，プロシージャ `naivesearch` 中で使用されるシフト `s` は 0 から `n-m+1` まで単調に増加していた．したがって，可逆版もオンラインアルゴリズムといえる．

提案手法には次のように改善できる点がいくつかある．本稿では乗算のたびに q の剰余を計算していないが，オーバーフローを防ぐためには演算を行うたびに剰余を計算することが望ましい．Rabin–Karp アルゴリズムの前処理と後処理における累乗計算はより効率的な方法が知られているが，対応する可逆アルゴリズムを空間複雑度が高くせずに実現する方法が不明である．また，アルゴリズムを整理して可読性を増しモジュール性を高める余地がある．例えば，前処理と後処理のルーチンは共通する処理が行われており，文献 [12] 以降で拡張された Janus の処理系の機構を用いたりさらに拡張したりして，抽象化して同一プロシージャで実現されることが望ましい．複数パターンを同時に探す場合に Rabin–Karp アルゴリズムは効果が高い．この場合に拡張することは容易であると予想される．これらの改善は今後の課題である．

文字列照合を行う基本的なアルゴリズムは他にも存在しており，本稿のような可逆シミュレーションがどれほど効率的に実現できるかは現時点ではよく分からない．例えば，有限オートマトンを用いる方法や Knuth–Moris–Patt アルゴリズムは単射でないステップをもつので，本稿のような可逆化を直接的に行った場合にオンラインであることを保つのは難しい．

本稿は，文献 [13] の内容が基になっておりその一部を改良したものである．本稿では引数渡し機構をもつ可逆プログラミング言語 Janus を用いた．Janus の引数渡し機構については文献 [15] でも拡張され考察されている．

■謝辞　本研究は，2013 年度から横山研究室の卒業研究・修士研究によって実施された成果を基にしたものである．これらの研究の多くに早期段階から発表会等で有益な助言をいただいた青山幹雄教授に研究室一同・卒業生一同心より感謝申し上げる．本研究は，2021 年度パッヘ研究奨励金 I-A-2 の助成を受けた．

参考文献

[1] Axelsen, H. B. and Glück, R.: What Do Reversible Programs Compute?, *Foundations of Software Science and Computation Structures. Proceedings* (Hofmann, M., ed.), Lecture Notes in Computer Science, Vol.6604, Springer, pp. 42–56 (online), doi:10.1007/978-3-642-19805-2_4 (2011).

[2] Axelsen, H. B. and Yokoyama, T.: Programming Techniques for Reversible Comparison Sorts, *Programming Languages and Systems* (Feng, X. and Park, S., eds.), Lecture Notes in Computer Science, Vol.9458, Springer-Verlag, pp. 407–426 (online), doi:10.1007/978-3-319-26529-2_22 (2015).

[3] Bennett, C. H.: Logical Reversibility of Computation, *IBM Journal of Research and Development*, Vol.17, No.6, pp. 525–532 (online), doi:10.1147/rd.176.0525 (1973).

[4] Bennett, C. H.: Time/space trade-offs for reversible computation, *SIAM Journal on Computing*, Vol.18, No.4, pp. 766–776 (online), doi:10.1137/0218053 (1989).

[5] Carothers, C. D., Perumalla, K. S. and Fujimoto, R. M.: Efficient Optimistic Parallel Simulations Using Reverse Computation, *ACM Transactions on Modeling and Computer Simulation*, Vol.9, No.3, pp. 224–253 (online), doi:10.1145/347823.347828 (1999).

[6] Cormen, T. H., Leiserson, C. E., Rivest, R. L. and Stein, C.: *Introduction to Algorithms*, MIT Press, 3rd edition (2009). 浅野哲夫，岩野和生，梅尾博司，山下雅史，和田幸一（訳）：アルゴリズムイントロダクション，第 3 版，近代科学社 (2013).

[7] Krakovsky, M.: Taking the Heat, *Communications of the ACM*, Vol.64, No.6, pp. 18–20 (online), doi:10.1145/3460214 (2021).

[8] Landauer, R.: Irreversibility and Heat Generation in the Computing Process, *IBM Journal of Research and Development*, Vol.5, No.3, pp. 183–191 (online), doi:10.1147/rd.53.0183 (1961).

[9] Lange, K.-J., McKenzie, P. and Tapp, A.: Reversible space equals deterministic space, *Journal of Computer and System Sciences*, Vol.60, No.2, pp. 354–367 (online), doi:10.1006/jcss.1999.1672 (2000).

[10] Lutz, C.: Janus: A time-reversible language (1986). Letter to R. Landauer. `https://tetsuo.jp/ref/janus.html`

[11] Orts, F., Ortega, G., Combarro, E. and Garzón, E.: A review on reversible quantum adders, *Journal of Network and Computer Applications*, Vol.170, p. 102810 (online), doi:10.1016/j.jnca.2020.102810 (2020).

[12] Yokoyama, T., Axelsen, H. B. and Glück, R.: Principles of a Reversible Programming Language, *Computing Frontiers. Proceedings*, ACM Press, pp. 43–54 (online), doi:10.1145/1366230.1366239 (2008).

[13] 平工真基，谷崎海良：可逆な文字列照合アルゴリズム：力任せ法と Rabin–Karp 法を対象として，南山大学 2021 年度卒業論文 (2022).

[14] 森田憲一：可逆計算，近代科学社 (2012).

[15] 新海由侑，田中秀明，横山哲郎：可逆プログラミング言語の引数渡し機構の拡張，情報処理学会論文誌プログラミング（PRO），Vol.7, No.4, pp. 21–36 (2014). `http://id.nii.ac.jp/1001/00102871/`

付録 A　補題

本稿で用いた補題とその証明を付録として掲載する．

補題 2 (剰余算の性質)**.** d と q を互いに素な正整数とする．任意の整数 x, y, a, b について以下が成り立つ:

$$x + a \equiv y + b \pmod{q} \iff x \equiv y \pmod{q} \tag{6}$$

$$dx \equiv dy \pmod{q} \iff x \equiv y \pmod{q} \tag{7}$$

証明. 式 7 のみを示す:

$$\begin{aligned}
& dx \equiv dy \pmod{q} \\
\iff\ & d(x-y) \equiv 0 \pmod{q} \\
\iff\ & x - y \equiv 0 \pmod{q} && \because d \text{ と } q \text{ は互いに素} \\
\iff\ & x \equiv y \pmod{q}
\end{aligned}$$

□

補題 3 (f の単射性)**.** d と q を互いに素な正整数とする．整数 h_t $(0 \le h_t \le q-1)$, 非負整数 i, 正整数 m, 長さ n $(i + m \le n - 1)$ の文字列 t について写像

$$f(h_t, i) = (d(h_t - t_i d^{m-1}) + t_{i+m}) \bmod q \tag{8}$$

は第 1 引数について単射である．

証明. $f(h_t, i) = f(h_p, i)$ ならば $h_t = h_p$ を示す.

$$\begin{aligned}
& f(h_t, i) = f(h_p, i) \\
\iff\ & (d(h_t - t_i d^{m-1}) + t_{i+m}) \equiv (d(h_p - t_i d^{m-1}) + t_{i+m}) \pmod{q} && \because f \text{ の定義} \\
\iff\ & d(h_t - t_i d^{m-1}) \equiv d(h_p - t_i d^{m-1}) \pmod{q} && \because \text{式 6} \\
\iff\ & h_t - t_i d^{m-1} \equiv h_p - t_i d^{m-1} \pmod{q} && \because \text{式 7} \\
\iff\ & h_t \equiv h_p \pmod{q} && \because \text{式 6} \\
\iff\ & h_t = h_p && \because 0 \le h_t, h_p \le q-1
\end{aligned}$$

□