# アルゴリズムとデータ構造

## アルゴリズムとは早く止まる漸化式

森 立平
mori@c.titech.ac.jp

2018 年 6 月 12 日

今日のメッセージ

- **アルゴリズム ≈ 早く止まる漸化式**
- **漸化式は「1」でなくて「0」で止める**
- 漸化式の中の条件の数は減らす
- **最大公約数は時間計算量 $O(n^2)$ で計算できる**

今日の演習の目標

- ユークリッドの互除法のプログラムを書けるようになる
- Binary GCD のプログラムを書けるようになる

今日の主な演習課題 (提出締切は)

1. ユークリッドの互除法のプログラムを完成させる
2. Binary GCD のプログラムを完成させる

今日の演習時間のワークフロー

1. GitHub のアカウントを取得し、Google Form で情報を送る (配布資料参照)
2. `https://github.com/alg2018/` にアクセスし、`Algorithms-Datastructures` の「プログラミング演習のルール」を読む
3. `https://github.com/alg2018/` にアクセスし、`GCD` にある課題に取り組む
4. プログラムを紙に書いて教員か TA に提出
5. `Algorithms-Datastructures/docs/git.pdf` を見て、`git` の使い方を覚える (来週以降に `git` を使用する)

# 1　アルゴリズムとは

# 2　ユークリッドの互除法

自然数 $x$ と $y$ の最大公約数 (greatest common divisor; GCD) をもとめる問題を考えよう。おそらくみんな知っているユークリッドの互除法という「アルゴリズム」がある。ユークリッドの互除法は次の等式に基づく。

$$\gcd(x,y) = \begin{cases} x, & \text{if } x = y \\ \gcd(x, y-x), & \text{if } x \leq y \\ \gcd(x-y, y), & \text{otherwise.} \end{cases} \tag{1}$$

この等式が正しいことは $x \leq y$ のとき、$(r \mid x) \wedge (r \mid y) \iff (r \mid x) \wedge (r \mid y-x)$ から明らか。上記の等式は任意の自然数 $x$ と $y$ について $\gcd(x,y)$ の値を一意に定める。例えば $x = 6, y = 15$ の場合、$\gcd(6,15) = \gcd(6,9) = \gcd(6,3) = \gcd(3,3) = 3$ と最大公約数が定まる。このように、上記の等式を繰り返し適用 (左辺を右辺で置き換える) していくと**必ず停止する**。このような等式のことを「**漸化式**」と呼ぶことにしよう。**漸化式はそのままアルゴリズムになる** (効率がいいかどうかは別として)。漸化式を立てると自然とアルゴリズムが設計できる訳である。

次にこのアルゴリズムの実行時間を考えてみよう。アルゴリズムの実行時間は基本的に「漸化式を適用する回数 $\times 1$ 回の漸化式の適用にかかる時間」で評価できる。まずは「漸化式を適用する回数」について考えよう。上記のユークリッドの互除法の場合、$\gcd(n,1)$ を計算しようとすると $n-1$ 回漸化式を適用する必要がある。$x \leq y$ のときは、$y < x$ になるまで $y$ から $x$ を引くので、$y - x$ を $y \,\%\, x$ に置き換えることができる。

$$\gcd(x,y) = \begin{cases} y, & \text{if } x \,\%\, y = 0 \\ x, & \text{if } y \,\%\, x = 0 \\ \gcd(x, y \,\%\, x), & \text{if } x \leq y \\ \gcd(x \,\%\, y, y), & \text{otherwise.} \end{cases} \tag{2}$$

この漸化式を使えば、$\gcd(n,1)$ は直接 1 と計算できる。このアルゴリズムの計算量は次章で詳しく評価することにして、もう少しこの漸化式を改善してみよう。今まで $x$ と $y$ を自然数としていたが、0 で漸化式を止めることにすると次の漸化式が得られる。

$$\gcd(x,y) = \begin{cases} y, & \text{if } x = 0 \\ x, & \text{if } y = 0 \\ \gcd(x, y \,\%\, x), & \text{if } x \leq y \\ \gcd(x \,\%\, y, y), & \text{otherwise.} \end{cases} \tag{3}$$

このようにしても今までと同じ計算結果になることは簡単に確認できる。また、$\gcd(x,0) = x$ というように**定義域が拡張される**がこれは $x \geq 1$ のときは妥当な定義であろう ($x$ と 0 の最大公約数は $(r \mid x) \wedge (r \mid 0)$ を満たす最大の整数 $r$ と考えれば、これは $x$ である)。しかし一方で $\gcd(0,0) = 0$ ということになる。これは一見すると不自然な定義にも見える ($\gcd(0,0) = \infty$ としたいところであろう)。これを正当化するには 2 通り方法がある。1 つ目は $x \preceq y \overset{\text{def}}{\iff} x \mid y$ と半順序を定義した場合に 0 は「最大」の数である。また、半順序の公理として $x \preceq x$ は任意の $x$ について成り立たないといけないので、$0 \mid 0$ を仮定することになる。これは $x \mid y \overset{\text{def}}{\iff} \exists z \in \mathbb{Z}, y = zx$ と考えれば自然であろう。そうすれば、$\gcd(0,0) = 0$ は正当化できる。もう一つの正当化の方法として、$\gcd(0,0) = 0$ とおけば、$(\mathbb{Z}_{\geq 0}, \gcd)$ は 0 を単位元とするモノイド (逆元を持つとは限らない群) となることが挙げられる。上のように、漸化式の停止条件を 1 ステップ遅らせることで、**関数の定義域が自然に拡張**された。この拡張された定義域を持つ関数を使ってアルゴリズムを設計すると、漸化式の適用回数が 1 回増えるが、**似たような計算を 2 箇所に書くことを防げる**ことが多い。そのため、「漸化式はできるだけ簡単なところで止める」、「**漸化式は「1」でなくて「0」で止める**」というのは漸化式を立てるコツである。また、もう一つのコツとして漸化式における**条件分岐は計算量を大きく増やさない範囲で減らした方がよい**。条件分岐を減ら

すことで最終的に次の漸化式が得られる。

$$\gcd(x,y) = \begin{cases} y, & \text{if } x = 0 \\ \gcd(y \,\%\, x, x), & \text{otherwise.} \end{cases} \tag{4}$$

この漸化式では $x \le y$ を期待しているが、$x > y$ の場合は単に引数を入れ替える操作をしている。例えば $x = 15, y = 6$ の場合、$\gcd(15,6) = \gcd(6,15) = \gcd(3,6) = \gcd(0,3) = 3$ と計算できる。条件分岐を減らしたせいで最初の 1 回分漸化式の適用が増えている。しかし、条件分岐を減らすことで通常は全体の計算時間は短くなる。ここまで漸化式を変形してきたが、(2)から(4)まではアルゴリズムの実行時間は大きくは変わらない。しかし、「アルゴリズムの簡潔さ」と「アルゴリズムの実行時間の速さ」を両立する(4)が一番優れている。このような漸化式を立てられるようになるため、この 2 つのコツを押さえておく必要がある。

- **漸化式は「1」でなくて「0」で止める**
- **条件分岐は計算量を大きく増やさない範囲で減らす**

このユークリッドの互除法は紀元前 300 年頃の「ユークリッド原論」に書かれており、世界最古のアルゴリズムと考えられている。

## 3 ユークリッドの互除法の時間計算量

ユークリッドの互除法の時間計算量を見積ろう。アルゴリズムの時間計算量は**入力の長さの関数**として見積る。入力 $x$ と $y$ が二進数表現で与えられるとすると、入力の長さは $n := \lceil \log(1+x) \rceil + \lceil \log(1+y) \rceil$ である。アルゴリズムの分野では log のように対数の底が省略されている場合は底は 2 とする。この入力の長さに対して、漸化式の適用回数を見積ろう。知りたいのは**与えられた入力の長さに対する最大の漸化式の適用回数**であるが、以下ではまず **与えられた漸化式の適用回数に対する最小の入力の長さ**を考える。

非負の整数 $x, y$ が $x < y$ を満たすとき、$r_0 := y$, $r_1 := x$ と置き、$r_{i+2} := r_i \,\%\, r_{i+1}$ と定義する。$r_m = 0$ となったところで数列は停止するとする。この $m$ はユークリッドの互除法の漸化式の適用回数 +1 に他ならない。フィボナッチ数列 $F_0, F_1, \ldots$ を

$$\begin{aligned} &F_0 := 0, && F_1 := 1 \\ &F_n := F_{n-1} + F_{n-2}, \qquad && n \ge 2 \end{aligned}$$

と定義する。フィボナッチ数列を使って、逆順にした数列 $r_m, r_{m-1}, \ldots, r_0$ を下から抑えることができる。

**定理 1.** 任意の $1 \le k \le m$ について $r_{m-k} \ge F_{k+1}$.

Proof. $k = 1, 2$ の場合は、$r_{m-1} \ge 1 = F_2$, $r_{m-2} \ge r_{m-1} + 1 \ge 2 = F_3$ より定理は成り立つ。$r_{i+2} = r_i \,\%\, r_{i+1} \le r_i - r_{i+1}$ より $r_i \ge r_{i+1} + r_{i+2}$ が成り立つことから、一般の $k$ についても明らか。 □

よって、ユークリッドの互除法で漸化式の適用が $m-1$ 回以上であれば、$x \ge F_m$ となることが分かった。対偶をとれば、「$x < F_m$ **であればユークリッドの互除法の漸化式の適用回数は** $m-2$ **以下**」であることが分かる。黄金比 $\phi := (1+\sqrt{5})/2$ に対して、$F_m > \frac{1}{\sqrt{5}}\phi^m - 1$ であるので、$1 + x \le \frac{1}{\sqrt{5}}\phi^m \Rightarrow x < F_m$ である。よって漸化式の適用回数は $\lceil \log_\phi[\sqrt{5}(1+x)] \rceil - 2$ 回以下。$\log_\phi \sqrt{5} \approx 1.6723$ なので、$\lceil \log_\phi(1+x) \rceil$ で漸化式の適用回数を上から抑えられる。入力長 $n$ については $x > 0$ と仮定すると、

$$\lceil \log_\phi(1+x) \rceil = \left\lceil \frac{\log(1+x)}{\log \phi} \right\rceil \le \frac{n}{2 \log \phi} + 1 \approx 0.72021\, n + 1$$

と漸化式の適用回数を入力長 $n$ の関数で上から抑えることができる。要は**入力長に比例した回数しか漸化式を適用する必要はない**。引き算で計算する(1)は最悪 $y$ 回漸化式を適用する必要があった。よって漸化式(1)より、漸化式(2)–(4)の方が効率的なアルゴリズムを与える。

次に一回の漸化式の適用に必要な計算時間を考える。一回の漸化式(4)の適用には $x$ が 0 かどうかのチェックと $y\,\%\,x$ の計算が必要である。前者は $O(n)$、後者は「小学生のアルゴリズム」を使えば $O(n^2)$ で計算ができる。よって、全体で $O(n^3)$ で最大公約数の計算ができた。

## 4 Binary GCD

ユークリッドの互除法の変種として Binary GCD というアルゴリズムがある。これは Stein によって 1961 年に発表された。Binary GCD は次の漸化式に基づく。

$$\gcd(x,y) = \begin{cases} y, & \text{if } x = 0 \\ x, & \text{if } y = 0 \\ 2\gcd(x/2, y/2), & \text{if } x \text{ and } y \text{ are even} \\ \gcd(x/2, y), & \text{if } x \text{ is even and } y \text{ is odd} \\ \gcd(x, y/2), & \text{if } x \text{ is odd and } y \text{ is even} \\ \gcd(x, (y-x)/2), & \text{if } x \text{ and } y \text{ are odd and } y \geq x \\ \gcd((x-y)/2, y), & \text{otherwise.} \end{cases} \tag{5}$$

この漸化式が正しいことは簡単に確かめられるだろう。条件分岐が多いので、元のユークリッドの互除法よりも悪いように見えるかもしれないが、この漸化式には**剰余 (%) の計算が含まれていない**。その代わり引き算と 2 で割る計算が含まれているが、これらは $O(n)$ 時間で計算できる。そのため、$x$ と $y$ が大きいときには、Binary GCD の方が高速になる。

## 5 Binary GCD の時間計算量

Binary GCD の計算量の解析は通常のユークリッドの互除法よりも簡単である。まずは漸化式を適用する回数を数えよう。1 回の適用で必ず二進数で一桁は減るので、漸化式の適用回数は高々 $n$ である。一方で一回の漸化式適用につき、引き算と 2 で割る計算が必要であるが、これらは $O(n)$ 時間でできる。また、偶奇のチェックは最下位ビットを見ればいいだけなので $O(1)$ 時間でできる。よって**全体の計算量は** $O(n^2)$ である。

## 6 今日の演習課題

ユークリッドの互除法 (4)を C 言語で書くと次のようになる。

```
unsigned int Euclidean_gcd_rec(unsigned int x, unsigned int y){
  if(x == 0) return y;
  return Euclidean_gcd_rec(y % x, x);
}
```

数式 (4)をそのままコピーしたようなものだと分かるだろう。次にこれを反復を使って書くと次のようになる。

```
unsigned int Euclidean_gcd_itr(unsigned int x, unsigned int y){
  while(x != 0){
    int z = x;
    x = y % x;
    y = z;
  }
  return y;
}
```

今日の演習課題

1. 上記のプログラムを gcd.c に書き写せ。コンパイルして実行せよ。出力される値はランダムな 10000 以下の自然数が互いに素になる確率である。これは $\frac{6}{\pi^2} \approx 0.6079$ に近いはずである。

2. Binary GCD を再帰で実装せよ (通常のユークリッドの互除法よりも遅くなるのでがっかりしないこと)。

3. [発展的課題] Binary GCD を再帰を使わないで反復を使って実装せよ。一般的な C 言語のコンパイラ (gcc, clang など) では関数 `__builtin_ctz(x)` が使用できる。これは `int` 型の `x` について `x` の二進数表現の最下位ビットから連続する 0 の個数を返す関数である。また、`>>` は (算術) 右シフト演算子である。例えば、「`x` を 2 で割れるだけ割る」のは `x >> __builtin_ctz(x)` と書ける。

## 7 おまけ: 連分数展開

実数 $z$ の (正則) 連分数展開とは

$$z = a_0 + \cfrac{1}{a_1 + \cfrac{1}{a_2 + \cfrac{1}{a_3 + \cfrac{1}{a_4 + \cdots}}}}$$

という表現のことである。ここで $a_0, a_1, a_2 \ldots$ は整数とする。実数 $z$ に対してこの連分数展開を得るのは簡単で、次の手続に従って計算すればよい

$$\begin{aligned} a_0 &= \lfloor z \rfloor, & x_0 &= z - a_0, \\ a_{i+1} &= \left\lfloor \frac{1}{x_i} \right\rfloor, & x_{i+1} &= \frac{1}{x_i} - a_i. \end{aligned}$$

ただし、$x_i = 0$ となったら停止する。非負の整数 $x, y$ が $x < y$ を満たしているときに、$z = x/y$ であるとすると,

$$\begin{aligned} a_0 &= 0, & x_0 &= \frac{x}{y} \\ a_1 &= \left\lfloor \frac{y}{x} \right\rfloor, & x_1 &= \frac{y \,\%\, x}{x} \end{aligned}$$

となる。よって、3章で定義した $r_0, r_1, r_2, \ldots,$ を用いると、$x_i = \frac{r_{i+1}}{r_i}$ である。また、$a_i = \lfloor \frac{r_{i-1}}{r_i} \rfloor$ であり、これはユークリッドの互除法における「商」である。つまり、有理数 $x/y$ の連分数展開は $\gcd(x, y)$ のユークリッドの互除法と対応しており、$a_0 = 0$ で $a_1, a_2, \ldots$ は商の列 $\lfloor \frac{r_0}{r_1} \rfloor, \lfloor \frac{r_1}{r_2} \rfloor, \ldots$ となる。この連分数展開は実数を有理数で近似したり、有理数をより分母の小さい有理数で近似するのに役に立つ。