プログラミング第一

10 月 2 日

今週の目標

- Scala のビルドツール sbt と ScalaTest を用いた単体テストができるようになる。

今日の課題 (提出締切は 10 月 15 日 (月))

1. フィボナッチ数を計算する複数のプログラムの作成とそれらを利用したテストをする。

今日のワークフロー

1. GitHub 上の `prg1-2018/assignment1` を fork する。
2. GitHub 上の **自分のアカウント名**`/assignment1` を clone してフィボナッチ数についての課題を終わらせる。
3. GitHub 上の **自分のアカウント名**`/assignment1` に push する。
4. GitHub 上の `prg1-2018/assignment1` に Pull request を送る (次回から課題の提出方法に関するワークフローは書きません)。

1 フィボナッチ数を計算するアルゴリズムとテスト

1.1 概要

プログラムを書くときにまず「効率は悪いけれども確実に正しく動作する実装」をまず書いてから「正しく動作するか少し不安だけど効率がよい実装」を書くことがあるかと思います。今回はフィボナッチ数を例に前者の実装を用いて後者の実装を sbt と ScalaTest を使ってテストしてみましょう。ここでは非負の n について、n 番目のフィボナッチ数 $\mathsf{fib}(n)$ を

$$\begin{aligned}\mathsf{fib}(0) &= 0\\ \mathsf{fib}(1) &= 1\\ \mathsf{fib}(n) &= \mathsf{fib}(n-1) + \mathsf{fib}(n-2), \qquad \text{for } n \geq 2\end{aligned}$$

と定義します。

1.2 効率は悪いが確実に正しく動作するアルゴリズム

フィボナッチ数の定義をそのままプログラムにしてしまえば確実に正しく動きます。

```
def fib_rec(n: Int): BigInt = n match {
  case 0 | 1 => n
  case _ => fib_rec(n-1) + fib_rec(n-2)
}
```

ここで `BigInt` は任意精度整数の型です (n 番目のフィボナッチ数は n に比べて指数関数的に大きいので単に `Int` としてしまうと $n = 48$ で桁溢れします)。

1.3 効率はまあまあよいが正しく動作するかちょっと不安なアルゴリズム

反復を使った $O(n)$ 回の整数の加算演算で n 番目のフィボナッチ数を計算するプログラムです。

Algorithm 1 n 番目のフィボナッチ数を計算するアルゴリズム

```
m ← n
a ← 1
b ← 0
while m ≥ 1 do
    (a, b) ← (a + b, a)
    m ← m − 1
end while
return b
```

1.4 効率はかなりよいが正しく動作するか結構不安なアルゴリズム

一般に

$$\begin{bmatrix} \mathsf{fib}(n+1) \\ \mathsf{fib}(n) \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} \mathsf{fib}(n) \\ \mathsf{fib}(n-1) \end{bmatrix}$$

という関係が $n \geq 1$ について成り立ちます。よって

$$\begin{bmatrix} \mathsf{fib}(n+1) \\ \mathsf{fib}(n) \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}^n \begin{bmatrix} 1 \\ 0 \end{bmatrix}$$

が成り立ちます。行列 $A := \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}$ とおきます。A^n を効率よく計算するためにはどのようにすればよいでしょうか。もしも n が 2 の冪であれば $A^2 = A \times A$, $A^4 = A^2 \times A^2$, $A^8 = A^4 \times A^4$, ... と計算していけば $\log_2 n$ 回の行列積の計算で A^n が計算できます。一般の n については

$$A^n = \begin{cases} A(A^2)^{(n-1)/2}, & \text{if } n \text{ is odd} \\ (A^2)^{n/2}, & \text{if } n \text{ is even} \end{cases}$$

という関係を用いればアルゴリズム 2が得られます。

Algorithm 2 $\mathsf{pow}(A, n)$: A^n を計算するアルゴリズム $(n \geq 0)$

```
if n = 0 then
    return I
else if n is odd then
    return A × pow(A^2, (n − 1)/2)
else
    return pow(A^2, n/2)
end if
```

このアルゴリズムは反復回数が $\lfloor \log_2 n \rfloor + 1$ なので高々 $2(\lfloor \log_2 n \rfloor + 1)$ 回の行列積の計算で A^n が計算できます。このアルゴリズムを用いれば $O(\log n)$ 回の整数演算で n 番目のフィボナッチ数が計算できます。

2 課題

まずフィボナッチ数の計算について定義通りの実装を除く残り 2 つの実装を Scala で書いてください。また、$O(\log n)$ 回の整数演算のプログラムを書くときには汎用的な pow は書く必要はなく、フィボナッチ数が計算できればよいです。そして定義通りの実装を使って $O(n)$ 回の整数演算をする実装をテストしてください。定義通りの実装はとても遅いのであまり大きい n では計算が終わりません。

次に $O(n)$ 回の整数演算をする実装を使って $O(\log n)$ 回の整数演算をする実装をテストしてください。この場合は $n = 100$ でもすぐに計算が終わります。この二段階のテストが上手くいったら自信を持って $O(\log n)$ 回の整数演算をする実装を利用することができます。

3 発展的課題: 多項式乗算を用いた線形漸化式の高速計算アルゴリズム

1.4章と同様に $O(\log n)$ 回の整数演算で n 番目のフィボナッチ数を計算する方法として母関数を使う方法がある。以下では一般の線形漸化式を1.4章の方法よりも高速に計算するアルゴリズムを考える。数列 $a_0, a_1, \ldots$ が $k+1$ 項間線形漸化式を満たすとは、ある $c_1, c_2, \ldots, c_k$ が存在して

$$a_n = \sum_{i=1}^{k} c_i a_{n-i} \tag{1}$$

が任意の $n \geq k$ について成り立つことをいう。数列 $a_0, a_1, \ldots$ の母関数 $G(x)$ を形式的冪級数

$$G(x) = \sum_{n \geq 0} a_n x^n$$

と定義する。形式的冪級数とは多項式を無限の次数に一般化したもので、足し算と掛け算は通常の多項式と同様に定義される。また、

$$\frac{1}{1-x} := (1-x)^{-1} := 1 + x + x^2 + \cdots$$

と定義すると、$(1-x)\frac{1}{1-x} = 1$ となる。つまり $\frac{1}{1-x}$ は $1-x$ の乗法に関する逆元となっている。同様に $p(0) = 1$ であるような形式的冪級数 $p(x)$ についても逆元 $p(x)^{-1}$ が定義できる (上記の x を $1 - P(x)$ に起き替えればよい)。

定理 1. 数列 $(a_n)_{n \geq 0}$ が $k+1$ 項間線形漸化式を満たす $\iff$ ある高々 k 次の多項式 $Q(x)$ で $Q(0) = 1$ を満たすものと高々 $k-1$ 次の多項式 $P(x)$ が存在して、数列 $(a_n)_{n \geq 0}$ の母関数 $G(x)$ が $P(x)/Q(x)$ と書ける。

Proof. ($\Rightarrow$). 数列 $(a_n)_{n \geq 0}$ が $k+1$ 項間線形漸化式 (1)を満たすと仮定する。k 次多項式 $Q(x) := 1 - \sum_{i=1}^{k} c_i x^i$ を定義する。$n \geq k$ について $G(x)Q(x)$ の x^n の係数は

$$a_n - \sum_{i=1}^{k} a_{n-i} c_i = 0$$

となる。つまり、$G(x)Q(x)$ は高々 $k-1$ 次の多項式である。

($\Leftarrow$). ある高々 k 次の多項式 $Q(x)$ で $Q(0) = 1$ を満たすものと高々 $k-1$ 次の多項式 $P(x)$ について $G(x) = P(x)/Q(x)$ とすると、$G(x)Q(x) = P(x)$ より、

$$a_n + \sum_{i=1}^{k} a_{n-i} q_i = p_n$$

が成り立つ。ここで、$(p_n)_{n \geq 0}$ と $(q_n)_{n \geq 0}$ はそれぞれ $P(x)$ と $Q(x)$ の係数である。$P(x)$ が高々 $k-1$ 次であることから、$n \geq k$ について $p_n = 0$ である。よって、数列 $(a_n)_{n \geq 0}$ は $k+1$ 項間線形漸化式を満たす。 □

フィボナッチ数の母関数について分母は $Q(x) = 1 - x - x^2$ となる。$G(x)Q(x)$ を考えることで $P(x) = x$ が得られる。よって、n 番目のフィボナッチ数は

$$[x^n] \frac{x}{1 - x - x^2}$$

である。ここで $[x^n]G(x)$ は形式的冪級数 $G(x)$ の x^n の係数である。一般に定理 1の条件を満たす多項式 $P(x)$ と $Q(x)$ について

$$[x^n]\frac{P(x)}{Q(x)}$$

を計算するアルゴリズムを考えよう。$n=0$ のときは、$P(0)$ が解である。$Q(x)Q(-x)$ は偶多項式 (奇数次数の係数は 0) なので $Q'(x^2)=Q(x)Q(-x)$ となるような高々 k 次の多項式 $Q'(x)$ が存在する。今興味があるのは

$$\frac{P(x)}{Q(x)}=\frac{P(x)Q(-x)}{Q(x)Q(-x)}=\frac{P(x)Q(-x)}{Q'(x^2)}$$

であるが、$Q'(x^2)$ が偶多項式であるので、$\frac{1}{Q'(x^2)}$ は偶形式的冪級数である。よって、n が偶数のとき (奇数のとき) は $P(x)Q(-x)$ の偶数次数 (奇数次数) だけを見ればよい。例えば n が偶数のときに $P'_e(x^2)$ を $P(x)Q(-x)$ の偶数次数の項だけからなる多項式とすると、

$$[x^n]\frac{P(x)}{Q(x)}=[x^n]\frac{P'_e(x^2)}{Q'(x^2)}=[x^{n/2}]\frac{P'_e(x)}{Q'(x)}$$

が得られる (最後の等式では x^2 を x に置き換えている)。同様に n が奇数のときに $xP'_o(x^2)$ を $P(x)Q(-x)$ の奇数次数の項だけからなる多項式とすると、

$$[x^n]\frac{P(x)}{Q(x)}=[x^n]\frac{xP'_o(x^2)}{Q'(x^2)}=[x^{n-1}]\frac{P'_o(x^2)}{Q'(x^2)}=[x^{(n-1)/2}]\frac{P'_o(x)}{Q'(x)}$$

が得られる。よってまとめると次の漸化式が得られる。

$$[x^n]\frac{P(x)}{Q(x)}=\begin{cases}P(0), & \text{if } n=0\\ [x^{n/2}]\frac{P'_e(x)}{Q'(x)}, & \text{if } n \text{ is even}\\ [x^{(n-1)/2}]\frac{P'_o(x)}{Q'(x)}, & \text{otherwise.}\end{cases}$$

ここで $Q'(x)$ は高々 k 次の多項式で $Q'(0)=1$ を満たし、$P'_e(x)$ と $P'_o(x)$ は高々 $k-1$ 次の多項式なので再帰的に漸化式を適用できる。漸化式を適用する回数は $\lfloor \log n \rfloor + 1$ 回で一回の漸化式の適用には高々 k 次の多項式の乗算を 2 回すればよい。そのため、高々 k 次の多項式の乗算に必要な「計算量」を $M(k)$ とおくと、全体の計算量は $O(M(k)\log n)$ となる。1.4章の行列を使う方法では $k\times k$ 行列同士の乗算に必要な計算量 $W(k)$ について、全体の計算量は $O(W(k)\log n)$ となる。素朴なアルゴリズムでは $W(k)=O(k^3)$ である。一方で k 次多項式の乗算は素朴な方法で $O(k^2)$、高速フーリエ変換 (FFT) を使うことで $O(k\log k)$ の計算量で計算ができる。よって、k が大きい場合にはこちらのアルゴリズムの方が1.4章のアルゴリズムより効率的である。フィボナッチ数の場合は $k=2$ なのでどちらのアルゴリズムでも大差はないが、FFT を用いず素朴に多項式乗算した場合に1.4章のアルゴリズムより整数演算の回数が少なく高速になるはずである。

[発展的課題] 素朴な多項式乗算アルゴリズムを用いて、上記のアルゴリズムを使ったプログラムを書いてテストせよ。

余談: ユークリッドの互除法とフィボナッチ数

ユークリッドの互除法は次のプログラムで表わされるアルゴリズムです (簡単のため引数が非負であることは仮定しています)。

```
def gcd(x: Int, y: Int): Int = {
  if(x == 0) y
  else gcd(y % x, x)
}
```

ユークリッドの互除法は「ユークリッド原論」に記されており、最古の非自明なアルゴリズムと考えられています。このアルゴリズムの反復回数 (gcd が呼ばれる回数) を見積ってみましょう。

定理 2. 非負整数のペア (x, y) が $y > x$ 満たすとき、ユークリッドの互除法で gcd(x, y) を計算したときに反復回数が n であるならば $y \geq \mathsf{fib}(n+1)$, $x \geq \mathsf{fib}(n)$ である。

証明. 帰納法により示す。$n = 0$ のとき、明らかに成り立つ。$n \leq k$ について、定理が成立していると仮定する。$n = k + 1$ のとき、$\mathsf{gcd}(\mathsf{y}\%\mathsf{x}, \mathsf{x})$ は k 回の反復で計算ができるので、帰納法の仮定により $x \geq \mathsf{fib}(k+1)$, $y\%x \geq \mathsf{fib}(k)$ が成り立つ。また、$y \geq x + y\%x \geq \mathsf{fib}(k+1) + \mathsf{fib}(k) = \mathsf{fib}(k+2)$ が成り立つ。 □

余談: べき乗に必要な最小の演算回数について

アルゴリズム 2は行列のべき乗に限らず任意のモノイドにおけるべき乗の計算に適用できます (モノイドとは結合法則を満たす 2 項演算 $\times$ と単位元を持つ集合のことです)。このアルゴリズムは $\lfloor \log_2 n \rfloor + \mathsf{popcount}(n) - 1$ 回のモノイドの演算 $\times$ で $A^n := \underbrace{A \times A \times \cdots \times A}_{n\text{個}}$ を計算します。ここで $\mathsf{popcount}(n)$ は n の二進数表現に含まれる 1 の数です。しかしこれは必ずしも最小の演算回数ではありません。例えば A^{15} の計算を考えてみましょう。アルゴリズム 2では 6 回の演算をします (A^2, A^4, A^8 を計算するのに 3 回、$A \times A^2 \times A^4 \times A^8$ を計算するのに 3 回)。しかし一方で $C = A^3$ を 2 回の演算で計算してから C^5 を 3 回の演算で計算すれば 5 回ですみます。関数 $l(n)$ を「A^n を計算するのに必要な最小の演算回数」と定義します。この関数 $l(n)$ は入力サイズ $\lfloor \log_2 n \rfloor + 1$ に関して多項式時間で計算できるのでしょうか？そのようなアルゴリズムは未だ知られていません。また直感に反する事実として $l(2n) = l(n)$ となるような n の存在が知られています。A^{2n} は A^n を二乗すればよいので $l(2n) = l(n) + 1$ となるような気がするのですがそれは一般には正しくないのです。しかもそれどころか $l(2n) = l(n) - 1$ となる n が存在します。例えば $n = 375494703$ のときです。これを実際にプログラムで確認できたらすごいです。$l(n)$ を計算するプログラムを書いてみるのはいい腕試しになります ($l(2n) = l(n)$ となるような n は見つけられます)。

他にも類似の問題として割り算も使える (例えば $A^7 = A^8/A$ と計算できる) としたときの最小演算回数 ($\times$ も / も 1 回と数える) を考えるのも面白いです。実用的には群 (全ての元が逆元を持つモノイド。a の逆元 a^{-1} は $a \times a^{-1} = a^{-1} \times a = e$ を満す元。ここで e は単位元) において逆元の計算が群演算 $\times$ に比べてずっと効率的な場合、そのような群の上でべき乗を効率的に計算するのに役に立ちます。上記のような性質を持つ群の具体例としては、暗号でよく利用されている楕円曲線上の群があります。