

3. 多項式計算のアルゴリズム

- べき乗の計算
- 多項式の計算

3-1: べき乗問題

入力: x, n

(ここで、入力サイズは、 n とします。)

出力: $f(x) = x^n$

素朴なべき乗の求め方

- アイディア
 - x を繰り返し、 n 回乗算する。

$$x^n = \prod_{i=1}^n x = \underbrace{x \bullet x \bullet \cdots \bullet x}_n$$

アルゴリズム `naïve_pow(x,n)`

入力: x, n

出力: x の n 乗

初期化が重要

```
1. f=1.0;  
2. for(i=0; i<N; i++){  
3.     f=f*x;  
4. }  
5. return f;
```

- アルゴリズム `naive_pow` の正当性。
(ほとんど明らかだが、証明することもできる。)

練習

(1)

アルゴリズム `naive_pow` の正当性に関する命題を設定せよ。(不変条件を定めよ。)

(2) (1) で設定した命題を数学的帰納法で証明せよ。

(1) のような命題(条件)を、不変条件(invariant)という。
ループ内の条件等は、特にループ不変条件という。
アサーション(assertion、表明)ともいう。

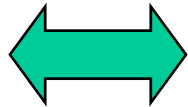
アルゴリズム `naïve_pow` の時間計算量

- 高々 n 回の繰り返し。
- 各繰り返し中では、1回の乗算と、1回の代入が行われているだけである。

$\therefore O(n)$ の時間計算量である。

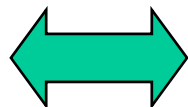
数学の記号とプログラム

$$\prod_{i=0}^{n-1} x_i$$



```
1. for( $i = 0; i < n; i++$ ) {  
2.    $f = f * x[i];$   
3. }
```

$$\sum_{i=0}^{n-1} x_i$$



```
1. for( $i = 0; i < n; i++$ ) {  
2.    $f = f + x[i];$   
3. }
```

高速なべき乗の求め方。

- 注意

- とりあえず、入力に制限を加える
- n が2のべき乗と仮定する。すなわち、ある整数 k が存在して、 $n = 2^k$ と表せる。

- アイディア

- 倍、倍に計算した方が高速に値を求められる。

$$x^i \bullet x^i = x^{2i}$$

- (一方、 $x^i \bullet x = x^{i+1}$)

例

3^8 を求める。

素朴な方法

$$\begin{aligned} 3^0 &= 1 \rightarrow 3^1 = 1 \cdot 3 = 3 \rightarrow 3^2 = 3 \cdot 3 = 9 \rightarrow 3^3 = 9 \cdot 3 = 27 \\ &\rightarrow 3^4 = 27 \cdot 3 = 81 \rightarrow 3^5 = 81 \cdot 3 = 243 \rightarrow 3^6 = 243 \cdot 3 = 729 \\ &\rightarrow 3^7 = 729 \cdot 3 = 2187 \rightarrow 3^8 = 2187 \cdot 3 = 6561 \end{aligned}$$

高速な方法

$$3^1 = 3 \rightarrow 3^2 = 3 \cdot 3 = 9 \rightarrow 3^4 = 9 \cdot 9 = 81 \rightarrow 3^8 = 81 \cdot 81 = 6561$$

アルゴリズム `fast_pow(x,n)`

入力: x, n (ただし、 $n = 2^k$)

出力: x の n 乗

```
1. f=x;  
2. for (i=1; i<k; i++) {  
3.     f=f*f;  
4. }  
5. return f;
```

命題FP1 (fast_powの正当性)

forループが k 回繰り返されたとき、
 f の値は、

$$f = x^{2^k}$$

である。

証明 繰り返し回数 k による帰納法による。
繰り返し回数 k のときの f の値を f_k と表す。

基礎: $k = 0$

アルゴリズム中のステップ1より、 $f_0 = x$ である。

$$\text{一方、右辺} = x^{2^0} = x^1 = x$$

よって、命題は成り立つ。

帰納: $0 < k$

$0 \leq k' < k$ なる全ての k' に対して、
命題が成り立つと仮定する(帰納法の仮定)

$k' = k - 1$ として、 $k - 1$ 回の繰り返しで、

$$f_{k-1} = x^{2^{k-1}}$$

である。よって、 k 回めの繰り返しでは、

$$f_k = f_{k-1} \cdot f_{k-1} = x^{2^{k-1}} \cdot x^{2^{k-1}} = x^{2 \cdot (2^{k-1})} = x^{2^k}$$

QED

fast_powの時間計算量

アルゴリズムは、明らかに、 k 回繰り返す。
繰り返し中は、1回の乗算しか行っていない。
したがって、

$$O(k)$$

の時間計算量である。

$$n = 2^k$$

$$\therefore k = \log_2 n$$

なので、結局

$$O(\log n)$$

の時間計算量である。

一般の自然数に対する高速な べき乗アルゴリズム

n が2のべき乗でないときを考える。

このとき、前の高速なべき乗アルゴリズムを
サブルーチンとして用いることができる。

n の2進数表現を次のように表す。

$$\begin{aligned}(n)_{10} &= (b_m b_{m-1} \cdots b_1 b_0)_2 \\ &= 2^m \times b_m + 2^{m-1} \times b_{m-1} + \cdots + 2^0 \times b_0\end{aligned}$$

このとき、 x^n は次のように表せる。

$$\begin{aligned} x^n &= x^{2^m b_m + 2^{m-1} b_{m-1} + \cdots + 2^0 b_0} \\ &= \left(x^{2^m}\right)^{b_m} \cdot \left(x^{2^{m-1}}\right)^{b_{m-1}} \cdots \left(x^1\right)^{b_0} \\ &= \prod_{i=0}^m \left(x^{2^i}\right)^{b_i} \end{aligned}$$

これより、一般の n に対する高速なアルゴリズムが得られる。

アルゴリズム `general_fast_pow(x,n)`

入力: x, n (n は一般の数)

出力: x の n 乗

```
1. f=1.0;
2. nを2進数  $(b_m b_{m-1} \cdots b_1 b_0)_2$  に変換する。
3. for(i=0; i<m; i++){
4.     if(  $b_i == 1$  ){
5.         f=f*fast_pow(x,  $2^i$  );
6.     }
7. }
8. return f;
```


general_fast_pow(x,n)の時間計算量

general_fast_pow(x, n)の時間計算量を $T_G(n)$ と表し、
fast_pow(x, 2^i) の時間計算量を $T_F(i)$ と表す。

3-7のループの繰り返しは、 $m = \log n$ 回繰り返される。

ループの各繰り返しにおける実行時間の総和により $T_G(n)$ を求める。

$$\begin{aligned}
T_G(n) &\leq \underbrace{T_F(0) + T_F(1) + \cdots + T_F(m)}_{\log n} + c_1 \\
&\leq \underbrace{\log 1 + \log 2 + \cdots + \log n}_{\log n} + c_1 \\
&\leq \underbrace{\log n + \log n + \cdots + \log n}_{\log n} + c_1 \\
&\leq \log^2 n + c_1 \\
&\therefore T_G(n) = O(\log^2 n)
\end{aligned}$$

さらなる高速化

$$\begin{aligned}x^n &= x^{2^m b_m + 2^{m-1} b_{m-1} + \cdots + 2^0 b_0} \\&= \left(x^{2^m}\right)^{b_m} \cdot \left(x^{2^{m-1}}\right)^{b_{m-1}} \cdots \left(x^1\right)^{b_0}\end{aligned}$$

であるが、 x^{2^m} を求めるための高速べき乗アルゴリズム
fast_pow(x, 2^m) の途中段階で全てべき乗

$$x^{2^{m-1}}, x^{2^{m-2}}, \cdots, x^1$$

が出現していることに注意する。

アルゴリズム $\text{super_fast_pow}(x,n)$

入力: x, n (n は一般の整数)

出力: x の n 乗

```
1. f=1.0;
2. tmp=x;
3. nを2進数  $(b_m b_{m-1} \cdots b_1 b_0)_2$  に変換する。
4. for (i=0; i<=m; i++) {
5.     if (  $b_i == 1$  ) {
6.         f=f*tmp;
7.     }
8.     tmp=tmp*tmp;
9. }
10. return f;
```

super_fast_pow(x,n)の時間計算量

super_fast_pow(x, n)の時間計算量を $T_s(n)$ と表す。

3-7のループの繰り返しは、 $m + 1 = \log n + 1$ 回
繰り返される。

ループの各繰り返しは、 $O(1)$ 時間で実現可能である。
よって、

である。
$$T_s(n) = O(\log n)$$

3-2: 多項式の計算

- 入力: $x, n, a_0, a_1, \dots, a_n$
- (ここで、入力サイズは、 n とします。)
- 出力:

$$f(x) = a_0 + a_1x + \dots + a_nx^n$$

$$= \sum_{i=0}^n a_i x^i$$

素朴な多項式の求め方

- アイディア

- 各項を素朴な乗算で計算し、
総和を求める。

$$f(x) = a_0 + a_1x + \cdots + a_nx^n$$

$$= \sum_{i=0}^n a_i x^i$$

$$= \sum_{i=0}^n a_i \left(\prod_{j=0}^{i-1} x \right)$$

アルゴリズム naive_poly()

入力: x, 次数n, 係数a[n]

出力:
$$f(x) = \sum_{i=0}^n a[i]x^i$$

```
1. fx=0;  
2. for(i=0; i<=N; i++){  
3.     tmp=a[i];  
4.     for(j=0; j<i; j++){  
5.         tmp=tmp*x;  
6.     }  
7.     fx=fx+tmp;  
8. }  
9. return fx;
```

3-6は、
第i項の計算

素朴な多項式計算アルゴリズム の正当性

命題NP1 (naive_polyの正当性1)

2.のforループが i 回繰り返されたとき、
tmpの値は、

$$a_i x^i$$

である。

証明

アルゴリズム中のステップ3より、 $tmp = a_i$ に設定される。

また、4の繰り返しは明らかに*i*回である。

したがって、 x は*i*回乗算される。
よって、

$$tmp = a_i x^i$$

である。

より厳密な帰納法でも
証明できる。

QED

命題NP2 (naive_polyの正当性2)

naive_polyは

$$f(x) = \sum_{i=0}^n a_i x^i$$

を計算する。

証明 次数 n に関する帰納法による。

基礎 $n = 0$

$$tmp = a_0 = f(x) \text{ であり正しい。}$$

帰納 $n > 0$

$0 \leq n' < n$ の時正しいと仮定する。

$n' = n - 1$ とする。

$n-1$ の繰り返しのときの f_n の値を $f_{n-1}(x)$ と書く。

このとき、帰納法の仮定より、

$$f_{n-1}(x) = \sum_{i=0}^{n-1} a_i x^i$$

が成り立つ。 n 回目の繰り返しでは、 $i = n$ なので、

$$tmp = a_n x^n \quad (\text{命題PL1より})$$

また、ステップ7より、

$$f(x) = f_{n-1}(x) + tmp$$

$$= \left(\sum_{i=0}^{n-1} a_i x^i \right) + a_n x^n$$

$$= \sum_{i=0}^n a_i x^i$$

QED

素朴な多項式計算アルゴリズム の計算時間

- ステップ5の部分が最も時間を多く繰り返される。

ステップ4のforループは、 i の値にしたがって、
 i 回繰り返される。

よって、時間計算量を $T(n)$ とすると、
 $T(n)$ は、次のように計算できる。

$$\begin{aligned} T(n) &= \sum_{i=0}^n i \\ &= 1 + 2 + 3 + \cdots + n \\ &= \frac{n(n+1)}{2} \\ &= O(n^2) \end{aligned}$$

以上から、naive_polyの最悪時間計算量は、

$$O(n^2)$$

であることがわかる。

また、繰り返し回数は、入力サイズ(n)だけに依存し、問題例(係数配列の状態)に依存しない。

よって、

$$\Theta(n^2)$$

の時間計算量を持つこともわかる。

補足：べき乗に高速なアルゴリズムを用いた場合

$$\begin{aligned} T(n) &= \sum_{i=0}^n \log i \\ &\leq \log 1 + \log 2 + \cdots + \log n \\ &< \log n + \log n + \cdots + \log n \\ &= n \log n \\ \therefore T(n) &= O(n \log n) \end{aligned}$$

と計算できるので、 $O(n \log n)$ のアルゴリズムといえる。

(べき乗の計算に、mathライブラリのpowを用いた場合、この計算量になると考えられる。)

ホーナーの方法

- アイディア
 - x をできるだけくりだしながら計算する。

$$\begin{aligned} f(x) &= a_0 + a_1x + \cdots + a_nx^n \\ &= (a_0 + (a_1 + (\cdots (a_{n-1} + (a_n) * x) * x) \cdots) * x * x) \end{aligned}$$

アルゴリズムを示すまえに、等式の正当性を証明する。

命題H1 (hornerの正当性1)

$$\begin{aligned} f(x) &= a_0 + a_1x + \cdots + a_nx^n \\ &= (a_0 + (a_1 + (\cdots (a_{n-1} + (a_n) * x) * x) \cdots) * x * x) \end{aligned}$$

証明 $a_0, a_1, \cdots, a_n$ に対して、

$$\begin{aligned} f_k(x) &\equiv a_{n-k} + a_{n-k+1}x + a_{n-k+2}x^2 + \cdots + a_nx^k \\ &= \sum_{i=0}^k a_{n-k+i}x^i \end{aligned}$$

と定義する。

このとき、各関数は次のような系列になる。

$$f_0(x) = a_n$$

$$f_1(x) = a_{n-1} + a_n x$$

$$f_2(x) = a_{n-2} + a_{n-1}x + a_n x^2$$

⋮

$$f_n(x) = a_0 + a_1x + a_2x^2 + \cdots + a_n x^n = f(x)$$

この $f_k(x)$ に関して、
次式を k に関する帰納法で命題を証明する。

$$f_k(x) = (a_{n-k} + (a_{n-k+1} + (\cdots (a_{n-1} + (a_n) * x) \cdots) * x) * x)$$

基礎 $k = 0$

$$f_0(x) = a_n = (a_n)$$

であり、満たされる。

帰納 $k > 0$

$0 \leq k' < k$ なる全ての k' で

$$f_{k'}(x) = a_{n-k'} + a_{n-k'+1}x + \cdots + a_n x^{k'}$$

$$= (a_{n-k'} + (a_{n-k'+1} + (\cdots (a_{n-1} + (a_n) * x) \cdots) * x) * x)$$

と仮定する。(帰納法の仮定)

$k' = k - 1$ として

$$f_{k-1}(x) = a_{n-k+1} + a_{n-k+2}x + \cdots + a_n x^{k-1}$$

$$= (a_{n-k+1} + (a_{n-k+2} + (\cdots (a_{n-1} + (a_n) * x) \cdots) * x) * x)$$

このとき、

$$a_{n-k} + a_{n-k+1}x + \cdots + a_n x^k$$

$$= a_{n-k} + (a_{n-k+1} + \cdots + a_n x^{k-1}) * x$$

$$= a_{n-k} + (a_{n-k+1} + (\cdots (a_{n-1} + (a_n) * x) \cdots) * x) * x$$

$$= (a_{n-k} + (a_{n-k+1} + (\cdots (a_{n-1} + (a_n) * x) \cdots) * x) * x)$$

帰納法の仮定を
用いている。

よって、命題は成り立つ。

QED

ホーナーの方法の計算手順

The diagram illustrates the computation steps of Horner's method using nested rectangles. The expression is $(a_0 + (a_1 + (\dots (a_{n-1} + (a_n) * x) * x) \dots) * x) * x$. The innermost term (a_n) is highlighted in a red rectangle. This is followed by $* x$, then $(a_{n-1} + \dots)$ is enclosed in a light red rectangle. This pattern continues with $* x$ and $(a_1 + \dots)$ enclosed in a light blue rectangle, and finally $* x$ and a_0 enclosed in the outermost light green rectangle. The nesting shows the iterative calculation from the innermost term outwards.

$$(a_0 + (a_1 + (\dots (a_{n-1} + (a_n) * x) * x) \dots) * x) * x$$

練習

次の多項式を通常の方法と、ホーナーの方法により計算せよ。

$$f(x) = x^5 - 12x^4 + 3x^3 - 8x^2 + x - 5$$

(1) $f(2)$

(2) $f(-3)$

アルゴリズム horner_poly()

入力: x , 次数 n , 係数 $a[n]$

出力: $f(x) = \sum_{i=0}^n a[i]x^i$

```
1. horner(int k, double x){  
2.     if(k==0){  
3.         return a[n];  
4.     }else{  
5.         f_k=a[n-k]+horner(k-1, x)*x;  
6.         return f_k;  
7.     }  
8. }
```

ホーナーの方法の計算時間

- 計算時間を $T(n)$ とすると次の漸化式を満たす。

$$T(n) = \begin{cases} c_0 & (n = 0) \\ c_1 + T(n-1) + c_2 & (n > 0) \end{cases}$$

ここで、 c_1 を加算に必要な計算時間とし、
 c_2 を乗算に必要な計算時間としている。

$c = \max\{c_1, c_2\}$ とすると、次のようにかける。

$$T(n) \leq \begin{cases} c_0 & (n = 0) \\ T(n-1) + 2c & (n > 0) \end{cases}$$

$$\begin{aligned}
T(n) &\leq T(n-1) + 2c \\
&\leq (T(n-2) + 2c) + 2c = T(n-2) + 4c \\
&\vdots \\
&\leq T(0) + 2nc & \leq (c_0 + 2c)n
\end{aligned}$$

これより、 $T(n) = O(n)$

よって、ホーナーの方法による多項式の計算は、
線形時間アルゴリズムである。

ホーナーの方法の繰り返し版

ホーナーの方法は、次のように、繰り返しを用いても実現できる。

```
1. horner(int k, double x){  
2.     double f_k=a[n];  
3.     for(k=1;k<=n;k++){  
4.         f_k=a[N-k]+( f_k )*x;  
5.     }  
6.     return f_k;  
7. }
```

このアルゴリズムも明らかに線形時間で実行される。

練習

- (1) ループによるホーナー法において、ループ不変条件を設定せよ。
- (2) (1)を数学的帰納法で証明せよ。