３．多項式計算アルゴリズム

- 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

- アィディア
  - 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=O;i＜N;i++){
3.   f=f*x;
4. }
5. return f;
```

- アルゴリズムnaïve_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

正当性を帰納法で証明するためには、どのような命題を設定すればよいか？

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。

- アィディア
  - 倍、倍に計算した方が高速にもとまる。

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

  - （一方、 $x^i \bullet x = x^{i+1}$ ）

例

3^8 を求める。

素朴な方法

$$3^0 = 1 \to 3^1 = 1 \bullet 3 = 3 \to 3^2 = 3 \bullet 3 = 9 \to 3^3 = 9 \bullet 3 = 27$$

$$\to 3^4 = 27 \bullet 3 = 81 \to 3^5 = 81 \bullet 3 = 243 \to 3^6 = 243 \bullet 3 = 729$$

$$\to 3^7 = 729 \bullet 3 = 2187 \to 3^8 = 2187 \bullet 3 = 6561$$

高速な方法

$$3^1 = 3 \to 3^2 = 3 \bullet 3 = 9 \to 3^4 = 9 \bullet 9 = 81 \to 3^8 = 81 \bullet 81 = 6561$$

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。

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、

$$f_k = f_{k-1} \bullet f_{k-1} = x^{2^{k-1}} \bullet x^{2^{k-1}} = x^{2^k}$$

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。

$$(n)_{10} = \left(b_m b_{m-1} \cdots b_1 b_0\right)_2$$

$$= 2^m \times b_m + 2^{m-1} \times b_{m-1} + \cdots + 2^0 \times b_0$$

このとき、 x^n は次のように表せる。

$$x^n = x^{2^m b_m + 2^{m-1} b_{m-1} + \cdots + 2^0 b_0}$$

$$= \left(x^{2^m}\right)^{b_m} \bullet \left(x^{2^{m-1}}\right)^{b_{m-1}} \bullet \cdots \bullet \left(x^1\right)^{b_0}$$

$$= \prod_{i=0}^{m} \left(x^{2^i}\right)^{b_i}$$

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 $\left(b_m b_{m-1} \cdots b_1 b_0\right)_2$ 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。

$$T_G(n) \leq \underbrace{T_F(0) + T_F(1) + \cdots + T_F(m)}_{\log n} + c_1$$

$$\leq \underbrace{\log 1 + \log 2 + \cdots + \log n}_{\log n} + c_1$$

$$\leq \underbrace{\log n + \log n + \cdots + \log n}_{\log n} + c_1$$

$$\leq \log^2 n + c_1$$

$$\therefore T_G(n) = O(\log^2 n)$$

さらなる高速化

$$x^n = x^{2^m b_m + 2^{m-1} b_{m-1} + \cdots + 2^0 b_0}$$

$$= \left(x^{2^m}\right)^{b_m} \bullet \left(x^{2^{m-1}}\right)^{b_{m-1}} \bullet \cdots \bullet \left(x^1\right)^{b_0}$$

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

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

が出現していることに注意する。

アルゴリズム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} ... 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、

$$T_S(n)=O(\log n)$$

である。

３－２:多項式の計算

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

$$f(x) = a_0 + a_1 x + \cdots + a_n x^n$$

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

素朴な多項式の求め方

- アィディア
  - 各項を素朴な乗算で計算し、総和を求める。

$$f(x) = a_0 + a_1 x + \cdots + a_n x^n$$

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

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

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）

naïve_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　$n = 0$

$tmp = a_0 = f(x)$ 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fx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、**

$$tmp = a_n x^n$$ **（命題PL1より）**

また、ステップ7より、

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

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

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

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。

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

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

- アィディア
  - xをできるだけくくりだしながら計算する。

$$f(x) = a_0 + a_1 x + \cdots + a_n x^n$$

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

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）

$$f(x) = a_0 + a_1 x + \cdots + a_n x^n$$
$$= (a_0 + (a_1 + (\cdots(a_{n-1} + (a_n) * x) * x)\cdots) * x * x)$$

証明　$a_0, a_1, \cdots, a_n$ に対して、

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

$$= \sum_{i=0}^{k} a_{n-k+i} x^i$$

と定義する。

このとき、各関数は次のような系列になる。

$$f_0(x) = a_n$$
$$f_1(x) = a_{n-1} + a_n x$$
$$f_2(x) = a_{n-2} + a_{n-1}x + a_n x^2$$
$$\vdots$$
$$f_n(x) = a_0 + a_1 x + a_2 x^2 + \cdots + a_n x^n = f(x)$$

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。

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

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**

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

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

と仮定する。(帰納法の仮定)

$k' = k - 1$　**として**

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

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

このとき、

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

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

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

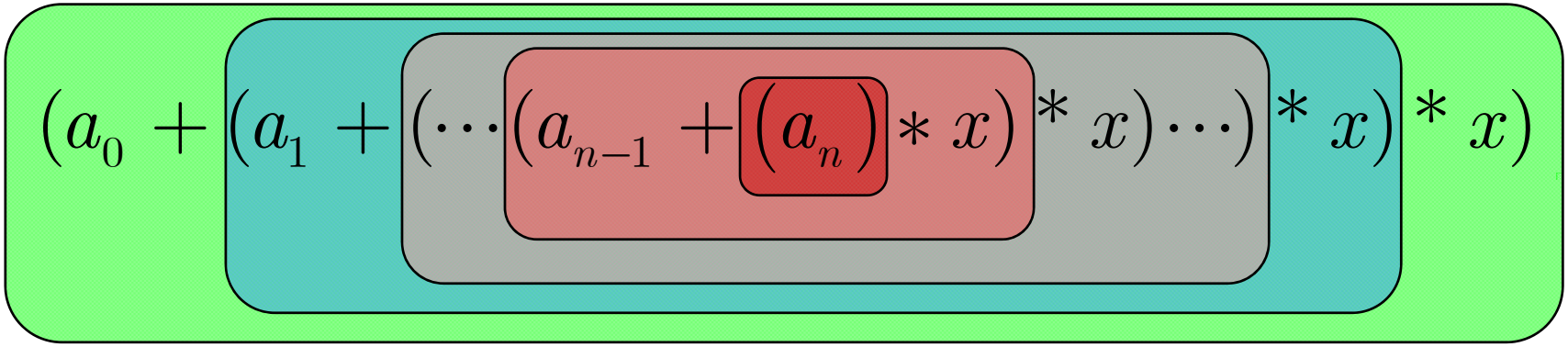
$$= (a_{n-k} + (a_{n-k+1} + (\cdots(a_{n-1} + (a_n) * x)\cdots) * x) * x)$$

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



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。

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

$$f(2)$$

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。

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

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

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。