I111 アルゴリズムとデータ構造
第1回: プログラミングの基礎
の続き

担当: 上原 隆平(uehara)

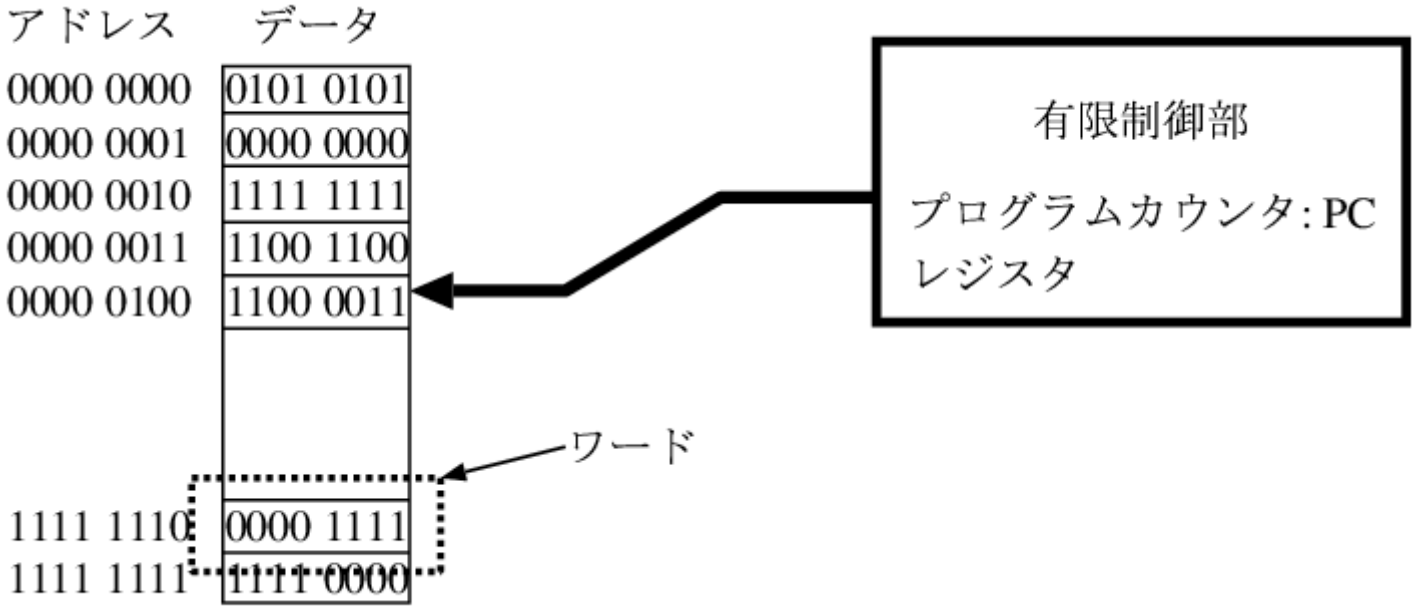
2015/04/10

計算モデルの話

そもそもコンピュータってどういう仕組みで動いているの？

- 計算モデルによって，アルゴリズムの記述や効率は違ってくる
  - なにが「基本演算」なのか？
  - どんなデータが記憶できるのか？
    - 自然数，実数（？），画像，音楽データ…?
- いくつかの標準的なモデルがある
  - チューリング機械：かのアラン・チューリングが考案．すべての議論の基礎となっている．
  - RAMモデル：アルゴリズムの話をするときはこれが標準．

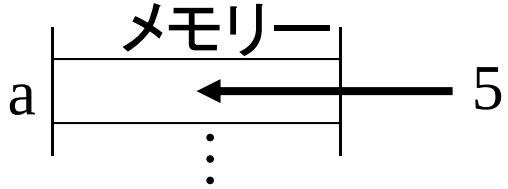
RAMモデル



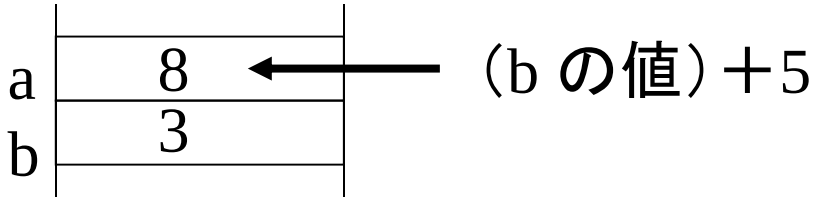
- 記憶装置とCPUからなる. (入出力は無視)
- 実際のCPUやメモリと本質的には同じ
- ランダムにデータをアクセスできる（Random Access Machine)
- C言語などでは，こうしたRAMモデルがなんとなく透けて見える体系になっている（ポインタ，配列など）

C言語の基礎: 代入文

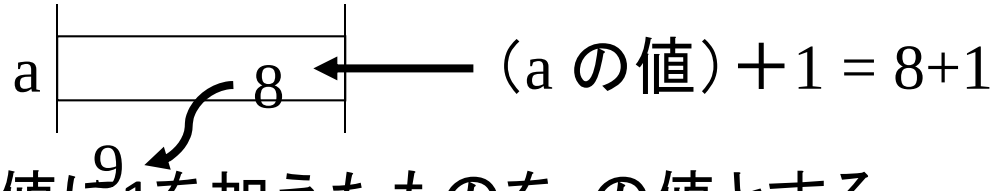
- a=5



  - aという名前の場所に数値5を格納
- a=b+5



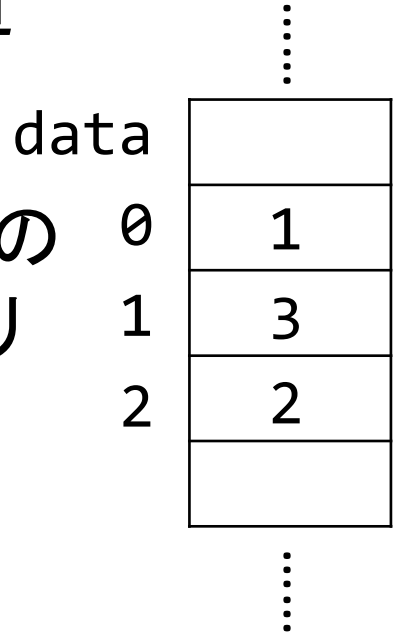
  - bという名前の場所に格納されている値(変数bの値)に5を加えた数値をaという名前の場所に格納
- a=a+1



  - 変数aの値に1を加えたものをaの値とする

C言語の基礎: 配列(1/2)

- 配列とは?
  同じ型(int, float, etc.)の値をメモリ上に連続して並べるデータ構造
- 例: int data[3]
  - dataという名前でint型の値の格納場所を3つメモリ上に連続して確保




```
int data[3];
data[0]=1;
data[2]=2;
data[1]=3;
```

C言語の基礎: 配列(2/2)
最大値を取得する

- 例: int data[100]に格納された値の最大値を計算する


```
int data[100];
int i,max;
/*data is initialized somehow*/
max=0;
for(i=0;i<100;i=i+1){
  if(max<data[i]) max=data[i];
}
printf(“maximum data = %d”,max);
```

Q: このプログラムは正しい？

C言語の基礎: 配列(2/2)
最大値を取得する

- 例: int data[100]に格納された値の最大値を計算する

正しくない！

```
int data[100];
int i,max;
/*data is initialized somehow*/
max=0;
for(i=0;i<100;i=i+1){
  if(max<data[i]) max=data[i];
}
printf("maximum data = %d",max);
```

dataの値が全て負のとき0が最大値になる！

Q: このプログラムは正しい？

C言語の基礎: 配列(2/2)
最大値を取得する

- 例: int data[100]に格納された値の最大値を計算する – 正しいプログラム

```
int data[100];
int i,max;
/*data is initialized somehow*/
max=data[0];
for(i=1;i<100;i=i+1){
  if(max<data[i]) max=data[i];
}
printf("maximum data = %d",max);
```

maxの値は常にdataの値のどれか

ミニ演習

- ミニ演習2: 次の関数は何をする？
  - collatz(5) と collatz(7) の出力を書け

```
collatz(正整数 n) {
  print(n);  // n を出力
  if (n == 1) return;
  if (n%2==0) collatz(n/2);
  else        collatz(3n+1);
}
```

(n%2==0)は「nが偶数」ということ

I111 アルゴリズムとデータ構造
第2回: アルゴリズムの基礎

担当: 上原 隆平 (uehara)

2015/04/10

内容

- 株の最大売却益を求めるアルゴリズム
  - 素朴 (naïve) な方法
  - ちょっとした改良
  - さらなる改良: $O(n^2)$ から $O(n)$へ
- $293^{10,000,000}$の下3桁を求めるアルゴリズム
  - 素朴な方法とその問題点
  - 10乗計算を利用した効率化
- 2進展開による冪乗計算
- 再帰関数

アルゴリズム(algorithm)とは

- 計算機を用いて解ける問題に対する解法を抽象的に記述
  - どんな入力に対しても正しい解を返す
  - 必ず終了する
  - 記述が曖昧でない

- プログラム: 計算機言語で記述したもの
  - 入力によっては暴走，停止しない


Al-Khwarizmi

良いアルゴリズムを設計する

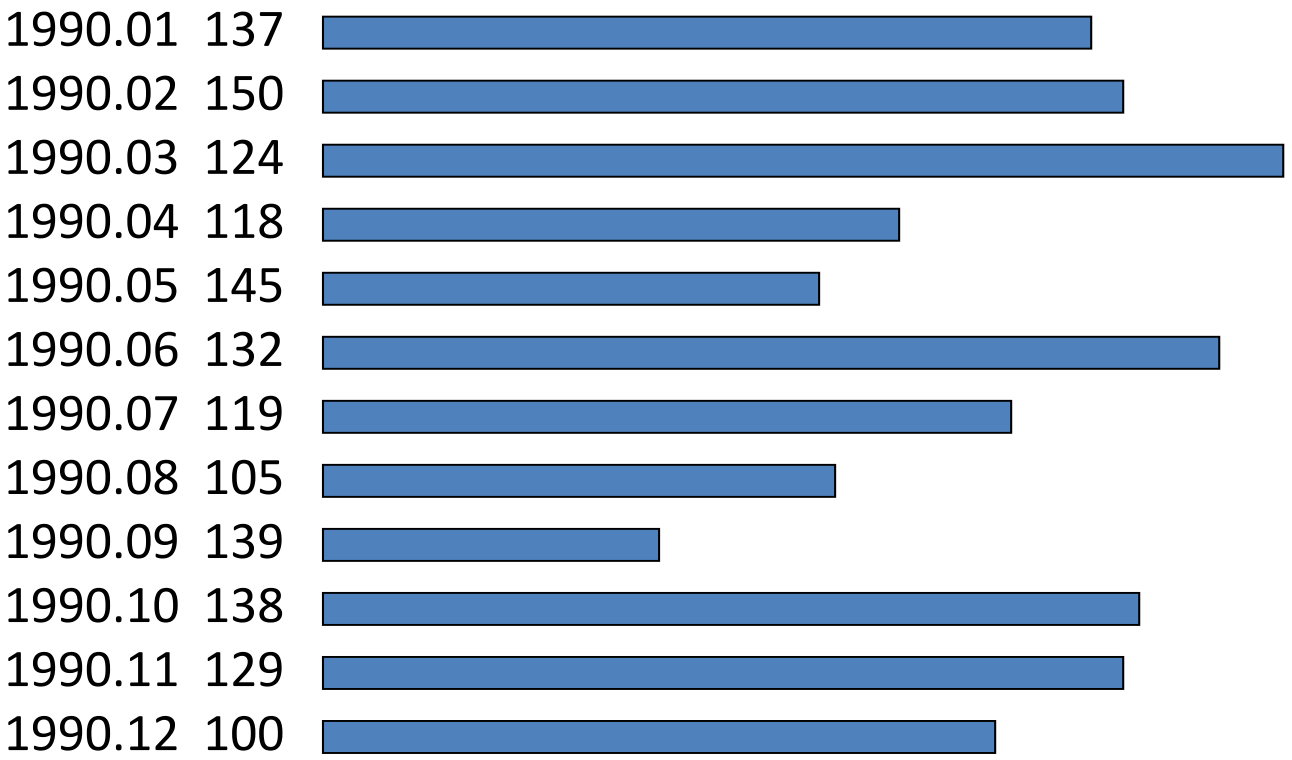
- 設計技法を身につける
- 計算時間, メモリ量の推定
- アルゴリズムの正しさの検証・証明

- 悪いアルゴリズム
  - 思いつき: アルゴリズム設計技法の知識の欠如
  - 作りっぱなし: アルゴリズムの動作の解析なし

最大売却益の求め方

例: 株で儲ける(最大売却益)

- 株を買って売ったときの収益の最大値は？



＊一回買って
一回売る

問題の定式化

- int sp[n]: 株価を蓄える配列 (nは適当な数)
- 時点 i に買って時点 j に売るとすると
  - 買値: sp[i]
  - 売値: sp[j]
  - 利益: sp[j] - sp[i]
- sp[j]-sp[i]の最大値
  max{sp[j] - sp[i] | 0<= i < j < n}
  を求める

アルゴリズムの外形

- 方式A

```
for i=0 to n-2
  for j=i+1 to n-1
    利益sp[j]-sp[i]を求める
```

- 方式B:

```
for j=1 to n-1
  for i=0 to j-1
    利益sp[j]-sp[i]を求める
```

方式Aのアルゴリズム

- 以下のアルゴリズムは効率的か？

```
最大売却益(sp[],n){/*sp[0]…sp[n-1]*/
  mxp=0; /*利益の最大値*/
  for i=0 to n-2
    for j=i+1 to n-1
      d = sp[j] – sp[i]; /*売却益*/
      if d > mxp then mxp = d;
                  /*最大売却益の更新*/
    endfor
  endfor
  return mxp;
}
```

方式Aのアルゴリズム

- 以下のアルゴリズムは効率的か？

```
最大売却益(sp[],n){/*sp[0]…sp[n-1]*/
  mxp=0; /*利益の最大値*/
  for i=0 to n-2
    for j=i+1 to n-1
      d = sp[j] - sp[i]; /*売却益*/
      if d > mxp then mxp = d;
                    /*最大売却益の更新*/
    endfor
  endfor
  return mxp;
}
```

iを固定すると、売却益が最大になるのは
sp[j]が最大になるとき
➔ 毎度sp[j]-sp[i]を計算する必要はない

方式Aのアルゴリズム(改良版)

```
最大売却益(sp[],n){ /*sp[0]…sp[n-1]*/
  mxp=0; /*利益の最大値*/
  for i=0 to n-2
    mxsp = sp[i];
    for j=i+1 to n-1
      if sp[j] > mxsp then mxsp = sp[j];
    endfor
    d = mxsp - sp[i];
    if d > mxp then mxp = d;
  endfor
  return mxp;
}
```

株価の最大値をmxspに

引き算がループの外側に

最大売却益の比較も外側に

アルゴリズムの外形

- 方式A

```
for i=0 to n-2
  for j=i+1 to n-1
    利益sp[j]-sp[i]を求める
```

- 方式B:

```
for j=1 to n-1
  for i=0 to j-1
    利益sp[j]-sp[i]を求める
```

方式Bのアルゴリズム

```
最大売却益(sp[],n){ /*sp[0]…sp[n-1]*/
  mxp=0; /*利益の最大値*/
  for j=1 to n-1
    mnsp = sp[j];
    for i=0 to j-1
      if sp[i] < mnsp then mnsp = sp[i];
    endfor
    d = sp[j] - mnsp;
    if d > mxp then mxp = d;
  endfor
  return mxp;
}
```

株価の最安値をmnspに

アルゴリズムの効率

n^2に比例する程度の量であることを表す記法

- 繰り返し回数
  - 方式(A): ループの回数は O(n^2)

第4回

$$\sum_{i=0}^{n-2}\sum_{j=i+1}^{n-1} 1 = \sum_{i=0}^{n-2}(n-1-i) = \frac{n^2-n}{2} \le n^2/2$$

  - 方式(B): ループの回数は O(n^2)

$$\sum_{j=1}^{n-1}\sum_{i=0}^{j-1} 1 = \sum_{j=1}^{n-1} j = \frac{n^2-n}{2} \le n^2/2$$

Q. ループの回数は減らせない？

アルゴリズムの改善
ループの回数を減らす

- 2つ目のforループの中身を考える
  - A方式:
    - MAX[i,n-1]を時点iから時点n-1までの最高値とする
    - MAX[1,n-1], MAX[2,n-1],…の順に計算
    - Q: MAX[i-1,n-1]を使ってMAX[i,n-1]が計算できる？

    NO
  - B方式:
    - MIN[0,j-1]を時点0から時点j-1までの最安値とする
    - MIN[0,0], MIN[0,1], …の順に計算
    - Q: MIN[0,j-1]を使ってMIN[0,j]を計算できる?

    YES! `MIN[0,j] = min(MIN[0,j-1],sp[j])`

方式Bのアルゴリズム

```
最大売却益(sp[],n){ /*sp[0] sp[n-1]*/
  mxp=0; /*利益の最大値*/
  for j=1 to n-1
    mnsp = sp[j];
    for i=0 to j-1
      if sp[i] < mnsp then mnsp = sp[i];
    endfor
    d = sp[j] - mnsp;
    if d > mxp then mxp = d;
  endfor
  return mxp;
}
```

- j=kのとき: mnspはsp[0]からsp[k-1]までの中の最安値
- j=k+1のとき: mnspはsp[0]からsp[k]までの中の最安値

j=kのときの最安値msfをとっておいてj=k+1のときmsfとsp[k]の最小値をとればそれがj=k+1の時の最安値

効率の良いアルゴリズム

- O(n) 時間で計算するアルゴリズム

```
最大売却益(sp[],n){ /*sp[0]…sp[n-1]*/
  mxp=0; /*利益の最高値*/
  msf = sp[0]; /*これまでの最安値*/
  for j=1 to n-1
    d = sp[j] - msf;
    if d > mxp then mxp = d;
    if sp[j] < msf then msf = sp[j];
  endfor
  return mxp;
}
```

$293^{10,000,000}$

冪乗の計算

冪乗の計算

- 問い: $293^{10,000,000}$の下位3桁の数字を求めよ
- 方法1: 素朴な解法
  293を10,000,000回掛けて下から3桁出力

```
int a,b;
a=1;
for(i=1;i<=10000000;i=i+1)
  a=a*293;
b=a%1000; /*下三桁*/
printf(“answer=%d”,b);
```

出題者

徳山豪教授
（東北大学）

- この方法の問題点は？

冪乗の計算
素朴な解法の問題点

- パソコンでは扱えないかもしれない
  - 32bitで整数を表わしたとすると最大値は
    $2^{31}-1 = 2{,}147{,}483{,}647$
  - 293を10,000,000回掛けると桁数は
    $10{,}000{,}000 \cdot \log_{10}293 \geqq 24{,}668{,}676$
- 非常に遅い
  - 単純に掛け算が10,000,000回
    （実はこれくらいならすぐ計算できてしまう…）

冪乗の計算
桁数爆発を抑える

- 必要なのは下三桁であることに注目
  - 293*293=85,849
  - 85,849*293
    = (85**1000**+849) * 293
    = **85*1000*293** + 849*293

下三桁には影響しない

- 毎回の計算で下三桁だけ求めればよい
  - a=a*293 ➔ a=(a*293)%1000

冪乗の計算
改良されたプログラム

```
int a;
a=1;
for(i=1;i<=10000000;i=i+1)
  a = (a*293) % 1000;
printf(“answer=%d”,a);
```

- 依然として10,000,000回の乗算と剰余が必要
  画期的な高速化は?

冪乗の高速化: 10乗の計算 (1/3)

- 方法1: 単純に10回掛ける
- 方法2: 2進数展開の利用
  - $10_{(10)} = 1010_{(2)} = 8 + 2$
  - 2乗を繰り返して x^2, x^4, x^8 を計算して
    $x^{10} = x^8 \times x^2$ とすると4回の乗算で済む
    - 例: $3^2 = 9$, $3^4 = 9^2 = 81$, $3^8 = 81^2 = 6{,}561$,
      $3^{10} = 3^8 \times 3^2 = 6{,}561 \times 9 = 59{,}049$

冪乗の高速化: 10乗の計算 (2/3)

- power10(x)を以下のように定義する:

```
int power10(int x){
  int x2,x4,x8;
  x2=(x*x)%1000; x4=(x2*x2)%1000;
  x8=(x4*x4)%1000;
  return (x2*x8)%1000;
}
```

- $293^{10,000,000}$は以下のように計算できる:

```
int i,x; x=293;
for(i=0;i<7;i=i+1) x=power10(x);
printf(“answer=%d”,x);
```

4 × 7= 28回の乗算

冪乗の高速化: 10乗の計算 (3/3)

- 計算の様子
  - 293^{10} mod 1000 = 249
  - 293^{100} mod 1000 = 1
  - $293^{1,000}$ mod 1000 = 1
  - $293^{10,000}$ mod 1000 = 1
  - $293^{100,000}$ mod 1000 = 1
  - $293^{1,000,000}$ mod 1000 = 1
  - $293^{10,000,000}$ mod 1000 = 1

オイラーの定理

- n と互いに素な x について

$$x^{\phi(n)} \bmod n = 1$$

- $\Phi(n)$ はオイラーのファイ関数と呼ばれ、n 以下で n と互いに素な数の個数を表す
  - $n = \prod_{i=1}^{d} p_i^{k_i}$ と素因数分解できるとき

$$\phi(n) = n \prod_{i=1}^{d} \left(1 - \frac{1}{p_k}\right)$$

- チェック: 二つの自然数が互いに素とは？

オイラーの定理を使った $293^{10,000,000} \bmod 1000$ 計算の高速化

- $\Phi(1000) = \Phi(2^3 \times 5^3) = 1000 \times 1/2 \times 4/5 = 400$
  - $\Phi(n)$ は n の素因数分解が出来れば計算できる
- 293 と 1000 は互いに素である
  $\Rightarrow 293^{\Phi(1000)} \bmod 1000 = 293^{400} \bmod 1000 = 1$

- $293^{10,000,000} \bmod 1000$
  $= (293^{400} \bmod 1000)^{25,000} \bmod 1000$
  $= 1^{25,000} \bmod 1000$
  $= 1$

冪乗計算: まとめ

- 方法1: 素朴な方法
  10,000,000回の乗算
- 方法2: 10乗を計算する手続きを利用
  全体で28回の乗算と剰余
- 方法3: オイラーの定理を利用
  手計算で十分

10乗より100乗を使う方が効率的か？

- 答え: No
- 理由: $x^{100} = x^{64} \times x^{32} \times x^{4}$ となるが、
  - x^{64} の計算に6回の乗算が必要 ($64 = 2^6$)
  - $x^{64} \times x^{32} \times x^{4}$ の計算に2回の乗算
  - $10^7 = 100^3 \times 10$ より全体の乗算回数は
    $3 \times 8 + 4 = 28$
    で10乗計算と同じ

2進数展開と比べたらどうなるか？

2進数展開のプログラム

```
int n,k,bit[100];
n=/*some integer*/;
k=0;
do {
  bit[k]=n%2; k=k+1;
  n=n/2;
} while (n>0);
while (k>0) {
  k=k-1;
  print(“%d”,bit[k]);
}
```

n=49の場合

```
  bit[0]=1  n=24
  bit[1]=0  n=12
  bit[2]=0  n=6
  bit[3]=0  n=3
  bit[4]=1  n=1
  bit[5]=1  n=0, k=6
```

出力: 110001

2進数展開を用いた冪乗の計算

- 整数 n の2進数表現を
  $n = b_k 2^k + b_{k-1} 2^{k-1} + \cdots + b_1 2^1 + b_0 2^0$
  とするとき

$$x^n = x^{b_k 2^k} \times x^{b_{k-1} 2^{k-1}} \times \cdots \times x^{b_1 2^1} \times x^{b_0 2^0}$$

  よって，$b_i = 1$ のところの積をとればよい

2進数展開を用いた冪乗の計算: プログラム

```
p=x; t=1;

if(n%2==1) t=t*p;
n=n/2;
do {
  if(n%2==1)
    t=t*p;
  p=p*p;
  n=n/2;
} while (n > 0)
printf("%d",t);
```

ビットが1の所を乗算

- 例: $n = 49_{(10)} = 110001_{(2)}$

n	p	t	乗算
49/2=24	x	x	1
24/2=12	x^2	↑	2
12/2=6	x^4	↑	3
6/2=3	x^8	↑	4
3/2=1	x^{16}	x^{17}	6
1/2=0	x^{32}	x^{49}	8

2進数展開を用いた冪乗の計算: プログラム

```
p=x; t=1;

if(n%2==1) t=t*p;
n=n/2;
do {
  if(n%2==1)
    t=t*p;
  p=p*p;
  n=n/2;
} while (n > 0)
printf(“%d”,t);
```

ビットが1の所を乗算

- Q: 繰り返し回数は？
- A: $\lfloor \log_2 n \rfloor$ 回
  - $2^{k-1} < n \leqq 2^k$ のとき k 回目の繰り返しで n=0
  - $\lfloor\ \rfloor$ は切り下げ

2進数展開を用いた $293^{10,000,000}$ の計算

- $10{,}000{,}000_{(10)} = 100110001001011010000000_{(2)}$
  - $x_2, x_4, x_8, \ldots$ の計算は23回の乗算
  - 8箇所のビット1それぞれに乗算1回
  - 合計で31回の乗算
- power10を用いたとき（28回）より効率が悪い！

いつでも10乗に基づく方法の方が効率がよいか？

- 答え: No!
- 反例: 293^{394}
  - 2進数展開: 12回の乗算
    - $394_{(10)} = 110001010_{(2)}$
    - $\lfloor \log_2 394 \rfloor = 8$
    - 1が4つ
  - power10: 16回の乗算
    - 4乗: 2回
    - 90乗: 4+4=8回
    - 300乗: 4+2=6回

おまけ

- では x^k を計算するときの乗算の最小回数はいくつなのか？
  - $x^2, x^4, x^8, x^{16}, \ldots$ を計算する方法で、$O(\log k)$という上界（それだけやれば少なくとも十分）は得られる
  - この方法で得られる回数が最小でない例：k=15
  - 興味のある人は以下を調べるとよい：

    The Art of Computer Programming, D. Knuth, Vol. 2, Chapter. 4.6.3. （邦訳もあり）

この本はコンピュータサイエンス業界のバイブル。
Knuthは業界の巨人。この本を書くためにKnuthはTEXを作った。