コンピュータサイエンス入門第二

永藤 直行

東京工業大学

4th quarter

Part I

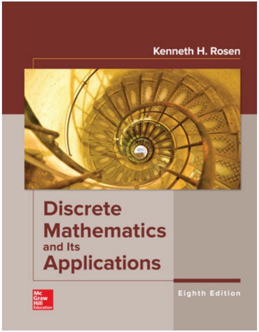
Prologue

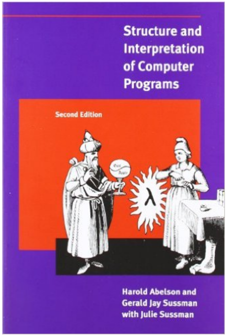
Prologue

Outline

参考図書

- Discrete Mathematics and its Applications
- Structure and Interpretation of Computer Programs
  https://web.mit.edu/6.001/6.037/sicp.pdf
- 計算機プログラムの構造と解釈 (日本語訳)
  http://sicp.iijlab.net/fulltext/xcont.html





Outline

講義概要

- 講義資料は https://sites.google.com/presystems.xyz/elementaryCS/ に置いてあります
- 講義スケジュール:
  1. 再帰
  2. よいアルゴリズム，わるいアルゴリズム，ふつうのアルゴリズム
  3. 大きい数と小さい数の計算 (整数や実数の計算)

目標

- 計算で表すとき便利な道具としての再帰と
- アルゴリズムを選択する際に注意すべきことを会得すること

Outline

1. 教科書，参考文献
2. 講義概要
3. 評価基準

評価基準

- 講義は全 7 回
- 宿題: 3 回 (計 10 点)
- 課題: 3 回 (計 75 点) と特別課題 (15 点)
- 課題提出:
  - 講義時間中に課題を出します
  - 提出方法はその都度指定します

Part II

再帰

再帰

Outline

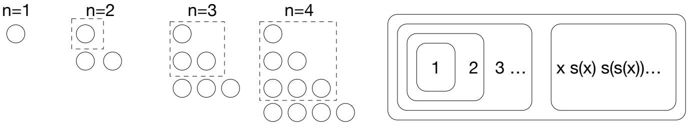
再帰への導入

- 対象（問題）を簡潔に明示的に表す方法
- たとえば下の絵，ひとつ絵を書いてその中央にその絵を書いている（Droste 効果)
- 対象を定義するときにより小さい部分を参照して定義する

再帰の例

- 数列 $a_n = \frac{n(n+1)}{2}$，$n \geq 1$
- 初項を 1，n 項を $a_n = n + a_{n-1}$ となる
- n 項を決めるのに $n - 1$ 項から決める
- まだ決まっていない最小の項を既に知っている項から決定する
- 再帰的定義:

$$a(n) = \begin{cases} 1 & \text{if } n = 1 \\ n + a(n-1) & \text{otherwise} \end{cases}$$



再帰的定義の原理

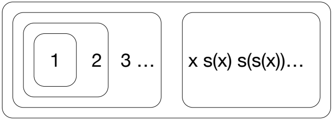
- 関数 f の定義に x より小さい要素についての評価値を利用して定義することを原始再帰 (primitive recursion) と呼ぶ
- f が x より小さい値について計算可能であり，いつも同じ値と仮定し，$f \upharpoonright x$ と書く
- 定義できた最大の x のつぎの値について定義するというのを繰り返すと全体について定義できる (δ-近似)

再帰の原理

順序数 (自然数は順序数) x として，

$$f(x) = G(f \upharpoonright x)$$

ここで，$f \upharpoonright x$ は f を x より小さい数に制限したもの，G は計算のしかたを表したもの



- G に相当するのが succ
- Basic step に近づけるように recursive step を定義
  - $a + b =$

Basic step:	$a + 0$	if $b = 0$
Recusive step:	succ(a + pred(b))	otherwise

Listing 1: 加算

```
# Recursive definition
import os

def succ (x):
  return(x+1)
def pred (x):
  return(x-1)
def add (a,b):
  if (b==0):
    return(a)
  else:
    return(succ(add(a,pred(b))))
```

Listing 2: 乗算

```
# Multiplication
#
def mult (a,b):
  if (b==0):
    return(0)
  else:
    return(add(add(0,a),mult(a,pred(b))))
```

再帰的定義の例 1

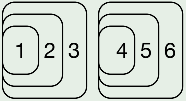
Factorial

- $n! = n \cdot (n-1) \cdot (n-2) \cdots 3 \cdot 2 \cdot 1$
- $n!$ を決めるのに $(n-1)!$ の値をつかう
- G に相当するのが $n \cdot (n-1)!$ ということ
- まだ決まっていない最小の値は決まっている値で最大の値のつぎになる

Listing 3: fact.py

```
def fact (n):
  if (n == 1):
    return(1)
  else:
    return(n*(fact(n-1)))
```

Example (4!)



フィボナッチ数

- 突然ですが次の問題を考えてみてください

うさぎとフィボナッチ数

- n ヶ月後のうさぎのつがいは何組？
- 最初一組のつがいだけ
- 2 ヶ月経つと メス 1 匹を生んでそのつがいが 1 組増える
- うさぎは決して死なない

月	生後 0 ヶ月	生後 2 ヶ月以下	合計
0	0	1	1
1	0	1	1
2	1	1	2
3	1	2	3
4	2	3	5

再帰的定義の例 2

- フィボナッチ数

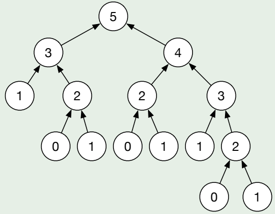
0	1	2	3	4	5···
1	1	2	3	5	8···

- fib($S(n)$) を決めるのに fib(n) と fib($n-1$) を使う
- G に相当するのが n と $n-1$ のフィボナッチ数を足し合わせるということ

Listing 4: フィボナッチ数

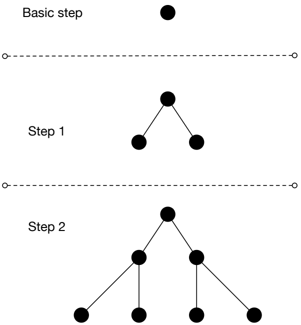
```
def fib (n):
  if (n==0):
    return(1)
  else:
    if (n==1):
      return(1)
    else:
      return(fib(n-1)+fib(n-2))
```

Example (fib(5))



Tree (木構造)

- 数以外も
- Basic step: vertex r は tree
- Recursive step: $T_1, T_2, \cdots, T_n$ それぞれ root を $r_1, r_2, \cdots, r_n$ とする tree として，r から $r_1, r_2, \cdots, r_n$ への edge 追加したものもまた tree である



集合の再帰的定義

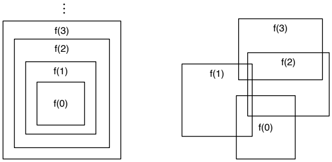
- Basic step: 集合の初期要素を定義
- Recursive step: 既にわかっている要素から新しい要素を定義する規則

Example (文字列の集合 Σ^*)

- Basic step: $\epsilon \in \Sigma^*$ (空列 ϵ も Σ^* に含まれる)
- Recursive step: $a \in \Sigma, w \in \Sigma^*$ ならば $aw \in \Sigma^*$
- E.g.: $\{\epsilon,\ a,\ aa,\ aaa, \cdots\}$

再帰は強力な道具

- 最初の元から順番に対象を構成
- 対象が順番に並べられるなら
- 問題を解く時も，最も小さい問題の解から順番に全体の解を構成することができる
- （いつもではないけど）



宿題 1

- まずは https://sites.google.com/presystems.xyz/elementarycs/top から four-ops-rec-skeleton.py をダウンロード
- 自然数上の引き算，商を求める関数，余りを求める関数を再帰で定義せよ

Outline

仮引数と実引数

- CS 第 1 では手続きに名前をつけて抽象化することをみた
- 関数は 0 個以上の仮引数というものをもつ
  - 下の例では `n`, `eps` などが仮引数
- 仮引数は関数の本体で有効である
- 関数を呼び出したときの値に束縛 (bind) されて，関数の本体では呼び出し時の値に置き換えられる
- 呼び出し時の値を実引数という
- 一般に変数は有効範囲 (scope) が決まっている
- 仮引数は関数本体が有効範囲である

Listing 5: newton.py

```
### Newton's method
def sqrt_iter (guess,n,eps,previous):
  def is_enough (guess,eps,previous):
    return(abs(previous-guess)<(2*eps))
  def improve (guess,n):
    return((guess+(n/guess))/2.0)
  if is_enough(guess,eps,previous):
     return(guess)
  else:
     return(sqrt_iter(improve(guess,n),n,eps,guess))
def sqroot1 (n,eps):
  return(sqrt_iter(1.0,n,(2*eps),0.0))
```

Listing 6: newton.py

```
def sqroot (n):
  ### Machine epsilon
  def eps_m ():
    epsilon, old, prod = 1.0, 0.0, 0.0
    cnt=0
    while (prod!=1.0):
      old = epsilon
      cnt=cnt+1
      epsilon=epsilon/2.0
      prod=epsilon+1.0
    return(old)
  return(sqroot1(n,eps_m()))
```

プログラムとしての再帰

- プログラムでも関数を定義するときに自身をもちいることができる
- 下のプログラムを実行してみるので引数の値に注意して見ていてください
- 関数は定義しただけでは実行されず，呼び出したときはじめて活性化され，仮引数が束縛される
  - 呼び出し時に環境が作成される: e.g. n は 3,2,1 と束縛される
  - その環境のもとで関数の本体が実行される

Listing 7: fact.py

```python
# Factorial
def fact (n):
  if (n == 1):
    return(1)
  else:
    return(n*(fact(n-1)))
```

呼び出し	環境
fact(3)	n←3
fact(2)	n←2
fact(1)	n←1

関数の評価と生成プロセス

fact(3) の生成プロセス

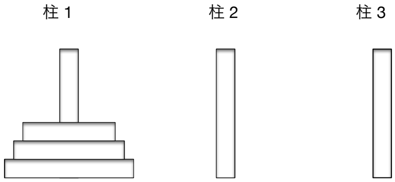
```
  fact(3)
=>(3 * fact(2)))
=>(3 * (2 * fact(1))))
=>(3 * (2 * 1)))
=>(3 * 2 ))
=>6
```

呼び出し	環境
fact(3)	n←3
fact(2)	n←2
fact(1)	n←1

Outline

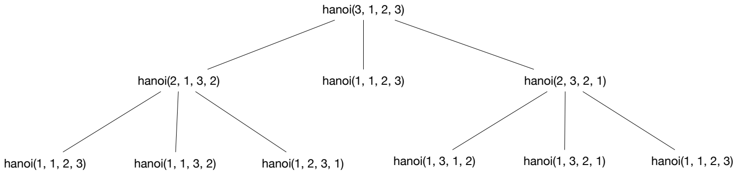
QUIZ 1

- ハノイの塔 (Tower of Hanoi) というパズルを解くプログラムを作成してください
- 目的:
  - 柱 1 から柱 2 へ移動
- 制約:
  - 柱から柱への移動は一度に 1 枚だけ
  - 大きい円盤をそれより小さい円盤の上にのせてはいけない
- 3a, 4b(CS2) クラスのサイト: https://sites.google.com/presystems.xyz/elementaryCS から hanoi-skeleton.py をダウンロードして使ってください



Quiz 1 の hint

- Basic step: 1 枚の移動
- Recursive step: n 枚から $n-1$ 枚の移動
- hanoi(n,a,b,c) は n 枚のディスクを a から b へ c を使って移動する関数
- Python でリストに要素を追加するときは append() を使う



Outline

繰り返しプログラムと再帰プログラム

- 第 1 ではレジスタマシンでいろいろな計算を実現しました
- ここではレジスタマシンと再帰関数の関係について見ていきます
- レジスタマシンと同様に再帰関数でも計算可能とみなす計算のクラスを考えてみます
- succ(y) は y のつぎの数という意味で CS 入門第一の +1 に対応
- 0 は定数 (引数なしの特別な関数)
- これら初期関数から再帰と関数合成で繰り返しプログラムと同じものが作れます
  - 関数合成は $f(g_1(x_1), \ldots, g_n(x_n))$
  - 再帰は basic step と recursive step で場合分け

末尾再帰呼び出し (Tail Recursive Call)

- Listing 8 と Listing 9 はどちらも再帰的定義ですが計算プロセスに違いがあります
- Listing 8 は fact(n) を得るために fact(n-1) の後の計算 (n ×□) を覚えてなければばならない
- Listing 9 はその必要はなく，計算の状態 “カウンタと途中までの積” を覚えておくだけで良い
- Listing 9 のような形を末尾再帰的といいます

Listing 8: fact.py

```
# Factorial
def fact (n):
  if (n == 1):
    return(1)
  else:
    return(n*(fact(n-1)))
```

Listing 9: fact.py

```
# Factorial
def fact_iter (n):
  def fact_iter1 (prod, cnt, max):
    if cnt > max:
      return(prod)
    else:
      return(fact_iter1(prod*cnt,cnt+1,
             max))
  return(fact_iter1(1,1,n))
```

末尾呼び出し (Tail Call)

- 末尾再帰は繰り返し `while`, `for` に書き換えられます
- 繰り返しは再帰関数の特別な場合と見なせる
- 再帰は関数を呼び出すたびに資源を消費するが，繰り返しは一定
- https://sites.google.com/presystems.xyz/elementaryCS の four-ops-rec.py も参考にしてみてください．

再帰プロセス

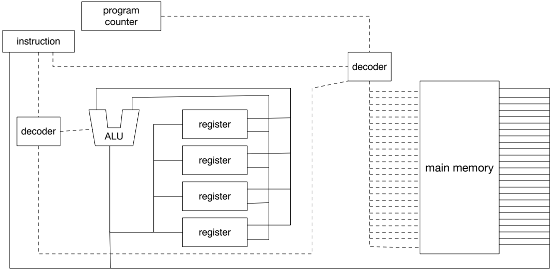
```
  fact(4)
=>(4 * fact(3))
=>(4 * (3 * fact(2)))
=>(4 * (3 * (2 * fact(1))))
=>(4 * (3 * (2 * 1)))
=>(4 * (3 * 2 ))
=>(4 * 6)
=>24
```

繰り返しプロセス

```
  fact-iter(4)
=>fact-iter1( 1,1,4)
=>fact-iter1( 1,2,4)
=>fact-iter1( 2,3,4)
=>fact-iter1( 6,4,4)
=>fact-iter1(24,5,4)
=>24
```

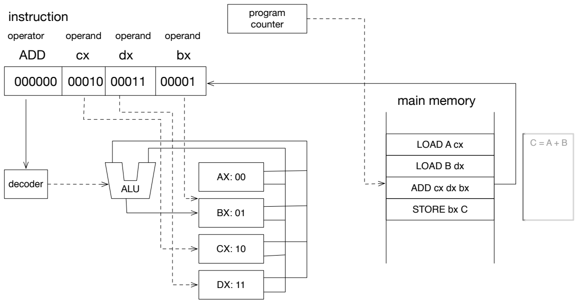
CPU とメモリ

- コンピュータの命令自体も符号化されてます
- CPU (Central Processing Unit) ごとに命令セットも符号も異なっています
- ここでは CPU が命令を実行するサイクルについて見てみます



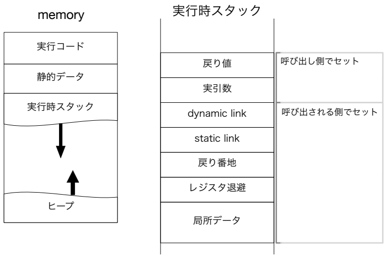
演算のサイクル

1. Instruction に命令をフェッチ
2. メインメモリからレジスタにデータを移動
3. ALU (Arithmetic and Logic Unit) がレジスタからデータを取り出す
4. ALU で演算
5. 結果をレジスタに書き込む
6. レジスタからメインメモリにデータを移動
7. ADD cx dx bx という命令を例にすると下図のようになります



関数を呼び出したとき

- 関数を呼び出すたびに下図の右側のように実行時スタックの領域に保存する
  - 活性レコード (activation record) という
- 終了すれば活性レコードは開放される
- 多くのプログラミング言語は末尾再帰を繰り返しに変換してくれないので繰り返し構文が用意されている



Outline

再帰のまとめ

- 再帰とはある問題をそれより簡単な同じ問題に分解して問題を解く方法
- 既にわかっているそれより前のものから求める
- 一見複雑な問題も再帰的に定義すると単純な計算で解ける
- それ以上分割出来ない問題を解き，結果からより大きい問題の結果を得るというもの (分割統治法 (Divide and Conquer) と呼ばれる)
- E.g. Quiz 1 では複雑な問題をより小さい同じ問題に分解して解いていっている
- 再帰は非常に強力な解法になる

課題 S

- https://sites.google.com/presystems.xyz/elementaryCS/ から fib-skeleton.py をダウンロード
- フィボナッチ数を求める問題です
- まだ，お話していない部分も含まれるのですぐに取り掛からなくていいです
- ソースコード中のコメントを参照して (2)-(4) の問に答えて見てください
- 提出は任意です
- 提出は出来たところまででいいです
- 出来なかったところはコメントにしてください

Part III

アルゴリズム

よいアルゴリズム，わるいアルゴリズム，ふつうのアルゴリズム

Outline

アルゴリズム

- 前回まで再帰関数についてお話しました
- これはアルゴリズム的に計算可能という概念を定義する上で重要な役割をします
- 関数 $f : D^n \to D$ が計算可能とは，それを計算するアルゴリズム（算法）が存在すること
  - アルゴリズムとは停止する命令の有限の列をいう
  - コンピュータにプログラムできるなにがしか
- たとえば,
  - $x \in D$ として $f(x) = G(f \upharpoonright x)$
  - D を自然数とすれば $(N; 0, S)$
  - $+, -, \times, \div, \mathrm{mod}, <$ を再帰や合成で構成
- 計算可能な関数のクラスも理論的に興味深いが，それは別の機会

良いアルゴリズム

- ここでは，すでに計算可能な関数のクラスがあって，その計算の複雑さについて議論する
- 計算のやり方には良い方法とわるい方法がある
- 良い方法というのは少ない計算で目的の値を求めること
- 良い悪いの尺度のひとつ時間計算量について見ていく
- $0, S, +, -, \times, \div, \mathrm{mod}, <$ のような演算が何回必要かを測る
- 単にプログラムが動けばいいでなくて

まずは最大公約数

- 関数 GCD をふたつの自然数の公約数のうち最大のものを求める関数とする
- アルゴリズムでは具体的に最大公約数をもとめる計算の仕方を議論する
- 計算の仕方はいくつか考えられる
  - アルゴリズムの書き方も自然言語で書いたりいろいろ

素朴な方法

ふたつの整数 x, y について 1 から順に $\min(x, y)$ まで割って，割り切れる最大の整数

ユークリッドの互除法

x_1, x_2 の最大公約数は x_1 を x_2 で割ったときの剰余と x_2 の最大公約数に等しい

$$
\begin{aligned}
x_1 &= n_1 x_2 + x_3 \\
x_2 &= n_2 x_3 + x_4 \\
x_3 &= n_3 x_4 + x_5 \\
&\vdots
\end{aligned}
$$

最大公約数 (Greatest Common Divisor) のプログラム

- それぞれの実行時間を比較するので見ていてください
  - 32204 と 14744 の最大公約数
  - 34 と 21 の最大公約数

Listing 10: gcd.py

```
def gcd(x,y):
  def min(x,y):
    if (x>y):
      return y
    else:
      return x
  gcm=1
  n=min(x,y)
  for i in range(1,n+1):
    if (x%i==0) and (y%i==0):
      gcm=i
  return (gcm)
```

Listing 11: gcd.py

```
def euclid(x1,x2):
  def euclid1(x1,x2):
    if x2==0:
      return (x1)
    else:
      return (euclid1(x2,x1%x2))
  def swap(x1,x2):
    return x2,x1
  if (x1<x2) :
    x1,x2 = swap(x1,x2)
  return(euclid1(x1,x2))
```

実行時間の違いについての考察

- Naive な方法
  - 割り算の回数を数えてみる
  - $2n$ 回で n は x か y の小さい方になっている
- ユークリッドの互除法
  - n よりは少ない回数で計算できる

euclid(32204,14744)

```
  euclid(32204,14744)
=>euclid1(14744,2716)
=>euclid1(2716,1164)
=>euclid1(1164,388)
=>euclid1(388,0)
=>388
```

euclid(34,21)

```
  euclid(34,21)
=>euclid1(21,13)
=>euclid1(13,8)
=>euclid1(8,5)
=>euclid1(5,3)
=>euclid1(3,2)
=>euclid1(2,1)
=>euclid1(1,0)
=>1
```

ユークリッドの互除法の演算回数の見積もり

- 最悪の場合について考える
- 割り算の商がいつまでも 1 であるとき最悪になる
- となり合う 2 つの Fibonacci 数のとき最悪になる
- ユークリッドの互除法を用いると下の左図のような等式の列を得る

$$
\begin{aligned}
r_0 &= q_1 r_1 + r_2, (r_2 < r_1) \\
r_1 &= q_2 r_2 + r_3, (r_3 < r_2) \\
r_2 &= q_3 r_3 + r_4, (r_4 < r_3) \\
&\vdots \\
r_{n-2} &= q_{n-1} r_{n-1} + r_n, (r_n < r_{n-1}) \\
r_{n-1} &= q_n r_n \\
&\text{where } r_0 (= x_1) > r_1 (= x_2)
\end{aligned}
$$

$$
\begin{aligned}
21 &= 13 + 8 \\
13 &= 8 + 5 \\
8 &= 5 + 3 \\
5 &= 3 + 2 \\
3 &= 2 + 1 \\
2 &= 2 \cdot 1
\end{aligned}
$$

ユーグリッドの互除法の演算回数

- $r_1 > r_2 > \ldots > r_{n-1} > r_n > r_{n+1} = 0$
- 小さい方の数 r_1 は $n+1$ 番目の Fibonacci 数 f_{n+1} であるかそれより大きい

$$
\begin{aligned}
r_0 &= r_1 + r_2 \\
r_1 &= r_2 + r_3 \\
r_2 &= r_3 + r_4 \\
&\vdots \\
r_{n-2} &= r_{n-1} + r_n \\
r_{n-1} &= q_n r_n \\
& \quad (q_n \geq 2, r_n < r_{n-1})
\end{aligned}
$$

$$
\begin{aligned}
r_n &\geq 1 (= f_2) & & \\
r_{n-1} &\geq 2r_n & &\geq 2f_2 (= f_3) \\
r_{n-2} &\geq r_{n-1} + r_n & &\geq f_3 + f_2 (= f_4) \\
&\vdots & & \\
r_2 &\geq r_3 + r_4 & &\geq f_{n-1} + f_{n-2} (= f_n) \\
r_1 &\geq r_2 + r_3 & &\geq f_n + f_{n-1} (= f_{n+1})
\end{aligned}
$$

ユークリッドの互除法の演算回数—つづき

- $r_1 = N$ として，

$$
\begin{array}{rcll}
N & \geq & f_{n+1} & \\
N & \geq & \phi^{n-1} & (\text{by } f_{n+1} > \phi^{n-1} \text{ for } n > 2 \\
 & & & \text{where } \phi = \frac{(1+\sqrt{5})}{2}) \\
\log_{10} N & \geq & (n-1)\log_{10}\phi & \\
\log_{10} N & \geq & (n-1)\log_{10}\phi > \frac{n-1}{5} & \\
n-1 & < & 5 \cdot \log_{10} N & \\
n-1 & < & 5 \cdot k & (N\text{を}k\text{桁の数として} \\
 & & & \log_{10} N < k) \\
n & \leq & 5 \cdot k & (n, k\text{は正整数}) \\
n & \leq & 5 \cdot (\lfloor \log_{10} N \rfloor + 1) &
\end{array}
$$

- Naive な方法の演算回数 N 回 よりは少ない

Quiz2: べき乗 (Power) の計算

- 下の例を参考にべき乗を求める良いアルゴリズムを考えてみてください
- b^8 を求めるのに乗算を 8 回と 3 回
- 8 回より 3 回の方が良い子
- power-skeleton.py が https://sites.google.com/presystems.xyz/elementaryCS/ に置いてあるので, ループに変更して実行時間を比較してみてください
- 提出はいつもの T2Schola からソースコードを提出

Example (b^8 の計算)

$$b\times(b\times(b\times(b\times(b\times(b\times(b\times b))))))$$

$$\begin{aligned} b^2 &= b \times b \\ b^4 &= b^2 \times b^2 \\ b^8 &= b^4 \times b^4 \end{aligned}$$

べき乗 (Power) のヒント

- 指数は偶数のとき半分に，奇数のとき -1 減る

$$
\begin{aligned}
b^n &= (b^{\frac{n}{2}})^2 \\
&\quad n\text{が偶数のとき} \\
b^n &= b \times b^{n-1} \\
&\quad n\text{が奇数のとき}
\end{aligned}
$$

Listing 12: power.py

```
def fast_power(b,n):
  def square(x):
    return(x*x)
  def is_even(n):
    if (n%2==0):
      return(True)
    else:
      return(False)
  if (n==0):
    return(1)
  else:
    if (is_even(n)):
      return(square(fast_power(b,n/2)))
    else:
      return((b*fast_power(b,n-1)))
```

べき乗計算の掛け算の回数

- 各呼び出しで掛け算の回数は 1 回
- 偶数は何回出てくるか，奇数は何回出てくるかを数える
- $T(n)$ を n 乗の計算に必要な掛け算の回数とする
- $T(n) = T(\lfloor \frac{n}{2} \rfloor) + g(n)$ である，g(n) は奇数が出現した回数
- $T(n) = \begin{cases} 0 & n = 0 \\ T(\lfloor \frac{n}{2} \rfloor) + 1 & n \geq 1 \end{cases}$
- $n \geq 1$ の場合について考える

$$
\begin{aligned}
T(n) &= T(\lfloor \tfrac{n}{2} \rfloor) + 1 \\
&= T(\lfloor \tfrac{n}{2^2} \rfloor) + 1 + 1 \\
&= T(\lfloor \tfrac{n}{2^3} \rfloor) + 1 + 1 + 1 \\
&\vdots \\
&= T(\lfloor \tfrac{n}{2^k} \rfloor) + 1 + \cdots + 1 \\
&= T(\lfloor \tfrac{n}{2^k} \rfloor) + k \\
& \tfrac{n}{2^k} = 1 \text{ から } k = \log_2 n,
\end{aligned}
$$

ゆえに全体で

$$T(n) = \log_2 n + g(n)$$

べき乗計算の掛け算の回数 —奇数の出現回数

- 何回奇数が出てくるかは n を二進表記した時の 1 の数による
  - $\frac{1}{2}$ するとは右に 1 ビットシフト
  - 最下位ビットが 1 のとき奇数
- $\lfloor \log_2 1000 \rfloor + 6$=15 回

偶数の出現回数

$\lfloor \log_2 1000 \rfloor = 9$

奇数の出現回数

$$
\begin{aligned}
(1000)_{10} = \quad & (1111101000)_2 \\
& \Rightarrow_{\frac{1}{2}} 0111110100 \\
& \Rightarrow_{\frac{1}{2}} 0011111010 \\
& \Rightarrow_{\frac{1}{2}} 0000111101 \\
& \Rightarrow_{-1} 0000111100 \\
& \Rightarrow_{\frac{1}{2}} 0000011110 \\
& \cdots \\
& \Rightarrow_{\frac{1}{2}} 0000000001 \\
& \Rightarrow_{-1} 0000000000
\end{aligned}
$$

Outline

時間計算量

- アルゴリズムの良し悪しを測る尺度についてみていきます
- 良し悪しの基準
  - 良いアルゴリズム，わるいアルゴリズムを実行時間を計測して比較
  - 入力に対し結果を得るまでの必要な演算 (四則演算や比較演算) の回数を数え上げて比較
- 実行時間は CPU や演算の実現方式に依存し計算しようとすると複雑な式になってしまう
- 演算回数 (時間計算量という) をもとに単純化して見積もることにする
  - CPU や実現方式といったものは無視
- 実行時間がどう変化するかの傾向を見積もれれば十分

Example

入力サイズ n として一方は n^2，もう一方は n^3 回演算を行うとすれば，n が十分に大きいときには前者の方が良い

Big–O 記法

- n を入力のサイズとして，演算回数 $f(n)$ で傾向をつかむ

Definition

2 つの関数 $f, g \colon N \rightarrow R^{>0}$ とする．

$$\forall n > k : f(n) \leq c \cdot g(n)$$

となるような c, k が存在するならば $f(n) \Rightarrow O(g(n))$ とする

Example

$f(n) = n^2 + 2n + 1$ は，たとえば $c = 4, k = 1$ として $g(n) = n^2$，よって $f(n)$ は $O(n^2)$ となる．

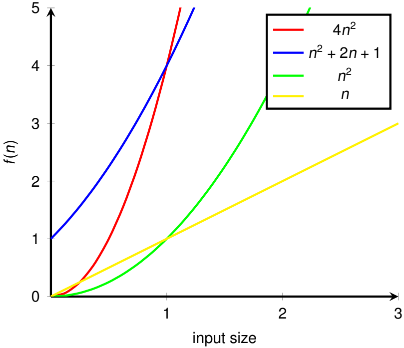
$f(n) = n^2 + 2n + 1$ は $O(n^2)$

- $n > 1(=k)$ で $2n \leq 2n^2, 1 \leq n^2$ なので

$$0 \leq n^2 + 2n + 1 \leq n^2 + 2n^2 + n^2 = 4n^2$$

- $c = 4, k = 1, g(n) = n^2$ として，
- $f(n) = n^2 + 2n + 1 < 4 \cdot n^2$ —$\{f(n) \leq c \cdot g(n)\}$
- $n \geq 2$ とすると $2n < n^2, 1 < n^2$

$$0 \leq n^2 + 2n + 1 \leq n^2 + n^2 + n^2 = 3n^2$$



ユーグリットの互除法とべき乗のアルゴリズムのオーダ

Example (ユーグリットの互除法)

- Lame の定理: 割り算の回数は，小さい数 n の桁数の 5 倍以下である

$\forall n \geq 10 : 5(\lfloor \log_{10} n \rfloor + 1) \leq 5(\log_{10} n + 1)$ よって $O(\log n)$

Example (べき乗)

- j を奇数の出現回数として，

$\forall n \geq 2 : (\log_2 n) + j < (1 + j) \log_2 n$ よって $O(\log n)$.

Outline

Sorting

- Sorting は 2 つの要素の比較と入れ替えだけを用いて，与えられた集合の要素のリストを昇順（降順）に並べたリストを作ること
- 例えば {3,2,4,1,5} と与えられていれば {1,2,3,4,5} と並べたリストを作成する
- データベースシステムなど多くの場面で利用されている
- 多くのアルゴリズムが提案されている
- ここではバブルソート (bubble sort)，挿入ソート (insertion sort)，クイックソート (quick sort) を例に機械的にアルゴリズムから計算量を求めていく方法を見てみる

Bubble Sort

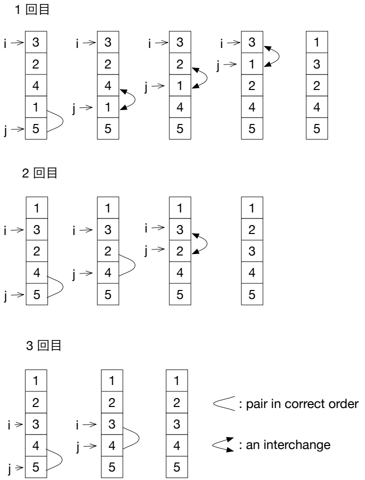
様相

Listing 13: 擬似コード

```
1 def bubble_sort(numbers):
2   for i in {1..n-1}:
3     for j in {n..i+1}:
4       if (numbers[j]<numbers[j-1]):
5         numbers[j],numbers[j-1]を入替え
6   return(numbers)
```

計算量

- 配列のサイズを n として
- 4-5 行で比較と入れ替えは 1 回ずつ
- 3 行の演算回数は各繰り返しの演算回数の和; $n - i$ 回繰り返すので $(n - i)$
- 2 行の演算回数は各繰り返しで $n - i$ を $(n - 1)$ 回繰り返すので
- 全体で
  $\sum_{i=1}^{n-1} n - i = \frac{n(n-1)}{2} = (\frac{n^2}{2} - \frac{n}{2}) < n^2$
- よって $O(n^2)$



Insertion Sort

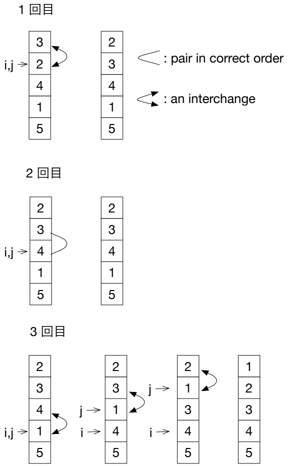
様相

Listing 14: 擬似コード

```
def insertion_sort(numbers):
  for i in {2..n}:
    j=i
    while (numbers[j]<numbers[j-1]):
      numbers[j],numbers[j-1]を入替え
      j=j-1
  return(numbers)
```

計算量

- 配列のサイズを n として $O(n^2)$ が求められるはず
- 各自で確かめてみてください (提出不要)



宿題 2

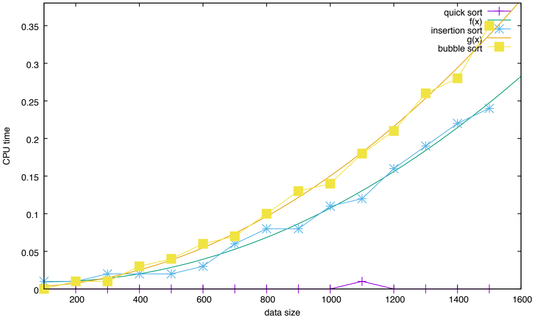
Quiz: Sort

- Input: 任意の長さの任意の整数の列
- Output: 昇順に整列した列
- https://sites.google.com/presystems.xyz/elementaryCS/ から sort-skeleton.py をダウンロード
- 各ソートアルゴリズムを完成させて実行時間を比較して見てください
- 提出は sort.py としてソースコードだけ提出

Quiz 3

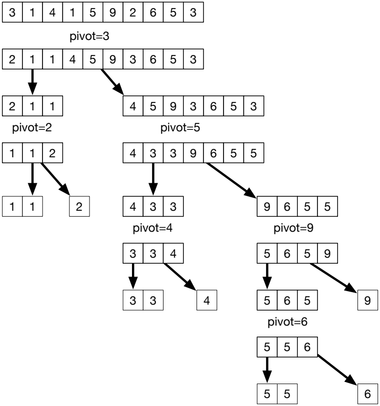
Quiz: Quick Sort

- Input: 任意の長さの任意の整数の列
- Output: 昇順に整列した列
- https://sites.google.com/presystems.xyz/elementaryCS/ から quick-sort-skeleton.py をダウンロード
- 各ソートアルゴリズムを完成させて実行時間を比較して見てください
- 提出は quick-sort.py としてソースコードだけ提出



Quiz 3 の hint

- Basic step: 異なる値がないときには終了
- Recursive step:
  1. 中間付近の値 pivot を選ぶ: findpivot()
  2. pivot より小さい値を左側に，大きい値を右側に集める: partition()
  3. 左側，右側でそれぞれソートする



Quick Sort の計算量

Listing 15: 擬似コード

```
def partition(numbers,left,right,pivot):
  while left<=right:
    numbers[left],numbers[right] を入替え
    while (numbers[left]<pivot):
      left=left+1
    while (numbers[right]>=pivot):
      right=right-1
  return(left)
def quick_sort1(numbers,front,rear):
  pivot=findpivot(numbers,front,rear)
  if (値がすべて同じ時):
    return(numbers)
  else:
    k=partition(numbers,front,rear,numbers[pivot])
    quick_sort1(numbers,front,k-1)
    quick_sort1(numbers,k,rear)
    return(numbers)
```

Best case

- Pivot が中央のとき最適になる
- findpivot と partition はそれぞれ $O(n)$
- 各繰り返しでの計算回数を $T(n)$ として
- $T(n) = 2T(\frac{n}{2}) + n$
- これを解いて $T(n) = n + n\log_2(n)$
- したがって，全体で $O(n\log n)$

Worst case

- 分割した一方が 1 個のとき最悪になる
- 各繰り返しでの計算回数を $T(n)$ として
- $T(n) = T(n-1) + T(1) + n$
- これを解いて
  $T(n) = \frac{n^2}{2} + (T(1) + \frac{1}{2})n + T(1)$
- したがって，$O(n^2)$

Master Theorem

Theorem

T を次の条件を充す単調関数とする．

$$T(n) = aT(\tfrac{n}{b}) + cn^d \text{ for } n > 1$$

ここで，$a \geq 1, b \geq 2, c > 0, d \geq 0$ の定数とし，$\frac{n}{b}$ は $\lfloor \frac{n}{b} \rfloor$ か $\lceil \frac{n}{b} \rceil$ とする．このとき，

$$T(n) = \begin{cases} O(n^d) & \text{if } a < b^d \\ O(n^d \log n) & \text{if } a = b^d \\ O(n^{\log_b a}) & \text{if } a > b^d \end{cases}$$

- 計算量が漸化式で表されるような場合はこの定理を利用することもできる
- Best case はこれに当てはまる $(a = b = 2, d = 1)$

Outline

よくみる時間計算量

時間計算量	呼び方	例
$O(1)$	Constant complexity	hash-table,
$O(\log n)$	Logarithmic complexity	binary search
$O(n)$	Linear complexity	loops
$O(n \log n)$	n log n complexity	quick sort
$O(n^b)$	Polynomial complexity	bubble sort, insertion sort
$O(b^n)$, where $b > 1$	Exponential complexity	tower of Hanoi
$O(n!)$	Factorial complexity	travelling-salesman problem

計算の複雑さのクラス

- 計算量をオーダであらわすことをみてきました
- 計算量的に実行可能であるかという境界はどこにあるか，という疑問が出てくるのは自然なこと
- ひとつは多項式境界 (polinomial bound) でこのクラスを P (Polynominal time) と表し，
- もう一つを指数境界 (exponential bound) でこのクラスを NP (Non-deterministic Polynomial time) と表す．
  - このふたつは，決定性か非決定性かの違いや，
  - 解を得るのに全数探索しなければならないかどうかの違いとして見て取れる
- 有名な問題が P ≠ NP かどうかという問題

Part IV

大きい数と小さい数の計算

大きい数と小さい数の計算

13 数の表現
- 非負整数の表現
- 負の数の表現
- 計算機内の計算
- 実数の表現
- 浮動小数の算術演算

14 誤差のおはなし
- 丸め誤差 (roundoff error)
- 情報落ち誤差
- 打ち切り誤差 (truncation error)

Outline

非負整数のコンピュータ内での表現

- 10 進数から 2 進数への変換

Example (10 進 ⇔2 進)

$$
\begin{array}{l}
2)19 \cdots 1 \\ \hline
2)9 \cdots 1 \\ \hline
2)4 \cdots 0 \\ \hline
2)2 \cdots 0 \\ \hline
2)1 \cdots 1 \\ \hline
\quad 0
\end{array}
$$

Low

High

$$
\begin{array}{c}
1 \times 2^0 = 1 \\ \hline
1 \times 2^1 = 2 \\ \hline
0 \times 2^2 = 0 \\ \hline
0 \times 2^3 = 0 \\ \hline
1 \times 2^4 = 16 \\ \hline
19
\end{array}
$$

負の数の表現

- 負の数をあらわすには補数表現をもちいます
- それでは 2 の補数 (2's complement) を求めてみましょう

2 の補数表現

1. 2 進表記において各ビットを反転する
2. それに 1 を足す

Example (-8~7 (2 進 4 桁) の 2 の補数表現)

$(1000)_{(2)} \Rightarrow (1001)_{(2)} \Rightarrow (1010)_{(2)} \Rightarrow (1011)_{(2)} \Rightarrow (1100)_{(2)} \Rightarrow$
$(1101)_{(2)} \Rightarrow (1110)_{(2)} \Rightarrow (1111)_{(2)} \Rightarrow (0000)_{(2)} \Rightarrow (0001)_{(2)} \Rightarrow$
$(0010)_{(2)} \Rightarrow (0011)_{(2)} \Rightarrow (0100)_{(2)} \Rightarrow (0101)_{(2)} \Rightarrow (0110)_{(2)} \Rightarrow$
$(0111)_{(2)} \Rightarrow (1000)_{(2)}$

- 最上位ビットがサインビットになっている

- 2 進 n 桁の数 a, b ($A_k, B_k \in \{1, 0\}$)
    - $a: A_{n-1}A_{n-2}\cdots A_1A_0$
    - $b: B_{n-1}B_{n-2}\cdots B_1B_0$
- a, b はそれぞれ

$$a = \sum_{k=0}^{n-1} 2^k A_k$$

$$b = \sum_{k=0}^{n-1} 2^k B_k$$

- b の各桁を反転させたものを $\overline{B_k}$ として b の補数 $\overline{b}$ は

$$\overline{b} = \sum_{k=0}^{n-1} 2^k \overline{B_k} = \sum_{k=0}^{n-1} 2^k (1 - B_k) = (2^n - 1) - b$$

計算機内の計算—Cont.

- $\overline{b} = (2^n - 1) - b$ より

$$\begin{aligned} a - b &\Rightarrow a - ((2^n - 1) - \overline{b}) \\ &= a + \overline{b} + 1 - 2^n \end{aligned}$$

- 引き算は補数を足すことで表す
- $\overline{b} + 1$ は 2 の補数
- -2^n は最上位の桁上がりは無視

Example (引き算の例)

- 4 桁の 2 進数と仮定して $6 - 3$ と $3 - 6$

$$\begin{array}{ccccc} & 0 & 1 & 1 & 0 \\ + & 1 & 1 & 0 & 1 \\ \hline & 0 & 0 & 1 & 1 \end{array} \qquad \begin{array}{ccccc} & 0 & 0 & 1 & 1 \\ + & 1 & 0 & 1 & 0 \\ \hline & 1 & 1 & 0 & 1 \end{array}$$

実数の表現

浮動小数 (floating point number)

- 浮動小数 ab^e
  - a は仮数 (significand or coefficient)， b は底 (base)， e は指数 (exponent) と呼ぶ
- $\frac{1}{b} \leq |a| < 1$ のとき正規浮動小数 (normalized floating point number) と云う
- 上のような変換を正規化 (normalizaiton) と云う
- 符号，指数，仮数で一意に決定できます

Example (正規浮動小数)

$$
\begin{array}{rcl}
1.234 & \Rightarrow & +0.1234 \times 10^{1} \\
-12.34 & \Rightarrow & -0.1234 \times 10^{2} \\
0.01234 & \Rightarrow & +0.1234 \times 10^{-2}
\end{array}
$$

実数の 2 進表記

- 実数も 2 進表記に変換した上で正規化します
- $13.6875_{(10)}$ を 2 進数へ変換してみます

Example (10 進実数 $13.6875_{(10)}$ を 2 進数へ)

$$
\begin{array}{l}
2)13 \cdots 1 \quad \textit{Low} \\
\underline{} \\
2)6 \cdots 0 \\
2)3 \cdots 1 \\
2)1 \cdots 1 \quad \textit{High} \\
0
\end{array}
$$

$$
\begin{array}{l}
.6875 \times 2 = 1.375 \quad \textit{High} \\
.375 \times 2 = 0.75 \\
.75 \times 2 = 1.50 \\
.5 \times 2 = 1.00 \quad \textit{Low}
\end{array}
$$

- $13_{(10)}$ は $1101_{(2)}$
- $.6875_{(10)}$ は $.1011_{(2)}$
- ゆえに，$13.6875_{(10)}$ は $1101.1011_{(2)}$ となる

実数の 2 進表記 - Cont.

- 得られた 2 進数を正規化します
- 最上位ビットが 1 になるようにします (注意: 正規化の定義と違っているので注意)

Example ($1101.1011_{(2)}$ を正規化)

$1101.1011_{(2)} \Rightarrow 0.11011011 \times 2^4$

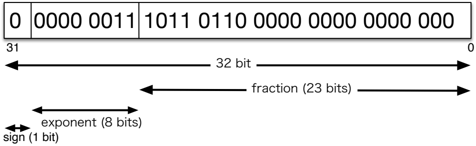
- 符号: +
- 指数: 4
- 仮数: 0.11011011

実数の 2 進表記 - Cont.

- 符号，指数，仮数が正規化によって決まります
- これを 32 ビットで表す
- 右に小数点を一つ移動
- 最上位ビットは必ず 1 になるので省略
- 規格 IEEE 754 はもうひと段階

Example (指数部，仮数部)

- 1.1011011×2^3 の符号，指数，仮数は以下のとおり
  - 符号 (sign): +
  - 指数 (exponent): 3
  - 仮数 (significand): 1.1011011



宿題 3: 浮動小数

- 実数の浮動小数表現をやってみてください

宿題 3

- 35.75 を浮動小数で表現してみてください
- T2Schola に小テストがあるのでそれに答えてください

宿題 3: 回答

- 2 進へ変換: 100011.11
- 正規化: 1.0001111×2^5
- 32 bit 形式に: 0 0000 0101 0001 1110 0000 0000 0000 000
  - 1. は省略
- 規格 IEEE 754 では下駄 (bais) をはかせるので $5 + 127 = 132 \Rightarrow 1000\ 0100$
- 32 bit 形式に: 0 1000 0100 0001 1110 0000 0000 0000 000

浮動小数演算の変な現象 1

- roundoff.py を実行してみます
- 結果が予測と少し違うことになります
- $0.6_{(10)}$ は $(0.1001\ldots)_{(2)}$, $0.4_{(10)}$ は $(0.0110\ldots)_{(2)}$, $0.2_{(10)}$ は $(0.0011\ldots)_{(2)}$ となります

Listing 16: roundoff.py

```
if (float(0.6)-float(0.4)==float(0.2)):
   print("is equal")
else:
   print("not equal")
```

浮動小数演算の変な現象 2

- machine_epsilon.py を実行してみます
- 結果が予測と少し違うことになります
- この原因についてみていきます

Listing 17: machine_epsilon.py

```
# Machine epsilon
import sys

epsilon, old, prod =1.0, 0.0, 0.0
cnt=0
while (prod!=1.0):
  print(epsilon)
  old = epsilon
  cnt=cnt+1
  epsilon=epsilon/2.0
  prod=epsilon+1.0
print("Calculated machine epsilon:",old)
print("System information in Python:",sys.float_info.epsilon)
```

浮動小数数の算術演算

- 正規化した 2 つの浮動小数 X, Y を $X = F_x \times 10^{e_x}, Y = F_y \times 10^{e_y}$ とする
- 乗算: $XY = (F_x \times 10^{e_x})(F_y \times 10^{e_y}) = F_x F_y \times 10^{e_x + e_y}$
- 除算: $\frac{X}{Y} = \frac{(F_x \times 10^{e_x})}{(F_y \times 10^{e_y})} = \frac{F_x}{F_y} \times 10^{e_x - e_y}$
- 加算・減算: $X \pm Y = (F_x \times 10^{e_x}) \pm (F_y \times 10^{e_y}) = (F_x \pm F_y \cdot 10^{e_y - e_x}) \times 10^{e_x}$
  - ただし，$e_x \geq e_y$
  - $F_y \cdot 10^{e_y - e_x}$ は指数を大きい方に揃えたときの Y の仮数
- 演算結果も正規化するので指数は調整が必要

Example (算術演算の例)

$$
\begin{array}{lll}
 & 0.2184 & \times 10^2 \\
\times & 0.2512 & \times 10^2 \\
\hline
 & 0.07998208 & \times 10^4 \\
= & 0.7998208 & \times 10^3
\end{array}
\qquad
\begin{array}{lll}
 & 0.2844 & \times 10^3 \\
+ & 0.4162 & \times 10^1 \\
\hline
 & 0.288562 & \times 10^3
\end{array}
$$

Outline

誤差とは

- コンピュータの中では実数は有限個の 0 と 1 の組み合わせ (浮動小数) で表しています
- なので，本来あるべき真値を適当な浮動小数で近似している
- 近似値-真値 を誤差という

丸め誤差 (roundoff error)

- 表現可能な範囲に丸めることを丸め誤差という (Listing 16 参照)
- 演算結果も丸める
- Z を演算結果とする
- d 桁だけ記憶できるとして，先頭の d 桁を F，残りを f とすると，$Z = F \cdot 10^{e_z} + f \cdot 10^{e_z - d}$
- f の値で四捨五入することにして

$$\begin{array}{ll} |f| < 0.5\text{のとき} & |Z| = |F| \cdot 10^{e_z - d} \\ |f| \geq 0.5\text{のとき} & |Z| = |F| \cdot 10^{e_z - d} + \cdot 10^{e_z - d} \end{array}$$

- 丸め誤差 ϵ_z とすれば

$$\begin{array}{ll} |f| < 0.5\text{のとき} & |\epsilon_z| = |f| \cdot 10^{e_z - d} \\ |f| \geq 0.5\text{のとき} & |\epsilon_z| = |1 - f| \cdot 10^{e_z - d} \end{array}$$

Example ($0.2844 \cdot 10^3 + 0.4162 \cdot 10^1$)

- $0.2844 \cdot 10^3 + 0.4162 \cdot 10^1 = 0.2885 \cdot 10^3 + 0.6200 \cdot 10^{-1}$
- $|Z| = 0.2885 \cdot 10^3 + 10^{3-4} = 0.2886 \cdot 10^3$
- $|\epsilon_z| = |1 - 0.6200| \cdot 10^{3-4} = 0.48 \cdot 10^{-1}$

情報落ち誤差 (loss of trailing digits)

- 絶対値が大きく異なる 2 つの数の加減算では小さい数が無視されることがある
- Listing 17 で見たような場合

打ち切り誤差 (truncation error)

- コンピュータでは無限に繰り返して値をもとめることはできない
- 有限回の計算で値を計算し，それを求める値の近似値としてもちいる
- このときの誤差を打切り誤差という

Example (sin(*x*) のマクローリン展開)

- $sin(x) = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \cdots + (-1)^n \frac{x^{2n+1}}{(2n+1)!} + \cdots$
- gnuplot で試してみてください

Example (平方根の計算)

- newton.py を参照 プログラム例
- $\sqrt{a}$ を求めてみる
- $f(x) = x^2 - a$ として $f(x) = 0$ となる x を求める
- $k+1$ 番目の近似値 x_{k+1} を

$$x_{k+1} = x_k - \frac{f(x_k)}{f'(x_k)} = \frac{1}{2}(x_k + \frac{a}{x_k})$$