

# コンピュータサイエンス入門第二

永藤 直行

4th quarter

# Part I

## Prologue

# Prologue

elementaryCS-  
2nd

Naoyuki  
Nagatou

教科書, 参考  
文献

講義概要

評価基準

1 教科書, 参考文献

2 講義概要

3 評価基準

# Outline

elementaryCS-  
2nd

Naoyuki  
Nagatou

教科書，参考  
文献

講義概要

評価基準

1 教科書，参考文献

2 講義概要

3 評価基準

# 参考図書

elementaryCS-  
2nd

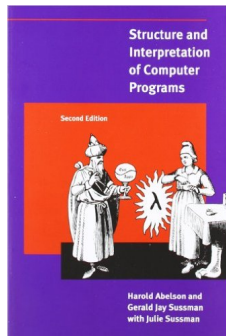
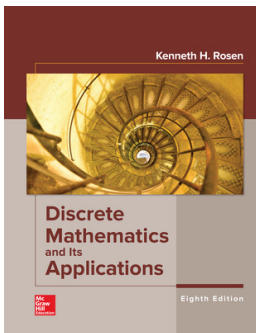
Naoyuki  
Nagatou

教科書, 参考  
文献

講義概要

評価基準

- Discrete Mathematics and its Applications
- Structure and Interpretation of Computer Programs  
<http://web.mit.edu/alexmv/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

elementaryCS-  
2nd

Naoyuki  
Nagatou

教科書，参考  
文献

講義概要

評価基準

1 教科書，参考文献

2 講義概要

3 評価基準

# 講義概要

elementaryCS-  
2nd

Naoyuki  
Nagatou

教科書, 参考  
文献

講義概要

評価基準

- この講義も Zoom で行います
- 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

## 目標

計算でものを表すとき便利な道具と注意すべきことを会得すること

# Outline

elementaryCS-  
2nd

Naoyuki  
Nagatou

教科書，参考  
文献

講義概要

評価基準

1 教科書，参考文献

2 講義概要

3 評価基準



# 評価基準

elementaryCS-  
2nd

Naoyuki  
Nagatou

教科書, 参考  
文献

講義概要

評価基準

- 講義は全 7 回
- 課題: 3 回 (計 85 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

elementaryCS-  
2nd

Naoyuki  
Nagatou

再帰  
(Recursion)

再帰への導入

再帰 (Recursion)

再帰的定義

木構造

プログラムと  
しての再帰

引数の有効範囲  
(Scope)

再帰プログラム

Quiz 1

繰り返しと  
再帰

末尾再帰 (Tail  
Recursion)

コンピュータの中では

再帰のまとめ

課題 S

4

## 再帰 (Recursion)

- 再帰への導入
- 再帰 (Recursion)
- 再帰的定義
- 木構造

5

## プログラムとしての再帰

- 引数の有効範囲 (Scope)
- 再帰プログラム

6

## Quiz 1

7

## 繰り返しと再帰

- 末尾再帰 (Tail Recursion)
- コンピュータの中では

8

## 再帰のまとめ

- 課題 S

# Outline

elementaryCS-  
2nd

Naoyuki  
Nagatou

再帰  
(Recursion)

再帰への導入

再帰 (Recursion)

再帰的定義

木構造

プログラムと  
しての再帰

引数の有効範囲  
(Scope)

再帰プログラム

Quiz 1

繰り返しと  
再帰

末尾再帰 (Tail  
Recursion)

コンピュータの中では

再帰のまとめ

課題 S

4

再帰 (Recursion)

- 再帰への導入
- 再帰 (Recursion)
- 再帰的定義
- 木構造

5

プログラムとしての再帰

- 引数の有効範囲 (Scope)
- 再帰プログラム

6

Quiz 1

7

繰り返しと再帰

- 末尾再帰 (Tail Recursion)
- コンピュータの中では

8

再帰のまとめ

- 課題 S

# 再帰への導入

elementaryCS-  
2nd

Naoyuki  
Nagatou

再帰  
(Recursion)

再帰への導入

再帰 (Recursion)

再帰的定義

本構造

プログラムと  
しての再帰

引数の有効範囲  
(Scope)

再帰プログラム

Quiz 1

繰り返しと  
再帰

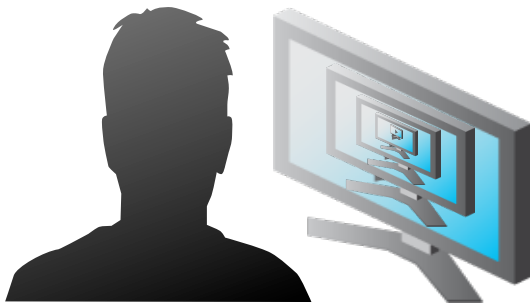
末尾再帰 (Tail  
Recursion)

コンピュータの中では

再帰のまとめ

課題 S

- 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（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

elementaryCS-  
2nd

Naoyuki  
Nagatou

再帰  
(Recursion)  
再帰への導入  
再帰 (Recursion)  
再帰的定義  
本構造

プログラムと  
しての再帰  
引数の有効範囲  
(Scope)  
再帰プログラム

Quiz 1

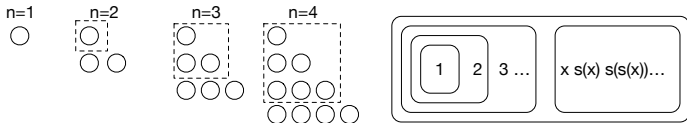
繰り返しと  
再帰

末尾再帰 (Tail  
Recursion)  
コンピュータの中では

再帰のまとめ  
課題 S

- 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知の項から決定する
- 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

elementaryCS-  
2nd

Naoyuki  
Nagatou

再帰  
(Recursion)

再帰への導入

再帰 (Recursion)

再帰的定義

本構造

プログラムと  
しての再帰

引数の有効範囲  
(Scope)

再帰プログラム

Quiz 1

繰り返しと  
再帰

末尾再帰 (Tail  
Recursion)

コンピュータの中では

再帰のまとめ

課題 S

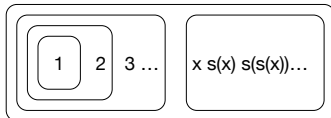
- 関数  $f$  の定義に  $x$  より小さい要素についての評価値を利用して定義することを原始再帰 (primitive recursion) と呼ぶ
- $f$  が  $x$  より小さい値について計算可能であり、いつも同じ値と仮定し、 $f \upharpoonright x$  と書く
- 定義できた最大の  $x$  のつぎの値について定義するというのを繰り返すと全体について定義できる ( $\delta$ -近似)

## 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



# 関数の再帰的定義

## 加算, 乗算の再帰的定義

- G に相当するのが succ
- Basic step に近づけるように recursive step を定義
  - $a + \text{succ}(b) =$   
Basic step:  $a + 0$  if  $b = 0$   
Recursive step:  $\text{succ}(a + b)$  otherwise

### Listing 1: 加算

```
1 # Recursive definition
2 import os
3
4 def succ (x):
5     return(x+1)
6 def pred (x):
7     return(x-1)
8 def add (a,b):
9     if (b==0):
10        return(a)
11    else:
12        return(succ(add(a, pred(b))))
```

### Listing 2: 乗算

```
15 # Multiplication
16 #
17 def mult (a,b):
18     if (b==0):
19         return(0)
20     else:
21         return(add(add(0, a), mult(a, pred(b))))
```

elementaryCS-  
2nd

Naoyuki  
Nagatou

再帰  
(Recursion)

再帰への導入

再帰 (Recursion)

再帰的定義

本構造

プログラムと  
しての再帰

引数の有効範囲  
(Scope)

再帰プログラム

Quiz 1

繰り返しと  
再帰

末尾再帰 (Tail  
Recursion)

コンピュータの中では

再帰のまとめ

課題 S



# 再帰的定義の例 1

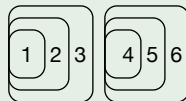
## Factorial

- $n! = n \cdot (n-1) \cdot (n-2) \cdots 3 \cdot 2 \cdot 1$
- $n!$  を決めるのに  $(n-1)!$  の値をつかう
- $G$  に相当するのが  $n \cdot (n-1)!$  という事
- 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

```
1 def fact (n):  
2     if (n == 1):  
3         return(1)  
4     else:  
5         return(n*(fact(n-1)))
```

### Example (4!)



elementaryCS-  
2nd

Naoyuki  
Nagatou

再帰  
(Recursion)

再帰への導入

再帰 (Recursion)

再帰的定義

本構造

プログラムと  
しての再帰

引数の有効範囲  
(Scope)

再帰プログラム

Quiz 1

繰り返しと  
再帰

末尾再帰 (Tail  
Recursion)

コンピュータの中では

再帰のまとめ

課題 S

# フィボナッチ数

elementaryCS-  
2nd

Naoyuki  
Nagatou

再帰  
(Recursion)

再帰への導入

再帰 (Recursion)

再帰的定義

本構造

プログラムと  
しての再帰

引数の有効範囲  
(Scope)

再帰プログラム

Quiz 1

繰り返しと  
再帰

末尾再帰 (Tail  
Recursion)

コンピュータの中では

再帰のまとめ

課題 S

- 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
1	0	1	1
2	0	1	1
3	1	1	2
4	1	2	3

# 再帰的定義の例 2

elementaryCS-2nd

Naoyuki  
Nagatou

再帰

(Recursion)

再帰への導入

再帰 (Recursion)

再帰的定義

本構造

プログラムと  
しての再帰

引数の有効範囲  
(Scope)

再帰プログラム

Quiz 1

繰り返しと  
再帰

末尾再帰 (Tail  
Recursion)

コンピュータの中では

再帰のまとめ

課題 S

## ● フィボナッチ数

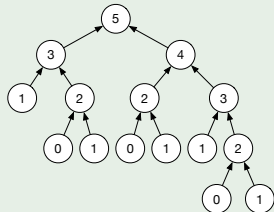
0	1	2	3	4	5...
1	1	2	3	5	8...

- $\text{fib}(S(n))$  を決めるのに  $\text{fib}(n)$  と  $\text{fib}(n-1)$  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

```
1 def fib (n):  
2   if (n==0):  
3     return(0)  
4   else:  
5     if (n==1):  
6       return(1)  
7     else:  
8       return(fib(n-1)+fib(n-2))
```

## Example (fib(5))



# Tree (木構造)

elementaryCS-  
2nd

Naoyuki  
Nagatou

再帰  
(Recursion)

再帰への導入

再帰 (Recursion)

再帰的定義

木構造

プログラムと  
しての再帰

引数の有効範囲  
(Scope)

再帰プログラム

Quiz 1

繰り返しと  
再帰

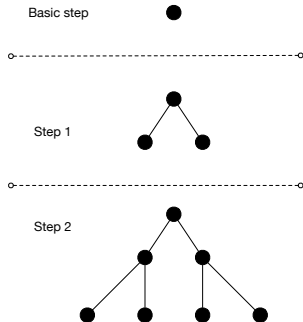
末尾再帰 (Tail  
Recursion)

コンピュータの中では

再帰のまとめ

課題 S

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



# 集合の再帰的定義

elementaryCS-  
2nd

Naoyuki  
Nagatou

再帰  
(Recursion)

再帰への導入

再帰 (Recursion)

再帰的定義

木構造

プログラムと  
しての再帰

引数の有効範囲  
(Scope)

再帰プログラム

Quiz 1

繰り返しと  
再帰

末尾再帰 (Tail  
Recursion)

コンピュータの中では

再帰のまとめ

課題 S

- 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 $\Sigma^*$ )

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

# 再帰は強力な道具

elementaryCS-  
2nd

Naoyuki  
Nagatou

再帰  
(Recursion)

再帰への導入

再帰 (Recursion)

再帰的定義

本構造

プログラムと  
しての再帰

引数の有効範囲  
(Scope)

再帰プログラム

Quiz 1

繰り返しと  
再帰

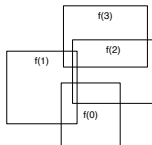
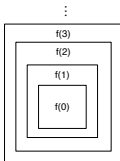
末尾再帰 (Tail  
Recursion)

コンピュータの中では

再帰のまとめ

課題 S

- 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

elementaryCS-  
2nd

Naoyuki  
Nagatou

再帰  
(Recursion)

再帰への導入

再帰 (Recursion)

再帰的定義

本構造

プログラムと  
しての再帰

引数の有効範囲  
(Scope)

再帰プログラム

Quiz 1

繰り返しと  
再帰

末尾再帰 (Tail  
Recursion)

コンピュータの中では

再帰のまとめ

課題 S

- 自然数上の四則演算を再帰で定義せよ (four-ops-rec.py 参照)
- 先の文字列の集合を参考に文字列の長さを求める関数を再帰的に定義せよ (str\_len.py 参照)
  - Hint: Basic step を空列は長さ 0 として, recursive step は一文字短い文字列より 1 長い
  - Python では `str[1:]` とすると文字列の 2 番目以降の文字列を得ることができる
  - 空列は `not str` で判定

# Outline

elementaryCS-  
2nd

Naoyuki  
Nagatou

再帰  
(Recursion)

再帰への導入

再帰 (Recursion)

再帰的定義

木構造

プログラムと  
しての再帰

引数の有効範囲  
(Scope)

再帰プログラム

Quiz 1

繰り返しと  
再帰

末尾再帰 (Tail  
Recursion)

コンピュータの中では

再帰のまとめ

課題 S

4

再帰 (Recursion)

- 再帰への導入
- 再帰 (Recursion)
- 再帰的定義
- 木構造

5

プログラムとしての再帰

- 引数の有効範囲 (Scope)
- 再帰プログラム

6

Quiz 1

7

繰り返しと再帰

- 末尾再帰 (Tail Recursion)
- コンピュータの中では

8

再帰のまとめ

- 課題 S



# 仮引数と実引数

elementaryCS-  
2nd

Naoyuki  
Nagatou

再帰  
(Recursion)

再帰への導入

再帰 (Recursion)

再帰的定義

本構造

プログラムと  
しての再帰

引数の有効範囲  
(Scope)

再帰プログラム

Quiz 1

繰り返しと  
再帰

末尾再帰 (Tail  
Recursion)

コンピュータの中では

再帰のまとめ

課題 S

- 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

```
1 ### Newton's method
2 def sqrt_iter (guess,n,eps,previous):
3     def is_enough (guess,eps,previous):
4         return(abs(previous-guess)<(2*eps))
5     def improve (guess,n):
6         return((guess+(n/guess))/2.0)
7     if is_enough(guess,eps,previous):
8         return(guess)
9     else:
10        return(sqrt_iter(improve(guess,n),n
11                           ,eps,guess))
12 def sqroot1 (n,eps):
13     return(sqrt_iter(1.0,n,(2*eps),0.0))
```

## Listing 6: newton.py

```
13 def sqroot (n):
14     ### Machine epsilon
15     def eps_m ():
16         epsilon, old, prod = 1.0, 0.0,
17             0.0
18         cnt=0
19         while (prod!=1.0):
20             old = epsilon
21             cnt=cnt+1
22             epsilon=epsilon/2.0
23             prod=epsilon+1.0
24         return(old)
25     return(sqroot1(n,eps_m()))
```

# プログラムとしての再帰

elementaryCS-  
2nd

Naoyuki  
Nagatou

再帰  
(Recursion)

再帰への導入

再帰 (Recursion)

再帰的定義

本構造

プログラムと  
しての再帰

引数の有効範囲  
(Scope)

再帰プログラム

Quiz 1

繰り返しと  
再帰

末尾再帰 (Tail  
Recursion)

コンピュータの中では

再帰のまとめ

課題 S

- 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

```
1 # Factorial
2 def fact (n):
3     if (n == 1):
4         return(1)
5     else:
6         return(n*(fact (n-1)))
```

呼び出し	環境
fact(3)	$n \leftarrow 3$
fact(2)	$n \leftarrow 2$
fact(1)	$n \leftarrow 1$

# 関数の評価と生成プロセス

elementaryCS-  
2nd

Naoyuki  
Nagatou

再帰  
(Recursion)

再帰への導入

再帰 (Recursion)

再帰的定義

本構造

プログラムと  
しての再帰

引数の有効範囲  
(Scope)

再帰プログラム

Quiz 1

繰り返しと  
再帰

末尾再帰 (Tail  
Recursion)

コンピュータの中では

再帰のまとめ

課題 S

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

elementaryCS-  
2nd

Naoyuki  
Nagatou

再帰  
(Recursion)

再帰への導入

再帰 (Recursion)

再帰的定義

木構造

プログラムと  
しての再帰

引数の有効範囲  
(Scope)

再帰プログラム

Quiz 1

繰り返しと  
再帰

末尾再帰 (Tail  
Recursion)

コンピュータの中では

再帰のまとめ

課題 S

4

再帰 (Recursion)

- 再帰への導入
- 再帰 (Recursion)
- 再帰的定義
- 木構造

5

プログラムとしての再帰

- 引数の有効範囲 (Scope)
- 再帰プログラム

6

Quiz 1

7

繰り返しと再帰

- 末尾再帰 (Tail Recursion)
- コンピュータの中では

8

再帰のまとめ

- 課題 S

# QUIZ 1

elementaryCS-  
2nd

Naoyuki  
Nagatou

再帰  
(Recursion)

再帰への導入

再帰 (Recursion)

再帰的定義

本構造

プログラムと  
しての再帰

引数の有効範囲  
(Scope)

再帰プログラム

Quiz 1

繰り返しと  
再帰

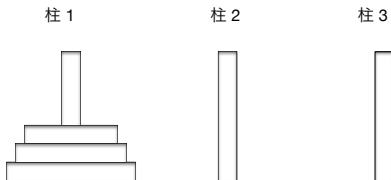
末尾再帰 (Tail  
Recursion)

コンピュータの中では

再帰のまとめ

課題 S

- 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はいけない
- 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

elementaryCS-  
2nd

Naoyuki  
Nagatou

再帰  
(Recursion)

再帰への導入

再帰 (Recursion)

再帰的定義

本構造

プログラムと  
しての再帰

引数の有効範囲  
(Scope)

再帰プログラム

## Quiz 1

繰り返しと  
再帰

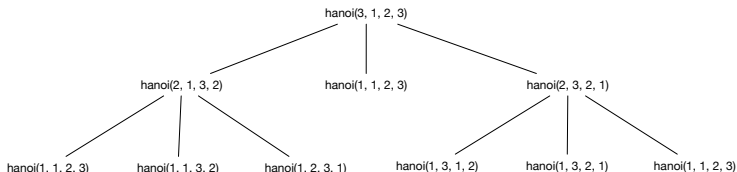
末尾再帰 (Tail  
Recursion)

コンピュータの中では

再帰のまとめ

課題 S

- Basic step: 1 枚の移動
- Recursive step:  $n$  枚から  $n - 1$  枚の移動
- $\text{hanoi}(n, a, b, c)$  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

elementaryCS-  
2nd

Naoyuki  
Nagatou

再帰  
(Recursion)

再帰への導入

再帰 (Recursion)

再帰的定義

木構造

プログラムと  
しての再帰

引数の有効範囲  
(Scope)

再帰プログラム

Quiz 1

繰り返しと  
再帰

末尾再帰 (Tail  
Recursion)

コンピュータの中では

再帰のまとめ

課題 S

4

再帰 (Recursion)

- 再帰への導入
- 再帰 (Recursion)
- 再帰的定義
- 木構造

5

プログラムとしての再帰

- 引数の有効範囲 (Scope)
- 再帰プログラム

6

Quiz 1

7

繰り返しと再帰

- 末尾再帰 (Tail Recursion)
- コンピュータの中では

8

再帰のまとめ

- 課題 S

# 繰り返しプログラムと再帰プログラム

elementaryCS-  
2nd

Naoyuki  
Nagatou

再帰  
(Recursion)

再帰への導入  
再帰 (Recursion)  
再帰的定義  
本構造

プログラムと  
しての再帰

引数の有効範囲  
(Scope)  
再帰プログラム

Quiz 1

繰り返しと  
再帰

末尾再帰 (Tail  
Recursion)

コンピュータの中では

再帰のまとめ

課題 S

- 以前、繰り返しでいろいろな計算を実現しました
- ここでは繰り返しと再帰の関係について見ていきます
- $\text{succ}(y)$  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  $f(g_1(x_1), \dots, g_n(x_n))$
  - 再帰は basic step と recursive step で場合分け



# 末尾再帰呼び出し (Tail Recursive Call)

elementaryCS-  
2nd

Naoyuki  
Nagatou

再帰  
(Recursion)

再帰への導入

再帰 (Recursion)

再帰的定義

本構造

プログラムと  
しての再帰

引数の有効範囲  
(Scope)

再帰プログラム

Quiz 1

繰り返しと  
再帰

末尾再帰 (Tail  
Recursion)

コンピュータの中では

再帰のまとめ

課題 S

- 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  $\text{fact}(n)$  を得るために  $\text{fact}(n-1)$  の後の計算 ( $n \times \square$ ) を覚えてなければならない
- 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: 再帰プロセス版

```
1 # Factorial
2 def fact (n):
3     if (n == 1):
4         return(1)
5     else:
6         return(n*(fact(n-1)))
```

## Listing 9: 繰り返しプロセス版

```
16 # Factorial
17 def fact_iter (n):
18     def fact_iter1 (prod, cnt, max):
19         if cnt > max:
20             return(prod)
21         else:
22             return(fact_iter1(prod*cnt, cnt+1,
23                                 max))
23     return(fact_iter1(1, 1, n))
```

# 末尾呼び出し (Tail Call)

elementaryCS-  
2nd

Naoyuki  
Nagatou

再帰  
(Recursion)

再帰への導入

再帰 (Recursion)

再帰的定義

本構造

プログラムと  
しての再帰

引数の有効範囲  
(Scope)

再帰プログラム

Quiz 1

繰り返しと  
再帰

末尾再帰 (Tail  
Recursion)

コンピュータの中では

再帰のまとめ

課題 S

- 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

## 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

elementaryCS-  
2nd

Naoyuki  
Nagatou

再帰  
(Recursion)

再帰への導入

再帰 (Recursion)

再帰的定義

本構造

プログラムと  
しての再帰

引数の有効範囲  
(Scope)

再帰プログラム

Quiz 1

繰り返しと  
再帰

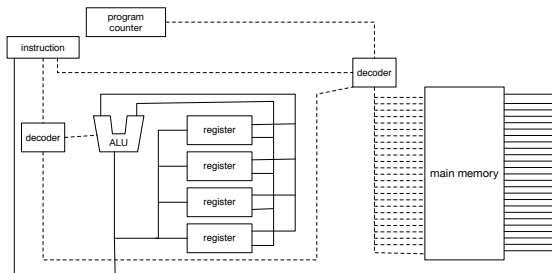
末尾再帰 (Tail  
Recursion)

コンピュータの中では

再帰のまとめ

課題 S

- 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

elementaryCS-  
2nd

Naoyuki  
Nagatou

再帰

(Recursion)

再帰への導入

再帰 (Recursion)

再帰的定義

本構造

プログラムと  
しての再帰

引数の有効範囲  
(Scope)

再帰プログラム

Quiz 1

繰り返しと  
再帰

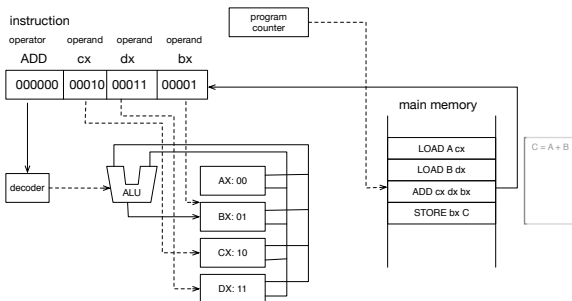
末尾再帰 (Tail  
Recursion)

コンピュータの中では

再帰のまとめ

課題 S

- 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

elementaryCS-2nd

Naoyuki  
Nagatou

再帰  
(Recursion)

再帰への導入  
再帰 (Recursion)  
再帰的定義  
本構造

プログラムと  
しての再帰

引数の有効範囲  
(Scope)  
再帰プログラム

Quiz 1

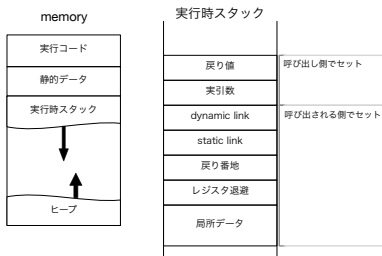
繰り返しと  
再帰

末尾再帰 (Tail  
Recursion)  
コンピュータの中では

再帰のまとめ

課題 S

- 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

elementaryCS-  
2nd

Naoyuki  
Nagatou

再帰  
(Recursion)

再帰への導入

再帰 (Recursion)

再帰的定義

木構造

プログラムと  
しての再帰

引数の有効範囲  
(Scope)

再帰プログラム

Quiz 1

繰り返しと  
再帰

末尾再帰 (Tail  
Recursion)

コンピュータの中では

再帰のまとめ

課題 S

4

再帰 (Recursion)

- 再帰への導入
- 再帰 (Recursion)
- 再帰的定義
- 木構造

5

プログラムとしての再帰

- 引数の有効範囲 (Scope)
- 再帰プログラム

6

Quiz 1

7

繰り返しと再帰

- 末尾再帰 (Tail Recursion)
- コンピュータの中では

8

再帰のまとめ

- 課題 S

# 再帰のまとめ

elementaryCS-  
2nd

Naoyuki  
Nagatou

再帰  
(Recursion)

再帰への導入

再帰 (Recursion)

再帰的定義

本構造

プログラムと  
しての再帰

引数の有効範囲  
(Scope)

再帰プログラム

Quiz 1

繰り返しと  
再帰

末尾再帰 (Tail  
Recursion)

コンピュータの中では

再帰のまとめ

課題 S

- 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

elementaryCS-  
2nd

Naoyuki  
Nagatou

再帰  
(Recursion)

再帰への導入

再帰 (Recursion)

再帰的定義

本構造

プログラムと  
しての再帰

引数の有効範囲  
(Scope)

再帰プログラム

Quiz 1

繰り返しと  
再帰

末尾再帰 (Tail  
Recursion)

コンピュータの中では

再帰のまとめ

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) の間に答えてみてください
- 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

elementaryCS-  
2nd

Naoyuki  
Nagatou

良いアルゴリズム, わるい  
アルゴリズム

アルゴリズムとは  
最大公約数を例に導入  
べき乗のアルゴリズム

時間計算量

Big-O Notation

Sorting

各種ソートアルゴリズム

Search  
Algorithms

計算の複雑さ  
のクラス

## 9 良いアルゴリズム, わるいアルゴリズム

- アルゴリズムとは
- 最大公約数を例に導入
- べき乗のアルゴリズム

## 10 時間計算量

- Big-O Notation

## 11 Sorting

- 各種ソートアルゴリズム

## 12 Search Algorithms

## 13 計算の複雑さのクラス

# Outline

elementaryCS-  
2nd

Naoyuki  
Nagatou

良いアルゴリズム,  
わるいアルゴリズム

アルゴリズムとは  
最大公約数を例に導入  
べき乗のアルゴリズム

時間計算量

Big-O Notation

Sorting

各種ソートアルゴリズム

Search  
Algorithms

計算の複雑さ  
のクラス

## 9 良いアルゴリズム, わるいアルゴリズム

- アルゴリズムとは
- 最大公約数を例に導入
- べき乗のアルゴリズム

## 10 時間計算量

- Big-O Notation

## 11 Sorting

- 各種ソートアルゴリズム

## 12 Search Algorithms

## 13 計算の複雑さのクラス

# アルゴリズム

elementaryCS-  
2nd

Naoyuki  
Nagatou

良いアルゴリズム、わるいアルゴリズム

アルゴリズムとは  
最大公約数を例に導入  
べき乗のアルゴリズム

時間計算量

Big-O Notation

Sorting

各種ソートアルゴリズム

Search

Algorithms

計算の複雑さ  
のクラス

- 前回まで再帰関数についてお話ししました
- 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  $f: D^n \rightarrow D$  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,
  - $D$  は最小（あるいは極小）があつて並んでいるという構造を持つ
  - $x \in D$  として  $f(x) = G(f \upharpoonright x)$
  - $D$  を自然数すれば  $(N; 0, S, +, -, \times, \div, \text{mod}, <)$
  - 計算を  $0, S, +, -, \times, \div, \text{mod}, <$  を再帰や合成で列として構成
- 別の方法として Turing machine を紹介（ホワイトボードに書きます）
- 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

elementaryCS-  
2nd

Naoyuki  
Nagatou

良いアルゴリ  
ズム, わるい  
アルゴリズム

アルゴリズムとは  
最大公約数を例に導入  
べき乗のアルゴリズム

時間計算量

Big-O Notation

Sorting

各種ソートアルゴリ  
ズム

Search  
Algorithms

計算の複雑さ  
のクラス

- ここでは, すでに計算可能な関数のクラスがあつて, 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
- $0, S, +, -, \times, \div, \text{mod}, <$  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

elementaryCS-  
2nd

Naoyuki  
Nagatou

良いアルゴリズム、わるいアルゴリズム

アルゴリズムとは  
最大公約数を例に導入  
べき乗のアルゴリズム

時間計算量

Big-O Notation

Sorting

各種ソートアルゴリズム

Search  
Algorithms

計算の複雑さ  
のクラス

- 関数 GCD をふたつの自然数の公約数のうち最大のもの
- 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

$$\begin{aligned}x_1 &= r_1 x_2 + x_3 \\x_2 &= r_2 x_3 + x_4 \\x_3 &= r_3 x_4 + x_5 \\&\vdots\end{aligned}$$

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

elementaryCS-  
2nd

Naoyuki  
Nagatou

良いアルゴリズム、わるいアルゴリズム

アルゴリズムとは  
最大公約数を例に導入  
べき乗のアルゴリズム

時間計算量

Big-O Notation

Sorting

各種ソートアルゴリズム

Search

Algorithms

計算の複雑さ  
のクラス

- それぞれの実行時間を比較するので見ていてください
  - 適当な数字思いつかないので
  - 10000000 と 10203040 の最大公約数
  - 62979284285501 と 62873258567731 の最大公約数

## Listing 10: gcd.py

```
1 def gcd(x,y):
2     def min(x,y):
3         if (x>y):
4             return y
5         else:
6             return x
7     gcm=1
8     n=min(x,y)
9     for i in range(1,n+1):
10        if (x%i==0) and (y%i==0):
11            gcm=i
12    return (gcm)
```

## Listing 11: gcd.py

```
1 def euclid1(x1,x2):
2     if x2==0:
3         return (x1)
4     else:
5         return (euclid1(x2,x1%x2))
6 def euclid(x1,x2):
7     def swap(x1,x2):
8         return x2,x1
9     if (x1<x2) :
10        x1,x2 = swap(x1,x2)
11    return(euclid1(x1,x2))
```

# 実行時間の違いについての考察

elementaryCS-  
2nd

Naoyuki  
Nagatou

良いアルゴリズム、わるいアルゴリズム

アルゴリズムとは  
最大公約数を例に導入  
べき乗のアルゴリズム

時間計算量

Big-O Notation

Sorting

各種ソートアルゴリズム

Search

Algorithms

計算の複雑さ  
のクラス

- Naive な方法
  - mod や比較の回数を数えてみる
  - $n$  回で  $n$  は  $x$  か  $y$  の小さい方になっている
- ユーグリッドの互除法
  - $n$  より少ない回数で計算できる

euclid(32204,14744)

```
euclid(32204,14744)
=>euclid1(14744,2656)
=>euclid1(2656,1162)
=>euclid1(1162,332)
=>euclid1(332,166)
=>euclid1(166,0)
=>166
```

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

elementaryCS-  
2nd

Naoyuki  
Nagatou

良いアルゴリズム, わるい  
アルゴリズム

アルゴリズムとは  
最大公約数を例に導入  
べき乗のアルゴリズム

時間計算量

Big-O Notation

Sorting

各種ソートアルゴリズム

Search

Algorithms

計算の複雑さ  
のクラス

- 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
- 下の図で  $q_i = 1$  for  $1 \leq i \leq n-1$  としたとき Fibonacci 数  $1, 1, 2, 3, 5, 8, 13, 21, \dots$  を逆にたどったもの

$$r_0 = q_1 r_1 + r_2$$

$$r_1 = q_2 r_2 + r_3$$

$$r_2 = q_3 r_3 + r_4$$

$$\vdots$$

$$r_{n-2} = q_{n-1} r_{n-1} + r_n$$

$$r_{n-1} = q_n r_n$$

where  $r_0 = x_1 > r_1 = x_2$

$$21 = 13 + 8$$

$$13 = 8 + 5$$

$$8 = 5 + 3$$

$$5 = 3 + 2$$

$$3 = 2 + 1$$

$$2 = 2 \cdot 1$$

# ユークリッドの互除法の演算回数

elementaryCS-  
2nd

Naoyuki  
Nagatou

良いアルゴリズム、わるいアルゴリズム

アルゴリズムとは  
最大公約数を例に導入  
べき乗のアルゴリズム

時間計算量

Big-O Notation

Sorting

各種ソートアルゴリズム

Search  
Algorithms

計算の複雑さ  
のクラス

- $r_1 > r_2 > \dots > r_{n-1} > r_n > r_{n+1} = 0$
- 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

$$r_0 = r_1 + r_2$$

$$r_1 = r_2 + r_3$$

$$r_2 = r_3 + r_4$$

$$\vdots$$

$$r_{n-2} = r_{n-1} + r_n$$

$$r_{n-1} = q_n r_n \\ (q_n \geq 2)$$

$$r_n \geq 1 (= f_2)$$

$$r_{n-1} \geq 2 (= f_3)$$

$$r_{n-2} \geq f_3 + f_2 (= f_4)$$

$$\vdots$$

$$r_{n-(l-1)} \geq f_l + f_{l-1}$$

$$\vdots$$

$$r_2 \geq f_{n-1} + f_{n-2} (= f_n)$$

$$r_1 \geq f_n + f_{n-1} (= f_{n+1})$$

# ユークリッドの互除法の演算回数—つづき

elementaryCS-  
2nd

Naoyuki  
Nagatou

良いアルゴリ  
ズム, わるい  
アルゴリズム

アルゴリズムとは  
最大公約数を例に導入  
べき乗のアルゴリズム

時間計算量

Big-O Notation

Sorting

各種ソートアルゴリ  
ズム

Search  
Algorithms

計算の複雑さ  
のクラス

- $r_1 = N$  として, このことから

$$\begin{aligned} N &\geq f_{n+1} \\ N &\geq \phi^{n-1} \end{aligned}$$

(by  $f_{n+1} > \phi^{n-1}$  for  $n > 2$   
where  $\phi = \frac{(1+\sqrt{5})}{2}$ )

$$\begin{aligned} \log_{10} N &\geq (n-1) \log_{10} \phi \\ \log_{10} N &\geq (n-1) \log_{10} \phi > \frac{n-1}{5} \\ n &< 5 \cdot \log_{10} N + 1 \end{aligned}$$

- 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

elementaryCS-  
2nd

Naoyuki  
Nagatou

良いアルゴリズム, わるい  
アルゴリズム

アルゴリズムとは  
最大公約数を例に導入  
べき乗のアルゴリズム

時間計算量

Big-O Notation

Sorting

各種ソートアルゴリズム

Search  
Algorithms

計算の複雑さ  
のクラス

- 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)

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

$$b^2 = b \times b$$

$$b^4 = b^2 \times b^2$$

$$b^8 = b^4 \times b^4$$

# 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

## Listing 12: power.py

```
1 def fast_power(b,n):
2     def square(x):
3         return(x*x)
4     def is_even(n):
5         if (n%2==0):
6             return(True)
7         else:
8             return(False)
9     if (n==1):
10        return(b)
11    else:
12        if (is_even(n)):
13            return(square(fast_power(b,n/2)))
14        else:
15            return((b*fast_power(b,n-1)))
```

$$b^n = (b^{\frac{n}{2}})^2$$

$n$ が偶数のとき

$$b^n = b \times b^{n-1}$$

$n$ が奇数のとき

elementaryCS-  
2nd

Naoyuki  
Nagatou

良いアルゴリ  
ズム, わるい  
アルゴリズム

アルゴリズムとは  
最大公約数を例に導入  
べき乗のアルゴリズム

時間計算量  
Big-O Notation

Sorting  
各種ソートアルゴリ  
ズム

Search  
Algorithms

計算の複雑さ  
のクラス

# Outline

elementaryCS-  
2nd

Naoyuki  
Nagatou

良いアルゴリズム,  
わるいアルゴリズム

アルゴリズムとは  
最大公約数を例に導入  
べき乗のアルゴリズム

時間計算量

Big-O Notation

Sorting

各種ソートアルゴリズム

Search  
Algorithms

計算の複雑さ  
のクラス

## 9 良いアルゴリズム, わるいアルゴリズム

- アルゴリズムとは
- 最大公約数を例に導入
- べき乗のアルゴリズム

## 10 時間計算量

- Big-O Notation

## 11 Sorting

- 各種ソートアルゴリズム

## 12 Search Algorithms

## 13 計算の複雑さのクラス

# 時間計算量

elementaryCS-  
2nd

Naoyuki  
Nagatou

良いアルゴリズム、わるいアルゴリズム

アルゴリズムとは  
最大公約数を例に導入  
べき乗のアルゴリズム

時間計算量

Big-O Notation

Sorting

各種ソートアルゴリズム

Search

Algorithms

計算の複雑さ  
のクラス

- 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演算 (四則演算や比較演算) の実行回数を見積もって比較
- 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) をもとに単純な式で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

elementaryCS-  
2nd

Naoyuki  
Nagatou

良いアルゴリズム,  
わるいアルゴリズム

アルゴリズムとは  
最大公約数を例に導入  
べき乗のアルゴリズム

時間計算量

Big-O Notation

Sorting

各種ソートアルゴリズム

Search  
Algorithms

計算の複雑さ  
のクラス

- $n$  を入力サイズとして、演算回数  $f(n)$  で傾向をつかむ

## Definition

2 つの関数  $f, g: N \rightarrow R^{>0}$  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 $O(n^2)$

elementaryCS-  
2nd

Naoyuki  
Nagatou

良いアルゴリズム,  
わるいアルゴリズム

アルゴリズムとは  
最大公約数を例に導入  
べき乗のアルゴリズム

時間計算量

Big-O Notation

Sorting

各種ソートアルゴリズム

Search  
Algorithms

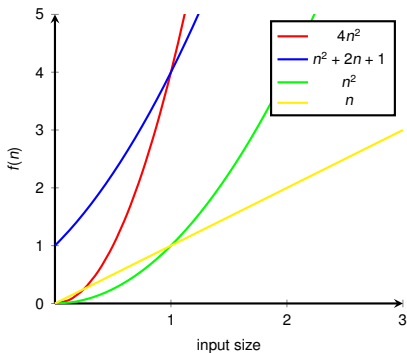
計算の複雑さ  
のクラス

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

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



# ユークリットの互除法とべき乗のアルゴリズムのオーダ

elementaryCS-  
2nd

Naoyuki  
Nagatou

良いアルゴリズム, わるい  
アルゴリズム

アルゴリズムとは  
最大公約数を例に導入  
べき乗のアルゴリズム

時間計算量

Big-O Notation

Sorting

各種ソートアルゴリズム

Search  
Algorithms

計算の複雑さ  
のクラス

## Example (ユークリットの互除法)

- **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

よって,  $\forall n \geq 10 : (5(\log_{10} n) + 1) < (5 + 1) \log_{10} n \Rightarrow O(\log n)$

## Example (べき乗)

- 指数を  $n$ , 2 進表記の 1 の数  $j$ , 偶数のときはいつも半分になるので  $m$  回偶数が出現するとして,

$$\begin{aligned} n &= 2^m \\ \log_2 n &= m \end{aligned}$$

よって,

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

# べき乗 (Power) の演算回数の見積もり

elementaryCS-  
2nd

Naoyuki  
Nagatou

良いアルゴリズム, わるい  
アルゴリズム

アルゴリズムとは  
最大公約数を例に導入  
べき乗のアルゴリズム

時間計算量

Big-O Notation

Sorting

各種ソートアルゴリズム

Search

Algorithms

計算の複雑さ  
のクラス

- 各繰り返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
- $n$  を指数,  $m$  を  $\frac{1}{2}$  する回数として  $\frac{n}{2^m} = 1$
- そのうち, 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
  - 最後 0 乗は計算しないので (1 の数)-1
- $\lfloor \log_2 1000 \rfloor + 5 = 14$  回

## 偶数の出現回数

$$\frac{n}{2^m} = 1$$

$$n = 2^m$$

$$\lfloor \log_2 n \rfloor = m$$

## 奇数の出現回数

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

# Outline

elementaryCS-  
2nd

Naoyuki  
Nagatou

良いアルゴリズム,  
わるいアルゴリズム

アルゴリズムとは  
最大公約数を例に導入  
べき乗のアルゴリズム

時間計算量

Big-O Notation

Sorting

各種ソートアルゴリズム

Search  
Algorithms

計算の複雑さ  
のクラス

## 9 良いアルゴリズム, わるいアルゴリズム

- アルゴリズムとは
- 最大公約数を例に導入
- べき乗のアルゴリズム

## 10 時間計算量

- Big-O Notation

## 11 Sorting

- 各種ソートアルゴリズム

## 12 Search Algorithms

## 13 計算の複雑さのクラス

# Sorting

elementaryCS-  
2nd

Naoyuki  
Nagatou

良いアルゴリズム、わるいアルゴリズム

アルゴリズムとは  
最大公約数を例に導入  
べき乗のアルゴリズム

時間計算量

Big-O Notation

Sorting

各種ソートアルゴリズム

Search  
Algorithms

計算の複雑さ  
のクラス

- **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）に並べたりリストを作ること
- 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してみる

# Bubble Sort

elementaryCS-  
2nd

Naoyuki  
Nagatou

良いアルゴリズム、わるいアルゴリズム

アルゴリズムとは  
最大公約数を例に導入  
べき乗のアルゴリズム

時間計算量

Big-O Notation

Sorting

各種ソートアルゴリズム

Search  
Algorithms

計算の複雑さ  
のクラス

## Listing 13: 擬似コード

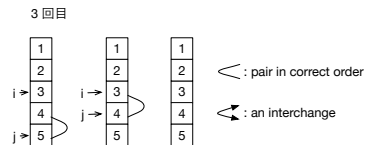
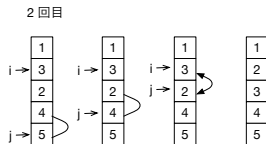
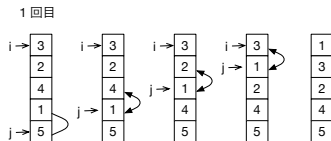
```
1 def bubble_sort(numbers):  
2     for i in {1..n-1}:  
3         for j in {n..i+1}:  
4             if (numbers[j]<numbers[j-1]):  
5                 numbers[j], numbers[j-1] を入替え  
6     return(numbers)
```

### 計算量

- 配列のサイズを  $n$  として
- 4-5 行は入力サイズによらず一定;  $c$
- 3-6 行の演算回数は各繰り返し演算回数の和;  $c$  を  $n-i$  回繰り返すので  $c(n-i)$
- 2-6 行の演算回数は各繰り返しで  $c(n-i)$  を  $(n-1)$  回繰り返すので
- 全体で

$$\sum_{i=1}^{n-1} c(n-i) = \frac{cn(n-1)}{2} = c\left(\frac{n^2}{2} - \frac{n}{2}\right)$$

## 様相



< : pair in correct order

<=> : an interchange

# Insertion Sort

elementaryCS-  
2nd

Naoyuki  
Nagatou

良いアルゴリズム, わるい  
アルゴリズム

アルゴリズムとは  
最大公約数を例に導入  
べき乗のアルゴリズム

時間計算量

Big-O Notation

Sorting

各種ソートアルゴリズム

Search  
Algorithms

計算の複雑さ  
のクラス

様相

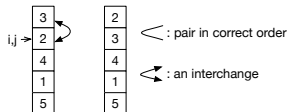
## Listing 14: 擬似コード

```
1 def insertion_sort(numbers):  
2     for i in {2..n}:  
3         j=i  
4         while (numbers[j]<numbers[j-1]):  
5             numbers[j],numbers[j-1]を入れ替え  
6             j=j-1  
7     return(numbers)
```

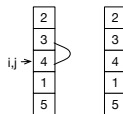
### 計算量 (宿題 2)

- 配列のサイズを  $n$  として  $O(n^2)$  が求められるはず
- 宿題とするので各自で確かめてください

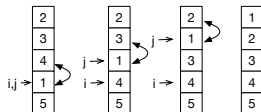
1 回目



2 回目



3 回目



# Quiz 3

elementaryCS-  
2nd

Naoyuki  
Nagatou

良いアルゴリズム、わるいアルゴリズム

アルゴリズムとは  
最大公約数を例に導入  
べき乗のアルゴリズム

時間計算量

Big-O Notation

Sorting

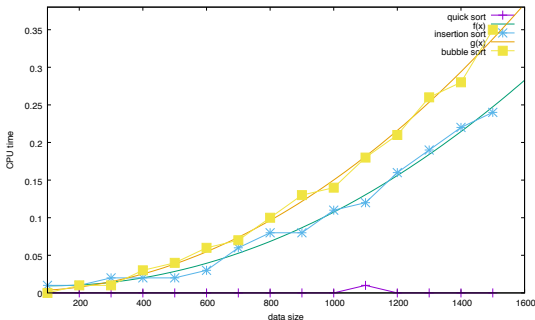
各種ソートアルゴリズム

Search  
Algorithms

計算の複雑さ  
のクラス

## 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 sort-skeleton.py をダウンロード
- quick sort を作成し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 の hint

elementaryCS-  
2nd

Naoyuki  
Nagatou

良いアルゴリズム、わるい  
アルゴリズム

アルゴリズムとは  
最大公約数を例に導入  
べき乗のアルゴリズム

時間計算量

Big-O Notation

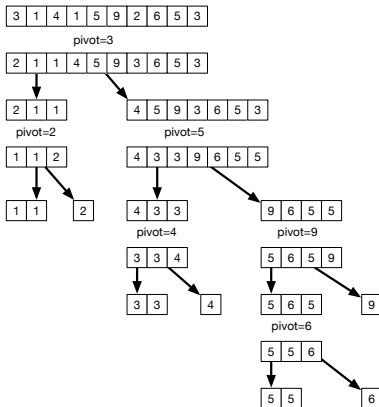
Sorting

各種ソートアルゴリズム

Search  
Algorithms

計算の複雑さ  
のクラス

- 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

elementaryCS-  
2nd

Naoyuki  
Nagatou

良いアルゴリズム、わるいアルゴリズム

アルゴリズムとは  
最大公約数を例に導入  
べき乗のアルゴリズム

時間計算量

Big-O Notation

Sorting

各種ソートアルゴリズム

Search  
Algorithms

計算の複雑さ  
のクラス

- 再帰関数の計算量の計算方法について見ていきます

## Listing 15: 擬似コード

```
1 def partition(numbers, left, right, pivot):
2     while left <= right:
3         numbers[left], numbers[right] を
            入替え
4         while (numbers[left] < pivot):
5             left = left + 1
6         while (numbers[right] >= pivot):
7             right = right - 1
8     return (left)
9 def quick_sort1(numbers, front, rear):
10    pivot = findpivot(numbers, front, rear)
11    if (値がすべて同じ時):
12        return (numbers)
13    else:
14        k = partition(numbers, front, rear,
            numbers[pivot])
15        quick_sort1(numbers, front, k-1)
16        quick_sort1(numbers, k, rear)
17    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

elementaryCS-  
2nd

Naoyuki  
Nagatou

良いアルゴリ  
ズム, わるい  
アルゴリズム

アルゴリズムとは  
最大公約数を例に導入  
べき乗のアルゴリズム

時間計算量

Big-O Notation

Sorting

各種ソートアルゴ  
リズム

Search  
Algorithms

計算の複雑さ  
のクラス

## Theorem

漸化式が  $T(n) = aT(\frac{n}{b}) + cn^d$  の形をしているときのオーダは  
以下のようなになる

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

- $aT(\frac{n}{b})$  は大きさ  $n$  の問題を  $\frac{n}{b}$  の大きさに  $a$  個に分けると  
いう意味
- $cn^d$  は再帰以外の計算量の意味

# Outline

elementaryCS-  
2nd

Naoyuki  
Nagatou

良いアルゴリズム,  
わるいアルゴリズム

アルゴリズムとは  
最大公約数を例に導入  
べき乗のアルゴリズム

時間計算量

Big-O Notation

Sorting

各種ソートアルゴリズム

Search  
Algorithms

計算の複雑さ  
のクラス

## 9 良いアルゴリズム, わるいアルゴリズム

- アルゴリズムとは
- 最大公約数を例に導入
- べき乗のアルゴリズム

## 10 時間計算量

- Big-O Notation

## 11 Sorting

- 各種ソートアルゴリズム

## 12 Search Algorithms

## 13 計算の複雑さのクラス

# 宿題 3

## Search

elementaryCS-  
2nd

Naoyuki  
Nagatou

良いアルゴリズム、わるいアルゴリズム

アルゴリズムとは  
最大公約数を例に導入  
べき乗のアルゴリズム

時間計算量

Big-O Notation

Sorting

各種ソートアルゴリズム

Search  
Algorithms

計算の複雑さ  
のクラス

- 余裕のある人は search algorithms に挑戦してみてください
- Linear search と binary search の計算量を求める
- Linear search と binary search のプログラムを作成し実行時間を比べる
- 参考までに binary search のソースコード (search.py) を <https://sites.google.com/presystems.xyz/elementaryCS/> に置いておきます
- 提出の必要はありません

# Outline

elementaryCS-  
2nd

Naoyuki  
Nagatou

良いアルゴリズム,  
わるいアルゴリズム

アルゴリズムとは  
最大公約数を例に導入  
べき乗のアルゴリズム

時間計算量

Big-O Notation

Sorting

各種ソートアルゴリズム

Search

Algorithms

計算の複雑さ  
のクラス

## 9 良いアルゴリズム, わるいアルゴリズム

- アルゴリズムとは
- 最大公約数を例に導入
- べき乗のアルゴリズム

## 10 時間計算量

- Big-O Notation

## 11 Sorting

- 各種ソートアルゴリズム

## 12 Search Algorithms

## 13 計算の複雑さのクラス

# 計算の複雑さのクラス

elementaryCS-  
2nd

Naoyuki  
Nagatou

良いアルゴリズム、わるいアルゴリズム

アルゴリズムとは  
最大公約数を例に導入  
べき乗のアルゴリズム

時間計算量

Big-O Notation

Sorting

各種ソートアルゴリズム

Search  
Algorithms

計算の複雑さ  
のクラス

- 計算量をオーダー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 (polynomial bound) でこのクラスを P (Polynomial 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  $P \neq NP$  かどうかという問題

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

誤差のおは  
なし

丸め誤差 (roundoff  
error)  
打ち切り誤差  
(truncation error)

## 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

elementaryCS-  
2nd

Naoyuki  
Nagatou

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

誤差のおは  
なし

丸め誤差 (roundoff  
error)

打ち切り誤差  
(truncation error)

14

## 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

15

## 誤差のおはなし

- 丸め誤差 (roundoff error)
- 打ち切り誤差 (truncation error)

# Outline

elementaryCS-  
2nd

Naoyuki  
Nagatou

## 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

## 誤差のおはなし

丸め誤差 (roundoff  
error)  
打ち切り誤差  
(truncation error)

14

## 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

15

## 誤差のおはなし

- 丸め誤差 (roundoff error)
- 打ち切り誤差 (truncation error)

# 非負整数のコンピュータ内での表現

elementaryCS-  
2nd

Naoyuki  
Nagatou

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

誤差のおは  
なし

丸め誤差 (roundoff  
error)

打ち切り誤差  
(truncation error)

## ● 10 進数から 2 進数への変換

### Example (10 進 $\Leftrightarrow$ 2 進)

$$\begin{array}{r} 2) 19 \dots 1 \\ \hline 2) 9 \dots 1 \\ \hline 2) 4 \dots 0 \\ \hline 2) 2 \dots 0 \\ \hline 2) 1 \dots 1 \\ \hline 0 \end{array} \quad \begin{array}{l} \text{Low} \\ \\ \\ \\ \text{High} \end{array}$$

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

# 負の数の表現

elementaryCS-  
2nd

## 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)

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

- Successor (1 足す) でつぎの数になるようになっている
- 最上位ビットがサインビットになっている

# 計算機内の計算

## 整数の減算

elementaryCS-  
2nd

Naoyuki  
Nagatou

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

誤差のおは  
なし

丸め誤差 (roundoff  
error)

打ち切り誤差  
(truncation error)

- 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.

elementaryCS-  
2nd

Naoyuki  
Nagatou

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

誤差のおは  
なし

丸め誤差 (roundoff  
error)

打ち切り誤差  
(truncation error)

- $\bar{b} = (2^n - 1) - b$  より

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

- 引き算は補数を足すことで表す
- $\bar{b} + 1$  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  $3 - 6$

$$\begin{array}{r} 0 \ 1 \ 1 \ 0 \\ + \ 1 \ 1 \ 0 \ 1 \\ \hline 0 \ 0 \ 1 \ 1 \end{array}$$

$$\begin{array}{r} 0 \ 0 \ 1 \ 1 \\ + \ 1 \ 0 \ 1 \ 0 \\ \hline 1 \ 1 \ 0 \ 1 \end{array}$$

# 実数の表現

## 浮動小数 (floating point number)

elementaryCS-  
2nd

Naoyuki  
Nagatou

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

誤差のおは  
なし

丸め誤差 (roundoff  
error)

打ち切り誤差  
(truncation error)

- 浮動小数  $ab^e$ 
  - $a$  は係数 (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 (normalization) 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)

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

# 実数の 2 進表記

elementaryCS-  
2nd

## 実数の表現

- 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)

$$\begin{array}{r} 2)13 \dots 1 \\ \hline 2)6 \dots 0 \\ \hline 2)3 \dots 1 \\ \hline 2)1 \dots 1 \\ \hline 0 \end{array} \begin{array}{l} \text{Low} \\ \\ \\ \text{High} \end{array}$$

$$\begin{array}{r} .6875 \times 2 = 1.375 \text{ High} \\ \underline{.375 \times 2 = 0.75} \\ .75 \times 2 = 1.50 \text{ Low} \\ \underline{.5 \times 2 = 1.00} \end{array}$$

- $13_{(10)}$  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.

elementaryCS-  
2nd

Naoyuki  
Nagatou

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

誤差のおは  
なし

丸め誤差 (roundoff  
error)

打ち切り誤差  
(truncation error)

- 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.

elementaryCS-  
2nd

Naoyuki  
Nagatou

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

誤差のおは  
なし

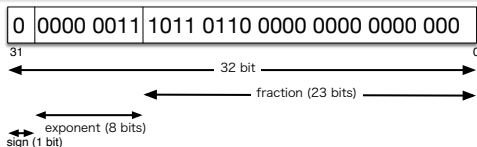
丸め誤差 (roundoff  
error)

打ち切り誤差  
(truncation error)

- 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
- IEEE 754 はもうひと段階

## Example (指数部, 仮数部)

- $1.1011011 \times 2^3$  の符号, 指数, 仮数は以下のとおり
  - 符号 (sign): +
  - 指数 (exponent): 3
  - 仮数 (fraction): 1.1011011



# Quiz: 浮動小数

elementaryCS-  
2nd

Naoyuki  
Nagatou

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

誤差のおは  
なし

丸め誤差 (roundoff  
error)

打ち切り誤差  
(truncation error)

- 実数の浮動小数表現をやってみてください

## 宿題

- 35.75 を浮動小数で表現してみてください

# Quiz: 回答

elementaryCS-  
2nd

Naoyuki  
Nagatou

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

誤差のおは  
なし

丸め誤差 (roundoff  
error)

打ち切り誤差  
(truncation error)

- 2 進へ変換: 100011.11
- 正規化:  $1.00011111 \times 2^5$
- 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 (bias) 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

elementaryCS-  
2nd

Naoyuki  
Nagatou

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

誤差のおは  
なし

丸め誤差 (roundoff  
error)

打ち切り誤差  
(truncation error)

- `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 16: `machine_epsilon.py`

```
1 # Machine epsilon
2 import sys
3
4 epsilon, old, prod = 1.0, 0.0, 0.0
5 cnt = 0
6 while (prod != 1.0):
7     print(epsilon)
8     old = epsilon
9     cnt = cnt + 1
10    epsilon = epsilon / 2.0
11    prod = epsilon + 1.0
12 print("Calculated machine epsilon:", old)
13 print("System information in Python:", sys.float_info.epsilon)
```

# 浮動小数数の算術演算

elementaryCS-  
2nd

Naoyuki  
Nagatou

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

誤差のおは  
なし

丸め誤差 (roundoff  
error)

打ち切り誤差  
(truncation error)

- 正規化した 2 つの浮動小数数  $X, Y$  を  
 $X = F_x \cdot 10^{e_x}, Y = F_y \cdot 10^{e_y}$  とする
- 乗算:  $XY = (F_x \cdot 10^{e_x})(F_y \cdot 10^{e_y}) = F_x F_y \cdot 10^{e_x+e_y}$
- 除算:  $\frac{X}{Y} = \frac{(F_x \cdot 10^{e_x})}{(F_y \cdot 10^{e_y})} = \frac{F_x}{F_y} \cdot 10^{e_x-e_y}$
- 加算・減算:  
 $X \pm Y = (F_x \cdot 10^{e_x}) \pm (F_y \cdot 10^{e_y}) = (F_x \pm F_y \cdot 10^{e_y-e_x}) \cdot 10^{e_x}$ 
  - 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る動作
- 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)

$$\begin{array}{rcl} & 0.2184 & \times 10^2 \\ \times & 0.2512 & \times 10^2 \\ \hline & 0.07998208 & \times 10^4 \\ = & 0.7998208 & \times 10^3 \end{array}$$

$$\begin{array}{rcl} & 0.2844 & \times 10^3 \\ + & 0.4162 & \times 10^1 \\ \hline & 0.288562 & \times 10^3 \end{array}$$

# Outline

elementaryCS-  
2nd

Naoyuki  
Nagatou

## 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

## 誤差のおはなし

丸め誤差 (roundoff  
error)  
打ち切り誤差  
(truncation error)

14

## 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

15

## 誤差のおはなし

- 丸め誤差 (roundoff error)
- 打ち切り誤差 (truncation error)

# 誤差とは

elementaryCS-  
2nd

Naoyuki  
Nagatou

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

誤差のおは  
なし

丸め誤差 (roundoff  
error)

打ち切り誤差  
(truncation error)

- 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)

elementaryCS-  
2nd

Naoyuki  
Nagatou

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

誤差のおは  
なし

丸め誤差 (roundoff  
error)

打ち切り誤差  
(truncation error)

- 表現可能な範囲に丸めることを丸め誤差という
- 演算結果も丸める
- $Z$  を  $X \oplus Y$  の演算結果とする
- $d$  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

$$|f| < 0.5 \text{ のとき } |Z| = |F| \cdot 10^{e_z-d}$$

$$|f| \geq 0.5 \text{ のとき } |Z| = |F| \cdot 10^{e_z-d} + 10^{e_z-d}$$

- 丸め誤差  $\epsilon_z$  とすれば

$$|f| < 0.5 \text{ のとき } |\epsilon_z| = |f| \cdot 10^{e_z-d}$$

$$|f| \geq 0.5 \text{ のとき } |\epsilon_z| = |1 - f| \cdot 10^{e_z-d}$$

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

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

# 打ち切り誤差 (truncation error)

elementaryCS-  
2nd

Naoyuki  
Nagatou

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

誤差のおは  
なし

丸め誤差 (roundoff  
error)

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ち切り誤差という

## 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 (平方の計算)

- ニュートン法
- $\sqrt{a}$  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

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