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

1. 教科書，参考文献
2. 講義概要
3. 評価基準

Outline

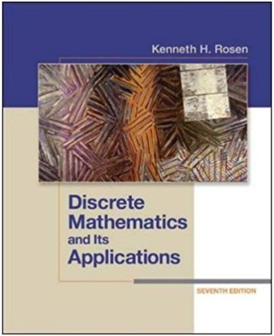
1 教科書，参考文献

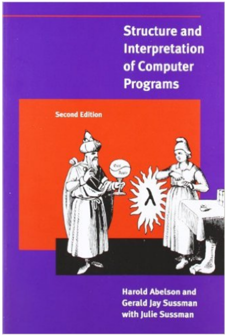
2 講義概要

3 評価基準

参考図書

- 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

1 教科書，参考文献

2 講義概要

3 評価基準

講義概要

- この講義も Zoom で行います
- 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

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
- 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

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
帰納法 (Induction)
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

Part II

再帰

再帰

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
帰納法 (Induction)
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
- 帰納法 (Induction)

6 繰り返しと再帰
- 末尾再帰 (Tail Recursion)

7 コンピュータの中では

8 再帰のまとめ
- 課題 S

Outline

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
- 帰納法 (Induction)

6 繰り返しと再帰
- 末尾再帰 (Tail Recursion)

7 コンピュータの中では

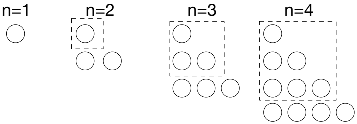
8 再帰のまとめ
- 課題 S

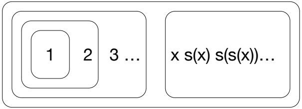
再帰への導入

- 対象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既にわかっている部分を参照して定義するということ

数列の再帰的定義

- 数列 $a_n = \frac{n(n+1)}{2}$, $n \leq 1$
- Baseic step (初項) を 1
- Recursive step (n 項) を $a_n = n + a_{n-1}$ と定義できる
- 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

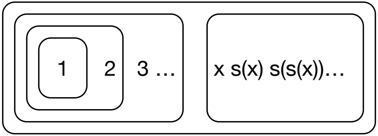
- 関数 f の定義に x より小さい要素についての評価を利用して定義することを原始再帰 (primitive recursion) と呼ぶ
- x より小さい値についての評価はいつも同じ値と仮定
- 定義できた最大の x のつぎの値について定義するというのを繰り返すと全体について定義できる (δ-近似)

Theorem (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
  - $a + \mathrm{succ}(b) = \mathrm{succ}(a + b)$

Basic step:	$a + 0$	if $b = 0$
Recusive step:	$\mathrm{succ}(a + b)$	otherwise

Listing 1: 加算

```
# Recursive definition
import os
import time

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

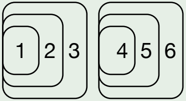
Factorial

- $n! = n \cdot (n-1) \cdot (n-2) \cdots 3 \cdot 2 \cdot 1$
- $n!$ 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: 階乗

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

Example (4!)



フィボナッチ数

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
帰納法 (Induction)
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

- フィボナッチ数

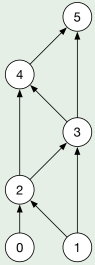
0	1	2	3	4	5···
1	1	2	3	5	8···

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

Example (fib(5))



集合の再帰的定義

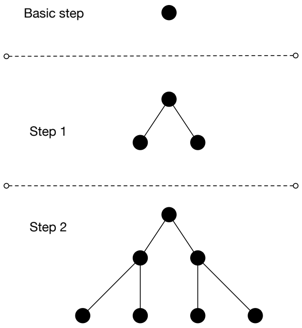
- Basic step: 集合の初期要素を定義
- Recursive step: 既にわかっている要素から新しい要素を定義する規則

Example (文字列の集合 $\Sigma*$)

- 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.: $\{\epsilon,\ a,\ aa,\ aaa, \cdots\}$

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



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

- 自然数上の四則演算を再帰で定義し直してみてください
- フィボナッチ数を求める再帰的な計算を繰り返しに直してみてください
- 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．
  - 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 あるいは len(str) で判定

Outline

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
- 帰納法 (Induction)

6 繰り返しと再帰
- 末尾再帰 (Tail Recursion)

7 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

- 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 法

```python
### 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 法

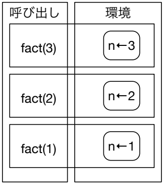
```python
def sqroot (n):
  ### Machine epsilon
  def eps_m ():
    epsilon, old, prod = 1.0, 0.0,
         0.0
    cnt=0
    while (prod!=1.0):
      old = epsilon
      cnt=cnt+1
      epsilon=epsilon/2.0
      prod=epsilon+1.0
    return(old)
  return(sqroot1(n,eps_m()))
```

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
  - n は 3,2,1 と束縛される

Listing 7: 階乗 (引数表示版)

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



関数の評価と生成プロセス

fact(4) の生成プロセス

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

帰納法の原理

- 帰納法は再帰的に定義されたものの性質を証明するテクニック
- 再帰的アルゴリズムの正当性を証明するのにも使える
- 下に帰納法の原理をあげておきます
- $\mathrm{succ}(y)$ は y のつぎの数という意味です (CS 入門第一の +1 に相当)
- 証明したいことがらを ϕ とします
- 原理の前提条件を充たすことを示します
  $\phi(0) \wedge (\phi(n) \Rightarrow \phi(\mathrm{succ}(n)))$

帰納法の原理

集合 $X = \{n : \phi(n)\}$ について，もし $0 \in X$ かつ $\forall y \in X : \mathrm{succ}(y) \in X$ であるなら，X はすべての自然数を含む

Outline

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
- 帰納法 (Induction)

6 繰り返しと再帰
- 末尾再帰 (Tail Recursion)

7 コンピュータの中では

8 再帰のまとめ
- 課題 S

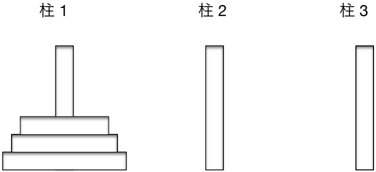
QUIZ 1

elementaryCS-2nd

Naoyuki Nagatou

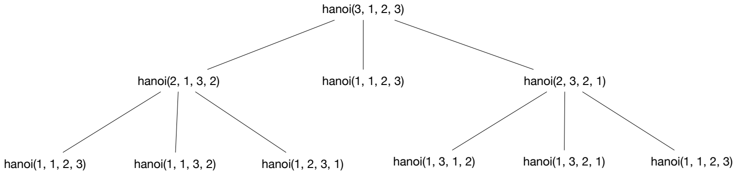
再帰 (Recursion)
再帰への導入
再帰 (Recursion)
再帰的定義
木構造
プログラムとしての再帰
引数の有効範囲 (Scope)
再帰プログラム
帰納法 (Induction)
繰り返しと再帰
末尾再帰 (Tail Recursion)
コンピュータの中では
再帰のまとめ
課題 S

- 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のせてはいけない
- 4b(CS2) クラスのサイト: https://sites.google.com/a/presystems.xyz/sample/home/elementary-computer-science 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



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
帰納法 (Induction)
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

繰り返しと再帰

- 以前，繰り返しでいろいろな計算を実現しました
- 再帰は繰り返しに変換できることがあります
- ここでは繰り返しと再帰の関係について見ていきます

末尾再帰呼び出し (Tail Recursive Call)

- 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

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

Listing 9: 繰り返しプロセス版

```
# 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

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
```

Outline

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
- 帰納法 (Induction)

6 繰り返しと再帰
- 末尾再帰 (Tail Recursion)

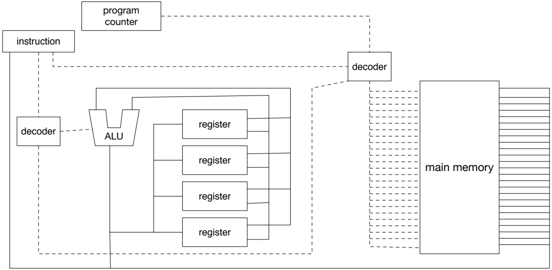
7 コンピュータの中では

8 再帰のまとめ
- 課題 S

CPU とメモリ

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
帰納法 (Induction)
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

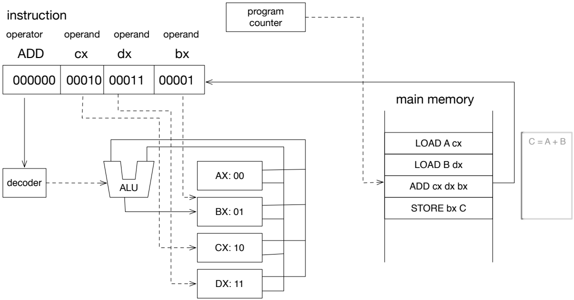
- コンピュータの命令自体も符号化されてます
- CPU (Central Processing Unit) ごとに命令セットも符号も異なっています
- ここでは CPU が命令を実行するサイクルについて見てみます



演算のサイクル

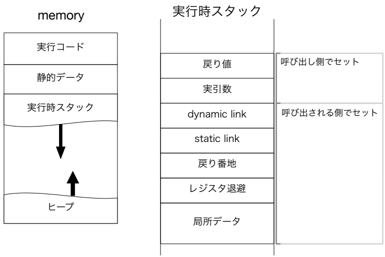
再帰 (Recursion)
再帰への導入
再帰 (Recursion)
再帰的定義
木構造
プログラムとしての再帰
引数の有効範囲 (Scope)
再帰プログラム
帰納法 (Induction)
繰り返しと再帰
末尾再帰 (Tail Recursion)
コンピュータの中では
再帰のまとめ
課題 S

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

- 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

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
- 帰納法 (Induction)

6 繰り返しと再帰
- 末尾再帰 (Tail Recursion)

7 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

- 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/a/presystems.xyz/sample/home/elementary-computer-science 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

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
帰納法 (Induction)

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)
まとめ

Part III

大きい数と小さい数の計算

大きい数と小さい数の計算

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)
まとめ

9 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

10 誤差のおはなし
- 丸め誤差 (roundoff error)
- 打ち切り誤差 (truncation error)

11 まとめ

Outline

9 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

10 誤差のおはなし
- 丸め誤差 (roundoff error)
- 打ち切り誤差 (truncation error)

11 まとめ

繰り返しや再帰によるその他の計算

- 繰り返しや再帰は自然数 n に対応する解を求めるような感じ
  - 数列は n 番目の数を求める: $\alpha : N \to N$
  - ハノイの塔も n 枚目の解を求める
- これを利用して関数の解を求める計算に利用する
- たとえば非線形方程式 $f(x) = 0$ の実根を求める
- 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

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

    で求める
  - x_{k+1} と x_k が十分近くなったら停止

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

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)
まとめ

- 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)

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

- 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
- circulation の実行

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)
まとめ

- 2 進 n 桁の数 a, b $(A_k, B_k \in \{1, 0\})$
  - $a : A_{n-1}A_{n-2} \cdots A_1 A_0$
  - $b : B_{n-1}B_{n-2} \cdots B_1 B_0$
- 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

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

- 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 $3 - 6$

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

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 (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)

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

実数の 2 進表記

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)
まとめ

- 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)

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

- $13_{(10)}$ は $1101_{(2)}$

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

- $.6875_{(10)}$ 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)

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

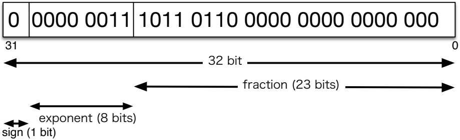
- 符号: +
- 指数: 4
- 仮数: 0.11011011

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
- 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 (fraction): 1.1011011



Quiz: 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

- 35.75 を浮動小数で表現してみてください

Quiz: 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

- 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 10: 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)

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

Outline

9 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

10 誤差のおはなし
- 丸め誤差 (roundoff error)
- 打ち切り誤差 (truncation error)

11 まとめ

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，本来数学の実数で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
- 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

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

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

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

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

Outline

9 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

10 誤差のおはなし
- 丸め誤差 (roundoff error)
- 打ち切り誤差 (truncation error)

11 まとめ

数値計算

- ここで取り上げたおはなしは数値計算（計算機科学の一分野）のなかの計算誤差をとりあげたもの
- シミュレーションなどではある関数の実際の数値を必要とする場合がある
    - 例えば，方程式 $f(x) = 0$ の x を数値的に求める
- 数値計算の手順
    - 最初に適当な 1 次近似 x_0 を選んで，
    - より良い近似を求め，
    - 適当な収束条件を満たすまで繰り返す（マシンイプシロンは収束条件の重要な指標）
- f は複雑なので数値的な解を求めるいろいろな算法を考察

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
ソートアルゴリズムの時間計算量的比較
Search Alogorithms
計算の複雑さのクラス

Part IV

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

12 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

13 時間計算量
- Big–O Notation

14 Sorting
- 各種ソートアルゴリズム
- ソートアルゴリズムの時間計算量的比較

15 Search Alogorithms

16 計算の複雑さのクラス

Outline

12 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

13 時間計算量
- Big–O Notation

14 Sorting
- 各種ソートアルゴリズム
- ソートアルゴリズムの時間計算量的比較

15 Search Alogorithms

16 計算の複雑さのクラス

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，
    - D は最小（あるは極小）があっ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 $(N; 0, S, +, -, \times, \div, \mathrm{mod}, <)$
    - G を $0, S, +, -, \times, \div, \mathrm{mod}, <$ の有限の列として構成
- 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

- 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
- 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の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

ユーグリッドの互除法

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
  - 適当な数字思いつかないので
  - 10,000,000 と 10,203,040 の最大公約数
  - 62979284285501 と 62873258567731 の最大公約数

Listing 11: Naive Algorithm

```
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 12: Euclidean Algorithm

```
def euclid1(x1,x2):
  if x2==0:
    return (x1)
  else:
    return (euclid1(x2,x1%x2))
def euclid(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
  - 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 よりは少ない回数で計算できる

euclid(32204,14744)

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

euclid(34,21)

```
  euculid(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
- 下の図で $q_i = 1$ としたとき Fibonacci 数そのもの

$$
\begin{aligned}
a_1 &= q_1 a_2 + a_3 \\
a_2 &= q_2 a_3 + a_4 \\
a_3 &= q_3 a_4 + a_5 \\
&\cdots \\
a_{k-2} &= q_k a_{k-1}
\end{aligned}
$$

ユークリッドの互除法の演算回数

- 計算に k 回必要ならば，小さい数は k 番目の Fibonacci 数であるかそれより大きい
- この定理から小さい方の数を N として

$$
\begin{aligned}
N &\geq \mathrm{fib}(k) \\
N &\geq \frac{\phi^k}{\sqrt{5}} \ (\text{where } \phi = \frac{(1+\sqrt{5})}{2}) \\
\log_\phi(N) &\geq \log_\phi(\frac{\phi^k}{\sqrt{5}}) \\
\log_\phi(N) + \log_\phi \sqrt{5} &\geq \log_\phi \phi^k \\
\log_5(N) + \log_5 \sqrt{5} &\geq k \\
\log_5(N) + \frac{1}{2} &\geq k
\end{aligned}
$$

- 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

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
ソートアルゴリズムの時間計算量的比較
Search Alogorithms
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
- ふつうの子より 3 回の方が良い子
- power-skeleton.py が https://sites.google.com/a/presystems.xyz/sample/home/elementary-computer-science 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 OCW-i 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

$$b^n = (b^{\frac{n}{2}})^2 \quad n\text{が偶数のとき}$$

$$b^n = b \times b^{n-1} \quad n\text{が奇数のとき}$$

Listing 13: 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==1):
    return(b)
  else:
    if (is_even(n)):
      return(square(fast_power(b,n/2)))
    else:
      return((b*fast_power(b,n-1)))
```

Outline

12 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

13 時間計算量
- Big–O Notation

14 Sorting
- 各種ソートアルゴリズム
- ソートアルゴリズムの時間計算量的比較

15 Search Alogorithms

16 計算の複雑さのクラス

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れば十分

Example

入力サイズ n として一方は $100n^2$，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 $f, g\colon N \to R^{>0}$ とする．

$$\forall n > k\colon f(n) \le c \cdot g(n)$$

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．

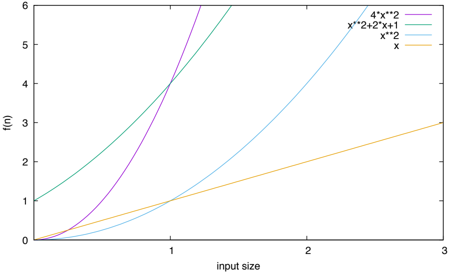
$f(n) = n^2 + 2n + 1$ は $O(n^2)$

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

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

- $c = 4, k = 1, g(n) = n^2$ となる．よって，
- $f(n) = n^2 + 2n + 1 < 4 \cdot n^2$ —$\{f(n) \leq c \cdot g(n)\}$
- $n \geq 2$ 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

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
ソートアルゴリズムの時間計算量的比較

Search Alogorithms

計算の複雑さのクラス

Example (ユーグリットの互除法)

- Lame の定理: 計算に k ステップ必要ならば，小さい数は k 番目の Fibonacci 数であるかそれより大きい

$$
\begin{aligned}
\mathrm{fib}(n) &= \frac{\phi^n}{\sqrt{5}} \\
&= \log_5(n) + \frac{1}{2} \\
&= \frac{1}{\log 5}\cdot \log n + \frac{1}{2} < c \log n \Rightarrow O(\log n)
\end{aligned}
$$

Example (べき乗)

- 指数を n，2 進表記の 1 の数 k，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，
$f(n) = (\log_2 n)+(k-1) = \frac{1}{\log 2}\cdot \log n+(k-1) < c \log n \Rightarrow O(\log n).$

べき乗 (Power) の演算回数の見積もり

- 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 1 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

- 二進表記にした時の桁数

$$\frac{n}{2^m} = 1$$
$$n = 2^m$$
$$\log_2 n = m$$

奇数の出現回数

$$\begin{aligned}
(1000)_{10} = \quad & (1111101000)_2 \\
& \Rightarrow_{\frac{1}{2}} 0111110100 \\
& \Rightarrow_{\frac{1}{2}} 0011111010 \\
& \Rightarrow_{\frac{1}{2}} 0000111110 \\
& \Rightarrow_{\frac{1}{2}} 0000011111 \\
& \ldots \\
& \Rightarrow_{\frac{1}{2}} 0000000011
\end{aligned}$$

Outline

12 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

13 時間計算量
- Big–O Notation

14 Sorting
- 各種ソートアルゴリズム
- ソートアルゴリズムの時間計算量的比較

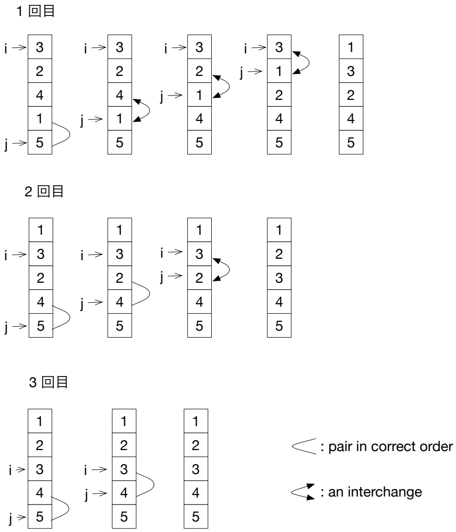
15 Search Alogorithms

16 計算の複雑さのクラス

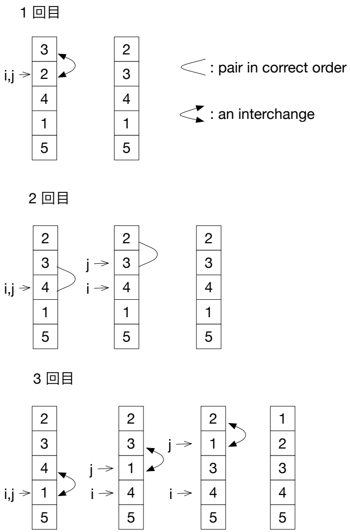
Sorting

- 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) を紹介する

Bubble Sort



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

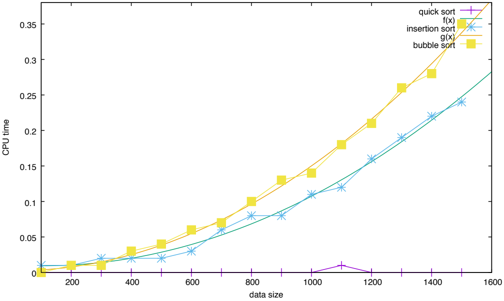
Sorting
各種ソートアルゴリズム
ソートアルゴリズムの時間計算量的比較

Search Alogorithms

計算の複雑さのクラス

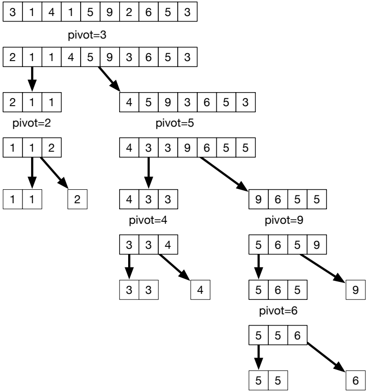
Quiz: Quick Sort

- Input: 任意の長さの任意の整数の列
- Output: 昇順に整列した列
- sort-skeleton.py を加筆して 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
- ソースコードの先頭にコメント行で quick sort のオーダを入れておいてください (option)



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
  2. pivot より小さい値を左側に，大きい値を右側に集める
  3. 左側，右側でそれぞれソートする



ソートアルゴリズムの時間計算量的比較

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
ソートアルゴリズムの時間計算量的比較

Search Alogorithms

計算の複雑さのクラス

Bubble と Insertion Sort

- 入力データのサイズを n として
- 各 i 回目の繰り返しについて $n-i$ だけ比較を繰り返す
- それを n 回繰り返すので全体で $\frac{n(n+1)}{2}$ 回繰り返すことになる
- $\frac{n(n+1)}{2} \leq c \cdot n^2 \Rightarrow O(n^2)$

Quick Sort

- Pivot が中央のとき最適になる
- このとき分割する回数 m として $\frac{n}{2^m} = 1$
- よって $\log_2 n = m$
- j 番目の要素で分割されたとしてそれぞれの分割で j 回と $n-j$ 回比較する
- よって合わせて n 回
- したがって，全体で $n \log n$ 回の比較する
- $n \log n \leq n \log n \Rightarrow O(n \log n)$
- （最悪は $O(n^2)$ 回）

Outline

12 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

13 時間計算量
- Big–O Notation

14 Sorting
- 各種ソートアルゴリズム
- ソートアルゴリズムの時間計算量的比較

15 Search Alogorithms

16 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
ソートアルゴリズムの時間計算量的比較

Search Alogorithms

計算の複雑さのクラス

- 余裕のある人は search algorithms のプログラムに挑戦してみてください
- Linear search と binary search のプログラムを作成し実行時間を比べる

Outline

12 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

13 時間計算量
- Big–O Notation

14 Sorting
- 各種ソートアルゴリズム
- ソートアルゴリズムの時間計算量的比較

15 Search Alogorithms

16 計算の複雑さのクラス

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 $P \neq NP$ かどうかという問題