情報科学(5)

アルゴリズムと計算量

復習: 問題解決とアルゴリズム
（cf.「情報」6.1.1）

- コンピュータによる問題解決ではアルゴリズムが重要になる

  - いきなりプログラムを書くことはまれ

※問題:

  - コンピュータにやらせたいこと全般
  - 計算問題から、長期間のサービスまで

例: 銀行
オンラインシステム

※モデル: 問題を抽象化したもの

アルゴリズムとは

- 9 世紀の数学者 al-Khwarizmi の名に因む
- 問題を解くための手順
  - **有限の手順**で答えを出すもの
  - 曖昧さが(あまり)ない
  - 特定のプログラミング言語に依らない
  - コンピュータに実行させるとは限らない
- 問題の例
  - 「整数 x が素数かどうかを判定する」
  - 「整数 x と y の最大公約数を求める」

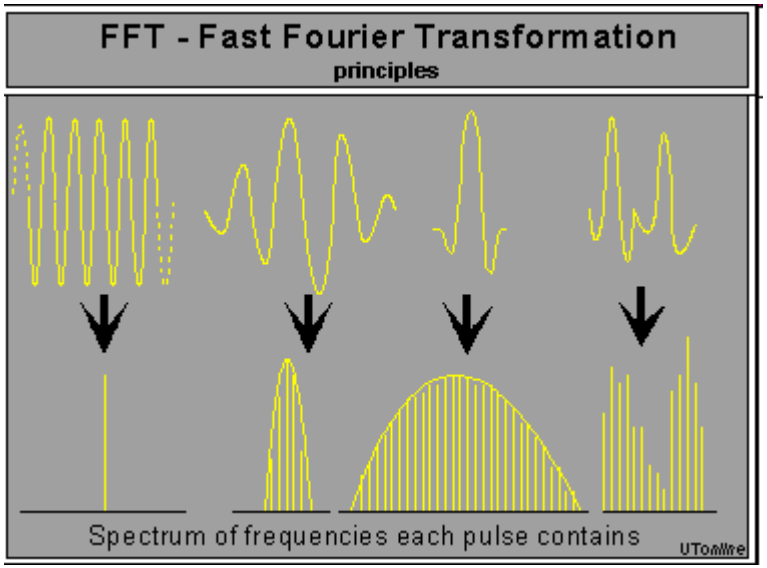
答えが出るかどうか
分からないものは
アルゴリズムとは言わない。
cf. コラッツ予想

アルゴリズムの役割

- 時間やメモリに関する性質を考えることができる
  - 良し悪しが分かる　　→ 計算量 (後述)
- 具体的な問題と独立して考えることができる
  - 問題における細かな点を無視して、解法の重要な点だけを考える
  - 異なる問題も同じアルゴリズムで解ける場合がある
- 具体的なプログラムと独立して考えることができる
  - プログラムにおける細かな点を無視して考えられる
  - 特定のプログラミング言語に依らない

1つの問題に
複数のアルゴリズムがあり得る

- アルゴリズムが違えば計算時間も変わる
- プログラミング・テクニックによる速度の違いよりも大きいことが多い
- 例: 高速フーリエ変換 (Cooley と Tukey が 1965 年に再発明)
  - 周期関数を周波数成分に分解するアルゴリズム
    - 単純な方法では成分数 ***N*** の2乗に比例する時間がかかるところを、このアルゴリズムでは $N \times \log(N)$ に比例する時間で済ませられる
  - 類似アルゴリズムである離散コサイン変換は、音声や映像の圧縮・信号処理などに幅広く使われている



アルゴリズムによる実行時間の違い

- アルゴリズムによる計算時間の違いを実際のプログラミングによって調べる
- 問題とアルゴリズム:
  - 平方根の計算: 数え上げ/二分法/ニュートン法
  - フィボナッチ数: 定義通り/数え上げ/行列
  - 最大公約数: 単純なもの/ユークリッドの互除法
  - 整列: 単純ソート/併合ソート

フィボナッチ数の計算

- 問題: フィボナッチ数列のk番目の数を求める
  - フィボナッチ数列: 1,1,2,3,5,8,13,21,34,…
  - k番目を f_k と書くと $f_k = f_{k-1} + f_{k-2}$ となる数列
    ただし $f_0 = f_1 = 1$
- アルゴリズム
  - 定義通りに計算する方法
  - 0 から順に数え上げる方法
  - 行列を用いる方法

定義通りに計算する フィボナッチ数

- 定義:
  - k番目を f_k と書くとき
  - $f_k = f_{k-1} + f_{k-2}$
  - ただし $f_0 = f_1 = 1$
- f_k を関数 `fibr(k)` にすればよい
- 計算時間は?

```
def fibr(k)
  if k==0 || k==1
    1
  else
    fibr(k-1)+fibr(k-2)
  end
end
```

0 から順に数え上げる
フィボナッチ数の計算

k	0	1	2	3	4	5	6	7	8	9	10	11	12
fib(k): f	1	1	2	3	5	8	13	21	34	55	89	144	233
1 つ前: p1	–	1	1	2	3	5	8	13	21	34	55	89	144
2 つ前: p2	–	–	1	1	2	3	5	8	13	21	34	55	89

0 から順に数え上げる フィボナッチ数の計算

```
def fibl(k)
  f=1
  p1=1
  for i in 2..k
    p2 = p1       #fib(i-2)
    p1 = f        #fib(i-1)
    f = p1 + p2   #fib(i)
  end
  f               #fib(k)
end
```

- 計算時間は?

スピード競争

- 共通資料から、bench.rb と fib.rb をダウンロード。

```
irb(main):004:0> load("./bench.rb")
irb(main):005:0> load ("./fib.rb")
irb(main):006:0> run("fibr", 10)
irb(main):007:0> run ("fibl", 10)

irb(main):009:0> for k in 10..24
irb(main):010:1>   run("fibr", k)
irb(main):011:1>   run("fibl", k)
irb(main):012:1> end
```

グラフの表示方法を変更するためには command という命令を用いる。主な命令には次のものがある。

- command("set logscale y") — グラフの Y 軸を対数スケールにする。y を x にすると X 軸が対数スケールになる。
- command("unset logscale") — 対数スケールをやめる
- command("set xrange [a:b]") — X 軸の範囲を a から b までにする。xrange を yrange にすると Y 軸の範囲を変更する。
- command("set autoscale") — 表示範囲を自動的に変更する。

実際には" "の内側は、グラフの表示に用いている GNUPLOT というソフトウェアに対する命令である。他の命令については GNUPLOT のマニュアルなどを参考にしてほしい。

```
irb(main):026:0> command("set logscale y")
irb(main):027:0> for k in 10..32
irb(main):028:1>   run("fibr", k)
irb(main):029:1> end
```

下 6 桁

```
def fibl6(k)
  f=1
  p1=1
  for i in 2..k
    p2 = p1
    p1 = f
    f = (p1 + p2) % 1000000
  end
  f
end
```

```
irb(main):030:0> reset()
irb(main):031:0> command("unset logscale")
irb(main):032:0> for m in 1..10
irb(main):033:1>   run("fibl6", 100000*m)
irb(main):034:1> end
```

行列を用いるフィボナッチ数の計算法（1）

- (f_{k+1}, f_k)をベクトルだと思うと

$$\begin{pmatrix} f_{k+1} \\ f_k \end{pmatrix} = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix} \begin{pmatrix} f_k \\ f_{k-1} \end{pmatrix}$$

$$= \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^k \begin{pmatrix} f_1 \\ f_0 \end{pmatrix} = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^k \begin{pmatrix} 1 \\ 1 \end{pmatrix}$$

という関係がある

行列 $\begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}$ の固有値は？

1. 0
2. ± 1
3. $\pm\sqrt{2}$
4. $\dfrac{1\pm\sqrt{3}}{2}$
5. $\dfrac{1\pm\sqrt{5}}{2}$

行列を用いるフィボナッチ数の計算法 (2)

$$\begin{pmatrix} f_{k+1} \\ f_k \end{pmatrix} = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^k \begin{pmatrix} 1 \\ 1 \end{pmatrix} = Q^k \begin{pmatrix} 1 \\ 1 \end{pmatrix}$$

- Q^k を高速に計算する方法はあるか?
- $Q^{2k}=(Q^k)^2$ という関係を利用する
  - 例: $Q^{16}=(Q^8)^2=((Q^4)^2)^2=((Q^2)^2)^2)^2$ ── 2乗を4回

$$Q^k = \begin{cases} E & (k=0) \\ (Q^{k/2})^2 & (k \text{ is even}) \\ Q(Q^{k-1}) & (k \text{ is odd}) \end{cases}$$

Q の 8 乗は？

練習 5.5 (行列の冪乗) 行列 A の冪乗 A^n を計算する関数 `matpower(a,n)` を次のようにして定義せよ。行列は 2×2 のものだけを扱うことにして、2 次元配列で表わすことにする。

a) 行列 `a`, `b` の積を求める `matmul(a,b)` を定義せよ。ヒント: 行列を 2×2 に限っているので次のような計算をするだけである。

$$\begin{pmatrix} a_{00} & a_{01} \\ a_{10} & a_{11} \end{pmatrix} \begin{pmatrix} b_{00} & b_{01} \\ b_{10} & b_{11} \end{pmatrix} = \begin{pmatrix} a_{00}b_{00} + a_{01}b_{10} & a_{00}b_{01} + a_{01}b_{11} \\ a_{10}b_{00} + a_{11}b_{10} & a_{10}b_{01} + a_{11}b_{11} \end{pmatrix}$$

b) 行列 `a` の自乗を求める `matsquare(a)` を定義せよ。

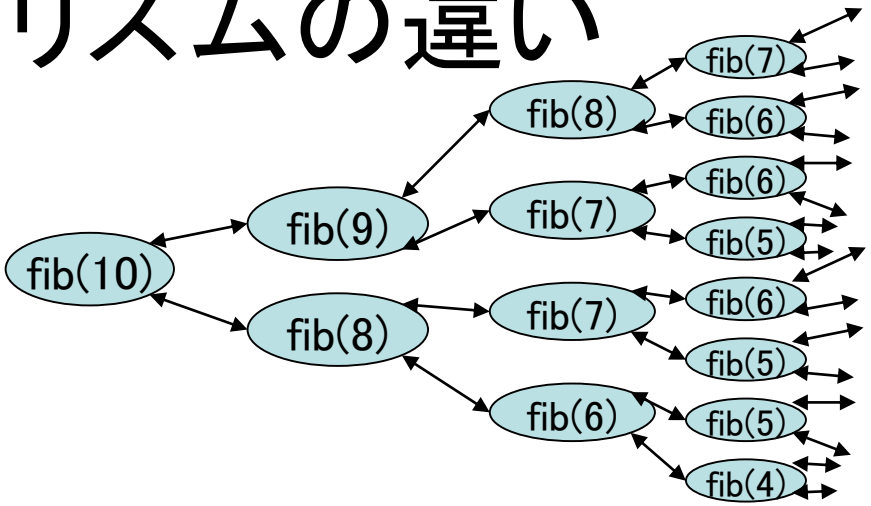
c) 式 5.11 に従って行列 `a` の `n` 乗を求める `matpower(a,n)` を定義せよ。

練習 5.6 (行列を用いた Fibonacci 数の計算) 練習 5.5 で定義した `matpower(a,n)` を用いて $\mathrm{fib}(k)$ を求める関数 `fibm(k)` を定義せよ。また、`fibm(k)` と `fibl(k)` が同じ答を出すことをいくつかの `k` について確認せよ。

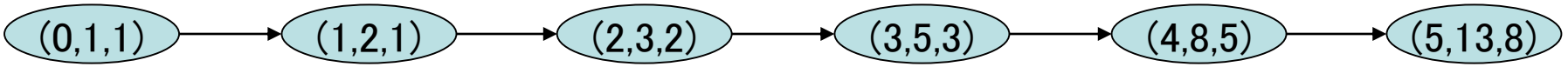
練習 5.7 (行列を用いた Fibonacci 数の計算 (下 6 桁)) 練習 5.6 で定義した関数を変更し、Fibonacci 数の下位 6 桁のみを求める `fibm6` を定義せよ。時間計測用の配布プログラムを用いて、大きな k に対する `fibm6(k)` の計算時間が $\log k$ に比例していることを確認せよ。

フィボナッチ数の計算: アルゴリズムの違い

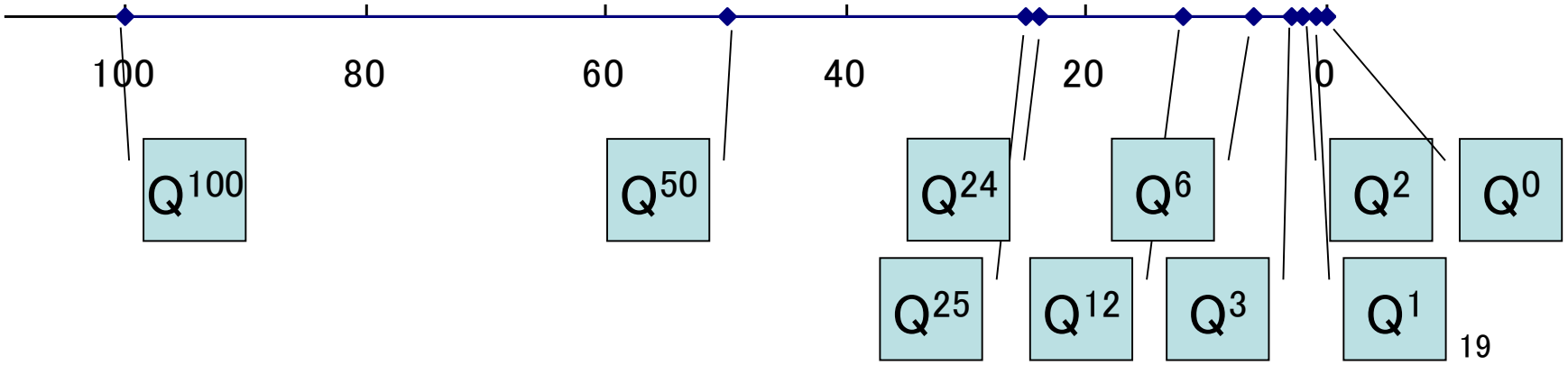
- 定義通り



- 数え上げ

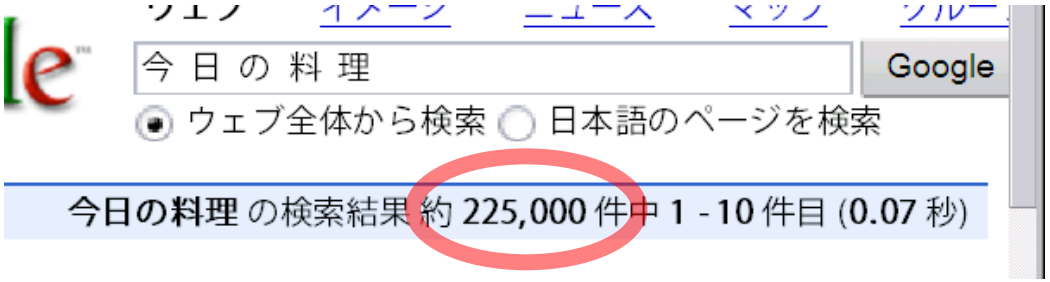


- 行列を使う



整列（sorting）

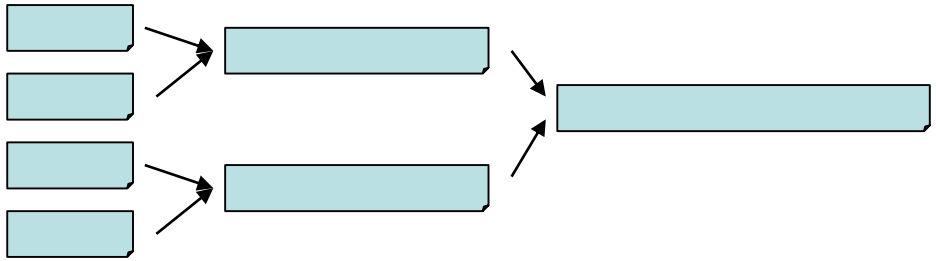
- 問題: n 個のデータを大きさ順に並べる
  (単純化): n 個の整数を小さい順に並べる
  - 色々な場面で使われる
  - データの「下ごしらえ」としても使われる
    例: 検索を高速化するために整列しておく
- データ数は、時として非常に大きくなる
  - 「全学生の氏名を学生証番号順に表示」
  - 「『今日の料理』という言葉を持つページを信頼度順に表示」



整列アルゴリズム

とても沢山ある

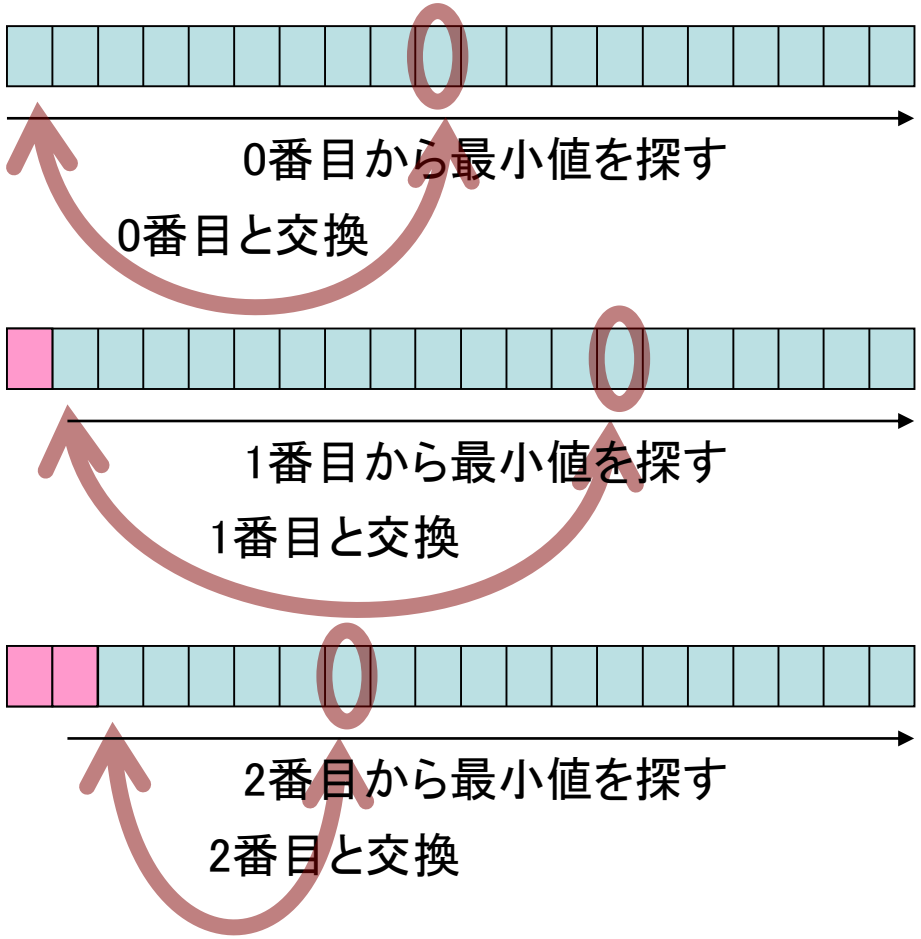
- 単純整列法: 1番小さいものを見つけ、2番目に小さいものを見つけ、…
- 併合整列法:



- ビン整列法
- 基数整列法
- クイック整列法

単純整列法

- 方法:
  - i=0,1,2,…について
    列の中でi番目に
    小さい数を見つけ、
    それをi番目へ移動
  - 「i番目に小さい数」は
    列のi番目以降の
    最小値



単純整列法

```
def simplesort(a)
  for i in 0..(a.length()-1)
    min_value = a[i]
    min_index = i
    for j in (i+1)..(a.length()-1)
      if a[j] < min_value
        min_value = a[j]
        min_index = j
      end
    end
    a[min_index] = a[i]
    a[i] = min_value
  end
  a
end
```

当面の最小値

その添字

より小さい値が見つかった場合

a[i]以降の最小値を見つける

a[i]とa[min_index]を交換

単純整列法

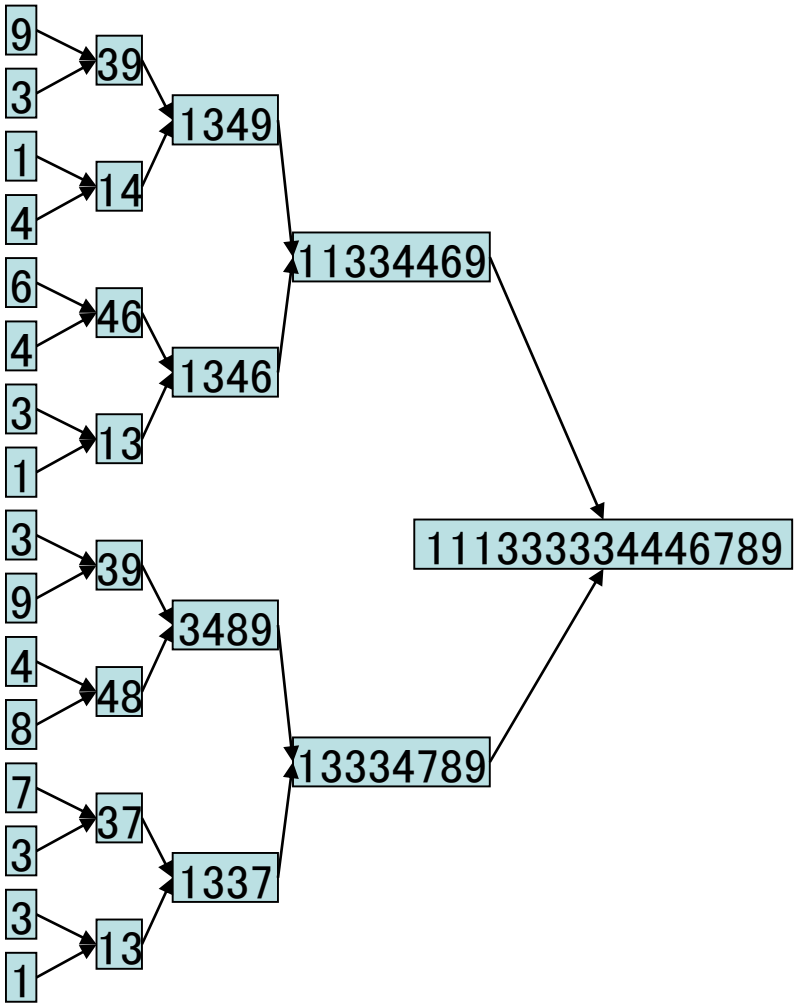
```
def simplesort(a)
  for i in 0..(a.length()-1)
    k = min_index(a,i)
    v = a[k]
    a[k] = a[i]
    a[i] = v
  end
  a
end
```

練習

- 配列 a の（先頭を0 番目としたときの）i 番目以降の値の中で、最小値が出現する番号を答える min_index(a,i) を追加して、単純整列法を完成させよ。

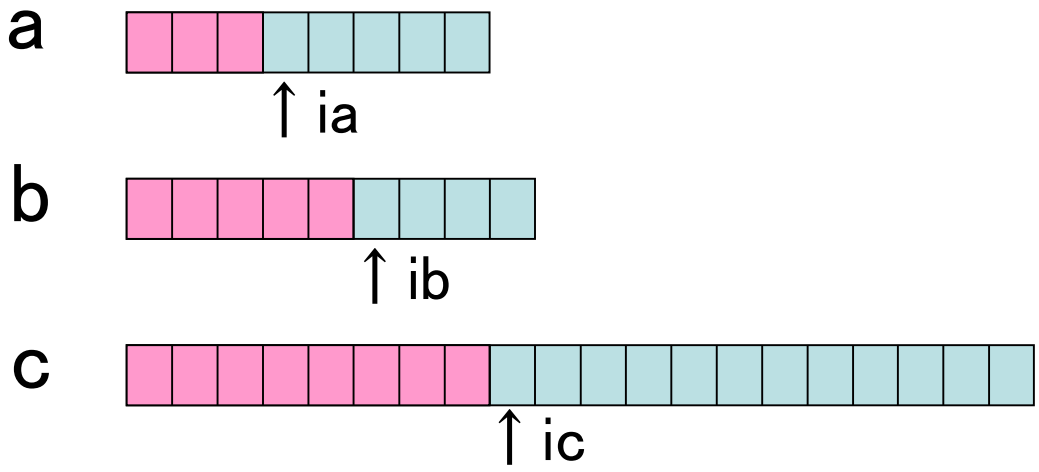
併合整列法

- 前提:
  - 整列済みの列が沢山ある
- 方法:
  - 2つの列を順序よくくっつけて1つにする(併合)
  - 列が1つになるまで繰り返す



併合整列法: 整列済の列の併合

- 方法 (a, bを併合したcを作る):
  - cを作る(長さは(aの長さ)+(bの長さ))
  - 2つの列の先頭を比べ、小さい方をコピーする
  - どちらかの最後に至るまで続ける
  - 残った列をコピーする



2つの列の併合

```
def merge(a,b)
  c = Array.new(a.length() + b.length())
  ia=0
  ib=0
  ic=0
  while ia < a.length() && ib < b.length()
    if a[ia] < b[ib] then
      c[ic] = a[ia]
      ia = ia + 1
    else
      c[ic] = b[ib]
      ib = ib + 1
    end
    ic = ic + 1
  end
  a,b のうちまだ残っている方を全て c に移す(省略)
  c
end
```

併合した配列

a,b,cの「先頭」添字番号

a, b の
先頭から
小さい方を
cに書き写す

練習

- 省略された部分を補って関数 merge を完成させよ。

併合整列法(再帰)

```
def mergesort(a)
  submergesort(a,0,a.length())
end

def submergesort(a,i,n)
  if n<2
    if n==0
      [ ]
    else
      [ (a[i]) ]
    end
  else
    merge(submergesort(a,i,n/2),submergesort(a,i+n/2,n-n/2))
  end
end
```

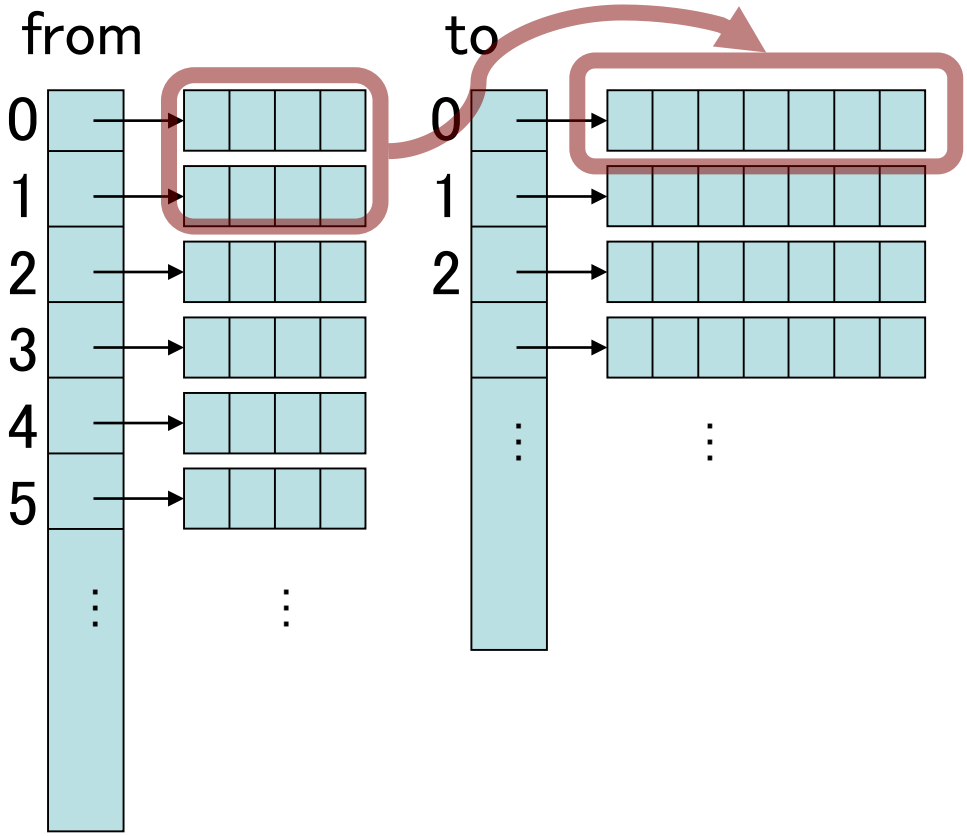
配列 a の i 番目から
n 個を整列した結果を
配列として返す

長さ 0 の配列

a[i] という一つの要素から成る配列

併合整列法: 併合を繰り返す

- 前提:
  - 併合される列は配列の配列fromにしまわれている
- 方法:
  - fromが長さ1の配列になるまで繰り返す
    - fromの半分の長さの配列toを作る
    - fromの(2i)番目と(2i+1)番目を併合してtoのi番目にしまう
    - fromをtoにする



併合整列法: 併合を繰り返す

```
def mergesort(a)
  n = a.length()
  from = Array.new(n)
  for i in 0..(n-1)
    from[i] = [ (a[i]) ]
  end
  while n > 1
    to = Array.new((n+1)/2)
    for i in 0..(n/2-1)
      to[i] = merge(from[i*2],from[i*2+1])
      if !is_even(n)
        to[(n+1)/2-1] = from[n-1]
      end
    end
    from = to
    n = (n+1)/2
  end
  from[0]
end
```

各要素が大きさ1の配列である配列fromを作る

半分の長さの配列toを作る

fromの2要素を併合してtoにコピー

fromが奇数個のときは、最後の要素は併合せずにコピーする

toをfromにする

fromが1要素になるまで続ける

最後はfromが1要素になっておりその中身が整列の結果

while 文は何回まわる？

配列 [3,1,4,1,5,9,2,6,5] に対して

1. 1
2. 2
3. 3
4. 4
5. 5

課題: アルゴリズムによる速度差の実測

- 平方根・整列の複数のアルゴリズムに対応したプログラムを作り、速度の違いを調べる
  - 平方根の場合: x/δの大きさによって時間がどう変化するかをグラフを描いてみる。1秒間に何回計算できるかで比べよ。
  - 整列の場合: ランダムな列を作り、それを整列させてみよ。列の長さで時間がどう変化するかをグラフを描いてみる。
- グラフから、実行時間を予測する式を推定せよ

```
load("./randoms.rb ")    # randoms(id,size,max)
load"./bench.rb")        # run(function_name, x, v)
load("./simplesort.rb") # simplesort(a)
load("./mergesort.rb")  # mergesort(a)

def compare_sort(max, step)
  for i in 1..(max/step)
    x=i*step
    a=randoms(i,x,1)
    run("simplesort", x, a)
    a=randoms(i,x,1)
    run("mergesort", x, a)
  end
end
```

計算量

- これまで見てきたように、アルゴリズムによってプログラムの実行時間は大きく変わり得る
- プログラムを作らずに、良いアルゴリズムを選ぶにはどうすればよいか?

→**計算量**

- アルゴリズムが答えを出すのに必要とする資源の見積り
- 資源 = 計算回数や記憶容量
- 通常はおおまかな式(オーダー, O)で見積り比較する

  ←それでも充分

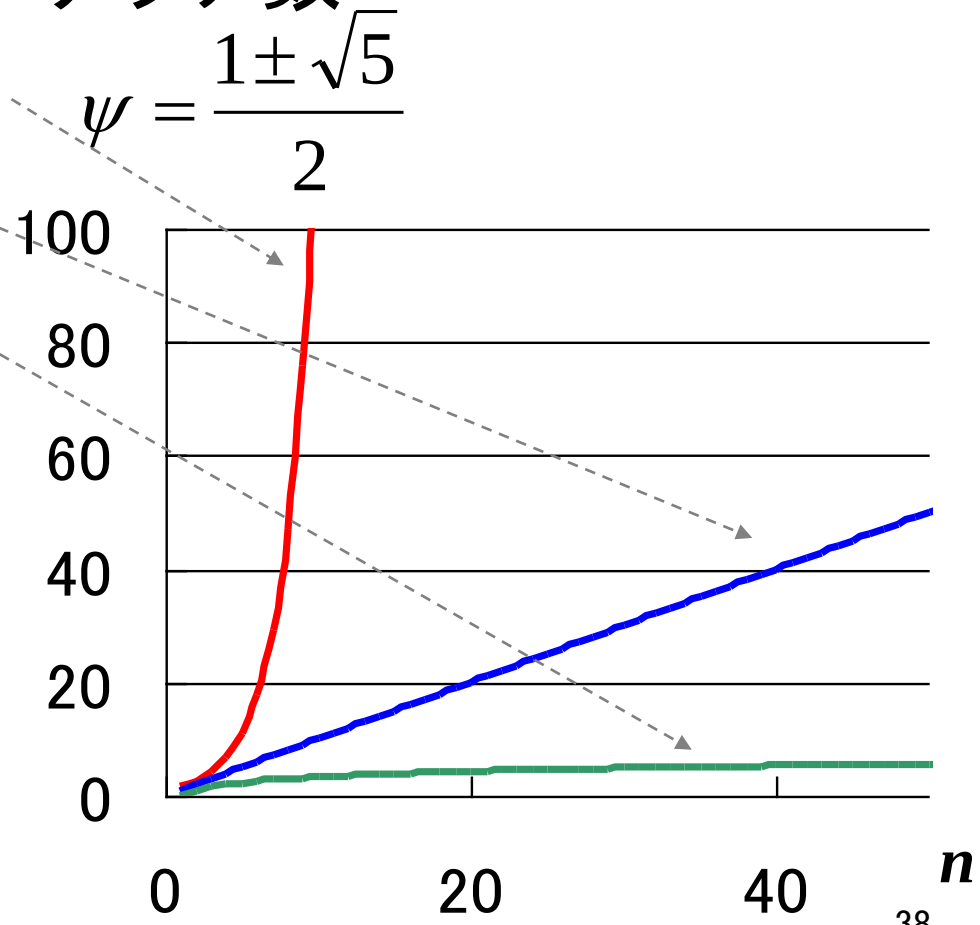
計算量の求め方

1. 各演算、関数呼び出し、判断、分岐をそれぞれ「1回の計算」とする
2. 入力の引数 x に対する計算回数の式を求める
3. 定数を消し、主要な項だけを残す（計算量のオーダー）

※3を前提にすると1, 2は厳密でなくともよくなる

計算量を求める: フィボナッチ数

- 問題: ***n*番目のフィボナッチ数**
  - 定義通り — **O(ψ^n)**
  - 数え上げ — **O(*n*)**
  - 行列 — **O(log *n*)**



計算量を求める: 整列アルゴリズム

問題: k以下の整数の長さnの列を整列

- 単純整列法 — $O(n^2)$
- 併合整列法 — $O(n \log n)$

n	10	100	1,000	10,000	100,000	1.0E+06	1.0E+07	1.0E+08	1.0E+09
n^2	100	10,000	1.0E+06	1.0E+08	1.0E+10	1.0E+12	1.0E+14	1.0E+16	1.0E+18
$n \log n$	33	664	9,966	132,877	1.7E+06	2.0E+07	2.3E+08	2.7E+09	3.0E+10

(k = 100の場合)

課題: 計算量と実測値の対応

- 紹介したアルゴリズムの計算量のオーダーを求めてみよ
- プログラムの実行時間の実測値と対応がとれているかを確認せよ
- 対応がとれない場合は、理由を考えよ