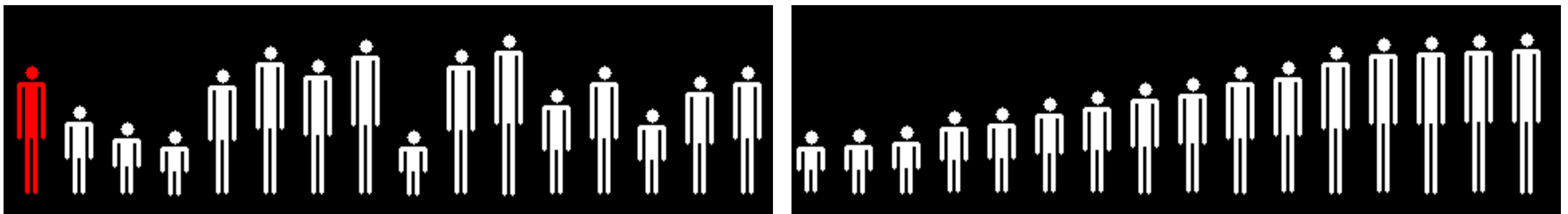
第II部 プログラミングを通して学ぶ情報科学の諸概念

第5章 アルゴリズムと計算量

- Fibonacci数を求めるアルゴリズム
- 計算時間の違いと計算量
- 行列を用いたFibonacci数のアルゴリズム
- 整列のアルゴリズム



復習

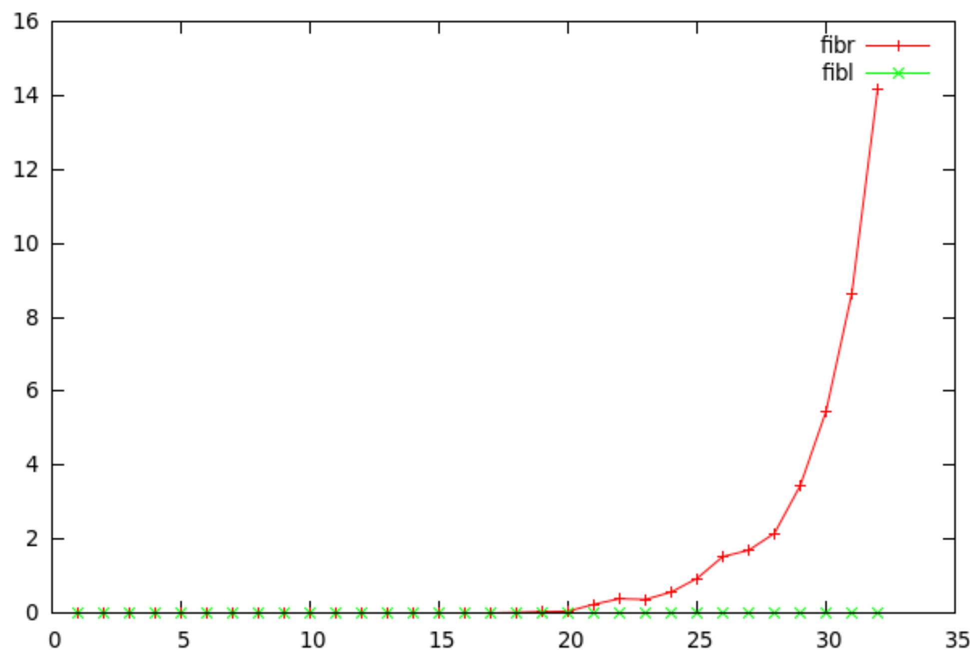
Fibonacci数の計算: アルゴリズムの違い

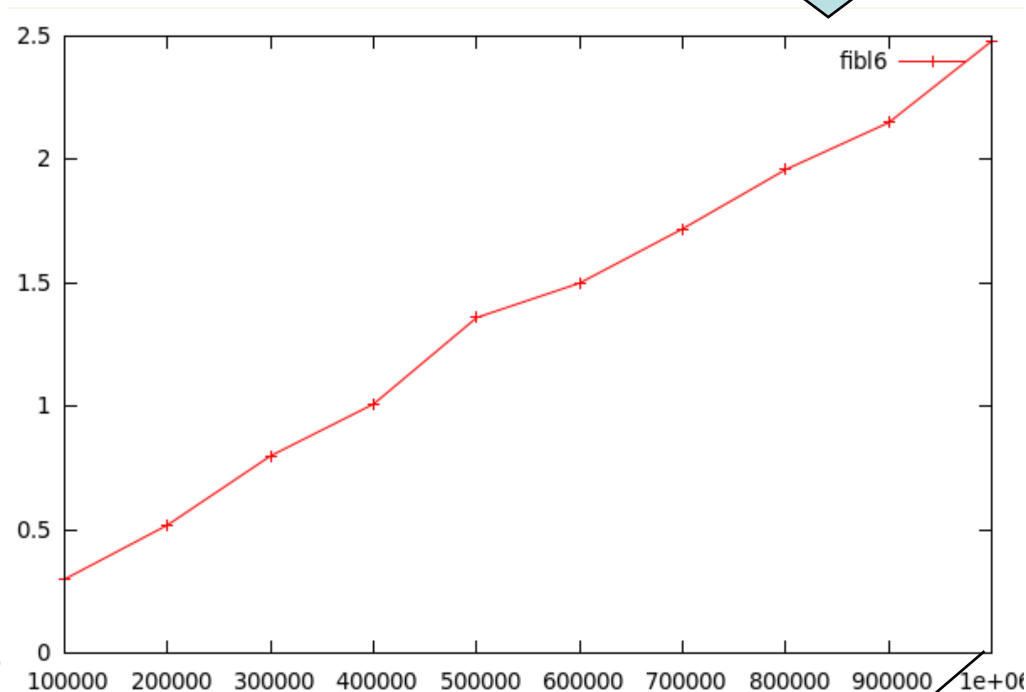
- 定義通り:O(🚒^{k})
- 数え上げ: O(n)
- 行列を使う: O(log n)

計算量と計算時間の対応
Fibonacci数

再帰　O(🚒)

数え上げ　$O(k)$





行列　$O(\log k)$ → ?

10^6

行列版Fibonacci数fibm(k)の計算時間

練習5.9で定義したfimb6(k)

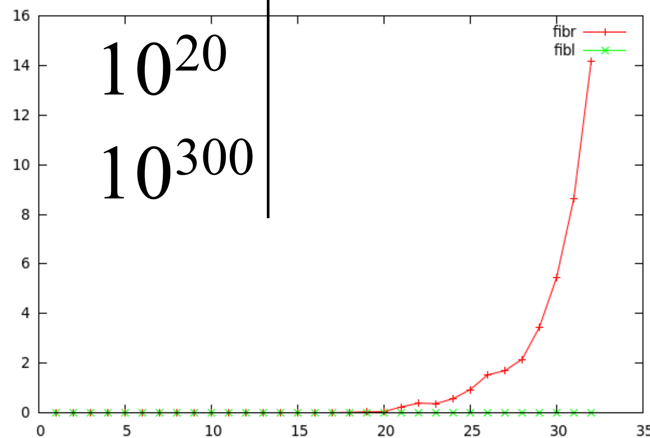
```
irb(main):001:0> load("./bench.rb")
irb(main):002:0> load("./fibm.rb")
irb(main):003:0> command("set logscale x")
irb(main):004:0> for e in 1..30
irb(main):005:0>     run("fibm6", 10**(10*e))
irb(main):006:0> end
```

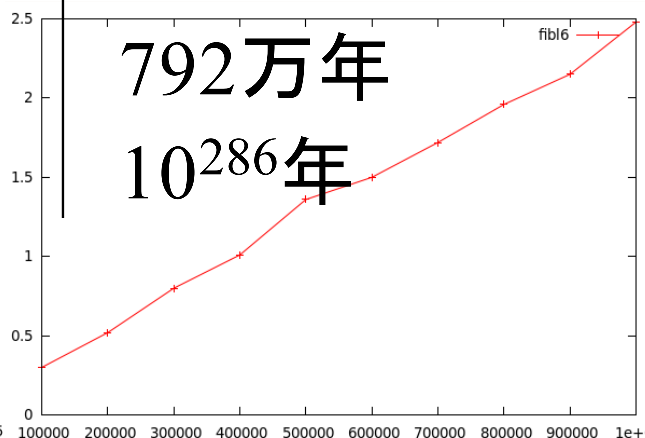
X軸を対数スケールに

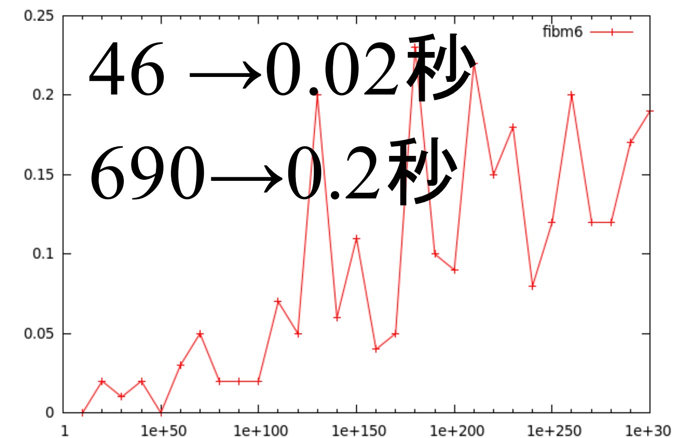
fibm6(10^{10}), fibm6(10^{20}), fibm6(10^{30}), …, fibm6(10^{300}) を測定

Fibonacci数の計算: アルゴリズムの違い

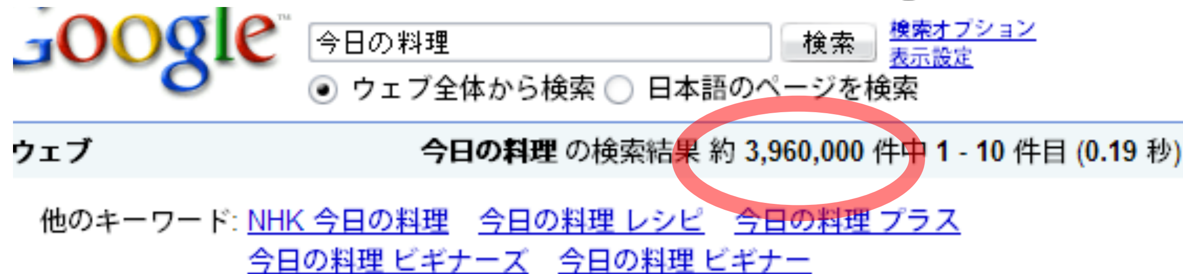
k	再帰 O(🚒^{k})	数え上げ O(k)	行列: O(log k)
32	490万 →14秒		
100	10^{21} →7千万年		
十万		0.25秒	11 →一瞬
百万		2.5秒	13 →一瞬
百億		7時間	23 →一瞬
10^{20}		792万年	46 →0.02秒
10^{300}		10^{286}年	690→0.2秒







整列 (sorting)

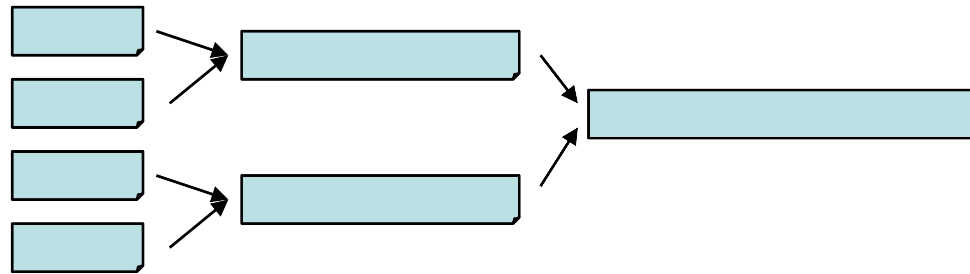


- 問題: n個のデータを大きさ順に並べる
  (単純化): n個の整数を小さい順に並べる
  - 色々な場面で使われる
  - データの「下ごしらえ」としても使われる
    例: 検索を高速化するために整列しておく
- データ数は、時として非常に大きくなる

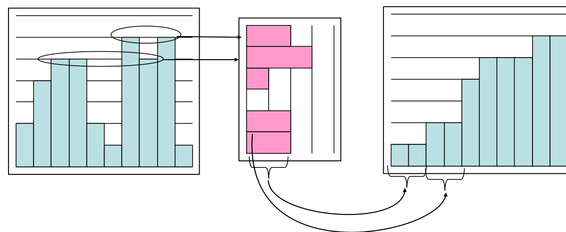
整列アルゴリズム

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:



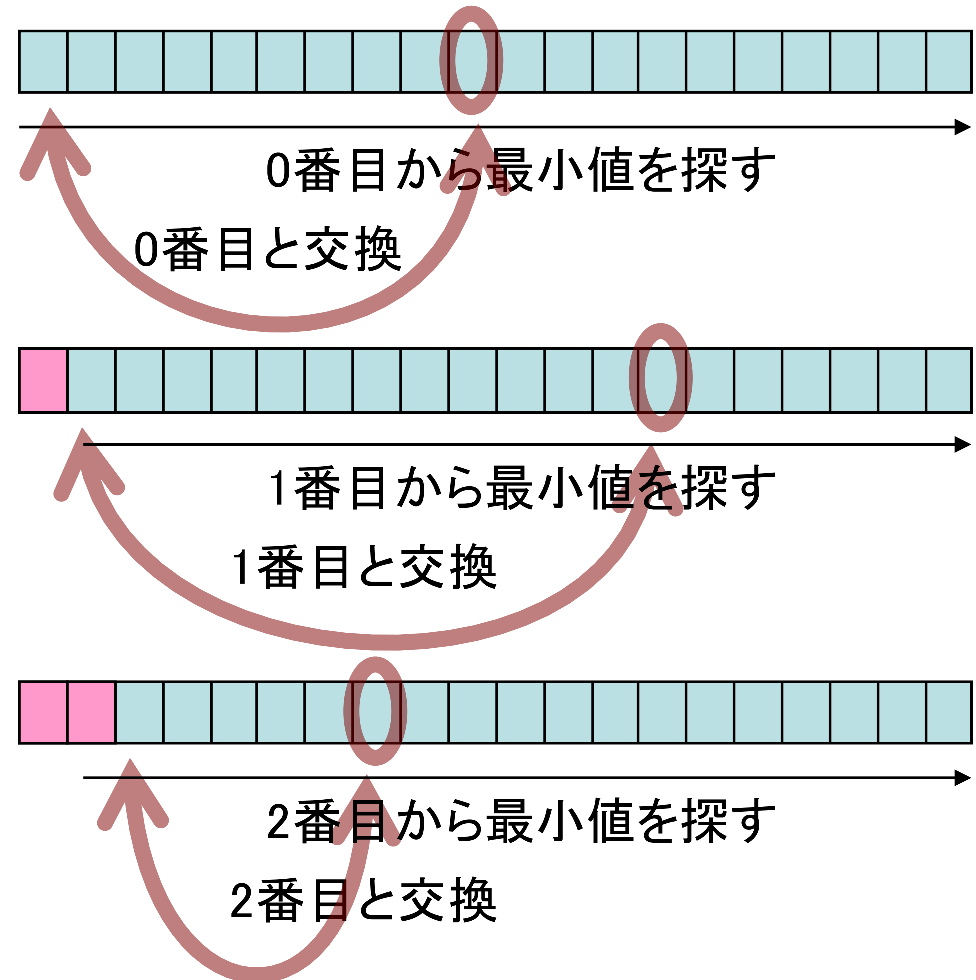
- クイック整列法: (名前だけ)

単純整列法

- ルール（配列の操作）
  - 一度に表にすることができるのは2枚まで
  - 一度に2枚のカードの場所を交換できる

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)
    k = min_index(a,i)
    v = a[i]
    a[i] = a[k]
    a[k] = v
  end
  a
end
```

a[i]以降の最小値の番号を見つける(練習)

a[i]とa[k]を交換
練習4.13をやっていればswap(a,i,k)でよい

整列された配列を答える

simplesort.rb

単純整列法: 最小値の発見

a[i]以降の最小値の番号を見つける(練習)

```
def min_index(a,i)
  暫定最小値の番号kをiにする
  jを(i+1)..(a.length()-1)まで変化させ
    a[j]がa[k]より小さかったら
      kをjにする
  繰り返しが終わった後のkが答え
end
```

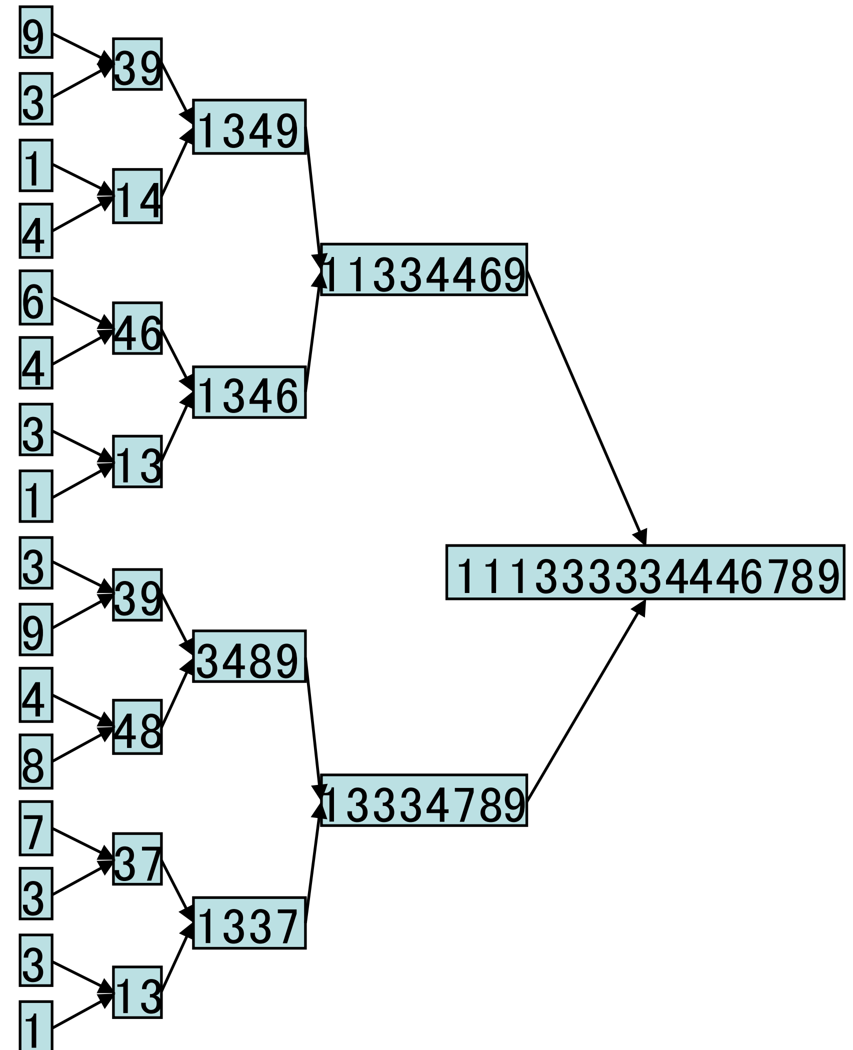
simplesort.rb

1. 完成したらrandoms.rb(配布プログラム)をloadして
   a=randoms(0,100,100)
   simplesort(a) を実行してみよう
2. 下線部を1000, 10000に変えたときの時間は?

併合整列法

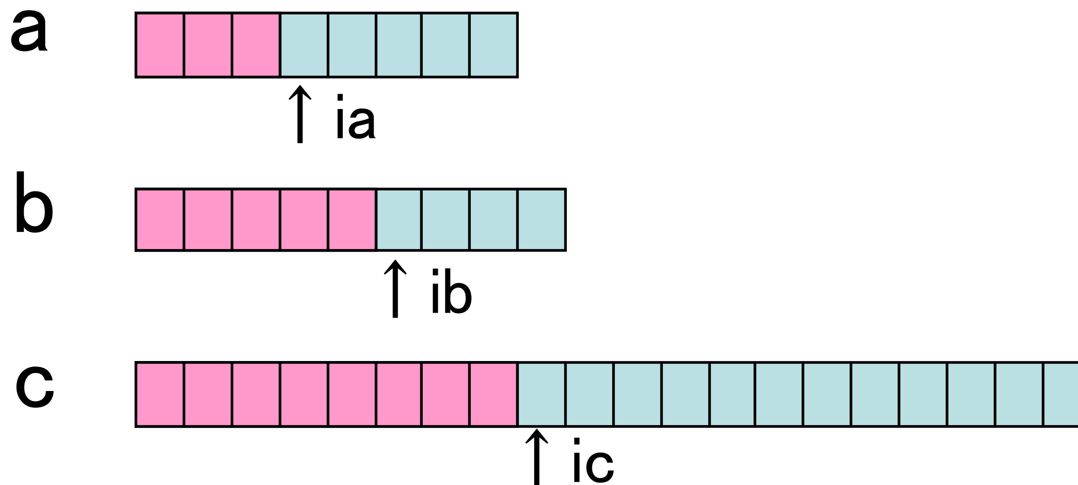
併合整列法

- 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 = make1d(a.length()+b.length())
  ia=0
  ib=0
  ic=0
  while ia < a.length() && ib < b.length()
    if a[ia] < b[ib]
      c[ic] = a[ia]
      ia = ia + 1
      ic = ic + 1
    else
      c[ic] = b[ib]
      ib = ib + 1
      ic = ic + 1
    end
  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

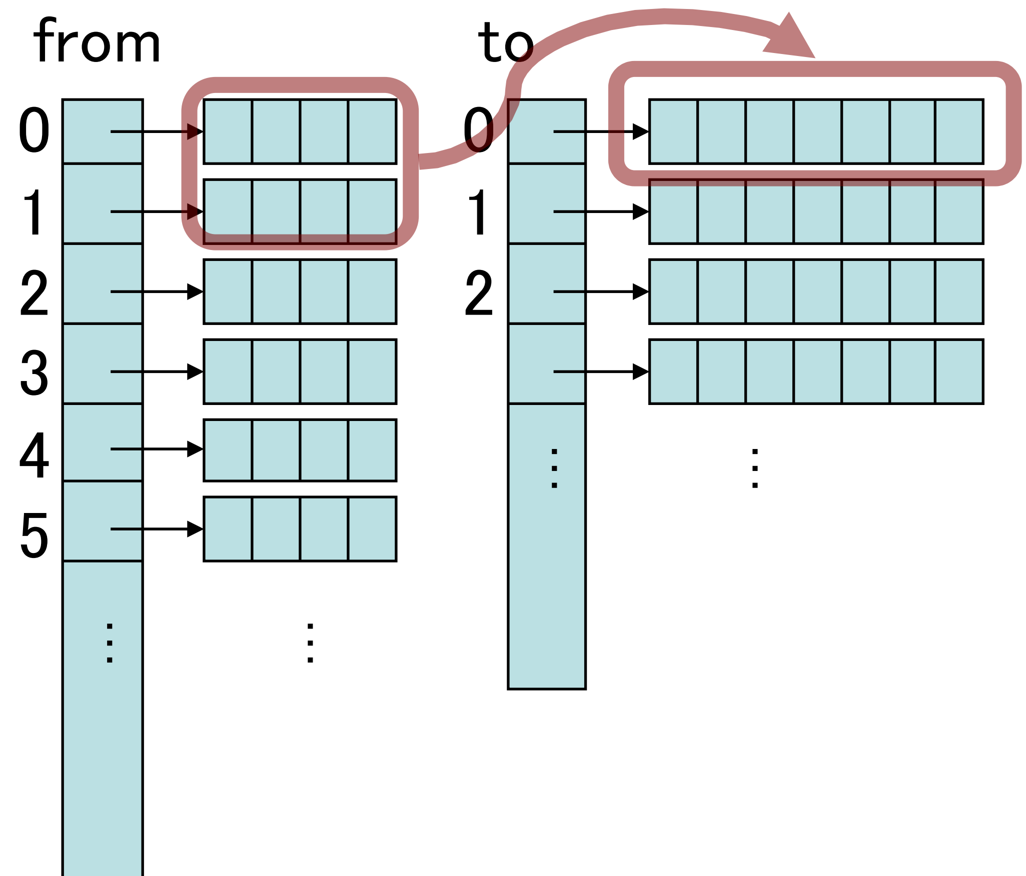
a, b の
先頭から
小さい方を
cに書き写す

試してみよう
merge([1,3,4,7], [2,6,7,8,9])

mergesort.rb

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)

mergeを完成させて
a=randoms(0,100,100)
mergesort(a) を実行してみよう

```
def mergesort(a)
  n = a.length()
  from = make2d(n,1)
  for i in 0..(n-1)
    from[i][0] = a[i]
  end
  while n > 1
    to = make1d((n+1)/2)
    for i in 0..(n/2-1)
      to[i] = merge(from[i*2], from[i*2+1])
    end
    if !is_even(n)
      to[(n+1)/2-1]=from[n-1]
    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

nが1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

mergesort.rb

計算量: 単純整列法

- aの大きさをnとする
- 計算の構造:

  for i in 0..n−1

    for j in (i+1)..(n−1)

      比較

- $$\sum_{i=0}^{n-1}(n-i) \quad = \quad \frac{1}{2}n(n-1)$$

  = $O(n^2)$

  ➢ データ量が10倍

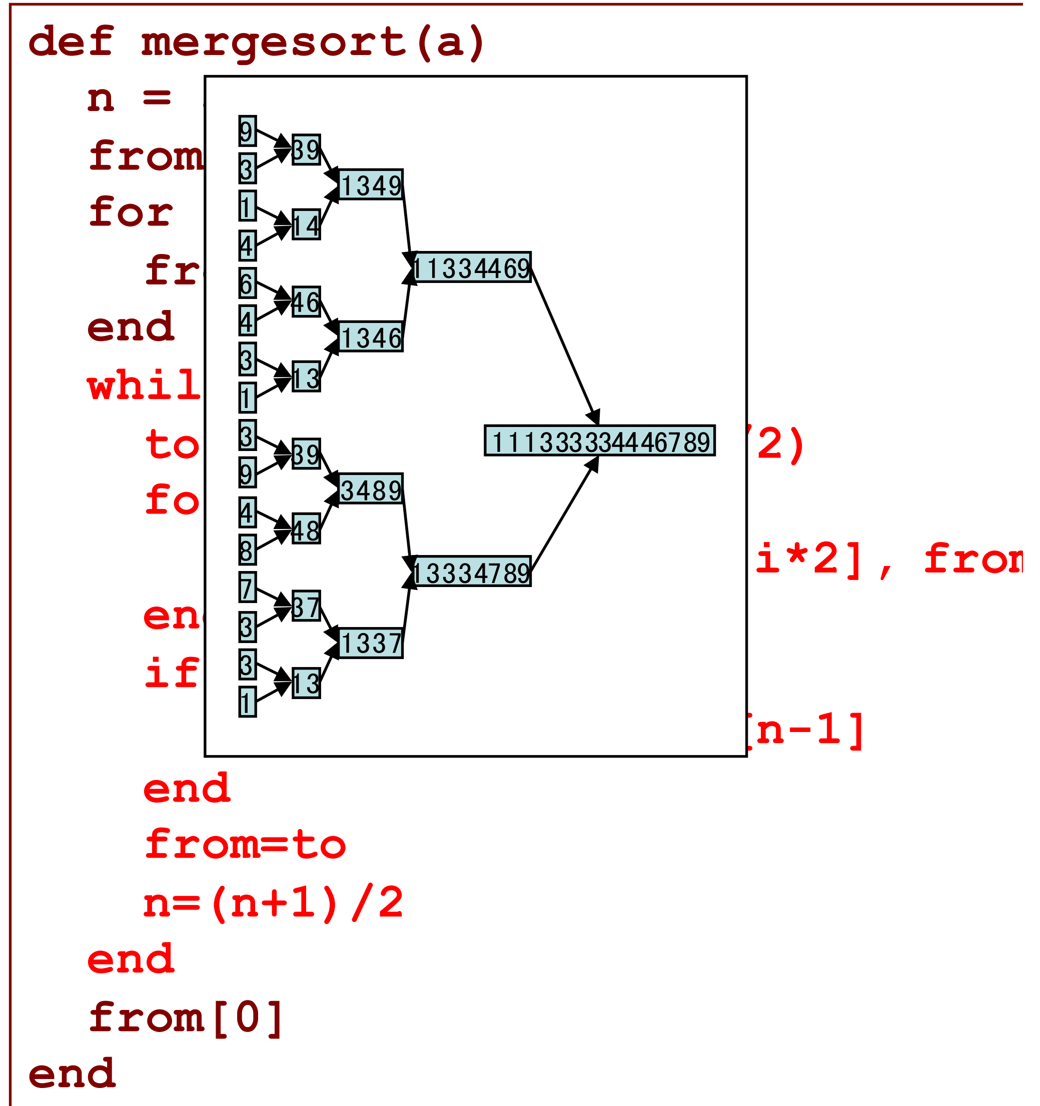
    →時間は約100倍

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

def min_index(a,i)
  暫定最小値の番号kをiにする
  jを(i+1)..(a.length()-1)まで変化させ
    a[j]がa[k]より小さかったら
      kをjにする
  繰り返しが終わった後のkが答え
end
```

計算量: 併合整列法

- nと各山の大きさの変化
  - nは毎回半分になる
  - 山の大きさは倍になる
  - n×山の大きさ=l_a
- while一回あたりの計算量
  - n/2×mergeの計算量
- merge(b,c)の計算量 =O(l_b+l_c)
- whileの回数 $\log_2(l_a)$
- 全体の計算量 = O($l_a \log l_a$)
  - データ量が10倍 →時間は10 log 10倍



速度の比較（練習5.13）

simplesort, mergesortを完成させて
compare_sort(配布)を使って
compare_sort(3000,100)
として、速度を比較してみよう。

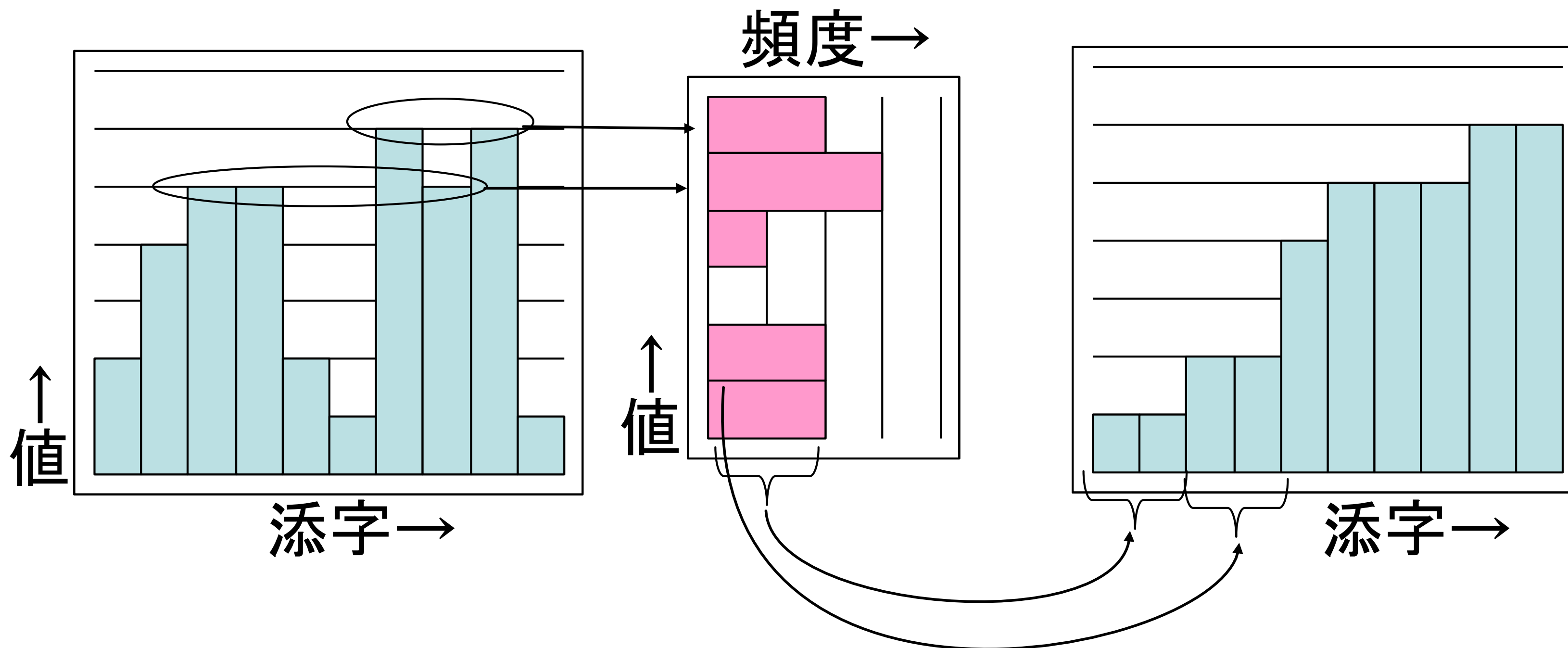
- 完成し、比較ができたら長さ3000のときに単純整列にかかった時間(秒)を投票せよ

```
…
simplesort(3000)...finished in ???????? seconds.
mergesort(3000)...finished in 0.430000 seconds.
1..30
irb(main):089:0>
```

ビン整列法

ビン整列法

- 考え方
  - ➢頻度分布(ヒストグラム)を作る
  - ➢各値の頻度だけ値を並べる

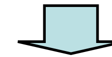


ビン整列法

1. 大きな配列を用意し、全て0にしておく
2. 列の各要素xについて、配列のx番目を1増やす
3. 配列を順に調べ、x番目がnならばxをn個並べる

9 3 1 4 6 4 3 1 3 9 4 8 7 3 3 1 2

0	1	2	3	4	5	6	7	8	9
0	0	0	0	0	0	0	0	0	0



0	1	2	3	4	5	6	7	8	9
0	3	1	5	3	0	1	1	1	2

1 1 1 2 3 3 3 3 3 4 4 4 6 7 8 9 9

おまけ: 整列アルゴリズムの可視化

```
include(Komaba::MiniStar)
def simplesort(a)
  for i in 0..(a.length() - 1)
    p=Person.new()
    p.x = i*30+60
    p.y = 400
    p.height = a[i]+40
    a[i]=p
  end
  for i in 0..(a.length()-1)
    k = min_index(a,i)
    a[k].flash(0.5)
    v = a[i]
    a[i] = a[k]
    a[k].move(i*30+60,400)
    a[k] = v
    v.move(k*30+60,400)
  end
  a
end
```

- include(Komaba::MiniStar): おまじない
- p=Person.new(): 人形を作る
- p.x = i*30+60: 位置を(i*30+60,400)にする
- p.height = a[i]+40: 身長をa[i]+40にする
- a[i]=p: a[i]に人形をしまう
- a[k].flash(0.5): 最小の人形を点滅
- a[k].move(i*30+60,400): 最小の人形をiの位置に移動
- v.move(k*30+60,400): iの位置の人形をkの位置に移動

より詳しい使い方は共通資料ページの「RubyとStar Rubyを使用するコンピュータにインストールする方法と用意されている関数の説明」

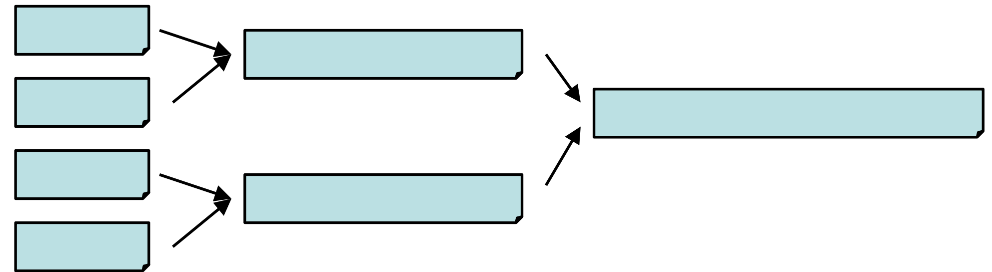
時間計算量と空間計算量

- いままでの計算量は「計算回数」に注目していた
  → **<u>時間</u>**計算量という
- 配列やリストのようなデータ構造を使う場合は、必要とする記憶領域の大きさも問題となる
  - 例: データ数の2乗の記憶領域が必要なアルゴリズムがあったとき、データ数が10^9個(1G個)あったら?
    - 「全てのデータの組」について表を作る場合
    - webページの総数は100億(= 10^{10})以上と推定されている

  → **<u>空間</u>**計算量
- 考え方は時間計算量と同様

空間計算量の例

- 併合ソート
  - データ個数 n の2倍 → O(n)



- ビンソート
  - データ範囲の幅 k とデータの個数 n の大きい方 → O(k+n)
  - 範囲が狭い場合のみ使える