プログラミング言語　第十回

担当：篠沢 佳久
櫻井 彰人

平成21年 6月 22日

本日の内容

- ファイル操作
- 多重ループ(2)
  - エラトステネスの篩
  - 並び替え(ソートアルゴリズム)
- 二次元配列(1)

- 練習問題①～④

ファイル操作

ファイルからの読み込み，書き込み

キーボードからの入力(復習)

- line = gets.chop
- line = gets.chomp

- gets
  - キーボードから文字列を読み込む
  - この場合，改行文字が文字列の最後に含む
- chop
  - 最後の一文字を削除する
- chomp
  - 最後の一文字が改行の場合のみ改行コードを削除する

- line には読み込まれた文字列が代入される
- 文字列のため，数字に「to_i」「to_f」を用いて数値に変換する

ファイルからの読み込み①

```
while 継続条件 do #継続条件が真の間、処理を繰り返す
                  #継続か否かの判定は、一番最初
  式1
  式2
   .
   .
end

# よくある使い方
第一回目のための準備
while 継続条件 do
  するべき作業
  次回への準備
end
```

一行読み込む

```
open( "HumptyDumpty.txt" ) { | f |
  ln = 1; l = f.gets          # 第一回目のための準備
  while !(l.nil?) do
    print( ln, "行目:", l )   # 作業
    ln += 1; l = f.gets       # 次回への準備
  end
}
```

```
1行目:Humpty Dumpty sat on a wall,
2行目:Humpty Dumpty had a great fall.
3行目:All the king's horses and all the king's men
4行目:Couldn't put Humpty together again.
```

http://www.authorama.com/through-the-looking-glass-6.html

よくある簡略表現

- 次のような簡略化が可能となる工夫がされている

```
open( "HumptyDumpty.txt" ) { |f|
  ln = 1; l = f.gets
  while ! (l.nil?) do
    print( ln, "行目:", l )
    ln += 1; l = f.gets
  end
}
```

```
open("HumptyDumpty.txt") { | f |
  ln = 1                    # 第一回目のための準備
  while l = f.gets  do      # 条件判定と次への準備
    print( ln, "行目:", l )  # 肝心の作業
    ln += 1                 # 次回への準備
  end
}
```

```
open("HumptyDumpty.txt") { | f |
  ln = 0
  while ( ln +=1; l = f.gets)  do
    print( ln, "行目:", l )
  end
}
```

禁止です。
でも、どうして動くのだろうか？

ファイルからの読み込み②

読み込みたいファイル名を記述

```
open( "ファイル名") { |f|
    while line = f.gets do

    end
}
```

文字列変数 line にファイルから一行読み込まれる

ファイルの末尾まで読み込まれるとwhile 文から抜け出る

ファイルからの読み込み③

```
open("HumptyDumpty.txt") { |f|
    while line = f.gets do
        puts( line )
    end
}
```

ファイルから一行づつ読み込む

一行出力

HumptyDumpty.txt

```
Humpty Dumpty sat on a wall.
Humpty Dumpty had a great fall.
All the king's horses and all the king's men
Couldn't put Humpty together again.
```

```
H:¥Ruby>ruby sample.rb
Humpty Dumpty sat on a wall.
Humpty Dumpty had a great fall.
All the king's horses and all the king's men
Couldn't put Humpty together again.
```


ファイルからの読み込み④

```
sum = 0
open( "file.txt" ) { |f|
  while line = f.gets do
      x = line.chomp.to_i
      print( x , " " )
      sum += x
  end
  print( "¥n 合計:" , sum )
}
```

file.txt

```
1
2
3
4
5
6
7
8
9
10
```

```
C:¥Ruby>ruby sample.rb
1 2 3 4 5 6 7 8 9 10
 合計:55
```

ファイルからの読み込み⑤

```
sum = 0
x =[]
open( "file1.txt" ) { |f|
  while line = f.gets do
      x = line.chomp.split(/,/)
      print( x[ 0 ] , " " , x[ 1 ] , "¥n" )
      sum += x[ 1 ].to_i
  end
  print( "¥n 合計:" , sum )
}
```

file.txt

```
A,1
B,2
C,3
D,4
E,5
F,6
G,7
H,8
I,9
J,10
```


ファイルからの読み込み⑤'

```
sum = 0
x =[]
open( "file1.txt" ) { |f|
  while line = f.gets do
      x = line.chomp.split(/,/)
      print( x[ 0 ] , " " , x[ 1 ] , "¥n" )
      sum += x[ 1 ].to_i
  end
  print( "¥n 合計:" , sum )
}
```

「,」で分割し，x[0]とx[1]に代入する

```
C:¥Ruby>ruby sample.rb
A 1
B 2
C 3
D 4
E 5
F 6
G 7
H 8
I 9
J 10

 合計:55
```


ファイルへの書き込み①

```
f = open( "text.txt" , "w" )
10.times{ |i|
  f.print( i , "¥n" )
}
f.close
```

type
ファイルの内容を見るコマンド

```
C:¥Ruby>ruby sample.rb

C:¥Ruby>type text.txt
0
1
2
3
4
5
6
7
8
9
```

ファイルへの書き込み②

```
f = open( "text.txt" , "w" )
10.times{ |i|
   f.print( i , "¥n" )
}
f.close
```

text.txt という名前のファイル（書き込み用）を準備

text.txt に書き込む

ファイルを閉じる（忘れずに！）

open("ファイル名" , "モード")

モード（省略した場合は「r」）
w　書き込み用
r　 読み込み用
a　追加書き込み用

ファイルへの書き込み③

```
s = open( "text.txt" , "w" )
open("HumptyDumpty.txt") { |f|
   while line = f.gets  do
      s.print( line )
   end
}
s.close
```

f　読み込み用

s　書き込み用

```
C:¥Ruby>ruby sample.rb

C:¥Ruby>type text.txt
Humpty Dumpty sat on a wall.
Humpty Dumpty had a great fall.
All the king's horses and all the king's men
Couldn't put Humpty together again.
```


エラトステネスの篩

素数を見つける方法（アルゴリズム）

二重ループ①

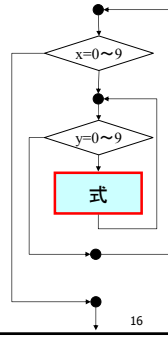
eachを用いた場合

```
(0..9).each{ |x|
   (0..9).each{ |y|
      式
   }
}
```

timesを用いた場合

```
10.times{ |x|
   10.times{ |y|
      式
   }
}
```

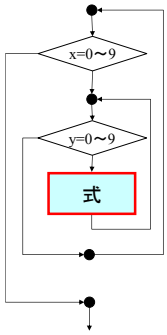
外側と内側の制御変数は異なる名前にする（この場合は，x と y）



二重ループ②

while を用いた場合

```
x = 0
while x < 10 do
      while y < 10 do
            式
            y += 1
      end
      x +=1
end
```



素数の判定

```
n=gets.chomp.to_i
(2..n-1).each{ |x|
    if n % x == 0 then
        print( "この数字は素数ではありません¥n" )
        break
    end
}
```

整数n
2からn-1で割り切れたら素数ではない

```
C:¥Ruby>ruby sample.rb
153
この数字は素数ではありません
```

素数を印字するプログラム

```
# 素数は、2～「自分-1」では割り切れない整数
# 2から10までの素数を印字しよう

(2..10).each{ |n|        # 素数候補
  (2..n-1).each{ |x|     # これで調べる
    break if n%x == 0 # 割り切れたら素数ではない
  }
  puts "#{n} は素数"
}
```

```
2 は素数
3 は素数
4 は素数
5 は素数
6 は素数
7 は素数
8 は素数
9 は素数
10 は素数
=> 2..10
```

あれ？
なぜ，全て素数と表示されるのでしょうか

素数の判定②

```
n=gets.chomp.to_i
p = 1
(2..n-1).each{ |x|
    if n % x == 0 then
        p = 0
        break
    end
}
if p == 0 then
    print( "この数字は素数ではありません¥n" )
else
    print( "この数字は素数です¥n" )
end
```

素数でないと判明した場合p=0とする

```
C:¥Ruby>ruby sample.rb
179
この数字は素数です
```

再: 素数を印字するプログラム

```
# 素数は、2～「自分-1」では割り切れない整数
# 2から10までの素数を印字しよう

(2..10).each{ |n|                # 素数候補
  p = 1                          # まだ素数候補
  (2..n-1).each{ |x|             # これで調べる
    (p=0; break) if n%x == 0 # 割り切れたら素数ではない
  }
  puts "#{n} は素数" if p==1  # 素数候補のままなら素数！
}
```

```
2 は素数
3 は素数
5 は素数
7 は素数
=> 2..10
```

しかし、効率が悪い。余分な割り算をたくさん行っている。
エラトステネスの篩にしたい。
篩はどうすれば表現できるか？

エラトステネスの篩

- 自然数nまでの素数を見つける方法（アルゴリズム）
- 計算機によって，問題を効率的に解く手順を「アルゴリズム」と呼びます
- 次ページから1から100までの素数を求める手順を示します

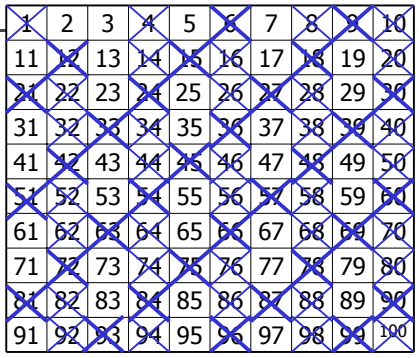
1から100までの配列を用意

1	2	3	4	5	6	7	8	9	10
11	12	13	14	15	16	17	18	19	20
21	22	23	24	25	26	27	28	29	30
31	32	33	34	35	36	37	38	39	40
41	42	43	44	45	46	47	48	49	50
51	52	53	54	55	56	57	58	59	60
61	62	63	64	65	66	67	68	69	70
71	72	73	74	75	76	77	78	79	80
81	82	83	84	85	86	87	88	89	90
91	92	93	94	95	96	97	98	99	100

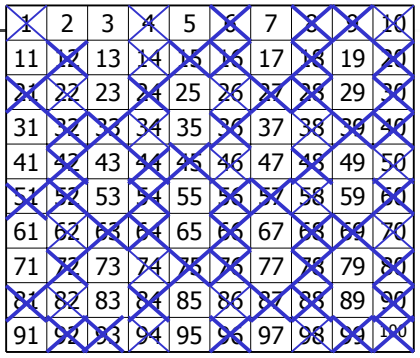
1を削除（0とする）　要素 2×2から2要素ごとに削除

~~1~~	2	3	~~4~~	5	~~6~~	7	~~8~~	9	~~10~~
11	~~12~~	13	~~14~~	15	~~16~~	17	~~18~~	19	~~20~~
21	~~22~~	23	~~24~~	25	~~26~~	27	~~28~~	29	~~30~~
31	~~32~~	33	~~34~~	35	~~36~~	37	~~38~~	39	~~40~~
41	~~42~~	43	~~44~~	45	~~46~~	47	~~48~~	49	~~50~~
51	~~52~~	53	~~54~~	55	~~56~~	57	~~58~~	59	~~60~~
61	~~62~~	63	~~64~~	65	~~66~~	67	~~68~~	69	~~70~~
71	~~72~~	73	~~74~~	75	~~76~~	77	~~78~~	79	~~80~~
81	~~82~~	83	~~84~~	85	~~86~~	87	~~88~~	89	~~90~~
91	~~92~~	93	~~94~~	95	~~96~~	97	~~98~~	99	~~100~~

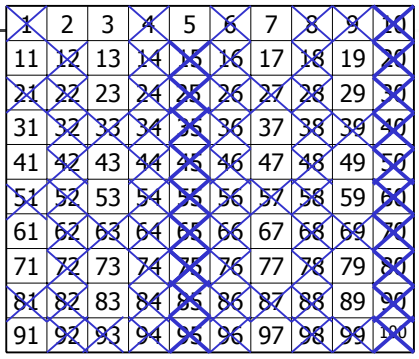
要素 3×2から3要素ごとに削除



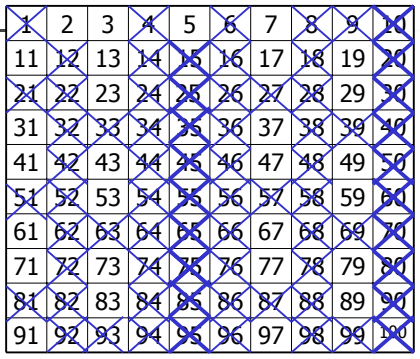
（無駄ですが…）要素 4×2から4要素ごとに削除



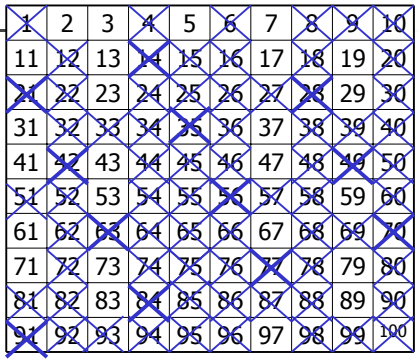
要素 5×2から5要素ごとに削除



（無駄ですが…）要素 6×2から6要素ごとに削除

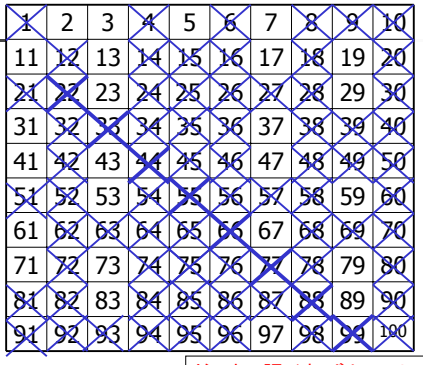


要素 7×2から7要素ごとに削除



8,9,10の場合も同様に

要素 11×2から11要素ごとに削除



どこまで調べればよいでしょう？

削除されていない数が素数

1	2	3	4	5	6	7	8	9	10
11	12	13	14	15	16	17	18	19	20
21	22	23	24	25	26	27	28	29	30
31	32	33	34	35	36	37	38	39	40
41	42	43	44	45	46	47	48	49	50
51	52	53	54	55	56	57	58	59	60
61	62	63	64	65	66	67	68	69	70
71	72	73	74	75	76	77	78	79	80
81	82	83	84	85	86	87	88	89	90
91	92	93	94	95	96	97	98	99	100

エラトステネスの篩のまとめ

- 基本方針
  - 配列 sieve の第 i 要素が
    - 0以外 なら素数候補（いつ「素数候補」から「素数」になるか？）
    - 0 なら素数でないこと確定

    とする

エラトステネスの篩のまとめ

手順

- 配列 sieve を 1 で埋める
  - 最初は、すべてが素数候補
- 配列の第2*2要素から2要素ごとに第100要素まで 0 を代入
- 配列の第3*2要素から3要素ごとに第100要素まで 0 を代入
- 配列の第4*2要素から4要素ごとに第100要素まで 0 を代入　# これは無駄だよね
- 配列の第5*2要素から5要素ごとに第100要素まで 0 を代入
- 配列の第100*2要素から100要素ごとに第100要素まで 0 を代入　# ???

エラトステネスの篩: プログラム例①

```
sieve = Array.new(101)
sieve.length.times { |i|
  sieve[i] = 1
}

sieve[0]=0
sieve[1]=0
(2..sieve.length-1).each { |i|
  (i*2).step(sieve.length-1, i) { |j|
      sieve[j]=0
  }
}

sieve.length.times { |i|
      print( "#{i} " ) if sieve[i] != 0
}
```

101個の配列を用意
sieve[0] は利用しない

初期設定
配列の全要素に1を代入

iは2から100まで変わる

jは2*iから100まで，iごとに変わる

素数でないことをチェック

要素が0でなければ素数と判定

エラトステネスの篩: プログラム例②

```
sieve = Array.new(101)
sieve.length.times { |i|
  sieve[i] = i # ちょっと工夫
}
sieve[0]=0
sieve[1]=0
(2..sieve.length-1).each { |i|
  (i*2).step(sieve.length-1, i) { |j|
      sieve[j]=0
  }
}
sieve.each { |p|
      print( "#{p} " ) if p!=0
}
```

初期設定
0ではなく要素番号を代入

エラトステネスの篩: 実行結果

100までの素数

```
C:¥ruby>ruby sample.rb
2 3 5 7 11 13 17 19 23 29 31 37 41 43 47 53 59 61 67
71 73 79 83 89 97
```

1000までの素数

```
C:¥Ruby>ruby sample.rb
2 3 5 7 11 13 17 19 23 29 31 37 41 43 47 53 59 61 67 71 73 79 83 89 97 101 103 107 109 113 127
37 139 149 151 157 163 167 173 179 181 191 193 197 199 211 223 227 229 233 239 241 251 257 263
71 277 281 283 293 307 311 313 317 331 337 347 349 353 359 367 373 379 383 389 397 401 409 419
31 433 439 443 449 457 461 463 467 479 487 491 499 503 509 521 523 541 547 557 563 569 571 577
93 599 601 607 613 617 619 631 641 643 647 653 659 661 673 677 683 691 701 709 719 727 733 739
51 757 761 769 773 787 797 809 811 821 823 827 829 839 853 857 859 863 877 881 883 887 907 911
29 937 941 947 953 967 971 977 983 991 997
```

この方法は0を無駄に代入している回数が多いです
改良すべき点はどこでしょうか？

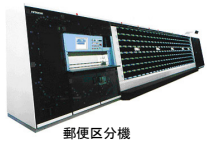
ソーティング

配列の要素の並び替え
バブルソート

ソート: 前書きその1

- 配列要素をある順序に並べ直すこと
  - 事務処理で、最も基本かつ重要な計算。過去さまざまな方法が工夫された
  - とことで、ソートの元の意味は仕分けることであるが、英語でもコンピュータ界では、今では、この意味に使う
  - なぜか？
- ちなみに、現在のソーターの例
  - 「もの」を流しながら、随時、仕分けをしていく

http://www.hokusho.co.jp/main/lineup/as.htm



郵便区分機

http://www.hitachi-omron-ts.co.jp/products/gazou/004.html

ソート: 前書きその2

- 配列要素をある順序に並べ直すこと
  - ソートの元の意味は仕分けることであるが、英語でもコンピュータ界では、今では、この意味に使う
  - なぜか？

This is the first horizontal card sorter, introduced by IBM in 1925 to operate at almost twice the speeds of the older Type 70 vertical sorter. This machine uses a direct magnetically operated control for the chute blades which replaced a much more complex mechanical device in the older machine. The Type 80 grouped all cards of similar classification (such as "sales by products") and at the same time arranged such classifications in numerical sequence. With 10,200 units on rental at the close of 1943, the Type 80 had the largest inventory for any machine at that time.

http://www-03.ibm.com/ibm/history/exhibits/attic3/attic3_136.html

The original Hollerith electric tabulating system did not have an adequate method for sorting cards. This became a problem in the 1900 agricultural census, so Herman Hollerith (1860-1929) developed an automatic sorter. The first one was a tabletop model with the bins arranged horizontally. Later, when his system was gaining favor commercially, Hollerith redesigned the sorter into a sturdier, vertical machine that would not take up too much space in small railroad offices. This 070 Vertical Sorting Machine of 1908 could operate at a rate of 250-270 cards a minute.

http://www-03.ibm.com/ibm/history/exhibits/attic3/attic3_004.html

ソーティングとは

- ソーティングとは
  - データを昇順（小さいものから大きいものへの順）に，もしくは，降順（大きいものから小さいものへの順）に並べ替えること

5, 3, 6, 2, 5, 4

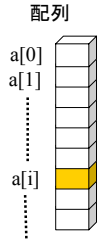
↓ ソートする(昇順)

2, 3, 4, 5, 5, 6

ソーティングの基本操作

- 本日扱うソーティングの対象となるデータ列は，一つの配列に入っているものとする．

・ ソーティングの基本操作
  - 配列要素間の
    - 比較操作と
    - 交換（移動）操作



ソーティングの例

配列a

	a
0	5
1	3
2	6
3	2
4	4
5	5

ソート

配列a

	a
0	2
1	3
2	4
3	5
4	5
5	6

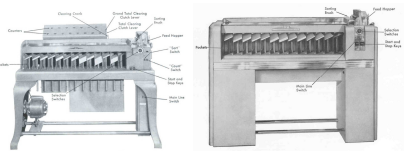
ソーティングのアルゴリズム

- ソーティングには，いろいろなアルゴリズムが知られている．

①馬鹿ソート，②選択ソート
③バブルソート，④シェーカーソート
⑤挿入ソート，　⑥シェルソート
⑦クイックソート，⑧マージソート
⑨基数ソート など

ソート: radix sort

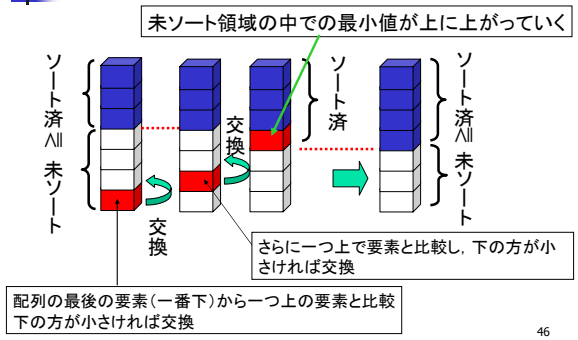
- 実は、区分機を使うと、昇順なり降順に並べることができる
  - 「できる」だけではなく、実際にそうしていた
- たとえば、10進3桁のID番号が振られているカードがあったとしよう
  - 一の位で、0,1,,,9に区分けする
    - 0～9の順に重ねると、下一桁では、0～9の順に並んでいる
  - その順序を崩さずに、十の位で、0,1,,,9に区分けする
    - 0～9の順に重ねると、下二桁では、00～99の順に並んでいる
  - その順序を崩さずに、百の位で、0,1,,,9に区分けする
    - 0～9の順に重ねると、下三桁では、000～999の順に並んでいる



ソート: bubble sort

- コンピュータでは、radix sort は、まず、使わない
- bubble sort も遅いので殆ど使わない。しかし、
  - わかり易いので教育用に
  - プログラムが短いので、ちょこっと使う時には便利
  - bubble とは「泡」
  - 配列の中を（縦に置く）、より小さいものが泡のように上に上がっていく

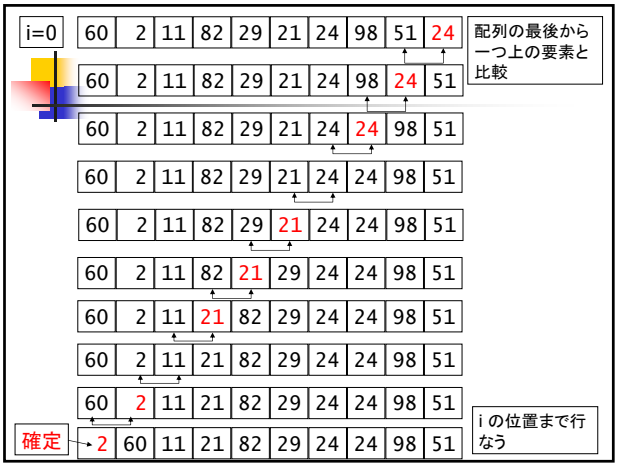
バブルソートのアルゴリズム①

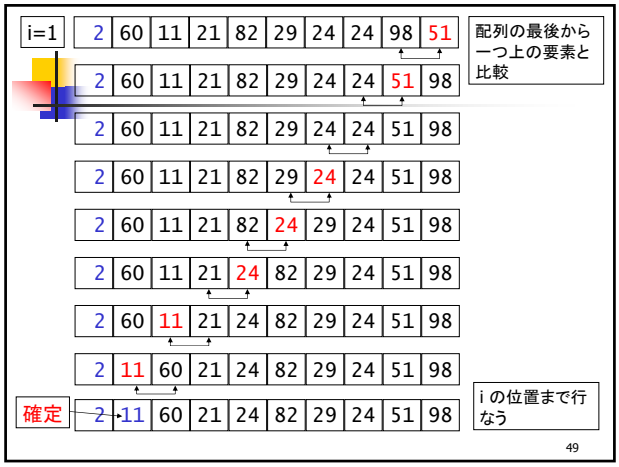


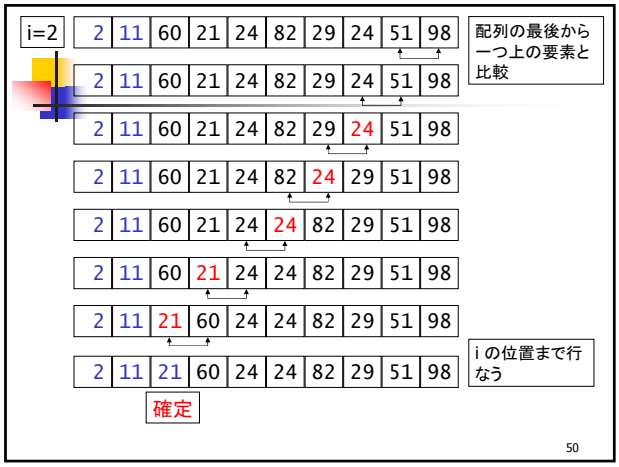
バブルソートの具体例

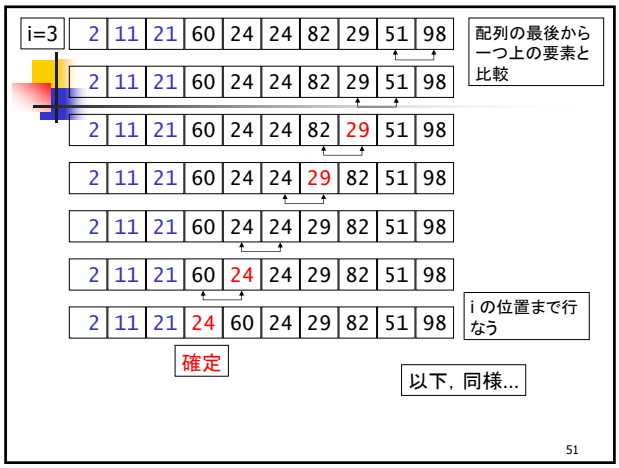
次ページから下記の配列に対してバブルソートを行なう例を示します

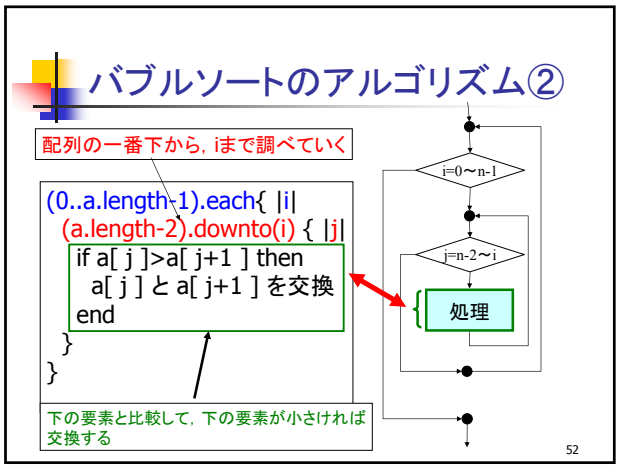
60	2	11	82	29	21	24	98	51	24

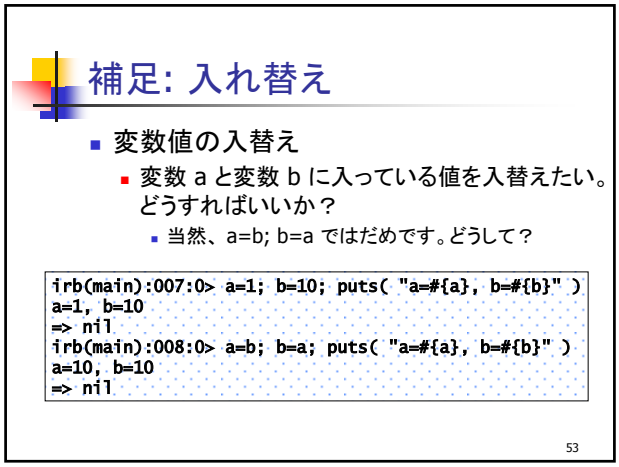


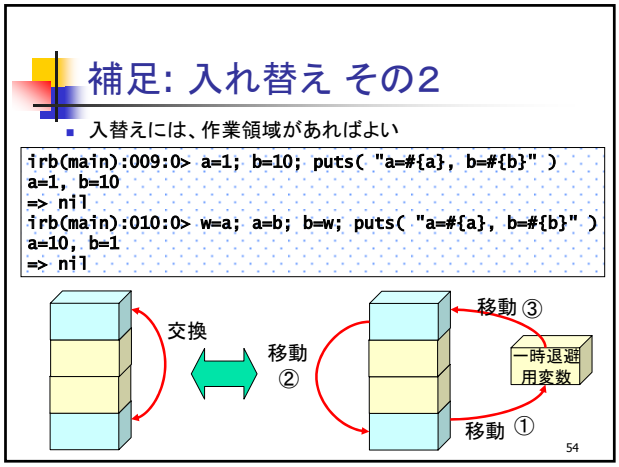

i=1
2 60 11 21 82 29 24 24 98 51
配列の最後から一つ上の要素と比較
2 60 11 21 82 29 24 24 51 98
2 60 11 21 82 29 24 24 51 98
2 60 11 21 82 29 24 24 51 98
2 60 11 21 82 24 29 24 51 98
2 60 11 21 24 82 29 24 51 98
2 60 11 21 24 82 29 24 51 98
2 11 60 21 24 82 29 24 51 98
確定
2 11 60 21 24 82 29 24 51 98
i の位置まで行なう
49


i=2
2 11 60 21 24 82 29 24 51 98
配列の最後から一つ上の要素と比較
2 11 60 21 24 82 29 24 51 98
2 11 60 21 24 82 29 24 51 98
2 11 60 21 24 82 24 29 51 98
2 11 60 21 24 24 82 29 51 98
2 11 60 21 21 24 24 82 29 51 98
2 11 21 60 24 24 82 29 51 98
2 11 21 60 24 24 82 29 51 98
確定
i の位置まで行なう
50


i=3
2 11 21 60 24 24 82 29 51 98
配列の最後から一つ上の要素と比較
2 11 21 60 24 24 82 29 51 98
2 11 21 60 24 24 82 29 51 98
2 11 21 60 24 24 29 82 51 98
2 11 21 60 24 24 29 82 51 98
2 11 21 60 24 24 29 82 51 98
2 11 21 24 60 24 29 82 51 98
確定
i の位置まで行なう
以下，同様...
51


バブルソートのアルゴリズム②
配列の一番下から，iまで調べていく
(0..a.length-1).each{ |i|
(a.length-2).downto(i) { |j|
if a[j]>a[j+1] then
a[j] と a[j+1] を交換
end
}
}
下の要素と比較して，下の要素が小さければ交換する
i=0～n-1
j=n-2～i
処理
52


補足: 入れ替え
変数値の入替え
変数 a と変数 b に入っている値を入替えたい。どうすればいいか？
当然、a=b; b=a ではだめです。どうして？
irb(main):007:0> a=1; b=10; puts("a=#{a}, b=#{b}")
a=1, b=10
=> nil
irb(main):008:0> a=b; b=a; puts("a=#{a}, b=#{b}")
a=10, b=10
=> nil
53


補足: 入れ替え その2
入替えには、作業領域があればよい
irb(main):009:0> a=1; b=10; puts("a=#{a}, b=#{b}")
a=1, b=10
=> nil
irb(main):010:0> w=a; a=b; b=w; puts("a=#{a}, b=#{b}")
a=10, b=1
=> nil
交換
移動 ②
移動 ③
一時退避用変数
移動 ①
54

補足: 入れ替え その3

- 配列要素に対しても同様

```
irb(main):001:0> x=[ 3 , 5 ]
=> [3, 5]
irb(main):002:0> work=x[0]
=> 3
irb(main):003:0> x[0]=x[1]
=> 5
irb(main):004:0> x[1]=work
=> 3
irb(main):005:0> p x
[5, 3]
=> nil
```


bubble sort のプログラム①

```
a=[ 4,1,8,2,6,5 ]
p a
(0..a.length-1).each{ |i|
  (a.length-2).downto(i) { |j|
    if a[j]>a[j+1] then
      w = a[j]
      a[j]=a[j+1]
      a[j+1]=w
    end
  }
}
p a
```

a[j]とa[i]を交換

```
C:¥Ruby>ruby sample.rb
[4, 1, 8, 2, 6, 5]
[1, 2, 4, 5, 6, 8]
```

bubble sort のプログラム②

```
a=[ 4,1,8,2,6,5 ]
p a
(0..a.length-1).each{ |i|
  (a.length-2).downto(i) { |j|
    if a[j]<a[j+1] then
      w = a[j]
      a[j]=a[j+1]
      a[j+1]=w
    end
  }
}
p a
```

逆順にソート
前頁のプログラムとどこが違うでしょうか

```
C:¥Ruby>ruby sample.rb
[4, 1, 8, 2, 6, 5]
[8, 6, 5, 4, 2, 1]
```


bubble sort のプログラム③

```
a=Array.new(10)
a.length.times{ |i|
   a[ i ] = rand( 10 )
}
p a
(0..a.length-1).each{ |i|
  (a.length-2).downto(i) { |j|
    if a[j]>a[j+1] then
      w = a[j]
      a[j]=a[j+1]
      a[j+1]=w
    end
  }
}
p a
```

```
C:¥Ruby>ruby sample.rb
[5, 0, 5, 3, 2, 7, 9, 4, 0, 7]
[0, 0, 2, 3, 4, 5, 5, 7, 7, 9]
```


二次元配列

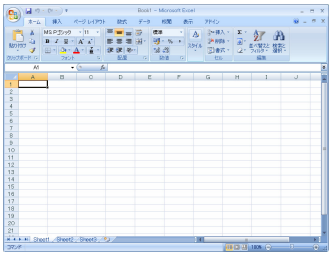
二次元配列の宣言
要素の参照，代入

2次元配列

- 表が使えると、随分便利です。
- スプレッドシートを思い起こしてください
  - スプレッドシートって何ですか？

二次元の配列＝二次元の表（行列）

- 表といえば、二次元かな。
- 表計算ソフトも2次元だしな。



二次元配列の宣言①

3×3の行列 a

$$a = \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix}$$

表の場合

1	2	3
4	5	6
7	8	9

Ruby での宣言

```
a=[
    [1,2,3],
    [4,5,6],
    [7,8,9]
]
```


二次元配列の宣言②

3×3の行列 a

1	2	3
4	5	6
7	8	9

Ruby での宣言

```
a=[
  [1,2,3],
  [4,5,6],
  [7,8,9]
]
```

「,」で区切る

さらに[]で囲む

二次元配列の宣言③

```
a=[
  [1,2,3],
  [4,5,6],
  [7,8,9]
]

p a
```

```
C:¥Ruby>ruby sample.rb
[[1, 2, 3], [4, 5, 6], [7, 8, 9]]
```

一次元配列

二次元配列は一次元配列の要素を一次元配列としているとみなせる

二次元配列の宣言④

```
a=[
  [1,2,3],
  [4,5,6],
  [7,8,9]
]

p a
```

一次元配列

一次元配列

二次元配列は一次元配列の要素を一次元配列としているとみなせる

二次元配列の要素の参照①

```
a=[
  [1,2,3],
  [4,5,6],
  [7,8,9]
]
```

a[0][0]	a[0][1]	a[0][2]
a[1][0]	a[1][1]	a[1][2]
a[2][0]	a[2][1]	a[2][2]

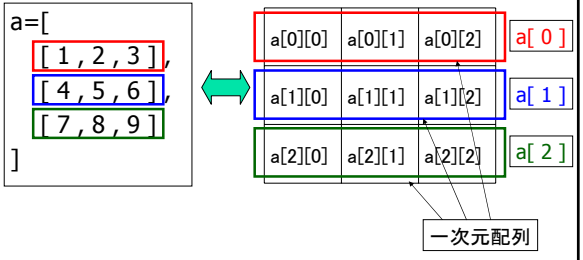
二次元配列の要素の参照②

```
a=[
  [ 1 , 2 , 3 ],
  [ 4 , 5 , 6 ],
  [ 7 , 8 , 9 ]
]

p a[ 0 ][ 0 ]
p a[ 0 ][ 1 ]
p a[ 0 ][ 2 ]

p a[ 1 ][ 0 ]
p a[ 1 ][ 1 ]
p a[ 1 ][ 2 ]

p a[ 2 ][ 0 ]
p a[ 2 ][ 1 ]
p a[ 2 ][ 2 ]
```

```
C:¥Ruby>ruby sample.rb
1
2
3
4
5
6
7
8
9
```


二次元配列の要素の参照③

```
a=[
  [ 1 , 2 , 3 ],
  [ 4 , 5 , 6 ],
  [ 7 , 8 , 9 ]
]
```



二次元配列の要素の参照④

```
a=[
  [ 1 , 2 , 3 ],
  [ 4 , 5 , 6 ],
  [ 7 , 8 , 9 ]
]

p a[ 0 ]
p a[ 1 ]
p a[ 2 ]
```

```
C:¥Ruby>ruby sample.rb
[1, 2, 3]
[4, 5, 6]
[7, 8, 9]
```

a[0]
a[1]
a[2]

二次元配列の要素の参照⑤

```
a=[
  [ 1 , 2 , 3 ],
  [ 4 , 5 , 6 ],
  [ 7 , 8 , 9 ]
]

p a.length
p a[ 0 ].length
p a[ 1 ].length
p a[ 2 ].length
```

配列a の要素数

```
C:¥Ruby>ruby sample.rb
3
3
3
3
```

a[0], a[1], a[2] の要素数

二次元配列の要素の参照⑥

要素数

a.length

1	2	3
4	5	6
7	8	9

a[0]
a[1]
a[2]

```
a=[
  [ 1 , 2 , 3 ],
  [ 4 , 5 , 6 ],
  [ 7 , 8 , 9 ]
]
```

```
a[ 0 ].length
a[ 1 ].length
a[ 2 ].length
```


二次元配列の宣言①

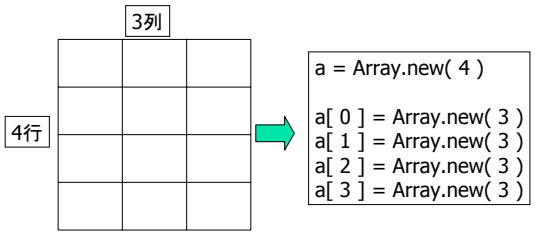
- 要素の値が分かっている場合

1	2	3
4	5	6
7	8	9
10	11	12

```
a=[
  [ 1 , 2 , 3 ],
  [ 4 , 5 , 6 ],
  [ 7 , 8 , 9 ],
  [ 10 , 11 , 12 ]
]
```

二次元配列の宣言②

- 要素数のみ分かっている場合



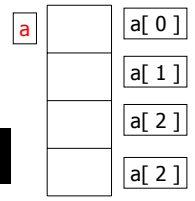
二次元配列の宣言②'

要素数4の配列a を作成

```
a = Array.new( 4 )

p a
```

```
C:¥Ruby>ruby sample.rb
[nil, nil, nil, nil]
```

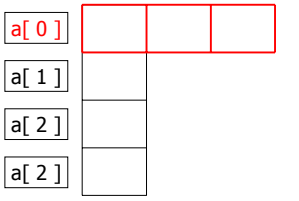
値が入っていないので「nil」となる



二次元配列の宣言②'

```
a = Array.new( 4 )
a[ 0 ] = Array.new( 3 )

p a
```

配列a[0]に要素が3の配列を作成



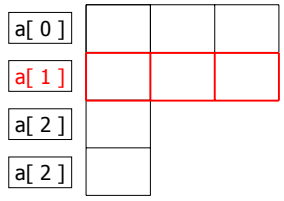

```
C:¥Ruby>ruby sample.rb
[[nil, nil, nil], nil, nil, nil]
```

a[0] のみ3個の要素を持つ配列

二次元配列の宣言②'

```
a = Array.new( 4 )
a[ 0 ] = Array.new( 3 )
a[ 1 ] = Array.new( 3 )

p a
```

配列a[1]に要素が3の配列を作成



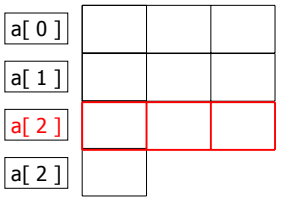
a[0] , a[1]
3個の要素を持つ配列

```
C:¥Ruby>ruby sample.rb
[[nil, nil, nil], [nil, nil, nil], nil, nil]
```


二次元配列の宣言②'

```
a = Array.new( 4 )
a[ 0 ] = Array.new( 3 )
a[ 1 ] = Array.new( 3 )
a[ 2 ] = Array.new( 3 )

p a
```

配列a[2]に要素が3の配列を作成



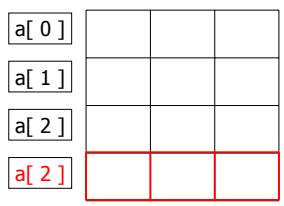
a[0] , a[1] , a[2]
3個の要素を持つ配列

```
C:¥Ruby>ruby sample.rb
[[nil, nil, nil], [nil, nil, nil], [nil, nil, nil], nil]
```


二次元配列の宣言②'

```
a = Array.new( 4 )
a[ 0 ] = Array.new( 3 )
a[ 1 ] = Array.new( 3 )
a[ 2 ] = Array.new( 3 )
a[ 3 ] = Array.new( 3 )
p a
```

配列a[3]に要素が3の配列を作成




```
C:¥Ruby>ruby sample.rb
[[nil, nil, nil], [nil, nil, nil], [nil, nil, nil], [nil, nil, nil]]
```

二次元配列の要素への代入

```
a = Array.new( 4 )

a[ 0 ] = Array.new( 3 )
a[ 1 ] = Array.new( 3 )
a[ 2 ] = Array.new( 3 )
a[ 3 ] = Array.new( 3 )

a[ 0 ][ 0 ] = 1
a[ 0 ][ 1 ] = 2
a[ 0 ][ 2 ] = 3
a[ 1 ][ 0 ] = 4
a[ 1 ][ 1 ] = 5
a[ 1 ][ 2 ] = 6
```

```
a[ 2 ][ 0 ] = 7
a[ 2 ][ 1 ] = 8
a[ 2 ][ 2 ] = 9

a[ 3 ][ 0 ] = 10
a[ 3 ][ 1 ] = 11
a[ 3 ][ 2 ] = 12

p a
```

```
C:¥Ruby>ruby sample.rb
[[1, 2, 3], [4, 5, 6], [7, 8, 9], [10, 11, 12]]
```


二次元配列の宣言③

aは配列と宣言

```
a = []

a[ 0 ] = []
a[ 0 ][ 0 ] = 1
a[ 0 ][ 1 ] = 2
a[ 0 ][ 2 ] = 3

a[ 1 ] = []
a[ 1 ][ 0 ] = 4
a[ 1 ][ 1 ] = 5
a[ 1 ][ 2 ] = 6
```

```
a[ 2 ] = []
a[ 2 ][ 0 ] = 7
a[ 2 ][ 1 ] = 8
a[ 2 ][ 2 ] = 9

a[ 3 ] = []
a[ 3 ][ 0 ] = 10
a[ 3 ][ 1 ] = 11
a[ 3 ][ 2 ] = 12

p a
```

a[0], a[1], a[2], a[3]が配列であることを宣言

```
C:¥Ruby>ruby sample.rb
[[1, 2, 3], [4, 5, 6], [7, 8, 9], [10, 11, 12]]
```


二次元配列の宣言③'

```
a =[]

p a
```

配列a を作成

```
C:¥Ruby>ruby sample.rb
[]
```

```
a =[]
a[ 0 ]=[]

p a
```

配列a[0] を作成

```
C:¥Ruby>ruby sample.rb
[[]]
```


二次元配列の宣言③"

```
a=[]
a[ 0 ] = []
a[ 0 ][ 0 ] = 1
a[ 0 ][ 1 ] = 2
a[ 0 ][ 2 ] = 3

p a
```

```
C:¥Ruby>ruby sample.rb
[[1, 2, 3]]
```

```
a=[]
a[ 0 ] = []
a[ 0 ][ 0 ] = 1
a[ 0 ][ 1 ] = 2
a[ 0 ][ 2 ] = 3

a[ 1 ] =[]

p a
```

```
C:¥Ruby>ruby sample.rb
[[1, 2, 3], []]
```


二次元配列の宣言③'''

```
a = []

a[ 0 ] = []
a[ 0 ][ 0 ] = 1
a[ 0 ][ 1 ] = 2
a[ 0 ][ 2 ] = 3

a[ 1 ] = []
a[ 1 ][ 0 ] = 4
a[ 1 ][ 1 ] = 5
a[ 1 ][ 2 ] = 6

p a
```

```
C:¥Ruby>ruby sample.rb
[[1, 2, 3], [4, 5, 6]]
```


二次元配列の宣言④

配列の宣言をしない場合

```
a = []

a[ 0 ][ 0 ] = 1
a[ 0 ][ 1 ] = 2
a[ 0 ][ 2 ] = 3

a[ 1 ][ 0 ] = 4
a[ 1 ][ 1 ] = 5
a[ 1 ][ 2 ] = 6
```

```
a[ 2 ][ 0 ] = 7
a[ 2 ][ 1 ] = 8
a[ 2 ][ 2 ] = 9

a[ 3 ][ 0 ] = 10
a[ 3 ][ 1 ] = 11
a[ 3 ][ 2 ] = 12

p a
```

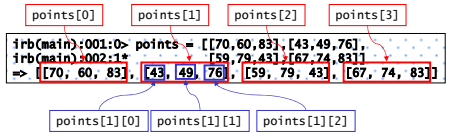
```
C:¥Ruby>ruby sample.rb
sample.rb:3: undefined method `[]=' for nil:NilClass (NoMethodError)
```

二次元配列の宣言のまとめ

- 要素の値が分かっている場合
- 要素数のみ分かっている場合
- 要素の値，要素数も分からない場合

二次元配列のまとめ①

- 一次元配列の一次元配列
  - Ruby では、子供の一次元配列の長さは異なってよい。Java でも同様。Cではダメ。



points.length は 4, points[0].length は 3

二次元配列のまとめ②

- 一次元配列の一次元配列だから

```
irb(main):001:0> points = [[1, 70, 60, 83], [2, 43, 49, 76], [3, 59, 79,
43], [4, 67, 74, 83]]
=> [[1, 70, 60, 83], [2, 43, 49, 76], [3, 59, 79, 43], [4, 67, 74, 83]]
irb(main):002:0> points[0]
=> [1, 70, 60, 83]
irb(main):003:0> points[0][0] = 999
=> 999
irb(main):004:0> points
=> [[999, 70, 60, 83], [2, 43, 49, 76], [3, 59, 79, 43], [4, 67, 74, 83]]
```

```
irb(main):004:0> points
=> [[999, 70, 60, 83], [2, 43, 49, 76], [3, 59, 79, 43], [4, 67, 74, 83]]
irb(main):005:0> p = points[0]
=> [999, 70, 60, 83]
irb(main):006:0> p[0] = 99
=> 99
irb(main):007:0> p
=> [99, 70, 60, 83]
irb(main):008:0> points
=> [[99, 70, 60, 83], [2, 43, 49, 76], [3, 59, 79, 43], [4, 67, 74, 83]]
```

p と points[0] が同じものとなる

注意!

p の要素を変えたら points[0] の要素が変わった！

練習問題

練習問題①～④

練習問題①

- 二つの整数を入力し，公約数を求めるプログラムを作成しなさい

```
C:¥Ruby>ruby sample.rb
64
12
1  2  4
```


練習問題②

- **Collatz-角谷の予想**
- 自然数nを選び，
  - 奇数ならば，3倍して1をたす。
  - 偶数ならば，2で割る。

これを繰り返すと，どんなnを選んでも，いつかは，1になる

```
3, 10, 5, 16, 8, 4, 2, 1
4, 2, 1
5, 16, 8, 4, 2, 1
6, 3, 10, 5, 16, 8, 4, 2, 1
7, 22, 11, 34, 17, 52, 26, 13, 40, 20, 10, 5, 16, 8, 4, 2, 1
8, 4, 2, 1
9, 28, 14, 7, 22, 11, 34, 17, 52, 26, 13, 40, 20, 10, 5, 16,
8, 4, 2, 1
10, 5, 16, 8, 4, 2, 1
11, 34, 17, 52, 26, 13, 40, 20, 10, 5, 16, 8, 4, 2, 1
```

練習問題②

- 1から10までCollatz-角谷の予想が正しいことを確認できるプログラムを作成しなさい

練習問題③

- 二次元配列 a 内に整数が記憶されているとする。
- 二重ループを用いて各要素の値を印字しなさい。

```
a=[
   [1,2,3],
   [4,5,6],
   [7,8,9]
  ]
```

```
a=[
   [1],
   [2,3],
   [4,5,6]
  ]
```


練習問題③'

- 二次元配列 a, b の和と差を印字するプログラムを二重ループを用いて書きなさい

```
a=[
   [1,2,3],
   [4,5,6],
   [7,8,9]
  ]
```

```
b=[
   [9,8,7],
   [6,5,4],
   [3,2,1]
  ]
```

```
C:¥Ruby>ruby sample.rb
10 10 10
10 10 10
10 10 10

-8 -6 -4
-2 0 2
4 6 8
```


練習問題④

- 配列 a 内に整数が記憶されているとする。
- この内容を、バブルソートの一行を書き換えることによって，偶数と奇数にわけ，配列 a に入れ直すことを実現しなさい。
- ただし，配列 a の前半に偶数、後半に奇数とし，偶数同士の順序、奇数同士の順序は保存せよ

配列a

```
a = [3,1,4,1,5,9,2,6,5,3,5,8,9,7,9,3,2,3]
```

```
[4, 2, 6, 8, 2, 3, 1, 1, 5, 9, 5, 3, 5, 9, 7, 9, 3, 3]
```


練習問題④

```
a = [3,1,4,1,5,9,2,6,5,3,5,8,9,7,9,3,2,3]
p a
(0..a.length-1).each{ |i|
  (a.length-2).downto(i) { |j|
    if a[j]>a[j+1] then
      w = a[j]
       a[j]=a[j+1]
      a[j+1]=w
    end
  }
}
p a
```

どこか一行変更してみて下さい

練習問題④

- ヒント
  - どういうときに交換すればいいのでしょうか？
  - a[j]が〇〇，かつ a[j+1]が〇〇の時，交換すればいいのかな？

練習問題

- 練習問題①から④を（できるだけ）（頑張って）行ないなさい。
- プログラムと実行結果をワープロに貼り付けて、keio.jp から提出して下さい。