プログラミング言語　第九回

担当：篠沢 佳久
櫻井 彰人

平成20年 6月 23日

本日の内容

- 一次元配列の復習
- 多重ループ
- ネスト（入れ子）構造
- 練習問題①～④

ネスト（入れ子）

- ネストする: 入れ子にすること

箱根の十二卵（田中一幸氏作） 左端は実際の鶏卵のLL玉ほど。右端は13番目のヒヨコ

http://dadandmam.whitesnow.jp/sub3-011.htm

配列の復習

一次元配列と繰り返し

配列の宣言

- 要素が分かっている場合
  - 配列名 = [値1, 値2, … , 値n]
- 要素数のみが決まっている場合
  - 配列名 = Array.new(要素数)
- 要素数が決まっていない場合
  - 配列名 = []

配列の宣言②

要素が分かっている場合

a=[4, 6, 7, 9, 10]

	a	
0	4	a[0]
1	6	a[1]
2	7	a[2]
3	9	a[3]
4	10	a[4]

配列の宣言②

要素数が分かっている場合

```
a= Array.new(5)

a[0]=4
a[1]=6
a[2]=7
a[3]=9
a[4]=10
```

要素数が決まっていない場合

```
a= []

a[0]=4
a[1]=6
a[2]=7
a[3]=9
a[4]=10
```


配列の要素の参照方法①

- 要素番号で要素の値を参照したい場合

a=[1,3,5,7,9]

```
a.length.times{ |i|
      print( a[ i ] , "¥n" )
}
```

```
5.times{ |i|
      print( a[ i ] , "¥n" )
}
```

```
(0..a.length-1).each{ |i|
   print( a[ i ] , "¥n" )
}
```

配列名.length
配列の要素数

配列の要素の参照方法②

- 要素を直接参照したい場合

```
a=[1,3,5,7,9]
a.each{ |i|
   print( i , "¥n" )
}
```

```
[1,3,5,7,9].each{ |i|
   print( i , "¥n" )
}
```


一次元配列のプログラム例

```
name = [ "A" , "B" , "C" , "D" , "E" ]
test = [ 85 , 60 , 5 , 100 , 50 ]
```

配列name

文字列型

	name
0	A
1	B
2	C
3	D
4	E

配列 test

整数型

	test
0	85
1	60
2	5
3	100
4	50

配列の要素への代入

```
name = [ "A" , "B" , "C" , "D" , "E" ]
test = [ 85 , 60 , 5, 100 , 50 ]

name[ 3 ] = "d"
test[ 3 ] = 90

p name
p test
```

```
C:¥Ruby>ruby sample.rb
["A", "B", "C", "d", "E"]
[85, 60, 5, 90, 50]
```


最後の要素への追加

```
name = [ "A" , "B" , "C" , "D" , "E" ]
test = [ 85 , 60 , 5, 100 , 50 ]

name[ name.length ] = "F"
test[ test.length ] = 70

p name
p test
```

```
C:¥Ruby>ruby sample.rb
["A", "B", "C", "D", "E", "F"]
[85, 60, 5, 100, 50, 70]
```

平均点を求める①

```
name = [ "A" , "B" , "C" , "D" , "E" ]
test = [ 85 , 60 , 5, 100 , 50 ]

sum = 0
test.length.times{ |i|
   sum += test[ i ]
}
print( "平均点 --> " , sum / test.length )
```

times を用いた方法

```
C:¥ruby>ruby sample.rb
平均点 --> 60
```


平均点を求める②

```
name = [ "A" , "B" , "C" , "D" , "E" ]
test = [ 85 , 60 , 5, 100 , 50 ]

sum = 0
(0..test.length-1).each{ |i|
   sum += test[ i ]
}
print( "平均点 --> " , sum / test.length )
```

each を用いた方法

```
C:¥ruby>ruby sample.rb
平均点 --> 60
```


平均点を求める③

```
name = [ "A" , "B" , "C" , "D" , "E" ]
test = [ 85 , 60 , 5, 100 , 50 ]

sum = 0
test.each{ |i|
   sum += i
}
print( "平均点 --> " , sum / test.length )
```

each を用いた方法
配列の要素を直接参照

前頁との違いに注意して下さい

```
C:¥ruby>ruby sample.rb
平均点 --> 60
```


平均点未満の名前を出力

```
name = [ "A" , "B" , "C" , "D" , "E" ]
test = [ 85 , 60 , 5, 100 , 50 ]
sum = 0
test.each{ |i|
   sum += i
}
average = sum / test.length

print( "平均点未満は...¥n" )
test.length.times{ |i|
   if average > test[ i ] then
      print( name[ i ] , ": " , test[ i ] , "点¥n" )
   end
}
```

平均点を求める

```
C:¥ruby>ruby sample.rb
平均点未満は...
C: 5
E: 50
```


最高点とその名前を出力

```
name = [ "A" , "B" , "C" , "D" , "E" ]
test = [ 85 , 60 , 5, 100 , 50 ]

max = 0
test.length.times{ |i|
   if test[ max ] < test[ i ] then
      max = i
   end
}
print( "最高点は" , name[ max ] , " の " , test[ max ] , "点です¥n" )
```

max
最高点の要素番号を格納する

現在の最高点と比較

```
C:¥ruby>ruby sample.rb
最高点はD の 100点です
```


最低点とその名前を出力

```
name = [ "A" , "B" , "C" , "D" , "E" ]
test = [ 85 , 60 , 5, 100 , 50 ]

min = 0
test.length.times{ |i|
   if test[ min ] > test[ i ] then
      min = i
   end
}
print( "最低点は" , name[ min ] , " の " , test[ min ] , "点です¥n" )
```

min
最低点の要素番号を格納する

現在の最低点と比較

```
C:¥ruby>ruby sample.rb
最低点はC の 5点です
```

検索①

```
name = [ "A" , "B" , "C" , "D" , "E" ]
test = [ 85 , 60 , 5, 100 , 50 ]

search = "B"
name.length.times{ |i|
   if name[ i ] == search then
        print( search , " の点は " , test[ i ] , "点です¥n" )
        break
   end
}
```

Bの点数を検索

```
C:¥ruby>ruby sample.rb
B の点は 60点です
```


配列の要素の検索

```
if name[ i ] == search then
```

配列name

	name
0	A
1	B
2	C
3	D
4	E

search = "B"

search と一致した場合，breakにより timesのループから抜け出るため，以降は照合しない

検索②

```
name = [ "A" , "B" , "C" , "D" , "E" ]
test = [ 85 , 60 , 5, 100 , 50 ]

line = ARGV[ 0 ]
search = line.chomp

name.length.times{ |i|
   if name[ i ] == search then
        print( search , " の点は " , test[ i ] , "点です¥n" )
        break
   end
}
```

コマンドライン引数

コマンドライン引数①

- ruby プログラム 値1 値2 値3
  - 値1，値2，・・・を引数と呼ぶ
- 値1は ARGV[0] に格納される
- 値2は ARGV[1] に格納される
- 値3は ARGV[2]に格納される
- ARGV[0]，ARGV[1]，ARGV[2] は文字列型

配列

コマンドライン引数②

- ruby プログラム 値1 値2 値3 … 値n
- 値1は ARGV[0] に格納される
- 値2は ARGV[1] に格納される
- 値nは ARGV[n]に格納される

検索②の実行結果

```
C:¥ruby>ruby sample.rb A
A の点は 85点です

C:¥ruby>ruby sample.rb B
B の点は 60点です

C:¥ruby>ruby sample.rb C
C の点は 5点です

C:¥ruby>ruby sample.rb D
D の点は 100点です

C:¥ruby>ruby sample.rb E
E の点は 50点です
```

ARGV[0] には"A¥n" が代入

line = ARGV[0]

line には "A¥n"

search = line.chomp

search には "A" が代入

二重ループ

一重ループ

$y = x^2$

```
10.times{ |x|
        y = x*x
        print( x , ": " , y , "¥n" )
}
```

```
(0..9).each{ |x|
        y = x*x
        print( x , ": " , y , "¥n" )
}
```

```
C:¥ruby>ruby sample.rb
0: 0
1: 1
2: 4
3: 9
4: 16
5: 25
6: 36
7: 49
8: 64
9: 81
```


二重ループの必要性①

$z = x^2 + y^2$

$0 \le x < 10, 0 \le y < 10$ の範囲で値を求めるには？

x=0 の時，yの値を0から9まで変えて z を求める

```
x = 0
(0..9).each{ |y|
        z = x*x + y*y
        print( x , " " , y , ": " , z , "¥n" )
}
```


x=1 の時，yの値を0から9まで変えて z を求める

```
x = 1
(0..9).each{ |y|
        z = x*x + y*y
        print( x , " " , y , ": " , z , "¥n" )
}
```

以下同様にx=9 まで同じことを繰り返し z を求める

```
x = 9
(0..9).each{ |y|
        z = x*x + y*y
        print( x , " " , y , ": " , z , "¥n" )

}
```


二重ループの必要性②

```
x = 0
(0..9).each{ |y|
        z = x*x + y*y
        print( x , " " , y , ": " , z , "¥n" )
}
```

xの値も0から9まで一つずつ増やしていけばよい

```
x = 1
(0..9).each{ |y|
        z = x*x + y*y
        print( x , " " , y , ": " , z , "¥n" )
}
```

```
x = 9
(0..9).each{ |y|
        z = x*x + y*y
        print( x , " " , y , ": " , z , "¥n" )
}
```


二重ループ

①のループによって，xは0から9まで変わる

①のループ

```
(0..9).each{ |x|
    (0..9).each{ |y|
        z = x*x + y*y
        print( " x = " , x , " y = " , y , ": z= " , z , "¥n" )
    }
}
```

②のループ

②のループによって，xは0から9まで変わる

二重ループの出力結果①

①のループ中 x=0 として
②のループの処理を行なう

```
C:¥ruby>ruby sample.rb
 x = 0 y = 0: z= 0
 x = 0 y = 1: z= 1
 x = 0 y = 2: z= 4
 x = 0 y = 3: z= 9
 x = 0 y = 4: z= 16
 x = 0 y = 5: z= 25
 x = 0 y = 6: z= 36
 x = 0 y = 7: z= 49
 x = 0 y = 8: z= 64
 x = 0 y = 9: z= 81
```

①のループ中 x=1 として
②のループの処理を行なう

```
x = 1 y = 0: z= 1
x = 1 y = 1: z= 2
x = 1 y = 2: z= 5
x = 1 y = 3: z= 10
x = 1 y = 4: z= 17
x = 1 y = 5: z= 26
x = 1 y = 6: z= 37
x = 1 y = 7: z= 50
x = 1 y = 8: z= 65
x = 1 y = 9: z= 82
```


二重ループの出力結果②

①のループ中 x=9 として
②のループの処理を行ない終了する

```
x = 9 y = 0: z= 81
x = 9 y = 1: z= 82
x = 9 y = 2: z= 85
x = 9 y = 3: z= 90
x = 9 y = 4: z= 97
x = 9 y = 5: z= 106
x = 9 y = 6: z= 117
x = 9 y = 7: z= 130
x = 9 y = 8: z= 145
x = 9 y = 9: z= 162
```


二重ループのまとめ①

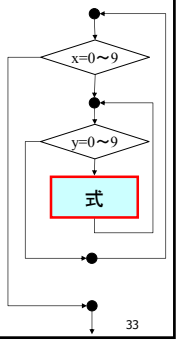
eachを用いた場合

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

timesを用いた場合

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

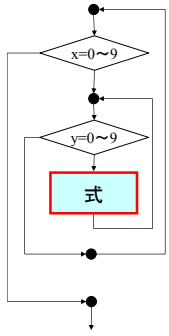
外側と内側の制御変数は異なる名前にする
（この場合は，x と y）



二重ループのまとめ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
```



二重ループの例

二重ループの例①

九九の表の表示プログラム

```
(1..9).each{ |x|
   (1..9).each{ |y|
      printf( " %d×%d=%2d" , x , y , x * y  )
   }
   print( "¥n" )
}
```

```
x=1
(1..9).each{ |y|
    printf( " %d×%d=%2d" , x , y , x * y  )
}
print( "¥n" )
```

x=1,2,...9と変わっていく

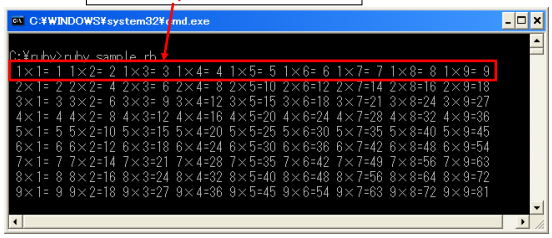
```
x=2
(1..9).each{ |y|
    printf( " %d×%d=%2d" , x , y , x * y  )
}
print( "¥n" )
```

```
x=9
(1..9).each{ |y|
    printf( " %d×%d=%2d" , x , y , x * y  )
}
print( "¥n" )
```


二重ループの例①

前頁の実行画面

xを1とし，yを1から9まで変える



二重ループの例②

対角行列の表示プログラム

```
(1..9).each{ |x|
        (1..9).each{ |y|
                if x == y then
                        print( "1 " )
                else
                        print( "0 " )
                end
        }
        print( "¥n" )
}
```

xとyの値が同じ→"1 "
異なる場合は→"0 "

```
C:¥ruby>ruby sample.rb
1 0 0 0 0 0 0 0 0
0 1 0 0 0 0 0 0 0
0 0 1 0 0 0 0 0 0
0 0 0 1 0 0 0 0 0
0 0 0 0 1 0 0 0 0
0 0 0 0 0 1 0 0 0
0 0 0 0 0 0 1 0 0
0 0 0 0 0 0 0 1 0
0 0 0 0 0 0 0 0 1
```


二重ループの例②の出力結果

```
C:¥ruby>ruby sample.rb
1 0 0 0 0 0 0 0 0
0 1 0 0 0 0 0 0 0
0 0 1 0 0 0 0 0 0
0 0 0 1 0 0 0 0 0
0 0 0 0 1 0 0 0 0
0 0 0 0 0 1 0 0 0
0 0 0 0 0 0 1 0 0
0 0 0 0 0 0 0 1 0
0 0 0 0 0 0 0 0 1
```

x=y=1の場合
x=y=2の場合
x=y=3の場合
x=y=4の場合
x=y=5の場合
x=y=6の場合
x=y=7の場合
x=y=8の場合
x=y=9の場合

二重ループの例③

```
(1..9).each{ |x|
    (1..9).each{ |y|
            if x == (10-y) then
                    print( "1 " )
            else
                    print( "0 " )
            end
    }
    print( "¥n" )
}
```

xと(10-y)の値が同じ→"1 "
異なる場合は→"0 "

```
C:¥ruby>ruby sample.rb
0 0 0 0 0 0 0 0 1
0 0 0 0 0 0 0 1 0
0 0 0 0 0 0 1 0 0
0 0 0 0 0 1 0 0 0
0 0 0 0 1 0 0 0 0
0 0 0 1 0 0 0 0 0
0 0 1 0 0 0 0 0 0
0 1 0 0 0 0 0 0 0
1 0 0 0 0 0 0 0 0
```


二重ループの例③の出力結果

```
C:¥ruby>ruby sample.rb
0 0 0 0 0 0 0 0 1
0 0 0 0 0 0 0 1 0
0 0 0 0 0 0 1 0 0
0 0 0 0 0 1 0 0 0
0 0 0 0 1 0 0 0 0
0 0 0 1 0 0 0 0 0
0 0 1 0 0 0 0 0 0
0 1 0 0 0 0 0 0 0
1 0 0 0 0 0 0 0 0
```

x=1,y=9
x=2,y=8
x=3,y=7
x=4,y=6
x=5,y=5
x=6,y=4
x=7,y=3
x=8,y=2
x=9,y=1

二重ループの例④

```
(1..9).each { |i|
  (1..i).each { |j|
        print( j )
  }
  print ( "¥n" )
}
```

j は 1から i まで変わる

```
C:¥ruby>ruby sample.rb
1
12
123
1234
12345
123456
1234567
12345678
123456789
```


二重ループの例④の出力結果

```
C:¥ruby>ruby sample.rb
1
12
123
1234
12345
123456
1234567
12345678
123456789
```

i=1の時, j=1
i=1の時, j=1～2
i=1の時, j=1～3
i=1の時, j=1～8
i=1の時, j=1～9

二重ループの例⑤

```
(1..9).each { |i|
  (1..(10-i)).each{ |j|
     print( j )
  }
  print ( "¥n" )
}
```

j は 1から 10-i まで変わる

```
C:¥ruby>ruby sample.rb
123456789
12345678
1234567
123456
12345
1234
123
12
1
```


二重ループの例⑤の出力結果

```
C:¥ruby>ruby sample.rb
123456789
12345678
1234567
123456
12345
1234
123
12
1
```

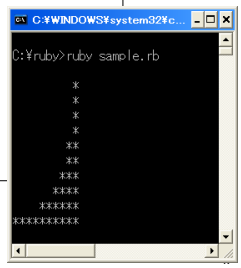
i=1の時, j=1～9
i=1の時, j=1～8
i=1の時, j=1～7
i=1の時, j=1～2
i=1の時, j=1

二重ループの例⑥

```
11.times { |i|
  d = Math.sqrt( 100 - i*i ).to_i
  (1..d).each{
     print( " " )
  }
  ((d+1)..10).each{
     print( "*" )
  }
  print( "¥n" )
}
```

d回は" "(空白)を表示

10-d回は"*"を表示



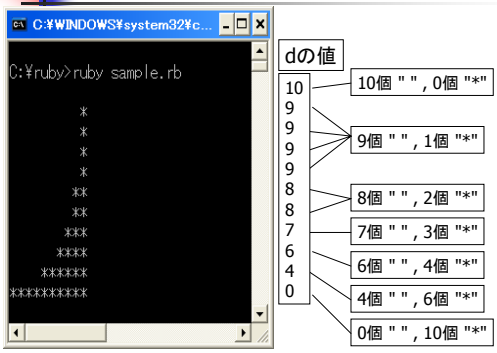
二重ループの例⑥

```
11.times { |i|
  d = Math.sqrt( 100 - i*i ).to_i
  print( d , "¥n" )
}
```

d の値はどう変わっていっているでしょうか

```
C:¥ruby>ruby sample.rb
10
9
9
9
9
8
8
7
6
4
0
```

二重ループの例⑥の出力結果

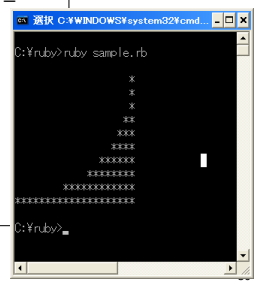


二重ループの例⑦
（どうしてこのような出力になるのでしょうか）

```
11.times { |i|
  d = Math.sqrt( 400 - 4*i*i ).to_i
  (1..d).each{
    print( " " )
  }
  ((d+1)..20).each{
    print( "*" )
  }
  print( "¥n" )
}
```

d回は" "（空白）を表示

20-d回は"*"を表示



二重ループの例⑦
（ヒント：dの値はどう変わっていくでしょうか）

```
11.times { |i|
  d = Math.sqrt( 400 - 4*i*i ).to_i
  print( d , "¥n" )
}
```

```
C:¥ruby>ruby sample.rb
20
19
19
19
18
17
16
14
12
8
0
```


Collatz-角谷の予想

```
x = gets.chomp.to_i
print( x , " " )
loop{
  if x % 2 == 0 then
      x = x / 2
  else
      x = x * 3 + 1
  end

  print( x , " " )
  break if x == 1
}
print( "¥n" )
```

```
C:¥ruby>ruby sample.rb
12
12 6 3 10 5 16 8 4 2 1
```


Collatz-角谷の予想の拡張

```
(1..10).each{ |x|
  print( x , " " )
  loop{
      if x % 2 == 0 then
          x = x / 2
      else
          x = x * 3 + 1
      end
      print( x , " " )
      break if x == 1
  }
  print( "¥n" )
}
```

自然数1からn(この場合は10)において，性質を確認するプログラム

xの値における性質を調べる

出力結果

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

「1」で終わっていることを確認

金利の計算①

10年後までの計算

```
r = 0.05
n = 10
m = 10000
(1..n).each{ |i|
   x = 10000.0 * ( 1.0 + r ) ** i.to_f
   printf( "%2d 年後は %d¥n" , i , x.to_i )
}
```

利率

元金

1年から10年まで変化

金利の計算①（出力結果）

```
C:¥Ruby>ruby sample.rb
 1 年後は 10500
 2 年後は 11025
 3 年後は 11576
 4 年後は 12155
 5 年後は 12762
 6 年後は 13400
 7 年後は 14071
 8 年後は 14774
 9 年後は 15513
10 年後は 16288
```


金利の計算②

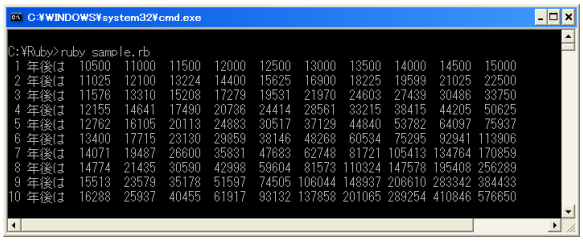
年，利率を変化させた場合

```
n = 10
m = 10000
(1..n).each{ |i|
   printf( "%2d 年後は " , i )
   (1..10).each{ |j|
      r = j * 0.05
      x = 10000.0 * ( 1.0 + r ) ** i.to_f
      printf( "%6d " , x.to_i )
   }
   print( "¥n" )
}
```

1年から10年まで変化

利率を0.05から0.5まで0.5ずつ変化

金利の計算②（出力結果）



エラトステネスの篩

素数を見つける方法（アルゴリズム）

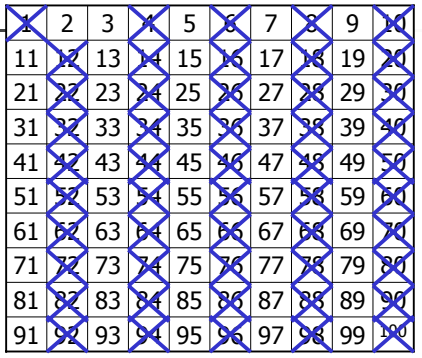
エラトステネスの篩

- 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

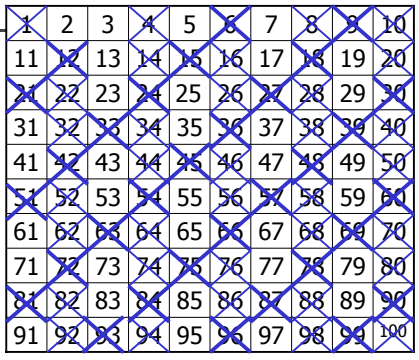
1から100までの配列を用意

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

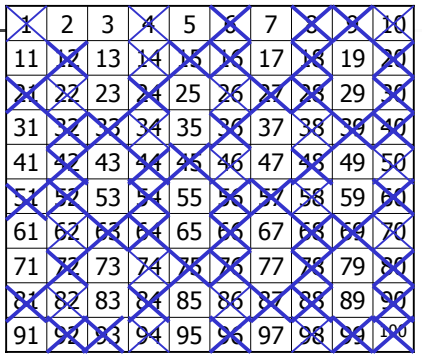
1を削除（0とする） 要素 2×2から2要素ごとに削除



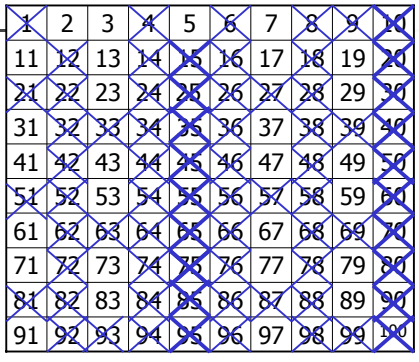
要素 3×2から3要素ごとに削除



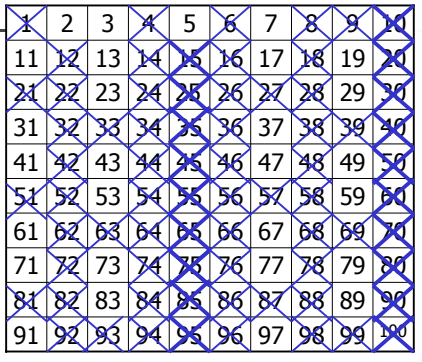
（無駄ですが…）要素 4×2から4要素ごとに削除



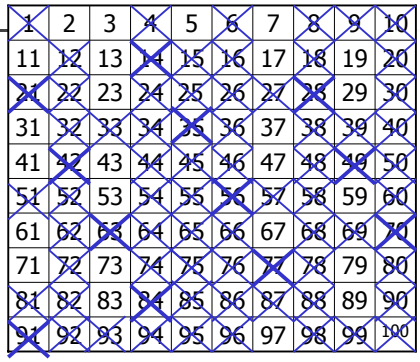
要素 5×2から5要素ごとに削除



（無駄ですが…）要素 6×2から6要素ごとに削除

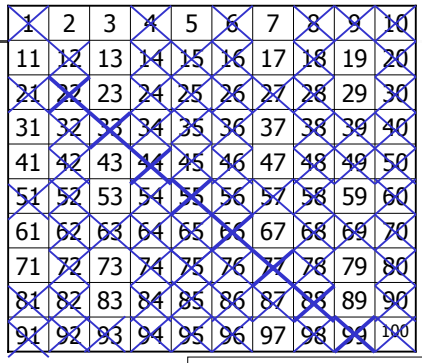


要素 7×2から7要素ごとに削除



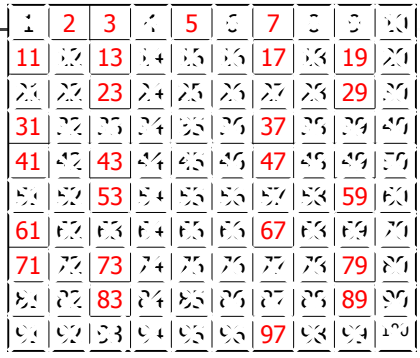
8,9,10の場合も同様に

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



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，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？

配列の初期化①

```
sieve = Array.new(10).fill{ 1 }
```

```
irb(main):004:0> sieve = Array.new(10).fill{ 1 }
=> [1, 1, 1, 1, 1, 1, 1, 1, 1, 1]
irb(main):005:0> p sieve
[1, 1, 1, 1, 1, 1, 1, 1, 1, 1]
=> nil
```

配列名=Array.new(要素数).fill{値}
配列の全要素は「値」となる

配列の初期化②

```
sieve = Array.new(10).fill{ |i| i }
```

```
irb(main):001:0> sieve = Array.new(10).fill{ |i| i }
=> [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

短く書けます

```
sieve = Array.new(10)

sieve.length.times{ |i|
        sieve[ i ] = i
}
```

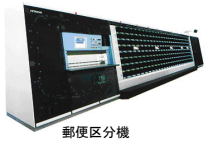

ソーティング

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_654.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

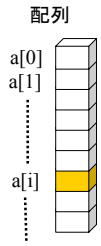
↓ ソートする(昇順)

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

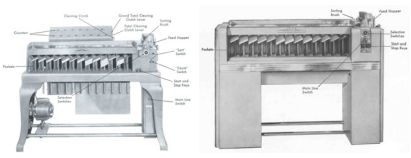
ソーティングのアルゴリズム

- 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

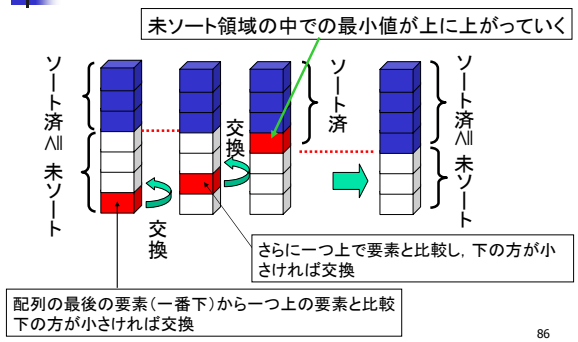
- 実は、区分機を使うと、昇順なり降順に並べることができる
  - 「できる」だけではなく、実際にそうしていた
- たとえば、10進3桁のID番号が振られているカードがあったとしよう
  - 一の位で、0,1,,,9に区分けする
    - 0～9の順に重ねると、下一桁では、0～9の順に並んでいる
  - その順序を崩さずに、十の位で、0,1,,,9に区分けする
    - 0 ～9の順に重ねると、下二桁では、00～99の順に並んでいる
  - その順序を崩さずに、百の位で、0,1,,,9に区分けする
    - 0 ～9の順に重ねると、下三桁では、000～999の順に並んでいる



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

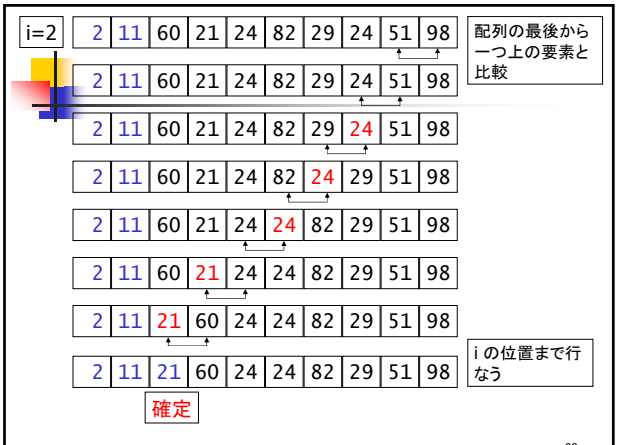
バブルソートのアルゴリズム①

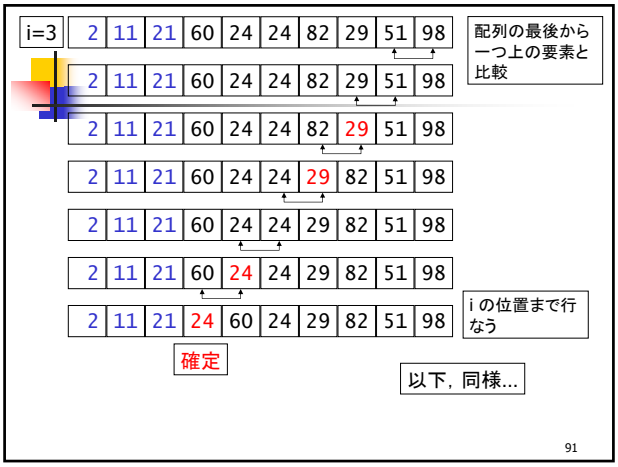


バブルソートの具体例

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



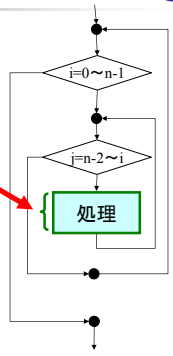


バブルソートのアルゴリズム②

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



補足: 入れ替え

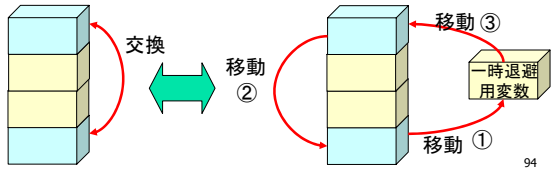
- 変数値の入替え
  - 変数 a と変数 b に入っている値を入替えたい。どうすればいいか？
    - 当然、a=b; b=a ではだめです。どうして？

```
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
```


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
```



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]
```


他のソーティングアルゴリズム

- 秋学期で学びます
  - アルゴリズム論
  - 理工学基礎実験(D実験)
- 3年生の春学期でも学びます
  - 管理工学実験・演習II(COM実験)

練習問題

練習問題①～④

練習問題①

- 下記のプログラムにおいて，配列 test の要素値の標準偏差を求めるプログラムを追加しなさい

```
name = [ "A" , "B" , "C" , "D" , "E" ]
test = [ 85 , 60 , 5, 100 , 50 ]
sum = 0
test.each{ |i|
   sum += i
}
average = sum / test.length
```

標準偏差

$$s = \sqrt{\frac{\sum_{i=1}^{n}(x_i - a)^2}{n-1}}$$

練習問題②

- x, y ともに0から10までの整数とする．この場合，

① xとyの和が10となる組み合わせ

② x^2とy^2の和が100となる組み合わせ

を二重ループを用いて求めなさい

練習問題③

- 下記のような出力を行なうプログラムを二重ループを用いて書きなさい

9行9列

```
C:¥Ruby>ruby sample.rb
1 0 0 0 0 0 0 0 1
0 1 0 0 0 0 0 1 0
0 0 1 0 0 0 1 0 0
0 0 0 1 0 1 0 0 0
0 0 0 0 1 0 0 0 0
0 0 0 1 0 1 0 0 0
0 0 1 0 0 0 1 0 0
0 1 0 0 0 0 0 1 0
1 0 0 0 0 0 0 0 1
```


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。