Ruby で実装するエラトステネスのふるい

小波秀雄

1　エラトステネスのふるい

自然数の集合から素数 (prime number) を選び出すためのアルゴリズム。ここでのねらいは，素朴な実装からスタートして Ruby らしく書いてみたり，アルゴリズムを工夫してみること — 速さの追求もさることながら，どんな道具が使えるか遊んでみます。

1.1　アルゴリズム

N までの素数を選び出すことを考えます (図 1)。

1. 集合 $A = \{2, 3, \ldots, N\}$ を用意する。
2. 最初の数 2 を選び，その倍数を A から削除する。
3. A の残りから 3 を選び，その倍数を A から削除する。
4. 削除すべき数がなくなったら終了。

素朴な実装

```
# erato_01.rb
def erato(n)
  numbers = (2..n).to_a             # 2 〜 n の配列を用意
  primes = []                        # 素数を放り込むための配列
  while true
    d = numbers.shift                # 配列の先頭要素を取り出して
    break if !d                      #   nil なら終了
    primes << d                      #   素数の配列に追加。
    to_delete = []                   # 取り除く候補を入れる配列
    numbers.each do |x|
      to_delete << x if x % d == 0   # d の倍数を取り除く候補に入れる
    end
    numbers -= to_delete             # d の倍数がそっくり引かれる
  end
  return primes
end
```

使ってみた技

- `d = numbers.shift; break if !d`
  `numbers` が空になっていたらループを脱出 (→ 2.3 節)
- `numbers -= to_delete`
  配列と配列の引き算

ベンチマーク

```
--------------
204.000000  91.330000 295.330000 (298.391229)
      user     system      total        real
n = 100000, 9592 primes, program:'erato_01.rb'
```

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~~ 101

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~~ 100 101

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~~ 101

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~~ 99 100 101

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~~ 100 101

図１　エラトステネスのふるいの進行のようす

2 実装の改善のために

2.1 ベンチマークテストのためのライブラリを活用しよう

実行時間の計測はアルゴリズム評価の基本。Ruby にはベンチマークテストのためのライブラリが添付されています。次のプログラムは Benchmark ライブラリの使用例で，もっとも簡単な使い方を示しています。

```
require 'benchmark'
puts Benchmark.measure{
  ar1 = (0...100000).to_a
  ar1.map!{|x| x * 2}
  ar2 = []
  (0...10000).step(5){|e| ar2 << e}
}
puts Benchmark::CAPTION
```

実行すると，下のようにレポートが出力されます。

```
> ruby bench.rb
  0.640000   0.300000   0.940000 (  0.935762)
      user     system      total        real
```

2.2 プロファイラで実行時間を分析しよう

プログラムが実行されるとき，どの部分に時間が使われているかを分析する道具がプロファイラです。Ruby では実行時の引数でプロファイラを呼び出すことができます。

```
ruby hoge.rb -r profile
```

これによってメソッドごとの実行時間の詳細なレポートが出力されます。

```
  %   cumulative   self              self     total
 time   seconds   seconds    calls  ms/call  ms/call  name
 45.61     0.26      0.26        1   260.00   360.00  Range#step
 22.81     0.39      0.13        1   130.00   190.00  Array#map!
 17.54     0.49      0.10     2000     0.05     0.05  Array#<<
 10.53     0.55      0.06     1000     0.06     0.06  Fixnum#*
```

評価のためのテンプレート

次のソースを用意しておいて，エラトステネスのふるいのソースを読み込んで走らせることにします。このまま走らせると，プロファイラは読み込まれず，ベンチマークテストが実行されます。

先頭行の -r の前の # を消すと，そのときにはベンチマークテストを実行せず（しても意味ない），プロファイラによる解析が行われます。

```
#!/usr/local/bin/ruby  # -r profile
require 'erato_01.rb'
program =  $LOADED_FEATURES.last
n = 300
w = []
if defined?(Profiler__) then
  w = erato(n)
else
  puts "--------------"
  require 'benchmark'
  puts Benchmark.measure{
    w = erato(n)
  }
  puts Benchmark::CAPTION
end
puts "n = #{n}, #{w.size} primes, program:'#{program}'"
```

2.3 活用できそうな Ruby の仕様

Ruby は配列 (Array) や範囲 (Range) クラスのメソッドとして豊富な仕掛けを用意しているので，それらを実装に使ってみましょう。

Array,Range まわりの定番のメソッド

irb で試してみましょう。

```
(5..10).to_a   #=> [5,6,7,8,9,10]      範囲オブジェクトから配列を生成
ar = [2,3,4,5,6]
ar.shift       #=> 2                   先頭の要素を取り出す
```

```
ar              #=> [3,4,5,6]           先頭の要素は削除されている
[].shift        #=> nil                 空の配列から取り出すと nil が返る
ar << 7         #=> [3,4,5,6,7]         末尾に要素を追加
ar.push(9)      #=> [3,4,5,6,7,9]       << と push は同じ動作
ar.pop          #=> 9                   末尾の要素を取り出す
ar              #=> [3,4,5,6,7]         末尾の要素は削除されている
(2..7).step(2){|e| puts e} #=> 2, 4, 6  飛び飛びに要素を拾って行く
ar = [1,2,3,2,3,4,2,3,4]
ar.delete(2)    #=> [1,3,3,4,3,4]       ある値をもつ要素だけを削除する
```

配列の集合算

```
irb(main):001:0> a = [1,2,3,4,5,6,7,8,9,10]
=> [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
irb(main):002:0> b = [2,4,6,8,10,12]
=> [2, 4, 6, 8, 10, 12]
irb(main):003:0> a - b
=> [1, 3, 5, 7, 9]
irb(main):004:0> c = (8..13).to_a
=> [8, 9, 10, 11, 12, 13]
irb(main):005:0> a & c
=> [8, 9, 10]
irb(main):006:0> a | c
=> [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13]
irb(main):007:0> a + c
=> [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 8, 9, 10, 11, 12, 13]
```

3 実装例

3.1 最初の実装への突っ込みどころ

- while true って C っぽくね？
  → 無限ループは loop を使ってみようか。

```
loop do
   puts "Stop ME!"
end
```

- if !d って気持ち悪い！
  unless を使おう。
- numbers.each do |x|; to_delete << x if x % d == 0; end ←倍数を引くのにいちいちこれはどう？
  プロファイラでチェック！

```
n = 1000, 168 primes, program:'erato_01.rb'
  %   cumulative   self              self     total
 time   seconds   seconds    calls  ms/call  ms/call  name
 70.29     4.07      4.07      168    24.23    33.45  Array#each
 13.13     4.83      0.76    15620     0.05     0.05  Fixnum#==
 12.78     5.57      0.74    15620     0.05     0.05  Fixnum#%
  2.59     5.72      0.15        1   150.00  5790.00  Object#erato
  0.86     5.77      0.05      999     0.05     0.05  Array#<<
```

  比較と剰余計算でそこそこ時間を食っているらしいぞ。
  → step でループを作ってみようか[*1]。
- to_delete に要素を放り込む手順がまずい
  → to_delete を作るのに，連続する整数についていちいち倍数であるかどうかをチェックして配列に加えるのではなく，倍数を単純に作って配列に加えればいいじゃん！
- primes は無駄じゃないかい。
  元の集合から合成数を引き去った集合が素数の集合になるのだから，わざわざ numbers の先頭の数を primes に移すなんてことはしないほうが速い。

[*1] 高速計算のための一般論としては，ループの中で if を使うことは，最適化を妨げる要因になります (Ruby における最適化がどうなっているかは知りませんが)。また回数の決まったループと条件判断を伴うループでは，後者の方が実行が遅くなりがちです。しかし，範囲オブジェクトの生成や each メソッドの実行にはかなりの時間が使われるようですので，高速化を図ろうとしたら，もっとこまかいチェックが必要です。

改良版その 1

```
# erato_02.rb
def erato(n)
  numbers = (2..n).to_a
  i = 0
  loop do
    break unless d = numbers[i]
    to_delete = []
    (2*d..n).step(d){|e| to_delete << e}
    numbers -= to_delete
    i += 1
  end
  return numbers
end
```

ベンチマーク

```
--------------
  6.910000   1.520000   8.430000 (  8.476773)
      user     system      total        real
n = 100000, 9592 primes, program:'erato_02.rb'
```

一気に改善しました。

3.2 アルゴリズムを見直す

処理の進行にともなって残りの集合がどうなっているかを見てみましょう。図 1 をみると，7 の倍数を消してしまった時点で，すでに素数しか残っていません。

ためしに erato_02.rb にデバッグラインを入れて，配列 numbers がどのように変化しているかを調べてみましょう。下は n = 23 としたときの出力です。

```
--------------
d = 2
[2, 3, 5, 7, 9, 11, 13, 15, 17, 19, 21]
d = 3
[2, 3, 5, 7, 11, 13, 17, 19]
d = 5
[2, 3, 5, 7, 11, 13, 17, 19]
d = 7
[2, 3, 5, 7, 11, 13, 17, 19]
  ...
```

d が 3 の時に numbers の要素は素数だけになり，以後は変化していません。一般には，N までの素数を求めるためには $\lfloor\sqrt{N}\rfloor$ の倍数以下の数をチェックすればよいことが，簡単な考察からわかります[*2]。$\lfloor\sqrt{N}\rfloor$ に相当する Ruby のメソッドは floor があります。

改良版その 2

```
# erato_03.rb
def erato(n)
  sqrtN = Math.sqrt(n).floor
  numbers = (2...n).to_a
  i = 0
  while i <= sqrtN
    d = numbers[i]
    to_delete = []
    (2*d..n).step(d){|e| to_delete << e}
    numbers -= to_delete
    i += 1
  end
  return numbers
end
```

ベンチマーク

```
--------------
  1.490000   0.590000   2.080000 (  2.092377)
      user     system      total        real
n = 100000, 9592 primes, program:'erato_03.rb'
```

効いてます。

この書き換えは？

下のように書き換えてみたらどうでしょうか。

```
#  (2*d..n).step(d){|e| to_delete << e}
#  numbers -= to_delete
   (2*d..n).step(d){|e| numbers.delete(e)}
```

2 行が 1 行に減ったのに，悲惨なことにプログラムは行ってしまったきりです。

[*2] $\lfloor x\rfloor$ というのは x と等しいかそれより小さい整数のうち最大のものを意味します。たとえば $N = 23$ であれば $\lfloor\sqrt{N}\rfloor = \lfloor 4.796\rfloor = 4$ となります。

3.3 配列まわりの処理の速度を調べてみる

次の 2 つの処理の速度を比較してみましょう。どちらも配列の要素を 1 個おきに削除しています。

```
ar = (0..N).to_a
#(1)
for i in 0 ... N/2
  ar.delete(i*2)
end
#(2)
(0..N).step(2){|e| ar[e] = nil}
ar.delete(nil)
```

図 2 の結果を見ると (1) は $O(N^2)$ で (2) は $O(N)$，しかも $N = 1000$ で，すでに (2) が圧勝してます。これは delete メソッドが配列のサイズを変更してしまうことからくる宿命です。配列から要素を取り除くときには，配列全体にわたるデータ移動が必要になるからです。

したがって，まず取り除くべき要素を nil など（既存の要素と重ならなければなんでもいいのですが）に置き換えておいて，その後 1 度だけ delete メソッドを使う方が格段に速いのです。

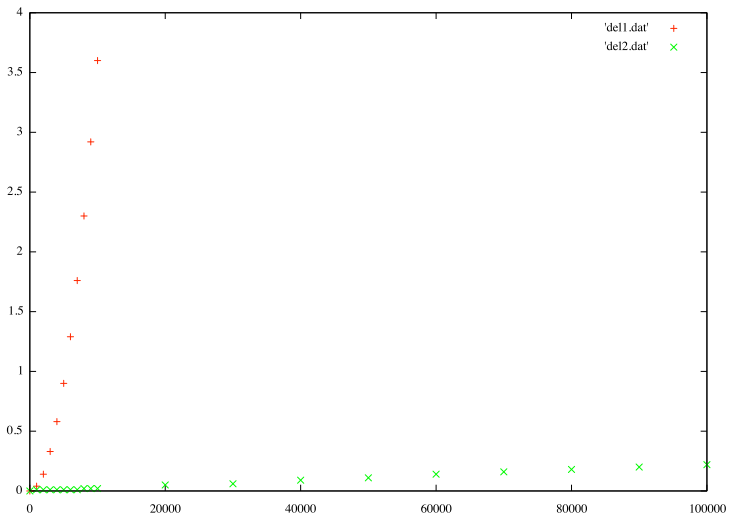

図 2　(1) +，(2) ×の実行時間と配列のサイズの関係

改良版その 3

delete メソッドを活用して，合成数を一気に引き去る発想で書かれたソースです。

```
# erato_04.rb
def erato(n)
  numbers = (0..n).to_a
  numbers[0],numbers[1] = nil
  for d in numbers
    next unless(d)
    break if (Math::sqrt(n) < d)
    (2*d..n).step(d){|e| numbers[e] = nil}
  end
  numbers.delete(nil)
  return numbers
end
```

最初に次のような配列 numbers を用意しています。

```
[nil,nil,2,3,4,...,n]
```

これを頭から nil を除いて読んでいき，2 の倍数（2 を除く）を nil に，3 の倍数（3 を除く）を nil に, 5 の倍数（5 を除く）を nil に,... と進んで行きます。

最後に numbers から nil を一気に削除します。

ベンチマーク

```
--------------
  1.070000   0.500000   1.570000 (  1.596298)
      user     system      total        real
n = 100000, 9592 primes, program:'erato_04.rb'
```

大成功！

まったく違う発想で

最後は lambda を使って，手続きを再帰的に実行するクールなやり方です。

```
# erato_05.rb
def erato(n)
  sieve = lambda {|f, primes, d, max|
    break primes if d > max
    f.call(f, primes.select {|s| (s==d) || (s%d != 0)}, d+1, max)
  }
  sieve.call(sieve, (2..n), 2, Math::sqrt(n))
end
```

このソースの説明はとても長くなるので省略。自分で勉強してください。この手法は普通のプログラミング言語にはあまりないものです。

ベンチマーク

```
--------------
 17.700000   7.620000  25.320000 ( 25.483979)
      user     system      total        real
n = 100000, 9592 primes, program:'erato_05.rb'
```

残念ながら速くはありませんでした。