Benchmarkuj z głową

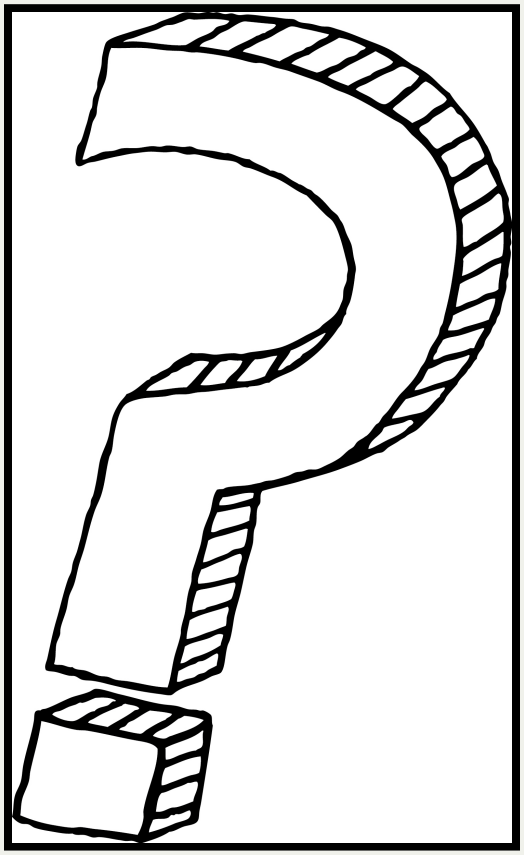
Radosław Bułat

KRUG, styczeń 2016r.

Disclaimer

Wszystkie testy zostały przeprowadzone na komputerze Macbook Pro z systemem OS X El Capitan, wyposażonym w procesor Intel Core i5 2,6 Ghz, 8GB Ram i dysk SSD. Programy, jeśli nie wyspecyfikowano inaczej, były odpalane przy użyciu interpretera rubiego MRI w wersji 2.3.0p0 (64bit).

Co to są benchmarki?

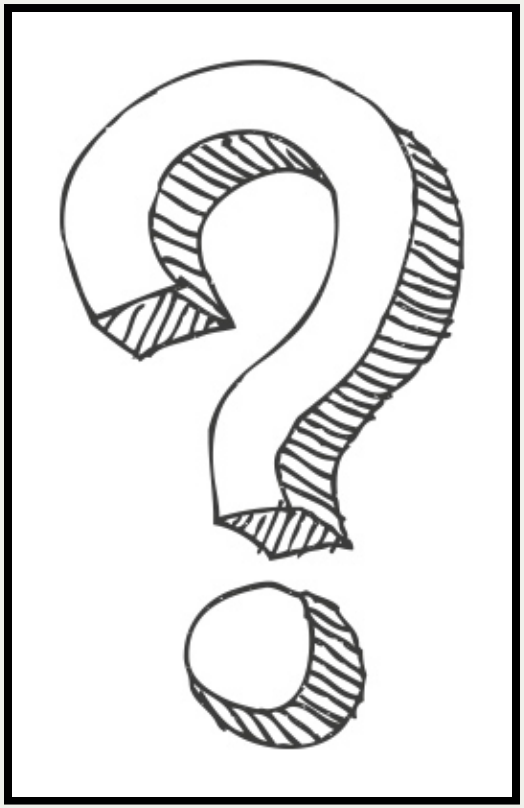


Testy przeprowadzone w celu zmierzenia wydajności programu.

Zwykle miarą jest czas, ale może to być także np. zużycie pamięci, ilości operacji I/O itp.

Testować można na różnych poziomie abstrakcji. Ta prezentacja dotyczy głównie mikro-benchmarków.

Po co benchmarkać?



Możesz nauczyć się przeprowadzać eksperymenty i poprawnie wyciągać wnioski.

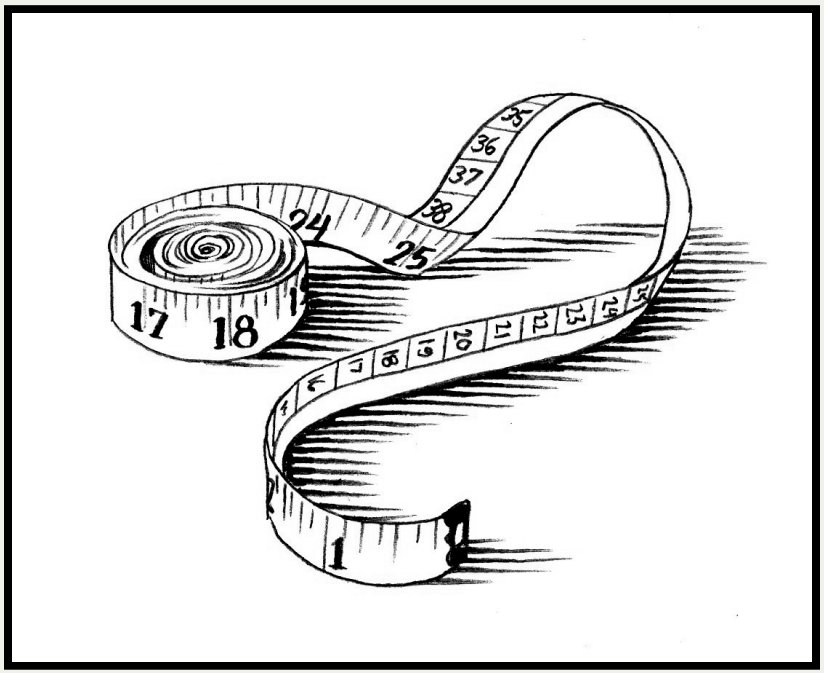
Poznasz lepiej kod, który testujesz (własny, gema, frameworka).

Poznasz dogłębniej język Ruby i/lub konkretną implementację (MRI w naszym przypadku).

Nauczysz się algorytmów, technik optymalizacyjnych i innych sztuczek.


Dla zabawy!

Jak poprawnie mierzyć?



Czy to jest dobra metoda?

```
$ time ruby my_script.rb
```

Tak, jeśli mierzymy faktycznie wykonanie **całego** skryptu (bootstrap VM, ładowanie gemów itp.).

Zwykle mierzymy jednak tylko **wycinek** kodu, więc ta metoda się nie sprawdza.

Manualnie

```
start = Time.now
N     = 100_000
N.times { "foo-bar-baz".gsub(/-/, ".") }
stop  = Time.now

puts "String#gsub: #{stop - start}s"
```

```
String#gsub: 0.223087s
```

Ok, ale komu chciałoby się pisać taki kod za każdym razem?

biblioteka benchmark (stdlib)

```
require "benchmark"

N = 100_000

Benchmark.bm do |x|
  x.report("String#gsub") do
    N.times { "foo-bar-baz".gsub(/-/, ".") }
  end

  x.report("String#tr") do
    N.times { "foo-bar-baz".tr("-", ".") }
  end
end
```

```
                    user     system      total        real
String#gsub  0.220000   0.000000   0.220000 (  0.218242)
String#tr    0.030000   0.000000   0.030000 (  0.031729)
```

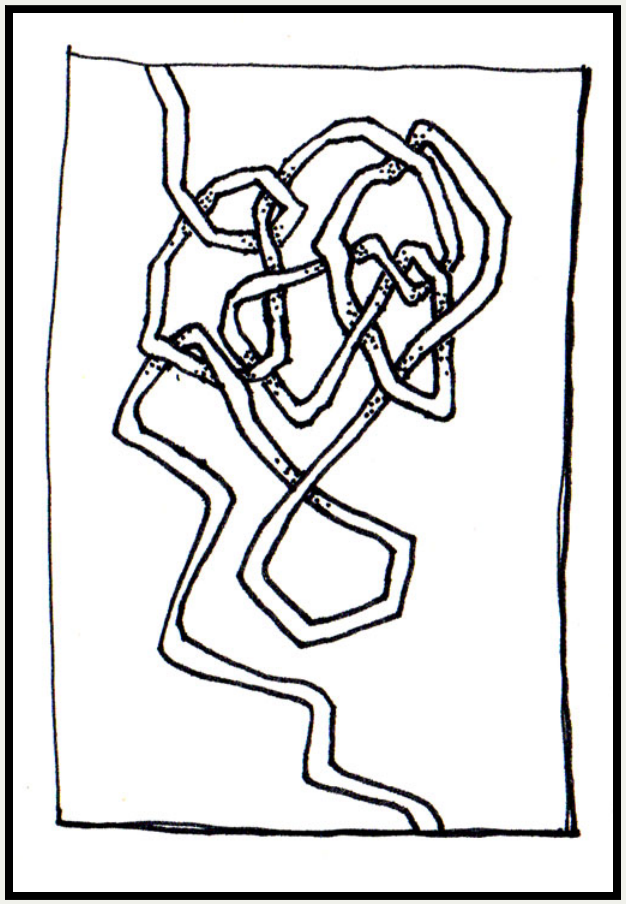
Co z wyznaczaniem odpowiedniej wartości N?

gem benchmark-ips

```
require "benchmark/ips"

Benchmark.ips do |x|
  x.report("String#gsub") do
    "foo-bar-baz".gsub(/-/, ".")
  end

  x.report("String#tr") do
    "foo-bar-baz".tr("-", ".")
  end

  x.compare!
end
```

```
Calculating -------------------------------------
          String#gsub    33.159k i/100ms
            String#tr   103.818k i/100ms
-------------------------------------------------
          String#gsub    436.803k (± 6.5%) i/s -      2.188M
            String#tr      2.732M (± 7.5%) i/s -     13.600M

Comparison:
            String#tr:  2731656.6 i/s
          String#gsub:   436802.9 i/s - 6.25x slower
```

A jest x razy wolniejsze od B. To nie zawsze takie proste.



```ruby
require "benchmark/ips"

N   = ARGV[0].to_i
arr = (1..N).to_a

Benchmark.ips do |x|
  x.report("Array#include?") do
    arr.include?(rand(arr.size))
  end

  x.report("Array#bsearch") do
    v = rand(arr.size)
    arr.bsearch { |e| e >= v }
  end

  x.compare!
end
```

$ ruby include-vs-bsearch.rb 10_000

```
Array#bsearch:   957959.3 i/s
Array#include?:    33271.6 i/s - 28.79x slower
```

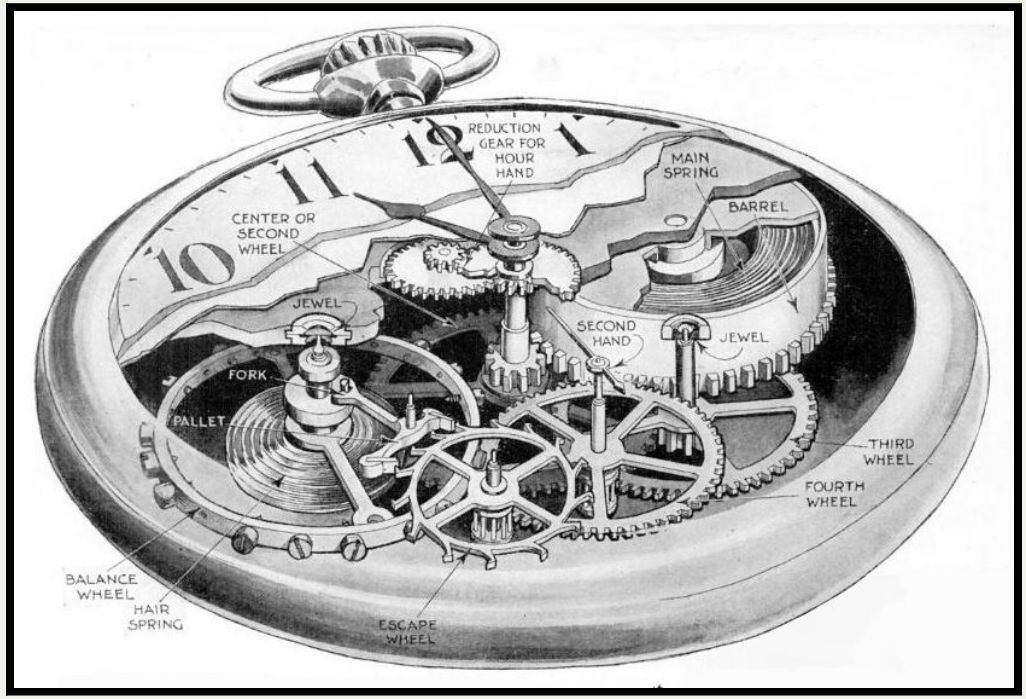
$ ruby include-vs-bsearch.rb 1_000

```
Array#bsearch:  1165847.6 i/s
Array#include?:   299444.3 i/s - 3.89x slower
```

28.79x slower vs 3.89x slower

Dlaczego?

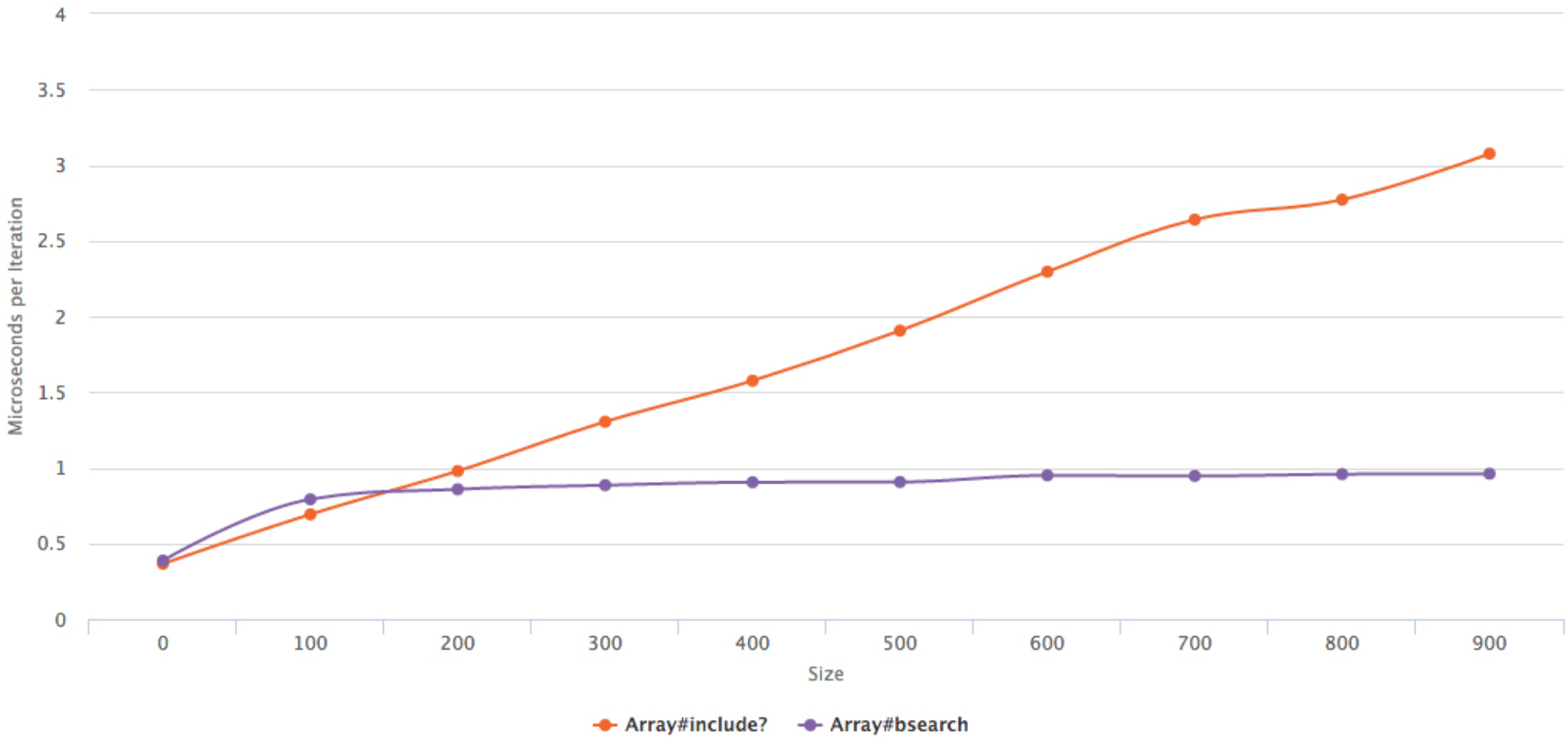
Złożoność czasowa



gem benchmark-bigo

```
require "benchmark/bigo"

Benchmark.bigo do |x|
  x.min_size = 0
  x.generator { |size| (1..size).to_a }

  x.report("Array#include?") do |arr, size|
    arr.include?(rand(size))
  end

  x.report("Array#bsearch") do |arr, size|
    v = rand(size)
    arr.bsearch { |e| e >= v }
  end

  x.chart!
end
```

Growth Chart

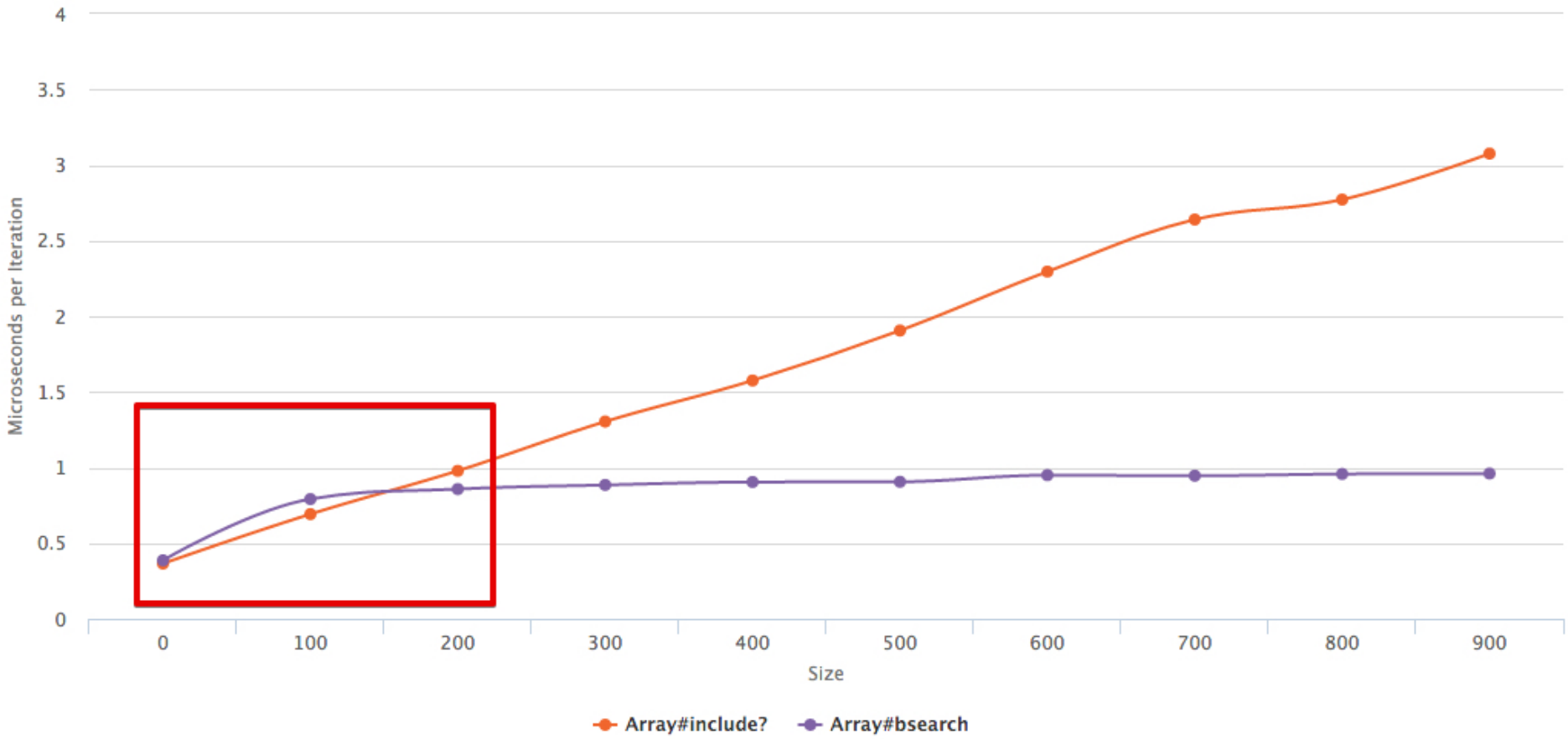


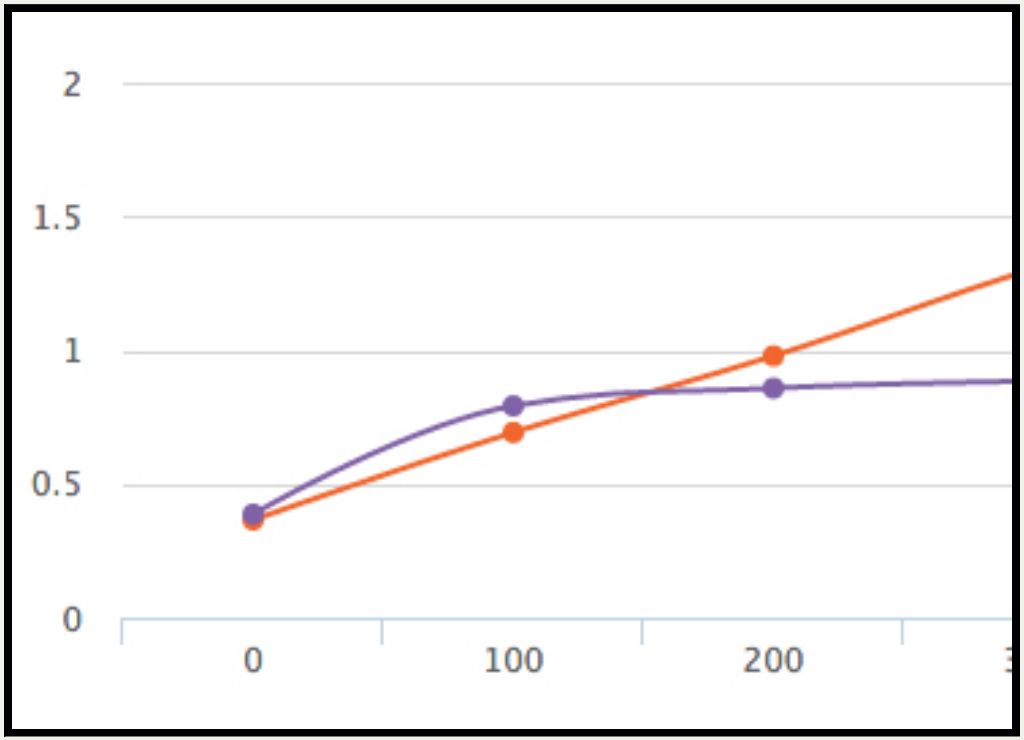
Array#include? - O(n)

Array#bsearch - O(log n)

Metoda bsearch ma mniejszą złożoność. Czy zatem powinniśmy zawsze ją preferować?

Growth Chart




2
1.5
1
0.5
0
0
100
200

$ ruby include-vs-bsearch.rb 50

```
Array#include?:  2443856.0 i/s
Array#bsearch:  1627241.2 i/s - 1.50x slower
```

$ ruby include-vs-bsearch.rb 150

```
Array#bsearch:  1395988.6 i/s
Array#include?:  1389880.7 i/s - 1.00x slower
```

Dlaczego?

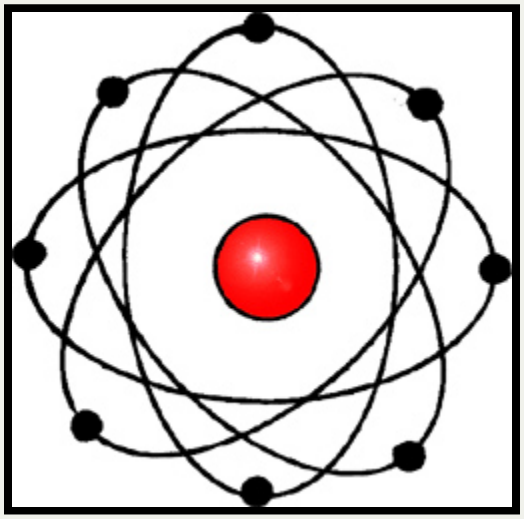
```
arr.include?(v)
```

Przekazuje sterowanie do implementacji w C. Metoda Fixnum#== nie jest wywoływana, chyba, że zostanie nadpisana.

```
arr.bsearch { |e| e >= v }
```

Przekazuje sterowanie do implementacji w C, ale każde porównanie to koszt wywołania i wykonania bloku, który jest zaimplementowany w czystym Rubym.

Jak zmierzyć atom linijką?



Problem: porównać wydajność operacji Fixnum#+ i Fixnum#*

```
require "benchmark/ips"

Benchmark.ips do |x|
  x.report("Fixnum#+") { 2+3 }
  x.report("Fixnum#*") { 2*3 }
  x.compare!
end
```

```
Fixnum#+: 10701978.3 i/s
Fixnum#*:  9650968.6 i/s - 1.11x slower
```

```
require "benchmark/ips"

Benchmark.ips do |x|
  x.report("Fixnum#+") { 2+3; 2+3; 2+3; 2+3; 2+3; 2+3; 2+3; 2+3; 2+3; 2+3; }
  x.report("Fixnum#*") { 2*3; 2*3; 2*3; 2*3; 2*3; 2*3; 2*3; 2*3; 2*3; 2*3; }
  x.compare!
end
```

```
Fixnum#+:  5481473.1 i/s
Fixnum#*:  3938857.4 i/s - 1.39x slower
```

Zwielokrotnienie mierzonej operacji w bloku powoduje zniwelowanie jakiegoś narzutu.

Co jest tym narzutem?

Blok opakowujący benchmark + pętla, która go wykonuje!

```
require "benchmark/ips"

Benchmark.ips do |x|
  x.report("empty block") { }
  x.report("Fixnum#+") { 2+3 }
  x.report("Fixnum#*") { 2*3 }
  x.compare!
end
```

```
empty block: 11844492.5 i/s
   Fixnum#+: 11076597.5 i/s - 1.07x slower
   Fixnum#*:  9812807.9 i/s - 1.21x slower
```

Jak zniwelować ten narzut?

Zamiast bloku przekazać odpowiednio skonstruowany string, który zostanie przed testem skompilowany.

```
require "benchmark/ips"

Benchmark.ips do |x|
  x.report("Fixnum#+", "2+3;" * 10_000)
  x.report("Fixnum#*", "2*3;" * 10_000)
  x.compare!
end
```

```
Fixnum#+:      9538.4 i/s
Fixnum#*:      6099.6 i/s - 1.56x slower
```

Niektóre przekonania są mitami.

```
require "benchmark/ips"

Benchmark.ips do |x|
  x.report("downcase",  '"FOOBAR".downcase;'  * 1_000)
  x.report("downcase!", '"FOOBAR".downcase!;' * 1_000)
  x.compare!
end
```

```
downcase!:    16357.6 i/s
 downcase:    11034.8 i/s - 1.48x slower
```

Czy metody modyfikujące obiekt w miejscu są zawsze szybsze?

Nie zawsze!

```
require "benchmark/ips"

Benchmark.ips do |x|
  x.report("gsub",  '"foobarbaz".gsub(/./, "z");'  * 10_000)
  x.report("gsub!", '"foobarbaz".gsub!(/./, "z");' * 10_000)
  x.compare!
end
```

```
gsub!:        28.9 i/s
 gsub:        28.9 i/s - 1.00x slower
```

To może gsub! zużywa mniej pamięci niż gsub?

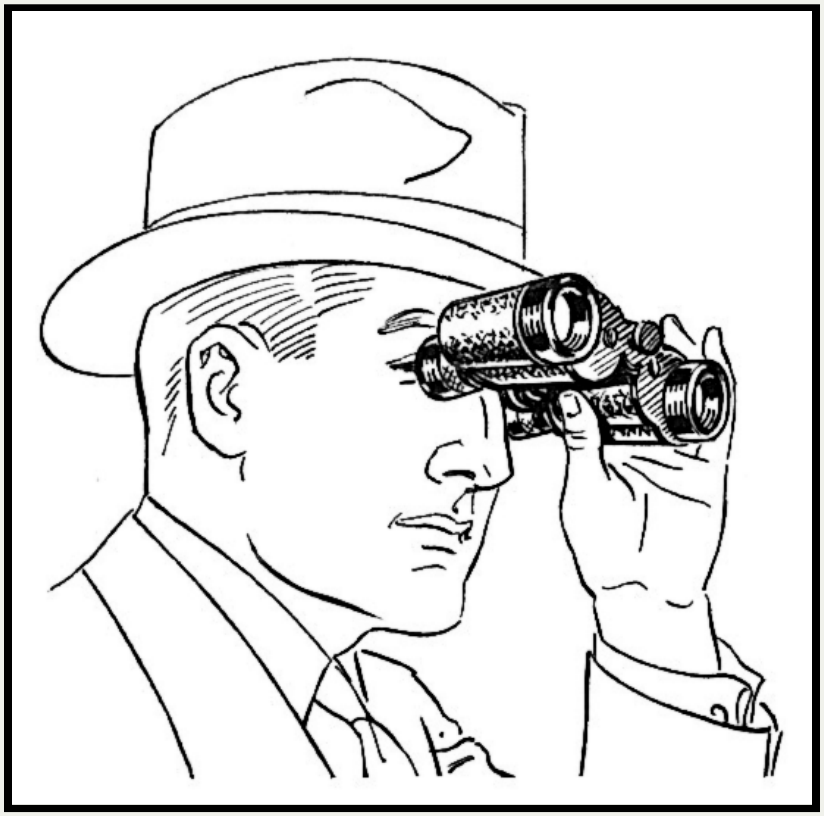
```ruby
require "objspace"

GC.disable

def memory_usage
  `ps -o rss= -p #{Process.pid}`.to_i / 1024
end

before = memory_usage
1_000_000.times { "foo-bar-baz".gsub(/-/, ".") }
after  = memory_usage

puts "Allocated memory: #{after - before}MB"
puts "Allocated strings: #{ObjectSpace.count_objects[:T_STRING]}"
```

```
Allocated memory: 696MB
Allocated strings: 5010147
```

```ruby
require "objspace"

GC.disable

def memory_usage
  `ps -o rss= -p #{Process.pid}`.to_i / 1024
end

before = memory_usage
1_000_000.times { "foo-bar-baz".gsub!(/-/, ".") }
after  = memory_usage

puts "Allocated memory: #{after - before}MB"
puts "Allocated strings: #{ObjectSpace.count_objects[:T_STRING]}"
```

```
Allocated memory: 695MB
Allocated strings: 5010147
```

Dociekliwość i spostrzegawczość



$$a = 1$$
$$b = 2$$
vs
$$a, b = 1, 2$$

Ktoś kiedyś poczynił taki o to benchmark

```ruby
require 'benchmark/ips'

def parallel
  a, b, c, d, e, f, g, h = 1, 2, 3, 4, 5, 6, 7, 8
end

def sequential
  a = 1
  b = 2
  c = 3
  d = 4
  e = 5
  f = 6
  g = 7
  h = 8
end

Benchmark.ips do |x|
  x.report('Parallel')   { parallel }
  x.report('Sequential') { sequential }
  x.compare!
end
```

```
Sequential:  6257646.4 i/s
  Parallel:  2770688.2 i/s - 2.26x slower
```

Ktoś inny zauważył, że przypisanie równoległe ma dodatkowy koszt utworzenia nowej tablicy, ale tylko gdy jest wykonywane jako ostatnia instrukcja w metodzie.

```
pry(main)> parallel
=> [1, 2, 3, 4, 5, 6, 7, 8]
pry(main)> sequential
=> 8
```

```ruby
require 'benchmark/ips'

def parallel
  a, b, c, d, e, f, g, h = 1, 2, 3, 4, 5, 6, 7, 8
  nil
end

def sequential
  a = 1
  b = 2
  c = 3
  d = 4
  e = 5
  f = 6
  g = 7
  h = 8
  nil
end

Benchmark.ips do |x|
  x.report('Parallel')   { parallel }
  x.report('Sequential') { sequential }
  x.compare!
end
```

```
  Parallel:  7317266.9 i/s
Sequential:  6149955.8 i/s - 1.19x slower
```

Jeśli faktycznie przypisanie równoległe miałoby być szybsze to pytajmy "dlaczego?". Bądźmy dociekliwi.

Powtórzmy benchmark za pomocą wersji ze stringiem.

```ruby
require 'benchmark/ips'

Benchmark.ips do |x|
  x.report('Parallel',
    'a, b, c, d, e, f, g, h = 1, 2, 3, 4, 5, 6, 7, 8;'         * 10_000)

  x.report('Sequential',
    'a = 1; b = 2; c = 3; d = 4; e = 5; f = 6; g = 7; h = 8;' * 10_000)

  x.compare!
end
```

```
Sequential:     3031.0 i/s
  Parallel:     3029.8 i/s - 1.00x slower
```

Znajdź różnicę

```
a = 1
b = 2
```

vs

```
a = 1; b = 2
```

؛

؟

```ruby
require 'benchmark/ips'

def parallel
  a, b, c, d, e, f, g, h = 1, 2, 3, 4, 5, 6, 7, 8
  nil
end

def sequential
  a = 1;
  b = 2;
  c = 3;
  d = 4;
  e = 5;
  f = 6;
  g = 7;
  h = 8;
  nil
end

Benchmark.ips do |x|
  x.report('Parallel')   { parallel }
  x.report('Sequential') { sequential }
  x.compare!
end
```

```
  Parallel:  7291129.5 i/s
Sequential:  6203390.0 i/s - 1.18x slower
```

Znajdź inną różnicę

```
a = 1
b = 2
```

vs

```
a = 1; b = 2
```

\n

?

```ruby
require 'benchmark/ips'

def parallel
  a, b, c, d, e, f, g, h = 1, 2, 3, 4, 5, 6, 7, 8
  nil
end

def sequential
  a = 1; b = 2; c = 3; d = 4; e = 5; f = 6; g = 7; h = 8;
  nil
end

Benchmark.ips do |x|
  x.report('Parallel')   { parallel }
  x.report('Sequential') { sequential }
  x.compare!
end
```

```
  Parallel:  7247107.0 i/s
Sequential:  7227897.0 i/s - 1.00x slower
```

WHAT?


WE NEED TO GO
DEEPER
troll-face.ru

```
puts RubyVM::InstructionSequence.compile("a = 1; b = 2;").disasm
```

```
0000 trace            1
0002 putobject_OP_INT2FIX_O_1_C_
0003 setlocal_OP__WC__0 3
0005 putobject        2
0007 dup
0008 setlocal_OP__WC__0 2
0010 leave
```

```
puts RubyVM::InstructionSequence.compile("a = 1;\nb = 2;").disasm
```

```
0000 trace                1
0002 putobject_OP_INT2FIX_O_1_C_
0003 setlocal_OP__WC__0 3
0005 trace                1
0007 putobject            2
0009 dup
0010 setlocal_OP__WC__0 2
0012 leave
```

diff

```
 trace             1
 putobject_OP_INT2FIX_O_1_C_
 setlocal_OP__WC__0 3
+trace             1
 putobject         2
 dup
 setlocal_OP__WC__0 2
 leave
```

trace 1

http://ruby-doc.org/core-2.3.0/TracePoint.html

```ruby
def foo
  a, b = 1, 2
end

def bar
  a = 1
  b = 2
end

TracePoint.new(:line) do |tp|
  p [tp.event, tp.lineno]
end.enable

foo
bar
```

```
[:line, 14]
[:line, 2]
[:line, 15]
[:line, 6]
[:line, 7]
```

disable_trace.rb

```
RubyVM::InstructionSequence.compile_option = {
  trace_instruction: false
}
```

$ ruby assignment.rb

```
  Parallel:  7295158.6 i/s
Sequential:  6285277.7 i/s - 1.16x slower
```

$ ruby -r./disable_trace assignment.rb

```
Sequential:  7915213.2 i/s
  Parallel:  7896697.9 i/s - 1.00x slower
```

Podsumowanie

Pytania?

Dziękuję!