Crunching numbers with Clojure

11 tips to boost your performance

Daniel Solano Gómez

Sattvik Software & Technology Resources, Ltd. Co.

Clojure/West 2012

Learning Clojure

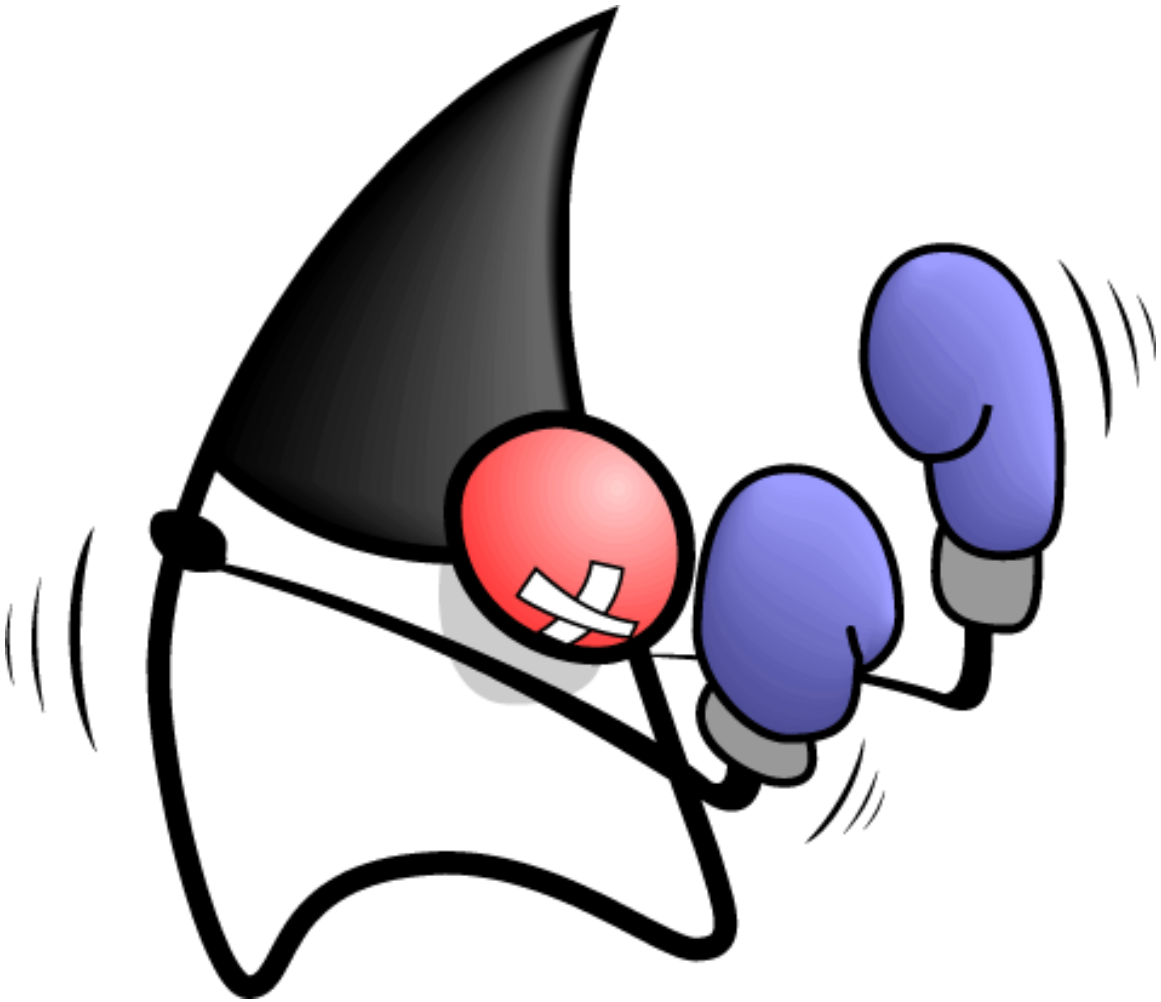
Project Euler.net

A platform for the inquiring mind to delve into unfamiliar areas and learn new concepts in a fun and recreational context.

A big problem

Image supplied by FreeFoto.com

A big problem



Sieve of Eratosthenes

0	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

Sieve of Eratosthenes

0	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

Sieve of Eratosthenes

0	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

Sieve of Eratosthenes

0	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

Sieve of Eratosthenes

0	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

Sieve of Eratosthenes

0	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

Sieve of Eratosthenes

0	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

Sieve of Eratosthenes

0	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

Sieve of Eratosthenes

0	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

Sieve of Eratosthenes

0	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

Sieve of Eratosthenes

0	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

Sieve of Eratosthenes

0	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

Sieve of Eratosthenes

0	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

Sieve of Eratosthenes

0	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

$$7 > \sqrt{30}$$

Sieve of Eratosthenes

0	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

$$7 > \sqrt{30}$$

A lazy implementation

A recursively-constructed lazy sequence

```
(def prime-seq
  ((fn inner [p primes]
    (lazy-seq
      (cons p
            (inner (next-prime (inc p)
                               primes)
                   (conj primes p)))))
   2 []))
```

A lazy implementation

Finding the next prime

```
(defn next-prime
  [n primes]
  (if (has-prime-factor? n primes)
    (recur (inc n) primes)
    n))
```

A lazy implementation

Trial by division

```
(defn has-prime-factor?
  [n primes]
  (some #(divides? n %) primes))
```

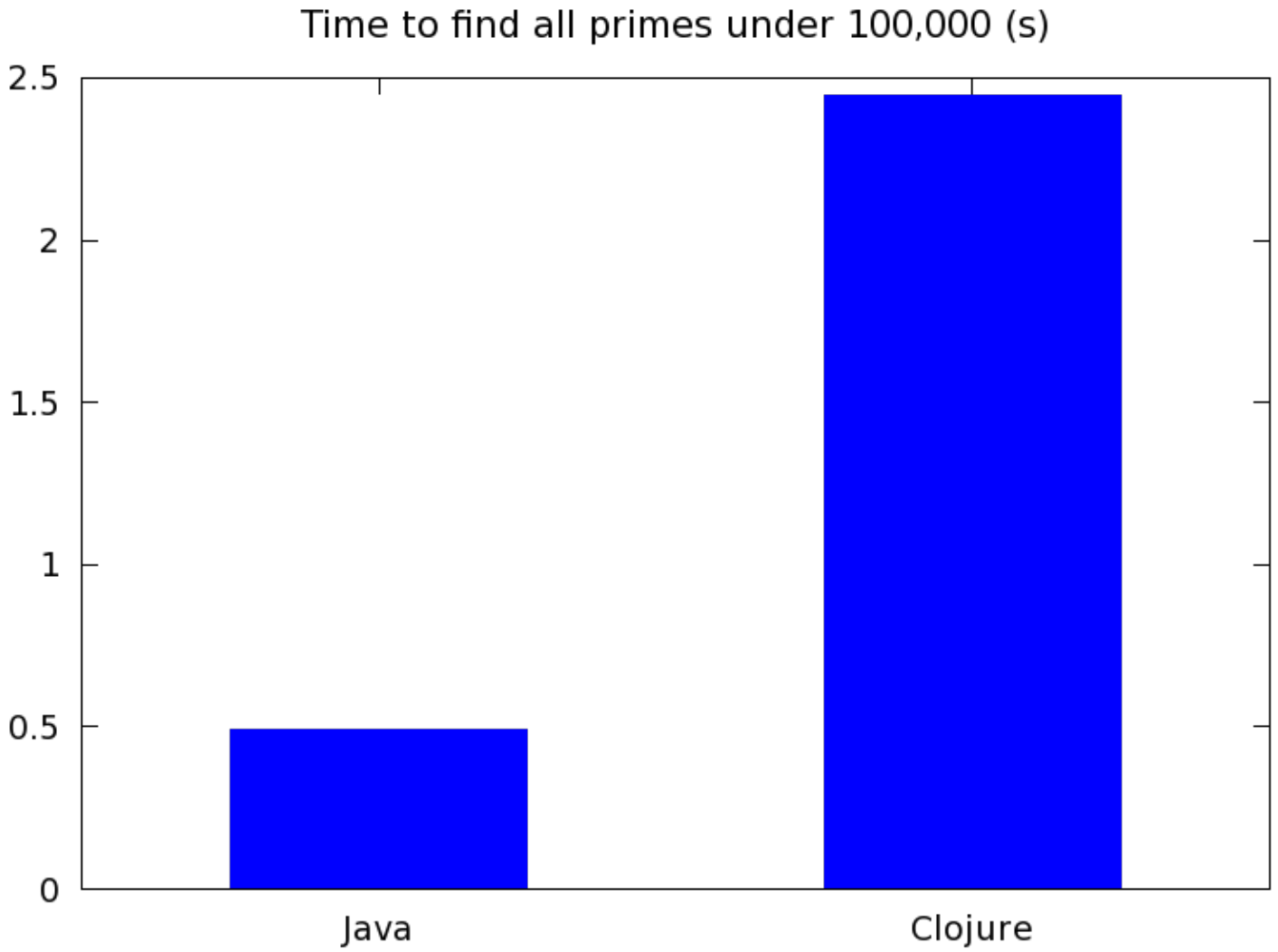
A lazy implementation

The final piece

```clojure
(defn divides?
  [n d]
  (zero? (rem n d)))
```

Performance

Clojure is nearly five times slower than Java



Motivation

Finding primes

Improving performance

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

Concluding thoughts

Top 10 hot spots

Basic lazy solution

Method/function	%
clojure.core/some	9
clojure.lang.Var.getRawRoot	8
clojure.lang.Numbers.ops	7
clojure.lang.Numbers.remainder	7
clojure.lang.Numbers.LongOps.isZero	5
java.lang.Number.longValue	5
clojure.lang.Numbers.LongOps.remainder	5
clojure.lang.Numbers.isZero	5
has-prime-factor?/fn	4
divides?	4
Total	59

Top 10 hot spots

Basic lazy solution

Method/function	%
clojure.core/some	9
clojure.lang.Var.getRawRoot	8
clojure.lang.Numbers.ops	7
clojure.lang.Numbers.remainder	7
clojure.lang.Numbers.LongOps.isZero	5
java.lang.Number.longValue	5
clojure.lang.Numbers.LongOps.remainder	5
clojure.lang.Numbers.isZero	5
has-prime-factor?/fn	4
divides?	4
Boxing-related	34

Primitive hinting

- Added in Clojure 1.3
- Can now hint function parameters and return values
- Supports long and double hints
- Only on functions with up to four arguments

Hinting divides?

```
(defn divides?
  [n d]
  (zero? (rem n d)))
```

Hinting divides?

```
(defn divides?
  [^long n ^long d]
  (zero? (rem n d)))
```

Performance

Primitive hints cut 15% of the time

Tip 1: Add primitive hints

- It’s essentially free.
- Breaks compatibility with Clojure 1.2 and earlier.

has-prime-factor?

Can you spot the problem?

```
(defn has-prime-factor?
  [^long n primes]
  (some #(divides? n %)
        primes))
```

has-prime-factor?

Stop when primes get too big

```
(defn has-prime-factor?
  [^long n primes]
  (some #(divides? n %)
        (take-while
          #(<= % (Math/sqrt n))
          primes)))
```

next-prime

Can you spot the problem?

```clojure
(defn next-prime
  [^long n primes]
  (if (has-prime-factor? n primes)
    (recur (inc n) primes)
    n))
```

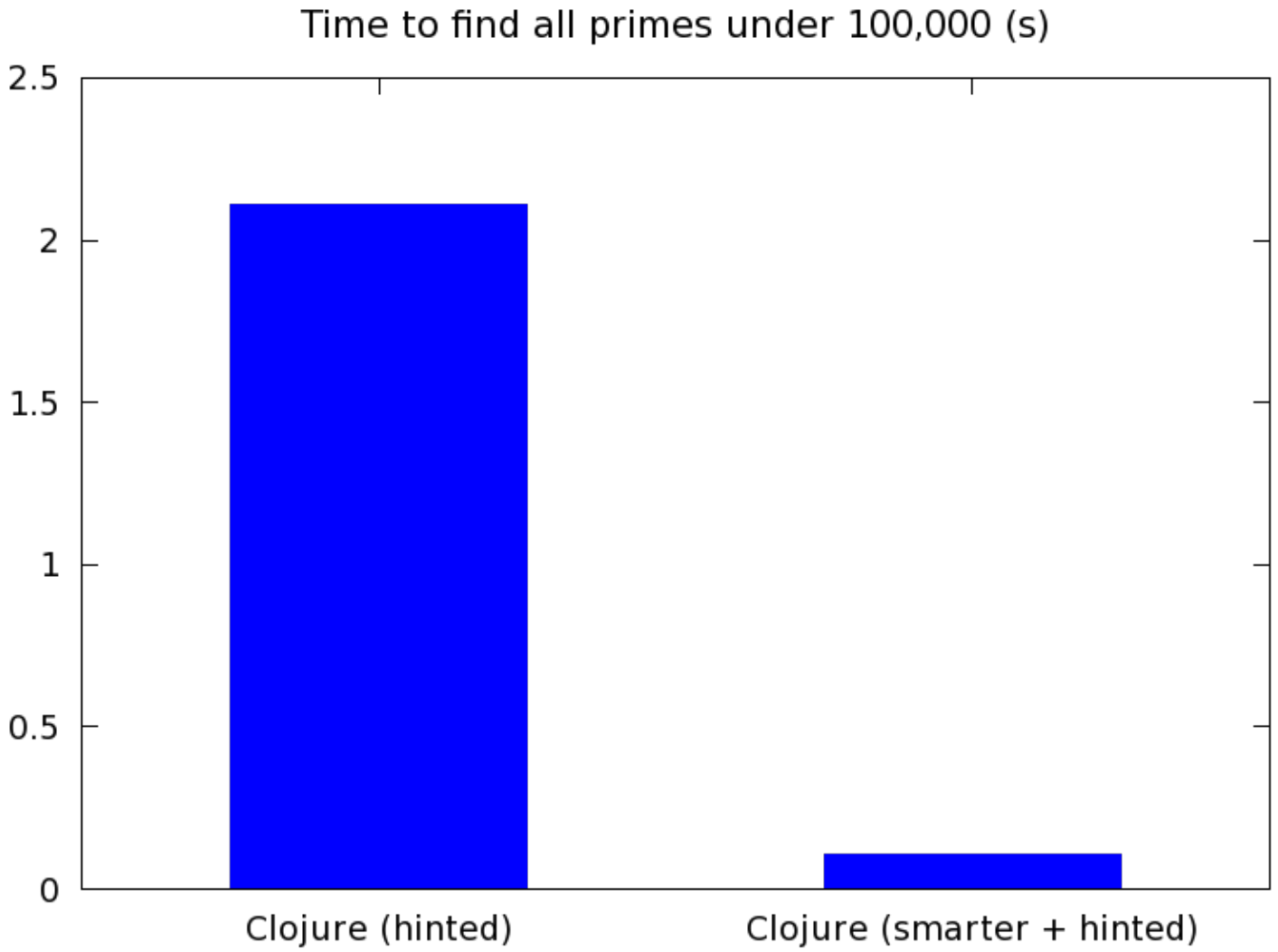
next-prime

Don’t check even numbers

```clojure
(defn next-prime
  [^long n primes]
  (if (has-prime-factor? n primes)
    (recur (+ n 2) primes)
    n))
```

Performance

Using a better algorithm makes a huge impact.

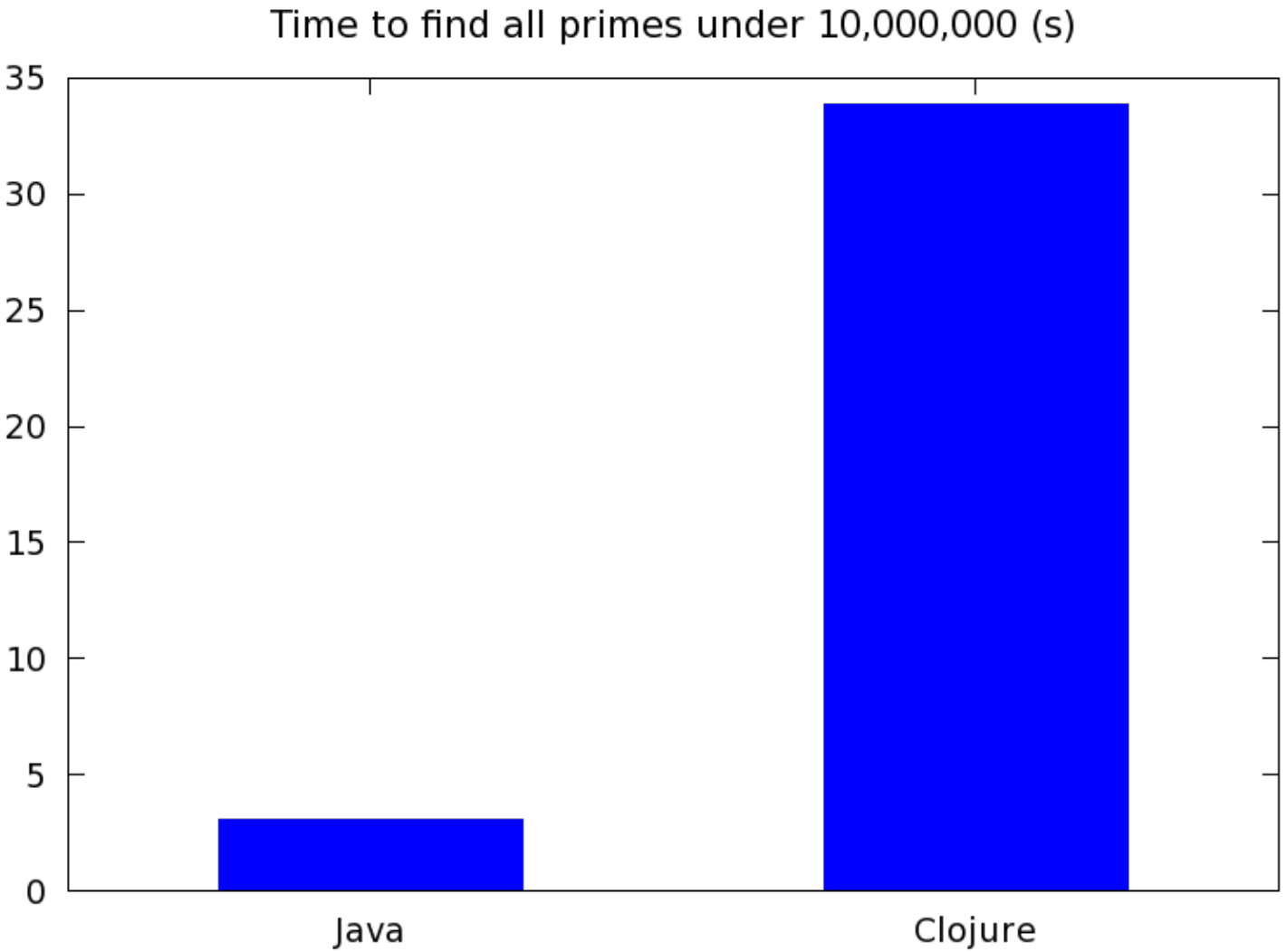


Tip 2: Algorithms matter

- Bad algorithms hurt
- Test your alorithm against realistic data sets

Performance

Now ten times slower than Java



Top 10 hot spots

Smarter, primitive-hinted lazy solution

Method/function	%
clojure.lang.Var.getRawRoot	10
clojure.core/take-while/fn	10
clojure.core/some	5
clojure.core/first	4
clojure.lang.RT.first	4
has-prime-factor?/fn	3
clojure.lang.RT.seq	3
clojure.core/seq	3
clojure.core/take-while	3
Total	45

Top 10 hot spots

Smarter, primitive-hinted lazy solution

Method/function	%
clojure.lang.Var.getRawRoot	10
clojure.core/take-while/fn	10
clojure.core/some	5
clojure.core/first	4
clojure.lang.RT.first	4
has-prime-factor?/fn	3
clojure.lang.RT.seq	3
clojure.core/seq	3
clojure.core/take-while	3
Sequence-related	30

Sequences in action

Tracing `has-prime-factor?`

```
(has-prime-factor? 11 [2 3 5 7])
```

Sequences in action

Tracing has-prime-factor?

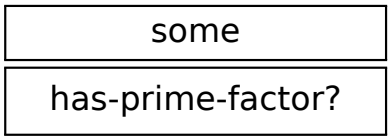
```
; (has-prime-factor? n primes)
;  n = 11
;  primes = [3 5 7]
(some #(divides n %)
      (take-while #(<= % (Math/sqrt n))
                  primes))
```

has-prime-factor?

Sequences in action

Tracing `has-prime-factor?`

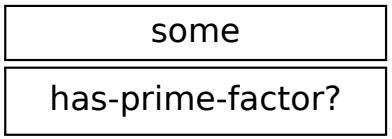
```
; (some pred coll)
;  pred = #(divides? 11 %)
;  coll = (take-while …)
(when (seq coll)
  (or (pred (first coll))
      (recur pred (next coll))))
```



Sequences in action

Tracing `has-prime-factor?`

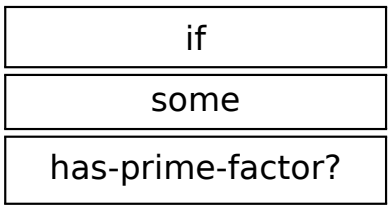
```
; when - macro expansion
;  pred = #(divides? 11 %)
;  coll = (take-while …)
(if (seq coll)
  (do
    (or (pred (first coll))
        (recur pred (next coll)))))
```



Sequences in action

Tracing `has-prime-factor?`

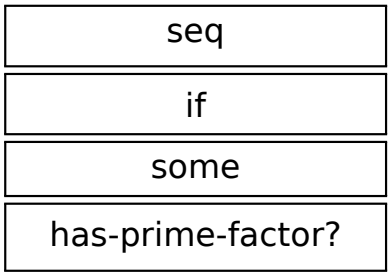
```
; (seq coll)
;   coll = (take-while …)
(seq coll)
```



Sequences in action

Tracing `has-prime-factor?`

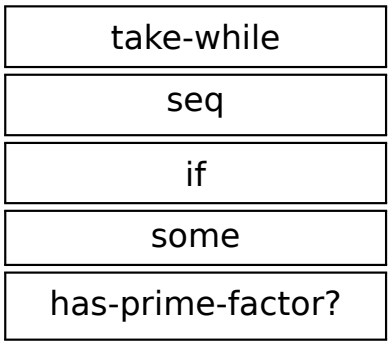
```
; coll
;  pred = #(<= % (Math/sqrt 11))
;  coll = [3 5 7]
(take-while pred coll)
```



Sequences in action

Tracing `has-prime-factor?`

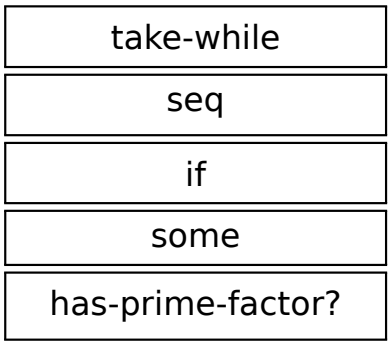
```
; (take-while pred coll)
;  pred = #(<= % (Math/sqrt 11))
;  coll = [3 5 7]
(lazy-seq
  (when-let [s (seq coll)]
    (when (pred (first s))
      (cons (first s)
            (take-while pred
                        (rest s))))))
```



Sequences in action

Tracing `has-prime-factor?`

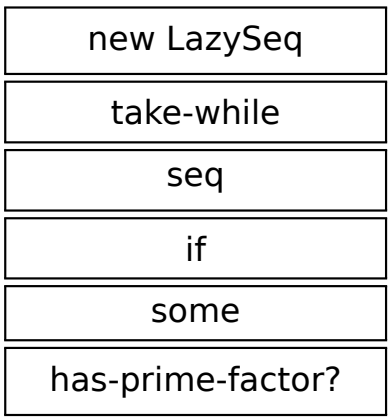
```
; lazy-seq - macro expansion
;  pred = #(<= % (Math/sqrt 11))
;  coll = [3 5 7]
(new clojure.lang.LazySeq
  (fn* []
    (when-let [s (seq coll)]
      (when (pred (first s))
        (cons (first s)
              (take-while pred
                          (rest s)))))))
```



Sequences in action

Tracing `has-prime-factor?`

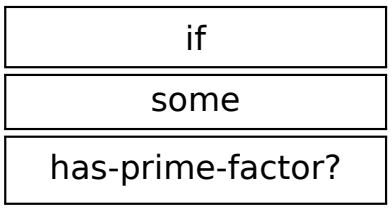
```
; lazy-seq - thunk expansion
;  pred = #(<= % (Math/sqrt 11))
;  coll = [3 5 7]
(new clojure.lang.LazySeq
  (fn* []
    (let* [temp# (seq coll)]
      (if temp#
        (do
          (let* [s temp#]
            (if (pred (first s))
              (do
                (cons (first s)
                      (take-while pred (rest s))))))))))))
```



Sequences in action

Tracing `has-prime-factor?`

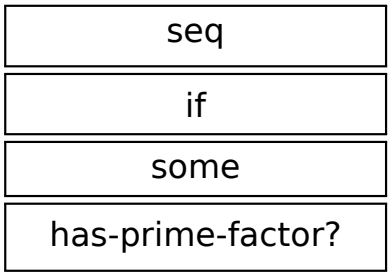
```
; (seq coll)
;   coll = LazySeq object
(seq coll)
```



Sequences in action

Tracing `has-prime-factor?`

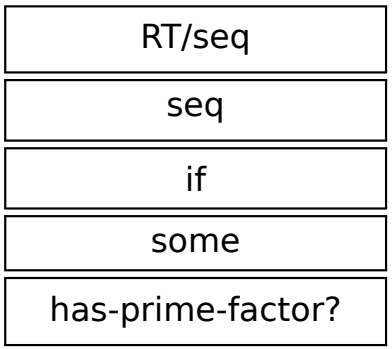
```
; (seq coll)
;   coll = LazySeq object
(RT/seq coll)
```



Sequences in action

Tracing `has-prime-factor?`

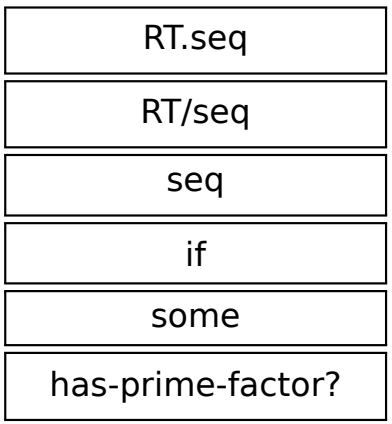
```
// RT.seq(Object coll)
//  coll = LazySeq object
if(coll instanceof ASeq)
    return (ASeq) coll;
else if(coll instanceof LazySeq)
    return ((LazySeq) coll).seq();
else
    return seqFrom(coll);
```



Sequences in action

Tracing `has-prime-factor?`

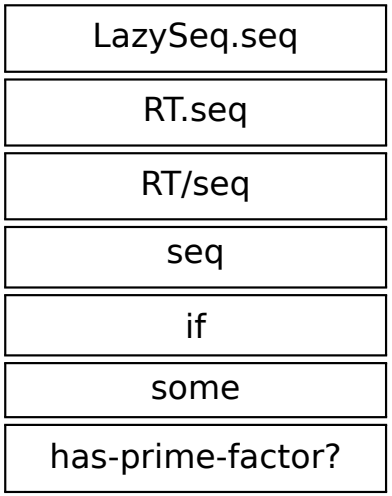
```
// LazySeq.seq()
//   fn = thunk
//   sv = null
//   s  = null
sval();
if(sv != null) {
    Object ls = sv;
    sv = null;
    while(ls instanceof LazySeq) {
        ls = ((LazySeq) ls).sval();
    }
    s = RT.seq(ls);
}
return s;
```



Sequences in action

Tracing has-prime-factor?

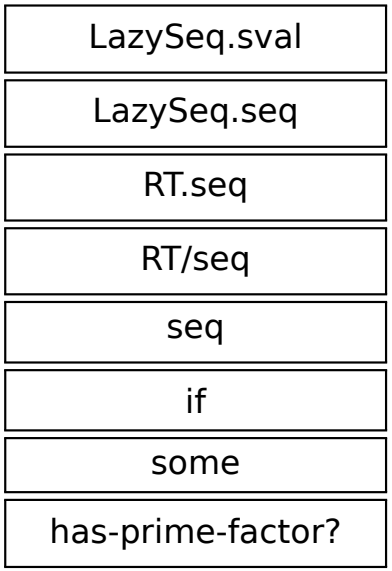
```
// LazySeq.sval()
//  fn = thunk
//  sv = null
//  s  = null
if (fn != null) {
    try {
        sv = fn.invoke();
        fn = null;
    } catch (RuntimeException e) {
        throw e;
    } catch (Exception e) {
        throw Util.runtimeException(e);
    }
}
if (sv != null)
    return sv;
return s;
```



Sequences in action

Tracing `has-prime-factor?`

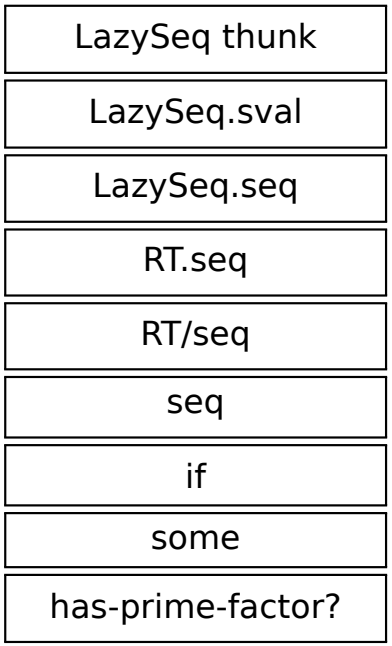
```
; lazy sequence thunk
;  pred = #(<= % (Math/sqrt 11))
;  coll = [3 5 7]
(let* [temp# (seq coll)]
  (if temp#
    (do
      (let* [s temp#]
        (if (pred (first s))
          (do
            (cons (first s)
                  (take-while pred (rest s)))))))))
```



Sequences in action

Tracing `has-prime-factor?`

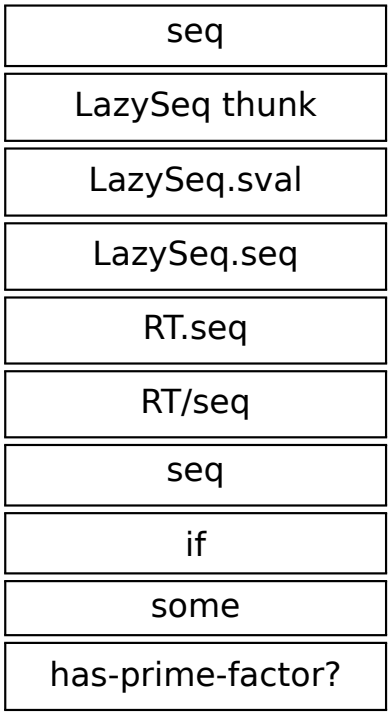
```
; (seq coll)
;   coll = [3 5 7]
(seq coll)
```



Sequences in action

Tracing `has-prime-factor?`

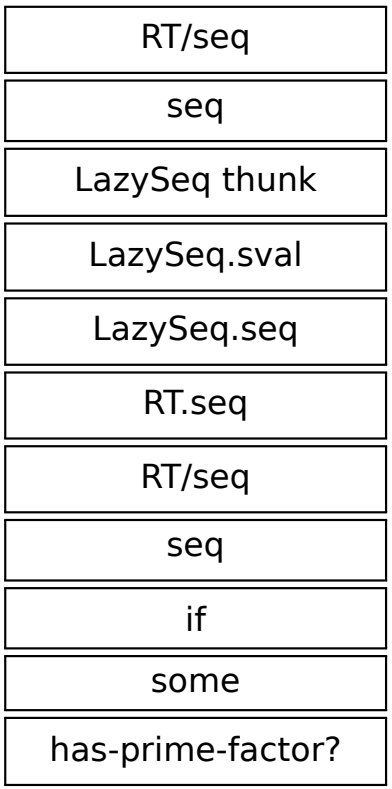
```
; (seq coll)
;   coll = [3 5 7]
(RT/seq coll)
```



Sequences in action

Tracing `has-prime-factor?`

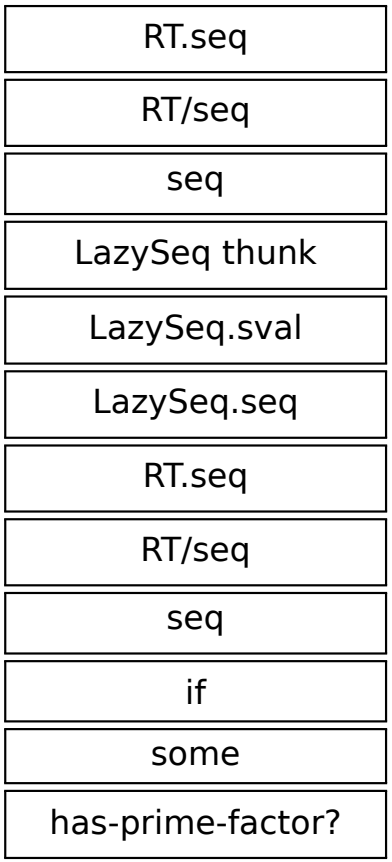
```
// RT.seq(Object coll)
//  coll = [3 5 7]
if(coll instanceof ASeq)
    return (ASeq) coll;
else if(coll instanceof LazySeq)
    return ((LazySeq) coll).seq();
else
    return seqFrom(coll);
```



Sequences in action

Tracing `has-prime-factor?`

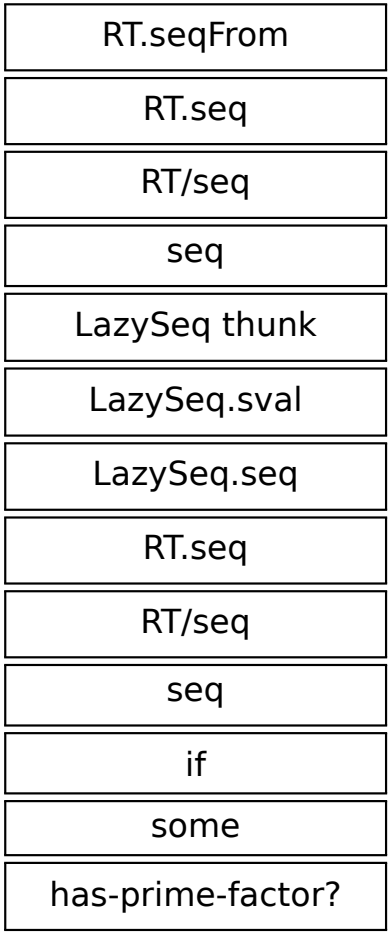
```
// RT.seqFrom(Object coll)
//  coll = [3 5 7]
if(coll instanceOf Seqable)
  return ((Seqable) coll).seq();
// more lines...
```



Sequences in action

Tracing `has-prime-factor?`

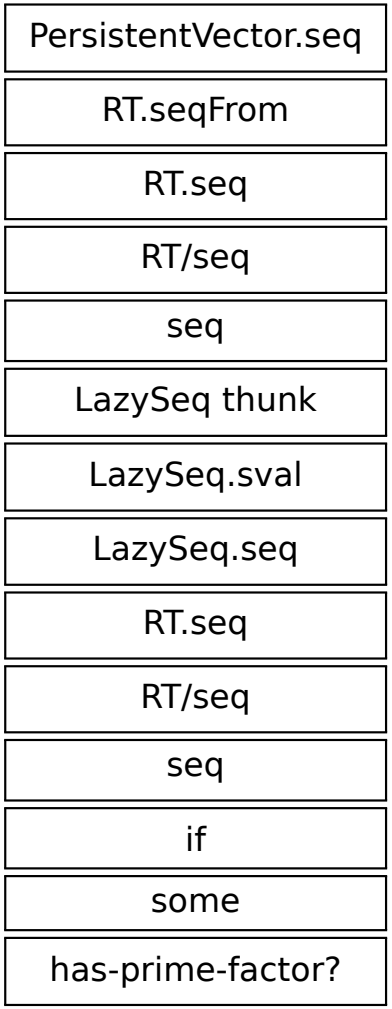
```
// PersistentVector.seq()
return chunkedSeq();
```



Sequences in action

Tracing `has-prime-factor?`

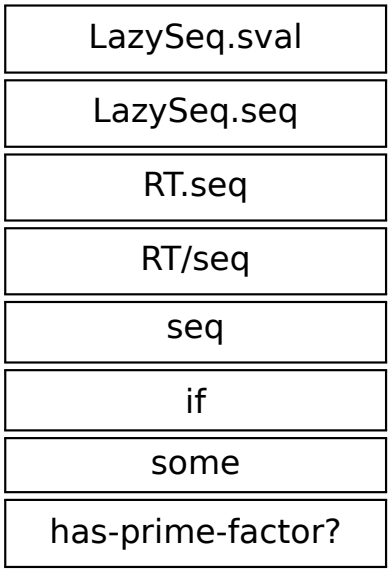
```
// PersistentVector.chunkedSeq()
if (count() == 0)
    return null;
return new ChunkedSeq(this, 0, 0);
```



Sequences in action

Tracing `has-prime-factor?`

```
; lazy sequence thunk
;  pred = #(<= % (Math/sqrt 11))
;  coll = [3 5 7]
(let* [temp# (seq coll)]
  (if temp#
    (do
      (let* [s temp#]
        (if (pred (first s))
          (do
            (cons (first s)
                  (take-while pred (rest s)))))))))
```



Sequences

An excellent abstraction with a performance cost.

Sidestepping sequences

Old `prime-seq`

```
(def prime-seq
  (cons 2
        ((fn inner [^long p
                    primes]
           (lazy-seq
             (cons p
                   (inner (next-prime (+ 2 p) primes)
                          (conj primes p)))))
           3 [])))
```

Sidestepping sequences

New `prime-seq`

```
(def prime-seq
  (cons 2
        ((fn inner [^long p
                    ^IPersistentCollection primes]
           (lazy-seq
             (cons p
                   (inner (next-prime (+ 2 p) primes)
                          (.cons primes p)))))
           3 [])))
```

Sidestepping sequences

Old `has-prime-factor?`

```
(defn has-prime-factor?
  [^long n primes]
  (some #(divides? n %)
        (take-while
          #(<= % (Math/sqrt n))
          primes)))
```

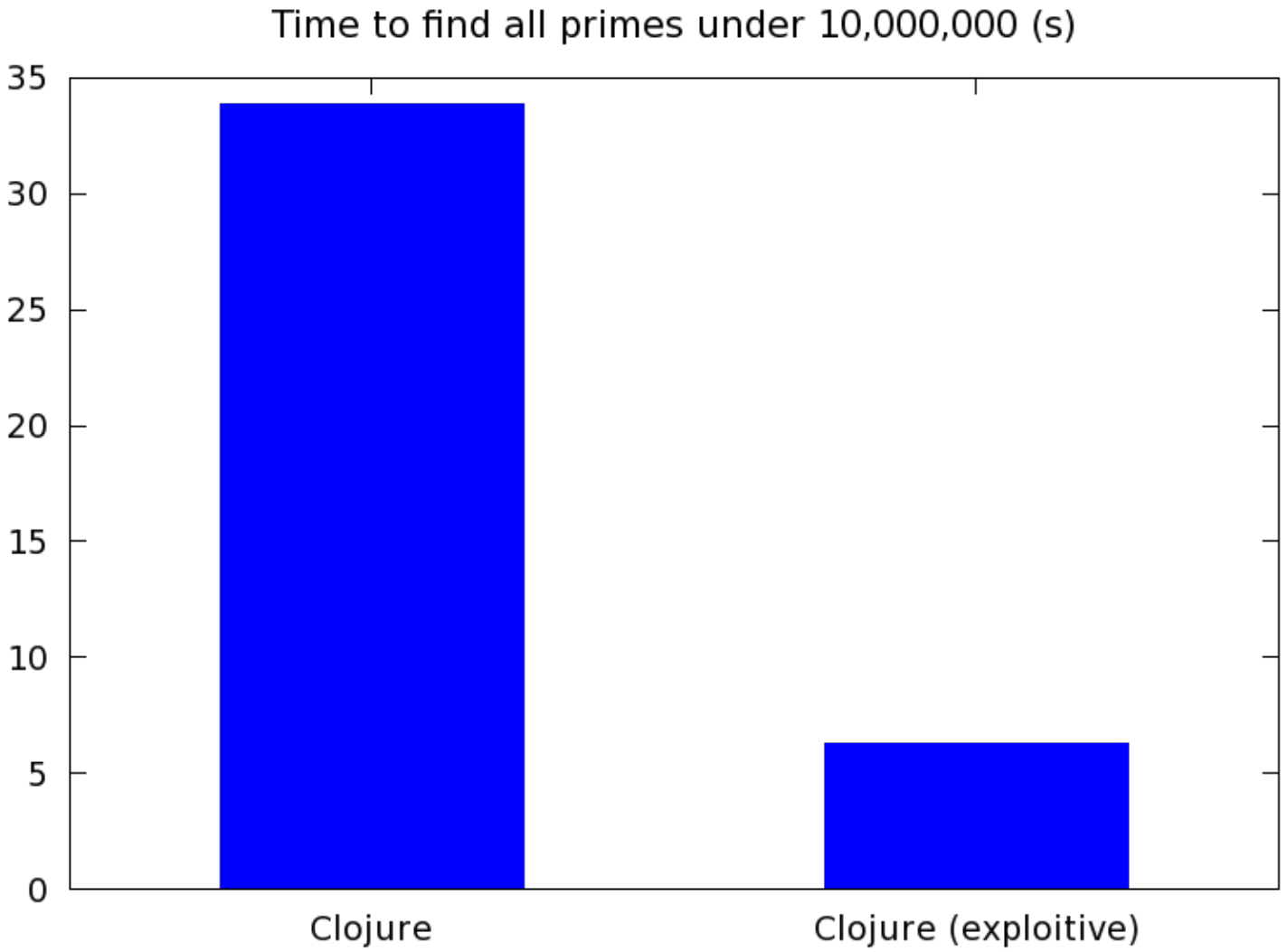
Sidestepping sequences

New `has-prime-factor?`

```
(defn has-prime-factor?
  [^long n ^IPersistentVector primes]
  (let [c     (.length primes)
        sqrtn (long (Math/sqrt n))]
    (loop [i 0]
      (when-not (= i c)
        (let [p (long (.nth primes i))]
          (cond
            (> p sqrtn)    false
            (divides? n p) true
            :else          (recur (inc i)))))))))
```

Performance

Exploiting types gives a 5× improvement



Tip 3: Exploit known types

- Increases performance considerably
- Minimises generality
- May result in non-idiomatic code

Top 10 hot spots

Exploitive lazy solution

Method/function	%
has-prime-factor?	27
clojure.lang.RT.longCast	9
divides?	9
clojure.lang.PersistentVector.nth	9
clojure.lang.PersistentVector.arrayFor	9
clojure.lang.Var.getRawRoot	5
clojure.lang.RT.intCast	5
clojure.lang.Numbers.inc	4
clojure.lang.PersistentVector.tailoff	4
java.lang.Number.longValue	4
Total	87

Top 10 hot spots

Exploitive lazy solution

Method/function	%
has-prime-factor?	27
clojure.lang.RT.longCast	9
divides?	9
clojure.lang.PersistentVector.nth	9
clojure.lang.PersistentVector.arrayFor	9
clojure.lang.Var.getRawRoot	5
clojure.lang.RT.intCast	5
clojure.lang.Numbers.inc	4
clojure.lang.PersistentVector.tailoff	4
java.lang.Number.longValue	4
Vector-related	40

Choosing a data structure

- Vectors
- Primitive vectors
- Arrays

Vectors

```
user=> (def v [1 2 3])
#'user/v
user=> (class v)
clojure.lang.PersistentVector
user=> (conj v 5.0)
[1 2 3 5.0]
user=> (conj v true)
[1 2 3 true]
user=> (.isPrim d (.nth ^PersistentVector v 1))
false
user=>
```

Primitive vectors

```
user=> (def v (vector-of :long 1 2 3))
#'user/v
user=> (class v)
clojure.core.Vec
user=> (isa? Vec IPersistentVector)
true
user=> (conj v 5.0)
[1 2 3 5]
user=> (conj v true)
ClassCastException java.lang.Boolean
cannot be cast to java.lang.Number
user=> (.isPrim d (.nth ^Vec v 1))
false
user=>
```

Arrays

```
user=> (def v (long-array [1 2 3]))
#'user/v
user=> (class v)
[J
user=> (conj v 5)
ClassCastException [J cannot be cast
to clojure.lang.IPersistentCollection
user=> (.isPrim d (aget ^longs v 1))
true
user=>
```

Converting to arrays

Old `has-prime-factor?`

```
(defn has-prime-factor?
  [^long n ^IPersistentVector primes]
  (let [c     (.length primes)
        sqrtn (long (Math/sqrt n))]
    (loop [i 0]
      (when-not (= i c)
        (let [p (long (.nth primes i))]
          (cond
            (> p sqrtn)    false
            (divides? n p) true
            :else          (recur (inc i)))))))))
```

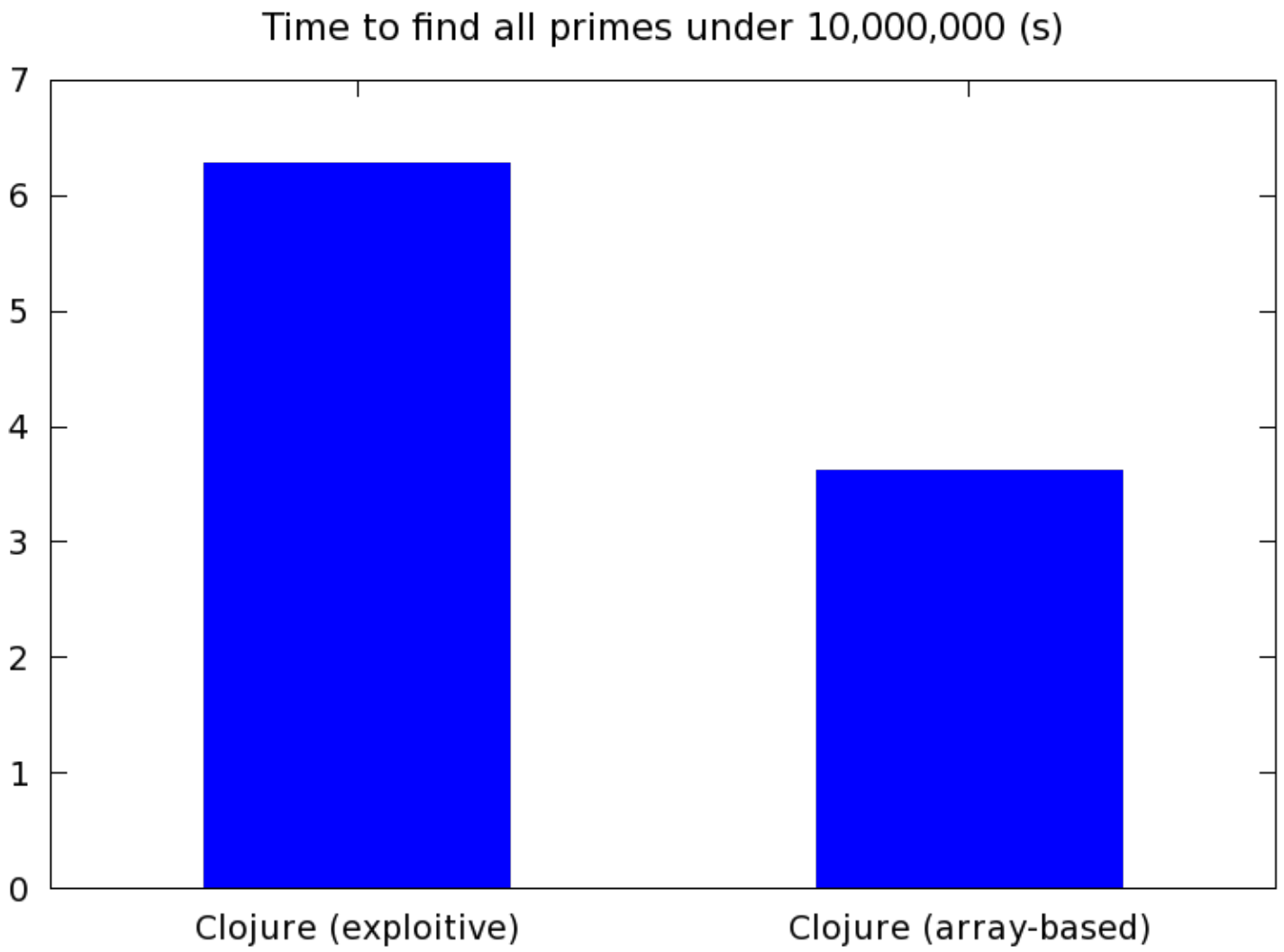
Converting to arrays

New has-prime-factor?

```
(defn has-prime-factor?
  [^long n ^longs primes]
  (let [c     (aget primes 0)
        sqrtn (long (Math/sqrt n))]
    (loop [i 1]
      (when-not (= i c)
        (let [p (aget primes i)]
          (cond
            (> p sqrtn)    false
            (divides? n p) true
            :else          (recur (inc i)))))))))
```

Performance

Using arrays cuts another 42%



Tip 4: Choose the right data structure

	Vectors	Primitive vectors	Arrays
Unboxed access?	No	No	Yes
Homogeneous?	No	Yes	Yes
Persistent?	Yes	Yes	No
Size for 1M longs (KiB)	28,726	9,198	7,812

Array implementation

Is it thread-safe?

Top 10 hot spots

Array-based lazy solution

Method/function	%
has-prime-factor?	34
divides?	18
clojure.lang.Var.getRawRoot	10
clojure.lang.RT.intCast	10
clojure.lang.Numbers.inc	9
clojure.lang.Numbers.isZero	8
next-prime	1
clojure.lang.Numbers.num	0
clojure.lang.RT.doubleCast	0
prime-seq/inner/fn	0
Total	90

Top 10 hot spots

Array-based lazy solution

Method/function	%
has-prime-factor?	34
divides?	18
clojure.lang.Var.getRawRoot	10
clojure.lang.RT.intCast	10
clojure.lang.Numbers.inc	9
clojure.lang.Numbers.isZero	8
next-prime	1
clojure.lang.Numbers.num	0
clojure.lang.RT.doubleCast	0
prime-seq/inner/fn	0
Maths safety-related	27

Maths safety

New behaviour as of Clojure 1.3

```
user=>
```

Maths safety

New behaviour as of Clojure 1.3

```
user=> (+ Long/MAX_VALUE 1)
```

Maths safety

New behaviour as of Clojure 1.3

```
user=> (+ Long/MAX_VALUE 1)
ArithmeticException integer overflow
user=>
```

Maths safety

Getting the old behaviour

```
user=> (+ Long/MAX_VALUE 1)
ArithmeticException integer overflow
user=> (+ Long/MAX_VALUE 1N)
```

Maths safety

Getting the old behaviour

```
user=> (+ Long/MAX_VALUE 1)
ArithmeticException integer overflow
user=> (+ Long/MAX_VALUE 1N)
9223372036854775808N
user=>
```

Maths safety

Unchecked operations

```
user=> (+ Long/MAX_VALUE 1)
ArithmeticException integer overflow
user=> (+ Long/MAX_VALUE 1N)
9223372036854775808N
user=> (unchecked-add Long/MAX_VALUE 1)
```

Maths safety

Unchecked operations

```
user=> (+ Long/MAX_VALUE 1)
ArithmeticException integer overflow
user=> (+ Long/MAX_VALUE 1N)
9223372036854775808N
user=> (unchecked-add Long/MAX_VALUE 1)
-9223372036854775808
user=>
```

Maths safety

Computer, disengage the safety protocols!

```
user=> (+ Long/MAX_VALUE 1)
ArithmeticException integer overflow
user=> (+ Long/MAX_VALUE 1N)
9223372036854775808N
user=> (unchecked-add Long/MAX_VALUE 1)
-9223372036854775808
user=> (set! *unchecked-math* true)
```

Maths safety

Computer, disengage the safety protocols!

```
user=> (+ Long/MAX_VALUE 1)
ArithmeticException integer overflow
user=> (+ Long/MAX_VALUE 1N)
9223372036854775808N
user=> (unchecked-add Long/MAX_VALUE 1)
-9223372036854775808
user=> (set! *unchecked-math* true)
true
user=>
```

Maths safety

Computer, disengage the safety protocols!

```
user=> (+ Long/MAX_VALUE 1)
ArithmeticException integer overflow
user=> (+ Long/MAX_VALUE 1N)
9223372036854775808N
user=> (unchecked-add Long/MAX_VALUE 1)
-9223372036854775808
user=> (set! *unchecked-math* true)
true
user=> (+ Long/MAX_VALUE 1)
```

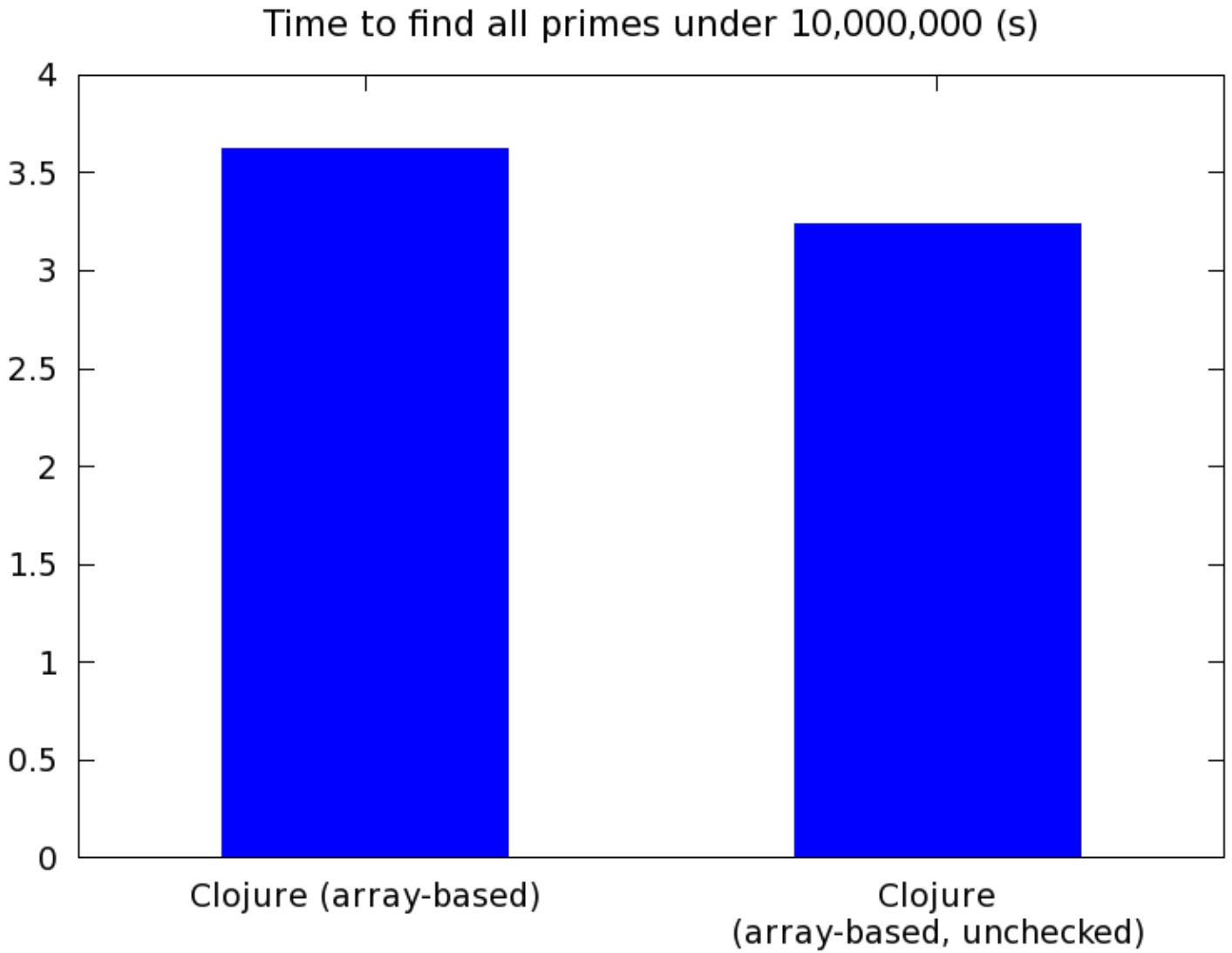
Maths safety

Computer, disengage the safety protocols!

```
user=> (+ Long/MAX_VALUE 1)
ArithmeticException integer overflow
user=> (+ Long/MAX_VALUE 1N)
9223372036854775808N
user=> (unchecked-add Long/MAX_VALUE 1)
-9223372036854775808
user=> (set! *unchecked-math* true)
true
user=> (+ Long/MAX_VALUE 1)
-9223372036854775808
user=>
```

Performance

Unchecked maths cuts another 10% off



Tip 5: Use unchecked arithmetic

- `*unchecked-math*` is a *compile time option*
- Leave off during development, turn on for production?
- Clojure 1.3 introduces `unchecked-`*`op`*`-int` operations

More Clojure arithmetic

```
user=>
```

More Clojure arithmetic

```
user=> (/ -8 6)
```

More Clojure arithmetic

```
user=> (/ -8 6)
-4/3
user=>
```

More Clojure arithmetic

```
user=> (/ -8 6)
-4/3
user=> (type (/ -8 6))
```

More Clojure arithmetic

```
user=> (/ -8 6)
-4/3
user=> (type (/ -8 6))
clojure.lang.Ratio
user=>
```

More Clojure arithmetic

```
user=> (/ -8 6)
-4/3
user=> (type (/ -8 6))
clojure.lang.Ratio
user=> (quot -8 6)
```

More Clojure arithmetic

```
user=> (/ -8 6)
-4/3
user=> (type (/ -8 6))
clojure.lang.Ratio
user=> (quot -8 6)
-1
user=>
```

More Clojure arithmetic

```
user=> (/ -8 6)
-4/3
user=> (type (/ -8 6))
clojure.lang.Ratio
user=> (quot -8 6)
-1
user=> (mod -8 6)
```

More Clojure arithmetic

```
user=> (/ -8 6)
-4/3
user=> (type (/ -8 6))
clojure.lang.Ratio
user=> (quot -8 6)
-1
user=> (mod -8 6)
4
user=>
```

More Clojure arithmetic

```
user=> (/ -8 6)
-4/3
user=> (type (/ -8 6))
clojure.lang.Ratio
user=> (quot -8 6)
-1
user=> (mod -8 6)
4
user=> (rem -8 6)
```

More Clojure arithmetic

```
user=> (/ -8 6)
-4/3
user=> (type (/ -8 6))
clojure.lang.Ratio
user=> (quot -8 6)
-1
user=> (mod -8 6)
4
user=> (rem -8 6)
-2
user=>
```

Tip 6: Use the right division

When doing integral division:

- Avoid `/` unless you really want to be rational.
- Use `quot` and `rem` for fast, native division operations.
- Use `mod` for true moduluar arithmetic.

Top 10 hot spots

Unchecked, array-based lazy solution

Method/function	%
has-prime-factor?	29
divides?	28
clojure.lang.Var.getRawRoot	17
clojure.lang.Numbers.isZero	13
next-prime	1
clojure.lang.Numbers.num	1
clojure.lang.RT.uncheckedDoubleCast	1
prime-seq/inner/fn	0
java.lang.Number.doubleValue	0
clojure.lang.Cons.next	0
Total	90

Top 10 hot spots

Unchecked, array-based lazy solution

Method/function	%
has-prime-factor?	29
divides?	28
clojure.lang.Var.getRawRoot	17
clojure.lang.Numbers.isZero	13
next-prime	1
clojure.lang.Numbers.num	1
clojure.lang.RT.uncheckedDoubleCast	1
prime-seq/inner/fn	0
java.lang.Number.doubleValue	0
clojure.lang.Cons.next	0
What's this?	17

Var binding in Clojure

```
user=> (def foo :foo)
#'user/foo
user=> foo
:foo
user=> (binding [foo :bar] foo)
IllegalStateException Can't dynamically bind
non-dynamic var: #'user/foo
user=> (def foo :bar)
#'user/foo
user=> foo
:bar
user=>
```

Introducing definline

Old divides?

```
(defn divides?
  [^long n ^long d]
  (zero? (rem n d)))
```

Introducing definline

Inlined divides?

```
(definline divides?
  [n d]
  `(zero? (rem ~n ~d)))
```

Introducing definline

Old has-prime-factor?

```
(defn has-prime-factor?
  [^long n ^longs primes]
  (let [c     (aget primes 0)
        sqrtn (long (Math/sqrt n))]
    (loop [i 1]
      (when-not (= i c)
        (let [p (aget primes i)]
          (cond
            (> p sqrtn)    false
            (divides? n p) true
            :else          (recur (inc i)))))))))
```

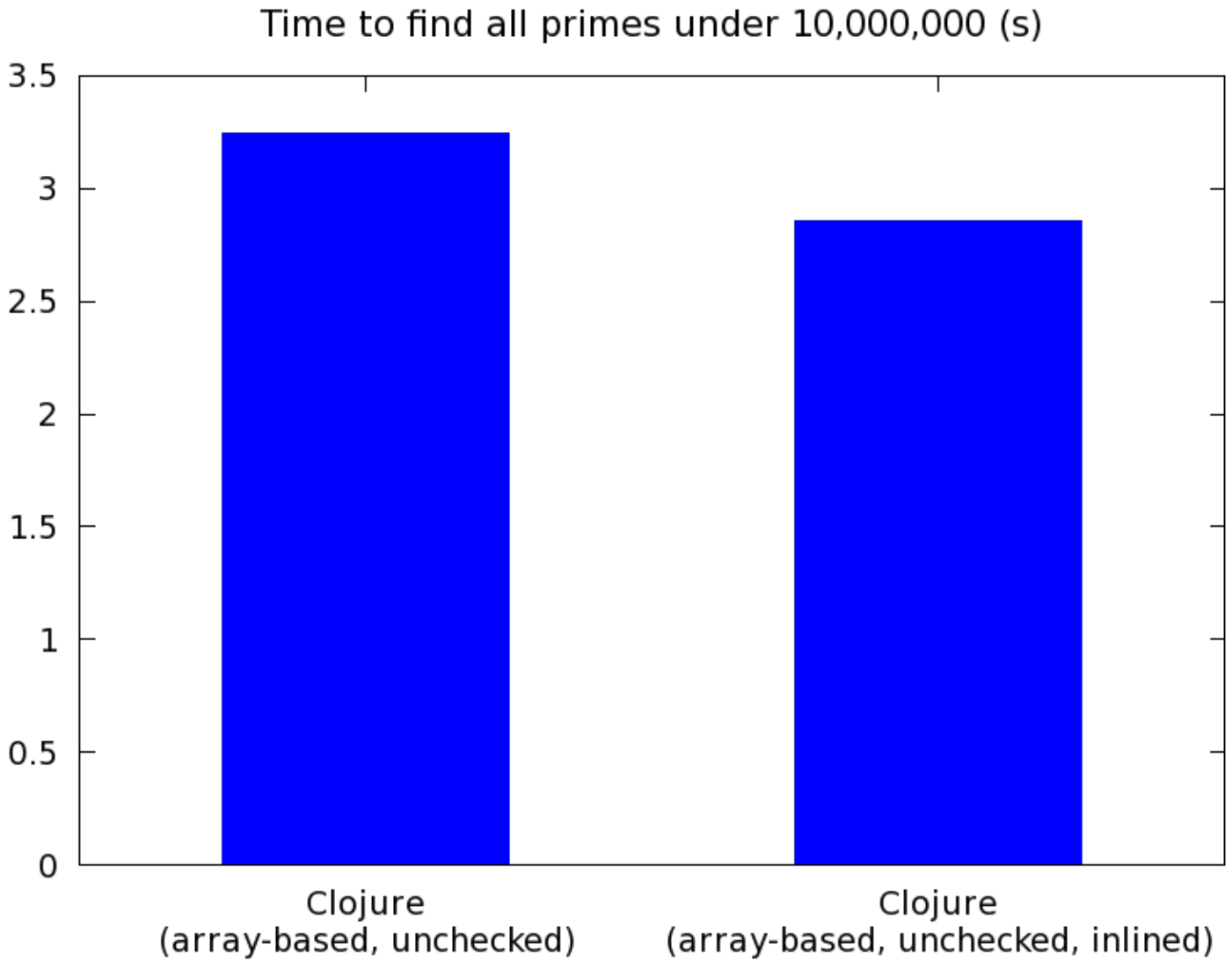
The horror!

Inlined `has-prime-factor?`

```clojure
(definline has-prime-factor?
  [n ^longs primes]
  `(let [c#     (aget ~primes 0)
         sqrtn# (long (Math/sqrt (double ~n)))]
     (loop [i# 1]
       (when-not (= i# c#)
         (let [p# (aget ~primes i#)]
           (cond
             (> p# sqrtn#)    false
             (divides? ~n p#) true
             :else            (recur (inc i#))))))))
```

Performance

Inlining reduces about another 10%



Tip 7: Inline hot functions

- Eliminates overhead of vars
- Makes code significantly harder to understand
- Best for small, frequently-called functions.

More on inlining...

```clojure
(defn area [radius]
  (* Math/PI radius radius))

(defn circumference [radius]
  (* 2 Math/PI radius))
```

More on inlining...

```
(def PI Math/PI)

(defn area [radius]
  (* PI radius radius))

(defn circumference [radius]
  (* 2 PI radius))
```

More on inlining...

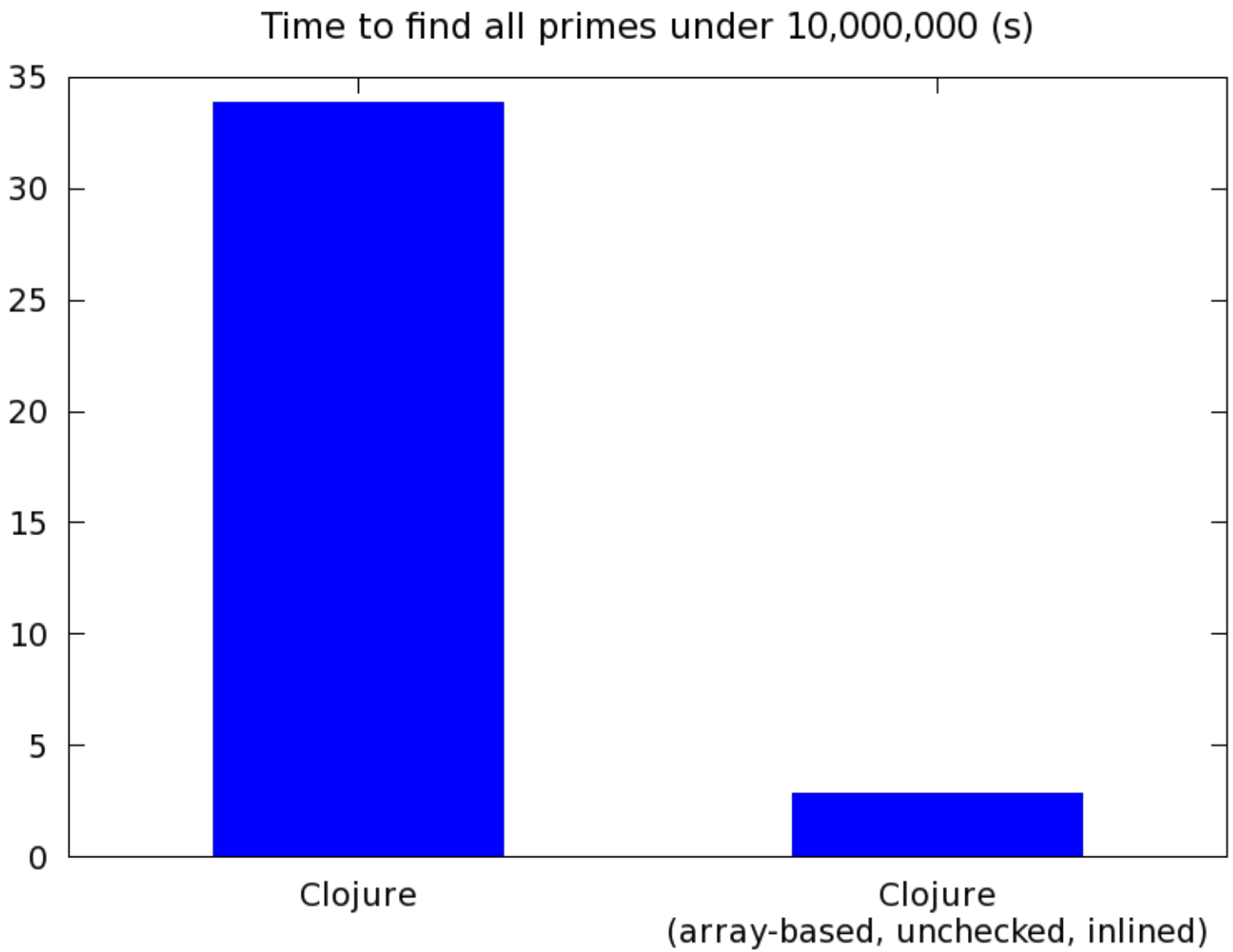
```
(def ^:const PI Math/PI)

(defn area [radius]
  (* PI radius radius))

(defn circumference [radius]
  (* 2 PI radius))
```

Tip 8: Inline constants

- Another freebie.
- Eliminates overhead of var lookup for long and double primitive constants.
- Clojure 1.2.1 and earlier simply ignore `^:const` metadata.

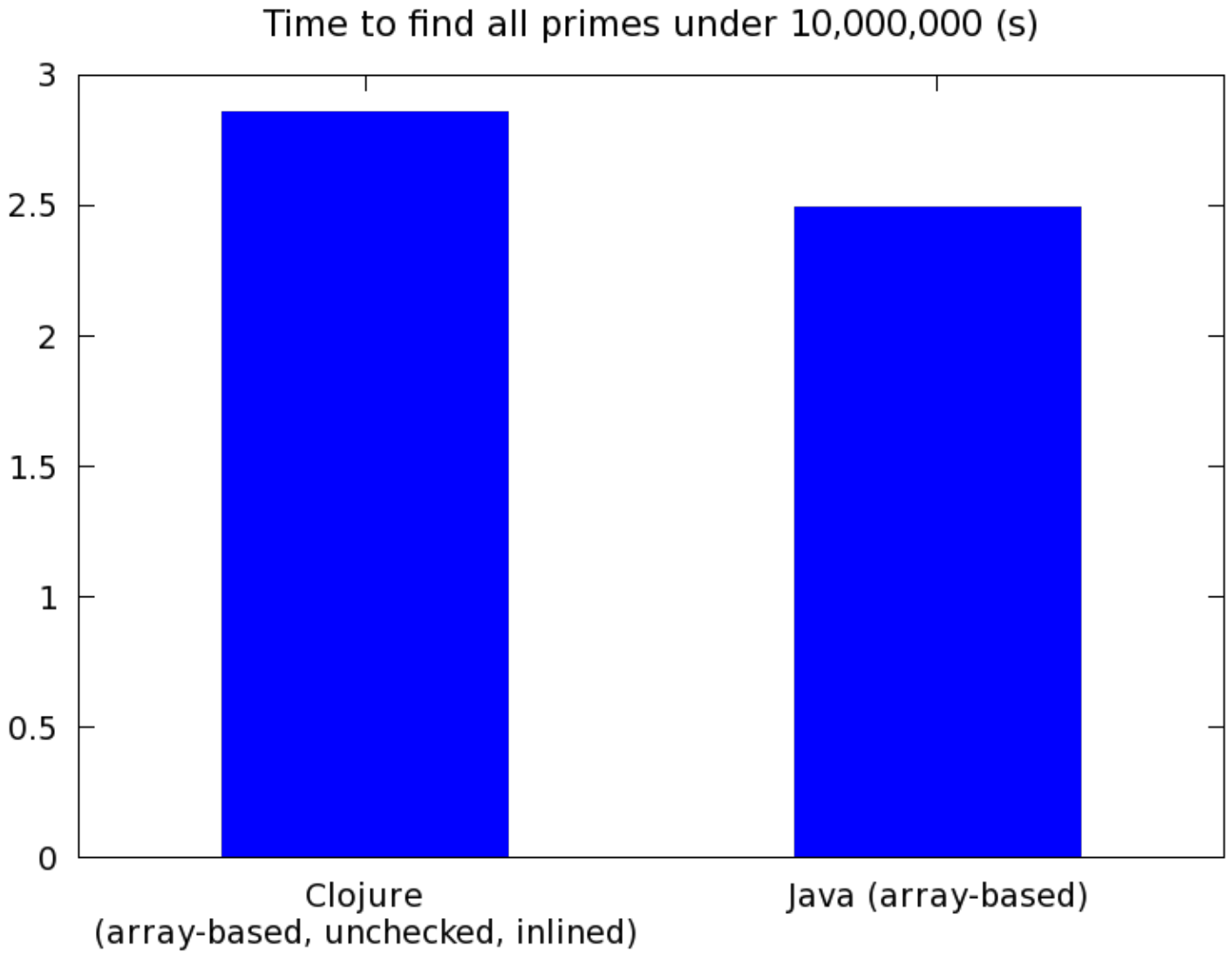
Performance

Clojure is now nearly twelve times faster



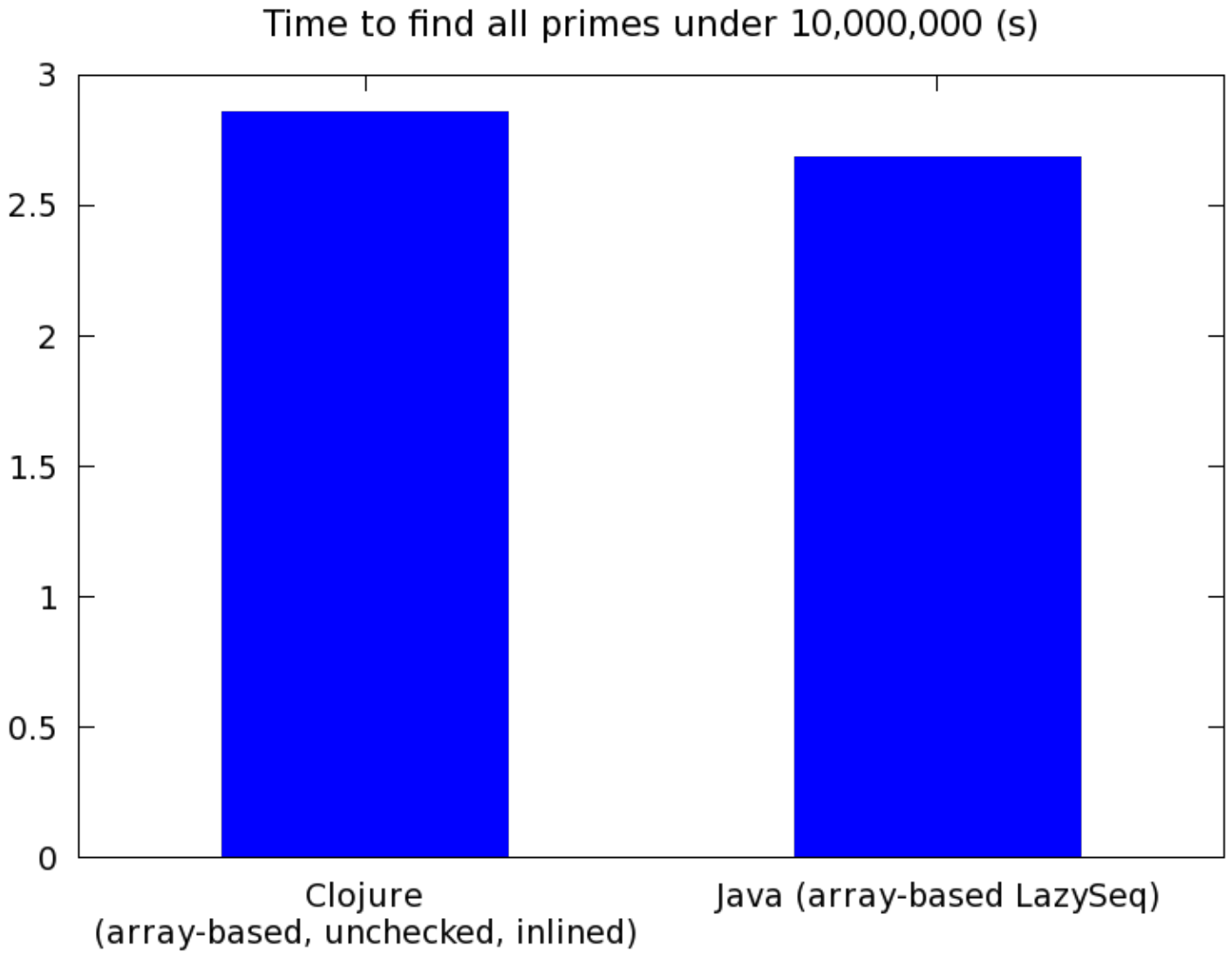
Performance

Clojure still about 15% slower than Java



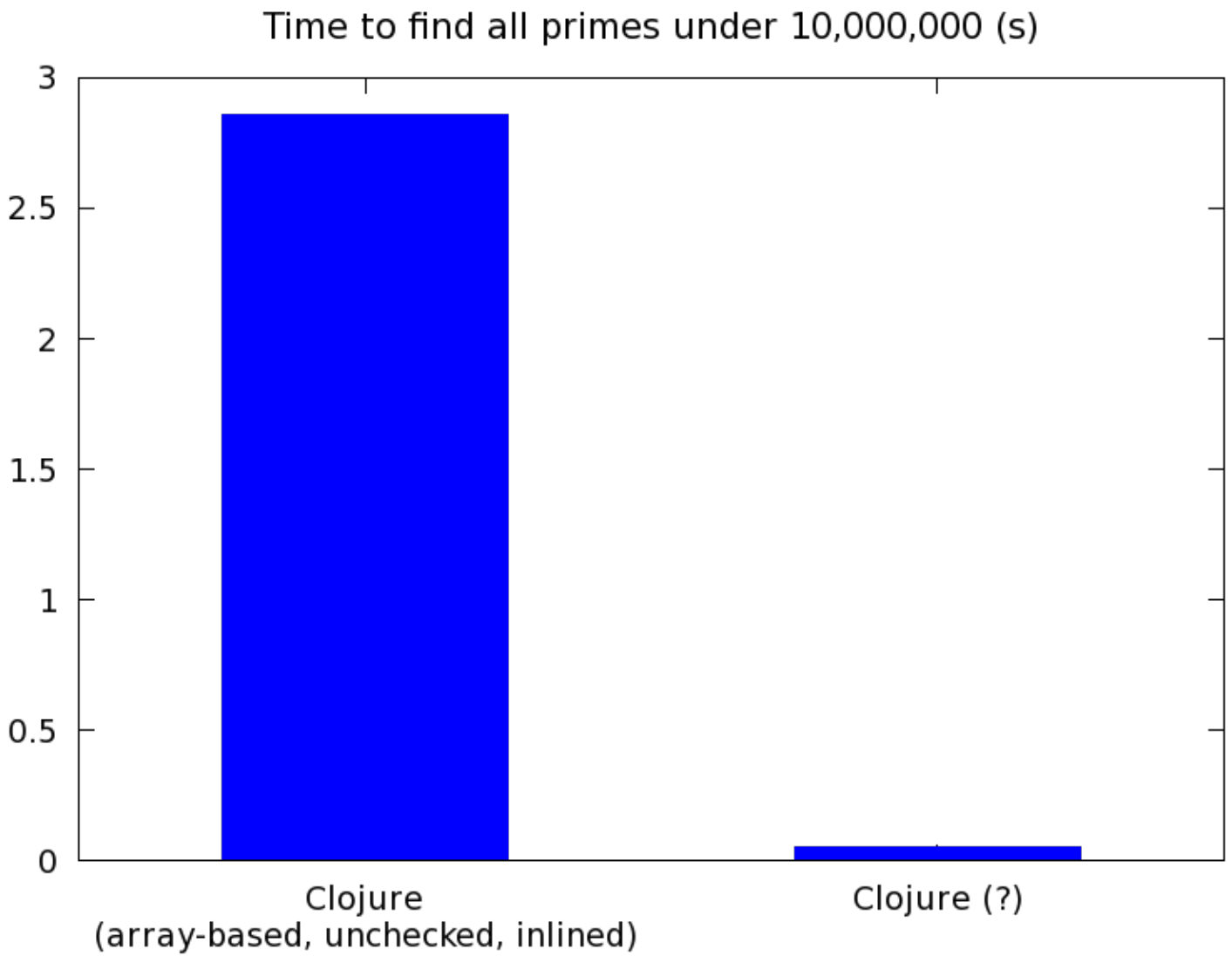
Performance

With Java wrapped in a LazySeq, Clojure is only 6% slower



Performance

We’re doing it wrong



Two mistakes

1. Bad algorithm: Trial division is much slower than a true Sieve of Eratosthenes.
2. Lazy approach adds significant overhead

An eager implementation

Returns an array of longs as a bitmap

```
(def ^:const even-mask 0x5555555555555555)

(defn get-primes
  [^long n]
  (let [sqrtn          (long (Math/sqrt (double n)))
        numlongs       (quot (+ n 63) 64)
        ^longs bitset (long-array numlongs even-mask)
        n              (* numlongs 64)]
    (aset bitset 0 (bit-xor even-mask 0x06))
    (loop [i 3
           j 9]
      (cond
        (>= i sqrtn) :done
        (zero? j)
          (let [array-off (quot i 64)
                long-off  (rem i 64)]
            (if (Numbers/testBit (aget bitset array-off) long-off)
              (recur (+ i 2) 0)
              (recur i (* i i))))
        (>= j n)
          (recur (+ i 2) 0)
        :else
          (let [array-off (quot j 64)
                long-off  (rem j 64)
                old-long  (aget bitset array-off)
                new-long  (Numbers/setBit old-long long-off)]
            (aset bitset array-off new-long)
            (recur i (+ i j))))))
    bitset))
```

Tip 9: Don’t be lazy

- Laziness requires use of the sequence abstraction.
- Some problems are easier solved efficiently using an eager/imperative algorithm.
- May result in non-idiomatic code.

Think about it...

Has someone else already solved this problem?

Using java.util.BitSet

```clojure
(def ^:const even-mask 0x5555555555555555)

(defn get-primes
  [^long n]
  (let [sqrtn     (long (Math/sqrt (double n)))
        numlongs  (quot (+ n 63) 64)
        bitset    (doto (BitSet/valueOf (long-array numlongs even-mask))
                    (.set 1)
                    (.clear 2))
        n         (.size bitset)]
    (loop [i 3
           j 9]
      (cond
        (>= i sqrtn) :done
        (zero? j)
          (if (.get bitset i)
            (recur (+ i 2) 0)
            (recur i (* i i)))
        (>= j n)
          (recur (+ i 2) 0)
        :else
          (do
            (.set bitset j)
            (recur i (+ i j)))))
    bitset))
```

Comparison

BitSet-based

```
(defn get-primes
  [^long n]
  (let [sqrtn          (long (Math/sqrt (double n)))
        numlongs       (quot (+ n 63) 64)
        bitset         (doto (BitSet/valueOf (long-array numlongs even-mask))
                         (.set 1)
                         (.clear 2))
        n              (.size bitset)]

    (loop [i 3
           j 9]
      (cond
        (>= i sqrtn) :done
        (zero? j)


            (if (.get bitset i)
              (recur (+ i 2) 0)
              (recur i (* i i)))
        (>= j n)
          (recur (+ i 2) 0)
        :else
          (do



            (.set bitset j)
            (recur i (+ i j)))))
    bitset))
```

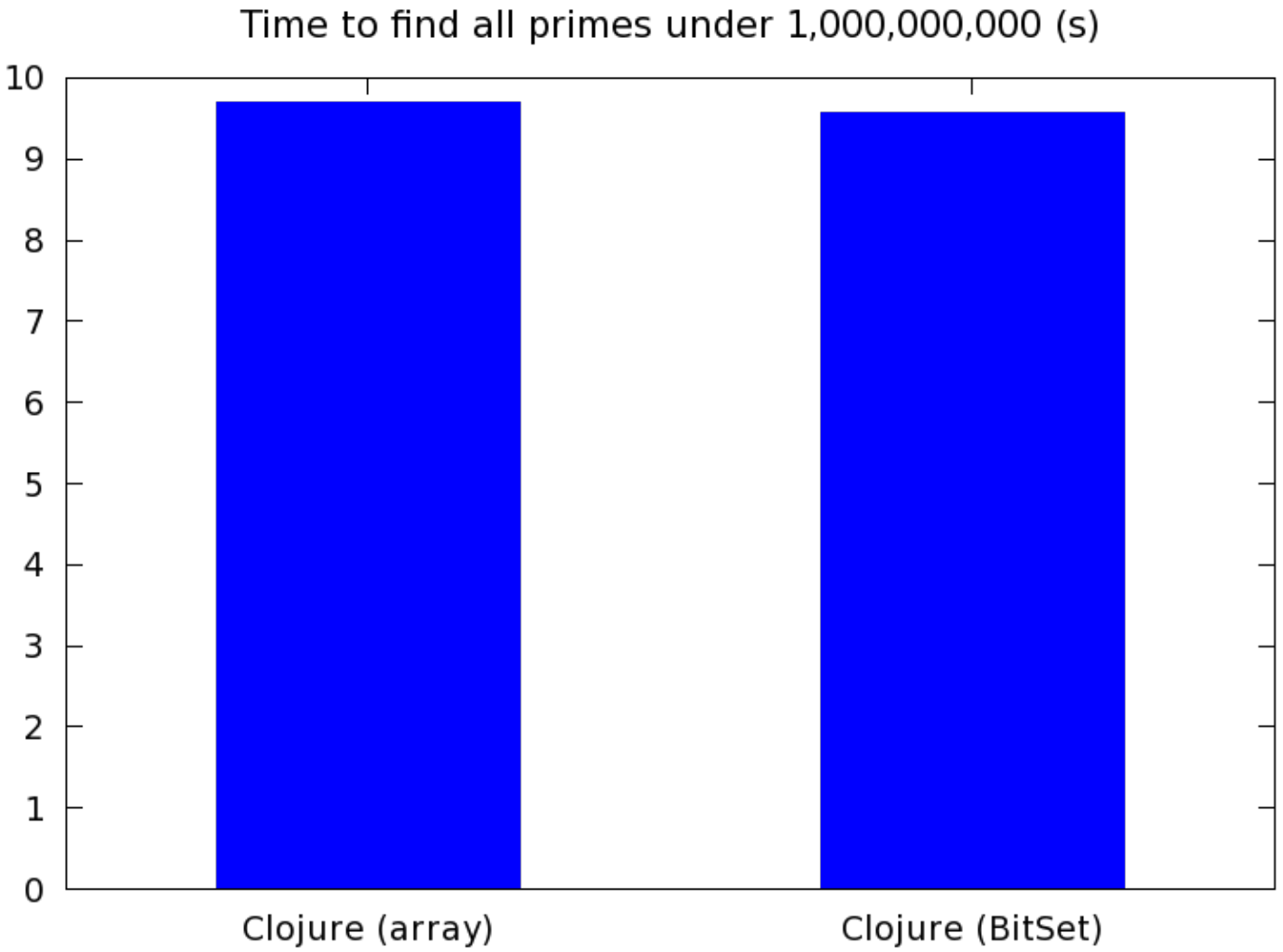
Comparison

Array-based

```
(defn get-primes
  [^long n]
  (let [sqrtn          (long (Math/sqrt (double n)))
        numlongs       (quot (+ n 63) 64)
        ^longs bitset (long-array numlongs even-mask)


        n              (* numlongs 64)]
    (aset bitset 0 (bit-xor even-mask 0x06))
    (loop [i 3
           j 9]
      (cond
        (>= i sqrtn) :done
        (zero? j)
          (let [array-off (quot i 64)
                long-off  (rem i 64)]
            (if (Numbers/testBit (aget bitset array-off) long-off)
              (recur (+ i 2) 0)
              (recur i (* i i))))
        (>= j n)
          (recur (+ i 2) 0)
        :else
          (let [array-off (quot j 64)
                long-off  (rem j 64)
                old-long  (aget bitset array-off)
                new-long  (Numbers/setBit old-long long-off)]
            (aset bitset array-off new-long)
            (recur i (+ i j)))))
    bitset))
```

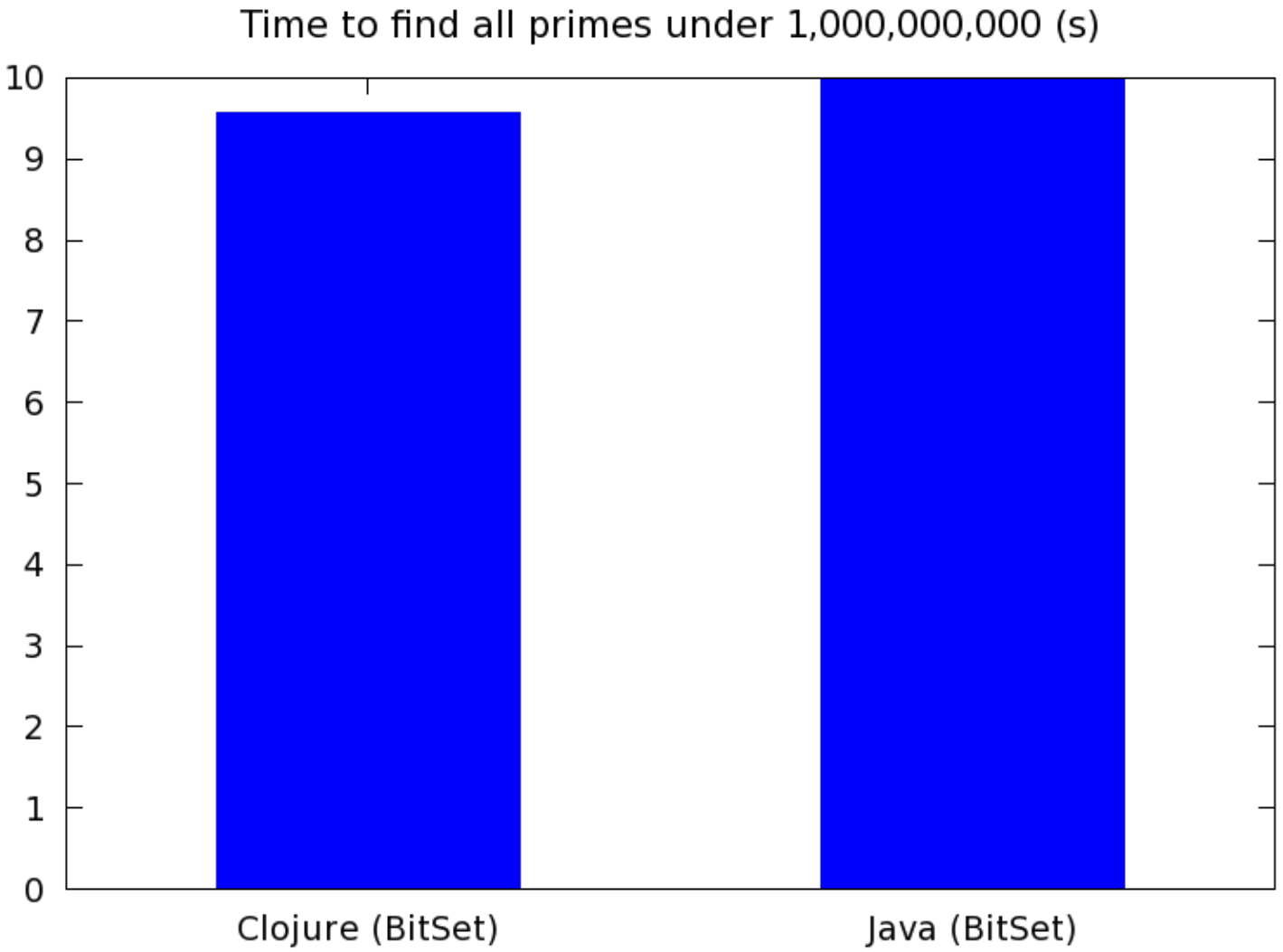
Performance

Both Clojure implementations perform about equally



Performance

Clojure slightly outperforms Java



Tip 10: Exploit the platform

The JVM has a rich ecosystem of open source libraries: *exploit it*.

Question

How do you diagnose a performance problem?

Answer

Assemble a performance analysis toolkit:

1. Profiler
2. Disassembler

Profilers

Why use a profiler?

- Measure actual code execution to find true bottlenecks
- Avoid premature optimization

Profilers

Where to get one?

- Oracle's Java includes a simple profiler.

```
java -agentlib:hprof=help
```

- Other commercial and open source profilers available

Disassemblers

Why use a disassembler?

- Identify locations of primitive boxing and other compiler inefficiencies
- Allows you to intelligently add type hints or explicit method calls

Disassemblers

Where to get one?

- Most JDK includes a simple disassembler, `javap`. Use `-c` option to see disassembled code.
- Other commercial and open source disassembler available

Tip 11: Use your tools

Use tools to help you properly analyse and debug performance problems.

Clojure performance wish list

- Persistent collections with primitive interfaces
- 32-bit arithmetic support
- `*warn-on-boxing*` or `*disable-boxing*` compile-time flag

11 Clojure performance tips

1. Use primitive hints
2. Algorithms matter
3. Exploit known types
4. Choose the right data structure
5. Use unchecked arithmetic
6. Use the right division
7. Inline hot functions
8. Inline constants
9. Don’t be lazy
10. Exploit the platform
11. Use your tools

Twitter @deepbluelambda

Blog http://www.deepbluelambda.org

Code https://github.com/sattvik/primes