

Size Doesn't Matter

or:
The ins and outs
of Minitest

or:
The ins and outs
of Minitest





ryan davis
founder

seattle ruby brigade

ryand-ruby@zenspider.com
www.zenspider.com/seattle.rb

Almost 11
years of
Ruby

84 slides in
30 minutes...
Keep up!



デービス・ライン
ライアン・デイビス
創設者

シアトルの Ruby の組

ryand-ruby@zenspider.com
www.zenspider.com/seattle.rb

What is Minitest?

- Replacement for ruby 1.8's test/unit.
 - Originally 90 lines of code.
- Available as a gem, and in ruby 1.9.
- Meant to be small, clean, and very fast.
- Provides a lot more than test/unit did.

6 Parts of Minitest

runner	The heart of the machine
minitest/unit	TDD API
minitest/spec	BDD API
minitest/mock	Simple mocking API
minitest/pride	IO pipelining example
minitest/bench	Abstract benchmark API

Minitest Runner

Every method starting with
"test" is run in a new context.

No Magic Allowed

- Even test discovery avoids ObjectSpace.
- Minimal metaprogramming.
- Uses plain classes and methods to do all work.

Test randomization prevents
order dependencies.

Verbose mode prints test times in a sortable format:

```
% ruby -Ilib test/test_meta.rb -v | sort -k2 -t= -nr | head -3  
MetaMagic#test_generate_meta_method = 0.03 s = .  
MetaMagic#test_complex_method = 0.02 s = .  
MetaMagic#test_simple_method = 0.01 s = .
```

Test summary provides useful statistics:

```
# Running tests:
```

```
.....  
.....  
.....  
.....  
.....  
.....  
.....  
.....  
.....  
.....  
.....
```

```
Finished tests in 0.510616s, 1339.5585 tests/s, 7428.2827 assertions/s.
```

```
684 tests, 3793 assertions, 0 failures, 0 errors, 0 skips
```

minitest/unit

Test Example

```
require "minitest/autorun"
```

```
class TestThingy < Minitest::Unit::TestCase
  def test_do_the_thing
    assert_equal 42, Thingy.do_the_thing
  end
end
```

Test Example

```
require "minitest/autorun"
```

Simple Subclass

```
class TestThingy < Minitest::Unit::TestCase
  def test_do_the_thing
    assert_equal 42, Thingy.do_the_thing
  end
end
```

Test Example

```
require "minitest/autorun"
```

Simple Subclass

```
class TestThingy < Minitest::Unit::TestCase
  def test_do_the_thing
    assert_equal 42, Thingy.do_the_thing
  end
end
```

Simple Method

Test Example

```
require "minitest/autorun"
```

Simple Subclass

```
class TestThingy < MiniTest::Unit::TestCase
  def test_do_the_thing
    assert_equal 42, Thingy.do_the_thing
  end
end
```

Simple Method

Magic
Free!

assertions

Positive Assertions

assert

assert_block

assert_empty

assert_equal

assert_in_delta

assert_in_epsilon

assert_includes

assert_instance_of

assert_kind_of

assert_match

assert_nil

assert_operator

assert_output

assert_raises

assert_respond_to

assert_same

assert_send

assert_silent

assert_throws

assert_equal diffs:

```
1) Failure:
test_failing_simple(TestSimple) [example.rb:8]:
Expected: 42
Actual: 24

2) Failure:
test_failing_complex(TestComplex) [example.rb:23]:
--- expected
+++ actual
@@ -22,7 +22,7 @@
   "line 22",
   "line 23",
   "line 24",
-  "line 25",
+  "something unexpected",
   "line 26",
   "line 27",
   "line 28",
```

Negative Assertions

refute

refute_empty

refute_equal

refute_in_delta

refute_in_epsilon

refute_includes

refute_instance_of

refute_kind_of

refute_match

refute_nil

refute_operator

refute_respond_to

refute_same

Utility Methods

pass

flunk

skip

mu_pp

capture_io

mu_pp_for_diff

Why all these extra assertions? They're more expressive!

```
assert ! obj
```

```
assert collection.include? obj
```

```
out, err = capture_io do
  do_something
end
assert_equal "output", out
assert_equal "", err
```

```
refute obj
```

```
assert_include collection, obj
```

```
assert_output "output", "" do
  do_something
end
```

Where is `assert_nothing_raised`?

Same place as `refute_silent`.

refute_silent

This block of code must **print** something.

What it is, I don't care.

How is this assertion of value to anyone?

Assert for the specific **output** you need.

assert_nothing_raised

This block of code must **do** something.

What it is, I don't care.

How is this assertion of value to anyone?

Assert for the specific **result** you need.

Passes Without Impl

```
require "minitest/autorun"
```

```
class TestThingy < MiniTest::Unit::TestCase
  def test_do_the_thing
    assert_nothing_raised do
      end
    end
  end
end
```

Doesn't Validate Behavior

```
require "minitest/autorun"
```

```
class TestThingy < Minitest::Unit::TestCase
  def test_do_the_thing
    assert_nothing_raised do
      do_the_thing
    end
  end
end
```

Do This Instead

```
require "minitest/autorun"
```

```
class TestThingy < MiniTest::Unit::TestCase
  def test_do_the_thing
    assert_equal 42, do_the_thing
  end
end
```

Do This Instead

```
require "minitest/autorun"
```

```
class TestThingy < Minitest::Unit::TestCase
  def test_do_the_thing
    assert_equal 42, do_the_thing
  end
end
```

If this raises, it's an **error**.
It's implied.

Do This Instead

```
require "minitest/autorun"
```

```
class TestThingy < MiniTest::Unit::TestCase
  def test_do_the_thing
    assert_equal 42, do_the_thing
  end
end
```

If this raises, it's an **error**.
It's implied.

Verifies behavior

minitest/spec

Spec Example

```
require "minitest/autorun"  
  
describe Thingy do  
  it "must do the thing" do  
    Thingy.do_the_thing.must_equal 42  
  end  
end
```

Specs Transform:

```
require "minitest/autorun"
```

```
class TestThingy < MiniTest::Unit::TestCase
  def test_0001_must_do_the_thing
    Thingy.do_the_thing.must_equal 42
  end
end
```

Specs Transform:

```
require "minitest/autorun"
```

Simple Subclass

```
class TestThingy < MiniTest::Unit::TestCase
  def test_0001_must_do_the_thing
    Thingy.do_the_thing.must_equal 42
  end
end
```

Specs Transform:

```
require "minitest/autorun"
```

Simple Subclass

```
class TestThingy < MiniTest::Unit::TestCase
  def test_0001_must_do_the_thing
    Thingy.do_the_thing.must_equal 42
  end
end
```

Simple Method

Specs Transform:

```
require "minitest/autorun"
```

Simple Subclass

```
class TestThingy < MiniTest::Unit::TestCase
  def test_0001_must_do_the_thing
    Thingy.do_the_thing.must_equal 42
  end
end
```

Simple Method

Magic
Free!

positive expectations

must_be

must_be_close_to

must_be_empty

must_be_instance_of

must_be_kind_of

must_be_nil

must_be_same_as

must_be_silent

must_be_within_delta

must_be_within_epsilon

must_equal

must_include

must_match

must_output

must_raise

must_respond_to

must_send

must_throw

negative expectations

wont_be

wont_be_within_delta

wont_be_close_to

wont_be_within_epsilon

wont_be_empty

wont_equal

wont_be_instance_of

wont_include

wont_be_kind_of

wont_match

wont_be_nil

wont_respond_to

wont_be_same_as

All for Free

must_equal **is** assert_equal
wont_equal **is** refute_equal
etc.

minitest/mock

Example

```
@mock = MiniTest::Mock.new
@mock.expect :meaning_of_life, 42
@mock.meaning_of_life # => 42
@mock.verify          # => true
```

But don't use this if
you don't need it.

Overmocking is evil

mock last

- Mocks should be the **last** tool you grab.
- The test should already be **written**.
- It should already **pass**.
- You **only** use mocks to detach from slow or unstable external resources.

mock high

- Don't mock a socket.
- Mock your reader method.
- Mock your library, not the protocol.

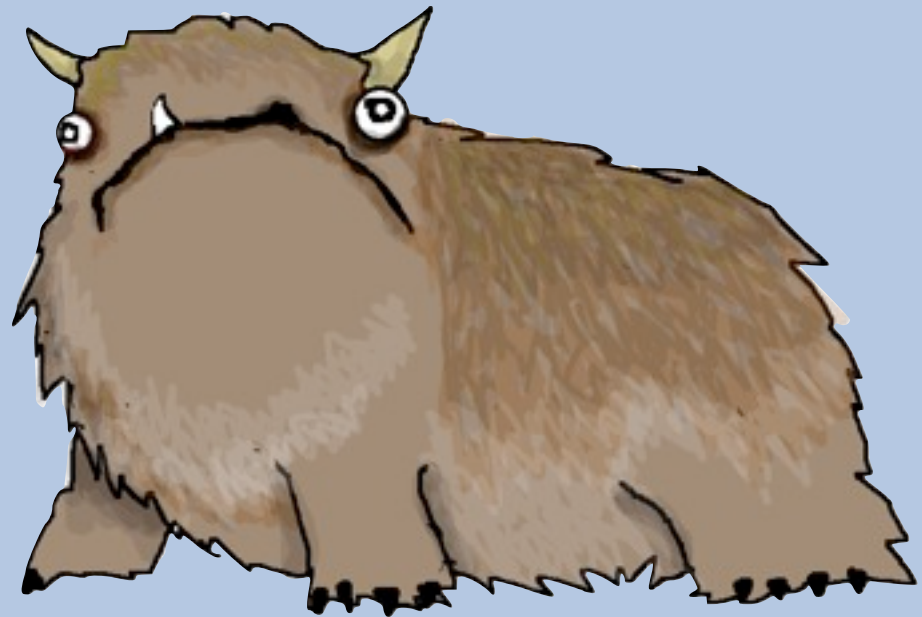
mock smart

- Make sure your tests can fail.
- Otherwise, they're useless.
- I see this *alot*.

mock smart

- Make sure your tests can fail.
- Otherwise, they're useless.
- I see this *alot*.

ALOT



<http://hyperboleandahalf.blogspot.com/2010/04/alot-is-better-than-you-at-everything.html>

“def is my stub framework”

Phil Hagelberg

```
obj = Thing.new
```

```
def obj.timestamp  
  Time.now + 10  
end
```

```
refute obj.done?
```

Use Liskov Substitution Principal (aka testing with subclasses)

```
class IRC  
  def read; ... end  
end
```

```
class TestIRC < IRC  
  def read  
    "happy"  
  end  
end
```

```
assert_equal "happy", @irc.next_line
```

Resist Mocks via Design

Designs that don't need mocking >
Designs that do need mocking

minitest/pride

Implementation

```
class PrideIO
  def initialize io
    @io = io
  end

  def print o
    # ...
  end

  def method_missing msg, *args
    @io.send msg, *args
  end
end
```

```
MiniTest::Unit.output = PrideIO.new(MiniTest::Unit.output)
```

Other Pipeline Ideas

- Plug it into a GUI.
- Emit to Growl or other notifiers.
- Record test statistics.

minitest/bench

minitest/unit example:

```
require 'minitest/benchmark' if ENV["BENCH"]

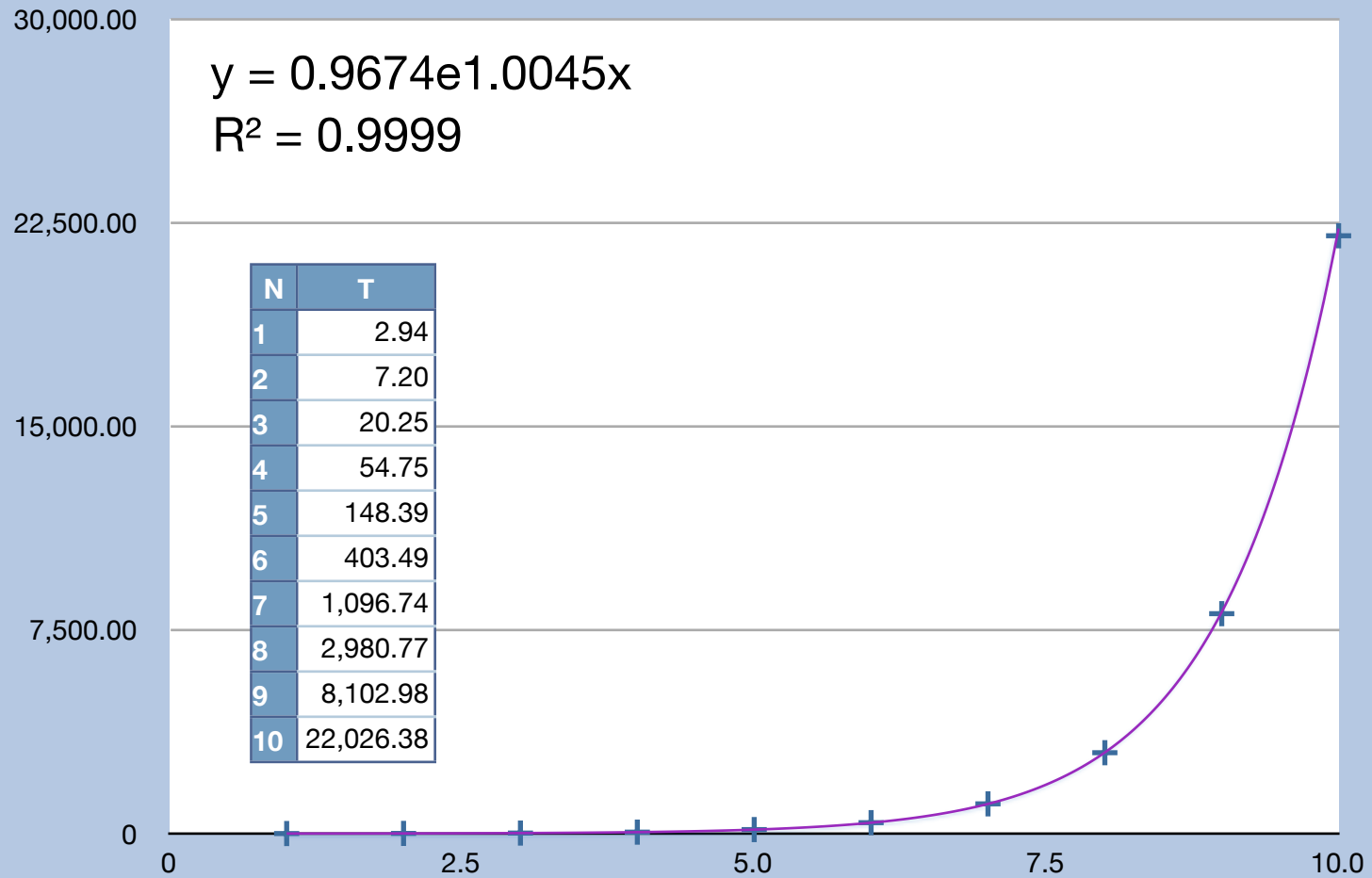
def bench_algorithm
  assert_performance_linear 0.9999 do |x|
    @obj.algorithm
  end
end
```

minitest/spec example:

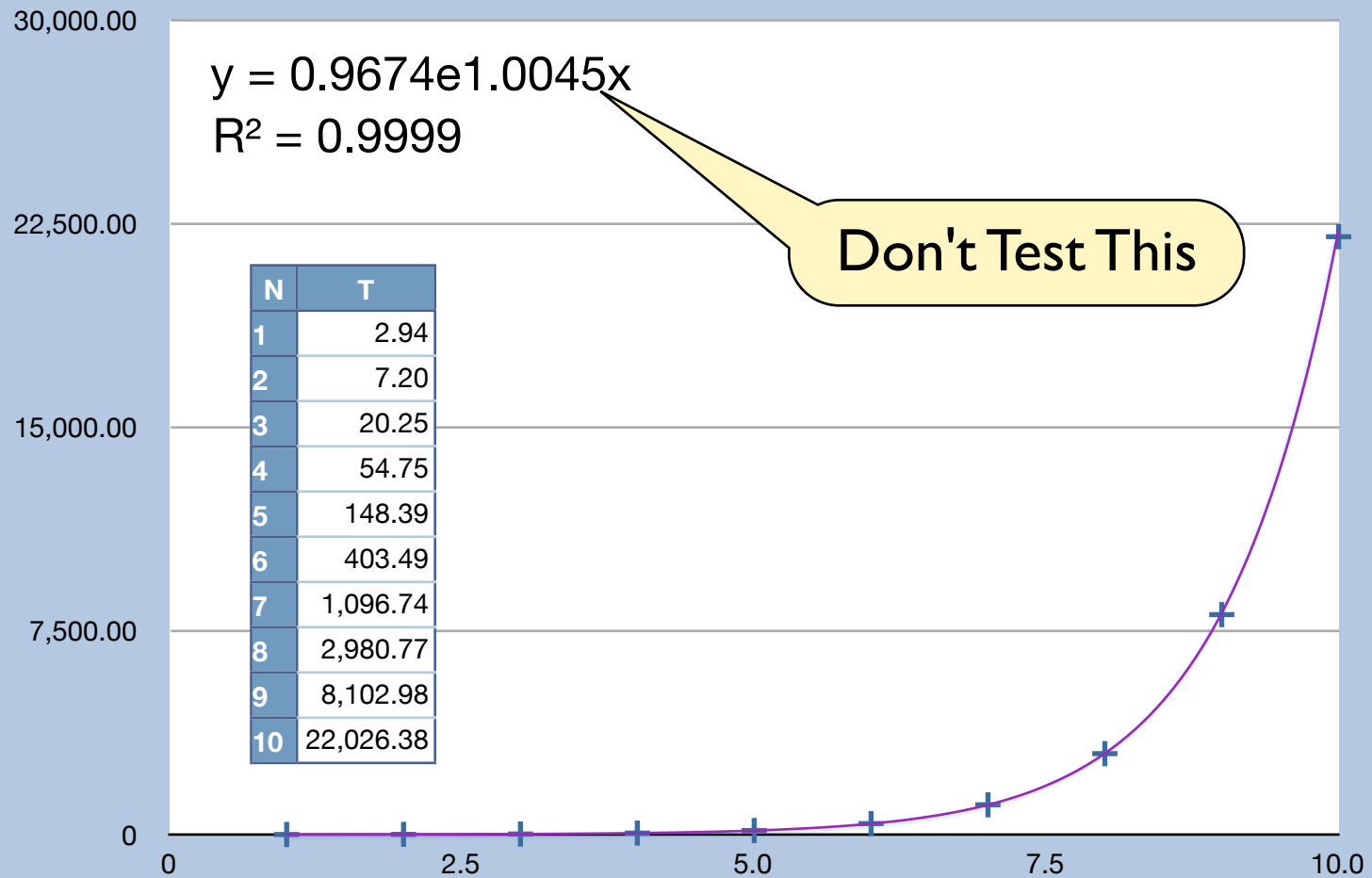
```
require 'minitest/benchmark' if ENV["BENCH"]

describe "my class" do
  if ENV["BENCH"] then
    bench_performance_linear "fast_algorithm", 0.9999 do
      @obj.fast_algorithm
    end
  end
end
```

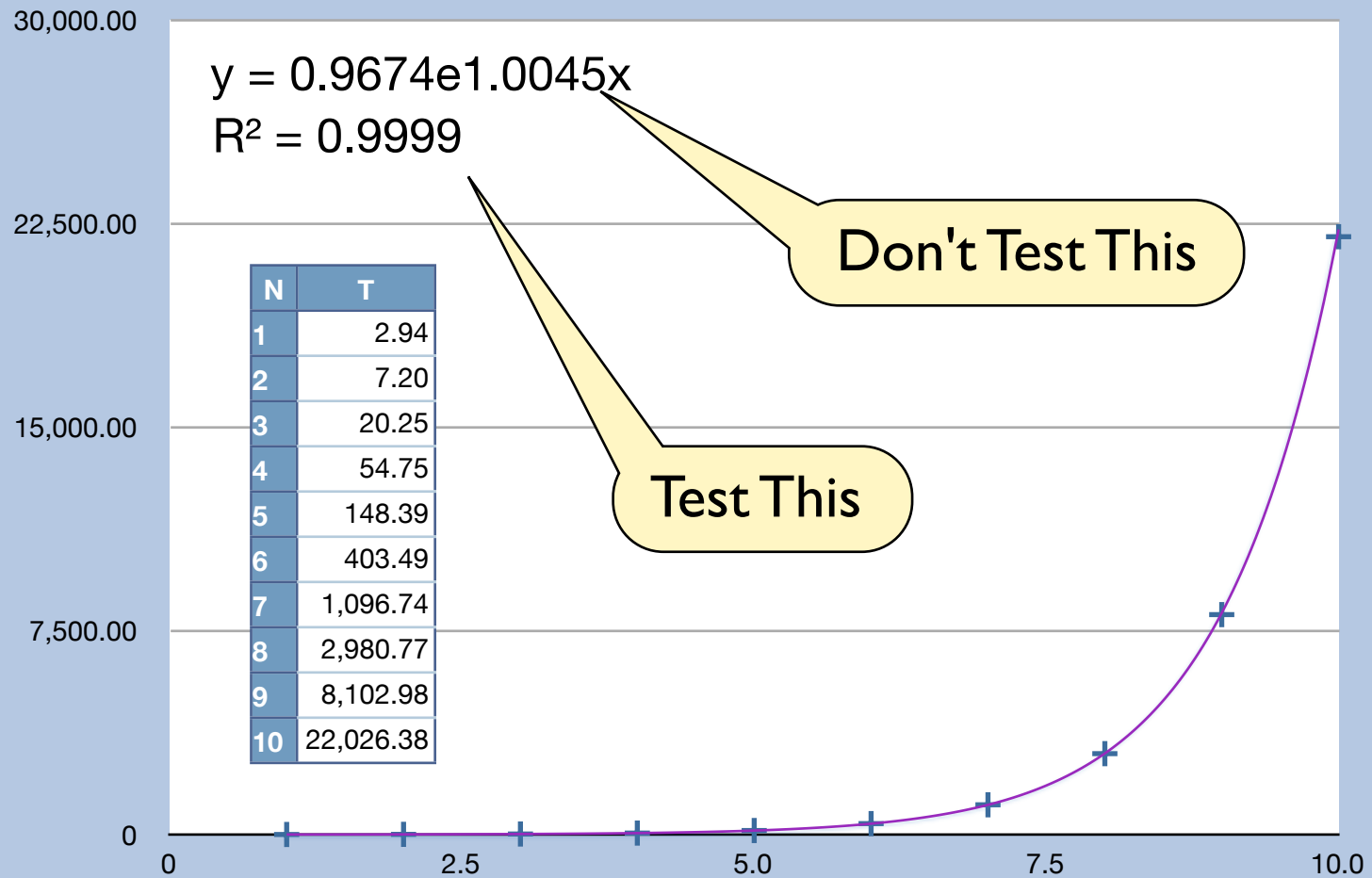
Curve Fitting



Curve Fitting



Curve Fitting



minitest/unit methods

assert_performance_constant

assert_performance_exponential

assert_performance_linear

assert_performance_power

assert_performance - Write your own!

minitest/spec methods

bench_performance_constant

bench_performance_exponential

bench_performance_linear

bench - Write your own!

Extending

Bacon

```
palindrome = lambda { |obj| obj == obj.reverse }  
  
it 'should be a palindrome' do  
  @ary.should.be.a palindrome  
end
```

minitest/unit

```
def assert_palindrome obj
  assert_equal obj, obj.reverse
end
```

```
assert_palindrome @ary
```

minitest/spec

```
def must_be_palindrome  
  self.must_equal self.reverse  
end
```

or:

```
infect_an_assertion(:assert_palindrome,  
                    :must_be_palindrome)
```

```
@ary.must_be_palindrome
```

Other Examples

- rubygems uses:
 - `assert_path_exists`
 - `refute_path_exists`
 - `assert_satisfied_by ver, req`
 - `assert_resolve expected, *specs`

Minitest Design Rationale

Less is More

Indirection is the Enemy

minitest/unit: assert_in_delta

```
def assert_in_delta exp, act, delta = 0.001, msg = nil
  n = (exp - act).abs
  msg = message(msg) {
    "Expected #{exp} - #{act} (#{n}) to be < #{delta}"
  }
  assert delta >= n, msg
end
```

minitest/unit: assert_in_delta

```
def assert_in_delta exp, act, delta = 0.001, msg = nil
  n = (exp - act).abs
  msg = message(msg) {
    "Expected #{exp} - #{act} (#{n}) to be < #{delta}"
  }
  assert delta >= n, msg
end
```

message takes a block to defer calculation until an assertion fails

minitest/unit: assert_in_delta

```
def assert_in_delta exp, act, delta = 0.001, msg = nil
  n = (exp - act).abs
  msg = message(msg) {
    "Expected #{exp} - #{act} (#{n}) to be < #{delta}"
  }
  assert delta >= n, msg
end
```

simple assertion is
all that is needed

message takes a
block to defer
calculation until an
assertion fails

minitest/unit: assert_in_delta

```
def assert_in_delta exp, act, delta = 0.001, msg = nil
  n = (exp - act).abs
  msg = message(msg) {
    "Expected #{exp} - #{act} (#{n}) to be < #{delta}"
  }
  assert delta >= n, msg
end
```

Only 2 other methods
need to be understood:
assert (9) & message (6)

simple assertion is
all that is needed

message takes a
block to defer
calculation until an
assertion fails

minitest/spec: n.must_be_close_to m

(This page intentionally left blank)

bacon:

n.should.be.close m, delta

```
class Object
  def should(*args, &block)
    Should.new(self).be(*args, &block)
  end
end

class Should
  def method_missing(name, *args, &block)
    name = "#{name}?" if name.to_s =~ /\w[^\?]\z/
    desc = # ...

    satisfy(desc) { |x| x.__send__(name, *args, &block) }
  end
end

class Numeric
  def close?(to, delta)
    (to.to_f - self).abs <= delta.to_f rescue false
  end
end
```

bacon:

n.should.be.close m, delta

I. Object#should

```
class Object
  def should(*args, &block)
    Should.new(self).be(*args, &block)
  end
end

class Should
  def method_missing(name, *args, &block)
    name = "#{name}?" if name.to_s =~ /\w[^\?]\z/
    desc = # ...

    satisfy(desc) { |x| x.__send__(name, *args, &block) }
  end
end

class Numeric
  def close?(to, delta)
    (to.to_f - self).abs <= delta.to_f rescue false
  end
end
```

bacon:

n.should.be.close m, delta

```
class Object
```

```
  def should(*args, &block)
    Should.new(self).be(*args, &block)
```

```
  end
```

```
end
```

1. Object#should

2. Should#new

```
class Should
```

```
  def method_missing(name, *args, &block)
    name = "#{name}?" if name.to_s =~ /\w[^\?]\z/
    desc = # ...
```

```
    satisfy(desc) { |x| x.__send__(name, *args, &block) }
```

```
  end
```

```
end
```

```
class Numeric
```

```
  def close?(to, delta)
    (to.to_f - self).abs <= delta.to_f rescue false
```

```
  end
```

```
end
```

bacon:

n.should.be.close m, delta

```
class Object
```

```
  def should(*args, &block)
    Should.new(self).be(*args, &block)
```

```
  end
```

```
end
```

1. Object#should

2. Should#new

```
class Should
```

```
  def method_missing(name, *args, &block)
    name = "#{name}?" if name.to_s =~ /\w[^\?]\z/
    desc = # ...
```

3. Should#close

```
    satisfy(desc) { |x| x.__send__(name, *args, &block) }
```

```
  end
```

```
end
```

```
class Numeric
```

```
  def close?(to, delta)
    (to.to_f - self).abs <= delta.to_f rescue false
```

```
  end
```

```
end
```

bacon:

n.should.be.close m, delta

```
class Object
```

```
  def should(*args, &block)
    Should.new(self).be(*args, &block)
```

```
  end
```

```
end
```

1. Object#should

2. Should#new

```
class Should
```

```
  def method_missing(name, *args, &block)
    name = "#{name}?" if name.to_s =~ /\w[^\?]\z/
    desc = # ...
```

3. Should#close

4. Should#satisfy

```
    satisfy(desc) { |x| x.__send__(name, *args, &block) }
```

```
  end
```

```
end
```

```
class Numeric
```

```
  def close?(to, delta)
    (to.to_f - self).abs <= delta.to_f rescue false
```

```
  end
```

```
end
```

bacon:

n.should.be.close m, delta

```
class Object
```

```
  def should(*args, &block)
    Should.new(self).be(*args, &block)
```

```
  end
```

```
end
```

1. Object#should

2. Should#new

```
class Should
```

```
  def method_missing(name, *args, &block)
    name = "#{name}?" if name.to_s =~ /\w[^\s]\z/
    desc = # ...
```

3. Should#close

4. Should#satisfy

```
    satisfy(desc) { |x| x.__send__(name, *args, &block) }
```

```
  end
```

```
end
```

5. Numeric#close?

```
class Numeric
```

```
  def close?(to, delta)
    (to.to_f - self).abs <= delta.to_f rescue false
```

```
  end
```

```
end
```

bacon:

n.should.be.close m, delta

```
class Object
```

```
  def should(*args, &block)
    Should.new(self).be(*args, &block)
```

```
  end
```

```
end
```

1. Object#should

2. Should#new

```
class Should
```

```
  def method_missing(name, *args, &block)
    name = "#{name}?" if name.to_s =~ /\w[^\?]\z/
    desc = # ...
```

3. Should#close

4. Should#satisfy

```
    satisfy(desc) { |x| x.__send__(name, *args, &block) }
```

```
  end
```

```
end
```

5. Numeric#close?

```
class Numeric
```

```
  def close?(to, delta)
    (to.to_f - self).abs <= delta.to_f rescue false
```

```
  end
```

```
end
```

Only 7 methods really need to be understood (50 loc):
Kernel.describe (3); Context: it (6), should (7); Should: be (8) satisfy (14), method_missing (9); Numeric.close? (3)

test/unit v1: assert_in_delta

```
public
def assert_in_delta(expected_float, actual_float, delta, message="")
  _wrap_assertion do
    {expected_float => "first float", actual_float => "second float", delta => "delta"}.each do |float, name|
      assert_respond_to(float, :to_f, "The arguments must respond to to_f; the #{name} did not")
    end
    assert_operator(delta, :>=, 0.0, "The delta should not be negative")
    full_message = build_message(message, <<EOT, expected_float, actual_float, delta)
    <?> and
    <?> expected to be within
    <?> of each other.
    EOT
    assert_block(full_message) { (expected_float.to_f - actual_float.to_f).abs <= delta.to_f }
  end
end
```

test/unit v1: assert_in_delta

```
public
def assert_in_delta(expected_float, actual_float, delta, message="")
  _wrap_assertion do
    {expected_float => "first float", actual_float => "second float", delta => "delta"}.each do |float, name|
      assert_respond_to(float, :to_f, "The arguments must respond to to_f; the #{name} did not")
    end
    assert_operator(delta, :>=, 0.0, "The delta should not be negative")
    full_message = build_message(message, <<EOT, expected_float, actual_float, delta)
    <?> and
    <?> expected to be within
    <?> of each other.
    EOT
    assert_block(full_message) { (expected_float.to_f - actual_float.to_f).abs <= delta.to_f }
  end
end
```

Need to understand 160 loc:

```
_wrap_assertion (14), assert_respond_to (15), assert_operator (12),
build_message (4+94:AssertionMessage,AssertionMessage::Literal,
AssertionMessage::Template and many methods), assert_block (7)
```

test/unit v2: assert_in_delta

```
public
def assert_in_delta(expected_float, actual_float, delta, message="")
  _wrap_assertion do
    _assert_in_delta_validate_arguments(expected_float,
                                       actual_float,
                                       delta)

    full_message = _assert_in_delta_message(expected_float,
                                             actual_float,
                                             delta,
                                             message)

    assert_block(full_message) do
      (expected_float.to_f - actual_float.to_f).abs <= delta.to_f
    end
  end
end
```

test/unit v2: assert_in_delta

```
public
def assert_in_delta(expected_float, actual_float, delta, message="")
  _wrap_assertion do
    _assert_in_delta_validate_arguments(expected_float,
                                       actual_float,
                                       delta)

    full_message = _assert_in_delta_message(expected_float,
                                           actual_float,
                                           delta,
                                           message)

    assert_block(full_message) do
      (expected_float.to_f - actual_float.to_f).abs <= delta.to_f
    end
  end
end
```

Looks better, but now you need to understand ~308 loc:

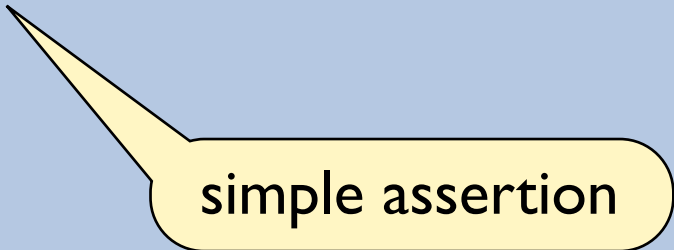
```
_assert_in_delta_validate_arguments (14), _assert_in_delta_validate_arguments (35),
_wrap_assertion (14), assert_respond_to (16), assert_operator (12), build_message
(4+192!), assert_block (7)
```

Simplified test/unit: assert_in_delta

```
public
def assert_in_delta(expected_float, actual_float, delta, message="")
  _wrap_assertion do
    assert_respond_to(expected_float, :to_f, "The arguments must respond to to_f; the first did not")
    assert_respond_to(actual_float, :to_f, "The arguments must respond to to_f; the second did not")
    assert_respond_to(delta, :to_f, "The arguments must respond to to_f; the delta did not")
    assert_operator(delta, :>=, 0.0, "The delta should not be negative")
    assert_operator(expected_float.to_f - actual_float.to_f).abs, :<=, delta.to_f, message
  end
end
```

Simplified test/unit: assert_in_delta

```
public
def assert_in_delta(expected_float, actual_float, delta, message="")
  _wrap_assertion do
    assert_respond_to(expected_float, :to_f, "The arguments must respond to to_f; the first did not")
    assert_respond_to(actual_float, :to_f, "The arguments must respond to to_f; the second did not")
    assert_respond_to(delta, :to_f, "The arguments must respond to to_f; the delta did not")
    assert_operator(delta, :>=, 0.0, "The delta should not be negative")
    assert_operator(expected_float.to_f - actual_float.to_f).abs, :<=, delta.to_f, message
  end
end
```



simple assertion

Simplified test/unit: assert_in_delta

```
public
def assert_in_delta(expected_float, actual_float, delta, message="")
  _wrap_assertion do
    assert_respond_to(expected_float, :to_f, "The arguments must respond to to_f; the first did not")
    assert_respond_to(actual_float, :to_f, "The arguments must respond to to_f; the second did not")
    assert_respond_to(delta, :to_f, "The arguments must respond to to_f; the delta did not")
    assert_operator(delta, :>=, 0.0, "The delta should not be negative")
    assert_operator(expected_float.to_f - actual_float.to_f).abs, :<=, delta.to_f, message
  end
end
```

simple assertion

no loops

Simplified test/unit: assert_in_delta

```
public
def assert_in_delta(expected_float, actual_float, delta, message="")
  _wrap_assertion do
    assert_respond_to(expected_float, :to_f, "The arguments must respond to to_f; the first did not")
    assert_respond_to(actual_float, :to_f, "The arguments must respond to to_f; the second did not")
    assert_respond_to(delta, :to_f, "The arguments must respond to to_f; the delta did not")
    assert_operator(delta, :>=, 0.0, "The delta should not be negative")
    assert_operator(expected_float.to_f - actual_float.to_f).abs, :<=, delta.to_f, message
  end
end
```

simple assertion

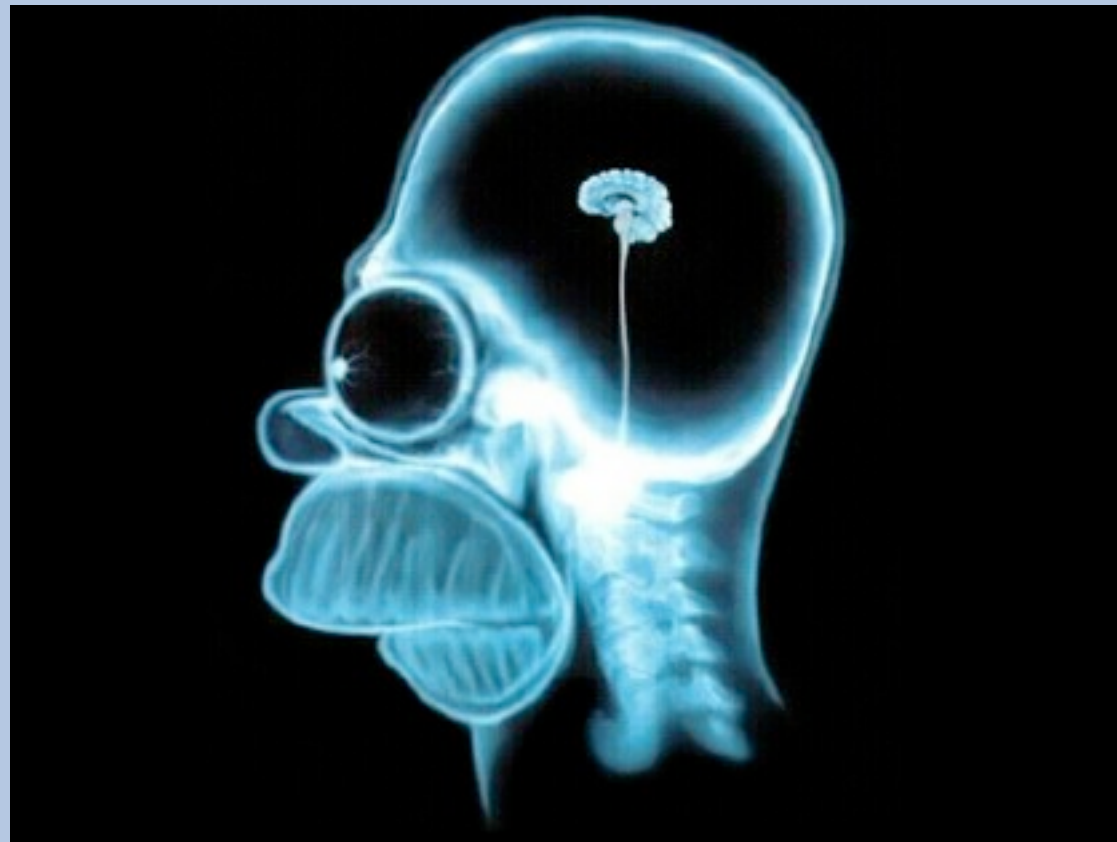
no loops

Need to understand 50 loc:

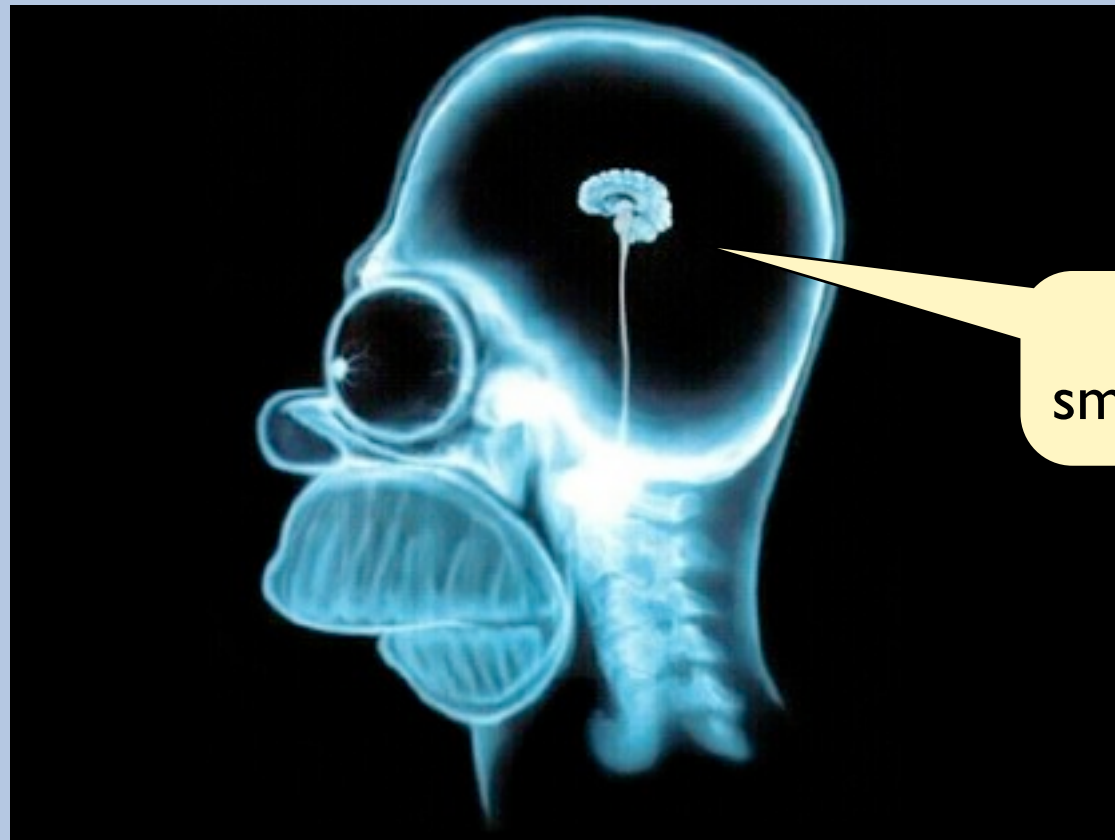
`_wrap_assertion` (14), `assert_respond_to` (15), `assert_operator` (12)

`rspec:`
`n.should be_close(m, d)`

rspec:
n.should be_close(m, d)



rspec:
n.should be_close(m, d)



I'm not smart enough

Test Framework Comparisons

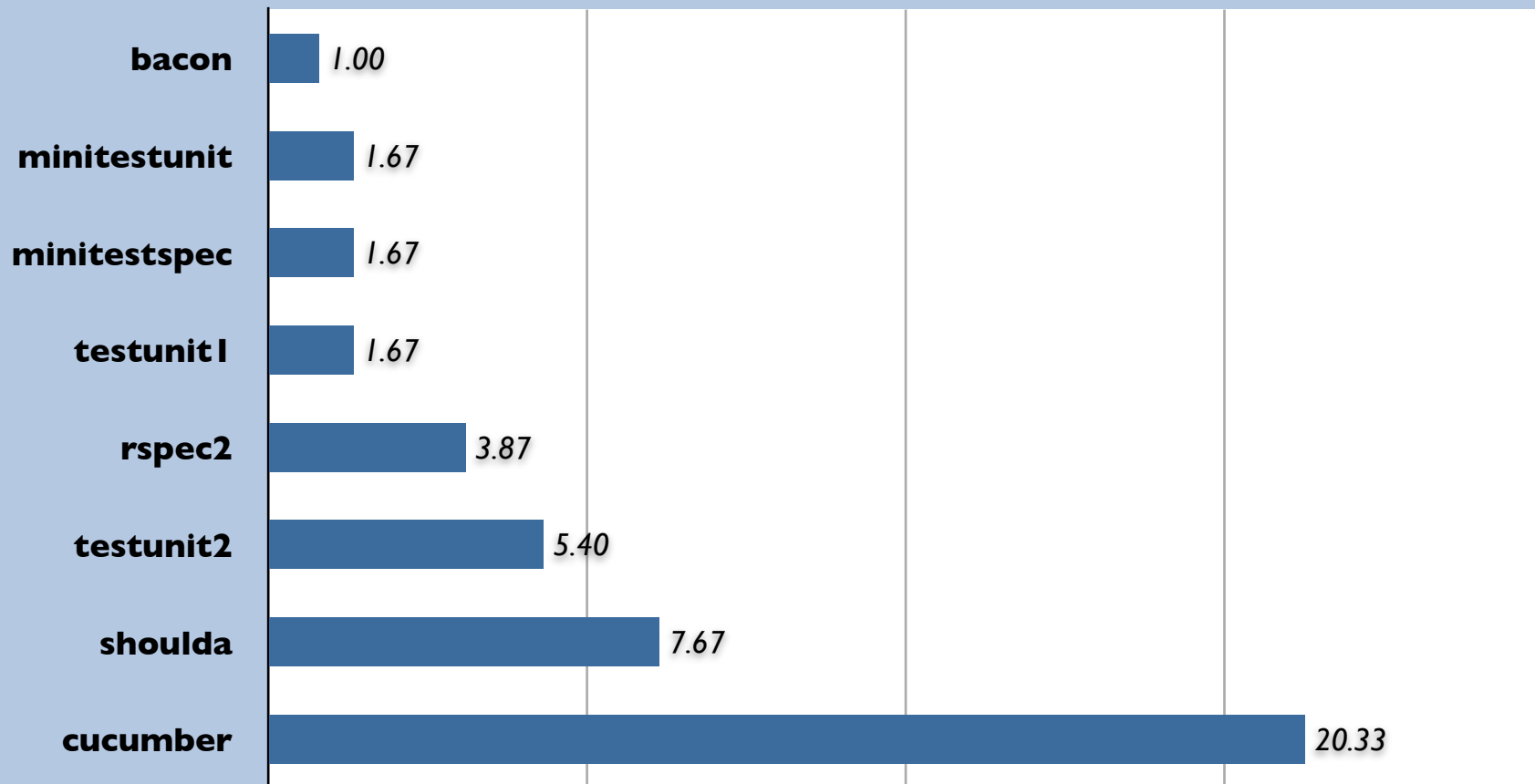
Positive and Negative Tests

(1,000 Tests, each)

framework	pos (s)	multiple	neg (s)	multiple	avg
bacon	0.15	(1.00 x)	0.45	(1.50 x)	(1.25 x)
minitestunit	0.25	(1.67 x)	0.30	(1.00 x)	(1.33 x)
minitestspec	0.25	(1.67 x)	0.38	(1.27 x)	(1.47 x)
rspec	0.58	(3.87 x)	0.63	(2.10 x)	(2.98 x)
rspec1	0.41	(2.73 x)	1.13	(3.77 x)	(3.25 x)
testunit1	0.25	(1.67 x)	1.47	(4.90 x)	(3.28 x)
shoulda	1.15	(7.67 x)	2.45	(8.17 x)	(7.92 x)
testunit2	0.81	(5.40 x)	3.21	(10.70 x)	(8.05 x)
cucumber	3.05	(20.33 x)	3.74	(12.47 x)	(16.40 x)

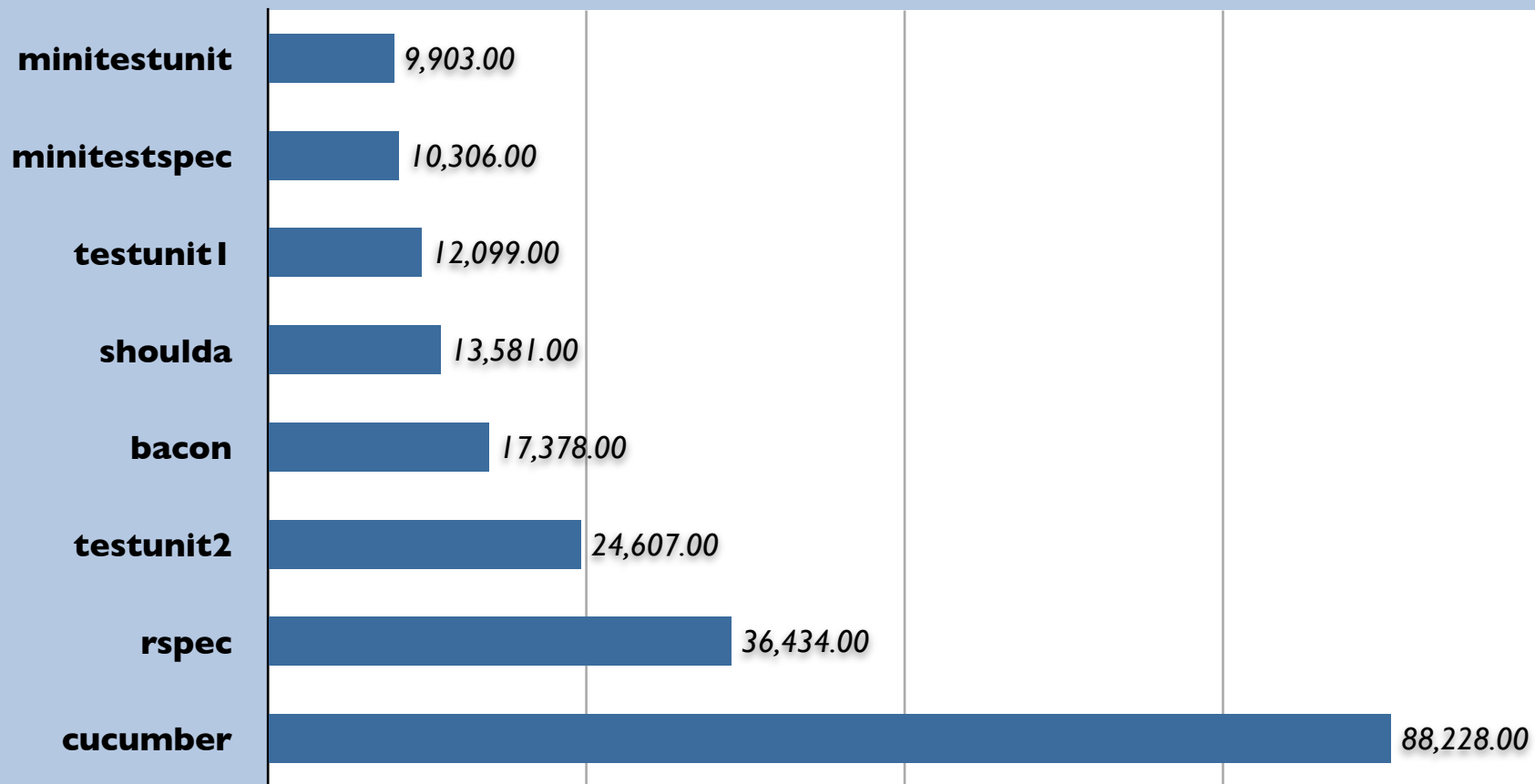
Positive Test Time

(in multiples of bacon) :D



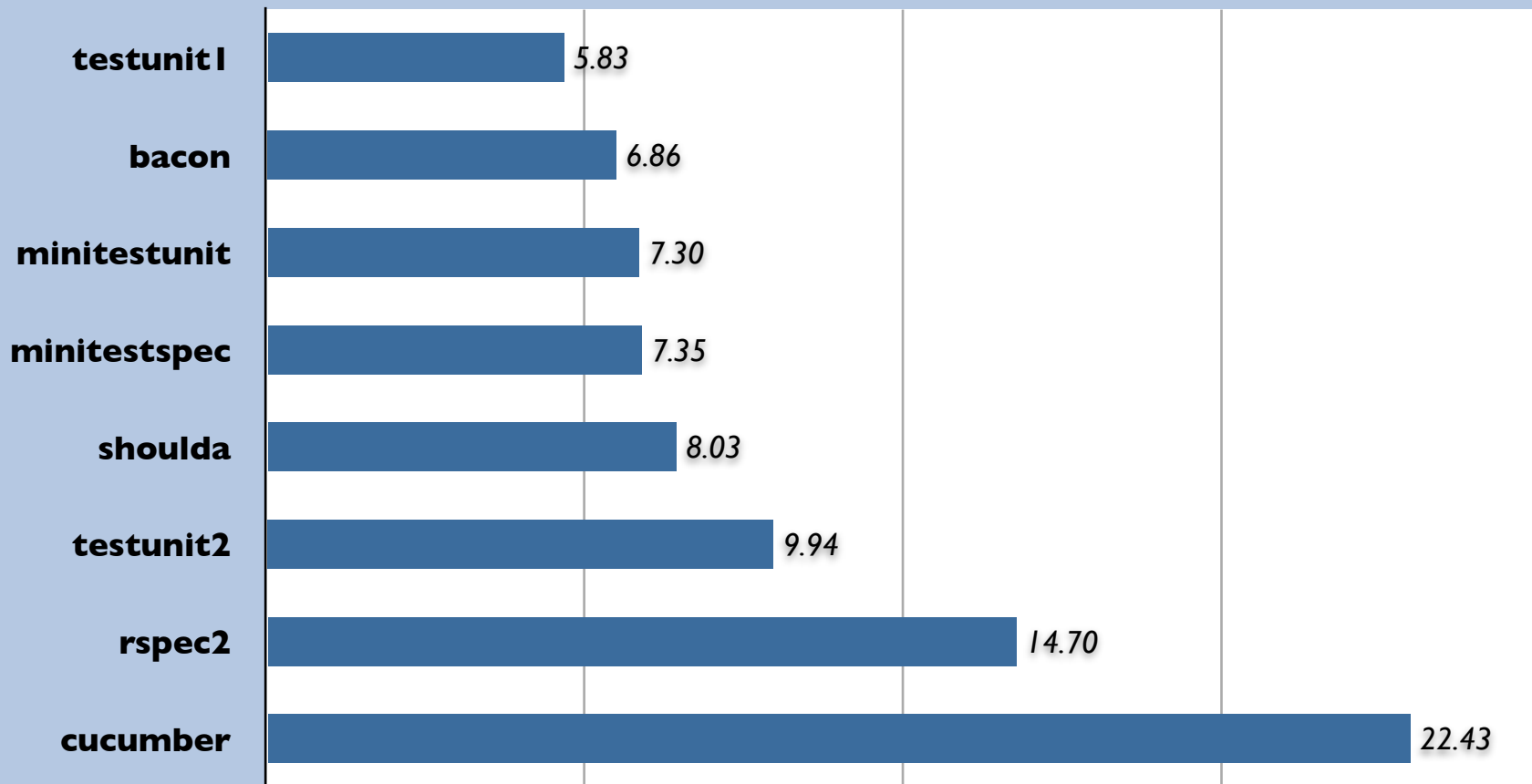
Lines of Code Executed

(trace = 1 test)



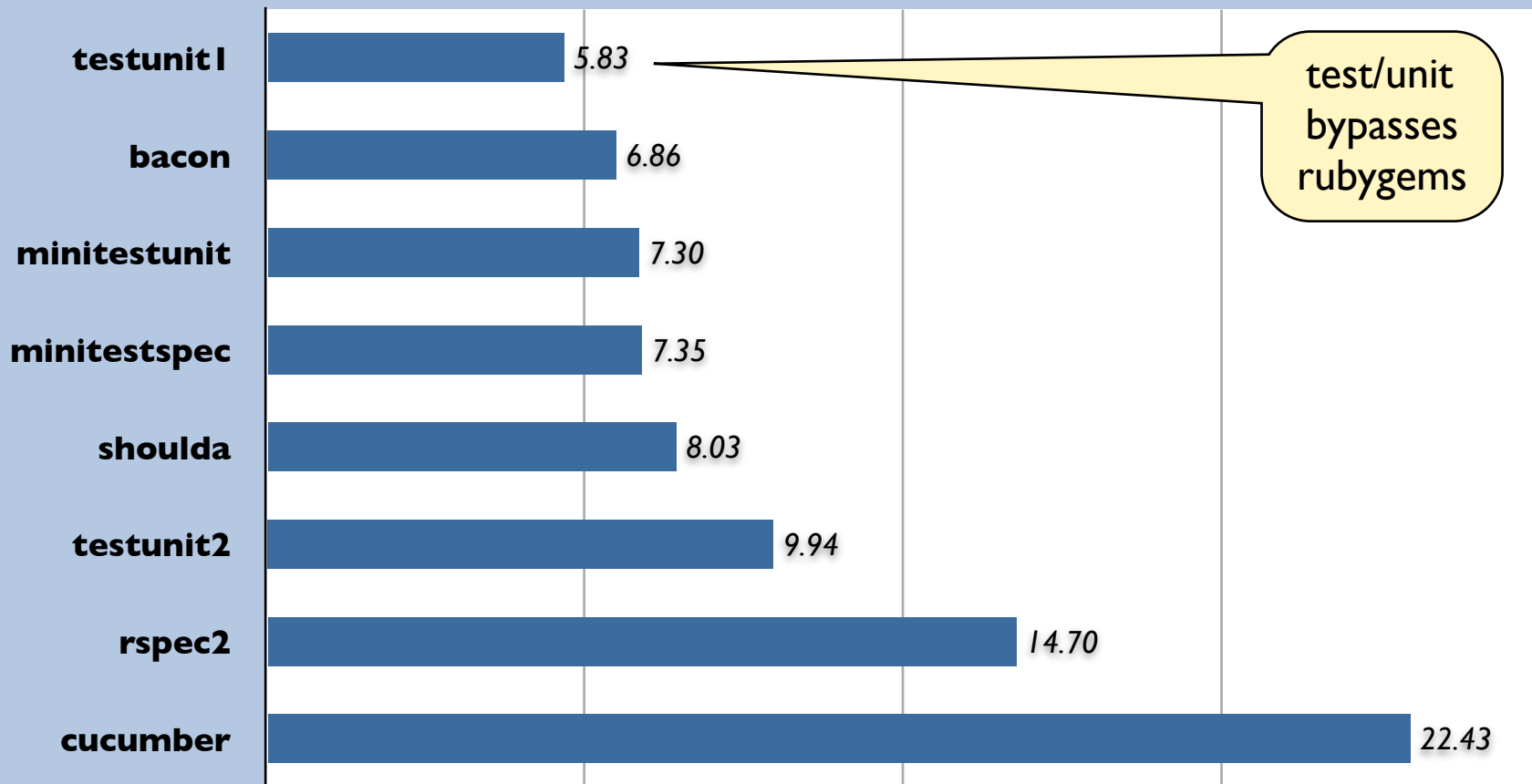
Startup Time

(100 runs w/ 1 test, in sec)



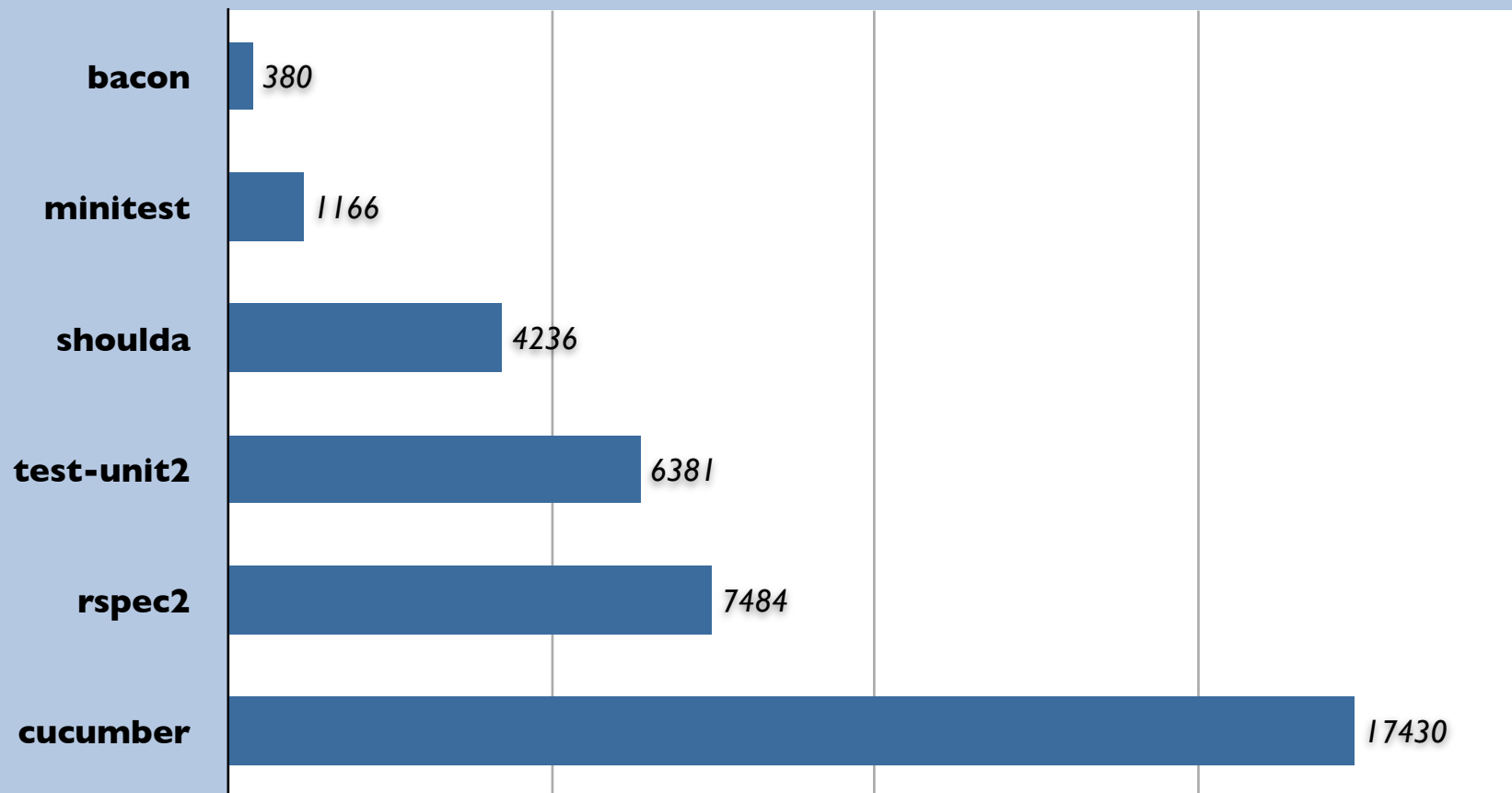
Startup Time

(100 runs w/ 1 test, in sec)



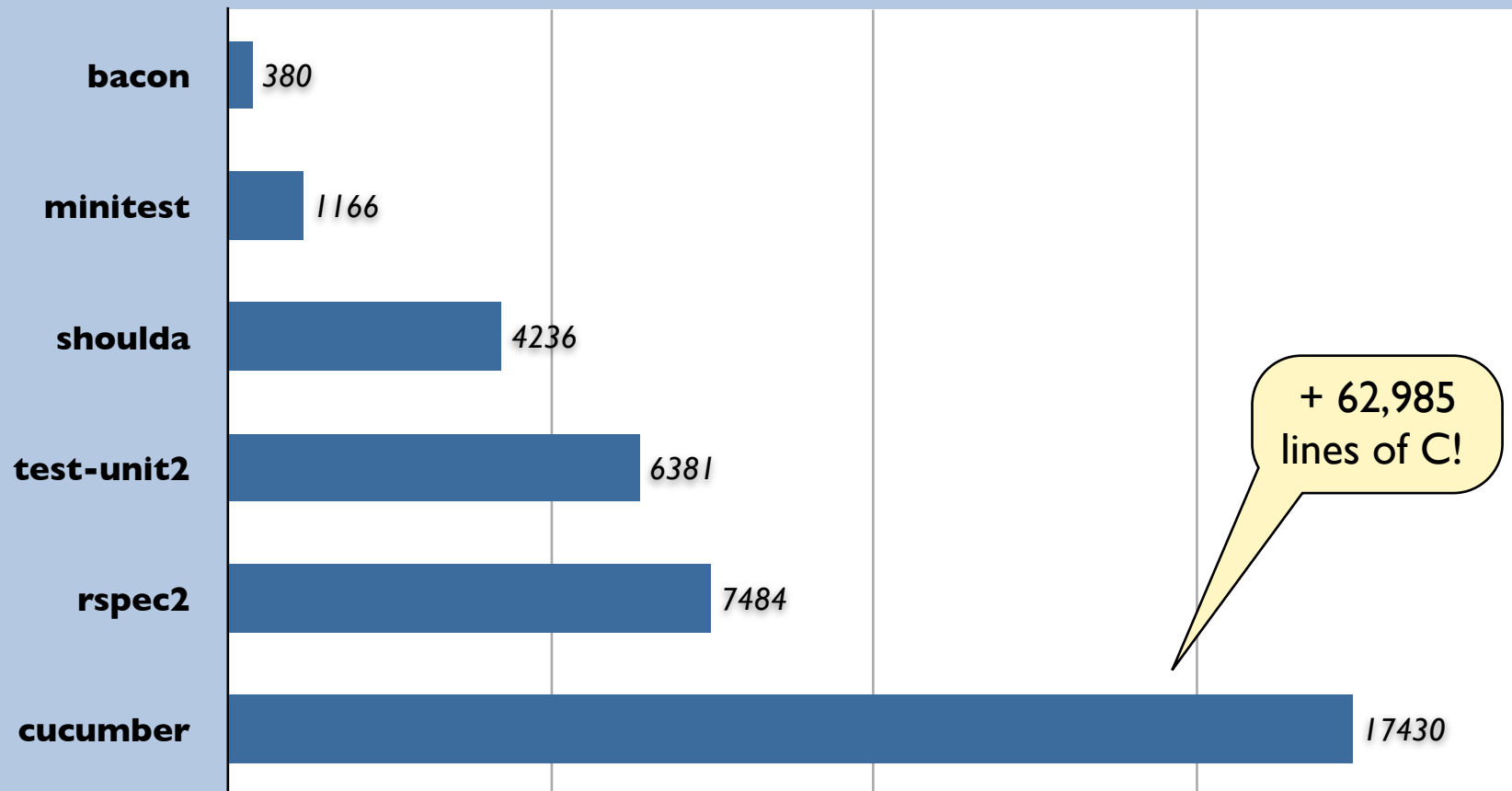
Lines of Code

(including dependencies)



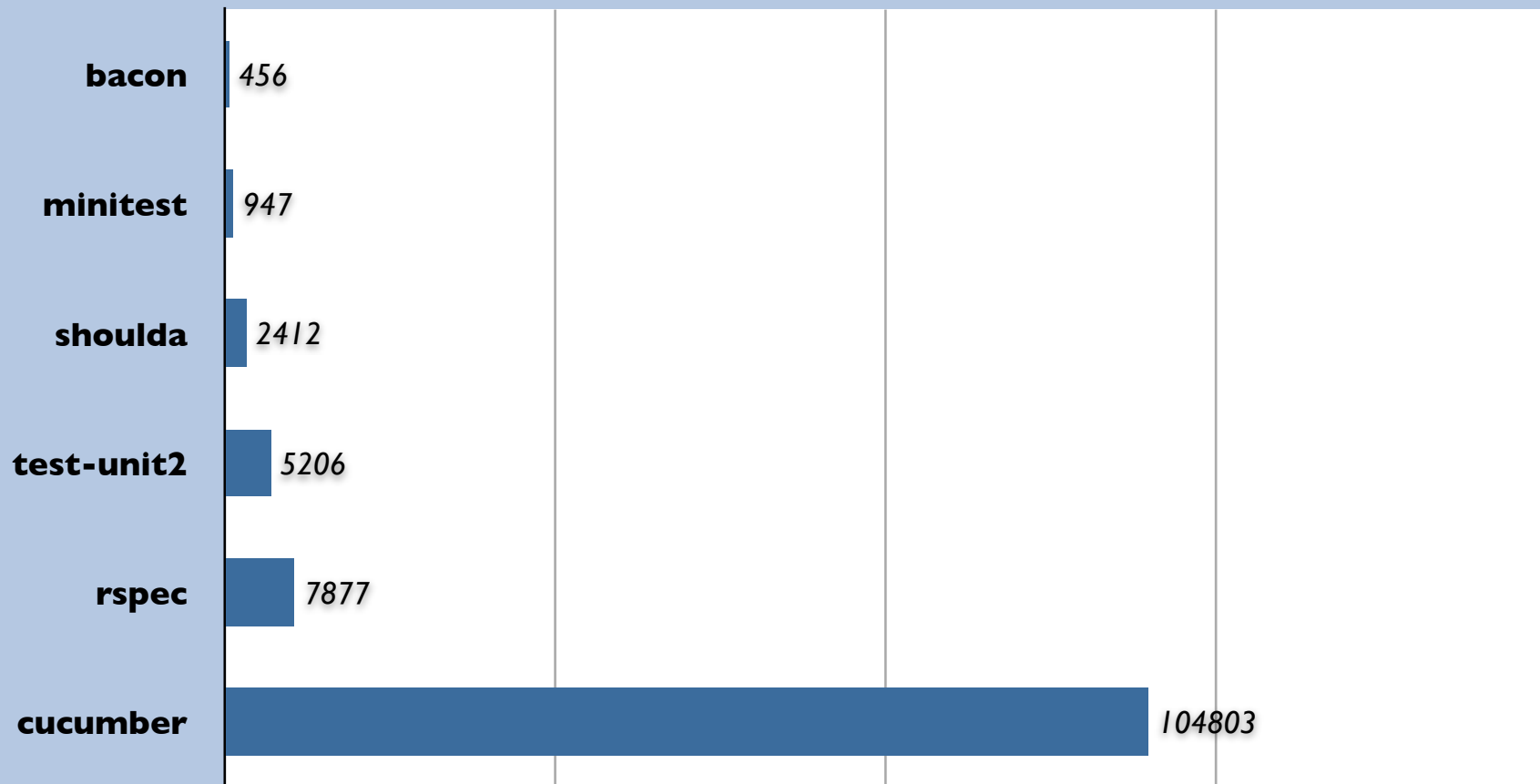
Lines of Code

(including dependencies)



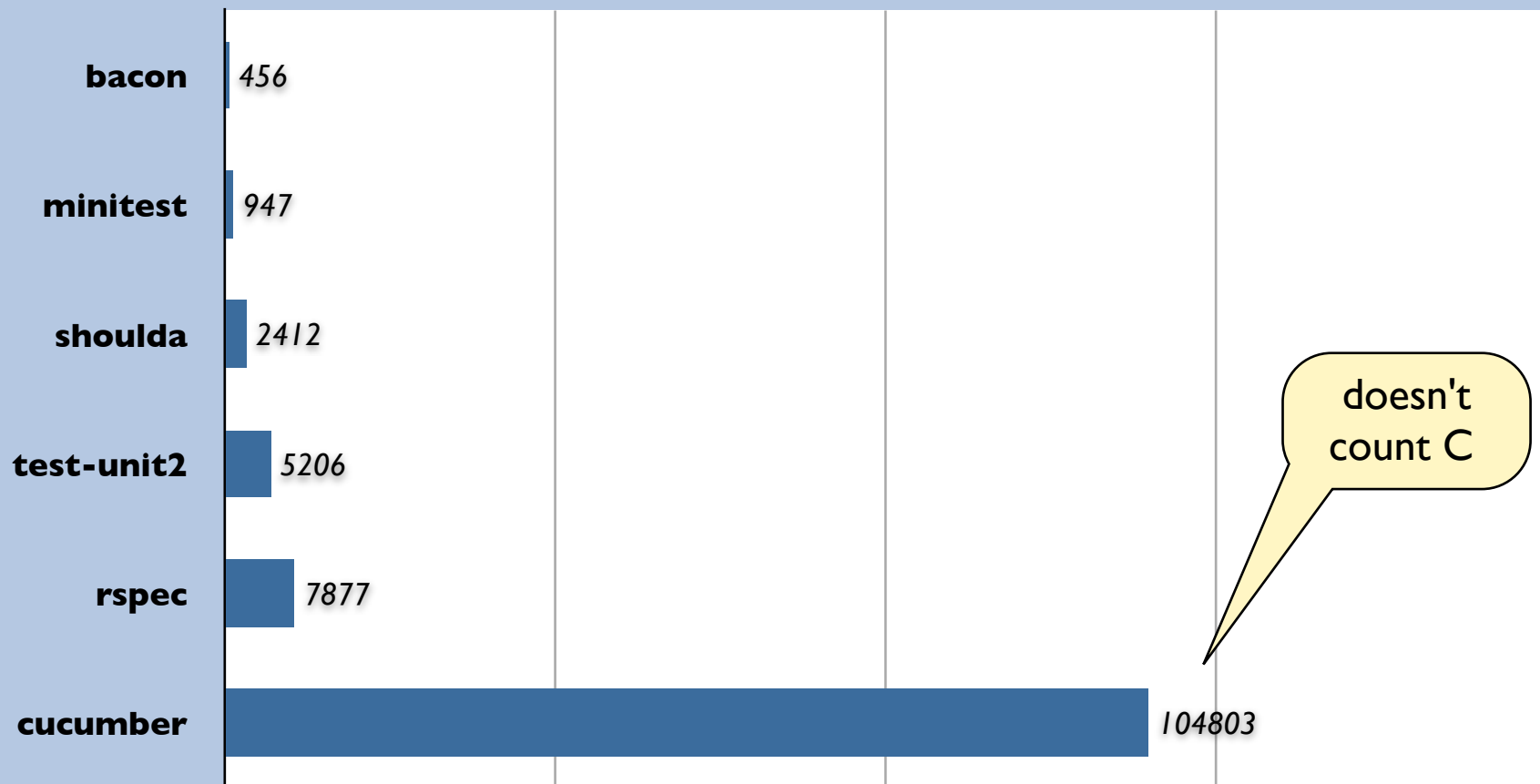
Flog (Complexity) Score

(including dependencies)



Flog (Complexity) Score

(including dependencies)



*All of this is very important to
me...*

But...

But the #1 thing to
improve your life is
not technology!

Stop using pre-ground grocery store black pepper!



Fresh ground
pepper is vastly
better, and cheap!



Finally,
A
Special
Thanks

to
Gregory
Brown



Cascadia Ruby Conf 2011 @ Seattle, Cascadia

and my Foster Kittens



they kept me
sane during the
rubygems
~~bullshit~~ drama

*without them, I
probably wouldn't be
speaking here today*

**Thank
You**