
The other day

```
iex(1)> defmodule RepeatN do
...(1)>   def repeat_n(_function, 0) do
...(1)>     # noop
...(1)>   end
...(1)>   def repeat_n(function, 1) do
...(1)>     function.()
...(1)>   end
...(1)>   def repeat_n(function, count) do
...(1)>     function.()
...(1)>     repeat_n(function, count - 1)
...(1)>   end
...(1)> end
{:module, RepeatN, ...}
```

```
iex(1)> defmodule RepeatN do
...(1)>   def repeat_n(_function, 0) do
...(1)>     # noop
...(1)>   end
...(1)>   def repeat_n(function, 1) do
...(1)>     function.()
...(1)>   end
...(1)>   def repeat_n(function, count) do
...(1)>     function.()
...(1)>     repeat_n(function, count - 1)
...(1)>   end
...(1)> end
{:module, RepeatN, ...}
iex(2)> :timer.tc fn -> RepeatN.repeat_n(fn -> 0 end, 100) end
{210, 0}
```

```
iex(1)> defmodule RepeatN do
...(1)>   def repeat_n(_function, 0) do
...(1)>     # noop
...(1)>   end
...(1)>   def repeat_n(function, 1) do
...(1)>     function.()
...(1)>   end
...(1)>   def repeat_n(function, count) do
...(1)>     function.()
...(1)>     repeat_n(function, count - 1)
...(1)>   end
...(1)> end
{:module, RepeatN, ...}
iex(2)> :timer.tc fn -> RepeatN.repeat_n(fn -> 0 end, 100) end
{210, 0}
iex(3)> list = Enum.to_list(1..100)
[...]
iex(4)> :timer.tc fn -> Enum.each(list, fn(_) -> 0 end) end
{165, :ok}
```

```
iex(1)> defmodule RepeatN do
...(1)>   def repeat_n(_function, 0) do
...(1)>     # noop
...(1)>   end
...(1)>   def repeat_n(function, 1) do
...(1)>     function.()
...(1)>   end
...(1)>   def repeat_n(function, count) do
...(1)>     function.()
...(1)>     repeat_n(function, count - 1)
...(1)>   end
...(1)> end
{:module, RepeatN, ...}
iex(2)> :timer.tc fn -> RepeatN.repeat_n(fn -> 0 end, 100) end
{210, 0}
iex(3)> list = Enum.to_list(1..100)
[...]
iex(4)> :timer.tc fn -> Enum.each(list, fn(_) -> 0 end) end
{165, :ok}
iex(5)> :timer.tc fn -> Enum.each(list, fn(_) -> 0 end) end
{170, :ok}
iex(6)> :timer.tc fn -> Enum.each(list, fn(_) -> 0 end) end
{184, :ok}
```


Success!

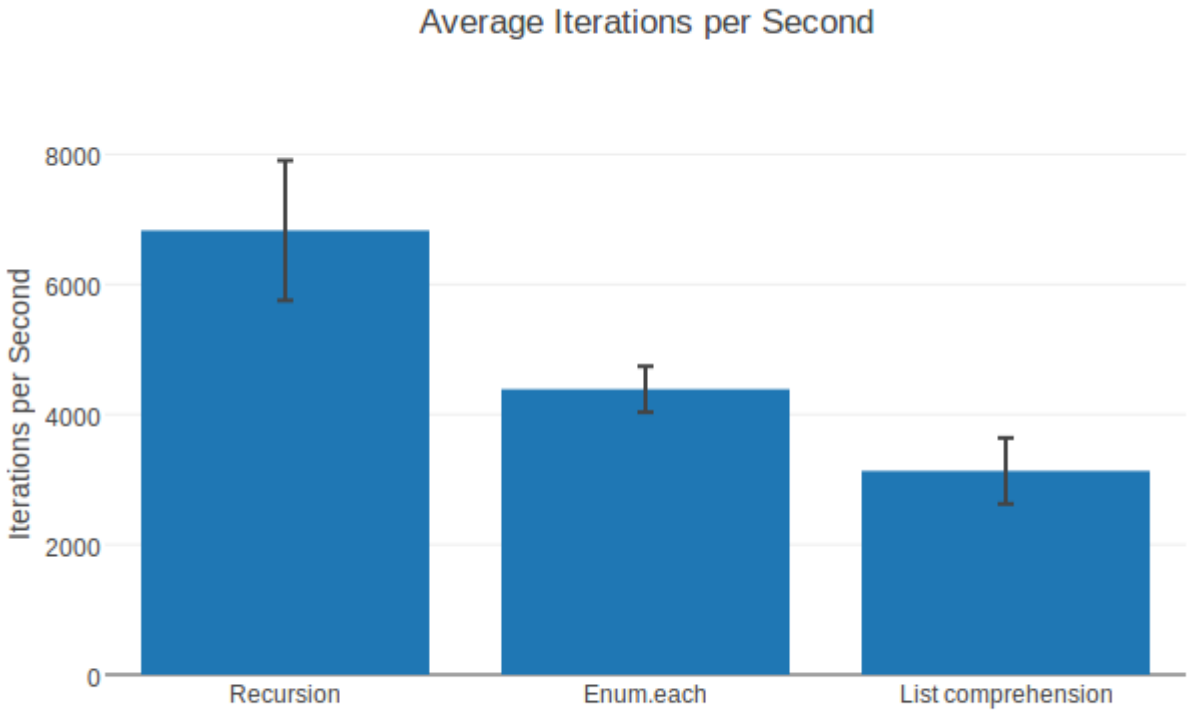
The End?


I HAVE NO
IDEA WHAT
I'M DOING

Numerous Failings!

- Way too few samples
- Realistic data/multiple inputs?
- No warmup
- Non production environment
- Does creating the list matter?
- Is repeating really the bottle neck?
- Repeatability?
- Setup information
- Running on battery
- Lots of applications running

```elixir
n      = 10_000
range = 1..n
list  = Enum.to_list range
fun   = fn -> 0 end

Benchee.run %{
  "Enum.each" =>
    fn -> Enum.each(list, fn(_) -> fun.() end) end,
  "List comprehension" =>
    fn -> for _ <- list, do: fun.() end,
  "Recursion" =>
    fn -> RepeatN.repeat_n(fun, n) end
}
```



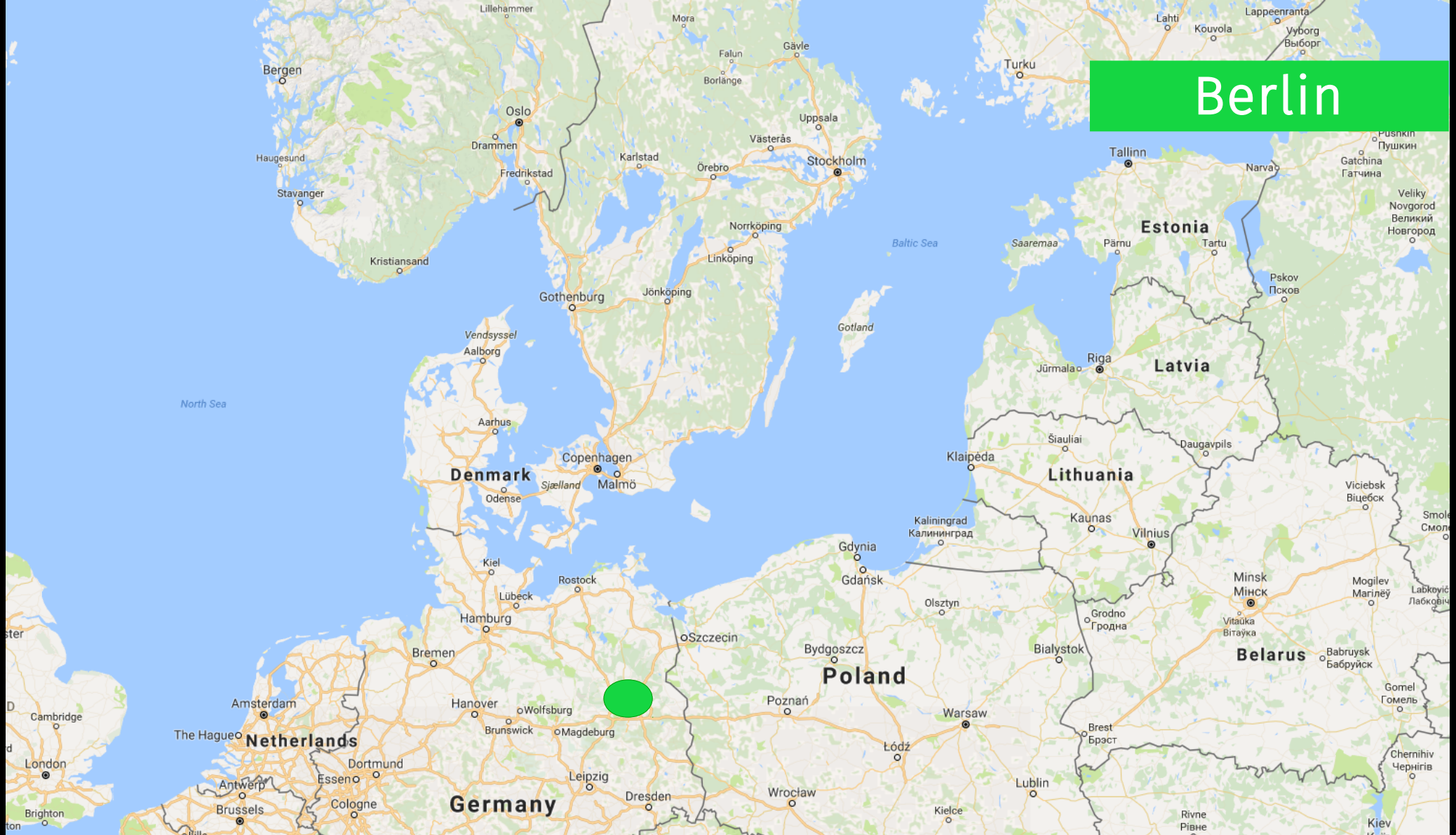

```
Name                           ips        average  deviation         median
Recursion                   6.83 K      146.41 µs    ±15.76%      139.00 µs
Enum.each                   4.39 K      227.86 µs     ±8.05%      224.00 µs
List comprehension          3.13 K      319.22 µs    ±16.20%      323.00 µs

Comparison:
Recursion                   6.83 K
Enum.each                   4.39 K - 1.56x slower
List comprehension          3.13 K - 2.18x slower
```

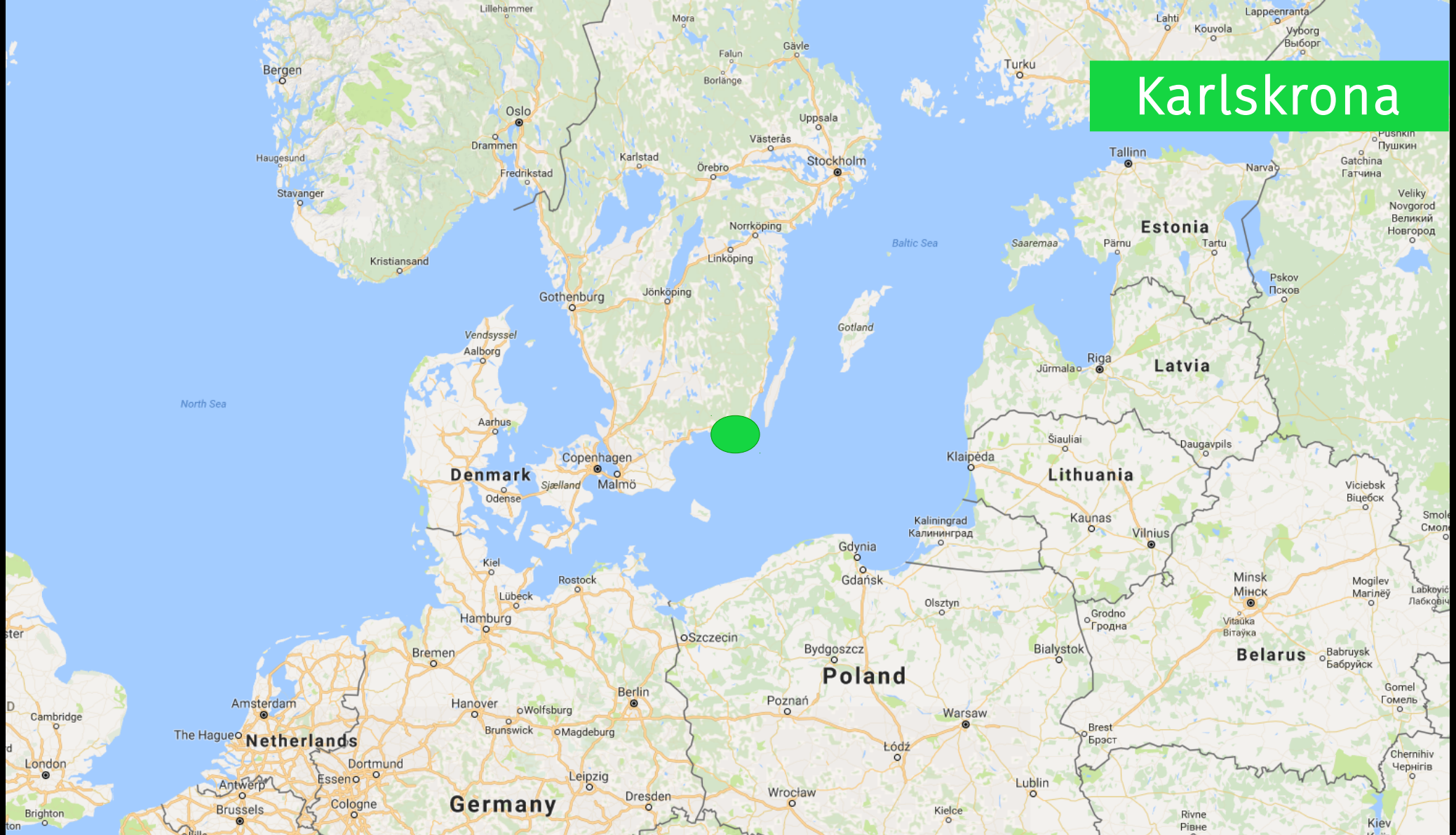
Stop guessing and Start Measuring
Benchmarking in Practice

Tobias Pfeiffer
@PragTob
pragtob.info
github.com/PragTob/benchee

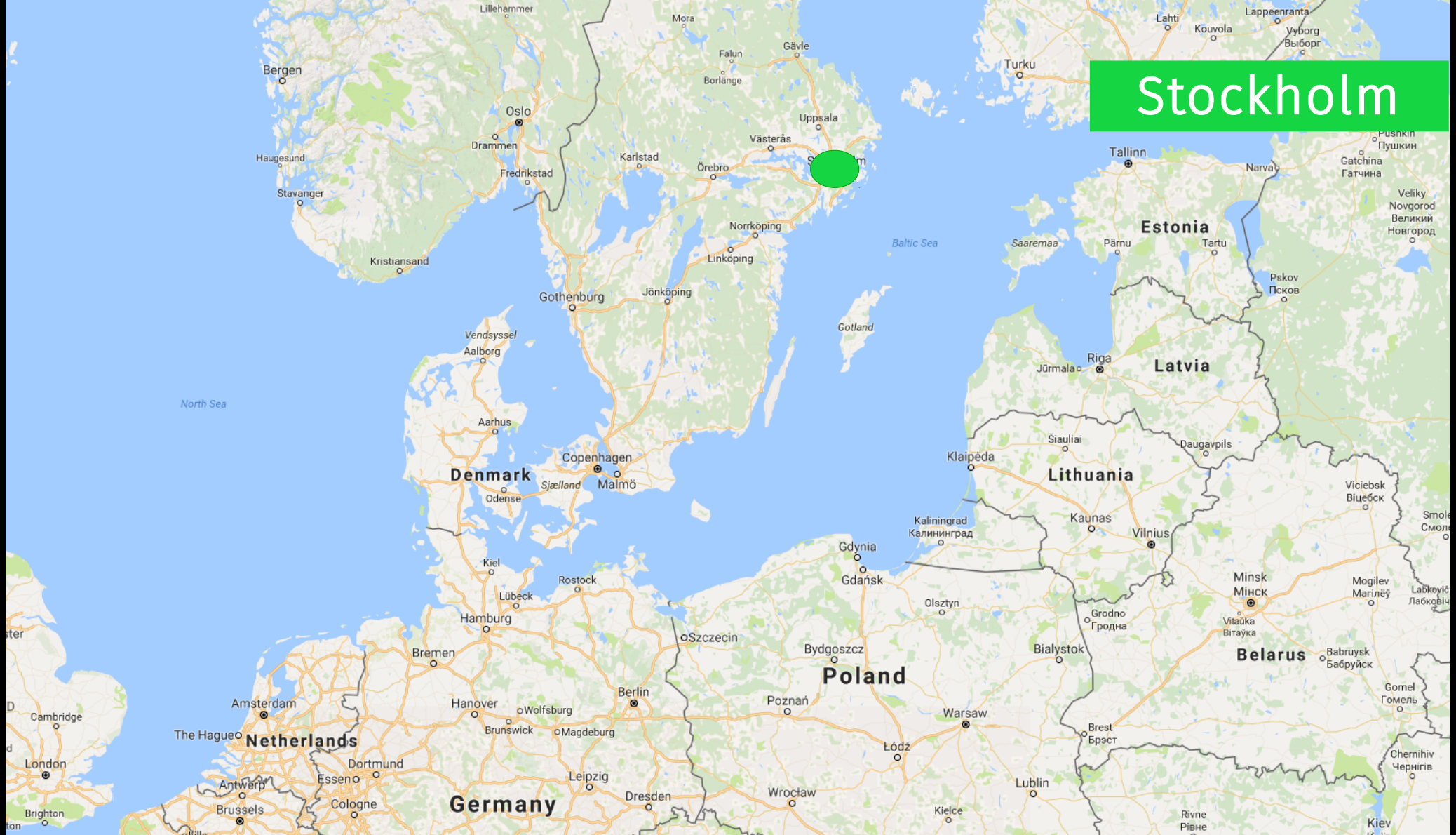
LIEFERY


(broken) Swedish


Berlin
Lillehammer
Mora
Falun
Gävle
Bergen
Oslo
Borlänge
Uppsala
Drammen
Västerås
Haugesund
Karlstad
Örebro
Stockholm
Fredrikstad
Stavanger
Norrköping
Linköping
Kristiansand
Gothenburg
Jönköping
Vendsyssel
Aalborg
North Sea
Aarhus
Copenhagen
Denmark
Sjælland
Malmö
Odense
Kiel
Rostock
Lübeck
Hamburg
Bremen
Amsterdam
Hanover
Wolfsburg
Brunswick
Magdeburg
The Hague
Netherlands
Cambridge
London
Antwerp
Dortmund
Essen
Cologne
Brussels
Brighton
Leipzig
Dresden
Germany
Szczecin
Gdynia
Gdańsk
Olsztyn
Bydgoszcz
Poland
Poznań
Warsaw
Łódź
Wrocław
Kielce
Lublin
Turku
Lahti
Kouvola
Lappeenranta
Vyborg
Выборг
Tallinn
Narva
Pushkin
Пушкин
Gatchina
Гатчина
Veliky Novgorod
Великий Новгород
Estonia
Saaremaa
Pärnu
Tartu
Baltic Sea
Pskov
Псков
Gotland
Riga
Jürmala
Latvia
Šiauliai
Daugavpils
Klaipėda
Lithuania
Kaliningrad
Калининград
Kaunas
Vilnius
Viciebsk
Віцебск
Minsk
Мінск
Mogilev
Магілёў
Grodno
Гродна
Vitaŭka
Вітаўка
Belarus
Bialystok
Babruysk
Бабруйск
Gomel
Гомель
Brest
Брэст
Chernihiv
Чернігів
Rivne
Рівне
Kiev


Karlskrona
Lillehammer
Mora
Falun
Gävle
Bergen
Borlänge
Oslo
Uppsala
Drammen
Västerås
Haugesund
Karlstad
Örebro
Stockholm
Fredrikstad
Stavanger
Norrköping
Linköping
Kristiansand
Gothenburg
Jönköping
Vendsyssel
Aalborg
Gotland
North Sea
Aarhus
Copenhagen
Denmark
Sjælland
Malmö
Odense
Kiel
Rostock
Lübeck
Hamburg
Bremen
Amsterdam
Hanover
Wolfsburg
Berlin
Brunswick
Magdeburg
The Hague
Netherlands
Dortmund
Essen
Antwerp
Cologne
Brussels
Germany
Leipzig
Dresden
Cambridge
London
Brighton
Szczecin
Gdynia
Gdańsk
Olsztyn
Bydgoszcz
Poland
Poznań
Warsaw
Łódź
Wrocław
Kielce
Lublin
Kaliningrad
Калининград
Klaipėda
Šiauliai
Jūrmala
Riga
Latvia
Daugavpils
Lithuania
Kaunas
Vilnius
Grodno
Гродна
Białystok
Minsk
Мінск
Belarus
Vitaŭka
Вітаўка
Brest
Брэст
Lahti
Kouvola
Lappeenranta
Vyborg
Выборг
Turku
Tallinn
Narva
Pushkin
Пушкин
Gatchina
Гатчина
Veliky
Novgorod
Великий
Новгород
Estonia
Pärnu
Tartu
Saaremaa
Baltic Sea
Pskov
Псков
Viciebsk
Віцебск
Mogilev
Магілёў
Babruysk
Бабруйск
Gomel
Гомель
Chernihiv
Чернігів
Kiev
Rivne
Рівне


Karlskrona


Stockholm
Lillehammer
Mora
Falun
Gävle
Bergen
Borlänge
Oslo
Uppsala
Drammen
Västerås
Haugesund
Karlstad
Örebro
Fredrikstad
Stavanger
Norrköping
Linköping
Kristiansand
Gothenburg
Jönköping
Vendsyssel
Aalborg
North Sea
Aarhus
Copenhagen
Denmark
Sjælland
Malmö
Odense
Kiel
Rostock
Lübeck
Hamburg
Bremen
Amsterdam
Hanover
Wolfsburg
Berlin
The Hague
Netherlands
Brunswick
Magdeburg
Cambridge
London
Dortmund
Essen
Antwerp
Leipzig
Brighton
Brussels
Cologne
Germany
Dresden
Baltic Sea
Gotland
Turku
Lahti
Kouvola
Lappeenranta
Vyborg
Выборг
Tallinn
Narva
Gatchina
Гатчина
Pushkin
Пушкин
Veliky
Novgorod
Великий
Новгород
Estonia
Saaremaa
Pärnu
Tartu
Pskov
Псков
Riga
Jürmala
Latvia
Šiauliai
Daugavpils
Klaipėda
Lithuania
Viciebsk
Віцебск
Kaunas
Vilnius
Kaliningrad
Калининград
Gdynia
Gdańsk
Olsztyn
Minsk
Мінск
Mogilev
Магілёў
Grodno
Гродна
Vitaŭka
Вітаўка
Szczecin
Bydgoszcz
Poland
Białystok
Belarus
Babruysk
Бабруйск
Poznań
Warsaw
Gomel
Гомель
Brest
Брэст
Łódź
Chernihiv
Чернігів
Wrocław
Lublin
Kielce
Rivne
Рівне
Kiev


Almost done!


Back to topic...

Concept vs Tool Usage

Erlang?

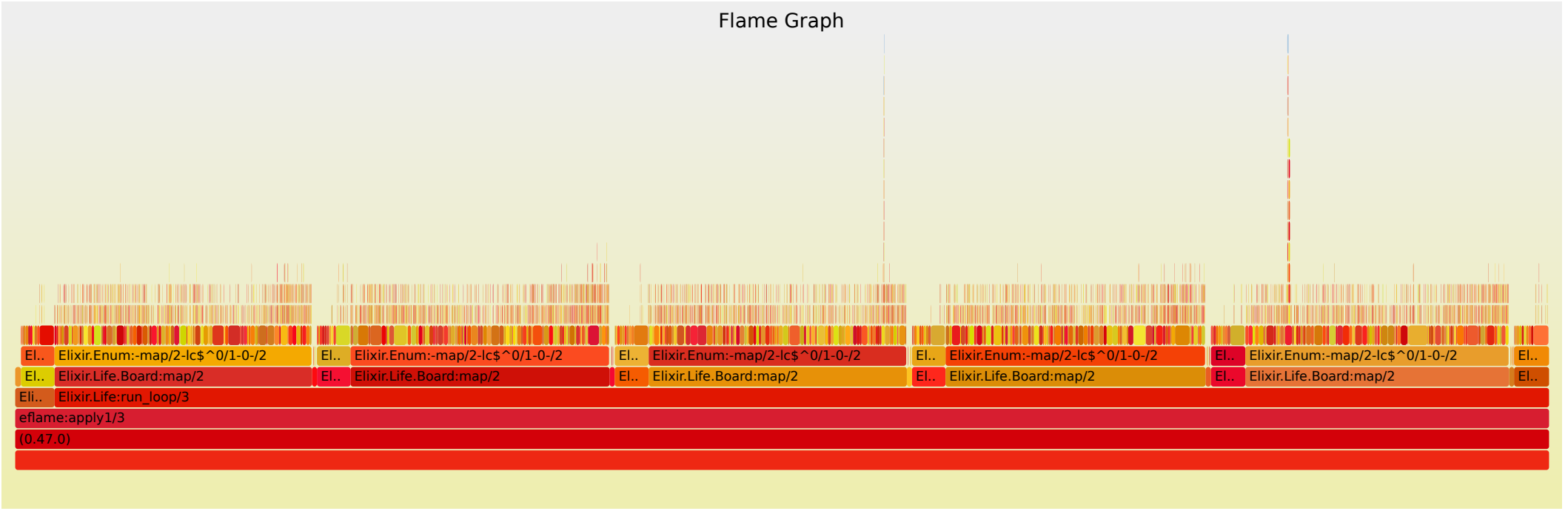
Use it from Erlang!

```
benchee:run(#{myFunc => fun() -> lists:sort(SomeList) end},
  [{warmup, 0}, {time, 2}]).
```

github.com/PragTob/benchee_erlang_try

github.com/barrel-db/rebar3_elixir_compile

Profiling vs. Benchmarking



http://learningelixir.joekain.com/profiling-elixir-2/

What to benchmark?

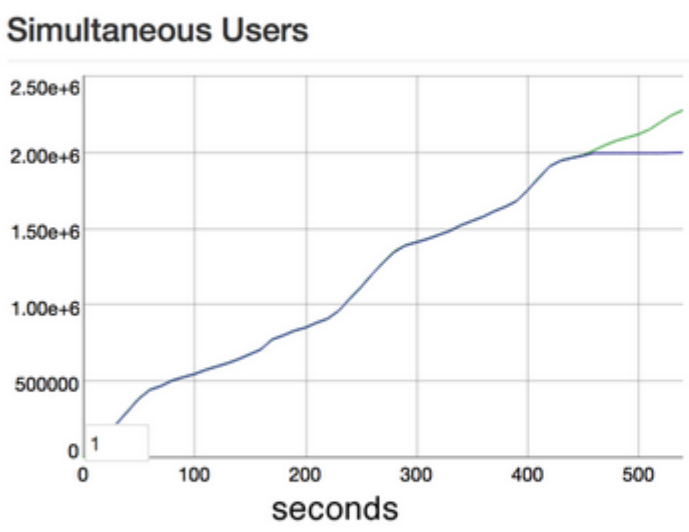
What to measure?

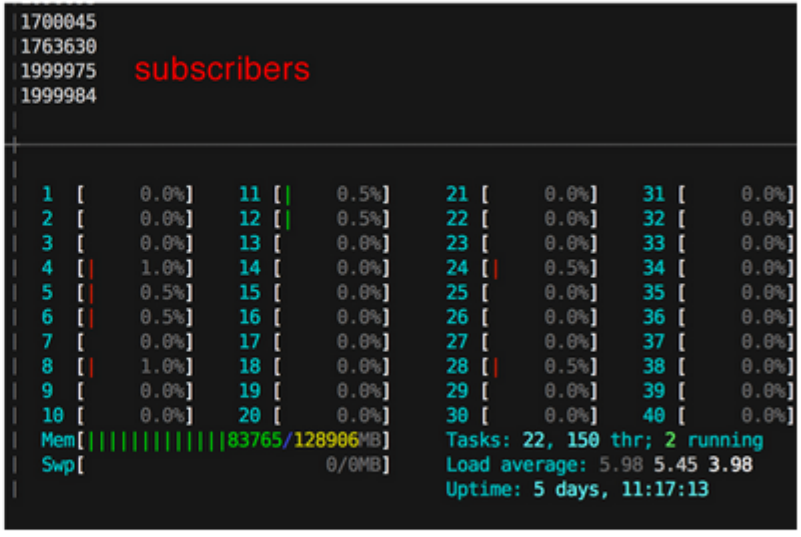
- Runtime?
- Memory?
- Throughput?
- Custom?

The famous post

The Road to 2 Million Websocket Connections in Phoenix

By Gary Rennie · about a year ago · v1.0.0





If you have been paying attention on Twitter recently, you have likely seen some increasing numbers regarding the number of simultaneous connections the Phoenix web framework can handle. This post documents some of the techniques used to perform the benchmarks.

What to measure?

- **Runtime!**
- Memory?
- Throughput?
- Custom?

How **long** will this take?

Enum.sort/1 performance

Name	Iterations per Second	Average	Deviation	median	minimum	maximum	Sample size
10k	584.69	1710.30 µs	±12.21%	1638.00 µs	1556.00 µs	5597.00 µs	2921
100k	42.93	23293.36 µs	±10.76%	21977.00 µs	20565.00 µs	30138.00 µs	215
1M	3.18	314895.69 µs	±9.95%	303440.50 µs	284252.00 µs	402633.00 µs	16
5M	0.49	2038061.67 µs	±9.72%	2000588.00 µs	1816395.00 µs	2297202.00 µs	3

Enum.sort/1 performance

Name	Iterations per Second	Average	Deviation	median	minimum	maximum	Sample size
10k	584.69	1710.30 µs	±12.21%	1638.00 µs	1556.00 µs	5597.00 µs	2921
100k	42.93	23293.36 µs	±10.76%	21977.00 µs	20565.00 µs	30138.00 µs	215
1M	3.18	314895.69 µs	±9.95%	303440.50 µs	284252.00 µs	402633.00 µs	16
5M	0.49	2038061.67 µs	±9.72%	2000588.00 µs	1816395.00 µs	2297202.00 µs	3

Did we make it **faster**?

What’s **fastest**?

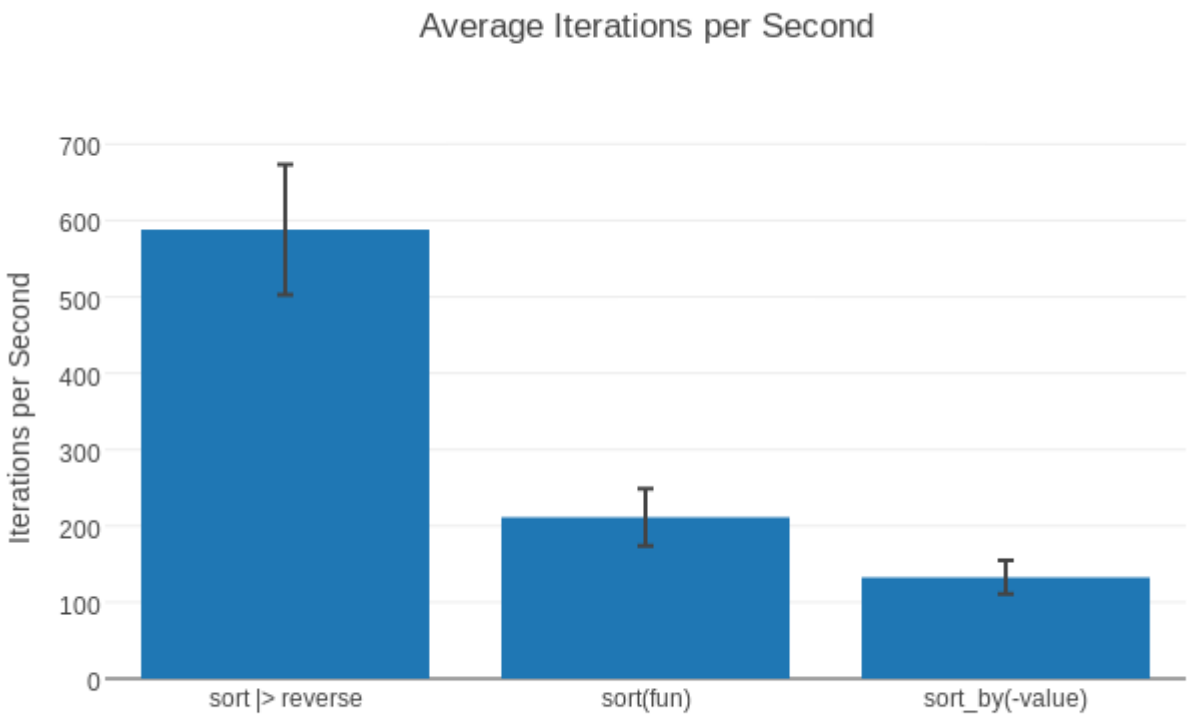
What's the **fastest** way to sort a list of numbers largest to smallest?

```elixir
list = 1..10_000 |> Enum.to_list |> Enum.shuffle

Benchee.run %{
  "sort(fun)" =>
    fn -> Enum.sort(list, &(&1 > &2)) end,
  "sort |> reverse" =>
    fn -> list |> Enum.sort |> Enum.reverse end,
  "sort_by(-value)" =>
    fn -> Enum.sort_by(list, fn(val) -> -val end) end
}
```

```elixir
list = 1..10_000 |> Enum.to_list |> Enum.shuffle

Benchee.run %{
  "sort(fun)" =>
    fn -> Enum.sort(list, &(&1 > &2)) end,
  "sort |> reverse" =>
    fn -> list |> Enum.sort |> Enum.reverse end,
  "sort_by(-value)" =>
    fn -> Enum.sort_by(list, fn(val) -> -val end) end
}
```

```elixir
list = 1..10_000 |> Enum.to_list |> Enum.shuffle

Benchee.run %{
  "sort(fun)" =>
    fn -> Enum.sort(list, &(&1 > &2)) end,
  "sort |> reverse" =>
    fn -> list |> Enum.sort |> Enum.reverse end,
  "sort_by(-value)" =>
    fn -> Enum.sort_by(list, fn(val) -> -val end) end
}
```

```elixir
list = 1..10_000 |> Enum.to_list |> Enum.shuffle

Benchee.run %{
  "sort(fun)" =>
    fn -> Enum.sort(list, &(&1 > &2)) end,
  "sort |> reverse" =>
    fn -> list |> Enum.sort |> Enum.reverse end,
  "sort_by(-value)" =>
    fn -> Enum.sort_by(list, fn(val) -> -val end) end
}
```




```
Name                           ips        average  deviation         median
sort |> reverse            587.86        1.70 ms    ±14.54%        1.58 ms
sort(fun)                  211.48        4.73 ms    ±17.70%        4.53 ms
sort_by(-value)            132.76        7.53 ms    ±16.60%        7.15 ms

Comparison:
sort |> reverse            587.86
sort(fun)                  211.48 - 2.78x slower
sort_by(-value)            132.76 - 4.43x slower
```

“Isn’t that the root of all evil?”

“More likely, not reading the sources is the source of all evil.”

Me, just now

Yup it's there

"We should forget about small efficiencies, say about 97% of the time: premature optimization is the root of all evil."

Donald Knuth, 1974

(Computing Surveys, Vol 6, No 4, December 1974)

The very next sentence

"Yet we should not pass up our opportunities in that critical 3%.
A good programmer (...) will be wise to look carefully at the critical code but only after that code has been identified."

Donald Knuth, 1974

(Computing Surveys, Vol 6, No 4, December 1974)

Prior Paragraph

"In established engineering disciplines a 12 % improvement, easily obtained, is never considered marginal; and I believe the same viewpoint should prevail in software engineering."

Donald Knuth, 1974

(Computing Surveys, Vol 6, No 4, December 1974)

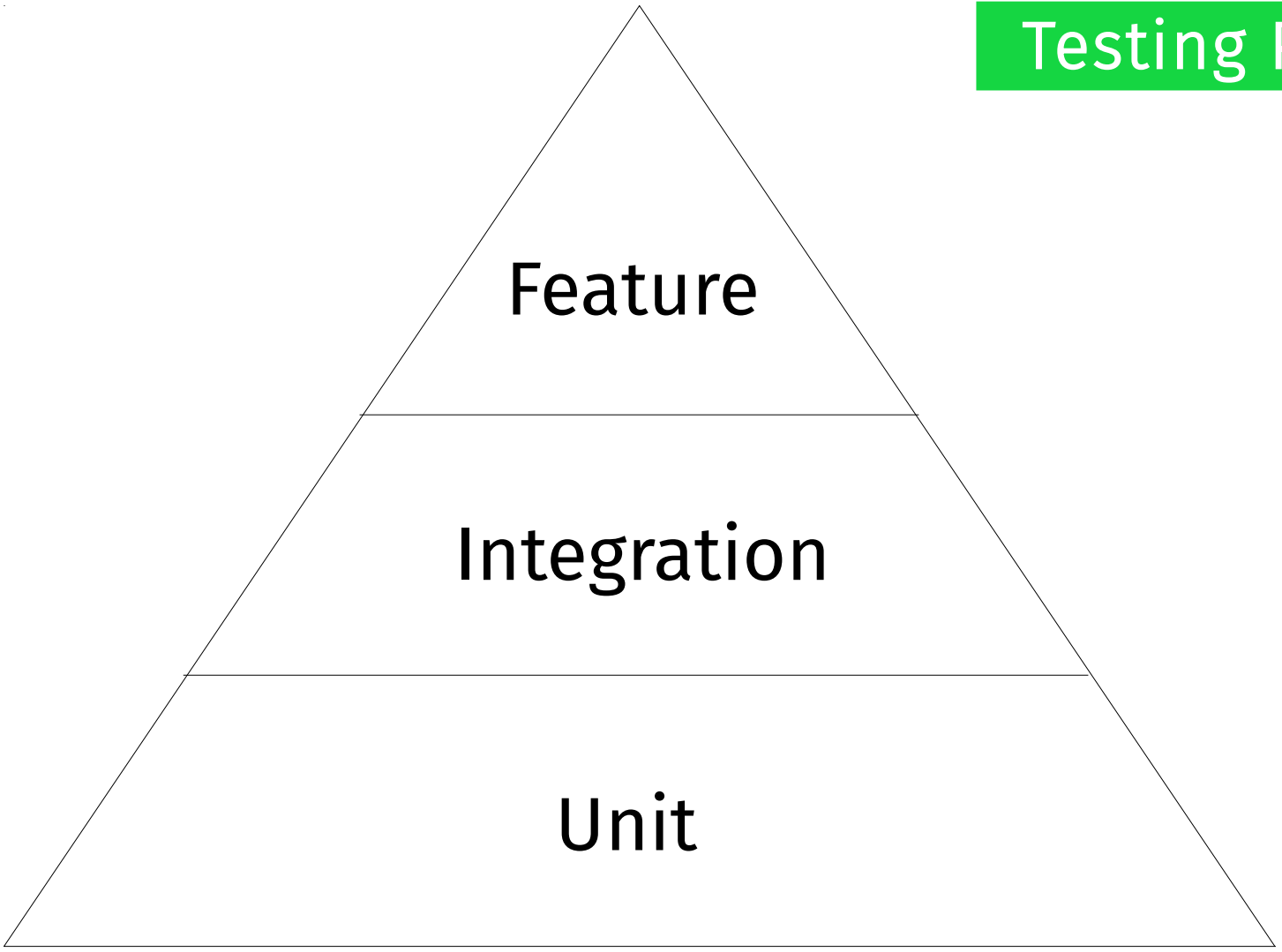
*"It is often a **mistake to make a priori judgments** about what parts of a program are really critical, since the universal experience of programmers who have been using measurement tools has been that **their intuitive guesses fail.**"*

Donald Knuth, 1974

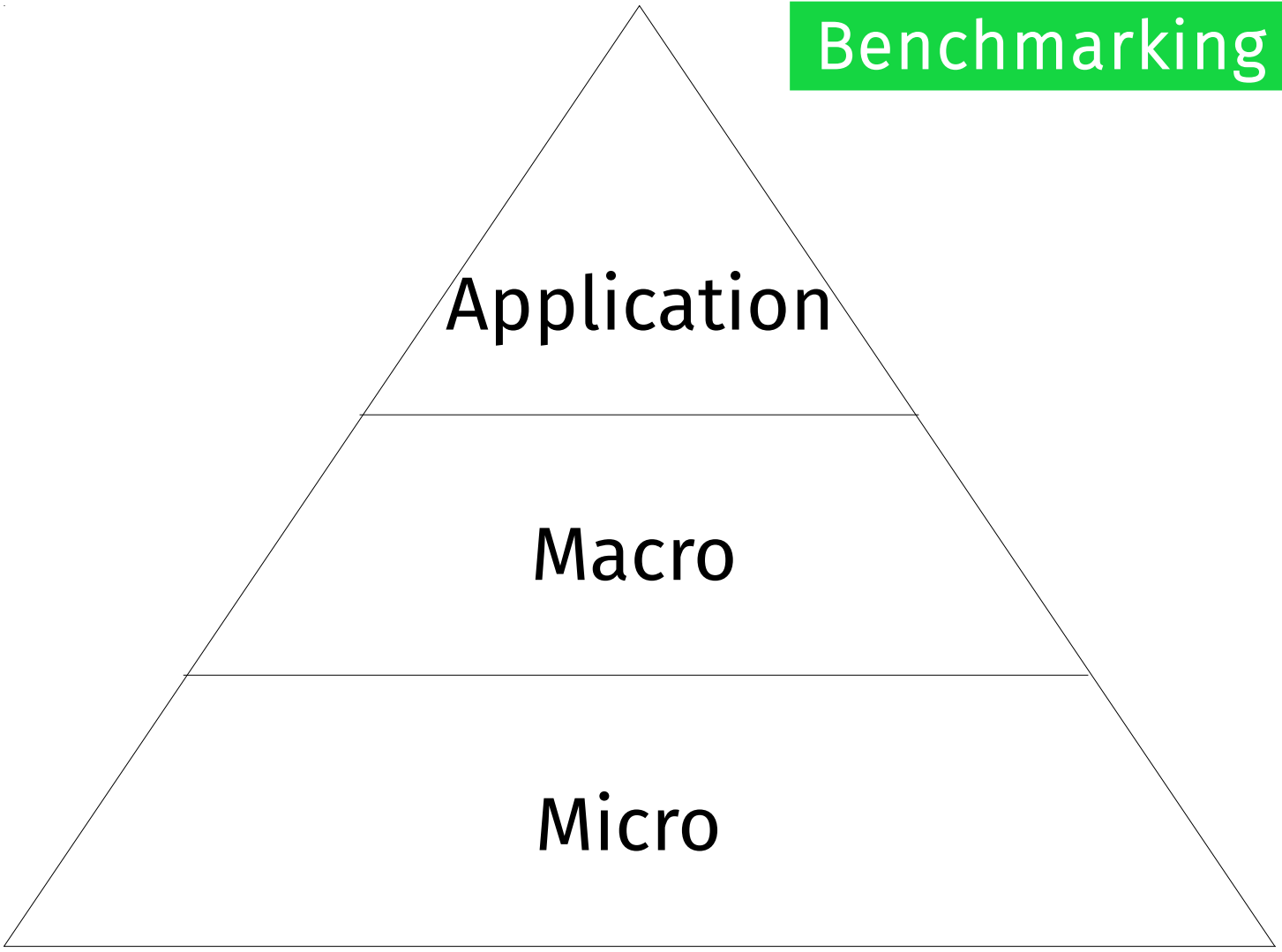
(Computing Surveys, Vol 6, No 4, December 1974)

Different **types** of benchmarks

Testing Pyramid



Benchmarking Pyramid



Results

20-updates (bar) | Data table | Latency | Framework overhead

Responses per second at 20 updates per request, Dell servers at ServerCentral (179 tests)

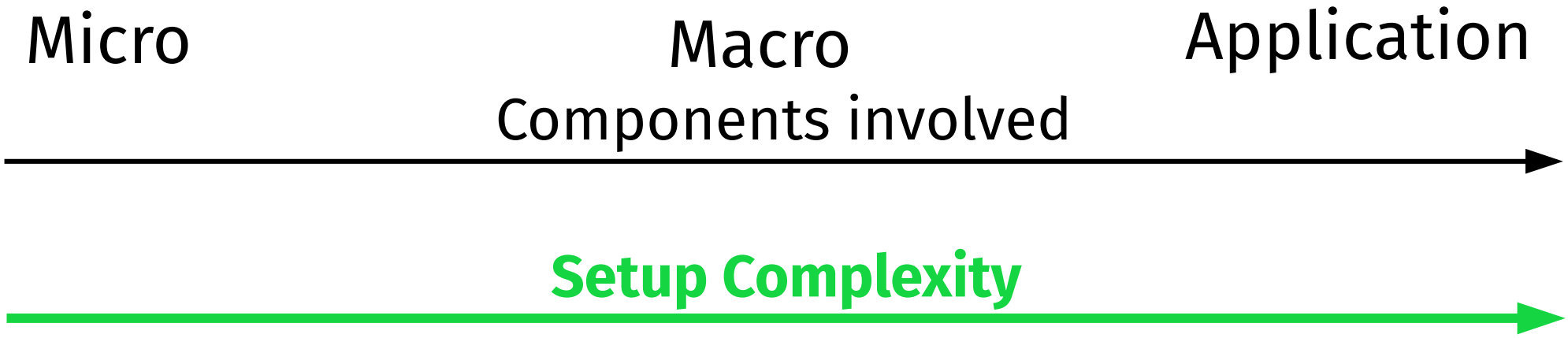
Framework	Performance (higher is better)	Cls	Lng	Plt	FE	Aos	DB	Dos	Orm	IA	Errors
fasthttp-postgresql-	3,050 100.0%	Plt	Go	Non	Non	Lin	Pg	Lin	Raw	Rea	0
wt-postgres	2,945 96.6%	Ful	C++	Non	Non	Lin	Pg	Lin	Ful	Rea	0
revenj.jvm	2,873 94.2%	Ful	Jav	Svt	Res	Lin	Pg	Lin	Ful	Rea	0
express-mongodb	2,841 93.1%	Mcr	JS	Non	Non	Lin	Mo	Lin	Ful	Rea	0
cutelyst-pf-pg-raw	2,735 89.7%	Plt	C++	Qt	Non	Lin	Pg	Lin	Raw	Rea	0
cutelyst-uwsgi-nginx	2,717 89.1%	Plt	C++	Qt	ngx	Lin	Pg	Lin	Raw	Rea	0
cutelyst-thread-pg-r	2,699 88.5%	Plt	C++	Qt	Non	Lin	Pg	Lin	Raw	Rea	0
mojolicious	2,479 81.3%	Ful	Prl	Non	Hyp	Lin	Pg	Lin	Raw	Rea	72
fasthttp	2,455 80.5%	Plt	Go	Non	Non	Lin	My	Lin	Raw	Rea	0
wt	2,341 76.8%	Ful	C++	Non	Non	Lin	My	Lin	Ful	Rea	0
ulib-mysql	2,317 76.0%	Plt	C++	Non	ULi	Lin	My	Lin	Mcr	Rea	0
fasthttp-mysql-prefo	2,280 74.8%	Plt	Go	Non	Non	Lin	My	Lin	Raw	Rea	0
cutelyst-pf-mysql-ra	1,996 65.4%	Plt	C++	Qt	Non	Lin	My	Lin	Raw	Rea	0
nodejs	1,982 65.0%	Plt	JS	njs	Non	Lin	My	Lin	Raw	Rea	0
aspnetcore-mvc-raw	1,976 64.8%	Ful	C#	Net	Non	Lin	Pg	Lin	Raw	Rea	0
cutelyst-thread-mysq	1,975 64.8%	Plt	C++	Qt	Non	Lin	My	Lin	Raw	Rea	0
aspnetcore-middlewar	1,955 64.1%	Mcr	C#	Net	Non	Lin	Pg	Lin	Raw	Rea	0
cutelyst-uwsgi-nginx	1,933 63.4%	Plt	C++	Qt	ngx	Lin	My	Lin	Raw	Rea	0
phoenix	1,915 62.8%	Mcr	Eli	Cow	Non	Lin	Pg	Lin	Ful	Rea	0
redstone-postgresql	1,857 60.9%	Mcr	Dar	Non	Non	Lin	Pg	Lin	Mcr	Rea	0

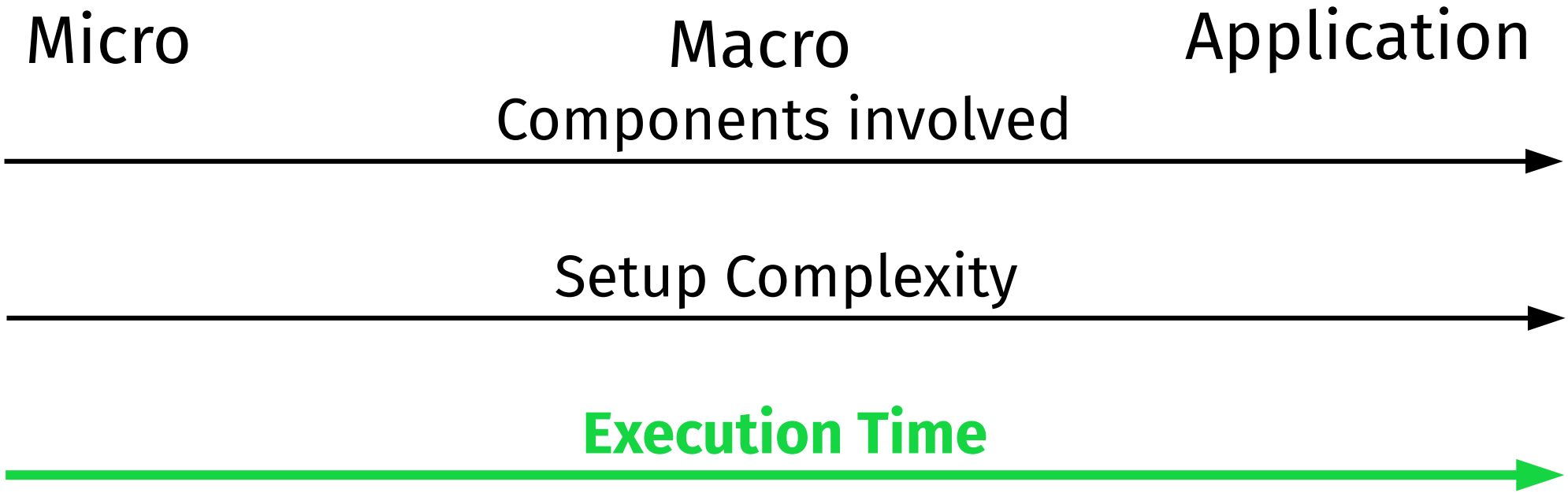
Micro

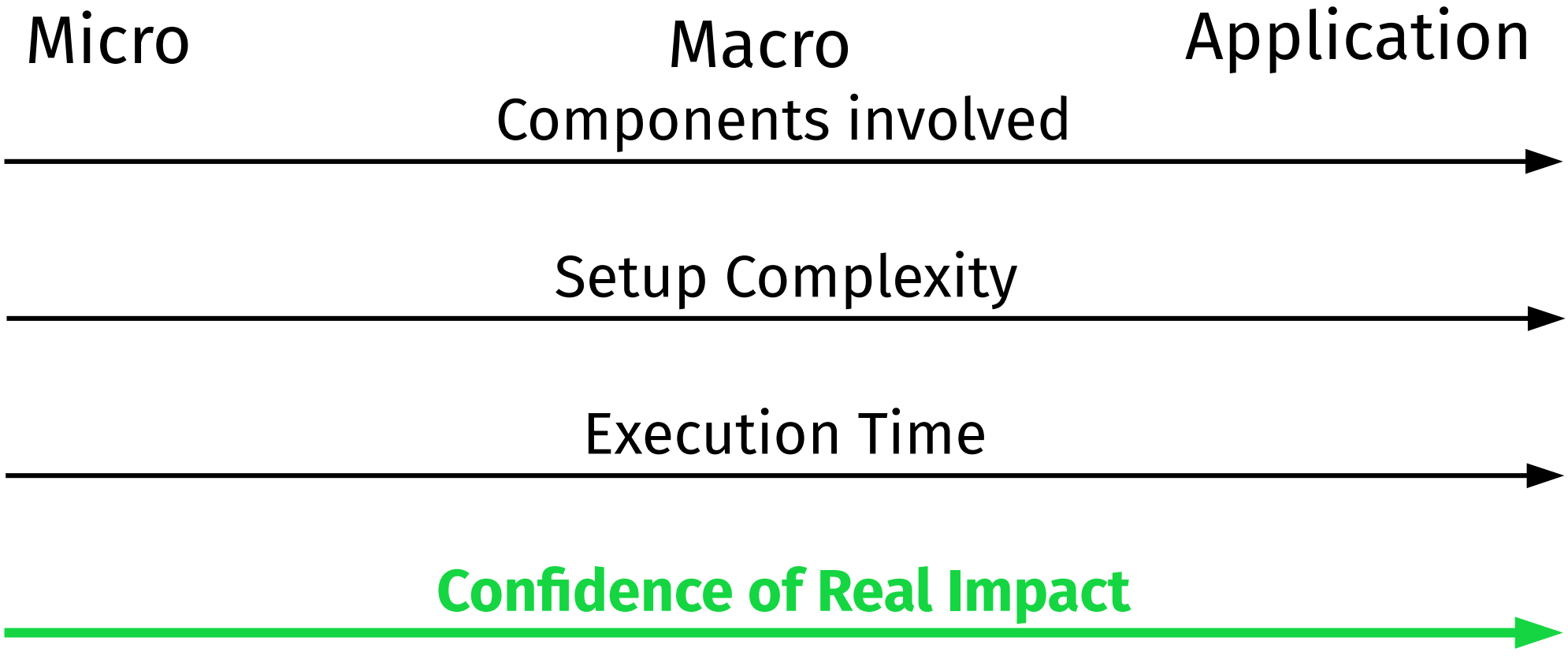
Macro

Application

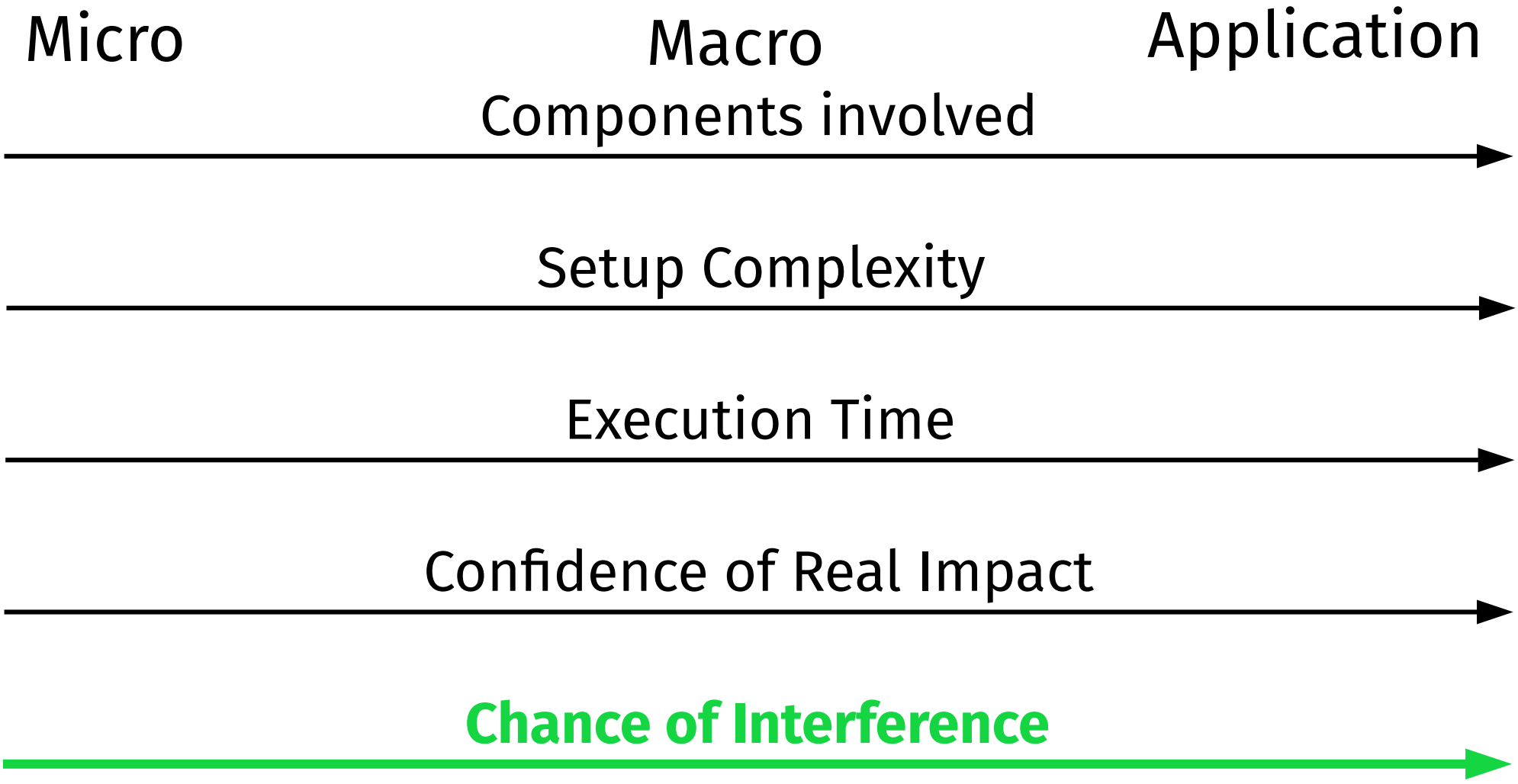

Micro
Macro
Application
Components involved

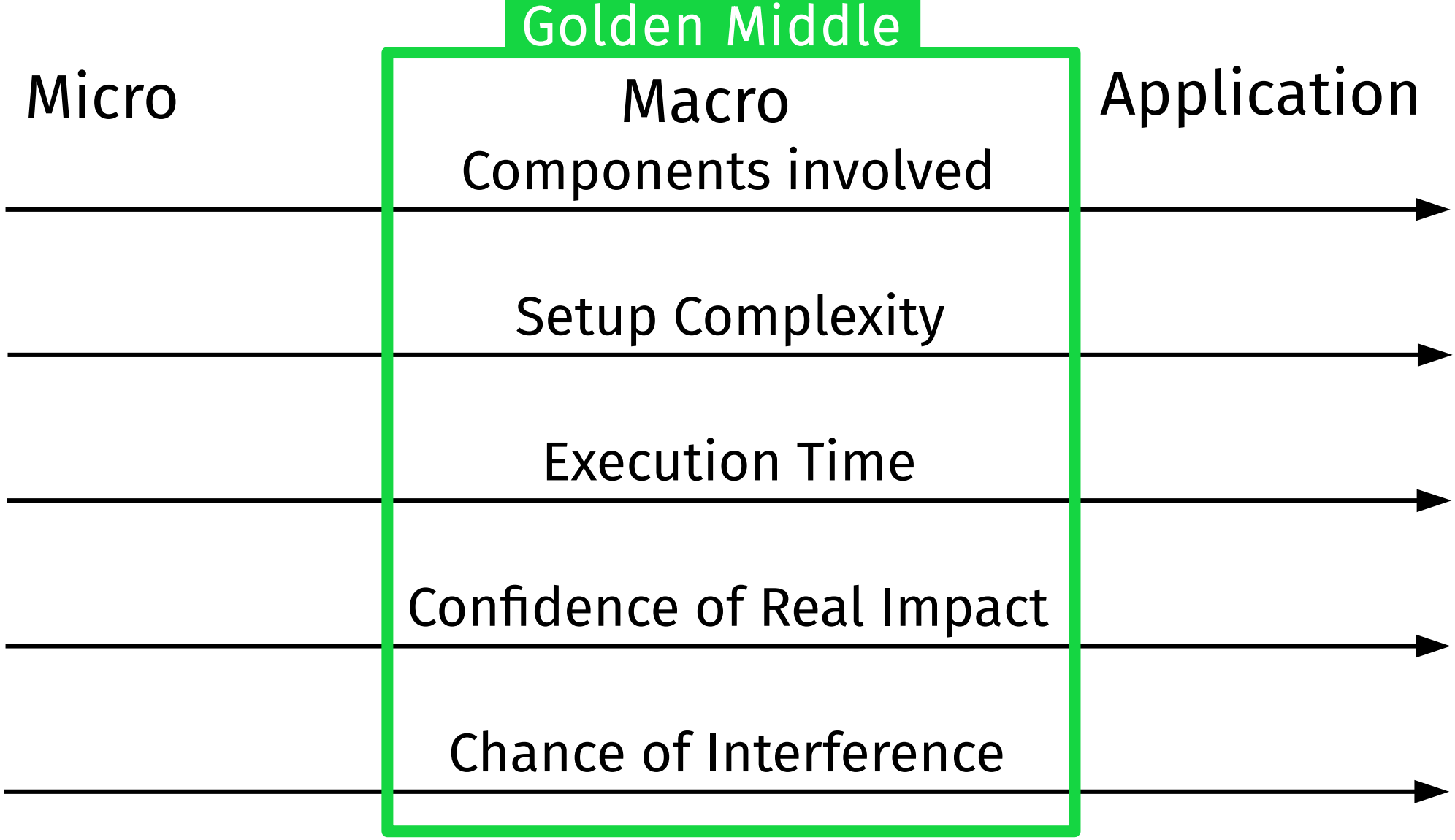

Micro
Macro
Application
Components involved
Setup Complexity

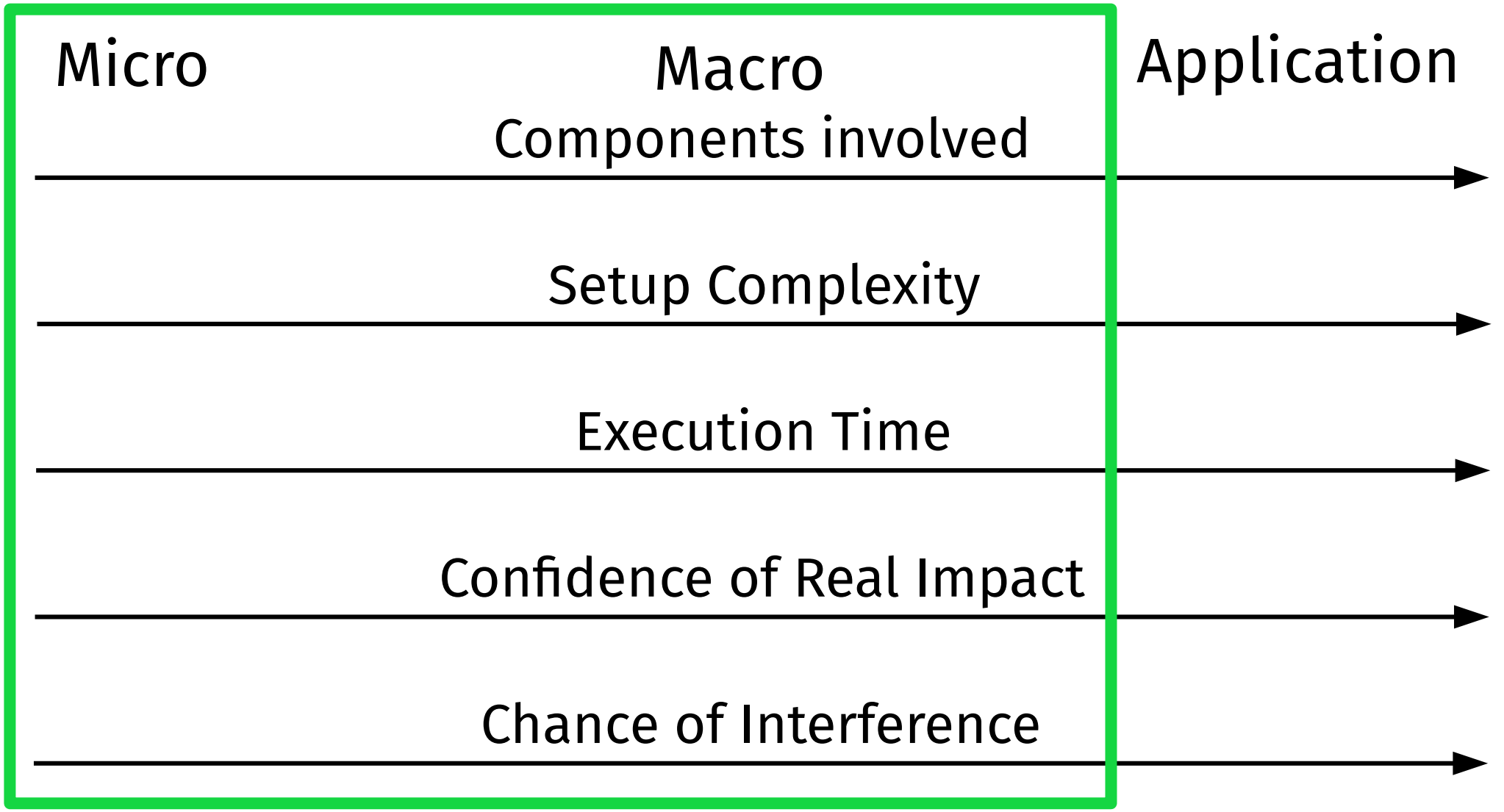

Micro
Macro
Application
Components involved
Setup Complexity
Execution Time


Micro
Macro
Application
Components involved
Setup Complexity
Execution Time
Confidence of Real Impact

Overly **specific** benchmarks &
exaggerated results


Micro
Macro
Application
Components involved
Setup Complexity
Execution Time
Confidence of Real Impact
Chance of Interference


Golden Middle
Micro
Macro
Application
Components involved
Setup Complexity
Execution Time
Confidence of Real Impact
Chance of Interference


Micro
Macro
Application
Components involved
Setup Complexity
Execution Time
Confidence of Real Impact
Chance of Interference

Good Benchmarking

What are you benchmarking for?

System Specification

- Elixir 1.4.4
- Erlang 19.3
- i5-7200U – 2 x 2.5GHz (Up to 3.10GHz)
- 8GB RAM
- Linux Mint 18.1 - 64 bit (Ubuntu 16.04 base)
- Linux Kernel 4.8.0-46

System Specification

```
tobi@comfy elixir_playground $ mix run bench/is_even.exs
Operating System: Linux
CPU Information: Intel(R) Core(TM) i5-7200U CPU @ 2.50GHz
Number of Available Cores: 4
Available memory: 8.05344 GB
Elixir 1.4.4
Erlang 19.3
Benchmark suite executing with the following configuration:
warmup: 10.00 s
time: 10.00 s
parallel: 1
inputs: none specified
Estimated total run time: 1.00 min
```


Interference free Environment

Logging & Friends

```
[info] GET /
[debug] Processing by Rumbl.PageController.index/2
  Parameters: %{}
  Pipelines: [:browser]
[info] Sent 200 in 46ms
[info] GET /sessions/new
[debug] Processing by Rumbl.SessionController.new/2
  Parameters: %{}
  Pipelines: [:browser]
[info] Sent 200 in 5ms
[info] GET /users/new
[debug] Processing by Rumbl.UserController.new/2
  Parameters: %{}
  Pipelines: [:browser]
[info] Sent 200 in 7ms
[info] POST /users
[debug] Processing by Rumbl.UserController.create/2
  Parameters: %{"_csrf_token" => "NUEUdRMNAiBfIHEeNwZkfA05PgAOJgAAf0ACXJqCjl7YojW+trdjdg==", "_utf8" => "✓", "user"
=> %{"name" => "asdasd", "password" => "[FILTERED]", "username" => "Homer"}}
  Pipelines: [:browser]
[debug] QUERY OK db=0.1ms
begin []
[debug] QUERY OK db=0.9ms
INSERT INTO "users" ("name","password_hash","username","inserted_at","updated_at") VALUES ($1,$2,$3,$4,$5) RETURNING
"id" ["asdasd", "$2b$12$.qY/kpo0Dec7vMK1ClJoC.Lw77c3oGllX7uieZILMlFh2hFpJ3F.C", "Homer", {{2016, 12, 2}, {14, 10, 28, 0}},
{{2016, 12, 2}, {14, 10, 28, 0}}]
```


Garbage Collection


Enum.each Raw Run Times
Run Time in microseconds
600
400
200
0
0
5k
10k
15k
20k
Sample number


Correct & Meaningful Setup

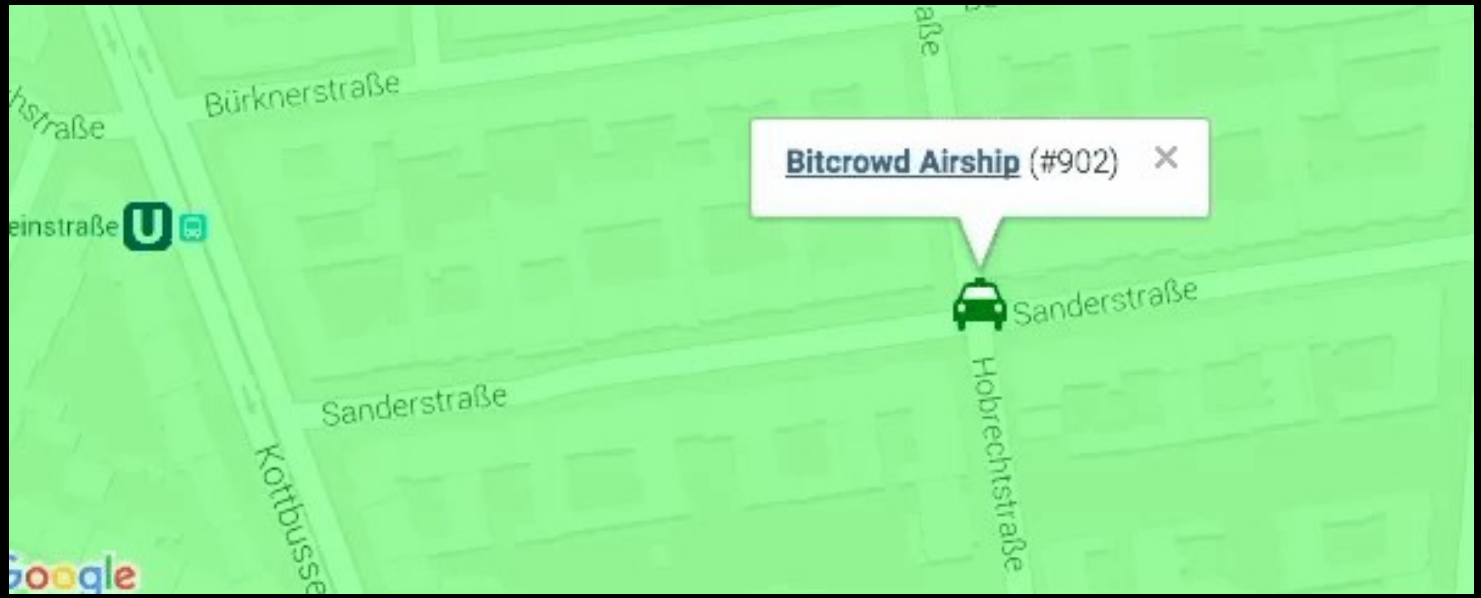

Warmup


Inputs matter!


Story Time

Boom


Boom
Elixir.DBConnection.ConnectionError


Bürknerstraße
Bitcrowd Airship (#902)
Sanderstraße
Sanderstraße
Hobrechtstraße
Kottbusser
Google

```elixir
Benchee.run %{
  "Using LatestCourierLocation" => fn(courier_id) ->
    LatestCourierLocation
    |> CourierLocation.with_courier_ids(courier_id)
    |> Repo.one
  end,
  "with_courier_ids + order" => fn(courier_id) ->
    CourierLocation.with_courier_ids(courier_id)
    |> Ecto.Query.order_by(desc: :time)
    |> Ecto.Query.limit(1)
    |> Repo.one
  end,
  "full custom" => fn(courier_id) ->
    CourierLocation
    |> Ecto.Query.where(courier_id: ^courier_id)
    |> Ecto.Query.order_by(desc: :time)
    |> Ecto.Query.limit(1)
    |> Repo.one
  end
}
```

```elixir
Benchee.run %{
  "Using LatestCourierLocation" => fn(courier_id) ->
    LatestCourierLocation
    |> CourierLocation.with_courier_ids(courier_id)
    |> Repo.one
  end,
  "with_courier_ids + order" => fn(courier_id) ->
    CourierLocation.with_courier_ids(courier_id)
    |> Ecto.Query.order_by(desc: :time)
    |> Ecto.Query.limit(1)
    |> Repo.one
  end,
  "full custom" => fn(courier_id) ->
    CourierLocation
    |> Ecto.Query.where(courier_id: ^courier_id)
    |> Ecto.Query.order_by(desc: :time)
    |> Ecto.Query.limit(1)
    |> Repo.one
  end
}
```

A real case

```elixir
Benchee.run %{
  "Using LatestCourierLocation" => fn(courier_id) ->
    LatestCourierLocation
    |> CourierLocation.with_courier_ids(courier_id)
    |> Repo.one
  end,
  "with_courier_ids + order" => fn(courier_id) ->
    CourierLocation.with_courier_ids(courier_id)
    |> Ecto.Query.order_by(desc: :time)
    |> Ecto.Query.limit(1)
    |> Repo.one
  end,
  "full custom" => fn(courier_id) ->
    CourierLocation
    |> Ecto.Query.where(courier_id: ^courier_id)
    |> Ecto.Query.order_by(desc: :time)
    |> Ecto.Query.limit(1)
    |> Repo.one
  end
}
```

Database View

```
Benchee.run %{
  "Using LatestCourierLocation" => fn(courier_id) ->
    LatestCourierLocation
    |> CourierLocation.with_courier_ids(courier_id)
    |> Repo.one
  end,
  "with courier ids + order" => fn(courier_id) ->
    CourierLocation.with_courier_ids(courier_id)
    |> Ecto.Query.order_by(desc: :time)
    |> Ecto.Query.limit(1)
    |> Repo.one
  end,
  "full custom" => fn(courier_id) ->
    CourierLocation
    |> Ecto.Query.where(courier_id: ^courier_id)
    |> Ecto.Query.order_by(desc: :time)
    |> Ecto.Query.limit(1)
    |> Repo.one
  end
}
```

Only Difference

Another job well done?

```
Name                                  ips        average  deviation         median
with_courier_ids + order             1.19 K      841.44 µs    ±67.64%      675.00 µs
full custom                          1.16 K      862.36 µs    ±56.06%      737.00 µs
Using LatestCourierLocation     0.00603 K   165897.47 µs     ±2.33%   165570.00 µs

Comparison:
with_courier_ids + order             1.19 K
full custom                          1.16 K - 1.02x slower
Using LatestCourierLocation     0.00603 K - 197.16x slower
```

Boom


Boom
Elixir.DBConnection.ConnectionError


Boom
Elixir.DBConnection.ConnectionError
Elixir.DBConnection.ConnectionError


Boom
Elixir.DBConnection.ConnectionError
Elixir.DBConnection.ConnectionError
Elixir.DBConnection.ConnectionError
Elixir.DBConnection.ConnectionError
Elixir.DBConnection.ConnectionError
Elixir.DBConnection.ConnectionError
Elixir.DBConnection.ConnectionError
Elixir.DBConnection.ConnectionError

Inputs to the rescue!

```
inputs = %{
  "Big 2.3 Million locations" => 3799,
  "No locations"              => 8901,
  "~200k locations"           => 4238,
  "~20k locations"            => 4201
}

Benchee.run %{
  ...
  "full custom" => fn(courier_id) ->
    CourierLocation
    |> Ecto.Query.where(courier_id: ^courier_id)
    |> Ecto.Query.order_by(desc: :time)
    |> Ecto.Query.limit(1)
    |> Repo.one
  end
}, inputs: inputs, time: 25, warmup: 5
```

Inputs to the rescue!

```
inputs = %{
  "Big 2.3 Million locations" => 3799,
  "No locations"              => 8901,
  "~200k locations"           => 4238,
  "~20k locations"            => 4201
}

Benchee.run %{
  ...
  "full custom" => fn(courier_id) ->
    CourierLocation
    |> Ecto.Query.where(courier_id: ^courier_id)
    |> Ecto.Query.order_by(desc: :time)
    |> Ecto.Query.limit(1)
    |> Repo.one
  end
}, inputs: inputs, time: 25, warmup: 5
```

Inputs to the rescue!

```
inputs = %{
  "Big 2.3 Million locations" => 3799,
  "No locations"              => 8901,
  "~200k locations"           => 4238,
  "~20k locations"            => 4201
}

Benchee.run %{
  ...
  "full custom" => fn(courier_id) ->
    CourierLocation
    |> Ecto.Query.where(courier_id: ^courier_id)
    |> Ecto.Query.order_by(desc: :time)
    |> Ecto.Query.limit(1)
    |> Repo.one
  end
}, inputs: inputs, time: 25, warmup: 5
```

Inputs to the rescue!

```
inputs = %{
  "Big 2.3 Million locations" => 3799,
  "No locations"              => 8901,
  "~200k locations"           => 4238,
  "~20k locations"            => 4201
}

Benchee.run %{
  ...
  "full custom" => fn(courier_id) ->
    CourierLocation
    |> Ecto.Query.where(courier_id: ^courier_id)
    |> Ecto.Query.order_by(desc: :time)
    |> Ecto.Query.limit(1)
    |> Repo.one
  end
}, inputs: inputs, time: 25, warmup: 5
```

Old

```
##### With input Big 2.3 Million locations #####
Comparison:
with_courier_ids + order           1.19 K
full custom                        1.16 K - 1.02x slower
Using LatestCourierLocation     0.00603 K - 197.16x slower

##### With input ~200k locations #####
Comparison:
Using LatestCourierLocation         3.66
full custom                        0.133 - 27.57x slower
with_courier_ids + order           0.132 - 27.63x slower

##### With input ~20k locations #####
Comparison:
Using LatestCourierLocation        38.12
full custom                        0.122 - 312.44x slower
with_courier_ids + order           0.122 - 313.33x slower

##### With input No locations #####
Comparison:
Using LatestCourierLocation      2967.48
full custom                        0.114 - 25970.57x slower
with_courier_ids + order           0.114 - 26046.06x slower
```

Old

```
##### With input Big 2.3 Million locations #####
Comparison:
with_courier_ids + order           1.19 K
full custom                        1.16 K - 1.02x slower
Using LatestCourierLocation  0.00603 K - 197.16x slower

##### With input ~200k locations #####
Comparison:
Using LatestCourierLocation       3.66
full custom                       0.133 - 27.57x slower
with_courier_ids + order          0.132 - 27.63x slower

##### With input ~20k locations #####
Comparison:
Using LatestCourierLocation      38.12
full custom                       0.122 - 312.44x slower
with_courier_ids + order          0.122 - 313.33x slower

##### With input No locations #####
Comparison:
Using LatestCourierLocation    2967.48
full custom                       0.114 - 25970.57x slower
with_courier_ids + order          0.114 - 26046.06x slower
```

Old

```
##### With input Big 2.3 Million locations #####
Comparison:
with_courier_ids + order          1.19 K
full custom                       1.16 K - 1.02x slower
Using LatestCourierLocation     0.00603 K - 197.16x slower

##### With input ~200k locations #####
Comparison:
Using LatestCourierLocation        3.66
full custom                      0.133 - 27.57x slower
with_courier_ids + order         0.132 - 27.63x slower

##### With input ~20k locations #####
Comparison:
Using LatestCourierLocation       38.12
full custom                      0.122 - 312.44x slower
with_courier_ids + order         0.122 - 313.33x slower

##### With input No locations #####
Comparison:
Using LatestCourierLocation     2967.48
full custom                      0.114 - 25970.57x slower
with_courier_ids + order         0.114 - 26046.06x slower
```

Combined Indexes

Combined Index

```
##### With input Big 2.3 Million locations #####
Comparison:
full custom                          3921.12
with_courier_ids + order               23.05 - 170.09x slower
Using LatestCourierLocation             5.98 - 655.74x slower

##### With input ~200k locations #####
Comparison:
full custom                          4272.84
with_courier_ids + order               14.20 - 300.91x slower
Using LatestCourierLocation             3.80 - 1125.59x slower

##### With input ~20k locations #####
Comparison:
full custom                          3792.97
with_courier_ids + order               78.93 - 48.06x slower
Using LatestCourierLocation            35.62 - 106.47x slower

##### With input No locations #####
Comparison:
full custom                           5.14 K
with_courier_ids + order              3.87 K - 1.33x slower
Using LatestCourierLocation           3.29 K - 1.56x slower
```

Combined Index

```
##### With input Big 2.3 Million locations #####
Comparison:
full custom                          3921.12
with_courier_ids + order               23.05 - 170.09x slower
Using LatestCourierLocation             5.98 - 655.74x slower

##### With input ~200k locations #####
Comparison:
full custom                          4272.84
with_courier_ids + order               14.20 - 300.91x slower
Using LatestCourierLocation             3.80 - 1125.59x slower

##### With input ~20k locations #####
Comparison:
full custom                          3792.97
with_courier_ids + order               78.93 - 48.06x slower
Using LatestCourierLocation            35.62 - 106.47x slower

##### With input No locations #####
Comparison:
full custom                           5.14 K
with_courier_ids + order               3.87 K - 1.33x slower
Using LatestCourierLocation            3.29 K - 1.56x slower
```

Insertion Time

```
# with an index on courier_id and one on time

Name                              ips        average  deviation         median
Inserting a location           366.90        2.73 ms    ±36.35%         2.29
ms

# with a combined index on courier_id and time

Name                              ips        average  deviation         median
Updating a location            283.41        3.53 ms    ±52.18%        2.77 ms
```

Excursion into Statistics

Average

```
average = total_time / iterations
```

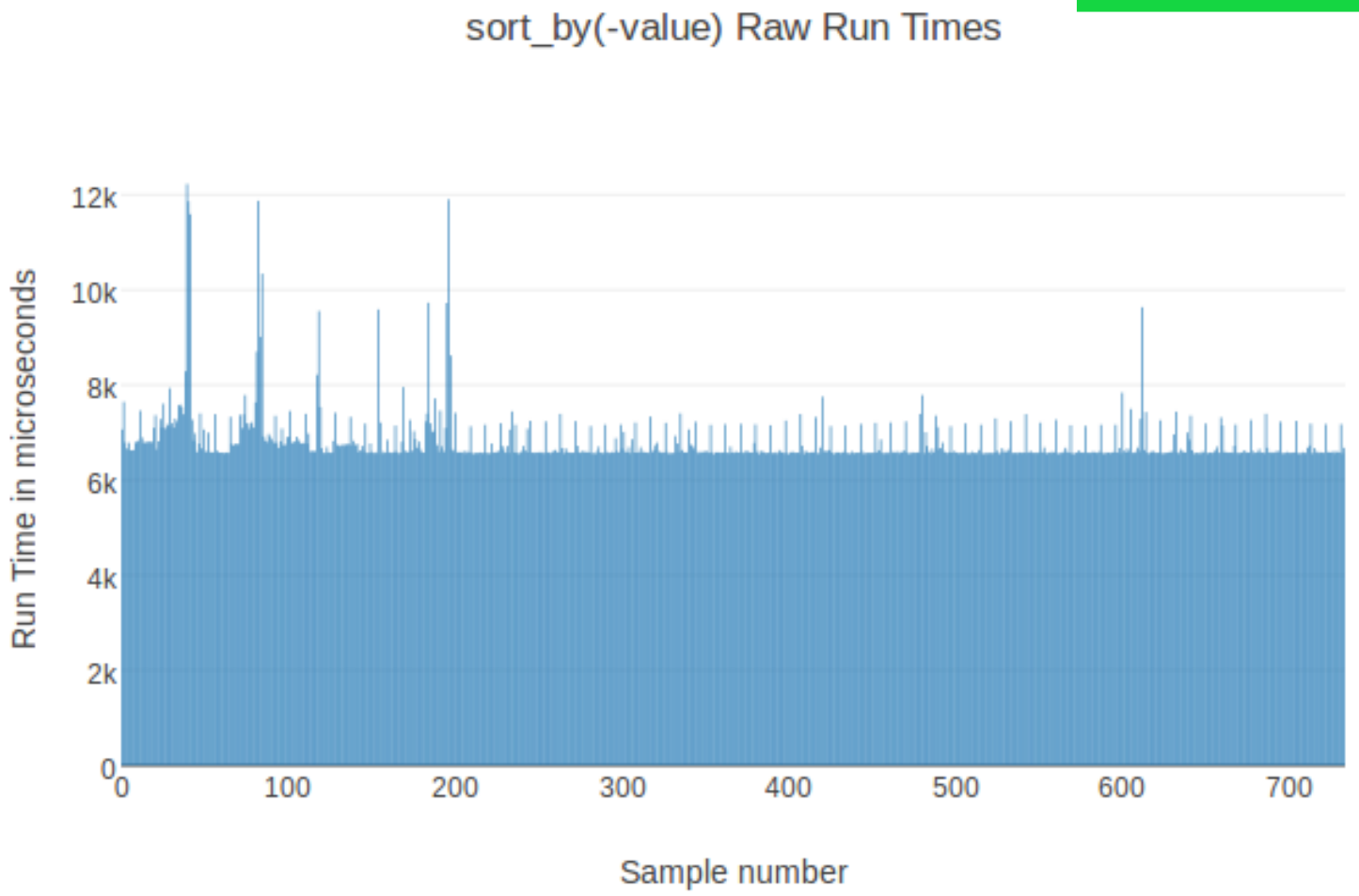
Standard Deviation

```
defp standard_deviation(samples, average, iterations) do
  total_variance = Enum.reduce samples, 0,  fn(sample, total) ->
    total + :math.pow((sample - average), 2)
  end
  variance = total_variance / iterations
  :math.sqrt variance
end
```

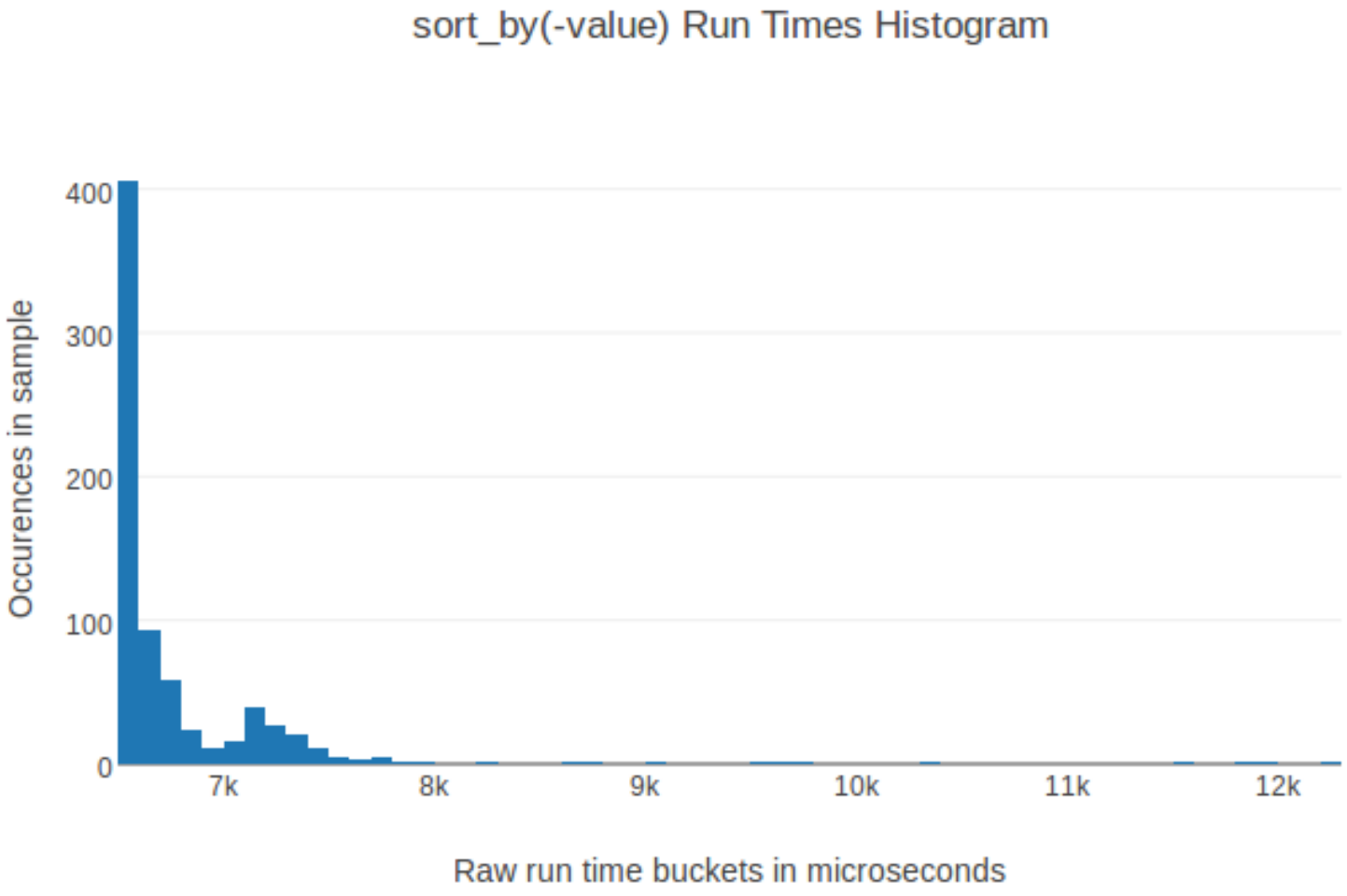
Spread of Values

```
defp standard_deviation(samples, average, iterations) do
  total_variance = Enum.reduce samples, 0,  fn(sample, total) ->
    total + :math.pow((sample - average), 2)
  end
  variance = total_variance / iterations
  :math.sqrt variance
end
```

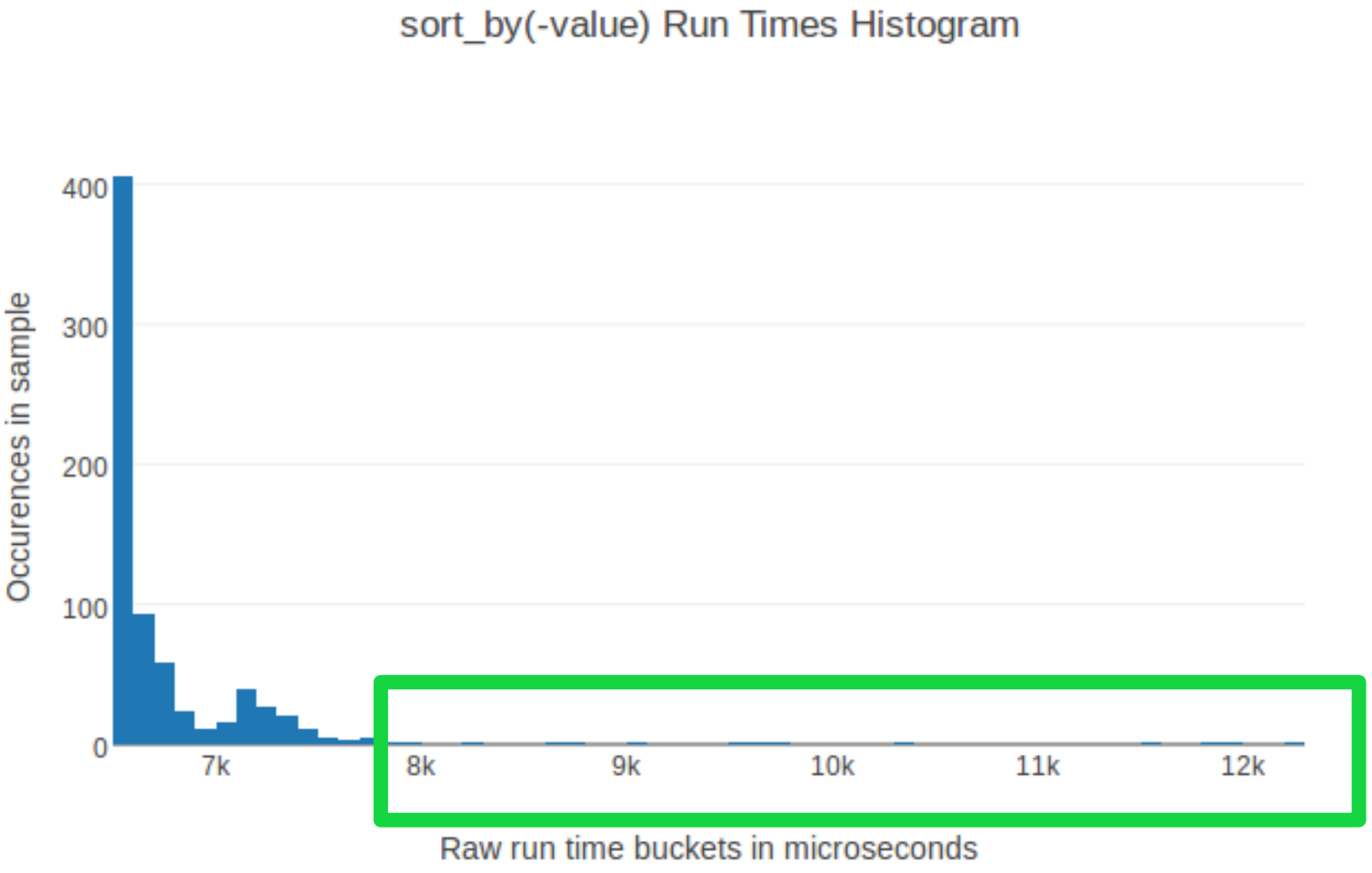
Raw Run Times



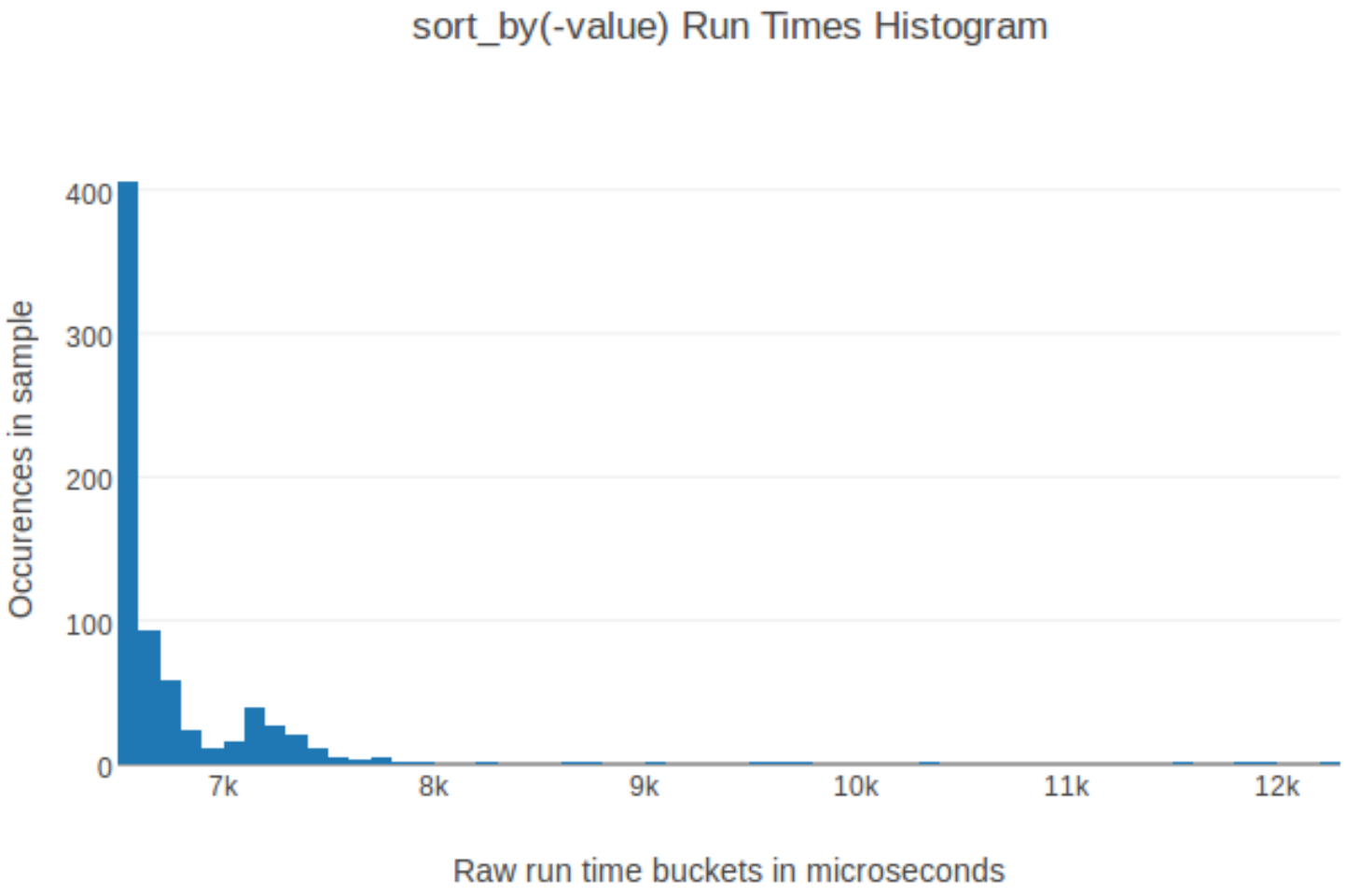
Histogram


sort_by(-value) Run Times Histogram
Occurences in sample
400
300
200
100
0
7k
8k
9k
10k
11k
12k
Raw run time buckets in microseconds

Outliers


sort_by(-value) Run Times Histogram
Occurences in sample
400
300
200
100
0
7k
8k
9k
10k
11k
12k
Raw run time buckets in microseconds

Low Standard Deviation

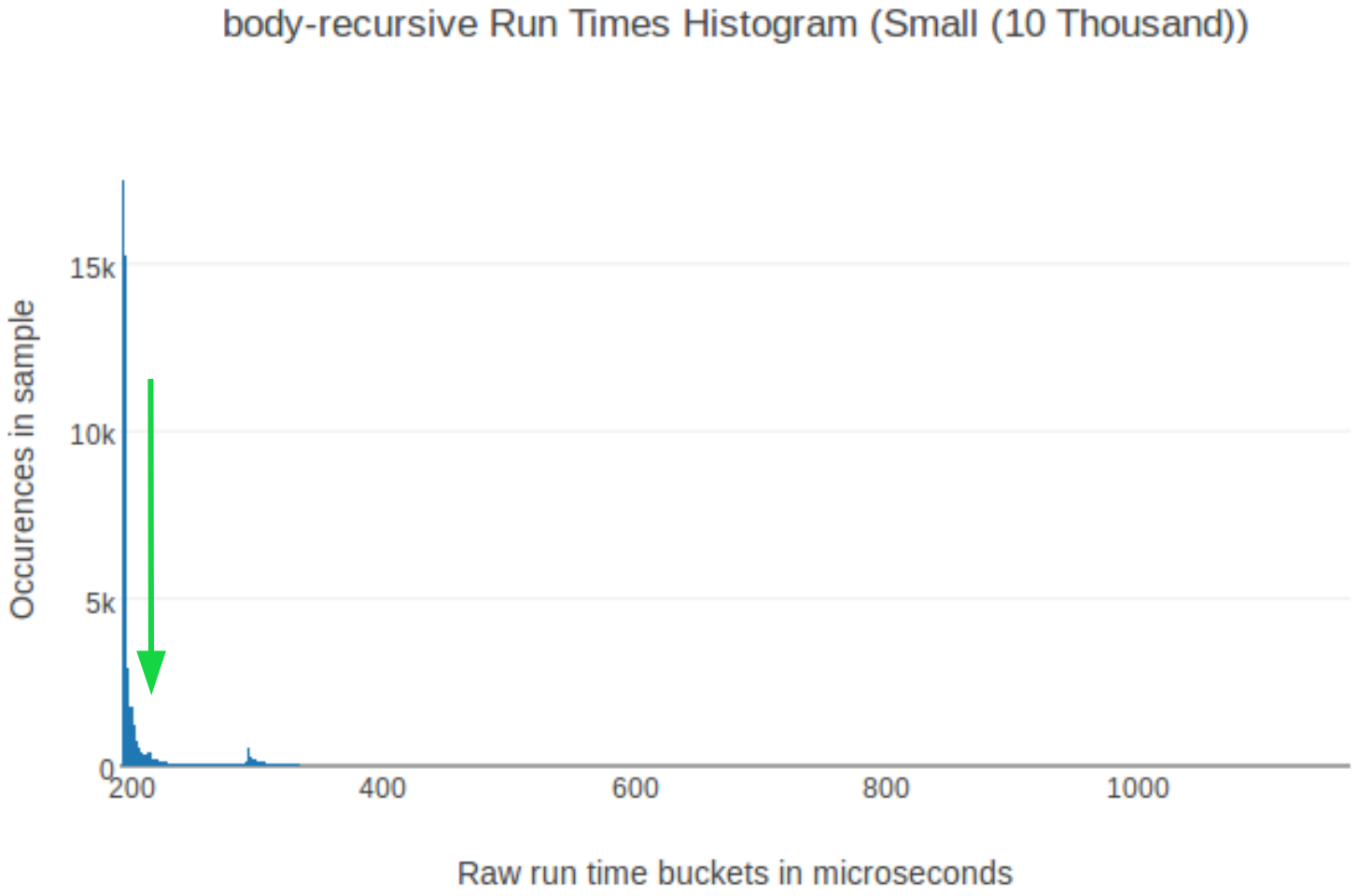



Standard Deviation
Average Iterations per Second
Iterations per Second
600
500
400
300
200
100
0
sort |> reverse
sort(fun)
sort_by(-value)

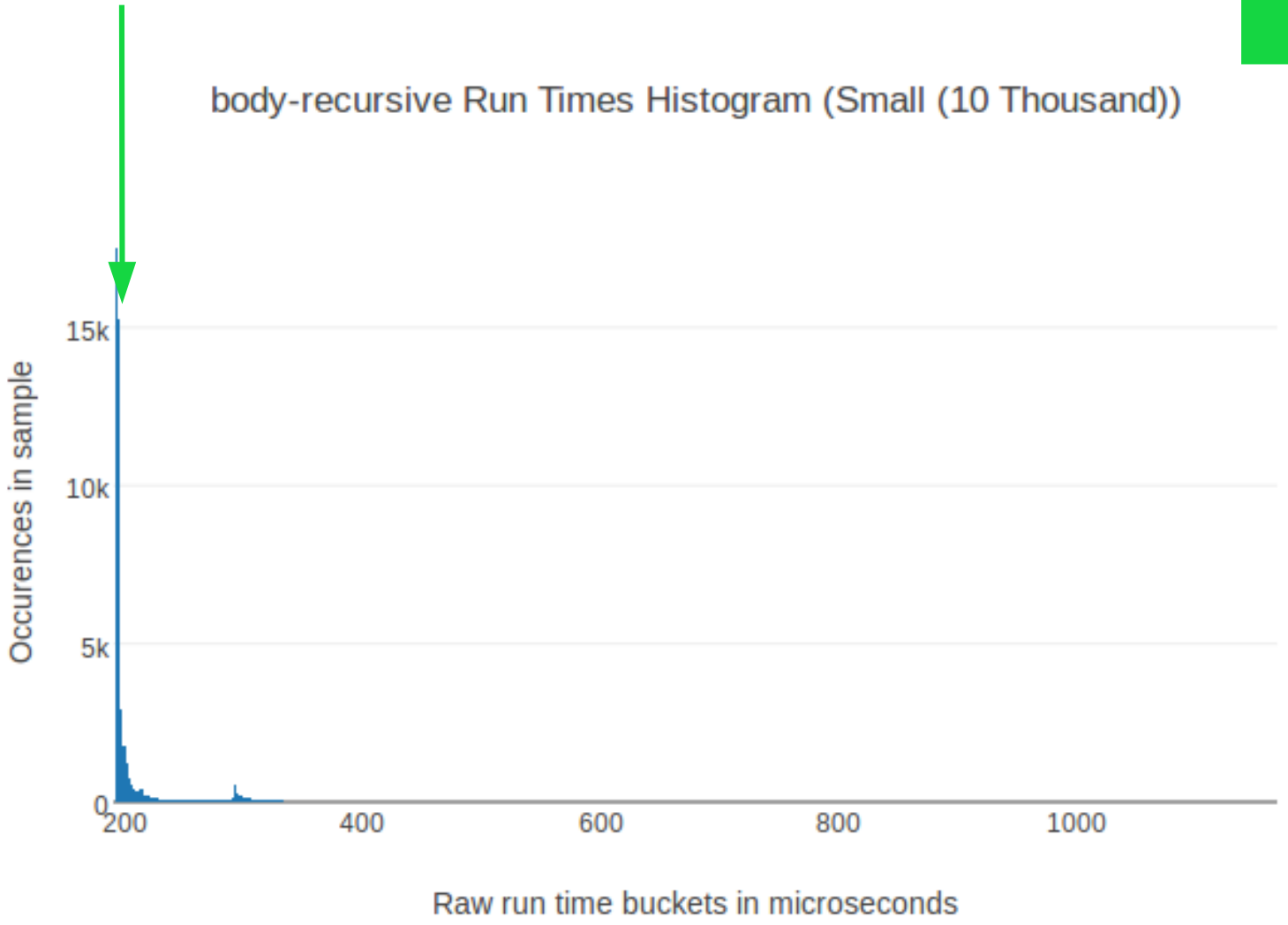
Median

```
defp compute_median(run_times, iterations) do
  sorted = Enum.sort(run_times)
  middle = div(iterations, 2)

  if Integer.is_odd(iterations) do
    sorted |> Enum.at(middle) |> to_float
  else
    (Enum.at(sorted, middle) +
      Enum.at(sorted, middle - 1)) / 2
  end
end
```

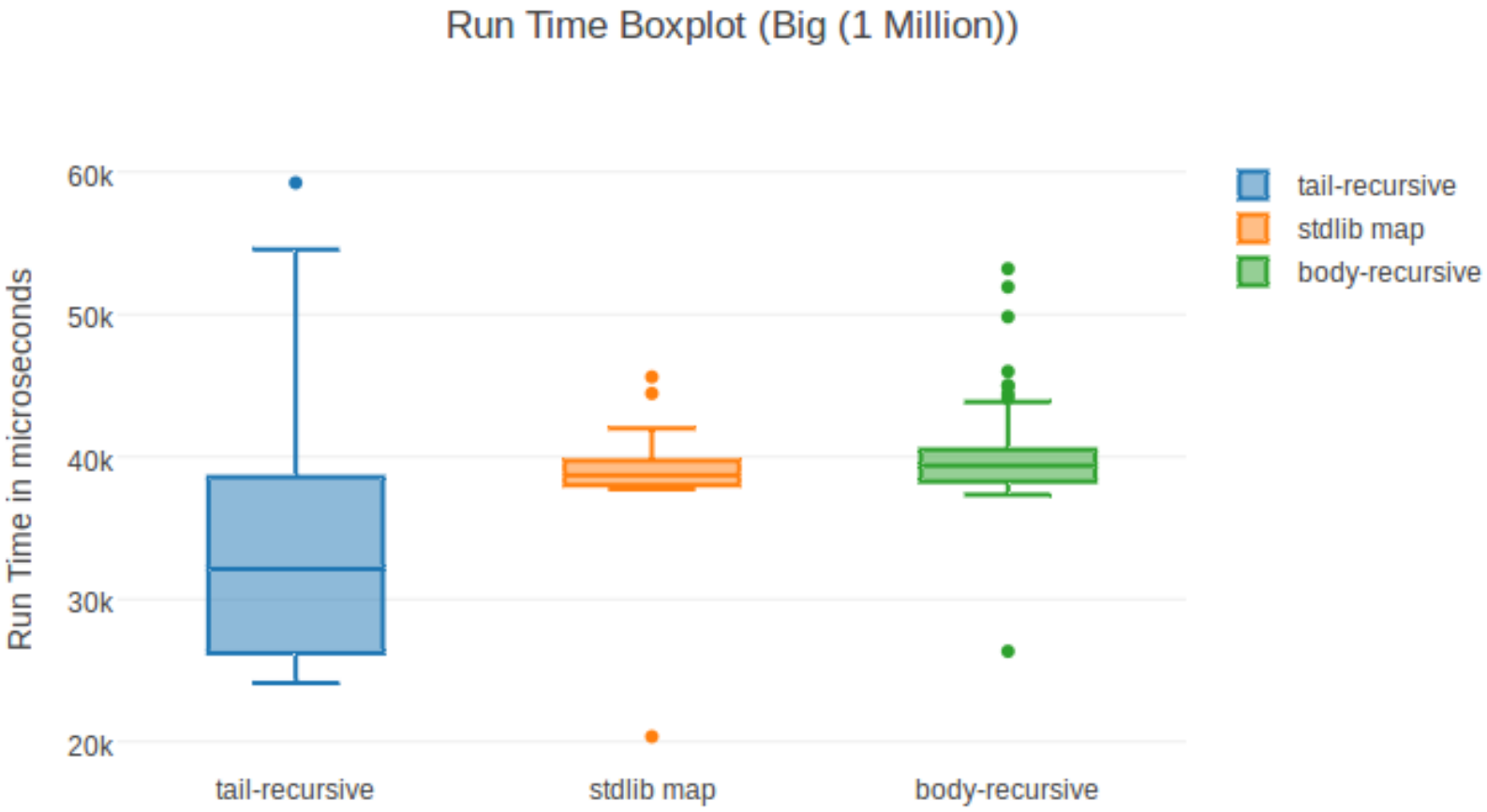
Average


body-recursive Run Times Histogram (Small (10 Thousand))
Occurences in sample
15k
10k
5k
0
200
400
600
800
1000
Raw run time buckets in microseconds

Median


body-recursive Run Times Histogram (Small (10 Thousand))
Occurences in sample
15k
10k
5k
0
200
400
600
800
1000
Raw run time buckets in microseconds

Boxplot


Run Time Boxplot (Big (1 Million))
Run Time in microseconds
60k
50k
40k
30k
20k
tail-recursive
stdlib map
body-recursive
tail-recursive
stdlib map
body-recursive

A transformation of inputs

```
config
|> Benchee.init
|> Benchee.system
|> Benchee.benchmark("job", fn -> magic end)
|> Benchee.measure
|> Benchee.statistics
|> Benchee.Formatters.Console.output
|> Benchee.Formatters.HTML.output
```

Structure

```
%{
  config: %{console: %{comparison: true, unit_scaling: :best},
    formatters: [...], inputs: nil, parallel: 1,
    print: %{benchmarking: true, configuration: true, fast_warning: true},
    time: 10000000, warmup: 2000000},
  system: %{elixir: "1.4.4", erlang: "19.3", ...},
  jobs: %{"flat_map" => #Function<1.132885813 in file:samples/run.exs>,
          "map.flatten" => #Function<2.132885813 in file:samples/run.exs>},
  run_times: %{
    __no_input: %{
      "flat_map" => [1546, 1154, 864, 1563, 1155, 942, ...],
      "map.flatten" => [1965, 1528, 2328, 1284, 1955, ...]
    }
  },
  statistics: %{
    __no_input: %{
      "flat_map" => %{average: 1025.0651905252316, ...},
      "map.flatten" => %{average: 1765.7764414573753, ...}
    }
  }
}
```

Structure

```
%{
  config: %{console: %{comparison: true, unit_scaling: :best},
    formatters: [...], inputs: nil, parallel: 1,
    print: %{benchmarking: true, configuration: true, fast_warning: true},
    time: 10000000, warmup: 2000000},
  system: %{elixir: "1.4.4", erlang: "19.3", ...},
  jobs: %{"flat_map" => #Function<1.132885813 in file:samples/run.exs>,
          "map.flatten" => #Function<2.132885813 in file:samples/run.exs>},
  run_times: %{
    __no_input: %{
      "flat_map" => [1546, 1154, 864, 1563, 1155, 942, ...],
      "map.flatten" => [1965, 1528, 2328, 1284, 1955, ...]
    }
  },
  statistics: %{
    __no_input: %{
      "flat_map" => %{average: 1025.0651905252316, ...},
      "map.flatten" => %{average: 1765.7764414573753, ...}
    }
  }
}
```

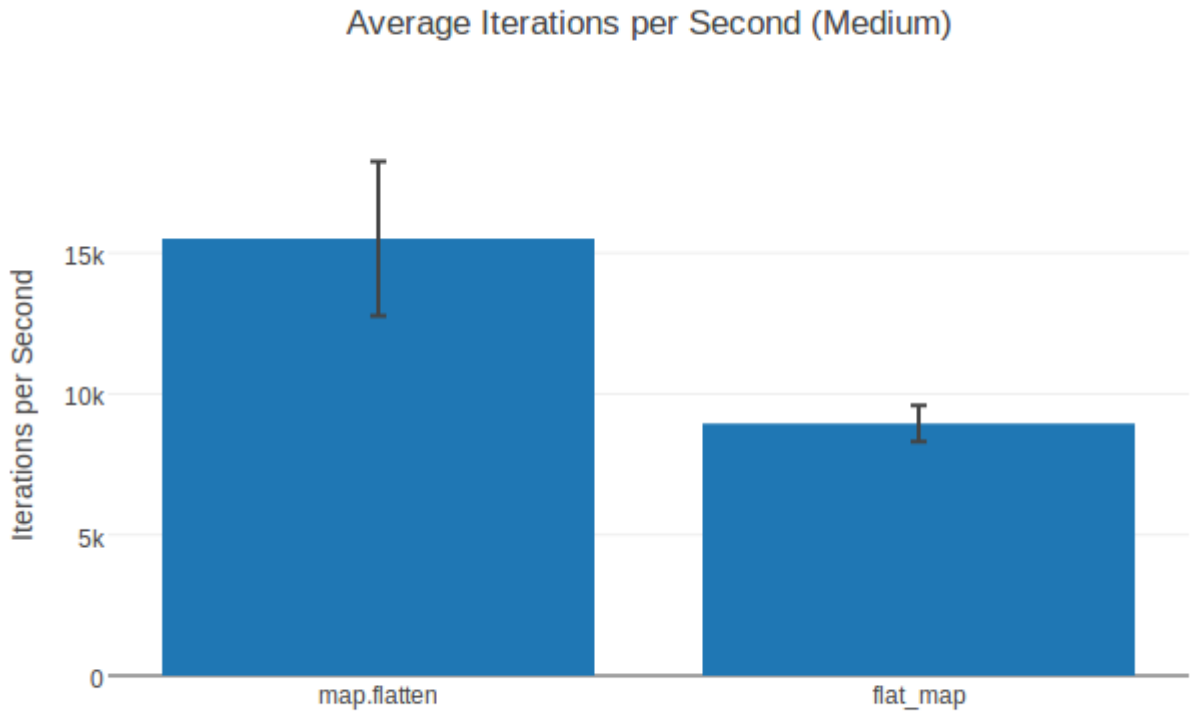
Plugin galore!

- benchee_csv
- benchee_json
- benchee_html

flat_map

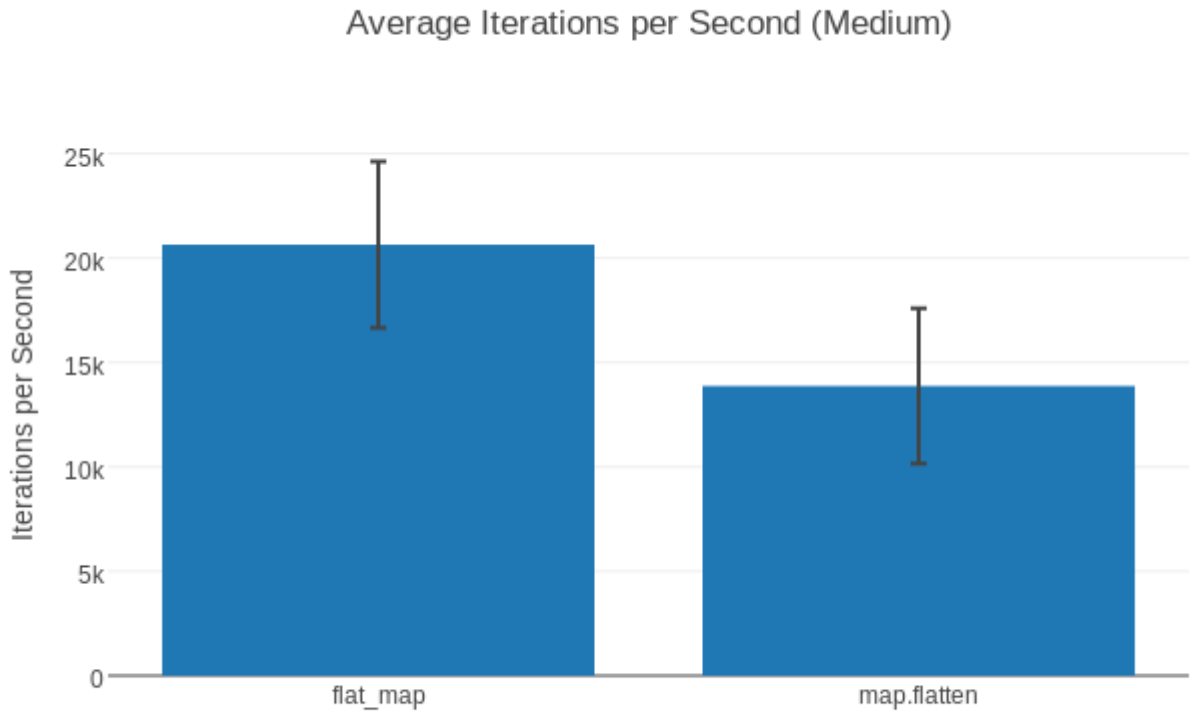
```elixir
map_fun = fn(i) -> [i, i * i] end

inputs = %{
  "Small"  => Enum.to_list(1..200),
  "Medium" => Enum.to_list(1..1000),
  "Bigger" => Enum.to_list(1..10_000)
}

Benchee.run(%{
  "flat_map" =>
    fn(list) -> Enum.flat_map(list, map_fun) end,
  "map.flatten" => fn(list) ->
    list
    |> Enum.map(map_fun)
    |> List.flatten
  end
}
```

Back in 1.3




```
##### With input Medium #####
Name                   ips        average  deviation         median
map.flatten        15.51 K       64.48 µs    ±17.66%       63.00 µs
flat_map            8.95 K      111.76 µs     ±7.18%      112.00 µs

Comparison:
map.flatten        15.51 K
flat_map            8.95 K - 1.73x slower
```

1.4!




```
##### With input Medium #####
Name                    ips        average  deviation         median
flat_map          20.63 K       48.46 µs    ±19.33%       48.00 µs
map.flatten       13.86 K       72.13 µs    ±26.86%       64.00 µs

Comparison:
flat_map          20.63 K
map.flatten       13.86 K - 1.49x slower
```

flat_map is slower than map |> flatten (and erlangs :lists.flatmap) #5082

Closed **PragTob** opened this issue on Aug 2 · 5 comments

PragTob commented on Aug 2 Contributor

(this is not strictly a bug but a somewhat surprising performance degradation I found, hope the issue tracker is still the appropriate place :))

josevalim commented on Aug 2 • edited

18 minutes later...

So Enum.flat_map never calls `++` and uses a reversal, that should theoretically consume less memory. `flat_map` also accepts any enumerable to be flattened while `:lists` can afford to work only on lists. In any case, can you please try these versions?

```elixir
# v1
def flat_map(enumerable, fun) when is_function(fun, 1) do
  Enum.reduce(enumerable, [], fn(entry, acc) ->
    case fun.(entry) do
      list when is_list(list) -> :lists.reverse(list, acc)
      other -> Enum.reduce(other, acc, &[&1 | &2])
    end
  end) |> :lists.reverse
end
```

```elixir
# v2
def flat_map_list([h | t], fun) do
  case fun.(h) do
    list when is_list(list) -> list ++ flat_map_list(t, fun)
    other -> Enum.to_list(other) ++ flat_map_list(t, fun)
  end
end
```

edit: forgot to mention, both versions are significantly faster for all use cases and v2 is even the fastest which is 🚀 🚀 🚀 - which is amazing :D It could have taken you at most 18 minutes or so :)

🎉 3

josevalim added a commit that closed this issue on Aug 2

`Optimize flat_map, closes #5082` 2dfdb36

josevalim commented on Aug 2

Thank you for benchmarking! ❤️

Enjoy Benchmarking ❤

Tack!

Tobias Pfeiffer
@PragTob
pragtob.info
github.com/PragTob/benchee

LIEFERY

Surprise findings

merge/2 vs merge/3

```elixir
base_map = (0..50)
           |> Enum.zip(300..350)
           |> Enum.into(%{})

# deep maps with 6 top level conflicts
orig = Map.merge base_map, some_deep_map
new = Map.merge base_map, some_deep_map_2

simple = fn(_key, _base, override) -> override end

Benchee.run %{
  "Map.merge/2" => fn -> Map.merge orig, new end,
  "Map.merge/3" =>
    fn -> Map.merge orig, new, simple end,
}
```

merge/2 vs merge/3

```
base_map = (0..50)
           |> Enum.zip(300..350)
           |> Enum.into(%{})

# deep maps with 6 top level conflicts
orig = Map.merge base_map, some_deep_map
new = Map.merge base_map, some_deep_map_2

simple = fn(_key, _base, override) -> override end

Benchee.run %{
  "Map.merge/2" => fn -> Map.merge orig, new end,
  "Map.merge/3" =>
    fn -> Map.merge orig, new, simple end,
}
```

merge/2 vs merge/3

Is merge/3 variant about...

- as fast as merge/2? (+-20%)
- 2x slower than merge/2
- 5x slower than merge/2
- 10x slower than merge/2
- 20x slower than merge/2

merge/2 vs merge/3

Is merge/3 variant about...

- as fast as merge/2? (+-20%)
- 2x slower than merge/2
- 5x slower than merge/2
- 10x slower than merge/2
- 20x slower than merge/2

merge/2 vs merge/3

Is merge/3 variant about...

- as fast as merge/2? (+-20%)
- 2x slower than merge/2
- 5x slower than merge/2
- 10x slower than merge/2
- 20x slower than merge/2

merge/2 vs merge/3

Is merge/3 variant about...

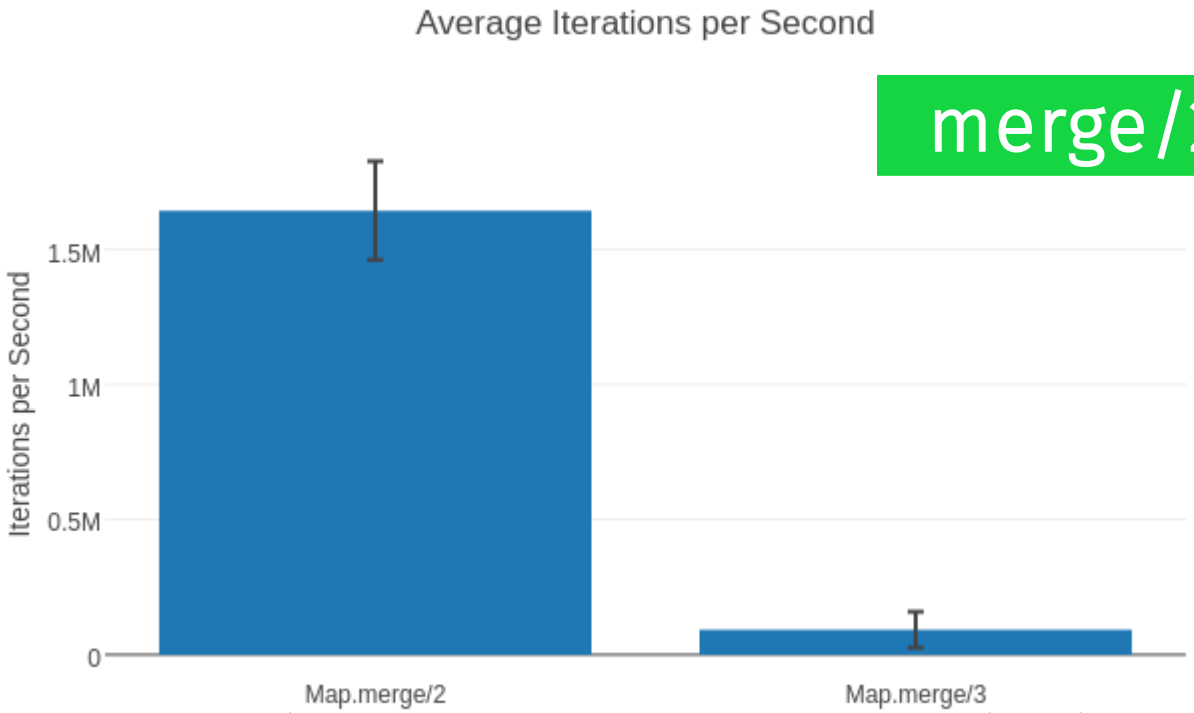
- as fast as merge/2? (+-20%)
- 2x slower than merge/2
- 5x slower than merge/2
- 10x slower than merge/2
- 20x slower than merge/2

merge/2 vs merge/3

Is merge/3 variant about...

- as fast as merge/2? (+-20%)
- 2x slower than merge/2
- 5x slower than merge/2
- 10x slower than merge/2
- 20x slower than merge/2

merge/2 vs merge/3




```
Name                  ips        average  deviation         median
Map.merge/2        1.64 M        0.61 µs    ±11.12%       0.61 µs
Map.merge/3      0.0921 M       10.86 µs    ±72.22%      10.00 µs

Comparison:
Map.merge/2        1.64 M
Map.merge/3      0.0921 M - 17.85x slower
```

Map again...

```elixir
defmodule MyMap do
  def map_tco(list, function) do
    Enum.reverse do_map_tco([], list, function)
  end

  defp do_map_tco(acc, [], _function) do
    acc
  end
  defp do_map_tco(acc, [head | tail], function) do
    do_map_tco([function.(head) | acc], tail, function)
  end
end
```

Argument order

```
defmodule MyMap do
  def map_tco_arg(list, function) do
    Enum.reverse do_map_tco_arg(list, function, [])
  end

  defp do_map_tco_arg([], _function, acc) do
    acc
  end
  defp do_map_tco_arg([head | tail], func, acc) do
    do_map_tco_arg(tail, func, [func.(head) | acc])
  end
end
```

Map again...

```elixir
defmodule MyMap do
  def map_tco(list, function) do
    Enum.reverse do_map_tco([], list, function)
  end

  defp do_map_tco(acc, [], _function) do
    acc
  end
  defp do_map_tco(acc, [head | tail], function) do
    do_map_tco([function.(head) | acc], tail, function)
  end
end
```

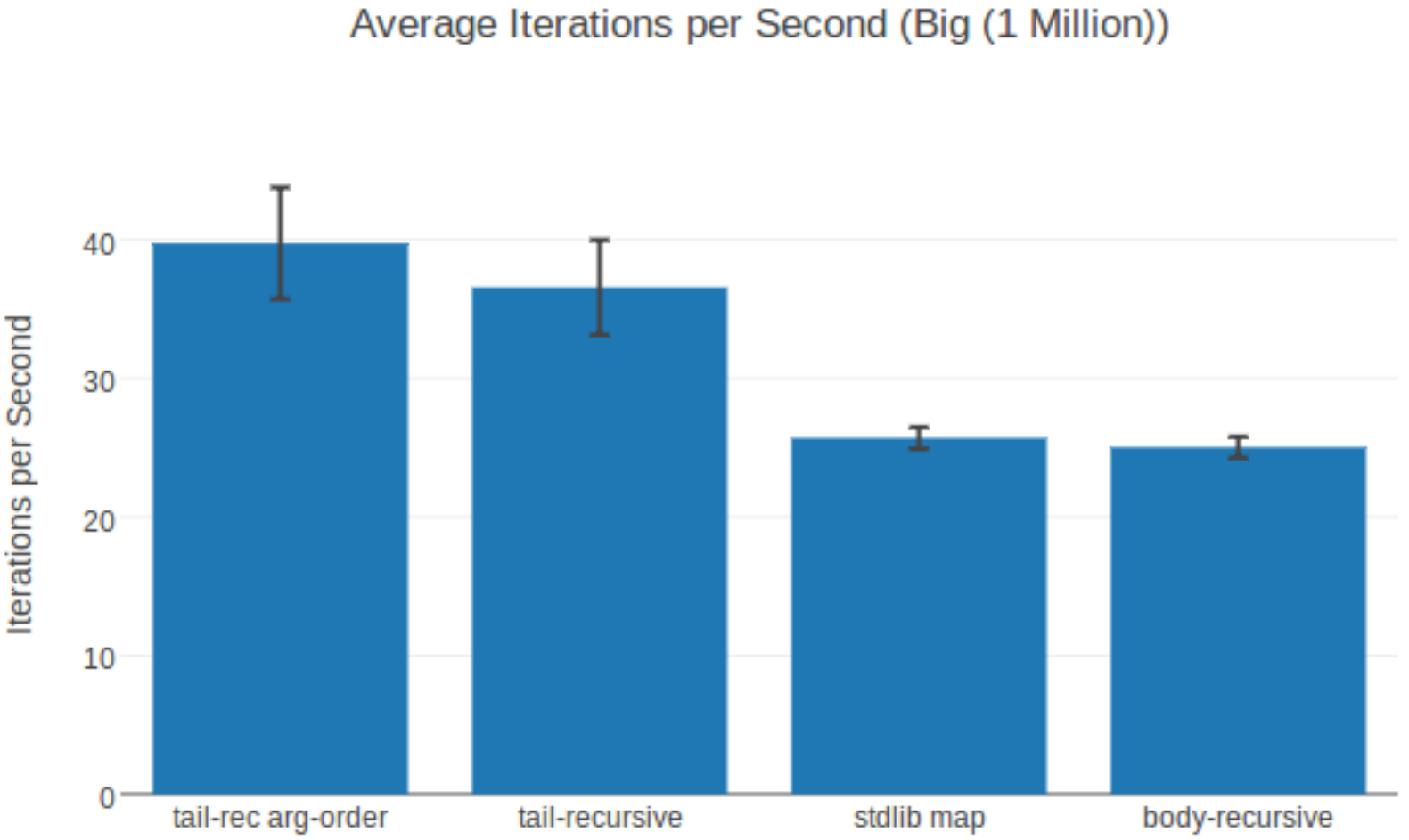
Argument order

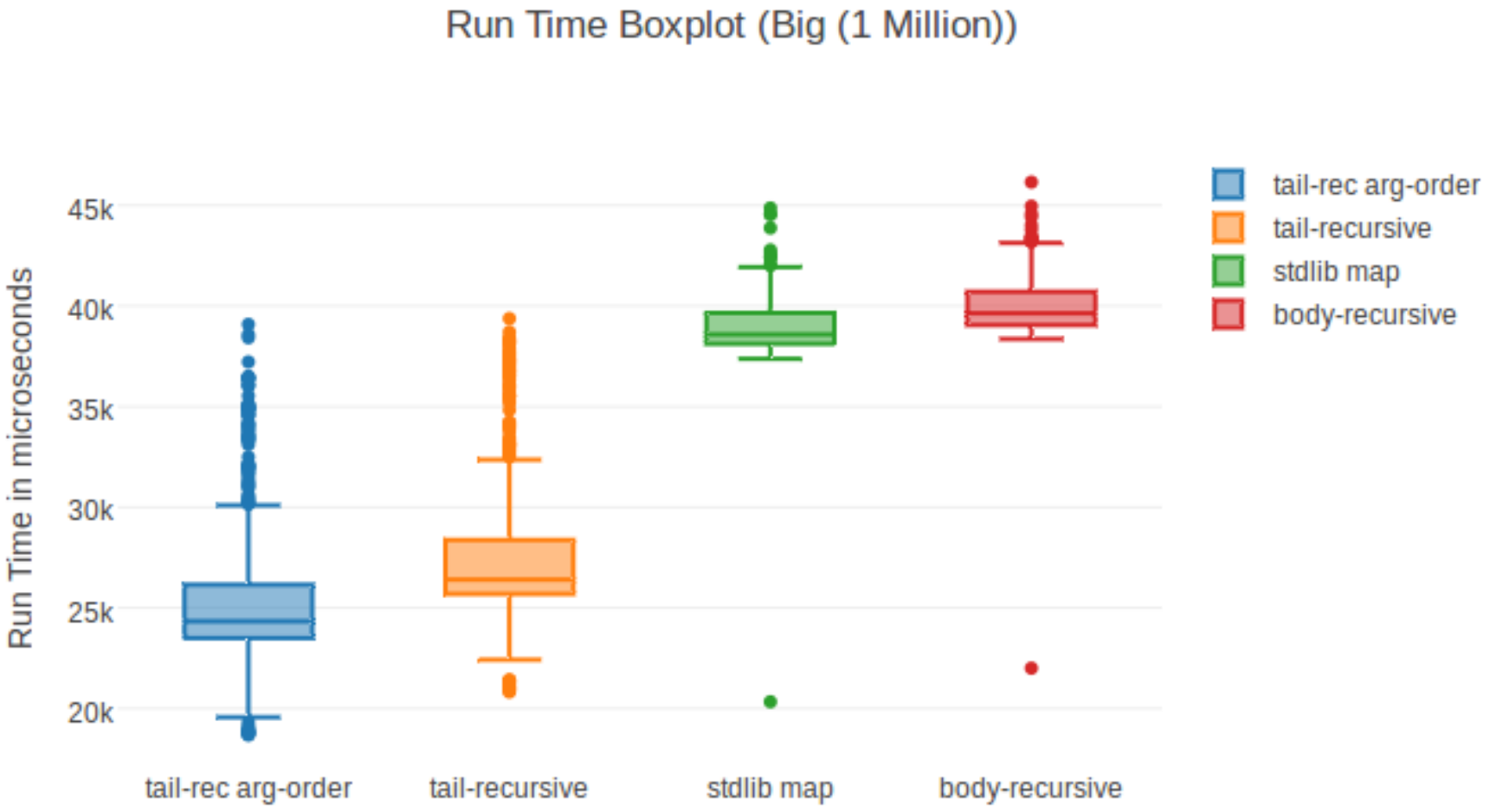
```
defmodule MyMap do
  def map_tco_arg(list, function) do
    Enum.reverse do_map_tco_arg(list, function, [])
  end

  defp do_map_tco_arg([], _function, acc) do
    acc
  end
  defp do_map_tco_arg([head | tail], func, acc) do
    do_map_tco_arg(tail, func, [func.(head) | acc])
  end
end
```

Does **argument order** make a difference?

```
##### With input Middle (100 Thousand) #####
Name                          ips        average  deviation         median
stdlib map                 490.02        2.04 ms     ±7.76%        2.07 ms
body-recursive             467.51        2.14 ms     ±7.34%        2.17 ms
tail-rec arg-order         439.04        2.28 ms    ±17.96%        2.25 ms
tail-recursive             402.56        2.48 ms    ±16.00%        2.46 ms

Comparison:
stdlib map                 490.02
body-recursive             467.51 - 1.05x slower
tail-rec arg-order         439.04 - 1.12x slower
tail-recursive             402.56 - 1.22x slower

##### With input Big (1 Million) #####
Name                          ips        average  deviation         median
tail-rec arg-order           39.76       25.15 ms    ±10.14%       24.33 ms
tail-recursive               36.58       27.34 ms     ±9.38%       26.41 ms
stdlib map                   25.70       38.91 ms     ±3.05%       38.58 ms
body-recursive               25.04       39.94 ms     ±3.04%       39.64 ms

Comparison:
tail-rec arg-order           39.76
tail-recursive               36.58 - 1.09x slower
stdlib map                   25.70 - 1.55x slower
body-recursive               25.04 - 1.59x slower
```

```
##### With input Middle (100 Thousand) #####
Name                          ips        average  deviation         median
stdlib map                 490.02        2.04 ms     ±7.76%        2.07 ms
body-recursive             467.51        2.14 ms     ±7.34%        2.17 ms
tail-rec arg-order         439.04        2.28 ms    ±17.96%        2.25 ms
tail-recursive             402.56        2.48 ms    ±16.00%        2.46 ms

Comparison:
stdlib map                 490.02
body-recursive             467.51 - 1.05x slower
tail-rec arg-order         439.04 - 1.12x slower
tail-recursive             402.56 - 1.22x slower

##### With input Big (1 Million) #####
Name                          ips        average  deviation         median
tail-rec arg-order          39.76       25.15 ms    ±10.14%       24.33 ms
tail-recursive              36.58       27.34 ms     ±9.38%       26.41 ms
stdlib map                  25.70       38.91 ms     ±3.05%       38.58 ms
body-recursive              25.04       39.94 ms     ±3.04%       39.64 ms

Comparison:
tail-rec arg-order          39.76
tail-recursive              36.58 - 1.09x slower
stdlib map                  25.70 - 1.55x slower
body-recursive              25.04 - 1.59x slower
```


Average Iterations per Second (Big (1 Million))
Iterations per Second
40
30
20
10
0
tail-rec arg-order
tail-recursive
stdlib map
body-recursive


Run Time Boxplot (Big (1 Million))
Run Time in microseconds
45k
40k
35k
30k
25k
20k
tail-rec arg-order
tail-recursive
stdlib map
body-recursive
tail-rec arg-order
tail-recursive
stdlib map
body-recursive


But…it
Can not be!

A wild José appears!

"The order of arguments will likely matter when we generate the branching code. The order of arguments will specially matter if performing binary matching."

José Valim, 2016

(Comment Section of my blog!)

TCO

```elixir
defmodule MyMap do
  def map_tco(list, function) do
    Enum.reverse do_map_tco([], list, function)
  end

  defp do_map_tco(acc, [], _function) do
    acc
  end
  defp do_map_tco(acc, [head | tail], func) do
    do_map_tco([func.(head) | acc], tail, func)
  end

  def map_body([], _func), do: []
  def map_body([head | tail], func) do
    [func.(head) | map_body(tail, func)]
  end
end
```

TCO

```
defmodule MyMap do
  def map_tco(list, function) do
    Enum.reverse do_map_tco([], list, function)
  end

  defp do_map_tco(acc, [], _function) do
    acc
  end
  defp do_map_tco(acc, [head | tail], func) do
    do_map_tco([func.(head) | acc], tail, func)
  end

  def map_body([], _func), do: []
  def map_body([head | tail], func) do
    [func.(head) | map_body(tail, func)]
  end
end
```

TCO

```
map_fun = fn(i) -> i + 1 end
inputs = %{
  "Small (10 Thousand)"   => Enum.to_list(1..10_000),
  "Middle (100 Thousand)" => Enum.to_list(1..100_000),
  "Big (1 Million)"       => Enum.to_list(1..1_000_000),
  "Bigger (5 Million)"    => Enum.to_list(1..5_000_000)
}

Benchee.run %{
  "tail-recursive" =>
    fn(list) -> MyMap.map_tco(list, map_fun) end,
  "stdlib map" =>
    fn(list) -> Enum.map(list, map_fun) end,
  "body-recursive" =>
    fn(list) -> MyMap.map_body(list, map_fun) end
}
```

```elixir
map_fun = fn(i) -> i + 1 end
inputs = %{
  "Small (10 Thousand)"   => Enum.to_list(1..10_000),
  "Middle (100 Thousand)" => Enum.to_list(1..100_000),
  "Big (1 Million)"       => Enum.to_list(1..1_000_000),
  "Bigger (5 Million)"    => Enum.to_list(1..5_000_000)
}

Benchee.run %{
  "tail-recursive" =>
    fn(list) -> MyMap.map_tco(list, map_fun) end,
  "stdlib map" =>
    fn(list) -> Enum.map(list, map_fun) end,
  "body-recursive" =>
    fn(list) -> MyMap.map_body(list, map_fun) end
}
```

TCO

TCO

```
map_fun = fn(i) -> i + 1 end
inputs = %{
  "Small (10 Thousand)"   => Enum.to_list(1..10_000),
  "Middle (100 Thousand)" => Enum.to_list(1..100_000),
  "Big (1 Million)"       => Enum.to_list(1..1_000_000),
  "Bigger (5 Million)"    => Enum.to_list(1..5_000_000)
}
```

```
Benchee.run %{
  "tail-recursive" =>
    fn(list) -> MyMap.map_tco(list, map_fun) end,
  "stdlib map" =>
    fn(list) -> Enum.map(list, map_fun) end,
  "body-recursive" =>
    fn(list) -> MyMap.map_body(list, map_fun) end
}
```

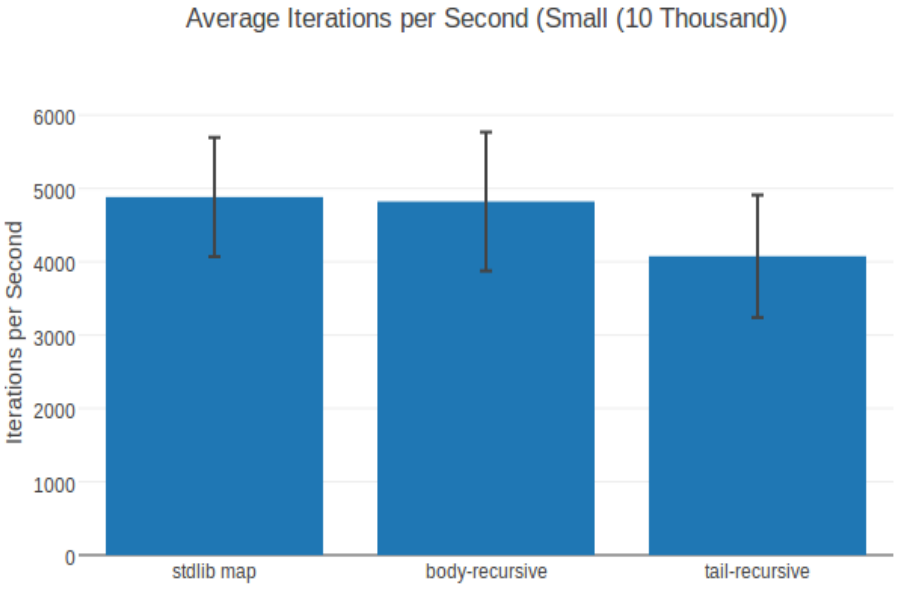
TCO

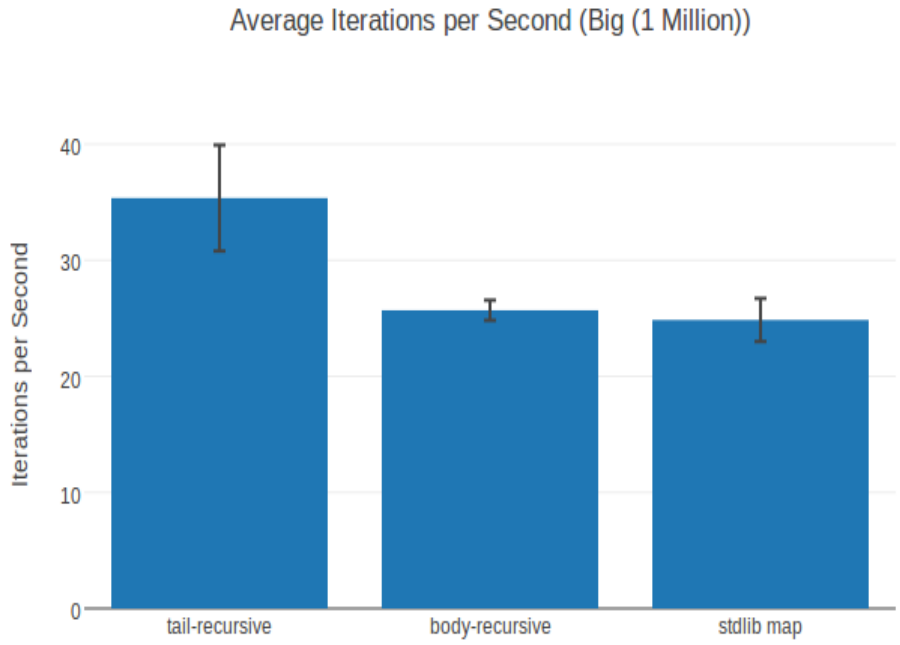
```
##### With input Small (10 Thousand) #####
Comparison:
body-recursive        5.12 K
stdlib map            5.07 K - 1.01x slower
tail-recursive        4.38 K - 1.17x slower

##### With input Middle (100 Thousand) #####
Comparison:
body-recursive        491.16
stdlib map            488.45 - 1.01x slower
tail-recursive        399.08 - 1.23x slower

##### With input Big (1 Million) #####
Comparison:
tail-recursive         35.36
body-recursive         25.69 - 1.38x slower
stdlib map             24.85 - 1.42x slower

##### With input Bigger (5 Million) #####
Comparison:
tail-recursive          6.93
body-recursive          4.92 - 1.41x slower
stdlib map              4.87 - 1.42x slower
```

TCO

```
##### With input Small (10 Thousand) #####
Comparison:
body-recursive        5.12 K
stdlib map            5.07 K - 1.01x slower
tail-recursive        4.38 K - 1.17x slower

##### With input Middle (100 Thousand) #####
Comparison:
body-recursive        491.16
stdlib map            488.45 - 1.01x slower
tail-recursive        399.08 - 1.23x slower

##### With input Big (1 Million) #####
Comparison:
tail-recursive         35.36
body-recursive         25.69 - 1.38x slower
stdlib map             24.85 - 1.42x slower

##### With input Bigger (5 Million) #####
Comparison:
tail-recursive          6.93
body-recursive          4.92 - 1.41x slower
stdlib map              4.87 - 1.42x slower
```

TCO


Average Iterations per Second (Small (10 Thousand))
Iterations per Second
6000
5000
4000
3000
2000
1000
0
stdlib map
body-recursive
tail-recursive


Average Iterations per Second (Big (1 Million))
Iterations per Second
40
30
20
10
0
tail-recursive
body-recursive
stdlib map