Why is Dual-Pivot Quicksort Fast?

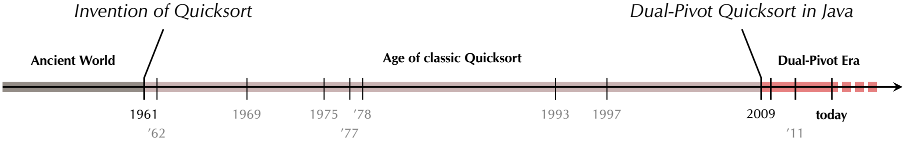
Sebastian Wild
wild@cs.uni-kl.de

TECHNISCHE UNIVERSITÄT KAISERSLAUTERN

29 September 2015

Theorietage 2015 Speyer

Sorting History



Sorting History

1961,62 **Hoare:** publication, first analysis

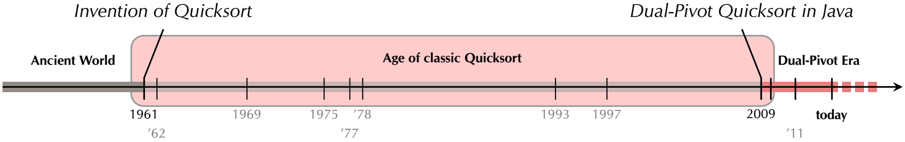
1969 **Singleton:** median-of-three & Insertionsort on small subarrays

1975-78 **Sedgewick:** analysis of many optimizations

1993 **Bentley, McIlroy:** duplicate elements & "ninther"

1997 **Musser:** $\mathcal{O}(n \log n)$ worst case by truncating recursion

$\rightsquigarrow$ Basic algorithm settled since 1961; latest tweaks from 1990's.
Since then: Almost identical in all programming libraries!



Sorting History

1961,62 **Hoare:** publication, first analysis

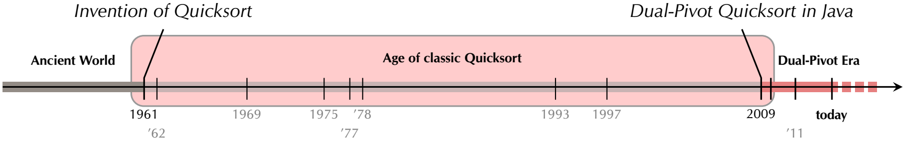
1969 **Singleton:** median-of-three & Insertionsort on small subarrays

1975-78 **Sedgewick:** analysis of many optimizations

1993 **Bentley, McIlroy:** duplicate elements & "ninther"

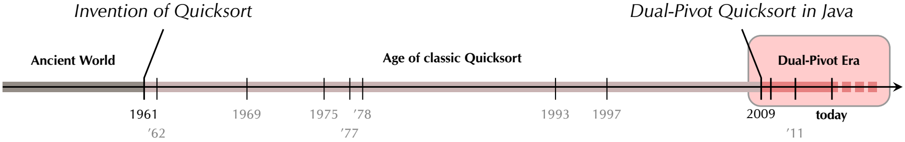
1997 **Musser:** $\mathcal{O}(n \log n)$ worst case by truncating recursion

$\rightsquigarrow$ Basic algorithm settled since 1961; latest tweaks from 1990's.
Since then: Almost identical in all programming libraries!



Sorting History

2008 – 2009	**Vladimir Yaroslavskiy** (developer at Sun) experiments with Quicksort with **two pivots**
11 Sep 2009	announcement on Java core library mailing list
29 Oct 2009	**inclusion** in development version of OpenJDK
2009 – 2011	optimizations by Joshua Bloch, Jon Bentley and others
28 July 2011	**public release** of Java 7 with Yaroslavskiy's Quicksort



Dual-Pivot Quicksort

Algorithm (Conceptual View)

1. Choose **two pivots** $P \leqslant Q$
2. For each element x, determine its **class**
    - **small** for $x < P$
    - **medium** for $P < x < Q$
    - **large** for $Q < x$

   by comparing x to **pivots** P and Q
3. Arrange elements according to classes:

4. Sort subarrays recursively.

How to implement 3 efficiently on arrays?

Dual-Pivot Quicksort

Algorithm (Conceptual View)

1. Choose **two pivots** $P \leqslant Q$
2. For each element x, determine its **class**
   - **small** for $x < P$
   - **medium** for $P < x < Q$
   - **large** for $Q < x$

   by comparing x to **pivots** P and Q
3. Arrange elements according to classes:

4. Sort subarrays recursively.

How to implement 3 efficiently on arrays?

Dual-Pivot Quicksort

Algorithm (Conceptual View)

1. Choose **two pivots** $P \leqslant Q$
2. For each element x, determine its **class**
   - **small** for $x < P$
   - **medium** for $P < x < Q$
   - **large** for $Q < x$

   by comparing x to **pivots** P and Q
3. Arrange elements according to classes:

4. Sort subarrays recursively.

How to implement 3 efficiently on arrays?

Dual-Pivot Quicksort

Algorithm (Conceptual View)

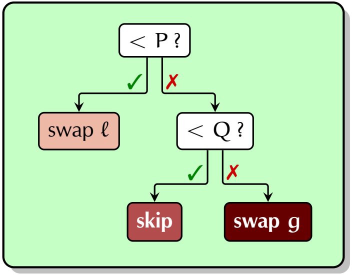
1. Choose **two pivots** $P \leqslant Q$
2. For each element x, determine its **class**
   - **small** for $x < P$
   - **medium** for $P < x < Q$
   - **large** for $Q < x$

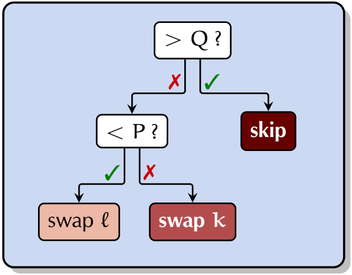
   by comparing x to **pivots** P and Q
3. Arrange elements according to classes:

4. Sort subarrays recursively.

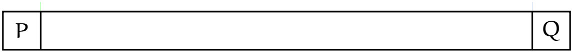
How to implement ❸ efficiently on arrays?

Yaroslavskiy's Algorithm

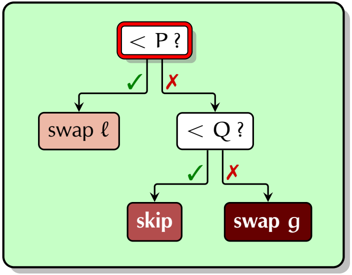


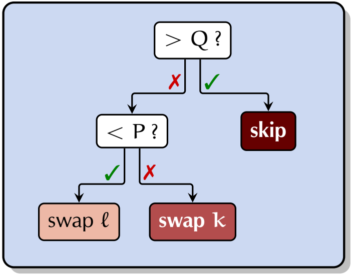


Invariant:

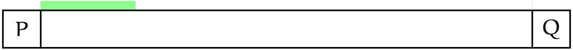


Yaroslavskiy's Algorithm

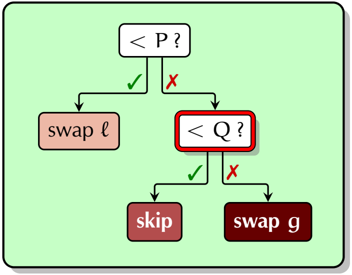


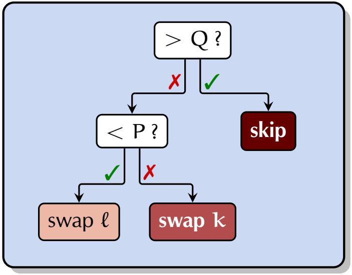


Invariant:

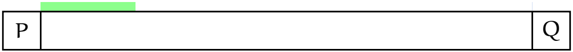


Yaroslavskiy's Algorithm

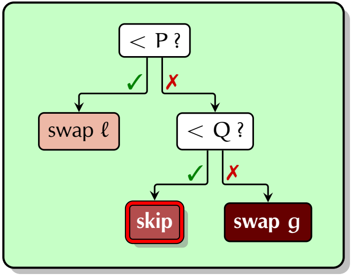


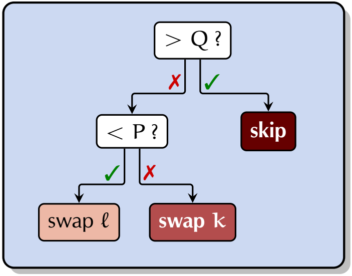


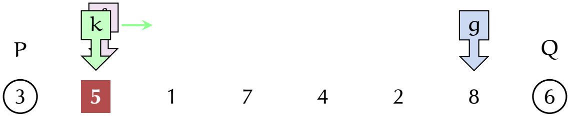
Invariant:



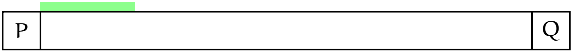
Yaroslavskiy's Algorithm



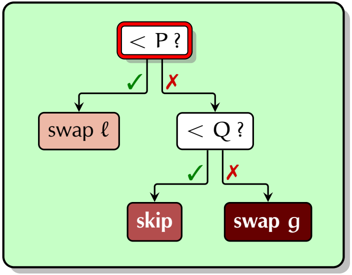


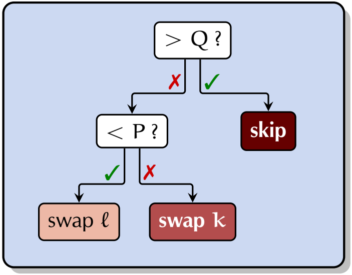


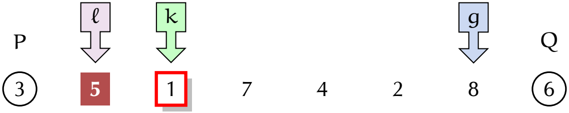
Invariant:



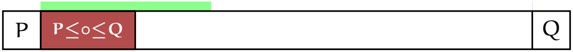
Yaroslavskiy's Algorithm



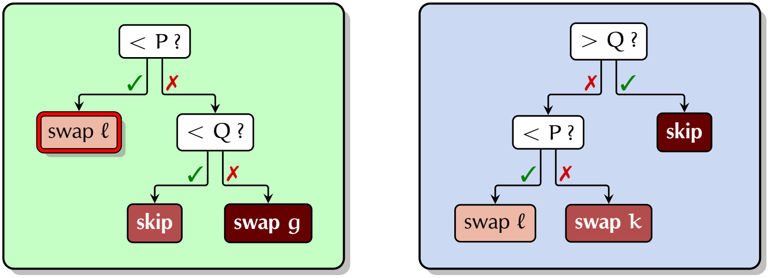




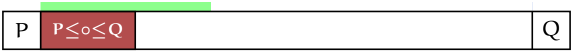
Invariant:



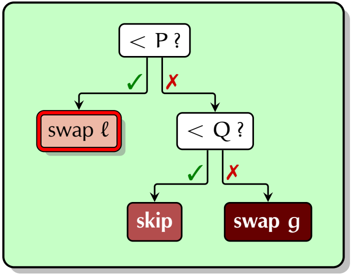
Yaroslavskiy's Algorithm

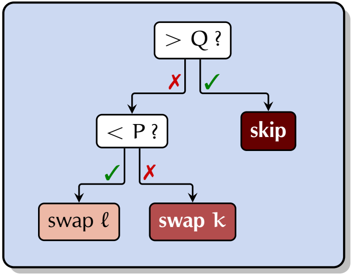


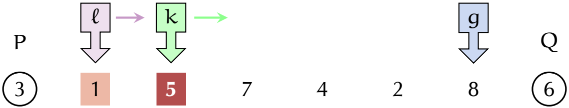
Invariant:



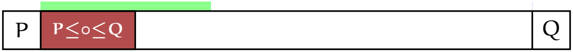
Yaroslavskiy's Algorithm



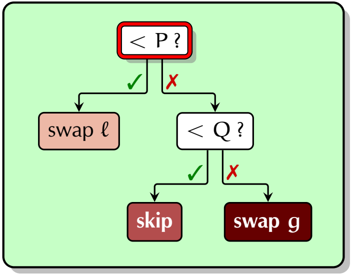


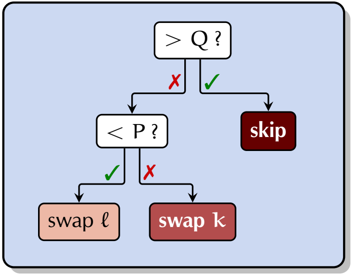


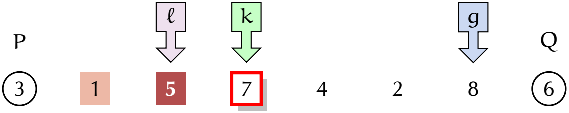
Invariant:



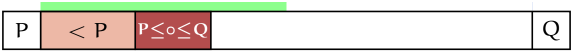
Yaroslavskiy's Algorithm



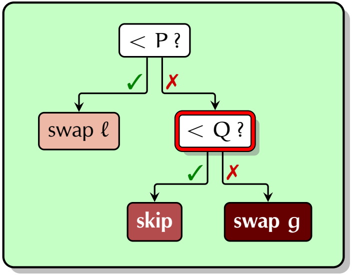


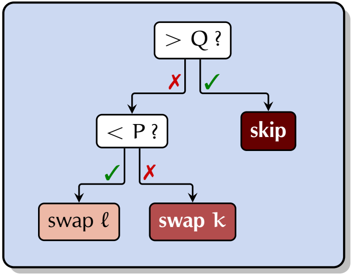


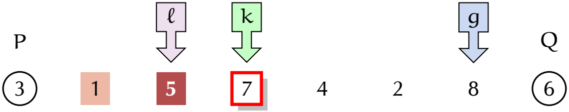
Invariant:



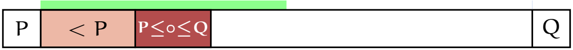
Yaroslavskiy's Algorithm



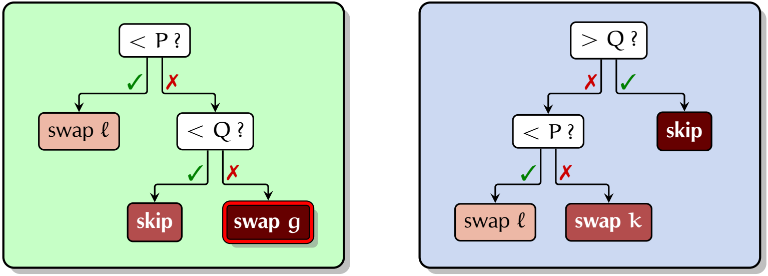


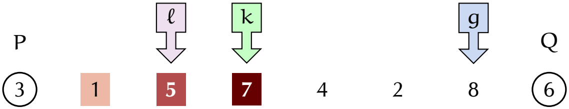


Invariant:

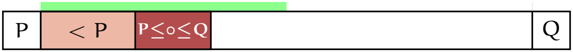


Yaroslavskiy's Algorithm

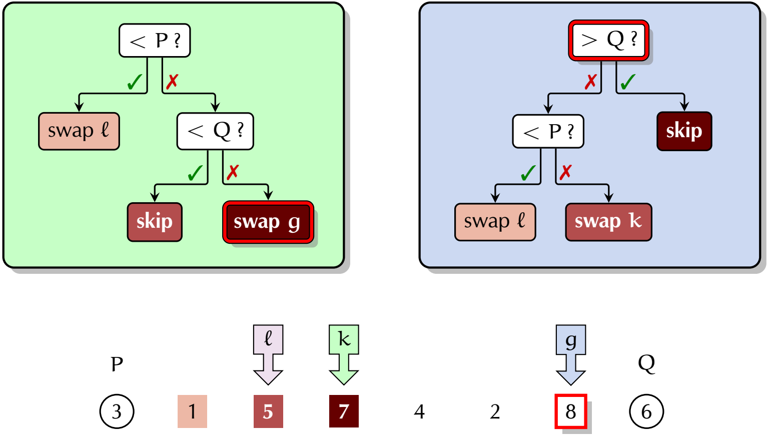




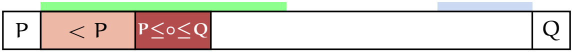
Invariant:



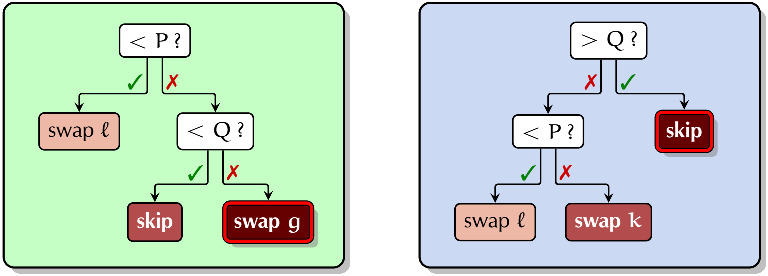
Yaroslavskiy's Algorithm

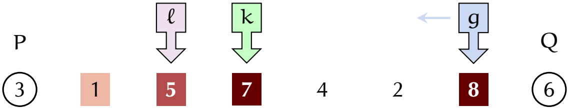


Invariant:

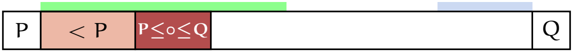


Yaroslavskiy's Algorithm

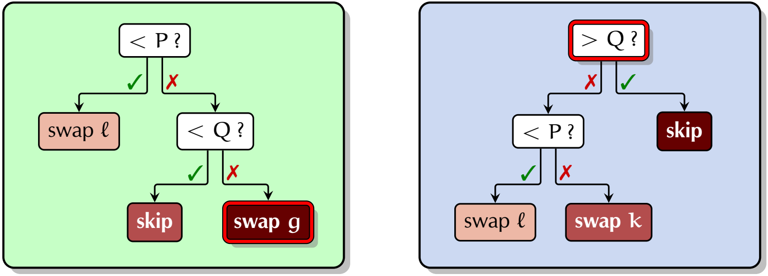


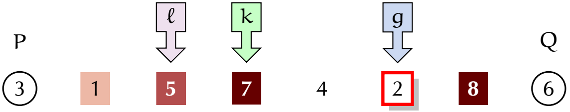


Invariant:

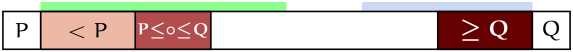


Yaroslavskiy's Algorithm

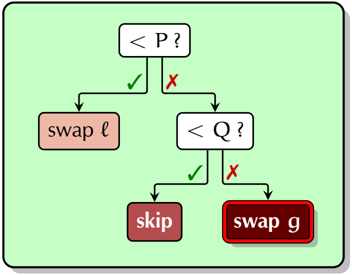


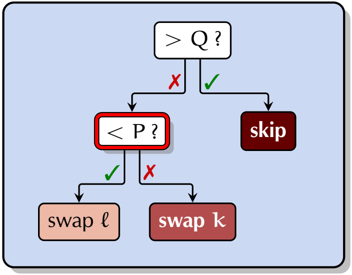


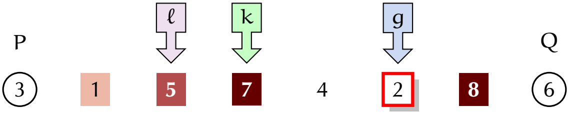
Invariant:



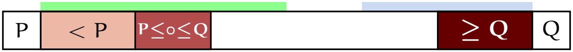
Yaroslavskiy's Algorithm



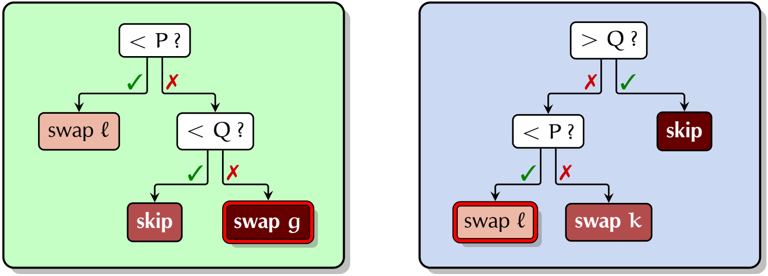


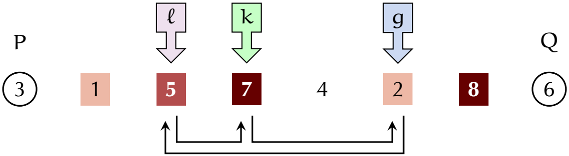


Invariant:

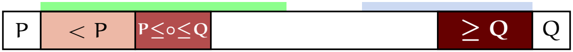


Yaroslavskiy's Algorithm

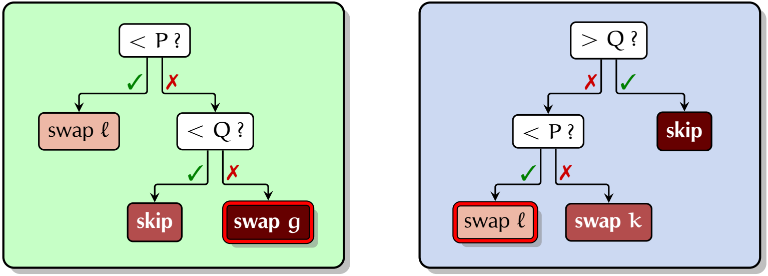


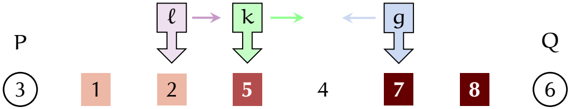


Invariant:

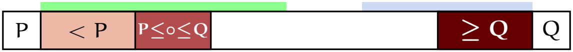


Yaroslavskiy's Algorithm

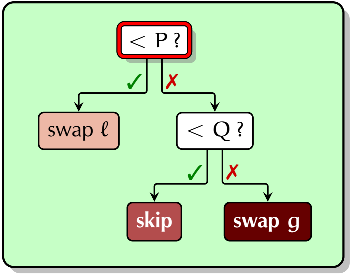


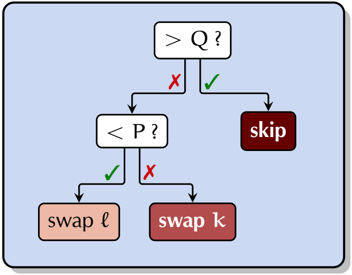


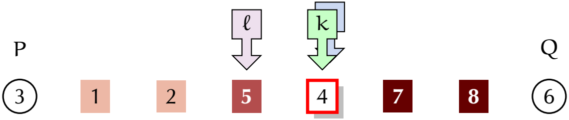
Invariant:



Yaroslavskiy's Algorithm

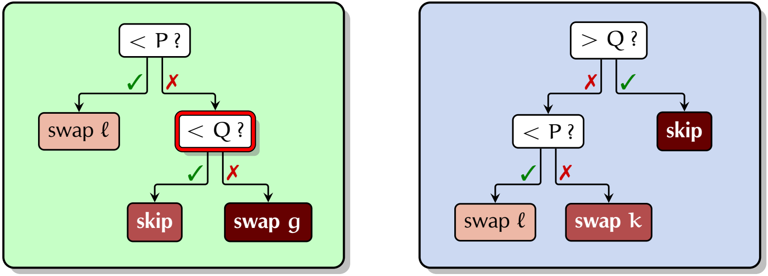


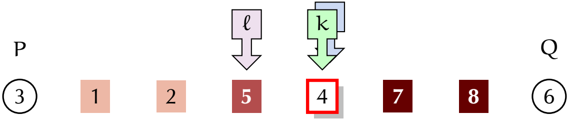




Invariant:

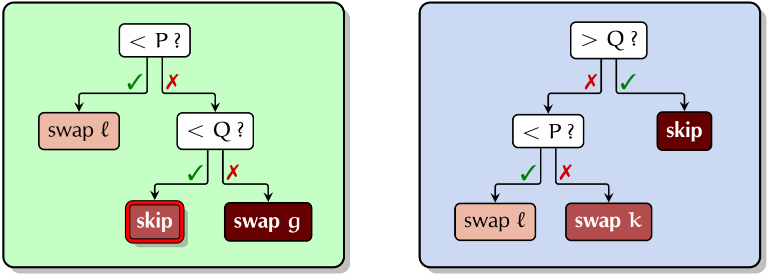
Yaroslavskiy's Algorithm

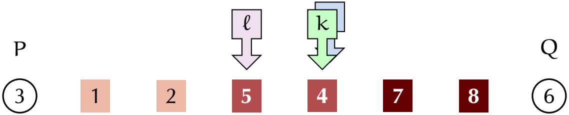




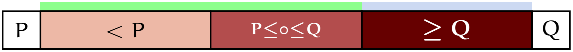
Invariant:

Yaroslavskiy's Algorithm

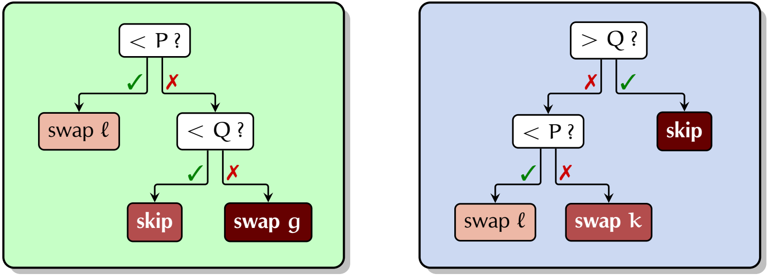




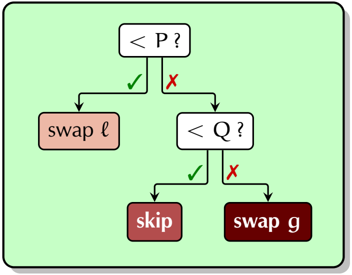
Invariant:

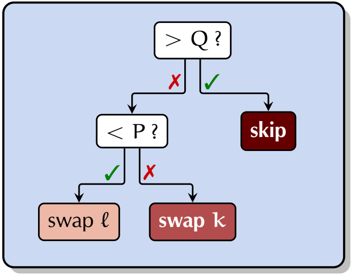


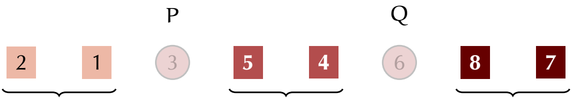
Yaroslavskiy's Algorithm



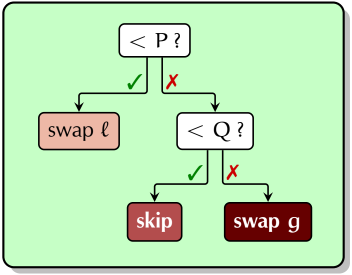
Yaroslavskiy's Algorithm

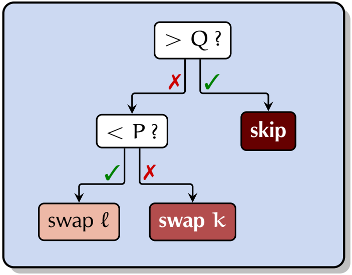






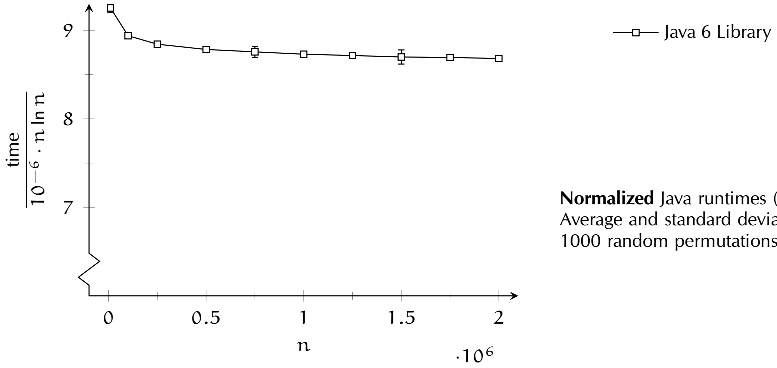
Yaroslavskiy's Algorithm





Running Time Experiments

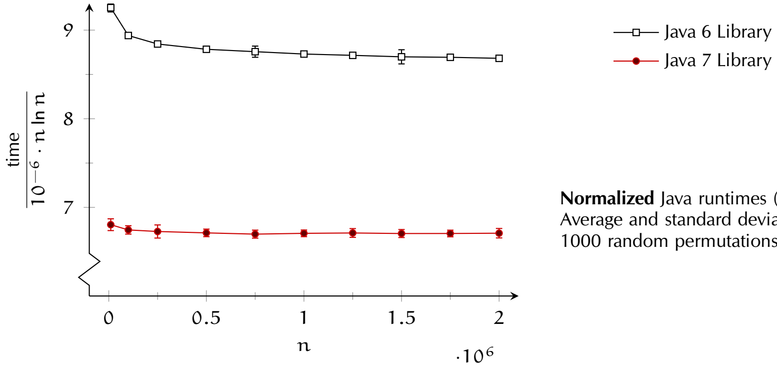
Why switch to new, unknown algorithm?



Normalized Java runtimes (in *ms*).
Average and standard deviation of 1000 random permutations per size.

Running Time Experiments

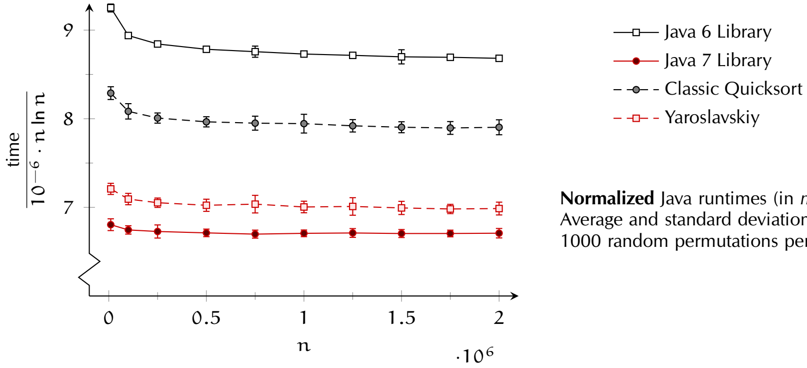
Why switch to new, unknown algorithm? Because it is faster!



Normalized Java runtimes (in *ms*).
Average and standard deviation of
1000 random permutations per size.

Running Time Experiments

Why switch to new, unknown algorithm? Because it is faster!

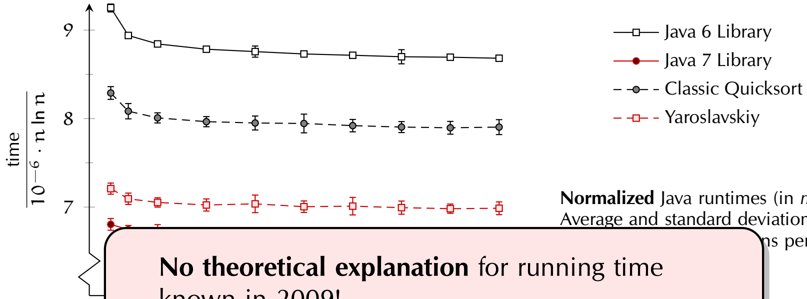


Normalized Java runtimes (in *ms*).
Average and standard deviation of 1000 random permutations per size.

- remains true for **basic** variants of algorithms: vs. !

Running Time Experiments

Why switch to new, unknown algorithm? Because it is faster!



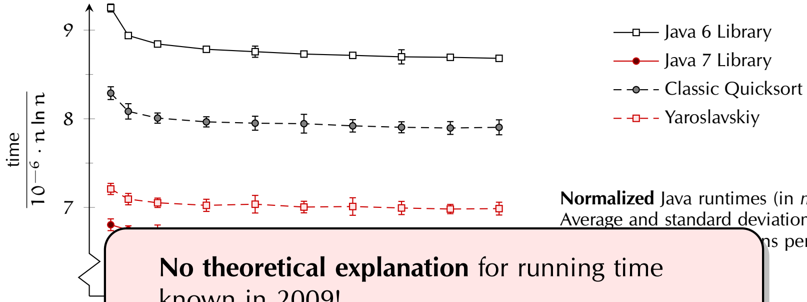
Normalized Java runtimes (in *ms*).
Average and standard deviation of
ns per size.

No theoretical explanation for running time known in 2009!

- remain

Running Time Experiments

Why switch to new, unknown algorithm? Because it is faster!



Normalized Java runtimes (in *ms*). Average and standard deviation of ns per size.

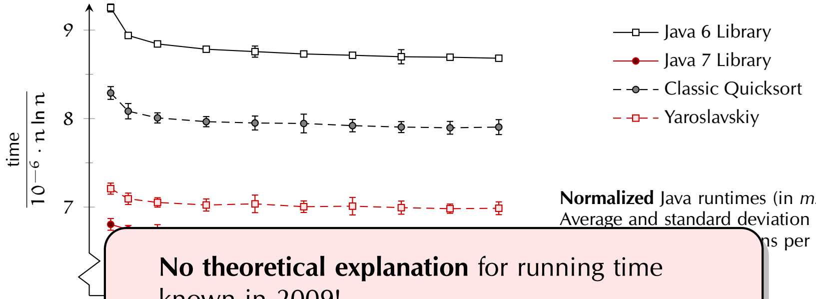
No theoretical explanation for running time known in 2009!

Only lucky experiments?

- remain

Running Time Experiments

Why switch to new, unknown algorithm? Because it is faster!



Normalized Java runtimes (in *ms*). Average and standard deviation of ... ns per size.

No theoretical explanation for running time known in 2009!

Only lucky experiments?

Why did noone come up with this earlier?

- remain

Track Record of Dual-Pivot Quicksort

- **Observation in practice:**
  Yaroslavskiy's Quicksort (YQS) **faster** than *classic Quicksort (CQS)* … **why?**

$\rightsquigarrow$ We did a mathematical analysis of YQS.

- **Traditional** cost measures do **not** explain observation!

	CQS	YQS	Relative
Running Time	(from various experiments)		$-10 \pm 2\%$
Comparisons	2	1.9	-5%
Swaps	$0.\overline{3}$	0.6	$+80\%$
Bytecode Instructions	18	21.7	$+20.6\%$
MMIX oops υ	11	13.1	$+19.1\%$
MMIX mems μ	$2.\overline{6}$	2.8	$+5\%$
Scanned Elements	2	1.6	-20%
Branch Mispredictions	0.57	0.58	$+2\%$

$\cdot n \ln n + O(n)$, average case results

Track Record of Dual-Pivot Quicksort

- **Observation in practice:**
  Yaroslavskiy's Quicksort (YQS) **faster** than *classic Quicksort (CQS)* ... **why?**

⇝ We did a mathematical analysis of YQS.

- **Traditional** cost measures do **not** explain observation!

	CQS	**YQS**	**Relative**
Running Time	(from various experiments)		**−10±2%**
Comparisons	2	1.9	−5%
Swaps	$0.\overline{3}$	0.6	+80%
Bytecode Instructions	18	21.7	+20.6%
MMIX oops υ	11	13.1	+19.1%
MMIX mems μ	$2.\overline{6}$	2.8	+5%
Scanned Elements	2	1.6	−20%
Branch Mispredictions	0.57	0.58	+2%

Track Record of Dual-Pivot Quicksort

- **Observation in practice:**
  Yaroslavskiy's Quicksort (YQS) **faster** than *classic Quicksort (CQS)* ... **why?**

$\rightsquigarrow$ We did a mathematical analysis of YQS.

- **Traditional** cost measures do **not** explain observation!

	CQS	YQS	Relative
Running Time	(from various experiments)		$-10 \pm 2\%$
Comparisons	2	1.9	-5%
Swaps	$0.\overline{3}$	0.6	$+80\%$
Bytecode Instructions	18	21.7	$+20.6\%$
MMIX oops υ	11	13.1	$+19.1\%$
MMIX mems μ	$2.\overline{6}$	2.8	$+5\%$
Scanned Elements	2	1.6	-20%
Branch Mispredictions	0.57	0.58	$+2\%$

$\cdot\, n \ln n + O(n)$, *average case results*

Only plausible explanation for running time: 20% less memory transfers in YQS.

Track Record of Dual-Pivot Quicksort

- **Observation in practice:**
  Yaroslavskiy's Quicksort (YQS) **faster** than *classic Quicksort (CQS)* … **why?**

$\rightsquigarrow$ We did a mathematical analysis of YQS.

- **Traditional** cost measures do **not** explain observation!

	CQS	YQS	Relative
Running Time	(from various experiments)		−10±2%
Comparisons	2	1.9	−5%
Swaps	$0.\overline{3}$	0.6	+80%
Bytecode Instructions	18	21.7	+20.6%
MMIX oops υ	11	13.1	+19.1%
MMIX mems μ	$2.\overline{6}$	2.8	+5%
Scanned Elements	2	1.6	−20%
Branch Mispredictions	0.57	0.58	+2%

$\cdot n \ln n + O(n)$, *average case results*

Only plausible explanation for running time: 20% less memory transfers in YQS.

Track Record of Dual-Pivot Quicksort

- **Observation in practice:**
  Yaroslavskiy's Quicksort (YQS) **faster** than *classic Quicksort (CQS)* . . . **why?**

⇝ We did a mathematical analysis of YQS.

- **Traditional** cost measures do **not** explain observation!

	CQS	YQS	Relative
Running Time	(from various experiments)		$-10 \pm 2\%$
Comparisons	2	1.9	-5%
Swaps	$0.\overline{3}$	0.6	$+80\%$
Bytecode Instructions	18	21.7	$+20.6\%$
MMIX oops υ	11	13.1	$+19.1\%$
MMIX mems μ	$2.\overline{6}$	2.8	$+5\%$
Scanned Elements	2	1.6	-20%
Branch Mispredictions	0.57	0.58	$+2\%$

$\cdot\, n \ln n + O(n)$, *average case results*

Only plausible explanation for running time: 20% less memory transfers in YQS.

Track Record of Dual-Pivot Quicksort

- **Observation in practice:**
  Yaroslavskiy's Quicksort (YQS) **faster** than *classic Quicksort (CQS)* … **why?**

$\rightsquigarrow$ We did a mathematical analysis of YQS.

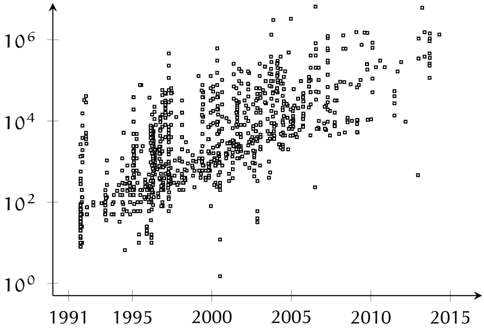
- **Traditional** cost measures do **not** explain observation!

	CQS	YQS	Relative
Running Time	(from various experiments)		$-10\pm2\%$
Comparisons	2	1.9	-5%
Swaps	$0.\overline{3}$	0.6	$+80\%$
Bytecode Instructions	18	21.7	$+20.6\%$
MMIX oops υ	11	13.1	$+19.1\%$

What happened?!

Only plausible explanation for running time: 20% less memory transfers in YQS.

The "Memory Wall"



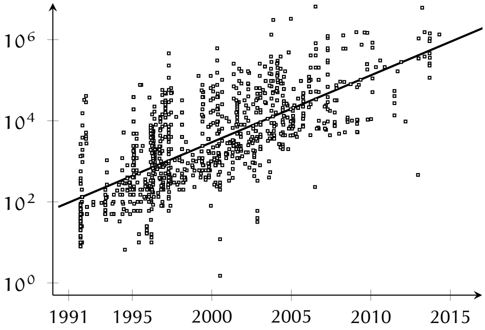
STREAM benchmark data with linear regressions

www.cs.virginia.edu/stream/by_date/Balance.html

- 20 years since Bentley and McIlroy developed the classic Quicksort implementation

. . . this most likely changes the game for sorting!

The “Memory Wall”



CPU speed (MFLOPS)

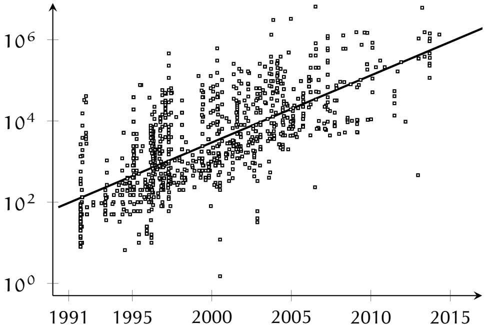
STREAM benchmark data with linear regressions
www.cs.virginia.edu/stream/by_date/Balance.html

- 20 years since Bentley and McIlroy developed the classic Quicksort implementation

$\rightsquigarrow$ relative cost of RAM accesses today 5 times as big

… *this most likely changes the game for sorting!*

The "Memory Wall"



Averaged annual growth rates:
46% CPU speed

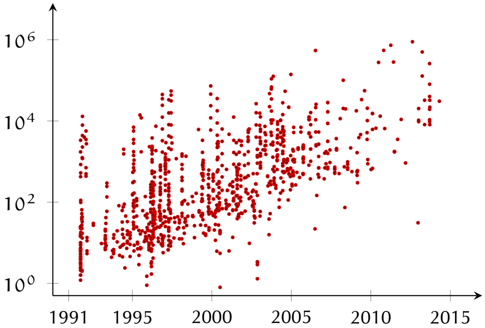
STREAM benchmark data with linear regressions
`www.cs.virginia.edu/stream/by_date/Balance.html`

- 20 years since Bentley and McIlroy developed the classic Quicksort implementation

$\rightsquigarrow$ relative cost of RAM accesses today 5 times as big

… this most likely changes the game for sorting!

The “Memory Wall”



CPU speed (MFLOPS)
RAM bandwidth (MWords/s)

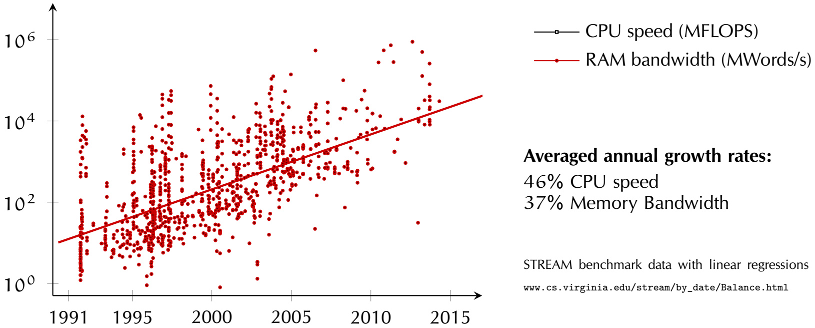
Averaged annual growth rates:
46% CPU speed

STREAM benchmark data with linear regressions
`www.cs.virginia.edu/stream/by_date/Balance.html`

- 20 years since Bentley and McIlroy developed the classic Quicksort implementation

. . . this most likely changes the game for sorting!

The "Memory Wall"



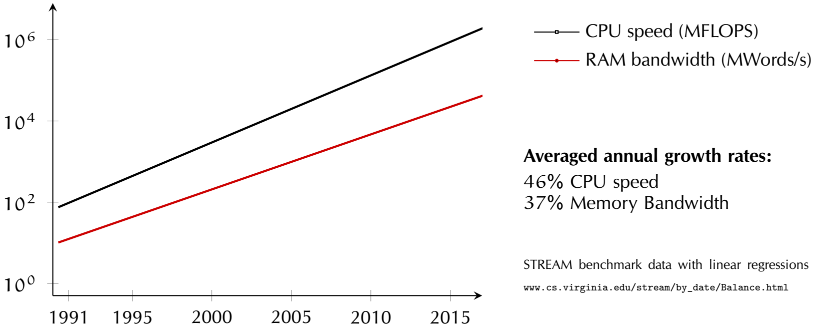
Averaged annual growth rates:
46% CPU speed
37% Memory Bandwidth

STREAM benchmark data with linear regressions
www.cs.virginia.edu/stream/by_date/Balance.html

- 20 years since Bentley and McIlroy developed the classic Quicksort implementation

... *this most likely changes the game for sorting!*

The “Memory Wall”



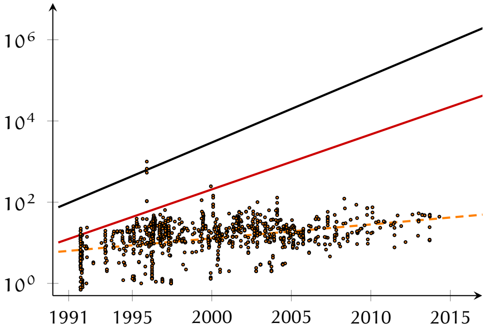
Averaged annual growth rates:
46% CPU speed
37% Memory Bandwidth

STREAM benchmark data with linear regressions
`www.cs.virginia.edu/stream/by_date/Balance.html`

- 20 years since Bentley and McIlroy developed the classic Quicksort implementation
- $\rightsquigarrow$ relative cost of RAM accesses today 5 times as big

… this most likely changes the game for sorting!

The “Memory Wall”



CPU speed (MFLOPS)
RAM bandwidth (MWords/s)
“imbalance”: $\frac{\text{speed}}{\text{bandwidth}}$

Averaged annual growth rates:

46% CPU speed
37% Memory Bandwidth
$\rightsquigarrow$ 8.2% Imbalance

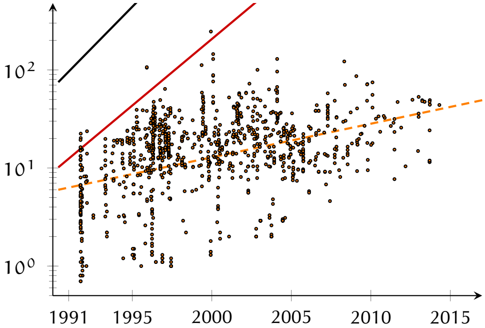
STREAM benchmark data with linear regressions

`www.cs.virginia.edu/stream/by_date/Balance.html`

- 20 years since Bentley and McIlroy developed the classic Quicksort implementation

… *this most likely changes the game for sorting!*

The "Memory Wall"



CPU speed (MFLOPS)
RAM bandwidth (MWords/s)
"imbalance": $\frac{\text{speed}}{\text{bandwidth}}$

Averaged annual growth rates:

46% CPU speed
37% Memory Bandwidth
$\rightsquigarrow$ 8.2% Imbalance

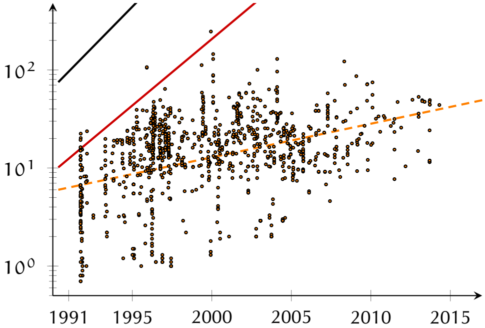
STREAM benchmark data with linear regressions
www.cs.virginia.edu/stream/by_date/Balance.html

- 20 years since Bentley and McIlroy developed the classic Quicksort implementation

$\rightsquigarrow$ relative cost of RAM accesses today 5 times as big

. . . this most likely changes the game for sorting!

The "Memory Wall"



Averaged annual growth rates:

46% CPU speed
37% Memory Bandwidth
$\rightsquigarrow$ 8.2% Imbalance

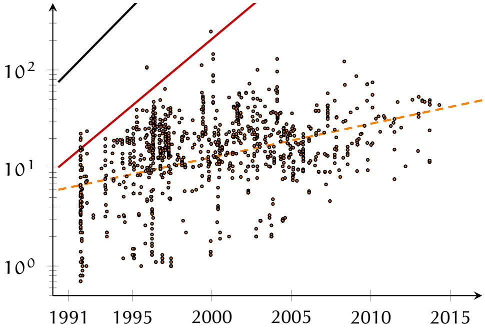
STREAM benchmark data with linear regressions
www.cs.virginia.edu/stream/by_date/Balance.html

- 20 years since Bentley and McIlroy developed the classic Quicksort implementation

$\rightsquigarrow$ relative cost of RAM accesses today 5 **times as big**

… *this most likely changes the game for sorting!*

The "Memory Wall"



Averaged annual growth rates:
46% CPU speed
37% Memory Bandwidth
$\rightsquigarrow$ 8.2% Imbalance

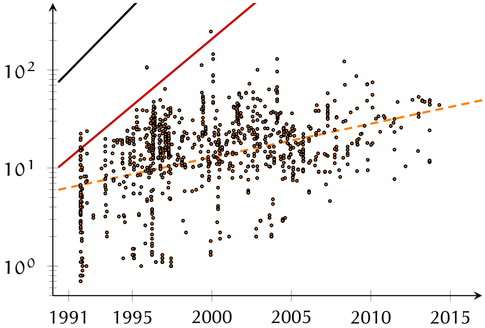
STREAM benchmark data with linear regressions
`www.cs.virginia.edu/stream/by_date/Balance.html`

- 20 years since Bentley and McIlroy developed the classic Quicksort implementation

$\rightsquigarrow$ relative cost of RAM accesses today **5 times as big**

... this most likely changes the game for sorting!

The "Memory Wall"



CPU speed (MFLOPS)
RAM bandwidth (MWords/s)
"imbalance": $\frac{\text{speed}}{\text{bandwidth}}$

Averaged annual growth rates:

46% CPU speed
37% Memory Bandwidth
$\rightsquigarrow$ 8.2% Imbalance

STREAM benchmark data with linear regressions

`www.cs.virginia.edu/stream/by_date/Balance.html`

- 20 years since Bentley and McIlroy developed the classic Quicksort implementation

$\rightsquigarrow$ relative cost of RAM accesses today **5 times as big**

… *this most likely changes the game for sorting!*

Analyzing Memory Transfers

*Need an **abstract** and **simple** cost model to approximate **memory transfer**.*

Analyzing Memory Transfers

*Need an **abstract** and **simple** cost model to approximate **memory transfer**.*

- abstract ⇝ machine-independent results
- simple ⇝ easy to analyze

- should **only** count memory accesses that are probably **not cached**

⇝ neither swaps nor comparisons are suitable measures!

My proposal: number of ***"scanned elements"***

- Machine model: Access to array only through iterators
- Iterators can
  - head left or right (one-directional)
  - advance to next position
  - read and write current position
- Cost: number of advances

. . . let's compare scanned elements for our Quicksorts!

Analyzing Memory Transfers

*Need an **abstract** and **simple** cost model to approximate **memory transfer**.*

- abstract $\rightsquigarrow$ machine-independent results
- simple $\rightsquigarrow$ easy to analyze

- should **only** count memory accesses that are probably **not cached**

$\rightsquigarrow$ neither swaps nor comparisons are suitable measures!

My proposal: number of ***“scanned elements”***

- Machine model: Access to array only through iterators
- Iterators can
  - read left or right (one-directionally)
  - advance to next position
  - read and write current position
- Cost: number of advances

… let’s compare scanned elements for our Quicksorts!

Analyzing Memory Transfers

*Need an **abstract** and **simple** cost model to approximate **memory transfer**.*

- abstract $\rightsquigarrow$ machine-independent results
- simple $\rightsquigarrow$ easy to analyze
- should **only** count memory accesses that are probably **not cached**

$\rightsquigarrow$ neither swaps nor comparisons are suitable measures!

My proposal: number of ***"scanned elements"***

- Machine model: Access to array only through iterators
- Iterators can
  - head left or right (one-directionally)
  - advance to next position
  - read and write current position

- Cost: number of advances

... let's compare scanned elements for our Quicksorts!

Analyzing Memory Transfers

*Need an **abstract** and **simple** cost model to approximate **memory transfer**.*

- abstract ⇝ machine-independent results
- simple ⇝ easy to analyze

- should **only** count memory accesses that are probably **not cached**

⇝ neither swaps nor comparisons are suitable measures!

My proposal: number of ***"scanned elements"***

- Machine model: Access to array only through **iterators**
- Iterators can
  - head left or right (one-directional!)
  - advance to next position
  - read and write current position
- Cost: number of advances

. . . let's compare scanned elements for our Quicksorts!

Analyzing Memory Transfers

*Need an **abstract** and **simple** cost model to approximate **memory transfer**.*

- abstract $\rightsquigarrow$ machine-independent results
- simple $\rightsquigarrow$ easy to analyze

- should **only** count memory accesses that are probably **not cached**

$\rightsquigarrow$ neither swaps nor comparisons are suitable measures!

My proposal: number of ***"scanned elements"***

- Machine model: Access to array only through **iterators**
- Iterators can
  - head left or right (one-directional!)
  - **advance** to next position
  - **read** and **write** current position
- Cost: number of advances

... let's compare scanned elements for our Quicksorts!

Analyzing Memory Transfers

*Need an **abstract** and **simple** cost model to approximate **memory transfer**.*

- abstract $\rightsquigarrow$ machine-independent results
- simple $\rightsquigarrow$ easy to analyze

- should **only** count memory accesses that are probably **not cached**

$\rightsquigarrow$ neither swaps nor comparisons are suitable measures!

My proposal: number of ***"scanned elements"***

- Machine model: Access to array only through **iterators**
- Iterators can
  - head left or right (one-directional!)
  - **advance** to next position
  - **read** and **write** current position
- Cost: number of advances

... let's compare scanned elements for our Quicksorts!

Analyzing Memory Transfers

*Need an **abstract** and **simple** cost model to approximate **memory transfer**.*

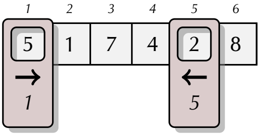
- abstract ⇝ machine-independent results
- simple ⇝ easy to analyze

- should **only** count memory accesses that are probably **not cached**

⇝ neither swaps nor comparisons are suitable measures!

My proposal: number of ***"scanned elements"***

- Machine model: Access to array only through **iterators**
- Iterators can
  - head left or right (one-directional!)
  - **advance** to next position
  - **read** and **write** current position



- Cost: number of advances

… *let's compare scanned elements for our Quicksorts!*

Analyzing Memory Transfers

*Need an **abstract** and **simple** cost model to approximate **memory transfer**.*

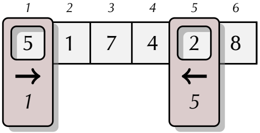
- abstract ⇝ machine-independent results
- simple ⇝ easy to analyze

- should **only** count memory accesses that are probably **not cached**

⇝ neither swaps nor comparisons are suitable measures!

My proposal: number of ***"scanned elements"***

- Machine model: Access to array only through **iterators**
- Iterators can
  - head left or right (one-directional!)
  - **advance** to next position
  - **read** and **write** current position
- Cost: number of advances



… let's compare scanned elements for our Quicksorts!

Analyzing Memory Transfers

*Need an **abstract** and **simple** cost model to approximate **memory transfer**.*

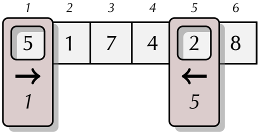
- abstract $\rightsquigarrow$ machine-independent results
- simple $\rightsquigarrow$ easy to analyze

- should **only** count memory accesses that are probably **not cached**

$\rightsquigarrow$ neither swaps nor comparisons are suitable measures!

My proposal: number of ***"scanned elements"***

- Machine model: Access to array only through **iterators**
- Iterators can
  - head left or right (one-directional!)
  - **advance** to next position
  - **read** and **write** current position



- Cost: number of advances

. . . let's compare scanned elements for our Quicksorts!

Scanned Elements in CQS and YQS

How many scanned elements (SE) do we need for partitioning?

Classic Quicksort

- array scanned exactly once

$\rightsquigarrow$ n scanned elements

Yaroslavskiy's Quicksort

How does this translate to sorting costs?

Classic Quicksort

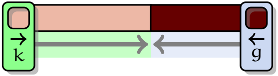
$\rightsquigarrow$ $2n \ln n$ SE overall

Yaroslavskiy's Quicksort

Scanned Elements in CQS and YQS

How many scanned elements (SE) do we need for partitioning?

Classic Quicksort



- array scanned exactly once

⇝ n scanned elements

Yaroslavskiy's Quicksort

⇝ $1.\overline{3}n$ SE on average

How does this translate to sorting costs?

Classic Quicksort

⇝ $2\,n \ln n$ SE overall

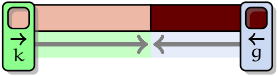
Yaroslavskiy's Quicksort

⇝ $1.6n \ln n$ SE overall

Scanned Elements in CQS and YQS

How many scanned elements (SE) do we need for partitioning?

Classic Quicksort



- array scanned exactly once

$\rightsquigarrow$ n scanned elements

Yaroslavskiy's Quicksort

$\rightsquigarrow$ $1.\overline{3}n$ SE on average

How does this translate to sorting costs?

Classic Quicksort

$\rightsquigarrow$ $2\,n \ln n$ SE overall

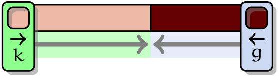
Yaroslavskiy's Quicksort

$\rightsquigarrow$ $1.\overline{6}n \ln n$ SE overall

Scanned Elements in CQS and YQS

How many scanned elements (SE) do we need for partitioning?

Classic Quicksort



- array scanned exactly once

⇝ n scanned elements

Yaroslavskiy's Quicksort

⇝ $1.3n$ SE on average

How does this translate to sorting costs?

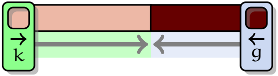
Classic Quicksort

Yaroslavskiy's Quicksort

Scanned Elements in CQS and YQS

How many scanned elements (SE) do we need for partitioning?

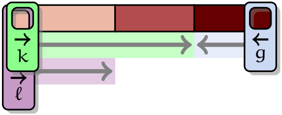
Classic Quicksort



- array scanned exactly once

⇝ n scanned elements

Yaroslavskiy's Quicksort



⇝ $1.\overline{3}n$ SE on average
worse than CQS!

How does this translate to sorting costs?

Classic Quicksort

⇝ $2\, n \ln n$ SE overall

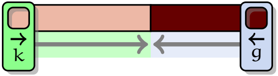
Yaroslavskiy's Quicksort

⇝ $1.6\, n \ln n$ SE total

Scanned Elements in CQS and YQS

How many scanned elements (SE) do we need for partitioning?

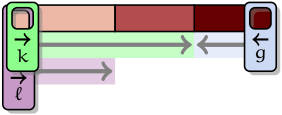
Classic Quicksort



- array scanned exactly once

$\rightsquigarrow$ n scanned elements

Yaroslavskiy's Quicksort



$\rightsquigarrow$ $1.\overline{3}n$ SE on average

worse than CQS!

How does this translate to sorting costs?

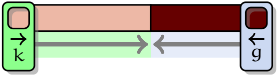
Classic Quicksort

Yaroslavskiy's Quicksort

Scanned Elements in CQS and YQS

How many scanned elements (SE) do we need for partitioning?

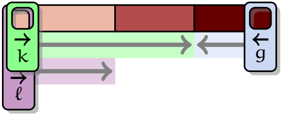
Classic Quicksort



- array scanned exactly once

⇝ n scanned elements

Yaroslavskiy's Quicksort



⇝ $1.\overline{3}n$ SE on average
worse than CQS!

How does this translate to sorting costs?

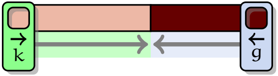
Classic Quicksort

Yaroslavskiy's Quicksort

Scanned Elements in CQS and YQS

How many scanned elements (SE) do we need for partitioning?

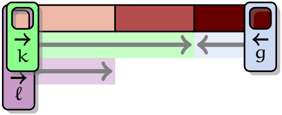
Classic Quicksort



- array scanned exactly once

$\rightsquigarrow$ n scanned elements

Yaroslavskiy's Quicksort



$\rightsquigarrow$ $1.\overline{3}n$ SE on average
worse than CQS!

How does this translate to sorting costs?

Classic Quicksort

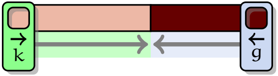
$\rightsquigarrow$ $2\,n \ln n$ SE overall

Yaroslavskiy's Quicksort

Scanned Elements in CQS and YQS

How many scanned elements (SE) do we need for partitioning?

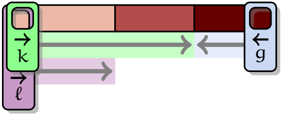
Classic Quicksort



- array scanned exactly once

⇝ n scanned elements

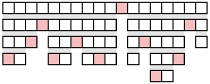
Yaroslavskiy's Quicksort



⇝ $1.\overline{3}n$ SE on average
worse than CQS!

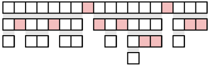
How does this translate to sorting costs?

Classic Quicksort



⇝ **2** $n \ln n$ SE overall

Yaroslavskiy's Quicksort

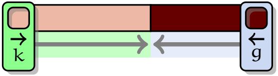


⇝ **1.6** $n \ln n$ SE overall

Scanned Elements in CQS and YQS

How many scanned elements (SE) do we need for partitioning?

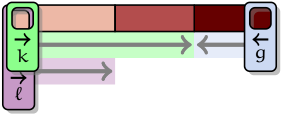
Classic Quicksort



- array scanned exactly once

$\rightsquigarrow$ n scanned elements

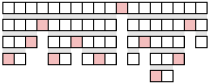
Yaroslavskiy's Quicksort



$\rightsquigarrow$ $1.\overline{3}n$ SE on average
worse than CQS!

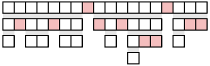
How does this translate to sorting costs?

Classic Quicksort



$\rightsquigarrow$ **2** $n \ln n$ SE overall

Yaroslavskiy's Quicksort

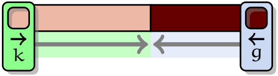


$\rightsquigarrow$ **1.6** $n \ln n$ SE overall

Scanned Elements in CQS and YQS

How many scanned elements (SE) do we need for partitioning?

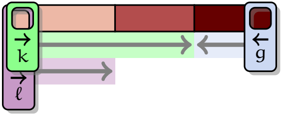
Classic Quicksort



- array scanned exactly once

⇝ n scanned elements

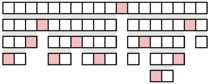
Yaroslavskiy's Quicksort



⇝ $1.\overline{3}n$ SE on average
worse than CQS!

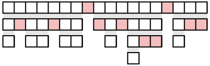
How does this translate to sorting costs?

Classic Quicksort



⇝ **2** $n \ln n$ SE overall

Yaroslavskiy's Quicksort



⇝ **1.6** $n \ln n$ SE overall

Track Record of Dual-Pivot Quicksort

- **Observation in practice:**
  Yaroslavskiy's Quicksort (YQS) **faster** than *classic Quicksort (CQS)* … **why?**

$\rightsquigarrow$ We did a mathematical analysis of YQS.

- **Traditional** cost measures do **not** explain observation!

	CQS	YQS	Relative
Running Time	(from various experiments)		−10±2%
Comparisons	2	1.9	−5%
Swaps	$0.\overline{3}$	0.6	+80%
Bytecode Instructions	18	21.7	+20.6%
MMIX oops υ	11	13.1	+19.1%
MMIX mems μ	$2.\overline{6}$	2.8	+5%
Scanned Elements	2	1.6	−20%
Branch Mispredictions	0.57	0.58	+2%

$\cdot n \ln n + O(n)$, *average case results*

Only plausible explanation for running time: 20% less memory transfers in YQS.

Track Record of Dual-Pivot Quicksort

- **Observation in practice:**
  Yaroslavskiy's Quicksort (YQS) **faster** than *classic Quicksort (CQS)* … **why?**

$\rightsquigarrow$ We did a mathematical analysis of YQS.

- **Traditional** cost measures do **not** explain observation!

	CQS	YQS	Relative
Running Time	(from various experiments)		−10±2%
Comparisons	2	1.9	−5%
Swaps	$0.\overline{3}$	0.6	+80%
Bytecode Instructions	18	21.7	+20.6%
MMIX oops υ	11	13.1	+19.1%
MMIX mems μ	$2.\overline{6}$	2.8	+5%
Scanned Elements	2	1.6	−20%
Branch Mispredictions	0.57	0.58	+2%

$\cdot n \ln n + O(n)$, *average case results*

Only plausible explanation for running time: 20% less memory transfers in YQS.

Track Record of Dual-Pivot Quicksort

- **Observation in practice:**
  Yaroslavskiy's Quicksort (YQS) **faster** than *classic Quicksort (CQS)* … **why?**

$\rightsquigarrow$ We did a mathematical analysis of YQS.

- **Traditional** cost measures do **not** explain observation!

	CQS	YQS	Relative
Running Time	(from various experiments)		$-10 \pm 2\%$
Comparisons	2	1.9	-5%
Swaps	$0.\overline{3}$	0.6	$+80\%$
Bytecode Instructions	18	21.7	$+20.6\%$
MMIX oops υ	11	13.1	$+19.1\%$
MMIX mems μ	$2.\overline{6}$	2.8	$+5\%$
Scanned Elements	2	1.6	-20%
Branch Mispredictions	0.57	0.58	$+2\%$

$\cdot n \ln n + O(n)$, *average case results*

Only plausible explanation for running time: 20% less memory transfers in YQS.

Conclusion

1. Dual-Pivot Quicksort most likely faster because of fewer **memory references**.

2. The "memory wall" calls for new view on classic algorithms.
   How about others?

3. Don't stop looking for theoretical explanations
   but do question models and assumptions!

Conclusion

1. Dual-Pivot Quicksort most likely faster because of fewer **memory references**.

2. The "memory wall" calls for new view on classic algorithms.
   How about others?

3. Don't stop looking for theoretical explanations,
   but do **question models** and assumptions!

Conclusion

1. Dual-Pivot Quicksort most likely faster because of fewer **memory references**.
2. The "memory wall" calls for new view on classic algorithms.
   How about others?
3. Don't stop looking for theoretical explanations,
   but do **question models** and assumptions!

Conclusion

1. Dual-Pivot Quicksort most likely faster because of fewer **memory references**.
2. The “memory wall” calls for new view on classic algorithms.
   How about others?
3. Don’t stop looking for theoretical explanations,
   but do **question models** and assumptions!