Why is Dual-Pivot Quicksort Fast?

Sebastian Wild
wild@cs.uni-kl.de

29 September 2015
Theorietage 2015 Speyer
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**: $\Theta(n \log n)$ worst case by truncating recursion

⇝ Basic algorithm settled since 1961; latest tweaks from 1990’s.

Since then: Almost identical in all programming libraries!
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**: $O(n \log n)$ worst case by truncating recursion

⇝ Basic algorithm settled since 1961; latest tweaks from 1990’s.

**Since then**: Almost identical in all programming libraries!
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
Algorithm (Conceptual View)

1. Choose **two pivots** $P \leq Q$
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:

   ![P Q]

4. Sort subarrays recursively.

*How to implement efficiently on arrays?*
Algorithm (Conceptual View)

1. Choose two pivots $P \leq Q$
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 efficiently on arrays?
Algorithm (Conceptual View)

1. Choose two pivots \( P \leq Q \)
2. For each element \( x \), determine its class
   - \textbf{small} for \( x < P \)
   - \textbf{medium} for \( P < x < Q \)
   - \textbf{large} for \( Q < x \)
   by comparing \( x \) to pivots \( P \) and \( Q \)
3. Arrange elements according to classes:
   \[
   \begin{array}{c}
   \text{small} \quad P \quad \text{medium} \quad Q \quad \text{large}
   \end{array}
   \]
4. Sort subarrays recursively.

How to implement efficiently on arrays?
Algorithm (Conceptual View)

1. Choose **two pivots** $P \leq Q$

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:

   ![Diagram showing elements arranged between $P$ and $Q$]

4. Sort subarrays recursively.

*How to implement efficiently on arrays?*
Yaroslavskiy’s Algorithm

\[
\begin{array}{c}
\text{Invariant:} \\
\begin{array}{c|c|c}
P & \text{ } & Q \\
\hline
3 & 5 & 1 \\
7 & 4 & 2 \\
8 & 6 & \\
\end{array}
\end{array}
\]
Yaroslavskiy’s Algorithm

Invariant:  

```plaintext
P | 3 5 1 7 4 2 8 6
Q
```
Yaroslavskiy’s Algorithm

Invariant: 

\[ \begin{array}{c|c|c}
P & & Q \\
\hline
3 & 5 & 1 \ 7 \ 4 \ 2 \ 8 \ 6 \\
\end{array} \]
Yaroslavskiy’s Algorithm

### Dual-Pivot Quicksort

**Invariant:**

\[
\begin{array}{c}
\text{P}\quad \text{P} \leq o \leq \text{Q}
\end{array}
\]
Yaroslavskiy’s Algorithm

Invariant:

Invariant:

Invariant:
Yaroslavskiy’s Algorithm

Invariant: $P \leq \sigma \leq Q$
Yaroslavskiy’s Algorithm

Invariant:

Dual-Pivot Quicksort

Sebastian Wild

2015-03-24
Yaroslavskiy’s Algorithm

Invariant: $P < P \leq Q$
Yaroslavskiy’s Algorithm

Invariant: $P \leq P \preceq Q$
Yaroslavskiy’s Algorithm

Invariant: $P < P \leq \circ \leq Q$
Yaroslavskiy’s Algorithm

![Diagram of Yaroslavskiy’s Algorithm]

**Invariant:**

\[
P < P \quad P \leq \circ \leq Q \quad Q
\]
Yaroslavskiy’s Algorithm

Invariant:

\[ P < P \leq P \leq Q \leq g \geq Q \]
Yaroslavskiy’s Algorithm

Invariant:

- \( P \) ≤ \( P \leq g \leq Q \)
- \( Q \) ≥ Q

Diagram:

- For \( < P \):
  - If yes, swap \( \ell \)
  - If no, skip

- For \( > Q \):
  - If yes, swap \( k \)
  - If no, swap \( g \)

Example:

- Array: 3, 1, 5, 7, 2, 8, 6
- P = 3, Q = 4

Flow:

- \( P \leq g \leq Q \)
Yaroslavskiy’s Algorithm

Invariant:

P < P  P ≤ o ≤ Q  ≥ Q  Q
Yaroslavskiy’s Algorithm

Invariant: $P < P \leq g \leq Q \geq Q$. 

Dual-Pivot Quicksort
2015-03-24
Yaroslavskiy’s Algorithm

Invariant:

\[
\begin{align*}
P & < P \\
P & \leq o \leq Q \\
\geq Q & Q
\end{align*}
\]
Yaroslavskiy’s Algorithm

Invariant: P < P ≤ o ≤ Q ≥ Q Q
Yaroslavskiy’s Algorithm

Invariant: 

Dual-Pivot Quicksort

2015-03-24
Yaroslavskiy’s Algorithm

Dual-Pivot Quicksort

Sebastian Wild
Yaroslavskiy’s Algorithm

Yaroslavskiy’s Algorithm is a variant of the quicksort algorithm that uses two pivot elements to partition the input array. The algorithm works by selecting two pivot elements, P and Q, and then partitioning the array based on these elements. The pivot elements are chosen such that all elements less than P are to the left, all elements equal to P are in the middle, and all elements greater than Q are to the right.

The algorithm recursively sorts the subarrays to the left and right of the pivot elements, resulting in a sorted array.

The diagram shows the decision tree for Yaroslavskiy’s Algorithm. The tree is divided into two parts: one for elements less than P and one for elements greater than Q.

The decision tree for elements less than P is as follows:
- If the element is less than P, swap with the leftmost element.
- If the element is greater than or equal to P, continue recursively.

The decision tree for elements greater than Q is as follows:
- If the element is greater than Q, swap with the rightmost element.
- If the element is less than or equal to Q, continue recursively.

The algorithm uses a similar process for elements equal to P and elements less than or equal to Q.

The diagram also includes a set of numbers (2, 1, 3, 5, 4, 6, 8, 7) which represent the input array. The algorithm partitions the array based on these numbers.
Yaroslavsky’s Algorithm

\[
\begin{align*}
\text{\textless{} P?} & \quad \text{\textgreater{} Q?} \\
\checkmark & \quad \checkmark \\
\text{swap } \ell & \quad \text{skip} \\
\text{\textless{} Q?} & \quad \text{\textgreater{} Q?} \\
\checkmark & \quad \checkmark \\
\text{skip} & \quad \text{swap } g \\
\text{\textless{} P?} & \quad \text{\textless{} P?} \\
\checkmark & \quad \checkmark \\
\text{skip} & \quad \text{swap } g \\
\text{\textgreater{} \checkmark} & \quad \text{\textgreater{} \checkmark} \\
\text{swap } \ell & \quad \text{swap } k
\end{align*}
\]
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!

![Graph showing normalized Java runtimes (in ms). Average and standard deviation of 1000 random permutations per size.]

Normalized Java runtimes (in ms). Average and standard deviation of 1000 random permutations per size.
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. -□-!
Why switch to new, unknown algorithm? Because it is faster!

No theoretical explanation for running time known in 2009!
Why switch to new, unknown algorithm? Because it is faster!

No theoretical explanation for running time known in 2009!

Only lucky experiments?
Running Time Experiments

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

Why did noone come up with this earlier?

No theoretical explanation for running time known in 2009!

Only lucky experiments?

Why did noone come up with this earlier?
**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!

<table>
<thead>
<tr>
<th>CQS</th>
<th>YQS</th>
<th>Relative</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Running Time</strong> (from various experiments)</td>
<td></td>
<td><strong>− 10 ± 2%</strong></td>
</tr>
<tr>
<td>Comparisons</td>
<td>2</td>
<td>1.9</td>
</tr>
<tr>
<td>Swaps</td>
<td>0.3</td>
<td>0.6</td>
</tr>
<tr>
<td>Bytecode Instructions</td>
<td>18</td>
<td>21.7</td>
</tr>
<tr>
<td>MMIX oops v</td>
<td>11</td>
<td>13.1</td>
</tr>
<tr>
<td>MMIX mems µ</td>
<td>2.6</td>
<td>2.8</td>
</tr>
<tr>
<td>Scanned Elements</td>
<td>2</td>
<td>1.6</td>
</tr>
<tr>
<td>Branch Mispredictions</td>
<td>0.57</td>
<td>0.58</td>
</tr>
</tbody>
</table>

$\approx n \ln n + O(n)$: average case results

---

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

<table>
<thead>
<tr>
<th></th>
<th>CQS</th>
<th>YQS</th>
<th>Relative</th>
</tr>
</thead>
<tbody>
<tr>
<td>Running Time (from various experiments)</td>
<td></td>
<td></td>
<td>−10±2%</td>
</tr>
<tr>
<td>Comparisons</td>
<td>2</td>
<td>1.9</td>
<td>−5%</td>
</tr>
<tr>
<td>Swaps</td>
<td>0.3</td>
<td>0.6</td>
<td>+80%</td>
</tr>
<tr>
<td>Bytecode Instructions</td>
<td>18</td>
<td>21.7</td>
<td>+20.6%</td>
</tr>
<tr>
<td>MMIX oops ψ</td>
<td>11</td>
<td>13.1</td>
<td>+19.1%</td>
</tr>
<tr>
<td>MMIX mems μ</td>
<td>2.6</td>
<td>2.8</td>
<td>+5%</td>
</tr>
<tr>
<td>Scanned Elements</td>
<td>2</td>
<td>1.6</td>
<td>−20%</td>
</tr>
<tr>
<td>Branch Mispredictions</td>
<td>0.57</td>
<td>0.58</td>
<td>+2%</td>
</tr>
</tbody>
</table>

\[ n \ln n + O(n) \] average case results

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

<table>
<thead>
<tr>
<th></th>
<th>CQS</th>
<th>YQS</th>
<th>Relative</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Running Time</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(from various experiments)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Comparisons</strong></td>
<td>2</td>
<td>1.9</td>
<td>−5%</td>
</tr>
<tr>
<td><strong>Swaps</strong></td>
<td>0.3</td>
<td>0.6</td>
<td>+80%</td>
</tr>
<tr>
<td><strong>Bytecode Instructions</strong></td>
<td>18</td>
<td>21.7</td>
<td>+20.6%</td>
</tr>
<tr>
<td><strong>MMIX oops υ</strong></td>
<td>11</td>
<td>13.1</td>
<td>+19.1%</td>
</tr>
<tr>
<td><strong>MMIX mems μ</strong></td>
<td>2.6</td>
<td>2.8</td>
<td>+5%</td>
</tr>
<tr>
<td><strong>Scanned Elements</strong></td>
<td>2</td>
<td>1.6</td>
<td>−20%</td>
</tr>
<tr>
<td><strong>Branch Mispredictions</strong></td>
<td>0.57</td>
<td>0.58</td>
<td>+2%</td>
</tr>
</tbody>
</table>

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

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

<table>
<thead>
<tr>
<th></th>
<th>CQS</th>
<th>YQS</th>
<th>Relative</th>
</tr>
</thead>
<tbody>
<tr>
<td>Running Time (from various experiments)</td>
<td></td>
<td>-10±2%</td>
<td></td>
</tr>
<tr>
<td>Comparisons</td>
<td>2</td>
<td>1.9</td>
<td>-5%</td>
</tr>
<tr>
<td>Swaps</td>
<td>0.3</td>
<td>0.6</td>
<td>+80%</td>
</tr>
<tr>
<td>Bytecode Instructions</td>
<td>18</td>
<td>21.7</td>
<td>+20.6%</td>
</tr>
<tr>
<td>MMIX oops μ</td>
<td>11</td>
<td>13.1</td>
<td>+19.1%</td>
</tr>
<tr>
<td>MMIX mems μ</td>
<td>2.6</td>
<td>2.8</td>
<td>+5%</td>
</tr>
<tr>
<td>Scanned Elements</td>
<td>2</td>
<td>1.6</td>
<td>-20%</td>
</tr>
<tr>
<td>Branch Mispredictions</td>
<td>0.57</td>
<td>0.58</td>
<td>+2%</td>
</tr>
</tbody>
</table>

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

Only plausible explanation for running time: 20% less memory transfers in YQS.
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
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Running Time (from various experiments)</td>
<td></td>
<td></td>
<td>-10±2%</td>
</tr>
<tr>
<td>Comparisons</td>
<td>2</td>
<td>1.9</td>
<td>-5%</td>
</tr>
<tr>
<td>Swaps</td>
<td>0.3</td>
<td>0.6</td>
<td>+80%</td>
</tr>
<tr>
<td>Bytecode Instructions</td>
<td>18</td>
<td>21.7</td>
<td>+20.6%</td>
</tr>
<tr>
<td>MMIX oops υ</td>
<td>11</td>
<td>13.1</td>
<td>+19.1%</td>
</tr>
<tr>
<td>MMIX mems μ</td>
<td>2.6</td>
<td>2.8</td>
<td>+5%</td>
</tr>
<tr>
<td>Scanned Elements</td>
<td>2</td>
<td>1.6</td>
<td>-20%</td>
</tr>
<tr>
<td>Branch Mispredictions</td>
<td>0.57</td>
<td>0.58</td>
<td>+2%</td>
</tr>
</tbody>
</table>

\[ n \ln n + O(n), \text{ average case results} \]

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

<table>
<thead>
<tr>
<th></th>
<th>CQS</th>
<th>YQS</th>
<th>Relative</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Running Time</strong> (from various experiments)</td>
<td></td>
<td></td>
<td>−10±2%</td>
</tr>
<tr>
<td><strong>Comparisons</strong></td>
<td>2</td>
<td>1.9</td>
<td>−5%</td>
</tr>
<tr>
<td><strong>Swaps</strong></td>
<td>0.3</td>
<td>0.6</td>
<td>+80%</td>
</tr>
<tr>
<td><strong>Bytecode Instructions</strong></td>
<td>18</td>
<td>21.7</td>
<td>+20.6%</td>
</tr>
<tr>
<td><strong>MMIX oops</strong></td>
<td>11</td>
<td>13.1</td>
<td>+19.1%</td>
</tr>
</tbody>
</table>

What happened?!

*Only plausible explanation for running time: 20% less memory transfers in YQS.*
The “Memory Wall”

- 20 years since Bentley and McIlroy developed the classic Quicksort implementation
- Relative cost of RAM accesses today 5 times as big
- ... this most likely changes the game for sorting!
The “Memory Wall”

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

… this most likely changes the game for sorting!

STREAM benchmark data with linear regressions
www.cs.virginia.edu/stream/by_date/Balance.html
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
- relative cost of RAM accesses today 5 times as big

… this most likely changes the game for sorting!
20 years since Bentley and McIlroy developed the classic Quicksort implementation.

Relative cost of RAM accesses today 5 times as big.

…this most likely changes the game for sorting!
The “Memory Wall”

20 years since Bentley and McIlroy developed the classic Quicksort implementation

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

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
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
- the relative cost of RAM accesses today is times as big

… this most likely changes the game for sorting!
The “Memory Wall”

- 20 years since Bentley and McIlroy developed the classic Quicksort implementation
  - relative cost of RAM accesses today 5 times as big
  
  … this most likely changes the game for sorting!

Averaged annual growth rates:
46% CPU speed
37% Memory Bandwidth
≈ 8.2% Imbalance

STREAM benchmark data with linear regressions
www.cs.virginia.edu/stream/by_date/Balance.html
The “Memory Wall”

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

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
\(~\sim 8.2\%\) 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
  
  ~~~ relative cost of RAM accesses today 5 times as big

  … this most likely changes the game for sorting!
The “Memory Wall”

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

→ relative cost of RAM accesses today 5 times as big

… this most likely changes the game for sorting!

Averaged annual growth rates:
46% CPU speed
37% Memory Bandwidth
≈ 8.2% Imbalance

STREAM benchmark data with linear regressions
www.cs.virginia.edu/stream/by_date/Balance.html
The “Memory Wall”

Averaged annual growth rates:
- 46% CPU speed
- 37% Memory Bandwidth
- ~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
- Relative cost of RAM accesses today \(5\) times as big

\(\ldots\) this most likely changes the game for sorting!
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 $\Rightarrow$ machine-independent results
- simple $\Rightarrow$ easy to analyze
- should **only** count memory accesses that are probably **not cached**
  $\Rightarrow$ 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 $\Rightarrow$ machine-independent results
- simple $\Rightarrow$ easy to analyze

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

$\Rightarrow$ 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

\[ \ldots \text{let’s compare scanned elements for our Quicksorts!} \]
Analyzing Memory Transfers

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

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

- should **only** count memory accesses that are probably **not cached**
  $\rightarrow$ 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 \(\Rightarrow\) machine-independent results
- simple \(\Rightarrow\) easy to analyze
- should only count memory accesses that are probably not cached
\(\Rightarrow\) 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 $\Rightarrow$ machine-independent results
- simple $\Rightarrow$ easy to analyze

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

\[ \ldots \text{let’s compare scanned elements for our Quicksorts!} \]
Analyzing Memory Transfers

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

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

- should only count memory accesses that are probably not cached
  $\rightarrow$ 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

$\ldots$ let’s compare scanned elements for our Quicksorts!
Analyzing Memory Transfers

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

- abstract $\rightarrow$ machine-independent results
- simple $\rightarrow$ 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 $\Rightarrow$ machine-independent results
- simple $\Rightarrow$ easy to analyze
- should only count memory accesses that are probably not cached
  $\Rightarrow$ 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!
How many scanned elements (SE) do we need for partitioning?

- **Classic Quicksort**: Array scanned exactly once on n scanned elements.
- **Yaroslavskiy’s Quicksort**: $1.3n$ SE on average.

How does this translate to sorting costs?

- **Classic Quicksort**: $2n \ln n$ SE overall.
- **Yaroslavskiy’s Quicksort**: $1.6n \ln n$ SE overall.
How many scanned elements (SE) do we need for partitioning?

**Classic Quicksort**

- array scanned exactly once
- $\approx n$ scanned elements

**Yaroslavskiy’s Quicksort**

- $1.3n$ SE on average

How does this translate to sorting costs?

**Classic Quicksort**

- $2n \ln n$ SE overall

**Yaroslavskiy’s Quicksort**

- $1.5n \ln n$ SE overall
**Scanned Elements in CQS and YQS**

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

**Classic Quicksort**
- Array scanned exactly once
- $\Rightarrow n$ scanned elements

**Yaroslavskiy’s Quicksort**
- $\Rightarrow 1.3n$ SE on average

*How does this translate to sorting costs?*

**Classic Quicksort**
- $2n \ln n$ SE overall

**Yaroslavskiy’s Quicksort**
- $1.6n \ln n$ SE overall
How many scanned elements (SE) do we need for partitioning?

**Classic Quicksort**

- array scanned exactly once
  - $\sim n$ scanned elements

**Yaroslavskiy’s Quicksort**

- $1.3n$ SE on average

How does this translate to sorting costs?

**Classic Quicksort**

- $2n \ln n$ 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
- $\leadsto n$ scanned elements

**Yaroslavskiy’s Quicksort**
- $\leadsto 1.3n$ SE on average worse than CQS!

How does this translate to sorting costs?

**Classic Quicksort**
- $2n \ln n$ SE overall

**Yaroslavskiy’s Quicksort**
- $1.6n \ln n$ SE overall
How many scanned elements (SE) do we need for partitioning?

**Classic Quicksort**
- array scanned exactly once
- $\sim n$ scanned elements

**Yaroslavskiy’s Quicksort**
- $\sim 1.3n$ SE on average
- worse than CQS!

How does this translate to sorting costs?

**Classic Quicksort**
- $2n \ln n$ SE overall

**Yaroslavskiy’s Quicksort**
- $1.6n \ln n$ SE overall
How many scanned elements (SE) do we need for partitioning?

**Classic Quicksort**

- array scanned exactly once
- \( \Rightarrow \) \( n \) scanned elements

**Yaroslavskiy’s Quicksort**

- \( \Rightarrow 1.3n \) SE on average
- worse than CQS!

How does this translate to sorting costs?

**Classic Quicksort**

\[ 2n \log n \text{ SE overall} \]

**Yaroslavskiy’s Quicksort**

\[ 1.6n \log n \text{ SE overall} \]
How many scanned elements (SE) do we need for partitioning?

**Classic Quicksort**
- Array scanned exactly once
- $\sim n$ scanned elements

**Yaroslavskiy’s Quicksort**
- $1.3n$ SE on average
- Worse than CQS!

How does this translate to sorting costs?

**Classic Quicksort**
- $2n \log n$ SE overall

**Yaroslavskiy’s Quicksort**
- $1.6n \log n$ SE overall
How many scanned elements (SE) do we need for partitioning?

**Classic Quicksort**
- Array scanned exactly once
- $\sim n$ scanned elements

**Yaroslavskiy’s Quicksort**
- $\sim 1.3n$ SE on average worse than CQS!

How does this translate to sorting costs?

**Classic Quicksort**
- $\sim 2n \ln n$ SE overall

**Yaroslavskiy’s Quicksort**
- $\sim 1.6n \ln n$ SE overall
How many scanned elements (SE) do we need for partitioning?

**Classic Quicksort**
- Array scanned exactly once
- \( \sim n \) scanned elements

**Yaroslavskiy’s Quicksort**
- \( \sim 1.3n \) SE on average
- worse than CQS!

How does this translate to sorting costs?

**Classic Quicksort**
- \( \sim 2n \ln n \) SE overall

**Yaroslavskiy’s Quicksort**
- \( \sim 1.6n \ln n \) SE overall
How many scanned elements (SE) do we need for partitioning?

**Classic Quicksort**
- Array scanned exactly once
- \( \sim n \) scanned elements

**Yaroslavskiy’s Quicksort**
- \( \sim 1.3n \) SE on average
- worse than CQS!

How does this translate to sorting costs?

**Classic Quicksort**
- \( \sim 2n \ln n \) SE overall

**Yaroslavskiy’s Quicksort**
- \( \sim 1.6n \ln n \) SE overall
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!

<table>
<thead>
<tr>
<th></th>
<th>CQS</th>
<th>YQS</th>
<th>Relative</th>
</tr>
</thead>
<tbody>
<tr>
<td>Running Time (from various experiments)</td>
<td></td>
<td></td>
<td>$-10 \pm 2%$</td>
</tr>
<tr>
<td>Comparisons</td>
<td>2</td>
<td>1.9</td>
<td>$-5%$</td>
</tr>
<tr>
<td>Swaps</td>
<td>0.3</td>
<td>0.6</td>
<td>+80%</td>
</tr>
<tr>
<td>Bytecode Instructions</td>
<td>18</td>
<td>21.7</td>
<td>+20.6%</td>
</tr>
<tr>
<td>MMIX oops $\nu$</td>
<td>11</td>
<td>13.1</td>
<td>+19.1%</td>
</tr>
<tr>
<td>MMIX mems $\mu$</td>
<td>2.6</td>
<td>2.8</td>
<td>+5%</td>
</tr>
<tr>
<td>Scanned Elements</td>
<td>2</td>
<td>1.6</td>
<td>$-20%$</td>
</tr>
<tr>
<td>Branch Mispredictions</td>
<td>0.57</td>
<td>0.58</td>
<td>+2%</td>
</tr>
</tbody>
</table>

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

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

<table>
<thead>
<tr>
<th></th>
<th>CQS</th>
<th>YQS</th>
<th>Relative</th>
</tr>
</thead>
<tbody>
<tr>
<td>Running Time</td>
<td>(from various experiments)</td>
<td></td>
<td>−10 ± 2%</td>
</tr>
<tr>
<td>Comparisons</td>
<td>2</td>
<td>1.9</td>
<td>−5%</td>
</tr>
<tr>
<td>Swaps</td>
<td>0.3</td>
<td>0.6</td>
<td>+80%</td>
</tr>
<tr>
<td>Bytecode Instructions</td>
<td>18</td>
<td>21.7</td>
<td>+20.6%</td>
</tr>
<tr>
<td>MMIX oops υ</td>
<td>11</td>
<td>13.1</td>
<td>+19.1%</td>
</tr>
<tr>
<td>MMIX mems μ</td>
<td>2.6</td>
<td>2.8</td>
<td>+5%</td>
</tr>
<tr>
<td>Scanned Elements</td>
<td>2</td>
<td>1.6</td>
<td>−20%</td>
</tr>
<tr>
<td>Branch Mispredictions</td>
<td>0.57</td>
<td>0.58</td>
<td>+2%</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th></th>
<th>CQS</th>
<th>YQS</th>
<th>Relative</th>
</tr>
</thead>
<tbody>
<tr>
<td>Running Time (from various experiments)</td>
<td></td>
<td></td>
<td>-10±2%</td>
</tr>
<tr>
<td>Comparisons</td>
<td>2</td>
<td>1.9</td>
<td>-5%</td>
</tr>
<tr>
<td>Swaps</td>
<td>0.3</td>
<td>0.6</td>
<td>+80%</td>
</tr>
<tr>
<td>Bytecode Instructions</td>
<td>18</td>
<td>21.7</td>
<td>+20.6%</td>
</tr>
<tr>
<td>MMIX oops υ</td>
<td>11</td>
<td>13.1</td>
<td>+19.1%</td>
</tr>
<tr>
<td>MMIX mems μ</td>
<td>2.6</td>
<td>2.8</td>
<td>+5%</td>
</tr>
<tr>
<td>Scanned Elements</td>
<td>2</td>
<td>1.6</td>
<td>-20%</td>
</tr>
<tr>
<td>Branch Mispredictions</td>
<td>0.57</td>
<td>0.58</td>
<td>+2%</td>
</tr>
</tbody>
</table>

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

Only plausible explanation for running time: 20% less memory transfers in YQS.
Dual-Pivot Quicksort most likely faster because of fewer memory references.

The “memory wall” calls for new view on classic algorithms. How about others?

Don’t stop looking for theoretical explanations, but do question models and assumptions!
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?

Don’t stop looking for theoretical explanations, but do question models and assumptions!
Dual-Pivot Quicksort most likely faster because of fewer **memory references**.

The “memory wall” calls for new view on classic algorithms. *How about others?*

Don’t stop looking for theoretical explanations, but do **question models** and assumptions!
Dual-Pivot Quicksort most likely faster because of fewer memory references.

The “memory wall” calls for new view on classic algorithms. How about others?

Don’t stop looking for theoretical explanations, but do question models and assumptions!