Average Case Analysis of Java 7's Dual Pivot Quicksort

Sebastian Wild

Fachbereich Informatik
TU Kaiserslautern

4. September 2012
Kolloquium zur Masterarbeit

Classic Quicksort

Classic Quicksort with Hoare's Crossing Pointer Technique

2	9	5	4	1	7	8	3	6

…by example

Classic Quicksort

Classic Quicksort with Hoare's Crossing Pointer Technique

2 9 5 4 1 7 8 3 ⑥

Select one element as **pivot**.

Classic Quicksort

Classic Quicksort with Hoare's Crossing Pointer Technique

2 9 5 4 1 7 8 3 6

Only value relative to pivot counts.

Classic Quicksort

Classic Quicksort with Hoare's Crossing Pointer Technique

Left pointer scans until first large element.

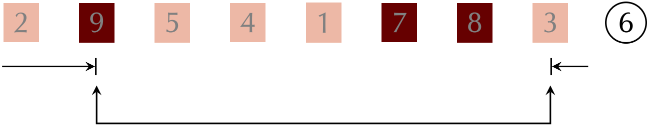
Classic Quicksort

Classic Quicksort with Hoare's Crossing Pointer Technique

Right pointer scans until first small element.

Classic Quicksort

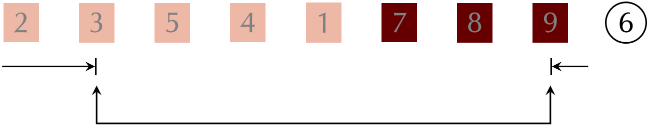
Classic Quicksort with Hoare's Crossing Pointer Technique



Swap out-of-order pair.

Classic Quicksort

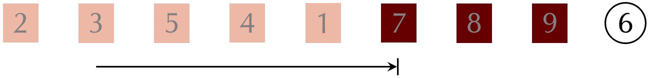
Classic Quicksort with Hoare's Crossing Pointer Technique



Swap out-of-order pair.

Classic Quicksort

Classic Quicksort with Hoare’s Crossing Pointer Technique

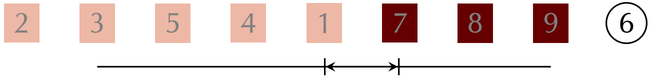


2	3	5	4	1	7	8	9	6

Left pointer scans until first large element.

Classic Quicksort

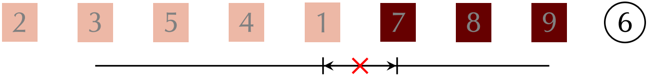
Classic Quicksort with Hoare's Crossing Pointer Technique



Right pointer scans until first small element.

Classic Quicksort

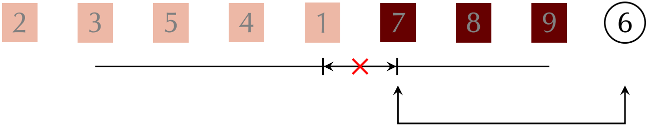
Classic Quicksort with Hoare's Crossing Pointer Technique



The pointers have crossed!

Classic Quicksort

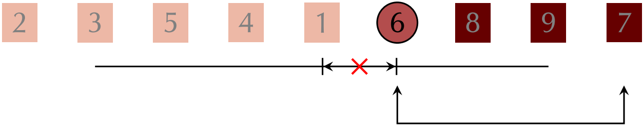
Classic Quicksort with Hoare's Crossing Pointer Technique



Swap pivot to final position.

Classic Quicksort

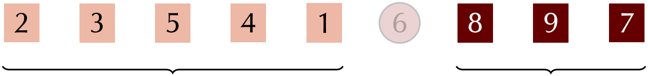
Classic Quicksort with Hoare's Crossing Pointer Technique



Partitioning done!

Classic Quicksort

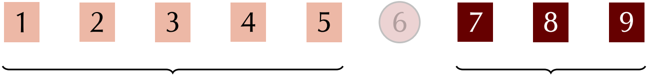
Classic Quicksort with Hoare's Crossing Pointer Technique



Recursively sort two sublists.

Classic Quicksort

Classic Quicksort with Hoare's Crossing Pointer Technique



Done.

Dual Pivot Quicksort

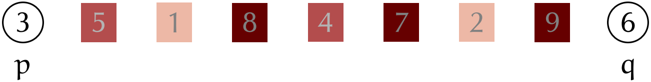
- “new” idea: use **two** pivots $p < q$

③	5	1	8	4	7	2	9	⑥
p								q

- How to do partitioning?

Dual Pivot Quicksort

- “new” idea: use **two** pivots $p < q$



- How to do partitioning?

1. For each element x, determine its **class**
   - **small** for $x < p$
   - **medium** for $p < x < q$
   - **large** for $q < x$

   by comparing x to p and/or q

2. Arrange elements according to classes

Dual Pivot Quicksort – Comparison Costs

How many comparisons to determine classes?

- Assume, we first compare with p.
  $\rightsquigarrow$ small elements need 1, others 2 comparisons
- on average: $\frac{1}{3}$ of all elements are small $\left(\frac{2}{n(n-1)} \sum_{1 \leqslant p < q \leqslant n} (p-1) \sim \frac{1}{3} n\right)$
  $\rightsquigarrow \frac{1}{3} \cdot 1 + \frac{2}{3} \cdot 2 = \frac{5}{3}$ comparisons per element
- if inputs are uniform random permutations, classes of x and y are **independent**
- $\rightsquigarrow$ **Any** partitioning method needs at least $\frac{5}{3}(n-2) \sim \frac{20}{12} n$ comparisons on average?

Dual Pivot Quicksort – Comparison Costs

How many comparisons to determine classes?

- Assume, we first compare with p.
  $\rightsquigarrow$ small elements need 1, others 2 comparisons
- on average: $\frac{1}{3}$ of all elements are small $\left(\frac{2}{n(n-1)} \sum_{1 \leqslant p < q \leqslant n} (p-1) \sim \frac{1}{3}n\right)$
  $\rightsquigarrow \frac{1}{3} \cdot 1 + \frac{2}{3} \cdot 2 = \frac{5}{3}$ comparisons per element
- if inputs are uniform random permutations,
  classes of x and y are **independent**
- $\rightsquigarrow$ **Any** partitioning method needs at least
  $\frac{5}{3}(n-2) \sim \frac{20}{12}n$ comparisons on average?
- **No:** Java 7's dual pivot Quicksort needs only $\sim \frac{19}{12}n$!
  This talk: *How is this possible?*

Beating the "Lower Bound"

- $\sim \frac{20}{12}n$ comparisons only needed,
  if there is **one** comparison **location**,
  then checks for x and y **independent**

- **But:** Can have **several** comparison locations!
  (e. g. in Yaroslavskiy, details follow)
  Here: Assume **two** locations $\mathtt{C}_1$ and $\mathtt{C}_2$ s. t.

  - $\mathtt{C}_1$ first compares with p.
  - $\mathtt{C}_2$ first compares with q.
  - $\mathtt{C}_1$ executed often, *iff* p is large.
  - $\mathtt{C}_2$ executed often, *iff* q is small.

- $\rightsquigarrow$ $\mathtt{C}_1$ executed often
  iff many small elements
  iff good chance that $\mathtt{C}_1$ needs only one comparison
  ($\mathtt{C}_2$ similar)

- $\rightsquigarrow$ less comparisons than $\frac{5}{3}$ per elements on average

Yaroslavskiy's Quicksort

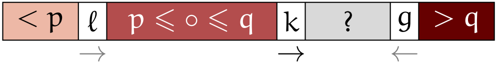
```
DualPivotQuicksortYaroslavskiy($A$, $left$, $right$)
if $right - left \geqslant 1$
    $p := A[left]$;   $q := A[right]$
    if $p > q$ then Swap $p$ and $q$ end if
    $\ell := left + 1$;   $g := right - 1$;   $k := \ell$
    while $k \leqslant g$
        if $A[k] < p$
            Swap $A[k]$ and $A[\ell]$ ;   $\ell := \ell + 1$
        else if $A[k] \geqslant q$
            while $A[g] > q$ and $k < g$ do $g := g - 1$ end while
            Swap $A[k]$ and $A[g]$ ;   $g := g - 1$
            if $A[k] < p$
                Swap $A[k]$ and $A[\ell]$ ;   $\ell := \ell + 1$
            end if
        end if
        $k := k + 1$
    end while
    $\ell := \ell - 1$;   $g := g + 1$
    Swap $A[left]$ and $A[\ell]$ ;   Swap $A[right]$ and $A[g]$
    DualPivotQuicksortYaroslavskiy($A$, $left$, $\ell - 1$)
    DualPivotQuicksortYaroslavskiy($A$, $\ell + 1$, $g - 1$)
    DualPivotQuicksortYaroslavskiy($A$, $g + 1$, $right$)
end if
```

Yaroslavskiy's Quicksort – Example

Yaroslavskiy's Dual Pivot Quicksort
(used in Oracle's Java 7 `Arrays.sort(int[])`)

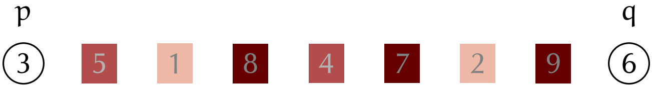
Select two elements as pivots.

Invariant:



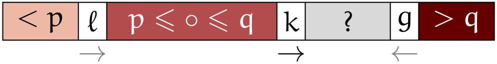
Yaroslavskiy's Quicksort – Example

Yaroslavskiy's Dual Pivot Quicksort
(used in Oracle's Java 7 `Arrays.sort(int[])`)



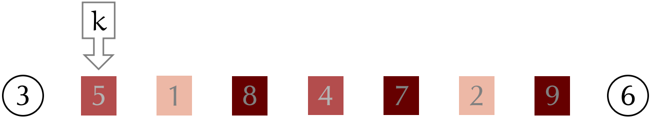
Only value relative to pivot counts.

Invariant:



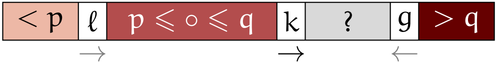
Yaroslavskiy's Quicksort – Example

Yaroslavskiy's Dual Pivot Quicksort
(used in Oracle's Java 7 `Arrays.sort(int[])`)



$A[k]$ is medium $\rightsquigarrow$ go on

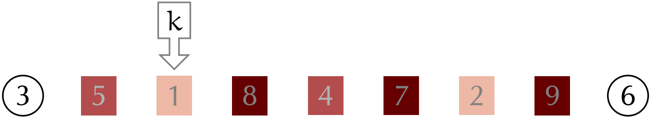
Invariant:



$< p$	ℓ	$p \leqslant \circ \leqslant q$	k	?	g	$> q$
	→		→		←	

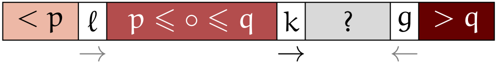
Yaroslavskiy's Quicksort – Example

Yaroslavskiy's Dual Pivot Quicksort
(used in Oracle's Java 7 `Arrays.sort(int[])`)



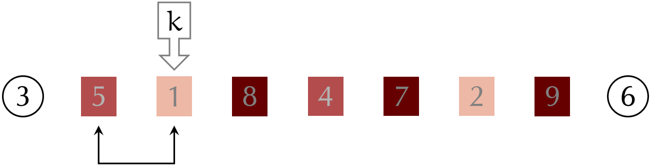
$A[k]$ is small $\rightsquigarrow$ Swap to left

Invariant:



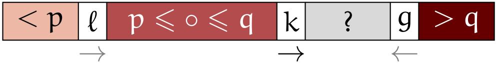
Yaroslavskiy's Quicksort – Example

Yaroslavskiy's Dual Pivot Quicksort
(used in Oracle's Java 7 `Arrays.sort(int[])`)



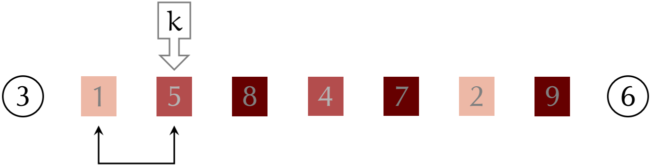
Swap small element to left end.

Invariant:



Yaroslavskiy's Quicksort – Example

Yaroslavskiy's Dual Pivot Quicksort
(used in Oracle's Java 7 `Arrays.sort(int[])`)



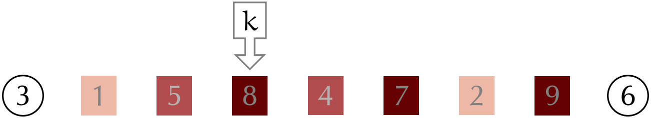
Swap small element to left end.

Invariant: $< p$ | ℓ | $p \leqslant \circ \leqslant q$ | k | ? | g | $> q$

$\ell \rightarrow$ $k \rightarrow$ $\leftarrow g$

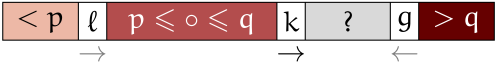
Yaroslavskiy's Quicksort – Example

Yaroslavskiy's Dual Pivot Quicksort
(used in Oracle's Java 7 `Arrays.sort(int[])`)



$A[k]$ is large $\rightsquigarrow$ Find swap partner.

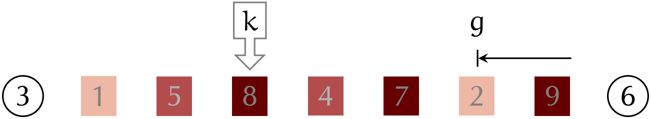
Invariant:


$< p$ | ℓ | $p \leqslant \circ \leqslant q$ | k | ? | g | $> q$

$\rightarrow$ $\rightarrow$ $\leftarrow$

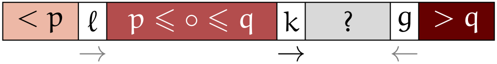
Yaroslavskiy's Quicksort – Example

Yaroslavskiy's Dual Pivot Quicksort
(used in Oracle's Java 7 `Arrays.sort(int[])`)



$A[k]$ is large $\rightsquigarrow$ Find swap partner:
g skips over large elements.

Invariant:

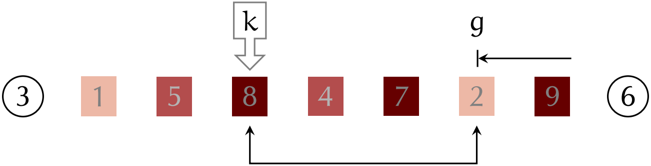


$< p$ | ℓ | $p \leqslant \circ \leqslant q$ | k | ? | g | $> q$

$\rightarrow$ $\rightarrow$ $\leftarrow$

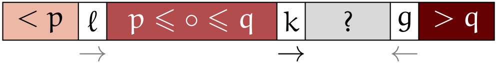
Yaroslavskiy's Quicksort – Example

Yaroslavskiy's Dual Pivot Quicksort
(used in Oracle's Java 7 `Arrays.sort(int[])`)



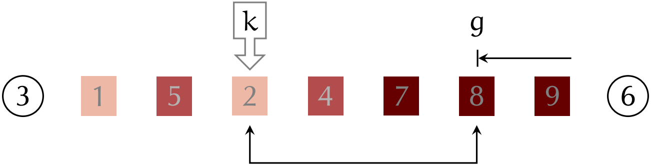
$A[k]$ is large $\rightsquigarrow$ Swap

Invariant:



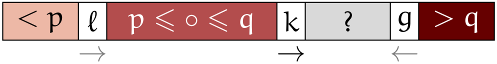
Yaroslavskiy's Quicksort – Example

Yaroslavskiy's Dual Pivot Quicksort
(used in Oracle's Java 7 `Arrays.sort(int[])`)



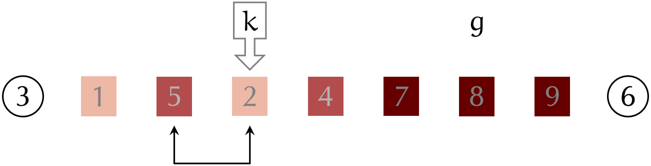
$A[k]$ is large ⇝ Swap

Invariant:



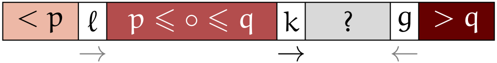
Yaroslavskiy's Quicksort – Example

Yaroslavskiy's Dual Pivot Quicksort
(used in Oracle's Java 7 `Arrays.sort(int[])`)



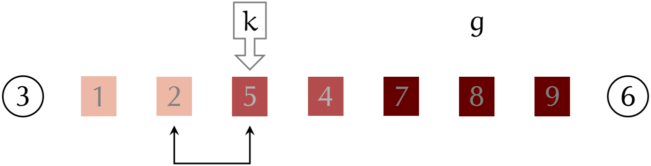
$A[k]$ is old $A[g]$, small ⇝ Swap to left

Invariant:



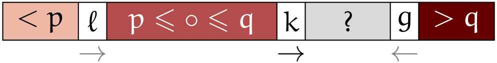
Yaroslavskiy's Quicksort – Example

Yaroslavskiy's Dual Pivot Quicksort
(used in Oracle's Java 7 `Arrays.sort(int[])`)



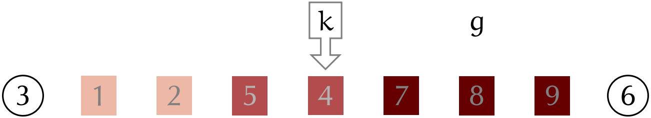
$A[k]$ is old $A[g]$, small ⇝ Swap to left

Invariant:



Yaroslavskiy's Quicksort – Example

Yaroslavskiy's Dual Pivot Quicksort
(used in Oracle's Java 7 `Arrays.sort(int[])`)



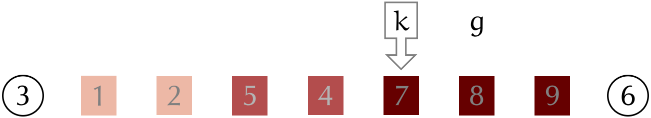
A[k] is medium ⇝ go on

Invariant:

$< p$	ℓ	$p \leqslant \circ \leqslant q$	k	?	g	$> q$
	→		→		←	

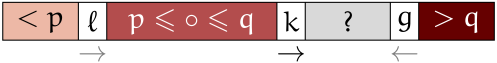
Yaroslavskiy's Quicksort – Example

Yaroslavskiy's Dual Pivot Quicksort
(used in Oracle's Java 7 `Arrays.sort(int[])`)



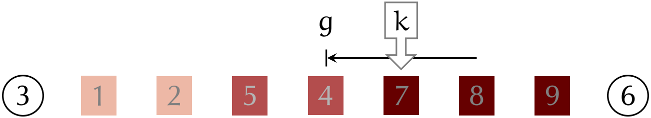
$A[k]$ is **large** $\rightsquigarrow$ Find swap partner.

Invariant:



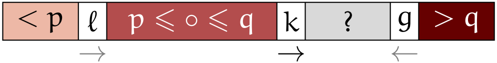
Yaroslavskiy's Quicksort – Example

Yaroslavskiy's Dual Pivot Quicksort
(used in Oracle's Java 7 `Arrays.sort(int[])`)



$A[k]$ is large $\rightsquigarrow$ Find swap partner:
g skips over large elements.

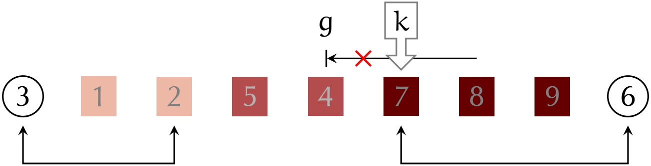
Invariant:



$< p$	ℓ	$p \leqslant \circ \leqslant q$	k	?	g	$> q$
	→		→		←	

Yaroslavskiy's Quicksort – Example

Yaroslavskiy's Dual Pivot Quicksort
(used in Oracle's Java 7 `Arrays.sort(int[])`)



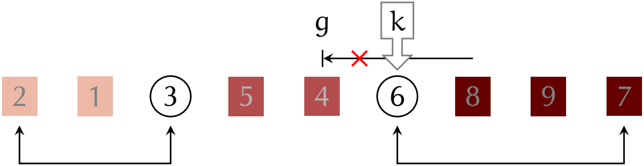
g and k have crossed!
Swap pivots in place

Invariant:

$< p$	ℓ	$p \leqslant \circ \leqslant q$	k	?	g	$> q$
	→		→		←	

Yaroslavskiy's Quicksort – Example

Yaroslavskiy's Dual Pivot Quicksort
(used in Oracle's Java 7 `Arrays.sort(int[])`)



g and k have crossed!
Swap pivots in place

Invariant:

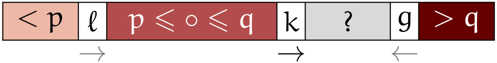
$< p$	ℓ	$p \leqslant \circ \leqslant q$	k	?	g	$> q$
	→		→		←	

Yaroslavskiy's Quicksort – Example

Yaroslavskiy's Dual Pivot Quicksort
(used in Oracle's Java 7 `Arrays.sort(int[])`)

Partitioning done!

Invariant:

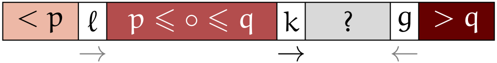


Yaroslavskiy's Quicksort – Example

Yaroslavskiy's Dual Pivot Quicksort
(used in Oracle's Java 7 `Arrays.sort(int[])`)

Recursively sort three sublists.

Invariant:



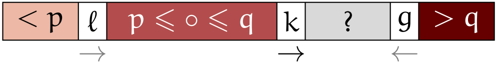
Yaroslavskiy's Quicksort – Example

Yaroslavskiy's Dual Pivot Quicksort
(used in Oracle's Java 7 `Arrays.sort(int[])`)

1 2 3 4 5 6 7 8 9

Done.

Invariant:



$< p$	ℓ	$p \leqslant \circ \leqslant q$	k	?	g	$> q$
	$\rightarrow$		$\rightarrow$		$\leftarrow$	

Yaroslavskiy's Quicksort

```
DUALPIVOTQUICKSORTYAROSLAVSKIY($A$, left, right)
if $right - left \geqslant 1$
    $p := A[left]$;    $q := A[right]$
    if $p > q$ then Swap $p$ and $q$ end if
    $\ell := left + 1$;    $g := right - 1$;    $k := \ell$
    while $k \leqslant g$
        if $A[k] < p$
            Swap $A[k]$ and $A[\ell]$ ;    $\ell := \ell + 1$
        else if $A[k] \geqslant q$
            while $A[g] > q$ and $k < g$ do $g := g - 1$ end while
            Swap $A[k]$ and $A[g]$ ;    $g := g - 1$
            if $A[k] < p$
                Swap $A[k]$ and $A[\ell]$ ;    $\ell := \ell + 1$
            end if
        end if
        $k := k + 1$
    end while
    $\ell := \ell - 1$;    $g := g + 1$
    Swap $A[left]$ and $A[\ell]$ ;    Swap $A[right]$ and $A[g]$
    DUALPIVOTQUICKSORTYAROSLAVSKIY($A$, left , $\ell - 1$ )
    DUALPIVOTQUICKSORTYAROSLAVSKIY($A$, $\ell + 1$ , $g - 1$ )
    DUALPIVOTQUICKSORTYAROSLAVSKIY($A$, $g + 1$ , right )
end if
```

- 2 comparison locations $\mathtt{C}_1$, $\mathtt{C}_2$
- $\mathtt{C}_1$ handles pointer k
  $\mathtt{C}_2$ handles pointer g
- $\mathtt{C}_1$ first checks $< p$
  $\mathtt{C}_1'$ if needed $\geqslant q$
- $\mathtt{C}_2$ first checks $> q$
  $\mathtt{C}_2'$ if needed $< p$

Analysis of Yaroslavskiy's Algorithm

- In this talk:
  - only number of comparisons (swaps similar)
  - only leading term asymptotics
  - some marginal cases excluded

  $\Big\}$ *all exact results in the paper*

- C_n expected #comparisons to sort random permutation of $\{1, \ldots, n\}$

- C_n satisfies **recurrence relation**

$$C_n = c_n + \frac{2}{n(n-1)} \sum_{1 \leqslant p < q \leqslant n} \left(C_{p-1} + C_{q-p-1} + C_{n-q} \right),$$

  with c_n expected #comparisons in **first** partitioning step

- recurrence solvable by standard methods

  $\rightsquigarrow$ $\boxed{\textbf{linear } c_n \sim a \cdot n \text{ yields } C_n \sim \tfrac{6}{5} a \cdot n \ln n.}$

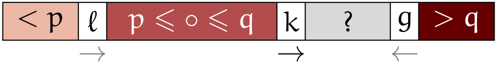
- $\rightsquigarrow$ need to compute c_n

Analysis of Yaroslavskiy's Algorithm

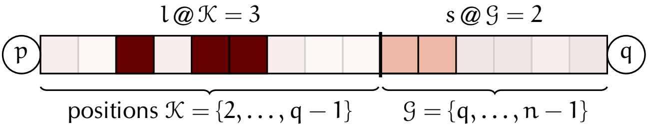
- **first** comparison for **all** elements (at $\mathtt{C}_1$ **or** $\mathtt{C}_2$)
  $\rightsquigarrow \sim n$ comparisons
- **second** comparison for **some** elements at $\mathtt{C}_1'$ resp. $\mathtt{C}_2'$
  … but how often are $\mathtt{C}_1'$ resp. $\mathtt{C}_2'$ reached?
- $\mathtt{C}_1'$: all **non**- small elements **reached** by pointer k.
  $\mathtt{C}_2'$: all **non**- large elements **reached** by pointer g.
- second comparison for medium elements **not avoidable**
  $\rightsquigarrow \sim \frac{1}{3}n$ comparisons in expectation
- $\rightsquigarrow$ it remains to count:
  large elements reached by k and
  small elements reached by g.

Analysis of Yaroslavskiy's Algorithm

- **Second** comparisons for small and large elements?
  Depends on **location**!
- $c_1' \rightsquigarrow$ **$l @ \mathcal{K}$**: number of **large** elements at positions $\mathcal{K}$.
  $c_2' \rightsquigarrow$ **$s @ \mathcal{G}$**: number of **small** elements at positions $\mathcal{G}$.
- Recall invariant:



$\rightsquigarrow$ k and g cross at (rank of) q



- for given p and q, $l @ \mathcal{K}$ **hypergeometrically** distributed
  $\rightsquigarrow \mathbb{E}\left[l @ \mathcal{K} \mid p, q\right] = (n-q)\frac{q-2}{n-2}$

Analysis of Yaroslavskiy's Algorithm

- law of total expectation:

$$\mathbb{E}\left[l\,@\,\mathcal{K}\right] = \sum_{1 \leqslant p < q \leqslant n} \Pr[\text{pivots}\,(p,q)] \cdot (n-q)\,\tfrac{q-2}{n-2} \;\sim\; \tfrac{1}{6}n$$

- Similarly: $\mathbb{E}\left[s\,@\,\mathcal{G}\right] \;\sim\; \frac{1}{12}n.$

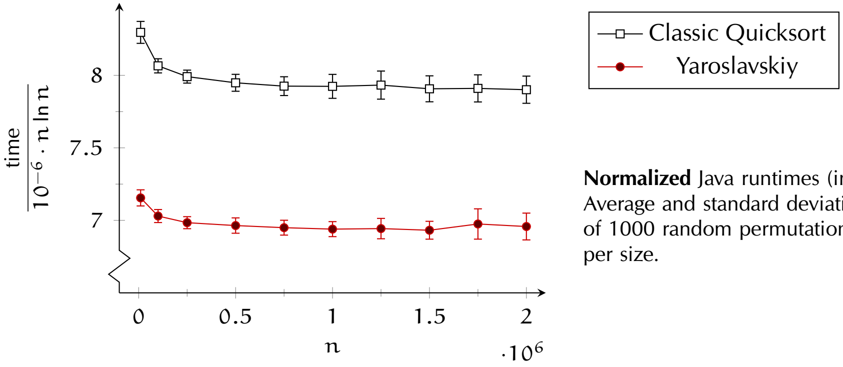
- Together: $c_n \sim (1 + \frac{1}{3} + \frac{1}{6} + \frac{1}{12})n = \frac{19}{12}n$
  $\rightsquigarrow$ **in total** $\sim \frac{6}{5} \cdot \frac{19}{12}\, n \ln n =$ **1.9** $n \ln n$ **comparison** on average

- **Classic Quicksort** needs $\sim$ **2** $n \ln n$ comparisons!

- **Swaps:**
  - $\sim$ **0.6** $n \ln n$ swaps for Yaroslavskiy's algorithm vs.
  - $\sim$ **0.$\overline{3}$** $n \ln n$ swaps for classic Quicksort

Summary

- We can exploit **asymmetries** to **save comparisons**!
- Many extra **swaps** might hurt.
- However, runtime studies favor dual pivot Quicksort: more than 10 % faster!



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

Open Questions

- Closer look at runtime: Why is Yaroslavskiy so fast in practice?
- Input distributions other than random permutations
  - equal elements
  - presorted lists
- Variances of Costs, Limiting Distributions?

Lower Bound on Comparisons

- How clever can dual pivot paritioning be?
- For lower bound, assume
  - random permutation model
  - pivots are selected uniformly
  - an **oracle** tells us, whether more small or more large elements occur
- $\rightsquigarrow$ 1 comparison for frequent extreme elements
  2 comparisons for middle and rare extreme elements

$$(n-2) \;+\; \tfrac{2}{n(n-1)} \sum_{1 \leqslant p < q \leqslant n} \big((q-p-1) \;+\; \min\{p-1, n-q\}\big)$$

$$\sim \tfrac{3}{2}n = \tfrac{18}{12}n$$

- Even with unrealistic oracle, not much better than Yaroslavskiy

Counting Primitive Instructions à la Knuth

- for implementations **MMIX** and **Java bytecode** determine exact expected overall costs
  - MMIX: processor cycles "oops" υ and memory accesses "mems" μ
  - Bytecode: #executed instructions
- divide program code into **basic blocks**
- count cost contribution for blocks
- determine expected execution frequencies of blocks
  - in first partitioning step
  - $\rightsquigarrow$ total frequency via recurrence relation

Counting Primitive Instructions à la Knuth

Results:

Algorithm	total expected costs
MMIX Classic	$\mathbf{(11\upsilon + 2.\overline{6}\mu)}(n+1)\mathcal{H}_n + (11\upsilon + 3.\overline{7}\mu)n + (-11.5\upsilon - 4.\overline{5}\mu)$
MMIX Yaroslavskiy	$\mathbf{(13.1\upsilon + 2.8\mu)}(n+1)\mathcal{H}_n + (-1.695\upsilon + 1.24\mu)n + (-1.678\overline{3}\upsilon - 1.79\overline{3}\mu)$
Bytecode Classic	$\mathbf{18}(n+1)\mathcal{H}_n + 2n - 15$
Bytecode Yaroslavskiy	$\mathbf{23.8}(n+1)\mathcal{H}_n - 8.71n - 4.74\overline{3}$

- ⇝ Classic Quicksort significantly better in both measures …
- *Why is Yaroslavskiy faster in practice?*

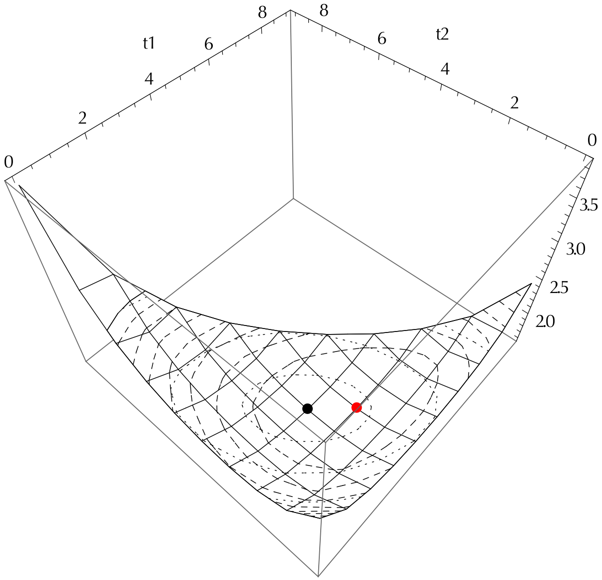
Pivot Sampling

- **Idea:** choose pivots from random **sample** of list
  - **median** for classic Quicksort
  - **tertiles** for dual pivot Quicksort

Pivot Sampling

- **Idea:** choose pivots from random **sample** of list
  - **median** for classic Quicksort
  - **tertiles** for dual pivot Quicksort**?**
  - or **asymmetric** order statistics?
- **Here:** sample of constant size k
  - choose pivots, such that t_1 elements $< p$,
    t_2 elements between p and q,
    $t_3 = k - 2 - t_1 - t_2$ larger $> q$
  - Allows to "push" pivot towards desired **order statistic** of list

Pivot Sampling



leading $n \ln n$ term coefficient of

Comparisons for $k = 11$

- tertiles (black dot)
  1.609 $n \ln n$
- minimum (red dot) at
  $(t_1, t_2, t_3) = (4, 2, 3)$
  1.585 $n \ln n$

$\rightsquigarrow$ **asymmetric** order statistics are better!