Average Case Analysis of Java 7's Dual Pivot Quicksort

Sebastian Wild Markus E. Nebel

[s_wild, nebel]@cs.uni-kl.de

Computer Science Department
University of Kaiserslautern

September 11, 2012
20th European Symposium on Algorithms

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



Select one element as **pivot**.

Classic Quicksort

Classic Quicksort with Hoare's Crossing Pointer Technique



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

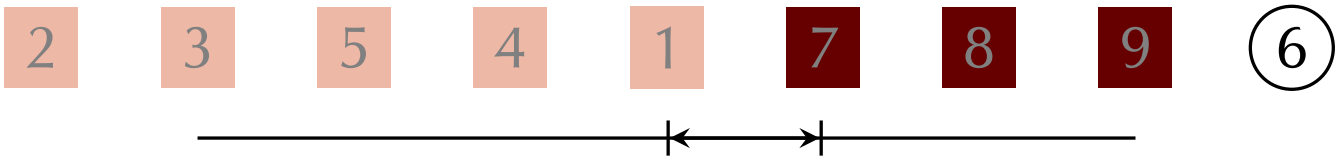
Classic Quicksort with Hoare's Crossing Pointer Technique



Left pointer scans until first large element.

Classic Quicksort

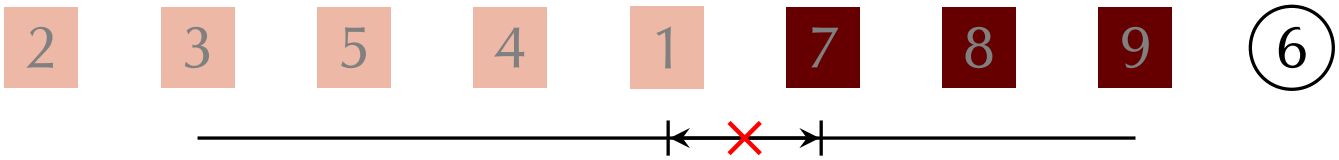
Classic Quicksort with Hoare's Crossing Pointer Technique



Right pointer scans until first small element.

Classic Quicksort

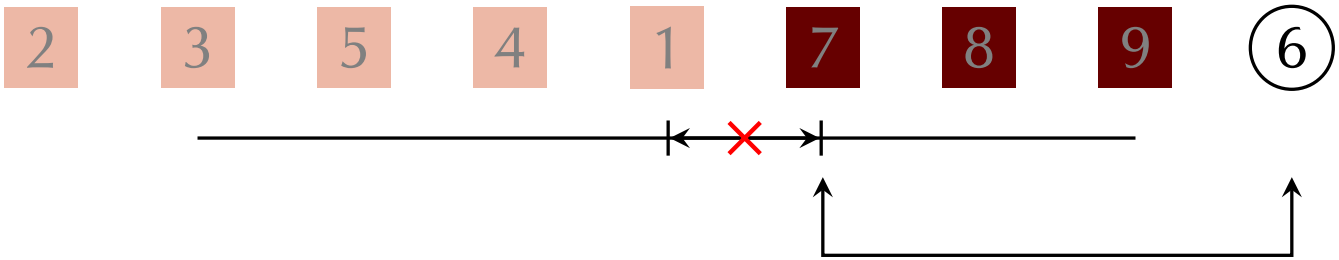
Classic Quicksort with Hoare's Crossing Pointer Technique



The pointers have crossed!

Classic Quicksort

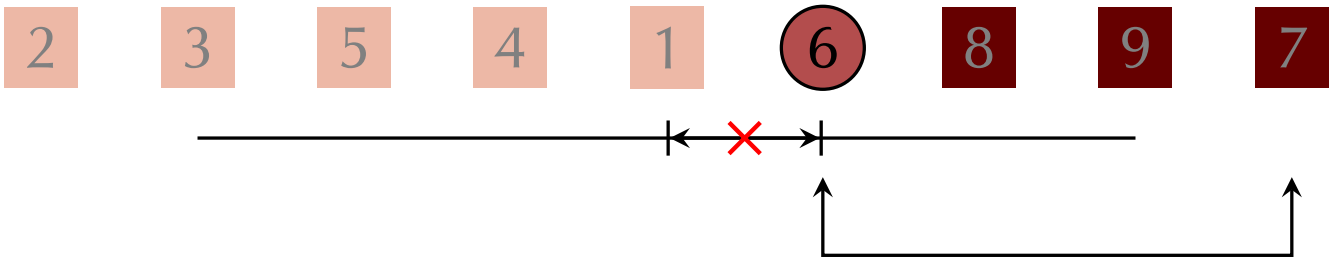
Classic Quicksort with Hoare's Crossing Pointer Technique



Swap pivot to final position.

Classic Quicksort

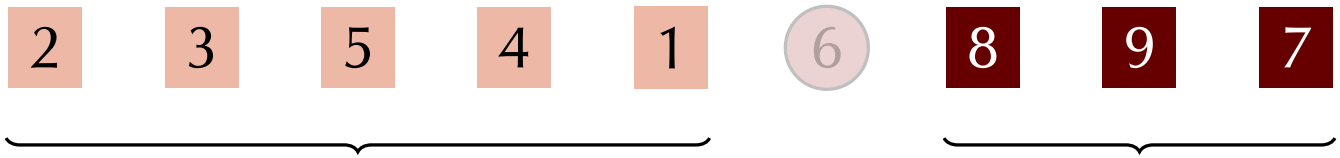
Classic Quicksort with Hoare's Crossing Pointer Technique



Partitioning done!

Classic Quicksort

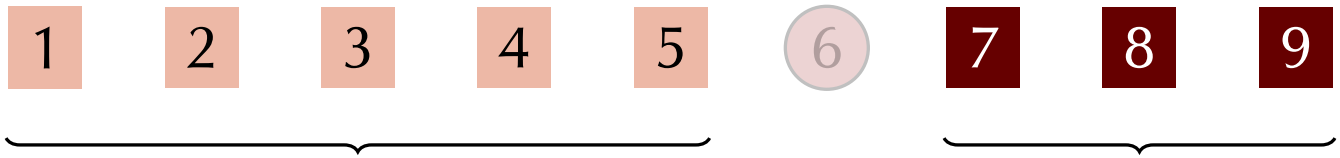
Classic Quicksort with Hoare's Crossing Pointer Technique



Recursively sort two sublists.

Classic Quicksort

Classic Quicksort with Hoare's Crossing Pointer Technique



Done.

Dual Pivot Quicksort

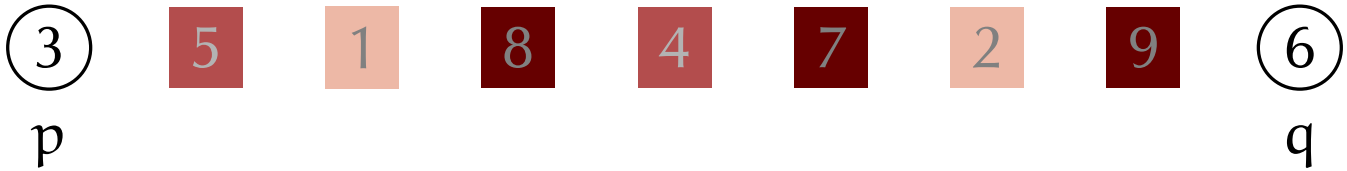
- "new" idea: use **two** pivots $p < q$



- 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 – Previous Work

- **Robert Sedgewick, 1975**
  - in-place dual pivot Quicksort implementation
  - **more** comparisons and swaps than classic Quicksort
- **Pascal Hennequin, 1991**
  - **comparisons** for list-based Quicksort with r pivots
  - $r = 2 \rightsquigarrow$ **same** #comparisons as classic Quicksort
    in one partitioning step: $\frac{5}{3}$ comparisons per element
  - $r > 2 \rightsquigarrow$ very small savings, but complicated partitioning

Dual Pivot Quicksort – Previous Work

- **Robert Sedgewick, 1975**
  - in-place dual pivot Quicksort implementation
  - **more** comparisons and swaps than classic Quicksort
- **Pascal Hennequin, 1991**
  - **comparisons** for list-based Quicksort with r pivots
  - $r = 2 \rightsquigarrow$ **same** #comparisons as classic Quicksort
    in one partitioning step: $\frac{5}{3}$ comparisons per element
  - $r > 2 \rightsquigarrow$ very small savings, but complicated partitioning

$\rightsquigarrow$ *Using two pivots does not pay.*

Dual Pivot Quicksort – Previous Work

- **Robert Sedgewick, 1975**
  - in-place dual pivot Quicksort implementation
  - **more** comparisons and swaps than classic Quicksort
- **Pascal Hennequin, 1991**
  - **comparisons** for list-based Quicksort with r pivots
  - $r = 2 \rightsquigarrow$ **same** #comparisons as classic Quicksort
    in one partitioning step: $\frac{5}{3}$ comparisons per element
  - $r > 2 \rightsquigarrow$ very small savings, but complicated partitioning

$\rightsquigarrow$ *Using two pivots does not pay.*

- **Vladimir Yaroslavskiy, 2009**
  - new implementation of dual pivot Quicksort
  - now used in Java 7's runtime library
  - **runtime studies**, no rigorous analysis

Dual Pivot Quicksort – Comparison Costs

How many comparisons to determine classes (small , medium or large) ?

- Assume, we first compare with p.
  $\rightsquigarrow$ small elements need 1, others 2 comparisons
- on average: $\frac{1}{3}$ of all elements are small
  $\rightsquigarrow \frac{1}{3} \cdot 1 + \frac{2}{3} \cdot 2 = \frac{5}{3}$ 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 (small , medium or large) ?

- Assume, we first compare with p.
  $\rightsquigarrow$ small elements need 1, others 2 comparisons
- on average: $\frac{1}{3}$ of all elements are small
  $\rightsquigarrow \frac{1}{3} \cdot 1 + \frac{2}{3} \cdot 2 = \frac{5}{3}$ 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!** (Stay tuned . . .)

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!

  Here: Assume **two** locations c_1 and c_2 s. t.

  - c_1 first compares with p.
  - c_2 first compares with q.
  - c_1 executed often, *iff* p is **large**.
  - c_2 executed often, *iff* q is **small**.

- $\rightsquigarrow$ c_1 executed often
  iff many small elements
  iff good chance that c_1 needs only one comparison
  (c_2 similar)

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

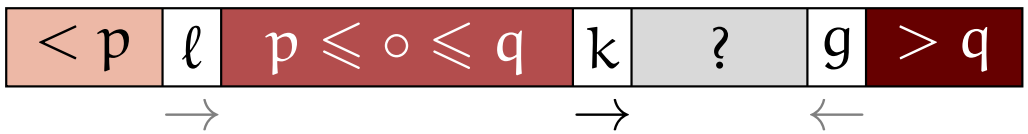
Yaroslavskiy's Quicksort – Example

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



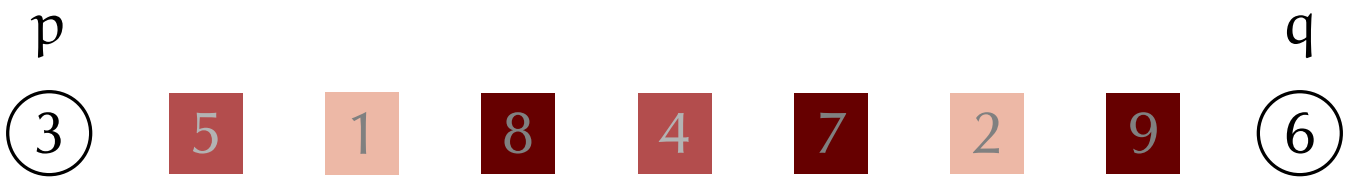
Select two elements as pivots.

Invariant:



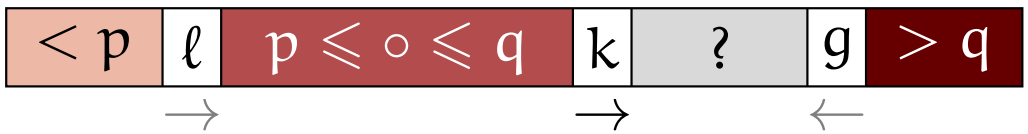
Yaroslavskiy's Quicksort – Example

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



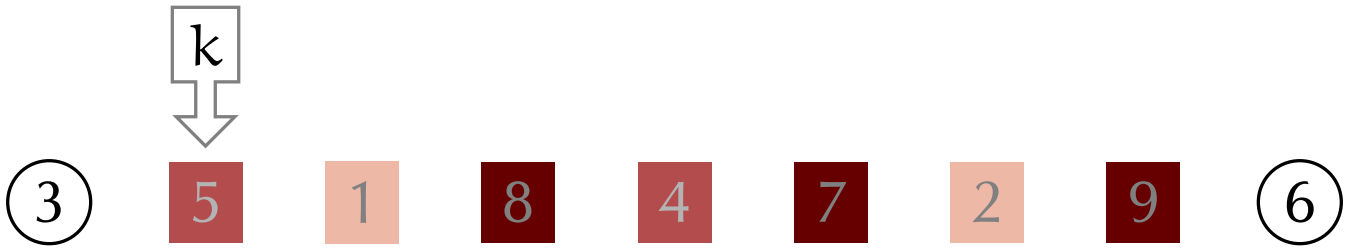
Only value relative to pivot counts.

Invariant:



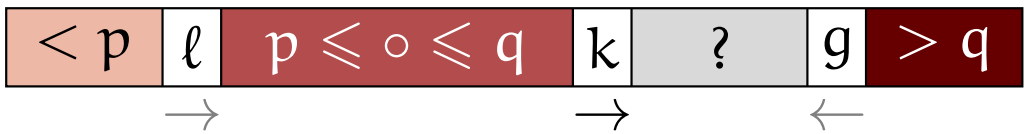
Yaroslavskiy's Quicksort – Example

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



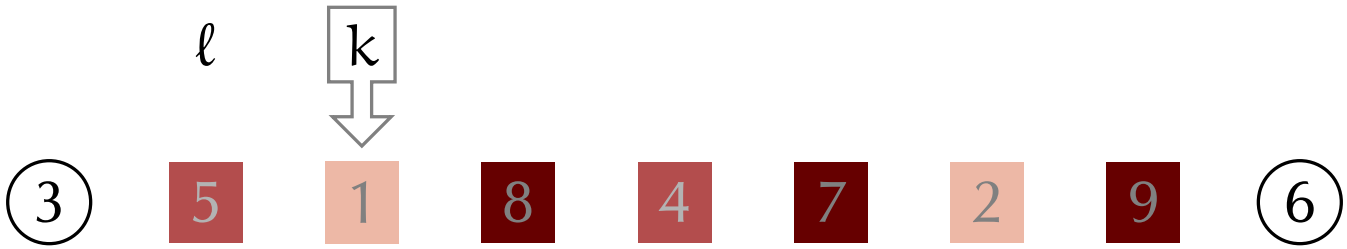
$A[k]$ is medium $\rightsquigarrow$ go on

Invariant:



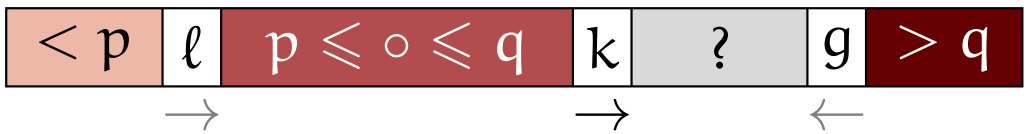
Yaroslavskiy's Quicksort – Example

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



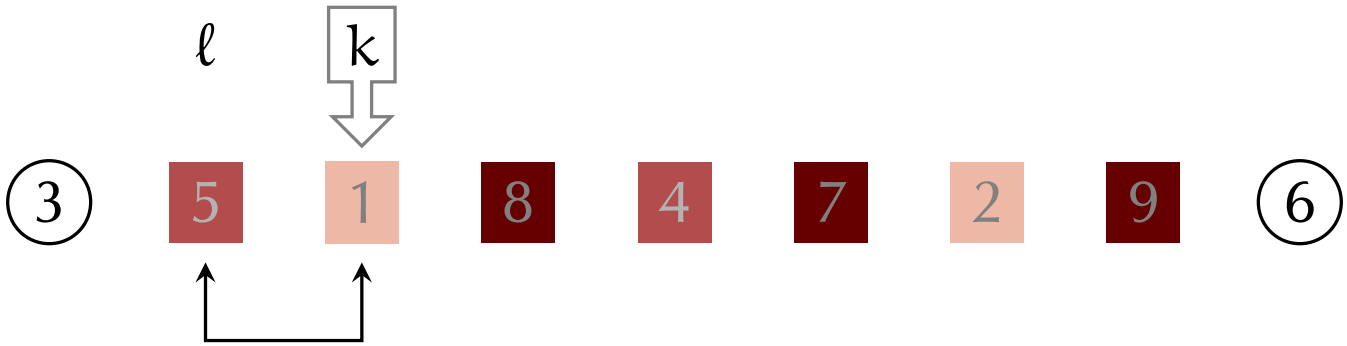
A[k] is small ⇝ Swap to left

Invariant:



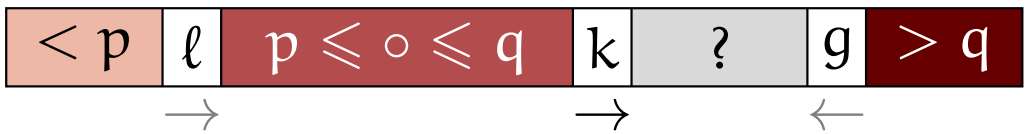
Yaroslavskiy's Quicksort – Example

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



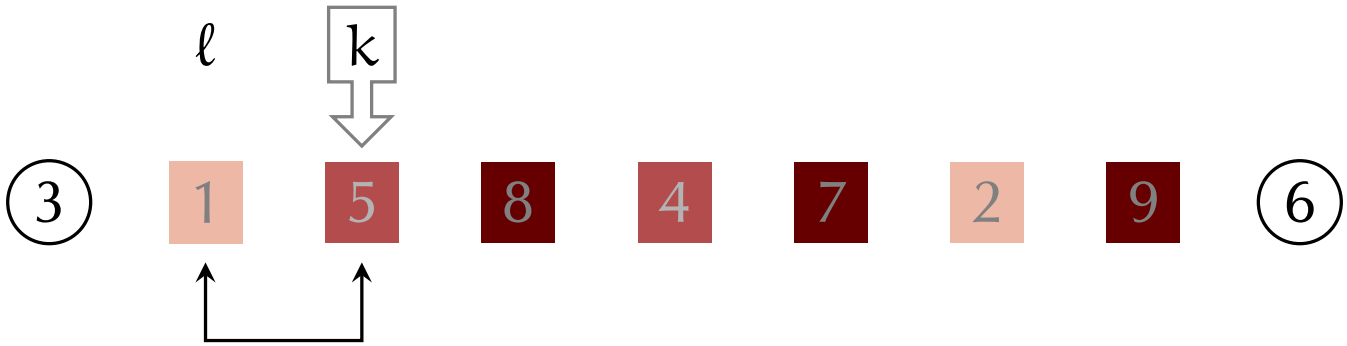
Swap small element to left end.

Invariant:



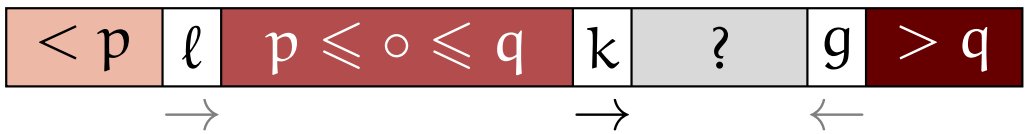
Yaroslavskiy's Quicksort – Example

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



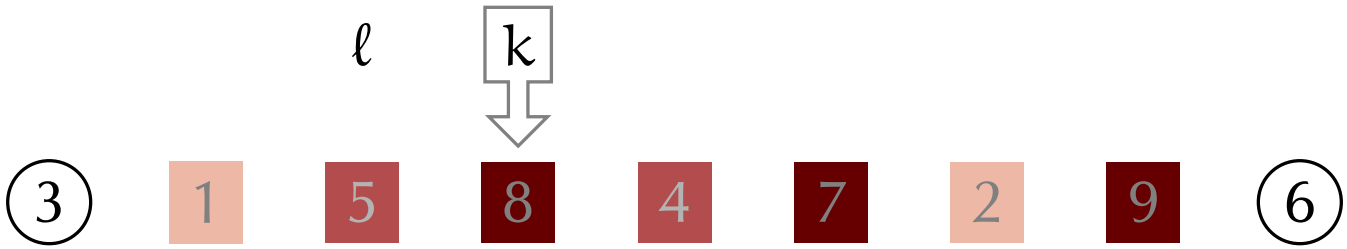
Swap small element to left end.

Invariant:



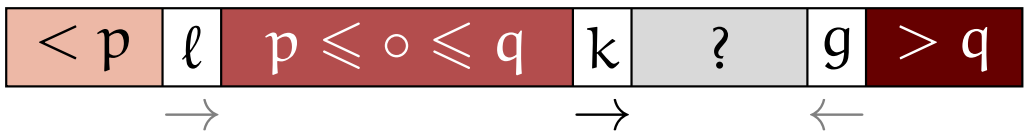
Yaroslavskiy's Quicksort – Example

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



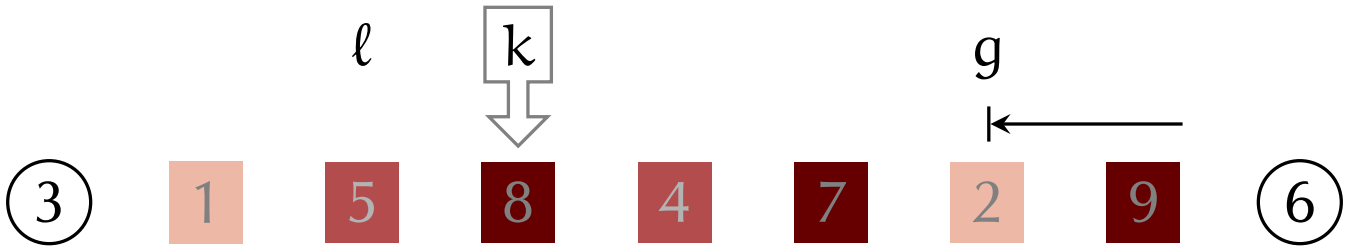
$A[k]$ is large $\rightsquigarrow$ Find swap partner.

Invariant:



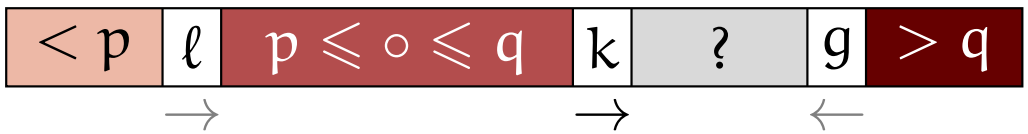
Yaroslavskiy's Quicksort – Example

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



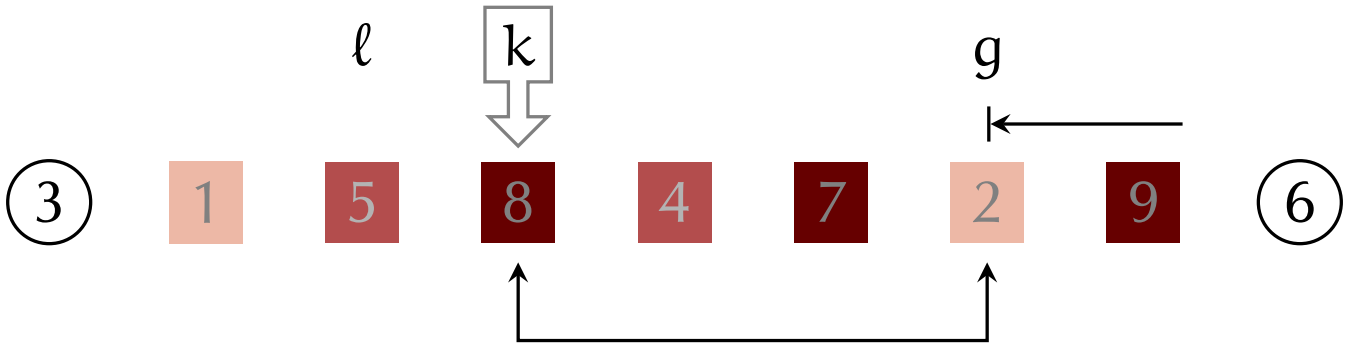
A[k] is large ⇝ Find swap partner:
g skips over large elements.

Invariant:



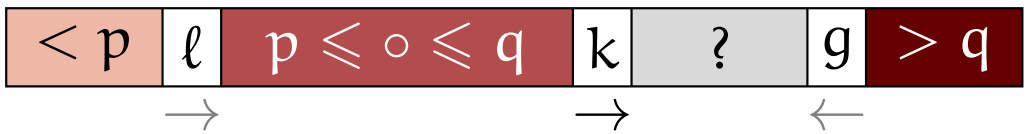
Yaroslavskiy's Quicksort – Example

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



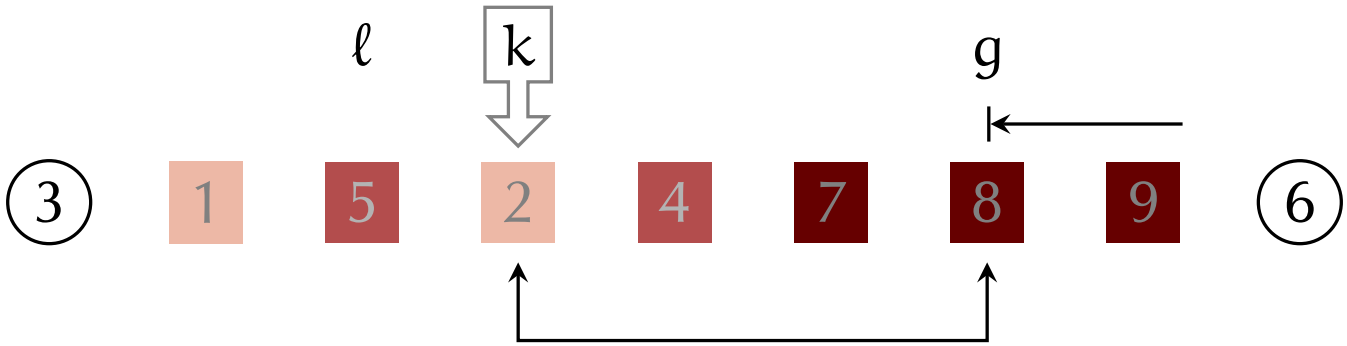
A[k] is large ⇝ Swap

Invariant:



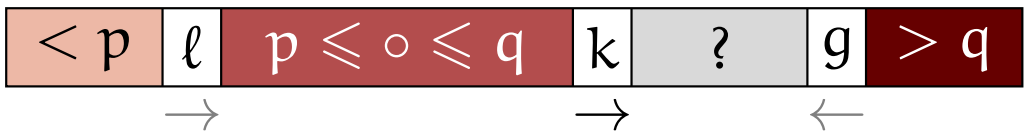
Yaroslavskiy's Quicksort – Example

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



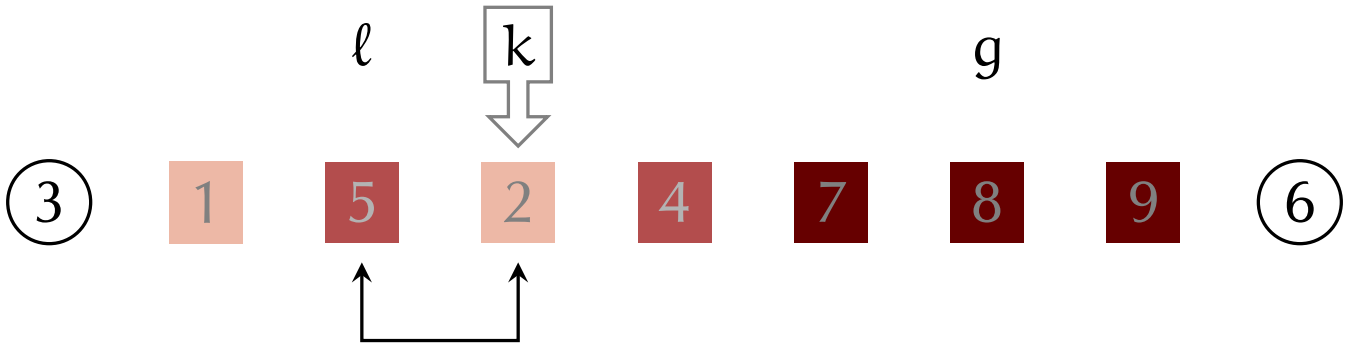
A[k] is large ⇝ Swap

Invariant:



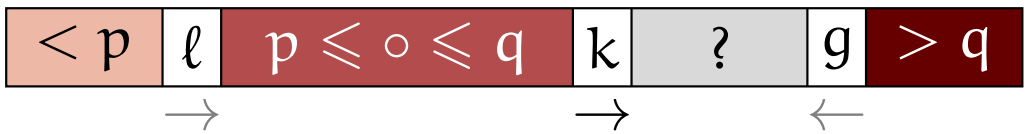
Yaroslavskiy's Quicksort – Example

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



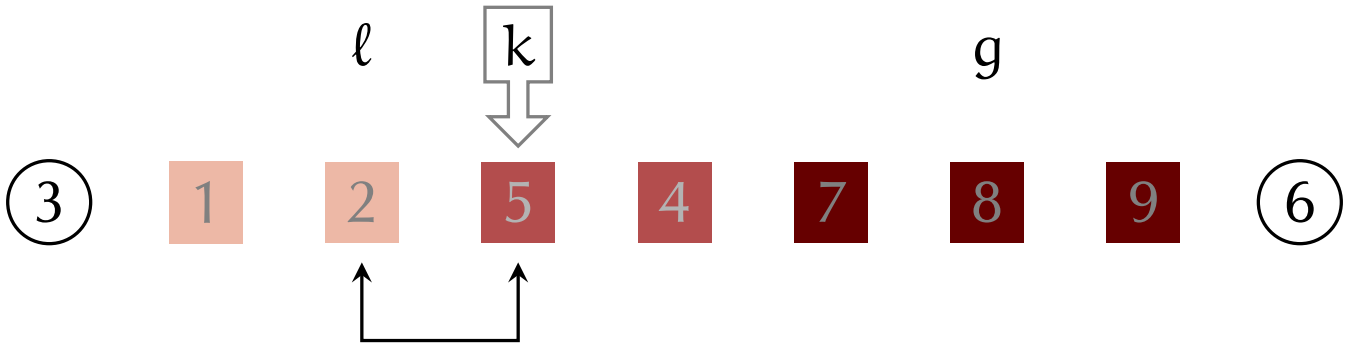
A[k] is old A[g], small ⇝ Swap to left

Invariant:



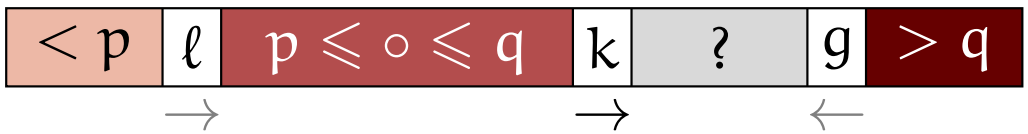
Yaroslavskiy's Quicksort – Example

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



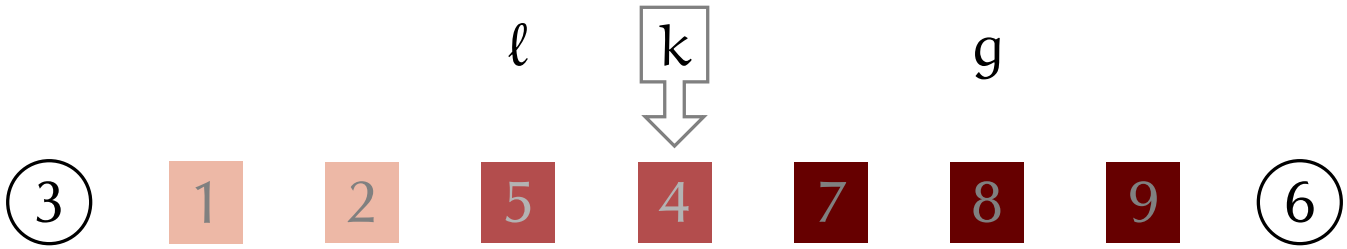
A[k] is old A[g], small $\rightsquigarrow$ Swap to left

Invariant:



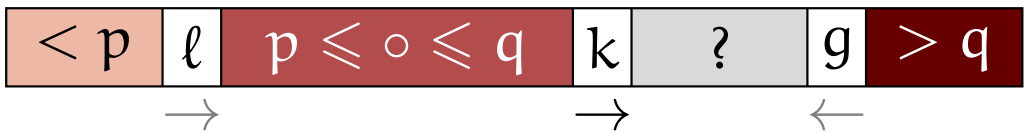
Yaroslavskiy's Quicksort – Example

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



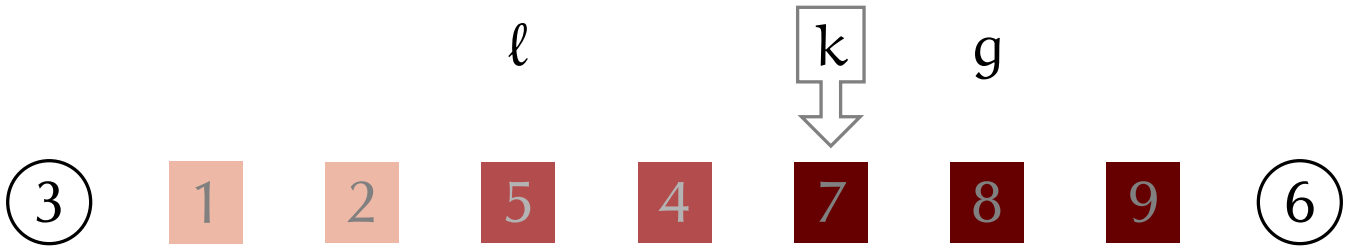
$A[k]$ is medium $\rightsquigarrow$ go on

Invariant:



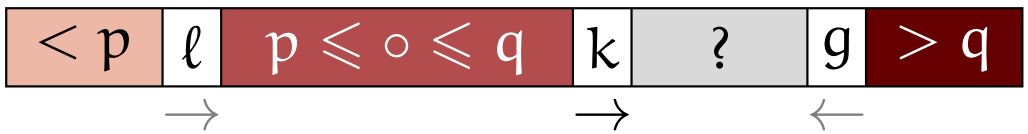
Yaroslavskiy's Quicksort – Example

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



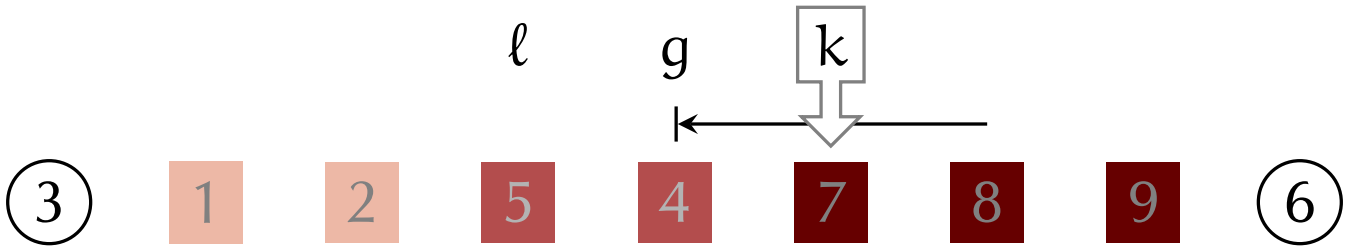
$A[k]$ is large $\rightsquigarrow$ Find swap partner.

Invariant:



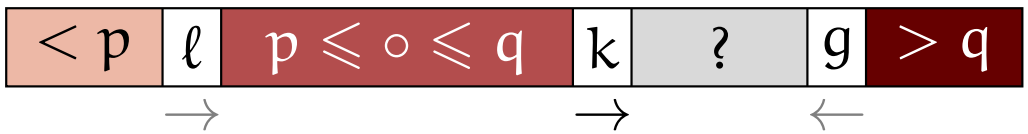
Yaroslavskiy's Quicksort – Example

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



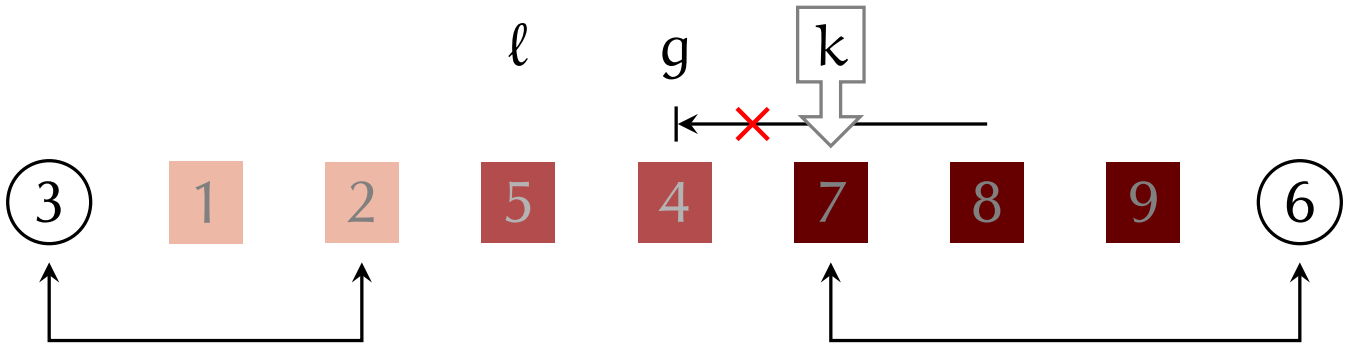
A[k] is large ⇝ Find swap partner:
g skips over large elements.

Invariant:



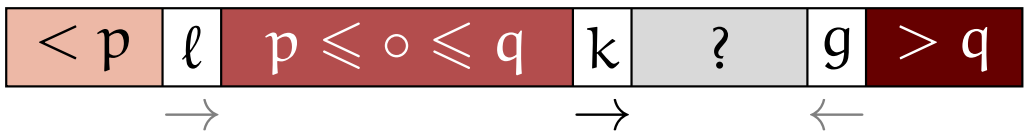
Yaroslavskiy's Quicksort – Example

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



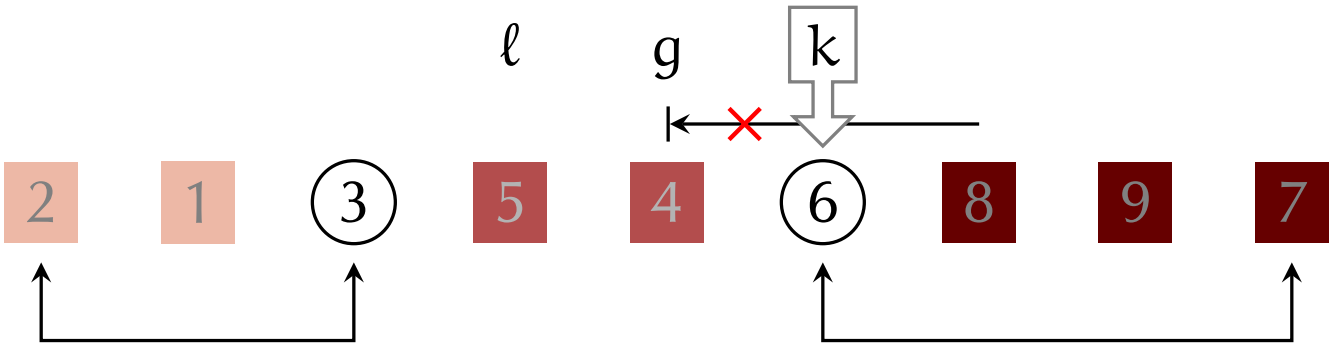
g and k have crossed!
Swap pivots in place

Invariant:



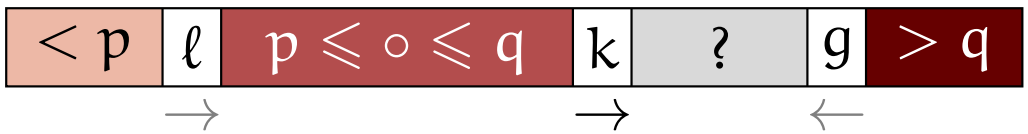
Yaroslavskiy's Quicksort – Example

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



g and k have crossed!
Swap pivots in place

Invariant:



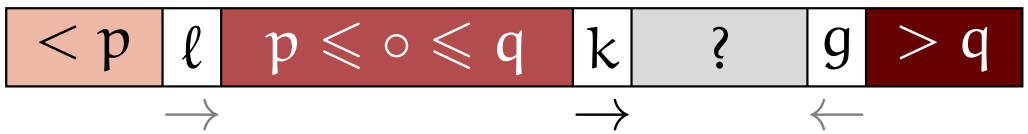
Yaroslavskiy's Quicksort – Example

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



Partitioning done!

Invariant:



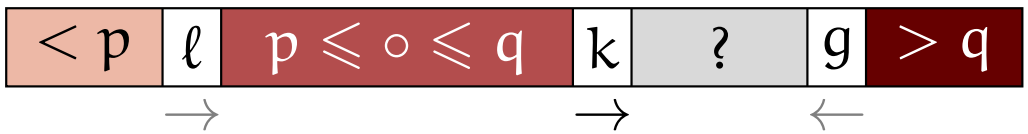
Yaroslavskiy's Quicksort – Example

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



Recursively sort three sublists.

Invariant:



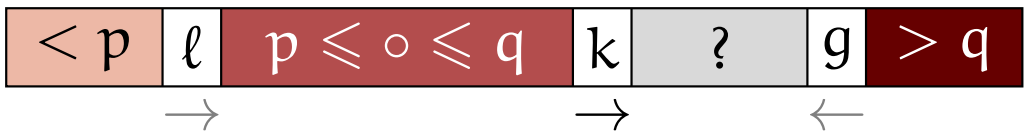
Yaroslavskiy's Quicksort – Example

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



Done.

Invariant:



Yaroslavskiy's Quicksort

```
DualPivotQuicksortYaroslavskiy(A, left, right)
 1  if right − left ⩾ 1
 2      p := A[left];    q := A[right]
 3      if p > q then Swap p and q end if
 4      ℓ := left + 1;    g := right − 1;    k := ℓ
 5      while k ⩽ g
 6          if A[k] < p
 7              Swap A[k] and A[ℓ] ;   ℓ := ℓ + 1
 8          else if A[k] ⩾ q
 9              while A[g] > q and k < g do g := g − 1 end while
10              Swap A[k] and A[g] ;    g := g − 1
11              if A[k] < p
12                  Swap A[k] and A[ℓ] ;    ℓ := ℓ + 1
13              end if
14          end if
15          k := k + 1
16      end while
17      ℓ := ℓ − 1;    g := g + 1
18      Swap A[left] and A[ℓ] ;    Swap A[right] and A[g]
19      DualPivotQuicksortYaroslavskiy(A,  left  , ℓ − 1 )
20      DualPivotQuicksortYaroslavskiy(A, ℓ + 1 , g − 1 )
21      DualPivotQuicksortYaroslavskiy(A, g + 1,  right  )
22  end if
```

Yaroslavskiy's Quicksort

```
DualPivotQuicksortYaroslavskiy(A, left, right)
 1  if right − left ⩾ 1
 2      p := A[left];    q := A[right]
 3      if p > q then Swap p and q end if
 4      ℓ := left + 1;    g := right − 1;    k := ℓ
 5      while k ⩽ g
 6  C_k     if A[k] < p
 7              Swap A[k] and A[ℓ] ;    ℓ := ℓ + 1
 8  C'_k    else if A[k] ⩾ q
 9  C_g         while A[g] > q and k < g do g := g − 1 end while
10              Swap A[k] and A[g] ;    g := g − 1
11  C'_g        if A[k] < p
12                  Swap A[k] and A[ℓ] ;    ℓ := ℓ + 1
13              end if
14          end if
15          k := k + 1
16      end while
17      ℓ := ℓ − 1;    g := g + 1
18      Swap A[left] and A[ℓ] ;    Swap A[right] and A[g]
19      DualPivotQuicksortYaroslavskiy(A,  left  , ℓ − 1 )
20      DualPivotQuicksortYaroslavskiy(A, ℓ + 1 , g − 1 )
21      DualPivotQuicksortYaroslavskiy(A, g + 1,  right  )
22  end if
```

- **2** comparison locations
- C_k handles pointer k
  C_g handles pointer g
- C_k first checks $< p$
  C'_k **if needed** $\geqslant q$
- C_g first checks $> q$
  C'_g **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

  } *all exact results in the paper*

- C_n expected #comparisons to sort random permutation of $\{1, \dots, n\}$
- C_n satisfies **recurrence relation**

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

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

- recurrence solvable by standard methods

  $\rightsquigarrow$ **linear** $c_n \sim a \cdot n$ yields $C_n \sim \frac{6}{5} a \cdot n \ln n$.

- $\rightsquigarrow$ need to compute c_n

Analysis of Yaroslavskiy's Algorithm

- **first** comparison for **all** elements (at C_k **or** C_g)
  $\rightsquigarrow \sim n$ comparisons

- **second** comparison for **some** elements at C'_k resp. C'_g
  ... but how often are C'_k resp. C'_g reached?

- C'_k: all **non-** small elements **reached** by pointer k.
  C'_g: all **non-** large elements **reached** by pointer g.

- second comparison for medium elements **not avoidable**
  $\rightsquigarrow \sim \frac{1}{3}n$ comparisons in expectation

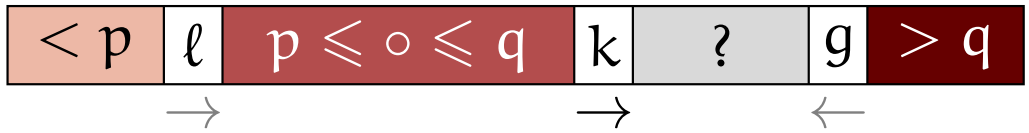
- $\rightsquigarrow$ it remains to count:
  large elements reached by k and
  small elements reached by g.

Analysis of Yaroslavskiy's Algorithm

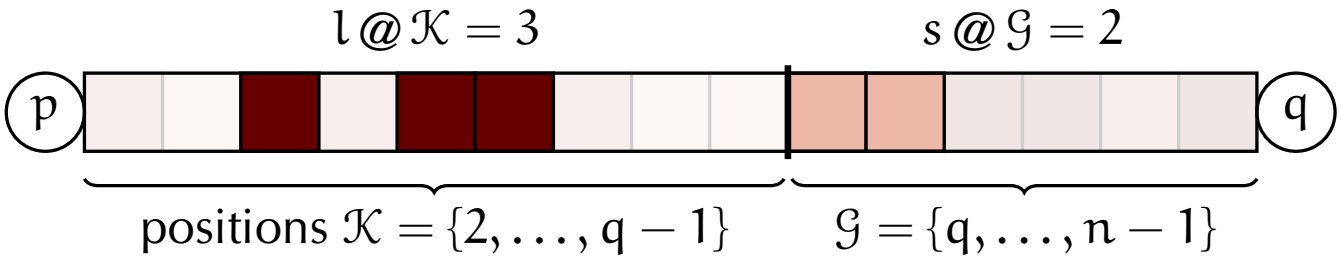
- **Second** comparisons for small and large elements?
  Depends on **location**!

- $C'_k \rightsquigarrow l@\mathcal{K}$: number of **large** elements at positions $\mathcal{K}$.
  $C'_g \rightsquigarrow s@\mathcal{G}$: 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
  $\rightsquigarrow \mathbb{E}[l@\mathcal{K} \mid p, q] = (n-q)\frac{q-2}{n-2}$

Analysis of Yaroslavskiy's Algorithm

- law of total expectation:

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

- Similarly: $\mathbb{E}[s \,@\, \mathcal{G}] \;\sim\; \tfrac{1}{12}n$.
- Summing up contributions:

$$\begin{aligned} c_n \;&\sim\; n && \text{first comparisons} \\ &\quad + \tfrac{1}{3}n && \text{medium elements} \\ &\quad + \tfrac{1}{6}n && \text{large elements at } c'_k \\ &\quad + \tfrac{1}{12}n && \text{small elements at } c'_g \\ \hline &= \tfrac{19}{12}n \end{aligned}$$

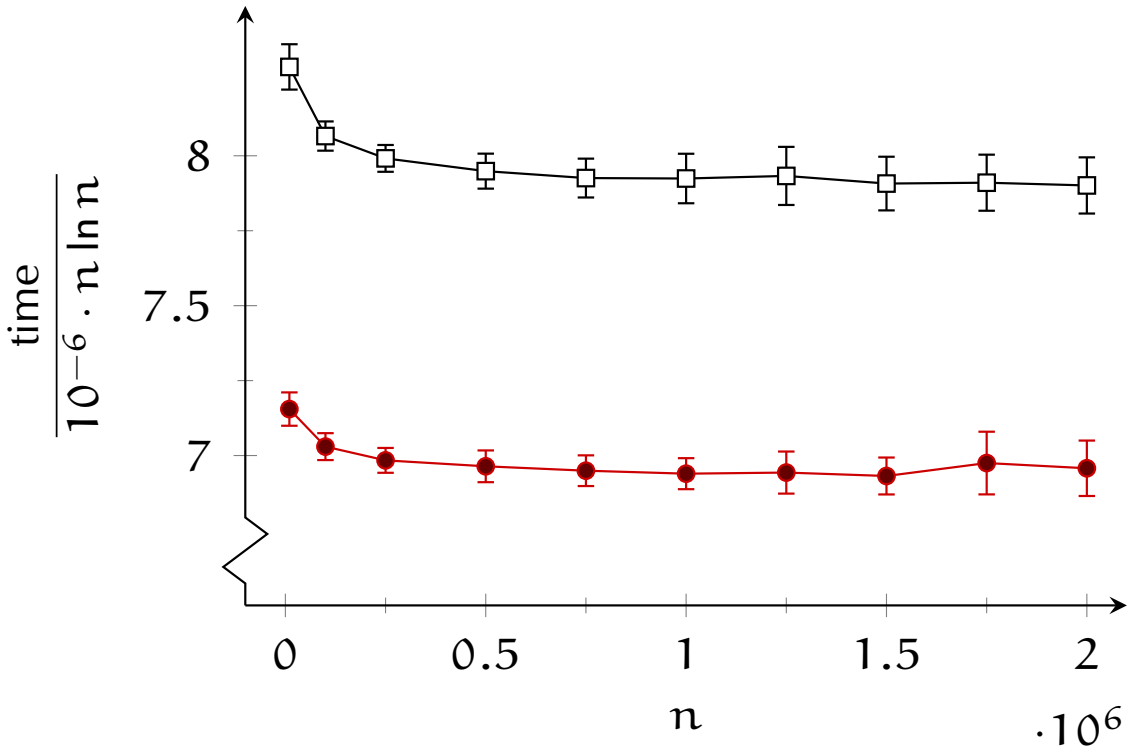
- **Recall:** "lower bound" was $\frac{20}{12}n$.

Results

- **Comparisons:**
  - Yaroslavskiy needs $\sim \frac{6}{5} \cdot \frac{19}{12}\, n \ln n = \mathbf{1.9}\, n \ln n$ on average.
  - Classic Quicksort needs $\sim \mathbf{2}\, n \ln n$ comparisons!
- **Swaps:**
  - $\sim \mathbf{0.6}\, n \ln n$ swaps for Yaroslavskiy's algorithm vs.
  - $\sim \mathbf{0.\overline{3}}\, n \ln n$ 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?
  - experimental studies?
- 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	$\mathbf{23.8}(n+1)\mathcal{H}_n - 8.71n - 4.74\overline{3}$

- $\rightsquigarrow$ Classic Quicksort significantly better in both measures ...
- *Why is Yaroslavskiy faster in practice?*

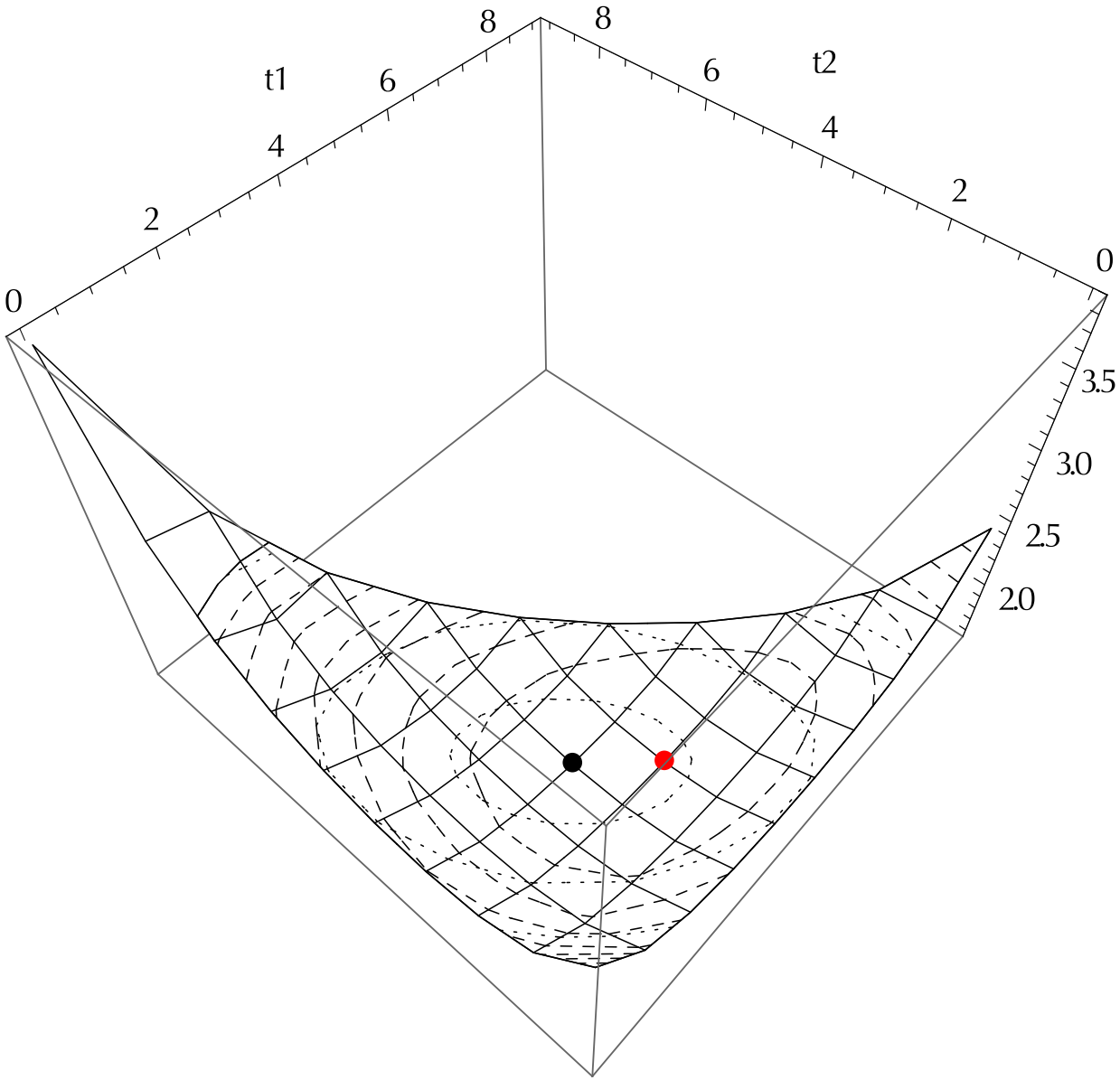
Pivot Sampling

- **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**
  $(t_1, t_2, t_3) = (3, 3, 3)$
  $\sim$ **1.609** $n \ln n$
- **minimum**
  $(t_1, t_2, t_3) = (4, 2, 3)$
  $\sim$ **1.585** $n \ln n$

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