Java 7′s Dual Pivot Quicksort – Analysis and Engineering

Markus E. Nebel

nebel@cs.uni-kl.de

based on joint work with Sebastian Wild

Seminaire Philippe Flajolet de combinatoire : 5 Décembre 2013

Sorting Algorithms in Practice

Many inventions
by algorithms comunity

vs.

Few methods
successful in practice

Sorting methods listed on Wikipedia

- C
- C++
- Java 6
- .NET
- Haskell

Quicksort
+Mergesort variant as stable sort

- Python **Timsort**

Sorting methods of standard libraries for random access data

Sorting Algorithms in Practice

Many inventions by algorithms comunity **vs.** Few methods successful in practice

Sorting methods listed on Wikipedia

- C
- C++
- Java 6
- .NET
- Haskell

Quicksort
+Mergesort variant as stable sort

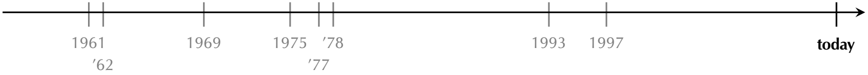
- Python **Timsort**

Sorting methods of standard libraries for random access data

History of Quicksort in Practice

- **1961,62 Hoare:** first publication, average case analysis
- **1969 Singleton:** median-of-three & Insertionsort on small subarrays
- **1975-78 Sedgewick:** detailled analysis of many optimizations
- **1993 Bentley, McIlroy:** *Engineering a Sort Function*
- **1997 Musser:** $\mathcal{O}(n \log n)$ worst case by bounded recursion depth

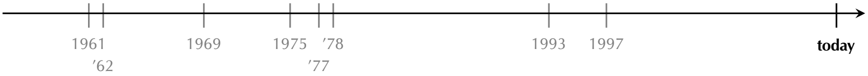
$\rightsquigarrow$ Basic algorithm settled since 1961; latest tweaks from 1990's.
Since then: Almost identical in all programming libraries!



History of Quicksort in Practice

- **1961,62 Hoare:** first publication, average case analysis
- **1969 Singleton:** median-of-three & Insertionsort on small subarrays
- **1975-78 Sedgewick:** detailled analysis of many optimizations
- **1993 Bentley, McIlroy:** *Engineering a Sort Function*
- **1997 Musser:** $\mathcal{O}(n \log n)$ worst case by bounded recursion depth

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



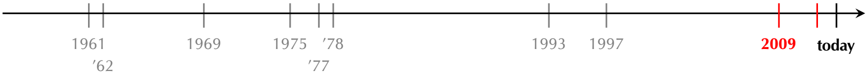
History of Quicksort in Practice

- **1961,62 Hoare:** first publication, average case analysis
- **1969 Singleton:** median-of-three & Insertionsort on small subarrays
- **1975-78 Sedgewick:** detailled analysis of many optimizations
- **1993 Bentley, McIlroy:** *Engineering a Sort Function*
- **1997 Musser:** $\mathcal{O}(n \log n)$ worst case by bounded recursion depth

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

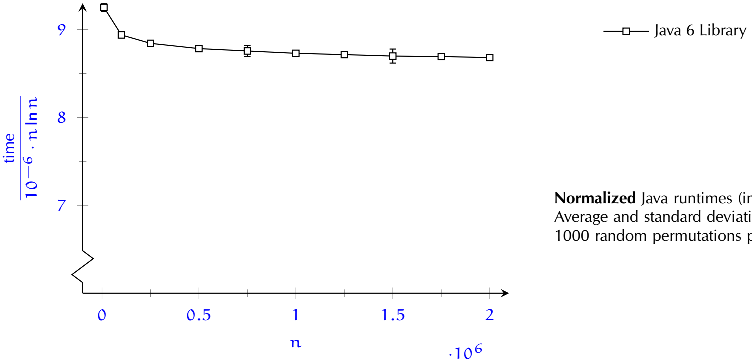
- **Until 2009**: **Java 7** switches to a new **dual pivot** Quicksort!

  Sept. 2009 **Vladimir Yaroslavskiy** announced algorithm on Java core library mailing list $\rightsquigarrow$ July 2011 **public release** of Java 7 with Yaroslavskiy's Quicksort.



Running Time Experiments

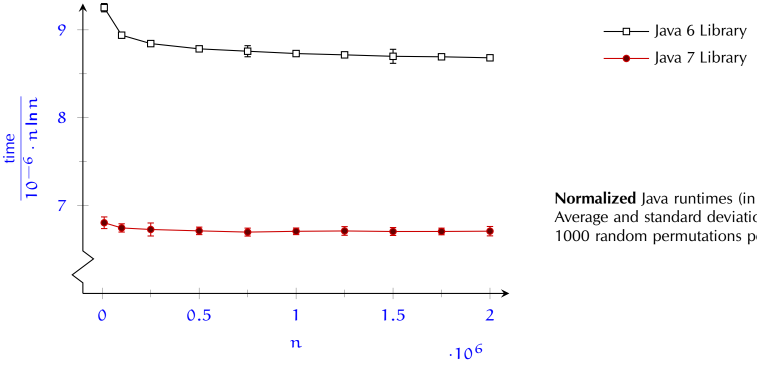
Why switch to new, unknown algorithm?



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

Running Time Experiments

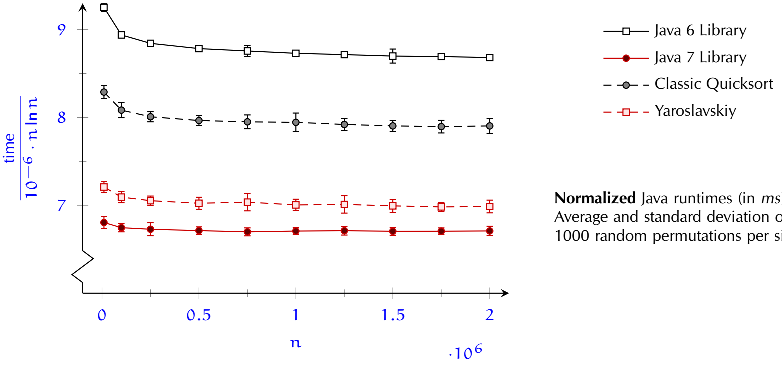
Why switch to new, unknown algorithm? Because it is faster!



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. - -□- -!

Dual Pivot Quicksort

- High Level Algorithm:
  1. Partition array arround two pivots $p \leqslant q$.
  2. Sort 3 subarrays recursively.

How to do partitioning?

Dual Pivot Quicksort

- High Level Algorithm:
  1. Partition array arround two pivots $p \leqslant q$.
  2. Sort 3 subarrays recursively.

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
  - $r = 2 \leadsto$ **same** #comparisons as classic Quicksort
    in one partitioning step: $\frac{5}{3}$ comparisons per element
  - $r > 2 \leadsto$ very small savings, but complicated partitioning

$\leadsto$ *Using two pivots does not pay, and ...*

... no theoretical explanation for impressive speedup.

Overview of talk

In this talk:

- We explain, why the new QS variant can be benefitcal even from a theoretical point of view,
- by providing a detailed average-case analysis (which carves out the reason for its success),
- this way provide more insight than running time measurements.
- Additionally, we discuss variations of the algorithm aiming for further improvements.

. . . stay tuned

Java 7's Dual Pivot Quicksort – Example

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

p								q
3	5	1	8	4	7	2	9	6

Select two elements as pivots.

Invariant:

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

Java 7's Dual Pivot Quicksort – Example

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

p								q
3	5	1	8	4	7	2	9	6

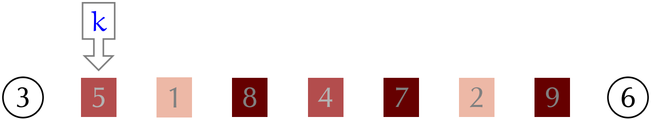
Only value relative to pivot counts.

Invariant:

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

Java 7's Dual Pivot Quicksort – Example

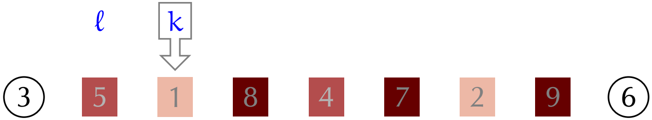
Yaroslavskiy's Dual Pivot Quicksort
(used in Oracle's Java 7 `Arrays.sort(int[])`)



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

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

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



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

Invariant:

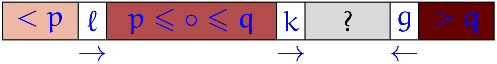
$< p$	ℓ	$p \leqslant \circ \leqslant q$	k	?	g	$> q$
	$\rightarrow$		$\rightarrow$		$\leftarrow$	

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

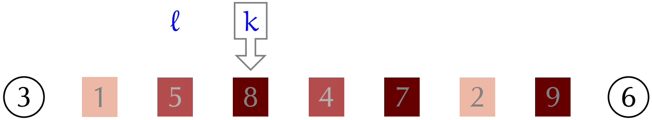
Swap small element to left end.

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

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

Java 7′s Dual Pivot Quicksort – Example

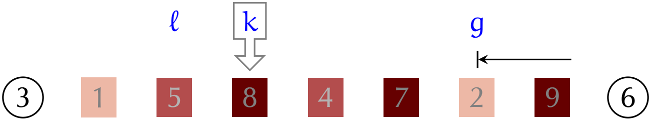
Yaroslavskiy's Dual Pivot Quicksort
(used in Oracle's Java 7 `Arrays.sort(int[])`)



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

Invariant: | $< p$ | ℓ | $p \leqslant \circ \leqslant q$ | k | ? | g | $> q$ |
$\ell \rightarrow$, $k \rightarrow$, $\leftarrow g$

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



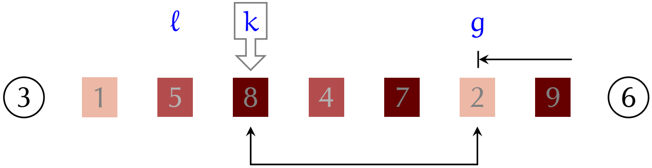
$A[k]$ is large $\rightsquigarrow$ Find swap partner:
g skips over large elements.

Invariant:

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

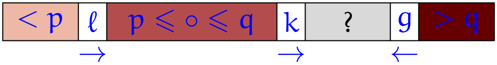
Java 7's Dual Pivot Quicksort – Example

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



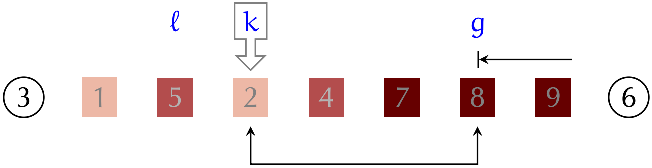
$A[k]$ is large $\rightsquigarrow$ Swap

Invariant:



Java 7’s Dual Pivot Quicksort – Example

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

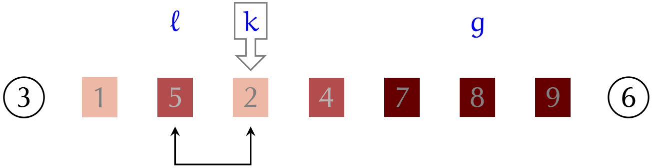


$A[k]$ is large $\leadsto$ Swap

Invariant: $< p$ | $\ell \rightarrow$ | $p \leqslant \circ \leqslant q$ | $k \rightarrow$ | ? | $\leftarrow g$ | $> q$

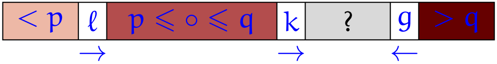
Java 7's Dual Pivot Quicksort – Example

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



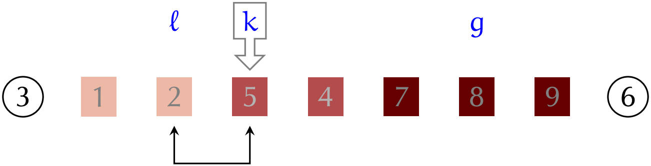
$A[k]$ is old $A[g]$, small $\rightsquigarrow$ Swap to left

Invariant:



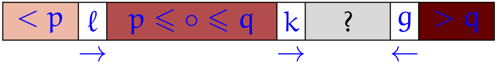
Java 7's Dual Pivot Quicksort – Example

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:



Java 7′s Dual Pivot Quicksort – Example

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

ℓ $\boxed{k}$ g

3 1 2 5 4 7 8 9 6

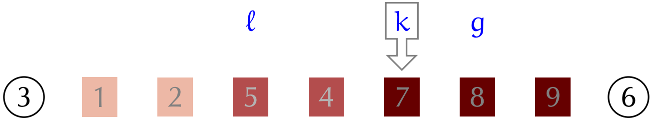
$A[k]$ is medium $\rightsquigarrow$ go on

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

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

Java 7's Dual Pivot Quicksort – Example

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

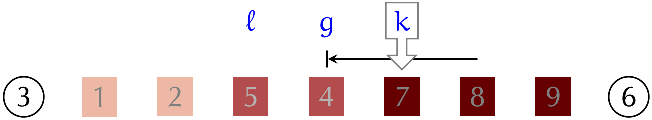


$A[k]$ is large $\rightsquigarrow$ Find swap partner.

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

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

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

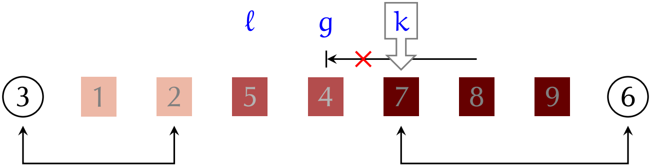


$A[k]$ is **large** $\leadsto$ Find swap partner:
g skips over large elements.

Invariant: $< p$ | $\overrightarrow{\ell}$ | $p \leqslant \circ \leqslant q$ | $\overrightarrow{k}$ | ? | $\overleftarrow{g}$ | $> q$

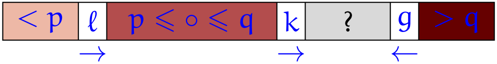
Java 7's Dual Pivot Quicksort – Example

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



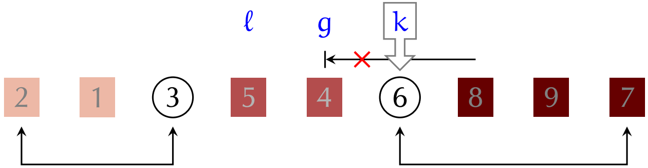
g and k have crossed!
Swap pivots in place

Invariant:



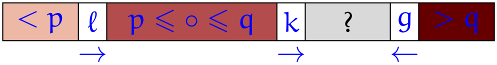
Java 7′s Dual Pivot Quicksort – Example

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



g and k have crossed!
Swap pivots in place

Invariant:



Java 7's Dual Pivot Quicksort – Example

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

2	1	3	5	4	6	8	9	7

Partitioning done!

Invariant:

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

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

$$\underbrace{2 \quad 1} \quad 3 \quad \underbrace{5 \quad 4} \quad 6 \quad \underbrace{8 \quad 9 \quad 7}$$

Recursively sort three sublists.

Invariant:

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

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

$$\underbrace{1 \quad 2} \quad 3 \quad \underbrace{4 \quad 5} \quad 6 \quad \underbrace{7 \quad 8 \quad 9}$$

Done.

Invariant:

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

Dual Pivot Quicksort – Comparison Costs

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

- Assume, we first compare x 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, knowledge about $x \neq y$ does not tell us whether y is small, medium or large.
- $\rightsquigarrow$ **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 x with p.
  $\leadsto$ small elements need 1, others 2 comparisons
- on average: $\frac{1}{3}$ of all elements are small
  $\leadsto \frac{1}{3} \cdot 1 + \frac{2}{3} \cdot 2 = \frac{5}{3}$ comparisons per element
- if inputs are uniform random permutations, knowledge about $x \neq y$ does not tell us whether y is small, medium or large.
- $\leadsto$ **Any** partitioning method needs at least
  $\frac{5}{3}(n-2) \sim \frac{20}{12}n$ comparisons on average?
- **No!**

Beating the "Lower Bound"

- $\sim \frac{20}{12}n$ comparisons only needed,
  if there is **one** comparison **location** (implying fixed order like "*first* p *then* q");
  only then checks for x and y are **independent**
- **But:** Can have **several** comparison locations!

  Here: Assume **two** locations $\mathtt{C}_1$ and $\mathtt{C}_2$ s. t.

  - $\mathtt{C}_1$ first compares with $\mathbf{p}$.
  - $\mathtt{C}_2$ first compares with $\mathbf{q}$.
  - $\mathtt{C}_1$ executed often, *iff* $\mathbf{p}$ is **large**.
  - $\mathtt{C}_2$ executed often, *iff* $\mathbf{q}$ 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 ≥ 1
    p := A[left];   q := A[right]
    if p > q then Swap p and q end if
    ℓ := left + 1;   g := right − 1;   k := ℓ
    while k ≤ g
        if A[k] < p
            Swap A[k] and A[ℓ];   ℓ := ℓ + 1
        else if A[k] ≥ 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[ℓ];   ℓ := ℓ + 1
            end if
        end if
        k := k + 1
    end while
    ℓ := ℓ − 1;   g := g + 1
    Swap A[left] and A[ℓ];   Swap A[right] and A[g]
    DUALPIVOTQUICKSORTYAROSLAVSKIY(A, left, ℓ − 1)
    DUALPIVOTQUICKSORTYAROSLAVSKIY(A, ℓ + 1, g − 1)
    DUALPIVOTQUICKSORTYAROSLAVSKIY(A, g + 1, right)
end if
```

Yaroslavskiy's Quicksort

DualPivotQuicksortYaroslavskiy(A, *left*, *right*)

```
     if right − left ≥ 1
         p := A[left];   q := A[right]
         if p > q then Swap p and q end if
         ℓ := left + 1;   g := right − 1;   k := ℓ
         while k ≤ g
C_k          if A[k] < p
                 Swap A[k] and A[ℓ] ;   ℓ := ℓ + 1
C'_k         else if A[k] ≥ q
C_g              while A[g] > q and k < g do g := g − 1 end while
                 Swap A[k] and A[g] ;   g := g − 1
C'_g             if A[k] < p
                     Swap A[k] and A[ℓ] ;   ℓ := ℓ + 1
                 end if
             end if
             k := k + 1
         end while
         ℓ := ℓ − 1;   g := g + 1
         Swap A[left] and A[ℓ] ;   Swap A[right] and A[g]
         DualPivotQuicksortYaroslavskiy(A, left , ℓ − 1 )
         DualPivotQuicksortYaroslavskiy(A, ℓ + 1 , g − 1 )
         DualPivotQuicksortYaroslavskiy(A, g + 1 , right )
     end if
```

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

  all exact results in paper
- C_n expected #comparisons to sort random permutation of $\{1, \dots, n\}$
- 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

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

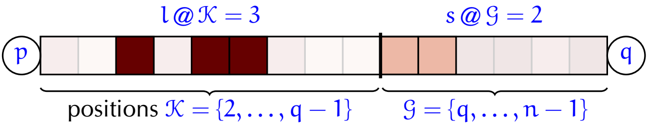
- **Second** comparisons for small and large elements?
  Depends on **location**!

- $c_k' \rightsquigarrow \mathrm{l@}\mathcal{K}$: number of **large** elements at positions $\mathcal{K}$.
  $c_g' \rightsquigarrow \mathrm{s@}\mathcal{G}$: number of **small** elements at positions $\mathcal{G}$.

- Recall invariant:

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

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



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

Analysis of Yaroslavskiy's Algorithm

- law of total expectation:

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

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

$c_n \sim n$	first comparisons
$+\frac{1}{3}n$	medium elements
$+\frac{1}{6}n$	large elements at C'_k
$+\frac{1}{12}n$	small elements at C'_g
$= \frac{19}{12}n$	

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

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

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

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

- Even with unrealistic oracle, not much better than Yaroslavskiy

Gathering 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!

Interestingly, the same partitioning yields a `Quickselect` algorithm needing a larger number of comparisons on average!

- **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

Gathering 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!

Interestingly, the same partitioning yields a `Quickselect` algorithm needing a larger number of comparisons on average!

- **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

Engineering Quicksort

Analogous to classic Quicksort

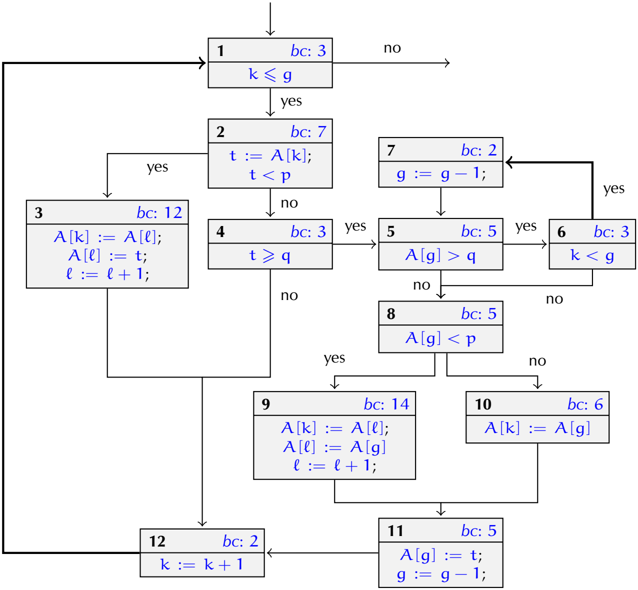
- switch to InsertionSort for subproblems of size $\leqslant w$,
- choose pivots from random **sample** of input
  - **median** for classic Quicksort
  - **tertiles** for dual pivot Quicksort

Engineering Quicksort

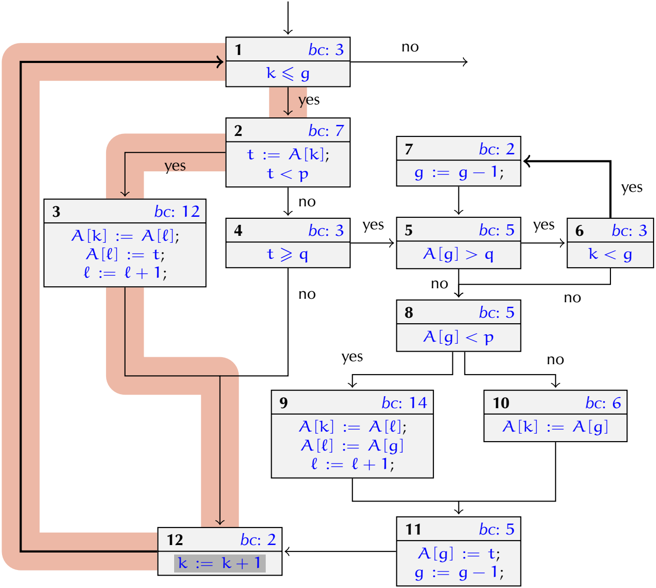
Analogous to classic Quicksort

- switch to InsertionSort for subproblems of size $\leqslant w$,
- choose pivots from random **sample** of input
  - **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

Control Flow Graph of Partitioning Loop



Control Flow Graph of Partitioning Loop



Cycle 1

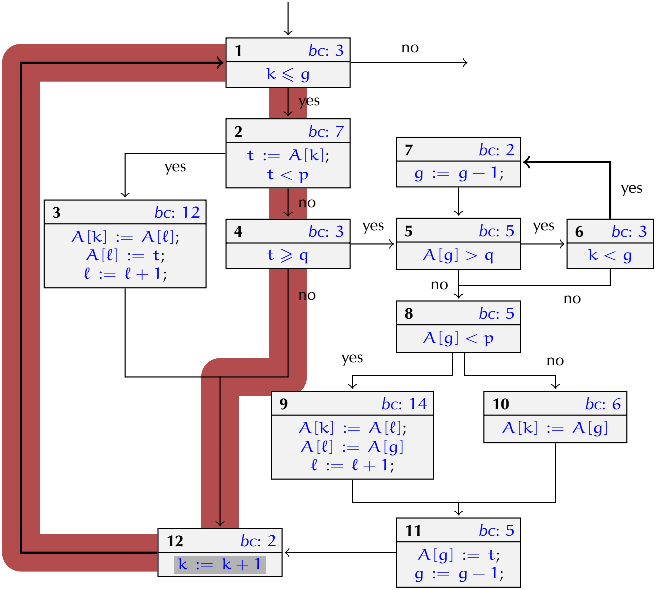
$A[k]$: small

$A[g]$: —

$\Delta(g-k)$: 1

Bytecode Instructions: 24

Control Flow Graph of Partitioning Loop



Cycle 2

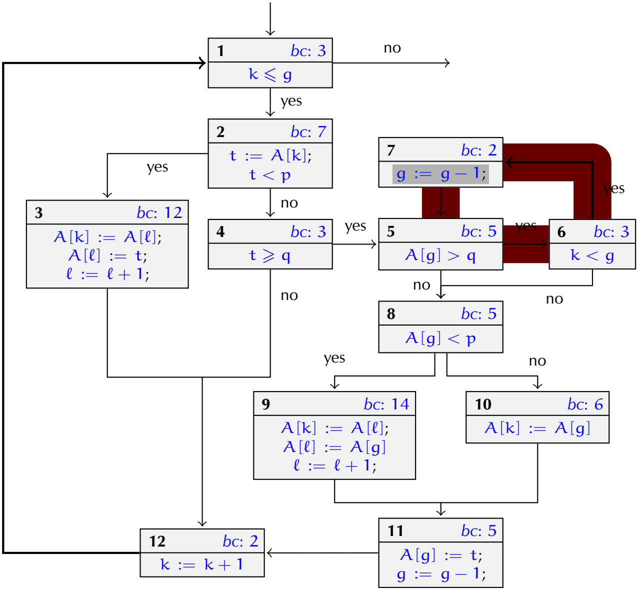
$A[k]$: medium

$A[g]$: —

$\Delta(g - k)$: 1

Bytecode Instructions: 15

Control Flow Graph of Partitioning Loop



Cycle 3

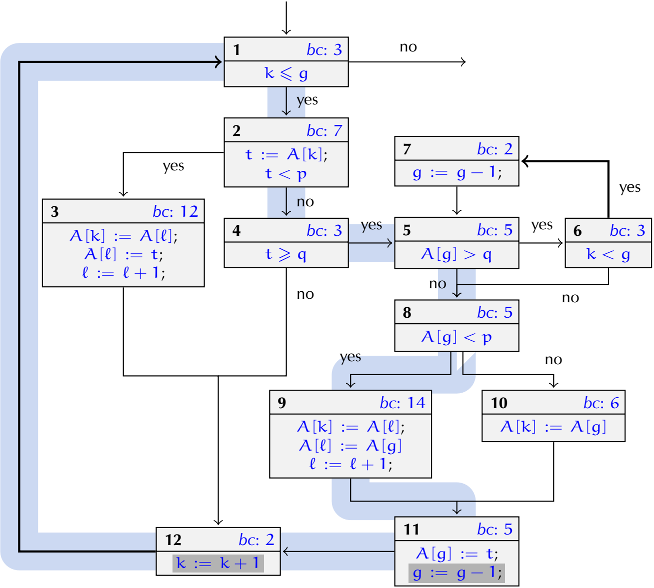
$A[k]$: large

$A[g]$: large

$\Delta(g - k)$: 1

Bytecode Instructions: 10

Control Flow Graph of Partitioning Loop



Cycle 4

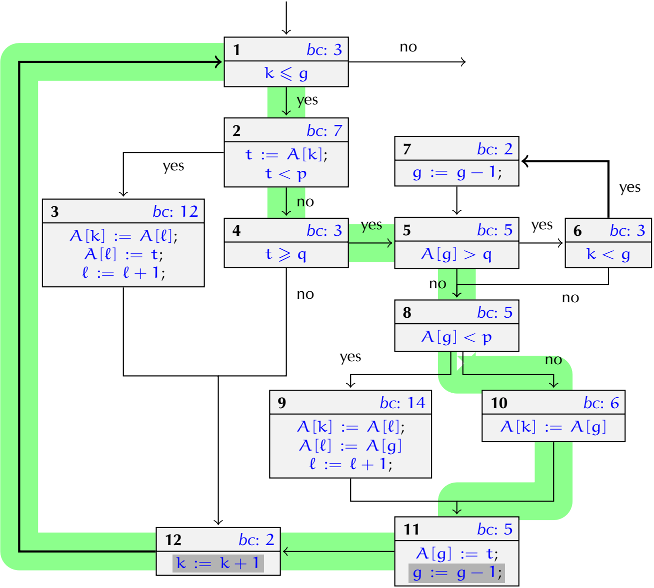
$A[k]$: large

$A[g]$: small

$\Delta(g - k)$: 2

Bytecode Instructions: 44

Control Flow Graph of Partitioning Loop



Cycle 5

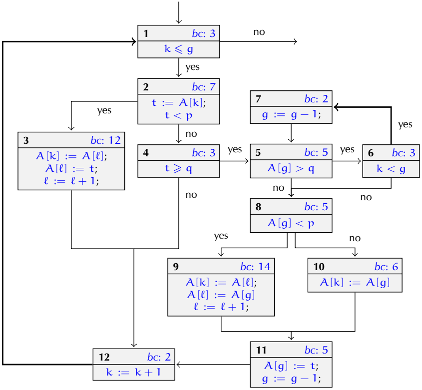
$A[k]$: large

$A[g]$: medium

$\Delta(g-k)$: 2

Bytecode Instructions: 36

Asymmetry



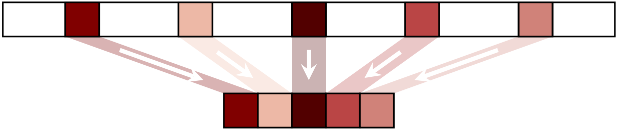
- Algorithm is **asymmetric:**
  - cycles have different cost
  - $\rightsquigarrow$ would rather execute cheap ones often
- cycles chosen by **classes** small, medium or large
- probability for classes depends on **pivot values**

$\rightsquigarrow$ *Maybe we can "influence pivot values accordingly"?*

Pivot Sampling

- Well-known optimization for **classic** Quicksort: **median-of-three**
  $\rightsquigarrow$ pivot closer to **median** of whole list

- In **JRE7 Quicksort implementation**: natural extension for 2 pivots:



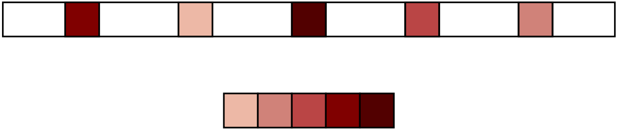
tertiles-of-five 

$\rightsquigarrow$ pivots closer to **tertiles** of whole list

- 9 other possibilities to pick p and q out of 5 elements:

Pivot Sampling

- Well-known optimization for **classic** Quicksort: **median-of-three**
  $\rightsquigarrow$ pivot closer to **median** of whole list
- In **JRE7 Quicksort implementation**: natural extension for 2 pivots:



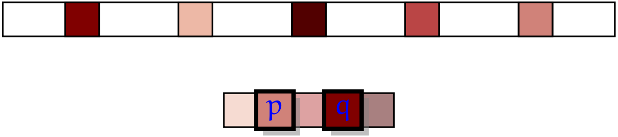
tertiles-of-five 
$\rightsquigarrow$ pivots closer to **tertiles** of whole list

- **9** other possibilities to pick p and q out of 5 elements:

Pivot Sampling

- Well-known optimization for **classic** Quicksort: **median-of-three**
  $\rightsquigarrow$ pivot closer to **median** of whole list

- In **JRE7 Quicksort implementation**: natural extension for 2 pivots:



tertiles-of-five 

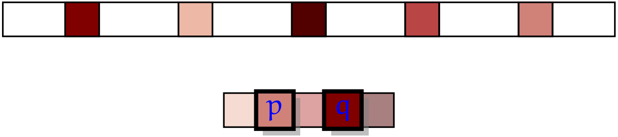
$\rightsquigarrow$ pivots closer to **tertiles** of whole list

- 9 other possibilities to pick p and q out of 5 elements:

Pivot Sampling

- Well-known optimization for **classic** Quicksort: **median-of-three**
  $\rightsquigarrow$ pivot closer to **median** of whole list

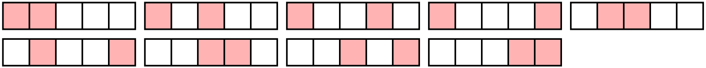
- In **JRE7 Quicksort implementation**: natural extension for 2 pivots:



tertiles-of-five 

$\rightsquigarrow$ pivots closer to **tertiles** of whole list

- **9** other possibilities to pick p and q out of 5 elements:



Optimizing Pivot Sampling

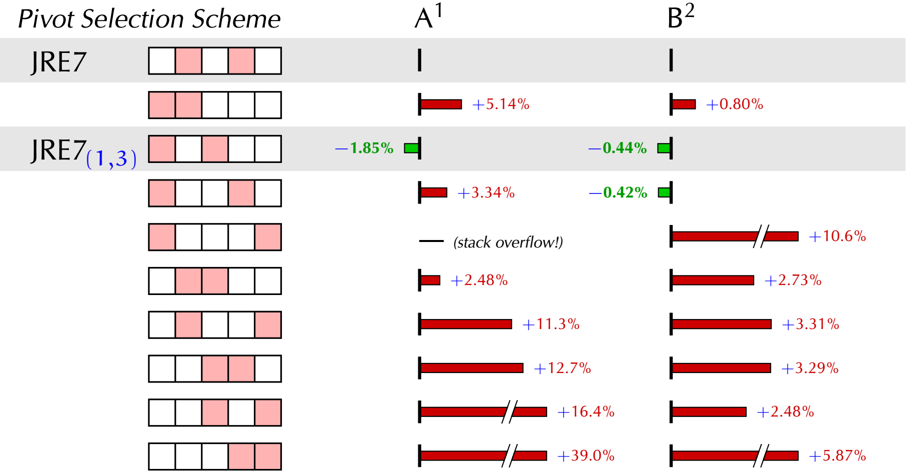
Which are "good" pivot selection schemes?

Is the symmetric choice  *best possible?*

- Need objective function to optimize
- Typical approaches to judge efficiency:
  - (A) Count number of **basic operations**.
    (Here: number of executed **Java Bytecode instructions**.)
  - (B) Measure total **running time**.

Optimizing Pivot Sampling

Relative performance of pivot sampling compared to tertiles-of-five:



[1] Average number of executed bytecodes on almost sorted lists of length 10^5.
[2] Average running time on random permutations of length 10^6.

Pivot Sampling

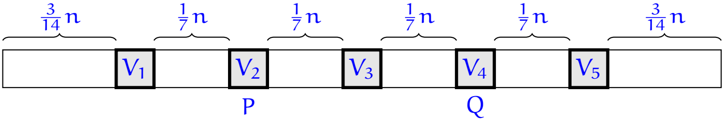


Figure : The five sample elements in Oracle's Java 7 implementation of Yaroslavskiy's dual-pivot Quicksort are chosen such that their distances are approximately as given above.

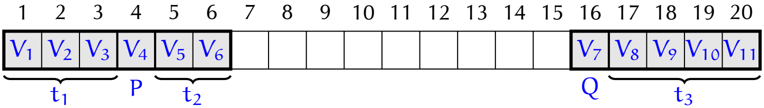


Figure : Location of the sample in our implementation of generalized pivot sampling, here with exemplary parameters $\mathbf{t} = (3, 2, 4)$. Only the non-shaded region is subject to partitioning with Yaroslavskiy's method.

Pivot Sampling

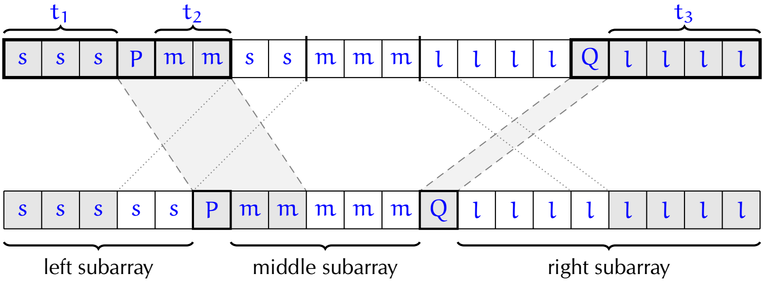


Figure : First row: State of the array just after partitioning the ordinary elements. The letters indicate whether the element at this location is smaller (s), between (m) or larger (l) than the two pivots P and Q. Sample elements are shaded. **Second row:** State of the array after pivots and sample parts have been moved to their partition. The "rubber bands" indicate moved regions of the array.

Pivot Sampling

Randomness preservation:

- As the sample was sorted, the left and middle subarrays have sorted prefixes of length t_1 and t_2 followed by a random permutation of the remaining elements. Similarly, the right subarray has a sorted suffix of t_3 elements. Hence, except for the trivial case $t = 0$, these subarrays are **not** randomly ordered!
- **Vital observation:** sorted part **always** lies completely inside the sample range for the next partitioning phase $\rightsquigarrow$ non-randomness only affects sorting of the sample, it does **not affect partitioning**.

Pivot Sampling

Randomness preservation:

- As the sample was sorted, the left and middle subarrays have sorted prefixes of length t_1 and t_2 followed by a random permutation of the remaining elements. Similarly, the right subarray has a sorted suffix of t_3 elements. Hence, except for the trivial case $\mathbf{t} = 0$, these subarrays are **not** randomly ordered!
- **Vital observation:** sorted part **always** lies completely inside the sample range for the next partitioning phase $\rightsquigarrow$ non-randomness only affects sorting of the sample, it does **not affect partitioning**.

Pivot Sampling

Furthermore:

- For our special case of a fully sorted prefix or suffix of length $s \geqslant 1$ and a fully random rest, we can simply use InsertionSort where the first s iterations of the outer loop are skipped. Our InsertionSort implementations then simply accept s as an additional parameter.
- We precisely **quantify** the savings resulting from skipping the first s iterations: Apart from per-call overhead, we save exactly what it would have costed us to sort this prefix/suffix with InsertionSort.

Analysis

- We assume the **i. i. d. uniform model**, i. e. the array is initially filled with n i. i. d. uniformly in $(0, 1)$ distributed random variables $U_1, \dots, U_n$.

- Then, we choose the **first k_l and last k_r elements as the sample** $\mathbf{V} = (U_1, \dots, U_{k_l}, U_{n-k_r+1}, \dots, U_n)$, from which the pivots $P := V_{(t_1+1)}$ and $Q := V_{(t_1+t_2+2)}$ are selected.

- For D the **spacings** induced by P and Q on the unit interval $[0, 1]$:

$$\mathbf{D} := (D_1, D_2, D_3) \quad := \quad (P,\ Q - P,\ 1 - Q)\ .$$

  By definition of our pivot sampling method, (D_1, D_2, D_3) are the spacings induced by two order statistics $V_{(t_1+1)}$ and $V_{(t_1+t_2+2)}$ of k i. i. d. uniform random variables $V_1, \dots, V_n$, so $\mathbf{D} = (\mathbf{D_1}, \mathbf{D_2}, \mathbf{D_3})$ **is Dirichlet $\mathbf{Dir}(t_1 + 1, t_2 + 1, t_3 + 1)$ distributed**.

Analysis

P and Q (equivalently spacings $\mathbf{D}$) $\rightsquigarrow$ **probability** for an ordinary element U **to be small, medium or large**, respectively:

- $U \in (0, P) \rightsquigarrow$ small (with probability D_1);
- $U \in (P, Q) \rightsquigarrow$ medium (with probability D_2;
- $U \in (Q, 1) \rightsquigarrow$ large (with probability D_3;

Also note that the event of equal keys has probability 0.

Partition sizes: result of $n - k$ independent repetitions of this experiment, so $\mathbf{I} = (I_1, I_2, I_3)$ (number of small, medium resp. large elements) **is multinomially $\mathbf{Mult}(n - k; D_1, D_2, D_3)$ distributed**.

Note that the subproblem sizes $\mathbf{J} = (J_1, J_2, J_3)$ including the sampled-out elements are completely determined by $\mathbf{I}$ via $\mathbf{J} = \mathbf{I} + \mathbf{t}$.

Analysis

By this process, the first partitioning phase only determines

- values (of pivots);
- ranks (of pivots);
- subproblem size.

About none of the other elements is known more than into which subproblem it belongs $\rightsquigarrow$ repeat this same process with the same distribution for subproblems on their **respective subinterval** of $(0, 1)$.

Analysis

Denoting by T_n the costs of the first partitioning step, we obtain the following **distributional recurrence** for the family $(C_n)_{n \in \mathbb{N}}$ of random variables:

$$C_n \stackrel{\mathcal{D}}{=} \begin{cases} T_n + C_{J_1} + C'_{J_2} + C''_{J_3}, & \text{for } n > w; \\ W_n, & \text{for } n \leqslant w. \end{cases} \tag{1}$$

Here W_n denotes the cost of InsertionSorting a random permutation of size n, $(C'_j)_{j \in \mathbb{N}}$ and $(C''_j)_{j \in \mathbb{N}}$ are independent copies of $(C_j)_{j \in \mathbb{N}}$ (identically distributed, totally independent, independent of T_n).

Analysis

Caution: Before recursion **not 100% accurate**: The savings for InsertionSort on already sorted parts of the sample are not considered!

However,

- for most interesting cost measures, the **resulting savings only depend on the length s** of this sorted part, not on the length of the whole array;
- denoting these savings by E_s, we pay E_{t_1} less for calls to left subarrays, E_{t_2} less for middle calls and E_{t_3} less for right subarrays;
- **discounting the future savings** $E_t := E_{t_1} + E_{t_2} + E_{t_3}$ of all three recursive calls directly in the **current call**, we can the total costs in the form given above, with a **reduced toll function** $\tilde{T}_n$.

Analysis

Taking expectations on both sides in (1), we find a recurrence relation for the **expected** costs $\mathbb{E}[C_n]$:

$$\mathbb{E}[C_n] = \begin{cases} \mathbb{E}[T_n] + \sum\limits_{\substack{\mathbf{j}=(j_1,j_2,j_3) \\ j_1+j_2+j_3=n-2}} \mathbb{P}(\mathbf{J}=\mathbf{j})\big(\mathbb{E}[C_{j_1}] + \mathbb{E}[C_{j_2}] + \mathbb{E}[C_{j_3}]\big), & \text{for } n > w; \\ \mathbb{E}[W_n], & \text{for } n \leqslant w. \end{cases} \tag{2}$$

The distribution of $\mathbf{J}$ has been given above; using well-known fact on multinomial distribution we obtain:

$$\mathbb{P}(\mathbf{J}=\mathbf{j}) = \frac{\binom{j_1}{t_1}\binom{j_2}{t_2}\binom{j_3}{t_3}}{\binom{n}{k}}.$$

Solving the recurrence

Theorem (Martínez and Roura 2001)

Let F_n *be recursively defined by*

$$F_n \quad = \quad \begin{cases} b_n, & \text{for } 0 \leqslant n < N; \\ t_n + \displaystyle\sum_{j=0}^{n-1} w_{n,j}\, F_j, & \text{for } n \geqslant N \end{cases} \tag{3}$$

where the toll function satisfies $t_n \sim K n^{\alpha} \log^{\beta} n$ *as* $n \to \infty$ *for constants* K, $\alpha \geqslant 0$ *and* $\beta > -1$. *Assume there exists a function* $w : [0,1] \to \mathbb{R}$, *such that*

$$\sum_{j=0}^{n-1} \left| w_{n,j} - \int_{j/n}^{(j+1)/n} w(z)\, dz \right| \quad = \quad O(n^{-d}) \tag{4}$$

for a constant $d > 0$. *With* $H := 1 - \displaystyle\int_0^1 z^{\alpha} w(z)\, dz$, *we have the following cases:*

1. *If* $H > 0$, *then* $F_n \sim \dfrac{t_n}{H}$.
2. *If* $H = 0$, *then* $F_n \sim \dfrac{t_n \ln n}{\tilde{H}}$ *with* $\tilde{H} = -(\beta + 1) \displaystyle\int_0^1 z^{\alpha} \ln z\, w(z)\, dz$.
3. *If* $H < 0$, *then* $F_n \sim \Theta(n^c)$ *for the unique* $c \in \mathbb{R}$ *with* $\displaystyle\int_0^1 z^c w(z)\, dz = 1$. $\square$

Solving the recurrence

Recurrence in the form of (2): We start again with the probabilistic equation above and condition the terms C_{J_1}, C_{J_2} and C_{J_3} on $\mathbf{J}$. For $n > w$, this gives

$$C_n = T_n \; + \sum_{l=1}^{3} \sum_{j=0}^{n-2} \mathbb{1}_{\{J_l = j\}} C_j \; .$$

Taking expectations on both sides and exploiting independence yields

$$\begin{aligned} \mathbb{E}\, C_n &= \mathbb{E}\, T_n \; + \sum_{l=1}^{3} \sum_{j=0}^{n-2} \mathbb{E}[\mathbb{1}_{\{J_l = j\}}]\, \mathbb{E}[C_j] \\ &= \mathbb{E}\, T_n \; + \sum_{j=0}^{n-2} \big(\mathbb{P}(J_1 = j) + \mathbb{P}(J_2 = j) + \mathbb{P}(J_3 = j)\big)\, \mathbb{E}\, C_j \, , \end{aligned}$$

which is a recurrence in CMT style with weights

$$w_{n,j} = \mathbb{P}(J_1 = j) + \mathbb{P}(J_2 = j) + \mathbb{P}(J_3 = j) \; .$$

Solving the recurrence

Note that

- the probabilities $\mathbb{P}(J_l = j)$ implicitly depend on n;
- $\mathbb{P}(J_l = j) = \mathbb{P}(I_l = j - t_l)$ for $l = 1, 2, 3$, can be computed using that the marginal distribution of I_l is $\mathrm{Bin}(n-k, D_l)$,
- yielding $\mathbb{P}(I_l = i) = \binom{N}{i}\frac{(t_l+1)^{\overline{i}}(k-t_l)^{\overline{N-i}}}{(k+1)^{\overline{N}}}$.

Shape function according to (3): With

$$w(z) = \sum_{l=1}^{3}(k-t_l)\binom{k}{t_l}z^{t_l}(1-z)^{k-t_l-1}$$

we find $\sum_{j=0}^{n-1}\left|w_{n,j} - \int_{j/n}^{(j+1)/n} w(z)\,dz\right| = O(n^{-1})$ and CMT applies (case 2) with $\alpha = 1$, $\beta = 0$ and $K = a$.

Solving the recurrence

Note that

- the probabilities $\mathbb{P}(J_l = j)$ implicitly depend on n;
- $\mathbb{P}(J_l = j) = \mathbb{P}(I_l = j - t_l)$ for $l = 1, 2, 3$, can be computed using that the marginal distribution of I_l is $\mathrm{Bin}(n - k, D_l)$,
- yielding $\mathbb{P}(I_l = i) = \binom{N}{i} \frac{(t_l+1)^{\overline{i}}(k-t_l)^{\overline{N-i}}}{(k+1)^{\overline{N}}}$.

Shape function according to (3): With

$$w(z) = \sum_{l=1}^{3} (k - t_l) \binom{k}{t_l} z^{t_l} (1 - z)^{k-t_l-1}$$

we find $\sum_{j=0}^{n-1} \left| w_{n,j} - \int_{j/n}^{(j+1)/n} w(z)\, dz \right| = O(n^{-1})$ and CMT applies (case 2) with $\alpha = 1$, $\beta = 0$ and $K = a$.

Solving the recurrence

This way we find:

Theorem

Let $\mathbb{E}[C_n]$ *be a sequence of numbers satisfying recurrence (2) for some constant* $w \geqslant k$ *and let the toll function* $\mathbb{E}[T_n]$ *be of the form* $\mathbb{E}[T_n] = an + O(1)$ *for a constant* a. *Then*

$$\mathbb{E}[C_n] \;=\; \boldsymbol{a} \cdot g(k, t_1, t_2, t_3) \cdot n \ln n \;+\; O(n)\,,$$

where g *is given by*

$$g(k, t_1, t_2, t_3) \;=\; \left(-\sum_{i=1}^{3} \frac{t_i + 1}{k + 1}(\mathcal{H}_{t_i+1} - \mathcal{H}_{k+1})\right)^{-1}.$$

$\Rightarrow$ results for number of comparisons, swaps and executed Java bytecodes (leading term independent of w).

Optimal Pivot Ranks

Challenge: Hard to separate optimal pivot ranks from optimal sample size.

Resort: Consider family of algorithms with $(k^{(j)})_{j\in\mathbb{N}}$, and $(t_i^{(j)})_{j\in\mathbb{N}}$ for $i = 1, 2, 3$ a sequences of non-negative integers which fulfill $k^{(j)} = t_1^{(j)} + t_2^{(j)} + t_3^{(j)}$ for every $j \in \mathbb{N}$. Moreover, assume $k^{(j)} \to \infty$ and $t_i^{(j)}/k^{(j)} \to \tau_i$ with $\tau_i \in [0, 1]$ for $i = 1, 2, 3$ as $j \to \infty$. Note that by definition we have $\tau_1 + \tau_2 + \tau_3 = 1$.

For each $j \in \mathbb{N}$, we can apply our findings for the expected number of comparisons, swaps and bytecodes respectively using parameters $k^{(j)}$ and $t^{(j)} \leadsto$ limiting behaviour of costs.

Optimal Pivot Ranks

Challenge: Hard to separate optimal pivot ranks from optimal sample size.

Resort: Consider family of algorithms with $(k^{(j)})_{j\in\mathbb{N}}$, and $(t_i^{(j)})_{j\in\mathbb{N}}$ for $i = 1, 2, 3$ a sequences of non-negative integers which fulfill $k^{(j)} = t_1^{(j)} + t_2^{(j)} + t_3^{(j)}$ for every $j \in \mathbb{N}$. Moreover, assume $k^{(j)} \to \infty$ and $t_i^{(j)}/k^{(j)} \to \tau_i$ with $\tau_i \in [0, 1]$ for $i = 1, 2, 3$ as $j \to \infty$. Note that by definition we have $\tau_1 + \tau_2 + \tau_3 = 1$.

For each $j \in \mathbb{N}$, we can apply our findings for the expected number of comparisons, swaps and bytecodes respectively using parameters $k^{(j)}$ and $\mathbf{t}^{(j)} \rightsquigarrow$ limiting behaviour of costs.

Optimal Pivot Ranks

We find that the overall number of comparisons, swaps resp. bytecodes converge to

$$\frac{a_C^*}{-\sum_{i=1}^{3} \tau_i \ln(\tau_i)}, \qquad \frac{a_S^*}{-\sum_{i=1}^{3} \tau_i \ln(\tau_i)} \quad \text{resp.} \quad \frac{a_{BC}^*}{-\sum_{i=1}^{3} \tau_i \ln(\tau_i)}.$$

with

$$\begin{aligned}
a_C^{(j)} &\to a_C^* := 1 + \tau_1 + \tau_2 + (\tau_1 + \tau_2)(\tau_3 - \tau_1) \\
a_S^{(j)} &\to a_S^* := \tau_1 + (\tau_1 + \tau_2)\tau_3 \\
a_{BC}^{(j)} &\to a_{BC}^* := 10 + 13\tau_1 + 5\tau_2 + 11(\tau_1 + \tau_2)\tau_3 + \tau_1(\tau_1 + \tau_2)
\end{aligned}$$

the **"constants"** showing up in before theorem.

Optimal Pivot Ranks

Optimal choices: The number of **comparisons** is minimized for

$$\tau_C^* \approx (0.428846, 0.268774, 0.302380) .$$

For this choice, the expected number of comparisons used is asymptotically $1.4931 n \ln n$. The minimal asymptotic number of executed **bytecodes** of roughly $16.3833 n \ln n$ is obtained for

$$\tau_{BC}^* \approx (0.206772, 0.348562, 0.444666) .$$

For **swaps no minimum** is attained in the open simplex; the corresponding coefficient approaches 0 as τ_1 and τ_2 simultaneously go to 0.

Note that

- the optimal choices heavily differ depending on the employed cost measure;
- the minima differ significantly from the symmetric choice $\tau = (\frac{1}{3}, \frac{1}{3}, \frac{1}{3})$.

Optimal Pivot Ranks

Optimal choices: The number of **comparisons** is minimized for

$$\tau_C^* \approx (0.428846, 0.268774, 0.302380) .$$

For this choice, the expected number of comparisons used is asymptotically $1.4931n \ln n$. The minimal asymptotic number of executed **bytecodes** of roughly $16.3833n \ln n$ is obtained for

$$\tau_{BC}^* \approx (0.206772, 0.348562, 0.444666) .$$

For **swaps no minimum** is attained in the open simplex; the corresponding coefficient approaches 0 as τ_1 and τ_2 simultaneously go to 0.

Note that

- the optimal choices heavily differ depending on the employed cost measure;
- the minima differ significantly from the symmetric choice $\tau = (\frac{1}{3}, \frac{1}{3}, \frac{1}{3})$.

Optimal Pivot Ranks

Optimal choices: The number of **comparisons** is minimized for

$$\tau_C^* \approx (0.428846, 0.268774, 0.302380)\ .$$

For this choice, the expected number of comparisons used is asymptotically $1.4931 n \ln n$. The minimal asymptotic number of executed **bytecodes** of roughly $16.3833 n \ln n$ is obtained for

$$\tau_{BC}^* \approx (0.206772, 0.348562, 0.444666)\ .$$

For **swaps no minimum** is attained in the open simplex; the corresponding coefficient approaches 0 as τ_1 and τ_2 simultaneously go to 0.

Note that

- the optimal choices heavily differ depending on the employed cost measure;
- the minima differ significantly from the symmetric choice $\tau = (\frac{1}{3}, \frac{1}{3}, \frac{1}{3})$.

Optimal Pivot Ranks

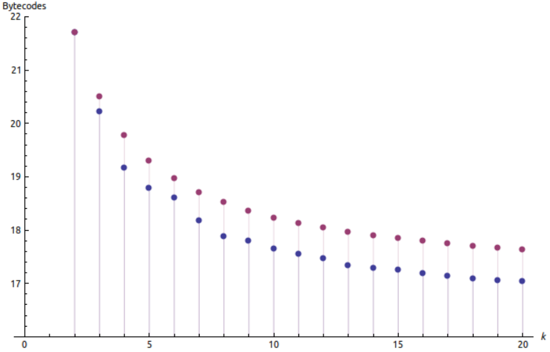


Figure : The leading term coefficient of the expected number of bytecodes used by generalized Yaroslavskiy for different sample sizes k (x-axis). Blue points show the optimal order statistics, purple points given the cost when choosing the tertiles of the sample.

Outlook and Conclusion

- We also have results for $k = 5$ and corresponding lower order terms
- dealing with comparison (also in InsertionSort and SampleSort), swaps and write accesses;
- there w come into play.

Thus, Java 7th quicksort is a **perfect textbook example** to demonstrate

- how well methods from AofA are developed;
- the depth of results obtainable (precise expectations, distributions, covariances, ...) by those methods;
- how AofA can guide engineering of an algorithm (pivot sampling, switch to `insertionsort`, ...).

However, our **sophisticated machinery fails to explain** the practical efficiency of Yaroslavskiy's algorithms (presumably) because of a lacking access to

- branch mispredictions and
- cache misses.

Outlook and Conclusion

- We also have results for $k = 5$ and corresponding lower order terms
- dealing with comparison (also in InsertionSort and SampleSort), swaps and write accesses;
- there w come into play.

Thus, Java 7th quicksort is a **perfect textbook example** to demonstrate

- how well methods from AofA are developed;
- the depth of results obtainable (precise expectations, distributions, covariances, ...) by those methods;
- how AofA can guide engineering of an algorithm (pivot sampling, switch to `insertionsort`, ...).

However, our **sophisticated machinery fails to explain** the practical efficiency of Yaroslavskiy's algorithms (presumably) because of a lacking access to

- branch mispredictions and
- cache misses.

Outlook and Conclusion

- We also have results for $k = 5$ and corresponding lower order terms
- dealing with comparison (also in InsertionSort and SampleSort), swaps and write accesses;
- there w come into play.

Thus, Java 7th quicksort is a **perfect textbook example** to demonstrate

- how well methods from AofA are developed;
- the depth of results obtainable (precise expectations, distributions, covariances, ...) by those methods;
- how AofA can guide engineering of an algorithm (pivot sampling, switch to `insertionsort`, ...).

However, our **sophisticated machinery fails to explain** the practical efficiency of Yaroslavskiy's algorithms (presumably) because of a lacking access to

- branch mispredictions and
- cache misses.

Thank you very much for your attention!