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
Many inventions by algorithms community vs. Few methods successful in practice

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

QuickSort
+ Mergesort variant as stable sort

Sorting methods listed on Wikipedia

Sorting methods of standard libraries for random access data
Many inventions by algorithms community vs. Few methods successful in practice

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

Quicksort + Mergesort variant as stable sort

Sorted methods listed on Wikipedia

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**: detailed analysis of many optimizations
- **1993 Bentley, McIlroy**: *Engineering a Sort Function*
- **1997 Musser**: $\Theta(n \log n)$ worst case by bounded recursion depth

Basic algorithm settled since 1961; latest tweaks from 1990’s. Since then: Almost identical in all programming libraries!
History of Quicksort in Practice

- **1961, 1962 Hoare**: first publication, average case analysis
- **1969 Singleton**: median-of-three & Insertionsort on small subarrays
- **1975-78 Sedgewick**: detailed analysis of many optimizations
- **1993 Bentley, McIlroy**: *Engineering a Sort Function*
- **1997 Musser**: $\Theta(n \log n)$ worst case by bounded recursion depth

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

**Since then**: Almost identical in all programming libraries!

Since then: Almost identical in all programming libraries!
1961, 62 Hoare: first publication, average case analysis

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

1975-78 Sedgewick: detailed analysis of many optimizations

1993 Bentley, McIlroy: *Engineering a Sort Function*

1997 Musser: $\Theta(n \log n)$ worst case by bounded recursion depth

~ 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 ~ July 2011 **public release** of Java 7 with Yaroslavskiy’s Quicksort.
Why switch to new, unknown algorithm?

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

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

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

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

Normalized Java runtimes (in ms). Average and standard deviation of 1000 random permutations per size.
High Level Algorithm:

1. Partition array around two pivots $p \leq q$.
2. Sort 3 subarrays recursively.

How to do partitioning?
High Level Algorithm:

1. Partition array around two pivots $p \leq q$.
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

Markus E. Nebel  
Java 7’s Dual Pivot Quicksort  
2013/12/05  
5 / 43
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
- \( r = 2 \rightarrow \text{same} \) #comparisons as classic Quicksort in one partitioning step: \( \frac{5}{3} \) comparisons per element
- \( r > 2 \rightarrow \) 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.*
In this talk:

- We explain, why the new QS variant can be beneficial 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[]))

Select two elements as pivots.

Invariant: $< \ p \ l \ p \leq \ o \leq \ q \ k \ ? \ g \ > \ q$
Java 7’s Dual Pivot Quicksort – Example

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

Only value relative to pivot counts.

Invariant:

\[
\begin{align*}
\text{Invariant:} & \quad \begin{array}{|c|c|c|c|c|c|c|c|}
\hline
\text{< p} & \text{<} & \text{p < o < q} & \text{<} & \text{?} & \text{>} & \text{q} & \text{>}
\hline
\end{array}
\end{align*}
\]
Yaroslavskiy’s Dual Pivot Quicksort (used in Oracle’s Java 7 Arrays.sort(int[]))

A[k] is medium \(\leadsto\) go on

Invariant:

\[
\begin{array}{cccccccc}
< p & l & p \leq o \leq q & k & ? & g & > q \\
\rightarrow & & \rightarrow & & & & \\
\end{array}
\]
Yaroslavskiy’s Dual Pivot Quicksort
(used in Oracle’s Java 7 Arrays.sort(int[]))

\[ A[k] \text{ is small} \rightarrow \text{Swap to left} \]

Invariant:

\[ \begin{array}{cccccccc}
\ell & k & \ell & p & q & k & ? & g & q \\
< & < & \leq & \leq & \leq & < & ? & > & < \\
\rightarrow & \rightarrow & \rightarrow & \leftarrow & \leftarrow & \leftarrow & \leftarrow & \leftarrow & \leftarrow \\
\end{array} \]
Yaroslavskiy’s Dual Pivot Quicksort (used in Oracle’s Java 7 Arrays.sort(int[]))

Invariant: \[ < p \quad l \quad p \leq o \leq q \quad k \quad ? \quad g > q \]

Swap small element to left end.
Yaroslavskiy’s Dual Pivot Quicksort
(used in Oracle’s Java 7 Arrays.sort(int[]))

Swap small element to left end.

Invariant: \[
\begin{array}{ccccccc}
< & p & l & p \leq o \leq q & k & ? & g & > & q
\end{array}
\]
Java 7’s Dual Pivot Quicksort – Example

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

\[ \begin{align*}
\ell & \quad k \\
3 & \quad 1 & 5 & 8 & 4 & 7 & 2 & 9 & 6
\end{align*} \]

\[ A[k] \text{ is large} \sim \text{Find swap partner.} \]

Invariant: \[ \begin{align*}
< p & \quad \ell & p \leq o \leq q & k & ? & g & > q \\
\rightarrow & & \rightarrow & & \leftarrow
\end{align*} \]
Yaroslavskiys’s Dual Pivot Quicksort
(used in Oracle’s Java 7 Arrays.sort(int[]))

A[k] is large \implies \text{Find swap partner:}
g skips over large elements.

Invariant: 
\begin{align*}
&< p \quad l \quad p \leq o \leq q \quad k \quad ? \quad g \quad > q \\
&\rightarrow \quad \rightarrow \quad \leftarrow
\end{align*}
Yaroslavskiy’s Dual Pivot Quicksort (used in Oracle’s Java 7 Arrays.sort(int[]))

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

Invariant: $< p \leq l \leq o \leq q \leq k ? g > q$
Java 7’s Dual Pivot Quicksort – Example

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

\[ A[k] \text{ is large} \implies \text{Swap} \]

\[
\begin{array}{cccccccc}
3 & 1 & 5 & 2 & 4 & 7 & 8 & 9 & 6 \\
& & & & k & & & & \\
& & & & \downarrow & & & & \\
& & & & l & & & & \\
\end{array}
\]

Invariant: \[< p \quad l \quad p \leq o \leq q \quad k \quad ? \quad g \quad > q\]
Yaroslavskiyy’s Dual Pivot Quicksort
(used in Oracle’s Java 7 Arrays.sort(int[]))

\[ \ell \quad k \quad g \]

3 1 5 2 4 7 8 9 6

\[ A[k] \text{ is old } A[g], \text{ small } \sim \text{ Swap to left} \]

Invariant:
\[
\begin{array}{c}
< p \\
\ell \\
p \leq o \leq q \\
k \\
? \\
g > q
\end{array}
\]
Java 7’s Dual Pivot Quicksort – Example

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

Invariant:

\[< p \quad \ell \quad p \leq \circ \leq q \quad k \quad ? \quad g \quad > q\]
Yaroslavskiy’s Dual Pivot Quicksort
(used in Oracle’s Java 7 Arrays.sort(int[]))

A[k] is medium \(\Rightarrow\) go on

Invariant:

\(\begin{array}{c}
\text{\(\ell\)}, \text{\(k\)}, \text{\(g\)}\\
1, 2, 5, 4, 7, 8, 9, 6
\end{array}\)
Yaroslavskiy’s Dual Pivot Quicksort
(used in Oracle’s Java 7 Arrays.sort(int[]))

A[k] is large ~ Find swap partner.

Invariant: < p l p ≤ o ≤ q k ? g > q
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: \( \langle p \; \ell \; p \leq o \leq q \; k \; ? \; g \; > q \)
Java 7’s Dual Pivot Quicksort – Example

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

3 1 2 5 4 7 8 9 6

l g k

Invariant:
< p l p ≤ o ≤ q k ? g > q

g and k have crossed!
Swap pivots in place
Yaroslavskiy’s Dual Pivot Quicksort
(used in Oracle’s Java 7 Arrays.sort(int[]))

Invariant:

$\ell \leq p \leq q \leq g \leq k$

$g$ and $k$ have crossed!
Swap pivots in place
Java 7’s Dual Pivot Quicksort – Example

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

Partitioning done!

Invariant: $< p \rightarrow l \rightarrow p \leq o \leq q \rightarrow k \rightarrow ? \rightarrow g \rightarrow > q$
Yaroslavskiy’s Dual Pivot Quicksort
(used in Oracle’s Java 7 Arrays.sort(int[]))

Recursively sort three sublists.

Invariant: \(< p \leq o \leq q \rightarrow k \rightarrow g > q\)
Java 7’s Dual Pivot Quicksort – Example

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

Invariant:

```
< p  l  p ≤ o ≤ q  k  ?  g  > q
```

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

- Assume, we first compare $x$ with $p$.
  - small elements need 1, others 2 comparisons

- on average: $\frac{1}{3}$ of all elements are small
  - $\frac{1}{3} \cdot 1 + \frac{2}{3} \cdot 2 = \frac{5}{3}$ 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.

- Any partitioning method needs at least
  - $\frac{5}{3} (n - 2) \sim \frac{20}{12} n$ comparisons on average?

- No!
Beating the “Lower Bound”

~ $\frac{20}{12} n$ 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 $C_1$ and $C_2$ s.t.

- $C_1$ first compares with $p$.
- $C_2$ first compares with $q$.

$\sim C_1$ executed often, iff $p$ is large.
$C_2$ executed often, iff $q$ is small.

$\sim C_1$ executed often
iff many small elements
iff good chance that $C_1$ needs only one comparison
($C_2$ similar)

$\sim$ less comparisons than $\frac{5}{3}$ per elements on average
Yaroslavskiy’s Quicksort

\begin{verbatim}
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        end if
12      end if
13      k := k + 1
14    end while
15    ℓ := ℓ − 1;  g := g + 1
16    Swap A[left] and A[ℓ];  Swap A[right] and A[g]
17  DUALPIVOTQUICKSORTYAROSLAVSKIY(A, left, ℓ − 1)
18  DUALPIVOTQUICKSORTYAROSLAVSKIY(A, ℓ + 1, g − 1)
19  DUALPIVOTQUICKSORTYAROSLAVSKIY(A, g + 1, right)
20  end if
\end{verbatim}
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      Ck 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  k := k + 1
15  end while
16  ℓ := ℓ - 1;  g := g + 1
17  Swap A[left] and A[ℓ];  Swap A[right] and A[g]
18  DUALPIVOTQUICKSORTYAROSLAVSKIY(A, left, ℓ - 1)
19  DUALPIVOTQUICKSORTYAROSLAVSKIY(A, ℓ + 1, g - 1)
20  DUALPIVOTQUICKSORTYAROSLAVSKIY(A, g + 1, right)
21  end if

- 2 comparison locations
- C_k handles pointer k
- C_{g} handles pointer g
- C_k first checks < p
- C_k if needed ≥ 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

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

- \( C_n \) satisfies recurrence relation

\[
C_n = c_n + \frac{2}{n(n-1)} \sum_{1\leq p<q\leq n} (C_{p-1} + C_{q-p-1} + C_{n-q}),
\]

with \( c_n \) expected \#comparisons in first partitioning step

- recurrence solvable by standard methods

\[\sim \quad \text{linear } c_n \sim a \cdot n \quad \text{yields } C_n \sim \frac{6}{5} a \cdot n \ln n.\]

- \( \sim \) need to compute \( c_n \)
Analysis of Yaroslavskiy’s Algorithm

- **first** comparison for **all** elements (at $C_k$ or $C_g$)
  $\sim \sim n$ 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**
  $\sim \sim \frac{1}{3} n$ comparisons in expectation

- $\sim$ it remains to count:
  - **large** elements reached by $k$ and
  - **small** elements reached by $g$. 
**Second** comparisons for **small** and **large** elements? Depends on **location**!

\[ C'_k \sim l@K \]: number of **large** elements at positions \( K \).

\[ C'_g \sim s@G \]: number of **small** elements at positions \( G \).

**Recall invariant:**

\[
\begin{array}{cccccc}
< p & \ell & p \leq \circ \leq q & k & ? & g > q \\
\rightarrow & \rightarrow & \leftarrow
\end{array}
\]

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

\[ l@K = 3 \quad s@G = 2 \]

\( \text{positions } K = \{2, \ldots, q-1\} \quad G = \{q, \ldots, n-1\} \)

**for given** \( p \) **and** \( q \), \( l@K \) **hypergeometrically** distributed

\[ E[l@K|p,q] = (n-q)\frac{q-2}{n-2} \]
Analysis of Yaroslavskiy’s Algorithm

- law of total expectation:
\[
E[l @ K] = \sum_{1 \leq p < q \leq n} Pr[pivots (p, q)] \cdot (n - q) \frac{q-2}{n-2} \sim \frac{1}{6} n
\]

- Similarly: \( E[s @ G] \sim \frac{1}{12} n \).

- Summing up contributions:
\[
c_n \sim n + \frac{1}{3} n + \frac{1}{6} n + \frac{1}{12} n
\]
\[
= \frac{19}{12} n
\]

Recall: “lower bound” was \( \frac{20}{12} n \).
Lower Bound on Comparisons

- How clever can dual pivot partitioning be?

- For lower bound, assume
  - random permutation model
  - pivots are selected uniformly
  - an oracle tells us, whether more small or more large elements occur

- $\sim 1$ comparison for frequent extreme elements
  - $2$ comparisons for middle and rare extreme elements

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

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

- Even with unrealistic oracle, not much better than Yaroslavskiy
Gathering Results

- **Comparisons:**
  - Yaroslavskiy needs $\sim \frac{6}{5} \cdot \frac{19}{12} n \ln n = 1.9 n \ln n$ on average.
  - Classic Quicksort needs $\sim 2 n \ln n$ comparisons!

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

- **Swaps:**
  - $\sim 0.6 n \ln n$ swaps for Yaroslavskiy’s algorithm vs.
  - $\sim 0.3 n \ln n$ swaps for classic Quicksort
Gathering Results

Comparisons:

- Yaroslavskiy needs $\sim \frac{6}{5} \cdot \frac{19}{12} \cdot n \ln n = 1.9 \cdot n \ln n$ on average.
- Classic Quicksort needs $\sim 2 \cdot n \ln n$ comparisons!

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

Swaps:

- $\sim 0.6 \cdot n \ln n$ swaps for Yaroslavskiy’s algorithm vs.
- $\sim 0.3 \cdot n \ln n$ swaps for classic Quicksort
Analogous to classic Quicksort

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

- switch to InsertionSort for subproblems of size $\leq w$,
- 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

1  \text{bc: 3}
   \text{k} \leq \text{g}
   \text{yes}

2  \text{bc: 7}
   \text{t} := \text{A}[k];
   \text{t} < \text{p}
   \text{no}
   \text{yes}

3  \text{bc: 12}
   \text{A[k]} := \text{A}[\ell];
   \text{A[\ell]} := \text{t};
   \text{\ell} := \ell + 1;

4  \text{bc: 3}
   \text{t} \geq \text{q}
   \text{no}
   \text{yes}

5  \text{bc: 5}
   \text{A[g]} > \text{q}
   \text{no}
   \text{yes}

6  \text{bc: 3}
   \text{k} < \text{g}
   \text{no}
   \text{no}

7  \text{bc: 2}
   \text{g} := \text{g} - 1;

8  \text{bc: 5}
   \text{A[g]} < \text{p}
   \text{no}
   \text{no}

9  \text{bc: 14}
   \text{A[k]} := \text{A}[\ell];
   \text{A[\ell]} := \text{A}[g]
   \text{\ell} := \ell + 1;

10 \text{bc: 6}
    \text{A[k]} := \text{A}[g]

11 \text{bc: 5}
    \text{A[g]} := \text{t};
    \text{g} := \text{g} - 1;

12 \text{bc: 2}
    \text{k} := \text{k} + 1

Markus E. Nebel
Java 7's Dual Pivot Quicksort
2013/12/05 19 / 43
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

\[ A[k]: \text{large} \]
\[ A[g]: \text{large} \]
\[ \Delta(g - k): 1 \]

Bytecode
Instructions: 10
Control Flow Graph of Partitioning Loop

Cycle 4

A[k]: large
A[g]: small

\(\Delta(g - k): 2\)

Bytecode
Instructions: 44
Cycle 5

A[k]: large
A[g]: medium

Δ(g − k): 2

Bytecode Instructions: 36
Algorithm is asymmetric:
- cycles have different cost
- \( \sim \) would rather execute cheap ones often
- cycles chosen by classes small, medium or large
- probability for classes depends on pivot values

\( \sim \) Maybe we can “influence pivot values accordingly”?
Pivot Sampling

- Well-known optimization for **classic** Quicksort: **median-of-three**
  \(\rightarrow\) pivot closer to **median** of whole list

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

  ![Diagram showing tertiles-of-five]

  **tertiles-of-five**
  \(\rightarrow\) pivots closer to **tertiles** of whole list

- 9 other possibilities to pick \(p\) and \(q\) out of 5 elements:
Well-known optimization for classic Quicksort: median-of-three

pivot closer to median of whole list

In JRE7 Quicksort implementation: natural extension for 2 pivots:

tertiles-of-five

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
  - pivot closer to median of whole list

- In JRE7 Quicksort implementation: natural extension for 2 pivots:
  - tertiles-of-five
  - 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**
  \( \leadsto \) pivot closer to **median** of whole list

- In **JRE7 Quicksort implementation**: natural extension for 2 pivots:
  \[ \begin{array}{cccccc}
  & * & * & * & * & * \\
  \end{array} \]

\[ \begin{array}{ccc}
  p & q \\
  \end{array} \]

**tertiles-of-five**
\[ \begin{array}{cccc}
  * & * & * & * \\
  \end{array} \]
\( \leadsto \) pivots closer to **tertiles** of whole list

- **9** other possibilities to pick \( p \) and \( q \) out of 5 elements:
  \[ \begin{array}{cccccc}
  * & * & * & * & * \\
  \end{array} \]
  \[ \begin{array}{cccccc}
  * & * & * & * & * \\
  \end{array} \]
  \[ \begin{array}{cccccc}
  * & * & * & * & * \\
  \end{array} \]
  \[ \begin{array}{cccccc}
  * & * & * & * & * \\
  \end{array} \]
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:

#### Pivot Selection Scheme

<table>
<thead>
<tr>
<th>Scheme</th>
<th>A(^1)</th>
<th>B(^2)</th>
</tr>
</thead>
<tbody>
<tr>
<td>JRE7</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>+5.14%</td>
<td>+0.80%</td>
</tr>
<tr>
<td>JRE7((1,3))</td>
<td>−1.85%</td>
<td>−0.44%</td>
</tr>
<tr>
<td></td>
<td>+3.34%</td>
<td>−0.42%</td>
</tr>
<tr>
<td></td>
<td>(stack overflow!)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>+2.48%</td>
<td>+2.73%</td>
</tr>
<tr>
<td></td>
<td>+11.3%</td>
<td>+3.31%</td>
</tr>
<tr>
<td></td>
<td>+12.7%</td>
<td>+3.29%</td>
</tr>
<tr>
<td></td>
<td>+16.4%</td>
<td>+2.48%</td>
</tr>
<tr>
<td></td>
<td>+39.0%</td>
<td>+5.87%</td>
</tr>
</tbody>
</table>

\(^1\)Average number of executed bytecodes on almost sorted lists of length \(10^5\).

\(^2\)Average running time on random permutations of length \(10^6\).
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 \( t = (3, 2, 4) \). 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 \( \rightarrow \) non-randomness only affects sorting of the sample, it does not affect partitioning.
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 $\sim$ non-randomness only affects sorting of the sample, it does not affect partitioning.
Furthermore:

- For our special case of a fully sorted prefix or suffix of length \( s \geq 1 \) 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.
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 \( U_1, \ldots, U_n \).

Then, we choose the **first \( k_1 \) and last \( k_r \) elements as the sample** \( V = (U_1, \ldots, U_{k_1}, U_{n-k_r+1}, \ldots, U_n) \), 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 \([0, 1]\):

\[
D := (D_1, D_2, D_3) := (P, Q - P, 1 - Q).
\]

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 \( V_1, \ldots, V_n \), so \( D = (D_1, D_2, D_3) \) is \textbf{Dirichlet Dir}(\( t_1 + 1, t_2 + 1, t_3 + 1 \)) distributed.
P and Q (equivalently spacings D) \( \sim \) probability for an ordinary element \( U \) to be small, medium or large, respectively:

- \( U \in (0, P) \sim \) small (with probability \( D_1 \));
- \( U \in (P, Q) \sim \) medium (with probability \( D_2 \));
- \( U \in (Q, 1) \sim \) 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 \( I = (I_1, I_2, I_3) \) (number of small, medium resp. large elements) is multinomially \( \text{Mult}(n - k; D_1, D_2, D_3) \) distributed.

Note that the subproblem sizes \( J = (J_1, J_2, J_3) \) including the sampled-out elements are completely determined by \( I \) via \( J = I + t \).
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 → repeat this same process with the same distribution for subproblems on their respective subinterval of \((0, 1)\).
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:

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

(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 $T_n$).
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 $E_{t1}$ less for calls to left subarrays, $E_{t2}$ less for middle calls and $E_{t3}$ less for right subarrays;
- discounting the future savings $E_t := E_{t1} + E_{t2} + E_{t3}$ 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 $\tilde{T}_n$. 
Taking expectations on both sides in (1), we find a recurrence relation for the expected costs $\mathbb{E}[C_n]$: 

$$
\mathbb{E}[C_n] = 
\begin{cases} 
\mathbb{E}[T_n] + \sum_{j=(j_1,j_2,j_3)} \mathbb{P}(J=j)(\mathbb{E}[C_{j_1}] + \mathbb{E}[C_{j_2}] + \mathbb{E}[C_{j_3}]), & \text{for } n > w; \\
\mathbb{E}[W_n], & \text{for } n \leq w.
\end{cases}
$$

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

$$
\mathbb{P}(J=j) = \frac{(j_1)_{t_1}(j_2)_{t_2}(j_3)_{t_3}}{(n)_{k}}.
$$
Solving the recurrence

**Theorem (Martínez and Roura 2001)**

Let $F_n$ be recursively defined by

$$F_n = \begin{cases} b_n, & \text{for } 0 \leq n < N; \\ t_n + \sum_{j=0}^{n-1} w_{n,j} F_j, & \text{for } n \geq N \end{cases}$$

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

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

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

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

\[\square\]
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 $J$. For $n > w$, this gives

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

Taking expectations on both sides and exploiting independence yields

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

$$= \mathbb{E} T_n + \sum_{j=0}^{n-2} \left( \mathbb{P}(J_1 = j) + \mathbb{P}(J_2 = j) + \mathbb{P}(J_3 = j) \right) \mathbb{E}[C_j] ,$$

which is a recurrence in CMT style with weights

$$\omega_{n,j} = \mathbb{P}(J_1 = j) + \mathbb{P}(J_2 = j) + \mathbb{P}(J_3 = j) .$$
Solving the recurrence

Note that

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

Shape function according to (3): With

\[
\omega(z) = \sum_{l=1}^{3} (k - t_l) \binom{k}{t_l} z^{t_l} (1 - z)^{k-t_l-1}
\]

we find \( \sum_{j=0}^{n-1} \left| \omega_{n,j} - \int_{j/n}^{(j+1)/n} \omega(z) \, dz \right| = O(n^{-1}) \) 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 $\text{Bin}(n - k, D_l)$,
- yielding $\mathbb{P}(I_l = i) = \binom{N}{i} \frac{(t_l + 1)^i (k - t_l)^{N - i}}{(k + 1)^N}$.

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 $K = \alpha$. 

Markus E. Nebel
Java 7’s Dual Pivot Quicksort
2013/12/05

36 / 43
Solving the recurrence

This way we find:

**Theorem**

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

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

where $g$ is given by

$$g(k, t_1, t_2, t_3) = \left(-\sum_{i=1}^{3} \frac{t_i + 1}{k + 1} (H_{t_i + 1} - H_{k + 1})\right)^{-1}.$$
**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 $t^{(j)}$, limiting behaviour of costs.
**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 \( t^{(j)} \) \( \sim \) limiting behaviour of costs.
We find that the overall number of comparisons, swaps resp. bytecodes converge to

\[
\frac{a^*_C}{-\sum_{i=1}^3 \tau_i \ln(\tau_i)}, \quad \frac{a^*_S}{-\sum_{i=1}^3 \tau_i \ln(\tau_i)} \quad \text{resp.} \quad \frac{a^*_{BC}}{-\sum_{i=1}^3 \tau_i \ln(\tau_i)}
\]

with

\[
\begin{align*}
  a^{(j)}_C &\rightarrow a^*_C := 1 + \tau_1 + \tau_2 + (\tau_1 + \tau_2)(\tau_3 - \tau_1) \\
  a^{(j)}_S &\rightarrow a^*_S := \tau_1 + (\tau_1 + \tau_2)\tau_3 \\
  a^{(j)}_{BC} &\rightarrow a^*_{BC} := 10 + 13\tau_1 + 5\tau_2 + 11(\tau_1 + \tau_2)\tau_3 + \tau_1(\tau_1 + \tau_2)
\end{align*}
\]

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 \( 1.4931n \ln n \). 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 \( \tau_1 \) and \( \tau_2 \) 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 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 \( \tau_1 \) and \( \tau_2 \) 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 \( \tau_1 \) and \( \tau_2 \) 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 \( \tau = \left( \frac{1}{3}, \frac{1}{3}, \frac{1}{3} \right) \).
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 Yaroslavskiya’s algorithms (presumably) because of a lacking access to
- branch mispredictions and
- cache misses.
Outlook and Conclusion

Thank you very much for your attention!