2.3 QUICKSORT

- quicksort
- selection
- duplicate keys
- system sorts

https://algs4.cs.princeton.edu
Two classic sorting algorithms: mergesort and quicksort

Critical components in the world’s computational infrastructure.

- Full scientific understanding of their properties has enabled us to develop them into practical system sorts.
- Quicksort honored as one of top 10 algorithms of 20th century in science and engineering.

Mergesort. [last lecture]

Quicksort. [this lecture]
Quicksort t-shirt

```java
private static boolean isSorted(Comparable[] a, int lo, int hi)
    { for (int i = lo + 1; i < a.length; i++)
        if (less(a[i], a[i-1])) return false;
    return true;
}

drawSort(wh); private static void exch(Object[] a, int i, int j)
    { Object tmp = a[i]; a[i] = a[j]; a[j] = tmp; }
    
//public static void main(String[] args) { String[] a = STDIN.readStrings();
//    int n = a.length; for (int i = 0; i < n; i++)
//        drawSort(a); drawArray(a, 0, n-1); }
```

CS @ Princeton
A brief history

Tony Hoare.
- Invented quicksort to translate Russian into English.
- Learned Algol 60 (and recursion) to implement it.

Bob Sedgewick.
- Refined and popularized quicksort.
- Analyzed many versions of quicksort.

Tony Hoare
1980 Turing Award

Bob Sedgewick
2.3 Quicksort

- quicksort
- selection
- duplicate keys
- system sorts
Quicksort overview

Step 1. Shuffle the array.

Step 2. Partition the array so that, for some $j$
- Entry $a[j]$ is in place. “pivot” or “partitioning item”
- No larger entry to the left of $j$.
- No smaller entry to the right of $j$.

Step 3. Sort each subarray recursively.
Quicksort partitioning demo

Repeat until i and j pointers cross.
- Scan i from left to right so long as \( a[i] < a[lo] \).
- Scan j from right to left so long as \( a[j] > a[lo] \).
- Exchange \( a[i] \) with \( a[j] \).
Quicksort partitioning demo

Repeat until i and j pointers cross.

- Scan i from left to right so long as \((a[i] < a[lo])\).
- Scan j from right to left so long as \((a[j] > a[lo])\).
- Exchange \(a[i]\) with \(a[j]\).

When pointers cross.

- Exchange \(a[lo]\) with \(a[j]\).

partitioned!
The music of quicksort partitioning (by Brad Lyon)

The value was larger than the pivot, so the lower one waits while the upper one comes down

We will now start coming down from the right

https://learnforeverlearn.com/pivot_music
Quicksort partitioning: Java implementation

```java
private static int partition(Comparable[] a, int lo, int hi) {
    int i = lo, j = hi+1;
    while (true) {
        while (less(a[++i], a[lo]))
            if (i == hi) break; // find item on left to swap

        while (less(a[lo], a[--j]))
            if (j == lo) break; // find item on right to swap

        if (i >= j) break; // check if pointers cross
        exch(a, i, j);
    }

    exch(a, lo, j); // swap with pivot
    return j; // return index of item now known to be in place
}
```

https://algs4.cs.princeton.edu/23quick/Quick.java.html
In the worst case, how many compares and exchanges does partition() make to partition a subarray of length $n$?

A. $\sim \frac{1}{2} n$ and $\sim \frac{1}{2} n$

B. $\sim \frac{1}{2} n$ and $\sim n$

C. $\sim n$ and $\sim \frac{1}{2} n$

D. $\sim n$ and $\sim n$
public class Quick
{
    private static int partition(Comparable[] a, int lo, int hi)
    { /* see previous slide */ }

    public static void sort(Comparable[] a)
    { stdRandom.shuffle(a);
        sort(a, 0, a.length - 1);
    }

    private static void sort(Comparable[] a, int lo, int hi)
    { if (hi <= lo) return;
        int j = partition(a, lo, hi);
        sort(a, lo, j-1);
        sort(a, j+1, hi);
    }
}

https://algs4.cs.princeton.edu/23quick/Quick.java.html
Quicksort trace

Quicksort trace (array contents after each partition)
Quicksort animation

50 random items

http://www.sorting-algorithms.com/quick-sort
Quicksort: implementation details

Partitioning in-place. Using an extra array makes partitioning easier (and stable), but it is not worth the cost.

Loop termination. Terminating the loop is more subtle than it appears.

Equal keys. Handling duplicate keys is trickier than it appears. [stay tuned]

Preserving randomness. Shuffling is needed for performance guarantee.

Equivalent alternative. Pick a random pivot in each subarray.
Quicksort: empirical analysis (1962)

Running time estimates:

- Algol 60 implementation.
- National Elliott 405 computer.

### Table 1

<table>
<thead>
<tr>
<th>Number of Items</th>
<th>Merge Sort</th>
<th>Quicksort</th>
</tr>
</thead>
<tbody>
<tr>
<td>500</td>
<td>2 min 8 sec</td>
<td>1 min 21 sec</td>
</tr>
<tr>
<td>1,000</td>
<td>4 min 48 sec</td>
<td>3 min 8 sec</td>
</tr>
<tr>
<td>1,500</td>
<td>8 min 15 sec*</td>
<td>5 min 6 sec</td>
</tr>
<tr>
<td>2,000</td>
<td>11 min 0 sec*</td>
<td>6 min 47 sec</td>
</tr>
</tbody>
</table>

* These figures were computed by formula, since they cannot be achieved on the 405 owing to limited store size.

Sorting n 6-word items with 1-word keys

Elliott 405 magnetic disc (16K words)
Quicksort: empirical analysis

Running time estimates:

- Home PC executes $10^8$ compares/second.
- Supercomputer executes $10^{12}$ compares/second.

<table>
<thead>
<tr>
<th></th>
<th>insertion sort ($n^2$)</th>
<th>mergesort ($n \log n$)</th>
<th>quicksort ($n \log n$)</th>
</tr>
</thead>
<tbody>
<tr>
<td>computer</td>
<td>thousand</td>
<td>million</td>
<td>billion</td>
</tr>
<tr>
<td>home</td>
<td>instant</td>
<td>2.8 hours</td>
<td>317 years</td>
</tr>
<tr>
<td>super</td>
<td>instant</td>
<td>1 second</td>
<td>1 week</td>
</tr>
</tbody>
</table>

Lesson 1. Good algorithms are better than supercomputers.
Lesson 2. Great algorithms are better than good ones.
Why do you think quicksort is faster than mergesort in practice?

A. Fewer compares.
B. Less data movement.
C. Both A and B.
D. Neither A nor B.
**Quicksort: worst-case analysis**

**Worst case.** Number of compares is $\sim \frac{1}{2} n^2$.

<table>
<thead>
<tr>
<th>lo</th>
<th>j</th>
<th>hi</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
<th>14</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>B</td>
<td>C</td>
<td>D</td>
<td>E</td>
<td>F</td>
<td>G</td>
<td>H</td>
<td>I</td>
<td>J</td>
<td>K</td>
<td>L</td>
<td>M</td>
<td>N</td>
<td>O</td>
<td>O</td>
<td>O</td>
<td>O</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>a[ ]</th>
</tr>
</thead>
<tbody>
<tr>
<td>lo</td>
</tr>
<tr>
<td>-----</td>
</tr>
<tr>
<td>A</td>
</tr>
</tbody>
</table>

| after random shuffle |
Quicksort: worst-case analysis

**Worst case.** Number of compares is $\sim \frac{1}{2} n^2$.

**Good news.** Worst case for quicksort is mostly irrelevant in practice.
- Exponentially small chance of occurring.
  (unless bug in shuffling or no shuffling)
- More likely that computer is struck by lightning bolt during execution.
Quicksort: probabilistic analysis

Proposition. The expected number of compares $C_n$ to quicksort an array of $n$ distinct keys is $\sim 2n \ln n$ (and the number of exchanges is $\sim \frac{1}{3} n \ln n$).

Recall. Any algorithm with the following structure takes $\Theta(n \log n)$ time.

```
public static void f(int n)
{
    if (n == 0) return;
    f(n/2);  // solve two problems
    f(n/2);  // of half the size
    linear(n); // do a linear amount of work
}
```

Intuition. Each partitioning step divides the problem into two subproblems, each of approximately one-half the size.

“close enough”
Quicksort: probabilistic analysis

**Proposition.** The expected number of compares $C_n$ to quicksort an array of $n$ distinct keys is $\sim 2n \ln n$ (and the number of exchanges is $\sim \frac{1}{3} n \ln n$).

**Pf.** $C_n$ satisfies the recurrence $C_0 = C_1 = 0$ and for $n \geq 2$:

$$C_n = (n+1) + \left( \frac{C_0 + C_{n-1}}{n} \right) + \left( \frac{C_1 + C_{n-2}}{n} \right) + \ldots + \left( \frac{C_{n-1} + C_0}{n} \right)$$

- Multiply both sides by $n$ and collect terms:

$$nC_n = n(n+1) + 2(C_0 + C_1 + \ldots + C_{n-1})$$

- Subtract from this equation the same equation for $n-1$:

$$nC_n - (n-1)C_{n-1} = 2n + 2C_{n-1}$$

- Rearrange terms and divide by $n(n+1)$:

$$\frac{C_n}{n+1} = \frac{C_{n-1}}{n} + \frac{2}{n+1}$$

**analysis beyond scope of this course**
Quicksort: probabilistic analysis

- Repeatedly apply previous equation:

\[
\frac{C_n}{n+1} = \frac{C_{n-1}}{n} + \frac{2}{n+1}
\]

= \frac{C_{n-2}}{n-1} + \frac{2}{n} + \frac{2}{n+1}

= \frac{C_{n-3}}{n-2} + \frac{2}{n-1} + \frac{2}{n} + \frac{2}{n+1}

= \frac{2}{3} + \frac{2}{4} + \frac{2}{5} + \ldots + \frac{2}{n+1}

- Approximate sum by an integral:

\[
C_n = 2(n+1) \left( \frac{1}{3} + \frac{1}{4} + \frac{1}{5} + \ldots + \frac{1}{n+1} \right)
\]

\approx 2(n+1) \int_3^{n+1} \frac{1}{x} \, dx

- Finally, the desired result:

\[
C_n \approx 2(n+1) \ln n \approx 1.39 \, n \lg n
\]
Quicksort properties

Quicksort analysis summary.

- Expected number of compares is $\sim 1.39 \, n \, \lg \, n$.
  [standard deviation is $\sim 0.65 \, n$]
- Expected number of exchanges is $\sim 0.23 \, n \, \lg \, n$.
- Min number of compares is $\sim n \, \lg \, n$.
- Max number of compares is $\sim \frac{1}{2} \, n^2$.

Context. Quicksort is a (Las Vegas) randomized algorithm.

- Guaranteed to be correct.
- Running time depends on outcomes of random coin flips (shuffle).
QuickSort properties

**Proposition.** QuickSort is an *in-place* sorting algorithm.

**Pf.**

- Partitioning: $\Theta(1)$ extra space.
- Function-call stack: $\Theta(\log n)$ extra space (with high probability).

---

can guarantee $\Theta(\log n)$ depth by recurring on smaller subarray before larger subarray (but requires using an explicit stack)

---

**Proposition.** QuickSort is *not* stable.

**Pf.** [ by counterexample ]

\[
\begin{array}{cccc}
 i & j & 0 & 1 & 2 & 3 \\
 B_1 & C_1 & C_2 & A_1 \\
 1 & 3 & B_1 & C_1 & C_2 & A_1 \\
 1 & 3 & B_1 & A_1 & C_2 & C_1 \\
 0 & 1 & A_1 & B_1 & C_2 & C_1 \\
\end{array}
\]
Insertion sort small subarrays.

- Even quicksort has too much overhead for tiny subarrays.
- Cutoff to insertion sort for \( \approx 10 \) items.

```java
private static void sort(Comparable[] a, int lo, int hi)
{
    if (hi <= lo + CUTOFF - 1)
    {
        Insertion.sort(a, lo, hi);
        return;
    }

    int j = partition(a, lo, hi);
    sort(a, lo, j-1);
    sort(a, j+1, hi);
}
```
Median of sample.

- Best choice of pivot item = median.
- Estimate true median by taking median of sample.
- Median-of-3 (random) items.

\[ \sim \frac{12}{7} n \ln n \text{ compares (14\% fewer)} \]
\[ \sim \frac{12}{35} n \ln n \text{ exchanges (3\% more)} \]

```java
private static void sort(Comparable[] a, int lo, int hi) {
    if (hi <= lo) return;

    int median = medianOf3(a, lo, lo + (hi - lo)/2, hi);
    swap(a, lo, median);

    int j = partition(a, lo, hi);
    sort(a, lo, j-1);
    sort(a, j+1, hi);
}
```
2.3 Quicksort

- quicksort
- selection
- duplicate keys
- system sorts
Selection

Goal. Given an array of \( n \) items, find item of rank \( k \).

Ex. Min \( (k = 0) \), max \( (k = n - 1) \), median \( (k = n / 2) \).

Applications.
- Order statistics.
- Find the “top \( k \).”

Use complexity theory as a guide.
- Easy \( O(n \log n) \) algorithm. How?
- Easy \( O(n) \) algorithm for \( k = 0, 1, 2 \). How?
- Easy \( \Omega(n) \) lower bound. Why?

Which is true?
- \( O(n) \) algorithm? [ is there a linear-time algorithm? ]
- \( \Omega(n \log n) \) lower bound? [ is selection as hard as sorting? ]
Quick-select

Partition array so that:
- Entry $a[j]$ is in place.
- No larger entry to the left of $j$.
- No smaller entry to the right of $j$.

Repeat in one subarray, depending on $j$; finished when $j$ equals $k$.

```java
public static Comparable select(Comparable[] a, int k) {
    StdRandom.shuffle(a);
    int lo = 0, hi = a.length - 1;
    while (hi > lo) {
        int j = partition(a, lo, hi);
        if (j < k) lo = j + 1;
        else if (j > k) hi = j - 1;
        else return a[k];
    }
    return a[k];
}
```
Quick-select: probabilistic analysis

**Proposition.** The expected number of compares $C_n$ to quick-select the item of rank $k$ in an array of length $n$ is $\Theta(n)$.

**Intuition.** Each partitioning step approximately halves the length of the array.

**Recall.** Any algorithm with the following structure takes $\Theta(n)$ time.

```java
public static void f(int n) {
    if (n == 0) return;
    linear(n);  // do a linear amount of work
    f(n/2);    // solve one problem of half the size
}
```

"close enough"

$n + n/2 + n/4 + \ldots + 1 \sim 2n$

**Careful analysis yields:** $C_n \sim 2n + 2k \ln (n/k) + 2(n-k) \ln (n/(n-k))$

\[ \leq (2 + 2 \ln 2) n \left| \text{max occurs for median } (k = n/2) \right. \]

\[ \approx 3.38 n \]
Quicksort quiz 3

What is the worst-case running time of quick select?

A. $\Theta(n)$
B. $\Theta(n \log n)$
C. $\Theta(n^2)$
D. $\Theta(2^n)$
Theoretical context for selection


---

**Time Bounds for Selection**

*Manuel Blum, Robert W. Floyd, Vaughan Pratt, Ronald L. Rivest, and Robert E. Tarjan*

Department of Computer Science, Stanford University, Stanford, California 94305

Received November 14, 1972

The number of comparisons required to select the $i$-th smallest of $n$ numbers is shown to be at most a linear function of $n$ by analysis of a new selection algorithm—PICK. Specifically, no more than $5.4305n$ comparisons are ever required. This bound is improved for extreme values of $i$, and a new lower bound on the requisite number of comparisons is also proved.

---

**Remark.** Constants are high $\Rightarrow$ not used in practice.

**Use theory as a guide.**

- Still worthwhile to seek **practical** linear-time (worst-case) algorithm.
- Until one is discovered, use quick-select (if you don’t need a full sort).
2.3 Quicksort

- quicksort
- selection
- duplicate keys
- system sorts
Duplicate keys

Often, purpose of sort is to bring items with equal keys together.

• Sort population by age.
• Remove duplicates from mailing list.
• Sort job applicants by college attended.

Typical characteristics of such applications.

• Huge array.
• Small number of key values.
When partitioning, how to handle keys equal to partitioning key?

A. 

B. 

C. Either A or B.
War story (system sort in C)

**Bug.** A `qsort()` call in C that should have taken seconds was taking minutes to sort a random array of 0s and 1s.

Why is `qsort()` so slow?

```
<table>
<thead>
<tr>
<th>A0</th>
<th>A1</th>
<th>A2</th>
<th>A3</th>
<th>A4</th>
<th>A5</th>
<th>A6</th>
<th>A7</th>
<th>A8</th>
<th>A9</th>
<th>A10</th>
<th>A11</th>
</tr>
</thead>
<tbody>
<tr>
<td>i</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>j</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

**skip over equal keys**

**stop scan on equal keys**
Duplicate keys: partitioning strategies

**Bad.** Don’t stop scans on equal keys.

\[ \Theta(n^2) \text{ compares when all keys equal} \]

```
B A A B A B B B C C C
```

```
A A A A A A A A A A A
```

**Good.** Stop scans on equal keys.

\[ \sim n \lg n \text{ compares when all keys equal} \]

```
B A A B A B C C B C B
```

```
A A A A A A A A A A A
```

**Better.** Put all equal keys in place. How?

\[ \sim n \text{ compares when all keys equal} \]

```
A A A B B B B B C C C
```

```
A A A A A A A A A A A
```
Problem. [Edsger Dijkstra] Given an array of \( n \) buckets, each containing a red, white, or blue pebble, sort them by color.

Operations allowed.
- \( \text{swap}(i, j) \): swap the pebble in bucket \( i \) with the pebble in bucket \( j \).
- \( \text{getColor}(i) \): determine the color of the pebble in bucket \( i \).

Performance requirements.
- Exactly \( n \) calls to \( \text{getColor}() \).
- At most \( n \) calls to \( \text{swap}() \).
- Constant extra space.
3-way partitioning

**Goal.** Use pivot $v = a[l_0]$ to partition array into **three** parts so that:

- Red: smaller entries to the left of $l_t$.
- White: equal entries between $l_t$ and $g_t$.
- Blue: larger entries to the right of $g_t$. 

---

**Diagram:**

*Before:*

- $v$ is the pivot.
- $l_0$ to $hi$.

*After:*

- $<v$ (red)
- $=v$ (white)
- $>v$ (blue)
- $l_0$ to $l_t$ to $g_t$ to $hi$.
Dijkstra’s 3-way partitioning algorithm: demo

- Let $v = a[lo]$ be pivot.
- Scan $i$ from left to right and compare $a[i]$ to $v$.
  - less: exchange $a[lt]$ with $a[i]$; increment both $lt$ and $i$
  - greater: exchange $a[gt]$ with $a[i]$; decrement $gt$
  - equal: increment $i$

![Diagram showing the partitioning process]

**Invariant**

- $<v$
- $=v$
- $\geq v$
- $lt$, $i$, $gt$
Dijkstra’s 3-way partitioning algorithm: demo

- Let $v = a[lo]$ be pivot.
- Scan $i$ from left to right and compare $a[i]$ to $v$.
  - less: exchange $a[lt]$ with $a[i]$; increment both $lt$ and $i$
  - greater: exchange $a[gt]$ with $a[i]$; decrement $gt$
  - equal: increment $i$
private static void sort(Comparable[] a, int lo, int hi) {
    if (hi <= lo) return;
    int lt = lo, gt = hi;
    Comparable v = a[lo];
    int i = lo + 1;
    while (i <= gt) {
        int cmp = a[i].compareTo(v);
        if (cmp < 0) exch(a, lt++, i++);
        else if (cmp > 0) exch(a, i, gt--);
        else i++;
    }
    sort(a, lo, lt - 1);
    sort(a, gt + 1, hi);
}
3-way quicksort: visual trace

equal to partitioning element
Which is worst-case number of compares to 3-way quicksort an array of length $n$ containing only 5 distinct values?

A. $\Theta(n)$

B. $\Theta(n \log n)$

C. $\Theta(n^2)$

D. $\Theta(2^n)$
## Sorting summary

<table>
<thead>
<tr>
<th></th>
<th>inplace?</th>
<th>stable?</th>
<th>best</th>
<th>average</th>
<th>worst</th>
<th>remarks</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>selection</strong></td>
<td>✔️</td>
<td></td>
<td>(\frac{1}{2} n^2)</td>
<td>(\frac{1}{2} n^2)</td>
<td>(\frac{1}{2} n^2)</td>
<td>(n) exchanges</td>
</tr>
<tr>
<td><strong>insertion</strong></td>
<td>✔️</td>
<td>✔️</td>
<td>(n)</td>
<td>(\frac{1}{4} n^2)</td>
<td>(\frac{1}{2} n^2)</td>
<td>use for small (n) or partially sorted arrays</td>
</tr>
<tr>
<td><strong>merge</strong></td>
<td>✔️</td>
<td></td>
<td>(\frac{1}{2} n \lg n)</td>
<td>(n \lg n)</td>
<td>(n \lg n)</td>
<td>(n \log n) guarantee; stable</td>
</tr>
<tr>
<td><strong>timsort</strong></td>
<td>✔️</td>
<td></td>
<td>(n)</td>
<td>(n \lg n)</td>
<td>(n \lg n)</td>
<td>improves mergesort when pre-existing order</td>
</tr>
<tr>
<td><strong>quick</strong></td>
<td>✔️</td>
<td></td>
<td>(n \lg n)</td>
<td>(2 n \ln n)</td>
<td>(\frac{1}{2} n^2)</td>
<td>(n \log n) probabilistic guarantee; fastest in practice</td>
</tr>
<tr>
<td><strong>3-way quick</strong></td>
<td>✔️</td>
<td></td>
<td>(n)</td>
<td>(2 n \ln n)</td>
<td>(\frac{1}{2} n^2)</td>
<td>improves quicksort when duplicate keys</td>
</tr>
<tr>
<td><strong>?</strong></td>
<td>✔️</td>
<td>✔️</td>
<td>(n)</td>
<td>(n \lg n)</td>
<td>(n \lg n)</td>
<td>holy sorting grail</td>
</tr>
</tbody>
</table>
2.3 Quicksort

- quicksort
- selection
- duplicate keys
- system sorts
Sorting applications

Sorting algorithms are essential in a broad variety of applications:

- Sort a list of names.
- Organize an MP3 library.
- Display Google PageRank results.
- List RSS feed in reverse chronological order.

- Find the median.
- Identify statistical outliers.
- Binary search in a database.
- Find duplicates in a mailing list.

- Data compression.
- Computer graphics.
- Computational biology.
- Load balancing on a parallel computer.

...
Engineering a system sort (in 1993)

Bentley–McIlroy quicksort.

- Cutoff to insertion sort for small subarrays.
- Pivot selection: median of 3 or Tukey’s ninther.
- Partitioning scheme: Bentley–McIlroy 3-way partitioning.

In the wild. C, C++, Java 6, ....
Replacement of quicksort in java.util.Arrays with new dual-pivot quicksort

Hello All,

I'd like to share with you new Dual-Pivot Quicksort which is faster than the known implementations (theoretically and experimental). I'd like to propose to replace the JDK's Quicksort implementation by new one.

...

The new Dual-Pivot Quicksort uses *two* pivots elements in this manner:

1. Pick an elements P1, P2, called pivots from the array.
2. Assume that P1 <= P2, otherwise swap it.
3. Reorder the array into three parts: those less than the smaller pivot, those larger than the larger pivot, and in between are those elements between (or equal to) the two pivots.
4. Recursively sort the sub-arrays.

The invariant of the Dual-Pivot Quicksort is:

[ < P1 | P1 <= & <= P2 } > P2 ]

...
Replacement of quicksort in java.util.Arrays with new dual-pivot quicksort

Date: Thu, 29 Oct 2009 11:19:39 +0000
Subject: Replace quicksort in java.util.Arrays with dual-pivot implementation

Changeset: b05abb410c52
Author: alanb
Date: 2009-10-29 11:18 +0000
URL: http://hg.openjdk.java.net/jdk7/tl/jdk/rev/b05abb410c52

6880672: Replace quicksort in java.util.Arrays with dual-pivot implementation
Reviewed-by: jjb
Contributed-by: vladimir.yaroslavskiy at sun.com, joshua.bloch at google.com, jbentley at avaya.com

! src/share/classes/java/util/Arrays.java
+ src/share/classes/java/util/DualPivotQuicksort.java

https://mail.openjdk.java.net/pipermail/compiler-dev/2009-October.txt
Dual-pivot quicksort

Use **two** pivots $p_1$ and $p_2$ and partition into three subarrays:

- Keys less than $p_1$.
- Keys between $p_1$ and $p_2$.
- Keys greater than $p_2$.

<table>
<thead>
<tr>
<th></th>
<th>$&lt; p_1$</th>
<th>$p_1$</th>
<th>$\geq p_1$ and $\leq p_2$</th>
<th>$p_2$</th>
<th>$&gt; p_2$</th>
</tr>
</thead>
<tbody>
<tr>
<td>lo</td>
<td>↑</td>
<td>↑</td>
<td>←</td>
<td>↑</td>
<td>↑</td>
</tr>
<tr>
<td>lt</td>
<td>↑</td>
<td>↑</td>
<td>←</td>
<td>↑</td>
<td>↑</td>
</tr>
<tr>
<td>gt</td>
<td>↑</td>
<td>↑</td>
<td>←</td>
<td>↑</td>
<td>↑</td>
</tr>
<tr>
<td>hi</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Recursively sort three subarrays (skip middle subarray if $p_1 = p_2$).

In the wild. Java 8, Python unstable sort, Android, …
Suppose you are the lead architect of a new programming language. Which principal sorting algorithm would you use for the system sort?

A. Mergesort (e.g., Timsort).
B. Quicksort (e.g., dual-pivot quicksort).
C. Both A and B.
D. Neither A nor B.
System sorts in Java 8 and Java 11

Arrays.sort() \text{ and } Arrays.parallelSort().
\begin{itemize}
  \item Has one method for Comparable objects.
  \item Has an overloaded method for each primitive type.
  \item Has an overloaded method for use with a Comparator.
  \item Has overloaded methods for sorting subarrays.
\end{itemize}

Algorithms.
\begin{itemize}
  \item Timsort for reference types.
  \item Dual-pivot quicksort for primitive types.
  \item Parallel mergesort for Arrays.parallelSort().
\end{itemize}

Q. Why use different algorithms for primitive and reference types?

Bottom line. Use the system sort!