2.3 QUICKSORT

- quicksort
- selection
- duplicate keys
- system sorts
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]
public static void quicksort(char[] items, int left, int right) {
    int i, j;
    char x, y;
    i = left; j = right;
    x = items[(left + right) / 2];
    do {
        while ((items[i] < x) && (i < right)) i++;
        while ((x < items[j]) && (j > left)) j--;
        if (i <= j) {
            y = items[i];
            items[i] = items[j];
            items[j] = y;
            i++; j--;
        }
    } while (i <= j);
    if (left < i) quicksort(items, left, i);
    if (i < right) quicksort(items, i, right);
}
2.3 Quicksort

- quicksort
- selection
- duplicate keys
- system sorts
Quicksort

Basic plan.
- **Shuffle** the array.
- **Partition** so that, for some $j$
  - entry $a[j]$ is in place
  - no larger entry to the left of $j$
  - no smaller entry to the right of $j$
- **Sort** each subarray recursively.

```
input  QUICKSORT EXAMPLE
shuffle KRATELEPUIMQCXOS
partition ECAIEK LPUTMQRXOS
not greater
not less
sort left ACEEIKLPUTMQRXOS
sort right ACEEIKLMPQRSTUX
result ACEEIKLMPQRSTUX
```
• Invented quicksort to translate Russian into English.
  [ but couldn't explain his algorithm or implement it! ]
• Learned Algol 60 (and recursion).
• Implemented quicksort.

Tony Hoare

Tony Hoare
1980 Turing Award

ALGORITHM 64
QUICKSORT
C. A. R. HOARE
Elliott Brothers Ltd., Borehamwood, Hertfordshire, Eng.

procedure quicksort (A,M,N); value M,N;
array A; integer M,N;
comment Quicksort is a very fast and convenient method of
sorting an array in the random-access store of a computer. The
entire contents of the store may be sorted, since no extra space is
required. The average number of comparisons made is 2(M-N) ln
(N-M), and the average number of exchanges is one sixth this
amount. Suitable refinements of this method will be desirable for
its implementation on any actual computer;
begin integer I,J;
if M < N then begin partition (A,M,N,I,J);
  quicksort (A,M,I);
  quicksort (A, I, N)
end
end quicksort.

Communications of the ACM (July 1961)
Tony Hoare

- Invented quicksort to translate Russian into English.
  - [ but couldn't explain his algorithm or implement it! ]
- Learned Algol 60 (and recursion).
- Implemented quicksort.

“There are two ways of constructing a software design: One way is to make it so simple that there are obviously no deficiencies, and the other way is to make it so complicated that there are no obvious deficiencies. The first method is far more difficult.”

“I call it my billion-dollar mistake. It was the invention of the null reference in 1965… This has led to innumerable errors, vulnerabilities, and system crashes, which have probably caused a billion dollars of pain and damage in the last forty years.”
Refined and popularized quicksort.

Analyzed many versions of quicksort.
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)

https://googledrive.com/host/0B2GQktu-wcTicjRaRjV1NmRFN1U/index.html
Quicksort: Java code for partitioning

```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;
        while (less(a[lo], a[--j]))
            if (j == lo) break;
        if (i >= j) break;
        exch(a, i, j);
    }
    exch(a, lo, j);
    return j;
}
```

Quicksort partitioning overview:
- **Before:**
  - Variables: `i`, `j`, `lo`, `hi`
  - State: `i < lo` and `j > hi`

- **During:**
  - Variables: `i`, `j`, `lo`, `hi`
  - State: `i < lo` and `j > hi`
    - `i` increments until a greater or equal element is found, or `i` reaches `hi`
    - `j` decrements until a lesser or equal element is found, or `j` reaches `lo`
    - `i` and `j` swap when they meet

- **After:**
  - Variables: `i`, `j`, `lo`, `hi`
  - State: `i` is now between `lo` and `j` which is greater than or equal to `lo`
    - The partitioning item is now in its correct position
    - `i` and `j` are still between `lo` and `hi`

- **Steps:**
  1. Partition around a pivot (optional)
  2. Recursively sort the two partitions

This algorithm partitions the array into two parts and recursively sorts them, achieving an efficient sorting method.
Q. How many compares (in the worst case) to partition an array of length $N$?

A. $\sim \frac{1}{4} N$
B. $\sim \frac{1}{2} N$
C. $\sim N$
D. $\sim N \log N$
E. I don't know.
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);
  }
}
Quicksort trace

<table>
<thead>
<tr>
<th>lo</th>
<th>j</th>
<th>hi</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>5</td>
<td>15</td>
</tr>
<tr>
<td>0</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>0</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>6</td>
<td>6</td>
<td>15</td>
</tr>
<tr>
<td>7</td>
<td>9</td>
<td>15</td>
</tr>
<tr>
<td>7</td>
<td>7</td>
<td>8</td>
</tr>
<tr>
<td>8</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>10</td>
<td>13</td>
<td>15</td>
</tr>
<tr>
<td>10</td>
<td>12</td>
<td>12</td>
</tr>
<tr>
<td>10</td>
<td>11</td>
<td>11</td>
</tr>
<tr>
<td>10</td>
<td>10</td>
<td>10</td>
</tr>
<tr>
<td>14</td>
<td>14</td>
<td>15</td>
</tr>
<tr>
<td>15</td>
<td>15</td>
<td>15</td>
</tr>
</tbody>
</table>

Initial values
random shuffle

no partition for subarrays of size 1

result

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 is not worth the cost.

Terminating the loop. Testing whether the pointers cross is trickier than it might seem.

Equal keys. When duplicates are present, it is (counter-intuitively) better to stop scans on keys equal to the partitioning item's key. stay tuned

Preserving randomness. Shuffling is needed for performance guarantee.
Equivalent alternative. Pick a random partitioning item in each subarray.
Quick sort: empirical analysis (1961)

Running time estimates:

- Algol 60 implementation.
- National-Elliott 405 computer.

<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.

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>computer</th>
<th>insertion sort ($N^2$)</th>
<th>mergesort ($N \log N$)</th>
<th>quicksort ($N \log N$)</th>
</tr>
</thead>
<tbody>
<tr>
<td></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.
Quicksort: best-case analysis

Best case. Number of compares is $\sim N \lg N$. 
Worst case. Number of compares is $\sim \frac{1}{2} N^2$. 

<table>
<thead>
<tr>
<th>a[]</th>
<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>initial values</td>
<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></td>
<td></td>
<td></td>
</tr>
<tr>
<td>random shuffle</td>
<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></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>14</td>
<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></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>14</td>
<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></td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>14</td>
<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></td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>14</td>
<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></td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>14</td>
<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></td>
</tr>
<tr>
<td>5</td>
<td>5</td>
<td>14</td>
<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></td>
</tr>
<tr>
<td>6</td>
<td>6</td>
<td>14</td>
<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></td>
</tr>
<tr>
<td>7</td>
<td>7</td>
<td>14</td>
<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></td>
</tr>
<tr>
<td>8</td>
<td>8</td>
<td>14</td>
<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></td>
</tr>
<tr>
<td>9</td>
<td>9</td>
<td>14</td>
<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></td>
</tr>
<tr>
<td>10</td>
<td>10</td>
<td>14</td>
<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></td>
</tr>
<tr>
<td>11</td>
<td>11</td>
<td>14</td>
<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></td>
</tr>
<tr>
<td>12</td>
<td>12</td>
<td>14</td>
<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></td>
</tr>
<tr>
<td>13</td>
<td>13</td>
<td>14</td>
<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></td>
</tr>
<tr>
<td>14</td>
<td>14</td>
<td>14</td>
<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></td>
</tr>
</tbody>
</table>
Quicksort: average-case analysis

**Proposition.** The average 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 $N(N + 1)$:

\[
\frac{C_N}{N + 1} = \frac{C_{N-1}}{N} + \frac{2}{N + 1}
\]
Quicksort: average-case 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:

\[
C_N \sim 2(N + 1) \ln N \approx 1.39N \lg N
\]
Quicksort: summary of performance characteristics

Quicksort is a (Las Vegas) randomized algorithm.
  - Guaranteed to be correct.
  - Running time depends on random shuffle.

Average case. Expected number of compares is $\sim 1.39 \, N \lg N$.
  - 39% more compares than mergesort.
  - Faster than mergesort in practice because of less data movement.

Best case. Number of compares is $\sim N \lg N$.

Worst case. Number of compares is $\sim \frac{1}{2} N^2$.
[ but more likely that lightning bolt strikes computer during execution ]
QuickSort properties

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

**Pf.**
- Partitioning: constant extra space.
- Depth of recursion: logarithmic extra space (with high probability).

---

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

**Pf.** [by counterexample]

<table>
<thead>
<tr>
<th>i</th>
<th>j</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>A₁</td>
<td>B₁</td>
<td>C₂</td>
<td>C₁</td>
</tr>
<tr>
<td>1</td>
<td>3</td>
<td>B₁</td>
<td>C₁</td>
<td>C₂</td>
<td>A₁</td>
</tr>
<tr>
<td>1</td>
<td>3</td>
<td>B₁</td>
<td>A₁</td>
<td>C₂</td>
<td>C₁</td>
</tr>
</tbody>
</table>
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);
}
```
Quicksort: practical improvements

Median of sample.

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

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

```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 the $k^{th}$ smallest item.

**Ex.** Min ($k = 0$), max ($k = N - 1$), median ($k = N/2$).

**Applications.**
- Order statistics.
- Find the "top $k$.”

**Use theory as a guide.**
- Easy $N \log N$ upper bound. How?
- Easy $N$ upper bound for $k = 1, 2, 3$. How?
- Easy $N$ lower bound. Why?

**Which is true?**
- $N \log N$ lower bound?   \[ \text{is selection as hard as sorting?} \]
- $N$ upper bound?  \[ \text{is there a linear-time algorithm?} \]
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.

codeSnippet

```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];
}
```

diagram

![Partition Array](image)
Quick-select: mathematical analysis

Proposition. Quick-select takes linear time on average.

Pf sketch.

- Intuitively, each partitioning step splits array approximately in half:
  \[ N + N/2 + N/4 + \ldots + 1 \sim 2N \text{ compares.} \]
- Formal analysis similar to quicksort analysis yields:
  \[ C_N = 2N + 2k \ln (N/k) + 2(N-k) \ln (N/(N-k)) \]
- Ex: \( (2 + 2 \ln 2) N \approx 3.38N \) compares to find median \( k = N/2 \).
Theoretical context for selection


Remark. Constants are high ⇒ 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.
Duplicate keys: stop on equal keys

Our partitioning subroutine stops both scans on equal keys.

Q. Why not continue scans on equal keys?
What is the result of partitioning the following array (skip over equal keys)?

A. 

B. 

C. 

D. I don't know.
Quicksort quiz 3

What is the result of partitioning the following array (stop on equal keys)?

```
| A | A | A | A | A | A | A | A | A | A | A | A | A | A | A |
```

- **A.**
  
- **B.**
  
- **C.**
  
- **D.**  *I don't know.*
Partitioning an array with all equal keys

<table>
<thead>
<tr>
<th>i</th>
<th>j</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>
<th>15</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
</tr>
<tr>
<td>1</td>
<td>15</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
</tr>
<tr>
<td>1</td>
<td>15</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
</tr>
<tr>
<td>2</td>
<td>14</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
</tr>
<tr>
<td>2</td>
<td>14</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
</tr>
<tr>
<td>3</td>
<td>13</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
</tr>
<tr>
<td>3</td>
<td>13</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
</tr>
<tr>
<td>4</td>
<td>12</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
</tr>
<tr>
<td>4</td>
<td>12</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
</tr>
<tr>
<td>5</td>
<td>11</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
</tr>
<tr>
<td>5</td>
<td>11</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
</tr>
<tr>
<td>6</td>
<td>10</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
</tr>
<tr>
<td>6</td>
<td>10</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
</tr>
<tr>
<td>7</td>
<td>9</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
</tr>
<tr>
<td>7</td>
<td>9</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
</tr>
<tr>
<td>8</td>
<td>8</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
</tr>
<tr>
<td>8</td>
<td>8</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
</tr>
</tbody>
</table>
Duplicate keys: partitioning strategies

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

[ \[ \sim \frac{1}{2} N^2 \] 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.

[ \[ \sim N \lg N \] 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 A

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

[ \[ \sim N \] 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 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 $j$.
- $\text{color}(i)$: color of pebble in bucket $i$.

Requirements.

- Exactly $N$ calls to $\text{color()}$.
- At most $N$ calls to $\text{swap()}$.
- Constant extra space.
3-way partitioning

Goal. Partition array into three parts so that:
- Entries between $\lt$ and $\gt$ equal to the partition item.
- No larger entries to left of $\lt$.
- No smaller entries to right of $\gt$.

Dutch national flag problem. [Edsger Dijkstra]
- Conventional wisdom until mid 1990s: not worth doing.
- Now incorporated into C library `qsort()` and Java 6 system sort.
Dijkstra 3-way partitioning demo

- Let $v$ be partitioning item $a[lo]$.
- Scan $i$ from left to right.
  - $(a[i] < v)$: exchange $a[lt]$ with $a[i]$; increment both $lt$ and $i$
  - $(a[i] > v)$: exchange $a[gt]$ with $a[i]$; decrement $gt$
  - $(a[i] == v)$: increment $i$

Invariant

<table>
<thead>
<tr>
<th>&lt;v</th>
<th>=v</th>
<th></th>
<th>&gt;v</th>
</tr>
</thead>
<tbody>
<tr>
<td>lt</td>
<td>i</td>
<td></td>
<td>gt</td>
</tr>
</tbody>
</table>

Play demo
Dijkstra 3-way partitioning demo

- Let $v$ be partitioning item $a[lo]$.
- Scan $i$ from left to right.
  - $(a[i] < v)$: exchange $a[lt]$ with $a[i]$; increment both $lt$ and $i$
  - $(a[i] > v)$: exchange $a[gt]$ with $a[i]$; decrement $gt$
  - $(a[i] == v)$: increment $i$

**Invariant**

\[
\begin{array}{c}
\leq v & = v & \text{gray} & > v
\end{array}
\]

- $lt$ - $i$ - $gt$
Dijkstra's 3-way partitioning: trace

<table>
<thead>
<tr>
<th>Iteration</th>
<th>Left</th>
<th>Index</th>
<th>Right</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>11</td>
<td>R B W W R W B R R R W B R</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>11</td>
<td>R B W W R W B R R W B R</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>11</td>
<td>B R W W R W B R R W B R</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>10</td>
<td>B R R W R W B R R R W B W</td>
</tr>
<tr>
<td>1</td>
<td>3</td>
<td>10</td>
<td>B R R W R W B R R W B</td>
</tr>
<tr>
<td>1</td>
<td>3</td>
<td>9</td>
<td>B R R B R W B R R W W</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
<td>9</td>
<td>B B R R R W B R R R W W</td>
</tr>
<tr>
<td>2</td>
<td>5</td>
<td>9</td>
<td>B B R R R W B R R R W W</td>
</tr>
<tr>
<td>2</td>
<td>5</td>
<td>8</td>
<td>B B R R R W B R R W W W</td>
</tr>
<tr>
<td>2</td>
<td>5</td>
<td>7</td>
<td>B B R R R R B R W W W W</td>
</tr>
<tr>
<td>2</td>
<td>6</td>
<td>7</td>
<td>B B R R R R B R W W W W</td>
</tr>
<tr>
<td>3</td>
<td>7</td>
<td>7</td>
<td>B B B R R R R R W W W W</td>
</tr>
<tr>
<td>3</td>
<td>8</td>
<td>7</td>
<td>B B B R R R R R R W W W W</td>
</tr>
<tr>
<td>3</td>
<td>8</td>
<td>7</td>
<td>B B B R R R R R R R W W W W</td>
</tr>
</tbody>
</table>

3-way partitioning trace (array contents after each loop iteration)
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;
    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
Duplicate keys: lower bound

**Sorting lower bound.** If there are \( n \) distinct keys and the \( i^{th} \) one occurs \( x_i \) times, any compare-based sorting algorithm must use at least

\[
\lg \left( \frac{N!}{x_1! \cdot x_2! \cdots x_n!} \right) \sim - \sum_{i=1}^{n} x_i \log \frac{x_i}{N} \\
\]

\( N \log N \) when all distinct; linear when only a constant number of distinct keys compares in the worst case.

**Proposition.** [Sedgewick-Bentley 1997]
Quick sort with 3-way partitioning is entropy-optimal.

**Pf.** [beyond scope of course]

**Bottom line.** Quicksort with 3-way partitioning reduces running time from linearithmic to linear in broad class of applications.
## Sorting summary

<table>
<thead>
<tr>
<th>inplace?</th>
<th>stable?</th>
<th>best</th>
<th>average</th>
<th>worst</th>
<th>remarks</th>
</tr>
</thead>
<tbody>
<tr>
<td>selection</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>insertion</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 ordered</td>
</tr>
<tr>
<td>shell</td>
<td>✔</td>
<td>$N \log_3 N$</td>
<td>?</td>
<td>$cN^{3/2}$</td>
<td>tight code; subquadratic</td>
</tr>
<tr>
<td>merge</td>
<td>✔</td>
<td>$\frac{1}{2}N \log N$</td>
<td>$N \log N$</td>
<td>$N \log N$</td>
<td>$N \log N$ guarantee; stable</td>
</tr>
<tr>
<td>timsort</td>
<td>✔</td>
<td>$N$</td>
<td>$N \log N$</td>
<td>$N \log N$</td>
<td>improves mergesort when preexisting order</td>
</tr>
<tr>
<td>quick</td>
<td>✔</td>
<td>$N \log N$</td>
<td>$2N \ln N$</td>
<td>$\frac{1}{2}N^2$</td>
<td>$N \log N$ probabilistic guarantee; fastest in practice</td>
</tr>
<tr>
<td>3-way quick</td>
<td>✔</td>
<td>$N$</td>
<td>$2N \ln N$</td>
<td>$\frac{1}{2}N^2$</td>
<td>improves quicksort when duplicate keys</td>
</tr>
<tr>
<td>?</td>
<td>✔ ✔</td>
<td>$N$</td>
<td>$N \log N$</td>
<td>$N \log N$</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.

...
War story (system sort in C)

A beautiful bug report. [Allan Wilks and Rick Becker, 1991]

We found that qsort is unbearably slow on "organ-pipe" inputs like "01233210":

```c
main (int argc, char**argv) {
    int n = atoi(argv[1]), i, x[100000];
    for (i = 0; i < n; i++)
        x[i] = i;
    for ( ; i < 2*n; i++)
        x[i] = 2*n-i-1;
    qsort(x, 2*n, sizeof(int), intcmp);
}

Here are the timings on our machine:
$ time a.out 2000
real  5.85s
$ time a.out 4000
real  21.64s
$ time a.out 8000
real  85.11s
```
War story (system sort in C)

**Bug.** A `qsort()` call that should have taken seconds was taking minutes.

At the time, almost all `qsort()` implementations based on those in:

- Version 7 Unix (1979): quadratic time to sort organ-pipe arrays.
- BSD Unix (1983): quadratic time to sort random arrays of 0s and 1s.
Engineering a system sort (in 1993)

Bentley-McIlroy quicksort.

- Cutoff to insertion sort for small subarrays.
- Partitioning item: median of 3 or Tukey's ninther.
- Partitioning scheme: Bentley-McIlroy 3-way partitioning.

Very widely used. 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, jhbentley at avaya.com

! make/java/java/FILES_java.gmk
! src/share/classes/java/util/Arrays.java
+ src/share/classes/java/util/DualPivotQuicksort.java

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

Use two partitioning items $p_1$ 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>$\uparrow$</td>
<td>$\uparrow$</td>
<td>$\uparrow$</td>
<td>$\uparrow$</td>
<td>hi</td>
</tr>
<tr>
<td>lt</td>
<td>$\uparrow$</td>
<td>$\uparrow$</td>
<td>$\uparrow$</td>
<td>$\uparrow$</td>
<td></td>
</tr>
</tbody>
</table>

Recursively sort three subarrays.

**Note.** Skip middle subarray if $p_1 = p_2$. degenerates to Dijkstra's 3-way partitioning
Dual-pivot partitioning demo

Initialization.

- Choose $a[lo]$ and $a[hi]$ as partitioning items.
- Exchange if necessary to ensure $a[lo] \leq a[hi]$. 

exchange $a[lo]$ and $a[hi]$
Dual-pivot partitioning demo

Main loop. Repeat until $i$ and $gt$ pointers cross.

- If $(a[i] < a[lo])$, exchange $a[i]$ with $a[lt]$ and increment $lt$ and $i$.
- Else if $(a[i] > a[hi])$, exchange $a[i]$ with $a[gt]$ and decrement $gt$.
- Else, increment $i$.
Finalize.

- Exchange $a[hi]$ with $a[++gt]$.

### 3-way partitioned

<table>
<thead>
<tr>
<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>↑ $lt$</td>
<td>↑ $gt$</td>
<td>↑ $hi$</td>
<td></td>
</tr>
</tbody>
</table>
Dual-pivot quicksort

Use two partitioning items $p_1$ 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></td>
<td>↑ lo</td>
<td>↑ lt</td>
<td>↑ gt</td>
<td>↑</td>
<td>↑ hi</td>
</tr>
</tbody>
</table>

Now widely used. Java 7, Python unstable sort, Android, ...
Three-pivot quicksort

Use **three** partitioning items $p_1$, $p_2$, and $p_3$ and partition into four subarrays:

- Keys less than $p_1$.
- Keys between $p_1$ and $p_2$.
- Keys between $p_2$ and $p_3$.
- Keys greater than $p_3$.

<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>$\geq p_2$ and $\leq p_3$</th>
<th>$p_3$</th>
<th>$&gt; p_3$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\uparrow$</td>
<td>$\downarrow$</td>
<td>$\downarrow$</td>
<td>$\downarrow$</td>
<td>$\downarrow$</td>
<td>$\downarrow$</td>
<td>$\downarrow$</td>
<td>$\uparrow$</td>
</tr>
<tr>
<td>lo</td>
<td>a0</td>
<td>a1</td>
<td>a2</td>
<td>a3</td>
<td>hi</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

---

Multi-Pivot Quicksort: Theory and Experiments

Shrinu Kushagra  
skushagr@uwaterloo.ca  
University of Waterloo

Alejandro López-Ortiz  
alopez-o@uwaterloo.ca  
University of Waterloo

Aurick Qiao  
a2qiao@uwaterloo.ca  
University of Waterloo

J. Ian Munro  
imunro@uwaterloo.ca  
University of Waterloo

---
Why do 2-pivot (and 3-pivot) quicksort perform better than 1-pivot?

A. Fewer compares.
B. Fewer exchanges.
C. Fewer cache misses.
D. I don't know.
Why do 2-pivot (and 3-pivot) quicksort perform better than 1-pivot?

A. Fewer compares.
B. Fewer exchanges.
C. Fewer cache misses.

<table>
<thead>
<tr>
<th>partitioning</th>
<th>compares</th>
<th>exchanges</th>
<th>entries scanned</th>
</tr>
</thead>
<tbody>
<tr>
<td>1-pivot</td>
<td>$2N \ln N$</td>
<td>$0.333N \ln N$</td>
<td>$2N \ln N$</td>
</tr>
<tr>
<td>median-of-3</td>
<td>$1.714N \ln N$</td>
<td>$0.343N \ln N$</td>
<td>$1.714N \ln N$</td>
</tr>
<tr>
<td>2-pivot</td>
<td>$1.9N \ln N$</td>
<td>$0.6N \ln N$</td>
<td>$1.6N \ln N$</td>
</tr>
<tr>
<td>3-pivot</td>
<td>$1.846N \ln N$</td>
<td>$0.616N \ln N$</td>
<td>$1.385N \ln N$</td>
</tr>
</tbody>
</table>

Reference: Analysis of Pivot Sampling in Dual-Pivot Quicksort by Wild–Nebel–Martínez

Bottom line. Caching can have a significant impact on performance.
Which sorting algorithm to use?

Many sorting algorithms to choose from:

<table>
<thead>
<tr>
<th>sorts</th>
<th>algorithms</th>
</tr>
</thead>
<tbody>
<tr>
<td>elementary sorts</td>
<td>insertion sort, selection sort, bubblesort, shaker sort, ...</td>
</tr>
<tr>
<td>subquadratic sorts</td>
<td>quicksort, mergesort, heapsort, shellsort, samplesort, ...</td>
</tr>
<tr>
<td>system sorts</td>
<td>dual-pivot quicksort, timsort, introsort, ...</td>
</tr>
<tr>
<td>external sorts</td>
<td>Poly-phase mergesort, cascade-merge, psort, ....</td>
</tr>
<tr>
<td>radix sorts</td>
<td>MSD, LSD, 3-way radix quicksort, ...</td>
</tr>
<tr>
<td>parallel sorts</td>
<td>bitonic sort, odd-even sort, smooth sort, GPUsort, ...</td>
</tr>
</tbody>
</table>
Which sorting algorithm to use?

Applications have diverse attributes.

- Stable?
- Parallel?
- In-place?
- Deterministic?
- Duplicate keys?
- Multiple key types?
- Linked list or arrays?
- Large or small items?
- Randomly-ordered array?
- Guaranteed performance?

Q. Is the system sort good enough?
A. Usually.
System sort in Java 7

Arrays.sort().

- Has method for objects that are Comparable.
- Has overloaded method for each primitive type.
- Has overloaded method for use with a Comparator.
- Has overloaded methods for sorting subarrays.

Algorithms.

- Dual-pivot quicksort for primitive types.
- Timsort for reference types.

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

**INEFFECTIVE SORTS**

**DEFINE HALFHEARTEDMERGESORT(LIST):**
- IF LENGTH(LIST) < 2:
  - RETURN LIST
- PIVOT = INT(LENGTH(LIST) / 2)
- A = HALFHEARTEDMERGESORT(LIST[:PIVOT])
- B = HALFHEARTEDMERGESORT(LIST[PIVOT:])
  // UMM...M
- RETURN [A, B] // HERE...SORRY.

**DEFINE FASTBOGOSORT(LIST):**
- AN OPTIMIZED BOGOSORT
- RUNS IN O(N Log N)
- FOR N FROM 1 TO LOG(LENGTH(LIST)):
  - SHUFFLE(LIST)
  - IF ISSORTED(LIST):
    - RETURN LIST
  - RETURN “KERNEL PAGE FAULT (ERROR CODE: 2)"

**DEFINE JOBINTERVIEWQUICKSORT(LIST):**
- OK SO YOU CHOOSE A PIVOT
- THEN DIVIDE THE LIST IN HALF
- FOR EACH HALF:
  - CHECK TO SEE IF IT’S SORTED
  - NO WAY, IT DOESN’T MATTER
  - COMPARE EACH ELEMENT TO THE PIVOT
    - THE BIGGER ONES GO IN A NEW LIST
    - THE EQUAL ONES GO INTO...OH
    - THE SECOND LIST FROM BEFORE
  - HANG ON, LET ME NAME THE LISTS
    - THIS IS LIST A
    - THE NEW ONE IS LIST B
  - PUT THE BIG ONES INTO LIST B
  - NOW TAKE THE SECOND LIST
    - CALL IT LIST UH, A2
  - WHICH ONE WAS THE PIVOT IN?
  - SCRATCH ALL THAT
  - IT JUST RECURSIVELY CALLS ITSELF
  - UNTIL BOTH LISTS ARE EMPTY
  - RIGHT?
  - NOT EMPTY, BUT YOU KNOW WHAT I MEAN
  - AM I ALLOWED TO USE THE STANDARD LIBRARIES?

**DEFINE PANICSORT(LIST):**
- IF ISSORTED(LIST):
  - RETURN LIST
- FOR N FROM 1 TO 10000:
  - PIVOT = RANDOM(0, LENGTH(LIST))
  - LIST = LIST[PIVOT:] + LIST[:PIVOT]
  - IF ISSORTED(LIST):
    - RETURN LIST
  - IF ISSORTED(LIST):
    - RETURN LIST
  - IF ISSORTED(LIST):
    - // THIS CAN’T BE HAPPENING
    - RETURN LIST
  - IF ISSORTED(LIST):
    - // COME ON COME ON
    - RETURN LIST
  - // OH JEEZ
  - // I’M GONNA BE IN SO MUCH TROUBLE
  - LIST = []
  - SYSTEM(“SHUTDOWN -H +5”)  // SYSTEM(“RM -RF /”)  // RM -RF ~/*
  - SYSTEM(“RM -RF /”)  // SYSTEM(“RD /S Q C:\ *”)  // PORTABILITY
  - RETURN [1, 2, 3, 4, 5]