QuickSort

Feb. 27, 2017

Acknowledgement: The course slides are adapted from the slides prepared by R. Sedgewick and K. Wayne of Princeton University.
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 piece recursively.

Sir Charles Antony Richard Hoare
1980 Turing Award
Shuffling

• Shuffling is the process of rearranging an array of elements randomly.
• A good shuffling algorithm is unbiased, where every ordering is equally likely.

• e.g. the Fisher–Yates shuffle (aka. the Knuth shuffle)

http://bl.ocks.org/mbostock/39566aca95eb03ddd526
Quicksort partitioning

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].

stop i scan because a[i] >= a[lo]
Quicksort partitioning

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

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

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 \(a[j]\).

\[\begin{array}{cccccccccccccccccc}
K & R & A & T & E & L & E & P & U & I & M & Q & C & X & O & S \\
\uparrow & \uparrow & & & & & & & & & & & \uparrow \\
lo & i & & & & & & & & & & & & j \\
\end{array}\]

stop \(j\) scan and exchange \(a[i]\) with \(a[j]\)
Quicksort partitioning

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

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

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 $a[j]$.

\[\begin{array}{cccccccccccccc}
K & C & A & T & E & L & E & P & U & I & M & Q & R & X & O & S
\end{array}\]

\[
\begin{array}{ccc}
\uparrow & \uparrow & \uparrow \\
lo & i & j
\end{array}\]

stop $i$ scan because $a[i] >= a[lo]$
Quicksort partitioning

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 $a[j]$. 

<table>
<thead>
<tr>
<th>K</th>
<th>C</th>
<th>A</th>
<th>T</th>
<th>E</th>
<th>L</th>
<th>E</th>
<th>P</th>
<th>U</th>
<th>I</th>
<th>M</th>
<th>Q</th>
<th>R</th>
<th>X</th>
<th>O</th>
<th>S</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></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>

$lo$ $i$ $j$
Quicksort partitioning

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

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 \( a[j] \).

\[
\begin{array}{cccccccccccccc}
K & C & A & T & E & L & E & P & U & I & M & Q & R & X & O & S \\
\uparrow & \uparrow & \uparrow \\
lo & i & j \\
\end{array}
\]

stop \( j \) scan and exchange \( a[i] \) with \( a[j] \)
Quicksort partitioning

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

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

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 $a[j]$.

stop i scan because $a[i] \geq a[lo]$
Quicksort partitioning

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

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

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 $a[j]$.

```
K C A I E L E P U T M Q R X O S
```

stop $j$ scan and exchange $a[i]$ with $a[j]$
Quicksort partitioning

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 $a[j]$.

<table>
<thead>
<tr>
<th>K</th>
<th>C</th>
<th>A</th>
<th>I</th>
<th>E</th>
<th>E</th>
<th>L</th>
<th>P</th>
<th>U</th>
<th>T</th>
<th>M</th>
<th>Q</th>
<th>R</th>
<th>X</th>
<th>O</th>
<th>S</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></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>

$lo$  $i$  $j$
Quicksort partitioning

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 $a[j]$.

stop $i$ scan because $a[i] >= a[lo]$
Quicksort partitioning

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 \( a[j] \).

\[ \uparrow \quad \uparrow \quad \uparrow \]
\[ \downarrow \quad \text{lo} \quad \downarrow \quad \text{j} \quad \text{i} \]

stop \( j \) scan because \( a[j] \leq a[lo] \)
Quicksort partitioning

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 \(a[j]\).
Quicksort partitioning

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!
Quick sort partitioning

Basic plan.

• Scan \( i \) from left for an item that belongs on the right.
• Scan \( j \) from right for an item that belongs on the left.
• Exchange \( a[i] \) and \( a[j] \).
• Repeat until pointers cross.

Partitioning trace (array contents before and after each exchange)
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;
}
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

### Initial Values
- Random shuffle:

<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>
<th>15</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>Q</td>
<td>U</td>
<td>I</td>
<td>C</td>
<td>K</td>
<td>S</td>
<td>O</td>
<td>R</td>
<td>T</td>
<td>E</td>
<td>X</td>
<td>A</td>
<td>M</td>
<td>P</td>
<td>L</td>
<td>E</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>K</td>
<td>R</td>
<td>A</td>
<td>T</td>
<td>E</td>
<td>L</td>
<td>E</td>
<td>P</td>
<td>U</td>
<td>I</td>
<td>M</td>
<td>Q</td>
<td>C</td>
<td>X</td>
<td>O</td>
<td>S</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>E</td>
<td>C</td>
<td>A</td>
<td>I</td>
<td>E</td>
<td>K</td>
<td>L</td>
<td>P</td>
<td>U</td>
<td>T</td>
<td>M</td>
<td>Q</td>
<td>R</td>
<td>X</td>
<td>O</td>
<td>S</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>A</td>
<td>C</td>
<td>E</td>
<td>E</td>
<td>I</td>
<td>K</td>
<td>L</td>
<td>P</td>
<td>U</td>
<td>T</td>
<td>M</td>
<td>Q</td>
<td>R</td>
<td>X</td>
<td>O</td>
<td>S</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>A</td>
<td>C</td>
<td>E</td>
<td>E</td>
<td>I</td>
<td>K</td>
<td>L</td>
<td>P</td>
<td>U</td>
<td>T</td>
<td>M</td>
<td>Q</td>
<td>R</td>
<td>X</td>
<td>O</td>
<td>S</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>A</td>
<td>C</td>
<td>E</td>
<td>E</td>
<td>I</td>
<td>K</td>
<td>L</td>
<td>P</td>
<td>U</td>
<td>T</td>
<td>M</td>
<td>Q</td>
<td>R</td>
<td>X</td>
<td>O</td>
<td>S</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>A</td>
<td>C</td>
<td>E</td>
<td>E</td>
<td>I</td>
<td>K</td>
<td>L</td>
<td>P</td>
<td>U</td>
<td>T</td>
<td>M</td>
<td>Q</td>
<td>R</td>
<td>X</td>
<td>O</td>
<td>S</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>A</td>
<td>C</td>
<td>E</td>
<td>E</td>
<td>I</td>
<td>K</td>
<td>L</td>
<td>P</td>
<td>U</td>
<td>T</td>
<td>M</td>
<td>Q</td>
<td>R</td>
<td>X</td>
<td>O</td>
<td>S</td>
</tr>
</tbody>
</table>

### Results
- No partition for subarrays of size 1

Quicksort trace (array contents after each partition):

```
AC E E I K L M O P Q R S T U X
```
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 a bit trickier than it might seem.

Staying in bounds. The \((j == 10)\) test is redundant (why?), but the \((i == hi)\) test is not.

Preserving randomness. Shuffling is needed for performance guarantee.

Equal keys. When duplicates are present, it is (counter-intuitively) better to stop on keys equal to the partitioning item's key.
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 $\sim N \lg N$.

Each partitioning process splits the array exactly in half.

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

| initial values | H A C B F E G D L I K J N M O |
| random shuffle | H A C B F E G D L I K J N M O |
| 0 7 14 D A C B F E G H L I K J N M O |
| 0 3 6 B A C D F E G H L I K J N M O |
| 0 1 2 A B C D F E G H L I K J N M O |
| 0 0 A B C D F E G H L I K J N M O |
| 2 2 A B C D F E G H L I K J N M O |
| 4 5 6 A B C D E F G H L I K J N M O |
| 4 4 A B C D E F G H L I K J N M O |
| 6 6 A B C D E F G H L I K J N M O |
| 8 11 14 A B C D E F G H J I K L N M O |
| 8 9 10 A B C D E F G H I J K L N M O |
| 8 8 A B C D E F G H I J K L N M O |
| 10 10 A B C D E F G H I J K L N M O |
| 12 13 14 A B C D E F G H I J K L M N O |
| 12 12 A B C D E F G H I J K L M N O |
| 14 14 A B C D E F G H I J K L M N O |
| A B C D E F G H I J K L M N O |
| a[ ] |
Worst case. Number of compares is $\sim \frac{1}{2} N^2$.

One of the subarrays is empty for every partition.
**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 this from 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 above 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:

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

\[
\sim 2(N + 1) \int_3^{N+1} \frac{1}{x} \, dx
\]

- Finally, the desired result:

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

Worst case. Number of compares is quadratic.
- \( N + (N - 1) + (N - 2) + \ldots + 1 \approx \frac{1}{2} N^2. \)
- More likely that your computer is struck by lightning bolt.

Average case. Number of compares is \( \sim 1.39 N \lg N. \)
- 39% more compares than mergesort.
- But faster than mergesort in practice because of less data movement.

Random shuffle.
- Probabilistic guarantee against worst case.
- Basis for math model that can be validated with experiments.

Caveat emptor. Many textbook implementations go \textbf{quadratic} if array
- Is sorted or reverse sorted.
- Has many duplicates (even if randomized!)
**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.**

<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></td>
<td></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>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>
<tr>
<td>0</td>
<td>1</td>
<td>A₁</td>
<td>B₁</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.
- Note: could delay insertion sort until one pass at end.

```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.

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

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

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

~ 12/7 N ln N compares (slightly fewer)
~ 12/35 N ln N exchanges (slightly more)
Quicksort with median-of-3 and cutoff to insertion sort: visualization
Selection

Goal. Given an array of $N$ items, find the $k^{th}$ largest.

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? is selection as hard as sorting?
- $N$ upper bound? is there a linear-time algorithm for each $k$?
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];
}
```
Proposition. Quick-select takes linear time on average.

Pf sketch.

• Intuitively, each partitioning step splits array approximately in half:
  \( N + N/2 + N/4 + \ldots + 1 \sim 2N \) compares.

• Formal analysis similar to quicksort analysis yields:

\[
C_N = 2N + k \ln \left( \frac{N}{k} \right) + (N - k) \ln \left( \frac{N}{N - k} \right)
\]

(2 + 2 \ln 2) N to find the median

Remark. Quick-select uses \( \sim \frac{1}{2} N^2 \) compares in the worst case, but (as with quicksort) the random shuffle provides a probabilistic guarantee.
Theoretical context for selection

**Proposition.** [Blum, Floyd, Pratt, Rivest, Tarjan, 1973] There exists a compare-based selection algorithm whose worst-case running time is linear.

**Remark.** But, constants are too 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.
Duplicate keys

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

- Sort population by age.
- Find collinear points.
- Remove duplicates from mailing list.
- Sort job applicants by college attended.

Typical characteristics of such applications.

- Huge array.
- Small number of key values.

<table>
<thead>
<tr>
<th>City</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Chicago</td>
<td>09:00:00</td>
</tr>
<tr>
<td>Chicago</td>
<td>09:00:03</td>
</tr>
<tr>
<td>Chicago</td>
<td>09:00:13</td>
</tr>
<tr>
<td>Chicago</td>
<td>09:01:00</td>
</tr>
<tr>
<td>Chicago</td>
<td>09:00:59</td>
</tr>
<tr>
<td>Houston</td>
<td>09:01:10</td>
</tr>
<tr>
<td>Houston</td>
<td>09:00:13</td>
</tr>
<tr>
<td>Phoenix</td>
<td>09:37:44</td>
</tr>
<tr>
<td>Phoenix</td>
<td>09:00:03</td>
</tr>
<tr>
<td>Phoenix</td>
<td>09:14:25</td>
</tr>
<tr>
<td>Seattle</td>
<td>09:10:25</td>
</tr>
<tr>
<td>Seattle</td>
<td>09:36:14</td>
</tr>
<tr>
<td>Seattle</td>
<td>09:22:43</td>
</tr>
<tr>
<td>Seattle</td>
<td>09:10:11</td>
</tr>
<tr>
<td>Seattle</td>
<td>09:22:54</td>
</tr>
</tbody>
</table>
Duplicate keys

Mergesort with duplicate keys.
Always between $\frac{1}{2} N \lg N$ and $N \lg N$ compares.

Quicksort with duplicate keys.
• Algorithm goes quadratic unless partitioning stops on equal keys!
• 1990s C user found this defect in `qsort()`.

Several textbook and system implementation also have this defect.

```
S T O P O N E Q U A L K E Y S
```

- swap
- if we don't stop on equal keys
- if we stop on equal keys
Duplicate keys: the problem

Mistake. Put all items equal to the partitioning item on one side.
Consequence. \( \sim \frac{1}{2} N^2 \) compares when all keys equal.

\[
\begin{array}{cccccccc}
C & C & C & C & & & & \\
\end{array}
\begin{array}{cccccccc}
\end{array}
\]

Recommended. Stop scans on items equal to the partitioning item.
Consequence. \( \sim N \log N \) compares when all keys equal.

\[
\begin{array}{cccccccc}
B & C & B & C & B & & & \\
\end{array}
\begin{array}{cccccccc}
\end{array}
\]

Desirable. Put all items equal to the partitioning item in place.

\[
\begin{array}{cccccccc}
C & C & C & C & & & & \\
\end{array}
\begin{array}{cccccccc}
\end{array}
\]
**3-way partitioning**

**Goal.** Partition array into 3 parts so that:
- Entries between $\lt$ and $\gt$ equal to partition item $v$.
- 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.
- New approach discovered when fixing mistake in C library `qsort()`.
- Now incorporated into `qsort()` and Java system sort.
Dijkstra 3-way partitioning

- Let $v$ be partitioning item $a[lo]$.
- Scan $i$ from left to right.
  - $(a[i] < v)$: exchange $a[lt]$ with $a[i]$ and increment both $lt$ and $i$
  - $(a[i] > v)$: exchange $a[gt]$ with $a[i]$ and decrement $gt$
  - $(a[i] == v)$: increment $i$
Dijkstra 3-way partitioning

- Let $v$ be partitioning item $a[10]$.
- Scan $i$ from left to right.
  - $(a[i] < v)$: exchange $a[lt]$ with $a[i]$ and increment both $lt$ and $i$
  - $(a[i] > v)$: exchange $a[gt]$ with $a[i]$ and decrement $gt$
  - $(a[i] == v)$: increment $i$
Dijkstra 3-way partitioning

• Let $v$ be partitioning item $a[10]$.
• Scan $i$ from left to right.
  - $(a[i] < v)$: exchange $a[lt]$ with $a[i]$ and increment both $lt$ and $i$
  - $(a[i] > v)$: exchange $a[gt]$ with $a[i]$ and decrement $gt$
  - $(a[i] == v)$: increment $i$
Dijkstra 3-way partitioning

1. Let \( v \) be partitioning item \( a[lo] \).
2. Scan \( i \) from left to right.
   - \((a[i] < v)\): exchange \( a[lt] \) with \( a[i] \) and increment both \( lt \) and \( i \)
   - \((a[i] > v)\): exchange \( a[gt] \) with \( a[i] \) and decrement \( gt \)
   - \((a[i] == v)\): increment \( i \)
Dijkstra 3-way partitioning

- Let v be partitioning item a[i0].
- Scan i from left to right.
  - (a[i] < v): exchange a[lt] with a[i] and increment both lt and i
  - (a[i] > v): exchange a[gt] with a[i] and decrement gt
  - (a[i] == v): increment i
Dijkstra 3-way partitioning

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

Scan $i$ from left to right.

- $(a[i] < v)$: exchange $a[lt]$ with $a[i]$ and increment both $lt$ and $i$
- $(a[i] > v)$: exchange $a[gt]$ with $a[i]$ and decrement $gt$
- $(a[i] == v)$: increment $i$

Dijkstra 3-way partitioning
Dijkstra 3-way partitioning

- Let $v$ be partitioning item $a[10]$.
- Scan $i$ from left to right.
  - $(a[i] < v)$: exchange $a[lt]$ with $a[i]$ and increment both $lt$ and $i$
  - $(a[i] > v)$: exchange $a[gt]$ with $a[i]$ and decrement $gt$
  - $(a[i] == v)$: increment $i$
Dijkstra 3-way partitioning

- Let $v$ be partitioning item $a[10]$.
- Scan $i$ from left to right.
  - $(a[i] < v)$: exchange $a[lt]$ with $a[i]$ and increment both $lt$ and $i$
  - $(a[i] > v)$: exchange $a[gt]$ with $a[i]$ and decrement $gt$
  - $(a[i] == v)$: increment $i$

### Invariant

<table>
<thead>
<tr>
<th>$&lt;$V</th>
<th>=$V$</th>
<th>&gt;$V$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$lt$</td>
<td>$i$</td>
<td>$gt$</td>
</tr>
</tbody>
</table>
Dijkstra 3-way partitioning

- Let \( v \) be partitioning item \( a[10] \).
- Scan \( i \) from left to right.
  - \( (a[i] < v) \): exchange \( a[lt] \) with \( a[i] \) and increment both \( lt \) and \( i \)
  - \( (a[i] > v) \): exchange \( a[gt] \) with \( a[i] \) and decrement \( gt \)
  - \( (a[i] == v) \): increment \( i \)

**Invariant**

\[
\begin{align*}
&<v & =v & \text{[gray]} & >v \\
&\downarrow & \downarrow & \downarrow
\end{align*}
\]
Dijkstra 3-way partitioning

- Let \( v \) be partitioning item \( a[10] \).
- Scan \( i \) from left to right.
  - (\( a[i] < v \)): exchange \( a[lt] \) with \( a[i] \) and increment both \( lt \) and \( i \)
  - (\( a[i] > v \)): exchange \( a[gt] \) with \( a[i] \) and decrement \( gt \)
  - (\( a[i] == v \)): increment \( i \)

\[
\begin{array}{cccccccccccc}
\end{array}
\]

**Invariant**

\[
< V \quad = V \quad \text{[\( \_ \_ \_ \_ \] \( \_ \_ \_ \_ \]} \quad > V \\
\]

\[
\begin{array}{c}
\uparrow \\
\uparrow \\
\uparrow \\
\end{array}
\]

\[
\begin{array}{c}
lt \quad i \quad gt \\
\end{array}
\]
Let $v$ be partitioning item $a[10]$.

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

Dijkstra 3-way partitioning

\[\begin{array}{cccccccccccccc}
\end{array}\]
Dijkstra 3-way partitioning

- Let $v$ be partitioning item $a[10]$.
- Scan $i$ from left to right.
  - $(a[i] < v)$: exchange $a[lt]$ with $a[i]$ and increment both $lt$ and $i$
  - $(a[i] > v)$: exchange $a[gt]$ with $a[i]$ and decrement $gt$
  - $(a[i] == v)$: increment $i$
Dijkstra 3-way partitioning

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

```
<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>P</th>
<th>P</th>
<th>P</th>
<th>P</th>
<th>P</th>
<th>P</th>
<th>V</th>
<th>W</th>
<th>Y</th>
<th>Z</th>
<th>X</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></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>
```

invariant

```
<\_ | =\_ | ____________ | >\_ \\
lt | i  | gt
```
Dijkstra 3-way partitioning

- Let \( v \) be partitioning item \( a[lo] \).
- Scan \( i \) from left to right.
  - \( (a[i] < v) \): exchange \( a[lt] \) with \( a[i] \) and increment both \( lt \) and \( i \)
  - \( (a[i] > v) \): exchange \( a[gt] \) with \( a[i] \) and decrement \( gt \)
  - \( (a[i] == v) \): increment \( i \)
Dijkstra 3-way partitioning algorithm

3-way partitioning.
• Let $v$ be partitioning item $a[lo]$.
• Scan $i$ from left to right.
  - $a[i]$ less than $v$: exchange $a[lt]$ with $a[i]$ and increment both $lt$ and $i$
  - $a[i]$ greater than $v$: exchange $a[gt]$ with $a[i]$ and decrement $gt$
  - $a[i]$ equal to $v$: increment $i$

Most of the right properties.
• In-place.
• Not much code.
• Linear time if keys are all equal.
Dijkstra's 3-way partitioning: trace

3-way partitioning trace (array contents after each loop iteration)
3-way quicksort: Java implementation

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
## Sorting summary

<table>
<thead>
<tr>
<th>inplace?</th>
<th>stable?</th>
<th>worst</th>
<th>average</th>
<th>best</th>
<th>remarks</th>
</tr>
</thead>
<tbody>
<tr>
<td>selection</td>
<td>✔️</td>
<td>$N^2/2$</td>
<td>$N^2/2$</td>
<td>$N^2/2$</td>
<td>$N$ exchanges</td>
</tr>
<tr>
<td>insertion</td>
<td>✔️</td>
<td>✔️</td>
<td>$N^2/2$</td>
<td>$N^2/4$</td>
<td>$N$ use for small $N$ or partially ordered</td>
</tr>
<tr>
<td>shell</td>
<td>✔️</td>
<td>?</td>
<td>?</td>
<td>$N$</td>
<td>tight code, subquadratic</td>
</tr>
<tr>
<td>merge</td>
<td>✔️</td>
<td>✔️</td>
<td>$N \log N$</td>
<td>$N \log N$</td>
<td>$N \log N$ guarantee, stable</td>
</tr>
<tr>
<td>quick</td>
<td>✔️</td>
<td>✔️</td>
<td>$N^2/2$</td>
<td>$2N \ln N$</td>
<td>$N \log N$ probabilistic guarantee fastest in practice</td>
</tr>
<tr>
<td>3-way quick</td>
<td>✔️</td>
<td>✔️</td>
<td>$N^2/2$</td>
<td>$2N \ln N$</td>
<td>$N$ improves quicksort in presence of duplicate keys</td>
</tr>
<tr>
<td>???</td>
<td>✔️</td>
<td>✔️</td>
<td>$N \log N$</td>
<td>$N \log N$</td>
<td>holy sorting grail</td>
</tr>
</tbody>
</table>