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

<table>
<thead>
<tr>
<th>input</th>
<th>Q U I C K S O R T E X A M P L E</th>
</tr>
</thead>
<tbody>
<tr>
<td>shuffle</td>
<td>K R A T E L E P U I M Q C X O S</td>
</tr>
<tr>
<td>partition</td>
<td>E C A I E K L P U T M Q R X O S</td>
</tr>
<tr>
<td>sort left</td>
<td>A C E E I K L P U T M Q R X O S</td>
</tr>
<tr>
<td>sort right</td>
<td>A C E E I K L M O P Q R S T U X</td>
</tr>
<tr>
<td>result</td>
<td>A C E E I K L M O P Q R S T U X</td>
</tr>
</tbody>
</table>
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 \( a[j] \).

\[ \text{stop } i \text{ scan because } a[i] \geq a[lo] \]
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 $a[j]$.

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 \( a[j] \).

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

↑

\[ i \]

↑

\[ j \]
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 \(j\) 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 $a[j]$.

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

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 $a[j]$.

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 \( a[j] \).

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

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 $a[j]$.

\[ \begin{array}{cccccccccccccccc}
K & C & A & I & E & E & L & P & U & T & M & Q & R & X & O & S \\
\uparrow & \uparrow & \uparrow & \uparrow & \uparrow & \uparrow & \uparrow & \uparrow & \uparrow & \uparrow & \uparrow & \uparrow & \uparrow & \uparrow & \uparrow & \uparrow \\
lo & j & i & \text{stop } j \text{ scan because } a[j] \leq a[lo] \end{array} \]
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.
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!
Quicksort 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)
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**

- \( a[lo] \)
- \( a[hi] \)

**During**

- Find item on left to swap
- Find item on right to swap
- Check if pointers cross

**After**

- Swap with partitioning item
- Return index of item now known to be in place
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>15</td>
<td>15</td>
</tr>
<tr>
<td>0</td>
<td>4</td>
<td>15</td>
</tr>
<tr>
<td>0</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>15</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>8</td>
<td>15</td>
</tr>
<tr>
<td>8</td>
<td>8</td>
<td>15</td>
</tr>
<tr>
<td>10</td>
<td>15</td>
<td>15</td>
</tr>
<tr>
<td>10</td>
<td>12</td>
<td>15</td>
</tr>
<tr>
<td>10</td>
<td>11</td>
<td>15</td>
</tr>
<tr>
<td>10</td>
<td>10</td>
<td>15</td>
</tr>
<tr>
<td>14</td>
<td>15</td>
<td>15</td>
</tr>
<tr>
<td>15</td>
<td>15</td>
<td>15</td>
</tr>
</tbody>
</table>

Quicksort trace (array contents after each partition)
Quicksort animation

50 random items

http://www.sorting-algorithms.com/quick-sort
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 \((i == h)\) 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></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>computer</td>
<td>thousand</td>
<td>million</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.
**Best case. Number of compares is $\sim N \lg N$.**

Each partitioning process splits the array exactly in half.

<table>
<thead>
<tr>
<th></th>
<th>a[]</th>
</tr>
</thead>
<tbody>
<tr>
<td>lo</td>
<td>j</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>initial values</td>
<td>H</td>
</tr>
<tr>
<td>random shuffle</td>
<td>H</td>
</tr>
<tr>
<td>0</td>
<td>7</td>
</tr>
<tr>
<td>0</td>
<td>3</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>6</td>
<td>6</td>
</tr>
<tr>
<td>8</td>
<td>11</td>
</tr>
<tr>
<td>8</td>
<td>9</td>
</tr>
<tr>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>10</td>
<td>10</td>
</tr>
<tr>
<td>12</td>
<td>13</td>
</tr>
<tr>
<td>12</td>
<td>12</td>
</tr>
<tr>
<td>14</td>
<td>14</td>
</tr>
</tbody>
</table>

**a[]**

- The table shows the array `a[]` with elements from `A` to `O`.
- The table is divided into `lo`, `j`, and `hi` columns, indicating the range of the array.
- `lo`, `j`, and `hi` values are used to split the array approximately in half, as shown in the `random shuffle` row.
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:

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

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

• Finally, the desired result:

\[
C_N \approx 2(N+1) \ln N \approx 1.39N \lg N
\]
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 \( \sim N \log N. \)
- 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 quadratic if array
- Is sorted or reverse sorted.
- Has many duplicates (even if randomized!)
**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>B_1</td>
<td>C_1</td>
<td>C_2</td>
<td>A_1</td>
</tr>
<tr>
<td>1</td>
<td>3</td>
<td>B_1</td>
<td>C_1</td>
<td>C_2</td>
<td>A_1</td>
</tr>
<tr>
<td>1</td>
<td>3</td>
<td>B_1</td>
<td>A_1</td>
<td>C_2</td>
<td>C_1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>A_1</td>
<td>B_1</td>
<td>C_2</td>
<td>C_1</td>
</tr>
</tbody>
</table>

can guarantee logarithmic depth by recurring on smaller subarray before larger subarray.
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);
}
```
Median of sample.

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

```java
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
class QuickSelect {
    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: 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:

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

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

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

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

Recommended. Stop scans on items equal to the partitioning item.

Consequence. \( \sim N \log N \) compares when all keys equal.

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

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

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

### 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>

\[lo\] \[hi\]
• 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 \( i \)
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 $i$

![Partitioning Diagram]

**Invariant**

<table>
<thead>
<tr>
<th>&lt;v</th>
<th>=v</th>
<th>gt</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 $a[lo]$.
- 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 $i$

**invariant**

$\begin{array}{|c|c|c|c|c|c|c|c|c|c|c|c|}
\hline
< v & = v & & & > v \\
\hline
lt & i & gt \\
\hline
\end{array}$
Dijkstra 3-way partitioning

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

 invariant

\[
\begin{array}{cccc}
< v & = v & [\text{space}] & > v \\
\downarrow & \uparrow & \downarrow & \uparrow \\
l_t & i & g_t & \end{array}
\]
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 \( i \)

 invariant
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 $i$

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

\[
\text{invariant}
\]

\[
\begin{array}{c|c|c|c}
< v & = v & \_ & \_ & > v \\
\hline
lt & i & gt
\end{array}
\]
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 \( i \)

\[
\begin{array}{cccccccccc}
  & & & & \text{lt} & & i & & \text{gt} \\
\end{array}
\]

\[
\begin{array}{cccccc}
  \text{invariant} & \text{\( <v \)} & \text{\( =v \)} & \text{\( \_\_\_\_\_ \)} & \text{\( >v \)} \\
  \text{\( \text{lt} \)} & \text{\( i \)} & \text{\( \text{gt} \)}
\end{array}
\]
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 $i$

**Invariant**

\[
\begin{array}{c|c|c|c}
< v & = v & \text{gray} & > v \\
\uparrow & \uparrow & \uparrow & \uparrow \\
lt & i & gt &
\end{array}
\]
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 $i$

![Diagram showing partitioning process]

**Invariant**

- $<v$
- $=v$
- $\ldots$
- $>v$
- $lt$
- $i$
- $gt$
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 $a[lo]$.

• 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 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)
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: Java implementation
3-way quicksort: visual trace
## Sorting summary

<table>
<thead>
<tr>
<th></th>
<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></td>
<td>N² / 2</td>
<td>N² / 2</td>
<td>N² / 2</td>
<td>N exchanges</td>
</tr>
<tr>
<td>insertion</td>
<td>✔ ✔</td>
<td>✔</td>
<td>N² / 2</td>
<td>N² / 4</td>
<td>N</td>
<td>use for small N or partially ordered</td>
</tr>
<tr>
<td>shell</td>
<td>✔</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 lg N</td>
<td>N lg N</td>
<td>N lg N</td>
<td>N log N guarantee, stable</td>
</tr>
<tr>
<td>quick</td>
<td>✔</td>
<td>✔</td>
<td>N² / 2</td>
<td>N lg N</td>
<td>N lg N</td>
<td>N log N probabilistic guarantee</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>fastest in practice</td>
</tr>
<tr>
<td>3-way quick</td>
<td>✔</td>
<td>✔</td>
<td>N² / 2</td>
<td>N lg N</td>
<td>N</td>
<td>improves quicksort in presence of duplicate keys</td>
</tr>
<tr>
<td>???</td>
<td>✔ ✔</td>
<td>✔</td>
<td>N lg N</td>
<td>N lg N</td>
<td>N lg N</td>
<td>holy sorting grail</td>
</tr>
</tbody>
</table>