**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.

**Shuffling**

**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)

**Quick sort 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 R A T E L E P U I M Q C X O S
  ↑  ↑  ↑
↑ i   j
```

---

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

---

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

---

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

\begin{tabular}{cccccccc}
K & C & A & T & E & L & E & P
\end{tabular}

\begin{tabular}{c}
\uparrow
\end{tabular}

\begin{tabular}{c}
\downarrow
\end{tabular}

\begin{tabular}{c}
\downarrow
\end{tabular}

\begin{tabular}{c}
i
\end{tabular}

\begin{tabular}{c}
j
\end{tabular}

\begin{tabular}{c}
\uparrow
\end{tabular}

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 \(a[j]\).

stop j scan and 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]`.

When pointers cross.


 pointers cross: exchange `a[lo]` with `a[j]`
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.

```java
private static int partition(Comparable[] a, int lo, int hi) {
    int i = lo, j = hi;
    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);
    }
    return j;
}
```

Quicksort: Java code for partitioning

```java
public static void sort(Comparable[] a) {
    int lo = 0, hi = a.length - 1;
    StdRandom.shuffle(a);
    sort(a, lo, hi);
    swap(a, lo, hi);
}
```

Quicksort trace (array contents before and after each exchange)

<table>
<thead>
<tr>
<th>Initial Values</th>
<th>after exchange 1</th>
<th>after exchange 2</th>
<th>after exchange 3</th>
<th>after exchange 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15</td>
<td>K B E M Q C X O S</td>
<td>T E M Q C X O S</td>
<td>E Q C X O S</td>
<td>C X O S</td>
</tr>
</tbody>
</table>

Quicksort trace (array contents after each partition)

```java
private static int partition(Comparable[] a, int lo, int hi) {
    int i = lo, j = hi;
    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);
    }
    return j;
}
```

Quicksort: Java implementation

```java
public class Quick {
    private static int partition(Comparable[] a, int lo, int hi) {
        // see previous slide */
        return partition(a, lo, hi);
    }
    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, j+1, hi);
        sort(a, lo, j-1);
    }
}
```

Quicksort trace (array contents after each partition)
Quick sort animation

http://www.sorting-algorithms.com/quick-sort

Quick sort 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 \((j = l)\) 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.

Quick sort: empirical analysis

Running time estimates:

- Home PC executes \(10^8\) compares/second.
- Supercomputer executes \(10^{12}\) compares/second.

<table>
<thead>
<tr>
<th>insertion sort (N)</th>
<th>merge sort (N \log N)</th>
<th>quicksort (N \log N)</th>
</tr>
</thead>
<tbody>
<tr>
<td>computer</td>
<td>million</td>
<td>billion</td>
</tr>
<tr>
<td>home</td>
<td>instant</td>
<td>2.8 hours</td>
</tr>
<tr>
<td>super</td>
<td>instant</td>
<td>1 second</td>
</tr>
</tbody>
</table>

Lesson 1. Good algorithms are better than supercomputers.
Lesson 2. Great algorithms are better than good ones.

Quick sort: best-case analysis

Best case. Number of compares is \(\sim N \log N\).

Each partitioning process splits the array exactly in half.

<table>
<thead>
<tr>
<th>le j</th>
<th>n</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></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></td>
</tr>
<tr>
<td>random shuffle</td>
<td></td>
<td>A</td>
<td>C</td>
<td>B</td>
<td>E</td>
<td>F</td>
<td>G</td>
<td>D</td>
<td>L</td>
<td>I</td>
<td>K</td>
<td>J</td>
<td>N</td>
<td>M</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>7</td>
<td>14</td>
<td>D</td>
<td>A</td>
<td>C</td>
<td>B</td>
<td>F</td>
<td>E</td>
<td>G</td>
<td>H</td>
<td>L</td>
<td>I</td>
<td>K</td>
<td>J</td>
<td>N</td>
</tr>
<tr>
<td>0</td>
<td>3</td>
<td>6</td>
<td>E</td>
<td>A</td>
<td>C</td>
<td>B</td>
<td>F</td>
<td>E</td>
<td>G</td>
<td>H</td>
<td>L</td>
<td>I</td>
<td>K</td>
<td>J</td>
<td>N</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>A</td>
<td>E</td>
<td>C</td>
<td>F</td>
<td>D</td>
<td>G</td>
<td>H</td>
<td>I</td>
<td>K</td>
<td>J</td>
<td>L</td>
<td>N</td>
<td>M</td>
</tr>
<tr>
<td>0</td>
<td>3</td>
<td>6</td>
<td>E</td>
<td>A</td>
<td>C</td>
<td>B</td>
<td>F</td>
<td>E</td>
<td>G</td>
<td>H</td>
<td>L</td>
<td>I</td>
<td>K</td>
<td>J</td>
<td>N</td>
</tr>
<tr>
<td>4</td>
<td>5</td>
<td>6</td>
<td>A</td>
<td>E</td>
<td>C</td>
<td>F</td>
<td>D</td>
<td>G</td>
<td>H</td>
<td>I</td>
<td>K</td>
<td>J</td>
<td>L</td>
<td>N</td>
<td>M</td>
</tr>
<tr>
<td>4</td>
<td>5</td>
<td>6</td>
<td>A</td>
<td>E</td>
<td>C</td>
<td>F</td>
<td>D</td>
<td>G</td>
<td>H</td>
<td>I</td>
<td>K</td>
<td>J</td>
<td>L</td>
<td>N</td>
<td>M</td>
</tr>
<tr>
<td>6</td>
<td>5</td>
<td>6</td>
<td>A</td>
<td>E</td>
<td>C</td>
<td>F</td>
<td>D</td>
<td>G</td>
<td>H</td>
<td>I</td>
<td>K</td>
<td>J</td>
<td>L</td>
<td>N</td>
<td>M</td>
</tr>
<tr>
<td>6</td>
<td>5</td>
<td>6</td>
<td>A</td>
<td>E</td>
<td>C</td>
<td>F</td>
<td>D</td>
<td>G</td>
<td>H</td>
<td>I</td>
<td>K</td>
<td>J</td>
<td>L</td>
<td>N</td>
<td>M</td>
</tr>
<tr>
<td>8</td>
<td>11</td>
<td>14</td>
<td>A</td>
<td>E</td>
<td>C</td>
<td>F</td>
<td>G</td>
<td>H</td>
<td>I</td>
<td>J</td>
<td>K</td>
<td>L</td>
<td>N</td>
<td>M</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>11</td>
<td>14</td>
<td>A</td>
<td>E</td>
<td>C</td>
<td>F</td>
<td>G</td>
<td>H</td>
<td>I</td>
<td>J</td>
<td>K</td>
<td>L</td>
<td>N</td>
<td>M</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>11</td>
<td>14</td>
<td>A</td>
<td>E</td>
<td>C</td>
<td>F</td>
<td>G</td>
<td>H</td>
<td>I</td>
<td>J</td>
<td>K</td>
<td>L</td>
<td>N</td>
<td>M</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>10</td>
<td>13</td>
<td>A</td>
<td>E</td>
<td>C</td>
<td>F</td>
<td>G</td>
<td>H</td>
<td>I</td>
<td>J</td>
<td>K</td>
<td>L</td>
<td>N</td>
<td>M</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>10</td>
<td>13</td>
<td>A</td>
<td>E</td>
<td>C</td>
<td>F</td>
<td>G</td>
<td>H</td>
<td>I</td>
<td>J</td>
<td>K</td>
<td>L</td>
<td>N</td>
<td>M</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>10</td>
<td>13</td>
<td>A</td>
<td>E</td>
<td>C</td>
<td>F</td>
<td>G</td>
<td>H</td>
<td>I</td>
<td>J</td>
<td>K</td>
<td>L</td>
<td>N</td>
<td>M</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>10</td>
<td>13</td>
<td>A</td>
<td>E</td>
<td>C</td>
<td>F</td>
<td>G</td>
<td>H</td>
<td>I</td>
<td>J</td>
<td>K</td>
<td>L</td>
<td>N</td>
<td>M</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>10</td>
<td>13</td>
<td>A</td>
<td>E</td>
<td>C</td>
<td>F</td>
<td>G</td>
<td>H</td>
<td>I</td>
<td>J</td>
<td>K</td>
<td>L</td>
<td>N</td>
<td>M</td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>13</td>
<td>14</td>
<td>A</td>
<td>E</td>
<td>C</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></td>
</tr>
<tr>
<td>12</td>
<td>13</td>
<td>14</td>
<td>A</td>
<td>E</td>
<td>C</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></td>
</tr>
<tr>
<td>12</td>
<td>13</td>
<td>14</td>
<td>A</td>
<td>E</td>
<td>C</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></td>
</tr>
<tr>
<td>12</td>
<td>13</td>
<td>14</td>
<td>A</td>
<td>E</td>
<td>C</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></td>
</tr>
<tr>
<td>12</td>
<td>13</td>
<td>14</td>
<td>A</td>
<td>E</td>
<td>C</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></td>
</tr>
<tr>
<td>14</td>
<td>14</td>
<td>14</td>
<td>A</td>
<td>E</td>
<td>C</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></td>
</tr>
<tr>
<td>14</td>
<td>14</td>
<td>14</td>
<td>A</td>
<td>E</td>
<td>C</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></td>
</tr>
</tbody>
</table>
Quicksort: worst-case analysis

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

One of the subarrays is empty for every partition.

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

Quicksort: summary of performance characteristics

Worst case. Number of compares is quadratic.

- $N + (N - 1) + (N - 2) + \ldots + 1 - \frac{1}{2} N^2$.
- 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!)

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.

Quicksort: practical improvements

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.

```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)

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?
- $N$ upper bound?
- Is selection as hard as sorting?
- 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];
}
```
Quick-select: mathematical analysis

**Proposition.** Quick-select takes linear time on average.

**Pf sketch.**
- Intuitively, each partitioning step splits array approximately in half:
  \[ N + N/2 + N/4 + \ldots + 1 = 2N \] compares.
- Formal analysis similar to quicksort analysis yields:
  \[ C_N = 2N + k \ln (N/k) + (N-k) \ln (N/(N-k)) \]
  \[ (2 + 2 \ln 2)N \] 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.

### 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()`.

**Mistake.** Put all items equal to the partitioning item on one side.

**Consequence.** \( \sim \frac{1}{2} N^2 \) compares when all keys equal.

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

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

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

```plaintext
before

<table>
<thead>
<tr>
<th>lt</th>
<th>v</th>
<th>gt</th>
</tr>
</thead>
</table>

after

<table>
<thead>
<tr>
<th>lt</th>
<th>v</th>
<th>v</th>
<th>gt</th>
</tr>
</thead>
</table>
```

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`.

```plaintext
invariant

<table>
<thead>
<tr>
<th>lt</th>
<th>v</th>
<th>v</th>
<th>gt</th>
</tr>
</thead>
</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`.

```plaintext
invariant

<table>
<thead>
<tr>
<th>lt</th>
<th>v</th>
<th>v</th>
<th>gt</th>
</tr>
</thead>
</table>
```
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 \( i \)

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

**Invariant**

\[<v \quad \equiv \quad >v\]

\[\begin{array}{cccccccc}
\lt & i & \gtf\n\end{array}\]

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 \( i \)

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

**Invariant**

\[<v \quad \equiv \quad >v\]

\[\begin{array}{cccccccc}
\lt & i & \gtf\n\end{array}\]
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:
  - $a[i] < v$: 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 $i$.

Most of the right properties:
- In-place.
- Not much code.
- Linear time if keys are all equal.

3-way quicksort: Java implementation

```java
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);
}
```

Dijkstra's 3-way partitioning: trace

Visual trace of quicksort with 3-way partitioning

3-way quicksort: visual trace

3-way partitioning trace (array contents after each loop iteration)
# 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>N / 2</td>
<td>N / 2</td>
<td>N / 2</td>
<td>N exchanges</td>
<td></td>
</tr>
<tr>
<td>insertion</td>
<td>✔️ ✔️</td>
<td>N / 2</td>
<td>N / 4</td>
<td>N</td>
<td>use for small N or partially ordered</td>
<td></td>
</tr>
<tr>
<td>shell</td>
<td>✔️</td>
<td>?</td>
<td>?</td>
<td>N</td>
<td>tight code, subquadratic</td>
<td></td>
</tr>
<tr>
<td>merge</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>
<td></td>
</tr>
<tr>
<td>quick</td>
<td>✔️</td>
<td>N / 2</td>
<td>N lg N</td>
<td>N lg N</td>
<td>N log N probabilistic guarantee fast in practice</td>
<td></td>
</tr>
<tr>
<td>3-way quick</td>
<td>✔️ ✔️</td>
<td>N / 2</td>
<td>N lg N</td>
<td>N</td>
<td>improves quicksort in presence of duplicate keys</td>
<td></td>
</tr>
<tr>
<td>???</td>
<td>✔️ ✔️</td>
<td>N lg N</td>
<td>N lg N</td>
<td>N lg N</td>
<td>holy sorting grail</td>
<td></td>
</tr>
</tbody>
</table>