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>not greater</td>
<td></td>
</tr>
<tr>
<td>not less</td>
<td></td>
</tr>
<tr>
<td>partitioning item</td>
<td></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

- 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].

\[ K \] \[ R \] \[ A \] \[ T \] \[ E \] \[ L \] \[ E \] \[ P \] \[ U \] \[ I \] \[ M \] \[ Q \] \[ C \] \[ X \] \[ O \] \[ S \]

\[ \uparrow \] \[ \uparrow \]

\[ \text{lo} \] \[ i \] \[ j \]

stop i scan because a[i] >= a[lo]
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 \( a[j] \).
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 $a[j]$. 

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

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 \( a[j] \).

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

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 $a[j]$.  

\[
\begin{array}{cccccccccc}
K & C & A & I & E & L & E & P & U & T \\
\hline
\uparrow & \uparrow & \uparrow & lo & i & j & M & Q & R & X \\
\end{array}
\]
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 $a[j]$.

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

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 $a[j]$.

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 $a[j]$.

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.


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

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

        if (i >= j) break;  // check if pointers cross
        exch(a, i, j);  // swap
    }
    exch(a, lo, j);  // swap with partitioning item
    return j;  // 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);
    }
}

shuffle needed for performance guarantee (stay tuned)
Quicksort trace

Quicksort trace (array contents after each partition)
Quicksort animation

50 random items

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

Partitioning in-place. Using an extra array makes partitioning easier (and stable), but 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 \((j == lo)\) 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></th>
<th>insertion sort ($N^2$)</th>
<th>mergesort ($N \log N$)</th>
<th>quicksort ($N \log N$)</th>
</tr>
</thead>
<tbody>
<tr>
<td>computer</td>
<td>thousand</td>
<td>million</td>
<td>billion</td>
</tr>
<tr>
<td>home</td>
<td>instant</td>
<td>2.8 hours</td>
<td>317 years</td>
</tr>
<tr>
<td>super</td>
<td>instant</td>
<td>1 second</td>
<td>1 week</td>
</tr>
</tbody>
</table>

**Lesson 1.** Good algorithms are better than supercomputers.
**Lesson 2.** Great algorithms are better than good ones.
**Best case. Number of compares is $\sim N \lg N$.**

Each partitioning process splits the array exactly in half.
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.
Quicksort: summary of performance characteristics

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

Average case. Number of compares is \(\sim N \lg N.\)
- 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 \textit{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>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>
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.

\[ \approx \frac{12}{7} \cdot N \ln N \text{ compares (slightly fewer)} \]
\[ \approx \frac{12}{35} \cdot N \ln N \text{ exchanges (slightly more)} \]

```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);
}
```
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];
}
```
Quick-select: mathematical analysis

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

**Pf sketch.**
- Intuitively, each partitioning step splits array approximately in half:
  \( N + N/2 + N/4 + \ldots + 1 \sim 2N \) 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 \( \frac{1}{2}N^2 \) 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.

```
STOP ONE EQUAL KEYS
```

- 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.

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

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

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

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

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

\[
\begin{array}{cccccccc}
C & C & C & C & C & C & C & C
\end{array}
\quad
\begin{array}{cccccccc}
\end{array}
\]
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 \texttt{qsort()}.
- Now incorporated into \texttt{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 \( i \)
• 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 $i$

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

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

**invariant**

\[
\begin{array}{cccc}
< v & = v & \text{[gray]} & > v \\
\uparrow & \uparrow & \uparrow & \downarrow \\
\text{lt} & i & \text{gt} & \\
\end{array}
\]
• 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

- 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 $i$

### invariant

```
< v  = v  P  = v  > v
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 $i$

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

\[
\begin{array}{cccc}
l_t & i & g_t \\
\text{\downarrow} & \text{\downarrow} & \text{\downarrow} \\
\end{array}
\]

\[
\begin{array}{cccc}
< v & = v & \text{blank} & > v \\
\uparrow & \uparrow & \uparrow & \uparrow \\
l_t & i & g_t \\
\end{array}
\]

\textbf{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 $i$
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 $i$

![Diagram showing the partitioning process]

**Invariant**

```
< V  = V  [ ]  > V
  |   |   |
  lt  i  gt
```
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 \( i \)

\[\begin{array}{ccccccccccccccc}
  lt & \downarrow & i & \downarrow & gt & \downarrow
\end{array}\]

invariant

\[\begin{array}{cccc}
  <v & =v & \text{bar} & >v \\
  lt & i & gt
\end{array}\]
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 \( i \)
• 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

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

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

\text{invariant}

\[\begin{array}{cccc}
< v & = v & \text{[gray]} & > v \\
\downarrow & \uparrow & \downarrow & \uparrow \\
lt & i & gt \\
\end{array}\]
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 \( i \)
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

equal to partitioning element
## 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>$N^2/2$</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>$N$</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>$N^2/2$</td>
<td>$N \lg N$</td>
<td>$N \lg N$</td>
<td>$N \log N$ probabilistic guarantee</td>
</tr>
<tr>
<td>3-way quick</td>
<td>✔</td>
<td></td>
<td>$N^2/2$</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>