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)

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.

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

Quickstort 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 $a[j]$.

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 \(a[j]\).

\[\begin{array}{cccccccc}
K & R & A & T & E & L & E & P
\end{array}\]

\[\begin{array}{cccccccc}
U & I & M & Q & C & X & O & S
\end{array}\]

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

\[\begin{array}{cccccccc}
lo & i & j & lo & i & j
\end{array}\]
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].

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 \( a[j] \).

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

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 \( a[j] \).

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

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 $a[j]$.

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

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

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

stop \( i \) scan because \( a[i] \geq a[lo] \)

When pointers cross.

- Exchange \( a[lo] \) with \( a[j] \).

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

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

```
// Quicksort trace

Before
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15

During
12 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
1 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15

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

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;
        exch(a, i, j);
        if (i >= j) break;
    }
    return j;
}
```

```
// Quicksort trace

Before
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15

During
12 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
1 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15

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

Quicksort trace

```
// Quicksort trace

Before
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15

During
12 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
1 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15

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

Quicksort: Java implementation

```java
public class Quick {
    private static int partition(Comparable[] a, int lo, int hi) {
        /* see previous slide */
        return index of item now known to be in place
    }
    public static void sort(Comparable[] a) {
        sort(a, 0, a.length - 1);
        StdRandom.shuffle(a);
    }
    private static int partition(Comparable[] a, int lo, int hi) {
        if (hi <= lo) return;
        int j = partition(a, lo, hi);
        sort(a, 0, j - 1);
        sort(a, j+1, hi);
    }
}
```

```
// Quicksort trace

Before
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15

During
12 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
1 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15

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

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>Computer</th>
<th>Insertion sort (N)</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 \(\sim N \log N\).
Each partitioning process splits the array exactly in half.

<table>
<thead>
<tr>
<th>n!</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
<th>14</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>A</td>
<td>B</td>
<td>C</td>
<td>D</td>
<td>E</td>
<td>F</td>
<td>G</td>
<td>H</td>
<td>I</td>
<td>J</td>
<td>K</td>
<td>L</td>
<td>M</td>
<td>N</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>A</td>
<td>B</td>
<td>C</td>
<td>D</td>
<td>E</td>
<td>F</td>
<td>G</td>
<td>H</td>
<td>I</td>
<td>J</td>
<td>K</td>
<td>L</td>
<td>M</td>
<td>N</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>A</td>
<td>B</td>
<td>C</td>
<td>D</td>
<td>E</td>
<td>F</td>
<td>G</td>
<td>H</td>
<td>I</td>
<td>J</td>
<td>K</td>
<td>L</td>
<td>M</td>
<td>N</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>A</td>
<td>B</td>
<td>C</td>
<td>D</td>
<td>E</td>
<td>F</td>
<td>G</td>
<td>H</td>
<td>I</td>
<td>J</td>
<td>K</td>
<td>L</td>
<td>M</td>
<td>N</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>A</td>
<td>B</td>
<td>C</td>
<td>D</td>
<td>E</td>
<td>F</td>
<td>G</td>
<td>H</td>
<td>I</td>
<td>J</td>
<td>K</td>
<td>L</td>
<td>M</td>
<td>N</td>
</tr>
<tr>
<td>5</td>
<td>5</td>
<td>A</td>
<td>B</td>
<td>C</td>
<td>D</td>
<td>E</td>
<td>F</td>
<td>G</td>
<td>H</td>
<td>I</td>
<td>J</td>
<td>K</td>
<td>L</td>
<td>M</td>
<td>N</td>
</tr>
<tr>
<td>6</td>
<td>6</td>
<td>A</td>
<td>B</td>
<td>C</td>
<td>D</td>
<td>E</td>
<td>F</td>
<td>G</td>
<td>H</td>
<td>I</td>
<td>J</td>
<td>K</td>
<td>L</td>
<td>M</td>
<td>N</td>
</tr>
<tr>
<td>7</td>
<td>7</td>
<td>A</td>
<td>B</td>
<td>C</td>
<td>D</td>
<td>E</td>
<td>F</td>
<td>G</td>
<td>H</td>
<td>I</td>
<td>J</td>
<td>K</td>
<td>L</td>
<td>M</td>
<td>N</td>
</tr>
<tr>
<td>8</td>
<td>8</td>
<td>A</td>
<td>B</td>
<td>C</td>
<td>D</td>
<td>E</td>
<td>F</td>
<td>G</td>
<td>H</td>
<td>I</td>
<td>J</td>
<td>K</td>
<td>L</td>
<td>M</td>
<td>N</td>
</tr>
<tr>
<td>9</td>
<td>9</td>
<td>A</td>
<td>B</td>
<td>C</td>
<td>D</td>
<td>E</td>
<td>F</td>
<td>G</td>
<td>H</td>
<td>I</td>
<td>J</td>
<td>K</td>
<td>L</td>
<td>M</td>
<td>N</td>
</tr>
<tr>
<td>10</td>
<td>10</td>
<td>A</td>
<td>B</td>
<td>C</td>
<td>D</td>
<td>E</td>
<td>F</td>
<td>G</td>
<td>H</td>
<td>I</td>
<td>J</td>
<td>K</td>
<td>L</td>
<td>M</td>
<td>N</td>
</tr>
<tr>
<td>11</td>
<td>11</td>
<td>A</td>
<td>B</td>
<td>C</td>
<td>D</td>
<td>E</td>
<td>F</td>
<td>G</td>
<td>H</td>
<td>I</td>
<td>J</td>
<td>K</td>
<td>L</td>
<td>M</td>
<td>N</td>
</tr>
<tr>
<td>12</td>
<td>12</td>
<td>A</td>
<td>B</td>
<td>C</td>
<td>D</td>
<td>E</td>
<td>F</td>
<td>G</td>
<td>H</td>
<td>I</td>
<td>J</td>
<td>K</td>
<td>L</td>
<td>M</td>
<td>N</td>
</tr>
<tr>
<td>13</td>
<td>13</td>
<td>A</td>
<td>B</td>
<td>C</td>
<td>D</td>
<td>E</td>
<td>F</td>
<td>G</td>
<td>H</td>
<td>I</td>
<td>J</td>
<td>K</td>
<td>L</td>
<td>M</td>
<td>N</td>
</tr>
<tr>
<td>14</td>
<td>14</td>
<td>A</td>
<td>B</td>
<td>C</td>
<td>D</td>
<td>E</td>
<td>F</td>
<td>G</td>
<td>H</td>
<td>I</td>
<td>J</td>
<td>K</td>
<td>L</td>
<td>M</td>
<td>N</td>
</tr>
</tbody>
</table>
Quicksort: worst-case analysis

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

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

One of the subarrays is empty for every partition.

Quicksort: average-case analysis

Proposition. The average number of compares $C_N$ to quicksort an array of $N$ distinct keys is $\sim 2N \ln N$ (and the number of exchanges is $\sim \frac{1}{3} N \ln N$).

Pf. $C_N$ satisfies the recurrence $C_N = C_0 = 0$ and for $N \geq 2$:

$C_N = (N+1) + \left( \frac{C_N + C_N-1}{N} \right) + \left( \frac{C_N + C_N-1}{N} \right) + \ldots + \left( \frac{C_N + C_N-1}{N} \right)$

$\downarrow$ partitioning

$\downarrow$ partitioning probability

$\downarrow$ left

$\downarrow$ right

Multiply both sides by $N$ and collect terms:

$\downarrow$ Subtract this from the same equation for $N-1$:

$\downarrow$ Rearrange terms and divide by $N(N+1)$:

$\frac{C_N}{N+1} = \frac{C_{N-1}}{N} + \frac{2}{N+1}$

Approximate sum by an integral:

$\int_3^{N+1} \frac{1}{x} \, dx$

Finally, the desired result:

$C_N \sim 2(N+1) \ln N \approx 1.39N \ln N$

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 $\sim 1.39 N \lg N$.

• 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 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></th>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>i</td>
<td>j</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>B</td>
<td>C</td>
<td>C</td>
</tr>
<tr>
<td>1</td>
<td>3</td>
<td>B</td>
<td>C</td>
<td>C</td>
</tr>
<tr>
<td>1</td>
<td>3</td>
<td>B</td>
<td>A</td>
<td>C</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>A</td>
<td>B</td>
<td>C</td>
</tr>
</tbody>
</table>

---

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.

\[
\begin{align*}
&\sim 12/7 \text{ N in N compares (slightly fewer)} \\
&\sim 12/35 \text{ N in N exchanges (slightly more)}
\end{align*}
\]

**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) 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. $\text{Min } (k = 0)$, $\text{max } (k = N - 1)$, 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!

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:
  \[ N + N/2 + N/4 + \ldots + 1 \sim 2N \text{ compares.} \]
- Formal analysis similar to quicksort analysis yields:
  \[ C_k = 2N + k \ln (N/k) + (N-k) \ln (N/(N-k)) \]
  \[ (2 + 2 \ln 2)N \text{ to find the median} \]

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


Remark. But, constants are too high $\Rightarrow$ 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.

### Mergesort with duplicate keys.

Always between $\frac{1}{2}N \log N$ and $N \log N$ 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.

### 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.

```
B A A B A B B B C C
A A A A A A A A A A A A
```

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

```
B A A B A B B B C C B C B
A A A A A A A A A A A A
```

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

```
A A A B B B B B B C C
A A A A A A A A A A A A
```

### 3-way partitioning

**Goal.** Partition array into 3 parts so that:
- Entries between $l_e$ and $g_t$ equal to partition item $v$.
- No larger entries to left of $l_e$.
- No smaller entries to right of $g_t$.

**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 \( a[lo] \).
• 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 \( i \)

\[ \]
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


def partition(A, lo, hi, v):
    lt = lo
    i = lo
    gt = hi
    while i <= gt:
        if A[i] < v:
            lt += 1
            i += 1
        elif A[i] > v:
            gt -= 1
        else:
            i += 1

Dijkstra 3-way partitioning


def partition(A, lo, hi, v):
    lt = lo
    i = lo
    gt = hi
    while i <= gt:
        if A[i] < v:
            lt += 1
            i += 1
        elif A[i] > v:
            gt -= 1
        else:
            i += 1

Invariant

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


def partition(A, lo, hi, v):
    lt = lo
    i = lo
    gt = hi
    while i <= gt:
        if A[i] < v:
            lt += 1
            i += 1
        elif A[i] > v:
            gt -= 1
        else:
            i += 1

Dijkstra 3-way partitioning


def partition(A, lo, hi, v):
    lt = lo
    i = lo
    gt = hi
    while i <= gt:
        if A[i] < v:
            lt += 1
            i += 1
        elif A[i] > v:
            gt -= 1
        else:
            i += 1

Invariant
• 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

\[<v \rightarrow v \rightarrow v >v\]

---

Dijkstra 3-way partitioning algorithm

• 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 $i$

3-way partitioning.
• In-place.
• Not much code.
• Linear time if keys are all equal.

Most of the right properties.
3-way partitioning: trace

```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);
}
```

3-way quicksort: Java implementation

```java
3-way partitioning trace (array contents after each loop iteration)
```

3-way quicksort: visual trace

```
Sorting summary
```

```
<table>
<thead>
<tr>
<th></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>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>N</td>
<td>tight code, subquadratic</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>
</tr>
<tr>
<td>quick</td>
<td>✔</td>
<td>N^2 / 2</td>
<td>2 N ln N</td>
<td>N lg N</td>
<td>N log N probabilistic guarantee</td>
</tr>
<tr>
<td>3-way quick</td>
<td>✔</td>
<td>N^2 / 2</td>
<td>2 N ln N</td>
<td>N</td>
<td>improves quicksort in presence of duplicate keys</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>
</tr>
</tbody>
</table>