Two classic sorting algorithms: mergesort and quicksort

Critical components in the world’s computational infrastructure.
- Full scientific understanding of their properties has enabled us to develop them into practical system sorts.
- Quicksort honored as one of top 10 algorithms of 20th century in science and engineering.

Mergesort. [last lecture]

Quicksort. [this lecture]
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 subarray recursively.

Tony Hoare

- Invented quicksort to translate Russian into English.  
  [but couldn't explain his algorithm or implement it!]
- Learned Algol 60 (and recursion).
- Implemented quicksort.

“ There are two ways of constructing a software design: One way is to make it so simple that there are obviously no deficiencies, and the other way is to make it so complicated that there are no obvious deficiencies. The first method is far more difficult. ”

“I call it my billion-dollar mistake. It was the invention of the null reference in 1965... This has led to innumerable errors, vulnerabilities, and system crashes, which have probably caused a billion dollars of pain and damage in the last forty years.”

Bob Sedgewick

- Refined and popularized quicksort.
- Analyzed many versions of quicksort.

Tony Hoare 1980 Turing Award

Algorithms

**ALGORITHM**

```plaintext
ALGORITHM \textsc{QUICKSORT}
\begin{algorithm}
\textbf{Pseudocode:}
\begin{enumerate}
\item \textbf{procedure} \textsc{quicksort} \( A \), \( \text{left} \), \( \text{right} \)
\item \textbf{begin}
\item \hspace{1em} \textbf{if} \( \text{left} < \text{right} \)
\item \hspace{2em} \textbf{then}
\item \hspace{3em} \textbf{partition} \( A\), \( \text{left} \), \( \text{right} \)
\item \hspace{3em} \textbf{end if}
\item \hspace{1em} \textbf{end procedure}
\end{enumerate}
\end{algorithm}
```

Communications of the ACM (July 1961)
Quick sort partitioning demo

**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 \( a[j] \).

---

The music of quick sort partitioning (by Brad Lyon)

https://googledrive.com/host/0B2GQktu-wcTicjRaRjV1NmRFN1U/index.html

---

Quick sort: 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;
}
```

---

When pointers cross.
- Exchange \( a[lo] \) with \( a[j] \).
QuickSort quiz 1

Q. How many compares (in the worst case) to partition an array of length \( N \)?

A. \(~ \frac{1}{4} N\)
B. \(~ \frac{1}{2} N\)
C. \(~ N\)
D. \(~ N \lg N\)
E. I don’t know.

QuickSort trace

<table>
<thead>
<tr>
<th>lo</th>
<th>j</th>
<th>hi</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
<th>14</th>
<th>15</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>5</td>
<td>15</td>
<td>E</td>
<td>C</td>
<td>A</td>
<td>I</td>
<td>E</td>
<td>K</td>
<td>R</td>
<td>A</td>
<td>T</td>
<td>E</td>
<td>L</td>
<td>E</td>
<td>P</td>
<td>U</td>
<td>I</td>
<td>M</td>
</tr>
<tr>
<td>0</td>
<td>3</td>
<td>4</td>
<td>E</td>
<td>C</td>
<td>A</td>
<td>E</td>
<td>I</td>
<td>K</td>
<td>L</td>
<td>P</td>
<td>U</td>
<td>T</td>
<td>M</td>
<td>Q</td>
<td>R</td>
<td>X</td>
<td>O</td>
<td>S</td>
</tr>
<tr>
<td>0</td>
<td>2</td>
<td>2</td>
<td>A</td>
<td>C</td>
<td>E</td>
<td>E</td>
<td>I</td>
<td>K</td>
<td>L</td>
<td>P</td>
<td>U</td>
<td>T</td>
<td>M</td>
<td>Q</td>
<td>R</td>
<td>X</td>
<td>O</td>
<td>S</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>A</td>
<td>C</td>
<td>E</td>
<td>E</td>
<td>I</td>
<td>K</td>
<td>L</td>
<td>P</td>
<td>U</td>
<td>T</td>
<td>M</td>
<td>Q</td>
<td>R</td>
<td>X</td>
<td>O</td>
<td>S</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>A</td>
<td>C</td>
<td>E</td>
<td>E</td>
<td>I</td>
<td>K</td>
<td>L</td>
<td>P</td>
<td>U</td>
<td>T</td>
<td>M</td>
<td>Q</td>
<td>R</td>
<td>X</td>
<td>O</td>
<td>S</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>4</td>
<td>A</td>
<td>C</td>
<td>E</td>
<td>E</td>
<td>I</td>
<td>K</td>
<td>L</td>
<td>P</td>
<td>U</td>
<td>T</td>
<td>M</td>
<td>Q</td>
<td>R</td>
<td>X</td>
<td>O</td>
<td>S</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>6</td>
<td>15</td>
<td>A</td>
<td>C</td>
<td>E</td>
<td>E</td>
<td>I</td>
<td>K</td>
<td>L</td>
<td>P</td>
<td>U</td>
<td>T</td>
<td>M</td>
<td>Q</td>
<td>R</td>
<td>X</td>
<td>O</td>
<td>S</td>
</tr>
<tr>
<td>7</td>
<td>9</td>
<td>15</td>
<td>A</td>
<td>C</td>
<td>E</td>
<td>E</td>
<td>I</td>
<td>K</td>
<td>L</td>
<td>M</td>
<td>O</td>
<td>P</td>
<td>T</td>
<td>Q</td>
<td>R</td>
<td>X</td>
<td>U</td>
<td>S</td>
</tr>
<tr>
<td>7</td>
<td>7</td>
<td>8</td>
<td>A</td>
<td>C</td>
<td>E</td>
<td>E</td>
<td>I</td>
<td>K</td>
<td>L</td>
<td>M</td>
<td>O</td>
<td>P</td>
<td>T</td>
<td>Q</td>
<td>R</td>
<td>X</td>
<td>U</td>
<td>S</td>
</tr>
<tr>
<td>7</td>
<td>8</td>
<td>8</td>
<td>A</td>
<td>C</td>
<td>E</td>
<td>E</td>
<td>I</td>
<td>K</td>
<td>L</td>
<td>M</td>
<td>O</td>
<td>P</td>
<td>T</td>
<td>Q</td>
<td>R</td>
<td>X</td>
<td>U</td>
<td>S</td>
</tr>
<tr>
<td>10</td>
<td>13</td>
<td>15</td>
<td>A</td>
<td>C</td>
<td>E</td>
<td>E</td>
<td>I</td>
<td>K</td>
<td>L</td>
<td>M</td>
<td>O</td>
<td>P</td>
<td>S</td>
<td>Q</td>
<td>R</td>
<td>T</td>
<td>U</td>
<td>X</td>
</tr>
<tr>
<td>10</td>
<td>12</td>
<td>12</td>
<td>A</td>
<td>C</td>
<td>E</td>
<td>E</td>
<td>I</td>
<td>K</td>
<td>L</td>
<td>M</td>
<td>O</td>
<td>P</td>
<td>R</td>
<td>Q</td>
<td>S</td>
<td>T</td>
<td>U</td>
<td>X</td>
</tr>
<tr>
<td>10</td>
<td>11</td>
<td>11</td>
<td>A</td>
<td>C</td>
<td>E</td>
<td>E</td>
<td>I</td>
<td>K</td>
<td>L</td>
<td>M</td>
<td>O</td>
<td>P</td>
<td>Q</td>
<td>R</td>
<td>S</td>
<td>T</td>
<td>U</td>
<td>X</td>
</tr>
<tr>
<td>10</td>
<td>10</td>
<td>10</td>
<td>A</td>
<td>C</td>
<td>E</td>
<td>E</td>
<td>I</td>
<td>K</td>
<td>L</td>
<td>M</td>
<td>O</td>
<td>P</td>
<td>Q</td>
<td>R</td>
<td>S</td>
<td>T</td>
<td>U</td>
<td>X</td>
</tr>
<tr>
<td>14</td>
<td>14</td>
<td>15</td>
<td>A</td>
<td>C</td>
<td>E</td>
<td>E</td>
<td>I</td>
<td>K</td>
<td>L</td>
<td>M</td>
<td>O</td>
<td>P</td>
<td>Q</td>
<td>R</td>
<td>S</td>
<td>T</td>
<td>U</td>
<td>X</td>
</tr>
<tr>
<td>15</td>
<td>15</td>
<td>15</td>
<td>A</td>
<td>C</td>
<td>E</td>
<td>E</td>
<td>I</td>
<td>K</td>
<td>L</td>
<td>M</td>
<td>O</td>
<td>P</td>
<td>Q</td>
<td>R</td>
<td>S</td>
<td>T</td>
<td>U</td>
<td>X</td>
</tr>
</tbody>
</table>

QuickSort trace (array contents after each partition)

QuickSort animation

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 trickier than it might seem.

Equal keys. When duplicates are present, it is (counter-intuitively) better to stop scans on keys equal to the partitioning item’s key.

Preserving randomness. Shuffling is needed for performance guarantee.

Equivalent alternative. Pick a random partitioning item in each subarray.

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>home</td>
<td>thousand</td>
<td>thousand</td>
<td>thousand</td>
</tr>
<tr>
<td></td>
<td>million</td>
<td>million</td>
<td>million</td>
</tr>
<tr>
<td></td>
<td>billion</td>
<td>billion</td>
<td>billion</td>
</tr>
<tr>
<td>super</td>
<td>instant</td>
<td>1 second</td>
<td>1 week</td>
</tr>
</tbody>
</table>

\[
\text{Running time estimates:} \\
\text{Best case. Number of compares is } \sim N \log N.
\]

<table>
<thead>
<tr>
<th>(a[i])</th>
<th>initial values</th>
<th>random (\text{shuffle})</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 7 14</td>
<td>H A C B F E G D L I K J N M O</td>
<td>H A C B F E G D L I K J N M O</td>
</tr>
<tr>
<td>0 7 14</td>
<td>D A C B F E G H L I K J N M O</td>
<td>D A C B F E G H L I K J N M O</td>
</tr>
<tr>
<td>0 7 14</td>
<td>2 A B C D F E G H L I K J N M O</td>
<td>2 A B C D F E G H L I K J N M O</td>
</tr>
<tr>
<td>0 7 14</td>
<td>4 A B C D E F G H L I K J N M O</td>
<td>4 A B C D E F G H L I K J N M O</td>
</tr>
<tr>
<td>0 7 14</td>
<td>6 A B C D E F G H L I K J N M O</td>
<td>6 A B C D E F G H L I K J N M O</td>
</tr>
<tr>
<td>0 7 14</td>
<td>8 A B C D E F G H I J K L M N O</td>
<td>8 A B C D E F G H I J K L M N O</td>
</tr>
<tr>
<td>0 7 14</td>
<td>10 A B C D E F G H I J K L M N O</td>
<td>10 A B C D E F G H I J K L M N O</td>
</tr>
<tr>
<td>0 7 14</td>
<td>12 A B C D E F G H I J K L M N O</td>
<td>12 A B C D E F G H I J K L M N O</td>
</tr>
<tr>
<td>0 7 14</td>
<td>14 A B C D E F G H I J K L M N O</td>
<td>14 A B C D E F G H I J K L M N O</td>
</tr>
<tr>
<td>0 7 14</td>
<td>A B C D E F G H I J K L M N O</td>
<td>A B C D E F G H I J K L M N O</td>
</tr>
</tbody>
</table>

Lesson 1. Good algorithms are better than supercomputers.
Lesson 2. Great algorithms are better than good ones.
QuickSort: worst-case analysis

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

<table>
<thead>
<tr>
<th>le j</th>
<th>hi 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14</th>
</tr>
</thead>
<tbody>
<tr>
<td>initial values</td>
<td>A B C D E F G H I J K L M N O</td>
</tr>
<tr>
<td>random shuffle</td>
<td>A B C D E F G H I J K L M N O</td>
</tr>
<tr>
<td>0 1 14</td>
<td>A B C D E F G H I J K L M N O</td>
</tr>
<tr>
<td>1 2 14</td>
<td>A B C D E F G H I J K L M N O</td>
</tr>
<tr>
<td>2 3 14</td>
<td>A B C D E F G H I J K L M N O</td>
</tr>
<tr>
<td>3 4 14</td>
<td>A B C D E F G H I J K L M N O</td>
</tr>
<tr>
<td>4 5 14</td>
<td>A B C D E F G H I J K L M N O</td>
</tr>
<tr>
<td>5 6 14</td>
<td>A B C D E F G H I J K L M N O</td>
</tr>
<tr>
<td>6 7 14</td>
<td>A B C D E F G H I J K L M N O</td>
</tr>
<tr>
<td>7 8 14</td>
<td>A B C D E F G H I J K L M N O</td>
</tr>
<tr>
<td>8 9 14</td>
<td>A B C D E F G H I J K L M N O</td>
</tr>
<tr>
<td>9 10 14</td>
<td>A B C D E F G H I J K L M N O</td>
</tr>
<tr>
<td>10 11 14</td>
<td>A B C D E F G H I J K L M N O</td>
</tr>
<tr>
<td>11 12 14</td>
<td>A B C D E F G H I J K L M N O</td>
</tr>
<tr>
<td>12 13 14</td>
<td>A B C D E F G H I J K L M N O</td>
</tr>
<tr>
<td>13 14 14</td>
<td>A B C D E F G H I J K L M N O</td>
</tr>
<tr>
<td>14 14 14</td>
<td>A B C D E F G H I J K L M N O</td>
</tr>
</tbody>
</table>

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 \( \sim \frac{1}{2} N \ln N \)).

**Pf.** \( C_N \) satisfies the recurrence \( C_0 = C_1 = 0 \) and for \( N \geq 2 \):

\[
C_N = (N+1) \left( \frac{C_{N-1}}{N} + \frac{C_{N-2}}{N} + \ldots + \frac{C_0}{N} \right)
\]

- Multiply both sides by \( N \) and collect terms:
  \[
  NC_N = N(N+1) + 2(C_0 + C_1 + \ldots + C_{N-1})
  \]
- Subtract from this equation the same equation for \( N - 1 \):
  \[
  NC_N - (N-1)C_{N-1} = 2N + 2C_{N-1}
  \]
- Rearrange terms and divide by \( N(N+1) \):
  \[
  \frac{C_N}{N+1} = \frac{C_{N-1}}{N} + \frac{2}{N+1}
  \]
- Repeatedly apply previous equation:
  \[
  \frac{C_N}{N+1} = \frac{C_{N-1}}{N} + \frac{2}{N+1}
  \]
- Approximate sum by an integral:
  \[
  C_N \sim 2(N+1) \left( \frac{1}{3} + \frac{1}{4} + \frac{1}{5} + \ldots + \frac{1}{N+1} \right)
  \]
- Finally, the desired result:
  \[
  C_N \sim 2(N+1) \ln N \approx 1.39N \ln N
  \]

QuickSort: summary of performance characteristics

QuickSort is a (Las Vegas) randomized algorithm.
- Guaranteed to be correct.
- Running time depends on random shuffle.

**Average case.** Expected number of compares is \( \sim 1.39N \ln N \).
- 39% more compares than mergesort.
- Faster than mergesort in practice because of less data movement.

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

**Worst case.** Number of compares is \( \sim \frac{1}{2} N^2 \).
[ but more likely that lightning bolt strikes computer during execution ]
Quick sort properties

**Proposition.** Quick sort is an in-place sorting algorithm.

**Pf.**
- Partitioning: constant extra space.
- Depth of recursion: logarithmic extra space (with high probability).

can guarantee logarithmic depth by recurring on smaller subarray before larger subarray (requires using an explicit stack)

**Proposition.** Quick sort is not stable.

**Pf.** [by counterexample]

<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>

Quick sort: practical improvements

**Insertion sort small subarrays.**
- Even quick sort has too much overhead for tiny subarrays.
- Cutoff to insertion sort for \(n \approx 10\) items.

```
private static void sort(Comparable[] a, int lo, int hi)
{
    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.

\[\sim 12/7\quad N\ln N\quad \text{comparisons (14\% fewer)}\]
\[\sim 12/35\quad N\ln N\quad \text{exchanges (3\% more)}\]

```
private static void sort(Comparable[] a, int lo, int hi)
{
    int median = medianOf3(a, lo, lo + (hi - lo)/2, hi);
    swap(a, lo, median);

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

2.3 Quick sort
Selection

**Goal.** Given an array of \( N \) items, find the \( k \)-th smallest item.

**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?

Quick-select: mathematical analysis

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

**PF sketch.**
- Intuitively, each partitioning step splits array approximately in half: \( N + N/2 + N/4 + \ldots + 1 \approx 2N \) compares.
- Formal analysis similar to quicksort analysis yields:

\[
C_N = 2N + 2k \ln (N / k) + 2(N - k) \ln (N / (N - k))
\]
- Ex: \((2 + 2 \ln 2)N \approx 3.38N \) compares to find median \( k = N/2 \).

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 \( k \).

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];
}

Theoretical context for selection


**Remark.** Constants are 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.
### 2.3 Quicksort

- quicksort
- selection
- duplicate keys
- system sorts

### Duplicate keys

Often, purpose of sort is to bring items with equal keys together.
- Sort population by age.
- 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: stop on equal keys

Our partitioning subroutine stops both scans on equal keys.

**Q. Why not continue scans on equal keys?**

### Quicksort quiz 2

What is the result of partitioning the following array (skip over equal keys)?

**A.**

```
A A A A A A A A A A A A

A A A A A A A A A A A A A A

A A A A A A A A A A A A A A
```

**B.**

```
A A A A A A A A A A A A A A

A A A A A A A A A A A A A A

A A A A A A A A A A A A A A
```

**C.**

```
A A A A A A A A A A A A A A

A A A A A A A A A A A A A A
```

**D.** *I don't know.*
What is the result of partitioning the following array (stop on equal keys)?

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

A.  
B.  
C.  
D.  I don't know.

Duplicate keys: partitioning strategies

Bad. Don’t stop scans on equal keys.
[ \( \sim \frac{1}{2} N^2 \) compares when all keys equal ]

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

Good. Stop scans on equal keys.
[ \( \sim N \lg N \) compares when all keys equal ]

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

Better. Put all equal keys in place. How?
[ \( \sim N \) compares when all keys equal ]

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

Dutch National Flag Problem

Problem. [Edsger Dijkstra] Given an array of \( N \) buckets, each containing a red, white, or blue pebble, sort them by color.

Operations allowed.
- \( \text{swap}(i, j) \): swap the pebble in bucket \( i \) with the pebble in bucket \( j \).
- \( \text{color}(i) \): color of pebble in bucket \( i \).

Requirements.
- Exactly \( N \) calls to \( \text{color]() \).
- At most \( N \) calls to \( \text{swap()} \).
- Constant extra space.
3-way partitioning

**Goal.** Partition array into three parts so that:
- Entries between lt and gt equal to the partition item.
- No larger entries to left of lt.
- No smaller entries to right of gt.

![Diagram showing partitioning](image)

**Dutch national flag problem.** [Edsger Dijkstra]
- Conventional wisdom until mid 1990s: not worth doing.
- Now incorporated into C library `qsort()` and Java 6 system sort.

Dijkstra's 3-way partitioning demo

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

![Diagram showing partitioning](image)

Dijkstra's 3-way partitioning: trace

<table>
<thead>
<tr>
<th>lt</th>
<th>i</th>
<th>gt</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>11</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>11</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>11</td>
</tr>
<tr>
<td>1</td>
<td>3</td>
<td>10</td>
</tr>
<tr>
<td>1</td>
<td>3</td>
<td>9</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
<td>9</td>
</tr>
<tr>
<td>2</td>
<td>5</td>
<td>9</td>
</tr>
<tr>
<td>2</td>
<td>5</td>
<td>8</td>
</tr>
<tr>
<td>2</td>
<td>6</td>
<td>7</td>
</tr>
<tr>
<td>3</td>
<td>7</td>
<td>7</td>
</tr>
<tr>
<td>3</td>
<td>8</td>
<td>7</td>
</tr>
</tbody>
</table>

Array contents after each loop iteration:

```
0  1  2  3  4  5  6  7  8  9  10  11
B  B  W  W  W  B  R  R  R  B  W  R
B  B  W  W  W  B  R  R  R  B  W  R
B  R  R  R  R  R  B  B  R  B  R  R
B  R  R  R  R  R  B  B  R  B  R  R
R  R  R  R  R  R  B  B  R  B  B  B
R  R  R  R  R  R  B  B  R  B  B  B
R  B  B  B  B  B  R  R  R  R  R  R
R  B  B  B  B  B  R  R  R  R  R  R
W  W  W  W  W  W  B  B  B  B  B  B
W  W  W  W  W  W  B  B  B  B  B  B
W  W  W  W  W  W  B  B  B  B  B  B
W  W  W  W  W  W  B  B  B  B  B  B
W  W  W  W  W  W  B  B  B  B  B  B
```

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

3-way quicksort: visual trace

Duplicate keys: lower bound

**Sorting lower bound.** If there are $n$ distinct keys and the $i^{th}$ one occurs $x_i$ times, any compare-based sorting algorithm must use at least

$$\log \left( \frac{n!}{x_1! x_2! \cdots x_n!} \right) \sim - \sum_{i=1}^{n} x_i \log \frac{x_i}{n}$$

$N \log N$ when all distinct; linear when only a constant number of distinct keys compares in the worst case.

**Proposition.** [Sedgewick-Bentley 1997]

*Quicksort with 3-way partitioning is entropy-optimal.*

**Pf.** [beyond scope of course]

**Bottom line.** Quicksort with 3-way partitioning reduces running time from linearithmic to linear in broad class of applications.

Sorting summary

<table>
<thead>
<tr>
<th>inplace?</th>
<th>stable?</th>
<th>best</th>
<th>average</th>
<th>worst</th>
<th>remarks</th>
</tr>
</thead>
<tbody>
<tr>
<td>✔</td>
<td>✔</td>
<td>$\frac{1}{6} N^2$</td>
<td>$\frac{1}{6} N^2$</td>
<td>$\frac{1}{6} N^2$</td>
<td>$N$ exchanges</td>
</tr>
<tr>
<td>✔</td>
<td>✔</td>
<td>$N$</td>
<td>$\frac{1}{6} N^2$</td>
<td>$\frac{1}{6} N^2$</td>
<td>use for small $N$ or partially ordered</td>
</tr>
<tr>
<td>✔</td>
<td>✔</td>
<td>$N \log N$</td>
<td>$\frac{1}{6} N^2$</td>
<td>$\frac{1}{6} N^2$</td>
<td>$N \log N$ guaranteed; stable</td>
</tr>
<tr>
<td>✔</td>
<td>✔</td>
<td>$\frac{1}{6} N \log N$</td>
<td>$\frac{1}{6} N \log N$</td>
<td>$\frac{1}{6} N \log N$</td>
<td>improves merge order</td>
</tr>
<tr>
<td>✔</td>
<td>✔</td>
<td>$N \log N$</td>
<td>$2 N \ln N$</td>
<td>$\frac{1}{6} N^2$</td>
<td>$N \log N$ probabilistic guarantee; fastest in practice</td>
</tr>
<tr>
<td>✔</td>
<td>✔</td>
<td>$N \log N$</td>
<td>$2 N \ln N$</td>
<td>$\frac{1}{6} N^2$</td>
<td>improves quicksort when duplicate keys</td>
</tr>
<tr>
<td>✔</td>
<td>✔</td>
<td>$N \log N$</td>
<td>$N \log N$</td>
<td>$N \log N$</td>
<td>holy sorting grail</td>
</tr>
</tbody>
</table>
2.3 QUICKSORT

Sorting applications

Sorting algorithms are essential in a broad variety of applications:

- Sort a list of names.
- Organize an MP3 library.
- Display Google PageRank results.
- List RSS feed in reverse chronological order.

- Find the median.
- Identify statistical outliers.
- Binary search in a database.
- Find duplicates in a mailing list.

- Data compression.
- Computer graphics.
- Computational biology.
- Load balancing on a parallel computer.

... 

War story (system sort in C)

A beautiful bug report. [Allan Wilks and Rick Becker, 1991]

We found that qsort is unbearably slow on "organ-pipe" inputs like "01233210":

```c
class main (int argc, char** argv) {
    int n = atoi(argv[1]), i, x[100000];
    for (i = 0; i < n; i++)
        x[i] = i;
    for ( ; i < 2*n; i++)
        x[i] = 2*n-i-1;
    qsort(x, 2*n, sizeof(int), intcmp);
}
```

Here are the timings on our machine:

- `time a.out 2000`
  - real 5.65s
- `time a.out 4000`
  - real 21.64s
- `time a.out 8000`
  - real 85.11s

Why is qsort so slow?

At the time, almost all qsort implementations based on those in:

- Version 7 Unix (1979): quadratic time to sort organ-pipe arrays.
- BSD Unix (1983): quadratic time to sort random arrays of 0s and 1s.
Engineering a system sort (in 1993)

Bentley-McIlroy quicksort.
- Cutoff to insertion sort for small subarrays.
- Partitioning item: median of 3 or Tukey's ninther.
- Partitioning scheme: Bentley-McIlroy 3-way partitioning.

Very widely used. C, C++, Java 6, ....

A beautiful mailing list post (Yaroslavskiy-Bloch-Bentley, October 2009)

Replacement of quicksort in java.util.Arrays with new dual-pivot quicksort

Hello All,

I'd like to share with you new Dual-Pivot Quicksort which is faster than the known implementations (theoretically and experimentally). I'd like to propose to replace the JDK's Quicksort implementation by new one.

... The new Dual-Pivot Quicksort uses two pivots elements in this manner:

1. Pick an elements P1, P2, called pivots from the array.
2. Assume that P1 <= P2, otherwise swap it.
3. Reorder the array into three parts: those less than the smaller pivot, those larger than the larger pivot, and in between are those elements between (or equal to) the two pivots.
4. Recursively sort the sub-arrays.

The invariant of the Dual-Pivot Quicksort is:

\[ [ < P1 | P1 <= & <= P2 > P2 ] \]

Dual-pivot quicksort

Use two partitioning items \( p_1 \) and \( p_2 \) and partition into three subarrays:
- Keys less than \( p_1 \).
- Keys between \( p_1 \) and \( p_2 \).
- Keys greater than \( p_2 \).

<table>
<thead>
<tr>
<th>(&lt; p_1)</th>
<th>(p_1)</th>
<th>(\geq p_1 ) and (&lt;= p_2)</th>
<th>(p_2)</th>
<th>(&gt; p_2)</th>
</tr>
</thead>
<tbody>
<tr>
<td>(\uparrow)</td>
<td>(\uparrow)</td>
<td>(\uparrow)</td>
<td>(\uparrow)</td>
<td></td>
</tr>
<tr>
<td>(\leq 10)</td>
<td>(\leq 10)</td>
<td>(&gt; 10)</td>
<td>(&gt; 10)</td>
<td></td>
</tr>
</tbody>
</table>

Recursively sort three subarrays.

Note. Skip middle subarray if \( p_1 = p_2 \).
**Dual-pivot partitioning demo**

**Initialization.**
- Choose $a[lo]$ and $a[hi]$ as partitioning items.
- Exchange if necessary to ensure $a[lo] \leq a[hi]$.

 Initialization. Tomatekqe

**Main loop.** Repeat until $i$ and $gt$ pointers cross.
- If $(a[i] < a[lo])$, exchange $a[i]$ with $a[lt]$ and increment $lt$ and $i$.
- Else if $(a[i] > a[hi])$, exchange $a[i]$ with $a[gt]$ and decrement $gt$.
- Else, increment $i$.

<table>
<thead>
<tr>
<th>$&lt; p_1$</th>
<th>$p_1$</th>
<th>$\geq p_1 \text{ and } \leq p_2$</th>
<th>?</th>
<th>$p_2$</th>
<th>$&gt; p_2$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\uparrow$</td>
<td>$\uparrow$</td>
<td>$\uparrow$</td>
<td>$\uparrow$</td>
<td>$\uparrow$</td>
<td>$\uparrow$</td>
</tr>
<tr>
<td>lo</td>
<td>lt</td>
<td>i</td>
<td>gt</td>
<td>hi</td>
<td></td>
</tr>
</tbody>
</table>

Diplomat partitioning demo

**Finalize.**
- Exchange $a[hi]$ with $a[++gt]$.

<table>
<thead>
<tr>
<th>$&lt; p_1$</th>
<th>$p_1$</th>
<th>$\geq p_1 \text{ and } \leq p_2$</th>
<th>$p_2$</th>
<th>$&gt; p_2$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\uparrow$</td>
<td>$\uparrow$</td>
<td>$\uparrow$</td>
<td>$\uparrow$</td>
<td>$\uparrow$</td>
</tr>
<tr>
<td>lo</td>
<td>lt</td>
<td>gt</td>
<td>hi</td>
<td></td>
</tr>
</tbody>
</table>

**Dual-pivot quicksort**

Use two partitioning items $p_1$ and $p_2$ and partition into three subarrays:
- Keys less than $p_1$.
- Keys between $p_1$ and $p_2$.
- Keys greater than $p_2$.

<table>
<thead>
<tr>
<th>$&lt; p_1$</th>
<th>$p_1$</th>
<th>$\geq p_1 \text{ and } \leq p_2$</th>
<th>$p_2$</th>
<th>$&gt; p_2$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\uparrow$</td>
<td>$\uparrow$</td>
<td>$\uparrow$</td>
<td>$\uparrow$</td>
<td>$\uparrow$</td>
</tr>
<tr>
<td>lo</td>
<td>lt</td>
<td>gt</td>
<td>hi</td>
<td></td>
</tr>
</tbody>
</table>

Now widely used. Java 7, Python unstable sort, Android, ...
Three-pivot quicksort

Use three partitioning items $p_1$, $p_2$, and $p_3$ and partition into four subarrays:
- Keys less than $p_1$.
- Keys between $p_1$ and $p_2$.
- Keys between $p_2$ and $p_3$.
- Keys greater than $p_3$.

\[
\begin{array}{cccccc}
< p_1 & \quad & p_1 & \quad & \geq p_2 & \quad & \leq p_3 & \quad & > p_3 \\
\uparrow 1o & \quad & \uparrow a_1 & \quad & \uparrow a_2 & \quad & \uparrow a_3 & \quad & \uparrow h_1
\end{array}
\]

Quicksort quiz 4

Why do 2-pivot (and 3-pivot) quicksort perform better than 1-pivot?
A. Fewer compares.
B. Fewer exchanges.
C. Fewer cache misses.
D. I don’t know.

Quicksort quiz 4

Which sorting algorithm to use?

Many sorting algorithms to choose from:

<table>
<thead>
<tr>
<th>sorts</th>
<th>algorithms</th>
</tr>
</thead>
<tbody>
<tr>
<td>elementary sorts</td>
<td>insertion sort, selection sort, bubblesort, shaker sort, ...</td>
</tr>
<tr>
<td>subquadratic sorts</td>
<td>quicksort, mergesort, heapsort, shellsort, samplesort, ...</td>
</tr>
<tr>
<td>system sorts</td>
<td>dual-pivot quicksort, timsort, introsort, ...</td>
</tr>
<tr>
<td>external sorts</td>
<td>Poly-phase mergesort, cascade-merge, psort, ....</td>
</tr>
<tr>
<td>radix sorts</td>
<td>MSD, LSD, 3-way radix quicksort, ...</td>
</tr>
<tr>
<td>parallel sorts</td>
<td>bitonic sort, odd-even sort, smooth sort, GPUsort, ...</td>
</tr>
</tbody>
</table>

Reference: Analysis of Pivot Sampling in Dual-Pivot Quicksort by Wild–Nebel–Martínez

Bottom line. Caching can have a significant impact on performance.

Beyond scope of this course.
Which sorting algorithm to use?

Applications have diverse attributes.
- Stable?
- Parallel?
- In-place?
- Deterministic?
- Duplicate keys?
- Multiple key types?
- Linked list or arrays?
- Large or small items?
- Randomly-ordered array?
- Guaranteed performance?

Q. Is the system sort good enough?
A. Usually.

System sort in Java 7

Arrays.sort().
- Has method for objects that are Comparable.
- Has overloaded method for each primitive type.
- Has overloaded method for use with a Comparator.
- Has overloaded methods for sorting subarrays.

Algorithms.
- Dual-pivot quicksort for primitive types.
- Timsort for reference types.

Q. Why use different algorithms for primitive and reference types?