2.3 Quicksort

- quicksort
- selection
- duplicate keys
- system sorts
Two classic sorting algorithms

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.

- Java sort for objects.
- Perl, C++ stable sort, Python stable sort, Firefox JavaScript, ...

Quicksort.

- Java sort for primitive types.
- C qsort, Unix, Visual C++, Python, Matlab, Chrome JavaScript, ...
Quicksort t-shirt

```java
public static void quicksort(char[] items, int left, int right)
{
    int i, j;
    char x, y;

    i = left; j = right;
    x = items[(left + right) / 2];

    do
    {
        while ((items[j] < x) && (i < right)) i++;
        while ((x < items[i]) && (j > left)) j--;

        if (i <= j)
        { 
            y = items[i];
            items[i] = items[j];
            items[j] = y;
            i++; j--;
        }
    }
    while (i <= j);

    if (left < j) quicksort(items, left, j);
    if (j < right) quicksort(items, i, right);
}
```
2.3 Quicksort

- quicksort
- selection
- duplicate keys
- system sorts
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
Quicksort 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].
Quicksort 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].

partitioned!
Quicksort: Java code for partitioning

```java
private static int partition(Comparable[] a, int lo, int hi) {
    int i = lo, j = hi+1;
    while (true) {
        while (less(a[++i], a[lo]))
            if (i == hi) break;
        while (less(a[lo], a[--j]))
            if (j == lo) break;

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

Quicksort partitioning overview

- **before**: $\downarrow$
  - $\downarrow$ lo
  - $\downarrow$ hi
- **during**: $\downarrow \leq v \downarrow \geq v$
  - $\downarrow \downarrow \downarrow$
  - $\downarrow i \downarrow j$
- **after**: $\downarrow \leq v \downarrow v \downarrow \geq v$
  - $\downarrow \downarrow \downarrow$
  - $\downarrow lo \downarrow j \downarrow hi$
public class Quick {
    private static int partition(Comparable[] a, int lo, int hi) {
        /* see previous slide */
    }

    public static void sort(Comparable[] a) {
        StdRandom.shuffle(a);
        sort(a, 0, a.length - 1);
    }

    private static void sort(Comparable[] a, int lo, int hi) {
        if (hi <= lo) return;
        int j = partition(a, lo, hi);
        sort(a, lo, j-1);
        sort(a, j+1, hi);
    }
}
Quicksort trace

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

Quicksort trace (array contents after each partition)

```
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
Q U I C K S O R T E X A M P L E
K R A T E L E P U I M Q C X O S
```

```
0 5 15
E C A I E K L P U T M Q R X O S
```

```
0 3 4
E C A E I K L P U T M Q R X O S
```

```
0 2 2
A C E E I K L P U T M Q R X O S
```

```
0 0 1
A C E E I K L P U T M Q R X O S
```

```
1 1
A C E E I K L P U T M Q R X O S
```

```
4 4
A C E E I K L P U T M Q R X O S
```

```
6 6 15
A C E E I K L P U T M Q R X O S
```

```
7 9 15
A C E E I K L M O P T Q R X U S
```

```
7 7 8
A C E E I K L M O P T Q R X U S
```

```
8 8
A C E E I K L M O P T Q R X U S
```

```
10 13 15
A C E E I K L M O P S Q R T U X
```

```
10 12 12
A C E E I K L M O P R Q S T U X
```

```
10 11 11
A C E E I K L M O P Q R S T U X
```

```
10 10
A C E E I K L M O P Q R S T U X
```

```
14 14 15
A C E E I K L M O P Q R S T U X
```

```
15 15
A C E E I K L M O P Q R S T U X
```

```
no partition
for subarrays
of size 1
```

result

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

random shuffle
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>Computer</th>
<th>Insertion Sort ($N^2$)</th>
<th>Mergesort ($N \log N$)</th>
<th>Quicksort ($N \log N$)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Home</td>
<td>instant</td>
<td>instant</td>
<td>instant</td>
</tr>
<tr>
<td></td>
<td>2.8 hours</td>
<td>1 second</td>
<td>0.6 sec</td>
</tr>
<tr>
<td>Super</td>
<td>instant</td>
<td>1 week</td>
<td>instant</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 $\sim N \lg N$. 

<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>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>H</td>
<td>A</td>
<td>C</td>
<td>B</td>
<td>F</td>
<td>E</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>O</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>H</td>
<td>A</td>
<td>C</td>
<td>B</td>
<td>F</td>
<td>E</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>O</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>
<td>M</td>
<td>O</td>
</tr>
<tr>
<td>0</td>
<td>3</td>
<td>6</td>
<td>B</td>
<td>A</td>
<td>C</td>
<td>D</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>
<td>M</td>
<td>O</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>A</td>
<td>B</td>
<td>C</td>
<td>D</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>
<td>M</td>
<td>O</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>A</td>
<td>B</td>
<td>C</td>
<td>D</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>
<td>M</td>
<td>O</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>2</td>
<td>A</td>
<td>B</td>
<td>C</td>
<td>D</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>
<td>M</td>
<td>O</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>A</td>
<td>B</td>
<td>C</td>
<td>D</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>
<td>M</td>
<td>O</td>
</tr>
<tr>
<td>0</td>
<td>2</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>L</td>
<td>I</td>
<td>K</td>
<td>J</td>
<td>N</td>
<td>M</td>
<td>O</td>
</tr>
<tr>
<td>0</td>
<td>3</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>L</td>
<td>I</td>
<td>K</td>
<td>J</td>
<td>N</td>
<td>M</td>
<td>O</td>
</tr>
<tr>
<td>0</td>
<td>4</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>L</td>
<td>I</td>
<td>K</td>
<td>J</td>
<td>N</td>
<td>M</td>
<td>O</td>
</tr>
<tr>
<td>0</td>
<td>1</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>J</td>
<td>I</td>
<td>K</td>
<td>L</td>
<td>N</td>
<td>M</td>
<td>O</td>
</tr>
<tr>
<td>0</td>
<td>0 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>N</td>
<td>M</td>
<td>O</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1</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>O</td>
<td>N</td>
</tr>
<tr>
<td>0</td>
<td>1</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>O</td>
<td>N</td>
</tr>
<tr>
<td>0</td>
<td>1</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>
<td>O</td>
</tr>
</tbody>
</table>

A B C D E F G H I J K L M N O
Quicksort: worst-case analysis

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

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

$$C_N = (N + 1) + \left( \frac{C_0 + C_{N-1}}{N} \right) + \left( \frac{C_1 + C_{N-2}}{N} \right) + \ldots + \left( \frac{C_{N-1} + C_0}{N} \right)$$

- Multiply both sides by $N$ and collect terms:

$$NC_N = N(N + 1) + 2(C_0 + C_1 + \ldots + C_{N-1})$$

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

$$NC_N - (N - 1)C_{N-1} = 2N + 2C_{N-1}$$

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

$$\frac{C_N}{N + 1} = \frac{C_{N-1}}{N} + \frac{2}{N + 1}$$
QuickSort: average-case analysis

- Repeatedly apply above equation:

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

\[
= \frac{C_{N-2}}{N-1} + \frac{2}{N} + \frac{2}{N+1}
\]

\[
= \frac{C_{N-3}}{N-2} + \frac{2}{N-1} + \frac{2}{N} + \frac{2}{N+1}
\]

\[
= \frac{2}{3} + \frac{2}{4} + \frac{2}{5} + \cdots + \frac{2}{N+1}
\]

- Approximate sum by an integral:

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

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

- Finally, the desired result:

\[
C_N \sim 2(N+1) \ln N \approx 1.39 N \log_2 N
\]
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!)
Proposition. Quicksort 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**

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.

```
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 In N compares (slightly fewer)
~ 12/35 N In N exchanges (slightly more)
Quicksort with median-of-3 partitioning and cutoff for small subarrays visualization
2.3 Quicksort

- quicksort
- selection
- duplicate keys
- system sorts
2.3 QuickSort

- quicksort
- selection
- duplicate keys
- system sorts
Selection

**Goal.** Given an array of $N$ items, find a $k^{th}$ 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? 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: 
  \[ N + N/2 + N/4 + \ldots + 1 \sim 2N \text{ compares}. \]
- Formal analysis similar to quicksort analysis yields:

  \[ C_N = 2N + 2k \ln (N/k) + 2(N-k) \ln (N/(N-k)) \]

  \( (2 + 2 \ln 2) N \) 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.
Theoretical context for selection


**Remark.** But, constants are too high ⇒ 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
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

**Mergesort with duplicate keys.** 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.
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 C C C
A A A A A A A A A A A
```

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

**Consequence.** $\sim N \lg N$ 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
```

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

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

### 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.
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 $i$
Dijkstra 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 $i$

![Dijkstra 3-way partitioning demo](image)
Dijkstra's 3-way partitioning: trace

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

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

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

compares in the worst case.

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

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

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

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

- quicksort
- selection
- duplicate keys
- system sorts
2.3 Quicksort

- quicksort
- selection
- duplicate keys
- system sorts
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.

...
Java system sorts

Arrays.sort().

- Has different method for each primitive type.
- Has a method for data types that implement Comparable.
- Has a method that uses a Comparator.
- Uses tuned quicksort for primitive types; tuned mergesort for objects.

Q. Why use different algorithms for primitive and reference types?
War story (C qsort function)

AT&T Bell Labs (1991). Allan Wilks and Rick Becker discovered that a qsort() call that should have taken seconds was taking minutes.

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.
Basic algorithm = quicksort.

- Cutoff to insertion sort for small subarrays.
- Partitioning scheme: Bentley-McIlroy 3-way partitioning.
- Partitioning item.
  - small arrays: middle entry
  - medium arrays: median of 3
  - large arrays: Tukey's ninther

Now widely used. C, C++, Java 6, ....
Tukey's ninther

Tukey's ninther. Median of the median of 3 samples, each of 3 entries.

- Approximates the median of 9.
- Uses at most 12 compares.

Q. Why use Tukey's ninther?
A. Better partitioning than random shuffle and less costly.
Achilles heel in Bentley-McIlroy implementation (Java system sort)

Q. Based on all this research, Java’s system sort is solid, right?

A. No: a killer input.
   • Overflows function call stack in Java and crashes program.
   • Would take quadratic time if it didn’t crash first.

% more 250000.txt
0
218750
222662
11
166672
247070
83339
...

250,000 integers between 0 and 250,000

% java IntegerSort 250000 < 250000.txt
Exception in thread "main"
java.lang.StackOverflowError
   at java.util.Arrays.sort1(Arrays.java:562)
   at java.util.Arrays.sort1(Arrays.java:606)
   at java.util.Arrays.sort1(Arrays.java:608)
   at java.util.Arrays.sort1(Arrays.java:608)
   at java.util.Arrays.sort1(Arrays.java:608)
   ...

Java's sorting library crashes, even if you give it as much stack space as Windows allows
System sort: Which algorithm to use?

Many sorting algorithms to choose from:

**Internal sorts.**
- Insertion sort, selection sort, bubblesort, shaker sort.
- Quicksort, mergesort, heapsort, samplesort, shellsort.
- Solitaire sort, red-black sort, splaysort, Yaroslavskiy sort, psort, ...

**External sorts.** Poly-phase mergesort, cascade-merge, oscillating sort.

**String/radix sorts.** Distribution, MSD, LSD, 3-way string quicksort.

**Parallel sorts.**
- Bitonic sort, Batcher even-odd sort.
- Smooth sort, cube sort, column sort.
- GPUsort.
System sort: Which algorithm to use?

Applications have diverse attributes.
- Stable?
- Parallel?
- Deterministic?
- Keys all distinct?
- Multiple key types?
- Linked list or arrays?
- Large or small items?
- Is your array randomly ordered?
- Need guaranteed performance?

Elementary sort may be method of choice for some combination.
Cannot cover all combinations of attributes.

Q. Is the system sort good enough?
A. Usually.
## Sorting summary

<table>
<thead>
<tr>
<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>$N^2/2$</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>$2N \ln N$</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>$2N \ln N$</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>$N$</td>
<td>holy sorting grail</td>
</tr>
</tbody>
</table>
2.3 QUICKSORT

- quicksort
- selection
- duplicate keys
- system sorts
2.3 **QUICKSORT**

- quicksort
- selection
- duplicate keys
- system sorts