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, Python stable sort.

Quicksort.
• Java sort for primitive types.
• C qsort, Unix, g++, Visual C++, Python.
- quicksort
- selection
- duplicate keys
- system sorts
Quicksort

Basic plan.

- **Shuffle** the array.
- **Partition** so that, for some $j$
  - element $a[j]$ is in place
  - no larger element to the left of $j$
  - no smaller element to the right of $j$
- **Sort** each piece recursively.

Sir Charles Antony Richard Hoare
1980 Turing Award

Quicksort overview
Quicksort partitioning

Basic plan.

- Scan $i$ from left for an item that belongs on the right.
- Scan $j$ from right for item item that belongs on the left.
- Exchange $a[i]$ and $a[j]$.
- Continue until pointers cross.

Partitioning trace (array contents before and after each exchange)
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;  // 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
}
```

Quicksort partitioning overview:

- **Before**: \( V \)
  - \( i \) before partition
  - \( j \) before partition
  - \( lo \)
  - \( hi \)

- **During**: \( V \leq V \) \( i \) \( j \) \( V \geq V \)
  - \( i \) during partitioning
  - \( j \) during partitioning

- **After**: \( V \leq V \) \( V \) \( V \geq V \)
  - \( i \) after partitioning
  - \( j \) after partitioning
  - \( lo \)
  - \( hi \)
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);
        sort(a, j+1, hi);
    }
}

shuffle needed for performance guarantee
Quicksort trace

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

<table>
<thead>
<tr>
<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>Q</td>
<td>U</td>
<td>I</td>
<td>C</td>
<td>K</td>
<td>S</td>
<td>O</td>
<td>R</td>
<td>T</td>
<td>E</td>
<td>X</td>
<td>A</td>
<td>M</td>
<td>P</td>
<td>L</td>
<td>E</td>
</tr>
<tr>
<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>
<td>Q</td>
<td>C</td>
<td>X</td>
<td>O</td>
<td>S</td>
</tr>
<tr>
<td>E</td>
<td>C</td>
<td>A</td>
<td>I</td>
<td>E</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>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>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>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>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>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>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>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>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>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>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>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>

initial values
random shuffle

no partition for subarrays of size 1

result

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

Quicksort trace (array contents after each partition)
QuickSort animation

50 random elements

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

Partitioning in-place. Using a spare 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) best to stop on elements equal to the partitioning element.
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></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>

<table>
<thead>
<tr>
<th></th>
<th>thousand</th>
<th>million</th>
<th>billion</th>
<th>thousand</th>
<th>million</th>
<th>billion</th>
<th>thousand</th>
<th>million</th>
<th>billion</th>
</tr>
</thead>
<tbody>
<tr>
<td>home</td>
<td>instant</td>
<td>2.8</td>
<td>317</td>
<td>instant</td>
<td>1</td>
<td>18</td>
<td>instant</td>
<td>0.3</td>
<td>6</td>
</tr>
<tr>
<td>super</td>
<td>instant</td>
<td>1</td>
<td>1</td>
<td>instant</td>
<td>instant</td>
<td>instant</td>
<td>instant</td>
<td>instant</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 \( \sim N \log N \).
Quicksort: worst case analysis

Worst case. Number of compares is $\sim N^2 / 2$. 
Quicksort: average-case analysis

**Proposition I.** The average number of compares $C_N$ to quicksort an array of $N$ elements 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 $N \geq 2$:

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

- 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:

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

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

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

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

• Approximate sum by an integral:

\[
C_N \sim 2(N + 1) \left(1 + \frac{1}{2} + \frac{1}{3} + \ldots + \frac{1}{N}\right)
\]

\[
\sim 2(N + 1) \int_1^N \frac{1}{x} dx
\]

• Finally, the desired result:

\[
C_N \sim 2(N + 1) \ln N \approx 1.39 N \ln N
\]
Quicksort: summary of performance characteristics

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

**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 input:
- Is sorted or reverse sorted.
- Has many duplicates (even if randomized!) [stay tuned]
Quicksort: practical improvements

Median of sample.
- Best choice of pivot element = median.
- Estimate true median by taking median of sample.

Insertion sort small subarrays.
- Even quicksort has too much overhead for tiny subarrays.
- Can delay insertion sort until end.

Optimize parameters.
- Median-of-3 random elements.
- Cutoff to insertion sort for \( \approx 10 \) elements.

Non-recursive version.
- Use explicit stack.
- Always sort smaller half first.

\[ \sim \frac{12}{7} N \ln N \text{ compares} \]
\[ \sim \frac{12}{35} N \ln N \text{ exchanges} \]
Quicksort with cutoff to insertion sort: visualization

- Input
- Result of first partition
- Left subarray partially sorted
- Both subarrays partially sorted
- Result

Quicksort with median-of-3 partitioning and cutoff for small subarrays
› quicksort
› selection
› duplicate keys
› system sorts
Selection

**Goal.** Find the $k^{th}$ largest element.

**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 $O(N \log N)$ upper bound.
- Easy $O(N)$ upper bound for $k = 1, 2, 3$.
- Easy $\Omega(N)$ lower bound.

**Which is true?**
- $\Omega(N \log N)$ lower bound?
- $O(N)$ upper bound?

---

**is selection as hard as sorting?**

**is there a linear-time algorithm for all $k$?**
Quick-select

Partition array so that:

• Element $a[j]$ is in place.
• No larger element to the left of $j$.
• No smaller element 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 roughly splits array in half:
  \[ N + N/2 + N/4 + \ldots + 1 \sim 2N \text{ compares.} \]
- Formal analysis similar to quicksort analysis yields:
  \[
  C_N = 2N + k \ln \left( \frac{N}{k} \right) + (N - k) \ln \left( \frac{N}{N - k} \right)
  \]

**Ex.** \((2 + 2 \ln 2) N\) compares to find the median.

**Remark.** Quick-select uses \(\sim N^2/2\) compares in worst case, but as with quicksort, the random shuffle provides a probabilistic guarantee.
Theoretical context for selection

**Challenge.** Design algorithm whose worst-case running time is linear.

**Proposition.** [Blum, Floyd, Pratt, Rivest, Tarjan, 1973] There exists a compare-based selection algorithm whose worst-case running time is linear.

**Remark.** But, algorithm is too complicated to be useful 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.
Generic methods

In our `select()` implementation, client needs a cast.

```java
Double[] a = new Double[N];
for (int i = 0; i < N; i++)
    a[i] = StdRandom.uniform();
Double median = (Double) Quick.select(a, N/2);
```

The compiler also complains.

```bash
% javac Quick.java
Note: Quick.java uses unchecked or unsafe operations.
Note: Recompile with -Xlint:unchecked for details.
```

Q. How to fix?
Generic methods

Pedantic (safe) version. Compiles cleanly, no cast needed in client.

```java
public class QuickPedantic {
    public static <Key extends Comparable<Key>> Key select(Key[] a, int k)
    { /* as before */ }

    public static <Key extends Comparable<Key>> void sort(Key[] a)
    { /* as before */ }

    private static <Key extends Comparable<Key>> int partition(Key[] a, int lo, int hi)
    { /* as before */ }

    private static <Key extends Comparable<Key>> boolean less(Key v, Key w)
    { /* as before */ }

    private static <Key extends Comparable<Key>> void exch(Key[] a, int i, int j)
    { Key swap = a[i]; a[i] = a[j]; a[j] = swap; }
}
```

Remark. Obnoxious code needed in system sort; not in this course (for brevity).

- quicksort
- selection
- duplicate keys
- system sorts
Duplicate keys

Often, purpose of sort is to bring records with duplicate 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 $\sim N \lg N$ 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 implementations also have this defect.

STOPONEQUALKEYS

 swap

if we don’t stop on equal keys

if we stop on equal keys
Duplicate keys: the problem

**Mistake.** Put all keys equal to the partitioning element on one side.
**Consequence.** \( \sim N^2 / 2 \) compares when all keys equal.

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

**Recommended.** Stop scans on keys equal to the partitioning element.
**Consequence.** \( \sim N \lg N \) compares when all keys equal.

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

**Desirable.** Put all keys equal to the partitioning element in place.

\[
\begin{array}{cccccccc}
\end{array}
\begin{array}{cccccccc}
\end{array}
\]
3-way partitioning

**Goal.** Partition array into 3 parts so that:

- Elements between \( \lt \) and \( \gt \) equal to partition element \( v \).
- No larger elements to left of \( \lt \).
- No smaller elements 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.
3-way partitioning: Dijkstra's solution

3-way partitioning:
- Let $v$ be partitioning element $a[lo]$.
- 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 $i$

**All the right properties.**
- In-place.
- Not much code.
- Small overhead if no equal keys.
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

before

<table>
<thead>
<tr>
<th>&lt;v</th>
<th>=v</th>
<th>&gt;v</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

during

<table>
<thead>
<tr>
<th>&lt;v</th>
<th>=v</th>
<th>&gt;v</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

after

<table>
<thead>
<tr>
<th>&lt;v</th>
<th>=v</th>
<th>&gt;v</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
3-way quicksort: visual trace

Visual trace of quicksort with 3-way partitioning
Duplicate keys: lower bound

**Sorting lower bound.** If there are $n$ distinct keys and the $i^{\text{th}}$ 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.
› selection
› duplicate keys
› comparators
› 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 news items in reverse chronological order.
- Find the median.
- Find the closest pair.
- Binary search in a database.
- Identify statistical outliers.
- Find duplicates in a mailing list.
- Data compression.
- Computer graphics.
- Computational biology.
- Supply chain management.
- Load balancing on a parallel computer.

Every system needs (and has) a system sort!
Java system sorts

Java uses both mergesort and quicksort.

- `Arrays.sort()` sorts array of `Comparable` or any primitive type.
- Uses quicksort for primitive types; mergesort for objects.

```java
import java.util.Arrays;

public class StringSort
{
    public static void main(String[] args)
    {
       String[] a = StdIn.readAll().split(" ");
       Arrays.sort(a);
       for (int i = 0; i < N; i++)
          StdOut.println(a[i]);
    }
}
```

Q. Why use different algorithms, depending on type?
Java system sort for primitive types

Engineering a sort function. [Bentley-McIlroy, 1993]

- Original motivation: improve `<qsort()>`.
- Basic algorithm = 3-way quicksort with cutoff to insertion sort.
- Partition on Tukey's ninther: median of the medians of 3 samples, each of 3 elements.

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

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

A killer input.
- Blows function call stack in Java and crashes program.
- Would take quadratic time if it didn’t crash first.

250,000 integers between 0 and 250,000

Java’s sorting library crashes, even if you give it as much stack space as Windows allows.
Achilles heel in Bentley-McIlroy implementation (Java system sort)

McIlroy's devious idea. [A Killer Adversary for Quicksort]

- Construct malicious input while running system quicksort, in response to elements compared.
- If $v$ is partitioning element, commit to $(v < a[i])$ and $(v < a[j])$, but don't commit to $(a[i] < a[j])$ or $(a[j] > a[i])$ until $a[i]$ and $a[j]$ are compared.

Consequences.

- Confirms theoretical possibility.
- Algorithmic complexity attack: you enter linear amount of data; server performs quadratic amount of work.

Remark. Attack is not effective if array is shuffled before sort.

Q. Why do you think system sort is deterministic?
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, Dobosiewicz sort, psort, ...

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

**Radix sorts.** Distribution, MSD, LSD, 3-way radix 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 records?
- 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>x</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>x</td>
<td>x</td>
<td>$N^2 / 2$</td>
<td>$N^2 / 4$</td>
<td>$N$ use for small $N$ or partially ordered</td>
</tr>
<tr>
<td>shell</td>
<td>x</td>
<td>?</td>
<td>?</td>
<td>$N$</td>
<td>tight code, subquadratic</td>
</tr>
<tr>
<td>quick</td>
<td>x</td>
<td></td>
<td>$N^2 / 2$</td>
<td>$2N \ln N$</td>
<td>$N \lg N$ probabilistic guarantee, fastest in practice</td>
</tr>
<tr>
<td>3-way quick</td>
<td>x</td>
<td></td>
<td>$N^2 / 2$</td>
<td>$2N \ln N$</td>
<td>$N$ improves quicksort in presence of duplicate keys</td>
</tr>
<tr>
<td>merge</td>
<td>x</td>
<td></td>
<td>$N \lg N$</td>
<td>$N \lg N$</td>
<td>$N \log N$ guarantee, stable</td>
</tr>
<tr>
<td>???</td>
<td>x</td>
<td>x</td>
<td>$N \lg N$</td>
<td>$N \lg N$</td>
<td>holy sorting grail</td>
</tr>
</tbody>
</table>
Which sorting algorithm?

<table>
<thead>
<tr>
<th>lifo</th>
<th>find</th>
<th>data</th>
<th>data</th>
<th>data</th>
<th>data</th>
<th>hash</th>
<th>data</th>
</tr>
</thead>
<tbody>
<tr>
<td>fifo</td>
<td>fifo</td>
<td>fifo</td>
<td>fifo</td>
<td>exch</td>
<td>fifo</td>
<td>fifo</td>
<td>exch</td>
</tr>
<tr>
<td>data</td>
<td>data</td>
<td>find</td>
<td>find</td>
<td>fifo</td>
<td>lifo</td>
<td>data</td>
<td>fifo</td>
</tr>
<tr>
<td>type</td>
<td>exch</td>
<td>hash</td>
<td>hash</td>
<td>find</td>
<td>type</td>
<td>link</td>
<td>find</td>
</tr>
<tr>
<td>hash</td>
<td>hash</td>
<td>heap</td>
<td>heap</td>
<td>hash</td>
<td>hash</td>
<td>leaf</td>
<td>hash</td>
</tr>
<tr>
<td>heap</td>
<td>heap</td>
<td>lifo</td>
<td>lifo</td>
<td>heap</td>
<td>heap</td>
<td>heap</td>
<td>heap</td>
</tr>
<tr>
<td>sort</td>
<td>less</td>
<td>link</td>
<td>link</td>
<td>leaf</td>
<td>link</td>
<td>exch</td>
<td>leaf</td>
</tr>
<tr>
<td>link</td>
<td>left</td>
<td>list</td>
<td>list</td>
<td>left</td>
<td>sort</td>
<td>node</td>
<td>left</td>
</tr>
<tr>
<td>list</td>
<td>leaf</td>
<td>push</td>
<td>push</td>
<td>less</td>
<td>find</td>
<td>lifo</td>
<td>less</td>
</tr>
<tr>
<td>push</td>
<td>lifo</td>
<td>root</td>
<td>root</td>
<td>lifo</td>
<td>list</td>
<td>left</td>
<td>lifo</td>
</tr>
<tr>
<td>find</td>
<td>push</td>
<td>sort</td>
<td>sort</td>
<td>link</td>
<td>push</td>
<td>find</td>
<td>link</td>
</tr>
<tr>
<td>root</td>
<td>root</td>
<td>type</td>
<td>type</td>
<td>list</td>
<td>root</td>
<td>path</td>
<td>list</td>
</tr>
<tr>
<td>leaf</td>
<td>list</td>
<td>leaf</td>
<td>leaf</td>
<td>sort</td>
<td>leaf</td>
<td>list</td>
<td>next</td>
</tr>
<tr>
<td>tree</td>
<td>tree</td>
<td>left</td>
<td>tree</td>
<td>tree</td>
<td>null</td>
<td>next</td>
<td>node</td>
</tr>
<tr>
<td>null</td>
<td>null</td>
<td>node</td>
<td>null</td>
<td>null</td>
<td>path</td>
<td>less</td>
<td>null</td>
</tr>
<tr>
<td>path</td>
<td>path</td>
<td>null</td>
<td>path</td>
<td>path</td>
<td>tree</td>
<td>root</td>
<td>path</td>
</tr>
<tr>
<td>node</td>
<td>node</td>
<td>path</td>
<td>node</td>
<td>node</td>
<td>exch</td>
<td>sink</td>
<td>push</td>
</tr>
<tr>
<td>left</td>
<td>link</td>
<td>tree</td>
<td>left</td>
<td>type</td>
<td>left</td>
<td>swim</td>
<td>root</td>
</tr>
<tr>
<td>less</td>
<td>sort</td>
<td>exch</td>
<td>less</td>
<td>root</td>
<td>less</td>
<td>null</td>
<td>sink</td>
</tr>
<tr>
<td>exch</td>
<td>type</td>
<td>less</td>
<td>exch</td>
<td>push</td>
<td>node</td>
<td>sort</td>
<td>sort</td>
</tr>
<tr>
<td>sink</td>
<td>sink</td>
<td>next</td>
<td>sink</td>
<td>sink</td>
<td>next</td>
<td>type</td>
<td>swap</td>
</tr>
<tr>
<td>swim</td>
<td>swim</td>
<td>sink</td>
<td>swim</td>
<td>swim</td>
<td>sink</td>
<td>tree</td>
<td>swim</td>
</tr>
<tr>
<td>next</td>
<td>next</td>
<td>swap</td>
<td>next</td>
<td>next</td>
<td>swap</td>
<td>push</td>
<td>tree</td>
</tr>
<tr>
<td>swap</td>
<td>swap</td>
<td>swim</td>
<td>swap</td>
<td>swap</td>
<td>swim</td>
<td>swap</td>
<td>type</td>
</tr>
</tbody>
</table>