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.

**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 example**

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

- **Shuffle**

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

- **Partition**

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

- **Sort left**

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

- **Sort right**

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

- **Result**

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

**Quicksort partitioning**

Repeat until $i$ and $j$ pointers cross.

- Scan $i$ from left to right so long as $a[i] < a[lo]$.
- Scan $j$ from right to left so long as $a[j] > a[lo]$.
- Exchange $a[i]$ with $a[j]$.

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

- **Stop** scan because $a[i] >= a[lo]$
Quicksort partitioning
Repeat until $i$ and $j$ pointers cross.
- Scan $i$ from left to right so long as $a[i] < a[lo]$.
- Scan $j$ from right to left so long as $a[j] > a[lo]$.
- Exchange $a[i]$ with $a[j]$.

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

$\uparrow$ $\uparrow$
$\downarrow$ $i$

$\uparrow$ $\downarrow$
$\downarrow$ $j$

stop 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].

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

\[ i \quad i \quad j \]

stop i scan because a[i] >= a[lo]

Quicksort partitioning

Repeat until i and j pointers cross.
• Scan i from left to right so long as a[i] < a[lo].
• Scan j from right to left so long as a[j] > a[lo].
• Exchange a[i] with a[j].

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

\[ j \quad j \quad i \]

Quicksort partitioning

Repeat until i and j pointers cross.
• Scan i from left to right so long as a[i] < a[lo].
• Scan j from right to left so long as a[j] > a[lo].
• Exchange a[i] with a[j].

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

\[ j \quad j \quad i \]

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 \( a[j] \).

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

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 $a[j]$.

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 $a[j]$.

stop $i$ scan because $a[i] >= a[lo]$
Quicksort partitioning

Repeat until \( i \) and \( j \) pointers cross.
- Scan \( i \) from left to right so long as \( a[i] < a[lo] \).
- Scan \( j \) from right to left so long as \( a[j] > a[lo] \).
- Exchange \( a[i] \) with \( a[j] \).

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

Partitioning trace (array contents before and after each exchange)

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

stop \( j \) scan because \( a[j] \leq a[lo] \)

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

pointers cross: exchange \( a[lo] \) with \( a[j] \)

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.
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: Java implementation

```java
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

Initial values

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

Random shuffle

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

Quicksort trace (array contents after each partition)

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

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 a bit trickier than it might seem.

**Staying in bounds.** The \((j = l)\) test is redundant (why?), but the \((i = h)\) test is not.

**Preserving randomness.** Shuffling is needed for performance guarantee.

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

---

Quicksort: empirical analysis

**Running time estimates:**
- Home PC executes \(10^8\) compares/second.
- Supercomputer executes \(10^{12}\) compares/second.

<table>
<thead>
<tr>
<th></th>
<th>insertion sort (N)</th>
<th>mergesort (N log N)</th>
<th>quicksort (N log N)</th>
</tr>
</thead>
<tbody>
<tr>
<td>computer</td>
<td>thousand</td>
<td>million</td>
<td>billion</td>
</tr>
<tr>
<td>home</td>
<td>instant</td>
<td>2.8 hours</td>
<td>317 years</td>
</tr>
<tr>
<td>super</td>
<td>instant</td>
<td>1 second</td>
<td>1 week</td>
</tr>
</tbody>
</table>

**Lesson 1.** Good algorithms are better than supercomputers.
**Lesson 2.** Great algorithms are better than good ones.

---

Quicksort: best-case analysis

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

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

- Multiply both sides by $N$ and collect terms:
- Subtract this from the same equation for $N-1$:
- Rearrange terms and divide by $N(N+1)$:

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

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

\[
NC_N - (N-1)C_{N-1} = 2N + 2C_{N-1}
\]

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

Approximate sum by an integral:

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

Finally, the desired result:

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

Quicksort: summary of performance characteristics

Worst case. Number of compares is quadratic.
- $N + (N-1) + (N-2) + \ldots + 1 = \frac{1}{2} N^2$.
- 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!)

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.

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.

```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);
}
```

~ 12/7 $N \ln N$ compares (slightly fewer)
~ 12/35 $N \ln N$ exchanges (slightly more)

**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 $\frac{1}{2} N \lg N$ 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.

Revised. Stop scans on items equal to the partitioning item.
Consequence. $\sim N \lg N$ compares when all keys equal.

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

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.

Dijkstra 3-way partitioning

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

Invariant

Dijkstra 3-way partitioning

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

Invariant
Dijkstra 3-way partitioning

1. Let v be partitioning item a[lo].
2. 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.
  - \( (a[i] \lt v) \): exchange \( a[lt] \) with \( a[i] \) and increment both \( lt \) and \( i \)
  - \( (a[i] \gt v) \): exchange \( a[gt] \) with \( a[i] \) and decrement \( gt \)
  - \( (a[i] = v) \): increment \( i \)

\[
\begin{array}{ccccccc}
& h & i & gt &\
& \downarrow & \downarrow & \downarrow &\
\end{array}
\]

Invariant

\[
\begin{array}{c|c|c|}
< v & = v & > v \\
\hline
\ll{lt} & \ll{i} & \ll{gt}
\end{array}
\]

Dijkstra 3-way partitioning

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

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

Invariant

\[
\begin{array}{c|c|c|}
< v & = v & > v \\
\hline
\ll{lt} & \ll{i} & \ll{gt}
\end{array}
\]
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 $i$

\[<v=v>^i\]

Dijkstra 3-way partitioning algorithm

3-way partitioning.

- Let $v$ be partitioning item $a[lo]$.  
- Scan $i$ from left to right.  
  - $a[i]$ less than $v$: exchange $a[lt]$ with $a[i]$ and increment both $lt$ and $i$  
  - $a[i]$ greater than $v$: exchange $a[gt]$ with $a[i]$ and decrement $gt$  
  - $a[i]$ equal to $v$: increment $i$

Most of the right properties.

- In-place.  
- Not much code.  
- Linear time if keys are all equal.

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: trace

Before: \[<v=v>\]

During: \[<v=v>\]

After: \[<v=v>\]
3-way quicksort: visual trace

equal to partitioning element

Sorting summary

<table>
<thead>
<tr>
<th></th>
<th>inplace?</th>
<th>stable?</th>
<th>worst</th>
<th>average</th>
<th>best</th>
<th>remarks</th>
</tr>
</thead>
<tbody>
<tr>
<td>selection</td>
<td>✔</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N exchanges</td>
</tr>
<tr>
<td>insertion</td>
<td>✔</td>
<td>✔</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>use for small or partially ordered</td>
</tr>
<tr>
<td>shell</td>
<td>✔</td>
<td>?</td>
<td>?</td>
<td>N</td>
<td></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></td>
<td>N log N guarantee, stable</td>
</tr>
<tr>
<td>quick</td>
<td>✔</td>
<td>N</td>
<td>2 N ln N</td>
<td>N lg N</td>
<td></td>
<td>N log N probabilistic guarantee, fastest in practice</td>
</tr>
<tr>
<td>3-way quick</td>
<td>✔</td>
<td>N</td>
<td>2 N ln N</td>
<td>N</td>
<td></td>
<td>improves quicksort in presence of duplicate keys</td>
</tr>
<tr>
<td>???</td>
<td>✔</td>
<td>✔</td>
<td>N lg N</td>
<td>N lg N</td>
<td>N lg N</td>
<td>holy sorting grail</td>
</tr>
</tbody>
</table>