**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**

**Shuffling**

- Shuffling is the process of rearranging an array of elements randomly.
- A good shuffling algorithm is unbiased, where every ordering is equally likely.

  - e.g. the Fisher–Yates shuffle (aka. the Knuth shuffle)

---

**Acknowledgement:** The course slides are adapted from the slides prepared by R. Sedgewick and K. Wayne of Princeton University.

http://bl.ocks.org/mbostock/39566ac95eb03dd0526
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 $a[j]$.

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

$lo \quad i \quad j$

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].

Lo K C A T E L E P U I M Q R X O S
\[\uparrow \quad \uparrow \quad \uparrow \]
\[\text{lo} \quad i \quad j\]

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 \( a[j] \).

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

\( \uparrow \) \( \uparrow \) \( \uparrow \)
lo i 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 \( a[j] \).

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

\( \uparrow \) \( \uparrow \) \( \uparrow \)
lo i 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 \( a[j] \).

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

\( \uparrow \) \( \uparrow \) \( \uparrow \)
lo i j
```

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 \( a[j] \).

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

\( \uparrow \) \( \uparrow \) \( \uparrow \)
lo i j
```

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 \( a[j] \).

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

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

\( \uparrow \)
\( \uparrow \)
\( \uparrow \)
\( i \ j \)

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

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

\( \uparrow \)
\( \uparrow \)
\( \uparrow \)
\( i \ j \)

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

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

\( \uparrow \)
\( \uparrow \)
\( \uparrow \)
\( i \ j \)

\( \text{partitioned!} \)
Quicksort partitioning

Basic plan.
• Scan i from left for an item that belongs on the right.
• Scan j from right for an item that belongs on the left.
• Exchange a[i] and a[j].
• Repeat until pointers cross.

Quicksort: Java code for partitioning

```java
private static int partition(Comparable[] a, int lo, int hi) {
    int i = lo, j = hi;
    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;
        if (j == lo) break;
        exch(a, i, j);
    }
    return i;
}
```

Quicksort trace

Quicksort: Java implementation

```java
public class Quick {
    private static int partition(Comparable[] a, int lo, int hi) {
        int i = lo, j = hi;
        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;
            if (j == lo) break;
            exch(a, i, j);
        }
        return i;
    }
}
```

Quicksort trace (array contents after each partition)
Quicksort animation

Quicksort: implementation details

Partitioning in-place. Using an extra array makes partitioning easier (and stable), but is not worth the cost.

Terminating the loop. Testing whether the pointers cross is a bit trickier than it might seem.

Staying in bounds. The \( i = h_i \) test is redundant (why?), but the \( i = h_i \) 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></td>
<td>computer thousand</td>
<td>million</td>
<td>billion</td>
</tr>
<tr>
<td>home</td>
<td>instant 2.8 hours</td>
<td>317 years</td>
<td>instant 1 second</td>
</tr>
<tr>
<td>super</td>
<td>instant 1 second</td>
<td>1 week</td>
<td>instant 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 \( \sim N \lg N \).

Each partitioning process splits the array exactly in half.
**Quicksort: worst-case analysis**

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

One of the subarrays is empty for every partition.

**Quicksort: average-case analysis**

**Proposition.** The average number of compares $C_N$ to quicksort an array of $N$ distinct keys is $\sim 2N \log_2 N$ (and the number of exchanges is $\sim \frac{1}{2}N \log_2 N$).

**Pf.** $C_N$ satisfies the recurrence

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

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

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

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

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

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

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

**Repeatedly apply above equation:**

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

- Substitute previous equation

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

- Approximate sum by an integral:

$$C_N = 2(N+1) \left( \frac{1}{3} + \frac{1}{4} + \frac{1}{5} + \ldots + \frac{1}{N+1} \right)$$

- Finally, the desired result:

$$C_N \sim 2(N+1) \log_2 N$$

**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 $\sim 1.39 N \log_2 N$.

- 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).

---

**Proposition.** Quicksort is not stable.

**Pf.**
- Can guarantee logarithmic depth by recurring on smaller subarray before larger subarray.
- Insertion sort small subarrays.
- Even quicksort has too much overhead for tiny subarrays.
- Cutoff to insertion sort for ≈10 items.
- Note: could delay insertion sort until one pass at end.

---

**Quicksort: practical improvements**

**Median of sample.**
- Best choice of pivot item = median.
- Estimate true median by taking median of sample.
- Median-of-3 (random) items.

\[
\begin{array}{cccc}
 i & j & 0 & 1 & 2 & 3 \\
 B_1 & C_1 & C_2 & A_1 \\
 1 & 3 & B_1 & C_1 & C_2 & A_1 \\
 1 & 3 & B_1 & A_1 & C_2 & C_1 \\
 0 & 1 & A_1 & B_1 & C_2 & C_1 \\
\end{array}
\]

**Quicksort with median-of-3 and cutoff to insertion sort: visualization**

- 12/7 N ln N compares (slightly fewer)
- 12/35 N ln N exchanges (slightly more)
Selection

Goal. Given an array of \( N \) items, find the \( k \)th largest.

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!

Quick-select: mathematical analysis

Proposition. Quick-select takes linear time on average.

Pf sketch.

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

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

\((2 + 2 \ln 2) N\) to find the median

Remark. Quick-select uses \( \frac{1}{2} N^2 \) compares in the worst case, but (as with quicksort) the random shuffle provides a probabilistic guarantee.

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 \( k \).

\[
\text{public static Comparable select(Comparable[] a, int k)} \{
\text{StdRandom.shuffle(a);}
\text{int lo = 0, hi = a.length - 1;}
\text{while (hi > lo) }
\text{int j = partition(a, lo, hi);}
\text{if (j < k) lo = j + 1;}
\text{else if (j > k) hi = j - 1;}
\text{else return a[k];}
\text{return a[k];}
\}
\]

Theoretical context for selection


Remark. But, constants are too high \( \Rightarrow \) not used in practice.

Use theory as a guide.

• Still worthwhile to seek practical linear-time (worst-case) algorithm.
• Until one is discovered, use quick-select if you don’t need a full sort.
**Duplicate keys**

Often, purpose of sort is to bring items with equal keys together.
- Sort population by age.
- Find collinear points.
- Remove duplicates from mailing list.
- Sort job applicants by college attended.

Typical characteristics of such applications.
- Huge array.
- Small number of key values.

**Mergesort with duplicate keys.**
Always between \( \frac{1}{2} N \lg N \) 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 \texttt{qsort()}. Several textbook and system implementation also have this defect.

**3-way partitioning**

**Goal.** Partition array into 3 parts so that:
- Entries between \( l_t \) and \( r_t \) equal to partition item \( v \).
- No larger entries to left of \( l_t \).
- No smaller entries to right of \( r_t \).

**Dutch national flag problem.** [Edsger Dijkstra]
- Conventional wisdom until mid 1990s: not worth doing.
- New approach discovered when fixing mistake in \texttt{C} library \texttt{qsort()}.
- Now incorporated into \texttt{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 $a[lit]$ 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 $i$

```
lt   i   gt
↓↓   ↓↓   ↓↓
P A B X W P P V P D P C Y Z
↓     ↓     ↓
lo   hi
```

Dijkstra 3-way partitioning

```
< v  = v  > v
↓  ↓  ↓
lt  i  gt
↓↓↓
A P B X W P P V P D P C Y Z
```
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

\[ \begin{align*}
\text{lt} & \quad i & \quad gt \\
A & \quad B & \quad P & \quad Z & \quad W & \quad P & \quad P & \quad V & \quad P & \quad D & \quad P & \quad C & \quad Y & \quad X \\
\end{align*} \]

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

\[ \begin{align*}
\text{lt} & \quad i & \quad gt \\
A & \quad B & \quad P & \quad Y & \quad W & \quad P & \quad P & \quad V & \quad P & \quad D & \quad P & \quad C & \quad Z & \quad X \\
\end{align*} \]
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 \( i \)

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

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 \( i \)

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

Dijkstra 3-way partitioning

Most of the right properties.
- In-place.
- Not much code.
- Linear time if keys are all equal.
Dijkstra’s 3-way partitioning: trace

```
0 1 2 3 4 5 6 7 8 9 10 11
0 0 11  R B  W W  R R  B R  W R  B R
0 1 11  R B  W W  R R  B R  W R  B R
1 2 11  R W  W W  B R  R R  B R  W R  B R
1 2 10  B R  R R  W B  W W  B R  R R  W B  R
1 3 10  B R  R R  W B  W W  B R  R R  W B  R
1 3  9  B R  R R  W B  W W  B R  R R  W B  R
2 4  9  B R  R R  W B  W W  B R  R R  W B  R
2 5  9  B R  R R  W B  W W  B R  R R  W B  R
2 5  8  B R  R R  W B  W W  B R  R R  W B  R
2 5  7  B R  R R  W B  W W  B R  R R  W B  R
2 6  7  B R  R R  W B  W W  B R  R R  W B  R
2 6  7  B R  R R  W B  W W  B R  R R  W B  R
3 7  7  B B  R R  W W  W W  B R  R R  W B  R
3 8  7  B B  R R  W W  W W  B R  R R  W B  R
3 8  7  B B  R R  W W  W W  B R  R R  W B  R
3 8  7  B B  R R  W W  W W  B R  R R  W B  R
```  
3-way partitioning trace (array contents after each loop iteration)

3-way quicksort: Java implementation

```
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);
}
```

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>N²/2</td>
<td>N²/2</td>
<td>N²/2</td>
<td>N exchanges</td>
</tr>
<tr>
<td>insertion</td>
<td>✔ ✔</td>
<td>N²/4</td>
<td>N²/4</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>2 N ln N</td>
<td>N lg N</td>
<td>N lg N</td>
<td>N log N probabilistic guarantee fastest in practice</td>
</tr>
<tr>
<td>3-way quick</td>
<td>✔</td>
<td>2 N ln N</td>
<td>N lg N</td>
<td>N lg N</td>
<td>improves quicksort in presence of duplicate keys</td>
</tr>
<tr>
<td>??</td>
<td>✔ ✔</td>
<td>N lg N</td>
<td>N lg N</td>
<td>N lg N</td>
<td>holy sorting grail</td>
</tr>
</tbody>
</table>