Two great sorting algorithms.

- Mergesort
  - Divide array into two halves.
  - Recursively sort each half.
  - Merge two halves to make sorted whole.

- Quicksort
  - Java sort for objects.
  - Cqsort, Unix qsort, Visual C++, Perl, Python.
  - Quick sort for primitive types.
  - Cqsort, Unix qsort, Visual C++, Perl, Python.

Full scientific understanding of their properties has enabled us to hammer them into practical system sorts.

Mergesort occupies a prominent place in world's computational infrastructure.

Quicksort honored as one of top 10 algorithms of 20th century in science and engineering.

Mergesort: Example

Mergesort: Java Implementation

```java
public class Merge {
    private static void sort(Comparable[] a, Comparable[] aux, int l, int r) {
        int m = (l + r) / 2;
        sort(a, aux, l, m);
        sort(a, aux, m + 1, r);
        merge(a, aux, l, m, r);
    }

    public static void sort(Comparable[] a) {
        Comparable[] aux = new Comparable[a.length];
        sort(a, aux, 0, a.length);
    }
}
```

Mergesort Analysis: Memory

Q. How much memory does mergesort require?
   - Original input array = N.
   - Auxiliary array for merging = N.
   - Local variables: constant.
   - Function call stack: \log_2 N.
   - Total = 2N + O(\log N).

Q. How much memory do other sorting algorithms require?
   - N + O(1) for insertion sort and selection sort.
   - In-place = N + O(\log N).

Challenge for the bored. In-place merge. [Kronrud, 1969]
Mergesort Analysis: Running Time

**Def.** \( T(N) \) = number of comparisons to mergesort an input of size \( N \).

Mergesort recurrence.

\[
T(N) = \begin{cases} 
0 & \text{if } N = 1 \\
T(\lceil N/2 \rceil) + T(\lfloor N/2 \rfloor) + \frac{N}{2} & \text{if } N > 1 
\end{cases}
\]

**Solution.** \( T(N) = O(N \log_2 N) \).

- Note: same number of comparisons for any input of size \( N \).
- We prove \( T(N) = N \log_2 N \) when \( N \) is a power of 2, and \( = \) instead of \( \leq \).

Proof by Induction

**Claim.** If \( T(N) \) satisfies this recurrence, then \( T(N) = N \log_2 N \).

\[
T(N) = \begin{cases} 
0 & \text{if } N = 1 \\
2T(N/2) + \frac{N}{2} & \text{if } N > 1, \text{ N power of 2}
\end{cases}
\]

**Pf.** [by induction on \( n \)]

- **Base case:** \( n = 1 \).
- **Inductive hypothesis:** \( T(n) = n \log_2 n \).
- **Goal:** show that \( T(2n) = 2n \log_2 (2n) \).

\[
T(2n) = 2T(n) + 2n \\
= 2n \log_2 n + 2n \\
= 2n \log_2 (2n) - 2n + 2n \\
= 2n \log_2 (2n)
\]

Proof by Recursion Tree

\[
T(N) = \begin{cases} 
0 & \text{if } N = 1 \\
2T(N/2) + \frac{N}{2} & \text{if } N > 1, \text{ N power of 2}
\end{cases}
\]

Mergesort: Practical Improvements

**Use sentinel.** Two statements in inner loop are array-bounds checking.

**Use insertion sort on small subarrays.**

- Mergesort has too much overhead for tiny subarrays.
- Cutoff to insertion sort for \( = 7 \) elements.

**Stop if already sorted.**

- Is biggest element in first half \( \leq \) smallest element in second half?
- Helps for nearly ordered lists.

**Eliminate the copy to the auxiliary array.** Save time (but not space) by switching the role of the input and auxiliary array in each recursive call.
Running time estimates:
- Home pc executes $10^8$ comparisons/second.
- Supercomputer executes $10^{12}$ comparisons/second.

### Lesson 1. Good algorithms are better than supercomputers.

- **Insertion Sort** ($N^2$):
  - Home: instant, 2.8 hours, 317 years
  - Super: instant, 1 second, 1.6 weeks

- **Mergesort** ($N \log N$):
  - Home: instant, 1 sec, 18 min
  - Super: instant, instant, instant

### Quicksort

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

**Q.** How do we partition in-place efficiently?

![Quicksort Partitioning](image)

```plaintext
input: BRAEMLSUMPQXOCR

1. Shuffle:
   - EXATBELMSORP
2. Partition:
   - EAMELDTPXRS
3. Sort left:
   - AEELMOPRSTX
4. Sort right:
   - AEELMOPRSTX
5. Result:
   - AEELMOPRSTX

Quick sort
```
Quicksort Example

private static int partition(Comparable[] a, int l, int r) {
    int i = l - 1;
    int j = r;
    while (true) {
        while (less(a[++i], a[j])) {
            if (i == r) break;
        }
        while (less(a[i], a[--j])) {
            if (j == l) break;
        }
        if (i >= j) break;
        exch(a, i, j);
    }
    exch(a, i, r);
    return i;
}

Quicksort: Java Implementation

public class Quick {
    public static void sort(Comparable[] a) {
        if (r <= l) return;
        int m = partition(a, l, r);
        sort(a, l, m - 1);
        sort(a, m + 1, r);
    }
}

public static int partition(Comparable[] a, int l, int r) {
    int i = l - 1;
    int j = r;
    while (true) {
        while (less(a[++i], a[j])) {
            if (i == r) break;
        }
        while (less(a[i], a[--j])) {
            if (j == l) break;
        }
        if (i >= j) break;
        exch(a, i, j);
    }
    exch(a, i, r);
    return i;
}

public class Quick {
    public static void sort(Comparable[] a) {
        StdRandom.shuffle(a);
        sort(a, 0, a.length - 1);
    }
    
    private static void sort(Comparable[] a, int l, int r) {
        if (r <= l) return;
        int m = partition(a, l, r);
        sort(a, l, m - 1);
        sort(a, m + 1, r);
    }
}

Partitioning in-place. Using a spare array makes partitioning easier, 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 \(i == r\) test is redundant, but the \(j == l\) test is not.

Preserving randomness. Shuffling is key for performance guarantee.

Equal keys. When duplicates are present, it is (counter-intuitively) best to stop on elements equal to partitioning element.
Quicksort: Performance Characteristics

Worst case. Number of comparisons is quadratic.
- \( N + (N-1) + (N-2) + \ldots + 1 \approx N^2 / 2. \)
- More likely that your computer is struck by lightning.

Caveat. Many textbook implementations go quadratic if input:
- Is sorted.
- Is reverse sorted.
- Has many duplicates.

Quicksort: Average Case

Average case running time.
- Roughly \( 2N \ln N \) comparisons. \( \rightarrow \) see next two slides
- Assumption: file is randomly shuffled.

Remarks.
- 39% more comparisons than mergesort.
- Faster than mergesort in practice because of lower cost of other high-frequency instructions.
- Caveat: many textbook implementations have best case \( N^2 \) if duplicates, even if randomized!

Theorem. The average number of comparisons \( C_N \) to quicksort a random file of \( N \) elements is about \( 2N \ln N \).

- The precise recurrence satisfies \( C_0 = C_1 = 0 \) and for \( N \geq 2 \):
  \[
  C_N = N + 1 + \frac{1}{N} \sum_{k=1}^{N-1} (C_k + C_{N-k})
  = N + 1 + \frac{1}{N} \sum_{k=1}^{N-1} C_{N-k}
  
  
  - Multiply both sides by \( N \) and subtract the same formula for \( N-1 \):
    \[
    NC_N - (N-1)C_{N-1} = N(N+1) - (N-1)N + 2C_{N-1}
    
  - Simplify to:
    \[
    NC_N = (N+1)C_{N-1} + 2N
    
  - Divide both sides by \( N(N+1) \) to get a telescoping sum:
    \[
    \frac{C_N}{N+1} = \frac{C_{N-1}}{N} + \frac{2}{N+1}
    = \frac{C_{N-1}}{N-1} + \frac{2}{N} + \frac{2}{N+1}
    = \frac{C_{N-1}}{N-2} + \frac{2}{N-1} + \frac{2}{N} + \frac{2}{N+1}
    = \ldots
    = \frac{C_3}{3} + \sum_{k=1}^{k} \frac{2}{N-k}
    
    - Approximate the exact answer by an integral:
      \[
      \frac{C_N}{N+1} = \sum_{k=1}^{N} \frac{2}{k} = \int_{1}^{N} \frac{2}{x} = 2 \ln N
      
    - Finally, the desired result:
      \[
      C_N = 2(N+1) \ln N = 1.39N \log_2 N. \]

Remarks.
- \( 39\% \) more comparisons than mergesort.
- Faster than mergesort in practice because of lower cost of other high-frequency instructions.
- Caveat: many textbook implementations have best case \( N^2 \) if duplicates, even if randomized!
3-Way Quicksort

Sorting Analysis Summary

Running time estimates:
- Home pc executes $10^8$ comparisons/second.
- Supercomputer executes $10^{12}$ comparisons/second.

<table>
<thead>
<tr>
<th>Computer</th>
<th>thousand</th>
<th>million</th>
<th>billion</th>
</tr>
</thead>
<tbody>
<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>1.6 weeks</td>
</tr>
</tbody>
</table>

Mergesort ($N \log N$)

<table>
<thead>
<tr>
<th>Computer</th>
<th>thousand</th>
<th>million</th>
<th>billion</th>
</tr>
</thead>
<tbody>
<tr>
<td>instant</td>
<td>1 sec</td>
<td>18 min</td>
<td></td>
</tr>
</tbody>
</table>

Quicksort ($N \log N$)

<table>
<thead>
<tr>
<th>Computer</th>
<th>thousand</th>
<th>million</th>
<th>billion</th>
</tr>
</thead>
<tbody>
<tr>
<td>instant</td>
<td>0.3 sec</td>
<td>6 min</td>
<td></td>
</tr>
<tr>
<td>instant</td>
<td>instant</td>
<td>instant</td>
<td></td>
</tr>
</tbody>
</table>

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

Quicksort: Practical Improvements

Median of sample.
- Best choice of pivot element = median.
- But how would you compute the median?
- Estimate true median by taking median of sample.

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

Optimize parameters.
- Median-of-3 random elements.
- Cutoff to insertion sort for ≈ 10 elements.

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

Duplicate Keys

Equal keys. Omnipresent in applications when purpose of sort is to bring records with equal keys together.
- Sort population by age.
- Finding collinear points.
- Remove duplicates from mailing list.
- Sort job applicants by college attended.

Typical application.
- Huge file.
- Small number of key values.
3-Way Partitioning

3-way partitioning. Partition elements into 3 parts:
  - Elements between i and j equal to partition element v.
  - No larger elements to left of i.
  - No smaller elements to right of j.

Dutch national flag problem.
  - Not done in practical sorts before mid-1990s.
  - Incorporated into Java system sort, C qsort.

Duplicate Keys

Theorem. [Sedgewick-Bentley] Quicksort with 3-way partitioning is optimal for random keys with duplicates.
Pf. Ties cost to entropy. Beyond scope of 226.

Practice. Randomized 3-way quicksort is linear time when many duplicates. (Try it!)
Selection

Quick select.
- Partition array so that:
  - element $a[i]$ is in its final place for some $i$
  - no larger element to the left of $i$
  - no smaller element to the right of $i$
- Repeat in one subarray, depending on $i$.

```java
public static void select(Comparable[] a, int k) {
    StdRandom.shuffle(a);
    int l = 0;
    int r = a.length - 1;
    while (r > l) {
        int i = partition(a, l, r);
        if (i > k) r = i - 1;
        else if (i < k) l = i + 1;
        else return;
    }
    // upon termination, a[k] contains kth smallest element
}
```

Selection

Find the $k^{th}$ largest element.
- Min: $k = 1$.
- Max: $k = N$.
- Median: $k = N/2$.

Application. Order statistics.

Easy. Min or max with $O(N)$ comparisons; median with $O(N \log N)$.

Challenge. $O(N)$ comparisons for any $k$.

Quick-Select Analysis

Property C. Quick-select takes linear time on average.
- Intuitively, each partitioning step roughly splits array in half.
- $N + N/2 + N/4 + \ldots < 2N$ comparisons.
- Formal analysis similar to quicksort analysis proves the average number of comparisons is
  $$2N + k \ln \left(\frac{N}{k}\right) + (N-k) \ln \left(\frac{N}{N-k}\right)$$
  
  Ex: $(2 + 2 \ln 2) N$ comparisons to find the median

Worst-case. The worst-case is $\Omega(N^2)$ comparisons, but as with quicksort, the random shuffle makes this case extremely unlikely.