Quick sort

Divide-and-conquer

1) Pick an arbitrary element called the pivot.

2) Partition into three subarrays:

   - pivot
   - pivot
   - > pivot
3) Recursively sort the left and right subarrays.

Any pivot is a quicksort. Ideally, pick one near
middle.

Want some \( \Theta(n) \) time partition. Can be tricky to write.

On \( n \geq 10 \) elements, insertion sort is faster, so make \( n \leq 10 \) elements a base case. Insertion sort then.
public static <T extends Comparable<? super T>> void quicksort(T[] a) {
    quicksort(a, 0, a.length - 1);
}

private static final int CUTOFF = 10;

public static <T extends Comparable<? super T>>
void quicksort(T[] a, int left, int right) {
    if (left + CUTOFF > right) {
        insertionSort(a, left, right);
        return;
    }

    int pivotPos = left; // Pick an arbitrary pivot.

    // Partition the array.
    swapReferences(a, pivotPos, right); // Move pivot out of the way.
    T pivot = a[right];
    int i = left - 1, j = right;
    while (true) {
        while (a[++i].compareTo(pivot) < 0) { } // Skip past smaller elements.
        while (a[--j].compareTo(pivot) > 0) { } // Skip past larger elements.
        if (i < j) { // a[i] is too large, and a[j] is too big, so swap!
            swapReferences(a, i, j);
        } else {
            break; // Done partitioning.
        }
    }

    swapReferences(a, i, right); // Move pivot to final location.

    quicksort(a, left, i - 1); // Sort smaller elements.
    quicksort(a, i + 1, right); // Sort larger elements.
}
Suppose pivot goes in position \( r \). 

- \( r \) smaller elements
- \( n - r - 1 \) larger elements

\( T(n) \): worst case time to quick sort \( n \) elements.

\[
T(n) = \max_{0 \leq r \leq n-1} (T(r) + T(n-r-1)) + \Theta(n)
\]
If \( r = 0 \) every time:

Total of \( \Omega(n^2) \), so

\[
T(n) \geq \Omega(n^2)
\]

\[
T(n) = \Theta(n^2)
\]

\[
\Rightarrow T(n) = \Theta(n^2)
\]
Median of three partitioning

Look at positions $0, n-1, \left\lfloor \frac{n-1}{2} \right\rfloor$

Use median of those three elements as the pivot.

Probably $O(n \log n)$ time. Could still be $\Omega(n^2)$!
"Simple" theoretically good solution:

Pick pivot uniformly at random.

Expected time of \( O(n \log n) \).

But grabbing good random numbers is relatively slow.
Mergesort:
+ Fewest comparisons of major algorithms.
- Lots of moving elements between arrays.
Java standard library's generics sorts
Quicksort:
- More comparisons
+ Fewer copies, better cache performance.
C++ standard library implementations
Java primitive sorting
Quick select

Find to find the kth smallest (or largest) element in sorted order, it selection problem.

Quick sort, but you figure out which subarray has element you want to do only its recursive.
call. Target ultimately in position k-1. Good / random pivots give $O(n)$ time. Worst-case $\Theta(n^2)$.

(usually want the median - $k = \lceil n/2 \rceil$)
Lower bound:

Only for comparison sorts.

Decision tree:

Rooted tree.

Each node represents some kind of info gathering operation/ question.
Children are the next possible operation.

Leaves are final answers output.

Decision trees describe some fixed algorithm with some fixed input size.
A deterministic sort

nodes represent comparisons

⇒ nodes have two children

leaves are the permutations to sort elements
Sort a, b, c

At least $n!$ leaves.

$\Rightarrow$ Height $\geq \log(n!)$
Worst-case comparisons is height of tree.

So our arbitrary comparison sort does $\geq$

$$\lg (n!) = \lg (n (n-1) (n-2) \ldots (2) 1)$$

$$\geq \lg n + \lg (n-1) + \ldots + \lg 2 + \lg 1$$

$$\geq \lg n + \lg (n-1) + \ldots + \lg \left(\frac{n}{2}\right)$$

$$\geq \frac{n}{2} \lg \left(\frac{n}{2}\right)$$
= \Omega(n \log n)
Radix sort
(not a comparison sort)

Say we were sorting ints between 0 and \(2^N-1\).

1) Sort by most significant bit.

0...0...0...0...0...1...1...1...

2) Sort each group by...
2nd most sig. bit, then each group by 3rd, 4th, etc.

Each int is seen once per phase.

64 = O(1) phases, so O(n) time.
Or could sort by least sig. bit, then 2nd least, 3rd least, etc.

Works if partition is stable: doesn't change order for ties.
Can sort strings alphabetically one letter at a time. If all strings have length $\leq w$, $O(wn)$ time to radix sort. Any comparison sort could take $\Omega(wn \log n)$. 
A multi-phase bucket sort

count how many of each district element there are.