Note about Merge Sort

- Typically done with a second array
- Why?
  - Since elements in both halves are already sorted, easier to compare elements at the beginning of the list/array
  - More memory allows faster movement of data

Note about Merge Sort

- If odd number of elements, you have two different options for splitting
  - Option 1: split so that the left half has one more than the right half (you are tested on this way)
  - Option 2: split so that the right half has one more than the left half
  - Either option works as long as you are consistent in your splitting (i.e. always give more to the left or always give more to the right)

Note about Merge Sort

- The way that you split matters!
- You must merge with the numbers you split from
  - Has to do with the way recursion works
  - Recursion breaks the sorting problem into a smaller sorting problem
    - Have to look at each half as a new sorting problem
    - Cannot sort with numbers from another problem unless you are merging with numbers you previously split from

Key Features of Algorithms

- Selection Sort
  - Swapping lowest (or highest) value
- Insertion Sort
  - Shifting values and inserting next value in correct spot
- Merge Sort
  - Recursive splitting
  - Recursive sorting and merging

Bubble Sort

- Starting from the beginning of the list, compare every adjacent pair, swap their position if they are not in the right order.
- After each iteration, one less element (the last one) is needed to be compared until there are no more elements left to be compared.

Quick Sort

- Pick an element, called a pivot, from the list.
- Reorder the list so that
  - all elements with values less than the pivot come before the pivot
  - all elements with values greater than the pivot come after it
  - equal values can go either way
  - After this partitioning, the pivot is in its final position. This is called the partition operation.
- Recursively sort the sub-list of lesser elements and the sub-list of greater elements.

Efficiency

- What is it?
- Why do we care?
- How do we measure?

What does "efficiency" mean?

- Measure of how long the algorithm takes to run to completion relative to the number of inputs it is given
  - For our sorting algorithms, inputs are arrays, ArrayLists, other lists, etc.

Why do we care?

- Inefficient algorithms require more calculations/comparisons/memory
  - Wasted time (→ wasted money)
  - Wasted power (→ being less "green")

How do we measure efficiency?

- Time? (Wall-clock time or execution time)
  - Advantages: easy to measure, meaning is obvious
  - Appropriate if you care about actual time
    - (e.g. real-time systems)
  - Disadvantages: applies only to specific data set, compiler, machine, etc.
- Typically not a good idea to measure by actual time
  - Computers perform at different speeds

How do we measure efficiency?

- Better idea? Number of times certain statements are executed:
  - Advantages: more general (not sensitive to speed of machine).
  - Disadvantages: doesn't tell you actual time, still applies only to specific data sets.

How do we measure efficiency?

- Symbolic execution times:
  - That is, *formulas* for execution times or statement counts in terms of input size.
  - Advantages: applies to all inputs, makes scaling clear.
  - Disadvantage: practical formula must be approximate, may tell very little about actual time.

How do we measure efficiency?

- Since we are approximating anyway, pointless to be precise about certain things:
  - Behavior on small inputs:
    - Can always pre-calculate some results.
    - Times for small inputs not usually important.
  - Constant factors (as in "off by factor of 2"):
    - Just changing machines causes constant-factor change.
- How to abstract away from (i.e., ignore) these things?
  - Order notation

How do we measure efficiency?

- Big O notation
- Number of calculations/comparisons
  - Best Case (lower bound)
  - Worst Case (upper bound)
  - Average Case
- Consider: Amount of space/memory used
  - Extra space necessary for algorithm

Why does it matter again?

- Computer scientists often talk as if constant factors didn't matter at all, only the difference of $\Theta(N)$ vs. $\Theta(N^2)$.
- In reality they do, but we still have a point: at some point, constants get swamped.

n	$16 \lg n$	$\sqrt{n}$	n	$n \lg n$	n^2	n^3	2^n
2	16	1.4	2	2	4	8	4
4	32	2	4	8	16	64	16
8	48	2.8	8	24	64	512	256
16	64	4	16	64	256	4,096	65,636
32	80	5.7	32	160	1024	32,768	4.2×10^9
64	96	8	64	384	4,096	262,144	1.8×10^{19}
128	112	11	128	896	16,384	2.1×10^9	3.4×10^{38}
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮
1,024	160	32	1,024	10,240	1.0×10^6	1.1×10^9	1.8×10^{308}
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮
2^{20}	320	1024	1.0×10^6	2.1×10^7	1.1×10^{12}	1.2×10^{18}	$6.7 \times 10^{315,652}$

Random or Deterministic

- Random → different every time
- Deterministic → consistent every time
- Which is more efficient?
  - E.g. int[] arr = {11, 12, 13, 14, 15, 16, 17, 18};
    Randomly find index of 18?
    Deterministically find index of 18?
  - Would it work if 18 was not in the array?

Iterative or Recursive

- Iterative more efficient in many cases
- Recursive can be more efficient in some cases (e.g. merge sort)
- Consider two different algorithms for calculating the 40th Fibonacci number
  - Recursive takes a really long time
  - Iterative only does each calculation once

Big O

- It measures the *growth rate* of the time the algorithm takes to complete with respect to the amount of data it is acting on.
- We write the big O of an algorithm as O(f(n)), where f(n) is a function of n.
  - E.g. $O(n^2)$ is pronounced "Oh of en squared".
  - It is a function, O, of the time an algorithm takes to run.
- Approximation means no constants, no coefficients, etc.

Big O

- Linear: $O(n)$
  - E.g. Searching an array of length n
- Quadratic: $O(n^2)$
  - E.g. Searching an 2-D array of size n by n
  - E.g. Comparing every element in a 1-D array of size n to every other element in the same array
- Exponential: $O(x^n)$
  - x is some constant
  - E.g. Recursive Fibonacci is $O(2^n)$

Big O

- Logarithmic: $O(\log n)$
  - for each element acted on, eliminate some fraction of the remaining inputs
  - E.g. repeatedly print the left half of an array or String
  - Cannot reduce by a constant amount, must be a consistent fraction
  - O(n log n) → merge sort
- Constant: $O(1)$
  - Same amount of time regardless of inputs
  - E.g. selecting a random element and printing it