8. Average-Case Analysis of Algorithms + Randomized Algorithms
1) Probability tools you've seen allow formal definition of "average case" running time of algorithms

2) Coupled with a few analysis tricks you'll see in more detail in 421 or elsewhere, you can analyze those algorithms, and

3) Adding randomness to algorithms can have surprising benefits, and again, you've got the basic tools needed to understand the issues and do the necessary analysis

4) Specifics: “average” case analysis of insertion sort and quicksort, and randomized quicksort

for $i = 2 \ldots n-1$ {
    $T = A[i]$
    $j = i - 1$
    while $j \geq 0$ && $T < A[j]$
        $A[j] = T$
        $j = j - 1$
    }

$A[j+1] = T$
Run Time

Worst Case: $O(n^2)$

( (n choose 2) swaps; \#comparisons = \#swaps + n - 1)

“Average Case”

? What’s an “average” input?

One idea (and about the only one that is analytically tractable): assume all $n!$ permutations of input are equally likely.
A permutation $\pi = (\pi_1, \pi_2, \ldots, \pi_n)$ of 1, ..., n is simply a list of the numbers between 1 and n, in some order. 

(i,j) is an inversion in $\pi$ if $i < j$ but $\pi_i > \pi_j$

E.g., 

$$\pi = (3 \ 5 \ 1 \ 4 \ 2)$$

has six inversions: (1,3), (1,5), (2,3), (2,4), (2,5), and (4,5)

Min possible: 0: 
$$\pi = (1 \ 2 \ 3 \ 4 \ 5)$$

Max possible: n choose 2:
$$\pi = (5 \ 4 \ 3 \ 2 \ 1)$$

Obviously, the goal of sorting is to remove inversions
Swapping an *adjacent* pair of positions that are *out-of-order* decreases the number of inversions by *exactly 1*. So..., number of swaps performed by insertion sort is exactly the number of inversions present in the input.

Counting them:

a. worst case: \( n \text{ choose } 2 \)

b. average case:

\[
I_{i,j} = \begin{cases} 
1 & \text{if } (i, j) \text{ is an inversion} \\
0 & \text{if not} 
\end{cases}
\]

\[
I = \sum_{i<j} I_{i,j}
\]

\[
E[I] = E \left[ \sum_{i<j} I_{i,j} \right] = \sum_{i<j} E \left[ I_{i,j} \right]
\]
There is a 1-1 correspondence between permutations having inversion \((i,j)\) versus not:

\[
\pi \quad \begin{array}{cccccc}
\ldots & a & \ldots & b & \ldots \\
\pi' \quad \begin{array}{cccccc}
\ldots & b & \ldots & a & \ldots \\
\end{array}
\end{array}
\]

So:

\[
E[I_{i,j}] = P(I_{i,j} = 1) = \frac{1}{2}
\]

\[
E[I] = \sum_{i<j} E[I_{i,j}] = \sum_{i<j} \frac{1}{2} = \binom{n}{2} \cdot \frac{1}{2}
\]

Thus, the expected number of swaps in insertion sort is \(\binom{n}{2}/2\) versus \(\binom{n}{2}\) in worst-case. I.e.,

The average run time of insertion sort (assuming random input) is about half the worst case time.
Quicksort also does swaps, but *non* adjacent ones.

Recall method:

Array $A[1..n]$


2. “Partition” ( $O(n)$ compares/swaps ) so that:

   $\{A[1], ..., A[i-1]\} < \{A[i] == \text{pivot}\} < \{A[i+1], ..., A[n]\}$

3. recursively sort $\{A[1], ..., A[i-1]\}$ & $\{A[i+1], ..., A[n]\}$
Worst case: already sorted (among others) –
\[ T(n) = n + T(n-1) \Rightarrow \]
\[ = n + (n-1) + (n-2) + \ldots + 1 = n(n+1)/2 \]

Best case: pivot is always median
\[ T(n) = 2 \cdot T(n/2) + n \]
\[ \Rightarrow \sim n \log_2 n \]

Average case: ?

Below. Will turn out to be \sim 40\% slower than best

Why?

Random pivots are “near the middle on average”
average-case analysis

Assume input is a random permutation of 1, ..., n, i.e., that all n! permutations are equally likely

Then 1st pivot A[1] is uniformly random in 1, ..., n

Important subtlety:
	pivots at all recursive levels will be random, too,
(unless you do something funky in the partition phase)
number of comparisons

Let $C_N$ be the average number of comparisons made by quicksort when called on an array of size $N$. Then:

$C_0 = C_1 = 0$ (a list of length $\leq 1$ is already sorted)

In the general case, there are $N-1$ comparisons: the pivot vs every other element (a detail: plus 2 more for handling the “pointers cross” test to end the loop). The pivot ends up in some position $1 \leq k \leq N$, leaving two subproblems of size $k-1$ and $N-k$. By Law of Total Expectation:

$$C_N = N + 1 + \frac{1}{N} \sum_{1 \leq k \leq N} (C_{k-1} + C_{N-k}) \quad \text{for } N \geq 2,$$

$I/N$ because all values $1 \leq k \leq N$ for pivot are equally likely.

\[ C_N = N + 1 + \frac{1}{N} \sum_{1 \leq k \leq N} (C_{k-1} + C_{N-k}) \quad \text{for } N \geq 2, \]

\[ C_N = N + 1 + \frac{2}{N} \sum_{1 \leq k \leq N} C_{k-1}. \]

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

\[ NC_N = (N + 1)C_{N-1} + 2N. \]
\[ NC_N = (N + 1)C_{N-1} + 2N. \]
\[
\frac{C_N}{N+1} = \frac{C_{N-1}}{N} + \frac{2}{N+1}
\]
\[
= \frac{C_{N-2}}{N-1} + \frac{2}{N} + \frac{2}{N+1}
\]
\[
= \quad \vdots \quad 
\]
\[
= \frac{C_2}{3} + \sum_{3 \leq k \leq N} \frac{2}{k+1}.
\]
\[
\frac{C_N}{N+1} \approx 2 \sum_{1 \leq k < N} \frac{1}{k} \approx 2 \int_1^N \frac{1}{x} \, dx = 2 \ln N,
\]

\[ 2N \ln N \approx 1.39N \lg N \]
So, *average* run time, averaging over *randomly ordered inputs*, \( = \Theta(n \log n) \).

A worst case input is still worst case: \( n^2 \) every time

(Is real data random?)

Is it possible to improve the worst case?
another idea: randomize the algorithm

Algorithm as before, except pivot is a \textit{randomly selected} element of $A[1]...A[n]$ \textit{(at top level; $A[i]..A[j]$ for subproblem $i..j$)}

Analysis is the same, but conclusion is different:

On \textit{any} fixed input, average run time is $n \log n$, \textit{averaged over repeated (random) runs of the algorithm}.

There are no longer any “bad inputs”, just “bad (random) choices.” Fortunately, such choices are improbable!
Average Case Analysis (of a deterministic alg):

1. for algorithm A, choose a sample space S and probability distribution P from which inputs are drawn
2. for \( x \in S \), let \( T(x) \) be the time taken by A on input \( x \)
3. calculate, as a function of the “size,” \( n \), of inputs,
   \[ \sum_{x \in S} T(x) \cdot P(x) \]
   which is the expected or average run time of A

For sorting, distrib is usually “all \( n! \) permutations equiprobable”

Insertion sort: \( E[\text{time}] \propto E[\text{inversions}] = \binom{n}{2}/2 = \Theta(n^2) \), about half the worst case

Quicksort: \( E[\text{time}] = \Theta(n \log n) \) vs \( \Theta(n^2) \) in worst case; fun with recurrences, sums & integrals
Randomized Algorithms (with non-random input):

1. for a randomized algorithm $A$, *input* $x$ is fixed, just as usual, from some space $I$ of possible inputs, but the algorithm may draw (and use) random samples $y = (y_1, y_2, ...)$ from a given sample space $S$ and probability distribution $P$. E.g., $y_i =$ “which pivot in subproblem $i$”

2. for *any* $x \in I$ and any $y \in S$, let $T(x,y)$ be the time taken by $A$ on input $x$ when $y$ is sampled from $S$

3. calculate, as a function of the “size,” $n$, of inputs,
\[
\max_{x \in I} \sum_{y \in S} T(x,y) \cdot P(y)
\]
   which is the expected or average run time of $A$ on a worst-case input

Randomized Quicksort: choosing pivots at random,
$E[\text{time}] = \Theta(n \log n)$ for *any* input. (For every input, there are some rare random choice sequences causing $n^2$ time.)
Key distinction:

If average case analysis of a (deterministic) algorithm $D$ says that average runtime $\ll$ worst case, then worst case inputs must be rare. But if you get one, your bad luck is permanent: $D$ will be slow time after time after time on that input…

If expected run time of a randomized algorithm $R$ is $\ll$ worst case, some inputs may be worse than others, but there are no bad inputs. If $R$ runs slowly (near worst case) once, on a specific input, your bad luck is transient; if you run it again you can expect it to run near the overall expectation.
Worst-case analysis is much more common than average-case analysis because:

- It’s often easier.
- To get meaningful average case results, a reasonable probability model for “typical inputs” is critical, but may be unavailable, or difficult to analyze.
- The results are often similar (e.g., insertion sort).

But in some important examples, average-case is sharply better (e.g., quicksort).

Randomized algorithms are very important in many areas; sometimes easier to argue that bad stuff is rare than to deterministically circumvent it (e.g., randomized qsort).

Fascinating and deep open problem: is this intrinsic?