All of the algorithms we have seen thus far (at least when fully coded) are **deterministic**, in that they will always behave the same way on the same input. There is sometimes an advantage to **randomized** algorithms, which are **nondeterministic** and make their choices randomly. We will see that randomization may provide a faster or simpler solution than the best known deterministic algorithm.

Randomization should not be confused with the idea of **average-case analysis**, though the two areas share some of the same mathematical tools. The average-case complexity of a problem is defined in terms of some probability distribution *on the possible inputs*. This always raises the question of whether this distribution accurately represents the natural occurrence of the inputs.

In a randomized algorithm the choices are made *by the algorithm* and the distribution of the choices is well-defined. We normally analyze such an algorithm in terms of the **worst-case expected complexity**. For example, suppose that on input x and choice sequence c the algorithm takes $T(x, c)$ time. The expected running time on a particular input x is $\overline{T}(x) = E_c(T(x, c))$, which is the sum over all c of $Pr(c)T(x, c)$. The overall worst-case expected running time $\overline{T}(n)$ is the maximum, over all x of size n, of $\overline{T}(x)$.

In a **Las Vegas** randomized algorithm, the answer is guaranteed to be correct no matter which random choices the algorithm makes. But the running time is a random variable, making the expected running time the most important parameter. Our first algorithm today, Quicksort, is a Las Vegas algorithm.

In a **Monte Carlo** randomized algorithm, on the other hand, the running time is bounded for all possible choices, but only *most* of the choices lead to the correct answer. If the algorithm is to be practical, we need to either:

- Have a way of checking an alleged answer to confirm that it is right. In this case we can repeat the Monte Carlo algorithm until we get a confirmed answer – this becomes a Las Vegas algorithm.
- Repeat the Monte Carlo algorithm many times and draw a conclusion from the distribution of the answers. For example, if the answer is a bit and we know that $3/4$ of the random choices give the right answer, we will show that it is overwhelmingly likely that the majority answer of many trials of the algorithm will be correct. With the Karger algorithm later in this lecture, we will need a more subtle analysis to get high confidence of our answer.

Of course when we repeat the same algorithm on the same input, we will need to know that the random choices in the various trials are **independent** of each other, so that we can use the tools of statistics to draw conclusions from the answers.

Quicksort

Quicksort is one of the basic sorting algorithms normally encountered in a data structures course. The idea is simple:

- Choose a pivot element p
- Divide the elements into those less than p, those equal to p, and those greater than p
- Recursively sort the "less than" and "greater than" groups
- Assemble the results into a single list

The algorithm as written here is **underspecified** because we haven't said how to choose the pivot. The choice turns out to be important! Any pivot will have some number k of elements less than it, and thus at most $n-k-1$ elements greater than it. The non-recursive parts of the algorithm pretty clearly take $O(n)$ time. We thus get a recurrence:

$$T(n) \leq T(k) + T(n-k-1) + O(n)$$

$$T(n) \leq T(k) + T(n-k-1) + O(n)$$

We can't analyze this recurrence without knowing something about k. The worst case is when $k = 0$ or $k = n-1$ – when the pivot element is the largest or smallest in the set. The recurrence then becomes $T(n) = T(n-1) + O(n)$, which has solution $O(n^2)$, very bad for a sorting algorithm.

But if we always knew that the pivot was the median element, we would have a recurrence $T(n) = 2T(n/2) + O(n)$, which we know solves to $T(n) = O(n \log n)$, the best we can hope for asymptotically for a comparison-based sorting algorithm.

Wait a minute, didn't we say we could find the median in linear time? We can, and we will in a few lectures, but it's not very *good* linear time. The bad constant gives us a bad constant in the $O(n \log n)$ time. And since the great virtue of Quicksort is that its running time is better than the other $O(n \log n)$ sorts *in its constant factor*, this is not the way to go.

The pivot-choosing method we will analyze is the most natural randomized one – we choose a pivot uniformly at random from all the items in the range we are currently sorting. In the real world we *don't* do this, because getting random numbers is relatively expensive. A typical scheme is to take the first, last, and middle element and choose the median of those three, which we can find with three comparisons.

If the order on the inputs is uniformly distributed, it doesn't matter how we choose the pivot – the expected result will be the same as for the randomized algorithm because the element we choose is equally likely to belong anywhere in the range. But choosing the first or the last element runs badly on some very likely orders, and even the median-of-three can run badly on a plausible set of orders.

For more about real Quicksort, see the following paper, a classic of the computer science literature:

Bentley and McIlroy, "Engineering a Sort Function", *Software Practice and Experience*, June 1993.

Here we will analyze the expected number of comparisons taken by the randomized version of Quicksort and show that it is $O(n \log n)$. The intuition behind this result is clear. If the pivot were always in the middle half of the input, each sublist would have at most $3n/4$ elements, and the depth of our recursion would be $O(\log n)$. (Note that the Master Theorem would *not* prove $O(n \log n)$ here, as $T(n) = 2T(3n/4)$ solves to $O(n^{\log_{4/3} 2})$, which is worse than quadratic.) Each item would be affected at most $O(\log n)$ times, so the total time would be $O(n \log n)$. It's not true that all the pivots are in the middle half, but *half of them* are, and the others don't do any *harm*, so we should get $O(n \log n)$. But we'll carry on with the more precise and more elegant argument from the notes.

The clever idea is to look at an arbitrary pair of elements and determine the probability that they will be compared in the course of the algorithm. Once we have this, we can use **linearity of expectations** to find the overall expected number of comparisons.

We let Z_{ij} be 1 if items i and j (in the true order) are ever compared, and 0 if they are not. The total number of comparisons is $\Sigma_{i<j} Z_{ij}$, and thus the expected total number is the sum of the expected values of the variables Z_{ij}. (Expected values add *whether or not* the random variables are independent.)

But because Z_{ij} has value 0 or 1, its expected value is just the probability that it is 1. We can compute this from i and j. Consider the elements from i through j in the true order. If i or j is the first element in this range chosen as a pivot, we will then compare i and j. If one of the $j-i-1$ other elements is chosen first, i will never be compared with j because they will go on different sides of the partition on that pivot. This tells us that the probability, and thus the expected value of Z_{ij}, is $\frac{2}{j-i+1}$.

Our expected number of comparisons is the sum, over all pairs $\{i, j\}$ with $1 \leq i < j \leq n$, of $\frac{2}{j-i+1}$. For each i, we have the sum over all $j > i$ of $\frac{2}{j-i+1}$, which is

$$2[1 + (1/2) + (1/3) + \ldots + (1/(n - i + 1)],$$

which is bounded above by the **Harmonic number**

$$H_n = \sum_{k=1}^{n} \frac{1}{k} < \ln n.$$

Since we have at most $2 \ln n$ expected comparisons for each i (and replacing H_{n-i+1} by $\ln n$ was not a big over-estimate), our total expected number of comparisons is at most $2n \ln n$ and probably not much less than that. We have established an $O(n \log n)$ bound, and have evidence of a rather small constant.

We now turn to a randomized algorithm to find a **minimum cut** in an undirected graph $G = (V, E)$. Here a cut is defined to be a set of edges C such that $(V, E \setminus C)$ is not connected, and a minumum cut has the minumum number of edges (unlike the minimum *capacity* cuts we studied in flow networks).

So the minimum cut of an unconnected graph is the empty set, and the size of the minimum cut is at most the minimum degree of any vertex (since cutting all the edges at one vertex separates it from the rest of the graph). This looks like the network flow problem, and in fact we can use network flow to find the minimum cut as follows: Fix a vertex s, and for each t set up a flow network from s to t with unit capacity in each direction for each undirected edge in E. The maximum flow in each network gives us a minimum cut among those that separate s from that t, and the minimum among these $n - 1$ cuts is a minimum among all cuts.

This is a poly-time algorithm, but recall that our implementation of Ford-Fulkerson with the Edmonds-Karp heuristic took $O(e^2n)$ time, which is $O(n^5)$ for a dense graph. Thus it might take us $O(n^6)$ to find the minimum cut. There is a better network-flow algorithm due to Karzanov that runs in $O(n^3)$, but this still gives us only $O(n^4)$ for the minimum cut.

Karger's Algorithm is a simple randomized method that *usually* finds the minimum cut. It is a Monte Carlo algorithm, and thus has some probability of giving the wrong answer, but we will show how to adjust the algorithm to make this probability smaller than any desired limit ϵ. The version we present will still take $O(n^4 \log(1/\epsilon))$ time, but there is a more involved version that runs in $O((n \log n)^2 \log(1/\epsilon))$ time.

The basic idea will be to choose a cut by a random process that has a *nontrivial* chance of producing the minimum cut, specifically at least $2/n^2$. If we run the basic procedure repeatedly, with independent random choices, the probability that the minimum of the resulting cuts is an actual minimum increases in a way that we can analyze.

The basic algorithm is simple. We maintain an **undirected multigraph** that starts off as the undirected graph G:

- While there are more than two vertices left,
- Choose an edge (u, v) uniformly at random
- Merge u and v, preserving all edges except those between u and v
- Return the edges between the last two vertices

I'll work through an example on the whiteboard. At the end of the algorithm we have a cut, because the two remaining vertices each represent a nonempty set of vertices and all the edges between the two sets have been preserved. But the cut is certainly not guaranteed to be minimum, because any edge has some chance of being chosen and removed.

A simple implementation of this algorithm would keep the vertices in a union-find data structure and make a pass through the edge list on each phase, to remove edges between equivalent vertices. This gives $O(ne) = O(n^3)$ time for the basic algorithm. With a different data structure we can do it in $O(n^2)$ – this is an exercise in *Randomized Algorithms* by Motwani and Raghavan.

Our main task here is to show that the basic algorithm has a nontrivial, at least $2/n^2$, chance of returning a minimum cut. Suppose that C is a particular minimum cut with k edges. Since every vertex of the graph has degree at least k, there are at least $kn/2$ edges and the probability that an edge of C is the first one chosen is at most $k/(nk/2) = 2/n$.

If all the edges of C have survived to a later point in the algorithm where there are ℓ vertices remaining, then because the degree of the graph is still at least k we have at most a probability of $k/(k\ell/2) = 2/\ell$ that an edge of C will be the next one chosen. We can thus place a lower bound on the probability that C will survive the entire process. There is at least a $1 - (2/n)$ chance of surviving the first choice, $1 - (2/(n-1))$ of surviving the second given that it survives the first, and so on:

$$P \geq \frac{n-2}{n} \cdot \frac{n-3}{n-1} \cdot \frac{n-4}{n-2} \cdot \ldots \cdot \frac{2}{4} \cdot \frac{1}{3}$$

The first $n-4$ terms on the top cancel with the last $n-4$ terms on the bottom, to give $\frac{2 \cdot 1}{n(n-1)}$ which is greater than $2/n^2$.

The chance that a single run of the basic Karger algorithm will return the wrong answer is thus high, but no higher than $1 - (2/n^2)$. We now have to calculate, for a given ϵ, how many times we have to run the basic algorithm to be assured that the chance that *all* the answers are wrong is less than ϵ.

Since the probabilities of independent events multiply, the probability of m wrong answers in a row (each of the m runs returning a cut other than C) is at most

$$\left(1 - \frac{2}{n^2}\right)^m .$$

Remember that if x is much less than 1, e^x is very close to $1 + x$. In particular, $(1 + x)^{1/x}$ is less than but very close to e. What we need here is the similar fact that for any positive number k, $(1 - \frac{1}{k})^k$ is smaller than $1/e$. Hence if our ϵ is $1/e$, we can set m to be $n^2/2$ and have an error probability less than ϵ. By setting m to $50n^2$, for example, our error probability goes down to e^{-100}, a safely small probability.