CPSC 536N: Randomized Algorithms 2041-15 Term 2

Lecture 1

Prof. Nick Harvey *University of British Columbia*

1 Course Overview

This is an introductory course in the design and analysis of **randomized algorithms**. I view this as a foundational topic in modern algorithm design, with many beautiful ideas and seemingly magical results. Many of the algorithms that we will discuss are used in various fields of computer science research. Some of the algorithms are even useful for practical software development, especially when processing extremely large data sets.

1.1 What are randomized algorithms?

It is important to understand that studying randomized algorithms is **not** the same as **average case analysis** of algorithms, in which one analyzes the performance of a deterministic algorithm when the inputs are drawn from a certain distribution. Instead, we will look at algorithms that can "flip random coins" and whose behavior depends on the outcomes of those coin flips. We will typically employ **worst-case analysis**, meaning that we analyze the performance and accuracy of the algorithm on its worst possible input.

Depending on the random coin flips, these algorithms might either run for a very long time, or output the wrong answer. At first, one might feel uncomfortable with algorithms that can exhibit such failures. However, typically it will be possibile to choose the failure probability to be extremely tiny. The following amusing quotation of Christos Papadimitriou justifies ignoring these small failure probabilities:

In some very real sense, computation is inherently randomized. It can be argued that the probability that a computer will be destroyed by a meteorite during any given microsecond of its operation is at least 2^{-100}.

This quotation makes a valid point, but it is presumably tongue-in-cheek. We will always view probability with the mindset of a mathematician rather than a statistician: probability distributions are purely mathematical objects, and they will not be used to model physical objects (like meteorites or data sets). Nevertheless, the algorithms that we will discuss are not just abstract mathematics; they can be usefully implemented on any physical computer with a decent pseudo-random generator.

1.2 Why?

Over the past thirty years, randomization has seen steadily increasing use in both academic computer science research and in the practice of software development. In modern algorithms research, it is more surprising to see a deterministic algorithm than a randomized algorithm. Twenty or thirty years ago, software developer managers may have banned randomization from their code; these days, randomization is widely viewed as a necessary technique for dealing with large data sets.

What do we gain by allowing small probabilities of failure?

- Randomized algorithms are very often *simpler* than the best known deterministic algorithm.

- Randomized algorithms are often more efficient (faster, or use less space) than the best known deterministic algorithm.
- There are some theoretical scenarios in which randomized algorithms can be *provably better* than the best possible deterministic algorithm. There are even scenarios in which no deterministic algorithm can do anything non-trivial — every interesting algorithm must necessarily be randomized.
- Sometimes ideas from randomization lead to interesting deterministic algorithms too.

1.3 Objectives

In this course, I intend to cover many of the key techniques, and to illustrate each technique with important applications. The two goals are:

- for you to understand these central techniques, so that you can understand them when you encounter them in papers, and perhaps use them in your own papers.
- to introduce you to the areas in which these techniques are useful, so that you will be comfortable if you encounter those areas again in the future, and possibly to entice you to work in these areas.

Some of the techniques that we discuss are useful in machine learning, data mining, databases, computer networks, etc.

1.4 Techniques

The main techniques we will encounter are:

1. Concentration (i.e., tail bounds)
2. Hashing
3. Dimensionality reduction
4. Streaming and sketching
5. Handling dependence

1.5 Strategies

Here is a list of some very high-level strategies for using randomization, as well as some applications (many of which we will see later) in which these strategies were successfully employed. Do not worry if these strategies are too vague to be understood at this point; hopefully seeing the application later will clarify what I mean.

1. *Avoiding pathological inputs.* Examples: QuickSort.
2. *Generating short "fingerprints".* Examples: Hashing, dimensionality reduction, streaming algorithms.
3. *Distilling a problem to its core.* Examples: computing minimum cuts, sparsification.
4. *Finding hay in a haystack.* Examples: polynomial identity testing, perfect matching.

5. *Coordinating in a distributed system.* Examples: consistent hashing, leader election.

6. *Rounding (converting continuous objects into discrete objects).* Examples: congestion minimization.

1.6 What won't we do?

Unfortunately there is too much beautiful work in this area and we can't cover everything. I could easily do dozens of lectures on this topic, but we would all run out of stamina before I run out of material. Some areas we definitely won't discuss include computational geometry, pseudorandomness, Markov Chain Monte Carlo, parallel computing, number theory, etc.

2 Example: Testing Equality

Suppose you download a large movie from an internet server. Before watching it, you'd like to check that your downloaded file has no errors, i.e., the file on your machine is identical to the file on the server. You would like to do this check without much additional communication, so sending the entire file back to the server is not a good solution. Ignoring cryptographic considerations, this is essentially the problem of computing a **checksum** and there are standard ways to do this, e.g., CRCs.

For concreteness, say that the file is n bits long, the server has the bitvector $a = (a_1, \ldots, a_n)$ and you have the bits $b = (b_1, \ldots, b_n)$. For standard checksums, the guarantee is:

- For "most" vectors a and b, the checksum will detect if they are not identical. So the guarantee is with respect to a supposed "distribution" on the vectors a and b.

However, as stated in Section 1, our mindset is **worst-case analysis**. We are interested in algorithms that work well even for the worst possible inputs, so this guarantee is too weak. Instead, we'd like a guarantee of this sort:

- For **every** vectors a and b, our algorithm will flip some random coins, and for most outcomes of the coins, will detect whether or not a and b are identical.

This guarantee differs in that any failures are no longer due to potentially bad inputs (a and b), but only due to potentially bad coin flips.

The main idea for our algorithm is to view the files as polynomials over a finite field $\mathbb{F}_p$. For those unfamiliar with fields, just think of $\mathbb{F}_p$ as being the set of integers $\{0, 1, \ldots, p-1\}$, and do all arithmetic modulo p. We need the following simple theorem about roots of polynomials.

Theorem 1 *Let $f(x)$ be a non-zero polynomial of degree at most d in a single variable x over any field. Then f has at most d roots (i.e., f evaluates to zero on at most d elements of the field).*

The proof follows from the facts that any polynomial can be uniquely factored into irreducibles, irreducibles of degree 1 have exactly one root, and irreducibles of degree greater than 1 have no roots.

The first step for our algorithm is to choose a prime number $p \in [2n, 4n]$. A fact known as Bertrand's Postulate implies that there always exists a prime in that range, so brute force search can find one in $\text{poly}(n)$ time.

Next, construct the polynomials $f_a(x) = \sum_{i=1}^{n} a_i x^i$ and $f_b(x) = \sum_{i=1}^{n} b_i x_i$. We will view these as polynomials over $\mathbb{F}_p$; in other words, when we evaluate the polynomials at some point x, compute the answer modulo p.

Define $g = f_a - f_b$. Note that $a = b$ if and only if g is the zero polynomial. On the other hand, if $a \neq b$, then g is a non-zero polynomial of degree at most n, so Theorem 1 implies it has at most n roots. So, if we pick an element $x \in \mathbb{F}$ uniformly at random then its probability of being a root of g is at most $n/|\mathbb{F}| = n/p \leq 1/2$.

This suggests the following algorithm. You and the server agree on the prime p. The server picks $x \in \mathbb{F}$ uniformly at random. It sends you x and $f_a(x)$, which are both elements of $\mathbb{F}$ and hence each can be represented with at most $\log(4n)$ bits. You compute $g(x) = f_a(x) - f_b(x)$. If $g(x) = 0$ the algorithm announces "a and b are equal". If $g(x) \neq 0$ the algorithm announces "a and b are not equal".

If $a = b$ then $g = 0$ so the algorithm never makes a mistake. If $a \neq b$ then this algorithm makes an error only x is a root of g, which happens with probability at most $1/2$.

2.1 Remarks

Two points are worth noting:

- This analysis is valid **for all** a and b and the only randomness used in our probabilistic analysis comes from the random choice of x.
- The algorithm only makes a mistake if $a \neq b$ and never makes a mistake if $a = b$. We say that such an algorithm has **one-sided error**.

The number of bits exchanged between you and the server is only $O(\log n)$. Furthermore, it is known that this is optimal: every randomized algorithm for this problem that succeds with constant probability requires $\Omega(\log n)$ bits of communication.

Recall our lists of techniques and strategies from Section 1. This example uses technique 2 (avoiding zeros of polynomials) and strategy 3 (finding hay in a haystack), where the field $\mathbb{F}$ is the haystack and the non-roots of g are the hay.

2.2 Decreasing the Failure Probability

Note that we only achieved a failure probability of $1/2$. As mentioned earlier, it is usually easy to decrease the failure probability down to any desired level. For example, if we pick the field size to be $p \in [kn, 2kn]$, then the failure probability is $\leq 1/k$ and the number of bits exchanged is $O(\log(p)) = O(\log(kn)) = O(\log(k) + \log(n))$.

Another possibility to reduce the failure probability is to keep $p \in [2n, 4n]$ but instead to evaluate the polynomials at multiple independent points. The server can pick $x_1, \ldots, x_\ell \in \mathbb{F}$ mutually independently. The algorithm declares the files to be equal if $g(x_i) = 0$ for all $i = 1, \ldots, \ell$. If $a \neq b$, the probability that this happens is

$$\Pr\left[\bigcap_{i=1}^{\ell} g(x_i) = 0\right] \;=\; \prod_{i=1}^{\ell} \Pr\left[\, g(x_i) = 0 \,\right] \;\leq\; 1/2^{\ell}.$$

Taking $\ell = \lg(k)$, the failure probability is again at most $1/k$, but the number of bits exchanged is $O(\ell \cdot \log(p)) = O(\log(k) \cdot \log(n))$, which is a bit worse than the previous scheme.