**CS 388R: Randomized Algorithms, Fall 2021** September 30, 2021

# Lecture 11: Fingerprinting

*Prof. Eric Price* *Scribe: Nathaniel Sauerberg and Giannis Daras*

**NOTE:** THESE NOTES HAVE NOT BEEN EDITED OR CHECKED FOR CORRECTNESS

## 1 Overview

In the last lecture we talked about Routing. In this lecture we are going to talk about Fingerprinting.

Fingerprinting in the context of randomized algorithms is, roughly speaking, a technique to easily compare two objects by comparing their hashes. When designing a fingerprinting algorithm the task is to design a hash family $\mathcal{H}$ such that if we select a function $h \in \mathcal{H}$, then for $x \neq y$, it is also true that $h(x) \neq h(y)$ with some decent probability. As we will see in this lecture, this idea has very useful applications such as verifying the result of a matrix multiplication, testing if two polynomials have all their coefficients equal and finding whether a string is a substring of a larger string.

## 2 Matrix Equality Testing (Freivalds' Algorithm [4])

Suppose that we are given the following decision problem: "*For some matrices* $A, B, C \in \mathbb{R}^{n\times n}$ *verify whether* $A \cdot B = C$.".

One way to solve this problem would be to analytically compute the product of $A, B$ and then compare entry-wise $A \cdot B, C$. A naive implementation of this idea would give time complexity $O(n^3)$. The best known algorithm for matrix multiplication gives complexity $O(n^{\log_2 7})$ [1].

We will now show that there is a Randomized Algorithm, that can check if $A \cdot B = C$ in $O(kn^2)$ time with error probability $\leq \frac{1}{2^k}$. To do so, we first prove the following lemma.

**Lemma 1** (Upper bound on the probability of a random vector to be in the null-space of a matrix.)**.** *Let* $W \in \mathbb{R}^{n\times n}$. *Let also* $r$ *be a random vector, drawn uniformly from* $\{0,1\}^n$. *Then,*

$$\mathbb{P}[W \cdot r = 0 | W \neq 0] \leq \frac{1}{2}. \tag{1}$$

*Proof.* Let $w_1^T, ..., w_n^T$ denote the rows of $W$. Since $W \neq 0$, there is a row of $W$ that is non-zero. Without loss of generality, assume that $w_1^T$ is the non-zero row and $w_{1j}$ a non-zero element of that row.

$$\mathbb{P}[W \cdot r = 0 | W \neq 0] \leq \mathbb{P}[w_1^T \cdot r = 0] \tag{2}$$

$$= \mathbb{P}\left[\left(\sum_{i \neq j} w_{1i} r_i\right) + w_{1j} = 0 | r_j = 1\right] \cdot \mathbb{P}[r_j = 1] + \mathbb{P}\left[\sum_{i \neq j} w_{1i} r_i = 0 | r_j = 0\right] \cdot \mathbb{P}[r_j = 0] \tag{3}$$

$$= \mathbb{P}\left[\left(\sum_{i \neq j} w_{1i} r_i\right) + w_{1j} = 0\right] \cdot \mathbb{P}[r_j = 1] + \mathbb{P}\left[\sum_{i \neq j} w_{1i} r_i = 0\right] \cdot \mathbb{P}[r_j = 0] \tag{4}$$

$$= \frac{1}{2}\left(\mathbb{P}\left[\left(\sum_{i \neq j} w_{1i} r_i\right) + w_{1j} = 0\right] + \mathbb{P}\left[\sum_{i \neq j} w_{1i} r_i = 0\right]\right). \tag{5}$$

Since we selected $w_{1j}$ to be a non-zero element of that row and we perform everything modulo 2, the two probabilities sum to 1, i.e.

$$\mathbb{P}\left[\left(\sum_{i \neq j} w_{1i} r_i\right) + w_{1j} = 0\right] + \mathbb{P}\left[\sum_{i \neq j} w_{1i} r_i = 0\right] = 1. \tag{6}$$

Hence, we have that:

$$\mathbb{P}[W \cdot r = 0 | W \neq 0] \leq \frac{1}{2}. \tag{7}$$

□

**Algorithm** We are going to use this lemma to develop our algorithm. Specifically, we want to see if $A \cdot B = C$. Instead, we are going to check if

$$(A \cdot B)r = Cr \iff \tag{8}$$

$$(A \cdot B - C)r = 0 \iff \tag{9}$$

$$Wr = 0, \tag{10}$$

for some $r$ sampled uniformly from $\{0,1\}^n$ and $W = A \cdot B - C$ and we are going to accept that $A \cdot B = C$ if $W \cdot r = 0$ and reject it otherwise.

**Error probability analysis** Now, observe that the algorithm never has false negatives. In other words, for any $r \in \{0,1\}^n$, if $A \cdot B = C$ then also $W \cdot r = 0$ trivially. Now, in case $A \cdot B \neq C$, then from the Lemma, we have that $\mathbb{P}[W \cdot r = 0 | W \neq 0] \leq \frac{1}{2}$. We are going to repeat this process $k$ times and we will output 1 only if all the times $W \cdot r = 0$. We denote with $r^1, ..., r^k$ the sampled vectors $r$.

Now, the probability of error in case $A \cdot B \neq C$ is:

$$\mathbb{P}[\text{error}] = \mathbb{P}[W \cdot r^1 = 0 \wedge W \cdot r^2 = 0 ... \wedge ... W \cdot r^k = 0 | W \neq 0] \leq \frac{1}{2^k}. \tag{11}$$

**Time Analysis** We can compute $A \cdot B \cdot r$ as $A(B \cdot r)$ which takes $O(n^2)$ time. Similarly, we can compute $C \cdot r$ in $O(n^2)$ time. If we repeat the process $k$ times to reduce the error probability to $\frac{1}{2^k}$, then the running time becomes $O(kn^2)$.

# 3 Polynomial Identity Testing

The next application we examine is Polynomial Identity Testing. We consider the following question: "Given degree $d$ univariate polynomials $P(x), Q(x)$ and a degree $2d$ univariate polynomial $R(x)$, is it true that $P(x)Q(x) \stackrel{?}{=} R(x)$?". A more general version of this problem is to examine if a given univariate polynomial $P(x)$ is everywhere equal to 0. We will focus on this general version of the problem.

This simple question can be extremely hard to answer since the expressions $P(x)$ can be very hard to re-write in the standard form. Instead, we are going to exploit a useful fact about polynomials.

**Theorem 2** (Fundamental Theorem of Algebra)**.** *Any univariate polynomial of degree $d$, has at most $d$ unique roots.*

**Attempt 0:** Deterministically, we could compute the value of a polynomial $P$ at any set of $d+1$ points. If $P$ is the zero polynomial, it will of course be 0 at every chosen point, and if $P$ is not the zero polynomial, it has at most $d$ roots so will be nonzero at at least one of the chosen points. Can we speed this up with randomization?

**Attempt 1:** We are going to pick $c_1$ random elements from a set S with $|S| = c_2 \cdot d$, let's say $S = [c_2 \cdot d]$.

If all of them evaluate $P(x)$ to 0, then we are going to output that the polynomial is the zero polynomial, otherwise we are going to output that it is not. Observe that there is zero probability that the algorithm will output that the polynomial is not the zero polynomial if it is (based on Theorem 2). In other words, the algorithm does not have any false negatives.

Now we will analyze the probability of picking a root for a single query. The worst case is when the set $S$ that we are picking our queries from contains all the roots. In that case, the probability of choosing one root is at most $\frac{1}{c_2}$. By repeating this procedure $c_1$ times we can make the probability of failure of our randomized algorithm arbitrarily small. For failure probability $1 - \delta$, we need to do $\log_{c_2} 1/\delta$ queries.

The problem with this attempt is that $P(d)$ can be $\approx (c_2 d)^d$, i.e. we need to store $d \log c_2 d$ bits to represent this number.

**Attempt 2:** We use the aforementioned idea, but this time, we are operating on a Finite Field $\mathbb{F}_p$ for a prime $p \geq 2d$. For such fields, there is a similar Theorem to Theorem 2 and the analysis is very similar.

**Extending to multivariate polynomials** We can extend this idea for multivariate polynomials. To do so, we are going to use the following lemma.

**Lemma 3** (Schwartz–Zippel)**.** *Let $P$ be a non-zero polynomial of $n$ variables and total degree $d \geq 0$ over a field $\mathbb{F}_p$. Let $S$ be a finite subset of $\mathbb{F}_p$ and let $r_1, r_2, ..., r_n$ be selected at random independently and uniformly from $S$. Then*

$$\mathbb{P}[P(r_1, r_2, \ldots, r_n) = 0] \leq \frac{d}{|S|}. \tag{12}$$

**Proposition 4.** *Let $x = (x_1, ..., x_n)$ uniformly drawn from $\mathbb{F}_p^n$ and $P$ be an $n$ variable polynomial of degree at most $d$ over the field $\mathbb{F}_p^n$ with $p > 2d$. Then,*

$$\mathbb{P}[P(x) = 0] \leq \frac{d}{p}. \tag{13}$$

We will now see how this Proposition can be useful to find quickly whether a string is a substring of a larger string. To do so, we will introduce the String Matching problem.

# 4 String Matching

Suppose we have a small string, and we want to check whether its a substring of a much larger string. For example, we might be doing a `CTRL-F` search in a large document. We'll focus on the decision version of the problem.

**Problem:**

Given: an $n$-bit string $a$ and an $m$-bit string $b$, where $n \geq m$

Output: whether $b$ is a substring of $a$

The naive deterministic algorithm is to compare every length $m$ substring of $a$ to $b$, which takes $O(nm)$. Can we do better with fingerprinting? One way to speed this up is to compute hashes of the substrings and compare the hashes rather than the full strings. Let's define $s_t$ to be the $t$-th substring of length $m$ of $a$, so $s_t = a_t a_{t+1} \cdots a_{t+m-1}$.

**Algorithm:**

- compute $h(b)$
- compute $h(s_1), h(s_2), \ldots h(s_{n-m+1})$
- if $h(b)$ matches the hash of a substring, either return "yes" (and be correct with good probability) or compare the full strings (for a Las Vegas style algorithm).

The key question now becomes how to define the hash functions.

## 4.1 Rabin-Carp Algorithm [2]

We will now show how we can use the Polynomial Identity Testing analysis and Rabin-Carp Algorithm to come up with an elegant solution to the String Matching problem.

We define a polynomial corresponding to any length $m$ substring $t = (t_1, ..., t_m)$ by $P_t(x) = \sum_{i=1}^{m} t_i x^{m-i}$. Then, following the previous section, we can compare any two strings $t$ and $r$ by comparing $P_t(x)$ and $P_r(x)$ mod some prime $q$ and for a randomly chosen $x \in [q]$.

The overall algorithm is to pick a prime $q$ and a random $x \in [q]$. Then, we compare $P_b(x)$ to $P_{s_t}(x)$ for every length $m$ substring $s_t$ of $a$. We output that there is substring match if $(P_b(x) - P_{s_t}(x) \mod q) = 0$ for some $s_t$, and that there is not a substring match otherwise.

Each pairwise comparison succeeds with with probability at least $1 - m/q$, where the probability is over the random choice of $x$. Therefore, if we take $q$ to be a prime on the order of $mn^2$, we can get success with high probability even after union bounding over the $O(n)$ substrings of $a$. Note that these comparisons are not independent since each one depends on the same random choice of $x$, but that's ok since we are union bounding.

One important detail remains: how to actually compute the $P_s(x)$. We've defined $P_s$ so that we can quickly compute each next value from the previous one, ie $P_{s_i}(x)$ from $P_{s_{i-1}}(x)$. For example:

$$P_{s_2}(x) = \sum_{i=2}^{m+1} a_i x^{m-i+1} = a_{m+1} + x P_{s_1}(x) - a_1 x^{m-1}$$

Recall that each $P_{s_i}(x)$ is just a number mod $q$, not a polynomial, since we're evaluating them for some specific $x$. Therefore, we can compute each $P_{s_i}(x)$ from $P_{s_{i-1}}(x)$ in constant time. Since these are numbers mod $q$, storing them takes only $\log q$ bits, which is fine since $q$ is poly$(n, m)$.

### 4.2 An Alternative Algorithm

The Rabin-Carp algorithm picks a random value $x$ at which to compare the polynomials, while the prime $q$ just allows the computations to be faster and doesn't need to be random. Our alternative approach flips this.

We can view $b$ and the substrings $s$ of $a$ as binary numbers with values up to $2^m - 1$. Then, we can pick a random prime $p$ and compare $b$ and $s$ mod $p$. We get the same 1-sided error as before: if $b = s$, then they they are equal mod $p$, but if $b \neq s$, there is still some chance they're equal mod $p$. If we let $c = b - s$, we're trying to check whether $c = 0 \mod p$, so the relevant question is how many primes $p$ divide $c$.

If we write $c$ as a product of primes, each prime must be at least 2, so the product of $k$ primes is at least $2^k$. Therefore, since $c < 2^m$, no more that $m$ primes can divide $c$[1]. Hence, if we draw $p$ from a set of at least $mn^2$ primes, we'll succeed with probability $1 - 1/n^2$ on each substring, and then union bounding over the $n$ substrings we still succeed with high probability. To get a set of $mn^2$ primes, we need to go up to primes of size $\Theta(mn^2 \log(mn^2))$ due to the density of the primes in the integers being about $(\log n)/n$, as discussed before.

Note that we can do the same type of rolling computation of the $s$ values as before. In addition, this algorithm assumes we can generate random primes of around a certain value. We will now show how it is possible to do that.

[1] This is a loose bound, and tighter analysis might be able to save a log factor.

# 5 Primality Checking

The most straightforward way is to generate a random integer around the desired value, check if it's prime, and if not repeat until you find a prime. Note that the density of primes in the integers is $\Theta(\frac{\log n}{n})$. Therefore, to generate a prime around size $L$, we'll need to check roughly $\log L$ integers in expectation before we find a prime, which isn't too high of a cost.

Our primality testing algorithms will rely on some number theory theorems that we won't prove.

## 5.1 Fermat Primality Test

**Theorem 5.** *(Fermat's Little Theorem) If $p$ is prime, then $a^{p-1} = 1 \mod p$ for any $a \neq 0 \mod p$.*

This is the first theorem. Over the integers, 1 and $-1$ are the only square roots of 1. This theorem basically says that this also holds if we are working over the integers mod $p$.

If we have a number $n$ and we'd like to check if $n$ is prime, the most obvious thing to do is to check if Fermat's little theorem holds. If it fails, then we know $n$ is not prime.

**Fermat Primality Test:**

- Given an integer $n$
- Pick $a$ uniformly at random from $\{1, \ldots, n-1\}$
- Check if $a^{n-1} = 1 \mod n$
- If yes, return "(probably) prime"
- If no, return "composite"

Note that our algorithm always correctly identifies primes as prime, but sometimes it will also think composite numbers are prime. The algorithm would work well if composite numbers are rarely thought to be prime. In other words, we'd like to claim that, for any composite number, many choices of $a$ will have $a^{n-1} \neq 1 \mod n$ and so will lead to $n$ being identified as composite.

Unfortunately, there exist composite numbers called Carmichael numbers that will cause our test to fail for any relative prime base $a$. In other words, the test succeeds only if we pick an $a$ that divides $n$, so for Carmichael numbers the Fermat test does so no better than randomly looking for divisors of $n$.

How do we get around this problem? We can add another test, with the hope that composite numbers that always pass the Fermat primality test will usually fail our other primality test.

## 5.2 Miller-Rabin Primality Test [3] [5]

Since we're trying to check if $n$ is prime, we can assume that $n$ is odd (if it was even, we'd just need to check that it wasn't 2). Then, let's factor out as many powers of 2 from $n-1$ as we can,

so write $n-1 = 2^q m$ for some odd $m$. Applying Fermat's little theorem to this form of $n-1$ tells us if $n$ is prime, then for any $a \neq 0 \mod n$, $a^{n-1} = a^{2^q m} = 1 \mod n$.

So if $n$ passes Fermat's test, then we've picked some $a$ where $a^{2^q m} = 1 \mod n$. We'd like to have another test that composite numbers passing Fermat's test will fail. This is where our second theorem comes in.

**Theorem 6.** *If $p$ is a prime and $a^2 = 1 \mod p$, then $a$ is either 1 or $-1$ mod $p$.*

Following the theorem, if $\sqrt{a^{2^q m}} = a^{2^{q-1} m}$ is not 1 or $-1$, we know for sure that $n$ is not prime. Additionally, if $a^{2^{q-1} m} = 1$, then we can apply the theorem again and check the value of $a^{2^{q-2} m}$. We can continue this process until we get the first non-1 value. If the value is not $-1$, we know for certain $n$ is composite, and if it is $-1$, then we're still uncertain and will guess that $n$ is prime.

**Miller-Rabin Primality Test:**

- Given an integer $n$
- Pick $a$ uniformly at random from $\{1, \ldots, n-1\}$
- Write $n - 1 = 2^q m$ for odd $m$
- If $a^{2^q m} \neq 1$, return "composite", else continue:
- Let $i = 0$
  - If $a^{2^{q-i} m} \mod n = 1$, increment $i$
  - Else, if $a^{2^{q-i} m} \mod n = -1$, return "prime"
  - Else $a^{2^{q-i} m} \mod n \notin \{1, -1\}$ so return "composite"

This algorithm has the same one-sided error as before: when we return composite, we are sure $n$ is composite, but when we return prime, $n$ are not entirely sure. Fortunately, it turns out that when $n$ is composite, it usually can't fool our algorithm. That is, for any composite $n$, the first non-1 value of in the sequence $a^{2^{q-i} m} \mod n$ will also not be $-1$ with probability at least 3/4 (where the probability is over the random choice of $a$).

Therefore, this algorithm has only constant failure probability and can be repeated with other choices of $a$ to get arbitrarily low failure probability.

**Historical Note:** This algorithm was first given by Miller in 1976 [3] in a version whose correctness relied on the Reimann Hypothesis. In 1980, Miller [5] improved it to be correct unconditionally.

# 6 A Probability Puzzle

## 6.1 The puzzle

We finished class by working on the following probability puzzle about sequences of coin flips.

Suppose we flip fair coins until the number of heads minus the number of tails is either $-10$ or 100. What is the probability of ending at $-10$?

Notationally, we can let $X_i \in \{\pm 1\}$ be the outcomes of the independent, random coin flips. We'll have $Y_i$ be the sum of the first $i$ flips, so $Y_0 = 0$ and $Y_i = Y_{i-1} + X_i$. The question is then the probability that $Y_i$ ever reaches $-10$.

Since we stop flipping coins when $Y_i$ reaches $-10$ or 100, we'll define the $Y$s to stop changing at that point. That is, we'll define $Y_i = \begin{cases} Y_{i-1} & \text{if } Y_i \in \{-10, 100\} \\ Y_{i-1} + X_i & \text{otherwise} \end{cases}$.

### 6.2 The Expectation Solution

They key observation is that the expectation of $Y_i$ is always 0 (for any $i$) since it is just the sum of coin flips that each have expected value 0. Also, with probability 1, $Y_i$ will eventually reach one of the endpoints and stop (recall that the standard deviation of $Y_n$ is $\Theta(\sqrt{n})$. Therefore, as $n$ goes to infinity, the expectation of $Y_n$ becomes dependent only on the two endpoint outcomes:

$$\lim_{n\to\infty} \mathbb{E}[Y_n] = 100\,\mathbb{P}[Y_n = 100] + -10\,\mathbb{P}[Y_n = -10]$$

Then, letting $p$ be the probability that $Y_i$ reaches $-10$, we can write $\mathbb{E}[Y_n] = 0 = 100(1-p) + -10p$ and then solve to get $p = 10/11$.

One potentially surprising implication of the result is that **the probability of reaching an endpoint depends only on the ratio of the endpoints.** For example, the probability of reaching $-100$ before 1000 and the probability reaching $-1$ before 10 would both also be $10/11$.

### 6.3 The Recurrence Solution

We can get the same solution by solving a recurrence. Let $f(i)$ be the probability that the sequences reaches $-10$, starting from a time $t$ where $Y_t = i$. Then we know that $f(-10) = 1$ and $f(100) = 0$. For other $i$, we can just condition on the outcome of the second coin flip to write $f(i) = 1/2 f(i + 1) + 1/2 f(i - 1)$. It turns out that $f(0) = 10/11$.

## References


[1] Josh Alman and Virginia Vassilevska Williams. A refined laser method and faster matrix multiplication. *Proceedings of the 2021 ACM-SIAM Symposium on Discrete Algorithms (SODA)*, page 522–539, Jan 2021.

[2] Richard M. Karp and Michael O. Rabin. Efficient randomized pattern-matching algorithms. *IBM Journal of Research and Development*, 31(2):249–260, March 1987.

[3] Gary L. Miller. Riemann's hypothesis and tests for primality. In *Proceedings of the Seventh Annual ACM Symposium on Theory of Computing*, STOC '75, page 234–239, New York, NY, USA, 1975. Association for Computing Machinery.

[4] Rajeev Motwani and Prabhakar Raghavan. *Randomized algorithms*. Cambridge university press, 1995.

[5] Michael O Rabin. Probabilistic algorithm for testing primality. *Journal of Number Theory*, 12(1):128–138, 1980.