2

The Power of Randomness: Fingerprinting and Freivalds' Algorithm

2.1 Reed-Solomon Fingerprinting

The proof systems covered in this survey derive much of their power and efficiency from their use of randomness. Before we discuss the details of such proof systems, let us first develop an appreciation for how randomness can be exploited to dramatically improve the efficiency of certain algorithms. Accordingly, in this section, there are no untrusted provers or computationally weak verifiers. Rather, we consider two parties, Alice and Bob, who trust each other and want to cooperate to jointly compute a certain function of their inputs.

2.1.1 The Setting

Alice and Bob live across the country from each other. They each hold a very large file, each consisting of n characters (for concreteness, suppose that these are ASCII characters, so there are $m = 128$ possible characters). Let us denote Alice's file as the sequence of characters $(a_1, \ldots, a_n)$, and Bob's as $(b_1, \ldots, b_n)$. Their goal is to determine whether their files are *equal*, i.e., whether $a_i = b_i$ for all $i = 1, \ldots, n$. Since the

files are large, they would like to minimize *communication*, i.e., Alice would like to send as little information about her file to Bob as possible.

A trivial solution to this problem is for Alice to send her entire file to Bob, and Bob can check whether $a_i = b_i$ for all $i = 1, \ldots, n$. But this requires Alice to send all n characters to Bob, which is prohibitive if n is very large. It turns out that no *deterministic* procedure can send less information than this trivial solution.[1]

However, we will see that if Alice and Bob are allowed to execute a *randomized* procedure that might output the wrong answer with some tiny probability, say at most 0.0001, then they can get away with a much smaller amount of communication.

2.1.2 The Communication Protocol

The High-Level Idea. The rough idea is that Alice is going to pick a hash function h at random from a (small) family of hash functions $\mathcal{H}$. We will think of $h(x)$ as a very short "fingerprint" of x. By fingerprint, we mean that $h(x)$ is a "nearly unique identifier" for x, in the sense that for any $y \neq x$, the fingerprints of x and y differ with high probability over the random choice of h, i.e.,

$$\text{for all } x \neq y, \Pr_{h \in \mathcal{H}}[h(x) = h(y)] \leq 0.0001.$$

Rather than sending a to Bob in full, Alice sends h and $h(a)$ to Bob. Bob checks whether $h(a) = h(b)$. If $h(a) \neq h(b)$, then Bob *knows* that $a \neq b$, while if $h(a) = h(b)$, then Bob can be very confident (but not 100% sure) that $a = b$.

The Details. To make the above outline concrete, fix a prime number $p \geq \max\{m, n^2\}$, and let $\mathbb{F}_p$ denote the set of integers modulo p. For the remainder of this section, we assume that all arithmetic is done *modulo* p without further mention.[2] This means that all numbers are

[1] The interested reader is directed to [177, Example 1.21] for a proof of this fact, based on the so-called *fooling set method* in communication complexity.

[2] The reason to perform all arithmetic modulo p rather than over the integers is to ensure that all numbers arising in the protocol can always be represented using just $\log_2(p) = O(\log(n) + \log(m))$ bits. If arithmetic were performed over the integers rather than modulo p, then the protocol covered in this section would require Alice

replaced with their remainder when divided by p. So, for example, if $p = 17$, then $(2 \cdot 3^2 + 4) \pmod{17} = 22 \pmod{17} = 5$.

The reason p must be chosen larger than n^2 is that the error probability of the protocol we are about to describe is less than n/p, and we wish this quantity to be bounded above by $1/n$ (larger choices of p will result in yet smaller error probabilities). The reason p must be chosen larger than the number of possible characters m is that the protocol will interpret Alice and Bob's inputs as vectors in $\mathbb{F}_p^n$ and check whether these vectors are equal. This means that we need a way to associate each possible character in Alice and Bob's inputs with a different element of $\mathbb{F}_p$, which is possible if and only if p is greater than or equal to m.

For each $r \in \mathbb{F}_p$, define $h_r(a_1, \ldots, a_n) = \sum_{i=1}^{n} a_i \cdot r^{i-1}$. The family $\mathcal{H}$ of hash functions we will consider is

$$\mathcal{H} = \{h_r : r \in \mathbb{F}_p\}. \tag{2.1}$$

Intuitively, each hash function h_r interprets its input $(a_1, \ldots, a_n)$ as the coefficients of a degree $n-1$ polynomial, and outputs the polynomial evaluated at r. That is, in our communication protocol, Alice picks a random element r from $\mathbb{F}_p$, computes $v = h_r(a)$, and sends v and r to Bob. Bob outputs EQUAL if $v = h_r(b)$, and outputs NOT-EQUAL otherwise.

2.1.3 The Analysis

We now prove that this protocol outputs the correct answer with very high probability. In particular:

- If $a_i = b_i$ for all $i = 1, \ldots, n$, then Bob outputs EQUAL for every possible choice of r.
- If there is even one i such that $a_i \neq b_i$, then Bob outputs NOT-EQUAL with probability at least $1 - (n-1)/p$, which is at least $1 - 1/n$ by choice of $p \geq n^2$.

---

to send to Bob an integer that may have magnitude more than 2^n, which would require more than n bits to represent. This is nearly as expensive as having Alice send her entire input to Bob.

The first property is easy to see: if $a = b$, then obviously $h_r(a) = h_r(b)$ for every possible choice of r. The second property relies on the following crucial fact, whose validity we justify later in Section 2.1.6.

Fact 2.1. For any two distinct (i.e., unequal) polynomials p_a, p_b of degree at most n with coefficients in $\mathbb{F}_p$, $p_a(x) = p_b(x)$ for at most n values of x in $\mathbb{F}_p$.

Let $p_a(x) = \sum_{i=1}^{n} a_i \cdot x^{i-1}$ and similarly $p_b(x) = \sum_{i=1}^{n} b_i \cdot x^{i-1}$. Observe that both p_a and p_b are polynomials in x of degree at most $n-1$. The value v that Alice sends to Bob in the communication protocol is precisely $p_a(r)$, and Bob compares this value to $p_b(r)$.

By Fact 2.1, if there is even one i such that $a_i \neq b_i$, then there are at most $n-1$ values of r such that $p_a(r) = p_b(r)$. Since r is chosen at random from $\mathbb{F}_p$, the probability that Alice picks such an r is thus at most $(n-1)/p$. Hence, Bob outputs NOT-EQUAL with probability at least $1 - (n-1)/p$ (where the probability is over the random choice of r).

2.1.4 Cost of the Protocol

Alice sends only two elements of $\mathbb{F}_p$ to Bob in the above protocol, namely v and r. In terms of bits, this is $O(\log n)$ bits assuming $p \leq n^c$ for some constant c. This is an *exponential improvement* over the $n \cdot \log m$ bits sent in the deterministic protocol (all logarithms in this monograph are to base 2 unless the base is explicitly specified otherwise). This is an impressive demonstration of the power of randomness.[3]

[3]Readers familiar with cryptographic hash functions such as SHA-3 may be in the habit of thinking of such a hash function as a fixed, deterministic function, and hence perplexed by the characterization of our protocol as randomized (as Alice just sends the hash function h and the evaluation $h(a)$ to Bob, where a is Alice's input vector). To this, we offer two clarifications. First, the communication protocol in this section actually does not require a cryptographic hash function. Rather, it uses a function chosen at random from the hash family given in Equation (2.1), which is in fact far simpler than any cryptographic hash family, e.g., it is not collision-resistant or one-way. Second, cryptographic hash functions such as SHA-3 really should be modeled as having been sampled at random from some large family. Otherwise, properties such as collision-resistance would be broken against non-uniform adversaries (i.e., adversaries permitted unlimited pre-processing). For example, collision-resistance

2.1.5 Discussion

We refer to the above protocol as Reed-Solomon fingerprinting because $p_a(r)$ is actually a random entry in an *error-corrected encoding* of the vector $(a_1, \ldots, a_n)$. The encoding is called the Reed-Solomon encoding. Several other fingerprinting methods are known. Indeed, all that we really require of the hash family $\mathcal{H}$ used in the protocol above is that for any $x \neq y$, $\Pr_{h \in \mathcal{H}}[h(x) = h(y)]$ is small. Many hash families are known to satisfy this property,[4] but Reed-Solomon fingerprinting will prove particularly relevant in our study of probabilistic proof systems, owing to its algebraic structure.

A few sentences on finite fields. A *field* is any set equipped with addition, subtraction, multiplication, and division operations, and such that these operations behave roughly the same as they do over the rational numbers.[5] So, for example, the set of real numbers is a field, because for any two real numbers c and d, it holds that $c+d$, $c-d$, $c \cdot d$, and (assuming $d \neq 0$) c/d are themselves all real numbers. The same holds for the set of complex numbers, and the set of rational numbers. In contrast, the set of integers is *not* a field, since dividing two integers does not necessarily yield another integer.

For any prime number p, $\mathbb{F}_p$ is also a field (a *finite* one). Here, the field operations are simply addition, subtraction, multiplication, and

of any fixed deterministic function h is broken by simply "hard-coding" into the adversary two distinct inputs x, x' such that $h(x) = h(x')$. This pre-processing attack does not work if h is chosen at random from a large family of functions, and the pre-processing has to occur prior to the random selection of h.

[4] Such hash families are called *universal*. The excellent Wikipedia article on universal hashing contains many constructions https://en.wikipedia.org/wiki/Universal_hashing.

[5] In more detail, the addition and multiplication operations in any field must be associative and commutative. They must also satisfy the distributive law, i.e., $a \cdot (b + c) = a \cdot b + a \cdot c$. Moreover, there must be two special elements in the field, denoted 0 and 1, that are additive and multiplicative identity elements, i.e., for all field elements a, it must hold that $a + 0 = a$ and $a \cdot 1 = a$. Every field element a must have an additive inverse, i.e., a field element $-a$ such that $a + (-a) = 0$. This ensures that subtraction can be defined in terms of addition of an additive inverse, i.e., $b - a$ is defined as $b + (-a)$. And every *nonzero* field element a must have a multiplicative inverse a^{-1} such that $a \cdot a^{-1} = 1$. This ensures that division by a nonzero field element a can be defined as multiplication by a^{-1}.

division modulo p. What we mean by division modulo p requires some explanation: for every $a \in \mathbb{F}_p \setminus \{0\}$, there is a unique element $a^{-1} \in \mathbb{F}_p$ such that $a \cdot a^{-1} = 1$. For example, if $p = 5$ and $a = 3$, then $a^{-1} = 2$, since $3 \cdot 2 \pmod 5 = 6 \pmod 5 = 1$. Division by a in $\mathbb{F}_p$ refers to multiplication by a^{-1}. So if $p = 5$, then in $\mathbb{F}_p$, $4/3 = 4 \cdot 3^{-1} = 4 \cdot 2 = 3$.

Much later in this monograph (e.g., Section 15.1), we will exploit the fact that for any prime *power* (i.e., p^k for some prime p and positive integer k), there is a unique finite field of size p^k, denoted $\mathbb{F}_{p^k}$.[6]

2.1.6 Establishing Fact 2.1

Fact 2.1 is implied by (in fact, equivalent to) the following fact.

Fact 2.2. Any nonzero polynomial of degree at most n over any field has at most n roots.

A simple proof of Fact 2.2 can be found online at [208]. To see that Fact 2.2 implies Fact 2.1, observe that if p_a and p_b are *distinct* polynomials of degree at most n, and $p_a(x) = p_b(x)$ for more than n values of $x \in \mathbb{F}_p$, then $p_a - p_b$ is a *nonzero* polynomial of degree at most n with more than n roots.

2.2 Freivalds' Algorithm

In this section, we see our first example of an efficient probabilistic proof system.

2.2.1 The Setting

Suppose we are given as input two $n \times n$ matrices A and B over $\mathbb{F}_p$, where $p > n^2$ is a prime number. Our goal is to compute the product matrix $A \cdot B$. Asymptotically, the fastest known algorithm for accomplishing this task is very complicated, and runs in time roughly $O(n^{2.37286})$ [5], [178]. Moreover, the algorithm is not practical. But for the purposes of this monograph, the relevant question is not how fast can one multiply

[6] More precisely, all finite fields of size p^k are isomorphic, roughly meaning they have the exact same structure, though they may not assign names to elements in the same manner.

two matrices—it's how efficiently can one *verify* that two matrices were multiplied correctly. In particular, can verifying the output of a matrix multiplication problem be done faster than the fastest known algorithm for actually multiplying the matrices? The answer, given by Freivalds in 1977 [115], is yes.

Formally, suppose someone hands us a matrix C, and we want to check whether or not $C = A \cdot B$. Here is a very simple randomized algorithm that will let us perform this check in $O(n^2)$ time.[7] This is only a constant factor more time than what is required to simply read the matrices A, B, and C.

2.2.2 The Algorithm

First, choose a random $r \in \mathbb{F}_p$, and let $x = (1, r, r^2, \ldots, r^{n-1})$. Then compute $y = Cx$ and $z = A \cdot Bx$, outputting YES if $y = z$ and NO otherwise.

2.2.3 Runtime

We claim that the entire algorithm runs in time $O(n^2)$. It is easy to see that generating the vector $x = (1, r, r^2, \ldots, r^{n-1})$ can be done with $O(n)$ total multiplication operations (r^2 can be computed as $r \cdot r$, then r^3 can be computed as $r \cdot r^2$, then r^4 as $r \cdot r^3$, and so on). Since multiplying an $n \times n$ matrix by an n-dimensional vector can be done in $O(n^2)$ time, the remainder of the algorithm runs in $O(n^2)$ time: computing y involves multiplying C by the vector x, and computing $A \cdot Bx$ involves multiplying B by x to get a vector $w = Bx$, and then multiplying A by w to compute $A \cdot Bx$.

2.2.4 Completeness and Soundness Analysis

Let $D = A \cdot B$, so that our goal is to determine whether the *claimed* product matrix C actually equals the *true* product matrix D. Letting $[n]$ denote the set $\{1, 2, \ldots, n\}$, we claim that the above algorithm satisfies the following two conditions:

[7] Throughout this monograph, we assume that addition and multiplication operations in finite fields take constant time.

- If $C = D$, then the algorithm outputs YES for every possible choice of r.
- If there is even one $(i, j) \in [n] \times [n]$ such that $C_{i,j} \neq D_{i,j}$, then Bob outputs NO with probability at least $1 - (n-1)/p$.

The first property is easy to see: if $C = D$, then clearly $Cx = Dx$ for all vectors x, so the algorithm will output YES for every choice of r. To see that the second property holds, suppose that $C \neq D$, and let C_i and D_i denote the ith row of C and D respectively. Obviously, since $C \neq D$, there is some row i such that $C_i \neq D_i$. Recalling that $x = (1, r, r^2, \ldots, r^{n-1})$, observe that $(Cx)_i$ is precisely $p_{C_i}(r)$, the Reed-Solomon fingerprint of C_i as in the previous section. Similarly, $(A \cdot B \cdot x)_i = p_{D_i}(r)$. Hence, by the analysis of Section 2.1.3, the probability that $(Cx)_i \neq (A \cdot B \cdot x)_i$ is at least $1 - (n-1)/p$, and in this event the algorithm outputs NO.

2.2.5 Discussion

Whereas fingerprinting saved communication compared to a deterministic protocol, Freivalds' algorithm saves *runtime* compared to the best known deterministic algorithm. We can think of Freivalds' algorithm as our first probabilistic proof system: here, the proof is simply the answer C itself, and the $O(n^2)$-time verification procedure simply checks whether $Cx = A \cdot Bx$.

Freivalds actually described his algorithm with a perfectly random vector $x \in \mathbb{F}_p^n$, rather than $x = (1, r, r^2, \ldots, r^{n-1})$ for a random $r \in \mathbb{F}_p$ (see Exercise 3.1). We chose $x = (1, r, r^2, \ldots, r^{n-1})$ to ensure that $(Cx)_i$ is a Reed-Solomon fingerprint of row i of C, thereby allowing us to invoke the analysis from Section 2.1.

2.3 An Alternative View of Fingerprinting and Freivalds' Algorithm

Recall from Section 2.1.5 that the fingerprinting protocol for equality testing can be viewed as follows. Alice and Bob replace their length-n vectors $a, b \in \mathbb{F}_p^n$ with so-called Reed-Solomon encodings of these vectors. These encodings are vectors of length $p \gg n$. They interpret a and b as

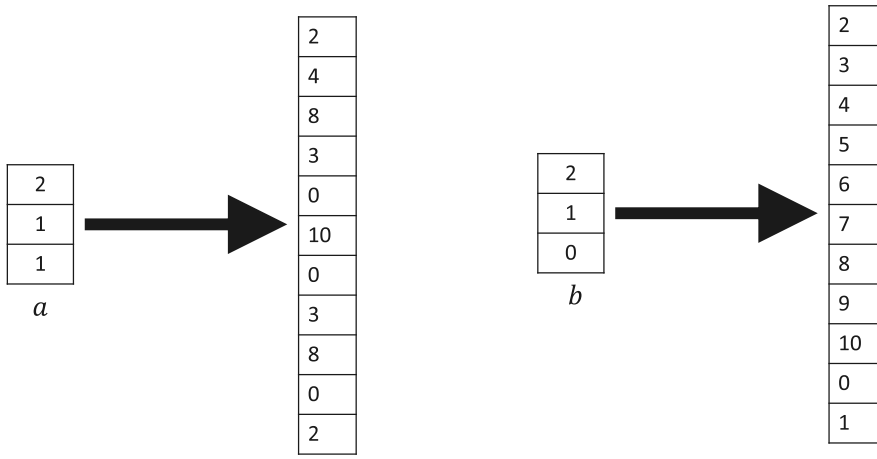

Figure 2.1: On the left is the vector $a = (2, 1, 1)$ of length 3 with entries interpreted as elements of the field $\mathbb{F}_{11}$, as well as its Reed-Solomon encoding. The Reed-Solomon encoding interprets a as the polynomial $p_a(x) = 2 + x + x^2$ and lists all evaluations of p_a over the field $\mathbb{F}_{11}$. On the right is the vector $b = (2, 1, 0)$ and its Reed-Solomon encoding.

specifying polynomials p_a and p_b over $\mathbb{F}_p$, and for each $r \in \mathbb{F}_p$, the r'th entry of the encodings of a and b are respectively $p_a(r)$ and $p_b(r)$. See Figure 2.1 for an example.

The Reed-Solomon encoding of a vector a is a *much* larger vector than a itself—whereas a has length n, the encoding of a has length p. The encoding is *distance-amplifying*: if a and b differ on even a single coordinate, then their encodings will differ on a $1 - (n-1)/p$ fraction of coordinates.[8] Due to the distance-amplifying nature of the code, it is enough for Alice to pick a single random entry of the encoding of her vector a and send it to Bob, who compares it to the corresponding entry of b's encoding.

[8]Reed-Solomon codes, and other encoding procedures used in this monograph, are typically called *error-correcting* codes rather than *distance-amplifying* codes. Distance-amplification of the encodings in fact implies error-correcting properties, meaning that if some entries of an encoding are corrupted, the "true" encoding can be recovered. However, no parties in any of the protocols in this monograph ever need to correct errors—only the distance-amplifying properties of the encoding procedure are exploited by the protocols.

Hence, checking equality of two vectors a and b was reduced to checking equality of a *single* (randomly chosen) entry of the encodings. Note that while the encodings of a and b are huge vectors, neither Alice nor Bob ever needed to materialize the full encodings—they both only needed to "access" a single random entry of each encoding.

Similarly, Freivalds' algorithm can be thought of as evaluating a single randomly chosen entry of the Reed-Solomon encoding of each row of the claimed answer C and the true answer D, and comparing the results. Evaluating just a single entry of the encoding of each row of D can be done in just $O(n^2)$ time, which is much faster than any known algorithm to compute D from scratch.

In summary, both protocols reduced the task of checking equality of two large objects (the vectors a and b in the fingerprinting protocol, and the claimed answer matrix and true answer matrix in Freivalds' algorithm) to checking equality of just a single random entry of distance-amplified encodings of those objects. While deterministically checking equality of the two large objects would be very expensive in terms of either communication or computation time, evaluating a single entry of the each object's encoding can be done with only logarithmic communication or in just linear time.

2.4 Univariate Lagrange Interpolation

The Reed-Solomon encoding of a vector $a = (a_1, \ldots, a_n) \in \mathbb{F}^n$ described in Section 2.3 interprets a as the *coefficients* of a univariate polynomial p_a of degree $n-1$, i.e., $p_a(X) = \sum_{i=1}^{n} a_i X^{i-1}$. There are other ways to interpret a as the description of a univariate polynomial q_a of degree $n-1$. The most natural such alternative is to view $a_1, \ldots, a_n$ as the *evaluations* of q_a over some canonical set of inputs, say, $\{0, 1, \ldots, n-1\}$. Indeed, as we now explain, for any list of n (input, output) pairs, there is a unique univariate polynomial of degree $n-1$ consistent with those pairs. The process of defining this polynomial q_a is called *Lagrange interpolation* for univariate polynomials.

Lemma 2.3 (Univariate Lagrange Interpolation)**.** Let p be a prime larger than n and $\mathbb{F}_p$ be the field of integers modulo p. For any vector $a =$

$(a_1, \ldots, a_n) \in \mathbb{F}^n$, there is a unique univariate polynomial q_a of degree at most $n-1$ such that

$$q_a(i) = a_{i+1} \text{ for } i = 0, \ldots, n-1. \tag{2.2}$$

Proof. We give an explicit expression for the polynomial q_a with the behavior claimed in Equation (2.2). To do so, we introduce the notion of Lagrange basis polynomials.

Lagrange basis polynomials. For each $i \in \{0, \ldots, n-1\}$, define the following univariate polynomial δ_i over $\mathbb{F}_p$:

$$\delta_i(X) = \prod_{k=0,1,\ldots,n-1:\ k \neq i} (X-k)/(i-k). \tag{2.3}$$

It is straightforward to check that $\delta_i(X)$ has degree at most $n-1$, since the product on the right hand side of Equation (2.3) has $n-1$ terms, each of which is a polynomial in X of degree 1. Moreover, it can be checked that δ_i maps i to 1 and maps all other points in $\{0, 1, \ldots, n-1\}$ to 0.[9] In this way, δ_i acts as an "indicator function" for input i, in that it maps i to 1 and "kills" all other inputs in $\{0, 1, \ldots, n-1\}$. δ_i is referred to as the i'th *Lagrange basis polynomial*.

For example, if $n = 4$, then

$$\delta_0(X) = \frac{(X-1)\cdot(X-2)\cdot(X-3)}{(0-1)\cdot(0-2)\cdot(0-3)} = -6^{-1}\cdot(X-1)(X-2)(X-3), \tag{2.4}$$

$$\delta_1(X) = \frac{(X-0)\cdot(X-2)\cdot(X-3)}{(1-0)\cdot(1-2)\cdot(1-3)} = 2^{-1}\cdot X(X-2)(X-3), \tag{2.5}$$

$$\delta_2(X) = \frac{(X-0)\cdot(X-1)\cdot(X-3)}{(2-0)\cdot(2-1)\cdot(2-3)} = -2^{-1}\cdot X(X-1)(X-3), \tag{2.6}$$

and

$$\delta_3(X) = \frac{(X-0)\cdot(X-1)\cdot(X-2)}{(3-0)\cdot(3-1)\cdot(3-2)} = 6^{-1}\cdot X(X-1)(X-2). \tag{2.7}$$

Expressing q_a in terms of the Lagrange basis polynomials. Recall that we wish to identify a polynomial q_a of degree $n-1$ such that

[9] Note, however, that $\delta_i(r)$ does *not* equal 0 for any points $r \in \mathbb{F}_p \setminus \{0, 1, \ldots, n-1\}$.

$q_a(i) = a_{i+1}$ for $i \in \{0, 1, \ldots, n-1\}$. We can define such a polynomial q_a in terms of the Lagrange basis polynomials as follows:

$$q_a(X) = \sum_{j=0}^{n-1} a_{j+1} \cdot \delta_j(X). \tag{2.8}$$

Indeed, for any $i \in \{0, 1, \ldots, n-1\}$, every term in the sum on the right hand side of Equation (2.8) other than the i'th evaluates to 0, because $\delta_j(i) = 0$ for $j \neq i$. Meanwhile, the i'th term evaluates to $a_{i+1} \cdot \delta_i(i) = a_{i+1}$ as desired. See Figure 2.2 for examples.

Establishing uniqueness. The fact that q_a defined in Equation (2.8) is the unique polynomial of degree at most $n-1$ satisfying Equation (2.2) holds because any two distinct polynomials of degree at most $n-1$ can agree on at most $n-1$ inputs. Since Equation (2.2) specifies the behavior of q_a on n inputs, this means that there cannot be two distinct polynomials of degree at most $n-1$ that satisfy the equation. □

Specifying a Polynomial Via Evaluations vs. Coefficients. Readers are likely already comfortable with univariate polynomials p of degree

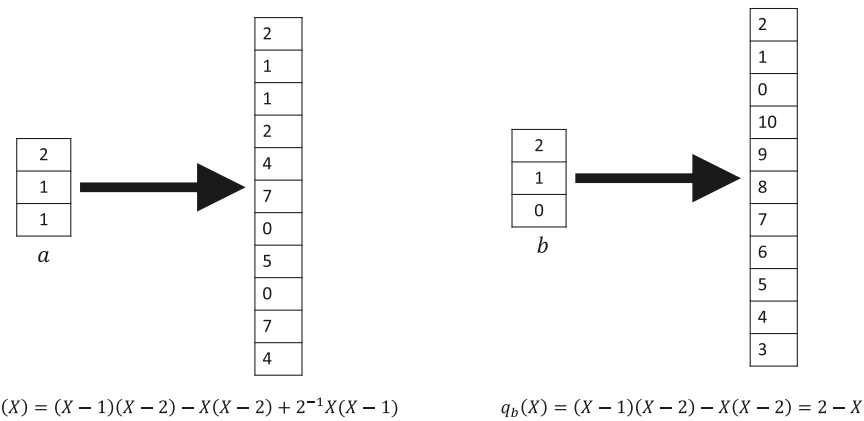

Figure 2.2: On the left is the vector $a = (2, 1, 1)$ of length 3 with entries interpreted as elements of the field $\mathbb{F}_{11}$, as well as its univariate low-degree extension encoding. This encoding interprets a as the evaluations of a univariate polynomial q_a over the input set $\{0, 1, 2\} \subseteq \mathbb{F}_{11}$, and the encoding lists all evaluations of q_a over the field $\mathbb{F}_{11}$. On the right is the vector $b = (2, 1, 0)$ and its low-degree extension encoding.

$(n-1)$ that are specified via coefficients in the standard monomial basis, meaning $c_0, \ldots, c_{n-1}$ such that

$$p(X) = c_0 + c_1 X + \cdots + c_{n-1} X^{n-1}.$$

As indicated before the statement of Lemma 2.3, the n evaluations $\{p(0), p(1), \ldots, p(n-1)\}$ can be thought of as an alternative specification of p. Just as the standard coefficients $c_0, c_1, \ldots, c_{n-1}$ uniquely specify p, so do prescribed evaluations at the n inputs $0, 1, \ldots, n-1$.

In fact, Equation (2.8) shows that these n evaluations of p can themselves be interpreted as coefficients for p, not over the standard monomial basis $\{1, X, X^2, \ldots, X^{n-1}\}$, but rather over the Lagrange polynomial basis $\{\delta_0, \delta_1, \ldots, \delta_{n-1}\}$. In other words, for $i \in \{0, 1, \ldots, n-1\}$, $p(i)$ is the coefficient of δ_i in the unique representation of p as a linear combination of Lagrange basis polynomials.

A Coding-Theoretic View. Given a vector $a = (a_1, \ldots, a_n) \in \mathbb{F}_p^n$, the polynomial q_a given in Lemma 2.3 is often called the *univariate low-degree extension* of a.[10] The viewpoint underlying this terminology is as follows. Consider the vector $\mathrm{LDE}(a)$ of length $p = |\mathbb{F}_p|$ whose ith entry is $q_a(i)$. If $p \gg n$, then $\mathrm{LDE}(a)$ is vastly longer than a itself. But $\mathrm{LDE}(a)$ contains a as a sub-vector, since, by design, $q_a(i) = a_{i+1}$ for $i \in \{0, \ldots, n-1\}$. One thinks of $\mathrm{LDE}(a)$ as an "extension" of a: $\mathrm{LDE}(a)$ "begins" with a itself, but includes a large number of additional entries. See Figure 2.2.

Such encoding functions, in which the vector a is a subset of its encoding $\mathrm{LDE}(a)$ are called *systematic*. The systematic nature of the low-degree extension encoding turns out to render it more useful in the context of interactive proofs and arguments than the Reed-Solomon encoding of Section 2.3 (see, for example, Section 10.3.2).

[10] Actually, many authors refer to *any* "reasonably low-degree" polynomial q satisfying $q(i) = a_{i+1}$ for $i \in \{0, 1, \ldots, n-1\}$ as a low-degree extension of a (however, there is always a *unique* extension polynomial q_a of a of degree at most $n-1$). What "reasonably low-degree" means varies by context, but typically the asymptotic costs of probabilistic proof systems that use univariate extension polynomials are unchanged so long as the extension polynomial has degree $O(n)$. At the bare minimum, the degree of the extension polynomial should be smaller than the size of the field over which the polynomial is defined. Otherwise, encoding a via the evaluation table of the extension polynomial will not be a distance-amplifying procedure.

Exactly as for the Reed-Solomon code in Section 2.3, LDE(a) is a distance-amplified encoding of a, in the sense that, for any two vectors $a, b \in \mathbb{F}_p^n$ that differ in even a single coordinate, LDE(a) and LDE(b) differ in at least a $1 - (n-1)/p$ fraction of entries. This fraction is very close to 1 if $p \gg n$.

A Note on Terminology. In the coding theory literature, a is referred to as a *message* and the encoding LDE(a) is called as the *codeword* corresponding to message a. Many authors use the term Reed-Solomon encoding and low-degree extension encoding interchangeably. Often, the distinction does not matter, as the set of codewords is the same regardless, namely the set of all evaluation tables of polynomials of degree at most $n-1$ over $\mathbb{F}_p$. All that differs between the two is the correspondence between messages and codewords, i.e., whether the message is interpreted as the coefficients of a polynomial of degree $n-1$, vs. as the evaluations of the polynomial over a canonical set of inputs such as $\{0, 1, \ldots, n-1\}$.

Algorithms for Evaluating $q_a(r)$. Suppose that, given a vector $a \in \mathbb{F}_p^n$, one wishes to evaluate the univariate low-degree extension q_a at some input $r \in \mathbb{F}$. How quickly can this be done? It turns out that $O(n)$ field additions, multiplications, and inversions are sufficient.[11]

If $r \in \{0, 1, \ldots, n-1\}$, then by definition, $q_a(r) = a_{r+1}$. So let us assume henceforth that $r \in \mathbb{F} \setminus \{0, 1, \ldots, n-1\}$.

Equation (2.8) offers an expression for $q_a(r)$ in terms of the Lagrange basis polynomials, namely

$$q_a(r) = \sum_{j=0}^{n-1} a_{j+1} \cdot \delta_j(r). \tag{2.9}$$

There are only n terms of this sum. However, evaluating the j'th term requires evaluating $\delta_j(r)$, and if this is done directly via its definition

[11] A single field inversion is a slower operation than a field addition or multiplication operation, often performed via the so-called Extended Euclidean algorithm. However, there are batch inversion algorithms that can perform n field inversions with roughly $3n$ field multiplications and one field inversion.

(Equation (2.3)), this requires $O(n)$ field operations per term, for a total time bound of $O(n \cdot n) = O(n^2)$.

Fortunately, it turns out that the n values $\delta_0(r), \delta_1(r), \ldots, \delta_{n-1}(r)$ can all be evaluated using just $O(n)$ additions, multiplications, and inversions *in total*. Once these values are all computed, the right hand side of Equation (2.9) can be computed with $O(n)$ additional field operations.

Here is how to evaluate $\delta_0(r), \delta_1(r), \ldots, \delta_{n-1}(r)$ with $O(n)$ additions, multiplications, and inversions. First, $\delta_0(r)$ can be evaluated with $O(n)$ such operations directly via its definition (Equation (2.3)).

Then, for each $i > 0$, given $\delta_{i-1}(r)$, $\delta_i(r)$ can be computed with a constant number of additional field subtractions, multiplications, and inversions. This is because the products defining $\delta_i(r)$ and $\delta_{i-1}(r)$ involve almost all of the same terms. For example, relative to

$$\delta_0(r) = \left(\prod_{k=1,\ldots,n-1} (r-k)\right)\left(\prod_{k=1,\ldots,n-1} (0-k)^{-1}\right),$$

the definition of

$$\delta_1(r) = \left(\prod_{k=0,2,3,\ldots,n-1} (r-k)\right) \cdot \left(\prod_{k=0,2,3,\ldots,n-1} (1-k)^{-1}\right)$$

is "missing" a factor of

$$(r-1) \cdot \Big(-(n-1)\Big)^{-1},$$

and has an "extra" factor of

$$(r-0)(1-0)^{-1} = r.$$

In other words, $\delta_1(r) = \delta_0(r) \cdot r \cdot (r-1)^{-1} \cdot (-(n-1))$. In general, for $i \geq 1$, the following key equation ensures that $\delta_i(r)$ can be computed from $\delta_{i-1}(r)$ with just $O(1)$ field additions, multiplications, and inversions:

$$\delta_i(r) = \delta_{i-1}(r) \cdot (r-(i-1)) \cdot (r-i)^{-1} \cdot i^{-1} \cdot (-(n-i)). \tag{2.10}$$

Theorem 2.4. Let $p \geq n$ be a prime number. Given as input $a_1, \ldots, a_n \in \mathbb{F}_p$, and $r \in \mathbb{F}_p$, there is an algorithm that performs $O(n)$ additions, multiplications, and inversions over $\mathbb{F}_p$, and outputs $q(r)$ for the unique univariate polynomial q of degree at most $n-1$ such that $q(i) = a_{i+1}$ for $i \in \{0, 1, \ldots, n-1\}$.

A Worked Example of Equation (2.10). When $n = 4$, explicit expressions for δ_0, δ_1, δ_2, and δ_3 were given in Equations (2.4)–(2.7). One can check that Equation (2.10) holds for each of these Lagrange basis polynomials. Indeed,

$$\begin{aligned}
\delta_0(r) &= -6^{-1} \cdot (r-1)(r-2)(r-3), \\
\delta_1(r) &= 2^{-1} \cdot r(r-2)(r-3) = \delta_0(r) \cdot r \cdot (r-1)^{-1} \cdot 1^{-1} \cdot (-(n-1)), \\
\delta_2(r) &= -2^{-1} \cdot r(r-1)(r-3) = \delta_1(r) \cdot (r-1) \cdot (r-2)^{-1} \cdot 2^{-1} \cdot (-(n-2)),
\end{aligned}$$

and

$$\delta_3(r) = 6^{-1} \cdot r(r-1)(r-2) = \delta_2(r) \cdot (r-2) \cdot (r-3)^{-1} \cdot 3^{-1} \cdot (-(n-3)).$$