7.4. Verifying Equality of Strings

We have seen that the idea of fingerprinting is useful in verifying identities of algebraic objects. In this section we introduce a different form of fingerprinting, motivated by the problem of testing the equality of two strings. As mentioned earlier, the string equality verification problem can be reduced to that of verifying polynomial identities. However, the new type of fingerprint introduced here has important benefits when extended to the pattern matching problem discussed later in Section 7.6.

Suppose that Alice maintains a large database of information. Bob maintains a second copy of the database. Periodically, they must compare their databases for consistency. Because transmission between Alice and Bob is expensive, they would like to discover the presence of an inconsistency without transmitting the entire database between them. Denote Alice's data by the sequence of bits $(a_1,\ldots,a_n)$, and Bob's by the sequence $(b_1,\ldots,b_n)$. It is clear that any deterministic consistency check that transmits fewer than n bits will fail if an adversary could decide which bits of either database to modify. We describe a randomized strategy that detects an inconsistency with high probability while transmitting far fewer than n bits of information.

We use the following simple fingerprint mechanism. Interpret the data as n-bit integers a and b, by defining $a = \sum_{i=1}^{n} a_i 2^{i-1}$ and $b = \sum_{i=1}^{n} b_i 2^{i-1}$. Define the fingerprint function $F_p(x) = x \bmod p$ for a prime p. Then Alice can transmit $F_p(a)$ to Bob, who in turn can compare this with $F_p(b)$. The hope is that if $a \neq b$, then it will also be the case that $F_p(a) \neq F_p(b)$. The number of bits to be transmitted is $O(\log p)$, which will be much smaller than n for a small prime p. This strategy can be easily foiled by an adversary for any fixed choice of p since, for any p and b, there exist many choices of a for which $a \equiv b \pmod{p}$. We get around this problem by choosing p at random.

For any number k, let $\pi(k)$ be the number of distinct primes less k. A well-known result in number theory is the Prime Number Theorem, which states that $\pi(k)$ is asymptotically $k/\ln k$. Consider now the non-negative integer $c = |a-b|$. The fingerprint defined above fails only when $c \neq 0$ and p divides c. How many primes can divide c? Define $N = 2^n$; we know that $c \leq N$.

Lemma 7.4: *The number of distinct prime divisors of any number less than 2^n is at most n.*

PROOF: Each prime number is greater than 1. If N has more than t distinct prime divisors, then $N \geq 2^t$. □

Choose a threshold τ that is larger than $n = \log N$. The number of primes smaller than τ is $\pi(\tau) \sim \tau/\ln\tau$. Of these, at most n can be divisors of c and cause our fingerprint function to fail. Therefore, we pick a random prime p smaller than τ for defining F_p. The number of bits of communication is $O(\log\tau)$. Choose

$\tau = tn \log tn$, for large t. The following theorem is immediate. The probability is taken over the random choice of p.

Theorem 7.5: $\mathbf{Pr}[F_p(a) = F_p(b) \mid a \neq b] \leq \dfrac{n}{\pi(\tau)} = O\left(\dfrac{1}{t}\right)$.

Thus, we get an error probability of at most $O(1/t)$, and the number of bits to be transmitted is $O(\log t + \log n)$. Choosing $t = n$ gives us an excellent strategy for this problem. We remark that the task of picking a random prime is non-trivial, primarily because verifying the primality of a number is difficult. Some algorithms for this purpose will be presented in Chapter 14.

► **Example 7.1:** This integer equality verification technique can be used to solve the problem alluded to at the end of Section 7.2. In verifying that a multivariate polynomial $Q(x_1, \ldots, x_n)$ is identically zero, we evaluate the polynomial at a random point. The problem is that the intermediate values arising in the evaluation of $q = Q(r_1, \ldots, r_n)$ could be extremely large. Of course, we do not really wish to compute q; our goal is to merely verify that $q = 0$. By the preceding discussion, it suffices to verify that $q \bmod p = 0$ for some small random prime p.

But how can we possibly hope to perform the verification without evaluating q explicitly? The trick is to use arithmetic modulo p while evaluating $Q(r_1, \ldots, r_n)$ and thereby obtain the residue of q modulo p directly, rather than first computing q and then reducing it modulo p. The intermediate values are all smaller than p, and p itself is chosen to be a small random prime. By Theorem 7.5, the probability of error does not increase significantly for a suitable choice of t.

7.5. A Comparison of Fingerprinting Techniques

It is useful at this point to compare the two types of fingerprinting techniques that we have seen so far. Suppose that we wish to verify the equality of two strings or vectors $\boldsymbol{a} = (a_1, \ldots, a_n)$ and $\boldsymbol{b} = (b_1, \ldots, b_n)$ with each component drawn from a finite alphabet Σ. We can encode the alphabet symbols using the set of numbers $\Gamma = \{0, 1, \ldots, k-1\}$, where $k = |\Sigma|$. It is then possible to view the two strings as the polynomials $A(z) = \sum_{i=0}^{n-1} a_i z^i$ and $B(z) = \sum_{i=0}^{n-1} b_i z^i$, each of which has integer coefficients and degree at most n. Clearly, the two vectors are identical if and only if the two polynomials are identical.

The fingerprinting technique of Sections 7.1 and 7.2 can be summarized as follows. Fix a prime number p greater than both $2n$ and k. View the polynomials $A(z)$ and $B(z)$ as polynomials over the field $\mathbb{Z}_p$. By our choice of p, the set Γ is contained in this field and arithmetic modulo p will not render identical any two non-identical polynomials. The fingerprint of the two polynomials is obtained by choosing a random element $r \in \mathbb{Z}_p$ and substituting it for the symbolic variable z. If $\boldsymbol{a} = \boldsymbol{b}$, then the two polynomials are identical and the fingerprint will also be identical; on the other hand, when $\boldsymbol{a} \neq \boldsymbol{b}$, the two polynomials are distinct

and the probability that their fingerprints turn out to be the same is at most n/p, and this is bounded by $1/2$ for our choice of p. For $k = 2$ and $p = O(n)$, this can be viewed as reducing the problem of comparing n-bit numbers to that of comparing $O(\log n)$-bit numbers.

The fingerprinting technique from Section 7.4 is in some sense a dual of the first technique. In this approach, we fix $z = 2$ and choose a random prime q of a reasonably small magnitude. The fingerprints are obtained by evaluating $A(2)$ and $B(2)$ over the field $\mathbb{Z}_q$. Thus, instead of fixing the field and evaluating at a random point in the field, the second type of fingerprint is obtained by fixing the point of evaluation and choosing a random field over which the evaluation is to be performed. By our analysis in Section 7.4, this also reduces the problem of comparing n-bit numbers to that of comparing $(\log n)$-bit numbers. However, as we will see in the next section, there are certain applications where the second type of fingerprinting proves to be more useful.

A third version of the fingerprinting approach works as follows. Assume that $k = 2$, and interpret the bit vectors $\boldsymbol{a}$ and $\boldsymbol{b}$ as the n-bit integers a and b. Fix a prime number $p > 2^n$. Choose a random polynomial $P(z)$ over the field $\mathbb{Z}_p$, and obtain the fingerprints by evaluating this polynomial at the integers a and b, performing all arithmetic over the field $\mathbb{Z}_p$, and then reducing the resulting values modulo a number of magnitude close to $\log n$. This is the main idea behind the construction of the so-called *universal hash functions* discussed in Section 8.4.

7.6. Pattern Matching

Consider now the problem of pattern matching in strings. A *text* is a string $X = x_1x_2\ldots x_n$ and a *pattern* is a string $Y = y_1y_2\ldots y_m$, both over a fixed finite alphabet Σ, such that $m \leq n$. Without loss of generality, we restrict ourselves to the case $\Sigma = \{0,1\}$. The pattern occurs in the text if there is a $j \in \{1,2,\ldots,n-m+1\}$ such that for $1 \leq i \leq m$, $x_{j+i-1} = y_i$. The pattern matching problem is that of finding an occurrence (if any) of a given pattern in the text. This problem can be trivially solved in $O(nm)$ time by trying for a match at all possible locations i; moreover, there are deterministic algorithms that achieve the best possible running time of $O(n+m)$.

We describe a Monte Carlo algorithm that also achieves a running time of $O(n+m)$; later, we will convert this into a Las Vegas algorithm. This algorithm is interesting despite the existence of linear-time deterministic algorithms because it is significantly simpler, has a "real-time" implementation (this is explained below), and generalizes to the problem of pattern matching in two-dimensional strings (or matrices).

Define the string $X(j) = x_jx_{j+1}\ldots x_{j+m-1}$ as the sub-string of length m in X that starts at position j. A match occurs if there is a choice of j, for $1 \leq j \leq n-m+1$, for which $Y = X(j)$. We make the solution unique by requiring that the algorithm find the smallest value of j such that $X(j) = Y$.

The brute-force $O(nm)$ time algorithm compares Y with each of the strings $X(j)$. Our randomized algorithm will choose a fingerprint function F and compare $F(Y)$ with each of the fingerprints $F(X(j))$. An error occurs if $F(Y) = F(X(j))$ but $Y \neq X(j)$. We would like to choose a function F that has a small probability of error and can be efficiently computed.

In fact, we use the same fingerprint function as in Section 7.4: for any string $Z \in \{0,1\}^m$, interpret Z as an m-bit integer and define $F_p(Z) = Z \bmod p$. Assume that p is chosen uniformly at random from the set of primes smaller than a threshold τ. Suppose that we interpret the strings Y and $X(j)$ as m-bit integers, and compare their fingerprints $F_p(Y)$ and $F_p(X(j))$ instead of trying to match each symbol in the two strings. The only possible error is that we get identical fingerprints when $Y \neq X(j)$. By Theorem 7.5, we bound the probability of such a *false match* as follows:

$$\mathbf{Pr}[F_p(Y) = F_p(X(j)) \mid Y \neq X(j)] \leq \frac{m}{\pi(\tau)} = O\left(\frac{m \log \tau}{\tau}\right).$$

Then, the probability that a false match occurs for any of the at most n values of j is $O((nm \log \tau)/\tau)$. We choose $\tau = n^2 m \log n^2 m$, and this gives

$$\mathbf{Pr}[\text{a false match occurs}] = O\left(\frac{1}{n}\right).$$

The Monte Carlo version of this algorithm simply compares the fingerprints of all $X(j)$ to that of Y, and outputs the first j for which a match occurs; the Las Vegas version will be described below. We first show that the running time of this algorithm is as claimed. For $1 \leq j \leq n - m + 1$,

$$X(j+1) = 2\left[X(j) - 2^{m-1}x_j\right] + x_{j+m}.$$

From this we obtain the recurrence

$$F_p(X(j+1)) = 2\left[F_p(X(j)) - 2^{m-1}x_j\right] + x_{j+m} \bmod p.$$

It is now clear that given the fingerprint of $X(j)$, the incremental cost of computing the fingerprint of $X(j+1)$ is $O(1)$ field operations. In fact, there is no need to use the more expensive operations of multiplication and division, because each x_j is 0 or 1. Thus, the total time required for this algorithm is $O(n+m)$ even under the more stringent log-cost RAM model. This efficient incremental update property is the main motivation for using the second form of fingerprinting; the reader may verify that more complex computations would be required if the first form of fingerprinting was used instead (see Section 7.5).

Theorem 7.6: *The Monte Carlo algorithm for pattern matching requires* $O(n+m)$ *time and has a probability of error* $O(1/n)$.

It is easy to convert this into a Las Vegas algorithm. Whenever a match occurs between the fingerprints of Y and some $X(j)$, we compare the strings Y and $X(j)$ in $O(m)$ time. If this is a false match, we detect it and abandon

the whole process in favor of using the brute-force $O(nm)$ time algorithm. The new algorithm does not make any errors and has expected running time $O((n+m)(1-1/n)+nm(1/n))$, which works out to be $O(n+m)$. An alternative Las Vegas version of this algorithm restarts the entire algorithm with a new random choice of p whenever a false match is detected. In the latter approach, the probability of having to restart more than t times is bounded by $1/n^t$. This leads to a very small variance in the running time. In contrast, the first approach has a relatively high probability of being forced to use the $O(nm)$ time algorithm, and hence has a high variance in the running time.

An alternative fingerprint function with a similar behavior is described in Problem 7.12. In Problem 7.13 it is required to show that this algorithm extends to the case of two-dimensional pattern matching.

The method for computing the fingerprints of the various $X(j)$'s will work well in on-line or real-time settings where the string X is provided incrementally, possibly a bit at a time. This feature is also useful when the text is extremely large and cannot be completely stored in the primary memory of a machine.

Exercise 7.4: Consider the fingerprint function used for polynomial identities and adapt it to the problem of testing string equality. Why is this not a good choice of a fingerprint for the pattern matching problem?

7.7. Interactive Proof Systems

We have seen the power of combining randomization and algebra in devising fingerprinting techniques with applications to efficient verification of simple identities involving objects such as matrices, polynomials, and strings. We have also seen that the basic idea used in the verification of the equality of two strings x and y could be taken a step further and be used for the efficient detection of a pattern y in a string x. How far can we push this approach?

Suppose, for example, the string x represents a graph G, and the "pattern" y represents some graph property P. Can we then use the ideas developed here for efficient "pattern matching" in terms of verifying the property P in G? More specifically, suppose that the pattern y corresponds to the property of *not* being an expanding graph. The problem of verifying this property belongs to ***NP*** and so there exist short proofs of non-expansion. Moreover, given such a proof, it is possible to efficiently verify its correctness. Thus, the pattern matching task can be efficiently performed provided the pattern y includes a "proof" of this fact, i.e., a description of a set of vertices in G that do not have too many neighbors. In this context, efficiency means time polynomial in the length of the inputs, and this requires that the proof itself be of polynomial length.

Suppose instead the pattern matching task corresponds to the verification of the property of being an expander. As we mentioned earlier (Section 6.7), this