1 Communication Complexity

Last time, we introduced a communication complexity problem: Alice and Bob each have an $n$-bit string (say $x = x_1x_2\ldots x_n$ and $y = y_1\ldots y_n$), and they want to determine whether $x = y$ while interchanging as few bits as possible. (It suffices to assume $x$ and $y$ are the same length, as Alice only has to send Bob $\log n$ bits to ensure both strings are of length $n$, which is low overhead.) This question has application to database consistency: two copies of the database are separately updated, and we want to ensure the two copies are the same.

It is too expensive for Alice to send Bob the entire string, which is what would be required by a deterministic algorithm. However, there is a simple randomized algorithm that does the job with low probability of error.

**Algorithm 1.** Translate $x$ into a number $N_x$ and $y$ into a number $N_y$ such that $N_x, N_y \leq 2^n$ for some $n$. Alice then randomly picks a prime $p \in \{1, \ldots, n^3\}$, and sends the pair $(N_x \pmod p, p)$. Bob asserts $x = y$ iff $N_x \pmod p \equiv N_y \pmod p$.

It requires $3\log n$ bits to represent $p$, and $3\log n$ bits for $N_x \pmod p$, so Algorithm 1 requires $O(\log n)$ communication bits.

**Analysis of Algorithm 1:** If $x = y$, Bob accepts, as $N_x \pmod p \equiv N_y \pmod p$. So suppose $x \neq y$. We need to determine $\Pr_p[N_x \pmod p \equiv N_y \pmod p]$. Note this is the same as $\Pr_p[N_x \equiv N_y]$. $N_x - N_y$ has at most $n$ prime factors, as $N_x - N_y \leq 2^n$. By the Prime Number Theorem, the density of primes from 1 to $n$ is approximately $n/\log n$, so the number of primes in the range $\{1, \ldots, n^3\}$ is about $n^3/3\log n$. So $\Pr_p[N_x \equiv N_y] \approx 1/n^2 \log n.$

The point: as $n$ grows large, $1/n^2$ becomes very small, so the error probability is very small.

There is no way to solve this problem deterministically without sending all the bits. Allowing randomness permits a drastic reduction in complexity. Let’s look at another solution to this problem.

**Algorithm 2.** Alice randomly chooses a prime $p \geq n^2$. We view $x$ and $y$ as
polynomials over $GF(p)$. That is to say,

$$P_x(t) = x_1 + x_2t + x_3t^2 + \cdots + x_nt^{n-1}$$

$$P_y(t) = y_1 + y_2t + y_3t^2 + \cdots + y_nt^{n-1}.$$ 

Then Alice randomly picks $a \in \{0, \ldots, p-1\}$, and sends to Bob the pair $(P_x(a), p)$. Bob accepts if $P_y(a) = P_x(a)$.

**Analysis of Algorithm 2:** As before, if $x = y$, Bob accepts with probability 1. If $x \neq y$, Pr[Bob accepts] $\leq (n-1)/p$. (This is due to the fact that two distinct degree $n$ polynomials over $GF(p)$ can coincide on at most $n$ points.) As before, the communication complexity is $O(\log n)$.

The general scheme in the case of both algorithms is that of error-correcting codes. Alice and Bob agree on such a code $C$. Alice computes $C(x)$ and sends a random bit of $C(x)$. Bob looks at the same bit in $C(y)$ and checks to see they are the same. The idea behind an error-correcting code is that if $x \neq y$, then $C(x)$ differs from $C(y)$ in many places. The first code used is the Chinese Remainder Code; the second, the Reed-Solomon Code.

**Theorem 1 (Chinese Remainder Theorem).** Suppose $n_1, n_2, \ldots, n_k$ are integers which are pairwise coprime. Then, for any given integers $a_1, a_2, \ldots, a_k$, there exists an integer $x$ solving the system of simultaneous congruences

$$x \equiv a_1 \pmod{n_1}$$
$$x \equiv a_2 \pmod{n_2}$$
$$\vdots$$
$$x \equiv a_k \pmod{n_k}.$$ 

Furthermore, all solutions $x$ to this system are congruent modulo the product $N = n_1n_2\ldots n_k$.

Let $p_1, p_2, \cdots, p_n$ be first $n$ primes. Let $m$ be any integer between 1 and $M = \prod_{i=1}^{k} p_i$ for some $k$ less than $n$. Now the Chinese remainder code of an integer $m$ that is less than $M$ is

$$CR(x) = \{m \mod p_1, m \mod p_2, \cdots, m \mod p_n\}.$$ 

Observe that if $m \neq l$ and, then $CR(m)$ and $CR(l)$ differ in at least $n-k$ places, when both $m$ and $l$ are atmost $M$.
Definition 1 (Reed-Solomon Codes). Say \( x = x_1 \ldots x_n \). View \( x \) as a polynomial, \( p_x(t) \), over \( \text{GF}(p) \).

\[
p_x(t) = x_0 + x_1 t + x_2 t^2 + \cdots + x_n t^n.
\]

Fix points \( y_1, \ldots, y_m \) over \( \text{GF}(p) \) such that \( m \gg n - 1 \). The Reed-Solomon code then is defined to be

\[
\text{RS}(x) = \langle y_1, p_x(y_1), y_2, p_x(y_2), \ldots, y_m, p_x(y_m) \rangle.
\]

If \( x \neq y \), then \( \text{RS}(x) \) and \( \text{RS}(y) \) differ in at least \( m - n \) places.

2 Pattern Matching

Suppose we have a text \( x = x_1 \ldots x_n \) and a pattern \( y = y_1 \ldots y_m \). We want to determine whether \( y \) occurs within \( x \) in the “consecutive substring” problem. A naive approach would give us \( \mathcal{O}(nm) \), and a more sophisticated deterministic approach gives us \( \mathcal{O}(m+n) \). We will construct a randomized algorithm that also has asymptotic complexity \( \mathcal{O}(m+n) \), but is much easier to implement.

Algorithm 3 (Karp-Rabin). This algorithm is written for a RAM that uses a fixed word size, say 32 bits, and we assume that an operation on any two words takes one step. This is called the “unit cost RAM model.” As before, translate string \( y \) into a number \( N_y \). Break \( x \) into blocks of length \( m \), and write \( x_i \) for one such block, so \( x_i = x_{i-1} \ldots x_{i+m-1} \). Produce the number \( N_{x_i} \) for each block \( x_i \). The typical number is large, so we do everything modulo a prime. Pick a prime \( p \in \{1, \ldots, 2^{32}\} \), so \( p \) fits within one word. The algorithm then compares \( N_y \pmod{p} \) with \( N_{x_1} \pmod{p} \), with \( N_{x_2} \pmod{p} \), etc.

Analysis of Algorithm 3: First note that \( y \) appears in \( x \) if at least one of the comparisons succeeds. The algorithm decides incorrectly if \( N_y \pmod{p} \equiv N_{x_j} \pmod{p} \) but \( N_y \not\equiv N_{x_j} \). That happens when \( p \) divides \( N_y - N_{x_j} \). So let’s estimate the error probability \( \epsilon \). We know that

\[
\epsilon \leq \sum_{i=1}^{n} \Pr[N(x_i) \neq N(y) \text{ and } N(x_i) \equiv N(y) \pmod{p}],
\]

3
Each $N(x_i)$ has at most $m$ prime factors. We chose $p$ at the beginning to be a prime such that $1 \leq p \leq 2^{32}$. By the Prime Number Theorem, there are about $2^{32}/32 = 2^{27}$ such primes. This means that

$$\epsilon \leq n \left( \frac{m}{2^{27}} \right)$$

so as long as $mn \ll 2^{27}$, our error probability is reasonable. Furthermore, if the algorithm returns a probable matching location, we can then deterministically check whether there is indeed a match there. That reduces our error probability to zero.

The best-known deterministic pattern-matching algorithm requires preprocessing and has hidden constants. In the Karp-Rabin algorithm, the (mod $p$) operation reduces the length of $m$ to $p$.

## 3 Fingerprints

The general theme in the above algorithms is the following: We want to know whether two objects $x$ and $y$ are the same or not. However, comparing $x$ and $y$ directly is expensive. Instead, we would like to define a function $h$ with following properties: Size of $h(x)$ small compared to the size of $x$. If $x = y$, then $h(x) = h(y)$, else $h(x)$ and $h(y)$ differ a lot. Such function $h$ is called a finger print function. Of course, we can have a one finger-print function. Thus we defined a family of functions, and showed that a random function has desired properties. Now, we will see one more example.

Suppose we have large strings $x$ and $y$, and a hashing function $h$ that produces much shorter fingerprints. We will compare $x$ and $y$ by comparing their fingerprints $h(x)$ and $h(y)$.

As an application, consider three $n \times n$ matrices: $A, B,$ and $C$. Our question to solve is whether $A \times B = C$. We can of course deterministically multiply all bits. That algorithm runs in time $O(n^3)$. As an alternative, we could multiply fingerprints of $A$ and $B$, and check whether that result is equal to the fingerprint of $C$.

**Algorithm 4.** Randomly pick an $n \times 1$ matrix $R$. If $ABR = CRC$, decide that $AB = C$. Otherwise, decide that $AB \neq C$.

*Analysis of Algorithm 4:* This algorithm runs in $O(n^2)$ time. We will now calculate the error probability that $AB \neq C$ even if the fingerprints match.
Letting $D = AB$, and writing $r_1, \ldots, r_n$ for the entries in $R$, we want to compare

$$ABR = DR = \begin{bmatrix} d_{11} & \cdots & \cdots \\ \vdots & \ddots & \vdots \\ \vdots & \vdots & d_{nn} \end{bmatrix} \begin{bmatrix} r_1 \\ \vdots \\ r_n \end{bmatrix}$$

with

$$CR = \begin{bmatrix} c_{11} & \cdots & \cdots \\ \vdots & \ddots & \vdots \\ \vdots & \vdots & c_{nn} \end{bmatrix} \begin{bmatrix} r_1 \\ \vdots \\ r_n \end{bmatrix}$$

Multiplied through, we get

$$DR = CR$$

$$d_{11}r_1 + \cdots + d_{1n}r_n = c_{11}r_1 + \cdots + c_{1n}r_n$$

$$r_1 = \frac{(d_{12}r_2 + \cdots + d_{1n}r_n) - (c_{12}r_2 + \cdots + c_{1n}r_n)}{d_{11}c_{11}}.$$ 

So the algorithm is only wrong if $r_1$ is one single number, so the error probability is very small.

Being the same at multiple positions in a row does not cause problems because we can look just at $d_{11}$ and $c_{11}$ given vectors $d_1$ and $c_1$.

### 4 Data Structures

Suppose $U$ is the universe, and $S \subseteq U$. Typically for this problem statement $|S| \ll |U|$. We want to store $S$ in a table, and perform the following operations: Query (is $k \in S$?), Insert and Delete.

For example, using a balanced binary tree, each operation requires $O(|S|)$ time and $O(|S| \log |S|)$ space.

We can reduce Query to $O(1)$ time. Query is usually the most common operation. The “Static Dictionary Operation” is: given $S$, store $S$ in a table so search takes constant time. It also uses the RAM model we considered before.

The most obvious solution is to create a table of size $|U|$ by using the characteristic function. But that table would be too large. Our goal is to get a table near the size of $S$. Our general strategy will be to define a function
\( h : U \rightarrow T \) that is one-one on \( S \), so that, given \( k \), we will compute \( h(k) \) and store \( k \) at the \( h(k) \)th location of \( T \).

**Definition 2 (2-universal).** Let \( \mathcal{H} \) be a set of functions from \( U \) to \( T \). We say \( \mathcal{H} \) is 2-universal if \( \forall x \neq y \in U, \forall \alpha, \beta \in T \)

\[
\Pr_{h \in \mathcal{H}}[h(x) = \alpha \text{ and } h(y) = \beta] = \frac{1}{|T|^2}.
\]

**Properties:** Say \( \mathcal{H} \) is 2-universal. Then \( \forall x \in U, \forall \alpha \in T \)

\[
\Pr_{h \in \mathcal{H}}[h(x) = \alpha] = \frac{1}{|T|}.
\]

To see this, pick a \( y \) that is not equal to \( x \).

\[
\Pr_{h \in \mathcal{H}}[h(x) = \alpha] = \sum_{\beta \in T} \Pr_{h \in \mathcal{H}}[h(x) = \alpha \text{ and } h(y) = \beta]
\]

\[
= |T| \times \frac{1}{|T|^2}
\]

\[
= \frac{1}{|T|}
\]

By a similar argument, the collision probability is as follows: \( \forall x \neq y \in U \),

\[
\Pr_{h \in \mathcal{H}}[h(x) = h(y)] = 1/|T|.
\]

The set of all functions over \( U \) is 2-universal. A more useful example is the following set \( \mathcal{H} \) of functions, where \( |\mathcal{H}| = p^2 \) for a prime \( p \).

Suppose \( U = T = \{0, \ldots, p - 1\} \). Let \( \mathcal{H} = \{h_{ab} \mid a, b \in \mathbb{Z}_p\} \), where

\[
h_{ab}(x) = ax + b \pmod{p}.
\]