1 Introduction

In the last lecture we studied Berlekamp-Welsh decoding of Reed-Solomon code. In this lecture we shall first look at another decoding algorithm for Reed-Solomon code, by P. Madhusudan (1994). Then we will discuss some applications of our first tool to design algorithms - Divide and Conquer Technique.

2 Madhusudan’s Decoding Algorithm of Reed-Solomon Code

First we state the main features of these two decoding algorithms of Reed-Solomon code.

Berlekamp-Welsh Algorithm

- needs solving system of linear equations
- corrects upto $\frac{1}{2}(n - k)$ errors

Madhusudan’s Algorithm

- corrects upto $n - 2\sqrt{nk}$ errors
- needs more algebraic operations

Let us restate the notations used for Reed-Solomon code from lecture 0.

- The data to be stored on a CD is divided into chunks of $b \times k$ bits and each chunk is coded separately.
- $F$ is finite field of size $2^b$.
- Each chunk is again divided into $k$ blocks of $b$ bits each, say $d_0, d_1, \cdots, d_{k-1}$.
- Each $d_i$ is treated as an element of field $F$.
- Let $e_0, e_1, \cdots, e_{n-1}$ be $n$ distinct elements of $F$. 
• Let \( f_j = P(e_j) \).

Then the original codeword corresponding to \( d_0d_1 \cdots d_{k-1} \) is

\[
f_0f_1 \cdots f_{n-1}
\]

Input to the decoding algorithm for a chunk assuming that at least \( t \) of the \( f_j \)s should remain unchanged is

\[
\hat{f}_0\hat{f}_1 \cdots \hat{f}_{n-1}
\]

Let \( Q(x,y) \) be a polynomial such that \( Q(e_j, \hat{f}_j) = 0 \) for \( j = 0 \) to \( n - 1 \). Also let \( D_x \) and \( D_y \) be the degrees of \( x \) and \( y \) respectively in \( Q \).

Then we can write \( Q(x,y) \) as

\[
Q(x,y) = \sum_{i=0}^{D_x} \sum_{j=0}^{D_y} \alpha_{ij} x^i y^j
\]

In the above equation, there are \((1+D_x)(1+D_y)\) different \( \alpha_{ij} \)s and \( n \) equations \( Q(e_j, \hat{f}_j) = 0 \) for \( j = 0 \) to \( n - 1 \).

So if \((1+D_x)(1+D_y) > n\) then \( Q \) exists and can be computed easily.

Now let us consider another polynomial

\[
R(x) = Q(x, P(x))
\]

As \( \deg P \) is \( k - 1 \), so

\[
\deg R \leq D_x + (k - 1)D_y
\]

But \( R(x) \) is zero on at least \( t \) distinct values since for at least \( t \) \( e_j \)s,

\[
R(e_j) = Q(e_j, P(e_j)) = Q(e_j, f_j) = Q(e_j, \hat{f}_j) = 0
\]

Hence if \( \deg R \leq D_x + (k - 1)D_y < t \), then \( R(x) \) must be the zero polynomial or, \( R(x) = 0 \) for all \( x \). Then

\[
R(x) = Q(x, P(x)) = 0
\]

\Rightarrow \( Q(x,y) \) becomes 0 when \( y = P(x) \)

\Rightarrow \( Q(x,y) = 0 \ (mod \ y = P(x)) \)

\Rightarrow \( (y = P(x)) \ | \ Q(x,y) \)

So the algorithm is

• Factor \( Q(x,y) \) into irreducible factors

• Collect all factors of the form \( y - P'(x) \)

• Use domain knowledge to identify right \( P(x) \)
where the *domain knowledge* is knowledge about some typical pattern followed by valid video data so that we can get the correct original video data from a list of candidates. If we choose \( D_x = \sqrt{kn} \) and \( D_y = \sqrt{\frac{n}{k}} \), then

\[
(1 + D_x)(1 + D_y) > \sqrt{kn} \cdot \sqrt{\frac{n}{k}} = n
\]

So condition for existence of polynomial \( Q \) holds.

Now for \( R(x) \) to be a zero polynomial,

\[
D_x + (k - 1)D_y < t \Rightarrow t > \sqrt{kn} + (k - 1)\sqrt{\frac{n}{k}} \geq 2\sqrt{kn}
\]

Hence at least \( 2\sqrt{kn} \) data should remain unchanged, or in other words, the algorithm can correct upto \( n - 2\sqrt{kn} \) errors.

This algorithm takes more than real time but was improved later. Further, in a true sense, it is not a decoding algorithm as it does not produce a single decoded output but a list of candidate outputs. Then we have to extract the correct output applying knowledge about video data.

### 3 Divide and Conquer Technique

Divide and Conquer is the first tool for designing efficient algorithms in Number Theory and Algebra that we will study. As this is a well known tool so we will study only some applications of this technique.

#### 3.1 Matrix Multiplication

**Problem** Given two \( n \times n \) matrices \( A \) and \( B \), compute \( A \times B \).

Time complexity of standard algorithm of matrix multiplication is \( O(n^3) \). But by using divide and conquer technique the time complexity can be reduced. For simplicity let us assume \( n = 2^m \) (else blow up the size filling rest of the entries with zeros).

Let

\[
A = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \quad B = \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix}
\]

Then

\[
A \times B = \begin{bmatrix} A_{11}B_{11} + A_{12}B_{21} & A_{11}B_{12} + A_{12}B_{22} \\ A_{21}B_{11} + A_{22}B_{21} & A_{21}B_{12} + A_{22}B_{22} \end{bmatrix}
\]

where each of \( A_{ij} \) s and \( B_{ij} \) s are \( 2^{m-1} \times 2^{m-1} \) matrices.

The most simple way would be to compute each of the individual products of \( A_{ij} \) and \( B_{kl} \) matrices and then performing the sum. If we denote time complexity of multiplication of two \( n \times n \) matrices as \( T(n) \) then the recursive formula of \( T(n) \) will be
\[ T(n) = 8T\left(\frac{n}{2}\right) + O(n^2) \]

where the \( O(n^2) \) term comes due to addition of \( \frac{n}{2} \times \frac{n}{2} \) matrices.

**Claim 3.1** All the terms in \( A \times B \) can be computed using only \( 7 \frac{n}{2} \times \frac{n}{2} \) matrix multiplications [Strassen’s Algorithm].

**Exercise 3.1** Prove Claim 3.1.

Then the improved recursive relation for time complexity of matrix multiplication of two \( n \times n \) matrices will be

\[ T(n) = 7T\left(\frac{n}{2}\right) + O(n^2) \]

Solving this recursive relation we get

\[ T(n) = O(n^{\log_2 7}) = O(n^{2.71}) \]

Time complexity of Strassen’s algorithm was still improved further by taking \( n \) as powers of larger integers than 2. The best known algorithm for matrix multiplication has time complexity as \( O(n^{2.37}) \) though it is strongly believed by the community that the best possible time complexity is \( \theta(n^2) \).

### 3.2 Extension of Matrix Multiplication

Advantage of better time complexity of Matrix Multiplication using Divide and Conquer can be extended to other problems like finding inverse of a matrix, finding the value of determinant and solving a system of linear equations. Here we give one relevant example of reduction of Matrix Inversion to Matrix Multiplication problem.

**Problem** Given an \( n \times n \) matrix \( A \), compute the matrix \( A^{-1} \).

Let

\[
A = \begin{bmatrix}
A_{11} & A_{12} \\
A_{21} & A_{22}
\end{bmatrix}
\]

Let

\[
A^{-1} = \begin{bmatrix}
B_{11} & B_{12} \\
B_{21} & B_{22}
\end{bmatrix}
\]

Then

\[
A \times A^{-1} = \begin{bmatrix}
A_{11}B_{11} + A_{12}B_{21} & A_{11}B_{12} + A_{12}B_{22} \\
A_{21}B_{11} + A_{22}B_{21} & A_{21}B_{12} + A_{22}B_{22}
\end{bmatrix} = I = \begin{bmatrix}
1 & 0 \\
0 & 1
\end{bmatrix}
\]

Equating corresponding terms we need to solve four equations to get the values of \( B_{ij} \) matrices.

**Exercise 3.2** Fill in the details to complete the above reduction.