CS1570 Lecture Notes

Claire Mathieu

Fall 2026

1 September 24: Huffman Coding Correctness & Polynomial Multiplication via FFT

1.1 Part 1: Proof of Correctness of Huffman Coding

We assume all character frequencies are distinct to simplify analysis. Our goal is to formally prove that the bottom-up greedy strategy of Huffman coding yields an optimal prefix-free code.

1.1.1 Foundational Facts on Code Trees

Fact 1. *In any optimal code, given two leaves l and l' where l is strictly deeper than l', the letter associated with l must have a frequency strictly less than the letter associated with l'.*

$$\textit{If } \textit{depth}(l) > \textit{depth}(l'), \textit{ then } \textit{freq}(x) < \textit{freq}(l') \textit{ where } x \textit{ is the letter at } l.$$

Fact 2. *Given any prefix-free code and two leaves l and l' located at the exact same depth, switching the letters associated with l and l' produces a new prefix-free code with the exact same cost.*

Proof of Fact 2. Let enc be the initial encoding tree, and let enc$'$ be the tree after swapping the positions of the letters x and y at leaves l and l'.

1. enc$'$ is still prefix-free since no leaf positions were added or modified; only the labels were swapped.
2. The total cost of the initial code is:

$$\text{cost}(\text{enc}) = \text{freq}(x) \cdot \text{depth}(l) + \text{freq}(y) \cdot \text{depth}(l') + \sum_{z \neq x,y} \text{freq}(z) \cdot \text{depth}(z)$$

   The cost of the swapped code is:

$$\text{cost}(\text{enc}') = \text{freq}(x) \cdot \text{depth}(l') + \text{freq}(y) \cdot \text{depth}(l) + \sum_{z \neq x,y} \text{freq}(z) \cdot \text{depth}(z)$$

   Since l and l' are at the same depth ($\text{depth}(l) = \text{depth}(l')$), substituting this equality yields:

$$\text{cost}(\text{enc}) = \text{cost}(\text{enc}')$$

□

Fact 3. *There exists an optimal prefix-free code where the two least frequent letters are siblings at the deepest level of the tree.*

Proof. This follows directly by applying Fact 1 and Fact 2 to pull the two elements with the lowest frequencies down to the maximum depth of an optimal tree and placing them side by side as siblings. □

1.1.2 The Huffman Coding Algorithm

Last class it was kind of informal, I ran out of time, skipped the base case. Let me try to do it better this time. **Input:** A set of letters S with associated frequencies.

1. **Base Case:** If $|S| = 2$, create a root node with two children assigned to the two letters. This yields a valid prefix-free encoding.
2. **Recursive Step:** If $|S| > 2$:
   - Let x and y denote the two least frequent letters of S.
   - Create a new virtual letter z with a combined frequency: $\text{freq}(z) = \text{freq}(x) + \text{freq}(y)$.
   - Define a reduced alphabet set: $S' \leftarrow S \setminus \{x, y\} \cup \{z\}$.
   - Recursively solve Huffman coding for S' to deduce an encoding mapping $\text{enc}_{S'}$.
   - Construct the final encoding enc_S for S from $\text{enc}_{S'}$:

$$\begin{aligned} & \text{enc}_S(x) \leftarrow \text{enc}_{S'}(z) \cdot 0 \\ & \text{enc}_S(y) \leftarrow \text{enc}_{S'}(z) \cdot 1 \\ \forall u \in S \setminus \{x, y\}, \quad & \text{enc}_S(u) \leftarrow \text{enc}_{S'}(u) \end{aligned}$$

Lemma 1. *The algorithm detailed above produces a valid prefix-free encoding.*

Proof. By induction on $|S|$. The base case $|S| = 2$ clearly forms a valid tree. Assuming it holds for $|S| - 1$, expanding the leaf z into two children preserves the prefix-free condition. You can spell it out with one or two additional sentences to make the proof fully detailed. □

1.1.3 Theorem: Optimality Proof

Theorem 1. *Huffman's algorithm outputs an optimal prefix-free code.*

Proof. We proceed by induction on the size of the alphabet $|S|$.

- **Base Case:** For $|S| = 2$, the algorithm outputs the only possible full binary tree.
- **Inductive Step:** Assume the algorithm is optimal for size $n - 1$. We consider an alphabet of size $|S| = n$.

  We need to upper bound the cost of the tree produced by our Huffman algorithm ($\text{enc}_S^{\text{Huff}}$). Looking at the algorithm, we see that it is is derived by expanding the solution found recursively for S' ($\text{enc}_{S'}^{\text{Huff}}$):

$$\text{cost}(\text{enc}_S^{\text{Huff}}) = \text{cost}(\text{enc}_{S'}^{\text{Huff}}) + \text{freq}(x) + \text{freq}(y)$$

  Let enc_S^* be an actual optimal prefix-free code tree for S such that that the two least frequent letters x and y are siblings in enc_S^* (by Fact 3, it exists.) If in that optimal code-tree we contract the sibling leaves x and y into a single parent node z, we obtain a valid prefix-free code tree for the smaller alphabet S', which we denote as $\text{enc}'_{S'}$.

  The relationship between the exact cost of the expanded tree and the contracted tree is given by:

$$\text{cost}(\text{enc}_S^*) = \text{cost}(\text{enc}'_{S'}) + \text{freq}(x) + \text{freq}(y)$$

  By our inductive hypothesis applied to S', the Huffman algorithm is optimal for S', so:

$$\text{cost}(\text{enc}_{S'}^{\text{Huff}}) \leq \text{cost}(\text{enc}'_{S'})$$

  Thus:

$$\text{cost}(\text{enc}_S^{\text{Huff}}) \leq \text{cost}(\text{enc}_S^*) = \text{OPT}$$

  Thus, Huffman coding on S is optimal.

  (That is not needed for the proof, but remark that since the cost of our Huffman tree cannot be strictly less than the absolute optimal cost, it must be exactly equal to it.)

□

1.2 Part 2: Polynomial Operations and Divide-and-Conquer Multiplication

Let a polynomial $A(x)$ of degree bound n be defined by its coefficients:

$$A(x) = a_0 + a_1 x + a_2 x^2 + \cdots + a_{n-1} x^{n-1}$$

1.2.1 Standard Polynomial Operations and Complexities

Operation	Mathematical Formulation	Naive Time	Target Optimized Time
Evaluation	$(A, x_0) \mapsto A(x_0) = \sum a_i x_0^i$	$O(n^2)$ or $O(n)$ (Horner's)	$O(n)$
Addition	$(A, B) \mapsto C = A + B \implies c_k = a_k + b_k$	$O(n)$	$O(n)$
Multiplication	$(A, B) \mapsto C = A \cdot B \implies c_k = \sum_i a_i b_{k-i}$	$O(n^2)$	$O(n \log n)$

Note on Multiplication: Standard convolution takes $O(n^2)$. Advanced Karatsuba/Strassen-type approaches can reduce this to $O(n^{\log_2 3}) \approx O(n^{1.58})$. Our goal today is to get the polynomial multiplication time down to $O(n \log n)$ using a divide-and-conquer framework.

1.2.2 Alternative Polynomial Representations

One key idea is to shift between two distinct representations of a polynomial:

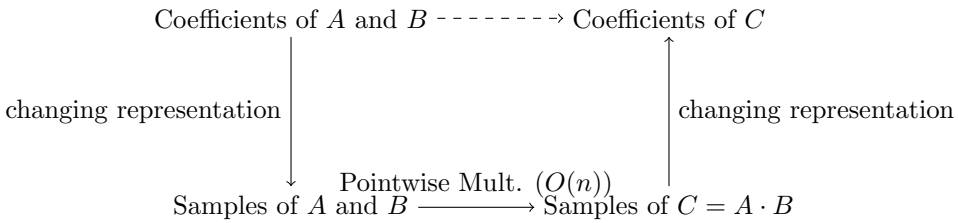
1. **Coefficient Representation:** Given by the ordered vector of coefficients: $(a_0, a_1, \ldots, a_{n-1})$.
2. **Point-Value (Sample) Representation:** A polynomial of degree less than or equal to $n-1$ is uniquely determined by its evaluation at n distinct sample points:

$$\{(x_0, y_0), (x_1, y_1), \ldots, (x_{n-1}, y_{n-1})\} \quad \text{where } y_i = A(x_i)$$

(Note: the AI which I used for the first draft of the lecture notes only wrote " polynomial of degree $n-1$, not " polynomial of degree at most $n-1$", which is what we need since we will use it with $2n-1$ points for a polynomial of degree at most $n-1$; thus what it wrote was just slightly off.)

Why use samples? While multiplying matrices in the coefficients representation requires doing convolutions and naïvely takes $O(n^2)$ time, multiplying two polynomials that are already in point-value form takes linear time $O(n)$ because we can compute each point-wise product with a single multiplication: $z_i = A(x_i) \cdot B(x_i)$.

Therefore, the global fast multiplication framework operates as follows:



1.2.3 The Divide-and-Conquer Strategy (Even/Odd Splitting)

To change the representation rapidly with divide-and-conquer, in the "divide" step we split $A(x)$ into its even-indexed and odd-indexed coefficients:

$$A_{\text{even}}(x) = a_0 + a_2 x + a_4 x^2 + a_6 x^3 + \ldots$$
$$A_{\text{odd}}(x) = a_1 + a_3 x + a_5 x^2 + a_7 x^3 + \ldots$$

Imagine that we have recursively computed the point-value representations of A_{even} and of A_{odd}, *at the same points.* In the conquer step, the value of polynomial A can then be evaluated as:

$$A(x_i) = A_{\text{even}}(x_i^2) + x_i \cdot A_{\text{odd}}(x_i^2).$$

Examining this, we see that in order to evaluate A at $\{x_0, x_1, \ldots, x_{n-1}\}$, we need to have evaluated A_{even} and A_{odd} at $\{x_0^2, x_1^2, \ldots, x_{n-1}^2\}$.

This creates a problem for the induction hypothesis. We have reduced computing the values of a polynomial of degree $n-1$ at n points, to computing the values of polynomials of degree $n/2-1$ but still at n points! For the recursive structure to work out, we would need the evaluation to be on polynomials whose degree is halved, but on a set of points whose cardinality if also halved!

But note that until now we have not specified the values of $x_0, x_1, \ldots, x_{n-1}$. We are free to take anything we want. To evaluate a polynomial of degree bound n at $2n$ sample points $\{x_0, x_1, \ldots, x_{2n-1}\}$ efficiently via recurrence, we need the set of squared points $\{x_0^2, x_1^2, \ldots, x_{2n-1}^2\}$ to collapse to half the size (n unique values).

1.2.4 Choosing the Sample Points: The Magic of Roots of Unity

- **Small Case Experiment:** Consider choosing 2 evaluation points. If we choose $\{-1, 1\}$ then the set of squares has size 1: $\{1\}$. Good start.

  Consider choosing 4 evaluation points. If we choose symmetric positive/negative pairs like $\{x_0, x_1, x_2, x_3\} = \{1, -1, i, -i\}$, computing their squares yields:

  $$\{1^2, (-1)^2, i^2, (-i)^2\} = \{1, 1, -1, -1\} = \{1, -1\}$$

  The set of squared values has a cardinality of exactly 2 and is the one on which we can recurse. This is the recursive structure we want.

Extending this properties to arbitrary powers of 2 leads us to the complex roots of unity. The n-th roots of unity are the n complex numbers satisfying $\omega^n = 1$, generating a collapsing structure under squaring that forms the mathematical backbone of the Fast Fourier Transform (FFT).

1.2.5 Matrix Formulation: The Vandermonde Matrix

The full evaluation operation of a polynomial across these specific sample points can be expressed as a linear system using a Vandermonde matrix V:

$$\begin{bmatrix} 1 & x_0 & x_0^2 & \ldots & x_0^{n-1} \\ 1 & x_1 & x_1^2 & \ldots & x_1^{n-1} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & x_{n-1} & x_{n-1}^2 & \ldots & x_{n-1}^{n-1} \end{bmatrix} \begin{bmatrix} a_0 \\ a_1 \\ \vdots \\ a_{n-1} \end{bmatrix} = \begin{bmatrix} y_0 \\ y_1 \\ \vdots \\ y_{n-1} \end{bmatrix}$$

By selecting x_i as the n-th roots of unity, this can be evaluated in $O(n \log n)$ time using the even/odd divide-and-conquer strategy. The inverse transformation, to get from the sample values to the coefficients (Interpolation), is achieved by multiplying the sample vector by the matrix inverse V^{-1}. It tursn out that it is well known that V^{-1} is $1/n$ times the complex conjugate of V, so it also lends itself to a similar recursive computation in the same time complexity.