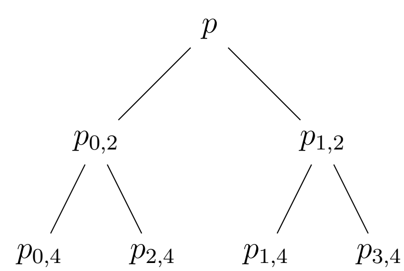
1 Polynomial Multiplication & FFT

Given a polynomial p with degree d, as an array of coefficients $\mathbf{a}$, take $l = \lceil \log_2(d+1) \rceil$ and $N = 2^l$. Pad the end of $\mathbf{a}$ with 0's until it has length N. The FFT algorithm will return an array $\mathbf{A}$ such that $\mathbf{A}[k] = p(\exp(-2\pi ik/N))$. We'll use the shorthand $w_k = \exp(-2\pi ik/N)$ and $R_N = \{\exp(-2\pi ik/N) : 0 \le k \le N-1\}$.

Define for $j \mid n$,

$$p_{i,j}(x) = \sum_{n=0}^{d/j-1} a_{i+jn} x^n,$$

that is, we keep the coefficients congruent to i modulo j. The recursive splitting of p, corresponds to the tree below.



At level i in the tree, we have the polynomials,

$$p_{0,2^i}, p_{2^{i-1},2^i}, p_{2^{i-2},2^i}, p_{2^{i-2}+2^{i-1},2^i}, \dots, p_{2^{i-1}-1,2^i}, p_{2^i-1,2^i}. \tag{1}$$

Note

$$p_{i,2^j}(x) = p_{i,2^{j+1}}(x^2) + x p_{i+2^j,2^{j+1}}(x^2) \tag{2}$$

since coefficients which are congruent mod 2^j but not mod 2^{j+1} must differ by a multiple of 2^j.

Let $m = 2^j$. Then $\widehat{\omega_k} = \omega_k^{N/m} = \exp(-2\pi i k/m)$ has the property:

$$\begin{aligned} p_{i,m}\left(\widehat{\omega_{k+\frac{m}{2}}}\right) &= p_{i,2m}(\widehat{\omega_{k+\frac{m}{2}}}^2) + \widehat{\omega_{k+\frac{m}{2}}} p_{i+m,2m}(\widehat{\omega_{k+\frac{m}{2}}}^2) \\ &= p_{i,2m}(\widehat{\omega_k}^2) + \widehat{\omega_k}(\widehat{w_{\frac{m}{2}}}) p_{i+m,2m}(\widehat{\omega_k}^2) \\ &= p_{i,2m}(\widehat{\omega_k}^2) - \widehat{\omega_k} p_{i+m,2m}(\widehat{\omega_k}^2) \end{aligned} \tag{3}$$

Assuming the algorithm evaluates (1) on $R_{N/2^{j+1}}$. By equations (2) and (3), we can combine these to evaluate the polynomials in level j at $R_{N/2^j}$.

To form an iterative algorithm we must compute bottom-up (instead of top-down). Our array $\mathbf{A}$ should be initialized so that $\mathbf{A}[k]$ is the coefficient of p present at position k in level l of the tree. This can be achieved by setting

$$\mathbf{A}[\mathrm{rev}(k)] = a_k$$

where $\mathrm{rev}(k)$, denotes the bit reversal of k.

Note $R_{N/2^j}$ consists of $\omega_k^{2^j} = \exp(-2\pi i k/2^{l-j})$. Since we're starting at level $l-1$ and working up, we can take $\hat{\omega} = \exp(-2\pi i/2^s) = \omega_1^{N/2^s}$ in iteration s (the iteration where we compute the polynomials at level $l-s$ on $R_{N/2^s}$).

```
for k = 0 to N − 1:
    A[rev(k)] := a_k

for s = 1 to l:
    m ← 2^s
    ω̂ ← exp(−2πi/m)
    for k = 0 to N − 1 by m:
        ω ← 1
        for j = 0 to m/2 − 1:
            t ← ωA[k + j + m/2]
            u ← A[k + j]
            A[k + j] ← u + t
            A[k + j + m/2] ← u - t
            ω ← ωω̂
return A
```

The innermost j-loop allows us to exploit the property in (3) by computing $\mathbf{A}$, m elements at a time. The j and k loops together compute $\mathbf{A}$.

Invariant: After the `k`-loop finishes executing, we have:

$$\mathbf{A}[n] = p_{\text{rev}(n),N/m}(\omega_n^{N/m}).$$

2 Polynomial Inverses

We can represent the inverse of a polynomial $f(x)$ as a power series centered at $x = 0$. Note: Such a series will exist as long as $f(0) \neq 0$.

So assume $f(x) \in \mathbb{F}[x]$ and $f(0) \neq 0$. Note that if $f_0 \in \mathbb{F}[x]$ has the property that

$$\frac{1}{f(x)} - f_0(x)$$

is a multiple of $x^{\lceil t/2 \rceil}$, then

$$\begin{aligned} \frac{1}{f(x)} - (f_0(x) - (f(x)f_0(x) - 1)f_0(x)) &= \frac{1}{f(x)} - 2f_0(x) + f(x)f_0(x)^2 \\ &= f(x)\left(\frac{1}{f(x)} - f_0(x)\right)^2 \end{aligned} \tag{4}$$

is a multiple of x^t. That is, $f^{-1}(x)$ and $f_0(x)$ agree on their first t terms.

```
IterativeInverse(A, t):
    m ← 1
    A₀ ← 1/a₀
    while (m < t):
        m << 1
        A₀ ← 2A₀ − AA₀²
```

3 Division

Suppose we are given two polynomials $f(x)$ and $g(x)$ of degrees m, n, respectively. We want to find $q(x), r(x)$ such that $f(x) = q(x)g(x) + r(x)$ and $\deg r(x) < n$.

It would be useful to invert g, but its constant term may be 0. However, we know $g_{n-1} \neq 0$ as g has degree n, so it's useful to consider the reverse polynomial: $g^R(x) = x^n g(1/x)$, which we know is invertible. Note

$$\begin{aligned} f^R(x) = x^m f(1/x) &= (x^{m-n} q(1/x))(x^n g(1/x)) + x^{m-n+1}(x^{n-1} r(1/x)) \\ &= q^R(x) g^R(x) + r^R(x) \end{aligned} \tag{5}$$

Therefore, $q^R(x) = f^R(x)[g^R(x)]^{-1} \bmod x^{m-n+1}$, so it suffices to compute $[g^R(x)]^{-1}$ to $n - m + 1$ terms. Thus, we have the following algorithm for polynomial division using only the `Inverse` and `PolyMult` subroutines.

- Reverse the coefficient arrays for f, g.
- Compute $\texttt{Inverse}(g^R, m - n + 1)$.
- Compute $q^R(x) = f^R(x)[g^R(x)]^{-1} \bmod x^{m-n+1}$.
- Compute q by reversing q^R.
- Compute r by $r = f - qg$.

4 Point Evaluation

We want to evaluate the polynomial $f(x)$ of degree n and we want to compute $f(1), \ldots, f(x_n)$.

Define

$$\begin{aligned} d_1 &= \prod_{i=1}^{\lceil n/2 \rceil - 1} (x - x_i) \\ d_2 &= \prod_{i=\lceil n/2 \rceil}^{n} (x - x_i) \end{aligned}$$

Using `PolyDiv`, we can write $A = q_1 d_1 + r_1 = q_2 d_2 + r_2$. Note

$$\begin{aligned} A(x_i) &= r_1(x_i), \quad \text{for } 1 \leq i \leq \lceil n/2 \rceil - 1 \\ A(x_i) &= r_2(x_i), \quad \text{for } \lceil n/2 \rceil \leq i \leq n. \end{aligned} \tag{6}$$

This divides the problem into evaluating each r_j on the appropriate set of inputs. This algorithm would be $O(n \log^2 n)$ if we ignore the cost of computing d_1, d_2. Due to this expense however, Horner's method is preferred.

5 Interpolation

Given point-value pairs, $\{(x_i, y_i)\}_{i=1}^{n+1}$ where the x_i are pairwise distinct, we want to find the unique polynomial $p(x)$ of degree n, through these points.

Define

$$P_i(x) = \prod_{j \neq i} \frac{x - x_j}{x_i - x_j}.$$

Note $P_i(x_j) = 0$ for $j \neq i$ and $P_i(x_i) = 1$. Thus,

$$p(x) = \sum_{i=1}^{n-1} y_i P_i(x).$$

For efficiency sake, `PolyInterpolate` will simply return the array of coefficients,

$$\mathbf{L}[i] = y_i \prod_{j \neq i} \frac{1}{x_i - x_j}.$$

Alternatively, we could instantiate a `PolyValGenerator` object P with the point value pairs. This would

- Compute/store the Lagrange coefficients, $\mathbf{L}$, as above.
- Store the x_i's and y_i's.

To evaluate P at $\hat{x}$, we compute

$$\hat{X} = \prod_{i=1}^{n+1} (\hat{x} - x_i)$$

and then return

$$\hat{X} \sum_{i=1}^{n+1} A_i/(\hat{x} - x_i).$$

Note: If $\hat{x} = x_j$, then we should instead return y_j.

6 Differentiation

We can compute the derivative of a polynomial from its coefficient array via

$$A[k] \leftarrow (k+1)A[k+1].$$

This enables us to perform Newton's method for root searching:

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}.$$