Fourier Transforms

Idea. Given the coefficients $p_0, p_1, \ldots, p_{n-1}$ and $q_0, q_1, \ldots, q_{n-1}$ of two polynomials $p + q$.

Step 1. (Evaluate) Compute ordered pairs of point-value representation
\[
\{(x_i, p(x_i)) \mid 0 \leq i \leq 2n-1\} \quad \text{and} \quad \{(x_i, q(x_i)) \mid 0 \leq i \leq 2n-1\}
\]

Step 2. (Pointwise Multiply) Compute
\[
y_i = p(x_i)q(x_i) \text{ for } 0 \leq i \leq 2n-1.
\]

Step 3. (Interpolate) Compute
\[
\alpha_0, \alpha_1, \ldots, \alpha_{2n-1} \text{ s.t.} \quad y_i = r(x_i) \text{ for } 0 \leq i \leq 2n-1
\]

The key to making this scheme efficient is to find suitable $x_i$'s so that interpolation and evaluation becomes efficiently executable.
Complex Numbers.

\[ i = \sqrt{-1} \] is the imaginary unit
\[ a + ib \] is a complex number, where \( a, b \in \mathbb{R} \)

Exponential representation of \( a + ib \):
\[ |a|e^{i\theta} \] where \( |a| = \sqrt{a^2 + b^2} \)
\[ \theta = \cos^{-1} \frac{a}{|a|^2 + b^2} \]
\[ = \tan^{-1} \frac{b}{a} \]

Roots of unity:
\[ w^n = 1, \quad w \] is the \( n \)th root of \( 1 \).
\[ w_n = e^{\frac{i2\pi}{n}} \] is the principle \( n \)th root of \( 1 \).
\[ w_0, w_1 = w_n, w_2, w_3, \ldots, w_{n-1} \]
are all \( n \) roots of \( 1 \).
Properties of Roots of Unity:

Roots of unity will be used as values $x_i$ in the FFT algorithm. We therefore need some of their properties.

1. $\omega_{dk}^n = \omega_k^n$ because $\omega_{dk}^n = e^{\frac{2\pi i dk}{2n}} = e^{\frac{2\pi i}{n}} = \omega_k^n$

2. $\omega_n^{n/2} = \omega_2 = -1$, for even $n$

3. $(\omega_n^k)^2 = \omega_n^{k/2}$, for even $n$

4. $\sum_{j=0}^{n-1} (\omega_n^k)^j = 0$, for $n > 1$ and $k > 1$ not divisible by $n$.

Proof (4):

$$\omega_n^k \sum_{j=0}^{n-1} (\omega_n^k)^j - \sum_{j=0}^{n-1} (\omega_n^k)^j$$

$$= \sum_{j=1}^{n-1} (\omega_n^k)^j - \sum_{j=0}^{n-1} (\omega_n^k)^j$$

$$= (\omega_n^k)^n - (\omega_n^k)^0 = 0$$

So,

$$\sum_{j=0}^{n-1} (\omega_n^k)^j = \frac{0}{\omega_n^k - 1} = 0$$

Since $\omega_n^k \neq 1$ for $k$ is not divisible by $n$. 
Discrete Fourier Transform.

A degree bound of a polynomial is an integer that exceeds its degree.

Step 1 of the general algorithm evaluates \( p(x) \) at \( n \) different values of \( x \), and we choose \( x_j = \omega_n^j \) for \( j = 0, 1, \ldots, n-1 \)

\[
p(x) = p_0 + p_1 x + p_2 x^2 + \cdots + p_{n-1} x^{n-1}
\]

- Define \((y_0, y_1, \ldots, y_{n-1})\) so that

\[
y_k = p(\omega_n^k) = \sum_{j=0}^{n-1} p_j \omega_n^{kj}
\]

- The vector \( y = (y_0, y_1, \ldots, y_{n-1}) \) is the **Discrete Fourier Transform** of \( p = (p_0, p_1, \ldots, p_{n-1}) \)

\[
y = \text{DFT}(p).
\]

- The FFT is an algorithm to compute DFT quickly.

- We will see that some algorithm can be used for interpolation \( \text{DFT}^{-1} \) which is necessary for step 3.
Implementation of Complex Numbers.

type Complex = record re, im: real end;

function Product (A, B : complex) : complex;
C.re := A.re * B.re - A.im * B.im;
C.im := A.re * B.im + A.im * B.re;
return C.

Similarly, one can write addition, subtraction and division of complex numbers.

Fast Fourier Transform (FFT).

Similar to \textit{n} time algorithm, the FFT uses divide-and-conquer, but in a different way. Assume \( n = 2^k \).

\begin{align*}
    p^{[0]}(x) &= p_0 + p_2 x + p_4 x^2 + \cdots + p_{n-2} x^{\frac{n-2}{2}} \\
    p^{[1]}(x) &= p_1 + p_3 x + p_5 x^2 + \cdots + p_{n-1} x^{\frac{n-2}{2}}
\end{align*}

So, \( p(x) = p^{[0]}(x^2) + x \cdot p^{[1]}(x^2) \).
In order to evaluate \( p(x) \) at the \( n \) \( n \)th root of unity, we just combine the evaluated values of \( p^{[0]}(x) \) and \( p^{[1]}(x) \) at these roots.

Suppose we have

\[
y^{[0]}_k = \text{DFT}(p^{[0]}), \quad y^{[1]}_k = \text{DFT}(p^{[1]}).
\]

This means that

\[
y^{[0]}_k = p^{[0]}(w^{kn}), \quad \text{for } 0 \leq k \leq \frac{n-2}{2}, \text{ and}
\]

\[
y^{[1]}_k = p^{[1]}(w^{kn}), \quad \text{for } 0 \leq k \leq \frac{n-2}{2}.
\]

We can now compute \( y = \text{DFT}(p) \) as:

For \( 0 \leq k \leq \frac{n-2}{2} \),

\[
y_k = p(w^k) = p^{[0]}(w^{2k}) + w^k p^{[1]}(w^{2k}).
\]

\[
y_{n+k} = y^{[0]}_k + w^k y^{[1]}_k.
\]

\[
y_{n+\frac{k}{2}} = p(w^{\frac{n+2k}{2}})
\]

\[
= p^{[0]}(w^{2k}) + w^{\frac{n+k}{2}} p^{[1]}(w^{2k})
\]

\[
= p^{[0]}(w^{2k}) - w^k p^{[1]}(w^{2k})
\]

\[
= y^{[0]}_k - w^k y^{[1]}_k.
\]
function FFT (p: polynomial) : point-values;
    n := n(p); if n=1 then return p;
    else
        p[0] := (p_0, p_2, ..., p_{n-2});
        p[1] := (p_1, p_3, ..., p_{n-1});
        y[0] := FFT(p[0]);
        y[1] := FFT(p[1]);
        w := 1 
        w_n = e^{i \frac{2\pi}{n}};
        for k := 0 to \frac{n-1}{2} do
            y_k := y_k[0] + w y_k[1]
            y_{n+k} := y_k[0] - w y_k[1];
        w := w \cdot w_n
        endfor
        return y
    endif

T(n) = 2T(\frac{n}{2}) + O(n)
    = O(n \log n).
The degree bound \( n \), used in function FFT must be at least twice the degree + 1 because in step 3 of the general algorithm we need that many point-value pairs to do the interpolation.

After calling \( \text{FFT}(p) \) and \( \text{FFT}(q) \) we get a point-value representation of \( r(x) = p(x) q(x) \) by multiplying the corresponding point-values:

\[
y_j = p(w_j^n) \cdot q(w_j^n)
\]

which takes only \( O(n) \) time.
Inverse DFT

The last step of the general approach, step 3, consists of interpolating $n$ point-value pairs at the $n$ $n$th roots of unity. Recall that DFT can be written as follows:

$$
\begin{bmatrix}
y_0 \\
y_1 \\
y_2 \\
\vdots \\
y_{n-1}
\end{bmatrix} =
\begin{bmatrix}
1 & 1 & 1 & \hdots & 1 \\
w_n & w_n^2 & w_n^3 & \hdots & w_n^{n-1} \\
w_n^2 & w_n^4 & w_n^6 & \hdots & w_n^{2(n-1)} \\
w_n^3 & w_n^6 & w_n^9 & \hdots & w_n^{3(n-1)} \\
w_n^{n-1} & w_n^{2(n-1)} & w_n^{3(n-1)} & \hdots & w_n^{(n-1)(n-2)}
\end{bmatrix}
\begin{bmatrix}
y_0 \\
y_1 \\
y_2 \\
\vdots \\
y_{n-1}
\end{bmatrix}
$$

because $x_i = w_n^i$ for $0 \leq i \leq n-1$. In order to compute $\text{DFT}^{-1}(y)$ we multiply with the inverse matrix, $V_n^{-1}$.

Claim

$V_n^{-1} = \frac{1}{n} \begin{bmatrix}
1 & 1 & 1 & \hdots & 1 \\
w_n & w_n^2 & w_n^3 & \hdots & w_n^{n-1} \\
w_n^2 & w_n^4 & w_n^6 & \hdots & w_n^{2(n-1)} \\
w_n^3 & w_n^6 & w_n^9 & \hdots & w_n^{3(n-1)} \\
w_n^{n-1} & w_n^{2(n-1)} & w_n^{3(n-1)} & \hdots & w_n^{(n-1)(n-2)}
\end{bmatrix}$

Proof. We need to show that $V_n^{-1}$ multiplied with $V_n$ gives the unit matrix.
\[ V_n^{-1} V_n = \frac{1}{n} \left( \sum_{j=0}^{n-1} w_n^{-jk} w_n^{jl} \right) \]

\[ = \frac{1}{n} \left( \sum_{j=0}^{n-1} w_n^{j(l-k)} \right) \]

\[ = \begin{cases} 
  0 & \text{if } l \neq k \\
  1 & \text{if } l = k 
\end{cases} \]

So, \( T = \text{DFT}^{-1}(y) \) is given by

\[ r_k = \frac{1}{n} \sum_{j=0}^{n-1} w_n^{-jk} y_j \]

Compare it: \( y_k = \sum_{j=0}^{n-1} w_n^{jk} p_j \)

The similarity between the two formulas allows us to reuse FFT algorithm for performing the interpolation.
We have:

\[ y(x) = y_0 + y_1 x + \cdots + y_{n-1} x^{n-1} \]

\[ y^{[0]}(x) = y_0 + y_2 x + y_4 x^2 + \cdots + y_{n-2} x^{\frac{n-2}{2}} \]

\[ y^{[1]}(x) = y_1 + y_3 x + y_5 x^2 + \cdots + y_{n-1} x^{\frac{n-1}{2}} \]

So, \( y(x) = y^{[0]}(x) + x \cdot y^{[1]}(x^2) \)

So, we compute recursively:

\[ r^{[0]} = \text{DFT}^{-1}(y^{[0]}) \quad \text{and} \quad r^{[1]} = \text{DFT}^{-1}(y^{[1]}) \]

This means:

\[ n r^{[0]} = y^{[0]}(w_n^{-k}) \quad \text{for} \quad 0 \leq k \leq \frac{n-2}{2}, \quad \text{and} \]

\[ n r^{[1]} = y^{[1]}(w_n^{-k}) \quad \text{for} \quad 0 \leq k \leq \frac{n-2}{2}. \]

So, we can compute: \( Y = \text{DFT}^{-1}(y) \):

For \( 0 \leq k \leq \frac{n-2}{2} : \)

\[ n r_k = y(w_n^{-k}) = y^{[0]}(w_n^{-2k}) + w_n^{-k} y^{[1]}(w_n^{-2k}) = n r^{[0]}_k + w_n^{-k} n r^{[1]}_k. \]

\[ n r_{\frac{n}{2} + k} = y(w_n^{-\left(\frac{n}{2}+k\right)}) = y^{[0]}(w_n^{-2k}) + w_n^{-\frac{n}{2}+k} y^{[1]}(w_n^{-2k}) = n r^{[0]}_k - w_n^{-k} n r^{[1]}_k. \]
From the previous page,

\[ n r_k = n r_k^{[0]} + w_n^{-k} n r_k^{[1]} \]
\[ r_k = r_k^{[0]} + w_n^{-k} r_k^{[1]} \]
\[ n r_{\frac{n+k}{2}} = n r_k^{[0]} - w_n^{-k} n r_k^{[1]} \]
\[ r_{\frac{n+k}{2}} = r_k^{[0]} - w_n^{-k} r_k^{[1]} \].

So, we can compute \( r \) by \( r := \frac{1}{n} \text{FFT}^{-1}(y) \)
where \( \text{FFT}^{-1} \) differs from \( \text{FFT} \) only at 2 places:

- in line (1): \( w_n = e^{i \frac{2\pi}{n}} \rightarrow w_n^{-1} = e^{-i \frac{2\pi}{n}} \)
- in line (2): \( w := w \cdot w_n \rightarrow w := w \cdot w_n^{-1} \).

We conclude that both \( \text{DFT} \) and \( \text{DFT}^{-1} \) can be computed in \( O(n \log n) \) time.

Summary: Let \( p(x) \) and \( q(x) \) be two polynomials with degree bound \( n = 2^k \). Then,

\[ r = \text{DFT}^{-1}(\text{DFT}_{2n}^n(p) \cdot \text{DFT}_{2n}^n(q)) \]

is such that \( r(x) = p(x)q(x) \) and it can be computed in \( O(n \log n) \) time.